APPLICATION NOTE

AN704

An upward migration path for the 80C51: the Philips XA architecture

Author: Greg Goodhue

1995 Apr 27

Philips
Semiconductors

PHILIPS

An upward migration path for the 80C51: the Philips XA architecture

AN704

Author: Greg Goodhue

The 80C51 is arguably the most used 8-bit microcontroller architecture in the world, and a vast amount of public and private code exists for this processor. The “XA” (Extended Architecture) microcontroller, developed by Philips Semiconductors, is a high performance 16-bit processor that retains source code compatibility with the original 80C51. By permitting simple translation of source code, the XA allows existing 80C51 code to be re-used with a higher performance 16-bit controller. This provides an upward mobility path to a 16-bit controller for 80C51 users that has not previously existed, while also bringing a low cost, high performance, general purpose 16-bit controller to the market. How can a modern 16-bit controller provide compatibility with the venerable 80C51 without badly compromising the architecture and performance?

DESIGN TRADEOFFS

Many tradeoffs must be made and considerations taken into account when creating an upward compatible processor that must also be high performance and low cost. Among the areas to be considered are the processor’s memory map and means of accessing memory, instruction set and methods of instruction execution, stack operation, interrupts, and special features added to enhance particular functions, such as multi-tasking, exception handling, and debugging features.

The goal of source code compatibility, rather than object code compatibility, was adopted for a number of reasons. First, absolute upward compatibility with an existing processor is by definition impossible if one of the goals of the new processor is to generally improve performance. By doing the same things in less time, the time related attributes of previously written code change.

Another consideration has to do with the fact that the 80C51 used all but one of the 256 opcodes available with an 8-bit opcode field. Adding more than a few new instructions or a new data type (such as 16-bit operations) would result in a very inefficient instruction encoding, and inefficient execution as well, for those new functions.

Creating a new instruction set that includes an exact copy of the 80C51 instruction set as a subset would also be very inefficient, since some subset of many new operations would act as duplicates of 80C51 instructions. For instance, a more powerful ADD instruction that can add any byte or word register to any other register is a superset of the 80C51 instruction to add a register to the accumulator. In such a case, there is no good argument to duplicate the original instruction precisely.

An 80C51 “mode” on an otherwise totally new (and therefore incompatible) processor was also considered. However, this approach would result in having in effect 2 processors on one chip, which would be confusing and not very cost effective. Mixing new, more efficient code with existing 80C51 code would require switching modes often, which would be very cumbersome and potentially hazardous. If a mode switch was skipped by accident in some seldom executed code sequence, the processor could suddenly find itself executing code using the wrong instruction set!

HOW IS IT DONE?

The team that created the XA architecture at Philips followed several rules in order to insure that 80C51 compatibility goals were met. First, translation for all (or nearly all) 80C51 instructions would be one to one. Multi-instruction combinations that could result in problems if split by an interrupt or otherwise compromise the integrity of the translation would be avoided. This has the effect of producing a simple, straightforward, and easily checkable translation.

Second, most 80C51 instructions should be a subset of new XA instructions. If that is not possible or doesn’t make sense in a particular case, the original 80C51 instruction would be included “as-is”, even though it might not fit the basic XA architecture’s philosophy.

Third, XA register, code memory, data memory, and Special Function Register addressing would be a superset of the 80C51 equivalents. The same idea applies to other features that are part of the CPU.

Finally, some compromises to these compatibility rules are allowed in cases where keeping absolute compatibility would adversely affect system cost, high level language support, or performance. The cost (in engineering time) of dealing with any incompatibilities must be kept to a minimum. Preferably, the issue should not even be noticeable to most customers.

MEMORY MAPPING

At the root of any potential compatibility between the XA and the 80C51 is the memory map. The XA takes a simple but effective approach to this issue: its memory map is a superset of the 80C51 memory map. Modes of addressing memory likewise duplicate the modes available on the 80C51, adds new modes, and enhances some of the old ones.

In translating 80C51 source code to the XA, particular registers are used to represent the accumulator (A) and the data pointer (DPTR). Although the XA can use any of the 14 general purpose byte registers int he register file as an accumulator, the 80C51 has some features that require the accumulator to be a specific byte register. These are primarily the parity flag and a few special instructions that intrinsically reference the accumulator in a way that could not be generalized in the XA. The latter are, specifically, instructions like: JZ, JNZ, MOVC A,@A+DPTR, MOVC @A+PC, and JMP @A+DPTR. Figure 1 shows the register file of the first XA derivative (the XA architecture can support some additional registers not implemented in the first part) and the registers used for 80C51 translation.

