XStend Board V1.2 Manual

XESS Corporation

Limited Warranty

X Engineering Software Systems Corp. (XESS) warrants that the Product, in the course of its normal use, will be free from defects in material and workmanship for a period of one (1) year and will conform to XESS's specification therefor. This limited warranty shall commence on the date appearing on your purchase receipt.

XESS shall have no liability for any Product returned if XESS determines that the asserted defect a) is not present, b) cannot reasonably be rectified because of damage occurring before XESS receives the Product, or c) is attributable to misuse, improper installation, alteration, accident or mishandling while in your possession. Subject to the limitations specified above, your sole and exclusive warranty shall be, during the period of warranty specified above and at XESS's option, the repair or replacement of the product. The foregoing warranty of XESS shall extend to repaired or replaced Products for the balance of the applicable period of the original warranty or thirty (30) days from the date of shipment of a repaired or replaced Product, whichever is longer.

THE FOREGOING LIMITED WARRANTY IS XESS'S SOLE WARRANTY AND IS APPLICABLE ONLY TO PRODUCTS SOLD AS NEW. THE REMEDIES PROVIDED HEREIN ARE IN LIEU OF a) ANY AND ALL OTHER REMEDIES AND WARRANTIES, WHETHER EXPRESSED OR IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, AND b) ANY AND ALL OBLIGATIONS AND LIABILITIES OF XESS FOR DAMAGES INCLUDING, BUT NOT LIMITED TO ACCIDENTAL, CONSEQUENTIAL, OR SPECIAL DAMAGES, OR ANY FINANCIAL LOSS, LOST PROFITS OR EXPENSES, OR LOST DATA ARISING OUT OF OR IN CONNECTION WITH THE PURCHASE, USE OR PERFORMANCE OF THE PRODUCT, EVEN IF XESS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

In the United States, some statutes do not allow exclusion or limitations of incidental or consequential damages, so the limitations above may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Table of Contents

Getting Help!

If you follow the instructions in this manual and you encounter problems, here are some places to get help:

- If you can't get the XStend Board hardware to work, send an e-mail message describing your problem to `fpga-bugs@xess.com` or check our web site at `http://www.xess.com/FPGA`.

- If you can't get your XILINX software tools installed properly, send an e-mail message describing your problem to `hotline@xilinx.com` or check their web site at `http://www.xilinx.com/support/searchtd.htm`.

1 XStend Overview

The XS40 and XS95 Boards offer a flexible, low-cost method of prototyping FPGA and CPLD designs. However, their small physical size limits the amount of support circuitry they can hold. The XStend Board removes this limitation by providing additional support circuitry that the XS40 and XS95 Boards can access through their breadboard interfaces.

The XStend Board contains resources that extend the range of applications of the XS Boards into three areas:

- The pushbuttons, DIP switches, LEDs, and prototyping area are useful for basic lab experiments. These features in combination with the XS Boards replicates the functionality of the older HW/UW FPGABOARD.

- The VGA monitor interface, PS/2 keyboard/mouse interface, and static RAM let the XS Boards be used in video and computing experiments.

- The stereo codec and dual-channel analog input/output circuitry are useful for processing of audio signals in combination with DSP circuits synthesized with XILINX's CORE generation software.

2 XStend Board Features

The XStend Board extends the capabilities of the XS40 and XS95 Boards by providing:

- mounting sockets for both an XS40 and an XS95 Board;
- additional bargraph LED and LED digits;
- pushbutton and DIP switches;
- an interface to VGA monitors;
- an interface to a PS/2-style keyboard or mouse;
- an additional 64 Kbytes of static RAM (optional);
- a stereo codec with left/right input and output channels.
- an interface to the XILINX Xchecker cable;
- a 2.75"×3.5" prototyping area with selectable 3.3V or 5V supply;
- a 42×2 header connector for add-on daughterboards.

These resources are shown in the simplified view of the XStend Board (Figure 1). Each of these resources will be described below.

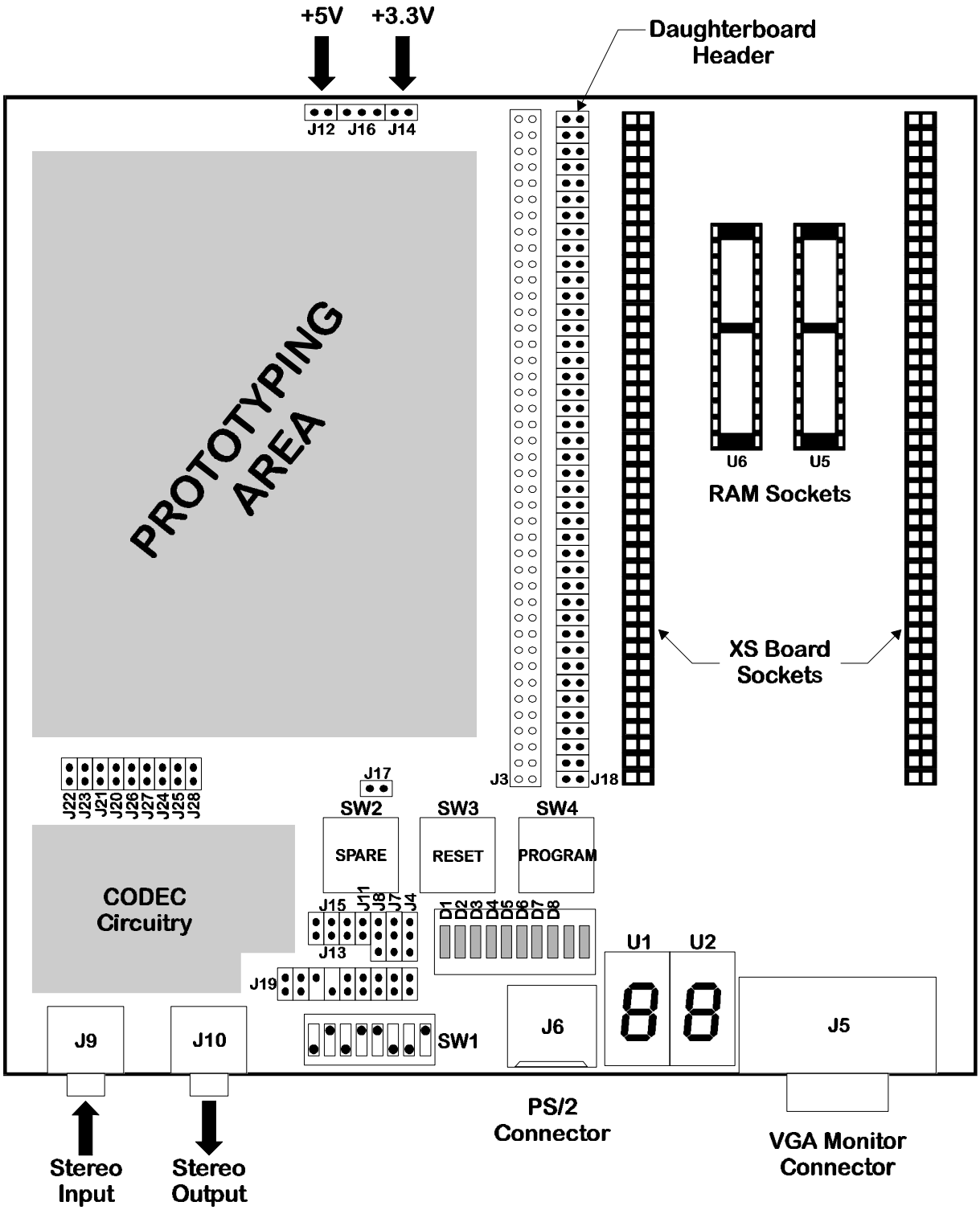


Figure 1: XStend Board layout.

2.1 XS40/XS95 Board Mounting Area

An XS40 or XS95 Board is mounted on the XStend Board using the XS Board mounting sockets. These sockets mate with the breadboard interface pins of the XS Boards to give them access to all the resources of the XStend Board. To use an XS40 Board with the XStend Board, insert it into the right-most columns of the socket strips. When using an XS95 Board, you should insert it into the left-most columns of the sockets. There are markings on the XStend Board to indicate the appropriate column for each type of XS Board.

If the XS Board is connected to a power supply through jack J9, then its power regulation circuitry will supply VCC and GND to the XStend Board through the mounting sockets. XS40 Boards with 3.3V FPGAs will supply both 3.3V and 5V to the XStend Board, while XS40 Boards with 5V FPGAs and XS95 Boards will supply only 5V.

☛ ***Warning: Version 1.0 of the XS40 Board with a 3.3V XC4000XL FPGA will not work with the XStend Board because it supplies 3.3V but no 5V! You must replace the XC4000XL FPGA with an XC4000E FPGA and remove the J8 jumper to switch the board to 5V operation.***

External voltage supplies can also be used with the XStend Board. A 5V power supply can be connected to header J12 and a 3.3V supply can be attached to header J14 as shown in Figure 2. These supplies will power the attached XS Board as well as the XStend electronics.

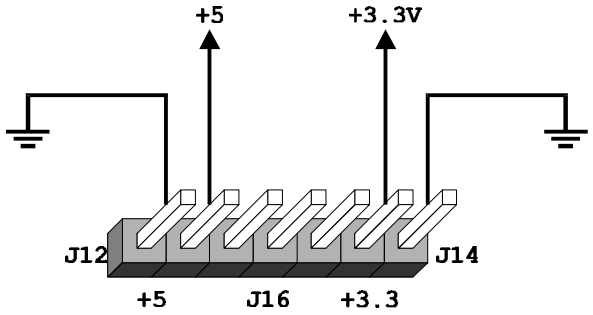


Figure 2: Connection of external power supplies to the XStend Board.

☛ ***Warning: Do not attach external voltage supplies while also supplying power to the XStend Board with an XS Board.***

☛ ***Warning: Never place shunts on either J12 or J14 or you will short the power supplies to ground and damage the XStend Board and the attached XS Board..***

2.2 LEDs

The XStend Board provides a bargraph LED with eight LEDs (D1—D8) and two more LED displays (U1 and U2) for use by an XS Board. All of these LEDs are active-low meaning that an LED segment will glow when a logic-low is applied to it.

The LEDs are enabled and disabled by setting the shunts on the 3-pin jumpers as described in Table 1 and as shown in Figure 3.

Table 1: Jumper settings for XStend LEDs.

Jumper	Setting
J8	Removing the shunt on this jumper disconnects the power from bargraph LEDs D1—D8. Placing the shunt on the lower two pins of the jumper enables the LEDs for use with an XS95 Board inserted in the XStend Board. The LEDs can be used with an XS40 Board if the shunt is placed on the upper two pins.
J4	Removing the shunt on this jumper disconnects the power from left LED digit U1. Placing the shunt on the lower two pins of the jumper enables the LED digit for use with an XS95 Board inserted in the XStend Board. The LED digit can be used with an XS40 Board if the shunt is placed on the upper two pins.
J7	Removing the shunt on this jumper disconnects the power from right LED digit U2. Placing the shunt on the lower two pins of the jumper enables the LED digit for use with an XS95 Board inserted in the XStend Board. The LED digit can be used with an XS40 Board if the shunt is placed on the upper two pins.

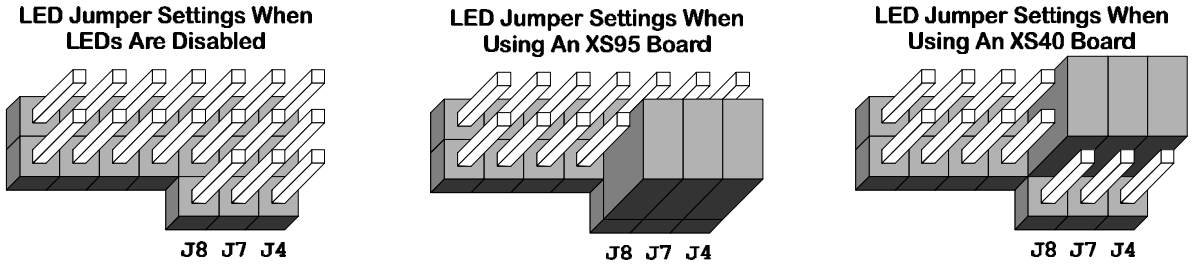


Figure 3: Shunt placement for setting the XStend Board LED supply voltages.

Listings 1 and 2 show the connections from the XS40 and XS95 Boards to the LEDs on the XStend Board expressed as UCF constraints (for the UCF syntax and usage tips, check out http://www.xilinx.com/techdocs/2449.htm).

Listing 1: Connections between the XStend LEDs and the XS40.

```
# LEFT LED DIGIT SEGMENT CONNECTIONS (ACTIVE-LOW)
NET LSB<0>         LOC=P3;
NET LSB<1>         LOC=P4;
NET LSB<2>         LOC=P5;
NET LSB<3>         LOC=P78;
NET LSB<4>         LOC=P79;
NET LSB<5>         LOC=P82;
NET LSB<6>         LOC=P83;
NET LDPB           LOC=P84;
#
# RIGHT LED DIGIT SEGMENT CONNECTIONS (ACTIVE-LOW)
NET RSB<0>         LOC=P59;
NET RSB<1>         LOC=P57;
NET RSB<2>         LOC=P51;
NET RSB<3>         LOC=P56;
NET RSB<4>         LOC=P50;
NET RSB<5>         LOC=P58;
NET RSB<6>         LOC=P60;
NET RDPB           LOC=P28;
#
# INDIVIDUAL LED CONNECTIONS (ACTIVE-LOW)
NET DB<1>          LOC=P41;
NET DB<2>          LOC=P40;
NET DB<3>          LOC=P39;
NET DB<4>          LOC=P38;
NET DB<5>          LOC=P35;
NET DB<6>          LOC=P81;
NET DB<7>          LOC=P80;
NET DB<8>          LOC=P10;
```

Listing 2: Connections between the XStend LEDs and the XS95.

```
# LEFT LED DIGIT SEGMENT CONNECTIONS (ACTIVE-LOW)
NET LSB<0>         LOC=P1;
NET LSB<1>         LOC=P2;
NET LSB<2>         LOC=P3;
NET LSB<3>         LOC=P75;
NET LSB<4>         LOC=P79;
NET LSB<5>         LOC=P82;
NET LSB<6>         LOC=P83;
NET LDPB           LOC=P84;
#
# RIGHT LED DIGIT SEGMENT CONNECTIONS (ACTIVE-LOW)
NET RSB<0>         LOC=P58;
NET RSB<1>         LOC=P56;
NET RSB<2>         LOC=P54;
NET RSB<3>         LOC=P55;
NET RSB<4>         LOC=P53;
NET RSB<5>         LOC=P57;
NET RSB<6>         LOC=P61;
NET RDPB           LOC=P34;
#
# INDIVIDUAL LED CONNECTIONS (ACTIVE-LOW)
NET DB<1>          LOC=P44;
NET DB<2>          LOC=P43;
NET DB<3>          LOC=P41;
NET DB<4>          LOC=P40;
NET DB<5>          LOC=P39;
NET DB<6>          LOC=P37;
NET DB<7>          LOC=P36;
NET DB<8>          LOC=P35;
```

2.3 Switches

The XStend has a bank of eight DIP switches and two pushbuttons (labeled SPARE and RESET) that are accessible from an XS Board. (There is a third pushbutton labeled PROGRAM which is used to initiate the programming of the XS40 Board. It is not intended to be a general-purpose input.)

