ALTERA

Applications Handbook

Applications Handbook

April 1992

Altera reserves the right to make changes, without notice, in the devices or the device specifications identified in this document. Altera advises its customers to obtain the latest version of device specifications to verify, before placing orders, that the information being relied upon by the customer is current. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty. Testing and other quality control techniques are used to the extent Altera deems such testing necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed. In the absence of written agreement to the contrary, Altera assumes no liability for Altera applications assistance, customer's product design, or infringement of patents or copyrights of third parties by or arising from use of semiconductor devices described herein. Nor does Altera warrant or represent any patent right, copyright, or other intellectual property right of Altera covering or relating to any combination, machine, or process in which such semiconductor devices might be or are used.

Altera's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Altera Corporation. As used herein:

1. Life support devices or systems are devices or systems that (a) are intended for surgical implant into the body or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.

2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

2610 Orchard Parkway
San Jose, CA 95134-2020
(408) 894-7000
Applications Hotline:
(800) 800-EPLD
Marketing Information:
(408) 894-7000

About this Applications Handbook

April 1992

This applications handbook provides application notes and briefs for engineers and engineering managers who seek practical ways to reduce design costs, improve design quality, and shorten design cycles. Design guidelines and tips on how to use Altera's Classic, MAX 5000, MAX 7000, STG, and SAM EPLDs, development systems, and development software are also provided. For complete information on device and software characteristics and specifications, see the Altera 1992 ***Data Book***. For information about Micro Channel EPLDs and the associated development system and software, refer to the Altera ***Micro Channel Adapter Handbook***.

For immediate assistance on technical questions, call:

Altera Applications Hotline

(800) 800-EPLD

For information on product availability, pricing, and order status, please contact your Altera Representative or Distributor. Phone numbers and addresses of Altera Sales Offices, Representatives, and Distributors are listed in this applications handbook.

If you have questions that cannot be answered by your Sales Representative or Distributor, call:

Altera Marketing

(408) 894-7000

or contact Altera by:

FAX

(408) 248-6924

Contents

April 1992

Section 2 Application Briefs

Replacing 20-Pin PAL & GAL Devices with EP330 Classic EPLDs

April 1992, ver. 4

Application Note 2

Introduction

The Altera EP330 Classic EPLD is a drop-in replacement for all common 20-pin PAL and GAL devices. The EP330 EPLD offers major advantages over its PAL/GAL counterparts, including very high speed (t_{PD} = 12 ns) and the low power consumption available with CMOS technology. These benefits, combined with a highly flexible architecture, make the EP330 EPLD the logical alternative to common 20-pin programmable logic devices. This application note describes the EP330 architecture, its advantages over common PAL/GAL devices, and the Altera design tools that support the EP330 EPLD.

EP330 Architecture

EP330 EPLDs, like PAL and GAL devices, implement sum-of-products logic with a programmable-AND/fixed-OR logic array. EP330 EPLDs allow up to 18 inputs and 8 outputs. Each of the 8 macrocells in the device contains one D flipflop and can implement 8 product terms of combinatorial logic. Each product term represents a 36-input AND gate that is fed by the true and complement signals of the 10 dedicated input pins and 8 I/O pins. An additional product term controls the Output Enable. Each I/O pin can be independently configured for input, output, or bidirectional operation. Pin 1 can be used to directly clock all registers, or used as an additional input to the AND array if no clocking is required. The flipflops are positive-edge-triggered, i.e., data is registered on the rising edge of the Clock signal. During device power-up, all registers are automatically reset to zero. For more information, see the *EP330 EPLD: High-Performance 8-Macrocell Device Data Sheet* in the Altera 1992 ***Data Book***.

EP330 Advantages

The EP330 EPLD has significant advantages over PAL and GAL devices, including superior I/O structure, macrocell resources, and Output Enable options.

I/O Structure & Device Resources

While PAL devices have fixed I/O architectures, EP330 macrocells are based on a flexible, user-configurable I/O structure. Macrocells can be configured for combinatorial or registered operation in active-high or active-low modes. Figure 1 shows the possible output modes, which can be selected on a macrocell-by-macrocell basis.

Figure 1. EP330 I/O Options

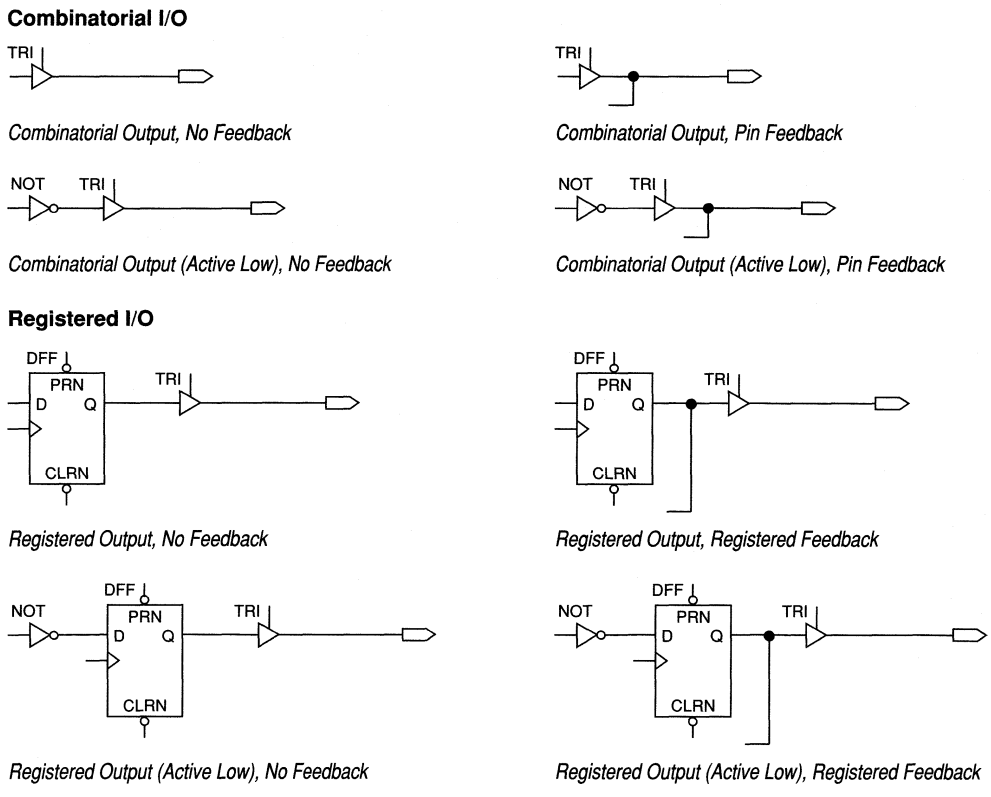


Table 1 shows the EP330 I/O configurations required to replace commonly used 20-pin PALs and GALs. Table 2 compares the Altera EP330 EPLD with competitive PAL/GAL devices.

Table 1. EP330 Configuration for 20-Pin PAL/GAL Replacement

PAL Part Number	EP330 Pin Numbers	EP330 Macrocell Numbers	I/O Configuration Mode	Output/Polarity	Feedback
10H8	12 to 19	1 to 8	Combinatorial	Comb./High	None
10L8	12 to 19	1 to 8	Combinatorial	Comb./Low	None
12H6	12	8	Combinatorial	None	Pin
	13 to 18	2 to 7	Combinatorial	Comb./High	None
	19	1	Combinatorial	None	Pin
12L6	12	8	Combinatorial	None	Pin
	13 to 18	2 to 7	Combinatorial	Comb./Low	None
	19	1	Combinatorial	None	Pin
14H4	12 to 13	7 to 8	Combinatorial	None	Pin
	14 to 17	3 to 6	Combinatorial	Comb./High	None
	18 to 19	1 to 2	Combinatorial	None	Pin
14L4	12 to 13	7 to 8	Combinatorial	None	Pin
	14 to 17	3 to 6	Combinatorial	Comb./Low	None
	18 to 19	1 to 2	Combinatorial	None	Pin
16C1	12 to 14	6 to 8	Combinatorial	None	Pin
	15	5	Combinatorial	Comb./Low	None
	16	4	Combinatorial	Comb./High	None
	17 to 19	1 to 3	Combinatorial	None	Pin
16H2	12 to 14	6 to 8	Combinatorial	None	Pin
	15 to 16	4 to 5	Combinatorial	Comb./High	None
	17 to 19	1 to 3	Combinatorial	None	Pin
16L2	12 to 14	6 to 8	Combinatorial	None	Pin
	15 to 16	4 to 5	Combinatorial	Comb./Low	None
	17 to 19	1 to 3	Combinatorial	None	Pin
16L8	12	8	Combinatorial	Comb./Low/Z	None
	13 to 18	2 to 7	Combinatorial	Comb./Low/Z	Comb.
	19	1	Combinatorial	Comb./Low/Z	None
16R4	12 to 13	7 to 8	Combinatorial	Comb./Low/Z	Comb.
	14 to 17	3 to 6	Registered	Reg./Low/Z	Reg.
	18 to 19	1 to 2	Combinatorial	Comb./Low/Z	Comb.
16R6	12	8	Combinatorial	Comb./Low/Z	Comb.
	13 to 18	2 to 7	Registered	Reg./Low/Z	Reg.
	19	1	Combinatorial	Comb./Low/Z	Comb.
16R8	12 to 19	1 to 8	Registered	Reg./Low/Z	Reg.
16P8	12	8	Combinatorial	Comb./Option/Z	None
	13 to 18	2 to 7	Combinatorial	Comb./Option/Z	Comb.
	19	1	Combinatorial	Comb./Option/Z	None
16RP4	12 to 13	7 to 8	Combinatorial	Comb./Option/Z	Comb.
	14 to 17	3 to 6	Registered	Reg./Option/Z	Reg.
	18 to 19	1 to 2	Combinatorial	Comb./Option/Z	Comb.
16RP6	12	8	Combinatorial	Comb./Option/Z	Comb.
	13 to 18	2 to 7	Registered	Reg./Option/Z	Reg.
	19	1	Combinatorial	Comb./Option/Z	Comb.
16RP8	12 to 19	1 to 8	Registered	Reg./Option/Z	Reg.
16V8	12 to 19	1 to 8	Combinatorial	Comb./Reg.	Comb./Reg.
			Registered	Option/Z	Reg.

Table 2. EPLD vs. PAL/GAL Features

Feature	EPLD	PAL/GAL Device		
	EP330	16L8	16R8	16V8
Array logic	AND/OR	AND/OR	AND/OR	AND/OR
Inputs	18	16	10	18
Outputs	8	8	8	8
Array input lines	36	32	32	32
Product terms	72	64	64	64
D flipflops	8	n.a.	8	8
Reprogrammable	Yes	No	No	Yes

Output Enable Flexibility

Many PAL and GAL devices use one of the 8 product terms in the macrocell to control the Output Enable (OE), leaving only 7 product terms for logic. In contrast, EP330 EPLDs provide a dedicated OE product term that allows the 8-input OR gate to remain intact. Furthermore, many PAL and GAL devices hard-wire the OE to pin 11. Since the EP330 OE logic is implemented directly in the AND array, you can program it to be active-high, active-low, or conditionally asserted by any of the selected inputs and feedback paths.

Design Tools

Altera offers the MAX+PLUS II development system for designing and implementing EP330 designs. The sophisticated MAX+PLUS II development system supports all Classic, MAX 5000, MAX 7000, and STG EPLDs. MAX+PLUS II includes schematic capture design entry; the Altera Hardware Description Language (AHDL), which supports state machine, Boolean equation, and truth table logic; waveform design entry; and automatic design partitioning into multiple devices. After you enter a design, MAX+PLUS II software automatically performs logic synthesis and minimization, and fits the design into the target EPLD(s). You can then program a device in seconds at your desktop to create customized working silicon.

Extensive third-party support also exists for design entry, design processing, and device programming. Altera also provides several free software utilities to quickly convert PAL and GAL designs into EP330 designs. You can convert JEDEC Files from a variety of 20-pin PAL/GAL devices with the PLD2EQN utility, or PALASM1 or 2 files with the PAL2ADF utility, into a MAX+PLUS II-compatible format. For information on other utilities, refer to *Application Brief 73 (Software Utility Programs)* in the Altera 1992 ***Data Book***, or contact Altera Applications at (800) 800-EPLD.

Integrating PAL, GAL & PLA Devices

with the EPM5032 MAX EPLD

April 1992, ver. 3

Application Note 16

Introduction

The EPM5032 EPLD is a high-speed member of the Altera MAX 5000 family of EPLDs. Its highly flexible architecture facilitates general-purpose logic design, and integrates existing functions based on multiple TTL, MSI, PLA, PAL, and GAL elements. Other benefits include flexible logic utilization and register control, programmable I/O pins, and fast system performance. For more information on the EPM5032 EPLD, see the *EPM5016 to EPM5192 EPLDs: High-Speed, High-Density MAX 5000 Devices Data Sheet* in the Altera 1992 ***Data Book***.

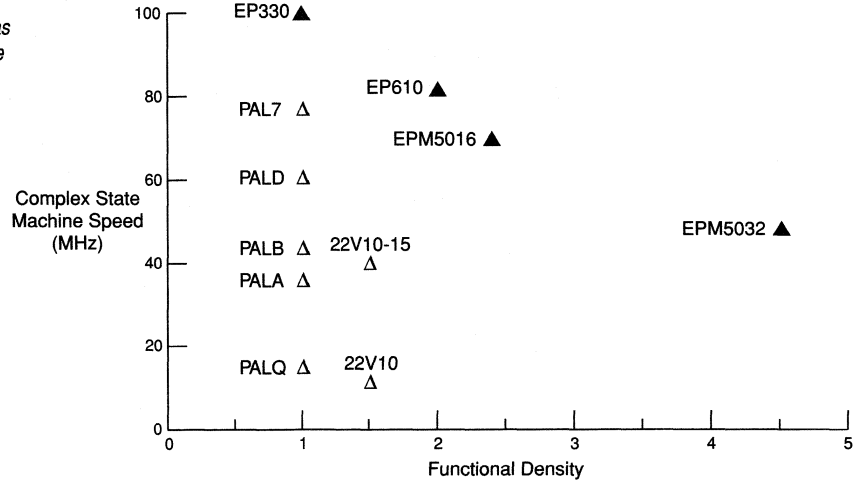
This application note describes the EPM5032 architecture and its advantages over multi-PAL, -GAL, and -PLA implementations; sample designs that show how to replace PLA, PAL, and TTL devices are also included.

EPM5032 vs. PAL, GAL & PLA Architectures

The EPM5032 EPLD provides 28 pins and 32 macrocells. It can be programmed to implement up to 64 latches or 42 flipflops, as well as powerful combinatorial and control-logic functions. This EPLD achieves counter frequencies up to 76.9 MHz. The EPM5032 can emulate PAL, GAL, and PLA structures, and can implement functions that are not possible with these devices. It provides more than four times the integration density of traditional PAL, GAL, and PLA devices, while supporting complex state machine Clock rates of 47 MHz. See Figure 1.

Figure 1. EPM5032 EPLD vs. PAL Devices

The EPM5032 EPLD has more than four times the functional density of traditional PALs.



The basic EPM5032 architecture is an evolution of the AND/OR array structure that uses available logic far more efficiently than PAL, GAL, or PLA devices. Statistical analysis of hundreds of designs shows that 70% of all applications use no more than 3 product terms. The fixed-product-term allocation of PALs usually wastes 5 of the 8 product terms. In contrast, the MAX 5000 EPLDs use a streamlined macrocell with 3 product terms, which can be supplemented—when required—with as many as 64 additional shareable expander product terms. While PLA structures provide more flexible logic distribution, the cost is a considerable loss in performance.

Shareable expanders increase the effective EPLD density and permit higher integration levels than standard devices. A single macrocell can implement very complex equations, leaving other macrocells free to perform additional functions. Macrocells can also share expanders to implement functions with common product terms (e.g., state machines) efficiently.

You can also use shareable expanders to create additional latches and flipflops. Up to 32 latches can be created without consuming macrocell registers. You can cross-couple 2 shareable expanders to form an SR latch or use six shareable expanders to implement a D register or latch. The MAX+PLUS II TTL MacroFunction Library provides several ready-made expander latches and registers. In contrast, PAL and GAL devices require additional macrocells to implement these functions.

Figure 2 shows how a single EPM5032 EPLD can replace multiple PALs, GALs, PLAs and additional TTL glue logic.

Figure 2. Using an EPM5032 EPLD to Replace PAL/GAL/PLA and TTL Glue Logic

One EPM5032 EPLD (32 macrocells) replaces these devices:

PAL/GAL/PLA	Macrocells		TTL	Macrocells
16L8	8	74161	(Counter)	2
16R8	8	74153	(Multiplexer)	4
20X10	10	7485	(Comparator)	4
22V10	10	74178	(Shift Register)	2
PLS105	12	74180	(Parity Generator)	2
PLS15x	12	74374	(8-Bit Register)	8
PLS16x	12			
PLUS405	16 (8 buried)			
GAL6001	18			
PLC473	11			

Any Standard PAL/GAL/PLA + **Added Glue Logic**

Replacing 7400 TTL Devices

EPM5032 register and I/O features can emulate 7400-series TTL devices and replace programmable logic and associated discrete TTL glue logic. Table 1 shows common TTL devices and their typical EPM5032 EPLD utilization.

Table 1. EPM5032 TTL Device Integration

Description	Device	% Used
4-Bit Latch	7475	8
4-Bit Magnitude Comparator	7485	13
8-Bit Shift Register	7491	19
Dual 4-to-1 Multiplexer	74153	4
Quad 2-to-1 Multiplexer	74157	5
4-Bit Binary Counter	74161	10
4-Bit Decade Counter	74162	10
Dual 4-Bit D Register	74173	16
4-Bit Shift Register	74179	10
9-Bit Parity Generator	74180	16

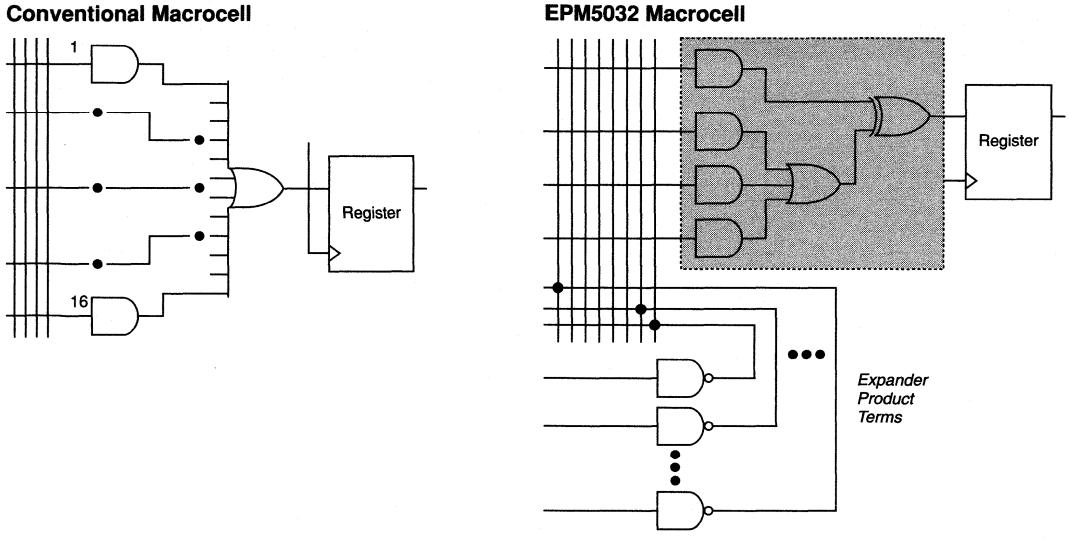
The MAX+PLUS II TTL MacroFunction Library contains over 300 TTL-equivalent functions to simplify design entry with 7400-series functions. Since the MAX+PLUS II Compiler automatically performs all design translation and optimization, you can implement a TTL macrofunction simply by entering the symbol in a schematic, or by including a Function Prototype in an Altera Hardware Description Language (AHDL) Text Design File (.TDF). The macrocell architecture provides true TTL emulation. Since the Compiler can also merge schematic, AHDL, waveform, and EDIF design files, you can create any part of a design (called a "project" in MAX+PLUS II) with the most appropriate entry method.

Replacing PAL Devices

PAL circuits have a programmable-AND/fixed-OR structure. Each macrocell generally includes 7 (e.g., 16L8) or 8 (e.g., 16R8) product terms. Advanced PALs, such as the 22V10 device, support variable allocation of product terms for macrocells, with up to 16 product terms per macrocell. However, it is not possible to reallocate product terms or share product terms between macrocells.

Figure 3 shows how the EPM5032 EPLD emulates a macrocell with a high product-term count. The EPM5032 macrocell directly implements the AND/OR logic expression of a PAL. The shareable expanders provide product-term expansion if an individual application needs more product terms. An EPM5032 macrocell can have as many as 64 product terms, far more than any existing PAL device. This architecture is ideal for projects with multiple high product-term-count outputs and common input signals.

Figure 3. Emulating High Product-Term-Count Macrocells

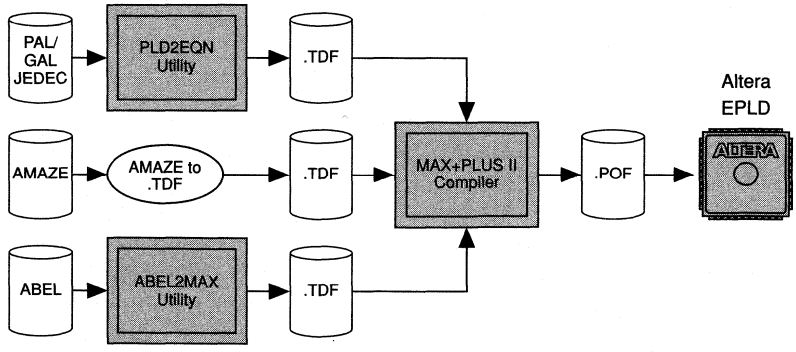


For example, state machines, arithmetic logic circuits, and complex combinatorial functions fit easily in MAX 5000 EPLDs such as the EPM5032, but are often impossible to design efficiently into PALs.

You can incorporate existing PAL and GAL designs into any MAX+PLUS II project (see Figure 4). Altera's PLD2EQN and ABEL2MAX utilities convert standard PAL and GAL JEDEC files and ABEL-generated .TT2

Figure 4. PAL, GAL, and PLA Design File Conversion

PLD2EQN and ABEL2MAX are free utility programs available from Altera's electronic bulletin board service (BBS). AMAZE-format files can be quickly converted to Altera Text Design File (.TDF) format.

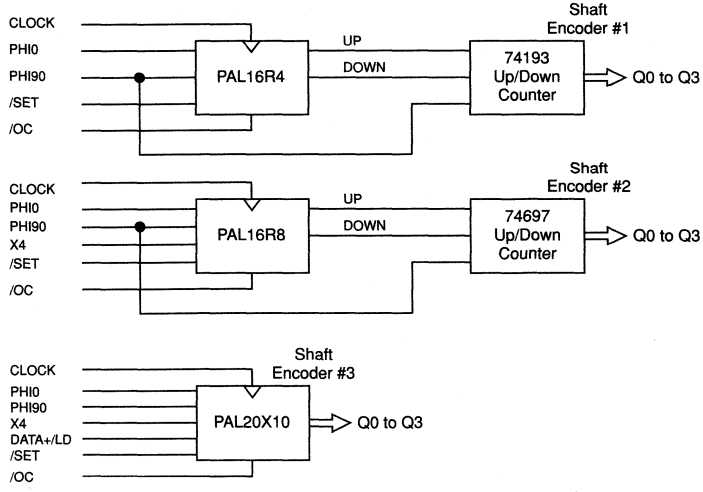


files, respectively, into AHDL TDFs. Both utilities are available free from Altera's electronic bulletin board service (BBS). (See *Electronic Bulletin Board Service* in this handbook.) You can process the TDFs with the MAX+PLUS II Compiler, enter the automatically generated symbols that represent the files in a schematic, and connect the symbols to implement multiple PAL, GAL, and PLA devices in a single project.

PAL Design Example

The *Shaft Encoders* example in the AMD ***PAL Device Handbook*** shows a multi-PAL/TTL implementation of a digital shaft encoder. The circuit measures shaft rotation by counting and comparing pulses produced by a pair of LEDs, a pair of photo sensors, and a rotating disk. The circuits are divided into three major blocks. The first is a set of registers that accepts a pair of 90° out-of-phase digital pulse trains as inputs, one from each photosensor. The registers discriminate the phase difference between the two signals and feed them into the decoder, which is the next block of the circuit. The decoder converts the register outputs into an up-and-down signal to control the final block, the counter. Outputs from the counter determine the position and direction of shaft rotation. Figure 5 shows the circuitry required: three PAL devices (16R4, 16R8, and 20X10) and two discrete TTL devices (74193 and 74697). The functionality of all five devices fits into a single EPM5032 EPLD, with room to implement additional logic.

Figure 5. Shaft Encoder



The 16R4 device implements a set of four registers for phase discrimination, and decoding circuitry that produces UP and DOWN control signals for a 74193 counter device (see Figure 5). Altera's PLD2EQN utility converts the original PAL JEDEC File to an equivalent AHDL TDF within minutes.

Although the PAL implementation includes a control signal (/OC) for disabling the tri-state buffer outputs, it is removed from the TDF because it is not needed in a fully integrated project. See Design File 1 later in this application note.

The second PAL device, 16R8, implements four discrimination registers and decoding circuitry with outputs that feed a counter. Unlike the first PAL, the decoded outputs are configured into UP/DOWN and COUNT control signals, and use a 74697 counter device instead of a 74193 device. PLD2EQN translates the 16R8 PAL JEDEC File into a TDF in minutes. The control signal /OC is also removed. See Design File 2 later in this application note.

The third PAL device (20X10) is more sophisticated. It implements register and decoding functions as well as an up/down counter. Because this PAL includes its own counter and its signals appear on output pins in the integrated project, the OCN_oen Output Enable control is retained in the TDF. See Design File 3 later in this application note.

After converting the 3 PAL designs into TDFs, you can open the TDFs with the MAX+PLUS II Text Editor and create symbols that represent the TDFs. Symbols and AHDL Function Prototypes for the 74193 and 74697 macrofunctions are available in the MAX+PLUS II TTL MacroFunction Library. Figure 6 shows how you enter and connect the PAL design symbols and macrofunction symbols in a MAX+PLUS II Graphic Design File (.GDF) to complete the project. Compilation quickly reveals that this project easily fits into a single EPM5032 EPLD, using less than 90% of the device. Only 29 macrocells and 29 shared expanders are needed to integrate the 3 PALs and the 2 discrete counters, leaving room to implement additional logic.

Figure 6. MAX+PLUS II Shaft Encoder Schematic

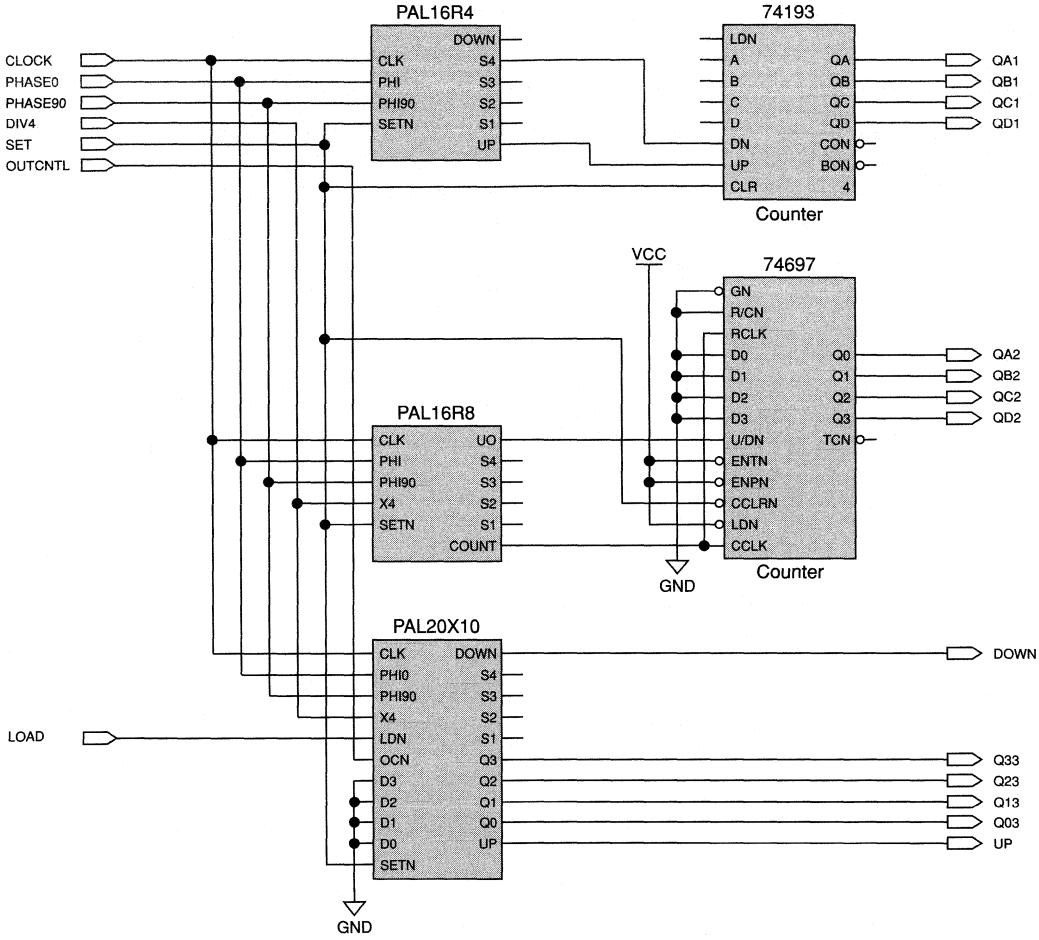


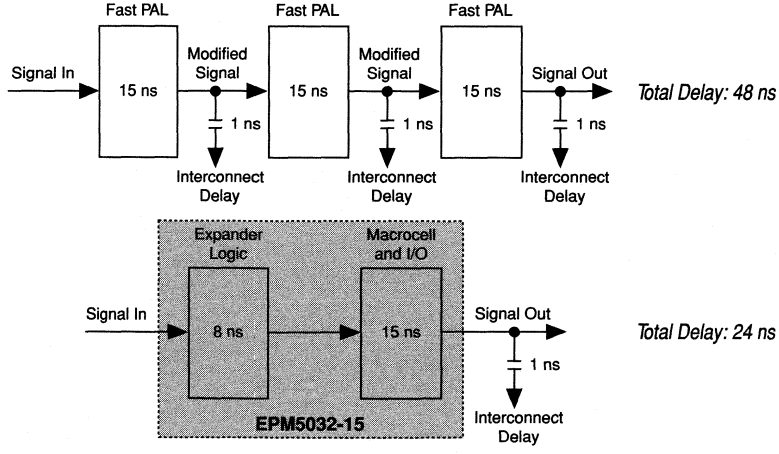
Figure 7 shows the pin-out from the Report File (.RPT) generated by the MAX+PLUS II Compiler.

Figure 7. Shaft Encoder Pin-Out (Report File Excerpt)

```
                    EPM5032
                    - - - - -
          SET -|1       28|- CLOCK
          GND -|2       27|- DIV4
     RESERVED -|3       26|- DOWN
     RESERVED -|4       25|- QA1
           UP -|5       24|- QA2
          Q33 -|6       23|- QB1
          VCC -|7       22|- VCC
          GND -|8       21|- GND
          Q23 -|9       20|- QB2
          Q13 -|10      19|- QC1
          Q03 -|11      18|- QC2
          QD2 -|12      17|- QD1
      PHASE90 -|13      16|- LOAD
       PHASE0 -|14      15|- OUTCNTL
                    - - - - -
```

The EPM5032 EPLD provides increased performance by replacing multiple PAL and PLA devices. The "single-device solution" saves more board space than standard devices and eliminates device-to-device delays incurred by projects that require multiple PALs or PLAs (see Figure 8). For example, a PAL design with three logic levels has a total delay of 48 ns: three 15-ns propagation delays and three 1-ns interconnect delays. You can implement the same project in the EPM5032-15 with one level of expander logic and one level of macrocell logic for a total delay of 24 ns, including the 1-ns interconnect delay to the next stage. This single-device implementation provides faster speeds and better overall system performance.

Figure 8. Replacing High-Speed PALs with the EPM5032-15 EPLD



Replacing PLAs

The EPM5032 EPLD can also implement PLA architectures. PLAs provide an additional programmable array not available in PALs, which allows variable product-term distribution.

Figure 9 shows a typical PLA structure, which consists of a programmable-AND array (containing 32 to 48 AND gates) that feeds a programmable-OR array. Each AND gate is fed by 8 to 12 input pins and feedback signals, plus their complements. The OR array allows logical ORing of any of the AND terms in the array. Typically, 8 to 12 programmed OR terms feed output pins, or feed back to combinatorial buffers or registers.

Figure 9. Traditional PLA Architecture

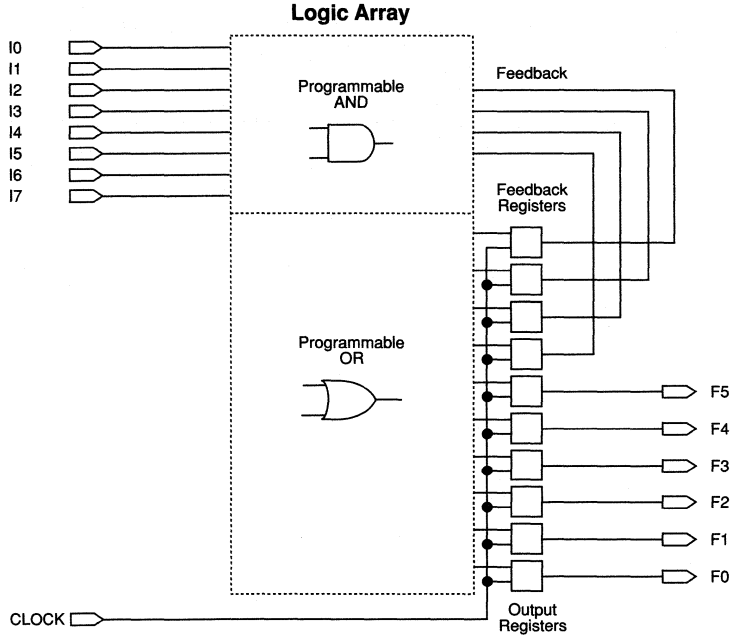
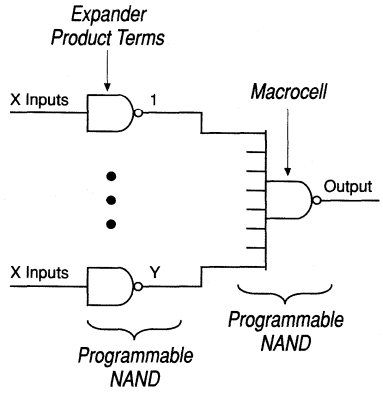


Figure 10 shows how the EPM5032 EPLD expander product-term array emulates the PLA programmable-AND/programmable-OR structure. As in high-product-term PAL applications, the unallocated product terms feed an inverted AND gate in the macrocell to create an AND/OR-equivalent NAND/NAND structure.

Figure 10. Emulating the PLA AND/OR Structure with the EPM5032 EPLD



You can incorporate existing PLAs into EPM5032 projects by converting the Signetics PLA source files (written in the AMAZE language) to the Altera AHDL format. Since the formats are similar, you can quickly convert the source files by hand with a standard text editor. Figure 11 shows both an AMAZE file and an AHDL TDF of a state machine beverage dispenser. Note the similarities in state and transition definition. After you transform an AMAZE file into a TDF, the MAX+PLUS II Compiler can process it directly.

Figure 11. AHDL vs. AMAZE File Formats

Altera AHDL

```
SUBDESIGN BEVDIS
(
   clock, coindrop, cupfull : INPUT;
   dropcup, pourdrnk        : OUTPUT
)
VARIABLE
  vending : MACHINE OF BITS (dropcup, pourdrnk)
            WITH STATES (s1 = B"00",
                         s2 = B"10",
                         s3 = B"01");
BEGIN
  vending.clk = clock;
  CASE vending IS
    WHEN s1 => IF coindrop THEN
                  vending = S2;
               END IF;
    WHEN s2 => vending = S3;
    WHEN s3 => IF cupfull THEN
                  vending = S1;
               END IF;
  END CASE;
END;
```

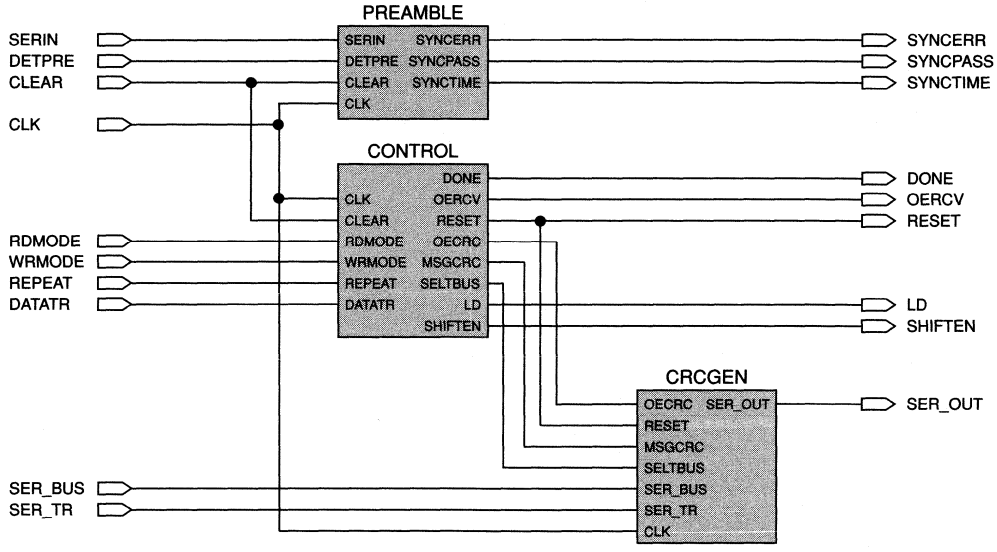
Signetics AMAZE

```
STATE VECTORS
S1 = 00b;
S2 = 10b;
S3 = 01b;
INPUT VECTORS
[coindrop, cupfull]
OUTPUT VECTORS
[dropcup, pourdrnk]
out1 = 00b;
out2 = 10b;
out3 = 01b;
TRANSITIONS
WHILE [S1]
    IF [/coindrop] THEN [s1] WITH [out1]
    IF [coindrop] THEN [s2] WITH [out2]
WHILE [s2] THEN [sS3] WITH [out3]
WHILE [s3]
    IF [cupfull] THEN [s1] WITH [out1]
```

PLA Design Example

The *Custom Communication Protocol—PLS105A, 157, 167, 168A* section of the Signetics ***Sequencer Solutions*** manual describes a serial communications interface with a custom protocol. The schematic shown in Figure 12 consists of the following three functional blocks, each of which is implemented in a Signetics PLA. You can integrate these three functions into a single EPM5032 EPLD with MAX+PLUS II software.

Figure 12. Serial Communications Interface

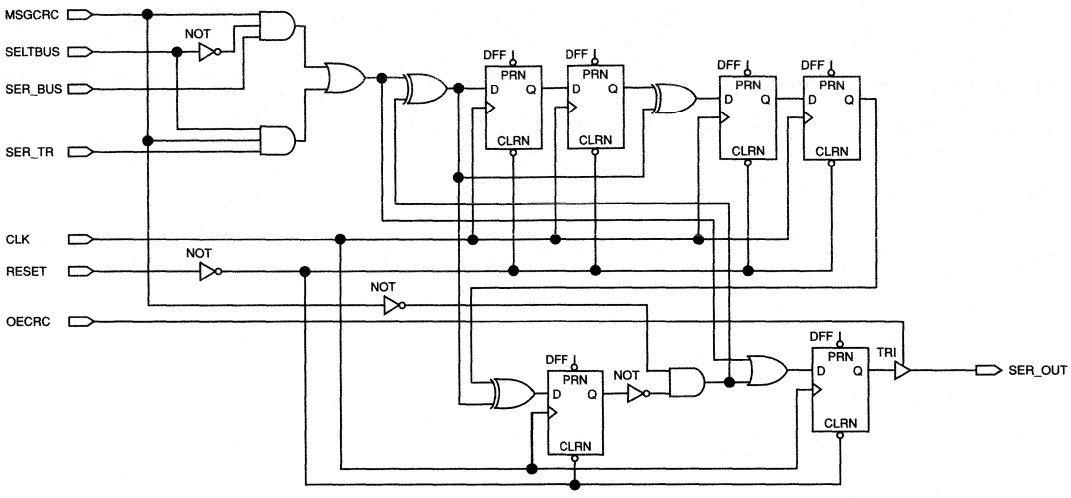


Preamble Sequence Detector This function (PREAMBLE) is a state machine that recognizes a bit pattern that signals the beginning of a valid data block. The preamble sequence detector is implemented in the Signetics PLS167A. After the Signetics AMAZE source file is transcribed into a TDF, the design fits into less than one-third of an EPM5032 EPLD.

Cyclic Redundancy Check (CRC) Controller The CRC controller (CONTROL) is a state machine that coordinates the loading of parallel-to-serial and serial-to-parallel shift registers and provides status of data and CRC transmissions. A Signetics source file is converted into a TDF to implement the CRC controller. The Signetics implementation requires 100% register utilization of a PLS105A, while Altera's CRC controller implementation uses less than 40% of the EPM5032 EPLD.

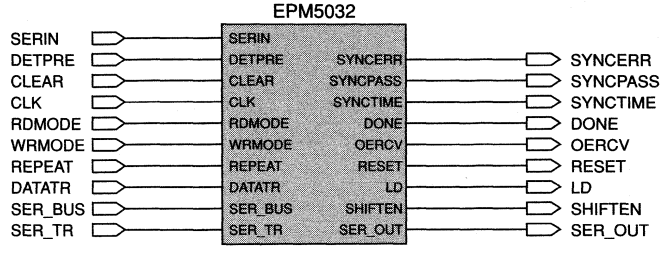
CRC Generator The 5-bit CRC generator (CRCGEN) detects errors. Figure 13 shows the schematic for the CRC generator. It is submitted directly to MAX+PLUS II for processing. This module requires 100% register utilization of a PLS157, but consumes less than 20% of the EPM5032 EPLD.

Figure 13. CRC Generator (CRCGEN.GDF)



MAX+PLUS II simplifies integration of the three functions. You process each of the three source files separately with the MAX+PLUS II Compiler, which automatically generates symbols that represent each function. You enter these symbols in the MAX+PLUS II Graphic Editor, connect them, and compile the final schematic, as shown in Figure 12. The PLS105A, PLS157, and PLS167A designs all fit into one EPM5032 EPLD (see Figure 14).

Figure 14. Integrated Serial Communications Interface



Conclusion

The EPM5032 EPLD offers better integration than PAL, GAL, or PLA devices and a flexible architecture that supports a broad range of applications. A single EPM5032 EPLD typically replaces three to four conventional PLDs, and can also provide extensive TTL integration. As shown in the sample projects, one EPM5032 can replace three PLAs or

three PALs and two TTL devices. Projects can be easily converted from PAL, GAL, or PLA devices to the EPM5032. The EPM5032 EPLD also offers superior logic utilization, more flexible register options, better control of I/O pins, and better system performance.

The DESFILE1.DOC, DESFILE2.DOC, and DESFILE3.DOC files, used to generate the PAL Shaft Encoder, are shown in Design Files 1, 2, and 3. These files are available as part of the self-extracting archive file EAN016.EXE, which you can download via modem from the Electronic Application Briefs directory of the Altera BBS at (408) 249-1100. After you extract the files, you can rename them as PAL16R4.TDF, PAL16R8.TDF, and PAL20X10.TDF, respectively, and connect them as shown in Figure 6 to create the shaft encoder project.

Design File 1. PAL 16R4-Equivalent AHDL File

```
% PLD2EQN version: 1.0 processed: Tue Mar 31 09:04:50 1992 %

TITLE "logic from JEDEC File SHFT_ONE";
DESIGN IS "PAL16R4" DEVICE IS "auto";

%
PAL16R4
SE1
%

SUBDESIGN PAL16R4
(
   CLK, PHI0, PHI90, SETN   : INPUT;
   DOWN, S4, S3, S2, S1, UP : OUTPUT;
)
VARIABLE
   down_din,
   s4_fbk, s4_din,
   s3_fbk, s3_din,
   s2_fbk, s2_din,
   s1_fbk, s1_din,
   up_din                   : NODE;
BEGIN

  DOWN      = TRI(down_din, VCC);
  !down_din =  PHI0 &  PHI90 &  s1_fbk &  s2_fbk &  s3_fbk & !s4_fbk
            # !PHI0 & !PHI90 & !s1_fbk & !s2_fbk & !s3_fbk &  s4_fbk
            #  PHI0 & !PHI90 &  s1_fbk & !s2_fbk & !s3_fbk & !s4_fbk
            # !PHI0 &  PHI90 & !s1_fbk &  s2_fbk &  s3_fbk &  s4_fbk;

  S4        = TRI(s4_fbk, VCC);
  s4_fbk    = DFF(s4_din, CLK, !GND, !GND);
  !s4_din   = !s3_fbk
            # !SETN;

  S3        = TRI(s3_fbk, VCC);
  s3_fbk    = DFF(s3_din, CLK, !GND, !GND);
  !s3_din   = !PHI90
            # !SETN;

  S2        = TRI(s2_fbk, VCC);
  s2_fbk    = DFF(s2_din, CLK, !GND, !GND);
  !s2_din   = !s1_fbk
            # !SETN;

  S1        = TRI(s1_fbk, VCC);
  s1_fbk    = DFF(s1_din, CLK, !GND, !GND);
  !s1_din   = !PHI0
            # !SETN;

  UP        = TRI(up_din, VCC);
  !up_din   = !PHI0 &  PHI90 & !s1_fbk & !s2_fbk &  s3_fbk & !s4_fbk
            #  PHI0 & !PHI90 &  s1_fbk &  s2_fbk & !s3_fbk &  s4_fbk
            #  PHI0 &  PHI90 &  s1_fbk & !s2_fbk &  s3_fbk &  s4_fbk
            # !PHI0 & !PHI90 & !s1_fbk &  s2_fbk & !s3_fbk & !s4_fbk;
END;
```

Design File 2. PAL 16R8-Equivalent AHDL File (Part 1 of 2)

```
% PLD2EQN version: 1.0 processed: Tue Mar 31 05:17:47 1992 %

TITLE "logic from JEDEC File SHFT_TWO";
DESIGN IS "PAL16R8" DEVICE IS "auto";

%
PAL16R8
SE2
%

SUBDESIGN PAL16R8
(
   CLK, PHI0, PHI90, X4, SETN : INPUT;
   UD, S4, S3, S2, S1, COUNT  : OUTPUT;
)
VARIABLE
   UD_fbk, UD_din,
   S4_fbk, S4_din,
   S3_fbk, S3_din,
   S2_fbk, S2_din,
   S1_fbk, S1_din,
   COUNT_fbk, COUNT_din       : NODE;
BEGIN

  UD         = TRI(UD_fbk, VCC);
  UD_fbk     = DFF(UD_din, CLK, !GND, !GND);
  !UD_din    = !S1_fbk &  S2_fbk & !S3_fbk &  S4_fbk
             # !S1_fbk &  S2_fbk &  S3_fbk &  S4_fbk
             # !S1_fbk &  S2_fbk &  S3_fbk & !S4_fbk
             #  S1_fbk &  S2_fbk &  S3_fbk & !S4_fbk
             #  S1_fbk & !S2_fbk &  S3_fbk & !S4_fbk
             #  S1_fbk & !S2_fbk & !S3_fbk & !S4_fbk
             #  S1_fbk & !S2_fbk & !S3_fbk &  S4_fbk
             # !S1_fbk &  S2_fbk & !S3_fbk &  S4_fbk;

  S4         = TRI(S4_fbk, VCC);
  S4_fbk     = DFF(S4_din, CLK, !GND, !GND);
  !S4_din    =  S3_fbk
             # !SETN;

  S3         = TRI(S3_fbk, VCC);
  S3_fbk     = DFF(S3_din, CLK, !GND, !GND);
  !S3_din    = !PHI90
             # !SETN;

  S2         = TRI(S2_fbk, VCC);
  S2_fbk     = DFF(S2_din, CLK, !GND, !GND);
  !S2_din    =  S1_fbk
             # !SETN;

  S1         = TRI(S1_fbk, VCC);
  S1_fbk     = DFF(S1_din, CLK, !GND, !GND);
  !S1_din    = !PHI0
             # !SETN;

  COUNT      = TRI(COUNT_fbk, VCC);
  COUNT_fbk  = DFF(COUNT_din, CLK, !GND, !GND);
```

Design File 2. PAL 16R8-Equivalent AHDL File (Part 2 of 2)

```
!COUNT_din  =  S1_fbk &  S2_fbk & !S3_fbk &  S4_fbk
            # !S1_fbk & !S2_fbk &  S3_fbk & !S4_fbk
            #  X4 & !S1_fbk &  S2_fbk & !S3_fbk & !S4_fbk
            #  X4 &  S1_fbk & !S2_fbk &  S3_fbk &  S4_fbk
            #  S1_fbk &  S2_fbk &  S3_fbk & !S4_fbk
            # !S1_fbk & !S2_fbk & !S3_fbk &  S4_fbk
            #  X4 & !S1_fbk &  S2_fbk &  S3_fbk &  S4_fbk
            #  X4 &  S1_fbk & !S2_fbk & !S3_fbk & !S4_fbk;
END;
```

Design File 3. PAL 20X10-Equivalent AHDL File (Part 1 of 2)

```
% PLD2EQN version: 2.0 processed: Tue Mar 31  17:36:20 1992
    command line options: m p                                %

TITLE "logic from JEDEC File SHFT_3";
DESIGN IS "PAL20X10" DEVICE IS "auto";
%
PAL20X10
SE3
%
SUBDESIGN PAL20X10
(
   CLK, PHI0, PHI90, X4, LDN, D3, D2,
   D1, D0, SETN, OCN                  : INPUT = GND;
   UP, Q0, Q1, Q2, Q3, S1, S2, S3,
   S4, DOWN                           : OUTPUT;
)
VARIABLE
   CLK_clk,
   UP_fbk, UP_din,
   Q0_fbk, Q0_din,
   Q1_fbk, Q1_din,
   Q2_fbk, Q2_din,
   Q3_fbk, Q3_din,
   S1_fbk, S1_din,
   S2_fbk, S2_din,
   S3_fbk, S3_din,
   S4_fbk, S4_din,
   DOWN_fbk, DOWN_din,
   OCN_oen                            : NODE;
BEGIN
  CLK_clk   = CLK;
  UP        = TRI(UP_fbk, OCN_oen);
  UP_fbk    = DFF(UP_din, CLK_clk, !GND, !GND);
  !UP_din   =(!PHI0 &  PHI90 & !S1_fbk & !S2_fbk &  S3_fbk & !S4_fbk
            #  PHI0 & !PHI90 &  S1_fbk &  S2_fbk & !S3_fbk &  S4_fbk)
            $ (PHI0 &  PHI90 &  X4 &  S1_fbk & !S2_fbk &  S3_fbk &
                S4_fbk
            # !PHI0 & !PHI90 &  X4 & !S1_fbk &  S2_fbk & !S3_fbk &
               !S4_fbk);

  Q0        = TRI(Q0_fbk, OCN_oen);
  Q0_fbk    = DFF(Q0_din, CLK_clk, !GND, !GND);
  !Q0_din   = (!LDN & !D0 &  SETN
            # !Q0_fbk &  LDN &  SETN)
            $ (UP_fbk &  LDN &  SETN & !DOWN_fbk
            # !UP_fbk &  LDN &  SETN &  DOWN_fbk);
```

Design File 3. PAL 20X10-Equivalent AHDL File (Part 2 of 2)

```
  Q1          = TRI(Q1_fbk, OCN_oen);
  Q1_fbk      = DFF(Q1_din, CLK_clk, !GND, !GND);
  !Q1_din     = (!LDN & !D1 &  SETN
              # !Q1_fbk &  LDN &  SETN)
              $ (UP_fbk & !Q0_fbk &  LDN &  SETN & !DOWN_fbk
              # !UP_fbk &  Q0_fbk &  LDN &  SETN &  DOWN_fbk);

  Q2          = TRI(Q2_fbk, OCN_oen);
  Q2_fbk      = DFF(Q2_din, CLK_clk, !GND, !GND);
  !Q2_din     = (!LDN & !D2 &  SETN
              #  LDN & !Q2_fbk &  SETN)
              $ (UP_fbk & !Q0_fbk & !Q1_fbk &  LDN &  SETN & !DOWN_fbk
              # !UP_fbk &  Q0_fbk &  Q1_fbk &  LDN &  SETN &  DOWN_fbk);

  Q3          = TRI(Q3_fbk, OCN_oen);
  Q3_fbk      = DFF(Q3_din, CLK_clk, !GND, !GND);
  !Q3_din     = (!LDN & !D3 &  SETN
              #  LDN & !Q3_fbk &  SETN)
              $ (UP_fbk & !Q0_fbk & !Q1_fbk &  LDN & !Q2_fbk &  SETN & !DOWN_fbk
              # !UP_fbk &  Q0_fbk &  Q1_fbk &  LDN &  Q2_fbk &  SETN &  DOWN_fbk);

  S1          = TRI(S1_fbk, OCN_oen);
  S1_fbk      = DFF(S1_din, CLK_clk, !GND, !GND);
  !S1_din     = (GND)
              $ (!SETN
              # !PHI0);

  S2          = TRI(S2_fbk, OCN_oen);
  S2_fbk      = DFF(S2_din, CLK_clk, !GND, !GND);
  !S2_din     = (GND)
              $ (!SETN
              # !S1_fbk);

  S3          = TRI(S3_fbk, OCN_oen);
  S3_fbk      = DFF(S3_din, CLK_clk, !GND, !GND);
  !S3_din     = (GND)
              $ (!SETN
              # !PHI90);

  S4          = TRI(S4_fbk, OCN_oen);
  S4_fbk      = DFF(S4_din, CLK_clk, !GND, !GND);
  !S4_din     = (GND)
              $ (!SETN
              # !S3_fbk);

  DOWN        = TRI(DOWN_fbk, OCN_oen);
  DOWN_fbk    = DFF(DOWN_din, CLK_clk, !GND, !GND);
  !DOWN_din   = (PHI0 &  PHI90 &  X4 &  S1_fbk &  S2_fbk &  S3_fbk & !S4_fbk
              # !PHI0 & !PHI90 &  X4 & !S1_fbk & !S2_fbk & !S3_fbk &  S4_fbk)
              $ (PHI0 & !PHI90 &  S1_fbk & !S2_fbk & !S3_fbk & !S4_fbk
              # !PHI0 &  PHI90 & !S1_fbk &  S2_fbk &  S3_fbk &  S4_fbk);
  OCN_oen     = !OCN;
END;
```

Integrating an Intelligent I/O Subsystem with a Single EPM5130 MAX EPLD

April 1992, ver. 3

Application Note 17

Introduction

When higher system performance is necessary, many designers first consider a faster microprocessor or a new microprocessor architecture. In many cases, however, sufficient speed can be achieved with an intelligent I/O subsystem that has been optimized for a particular task. Transferring I/O processing to intelligent subsystems also allows the system processor to dedicate more processing power to primary system functions.

This application note describes how the custom logic requirements of such a subsystem can be integrated into a single Altera user-configurable EPM5130 MAX EPLD, replacing over 75 standard TTL packages. The sample design used to illustrate the design process is an intelligent T1 serial coprocessor.

This application note also discusses how you can use MAX+PLUS II, Altera's advanced development system, to enter, compile, and program an intelligent I/O subsystem design (designs are called "projects" in MAX+PLUS II). Familiarity with the EPM5130 EPLD and MAX+PLUS II is assumed.

T1 Serial Coprocessor

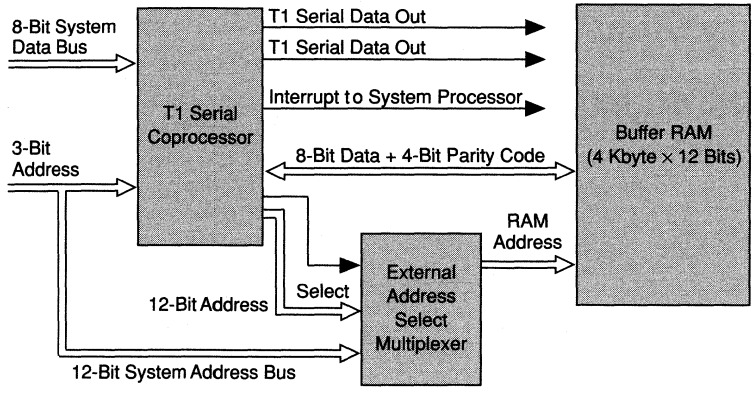
The T1 serial coprocessor is a useful example of an intelligent I/O subsystem because it contains the types of logic that are common to most digital designs, such as decoders and control state machines. Its error detection and correction (EDAC), RAM control and address generation, and parallel-to-serial conversion features are also widely used in system design. Furthermore, digital data communication of both voice and data plays an increasingly important role in modern systems.

The T1 protocol for serial data transmission has been the basis for most digital voice communications since its introduction in the early 1960s, and was revised to include support for digital data communications. The protocol is based on a pulse-code modulation system in which multiple channels are time-division multiplexed onto a 1.544-MHz channel. The EPM5130 EPLD's flexible architecture and the MAX+PLUS II development system make it possible to design and integrate a T1 serial transmitter subsystem in a matter of hours.

Figure 1 shows a subsystem that uses the T1 serial coprocessor implemented in an EPM5130 EPLD. The serial coprocessor consists of two T1 serial transmitters, control and address generation logic for a buffer RAM, and the I/O subsystem control logic. The system processor writes data for

Figure 1. Block Diagram of a Subsystem with a T1 Serial Coprocessor

Altera's EPM5130 EPLD can integrate the logic requirements of high-performance I/O subsystems such as the intelligent T1 serial coprocessor.



serial transmission through the T1 serial coprocessor into the buffer RAM. When data transfer is complete, the system processor signals to the serial coprocessor that data can be transmitted. The serial coprocessor then transfers the data to the transmitters for serialization. The buffer RAM control logic also features error detection and correction. Data can be sent over either T1 serial channel with individual variations in protocol. Once all of the data in the buffer RAM is transferred, the processor is interrupted, and the cycle can repeat.

Although this sample project shows a T1 serial transmission application operating at 1.544 MHz, you can apply the same concept to create high-performance subsystems for applications such as local area networks or disk controllers. The EPM5130 EPLD supports serial data rates of up to 50 Mbits per second. It is user-configurable to allow logic customization to fit the application, rather than compromising the application by fitting it into available standard components.

EPM5130 Overview

The EPM5130 EPLD is a member of Altera's high-density MAX 5000 family. The EPLD consists of 128 flexible macrocells that are grouped into 8 Logic Array Blocks (LABs). Each LAB contains 16 macrocells and 32 shareable expander product terms (expanders). Shareable expanders are freely allocatable product terms that can be used and shared by any macrocell function within an LAB. These functions can be purely combinatorial, or can be register control functions such as array (asynchronous) Preset, array Clear, and the register Clock signal.

A Programmable Interconnect Array (PIA) routes signals between the LABs. All 52 decoupled I/O pins and all 128 macrocell outputs enter the PIA. Each LAB can then receive any of the signals it requires from the PIA. The PIA has sufficient routing resources to accommodate the most complex logic. A fixed interconnect delay across the PIA also eliminates skew, allowing accurate timing performance prediction before the project is completed. Refer to the Altera 1992 ***Data Book*** for more information about EPM5130 architecture and performance.

You can create and program EPM5130 designs with Altera's MAX+PLUS II software, which provides a complete CAE design environment featuring hierarchical design entry, logic synthesis, automatic error location, timing simulation, and programming. For more information about MAX+PLUS II, refer to the *PLDS-HPS, PLS-HPS, PLS-OS & PLS-ES Data Sheet* in the Altera 1992 ***Data Book*** and *Product Information Bulletin 12 (MAX+PLUS II and Windows 3.0)*.

Design Entry

MAX+PLUS II provides three design entry methods, allowing you to choose the most appropriate method for each function: hierarchical schematic capture, waveform design entry, and the Altera Hardware Description Language (AHDL), a high-level text design language that supports state machines, Boolean equations, and truth tables.

The serial coprocessor project described in this application note uses Graphic Design Files (.GDF) and an AHDL Text Design File (.TDF). The control functions are implemented as AHDL state machines. AHDL supports advanced features such as automatic state bit assignments for state machine applications. For additional information about AHDL, refer to *Application Note 22 (Designing with AHDL)* in this handbook.

Figure 2 shows a sample GDF, EDAC.GDF, which implements logical error detection and correction. This file includes primitives and macrofunctions that are provided with MAX+PLUS II. It also includes a custom macrofunction (`7486ME`), which is discussed later in the application note.

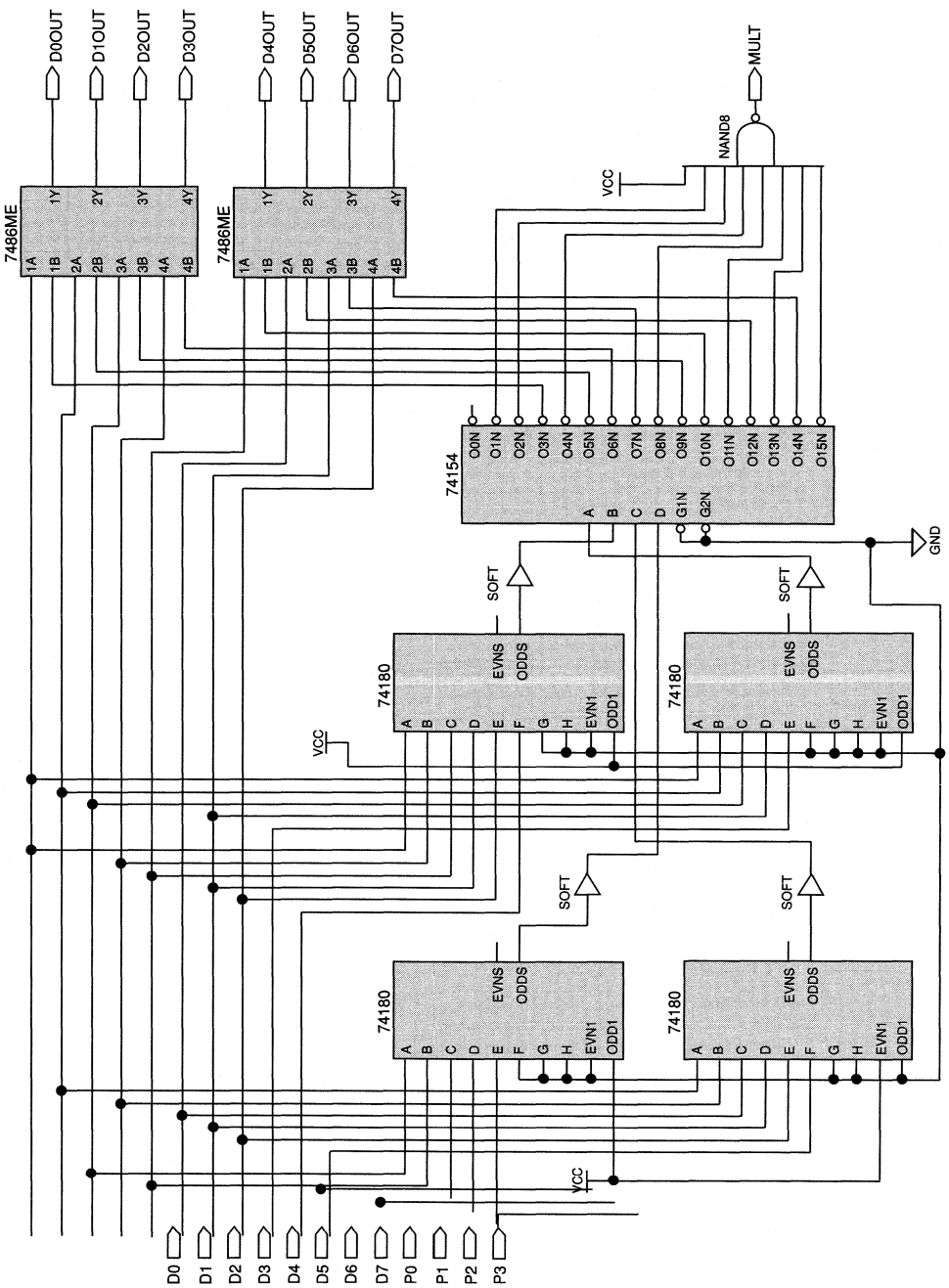
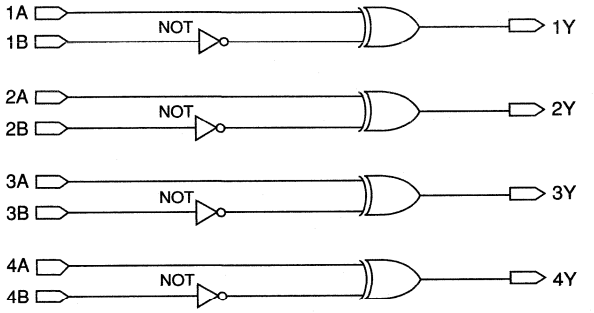


Figure 2. Error Detection and Correction Function (EDAC.GDF)

The custom logic function EDAC is implemented with primitive gates, TTL macrofunctions, and the custom logic function 7486ME in the top-level design file EDAC.GDF

You can also create custom functions with graphic, text, or waveform design entry. For example, the function 7486ME used in EDAC.GDF is a schematic representation of a quad, two-input XOR function based on the standard TTL 7486 macrofunction, but has one input of each XOR gate inverted. See Figure 3.

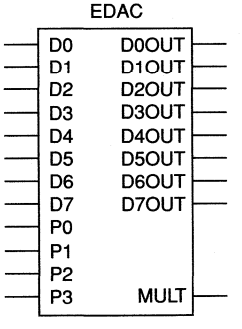
Figure 3. Custom Logic Function 7486ME



When you save a GDF or compile a TDF or Waveform Design File (.WDF), MAX+PLUS II automatically creates a symbol that represents the design file. You can use these symbols to quickly integrate the function into other designs. Figure 4 shows the automatically generated symbol for the EDAC function, which is integrated into the top level of the serial coprocessor design.

Figure 4. EDAC Symbol

The symbol for EDAC.GDF is automatically generated by MAX+PLUS II. This symbol can be easily integrated into other GDFs and TDFs.



Control functions are often represented most conveniently as state machines. You can create AHDL TDFs that include state machines with the MAX+PLUS II Text Editor or any standard text editor. Figure 5 shows the TDF that implements the state machine T1ST. This state machine is integrated into both T1 transmitters, where it controls the serialization of data.

Figure 5. T1ST State Machine

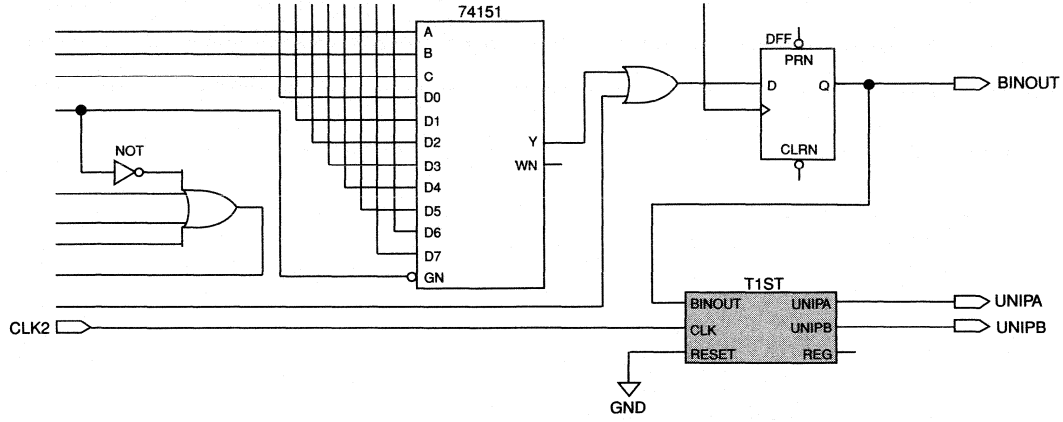
State machines such as T1ST are created with the Altera Hardware Description Language (AHDL).

```
TITLE "T1ST State Machine";

SUBDESIGN t1st
(
    binout, clk, reset : INPUT;
    unipa, unipb, reg  : OUTPUT;
)

VARIABLE
    seqnc: MACHINE OF BITS (q0,q1,q2)
           WITH STATES (s0,s1,s2,s3);

    outputs[2..0] : NODE;

BEGIN
    seqnc.clk   = clk;
    seqnc.reset = reset;

    (reg,unipa,unipb) = outputs[2..0];

    CASE (seqnc) IS
        WHEN s0 =>
            outputs[2..0] = b"000";
            IF (!binout) THEN
                seqnc = s1;
            ELSEIF (binout) THEN
                seqnc = s0;
            END IF;

        WHEN s1 =>
            outputs[2..0] = b"001";
            IF (!binout) THEN
                seqnc = s2;
            ELSEIF (binout) THEN
                seqnc = s1;
            END IF;

        WHEN s2 =>
            outputs[2..0] = b"100";
            IF (!binout) THEN
                seqnc = s3;
            ELSEIF (binout) THEN
                seqnc = s2;
            END IF;

        WHEN s3 =>
            outputs[2..0] = b"110";
            IF (!binout) THEN
                seqnc = s0;
            ELSEIF (binout) THEN
                seqnc = s1;
            END IF;
    END CASE;
END;
```

Completed TDFs can be compiled, simulated, and incorporated into a hierarchical design. Figure 6 shows the symbol for the T1ST state machine and other TTL macrofunctions in a portion of the T1 transmitter schematic.

Figure 6. T1ST Integrated with Other TTL Macrofunctions

Automatic symbol generation allows state machines such as T1ST to be quickly integrated into schematics.



Design Processing

After you enter a project for the EPM5130 EPLD, you can submit it to the MAX+PLUS II Compiler for processing. Compilation typically takes five to fifteen minutes on an 80386-based PC. The Compiler extracts a netlist from each design file, checks for conformity with design rules, and then performs logic synthesis, minimizing and optimizing the project for the EPM5130 architecture. If any errors are detected during compilation, a Message Processor window opens, which can locate the error.

Once a design has been synthesized, the Compiler's Fitter module maps it into the EPM5130 EPLD. The Fitter uses a set of heuristic algorithms that free you from manual routing and interconnect procedures. The Fitter also generates a Report File (.RPT), that provides device utilization information for complete projects, or for portions of the project if the project is partitioned into multiple EPLDs. Figure 7 shows a portion of the Report File with device utilization information for EDAC.GDF.

Figure 7. Report File Excerpt for EDAC.GDF

The Report File is automatically generated during project compilation and contains information on device utilization.

```
                                                    C:\MAX2WORK\MAXAB\EDAC.RPT
**RESOURCE USAGE**

                                                    Shareable    External
Logic Array Block  Macrocells    I/O Pins          Expanders  Interconnect

A:     MC1 - MC16    0/16 (  0%)  0/ 8(  0%)    0/32(  0%)  0/24(  0%)
B:     MC17 - MC32   0/16 (  0%)  0/ 8(  0%)    0/32(  0%)  0/24(  0%)
C:     MC33 - MC48   0/16 (  0%)  0/ 8(  0%)    0/32(  0%)  0/24(  0%)
D:     MC49 - MC64   3/16 ( 18%)  1/ 8( 12%)    4/32( 12%)  1/24(  4%)
E:     MC65 - MC80   0/16 (  0%)  4/ 8( 50%)    0/32(  0%)  0/24(  0%)
F:     MC81 - MC96   0/16 (  0%)  0/ 8(  0%)    0/32(  0%)  0/24(  0%)
G:     MC97 - MC112  0/16 (  0%)  0/ 8(  0%)    0/32(  0%)  0/24(  0%)
H:    MC113 - MC128 16/16 (100%)  8/ 8(100%)   12/32( 37%)  5/24( 20%)

Total dedicated input pins used:                    12/20    (60%)
Total I/O pins used:                                 9/64    (14%)
Total macrocells used:                              19/128   (15%)
Total shareable expanders used:                     16/256   ( 6%)
Total shareable expanders not available (n/a)        0/256   ( 0%)

Total input pins required:                          12
Total output pins required:                          9
Total bidirectional pins required:                   0
Total macrocells required:                          19
Total shareable expanders in database:              16

Synthesized macrocells:                              0/128   ( 0%)
```

Timing Analysis & Simulation

After successful compilation, you can perform source-level delay prediction, event-driven timing simulation, static timing analysis, and functional simulation.

Source-level delay prediction allows you to select one signal as a source and another signal as a destination, then calculate the worst-case delay between the two signals. In EDAC.GDF, for example, if `D1` is specified as the source node and `D1OUT` as the destination node, the worst-case delay path is calculated through the `74180` parity generator/checkers.

You can perform event-driven timing simulation with the interactive timing simulator that verifies the performance and functionality of a project. The MAX+PLUS II Simulator has a resolution of 1/10 ns and can be configured to monitor the project for glitches, oscillations, and setup and hold violations. Stimuli for the simulation can be defined in a Vector File (.VEC) format or created in a graphical WDF in the MAX+PLUS II Waveform Editor. Simulation output, in either table or waveform format, is displayed on-screen or can be printed out. Figure 8 shows the Table File (.TBL) for EDAC.GDF, which contains input and output vectors from a typical simulation.

Figure 8. Table File for EDAC.GDF

Simulation results can be viewed in table format.

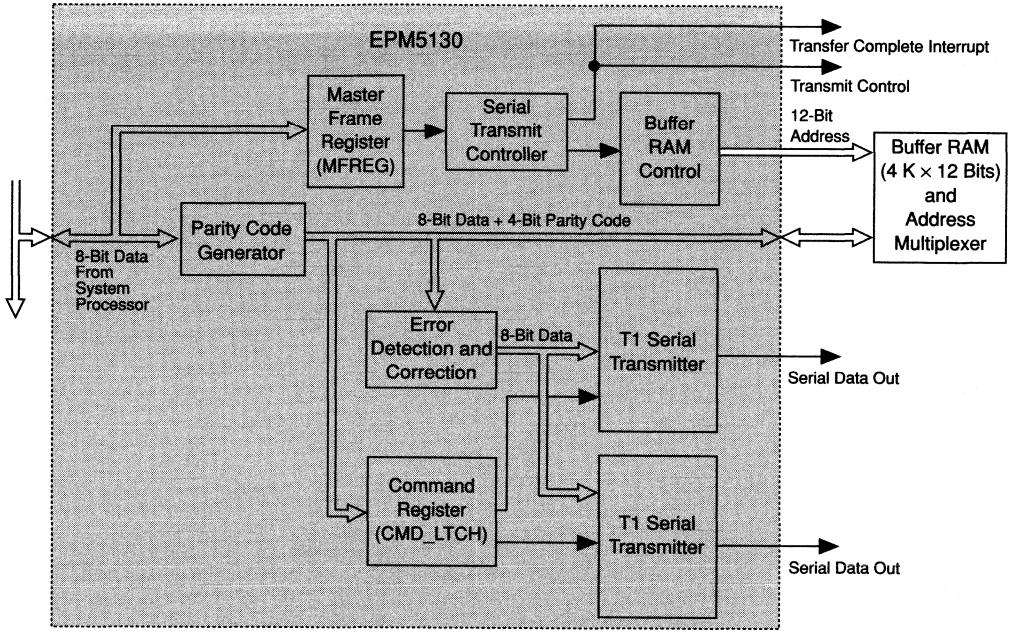
```
INPUTS  D7 D6 D5 D4 D3 D2 D1 D0 P3 P2 P1 P0;
OUTPUTS D7OUT D6OUT D5OU5 D4OUT D3OUT DD2OUT D1OUT D0OUT MULT ;
RADIX HEX ;
UNIT ns ;
PATTERN
%                                    D D D D D D D D   %
%                                    7 6 5 4 3 2 1 0 M %
%                                    O O O O O O O O U %
%          D D D D D D D D P P P P U U U U U U U U L %
%          7 6 5 4 3 2 1 0 3 2 1 0 T T T T T T T T T %
     0.0> 1 0 1 0 1 0 1 0 1 0 1 0 X X X X X X X X X
    23.0> 1 0 1 0 1 0 1 0 1 0 1 0 X X X X X X X X 1
    87.0> 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
  2500.0> 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1
  2529.9> 1 1 1 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 1
  2587.0> 1 1 1 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 0 1
  2603.0> 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1
  5000.0> 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
  5029.9> 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1
  5086.9> 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1
  5087.0> 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
  7500.0> 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
  7529.9> 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 1 1 0 1
  7587.0> 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1
  7603.0> 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
 10000.0> 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1
 10029.9> 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 1 1
 10086.9> 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 1 1
 10102.9> 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1
 12500.0> 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
 12529.9> 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1
 12586.9> 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
 17500.0> 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
 17529.9> 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1
 17586.0> 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1
 17587.0> 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
;
```

Advanced timing analysis tools provide detailed timing information about a design. The Timing Analyzer creates tables that list delay information between sets of nodes in a given design. For example, you can generate a table listing all inputs and outputs, and the worst-case delays between them. The Timing Analyzer can also calculate the setup times for all registers in a design or the highest frequency of operation. For a complete description of timing analysis, consult MAX+PLUS II Help.

Design Methodology

You can efficiently implement projects in the EPM5130 and other MAX 5000 EPLDs with hierarchical design entry. The EPM5130 EPLD supports both top-down and bottom-up design approaches for system-level functions. For example, you can create the serial coprocessor by first using the top-down method to draw a functional block diagram as shown in Figure 9. Each top-level functional block is partitioned into smaller blocks.

Figure 9. T1 Serial Coprocessor Block Diagram



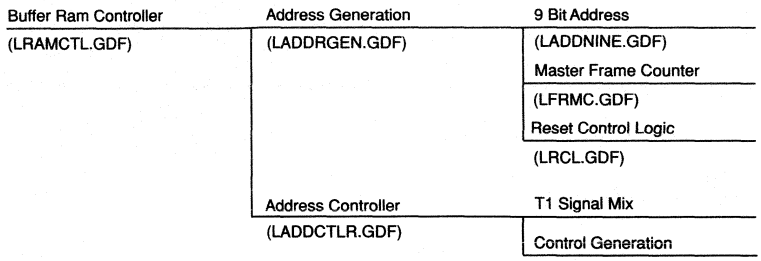
You can repeat this process until the design consists entirely of simple functional blocks.

These functional blocks are then logically implemented in a bottom-up approach. As each simple logic function is designed, MAX+PLUS II automatically creates a symbol to represent that function. MAX+PLUS II compiles and simulates each of these low-level functions independently then integrates the simple functions into more complex functions by connecting the symbols together to create the next level of hierarchy. The files created by integrating multiple low-level functions are similarly compiled and simulated before being integrated into the next level of the project. This bottom-up process is repeated until all functions are integrated to create the complete T1 serial coprocessor.

The design file for the buffer RAM controller block of the T1 serial coprocessor is implemented with the bottom-up method. Figure 10 shows the hierarchical definition of the buffer RAM controller called LRAMCTL.GDF.

LRAMCTL.GDF is composed of the address generator LADDRGEN.GDF and the address controller LADDCTLR.GDF. The lowest-level functions of LADDRGEN.GDF—a 9-bit address generator, master frame counter, and

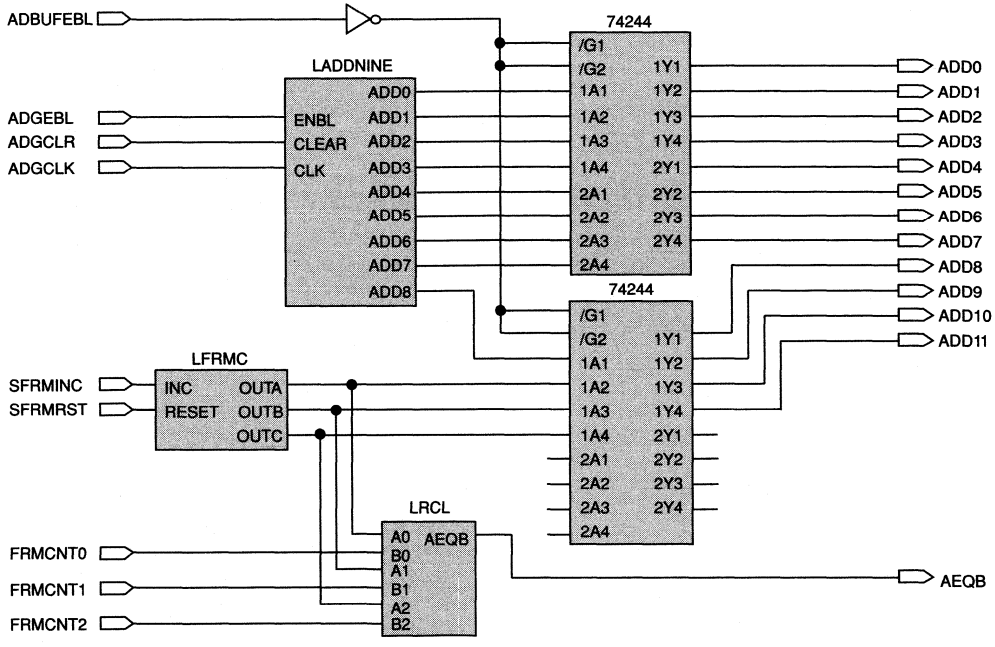
Figure 10. Hierarchical Definition of Buffer RAM Controller



Reset control logic—are implemented in the schematics LADDNINE.GDF, LFRMC.GDF, and LRCL.GDF, respectively. The automatically generated symbols for these files are connected to create the address generator implemented by LADDRGEN.GDF. See Figure 11.

Figure 11. Address Generator (LADDRGEN.GDF)

Functions such as the address generator for the buffer RAM can be individually compiled and simulated before being integrated into the next level of logic.



You can use MAX+PLUS II to compile and simulate both LADDRGEN.GDF and the address control generator LADDCTLR.GDF, which is also built from simple logic functions that were independently compiled and simulated. The symbols for these files can then be integrated into LRAMCTL.GDF, the buffer RAM controller shown in Figure 12. LRAMCTL.GDF can then be simulated and integrated into the top level of the serial coprocessor.

Figure 12. Buffer Ram Controller (LRAMCTL.GDF)

The buffer RAM controller is built out of two custom logic functions.

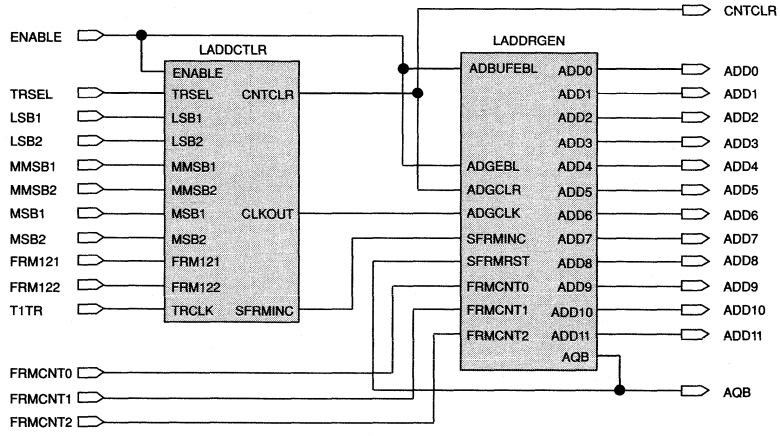
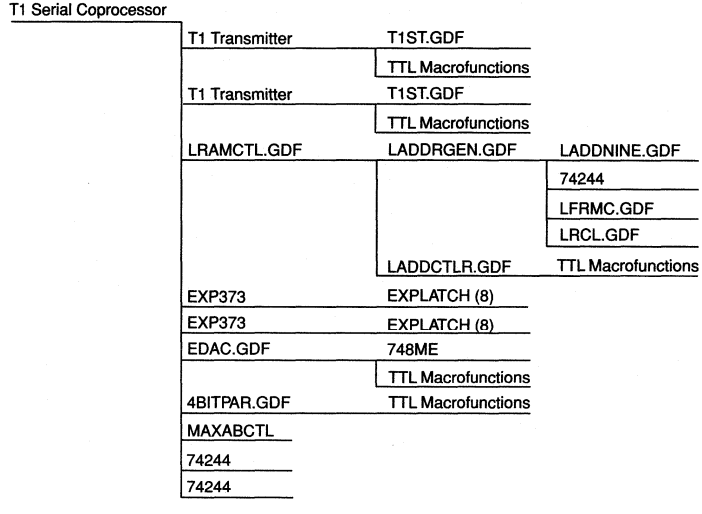


Figure 13 shows the hierarchical structure of the complete serial coprocessor project.

Figure 13. Hierarchical Design Definition of the T1 Serial Coprocessor



T1 Serial Coprocessor Example

The T1 serial coprocessor block diagram (shown in Figure 9) illustrates that a 4-bit parity code for error detection and correction is generated while the data bytes are written through the EPM5130 EPLD to RAM. The resulting 12 bits of data are stored in buffer RAM. The data for transmission is broken up into master frames. Each master frame consists of 12 frames, and each frame consists of 24 data bytes. The buffer RAM can hold 8 master frames at a time. The processor also writes 2 header command words for each master frame. After the system processor has transferred up to 8 master frames into buffer RAM, the processor writes the number of master frames into the master frame count register and issues a start command to the serial coprocessor.

The serial transmit controller is a state machine that controls the coprocessor's functions. Upon receiving the start command, the serial transmit controller takes control of the data paths from the system processor and reads the command words for the first master frame. Buffer RAM addresses are generated by buffer RAM control. The command words specify protocol information and the transmitter to be used for transmission. Once the command words are read, the data bytes of the first master frame are transferred from RAM to the specified T1 transmitter.

During the transfer from RAM to the T1 transmitter, the data passes through the EDAC circuitry, which uses a modified Hamming code. The data byte is then latched into the specified T1 serial transmitter, serialized, and transmitted. After a complete master frame has been transferred, the buffer RAM controller queues the next master frame so the next transfer can begin. This process continues until all frames have been transferred without processor intervention; then the control state machine issues an interrupt to the system processor, indicating that the transfer is complete and the next cycle can begin.

This section discusses the following T1 serial coprocessor features:

- Error detection and correction (EDAC)
- Buffer RAM control
- Serial transmit controller
- MFREG and CMD_LTCH
- T1 Transmitter

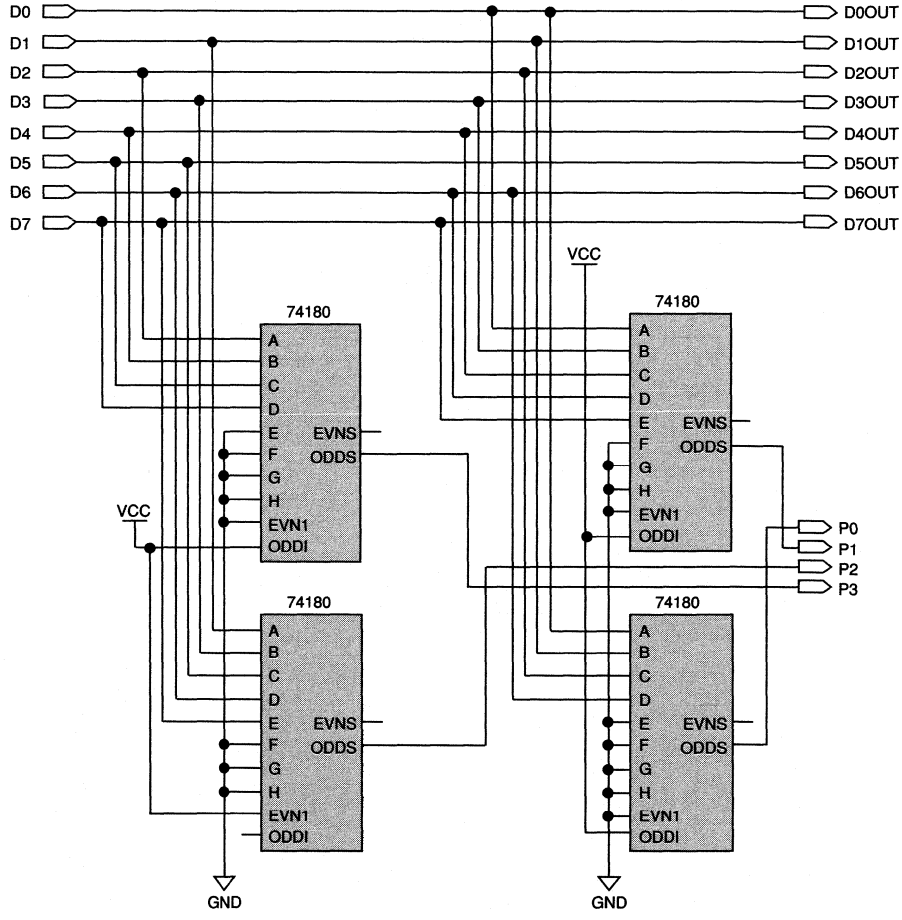
Error Detection and Correction (EDAC)

The EDAC circuitry consists of the 4-bit parity code generator and the EDAC function. The circuitry detects and corrects all single-bit errors and more than 75% of multiple-bit errors. (The multiple-bit error detection feature is not used in this sample design.)

The parity code generator 4BITPAR.GDF, shown in Figure 14, uses four 74180 parity generator macrofunctions to generate the 4-bit code. Each parity code bit is generated by a different set of data bits, and each data bit is used to generate two of the code bits. When the system processor writes a data byte to the transmit buffer RAM, the 4-bit parity code is automatically generated and written into the RAM with the data byte. The 4BITPAR function consumes four macrocells and replaces four TTL MSI components.

Figure 14. Four-Bit Parity Code Generator (4BITPAR)

4BITPAR generates error detection and correction codes. It consists of four 74180 macrofunctions from the MAX+PLUS II TTL MacroFunction Library.



EDAC.GDF, shown in Figure 2, uses the 4-bit parity code to detect and correct single-bit errors in the data word. The 74180 macrofunctions in EDAC.GDF generate a 4-bit code based on the eight data bits and the four parity bits. This 4-bit code is decoded by the 74154 macrofunction. The 74154 output is then used to correct any single-bit errors in the data byte. While the serial transmit controller is transferring data to one of the T1 transmitters, error detection and correction is performed automatically. EDAC consumes ten macrocells and replaces seven TTL packages.

Buffer RAM Control

LRAMCTL.GDF, shown in Figure 12, is the controller for the 4 Kbytes × 12-bit buffer RAM , replaces 8 TTL MSI packages, and consumes 15 macrocells. It consists of 2 functions: LADDCTLR and LADDRGEN. LADDCTLR multiplexes control signals from the T1 serial transmitters to select the signals from the active transmitter, which are then decoded to generate control signals for LADDRGEN. LADDRGEN generates the address ADD0 to ADD11 for the buffer RAM.

The Clear function for the ADD0 to ADD9 address register is implemented with complex combinatorial logic. A portion of the Report File for LRAMCTL.GDF shows the macrocells used for the address bit registers (see Figure 15). The Compiler's Logic Synthesizer module allocates the shareable expanders of the EPM5130 EPLD to implement the function. Both the total number of shareable expanders used by each macrocell and the number of shared expanders are shown in the Report File. Shared expanders are produced when the Logic Synthesizer scans the project for common product terms and places them on expanders, ensuring the most efficient design implementation. The ADD0 to ADD9 Clear function is implemented with ten expanders, which are shared by all macrocells associated with the function.

Figure 15. Report File Excerpt for LRAMCTL.GDF

The Report File shows macrocell fan-in and expander usage and sharing. Expander sharing helps ensure efficient use of device resources.

```
                                                C:\MAX2WORK\MAXAB\LRAMCTL.RPT
**OUTPUTS**

                                  Shareable Expanders      Fan-In
Pin   MCell   LAB   Primitive   Total   Shared   N/A    INP   FBK   Name

51      65     E       DFF       10       10      0      3    11    ADD0
52      66     E       DFF       10       10      0      3    13    ADD1
53      67     E       DFF       10       10      0      3    13    ADD2
54      68     E       DFF       10       10      0      3    14    ADD3
41      49     D       DFF       10       10      0      3    13    ADD4
42      50     D       DFF       10       10      0      3    15    ADD5
45      51     D       DFF       10       10      0      3    15    ADD6
46      52     D       DFF       11       10      0      3    16    ADD7
47      53     D       DFF       10       10      0      3    15    ADD8
```

The fan-in for each macrocell is also shown in the Report File. For example, ADD7 has a total fan-in of 19 signals, illustrating the MAX 5000 architecture's wide logic gating capability. Implementing such a function in a logic cell array (LCA) device would require multiple logic levels, resulting in severe performance degradation. The EPM5130 EPLD, on the other hand, can have any of 212 signals feeding into each macrocell.

Serial Transmit Controller

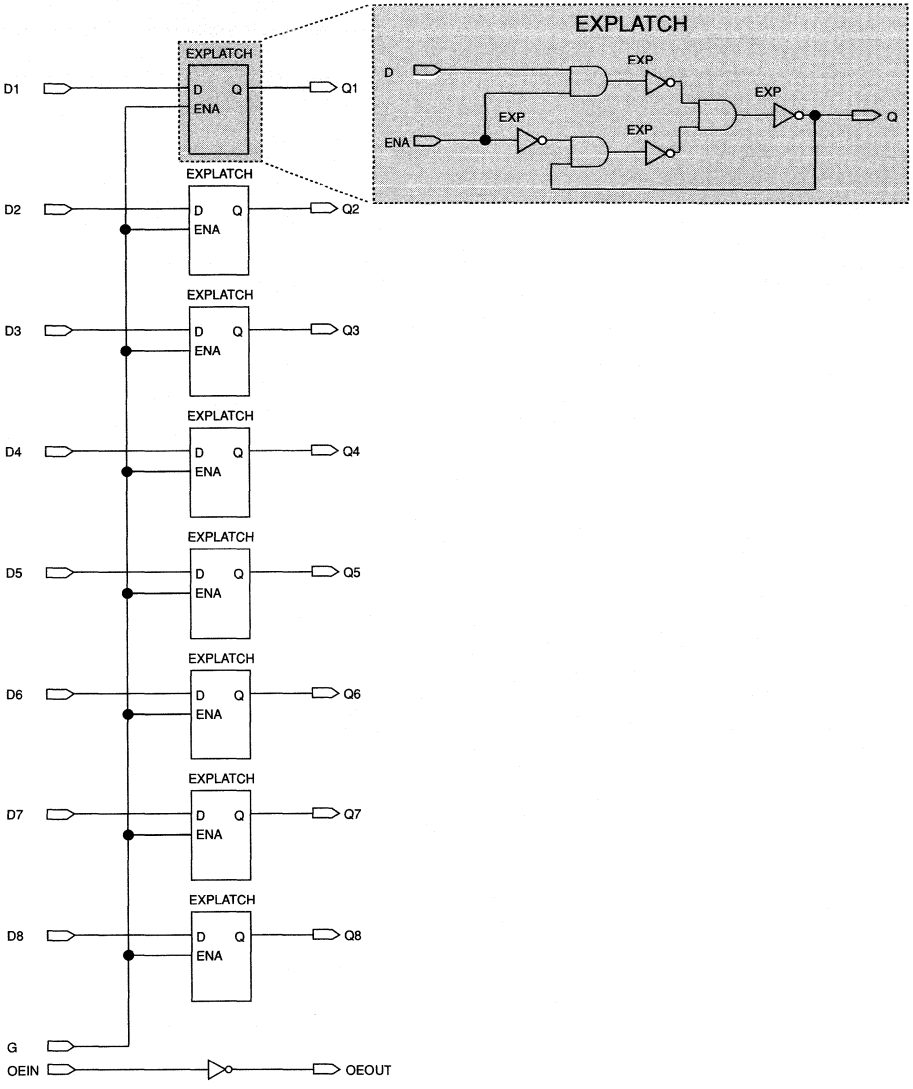
The serial transmit controller, MAXABCTL.TDF, is the controlling state machine for the T1 serial I/O subsystem. Once the system processor has written the number of master frames for transfer into the MFREG register, the outputs of MAXABCTL.TDF are used to control the subsystem. INTRO is the interrupt to the system processor, indicating that all master frames are transferred. A symbol is generated for MAXABCTL.TDF during compilation, then integrated into the top-level design. MAXABCTL.TDF uses nine macrocells and replaces four and a half 7474 macrofunctions.

MFREG and CMD_LTCH

MFREG and CMD_LTCH (shown in Figure 9) are latches used to control the subsystem. The system processor writes the number of master frames to be transmitted into MFREG. MAXABCTL then receives the signal SFRAME and the serial transmission cycle begins. MAXABCTL writes to CMD_LTCH, using one of the command words that are stored for each master frame. This word controls certain aspects of the T1 transmission cycle. Both MFREG and CMD_LTCH are implemented with EXP373 latches, which are input latches composed entirely of expanders (see Figure 16). The flexible MAX 5000 architecture allows cross-coupling of expander product terms to form individual latches such as EXPLATCH, which is also shown in Figure 16. EXP373.GDF consists of eight EXPLATCH latches. The EPM5130 EPLD can implement up to 128 SR latches on expander product terms or more than 80 flow-through latches without using a single macrocell. By implementing latches on expander product terms, macrocells are conserved for functions that require the full resources of the macrocell. The two EXP373 octal latches implement 11 latch bits by using 33 expanders and no macrocells.

Figure 16. Input Latch with EXPLATCH Macrofunctions (EXP373.GDF)

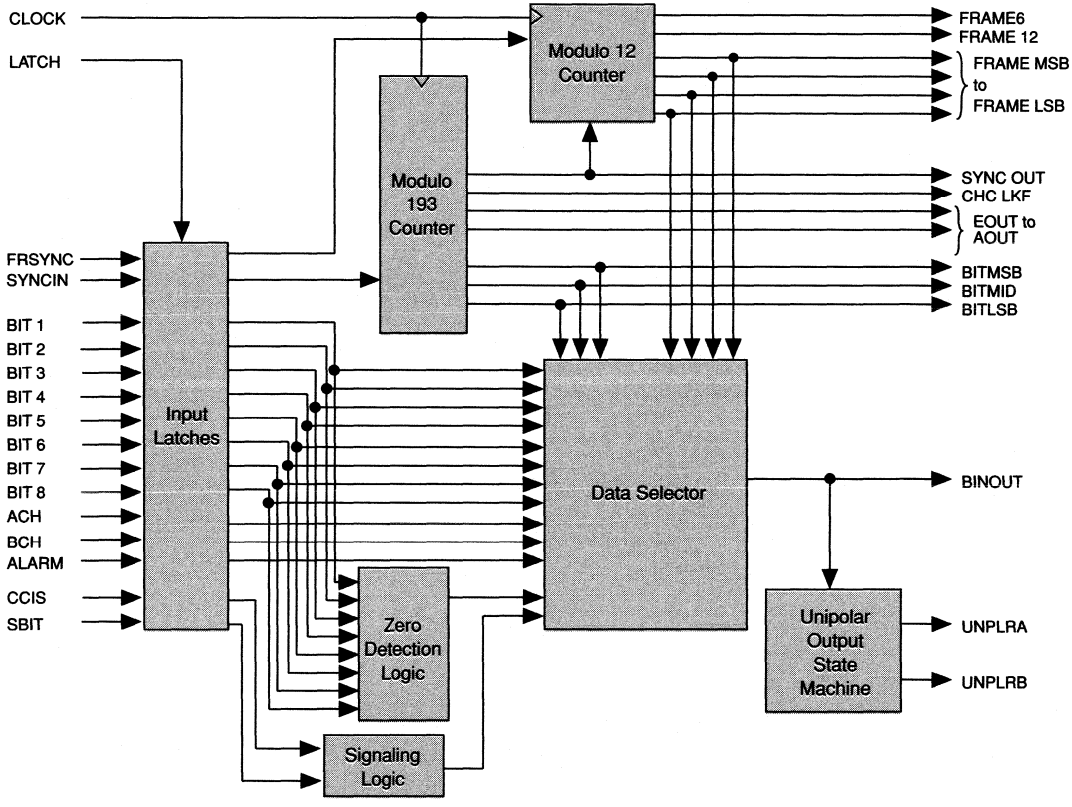
The EPM5130 EPLD can implement more than 80 D latches on expander product terms without consuming a single macrocell.



T1 Transmitter

Figure 17 shows a block diagram of the T1 transmitter. The T1 protocol uses pulse code modulation to time-multiplex 24 channels of data into a 1.544-MHz carrier frequency. The 1.544-MHz Clock is used to generate bit and channel timing via a modulo 193 counter. A separate modulo 12 counter is used for frame timing. Each of the transmitters consumes 39 macrocells and replaces 13 TTL MSI components.

Figure 17. T1 Serial Transmitter Block Diagram



All inputs to the transmitter are latched during the transmission of the eighth bit of each data byte. The data selector, controlled by the bit counter and the frame counter, provides the proper sequence of bits on signal BINOUT. The state machine T1ST uses BINOUT to provide unipolar outputs to create a single bipolar data transmission. BINOUT is a 12-product-term equation with a fan-in of 20 signals.

Table 1 shows the full complement of the TTL functions used in the T1 serial coprocessor. In this project, 53 TTL MSI functions were replaced, along with SSI gate functions such as AND, NAND, and OR gates of various widths. The number of packages required to implement these SSI functions is conservatively estimated at 22. This number is based on the width of the gates and ignores the fact that some of the functions, such as the wide NOR and AND gates, are not commercially available.

Table 1. TTL Functions Used in the T1 Serial Coprocessor

T1 Subsystem Component	TTL MSI or SSI Functions Used	Number of Functions Used per Subsystem
4BITPAR	74180	4
EDAC	74180	4
	7486	2
	74154	1
LRAMCTRL	74162	3
	74244	2
	74193	1
	7485	1
	74157	1
MAXABCTL	7474	4
Top-Level Schematic	74244	2
	74373	2
TITR (2 times)	7474	12
	74161	6
	74373	4
	74153	2
	74151	2
TTL Gate Logic	Inverter	25
	2-Input gate	32
	3-Input gate	14
	4-Input gate	8
	8-Input gate	3
Total functions used in T1 Serial Coprocessor = 75 total packages		

This project replaces 75 or more TTL packages—equivalent to an entire board of TTL logic. Implementing the T1 serial coprocessor in an EPM5130 can thus reduce the number of printed circuit boards used in a system. At the same time, system performance improves through the independent operation of the subsystem.

Furthermore, the entire process of functionally defining the serial coprocessor, entering the design, and simulating the logic at all levels, can take as little as one week. In contrast, laying out and routing tasks alone would take about the same time for a multi-layer PCB version of this I/O subsystem. You can also change project and logic programmed into the EPM5130 EPLD by simply erasing and reprogramming the device. Changes to the PCB version, on the other hand, usually require many cuts and jumpers and take considerably longer.

Conclusion

A single EPM5130 EPLD can integrate over 50 TTL MSI components plus a large amount of glue logic. The flexible EPM5130 EPLD architecture offers efficient implementation of both simple and complex logic functions, with true emulation of TTL devices. Integration is quick and efficient as a result of the chip architecture and the advanced design environment offered by MAX+PLUS II software.

DSP/Imaging Applications with the EPS448 SAM EPLD

April 1992, ver. 3

Application Note 19

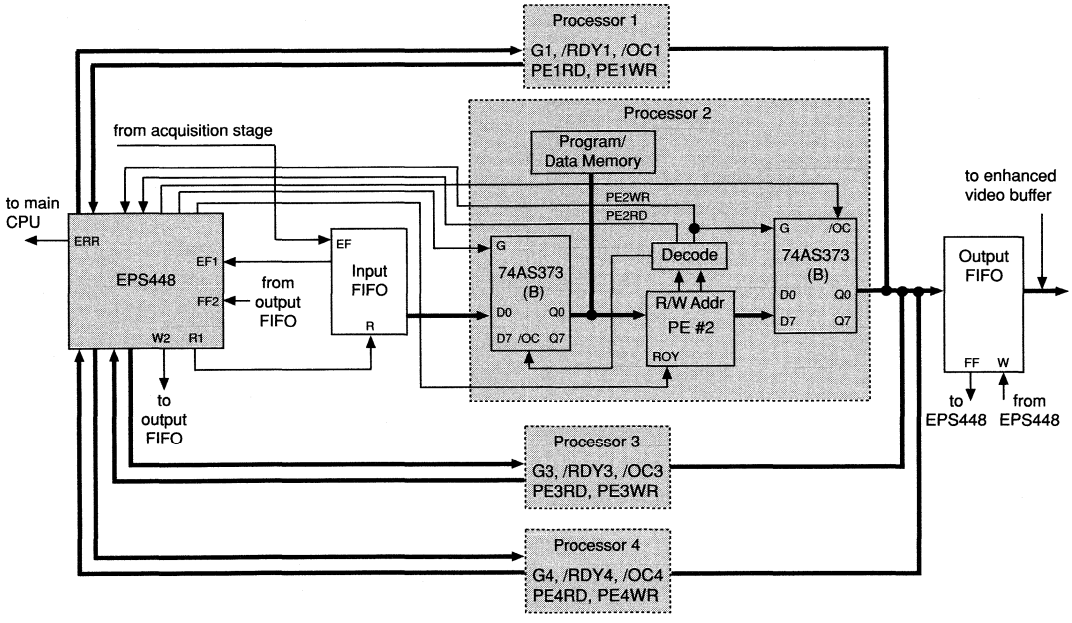
Introduction

Digital signal processing (DSP) and imaging systems encompass a wide range of applications, from satellite image processing and medical imaging to traditional areas such as radar, sonar, and acoustic image-processing. The goal of image processing is to transform the two-dimensional representation of an image into a more desirable form. However, use of the technology is often limited by the data throughput of the system. To improve data throughput, you can use Altera's EPS448 user-configurable Stand-Alone Microsequencer (SAM) EPLD at critical points in the system.

The EPS448 EPLD is an intelligent, programmable microsequencer that offers 30-MHz performance in 28-pin DIP and J-lead packages. It is ideal for use in DSP/imaging systems, since it provides a fast, compact, low-power control element in the datapath to off-load data housekeeping tasks from DSP processors. Figure 1 illustrates data flow controlled by an EPS448 EPLD for a typical DSP application.

Figure 1. Sample EPS448 Design

The EPS448 EPLD controls the reading of data into the input first in/first out (FIFO), writing of data into the output FIFO, FIFO status checks, and data flow for the four processor rows in this cell.



The EPS448 EPLD is a highly integrated coefficient generator for a variety of DSP algorithms. In addition, the device's exact timing capabilities make it popular for use in the video output stages of DSP/imaging systems. This application note covers the following topics:

- Imaging operations
- Techniques to improve performance
- Parallel data-flow control
- Read-data timing
- Write-data timing
- Data-flow microcode
- Other parallel processing control
- FIR filter implementation with SAM EPLDs

Familiarity with EPS448 architecture and performance is assumed. See the *EPS448 SAM EPLD Data Sheet* in the Altera 1992 ***Data Book*** for more information.

Imaging Operations

Figure 2 shows the four major analysis stages for DSP/imaging systems.

- Image formation
- Image enhancement and restoration
- Feature extraction
- Classification

Figure 2. Typical Image Analysis Stages

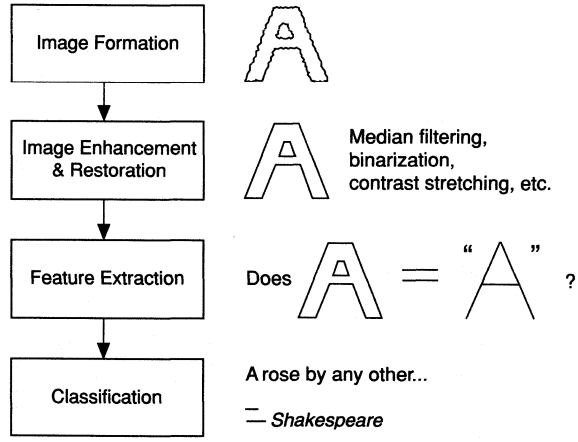


Image Formation

During image formation, each picture element (pixel) of an image is represented by a number. This number can, for example, represent the

absorption characteristic of body tissue for x-ray imaging, a temperature profile of a region for infrared imaging, or the brightness of objects in a particular cross-section for a picture taken by a camera. The sampling rate of the image must be high enough to maintain the useful information contained in the image, which is determined by the image bandwidth. Once a sufficient sampling rate is obtained, the sampled image must be quantized into some finite number of gray levels with an analog-to-digital conversion. Quantization is the process in which each pixel is assigned a value based on the average level of gray contained within its area.

Image Enhancement & Restoration

Image enhancement and restoration follow quantization. Image enhancement emphasizes particular features of an image by using a quantitative model built to enhance specific features. For instance, a model can use contrast stretching to stretch gray levels in a small region to occupy a larger region, or median filtering to replace each pixel with the median value of all pixels in its immediate area. These techniques emphasize particular image characteristics; they do not generate new information about the image.

Image restoration approximates an image that has known problems (e.g., blurring or poor contrast) and changes it through median filtering, binarization, and contrast stretching. Unlike image enhancement, image restoration is subjective and does not depend on specific quantitative analysis.

Feature Extraction

During feature extraction, pixels are grouped with other neighboring pixels to create small subwindows, which vary in size from 3×3 to 64×64 pixels. Real-time calculations are required to measure features such as edge density and variance and to create histograms. ("Real time" is the ability to digitize, process, and display signals at standard video rates without any perceptible delay from the DSP overhead.)

Classification

Feature-based image classification performs the real-time calculations. It needs a large amount of throughput that is based on the subwindow size. For example, a typical television quality image of 512×512 pixels and a frame rate of 30 frames per second has a sampling data rate of about 10 MHz. Therefore, the feature-extraction stage for a subwindow of 3×3 pixels requires a memory-access rate of 108 frames per second. Architectures that handle these sampling and memory-access rates and permit real-time image analysis are now available.