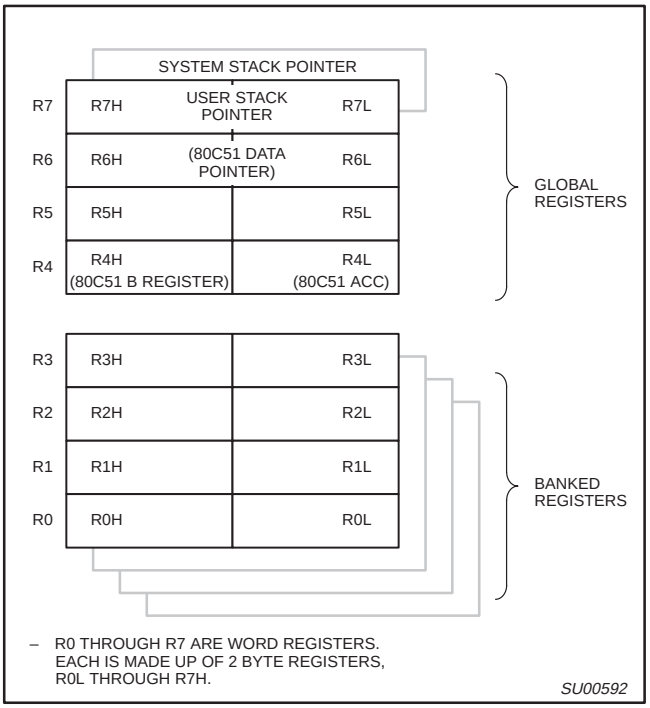

Figure 1. XA Register File

An alternate program status word (PSW) was created on the XA to duplicate the 80C51 PSW and contains the P (parity) flag as well as the F1 and F0 user defined flags that are not found in the native XA PSW. The XA PSW, on the other hand, adds some new status flags and system controls to expand its capabilities.

The XA register file duplicates the 4 banks of 8 bytes that are found in the 80C51. An 80C51 compatibility mode determines whether these locations appear both as registers and as the lower 32 bytes of data memory as they do on the 80C51. The more standard scheme of keeping the register file separate from the data memory is the default on the XA. Besides being “cleaner”, the separation of the register file from data memory allows for a higher performance implementation of the XA processor core at some point in the future if and when 80C51 compatibility is no longer required. Figure 2 shows the overlap of data memory and the register file in compatibility mode. This shows only this one aspect of the XA memory map, not a general view of the memory.