When closed or ON, each DIP switch pulls the connected pin of the XS Board to ground. When the DIP switch is open or OFF, the pin is pulled high through a 10KΩ resistor.

☛ ***When not being used, the DIP switches should be left in the open or OFF configuration so the pins of the XS Board are not tied to ground and can freely move between logic low and high levels.***

When pressed, each pushbutton pulls the connected pin of the XS Board to ground. Otherwise, the pin is pulled high through a 10 KΩ resistor.

Listings 3 and 4 show the connections from the XS40 and XS95 Boards to the switches on the XStend Board expressed as UCF constraints.

Listing 3: Connections between the XStend DIP and pushbutton switches and the XS40.

```
# DIP SWITCH CONNECTIONS
NET DIPSW<1>      LOC=P7;
NET DIPSW<2>      LOC=P8;
NET DIPSW<3>      LOC=P9;
NET DIPSW<4>      LOC=P6;
NET DIPSW<5>      LOC=P77;
NET DIPSW<6>      LOC=P70;
NET DIPSW<7>      LOC=P66;
NET DIPSW<8>      LOC=P69;
#
# PUSHBUTTON SWITCH CONNECTIONS (ACTIVE-LOW)
NET SPAREB        LOC=P67;
NET RESETB        LOC=P37;
```

Listing 4: Connections between the XStend DIP and pushbutton switches and the XS95.

```
# DIP SWITCH CONNECTIONS
NET DIPSW<1>      LOC=P6;
NET DIPSW<2>      LOC=P7;
NET DIPSW<3>      LOC=P11;
NET DIPSW<4>      LOC=P5;
NET DIPSW<5>      LOC=P72;
NET DIPSW<6>      LOC=P71;
NET DIPSW<7>      LOC=P66;
NET DIPSW<8>      LOC=P70;
#
# PUSHBUTTON SWITCH CONNECTIONS (ACTIVE-LOW)
NET SPAREB        LOC=P67;
NET RESETB        LOC=P10;
```

2.4 VGA Interface

The XStend Board provides an XS Board with an interface to a VGA monitor through connector J5. (Version 1.2 and higher of the XS Boards already have their own VGA interfaces, so the XStend circuitry is redundant for them.) The XS Board can drive the active-low horizontal and vertical sync signals that control the width and height of the video frame. The XS Board also has access to two bits each of red, green, and blue color signals so it can generate pixels in any of $2^2 \times 2^2 \times 2^2 = 64$ different colors.

Listings 5 and 6 show the connections from the XS40 and XS95 Boards to the VGA interface of the XStend Board. (These pin assignments are identical to the pin assignments for the XS Boards which have their own VGA interfaces.)

Listing 5: Connections between the XStend VGA interface and the XS40.

```
# VGA CONNECTIONS
NET VSYNCB         LOC=P67;
NET HSYNCB         LOC=P19;
NET RED<1>         LOC=P18;
NET RED<0>         LOC=P23;
NET GREEN<1>       LOC=P20;
NET GREEN<0>       LOC=P24;
NET BLUE<1>        LOC=P26;
NET BLUE<0>        LOC=P25;
```

Listing 6: Connections between the XStend VGA interface and the XS95.

```
# VGA CONNECTIONS
NET VSYNCB         LOC=P24;
NET HSYNCB         LOC=P15;
NET RED<1>         LOC=P14;
NET RED<0>         LOC=P18;
NET GREEN<1>       LOC=P17;
NET GREEN<0>       LOC=P19;
NET BLUE<1>        LOC=P23;
NET BLUE<0>        LOC=P21;
```

2.5 PS/2 Keyboard Interface

The XStend Board provides an XS Board with a PS/2-style interface (mini-DIN connector J6) to either a keyboard or a mouse. The XS Board receives two signals from the PS/2 interface: a clock signal and a serial data stream that is synchronized with the falling edges on the clock signal.

Listings 7 and 8 show the connections from the XS40 and XS95 Boards to the PS/2 interface of the XStend Board (expressed as UCF constraints):

Listing 7: Connections between the XStend PS/2 interface and the XS40.

```
# PS/2 KEYBOARD CONNECTIONS
NET KB_CLK         LOC=P68;
NET KB_DATA        LOC=P69;
```

Listing 8: Connections between the XStend PS/2 interface and the XS95.

```
# PS/2 KEYBOARD CONNECTIONS
NET KB_CLK         LOC=P26;
NET KB_DATA        LOC=P70;
```

2.6 RAMs

The XStend Board adds an additional 64 KBytes of RAM to the 32 KBytes already on the XS Board. The XStend RAM connects to the same pins as the XS Board RAM for the address bus, data bus, write-enable, and output-enable. The chip-selects of the XStend Board RAMs are connected to different pins so all the RAMs can be individually selected.

Here are the connections from the XS40 and XS95 Boards to their own RAMs and the RAMs of the XStend Board (expressed as UCF constraints):

Listing 9: Connections between the XStend RAMs and the XS40.

```
NET D<0>          LOC=P41;    # DATA BUS
NET D<1>          LOC=P40;
NET D<2>          LOC=P39;
NET D<3>          LOC=P38;
NET D<4>          LOC=P35;
NET D<5>          LOC=P81;
NET D<6>          LOC=P80;
NET D<7>          LOC=P10;
NET A<0>          LOC=P3;     # LOWER BYTE OF ADDRESS
NET A<1>          LOC=P4;
NET A<2>          LOC=P5;
NET A<3>          LOC=P78;
NET A<4>          LOC=P79;
NET A<5>          LOC=P82;
NET A<6>          LOC=P83;
NET A<7>          LOC=P84;
NET A<8>          LOC=P59;    # UPPER BYTE OF ADDRESS
NET A<9>          LOC=P57;
NET A<10>         LOC=P51;
NET A<11>         LOC=P56;
NET A<12>         LOC=P50;
NET A<13>         LOC=P58;
NET A<14>         LOC=P60;
NET WEB           LOC=P62;    # ACTIVE-LOW WRITE-ENABLE FOR ALL RAMS
NET OEB           LOC=P61;    # ACTIVE-LOW OUTPUT-ENABLE FOR ALL RAMS
NET CEB           LOC=P65;    # ACTIVE-LOW CHIP-ENABLE FOR XS40 RAM
NET LCEB          LOC=P7;     # ACTIVE-LOW CHIP-ENABLE FOR LEFT XSTEND RAM
NET RCEB          LOC=P8;     # ACTIVE-LOW CHIP-ENABLE FOR RIGHT XSTEND RAM
```

Listing 10: Connections between the XStend RAMs and the XS95.

```
NET D<0>          LOC=P44;    # DATA BUS
NET D<1>          LOC=P43;
NET D<2>          LOC=P41;
NET D<3>          LOC=P40;
NET D<4>          LOC=P39;
NET D<5>          LOC=P37;
NET D<6>          LOC=P36;
NET D<7>          LOC=P35;
NET A<0>          LOC=P75;    # LOWER BYTE OF ADDRESS
NET A<1>          LOC=P79;
NET A<2>          LOC=P82;
NET A<3>          LOC=P84;
NET A<4>          LOC=P1;
NET A<5>          LOC=P3;
NET A<6>          LOC=P83;
NET A<7>          LOC=P2;
NET A<8>          LOC=P58;    # UPPER BYTE OF ADDRESS
NET A<9>          LOC=P56;
NET A<10>         LOC=P54;
NET A<11>         LOC=P55;
NET A<12>         LOC=P53;
NET A<13>         LOC=P57;
```

```
NET A<14>          LOC=P61;
NET WEB            LOC=P63;    # ACTIVE-LOW WRITE-ENABLE FOR ALL RAMS
NET OEB            LOC=P62;    # ACTIVE-LOW OUTPUT-ENABLE FOR ALL RAMS
NET CEB            LOC=P65;    # ACTIVE-LOW CHIP-ENABLE FOR XS95 RAM
NET LCEB           LOC=P6;     # ACTIVE-LOW CHIP-ENABLE FOR LEFT XSTEND RAM
NET RCEB           LOC=P7;     # ACTIVE-LOW CHIP-ENABLE FOR RIGHT XSTEND RAM
```

2.7 Stereo Codec

The XStend Board has a stereo codec that accepts two analog input channels from jack J9, digitizes the analog values, and sends the digital values to the XS Board as a serial bit stream. The codec also accepts a serial bit stream from the XS Board and converts it into two analog output signals which exit the XStend Board through jack J10.

The codec is configured by placing shunts on the jumpers as indicated in Table 2 and Figure 4.

Table 2: Jumper settings for XStend codec.

Jumper	**Setting**
J11	Placing a shunt on this jumper disables the codec by holding it in the reset state.
J13, J15	Placing shunts across two of the three pins of these jumpers selects the digital de-emphasis for different sampling rates:
0 0	De-emphasis for 32 KHz
0 1	De-emphasis for 44.1 KHz
1 0	De-emphasis for 48 KHz
1 1	De-emphasis off
J17	Removing this shunt prevents the codec's serial data output from reaching the XS Board.
J20-J27	Removing these shunts interrupts the flow of the analog signals into and out of the codec.
J28	This header provides access to the analog VCC and GND signals. **A shunt should never be placed on this header!**

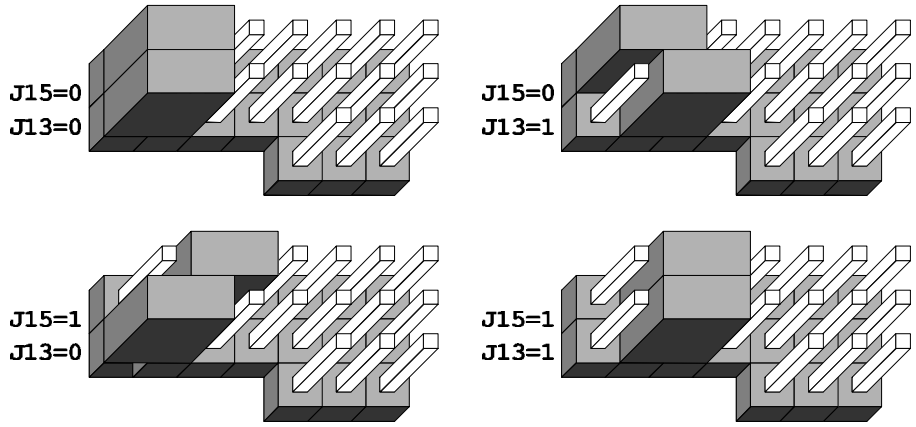


Figure 4: Shunt placement for setting the codec sampling rate de-emphasis.

Listings 11 and 12 show the connections from the XS40 Board to the codec interface on the XStend Board (expressed as UCF constraints):

Listing 11: Connections between the XStend stereo codec and the XS40.

```
# STEREO CODEC CONNECTIONS
NET MCLK          LOC=P9;     # MASTER CLOCK TO CODEC
NET LRCK          LOC=P66;    # LEFT/RIGHT CODEC CHANNEL SELECT
NET SCLK          LOC=P77;    # SERIAL DATA CLOCK
NET SDOUT         LOC=P6;     # SERIAL DATA OUTPUT FROM CODEC
NET SDIN          LOC=P70;    # SERIAL DATA INPUT TO CODEC
NET CCLK          LOC=P44;    # CONTROL SIGNAL CLOCK
NET CDIN          LOC=P45;    # SERIAL CONTROL INPUT TO CODEC
NET CSB           LOC=P46;    # SERIAL CONTROL CHIP SELECT
```

Listing 12: Connections between the XStend codec and the XS95.

```
# STEREO CODEC CONNECTIONS
NET MCLK          LOC=P11;    # MASTER CLOCK TO CODEC
NET LRCK          LOC=P5;     # LEFT/RIGHT CODEC CHANNEL SELECT
NET SCLK          LOC=P72;    # SERIAL DATA CLOCK
NET SDOUT         LOC=P66;    # SERIAL DATA OUTPUT FROM CODEC
NET SDIN          LOC=P71;    # SERIAL DATA INPUT TO CODEC
NET CCLK          LOC=P46;    # CONTROL SIGNAL CLOCK
NET CDIN          LOC=P47;    # SERIAL CONTROL INPUT TO CODEC
NET CSB           LOC=P48;    # SERIAL CONTROL CHIP SELECT
```

The analog stereo input and output signals enter and exit the XStend Board through the 1/8” jacks J9 and J10, respectively. The output of an audio CD player can be input through J9 and a set of small stereo headphones can be connected to J10 for listening to the processed output.

The analog signals that go in and out of the codec chip pass through jumpers J20-J27. Shunts should be placed on all these jumpers so that the analog signals are not interrupted. The digitized data output from the codec passes through jumper J17 on its way to the XS Board inserted in the XStend Board. A shunt should be placed on J17 when the codec is being used. Because the serial data output of the codec is not tristatable and because it shares the input to the XS Board

with other resources on the XStend Board, the shunt on J17 should be removed when the codec is not being used.

☛**Never place a shunt on header J28! J28 provides access to the VCC and GND of the analog section of the XStend Board. Placing a shunt on J28 will damage the XStend Board.**

2.8 XILINX Xchecker Interface

The XS Board inserted in the XStend Board can be configured and tested using a XILINX Xchecker cable attached to header J19. When using the Xchecker cable, you must not connect the cable between the XS Board and the parallel port of the PC. In addition, when using the Xchecker cable with an XStend/XS40 combination, you must make the following adjustments to the XS40 Board:

- Remove the shunts from jumpers J4, J6, J10 and J11 of the XS40 Board;
- Remove the serial EPROM from socket U7.

The connections between the Xchecker cable and the XS40 and XS95 Boards are listed in Table 3. The configuration and readback signals are not applicable to the XS95 Board, so only the JTAG and VCC/GND signals are listed for it.

Table 3: Connections between the XStend Board Xchecker interface and the XS40 and XS95 Boards.

Xchecker Pin	**XS40 Pin**	**XS95 Pin**
1 – VCC (+5V)	2	78
2 – RT	32	N/A
3 – GND	52	49
4 – RD	30	N/A
6 – TRIG	7	N/A
7 – CCLK	73	N/A
9 – DONE	53	N/A
10 – TDI	15	28
11 – DIN	71	N/A
12 – TCK	16	30
13 – PROGRAM	55	N/A
14 – TMS	17	29
15 – INIT	41	N/A
16 – CLKI	13	N/A
17 – RST	8	N/A
18 – CLKO	9	N/A

2.9 Prototyping Area

The XStend Board has a prototyping area consisting of component through-holes on an 0.1"×0.1" grid interspersed with a network of alternating VCC and GND buses as shown in Figure 5. The buses carrying VCC run on the top side of the XStend Board while the GND buses run on the bottom side. The VCC and GND buses have connection holes in which a small wire can be soldered to make a connection to a nearby component through-hole.