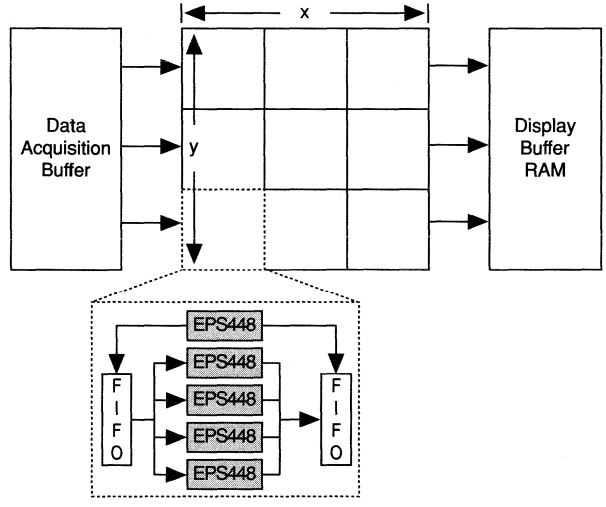
Techniques to Improve Performance

Real-time image analysis requires fast and efficient throughput, which you can achieve through the popular systolic approach. In a systolic system, a set of regularly interconnected cells is formed, with a dedicated processor element (PE) performing each task. Data is pipelined between cells and communicates with the outside world only at the boundary elements. Each stage of the pipeline has its own data-storage buffer and does not use global memory. This storage method allows the cascaded processors to store data between them, permitting new data to enter the pipeline while previous data is processed. The critical performance factors of a systolic system are the speed of the individual processors and the number of stages in the pipeline.

To increase system throughput with better real-time performance, you can expand the pipelined processors into a processor matrix by adding parallel processors (see Figure 3). This method allows similar data elements, such as blocks of pixels, to be processed synchronously, multiplying overall system throughput. While this method improves system performance, it complicates data management.

Figure 3. Processor Matrix

A systolic array consists of a number of pipelined and parallel processing cells. The EPS448 EPLD can be used to control the system array and elements within a particular cell.



The need for PE speed has led to advances in processors dedicated to DSP applications, such as the TMS320 family. Careful system data flow design is critical to system performance, particularly in a heavily pipelined system. Ideally, data should flow automatically through the system without involving the PEs. This method maximizes useful imaging cycles throughout the system by minimizing the housekeeping overhead on the PEs.

You can use both software and hardware to control data flow between processors. If data rates are slow enough, the software can poll status of the

processors to determine when they are ready to accept or generate data. However, this method severely hampers system performance and is being replaced by hardware interrupt-driven approaches, even in systems less dependent on data throughput. On the other hand, hardware interrupts require significant software overhead to call and service interrupt routines, and to restore the processor to its original task. This overhead rises to an unacceptable level in a multi-processor DSP system.

Performance improves significantly in a system that uses PEs exclusively for performing the necessary algorithms, without wasting time for handshaking. Until the advent of high-performance microsequencers, however, this approach was very costly because it required large amounts of hardware to synchronize the processors. Since EPS448 SAM EPLDs are microcoded, you can implement this technique with a sizable reduction in hardware and cost.

Parallel Data-Flow Control

To provide a "transparent" interface between processors, data buffers—e.g., RAMs, first in/first outs (FIFOs), and register files—are used to temporarily store the data output from one processor when subsequent processors are not ready to accept incoming data. These data buffers ensure optimum system performance. The parallel processor arrangement shown in Figure 3 consists of four processors within a cell that share common input and output FIFO data buffers. This structure can be expanded in both the x and y directions to yield proper performance for specific applications. You can use EPS448 EPLDs not only to control data flow for the pipelined and parallel rows, but also to control data flow within a particular cell.

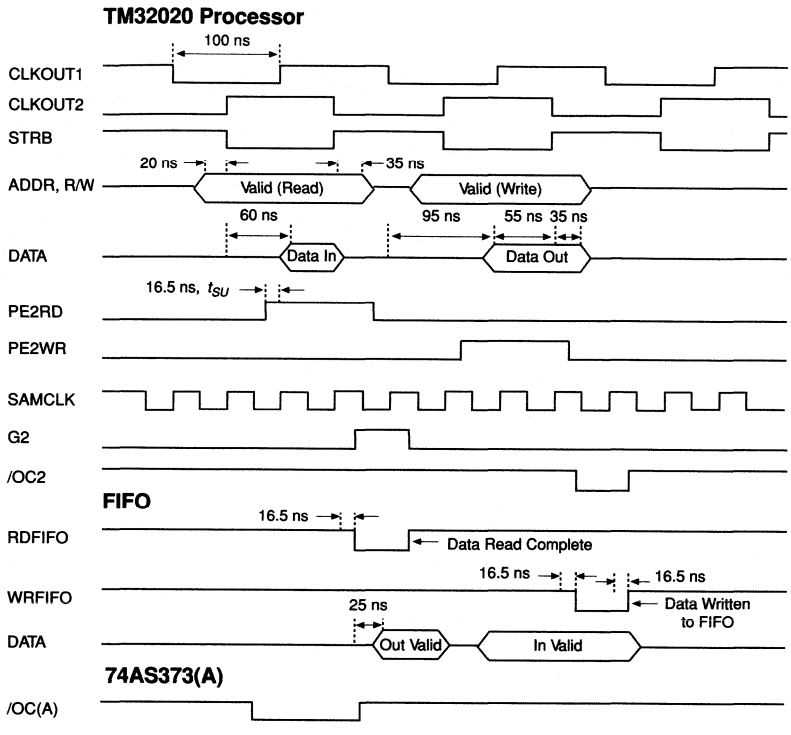
Figure 3 shows one cell in this large array. This cell contains an input FIFO, four PEs, an output FIFO, and an EPS448 microsequencer that controls the data flow. Each PE shares input and output FIFOs that absorb incoming and outgoing data. Within each FIFO, one 74AS373 device is connected to the input bus and one to the output bus of each PE to latch incoming and outgoing data.

The EPS448 EPLD performs error checking for the system and distributes data to the PEs by controlling the write and read functions of the 74AS373 devices. It simultaneously supports all data transfers including full error checking between the four PEs, thus off-loading this data housekeeping from the DSP processors.

The first task in handling this data flow is managing the FIFO read and write cycles. Figure 4 shows the timing diagrams that represent the FIFO processor and the EPS448 signals required for read and write cycles. The number 2 in PE2RD, PE2WR, G2, and /OC2 represents Processor 2. The widely used TMS32020 processor is shown, although any processor is suitable. This processor has two output Clocks, CLKOUT1 and CLKOUT2. Both output Clocks have 200-ns cycles, with CLKOUT2 lagging by 50 ns.

Figure 4. Data-Flow Timing

These signals are required to read data into the input FIFO and write data into the output FIFO.



Another output from Processor 2 is /STRB, which is synchronized with CLKOUT2. Its rising edge determines when a read or write is complete.

Other relevant processor outputs are the address and R/W (read/write) signals. These signals are issued 20 ns before /STRB goes low. They are decoded to generate PE2RD and PE2WR, which are input signals to the EPS448 EPLD, and indicate that a read or write has been requested.

Read-Data Timing

Processor 2 reads data in a two-phase cycle (shown in Figure 1). During the first phase, the processor reads the data on its input latch: the output from the 74AS373(A) is enabled, the processor reads this data, and the outputs are disabled. In the second phase, data is read from the FIFO to the 74AS373(A). The entire cycle begins when /STRB goes low. The /OC(A) (which is a decode of the processor's /STRB), R/W, and address signals activate at this time and enable the outputs of 74AS373(A). The 74AS373(A) remains enabled until /STRB goes high. At this point, Processor 2 reads the data of 74AS373(A) and /OC(A) is deasserted, i.e., 74AS373(A) is disabled.

Once Processor 2 has read the data, the EPS448 EPLD must refill the 74AS373(A). The EPLD inititates the read from the FIFO into 74AS373(A)

when it detects PE2RD on the rising edge of the Clock to the EPS448 device (SAMCLK), 50 ns after /STRB goes low. The EPS448 EPLD waits for one Clock cycle before it drives /RFIFO low. /RFIFO requests the FIFO to output a word. At this time, G, the input enable to the 74AS373(A), is also activated by the EPS448 EPLD. /RFIFO and G are actually asserted on the EPS448 pins 16.5 ns (Clock-to-output delay) after SAMCLK. If the FIFO has an access time of 25 ns, the output data of the FIFO appears at the inputs of the 74AS373(A), 41.5 ns (16.5 + 25 ns) after PE2RD is detected. For 50 ns, the /RFIFO pulse is low. Data is valid at 74AS373(A) for 25 ns (50 ns /RFIFO low, 25 ns FIFO access time), meeting the setup time of the latch. On the rising edge of /RFIFO, the data is latched into 74AS373(A).

This process requires two SAMCLK signals (one detects PE2RD and one asserts /RFIFO to the FIFO), and half the 200-ns processor cycle time. If the FIFO is empty when the read is requested, i.e., if EF1 is high, the EPS448 EPLD deactivates the processor's RDY signal and informs the system processor of the error status. In a real system, the EPS448 EPLD may have a set of routines to allow the PE to recover dynamically from the error. The EPS448 EPLD provides up to 448 microcoded states to support this complex error recovery.

Write-Data Timing

PE2WR becomes active when Processor 2 issues a write cycle. Figure 4 shows that after PE2WR goes high, the inputs of 74AS373(B) are enabled since G is tied to PE2WR, and the latch is ready to accept a new data word. To write data into the output FIFO, /OC(B) and /WFIFO appear 16.5 ns (one Clock-to-output delay) after the Clock following the detection of PE2WR. Processor 2 then takes control of the FIFO data bus and begins the FIFO read. After 50 ns, the /OC(B) and /WFIFO are deasserted (i.e., go high) and data from the 74AS373(B) is written into the FIFO.

The write process also requires two SAMCLK signals: one detects PE2WR and one asserts /W to the FIFO. If the FIFO is full, i.e., FF2 is high, at the time of the write request, the EPS448 EPLD deactivates RDY to the processor and informs the system processor of the error status.

Data-Flow Microcode

EPS448 subsystem control tasks include checking the processors for read and write requests, servicing these requests, and checking for FIFO error conditions. You can use the SAM Assembly Language (ASM) to enter EPS448 designs (see Figure 5). ASM consists of simple IF constructs. Output specifications for any particular state are enclosed in brackets, and appear in the same order in which they are placed in the Outputs Section, e.g., the first output in brackets represents /RFIFO. You can use macros to substitute text for commonly used output strings. For instance, IDLE substitutes for G = 0 (inactive), RDY = 1 (active), and /OC = 1 (inactive). Data-flow control starts at the label PE1SRVC.

The starting point for data-flow control in PE1SRVC is the same as that of the PE2SRVC subroutine used to detail processor read and write cycles.

Figure 5. Data-Flow Microcode for EPS448 Design

```
OUTPUTS:  /RFIFO, /WFIFO, G1, RDY1, /OC1, G2, RDY2, /OC2, ..., ERR

MACROS:   IDLE = 011                                  % Similar to PE2SRVC          %

PE2SRVC:  IF EF1 + FF2 THEN [11 001 001 ... 1]
          GOTO SYSERR;                                % Processor not ready          %
             ELSEIF PE2RD THEN [11 IDLE IDLE ... 0]
          GOTO RDPE2;                                 % Ready to read               %
             ELSEIF PE2WR THEN [11 IDLE IDLE ... 0]
          GOTO WRPE2;                                 % Ready to write              %
          ELSE [11 001 001 ... 0] GOTO SYNCH;         % Error, need to synchronize  %

RDPE2:    [01 IDLE 111 ... 0] GOTO PE3SRVC;           % Read from input FIFO        %
WRPE2:    [10 IDLE 010 ... 0] GOTO PE3SRVC;           % Write to output FIFO        %

PE3SRVC:                                              % Similar to PE2SRVC          %
PE4SRVC:                                              % Similar to PE2SRVC          %

SYSERR:                                               % Recovery, wait for error
                                                        conditions to clear       %
SYNCH:                                                % Stop processors,
                                                        synch subsystem           %
```

One of four different states becomes active when PE2SRVC is called. This label is reached when Processor 1 has been serviced.

If the input FIFO is empty or the output FIFO is full (EF1 + FF2) at the label PE2SRVC, the first state becomes active, which is an error condition. When this error occurs, Processor 2 RDY signals are forced low, and a transition to SYSERR occurs. You can avoid this system error by choosing the FIFO depths according to the worst-case delays. If either error occurs (e.g., input FIFO is empty or output FIFO is full), you can customize the SYSERR subroutine for a particular system. SYSERR can inform the main CPU of the subsystem error and clear the error status of the EPS448 EPLD.

In the second state, if PE2RD is true, all outputs are static in anticipation of the required read cycle, assuming no errors occur at the label PE2SRVC. The label RDPE2 becomes active on the next SAMCLK, when /RFIFO and G2 become active. This state causes a read from the FIFO to 74AS373(A). On the next SAMCLK, a transition to PE3SRVC occurs when Processor 3 is serviced and all outputs relating to Processor 2 are deactivated.

The third state occurs if PE2WR is true at the PE2SRVC label, which prompts the beginning of a write cycle. All outputs remain static until SAMCLK causes a transition to the only state at the WRPE2 label. This state's outputs force /WFIFO and /OC low, allowing the processor to write data to the output FIFO. On the following SAMCLK, a transition to PE3SRVC occurs when Processor 3 is serviced and all outputs relating to Processor 2 are deactivated.

In the fourth state, neither error condition is true and both PE2RD and PE2WR are inactive. This error informs you that the processors are not synchronized. The subsystem is designed so that Processor 2 lags behind Processor 1 by 100 ns, which is the same amount of time needed to read or write from Processor 1. In addition, it is assumed that all processors in the image-processing system execute instructions requiring the same amount of time. If Processor 1 has just finished reading data, Processor 2 should also be ready to read data when PE2SRVC is reached. If Processor 2 is not ready when Processor 1 finishes, a synchronization error has occurred, and the RDY signals of all processors are pulled low. You must then create a subroutine that causes a transition to SYNCH to resynchronize the subsystem.

This EPS448 subsystem increases data throughput and ensures consistent data flow from the input stage, through the processors, to the output FIFO. The microsequencer provides data-flow control, while relieving the processors of synchronization and FIFO error-condition checking. The EPS448 EPLD runs at 20 MHz to allow 4 processor reads and 4 processor writes to occur continually every 800 ns, generating an output data word every 200 ns. (This is the same amount of time required by a processor to perform one instruction.) Without the microsequencer, each PE would require 2 additional cycles of overhead between the generation of each data word to check for data at the input and output FIFOs. A 30-MHz EPS448 EPLD is also available to support faster processors and subsystems that have more than 4 processors for each microsequencer.

Other Parallel Processing Control

Parallel approaches that require "local neighborhood" operations are also used in spatial operation techniques for image enhancement. Calculations are performed on a pixel and its nearest neighbors. For example, if a 5×5 pixel area of an image is to be analyzed, a 5×5 array of processors would ideally perform the algorithm. The EPS448 EPLD controls the loading of the pixel words into the processors. Five bits from the image's first line are loaded into the top row of processors. The EPS448 counts through the unnecessary pixels in the line following the adjoining pixels, the blanking pixels beyond the edge of the screen, and the unnecessary pixels preceding the active ones in the second line. The EPS448 then enables the five pixels on the second line into the second row of processors, and the process continues. The EPS448 EPLD is well suited to this application because it has an on-chip counter and performs subroutine calls and looping.

FIR Filter Implementation with SAM EPLDs

The finite-duration impulse response (FIR) filter is a basic building block in DSP/imaging applications. The EPS448 EPLD is ideal for creating an *n*-tap FIR filter, and for using a multiplier-accumulator (MAC) for coefficient generation. See Figure 6. An 8-tap FIR filter requires 8 data words and 8 coefficients as inputs to a MAC. The EPS448 EPLD can supply coefficients to a MAC as fast as 30 MHz, enabling the device to act as a fast PROM in delivering coefficients to the input registers of the MAC. The EPS448 EPLD also replaces the address counter and provides necessary control lines.

Figure 6. Selectable Coefficient Generation

The EPS448 acts as a large EPROM plus address logic for storing coefficient values. A selectable 8-tap, 12-bit filter can be implemented with a 12 × 12-bit MAC.

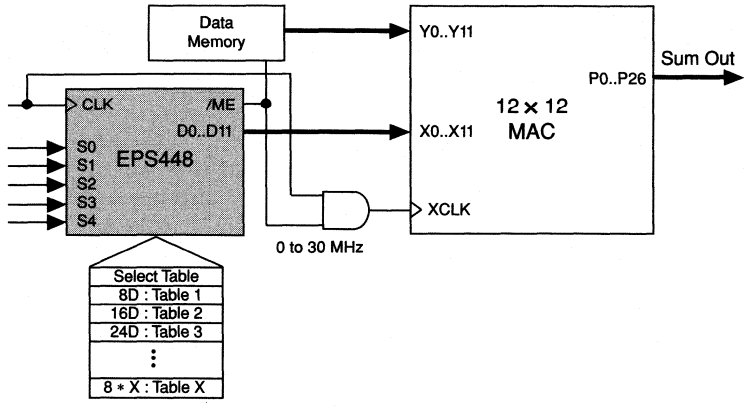
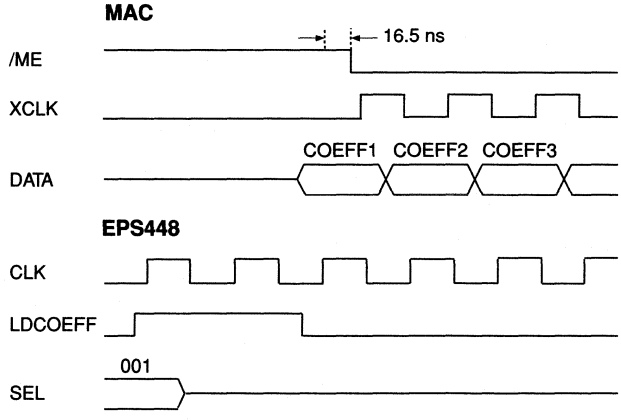


Figure 7 shows the signals necessary to interface the EPS448 to a 12 × 12 bit MAC that has 35-ns multiply-accumulate times.

Figure 7. Coefficient Generator Timing

The figure shows the timing and signals required for the 8-bit tap filter. SEL and LDCOEFF are the necessary control signals into the EPS448.



The EPS448 EPLD has 448 microcode addresses, each containing a 36-bit word that consists of 20 internal control bits and 16 output bits. As a result, the EPLD contains a 448 × 36 PROM and a stack that can implement a 448-way branch in only 2 Clock cycles. The EPS448 EPLD can thus store and quickly access coefficient tables of varying sizes to implement multiple algorithms.

Figure 8 shows ASM code that allows the EPS448 EPLD to provide selectable coefficient strings to implement 8-tap filters. If `LDCOEFF` from the first

Figure 8. EPS448 Microcode for Coefficient Generator

```
INPUTS:    S4, S3, S2, S1, S0, LDCOEFF          % Selects are 3rd, 4th, & 5th for mask   %

OUTPUTS:   COEFF11 ... COEFF0, /ME              % 12 coefficients and memory enable      %

SELECT:       IF LDCOEFF [0 ... 0 1]            % Select table                           %
              ANDPUSHI 11111000 GOTO TABLESEL;
              ELSE [0 ... 0 1]  GOTO SELECT;    % Wait for LDCOEFF                       %

TABLESEL:  [0 ... 0 1] RETURN;                  % Transition to selected table           %

8D:        [H4C 0] CONTINUE;                    % Hex coefficients, /ME                   %
           [OB5 0] CONTINUE;
           [679 0] CONTINUE;
           [D40 0] CONTINUE;
           [51A 0] CONTINUE;
           [D41 0] CONTINUE;
           [A09 0] CONTINUE;
           [624 0] CONTINUE;
16D:       [904 0] CONTINUE;
           [4B8 0] CONTINUE;
```

instruction (IF LDCOEFF ANDPUSHI 11111000) is true, the EPS448 inputs are ANDed with the constant 11111000, and the result is pushed onto the EPS448 internal stack. This process guarantees that the three least significant bits will be zeros. The remaining five input bits are the select inputs that determine which coefficient table to access.

After the ANDPUSHI, the sequencer is instructed to RETURN. This instruction pops the top-of-stack, making it the next label (i.e., address location) in the EPS448. For instance, if the select inputs are 00001, address 00001000 * 11111000 = 8D (decimal) is off the stack and becomes the next label; if the select inputs are 00010, address 16D is the next label. In this example, the 11111000 mask forces a branch to an address that is a multiple of 8 (the number of taps). If the number of taps is changed, the mask is also modified. For example, if a 12-tap filter is required, the mask is changed to 11110100. For this 8-tap filter, and select inputs 00001, a transition occurs after the RETURN command to address 8D on the Clock. (Address 8D is the address of the first coefficient table.)

After the transition to address 8D, the first coefficient and memory enable /ME are output. /ME enables the data memory and is ANDed with the system Clock to generate XCLK, which clocks in the *x* data bits. XCLK requires a 15-ns data setup time and a 3-ns hold time. Since the coefficient outputs arrive 15 ns before XCLK, this condition is met. Hereafter, a new 12-bit coefficient is loaded into the *x* data input registers of the MAC on every system Clock.

When the last coefficient is supplied to the MAC, the FIR filter is implemented, and the EPS448 EPLD issues a flag to the data source to

inform it that data should no longer be enabled. At this point, a transition to SELECT occurs, providing access to the appropriate location (depending on the inputs) in the coefficient table. SELECT may access the same set of coefficients or any other set in the table. Because the EPS448 EPLD has 448 memory locations, 56 possible table locations can be accessed for this 8-tap filter.

To choose the appropriate set of coefficients and to meet the setup time for XCLK, a 3-Clock cycle latency occurs whether there are 56 eight-tap filters, equal to 448 states, or 224 two-tap filters. This latency is possible because the EPS448 EPLD performs a 448-way branch in only 2 Clock cycles. After the third Clock cycle, coefficient data becomes available every Clock cycle, or as fast as 30 MHz. This approach has advantages over both FIFO and PROM approaches. Although data can be written directly if FIFOs are used, very fast FIFOs, with access times of 35 ns, are very expensive.

You must also reload coefficients before they are reused—which requires *n* Clock cycles, where *n* is the number of taps—or you must use a retransmit feature. Retransmit slows the cycle time of a 35-ns FIFO to 120 ns.

The EPS448 EPLD provides advantages over PROMs, including power savings, cost reduction, and greater integration. The EPS448 device does not need the address counters required with PROMs. Also, if you use multiple PROMs, the PROM address counter must be buffered to account for filter length or width. Using multiple PROMs consumes more board area because of the additional devices and the routing of address lines. The EPS448 provides the benefit of higher integration while typically consuming only 90 mA at 30 MHz.

Conclusion

The programmable EPS448 architecture is ideal for implementing complex designs typical of DSP/imaging systems. Because of its flexibility and high performance, the EPS448 EPLD is well suited for controlling data flow for both parallel and pipelined systems, and for coefficient generation. Additionally, the EPS448 EPLD provides both low-power operation and high performance (30 MHz).

Fast Bus Controllers with the EPM5016 MAX EPLD

April 1992, ver. 3

Application Note 20

Introduction

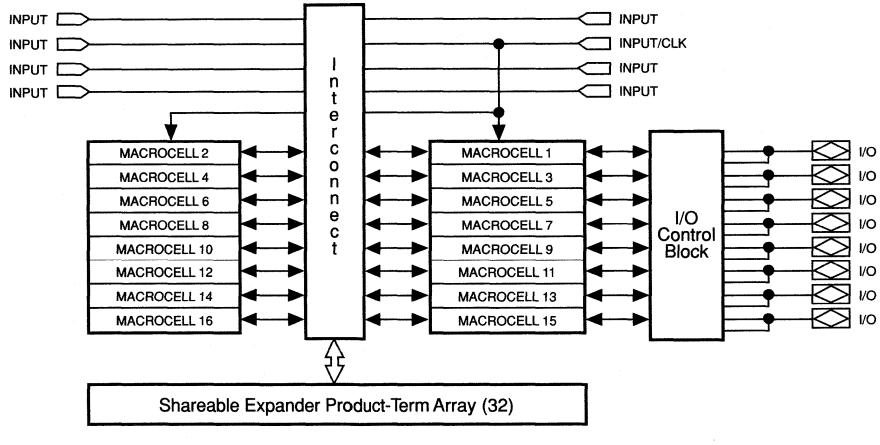
Today's advanced microprocessors can operate in systems with Clock speeds of 33 MHz and greater. To realize the full potential of these microprocessors, their memory interfaces must be equally fast. However, high-performance memory devices are expensive. You can improve the cost/performance tradeoff with a combination of fast and slow memories. Wait states are added into the microprocessor bus cycle to accommodate the slower memories.

This application note describes how to integrate the wait-state and bus-control logic into an Altera EPM5016 MAX EPLD, which is then integrated into an 80386 microsystem design. It also explains how to create and process the design with MAX+PLUS II software.

EPM5016 Overview

The EPM5016 EPLD has propagation delays of 15 ns, system Clock speeds of 66 MHz, and counter frequencies of 100 MHz. It also has 24-mA I/O drivers that enable it to directly connect to buses. Available in a windowed ceramic dual in-line package (DIP), or plastic one-time-programmable (OTP) DIP, plastic J-lead chip carrier (PLCC), and 300-mil small-outline integrated circuit (SOIC) packages, the EPM5016 accommodates designs with up to 15 inputs and 8 outputs. The EPM5016 architecture is based on a single flexible Logic Array Block (LAB) that includes 3 components: the macrocell array, the shareable expander product-term array, and the I/O control block. See Figure 1.

Figure 1. EPM5016 EPLD Block Diagram

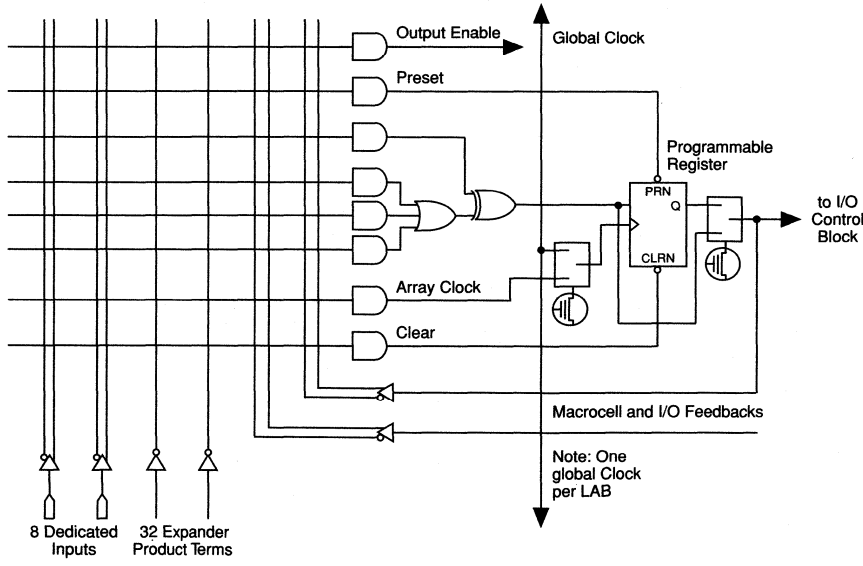


The macrocell array contains 16 macrocells. Each macrocell has a programmable-AND/fixed-OR array and a configurable register that provides D, T, JK, SR, or flow-through latch operation with independent programmable Clock options (see Figure 2). Each macrocell also contains 4 product terms for logic implementation. If necessary, the shareable expander product-term array can supply up to 32 additional product terms. Each shareable expander can be used and shared by all macrocells.

The EPM5016 EPLD has eight I/O pins with 24-mA output drivers to allow direct interfacing to a variety of system buses. All I/O pins are individually configurable for dedicated input, dedicated output, or bidirectional operation. Each pin has a dedicated feedback and a tri-state buffer. Macrocells and I/O pins have separate feedbacks—a feature called "dual feedback"—that enable you to bury macrocell logic and retain the pins for inputs. For complete details about EPM5016 architecture and timing, see the *EPM5016 to EPM5192 EPLDs: High-Speed, High-Density MAX 5000 Devices Data Sheet* in the Altera 1992 ***Data Book***.

You create and program EPM5016 designs with Altera's MAX+PLUS II development system. MAX+PLUS II is a complete CAE package that offers hierarchical graphic, text, and waveform design entry; automatic design compilation and fitting; functional and timing simulation; full timing analysis; and device programming. The MAX+PLUS II Compiler features advanced logic synthesis algorithms that ensure the most efficient use of EPLD resources. The combination of flexible EPLD architecture and

Figure 2. EPM5016 Macrocell



advanced CAE tools ensures rapid design cycles, allowing a design to go from conception to completion in a single day.

Bus Controller Functional Description

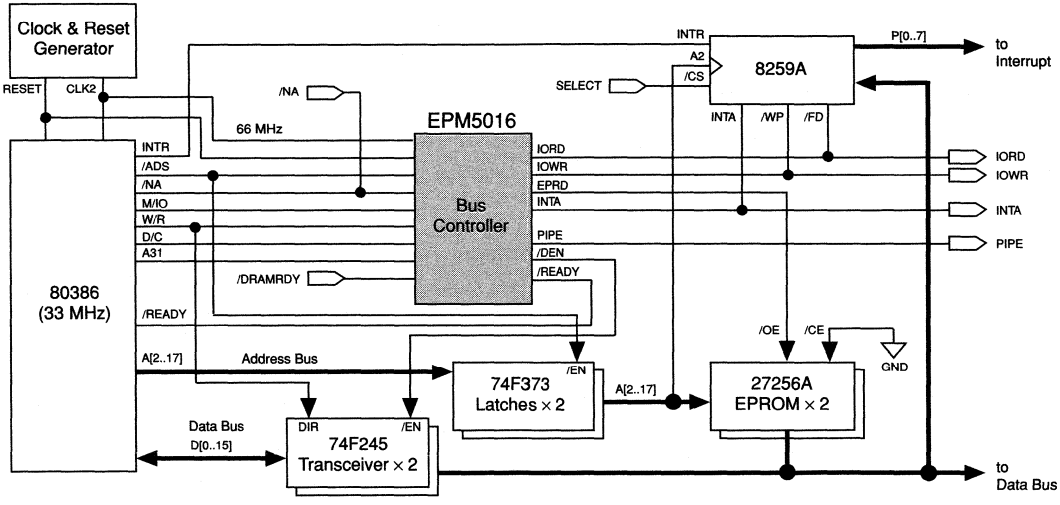
Figure 3 shows the block diagram of an 80386 microsystem that incorporates peripheral logic, memory, and an 8259A interrupt controller device. The EPM5016 EPLD serves as the system bus controller and decodes the 80386 status signals to control the peripheral logic (i.e., data transceiver), interrupt controller, and other external logic. It also extends bus cycles by adding wait states to interface to slower peripherals and memory devices.

The 80386 halts processing to allow wait states to be added into the bus cycle when the signal /READY is high. The slash (/) indicates an active-low signal. The EPM5016 bus controller tracks each bus cycle operation and causes /READY to go high when wait states are needed. For example, read operations from 200-ns EPROM memory in 33-MHz systems require 14 wait states.

The EPM5016 EPLD also decodes the bus-control signals IORD (I/O read), IOWR (I/O write), and INTA (interrupt acknowledge). The 24-mA output drivers on the EPM5016 EPLD eliminate the need to buffer these bus signals externally.

The 16 data signals originating from the 80386 are isolated from the system data bus with two 74F245 8-bit transceivers. The tri-state control on the transceivers is provided by the signal /DEN from the EPM5016. The direction signal is controlled directly by the read/write signal (W/R).

Figure 3. 80386 Subsystem Block Diagram



Two 74373 devices (8-bit latches) latch the 80386 address signals at the beginning of the bus cycle to maintain a valid address throughout the cycle. The latches are controlled by the 80386 signal /ADS. The high performance of the EPM5016 EPLD easily supports the 33-MHz bus cycles. /ADS can be connected directly to the address latches since the EPM5016 control signals for the peripheral logic are active before the end of the first bus cycle; thus a 66-MHz (2 × 33-MHz) Clock can be used to clock the design. In addition, the wait-state generator offers greater granularity with 15-ns cycles than with 30-ns cycles.

Bus Controller Interface Signals

The design for the bus controller circuit requires 9 inputs: all 8 dedicated EPM5016 inputs and a single I/O pin. Five of the 9 inputs to the EPM5016 EPLD are signals from the 80386 microprocessor. Table 1 shows the functions of these 5 signals. The other inputs are described below.

Table 1. EPM5016 Input Functions

Input	Function
M/IO	Memory or I/O
W/R	Read or write status
D/C	Data or control status
A31	Address bit 31 for memory mapping of the EPROM
/ADS	Address data strobe indicating the beginning of the bus cycle

Systems with functions set up for pipelining require external logic to generate the /NA signal (next address). /NA feeds both the 80386 and the EPM5016 bus controller. When /NA is activated, the 80386 places the next address on the address signals so that they can be latched. Applications that require pipelining receive minimal benefit from wait states. In such cases, the /NA signal is used simply to disable the EPM5016 EPLD.

/DRAMRDY is an externally generated signal that, when high, halts the 80386 by causing the /READY signal to go high. When /DRAMRDY goes low, the 80386 continues processing.

The 66-MHz Clock (CLK2) within the EPM5016 EPLD feeds a toggle flipflop to generate a divide-by-two signal (CLK) that matches the 33-MHz system Clock signal. CLK tracks the microprocessor Clock phase.

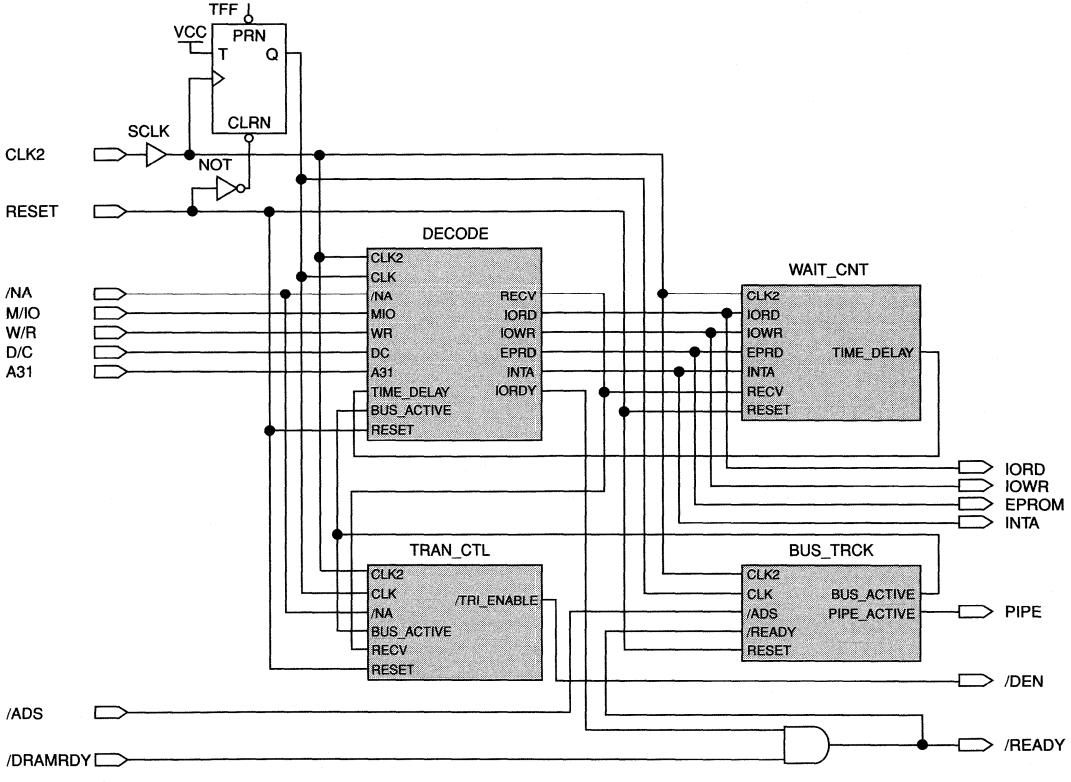
RESET is connected directly to the microprocessor's Reset signal. RESET feeds the Preset or Reset input of each register to set the EPM5016 EPLD to the correct start-up state.

Designing with MAX+PLUS II

Figure 4 shows the BUS_CNTL.GDF, the schematic for the EPM5016 bus controller. You can create logic designs (called "projects" in MAX+PLUS II) with up to eight levels of hierarchy. MAX+PLUS II supports graphic, text, and waveform design entry methods. In this project, the four symbols BUS_TRCK, DECODE, WAIT_CNT, and TRAN_CTL represent Text Design Files (.TDF) created in the Altera Hardware Description Language (AHDL). AHDL allows you to describe logic with behavioral descriptions such as state machines, arithmetic functions, comparator functions, truth tables, and Boolean equations.

BUS_CNTL.GDF, the project's top-level design file, is created with the MAX+PLUS II Graphic Editor. When each TDF is compiled, the four symbols are automatically created. Inputs to the text files are represented as pinstubs on the left side of the symbol; outputs are on the right. In this project, the BUS_TRCK, TRAN_CTL, and DECODE functions are state machines. WAIT_CNT is a counter with synchronous Preload and Enable.

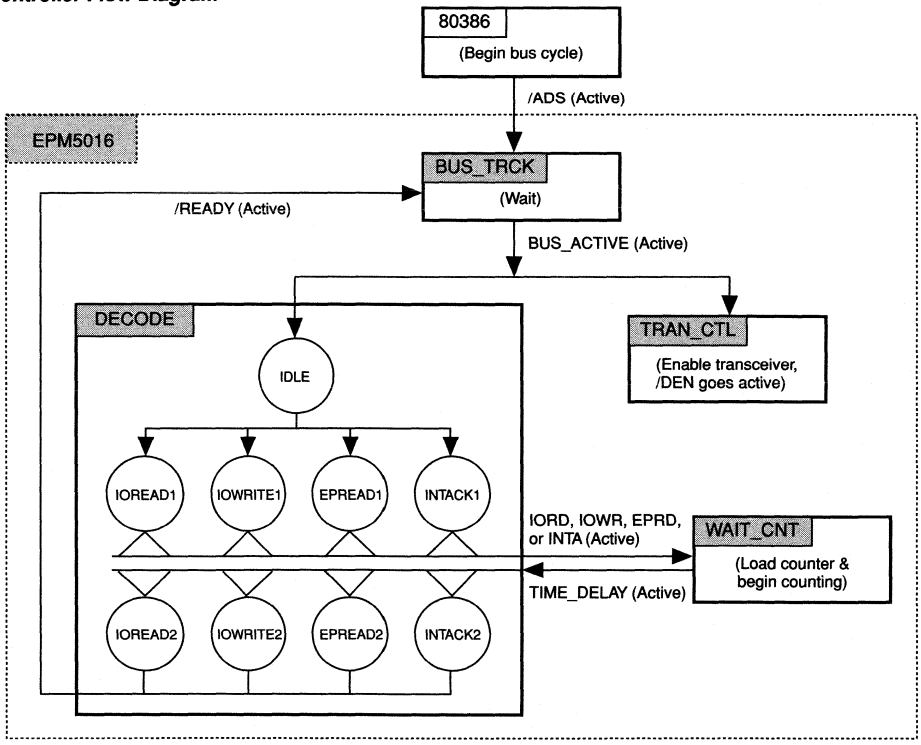
Figure 4. Bus Controller (BUS_CNTL.GDF)



Design Description

Figure 5 shows the flow diagram for the bus controller. First, the 80386 causes /ADS to go low, indicating that a bus cycle has begun. /ADS then causes BUS_TRCK to activate the signal BUS_ACTIVE, which indicates that the processor is in an active bus cycle. BUS_ACTIVE feeds the functions DECODE and TRAN_CTL. The TRAN_CTL function then scans the 80386 control signals and enables the data transceiver buffers when they are required. DECODE also scans the 80386 control signals and decodes them into specific control signals (IORD, IOWR, EPRD, and INTA) that drive the peripheral logic and the block function, WAIT_CNT. WAIT_CNT is a loadable 4-bit counter that counts the required number of wait states for bus cycles. When WAIT_CNT finishes the wait state count, it causes time_delay to go low, enabling DECODE to release the /READY signal and finish the bus cycle.

Figure 5. Bus Controller Flow Diagram



BUS_TRCK.TDF

The bus tracker function, BUS_TRCK, is a state machine that determines when the 80386 is in a bus cycle. Figure 6 shows the state diagram for BUS_TRCK.

Figure 6. BUS_TRCK State Diagram

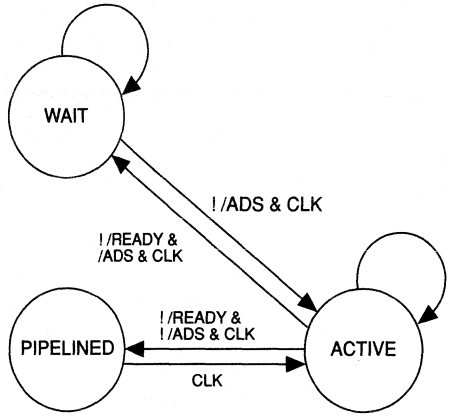


Figure 7 shows the TDF for BUS_TRCK, which has two output signals: bus_active and pipe_active (only bus_active is used in the top level of the project). The bus_active signal is active-low, indicating that the 80386 is executing a non-pipelined bus cycle.

Figure 7. BUS_TRCK.TDF (Part 1 of 2)

```
SUBDESIGN bus_trck
(
    clk2        : INPUT;    % 80386 2x Clock (66 MHz)              %
    clk         : INPUT;    % 80386 33-MHz Clock                   %
    /ads        : INPUT;    % Low to begin bus cycles              %
    /ready      : INPUT;    % Low to end bus cycles                %
    reset       : INPUT;    % High to reset                        %
    bus_active  : OUTPUT;   % Low during active bus cycles         %
    pipe_active : OUTPUT;   % Low after pipelined bus cycles       %
)
VARIABLE
    bus_cycle : MACHINE OF BITS (bus[1..0])  % Define state bits  %
                WITH STATES ( wait,          % and state names    %
                              active,
                              pipelined );
BEGIN
    DEFAULTS
        bus_active  = VCC;  % Define signals to be normally high %
        pipe_active = VCC;
    END DEFAULTS;
    bus_cycle.clk = clk2;   % State machine runs at 66 MHz         %
    bus_cycle.reset = reset;
    CASE ( bus_cycle ) IS
        WHEN wait =>                          % Check for clk to   %
            IF (!/ads & clk) THEN             % determine phase    %
                bus_cycle = active;
            END IF;
        WHEN active =>
            IF (!/ready & /ads & clk) THEN
                bus_cycle = wait;
```

Figure 7. BUS_TRCK.TDF (Part 2 of 2)

```
            ELSIF (!/ready & !/ads & clk) THEN
                bus_cycle = pipelined;
            END IF;
            bus_active = GND;                  % Activate bus_active  %
        WHEN pipelined =>
            IF (clk) THEN
                bus_cycle = active;
            END IF;
            pipe_active = GND;                 % Activate pipe_active %
    END CASE;
END;
```

The bus_cycle state machine is defined in the Variable Section of the TDF. The state bits are named bus1 and bus0 in the format bus[1..0], and the states are defined as wait, active, and pipelined. MAX+PLUS II automatically generates state assignments for each state during compilation.

After the BEGIN keyword, the Clock and the Reset for the state machine are defined with the statements bus_cycle.clk = clk2 and bus_cycle.reset = reset. Next, the transition equations for the state machine are defined with Case and If Statements for the conditional state transfers. When /ads goes low and CLK is high in the state wait, then the state machine changes to the state active, and the bus_active signal goes low.

TRAN_CTL.TDF

The transceiver controller state machine, TRAN_CTL, controls the Output Enable of the two 74245 data transceivers in the data bus. See Figure 8.

Figure 8. TRAN_CTL State Diagram

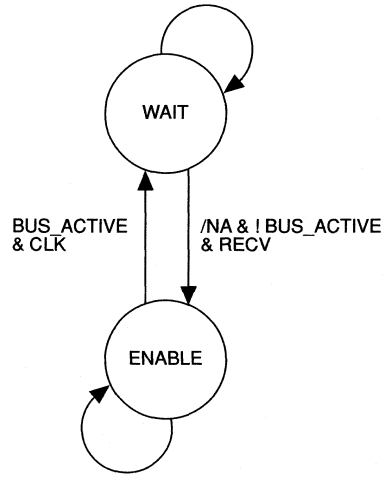


Figure 9 shows the TDF for TRAN_CTL, which contains a single-bit state machine called tran_en. When a non-pipelined bus cycle is executed with the statement /na & !bus_active, tran_en reads the 80386 control signals to determine whether the transceiver needs to be enabled. The enable is released when the bus cycle is completed with the statement bus_active & clk.

Figure 9. TRAN_CTL.TDF

```
SUBDESIGN tran_ctl
(
    clk             : INPUT;     % 80386 Clock                    %
    clk2            : INPUT;     % 80386 2x Clock (66 MHz)        %
    /na             : INPUT;     % Low to begin bus cycles        %
    bus_active      : INPUT;     % Low during active bus cycles   %
    recv            : INPUT;     % Low for recover                %
    reset           : INPUT;     % High for Reset of device       %

    /tri_enable     : OUTPUT;    % Low to enable io transceiver   %
)

VARIABLE

    tran_en : MACHINE OF BITS (tran)
              WITH STATES (wait, enable);

BEGIN
    tran_en.clk = clk2;
    tran_en.reset = reset;

    CASE (tran_en) IS
        WHEN wait =>
           IF (/na & !bus_active & recv) THEN
              tran_en = enable;
           END IF;
           /tri_enable = VCC;          % Set /tri_enable high  %
        WHEN enable =>                 % in the state wait     %
           IF (bus_active & clk) THEN
              tran_en = wait;
           END IF;
    END CASE;
END;
```

DECODE.TDF

DECODE is a state machine that tracks the 80386 control signals to decode the bus cycle (shown in Figure 5). The TDF for DECODE is shown in Figure 10. The state machine io_state has ten states and uses six state registers (two more than the required four) because all of the registers are used as outputs. The state register outputs are defined in the Variable Section as binary numbers. AHDL allows all values to be represented in binary, octal, decimal, or hexadecimal formats.

The io_state state machine waits in its initial state, called idle. When a bus cycle occurs, i.e., when bus_active goes low, the control signals from the 80386 are tested to see if wait states are required. If so, the state machine moves to a state that deactivates /READY to the 80386 and waits for the wait-state counter to time out. The time_delay signal, an output from WAIT_CNT, indicates that the wait-state counter has finished counting the required number of wait states. For example, a read from EPROM takes the state machine to epread1, where it waits for time_delay to go high. When time_delay goes high, io_state moves to epread2 to enable the processor, then returns to the idle state on the next Clock cycle.

Figure 10. DECODE.TDF (Part 1 of 2)

```
SUBDESIGN decode
(
    clk2        : INPUT;    % 80386 2x Clock (66 MHz)              %
    clk         : INPUT;    % 80386 Clock                          %
    /na         : INPUT;    % Low to begin bus cycles              %
    mio         : INPUT;    % High for memory, low for io cycles   %
    wr          : INPUT;    % High for write, low for read cycles  %
    dc          : INPUT;    % High for data, low for ctrl cycles   %
    a31         : INPUT;    % Processor address line A31           %
    time_delay  : INPUT;    % Time delay input                     %
    bus_active  : INPUT;    % Low during active bus cycles         %
    reset       : INPUT;    % High for Reset of device             %
    recv        : OUTPUT;   % Low during float and recovery        %
    iord        : OUTPUT;   % Low to read io                       %
    iowr        : OUTPUT;   % Low to write io                      %
    eprd        : OUTPUT;   % Low to read EPROMs                   %
    inta        : OUTPUT;   % Low for interrupt acknowledge        %
    iordy       : OUTPUT;   % Low to indicate ready                %
)
VARIABLE
    io_state : MACHINE OF BITS ( io[5..0]      % State assignments %
               WITH STATES ( idle   = b"111111", % "b" denotes     %
                             ioread1  = b"011111", % a binary number %
                             ioread2  = b"011101",
                             iowrite1 = b"011110",
                             iowrite2 = b"111101",
                             epread1  = b"110111",
                             epread2  = b"110101",
                             intack1  = b"111011",
                             intack2  = b"111001",
                             recover  = b"111110" );
BEGIN
    (iord,iowr,eprd,inta,iordy,recv) = io[]; % Assign outputs to   %
    io_state.clk = clk2;                     % state register bits %
    io_state.reset = reset;
    CASE ( io_state ) IS
        WHEN idle =>
            IF (/na & !bus_active & clk) THEN
                IF (a31 & mio & dc & !wr) THEN
                    io_state = epread1;
                ELSIF (!a31 & !mio & dc & wr) THEN
                    io_state = iowrite1;
                ELSIF (!a31 & !mio & dc & !wr) THEN
                    io_state = ioread1;
```

Figure 10. DECODE.TDF (Part 2 of 2)

```
                ELSIF (!a31 & !mio & !dc & !wr) THEN
                    io_state = intack1;
                ELSIF (mio & !dc & wr) THEN
                    io_state = iowrite2;
                ELSE io_state = recover;
                END IF;
            END IF;
        WHEN epread1 =>
            IF (time_delay) THEN
                io_state = epread2;
            END IF;
        WHEN epread2 =>
            IF (clk) THEN
                io_state = idle;
            END IF;
        WHEN iowrite1 =>
            IF (time_delay) THEN
                io_state = iowrite2;
            END IF;
        WHEN iowrite2 =>
            IF (!mio & clk) THEN
                io_state = recover;
            ELSIF (mio & clk) THEN
                io_state = idle;
            END IF;
        WHEN ioread1 =>
            IF (time_delay) THEN
                io_state = ioread2;
            END IF;
        WHEN ioread2 =>
            IF (clk) THEN
                io_state = recover;
            END IF;
        WHEN intack1 =>
            IF (time_delay) THEN
                io_state = intack2;
            END IF;
        WHEN intack2 =>
            IF (clk) THEN
                io_state = recover;
            END IF;
        WHEN recover =>
            IF (time_delay & clk) THEN
                io_state = idle;
            END IF;
    END CASE;
END;
```

WAIT_CNT.TDF

Figure 11 shows the TDF for WAIT_CNT, which is a counter that generates the wait states required to accommodate the slow hardware modules in the system. It is idle when it is not counting wait states. The wait-state counter is activated by iord, iowr, inta, or eprd, all of which originate from DECODE. Once activated, the counter loads a preassigned value and

begins to count until the time-out count reaches zero. At this time, the signal time_delay goes low, releasing the bus for the next cycle.

The constants at the beginning of WAIT_CNT.TDF define the different number of wait states required for each type of read/write function. The constants are then included in the equations that are used to load the timer. Constants allow you to place descriptive names rather than numbers into equations. For example, in WAIT_CNT.TDF, EPROM is assigned the value 1 in the Constant Statement. When data is being read from EPROM, the timer is loaded with this value. The cycle is completed once the counter reaches 15 (14 Clock wait states).

Figure 11. WAIT_CNT.TDF

```
CONSTANT I/O      = 5;     % Define constants              %
CONSTANT EPROM    = 1;
CONSTANT RECOVER  = 10;
CONSTANT TIME_UP  = 15;
CONSTANT IDLE     = 0;

SUBDESIGN wait_cnt
(
    clk2        : INPUT;  % 80386 CLK2                    %
    iord        : INPUT;  % Low to read io                %
    iowr        : INPUT;  % Low to write io               %
    eprd        : INPUT;  % Low to read EPROMs            %
    inta        : INPUT;  % Low for interrupt acknowledge %
    recv        : INPUT;  % Low during float and recovery %
    reset       : INPUT;  % High to reset                 %
    time_delay  : OUTPUT; % Time delay output             %
)
VARIABLE
    timer[3..0] : DFF;
BEGIN
    timer[].clk  = clk2;
    timer[].clrn = !reset;

    IF (timer[] == IDLE) THEN              % Load counter %
        IF (!iord # !iowr # !inta) THEN
            timer[] = I/O;
        ELSIF (!eprd) THEN
            timer[] = EPROM;
        ELSIF (!recv) THEN
            timer[] = RECOVER;
        ELSE timer[] = IDLE;
        END IF;
    ELSIF (timer[] == TIME_UP) THEN        % Check if count is done %
        time_delay = VCC;
        IF (iord & iowr & eprd & inta) THEN
            timer[] = IDLE;
        ELSE timer[] = TIME_UP;
        END IF;
    ELSE timer[] = timer[] + 1;            % Count %
    END IF;
END;
```

The conditional equations for the counter in WAIT_CNT.TDF are created with If Statements. The counter registers are defined in the Variable Section with the statement `timer[3..0]:DFF`, which assigns the names `timer3`, `timer2`, `timer1`, and `timer0` to four D flipflops. The statement `timer = timer + 1` at the end of WAIT_CNT.TDF causes the registers to count.

Design Processing

The MAX+PLUS II Compiler extracts netlists from the graphic and text files that define the hierarchical BUS_CNTL project and checks them against design rules. It then synthesizes the project for the MAX 5000 architecture. If the Compiler detects any errors during compilation, it opens the design file containing the error and highlights the exact location of the error.

The Compiler's Fitter module fits the project into the EPM5016 EPLD using a set of heuristic algorithms that automatically route project signals. The Compiler also generates a Report File (.RPT) that documents the resource utilization for the project. Figure 12 shows a portion of the Report File for the BUS_CNTL project. Sixteen macrocells are required for this design, 8 of which are buried. In addition, 19 expanders are used, leaving 13 expanders for additional logic.

Figure 12. Report File Excerpt (BUS_CNTL.RPT)

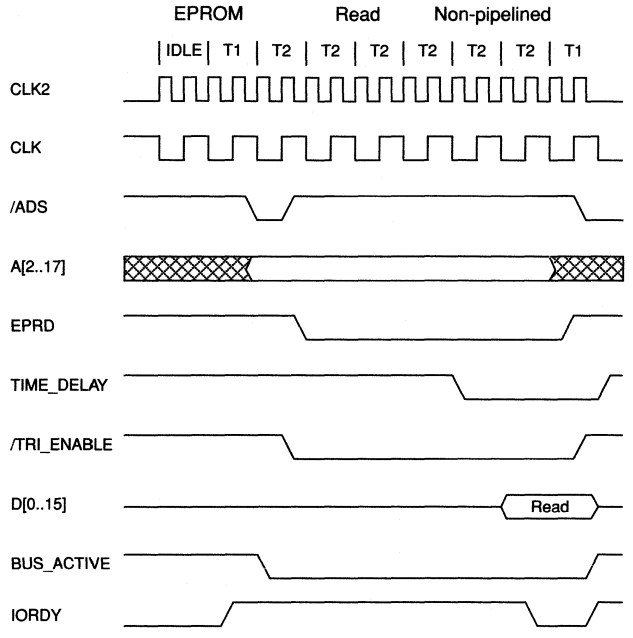
```
** RESOURCE USAGE **

                                                    Shareable
Logic Array Block     Macrocells    I/O Pins        Expanders

A:      MC1 - MC16    16/16(100%)   8/ 8(100%)   19/32( 59%)

Total dedicated input pins used:                  8/  8  (100%)
Total I/O pins used:                              8/  8  (100%)
Total macrocells used:                            16/ 16 (100%)
Total shareable expanders used:                   19/ 32 ( 59%)
Total shareable expanders not available (n/a):    2/  32 ( 69%)

Total input pins required:                        9
Total output pins required:                       7
Total bidirectional pins required:                0
Total macrocells required:                        16
Total shareable expanders in database:            17

Synthesized macrocells                            0/  16  ( 0%)
```

A Read from EPROM

Figure 13 shows the bus cycle timing for the 80386. In the EPROM interface, the `OE` input of each EPROM device is connected directly to the `EPRD` signal from the EPM5016 bus controller, and `/CE` is always grounded. The wait-state requirement is calculated by adding worst-case delays and comparing the total delay time to the 80386 bus cycle time.

The EPROM output signals become valid 200 ns after an address is placed on the address signals. To activate the `/READY` signal, the number of wait

Figure 13. 80386 Bus Cycle Timing Diagram



states required must include the sum of the EPROM address-to-valid delay and the latch Clock-to-output delay less one state machine cycle. Thus, the total delay required, in nanoseconds, is

$$t_{WAIT} = t_{EPROM} + t_{LATCH} - t_{CYCLE}$$
$$198 = 200 + 13 - 15$$

Each clocked wait state is 15 ns. Therefore, 14 wait-state cycles are required to meet the specification of the EPROM hardware (15 * 14 = 210; 210 > 198).

Design Verification

You can create input vectors for simulation either in text format with a Vector File (.VEC), or by drawing the waveforms in a Simulator Channel File (.SCF) in the MAX+PLUS II Waveform Editor. In BUS_CNTL.VEC, the sample Vector File shown in Figure 14, the input vectors simulate a read from the EPROM.

Figure 14. Sample Vector File (BUS_CNTL.VEC)

```
GROUP CREATE timer = |WAIT_CNT:66|timer3.Q
                     |WAIT_CNT:66|timer2.Q
                     |WAIT_CNT:66|timer1.Q
                     |WAIT_CNT:66|timer0.Q ;

OUTPUTS CLK READY EPRD DEN
        |TRAN_CTL:59|tran_en
        |BUS_TRCK:36|bus_cycle
        |DECODE:65|io_state
        timer
        CLK ;

INPUTS  RESET;
PATTERN
0>    1          % Default time unit is ns %
100>  0;

INPUTS  CLK2;     % Define CLK2 to be 66 MHZ %
INTERVAL  7.5;
START  200;
STOP  5us;
PATTERN  1 0;

% Define static inputs %
INPUTS  W/R PA31 NA M/IO D/C DRAMRDY;
PATTERN
0>       0   1   1   1    1   1;

INPUTS ADS;
PATTERN
0>       1
430>     0
490>     1;
```

The Group Create Section in the Vector File groups the timer bits together into a bus. Input vectors are entered in Pattern Sections. The input RESET, for example, is defined to be a 1 for the first 100 ns, then 0 until the end of simulation. For the CLK2 input, the Interval Section defines the cycle transition time as 7.5 ns. The Start and Stop Sections define the period of time for the vector pattern to repeat. The next Pattern Section causes CLK2 to toggle. The other input signals are similarly specified.

The outputs from simulation can also be viewed in either graphic or text format. The MAX+PLUS II Simulator automatically generates an SCF that contains waveforms of simulation inputs and outputs, which can be viewed and edited in the Waveform Editor. For example, if some inputs must be changed for further simulation, you can edit the SCF in the Waveform Editor and then submit it to the Simulator again for further simulation.

Conclusion

The MAX 5000 architecture and the advanced MAX+PLUS II development system allow you to enter and fit a broad range of complex designs into the EPM5016 EPLD. In addition, the high-drive, 24-mA outputs of the EPM5016 enhance its usability by allowing direct connection to most system buses. The EPM5016 EPLD meets the challenge of faster systems by offering counter speeds up to 100 MHz. Complex state machines, such as the bus controller in this application note, can be clocked at up to 66 MHz, twice the speed of some of today's fastest processors.

References

Intel Corporation. *386 Microprocessor Hardware Reference Manual* (1988).
Intel Corporation. *386DX Microprocessor Data Sheet* (April 1989).

Designing a FIFO Controller with an EPM5064 MAX EPLD

April 1992, ver. 2

Application Note 21

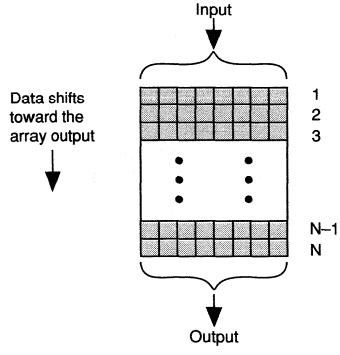
Introduction

Interacting digital subsystems often consume and produce data at different rates and times. A first in/first out (FIFO) buffer in a data path can bridge this rate and time mismatch so systems function correctly and efficiently. With the appropriate FIFO buffer, data written into the FIFO is read in the order it was entered, but at an independent rate. This application note describes how to design an 8-Kbyte × 16-bit FIFO controller with MAX+PLUS II development software and an Altera EPM5064 EPLD, a 64-macrocell MAX 5000 device.

Types of FIFOs

Two types of FIFO buffers are available: shift FIFOs and pointer FIFOs. A shift FIFO is typically constructed from a linear array of registers, where the number of registers equals the number of bits in the data word. See Figure 1. Data stored in the FIFO is written to the first location in the register array. Subsequent data is also written to this location, causing previously written data to shift toward the output of the array. When data has shifted to the last location of the register array, it can be read.

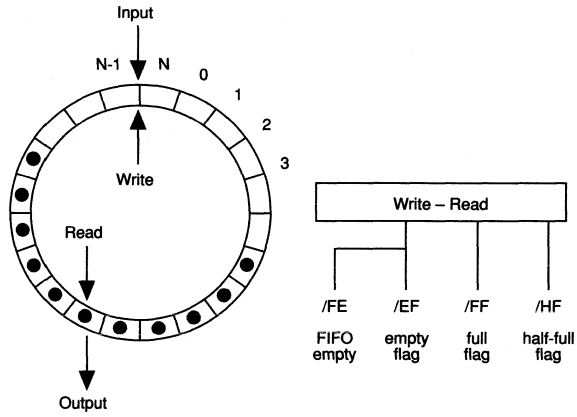
Figure 1. FIFO Register Array



Although a shift FIFO is simpler to implement than a pointer FIFO, it has two drawbacks. First, each shift FIFO has a fall-through time delay, which is the time required for valid data to propagate from the input to the output of an empty FIFO. Secondly, the read and write rates of a shift FIFO must be the same. Since data shifts out of the FIFO only when data shifts in, data is read into the register array only as fast as data is written to the FIFO.

In contrast, a pointer FIFO does not have a fall-through time delay and can accommodate different read and write rates. Data is stored in an array (e.g., RAM array). Read pointers store the next array address from which data is read, and write pointers store the next address to be written to this array. If these pointers consist of counters that roll over (i.e., restart at 0) at a value equal to the size of the FIFO, the FIFO structure becomes a circle. See Figure 2.

Figure 2. Circular Pointer FIFO



You can determine the FIFO status (full, partially full, or empty) by adding the following up/down counter to the design:

amount of data = (number of writes) – (number of reads)

If the amount of data is equal to zero, the FIFO is empty. If the number is *n* (where *n* is the size of the FIFO array), the FIFO is full. Any other value indicates that the FIFO is partially full.

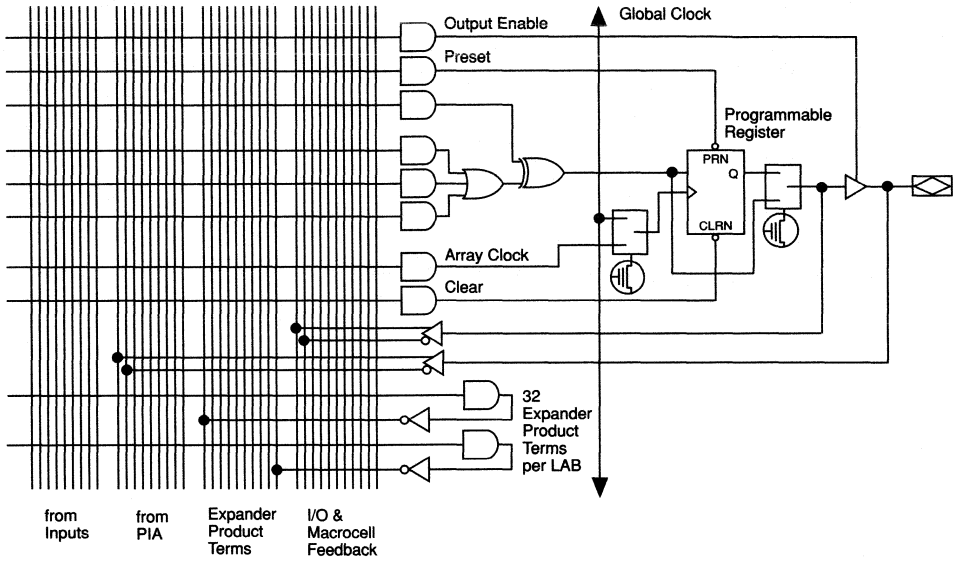
Design Options

Pointer FIFOs are available in a single package that contains read and write pointers and a RAM array. With this option, however, the size of the RAM arrays and the type of control functions available are limited. You can work around this limitation by using a standard static RAM (SRAM) device and implementing the control logic in an EPLD to maximize buffer-size options and meet custom interface-control requirements. The EPLD/SRAM solution also significantly reduces cost. An 8-Kbyte × 16-bit FIFO implemented with SRAMs and an EPLD costs less than half the price of a comparable off-the-shelf solution with two 8-Kbyte × 8-bit FIFOs.

EPM5064 EPLD Overview

The EPM5064 EPLD contains 64 flexible macrocells. Each macrocell has a register that can be programmed for D, T, JK, or SR operation and asynchronous Preset and Clear. Each macrocell can also be configured as a flow-through latch or bypassed for purely combinatorial operation. See Figure 3.

Figure 3. EPM5064 Macrocell

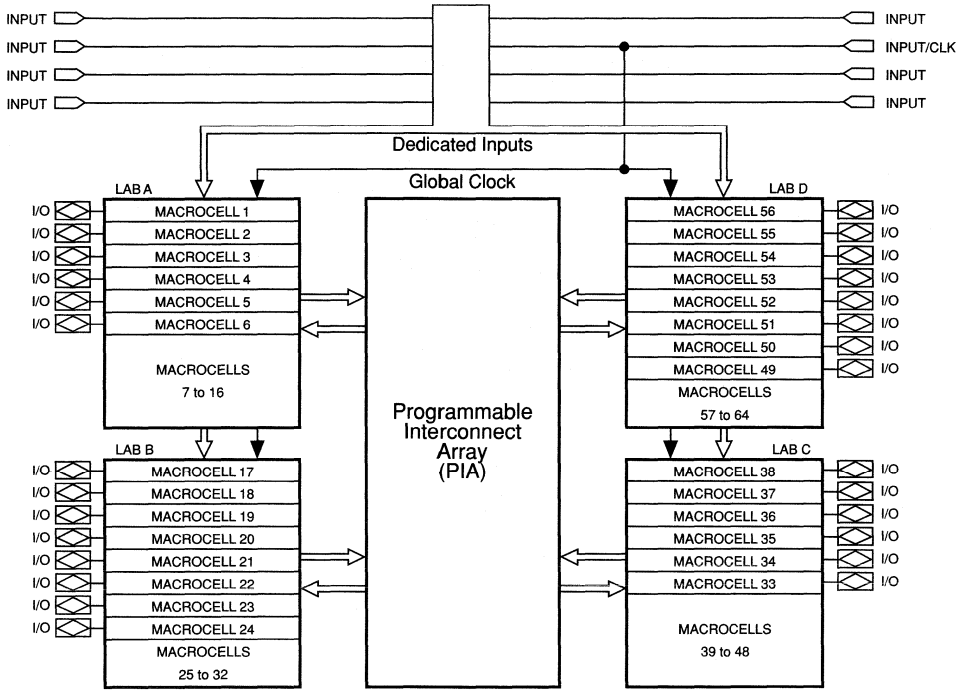


The 64 macrocells are grouped into 4 Logic Array Blocks (LABs), each containing 16 macrocells and 32 shareable expanders. See Figure 4. Shareable expanders are freely allocatable product terms that can be used and shared by any macrocell within the LAB.

A Programmable Interconnect Array (PIA) routes signals between the various LABs in the EPM5064 EPLD. The PIA, which has access to all 28 I/O pins and 64 macrocell outputs in the EPLD, gives each LAB access to all signals on the PIA. The PIA has enough routing resources to implement the most complex design. A fixed interconnect delay across the PIA eliminates skew and produces consistent, predictable performance.

The EPM5064 EPLD is an ideal choice for implementing an FIFO controller because the application requires a large amount of buried logic and has a low input and output pin requirement. This device has the internal logic resources to implement these buried functions, yet the 44-pin windowed ceramic and plastic J-lead (JLCC and PLCC) packages require only 1/2 square inch of board space.

Figure 4. EPM5064 Block Diagram



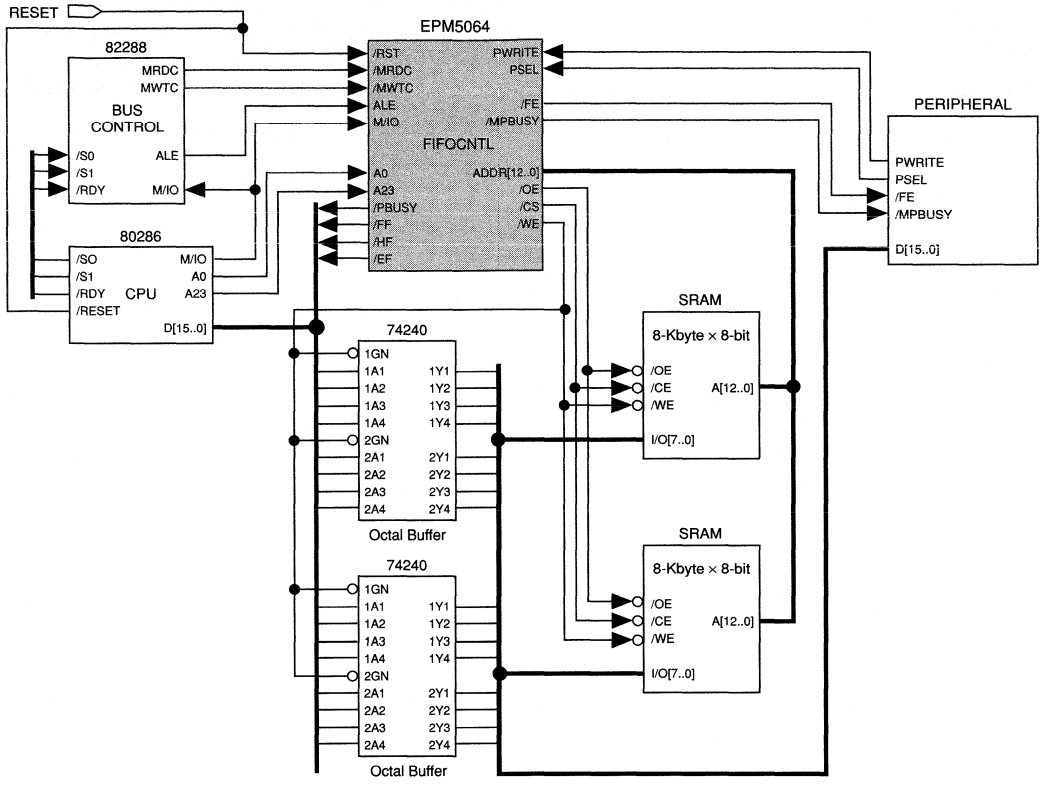
Refer to the *EPM5016 to EPM5192 EPLDs: High-Speed, High-Density MAX 5000 Devices Data Sheet* in the Altera 1992 ***Data Book*** for additional information on the EPM5064 architecture and performance.

Design Example

Figure 5 shows a typical pointer FIFO buffer application implemented with an EPM5064 EPLD and a pair of SRAM devices. The input to the FIFO is a 10-MHz 80286 microprocessor. The FIFO controller interfaces to the `/mrdc`, `/mwtc`, `m/io`, and `ale` signals from the 80286 and 82288 bus controllers. Chip select is generated by decoding the address lines `a0` and `a23`.

The microprocessor can read the FIFO status from the four active-low tri-stated lines that are connected directly to the microprocessor's data bus: `/ef` (empty flag), `/hf` (half-full flag), `/ff` (full flag), and `/pbusy` (peripheral busy). The microprocessor bus and the peripheral bus are isolated by a pair of 74240 octal buffer devices controlled by the `/we` (write enable) output of the EPM5064 FIFO controller.

Figure 5. EPM5064 FIFO Application



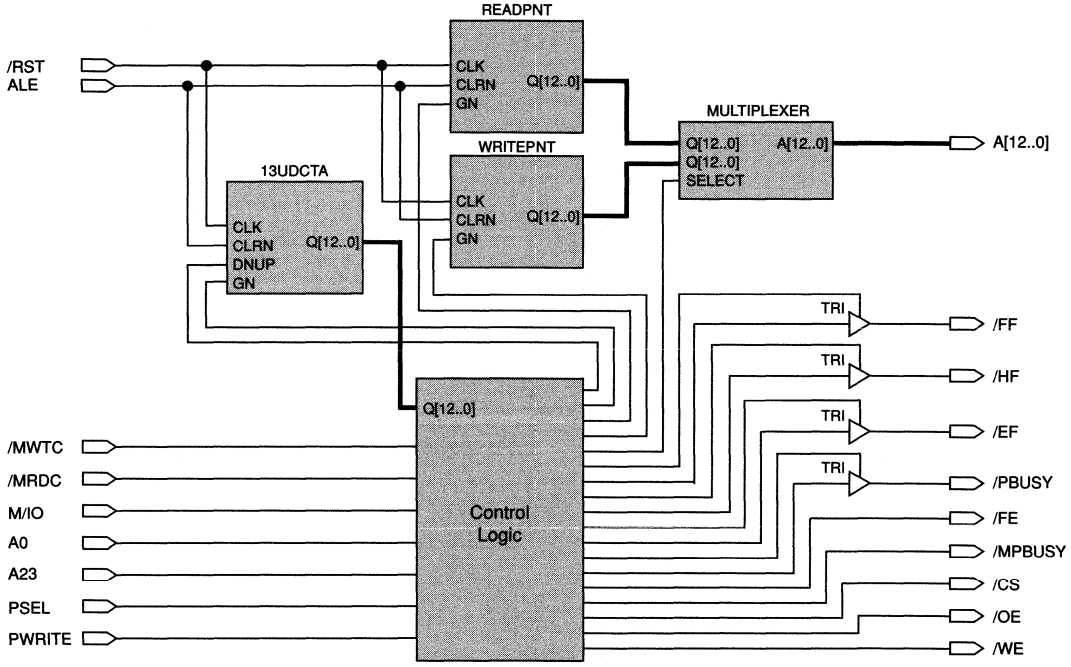
In this example, an arbitrary peripheral is connected to the output of the FIFO. This peripheral selects the FIFO with the `psel` (peripheral select) line and strobes data out of the FIFO with the `pwrite` (peripheral write) line. The peripheral determines the status of the FIFO by reading the `/fe` (FIFO empty) and `/mpbusy` (microprocessor busy) signals.

The randomly accessible storage array is provided by two 8-Kbyte × 8-bit SRAM devices and stores up to 8192 16-bit data words (the same number of bits as the 80286 microprocessor's data bus). The FIFO controller implemented in the EPM5064 EPLD provides the SRAM with all required control and address lines, including `/oe` (Output Enable), `/cs` (chip select), `/we`, and address lines `a0` to `a12`.

FIFO Controller

Figure 6 shows the block diagram for the EPM5064 FIFO controller. Two 13-bit counters provide the read and write pointers for the EPM5064 device. The read pointer counter contains the address where the next read data is stored; the write pointer counter contains the address where the next write data is stored. A third 13-bit counter, 13UDCTA, indicates the FIFO status (full, half full, or empty).

Figure 6. EPM5064 FIFO Contoller



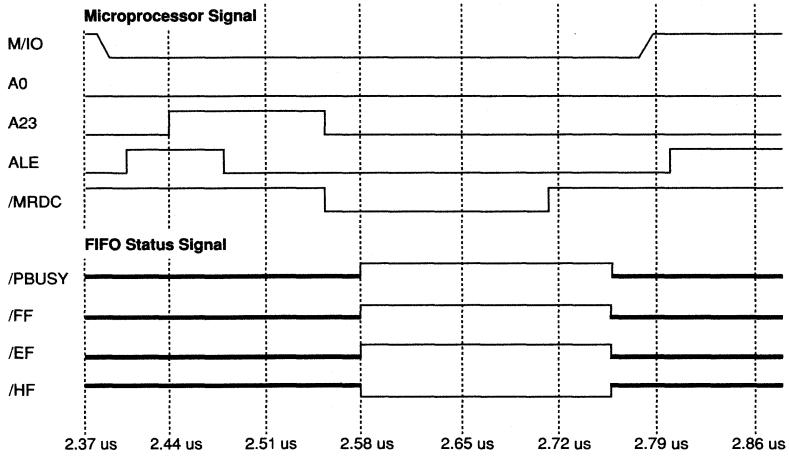
When a FIFO read or write operation is required, a 26- to 13-bit address-pointer multiplexer selects the read or write pointer outputs and drives them onto the SRAM address bus. The control logic block decodes FIFO requests, provides FIFO status outputs, and controls the address-pointer multiplexer.

FIFO Read Cycle

To obtain a FIFO status, an I/O read cycle may be performed with a0 low and a23 high. You can use the MAX+PLUS II Simulator to perform a full timing simulation of all critical cycles of the FIFO control with 0.1-ns resolution.

Figure 7 shows the results of a timing simulation for a status read cycle. The four FIFO status outputs, connected to the data bus, go from a high-impedance state to an active state 25 ns after the /mrdc line from the 80286 microprocessor goes low. The outputs can then be read in by the microprocessor. The /pbusy, /ff, and /ef lines are high, and the /hf line is low in response to the /mrdc strobe. These conditions indicate that the peripheral is not currently using the FIFO and is more than half full. The microprocessor can now perform a FIFO write.

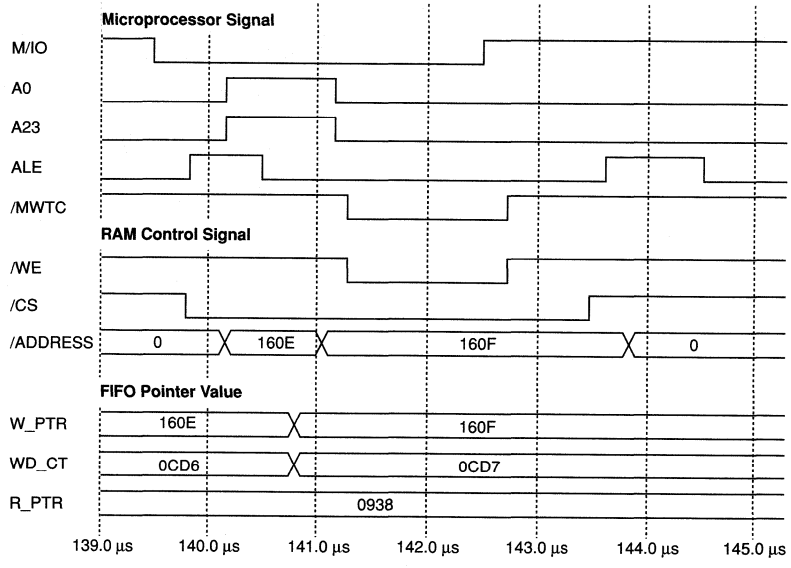
Figure 7. FIFO Status Read Cycle



FIFO Write Cycle

Figure 8 shows the results of a FIFO write simulation. The microprocessor selects a FIFO write with the a0, a23, and m/io lines. In response, the /cs output from the FIFO controller goes low to select the appropriate SRAM devices. The falling edge of the ale input signal increments the write pointer from 160E hex to 160F hex. The size counter also increments the pointer from 0CD6 hex to 0CD7 hex, indicating that data in the FIFO has increased by one word. Since no read activity occurs, the read pointer stays constant at 0938 hex. The SRAM address, which is derived from the write pointer, is presented to the SRAM address bus by the multiplexer, which selects between the read and write pointers. Finally, in response to the /mwtc signal from the 80286 microprocessor, the FIFO controller sends out the /we signal to strobe data into the SRAM.

Figure 8. FIFO Write Cycle

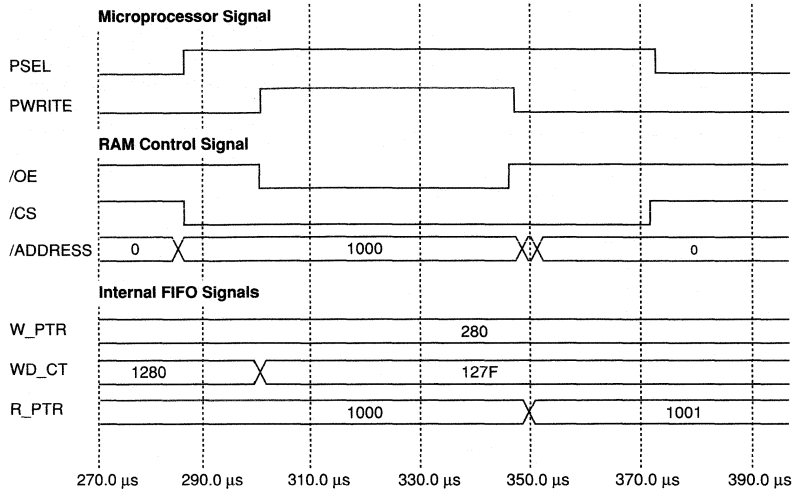


Peripheral FIFO Read Cycle

Figure 9 shows sample simulation outputs for a FIFO read cycle. The peripheral checks that the `/mpbusy` and `/fe` lines are inactive. Since both lines are high, the peripheral takes control of the FIFO by raising the `psel` line. The current read location address, `1000` hex, is placed on the SRAM address bus. The `/cs` control for the SRAM array is selected to prepare for a data write.

When the peripheral is ready to accept a data word, it strobes the `prwrite` line on the FIFO controller. In response, the FIFO controller sends a `/oe` signal to the SRAM array, causing the selected data word to be written to the peripheral bus. The leading edge of `pwrite` decrements the status counter from `1280` hex to `127F` hex, indicating that the data stored in the FIFO has decreased by one word. With the falling edge of the `pwrite` strobe, the read pointer is incremented from `1000` hex to `1001` hex.

Figure 9. FIFO Read Cycle



Design Entry

Figure 10 shows FIFOCNTL.TDF, a top-level Text Design File (.TDF) implemented with the Altera Hardware Description Language (AHDL). You can enter the FIFO controller design (designs are called "projects" in MAX+PLUS II) with the MAX+PLUS II Text Editor or any other ASCII text editor. For more information on AHDL, refer to MAX+PLUS II Help.

Figure 10. AHDL Text Design File for FIFO Buffer (Part 1 of 4)

```
Title "FIFO Controller Application for EPM5064 EPLD";

CONSTANT SIZE_OF_FIFO  = h"1FFF";
CONSTANT HALF_FIFO     = h"1000";
CONSTANT EMPTY         = h"0000";

FUNCTION 13udcta (clk, clrn, dnup, gn)    % inputs to function  %
    RETURNS (q[12..0]);                   % outputs to function %

DESIGN IS fifocntl
DEVICE IS EPM5064-1;

SUBDESIGN fifocntl
(
% MPU Inputs %
    /rst            : INPUT;    % reset         %
    /mwtc           : INPUT;    % /write        %
    /mrdc           : INPUT;    % /read         %
    a0              : INPUT;    % chip select   %
    a23             : INPUT;    % chip select   %
    ale             : INPUT;    % data strobe   %
    m/io            : INPUT;

% Peripheral Inputs %
    psel            : INPUT;
    pwrite          : INPUT;

% FIFO Status Outputs %
    /ef             : OUTPUT;   % empty flag               %
    /hf             : OUTPUT;   % half-full flag           %
    /ff             : OUTPUT;   % full flag                %
    /mpbusy         : OUTPUT;   % micro using fifo         %
    /fe             : OUTPUT;   % half-full flag           %
    /pbusy          : OUTPUT;   % peripheral using fifo    %

% SRAM Interface %
    addr[12..0]     : OUTPUT;   % SRAM address             %
    /cs             : OUTPUT;   % SRAM chip select         %
    /oe             : OUTPUT;   % SRAM output enable       %
    /we             : OUTPUT;   % SRAM write enable        %
)
```

Figure 10. AHDL Text Design File for FIFO Buffer (Part 2 of 4)

```
%                  Internal Node Assignments                          %
VARIABLE
    w_ptr[12..0]        : dff;          % SRAM write address        %
    r_ptr[12..0]        : dff;          % SRAM read address         %
    wd_ct               : 13udcta;      % word counter              %
    selhold             : latch;        % select latch              %
    rdhold              : latch;        % busy read latch           %
    fsel                : node;         % active-high select        %
    write               : node;         % write enabled             %
    read                : node;         % read enabled              %
    ramoe               : node;
    ramcs               : node;
    ramwea              : node;
    data_strobe         : node;         % active-high strobe        %
    status              : node;
    peflag              : node;         % peripheral fifo empty     %
    mbflag              : node;
    sram_addr[12..0]    : mcell;        % SRAM address              %
    not_empty           : mcell;        % internal not empty        %
    not_full            : mcell;        % internal not full         %
    half_full           : mcell;        % internal half full        %
    fflag               : tri;
    hflag               : tri;
    eflag               : tri;
    pbflag              : tri;

BEGIN

%                    FIFO Control Logic                                %
%                                                                      %
% This section defines all control logic for the status outputs        %
% The FIFO select and FIFO read latches are also defined.              %

    selhold.d       = a0 & a23 & !m/io;    % active-high FIFO select %
    selhold.ena     = ale;                 % latch                   %
    fsel            = selhold.q;

    rdhold.d        = !a0 & a23 & !m/io;   % active-high read select %
    rdhold.ena      = ale;                 % latch                   %
    status          = rdhold.q;

    data_strobe     = !ale;                % active-high data strobe %
    write           = !/mwtc;              % write decoded from r/w  %
    read            = !/mrdc;              % read decoded from r/w   %

% These control signals are sent back to the MPU to indicate           %
% that the FIFO is empty, half-full, or full.                          %

    not_empty       = (wd_ct.q[] != EMPTY);
    half_full       = (wd_ct.q[]  < HALF_FIFO);
    not_full        = (wd_ct.q[] != SIZE_OF_FIFO);
```

Figure 10. AHDL Text Design File for FIFO Buffer (Part 3 of 4)

```
    pbflag.in       = !psel;
    /pbusy          = pbflag.out;
    pbflag.oe       = status & read;

    eflag.in        = not_empty;
    /ef             = eflag.out;            % output ready flag          %
    eflag.oe        = status & read;

    fflag.in        = not_full;
    /ff             = fflag.out;            % input ready flag           %
    fflag.oe        = status & read;

    hflag.in        = half_full;
    /hf             = hflag.out;            % half-full status flag      %
    hflag.oe        = status & read;

% These control signals are sent to the peripheral to indicate            %
% that the FIFO is busy with the microprocessor unit and/or empty.        %

    peflag          = (wd_ct.q[] != EMPTY);
    /fe             = peflag;               % peripheral fifo empty flag %

    mbflag          = !fsel;
    /mpbusy         = mbflag;

% SRAM control signals %

    ramwe           = !(fsel AND write AND not_full);
    /we             = ramwe;
    ramoe           = !(psel AND pwrite AND not_empty);
    /oe             = ramoe;
    ramcs           = !(fsel OR psel);
    /cs             = ramcs;

%               Read/Write Pointer and Control Logic                      %
%                                                                         %
% This section controls the updating of the read, write, and word         %
% counters. A read operation increments the read pointer and              %
% decrements the word counter. A write operation increments the           %
% write and word counter.                                                 %

% An external Reset sets the read and write addresses and the             %
% number of words in the FIFO to zero.                                    %

    w_ptr[].clrn    = /rst;
    r_ptr[].clrn    = /rst;
    wd_ct.clrn      = /rst;
```

Figure 10. AHDL Text Design File for FIFO Buffer (Part 4 of 4)

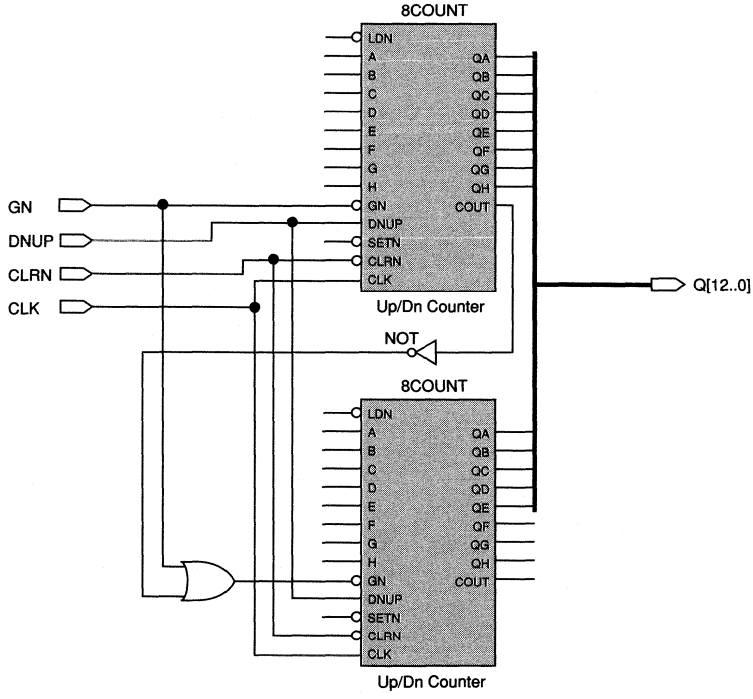
```
% The end of the microprocessor unit's data strobe signal is used to       %
% update the status of the FIFO controller after a read or write operation.%

    w_ptr[].clk     = data_strobe;
    r_ptr[].clk     = !pwrite;
    wd_ct.clk       = (data_strobe AND fsel) or (pwrite AND psel);

% FIFO Read Operation  %
    IF (psel AND not_empty) THEN
        wd_ct.gn    = GND;
        wd_ct.dnup  = VCC;
        r_ptr[]     = r_ptr[] + 1;
        w_ptr[]     = w_ptr[];

% FIFO Write Operation %
    ELSIF (fsel AND not_full) THEN
        wd_ct.gn    = GND;
        wd_ct.dnup  = GND;
        r_ptr[]     = r_ptr[];
        w_ptr[]     = w_ptr[] + 1;

% FIFO Idle            %
    ELSE
        wd_ct.gn    = VCC;
        r_ptr[]     = r_ptr[];
        w_ptr[]     = w_ptr[];
    END IF;

%                              Multiplexer                                  %
%                                                                           %
% The read pointer drives the static RAM address bus during read            %
% operations and the write pointer drives during write operations.          %

    IF fsel THEN
        sram_addr[] = w_ptr[];
    ELSIF psel THEN
        sram_addr[] = r_ptr[];
    END IF;

    addr[] = sram_addr[];

END;
```

The FIFOCNTL.TDF file starts with a Title and three Constant Statements that correspond to the empty, half-full, and full values of the FIFO. These constants are later used for arithmetic comparators to determine the status of the FIFO.

A Function Prototype Statement is included for `13UDCTA`, a 13-bit up/down counter implemented in a Graphic Design File (.GDF). The `wd_ct` variable is then declared as an instance of the `13UDCTA` macrofunction in the Variable Section. The Function Prototype Statement can be used to easily integrate any other design file into a TDF.

Figure 11 shows 13UDCTA.GDF, which contains two cascading 8-bit counters (`8COUNT`) from the MAX+PLUS II TTL MacroFunction Library. In one of the counters, three higher-order bits (`QF`, `QG`, and `QH`) are not connected. These bits are not required in the project and are removed during MAX+PLUS II logic synthesis.

Figure 11. 13UDCTA.GDF (13-Bit Counter)



Next in the TDF, the input and output pin names are defined in the Subdesign Section with the keywords INPUT and OUTPUT. In a top-level TDF, these pin names correspond to actual EPLD pin names.

Following the Subdesign Section is the Variable Section, which contains the internal node assignments such as w_ptr[12..0] and w_ptr[12..0]. These read and write pointers are 13-bit counters that simply count each read or write cycle and roll over to zero at a value of 8191. The register bits are defined by 13 D flipflops with the following statements:

```
w_ptr[12..0] : dff
w_ptr[12..0] : dff
```

Control Logic

After all internal nodes are declared, the design logic is specified in the Logic Section, which starts with the keyword BEGIN. First, the FIFO control logic is defined with the FIFO select and read latches, selhold and rdhold. Since both selhold and rdhold are flow-through D latches, the D and Enable inputs and Q outputs are explicitly defined.

Next, the tri-stated status signals /pbusy, /ef, /ff, and /hf, which are read by the microprocessor, are defined. An arithmetic comparison is used to define the active-low signals not_empty, half_full, and not_full. In all three cases, the status counter wd_ct.q[] is compared to previously defined constant values. The signal not_empty is high when wd_ct.q[] is not equal to EMPTY. When wd_ct.q[] is less than HALF_FIFO, the signal half_full is high. Finally, not_full is defined to be high when wd_ct.q[] is not equal to SIZE_OF_FIFO. The signals not_empty, half_full, and not_full are then defined as inputs to the tri-state buffers that have the status signals /ef, /hf, and /ff as outputs.

The peripheral status signals are then defined. When the status counter wd_ct.q[] is not equal to zero, the FIFO is not empty, so /fe is defined to be high (inactive). /mpbusy goes to active low when the FIFO is selected by the microprocessor, i.e., when the latched signal FSEL is high.

The final signals defined in the FIFO Control Logic section are the RAM control signals /we, /oe, and /cs.

Read/Write Pointers and Status Control

The read/write pointer and status control block are then defined. First, the Clear and Clock signals to the three 13-bit counters w_ptr, r_ptr, and wd_ct are specified. The group w_ptr[12..0].clk can be abbreviated to w_prt[].clk. The empty brackets represent the entire group.

The operation of all three counters is defined with an If Statement. If the peripheral selects the FIFO for a read when the FIFO is not empty, wd_ct (status counter) is decremented and r_ptr (read pointer) is incremented. Since wd_ct is a predefined counter, it is decremented by pulling its active-low Enable line to ground (wd_ct.gn = GND) and selecting it to count up on the next Clock cycle (wd_ct.dnup = VCC). The 13 D flipflops that make up r_ptr are controlled by defining r_ptr[] = r_ptr[] + 1. During FIFO reads, w_ptr stays constant (w_ptr[] = w_ptr[]).

When the microprocessor selects the FIFO for a write and the FIFO is not full, wd_ct and w_ptr are incremented. To specify these conditions, the Enable line wd_ct is pulled to ground and the counter is selected to count up (wd_ct.dnup = GND). Now w_ptr is incremented with the statement w_ptr[] = w_ptr[] + 1. During FIFO writes, r_ptr stays constant (r_ptr[] = r_ptr[]).

When neither a read nor a write operation is specified, all counters stay at their current value. The Enable line for wd_ct is set to VCC. Both r_ptr and w_ptr are defined so their next value equals their present value.

Multiplexer

The multiplexer block is also implemented with an If Statement. If the microprocessor selects the FIFO, the address placed on the RAM address bus is derived from the w_ptr counter. If the peripheral selects the FIFO, r_ptr is placed on the address bus.

Design Compilation

You can compile the FIFO Controller project with the MAX+PLUS II Compiler. The Compiler synthesizes the logic and generates a Report File (.RPT) that lists the minimized equations and resource usage for the project. The project in this application note requires 62 of the 64 macrocells available in the EPM5064 EPLD, yielding 97% device utilization.

You can also generate a timing Simulator Netlist File (.SNF) that contains all information for timing simulation and a Programmer Object File (.POF) for programming the EPM5064 device.

Conclusion

The EPM5064 EPLD is an excellent device for implementing an FIFO controller because the application requires a large number of buried resources. The EPM5064 device also has flexible architecture and requires a minimum of board space (only 1/2 inch square). You can create the project with the advanced MAX+PLUS II software, which allows you to enter the project quickly and efficiently in the format that best suits your requirements.

Designing with AHDL

April 1992, ver. 2 **Application Note 22**

Introduction

The Altera Hardware Description Language (AHDL) is a powerful, flexible text-based language for creating EPLD designs (called "projects" in MAX+PLUS II). AHDL incorporates the features offered by text-based languages such as ABEL and CUPL, and adds the following significant enhancements:

- Eliminates the need to explicitly define the inputs of highly complex combinatorial functions in a schematic design
- Simplifies comparator design with equality operators
- Streamlines adder and counter designs with built-in arithmetic capability
- Allows bus-based microprocessor interface designs with AHDL bus syntax
- Supports several behavioral constructs that simplify state machine design
- Supports Classic, MAX 5000, MAX 7000, and Synchronous Timing Generator (STG) EPLDs
- Allows multi-device partitioning
- Allows AHDL Text Design Files (.TDF) to be mixed with other text, graphic, and waveform files for hierarchical designs

Although you can use any ASCII text editor to create an AHDL TDF, the MAX+PLUS II Text Editor has definite advantages. Since AHDL is fully integrated into MAX+PLUS II software, you can enter, compile, and debug your design within a single development system. MAX+PLUS II also offers delay prediction, which reports the worst-case timing for any signal path in an AHDL design. These features allow you to create AHDL designs quickly and efficiently.

The following topics are discussed in this application note:

- Overview of AHDL design structure
- How to use AHDL
- AHDL applications
- Top-down design approach

AHDL Design Structure

An AHDL logic design must contain, at a minimum, a Subdesign Section and a Logic Section. All other sections and statements are optional. An AHDL TDF at the top level of a hierarchical project hierarchy can contain a Design Section only to specify the physical EPLD resources.

The following AHDL sections are discussed here:

- Subdesign Section & Logic Section
- Design Section
- Variable Section

For complete information on these and other AHDL sections, statements, and elements, refer to MAX+PLUS II Help.

Subdesign Section & Logic Section

Most AHDL files begin with a Subdesign Section and a Logic Section. The Subdesign Section declares the input, output, and bidirectional ports of the file. The Logic Section is the body of the Subdesign Section and specifies the logical operation of the design file. Unless you are only using the TDF to make physical EPLD resource assignments, you must include a Subdesign and Logic Section in all AHDL files.

The logical operation of a project is defined by one or more of the following statements and constructs in the Logic Section:

- Boolean equations for logical connections and signal flows
- If and Case Statements for conditional logic
- Truth Table Statements for decode logic
- Default Statements for specifying default constant values for variables
- In-line references for implementing macrofunctions and primitives

Design Section

The Design Section assigns logic to a particular pin, macrocell, chip, clique, or logic option, and specifies the placement of design logic in a device. If you do not include a Design Section in your TDF, the Compiler automatically fits the project into the best combination of EPLDs and assigns the resources within them. The Compiler ignores the Design Section unless it is in the top-level file of a project hierarchy.

You can incorporate one or more of the following subsections into a Design Section:

- Clique Assignment Statement to keep logic together in a single Logic Array Block (LAB)
- EPLD Specification to assign a chip (i.e., a block of logic) to a specific device
- Pin Connection Statement to specify board-level pin connections
- Resource Assignment Statement to assign nodes to specific chips, or to specific pins or macrocells in one or more chips, and to assign logic options

For information on how to make resource and device assignments and how to partition a project among multiple devices, see *Application Brief 93 (Partitioning a Project with MAX+PLUS II Software)* in this handbook.

Variable Section

The Variable Section declares variables used in the Logic Section. AHDL variables define buried logic. You can incorporate one or more of the following subsections into a Variable Section:

- Instance Declaration to declare each instance of a primitive or macrofunction
- Node Declaration to hold input or output information that is not declared in the Subdesign Section or elsewhere in the Variable Section
- Register Declaration to declare registers, including D, T, JK, and SR flipflops and latches
- State Machine Declaration to declare a state machine name, its states, and optionally, the state bits

Using AHDL

AHDL supports a broad range of applications. You can create entire hierarchical projects with AHDL, or mix AHDL files with schematic, waveform, Electronic Design Interchange Format (EDIF), and other design descriptions in a hierarchy. AHDL is especially well suited for complex combinatorial logic, state machines, and truth tables.

The following topics are discussed in this section:

- Implementing conditional logic (If and Case Statements)
- Implementing decode logic (Truth Table Statement)
- Specifying default values for constants (Defaults Statement)
- Assigning logic options (Options and Resource Assignment Statements)
- Back-annotating, preserving, and editing assignments (Design Section)
- Implementing a macrofunction or primitive in AHDL
- Creating groups in AHDL
- Implementing a state machine

Implementing Conditional Logic (If & Case Statements)

You can implement conditional logic in AHDL with If Statements or Case Statements. The If Statement describes conditional logic based on the evaluation of one or more Boolean expressions. The keywords `ELSE` and `ELSIF` prioritize the order in which conditions are evaluated. The Case Statement lists alternative conditions based on the value of the variable, group, or expression following the `CASE` keyword.

Figure 1 illustrates how you can use the two statements to define the same logic.

Figure 1. If and Case Statements Used for Conditional Logic

Conditional Logic with If Statement

```
SUBDESIGN ifthen
(
    a[7..0], b[7..0]          : INPUT;
    c[7..0], d[7..0], s[2..0] : INPUT;
    out[7..0]                 : OUTPUT;
)

 BEGIN

    IF s[] == 0 THEN
        out[] = a[];
    ELSIF    s[] == 1 THEN
        out[] = b[];
    ELSIF    s[] == 2 THEN
        out[] = c[];
    ELSE
        out[] = d[];
    END IF;

END;
```

Conditional Logic with Case Statement

```
SUBDESIGN ncase
(
    a[7..0], n[7..0]          : INPUT;
    c[7..0], d[7..0], s[2..0] : INPUT;
    out[7..0]                 : OUTPUT;
)

BEGIN

    CASE s[] IS
        WHEN 0 =>
            out[] = a[];
        WHEN 1 =>
            out[] = b[];
        WHEN 3 =>
            out[] = c[];
        WHEN OTHERS =>
            out[] = d[];
    END CASE;

END;
```

These methods differ in two ways:

- An If Statement can evaluate Boolean expressions, whereas the Case Statement can only compare one Boolean expression to a constant.
- The interpretation of a complex If Statement may generate logic that requires more product terms than are available on the target EPLD or that cannot be minimized by the Compiler.

Implementing Decode Logic (Truth Table Statement)

You can use the Truth Table Statement in AHDL to decode output values for a set of inputs. The Truth Table Statement eliminates the need to extract Boolean expressions from a function table because it directly implements the table. Truth tables are ideally suited for expressing BCD decoders, address decoders, and state machine transition logic. Figure 2 shows a sample TDF for a decoder that contains a Truth Table Statement.

Figure 2. Decoder (Truth Table Statement) (TBLEX.TDF)

```
TITLE "TABLE EXAMPLE";

SUBDESIGN tblex
(
    A[2..0]       : INPUT;
    Q[7..0]       : OUTPUT;
)

VARIABLE
    TABLE  A[3..0]    =>  Q7,  Q6,  Q5,  Q4,  Q3,  Q2,  Q1,  Q0;
                   0  =>  0,   0,   0,   0,   0,   0,   0,   1;
                   1  =>  0,   0,   0,   0,   0,   0,   1,   0;
                   2  =>  0,   0,   0,   0,   0,   1,   0,   0;
                   3  =>  0,   0,   0,   0,   1,   0,   0,   0;
                   4  =>  0,   0,   0,   1,   0,   0,   0,   0;
                   5  =>  0,   0,   1,   0,   0,   0,   0,   0;
                   6  =>  0,   1,   0,   0,   0,   0,   0,   0;
                   7  =>  1,   0,   0,   0,   0,   0,   0,   0;
    END TABLE;
END;
```

Specifying Default Values for Constants (Defaults Statement)

The AHDL Defaults Statement specifies default constant values for variables used in Truth Table, If, and Case Statements. The default constant values, VCC or GND, simplify complex projects by eliminating the need for you to explicitly define all possible input conditions.

In Figure 3, a Defaults Statement defines VCC as the default for the /detonate signal. When fire or tamper are a logical high, /detonate is defined as GND. In all other cases, /detonate defaults to VCC. When fire or tamper are a logical low, the equation in the Defaults Statement is activated, and /detonate is defined as VCC.

Figure 3. Default Constant Values (Defaults Statement) (BOMB1.TDF)

```
SUBDESIGN bomb1
(
    fire, tamper : INPUT;
    /detonate    : OUTPUT;
)
BEGIN
    DEFAULTS
        /detonate = VCC;
    END DEFAULTS;

    IF fire # tamper THEN
        /detonate = GND;
    END IF;
END;
```

You can also simplify the design process by incorporating Defaults Statements in complex truth table projects. For example, the truth table shown in Figure 4 decodes an 8-bit address bus. If you do not use a Defaults Statement, you must make 256 entries to ensure that the outputs of the truth table are tied to VCC when the inputs are not decoded. For example, you can define only 8 of the 256 possible values, then use a Defaults Statement to specify a default value for the remaining 248 entries.

Figure 4. Default Constant Values to Simplify a Truth Table (DEFAULT.TDF)

```
SUBDESIGN default
(
    add[7..0]          : INPUT;
    decode[3..0]       : OUTPUT;
)

BEGIN
    DEFAULTS
        decode[3..0] = VCC;
    END DEFAULTS;

    TABLE
        add[7..0]     =>  decode[3..0];
        B"11001001"   =>  0;
        B"00111010"   =>  1;
        B"10101111"   =>  2;
        B"00001100"   =>  3;
        B"11110111"   =>  4;
        B"10000001"   =>  5;
        B"11100101"   =>  6;
        B"00000001"   =>  7;
    END TABLE;
END;
```

Assigning Logic Options (Options & Resource Assignment Statements)

You can set the macrocell Turbo Bit, Parallel Expanders, XOR Synthesis, and Minimization logic options in your top-level TDF with the Resource Assignment Statement or Options Statement. You can also enter logic option assignments with menu commands in the MAX+PLUS II Text Editor and Compiler. See MAX+PLUS II Help for details on logic options and making logic option assignments.

COMBO.TDF, the simple combinatorial circuit in Figure 5, shows a Design Section, a Subdesign Section, and a Logic Section. The Design Section contains an EPLD Specification and an Options Statement that set the device and logic options for COMBO.TDF. The logic options are set for the entire EPM5016 EPLD.

Figure 5. Logic Option Assignments (Options Statement)

```
DESIGN IS combo

BEGIN
    DEVICE IS "EPM5016DC-17";  % Defines the EPLD and speed grade  %
    OPTIONS SECURITY = OFF;     % Security Bit is off              %
    OPTIONS XOR_SYNTHESIS = ON; % XOR Synthesis is on             %
END;

SUBDESIGN combo                 % Subdesign Section begins        %
(
    a, b, c, d   : INPUT;
    y, z         : OUTPUT;
)

BEGIN                           % Logic Section begins            %
    y = (a $ b) & c;
    z = a & b & c & d;
END;                            % Logic Section ends              %
                                % Subdesign Section ends          %
```

You can also assign logic options on a macrocell-by-macrocell basis in the Resource Assignment Statement. Figure 6 shows COMBO2.TDF, which adds pin and macrocell assignments to the COMBO.TDF file. These assignments follow the BEGIN keyword and are followed by the resource type, which can be INPUT, OUTPUT, BIDIR, or BURIED. The XOR Synthesis option is turned on for all macrocells listed in the Resource Assignment Statement except for the macrocell containing node t.

Figure 6. Logic Option Assignments (Resource Assignment Statement) (Part 1 of 2)

```
DESIGN IS combo2

BEGIN
    DEVICE IS "EPM5016DC-17"

    BEGIN
        OPTIONS SECURITY = OFF;
        OPTIONS XOR_SYNTHESIS = ON;
        a @ 1    : INPUT;
        b @ 2    : INPUT;
        c @ 9    : INPUT;
        d @ 10   : INPUT;
        t @ mc2  : BURIED (XOR_SYNTHESIS = OFF);
        y @ 18   : OUTPUT;
        z @ 17   : OUTPUT;
        bi @ 4   : OUTPUT;
        ck @ 3   : INPUT;
        oe @ 12  : INPUT;
    END;
END;
```

Figure 6. Logic Option Assignments (Resource Assignment Statement) (Part 2 of 2)

```
SUBDESIGN combo
(
    a,b,c,d,oe,ck      : INPUT;
    y,z,bi             : OUTPUT;
)

VARIABLE

    t                  : DFF;

BEGIN
    t.d = a $ b;
    t.clk = ck;
    y = t.q & c;
    z = a & b & c & d;
    bi = tri(y,oe);
END;
```

Back-Annotating, Preserving & Editing Assignments (Design Section)

Back-annotation copies resource and device assignments in the compiled project back into the original design files for the project. This process ensures that subsequent compilations produce the same fit.

Device and resource assignments created with the **Pin/MC/Chip**, **Clique**, **Device**, and **Enter Assignments** commands in the Graphic, Text, and Waveform Editors are stored in the Probe & Resource Assignment File (.PRB) for the project. Device and resource assignments can also be made in the Design Section of a TDF that is at the top of the project hierarchy. If your top-level file is not a TDF, you can make assignments in a top-level TDF with the same name and process it together with the top-level design file. This top-level TDF must contain a Design Section only. The back-annotation procedure you choose depends on which method you have used to create device and resource assignments.

You can back-annotate resource and device assignments into the .PRB file by following these steps:

1. Choose the **Project Back-Annotate** command from the File menu.
2. Select the types of assignments to be back-annotated.
3. Choose **OK**.

The **Project Back-Annotate** command copies the selected assignments from the most recent successful compilation into the .PRB file, overwriting the previous assignments. All back-annotated resource and device assignments are listed under *Existing Assignments* in the **Enter Assignments** and **Device** dialog boxes, respectively.

To back-annotate assignments into the Design Section of a TDF, you must copy the Fit File (.FIT) generated by the Compiler into the Design Section of the top-level TDF.

The Fit File records the pin, buried macrocell, chip, and device assignments for a MAX+PLUS II project. It also records all external pin connections specified with AHDL Pin Connection Statements. The Fit File uses AHDL Design Section syntax (see MAX+PLUS II Help for information on AHDL syntax).

You can copy the assignments in the Fit File to a top-level TDF and edit them to change the resource assignments of the project. You should never edit the Fit File. The Compiler gives TDF assignments top priority. Editing a TDF that contains assignments from the Fit File is especially useful when you are grouping resources into a single MAX 5000 or MAX 7000 LAB to maximize critical-path performance. For example, Figure 7 shows SAMPLE.GDF; the Fit File for this GDF is shown in Figure 8.

