UM10038

ISP1583 Hi-Speed USB Device Split Bus Eval Kit

Rev. 04 — 24 April 2007 **User manual**

Document information

Info	Content
Keywords	isp1582; isp1583; peripheral controller; split bus; scanner; printer
Abstract	This document explains the implementation of the ISP1583 split bus mode.

Revision history

Rev	Date	Description
04	20070424	Removed PLCC_P89C58. Updated Fig 2 and Fig 22.
03	20070117	Updated the following: • Ported to the latest NXP template. • Updated Section 9.
02	20031007	Updated the following: • All figures related to the eval kit. • Section 3 • Section 5.1 • Section 6.2 • Section 6.5 • Section 8.3 • Section 9 • Section 11
01	20030908	First release.

Contact information

For additional information, please visit: **http://www.nxp.com**

For sales office addresses, please send an email to: **salesaddresses@nxp.com**

1. Introduction

The ISP1583 Hi-Speed Universal Serial Bus (USB) device split bus eval kit enables you to evaluate the features of the ISP1583 in split bus mode. In this mode, the ISP1583 Parallel I/O (PIO) and Generic Direct Memory Access (GDMA) device slave modes are evaluated.

This split bus eval board has onboard the ISP1583, Xilinx XC95288XL, SRAM and 8051 series microcontroller. The kit allows you to connect the ISP1583 to any generic processor when it is configured to separate address and data bus mode (generic processor mode).

Fig 1 shows the ISP1583 split bus board.

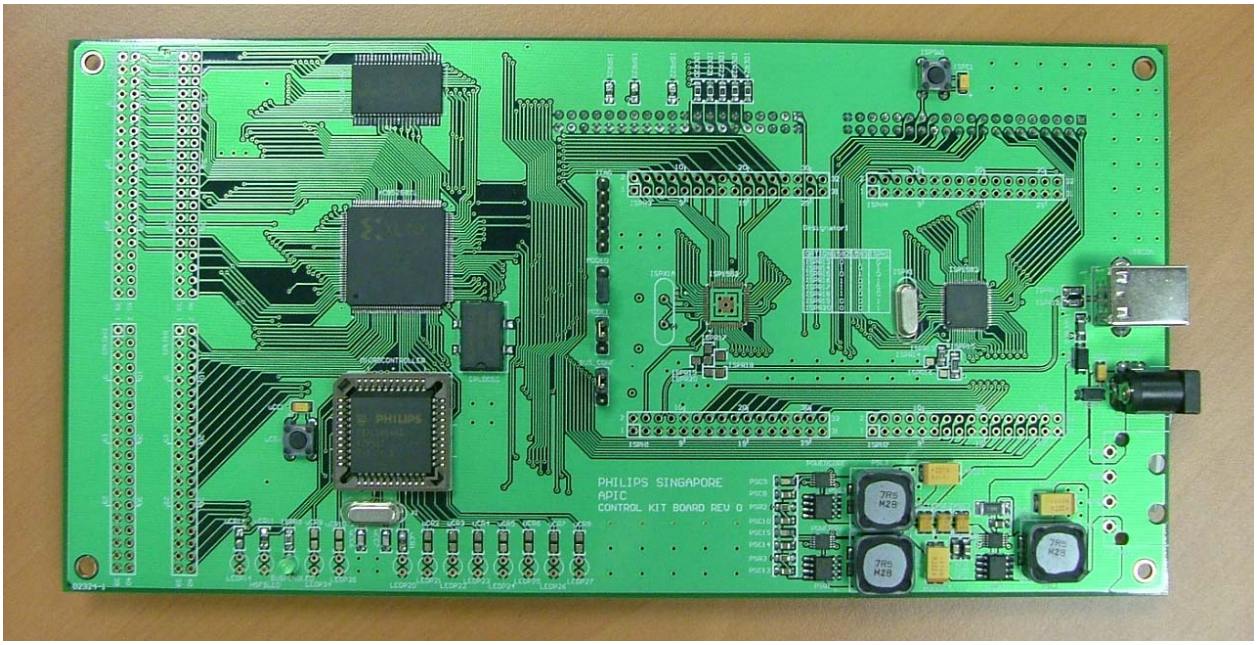


Fig 1. ISP1583 split bus board

2. System requirements

PC host

- Hi-Speed USB Host Controller*
- Ping pong application for GDMA for Microsoft Windows 2000 and Windows XP

Device

- 12 V DC power supply

Firmware

- Keil C Compiler*
- Firmware for the split bus eval kit

* - Denotes that the item will not be included in the eval kit.

3. Block diagram

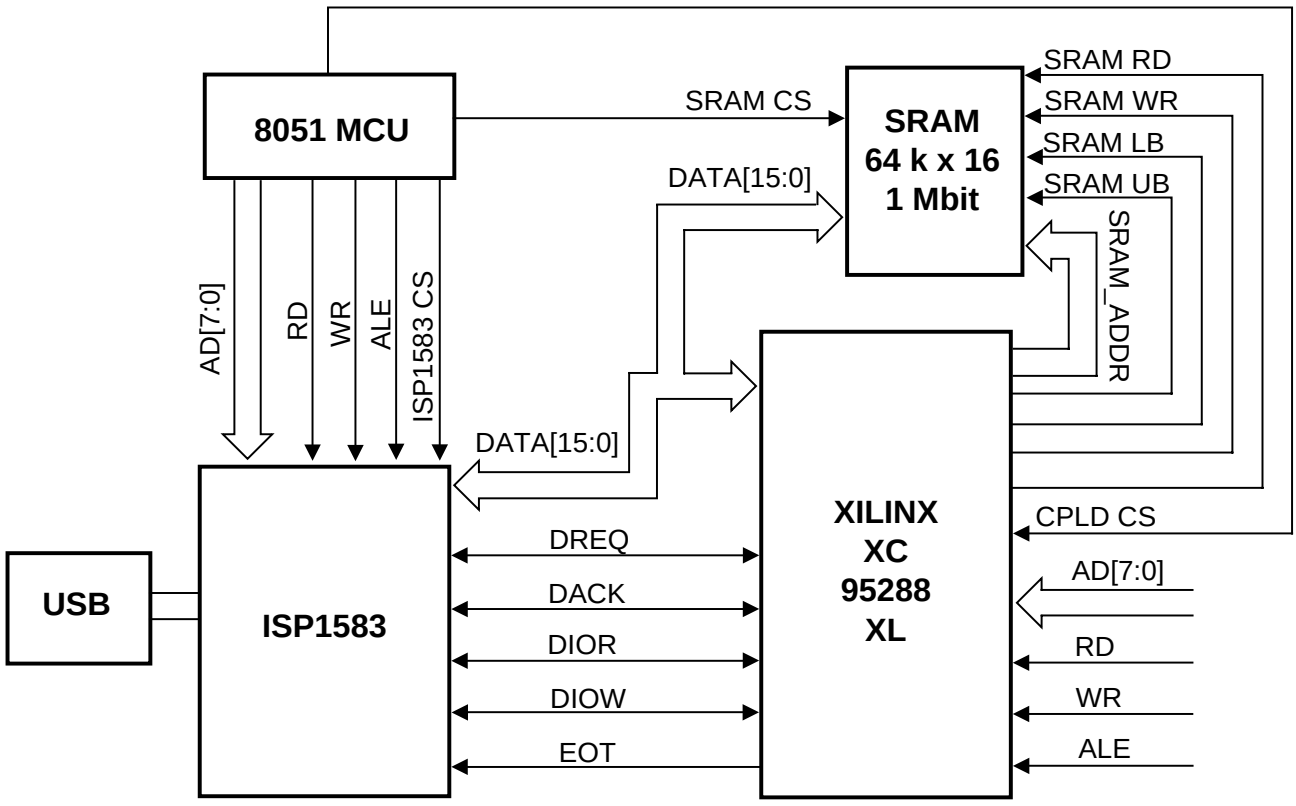


Fig 2. ISP1583 split bus eval board block diagram

Fig 2 shows the ISP1583 configured to operate in split bus mode. Xilinx XC95288XL acts as the local DMA controller. On the split bus kit, data from the DMA or PIO access is stored in the SRAM.

4. PCB layout

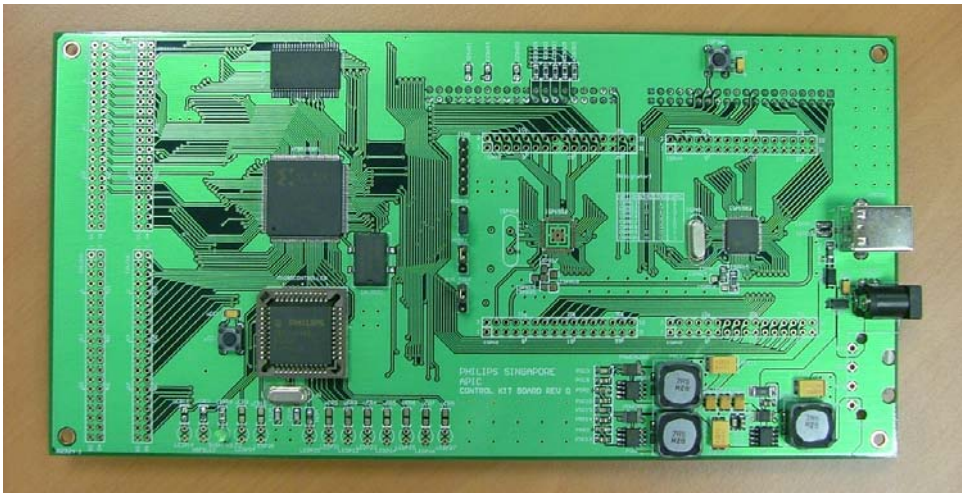


Fig 3. ISP1583 split bus eval board

Fig 3 shows the Printed-Circuit Board (PCB) layout and placement of components on the ISP1583 split bus eval board. The PCB is designed for future expansion for the ISP1582.

5. Component placement

5.1 ISP1583

Fig 4. Location of the ISP1583 on the eval board

5.2 ISP1582

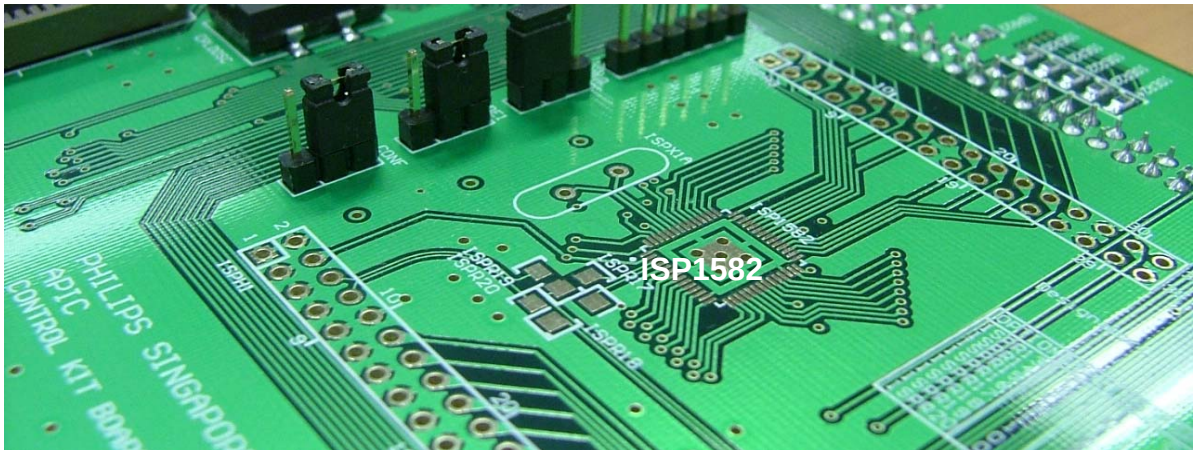


Fig 5. ISP1582 footprint

There is a footprint for the ISP1582 to cater for future expansion of the split bus kit for the ISP1582.

5.3 Xilinx XC95288XL CPLD

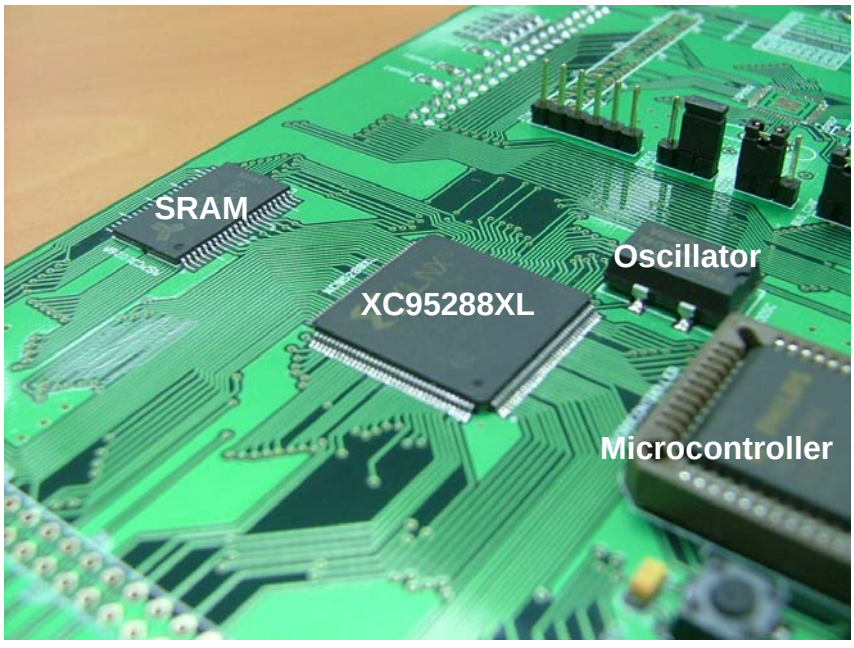


Fig 6. Xilinx XC95288XL

Xilinx XC95288XL acts as a generic DMA controller for DMA transfer.