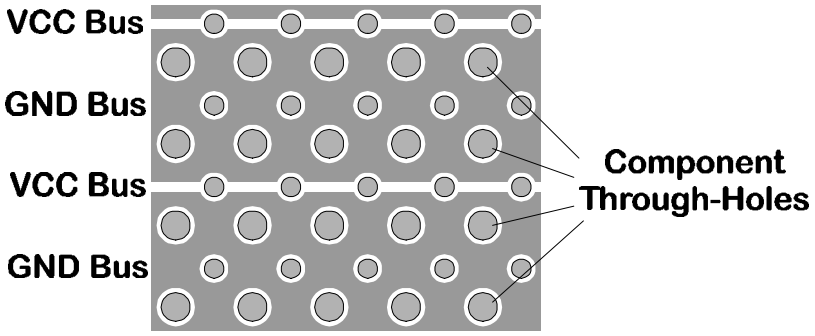


Figure 5: Top-side view of the network of VCC and GND buses around the component through-holes in the XStend Board prototyping area.

The placement of the shunt on jumper J16 will determine whether the VCC buses in the prototyping area carry either 5V or 3.3V (see Figure 6). Of course the jumper selection will have no effect unless you have both these voltages supplied to the XStend Board either by the XS Board or by connecting external power supplies.

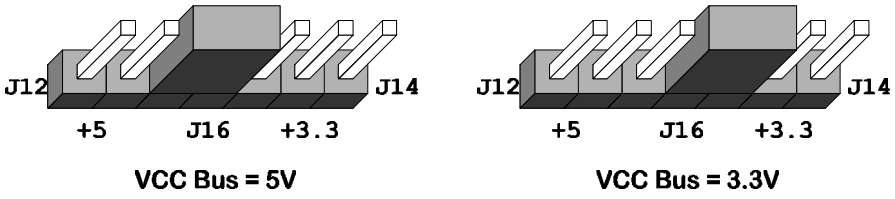


Figure 6: Shunt placement for setting the VCC bus voltage..

Connections from the XS Board to the prototyping area are made through connector J3. The arrangement of pins on this connector exactly matches the arrangement of pins on the XS40 Board. For example, the pin at the bottom-left of J3 on the XStend Board corresponds to pin 21 at the bottom-left of the XS40 Board.

The XS95 Board has a completely different pin arrangement than the XS40. Therefore each pin on J3 is explicitly labelled with the corresponding pin number on the XS95 Board. For example, the pin at the bottom-left of J3 on the XStend Board is connected to pin 68 near the top-left of the XS95 Board.

2.10 Daughterboard Connector

Daughterboards with specialized circuitry can be connected to the XStend board through connector J18. This 42×2 connector brings all the I/O and VCC/GND from the XS40 or XS95 Board to the daughterboard.

3 XStend Board Programmer's Model

The interconnections of the XStend Board resources and an XS40 or XS95 Board are shown in **Figure 7** and **Figure 8**, respectively. These figures remove much of the extraneous detail of the actual schematics, so we refer to them as *programmer's models*.

Items within the shaded area in each figure correspond to circuitry housed on the XS Board. The remaining items are XStend Board resources.

A cursory glance at the figures reveals that many of the resources share connections. For example, the codec, DIP switch, and microcontroller port P1 are all connected to the same set of pins on the FPGA or CPLD. So any design has to ensure that only one of these resources is outputing data at any particular time. (Hence the need in some designs to place the DIP switches in the OPEN position, or remove the shunt through which the codec SDOUT drives serial data, or keep the microcontroller in the reset state.)

Table 4 and **Table 5** list the same interconnection data for the XS40 and XS95 Boards, respectively, in a tabular format which makes it easier to see which resources share common connections.

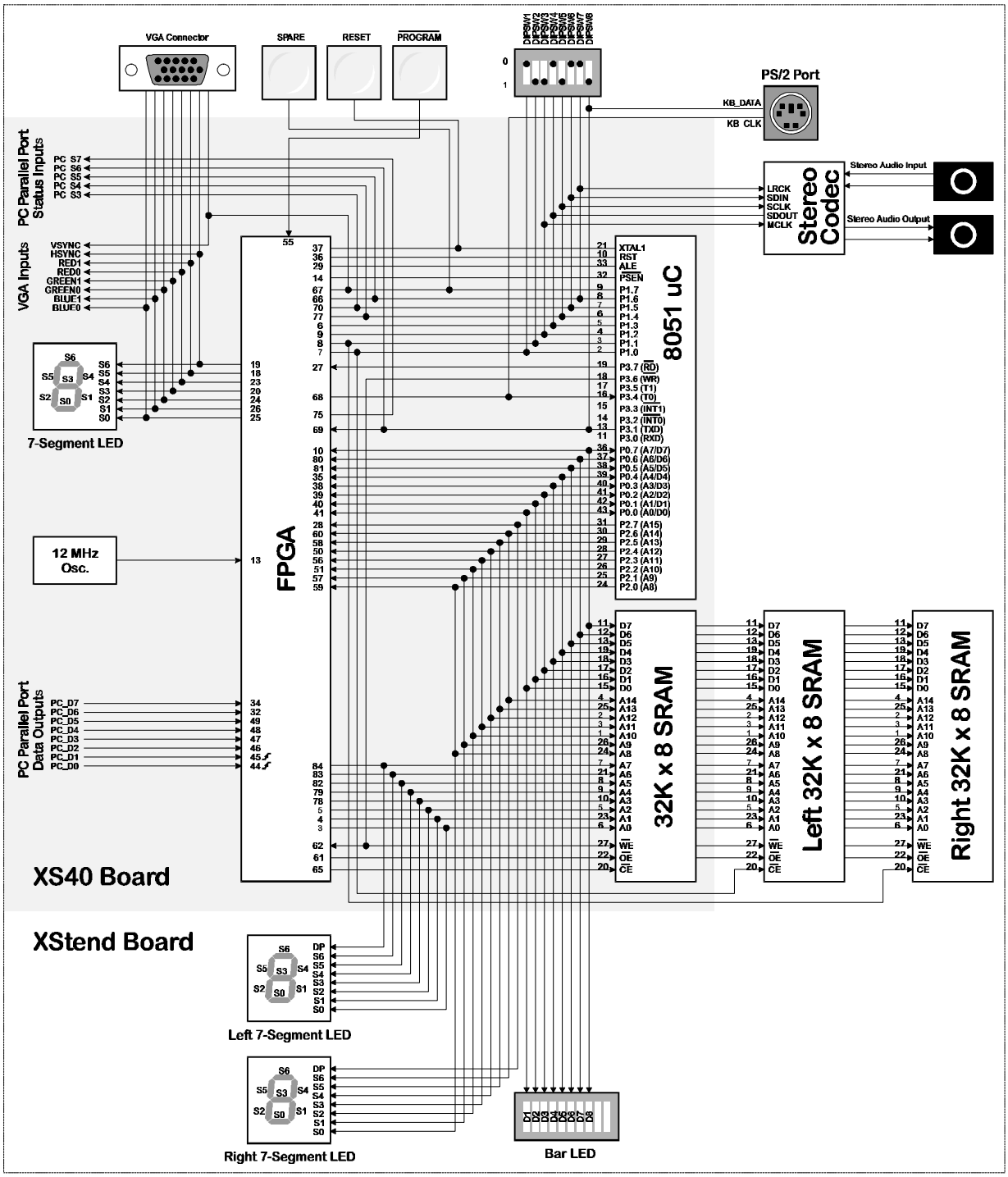


Figure 7: Programmer's model of the XS40/XStend Board combination.

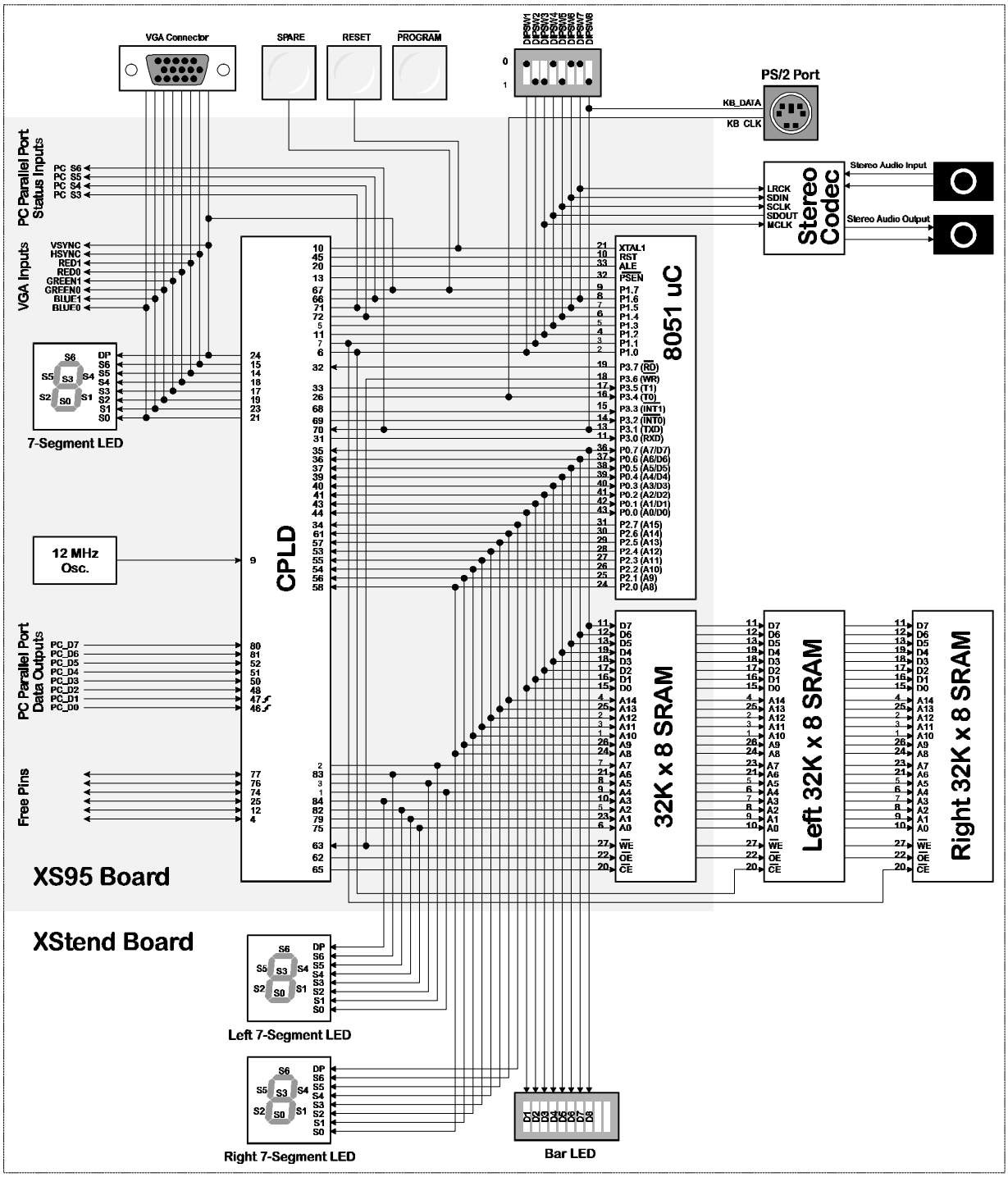


Figure 8: Programmer's model of the XS95/XStend Board combination.

Table 4: Connections between the XS40 Board and the XStend Board resources.

XS40 Pin (J1,J3,J18)	Power/ GND	DIP Switch	Push-buttons	LEDs	VGA Interface	PS/2 Interface	RAMs	Stereo Codec	8051 uC	PC Parallel Port	Oscillator	Function	UW-FPGA BOARD Pin
2	+5V											+5V power source	
3				LSB0			A0					Left LED segment; RAM address line	P35
4				LSB1			A1					Left LED segment; RAM address line	P36
5				LSB2			A2					Left LED segment; RAM address line	P29
6		DIPSW4						SDOUT	P1.3			DIP switch; codec serial data output; uC I/O	P24
7		DIPSW1					LCEB		P1.0			DIP switch; left RAM chip-enable, uC I/O port	P19
8		DIPSW2					RCEB		P1.1			DIP switch; right RAM chip-enable, uC I/O port	P20
9		DIPSW3						MCLK	P1.2			DIP switch; codec master clock; uC I/O port	P23
10				DB8			D7		P0.7			LED; RAM data line; uC muxed address/data line	P61
13											CLK	XS Board oscillator	
14									PSENB			uC program store-enable	
15												JTAG TDI; DIN	
16												JTAG TCK; CCLK	
17												JTAG TMS	
18				S5	RED1							XS Board LED segment; VGA color signal	
19				S6	HSYNCB							XS Board LED segment; VGA horiz. sync.	
20				S3	GREEN1							XS Board LED segment; VGA color signal	
23				S4	RED0							XS Board LED segment; VGA color signal	
24				S2	GREEN0							XS Board LED segment; VGA color signal	
25				S0	BLUE0							XS Board LED segment; VGA color signal	
26				S1	BLUE1							XS Board LED segment; VGA color signal	
27									P3.7 (RD_)			uC read line	
28				RDPB					P2.7			Right LED decimal-point; uC I/O port	P41
29									ALEB			uC address-latch-enable	
30												Serial EEPROM chip-enable	
32										PC_D6		PC parallel port data output	
34										PC_D7		PC parallel port data output	
35				DB5			D4		P0.4			LED; RAM data line; uC muxed address/data line	P66
36									RST			uC reset	
37			RESETB						XTAL1			Pushbutton; uC clock	P56
38				DB4			D3		P0.3			LED; RAM data line; uC muxed address/data line	P57
39				DB3			D2		P0.2			LED; RAM data line; uC muxed address/data line	P58
40				DB2			D1		P0.1			LED; RAM data line; uC muxed address/data line	P59
41				DB1			D0		P0.0			LED; RAM data line; uC muxed address/data line	P60
44								CCLK		PC_D0		Codec control line; PC parallel port data output	
45								CDIN		PC_D1		Codec control line; PC parallel port data output	
46								CSB		PC_D2		Codec control line; PC parallel port data output	
47										PC_D3		PC parallel port data output	
48										PC_D4		PC parallel port data output	
49										PC_D5		PC parallel port data output	
50				RSB4			A12		P2.4			Right LED segment; RAM address line; uC I/O port	P48
51				RSB2			A10		P2.2			Right LED segment; RAM address line; uC I/O port	P45
52	GND											Power supply ground	
54	5.0V/3.3V											5V/3.3V power supply (4000E/4000XL)	
55			PROGRAM									XS40 configuration control	P55
56				RSB3			A11		P2.3			Right LED segment; RAM address line; uC I/O port	P51
57				RSB1			A9		P2.1			Right LED segment; RAM address line; uC I/O port	P47
58				RSB5			A13		P2.5			Right LED segment; RAM address line; uC I/O port	P50
59				RSB0			A8		P2.0			Right LED segment; RAM address line; uC I/O port	P46
60				RSB6			A14		P2.6			Right LED segment; RAM address line; uC I/O port	P49
61							OEB					RAM output-enable	
62							WEB		P3.6 (WR_)			RAM write-enable; uC I/O port	
65							CEB					XS Board RAM chip-enable	
66		DIPSW7						LRCK	P1.6	PC_S5		DIP switch; codec left-right channel switch; uC I/O port; P	P27
67			SPAREB		VSYNCB				P1.7			Pushbutton; VGA vert. sync.; uC I/O port	P18
68						KB_CLK			P3.4 (T0)			PS/2 keyboard clock; uC I/O port	
69		DIPSW8				KB_DATA			P3.1 (TX	PC_S6		DIP switch; PS/2 keyboard serial data; uC I/O port; PC par	P28
70		DIPSW6						SDIN	P1.5	PC_S3		DIP switch; codec serial input data; uC I/O port; PC parall	P26
71												JTAG TDI; DIN	
72												JTAG TDO; DOUT	
73												JTAG TCK; CCLK	
75										PC_S7		JTAG TDO; DOUT; PC parallel port status input	
77		DIPSW5						SCLK	P1.4	PC_S4		DIP switch; codec serial I/O clock; uC I/O port; PC parallel	P25
78				LSB3			A3					Left LED segment; RAM address line	P44
79				LSB4			A4					Left LED segment; RAM address line	P38
80				DB7			D6		P0.6			LED; RAM data line; uC muxed address/data line	P62
81				DB6			D5		P0.5			LED; RAM data line; uC muxed address/data line	P65
82				LSB5			A5					Left LED segment; RAM address line	P40
83				LSB6			A6					Left LED segment; RAM address line	P39
84				LDPB			A7					Left LED decimal-point; RAM address line	P37

Table 5: Connections between the XS95 Board and the XStend Board resources.