Figure 7. SAMPLE.GDF

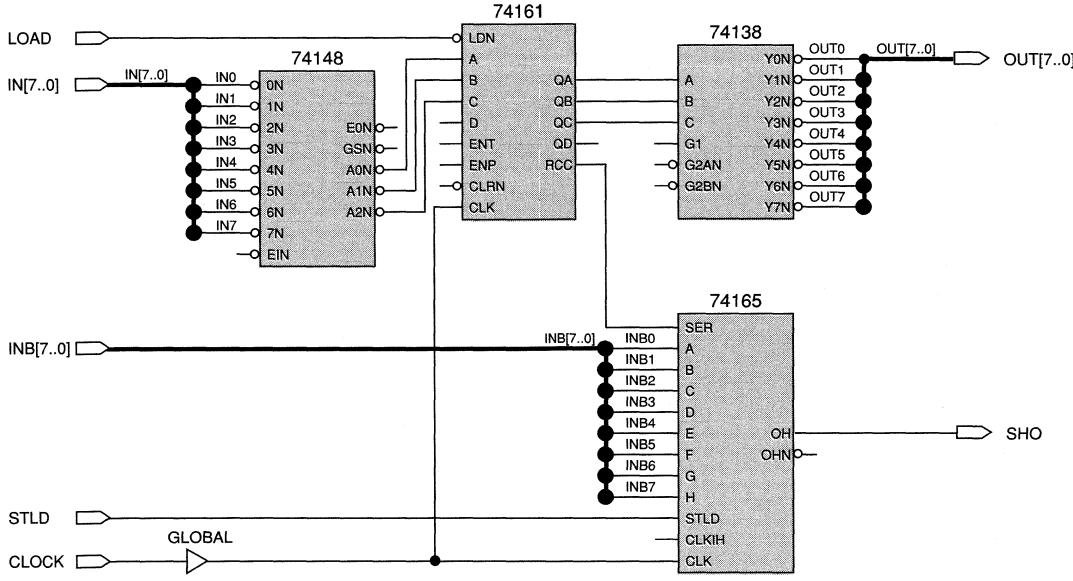


Figure 8. Fit File for SAMPLE.GDF (SAMPLE.FIT)

```
DESIGN IS "sample"
BEGIN
    DEVICE "SAMPLE" IS "EPM5064"
    BEGIN
        clock                          @  34   : INPUT  ;
        inb0                           @  33   : INPUT  ;
        inb1                           @  23   : INPUT  ; %MC24%
        inb2                           @  22   : INPUT  ; %MC23%
        inb3                           @  20   : INPUT  ; %MC22%
        inb4                           @  19   : INPUT  ; %MC21%
        inb5                           @  18   : INPUT  ; %MC20%
        inb6                           @  17   : INPUT  ; %MC19%
        inb7                           @  16   : INPUT  ; %MC18%
        in0                            @   7   : INPUT  ; %MC5%
        in1                            @  15   : INPUT  ; %MC17%
        in2                            @  30   : INPUT  ; %MC38%
        in3                            @  29   : INPUT  ; %MC37%
        in4                            @  31   : INPUT  ;
        in5                            @  13   : INPUT  ;
        in6                            @  12   : INPUT  ;
        in7                            @  11   : INPUT  ;
        load                           @   9   : INPUT  ;
        stld                           @  35   : INPUT  ;
        out0                           @  44   : OUTPUT ; %MC55%
        out1                           @  42   : OUTPUT ; %MC54%
        out2                           @  41   : OUTPUT ; %MC53%
        out3                           @  40   : OUTPUT ; %MC52%
        out4                           @  39   : OUTPUT ; %MC51%
        out5                           @  38   : OUTPUT ; %MC50%
        out6                           @  37   : OUTPUT ; %MC49%
        out7                           @   1   : OUTPUT ; %MC56%
        sho                            @   8   : OUTPUT ; %MC6%
        |74161:23|QA                   @ MC64  : BURIED ;
        |74161:23|QB                   @ MC63  : BURIED ;
        |74161:23|QC                   @ MC62  : BURIED ;
        |74161:23|QD                   @ MC61  : BURIED ;
        |74165:13|:37                  @ MC46  : BURIED ;
        |74165:13|:38                  @ MC57  : BURIED ;
        |74165:13|:65                  @ MC15  : BURIED ;
        |74165:13|:70                  @ MC43  : BURIED ;
        |74165:13|:79                  @ MC12  : BURIED ;
        |74165:13|:84                  @ MC7   : BURIED ;
        |74165:13|:93                  @ MC5   : BURIED ; %PIN 7%
    END;
END;
```

You can group the inputs and outputs of the 74165 shift register in SAMPLE.GDF into a single LAB in the EPM5064 EPLD. The assignments from the Fit File for SAMPLE.GDF can be copied into a TDF with the same name (SAMPLE.TDF) and edited. Figure 9 shows a TDF that contains Fit File assignments that were edited to assign the shift register function to macrocells in the same LAB.

Figure 9. Edited Assignments for SAMPLE.GDF (SAMPLE.TDF)

The assignments from the Fit File were copied to a TDF and edited to assign all macrocells for the 74165 macrofunction to LAB B.

```
DESIGN IS "sample"
BEGIN
    DEVICE "SAMPLE" IS "EPM5064"
    BEGIN
        clock                          @  34   : INPUT  ;
        inb0                           @  33   : INPUT  ;
        inb1                           @  23   : INPUT  ; %MC24%
        inb2                           @  22   : INPUT  ; %MC23%
        inb3                           @  20   : INPUT  ; %MC22%
        inb4                           @  19   : INPUT  ; %MC21%
        inb5                           @  18   : INPUT  ; %MC20%
        inb6                           @  17   : INPUT  ; %MC19%
        inb7                           @  16   : INPUT  ; %MC18%
        in0                            @   7   : INPUT  ; %MC5%
        in1                            @  15   : INPUT  ; %MC17%
        in2                            @  30   : INPUT  ; %MC38%
        in3                            @  29   : INPUT  ; %MC37%
        in4                            @  31   : INPUT  ;
        in5                            @  13   : INPUT  ;
        in6                            @  12   : INPUT  ;
        in7                            @  11   : INPUT  ;
        load                           @   9   : INPUT  ;
        stld                           @  35   : INPUT  ;
        out0                           @  44   : OUTPUT ; %MC55%
        out1                           @  42   : OUTPUT ; %MC54%
        out2                           @  41   : OUTPUT ; %MC53%
        out3                           @  40   : OUTPUT ; %MC52%
        out4                           @  39   : OUTPUT ; %MC51%
        out5                           @  38   : OUTPUT ; %MC50%
        out6                           @  37   : OUTPUT ; %MC49%
        out7                           @   1   : OUTPUT ; %MC56%
        sho                            @   8   : OUTPUT ; %MC6%
        |74161:23|QA                   @ MC64  : BURIED ;
        |74161:23|QB                   @ MC63  : BURIED ;
        |74161:23|QC                   @ MC62  : BURIED ;
        |74161:23|QD                   @ MC61  : BURIED ;
        |74165:13|:37                  @ MC31  : BURIED ;
        |74165:13|:38                  @ MC30  : BURIED ;
        |74165:13|:65                  @ MC29  : BURIED ;
        |74165:13|:70                  @ MC28  : BURIED ;
        |74165:13|:79                  @ MC27  : BURIED ;
        |74165:13|:84                  @ MC26  : BURIED ;
        |74165:13|:93                  @ MC25  : BURIED ;
    END;
END;
```

Implementing a Macrofunction or Primitive

With AHDL, you can can describe a function in a Function Prototype Statement. The function can then be incorporated into an AHDL file with an in-line reference or a variable declaration.

A Function Prototype Statement lists the function name and input, output, and bidirectional ports. Machine input and output ports can also be used for macrofunctions that import or export state machine signals. You do not need to use a Function Prototype Statement for MAX+PLUS II primitives unless you wish to change the default order of the inputs.

A Function Prototype Statement has the same function as a symbol in MAX+PLUS II schematic designs: it represents a design file with the same name. The following example shows a Function Prototype Statement for the 74174B macrofunction:

```
FUNCTION 74174B (d[6..1],clrn,clk)
    RETURNS (q[6..1]);
```

You can implement a primitive or macrofunction in an AHDL file in a single line with a variable declaration or an in-line reference. A variable declaration provides a symbolic name for an instance of a primitive or macrofunction.

Figure 10 shows how you can implement the 74174B bus macrofunction with a variable declaration or an in-line reference.

Figure 10. 74174B Bus Macrofunction Implementation (Part 1 of 2)

Variable Declaration

```
TITLE "Variable declaration example";

FUNCTION 74174B (d[6..1],clrn,clk)
    RETURNS (q[6..1]);

SUBDESIGN varbus
(
  data[6..1], clear, clock    : INPUT;
  out[6..1]                   : OUTPUT;
)

VARIABLE
  HEXLTCH : 74174b; % The "b" suffix indicates a bus macrofunction %
BEGIN
  HEXLTCH.clk         = clock;
  HEXLTCH.clrn        = clear;
  HEXLTCH.d[6..1]     = data[6..1];
  out[6..1]           = HEXLTCH.q[6..1];
END;
```

Figure 10. 74174B Bus Macrofunction Implementation (Part 2 of 2)

In-line Reference

```
TITLE "In-line reference example";

FUNCTION 74174B (d[6..1],clrn,clk)
    RETURNS(q[6..1]);

SUBDESIGN inlnbus
(
    data[6..1],clear,clock    : INPUT;
    out[6..1]                 : OUTPUT;
)

BEGIN

    out[] = 74174B (data[],clear,clock);

END;
```

The compact format of an in-line reference provides an easy, effective way to implement a macrofunction or primitive. However, an in-line reference requires you to match the order of the nodes listed in the Function Prototype, while a variable declaration allows you to list the inputs and outputs of a primitive or macrofunction in any order.

Creating Groups in AHDL

Symbolic names and ports of the same type can be declared and used as a group in Boolean expressions and equations. A group, which can include up to 256 members (or bits), is treated as a collection of nodes and acted upon as one unit. Groups in AHDL are analogous to buses in schematic design files.

Groups in AHDL can be declared in two ways:

- As a decimal group consisting of a symbolic name or port followed by a range of decimal numbers enclosed in brackets, e.g., `data[15..0]`
- As a sequential group consisting of a list of symbolic names, ports, or numbers separated by commas and enclosed in parentheses, e.g., `(a,b,c,d)`

You can use groups in AHDL statements to implement bus-oriented projects. Figure 11 shows how eight signals can be ANDed in group notation.

Figure 11. Group Notation (ANDBUS.TDF)

```
SUBDESIGN andbus
(
    in_a[7..0], in_b[7..0]    : INPUT;
    f_out[7..0]               : OUTPUT;
)

BEGIN

    f_out[] = in_a[] & in_b[];

END;
```

Implementing a State Machine

You can implement a state machine in AHDL with a State Machine Declaration, which declares the name of the state machine, its states, and, optionally, the bits of the machine. Figure 12 shows a state diagram with transitions between four states.

Figure 12. State Diagram for STATE.TDF

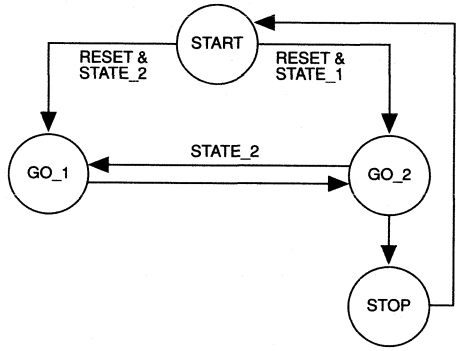


Figure 13 shows a TDF that implements the logic in the state diagram shown in Figure 12 with a State Machine Declaration and a Case Statement.

Figure 13. AHDL State Machine (STATE.TDF) (Part 1 of 2)

```
SUBDESIGN state
(
    clock                  : INPUT;
    reset, state_0         : INPUT;
    state_1, state_2       : INPUT;
    out[2..0]              : OUTPUT;

)
```

Figure 13. AHDL State Machine (STATE.TDF) (Part 2 of 2)

```
VARIABLE

    ss : MACHINE OF BITS (q[2..0])
         WITH STATES     (start = b"000",
                          go_1  = b"001",
                          go_2  = b"011",
                          stop  = b"111" );
BEGIN
    out[] = q[];
    ss.clk = clock;

    CASE (ss) IS
        WHEN start =>
            IF (reset & state_0) THEN
                ss = go_1;
            ELSIF (reset & state_1) THEN
                ss = go_2;
            END IF;
        WHEN go_1 =>
            ss = go_2;
        WHEN go_2 =>
            IF (state_2) THEN
                ss = go_1;
            ELSE
                ss = stop;
            END IF;
        WHEN stop =>
            ss = start;
    END CASE;

END;
```

AHDL Applications

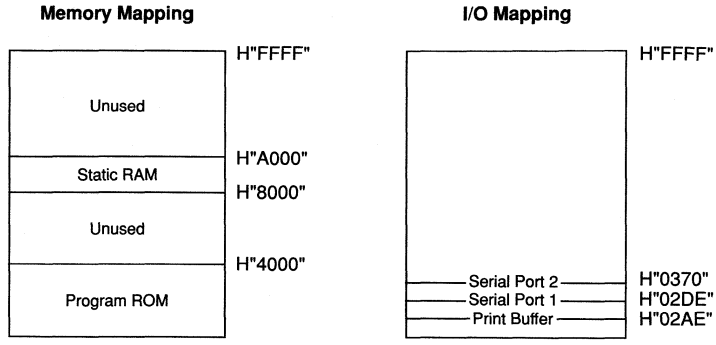
The applications in this section show all AHDL keywords in upper case and user-defined symbolic names in lower case. Consistent formatting (e.g., indentation of If and Case Statements) and comments, enclosed in percent symbols (%), are used to increase readability. The following applications are described here:

- Address decoder
- Counter
- Accumulator
- DRAM controller state machine
- Synchronous detector state machine

Address Decoder

EPLDs are frequently used to decode microprocessor address lines that provide system memory and I/O chip selects. I/O and memory mapping for a generalized 16-bit microprocessor is shown in Figure 14. This system contains a bank of ROM and a bank of RAM in memory space. I/O space includes a print buffer and two serial ports.

Figure 14. 16-Bit Microprocessor Memory and I/O Space



You can use AHDL groups and comparison operators (i.e., comparators) or a Truth Table Statement to implement an address decoder. The comparators provide quicker and more intuitive integration when multiple lines of a table are required; the truth table method provides a more compact format when the address range can be decoded with one line in a table. However, both methods produce the same results.

Figure 15 shows an address decoder for a generalized 16-bit microprocessor system, implemented with AHDL groups and comparators. You can easily modify this function to implement the address decoding for any microprocessor-based system.

Figure 15. Address Decoder (Groups & Comparators)

```
SUBDESIGN decode
% The AHDL Subdesign Section defines the input,     %
% output, and bidirectional ports of the file.      %

(
    addr[15..0],m/io          : INPUT;
    rom, ram, print, sp[2..1] : OUTPUT;
)

BEGIN
    % The BEGIN keyword marks the beginning of the %
    % Logic Section, which describes the logical   %
    % operation of the design file.                %

    rom     =   m/io & (addr[] <  H"4000");
    ram     =   m/io & (addr[] >= H"8000") & (addr[] < H"A000");
    print   =   !m/io & (addr[] == H"02AE");
    sp1     =   !m/io & (addr[] == H"02DE");
    sp2     =   !m/io & (addr[] == H"0370");
END;
```

Figure 16 shows the same address decoder implemented with a truth table.

Figure 16. Address Decoder (Truth Table Statement)

```
SUBDESIGN decode2
(
    addr[15..0], m/io          : INPUT;
    rom, ram, print, sp[2..1] : OUTPUT;
)
BEGIN
    TABLE
        m/io,    addr[15..0]           =>   rom, ram, print, sp[];
        1,       B"00XXXXXXXXXXXXXX" =>    1,   0,   0,     B"00";
        1,       B"100XXXXXXXXXXXX" =>     0,   1,   0,     B"00";
        0,       B"0000001010101110" =>    0,   0,   1,     B"00";
        0,       B"0000001011011110" =>    0,   0,   0,     B"01";
        0,       B"0000001101110000" =>    0,   0,   0,     B"10";
    END TABLE;
END;
```

Counter

Most designs require at least one counter for timing or control logic. EPLDs contain a flexible mix of combinatorial and registered logic, making them ideal for creating counters. The MAX+PLUS II TTL MacroFunction Library contains all common TTL building blocks for integrating counter logic.

Figure 17 shows an AHDL design file that integrates two 74161 counters to create an 8-bit counter. This file can be easily modified to connect several 74161 macrofunctions or other TTL counters. AHDL allows you to create a counter of any width.

Figure 17. 8-Bit Counter with 74161 Macrofunctions (TTLCOUNT.TDF)

```
FUNCTION 74161    (d,c,b,a,ldn,enp,ent,clrn,clk)
    RETURNS       (rco,qd,qc,qb,qa);
% The Function Prototype defines the inputs %
% and outputs of the 74161 macrofunction.   %
SUBDESIGN ttlcount
(
    clk, ld, en, clr, d[7..0] : INPUT;
    c8, q[7..0]               : OUTPUT;
)
VARIABLE
    c4                        : NODE;
BEGIN
    (c4,q[3..0]) = 74161(d[3..0], !ld, en, en, !clr, clk);
    (c8,q[7..0]) = 74161(d[7..4], !ld, en, c4, !clr, clk);
END;
```

You can also implement counters with the powerful AHDL operators and statements. Figure 18 shows a custom 9-bit counter that provides the same load, hold, and count functions as the 74161 macrofunction.

Figure 18. Counter with Load and Enable (AHDLCNT.TDF)

```
SUBDESIGN ahdlcnt
(
    clk, ld, en, clr, d[8..0] : INPUT;
    q[8..0]                   : OUTPUT;
)
VARIABLE
    count[8..0]               : DFF;
BEGIN
    count[].clk = clk;
    count[].clrn = !clr;

    IF ld THEN
        count[] = d[];
    ELSIF en THEN
        count[] = count[] + 1;
    ELSE
        count[] = count[];
    END IF;

    q[] = count[];
END;
```

You can modify the sample file in Figure 18 to integrate a counter of any width by changing the width of the d[ ], q[ ], and count[ ] groups. You can specify additional conditional logic to incorporate more counter functions. For example, Figure 19 shows how you can add count-down capability to the previous design with an additional input signal and a slight modification to the If Statement.

Figure 19. Up/Down Counter with Load and Enable (AHDLCNT2.TDF) (Part 1 of 2)

```
SUBDESIGN ahdlcnt2
(
    clk, ld, en, clr, up/down, d[8..0] : INPUT;
    q[8..0]                            : OUTPUT;
)
VARIABLE
    count[8..0]                        : DFF;
BEGIN
    count[].clk = clk;
    count[].clrn = !clr;
```

Figure 19. Up/Down Counter with Load and Enable (AHDLCNT2.TDF) (Part 2 of 2)

```
    IF ld THEN
        count[] = d[];
    ELSIF en THEN
        IF up/down THEN
            count[] = count[] + 1;
        ELSE
            count[] = count[] - 1;
        END IF;
    ELSE
        count[] = count[];
    END IF;

    q[] = count[];
END;
```

Accumulator

AHDL provides arithmetic and comparison operators to create compact arithmetic equations. With these operators, you can design an accumulator that stores the result of an adder and feeds the result back to the adder. This function allows you to add a series of numbers into a sum.

Figure 20 shows a sample file for a 16-bit accumulator. The 16-bit result is cleared to zero when the Clear input signal is high. Each Clock adds the value represented by the 8-bit bus (`add[ ]`) to the previously calculated sum in a bank of registers (`result[ ]`). The result is connected to the output signals (`sum[ ]`).

Figure 20. 16-Bit Accumulator (16INT.TDF)

```
SUBDESIGN 16int
(
    clock, add[7..0], clear    : INPUT;
    sum[15..0]                 : OUTPUT;
)

VARIABLE
    result[15..0]              : DFF;

BEGIN
    result[].clk = clock;
    result[].clrn = !clear;
    result[] = result[] + (0, add[]);
    sum[] = result[];
END;
```

DRAM Controller State Machine

Dynamic RAM (DRAM) devices are useful for inexpensive, efficient, high-density storage. However, DRAMs require controllers that ensure proper operation. Part of the controller typically uses a state machine that controls the active-low row address and column address strobes (/ras and /cas) and select lines (s[ ]) that select the row, column, or refresh addresses.

Figure 21 shows the AHDL implementation of a simple DRAM controller that controls these signals. This example uses state transitions defined with Case and If Statements that can be modified to create a more complex controller. Refer to "AHDL Top-Down Design" in this application note for information on how to implement an 8-MByte DRAM controller.

Figure 21. DRAM Controller State Machine (DRAM_SM.TDF) (Part 1 of 2)

```
SUBDESIGN dram_sm
(
    clk, /reset, /mreq, refresh        : INPUT;
    /ras, /cas, s[1..0]                : OUTPUT;
)
VARIABLE
    control :    MACHINE OF BITS (ras, cas, sel[1..0])
                     WITH STATES (
                              idle = B"0000",
                              strobe_row = B"1000",
                              strobe_col = B"1110",
                              refresh_dram = B"1011" );

BEGIN
    % When a memory request (/mreq) is received, the          %
    % controller generates a /ras signal, then switches a     %
    % multiplexer select and generates a /cas signal. When    %
    % a refresh is requested, the controller activates /ras   %
    % and waits for a refresh request to be deasserted        %
    % before going back to the idle state.                    %

    control.clk = clk;
    control.reset = !/reset;

    CASE control IS
        WHEN idle =>
            IF refresh THEN
                control = refresh_dram;
            ELSIF !/mreq THEN
                control = strobe_row;
            END IF;
        WHEN strobe_row =>
            control = strobe_col;
```

Figure 21. DRAM Controller State Machine (DRAM_SM.TDF) (Part 2 of 2)

```
        WHEN strobe_col =>
            control = idle;
        WHEN refresh_dram
            IF !refresh THEN
                control = idle;
            END IF;
    END CASE;

    /ras = !control.ras;
    /cas = !control.cas;
     s[] =  control.sel[];
END;
```

Synchronization Detector State Machine

To specify the transitions of a state machine, you must conditionally assign the states within a single behavioral construct. You can represent the state machine transitions with a truth table. Figure 22 shows a function that synchronizes a serial receiver to an incoming data stream. This synchronization detector state machine searches an incoming serial data stream for a pattern of six successive 1 values.

Figure 22. Synchronization Detector State Machine (SYNC_DET.TDF) (Part 1 of 2)

```
SUBDESIGN sync_det
(
    clock, data_in    : INPUT;
    sync              : OUTPUT;
)

VARIABLE
    detect : MACHINE OF BITS (q[2..0])
                     WITH STATES (
                         zero,
                         one,
                         two,
                         three,
                         four,
                         five );

BEGIN
    detect.clk = clock;
    TABLE
        detect, data_in  => detect,  sync;
        zero,   1        => one,     0;
        one,    1        => two,     0;
        one,    0        => zero,    0;
        two,    1        => three,   0;
        two,    0        => zero,    0;
```

Figure 22. Synchronization Detector State Machine (SYNC_DET.TDF) (Part 2 of 2)

```
        three,   1        => four, 0;
        three,   0        => zero, 0;
        four,    1        => five, 0;
        four,    0        => zero, 0;
        five,    1        => five, 1;
        five,    0        => zero, 0;
    END TABLE;
END;
```

This project samples the incoming data stream and moves to the subsequent state whenever the serial input is 1. Any 0 sampled in the stream causes the state machine to make a transition back to state `zero`. The synchronous signal is true whenever the machine is in state `five` (i.e., five successive 1 values have been sampled) and the `data_in` is at 1.

State names are not assigned state values in this example because the MAX+PLUS II Compiler automatically creates an efficient set of state bit assignments. However, some state machines have state bits that are used as outputs, in which case you must explicitly specify the state bit assignments as shown in Figure 21.

AHDL Top-Down Design

The following steps illustrate the top-down approach to AHDL design:

1. Create a block diagram of the project. In this step, you define the major functions and the communication between them.
2. Declare the inputs and outputs of the project. You can determine the required input, output, and bidirectional signals from the block diagram, and then declare them in the Subdesign Section.
3. Declare the major functions of the project. Each block of the diagram is a major function that you can easily translate into AHDL variables. You then declare these variables in the Variable Section.
4. Describe the major functions. You implement the internal logic for each of the major functions in the Logic Section.

You can enter comments in the file to enhance its readability.

The following example uses this top-down design method to create a DRAM controller that controls an 8-Mbyte DRAM system organized in 32-bit words.

Step 1: Create a Block Diagram

Figure 23 shows the block diagram of an 8-Mbyte DRAM controller that controls all access to dynamic RAM organized on a 32-bit bus. The controller interfaces directly to a microprocessor address and control bus and a 20-MHz Clock signal. It also provides reset capability. This system produces row address strobe (RAS), column address strobe (CAS), and the dynamic RAM address. In addition, the controller provides the data strobe-acknowledge (DSACK) to the processor. The controller consists of a refresh timer, a refresh counter, an address multiplexer, and a RAS/CAS generator.

Figure 23. 8-MByte DRAM Controller Block Diagram

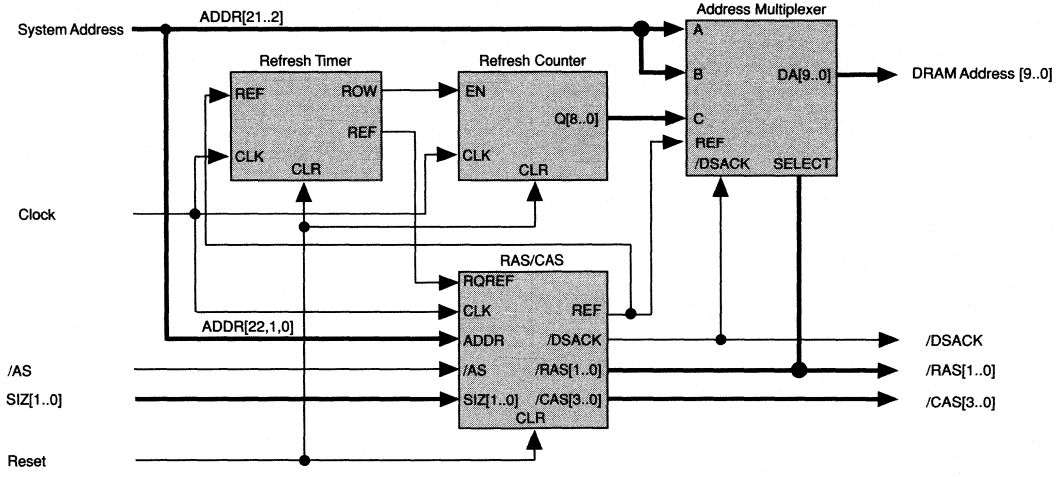
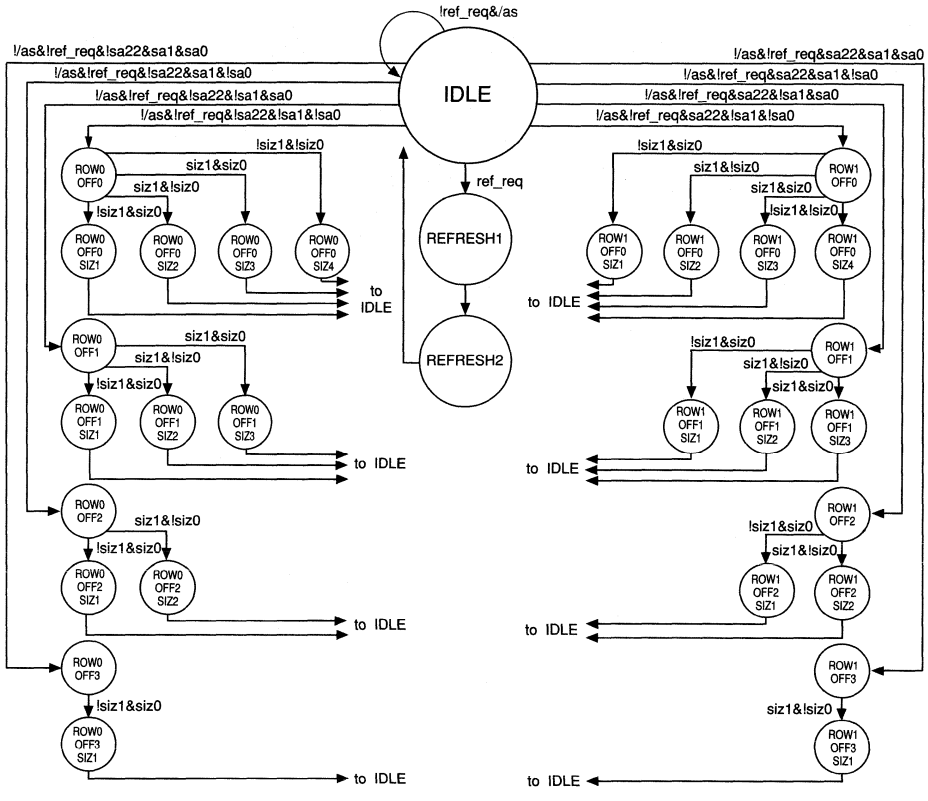


Figure 24 shows the state diagram for RAS/CAS generation.

Figure 24. State Transition Diagram for RAS/CAS Generator



Step 2: Declare the Inputs & Outputs

You can determine the interface to the DRAM controller from the block diagram and declare the inputs and outputs for this project in the Subdesign Section (see Figure 25).

Figure 25. DRAM Controller Input/Output Declarations (Subdesign Section)

```
TITLE "8-Megabyte DRAM Controller";

CONSTANT MAX_REF_TIMER = 311;

SUBDESIGN dram_ctl
(
        sa[22..0],    : INPUT      % system address              %
        clock,        : INPUT      % 20-MHz Clock                %
        size[1..0],   : INPUT      % size of memory operation    %
        /as,          : INPUT      % address strobe              %
        reset         : INPUT;     % master Reset                %

        da[9..0],     : OUTPUT     % DRAM address                %
        /dsack,       : OUTPUT     % data strobe acknowledge     %
        /ras[1..0],   : OUTPUT     % row address strobe          %
        /cas[3..0]    : OUTPUT;    % column address strobe        %
)
```

Step 3: Declare the Major Functions

Each block in the block diagram is a major function of the DRAM controller. You can declare these major functions as variables in the Variable Section of an AHDL file. See Figure 26.

Figure 26. DRAM Controller Function Declarations (Variable Section) (Part 1 of 2)

```
VARIABLE

    address_mux[9..0]: NODE;

% The address multiplexer is a purely combinatorial     %
% function. It selects the row, column, or refresh      %
% address based on the current cycle. The variable      %
% type NODE defines a general-purpose signal that       %
% that allows intermediate logic to be implemented.     %

    ref_address[8..0]: DFF;

% The refresh counter contains a 9-bit counter that     %
% requires 9 registers. The variable type DFF defines   %
% a D flipflop.                                         %

    ref_timer[8..0]   : DFF;
    timeout           : NODE;

% The refresh timer also requires a 9-bit               %
% counter. This function provides a timeout             %
% signal that is assigned to type NODE.                 %
```

Figure 26. DRAM Controller Function Declarations (Variable Section) (Part 2 of 2)

```
    ref_req : DFF;
    control : MACHINE OF BITS (cas[3..0],ras[1..0],dsack,ref)
              WITH STATES (
                 idle          = B"00000011",
                 row0off0      = B"00000111",
                 row0off1      = B"00000111",
                 row0off2      = B"00000111",
                 row0off3      = B"00000111",
                 row0off0siz1  = B"00010101",
                 row0off0siz2  = B"00110101",
                 row0off0siz3  = B"01110101",
                 row0off0siz4  = B"11110101",
                 row0off1siz1  = B"00100101",
                 row0off1siz2  = B"01100101",
                 row0off1siz3  = B"11100101",
                 row0off2siz1  = B"01000101",
                 row0off2siz2  = B"11000101",
                 row0off3siz1  = B"10000101",
                 row1off0      = B"00001011",
                 row1off1      = B"00001011",
                 row1off2      = B"00001011",
                 row1off3      = B"00001011",
                 row1off0siz1  = B"00011001",
                 row1off0siz2  = B"00111001",
                 row1off0siz3  = B"01111001",
                 row1off0siz4  = B"11111001",
                 row1off1siz1  = B"00101001",
                 row1off1siz2  = B"01101001",
                 row1off1siz3  = B"11101001",
                 row1off2siz1  = B"01001001",
                 row1off2siz2  = B"11001001",
                 row1off3siz1  = B"10001001",
                 refresh0      = B"00001110",
                 refresh1      = B"00001110");

% The RAS/CAS generator contains a latch for refresh    %
% request and a state machine called control. The       %
% state names are extracted from the state diagram.     %
```

Step 4: Describe the Major Functions

Each of the major functions is described with AHDL statements. Figure 27 shows the Logic Section of the AHDL TDF. Comments in this section describe the operation of each function.

Figure 27. DRAM Controller Function Descriptions (Logic Section) (Part 1 of 4)

```
BEGIN

%                Address Multiplexer                    %
% The address multiplexer controls the address that is  %
% connected to the DRAM devices. During the idle state  %
% and the RAS-generation states, this multiplexer       %
% selects the row address (sa[21..12]). During the      %
% CAS-generation states, the column address (sa[11..2]) %
% is selected. During refresh cycles, the refresh       %
% address is selected.                                  %

    IF ref THEN
        da[8..0] = ref_address[];
    ELSIF dsack THEN
        da[] = sa[11..2];
    ELSE
        da[] = sa[21..12];
    END IF;

    da[] = address_mux[];

%                Refresh Counter                        %
% The refresh counter increments the refresh address    %
% every refresh cycle. The counter is initialized to    %
% zero by a refresh signal.                             %

    ref_address[].clk = clock;
    ref_address[].clrn = !reset;

    IF timeout THEN
        ref_address[] = ref_address[] + 1;
    ELSE
        ref_address[] = ref_address[];
    END IF;

%                Refresh Timer                          %
% A refresh cycle is required on all 512 rows every     %
% 8 ms. With a Clock frequency of 20 MHz, the number    %
% of cycles between refresh requests is 312. The refresh %
% timer is a free-running counter that counts from 0 to %
% 311. Refresh request is generated when the counter is %
% at 311.                                               %

    ref_timer[].clk = clock;
    ref_timer[].clrn = !reset;

    IF (ref_timer[] == MAX_REF_TIMER) THEN
        ref_timer[] = 0;
        timeout = VCC;
    ELSE
        ref_timer[] = ref_timer[] + 1;
    END IF;
```

Figure 27. DRAM Controller Function Descriptions (Logic Section) (Part 2 of 4)

```
%                RAS/CAS Generator                                  %
% The RAS/CAS generator controls cycling through the               %
% RAS, CAS, and refresh cycles. The RAS/CAS generator              %
% waits in the idle state until either a refresh                   %
% request (refreq) or an address strobe (/as) is                   %
% received. When a refresh is received, /ras1 and /ras0            %
% are asserted for two Clock cycles. When the first cycle          %
% after /as is received, the appropriate /ras is                   %
% asserted. The next Clock cycle asserts the                       %
% appropriate /cas signals based on the offset and                 %
% size of the request. The /dsack signal is also                   %
% asserted during this cycle. The third cycle                      %
% deasserts /ras, /cas, and /dsack, and returns to                 %
% the idle state.                                                  %

    ref_req.clk = clock;
    ref_req = timeout
        # ref_req & control.ref;
    control.clk = clock;
    control.reset = reset;

    CASE control IS
        WHEN idle =>
            IF ref_req THEN
                control = refresh0;
            ELSIF !/as & (sa[1..0] == 0) THEN
                IF sa22 THEN
                    control = row1off0;
                ELSE
                    control = row0off0;
                END IF;
            ELSIF !/as & (sa[1..0] == 1) THEN
                IF sa22 THEN
                    control = row1off1;
                ELSE
                    control = row0off1;
                END IF;
            ELSIF !/as & (sa[1..0] == 2) THEN
                IF sa22 THEN
                    control = row1off2;
                ELSE
                    control = row0off2;
                END IF;
            ELSIF !/as & (sa[1..0] == 3) THEN
                IF sa22 THEN
                    control = row1off3;
                ELSE
                    control = row0off3;
                END IF;
            END IF;
```

Figure 27. DRAM Controller Function Descriptions (Logic Section) (Part 3 of 4)

```
WHEN row0off0 =>
    IF (size[] == 1) THEN
        control = row0off0siz1;
    END IF;
    IF (size[] == 2) THEN
        control = row0off0siz2;
    END IF;
    IF (size[] == 3) THEN
        control = row0off0siz3;
    END IF;
    IF (size[] == 0) THEN
        control = row0off0siz4;
    END IF;
WHEN row0off1 =>
    IF (size[] == 1) THEN
        control = row0off1siz1;
    END IF;
    IF (size[] == 2) THEN
        control = row0off1siz2;
    END IF;
    IF (size[] == 3) THEN
        control = row0off1siz3;
    END IF;
WHEN row0off2 =>
    IF (size[] == 1) THEN
        control = row0off2siz1;
    END IF;
    IF (size[] == 2) THEN
        control = row0off2siz2;
    END IF;
WHEN row0off3 =>
    IF (size[] == 1) THEN
        control = row0off3siz1;
    END IF;
WHEN row1off0 =>
    IF (size[] == 1) THEN
        control = row1off0siz1;
    END IF;
    IF (size[] == 2) THEN
        control = row1off0siz2;
    END IF;
    IF (size[] == 3) THEN
        control = row1off0siz3;
    END IF;
    IF (size[] == 0) THEN
        control = row1off0siz4;
    END IF;
WHEN row1off1 =>
    IF (size[] == 1) THEN
        control = row1off1siz1;
    END IF;
```

Figure 27. DRAM Controller Function Descriptions (Logic Section) (Part 4 of 4)

```
            IF (size[] == 2) THEN
                control = row1off1siz2;
            END IF;
            IF (size[] == 3) THEN
                control = row1off1siz3;
            END IF;
        WHEN row1off2 =>
            IF (size[] == 1) THEN
                control = row1off2siz1;
            END IF;
            IF (size[] == 2) THEN
                control = row1off2siz2;
            END IF;
        WHEN row1off3 =>
            IF (size[] == 1) THEN
                control = row1off3siz1;
            END IF;
        WHEN refresh0 =>
            control = refresh1;
        WHEN OTHERS =>
            control = idle;
    END CASE;

    /dsack = !dsack;
    /ras[] = !ras[];
    /cas[] = !cas[];
END;
```

Conclusion

AHDL provides a wide range of design solutions for Classic, MAX 5000, MAX 7000, and STG EPLDs. You can efficiently implement truth tables, state machines, counters, and complex combinatorial logic. The MAX+PLUS II TTL MacroFunction library provides over 300 TTL, bus, EPLD-optimized, and application-specific functions that can be incorporated into your AHDL files. You can also combine AHDL files with schematic and waveform files and compile and simulate them with MAX+PLUS II software.

Controlling Complex CCD Imaging Systems

with the EPS464 STG EPLD

April 1992, ver. 2

Application Note 25

Introduction

Advances in VLSI-scale microelectronics have allowed manufacturers of video and optical equipment to replace optical film storage media with arrays of light-sensitive electronic imaging elements. These video systems use analog charge-coupled device (CCD) technology to retain image data, and a high-speed digital control system to convert the image into a standard video format. The Altera EPS464 Synchronous Timing Generator (STG) Erasable Programmable Logic Device (EPLD) is the ideal device to implement the necessary digital control logic.

This application note covers the following topics:

- EPS464 architecture
- MAX+PLUS II design entry
- Control requirements for a CCD-based video camera
- National Television System Committee (NTSC) specifications
- Analog signal control
- CCD sensor control
- Implementing the design
- Entering the design
- Fitting the design into an EPS464 EPLD
- Choosing the appropriate EPS464 package

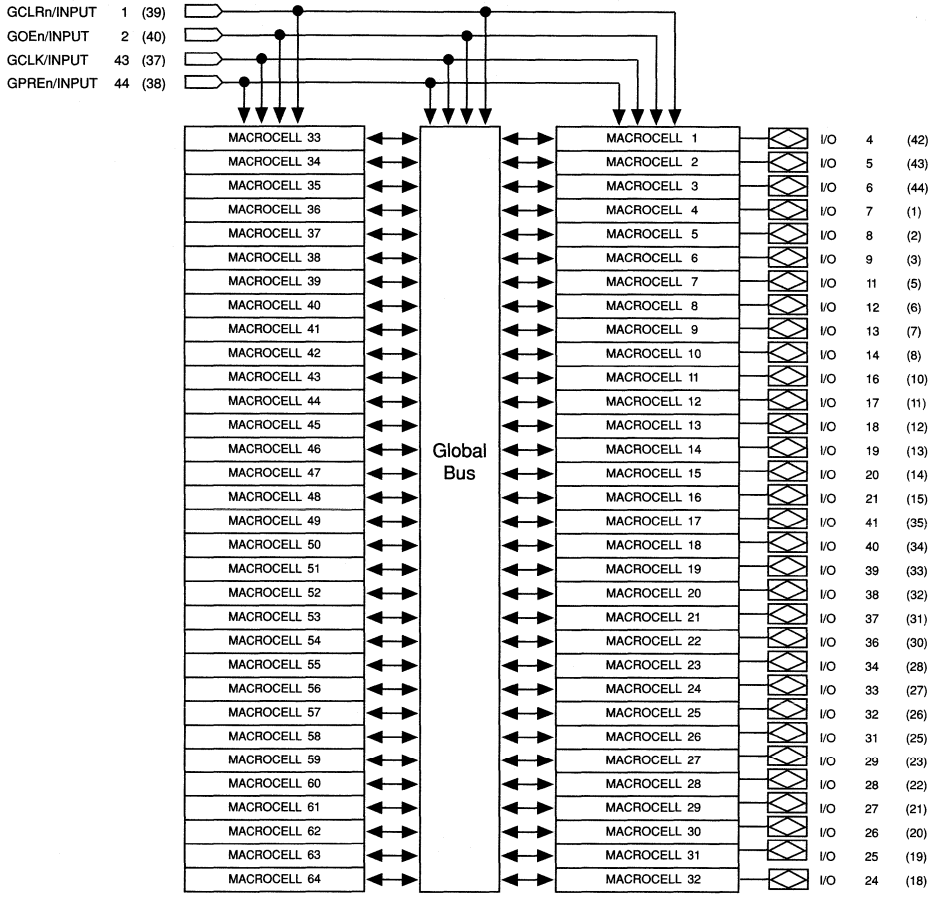
EPS464 Architecture

The EPS464 EPLD is ideally suited for generating complex waveforms, control signals, and state machines. Figure 1 shows a block diagram of the EPS464 EPLD. It contains 64 macrocells optimized for timing and waveform synthesis applications. Thirty-two of these macrocells are connected to I/O pins, while the other 32 are available for buried logic, such as state machine registers and user-defined *n*-bit counters. This large number of macrocells can implement multiple registered and combinatorial functions in a single EPS464. A global bus, which feeds all macrocells within the device, allows system speeds of 66 MHz and prevents placement-dependent interconnect delays between device resources.

The EPS464 macrocell is designed for maximum performance and utility. Four of the five product terms in the macrocell serve dual roles, either as dedicated control or data signals, or as additional shareable expander product terms that can be used by other macrocells. The register control inputs can use product terms for complex control functions or be driven directly from input pins for fast, global control. For more details, see the *EPS464 STG EPLD: Synchronous Timing Generator Data Sheet.*

Figure 1. EPS464 Block Diagram

Numbers in parentheses are for plastic quad flat pack (PQFP) packages.



The user-configurable EPS464 EPLD can accommodate a variety of independent logic functions used in waveform generation and random logic integration applications. Since the device uses EPROM cells, you can easily erase it to perform quick and cost-effective design iterations during development and debug cycles.

The 44-pin EPS464 EPLD is available in windowed ceramic J-lead (JLCC) and one-time-programmable (OTP) plastic J-lead (PLCC) chip carrier packages, as well as plastic quad flat pack (PQFP) packages. Erasable ceramic devices are typically used for prototyping, while lower-cost OTP plastic packages are ideal for larger production quantities. For high-volume

production, Altera offers the MPS464, a mask-programmed logic device (MPLD) version of the EPS464. To generate the MPS464, Altera converts the original design files directly into a masked device, providing substantial cost and time savings.

MAX+PLUS II Design Entry Support

The EPS464 EPLD is supported by the MAX+PLUS II development system, which offers design entry, compilation and logic synthesis, simulation, and programming software in a single integrated package. MAX+PLUS II software runs under Windows 3.0 or higher on a 386-based IBM PC-AT, PS/2, or compatible computer. A Compiler-only version of MAX+PLUS II is also available on Sun SPARCstations.

You can create your MAX+PLUS II design (called a "project" in MAX+PLUS II) with three different entry methods, selecting the most appropriate design tool for each portion of the project, and then combine the portions in a hierarchical project. Schematic entry is made easy with the MAX+PLUS II TTL MacroFunction Library, which contains over 300 logic and TTL functions, including application-specific functions such as phase-locked loops and the `NTSC` function described later in this application note. By using these functional blocks rather than logic primitives, you can greatly enhance productivity and reduce debugging time.

The Altera Hardware Description Language (AHDL) is a sophisticated text-based design entry language that allows you to define state machines, complex transition patterns, truth tables, and Boolean logic expressions. The `NTSC` function described in this application note is created as an AHDL Text Design File (.TDF). *Application Note 22 (Designing with AHDL)* in this handbook and MAX+PLUS II Help provide detailed information on how to use AHDL.

The third design entry method is waveform design entry. It allows you to enter designs by describing the relationships between input and output signals. Waveform design entry is ideal when input and output waveforms are specified during design definition: you draw the timing diagrams, and MAX+PLUS II generates the most efficient logic necessary to create the prescribed output signals.

Overview of CCD Imaging Applications

The market for hand-held video cameras has grown dramatically in the last few years. As these cameras have become more compact, consumers have demanded more features, challenging design engineers to make the cameras smaller and more powerful at the same time. The most common approach to meet these demands has been to develop full-custom ASICs or gate arrays. While this approach has resulted in smaller cameras and a growing number of features, the long time required to develop the chips and the high cost of the engineering work present a major drawback. Also, there is always the possibility that a custom device will not work correctly and must therefore undergo additional design iterations. In the consumer

electronics industry, where "time to market" is everything, such delays often make the difference between a successful product and an "also-ran."

The complex digital control system for a hand-held video camera requires a combination of high speed and high density. Not only must the system retrieve image data from the imaging element, it must also process the analog data into a standard format for broadcast or storage on magnetic tape. All of the functional blocks must work together and be perfectly synchronized.

Most hand-held cameras use a CCD element as the image capture medium. The CCD array does an excellent job of digitizing the incident images at high speed; however, the higher the resolution of the array, the faster the digital control system must operate. An array with a resolution of 910 × 525 pixels generates over 450,000 data points for every image. Since the NTSC standard specifies 30 images per second, the control system must operate at 14.318 MHz.

The EPS464 EPLD offers the perfect solution for this type of application. It combines the high density and high performance of CMOS EPROM technology with the flexibility of Altera's EPLD architecture, allowing multiple design iterations and faster design cycles.

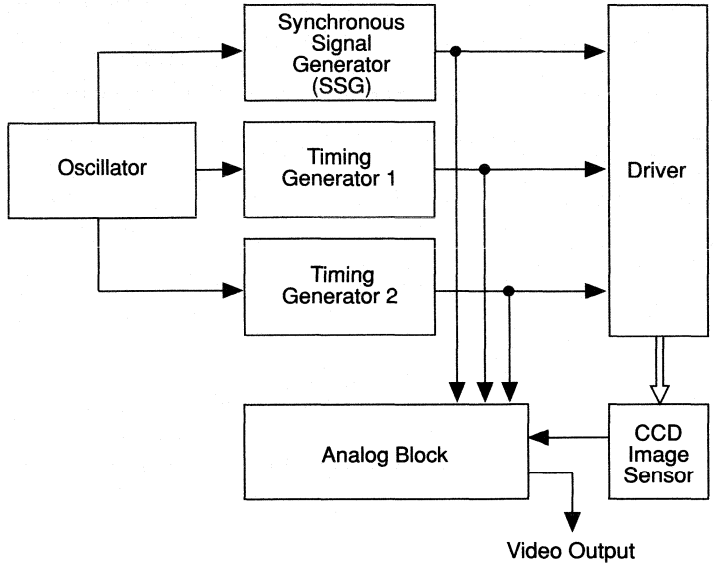
Control System Design

The most challenging part of a digital video-capture design is the control system: it must operate at high speeds to handle the data bandwidth, and it must be highly integrated to create complex waveform and timing signals. Figure 2 shows a typical implementation of a controller for a hand-held 8-mm video camera. The application uses a 910 × 525-bit CCD imaging array and circuitry to format the image data into the NTSC video standard.

The design contains three main blocks: a horizontal (X) counter, a vertical (Y) counter, and a synchronous signal generator (SSG). In Figure 2, the two counters are shown as timing generators that are used to format the video display into pixels per line and lines per screen. The SSG uses these counter values to generate the necessary control patterns and formatting signals.

The size of the counters varies according to the desired video format and the resolution of the display. The NTSC standard specifies an interlaced display scheme; therefore, the system must count to one-half of the 525 lines in the display, or 262.5 lines. However, the count is difficult to control since it depends on half-Clock cycles. To solve this problem, the screen is divided into a series of 1,050 half-lines, with each frame using 525 half-lines. By offsetting the display by one half of a horizontal line and displaying the two "half-pictures" in rapid succession, the interlace is transparent to the viewer, and the result is a very good representation of the original picture. This approach takes advantage of the limitations of human sight: although the human eye is good at distinguishing a wide range of colors, it

Figure 2. Typical CCD Controller



is not so good at distinguishing small or rapid movements. Consequently, an image updated 30 times per second appears to have fluid motion.

Given these system requirements, the horizontal counter needs 10 bits to count the 910 pixels per line and the vertical counter needs 11 bits to count the 1,050 lines per screen. The 1,050 half-line approach allows the counters to use integers rather than a cumbersome scheme of half-Clock cycles.

NTSC Specification

The NTSC standard has three key specifications: lines per screen, line period, and screens per second. Since each image must be synchronized by the display hardware, and any variation from the standard would cause the image to roll on the screen, the lines-per-screen specification is fixed at 525. However, not all 525 lines are actually shown on the display. Twenty are used for formatting and equalization intervals, while the remaining 505 are dedicated to image data.

The second specification is the amount of time needed to display a line. Each line is encoded during a 63.556-µs period established by the incoming video signal. To encode the 910 horizontal pixels in the application, the control system must divide each horizontal time period into 910 individual segments and sample the analog video signal at each of those points. Each sample corresponds to the clocking out of the next word of analog data from the CCD array. With a horizontal time period of 63.556 µs, the sampling rate is 14.318 MHz.

The third NTSC specification requires an image to be updated 30 times per second, requiring a refresh every 33.33 ms. Since the NTSC is an interlaced format with two frames per image, the actual frame refresh rate is 60 Hz. This rate is easily achieved with the EPS464 EPLD as the digital control system.

Analog Signal Control

The final video output is an analog signal that combines analog voltages representing intensity and color information with digital formatting pulses. Although digital and analog levels are combined with a special-purpose analog mixing device, mixing is actually performed under the control of the digital system. The result is a single composite video signal for broadcast or storage on tape.

Figure 3 shows a condensed diagram of the actual analog video signal. During the first portion of the waveform, it appears to be a well-behaved digital signal. There are narrow pulses, but no analog levels. This section of the waveform corresponds to the first four phases in the NTSC standard: pre-equalization, serration, post-equalization, and subphase. During these first four phases (labeled i, ii, iii, and iv) the display unit is stabilizing and equalizing, but no display is drawn on the screen. During the subsequent "video-active" phase (v), the actual image data is encoded as a varying analog signal and is displayed during the horizontal sweep of the display device.

Figure 3. NTSC Analog Signal

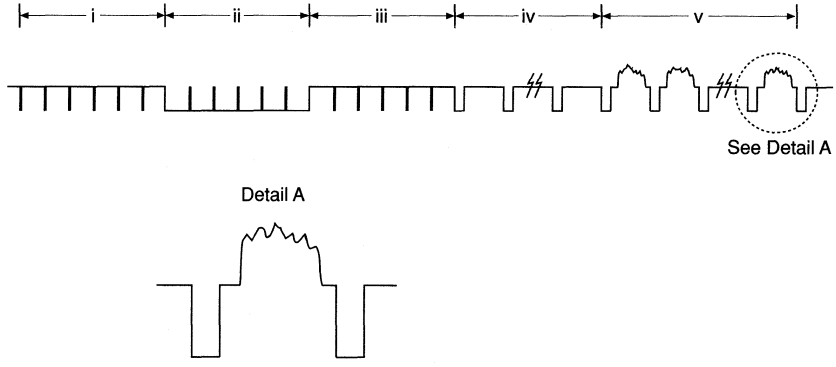
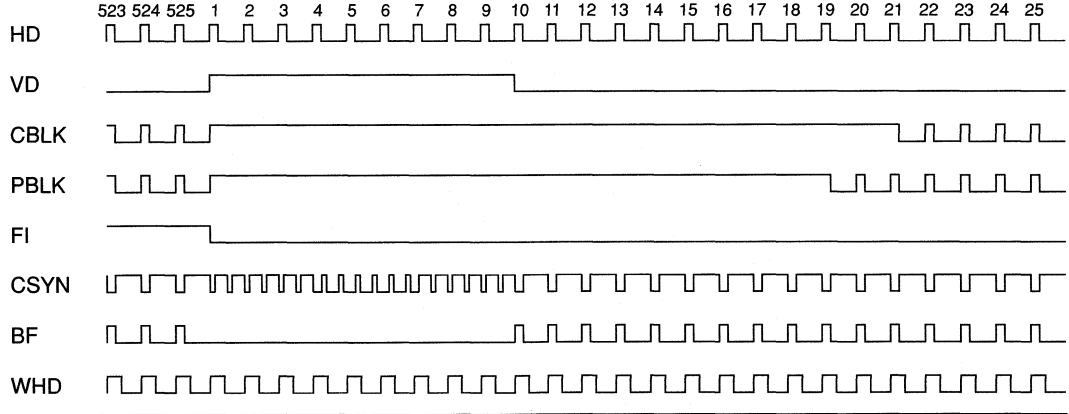


Figure 3 also shows an enlarged portion of a single horizontal line. This figure contains image information, as shown in the analog voltage sweep. The narrow vertical spikes at the ends of the analog sweep are the horizontal synchronization pulses—commonly referred to as the horizontal display (HD) signal—which indicate the start of a horizontal sweep of the video display. The analog signal between each of these pulses represents an encoding of the image data, and is spread across the entire horizontal sweep of the display unit.

During the first four phases, the pulses occur twice as often as during the video-active phase. These pulses are the half-line pulses. The wider pulses in the video-active phase represent two half-lines combined into a full-line for display purposes.

The video signal is a composite of the analog data and the digital control signals. Figure 4 shows a sample of a digital portion of the NTSC digital signals that are combined with the image information to create the analog video signal.

Figure 4. NTSC Digital Signal Set



CCD Sensor Control

Each of the positive pulses on HD represents the start of a line, and every positive pulse on vertical display (VD) represents another frame. In the NTSC-specified interlacing scheme, each image consists of two frames; therefore, each full image on the screen has two VD pulses. More importantly, the digital patterns are slightly different between the first and second frames since they are offset by one half-row. After every HD pulse, the control system begins clocking data out of the next row in the CCD element. This function corresponds to incrementing the address. After every VD pulse, the control system returns to the top of the CCD element and begins retrieving a new frame.

The interlacing scheme was originally used because the electron gun in a CRT could not deflect quickly enough to draw a full image accurately. By interlacing the two images, the system presented the same video information without placing additional demands on the display hardware. The CCD imaging element, on the other hand, operates at extremely high speed and can clock out the image data much faster than is necessary for the NTSC standard. The element must accommodate the format, however, and must be divided into two frames to simulate the interlacing required by the CRT.

Each HD pulse is translated to retrieve either odd or even lines from the CCD element. The first HD pulse starts the image at the upper-left-hand corner of the display and clocks out the first physical line in the sensor. This process is repeated for all odd lines in the display; then a pulse on VD resets the horizontal counter. The system then starts on the second frame and returns even lines of image data, which are combined with the odd lines of data to form a full image made up of two frames.

Design Implementation

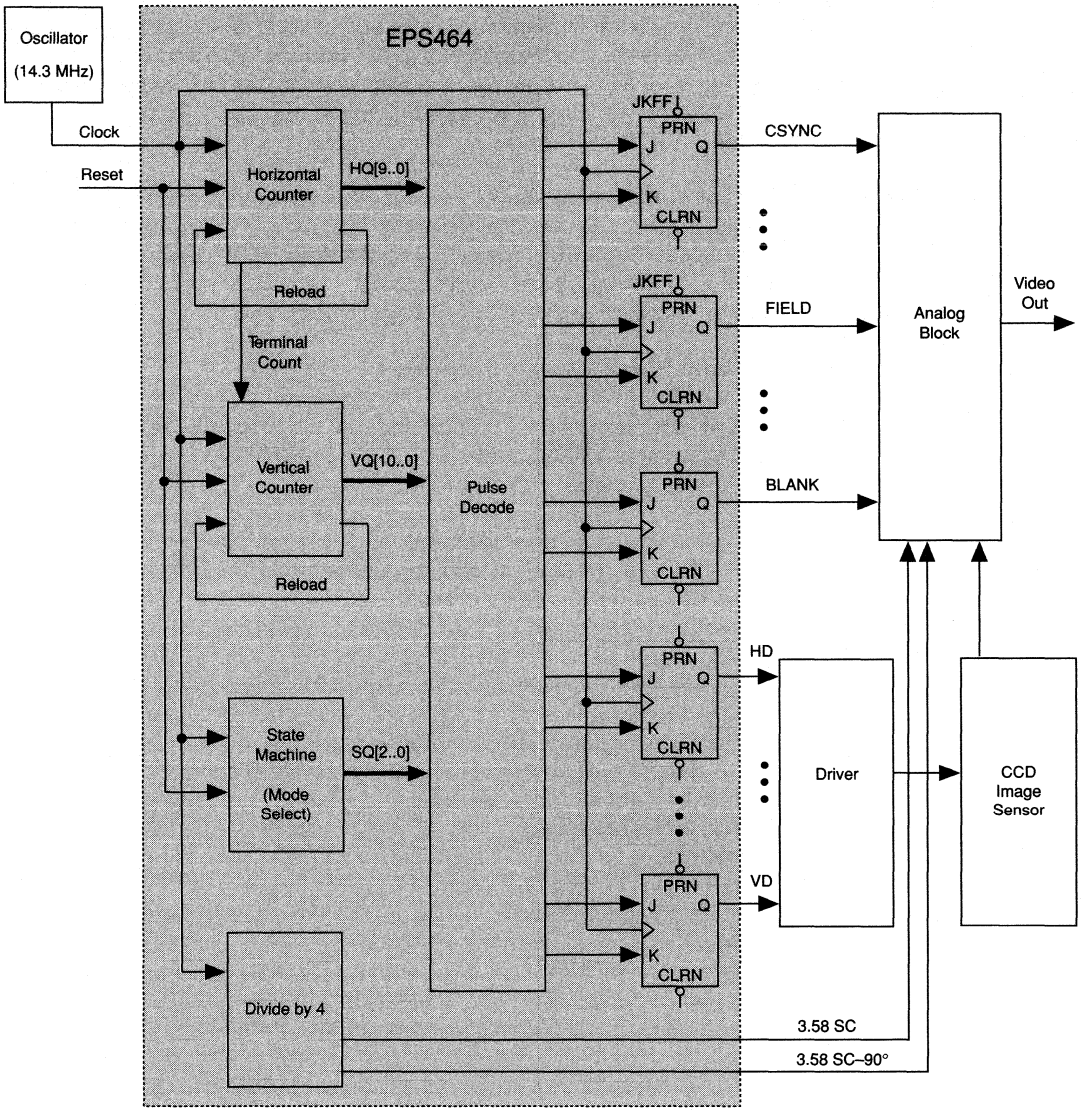
You can implement this CCD imaging system control in several ways. One way is to use a PROM and a counter. The PROM contains bit patterns for the various control signals (CSYNC, HD, VD, etc.), which are clocked out at the highest system Clock frequency (14.318 MHz), with each subsequent address generating the necessary control words. Since no high-level design tools exist to support this approach, you must hand-encode the control words into the PROM. This multi-device solution requires many ICs.

Another common approach is to implement the entire control system with discrete TTL ICs. In this approach, the counters and associated decode logic, shown in Figure 2, are replaced by a variety of SSI and MSI-components. The SSG—an off-the-shelf circuit—typically would not be replaced with TTL devices. Still, this solution requires many more than three ICs. In addition, you may not be able to implement a special feature if no SSG is available that meets the system requirements. Although you can replace the SSG with TTL building blocks, the dramatic increase in the number of devices required can pose a problem.

The custom solution provided by an ASIC or gate array requires long development times and high non-recurring engineering (NRE) charges, excluding all but the largest firms from this option. In addition, once the circuit is fabricated, it cannot be modified for minor feature changes. The ASIC or gate-array solution is therefore viable only for long-term, large production runs.

By far the best choice is a programmable logic device. Figure 5 shows this application implemented in the Altera EPS464 EPLD. All five subfunctions (horizontal counter, vertical counter, state machine, pulse decode module, and clock divider) fit into a single EPS464 and can be easily modified to meet the requirements of any application.

Figure 5. EPS464 Implementation of the CCD Imaging System Control Application



This project requires 21 macrocells for the 2 counters, 1 macrocell per output signal, and 3 macrocells for the state machine. In this application, 6 output signals are implemented, requiring a total of 35 macrocells. These complex outputs are created using JK flipflops with multiple-product-term inputs. Each of these product terms either sets (i.e., to one) or resets (i.e., to zero) the output waveforms, based on the counter values and phases of the NTSC signal.

Design Entry

This application note shows you how to implement the NTSC control function with an AHDL TDF. The complete NTSC.TDF is also provided as an application-specific function in the MAX+PLUS II TTL MacroFunction Library. You can use it as a template to generate your own custom control function. Figure 6 at the end of this application note shows NTSC.TDF.

The application consists of three parts:

- Two counters, one for vertical and one for horizontal scanning
- The pulse decode module
- A state machine

The two counters are used to control the horizontal and vertical scanning of the CCD image element and to generate the NTSC synchronization format. Since you can easily change terminal count and reset values in a TDF, you can quickly adapt your design to new system requirements.

Figure 6 shows the TDF that describes a complete NTSC project. The counters are designed with a Gray-code sequence for greater reliability. They include terminal count conditions that cause them to synchronously reset to zero, as well as a global asynchronous Reset control signal (master Reset). The counters also use toggle flipflops (`TFF` primitives), which provide the most efficient way to create counters in the Altera EPLD architecture.

The state machine is implemented in the Variable Section with a State Machine Declaration that declares the name, states, and bits of the state machine. Transitions are based on the comparison of counter values and constants. The outputs are generated as JK flipflops defined in each state. The comments in the file provide information about the project.

Design Fitting

When the MAX+PLUS II Compiler processes and synthesizes a project, it optimizes it for the EPS464 EPLD by applying fitting and synthesis rules that are specified for the EPS464 architecture. The most efficient fit and maximum performance are therefore ensured.

Routing and fitting is 100% automatic. The EPS464 EPLD uses a predictable macrocell architecture that allows the software to compile and fit the project in minutes. MAX+PLUS II simulation and design analysis tools

provide worst-case performance data for the project, based on actual characterization of the architecture.

EPS464 Packages

Since hand-held video cameras must be as compact as possible, the 44-pin PQFP EPS464 EPLD is an excellent choice. This package combines low cost with a small footprint. Compared to the discrete device implementation, it uses one-third the number of ICs, significantly reducing the required board space. Also, since you only need one device, you will have reduced power consumption and increased reliability. This type of integration makes the EPS464 an ideal candidate for a wide variety of hand-held or compact systems that require fast, complex controllers.

Conclusion

Video control system designers can implement digital control logic in a variety of ways. TTL and PROM-based approaches are used for prototyping and low-volume applications, but are awkward and require a large number of devices. ASIC solutions are practical for high-volume applications, but require a long design cycle and cannot be modified once production begins. The EPS464 EPLD combines the advantages of the TTL and ASIC approaches without these drawbacks. The high-density EPS464 integrates complete video control designs in a single device. Being user-configurable, it allows fast design cycles and modifications. In addition, MAX+PLUS II design tools provide efficient design entry, compilation, and simulation support.

Figure 6. NTSC.TDF (Part 1 of 9)

```
%
|   NTSC Waveform Generator
|   03/16/92    version 1.1
|
%

TITLE "NTSC Waveform Generator";

%
|   Functional Description
|
|   The NTSC MacroFunction uses Clock and active-low Reset input signals to
|   generate a digital representation of the NTSC control signals CSYNC, HD, VD,
|   FIELD, BURST, and BLANK. These signals can be used to control video imaging
|   or display equipment.
|
|   To implement a complete NTSC pattern, this macrofunction contains a 6-state
|   state machine with the following states: power-up (_power_up),
|   pre-equalization (_pre_eq), serration (_serra), post-equalization (_post_eq),
|   sub-phase (_sub_ph), and video-active (_vid_act). During each of these
|   distinct states, the outputs are generated according to a different
|   set of rules, and the state machine controls the output generation.
|
|   Each horizontal line contains 910 pixels. The NTSC MacroFunction treats each
|   line as a pair of half-lines, so the HCNT_ROLL_OVER constant is set to 453
|   (one less than the half-line value of 454, which provides 910 pixels when
|   multiplied by two).
|
|   The constant VCNT_ROLL_OVER is set to 525, representing the 525 lines per
|   screen specified by the NTSC standard. Each screen contains two frames
|   that each have 262.5 lines to provide interlacing. Since non-integer
|   counters are very hard to control, the system counts to 525 half-lines,
|   then repeats the count sequence for the second frame. The two frames are
|   offset by one half-line so that a complete picture is formed with the
|   analog video data from both of the frames.
|
|   For more detailed information on NTSC pattern generation, consult the
|   NTSC standards or any textbook on video applications.
%


%
|   PATTERN CONSTANTS
|
|   Constants shown for phase conditions (eg: START_PRE_EQ) are decoded off the
|   vertical counter. State transitions are based on the line that is currently
|   being decoded. The values are shown as binary numbers (denoted by the
|   B"xxxx" format), and represent the Gray-code values that correspond to the
|   decimal values shown in a comment on each line.
```

Figure 6. NTSC.TDF (Part 2 of 9)

```
|   Constants for the J and K inputs (e.g., HD_J) are also Gray-code values,
|   with the corresponding decimal value shown in a comment. These constants
|   are compared with the horizontal counter to make pulses in the output
|   signals.
|
|   The constants with names that begin with "START" are starting points for the
|   correspondingly named NTSC phase. For this model we assume that the NTSC
|   standard has 5 phases: pre-equalization (START_PRE_EQ), serration
|   (START_SERRA), post-equalization (START_POST_EQ), sub-phase (START_SUB_PH),
|   and video-active (START_VID_ACT). Each phase starts at defined vertical lines,
|   which are set by the first 5 constants given below. Since they are decoding a
|   Gray-code counter, the equivalent decimal value is shown in a comment (e.g.,
|   video active starts on the 20th line in the scan).
%

CONSTANT START_PRE_EQ  = B"0000000000";   % line 0       D"0"    %
CONSTANT START_SERRA   = B"0000000101";   % line 3       D"6"    %
CONSTANT START_POST_EQ = B"0000001010";   % line 6       D"12"   %
CONSTANT START_SUB_PH  = B"0000011011";   % line 9       D"18"   %
CONSTANT START_VID_ACT = B"0000111100";   % line 20      D"40"   %

%
|   These constants define the transitions on the various NTSC output signals
|   during the 5 phases of the NTSC standard. In each case, the constant shown
|   represents a value that is compared to the horizontal pixel counter. In the
|   case of the CSYNC signal the output depends on the current phase, so 5
|   constants are given to completely specify the operation. The hd, blank, and
|   burst signals behavethe same in all 5 states, so only one constant pair is
|   given. Note that a "J" and a "K" constant are always given because the output
|   pulses on all the output signals must be turned on and then turned
|   off again at specific horizontal count values. The "J" is routed to the J
|   input of a JK flipflop that asserts the output when true, and the "K" is
|   routed to the K input to a JK flipflop that deasserts the output
|   when true.
|
|   By setting the value of CSYNC_PRE_EQ_J to the Gray-code equivalent of
|   decimal 36, the csync signal is asserted at the 36th horizontal bit.
|   By setting CSYNC_PRE_EQ_K equal to the Gray-code equivalent to decimal 1,
|   csync is deasserted at the first horizontal bit. During the pre-
|   equalization phase, 6 "half-lines" exist (i.e., the constant
|   START_SERRA is set to the Gray-code equivalent of decimal 6), so the
|   csync signal will make 6 pulses during pre-equalization phases. For
|   this reason, the falling edge at the first point in a horizontal scan ends
|   the high-level signal for the previous line. Since csync is an active-low
|   signal, a pulse is created that lasts from the 1st to the 36th
|   bit on the horizontal scan, and then repeats for a total of 6 lines.
|
|   The active-low csync signal behaves as follows during the pre-equalization
|   phase: The first (falling) edge occurs when the horizontal counter is equal
|   to the Gray-code equivalent of "1" and the second (rising) edge occurs when
|   the horizontal counter is equal to the Gray-code equivalent of "36."
```

Figure 6. NTSC.TDF (Part 3 of 9)

```
|   These edges should technically occur at "0" and "35," but the transitions are
|   all offset by one to avoid synchronization problems with the boundary between
|   the two frames in the interlaced screen format.
|
|   1      36                   454/1     36                   454/1
|           ________________|  |____        ________________|  |____
|   |      |                |  |    |      |                |  |    |
|   |      |                   |    |      |                   |    |
|   |______|                   |____|      |                   |_ . .
|
|   {-----  first line  -----}{-----  second line  -----}
|
|   Counter values are compared to constants in the "WHEN_pre_eq" clause
|   in the CASE line-decode Case Statement. Corresponding WHEN clauses
|   are provided for all 5 NTSC phases. Each WHEN clause defines each of
|   the 6 output signals during each of the 5 states.
%

CONSTANT CSYNC_PRE_EQ_J    = B"000110110";    %   D"36"   %
CONSTANT CSYNC_PRE_EQ_K    = B"000000001";    %   D"1"    %
CONSTANT CSYNC_SERRA_J     = B"101000100";    %   D"391"  %
CONSTANT CSYNC_SERRA_K     = B"000000001";    %   D"1"    %

CONSTANT CSYNC_POST_EQ_J   = B"000110110";    %   D"36"   %
CONSTANT CSYNC_POST_EQ_K   = B"000000001";    %   D"1"    %

CONSTANT CSYNC_SUB_PH_J    = B"001100110";    %   D"68"   %
CONSTANT CSYNC_SUB_PH_K    = B"000000001";    %   D"1"    %

CONSTANT CSYNC_VID_ACT_J   = B"001100110";    %   D"68"   %
CONSTANT CSYNC_VID_ACT_K   = B"000000001";    %   D"1"    %

CONSTANT HD_J              = B"001110110";    %   D"91"   %
CONSTANT HD_K              = B"000000001";    %   D"1"    %

CONSTANT BLANK_J           = B"011010111";    %   D"154"  %
CONSTANT BLANK_K           = B"000000001";    %   D"1"    %

CONSTANT BURST_J           = B"001000001";    %   D"126"  %
CONSTANT BURST_K           = B"001101110";    %   D"75"   %

%
|   COUNTER CONSTANTS
|
|   These constants specify rollover and terminal count values. 453 is the end
|   of the first half-line. Half-lines are used to accommodate the interlacing
|   scheme specified by NTSC and allow you to use an integer value to count
|   the lines per screen. Full lines would require the function to count to
|   262.5 (a half-line) and then switch frames. It is difficult to count in
|   non-integer mode with digital logic.
```

Figure 6. NTSC.TDF (Part 4 of 9)

```
|   The VCNT_ROLL_OVER value of 525 is the end of the first frame in the two-frame
|   interlaced model. These values can be changed to match other video display
|   formats.
%

CONSTANT HCNT_ZERO          = B"000000000";    %  D"0" for start value    %
CONSTANT HCNT_ROLL_OVER     = B"100100111";    %  D"453" for hhalf        %
CONSTANT VCNT_ROLL_OVER     = B"1100001010";   %  D"525"                  %

%
|   No particular device is specified for this macrofunction. In its current
|   form, it uses 31 macrocells in the EPS464 EPLD, and 32 macrocells in
|   MAX 5000 and MAX 7000 EPLDs.
%


DESIGN IS "ntsc"
   DEVICE IS "AUTO";

SUBDESIGN ntsc
(
    clock           : INPUT = GND; % System global Clock          %
    reset           : INPUT = VCC; % System Reset (active low)    %

    % NTSC output signals  %
    csync           : OUTPUT;
    hd              : OUTPUT;
    vd              : OUTPUT;
    fld             : OUTPUT;
    blank           : OUTPUT;
    burst           : OUTPUT;
)

%
|   The line_decode state machine controls the NTSC waveforms. The
|   outputs csync, hd, vd, fld, blank, and burst are all created with
|   JK flipflops.
%
VARIABLE

    line_decode: MACHINE OF BITS  (q[2..0] )
                 WITH STATES (_power_up,_pre_eq,_serra,_post_eq,_sub_ph,_vid_act);

    gclk            :   NODE;   % Global Clock node    %
    grst            :   NODE;   % Global Reset node    %
    csync_ff        :   JKFF;
    hd_ff           :   JKFF;
    vd_ff           :   JKFF;
    fld_ff          :   JKFF;
    blank_ff        :   JKFF;
    burst_ff        :   JKFF;
```

Figure 6. NTSC.TDF (Part 5 of 9)

```
    pwr_up          :   NODE;
    cnt_reset       :   NODE;

    %
    |   These counter variables have 11 bits for lines per screen (10 in the
    |   counter, and line_ff as the terminal bit), and 10 bits for pixels per line
    |   (9 in the counter and h_odd as the terminal bit).
    %

    v[9..0]         :   TFF;    % Vertical counter bits       %
    h[8..0]         :   TFF;    % Horizontal counter bits     %
    line_ff         :   TFF;    % First/second halfline_ff    %
    h_odd           :   TFF;    % Odd horizontal bit          %
    field_ff        :   TFF;    % Odd/even field_ff indicator %
    vcnt_reset      :   NODE;   % Terminal/reset condition    %
    hcnt_reset      :   NODE;   % Terminal/reset condition    %

BEGIN

    gclk = GLOBAL(clock);
    grst = reset;

    %
    |   Set up flipflop and state machine Clocks and Clears
    %

    line_decode.clk         =   gclk;
    line_decode.reset       =   !grst;

    v[].clk                 =   gclk;
    h[].clk                 =   gclk;
    h_odd.clk               =   gclk;
    field_ff.clk            =   gclk;
    line_ff.clk             =   gclk;

    vd_ff.clk               =   gclk;
    hd_ff.clk               =   gclk;
    csync_ff.clk            =   gclk;
    fld_ff.clk              =   gclk;
    blank_ff.clk            =   gclk;
    burst_ff.clk            =   gclk;
```

Figure 6. NTSC.TDF (Part 6 of 9)

```
%
|   Define state transitions.
%

CASE  line_decode  IS

    WHEN _power_up =>
         pwr_up          =   VCC;
         cnt_reset       =   VCC;
         csync_ff.j      =   VCC;
         hd_ff.j         =   VCC;
         vd_ff.k         =   VCC;
         fld_ff.k        =   VCC;
         blank_ff.k      =   VCC;
         burst_ff.j      =   VCC;
         line_decode     =   _pre_eq;

    WHEN _pre_eq    =>
        csync_ff.j =   (h[] == CSYNC_PRE_EQ_J);
        csync_ff.k =   (h[] == CSYNC_PRE_EQ_K);
        vd_ff.k    =   VCC;
        hd_ff.j    =   ((h[] == HD_J) & !line_ff & !field_ff)
                   #   ((h[] == HD_J) & !line_ff & field_ff);
        hd_ff.k    =   ((h[] == HD_K) & !line_ff & !field_ff)
                   #   ((h[] == HD_K) & !line_ff & field_ff);
        fld_ff.k   =   VCC;
        blank_ff.k =   VCC;
        burst_ff.j =   VCC;

        IF v[] == START_SERRA  THEN
            line_decode = _serra ;
        END IF;

    WHEN _serra     =>
        csync_ff.j =   (h[] == CSYNC_SERRA_J);
        csync_ff.k =   (h[] == CSYNC_SERRA_K);
        hd_ff.j    =   ((h[] == HD_J) & !line_ff & !field_ff)
                   #   ((h[] == HD_J) & !line_ff & field_ff);
        hd_ff.k    =   ((h[] == HD_K) & !line_ff & !field_ff)
                   #   ((h[] == HD_K) & !line_ff & field_ff);
        vd_ff.k    =   VCC;
        fld_ff.j   =   VCC;
        blank_ff.k =   VCC;
        burst_ff.j =   VCC;

        IF v[] == START_POST_EQ  THEN
            line_decode = _post_eq ;
        END IF;
```

Figure 6. NTSC.TDF (Part 7 of 9)

```
WHEN _post_eq  =>
    csync_ff.j =   h[] == CSYNC_POST_EQ_J;
    csync_ff.k =   h[] == CSYNC_POST_EQ_K;
    hd_ff.j    =   ((h[] == HD_J) & !line_ff & !field_ff)
               #   ((h[] == HD_J) & !line_ff & field_ff);
    hd_ff.k    =   ((h[] == HD_K) & !line_ff & !field_ff)
               #   ((h[] == HD_K) & !line_ff & field_ff);
    vd_ff.k    =   VCC;
    fld_ff.j   =   VCC;
    blank_ff.k =   VCC;
    burst_ff.j =   VCC;

    IF v[] == START_SUB_PH  THEN
        line_decode = _sub_ph ;
    END IF;

WHEN _sub_ph   =>
    csync_ff.j =   ((h[] == CSYNC_SUB_PH_J) & !line_ff & !field_ff)
               #   ((h[] == CSYNC_SUB_PH_J) & !line_ff & field_ff);
    csync_ff.k =   ((h[] == CSYNC_SUB_PH_K) & !line_ff & !field_ff)
               #   ((h[] == CSYNC_SUB_PH_K) & !line_ff & field_ff);
    hd_ff.j =   ((h[] == HD_J) & !line_ff & !field_ff)
               #   ((h[] == HD_J) & !line_ff & field_ff);
    hd_ff.k =   ((h[] == HD_K) & !line_ff & !field_ff)
               #   ((h[] == HD_K) & !line_ff & field_ff);
    vd_ff.j    =   VCC;
    fld_ff.j   =   VCC;
    burst_ff.j =   ((h[] == BURST_J) & !line_ff & !field_ff)
               #   ((h[] == BURST_J) & !line_ff & field_ff);
    burst_ff.k =   ((h[] == BURST_K) & !line_ff & !field_ff)
               #   ((h[] == BURST_K) & !line_ff & field_ff);
    blank_ff.k =   VCC;

    IF v[] == START_VID_ACT  THEN
        line_decode = _vid_act;
    END IF;

WHEN _vid_act  =>
    csync_ff.j =   ((h[] == CSYNC_VID_ACT_J) & !line_ff & !field_ff)
               #   ((h[] == CSYNC_VID_ACT_J) & !line_ff & field_ff);
    csync_ff.k =   ((h[] == CSYNC_VID_ACT_K) & !line_ff & !field_ff)
               #   ((h[] == CSYNC_VID_ACT_K) & !line_ff & field_ff);
    hd_ff.j    =   ((h[] == HD_J) & !line_ff & !field_ff)
               #   ((h[] == HD_J) & !line_ff & field_ff);
    hd_ff.k    =   ((h[] == HD_K) & !line_ff & !field_ff)
               #   ((h[] == HD_K) & !line_ff & field_ff);
    vd_ff.j    =   VCC;
    fld_ff.j   =   VCC;
    burst_ff.j =   ((h[] == BURST_J) & !line_ff & !field_ff)
               #   ((h[] == BURST_J) & !line_ff & field_ff);
    burst_ff.k =   ((h[] == BURST_K) & !line_ff & !field_ff)
               #   ((h[] == BURST_K) & !line_ff & field_ff);
```

Figure 6. NTSC.TDF (Part 8 of 9)

```
    blank_ff.j      =   ((h[] == BLANK_J) & !line_ff & !field_ff)
                    #   ((h[] == BLANK_J) & !line_ff & field_ff);
    blank_ff.k      =   ((h[] == BLANK_K) & !line_ff & !field_ff)
                    #   ((h[] == BLANK_K) & !line_ff & field_ff);

    IF v[] == START_PRE_EQ  THEN
        line_decode = _pre_eq;
    END IF;

END CASE;

%
|   Establish the output connections.
%

csync       =   csync_ff;
hd          =   hd_ff;
vd          =   vd_ff;
fld         =   fld_ff;
burst       =   burst_ff;
blank       =   blank_ff;

%
|   These equations are for line_ff and field_ff. When the horizontal counter
|   is in the first half of the scan-line, the line_ff output is low. A
|   high indicates that the counter is in the second half of the scan-line. To
|   make an efficient circuit, the vertical counter uses line_ff as the
|   terminal bit. The field_ff is used to track the current field. During odd
|   fields (1st physical field), the field_ff output is low. During even
|   fields, it is high.
%
line_ff     =   (h[] == HCNT_ROLL_OVER);
field_ff    =   (v[] == VCNT_ROLL_OVER & h[] == HCNT_ROLL_OVER);
%
|   In the Gray-code equations, each bit toggles when the previous bit is 1,
|   the odd bit is 1, and all other bits between them are 0.
%
IF !field_ff THEN

    v[] = (((v[], line_ff) == B"X1000000001") & !vcnt_reset # vcnt_reset & v9,
        ((v[], line_ff)    == B"XX100000001") & !vcnt_reset # vcnt_reset & v8,
        ((v[], line_ff)    == B"XXX10000001") & !vcnt_reset # vcnt_reset & v7,
        ((v[], line_ff)    == B"XXXX1000001") & !vcnt_reset # vcnt_reset & v6,
        ((v[], line_ff)    == B"XXXXX100001") & !vcnt_reset # vcnt_reset & v5,
        ((v[], line_ff)    == B"XXXXXX10001") & !vcnt_reset # vcnt_reset & v4,
        ((v[], line_ff)    == B"XXXXXXX1001") & !vcnt_reset # vcnt_reset & v3,
        ((v[], line_ff)    == B"XXXXXXXX101") & !vcnt_reset # vcnt_reset & v2,
        ((v[], line_ff)    == B"XXXXXXXXX11") & !vcnt_reset # vcnt_reset & v1,
        ((v[], line_ff)    == B"XXXXXXXXX0") & !vcnt_reset # vcnt_reset & v0
        )
        & (h[] == HCNT_ROLL_OVER);
ELSE
```

Figure 6. NTSC.TDF (Part 9 of 9)

```
        v[] = (((v[],!line_ff) == B"X1000000001") & !vcnt_reset # vcnt_reset & v9,
            ((v[], !line_ff)   == B"XX100000001") & !vcnt_reset # vcnt_reset & v8,
            ((v[], !line_ff)   == B"XXX10000001") & !vcnt_reset # vcnt_reset & v7,
            ((v[], !line_ff)   == B"XXXX1000001") & !vcnt_reset # vcnt_reset & v6,
            ((v[], !line_ff)   == B"XXXXX100001") & !vcnt_reset # vcnt_reset & v5,
            ((v[], !line_ff)   == B"XXXXXX10001") & !vcnt_reset # vcnt_reset & v4,
            ((v[], !line_ff)   == B"XXXXXXX1001") & !vcnt_reset # vcnt_reset & v3,
            ((v[], !line_ff)   == B"XXXXXXXX101") & !vcnt_reset # vcnt_reset & v2,
            ((v[], !line_ff)   == B"XXXXXXXXX11") & !vcnt_reset # vcnt_reset & v1,
            ((v[], !line_ff)   == B"XXXXXXXXX0") & !vcnt_reset # vcnt_reset & v0
            )
            & (h[] == HCNT_ROLL_OVER);

    END IF;

    vcnt_reset = (v[] == VCNT_ROLL_OVER # cnt_reset);

    h_odd      = h_odd # !pwr_up;

    h[]        = (((h[], h_odd) == B"X100000001") & !hcnt_reset # hcnt_reset & h8,
                  ((h[], h_odd)  == B"XX10000001") & !hcnt_reset # hcnt_reset & h7,
                  ((h[], h_odd)  == B"XXX1000001") & !hcnt_reset # hcnt_reset & h6,
                  ((h[], h_odd)  == B"XXXX100001") & !hcnt_reset # hcnt_reset & h5,
                  ((h[], h_odd)  == B"XXXXX10001") & !hcnt_reset # hcnt_reset & h4,
                  ((h[], h_odd)  == B"XXXXXX1001") & !hcnt_reset # hcnt_reset & h3,
                  ((h[], h_odd)  == B"XXXXXXX101") & !hcnt_reset # hcnt_reset & h2,
                  ((h[], h_odd)  == B"XXXXXXXX11") & !hcnt_reset # hcnt_reset & h1,
                  ((h[], h_odd)  == B"XXXXXXXX0") & !hcnt_reset # hcnt_reset & h0
                  );

    hcnt_reset = (h[] == HCNT_ROLL_OVER) # pwr_up # cnt_reset;

END;
```

EPLD/MPLD Design Guidelines

April 1992, ver. 2

Application Note 26

Introduction

Altera's EPLD-to-MPLD conversion program provides a turn-key solution for converting an EPLD design into a mask-programmed logic device (MPLD). The combination of Altera EPLDs and MPLDs provides the quick development and production ramp-up times of EPLDs and the lower unit prices of MPLDs for high-volume production.

Drop-in compatibility is the cornerstone of Altera's conversion program. Altera guarantees that MPLDs will meet the worst-case parameters of the original EPLDs, including Clock-to-output times, propagation delays, and setup and hold time specifications. MPLDs are carefully designed so that their output drive characteristics also closely match the performance of equivalent EPLDs.

Since MPLD power consumption (approaching zero standby current) is dramatically lower than that of equivalent EPLDs, you can use MPLDs in low-power applications. Refer to individual MPLD data sheets for I_{CC} vs. frequency curves.

As part of the EPLD-to-MPLD conversion program, Altera manages all aspects of device testability, including design-for-testability (DFT) and automatic test vector generation (ATVG). Altera's patented approach to DFT places no restrictions on the designer's creativity. Therefore, no last-minute changes are required to enhance testability, and no simulation vectors need to be generated.

This application note identifies potential logic and timing irregularities in EPLD designs. Understanding possible pitfalls and carefully following the guidelines presented in this application note will help to ensure that your EPLD design is converted into a flawless MPLD. The following subjects are included:

- ❑ Five types of Clocks:
  - Global Clocks
  - Gated Clocks
  - Multi-level logic Clocks
  - Ripple Clocks
  - Multi-Clock systems
- ❑ Clear and Preset configurations
- ❑ Registering combinatorial outputs
- ❑ Synchronizing asynchronous inputs

- ❑ Avoiding race conditions
- ❑ Working with minimum delays
- ❑ Power-on Reset vs. master Reset signal
- ❑ Avoiding stuck states
- ❑ When to use expander latches and flipflops
- ❑ Design reliability checklist

Design Reliability

Thorough worst-case analysis of the EPLD is essential before MPLD conversion begins. Without such analysis, neither the EPLD nor the MPLD can be guaranteed to operate satisfactorily at extreme operating voltages or temperatures, or when variations occur in the device fabrication process. The MPLD design can only be as reliable as the original EPLD design.

Detailed timing analysis must be part of the debugging process. Traditional board- or system-level prototyping can often miss subtle timing problems that careful timing analysis will reveal.

Clocking

Reliable clocking is critical to the successful operation of any digital design, regardless of whether it is constructed with discrete logic, programmable logic, or full-custom silicon. Poorly designed clocking leads to erratic behavior when extreme temperature, voltage, or fabrication process variations occur, and is difficult and expensive to debug.

Several types of clocks are commonly used in designs (called "projects" in MAX+PLUS II). Altera EPLDs have an array (asynchronous) clocking feature that supports any clocking scheme required by an application. Clock configurations can be divided into the following four types: global Clocks, gated Clocks, multi-level logic Clocks, and ripple Clocks. Multi-Clock systems can consist of any combination of these four clocking schemes.

Global Clocks

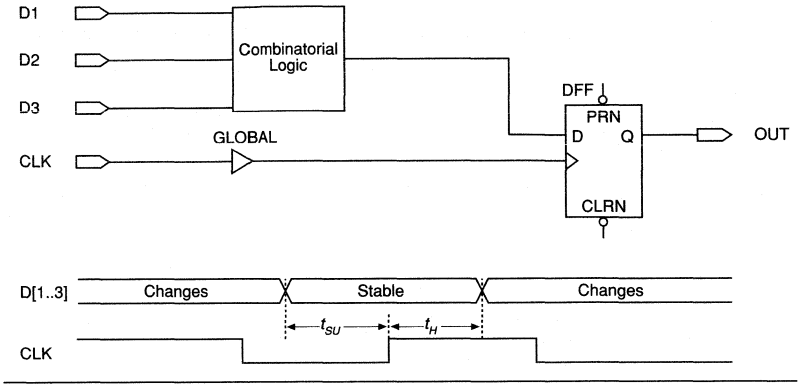
Global (or synchronous) clocks are the simplest and most predictable Clocks for a project. With a truly synchronous system, a single master Clock, driven by an input pin, clocks every flipflop in the project.

You should use global Clocks in your projects whenever possible. Altera EPLDs feature a global Clock pin that can be connected directly to every register in the device. Such global Clocks provide the fastest Clock-to-output delay for the device.

Figure 1 shows an example of a global Clock used in a MAX 5000 EPLD. The `GLOBAL` primitive, provided with MAX+PLUS II software, implements the global Clock for the Clock pin. The timing waveforms in Figure 1 reveal that inputs `D[1..3]`, which form the data input to the flipflop, are subject to a setup and hold time constraint. Setup and hold time requirements are

Figure 1. Global Clock

The preferred method for clocking an EPLD or MPLD is to use the global Clock pin to clock every register in the device. Data pins are then subject only to setup and hold time requirements with respect to the Clock.



given in individual EPLD data sheets or can be calculated with the MAX+PLUS II Timing Analyzer.

If the setup and hold requirements cannot be met in an application, you must synchronize the inputs with the Clock. See "Asynchronous Inputs" later in this application note.

Gated Clocks

In many applications, it is not possible or practical to use a single global Clock throughout the entire project. Altera EPLDs feature product-term logic array Clocks that allow arbitrary functions to clock individual flipflops. However, when you use array Clocks, you should analyze the Clock functions carefully to prevent glitches.

An array Clock is commonly used to construct a gated Clock. Gated Clocks are often associated with a microprocessor interface that uses address lines as a gate for a write strobe. In general, however, a gated Clock exists whenever a flipflop is clocked by a combinatorial function.

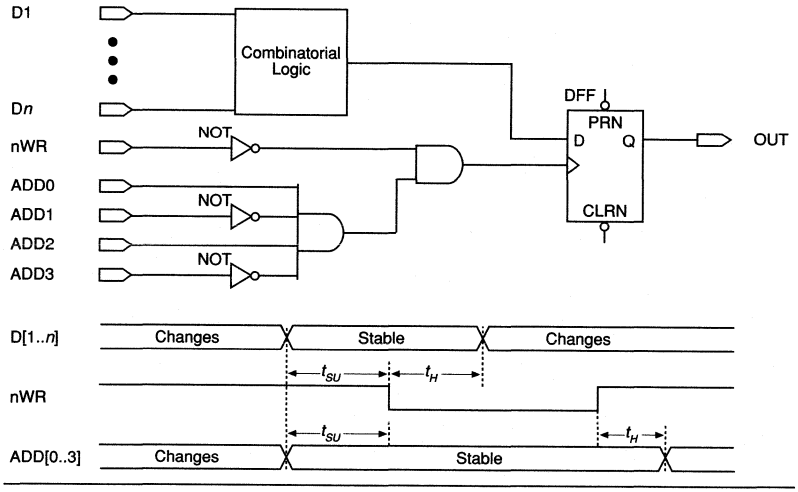
Gated Clocks can be just as reliable as global Clocks if the following conditions are met:

- The logic driving the Clock must consist of a single AND or OR gate. If any additional logic is used, race conditions may produce glitches under certain operating conditions. See "Multi-Level Logic Clocks" later in this application note.

- Only one input to the logic gate should act as the actual Clock; all other inputs to the logic gate must be considered address or control lines that are subject to setup and hold time constraints with respect to the Clock signal.

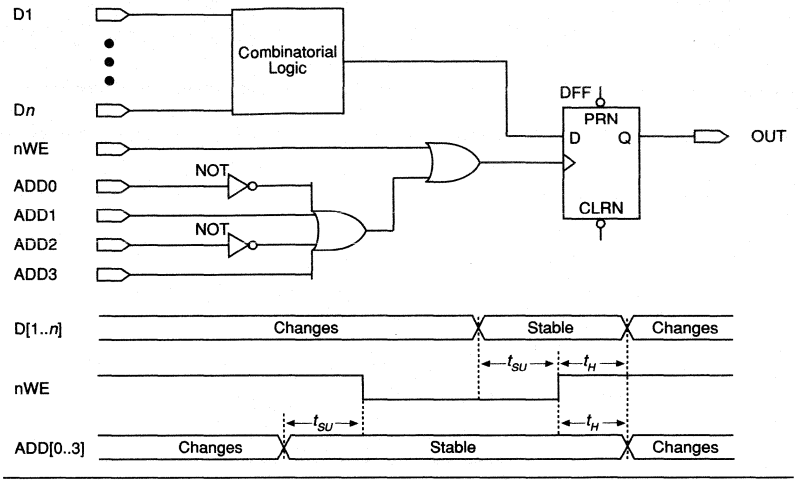
Figures 2 and 3 show examples of reliable gated Clocks. In Figure 2, an AND gate produces the gated Clock; Figure 3 uses an OR gate. In these examples, pins nWR and nWE are considered to be the Clock pins and pins ADD[0..3] are address pins. The data to both flipflops is generated by random logic from signals D[1..*n*].

Figure 2. AND-Gate Clock



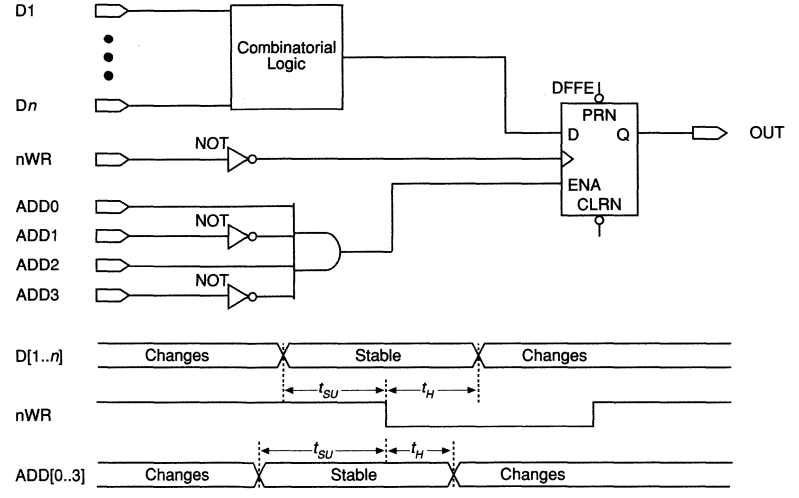
The waveforms in Figures 2 and 3 show the relevant setup and hold time requirements. The address lines for both projects must be stable throughout the time during which the Clock remains active (nWR and nWE are active low). If the address lines fail to remain stable for the time indicated, a glitch may appear at the Clock, causing the flipflop to change state incorrectly. Data pins D[1..*n*], on the other hand, must meet the standard setup and hold specifications surrounding only the active edges of nWR and nWE.

Figure 3. OR-Gate Clock



You can often improve the reliability of a project by converting gated Clocks to global Clocks. Figure 4 shows how you can redesign the circuit in Figure 2 with a global Clock. The address lines now control the Enable input to DFFE, an enable D flipflop provided with MAX+PLUS II software. When ENA is high, the value at the D input clocks into the flipflop; when ENA is low, the current value is maintained.

Figure 4. AND-Gate Clock Converted to Global Clock



The timing waveforms of the redesigned circuit in Figure 4 show that the address lines do not need to be stable throughout the time during which nWR is active. Instead, they only need to meet the same setup and hold time requirements as the data pins.

Figure 5 shows an example of an unreliable gated Clock. The RCO output of a 3-bit synchronous up counter is used to clock a flipflop. However, multiple inputs to the counter act as the Clock, violating one of the requirements for reliable gated Clocks. None of the flipflops that generate the RCO signal can be considered the actual Clock line, since all flipflops change at approximately the same time. Thus, the RCO line may experience a glitch as the counter counts from 3 to 4, as shown in the timing waveforms in Figure 5.

Figure 5. Unreliable Gated Clock

The timing waveforms show how the RCO signal may experience a glitch as the counter goes from 3 to 4.

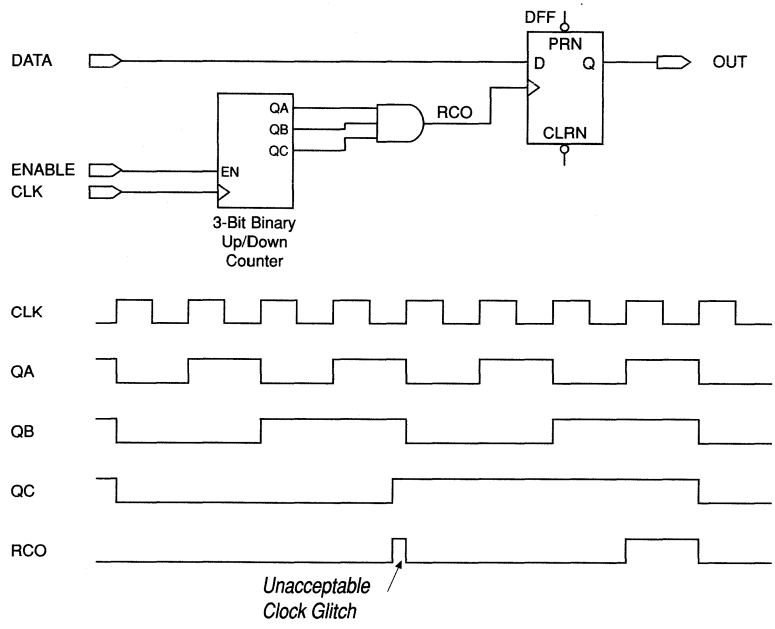
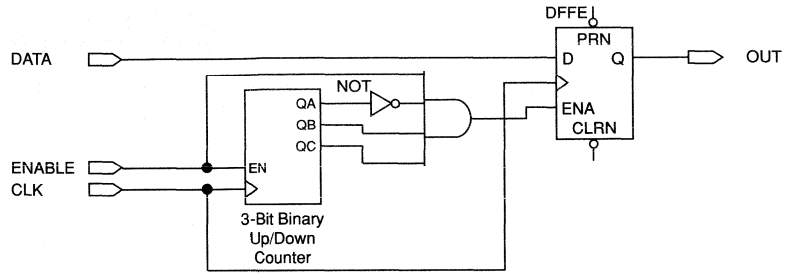


Figure 6 shows a reliable, globally-clocked version of the unreliable counter circuit in Figure 5. RCO controls the Enable input to DFFE. No additional macrocells are required for this change.

Figure 6. Unreliable Gated Clock Converted to Global Clock

This circuit is equivalent to the circuit shown in Figure 5.

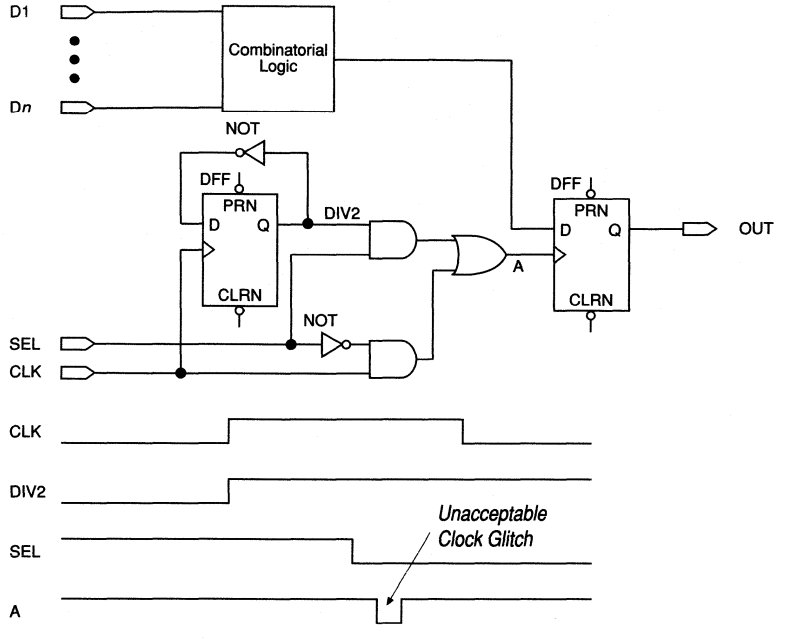


Multi-Level Logic Clocks

When the combinatorial logic generating a gated Clock expands beyond one level (i.e., beyond a single AND or OR gate), it becomes difficult to verify the reliability of the project. Static hazards that are not apparent without extensive prototyping or simulation may exist. In general, you should not use multiple levels of combinatorial logic to clock flipflops in EPLD or MPLD projects.

Figure 7 shows a simple example of a multi-level Clock that contains a hazard. The Clock is the output of a multiplexer controlled by the pin SEL. The inputs to the multiplexer are a Clock (CLK) and a divide-by-two version of the same Clock (DIV2). The timing waveforms in Figure 7 show that a static hazard exists when the SEL line changes while both Clocks are a logical 1. The degree of the hazard depends on the operating conditions.

Figure 7. Multi-Level Clock with Static Hazard



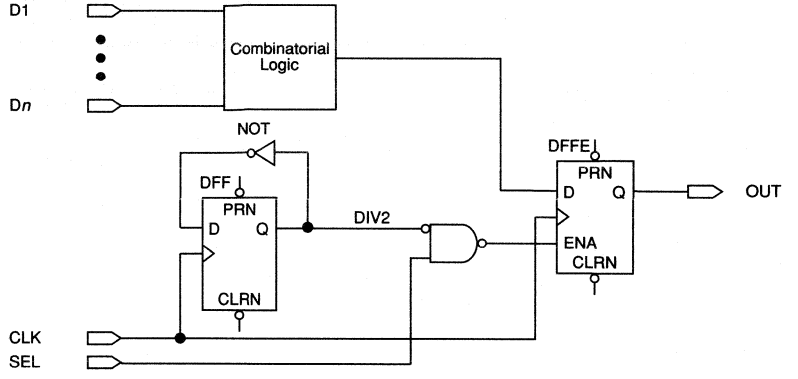
You can remove the hazards of multi-level logic. For example, you can try inserting redundant logic into the project. However, the MAX+PLUS II Compiler may remove this logic during logic synthesis, making it difficult to verify whether a hazard has actually been removed. Therefore, you should find a different way to implement the function instead.

Figure 8 shows alternative single-level clocking for the circuit in Figure 7. Here the `SEL` pin and the `DIV2` signal are used to control the Enable input to an enable D flipflop, rather than the Clock pin of the flipflop. No additional macrocells are required for this circuit.

Different systems require different approaches to removing multi-level Clocks. Feel free to contact Altera Applications at (800) 800-EPLD for assistance.

Figure 8. Single-Level Clock with No Static Hazard

This circuit is logically equivalent to the circuit shown in Figure 7, but is much more reliable.

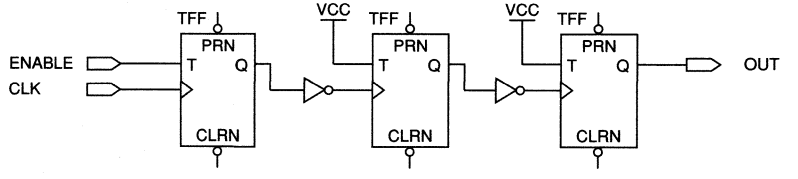


Ripple Clocks

Another popular clocking scheme is to use ripple Clocks, i.e., to use the output of one flipflop as the Clock input to a second. If used carefully, ripple Clocks can be just as reliable as global Clocks. However, ripple Clocks complicate the timing calculations associated with a circuit. A ripple Clock introduces a large skew between the clocking of the flipflops in a ripple chain, and can stretch out the worst-case setup times, hold times, and Clock-to-output delays for a circuit, slowing down the actual system speed.

Ripple Clocks often occur when a ripple counter is constructed from toggle flipflops, with the output of one flipflop clocking the input of the next. See Figure 9.

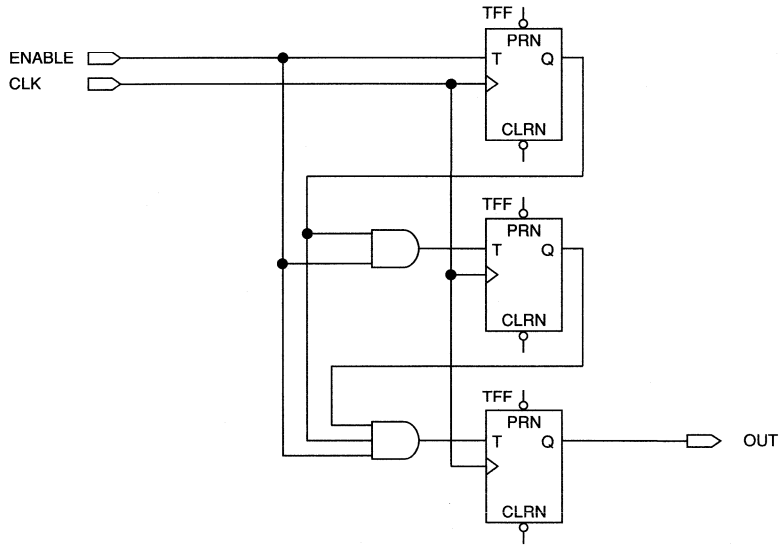
Figure 9. Ripple Clock



Synchronous counters are often preferable to ripple counters because they have a faster Clock-to-output time and require the same number of macrocells. Figure 10 shows the circuit in Figure 9 converted to a synchronous counter with global clocking. This implementation requires the same number of macrocells and has a faster Clock-to-output time. The

Figure 10. Ripple Clock Converted to Global Clock

This synchronous three-bit counter is an alternative to the ripple counter shown in Figure 9. It uses the same three macrocells and has a shorter Clock-to-output delay.



MAX+PLUS II TTL MacroFunction Library offers a wide variety of synchronous counter macrofunctions.

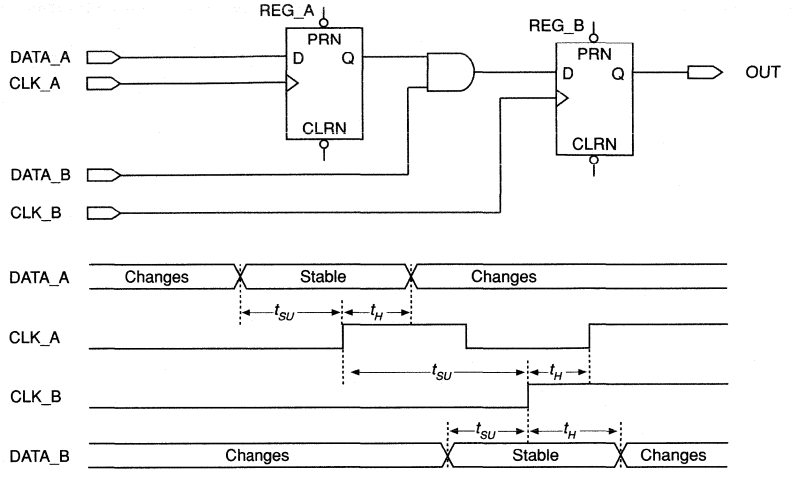
Multi-Clock Systems

Many applications require the use of multiple Clocks within a single EPLD. Common examples are designs that interface between two asynchronous microprocessors, or between a microprocessor and an asynchronous communications channel. These applications introduce additional timing constraints due to the setup and hold time requirements between the Clock signals. They may also require synchronization of some asynchronous signals.

Figure 11 shows an example of a multi-Clock system. `CLK_A` is used to clock `REG_A`; `CLK_B` is used to clock `REG_B`. Because `REG_A` drives the combinatorial logic that goes to `REG_B`, `CLK_A` has setup and hold time requirements with respect to `CLK_B`. Since `REG_B` does not drive the logic feeding `REG_A`, there is no setup time requirement on the leading edge of `CLK_B` with respect to `CLK_A`. Also, there is no requirement on the relative falling edges of `CLK_A` and `CLK_B` because the falling edges do not affect the flipflops.

Figure 11. Multi-Clock System

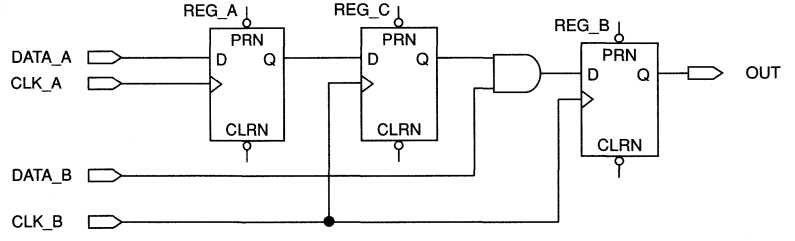
The timing waveforms show that the leading edge of CLK_A has a setup and hold constraint with respect to the leading edge of CLK_B.



In circuits that have two independent Clocks, such as in Figure 11, it is impossible to ensure that the setup and hold times between them are met. In such cases, you must synchronize the circuit. Figure 12 shows how the REG_A values are synchronized with CLK_B before they are used. A new flipflop, REG_C, is clocked by CLK_B, which ensures that its outputs will meet the setup time of REG_B. However, this approach delays the output by one Clock cycle.

Figure 12. Multi-Clock System with Synchronized Register Output

If CLK_A and CLK_B are independent, then the output of REG_A must be synchronized by REG_C before it feeds REG_B.

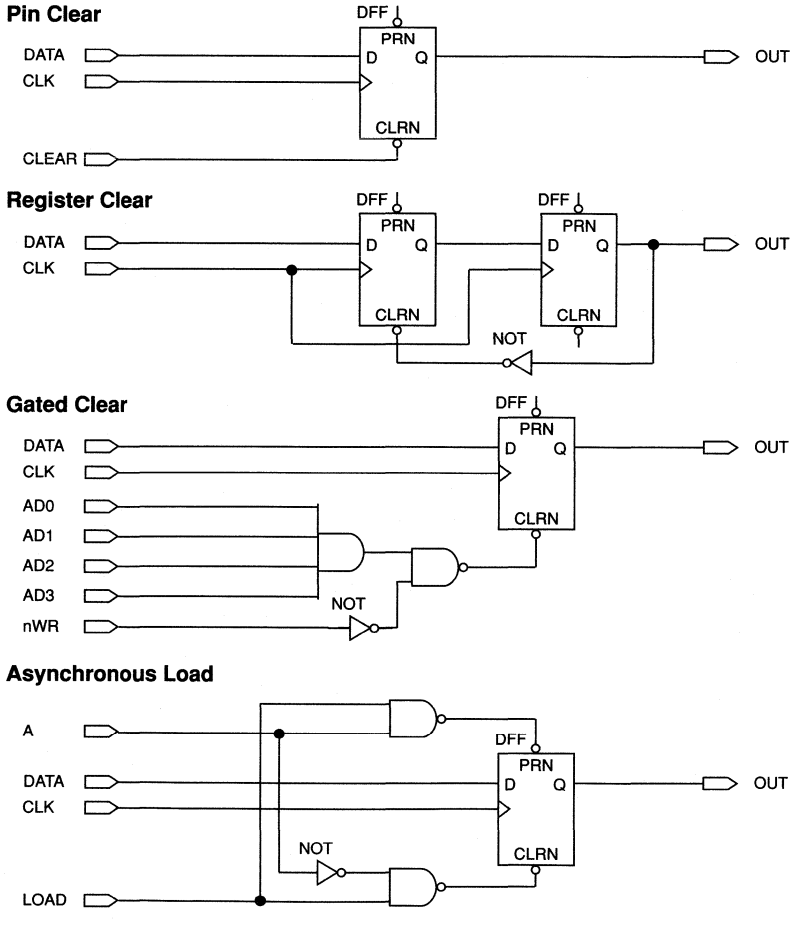


Clear & Preset Signals

Clear and Preset signals demand the same careful consideration as Clocks, because they are also sensitive to race conditions and hazards. As with Clocks, the best Clear or Preset configuration is to drive the signal directly from a device pin. It is good practice to have a single master Reset pin that feeds the Clear or Preset of every flipflop in the project. If you must generate Clear and Preset signals from within the device, create them according to the guidelines in "Gated Clocks."