6. Header and connector placement

6.1 USB and DC power input supply connectors

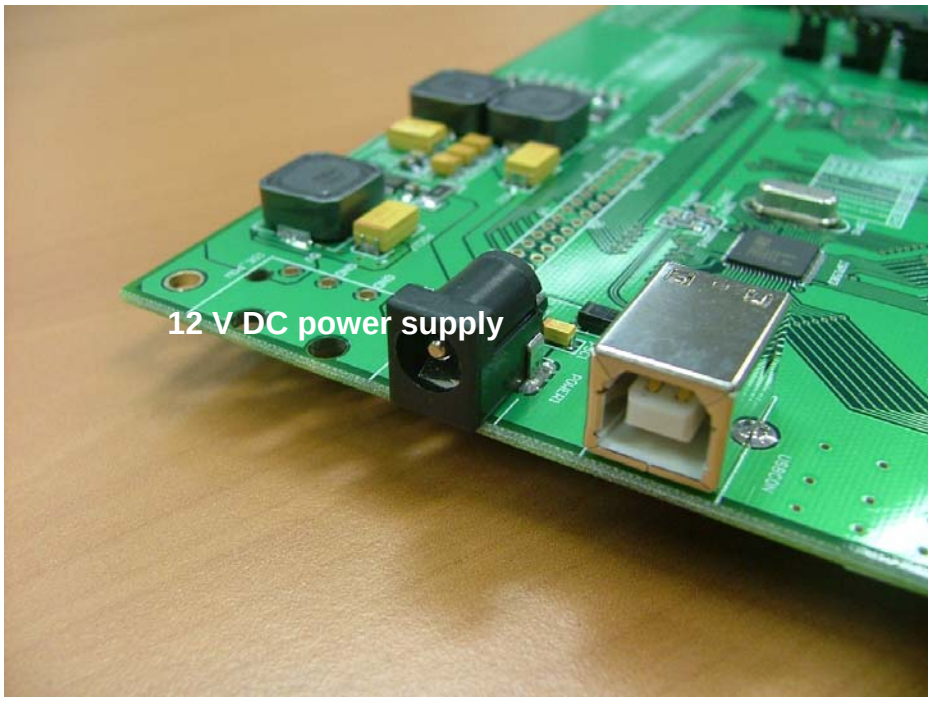
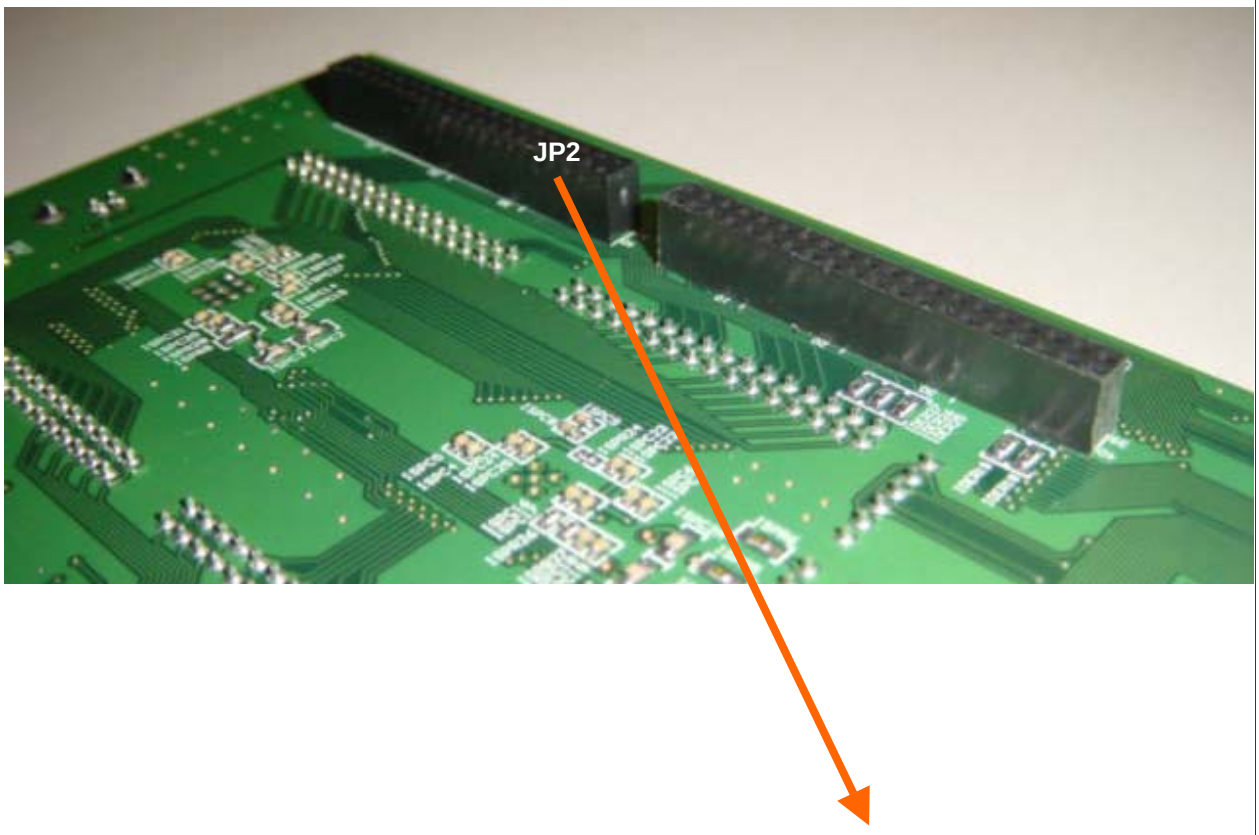


Fig 7. DC power supply input and USB connectors

The ISP1583 USB connector is next to the 12-V DC power supply input.

6.2 ISP1583 processor expansion bus



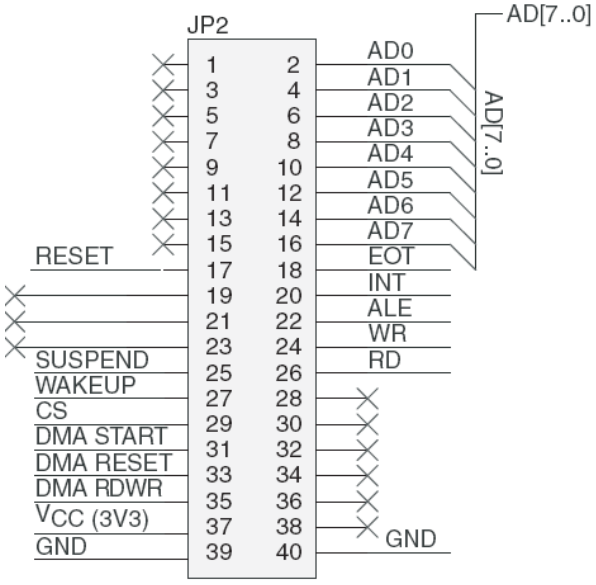


Fig 8. JP2

JP2 act as an expansion bus to connect to another processor or microcontroller.

6.3 ISP1583 DMA expansion bus

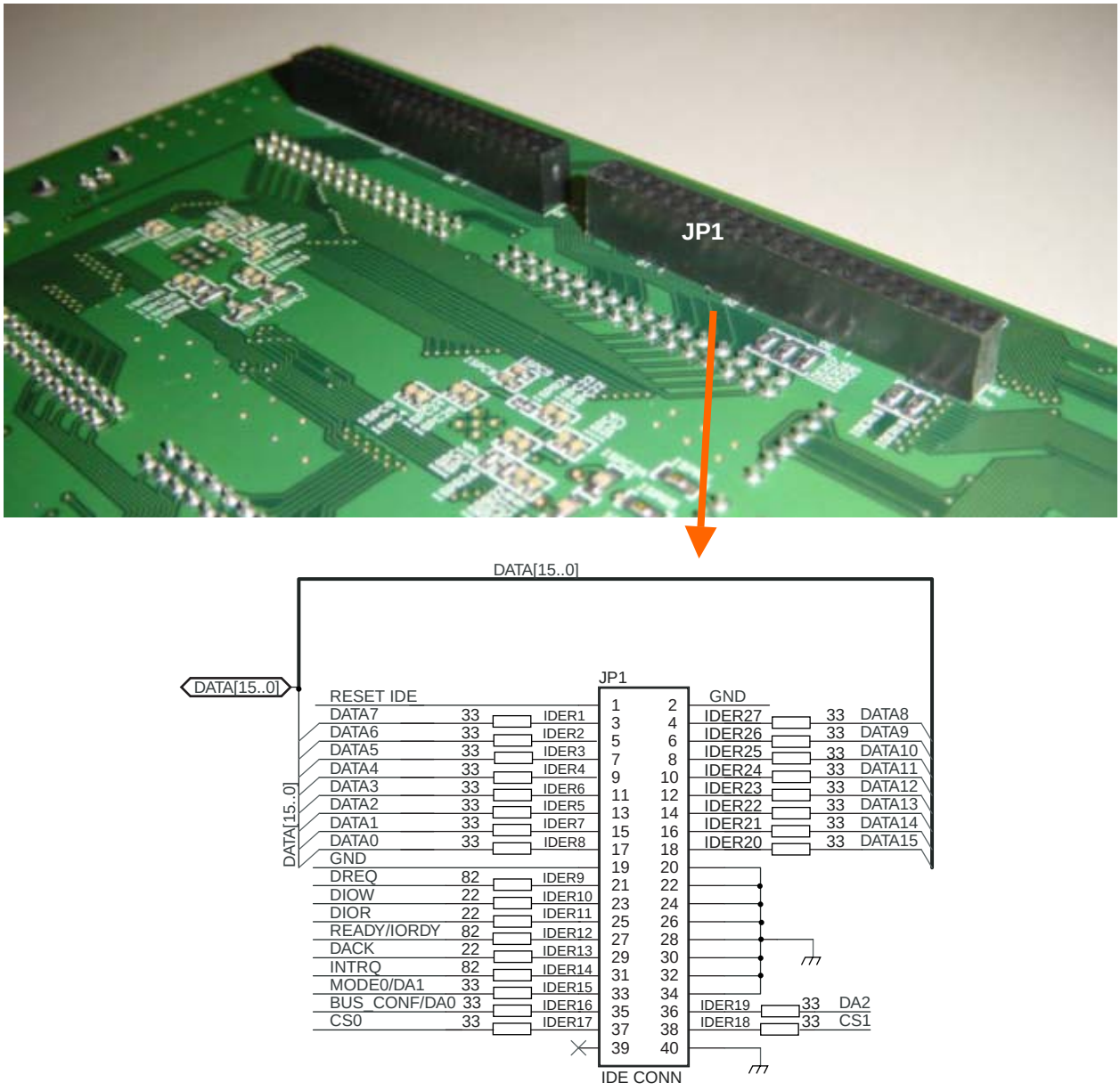
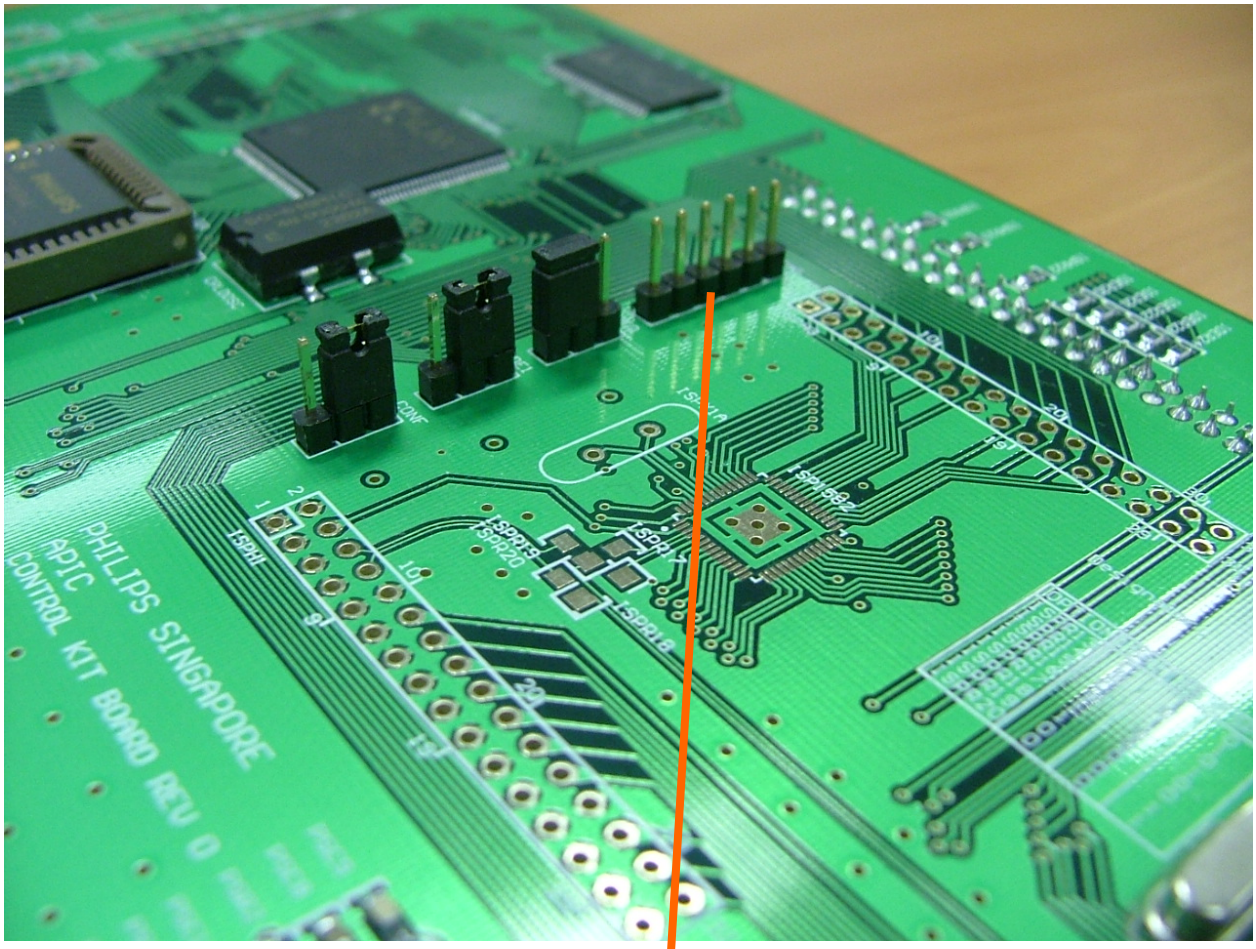


Fig 9. JP1

JP1 acts as a DMA expansion bus to connect to an external DMA controller and the ISP1583's 16-bit data bus.

6.4 JTAG header



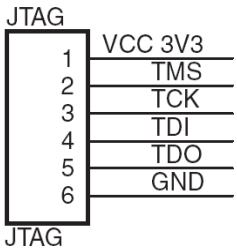


Fig 10. JTAG

The JTAG header allows you to reprogram Xilinx XC95288XL for user-defined operations.