XS95 Pins (J2)	Power/ GND	DIP Switch	Push-buttons	LEDs	VGA Interface	PS/2 Interface	RAMs	Stereo Codec	8051 Uc	PC Parallel Port	Oscillator	Function	UW-FPGA BOARD Pin
1				LSB0			A4					Left LED segment; RAM address line	P35
2				LSB1			A7					Left LED segment; RAM address line	P36
3				LSB2			A5					Left LED segment; RAM address line	P29
4												Uncommitted XS95 I/O pin	
5		DIPSW4						SDOUT	P1.3			DIP switch; codec serial data output; uC I/O	P24
6		DIPSW1					LCEB		P1.0			DIP switch; left RAM chip-enable, uC I/O port	P19
7		DIPSW2					RCEB		P1.1			DIP switch; right RAM chip-enable, uC I/O port	P20
9											CLK	XS Board oscillator	
10			RESETB						XTAL1			Pushbutton; uC clock	P56
11		DIPSW3						MCLK	P1.2			DIP switch; codec master clock; uC I/O port	P23
12												Uncommitted XS95 I/O pin	
13									PSENB			uC program store-enable	
14				S5	RED1							XS Board LED segment; VGA color signal	
15				S6	HSYNCB							XS Board LED segment; VGA horiz. sync.	
17				S3	GREEN1							XS Board LED segment; VGA color signal	
18				S4	RED0							XS Board LED segment; VGA color signal	
19				S2	GREEN0							XS Board LED segment; VGA color signal	
20									ALEB			uC address-latch-enable	
21				S0	BLUE0							XS Board LED segment; VGA color signal	
23				S1	BLUE1							XS Board LED segment; VGA color signal	
25												Uncommitted XS95 I/O pin	
26						KB_CLK			P3.4 (T0)			PS/2 keyboard clock; uC I/O port	
28												JTAG TDI; DIN	
29												JTAG TMS	
30												JTAG TCK; CCLK	
31									P3.0 (RXD)			uC I/O port	
32									P3.7 (RD_)			uC I/O port	
33									P3.5 (T1)			uC I/O port	
34				RDPB					P2.7			Right LED decimal-point; RAM address line; uC I/O port	P41
35				DB8			D7		P0.7			LED; RAM data line; uC muxed address/data line	P61
36				DB7			D6		P0.6			LED; RAM data line; uC muxed address/data line	P62
37				DB6			D5		P0.5			LED; RAM data line; uC muxed address/data line	P65
39				DB5			D4		P0.4			LED; RAM data line; uC muxed address/data line	P66
40				DB4			D3		P0.3			LED; RAM data line; uC muxed address/data line	P57
41				DB3			D2		P0.2			LED; RAM data line; uC muxed address/data line	P58
43				DB2			D1		P0.1			LED; RAM data line; uC muxed address/data line	P59
44				DB1			D0		P0.0			LED; RAM data line; uC muxed address/data line	P60
45									RST			uC reset	
46								CCLK		PC_D0		Codec control line; PC parallel port data output	
47								CDIN		PC_D1		Codec control line; PC parallel port data output	
48								CSB		PC_D2		Codec control line; PC parallel port data output	
49	GND											Power supply ground	
50										PC_D3		PC parallel port data output	
51										PC_D4		PC parallel port data output	
52										PC_D5		PC parallel port data output	
53				RSB4			A12		P2.4			Right LED segment; RAM address line; uC I/O port	P48
54				RSB2			A10		P2.2			Right LED segment; RAM address line; uC I/O port	P45
55				RSB3			A11		P2.3			Right LED segment; RAM address line; uC I/O port	P51
56				RSB1			A9		P2.1			Right LED segment; RAM address line; uC I/O port	P47
57				RSB5			A13		P2.5			Right LED segment; RAM address line; uC I/O port	P50
58				RSB0			A8		P2.0			Right LED segment; RAM address line; uC I/O port	P46
59												JTAG TDO; DOUT	
61				RSB6			A14		P2.6			Right LED segment; RAM address line; uC I/O port	P49
62							OEB					RAM output-enable	
63							WEB		P3.6 (WR_)			RAM write-enable; uC I/O port	
65							CEB					XS Board RAM chip-enable	
66		DIPSW7						LRCK	P1.6	PC_S5		DIP switch; codec left-right channel select; uC I/O port; PC	P27
68									P3.3 (INT1_)			uC I/O port	
69									P3.2 (INT0_)			uC I/O port	
70		DIPSW8				KB_DATA			P3.1 (TX	PC_S6		DIP switch; PS/2 keyboard serial data; uC I/O port; PC par	P28
71		DIPSW6						SDIN	P1.5	PC_S3		DIP switch; codec serial input data; uC I/O port; PC parall	P26
72		DIPSW5						SCLK	P1.4	PC_S4		DIP switch; codec serial clock; uC I/O port; PC parallel po	P25
74												Uncommitted XS95 I/O pin	
75				LSB3			A0					Left LED segment; RAM address line	P44
76												Uncommitted XS95 I/O pin	
77												Uncommitted XS95 I/O pin	
78	+5V											+5V power source	
79				LSB4			A1					Left LED segment; RAM address line	P38
80										PC_D7		PC parallel port data output	
81										PC_D6		PC parallel port data output	
82				LSB5			A2					Left LED segment; RAM address line	P40
83				LSB6			A6					Left LED segment; RAM address line	P39
84				LDPB			A3					Left LED decimal-point; RAM address line	P37
24,67			SPAREE	DP	VSYNCB				P1.7			Pushbutton; XS Board LED decimal-point; VGA horiz. syn	P18

4 Example Designs for the XStend Board

With the programmer's models in hand, several example designs can be built using the XStend Board coupled with an XS40 or XS95 Board.

4.1 Displaying Switch Settings on the LEDs

This example creates a circuit that displays the settings of the DIP switches on the LEDs and LED digits of the XStend and XS Boards. The particular set of LEDs which is activated is selected by the SPARE and RESET pushbuttons. The VHDL code for this example is shown in **Listing 13**.

The steps for compiling and testing the design using an XS40 combined with an XStend Board are as follows:

- Synthesize the VHDL code in the SWTCH40\SWITCHES.VHD file for an XC4005XL FPGA.
- Compile the synthesized netlist using the SWTCH40.UCF constraint file (**Listing 14**).
- Mount an XS40 Board in the XStend Board and attach the downloading cable from the XS40 to the PC parallel port. Apply 9VDC though jack J9 of the XS40. Place shunts on jumpers J4, J7, and J8 of the XStend Board to enable the LED displays. Remove the shunt on jumper J17 to keep the XStend codec serial output from interfering with the DIP switch logic levels.
- Download the SWTCH40.BIT file into the XS40/XStend combination with the command: `XSLOAD SWTCH40.BIT`.
- Set the DIP switches and press the SPARE and RESET pushbuttons. Observe the results on the LEDs.

The steps for compiling and testing the design using an XS95 combined with an XStend Board are as follows:

- Synthesize the VHDL code in the SWTCH95\SWITCHES.VHD file for an XC95108 CPLD.
- Compile the synthesized netlist using the SWTCH95.UCF constraint file (

Listing 15).

- Generate an SVF file for the design.

- Mount an XS95 Board in the XStend Board and attach the downloading cable from the XS95 to the PC parallel port. Apply 9VDC though jack J9 of the XS95. Place shunts on jumpers J4, J7, and J8 of the XStend Board to enable the LED displays. Remove the shunt on jumper J17 to keep the XStend codec serial output from interfering with the DIP switch logic levels.

- Download the SWTCH95.SVF file into the XS95/XStend combination with the command: `XSLOAD SWTCH95.SVF`.

- Set the DIP switches and press the SPARE and RESET pushbuttons. Observe the results on the LEDs.

Listing 13: VHDL code for using the XStend LEDs and switches.

```
001- LIBRARY IEEE;
002- USE IEEE.STD_LOGIC_1164.ALL;
003-
004- ENTITY switches IS
005-  PORT
006-  (
007-    dipsw: IN STD_LOGIC_VECTOR(8 DOWNTO 1); -- DIP switches
008-    spareb: IN STD_LOGIC; -- SPARE pushbutton
009-    resetb: IN STD_LOGIC; -- RESET pushbutton
010-
011-    s: OUT STD_LOGIC_VECTOR(6 DOWNTO 0);    -- XS Board LED digit
012-    lsb: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  -- XStend left LED digit
013-    rsb: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  -- XStend right LED digit
014-    db: OUT STD_LOGIC_VECTOR(8 DOWNTO 1);   -- XStend bargraph LED
015-
016-    oeb: OUT STD_LOGIC;   -- output enable for all RAMs
017-    rst: OUT STD_LOGIC    -- microcontroller reset
018-  );
019- END switches;
020-
021- ARCHITECTURE switches_arch OF switches IS
022- BEGIN
023-  -- this prevents accidental activation of the RAMs or microcontroller
024-  oeb <= '1'; -- disable all the RAM output drivers
025-  rst <= '1'; -- disable the microcontroller
026-
027-  -- light the XS Board LED digit with the pattern from the
028-  -- DIP switches if both pushbuttons are pressed.
029-  -- these LED segments are active-high.
030-  s <= dipsw(7 DOWNTO 1) WHEN (spareb='0' AND resetb='0') ELSE
031-        "0000000";  -- otherwise keep LED digit dark
032-
033-  -- light the XStend left LED digit with the pattern from the
034-  -- DIP switches if the RESET pushbutton is pressed.
035-  -- these LED segments are active low.
036-  lsb <= NOT(dipsw) WHEN (spareb='1' AND resetb='0') ELSE
037-        "11111111"; -- otherwise keep the LED digit dark
```

```
038-
039-  -- light the XStend right LED digit with the pattern from the
040-  -- DIP switches if the SPARE pushbutton is pressed.
041-  -- these LED segments are active low.
042-  rsb <= NOT(dipsw) WHEN (spareb='0' AND resetb='1') ELSE
043-        "11111111"; -- otherwise keep the LED digit dark
044-
045-  -- light the XStend bargraph LED with the pattern from the
046-  -- DIP switches if neither pushbutton is pressed
047-  -- these LED segments are active low.
048-  db <= NOT(dipsw) WHEN (spareb='1' AND resetb='1') ELSE
049-        "11111111"; -- otherwise keep the bargraph LED dark
050- END switches_arch;
```

Listing 14: XS40 UCF file for the LED/switch example.

```
001- net s<0>     loc=p25;    // XS40 board led digit segments
002- net s<1>     loc=p26;
003- net s<2>     loc=p24;
004- net s<3>     loc=p20;
005- net s<4>     loc=p23;
006- net s<5>     loc=p18;
007- net s<6>     loc=p19;
008- net rst      loc=p36;    // microcontroller reset
009- net oeb      loc=p61;    // RAM output enable
010- net dipsw<1> loc=p7;     // DIP switch inputs
011- net dipsw<2> loc=p8;
012- net dipsw<3> loc=p9;
013- net dipsw<4> loc=p6;
014- net dipsw<5> loc=p77;
015- net dipsw<6> loc=p70;
016- net dipsw<7> loc=p66;
017- net dipsw<8> loc=p69;
018- net spareb   loc=p67;    // SPARE pushbutton input
019- net resetb   loc=p37;    // RESET pushbutton input
020- net lsb<0>   loc=p3;     // XStend left led digit segments
021- net lsb<1>   loc=p4;
022- net lsb<2>   loc=p5;
023- net lsb<3>   loc=p78;
024- net lsb<4>   loc=p79;
025- net lsb<5>   loc=p82;
026- net lsb<6>   loc=p83;
027- net lsb<7>   loc=p84;
028- net rsb<0>   loc=p59;    // XStend right led digit segments
029- net rsb<1>   loc=p57;
030- net rsb<2>   loc=p51;
031- net rsb<3>   loc=p56;
032- net rsb<4>   loc=p50;
033- net rsb<5>   loc=p58;
034- net rsb<6>   loc=p60;
035- net rsb<7>   loc=p28;
036- net db<1>    loc=p41;    // XStend bargraph led segments
037- net db<2>    loc=p40;
038- net db<3>    loc=p39;
039- net db<4>    loc=p38;
040- net db<5>    loc=p35;
041- net db<6>    loc=p81;
042- net db<7>    loc=p80;
043- net db<8>    loc=p10;
```

Listing 15: XS95 UCF file for the LED/switch example.

```
001- net s<0>     loc=p21;    // XS Board LED digit segments
002- net s<1>     loc=p23;
003- net s<2>     loc=p19;
004- net s<3>     loc=p17;
005- net s<4>     loc=p18;
006- net s<5>     loc=p14;
007- net s<6>     loc=p15;
008- net rst      loc=p45;    // microcontroller reset
009- net oeb      loc=p62;    // RAM output enable
010- net dipsw<1> loc=p6;     // DIP switch inputs
011- net dipsw<2> loc=p7;
012- net dipsw<3> loc=p11;
013- net dipsw<4> loc=p5;
014- net dipsw<5> loc=p72;
015- net dipsw<6> loc=p71;
016- net dipsw<7> loc=p66;
017- net dipsw<8> loc=p70;
018- net spareb   loc=p67;    // SPARE pushbutton input
019- net resetb   loc=p10;    // RESET pushbutton input
020- net lsb<0>   loc=p1;     // XStend left LED digit segments
021- net lsb<1>   loc=p2;
022- net lsb<2>   loc=p3;
023- net lsb<3>   loc=p75;
024- net lsb<4>   loc=p79;
025- net lsb<5>   loc=p82;
026- net lsb<6>   loc=p83;
027- net lsb<7>   loc=p84;
028- net rsb<0>   loc=p58;    // XStend right LED digit segments
029- net rsb<1>   loc=p56;
030- net rsb<2>   loc=p54;
031- net rsb<3>   loc=p55;
032- net rsb<4>   loc=p53;
033- net rsb<5>   loc=p57;
034- net rsb<6>   loc=p61;
035- net rsb<7>   loc=p34;
036- net db<1>    loc=p44;    // XStend bargraph LED segments
037- net db<2>    loc=p43;
038- net db<3>    loc=p41;
039- net db<4>    loc=p40;
040- net db<5>    loc=p39;
041- net db<6>    loc=p37;
042- net db<7>    loc=p36;
043- net db<8>    loc=p35;
```

4.2 Displaying Graphics from RAM Through the VGA Interface

This section discusses the timing for the signals that drive a VGA monitor and describes a VHDL module that will let you drive a monitor with a picture stored in RAM.