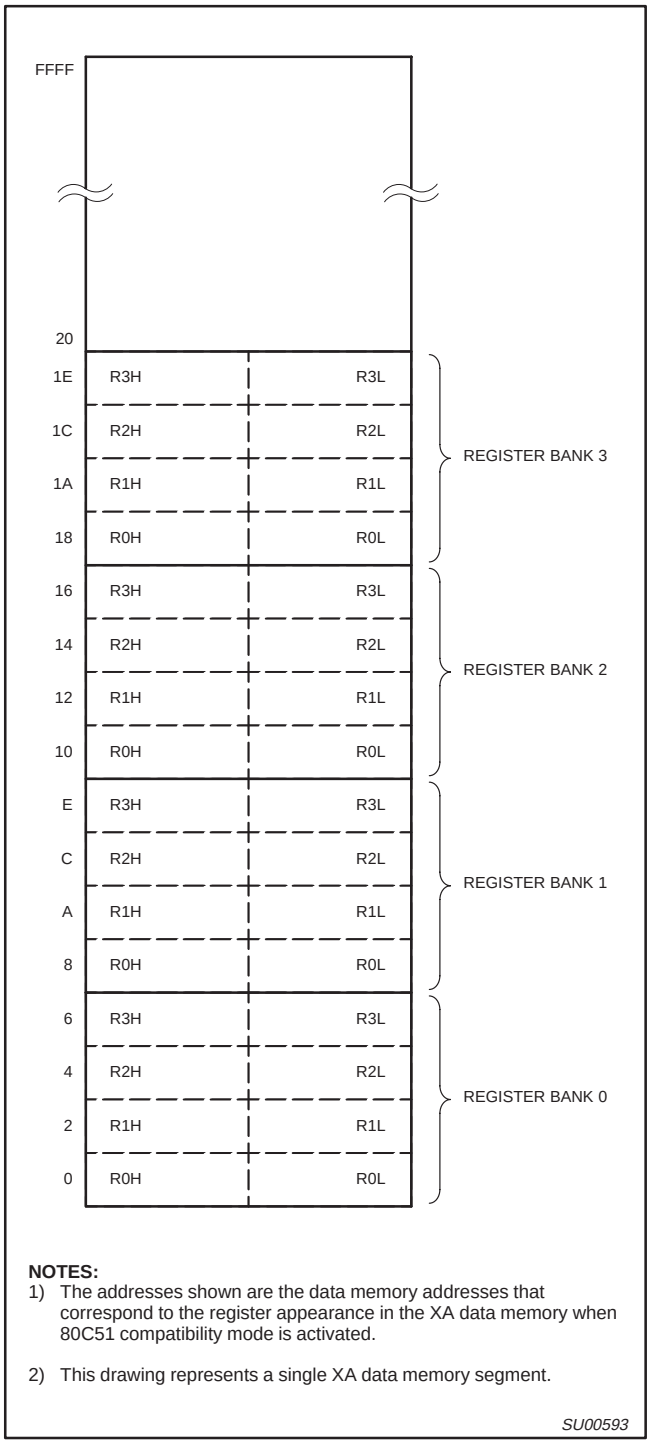

Figure 2. XA Register File and Data Memory Overlap

A second aspect of XA memory addressing is also controlled by the aforementioned 80C51 compatibility mode. In the XA, indirect memory accesses normally make use of a 16-bit pointer register, which may be any of the word registers in the register file. The 80C51, however, allows only the 2 single-byte registers R0 and R1 to used for indirect references. The XA is forced use the first 2 single-byte registers in the currently selected bank as byte pointers rather than word pointers when the 80C51 compatibility mode is activated. Thus, translated 80C51 code typically must be run with the compatibility mode activated.

The data memory map for a single XA data segment looks just like the entire data memory map for an 80C51. This leads to the possibility of using a single XA to perform the function of several 80C51s, with a separate data segment and code area allocated to a task that was originally performed by one 80C51. The XA includes hardware support for multi-tasking operation in order to allow for this and other interesting possibilities.

The XA retains the direct and indirect addressing modes of the 80C51, although both are greatly expanded in capability, as shown in figure 3. The direct data addressing has been increased to use up to 1K bytes of data memory. Indirect addressing is done in 64K byte segments, for a total of up to 16 megabytes. Both types of addressing seamlessly switch from internal to external data memory wherever the boundary exists between the two for a particular chip. In this manner, the processor stack may also be extended off-chip up to nearly 64K bytes if necessary. Because of the seamless internal to external memory transition, the XA would not normally attempt off-chip data accesses at the low memory addresses that correspond to the on-chip data RAM. For that reason, the 80C51 MOVX instruction is included on the XA in order to allow translated code to run without changes in the external memory address map. This works because MOVX always forces data to be read from off-chip memory.