6.5 ISP1583 processor selector

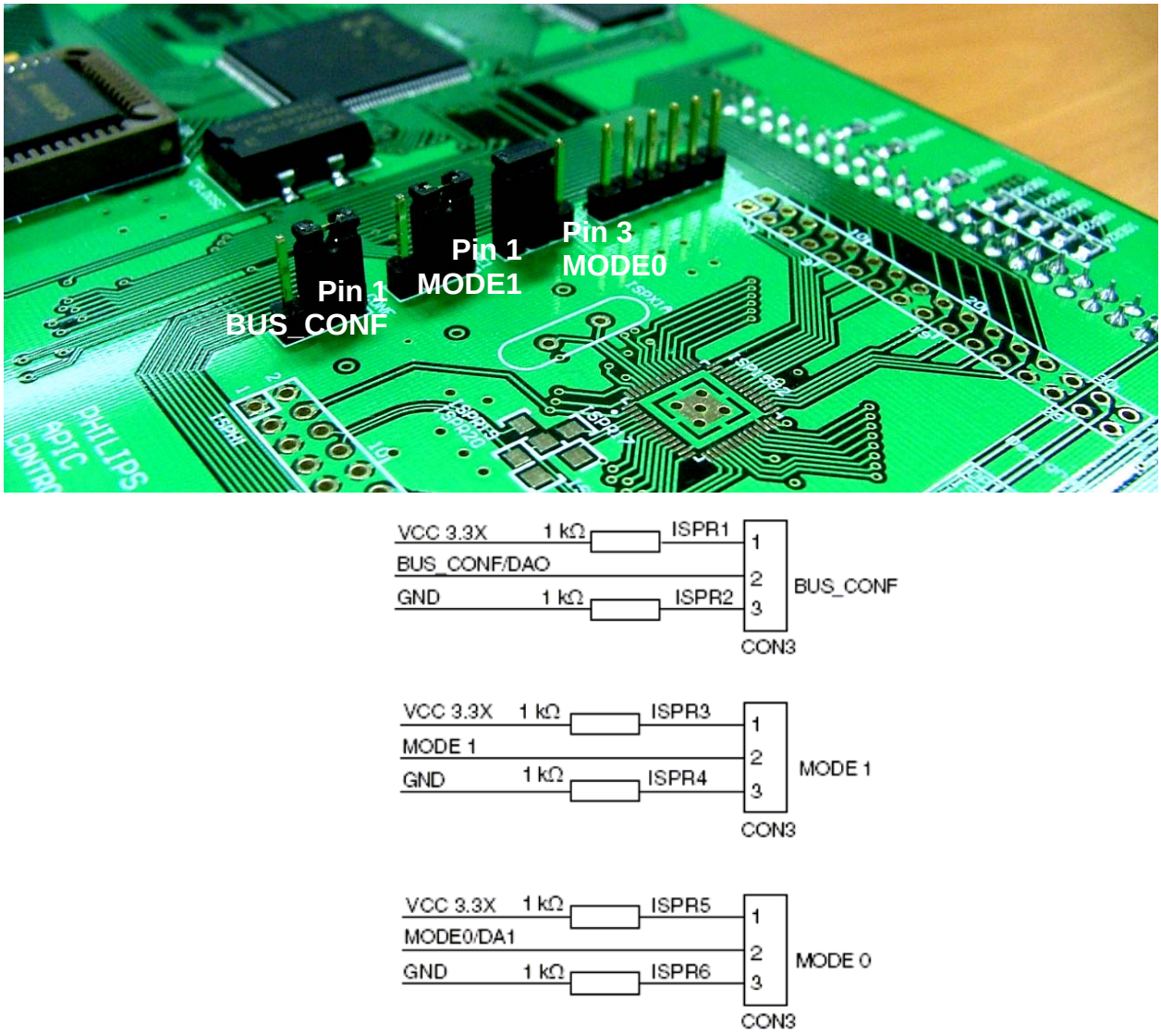


Fig 11. BUS_CONF, MODE1 and MODE0

The ISP1583 split bus eval kit is configured to run under the multiplexed 8-bit address and data bus (split bus mode).

7. Switch and LED placement

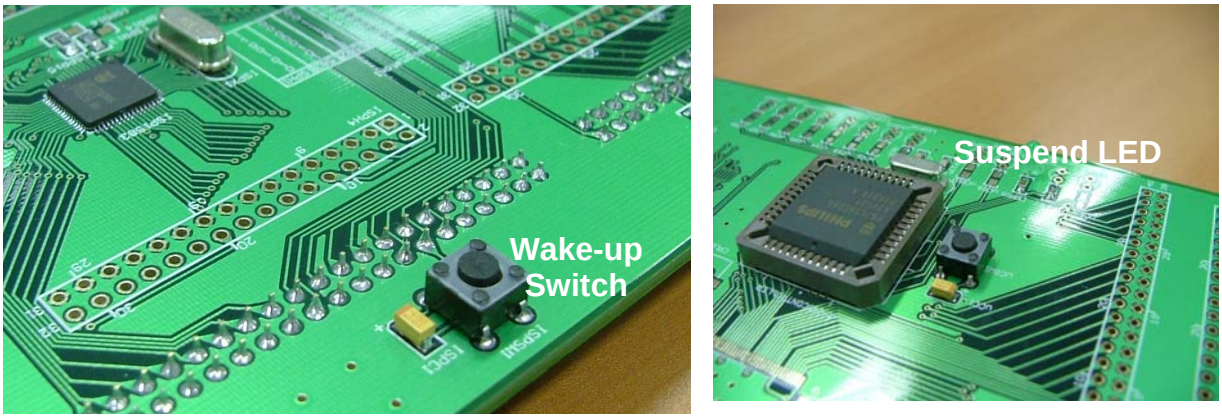


Fig 12. Switch and LED placement

The wake-up switch is connected to the ISP1583's wake-up pin, which will wake-up the ISP1583 when it is in suspend mode. The suspend LED when lit indicates that the ISP1583 is in suspend mode.

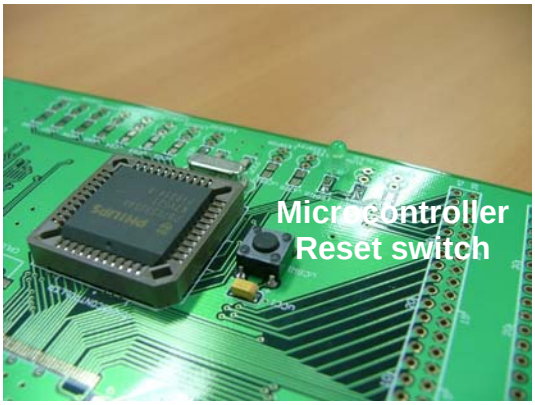


Fig 13. Microcontroller reset switch

Microcontroller reset switch resets the microcontroller, which in turn resets the ISP1583.

8. ISP1583 split bus eval kit set-up procedure

8.1 Split bus kit set-up procedure

Caution: Ensure that the BUS_CONF, MODE0 and MODE1 pins are at the default setting.

Table 1. Split bus mode setting

Processor mode	BUS_CONF pin	MODE1 pin	MODE0 pin
Split bus mode	2 to 3	2 to 3	1 to 2

1. Insert the 12-V DC power supply that is supplied together with the kit to the DC jack and switch on the power.

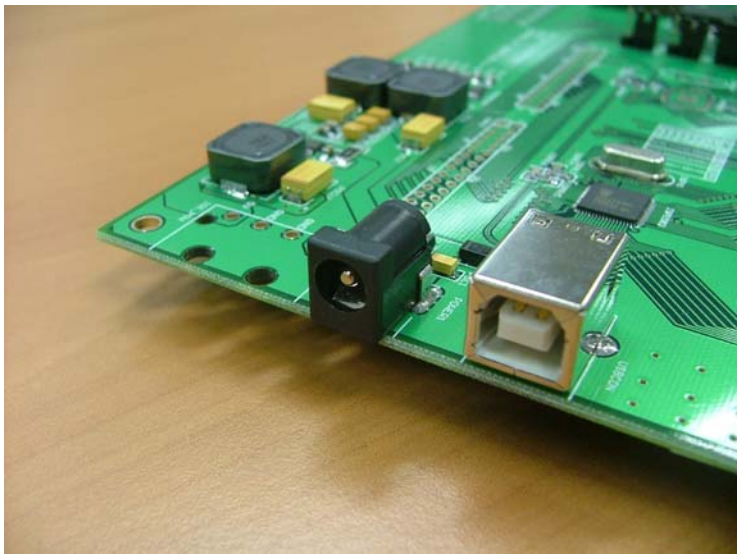


Fig 14. Power socket

2. Press the microcontroller reset switch.

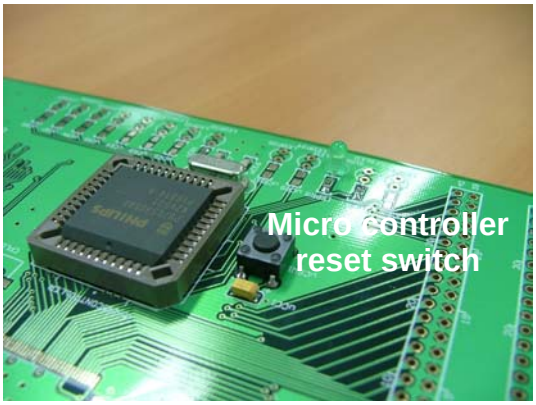


Fig 15. Microcontroller reset switch

3. Plug in the USB cable to the ISP1583 USB connector.
4. After successful enumeration, run the USB device applet on the host PC.

8.2 Split bus kit host PC set-up and bus enumeration procedure

If it is the first time the eval board is connected to the host PC, the host operating system Device Manager will prompt for the installation of the INF and the driver. Select the location of Phkit.inf and Phkit.sys from the ISP1583 eval diskette and complete the installation procedure.

On successful installation, you will see the device added in the Computer Management window under Device Manager as shown in Fig 16.

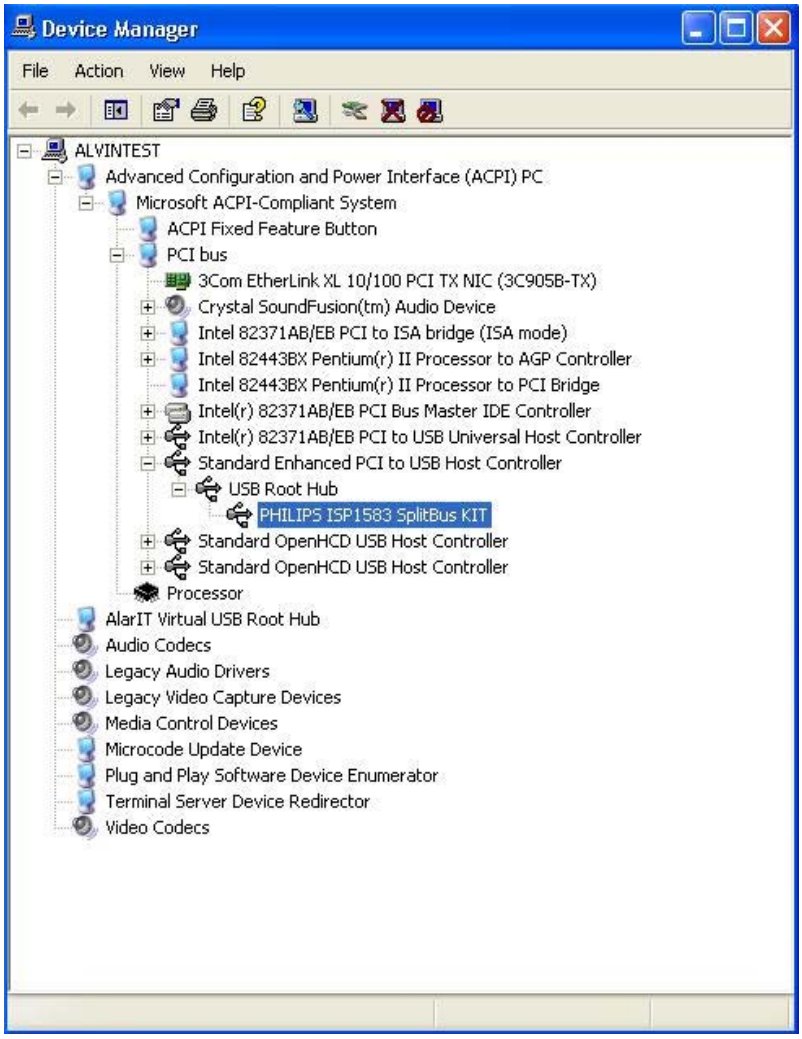


Fig 16. Hi-Speed USB device on NXP ISP1561 EHCI Hi-Speed USB controller

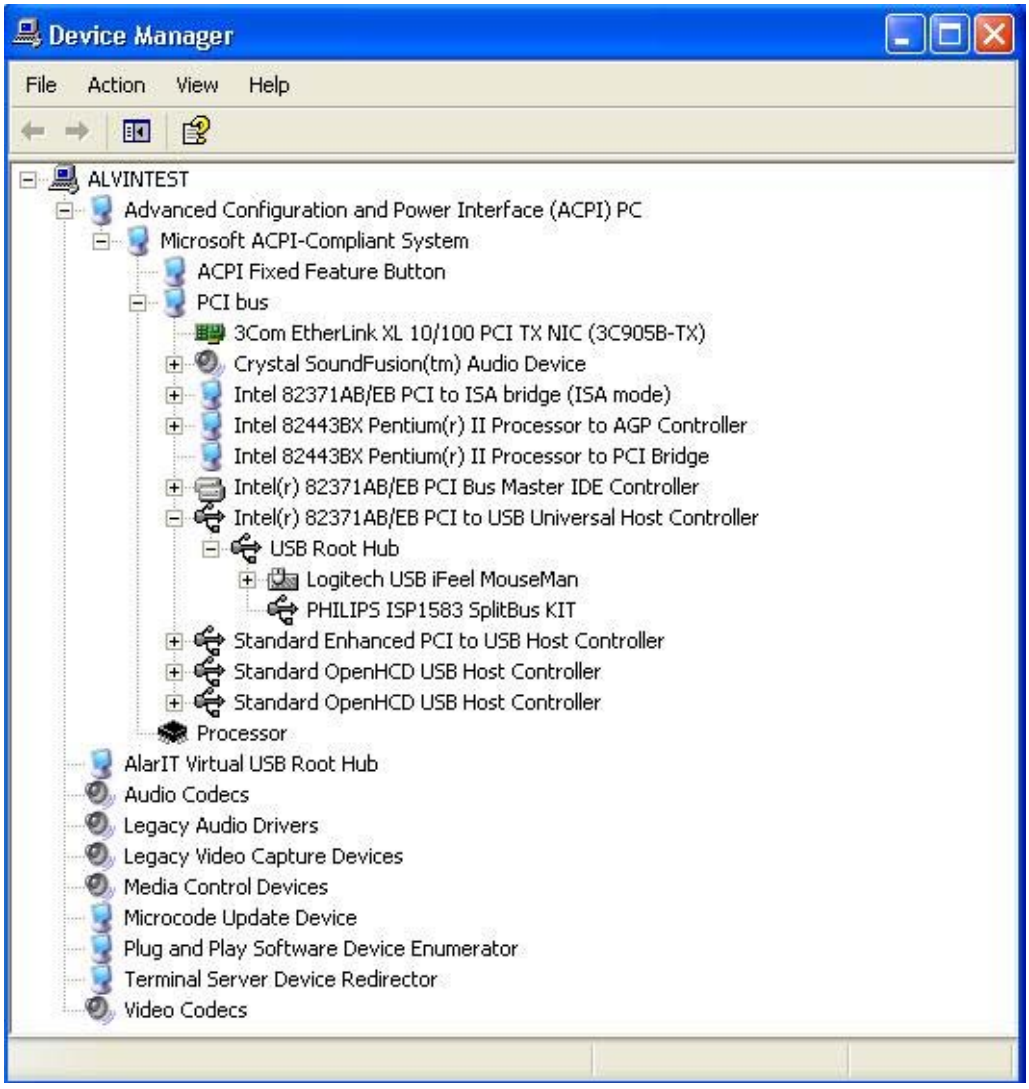


Fig 17. Original USB device on Intel UHCI USB controller

8.3 Split bus kit test application

This application allows you to perform data transfer using GDMA slave mode of the ISP1583. It sends data to the ISP1583 and reads it back, and so checks for data integrity.