4.2.1 VGA Color Signals

There are three signals -- red, green, and blue -- that send color information to a VGA monitor. These three signals each drive an electron gun that emits electrons which paint one primary color at a point on the monitor screen. Analog levels between 0 (completely dark) and 0.7 V (maximum brightness) on these control lines tell the monitor what intensities of these three primary colors to combine to make the color of a dot (or *pixel*) on the monitor's screen.

Each analog color input can be set to one of four levels by two digital outputs using a simple two-bit digital-to-analog converter (see **Figure 9**). The four possible levels on each analog input are combined by the monitor to create a pixel with one of $4 \times 4 \times 4 = 64$ different colors. So the six digital control lines let us select from a palette of 64 colors.

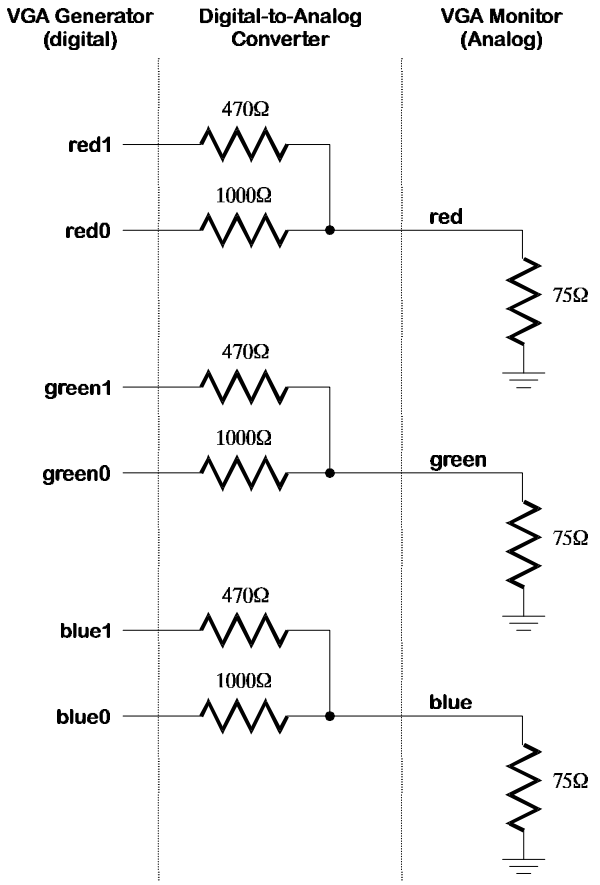


Figure 9: Digital-to-analog interface to a VGA monitor.

4.2.2 VGA Signal Timing

A single dot of color on a video monitor doesn't impart much information. A horizontal line of pixels carries a bit more information. But a *frame* composed of multiple lines can present an image on the monitor screen. A frame of VGA video typically has 480 lines and each line usually contains 640 pixels. In order to paint a frame, there are deflection circuits in the monitor that move the electrons emitted from the guns both left-to-right and top-to-bottom across the screen. These deflection circuits require two synchronization signals in order to start and stop the deflection circuits at the right times so that a line of pixels is painted across the monitor and the lines stack up from the top to the bottom to form an image. The timing for the VGA synchronization signals is shown in **Figure 10**.

Negative pulses on the *horizontal sync* signal mark the start and end of a line and ensure that the monitor displays the pixels between the left and right edges of the visible screen area. The actual pixels are sent to the monitor within a 25.17 μs window. The horizontal sync signal drops low a minimum of 0.94 μs after the last pixel and stays low for 3.77 μs. A new line of pixels can begin a minimum of 1.89 μs after the horizontal sync pulse ends. So a single line occupies 25.17 μs of a 31.77 μs interval. The other 6.6 μs of each line is the *horizontal blanking interval* during which the screen is dark.

In an analogous fashion, negative pulses on a *vertical sync* signal mark the start and end of a frame made up of video lines and ensure that the monitor displays the lines between the top and bottom edges of the visible monitor screen. The lines are sent to the monitor within a 15.25 ms window. The vertical sync signal drops low a minimum of 0.45 ms after the last line and stays low for 64 μs. The first line of the next frame can begin a minimum of 1.02 ms after the vertical sync pulse ends. So a single frame occupies 15.25 ms of a 16.784 ms interval. The other 1.534 ms of the frame interval is the *vertical blanking interval* during which the screen is dark.

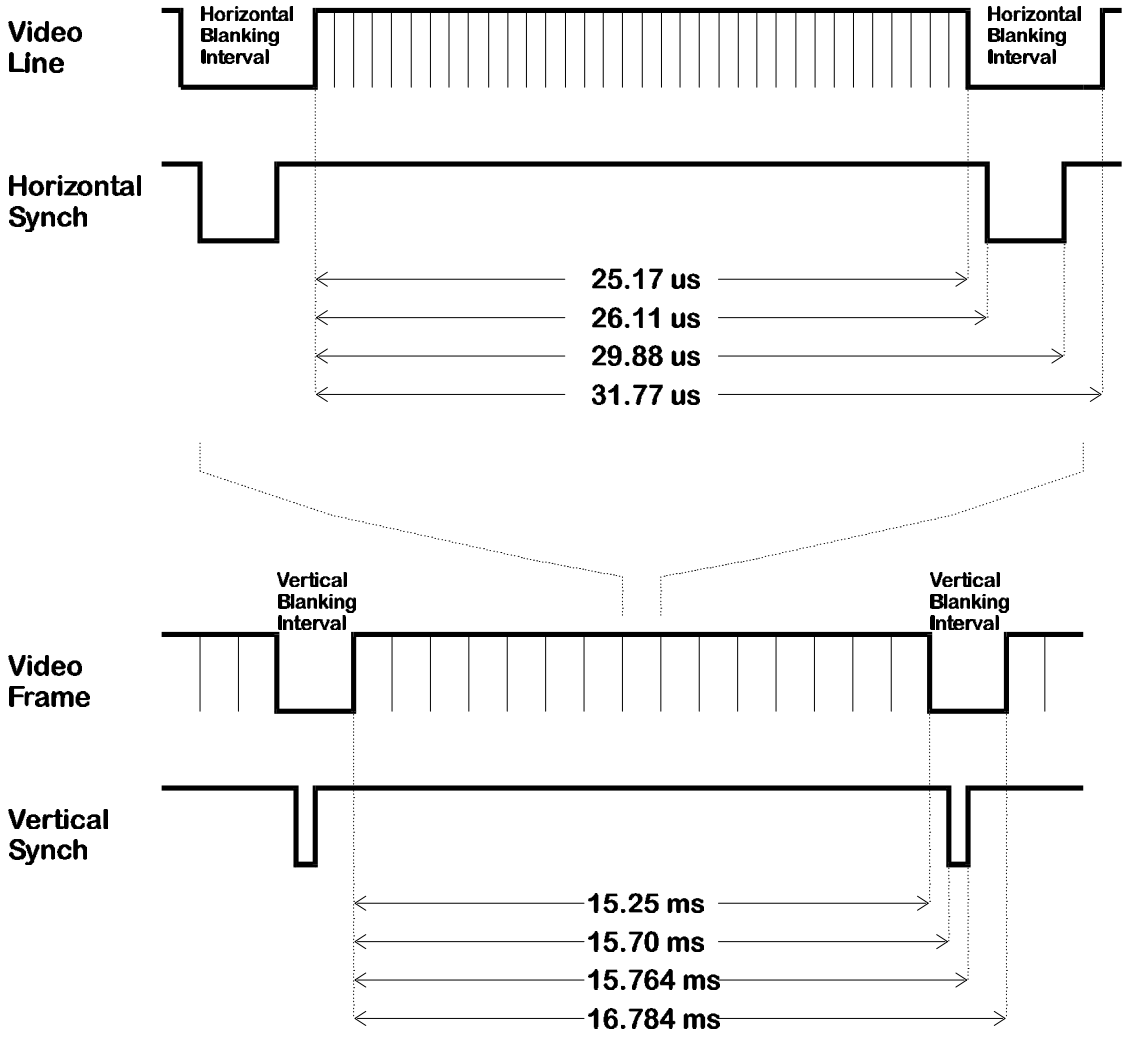


Figure 10: VGA signal timing.

4.2.3 VGA Signal Generator Algorithm

Now we have to figure out a process that will send pixels to the monitor with the correct timing and framing. We can store a picture in the RAM of the XS Board. Then we can retrieve the data from the RAM, format it into lines of pixels, and send the lines to the monitor with the appropriate pulses on the horizontal and vertical sync pulses.

The pseudocode for a single frame of this process is shown in **Listing 16**. The pseudocode has two outer loops: one which displays the *L* lines of visible pixels, and another which inserts the *V* blank lines and the vertical sync pulse. Within the first loop, there are two more loops: one which sends the *P* pixels of each video line to the monitor, and another which inserts the *H* blank pixels and the horizontal sync pulse.

Within the pixel display loop, there are statements to get the next byte from the RAM. Each byte contains four two-bit pixels. A small loop iteratively extracts each pixel to be displayed from the

lower two bits of the byte. Then the byte is shifted by two bits so the next pixel will be in the right position during the next iteration of the loop. Since it has only two bits, each pixel can store one of four colors. The mapping from the two-bit pixel value to the actual values required by the monitor electronics is done by the `COLOR_MAP()` routine.

Listing 16: VGA signal generation pseudocode.

```
/* send L lines of video to the monitor */
for line_cnt=1 to L
      /* send P pixels for each line */
      for pixel_cnt=1 to P
            /* get pixel data from the RAM */
            data = RAM(address)
            address = address + 1
            /* RAM data byte contains 4 pixels */
            for d=1 to 4
                  /* mask off pixel in the lower two bits */
                  pixel = data & 00000011
                  /* shift next pixel into lower two bits */
                  data = data>>2
                  /* get the color for the two-bit pixel */
                  color = COLOR_MAP(pixel)
                  send color to monitor
                  d = d + 1
            /* increment by four pixels */
            pixel_cnt = pixel_cnt + 4
      /* blank the monitor for H pixels */
      for horiz_blank_cnt=1 to H
            color = BLANK
            send color to monitor
            /* pulse the horizontal sync at the right time */
            if horiz_blank_cnt>HB0 and horiz_blank_cnt<HB1
                  hsync = 0
            else
                  hsync = 1
            horiz_blank_cnt = horiz_blank_cnt + 1
      line_cnt = line_cnt + 1
/* blank the monitor for V lines and insert vertical sync */
for vert_blank_cnt=1 to V
      color = BLANK
      send color to monitor
      /* pulse the vertical sync at the right time */
      if vert_blank_cnt>VB0 and vert_blank_cnt<VB1
            vsync = 0
      else
            vsync = 1
      vert_blank_cnt = vert_blank_cnt + 1
/* go back to start of picture in RAM */
address = 0
```

Figure 11 shows how to pipeline certain operations to account for delays in accessing data from the RAM. The pipeline has three stages:

Stage 1: The circuit uses the horizontal and vertical counters to compute the address where the next pixel is found in RAM. The counters are also used to determine the firing of the sync pulses and whether the video should be blanked. The pixel data from the RAM, blanking signal, and sync pulses are latched at the end of this stage so they can be used in the next stage.

Stage 2: The circuit uses the pixel data and the blanking signal to determine the binary color outputs. These outputs are latched at the end of this stage.

Stage 3: The binary color outputs are applied to the DAC, which sets the intensity levels for the monitor's color guns. The actual pixel is painted on the screen during this stage.

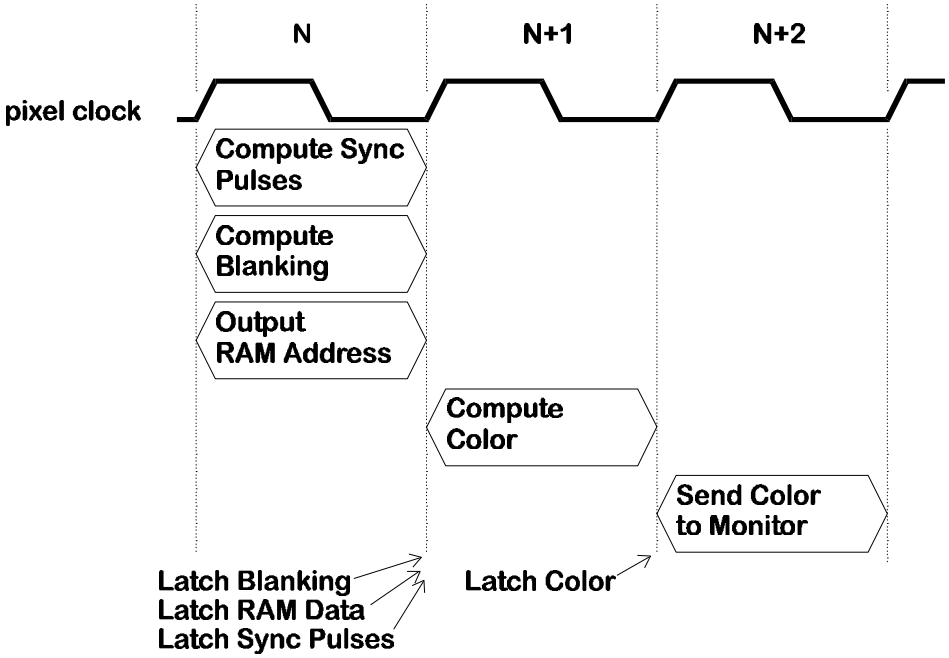


Figure 11: Pipelining of VGA signal generation tasks.

4.2.4 VGA Signal Generator in VHDL

The pseudocode and pipeline timing in the last section will help us to understand the VHDL code for a VGA signal generator shown in **Listing 17**. The inputs and outputs of the circuit as defined in the entity declaration are as follows:

clk: The input for the 12 MHz clock of the XS Board is declared here. This clock sets the maximum rate at which pixels can be sent to the monitor. The time interval within each line for transmitting viewable pixels is 25.17 μs, so this VGA generator circuit can only put a maximum of 25.17 ms × 12 MHz = 302 pixels on each line. For purposes of storing images in the RAM, it is convenient to reduce this to 256 pixels per line and blank the remaining 46 pixels. Half of these blank pixels are placed before the 256 viewable pixels and half are

placed after them on a line. This centers the viewable pixels between the left and right edges of the monitor screen.

reset: This line declares an input, which will reset all the other circuitry to a known state.

hsyncb, vsyncb: The outputs for the horizontal and vertical sync pulses are declared. The hsyncb output is declared as a buffer because it will also be referenced within the architecture section as a clock for the vertical line counter.

rgb: The outputs that control the red, green, and blue color guns of the monitor are declared here. Each gun is controlled by two bits, so there are four possible intensities for each color. Thus, this circuit can produce $4 \times 4 \times 4 = 64$ different colors.

address, data: These lines declare the outputs for driving the address lines of the RAM and the inputs for receiving the data from the RAM.

ceb, oeb, web: These are the declarations for the outputs which drive the chip-select, output-enable, and write-enable control lines of the RAM.