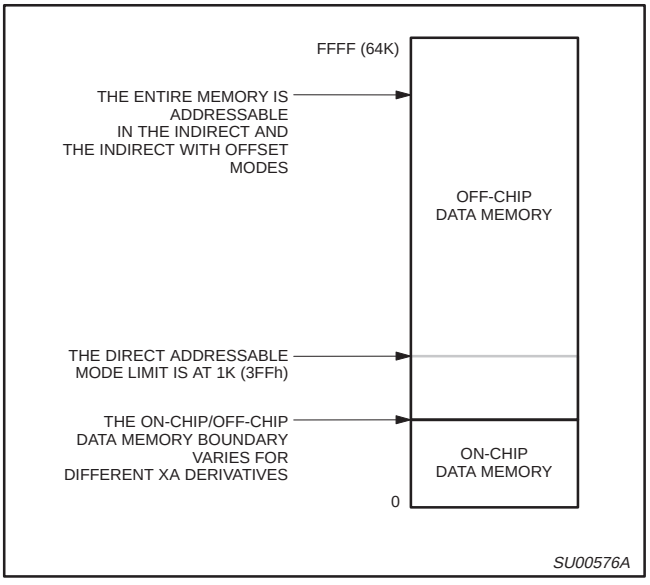

Figure 3. XA Memory Addressing

On the 80C51, the special function registers (SFRs) were mapped into the direct address space starting at location 128, through the end of that space at location 255. Since the 80C51 only allowed SFR access by direct addresses, where the entire address is encoded into the instruction, the XA does not need to duplicate its SFRs in exactly the same area or at the same specific addresses. In order to simplify the memory map, expand the SFR space, and expand the directly addressed data space, the XA defines a totally separate SFR space that is not logically related to the rest of data memory. To translate 80C51 source code, the original SFR name is kept in the translated code, unless the name was changed for some reason. In any case, as long as the reference is by name, a code translator need not try to determine which SFR it is, or where it belongs on a particular XA derivative. If 80C51 source code for some reason references an SFR by its address, a code translator might attempt to look it up in an SFR map for the 80C51 derivative to which the code was targeted.

A second mode control in the XA applies to 80C51 translated code, although it may be used in pure XA applications as well. This is the Page Zero, or PZ, mode. This mode forces the XA to only allow 64K of address space in both the data and code memories. The purpose is to reduce the overhead required to support the extra address space if it is not needed, such as in “single-chip” systems that do not use any off-chip data or program. Besides saving stack space for 24-bit subroutine and interrupt return addresses (reduced to 16 bits in PZ mode), overall XA operation is faster by having smaller stack pushes and pops. Since the 80C51 supported only 64K of code and data space, translated 80C51 code will likely fit into the same category.

There are other changes in the processor stack on the XA, besides the need to save 24 bits of return address when not running in the Page Zero mode. First, a great deal of extra hardware in the processor would be required to allow both byte and word pushes and pops on the stack, especially since word operations could then sometimes be mis-aligned from word address boundaries in the data memory, so stack operations on the XA are always done in word increments. Mis-aligned word operations, aside from being difficult to implement, would be very inefficient, since they would have to be split up into multiple byte operations. This means that translated 80C51 code run on the XA will tend to use somewhat more stack space than it did originally. The automatic save of the PSW during interrupts on the XA might also increase stack usage in some cases, since a few 80C51 programs may have been able to omit saving the PSW during interrupt processing.

Secondly, the XA stack has been altered so that the direction of growth is downward, conforming to the industry standard for stack operation on 16-bit processors. There is also a necessary relationship between the stack growth direction and the order in which the bytes of a word are stored in memory for a processor that is capable of stack relative addressing, as can be done with the XA. This relationship required that the stack grow downward since data on the XA is stored in memory with the low order byte of a word at the lower address (sometimes referred to as Little Endian storage order).

These differences in stack operation may require some changes to be made by the user for any 80C51 source code translated to the XA. In most cases, the change would be limited to choosing a different starting address for the stack.

A look at interrupt processing presents some other issues for 80C51 compatibility. In order to allow more powerful handling of interrupts, the XA has to make some compromises. Besides the previously mentioned fact that the PSW is automatically saved on the stack, which would have been done explicitly in 80C51 interrupt service code, the return address on the stack is also different if Page Zero mode is not active. So, any code written for the 80C51 which relied

in some manner on manipulating the return address on the stack, or on the PSW not being saved and restored automatically, will require modification. Both of these situations should be very rare. The standard (non-Page Zero mode) XA interrupt stack frame is shown in Figure 4.

CPU FEATURES

Another difference in interrupt processing is that the XA uses a more efficient and flexible vector table for interrupts and exceptions instead of the fixed vector scheme of the 80C51. The vector table must reside at the bottom of the code memory, since this is the only region that is guaranteed to always exist in a system that uses on-chip ROM or EPROM for the program. Thus, during 80C51 code translation, code found at the 80C51 interrupt service locations must be moved to another location. Of course, an interrupt vector table must be added to any translated 80C51 program that makes use of interrupts, and a reset vector entry must be created for all XA programs.