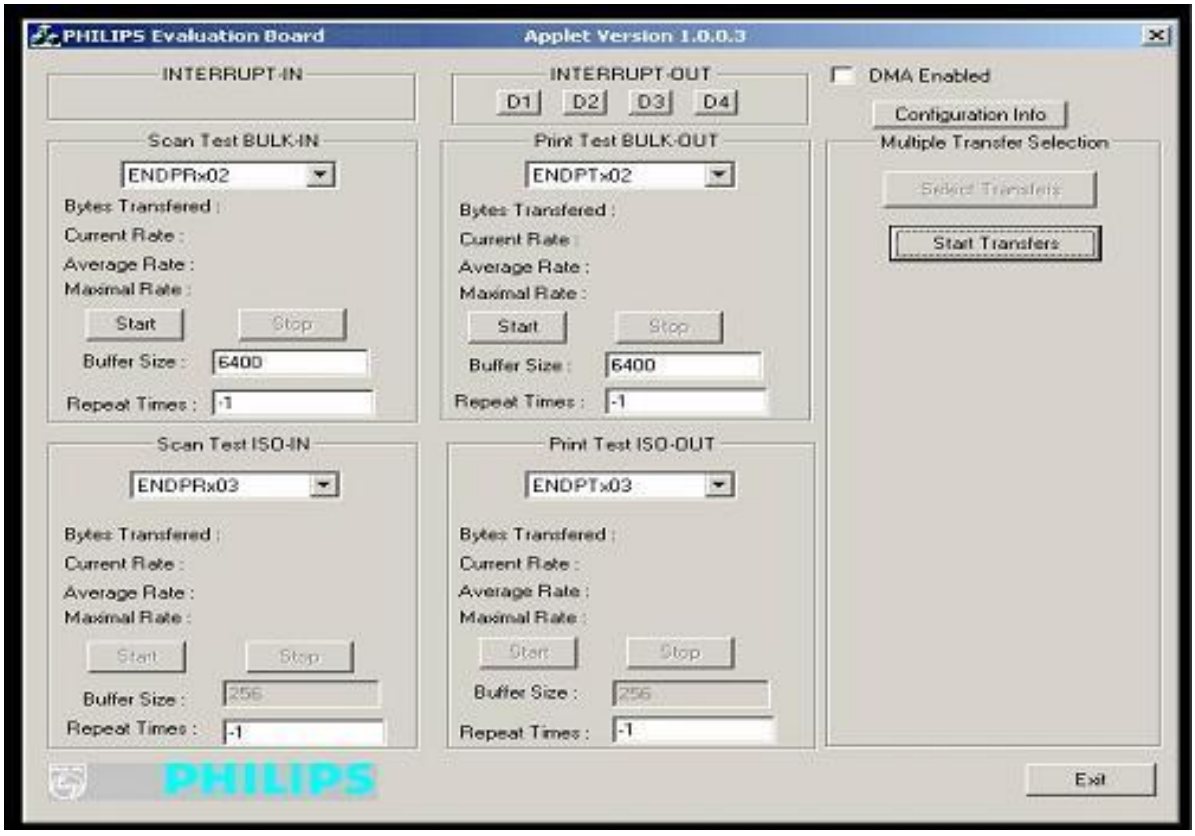


Fig 18. Applet

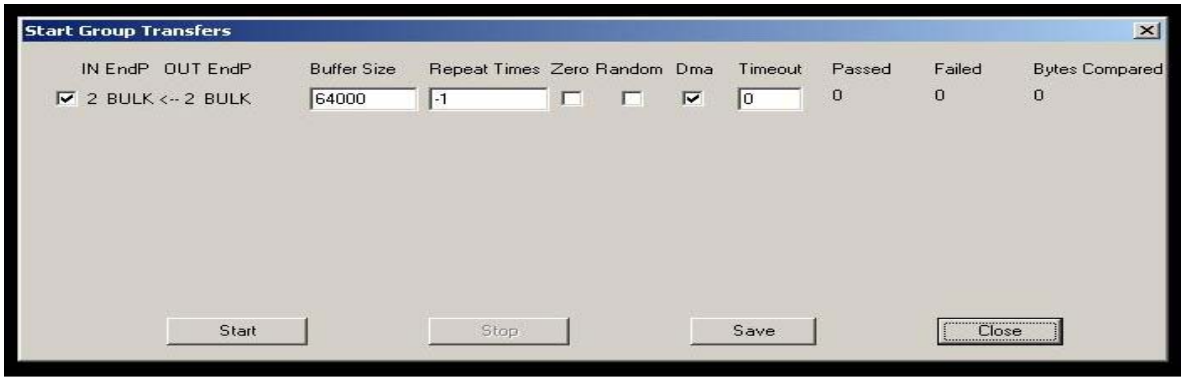


Fig 19. Start Group Transfers

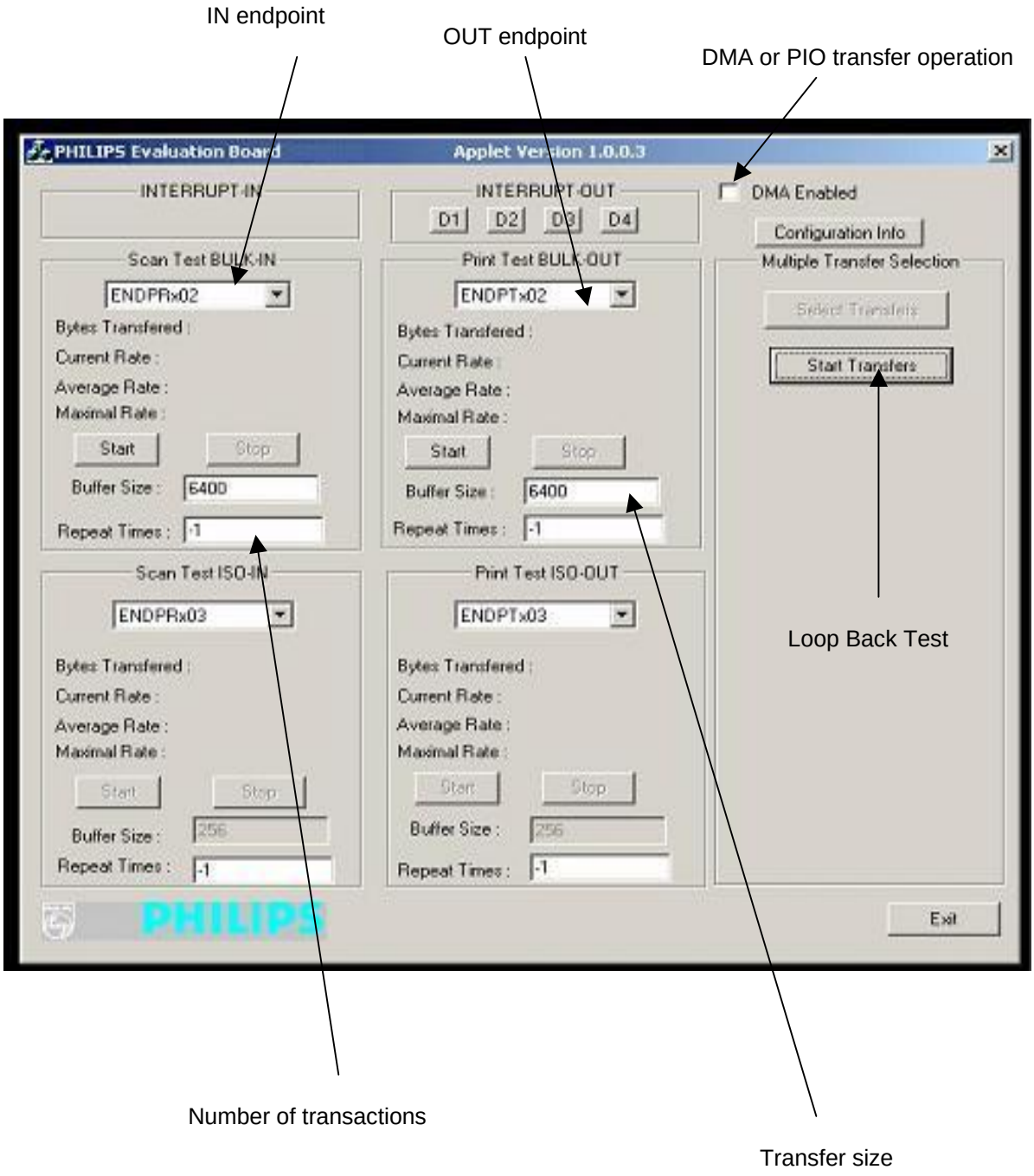


Fig 20. Applet in detail

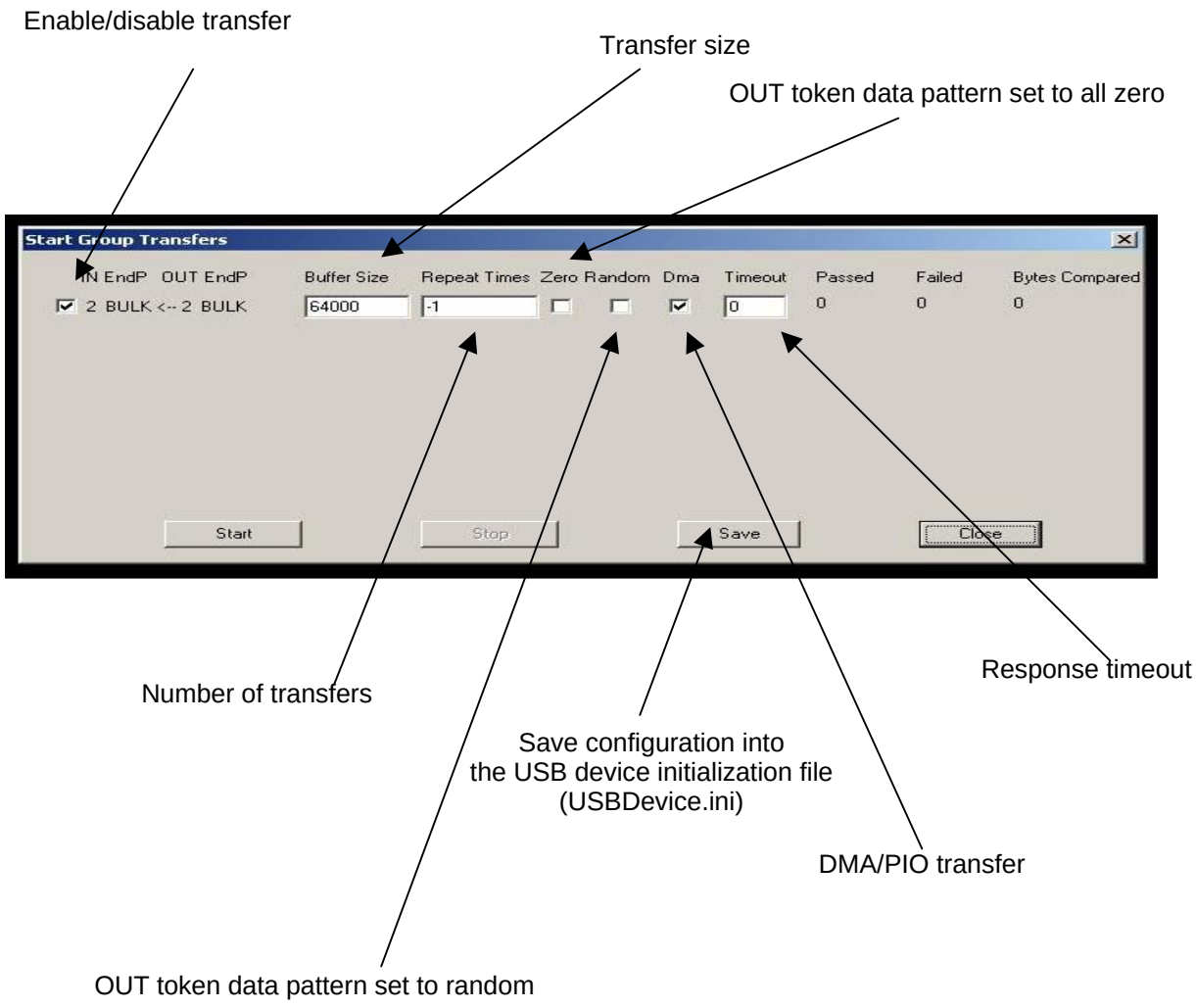


Fig 21. Start Group Transfers in detail

Table 2. Endpoint description

The test applet and the ISP1583 eval board support three test modes: loopback, print and scan. The firmware uses I/O accesses on this endpoint.