The preamble of the architecture section declares the following resources:

hcnt, vcnt: The counters that store the current horizontal position within a line of pixels and the vertical position of the line on the screen are declared on these lines. We will call these the horizontal or pixel counter, and the vertical or line counter, respectively. The line period is 31.77 μs that is 381 clock cycles, so the pixel counter needs at least nine bits of resolution. Each frame is composed of 528 video lines (only 480 are visible, the other 48 are blanked), so a ten bit counter is needed for the line counter.

pixrg: This is the declaration for the eight-bit register that stores the four pixels received from the RAM.

blank, pblank: This line declares the video blanking signal and its registered counterpart that is used in the next pipeline stage.

Within the main body of the architecture section, these following processes are executed:

inc_horiz_pixel_counter: This process describes the operation of the horizontal pixel counter. The counter is asynchronously set to zero when the reset input is high. The counter increments on the rising edge of each pixel clock. The range for the horizontal pixel counter is [0,380]. When the counter reaches 380, it rolls over to zero on the next cycle. Thus, the counter has a period of 381 pixel clocks. With a pixel clock of 12 MHz, this translates to a period of 31.75 μs.

inc_vert_line_counter: This process describes the operation of the vertical line counter. The counter is asynchronously set to zero when the reset input is high. The counter increments on the rising edge of the horizontal sync pulse after a line of pixels is completed. The range for the horizontal pixel counter is [0,527]. When the counter reaches 527, it rolls over to

zero on the next cycle. Thus, the counter has a period of 528 lines. Since the duration of a line of pixels is 31.75 μs, this makes the frame interval equal to 16.76 ms.

generate_horiz_sync: This process describes the operation of the horizontal sync pulse generator. The horizontal sync is set to its inactive high level when the reset is activated. During normal operations, the horizontal sync output is updated on every pixel clock. The sync signal goes low on the cycle after the pixel counter reaches 291 and continues until the cycle after the counter reaches 337. This gives a low horizontal sync pulse of (337-291)=46 pixel clocks. With a pixel clock of 12 MHz, this translates to a low-going horizontal sync pulse of 3.83 μs. The sync pulse starts 292 clocks after the line of pixels begin, which translates to 24.33 μs. This is less than the 26.11 μs we stated before. The difference of 1.78 ms translates to 21 pixel clocks. This time interval corresponds to the 23 blank pixels that are placed before the 256 viewable pixels (minus two clock cycles for pipelining delays).

generate_vert_sync: This process describes the operation of the vertical sync pulse generator. The vertical sync is set to its inactive high level when the reset is activated. During normal operations, the vertical sync output is updated after every line of pixels is completed. The sync signal goes low on the cycle after the line counter reaches 493 and continues until the cycle after the counter reaches 495. This gives a low vertical sync pulse of (495-493)= 2 lines. With a line interval of 31.75 μs, this translates to a low-going vertical sync pulse of 63.5 μs. The vertical sync pulse starts 494×31.75 μs = 15.68 ms after the beginning of the first video line.

Line 91: This line describes the computation of the combinatorial blanking signal. The video is blanked after 256 pixels on a line are displayed, or after 480 lines are displayed.

pipeline_blank: This process describes the operation of the pipelined video blanking signal. Within the process, the blanking signal is stored in a register so it can be used during the next stage of the pipeline when the color is computed.

Lines 104 -- 106: On these lines, the RAM is permanently selected and writing to the RAM is disabled. This makes the RAM look like a ROM, which stores video data. In addition, the outputs from the RAM are disabled when the video is blanked since there is no need for pixels during the blanking intervals. This isn't really necessary since no other circuit is trying to access the RAM.

Line 113: The address in RAM where the next four pixels are stored is calculated by concatenating the lower nine bits of the line counter with bits 7,6,5,4,3 and 2 of the pixel counter. With this arrangement, the line counter stores the address of one of $2^9 = 512$ pages. Each page contains $2^6 = 64$ bytes. Each byte contains four pixels, so each page stores one line of 256 pixels. The pixel counter increments through the bytes of a page to get the pixels for the current line. (Note that we don't need to use bits 1 and 0 of the pixel counter when computing the RAM address since each byte contains four pixels.) After the line is displayed, the line counter is incremented to point to the next page.

update_pixel_register: This process describes the operation of the register that holds the byte of pixel data read from RAM. The register is asynchronously cleared when the VGA circuit is reset. The register is updated on the rising edge of each pixel clock. The pixel register is loaded with data from the RAM whenever the lowest two bits of the pixel counter are both zero. The active pixel is always in the lower two bits of the register. Each pixel in the RAM data byte is shifted into the active position by right shifting the register two bits on each rising clock edge.

map_pixel_to_rgb: this process describes the process by which the current active pixel is mapped into the six bits that drive the red, green and blue color guns. The register is set to zero (which displays as the color black) when the reset input is high. The color register is clocked on the rising edge of the pixel clock since this is the rate at which new pixel values arrive. The value clocked into the register is a function of the pixel value and the blanking input. When the pipelined blanking input is low (inactive), the color displayed on the monitor is red, green, blue, or white depending upon whether the pixel value is 00, 01, 10, or 11, respectively. When the pipelined blanking input is high, the color register is loaded with zero (black).

Listing 17: VHDL code for a VGA generator.

```
001- LIBRARY IEEE;
002- USE IEEE.STD_LOGIC_1164.ALL;
003- USE IEEE.std_logic_unsigned.ALL;
004-
005- ENTITY vga_generator IS
006-  PORT
007-  (
008-    clk:      IN STD_LOGIC;     -- VGA dot clock
009-    reset:    IN STD_LOGIC;     -- asynchronous reset
010-    hsyncb:   OUT STD_LOGIC;    -- horizontal (line) sync
011-    vsyncb:   OUT STD_LOGIC;    -- vertical (frame) sync
012-    rgb:      OUT STD_LOGIC_VECTOR(5 DOWNTO 0); -- red,green,blue colors
013-    address:  OUT STD_LOGIC_VECTOR(14 DOWNTO 0);-- address into video RAM
014-    data:     IN STD_LOGIC_VECTOR(7 DOWNTO 0);  -- data from video RAM
015-    ceb:      OUT STD_LOGIC;    -- video RAM chip enable
016-    oeb:      OUT STD_LOGIC;    -- video RAM output enable
017-    web:      OUT STD_LOGIC     -- video RAM write enable
018-  );
019- END vga_generator;
020-
021- ARCHITECTURE vga_generator_arch OF vga_generator IS
022-   SIGNAL hcnt: STD_LOGIC_VECTOR(8 DOWNTO 0); -- horizontal pixel counter
023-   SIGNAL vcnt: STD_LOGIC_VECTOR(9 DOWNTO 0); -- vertical line counter
024-   SIGNAL pixrg: STD_LOGIC_VECTOR(7 DOWNTO 0); -- byte register for 4 pix
025-   SIGNAL blank: STD_LOGIC;     -- video blanking signal
026-   SIGNAL pblank: STD_LOGIC;    -- pipelined video blanking signal
027-   SIGNAL int_hsyncb: STD_LOGIC; -- internal horizontal sync.
028- BEGIN
029-
030-   inc_horiz_pixel_counter:
031-   PROCESS(clk,reset)
032-   BEGIN
033-     IF reset='1' THEN -- reset asynchronously clears pixel counter
034-       hcnt <= "000000000";
```

```
035-     ELSIF (clk'EVENT AND clk='1') THEN
036-       IF hcnt<380 THEN -- pixel counter resets after 381 pixels
037-         hcnt <= hcnt + 1;
038-       ELSE
039-         hcnt <= "000000000";
040-       END IF;
041-     END IF;
042-   END PROCESS;
043-
044-   inc_vert_line_counter:
045-   PROCESS(int_hsyncb,reset)
046-   BEGIN
047-     IF reset='1' THEN -- reset asynchronously clears line counter
048-       vcnt <= "0000000000";
049-     ELSIF (int_hsyncb'EVENT AND int_hsyncb='1') THEN
050-       IF vcnt<527 THEN -- vert. line counter rolls-over after 528 lines
051-         vcnt <= vcnt + 1;
052-       ELSE
053-         vcnt <= "0000000000";
054-       END IF;
055-     END IF;
056-   END PROCESS;
057-
058-   generate_horiz_sync:
059-   PROCESS(clk,reset)
060-   BEGIN
061-     IF reset='1' THEN -- reset asynchronously inactivates horiz sync
062-       int_hsyncb <= '1';
063-     ELSIF (clk'EVENT AND clk='1') THEN
064-       IF (hcnt>=291 AND hcnt<337) THEN
065-       -- horiz. sync is low in this interval to signal start of new line
066-         int_hsyncb <= '0';
067-       ELSE
068-         int_hsyncb <= '1';
069-       END IF;
070-     END IF;
071-     hsyncb <= int_hsyncb; -- output the horizontal sync signal
072-   END PROCESS;
073-
074-   generate_vert_sync:
075-   PROCESS(int_hsyncb,reset)
076-   BEGIN
077-     IF reset='1' THEN -- reset asynchronously inactivates vertical sync
078-       vsyncb <= '1';
079-     -- vertical sync is recomputed at the end of every line of pixels
080-     ELSIF (int_hsyncb'EVENT AND int_hsyncb='1') THEN
081-       IF (vcnt>=490 AND vcnt<492) THEN
082-       -- vert. sync is low in this interval to signal start of new frame
083-         vsyncb <= '0';
084-       ELSE
085-         vsyncb <= '1';
086-       END IF;
087-     END IF;
088-   END PROCESS;
089-
090-   -- blank video outside of visible region: (0,0) -> (255,479)
091-   blank <= '1' WHEN (hcnt>=256 OR vcnt>=480) ELSE '0';
092-   -- store the blanking signal for use in the next pipeline stage
093-   pipeline_blank:
094-   PROCESS(clk,reset)
```

```
095-   BEGIN
096-     IF reset='1' THEN
097-       pblank <= '0';
098-     ELSIF (clk'EVENT AND clk='1') THEN
099-       pblank <= blank;
100-     END IF;
101-   END PROCESS;
102-
103-   -- video RAM control signals
104-   ceb <= '0';      -- enable the RAM
105-   web <= '1';      -- disable writing to the RAM
106-   oeb <= blank;    -- enable the RAM outputs when video is not blanked
107-
108-   -- The video RAM address is built from the lower 9 bits of the vert
109-   -- line counter and bits 7-2 of the horizontal pixel counter.
110-   -- Each byte of the RAM contains four 2-bit pixels.  As an example,
111-   -- the byte at address ^h1234=^b0001,0010,0011,0100 contains the pixls
112-   -- at (row,col)=(^h048,^hD0),(^h048,^hD1),(^h048,^hD2),(^h048,^hD3).
113-   address <= vcnt(8 DOWNTO 0) & hcnt(7 DOWNTO 2);
114-
115-   update_pixel_register:
116-   PROCESS(clk,reset)
117-   BEGIN
118-     IF reset='1' THEN -- clear the pixel register on reset
119-       pixrg <= "00000000";
120-     -- pixel clock controls changes in pixel register
121-     ELSIF (clk'EVENT AND clk='1') THEN
122-       -- the pixel register is loaded with the contents of the video
123-       -- RAM location when the lower two bits of the horiz. counter
124-       -- are both zero.  The active pixel is in the lower two bits
125-       -- of the pixel register.  For the next 3 clocks, the pixel
126-       -- register is right-shifted by two bits to bring the other
127-       -- pixels in the register into the active position.
128-       IF hcnt(1 DOWNTO 0)="00" THEN
129-         pixrg <= data;   -- load 4 pixels from RAM
130-       ELSE
131-         pixrg <= "00" & pixrg(7 DOWNTO 2); -- right-shift pixel register
132-       END IF;
133-     END IF;
134-   END PROCESS;
135-
136-   -- the color mapper translates each 2-bit pixel into a 6-bit
137-   -- color value.  When the video signal is blanked, the color
138-   -- is forced to zero (black).
139-   map_pixel_to_rgb:
140-   PROCESS(clk,reset)
141-   BEGIN
142-     IF reset='1' THEN    -- blank the video on reset
143-       rgb <= "000000";
144-     ELSIF (clk'EVENT AND clk='1') THEN -- update the color every clock
145-       -- map the pixel to a color if the video is not blanked
146-       IF pblank='0' THEN
147-         CASE pixrg(1 DOWNTO 0) IS
148-           WHEN "00"   => rgb <= "110000";  -- red
149-           WHEN "01"   => rgb <= "001100";  -- green
150-           WHEN "10"   => rgb <= "000011";  -- blue
151-           WHEN OTHERS => rgb <= "111111";  -- white
152-         END CASE;
153-       ELSE   -- otherwise, output black if the video is blanked
154-         rgb <= "000000"; -- black
```

```
155-       END IF;
156-     END IF;
157-   END PROCESS;
158-
159- END vga_generator_arch;
```

Listing 18: XS40 UCF file for the VGA signal generator.

```
001- net clk            loc=p13;
002- net reset          loc=p44;
003- net data<0>        loc=p41;
004- net data<1>        loc=p40;
005- net data<2>        loc=p39;
006- net data<3>        loc=p38;
007- net data<4>        loc=p35;
008- net data<5>        loc=p81;
009- net data<6>        loc=p80;
010- net data<7>        loc=p10;
011- net address<0>     loc=p3;
012- net address<1>     loc=p4;
013- net address<2>     loc=p5;
014- net address<3>     loc=p78;
015- net address<4>     loc=p79;
016- net address<5>     loc=p82;
017- net address<6>     loc=p83;
018- net address<7>     loc=p84;
019- net address<8>     loc=p59;
020- net address<9>     loc=p57;
021- net address<10>    loc=p51;
022- net address<11>    loc=p56;
023- net address<12>    loc=p50;
024- net address<13>    loc=p58;
025- net address<14>    loc=p60;
026- net ceb            loc=p65;
027- net web            loc=p62;
028- net oeb            loc=p61;
029- net rgb<0>         loc=p25;
030- net rgb<1>         loc=p26;
031- net rgb<2>         loc=p24;
032- net rgb<3>         loc=p20;
033- net rgb<4>         loc=p23;
034- net rgb<5>         loc=p18;
035- net hsyncb         loc=p19;
036- net vsyncb         loc=p67;
```

Listing 19: XS95 UCF file for the VGA signal generator.

```
001- net clk            loc=p9;
002- net reset          loc=p46;
003- net data<0>        loc=p44;
004- net data<1>        loc=p43;
005- net data<2>        loc=p41;
006- net data<3>        loc=p40;
007- net data<4>        loc=p39;
008- net data<5>        loc=p37;
009- net data<6>        loc=p36;
010- net data<7>        loc=p35;
011- net address<0>     loc=p75;
012- net address<1>     loc=p79;
```

```
013- net address<2>     loc=p82;
014- net address<3>     loc=p84;
015- net address<4>     loc=p1;
016- net address<5>     loc=p3;
017- net address<6>     loc=p83;
018- net address<7>     loc=p2;
019- net address<8>     loc=p58;
020- net address<9>     loc=p56;
021- net address<10>    loc=p54;
022- net address<11>    loc=p55;
023- net address<12>    loc=p53;
024- net address<13>    loc=p57;
025- net address<14>    loc=p61;
026- net ceb            loc=p65;
027- net web            loc=p63;
028- net oeb            loc=p62;
029- net rgb<0>         loc=p21;
030- net rgb<1>         loc=p23;
031- net rgb<2>         loc=p19;
032- net rgb<3>         loc=p17;
033- net rgb<4>         loc=p18;
034- net rgb<5>         loc=p14;
035- net hsyncb         loc=p15;
036- net vsyncb         loc=p24;
```

The steps for compiling and testing the VGA design using an XS40 combined with an XStend Board are as follows:

- Synthesize the VHDL code in the VGA40\VGA.VHD file for an XC4005XL FPGA.
- Compile the synthesized netlist using the VGA40.UCF constraint file (**Listing 18**).
- Mount an XS40 Board in the XStend Board and attach the downloading cable from the XS40 to the PC parallel port. Apply 9VDC though jack J9 of the XS40. Place shunts on jumpers J4, J7, and J8 of the XStend Board to enable the LED displays. Remove the shunt on jumper J17 to keep the XStend codec serial output from interfering with the DIP switch logic levels. Set all the DIP switches to the OPEN position.
- Attach a VGA monitor to the DB-HD15 connector (J5).
- Download the VGA40.BIT file and a video test pattern into the XS40/XStend combination with the command: `XSLOAD TESTPATT.HEX VGA40.BIT`.
- Release the reset to the VGA circuitry with the command: `XSPORT 0`.
- Observe the color bars on the monitor screen.