If you use a gated Clear or Preset, a single pin or flipflop should act as the source of the Clear or Preset, while all other signals act as address or control lines. The address or control lines must remain stable while the Clear or Preset is active. Figure 13 shows four examples of acceptable Clear and Preset configurations. You should never generate Clear and Preset

Figure 13. Acceptable Preset and Clear Configurations



signals with multi-level logic or with single-level logic that contains a race condition. See "Race Conditions" later in this application note for more details.

Combinatorial Outputs

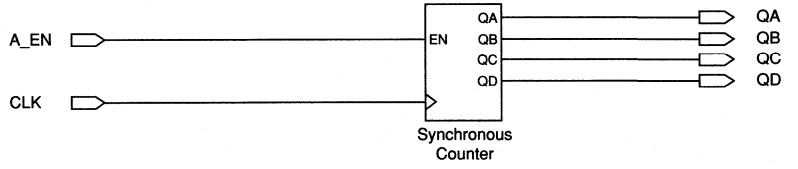
EPLD output pins that drive an edge- or level-sensitive signal elsewhere in the system must be treated just as carefully as internal Clock, Clear, and Preset signals. Whenever possible, you should register hazard-sensitive combinatorial outputs at the output of the EPLD. If you cannot register a hazard-sensitive output, then it should meet the gated clocking conditions discussed in "Gated Clocks." You should never use multi-level logic to drive a hazard-sensitive output.

Asynchronous Inputs

By definition, asynchronous inputs cannot always meet the setup and hold time requirements of the flipflops they feed. Therefore, asynchronous inputs can often cause an incorrect value to be clocked into a flipflop, or cause a flipflop to enter a metastable state in which its output is not recognized as a 1 or a 0. If not properly handled, metastability can lead to serious system reliability problems.

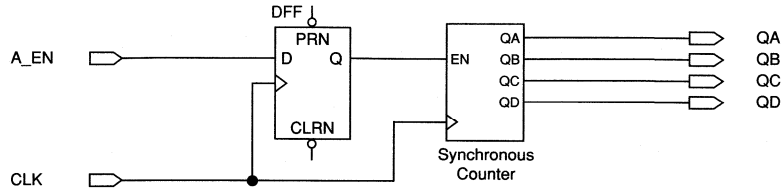
Figure 14 shows a binary counter with an asynchronous signal controlling the Enable input. When the Enable input violates the setup or hold time constraint of the counter, each bit of the counter may behave erratically. One bit may count up while another bit holds, causing the counter to enter an invalid state. In addition, any one of the bits may become metastable, creating problems in other portions of the circuit.

Figure 14. Binary Counter with Asynchronous Enable



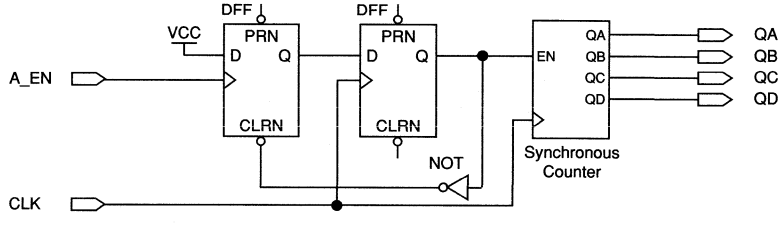
Synchronizing the Enable signal with an extra flipflop removes the reliability problem by guaranteeing that the counter's setup time will not be violated. Figure 15 shows one method of synchronization. Although the synchronizing flipflop may still experience metastability, it should stabilize before the next Clock edge. In general, to avoid metastability problems in an EPLD or MPLD, you should never have an asynchronous signal fan out to more than one flipflop within a device.

Figure 15. Binary Counter with One Synchronizing Flipflop



An alternative method of synchronizing an asynchronous input is shown in Figure 16. The input drives a Clock to a flipflop with the data input tied to V_{CC}. This circuit is useful for detecting asynchronous events that may be shorter than the duration of the Clock period.

Figure 16. Binary Counter with Two Synchronizing Flipflops



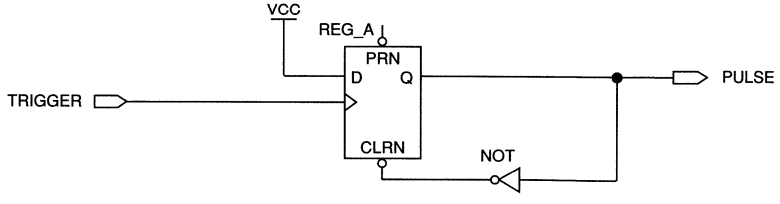
Race Conditions

A race condition exists in a circuit when a signal follows two or more paths to a common circuit element. There are countless examples of race conditions; however, you should never design circuits whose reliable operation depends on a predictable skew between two different signal paths. Internal delays are never predictable and vary with the fabrication process, as well as with the operating temperature and supply voltage.

Figure 17 shows a common example of a race condition in an asynchronous pulse generator. The circuit is not reliable because there is no guarantee

Figure 17. Asynchronous Pulse Generator with a Race Condition

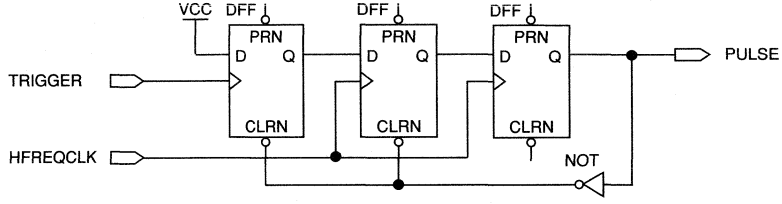
This pulse generator creates race conditions because there is no guarantee of how long the pulse signal will be active.



that the pulse from REG_A will be long enough to meet the minimum pulse-width requirements elsewhere in the circuit. Instead, you should use a synchronous pulse generator, as shown in Figure 18. In this figure, two flipflops are used to create a pulse from a high-frequency Clock.

Figure 18. Synchronous Pulse Generator

This pulse generator is a reliable alternative to the asynchronous pulse generator shown in Figure 17, but requires two extra macrocells and a high-frequency Clock pin.



If you follow these guidelines for constructing Clocks, Clears, and Presets, you should be able to create circuits that do not contain any catastrophic race conditions.

Minimum Delays

You must never use macrocells or expander product terms within an EPLD to create an intentional delay or asynchronous pulse. The delay of these elements varies with temperature, power supply voltage, and device fabrication process, so race conditions can occur and create an unreliable circuit.

Figures 19 and 20 show examples of how the MAX+PLUS II EXP and MCELL primitives can be misused to create an intentional delay. The circuit in Figure 19 attempts to vary the setup, hold, and Clock-to-output specifications of the flipflops. However, the best-case delay of a macrocell or expander cannot be guaranteed. These delays only increase the circuit's sensitivity to operating conditions and decrease its reliability.

Figure 19. Misused MCELL and EXP Primitives

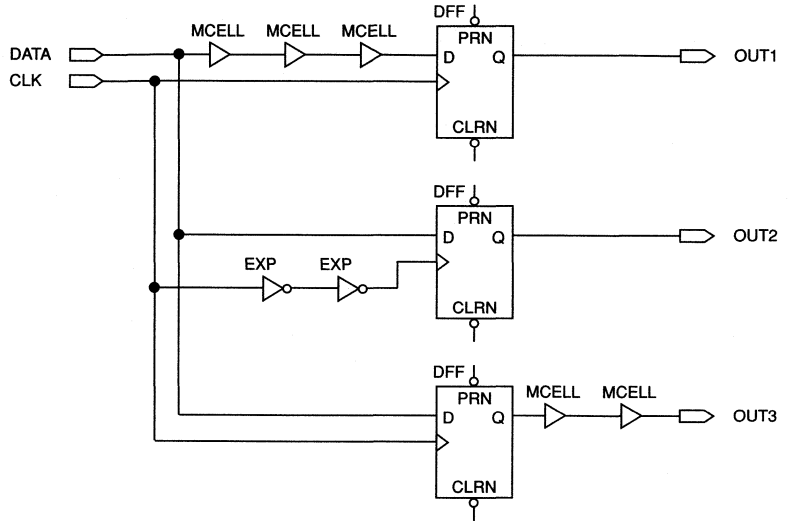
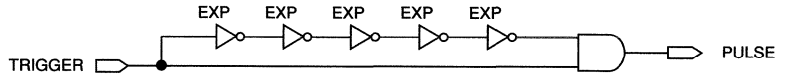


Figure 20 shows an attempt to create an asynchronous pulse generator with EXP primitives. A reliable alternative is the synchronous pulse generator shown in Figure 18.

Figure 20. Unreliable Timing-Dependent Asynchronous Pulse Generator



Power-On Reset vs. Master Reset Signal

Although Altera EPLDs have built-in Power-On Reset (POR) circuitry to guarantee that the flipflops are cleared before use, the MPLDs do not. Do not rely on this feature in your EPLD project if you intend to convert it into an MPLD.

Your design should use a board-level Reset signal to clear every flipflop to ensure proper operation in both the EPLD and the MPLD. If a master Reset is not available, you must ensure that all flipflops are initialized by the system before they are used. Thus, all data flipflops should be written to, all counters should be cleared (synchronously or asynchronously), and all state machines should be forced to a known state.

Stuck States

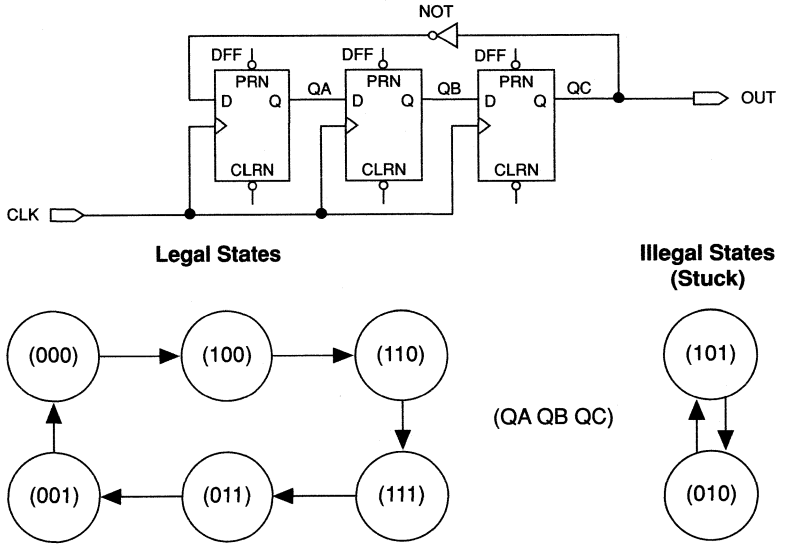
Many sequential circuits have states (i.e., flipflop value combinations) that are considered illegal or "don't cares" by the system. If a flipflop somehow enters one of these illegal states, it becomes stuck in that state. Therefore, a

project should force any flipflops that enter an illegal state to go to a legal state on the next Clock cycle.

In state machines, it is common practice to force all unused states to go to the idle state unconditionally. The problem of stuck states, however, is not limited to state machines. Figure 21 shows a schematic that contains a stuck state. If the flipflops in this circuit somehow achieved the values `101` or `010`, they would never leave the illegal loop.

Figure 21. Stuck Illegal States

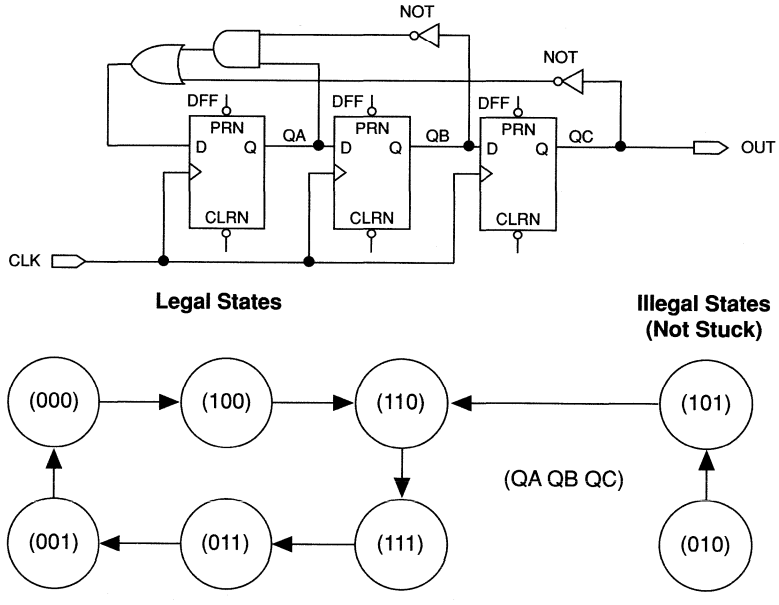
This circuit has two illegal states that are stuck.



The simple changes shown in Figure 22 cause the illegal states to loop back to a legal state in at most two Clock cycles.

Figure 22. Unstuck Illegal States

This circuit has the same illegal states as the circuit shown in Figure 21, but they are no longer stuck.



Expander Latches & D Flipflops

MAX 5000 EPLDs contain shareable expander product terms that you can use to create additional latches or flipflops in a device. Although expander latches greatly increase the flexibility of a MAX 5000 EPLD, you should use them only after all dedicated latches and flipflops in the device have been used.

When implementing expander latches or expander flipflops in a project targeted for an MPLD, you should use only the EXPDFF, EXPLATCH, NORLTCH, and NANDLTCH symbols provided with the MAX+PLUS II TTL MacroFunction Library; you should not create your own cross-coupled structures.

Conclusion

To ensure 100% success with an EPLD-to-MPLD conversion, you must understand the potential logic and timing irregularities discussed in this application note and apply the recommended design rules to your project. Board-level prototyping, although invaluable for general debugging and project validation, cannot be expected to replace careful worst-case timing analysis.

The following checklist covers basic design reliability issues that should be addressed before you submit an EPLD design for MPLD conversion. If you have any questions about how these guidelines apply to a particular project, or would like a copy of the ***MPLD Conversion Information & Order Forms*** workbook, contact Altera Applications at (800) 800-EPLD.

Design Reliability Checklist

- ❑ Ensure that Clocks are glitch-free.
- ❑ Never use Clocks consisting of more than one level of combinatorial logic.
- ❑ Carefully calculate setup and hold times for multi-Clock systems.
- ❑ Synchronize signals between two-Clock systems when the setup and hold conditions cannot be met.
- ❑ Ensure that Clear and Preset signals do not contain race conditions.
- ❑ Register all glitch-sensitive outputs.
- ❑ Synchronize all asynchronous inputs.
- ❑ Ensure that no internal race conditions exist.
- ❑ Never rely on minimum delays for pin-to-pin or internal delays.
- ❑ Use the MAX+PLUS II Timing Analyzer and Simulator to calculate setup and hold times.
- ❑ Do not rely on Power-On Reset. Use a master Reset pin to clear all flipflops.
- ❑ Remove any stuck states from state machines or synchronous logic.
- ❑ Do not create your own cross-coupled structures. Instead, use the EXPDFF, EXPLATCH, NORLTCH, and NANDLTCH symbols provided in the MAX+PLUS II TTL MacroFunction Library.

Implementing EISA Interfaces with MAX EPLDs

April 1992, ver. 1

Application Note 27

Introduction

In the past, the lack of interface ICs for Extended Industry Standard Architecture (EISA) buses made it difficult to develop EISA interface products. Now, Altera's MAX 5000 and MAX 7000 EPLDs provide sufficient logic density to meet the requirements for complex, fast EISA bus interfaces. This application note describes a design that implements a bus master controller for an EISA interface card, and discusses how to create the design with MAX+PLUS II software and Altera EPLDs.

EISA Bus Evolution

In 1981, IBM developed an 8-bit bus architecture for the Personal Computer (PC), called the Industry Standard Architecture (ISA) bus. In 1984, when IBM developed the PC-AT with the 80286 CPU, it extended the ISA bus to 16 bits to take advantage of the new CPU capabilities. The new PC-AT ISA bus was compatible with the 8-bit ISA cards from the original PC bus architecture, ensuring that the older 8-bit cards would not become obsolete. However, with the 80386 CPU, which provided a full 32-bit datapath, IBM abandoned the ISA bus in favor of the new Micro Channel Architecture (MCA) bus. The MCA bus uses a different form factor and electrical specification than the ISA bus, making the 8- and 16-bit ISA interface cards incompatible with IBM 80386 computers.

Other vendors decided that, since many ISA-based interface cards were already installed, and since they already had a significant investment in the ISA technology, they would develop a new backwards-compatible bus standard. Therefore, a group of nine PC manufacturers developed the EISA bus. The EISA bus architecture is compatible with existing 8- and 16-bit PC add-on cards and supports EISA add-on cards. EISA add-on cards contain many enhancements, including a standard 32-bit datapath, a self-configuration system, and synchronous Direct Memory Access (DMA) burst transfer capability.

ISA Limitations

The ISA interface uses a peripheral input/output (PIO) adapter, a special-purpose integrated circuit that controls the bus interface between the CPU and add-on cards. The PIO implements an asynchronous data transfer protocol that requires one CPU interrupt per block of data transferred. All non-DMA transfers must pass through the PIO, limiting the system to an effective bandwidth of 2 Mbytes/s with an 8.33-MHz bus Clock. This approach is called master-slave protocol: the master is the system CPU, and the slave is the target add-on card.

While the PIO master-slave approach works well in the 8- and 16-bit ISA cards, the 32-bit processor and the need for a true 32-bit bus standard demanded a solution that would bypass PIO limitations. The newer EISA architecture removes the asynchronous control restrictions and supplies extensions to the data bandwidth and transfer operations, while still supporting the existing ISA interface products.

EISA Advantages

The EISA architecture supports PIO and asynchronous DMA controller bus transfers, and is compatible with both ISA and EISA add-on cards. It allows the system to translate bus cycles of various cards that support a wide range of bus widths and formats. All existing operating systems and most software packages run without modification on EISA platforms when used with the EISA system configuration utility and special device drivers.

During power-up, the EISA system configuration utility reads a unique signature ID from each EISA add-on card, identifies each card by I/O type and slot number, and stores the information in local RAM. This information is then used to configure each add-on card so that the device interfaces take advantage of the hardware capabilities of the EISA bus. Each hardware element uses either ROM- or RAM-based drivers that are loaded into system memory during power-up to interface to the computer system. With a software-based interface, you can easily make minor changes to existing products without major hardware changes.

Bus Masters

The intelligent EISA add-on card can request control of the bus, while the ISA add-on card is forced to use the PIO slave mode for all transfer cycles. The system motherboard arbitrates between multiple EISA bus master add-on cards and grants access to the bus based on the encoded priority in each bus master. This process allows several intelligent bus masters to share bandwidth resources. Once the motherboard has granted control of the bus, the bus master generates all address, control, and data signals until the transfer is complete or the bus master is preempted by a higher-level bus master or host-DMA refresh cycle.

The ability to control the data bus without CPU processor intervention allows the EISA bus to transfer large blocks of data at 33 Mbytes/s while the system motherboard simultaneously executes tasks that do not require bus-access cycles. In some complex motherboards, data and instruction caches are built in, allowing the CPU to execute very complex tasks without memory- or storage-access cycles. A bus master can take advantage of bus latency, thereby enhancing system performance.

You have three design options to take advantage of these enhancements in bus technology:

- You can rely on off-the-shelf interface components that implement most of the bus protocols in silicon. However, this approach does not allow you to modify features of the interface and restricts your design flexibility.
- You can implement the interface in discrete TTL logic or PALs. However, this choice requires more board space for the interface portion of the design and increases power consumption.
- You can use Altera's MAX 5000 and MAX 7000 EPLDs.

The high-performance, high-density EPM7256 EPLD has the speed and capacity to implement an EISA interface in a single device, simultaneously providing design flexibility and reducing board-space requirements. Fabricated on a 0.8-micron EPROM technology, the EPM7256 EPLD provides in-system speeds of 62.5 MHz and propagation delays of 20 ns. With 256 macrocells, the EPM7256 can implement complete system-level designs. An EISA interface design can also be implemented in multiple MAX 5000 EPLDs.

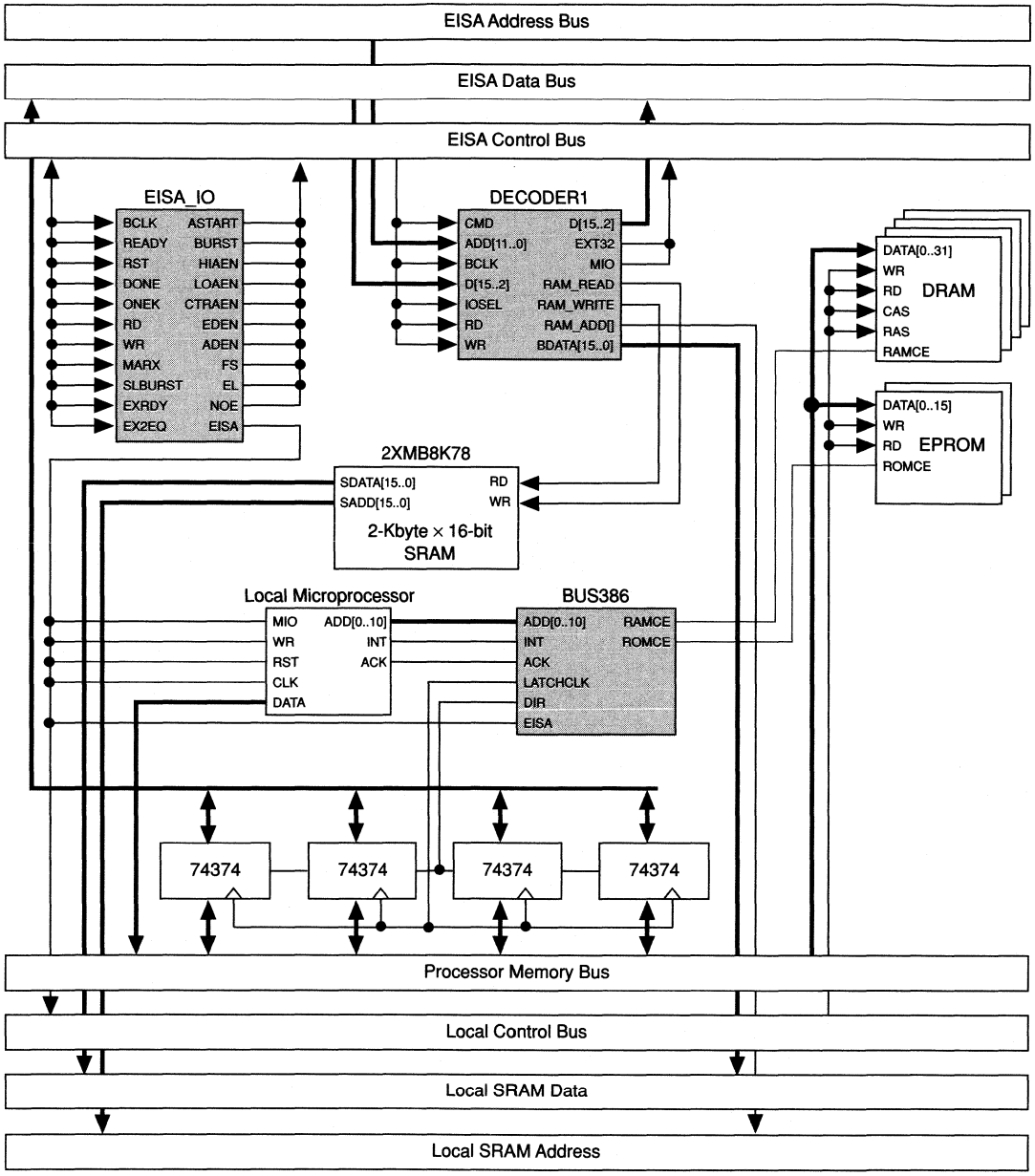
Logic density and board-space savings are only two of the advantages of the EPM7256 EPLD. You can also make quick design changes by erasing and reprogramming the device, and enter and simulate the entire design with the highly integrated, user-friendly MAX+PLUS II software. The EPM7256 EPLD provides a programmable speed/power option that allows speed-critical portions of the design to run at high speed while the remainder runs at reduced speed and low power. This feature enables you to save 50% or more power, reducing add-on card power requirements.

EISA Design Description

Figure 1 shows a block diagram of the EISA bus interface design. The EISA bus master interface contains three subdesigns: an EISA bus address decoder, interface logic for a local board processor, and an EISA bus arbitration unit. All three functions are easily implemented in a single EPM7256 EPLD, with plenty of device resources left over for additional logic. As an alternative, you can partition this design into two MAX 5000 EPLDs: an EPM5064 and an EPM5130.

You can implement the bus master interface with Altera's PC-based MAX+PLUS II development software, or with third-party CAE tools (e.g., Cadence, Viewlogic, Mentor Graphics, Synopsys) and the MAX+PLUS II workstation-based software. MAX+PLUS II supports hierarchical designs, which allow you to divide design logic into functional blocks, so that you can compile and simulate them before they are integrated into the overall design.

Figure 1. EISA Bus Interface Design Block Diagram

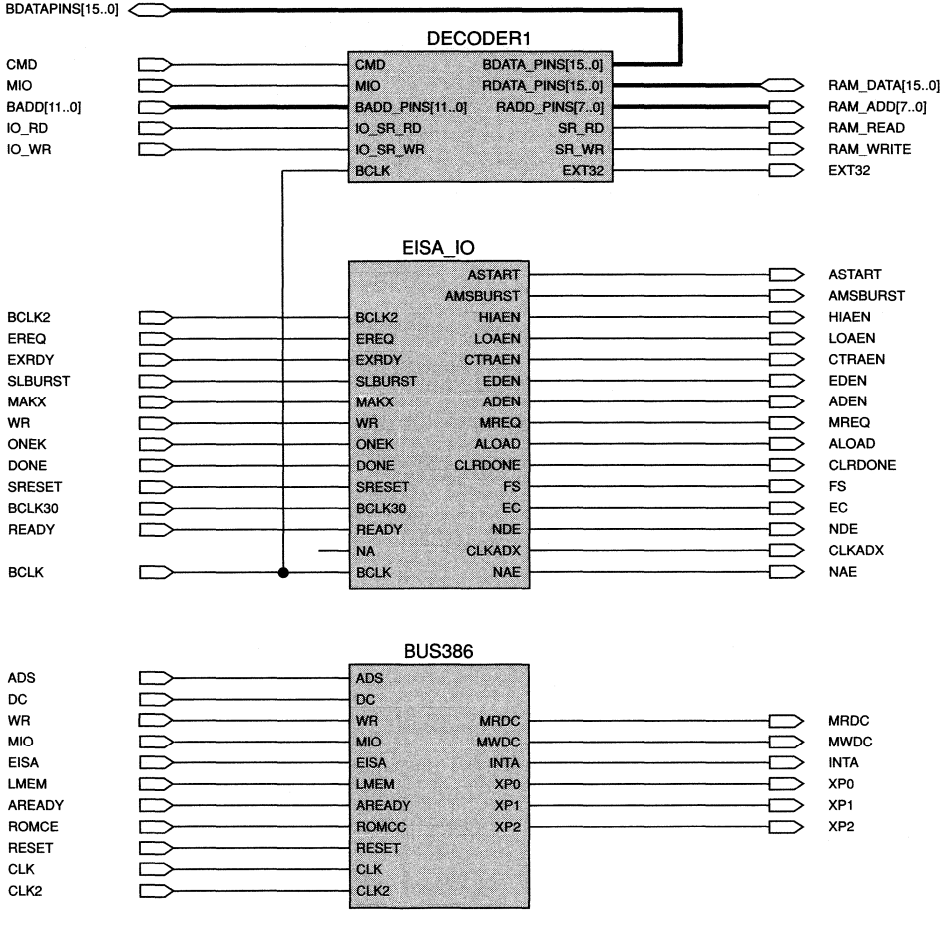


MAX+PLUS II supports three design entry methods:

- Graphic designs describe logic primitives and macrofunctions with schematics.
- Text designs use the Altera Hardware Description Language (AHDL) to describe the behavior of logic circuits.
- Waveform designs allow you to describe a design in terms of expected input and desired output waveforms.

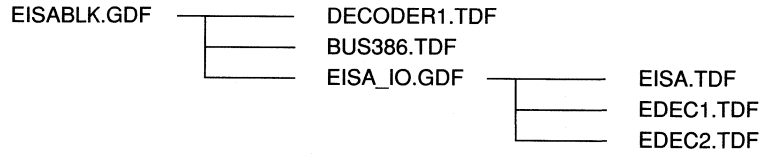
The design (called a "project" in MAX+PLUS II) presented in this application note uses the graphic and text design entry methods. Figure 2 shows the top-level Graphic Design File (.GDF), EISABLK.GDF, for the EISA bus interface project. EISABLK.GDF includes the address decoder DECODER1

Figure 2. Bus Interface Design (EISABLK.GDF)



and interface logic BUS386, which are entered as AHDL Text Design Files (.TDF). The logic for the bus arbitration state machine EISA_IO contains three TDFs. Figure 3 shows the hierarchy for the EISA bus interface project.

Figure 3. Hierarchy for EISABLK Project



EISA Bus I/O Decoder (DECODER1.TDF)

To take advantage of the EISA bus, EISA add-on cards must implement a variety of features not found on standard ISA cards, including a multi-register protocol for command queuing, control and status registers, and system-accessible power-on system test (POST) registers. The POST registers provide a configuration ID signature, power-on self-test status, and configuration registers. The EISA I/O bus uses 16 bits to uniquely identify I/O cards and to address the registers that implement the POST registers and status bits. Four of the bits identify each card, and the other 12 bits identify POST, status, and control registers on the add-on card. The EISA bus can support up to 16 add-on cards. These register addresses are separate from the memory-mapped addresses of the devices and the memory that implements the actual functionality of an add-on card.

The mailbox registers, accessible from both the EISA interface and the local processor, are a set of indexed registers on the add-on card that enable the host to send commands to the add-on card without waiting for the current command to be completed. Mailbox registers are extremely useful for multi-tasking operating systems in which multiple programs are sharing the resources of a single add-on card. If the EISA bus and the local processor try to access a mailbox register simultaneously, the index gives priority to the local processor on the add-on card. The motherboard takes advantage of the index queuing capability with a Common Access Method (CAM) device driver, which allows up to 16 commands to be sent to the add-on card without waiting for a single command to be completed. The local processor reads and processes each command, then sends the appropriate data and status back to the motherboard.

The EISA bus I/O decoder determines whether the command, status, POST, or mailbox registers are being addressed, and translates the bus address into the appropriate address and read/write control signals for a local SRAM. The SRAM provides a simple register bank to store the configuration information. The EPM7256 EPLD determines whether the

operation is an I/O or memory operation, then maps the decoded EISA mailboxes to SRAM addresses.

Figure 4 shows a portion of the EISA bus I/O decoder file DECODER1.TDF. The complete file is available from the Altera electronic bulletin board service (BBS). For information on AHDL structure and syntax, refer to MAX+PLUS II Help.

This file uses a number of Constant Statements (keyword CONSTANT) to map local SRAM addresses. You can modify them as your design requirements change. The following example shows a single Constant Statement:

```
CONSTANT    G_CONF_REG = H"100";
```

The Constant Statements for this file are contained in an Include File (DECODER1.INC) and integrated into DECODER1.TDF with an Include Statement (keyword INCLUDE). When you compile the file, the MAX+PLUS II Compiler replaces the Include Statement with the contents of DECODER1.INC. By storing the SRAM addresses in an Include File, you can easily modify the design without risking changes to the rest of the circuit.

Figure 4. EISA Address Decoder Excerpt (DECODER1.TDF) (Part 1 of 3)

```
INCLUDE "DECODER1.INC";                  % EISA addressing constants %

DESIGN IS decoder1 DEVICE IS "auto";     % Use auto for subdesigns   %

SUBDESIGN decoder1                       % Define name of subdesign  %
(
    cmd, mio,                  : INPUT;
    badd_pins[11..0],          : INPUT;
    io_sr_rd, io_sr_wr, bclk   : INPUT;   % Define dedicated input pins       %
    bdata_pins[15..0],         : BIDIR;
    rdata_pins[15..0]          : BIDIR;   % Define bidirectional 16-bit buses %
    radd_pins[7..0],           : OUTPUT;
    sr_rd, sr_wr, ext32        : OUTPUT;  % Define dedicated output pins      %
)
VARIABLE

% Define 6-state state machine to allow Compiler to choose width necessary %
    bus_state                  : MACHINE WITH STATES ( idle,
                                                       get_sr_rd_address,
                                                       get_sr_wr_address,
                                                       sr_write_data,
                                                       send_data,
                                                       read_data);
    bdata[15..0]          : DFF;    % Data read from bus register  %
    rdata[15..0]          : DFF;    % Data read from SRAM register %
    bdata_tri[15..0]      : TRI;    % EISA bus tri-state buffer    %
    rdata_tri[15..0]      : TRI;    % Local SRAM tri-state buffer  %
```

Figure 4. EISA Address Decoder Excerpt (DECODER1.TDF) (Part 2 of 3)

```
    badd[11..0]          : DFF;    % EISA bus address register    %
    radd[7..0]           : DFF;    % Local SRAM address register  %

BEGIN

% Change bus state only when bus Clock and memory input/output control are true %
    bus_state.clk =  bclk & !mio;

% Latch bus address only when bus clock and command control signal are true     %
    badd[].clk    =  bclk & cmd;

    ext32         =  GND;             % Always asserted, indicates 32-bit transfer    %

    badd[]        =  badd_pins[];     % Connect bus input pins to addressing input    %
    radd_pins[]   =  radd[];          % Connect register outputs to RAM address pins  %
    bdata[]       =  bdata_pins[];    % Connect bus data pins to data register input  %
    rdata[]       =  rdata_pins[];    % Connect SRAM data pins to SRAM data register  %

    rdata_tri[]   =  rdata[];         % Connect local register outputs to tri-state buffers %

    bdata_tri[]   =  bdata[];         % Connect bus register outputs to tri-state buffers   %
    bdata_pins[]  =  rdata_tri[];     % Connect tri-state outputs to EISA bus pins          %
    rdata_pins[]  =  bdata_tri[];     % Connect tri-state outputs to local bus pins         %

    IF badd[] == ID0 THEN             % ID0 is decoded           %
       radd[] =  ID0_LOC;             % Set local SRAM addresses %
    END IF;

    IF badd[] == G_CONF_REG THEN      % Decode global configuration register %
       radd[] =  G_CONF_LOC;          % Set local SRAM address               %
    END IF;

    IF badd[] == S_INT_EN THEN        % System interrupt enable register %
       radd[] =  S_INT_LOC;           % Set local SRAM address           %
    END IF;

    IF badd[] == S_PORT0 THEN         % Semaphore port PORT O  %
       add[]  =  S_PORT0_LOC;         % Set local SRAM address %
    END IF;


    IF badd[] == L_DOORBELL_EN THEN   % Local doorbell enable  %
       radd[] =  L_DRBLL_EN_LOC;      % Set local SRAM address %
    END IF;

        .
        .
        .

    % BUSADD is a five-state state machine used to poll host requests %

    CASE bus_state  IS                    % State machine of bus address            %
    WHEN idle =>                          % Wait for read or write from motherboard %
        IF io_sr_wr THEN                  % If write request from motherboard %
           bus_state = get_sr_wr_address; % Set state to write state          %
        END IF;
```

Figure 4. EISA Address Decoder Excerpt (DECODER1.TDF) (Part 3 of 3)

```
        IF io_sr_rd THEN                    % If read request from motherboard %
           bus_state = get_sr_rd_address;   % Translate to local SRAM address  %
        END IF;

    WHEN read_data =>
        sr_rd             = VCC;            % Issue a read strobe to SRAM       %
        bdata_tri[].oe    = GND;            % Disable output of bus data buffer %
        bus_state         = send_data;      % Move to send data state           %

    WHEN sr_write_data =>
        radd[].clk        = VCC;
        bdata_tri[].oe    = VCC;
        sr_wr             = VCC;            % Issue write strobe signal      %
        bus_state         = idle;           % Move to idle state             %

    WHEN  send_data =>                      % If data ready transfer to host %
        bdata_tri[].oe    = GND;
        rdata[].clk       = VCC;            % Clock data into RAM latch  %
        rdata_tri[].oe    = VCC;            % Enable data onto host bus  %
        bus_state         = idle;

    % Read to EISA bus from local SRAM %

    WHEN  get_sr_rd_address =>              % When data is read from SRAM %
        radd[].clk        = VCC;
        bus_state         = read_data;      % Advance to read data state  %

    % Write from EISA bus to local SRAM %

    WHEN get_sr_wr_address =>

        rdata_tri[].oe    = GND;            % Disable tri-state host driver   %
        bdata[].clk       =  VCC;           % Clock bus data into BDATA latch %
        bdata_tri[].oe    = VCC;            % Enable data onto SRAM data bus  %
        bus_state         = sr_write_data;  % Move to SRAM write data state   %
        radd[].clk        = VCC;
   END CASE;
END;
```

Figure 5 shows a state machine diagram for DECODER1.TDF.

Local Microprocessor Interface (BUS386.TDF)

Many high-performance systems use a local microprocessor for time-consuming tasks such as command queuing and complex data buffering functions. In BUS386.TDF, the local microprocessor interface, the control logic allows the microprocessor to maintain the data cache, read queued commands from the host, and transfer status and control information to and from the host. It also provides read/write control glue logic, local processor memory mapping, wait-state generation for SRAM and EPROM timing, and interrupt acknowledgment time-out. The design is implemented as a state machine; inputs are decoded synchronously to determine a change of state.

Figure 5. EISA Address Decoder State Machine Diagram

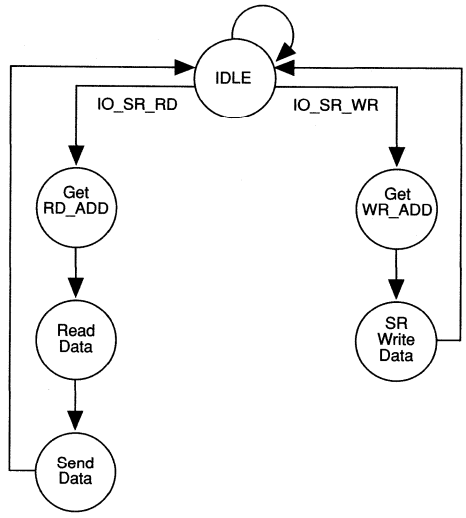


Figure 6 shows a portion of BUS386.TDF, which implements the BUSTATE state machine. This state machine controls the interface logic and puts control requests into sequence. The complete file is available on the Altera BBS. BUSTATE is clocked by CLK2 at 16.66 MHz (twice the frequency of the system bus Clock BCLK), and is qualified by BCLK and the EISA bus control signal MIO. The RESET input signal initializes BUSTATE and clears the system at power-up.

Figure 6. Local Microprocessor Interface (BUS386.TDF) (Part 1 of 2)

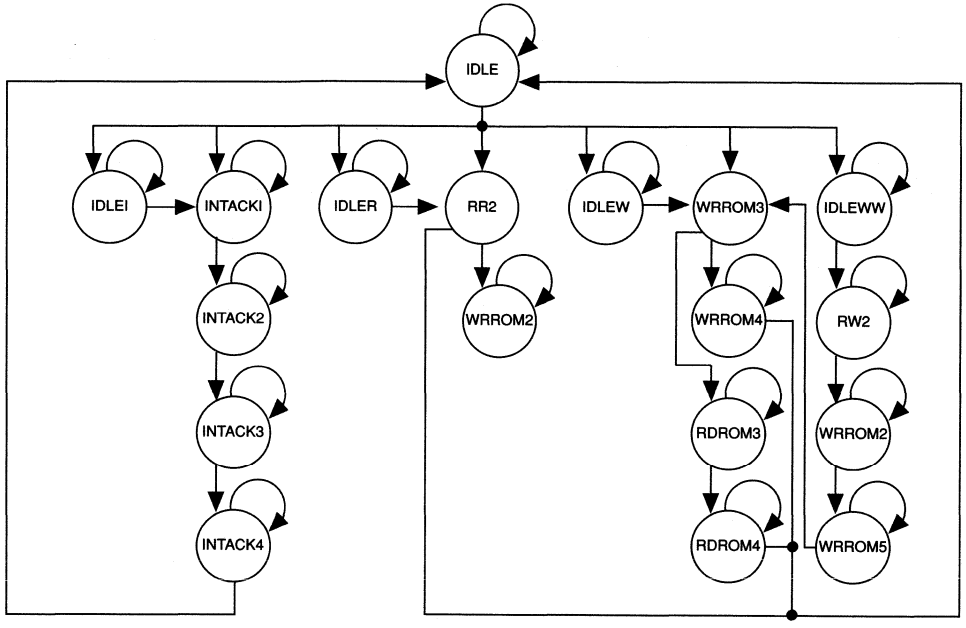
```
SUBDESIGN BUS386
(
    ads, dc, wr, mio, eisa,    : INPUT;
    lmem, aready, romce,       : INPUT;
    reset, clk, clk2           : INPUT;            % Define dedicated inputs  %

    mrdc, mwtc, inta,          : OUTPUT;
    xp0, xp1, xp2              : OUTPUT;           % Define dedicated outputs %
)
VARIABLE
    bustate: MACHINE OF BITS (q[5..0])
            WITH STATES      (
                    rstctl   =   B"000000",      % Bus held in Reset     %
                    idle     =   B"111111",      % Bus in high impedance %
                    idler    =   B"111100",
                    idleww   =   B"111101",
                    idlewz   =   B"111110",
                    idlei    =   B"111001",
                    rr2      =   B"011001",
                    rdrom2   =   B"101011",      % Read firmware EPROM #2  %
                    rdrom3   =   B"011010",      % Read firmware EPROM #3  %
```

Figure 6. Local Microprocessor Interface (BUS386.TDF) (Part 2 of 2)

```
                    rdrom4   =    B"011000",   % Read firmware EPROM #4  %
                    rw2      =    B"101001",
                    wrrom2   =    B"101011",
                    wrrom25  =    B"101101",
                    wrrom3   =    B"111000",
                    wrrom4   =    B"111000",
                    intack1  =    B"110001",   % Interrupt acknowledge #1 %
                    intack2  =    B"110011",   % Interrupt acknowledge #2 %
                    intack3  =    B"110010",   % Interrupt acknowledge #3 %
                    intack4  =    B"110000");  % Interrupt acknowledge #4 %
BEGIN
    bustate.clk      = clk2;                   % Double BCLK freqency to 16.66 MHz   %
    bustate.reset    = reset;                  % System in Reset until power is good %
    mrdc             =   q0;
    mwtc             =   q1;
    inta             =   q2;
    xp0              =   q3;
    xp1              =   q4;
    xp2              =   q5;

        CASE bustate IS
            WHEN rstctl =>
                bustate = idle;

            WHEN idle =>
                IF eisa & !lmem & !ads & clk & (dc # mio) & !wr THEN
                    bustate  = idler;
                ELSIF eisa & !lmem & !ads & clk & (!romce # !mio) & wr & dc THEN
                    bustate  = idleww;
                ELSIF eisa & !lmem & !ads & clk & romce & mio & wr & dc THEN
                    bustate  = idlewz;
                ELSIF eisa & !lmem & !ads & clk & !mio & !wr & !dc THEN
                    bustate  = idlei;
                ELSIF !eisa & !lmem & !ads & clk & (mio # dc) & !wr THEN
                    bustate  = rr2;
                ELSIF !eisa & !lmem & !ads & clk & !mio & !wr  & !dc THEN
                    bustate  = intack1;
                ELSIF !eisa & !lmem & !ads & clk & (!romce # !mio) & wr & dc THEN
                    bustate  = rw2;
                ELSIF !eisa & !lmem & !ads & clk & romce & mio & wr THEN
                    bustate  = wrrom3;
                ELSE
                    bustate  = idle;
                END IF;

            WHEN idler =>
                IF !eisa & clk THEN
                    bustate  = rr2;
                END IF;

            WHEN idleww =>
                IF !eisa & clk THEN
                    bustate  = rw2;
                ELSE
                    bustate  = idleww;
                END IF;
```

Figure 7 shows a state machine diagram for BUS386.TDF.

Figure 7. Local Microprocessor Interface State Machine Diagram



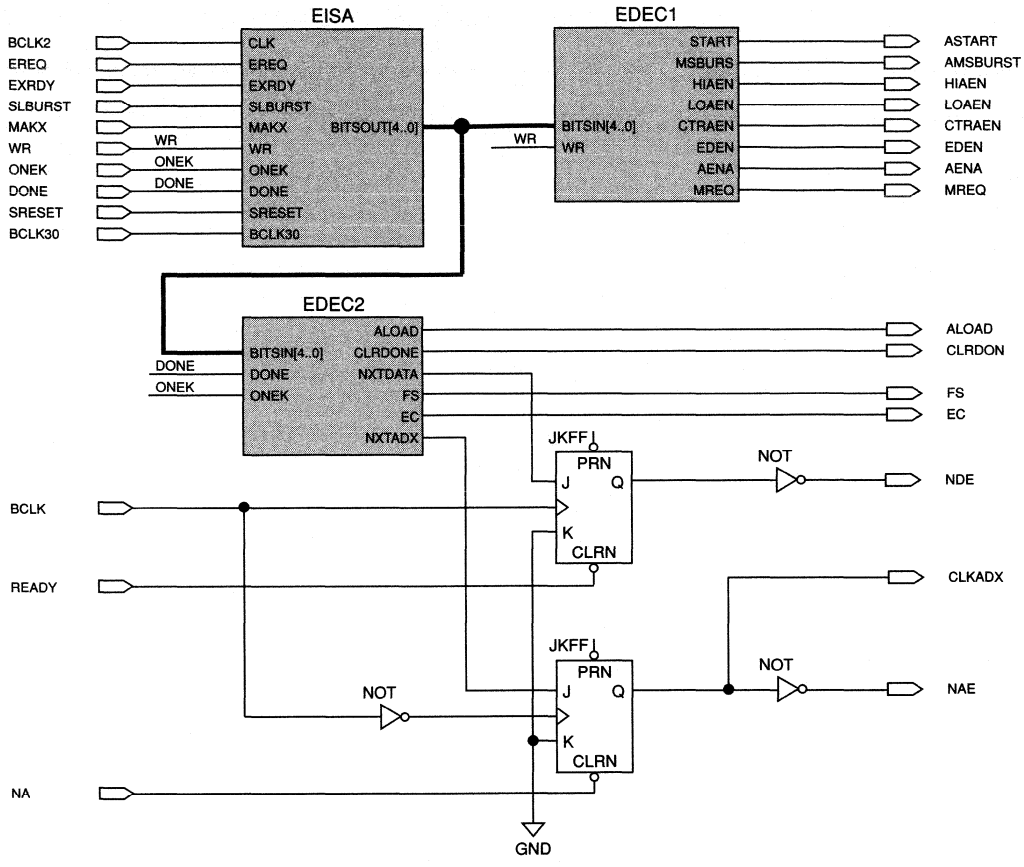
Bus Arbitrator (EISA_IO.GDF)

The complex bus-master capability of the EISA bus requires the EISA add-on cards to be correctly interfaced with each other and with the motherboard to avoid contention and bus conflicts. This process is called bus arbitration. The EISA bus controller (EBC) IC on the motherboard controls bus arbitration, but each add-on card can request temporary control of the system bus to transfer data.

The same arbitration must also be implemented on each add-on card that supports a bus master. You must also configure the EISA bus controller at power-up for burst-transfer capability. MAX 5000 and MAX 7000 EPLDs enable you to perform full 32-bit transfers on the EISA add-on card, instead of the standard 16-bit transfer.

Figure 8 shows the bus arbitrator EISA_IO.GDF. This file contains three file symbols (`EISA`, `EDEC1`, and `EDEC2`) and two flipflops. `EISA` evaluates the EISA bus control signals and generates an internal control word that is routed to `EDEC1` and `EDEC2`. `EDEC1` and `EDEC2` read the control word and generate the appropriate output control signals. The TDFs for the `EISA`, `EDEC1`, and `EDEC2` functions are shown later in this application note.

Figure 8. Bus Arbitration State Machine (EISA_IO.GDF)



When the local microprocessor on the expansion card sends a signal to EISA_IO to request control of the EISA bus, the bus arbitrator requests access to the system bus by asserting the EISA MREQ*x* signal (where *x* is a unique 8-bit computer slot ID). The state machine then waits for the motherboard's EBC to grant access to the bus, and asserts the MAK*x* signal. The EBC, which acts as the bridge between different masters and slaves, performs all cycle translations required to talk to ISA and EISA boards with 8-, 16-, and 32-bit data widths.

After the add-on card has control of the bus, the arbitration state machine generates the START signal to begin the transfer cycle. When the first cycle is run, the motherboard asserts the SLBURST signal low, indicating that it can support burst cycles. If the add-on card has more data to transfer and

supports burst cycles, the add-on card asserts MSBURST to indicate that it will generate burst cycles. This "handshaking" function supports the EISA bus master standard. The EISA interface card takes control of the bus and generates all address and control signals, freeing the host CPU to execute commands and process data stored in the motherboard's local memory.

Input to EISA Control Bus (EISA.TDF)

Figure 9 shows EISA.TDF, which evaluates the EISA bus control signals. This file gives the add-on card access to the EISA bus through the EISA bus control signals, which are described later in this application note. At the beginning of a typical bus cycle, the START control signal goes low for one Clock period. For burst transfer cycles, the MSBURST signal also goes low.

Figure 9. Input to EISA Control Bus (EISA.TDF) (Part 1 of 2)

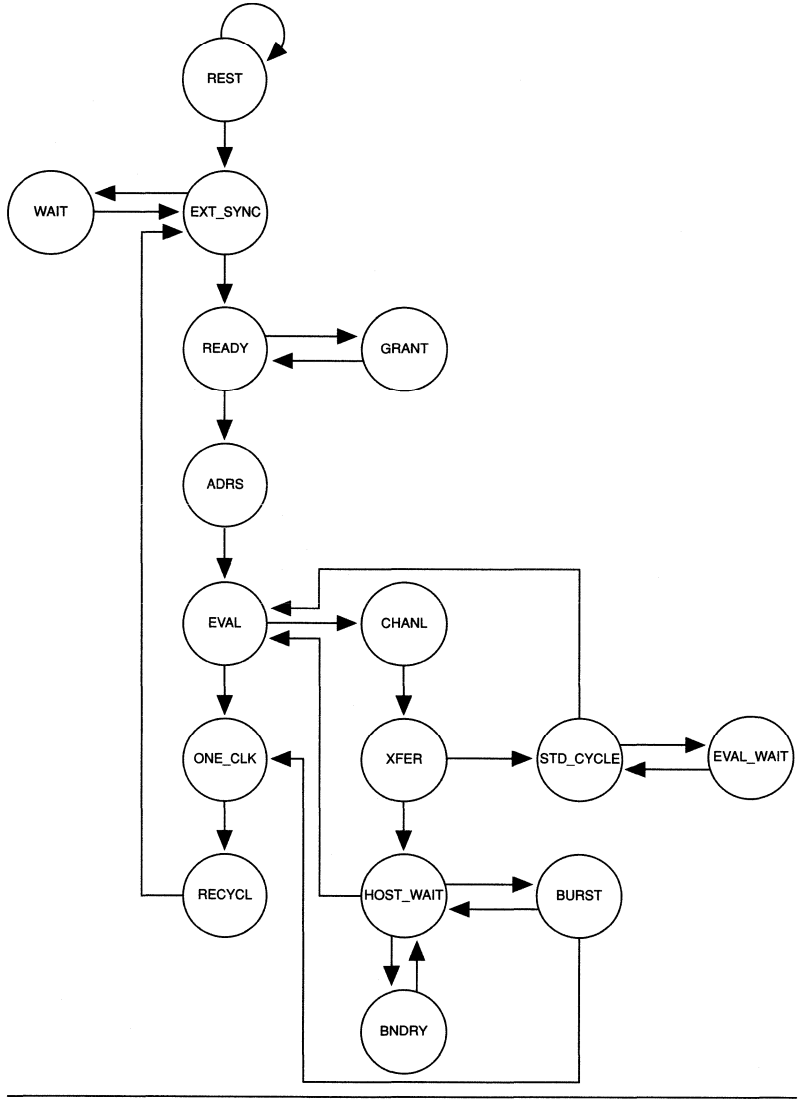
```
SUBDESIGN eisa
(
    clk, ereq,   exrdy, slburst, makx  : INPUT;
    onek, done, sreset, bclk30         : INPUT;
    bitsout[4..0]                      : OUTPUT;
)
VARIABLE
    esm:  MACHINE OF BITS (qbits[4..0])
        with states(
% B4=START, B3=MSBURST, B2=DONE, B1=MREQ, B0=EREQ %
                    rest         = b"11111", % Held in Reset state %
                    wait         = b"00111",
                    ext_sync     = b"11110",
                    ready        = b"11101",
                    grant        = b"11100",
                    adrs         = b"11011",
                    eval         = b"11010",
                    chanl        = b"11001",
                    xfer         = b"11000",
                    std_cycle    = b"10111",
                    eval_wait    = b"10110",
                    host_wait    = b"10101",
                    bndry        = b"10011",
                    burst        = b"10001",
                    recycl       = b"01101",
                    one_clk      = b"01100");
BEGIN
    bitsout[]    = qbits[4..0];
    esm.clk      = clk;
    esm.reset    = sreset;

    CASE (esm) IS
        WHEN rest =>
            IF (!bclk30)  THEN
                esm = rest;              % Held in Reset state         %
            ELSE
                esm = ext_sync;          % BCLK signal is synchronized %
            END IF;
            WHEN wait => esm = ext_sync;
```

Figure 9. Input to EISA Control Bus (EISA.TDF) (Part 2 of 2)

```
        WHEN ext_sync =>
             IF (!ereq) THEN
                  esm = ready;              % Wait for EISA request from processor %
             ELSE
                  esm = wait;
             END IF;
        WHEN ready =>
             IF (!makx) THEN
                  esm = adrs;               % Assert MREQx                 %
             ELSE
                  esm = grant;              % Sample MACKx on rising BCLK %
             END IF;
        WHEN grant => esm = ready;          % Wait to sample on the rising edge %
        WHEN adrs => esm = eval;            % Cycle starts here                 %

        WHEN eval =>
             IF (done) THEN
                  esm = one_clk;
             ELSE
                  esm = chanl;              % Present LA[] and MIO          %
             END IF;
        WHEN chanl => esm = xfer;           % Present BE[] and WR and START %

        WHEN xfer =>
             IF (!slburst & onek & !done) THEN
                  esm = bndry;              % Burst if not a 1 Kbyte page boundary %
             ELSE
                  esm = std_cycle;          % Otherwise use 2BCLK                  %
             END IF;
        WHEN std_cycle =>
             IF (!exrdy) THEN
                  esm = eval_wait;          % Deassert START, check for wait states %
             ELSE
                  esm = eval;               % Perform next address in eval          %
             END IF;
        WHEN eval_wait => esm = std_cycle; % Wait and return to check again       %

        WHEN bndry =>
             IF (!exrdy) THEN
                  esm = host_wait;          % Now in burst mode, check for waitnst  %
             ELSIF (!onek) THEN
                  esm = eval;
             ELSE
                  esm = burst;              % If not, enter burst cycles %
             END IF;
        WHEN host_wait => esm = bndry;

        WHEN burst =>
             IF (done) THEN
                  esm = one_clk;            % 1 Kbyte memory limitation %
             ELSE
                  esm = bndry;
             END IF;
        WHEN one_clk => esm = recycl;
                  WHEN recycl => esm = ext_sync;
    END CASE;
END;
```

A state machine diagram for EISA.TDF is shown in Figure 10.

Figure 10. EISA.TDF State Machine Diagram



Outputs to EISA Control Bus (EDEC1.TDF & EDEC2.TDF)

The outputs of the `EISA` function are decoded by the `EDEC1` and `EDEC2` functions. `EDEC1` decodes the 5-bit control bus `bitsin[4..0]` and asserts

the appropriate control signals onto the EISA bus at the beginning of a bus transfer. In this design, the EXT32 signal is always low, which indicates to the EBC that the bus will transfer only 32-bit data (i.e., double words or "dwords"). EDEC1.TDF includes the MSBURST control signal used to synchronize a "Type C" DMA bus cycle (refer to the Compaq *Technical Reference Guide: Extended Industry Standard Architecture Expansion Bus* for more information). Figure 11 shows EDEC1.TDF.

Figure 11. Outputs to EISA Control Bus (EDEC1.TDF) (Part 1 of 2)

```
CONSTANT rst = B"11111";
CONSTANT w   = B"00111";
CONSTANT xx  = B"11110";
CONSTANT y   = B"11101";
CONSTANT z   = B"11100";
CONSTANT a   = B"11011";
CONSTANT b   = B"11010";
CONSTANT c   = B"11001";
CONSTANT d   = B"11000";
CONSTANT e   = B"10111";
CONSTANT ew  = B"10110";
CONSTANT hw  = B"10101";
CONSTANT h   = B"10011";
CONSTANT k   = B"10010";
CONSTANT l   = B"10001";
CONSTANT kw  = B"01110";
CONSTANT r   = B"01101";
CONSTANT q   = B"01100";

SUBDESIGN edec1
(
    bitsin[4..0], wr              : INPUT;
    start, msburst, hiaen,        : OUTPUT;
    loaen, ctraen, eden, aena,    : OUTPUT;
    mreq                          : OUTPUT;
)

VARIABLE
    st[4..0] : NODE;
BEGIN

    st[]     =            bitsin[];
    start    =            (st[]== d ) #
                          (st[]== c );

    !mburst  =            (st[]== h ) #
                          (st[]== hw) #
                          (st[]== l);

    !ctraen  =    (!wr & ((st[]== b) # % Read function uses counter %
                          (st[]== c ) #
                          (st[]== d ) #
                          (st[]== e ) #
                          (st[]== ew) #
                          (st[]== h ) #
                          (st[]== hw) #
                          (st[]== l )));
```

Figure 11. Outputs to EISA Control Bus (EDEC1.TDF) (Part 2 of 2)

```
!hiaen  =  (((st[]== b ) #          % High address is considered %
             (st[]== c ) #
             (st[]== d ) #
             (st[]== e ) #
             (st[]== ew) #
             (st[]== h ) #
             (st[]== hw) #
             (st[]== l )));

!loaen  =  (wr &  ((st[]== b ) #  % Write functions use latches %
            (st[]== c ) #
            (st[]== d ) #
            (st[]== e ) #
            (st[]== ew) #
            (st[]== h ) #
            (st[]== hw) #
            (st[]== l )));

!mreq   =  (st[]== y ) #
           (st[]== z ) #
           (st[]== a ) #
           (st[]== b ) #
           (st[]== c ) #
           (st[]== d ) #
           (st[]== e ) #
           (st[]== ew) #
           (st[]== h ) #
           (st[]== hw) #
           (st[]== l );

!aena   = (st[]== b ) #
           (st[]== c ) #
           (st[]== d ) #
           (st[]== e ) #
           (st[]== ew) #
           (st[]== h ) #
           (st[]== hw) #
           (st[]== l );

!eden   = (st[]== b ) #
           (st[]== c ) #
           (st[]== d ) #
           (st[]== e ) #
           (st[]== ew) #
           (st[]== h ) #
           (st[]== hw) #
           (st[]== l );

END;
```

The EDEC2 function reads the internal control bus and uses the DONE and ONEK control signals from the EISA bus to determine when a bus cycle is completed or a page boundary has been encountered. Figure 12 shows EDEC2.TDF.

Figure 12. Outputs to EISA Control Bus (EDEC2.TDF) (Part 1 of 2)

```
CONSTANT rst = B"11111";
CONSTANT w   = B"00111";
CONSTANT XX  = B"11110";
CONSTANT y   = B"11101";
CONSTANT z   = B"11100";
CONSTANT a   = B"11011";
CONSTANT b   = B"11010";
CONSTANT c   = B"11001";
CONSTANT d   = B"11000";
CONSTANT e   = B"10111";
CONSTANT ew  = B"10110";
CONSTANT hw  = B"10101";
CONSTANT h   = B"10011";
CONSTANT k   = B"10010";
CONSTANT l   = B"10001";
CONSTANT kw  = B"01110";
CONSTANT r   = B"01101";
CONSTANT q   = B"01100";

SUBDESIGN edec2
(
    bitsin[4..0]              : INPUT;
    done, onek                : INPUT;
    aload, clrdone, nxtdata,  : OUTPUT;
    fs, ec, nxtadx            : OUTPUT;
)

VARIABLE
    st[4..0] : NODE;

BEGIN
    st[] = bitsin[];

    !clrdone = (st[] == q)  #
               (st[] == w)  #
               (st[] ==XX)  #
               (st[] == y)  #
               (st[] == z)  #
               (st[] == a);

    nxtdata =  (st[] == l) #
              ((st[] == b) & (!onek # done)); % EISA data is read in %

    nxtadx  =  (st[] == a) #
               (st[] == h);

    aload   =  (st[] == b);            % load address counter here %


    fs      =  (st[] == c) #
               (st[] == d) #
               (st[] == e) #
               (st[] == ew)#
               (st[] == hw)#
               (st[] == h) #
               (st[] == l) #
               (st[] == r) #
               (st[] == q);
```

Figure 12. Outputs to EISA Control Bus (EDEC2.TDF) (Part 2 of 2)

```
    ec      =  (st[] == y) #
               (st[] == z) #
               (st[] == a) #
               (st[] == b) #
               (st[] == c) #
               (st[] == d) #
               (st[] == e) #
               (st[] == ew)#
               (st[] == h) #
               (st[] == hw)#
               (st[] == r) #
               (st[] == q);

END;
```

System Signal Description

This section describes all signals in the EISA bus interface, which are divided into the following categories:

- Local Signals
- EISA Signals
- Bus Control Signals

Local Signals

Local add-on card signals control and synchronize all local card functions. The local signals in this project pass messages, provide status information, control data flow, and coordinate simple handshaking to and from the local microprocessor.

A_ENA — Enables the tri-state EISA control signals onto the EISA bus.

ALOAD — Indicates that the address counter/latch should load a new address on the next Clock. This signal is used on the first address of each transfer.

BCLK2 — Frequency-doubled version of BCLK. Since the state machine operates only on rising edges, a rising edge must be generated on every rising and falling edge of BCLK.

BCLK30 — Delayed version of BCLK used to synchronize the state machine Clock in the EISA bus state machine

CLRDONE — Clears the DONE signal at the end of a cycle.

DONE	Indicates when the originating expansion card has completed its transfer. This signal must be synchronized so that the burst transfer terminates before an "extra" cycle is performed. Synchronization can be difficult because the EISA addresses are pipelined 1.5 BCLK cycles ahead of the data at the corresponding address. For example, in the EISA bus interface project, the READ and WRITE signals overlap, i.e., the READ operation does not need to be finished before the WRITE signal is asserted.
EC	Indicates that the EISA cycles are active (i.e., the data buffers are still enabled). This signal is used to eliminate bus contention.
EDEN	Buffer Enable for transferring data buffers to the EISA bus
EISA	Starts the bus arbitration state machine that arbitrates for the EISA bus. This signal must be asserted long enough to be recognized by the state machine even when a long BCLK stretch occurs. This function typically takes 120 ns, so the signal should be asserted for at least 160 ns, or be latched by the initiating device and cleared by the bus arbitration state machine.
NEXTDATA	Enables the next data word onto the bus during the next BCLK rising edge.
NF	Synchronizes the DONE signal to the bus phase.
ONEK	Recognizes an address that comes one data word before the 1-Kbyte bus-transfer memory limit is reached. This signal is necessary because the EISA standard restricts burst transfers to within the 1-Kbyte memory limitation. If ONEK is asserted, the state machine does not rearbitrate, but simply performs two BLCK cycles before it returns to bursting. It performs the same function at the beginning of the cycle.

The following three signals control address generation in the system. For example, in the EISA bus interface, CTRAEN controls the count address, and HIAEN and LOAEN control the data latches. This control mechanism is necessary because system timing constraints require both data latches to be loaded with one half of the 32-bit data dword before data can be read onto the EISA bus. In a burst operation, these controls must supply a dword every 120 ns; therefore, a 16-bit word is either read or written every 60 ns.

CTRAEN — Buffer Enable for read operations. This signal enables the lower address bits EADS[9..2].

HIAEN — Buffer Enable for the upper EADS[31..10] address bits and byte enable signals BE[3..0]. These signals are always enabled during 32-bit transfers on the EISA bus.

LOAEN — Buffer Enable for write operations. This signal enables the lower address bits EADS[9..2].

EISA Signals

EISA signals connect the add-on card to the EISA bus. All EISA signals are synchronized by BCLK, the EISA bus Clock. EISA control signals allow multiple types of add-on cards to share I/O bus bandwidth through a standardized set of bus cycles. The motherboard performs the cycle-size translation.

A_MSBURST — EISA MSBURST signal before it goes through a buffer. This signal indicates to the system that burst cycles will be performed.

A_START — EISA START signal before it goes through a buffer. This signal indicates the beginning of an EISA cycle.

BCLK — Bus Clock signal from the motherboard

EXRDY — Indicates whether wait states are required for the cycle to finish. This signal was not used in the EISABLK project because no wait states exist. However, you may need to include it in other projects. Due to the BCLK delay and the short hold time, EXRDY may need to be delayed or latched to be properly recognized.

MREQ — Requests control of the EISA bus.

SLBURST — Indicates to the bus master that the memory is sufficient to perform burst transfers.

Bus Control Signals

Bus control signals provide the basic system Clock timing, address validation, and read/write strobing between the local microprocessor and the EPROM and local SRAM buffers.

ADS	Address valid strobe signal
CLK	25-MHz system Clock signal
CLK2	50-MHz system Clock signal
DONE1	Unlatched version of the DONE signal
LMEM	Control signal for bus arbitration state machine. When this signal is high during an 80386 local processor bus cycle, the cycle is translated across the EISA bus.
MIO	Memory I/O signal
NA	Next-address pipelining signal. This signal indicates that the computer should initiate address pipelining.
NAE	Indicates to the glue logic that next address is required. This signal is connected to the data (Q) output of the flipflop with the NXTADX signal as the input.
NDE	Indicates to the glue logic that the next data word is required. This signal is connected to the data (Q) output of a flipflop with the NXTDATA signal as the input.
READY	Cycle complete signal
ROMCE	Indicates that ROM is being accessed.
WR	Write/read signal
X0-4	State variables used to distinguish between states in which READY and NA have the same value.
XP0-2	Determines whether a local cycle needs a READY signal.

Design Entry

You can enter the design for the EISA bus interface (EISABLK.GDF) with the MAX+PLUS II Graphic Editor. The design fits into one Altera EPM7256 EPLD, requiring 88 macrocells, and leaving 168 macrocells available for future expansion, and uses 101 pins, leaving 63 for future design additions. All files in this project are available on the Altera Applications BBS at (408) 249-1100. You can download the files in the self-extracting archive file AN27.EXE. Contact Altera Applications at (800) 800-EPLD for more information.

Conclusion

The local processor interface that decodes the local processor memory map and the bus arbitration state machine require a large number of macrocells and a high I/O count to transfer 32-bit datawords to the datapath. The EPM7256 EPLD is the first programmable logic device with sufficient logic density and I/O capabilities to support the level of integration required for this design. This design can also be partitioned among multiple MAX 5000 EPLDs.

This 32-bit EISA bus interface design offers a high-performance transfer rate. Registered performance timing analysis with MAX+PLUS II software shows that an EPM7256 EPLD programmed with this design can be clocked at significantly faster speeds than the 8.33-MHz bus speeds found in existing EISA motherboards. This high-speed device performance ensures that EISA interface boards designed today will work with systems designed in the future.

Waveform Design Entry

April 1992, ver. 1

Application Note 28

Introduction

The Altera MAX+PLUS II development system includes three powerful design editors: the Graphic, Text, and Waveform Editors. While most designers are familiar with the text and graphic design entry tools, the Waveform Editor has proven to be one of the most versatile features of the MAX+PLUS II software. It is not only a tool for quickly entering test vectors and viewing simulation results, it is also a design editor that can be used to describe logic. This application note discusses the following topics:

- When to choose waveform design entry
- Fundamental concepts of waveform design entry
- Waveform design entry guidelines
- Sample designs
- Special considerations for waveform design entry

In general, this application note does not describe the mechanics of entering or editing a file in the Waveform Editor. See MAX+PLUS II Help for detailed information on entering and manipulating nodes and waveforms.

When to Choose Waveform Design Entry

MAX+PLUS II software provides three design entry options that allow you to choose the right tool to develop each section of a design. The MAX+PLUS II Graphic Editor allows you to enter a design (called a "project" in MAX+PLUS II) as a schematic with an extensive library of basic gate primitives and TTL macrofunctions. You can use the MAX+PLUS II Text Editor to enter projects that include complex state machines, decoding operations, prioritized sequential logic, and TTL macrofunctions in the Altera Hardware Description Language (AHDL). You can also create Waveform Design Files (.WDF) with the MAX+PLUS II Waveform Editor. WDFs are best suited to circuits that have well-defined sequential inputs and outputs, such as state machines, counters, and registers; combinatorial functions decoded from counters; and other repeating functions. Waveform design entry is not recommended for arbitrary combinatorial logic.

Fundamental Concepts of Waveform Design Entry

A WDF consists of a combination of sequential input node logic levels and desired output logic levels that define the behavior of a circuit. WDFs can contain both ordinary and state machine (i.e., symbolic) inputs, and combinatorial, registered, and state machine outputs. Buried logic—consisting of combinatorial, registered, or state machine logic—is optional. It is used only to indicate the type of logic that may lie between the inputs and outputs. For each different combination of input waveforms, you

draw the desired output waveforms. Between logic level transitions, each set of stable waveforms constitutes a separate "time-slice."

Although a WDF includes a horizontal time scale, it functions only as an order-of-occurrence indicator, not an absolute time scale. However, since both output- and buried-type registered and state machine nodes are clocked, changes on inputs that affect these nodes require a non-zero time-slice to provide the setup time. The minimum setup time is 0.1 ns, the smallest time increment available in the Waveform Editor. The hold time is always zero.

The MAX+PLUS II Compiler interprets a WDF as the relationship between inputs and outputs, which are mapped as a truth table with one row of inputs and outputs for each time-slice in the file. The Compiler generates the logic necessary to create the outputs on the basis of these inputs. The input portion of this truth table consists of the current values of all input and output nodes. The output portion of the table consists of the current value of combinatorial output nodes, in addition to the value of registered or state machine output nodes from the *next* time-slice. Both buried and output waveforms that represent registered and state machine logic may depend on the value of combinatorial nodes that have remained stable from the previous time-slice. Therefore, logic is interpreted sequentially.

The Compiler uses a "minimum logic" rule when generating the equations for WDFs, also known as "what you see is what you get" (WYSIWYG). The Compiler creates equations for outputs on the basis of the inputs, and groups these equations together for logic synthesis. The exact input conditions that you specify are guaranteed to produce the requested outputs, but no assumptions are made for unspecified combinations of inputs. The Compiler takes the inputs you have drawn, and, *based on the order in which they are drawn,* generates the logic needed to create the outputs you have specified. The resulting circuit is guaranteed to operate properly if it is presented with the exact sequence of input waveforms shown in the WDF. However, if all possible combinations of inputs are not specified, the Compiler assumes that all output nodes have a "don't care" (X) value for any combination of inputs not specified in the WDF. The Compiler uses these don't care conditions to generate only the minimum logic necessary to implement the requested outputs and to use EPLD resources as efficiently as possible. The minimum logic rule provides efficient logic synthesis, but requires you to define a function completely. If you do not specify all possible combinations of the inputs, the Compiler may not create the desired function.

Waveform Design Entry Guidelines

The following "golden rules" are essential for effective waveform design entry. Examples that illustrate these guidelines are given later in this application note.

- ❑ Use a WDF to define all logic for a project, or to define a bottom-level file in a hierarchical project. You cannot use WDFs at intermediate levels of hierarchy.
- ❑ Describe all combinations of inputs and completely specify the desired outputs based on those inputs.
- ❑ Ensure that each combination of input and output nodes is unique. If different outputs exist for identical combinations of inputs, the Compiler cannot determine which outputs are "correct," and cannot add buried logic to remove ambiguity.
- ❑ For sequential logic, draw inputs and outputs in the order in which they will occur during normal operation of the circuit.
- ❑ For non-sequential logic, e.g., purely combinatorial functions, use a waveform separator—shown as a thin vertical line—to specify the beginning of a new waveform sequence that does not depend on the preceding sequence.
- ❑ Limit purely combinatorial functions to those with a small number of inputs. As the number of inputs increases, the possible combinations of these inputs expand exponentially. WDFs are ideally suited for functions with a limited number of sequential inputs and a large number of outputs.
- ❑ If a function is cyclical, show the last set of conditions looping back to the first by repeating the first time-slice at the end of the cycle.
- ❑ Specify a Clock input as a "secondary input" for each registered and state machine node. You can specify Clock, Reset, and Preset signals as secondary inputs when you create or edit the node.
- ❑ Specify transitions on registered functions, i.e., buried and output nodes that represent registered or state machine logic, only at rising Clock transitions.
- ❑ To satisfy the setup time, change the logic levels of inputs that affect registered functions at least 0.1 ns before a rising Clock transition.
- ❑ For clarity, Altera recommends that you draw inputs that affect registers only on falling Clock edges. Since there is no delay between inputs and combinatorial outputs that are not register-dependent, you should also draw transitions on combinatorial outputs at the same time as the transitions on the inputs, i.e., on falling Clock edges.
- ❑ You can import and export signals and state machines to other design files. An imported state machine does not require the setup time and Clock input that are needed for buried and output state machines created in the WDF.
- ❑ Use any grid size or time-slice length you like—the time scale is arbitrary and indicates only a sequential order of operation, not a specific response time.

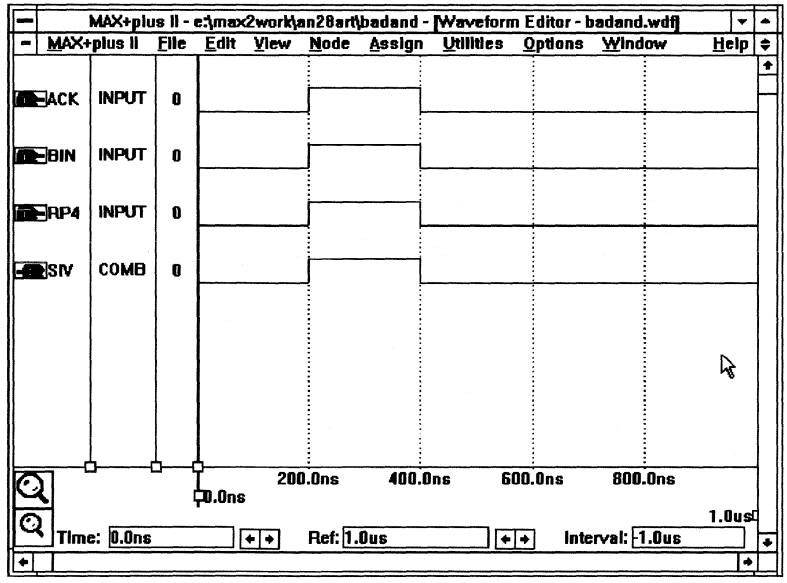
Sample Designs

The following examples illustrate waveform design entry and design guidelines.

Example 1: 3-Input AND Gate

Figure 1 shows BADAND.WDF, a simple 3-input AND function drawn at an arbitrary 200-ns grid size. The first and second 200-ns intervals describe the combinations of the ACK, BIN, and RP4 inputs that cause the output SIV to be low and high, respectively.

Figure 1. Erroneous AND3 Function (BADAND.WDF)



MAX+PLUS II issues the following warning messages when you compile the project:

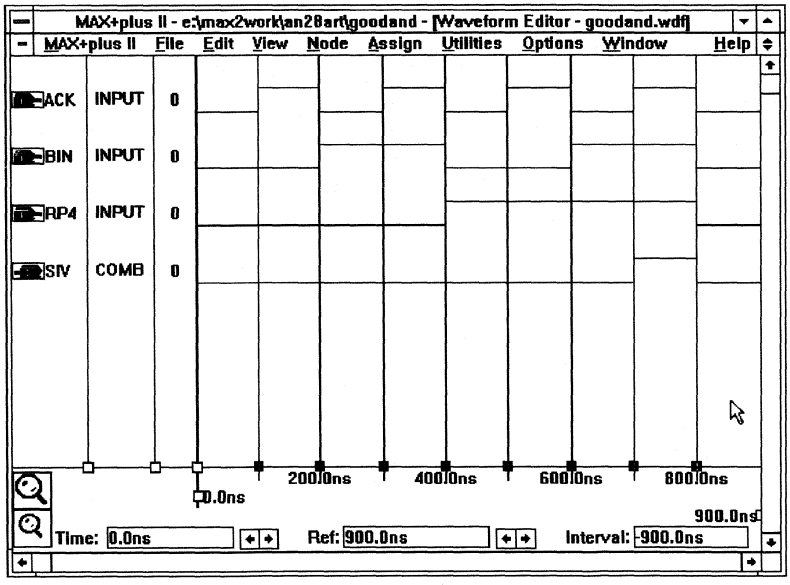
```
Ignored unnecessary input pin 'BIN'
Ignored unnecessary input pin 'RP4'
```

This WDF declares one input combination that causes SIV to be high and one that causes it to be low, but gives no other information because it omits the six other possible input combinations. You might assume that SIV would be set to low for unspecified combinations of inputs, but the Compiler's minimum logic rule sets SIV equal to ACK. Since the Compiler assumes don't care values for unspecified combinations of inputs, it sets SIV equal to ACK because it produces the minimum logic necessary to

generate the desired outputs specified in the WDF. If you do not fully specify the behavior of a circuit, the Compiler always takes advantage of this "incompleteness" to minimize logic, and may yield unexpected results. In this case, the expected result was `SIV = ACK & BIN & RP4;` the actual result is `SIV = ACK.`

Figure 2 shows GOODAND.WDF, which ensures completeness by defining all of the eight possible combinations of inputs for the `AND3` function in Figure 1. This WDF also includes waveform separators between each of the time-slices.

Figure 2. Corrected AND3 Function (GOODAND.WDF)



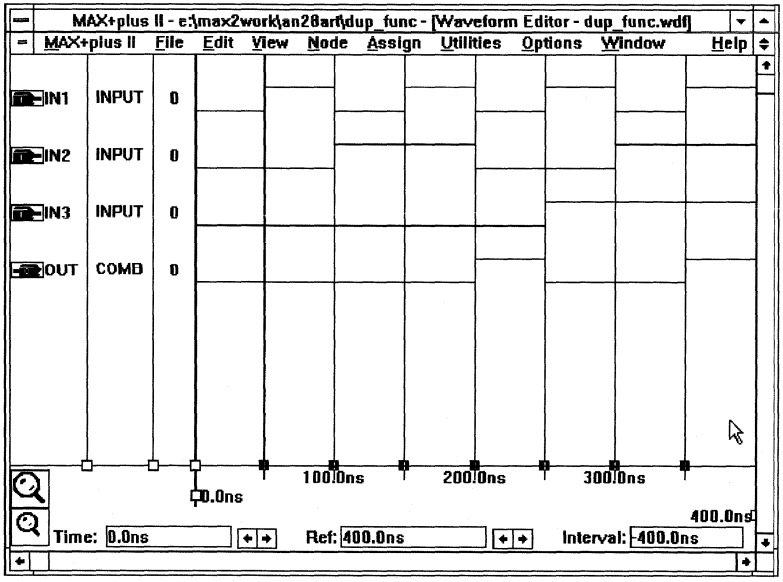
GOODAND.WDF shows that waveform separators are crucial in defining purely combinatorial functions. Although the time scale in a WDF does not imply a specific response time, it does imply a sequential order of operation. In sequential logic, you draw waveforms as they will occur when the circuit is operated normally. In arbitrary logic, you use a waveform separator to break up the order of occurrence implied by the time scale. A waveform separator is a break that specifies the beginning of a new waveform sequence that is not dependent on the preceding sequence. The time-slice and grid size change from BADAND.WDF to GOODAND.WDF is unimportant. The Compiler does not synthesize logic for a particular propagation delay; instead, it synthesizes the logic necessary to generate the signals in the shortest possible time, based on the architecture of the target EPLD family.

This example also illustrates the importance of limiting inputs for purely combinatorial functions. When it generates the logic for a WDF, the Compiler uses the present value of all inputs and outputs to create the desired outputs. As the number of inputs increases, the possible combinations of these inputs expand exponentially. A 2-input function has 4 possible combinations of inputs, a 4-input function has 16, etc. Since you must explicitly declare every possible combination, you should use a WDF only if a function has a small number of inputs.

Example 2: 3-Bit Address Decoder

When you create a WDF, you must ensure that each combination of inputs is associated with only one set of outputs, since the Compiler cannot determine which outputs to use if a conflict exists. Figure 3 shows DUP_FUNC.WDF, a 3-bit address decoder that includes duplicate input conditions. Different outputs are shown on the OUT node for the same set of inputs in the time-slices 0 to 50 ns and 200 to 250 ns.

Figure 3. Illegal Duplicate Input Conditions (DUP_FUNC.WDF)



MAX+PLUS II issues the following error message when you compile the project:

```
Node 'OUT' has same set of inputs but different values at
  0.0ns and 200.0ns
```

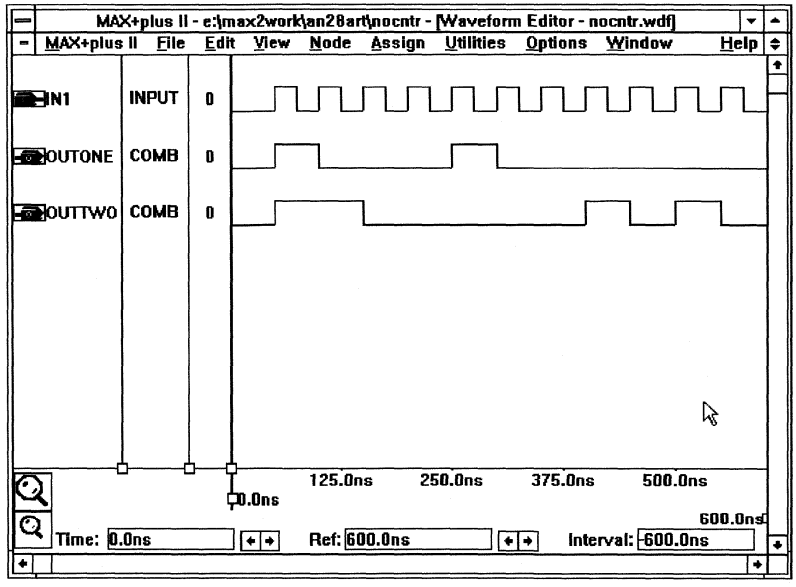
The inclusion of waveform separators in the file does not affect the error. The function compiles correctly only if each output is defined by a unique combination of inputs. To make the combination of inputs unique in this example, edit the interval 200 to 250 ns on the IN3 waveform in Figure 3 to have a high logic level. This change makes the three inputs step sequentially through all binary values from 0 to 7.

Example 3: Counter Decoder

The Compiler cannot insert logic to make an incompletely specified WDF compile correctly. For example, Figure 4 shows NOCNTR.WDF, a file that attempts to create an arbitrary series of transitions on the outputs OUTONE and OUTTWO based on the number of transitions that have occurred on IN1.

Figure 4. Erroneous Counter Decoder (NOCNTR.WDF)

This file includes outputs that change at arbitrary transitions.



Since the Compiler does not count transitions, compiling NOCNTR.WDF causes the following errors:

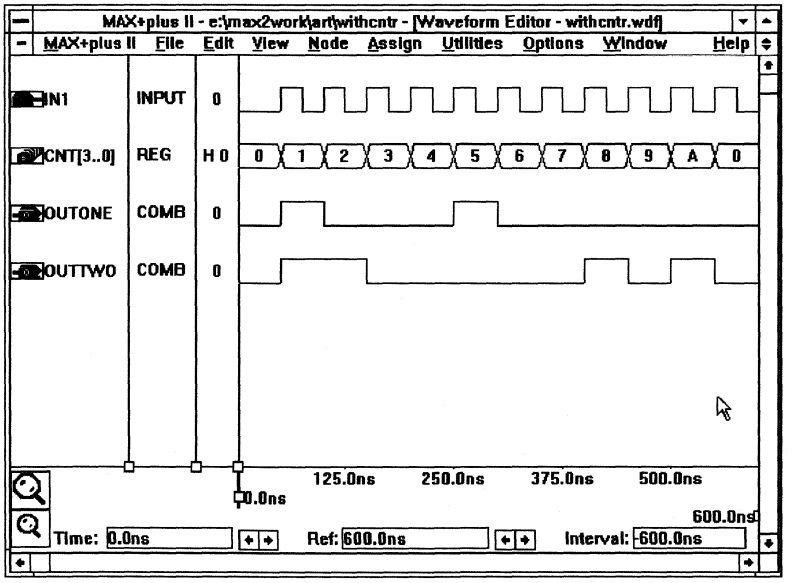
```
Node 'OUTONE' has same set of inputs but different values at
   50.0ns and 150.0ns
Node 'OUTTWO' has same set of inputs but different values at
   50.0ns and 200.0ns
```

The Compiler does not add buried logic such as sequence counters to remove ambiguity from conflicting outputs. Changes in WDF outputs must result from changes on input or buried nodes. Buried node changes must, in turn, result from changes on other nodes.

Figure 5 shows WITHCNTR.WDF, a file that includes the counter `CNT[3..0]` to correctly implement the desired outputs specified in NOCNTR.WDF. The Compiler decodes the value of the buried register `CNT[3..0]` to derive the logic for the combinatorial `OUTONE` and `OUTTWO` outputs. The `IN1` signal is specified as the Clock input to `CNT[3..0]`. Because `CNT[3..0]` is a registered function, it has transitions only at positive Clock edges. Because the counter decoder is a sequential function, no waveform separators are required in the WDF.

Figure 5. Corrected Counter Decoder (WITHCNTR.WDF)

This file includes outputs that change at specified counter values.



You can specify a Clock as a "secondary input" to a registered or state machine node when you create or edit the node with the Waveform Editor's **Enter Node** command. A quick way to create a Clock waveform at the current grid size is to select the whole Clock waveform by clicking Button 1 on the Value field, choose the **Overwrite Count Value** command (Edit menu), and choose **OK** to accept the default value.

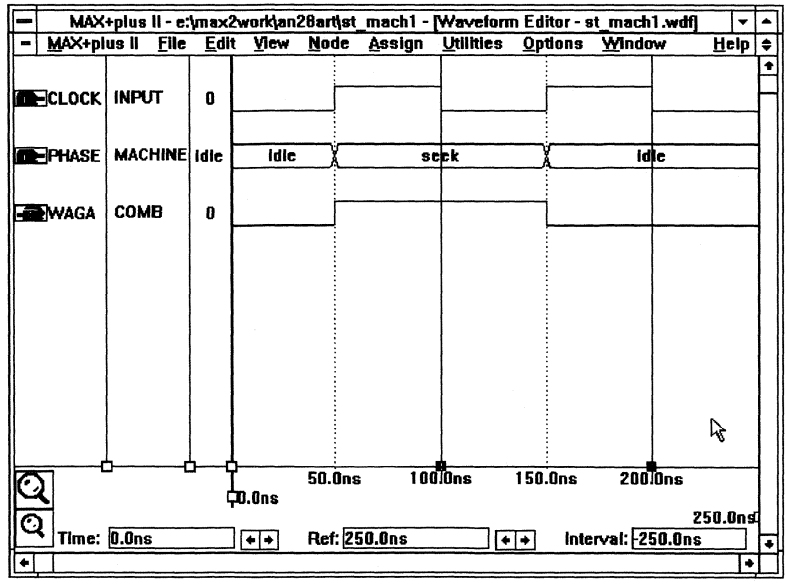
The counter `CNT[3..0]` can be created within the WDF as a group of buried nodes (as shown in Figure 5), or created in a separate design file and imported as a group of input nodes. As an alternative, you can add a buried state machine node and give it a different state name in each of the different time-slices. The Compiler then adds any extra registers that are needed to create the transitions on `OUTONE` and `OUTTWO`.

Example 4: Simple State Machine

Figure 6 shows ST_MACH1.WDF, a simple state machine. Each state has only two possible destinations: the current state (do nothing) and one other state. The `CLOCK` signal is specified as the Clock input to the `PHASE` machine. Under these conditions, the state transition equations are very simple, and the state machine lends itself to the WDF description. In this type of cyclical function, the last state is the same as the first, so the last

Figure 6. Simple State Machine (ST_MACH1.WDF)

Each state in this state machine has only two possible destinations.



time-slice for the function must be the same as the first to "close the loop." In this example, all combinations of inputs are specified; however, waveform separators are used to eliminate any sequential dependencies.

State machines are natural candidates for WDF implementation. State transitions can be easier to understand in a waveform representation than bubble diagrams, and often yield a better design. However, you must remember that in purely combinatorial functions, only the various logic levels on the current inputs and current combinatorial outputs are used to generate the outputs. In state machines and other types of registered logic, setup times and positive Clock edges also affect output signals.

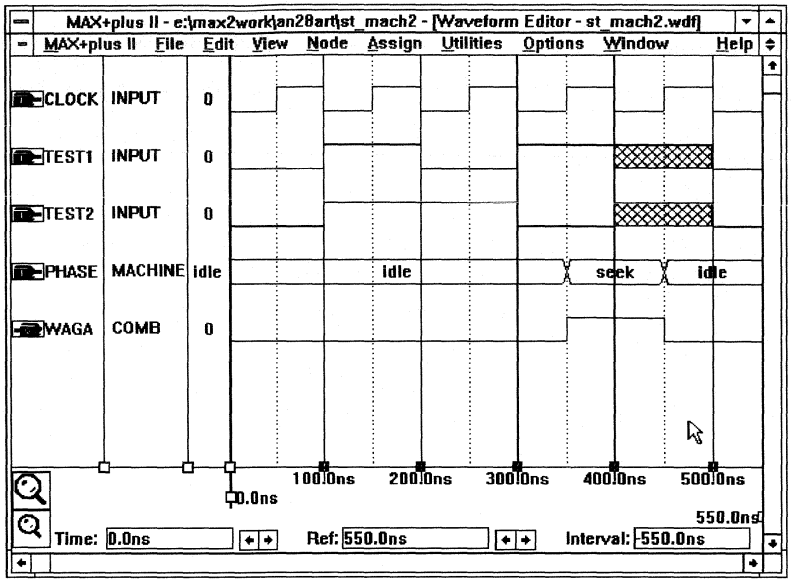
In ST_MACH1.WDF, the PHASE state machine is implemented as buried logic in the WDF. You can also create a state machine in a separate design file and use its outputs in a WDF, or vice versa. The Compiler preserves state names across the hierarchy of design files, allowing you to use the design entry tool that is most appropriate for a particular function. If the PHASE state machine in ST_MACH1.WDF had been imported from a different design file, it would appear as a node with an I/O type (shown in the node handle) of input (I) instead of buried (B). An imported input state machine does not require the setup time and Clock input that are needed for buried and output (O) state machines created in the WDF.

Example 5: State Machine with Conditional Branching Inputs

ST_MACH2.WDF, shown in Figure 7, adds complexity to the function shown in Figure 6 by including the conditional branching inputs TEST1 and TEST2. Each of the possible combinations of these inputs is specified in four contiguous 100-ns time-slices, with the conditions that cause the machine to go from the idle state to the seek state given in the fourth time-slice (TEST1=TRUE and TEST2=FALSE). As in Figure 6, the CLOCK signal is specified as the Clock input to the PHASE machine.

Figure 7. State Machine with Conditional Branching Inputs (ST_MACH2.WDF)

This state machine specifies all possible combinations of inputs.



The waveforms are drawn in this order for convenience: since the input waveforms are not necessarily expected to be sequential, waveform separators at 200-ns intervals disassociate any sequential order of occurrence for the conditional branching inputs TEST1 and TEST2. Without a waveform separator, the value of registered or state machine output nodes from the next time-slice are included in the input portion of the Compiler's truth table description of the current time-slice. The only sequential order of occurrence is implied by the CLOCK input, which goes high halfway through each 100-ns time segment defined by the pairs of waveform separators. Buried and output nodes that represent registered and state machine logic are registered functions that can have transitions only at positive Clock edges. Therefore, the TEST1 and TEST2 inputs change on the falling edge of the CLOCK signal to implement the required non-zero setup time for registered and state machine nodes. The PHASE machine changes states on the subsequent rising CLOCK edges at 350 and 450 ns.

Example 6: Complex State Machine

A complex state machine can have multiple branching paths. If a state machine has more than one possible destination from any given state, you can use waveform separators to break up the order of occurrence implied by the time scale. Figure 8 shows ST_MACH3.WDF, a state machine that uses a waveform separator to implement two possible destinations from state `idle`.

Figure 8. Complex State Machine (ST_MACH3.WDF)

A complex state machine can use waveform separators to indicate that multiple input-output combinations can occur at the same time.

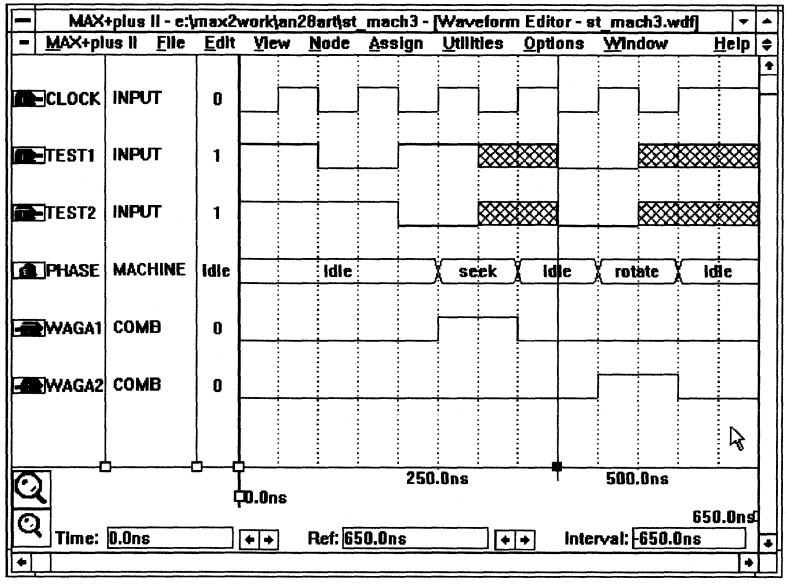


Table 1 shows the four combinations of the two inputs during state `idle`, which provide a complete specification during that state. Table 1 also shows how don't care conditions during states `seek` and `rotate` ensure that the state machine is completely specified, and that the state machine defaults directly to state `idle` on the next positive Clock edge. These six combinations provide a complete set of equations for the state transition sequence `idle-to-seek-to-idle`.

Table 1. Conditions for ST_MACH3.WDF

Time (ns)	Current State	Condition
50	idle	When TEST1 is true and TEST2 is true and the current state is idle, then the next state is idle.
150	idle	When TEST1 is false and TEST2 is true and the current state is idle, then the next state is idle.
250	idle	When TEST1 is true and TEST2 is false and the current state is idle, then the next state is seek.
350	seek	When TEST1 and TEST2 are any values and the current state is seek, then the next state is idle.
450	idle	When TEST1 is false and TEST2 is false and the current state is idle, then the next state is rotate.
550	rotate	When TEST1 and TEST2 are any values and the current state is rotate, then the next state is idle.

The last time-slice in the file (600 to 650 ns) closes both of the cyclical loops in the state machine by essentially duplicating both the first time-slice (0 to 50 ns) and the time-slice immediately following the waveform separator (400 to 450 ns). Both loops are closed because the don't care waveforms in the 600-to-650 ns time-slice are equivalent to the high and low waveforms in both the 0-to-50 and 400-to-450 ns time-slices. The TEST1 and TEST2 inputs change on falling Clock edges to ensure that the idle, seek, and rotate states are valid at the next rising Clock edge.

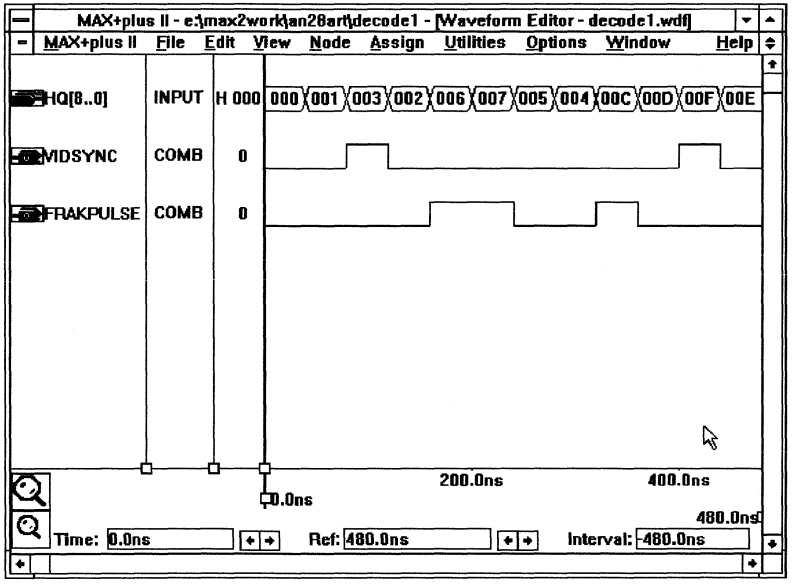
Example 7: Combining WDFs & TDFs

The inputs to a WDF are often counter values or state values, which can control the logical flow, generate other functions, or can be decoded to generate complex output patterns. A good example is a dishwasher control that sends out a series of control pulses that are not easily expressed as a repeating function. Waveforms with transitions at arbitrary counter values are suitable for WDFs. Functions that have a mathematical basis, such as the control signals in the NTSC video specification described in *Application Note 25 (Controlling Complex CCD Imaging Systems with the EPS464 STG EPLD)* in this handbook, may be better suited to AHDL implementation.

A WDF can function as a self-documenting design file that is easily updated, even by engineers who are only minimally familiar with the original application. Figure 9 shows part of DECODE1.WDF, which uses the value of a 9-bit counter as the input, and generates two output signals that are based on specific count values. Although these output functions can be described in an AHDL TDF, the graphical display of the WDF clearly shows the input-output relationships, even in very long files.

Figure 9. DECODE1.WDF

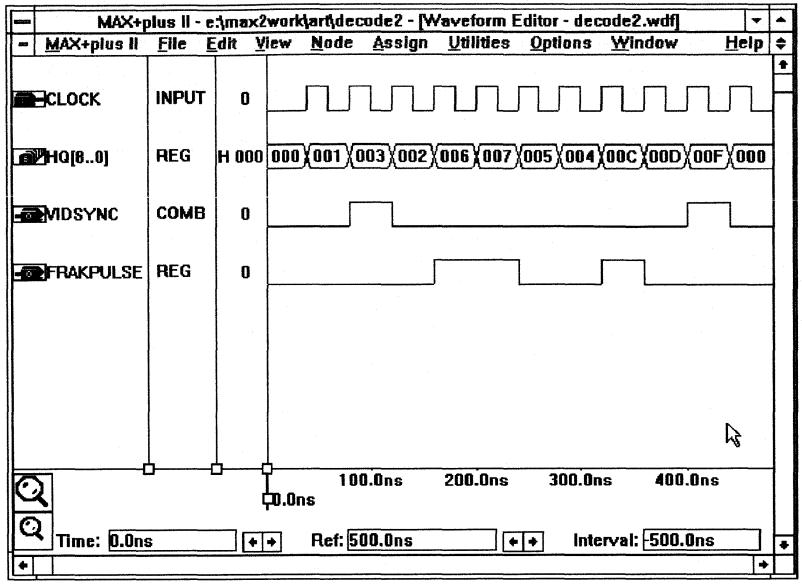
The Gray-code counter HQ[8..0] used in this WDF is created in a separate design file.



The counter for Figure 9 is imported from a different design file, which would typically be an AHDL TDF. The TDF format is very compact, easy to modify, and ideal for creating complex counters. You can also create counters in a WDF, as shown in DECODE2.WDF. See Figure 10.

Figure 10. DECODE2.WDF

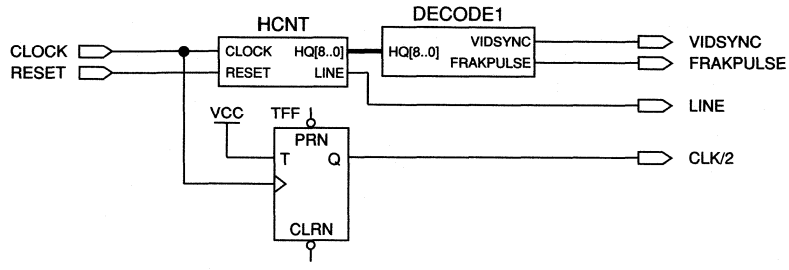
The Gray-code counter HQ[8..0] is created in this WDF rather than in a separate design file.



This WDF includes an additional Clock input that is not present in DECODE1.WDF. The counter `HQ[8..0]` is based on this Clock signal. The output `FRAKPULSE` is also a registered rather than a combinatorial node, and is clocked by the same `CLOCK` node that drives the counter `HQ[8..0]`. Implementing `FRAKPULSE` as a registered output rather than a combinatorial output reduces its potential susceptibility to glitches, because it is explicitly clocked by the same `CLOCK` signal as the counter. To implement the same functionality in DECODE1.WDF (shown in Figure 9), you must add the same input Clock signal that drives the `HQ[8..0]` counter to the file, and specify it as the Clock input to the counter with the **Enter Node** command.

Figure 11 shows PULSER.GDF, which implements a top-level connection in a Graphic Design File (.GDF) between the complex counter decoding in DECODE1.WDF, and a complex Gray-coded counter in a TDF (not shown). You can combine all three MAX+PLUS II design entry methods (text, graphic, and waveform) in a project hierarchy.

Figure 11. PULSER.GDF



Special Considerations

The MAX+PLUS II Waveform Editor is a powerful tool for entering and simulating circuit designs. To benefit from this design entry tool, you should understand the following special simulation requirements and design entry limitations.

State Machine Simulation

When you use a WDF to create a state machine, the file does not actually contain one node per state bit: it simply contains a virtual node with the name of the state machine. The Compiler generates state bits during logic synthesis. When you wish to simulate the project, you must create a Simulator Channel File (.SCF) or Vector File (.VEC) that contains the state bits actually used in the state machine. Fortunately, the Waveform Editor's **Create Default Channel** command can automatically generate a default SCF that contains all nodes in the synthesized project, including state bits. These state bits are automatically combined into a group (bus) with the same name as the state machine. The waveform of this state machine group also displays actual state names instead of numerical values when the logic levels of the state bits correspond to a known state name.

If the SCF or Vector File used for simulation includes state machine groups, the Simulator can automatically update all Compiler-generated state bit names within each group if they change when you recompile the project.

Refer to MAX+PLUS II Help for additional information on SCFs and Vector Files. See *Application Brief 92 (Simulating Internal Nodes with MAX+PLUS II Software)* in this handbook for more information on simulating state machines.

Suitable Applications

Waveform design entry works best for applications that are sequential, have limited inputs, require limited branching, and can be completely expressed.

Some projects that seem appropriate for waveform design entry can turn out to be unwieldy. For example, if many thousands of Clock transitions occur between output changes, the size of a WDF can grow very rapidly to the point where it is difficult to get a perspective on the entire project. In addition, output changes that occur at widely spaced and hard-to-read Gray-code count values can be impractical. Such "long-time projects" are often more easily implemented in AHDL and expressed as repeating functions or simplified with constants.

Even if a WDF is not suitable for describing a complete project, it can be a powerful tool when combined with other types of design files (e.g., GDFs and TDFs) in a hierarchy.

Conclusion

The Waveform Editor is a remarkably versatile and feature-rich tool that can be used for a wide variety of design entry applications as well as simulation. Circuits with sequential inputs and outputs, state machines, and counter decoding are easy to implement with a WDF. WDFs can also be included in a hierarchy of other design files. By understanding the advantages and limitations of waveform design entry, you can benefit from Altera's newest design entry tool.

Asynchronous Latches in Classic, MAX & STG EPLDs

April 1992, ver. 4

Application Brief 9

Introduction

This application brief discusses how to implement asynchronous latches in Altera Classic, MAX 5000, MAX 7000, and STG EPLDs with the MAX+PLUS II development software.

The following topics are covered:

- SR Latches
- Transparent D Latches

SR Latches

The MAX+PLUS II TTL MacroFunction Library provides two SR latches, NANDLTCH and NORLTCH, that are built from cross-coupled expanders. (Refer to MAX+PLUS II Help for information on MAX+PLUS II libraries.) The MAX 5000, MAX 7000, and STG EPLD families support latches built from expanders.

You can also build SR latches with logic primitives and the MCELL buffer used for EPLD macrocell feedback. Latches built with MCELL buffers are supported by Classic, MAX 5000, MAX 7000, and STG EPLDs.

The MAX+PLUS II automatic-symbol-generation feature allows you to create symbols for these latches, which you can then enter as functional blocks in the MAX+PLUS II project.

Figure 1 shows the graphic implementation of the NAND-NAND and NOR-NOR SR latches.

Figure 1. Schematic Implementation of NAND–NAND and NOR–NOR SR Latches

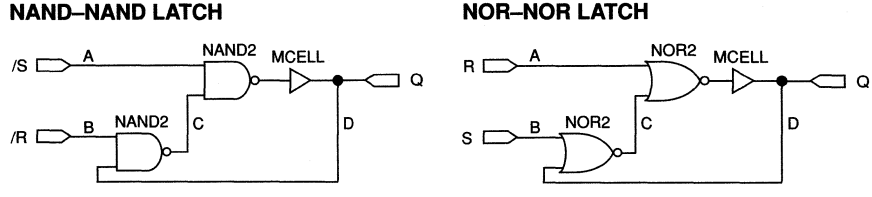


Figure 2 shows the AHDL Text Design File (.TDF) equivalents for these two functions.

Figure 2. AHDL Implementation of NAND–NAND and NOR–NOR SR Latches

AHDL Equivalent of NAND–NAND

```
SUBDESIGN nandnand
(
     /s, /r              : INPUT;
     q                   : OUTPUT;
)

VARIABLE

     a, b, c, d          : NODE;

BEGIN

     a = /s;
     b = /r;
     c = b !& d;
     d = mcell (a !& c);
     q = d;

END;
```

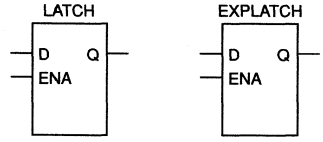
AHDL Equivalent of NOR–NOR

```
SUBDESIGN nornor
(
     s, r                : INPUT;
     q                   : OUTPUT;
)

VARIABLE

     a, b, c, d          : NODE;

BEGIN

     a = r;
     b = s;
     c = b !# d;
     d = mcell (a !# c);
     q = d;

END;
```

Transparent D Latches

MAX+PLUS II software provides both a LATCH primitive and an EXPLATCH macrofunction to implement transparent D latches (see Figure 3).

Figure 3. LATCH Primitive and EXPLATCH Macrofunction



The LATCH primitive is directly supported in MAX 5000 EPLDs. Each macrocell contains a dynamic register that can be configured as a transparent D latch. For Classic, MAX 7000, and STG EPLDs, the logic necessary for a latch is created by the MAX+PLUS II Logic Synthesizer. The EXPLATCH macrofunction, built from cross-coupled expanders, is supported by MAX 5000, MAX 7000, and STG EPLDs.

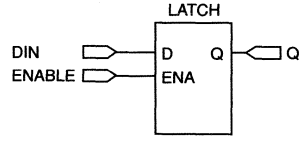
LATCH and EXPLATCH both operate in the same way: when a high signal is applied to the ENA input, the latch passes data from D to Q. When ENA is low, the state of Q is maintained, regardless of the state of the D input.

You can use these latches in AHDL files and in logic schematics. In AHDL Text Design Files (.TDF) they are implemented with variable declarations or in in-line references. See Figure 4.

Figure 4. LATCH and EXPLATCH Implemented in AHDL & Schematics

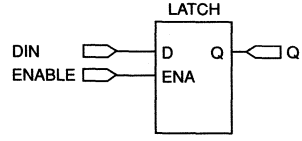
Latch Implemented with Variable Declaration

```
SUBDESIGN dlat1
(
    din, enable         :INPUT;
    q                   :OUTPUT;
)

VARIABLE
    fester              :LATCH;
BEGIN
    fester.d   = din;
    fester.ena = enable;
    q      = fester.q;
END;
```



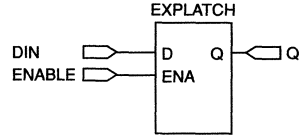
Latch Implemented with In-Line Reference

```
SUBDESIGN dlat2
(
    din, enable         :INPUT;
    q                   :OUTPUT;
)

BEGIN
    q = latch (din, enable);
END;
```



EXPLATCH

```
INCLUDE "explatch.inc";

SUBDESIGN explat
(
    din, enable         :INPUT;
    q                   :OUTPUT;
)

BEGIN
    q = explatch (din, enable);
END;
```



The AHDL file containing the EXPLATCH must also contain an Include Statement, because EXPLATCH is a macrofunction, not a primitive.

Conclusion

Asynchronous latches are simple to implement in Altera Classic, MAX 5000, MAX 7000, and STG EPLDs with the primitives and macrofunctions provided with MAX+PLUS II development software. You can implement SR latches either with expanders or with macrocells. Transparent D latches are implemented with the built-in latch capability of the MAX 5000 register, or with expanders or macrocells.