Pipe number	Endpoint type	Operations
2[1]	Bulk OUT	This pipe is defined as the bulk OUT pipe.
3*	Bulk IN	This pipe is defined as the bulk IN pipe.

[1] The pipe number is not the endpoint number. The value may vary if you have a different endpoint configuration.

Three test modes:

- Scan mode: The ISP1583 eval board acts like a scanner. It sends data packets to the host PC as fast as possible. This mode is used to evaluate the bulk IN transfer rate.
- Print mode: The ISP1583 eval board acts like a printer. It receives data packets from the host PC as fast as possible. This mode is used to evaluate the bulk OUT transfer rate.
- Loopback mode: In this mode, the ISP1583 eval board receives data packets on isochronous (or bulk) OUT endpoint and sends them back to the host PC on isochronous (or bulk) IN endpoint. This mode is used to test the data integrity of transfers.

The “Buffer Size” setting on the test applet is determined by the firmware and hardware ability of the eval board. For the split bus kit, the maximal size is limited to 64000 for bulk transfer.

The “Repeat Times” for loopback test controls the numbers of iterations of loopback, which is useful for debugging. Value “-1” means it is infinite.

Remark: At a particular time, you can only perform either scan or print. Both these operations cannot be performed simultaneously.

9. Schematics

9.1 ISP1583 split bus eval board

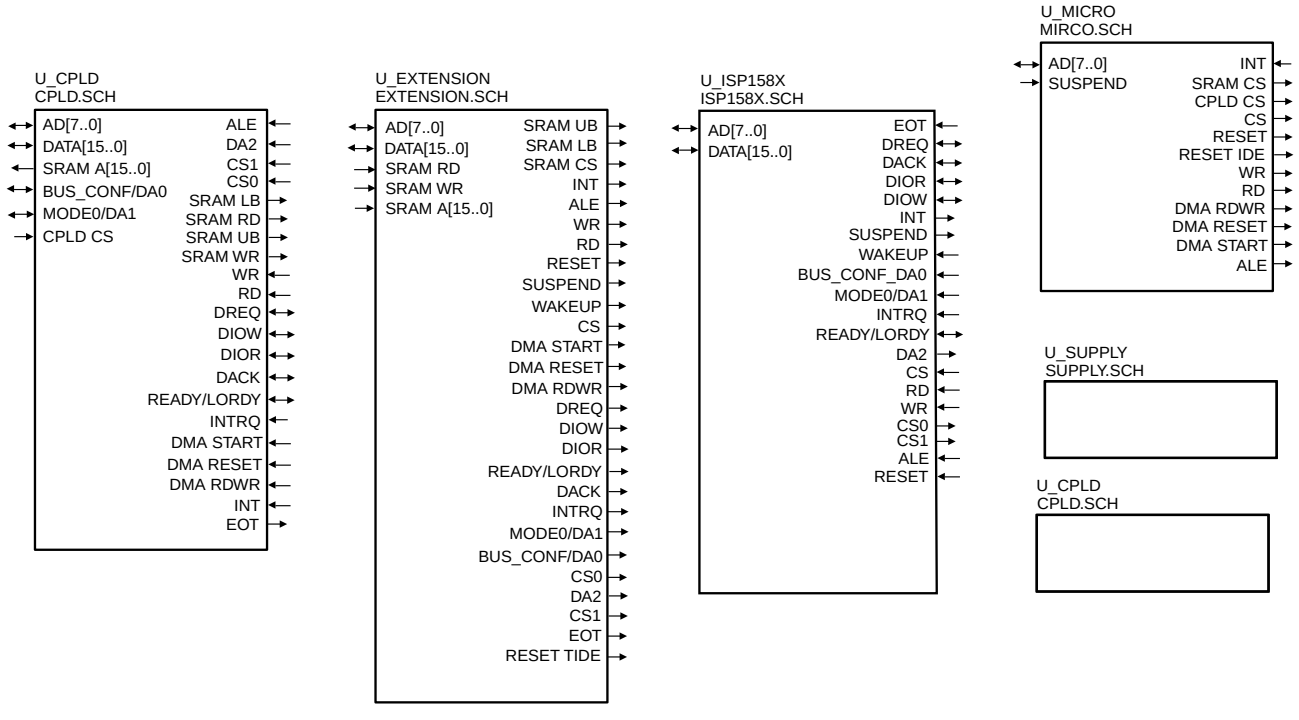


Fig 22. Block diagram of the eval kit

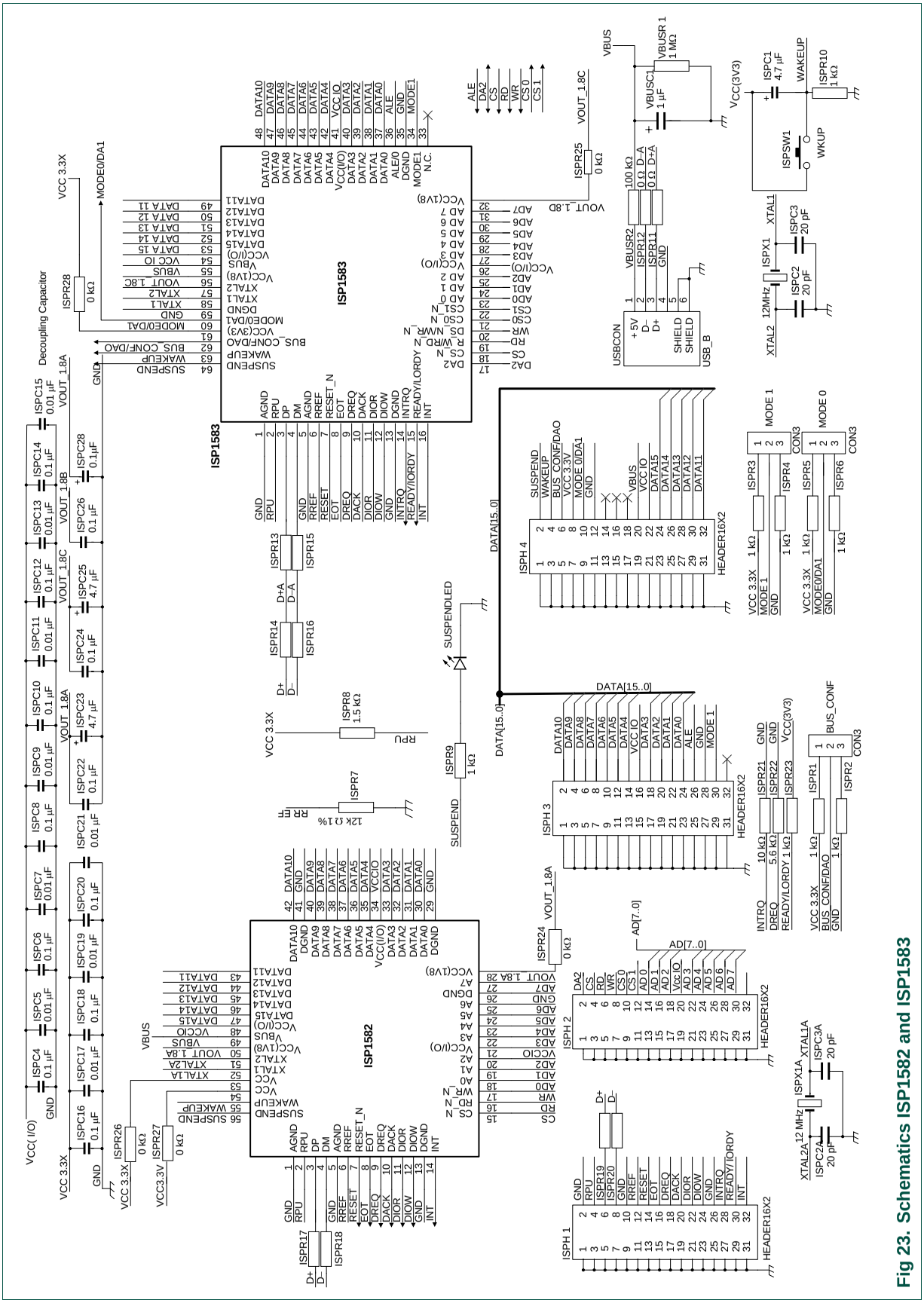


Fig 23. Schematics ISP1582 and ISP1583

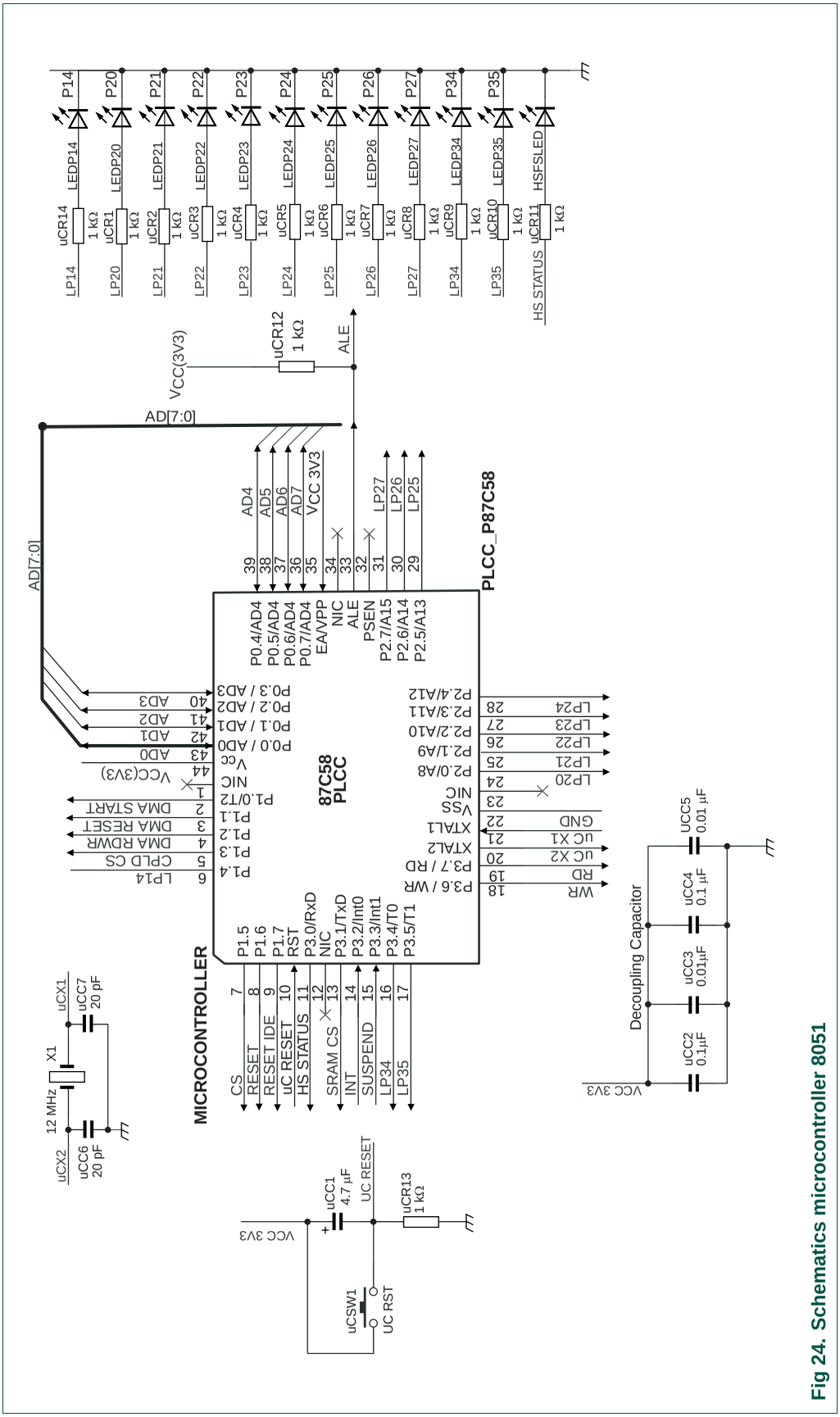


Fig 24. Schematics microcontroller 8051

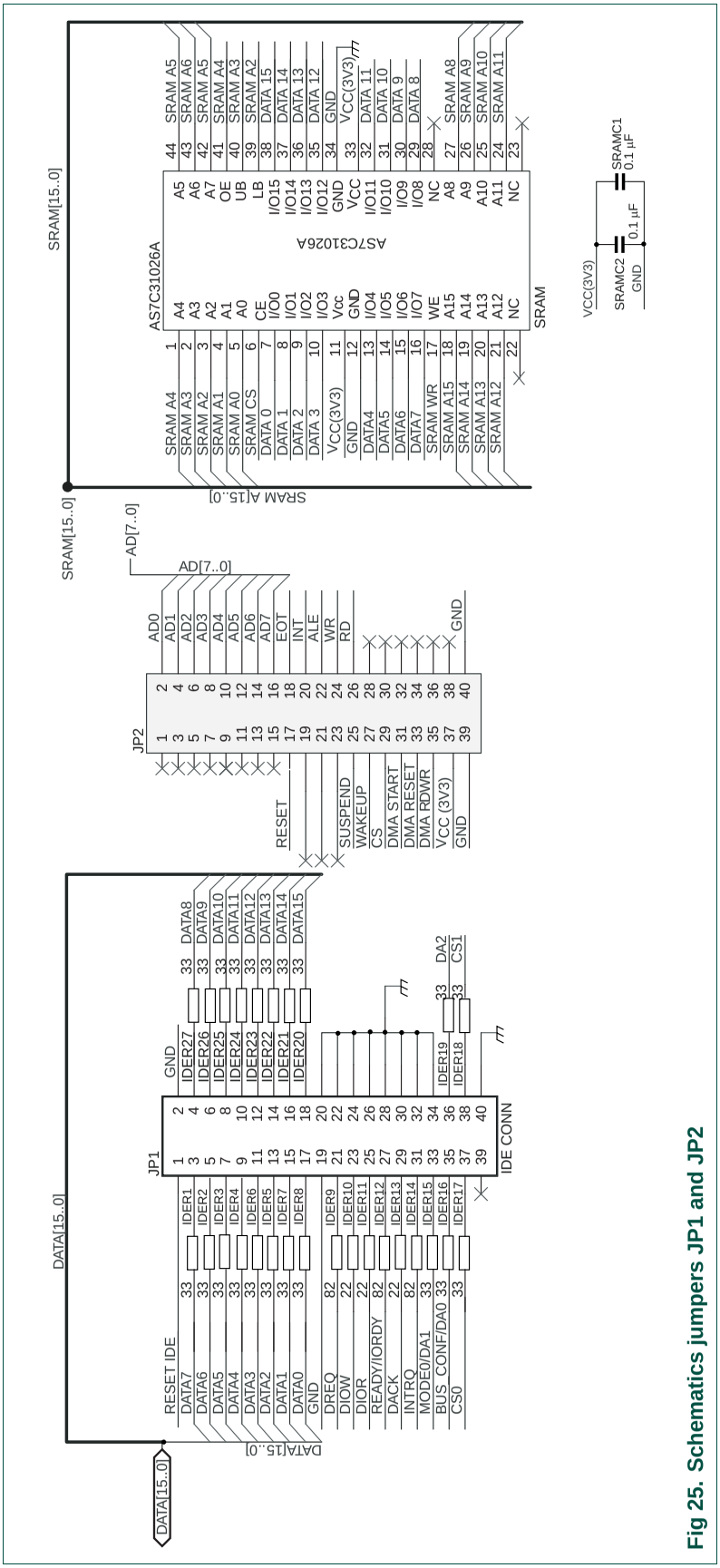


Fig 25. Schematics jumpers JP1 and JP2

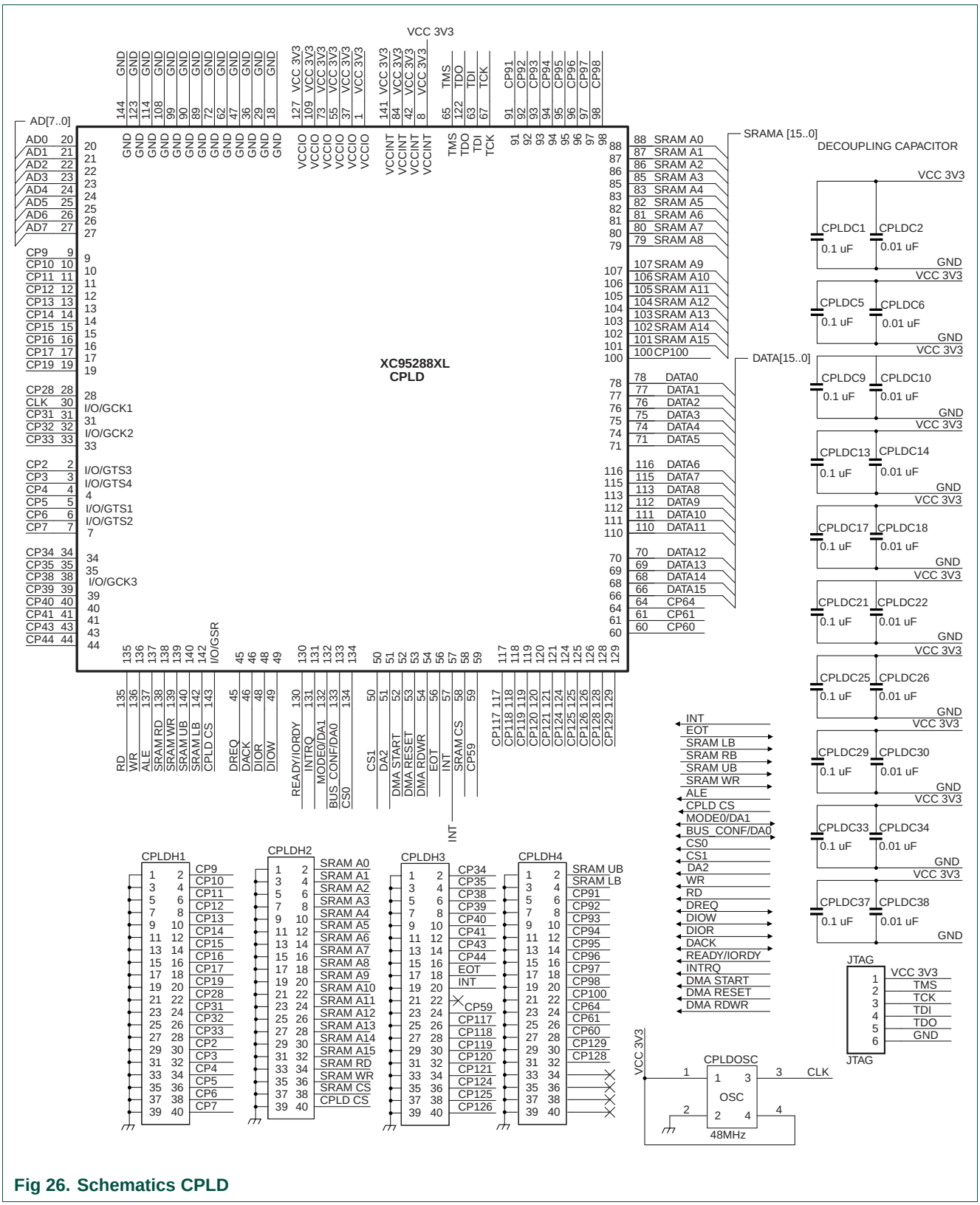


Fig 26. Schematics CPLD

10. Bill of material

10.1 ISP1583 split bus eval board

Table 3. Bill of material of the ISP1583 split bus eval board

Part Type	Designator	Footprint
0 kΩ	ISPR24 ISPR25 ISPR28 ISPR27 ISPR26	805
0 Ω	ISPR11 ISPR12	805
0.01 µF	CPLDC10 CPLDC6 CPLDC2 CPLDC18 CPLDC14 uCC3 uCC5 CPLDC38 CPLDC34 CPLDC30 CPLDC26 CPLDC22 SC1B SC1D SC1F SC1H	805
0.01 µF	SC17 SC15 SC19 ISPC5 SC21 SC7 SC5 SC9 SC13 SC11 ISPC19 ISPC17 ISPC21 ISPC9 ISPC7 ISPC11 ISPC15 ISPC13	603
0.1 µF	CPLDC1 PSC15 PSC5 uCC2 PSC9 uCC4 CPLDC5 CPLDC25 CPLDC21 CPLDC37 CPLDC33 CPLDC29 CPLDC17 CPLDC13 CPLDC9 SC1A PSC2 SRAMC1 SRAMC2 SC1I SC1C SC1G	805
0.1 µF	SC18 SC16 ISPC4 SC20 SC14 SC8 SC6 SC12 SC10 ISPC6 ISPC20 ISPC18 ISPC26 ISPC28 ISPC22 ISPC24 ISPC16 ISPC10 ISPC8 ISPC14 ISPC12 SC4	603
1 µF	PSC10 PSC13 PSC14 PSC4 PSC6 PSC8 VBUSC1	805
1 kΩ	uCR12 ISPR3 uCR11 uCR3 uCR2 uCR1 ISPR4 uCR5 uCR7 uCR6 ISPR5 ISPR6 uCR4 uCR14 uCR9 ISPR9 uCR10 uCR8 ISPR2 ISPR1 ISPR10 uCR13 ISPR23	805
1 MΩ	VBUSR1	-
1.5 kΩ	ISPR8	805
10 µF	PSC16 PSC3 PSC7	CASE B
10 kΩ	ISPR21	805
10 Ω	PSR2 PSR3 PSR1	805
12 kΩ 1 %	ISPR7	805
12 MHz	SX1 SX1B ISPX1 ISPX1A	XTAL-HC49/4H
12 MHz	X1	XTAL-CSM4A
1791	POWER5V POWERCORE	1791
1791/1762	POWERIO	1791
20 pF	SC3B SC2B ISPC2A SC3 SC2 ISPC3A uCC7 ISPC3 ISPC2 uCC6	805
22 Ω	IDER11 IDER10 IDER13	805
220 µF	PSC17 PSC12 PSC11	CASE D
33 Ω	IDER21 IDER22 IDER19 IDER18 IDER7 IDER3 IDER4 IDER5 IDER2 IDER8 IDER6 IDER1 IDER23 IDER17 IDER20 IDER16 IDER15 IDER27IDER26 IDER25 IDER24	805
4.7 µF	ISPC1 PSC1 uCC1	CASE A

Part Type	Designator	Footprint
4.7 µF	ISPC23 ISPC25	-
48 MHz	CPLDOSC	XTAL-SG615
5.6 kΩ	ISPR22	805
7 µH	PSL3 PSL1 PSL2	CDRH125
82 Ω	IDER9 IDER14 IDER12	805
CON3	MODE1 MODE0 BUS_CONF	CON3
CPLD	XC95288XL	TQFP_144
Diode	PSD2 PSD1	SMA
Header 16X2	H1 H2 H3 H4	HEADERB 16X2
Header 16X2	ISPH3 ISPH1 ISPH2 ISPH4	HEADER 16X2
IDE CONN	JP1	HEADERB 20X2
ISP1582	ISP1582	HVQFN56-SMT
ISP1582	SOCKET1582	SOCKET56
ISP1583	ISP1583	LQFP64-SMT
ISP1583	SOCKET1583	SOCKET64
JTAG	JTAG	HEADER 6
NDS8958	PSQ1 PSQ3 PSQ2	NDS8958
P14	LEDP14	LED
P20	LEDP20	LED
P21	LEDP21	LED
P22	LEDP22	LED
P23	LEDP23	LED
P24	LEDP24	LED
P25	LEDP25	LED
P26	LEDP26	LED
P27	LEDP27	LED
P34	LEDP34	LED
P35	LEDP35	LED
PLCC_P87C58	Microcontroller	PLCC44
POWER4	IDE_POW	POWER4
SRAM	AS7C31026A	TSOP44