The steps for compiling and testing the design using an XS95 combined with an XStend Board are as follows:

- Synthesize the VHDL code in the VGA95\VGA.VHD file for an XC95108 CPLD.

- Compile the synthesized netlist using the VGA95.UCF constraint file (**Listing 19**).
- Generate an SVF file for the design.
- Mount an XS95 Board in the XStend Board and attach the downloading cable from the XS95 to the PC parallel port. Apply 9VDC though jack J9 of the XS40. Place shunts on jumpers J4, J7, and J8 of the XStend Board to enable the LED displays. Remove the shunt on jumper J17 to keep the XStend codec serial output from interfering. Set all the DIP switches to the OPEN position.
- Attach a VGA monitor to the DB-HD15 connector (J5).
- Download the VGA95.SVF file and a video test pattern into the XS95/XStend combination with the command: `XSLOAD TESTPATT.HEX VGA95.SVF`.
- Release the reset to the VGA circuitry with the command: `XSPORT 0`.
- Observe the color bars on the monitor screen.

4.3 Reading Keyboard Scan Codes Through the PS/2 Interface

This example creates a circuit that accepts scan codes from a keyboard attached to the PS/2 interface of the XStend Board. The binary pattern of the scan code is displayed on the bargraph LEDs. In addition, if a scan code for one of the keys '0'—'9' arrives, then the numeral will be displayed on the right LED display of the XStend Board.

The format of the scan code transmissions from the keyboard are shown in **Figure 12**. The keyboard electronics drive the clock and data lines. The start of a scan code transmission is indicated by a low level on the data line on the falling edge of the clock. The eight bits of the scan code follow (starting with the least-significant bit) on successive falling clock edges. These are followed by an odd-parity bit and then a high-level stop bit.

When the clock line goes high after the stop bit, the receiver (in this case, the FPGA or CPLD on the XS Board inserted in the XStend Board) can pull the clock line low to inhibit any further transmissions. After the clock line is released and it returns to a high level, the keyboard can send another scan code. If the receiver never pulls the clock line low, then the keyboard will send scan codes whenever a key is pressed.

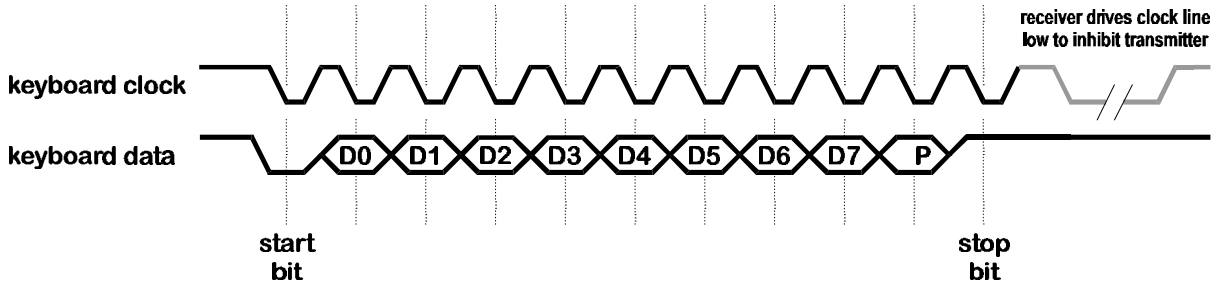


Figure 12: Keyboard data transmission waveforms.

The VHDL code for this example is shown in

Listing 20. The inputs and outputs of the circuit as defined in the entity declaration are as follows:

rst: This output drives the reset pin of the microcontroller on the XS Board.

oeb: This output drives the output-enable pin of the RAM on the XS Board.

kb_data: The scan code bits enter through this input.

kb_clk: The keyboard clock signal enters through this input.

db: These outputs drive the segments of the bargraph LED on the XStend Board.

rsb: These outputs drive the segments of the right LED digit on the XStend Board.

Within the main body of the architecture section, these operations occur:

Lines 22 & 23: The microcontroller reset pin and the RAM output-enable pin are driven high so these chips cannot interfere while receiving data from the keyboard.

Lines 25 & 26: The keyboard clock passes through an input buffer and then a global clock buffer before it reaches the rest of the circuitry. (These buffers are declared on lines 18 and 19, respectively.) The global clock buffer distributes the clock signal with minimal skew in the XS40 Board FPGA. These statements are not used with the CPLD in the XS95 Board.

gather_scancode: On every falling edge of kb_clk, this process shifts the data bit on the kb_data input into the most-significant bit of a 10-bit shift register. After 11 clock cycles, the lower 8 bits of the register will contain the scan code, the upper 2 bits will store the stop and parity bits, and the start bit will have been shifted through the entire register and discarded.

Line 38: The value in the shift register is inverted and applied to the segments of the LED bargraph. Since the bargraph segments are active-low, a segment will light for every '1' bit in the shift register. The LED segment drivers are not registered so there will be some flickering as the shift register contents change.

Lines 40-51: If the scan code in the shift register matches the codes for the digits 0-9, then the right LED digit segments will be activated to display the corresponding digit. If the scan code does not match one of these codes, the letter 'E' is displayed.

The steps for compiling and testing the design using an XS40 combined with an XStend Board are as follows:

- Synthesize the VHDL code in the KEYBRD40\KEYBRD.VHD for an XC4005XL FPGA.
- Compile the synthesized netlist using the KEYBRD40.UCF constraint file (

Listing 21).

- Mount an XS40 Board in the XStend Board and attach the downloading cable from the XS40 to the PC parallel port. Apply 9VDC though jack J9 of the XS40. Place shunts on jumpers J4, J7, and J8 to enable the LEDs. Remove the shunt on jumper J17 to keep the XStend codec from interfering. Set all the DIP switches to the OPEN position.

- Attach a keyboard to the PS/2 connector of the XStend Board.

- Download the KEYBRD40.BIT file into the XS40/XStend combination with the command: `XSLOAD KEYBRD40.BIT`.

- Press keys on the keyboard and observe the results on the LED displays.

The steps for compiling and testing the design using an XS95 combined with an XStend Board are as follows:

- Synthesize the VHDL code in the KEYBRD95\KEYBRD.VHD for an XC95108 CPLD.

- Compile the synthesized netlist using the KEYBRD95.UCF constraint file (**Listing 23**).

- Generate an SVF file for the design.

- Mount an XS95 Board in the XStend Board and attach the downloading cable from the XS95 to the PC parallel port. Apply 9VDC though jack J9 of the XS95. Place shunts on jumpers J4, J7, and J8 to enable the LEDs. Remove the shunt on jumper J17 to keep the XStend codec from interfering. Set all the DIP switches to the OPEN position.

- Download the KEYBRD95.SVF file into the XS95/XStend combination with the command: `XSLOAD KEYBRD95.SVF`.

- Press keys on the keyboard and observe the results on the LED displays.

Listing 20: VHDL code for receiving keyboard scan codes from the PS/2 interface.

```
001- LIBRARY IEEE;
002- USE IEEE.STD_LOGIC_1164.ALL;
003-
004- ENTITY kbd_read IS
005-   PORT
006-   (
007-     rst: OUT STD_LOGIC;        -- uC reset
008-     oeb: OUT STD_LOGIC;        -- RAM output enable
009-     kb_data: IN STD_LOGIC;     -- serial data from the keyboard
010-     kb_clk: IN STD_LOGIC;      -- clock from the keyboard
011-     db: OUT STD_LOGIC_VECTOR(8 DOWNTO 1);  -- bargraph LED
012-     rsb: OUT STD_LOGIC_VECTOR(6 DOWNTO 0)  -- right LED digit
013-   );
014- END kbd_read;
015-
016- ARCHITECTURE kbd_read_arch OF kbd_read IS
017- SIGNAL scancode: STD_LOGIC_VECTOR(9 DOWNTO 0);
018- COMPONENT ibuf PORT(i: IN STD_LOGIC; o: OUT STD_LOGIC); END COMPONENT;
019- COMPONENT bufg PORT(i: IN STD_LOGIC; o: OUT STD_LOGIC); END COMPONENT;
020- SIGNAL buf_clk0, buf_clk1: STD_LOGIC;
021- BEGIN
022-   rst <= '1';      -- keep the uC in the reset state
023-   oeb <= '1';      -- disable the RAM output drivers
024-
025-   b0: ibuf PORT MAP(i=>kb_clk,o=>buf_clk0);   -- buffer the clock from
026-   b1: bufg PORT MAP(i=>buf_clk0,o=>buf_clk1); -- the keyboard
027-
028-   -- shift keyboard data into the MSb of the scancode register
029-   -- on the falling edge of the keyboard clock
030-   gather_scancode:
031-   PROCESS(buf_clk1,scancode)
032-   BEGIN
033-     IF(buf_clk1'EVENT AND buf_clk1='0') THEN
034-       scancode <= kb_data & scancode(9 DOWNTO 1);
035-     END IF;
036-   END PROCESS;
037-
038-   db <= NOT(scancode(7 DOWNTO 0)); -- show the scancode on the bargraph
039-
040-   -- display the key that was pressed on the right LED digit
041-  rsb <=  "1101101" WHEN scancode(7 DOWNTO 0)="00010110" ELSE -- 1
042-          "0100010" WHEN scancode(7 DOWNTO 0)="00011110" ELSE -- 2
043-          "0100100" WHEN scancode(7 DOWNTO 0)="00100110" ELSE -- 3
044-          “1000101" WHEN scancode(7 DOWNTO 0)="00100101" ELSE -- 4
045-          "0010100" WHEN scancode(7 DOWNTO 0)="00101110" ELSE -- 5
046-          "0010000" WHEN scancode(7 DOWNTO 0)="00110110" ELSE -- 6
047-          "0101101" WHEN scancode(7 DOWNTO 0)="00111101" ELSE -- 7
048-          "0000000" WHEN scancode(7 DOWNTO 0)="00111110" ELSE -- 8
049-          "0000100" WHEN scancode(7 DOWNTO 0)="01000110" ELSE -- 9
050-          "0001000" WHEN scancode(7 DOWNTO 0)="01000101" ELSE -- 0
051-          "0010010";                                          -- E
052- END kbd_read_arch;
```

Listing 21: XS40 UCF file for the PS/2 keyboard interface.

```
001- net rst      loc=p36;
002- net oeb      loc=p61;
003- net kb_data  loc=p69;
004- net kb_clk   loc=p68;
005- net rsb<0>   loc=p59;
006- net rsb<1>   loc=p57;
007- net rsb<2>   loc=p51;
008- net rsb<3>   loc=p56;
009- net rsb<4>   loc=p50;
010- net rsb<5>   loc=p58;
011- net rsb<6>   loc=p60;
012- net db<1>    loc=p41;
013- net db<2>    loc=p40;
014- net db<3>    loc=p39;
015- net db<4>    loc=p38;
016- net db<5>    loc=p35;
017- net db<6>    loc=p81;
018- net db<7>    loc=p80;
019- net db<8>    loc=p10;
```

Listing 23: XS95 UCF file for the PS/2 keyboard interface.

```
001- net rst      loc=p45;
002- net oeb      loc=p62;
003- net kb_data  loc=p70;
004- net kb_clk   loc=p26;
005- net rsb<0>   loc=p58;
006- net rsb<1>   loc=p56;
007- net rsb<2>   loc=p54;
008- net rsb<3>   loc=p55;
009- net rsb<4>   loc=p53;
010- net rsb<5>   loc=p57;
011- net rsb<6>   loc=p61;
012- net db<1>    loc=p44;
013- net db<2>    loc=p43;
014- net db<3>    loc=p41;
015- net db<4>    loc=p40;
016- net db<5>    loc=p39;
017- net db<6>    loc=p37;
018- net db<7>    loc=p36;
019- net db<8>    loc=p35;
```

4.4 Inputing and Outputing Stereo Signals Through the Codec

The stereo codec on the XStend Board is capable of digitizing two analog signals to 20 bits of resolution while simultaneously generating two analog signals from 20-bit values. A high-level view of the codec chip is shown on the right-half of **Figure 13**. Two analog inputs (which are tpically the left and right channels of a stereo audio signal) enter the codec and are digitized into two 20-bit values by analog-to-digital converters (ADCs). These values are loaded into shift registers which are shifted out of a single pin of the codec under control of a shift clock and a left/right channel selector control input. At the same time, 20-bit values are alternately shifted into two shift registers in the codec which feed digital-to-analog converters (DACs) that drive two analog outputs. Signals on these outputs are typically the left and right channels of a stereo audio signal.

If the FPLD is handling these values in a bit-parallel manner, then the FPLD must contain a set of shift registers which convert the serial input stream into 20-bit values and another set which converts 20-bit values into a serial output stream. This is shown in the left-half of **Figure 13**. The gating of these shift registers onto the serial input and output pins is synchronized with the same left/right channel select signal used by the codec chip.

In addition to the shift registers, the FPLD needs circuitry to read and write them and to indicate when they are full and empty. Since the codec ADCs and DACs generate and consume data at a set sample rate, it is also necessary to build circuitry which detects overflow and underflow of the FPLD shift registers if they are not read or written in time.

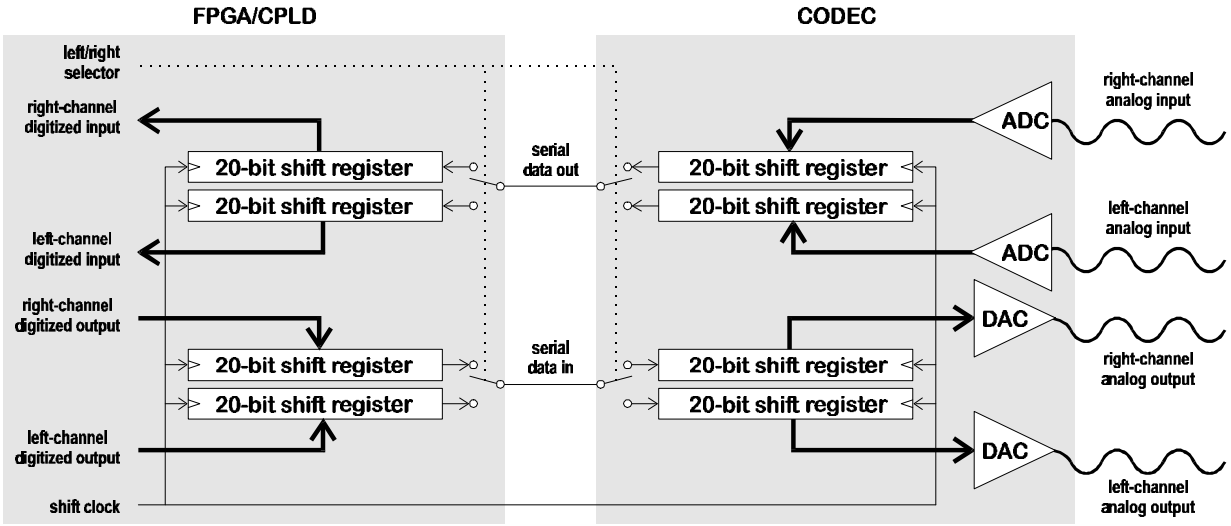


Figure 13: Connections between the XStend codec chip and the XS Board FPGA or CPLD.