EP1810 Classic EPLD as a Bar Code Decoder

April 1992, ver. 3

Application Brief 27

Introduction

This application brief describes how to implement a bar code decoder in an Altera EP1810 Classic EPLD with MAX+PLUS II development software. The EP1810 EPLD decodes a generic bar code, stores the decoded data byte, and alerts a microprocessor that data is ready. This application brief covers the following subjects:

- Bar code description
- Bar code decoder design
- Design processing
- Design verification

The completed bar code decoder design (a design is called a "project" in MAX+PLUS II) is created with two design entry methods: hierarchical schematic capture (including TTL macrofunctions) for defining specific functions and Altera Hardware Design Language (AHDL) for specifying an internal controller.

What Is a Bar Code?

A bar code represents binary data or program information in a graphical format. With a reader (a light pen or other scanning device) and the requisite bar code decoder, you can read information quickly and accurately. This application brief assumes that you are using a light pen. For applications where it is particularly important that a first-time read be successful, a bar code system yields better results than optical character recognition (OCR). Bar codes are also suitable for applications that cannot use a magnetic strip or other media to represent information. Both flexibility and low cost make bar code systems extremely popular.

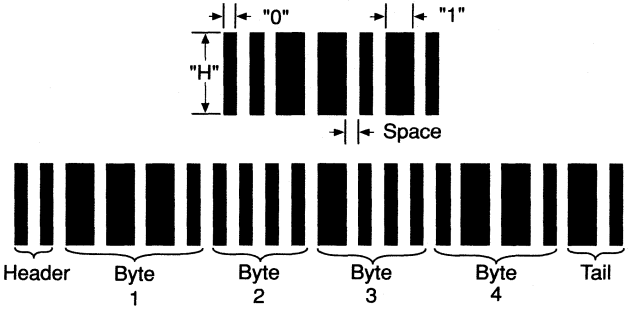
Physical Specifications

Although bar code applications have many variations, they share enough common features to illustrate a "generic" case. See Figure 1. The following features are common to all bar code applications.

- All bar codes contain 0, 1, and space characters.
- The 0 and space characters always have the same width; the 1 is twice that width.
- All bar codes have a header and a tail.
- All data follows the header and ends with the tail.
- All bar codes have maximum limits on the number of data bytes.

Figure 1. Sample Bar Code

The generic bar code described here has a header consisting of a 00 sequence, followed by a checksum byte. The tail is a 10 sequence.



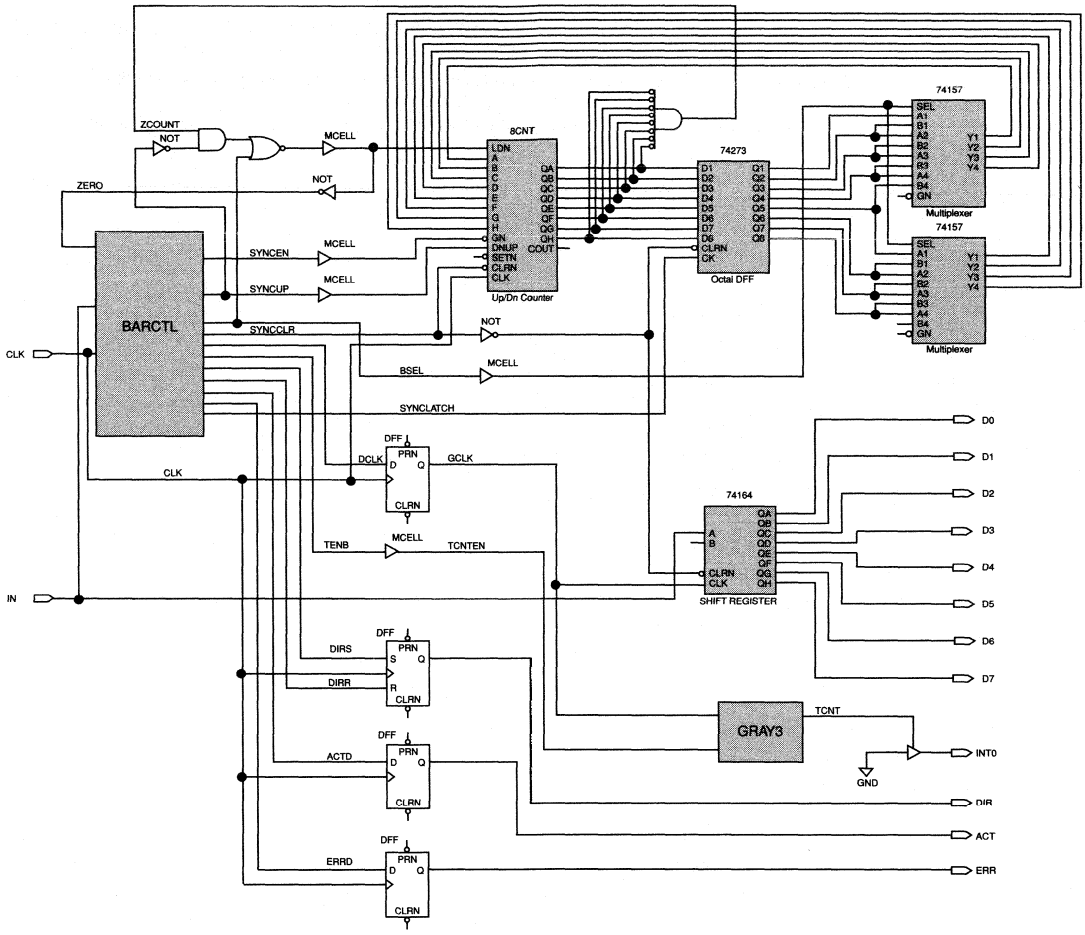
To ensure accurate bar code detection, the light pen must scan the bar code at relatively constant speed. Therefore, bar codes should be fairly short, since it is easier to move the pen at constant speed over a short distance than over a long one. Before reading the bar code, the light pen output is low, indicating that the light pen is inactive (i.e., in a white region). As the light pen reaches the black region of the header, its output goes high. If the light pen's speed varies too much, the data is read incorrectly. However, the system can easily detect errors of this type by comparing checksums.

Bar Code Decoder Design

Figure 2 shows a MAX+PLUS II Graphic Design File (.GDF) of the bar code decoder (BARDECOD.GDF). The synchronous counter is implemented by one 74273 and two 74157 macrofunctions from the MAX+PLUS II TTL macrofunction library and one custom function (8CNT), which is available from Altera Applications. Terminal-count circuitry consists of logic and an MCELL primitive. The byte counter, GRAY3, is implemented in a separate AHDL Text Design File (.TDF) described later in the application brief. The shift register is implemented by combining a 74164 macrofunction with eight output pins. The status information module consists of a tri-state buffer (TRI) and output pin configured to mimic an open collector output, and status flags implemented by SRFF and DFF primitives.

The bar code data is fed into the EP1810 EPLD through the IN input. The synchronous counter uses the data to determine the input data read rate, which the state machine uses to decode incoming data. The decoded data is then stored in a shift register. Once eight bits of data have been shifted into the shift register, an open collector interrupt to the microprocessor (INT0) goes low. Various status outputs (DIR, ACT, and ERR) indicate the current state of the bar code decoder.

Figure 2. Bar Code Decoder (BARDECOD.GDF)



The bar code decoder consists of five parts:

- Synchronous counter
- Byte counter
- Shift register
- Status outputs
- Controller state machine

Synchronous Counter

The bar code decoder uses the bar code header's leading single-width black region to measure the width of a 0. When the light pen passes over the first black region, the synchronous counter begins counting, and stops only when the light pen encounters the beginning of the adjacent white region. The count value thus reflects the amount of time required to read the width of a space or 0 bar. Twice the count value is required to read a 1 bar.

The count value is an initial sampling of the bar code data. Since the light pen's scanning speed varies slightly, the count value is only an approximate measure of the width. Therefore, you should sample data in the center of the black and white regions rather than on the edges. The first sampling occurs in the middle of the space following the leading black region, when the synchronous counter reaches half of its total value. Correct sampling timing is achieved by bit-shifting the latched synchronous counter with a pair of 74157 multiplexers. The counter is reset, and the next sampling occurs when it reaches the full count value.

The values of data samples correspond to the bar-encoded data. If a black-white region is read, the light pen has passed over a single-width black region followed by a single-width white region, indicating a 0. If a black-black-white region is read, the light pen has passed over two adjacent black regions followed by a white region, indicating a 1. If any other combinations starting with black are read, or if two adjacent white regions are read, the code is invalid.

Byte Counter

The byte counter counts the number of bits shifted in. When a complete byte has been read, the byte counter signals the microprocessor. The byte counter is implemented with a Gray-code sequence, in which only one bit changes between any two count transitions, ensuring that the outputs are glitch-free and preventing spurious outputs from inadvertently interrupting the microprocessor. The byte counter has an open-collector output to the microprocessor (INT0) that is created by connecting the input of a tri-state driver to ground and selectively enabling the tri-state (TCNT). Figure 3 shows the byte counter implemented in the AHDL file GRAY3.TDF. The GRAY3 symbol for the TDF is incorporated into BARDECOD.GDF, the top-level bar code decoder schematic.

Shift Register

Input data is stored in a 74164 shift register. The shift register Clock is fed by the controller state machine to ensure that data is latched at the appropriate time. The 74164 outputs must be buffered and connected to

Figure 3. Byte Counter Text Design File (GRAY3.TDF)

This eight-stage Gray-code counter uses the high-level AHDL format.

```
TITLE  "GRAY3 3-Bit Gray Counter";

SUBDESIGN gray3
(
    clock, tcnten     : INPUT;
    tcnt              : OUTPUT;
)
VARIABLE
    gray : MACHINE OF BITS ( q[2..0] )
           WITH STATES (
                          s0 = b"000",
                          s1 = b"001",
                          s2 = b"011",
                          s3 = b"010",
                          s4 = b"110",
                          s5 = b"111",
                          s6 = b"101",
                          s7 = b"100" );
BEGIN
    gray.clk = clock;
    tcnt = tcnten & s0;

    IF tcnten THEN
        CASE (gray) IS
            WHEN s0 => gray = s1;
            WHEN s1 => gray = s2;
            WHEN s2 => gray = s3;
            WHEN s3 => gray = s4;
            WHEN s4 => gray = s5;
            WHEN s5 => gray = s6;
            WHEN s6 => gray = s7;
            WHEN s7 => gray = s0;
        END CASE;
    END IF;
END;
```

the microprocessor bus. The shift register is implemented by a 74164 macrofunction and OUTPUT primitives.

Status Outputs

Special outputs allow external devices to determine the bar code decoder status. The status information consists of the open collector INTO output and the SRFF and DFF primitive status flags (DIR, ACT, and ERR).

A direction signal, DIR, permits backwards scanning of a bar code. If inactive, the DIR signal indicates that the tail was read before the header. In this instance, all data and checksum words are reversed, and the microprocessor makes the compensating transformations.

The ACT signal indicates that a bar code is being scanned in. This signal is useful for a variety of error handling or initialization tasks. For example, a microprocessor can poll this signal and enter special routines dedicated to bar code reading.

The ERR signal indicates that the decoder encountered an illegal sequence of white and black regions, e.g., the bar code may have been unreadable or the scan rate may have not been uniform enough. The microprocessor can use ERR to dump illegal reads without first computing and comparing a checksum against the checksum byte passed to the microprocessor by the bar code.

Controller State Machine

The bar code decoder must determine if a header is valid and then convert the white and black bar code regions to machine-readable data. It also coordinates activities of the synchronous counter, shift register, byte counter, and status-generation circuitry. This coordination is best achieved with a centralized state controller state machine, BARCTL, shown in Figure 4.

The following control states are used:

IDLE The machine idles until the light pen reads a black region; then the synchronous counter is started.

SYNC The synchronous counter counts while in this state. The machine stays in this state until the light pen reads the first white region. The count corresponds to the width of the first black bar.

LATCH The machine latches the count value into a holding register. For the next count cycle, the control line BSEL shifts the count value to the right by one bit. Subsequently, the synchronous count expires on every modulus of the full latched count, and triggers reading of the input stream.

HDR1, HDR2, FWD, REV, ERR The machine begins reading the rest of the header bits. Two sequences are possible, depending on whether a 00 (forward) sequence or a 01 (backward) sequence is read. Any improper reads cause the state machine to go to the ERR state. Direction status is latched, depending on state FWD or REV. Once the header information is read, data and checksum bytes are read into the shift register.

ACT1, ACT2, ACT3, ACT4 These four states are the active reading states. The ACT1-ACT3 loop indicates that a 0 was read. The ACT1-ACT2-ACT4 loop corresponds to reading of a 1. Reading stops when a long white space is read, indicating the end of the bar code.

Figure 4. Bar Code Controller State Diagram

Bar code decoding is coordinated by the state machine BARCTL.

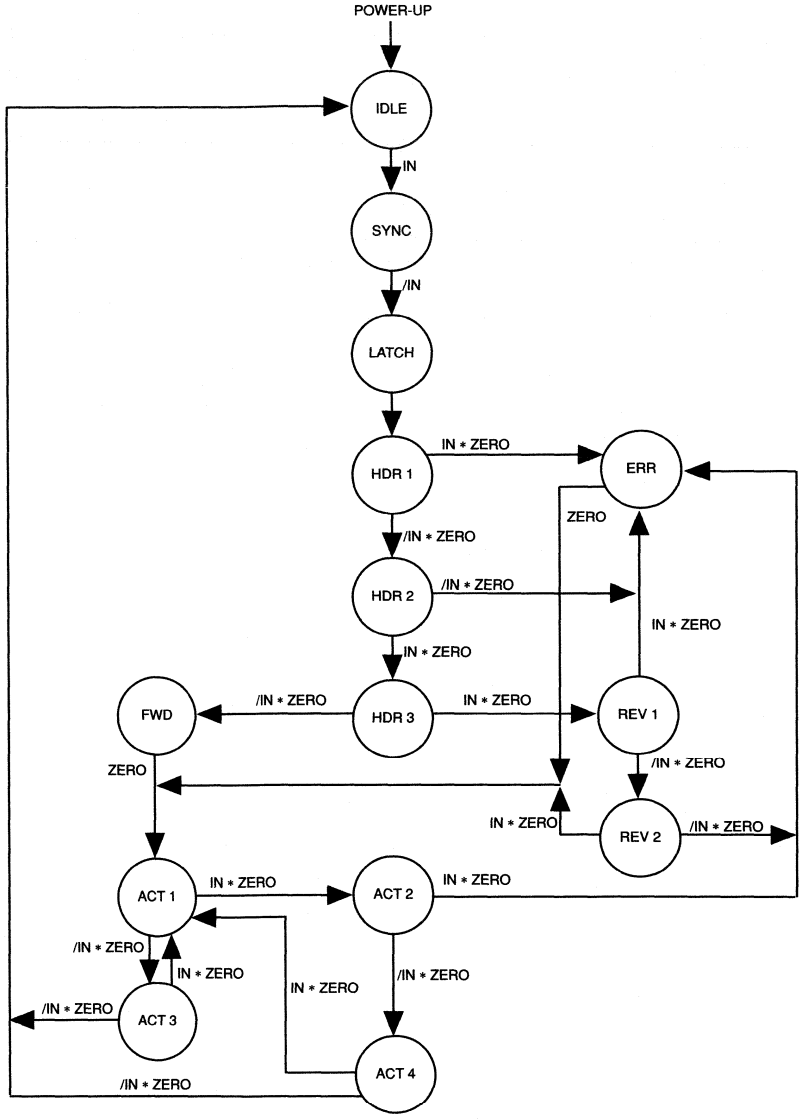


Figure 5 shows the controller implemented in a TDF (BARCTL.TDF).

Figure 5. Bar Code Control (BARCTL.TDF) (Part 1 of 2)

```
TITLE "Bar Code Controller";

SUBDESIGN barctl
(
    clock, zero, in                 : INPUT;

    syncen, syncup, syncclr         : OUTPUT;
    bsel,dclk, tenb, dirs           : OUTPUT;
    dirr, actd, errd                : OUTPUT;
    synclatch                       : OUTPUT;
)

VARIABLE
    barctl : MACHINE OF BITS ( q[3..0] )
             WITH STATES (
                          idle, sync, latch,
                          hdr1, hdr2, hdr3,
                          fwd, rev1, rev2,
                          act1, act2, act3, act4,
                           err ) ;
BEGIN

    barctl.clk   = clock;
    dirs         = fwd;
    dirr         = idle;
    actd         = !idle;
    errd         = err & !idle;
    syncclr      = idle & !in;
    syncup       = sync;
    syncen       = idle;
    synclatch    = latch;
    dclk         = act2 # act3;
    tenb         = act2 # act3 # act4;
    bsel         = latch;

    CASE barctl IS
        WHEN idle =>
            IF in THEN
                barctl = sync;
            END IF;
        WHEN sync =>
            IF !in THEN
                barctl = latch;
            END IF;
        WHEN latch =>
                barctl = hdr1;
        WHEN hdr1 =>
            IF !in & zero THEN
                barctl = hdr2;
            ELSIF zero THEN
                barctl = err;
            END IF;
        WHEN hdr2 =>
            IF in & zero THEN
                barctl = hdr3;
            ELSIF zero THEN
                barctl = err;
            END IF;
```

Figure 5. Bar Code Control (BARCTL.TDF) (Part 2 of 2)

```
        WHEN hdr3 =>
            IF in & zero THEN
                barctl = rev1;
            ELSIF zero THEN
                barctl = fwd;
            END IF;
        WHEN fwd =>
            IF zero THEN
                barctl = act1;
            END IF;
        WHEN rev1 =>
            IF in & zero THEN
                barctl = err;
            ELSIF zero THEN
                barctl = rev2;
            END IF;
        WHEN rev2 =>
            IF in & zero THEN
                barctl = act1;
            ELSIF zero THEN
                barctl = err;
            END IF;
        WHEN act1 =>
            IF in & zero THEN
                barctl = act2;
            ELSIF zero THEN
                barctl = act3;
            END IF;
        WHEN act2 =>
            IF in & zero THEN
                barctl = err;
            ELSIF zero THEN
                barctl = act4;
            END IF;
        WHEN act3 =>
            IF in & zero THEN
                barctl = act1;
            ELSIF zero THEN
                barctl = idle;
            END IF;
        WHEN act4 =>
            IF in & zero THEN
                barctl = act1;
            ELSIF zero THEN
                barctl = err;
            END IF;
        WHEN err =>
            IF zero THEN
                barctl = act1;
            END IF;
    END CASE;
END;
```

When you compile a TDF, MAX+PLUS II automatically generates a corresponding symbol that can be incorporated into a GDF. TDF symbols, TTL macrofunctions, and primitives are assembled in the MAX+PLUS II Graphic Editor to create the hierarchical project shown in Figure 2.

Design Processing

After generating GDFs and TDFs in the bar code controller project, you use the MAX+PLUS II Compiler to process the project and generate a Programmer Object File (.POF) or JEDEC File (.JED) for programming the EPLD. The Report File (.RPT) generated by MAX+PLUS II indicates that this design requires the following EP1810 resources: 43 of the 48 macrocells, 2 of the 16 inputs, and only 34% of the available logic.

Design Verification

The MAX+PLUS II Simulator allows you to test your design before programming the EPLD. In this example, a serial input stream corresponding to a valid bit stream is read by the design. It properly sequences through the states, latches the data, and interrupts a processor. Figure 6 shows a bar code that can be used as a test case for this design.

Figure 6. Simulation Data

Simulation is driven by input data that emulates this sequence. Logical behavior is monitored in the MAX+PLUS II Simulator.

You can obtain design files for the bar code reader from Altera's electronic bulletin board service (BBS) or by calling Altera Applications at (800) 800-EPLD.

You can verify the controller state machine by comparing the outputs (Q0 to Q3) to the state table values in Figure 5.

Conclusion

As the bar code decoder example shows, the speed and density of the EP1810 EPLD enable it to implement complex functions, significantly reducing IC count and system power requirements and minimizing system size and cost. The combination of the EP1810 EPLD and the MAX+PLUS II software allows you to enter and verify your projects quickly, and ultimately increases product reliability.

Estimating a Design Fit

April 1992, ver. 3 **Application Brief 60**

Introduction

This application brief explains how to estimate whether a design will fit into an Altera Classic, MAX 5000, MAX 7000, or STG EPLD. When estimating a design fit, you must consider both the timing and the physical fit:

- The *timing fit* analysis determines whether the target EPLD can meet the timing requirements of the design (called a "project" in MAX+PLUS II). Before you can choose the appropriate EPLD(s), you must be familiar with the basic Clock frequency and critical timing paths. Critical timing paths are determined by the total propagation delay (t_{PD}), maximum Clock frequency (f_{CNT}), setup time (t_{SU}), and Clock-to-output delay (t_{CO1}). For more information on timing delays, see *Application Brief 100 (Understanding EPLD Timing)* in this handbook.

- The *physical fit* analysis determines whether the device provides the resources necessary to fit the design's pin and macrocell requirements. This application brief describes how to estimate a physical fit.

If the design has been entered in MAX+PLUS II, you can accurately and easily estimate a fit by using automatic device selection and compiling the design. The MAX+PLUS II Compiler automatically chooses EPLD(s) that best fit the design. This application brief provides guidelines for estimating a fit before entering a design. Familiarity with EPLD architectures and characteristics is assumed. Refer to individual EPLD data sheets for more information.

Pin Count

Each device has a fixed number of dedicated input and I/O pins. Dedicated input pins serve as inputs only, while I/O pins can be configured for input, output, or bidirectional operation. Since output and bidirectional signals can pass only through I/O pins, the number of output and bidirectional signals in the design must be less than or equal to the number of I/O pins available in the target EPLD(s):

(output signals + bidirectional signals in design) I/O pins in EPLD

In addition, the number of input, output, and bidirectional signals in the design must be less than or equal to the total number of dedicated input and I/O pins in the target EPLD(s):

(input signals + output signals + bidirectional signals in design) (dedicated inputs + I/O pins in EPLD)

If both conditions are met, the target EPLD(s) will satisfy the pin-count requirements of the design.

Macrocell Count

A macrocell is the basic building block used to implement logic. Since each device has a fixed number of macrocells, the number of macrocells required for a design must be smaller than the number of macrocells available in the target EPLD. The number of macrocells required for a design varies for different Altera EPLD families:

- *Classic EPLDs*: The number of macrocells required is the sum of the design output signals, input signals brought in through I/O pins, and buried registers used by TTL macrofunctions and other flipflops:

  (outputs + I/O pin inputs + buried registers in design) < macrocells in EPLD

- *MAX 5000, MAX 7000,* and *STG EPLDs*: The number of macrocells required is the sum of outputs and buried registers used by TTL macrofunctions and other flipflops:

  (outputs + buried registers in design) < macrocells in EPLD

If the estimated number of macrocells is less than 80% of the target EPLD's total number of macrocells, the design will probably fit into the EPLD; if the estimate is greater than 90%, the design may require a larger EPLD.

A typical design includes TTL macrofunctions, flipflops, and basic combinatorial gates (e.g., `NAND`, `OR`, `XOR`). Since basic gates and the combinatorial logic in TTL macrofunctions are implemented in the logic array, they do not require an entire macrocell, and can be excluded from the macrocell estimate. Registered functions, such as shift registers and counters, use one macrocell per bit. For example, a 4-bit counter uses four registers to implement the counter logic.

The "MacroMunching" feature of the MAX+PLUS II Compiler allows you to use macrofunctions without losing resource efficiency. For example, if your design uses only six of the eight outputs of a `74374` octal register macrofunction, the Compiler's Logic Synthesizer module automatically removes the logic associated with the two unused outputs. In this case, the `74374` macrofunction uses six macrocells rather than eight.

Conclusion

To estimate a design fit, you must evaluate the timing functionality and physical resources of the target EPLD. To estimate a timing fit, you must have a basic understanding of the EPLD architecture and AC timing parameters. To estimate a physical fit, you must count the pins and macrocells used in the design. If the number of pins and macrocells in a design is smaller than the pin and macrocell count of the target EPLD, the design should fit into the EPLD.

Multiway Branching with the EPS448 SAM EPLD

April 1992, ver. 3

Application Brief 63

Introduction

The EPS448 Stand-Alone Microsequencer (SAM) EPLD can perform a four-way branch in a single Clock cycle. For designs that require more than four-way branching, various techniques are available. This application brief describes two processes:

- How to perform multiway branching
- How to access more than four product terms per branch

Familiarity with the EPS448 architecture and the SAM Assembly Language (ASM) or Altera State Machine Input Language (ASMILE) is assumed.

Beyond Four-Way Branching

The EPS448 EPLD performs multiway branching in three different ways:

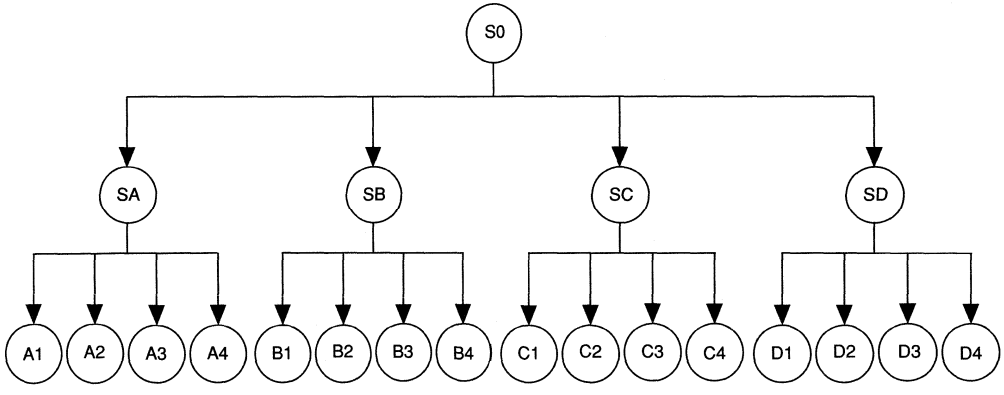
- Linked branching
- Dispatch routine
- Counter-conditioned branching

Linked Branching

Linked branching (see Figure 1) uses 2 Clock cycles to perform a 16-way branch. On the first Clock cycle, the machine completes a 4-way branch to

Figure 1. Linked Branching

Intermediate states allow a 16-way branch in 2 Clock cycles. In the first Clock cycle, the machine branches to 1 of 4 intermediate states (SA to SD). On the second Clock edge, the machine performs another 4-way branch to 1 of 16 states (A1 to A4, B1 to B4, C1 to C4, D1 to D4).



4 intermediate states; on the second Clock cycle, it finishes with a 4-way branch out of each of these intermediate states. You can use either ASMILE or ASM for linked branching.

In applications running at low frequency, you may be able to double the EPS448 Clock frequency, use linked branching, and give the appearance of a multiway branch in a single system Clock.

Dispatch Routine

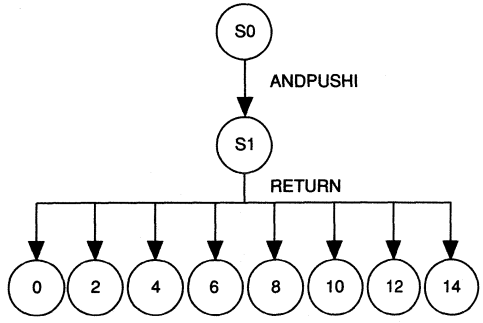
The dispatch routine can perform any number of branches. Inputs are pushed onto the stack in the first Clock cycle and are used as the next-state address in the second Clock cycle. The ASM commands `PUSHI` or `ANDPUSHI` push the inputs onto the stack, and the `RETURN` command causes a branch to that address.

In the sample code shown in Figure 2, `ANDPUSHI` pushes the inputs onto the stack after first masking them with the binary number `00001110B`, where input `I7` is the most significant bit and input `I0` is the least significant bit. Consequently, an even decimal number between 0 and 14 is pushed onto the stack, matching the binary number for `I3 I2 I1 I0`.

The `RETURN` command causes the top-of-stack to become the next address. Since this value is an even decimal number between 0 and 14, an 8-way branch to one of these addresses is performed. To complete the branch, the 8 even memory addresses between 0 and 14 must contain the 8 potential next states. By using an absolute label instead of a relative label, you can place instructions in a particular memory location. Absolute labels must represent a legal and unique memory location within the EPS448 EPLD.

Figure 2. Dispatch Routine

The dispatch routine performs an 8-way branch in 2 Clock cycles. On the first Clock edge, the number represented by the top-of-stack is used as the next address. Since I3, I2, and I1 can be used to represent an even number between 0 and 14, each of these addresses is a potential next state.




```
S0:   [STATE0]  ANDPUSHI  00001110B;
      % Push I3, I2, and I1 onto the stack %

S1:   [STATE1]  RETURN;
      % Jump to the top of stack %

0D:   [OUT0]  JUMP NEXT0;
2D:   [OUT2]  JUMP NEXT2;
4D:   [OUT4]  JUMP NEXT4;
6D:   [OUT6]  JUMP NEXT6;
8D:   [OUT8]  JUMP NEXT8;
10D:  [OUT10] JUMP NEXT10;
12D:  [OUT12] JUMP NEXT12;
14D:  [OUT14] JUMP NEXT14;
```

Addresses and numbers specified with ASMILE or ASM must start with a digit and end with H (hexadecimal), D (decimal), or B (binary). For example, the instruction 4D: [OUT4] JUMP NEXT4 in Figure 2 is placed in address 4D of the microcode. It is executed if /I3*I2*/I1 is true at the end of the ANDPUSHI command, because input I2 (high) translates to 4D.

The following list shows examples of absolute, relative, and illegal labels:

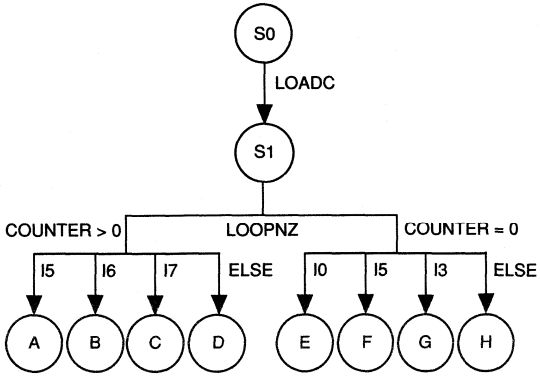
Absolute labels: 10001B, 0F2H, 44D
Relative labels: F2H, 44D
Illegal labels: 44, 0F2D, 10001H

Counter-Conditioned Branching

Counter-conditioned branching uses the counter as a flag and performs a two-way branch with the LOOPNZ command, based on the value of the counter. The two branched addresses can be multiway-branch addresses that permit two additional four-way branches. This branching scheme creates an eight-way branch in a single Clock cycle, as shown in Figure 3. The code in Figure 3 shows an example of the actual ASM syntax required. The LOADC command at the S0 label sets the counter to 1 or 0, based on the current value of the I1 input. The next instruction, at label S1, performs the branch with LOOPNZ. Based on the value in the counter, the next state will come from label ABCD or from label EFGH. Since both of these labels are in the multiway-branch block, the actual next instruction depends on the input values. If the counter is set to 1 and input condition /I0*I5 is true, then the next command will be JUMP NEXTF.

Figure 3. Counter-Conditioned Branching

An 8-way branch can be performed in a single Clock cycle if the counter has been set as a flag. In the code shown here, LOADC sets the counter flag based on the input condition I1. LOOPNZ performs a 2-way branch based on the flag to label ABCD or EFGH, both of which are 4-way branch locations.




```
S0:   IF I1 THEN [OUTS0] LOADC 1D;
      ELSE [OUTS0] LOADC 0D;

S1:   LOOPNZ ABCD ONZERO EFGH;

ABCD: IF I5 THEN [OUTA] JUMP NEXTA;
      ELSEIF I6 THEN [OUTB] JUMP NEXTB;
      ELSEIF I7 THEN [OUTC] JUMP NEXTC;
      ELSE [OUTD] JUMP NEXTD;

EFGH: IF I0 THEN [OUTE] JUMP NEXTE;
      ELSEIF I5 THEN [OUTF] JUMP NEXTF;
      ELSEIF I3 THEN [OUTG] JUMP NEXTG;
      ELSE [OUTH] JUMP NEXTH;
```

More Than Four Product Terms per Condition

A design that runs out of product terms for a branch condition, or generates the SAM+PLUS Design Processor (SDP) error message `Predicate too long`, can still be fitted after some adjustments. For example, the first branch condition in the following ASMILE code contains five product terms, while the second condition contains only one:

```
START:  IF   I5*I0' +
             I5*I1' +
             I5*I2' +
             I5*I3' +
             I5*I4'              THEN S1
        IF   I2*I1*I2*I3*I4      THEN S0
        S2
```

You can make a tradeoff between the number of branches and the number of product terms. The SDP automatically partitions the first branch, so that there are actually two branches to `S1`—one branch with four product terms and another with one product term:

```
START:  IF   I5*I0' +
             I5*I1' +
             I5*I2' +
             I5*I3'              THEN S1
        IF   I5*I4'              THEN S1
        IF   I2*I1*I2*I3*I4      THEN S0
        S2
```

This approach reduces the number of possible branches to three. In cases where all four branches are needed, reordering the branches may reduce the number of product terms. Also, reordering may make better use of the built-in prioritization of the EPS448 architecture, since the second branch condition can assume that the first condition has failed. In the previous example, the first condition can be factored into `I5*/(I0*I1*I2*I3*I4)`, and the branch order can be rearranged so that the `S0` branch is considered first. Then, the `S1` branch condition can assume that the expression `I0*I1*I2*I3*I4` has failed and need not be tested. The following example shows the previous code reordered, with one product term per condition (a fourth branch can easily be added):

```
START:  IF   I0*I1*I2*I3*I4      THEN S0
        IF   I5                  THEN S1
        S2
```

Conclusion

The flexibility of the EPS448 architecture allows you to perform multiway branching in a number of ways. If more than four product terms exist per condition, you can reduce the number of terms through partitioning and reordering.

Vertical Cascading of EPS448 SAM EPLDs

April 1992, ver. 3

Application Brief 65

Introduction

When an application requires more states or more microcode depth than a single EPS448 Stand-Alone Microsequencer (SAM) EPLD can provide, i.e., more than 448 states or words, multiple SAM EPLDs can be cascaded vertically to accommodate the additional states. This application brief describes how to vertically cascade EPS448 EPLDs by passing control from one EPLD to another. The best method for a specific application depends on how the sequencer is most easily partitioned. The following topics are discussed:

- ❑ Simple vertical cascading
- ❑ Addressed-branch cascading
- ❑ Vertical subroutine calls
- ❑ Master/slave cascading
- ❑ Design tips on cascading EPS448 EPLDs

Familiarity with the EPS448 EPLD and the SAM Assembly Language (ASM) is assumed.

Simple Vertical Cascading

When EPS448 EPLDs are vertically cascaded, the outputs of all the devices are tied together, forming a tri-state output bus (see Figure 1). One EPLD at a time controls the output bus; all others are disabled by tri-state buffers. The controlling EPLD performs sequencing until it encounters a sequence found in another EPS448 EPLD. It then tri-states its outputs, and the other EPS448 EPLD takes control of the bus.

The simplest method of vertically cascading EPS448 EPLDs is to use one input pin to signal to a device that it must take control of the output bus (see Figure 1). A control line to each EPS448 device (e.g., `nGO1`) is pulled up through a 1-k resistor. When the active EPS448 EPLD passes control to another EPS448 device, it pulls down the appropriate `nGO` signal and jumps to an idle state, where it tri-states its outputs. This configuration allows conditional `JUMP` instructions between EPS448 EPLDs.

Figure 1. Vertical Cascading

When EPS448 EPLDs are vertically cascaded, the outputs are tied together. While one EPS448 has control of the output bus, the others are tri-state-disabled. Each EPS448 EPLD has one input (e.g., nG01) that enables it onto the bus.

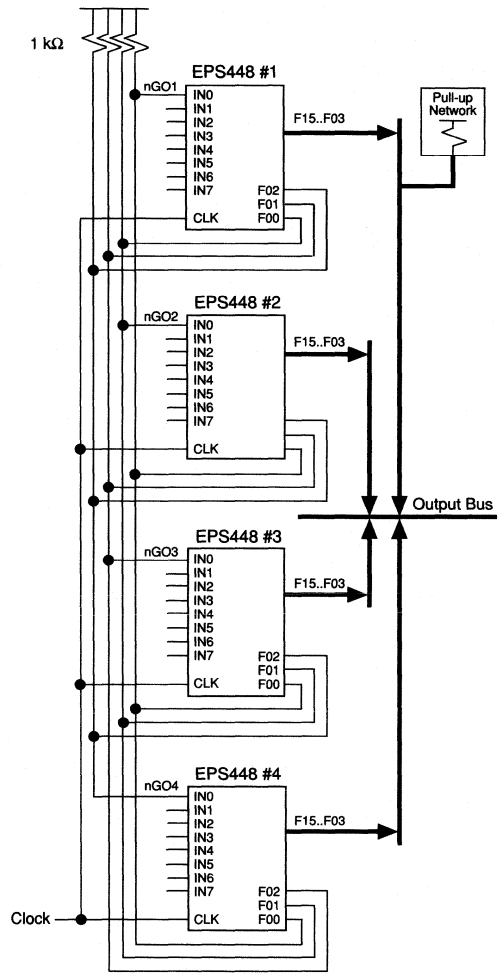
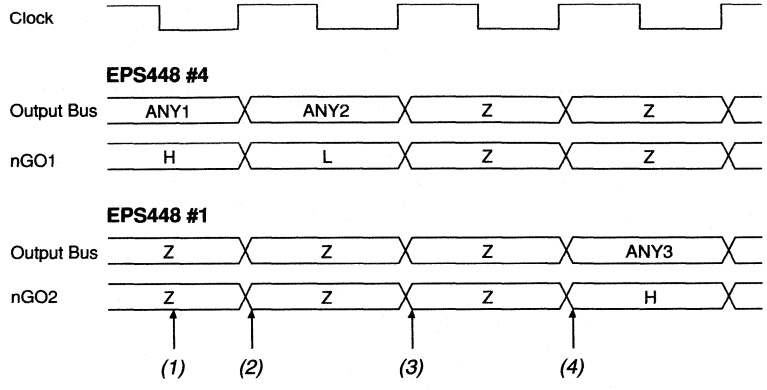


Figure 2 shows the timing associated with this simple vertical cascading approach. Initially, EPS448 #4 has control of the output bus and EPS448 #1 is idle. EPS448 #1 outputs are tri-stated during this period. To pass control, EPS448 #4 brings the nGO1 signal low to activate EPS448 #1. On the next Clock cycle—called the transition Clock period—the outputs of EPS448 #1 and EPS448 #4 are both tri-stated. EPS448 #1 becomes active and takes control of the output bus. EPS448 #4 becomes idle on the following Clock.

During the transition Clock period, the tri-state is disabled in both EPLDs. This idle period prevents potential glitches (i.e., bus contention with the other EPLDs) during transitions from high-impedance to a valid output, or

Figure 2. Control Transfer Function

To pass control between two EPLDs (EPS448 #4 and EPS448 #1), EPS448 #4 starts active (1), then brings nG01 low (2) to pass control. On the next Clock period, both EPLDs are disabled (3) before EPS448 #1 takes control of the output bus (4).

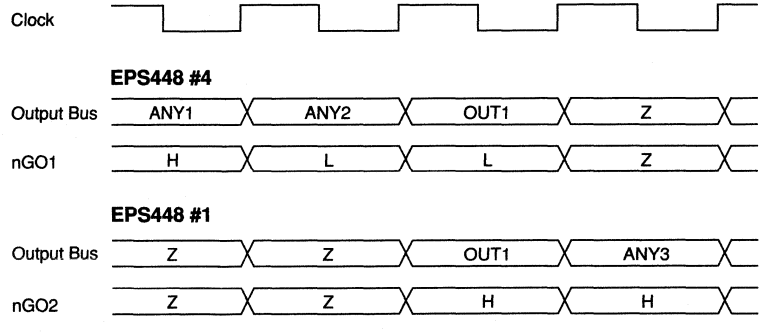


vice versa. During this Clock period, the control lines (nGO1 to nGO4) float unless they are pulled up by the resistors.

Figure 3 illustrates another method in which both EPS448 EPLDs drive the same value (OUT1) on the output bus for the transition Clock period. This method removes the need for pull-up resistors, but it may introduce temporary bus contention, possibly causing a current surge into the EPS448 EPLDs and inducing noise or ground bounce elsewhere on the board. However, bus contention does not impair EPLD operation.

Figure 3. Alternative Control Transfer Function

To avoid the Clock cycle where the output bus is left floating, both EPLDs drive the same value onto the bus for a single Clock cycle.



Each of the EPS448 EPLDs must have its own design file (with the extension .ASM or .SMF). A segment of the code used in EPS448 #1 is shown in Figure 4. While the machine is inactive, it remains in IDLE with the outputs disabled. When it receives an active-low nGO1 signal from pin 1NO, it jumps to the label START, takes control of the output bus, and holds nGO2 to nGO3 high, preventing the other EPLDs from competing for control of the output bus.

Figure 4. ASM Code for Vertical Cascading

This ASM syntax is for EPS448 #1. This EPLD sits in the IDLE state until the nGO1 signal from pin IN0 goes low. It then jumps to START and takes control of the output bus. When EPS448 #1 is ready to give up control of the outputs, it jumps to QUIT and brings one of the three nGO lines low to activate the next machine. Finally, it jumps back to the IDLE state.

```
MACROS:  % F F F F F F  .......   F %
         % 0 1 2 3 4 5  .......  15 %
  HOLD = " 1 1 1 "
  GO2  = " 0 1 1 "
  GO3  = " 1 0 1 "
  GO4  = " 1 1 0 "
  OUT1 = " 1 0 1 ........... 0 "    % Insert desired   %
  ANY3 = " 1 0 0 ........... 1 "    % output values    %
  ANY2 = " 0 0 1 ........... 1 "    % for these states %

PROGRAM:
IDLE:   IF /IN0 THEN [Z] JUMP START;
        ELSE [Z] JUMP IDLE;
START:  [HOLD ANY3] CONTINUE;
        •
        •
        •
QUIT:   IF IN3 * IN4 THEN [GO2 ANY2] JUMP IDLE;
        ELSEIF    IN3 THEN [GO3 ANY2] JUMP IDLE;
        ELSE               [GO4 ANY2] JUMP IDLE;
```

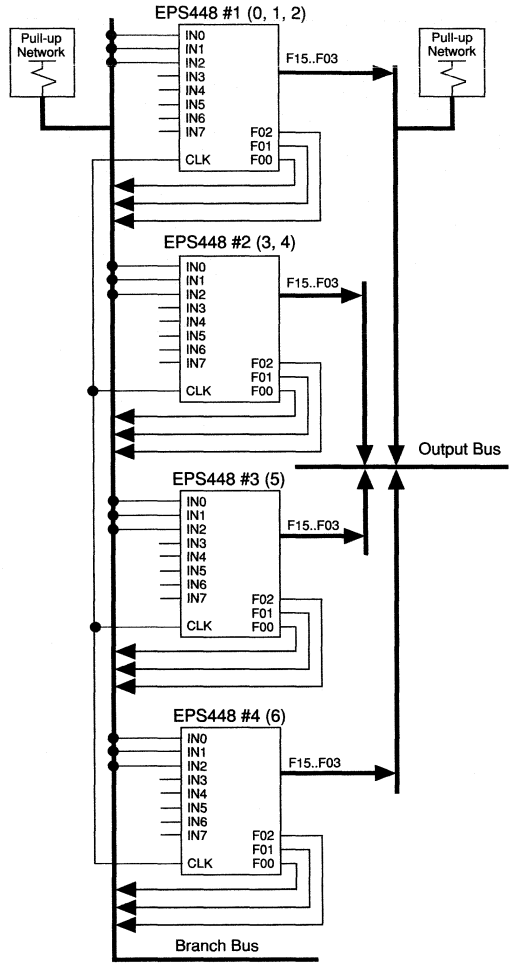
When EPS448 #1 is ready to pass control to another EPS448 EPLD, it jumps to QUIT and pulls the appropriate nGO control line low. For example, when GO2 is the output, the EPS448 #1 F00 output goes low, bringing the signal nGO2 low and activating EPS448 #2 on the next rising edge of the Clock. In this example, QUIT is the label of a multiway branch instruction, so any one of the other three EPS448 EPLDs can be activated next. After activating the next machine, EPS448 #1 jumps to IDLE and waits to receive control again.

Addressed-Branch Cascading

As your design requires more EPS448 EPLDs, you may find an addressed branching configuration more convenient (see Figure 5). A branch bus lies alongside the output bus. Each EPS448 EPLD examines the branch bus and stays idle until an address assigned to it appears on the bus. The EPS448 then takes control of both buses. Since each EPLD has access to the branch bus, each EPLD can call any one of the other EPLDs.

Figure 5. Addressed-Branch Cascading

In addressed-branch cascading, each EPS448 EPLD becomes active when it sees one of its assigned addresses on the branch bus. Each EPLD can recognize more than one address. For example, EPS448 #1 takes control if it sees addresses 0, 1, or 2 on the branch bus.



A single EPS448 EPLD can have several addresses assigned to it, with each address corresponding to a different starting location in its microcode. Thus, a single EPLD can contain more than one block of independent states. In such a case, each address on the branch bus corresponds to a block of states within an EPLD.

The branch bus in Figure 5 consists of three output bits (F00, F01, and F02) that are passively pulled up by 1-k resistors and controlled by the active EPS448 EPLD. Values on the branch bus serve as branch addresses that access discrete blocks of states. EPS448 #1 is assigned branch addresses 0, 1, and 2, that each correspond to a different sequence of states within the EPLD. While EPS448 #1 has control of the output bus, it drives the branch bus with the address of its current sequence so that other machines are not activated. When it is ready to pass control, it drives a new value onto the branch bus. For example, when EPS448 #1 outputs the branch address 3, EPS448 #2 assumes control.

Figure 6 shows a portion of the ASM code required for EPS448 #1; this code is similar to the code for simple cascading. While EPS448 #1 is idle, it must look for any of its three possible branch addresses (BA0, BA1, and BA2) to become valid. If one of these values appears on the branch bus, EPS448 #1 jumps to the start of the corresponding sequence (START0, START1, or START2).

When EPS448 #1 is ready to give up control, it jumps to QUIT until it calls the next address. The EPS448 EPLD calls a different routine for each branch by specifying a different value on the branch bus. The GO0 to GO7 macros correspond to binary branch addresses, i.e., G03 = "011".

Figure 6. ASM Code for Addressed-Branch Cascading (Part 1 of 2)

The addressed cascading shown in Figure 5 uses three input lines (IN0, IN1, and IN2) to address different routines within a single EPLD. By specifying different addresses at the QUIT label, any routine in any other device can be called.

```
MACROS:  % F F F F  .....   F %
         % 0 1 2 3  .....  15 %

   GO0 = " 0 0 0 "
   GO1 = " 0 0 1 "
   GO3 = " 0 1 0 "
   GO3 = " 0 1 1 "
   GO4 = " 1 0 1 "
   GO5 = " 1 0 1 "
   GO7 = " 1 1 1 "
  ANY2 = " 1  ......  0 "
  ANY3 = " 0  ......  1 "

EQUATIONS:                          % Input Address Decodes %
           BA0 = /IN2*/IN1*/IN0;  % Branch Address 0      %
           BA1 = /IN2*/IN1*IN0 ;  % Branch Address 1      %
           BA2 = /IN2*IN1*/IN0 ;  % Branch Address 2      %
```

Figure 6. ASM Code for Addressed-Branch Cascading (Part 2 of 2)

```
PROGRAM:

IDLE:       IF BA0 THEN [Z] JUMP START0;
            ELSEIF BA1 THEN [Z] JUMP START1;
            ELSEIF BA2 THEN [Z] JUMP START2;
            ELSE [Z] JUMP IDLE;

START0:     [GO0 ANY3] CONTINUE;

START1:     [GO1 ANY3] CONTINUE;

START2:     [GO2 ANY3] CONTINUE;
            .
            .
            .
QUIT:       IF IN5 * IN6 THEN [GO3 ANY2] JUMP IDLE;
            ELSEIF   IN6 THEN [GO4 ANY2] JUMP IDLE;
            ELSEIF   IN7 THEN [GO5 ANY2] JUMP IDLE;
            ELSE              [GO7 ANY2] JUMP IDLE;
```

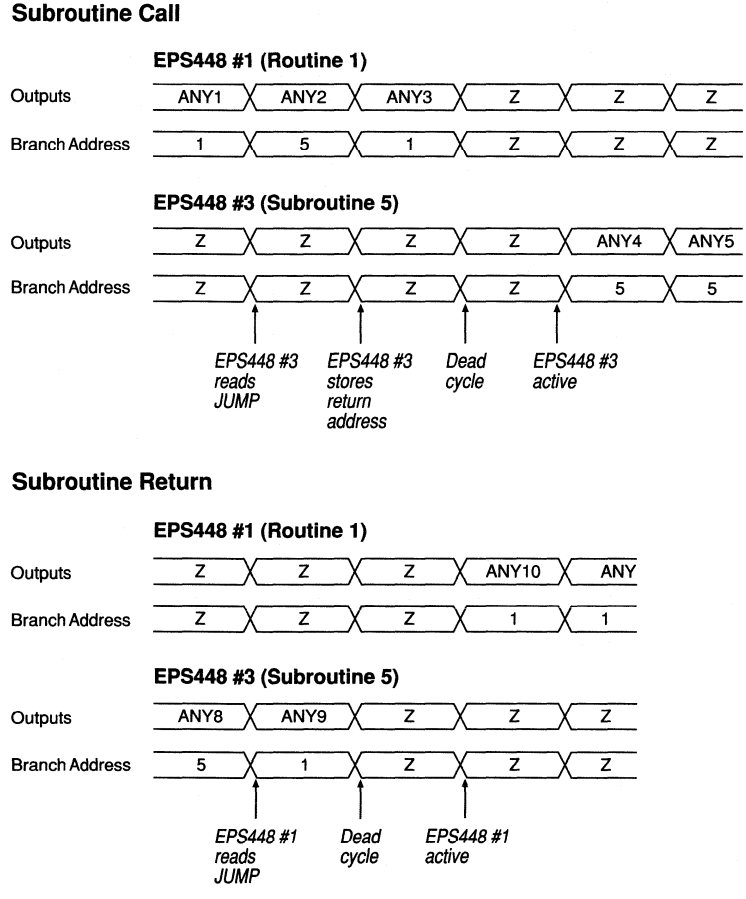

Vertical Subroutine Calls

The configurations discussed so far have used a JUMP function between EPS448 EPLDs. However, subroutine calls can be performed between EPLDs. A subroutine call differs from a jump in that it must eventually jump back to the machine that called it. The subroutine must be able to recall the return address of the calling EPLD. The return address can be stored on the EPS448 internal stack. Subroutines are most easily implemented with the branch bus configuration shown in Figure 5.

The time required for passing control is illustrated in Figure 7. Before the subroutine call, EPS448 #1 is in control of the bus and is executing the block of states starting with 1 (branch address 1). It drives the branch bus with the value 1 until it is ready to call a subroutine (branch address 5) within another EPLD (EPS448 #3). To start the subroutine call, EPS448 #1 specifies the address of subroutine 5 on the branch bus, which wakes up EPS448 #3. EPS448 #1 then applies return address 1 so that EPS448 #3 can store the return value on its internal stack. EPS448 #1 tri-states both the branch and output buses on the next Clock cycle. A "dead" Clock cycle follows, after which EPS448 #3 takes control of the output bus.

Figure 7. Timing of a Vertically Cascaded Subroutine

In a sample cascaded subroutine call, EPS448 #1 calls subroutine 5 from EPS448 #3 by placing 5 on the branch bus. Next, EPS448 #1 places its own address (1) on the branch bus to tell the subroutine where to return. Finally, EPS448 #1 tri-states, and EPS448 #3 takes control of the output bus on the next Clock period. When EPS448 #3 (5) is ready to return, it puts the return value (1) on the branch bus and then idles.



The syntax required for EPS448 #3 is shown in Figure 8. After EPS448 #3 reads its address, it still leaves its outputs tri-stated for two more Clock cycles. The `ANDPUSHI` opcode is used to store the return address in the stack by masking off inputs `IN3` to `IN7`. See *Application Brief 63 (Multiway Branching with the EPS448 SAM EPLD)* in this handbook for information on how to use `ANDPUSHI` with dispatch routines.

To return from the subroutine, EPS448 #3 applies the return address (branch address 1) on the branch bus for one Clock cycle (see Figure 7). EPS448 #1 reads this address and takes control of the output bus by jumping to the address at the top-of-stack. The `RETURN` opcode at the `QUIT` label accomplishes this task. This address must be between 0 and 7 because the inputs `IN3` to `IN7` are masked off when the `ANDPUSHI` opcode pushes the return address onto the stack. In this example, the top-of-stack is 1, and

Figure 8. ASM Code for Vertically Cascaded Subroutine

In this example, EPS448 #3 is used as a subroutine at branch address 5. After it is called, it stores the return address on the stack with the ANDPUSHI command. The RETURN command jumps to an absolute address, which puts the correct return address onto the branch bus.

```
MACROS:     % Output values for branch bus %
            GO0 = "000"     GO1 = "001"     GO2 = "010"
            GO3 = "011"     GO4 = "100"     GO5 = "101"
            GO6 = "110"     GO7 = "111"

            % Output bus values %
            ANY3 = "10...1", ANY8 = "00...1", ANY9 = "01...0"

EQUATIONS:                          % Branch address inputs %
            BA5 = IN2*/IN1*IN0;     % Branch address 5       %

PROGRAM:

            % Idle or start the subroutine %
IDLE:       IF BA5 THEN [Z] ANDPUSHI 7H GOTO START6;
            ELSE [Z] JUMP IDLE;

START5:     [Z] CONTINUE;           % Dead cycle      %
            [GO5 ANY3] CONTINUE;    % Take control of %
                                    % output bus      %
            .
            .
            .
QUIT5:      [GO5 ANY8] RETURN;      % Subroutine return %

            % Jump table (reserved locations) %
0D: [GO0 ANY9] JUMP IDLE;           % Also the power-up state %
1D: [GO1 ANY9] JUMP IDLE;
2D: [GO2 ANY9] JUMP IDLE;
3D: [GO3 ANY9] JUMP IDLE;
4D: [GO4 ANY9] JUMP IDLE;
5D: [GO5 ANY9] JUMP IDLE;
6D: [GO6 ANY9] JUMP IDLE;
7D: [GO7 ANY9] JUMP IDLE;           % Error condition %
```

a jump to address 1D (1 decimal) is performed. Address 1D has the correct output specification to signal EPS448 #1 to take control of the buses. Finally, EPS448 #3 jumps back to IDLE.

Branch address 7 is unusable because it corresponds to the default condition of all branch-address bits pulled up. If address 7D is reached, an error has occurred and all cascaded EPS448 EPLDs must be reset.

During power-up, the EPS448 EPLD starts in state 0D. All machines jump to their idle state on the next Clock cycle, and the EPLD that initiates control after power-up jumps to its correct starting state.

After making the subroutine call, EPS448 #1 must go into a high-impedance state and wait for the routine to finish: it can return to its IDLE starting state or it can stay in a separate loop. In the first case, EPS448 #1 regains control by recognizing its address, effectively restarting. In the second, EPS448 #1 returns to the state it was in when the call was made.

Master/Slave Cascading

Another vertical cascading method is the master/slave configuration. See Figure 9. A single EPS448 device (the master) controls the overall sequencing by calling routines within other EPS448 EPLDs (slave #1 through slave #4). The slaves are cascaded with all but one of the outputs connected to the output bus. The single output (F00), called nDone in Figure 9, is passively pulled up through the resistor. When an active slave EPLD has finished executing its routine, it pulls the nDone line low, alerting the master that it should jump to the next routine.

Any number of control lines can run from the master to a given slave depending on the number of routines within the slave. For n control lines, the slave can have $2^n - 1$ routines. When all control lines are high, the slave is idle. In this example, slave #1 has two inputs and thus can contain up to three routines. When both CONTROL0 and CONTROL1 are high, slave #1 is idle. An unlimited number of slaves can be added to this system. Furthermore, unlike the address-branch cascading method, the number of inputs to the other EPLDs does not increase as more routines are added.

Figure 9. Master/Slave Cascading

In a master/slave configuration, one EPS448 EPLD (the master) controls sequencing. It calls routines from any number of slaves. When the slaves are finished, they pull the nDONE line low.

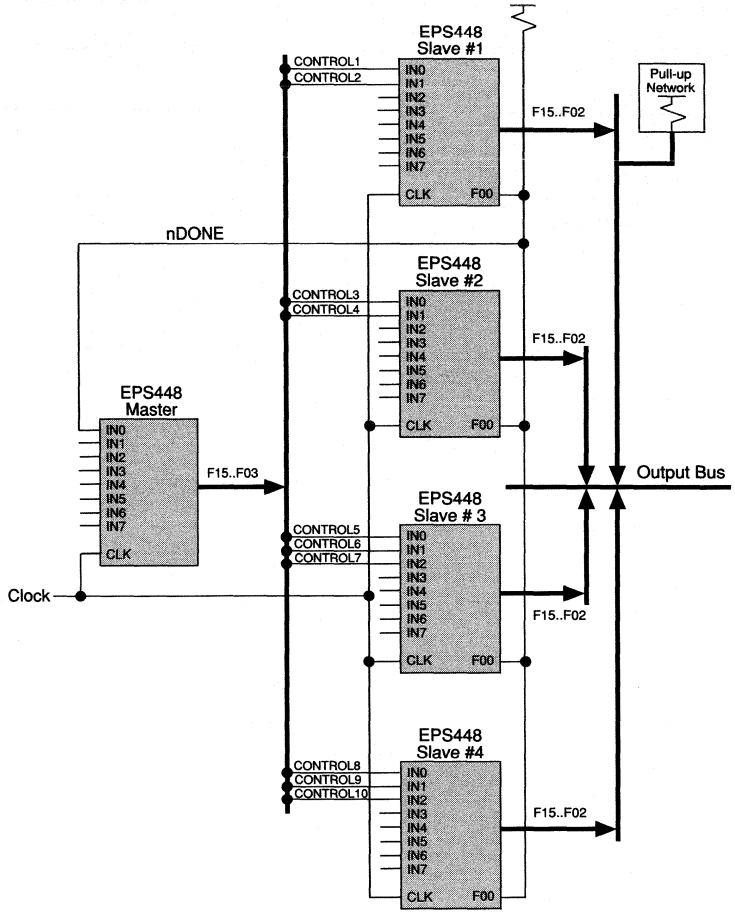


Figure 10 shows the timing associated with the master/slave configuration. Slave #1 originally has control of the output bus. It signifies that it is done by pulling nDONE low. The master responds by initiating the slave #2 machine, which then takes over the output bus. The output bus is undefined for a single Clock cycle.

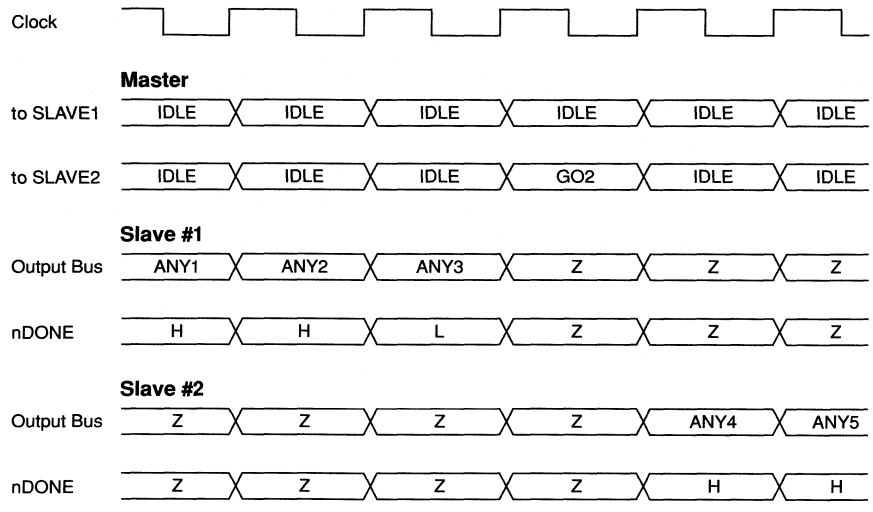


Figure 10. Timing of Master/Slave Cascading

In the example of master/slave cascading shown in Figure 9, slave #1 starts with control of the output bus. When it is done, it brings nDONE low. The Master then initiates slave #2 by sending it an active starting address.

Final Tips on Vertical Cascading

The following tips provide guidelines for vertical cascading:

- Each technique for vertically cascading EPS448 EPLDs consumes output and input pins from each device to accommodate control passing. If needed, you can create additional outputs by horizontally cascading EPLDs (see the ***SAM+PLUS Reference Guide*** for more information). You can generate additional inputs by multiplexing several signals and reducing them to the eight inputs available with the EPS448 EPLD.

- Partitioning, the process of dividing the code or state segments among multiple EPS448 EPLDs, is the most important step in creating a vertically cascaded sequencer. Most large sequential applications have natural divisions that indicate where partition borders should fall. The general goal in partitioning is to create blocks of sequences that fit within a single EPS448 EPLD, and to minimize control transfers from one EPLD to another.

- All Clock pins should be tied together, as should all Reset pins. Inputs to different EPS448 EPLDs need not be tied together. In fact, it may be desirable for different sections or routines of the overall sequencer to branch on the basis of different input signals.

- In vertical cascading, the output lines from the resulting sequencer originate at a tri-state output bus. The output bus signals should never be connected to system Clock or Latch Enable inputs. As in any tri-state bus, the output bus of a cascaded EPS448 EPLD is not guaranteed to be glitch-free during transitions from high-impedance to output valid, or vice versa.

- Only one EPLD can become active on power-up. All other EPLDs should jump to IDLE, and therefore should have the following line in their source files:

```
OD: [Z] JUMP IDLE.
```

- If every EPLD goes IDLE without first calling another EPLD (or bringing nDONE low), the system is deadlocked in an idle state. You can avoid this problem with proper coding, or with a watchdog timer implemented in one of the EPS448 EPLDs. The watchdog timer counts to a final count value that corresponds to a fixed time period. During normal operation, the other EPS448 EPLDs periodically clear the watchdog counter, preventing it from reaching its final count value. Therefore, if the timer reaches its final count value, a system error has occurred. The watchdog timer should then reset the system or generate the appropriate system error status.

Conclusion

You should select a method of vertical cascading—simple, addressed-branch, vertical subroutine, or master/slave cascading—based on how the sequencer is best partitioned. By vertically cascading EPS448 EPLDs, you can easily accommodate those applications that require more states or more microcode depth than a single EPS448 EPLD can provide.

Fitting Complex Designs in MAX 5000 EPLDs

April 1992, ver. 3

Application Brief 77

Introduction

MAX+PLUS II development software uses a combination of advanced logic synthesis techniques and a heuristic fitter to efficiently map logic designs (called "projects" in MAX+PLUS II) into MAX 5000 EPLDs. The MAX+PLUS II Compiler typically synthesizes even the most complex projects in less than five minutes.

However, certain projects are more difficult to fit. These projects contain very complex combinatorial logic or require more macrocells than are available in the target EPLD. The MAX+PLUS II Compiler uses logic synthesis techniques specifically developed to fit these projects quickly and, in most cases, automatically.

Sometimes projects require subtle modifications to enable the Compiler to synthesize logic and obtain a fit. This application brief discusses techniques to fit complex projects and covers the following topics:

- Logic synthesis
- Common Compiler error messages
- Fitting a project with too many macrocells
- Simplifying complex combinatorial functions
- Placing `SOFT` & `MCELL` buffers

Logic Synthesis

The Logic Synthesizer module of the MAX+PLUS II Compiler uses a number of algorithms to ensure that the EPLD macrocell structure is used as efficiently as possible. For example, if a project contains too many macrocells for a specified MAX 5000 EPLD, the Logic Synthesizer can allocate logic from buried combinatorial macrocells to expander product terms (shared logic expanders). It also resynthesizes projects that contain too many shared expanders by transferring logical expressions implemented on expanders to macrocells. Logic from up to three shared expanders can be transferred into each macrocell. The Logic Synthesizer balances the number of macrocells and expander product terms to maximize efficient use of resources. Thus, the Compiler can fit projects that originally required too many instances of a particular logical resource; most complex projects are automatically fitted.

Compiler Messages

If a project is too large or complex, the MAX+PLUS II Compiler generates an error message that specifies the problem and, if applicable, indicates the error location. The following examples show how most logic synthesis and fitting errors are expressed.

- **Project requires too many** *<number/number>* **macrocells**

  This message indicates that the current project contains too many macrocells for the specified MAX 5000 EPLD. The ratio *<number/number>* is the ratio of macrocells in the project to the macrocells available in the EPLD.

- **Project too complex** [*<text>*]

  This message indicates that an equation for a node, in its current form, is too complex for a MAX 5000 EPLD. The Compiler may generate this message while processing complex combinatorial expressions.

- **Project requires too many** *<number/number>* **shareable expanders** [for *<text>*]

  This message indicates that the project is too complex and requires too many shareable expanders for the target MAX 5000 EPLD. The ratio *<number/number>* is the ratio of shareable expanders in the project to the total number of available shareable expanders. The *<text>* portion of the message may specify the particular macrocell or Logic Array Block (LAB) that requires too many expanders.

Once the Compiler generates an error message, simple modifications often achieve a fit. The following sections provide suggestions on how to obtain or improve a fit. MAX+PLUS II Help explains all Compiler error messages. For additional assistance, call Altera Applications at (800) 800-EPLD.

How to Fit a Project with Too Many Macrocells

MAX+PLUS II generates an error message when a project contains too many macrocells for the specified EPLD, and the Logic Synthesizer has already unsuccessfully tried to allocate buried macrocells to shared expanders in the project. If this error occurs, macrocells must be eliminated. The following guidelines may help reduce the number of macrocells:

- You can eliminate any MCELL or SOFT buffers that you have manually placed into combinatorial logic.

- You can remove SOFT buffers from Altera-provided combinatorial TTL macrofunctions. SOFT buffers are placed in complex combinatorial macrofunctions to partition the logic into multiple macrocells. Logic synthesis usually removes the SOFT buffers that are not required inside these macrofunctions. However, you can sometimes reduce macrocell count by removing one or more buffers. Table 1 lists combinatorial macrofunctions with and without SOFT buffers. For example, the 7485 macrofunction contains four SOFT buffers. You can remove one or more buffers, particularly if the inputs or outputs are single product terms. However, you must first copy the Altera-provided macrofunction from the appropriate subdirectory of the

Table 1. Complex Combinatorial TTL Macrofunctions

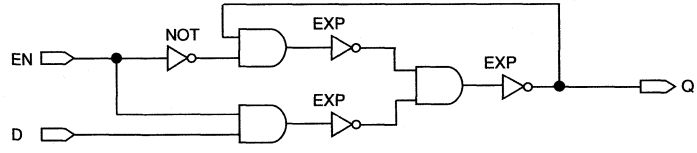
Function	Macrofunctions with Soft Buffers	Macrofunctions without Soft Buffers
Arithmetic Function	8FADD, MULT4, 7480, 7483, 7481	MULT2, MULT4, 7482, 74183
Comparator	8MCOMP, 7485, 74518	
Converter		74184, 74185
Decoder		7442, 7443, 7444, 7445, 7446, 7447, 7448, 7449, 74138, 74139, 74154, 74155, 74156
Latch	74116, 74259, 74279	
Multiplexer		74151, 74153, 74157, 74298
Parity Checker	74180, 74280	

\MAXPLUS2\MAX2LIB directory, place it in the same directory as your project, and then make the necessary edits to the copied macrofunction.

- You can implement buried registers or latches with shareable expanders to reduce the macrocell count. Expanders can be cross-coupled to form a variety of register and latching functions. Figure 1 shows a transparent D latch placed on cross-coupled expanders. Expander-based macrofunctions are available in the MAX+PLUS II TTL MacroFunction Library.

Figure 1. Transparent D Latch Implemented with Cross-Coupled Shared Expanders

If a project contains too many macrocells, buried register functions can be placed on cross-coupled shared expanders.



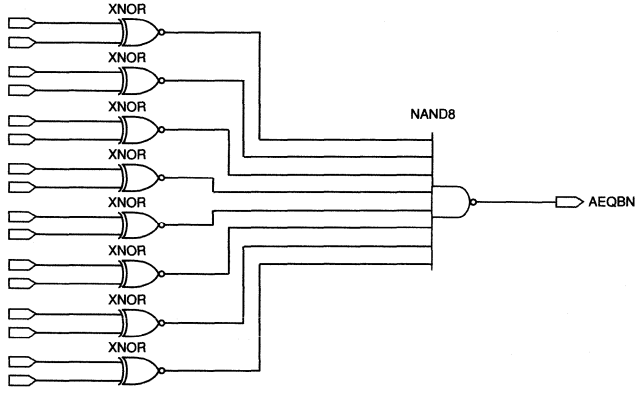
If these techniques do not yield the desired fit, contact the Altera Applications Department at (800) 800-EPLD for assistance. As a last resort, you may have to remove logic from the project to achieve a fit, or you may have to partition the project into more than one MAX 5000 EPLD.

How to Simplify Complex Combinatorial Functions

The MAX+PLUS II Compiler generates error messages when a project's combinatorial logic is too complex. This complexity can result from cascading several combinatorial macrofunctions, such as adders and comparators, or from heavy use of XOR functions. Figure 2 shows complex combinatorial logic caused by heavy use of the XNOR function.

Figure 2. Complex Combinatorial Logic

Heavy use of the XNOR function creates complex logic.



Placing SOFT & MCELL Buffers

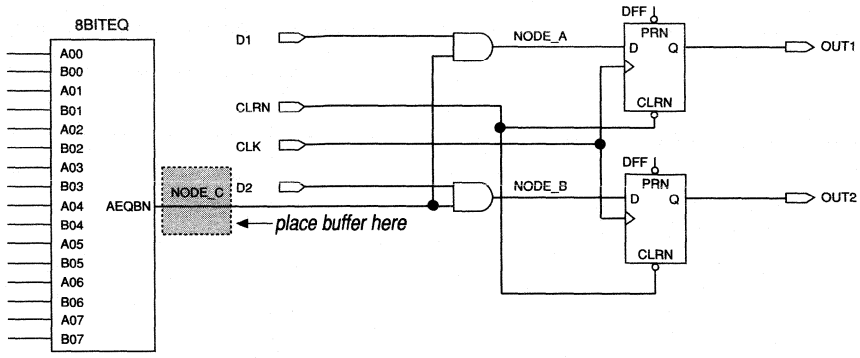
To separate complex combinatorial expressions, you can insert one or more SOFT or MCELL buffers into a project. Placing a SOFT or MCELL buffer consumes a macrocell to implement a portion of a complex logic expression. The complex expression is then distributed over two or more macrocells and simplified. Although SOFT or MCELL buffers in a project can affect the timing of a project, proper placement of these buffers can significantly simplify the design, and therefore reduce the number of shared expanders and macrocells.

The best location for a SOFT buffer may not be obvious. Although the Compiler's `logic too complex` message specifies an error location, the identified node name indicates the output of the complex expression, which is usually not the correct location for a SOFT buffer. To find the best location, you must analyze the logic feeding the node to determine the cause of the expression's complexity.

In the schematic shown in Figure 3, the Compiler has flagged NODE_A as an expression that is too complex. Since NODE_B has the same structure (and thus the same complexity) as NODE_A, you can insert a SOFT buffer at NODE_C to simplify the complex expressions of both NODE_A and NODE_B.

Figure 3. SOFT Buffer Minimizes Multiple Complex Expressions

Inserting a SOFT buffer at NODE_C reduces the complexity of the logic for both NODE_A and NODE_B.



Guidelines for Placing SOFT Buffers

You can use the following guidelines to prevent and resolve fitting problems and create a more efficient fit that can include additional logic in the target EPLD:

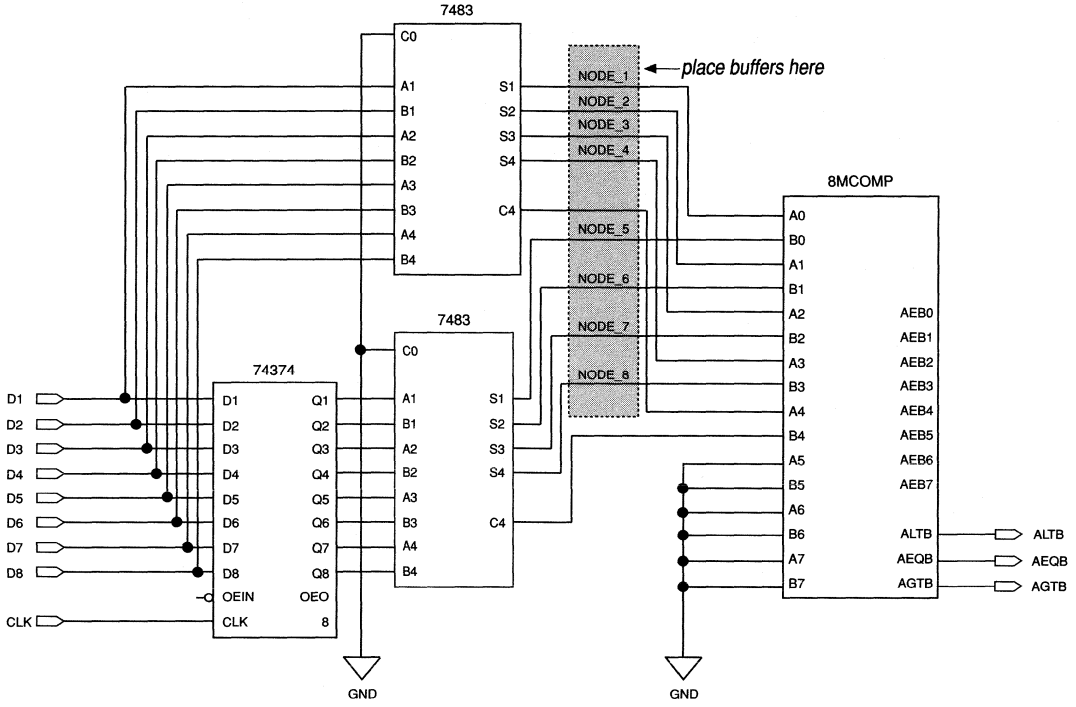
- Complex combinatorial expressions that feed flipflop controls and tri-state buffers are good locations for SOFT buffers. The Output Enable tri-state input and the Preset, Clear, and Clock inputs to flipflops all have a single product term associated with them. If the expression feeding a control input is complex or feeds multiple locations, it may require a large number of shared expanders. Placing a SOFT buffer between the complex expression and the flipflop input reduces the number of expanders used.

- Complex combinatorial outputs of macrofunctions are good locations for SOFT buffers if they do not feed a flipflop input or an I/O pin. You may need to isolate complex expressions before routing them into another macrofunction. Inputs to macrofunctions are also good locations for SOFT buffers if the macrofunctions are fed by complex expressions.

  For example, Figure 4 shows the outputs of two 7483 adders that feed an 8-bit magnitude comparator. Both the 7483 and the comparator contain complex logic, but placing SOFT buffers at NODE_1 through NODE_8 significantly reduces the complexity of the overall function.

Figure 4. SOFT Buffers Minimize Complex Expressions that Feed an 8-Bit Magnitude Comparator

Cascading complex combinatorial macrofunctions may require SOFT buffers. In this example, NODE_1 through NODE_8 require SOFT buffers.

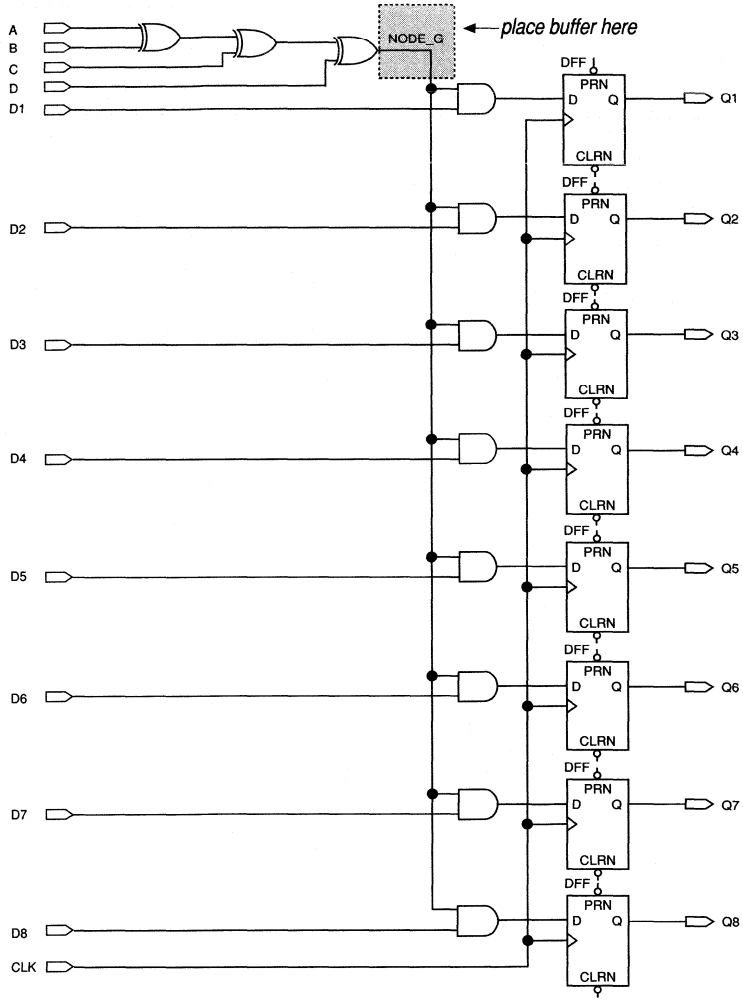


❑ Complex combinatorial expressions that feed many different locations are good locations for SOFT buffers. If a complex expression feeds multiple Logic Array Blocks (LABs), the logic associated with the expression is duplicated in each LAB fed by the expression, and the number of shared expanders is increased. Placing a SOFT buffer at the complex combinatorial output reduces the number of expanders. The Report File (.RPT) generated by the MAX+PLUS II Compiler contains a list of all duplicated expanders.

Figure 5 shows a project that has 1 output in each of the 8 LABs of an EPM5128 EPLD. The project consumes 5 shareable expanders in each LAB, a total of 40 expanders. Inserting a SOFT buffer at NODE_G reduces the required number of expanders to only 3.

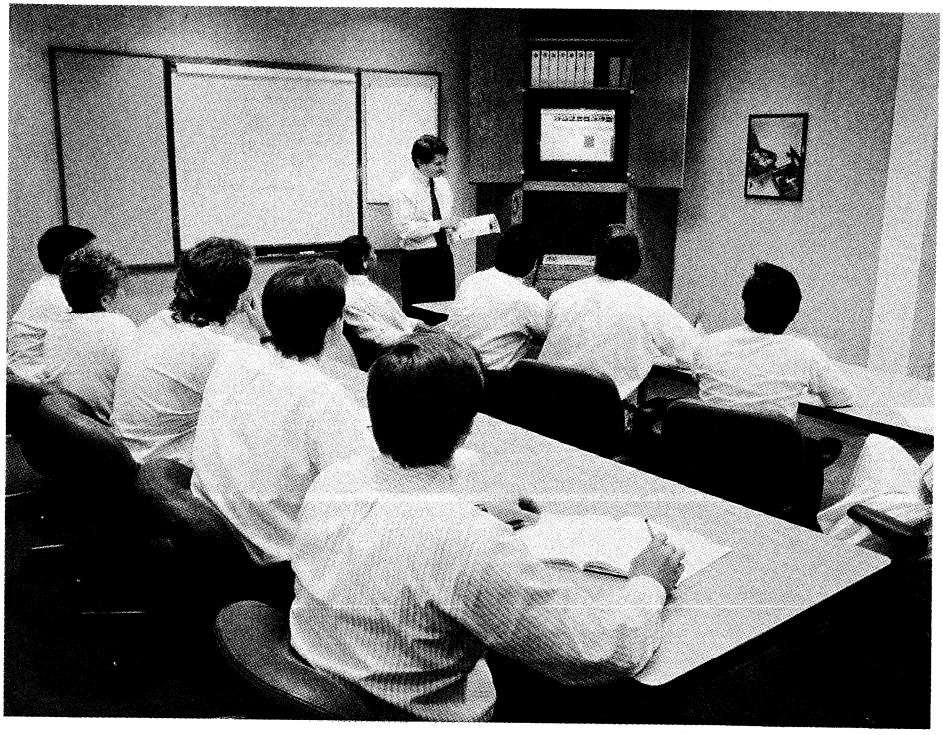
Figure 5. SOFT Buffer Minimizes a Complex Expression that Feeds Multiple LABs

If a logical expression feeds multiple LABs, it may be a good place for a SOFT buffer. In this example, inserting a SOFT buffer at NODE_G reduces the number of expanders from 40 to 3.



Conclusion

When MAX+PLUS II Compiler error messages indicate that logic is too complex or that a fit cannot be found, you can use a variety of methods to help achieve a fit. By adding or removing strategic MCELL and SOFT buffers or implementing registers and latches with shareable expanders, you can modify complex projects to fit into a single MAX 5000 EPLD.



Troubleshooting EPLD Programming Problems

April 1992, ver. 1

Application Brief 81

Introduction

Altera's Windows-based MAX+PLUS II development system provides an efficient way to develop and test your designs. MAX+PLUS II allows you to enter, compile, and simulate a design with minimum effort and maximum speed. Altera's programming hardware lets you program EPLDs within a matter of seconds. You can also apply functional test vectors to a device to compare these vectors with software simulation results. Although MAX+PLUS II is easy to use, you may occasionally encounter programming problems (i.e., MAX+PLUS II issues an error message).

This application brief provides solutions to the most common programming problems and describes the latest programming hardware and software. A troubleshooting checklist is also provided to help you and Altera Applications Engineers diagnose the programming problem as quickly as possible.

If you encounter a functional problem (i.e., programming and compilation are successful, yet the EPLD does not perform correctly), refer to *Application Brief 87 (Troubleshooting Functional Problems in EPLDs)* in this handbook.

This application brief discusses the following topics:

- Programming hardware
- Programming software
- How to calibrate your programming hardware
- How to update silicon IDs
- Common programming errors

Programming Hardware

Altera offers a range of programming hardware, including Logic Programmer cards, programming units, and a variety of adapters, giving you the flexibility to assemble a programming system that is compatible with your particular computer. A typical hardware setup consists of a Logic Programmer card, a programming unit, and a programming adapter (for those devices that require it). Many programming problems are a direct result of hardware incompatibility. Table 1 contains the hardware compatibility information for a specific EPLD. While this table contains the most recent information available at the time this handbook was printed, it is updated and published each quarter in the Altera *Applications Engineering News & Views* customer newsletter. Always check the newsletter for the latest hardware compatibility information.

Table 1. EPLD/Adapter Compatibility

EPLD *Notes (1), (2)*	Package	Adapter *Note (3)*
EP310, EP320	DIP	None required
EP330	DIP	None required
	J-Lead	PLEJ330
	SOIC	PLES330
EP600/610/610T/630	DIP	PLED610
	J-lead	PLEJ610
	SOIC	PLES610
EP900/910/910A/910T	DIP	PLED910
	J-lead	PLEJ910
EP1800/1810/1810T/1830	J-lead	PLEJ1810
	J-lead	PLMJ1810 *(4)*
	PGA	PLEG1810
EPB2001	J-lead	PLEJ2001
EPM5016	DIP	PLED5016
	J-lead	PLEJ5016
	SOIC	PLES5016
EPM5032	DIP	PLED5032
	DIP	PLMD5032 *(4)*
	J-lead	PLEJ5032
	J-lead	PLMJ5032 *(4)*
	SOIC	PLES5032
EPM5064	J-lead	PLEJ5064
	J-lead	PLMJ5064 *(4)*
EPM5128	J-lead	PLEJ5128
	J-lead	PLMJ5128 *(4)*
	PGA	PLEG5128
EPM5130	J-lead	PLEJ5130
	J-lead	PLMJ5130 *(4)*
	PGA	PLEG5130
	QFP	PLEQ5130
	QFP (with carrier)	PLMZ5130 *(4)*
EPM5192	J-lead	PLMJ5192 *(4)*
	PGA	PLMG5192 *(4)*
EPM7032	J-lead	PLMJ7032-44 *(4)*
	QFP	PLMQ7032-44 *(4)*
EPM7096	J-lead	PLMJ7096-84 *(4)*
EPM7192	PGA	PLMG7192-160 *(4)*
EPM7256	PGA	PLMG7256-192 *(4)*
EPS448	DIP	PLED448
	J-lead	PLEJ448
EPS464	J-lead	PLEJ464
	J-lead	PLMJ464 *(4)*
	QFP	PLEQ464
	QFP	PLMQ464 *(4)*

Notes to table::

(1) All EPLDs, except MAX 7000 EPLDs, can be programmed with the PL-MPU or PLE3-12A programming unit. MAX 7000 EPLDs can be programmed only with the PL-MPU unit.
(2) The LP4, LP5, or LP6 programming card can be used with all EPLDs.
(3) Refer to the *PLED/J/G/S/Q & PLMD/J/G/S/Q Data Sheet* in the Altera 1992 ***Data Book*** for more information.
(4) This adapter supports functional testing and continuity checking and can be used only with the PL-MPU programming unit.

Logic Programmer Cards

The MAX+PLUS II software currently supports the LP5 Programmer Card for use with Micro Channel computers and the LP6 Programmer Card for use with PC-AT, PS/2, and compatible computers. MAX+PLUS II also supports the LP4, an older programmer card for the PC-AT.

Programming Units

MAX+PLUS II software currently supports two programming units: the PLE3-12A (no longer in production) and the Master Programming Unit (MPU). The MPU has the following advantages:

- The MPU can program all Altera devices, including the MAX 7000 and Synchronous Timing Generator (STG) EPLDs. The PLE3-12A programs only the Classic, MAX 5000, Stand-Alone Microsequencer (SAM), and Micro Channel EPLDs.

- The MPU has functional testing capabilities. You can apply MAX+PLUS II simulation vectors to a programmed device and compare the actual device outputs to simulated outputs. The PLE3-12A does not support this feature.

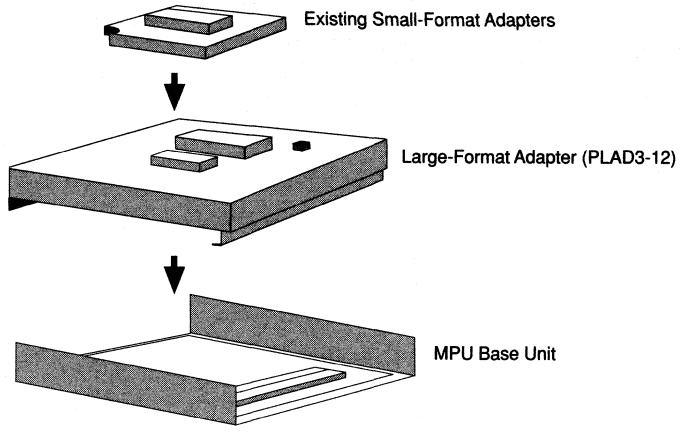
Adapters

Altera produces two types of adapters: large- and small-format. While the MPU can use both formats, the PLE3-12A supports only small-format adapters. Large-format adapters used with the MPU have two major advantages over small-format adapters:

- Large-format adapters can perform functional testing. You cannot use small-format adapters to functionally test programmed EPLDs, even if you have an MPU.

- The MPU comes with a large-format adapter, the PLAD3-12, that emulates a PLE3-12A programming unit. All small-format adapters can be plugged into this large-format adapter to program EPLDs (see Figure 1).

Figure 1. Small- and Large-Format Adapter Assembly



Programming Software

Altera offers three programming software packages: MAX+PLUS II Programmer, MAXPROG, and LogicMap II. MAXPROG and LogicMap II are stand-alone programs, while the MAX+PLUS II Programmer is integrated into the MAX+PLUS II development system. Since most programming problems result from software incompatibility, you should always use the latest programming software. Refer to the current *Altera Applications Engineering News & Views* customer newsletter or call Altera Applications at (800) 800-EPLD for information on the latest software versions.

MAX+PLUS II Programmer

The MAX+PLUS II Programmer allows you to program Classic, MAX 5000, MAX 7000, and STG EPLDs. It can program EPLDs with Programmer Object Files (.POF) generated with MAX+PLUS or MAX+PLUS II, or JEDEC files (.JED) generated with A+PLUS or MAX+PLUS II. These capabilities make it the most versatile of the three programming software packages.

MAXPROG

MAXPROG allows you to program MAX 5000 devices. Free copies of the latest MAXPROG software are available from the Altera electronic bulletin board service (BBS) at (408) 249-1100.

LogicMap II

LogicMap II, which was originally provided as part of the A+PLUS software package, allows you to program all Classic, Micro Channel, and SAM

EPLDs with JEDEC Files created with A+PLUS or MAX+PLUS II. It cannot program EPM5016 and EPM5032 MAX EPLDs with JEDEC Files created by MAX+PLUS II. You can obtain a free copy of the latest version of LogicMap II from the Altera BBS.

Calibrating Programming Hardware

The Altera Field Diagnostic (AFD) utility checks the calibration of your programming hardware. AFD also determines whether the programming card is recognized, and whether the programming voltages are within tolerance. Simply type `AFD` at the DOS prompt to run AFD. The program first displays your system configuration. It then checks the integrity of the programming hardware at each of the MPU pins. You need a voltmeter to test each pin. You can download the latest version of AFD from the Altera BBS.

Warning: Running AFD version 8.1 could damage the LP6 card. Make sure you have the latest copy (version 8.2 or later).

Updating Silicon IDs

Altera Classic, MAX 5000, MAX 7000, and STG EPLDs contain silicon IDs that uniquely identify each version of an EPLD. Since some device versions are released after a MAXPROG or MAX+PLUS II update, you may need to reconfigure the programming software in these packages to recognize new IDs. If MAXPROG or MAX+PLUS II is not reconfigured, the message `Unrecognized EPLD` or `Device version is not enabled` appears when you attempt to program an EPLD with a new silicon ID.

The latest versions of MAX+PLUS II and MAXPROG contain information to program all currently available EPLDs. Table 2 shows silicon IDs and passwords for the newest Altera EPLDs.

Table 2. Silicon IDs and Passwords

EPLD	Version	Password
EPM5032	I, J, K	GFGDML
EPM5064	D, F	FLHLMI
EPM5130J (84-pin)	E	FBS9IQ
EPM5130 (100-pin)	E	FBS9DH
EPM5192	E	FLSUA7
EPM7032	B	IDI6UK

The first letter in the date code printed on top of the EPLD package indicates the device version. To reconfigure your programming software for new silicon IDs, use the following simple procedure(s). The version 'F' EPM5064 EPLD is used as an example; for other silicon IDs, simply substitute the appropriate name and password from the Table 2.

MAXPROG & MAX+PLUS (DOS) for MAX 5000 EPLDs:

MAX+PLUS (DOS) users *must* use MAXPROG to program EPLDs.

1. At the DOS prompt in the MAXPROG directory, type `MAXSID` and press ↵.
2. You are prompted for the EPLD name. Type `EPM5064` and press ↵.
3. You are prompted for the password. Type `FLHLMI` and press ↵.

The message `Silicon ID "ALTR (2)" enabled for EPM5064` is displayed.

MAX+PLUS II for Classic, MAX 5000, MAX 7000 & STG EPLDs:

1. Choose **Programmer** from the MAX+PLUS II menu.
2. Choose **Select Device** from the Options menu.
3. Choose *MAX 5000* in the *Device Family* drop-down list box.
4. Choose *EPM5064* from the *Available Devices* box.
5. Choose **Enable** to open the **Enable New Device Version** dialog box.
6. Type the password `FLHLMI` in the *Password* box.
7. Choose **Add** and then choose **OK**.
8. Choose **OK** to close the **Select Device** dialog box.

If you have any questions about these procedures or passwords for new silicon IDs, call Altera Applications at (800) 800-EPLD.

Common Programming Errors

This section lists the most common programming errors and describes the quickest and easiest ways to solve them. It assumes that you are using the latest MAX+PLUS II Programmer programming software (though the error messages are similar for MAXPROG and LogicMap II) and correct programming hardware. For detailed descriptions of all messages, refer to MAX+PLUS II Help, which provides information on possible causes of problems and suggests actions to fix them.

`Unrecognized device or socket is empty`

Cause 1:

The EPLD is not supported by your programming software.

Action:

Make sure that the EPLD that you are trying to program is supported by your programming software. Call Altera Applications to obtain the password that updates the MAX+PLUS II Programmer module for new devices. To add new devices while in the Programmer, refer to the "Updating Silicon IDs" section of this application brief.

Cause 2:

The device has been handled improperly.

Action:

Be extremely careful when using high-pin-count J-lead and quad flat pack (QFP) packages. Pins can bend easily if the device is mishandled or inserted into the programming adapter incorrectly. VCC and GND pins can also short together and cause this error.

Cause 3:

The adapter contacts are dirty.

Action:

Clean the adapter socket with rubbing alcohol on a regular basis. The socket contacts become dirty after several hundred parts are programmed.

Cause 4:

The Security Bit is turned on.

Action:

Be sure that the device is completely erased before reprogramming it. EPROM-based Classic, MAX 5000, and MAX 7000 EPLDs should be exposed to continuous UV light for one hour to ensure complete erasure. While an extra 15 minutes will not damage the device, Altera does not recommend excessive erasure times (i.e., overnight).

Device is not erased

Cause 1:

The erasure time was too short.

Action:

Expose EPROM-based Classic, MAX 5000, and MAX 7000 EPLDs to continuous UV light for 1 hour. An extra 15 minutes will not damage the device. However, Altera does not recommend excessive erasure times (i.e. overnight).

Cause 2:

The specifications on your UV eraser are incorrect or the eraser needs a new bulb.

Action:

Check the specifications of your UV eraser, ensuring that its wavelength is not greater than 4,000 Å. Altera recommends a wavelength of 2,537 Å, and a power rating of at least 12,000 $\mu W/cm^2$. Check the eraser bulb and replace it if it is not working.

Programming hardware is not installed

Cause 1:

The programming hardware connections are loose.

Action:

Make sure that the cable connecting the Logic Programmer card to the MPU is securely seated.

Cause 2:

The Logic Programmer card address does not match the software configuration file.

Action:

Check the DIP switch setting on the Logic Programmer card against the software configuration file. The address of the Logic Programmer card is initially set to 280. If this address is being used by another card, you can set the DIP switches to another value. The software configuration file must then be revised to reflect the new address.

In MAX+PLUS II, choose the **Auto-Setup** button in the **Hardware Setup** dialog box in the Programmer. The MAXPLUS2.INI file is automatically updated with the new address. In MAX+PLUS, you must edit the Programmer Address line of the MAXPLUS.CFG file. In A+PLUS, you must run option 3 of the A+PLUS INSTALL program. The new programming address will be reflected in the EPLD.SYS file.

Cause 3:

The AT bus is non-standard.

Action:

Enter the hardware specifications directly into the *Hardware Type* and *I/O address* box in the MAX+PLUS II **Hardware Setup** dialog box. You cannot use the **Auto-Setup** button in the **Hardware Setup** dialog box if your AT bus is non-standard. Several IBM PC-AT-compatible computers contain non-standard buses that do not recognize Altera programming cards when they are used together with MAX+PLUS II.

Cause 4:

Your hardware has been damaged.

Action:

Run AFD to confirm whether the hardware is performing to specifications. If AFD fails, call Altera Applications at (800) 800-EPLD for further assistance.

Warning: Running AFD version 8.1 could damage the LP6 card. Make sure you have the latest copy (version 8.2 or later).

Device is backwards in socket

Cause 1:
The device orientation is incorrect.

Action:
Be sure that the device is oriented correctly in the socket. This problem occurs most commonly with J-lead EPLDs. Position J-lead EPLDs so that the Altera logo is upside down in relation to the MPU.

Cause 2:
The device has not been handled properly.

Action:
Be extremely careful when using high-pin-count J-lead and QFP packages. Pins can bend easily if the device is mishandled or inserted into the programming adapter incorrectly. VCC and GND pins can also short together, resulting in an `Unrecognized device or socket is empty` error message.

Incorrect adapter or hardware error

Cause 1:
You did not use the "A" adapter for your MAX 5000 device.

Action:
If you are programming MAX 5000 EPLDs, make sure that the name of your small format adapter has an "A" suffix. Adapters that are not marked with this suffix are not compatible with current programming software.

Cause 2:
The hardware has been damaged.

Action:
Run AFD to confirm whether the hardware is performing properly. If the hardware fails the AFD check, call Altera Applications at (800) 800-EPLD for further assistance.

Warning: Running AFD version 8.1 could damage the LP6 card. Make sure you have the latest copy (version 8.2 or later).

Conclusion

Altera programming hardware and software typically provide simple, troublefree operation. Most programming problems can be eliminated by updating the software and hardware. Most other problems are explained in this application brief. If programming problems persist or if you have any questions about programming issues, contact Altera Applications at (800) 800-EPLD.

Troubleshooting Checklist

If you continue to have programming problems, feel free to call Altera Applications at (800) 800-EPLD. Please have the answers to the following questions available to help Altera Applications Engineers solve the problem quickly.

1. What type of computer are you using? _____
   _____
2. Which programming software version are you using? _____
   _____
3. Which programming hardware are you using? _____
   _____
4. What is the exact error message(s) you receive? _____
   _____
5. Which EPLD are you trying to program? _____
   (Name, package, and date code can be found on the top of the device.)
6. How many devices have failed? _____
7. Do the devices have the same date code? ❑Y ❑N
8. Have you been able to program other EPLDs (same type) with different date codes? ❑Y ❑N
9. Have you been able to program other EPLDs? ❑Y ❑N
10. Have you run AFD? ❑Y ❑N

Emulating Internal Buses in Classic, MAX & STG EPLDs

April 1992, ver. 3

Application Brief 82

Introduction

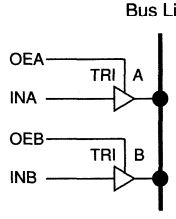
Altera's Classic, MAX 5000, MAX 7000, and Synchronous Timing Generator (STG) EPLDs allow internal buses to be emulated with multiplexing logic that replaces tri-state functions. A series of simple 2-to-1 multiplexers can create buses with two sets of input sources. Four-to-1 (and larger) multiplexers can create buses with four or more sources. Multiplexing also saves device resources and helps eliminate timing and loading problems. This application brief describes how to use multiplexers for different bus configurations and explains the benefits of this approach.

Two-Source Bus Configurations

Figure 1 shows the simplest bus configuration, a 1-bit bus created by connecting the outputs of two tri-state buffers (TRI) to a single bus line. The function table shows the possible states of the bus. When tri-state buffer A is enabled, the input to that buffer (INA) appears on the bus. When tri-state buffer B is enabled, the input to that buffer (INB) appears on the bus. If neither buffer is enabled, the bus is in a high-impedance "floating" state. If a bus is tied high with a pull-up resistor to prevent it from floating, the bus defaults to a high state if no TRI buffers are driving it.

Figure 1. One-Bit Bus

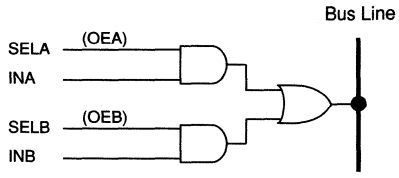
Two tri-state buffers can create a simple bus.



INA	OEA	INB	OEB	BUS LINE
X	0	X	0	Z or 1
X	0	0	1	0
X	0	1	1	1
0	1	X	0	0
1	1	X	0	1

Figure 2 shows two AND2 gates and an OR2 gate that emulate the tri-state functions of Figure 1. Each AND2 gate has a data input (INA or INB), and a select input (SELA or SELB) that represents the original Output Enable control. The function table shows that the AND/OR logic exactly emulates the original tri-state functions if one of the two outputs is always selected. If neither output is selected, the output of the AND/OR logic is low.

Figure 2. AND/OR Logic Emulating Tri-State Functions

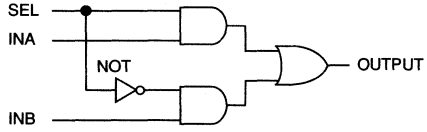


INA	SELA	INB	SELB	BUS LINE
X	0	X	0	0
X	0	0	1	0
X	0	1	1	1
0	1	X	0	0
1	1	X	0	1

The select controls are mutually exclusive, since only one input is ever enabled onto a bus at any given time. Therefore, you can encode them into a single input by making SELA the common select input, and then feeding the inverse of this signal into the previous SELB input. Figure 3 shows that these steps transform the AND/OR logic into a typical 2-to-1 multiplexer. The multiplexer functions in the same way as the AND/OR logic in Figure 2, except that the two select signals have been encoded into a single line.

Figure 3. Multiplexer Created with AND/OR Logic and Select Controls

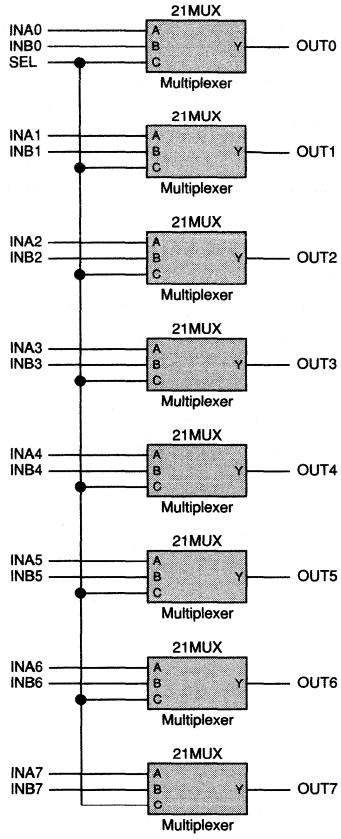
Encoded select lines transform the AND/OR logic from Figure 2 into a multiplexer.



INA	INB	SEL	OUTPUT
X	0	0	0
X	1	0	1
0	X	1	0
1	X	1	1

Additional 2-to-1 multiplexers, all controlled by a common select signal, can create wider buses. One multiplexer is necessary for each bit of the bus. For example, Figure 4 shows eight 2-to-1 multiplexers emulating a byte-wide bus.

Figure 4. Eight 2-to-1 Multiplexers Emulating a Byte-Wide Bus

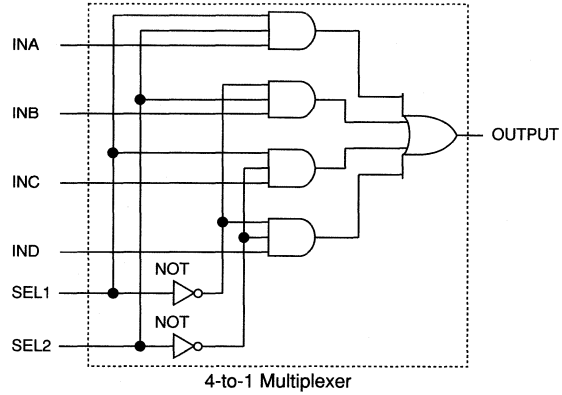


Buses with Three or More Sources

Larger multiplexers with multiple select inputs can emulate buses with three or more sources. Figure 5 shows how a 4-to-1 multiplexer can create a bus with four sources. Figure 5 also includes a truth table with the proper encoding for the select inputs. This type of multiplexer can also implement buses with two or three sources.

Additional multiplexers with shared select lines can create buses of nearly any width. For example, five 4-to-1 multiplexers can create a 5-bit-wide bus with two, three, or four sets of inputs.

Figure 5. 4-to-1 Multiplexer Implementing a Bus with up to Four Sources



INA	INB	INC	IND	SEL2	SEL1	OUTPUT
X	X	X	0	0	0	0
X	X	X	1	0	0	1
X	X	0	X	0	1	0
X	X	1	X	0	1	1
X	0	X	X	1	0	0
X	1	X	X	1	0	1
0	X	X	X	1	1	0
1	X	X	X	1	1	1

Implementing Bus Functions with AHDL

The Altera Hardware Description Language (AHDL) provides a quick alternative to schematic entry for implementing bus functions with multiplexing techniques. AHDL Text Design Files (.TDF) can implement buses with nearly any number of inputs and of nearly any width.

For example, Figure 6 shows the lines of AHDL code required to create an 8-bit bus with three sources. The data inputs are `a7` to `a0`, `b7` to `b0`, and `c7` to `c0`. The two select inputs, `sel1` and `sel0`, can be treated as an encoded group in AHDL. These select lines control which set of input signals are connected to the outputs through a series of simple If Statements.

You can easily modify this file to create a bus with more sources. For multiplexers with more than four data inputs, one more bit must also be added to the `sel` group (e.g., `sel[2..0]`) for each factor-of-two increase in the number of data inputs. For example, an 8-input multiplexer requires three select bits.

Figure 6. AHDL Implementation of Eight-Bit Bus with Three Sources

```
SUBDESIGN BUSMUX
(
   a[7..0], b[7..0], c[7..0], sel[1..0] : INPUT;
   out[7..0]                            : OUTPUT;
)
BEGIN
   IF (SEL[]==0) THEN
      OUT[]=A[];
   END IF;
   IF (SEL[]==1) THEN
      OUT[]=B[];
   END IF;
   IF (SEL[]==2) THEN
      OUT[]=C[];
   END IF;
END;
```


You can vary the width of the bus by changing the input and output group widths. For example, the declaration `a[5..0]` creates a 6-bit wide set of `A` inputs.

For more information on AHDL syntax, refer to *Application Note 22 (Designing with AHDL)* in this handbook or to MAX+PLUS II documentation.

Why Emulate Tri-State Functions?

Using multiplexing to emulate tri-state functions saves macrocells and I/O pins for applications that would otherwise require a bus external to the EPLD. Figure 7 shows a 4-to-1 multiplexer in a single macrocell that emulates a bus line with four sources. With conventional tri-stating techniques, the same function requires four macrocells and four I/O pins, as shown in Figure 8. By implementing the switching functions with the product terms inside the macrocell instead of the tri-state buffers and I/O pins external to the macrocell, the multiplexing approach saves three macrocells and three I/O pins.

Emulating tri-stated buses with logic eliminates timing hazards such as bus contention, which occurs when two or more tri-state outputs are simultaneously enabled onto a single bus line. This condition (usually

Figure 7. 4-to-1 Multiplexer in a Single Macrocell

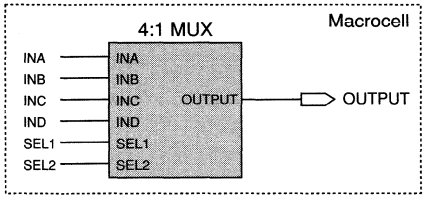
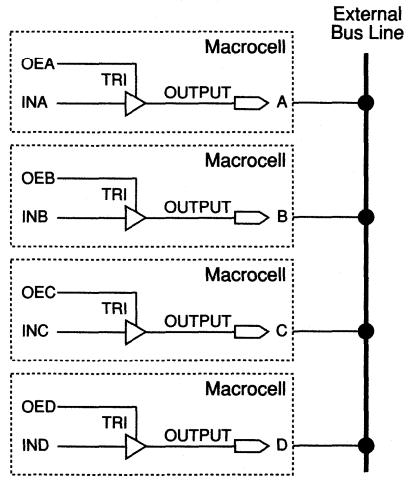


Figure 8. 4-to-1 Multiplexer Implemented with Traditional Tri-State Logic

A four-source bus that uses tri-stating requires four macrocells and I/O pins.



unintended) can cause an unpredictable logic level to propagate if one or more buffers are driving high while the other buffers are driving low, or vice versa. The select controls for simple AND/OR logic (shown in Figure 2) can both be enabled at the same time, but the result is a known logic level. The select controls can never be enabled at the same time if they are encoded, as in the true multiplexer configuration shown in Figure 3.

The only potential timing hazard for a multiplexer configuration is output glitching caused by input signal skew. EPLD architecture minimizes skew difficulties and glitching is seldom a problem in real designs. However, you must exercise care when driving edge-sensitive logic from multiplexer outputs.

Replacing tri-stated buses with logic also reduces capacitive loading limitations. High fan-outs to traditional buses create high capacitive loads that decrease bus bandwidth. Macrocells and feedback paths in EPLDs have constant delays, regardless of the number of signals entering the macrocell. If you implement buses and associated control logic with multiplexers, loading need not be considered as a design issue.

Conclusion

Although Altera's Classic, MAX 5000, MAX 7000, and STG EPLDs do not have internal tri-state capabilities, the tri-state function can be emulated with signal multiplexing techniques. You can use multiplexing to create buses of nearly any size in the EPLD. Multiplexing also provides the additional benefits of saving device resources, and eliminating timing and loading problems.

Programmable Frequency Divider with the EP610 Classic EPLD

April 1992, ver. 3

Application Brief 83

Introduction

The Altera EP610 EPLD combines the industry-standard Altera Classic architecture with advanced 1.0-micron CMOS EPROM technology to produce a high-speed (t_{PD} = 15 ns), 24-pin EPLD. This application brief compares the EP610 EPLD with the 22V10 device, focusing on architecture and design support. It also includes an example that illustrates EP610 EPLD features.

Architecture

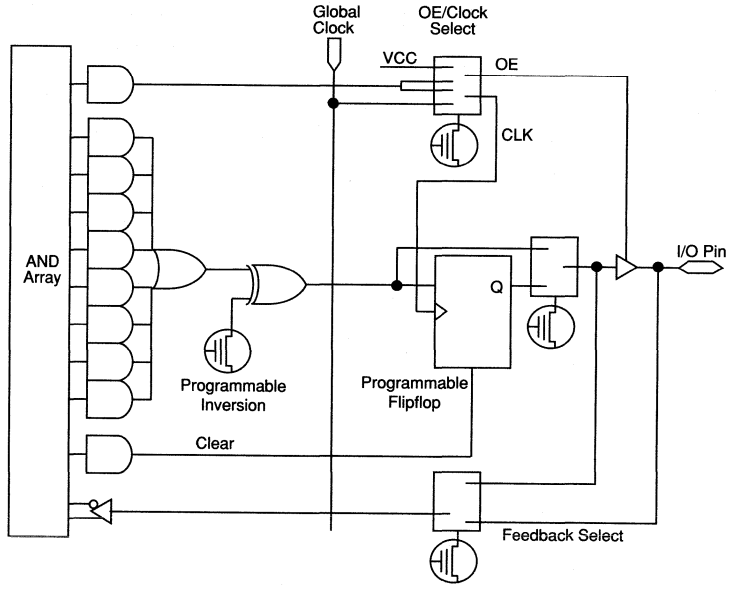
The EP610 EPLD and the 22V10 device have several features in common: both are available in a 24-pin package, both generate Boolean logic expressions with a sum-of-products array, and both process Boolean functions through macrocells (see Figure 1).