A major enhancement to the XA is the addition of a general purpose interrupt priority scheme that can support up to 15 levels, compared to only 2 on standard 80C51 parts, and up to 4 on enhanced parts. This addition, however, requires some changes in the way interrupt priorities are handled. Two-priority interrupt systems on 80C51 derivatives used a single bit in a priority register to select the two levels. Four-priority systems extended this to two bits, but in 2 different registers for each interrupt source. Extending that approach to 15 levels would entail 4 bits in 4 different registers for each interrupt source, which is getting a bit ridiculous. For the XA, a more reasonable approach was taken: 4 bits in a single register control the priority of each interrupt source. Priorities for 2 separate interrupts are contained in each 8-bit priority register.

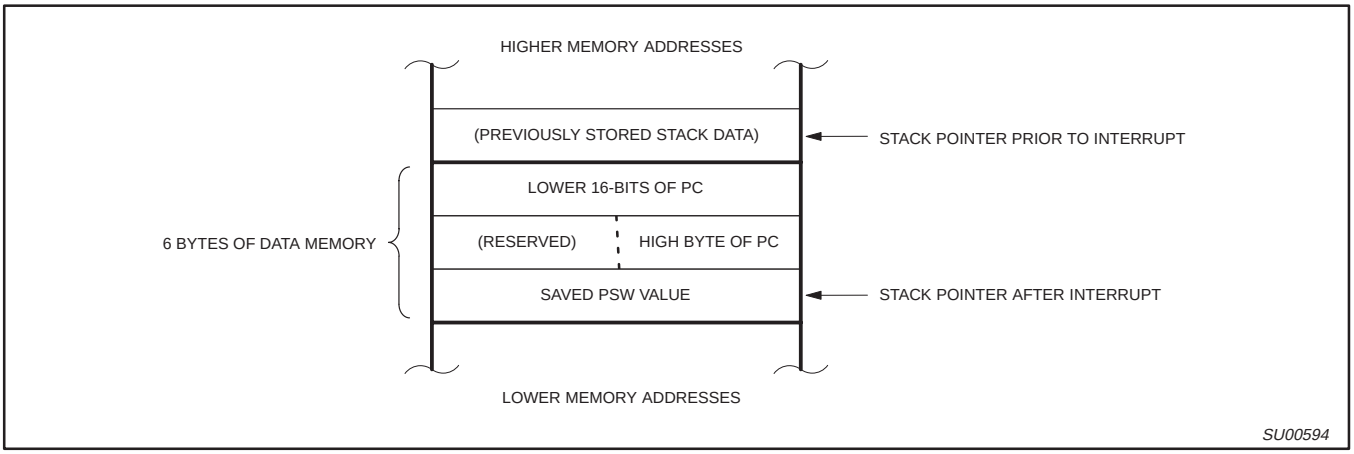

Figure 4. Standard Interrupt Stack Frame on the XA

PERIPHERALS, ON AND OFF-CHIP

Another subject to look at is hardware compatibility. While complete hardware compatibility with the 80C51 was not a primary goal during the XA architecture development, hardware compatibility was retained whenever possible and practical. This particularly concerns peripheral devices such as UARTs, Timers, etc., and the processor's external bus system.

In the case of peripherals that are the same as those customarily found on the 80C51, these have been made to function as close as possible to the original, with some transparent enhancements such as framing error detection, overrun detection, and break detection in the UARTs. One exception to this general compatibility is that timer mode 0 of the standard timers 0 and 1, which is the rarely used 8048 compatible timer mode, has been replaced with a much more useful 16-bit auto-reload mode. In the future, further enhanced peripheral functions will likely lead eventually to completely new implementations that are not backward compatible with the 80C51.

Since there is no supposed relationship between the original oscillator frequency of an 80C51 system and a similar XA system using translated code, the exact relationship of peripheral speeds to the oscillator need not be preserved. For more flexibility in timer rates and therefore UART baud rates, the XA timers and some other peripherals are operated from a special clock whose rate is user programmable. The choices are the CPU clock divided by 4, 16, or 64, giving a wide range of uses. This function, like anything else in an application that is time critical, will need to be visited by the user when translated 80C51 code is used to drive XA peripherals.