uC RST	uCSW1	SW-TACT
USB_B	USBCON	USB_A
WKUP	ISPSW1	SW-TACT

11. VHDL code for the Xilinx XC95288XL DMA controller

The following code integrates the DMA slave, DMA master and PIO.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity topLevel is
    port (

--    Global Clock Input
      CLK:     in STD_LOGIC;
      DBG_CLK: out STD_LOGIC;
      DBG_STATE:  out STD_LOGIC_VECTOR(1 downto 0);

--    Microcontroller burst and mode access
         uC_AD:   inout STD_LOGIC_VECTOR (7 downto 0);
          CS:     in STD_LOGIC;
          ALE:    in STD_LOGIC;
          uC_WRITE: in STD_LOGIC;

--    DMA burst and mode selection
      WIDTH:      out STD_LOGIC;         -- debug pin
         MODE_OUT:   OUT STD_LOGIC_VECTOR (1 downto 0);-- debug pin

--    DMA control signal
-- ********************************************************************************
--  ISP158x_MS * DMA_START * DMA_RDWR * DMA_RESET *   Operation
--  ================================================================================
--      1             1           1            1    D14 Master Read, SRAM Read
--      1           1           0            1    D14 Master Write, SRAM Write
--      0           1           1            1    D14 Slave Read, SRAM Read
--      0           1           0            1    D14 Slave Write, SRAM Write
--      x           x           x            0    D14 DMA Reset
--      x             0           x            1    D14 DMA STOP
-- ********************************************************************************

           ISP1581_MS:       out STD_LOGIC;   -- debug pin
           DBG_DMA_START:    out STD_LOGIC;   -- debug pin
              DBG_DMA_RDWR:  out STD_LOGIC;   -- debug pin
           DBG_DMA_RESET:    out STD_LOGIC;   -- debug pin

         EOT:       OUT STD_LOGIC;

--    DMA bus and control signal

           DIOW:    inout STD_LOGIC;    -- dma control pin
           DIOR:    inout STD_LOGIC;    -- ""
           DACK:    inout STD_LOGIC;    -- ""
           DREQ:    inout STD_LOGIC;    --    ""

--    SRAM and flash control signal and bus
           SRAM_ADDR:     out STD_LOGIC_VECTOR(15 downto 0);
           SRAM_WR:    out STD_LOGIC;
```

```
          SRAM_RD:    out STD_LOGIC;
          SRAM_UB:    out STD_LOGIC;
          SRAM_LB:    out STD_LOGIC;
          SRAM_DATA_MSB: inout STD_LOGIC_VECTOR(7 downto 0);
         SRAM_DATA_LSB:  inout STD_LOGIC_VECTOR(7 downto 0);
          UC_READ:    in  STD_LOGIC

    );
end toplevel;



architecture rtl of toplevel is

constant asserted,read:     STD_LOGIC := '1';
constant deasserted,write:     STD_LOGIC := '0';

signal ms:          STD_LOGIC;
signal DMA_START:         STD_LOGIC;
signal DMA_RDWR:       STD_LOGIC;
signal DMA_RESET:         STD_LOGIC;
signal update_dma_sig:       STD_LOGIC;
signal dma_status:        STD_LOGIC_VECTOR(5 downto 0);

signal PIO:         STD_LOGIC;

signal MS_z:           STD_LOGIC;
signal MS_zz:          STD_LOGIC;
signal DMA_START_z:       STD_LOGIC;
signal DMA_START_zz:         STD_LOGIC;
signal DMA_RDWR_z:        STD_LOGIC;
signal DMA_RDWR_zz:       STD_LOGIC;
signal DMA_RESET_z:       STD_LOGIC;
signal DMA_RESET_zz:         STD_LOGIC;
signal DMA_RESET_zzz:     STD_LOGIC;

signal dma_set:        std_logic;
signal mode:           std_logic_vector(3 downto 0);
signal mode_z:      std_logic_vector(3 downto 0);
signal mode_zz:        std_logic_vector(3 downto 0);
signal transfer_counter:     std_logic_vector(15 downto 0);
signal transfer_set_lsb:     std_logic;
signal transfer_set_msb:     std_logic;


signal SRAM_ADDR_COUNTER:    integer range 0 to 1048575;
signal MSB_WR_FLAG,LSB_WR_FLAG: STD_LOGIC;
signal COUNTER_FLAG:         STD_LOGIC;
signal MSB_RD_FLAG,LSB_RD_FLAG: STD_LOGIC;
signal sram_wr_a1:        std_logic;
signal sram_wr_a2:        std_logic;
signal sram_wr_a3:        std_logic;
signal reset_address_flag:   std_logic;
signal sram_rd_a1:        std_logic;
signal sram_rd_a2:        std_logic;
signal sram_rd_a3:        std_logic;
```