However, the two devices differ in important ways:

- The most obvious difference is the number of macrocells offered: the EP610 EPLD contains 16 macrocells; the 22V10 contains only 10.
- All EP610 macrocells contain a programmable flipflop that can be configured for D, T, JK, or SR operation. In contrast, the 22V10 device has a fixed D flipflop architecture. For example, a 10-bit counter implemented in a 22V10 requires 10 product terms to generate the 10th bit, while the same counter in an EP610 EPLD requires only a single product term for each bit when the internal registers are configured for T flipflop operation.
- The EP610 registers are individually configured to be clocked from a global or array Clock, while all of the 22V10 registers are clocked from a single global Clock source.
- The EP610 EPLD has 128 equally distributed product terms, 8 per macrocell. Individual product terms added to the Clock, Clear, and Output Enable inputs of each EP610 macrocell give you much greater flexibility than the 22V10 device, which has variable product-term distribution and contains 8 to 16 product terms per macrocell.
- The EP610 EPLD has a zero-power option that greatly reduces current consumption for low-frequency applications. The 22V10 only runs in full-power mode.

Figure 1. EP610 and 22V10 Macrocells

EP610 Macrocell



22V10 Macrocell

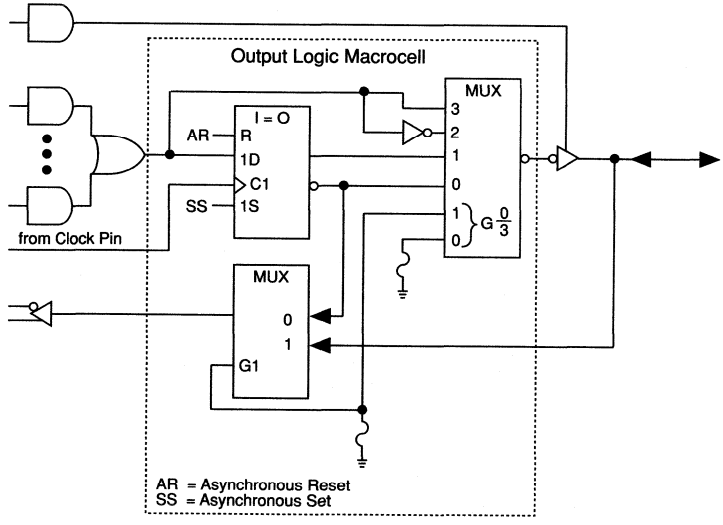


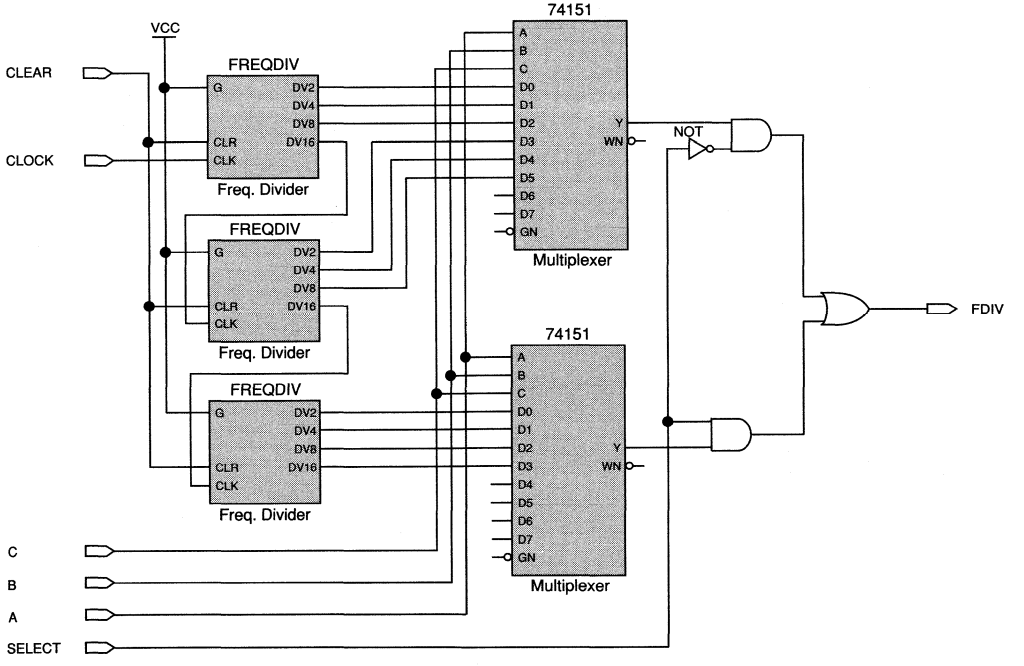
Table 1 summarizes the differences between the two devices.

Table 1. EP610 EPLD vs. 22V10 Device

Features	EP610	22V10
Macrocells	16	10
Programmable Clocks	Yes	No
Programmable Flipflops	Yes	No
Zero Power	Yes	No

The EP610 architecture provides greater register density and flexibility than the 22V10, allowing you to incorporate more logic. Figure 2 shows a programmable frequency divider, which is a common application for PLDs.

Figure 2. Programmable Frequency Divider



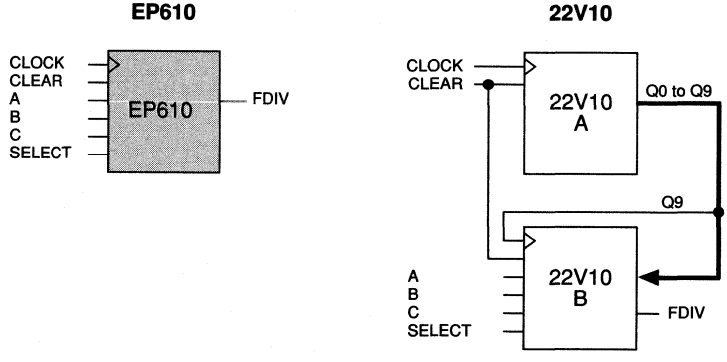
This frequency divider accepts a frequency, divides it by powers of two (2^1, 2^2, 2^3, ...), then produces a selected output frequency based upon frequency-select inputs. Each of the three cascaded frequency dividers receives a Clock from either an input or the previous frequency divider. The divider frequencies are connected to two 74157 multiplexers, each of which routes a selected Clock—defined by the select address (A, B, C)—to its output. Finally, the 2-to-1 multiplexer chooses a single frequency.

The EP610 Solution

The EP610 EPLD is a better choice than the 22V10 device, both for hardware implementation and for design entry.

The EP610 EPLD easily integrates the programmable frequency divider into a single device to produce the selected frequency FDIV (see Figure 3). On the other hand, the 22V10 device is not well suited to this application. Since it contains only 10 macrocells, you need two 22V10 devices to implement the 12-bit counter. Device A must function as a 10-bit counter; its outputs Q0 to Q9 must supply part of the input to Device B. Device B is clocked by Q9. Thus, Devices A and B form a ripple counter to create the two most significant bits of the 12-bit counter function. Frequency-select inputs to Device B control the Clock output multiplexer that ultimately produces FDIV.

Figure 3. EP610 vs. 22V10 as Programmable Frequency Divider



You enter, compile, verify, and program the logic design for the EP610 EPLD with MAX+PLUS II development software. MAX+PLUS II provides over 300 TTL and custom macrofunctions that speed up design entry for designs such as this programmable frequency divider. The MAX+PLUS II Compiler automatically fits the design into the EP610, and produces a standard Programmer Object File (.POF) and an optional JEDEC File (.JED) to program the device. The Compiler automatically performs logic reduction, configures the data paths within the macrocells, and chooses

the best flipflop configuration (D, T, JK, or SR). The result is a working design produced quickly and efficiently.

The 22V10 design is entered with the Texas Instruments Prologic PLD compiler (shown in Figure 4). Prologic is a text-based entry language; therefore, each counter and multiplexer must be expressed as a separate Boolean function. This task is time-consuming and error-prone.

Conclusion

Together, the EP610 EPLD and MAX+PLUS II software offer the ideal solution for implementing a programmable frequency divider in a single device.

Figure 4. Frequency Divider File Created with Prologic (Part 1 of 4)

```
Listing 1:  Device A


title {  Device:          PAL22V10
    Application: 12 Bit Programmable Frequency Divider:  Device A
    Source:      Designer Name         9/89   }

include   p22v10:     /* specify that target device is PAL22V10 */

define   CLK = pin1  :     /* define input pins */
define   CLR = pin2  :

define   Q0  = pin14 :     /* define output pins */
define   Q1  = pin15 :
define   Q2  = pin16 :
define   Q3  = pin17 :
define   Q4  = pin18 :
define   Q5  = pin19 :
define   Q6  = pin20 :
define   Q7  = pin23 :
define   Q8  = pin22 :
define   Q9  = pin21 :

/* define equations to implement lower 10 bits of counter */

Q0.d =   !(Q0.q) & !CLR ;

Q1.d =   (!Q0.q & Q1.q | Q0.q & Q1.q) & !CLR ;

Q2.d =   (!Q0.q & Q2.q |!Q1.q & Q2.q | Q0.q & Q1.q & !Q2.q) & !CLR ;

Q3.d =   (!Q0.q & Q3.q
         |!Q1.q & Q3.q
         |!Q2.q & Q3.q
         | Q0.q & Q1.q & Q2.q & !Q3.q) & !CLR ;

Q4.d =   (!Q0.q & Q4.q
         |!Q1.q & Q4.q
         |!Q2.q & Q4.q
         |!Q3.q & Q4.q
         | Q0.q & Q1.q & Q2.q & Q3.q & !Q4.q) & !CLR ;
```

Figure 4. Frequency Divider File Created with Prologic (Part 2 of 4)

```
Q5.d =   ( !Q0.q & Q5.q
         | !Q1.q & Q5.q
         | !Q2.q & Q5.q
         | !Q3.q & Q5.q
         | !Q4.q & Q5.q
         |  Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & !Q5.q) & !CLR ;

Q6.d =   ( !Q0.q & Q6.q
         | !Q1.q & Q6.q
         | !Q2.q & Q6.q
         | !Q4.q & Q6.q
         | !Q3.q & Q6.q
         | !Q5.q & Q6.q
         |  Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q & !Q6.q) & !CLR ;

Q7.d =   ( !Q0.q & Q7.q
         | !Q1.q & Q7.q
         | !Q2.q & Q7.q
         | !Q3.q & Q7.q
         | !Q4.q & Q7.q
         | !Q5.q & Q7.q
         | !Q6.q & Q7.q
         |  Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q & Q6.q & !Q7.q) & !CLR ;

Q8.d =   ( !Q0.q & Q8.q
         | !Q1.q & Q8.q
         | !Q2.q & Q8.q
         | !Q4.q & Q8.q
         | !Q3.q & Q8.q
         | !Q5.q & Q8.q
         | !Q6.q & Q8.q
         | !Q7.q & Q8.q
         |  Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q & Q6.q & Q7.q & !Q8.q) & !CLR ;

Q9.d =   ( !Q0.q & Q9.q
         | !Q1.q & Q9.q
         | !Q2.q & Q9.q
         | !Q4.q & Q9.q
         | !Q3.q & Q9.q
         | !Q5.q & Q9.q
         | !Q6.q & Q9.q
         | !Q7.q & Q9.q
         | !Q8.q & Q9.q
         | !Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q & Q6.q & Q7.q & Q8.q & !Q9.q) & !CLR ;

/* permanently enable all counter outputs */
Q0.oe = 1;     Q1.oe = 1;     Q2.oe = 1;     Q3.oe = 1;     Q4.oe = 1;
Q5.oe = 1;     Q6.oe = 1;     Q7.oe = 1;     Q8.oe =1;      Q9.oe = 1;

/* define outputs as active high */
Q0 = q;        Q1 = q;        Q2 = q;        Q3 = q;        Q4 = q;
Q5 = q;        Q6 = q;        Q7 = q;        Q8 = q;        Q9 = q;
```

Figure 4. Frequency Divider File Created with Prologic (Part 3 of 4)

```
/* define some test vectors to verify that counter is working properly */
test_vectors { /*CLK      CLR      Q3       Q2       Q1       Q0              */
               pin1     pin2     pin17    pin16    pin15    pin14;
               C        1        L        L        L        L    ;  /*RESET*/
               C        1        L        L        L        L    ;  /*RESET*/
               C        0        L        L        L        H    ;  /*  1  */
               C        0        L        L        H        L    ;  /*  2  */
               C        0        L        L        H        H    ;  /*  3  */
               C        0        L        H        L        L    ;  /*  4  */
               C        0        L        H        L        H    ;  /*  5  */
               C        0        L        H        H        L    ;  /*  6  */
               C        0        L        H        H        H    ;  /*  7  */
               C        0        H        L        L        L    ;  /*  8  */
               C        0        H        L        L        H    ;  /*  9  */
               C        0        H        L        H        L    ;  /*  10 */
               C        0        H        L        H        H    ;  /*  11 */
               C        0        H        H        L        L    ;  /*  12 */
               C        0        H        H        L        H    ;  /*  13 */
               C        0        H        H        H        L    ;  /*  14 */
               C        0        H        H        H        H    ;  /*  15 */
               C        0        L        L        L        L    ;  /*  0  */
               C        1        L        L        L        L    ;  /*RESET*/
```

Listing 2: Device B

```
title {  Device:          PAL22V10
         Application:     12 Bit Programmable Frequency Divider:  Device B
                          2MSB and Multiplexing Control
         Source:          9/89   }

include  p22v10;          /* specify that target device is PAL22V10  */

define   CLK = pin1 ;     /* clock input is from Q9 on device A      */
define   CLR = pin2 ;     /* clear goes to pin 2 on both devices     */

define   A = pin3 ;       /* select inputs for multiplexing outputs  */
define   B = pin4 ;       /* select Input   Q Output                 */
define   C = pin5 ;       /* ABCD  = 0      divided by 2             */
define   D = pin6 ;       /*     1        4       */
                 /*     2        8 etc.     */

define   Q0 = pin7  ;     /* define counter inputs from device A     */
define   Q1 = pin8  ;
define   Q2 = pin9  ;
define   Q3 = pin10 ;
define   Q4 = pin11 ;
define   Q5 = pin13 ;
define   Q6 = pin14 ;
define   Q7 = pin15 ;
define   Q8 = pin16 ;
define   Q9 = pin17 ;

define   Q10 = pin19 ;         /* define 2 MSB of 12 bit counter      */
define   Q11 = pin20 ;

define   FDIV = pin18 ;        /* divided output                      */

/* define equations to implement 2 MSB of counter                     */

    Q10.d = !(Q10.q) & !CLR ;
    Q11.d = (!Q10.q & Q11.q | Q10.q & Q11.q) & !CLR ;
```

Figure 4. Frequency Divider File Created with Prologic (Part 4 of 4)

```
/* define equations to control multiplexing of outputs */

    FDIV     =   Q0       &  !A  &  !B  &  !C  &  !D
              |   Q1       &  !A  &  !B  &  !C  &   D
              |   Q2       &  !A  &  !B  &   C  &  !D
              |   Q3       &  !A  &  !B  &   C  &   D
              |   Q4       &  !A  &   B  &  !C  &  !D
              |   Q5       &  !A  &   B  &  !C  &   D
              |   Q6       &  !A  &   B  &   C  &  !D
              |   Q7       &  !A  &   B  &   C  &   D
              |   Q8       &   A  &  !B  &  !C  &  !D
              |   Q9       &   A  &  !B  &  !C  &   D
              |   Q10.q    &   A  &  !B  &   C  &  !D    /*".q" used on terms that */
              |   Q11.q    &   A  &  !B  &   C  &   D    /* are internal feedbacks */

/* permanently enable all outputs */
 Q10.oe = 1; Q11.oe = 1;  FDIV.oe = 1;

/* define outputs as active high */
 Q10 = q;    Q11 = q;
```

DMA Controller with the EPM5064 MAX EPLD

April 1992, ver. 2

Application Brief 84

Introduction

A direct memory access (DMA) controller increases the performance of a peripheral subsystem by coordinating data transfer between peripheral and subsystem memory. A DMA controller is useful when a design requires data transfer rates that are too fast for a microprocessor. DMA can be used by disk-drive controllers to quickly transfer large blocks of data, by high-speed serial subsystems to maintain communications link bit rates, and by dedicated graphics processors that update images stored within video memory.

You can implement DMA controllers in several ways. Standard off-the-shelf DMA controller devices are available (e.g., AM9517A and AM8237A devices), which provide a single-chip, application-specific solution. However, these devices may not be suitable due to speed, power consumption, or protocol requirements. For example, the AM9517A device transfers data at 2.5 Mbytes per second and requires a specific set of control commands that is normally not used with standard DMA controllers.

If higher performance or custom functions are required, you can implement the logic for a DMA controller with discrete TTL devices (e.g., the 7400-series). This approach supports custom DMA protocols and tailored performance, but has a number of drawbacks: TTL logic consumes high power, requires a large amount of PC board space, and the high component count and power dissipation reduce overall system reliability.

EPLDs offer the ideal solution, integrating the advantages of a standard off-the-shelf DMA controller device with the higher performance and flexibility of the TTL device approach. This application brief describes a DMA controller implemented with an Altera EPM5064 EPLD that achieves data transfer rates of up to 20 megaword per second between peripheral and subsystem memory. This design (designs are called "projects" in MAX+PLUS II) consists of Graphic Design Files (.GDF) and Text Design Files (.TDF) entered with the Altera MAX+PLUS II development software.

EPM5064 Overview

The EPM5064 EPLD is a user-configurable, high-performance, high-density MAX 5000 device. It offers a fast 25-ns input-to-output combinatorial delay and 50-MHz 16-bit counter frequencies. This EPLD is offered in 44-pin windowed ceramic and plastic J-lead chip carrier packages that have 8 dedicated inputs and 28 bidirectional I/O pins. Commercial, industrial, and military temperature-range versions are available.

The EPM5064 EPLD contains 64 macrocells, each with a register that can be programmed for D, T, JK, SR, or flow-through latch operation, or bypassed entirely for purely combinatorial functions. All macrocell registers also include asynchronous Clear and Preset controls. The macrocells are grouped into 4 Logic Array Blocks (LABs), each containing 16 macrocells and 32 shareable expanders. Shareable expanders are freely allocatable product terms that can be used and shared by any macrocell within the LAB.

A Programmable Interconnect Array (PIA) routes signals between the various LABs. The PIA, which is fed by the 28 I/O pins and 64 macrocell feedbacks, provides the resources necessary to ensure 100% signal routability. A fixed interconnect delay across the PIA eliminates skew and provides consistent, predictable performance. For additional information on device architecture, refer to the *EPM5016 to EPM5192 EPLDs: High-Speed, High-Density MAX 5000 Devices Data Sheet* in the Altera 1992 ***Data Book***.

DMA Process

Figure 1 shows a DMA controller that supports the microprocessor unit, a peripheral, and the memory block of a peripheral subsystem.

Figure 1. Sample Peripheral Subsystem with an EPM5064 DMA Controller

A peripheral subsystem can be designed for any custom bus protocol with a custom EPLD DMA Controller.

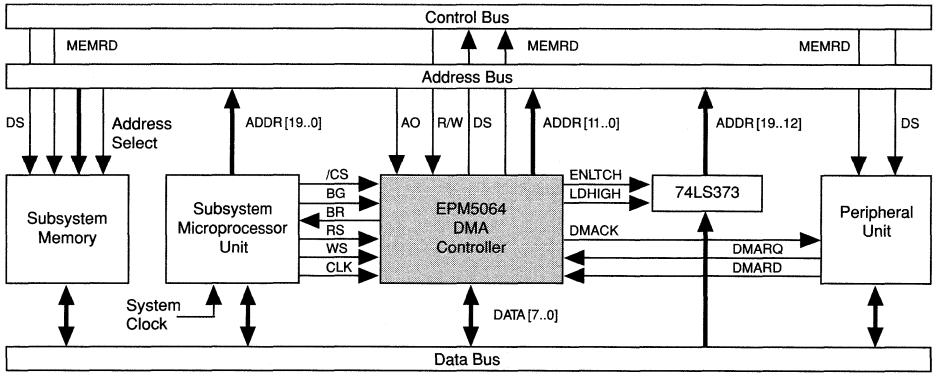
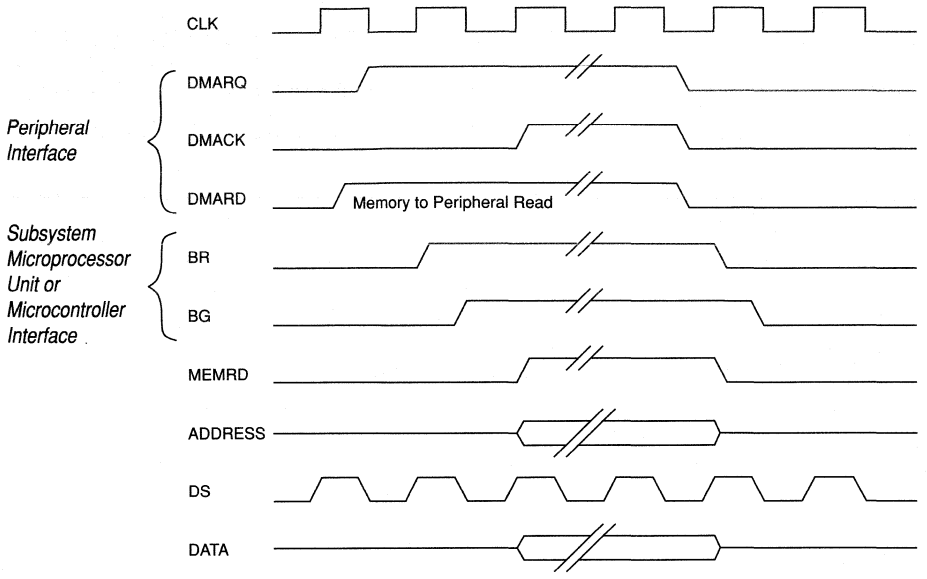


Figure 2 shows the timing associated with the DMA control signals. A custom-designed DMA controller implemented in an EPM5064 EPLD can improve performance of the peripheral subsystem by providing faster data transfers than are possible with the microprocessor alone. Using a DMA controller also frees the microprocessor to perform other tasks.

To start the DMA process, the microprocessor must first select and initialize the DMA controller; then it must write the starting and ending addresses

Figure 2. Generic DMA Timing



and the control information to the controller. The control data indicates whether the transfer addresses should be incremented or decremented.

After initialization, the peripheral starts the DMA transfer by asserting the DMARQ (DMA request) input to the controller. DMARD (DMA read), another input from the peripheral, indicates the direction of the DMA transfer. When DMARD is high, the current DMA cycle is a memory-read cycle; when it is low, it is a memory-write cycle.

After DMARQ and DMARD are asserted, the DMA controller asserts the BR (bus request) input to the microprocessor. The microprocessor then drives BG (bus grant) high and releases control of the buses. The DMA controller asserts DMACK (DMA acknowledge) to inform the peripheral that it controls the buses, and the DMA cycle can begin.

To perform the DMA transfer, the control and address buses must transfer data between peripheral and subsystem memory. DS (data strobe) is a control bus signal, driven by the DMA controller, that strobes the data between the peripheral and subsystem memory during the DMA transfer. The DMA controller simultaneously drives MEMRD (memory read) with the same logic level as DMARD and the address bus with the desired memory address. MEMRD indicates whether a transfer is a read (MEMRD high) or write (MEMRD low) operation.

On each Clock cycle during the DMA transfer, the peripheral writes a new data word on the bus or reads a data word off the bus until all data words have been transferred. On every rising edge of DS, the DMA controller drives a new address, thus transferring a single data word. This process is called the burst mode of DMA transfer.

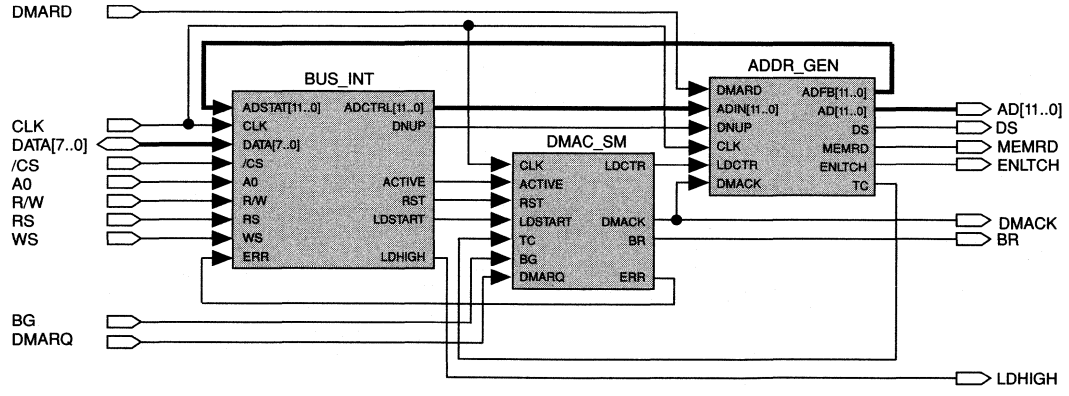
After the transfer is complete, the DMA controller bus signals are tri-stated, and bus control is returned to the microprocessor. The controller negates BR to inform the microprocessor that it is finished with the bus. Simultaneously, the address (A[11..0]), DS, and MEMRD signals are tri-stated. When the microprocessor regains bus control, it resumes execution from its previous state.

EPM5064 DMA Controller

You can use an EPM5064 EPLD and an external 74373 byte-wide address latch to implement the DMA controller subsystem. This configuration supports a 20-bit address bus; wider buses can be supported with additional external address latches or by using other Altera EPLDs that have more I/O (e.g., the EPM7096 device). The 74373 latches the upper 8 bits of the DMA transfer address from the subsystem data bus, while the lower 12 bits are generated by the EPM5064 EPLD. The lower 12 bits allow DMA transfers of up to 4048 words without processor intervention. The EPM5064 EPLD controls the external address latch from the LDHIGH (load high-order address) and ENLTCH (enable address latch) signals. LDHIGH loads the upper address bits from the data bus into the address latch. ENLTCH enables the latch to drive its contents onto the address bus.

Figure 3 shows DMA_C.GDF, a DMA controller implemented in an EPM5064 EPLD. The hierarchical design consists of three basic functional blocks: a bus interface unit (BUS_INT), a DMA control state machine

Figure 3. DMA Controller Schematic (DMA_C.GDF)

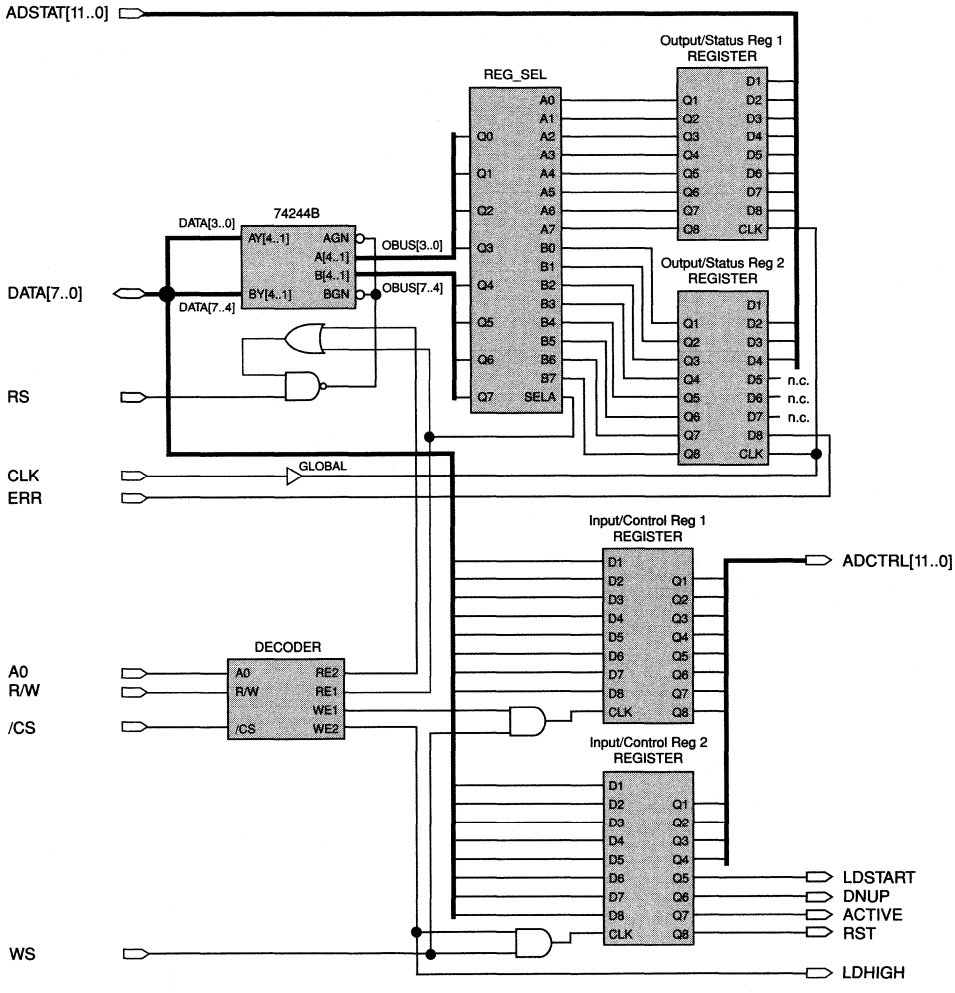


(DMAC_SM), and an address generator (ADDR_GEN). Both BUS_INT and ADDR_GEN are also hierarchical and contain other lower-level functions.

Bus Interface

Figure 4 shows BUS_INT.GDF, the portion of DMA_C.GDF that implements the bus interface unit. DECODER, shown in the lower left corner of BUS_INT.GDF, uses microprocessor signals to control data flow within the bus interface unit. The TTL 74244B macrofunction (which is functionally equivalent to the 74244 octal buffer) is an I/O buffer that supports the bidirectional data bus DATA[7..0]. The input/control registers receive all inputs from the data bus, and the output/status registers store information

Figure 4. Bus Interface Unit Schematic (BUS_INT.GDF)



to be read onto the data bus. REG_SEL selects data from one of the two output/status registers to appear at the I/O buffer. The bus interface unit supports the following operations:

- EPM5064 DMA controller initialization
- Write operation
- Read operation

EPM5064 DMA Controller Initialization

The subsystem microprocessor communicates with the bus interface unit to select and initialize the EPM5064 EPLD. The microprocessor must first address the DMA controller by driving /CS (chip select) low. A write operation is started when A0 and R/W select one of the input/control registers and WS (write strobe) is asserted. For example, when A0 is low, R/W is low, and WS has a rising edge, the eight bits of the data bus are written into input/control Register 1.

Table 1 shows the bus interface decoding scheme for the microprocessor signals.

Table 1. Bus Interface Decoding Scheme

RS	WS	/CS	R/W	A0	Action
⌐	L	L	H	H	Read output/status Register 1
⌐	L	L	H	L	Read output/status Register 2
L	⌐	L	L	H	Write input control Register 1
L	⌐	L	L	L	Write input control Register 2

Write Operation

After selecting the proper register and asserting WS, the microprocessor writes the least significant eight bits of the starting address, A[7..0], into input/control Register 1. Then A[11..8] and the microprocessor control signals ACTIVE (DMA active), LDSTART (load starting address), DNUP (down up), and RST (reset) are written into input/control Register 2. ACTIVE and LDSTART, the input signals to the state machine, indicate that the starting address must be transferred to the ADDR_GEN block. The DNUP signal specifies whether the DMA address will be decremented (DNUP high) or incremented (DNUP low). RST, the final microprocessor signal, reverts the DMA control state machine to the initial state.

Read Operation

During the DMA process, the microprocessor may read either one of the output/status registers for the current DMA address or DMA controller status. The least significant eight bits of the DMA address, stored in output/status Register 1, is fed by the ADDR_GEN function. Output/status Register 2 receives the most significant nibble (i.e., most significant four bits), A[11..8], and the status signal, ERR (error). The ERR signal, which comes from the DMA control state machine, indicates an error condition during a DMA cycle. The REG_SEL function selects which output/status register feeds the I/O buffer. The I/O buffer is controlled by RS (read strobe) and provides the tri-stating necessary to ensure proper bidirectional operation.

DMA Control State Machine

Figure 5 shows the state diagram for DMAC_SM, the DMA control state machine. The state machine consists of five states: INIT, LOAD, START, TRANSFER, and ERROR.

Figure 5. DMA Control State Machine

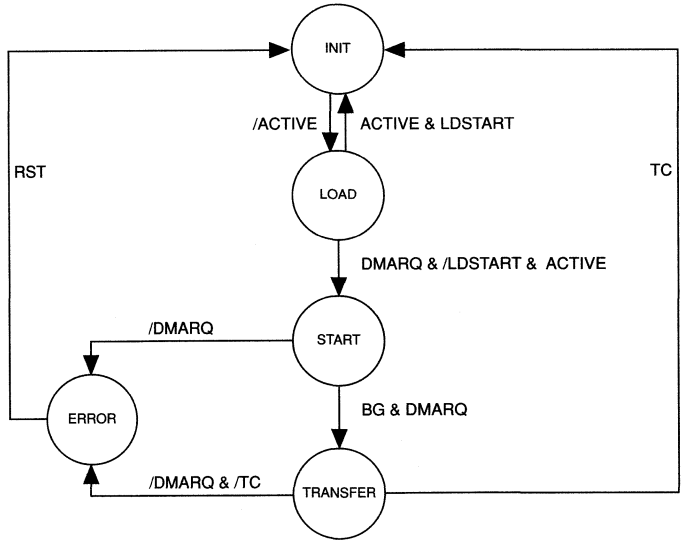


Figure 6 shows the DMAC_SM function implemented in an Altera Hardware Description Language (AHDL) TDF. AHDL supports Boolean equations, truth tables, If Statements, and Case Statements that facilitate state machine design. DMAC_SM.TDF uses an If Statement to describe the state transitions.

Figure 6. DMA_SM.TDF (Part 1 of 2)

```
TITLE   "EPM5064 DMA Control State Machine";

SUBDESIGN   DMAC_SM
(
%  State Machine Inputs    %
    CLK     : INPUT;        % subsystem Clock                  %
    BG      : INPUT;        % bus grant                        %
    DMARQ   : INPUT;        % DMA request                      %
    ACTIVE  : INPUT;        % activate initialization sequence %
    RST     : INPUT;        % reset error condition            %
    LDSTART : INPUT;        % load starting address            %
    tc      : INPUT;        % terminal count                   %

%  State Machine Outputs  %
    ERR     : OUTPUT;       % error condition %
    DMACK   : OUTPUT;       % DMA acknowledge %
    BR      : OUTPUT;       % bus request     %
    LDCTR   : OUTPUT;       % load counter    %
)

VARIABLE
    SYSCLK  : NODE;                                    % declare system Clock      %
    CYCLE   : MACHINE OF BITS  (q[3..0])               % define machine cycle with %
              WITH STATES      (INIT      = B"0000", % four state bits q[3..0]   %
                               LOAD       = B"0001",
                               START      = B"1000",
                               TRANSFER   = B"1100",
                               ERROR      = B"0010" );

BEGIN
    SYSCLK          = GLOBAL(CLOCK);   % use global Clock          %
    CYCLE.CLK       = SYSCLK;          % state machine Clock       %
    CYCLE.RESET     = RST;             % state machine Reset       %
    BR              = q3;              % use state bits as outputs %
    DMACK           = q2;
    ERR             = q1;
    LDCTR           = q0;

% define state transitions %
    CASE CYCLE IS
        WHEN INIT =>
            IF (ACTIVE & LDSTART) THEN
                CYCLE = LOAD;
            END IF;

        WHEN LOAD =>
            IF (!ACTIVE) THEN
                CYCLE = INIT;
            ELSIF (DMARQ & !LDSTART) THEN
                CYCLE = START;
            END IF;
```

Figure 6. DMA_SM.TDF (Part 2 of 2)

```
        WHEN START =>
            IF (!DMARQ) THEN
                CYCLE = ERROR;
            ELSIF (BG) THEN
                CYCLE = TRANSFER;
            END IF;

        WHEN TRANSFER =>
            IF (TC) THEN
                CYCLE = INIT;
            ELSIF (!DMARQ) THEN
                CYCLE = ERROR;
            END IF;

        WHEN ERROR =>
            IF (RST) THEN
                CYCLE = INIT;
            END IF;

    END CASE;
END;
```


INIT

DMAC_SM is in the INIT state at power-up. While in this state, the microprocessor enables the DMA controller. After the ACTIVE and LDSTART inputs are asserted, the machine proceeds to the LOAD state.

LOAD

When the LOAD state is entered, LDCTR (load counter) drives the starting address into ADDR_CNT. The microprocessor deasserts LDSTART after it has written the ending address to the DMA controller. The peripheral then activates DMARQ. When LDSTART is low and DMARQ is high, the machine enters the START state. DMARQ must remain asserted until the transfer is complete.

START

In the START state, DMAC_SM outputs BR to acquire control of the subsystem buses from the microprocessor. The microprocessor then sends BG to the state machine in response to BR. If the peripheral deasserts DMARQ while in this state, the machine makes a transition to the ERROR state. Otherwise, DMAC_SM proceeds to the TRANSFER state when BG goes high.

TRANSFER

DMACK is asserted in the TRANSFER state, indicating that the ADDR_GEN block will begin generating the DMA transfer addresses. The state machine is reset after receiving the TC (terminal count) signal from ADDR_GEN. If DMARQ is deasserted before TC is high, DMAC_SM again enters the ERROR state. This condition may occur, for example, if a system power failure disrupts the DMA process.

ERROR

The ERROR state occurs when a DMARQ low input is received in the START or TRANSFER state. The RST input of the machine must be asserted to clear the error condition. The state machine then returns to the INIT state.

DMA Address Generation

Figure 7 shows the address generator ADDR_GEN.GDF, which contains three lower-level functions: ADDR_CNT, END_COMP, and OUT_BUF. ADDR_CNT is a loadable counter that generates the lower 12 bits of the DMA transfer addresses. END_COMP compares the current address with the ending address and terminates the DMA transfer when the last address is reached. OUT_BUF enables the current DMA transfer address onto the address bus.

The address range is determined by the starting and ending addresses read from the microprocessor. Depending on the system convention, the addresses may be generated in increasing or decreasing order. With 12 bits, variable address ranges up to 4048 words are achievable. When the last address has been reached, the state machine is reset and the DMA controller can be initialized for a new DMA transfer.

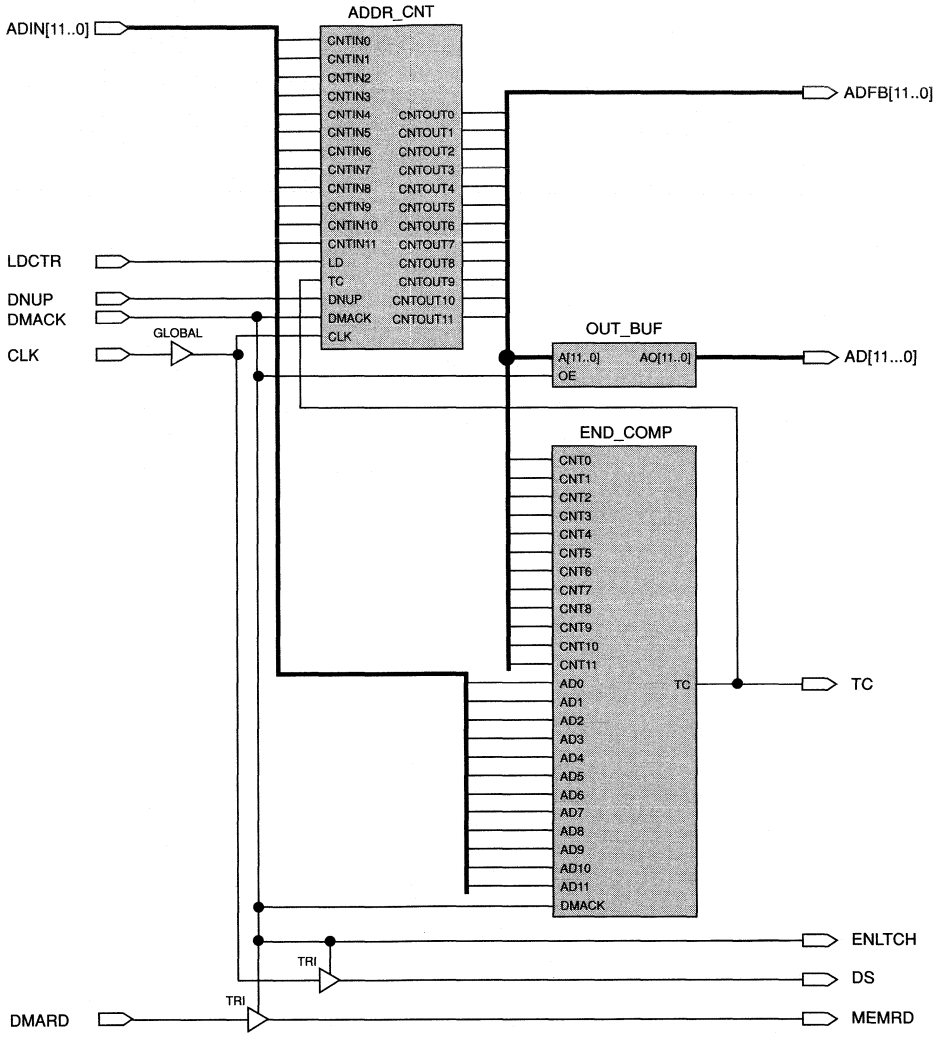
Control Signals

LDCTR, DMACK, and DNUP are control inputs from the DMA control state machine. When LDCTR is asserted and CLK has a rising edge, the 12-bit starting address is loaded into ADDR_CNT. The ADDR_CNT counter begins counting when LDCTR is deasserted and DMACK is asserted. DNUP is from the bus interface unit and determines the count direction. When DNUP is high, the current address is incremented; when DNUP is low, the address is decremented.

Output Signals

MEMRD, ENLTCH, DS, and TC are signal outputs of ADDR_GEN.GDF. The MEMRD signal, which indicates the direction of DMA transfer, is driven onto the control bus when DMA begins. ENLTCH enables the external latch driving the high-order address onto the address bus. DS provides the gating signal that the peripheral uses to strobe the next valid address. During a DMA transfer, DS follows the subsystem Clock; otherwise, it is

Figure 7. Address Generator Schematic (ADDR_GEN.GDF)

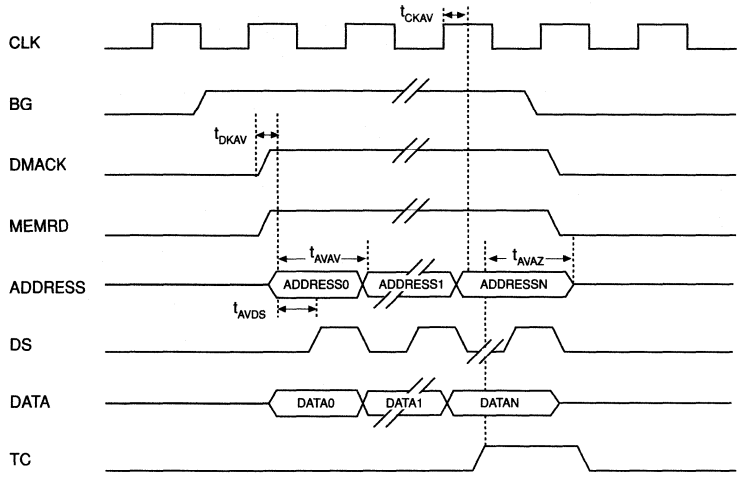


tri-stated. When the required addresses have been generated, the TC signal becomes active, signifying that the DMA transfer is complete.

Performance

Figure 8 shows the critical timing parameters for the DMA controller DMA_C: t_{DKAV} (DMACK to first valid address), t_{AVAZ} (last address valid to next address valid), t_{AVAV} (address valid to next address valid), t_{CKAV} (Clock to address valid), and t_{AVDS} (address valid to data strobe). The table inFigure 8 also lists parameter values. The speed at which the EPM5064

Figure 8. Critical Timing for DMA Controller DMA_C



Symbol	Parameter	EPM5064 EPLD
t_{DKAV}	DMACK to first address valid	28 ns
t_{AVAZ}	Last address valid to bus tri-state	33 ns
t_{AVAV}	Address valid to next address valid	50 ns
t_{CKAV}	Clock to address valid	15 ns
t_{ADVS}	Address valid to data strobe	12 ns

EPLD can generate successive addresses is shown by t_{AVAV}. This parameter determines the maximum Clock rate at which the DMA controller can operate (20 MHz).

Conclusion

The EPM5064 EPLD is a generic EPLD that you can use to implement complete peripheral interfaces. The integration density, power, speed, and flexibility of the EPM5064 EPLD make it an excellent choice for a peripheral subsystem DMA controller.

The DMA controller described in this application brief can transfer data at 20 megawords per second. It fits in a single EPM5064 EPLD, which requires only 1/2 square inch of board space. The design files for this EPM5064 DMA controller are available from the Altera electronic bulletin board service (BBS) by calling (408) 249-1100.

Designing & Simulating Bidirectional Buses

with MAX+PLUS II Software

April 1992, ver. 1

Application Brief 86

Introduction

Digital designs often contain sets of related signals that originate at a single device and are routed through the circuit as a group, i.e., bus. Buses are commonly used in microprocessor-based systems, where the microprocessor issues address and data information that is conveyed to multiple devices in the system. Since microprocessors and their peripheral devices both send and receive data, buses are commonly bidirectional.

Typically, a device's data bus is shared by several other devices in the system. The bidirectional pins of each device must be able to switch to a high-impedance state to disconnect from the bus when pins are not driving out. Since high impedance is considered a third logic state, it is referred to as "tri-state."

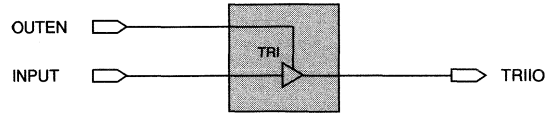
With the Altera MAX+PLUS II development system, you can create and simulate designs (called "projects" in MAX+PLUS II) that contain bidirectional buses. In fact, the MAX+PLUS II TTL MacroFunction Library contains over 25 macrofunctions designed for bus applications. To implement a project that contains a bidirectional bus, you can use Classic, MAX 5000, MAX 7000, and Synchronous Timing Generator (STG) EPLDs, all of which have the density and architectural flexibility required for bidirectional bus logic.

This application brief describes how to design, simulate, and implement logic that contains bidirectional buses with Altera EPLDs and software. Familiarity with Altera EPLD architecture and MAX+PLUS II software is assumed. For complete information on these topics, refer to individual EPLD datasheets and MAX+PLUS II Help.

Tri-State Buffer Design & Simulation

You can incorporate tri-state buffers into both Graphic Design Files (.GDF) and Text Design Files (.TDF). The GDF shown in Figure 1 contains a single tri-state buffer primitive (TRI). Note that the output of a tri-state buffer must be connected directly to either an output or bidirectional pin.

Figure 1. Sample GDF with Tri-State Buffer



You can also implement the same tri-state buffer function in a TDF with either an in-line reference or variable declaration. See Figure 2.

Figure 2. Sample TDFs with Tri-State Buffer

In-line Reference

```
SUBDESIGN "tri-state"
(
    outen, inputa       : INPUT;
    triioa              : OUTPUT;
)

% In-line Reference %
BEGIN
    triioa = TRI(inputa, outen);
END;
```

Variable Declaration

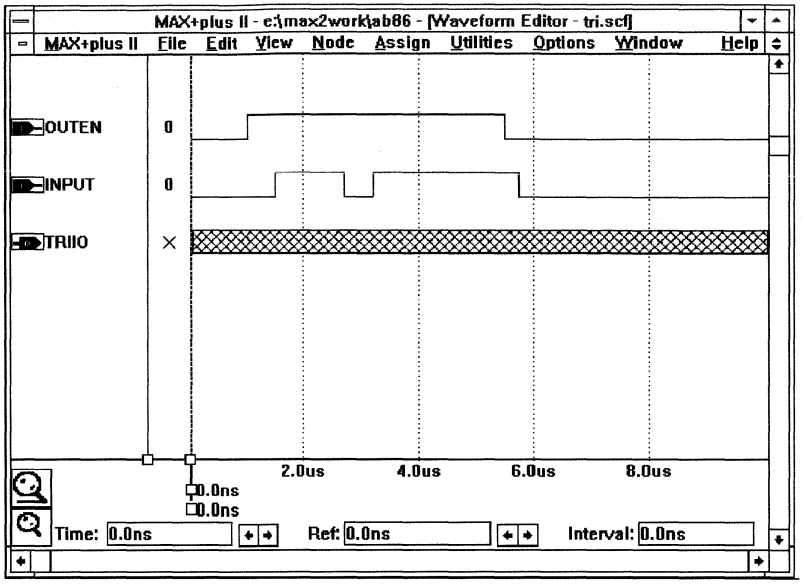
```
SUBDESIGN "tri-state"
(
    outen, inputb       : INPUT;
    triiob              : OUTPUT;
)

VARIABLE
    trib                : tri;

% Variable Declaration %
BEGIN
    trib.in = inputb;
    trib.oe = outen;
    triiob  = trib.out;
END;
```

After you enter and successfully compile a project, you can simulate it with the MAX+PLUS II Simulator. The Simulator accepts either graphical waveform Simulator Channel Files (.SCF) or ASCII Vector Files (.VEC) to provide the input vectors necessary to drive simulation.

Use the MAX+PLUS II Waveform Editor to create an SCF that contains the waveform stimulus for the circuit. With the **Create Default Channel** command, you can create an SCF with all input, output, and buried nodes in the compiled project. Figure 3 shows an SCF for the tri-state buffer shown in Figures 1 and 2. Before you simulate the project, `TRIIO` is shown as an unknown (`X`) logic level.

Figure 3. Simulator Channel File for Sample Tri-State Buffer

TRIIO is shown as an unknown logic (X) level until you simulate the project.



To create an ASCII Vector File with test vectors that consists of logical 1's and 0's, use the MAX+PLUS II Text Editor or any other ASCII text editor. Figure 4 shows a Vector File for the tri-state buffer shown in Figures 1 and 2.

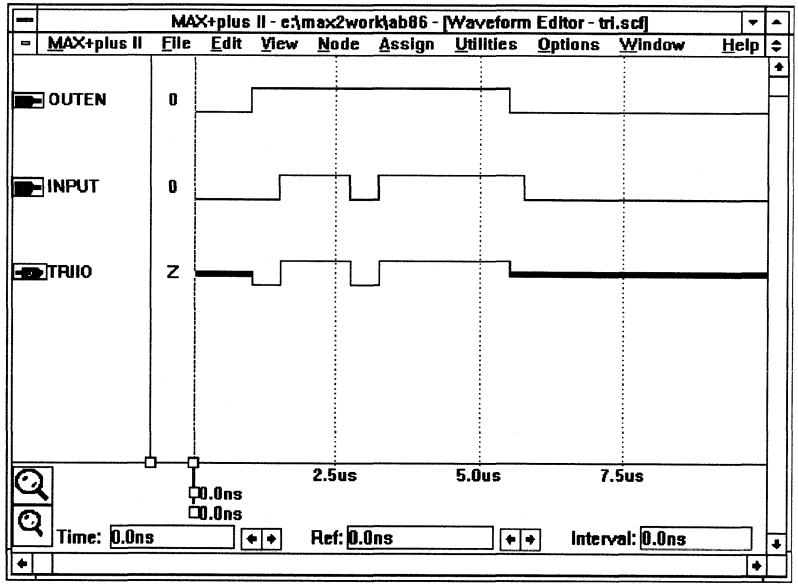
Figure 4. Vector File for Sample Tri-State Buffer

```
UNIT us;
START 0 us;
STOP 10 us;
INPUTS OUTEN INPUT;
PATTERN

0.00 > 0 0
1.00 > 1 0
1.50 > 1 1
2.75 > 1 0
3.25 > 1 1
5.50 > 0 1
5.75 > 0 0;

OUTPUTS TRIIO;
```

Figure 5 shows the SCF generated by simulating the tri-state buffer project. Whenever the OUTEN signal is low, the tri-state buffer's output shifts to the high-impedance state (indicated by the waveform with the thick line). Otherwise, the buffer's output is identical to its input.

Figure 5. Simulation Results for Sample Tri-State Buffer

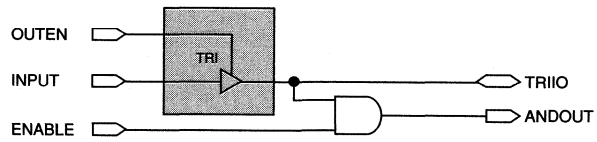
High-impedance state for TRIIO is represented by a thick line.



Bidirectional Signal Design & Simulation

Figure 6 illustrates how you can easily create a bidirectional circuit with a tri-state buffer. In this application, the output pin in Figure 1 is replaced with a bidirectional pin called TRIIO. The output of the tri-state buffer is connected to one input of an AND2 gate; the other AND2 input is driven by the ENABLE signal. Finally, an output pin called ANDOUT is connected to the AND2 output.

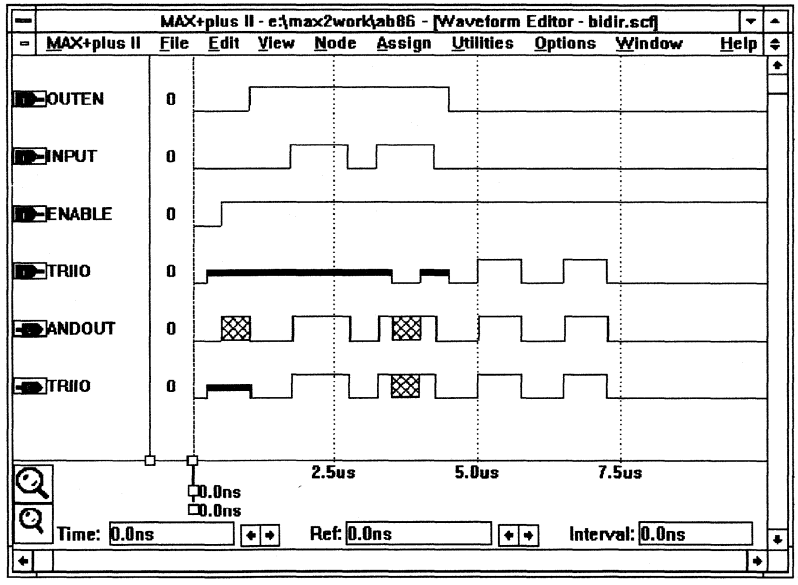
Figure 6. Sample GDF with Bidirectional Pin



A bidirectional pin must appear as both an input and an output in the SCF or Vector File used for simulation. The **Create Default Channel** command automatically inserts both an input and an output node.

Whenever a bidirectional pin is driving out, the input must be in a high-impedance (Z) state; otherwise bus contention occurs. Figure 7 shows the simulation waveforms for the project shown in Figure 6: TRIIO (input) and ANDOUT both track the input waveform, except when the TRIIO (input) changes to a logic low at 3.5 µs.

Figure 7. Simulation Results for Sample Bidirectional Pin

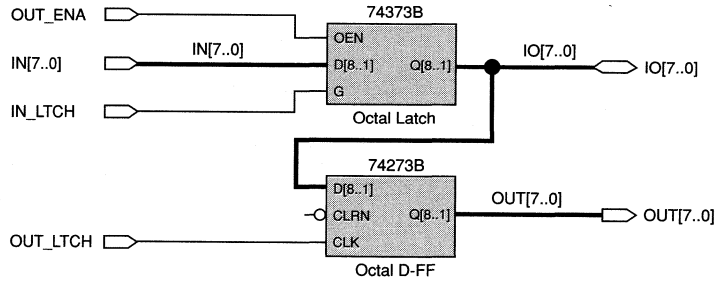


The transition of TRIIO (input) causes bus contention, which the MAX+PLUS II Simulator interprets as an unknown (X) logic level. This unknown value, in turn, is propagated to TRIIO (output) and ANDOUT. To avoid bus contention, configure TRIIO (input) by changing the Output Enable signal OUTEN to 0, TRIIO (output) and ANDOUT are then both identical to TRIIO (input).

Bidirectional Bus Design & Simulation

MAX+PLUS II software includes macrofunctions specifically created to support bus applications. Figure 8 shows a GDF that contains two such macrofunctions: a 74373B octal latch and a 74273B octal D flipflop.

Figure 8. Sample GDF with Bidirectional Bus



The buses (thick lines) take the place of the eight discrete input and output lines, so that the project can be implemented quickly and neatly.

The eight inputs IN[7..0] connect to the 74373B octal D latch. The inputs are latched to the 74373B macrofunction on the rising edge of IN_LTCH. Tri-state control to the bidirectional bus is provided by the OUT_ENA signal. Signals entering via the bidirectional bus IO[7..0] are latched to the 74273B by OUT_LTCH and appear on the output bus OUT[7..0].

Table 1 lists the MAX+PLUS II TTL macrofunctions that are optimized for bus applications.

Table 1. MAX+PLUS II Bus Macrofunctions (Part 1 of 2)

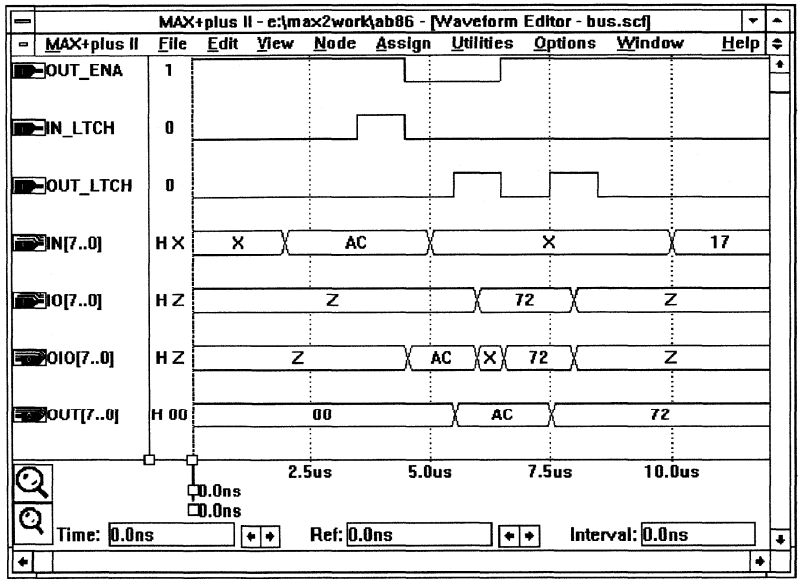
Macrofunction	Description
BARRLSTB	8-Bit Barrel Shifter
MULT4B	4-Bit Parallel Binary Multiplier
16CUDSRB	16-Bit Universal Counter/Shifter Register
8FADDB	8-Bit Full Adder
8MCOMPB	8-Bit Magnitude Comparator
74151B	8-line-to-1-line Multiplexer
74164B	Serial-In Parallel-Out Shift Register
74165B	Parallel Load 8-Bit Shift Register
74174B	Hex D Flipflop with Common Clear
74180B	9-Bit Odd/Even Parity Generator/Checker
74240B	Octal Tri-State Buffer with Inverting Outputs
74241B	Octal Tri-State Buffer

Table 1. MAX+PLUS II Bus Macrofunctions (Part 2 of 2)

74244B	Octal Tri-State Buffer
74273B	Octal D Flipflop with Asynchronous Clear
74280B	9-Bit Odd/Even Parity Generator/Checker
74373B	Octal D-Type Transparent Latch with Output Enable and Tri-State
74374B	Octal D Flipflop with Output Enable and Tri-State
74518B	8-Bit Identity Comparator
74821B	10-Bit D Flipflop with Tri-State Outputs
74822B	10-Bit D Inverting Flipflop with Tri-State Outputs
74823B	9-Bit D Flipflop with Tri-State Outputs
74824B	9-Bit D Inverting Flipflop with Tri-State Outputs
74825B	Octal D Flipflop with Tri-State Outputs
74826B	Octal D Inverting Flipflop with Tri-State Outputs
74841B	10-Bit D Latch with Tri-State Outputs
74842B	10-Bit D Inverting Latch with Tri-State Outputs

The rules for simulating bidirectional nodes also apply to bidirectional buses. The default SCF created by the Waveform Editor automatically contains the input bus `IO[7..0]`. However, since two buses cannot have the same name, the outputs `IO7` through `IO0` are not grouped. Use the **Enter Group** command to group these signals into a bus. Figure 9 shows outputs `IO7` through `IO0` grouped into a bus called `OIO[7..0]`.

Figure 9. Simulation Results for Sample Bidirectional Bus



During simulation, you must ensure that the input bus IN[7..0] is configured as high impedance when the bidirectional bus is driven as an output (OUT_ENA is low). If it is not, logic contention occurs on the output bus OIO[7..0] between 6 µs and 6.5 µs. The tri-state buffers also must be shifted to high impedance when the bidirectional bus IO[7..0] is driven as an input between 6 µs and 8 µs.

Conclusion

You must follow three simple rules to design, simulate, and implement bidirectional bus logic:

1. The output of a tri-state buffer (TRI) must be connected to an output (OUTPUT) or bidirectional (BIDIR) pin primitive.
2. Bidirectional signal names must appear as both inputs and outputs in the SCF or Vector File used for simulation.
3. During simulation, the input of the bidirectional signal must shift to high impedance when the I/O pin operates as an output. Whenever the I/O pin operates as an input, the tri-state buffer must shift to high impedance (i.e., the Output Enable of the tri-state buffer must be disabled).

Understanding these concepts will allow you to quickly and efficiently create bidirectional bus applications for your digital designs.

Troubleshooting Functional Problems in EPLDs

April 1992, ver. 1

Application Brief 87

Introduction

When troubleshooting an EPLD problem, you must first determine whether it is a functional or a programming problem. If you encounter a programming problem (e.g., if MAX+PLUS II generates an error during programming, the software halts during programming, or programming takes an unusually short or long time), refer to *Application Brief 81 (Troubleshooting EPLD Programming Problems)* in this handbook.

This application brief discusses common functional and timing problems and offers some solutions. A troubleshooting checklist is also provided to help you and Altera diagnose the problem.

When Does the Problem Occur?

To troubleshoot a functional EPLD problem, first determine whether the problem occurs immediately after power-up or after successful operation.

If the Problem Occurs Immediately after Power-Up...

If the problem occurs just after you have applied power to the EPLD, try the following solutions:

- If the EPLD is EPROM-based, erase it for one hour and reprogram it. Altera EPROM-based EPLDs must be erased for one hour with UV light with a wavelength of 2,537 Å before they are reprogrammed. The one-hour erasure time assumes a lamp with a 12,000 $\mu W/cm^2$ power rating; lower-power erasers will take longer.

- Check the V_{CC} rise time with an oscilloscope. It should not exceed 50 ms for a Classic EPLD and 10 ms for a MAX 5000, MAX 7000, or Synchronous Timing Generator (STG) EPLD, and it should have a smooth ramp-up. V_{CC} rise times that are too long may cause the EPLD to power up in an incorrect state. Also check for excessive overshoot or ringing/oscillation upon power-up, which can cause latch-up if the minimum or maximum V_{CC} ratings of the EPLD are violated.

- Check the decoupling capacitors. All Altera EPLDs should have a decoupling capacitor (with capacitance from 0.1 to 0.2 µF) across each pair of VCC and GND pins connected directly at the device. Decoupling capacitors isolate the device from VCC and GND signal noise, ensuring proper operating conditions for the EPLD and other devices on the board.

- ❑ Check the ground path. Use multi-layer boards with separate VCC and GND planes when using high-speed EPLDs. Avoid wire-wrapping. When subjected to a rapidly switching current, ground-path inductance can create a voltage that introduces system noise. Refer to *Application Brief 89* (*Minimizing Output Switching Noise in Altera EPLDs*) in this handbook for more information.

- ❑ Check the EPLD pin-out and ensure that all pins are connected according to the Report File (.RPT) for the design. Unused dedicated inputs should be connected to GND. Reserved pins, which must be left unconnected, are associated with buried macrocells that generate signals that are fed back into the EPLD. Since these signals are used by other logic in the device, the EPLD may not operate correctly if the reserved pins are connected to VCC, GND, or any other signals. If you connect some of the reserved output pins, the EPLD may also consume excessive current and become hot.

- ❑ Ensure that the leads of the EPLD are straight and do not touch each other. The corner leads of J-lead packages can become damaged if the devices are not inserted properly into the programming, prototyping, or production sockets.

- ❑ Ensure that the EPLD is fully powered up before you apply any inputs to the device. If the voltages of the input signals exceed V_{CC}, the EPLD may latch up. During latch-up, the EPLD will consume excessive current and become hot, and will not work.

If the Problem Occurs after Successful Operation...

If the problem occurs after the EPLD has been operating successfully for some time, try the following solutions:

- ❑ If the EPLD is in a windowed ceramic package, make sure the window is covered. Because a windowed ceramic EPLD can be erased with UV light, ambient light (e.g., fluorescent and incandescent light as well as sunlight) can erase the device. If the window has been uncovered for some time during operation, you should completely erase the EPLD, reprogram it, and cover the window with an opaque label.

- ❑ Ensure that the input voltages never exceed V_{CC} (you can check voltages with an oscilloscope). If they do, latch-up can occur, and the EPLD may become hot and/or consume excessive current.

- ❑ Ensure that the leads of the EPLD are straight and do not touch each other. The corner leads of J-lead packages can become damaged if the devices are not inserted carefully into the programming, prototyping, or production sockets.

- Check the design for Preset, Clear, and Clock inputs to registers that are generated by the logic array. These signals are prone to glitches because intermediate states between logic transitions may be long enough to trigger register controls. Inputs to registers can be made glitch-free by passing them through another register. Also check for setup and hold violations that may occur over system temperature or voltage variations. Refer to *Application Brief 94 (Solutions to Design Timing Problems for Altera EPLDs)* in this handbook for more information on reliable EPLD design practices.

- Simulate the design for all phases of operation. MAX+PLUS II offers functional and timing simulation tools that allow full analysis of a design. If you have the appropriate testing hardware, MAX+PLUS II also offers functional testing. You can use simulation and functional testing to spot subtle problems you may have missed when creating your design. Simulation will also help you to understand EPLD timing and signal delay.

Solutions to Timing Problems

If a problem is caused by incorrect timing, you will find the following solutions helpful. Also, refer to *Application Brief 100 (Understanding EPLD Timing)* in this handbook for details on timing delays and characteristics of Altera EPLDs.

- Slow the Clock down. A register in the design may have missed a setup or hold time. Simulate the design and check setup and hold times. The MAX+PLUS II Simulator contains setup/hold violation monitors. The MAX+PLUS II Timing Analyzer also offers setup/hold analysis.

- Heat up or cool down the EPLD to see whether the device operates correctly at one temperature but fails at another. Temperature changes may cause changes in the propagation delays of the EPLD. If the EPLD fails at any temperature within its specified operating range, the problem is probably timing-related.

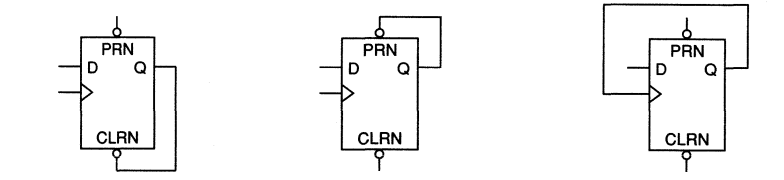
- Try another EPLD of the same type with a different date code (the date code is the set of digits next to or under the part number). EPLDs with different date codes may have slightly different performances. All Altera EPLDs are guaranteed to have certain maximum delays according to their speed grade. However, some EPLDs may be faster than specified. A faster EPLD may uncover problems in a design that relies on a minimum delay or a slow feedback path.

- Try an EPLD of the same type with a faster speed grade. Your design may include complex combinatorial logic that requires the input signals to take several paths to generate the desired output. These input signals may have to be implemented by several logic elements. Each path may have a different delay, depending on where it comes

from and how many logic elements it passes through. Check the Report File (.RPT) to ensure that speed-critical paths have been implemented as expected. With MAX+PLUS II, you can use cliques to group logic and specify speed-critical paths for design compilation.

- ❑ Check your design for feedback paths that control registers and are generated by the logic array. Pay particular attention to Clear or Preset signals generated by feedback paths from registers. You should not use self-clearing, self-presetting, or self-clocking registers (shown in Figure 1) because feedback paths may be faster than anticipated, and may cause the registers to react at incorrect times.

Figure 1. Self-Clearing, Presetting, and Clocking Registers



If you cannot solve a functional problem, contact Altera Applications at (800) 800-EPLD. Please answer the questions on the following Troubleshooting Questionnaire before calling, so that the Altera Applications Engineer can help you solve the problem promptly and satisfactorily.

Troubleshooting Checklist

Answer the following questions during the debugging process. If you subsequently need to contact Altera Applications for assistance, these answers will be easily accessible.

General Information

1. Which EPLD are you using (e.g., EP610-25)? _____
2. What package type are you using (e.g., DIP, J-lead)? _____
3. How many EPLDs have failed? _____
4. How many EPLDs have been tested? _____
5. What are the date codes for the problem EPLDs? _____
6. What hardware setup do you have (e.g., Altera, Data I/O)? _____

7. Have the EPLDs ever functioned correctly in your current setup? ❑ Y ❑ N
8. Are the EPLDs new (i.e., straight from the tube)? ❑ Y ❑ N
9. Have the EPLDs been erased and reprogrammed? ❑ Y ❑ N
10. Did you erase the EPLDs for at least one hour? ❑ Y ❑ N
11. Have you simulated this design? ❑ Y ❑ N
12. If you are using MAX+PLUS II, have you functionally tested the EPLD with the MPU? ❑ Y ❑ N
13. If you are using a Classic or MAX 7000 EPLD, is the Turbo Bit turned on? ❑ Y ❑ N
14. If the EPLD is ceramic, has the window been covered? ❑ Y ❑ N
15. Are the leads of the EPLD straight? ❑ Y ❑ N

Troubleshooting

1. What are the V_{CC} rise times? _____
2. Is the V_{CC} ramp-up smooth? ❑ Y ❑ N
3. Have you checked for V_{CC} overshoot and ringing/oscillation? ❑ Y ❑ N
4. Have you used decoupling capacitors across each pair of VCC and GND signals? ❑ Y ❑ N
5. Are the decoupling capacitors adequate? (Altera recommends 0.2 µF.) ❑ Y ❑ N
6. Are you using a wire-wrapped board? ❑ Y ❑ N
7. Does the board have a ground plane? ❑ Y ❑ N
8. Are all pins connected according to the Report File? ❑ Y ❑ N
9. At what frequency is the design running? _____
10. What are the input rise times? _____
11. Are the inputs arriving at the device after power-up? ❑ Y ❑ N

12. Do the input voltages ever exceed V_{CC}? ❑ Y ❑ N

13. Does the design contain array-generated signals that clock, preset or clear registers? ❑ Y ❑ N

Advanced Troubleshooting

14. Does slowing the Clock have any effect? ❑ Y ❑ N

15. Does heating/cooling the EPLD have any effect? ❑ Y ❑ N

16. Does trying different date codes have any effect? ❑ Y ❑ N

17. Does trying a faster speed grade have any effect? ❑ Y ❑ N

18. Are the EPLDs socketed or soldered to the board? _____

19. If socketed, what type of socket are you using? _____

20. What resistive and capacitive loads are the outputs driving? _____

Describe the Problem

The more information you provide, the faster Altera Applications Engineers will be able to analyze the failure. A description of the problem should include the following information:

❑ The exact conditions leading up to the failure, and the exact nature of the failure. Which pins are failing? What should their outputs be? What are their outputs? For example: "10 ms after power-up, I apply 5 V to pin 3 and pin 5. Pin 7 should go high because it is the logical AND of pin 3 and pin 5, but it stays low."

- ❑ The types of files used in and generated for the design.
  - A+PLUS: LogiCaps Schematic Design Files (.SD), Altera Design Files (.ADF), State Machine Files (.SMF), and JEDEC Files (.JED)
  - SAM+PLUS: Assembly Language Files (.ASM), State Machine Files (.SMF), and JEDEC Files (.JED)
  - MCMap: MCMap Design File (.MC1), Adapter Description File (.ADF), and JEDEC Files (.JED)
  - MAX+PLUS (DOS): Graphic Design Files (.GDF), Text Design Files (.TDF), Compiler Netlist Files (.CNF), Hierarchy Interconnect Files (.HIF), Simulator Netlist Files (.SNF), Simulator Channel Files (.SCF), Programmer Object Files (.POF), EDIF Input Files (.EDF), EDIF Output Files (.EDO), EDIF Command Files (.EDC), and Altera Design Files (.ADF)
  - MAX+PLUS II: Graphic Design Files (.GDF), Text Design Files (.TDF), Waveform Design Files (.WDF), Compiler Netlist Files (.CNF), Hierarchy Interconnect Files (.HIF), Simulator Netlist Files (.SNF), Simulator Channel Files (.SCF), and Programmer Object Files (.POF), JEDEC Files (.JED), Probe & Resource Assignment Files (.PRB), *<Project Name>*.INI files, EDIF Input Files (.EDF), EDIF Output Files (.EDO), EDIF Command Files (.EDC), and Altera Design Files (.ADF)

- ❑ A schematic of the device on the board, including all surrounding devices. How are the pins connected? What are the capacitive loads on the pins?

- ❑ All relevant plots, graphs, and pictures. Include any other information.

- ❑ For a MAX+PLUS or MAX+PLUS II design, include a Simulator Channel File (.SCF) of the input conditions that create the failure.

References

For more information on recommended design practices and solving EPLD problems, consult the following application notes and briefs in this handbook:

Application Note 26 (EPLD/MPLD Design Guidelines)
Application Brief 81 (Troubleshooting EPLD Programming Problems)
Application Brief 89 (Minimizing Output Switching Noise in Altera EPLDs)
Application Brief 94 (Solutions to Design Timing Problems for Altera EPLDs)
Application Brief 100 (Understanding EPLD Timing)

Optimizing Counter Design for Altera EPLDs

April 1992, ver. 1

Application Brief 88

Introduction

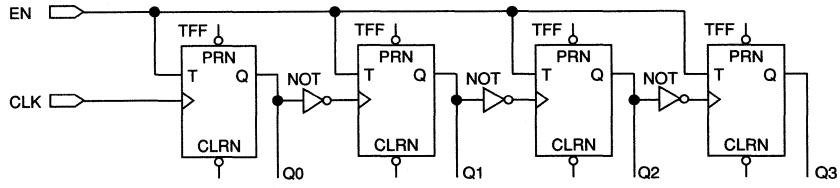
Most logic designs contain one or more counters, which typically require a large number of routing resources to drive the logic that feeds the counter registers. When routing resources are limited, you can use a macrocell for a look-ahead carry function to reduce the number of routing resources required for the counter.

This application brief describes different counter types and shows how a look-ahead carry function can greatly improve the performance of your design. The application brief provides sample designs generated with the MAX+PLUS II software, and a custom macrofunction for implementing look-ahead carry functions in any design.

Basic Counter Design

There are two types of counters: asynchronous (ripple) counters and synchronous counters. In an asynchronous counter, each register is clocked from the output of the previous register (see Figure 1). The counter, therefore, requires little logic, but is slow because the outputs are not synchronized.

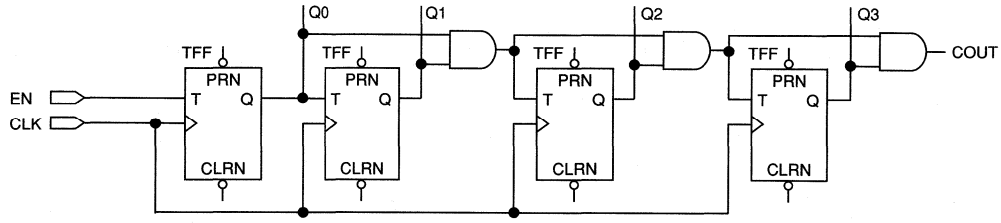
Figure 1. Asynchronous Counter



In synchronous counters, on the other hand, all registers are clocked from a common system Clock. The count function is built with logic that feeds the data input of the registers. Figure 2 shows an example of a synchronous counter.

This counter uses the output of less significant registers to enable the "toggle" input of the more significant registers. The output transitions are synchronized to the Clock input, providing a synchronized output for driving other logic. The maximum Clock frequency is the reciprocal of the propagation delay from any register to the next register, plus a nominal setup time. As a result, the speed of the counter does not vary with the number of counter bits.

Figure 2. Synchronous Counter



The routing requirements necessary for the synchronous carry function can become prohibitive. Synchronous counters require *n*–1 routing channels for an *n*-bit counter. For example, the 4-bit counter in Figure 2 shows how bits Q0, Q1, and Q2 are routed to the carry of Q3. Although this requirement is not a problem for counters with fewer than 16 bits, counters with more than 16 bits can consume routing resources that may be needed for other functions in the design. However, by using look-ahead carry functions, you can drastically reduce the number of routing channels needed to make a synchronous counter.

Counters in EPLDs

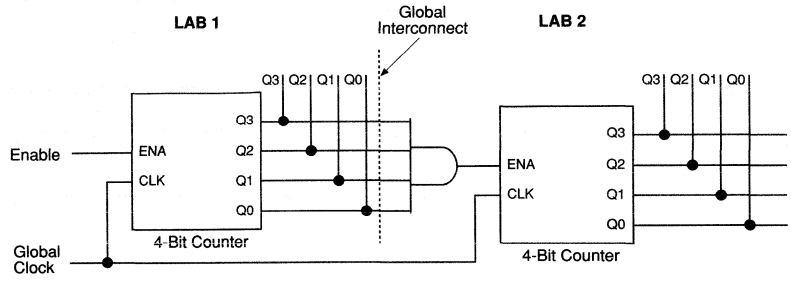
The modular Altera EPLD architecture contains both local and global routing resources. While you may use any number of local routing resources, global routing resources are limited in large EPLDs. To reduce the number of global routing resources used, you can keep counter bits together in a single quadrant or Logic Array Block (LAB). However, occasionally, the registers of a counter exceed the size of an LAB, or pin-out constraints require them to be placed in different LABs. Without a look-ahead counter, the outputs of all registers of a counter need to be routed across the global routing channels. However, if you allocate a single macrocell for a look-ahead carry function, the same counter can be implemented with only one global routing channel.

Look-Ahead Counter

Figure 3 shows an example of an 8-bit counter with the lower four counter bits in a different LAB than the upper four counter bits. Four routing lines are required to connect the logic.

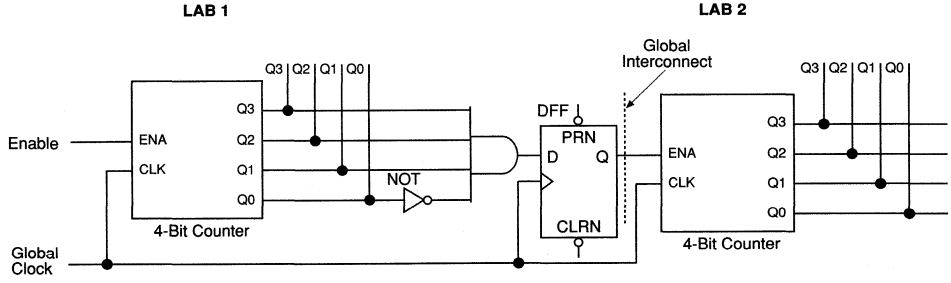
Using a look-ahead carry function, you can achieve the same functionality and timing with only one routing channel. The carry is logically determined one Clock cycle before it is needed and then stored in a register. Only a single carry value needs to propagate globally, but it is immediately available to the higher bits.

Figure 3. 8-Bit Counter



The look-ahead counter, shown in Figure 4, uses one extra register to hold a carry value. A second register is needed to hold a borrow value if the counter has both up and down operation. By using one or two extra registers, you can retain the performance of the counter with only minimal use of global routing resources.

Figure 4. Look-Ahead Counter



Look-Ahead Carry Function

Since look-ahead counters are registered, they must activate the carry function one Clock cycle early. For example, the look-ahead carry function in Figure 4 goes high when the first four counter bits are a binary 1110, ensuring that the carry signal enables the upper counter bits when the lower counter bits are a binary 1111. If you want to add more functionality to the counter, such as enable and up/down functions, you must incorporate the control signals into the carry logic, as shown in the following example implemented in the Altera Hardware Description Language (AHDL):

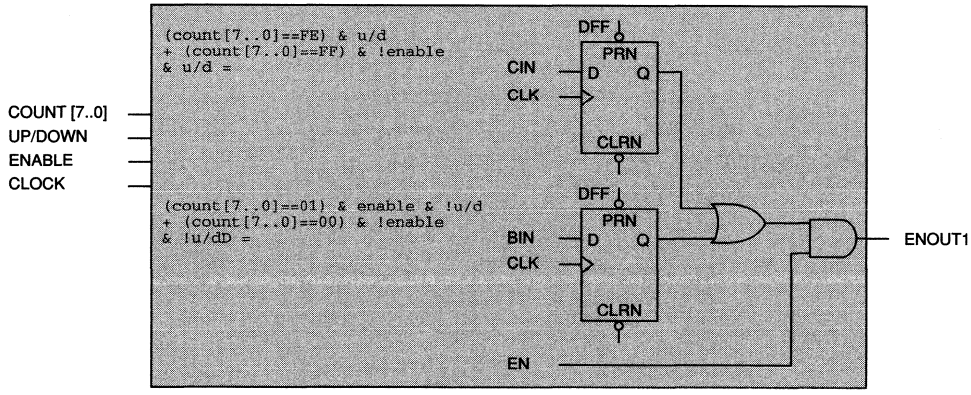
```
cin     = (count[3..0] == E) &  enable & u/d
        + (count[3..0] == F) & !enable & u/d & !load
        + (count[3..0] == F) & !enable & u/d &  load
        & (data[3..0]  == F)
```

If the counter uses the up/down function, then an additional register is needed for a look-ahead borrow:

```
bin     = (count[3..0] == 1) &  enable & !u/d
        + (count[3..0] == 0) & !enable & !u/d & !load
        + (count[3..0] == 0) & !enable & !u/d &  load
        & (data[3..0]  == 0)
```

Figure 5 shows such a macrofunction with both a carry and a borrow. The outputs of the registers are combined with the global Enable signal to enable the higher register bits.

Figure 5. Macrofunction with Carry and Borrow



Alternatively, you can combine the CIN and BIN equations into a single carry/borrow function and feed the result into a single look-ahead register. This approach requires one less macrocell, but may reduce the speed of the design.

Carry Macrofunction

Figure 6 shows an AHDL macrofunction called CARRY that bundles the registers and associated logic for easy implementation. This macrofunction uses two registers and two global routing lines when the up/down function is needed. If the up/down input is permanently enabled for only up or down operation, the MAX+PLUS II software automatically removes the extra register and its associated logic.

Figure 6. CARRY Macrofunction with 8-Bit Counter (CARRY.TDF)

```
SUBDESIGN carry
(
    clk                 : INPUT;
    enable              : INPUT;
    u/d                 : INPUT;
    load                : INPUT;
    data[7..0]          : INPUT;
    count[7..0]         : INPUT;
    carry               : OUTPUT;
)

VARIABLE
    up, down            : DFF;
BEGIN
    (up,down).clk = clk;

    up      = (count[7..0] == h"FE") &  enable & u/d
            # (count[7..0] == h"FF") & !enable & u/d & !load
            # (count[7..0] == h"FF") & !enable & u/d &  load
            & (data[7..0]  == h"FF");

    down    = (count[7..0] == 1) &  enable & !u/d
            # (count[7..0] == 0) & !enable & !u/d & !load
            # (count[7..0] == 0) & !enable & !u/d &  load
            & (data[7..0]  == 0);

    carry   = (up # down) & enable ;

END;
```

You can easily modify the macrofunction to accommodate counters of any size. For example, with the following modifications, you can change CARRY.TDF to a 14-bit counter:

1. Change `count[7..0]` to `count[13..0]`.
2. Change all occurrences of `FE` to `3FFE`.
3. Change all occurrences of `FF` to `3FFF`.

After making these changes in CARRY.TDF, you must save the file under the filename `C_LOOK_A` and create the symbol that represents the file. In the MAX+PLUS II Text Editor, choose the **Create Default Symbol** command from the File menu. Once you have updated the macrofunction, you can use it in other design files.

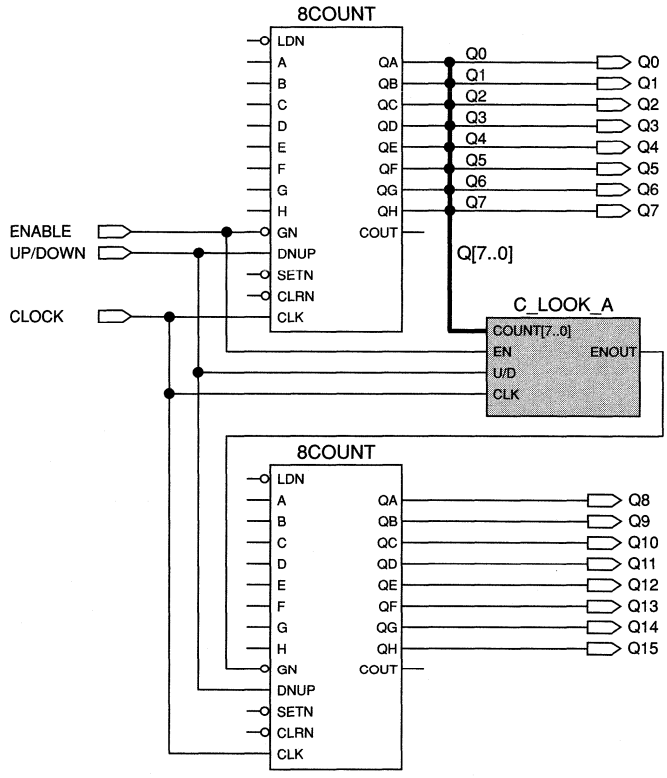
Design Implementation

The following steps add the CARRY macrofunction to a design with limited routing resources:

1. Modify and compile the look-ahead macrofunction for the number of bits required.
2. Enter the CARRY symbol in the schematic.

Figure 7 shows the schematic of a 16-bit counter placed into an EP1810 EPLD. The design takes advantage of the 8 local macrocells in each quadrant. The carry function CARRY.TDF has been edited and renamed to C_LOOK_A, and now supports an 8-bit look-ahead function. The lower 8 bits are assigned to the local macrocells in a quadrant. The carry function is assigned to a global macrocell in the same quadrant. Taking advantage of the look-ahead carry, this design uses only one globally routed macrocell instead of the 8 global macrocells that would otherwise have been required.

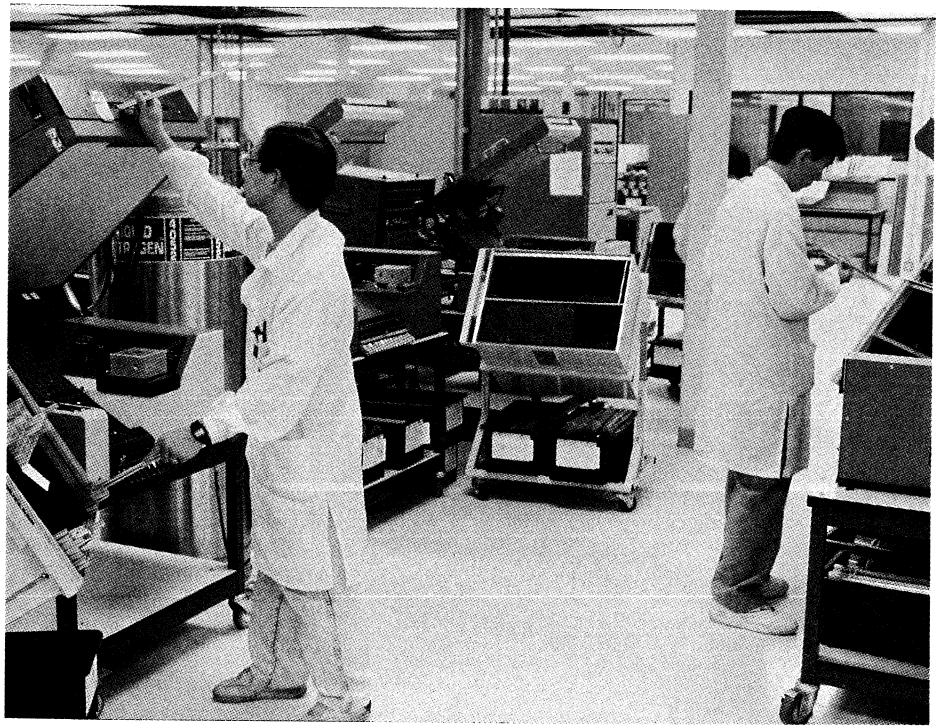
Figure 7. 16-Bit Counter with Look-Ahead Carry (C_LOOK_A)



Logic designs implemented with MAX 5000 and MAX 7000 EPLDs can also benefit from a look-ahead carry. Since the MAX architecture provides LABs of 16 macrocells, counters are best broken into 15-bit groups with a single look-ahead register in the same LAB. An up/down counter would use 14 bits and 2 look-ahead registers in the same LAB.

Conclusion

Look-ahead carry functions reduce routing resources for large counters while retaining the same operating frequency. Designs that need to ration global routing resources can use CARRY.TDF to drastically reduce the global lines used for the counter function. You can download CARRY.TDF via modem from Altera's electronic bulletin board service (BBS) by calling (408) 249-1100.



Minimizing Output Switching Noise in Altera EPLDs

April 1992, ver. 1

Application Brief 89

Introduction

As digital devices become faster, their output switching times decrease. Faster switching times cause higher transient currents in outputs as they discharge load capacitances. These higher currents can lead to the board-level phenomenon known as "ground bounce." You can minimize the effect of ground bounce on your design by understanding the key elements that contribute to it: load capacitance, ground path inductance, and the number of switching outputs. This application brief discusses these factors and suggests design practices that help minimize ground bounce.

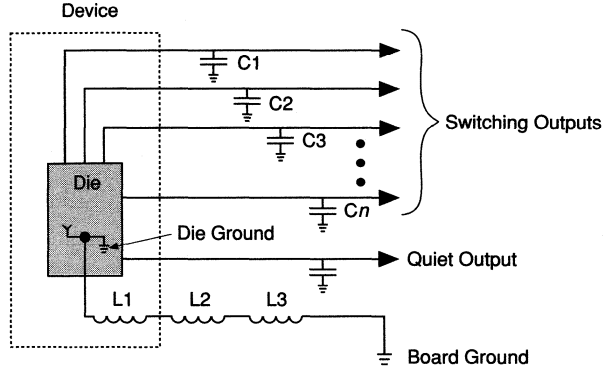
Ground Bounce

Figure 1 shows a simple model for analyzing ground bounce. The external components driven by the device appear to that device as capacitive loads (C1 to C*n*). These capacitive loads store a charge determined by the following equation:

$$\text{charge } (Q) = [\text{voltage } (V) \times \text{capacitance } (C)]$$

Thus, the charge increases as the voltage and/or load capacitance increases.

Figure 1. Ground Bounce Model



The environment and ground path of a device have intrinsic inductances (shown in Figure 1 as L1, L2, and L3). L1 is the inductance of the bond wire from the device's die to its package pin, and of the pin itself. L2 is the inductance of the connection mechanism between the device's ground pin and the printed circuit board (PCB). This inductance is greatest when the

device is connected to the board through a socket. L3 is the inductance of the PCB trace between the device and the board location where other devices in the system reference ground.

Ground bounce occurs when multiple outputs switch from high to low. The transition causes the charge stored in the load capacitances to flow into the device. The sudden rush of current (di/dt) exits the device through the inductances (L) to board ground, generating a voltage (V) determined by the equation V = L(di/dt). This voltage difference between board ground and device ground causes the relative ground level for low ("quiet") outputs to temporarily rise or "bounce." Although the in-rush of current is brief, the magnitude of the bounce can be large enough to trigger other devices on the board.

Capacitive Loading on Outputs

Capacitive loading on the switching and quiet outputs affects the magnitude of ground bounce differently.

Switching Outputs

When the capacitive loading on the switching outputs increases, the amount of charge available for instantaneous switching increases, which in turn increases the magnitude of ground bounce. Depending on the device, ground bounce increases with capacitive loading until the loading is approximately 200 pF per device output. At this point, the device output buffers reach their maximum current-carrying capacity and inductive factors become dominant.

One method to reduce the capacitive load and hence ground bounce is to connect the EPLD's switching outputs to a bus driver IC. The outputs of the bus driver IC drive the heavy capacitive loads, reducing the loading on the EPLD, thus minimizing ground bounce for the EPLD's quiet outputs.

Some bus applications use pull-up resistors to create a default high value for the bus. These resistors cause the load capacitances to charge up to the maximum voltage. Consequently, the driving device produces a higher level of ground bounce. Therefore, you should eliminate pull-up resistors in applications in which ground bounce is a concern, or design the bus logic to use pull-down resistors instead.

Quiet Outputs

An increase in capacitive loading on quiet outputs acts as a low-pass filter and tends to dampen ground bounce. Capacitive loading on a quiet output can reduce the magnitude of ground bounce by as much as 200 to 300 mV. However, an increase in capacitive loading on a quiet output can increase the noise seen on other quiet outputs.

Sockets & Board Inductances

Two elements of inductance L2 are socket usage and board trace length. Sockets can cause ground bounce voltage to increase by as much as 100%. You can often dramatically reduce the magnitude of ground bounce on the board by eliminating sockets.

The length of the board trace has a much smaller effect on ground bounce than sockets. For PCBs with a ground plane, the voltage drop across L3 is negligible, because L3 is significantly less than L2. The inductance in a 3-inch trace increases ground bounce for a quiet output by approximately 100 mV. Nevertheless, trace length should be kept to a minimum. As traces become longer, transmission line effects may cause other noise problems.

Multiple Switching Outputs

The number of switching outputs also affects ground bounce. As the number of switching outputs increases, the total charge stored also increases. The total charge is equal to the sum of the stored charges for each switching output. Therefore, the amount of current that must sink to ground increases as the number of switching outputs increases. Ground bounce can increase by as much as 40 to 50 mV for each additional output that is switching.

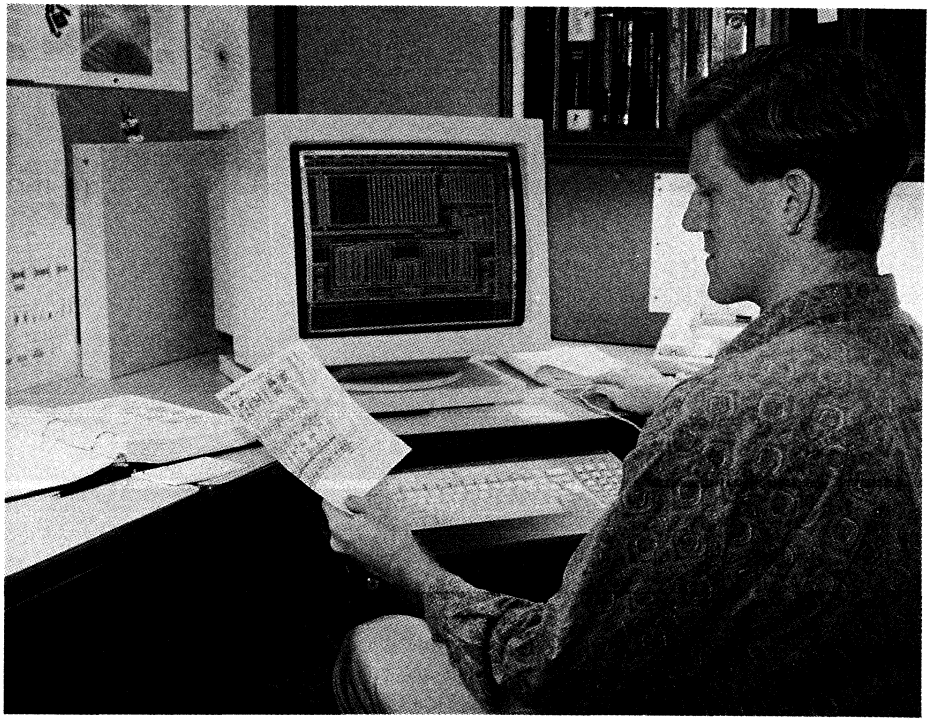
To reduce ground bounce, limit the number of outputs that can switch simultaneously in your design to eight or fewer. For functions such as counters, you can use Gray coding as an alternative to standard sequential binary coding, since only one bit switches at a time.

In extreme cases, adding resistors (10 to 30 is usually adequate) in series to each of the switching outputs in a high-speed logic device can limit the current flow into each of the outputs, and thus reduce ground bounce to an acceptable level.

Conclusion

Ground bounce is a board-level phenomenon. Since many factors affect its magnitude, no standard test methods allow you to predict the amount of ground bounce for all possible board environments. You can only test the device under a given set of conditions to determine the relative contributions of each condition and of the device itself. Load capacitance, inductance of sockets, and the number of switching outputs are the predominant factors that influence the magnitude of ground bounce in EPLDs.

To reduce the magnitude of ground bounce, Altera recommends limiting load capacitance by buffering loads with an external device such as the 74244 IC bus driver, or by reducing the number of devices connected to the driven bus. You can also eliminate sockets whenever possible and/or reduce the number of outputs that can switch simultaneously. These design practices should help you create high-speed logic designs that operate without problems over a wide range of board conditions.



External Clock Sources for Altera EPLDs

April 1992, ver. 1

Application Brief 91

Introduction

In circuit designs that use registered and sequential logic, you need Clock signals to load register inputs and/or control state machine operation. In some systems, you can control registered logic with a system Clock. Designs that use a system Clock are synchronous, and are typically clocked with a global signal generated by dedicated Clock circuitry in the system. Synchronous designs include PC motherboards, video adapters, network interface units, and printers.

A system Clock is usually sufficient for the clocking requirements of Altera EPLDs. However, if no system Clock is available, you must generate a Clock to control the registered functions in the digital logic. You can generate a Clock using a variety of techniques. This application brief discusses the two most common methods used in EPLD circuits:

- Direct current (DC) Clock generator
- Resistor/capacitor (RC) crystal network

DC Clock Generator

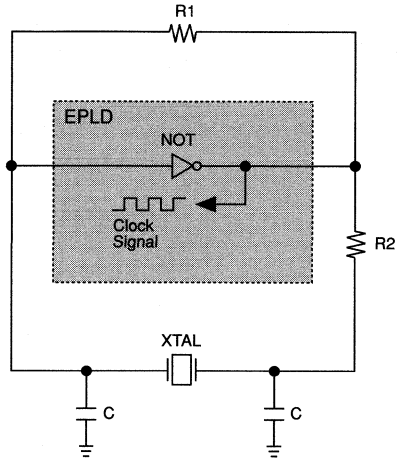
Nearly every design implemented in programmable logic uses a monotonic Clock signal (i.e., a Clock with a regular cycle) to synchronize incoming signals, latch data buses, drive state machines, or control registered logic. In circuits that contain an external processor, the system Clock can drive the EPLD Clock.

However, when the EPLD is the processor or control circuit and a system Clock is not available, you can generate a Clock signal using a stand-alone DC Clock generator. A DC Clock generator is a 4-terminal device that is available off-the-shelf from several manufacturers in surface or through-hole mountings. It provides a steady, temperature-compensated Clock signal from a range of input VCC signals. A DC Clock generator is often used for PC motherboards, interface cards, and stand-alone controllers, and is well-suited to EPLDs.

RC Crystal Network

If cost or space restrictions prevent you from using a DC Clock generator, you can use the logic in an Altera EPLD to provide feedback in an RC crystal network. Figure 1 shows how an inverter in the EPLD can provide the restoring feedback signal to drive an RC crystal network at its fundamental frequency. Be sure to follow the crystal manufacturer's recommendations for capacitor and resistor values for a particular operating frequency.

Figure 1. RC-Inverter Network with an Altera EPLD



Even if you are familiar with generating Clocks using pure RC networks, and are comfortable with the mathematics necessary to calculate the fundamental frequency, you should not make an external RC network with EPLD inverters. The specified propagation speed of the EPLD inverter path is a "worst-case" value, and the actual speed is typically much faster than specified. If you do not know the exact propagation delay (which varies with temperature and V_{CC}), your results may not be reliable.

Conclusion

When you select a Clock source for EPLDs, you should look for system-level clocking resources provided by a motherboard or a host controller. These resources are guaranteed to provide accurate and stable Clock sources. If your design does not have a system Clock, you must create a Clock. DC Clock generators are the most reliable and accurate because they are not sensitive to minor variations in the V_{CC} voltage and compensate for temperature fluctuations. The RC crystal network is also acceptable, but requires precise control of resistor and capacitor values.

Simulating Internal Nodes with MAX+PLUS II Software

April 1992, ver. 1

Application Brief 92

Introduction

The MAX+PLUS II Simulator and Waveform Editor allow you to verify logic graphically, both at the EPLD pins and at internal nodes within the device. This application brief describes how to simulate internal nodes with the MAX+PLUS II software. The following topics are discussed:

- Differences between functional and timing netlist files created for simulation
- Generating lists of nodes that can be simulated
- How nodes are named
- Simulating internal nodes in state machines and combinatorial logic
- Creating simulation input files

Functional vs. Timing Simulator Netlist Files

When you compile a design (called a "project" in MAX+PLUS II), the MAX+PLUS II Compiler allows you to create either a functional or timing Simulator Netlist File (.SNF) to use with the MAX+PLUS II Simulator. A functional SNF is generated when the Compiler runs only the Compiler Netlist Extractor, Database Builder, and optional Functional SNF Extractor modules. Since no logic synthesis or fitting is performed, timing information is not included in the functional SNF. This file contains the functional behavior of the project, and includes all nodes from the original design files.

A timing SNF is generated during when you run a full compilation with the optional Timing SNF Extractor module turned on. This file contains all information for timing simulation and timing analysis. It describes the fully optimized circuit after logic minimization, synthesis, fitting, and partitioning have been completed. This process invariably eliminates some of the original nodes that defined the circuit, particularly in combinatorial logic. Only nodes that are associated with "hard logic" functions—including input, output, and bidirectional pins; `MCELL` buffers; registers; and latches—are certain to remain in the fully synthesized design and therefore in the timing SNF. `SOFT` and `TRI` buffers may also be retained if they are not eliminated during logic synthesis. Other nodes that have been synthesized away cannot be simulated.

Listing Nodes & Groups for Simulation

You can list all nodes and groups (buses) in a functional or timing SNF in either the Waveform Editor or the Simulator. All nodes that appear in the SNF can be simulated.

Waveform Editor

In the Waveform Editor, you can create a default Simulator Channel File (.SCF). You can also create an optional Table File (.TBL), which has the same format as a Vector File (.VEC). Either file can then be edited to provide the vector inputs for simulation.

To create and print an SCF and an optional Table File that contain all nodes in the project:

1. Choose the **Create Default Channel** command (File menu) in the Waveform Editor, then choose the **List** button to list all nodes and groups in the *Available Nodes & Groups* box. Since the default asterisk (*) wildcard character in the *Node/Group* box matches all names, and the default *All* checkbox under *Type* matches all nodes and group types, the **List** button automatically displays all nodes and groups in the *Available Nodes & Groups* box.
2. Choose the right direction button (=>) to move all nodes and groups to the *Selected Nodes & Groups* box, and choose **OK** to create a default Simulator Channel File (.SCF).
3. Choose the **Save** command (File menu), type a filename, and choose **OK** to save the file.
4. (Optional) To create an ASCII Table File (.TBL) that contains all nodes and groups in the new SCF, choose the **Create Table File** command (File menu), type a filename, and choose **OK**.
5. When the SCF or Table File is open, choose the **Print** command (File menu) to print the file. If necessary, first choose the **Open** command (File menu) to open the file.

You can exclude nodes from the SCF and Table File by changing the pattern-matching string in the *Node/Group* box and the *Type* checkboxes, and by moving specific nodes into the *Selected Nodes & Groups* box in the **Create Default Channel** dialog box.

Create Default Channel also includes an *Update Existing Channel* option that allows you to add nodes from the current SNF to an existing SCF, for example, after you have recompiled a project. You can also add nodes to an existing SCF with the **Enter Node** command (Node menu). The **Enter Node** dialog box has a **List** button that lists nodes and groups in the current SNF in the same way as **Create Default Channel**.

For information on creating simulation input vectors in SCFs or Vector Files, refer to MAX+PLUS II Help.

Simulator

In the Simulator, you can list nodes and groups with the **Initialize** command (Options menu), and optionally save this list to a History File (.HST). Unlike the SCF or Vector File, however, you cannot easily edit the History File to create simulation inputs.

To create a History File, list nodes and groups, and print the file:

1. Choose the **Inputs/Outputs** command (File menu) in the Simulator, select *History (.HST)* and *New*, type a filename, and choose **OK** to create a new History File.
1. Choose the **Initialize** command (Options menu) in the Simulator, then choose the **List** button to list all nodes and groups in the *Nodes & Groups* box. Since the default asterisk (*) wildcard character in the *Node/Group* box matches all names, and the default *All* checkbox under *Type* matches all nodes and group types, the **List** button automatically displays all nodes and groups in the *Nodes & Groups* box.
2. Choose **OK** to close the **Initialize** dialog box. The History File records all nodes and groups listed in step 2 as the output of `NODE LIST` and `GROUP LIST` commands.
3. Choose the **Open** command (File menu), open the History File, and choose the **Print** command (File menu) to print the file.

Identifying Node & Group Types

Node and group types in the SNF are listed slightly differently in the SCF, Table File, or History File created with the preceding procedures:

SCF:	Node and groups are assigned the type `I` (input), `O` (output), or `B` (buried), which is shown in the node/group handle. Each bidirectional node is listed twice, once as an input and once as an output.
Table File:	Nodes are listed in the Inputs, Outputs, and Buried Sections; each group name and its members are listed in a separate Group Create Section. Bidirectional nodes are listed twice, as both inputs and outputs.
History File:	Nodes and groups are listed as `IN`, `OUT`, `I/O`, or `BUR`. Bidirectional nodes are listed only once with type `I/O`.

Input, output, and bidirectional nodes correspond to EPLD input, output, and bidirectional pins, respectively. These nodes have the same names as their corresponding pins. Nodes defined as buried are internal to the device; they have no direct pin connection.

How Nodes are Named

Nodes are listed in the SCF, Table File, or History File only once, under the node name with the highest priority. However, MAX+PLUS II allows you to use a lower-priority name when you specify nodes in the Simulator,

create a file that provides simulation vectors, or locate a node in a design file.

Node names may take one of three basic forms. These three types of nodes are described in the following order:

Priority 3: Pinstub/port-based names (lowest priority)
Priority 2: Explicitly named nodes (medium priority)
Priority 1: Probes (highest priority)

MAX+PLUS II node-naming conventions may cause the name of a particular node to vary between functional and timing SNFs, because logic synthesis removes unneeded nodes. The following sections give examples of the differences between the two types of SNFs.

Each Priority 3 and Priority 2 node name in a lower-level file of a hierarchical project also incorporates a hierarchical pathname, which traces the node name along a path down through multiple levels of the hierarchy. Hierarchical pathnames are described later in this application brief, in "Hierarchical Pathnames in Priority 3 & Priority 2 Node Names."

Priority 3: Pinstub/Port-Based Names

The lowest-priority node names are based on their status as inputs or outputs of primitives. These names are used in Text Design Files (.TDF) in the Altera Hardware Description Language (AHDL) and in Graphic Design Files (.GDF). They are not used in Waveform Design Files (.WDF), in which all nodes are explicitly named. Different types of names are assigned for the following categories of primitives:

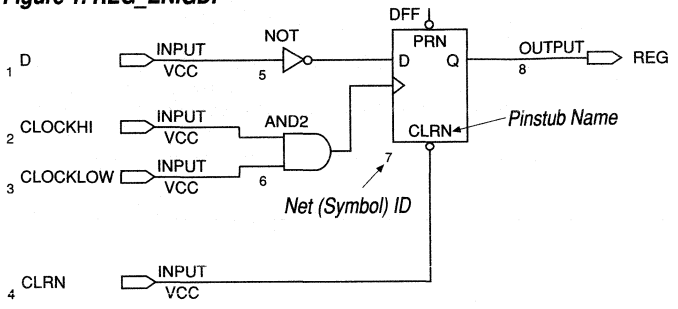
- I/O pins & ports
- Other primitives
- Primitive instance names in AHDL TDFs

I/O Pins & Ports

A node that feeds or is fed by a pin in a GDF or WDF automatically takes the name of a pin. In TDFs, pin names are called port names. Nodes that feed or are fed by ports also take the port name. Figure 1 shows REG_EN.GDF and the Priority 3 pin/port-based names for each node in the file.

TDF port types also include `MACHINE INPUT` and `MACHINE OUTPUT`. These port names exist only in lower-level files in a hierarchy, and always have hierarchical node names. (See "Hierarchical Pathnames in Priority 3 & Priority 2 Node Names" later in this application brief.)

Figure 1. REG_EN.GDF



Node Name	
Functional SNF	**Timing SNF**
D	D
CLOCKHI	CLOCKHI
CLOCKLOW	CLOCKLOW
CLRN	CLRN
(none)	:7.CLRN
:6	:7.CLK
:5	:7.D
:7	:7
REG	REG

Other Primitives

Each primitive has a unique identification number. The MAX+PLUS II Graphic Editor assigns a symbol ID to every symbol in a GDF, located in the bottom left-hand corner of the symbol. See Figure 1. The Compiler assigns net ID numbers—which function in the same way as symbol ID numbers—to primitives in TDFs and WDFs.

Each input and output signal of a primitive has a default pinstub name (called a "stub name" in a TDF). The lowest-priority node names combine the net ID and pinstub name. These names have the following Backus-Naur Form (BNF) format:

:*<net ID>*.*<pinstub name>*

Table 1 shows the pinstub names for all primitives and ports.

Table 1. Pinstub/Stub Names

Primitive/Port	Input Pinstub/Stub	Output Pinstub/Stub
INPUT *(1)*	n/a	*<pin name>*
OUTPUT *(1)*	*<pin name>*	n/a
MACHINE INPUT	n/a	*<pin name>*
MACHINE OUTPUT	*<pin name>*	n/a
BIDIR *(1)*	*<pin name>*	*<pin name>*
MCELL *(1)*, SOFT *(2)*, NOT, EXP, GLOBAL	IN	OUT
TRI *(2)*	IN, OE	OUT
DFF *(1)*	D, CLK, CLRN, PRN	Q
DFFE *(1)*	D, CLK, CLRN, PRN, ENA	Q
JKFF *(1)*	J, K, CLK, CLRN, PRN	Q
JKFFE *(1)*	J, K, CLK, CLRN, PRN, ENA	Q
LATCH *(1)*	D, ENA	Q
SRFF *(1)*	S, R, CLK, CLRN, PRN	Q
SRFFE *(1)*	S, R, CLK, CLRN, PRN, ENA	Q
TFF *(1)*	T, CLK, CLRN, PRN	Q
TFFE *(1)*	T, CLK, CLRN, PRN, ENA	Q
AND, BAND, BNAND, BNOR, BOR, NAND, NOR, OR, XNOR, XOR	IN1, IN2, ... IN*n* (where *n* equals the number of inputs)	OUT
GND, VCC	n/a	n/a

Notes:
(1) Hard logic function.
(2) Hard logic function if not eliminated by logic synthesis.