The standard XA external bus interface includes all of the familiar 80C51 bus signals: ALE, $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{EA}}$, the multiplexed address and data bus, and address-only lines. However, some additional signals have been added and changes have been made in some of the details. For instance, the XA supports both 8-bit and 16-bit bus widths, using a second write signal to distinguish byte writes on a 16-bit bus. A WAIT line allows external circuitry to insert wait states into bus cycles for slow peripherals or program memories.

The largest change in the XA bus from the 80C51 is in the mapping of the multiplexed address and data lines. The 80C51 has a somewhat inefficient mapping that requires an ALE (Address Latch Enable) cycle in order to latch the least significant bits of an address for all external bus cycles. This was not a concern for the 80C51 due to its machine cycle timing, which allowed plenty of time for an ALE pulse. For the XA, which has no extra cycles during instruction execution, any extra strobes required on the bus during code fetches will likely take away time that could be used to execute instructions. As a result, the XA drives the 4 lower address lines directly, and does not require them to be latched. This means that the XA can fetch as many as 16 bytes of code between ALE cycles. The multiplexed address and data bus begins with the fifth address line (A4), paired with the first data line (D0), and continues to the width of the bus, either 8 or 16 bits. Above that will be more always-driven address lines, if more are needed by the application. Since the XA allows programming the number of address lines, those above the multiplexed portion of the bus need not be driven by the XA if they are not needed, leaving them free for other functions.

These changes mean that an XA device may be made pin compatible with a similar 80C51 derivative if the external bus is not used. Small changes to the external hardware must be made if the external bus is in use. Internally programmable bus cycle timing control on the XA allows programming the duration of all of the bus cycles, allowing nearly all memory and peripheral devices to be used on the XA bus without the need for an external WAIT state generator or any other additional circuitry.

INSTRUCTIONS REVISITED

The earlier mentioned goal of the XA to map nearly every 80C51 instruction to a single XA instruction was met. Just one 80C51 instruction cannot be replaced by single XA instruction. That instruction is XCHD (exchange digit), a seldom used 80C51 instruction. This unusual instruction exchanges the lower nibble of the 80C51 accumulator with a nibble at an internal RAM address pointed to by byte register R0 or R1. The XA would have required additional special circuitry in order to support this operation. As a result, it was decided to allow a multi-instruction sequence in this case, since the instruction is rarely used. The sequence used to replace XCHD is:

```
PUSH    R4H         ; save temporary register.
MOV     R4H, (Ri)   ; get second operand.
RR      R4H, #4     ; swap one byte.
RR      R4L, #4     ; swap second byte (the "A" register).
RL      R4, #4      ; swap word, result is swapped nibbles in A
                      and R4H.
MOV     (Ri), R4H   ; store result.
POP     R4H         ; restore temporary register.
```

Some additional code may be needed if an application requires this sequence to be un-interruptable for some reason. All other 80C51 instructions translate one-to-one to XA instructions. Since the XA instruction set and memory model are a superset of the 80C51, and since most mnemonics and names were kept the same, 80C51 code translated for the XA looks nearly the same as the original. Some examples are shown below.

Table 1. Examples of 80C51 to XA Source Code Translation

TYPE OF OPERATION	80C51 SOURCE CODE		XA SOURCE CODE	
Move immediate to SFR.	MOV	TCON,#00h	MOV.B	TCON,#00h
Move direct address to accumulator.	MOV	A,TstDat	MOV.B	R4L,TstDat
Move register to register.	MOV	R5,A		
Arithmetic with 2 registers.	ADD	A,R1	ADD.B	R4L,R0H
Arithmetic with register and immediate.	SUBB	A,#'0'	SUBB.B	R4L,#'0'
Increment a register.	INC	R0	ADDS.B	R0L,#1
Test a register.	CJNE	A,#'0',Cmd1	CJNE.B	R4L,#'0',Cmd1
Clear a bit.	CLR	RxFlag	CLR	RxFlag
Set a bit.	SETB	EX1	SETB	EX1
Test a bit.	JNB	RcvRdy,Wait	JNB	RcvRdy,Wait
Subroutine call.	ACALL	Test	CALL	Test
Subroutine return	RET		RET	
Push register onto stack.	PUSH	ACC	PUSH.B	R4L
Pop register from stack.	POP	ACC	POP.B	R4L

Details of instruction translation for the entire 80C51 instruction set are available in the Philips XA User Guide.