```
signal sram_rd_a4:        std_logic;
signal sram_rd_a5:        std_logic;

type op_code is (S0,S1,S2,S3);
signal STATE:       op_code;
signal ADDR_COUNTER:         integer range 0 to 1048575;
signal m_ADDR_COUNTER:    integer range 0 to 1048575;
signal DACK_IN:         std_logic;
signal DREQ_IN:         STD_LOGIC;
signal DREQ_z:       STD_LOGIC;
signal DREQ_zz:         STD_LOGIC;

signal DREQ_OUT,INCREMENT:   STD_LOGIC;
signal COUNTER:         STD_LOGIC_VECTOR(15 downto 0);
signal dma_start_enable:     STD_LOGIC;
signal sram_a1:         std_logic;
signal sram_a2:         std_logic;
signal sram_a3:         std_logic;
signal sram_a4:         std_logic;
signal sram_a5:         std_logic;
signal sram_a6:         std_logic;
signal sram_a7:         std_logic;
signal DIOR_IN:         STD_LOGIC;
signal DIOR_z, DIOR_zz:      STD_LOGIC;
signal DIOW_z, DIOW_zz:      STD_LOGIC;
signal DACK_z, DACK_zz:      STD_LOGIC;
signal DIOW_IN:         STD_LOGIC;
signal dack_out:        std_logic;
signal PIO_reset:       std_logic;

signal read_flag:       std_logic;
signal write_flag:         std_logic;



begin

   EOT <= deasserted ;
   DREQ_z <= DREQ;

   PIO_reset <= DMA_STATUS(5);
   PIO <= DMA_STATUS(4);
   MS_z <= DMA_STATUS(3);
   DMA_START_z  <= DMA_STATUS(2);
   DMA_RDWR_z  <= DMA_STATUS(1);
   DMA_RESET_z <= DMA_STATUS(0);


   DBG_CLK  <= CLK;        -- debug output
   isp1581_ms <= ms;            -- debug output
   DBG_DMA_RESET <= DMA_RESET;     -- debug output
   DBG_DMA_START <= DMA_START;     -- debug output
   DBG_DMA_RDWR  <= DMA_RDWR;         -- debug output
   MODE_OUT <= MODE(2 downto 1);      -- debug output
   WIDTH <= MODE(3);         -- debug output
```

```
ALE_Process:
-- address decoding flag set in this process
process(CS, ALE)
begin

   if CS = deasserted  then
      if falling_edge(ALE) then

         case uC_AD is

         when "10010101" =>        -- 95 dma signal
            update_dma_sig <= asserted;
            DMA_SET <= deasserted;
            TRANSFER_SET_LSB <= deasserted;
            TRANSFER_SET_MSB <= deasserted;
            LSB_WR_FLAG <= deasserted;
            MSB_WR_FLAG <= deasserted;
            LSB_RD_FLAG <= deasserted;
            MSB_RD_FLAG <= deasserted;
         when "11110010" =>        -- F2 dma mode
            DMA_SET <= asserted;
            update_dma_sig <= deasserted;
            TRANSFER_SET_LSB <= deasserted;
            TRANSFER_SET_MSB <= deasserted;
            LSB_WR_FLAG <= deasserted;
            MSB_WR_FLAG <= deasserted;
            LSB_RD_FLAG <= deasserted;
            MSB_RD_FLAG <= deasserted;
         when "11110000"=>      -- F0 LSB of transfer count
            TRANSFER_SET_LSB <= asserted;
            TRANSFER_SET_MSB <= deasserted;
            update_dma_sig <= deasserted;
            DMA_SET <= deasserted;
            LSB_WR_FLAG <= deasserted;
            MSB_WR_FLAG <= deasserted;
            LSB_RD_FLAG <= deasserted;
            MSB_RD_FLAG <= deasserted;
         when "11110001" =>        -- F1 MSB of transfer count
            TRANSFER_SET_MSB <= asserted;
            TRANSFER_SET_LSB <= deasserted;
            update_dma_sig <= deasserted;
            DMA_SET <= deasserted;
            LSB_WR_FLAG <= deasserted;
            MSB_WR_FLAG <= deasserted;
            LSB_RD_FLAG <= deasserted;
            MSB_RD_FLAG <= deasserted;
         when "11110100"=>         -- F4 LSB of PIO WR
            LSB_WR_FLAG <= asserted;
            MSB_WR_FLAG <= deasserted;
            LSB_RD_FLAG <= deasserted;
            MSB_RD_FLAG <= deasserted;
            update_dma_sig <= deasserted;
            DMA_SET <= deasserted;
            TRANSFER_SET_LSB <= deasserted;
```

```
            TRANSFER_SET_MSB <= deasserted;
         when "11110101"=>        -- F5 MSB of PIO WR
            LSB_WR_FLAG <= deasserted;
            MSB_WR_FLAG <= asserted;
            LSB_RD_FLAG <= deasserted;
            MSB_RD_FLAG <= deasserted;
            update_dma_sig <= deasserted;
            DMA_SET <= deasserted;
            TRANSFER_SET_LSB <= deasserted;
            TRANSFER_SET_MSB <= deasserted;
         when "11110110"=>        -- F6 LSB of PIO RD
            LSB_WR_FLAG <= deasserted;
            MSB_WR_FLAG <= deasserted;
            LSB_RD_FLAG <= asserted;
            MSB_RD_FLAG <= deasserted;
            update_dma_sig <= deasserted;
            DMA_SET <= deasserted;
            TRANSFER_SET_LSB <= deasserted;
            TRANSFER_SET_MSB <= deasserted;
         when "11110111"=>        -- F7 MSB of PIO RD
            LSB_WR_FLAG <= deasserted;
            MSB_WR_FLAG <= deasserted;
            LSB_RD_FLAG <= deasserted;
            MSB_RD_FLAG <= asserted;
            update_dma_sig <= deasserted;
            DMA_SET <= deasserted;
            TRANSFER_SET_LSB <= deasserted;
            TRANSFER_SET_MSB <= deasserted;

         when others =>
--reset all variables used in this process

            update_dma_sig <= deasserted;
            DMA_SET <= deasserted;
            TRANSFER_SET_LSB <= deasserted;
            TRANSFER_SET_MSB <= deasserted;
            LSB_WR_FLAG <= deasserted;
            MSB_WR_FLAG <= deasserted;
            LSB_RD_FLAG <= deasserted;
            MSB_RD_FLAG <= deasserted;

         end case;
      end if;

   else
      -- reset all variables used in this process
      update_dma_sig <= deasserted;
      DMA_SET <= deasserted;
      TRANSFER_SET_LSB <= deasserted;
      TRANSFER_SET_MSB <= deasserted;
      LSB_WR_FLAG <= deasserted;
      MSB_WR_FLAG <= deasserted;
      LSB_RD_FLAG <= deasserted;
      MSB_RD_FLAG <= deasserted;

   end if;
```

```
end process;

update_DMA_parameters:
--decoding transfer counter, dma mode and dma signal(DMA Reset, DMA RdWr, DMA Start, -
--Ms)

process(CS, uC_WRITE)
begin

      if CS = deasserted then


         if rising_edge(uC_WRITE) then

            if update_dma_sig = asserted then
               dma_status <= uC_AD(5 downto 0);
            end if;
            if DMA_SET = asserted then
               MODE_z <= uC_AD(3 downto 0);
            end if;
            if TRANSFER_SET_LSB = asserted then
               TRANSFER_COUNTER(7 downto 0) <= uC_AD;
            end if;

            if TRANSFER_SET_MSB = asserted then
               TRANSFER_COUNTER(15 downto 8) <= uC_AD;
            end if;

            if DMA_RESET = deasserted  then
               TRANSFER_COUNTER <= "0000000000000000";
            end if;
         end if;
      end if;

end process;


update_addr:
-- update sram address
process(pio, sram_addr_counter, addr_counter, m_addr_counter, ms)
begin
   if pio = asserted then
      SRAM_ADDR <= CONV_STD_LOGIC_VECTOR(SRAM_ADDR_COUNTER,16);

   elsif ms = asserted then
      SRAM_ADDR <= CONV_STD_LOGIC_VECTOR(m_ADDR_COUNTER,16);

   else
      SRAM_ADDR <= CONV_STD_LOGIC_VECTOR(ADDR_COUNTER,16);

   end if;

end process;


Read_Process:
```

```
-- SRAM read and write
Process (pio, CS, MSB_RD_FLAG, LSB_RD_FLAG, uC_WRITE, MSB_WR_FLAG, LSB_WR_FLAG, clk,
ale, sram_a1)
variable DELAY: STD_LOGIC;
begin

      if falling_edge(CLK) then
         if pio = asserted and CS = deasserted then
         if MSB_WR_FLAG=asserted then

            SRAM_RD <= asserted;
            SRAM_LB <= asserted;
            sram_wr_a1 <= uC_WRITE;
            sram_wr_a2 <= sram_wr_a1;
            sram_wr_a3 <= sram_wr_a2;
            SRAM_UB <= (sram_wr_a3 OR (sram_wr_a2));
            SRAM_WR <= (sram_wr_a3 OR (sram_wr_a2));

         elsif LSB_WR_FLAG=asserted then

            SRAM_RD <= asserted;
            SRAM_UB <= asserted;
            sram_wr_a1 <= uC_WRITE;
            sram_wr_a2 <= sram_wr_a1;
            sram_wr_a3 <= sram_wr_a2;
            SRAM_LB <= (sram_wr_a3 OR (sram_wr_a2));
            SRAM_WR <= (sram_wr_a3 OR (sram_wr_a2));

         elsif MSB_RD_FLAG = asserted then
            SRAM_WR <= asserted;
            sram_rd_a1 <= uC_READ;
            sram_rd_a2 <= sram_rd_a1;
            sram_rd_a3 <= sram_rd_a2;
            sram_rd_a4 <= sram_rd_a3;
            sram_rd_a5 <= sram_rd_a4;
            SRAM_LB <= asserted;
            SRAM_UB <= (sram_rd_a4 OR (not(sram_rd_a5)));
            SRAM_RD <= (sram_rd_a4 OR (not(sram_rd_a5)));

         elsif LSB_RD_FLAG=asserted then
            SRAM_WR <= asserted;
            sram_rd_a1 <= uC_READ;
            sram_rd_a2 <= sram_rd_a1;
            sram_rd_a3 <= sram_rd_a2;
            sram_rd_a4 <= sram_rd_a3;
            sram_rd_a5 <= sram_rd_a4;
            SRAM_LB <= (sram_rd_a4 OR (not (sram_rd_a5)));
            sram_rd <= (sram_rd_a4 OR (not(sram_rd_a5)));
            SRAM_UB <= asserted;

         else
            SRAM_LB <= asserted;
            SRAM_UB <= asserted;
            SRAM_WR <= asserted;
            SRAM_RD <= asserted;
            sram_rd_a1 <= uC_READ;
```

```
      sram_rd_a2 <= sram_rd_a1;
      sram_rd_a3 <= sram_rd_a2;
      sram_rd_a4 <= sram_rd_a3;
      sram_rd_a5 <= sram_rd_a4;

   end if;

   end if;

if ms = deasserted and pio = deasserted then
   SRAM_LB <= deasserted;
   SRAM_UB <= deasserted;

   if DMA_RESET = deasserted then
      SRAM_WR <= asserted;
      SRAM_RD <= asserted;
      ADDR_COUNTER <= 0;
      DELAY := asserted;
   end if;

   if DMA_RDWR = WRITE then

      case STATE is

      when S2 =>

         if DREQ_IN = asserted then
            SRAM_RD <= deasserted;
         else
            SRAM_RD <= asserted;
         end if;

      when S3 =>

         SRAM_RD <= asserted;

      when S0 =>

         if DACK_out = deasserted then
            ADDR_COUNTER <= ADDR_COUNTER + 1;
         else
            ADDR_COUNTER <= ADDR_COUNTER;
         end if;

      when others => NULL;

      end case;

   else

      case STATE is

      when S0 =>

         if DACK_out = deasserted then
            SRAM_WR <= deasserted;
```

```
            else
               SRAM_WR <= asserted;
               DELAY := asserted;

            end if;

         when S1 =>

            SRAM_WR <= asserted;

         when S2 =>

            if DELAY = deasserted then
               ADDR_COUNTER <= ADDR_COUNTER + 1;
            else
               ADDR_COUNTER <= ADDR_COUNTER;
               DELAY := deasserted;
            end if;

         when others => NULL;

         end case;
      end if;


end if;

if ms = asserted and pio = deasserted then
   SRAM_LB <= deasserted;
   SRAM_UB <= deasserted;
   if DMA_RESET = deasserted  then

      SRAM_RD <= asserted;
      SRAM_WR <= asserted;
      INCREMENT <= asserted;

   else

      case MODE (2 downto 1) is

      when "00" =>


         if DMA_RDWR = WRITE then
            if diow_in = deasserted then
               increment <= deasserted;
            else
               increment <= asserted;
            end if;

            sram_a1 <= diow_in;
            sram_a2 <= sram_a1;
            sram_a3 <= sram_a2;
            sram_a4 <= sram_a3;
            sram_a5 <= sram_a4;
            sram_a6 <= sram_a5;
```

```
      sram_a7 <= sram_a6;
      sram_wr <= ((NOT sram_a7) OR sram_a6);


   else
      if dior_in = deasserted then
         increment <= deasserted;
      else
         increment <= asserted;
      end if;
      sram_a1 <= dior_in;
      sram_a2 <= sram_a1;
      sram_a3 <= sram_a2;
      sram_a4 <= sram_a3;
      sram_a5 <= sram_a4;
      sram_a6 <= sram_a5;
      sram_a7 <= sram_a6;
      sram_rd <= ((NOT sram_a7) OR sram_a6);


   end if;


when "01" =>

   if DMA_RDWR = WRITE then
      if dack_in = deasserted then
         increment <= deasserted;
      else
         increment <= asserted;
      end if;

      sram_a1 <= dack_in;
      sram_a2 <= sram_a1;
      sram_a3 <= sram_a2;
      sram_a4 <= sram_a3;
      sram_a5 <= sram_a4;
      sram_a6 <= sram_a5;
      sram_a7 <= sram_a6;
      sram_wr <= ((NOT sram_a7) OR sram_a6);


   else

      if dior_in = deasserted then
         increment <= deasserted;
      else
         increment <= asserted;
      end if;

      sram_a1 <= dior_in;
      sram_a2 <= sram_a1;
      sram_a3 <= sram_a2;
      sram_a4 <= sram_a3;
      sram_a5 <= sram_a4;
      sram_a6 <= sram_a5;
```