Since all primitives except OUTPUT drive a single output node, the *.<pinstub name>* is omitted and the *:<net ID>* alone represents the name of the output node. However, you can optionally use the output pinstub name to refer to the node.

The Priority 3 (lowest priority) names may take different formats in functional and timing SNFs:

Functional SNF: All lowest-priority names are expressed as the output of a symbol, and appear as *:<net ID>*, where the ID is that of the primitive that feeds the node. Input pinstub names are not used.

Timing SNF: Primitives in the original design files may have been synthesized away. Therefore, if a node feeds a hard logic function, it is often renamed with the input pinstub name of that function. Otherwise, names have the same format as those in a functional SNF.

For example, in the functional SNF, the node that feeds the Clock input to the DFF in Figure 1 is called :6, which is the output of the AND2 gate. In the timing SNF, this node is called :7.CLK.

When two hard logic functions are connected, the node that connects them may appear as both the output of the first function and the input of the second in the functional and/or timing SNF. Such nodes appear twice because the physical EPLD implementation of the project will create a timing delay between them. However, in a functional SNF, no delay is assumed during simulation, even though both names appear in the file. If a node appears only once, no delay between the two functions can be assumed, and the node takes the shorter of the two possible names. For example, in Figure 1 the register DFF is connected to the output pin REG, and both the register output (:7) and the pin name appear in the timing and functional SNF.

Primitive Instance Names in AHDL TDFs

You can implement a primitive with an Instance Declaration in the Variable Section of a TDF or with an in-line reference. When you declare an instance of a primitive, the *<net ID>* of that primitive is replaced with the instance name in the SNF. Figure 2 shows a sample TDF in which mydff is declared as an instance of a D flipflop (DFF). The inputs and output of mydff are expressed by appending a period plus *<pinstub name>* to the instance name, i.e., as mydff.d, mydff.clk, mydff.prn, mydff.clrn, and mydff (or mydff.q).

Figure 2. TDF with Instance Declaration and Named Node

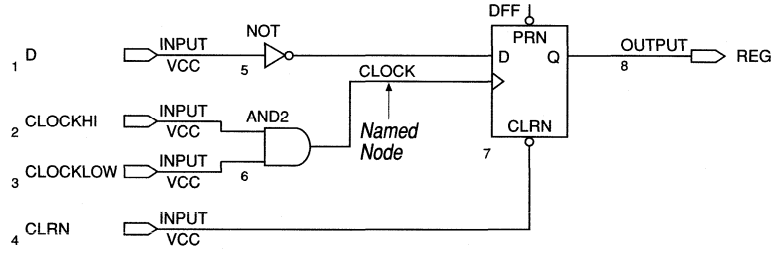
```
TITLE "4-Bit Counter with a 74161 and DFF";
FUNCTION 74161 (a, b, c, d, ldn, enp, ent, clrn, clk)
    RETURNS (rco, qa, qb, qc, qd,)
SUBDESIGN 4_bit
(
    d_in, clk           : INPUT;
    d_out, rco, q[3..0] : OUTPUT;
)
VARIABLE
    mydff    : DFF;
    notclock : NODE;
       ⋮
BEGIN
    notclock = !clk ;
    d_out = mydff.q ;
      % .q stub name is optional on right side of equation %
    mydff.d = d_in ;
      % .d stub name is optional on left side of equation  %
    mydff.clk = notclock ;
    (rco, q[3..0]) = 74161(,,,,,,,,clk);
       ⋮
END;
```

Priority 2: Explicitly Named Nodes

The Priority 3 (lowest-priority) pinstub/port-based name for a node can be replaced by a Priority 2 user-defined node name.

All WDF nodes are named when you create them with the **Enter Node** command (Node menu). In a GDF, you name a node by attaching text to a node (line). Figure 3 shows an updated version of Figure 1, where the node feeding the Clock input to the `DFF` has been named `CLOCK`. This name replaces the Priority 3 names of `:6` and `:7.CLK` in the functional and timing SNF, respectively.

Figure 3. REG_EN.GDF with Named Clock Node



You create a named node in a TDF by declaring a node in the Variable Section. For example, the node `notclock` in Figure 2 is declared in the Variable Section.

Hierarchical Pathnames in Priority 3 & Priority 2 Node Names

Each node in a lower-level file of a hierarchical project has a hierarchical pathname, which traces the node name along a path through multiple levels of the hierarchy. This path includes the *<function name>* and *<net ID>* of each design file (macrofunction) that is higher up in the hierarchy. The net ID has the same function as in a Priority 3 node name, i.e., to distinguish between multiple occurrences of the same item in the same level of hierarchy. The last item in this hierarchy path is a node name in the current file.

The hierarchical path plus node name has the following BNF syntax:

| *<function name>* : *<net ID>* . . . | *<Priority 3 or Priority 2 node name>*

The vertical bar (|) marks the start of a *<function name>*, which consists of the name of a macrofunction or an instance of a macrofunction declared in the Variable Section of a TDF. The colon (:) separates the function name from its *<net ID>*. Projects with multiple levels of hierarchy repeat this pattern, with a vertical bar separating the *<function name>*:*<net ID>* of each

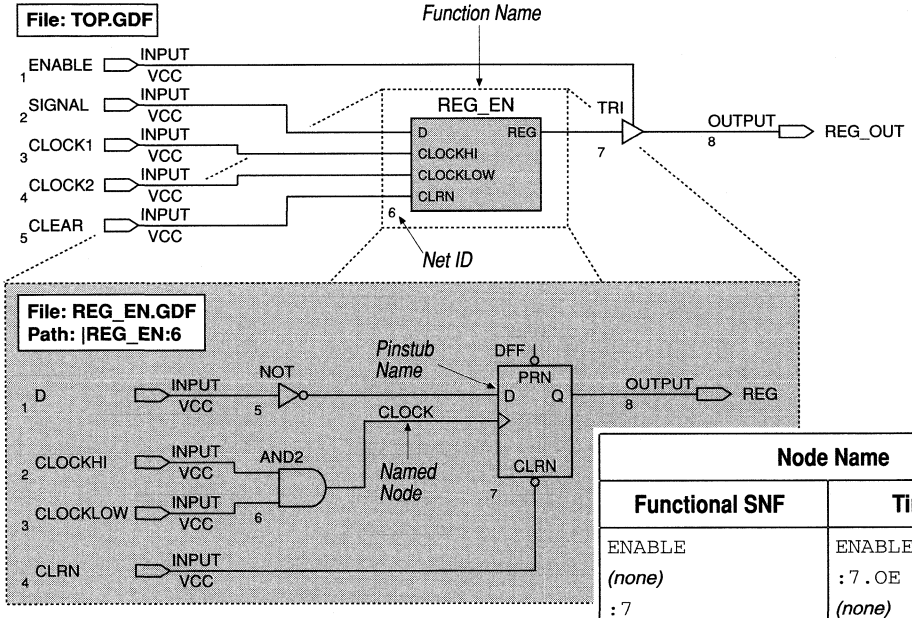
hierarchical level. For example, in Figure 2, you can add the following line to the Variable Section to declare `counter` as an instance of the `74161` counter:

```
counter : 74161
```

The previous Priority 3 (lowest-priority) node name of the output of the counter's first register is `|74161:35|qa`. After assigning the `counter` variable, it is `|counter|qa`.

Figure 4 shows the TOP project, another sample hierarchical project that illustrates how the various types of node names appear in different design files. TOP includes the file REG_EN.GDF shown in Figure 3. The available

Figure 4. TOP Project



Node Name	
Functional SNF	**Timing SNF**
ENABLE	ENABLE
(none)	:7.OE
:7	*(none)*
SIGNAL	SIGNAL
CLOCK1	CLOCK1
CLOCK2	CLOCK2
|REG_EN:6|:7	|REG_EN:6|:7
|REG_EN:6|:5	|REG_EN:6|:7.D
|REG_EN:6|CLOCK	|REG_EN:6|CLOCK
CLEAR	CLEAR
(none)	|REG_EN:6|:7.CLRN
REG_OUT	REG_OUT

nodes and node names differ in the functional and timing SNF. For example, in the functional SNF for the TOP project, the full hierarchical name for the node that feeds the D input of the register is |REG_EN:6|:5, which is the output of the NOT primitive (net ID 5), which in turn is inside the macrofunction REG_EN (net ID 6). In the timing SNF, the same node is renamed with reference to the input pinstub of the register, as |REG_EN:6|:7.D.

Node names in the SNF always include an absolute pathname that starts with a vertical bar. This bar indicates that the first *<function name>* is the top of the hierarchy. When locating or manipulating a node from within a design file, you can use a pathname that is relative to the current file, and omit the initial vertical bar and the names of files further up the hierarchy. For example, you can specify a relative pathname when you search for a node, assign a logic option, or enter a probe name from within the Graphic, Text, or Waveform Editor.

Priority 1: Probes

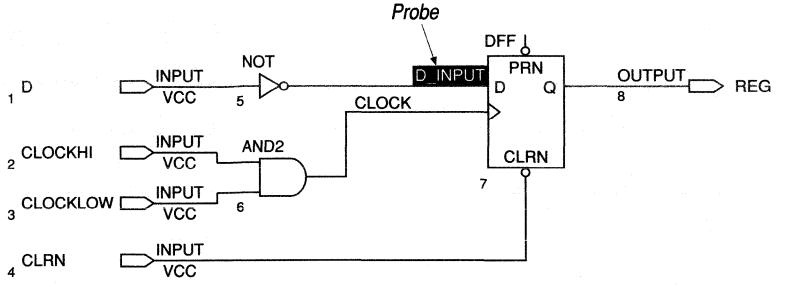
You can replace the entire hierarchical path and name of a lower-priority (Priority 3 or Priority 2) node name with a single user-defined probe name. With the **Probe** or **Enter Assignments** command (Assign menu) in the Graphic, Text, or Waveform Editor, you can enter a probe on a node at any level in the hierarchy. In a GDF, you can attach a probe to an input or output pinstub of any symbol. Probes are displayed in the schematic if the **Show Probes** command (Options menu) is turned on.

For example, to place a probe on the D input to the register in REG_EN.GDF:

1. Open TOP.GDF, the top-level file in the project, and double-click Button 1 on the REG_EN symbol to open the file REG_EN.GDF.
2. Click Button 1 on the D pinstub of the DFF symbol. The symbol and the pinstub are selected.
3. Choose the **Probe** command (Assign menu). The **Probe** dialog box shows :7.D, the relative hierarchical name for the node (i.e., within the current file), which will be replaced by the probe name.
4. Type a probe name in the dialog box and choose **OK**.

MAX+PLUS II substitutes the probe name for the existing lower-priority name in the Probe & Resource Assignment File (.PRB) as an alias for the hierarchical path and node name. Figure 5 shows how the probe D_INPUT is placed on the D input to the register in REG_EN.GDF (part of the TOP project), and is visible in the schematic. In a functional SNF, this name replaces the full hierarchical node name |REG_EN:6|:5; in a timing SNF, it replaces the node name |REG_EN:6:|7.D.

Figure 5. REG_EN.GDF with Probe



Node Name		Probe Name
Functional SNF	**Timing SNF**	
|REG_EN:6|:5	|REG_EN:6|:7.D	D_INPUT
|REG_EN:6|CLOCK	|REG_EN:6|CLOCK	D_INPUT.CLK
|REG_EN:6|:7	|REG_EN:6|:7	D_INPUT.Q
(none)	|REG_EN:6|:7.CLRN	D_INPUT.CLRN

Probes have another useful feature: if you place a probe on a primitive, you can use the probe name as a base name for accessing the other inputs and outputs to a primitive. You simply append a period (.) plus the appropriate *<pinstub name>*, as shown in Table 1, of another input or output. The table in Figure 5 shows probe name variations and equivalent node names for the DFF inputs and outputs.

For more information on internal node name syntax, see "Hierarchical Node & Symbol Names" under *Hierarchy Display Basic Tools* in MAX+PLUS II Help.

Simulating State Machines

Since state machines often control important segments of a project, simulating state machines is especially important. You may need to simulate one or more items in a state machine, the most common of which are listed below:

- State machine inputs
- Decoded outputs
- State register bits
- Present states

State Machine Inputs

State machine inputs feed combinatorial logic, which in turn feeds the D inputs to state bits (state registers). However, state machine inputs appear in a timing SNF only when they are fed by EPLD input pins or other hard logic nodes. If the state machine inputs are fed by combinatorial logic, they appear in the functional SNF only, and are accessible only in a functional simulation.

If you wish, you can simulate the inputs to state registers. See "State Bits" later in this application brief.

Decoded Outputs

Decoded outputs are also combinatorial, and appear in a timing SNF only if they feed output or bidirectional pins or other hard logic nodes. Only functional simulation is possible if the nodes do not feed hard logic nodes.

State Bits

In a WDF, you can create a node that represents a state machine. The Compiler automatically creates and names state bits (the outputs of state registers). In an AHDL TDF, state bits are optionally assigned names in the State Machine Declaration in the Variable Section. The syntax of this declaration is:

<state machine name> : MACHINE [OF BITS (*<state bit names>*)]
WITH STATES (*<state names>*) ;

After compilation, the state bits in the SNF are automatically combined into a group (bus) with the same name as the state machine, regardless of whether you named them in the original design file. Therefore, the default SCF you create with the **Create Default Channel** command in the Waveform Editor automatically contains a group with the same name as each state machine in the project. (A Table File generated from this SCF also contains these groups.) For example, Figure 6 shows 4_STATE.TDF, which includes the statemach state machine. The SNF for the 4_STATE project contains the group statemach, which consists of the state bits q1 and q0.

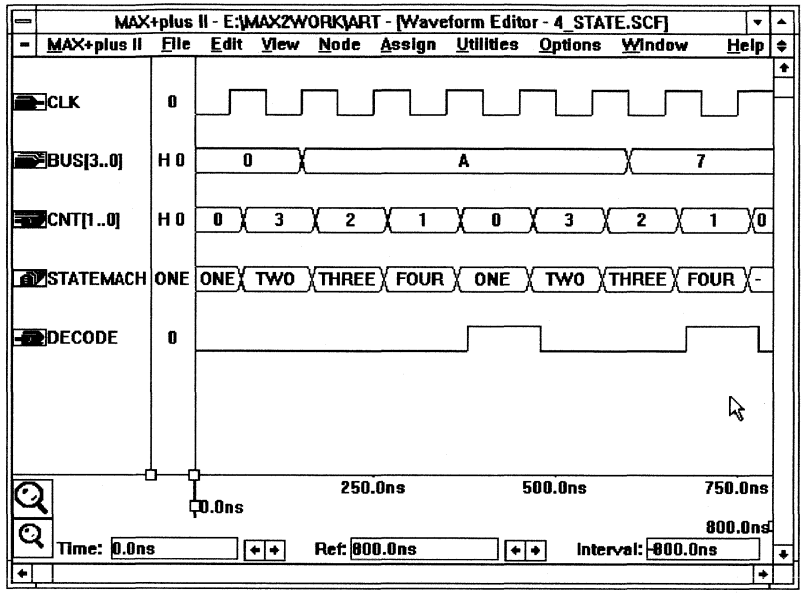
Figure 6. Sample State Machine (4_STATE.TDF)

```
TITLE "A Simple State Machine";

DESIGN IS 4_state;
SUBDESIGN 4_state
(
    clk, bus[3..0]    : INPUT;
    cnt[1..0], decode : OUTPUT;
)
VARIABLE
    statemach : MACHINE OF BITS (q[1..0])
                WITH STATES
                   (one, two, three, four);
BEGIN
    statemach.clk = clk;
    cnt[ ] = q[ ];

    CASE statemach IS
        WHEN one =>
            statemach = two;
            decode = (bus[] == H"A");
        WHEN two =>
            statemach = three;
        WHEN three =>
            statemach = four;
        WHEN four =>
            statemach = one;
            decode = (bus[] == H"7");
    END CASE;
END;
```

State machine design entry in TDFs and WDFs does not require you to fully define the value of each bit for each state. The MAX+PLUS II Compiler automatically creates state assignments. When you allow the Compiler to create state assignments, it is especially useful to simulate the "state machine groups" rather than the individual state bits.

The Waveform Editor displays state names rather than numerical group values in state machine group waveforms if the logic levels of the state bits correspond to a known state name. You can also enter numerical values or state names when you edit state machine group waveforms in an SCF. Figure 7 shows an SCF with the simulation waveforms for the 4_STATE project. Even if the Compiler-generated state bit names change when you recompile a project, the Simulator can automatically update state machine groups in the SCF or Vector File to include the new names.

Figure 7. 4_STATE.SCF



You can also simulate the inputs to state registers by appending the `.D`, `.CLRN`, and `.CLK` pinstub names to the state bit name, in the same way that you append a pinstub name to a probe name. The `.CLRN` input retains the active-high logical sense of the Reset input to the state machine.

If you declare a state bit but do not fully define the value of each bit for each state, functional and timing simulation results can differ in cases where no state value is assigned to a state bit for one or more states. To ensure accurate results, you should always perform timing simulation on a state machine.

Present States

You can determine the present state of a state machine by simulating state machine groups, as described in "State Bits." In addition, a functional SNF includes buried nodes with state names that can be included in your SCF or Vector File. These are not real nodes; however, when the waveform of a node with a state name is high, that state is the current state.

You can use one of the following three methods to create an SCF or Vector File that includes state machine groups for a functional or timing SNF. After simulation, the SCF generated by the Simulator shows the present state of the state machine in the group waveform.

- In the Waveform Editor, use the **Create Default Channel** command (File menu) to create a default SCF that automatically includes the state machine group, as described earlier in this application brief.
- If you have an existing SCF and wish to add a state machine group, use the *Update Existing Channel* option in the Waveform Editor's **Create Default Channel** dialog box. You can also use the **Enter Node** command (Node menu) to add individual state bit nodes, then use the **Enter Group** command (Node menu) to combine them into a group with the same name as the state machine.
- In the Text Editor, enter the state machine name in the appropriate section (Inputs, Outputs, or Buried) of a Vector File, enter state names for simulation in a Pattern Section, then import the file into the Waveform Editor with the **Import Vector File** command (File menu). No Group Create Section is required in the Vector File for a state machine that was created in the project's design files.

Refer to MAX+PLUS II Help for information on editing node and group waveforms.

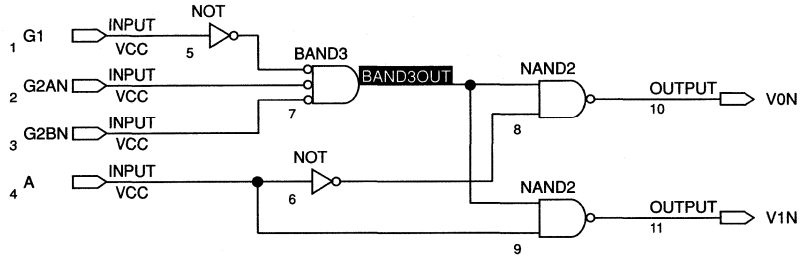
Simulating Combinatorial Nodes

When you fully compile a project, the MAX+PLUS II Compiler uses SALSA (Speedy Altera Logic Simplifying Algorithm) and De Morgan's inversion to remove redundant logic and minimize the number of product terms. During this process, combinatorial logic is often reduced or changed so that the timing SNF for the project does not contain the same combinatorial nodes as the original design files or the functional SNF.

If you wish to simulate buried combinatorial logic, you should use a functional SNF, which retains all nodes in the original design files.

Figure 8 shows DECODE.GDF, a combinatorial circuit for a decoder with three enable lines called G1, G2AN, and G2BN. To activate the outputs, all three of the enable lines feeding the BAND3 gate must be active. To functionally simulate the combinatorial logic in this project, you can monitor node :7, the output of the BAND3 gate. In Figure 8 the probe BAND3OUT has been attached to the output of BAND3 to replace the :7 name.

Figure 8. Sample Combinatorial Circuit (DECODE.GDF)



Finding Nodes

After you have listed all nodes that can be simulated in an SCF, Table File, or History File, you may wish to locate them in the design files. The **Find Node/Symbol** command (Utilities menu) in the Graphic, Text, and Waveform Editors can locate a node or symbol in any file within the current hierarchy, including node names that have been replaced with probe names.

To find a node in a design file:

1. Choose the **Find Node/Symbol** command (Utilities menu).
2. To find a node in the current file:

   Type the node name or **:***<net ID number>***.***<pinstub name>*. This name is automatically appended to the hierarchy path that leads to the current file, which is displayed in the **Find Node/Symbol** dialog box for your reference. If you omit the **.***<pinstub name>*, this command locates a primitive or macrofunction instance name or symbol.

   To find a node in a different file:

   Type the hierarchy path to that file, followed by the node name or **:***<net ID number>***.***<pinstub name>*.

   To find a node that has been renamed with a probe name:

   Type the probe name. No hierarchy path is needed.

3. Choose **OK**.

MAX+PLUS II automatically opens the file that contains the node and highlights it.

For example, if the current project is TOP (as shown in Figure 4) and you choose **Find Node/Symbol** while REG_EN.GDF is displayed, the dialog box for the command displays |REG_EN:6, which is the hierarchy path that leads to the current file. Type :6 and choose **OK** to find and highlight the AND2 gate and its output node in REG_EN.GDF.

Using Internal Nodes in Simulation

You can quickly place nodes into an SCF or Vector File for simulation with the **Create Default Channel** and **Create Table File** commands described earlier in this application brief. The logic levels in these default files can be easily edited to provide the inputs for simulation. SCFs are especially useful if you create logic in a WDF, since you can copy the waveforms of inputs from the WDF into the SCF to create the simulation inputs. Figure 7 shows an SCF for the 4_STATE project; Figure 9 shows a sample Vector File for the DECODE project.

Figure 9. DECODE.VEC

This file can be created by editing a Table File generated in the Waveform Editor.

```
INPUTS G2BN G2AN G1 A ;
OUTPUTS Y1N Y0N ;
BURIED BAND3OUT ;
UNIT ns ;
RADIX HEX ;
PATTERN
   0.0>  0 0 0 0
1000.0>  0 0 0 1
2000.0>  0 0 1 0
3000.0>  0 0 1 1
4000.0>  0 1 0 1
5000.0>  0 1 0 0
6000.0>  0 1 1 1
7000.0>  0 1 1 0
8000.0>  1 0 0 1
9000.0>  1 0 0 0
10000.0> 1 0 1 1
11000.0> 1 0 1 0
12000.0> 1 1 0 1
13000.0> 1 1 0 0
14000.0> 1 1 1 1
15000.0> 1 1 1 0 ;
```

Conclusion

Simulation with MAX+PLUS II makes verifying logic quick and simple. The MAX+PLUS II Waveform Editor automatically generates waveform or text files that list all nodes in the functional SNF or timing SNF for the project. You can easily edit both types of automatically generated files to create simulation inputs. Functional simulation offers access to all nodes, including combinatorial and state machine logic. Timing simulation is available for the fully synthesized project. You can use any of three types of

names to identify nodes: the port/pinstub-based node names and explicitly named nodes clearly show the hierarchical path; probes offer shorter, more functional names. This flexibility allows you to choose the best method for identifying and simulating internal nodes, and to perform a thorough simulation in the shortest possible time.

Partitioning a Project
with MAX+PLUS II Software

April 1992, ver. 1 **Application Brief 93**

Introduction

MAX+PLUS II has eliminated the design size barrier by allowing you to create large designs (called "projects" in MAX+PLUS II) without regard to the capacity of a particular EPLD. The Partitioner module of the MAX+PLUS II Compiler divides large projects that cannot fit into a single device into multiple devices from the same EPLD family, either automatically or according to your specifications. To guide partitioning, you can make pin and buried macrocell assignments and group the logic for speed-critical paths. After compilation, you can view the results of partitioning, make any necessary changes, and recompile the design. This iterative approach speeds the development of even the most complex projects.

This application brief discusses the following topics:

- ❑ Types of partitioning
- ❑ Assignments with menu commands
- ❑ Assignments in an Altera Hardware Description Language (AHDL) Text Design File (.TDF)
- ❑ Assignment priority
- ❑ Board-level assignments
- ❑ Project processing & simulation
- ❑ Back-annotating, preserving & editing assignments

Other factors, such as logic option assignments and logic minimization, can affect how the project is synthesized and partitioned. For complete, up-to-date information on partitioning and logic synthesis, refer to MAX+PLUS II Help.

Types of Partitioning

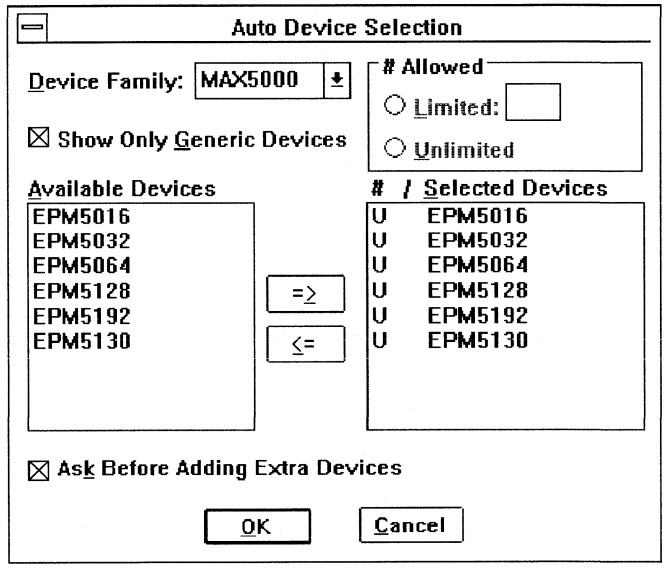
You can partially or fully specify how your project should be partitioned, or the MAX+PLUS II Compiler can automatically partition logic.

Automatic Partitioning

Automatic partitioning is the simplest way to partition projects that do not fit into a single device. The Compiler automatically partitions a project if you do not make any device assignments. Automatic partitioning also occurs if you specify AUTO as the device in the Design Section of a top-level TDF or with the **Device** command in the top-level design file.

The Compiler's **Auto Device Selection** command allows you to specify the device family for your project. See Figure 1. The Compiler automatically chooses one or more device(s) from the specified family to fit your project. For additional control, you can specify the speed grade and for the device(s) to generate specific timing information for simulation and timing analysis and the package to generate the correct pin-out in the Report File (.RPT).

Figure 1. Dialog Box for Auto Device Selection Command



If you turn on the *Ask Before Adding Extra Devices* option in the **Auto Device Selection** dialog box and your project does not fit into the specified EPLD(s), the Compiler opens the **Override User Assignments** dialog box during compilation. You can then instruct the Compiler to ignore different types of assignments or add additional EPLDs so it can adjust the project fit during compilation.

The following steps direct the Compiler to automatically select devices:

1. In the Graphic, Text, or Waveform Editor, set the device type to AUTO or leave it unspecified in the design file.
2. In the Compiler, choose **Auto Device Selection** (Options menu).
3. Choose an EPLD family from the *Device Family* drop-down list box. All EPLDs in the selected family appear in the *Available Devices* box.
4. (Optional) Turn on the *Show Only Generic Devices* option to show only device names without package types, temperature ratings, and speed grades in the *Available Devices* box.

5. (Optional) To add EPLDs to the list of selected devices, double-click Button 1 on a single name in the *Available Devices* box or select one or more EPLD names and choose the right direction button (=>).
6. (Optional) To remove EPLDs from the list of selected devices, double-click Button 1 on a single name in the *#/Selected Devices* box or select one of more EPLD names and choose the left direction button (<=).
7. (Optional) Select one or more EPLDs from the *#/Selected Devices* box and select the *Limited* or *Unlimited* option under *# Allowed.* If you select *Limited,* type a number from 1 to 99 in the text box to limit the number of EPLDs that can be used by the Compiler to fit a project.
8. (Optional) Turn on the *Ask Before Adding Extra Devices* option to direct the Compiler to ask before adding devices during partitioning.
9. Choose **OK**. The settings are saved in the *<project name>*.INI file.

User-Guided Partitioning

You can direct the Compiler to partition a project according to your specifications by making resource (including pin, macrocell, chip, and clique) and device assignments in the top-level design file in the Graphic, Text, or Waveform Editor, or in the Design Section of a top-level TDF. (Logic options are also considered resource assignments, but are not discussed in the application brief.)

To guide the Compiler's Partitioner module, you can:

- Assign nodes in a project to particular pins or macrocells in one or more chips.
- Make chip assignments to specify which logic blocks must be placed in the same device.
- Make clique assignments to specify which logic blocks are to be placed in the same device, and in the same Logic Array Block (LAB) when possible.
- Assign one member of a clique to a chip or to a macrocell or pin within a chip. This type of assignment effectively assigns the entire clique to a chip.
- Map the logic blocks you have previously defined with chip assignments to specific devices.
- Define logic for individual devices as separate projects, place all the symbols in a single schematic, and use the connected-pins feature to specify which device pins are connected together on your board.

Assignments with Menu Commands

You can make resource and device assignments in the top-level design file with the **Device, Enter Assignments, Clique,** and **Pin/MC/Chip** commands (Assign menu) in the Graphic, Text, or Waveform Editor. You can also make them in the Design Section of a top-level TDF. However, you should not use both assignment methods in a single project.

You can place resource assignments on any logic function. However, assignments are only retained on hard logic functions, i.e., functions that are not removed during logic synthesis. Hard logic functions include SOFT and TRI buffers not removed by logic synthesis, and INPUT, OUTPUT, BIDIR, MCELL, LATCH, and all flipflop primitives. Assignments on macrofunctions are retained only on hard logic functions within the macrofunction logic. Assignments on primitives within a macrofunction override any assignment on the macrofunction as a whole. The Compiler automatically places any logic not assigned.

You can enter the following types of assignments:

- Pins
- Macrocells
- Chips
- Cliques
- Devices

Pin, Macrocell & Chip Assignments

You can assign a single hard logic function to a specific pin or macrocell within a chip, and assign one or more hard logic functions to a specific chip.

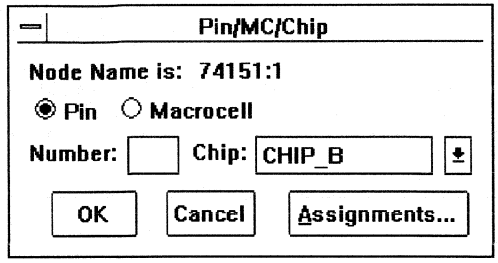
Assigning a node to a particular pin ensures that the signal is always associated with that pin, regardless of future changes to the project. Assigning logic to a specific macrocell can minimize the number of signal breaks and eliminate unnecessary timing delays. Assigning logic to a chip keeps a group of hard logic functions together in one device when a project is partitioned into multiple devices. Chip assignments minimize the number of signals that travel between devices and ensure that no unnecessary device-to-device delays occur on critical timing paths.

You can enter pin, buried macrocell, and chip assignments in the top-level design file with either the **Pin/MC/Chip** or **Enter Assignments** command in the Graphic, Text, or Waveform Editor, or in the Design Section of a top-level TDF.

To enter pin, macrocell, and chip assignments with the **Pin/MC/Chip** command, follow these steps:

1. (Optional) Select a single hard-logic node or symbol for a pin or macrocell assignment, or one or more items for a chip assignment.
2. Choose **Pin/MC/Chip** from the Graphic, Text, or Waveform Editor's Assign menu or double-click Button 1 on a pin primitive in the Graphic Editor to open the **Pin/MC/Chip** dialog box. See Figure 2.

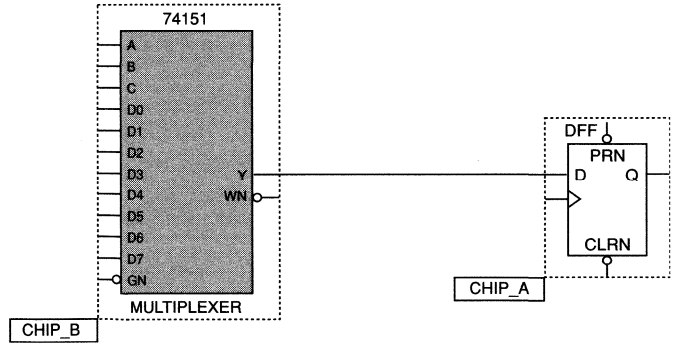
Figure 2. Dialog Box for Pin/MC/Chip Command



3. (Single node or symbol only) Select *Pin* or *Macrocell* and type a number in the *Number* box. All existing assignments are listed in the drop-down list box.
4. Type a chip name in the *Chip* box or select an existing chip name from the drop-down list box. The chip name is optional for a project that fits into a single EPLD (the chip name defaults to the project name). The current pin or macrocell number is assigned to the specified chip.
5. (Optional) If you wish to enter additional resource assignments, choose the **Assignments** button to open the **Enter Assignments** dialog box.
6. Choose **OK** to save all assignment changes, which are stored in the Probe & Resource Assignment File (.PRB) for the project.

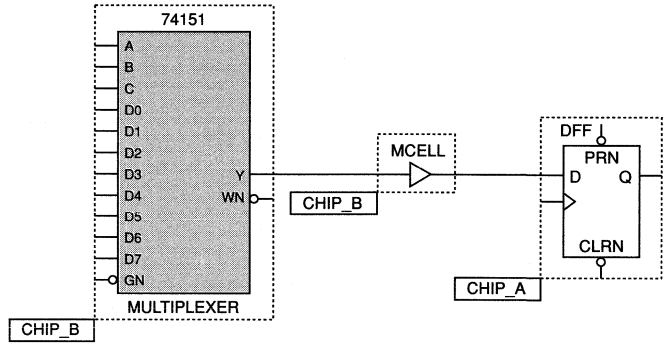
If **Show Pins/MCs/Chips** or **Show All Assignments** is turned on in the Graphic Editor, a graphic representation of the pin, macrocell, or chip assignment appears on any symbol with an assignment. Figure 3 shows an example of chip assignments in a Graphic Design File (.GDF). The `74151` macrofunction is assigned to `CHIP_B`; the `DFF` primitive is assigned to `CHIP_A`. `CHIP_A` and `CHIP_B` can be mapped to specific EPLDs with the **Device** command or you can allow the Compiler to choose devices automatically.

Figure 3. Chip Assignments in a Graphic Design File



However, since the 74151 macrofunction contains no hard logic functions, the chip assignment for it is meaningless, and the macrofunction is placed in the same chip as the DFF primitive. Consequently, CHIP_A is fed by all 12 input signals and not by the single signal specified in the chip assignment. To ensure that the 74151 logic is assigned to CHIP_B, you must add an MCELL buffer with an assignment as shown in Figure 4. With this change, only the output of the MCELL feeds CHIP_B.

Figure 4. Chip Assignment on an MCELL Buffer



Clique Assignments

A clique is a group of hard logic functions that are always kept in the same chip. In addition, if a project is partitioned into one or more devices that have multiple Logic Array Blocks (LABs), cliques are placed in the same LAB when possible, optimizing logic placement for high-speed operation. Clique assignments minimize the number of signals that travel between LABs and between devices and ensure that there are no unnecessary LAB-to-LAB or device-to-device delays on critical timing paths.

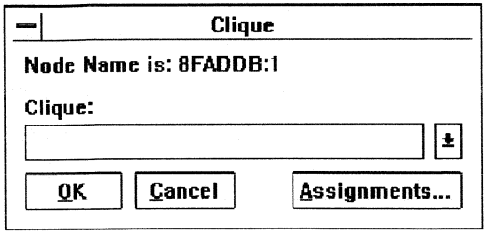
The MAX+PLUS II Compiler automatically assigns a clique to a chip and the corresponding device. Since all members of a clique are fitted into the same device, assigning one clique member to a chip effectively assigns the entire clique to the device.

You can enter a clique assignment with either the **Clique** or **Enter Assignments** command in the Graphic, Text, or Waveform Editor, or in the Design Section of a top-level TDF.

To enter an assignment with the **Clique** command, follow these steps:

1. Select one or more nodes and/or symbols that represent hard logic.
2. Choose **Clique** from the Assign menu in the Graphic, Text, or Waveform Editor to open the **Clique** dialog box. See Figure 5.

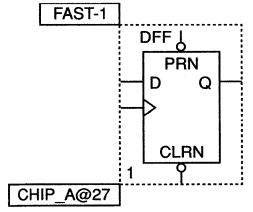
Figure 5. Dialog Box for Clique Command



3. Type a clique name in the *Clique* box, or select an existing name from the drop-down list box to add the selected hard logic to a previously defined clique.
4. (Optional) If you wish to enter additional resource assignments, choose the **Assignments** button to open the **Enter Assignments** dialog box.
5. Choose **OK** to save all assignment changes to the project's .PRB file.

If **Show Cliques** or **Show All Assignments** is turned on in the Graphic Editor, the name of the clique appears next to the top left corner of any symbol assigned to the clique. Figure 6 shows clique and chip assignments on a D flipflop.

Figure 6. Clique and Chip Assignments on a D Flipflop (DFF)



Device Assignments

A device assignment maps the logic blocks defined by chip assignments to specific EPLDs. You can use a device assignment to assign all logic in a single-EPLD project to a particular device, or to map all chip assignments in a multi-EPLD project to specific devices. You can also set the Security Bit option for each device, and set the Turbo Bit for devices with that option.

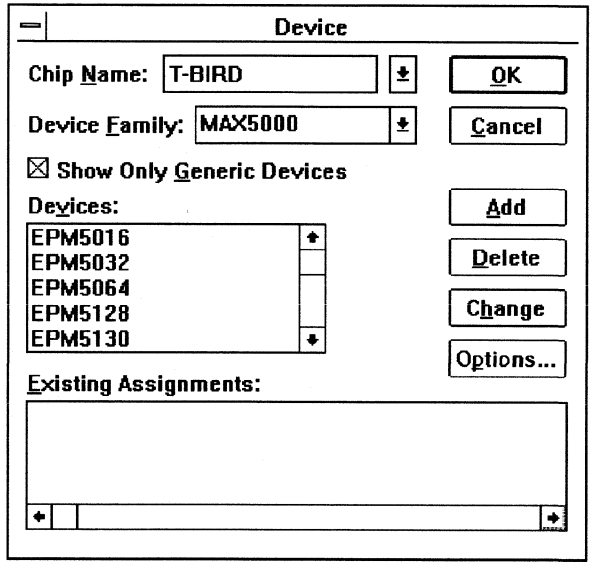
If you do not make device assignments, the MAX+PLUS II Compiler uses information in the Compiler's **Auto Device Selection** dialog box to select the target EPLD(s). Each chip can be assigned to only one EPLD. If the logic does not fit into the specified EPLD, compilation halts, and the **Override User Assignments** dialog box appears.

You can enter a device assignment with the **Device** command in the Graphic, Text, or Waveform Editor or in the Design Section of a top-level TDF.

To assign a chip in the current project to a device, follow these steps:

1. Choose **Device** from the Assign menu in the Graphic, Text, or Waveform Editor to open the **Device** dialog box. See Figure 7.

Figure 7. Dialog Box for Device Command



2. Type the name of a previously assigned chip in the *Chip Name* box, or select an existing name from the drop-down list box.
3. Select a device family from the *Device Family* drop-down list box.
4. (Optional) Turn on the *Show Only Generic Devices* option to show only device names without package types, operating temperatures, and speed grades in the *Available Devices* box.
5. Select a device from the *Devices* box.
6. (Optional) Choose the **Options** button to open the **Device Options** dialog box and specify the setting for the Turbo Bit and Security Bit.
7. Choose the **Add** button to add the new device assignments to the *Existing Assignments* box. You must choose **Add** to add a new device assignment to the project: only the assignments listed in the *Existing Assignments* box are saved when you choose **OK**.
8. Choose **OK** to save all assignment changes to the project's .PRB file.

Assignments in an AHDL TDF

The Altera Hardware Description Language (AHDL) is completely integrated into MAX+PLUS II. You can enter your TDFs with the MAX+PLUS II Text Editor or with any standard text editor. You can then compile, simulate, and program your projects from within MAX+PLUS II. AHDL supports various features, including Boolean equation, state machine, and truth table design.

You can include the optional Design Section in a top-level TDF to specify pin, buried macrocell, chip, clique, and device assignments and connected pins. The Compiler ignores the Design Section unless it is at the top level of the project hierarchy. Alternatively, you can enter assignments, except for connected pins, with the Assign menu commands in a TDF, as described earlier in this application brief. You should not, however, use both methods within a single project.

You can use the following statements to partition a project in a TDF:

- Resource Assignment Statement for pin, macrocell, and chip assignments
- Clique Assignment Statement for clique assignments
- EPLD Specification for device assignments

Resource Assignment Statement

The optional Resource Assignment Statement, which can be repeated one or more times, assigns nodes in the project to particular pins or macrocells in one or more chips. The pin type can be identified with the `INPUT`, `OUTPUT`, or `BIDIR` assignment statements. Buried macrocells are identified using `BURIED`. Macrofuctions in which hard logic functions are to be grouped in the same chip are identified with the `MACRO` assignment statement. The following TDF excerpt shows a Resource Assignment Statement that specifies pin and macrocell assignments for the chip `u1`:

```
DEVICE u1 IS EPM5032
BEGIN
    in_1 @ 9, in_2 @ 10, in_3 @ 11     : INPUT;
    out7 @ 12                          : OUTPUT;
    |74163:5|QA @ mc10                 : BURIED;
END;
```

The "at" symbol (`@`) separates the pin or macrocell number from the hierarchical node name. Additional hierarchical node names and pin or macrocell numbers are separated by commas (`,`). The last pin or macrocell number is followed by a colon (`:`), which is followed by the pin or macrocell type. In this example, the `QA` output of a `DFF` within the `74163` macrofunction is a buried macrocell feedback placed at macrocell 10 (`mc10`).

Clique Assignment Statement

The optional Clique Assignment Statement, which can be repeated one or more times, allows you to group hard logic functions. A clique always keeps the logic in the same chip. In addition, if the Compiler fits a project into one or more devices that have multiple LABs, cliques are placed in the same LAB when possible, optimizing logic placement for high-speed operation. A Clique Assignment Statement defines each group of nodes and/or macrofunctions that must remain together.

A Clique Assignment Statement has the same format as a Resource Assignment Statement except that it is preceded by the keyword CLIQUE. However, you cannot assign pins and macrocells in a Clique Assignment Statement. The following TDF excerpt shows a Clique Assignment Statement that assigns x_stepper and out7 to the clique x_coord:

```
CLIQUE x_coord
BEGIN
    |x_stepper        : MACRO;
    out7             : OUTPUT;
END;
```

The keyword CLIQUE is followed by a unique clique name, followed by the nodes and macrofunctions to be assigned to the clique, enclosed within BEGIN and END keywords.

EPLD Specification

The EPLD Specification allows you to assign a chip to a device. The chip name is optional for a project that fits into a single EPLD. The following TDF excerpt assigns the chip u1 to an EPM5032 EPLD:

```
DEVICE u1 IS EPM5032;
```

The keyword DEVICE is followed by a chip name, which is followed by the keyword IS and the device name. The device name is the target Altera EPLD. The chip name and the device name are optionally enclosed in double quotation marks ("). The device name can also specify package, temperature, and speed grade (e.g., EPM5032JC-1).

Automatic device selection is the default if you do not include an EPLD Specification, but you may also specify AUTO (for automatic device selection) instead of an EPLD name. If you specify AUTO, you should specify a device family with the Compiler's **Auto Device Selection** command. A semicolon (;) ends the EPLD Specification.

Sample AHDL Text Design File

Figure 8 shows an AHDL TDF for a multi-device project. This example includes Resource Assignment Statements, Clique Assignment Statements, and EPLD Specifications.

Figure 8. Sample TDF with Resource and Device Assignments

```
DESIGN IS xycntrlr
BEGIN
    DEVICE u1 IS EPM5032
    BEGIN
        in_1 @ 9, in_2 @ 10, in_3 @ 11     : INPUT;
        out7 @ 12                          : OUTPUT;
        |74163:5|QA @ mc10                 : BURIED;
        |74163:5|QB @ mc11                 : BURIED;
    END;

% The next EPLD specification includes clique assignments. %

    DEVICE u2 IS AUTO
    BEGIN
        num1, num2, num3, num4             : INPUT;
        out1, out2, out3, out4             : OUTPUT;
        |74161                             : MACRO;
    END;

    CLIQUE x_coord
    BEGIN
        |x_stepper                         : MACRO;
        out7                               : OUTPUT;
    END;

    CLIQUE y_coord
    BEGIN
        |y_stepper                         : MACRO;
        out1                               : OUTPUT;
    END;
END;
```

The TDF in Figure 8 has the following characteristics:

- The keywords `DESIGN IS` must be the first words in the Design Section. They are followed by the design name, which must be the same as the project name. The keywords `BEGIN` and `END` enclose the EPLD Specifications, Resource Assignment Statements, and Clique Assignment Statements.
- The `out7` signal is in chip `u1`, so all logic in the clique `x_coord` is assigned to chip `u1`. Similarly, since `out1` is in chip `u2`, all logic in

clique y_coord is assigned to chip u2. Chip u2 is specified as AUTO, so the Compiler fits all logic in chip u2 into the smallest possible EPLD specified in the Compiler's **Auto Device Selection** dialog box.

- Chip u2 must be assigned to the same EPLD family as chip u1 because MAX+PLUS II logic synthesis is architecture-dependent. The Compiler must assign all logic in a multi-device project to EPLDs from the same family.

- MACRO specifies that all hard logic functions in the entire hierarchy of the macrofunction are included in the chip or clique, rather than just a node. When a macrofunction is assigned to a clique or chip, all hard logic functions for the macrofunction are included in that clique or chip assignment. However, you can enter different assignments on individual hard logic functions within the macrofunction to override the MACRO assignment for the macrofunction as a whole.

Assignment Priority

The MAX+PLUS II Compiler prioritizes resource and device assignments to avoid conflicts. An assignment with a higher priority always overrides an assignment with a lower priority. The Compiler accepts resource and device assignments in the following order:

Priority 1	Top-level TDF
Priority 2	Probe & Resource Assignment File (.PRB), which contains assignments made with Assign menu commands in the Graphic, Text, and Waveform Editors
Priority 3	Fit File (.FIT) from the last successful compilation

If the Compiler cannot find any assignments, it places the logic automatically. You can copy the assignments from the Fit File into the Design Section of a TDF to preserve the current assignments in future compilations.

Board-Level Assignments

The optional AHDL Pin Connection Statement (keyword CONNECTED_PINS) allows you to simulate hard-wire pin connections, i.e., connections on external device pins. A Pin Connection Statement allows you to connect a single output of a device to inputs on the same device or on other devices. You can only place connected pin assignments on INPUT, OUTPUT, or BIDIR primitives that correspond to EPLD pins in the top-level design file of the project.

The Pin Connection Statement only affects simulation; it does not affect compilation.

Although the connected-pins feature is available only in TDFs for MAX+PLUS II version 2.11, it can be used in conjunction with any type of top-level design file.

You can use a Pin Connection Statement for the following purposes:

- To tie inputs and outputs of top-level macrofunctions together. Connected pins are useful for the following reasons:
  - Projects can be easily partitioned on macrofunction boundaries.
  - You can combine several completed functions into a single project to perform timing simulation. You can enter a symbol for each function in a top-level GDF, or the name of each function in a top-level TDF, and connect corresponding pins through the Pin Connection Statement. The example in Figure 9 shows a `74151` macrofunction with the output `Y` and a `DFF` primitive with the input `D`.

Figure 9. Project with Connected Pins

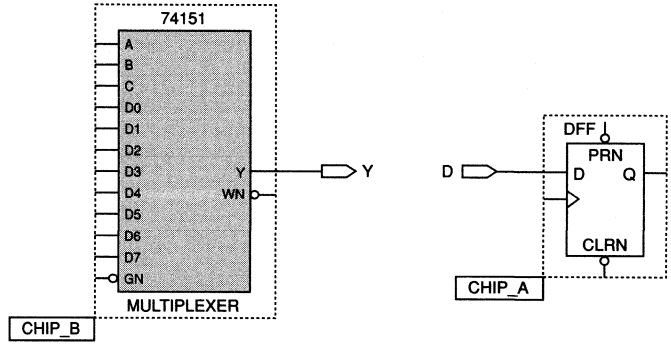


Figure 10 shows the Pin Connection Statement that connects the output of the macrofunction to the input of the `DFF` primitive in Figure 9.

Figure 10. Pin Connection in a TDF

```
DESIGN IS top1
BEGIN
    CONNECTED_PINS
    BEGIN
        d       : INPUT;
        y       : OUTPUT;
    END;
END;
```

- To tie multiple bidirectional pins together into a bidirectional (tri-state) bus. The Simulator can easily simulate the behavior of the bidirectional bus and detect any logic contention errors.

- To generate Global Clock and Output Enable signals within a project. For example, a Clock can be created by one device and used on another device. These signals can be routed from the output pin of one device and into the appropriate input pin of another device.

The keyword CONNECTED_PINS, followed by the name and type of the pins to be tied together, is enclosed by the keywords BEGIN and END. You must use a separate Pin Connection Statement for each set of pins. Figure 11 shows multiple pin connections, including a bidirectional bus.

Figure 11. Multiple Pin Connection Statements Including a Bidirectional Bus

```
DESIGN IS top2
BEGIN
    CONNECTED_PINS
    BEGIN
        in1                         : INPUT;
        bidir1, bidir2, bidir3      : BIDIR;
    END;

    CONNECTED_PINS
    BEGIN
        d                           : INPUT;
        y                           : OUTPUT;
    END;
END;
```

Project Processing & Simulation

The Partitioner module of the Compiler uses the fully synthesized project database to determine the number of devices required to fit the project. The Partitioner first allocates any logic associated with assignments that identify critical timing paths (i.e., clique assignments), device types, and package configurations. All other logic is then placed in the remaining EPLD resources. The Partitioner can automatically add more devices for projects that are too large to fit into the devices that you have specified.

The Partitioner splits project logic into as few EPLDs as possible to minimize the number of connections between EPLDs, and to maximize resource utilization. Since the logic is partitioned along macrocell boundaries, unassigned macrocells may be moved among devices to obtain the optimum fit. The Partitioner does not change logic to achieve a fit, since macrocell functionality is determined by logic synthesis prior to partitioning.

The results of partitioning are passed to the Fitter module, which fits the partitioned logic into the specified EPLDs. Macrocells are arranged to minimize routing and timing based on user assignments and heuristic fitting algorithms. The Fitter generates a Report File (.RPT) and a Fit File that report the results of partitioning. You can use the Fit File to preserve resource assignments for future compilations (see "Back-annotating,

Preserving & Editing Assignments" in this application brief). If the Fitter cannot fit the partitioned logic into the EPLD(s), the project database is returned to the Partitioner for additional processing. This process repeats until the entire project fits.

MAX+PLUS II can also simulate partitioned projects. During timing simulation, multi-device projects are treated as a single project, with the device-to-device delays incorporated into the timing parameters for each device. During compilation, the Compiler generates a functional or timing Simulator Netlist File (.SNF) for the project that can be used by the MAX+PLUS II Simulator for complete functional or timing simulation. Graphical Simulator Channel Files (.SCF) and text-based Vector Files (.VEC) can provide the input stimulus for the simulation.

Back-Annotating, Preserving & Editing Assignments

Back-annotation copies resource and device assignments in the compiled project back into the original design files for the project. This process ensures that subsequent compilations produce the same fit.

Device and resource assignments created with the **Pin/MC/Chip**, **Clique**, **Device**, and **Enter Assignments** commands in the Graphic, Text, and Waveform Editors are stored in the .PRB file for the project. Device and resource assignments can also be made in the Design Section of a TDF that is at the top of the project hierarchy. If your top-level file is not a TDF, you can make assignments in a top-level TDF with the same name and process it together with the top-level design file. This top-level TDF must contain a Design Section only. The back-annotation procedure you choose depends on which method you have used to create device and resource assignments.

You can back-annotate resource and device assignments into the .PRB file by following these steps:

1. Choose the **Project Back-Annotate** command from the File menu.
2. Select the types of assignments to be back-annotated.
3. Choose **OK**.

The **Project Back-Annotate** command copies the selected assignments from the most recent successful compilation into the .PRB file, overwriting the previous assignments. All back-annotated resource and device assignments are listed under *Existing Assignments* in the **Enter Assignments** and **Device** dialog boxes, respectively.

To back-annotate assignments into the Design Section of a TDF, you must copy the Fit File (.FIT) generated by the Compiler into the Design Section of the top-level TDF.

The Fit File records the pin, buried macrocell, chip, and device assignments for a MAX+PLUS II project. It also records all external pin connections

specified with AHDL Pin Connection Statements. The Fit File uses AHDL Design Section syntax (see MAX+PLUS II Help for information on AHDL syntax).

You can copy the assignments in the Fit File to a top-level TDF and edit them to change the resource assignments of the project. You should never edit the Fit File. The Compiler gives TDF assignments top priority. Editing a TDF that contains assignments from the Fit File is especially useful when you are grouping resources into a single MAX 5000 or MAX 7000 LAB to maximize critical-path performance. For example, Figure 12 shows SAMPLE.GDF; the Fit File for this GDF is shown in Figure 13.

Figure 12. SAMPLE.GDF

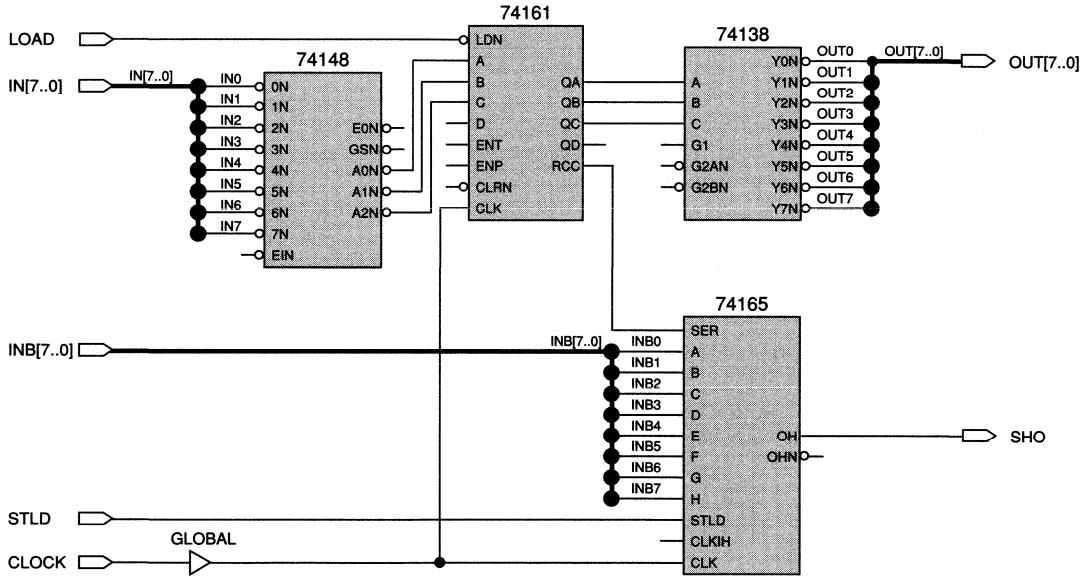


Figure 13. Fit File for SAMPLE.GDF (SAMPLE.FIT)

```
DESIGN IS "sample"
BEGIN
    DEVICE "SAMPLE" IS "EPM5064"
    BEGIN
        clock                          @  34   : INPUT  ;
        inb0                           @  33   : INPUT  ;
        inb1                           @  23   : INPUT  ; %MC24%
        inb2                           @  22   : INPUT  ; %MC23%
        inb3                           @  20   : INPUT  ; %MC22%
        inb4                           @  19   : INPUT  ; %MC21%
        inb5                           @  18   : INPUT  ; %MC20%
        inb6                           @  17   : INPUT  ; %MC19%
        inb7                           @  16   : INPUT  ; %MC18%
        in0                            @   7   : INPUT  ; %MC5%
        in1                            @  15   : INPUT  ; %MC17%
        in2                            @  30   : INPUT  ; %MC38%
        in3                            @  29   : INPUT  ; %MC37%
        in4                            @  31   : INPUT  ;
        in5                            @  13   : INPUT  ;
        in6                            @  12   : INPUT  ;
        in7                            @  11   : INPUT  ;
        load                           @   9   : INPUT  ;
        stld                           @  35   : INPUT  ;
        out0                           @  44   : OUTPUT ; %MC55%
        out1                           @  42   : OUTPUT ; %MC54%
        out2                           @  41   : OUTPUT ; %MC53%
        out3                           @  40   : OUTPUT ; %MC52%
        out4                           @  39   : OUTPUT ; %MC51%
        out5                           @  38   : OUTPUT ; %MC50%
        out6                           @  37   : OUTPUT ; %MC49%
        out7                           @   1   : OUTPUT ; %MC56%
        sho                            @   8   : OUTPUT ; %MC6%
        |74161:23|QA                   @ MC64  : BURIED ;
        |74161:23|QB                   @ MC63  : BURIED ;
        |74161:23|QC                   @ MC62  : BURIED ;
        |74161:23|QD                   @ MC61  : BURIED ;
        |74165:13|:37                  @ MC46  : BURIED ;
        |74165:13|:38                  @ MC57  : BURIED ;
        |74165:13|:65                  @ MC15  : BURIED ;
        |74165:13|:70                  @ MC43  : BURIED ;
        |74165:13|:79                  @ MC12  : BURIED ;
        |74165:13|:84                  @ MC7   : BURIED ;
        |74165:13|:93                  @ MC5   : BURIED ; %PIN 7%
    END;
END;
```

You can group the inputs and outputs of the 74165 shift register in SAMPLE.GDF into a single LAB in the EPM5064 EPLD. The assignments from the Fit File for SAMPLE.GDF can be copied into a TDF with the same name (SAMPLE.TDF) and edited. Figure 14 shows a TDF that contains Fit File assignments that were edited to assign the shift register function to macrocells in the same LAB.

Figure 14. Edited Assignments for SAMPLE.GDF (SAMPLE.TDF)

The assignments from the Fit File were copied to a TDF and edited to assign all macrocells for the 74165 macrofunction to LAB B.

```
DESIGN IS "sample"
BEGIN
    DEVICE "SAMPLE" IS "EPM5064"
    BEGIN
        clock                          @  34   : INPUT  ;
        inb0                           @  33   : INPUT  ;
        inb1                           @  23   : INPUT  ; %MC24%
        inb2                           @  22   : INPUT  ; %MC23%
        inb3                           @  20   : INPUT  ; %MC22%
        inb4                           @  19   : INPUT  ; %MC21%
        inb5                           @  18   : INPUT  ; %MC20%
        inb6                           @  17   : INPUT  ; %MC19%
        inb7                           @  16   : INPUT  ; %MC18%
        in0                            @   7   : INPUT  ; %MC5%
        in1                            @  15   : INPUT  ; %MC17%
        in2                            @  30   : INPUT  ; %MC38%
        in3                            @  29   : INPUT  ; %MC37%
        in4                            @  31   : INPUT  ;
        in5                            @  13   : INPUT  ;
        in6                            @  12   : INPUT  ;
        in7                            @  11   : INPUT  ;
        load                           @   9   : INPUT  ;
        stld                           @  35   : INPUT  ;
        out0                           @  44   : OUTPUT ; %MC55%
        out1                           @  42   : OUTPUT ; %MC54%
        out2                           @  41   : OUTPUT ; %MC53%
        out3                           @  40   : OUTPUT ; %MC52%
        out4                           @  39   : OUTPUT ; %MC51%
        out5                           @  38   : OUTPUT ; %MC50%
        out6                           @  37   : OUTPUT ; %MC49%
        out7                           @   1   : OUTPUT ; %MC56%
        sho                            @   8   : OUTPUT ; %MC6%
        |74161:23|QA                   @ MC64  : BURIED ;
        |74161:23|QB                   @ MC63  : BURIED ;
        |74161:23|QC                   @ MC62  : BURIED ;
        |74161:23|QD                   @ MC61  : BURIED ;
        |74165:13|:37                  @ MC31  : BURIED ;
        |74165:13|:38                  @ MC30  : BURIED ;
        |74165:13|:65                  @ MC29  : BURIED ;
        |74165:13|:70                  @ MC28  : BURIED ;
        |74165:13|:79                  @ MC27  : BURIED ;
        |74165:13|:84                  @ MC26  : BURIED ;
        |74165:13|:93                  @ MC25  : BURIED ;
    END;
END;
```

Conclusion

With MAX+PLUS II, you can partition a large project among multiple EPLDs automatically or by making resource and device assignments to ensure the best design fit. You can make assignments with Assign menu commands in the Graphic, Text, or Waveform Editor, or in the Design Section of a top-level TDF. You can also specify hard-wire pin connections for easy design verification. In addition, all assignments can be back-annotated to ensure that subsequent compilations produce the same fit.

Solutions to Design Timing Problems

for Altera EPLDs

April 1992, ver. 1

Application Brief 94

Introduction

When creating logic designs for EPLDs, you must adhere to sound design practices. Circuits fail for a variety of reasons, one of which is reliable timing. Factors such as silicon process and temperature variation can adversely affect circuit timing. You can, however, avoid timing problems by using synchronous designs wherever possible. In addition, synchronous designs also improve in-circuit performance, reduce debugging time, and increase circuit reliability.

This application brief describes how to create synchronous designs to eliminate common problems that can affect circuit reliability and performance. The following topics are discussed:

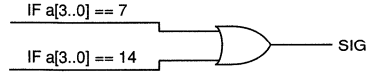
- Eliminating glitches
- Eliminating ripple counters
- Registering asynchronous input signals

Eliminating Glitches

Altera's Classic, MAX 5000, MAX 7000, and STG architectures provide a single product term to implement secondary register functions such as array Clock, Preset, and Clear signals. For most functions, a single product term is sufficient. However, when a function requires two or more product terms, timing delays, and therefore potential glitches, may be introduced.

For example, the basic combinatorial circuit shown in Figure 1 consists of two AND terms connected by a single OR term. If the bus `a[3..0]` is equal to 7 or 14, `SIG` will be high, otherwise `SIG` will be low. If `SIG` is a combinatorial function feeding an output pin or a flipflop data input, the function can be easily implemented in any Classic, MAX 5000, MAX 7000, or STG EPLD. However, if `SIG` is an array Clock, Preset, or Clear signal, the function may cause a glitch since the two product terms require an additional macrocell or shareable expander.

Figure 1. Basic Combinatorial Circuit

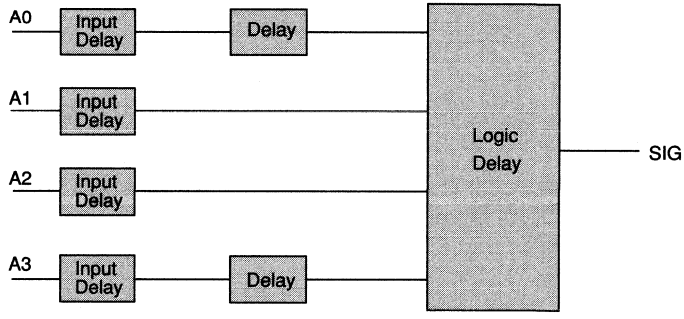


Altera EPLDs provide a number of architectural features that give you much greater design flexibility, but may introduce additional timing delays. For example, Classic EPLDs can allocate resources—such as logic from

other macrocells—that allow you to implement complex logic. The MAX+PLUS II Compiler uses these resources to automatically fit logic into an EPLD in the most efficient way. However, use of allocatable resources causes timing delays.

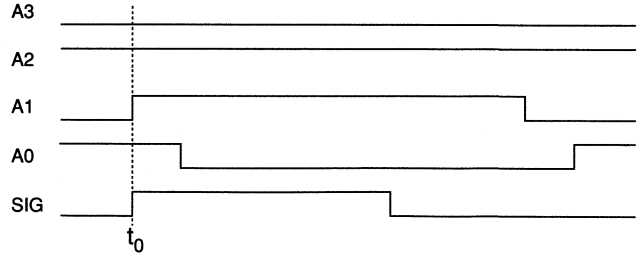
The MAX architecture ensures 100% routability by linking small, high-performance, flexible array modules called Logic Array Blocks (LABs) via a dedicated progrxammable network called the Programmable Interconnect Array (PIA). Logic expanders in each LAB provide additional logic resources to any macrocell in an LAB. However, all signals from macrocells and expanders that cross the PIA necessarily incur a timing delay, and therefore may cause a glitch in the output of a combinatorial circuit. Figure 2 shows the timing model for the basic combinatorial circuit and the timing delays incurred.

Figure 2. Combinatorial Circuit Timing Model



This timing model shows that `A0` and `A3` go through a delay before reaching the logic to generate `SIG`. This delay may result from shareable expanders, macrocells, or signals coming from the PIA. The waveform for this circuit (Figure 3) shows that while the `a[3..0]` signals are applied to the device at the same time (t_0), there is a delay between `A1` and `A0`. This delay causes a logic 7 (`a3`=0, `a2`=1, `a1`=1, `a0`=1) to temporarily appear during the transition from a logic 5 to a logic 6, resulting in a glitch.

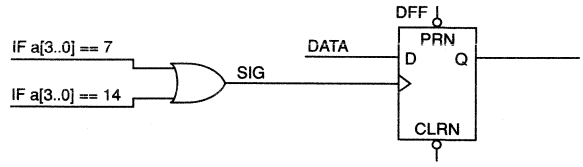
Figure 3. Combinatorial Circuit Waveform



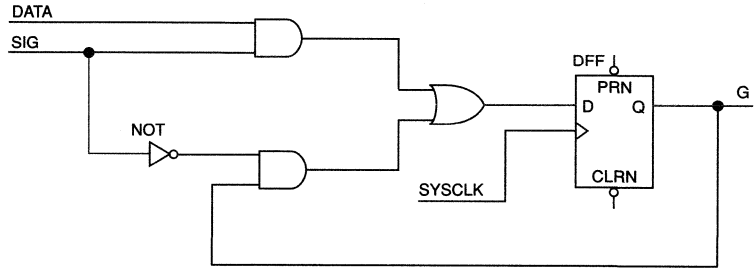
You can easily protect your circuit from this kind of glitch by implementing a synchronous clocking scheme to modify complex logic that defines a Clock signal. Synchronous clocking uses a single pin-driven system Clock rather than an asynchronous (logic-driven) Clock. See Figure 4.

Figure 4. Comparison of Array and Synchronous Clocking

Array Clocking (Logic-Driven)

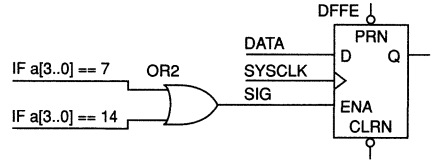


Synchronous Clocking (Pin-Driven)



In the synchronous clocking scheme, `SIG` is a part of the `D` input, and the Clock is pin-driven. This circuit is known as an enabling circuit for a register. The MAX+PLUS II TTL Macrofuntion Library provides `DFFE`, a D flipflop that implements this type of circuit. See Figure 5.

Figure 5. Enable D Flipflop Primitive (DFFE)



You can also use this method to simplify complex logic that drives Preset and Clear signals. Figure 6 shows `SIG` directly driving a Clear input to a register.

Figure 6. Asynchronous Clear

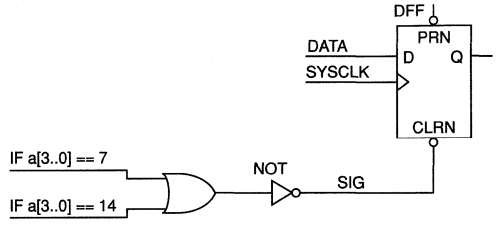
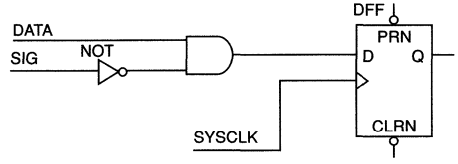


Figure 7 shows how you can implement a synchronous Clear.

Figure 7. Synchronous Clear



Asynchronously loaded counters using the Preset and Clear signals previously discussed also create problems with design reliability. Whenever possible, use a synchronously loaded counter, such as a 74163 (4-bit binary up counter).

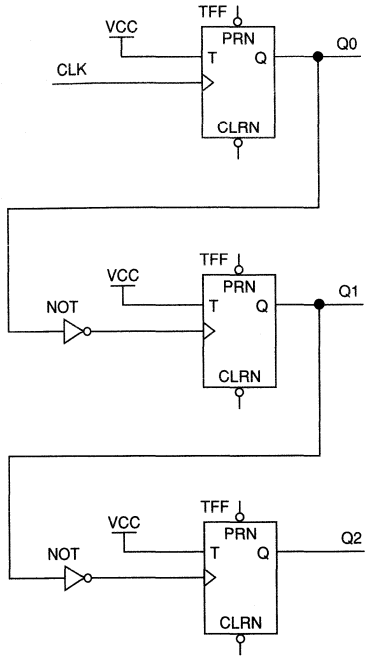
For more information on how to implement gated Clocks, Clears, and Presets safely, see *Application Note 26 (EPLD/MLPD Design Guidelines)* in this handbook.

Eliminating Ripple Counters

Another potential timing problem can be caused by ripple clocking, a common clocking scheme for counters. Ripple clocking relies on the output of one register to generate the Clock for another, rather than use a common system Clock. An example of a ripple-clocked counter is a 74193 asynchronous up/down counter. Figure 8 shows another example of ripple clocking.

The Clock period of the circuit in Figure 8 depends on the time it takes for the change in the least significant bit (LSB) of the counter to propagate through to the most significant bit (MSB). This period depends on the size of the counter. Therefore, as ripple counters grow, system speeds decrease. In general, you should use a synchronous counter whenever possible.

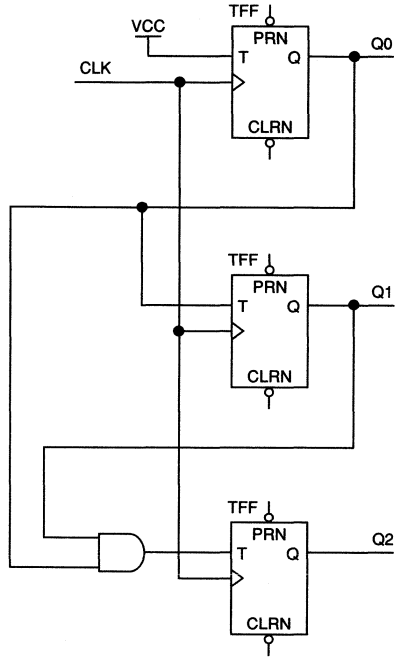
Figure 8. Ripple Clocking



When the ripple counter is redesigned for synchronous operation, as shown in Figure 9, the system speed is determined by the setup and hold times of the registers, rather than the number of bits in the counter. Without requiring additional resources, the synchronous implementation improves the speed of the counter, and eliminates the possibility of invalid count values.

You can also improve design reliability by making careful resource assignments in MAX+PLUS II. Signals that cross the PIA in MA devices incur a timing delay. To eliminate this delay, you can make clique assignments in MAX+PLUS II to place consecutive counter bits in macrocells that are all in the same LAB.

Figure 9. Synchronous Counter



Registering Asynchronous Input signals

Signals entering an EPLD are often not synchronized with the global Clock. If these signals are used as inputs to any registered function (counters, state machines, controllers, etc.), setup and hold violations can occur. These violations can cause invalid data to be clocked into the register, or place the register in a metastable state. Figure 10 shows a state machine with asynchronous inputs.

Figure 10. State Machine with Asynchronous Inputs

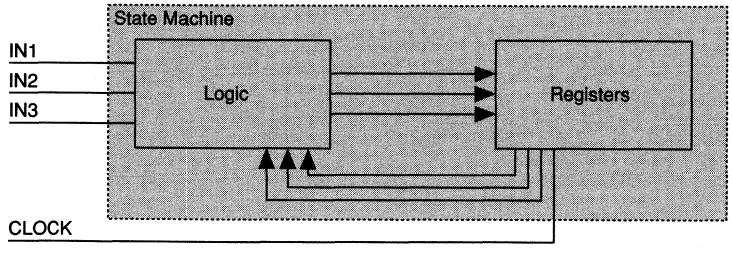
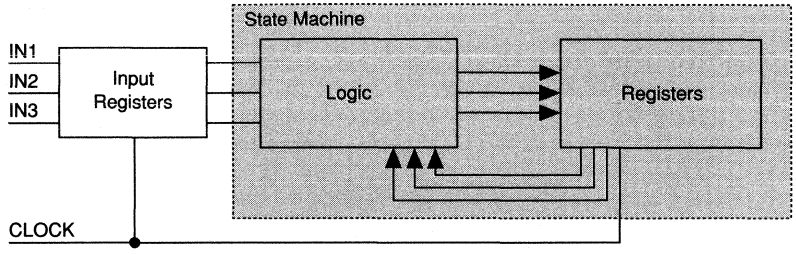


Figure 11 shows how you can add input registers to ensure that the setup and hold times for the state machine are met. Although the additional registers are still exposed to possible setup and hold violations, the state registers are prevented from going into a metastable state, and the state machine does not enter an undefined state.

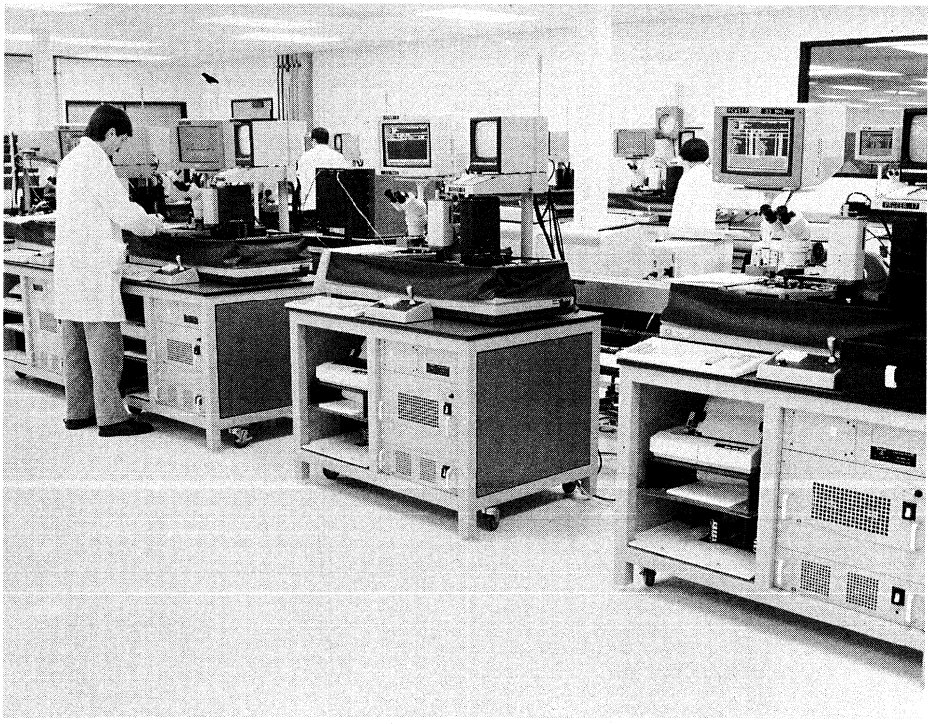
Figure 11. State Machine with Additional Latching



Conclusion

Altera EPLDs provide great flexibility for implementing logic-intensive designs. Advanced features such as shareable expanders and the PIA increase the logic capacity of EPLDs. While these features can introduce timing problems, you can easily avoid such problems by using synchronous circuits. Synchronous designs also protect your circuit against timing problems illuminated by external factors such as silicon process and temperature variation. The timing analysis features provided with MAX+PLUS II allow you to observe timing relationships quickly and easily, helping you understand and account for timing delays between nodes in your designs. For more information on EPLD timing models, see *Application Brief 100 (Understanding EPLD Timing)* in this handbook.

In general, you should use synchronous designs whenever possible. Not only are they immune to many of the timing problems of asynchronous designs, they are also easier to implement and test.



Designing Phase-Locked Loops with Altera EPLDs

April 1992, ver. 1

Application Brief 95

Introduction

Phase-locked loops (PLLs) are used in many different applications, ranging from video synchronization to scientific instrumentation. In these circuits, a PLL matches a reference oscillator to an incoming data signal and adjusts the reference oscillator to lock the frequency and phase. Historically, a wide range of analog PLLs have been available to build control circuits based on PLL operating principles. However, the popularity of analog circuits is decreasing as programmable logic vendors supply devices with the speed and density necessary to create digital implementations of these analog functions. Altera's EPS464 Synchronous Timing Generator (STG) EPLD has sufficient density to implement a complex control system and additional functions such as PLLs, and is ideal for video controller applications that include PLLs.

When a PLL is implemented as digital logic, the term "phase-locked loop" is not completely accurate. The digital portion of the circuit is used only to detect the differences in phase and frequency, but does not perform the correcting or restoring functions to bring the data signals under control. This restoring path (the "loop" in the phase-locked loop) is implemented with external components or discrete circuitry. Therefore, the term "digital phase detector" better describes an EPLD's role in a complete PLL circuit. This application brief decribes how to implement the digital portion of the circuit, i.e., the digital phase detector, in an Altera EPLD.

This application brief discusses the following topics:

- ❑ PLL description
- ❑ Implementing a PLL in Altera EPLDs
- ❑ Types of phase detectors

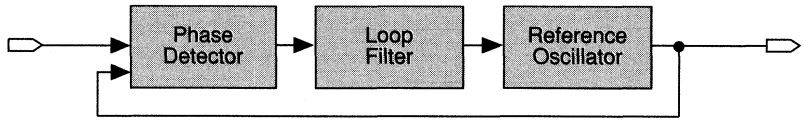
PLL Description

A phase-locked loop is an electronic circuit that provides information on the phase and frequency differences between an input signal and a reference signal. It also generates the control signals necessary to synchronize the reference signal with the data input. A PLL consists of the following elements:

- ❑ Phase detector
- ❑ Loop filter
- ❑ Reference oscillator

Figure 1 shows a block diagram of a simple PLL. The phase detector determines the time difference between corresponding rising (or falling) edges in the two input signals, and indicates this difference by asserting control signals through the loop filter to the voltage- or current-controlled reference oscillator. The oscillator is configured so that the signal generated by the phase detector adjusts the frequency to bring the reference signal into phase lock and frequency lock with the input signal. The loop filter, a low-pass filter, enables the circuit to ignore small phase differences so that the correcting signal does not send the reference oscillator into an unpredictable response cycle.

Figure 1. Simple Phase-Locked Loop



Implementing a PLL in Altera EPLDs

Delay paths for PLL signals must be as closely matched as possible. The path for the incoming data signal should be identical to the path for the reference oscillator signal. Any variation appears as a phase error, and subjects the system to narrow spikes of over-correction. To prevent these spikes, you can place a low-pass filter outside the digital device. Since the macrocell-based architecture of Altera EPLDs provides predictable delays, and the delays between different data paths are closely matched, EPLDs are an excellent choice for the digital phase detector used in PLL circuits.

Altera has developed the application-specific PLL macrofunction to implement the digital portion of a phase detector. The PLL macrofunction, provided in the MAX+PLUS II TTL MacroFunction Library, has an nSET input. This input is provided primarily for simulation purposes, but the application circuits may also use them for initialization control. Because the PLL macrofunction is based on self-latching circuits, you must provide an input that resets the entire path. All macrofunction inputs have intelligent default values so that unused inputs can simply be left unconnected.

Types of Phase Detectors

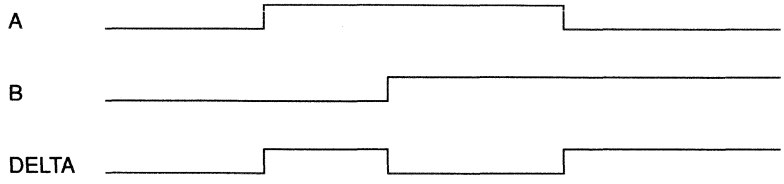
To create the phase detector, you must develop a circuit that detects when the two incoming signals have different logic values, and also remembers which signal changes and which remains constant. Phase detectors can be implemented in Altera EPLDs with many different design approaches. The following implementations are discussed in this application brief, from simplest to most complex:

- XOR gate
- SR latch
- NAND latch

XOR Gate

The simplest phase detector is an XOR gate. Figure 2 shows a waveform description of an XOR gate. The XOR output (DELTA) is asserted whenever the logic levels on the A and B inputs differ. Although the XOR gate detects phase differences, it does not indicate which input signal changes first, thus limiting its usefulness for control feedback-signal generation.

Figure 2. XOR Gate Waveforms



SR Latch

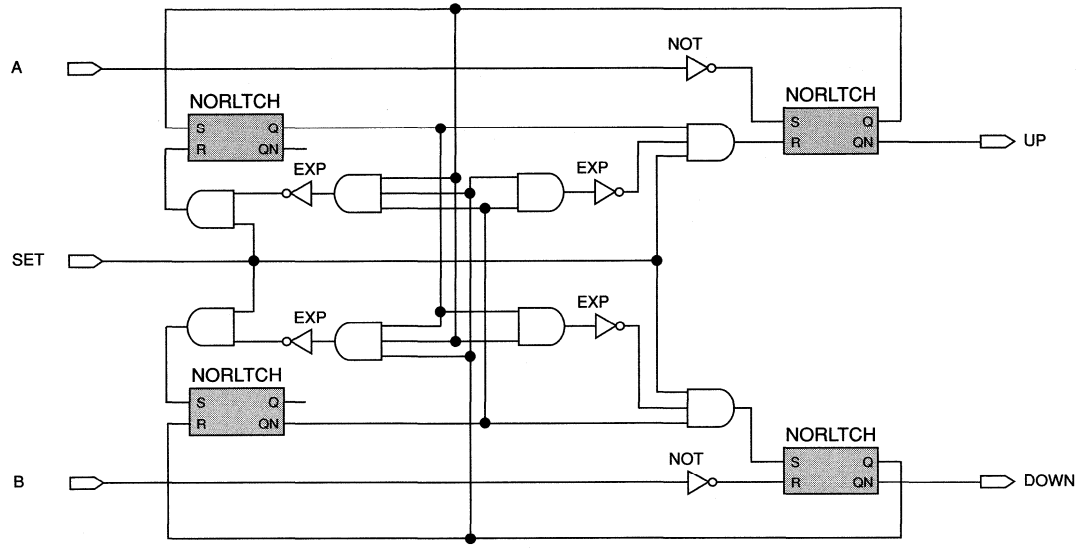
You can implement a better, more complex phase detector with SR latches, which can be created by cross-coupling shareable expander product terms in Altera MAX 5000, MAX 7000, and STG EPLDs. Shareable expanders are freely allocatable product terms that you can use to create complex gating functions. Since the delays are fixed and nearly identical across the device, this cross-coupling provides good circuit response.

Figure 3 shows a PLL that is implemented with expander-based SR latches (NORLTCH), which are provided in the MAX+PLUS II TTL MacroFunction library. The circuit "remembers" which input signal rises or falls first. Despite its complexity, this type of circuit provides very good response and requires only 12 expanders and 2 macrocells.

NOT gates on the A and B inputs make this circuit a rising-edge detector. The ideal PLL provides rising- and falling-edge detection, but many applications require detection on only one edge. This circuit responds with correction signals that bring the system into phase and frequency lock if the rising edges of the two inputs do not always occur together, i.e., are not in phase. Signals that are out of phase tend to have significantly mismatched edges.

The circuit uses feedback to bring a reference oscillator into phase and frequency lock with an incoming data signal. Some applications require a simple phase detector between two unrelated data signals, with no need to control a reference oscillator. In a system where the two signals have the same frequency but have a duty-cycle difference, so that the rising edges always coincide, the rising-edge SR-based detector does not work. In this case, a falling-edge detector is appropriate. You can create a falling-edge detector by removing the NOT gates from the A and B inputs.

Figure 3. Phase-Locked Loop (SR Latch Implementation)



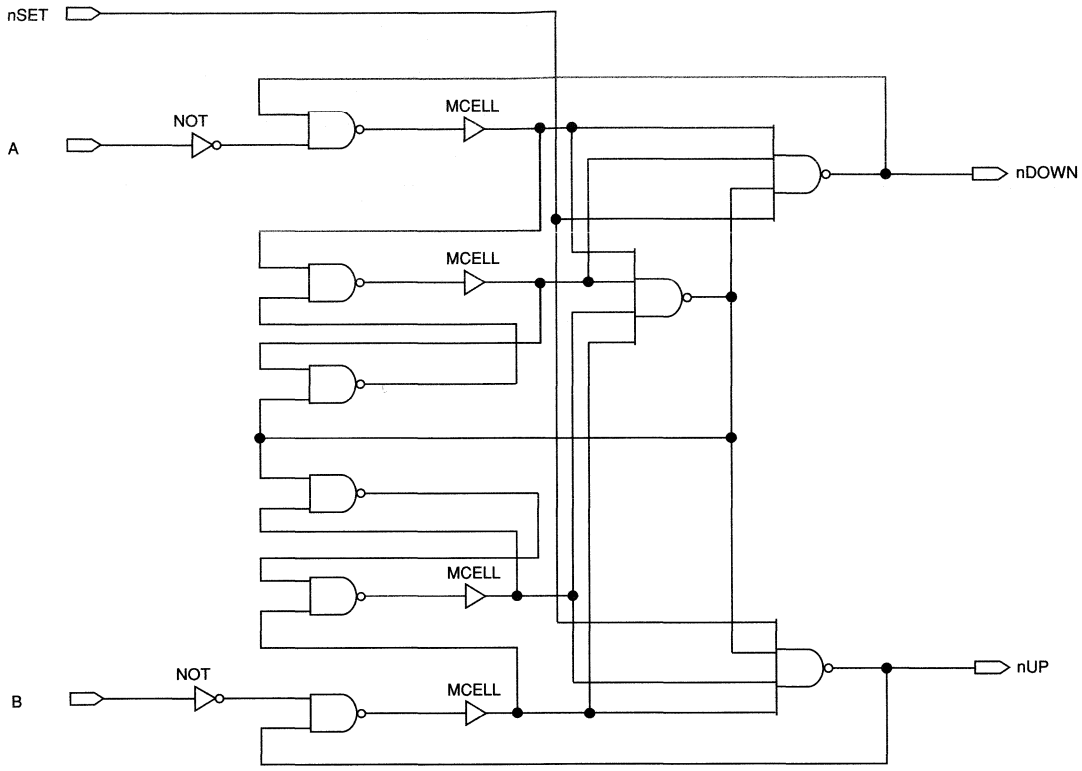
The expanders in this circuit require only two macrocells. Proper operation of this circuit depends on expander product terms, which have one drawback: at power-up, their logic state is not guaranteed. If you need initial stability and predictability, or if you need to perform a timing simulation for your project, you should use macrocells instead of expanders to implement the circuit.

NAND Latch

The PLL in Figure 4 is a complex rising-edge detector that uses cross-coupled NAND gates with macrocell feedbacks. This approach uses six macrocells rather than two, but provides a more stable, predictable circuit.

A single-ended phase detector (rising or falling edge) is usually sufficient to control most circuits, but in some cases even the smallest phase difference is unacceptable. These circuits require a dual-edge detector that allows the system to respond immediately. For example, assume that the rising edges on the two input signals of a rising-edge detector are coincident. If the two signals remain TRUE for 1 µs, then have a 5-ns difference between the falling edges, the correction does not occur until the 5-ns difference is seen on the subsequent rising edges and detected by the single-ended detector. A dual-ended edge detector is the best choice for such a highly sensitive circuit.

Figure 4. Rising-Edge Detector (Macrocell Implementation)

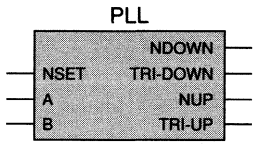


Although you can use other devices to implement this function, an EPLD is ideal because of its predictable macrocell timing. It also allows you to combine the response of separate rising- and falling-edge detectors to implement the circuit.

The PLL macrofunction, shown in Figure 5, contains rising- and falling-edge detectors. The logic for the rising- and falling-edge detectors is identical, except that the rising-edge detector has inverters on the inputs, and uses an AND gate to multiplex the two output signals (since the outputs are active low). PLL is available in the MAX+PLUS II TTL MacroFunction Library, or can be downloaded from the Altera Applications electronic bulletin board service (BBS).

In a well-designed PLL, the nDOWN and nUP signals should never be active at the same time, or the feedback will cause erratic oscillations and unpredictable circuit response, and the system will require many data

Figure 5. PLL Macrofunction



cycles to return to phase and frequency lock. You should simulate PLL designs and test the programmed EPLD in-system to ensure that the EPLD performs as expected.

The TRI-DOWN and TRI-UP outputs are provided for PLL circuits that require mutually exclusive control signals, one of which is always tri-stated. For example, certain types of controlled oscillators require this feature. In the PLL macrofunction, TRI-DOWN drives GND when it is active, and is tri-stated when it is inactive. TRI-UP drives VCC when it is active, and is tri-stated when it is inactive.

Table 1 shows simulation results for the PLL macrofunction implemented in the EPS464 STG and single-LAB MAX 5000 EPLDs. This analysis does not include bench tests, and only indicates the differences in performance between the EPLDs. To obtain the fastest and most reliable performance, you should group all primitives in a PLL macrofunction into the same Logic Array Block (LAB) when the macrofunction is placed in a multi-LAB device. You can group the primitives by entering clique assignments in MAX+PLUS II or by specifically assigning hard logic functions (i.e., a function that is not removed during logic synthesis) to macrocells and pins in the same LAB.

Table 1. PLL Macrofunction Performance Analysis

Device Family	Device	Width (ns)	Precision (ns)	Delay (ns)
STG	EPS464-20	11	1	31
MAX 5000	EPM5016-15	8	1	31
	EPM5032-15	8	1	31
	EPM5064-25	16	1	57
	EPM5128-25	16	1	57

The widths in Table 1 show the narrowest signal pulse that PLL can resolve without causing oscillation. The pulse size is associated with the feedback delay of the self-latching circuit in the device. The precision values show the smallest phase difference that can be detected by the device. PLL simulation tests in MAX+PLUS II show a resolution of 1 ns. However, in-system experiments show that the phase difference is closer to 2 or 3 ns. The delay values show how much time elapses between the phase difference on the inputs and the assertion of the correct difference signal that brings the reference oscillator back into lock. While the EPS464 EPLD and the single-LAB MAX 5000 devices have similar performance characteristics, the higher density of the EPS464 makes it the ideal EPLD for PLL implementation.

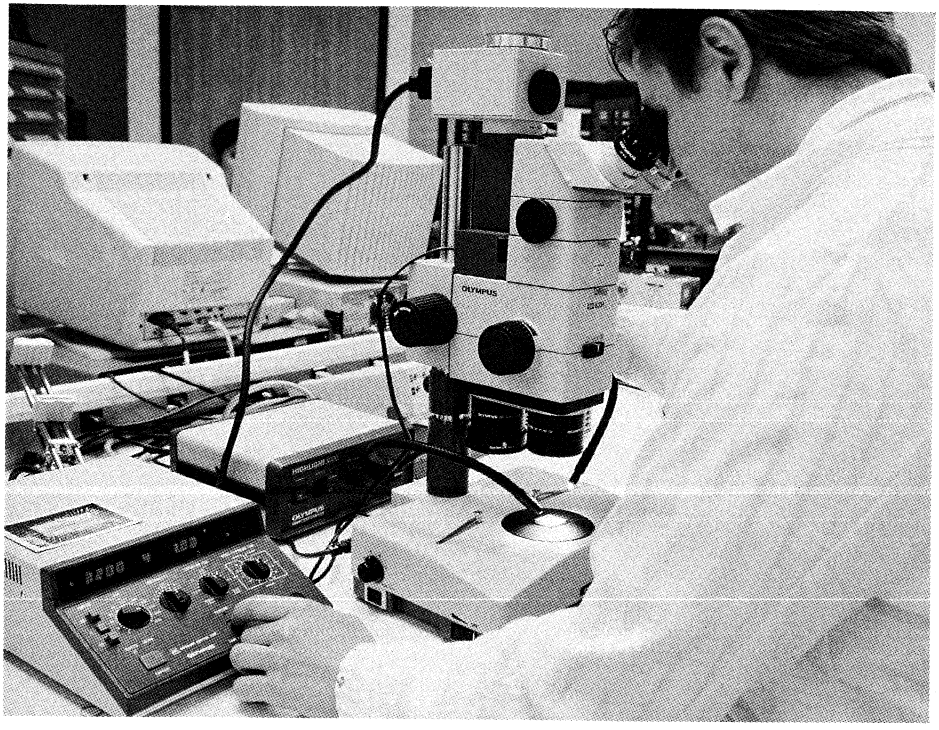
Other PLL Possibilities

The dual-ended PLL macrofunction provided in the MAX+PLUS II TTL MacroFunction Library conserves device resources and provides excellent response time. If silicon efficiency is not the most important consideration for your design, you can easily implement elaborate phase detection schemes that use dozens of macrocells to "vote" and "poll" in large EPLDs.

You can also use different resources in the EPLD to implement the same function by replacing the NAND gates with AND and EXP primitives. This implementation is harder to simulate because you must set and reset the function in the correct sequence; however, you can use it if you have spare expanders and wish to save macrocells in your EPLD design.

Conclusion

The increased density and speed of Altera EPLDs make it possible to integrate functions previously reserved for analog circuitry. While high-precision work should be reserved for ultra-precision linear circuits, you can implement applications that require functions such as PLLs and edge detectors with the digital logic provided by EPLDs.


OLYMPUS

Generating Library Mapping Files

April 1992, ver. 2

Application Brief 96

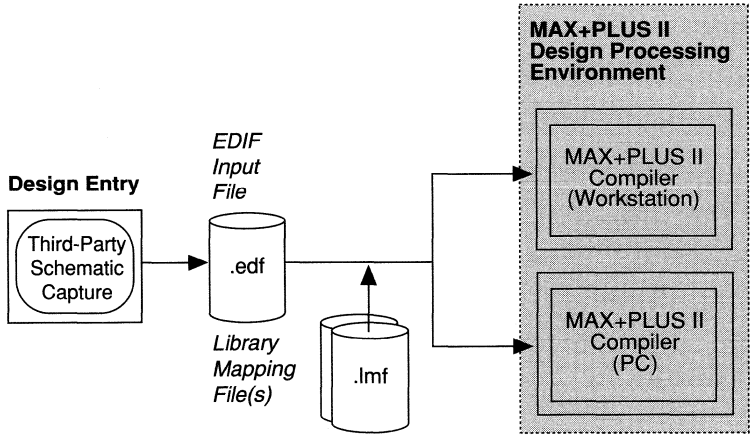
Introduction

Altera's MAX+PLUS II development systems for the PC and workstation can compile and implement logic designs created with third-party PC- or workstation-based design entry tools. Third-party CAE tools convert the entered design into an Electronic Data Interchange Format (EDIF) file so that it can be read by MAX+PLUS II software.

With Library Mapping Files (**.lmf**) and the MAX+PLUS II EDIF Netlist Reader, you can convert a design file created with a third-party CAE tool into an EDIF Input File (**.edf**). The MAX+PLUS II Compiler can compile this file directly. Figure 1 shows this design conversion process.

This application brief discusses LMFs and their syntax and offers design guidelines. It also includes a list of existing LMF mappings.

Figure 1. Third-Party Design Conversion Process with Library Mapping Files



LMF Description

LMFs map EDIF cell and port names, i.e., the third-party function and pin names, to compatible Altera macrofunction or primitive names and their pin names. Lower-level logic descriptions of the function are not passed to the MAX+PLUS II Compiler, since Altera macrofunctions are optimized for Altera EPLD architectures. LMF mapping is always required, even if the EDIF function and pin names match the Altera macrofunction and pin names.

Altera provides LMFs with over 125 TTL macrofunction and primitive mappings for Viewlogic Systems, Cadence Design Systems, and Mentor Graphics CAE systems. LMFs are editable text files with a straightforward syntax, so you can easily build and customize LMFs for your own designs.

LMF Syntax

Figure 2 shows an excerpt from a Viewlogic LMF.

Figure 2. Viewlogic Library Mapping File Excerpt (vwl_bas.lmf)

```
LIBRARY             VWL1

BEGIN
    FUNCTION        74373       (D1, D2, D3, D4, D5, D6, D7, D8, G, OEN)
        RETURNS                 (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8)
    FUNCTION        "74LS373"   ("1D", "2D", "3D", "4D", "5D", "6D", "7D", "8D",
                                "G", "~OC")
        RETURNS                 ("1Q", "2Q", "3Q", "4Q", "5Q", "6Q", "7Q", "8Q")
END

% Octal tri-state buffer %
BEGIN
    FUNCTION        74244       (1GN, 1A1, 1A2, 1A3, 1A4, 2GN, 2A1, 2A2, 2A3, 2A4)
        RETURNS                 (1Y1, 1Y2, 1Y3, 1Y4, 2Y1, 2Y2, 2Y3, 2Y4)
    FUNCTION        "74LS244"   ("~G", "A1", "A2", "A3", "A4", "", "", "", "", "")
        RETURNS                 ("Y1", "Y2", "Y3", "Y4", "", "", "", "")
END
```

LMFs are case sensitive. Within the LMF, the Altera input/output pin lists are in capital letters. Use only lowercase letters for the Altera macrofunction name. The case for the EDIF cell names and EDIF input/output pin lists within the LMF must match the case in the EDIF netlist.

All LMFs begin with a declaration of the library name. MAX+PLUS II uses this name as a reference. The actual mapping of a macrofunction occurs in a six-line construct with the following format:

Line 1: The keyword `BEGIN`.

Line 2: The keyword `FUNCTION`, followed by the Altera macrofunction name (in lowercase letters), and a comma-separated list of all the macrofunction input pin names. The input list is enclosed in parentheses (`()`).

Line 3: The keyword `RETURNS`, followed by a comma-separated list of all the macrofunction output pin names. The output list is enclosed in parentheses.

Line 4: The keyword FUNCTION, followed by the EDIF cell name enclosed in double quotation marks ("), and a comma-separated list of all third-party function input pin names, each enclosed in double quotation marks. The input list is enclosed in parentheses.

When generating the EDIF netlist file for a design, some EDIF netlist writers, such as EDIFNET from Mentor Graphics, place the function's complete file pathname (e.g., /user/ls_lib/$74LS193) within the cell declaration section of the EDIF netlist file. The EDIF cell name in Line 4 must also contain this same pathname, enclosed in double quotation marks, to map the cell to an Altera macrofunction.

Line 5: The keyword RETURNS, followed by a comma-separated list of all the third-party function output pin names, each enclosed in double quotation marks. The output list is enclosed in parentheses.

Line 6: The keyword END.

The order of the pin names within the inputs and outputs lists of Line 4 and Line 5 must match the pin name order of the Altera macrofunction in Line 2 and Line 3.

You can find pin names for Altera macrofunctions in the Function Prototype Statements that are stored in the Altera Include Files (**.inc**). These files are located in the **/usr/maxplus2/max2lib/max2inc** directory on the workstation or the **\maxplus2\max2inc** directory on the PC.

To add user comments and documentation to LMFs, enclose text within percent symbols (%) anywhere outside of the mapping construct, i.e., before the BEGIN or after the END statements. Nested comments are not allowed.

Placeholders

Some third-party TTL functions, although functionally similar to their Altera counterparts, may not map pin-for-pin to Altera TTL macrofunctions. For example, the Altera 74244 octal tri-state buffer function in Figure 2 contains all eight buffers and the associated control logic of the actual TTL device. However, the 74LS244 function provided by Viewlogic implements only four of the eight buffers of the TTL function. Although the Viewlogic 74LS244 macrofunction contains four fewer output pins and five fewer input pins than the Altera 74244 macrofunction, you can still use LMFs to map this function by taking advantage of placeholders. Placeholders are spaces, closed quotes, or global VCC or GND designations used to equate the number of EDIF ports with the number of Altera ports for a particular macrofunction. The following examples illustrate possible uses for placeholders.

Example 1

If there are more Altera input pins than EDIF input ports, map the extra Altera input pins to VCC or GND to make them inactive. In the following example, the active-low PRN pin of the Altera DFF primitive must be driven by VCC to become inactive. The placeholder is the global-high or global-low signal defined by the third-party tool. Be careful to choose intelligent default values.

```
BEGIN
    FUNCTION    dff      (CLRN, CLK, D, PRN)
        RETURNS          (Q)
    FUNCTION    "LS174" ("-CL", "CLOCK", "D<0>", "VCC")
        RETURNS          ("Q<0>")
END
```

Example 2

If there are fewer Altera input pins than EDIF input ports, insert an extra comma at the end of the Altera macrofunction input list for each of the EDIF input ports that cannot be mapped to the Altera macrofunction. In the following example, only the A<0> and B<0> ports of the LS21 function are mapped onto the AND2 primitive. The C<0> and D<0> ports must not be driven by signals within the EDIF Input File; if they are, the Compiler reports that the signal driving the unmapped pin is a node without a destination.

```
BEGIN
    FUNCTION    and2     (IN1, IN2, , )
        RETURNS          (OUT)
    FUNCTION    "LS21"   ("A<0>", "B<0>", "C<0>", "D<0>")
        RETURNS          ("Y<0>")
END
```

Example 3

If there are more Altera output pins than EDIF output ports, the RETURNS section for the EDIF function must contain an extra comma (,) and double quotation marks for each of the unmapped outputs in the Altera macrofunction. In the following example, the Compiler automatically removes the Q1 and Q2 outputs of the customized BILL macrofunction and all logic within the BILL macrofunction required to generate these outputs.

```
BEGIN
    FUNCTION    bill     (CLRN, BOG, V, PRTY)
        RETURNS          (Q0, Q1, Q2)
    FUNCTION    "TED"    ("-CL", "-BGS", "Z", "-PRTY")
        RETURNS          ("Q0", "", "")
END
```

Example 4

If there are fewer Altera output pins than EDIF output ports, the RETURNS section for the Altera macrofunction must contain commas for each of the EDIF ports that do not map to an Altera macrofunction port. In this example, if the -Q<0> port drives a net within the EDIF netlist, the MAX+PLUS II Compiler generates a warning because no source pin is driving the net.

```
BEGIN
    FUNCTION    dff      (CLRN, CLK, D, PRN)
        RETURNS          ( ,Q)
    FUNCTION    "LS74"   ("-CL", "CLOCK", "D<0>", "-PR")
        RETURNS          ("-Q<0>", "Q<0>")
END
```

LMF Design Guidelines

When you create LMF entries, follow these guidelines for faultless macrofunction conversion:

- Although pin order is not important within an individual input or output listing, it is critical between corresponding EDIF cells and Altera macrofunctions.
- EDIF cell and Altera macrofunction pin lists must have the same number of members. Use placeholders for any pin-count mismatch.
- LMFs are case-sensitive. Within the LMF, the Altera input/output pin lists are in capital letters. Use only lowercase letters for the Altera macrofunction name. The case for the EDIF cell names and EDIF input/output pin lists within the LMF must match the case in the EDIF netlist.
- The third-party function name and individual third-party pin names must be enclosed in double quotation marks.
- If multiple mappings exist, only the first instance of a mapping is used.
- Place comments anywhere outside of the mapping construct, i.e., before the BEGIN or after the END statement. Comments are any alphanumeric text enclosed by percent symbols. Nested comments are not allowed.
- Use white space (spaces, tabs, carriage returns) to improve readability. If pin names span multiple lines, no continuation character is required.

Existing LMF Listings

Table 1 shows Mentor Graphics, Cadence, and Viewlogic TTL macrofunction mappings. In addition, all Altera-provided LMFs contain mappings for all basic gates (e.g., AND, XOR, and DFF) and architecture directive elements such as synchronous Clocks (GLOBAL), expander product terms (EXP), soft buffers (SOFT), and macrocell buffers (MCELL).

Table 1. TTL Function Mappings in Altera-Provided LMFs (Part 1 of 3)

Altera	Mentor Graphics	Cadence	Viewlogic
7400	74LS00	LS00	74LS00
7402	74LS02	LS02	74LS02
7404	74LS04	LS04	74LS04
7408	74LS08	LS08	74LS08
7410	74LS10	LS10	74LS10
7411	74LS11	LS11	74LS11
7420	74LS20	LS20	74LS20
7421	74LS21	LS21	74LS21
7427	74LS27	LS27	74LS27
7428	74LS28	LS28	74LS28
7430	74LS30	LS30	74LS30
7432	74LS32	LS32	74LS32
7437	74LS37	LS37	74LS37
7440	74LS40	LS40	74LS40
7442	74LS42	LS42	74LS42
7451	74LS51	LS51	74LS51
7454	74LS54	LS54	74LS54
7455	74LS55	LS55	74LS55
7473	74LS73A	LS73	74LS73A
7474	74LS74A	LS74	74LS74A
7475	74LS75	LS75	74LS75
7476	74LS76A	LS76	74LS76A
7477	74LS77	—	74LS77
7478	—	LS78	74LS78A
7483	74LS83A	LS83	74LS83A
7485	74LS85	LS85	74LS85
7486	74LS86	LS86	74LS86
7490	74LS90	LS90	74LS90
7491	74LS91	LS91	74LS91
7492	74LS92	LS92	74LS92
7493	74LS93	LS93	74LS93
7495	74LS95B	LS95	74LS95B
7496	74LS96	LS96	74LS96
74107	74LS107A	LS107	74LS107A
74109	74LS109A	LS109	74LS109A
74112	74LS112A	LS112	74LS112A
74113	74LS113A	LS113	74LS113A
74114	74LS114A	LS114	74LS114A
tri	74LS126A	LS126	74LS126A

Table 1. TTL Function Mappings in Altera-Provided LMFs (Part 2 of 3)

Altera	Mentor Graphics	Cadence	Viewlogic
74133	74LS133	LS133	74LS133
74138	74LS138	LS138	74LS138
74139	74LS139A	LS139	74LS139
74147	74LS147	LS147	74LS147
74148	74LS148	LS148	74LS148
74151	74LS151	LS151	74LS151
74153	74LS153	LS153	74LS153
74154	74LS154	LS154	—
74155	74LS155A	LS155	74LS155A
74156	74LS156	LS156	74LS156
74157	74LS157	LS157	74LS157
74158	74LS158	LS158	74LS158
74160	74LS160A	LS160	74LS160A
74161	74LS161A	LS161	74LS161A
74162	74LS162A	LS162	74LS162A
74163	74LS163A	LS163	74LS163A
74164	74LS164	LS164	74LS164
74165	74LS165	LS165	74LS165
74166	74LS166	LS166	74LS166
74168	74LS168A	—	—
74169	74LS169A	LS169	74LS169
74173	74LS173A	LS173	74LS173A
74174	74LS174	LS174	74LS174
74175	74LS175	LS175	74LS175
74181	74LS181	LS181	74LS181
74183	74LS183	LS183	74LS183
74190	74LS190	LS190	74LS190
74191	74LS191	LS191	74LS191
74192	—	LS192	74LS192
74193	74LS193	LS193	74LS193
74194	74LS194A	LS194A	74LS194A
74195	74LS195A	LS195	74LS195A
74196	74LS196	LS196	74LS196
74197	74LS197	LS197	74LS197
74240	74LS240	LS240	74LS240
74241	74LS241	LS241	74LS241
74244	74LS244	LS244	74LS244
74251	74LS251	LS251	74LS251
74253	74LS253	LS253	74LS253

Table 1. TTL Function Mappings in Altera-Provided LMFs (Part 3 of 3)

Altera	Mentor Graphics	Cadence	Viewlogic
74257	74LS257A	LS257	74LS257
74258	74LS258A	LS258	74LS258
74259	—	LS259	74LS259
74260	74LS260	LS260	74LS260
74261	—	LS261	—
xnor	74LS266	LS266	74LS266
74273	74LS273	LS273	74LS273
74279	—	LS279	74LS279
74280	74LS280	LS280	74LS280
74283	74LS283	LS283	74LS283
74290	74LS290	LS290	74LS290
74293	74LS293	LS293	74LS293
74299	—	LS299	74LS299
74348	74LS348	LS348	74LS348
74353	74LS353	LS353	74LS353
74365	74LS365A	LS365	74LS365A
74366	74LS366A	LS366	74LS366A
74367	74LS367A	LS367	74LS367A
74368	74LS368A	LS368	74LS368A
74373	74LS373	LS373	74LS373
74374	74LS374	LS374	74LS374
74377	74LS377	LS377	74LS377
74379	—	LS379	74LS379
74381	74LS381	LS381	74LS381
74386	—	LS386	—
74390	—	LS390	74LS390
74393M	74LS393	LS393	74LS393

Conclusion

LMFs map third-party function and pin names to compatible Altera macrofunction or primitive names, so that MAX+PLUS II can easily compile EDIF netlist files generated from hierarchical logic designs created with workstation CAE tools. You can use LMFs provided by Altera, or you can customize or generate your own LMFs to include new mappings. LMFs can help you use your workstation CAE tools with MAX+PLUS II for an efficient design solution.

Integrating EDIF Files with AHDL Files

in Hierarchical Projects

April 1992, ver. 1

Application Brief 97

Introduction

Altera's PC- and workstation-based MAX+PLUS II development systems allow you to combine designs created with multiple third-party workstation CAE tools into a single design, so you can choose the entry method best suited to each portion of the logic. On the workstation, you can enter designs as text files created in the Altera Hardware Description Language (AHDL), as third-party schematics, or as a mix of both text and schematics.

You can take advantage of multiple design entry methods by creating hierarchical schematic descriptions with third-party CAE tools and then converting them into industry-standard Electronic Design Interchange Format (EDIF) netlist files. EDIF netlist files can be completely self-contained designs, or can integrate one or more separate EDIF netlist files or AHDL Text Design Files (**.tdf**). You can then compile EDIF Input Files (**.edf**) and any associated AHDL TDFs in MAX+PLUS II. In addition, your design (called a "project" in MAX+PLUS II) can include Waveform Design Files (**.wdf**) or Graphic Design Files (**.gdf**) created with MAX+PLUS II. MAX+PLUS II can integrate different types of design files, making it a very flexible and open development system.

Library Mapping Files

An EDIF Input File contains the logical description of a design, defined in terms of primitive gates and higher-level function names. Altera provides Library Mapping Files (**.lmf**) that map EDIF cell and port names, i.e., the third-party function and pin names, from the most popular CAE vendor libraries to equivalent MAX+PLUS II primitives and macrofunctions and their pin names. LMFs can map simple primitive gates, high-level functions, or even complete subdesigns. Figure 1 shows a sample LMF that maps an `$AND2` gate from the Mentor Graphics generic library to the equivalent MAX+PLUS II primitive (`AND2`).