One side effect of source code compatibility of the XA with the 80C51 is that the number of bytes required to encode some instructions changes between the two processors. In most cases, this is not a major concern, however it does raise issues with the translated code for some situations. A simple example of this is that a conditional branch could have the target address move out of range when translated code is re-assembled. This should be a rare occurrence since the range of short relative branches on the XA has been doubled to 256 bytes forward or backward. The same issue does not exist for farther jumps and calls since the XA extends that range to beyond the entire 80C51 address range.

The precise length of a branch instruction is of concern in certain cases, such as a table of jump instructions entered using the JMP @A+DPTR instruction of the 80C51. The XA instruction set includes this jump, but does not include a 2-byte replacement for the 80C51 AJMP instruction which is often used in jump tables. The user will have to make small changes to the indexing into such a table if it is translated to run on the XA.

A similar issue can arise for a translation of the 80C51 instruction MOVC A,@A+PC, since the distance from this instruction to the lookup table that it is accessing may change. The solution is the same as for JMP @A+DPTR: some user intervention to adjust the table index.

User intervention will also be needed in any case where the timing of instructions in the original 80C51 code is of importance. The XA reduces the execution time of each instruction to the minimum possible with its internal hardware implementation. Also, instructions are normally fetched into a small queue prior to being needed to continue execution, which can lend additional uncertainty to execution times. The execution time of loops or the time between particular instructions can be calculated and adjusted by the use of NOPs, delay loops, or other means of matching timing. Also, any variable execution timing of the same code due to it being entered in different ways can be handled with certain coding techniques. An example would be a loop that is entered by "falling through" the preceding code on the first instance and branching back to be repeated on subsequent occasions. The branch back takes extra time not seen on the first entrance to the code due to the necessity of "flushing" the queue on a branch. The solution in this case is to add a branch instruction prior to the loop branching to the first instruction of the loop. Then, each cycle through the loop acquires the same timing. Of course, a simple source code translator cannot sense such cases and attempt to deal with them automatically.

AN EXAMPLE

As an example of translating 80C51 source code into XA source code, an actual piece of 80C51 code from a working application was taken and translated using the rules that were presented above. The results of the simple one-to-one translation are shown below.

Table 2. Sample 80C51 Routines Translated for the XA

Original 80C51 source code:				Translated XA source code:			
; Sets up UART and Timer (for baud rate generation), prints a string,							
; and prints a hexadecimal value.							
Start:	MOV	SCON,#42h	; Set UART for 8-bit variable rate.	Start:	MOV.B	SCON,#42h	
	MOV	TMOD,#20h	; Set Timer1 for 8-bit auto-reload.		MOV.B	TMOD,#20h	
	MOV	TCON,#00h	; Stop timer 1 and clear flag.		MOV.B	TCON,#00h	
	MOV	TL1,#0FDh	; Set timer for 9600 baud @ 11.0592 MHz.		MOV.B	TL1,#0FDh	
	MOV	TH1,#0FDh	; Set reload register for same rate.		MOV.B	TH1,#0FDh	
	MOV	A,PCON	; Make sure SMOD bit in PCON is		MOV.B	R4L,PCON	
	CLR	ACC.7	; cleared for this baud rate.		CLR	R4L.7	
	MOV	PCON,A			MOV.B	PCON,R4L	
	SETB	TR1	; Start timer		SETB	TR1	
	MOV	DPTR,#Msg1	; Send a stored message.		MOV.W	R6,#Msg1	
	ACALL	Msg			CALL	Msg	
	MOV	A,P1	; Send Port 1 value as hexadecimal.		MOV.B	R4L,P1	
	ACALL	PrByte			CALL	PrByte	
	.	.			.	.	
	.	.			.	.	
	.	.			.	.	
;**********							
; Subroutines							
;**********							
; Print byte routine: print ACC contents as ASCII							
; hexadecimal.							
PrByte:	PUSH	ACC		PrByte:	PUSH.B	ACC	
	SWAP	A			RL.B	R4L,#4	
	ACALL	HexAsc			CALL	HexAsc	
	ACALL	XmtByte			CALL	XmtByte	
	POP	ACC			POP.B	ACC	
	ACALL	HexAsc	; Print nibble in ACC as ASCII hex.		CALL	HexAsc	
	ACALL	XmtByte			CALL	XmtByte	
	RET				RET		
; Hexadecimal to ASCII conversion routine.							
; Converts a nibble to ASCII hex.							
HexAsc:	ANL	A,#0FH		HexAsc:	AND.B	R4L,#0FH	
	JNB	ACC.3,NoAdj			JNB	R4L.3,NoAdj	
	JB	ACC.2,Adj			JB	R4L.2,Adj	
	JNB	ACC.1,NoAdj			JNB	R4L.1,NoAdj	
Adj:	ADD	A,#07H		Adj:	ADD.B	R4L,#07H	
NoAdj:	ADD	A,#30H		NoAdj:	ADD.B	R4L,#30H	
	RET				RET		