```
                  sram_a7 <= sram_a6;
                  sram_rd <= ((NOT sram_a7) OR sram_a6);

               end if;

            when "10" =>

               if DMA_RDWR = WRITE then

                                                sram_a1 <= dack_in;
                  sram_a2 <= sram_a1;
                  sram_a3 <= sram_a2;
                  sram_a4 <= sram_a3;
                  sram_a5 <= sram_a4;
                  sram_a6 <= sram_a5;
                  sram_a7 <= sram_a6;
                  sram_wr <= ((NOT sram_a7) OR sram_a6);


               else
                  sram_a1 <= dack_in;
                  sram_a2 <= sram_a1;
                  sram_a3 <= sram_a2;
                  sram_a4 <= sram_a3;
                  sram_a5 <= sram_a4;
                  sram_a6 <= sram_a5;
                  sram_a7 <= sram_a6;
                  sram_rd <= ((NOT sram_a7) OR sram_a6);

               end if;

               if dack_in = deasserted then
                  increment <= deasserted;
               else
                  increment <= asserted;
               end if;


            when others => NULL;

            end case;
         end if;
      end if;

      if DMA_RESET = deasserted then

         ADDR_COUNTER <= 0;
         DELAY := asserted;
         INCREMENT <= asserted;
      end if;

   end if;

end process;
```

```
WRITE_READ_Process:
-- direction control of AD to read and write mode

process(pio, CS, MSB_WR_FLAG, LSB_WR_FLAG, SRAM_DATA_MSB, SRAM_DATA_LSB, ALE, uC_AD,
   MSB_RD_FLAG, LSB_RD_FLAG, uC_READ, CLK)
begin
   if pio = deasserted then
      uC_AD <= "ZZZZZZZZ";
      SRAM_DATA_MSB <= "ZZZZZZZZ";
      SRAM_DATA_LSB <= "ZZZZZZZZ";

   else
      if CS = deasserted  then

         if falling_edge(CLK) then

            if MSB_WR_FLAG=asserted and uC_WRITE = deasserted then
               SRAM_DATA_MSB<= uC_AD;
               SRAM_DATA_LSB<="ZZZZZZZZ";
               uC_AD <= "ZZZZZZZZ";
            elsif LSB_WR_FLAG=asserted and uC_WRITE = deasserted then
               SRAM_DATA_LSB<=uC_AD;
               SRAM_DATA_MSB<="ZZZZZZZZ";
               uC_AD <= "ZZZZZZZZ";
            elsif MSB_RD_FLAG=asserted and uC_READ=deasserted then
               uC_AD <= sram_DATA_MSB;
               SRAM_DATA_MSB <= "ZZZZZZZZ";
               SRAM_DATA_LSB <= "ZZZZZZZZ";
            elsif LSB_RD_FLAG=asserted and uC_READ=deasserted then
               uC_AD <= sram_DATA_LSB;
               SRAM_DATA_LSB <= "ZZZZZZZZ";
               SRAM_DATA_MSB <= "ZZZZZZZZ";
            else
               uC_AD <= "ZZZZZZZZ";
               SRAM_DATA_MSB <= "ZZZZZZZZ";
               SRAM_DATA_LSB <= "ZZZZZZZZ";
            end if;

         end if;
      end if;

   end if;

end process;


ADDRESS_PROCESS:
--PIO address logic and address reset logic
--the address is reset whenever there is a change from read to write or vice versa

process(pio, CS, ALE, uC_AD, uC_WRITE, reset_address_flag, sram_addr_counter,
counter_flag,
   uC_READ,read_flag,write_flag)

begin
   if pio = asserted then
```

```
if CS = deasserted  then
   if falling_edge(ALE) then

      case uC_AD is


      when "11110100"=>           -- F4 LSB WR
         if COUNTER_FLAG = deasserted then
            SRAM_ADDR_COUNTER <= SRAM_ADDR_COUNTER;
            COUNTER_FLAG<=deasserted;

         else
            SRAM_ADDR_COUNTER <= SRAM_ADDR_COUNTER + 1;
            COUNTER_FLAG<=deasserted;
         end if;


      when "11110110"=>           -- F6 LSB RD
         if COUNTER_FLAG = deasserted then
            SRAM_ADDR_COUNTER <= SRAM_ADDR_COUNTER;
            COUNTER_FLAG<=deasserted;

         else
            SRAM_ADDR_COUNTER <= SRAM_ADDR_COUNTER + 1;
            COUNTER_FLAG<=deasserted;
         end if;


      when "11110101"=>           -- F5 MSB WR
         COUNTER_FLAG<=asserted;

      when "11110111"=>           -- F7 MSB RD
         COUNTER_FLAG<=asserted;

      when others =>
         SRAM_ADDR_COUNTER <= SRAM_ADDR_COUNTER;

      end case;

   end if;


   if uC_READ = deasserted then
      read_flag <= deasserted;
   end if;

   if uC_WRITE = deasserted then
      write_flag <= deasserted;
   end if;

   if read_flag = deasserted and uC_WRITE = deasserted then
      -- reset address
      SRAM_ADDR_COUNTER<=0;
      COUNTER_FLAG<=deasserted;
      read_flag <= asserted;
   elsif write_flag = deasserted and uC_READ = deasserted then
```

```
            -- reset address
            SRAM_ADDR_COUNTER<=0;
            COUNTER_FLAG<=deasserted;
            write_flag <= asserted;
         end if;
      end if;
   end if;
end process;

STATE_MACHINE_Process:

process(DMA_RESET, state, clk, ms, pio)
begin
   if ms = deasserted then
      if DMA_RESET = deasserted then
         STATE <= S0;
         DBG_STATE <= "00";

      elsif falling_edge(CLK) then

          if ((DREQ_IN = asserted) or
            (DREQ_IN = deasserted and DACK_out = deasserted) or
            (STATE /= S0)) then

            case  STATE is

            when S0 =>
               STATE <= S1;
               DBG_STATE <= "01";
            when S1 =>
               STATE <= S2;
               DBG_STATE <= "10";
            when S2 =>
               STATE <= S3;
               DBG_STATE <= "11";
            when S3 =>
               STATE <= S0;
               DBG_STATE <= "00";
            when others =>
               STATE <= S0;
               DBG_STATE <= "00";

            end case;
         else
            STATE <= S0;
         end if;
      end if;
   end if;
end process;


DACK_Process:
-- cpld master dack logic
process(DMA_RESET, CLK, ms, pio)
begin
   if ms = deasserted  and pio = deasserted then
```

```
if DMA_RESET = deasserted then

   DACK_OUT <= asserted;
   DACK <= asserted;

elsif falling_edge(CLK) then

   case MODE(2 downto 1) is

   when "00" =>

      if STATE = S2 then
         DACK_out <= not DREQ_IN;
      else
         DACK_out <= DACK_out;
      end if;

   when others =>

      if DMA_RDWR = READ and MODE(2 downto 1) = "01" then

         if STATE = S2 then
            DACK_out <= not DREQ_IN;
         else
            DACK_out <= DACK_out;
         end if;

      else

         case STATE is

         when S3 =>

            if DREQ_IN = asserted then
               DACK_out <= deasserted;
            else
               DACK_out <= asserted;
            end if;

         when S1 =>

            if DREQ_IN = asserted then

               DACK_out <= asserted;

            elsif DREQ_IN = deasserted then

               DACK_out <= deasserted;

            end if;

         when S2 =>

            if DREQ_IN = deasserted then

               DACK_out <= asserted;
```

```
                else
                   DACK_out <= DACK_out;

                end if;

             when others => NULL;

             end case;
          end if;
       end case;
       -- delay the output by one clock
       DACK <= DACK_out;
    end if;

  else
     dack <= 'Z';

  end if;
end process;



DIOR_DIOW_Process:
-- cpld master diow and dior logic

process(DMA_RESET, CLK, ms, pio)
begin
  if ms = deasserted and pio = deasserted then

     if DMA_RESET = deasserted then

        DIOR <= asserted;
        DIOW <= asserted;

     elsif falling_edge(CLK) then

        if DACK_out = deasserted then

        case MODE(2 downto 1) is

        when "00" =>

           case STATE is

           when S3 =>

              if DMA_RDWR = WRITE then
                 DIOW <= deasserted;
                 DIOR <= asserted;
              else
                 DIOW <= asserted;
                 DIOR <= deasserted;
              end if;

           when S1 =>
```

```
      if DMA_RDWR = WRITE then
         DIOW <= asserted;
         DIOR <= asserted;
      else
         DIOW <= asserted;
         DIOR <= asserted;
      end if;

   when S0 =>

      if DMA_RDWR = WRITE then
         DIOW <= deasserted;
         DIOR <= asserted;
      else
         DIOW <= asserted;
         DIOR <= deasserted;
      end if;

   when S2 =>

      if DMA_RDWR = WRITE then
         DIOW <= asserted;
         DIOR <= asserted;
      else
         DIOW <= asserted;
         DIOR <= asserted;
      end if;

   when others => NULL;

   end case;

when "01" =>
   case STATE is

   when S3 =>

      if DMA_RDWR = WRITE then
         DIOW <= asserted;
         DIOR <= asserted;
      else
         DIOW <= asserted;
         DIOR <= deasserted;
      end if;

   when S1 =>

      if DMA_RDWR = WRITE then
         DIOW <= asserted;
         DIOR <= asserted;
      else
         DIOW <= asserted;
         DIOR <= asserted;
      end if;
```

```
            when S0 =>

               if DMA_RDWR = WRITE then
                  DIOW <= asserted;
                  DIOR <= asserted;
               else
                  DIOW <= asserted;
                  DIOR <= deasserted;
               end if;

            when S2 =>

               if DMA_RDWR = WRITE then
                  DIOW <= asserted;
                  DIOR <= asserted;
               else
                  DIOW <= asserted;
                  DIOR <= asserted;
               end if;

            when others => NULL;

            end case;

         when others => NULL;

         end case;

         end if;

      end if;

   else
      dior <= 'Z';
      diow <= 'Z';

   end if;
end process;


SRAM_ADDR_Process:
-- master mode address logic

process(DMA_RESET, INCREMENT, ms, pio)
begin
   if ms = asserted and pio = deasserted then
      if DMA_RESET = deasserted then

         m_ADDR_COUNTER <= 0;

      elsif rising_edge(INCREMENT) then
         if dma_start = asserted then
            m_ADDR_COUNTER <= m_ADDR_COUNTER + 1;
         end if;

      end if;
```

```
   end if;
end process;


DMA_COUNTER:
-- master mode counter
process(DMA_RESET, dreq_out, INCREMENT, ms, pio)
begin

   if ms = asserted and pio = deasserted then
      if DMA_RESET = deasserted or DREQ_OUT = deasserted  then
         COUNTER <= "0000000000000000";

      elsif falling_edge(INCREMENT) then
         if MODE(3) = asserted then
            COUNTER <= COUNTER + 2;
         else
            COUNTER <= COUNTER + 1;
         end if;
      end if;
   end if;

end process;

DREQ_Process:
-- master mode dreq logic

process(DMA_RESET, MS, CLK, pio)
begin
   if ms = asserted and pio = deasserted then
      if DMA_RESET = deasserted  then

         DREQ <= deasserted;
         DREQ_OUT <= deasserted;
         dma_start_enable <= asserted;


      elsif falling_edge(CLK)

         if ((COUNTER(15 downto 0) >= TRANSFER_COUNTER(15 downto 0)) and
             COUNTER(15 downto 0) /= "0000000000000000") then

            DREQ <= deasserted;
            DREQ_OUT <= deasserted;
            dma_start_enable <= deasserted;
         elsif DMA_START = asserted and dma_start_enable = asserted then

            DREQ <= asserted;
            DREQ_OUT <= asserted;
         elsif DMA_START = deasserted then
            dma_start_enable <= asserted;

         end if;

      end if;
   else
```

```
      dreq <= 'Z';
      DREQ_OUT <= deasserted;
      dma_start_enable <= asserted;

   end if;
end process;

--------    synchronising the inputs         --------

sync_process1:
process(CLK, DMA_RESET_z)
begin
   if DMA_RESET_z = deasserted then
      DMA_RESET_zz <= deasserted;
      DMA_RESET_zzz <= deasserted;
   elsif falling_edge(CLK) then

      MS_zz <= MS_z;
      MS <= MS_zz;

      DMA_START_zz  <= DMA_START_z;
      DMA_START  <= DMA_START_zz;

      DMA_RDWR_zz  <= DMA_RDWR_z ;
      DMA_RDWR  <= DMA_RDWR_zz ;

      DMA_RESET_zz <= DMA_RESET_z ;
      DMA_RESET_zzz <= DMA_RESET_zz ;

   end if;
end process;

   -- sync dma_reset
   DMA_RESET <= DMA_RESET_zzz and DMA_RESET_z;


sync_PROCESS2:
process(CLK, DMA_RESET)
begin
   if DMA_RESET = deasserted then
      DIOR_z <= deasserted;
      DIOR_zz <= deasserted;
      DIOW_z <= deasserted;
      DIOW_zz <= deasserted;
      DACK_z <= deasserted;
      DACK_zz <= deasserted;
      DREQ_zz <= deasserted;
      DREQ_IN <= deasserted;
   elsif falling_edge(CLK) then
      MODE_zz <= MODE_z;
      MODE <= MODE_zz;
      DIOR_z <= DIOR;
      DIOR_zz <= DIOR_z;
      DIOW_z <= DIOW;
      DIOW_zz <= DIOW_z;
```

```
      DACK_z <= DACK;
      DACK_zz <= DACK_z;
      DREQ_zz <= DREQ_z;
      DREQ_IN <= DREQ_zz;
   end if;
end process;

   DIOR_IN <= DIOR AND DIOR_zz;
   DIOW_IN <= DIOW AND DIOW_zz;
   DACK_IN <= DACK AND DACK_zz;

end rtl;
```

12. References

- ISP1582 Hi-Speed Universal Serial Bus Peripheral Controller data sheet
- ISP1583 Hi-Speed Universal Serial Bus Peripheral Controller data sheet

13. Legal information

13.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

13.2 Disclaimers

General — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of a NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is for the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

13.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

14. Contents

For more information, please visit: http://www.nxp.com.
For sales office addresses, email to: salesaddresses@nxp.com.

Date of release: 24 April 2007
Document identifier: UM10038_4