Figure 1. Sample Library Mapping File

```
BEGIN
    FUNCTION    and2                (IN1, IN2)      % Altera symbol and input port  %
                                                    % names                          %
        RETURNS                     (OUT)           % Altera output port names       %
    FUNCTION    "user/genlib/$and2"("I0", "I1")     % Third-party symbol & input     %
                                                    % port names                      %
        RETURNS                     ("OUT")         % Third-party output port names %
END
```

Since LMFs are simple ASCII files, you can expand the list of mappings by creating or customizing your own LMFs. This LMF expansion is especially helpful if you are using proprietary ASIC libraries or creating libraries of custom functions. For more information about LMFs, refer to *Application Brief 96 (Generating Library Mapping Files)* in this handbook.

Combining EDIF & AHDL Designs

In the simplest case, a third-party design consists of a single-level schematic that contains only basic gate-level primitives such as AND and OR gates. More complex designs can contain TTL macrofunctions such as 74LS163 and 74LS373. In the most complex designs, custom symbols are added to the schematic to represent complete subdesigns. You can then use one of a variety of design entry methods to create the subdesigns. This design approach is known as the hierarchical or top-down design method.

In most CAE tools, TTL macrofunction symbols do not contain lower-level schematics and are considered "hollow bodies." During EDIF translation, if an EDIF netlist writer encounters a symbol that does not have an associated schematic, it automatically writes the symbol name and the port names in the EDIF netlist file in the form of cell descriptions (see Figure 2). However, for some CAE tools, you must assign attributes or body properties to these symbols to prevent the extraction of any information below the symbols. Similarly, you can add these attributes or properties to custom symbols so the EDIF netlist writer writes only the symbol and port names to the EDIF netlist file.

Figure 2. Sample Cell Descriptions

```
(cell (rename and2 "/user/gen_lib/$and2")
    (cellType generic)(view v
        (viewType netlist)(interface
            (port
        (rename OUT "OUT")
        (direction output))
            (port
        (rename I1"I1")(direction input))
            (port
        (rename I0 "I0")(direction input))
)))
```

When the MAX+PLUS II Compiler reads the EDIF Input File and LMF(s), it replaces the basic gates, functions, and custom symbols in the EDIF Input File with the equivalent function mapped by the LMF. If no equivalent Altera TTL macrofunction exists in the LMF for a particular cell in the EDIF Input File, the Compiler can map an AHDL description, as well as a graphic or waveform description, if available. For example, you can create a TDF that includes an AHDL function prototype that is functionally equivalent to the macrofunction and add the TDF name and the

corresponding EDIF cell name to the LMF. The AHDL description then replaces all instances of that EDIF cell.

The Compiler reads the top-level design file first. This file can be an EDIF Input File, an AHDL TDF, or a graphic design file created in MAX+PLUS II. If the top-level design file includes other subdesign files, the Compiler first checks the local directory for an EDIF Input File, GDF, WDF, or TDF with the same subdesign name and port names, then searches the MAX+PLUS II TTL MacroFunction Library and any user-defined libraries. In the workstation-based MAX+PLUS II, user libraries are specified in the **maxplus2.ini** file. To modify libraries for the PC-based MAX+PLUS II, you must use the **User Libraries** (Options menu) command.

The following sections of this application brief provide detailed instructions for combining AHDL TDFs with EDIF netlist files generated from Cadence Design Systems, Viewlogic Systems, and Mentor Graphics schematic designs. To incorporate an AHDL TDF into a higher-level schematic design, you must create a hollow-body symbol for the TDF. Hollow-body symbols are empty symbols that do not contain lower-level schematics.

Cadence

To create a hollow-body symbol that integrates an AHDL TDF into a schematic design created with Cadence's ValidGED, follow these steps:

1. Create a `BODY` file in ValidGED.
2. Create a part drawing in ValidGED by editing a ***<design name>*.part** file, then add `DRAWING` and `DEFINE` bodies to this drawing.
3. Create a dummy graphic file in ValidGED that contains only the ***<design name>*.body**.
4. Add the line `output chips` to your **compiler.cmd** file and compile the dummy graphic file. The ValidCOMPILER creates a file called **chips.dat** in your ***<current directory>*.wrk** file.
5. Rename the **chips.dat** file to **chips_prt** and move it to the ***<design name>*** directory containing your **body.x.x** file. You can delete all files in this directory except for the **body.x.x** and **chips_prt** files. You can then use this symbol in your design.

When you create an EDIF netlist file with **wedifnet**, Cadence's EDIF netlist writer, you must specify the hierarchical EDIF output format in your **wedifnet.cmd** file. If any EDIF cell has both a lower level of hierarchy and a mapping in an LMF, the LMF takes precedence.

Viewlogic

To create a hollow-body symbol that integrates an AHDL TDF into a schematic design created with Viewlogic's Viewdraw, follow these steps:

1. Choose the **Open Viewdraw Symbol** command from the Window menu.

2. Set the `level` attributes to `STD` and choose the **Change Block Type Module** command to indicate that the symbol has no hierarchy. You do not need to create hollow-body symbols for top-level schematics; they are only necessary for lower-level designs.

3. Start **edifneto** to write out the EDIF netlist file. You must specify the `-l std` option so that all functions with the `STD` attribute are not flattened.

4. The **edifneto** utility generates an EDIF netlist file with the name ***<project name>*.edn** in the ***<project name>*** directory. You should rename this file to ***<project name>*.edf**.

Mentor Graphics

To create a hollow-body symbol that incorporates an AHDL TDF into a schematic design created with Mentor Graphics' NETED, follow these steps:

1. Create a symbol with SYMED (Mentor Graphics' symbol editor).

2. Add the properties `COMP` and `ALTERA_MF` to the symbol. The `COMP` property defines the symbol name. The `ALTERA_MF` property is an arbitrary property name that is assigned to any custom symbol that is not described at a lower level by an associated schematic.

3. Use Mentor Graphics' EXPAND utility to convert the schematic into a Mentor Graphics database. You must specify that any symbol with the property `ALTERA_MF` should be treated as a primitive, and therefore will not be flattened during processing.

4. Use EDIFNET, Mentor Graphic's netlist writer, to translate the database into an EDIF netlist file. Make sure you use the `rename` option with EDIFNET. Without this option the EDIFNET utility automatically changes names that are illegal in EDIF to legal names, which may spoil the mapping for a symbol. The `rename` option ensures that EDIFNET preserves the original symbol name in the EDIF netlist file so that the mapping is correct. Use the `-n` option with EDIFNET to ensure that the `rename` construct in the EDIF netlist file contains both the original and new names.

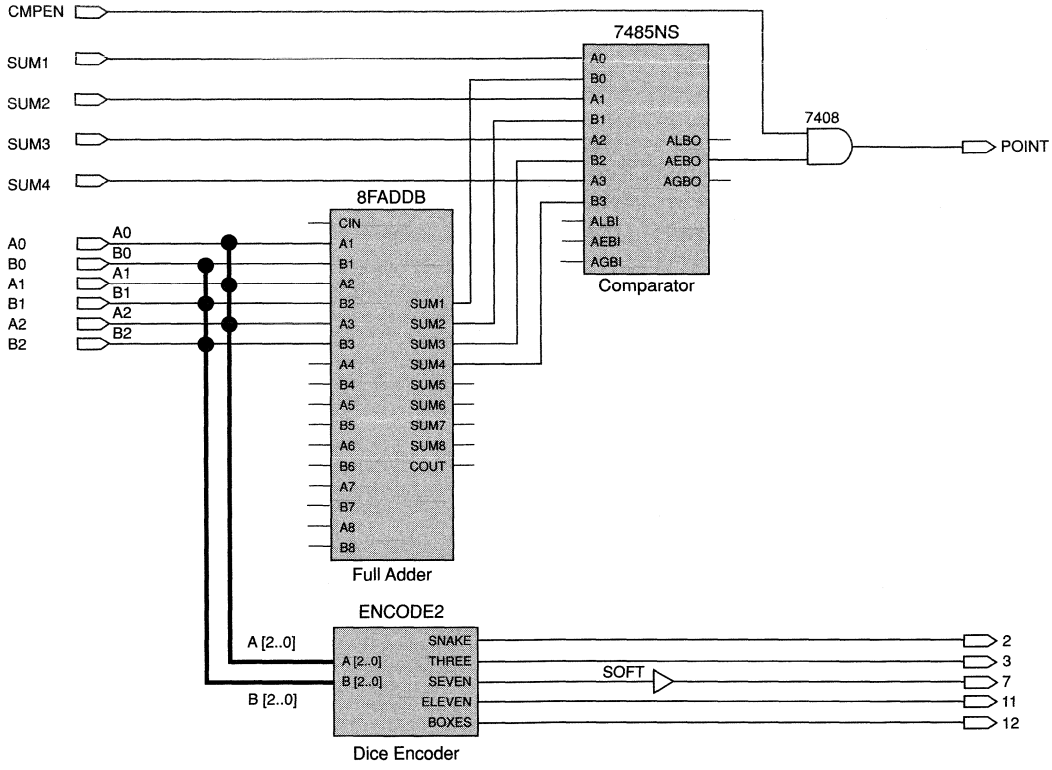
The EDIFNET utility writes the EDIF netlist file **edif.netlist** to the top-level design directory. You must rename this file to ***<project name>*.edf** so the MAX+PLUS II Compiler can compile it as an EDIF Input File.

Example

Figure 3 shows a Mentor Graphics' NETED schematic with three functions: 7485NS, 8FADDB, and ENCODE2. The symbols for the functions are created with SYMED.

Figure 3. NETED Schematic

The SOFT buffer is provided in the alt_max2 Altera Symbol Library for use in NETED schematics.



The logic for the 7485NS symbol is described in a NETED schematic. 8FADDB is mapped in an LMF to a MAX+PLUS II 8-bit fast adder macrofunction with the same name. See Figure 4.

Figure 4. 8FADDB Library Mapping File

```
BEGIN
    FUNCTION    8faddb     (CIN, A1, A2, A3, A4, A5, A6, A7, A8, B1, B2, B3, B4,
                           B5, B6, B7, B8)
        RETURNS            (COUT, SUM1, SUM2, SUM3, SUM4, SUM5, SUM6, SUM7, SUM8)
    FUNCTION "/maxplus/examples/maxgame/8FADDB"
                           ("CIN", "A1", "A2", "A3", "A4", "A5","A6", "A7", "A8",
                           "B1", "B2", "B3", "B4", "B5", "B6", "B7", "B8")
        RETURNS            ("COUT", "SUM1", "SUM2", "SUM3",
                           "SUM4", "SUM5", "SUM6", "SUM7", "SUM8")
END
```

ENCODE2 describes a function that encodes two 3-bit buses to generate five unique outputs (see Figure 5). Since 8FADDB and ENCODE2 are not defined by lower-level NETED schematics, the ALTERA_MF property is added to the symbols to prevent the EXPAND utility from extracting data from below the symbols.

Figure 5. ENCODE2.TDF

```
TITLE "TDF that performs the ENCODE function";
DESIGN IS encode2;

SUBDESIGN encode2
(
    a[2..0], b[2..0]                        : INPUT;
    snake, three, seven, eleven, boxes      : OUTPUT;
)

BEGIN

    snake = (a[]==0 AND b[]==0);

    three = (a[]==0 AND b[]==1) OR
            (a[]==1 AND b[]==0);

    seven = (a[]==0 AND b[]==5) OR
            (a[]==1 AND b[]==4) OR
            (a[]==2 AND b[]==3) OR
            (a[]==3 AND b[]==2) OR
            (a[]==4 AND b[]==1) OR
            (a[]==5 AND b[]==0);

    eleven = (a[]==4 AND b[]==5) OR
             (a[]==5 AND b[]==4);

    boxes = (a[]==5 AND b[]==5);

END;
```

Figure 6 shows the corresponding LMF for ENCODE2.

Figure 6. ENCODE2.LMF

```
BEGIN
    FUNCTION    encode2     (A2, A1, A0, B1, B2, B0)
        RETURNS             (SNAKE, THREE, SEVEN, ELEVEN, BOXES)
    FUNCTION    "/maxplus/examples/maxgame/encode2"
                            ("A2", "A1", "A0", "B2", B1", "B0")
        RETURNS             ("SNAKE", "THREE", "SEVEN", "ELEVEN", "BOXES")
END
```


LMF mappings are not required for all symbols. If the third-party symbol and port names match the primitive or macrofunction and its port names, MAX+PLUS II can call the lower-level design description without an LMF mapping. However, Altera recommends that you always use LMFs since they explicitly define mappings.

UNIX files are case-sensitive. You must make sure that all LMFs for workstation versions of MAX+PLUS II have the proper case for correct mapping. You should also provide mappings for all custom symbols so that MAX+PLUS II can find the corresponding macrofunction.

Conclusion

Altera's MAX+PLUS II software allows you to create logic designs with a variety of design entry methods. The ability to combine AHDL TDFs with EDIF netlist files generated from schematics allows you to describe each portion of your design with the most natural design entry method. This powerful MAX+PLUS II feature helps to speed up the design process and shorten overall development time.

Configuring Your PC
for MAX+PLUS II Software

April 1992, ver. 1 **Application Brief 98**

Introduction

IBM Personal Computers (PCs) and PC-compatible computers can be configured in a variety of ways to improve system operation and performance. For example, memory buffers, high memory, disk caching, and extended and expanded memory are system features controlled by computer configuration. This application brief recommends the following procedures to improve system performance while running MAX+PLUS II on a PC:

- ❑ Optimize your hardware configuration
- ❑ Fine-tune the CONFIG.SYS system configuration file
- ❑ Set up a Windows swap file
- ❑ Maintain your hard disk
- ❑ Check the system memory configuration.

For detailed information on optimizing your system for Windows, refer to the *Microsoft Windows User's Guide*.

Optimize the Hardware Configuration

Altera recommends the following system configuration for optimum MAX+PLUS II performance:

- ❑ 33-MHz 386DX- or 486-based PC-AT, PS/2, or compatible computer
- ❑ 8 Mbytes of memory, configured as extended memory
- ❑ DOS version 5.0 or higher
- ❑ Microsoft Windows version 3.1 or higher running in enhanced mode only
- ❑ 30 Mbytes of free disk space before installation of MAX+PLUS II on a hard drive with a minimum of 17 ms access time and 15 Mbits/s transfer rate

Fine-Tune CONFIG.SYS

To fine-tune the CONFIG.SYS file and improve the performance of MAX+PLUS II running under Windows, you should use the following variable settings:

```
FILES=30
BUFFERS=20
DEVICE=C:\DOS\SMARTDRV.SYS 2048 1024
DEVICE=C:\DOS\HIMEM.SYS
DOS=HIGH,UMB
DEVICEHIGH=C:\SYSTEM\<one or more names of device drivers used with your
    computer>
```

FILES Variable

FILES specifies the maximum number of files each application can have open at one time. By increasing or decreasing this number, you can control how many windows can be open in MAX+PLUS II. Altera recommends that you set FILES to 30.

BUFFERS Variable

BUFFERS sets the number of disk buffers that are allocated in memory when your computer starts. These buffers may improve disk access time. The larger the number, the more conventional memory is used. Altera recommends that you set BUFFERS to 20.

DEVICE Variable

DEVICE specifies device drivers. The CONFIG.SYS file should define only the device drivers that you absolutely need. Microsoft supplies two drivers that can help to run MAX+PLUS II efficiently: SMARTDrive, a disk-caching program, and HIMEM, an extended memory manager. You can also use a third-party extended memory manager such as QEMM by Quarterdeck.

SMARTDrive

Microsoft SMARTDrive is a disk-caching program for computers with a hard disk and extended or expanded memory. It reduces the amount of time the computer needs to read data from the hard disk by allocating a block of memory to hold the last set of data read from the hard disk. SMARTDrive has two variables: the first defines the maximum SMARTDrive size, the second defines the minimum SMARTDrive size (in Kbytes). SMARTDrive uses the range defined by the two variables to determine the actual size of memory to be used for caching the hard disk. Altera recommends setting the first variable to 2048 and the second to 1024.

SMARTDrive is provided with Microsoft DOS 5.0 and Windows 3.1. If you have both products, you should use the version of SMARTDrive that is shipped with DOS 5.0.

HIMEM

Microsoft HIMEM manages the extended memory of the High Memory Area. It supervises your system's extended memory so that no two applications can use the same memory at the same time, allowing Windows to take full advantage of available memory and run at maximum speed.

HIMEM is provided with Microsoft DOS 5.0 and Windows 3.1. If you have both products, you should use the version of HIMEM that is shipped with DOS 5.0.

DOS Variable

You should set the `DOS` variable, available with DOS 5.0, to `HIGH`. This option prompts DOS to load itself into extended memory, freeing most of the conventional memory for executing programs. The `UMB` option tells DOS to allocate memory for device drivers that are loaded into high memory.

DEVICEHIGH Variable

The `DEVICEHIGH` variable, available with DOS 5.0, loads the specified device driver into high memory. Refer to the specific device driver manual to ensure that the driver can be loaded in upper memory.

Set up a Windows Swap File

Microsoft Windows provides Swapfile, a program that uses the hard disk for virtual memory. The swap file reserves space on the hard disk to store code and data temporarily when available memory runs low. Windows provides a temporary and a permanent swap file. By expressly creating a permanent 6- to 8-Mbyte swap file, i.e., one that stays on the hard disk until you remove it, you can greatly enhance system performance.

To set up the swap file with the Swapfile application, go through the following steps:

1. Type `win /r` ↵ at the DOS prompt. The Program Manager window is displayed.
2. Choose **Run** from the File menu. The **Run** dialog box is displayed.
3. Type `swapfile` in the *Command Line* box.
4. Turn off the *Run Minimized* option.
5. Choose **OK**.

Swapfile examines your hard disk drive partitions to see how much contiguous (i.e., unfragmented) space is available on each partition, and displays information about the first partition that has sufficient space. If the swap file is larger than the largest contiguous memory space available on the hard disk, you must delete files from your hard disk or use an unfragmenting program such as Norton Speed Disk. For information on how to run Swapfile, see the ***Microsoft Windows User's Guide***.

To make sure that the swap file is correctly set up for Windows, double-click Button 1 on the Windows Setup icon in the Program Manager's Main group window. The **Windows Setup** dialog box shows the number of Kbytes used for the swap file.

Maintain Your Hard Disk

Regular disk maintenance speeds up disk access and improves MAX+PLUS II operation.

Altera recommends that you use Norton Disk Utilities regularly. Use Norton Speed Disk to unfragment the hard disk and move files to contiguous areas, and Norton Disk Doctor to repair disk problems.

Check Memory Configuration

You should use the DOS 5.0 `MEM` command to verify that your system memory is configured correctly. The following information is displayed:

- Number of bytes of total conventional memory
- Number of bytes available to MS-DOS
- Largest executable program size. You should have at least 570 Kbytes available for executable programs. If you have less than that, you should use the DOS 5.0 `DEVICEHIGH` variable to load device drivers into upper memory.
- Number of bytes of total contiguous extended memory
- Number of bytes available in contiguous extended memory
- Number of bytes available in XMS memory. This number must be at least 4 Mbytes. If this number is low, you should add memory to your machine to enhance your system configuration.
- MS-DOS resident in High Memory Area

The following list shows the type of information displayed with the `MEM` command:

```
655360   bytes total conventional memory
655360   bytes available to MS-DOS
584832   largest executable program size
7602176  bytes total contiguous extended memory
0        bytes available contiguous extended memory
5270528  bytes available XMS memory
         MS-DOS resident in High Memory Area
```

Conclusion

Proper PC configuration provides optimum MAX+PLUS II performance. If you have any questions on configuring your PC, call Altera Applications at (800) 800-EPLD.

Design Flow between Workstations & PCs

April 1992, ver. 1

Application Brief 99

Introduction

While the workstation's power and standardized tools are best suited for complex design entry and board-level simulation, the PC provides easy-to-use and less expensive design and compilation tools. The PC-based MAX+PLUS II development system draws on the strengths of both platforms. You can create a design on the SPARCstation with a third-party CAE tool, compile it on the PC, and then use a third-party simulator for board-level simulation. The bridge between the two platforms is the Electronic Design Interchange Format (EDIF 2 0 0), which is an industry-standard netlist that allows you to transfer designs from one CAE tool to another. Most CAE software packages that produce and read EDIF netlist files can interface with MAX+PLUS II, which contains an EDIF Netlist Reader and EDIF Netlist Writer for quick and easy processing.

If you prefer to compile your EPLD designs on the SPARCstation platform, Altera also offers a workstation-based MAX+PLUS II Compiler. For more information, contact the Altera Applications department at (800) 800-EPLD.

This application brief describes the interface between the PC-based MAX+PLUS II software and third-party workstation CAE tools. It includes specific guidelines for the Cadence, Viewlogic, and Mentor Graphics interfaces. Extensive knowledge of your CAE tools and some understanding of MAX+PLUS II are assumed.

Design Flow

The following steps show how to take advantage of the best of both platforms and how to make entering, processing, simulating, and programming quick and easy:

1. Enter the design on the SPARCstation with your CAE tool and generate an EDIF netlist file, then change the filename extension to **.edf**.

2. Compile the EDIF Input File (**.edf**) with the PC-based MAX+PLUS II Compiler. The Compiler's EDIF Netlist Writer generates an EDIF Output File (**.edo**) that contains post-synthesis timing and architectural information.

3. Transfer the EDIF Output File to the SPARCstation for device- or board-level simulation with your third-party simulation tool.

4. Program one or more EPLDs either with the MAX+PLUS II programming hardware and software on the PC, or with your third-party workstation programming tools.

Design Entry

You have two options for entering designs on the workstation for MAX+PLUS II:

- You can use your workstation CAE schematic capture program to enter a design and generate an EDIF netlist file. With a few minor conversions (discussed in later sections of this application brief), this file becomes an EDIF Input File that MAX+PLUS II can compile.

- You can use any standard ASCII text editor to enter a MAX+PLUS II-compatible Altera Hardware Description Language (AHDL) Text Design File (**.tdf**).

You can also combine AHDL and EDIF design entry. For more information, refer to *Application Brief 97 (Integrating EDIF Files with AHDL Files in Hierarchical Projects)* in this handbook.

For best results, enter a logic design as a hierarchical design (called a "project" in MAX+PLUS II). A hierarchical project is a multi-level, modular description of a logic design. Each of the lower-level functions contains a piece of the complete project and is connected through its input and output ports to the design file at the next higher level. Hierarchical projects enhance readability, clarity, and modularity, thus decreasing testing time and ultimately improving time-to-market. The MAX+PLUS II EDIF Netlist Reader reads hierarchical EDIF Input Files with the help of Library Mapping Files (**.lmf**).

Using Library Mapping Files

LMFs map the workstation functions to equivalent MAX+PLUS II primitives or macrofunctions, which are optimized for Altera EPLD architectures. Altera provides LMF libraries for Cadence, Viewlogic, and Mentor Graphics. You can also customize existing LMFs or create your own LMF for a particular function that is not mapped in the Altera-provided LMFs. For more information on creating LMFs and a list of existing MAX+PLUS II macrofunction mappings, refer to *Application Brief 96 (Generating Library Mapping Files)* in this handbook.

Follow these steps to use LMFs:

1. On your workstation, enter your logic design with functions that are equivalent to the MAX+PLUS II TTL macrofunctions and primitives that are mapped in the Altera-provided LMF. To use macrofunctions and primitives that are not mapped in this LMF, you should also modify an existing LMF or create a new LMF that includes the additional mappings.

2. Use the EDIF netlist writer for your CAE tool to generate a hierarchical EDIF netlist file for your design that preserves the function names. Change the filename extension to **.edf**.

3. Start MAX+PLUS II. If necessary, open the Compiler.

4. To specify an LMF, choose **EDIF Netlist Reader** (Options menu) select *LMF #1*, and type or select the name of the desired LMF (usually the Altera-provided LMF). If you have created or customized an LMF with project-specific mappings, you can specify it as *LMF #2*.

Integrating Other Design Files into EDIF Netlist Files

You can also create a TDF, Waveform Design File (**.wdf**), or Graphic Design File (**.gdf**) with the PC-based MAX+PLUS II software and integrate the file into a higher-level schematic design on the workstation.

To integrate a TDF, WDF, or GDF into a schematic design:

1. Create a "hollow-body" symbol in your workstation design that represents the MAX+PLUS II TDF, WDF, or GDF. A hollow-body symbol is an empty symbol that does not contain a lower-level schematic.

2. Use the EDIF netlist writer for your CAE tool to generate a hierarchical EDIF netlist file for your design that includes this hollow-body workstation symbol name. Change the filename extension to **.edf**.

3. Create an LMF that maps the hollow-body workstation symbol. You can specify this LMF as an input to the Compiler in MAX+PLUS II, as described earlier.

For more information on integrating TDFs, WDFs, or GDFs into schematic designs, refer to *Application Brief 97 (Integrating EDIF Files with AHDL Files in Hierarchical Projects)* in this handbook.

Design Processing

The MAX+PLUS II Compiler can synthesize and minimize the design logic in the EDIF Input File. The Compiler automatically optimizes designs for Classic, MAX 5000, MAX 7000, or STG architecture; selects an appropriate device from the device family you specified; and fits the design into one or more EPLDs of that family. You can choose specific devices, or the Compiler can select them automatically. The EDIF Netlist Writer module of the Compiler also generates an EDIF Output File that can be exported to the workstation.

Design Simulation

You can transfer the MAX+PLUS II EDIF Output File to your workstation to perform device- and board-level simulation. Several CAE systems include their own EDIF netlist readers that prepare the MAX+PLUS II EDIF Output File for simulation on the workstation. Others use third-party EDIF netlist readers, such as Logic Modeling SmartModels.

Device Programming

The MAX+PLUS II Compiler produces a Programmer Object File (**.pof**) and, optionally, a JEDEC File (**.jed**). You can program a device with these programming files, the MAX+PLUS II software, and Altera programming hardware. The hardware consists of a programming card, a Master Programming Unit (MPU), and the appropriate programming adapter.

You can also use third-party programming hardware and software to program an EPLD with the POF or JEDEC File on the workstation. You must transfer the binary POF or JEDEC File with binary-mode file transfer protocol (FTP).

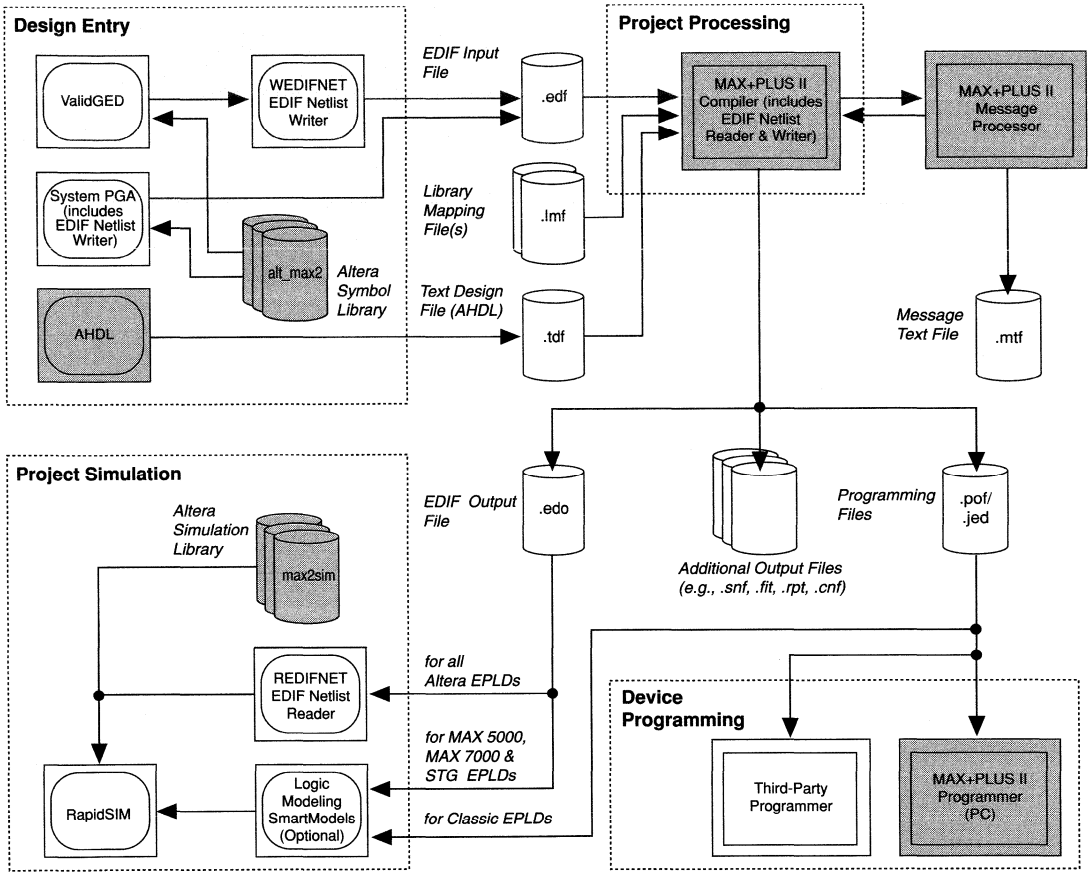
Cadence Interface

You can enter your design on the SPARCstation with Cadence CAE tools, compile it on the PC with MAX+PLUS II, and simulate it with Cadence simulation tools. Figure 1 illustrates this design flow.

The following software is required:

- MAX+PLUS II
- ValidGED version 10.0 or higher or SystemPGA version 2.1 or higher
- **wedifnet** version 1.0 or higher
- ValidCOMPILER version 1.4 or higher
- LSTTL_lib (from Cadence)
- **redifnet** version 1.0 or higher or Logic Modeling SmartModels version 31 or higher
- RapidSIM simulator version 1.0 or higher

Figure 1. Altera/Cadence Interface



For more detailed information on the Cadence interface, refer to the stand-alone version of *Application Note 29 (EPLD Design with Valid Logic and MAX+PLUS II Software)*.

Design Entry Guidelines

You can enter a design in one of Cadence's schematic capture packages (ValidGED or SystemPGA), then create an EDIF netlist file either with Cadence's EDIF netlist writer (**wedifnet**) or the EDIF netlist writer built into SystemPGA. The Altera symbol library **alt_max2** contains Altera primitives, including `MCELL`, `SOFT`, `EXP`, and `GLOBAL`, for use in ValidGED and SystemPGA. For a copy of this library, contact Altera Applications.

Follow these guidelines when entering designs in ValidGED or SystemPGA:

- To produce an EDIF netlist file that MAX+PLUS II can read, use the hierarchical expansion style in the **wedifnet.cmd** file.
- The **wedifnet** utility and System PGA's EDIF netlist writer both create an EDIF netlist file with the extension **.edif**. You must change the extension to **.edf** before MAX+PLUS II can process it as an EDIF Input File.

Design Processing Guidelines

The MAX+PLUS II Compiler reads the EDIF Input File as it would any other design file and then compiles it with **vld_bas.lmf**, the Altera-provided LMF for Cadence that includes mappings for many standard macrofunctions and primitives. The Compiler then generates a POF and an optional JEDEC File for programming one or more Altera devices, and creates an EDIF Output File for simulation.

Follow these steps when compiling a design (project) with MAX+PLUS II:

1. To specify the EDIF Input File as a project in MAX+PLUS II, choose the **Project Name** command (File menu), then enter or select the correct filename with the extension **.edf**.
2. To specify an LMF, choose **EDIF Netlist Reader** (Options menu) and in the dialog box type or select the name of the desired LMF (usually the Altera-provided **vld_bas.lmf**). If you wish to specify project-specific mappings, you can create your own LMF or customize a version of **vld_bas.lmf** and specify it as *LMF #2*.
3. To create an EDIF Output File:
   - Choose the **EDIF Netlist Writer** command (Options menu) and turn on the *Generate EDIF Output File(s)* option in the dialog box.
   - Turn on the *EDIF Command File* option and specify the Altera-supplied **valid.edc** file. This EDIF Command File (**.edc**) renames the NOT, NAND, AND, OR, XOR, DELAY, TRI, LATCH, DFF, DFFE, and FILTER cells so that they are preceded by an "AL" in the EDIF Output File. Renaming these cells eliminates conflicts with the corresponding elements in the Cadence Standard Library. See Figure 2.

Figure 2. The valid.edc EDIF Command File

This file renames cells in the EDIF Output File to avoid conflicts with the Cadence library elements.

```
CELL    NOT     ALNOT
CELL    NAND    ALNAN
CELL    AND     ALAND
CELL    OR      ALOR
CELL    XOR     ALXOR
CELL    DELAY   ALDELAY
CELL    TRI     ALTRI
CELL    LATCH   ALLATCH
CELL    DFF     ALDFF
CELL    DFFE    ALDFFE
CELL    FILTER  ALFILTER
```


Design Simulation Guidelines

To prepare the compiled design for simulation in RapidSIM, use the Cadence EDIF netlist reader (**redifnet**), with the Altera **max2sim** simulation library to read the EDIF Output File. For a free copy of the **max2sim** library, call Altera Applications at (800) 800-EPLD. You can also use Logic Modeling SmartModels, which support EDIF Output Files for MAX 5000, MAX 7000, and STG EPLDs, and JEDEC Files for Classic EPLDs.

Follow these guidelines when preparing designs for simulation with RapidSIM:

- To transfer designs to RapidSIM, use **max2sim** with **redifnet**.
- The **redifnet** utility converts the EDIF Output File into ValidGED body and connectivity files for use in RapidSIM. To eliminate filename confusion, place the EDIF Output File in a destination design directory.

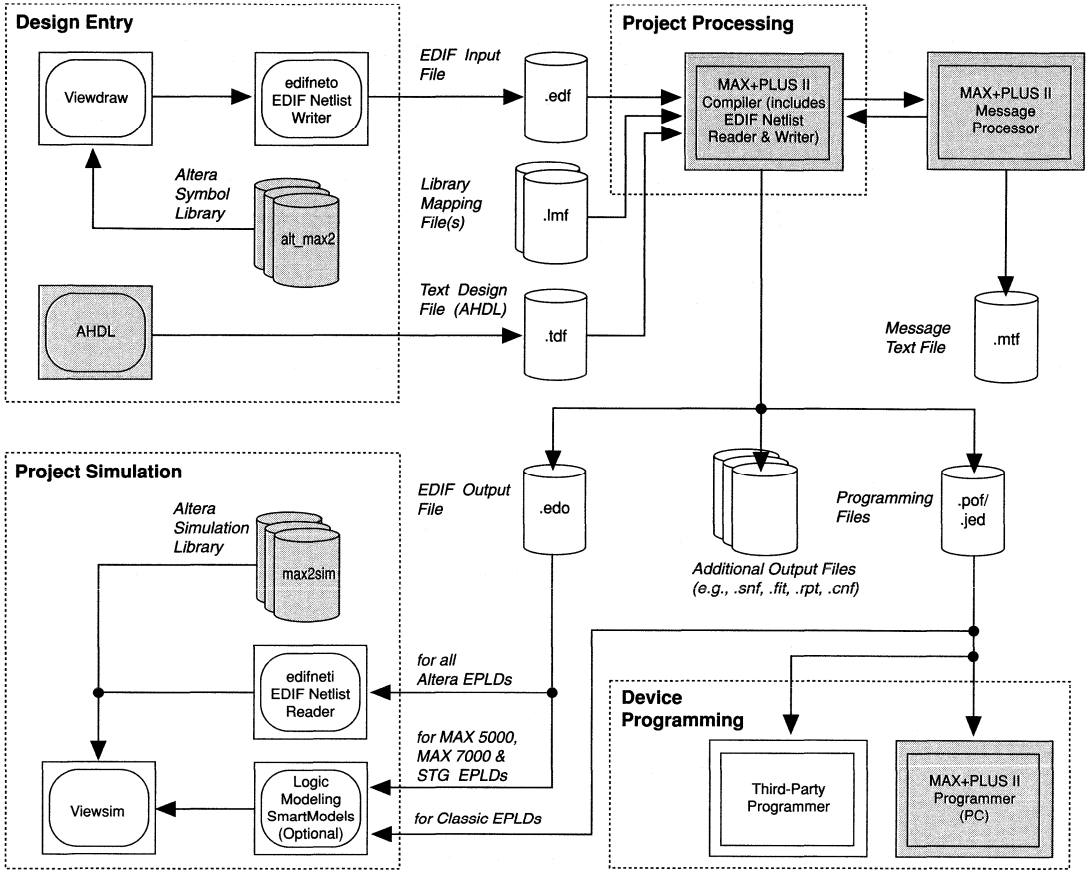
Viewlogic Interface

You can enter your design on the SPARCstation with Viewlogic CAE tools, compile it on the PC with MAX+PLUS II, and simulate it with Viewlogic simulation tools. Figure 3 illustrates this design flow.

The following software is required:

- MAX+PLUS II
- Viewdraw version 4.0 or higher
- Viewsim version 4.0 or higher
- Viewwave version 4.0 or higher
- 74LS library (from Viewlogic)
- **edifneto** version 4.0 or higher
- **edifneti** version 1.0 or higher or Logic Modeling SmartModels version 31 or higher

Figure 3. Altera/Viewlogic Interface



For more detailed information on the Viewlogic interface, refer to the stand-alone version of *Application Note 30 (EPLD Design with Viewlogic and MAX+PLUS II Software)*.

Design Entry Guidelines

You can enter the design in Viewlogic's schematic capture package (Viewdraw), then create an EDIF netlist file with Viewlogic's EDIF netlist writer (**edifneto**). The Altera symbol library **alt_max2** contains Altera primitives, including MCELL, SOFT, EXP, and GLOBAL, for use in Viewdraw. For a copy of this library, contact Altera Applications.

Follow these guidelines when entering designs in Viewdraw:

- Assign the attribute `PINTYPE=INOUT` to all Viewlogic bidirectional pins.
- Change the attribute for all Viewlogic primitive tri-state outputs from `PINTYPE=TRI` to `PINTYPE=OUT` or use the Altera-modified output primitives from the **alt_max2** Altera symbol library.
- Altera recommends using a `74LS74` function to implement a D flipflop and a `74LS373` function to implement a latch instead of Viewlogic's `UDFDL` universal D flipflop and D latch. Although MAX+PLUS II maps the `UDFDL` function, you should avoid using it because the flipflop and the latch cannot be used at the same time.
- All standard macrofunctions and symbols that represent Altera TDFs in the Viewlogic design must have the attribute `level=std`. You must also choose the **Change Block Type Module** command to indicate that the symbol has no hierarchy.
- Use the `-l std` option with **edifneto**. This statement forces the EDIF netlist writer to flatten the EDIF netlist file only to the standard level.
- The **edifneto** utility creates an EDIF netlist file with the extension **.edn**. You must change the extension to **.edf** before MAX+PLUS II can process it as an EDIF Input File.

Design Processing Guidelines

The MAX+PLUS II Compiler reads the EDIF Input File as it would any other design file and then compiles it with **vwl_bas.lmf**, the Altera-provided LMF for Viewlogic that includes mappings for many standard macrofunctions and primitives. The Compiler then generates a POF or an optional JEDEC File that can be used to program one or more Altera devices, and creates an EDIF Output File for simulation.

Follow these steps when compiling a design (project) with MAX+PLUS II:

1. To specify the EDIF Input File as a project in MAX+PLUS II, choose the **Project Name** command (File menu), then enter or select the correct filename with the extension **.edf**.
2. To specify an LMF, choose **EDIF Netlist Reader** (Options menu) and turn on the *LMF #1* option (Options menu), then type or select the name of the desired LMF (usually the Altera-provided **vwl_bas.lmf**). If you wish to specify project-specific mappings, you can customize a version of **vwl_bas.lmf** or create your own and specify it as *LMF #2*.

Figure 4. The vwl.edc EDIF Command File

This file renames cells in the EDIF Output File to avoid conflicts with the Viewlogic library elements.

```
CELL        AND         A_AND
CELL        NAND        A_NAN
CELL        OR          A_OR
CELL        NOT         A_NOT
CELL        XOR         A_XOR
CELL        DELAY       A_DELAY
CELL        TRI         A_TRI
CELL        LATCH       A_LATCH
CELL        DFF         A_DFF
CELL        DFFE        A_DFFE
CELL        FILTER      A_FILTER
```

3. Turn on the *VCC* option in the **EDIF Netlist Reader** dialog box and type VDD in the box.

4. To create an EDIF Output File:

   - Choose the **EDIF Netlist Writer** command (Options menu) and turn on the *Generate EDIF Output File(s)* option in the dialog box.
   - Turn on the *EDIF Command File* option and specify the Altera-supplied **vwl.edc** file. This EDIF Command File renames the AND, NAND, OR, NOT, DELAY, XOR, TRI, LATCH, DFF, DFFE, and FILTER cells in the MAX+PLUS II EDIF Output File so that they are all preceded by "A_." Renaming these cells eliminates conflicts with the corresponding elements in the Viewlogic library. See Figure 4.
   - Select the *Property* option for *Delay Constructs* in the **EDIF Netlist Writer** dialog box. This option instructs the Compiler to generate timing constructs as attributes (or properties) rather than port and path delays.

Design Simulation Guidelines

To prepare the compiled design for simulation in Viewsim, use Viewlogic's EDIF netlist reader (**edifneti**), with the Altera **max2sim** simulation library to read the EDIF Output File. For a free copy of the **max2sim** library, contact Altera Applications. You can also use Logic Modeling SmartModels, which support EDIF Output Files for MAX 5000, MAX 7000, and STG EPLDs, and JEDEC Files for Classic EPLDs.

Follow these guidelines when preparing the design for simulation with Viewsim:

- To correctly transfer designs to Viewsim, the Viewlogic simulator, use **max2sim**, the Altera simulation library, with **edifneti**, the Viewlogic netlist reader.
- Use the Altera-supplied configuration file by specifying the option `-a edifatts.cfg` on the command line when you run **edifneti**. This file installs "at" symbols (`@`) on the timing parameters inside the wire file so that Viewsim can recognize the timing data.
- Change the wirelist cell names from Module (`m`) to Wire (`w`) type so that Viewsim can recognize the Altera timing information. Use the **m2w** utility to convert the cell designations in the wirelist file from type `m` to type `w`.
- Run **vsm** to produce a Viewsim file (**.vsm**), then edit the file to change `VCC` back to `VDD`.

Mentor Graphics Interface

You can enter your design on the SPARCstation with the CAE tools from Mentor Graphics release 7.0, compile it on the PC with MAX+PLUS II, and simulate the design with Mentor Graphics simulation tools. Figure 5 illustrates this design flow.

The following software is required:

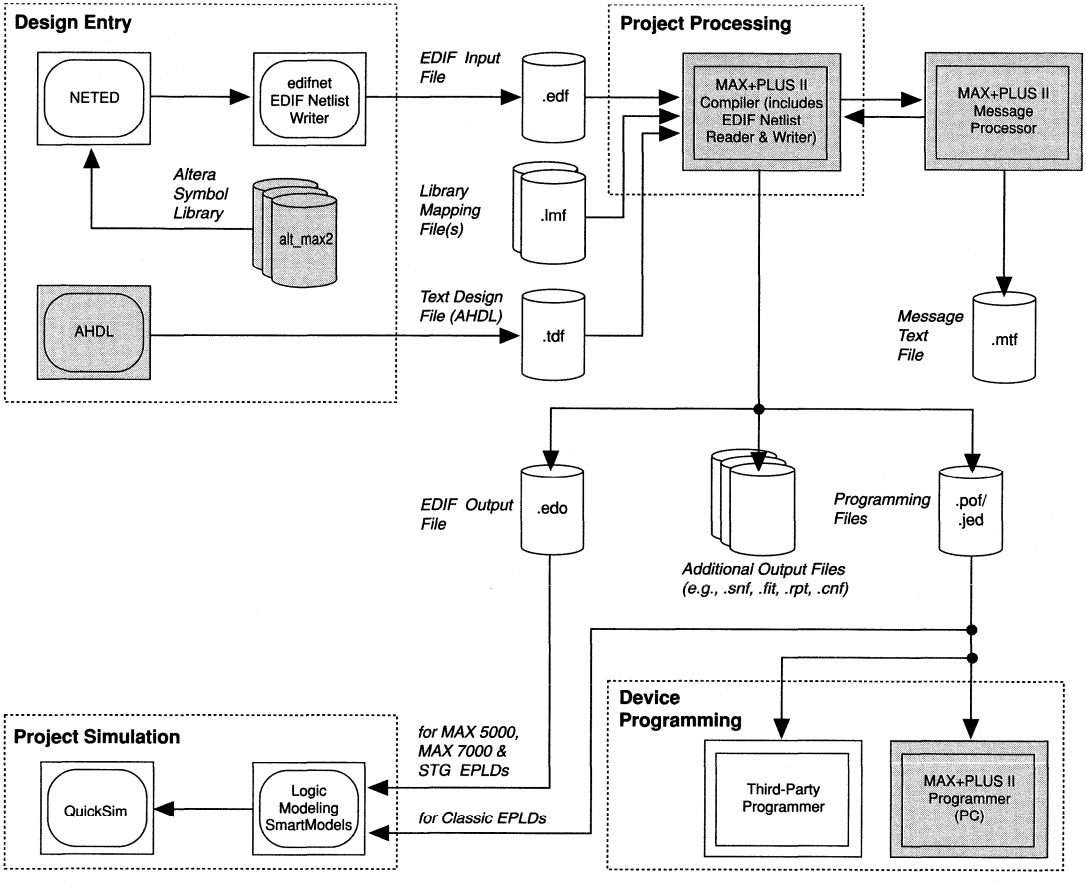
- MAX+PLUS II
- NETED version 7.0
- EXPAND version 7.0
- QuickSim version 7.0
- EDIFNET version 7.0_2.5
- 74LS_LIB (from Mentor Graphics)
- Logic Modeling SmartModels version 31 or higher

For more detailed information on the Mentor Graphics interface, contact Altera Applications.

Design Entry Guidelines

You can enter the design in the Mentor Graphics schematic capture package (NETED), then create an EDIF netlist file with the Mentor Graphics EDIF netlist writer (EDIFNET). The Altera symbol library **alt_max2** contains Altera primitives, including `MCELL`, `SOFT`, `EXP`, and `GLOBAL`, for use in NETED. For a copy of this library, contact Altera Applications.

Figure 5. Altera/Mentor Graphics Interface



Follow these guidelines when entering a design in NETED:

- If you use the PORTOUT symbol in NETED to specify output and bidirectional pins, all pins will be identified as bidirectional pins in the EDIF netlist file. To avoid this error, create a top-level symbol for the design and assign the OUT property to output pins and the IXO property to bidirectional pins.

- To create a hierarchical EDIF netlist file, assign the ALTERA_MF property to macrofunctions that should not be flattened. This property instructs the EXPAND utility to pass the function name directly to the EDIF netlist file. Then create an LMF to map the EDIF netlist function to an Altera TDF, WDF, or GDF.

- Make sure you use the `-n` option with EDIFNET. The EDIFNET utility automatically changes names that are illegal in EDIF to legal names, and may spoil the mapping for a symbol. However, using the `-n` option ensures that EDIFNET preserves the original symbol name in the EDIF netlist file so that the mapping is correct. Use the `-n` option with EDIFNET to ensure that the `rename` construct in the EDIF netlist file contains both the original and new names.

- The EDIFNET utility produces an EDIF netlist file called **edif.netlist**. You must rename it to ***<design name>*.edf** so that MAX+PLUS II can process it as an EDIF Input File.

Design Processing Guidelines

The MAX+PLUS II Compiler reads the EDIF Input File as it would any other design file, and then compiles it with **mnt_bas.lmf**, the Altera-provided LMF for Mentor Graphics that includes mappings for most standard macrofunctions and primitives. It then generates a POF and an optional JEDEC File that can be used to program one or more Altera devices, and an EDIF Output File for simulation.

Follow these steps when compiling a design (project) in MAX+PLUS II:

1. To specify the EDIF Input File as a project in MAX+PLUS II, choose the **Project Name** command (File menu), then enter or select the correct filename with the extension **.edf**.

2. To specify an LMF, choose EDIF Netlist Reader (Options menu) and turn on the *LMF #1* option, then type or select the name of the desired LMF (usually the Altera-provided **mnt_bas.lmf**). If you wish to specify project-specific mappings, you can create your own LMF or customize a version of **mnt_bas.lmf** and specify it as *LMF #2*.

3. Turn on the *GND* option under *Signal Names* in the **EDIF Netlist Reader** dialog box and type `GROUND` in the text box.

4. To create an EDIF Output File:

   - Choose the **EDIF Netlist Writer** command (Options menu) and turn on the *Generate EDIF Output File(s)* option in the dialog box.
   - Make sure the *EDIF Command File* option in the **EDIF Netlist Writer** dialog box is turned off.

Design Simulation Guidelines

After you have compiled the design with MAX+PLUS II, you should prepare the EDIF Output File for device- or board-level simulation in Mentor Graphics' QuickSim.

Follow these guidelines when preparing a design for simulation in QuickSim:

- Change the filename extension of the EDIF Output File from **.edo** to **.edif** for use with the Mentor Graphics tools.
- For MAX 5000, MAX 7000, and STG devices, submit the EDIF Output File to the Logic Modeling SmartModels to prepare the design for simulation in QuickSim. For Classic EPLDs, submit the MAX+PLUS II JEDEC File to the SmartModels to prepare the design for simulation in QuickSim.

Conclusion

MAX+PLUS II is the only programmable logic development system for the PC that has a fully integrated EDIF Netlist Reader and EDIF Netlist Writer. MAX+PLUS II allows you to enter your design on your Sun Workstation, compile it on a PC, and transfer it back to the workstation for simulation. Altera provides simulation libraries and LMFs that let MAX+PLUS II interface easily with popular workstation products from three vendors: Cadence, Viewlogic, and Mentor Graphics. Most other workstation CAE tools that produce an EDIF netlist file can also interface with MAX+PLUS II. For more information on the design flow between workstations and PCs, contact Altera Applications at (800) 800-EPLD.

Understanding EPLD Timing

April 1992, ver. 1

Application Brief 100

Introduction

Combining advanced technology CMOS EPLDs with the sophisticated MAX+PLUS II development system has made Altera a formidable force in programmable logic. Altera EPLDs provide device performance that is consistent from simulation to application. Before programming an EPLD, you can determine the worst-case timing delays for any design. You can calculate propagation delays with the MAX+PLUS II Timing Analyzer, or with the timing models given in this application brief and the timing parameters listed in individual EPLD data sheets. Both methods yield the same results.

This application brief defines EPLD internal delay parameters and AC timing characteristics, and illustrates the timing model for each family of Altera devices.

Familiarity with EPLD architecture and characteristics is assumed. Refer to individual device data sheets for complete descriptions of the architectures.

Internal EPLD Delay Parameters

Within an EPLD, timing delays contributed by individual architectural elements are called microparameters. The following list defines microparameters for Classic, MAX 5000, MAX 7000, and Synchronous Timing Generator (STG) EPLDs. Individual device data sheets for MAX 5000, MAX 7000, and STG EPLDs give the values for these parameters; microparameters for Classic EPLDs are listed in this application brief.

t_{IN} Input pad and buffer delay. In Classic, MAX 5000, and STG EPLDs, it is the time required for a dedicated input pin to drive the true and complement data input signal into the logic array(s). In MAX 7000 devices, it is the time required for a dedicated input pin to drive the input signal into the Programmable Interconnect Array (PIA) or into the global control array.

t_{IO} I/O input pad and buffer delay. This delay applies to I/O pins used as inputs. In Classic EPLDs, it is the delay added to t_{IN}. In MAX 5000 EPLDs with a single Logic Array Block (LAB) and STG devices, it is the delay from the I/O pin to the logic arrays. In MAX 7000 and multi-LAB MAX 5000 EPLDs, it is the delay from the I/O pin to the PIA.

t_{PIA} Programmable Interconnect Array delay. In MAX 7000 and multi-LAB MAX 5000 EPLDs, it is the delay incurred on signals that require routing through the PIA.

t_{EXP} Expander array delay. In MAX 5000 and STG EPLDs, it is the delay of a signal through the AND-NOT structure of the shared expander product-term array that is fed back into the logic array.

t_{SEXP} Shared expander array delay. In MAX 7000 EPLDs, it is the delay of a signal through the AND-NOT structure of the shared expander product-term array that is fed back into the logic array.

t_{PEXP} Parallel expander delay. In MAX 7000 EPLDs, it is the additional delay incurred by adding parallel expander product terms to the macrocell product terms. For each group of up to five parallel expanders added to a single function, an additional t_{PEXP} delay is added to the timing path.

t_{ICS} Global Clock delay. In Classic and MAX 5000 EPLDs, it is the delay from the dedicated Clock pin to a register's Clock input.

t_{GLOB} Global control delay. In MAX 7000 and STG EPLDs, it is the delay from a dedicated input pin to any global control function in a macrocell or I/O control block.

t_{LAC} Logic array control delay. In MAX 5000, MAX 7000, and STG EPLDs, it is the AND array delay for register control functions such as Preset, Clear, and Output Enable.

t_{IC} Array Clock delay. The delay through a macrocell's Clock product term to the register's Clock input.

t_{EN} Register Enable delay. In MAX 7000 EPLDs, it is the register AND array delay from the PIA to the register Enable input.

t_{CLR} Register Clear time. The delay from the time when the register's asynchronous Clear input is asserted to the time the register output stabilizes at logical low.

t_{PRE} Register Preset time. The delay from the time when the register's asynchronous Preset input is asserted to the time the register output stabilizes at logical high.

t_{LAD} Logic array delay. The time a logic signal requires to propagate through a macrocell's AND-OR-XOR structure.

t_{RD} Register delay. In MAX 5000, MAX 7000, and STG EPLDs, it is the delay from the rising edge of the register's Clock to the time the data appears at the register output.

t_{SU} Register setup time. The time required for a signal to be stable at the register input before the Clock's rising edge to ensure that the register correctly stores the input data.

t_{H} Register hold time. The time required for a signal to be stable at the register input after the register Clock's rising edge to ensure that the register correctly stores the input data.

t_{COMB} Combinatorial buffer delay. In MAX 5000, MAX 7000, and STG EPLDs, it is the delay from the time when a combinatorial logic signal bypasses the programmable register to the time it becomes available at the macrocell output.

t_{LATCH} Latch delay. In MAX 5000 EPLDs, it is the propagation delay through the programmable register when it is configured as a flow-through latch.

t_{FD} Feedback delay. In Classic EPLDs, it is the delay of a macrocell output fed back into the logic array. In single-LAB MAX 5000 and STG EPLDs, it is the delay of a macrocell output fed back into the logic array. In multi-LAB MAX 5000 EPLDs, it is the delay of a macrocell output fed back into the LAB's logic array or to a PIA input.

t_{OD} Output buffer and pad delay. The delay from the macrocell output, through the tri-state output buffer, to the output pin.

t_{XZ} Output buffer disable delay. The delay required for high impedance to appear at the output pin after the output buffer's Enable control is disabled.

t_{ZX} Output buffer enable delay. The delay required for the macrocell output to appear at the output pin after the output buffer's Enable control is enabled.

t_{LPA} Low-power adder. In MAX 7000 EPLDs, it is the delay associated with macrocells in low-power operation. In low-power mode, t_{LPA} must be added to the logic array delay (t_{LAD}), register control delay (t_{LAC}, t_{IC}, or t_{EN}), and the shared-expander delay (t_{SEXP}) paths.

AC Timing Characteristics

The AC timing characteristics called macroparameters represent actual pin-to-pin timing characteristics. Each macroparameter consists of a combination of internal delay elements (microparameters). The data sheet for each EPLD gives timing macroparameters that characterize the AC operating specifications. These are worst-case values, derived from extensive performance measurements and guaranteed by testing. The following list defines macroparameters for Classic, MAX 5000, MAX 7000, and STG EPLDs.

t_{PD1} Dedicated input pin to non-registered output delay. The time required for a signal on any dedicated input pin to propagate through the combinatorial logic in a macrocell and appear at an external EPLD output pin.

t_{PD2} I/O pin input to non-registered output delay. The time required for a signal on any I/O pin input to propagate through the combinatorial logic in a macrocell and appear at an external EPLD output pin.

t_{PZX} Tri-state to active output delay. The time required for an input transition to change an external output from a tri-state (high-impedance) logic level to a valid high or low logic level.

t_{PXZ} Active output to tri-state delay. The time required for an input transition to change an external output from a valid high or low logic level to a tri-state (high-impedance) logic level.

t_{CLR} Time to clear register delay. The time required for a low signal to appear at the external output, measured from the input transition.

t_{SU} Global Clock setup time. The time data must be present at the input pin before the global (synchronous) Clock signal is asserted at the Clock pin.

t_{H} Global Clock hold time. The time the data must be present at the input pin after the global Clock signal is asserted at the Clock pin.

t_{CO1} Global Clock to output delay. The time required to obtain a valid output after the global Clock is asserted at the Clock pin.

t_{CNT} Minimum global Clock period. The minimum period maintained by a globally clocked counter.

t_{ASU} Array Clock setup time. The time data must be present at the input pin before an array (asynchronous) Clock signal is asserted at an input pin.

t_{AH} Array Clock hold time. The time data must be present at the input pin after an array Clock signal is asserted at an input pin.

t_{ACO1} Array Clock to output delay. The time required to obtain a valid output after an array Clock signal is asserted at an input pin.

t_{ACNT} Minimum array Clock period. The minimum period maintained by a counter when it is clocked by a signal from the array.

EPLD Timing Models

Timing models are simplified block diagrams that illustrate propagation delays through Altera EPLDs. Logic can be implemented in different paths. You can trace the actual paths used by examining the equations listed in the MAX+PLUS II Report File (.RPT) for the project. You then add

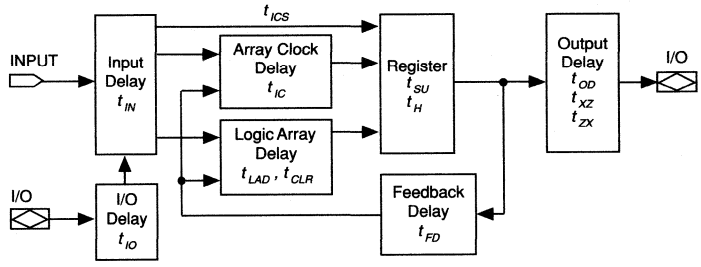
the appropriate microparameters to calculate the propagation delays through the EPLD.

Classic EPLDs

The Classic architecture provides registered and combinatorial operation. Registers can be clocked from a global Clock or through an array (product-term) Clock, and can be asynchronously cleared. When the global Clock is used, the Output Enable can be controlled by a product term. Figure 1 shows the timing model for Classic devices.

Figure 1. Classic EPLD Timing Model

If the register is bypassed, the delay between the logic array and the output buffer is zero.



Tables 1 through 4 show the internal delay parameters for all Classic EPLDs.

Table 1. EP330 Timing Parameters (in ns)

Parameter	EP330-12	EP330-15
t_{IN}	3	4
t_{IO}	1	1
t_{LAD}	6	7
t_{OD}	3	4
t_{ZX}	3	4
t_{XZ}	3	4
t_{SU}	3	4
t_{H}	0	0
t_{IC}	n/a	n/a
t_{ICS}	2	3
t_{FD}	1	1
t_{CLR}	n/a	n/a

Table 2. EP610 Timing Parameters (in ns)

Parameter	EP610-15	EP610-20	EP610-25	EP610-30	EP610-35
t_{IN}	4	5	5	6	7
t_{IO}	2	2	2	2	2
t_{LAD}	6	9	14	17	19
t_{OD}	5	6	6	7	9
t_{ZX}	5	6	6	7	9
t_{XZ}	5	6	6	7	9
t_{SU}	6	8	8	8	8
t_{H}	6	8	12	12	12
t_{IC}	6	9	14	17	19
t_{ICS}	2	3	4	4	4
t_{FD}	1	1	3	5	8
t_{CLR}	6	9	16	17	21

Table 3. EP910 Timing Parameters (in ns)

Parameter	EP910-30	EP910-35	EP910-40
t_{IN}	7	8	8
t_{IO}	3	3	3
t_{LAD}	16	19	22
t_{OD}	7	9	10
t_{ZX}	7	9	10
t_{XZ}	7	9	10
t_{SU}	10	10	10
t_{H}	15	15	15
t_{IC}	16	19	22
t_{ICS}	4	5	6
t_{FD}	4	6	8
t_{CLR}	19	22	25

Table 4. EP1810 Timing Parameters (in ns)

Parameter	EP1810-20	EP1810-25	EP1810-35	EP1810-40	EP1810-45
t_{IN}	5	7	7	7	7
t_{IO}	2	3	5	5	5
t_{LAD}	9	12	19	23	27
t_{OD}	6	6	9	10	11
t_{ZX}	6	6	9	10	11
t_{XZ}	6	6	9	10	11
t_{SU}	8	10	10	11	11
t_{H}	8	10	15	17	18
t_{IC}	9	12	19	23	27
t_{ICS}	4	5	4	6	8
t_{FD}	3	3	6	6	7
t_{CLR}	9	12	24	28	32

MAX 5000 EPLDs

The MAX 5000 architecture supports many functions. The macrocell array provides registered, combinatorial, or flow-through latch operation. The registers can be clocked from a global Clock line or through product-term arrays, and can be asynchronously preset and cleared. Separate product terms control the Output Enable and logic inversion. The array of shared expander product terms provides additional product terms to implement complex logic.

MAX 5000 EPLDs are divided into two categories: single- and multi-LAB EPLDs. Figure 2 shows the timing model for the single-LAB EPM5016 and EPM5032 EPLDs.

Figure 2. Single-LAB MAX 5000 EPLD Timing Model

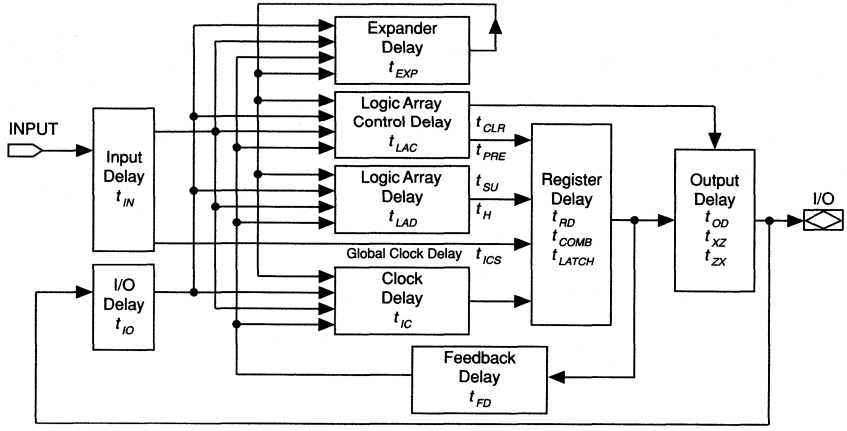
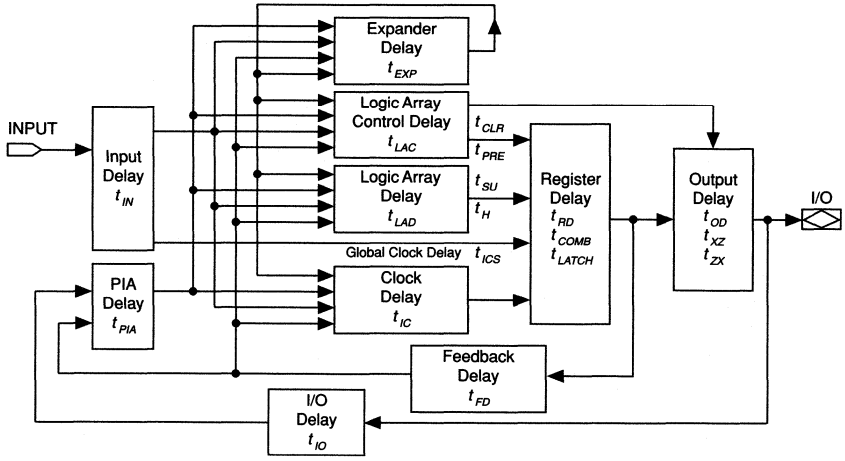


Figure 3 shows the timing model for the multi-LAB MAX 5000 EPLDs: the EPM5064, EPM5128, EPM5130, and EPM5192 EPLDs. In multi-LAB devices, the Programmable Interconnect Array (PIA) routes signals between different LABs. All I/O inputs come into the logic array through the PIA. Signals routed through the PIA incur an additional delay.

Figure 3. Multi-LAB MAX 5000 EPLD Timing Model

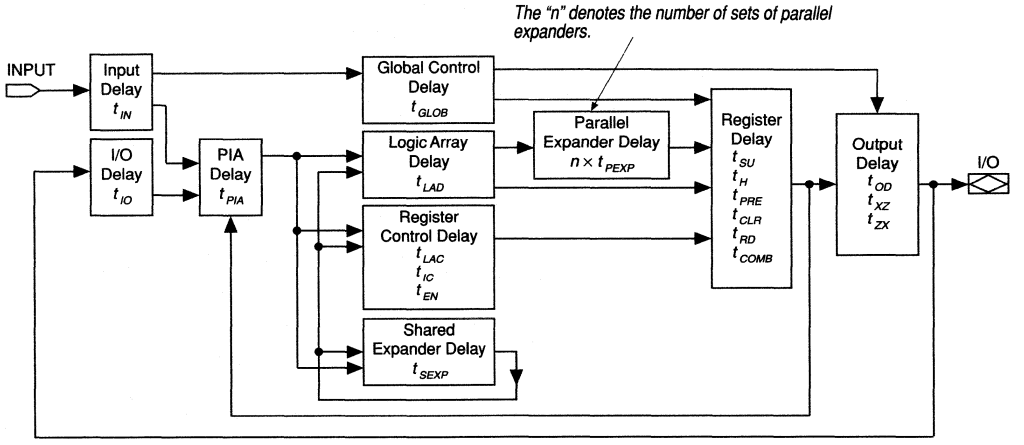


MAX 7000 EPLDs

The MAX 7000 architecture differs from the MAX 5000 architecture in several ways. This new architecture has globally routed signals for register

Clock and Clear and tri-state buffer Output Enable. It has two types of expander product terms, shared and parallel, that can be used to implement complex logic. Each macrocell can be set for low-power operation to reduce power dissipation in the EPLD. Figure 4 shows the timing model for MAX 7000 EPLDs.

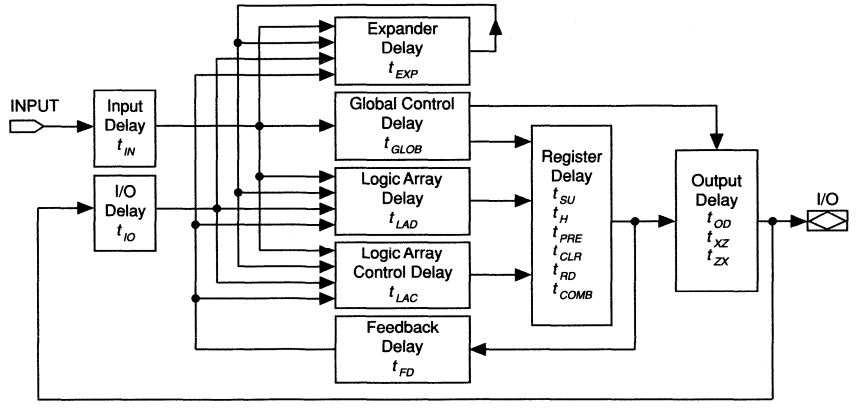
Figure 4. MAX 7000 EPLD Timing Model



STG EPLDs

The EPS464 STG EPLD is architecturally similar to MAX 7000 devices, but lacks the PIA and parallel expanders. The timing model for the EPS464 EPLD is shown in Figure 5.

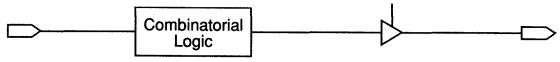
Figure 5. EPS464 STG EPLD Timing Model



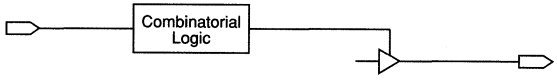
Calculating Timing Delays

You can calculate pin-to-pin timing delays for any device with the correct timing model and internal delay parameters. Each AC timing macroparameter is represented by a combination of internal delays. Figure 6 illustrates the various macroparameters. To calculate the delay for a signal that follows a different path through the EPLD, refer to the timing models shown in Figures 2 to 5 to determine which microparameters to add together.

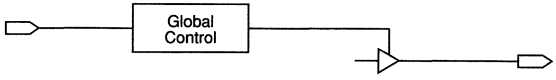
Figure 6. AC Timing Parameters (Part 1 of 3)



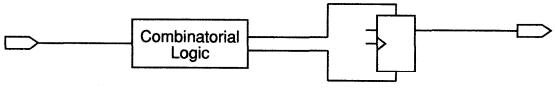
Classic	t_{PD1}	=	$t_{IN} + t_{LAD} + t_{OD}$
	t_{PD2}	=	$t_{IO} + t_{IN} + t_{LAD} + t_{OD}$
MAX 5000	t_{PD1}	=	$t_{IN} + t_{LAD} + t_{COMB} + t_{OD}$
(single-LAB)	t_{PD2}	=	$t_{IO} + t_{LAD} + t_{COMB} + t_{OD}$
MAX 5000	t_{PD1}	=	$t_{IN} + t_{LAD} + t_{COMB} + t_{OD}$
(multi-LAB)	t_{PD2}	=	$t_{IO} + t_{PIA} + t_{LAD} + t_{COMB} + t_{OD}$
MAX 7000	t_{PD1}	=	$t_{IN} + t_{PIA} + t_{LAD} + t_{COMB} + t_{OD}$
	t_{PD2}	=	$t_{IO} + t_{PIA} + t_{LAD} + t_{COMB} + t_{OD}$
STG (EPS464)	t_{PD1}	=	$t_{IN} + t_{LAD} + t_{COMB} + t_{OD}$
	t_{PD2}	=	$t_{IO} + t_{LAD} + t_{COMB} + t_{OD}$



Classic	t_{PXZ}, t_{PZX}	=	$t_{IN} + t_{LAD} + (t_{XZ}$ or $t_{ZX})$
MAX 5000	t_{PXZ}, t_{PZX}	=	$t_{IN} + t_{LAC} + (t_{XZ}$ or $t_{ZX})$
STG (EPS464)	t_{PXZ}, t_{PZX}	=	$t_{IN} + t_{LAC} + (t_{XZ}$ or $t_{ZX})$

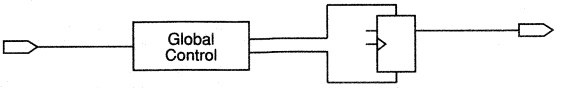


MAX 7000	t_{PXZ}, t_{PZX}	=	$t_{IN} + t_{GLOB} + (t_{XZ}$ or $t_{ZX})$
STG (EPS464)	t_{PXZ}, t_{PZX}	=	$t_{IN} + t_{GLOB} + (t_{XZ}$ or $t_{ZX})$

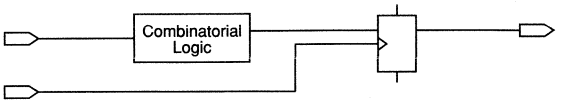


Classic	t_{CLR}	=	$t_{IN} + t_{CLR} + t_{OD}$
MAX 5000	t_{PRE}, t_{CLR}	=	$t_{IN} + t_{LAC} + (t_{PRE}$ or $t_{CLR}) + t_{OD}$
MAX 7000	t_{PRE}, t_{CLR}	=	$t_{IN} + t_{PIA} + t_{LAC} + (t_{PRE}$ or $t_{CLR}) + t_{OD}$
STG (EPS464)	t_{PRE}, t_{CLR}	=	$t_{IN} + t_{LAC} + (t_{PRE}$ or $t_{CLR}) + t_{OD}$

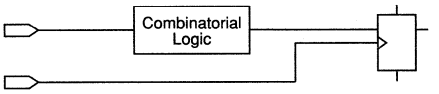
Figure 6. AC Timing Parameters (Part 2 of 3)



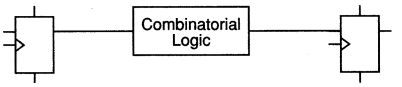
MAX 7000	t_{GCLR}	=	$t_{IN} + t_{GLOB} + t_{CLR} + t_{OD}$
STG (EPS464)	t_{GPRE}, t_{GCLR}	=	$t_{IN} + t_{GLOB} + (t_{PRE}$ or $t_{CLR}) + t_{OD}$



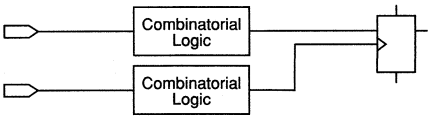
Classic	t_{SU}	=	$(t_{IN} + t_{LAD}) - (t_{IN} + t_{ICS}) + t_{SU}$
MAX 5000	t_{SU}	=	$(t_{IN} + t_{LAD}) - (t_{IN} + t_{ICS}) + t_{SU}$
MAX 7000	t_{SU}	=	$(t_{IN} + t_{PIA} + t_{LAD}) - (t_{IN} + t_{GLOB}) + t_{SU}$
STG (EPS464)	t_{SU}	=	$(t_{IN} + t_{LAD}) - (t_{IN} + t_{GLOB}) + t_{SU}$



Classic	t_{H}	=	$(t_{IN} + t_{ICS}) - (t_{IN} + t_{LAD}) + t_{H}$
MAX 5000	t_{H}	=	$(t_{IN} + t_{ICS}) - (t_{IN} + t_{LAD}) + t_{H}$
MAX 7000	t_{H}	=	$(t_{IN} + t_{GLOB}) - (t_{IN} + t_{PIA} + t_{LAD}) + t_{H}$
STG (EPS464)	t_{H}	=	$(t_{IN} + t_{GLOB}) - (t_{IN} + t_{LAD}) + t_{H}$

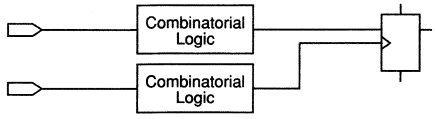


Classic	t_{CNT}	=	$t_{FD} + t_{LAD} + t_{SU}$
MAX 5000	t_{CNT}	=	$t_{RD} + t_{FD} + t_{LAD} + t_{SU}$
MAX 7000	t_{CNT}	=	$t_{RD} + t_{PIA} + t_{LAD} + t_{SU}$
STG (EPS464)	t_{CNT}	=	$t_{RD} + t_{FD} + t_{LAD} + t_{SU}$

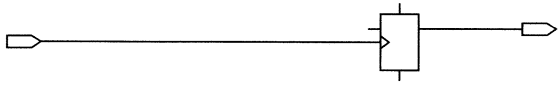


Classic	t_{ASU}	=	$(t_{IN} + t_{LAD}) - (t_{IN} + t_{IC}) + t_{SU}$
MAX 5000	t_{ASU}	=	$(t_{IN} + t_{LAD}) - (t_{IN} + t_{IC}) + t_{SU}$
MAX 7000	t_{ASU}	=	$(t_{IN} + t_{PIA} + t_{LAD}) - (t_{IN} + t_{PIA} + t_{IC}) + t_{SU}$
STG (EPS464)	t_{ASU}	=	$(t_{IN} + t_{LAD}) - (t_{IN} + t_{IC}) + t_{SU}$

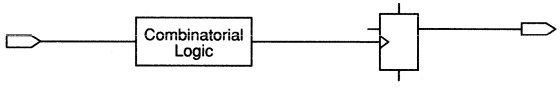
Figure 6. AC Timing Parameters (Part 3 of 3)



Classic	t_{AH}	=	$(t_{IN} + t_{IC}) - (t_{IN} + t_{LAD}) + t_H$
MAX 5000	t_{AH}	=	$(t_{IN} + t_{IC}) - (t_{IN} + t_{LAD}) + t_H$
MAX 7000	t_{AH}	=	$(t_{IN} + t_{PIA} + t_{IC}) - (t_{IN} + t_{PIA} + t_{LAD}) + t_H$
STG (EPS464)	t_{AH}	=	$(t_{IN} + t_{IC}) - (t_{IN} + t_{LAD}) + t_H$



Classic	t_{CO1}	=	$t_{IN} + t_{ICS} + t_{OD}$
MAX 5000	t_{CO1}	=	$t_{IN} + t_{ICS} + t_{RD} + t_{OD}$
MAX 7000	t_{CO1}	=	$t_{IN} + t_{GLOB} + t_{RD} + t_{OD}$
STG (EPS464)	t_{CO1}	=	$t_{IN} + t_{GLOB} + t_{RD} + t_{OD}$



Classic	t_{ACO1}	=	$t_{IN} + t_{IC} + t_{OD}$
MAX 5000	t_{ACO1}	=	$t_{IN} + t_{IC} + t_{RD} + t_{OD}$
MAX 7000	t_{ACO1}	=	$t_{IN} + t_{PIA} + t_{IC} + t_{RD} + t_{OD}$
STG (EPS464)	t_{ACO1}	=	$t_{IN} + t_{IC} + t_{RD} + t_{OD}$

Examples

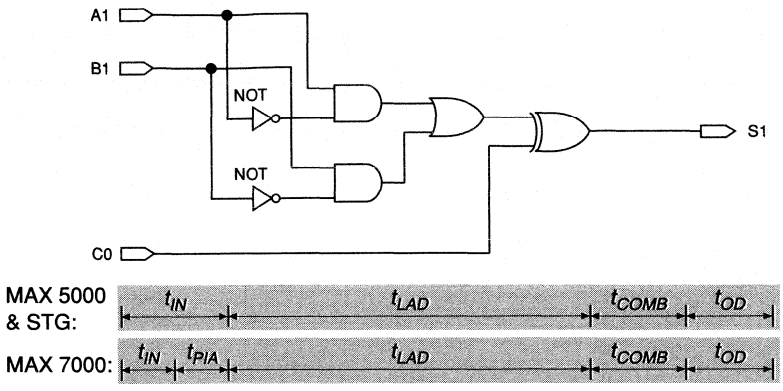
The following examples show how to use microparameters to calculate the delays for real applications.

Example 1: 7483 TTL Macrofunction

You can analyze the timing delays for macrofunctions that have been subjected to minimization and logic synthesis. A MAX+PLUS II Report File that includes the optional Equations Section lists the synthesized logic equations. These equations are structured so that you can quickly determine the logic configuration. For example, Figure 7 shows part of a `7483` TTL macrofunction (a 4-bit full adder). The Report File gives the following equations for `S1`, the least significant bit of the adder:

```
S1      = OUTPUT ( _MC021 , VCC );
_MC021  = MCELL  ( _EQ026 $ C0  );
_EQ026  = B1  & !A1
        # !B1 &  A1;
```

Figure 7. Adder Logic Timing for MAX 5000, MAX 7000, and STG Architecture



`S1` is the output of macrocell 21 (`_MC021`), which contains combinatorial logic. The combinatorial logic `MCELL ( _EQ026 $ C0 )` represents the `XOR` of the intermediate equation `_EQ026` and the carry-in `C0`. In turn, `_EQ026` is logically equivalent to the `XOR` of inputs `B1` and `A1`.

Therefore, the timing delay for `S1` is $t_{IN} + t_{LAD} + t_{COMB} + t_{OD}$ for MAX 5000 and STG EPLDs, and $t_{IN} + t_{PIA} + t_{LAD} + t_{COMB} + t_{OD}$ for MAX 7000 EPLDs.

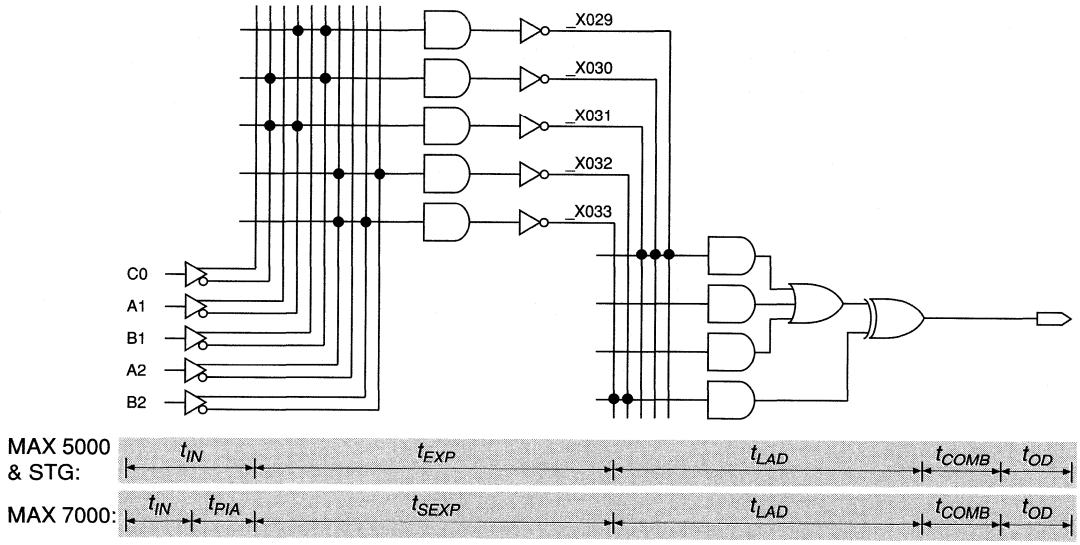
Example 2: S2 Adder Bit

For complex logic that requires expanders (represented as _X<*number*> in Report Files), the expander array delay, t_{EXP} (or t_{SEXP} for MAX 7000 EPLDs), is added to the delay element. The second bit of the `7483` adder macrofunction, `S2`, requires shared expanders. The equations are:

```
S2      =  _MC019;
_MC019  =  MCELL( _EQ023  $  _EQ024 );
_EQ023  =  _X029  &  _X030  &  _X031;
_X029   =  EXP( !B1  &  !A1 );
_X030   =  EXP( !B1  &  !C0 );
_X031   =  EXP( !A1  &  !C0 );
_EQ024  =  _X032  &  _X033;
_X032   =  EXP( !B2  &  A2  );
_X033   =  EXP(  B2  &  A2  );
```

Figure 8 shows how you can map the logic structure onto the MAX 5000 and MAX 7000 architectures with these equations. For MAX 5000 and STG EPLDs, the timing delay for `S2` is $t_{IN} + t_{EXP} + t_{LAD} + t_{COMB} + t_{OD}$; for MAX 7000 EPLDs, it is $t_{IN} + t_{PIA} + t_{SEXP} + t_{LAD} + t_{COMB} + t_{OD}$.

Figure 8. Adder Equations Mapped to MAX 5000, MAX 7000, and STG Architecture



Example 3: S2 Adder Bit With Parallel Expanders (MAX 7000)

The Compiler uses parallel expanders if the Parallel Expanders logic synthesis option is turned on when a project is compiled for MAX 7000 EPLDs. When parallel expanders are used in this case, no shareable expanders are used, and the timing delay for the `S2` bit of the `7483` becomes $t_{IN} + t_{PIA} + t_{LAD} + t_{PEXP} + t_{COMB} + t_{OD}$.

Example 4: S1 Adder Bit in Low-Power Mode (MAX 7000)

If a macrocell in a MAX 7000 EPLD is set for low-power mode, then you must add the low-power adder delay to the total delay through that macrocell. In Figure 7, the `S1` delay thus becomes $t_{IN} + t_{PIA} + t_{LPA} + t_{LAD} + t_{COMB} + t_{OD}$.

Conclusion

The architectures of Altera EPLDs have fixed internal timing delays that are independent of routing. You can determine the worst-case timing delays for any design before programming a device. Total delay paths (macroparameters) can be expressed as the sums of internal timing delays (microparameters). Timing models illustrate the internal delay paths for EPLDs and show how these microparameters affect each other. You can use MAX+PLUS II development tools to automatically calculate delay paths, or hand-calculate delay paths by adding the microparameters for an appropriate timing model. With this ability to predict worst-case timing delays, you can be confident of a design's in-system timing performance.

Electronic Bulletin Board Service

April 1992, ver. 3

Introduction

Altera provides an electronic bulletin board service (BBS) for continuous access to up-to-date EPLD and development tool information, electronic application notes and briefs, data sheet updates, customer newsletters, and useful utility programs. The BBS also supports file transfers to and from the Altera Applications Engineering Department. Owners of A+PLUS, MAX+PLUS, and MAX+PLUS II software can refer to their user guides for detailed information on using the BBS.

Modem Number:
(408) 249-1100

The telephone number for the BBS is (408) 249-1100. To connect to the BBS via modem, the following equipment and configuration are required:

- ❑ 1200 or 2400 baud rate
- ❑ Bell Standard 212A or compatible modem
- ❑ Data format: 8 data bits, 1 stop bit, no parity

The following file transfer protocols are supported:

- ❑ Xmodem (Checksum)
- ❑ Xmodem-CRC (CRC)
- ❑ Ymodem (1K-Xmodem)
- ❑ Ymodem-G (1K-Xmodem-G)
- ❑ ASCII (Non-Binary)
- ❑ Kermit

Logging On

After the BBS connection has been established, you can choose between graphic (for EGA or VGA displays) or non-graphic display mode. You are then prompted for your name; if you are a new user, you can also choose a password. Each name and password are recorded for future log-ons.

A series of screens appears automatically: the Altera News screen, the Personal Mail screen, and the Settings screen. The Main Menu, from which you can access all functions, appears next. The most commonly used functions are: **F)ile Directories**, **U)pload a File**, and **D)ownload a File**. On-line help is available with the **H)elp Functions** command. The **F)ile Directories** command displays a list of directories containing files that can be downloaded. (Uploaded files are stored in a private directory.)

File Uploading

The File Upload service is available for uploading files that require analysis or correction by an Altera Applications Engineer. All files that are uploaded to the Altera BBS are automatically stored in a private directory. When a file is uploaded, the file description should include the name of the Altera Applications Engineer who has been asked to examine the file.

File Downloading

You can copy files from the following eight directories:

1. From-Altera File Directory

This directory is a general directory for downloading files from Altera Applications, for example, after a problem or question has been analyzed.

2. Electronic Application Briefs

This directory contains Electronic Application Briefs (EABs) and Notes (EANs) with up-to-date information on using Altera EPLDs effectively.

3. Electronic Application Utilities

This directory contains Electronic Application Utilities (EAUs) that complement Altera software and aid EPLD design. Two commonly used utilities are described below. All utilities are described in *Application Brief 73 (Software Utility Programs)* in the Altera 1992 ***Data Book***.

PLD2EQN The PLD2EQN utility converts common PAL/GAL/PLA JEDEC Files to Altera Hardware Description Language (AHDL) files that are compatible with the MAX+PLUS and MAX+PLUS II software. This utility can also produce an Altera Design File (.ADF) that is compatible with A+PLUS software.

ABEL2MAX The ABEL2MAX utility converts .TT2 files generated by Data I/O's ABEL software (version 4.0 or higher) into AHDL files that are compatible with the MAX+PLUS and MAX+PLUS II software.

4. Altera Customer Newsletters

This directory contains Altera newsletters that provide current news on EPLDs and development tools, and a Question & Answer forum that answers many common questions asked by Altera customers.

5, 6 & 7. A+PLUS, MAX+PLUS & MAX+PLUS II Macrofunction Exchange Libraries

These directories are used to publicly exchange A+PLUS, MAX+PLUS, and MAX+PLUS II macrofunctions. Customers can download any macrofunctions in this directory. Macrofunctions that have been uploaded to be shared with other users are also placed here by the system operator ("Sysop").

8. Data I/O Support for Altera Devices

This directory lists the Data I/O software and hardware required to program Altera EPLDs.

Sales Offices, Distributors & Representatives

April 1992

Altera U.S. Sales Offices

NORTHERN CALIFORNIA
Altera Corporation
2610 Orchard Parkway
San Jose, CA 95134-2020
TEL: (408) 894-7900
FAX: (408) 428-0463

SOUTHERN CALIFORNIA
Altera Corporation
6 Morgan, Suite 100
Irvine, CA 92718
TEL: (714) 587-3002
FAX: (714) 587-9789

GEORGIA
Altera Corporation
1080 Holcomb Bridge Road
Bldg. 100, Suite 300
Roswell, GA 30076
TEL: (404) 594-7621
FAX: (404) 998-9830

ILLINOIS
Altera Corporation
200 W. Higgins Road
Suite 322
Schaumburg, IL 60195
TEL: (708) 310-8522
FAX: (708) 310-8589

MASSACHUSETTS
Altera Corporation
945 Concord Street
Framingham, MA 01701
TEL: (508) 626-0181
FAX: (508) 879-0698

NEW JERSEY
Altera Corporation
981 U.S. Highway 22
Suite 2000
Bridgewater, NJ 08807
TEL: (201) 526-9400
FAX: (201) 526-5471

TEXAS
Altera Corporation
Signature Place
14785 Preston Road
Suite 550
Dallas, TX 75240
TEL: (214) 233-1491
FAX: (214) 233-1493

Altera International Sales Offices

UNITED STATES
(CORPORATE HEADQUARTERS)
Altera Corporation
2610 Orchard Parkway
San Jose, CA 95134-2020
USA
TEL: (408) 894-7000
TLX: 888496
FAX: (408) 248-7097

BELGIUM
(EUROPEAN HEADQUARTERS)
Altera Europe
25 Avenue de Beaulieu
B-1160 Bruxelles
Belgium
TEL: (32) 2-660 20 77
TLX: (886) 27087901(AVVALB)
FAX: (32) 2-660 52 25

FRANCE
Altera France
Zac La Sabliere
4, Rue Maryse Bastie
91430-IGNY
France
TEL: (33) 1 69 85 5630
FAX: (33) 1 69 85 5614

GERMANY
Altera GmbH
Bahnhofstraße 9
D-8057 Eching
Germany
TEL: (49) 89/3196014
TLX: (841) 5213250
FAX: (49) 89/3192193

Altera International Sales Offices

(continued)

ITALY
Altera Italia
Corso Lombardia 75
Autoporto Pescarito
San Mauro, Torinese
10099 Torino
Italy
TEL: (39) 11 223 8588
FAX: (39) 11 223 8589

JAPAN
Altera Japan K.K.
Ichikawa Gakugeidai Building
2nd Floor
12-8 Takaban 3-chome
Meguro-ku, Tokyo 152
Japan
TEL: (81) 33 716-2241
FAX: (81) 33 716-7924

UNITED KINGDOM
Altera UK
5 Tower Court
Horns Lane
Princes Risborough, Bucks HP17 OAJ
England
TEL: (44) 844 275-285
TLX: (851) 94016389 (TIME G)
FAX: (44) 844 275-599

North American Distributors

Alliance Electronics

Almac/Arrow

Arrow/Schweber Electronics Group

Future Electronics (Canada only)

Newark Electronics

Pioneer-Standard Electronics

Pioneer Technologies Group

Semad (Canada only)

Wyle Laboratories

U.S. Sales Representatives

ALABAMA
Montgomery Marketing, Inc.
1910 Sparkman Drive
Huntsville, AL 35816
(205) 830-0498

ARIZONA
Oasis Sales, Inc.
301 E. Bethany Home Road #A135
Phoenix, AZ 85012
TEL: (602) 277-2714
FAX: (602) 263-9352

ARKANSAS
Technical Marketing, Inc.
3320 Wiley Post Road
Carrollton, TX 75006
(214) 387-3601

CALIFORNIA
Altera Corporation
2610 Orchard Parkway
San Jose, CA 95134-2020
TEL: (408) 894-7900
FAX; (408) 428-0463

Premier Technical Sales
23566 Woodhaven Place
Auburn, CA 95603
TEL: (916) 268-3357
FAX: (916) 268-0192

CALIFORNIA *(continued)*
QuadRep Southern, Inc.
28720 Roadside Drive, Suite 227
Agoura, CA 91301
(818) 597-0222

QuadRep Southern, Inc.
4 Jenner Street, Suite 120
Irvine, CA 92718
(714) 727-4222

QuadRep Southern, Inc.
7585 Ronson Road, Suite 100
San Diego, CA 92111
(619) 560-8330

COLORADO
Lange Sales, Inc.
1500 W. Canal Court, Bldg. A, Suite 100
Littleton, CO 80120
(303) 795-3600

CONNECTICUT
Technology Sales, Inc.
237 Hall Avenue
Wallingford, CT 06492
(203) 269-8853

DELAWARE
BGR Associates
Evesham Commons
525 Route 73, Suite 100
Marlton, NJ 08053
(609) 983-1020

DISTRICT OF COLUMBIA
Robert Electronic Sales
5525 Twin Knolls Road, Suite 325
Columbia, MD 21045
(301) 995-1900

FLORIDA
EIR, Inc.
1057 Maitland Center Commons
Maitland, FL 32751
(407) 660-9600

GEORGIA
Montgomery Marketing, Inc.
3000 Northwoods Pkwy., Suite 110
Norcross, GA 30071
(404) 447-6124

IDAHO
Lange Sales, Inc.
5440 Franklin Street, Suite 110
Boise, ID 83705
(208) 345-8207

Westerberg & Associates, Inc.
12505 NE Bel-Red Road, Suite 112
Bellevue, WA 98005
(206) 453-8881

ILLINOIS
AEM, Inc.
11520 St. Charles Rock Road, Suite 131
Bridgeton, MO 63044
(314) 298-9900

Oasis Sales Corporation
1101 Tonne Road
Elk Grove Village, IL 60007
(708) 640-1850

INDIANA
Electro Reps, Inc.
7240 Shadeland Station, Suite 275
Indianapolis, IN 46256
(317) 842-7202

IOWA
AEM, Inc.
4001 Shady Oak Drive
Marion, IA 52302
(319) 377-1129

KANSAS
AEM, Inc.
8859 Long Street
Lenexa, KS 66215
(913) 888-0022

KENTUCKY
Electro Reps, Inc.
7240 Shadeland Station, Suite 275
Indianapolis, IN 46256
(317) 842-7202

LOUISIANA
Technical Marketing, Inc.
2901 Wilcrest Drive, Suite 139
Houston, TX 77042
(713) 783-4497

MAINE
Technology Sales, Inc.
332 Second Avenue
Waltham, MA 02154
(617) 890-5700

MARYLAND
Robert Electronic Sales
5525 Twin Knolls Road, Suite 325
Columbia, MD 21045
(301) 995-1900

MASSACHUSETTS
Technology Sales, Inc.
332 Second Avenue
Waltham, MA 02154
(617) 890-5700

MICHIGAN
Rathsburg Associates, Inc.
34605 Twelve Mile Road
Farmington Hills, MI 48331
(313) 489-1500

MINNESOTA
Cahill, Schmitz & Cahill, Inc.
315 N. Pierce
St. Paul, MN 55104
(612) 646-7217

MISSISSIPPI
Montgomery Marketing, Inc.
3000 Northwoods Parkway, Suite 110
Norcross, GA 30071
(404) 447-6124

MISSOURI
AEM, Inc.
11520 St. Charles Rock Road, Suite 131
Bridgeton, MO 63044
(314) 298-9900

MONTANA
Lange Sales, Inc.
1500 W. Canal Court, Bldg. A, Suite 100
Littleton, CO 80120
(303) 795-3600

NEBRASKA
AEM, Inc.
4001 Shady Oak Drive
Marion, IA 52302
(319) 377-1129

U.S. Sales Representatives

(continued)

NEVADA
Premier Technical Sales
23566 Woodhaven Place
Auburn, CA 95603
TEL: (916) 268-3357
FAX: (916) 268-0192

NEW HAMPSHIRE
Technology Sales, Inc.
332 Second Avenue
Waltham, MA 02154
(617) 890-5700

NEW JERSEY
BGR Associates
Evesham Commons
525 Route 73, Suite100
Marlton, NJ 08053
(609) 983-1020

ERA, Inc.
354 Veterans Memorial Highway
Commack, NY 11725
(516) 543-0510

NEW MEXICO
Nelco Electronix
3240 C Juan Tabo Blvd. NE
Albuquerque, NM 87111
(505) 293-1399

NEW YORK
ERA, Inc. (New York Metro)
354 Veterans Memorial Highway
Commack, NY 11725
(516) 543-0510

Technology Sales, Inc.
470 Perinton Hills Office Park
Fairport, NY 14450
(716) 223-7500

Technology Sales, Inc.
903 Hanshaw Road
Ithaca, NY 14850
(607) 257-7070

NORTH CAROLINA
Montgomery Marketing, Inc.
P.O. Box 520
Cary, NC 27512
(919) 467-6319

Montgomery Marketing, Inc.
1200 Trinity Road
Raleigh, NC 27607
(919) 851-0010

NORTH DAKOTA
Cahill, Schmitz & Cahill, Inc.
315 N. Pierce
St. Paul, MN 55104
(612) 646-7217

OHIO
The Lyons Corporation
4812 Frederick Road, Suite 101
Dayton, OH 45414
(513) 278-0714

The Lyons Corporation
4615 W. Streetsboro Road
Richfield, OH 44286
(216) 659-9224

OHIO
The Lyons Corporation
248 N. State Street
Westerville, OH 43081
(614) 895-1447

OKLAHOMA
Technical Marketing, Inc.
3320 Wiley Post Road
Carrollton, TX 75006
(214) 387-3601

OREGON
Westerberg & Associates, Inc.
7165 SW Fir Loop
Portland, OR 97223
(503) 620-1931

PENNSYLVANIA
BGR Associates
Evesham Commons
525 Route 73, Suite 100
Marlton, NJ 08053
(609) 983-1020

The Lyons Corporation
4812 Frederick Rd., Suite 101
Dayton, OH 45414
(513) 278-0714

PUERTO RICO
Technology Sales, Inc.
Edificio Rali 219
San German, PR 00753
(809) 892-4745

RHODE ISLAND
Technology Sales, Inc.
332 Second Avenue
Waltham, MA 02154
(617) 890-5700

SOUTH CAROLINA
Montgomery Marketing, Inc.
1200 Trinity Road
Raleigh, NC 27607
(919) 851-0010

SOUTH DAKOTA
Cahill, Schmitz & Cahill, Inc.
315 N. Pierce
St. Paul, MN 55104
(612) 646-7217

TENNESSEE
Montgomery Marketing, Inc.
1910 Sparkman Drive
Huntsville, AL 35816
(205) 830-0498

TEXAS
Technical Marketing, Inc.
3320 Wiley Post Road
Carrollton, TX 75006
(214) 387-3601

Technical Marketing, Inc.
2901 Wilcrest Drive, Suite 139
Houston, TX 77042
(713) 783-4497

Technical Marketing, Inc.
1315 Sam Bass Circle, Suite B-3
Round Rock, TX 78681
(512) 244-2291

UTAH
Lange Sales, Inc.
1864 S. State, Suite 295
Salt Lake City, UT 84115
(801) 487-0843

VERMONT
Technology Sales, Inc.
332 Second Avenue
Waltham, MA 02154
(617) 890-5700

VIRGINIA
Robert Electronic Sales
5525 Twin Knolls Road, Suite 325
Columbia, MD 21045
(301) 995-1900

WASHINGTON
Westerberg & Associates, Inc.
12505 NE Bel-Red Road, Suite 112
Bellevue, WA 98005
(206) 453-8881

WEST VIRGINIA
Robert Electronic Sales
5525 Twin Knolls Road, Suite 325
Columbia, MD 21045
(301) 995-1900

WISCONSIN
Cahill, Schmitz & Cahill, Inc.
315 N. Pierce
St. Paul, MN 55104
(612) 646-7217

Oasis Sales Corporation
1305 N. Barker Road
Brookfield, WI 53005
(414) 782-6660

WYOMING
Lange Sales, Inc.
1500 W. Canal Court, Bldg. A, Suite 100
Littleton, CO 80120
(303) 795-3600

Canadian Sales Representatives

ALBERTA
Kaytronics
6815-8 Street NE, Suite 179
Calgary, Alberta T2E 7H7
Canada
(403) 275-7000

BRITISH COLUMBIA
Kaytronics
#102-4585 Canada Way
Burnaby, BC V5G 4L6
Canada
(604) 294-2000

QUEBEC
Kaytronics
5800 Thimens Boulevard
Ville St. Laurent, Quebec H4S 1S5
Canada
(514) 745-5800

ONTARIO
Kaytronics
331 Bowes Road, Unit 1
Concord, Ontario L4K 1J2
Canada
(416) 669-2262

Kaytronics
300 March Road, Suite 603
Kanata, Ontario K2K 2E2
Canada
(613) 564-0087

International Distributors

ARGENTINA
YEL S.R.L.
Virrey Cevallos 143
1077 Buenos Aires
Argentina
TEL: (54) 1-45-7140/7163
TLX: (390) 18605 (YEL AR)
FAX: (54) 1-440-1533

AUSTRALIA
Veltek Pty. Ltd.
18 Harker St.
Burwood, Victoria 3125
Australia
TEL: (61) 3-808-7511
FAX: (61) 3-808-5473

International Distributors

(continued)

AUSTRIA
Hitronik
St.-Veit-Gasse 51
1130 Wien
Austria
TEL: (43) 222-824-199
TLX: (847) 134404 (HIT)
FAX: (43) 222-828-557-2

BELGIUM
D&D Electronics
VIIe Olympiadelaan 93
2020 Antwerpen
Belgium
TEL: (32) 3-827-7934
TLX: (846) 73121 (DDELEC BU)
FAX: (32) 3-828-7254

BRAZIL
Uniao Digital Ltda.
Rua Texas 622
Brooklin Novo 04557
São Paulo SP
Brazil
TEL: (55) 11 533-0967
FAX: (55) 11 533-6780

DENMARK
E.V. Johanssen Elektronik A/S
Titangade 15
2200 Koebenhavn N
Denmark
TEL: (45) 31 83 90 22
TLX: (855) 16522 (EVICAS DK)
FAX: (45) 31 83 92 22

FINLAND
Yleiselektroniikka Oy
P.O. Box 73
Luomannotko 6
SF-02201 Espoo
Finland
TEL: (358) 0-452-1622
TLX: (857) 123212 (YLEOY SF)
FAX: (358) 0-452-3337

FRANCE
Tekelec Airtronic SA
Cité des Bruyères
Rue Carle Vernet
92310 Sèvres
France
TEL: (33) 1 46 23 24 25
TLX: (842) 634018 (TKLEC A F)
FAX: (33) 1 45 07 21 91

GERMANY
Electronic 2000
Vertriebs-AG
Stahlgruberring 12
8000 München 82
Germany
TEL: (49) 89/42001-0
TLX: (841) 522561 (ELEC D)
FAX: (49) 89/42001-209

HONG KONG
RTI Industries Co. Ltd.
B23, 3/F Proficient Industrial Center
6 Wang Kwun Road
Kowloon
Hong Kong
TEL: (852) 795-7421
FAX: (852) 795-7839

INDIA
Sritech Information Technology (P) Ltd.
744/51, 2nd Floor, Chintal Plaza
33rd Cross, 10th Main
4th Block, Jayanagar
Bangalore 560 011
India
TEL: (91) 812-640-661
TLX: (953) 08458162 (SRIS IN)
FAX: (91) 812-643-608

ISRAEL
Vectronics Ltd.
60 Medinat Hayehudim Street
P.O. Box 2024
Herzlia B 46120
Israel
TEL: (972) 52-556-070
TLX: (922) 342579 (VECO IL)
FAX: (972) 52-556-508

Active Technologies
2200 Marcus Avenue
New Hyde Park, NY 11402
USA
TEL: (516) 488-1226
FAX: (516) 488-1245

ITALY
Inter-Rep S.p.A.
Via Orbetello 98
10148 Torino
Italy
TEL: (39) 11/29191
TLX: (843) 221422 (IR TO I)
FAX: (39) 11/2165915

Lasi Elettronica S.p.A.
Viale Fulvio Testi, 208
29126 Milano
Italy
TEL: (39) 2 66101370

JAPAN
Altima Corporation
Hakusan High-Tech Park
1-22-2 Hakusan, Midori-Ku
Yokohama City 226
Japan
TEL: (81) 45-939-6113
FAX: (81) 45-939-6114

Paltek Corporation
5-2-12 Kamiyoga
Setagaya-Ku
Tokyo 158
Japan
TEL: (81) 33-707-5455
TLX: (781) 02425205 (PALTEK J)
FAX: (81) 33-707-5338

KOREA
MJL Korea, Ltd.
Samwhan Camus Bldg.
17-3 Youido Dong
Yeungdeungpo Ku
Seoul, 150-010
Korea
TEL: (82) 2-784-8000
TLX: 843457 (MJL MORV)
FAX: (82) 2-784-4644

NETHERLANDS
Koning en Hartman
1 Energieweg
2627 AP Delft
Netherlands
TEL: (31) 15 609906
TLX: 38250 (KOHA NL)
FAX: (31) 15 619194

NORWAY
Eltron A/S
Aslakveien 20 F
0753 Oslo 7
Norway
TEL: (47) 2-500650
TLX: (856) 77144 (ELTRO N)
FAX: (47) 2-502777

SINGAPORE
Serial System Marketing
21 Moonstone Lane
Poh Leng Building #02-01
Singapore 1232
TEL: (65) 293-8830
FAX: (65) 291-2673

SPAIN
Selco
Paseo de la Habana, 190
28036 Madrid
Spain
TEL: (34) 1-326-4213
TLX: (831) 45458 (EPAR E)
FAX: (34) 1-259-2284

SWEDEN
Nortec Electronics AB
Parkvagen, 2A
Box 1830
S-171 27 Solna
Sweden
TEL: (46) 8 705 18 00
FAX: (46) 8 83 69 18

SWITZERLAND
Eljapex AG
Hardstraße 72
5430 Wettingen
Switzerland
TEL: (41) 56 27 57 77
FAX: (41) 56 26 14 86

TAIWAN
Galaxy Far East Corp.
8F-6, 390 Sec. 1
Fu Hsing South Road
Taipei
Taiwan R.O.C.
TEL: (886) 2-705-7266
TLX: (785) 26110 (GALAXYER)
FAX: (886) 2-708-7901

Jeritron Ltd.
5F, 1182 Cheng-Teh Road
Taipei
Taiwan R.O.C.
TEL: (886) 2-882-0710
FAX: (886) 2-882-3154

THAILAND
Nu-Era Co. Ltd.
2077/8 Ramkhamhang 37
Huamark, Bangapi
Bangkok 10240
Thailand
TEL: (66) 2-318-6453
FAX: (66) 2-318-6454

UNITED KINGDOM
Ambar Cascom Ltd.
Rabans Close
Aylesbury, Bucks HP19 3RS
England
TEL: (44) 296 434141
TLX: (851) 837427 (AMBAR G)
FAX: (44) 296 29670

Thame Components Ltd.
Thame Park Road
Thame, Oxon 0X9 3UQ
England
TEL: (44) 844 261188
TLX: (851) 837917 (MEMEC G)
FAX: (44) 844 261681

Abbreviations

April 1992

ADF	Altera Design File
AHDL	Altera Hardware Description Language
ASCII	American Standard Code for Information Interchange
ASIC	Applications specific integrated circuit
ASM	SAM Assembly Language
ASMILE	Altera State Machine Input Language
ATVG	Automatic test vector generation
BBS	Electronic bulletin board service
BCD	Binary-coded decimal
BNF	Backus-Naur form
CAE	Computer-aided engineering
CAS	Column-address strobe
CCD	Charge-coupled device
CerDIP	Ceramic dual in-line package
CMD	Command File
CNF	Compiler Netlist File
CPU	Central processing unit
CRC	Cyclic redundancy check
CRT	Cathode-ray tube
DFT	Design-for-testability
DIP	Dual in-line package
DMA	Direct Memory Access
DRAM	Dynamic random access memory
DSP	Digital signal processing
EAB	Electronic Application Brief
EAN	Electronic Application Note
EAU	Electronic Application Utility
EBC	EISA bus controller
EDAC	Error detection and correction
EDC	EDIF Command File
EDF	EDIF Input File
EDIF	Electronic Design Interchange Format
EDO	EDIF Output File
EEPROM	Electrically erasable programmable read-only memory
EISA	Extended Industry Standard Architecture
EPLD	Erasable Programmable Logic Device
EPROM	Erasable programmable read-only memory
FIFO	First in/first out
FIN	Fitter Input File
FIR	Finite-duration impulse response

FIT	Fit File
FPGA	Field-programmable gate array
FTP	File transfer protocol
GAL	Gate array logic
GDF	Graphic Design File
HIF	Hierarchy Interconnect File
HST	History File
IC	Integrated circuit
INC	Include File
INI	Initialization file (MAX+PLUS2.INI or <project *name*>.INI)
ISA	Industry Standard Architecture
JED	JEDEC File
JLCC	J-lead chip carrier
LAB	Logic Array Block
LCA	Logic cell array
LED	Light-emitting diode
LEF	Logic Equation File
LMF	Library Mapping File
LOG	Log File
LSB	Least significant bit
MAC	Multiplier-accumulator
MAX	Multiple Array MatriX
MC	Macrocell
MCA	Micro Channel Architecture
MMF	MAX+PLUS II Message File
MPLD	Mask-Programmed Logic Device
MPU	Master Programming Unit
MSB	Most significant bit
MSI	Medium-scale integration
MTF	Message Text File
NRE	Non-recurring engineering
NTSC	National Television System Committee
OCR	Optical character recognition
OTP	One-time programmable
PAL	Programmable array logic
PC	Personal computer
PCB	Printed circuit board
PDIP	Plastic dual in-line package
PE	Processor element
PGA	Pin-grid array
PIA	Programmable Interconnect Array
PIO	Peripheral Input/Output
PLA	Programmable logic array
PLCC	Plastic J-lead chip carrier
PLD	Programmable logic device
PLF	Programmer Log File
PLL	Phase-locked loop
POF	Programmer Object File

POR	Power-On Reset
PQFP	Plastic quad flat pack
PRB	Probe & Resource Assignment File
PROM	Programmable read-only memory
QFP	Quad flat pack
RAM	Random access memory
RAS	Row-address strobe
RC	Resistor/capacitor
ROM	Read-only memory
RPT	Report File
SALSA	Speedy Altera Logic Simplifying Algorithm
SAM	Stand-Alone Microsequencer
SCC	Serial communication control
SCF	Simulator Channel File
SIF	Simulator Initialization File
SMF	State Machine File
SNF	Simulator Netlist File
SOIC	Small-outline integrated circuit
SRAM	Static random access memory
SSG	Synchronous signal generator
SSI	Small-scale integration
STG	Synchronous Timing Generator
SYM	Symbol File
TAO	Timing Analyzer Output File
TBL	Table File
TDF	Text Design File
TTL	Transistor-to-transistor logic
VEC	Vector File
VLSI	Very large-scale integration
WDF	Waveform Design File
XMS	Extended Memory Specification
XTAL	Crystal

April 1992

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Altera Corporation
2610 Orchard Parkway
San Jose, CA 95134-2020
Telephone: (408) 894-7000

Altera Europe
25 Ave de Beaulieu
B-1160 Bruxelles
Belgium
Telephone: (32) 2-660 20 77

Altera Japan K.K.
Ichikawa Gakugeidai Bldg.
Second Floor
12-8 Takaban 3-Chome
Meguro-Ku, Tokyo 152
Telephone: 03 (3716)/2241