Original 80C51 source code:

```
; Message string transmit routine.

Msg:    PUSH    ACC
        MOV     R0,#0           ; R0 is character pointer (string
MsgL:   MOV     A,R0            ;   length is limited to 256 bytes).
        MOVC    A,@A+DPTR       ; Get byte to send.
        CJNE    A,#0,Send       ; End of string is indicated by a 0.
        POP     ACC
        RET

Send:   ACALL   XmtByte         ; Send a character.
        INC     R0              ; Next character.
        SJMP    MsgL

Msg1:   DB      0Dh,0Ah,0Dh,0Ah
        DB      'Port 1 value = ', 0

; Wait for UART ready, then send a byte.

XmtByte: JNB    TI,$
        CLR     TI
        MOV     SBUF,A
        RET
```

Translated XA source code:

```
Msg:    PUSH.B  ACC
        MOV.B   R0L,#0
MsgL:   MOV.B   R4L,R0L
        MOVC.B  A,[A+DPTR]
        CJNE.B  R4L,#0,Send
        POP.B   ACC
        RET

Send:   CALL    XmtByte
        ADDS.B  R0L,#1
        BR      MsgL

Msg1:   DB      0Dh,0Ah,0Dh,0Ah
        DB      'Port 1 value = ',0

XmtByte: JNB    TI,$
        CLR     TI
        MOV.B   SBUF,R4L
        RET
```

The translated XA code looks very much like the 80C51 source code and can easily be read by anyone familiar with the original program. Statistics for this example are shown in the following table.

Table 3. Statistics on Sample 80C51 to XA Code Translation

STATISTIC	80C51 CODE	XA TRANSLATION	COMMENTS
Bytes to encode	107	151	– Includes NOPs added for branch alignment on XA.
Clocks to execute	840	212	– Raw execution time for instructions in code, without flow analysis. Conditional branch times calculated as if half taken, half not taken.
Time to execute @ 20 MHz	42 sec	10.6 sec	– A 4x speed improvement for a simple translation with no optimization.

SOME XA ENHANCEMENTS

The subject of this article has been how the new Philips XA microcontroller architecture supports upward compatibility with the 80C51. The XA adds quite a bit to the equation beyond mere 80C51 compatibility, which has barely been touched upon here. In addition to high performance and very compact instruction encoding, the XA is specifically designed for high level language support for compilers such as C, has many features to support multi-tasking, with protected features and separate memory spaces, many 32-bit operations in addition to general 16-bit arithmetic, and greatly enhanced interrupt processing, to name a few. A complete description of all of these features and many more may be found in the XA User Guide and data sheets for specific parts.

THE UPWARD SPIRAL

Many openings have been left in the XA architecture for even more enhancements in the future, such as full pipelining, complete 32-bit operation support, or a faster peripheral bus. The XA is the foundation of a new microcontroller derivative family in a manner similar to the very popular 80C51 family. Many other advanced microcontroller architectures have been brought to market since the 80C51 was designed years ago. But until now, none has allowed the enormous quantities of 80C51 code that users have on file to be re-used with minimal effort on a state-of-the-art 16-bit processor. With the Philips XA, that is now possible, while getting the benefit of a modern 16-bit processor with few compromises.

An upward migration path for the 80C51: the Philips XA architecture

AN704

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS

Philips Semiconductors and Philips Electronics North America Corporation Products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation Product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation Products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from such improper use or sale.

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381

Philips Semiconductors and Philips Electronics North America Corporation register eligible circuits under the Semiconductor Chip Protection Act.
© Copyright Philips Electronics North America Corporation 1995
All rights reserved. Printed in U.S.A.