The FPLD circuitry can be decomposed into three modules:

- a clock generator module which outputs the serial data shift clock and the left/right channel select signals;

- a channel module which contains the shift registers, buffers, read/write control, and overflow/underflow detection circuitry for a single input/output stream of data;

- a top-level module which combines the clock generator module with two channel modules to form a complete codec interface circuit.

The VHDL code for the clock generator module is detailed in **Listing 24**. The inputs and outputs of the clock generator as defined in the entity declaration are as follows:

clk: This is the main clock input which is typically the 12 MHz clock from the XS Board.

reset: This input synchronously resets the counter the clock generator.

mclk: This output is the master clock for the codec chip.

sclk: This output is the clock for synchronizing serial data transfers between the FPLD and the codec.

lrck: This output controls the activation of the left and right channel circuitry in the codec and the FPLD.

bit_cntr: These outputs indicate the current bit being transmitted and received in the serial data streams.

subcycle_cntr: The duration of each serial data bit is divided into four phases and these outputs indicate the current phase.

Within the main body of the clock generator architecture section, these operations occur:

gen_clock: This process increments the sequencing counter and toggles the left/right channel selector when the count reaches the duration for which a channel is active. The codec chip requires that the channel duration be either 128, 192, or 256 master clock periods in length. Thus, the total time to handle both channels is 256, 384, or 512 clock periods. This sets the sampling rate. So using a channel duration of 128 with a 12 MHz clock gives a sampling rate of 46.875 KHz that is sufficient for audio.

Lines 45-47: The various clocks are output on these lines. The master clock and left/right selector have already been discussed. The serial data shift clock is one-quarter of the master clock. So transmitting or receiving a 20-bit value will require $4 \times 20 = 80$ clock periods, and this will fit within the shortest possible channel duration.

Line 48: The position of the current data bit in the serial stream for a channel is output here. Since each bit has a duration of four clock periods, the position of the bit in the stream is just the sequence counter with the two least-significant bits removed.

Line 49: The position within a bit is output on this line. This is given by the two least-significant bits of the sequence counter.

Listing 24: VHDL code for the codec clock generator module.

```
001- LIBRARY IEEE,codec;
002- USE IEEE.STD_LOGIC_1164.ALL;
003- USE IEEE.std_logic_unsigned.ALL;
004- USE codec.codec.ALL;
005-
006- ENTITY clkgen IS
007-   GENERIC
008-   (
009-     channel_duration: POSITIVE := 128      -- must be 128, 192, or 256
010-   );
011-   PORT
012-   (
013-     -- interface I/O signals
014-     clk: IN STD_LOGIC;   -- clock input
015-     reset: IN STD_LOGIC; -- synchronous active-high reset
016-     -- codec chip clock signals
017-     mclk: OUT STD_LOGIC; -- master clock output to codec
018-     sclk: OUT STD_LOGIC; -- serial data clock to codec
019-     lrck: OUT STD_LOGIC; -- left/right codec channel select
020-     bit_cntr: OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
021-     subcycle_cntr: OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
022-   );
023- END clkgen;
024-
025- ARCHITECTURE clkgen_arch OF clkgen IS
026- SIGNAL lrck_int: STD_LOGIC;
027- SIGNAL seq: STD_LOGIC_VECTOR(7 DOWNTO 0);
028- BEGIN
029-   gen_clock:
030-   PROCESS(clk,seq,lrck_int)
031-   BEGIN
032-     IF (clk'EVENT AND clk='1') THEN
033-       IF(reset=yes) THEN -- synchronous reset
034-         seq <= (OTHERS=>'0');
035-         lrck_int <= left;  -- start with left channel of codec
036-       ELSIF(seq=channel_duration-1) THEN
037-         seq <= (OTHERS=>'0');  -- reset seq every channel period
038-         lrck_int <= NOT(lrck_int);  -- toggle chan select every period
039-       ELSE
040-         seq <= seq+1; -- normally, just inc the sequencer
041-         lrck_int <= lrck_int; -- don’t change channel selector
042-       END IF;
043-     END IF;
044-   END PROCESS;
045-   lrck <= lrck_int;  -- output the channel selector to the codec
046-   mclk <= clk;       -- codec master clock equals input clock
047-   sclk <= seq(1);    -- serial shift clock is 1/4 of the master clock
048-   bit_cntr <= seq(7 DOWNTO 2); -- which bit period is being processed
049-   subcycle_cntr <= seq(1 DOWNTO 0); -- position within bit
050- END clkgen_arch;
```

The VHDL code for the channel module is shown in **Listing 25**. The inputs and outputs of the clock generator as defined in the entity declaration are as follows:

clk: This is the main clock input which is typically the 12 MHz clock from the XS Board.

reset: This input synchronously resets the channel.

chan_on: A high level on this input activates the channel. This input is usually controlled by the left/right channel selector.

bit_cntr: These inputs inform the channel of the index of the serial data bit currently being transmitted and received.

chan_sel: A high level on this input enables the interface that lets the shift registers be read and written. (Note that despite its name, this input is *not* controlled by the left/right channel selector.)

rd: A high level on this input outputs the value stored in the shift register connected to the ADC.

wr: A high level on this input writes a new value into the shift register connected to the DAC.

adc_out: The bits stored in the ADC shift register are read out in parallel through these outputs..

dac_in: The DAC shift register is loaded in parallel with bits passed through these inputs.

adc_out_rdy: This output goes high after all the bits have been shifted from the codec into the ADC shift register.

adc_overrun: This output goes high if new serial data is shifted into the ADC shift register before the old contents have been read out through the parallel outputs.

dac_in_rdy: This output goes high after all the bits in the DAC shift register have been shifted over to the codec.

dac_underrun: This output goes high if the DAC shift register starts shifting data over to the codec before it has been written through the parallel inputs.

sdin: The serial data stream for the codec DAC is shifted out through this output. (Note that this output takes its name from the pin it is connected to on the codec chip; it is *not* an input.)

sdout: The serial data stream from the codec ADC is shifted in through this input. (Note that this input takes its name from the pin it is connected to on the codec chip; it is *not* an output.)

Within the main body of the channel module architecture section, these operations occur:

rcv_adc: This process receives serial data from the ADC in the codec. The ADC shift register is cleared upon reset and a flag is set which indicates the shift register does not contain all the bits from the ADC. Once the reset is removed and the channel is active, bits are shifted into

the register during the second subcycle of each bit period (the subcycles are numbered 0, 1, 2 and 3). Accepting data on the second subcycle gives the serial data bit plenty of time to stabilize. The first bit of the serial data (when the bit counter equals 0) contains no data and is discarded. Then bits 1,2,..., up to the width of the ADC data value are pushed into the shift register. Then the shifting stops. The shift register is marked as 'not full' as soon as a single bit is shifted in so that the value will not be inadvertently read. The shift register status changes to full as soon as the last bit enters the shift register.

Line 66: The contents of the shift register are output in a parallel format on this line. These outputs are not latched and will change as bits are shifted into the register.

Line 69: A flag is maintained that indicates whether the contents of the ADC shift register have been read. The flag is set when the ADC register for the channel is full and it is selected for a read operation. The flag will stay set after the read operation is complete. Reading the register does not empty it. The shift register is no longer full only when the first bit of the next sample is shifted into it. This will reset the read flag.

read_adc: This process updates the flag that indicates whether the ADC shift register has been read.

Lines 84—85: A status output is asserted when the data in the ADC shift register is ready for reading. Reads are permitted when the register is full and has not yet been read. This output is cleared as soon as a read occurs or new data is shifted into the register.

detect_adc_overrun: This process monitors the ADC shift register and flags an error condition if the register begins accepting bits from the current sample period but the data from the previous period has not yet been read.

tx_dac: This process transmits serial data to the DAC in the codec. The DAC shift register is cleared upon reset and a flag is set which indicates the shift register contains no bits for the DAC. After the reset is removed, the register can be loaded in parallel if the channel is selected for a write operation. If no write operation is in process but the channel is active, then data is shifted out to the codec on the second subcycle. (This gives the data some hold time so the codec chip can clock it in reliably.) No data is output during the first bit period because the codec discards this bit, but a flag is set which indicates the register is no longer empty and a serial transmission is in process. Then bits 1,2,..., up to the width of the DAC data value are shifted out. As the last bit is output, the flag is set to show the shift register is now empty.

Line 124: The DAC serial data input of the codec chip is driven by the most-significant bit of the DAC shift register.

Line 127: A flag is maintained that indicates whether the DAC shift register has been written. The flag is set when the DAC register for the channel is empty and it is selected for a write operation. The flag will stay set after the write operation is complete. Writing the register

does not fill it. The shift register is full only when the first bit of the next sample period is shifted out of it. This will reset the write flag.

write_dac: This process updates the flag that indicates whether the DAC shift register has been written.

Lines 142—143: A status output is asserted when the DAC shift register is ready to be written with new input data. Writes are permitted when the register is empty and has not yet been written. This output is cleared as soon as a write occurs or when data bits start shifting out of the register.

detect_dac_underrun: This process monitors the DAC shift register and flags an error condition if the register starts shifting out data but has not yet been written with a new data value for the current sample period.

Listing 25: VHDL code for the codec channel module.

```
001- LIBRARY IEEE,codec;
002- USE IEEE.STD_LOGIC_1164.ALL;
003- USE IEEE.std_logic_unsigned.ALL;
004- USE codec.codec.ALL;
005-
006- ENTITY channel IS
007-   GENERIC
008-   (
009-     dac_width: POSITIVE := 20;
010-     adc_width: POSITIVE := 20
011-   );
012-   PORT
013-   (
014-     -- interface I/O signals
015-     clk: IN STD_LOGIC;    -- clock input
016-     reset: IN STD_LOGIC;  -- synchronous active-high reset
017-     chan_on: IN STD_LOGIC;
018-     bit_cntr: IN STD_LOGIC_VECTOR(5 DOWNTO 0);
019-     subcycle_cntr: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
020-     chan_sel: IN STD_LOGIC;  -- select left/right codec chan for rd/wr
021-     rd: IN STD_LOGIC;        -- read from the codec ADC
022-     wr: IN STD_LOGIC;        -- write to the codec DAC
023-     adc_out: OUT STD_LOGIC_VECTOR(adc_width-1 DOWNTO 0);-- frm codec ADC
024-     dac_in: IN STD_LOGIC_VECTOR(dac_width-1 DOWNTO 0);  -- to codec DAC
025-     adc_out_rdy: OUT STD_LOGIC;  -- ADC output is ready to be read
026-     adc_overrun: OUT STD_LOGIC;  -- output from ADC channel overwritten
027-     dac_in_rdy: OUT STD_LOGIC;   -- DAC input is ready to be written
028-     dac_underrun: OUT STD_LOGIC; -- input to DAC did not arrive in time
029-     -- codec chip I/O signals
030-     sdin: OUT STD_LOGIC;  -- serial output to codec DAC
031-     sdout: IN STD_LOGIC   -- serial input from codec ADC
032-   );
033- END channel;
034-
035- ARCHITECTURE channel_arch OF channel IS
036- SIGNAL dac_shfreg: STD_LOGIC_VECTOR(dac_width-1 DOWNTO 0);
037- SIGNAL dac_empty: STD_LOGIC;  -- DAC shift register is empty
038- SIGNAL dac_wr: STD_LOGIC;     -- DAC channel has been written
```

```
039- SIGNAL dac_wr_nxt: STD_LOGIC; -- DAC channel has been written
040- SIGNAL dac_in_rdy_int: STD_LOGIC; -- int. ver. DAC is ready for input
041- SIGNAL adc_shfreg: STD_LOGIC_VECTOR(adc_width-1 DOWNTO 0);
042- SIGNAL adc_full: STD_LOGIC;   -- ADC shift register is full
043- SIGNAL adc_rd: STD_LOGIC;     -- ADC channel has been read
044- SIGNAL adc_rd_nxt: STD_LOGIC; -- the ADC channel has been read
045- SIGNAL adc_out_rdy_int: STD_LOGIC;  -- int. ver. ADC ready for output
046- BEGIN
047-   -- receives data from codec ADC
048-   rcv_adc:
049-   PROCESS(clk,chan_on,subcycle_cntr,bit_cntr,adc_shfreg,sdout)
050-   BEGIN
051-     IF(clk'EVENT AND (clk=yes)) THEN
052-       IF(reset='1') THEN
053-         adc_shfreg <= (OTHERS=>'0');
054-         adc_full <= no;
055-       ELSIF((chan_on=yes) AND (subcycle_cntr=1)) THEN
056-         IF((bit_cntr>=1) AND (bit_cntr<adc_width)) THEN
057-           adc_full <= no;
058-           adc_shfreg <= adc_shfreg(adc_width-2 DOWNTO 0) & sdout;
059-         ELSIF(bit_cntr=adc_width) THEN
060-           adc_full <= yes;
061-           adc_shfreg <= adc_shfreg(adc_width-2 DOWNTO 0) & sdout;
062-         END IF;
063-       END IF;
064-     END IF;
065-   END PROCESS;
066-   adc_out <= adc_shfreg;
067-
068-   -- handle reading of ADC data from codec interface
069-   adc_rd_nxt <= yes WHEN (adc_full=yes AND chan_sel=yes AND rd=yes) OR
070-                          (adc_full=yes AND adc_rd=yes)
071-             ELSE no;
072-   read_adc:
073-   PROCESS(clk,adc_rd_nxt)
074-   BEGIN
075-     IF(clk'EVENT AND clk='1') THEN
076-       IF(reset=yes) THEN
077-         adc_rd <= no;
078-       ELSE
079-         adc_rd <= adc_rd_nxt;
080-       END IF;
081-     END IF;
082-   END PROCESS;
083-   -- ADC data is ready for reading if reg is full and not read yet
084-   adc_out_rdy_int <= yes WHEN adc_full=yes AND adc_rd=no ELSE no;
085-   adc_out_rdy <= adc_out_rdy_int;
086-
087-   -- detect and signal overwriting of data from the codec ADC channels
088-   detect_adc_overrun:
089-   PROCESS(clk,reset,bit_cntr,chan_on,adc_out_rdy_int)
090-   BEGIN
091-     IF(clk'EVENT AND clk='1') THEN
092-       IF(reset=yes) THEN
093-         adc_overrun <= no;
094-       ELSIF(bit_cntr=1 AND chan_on=yes AND adc_out_rdy_int=yes) THEN
095-         adc_overrun <= yes;
096-       END IF;
097-     END IF;
098-   END PROCESS;
```

```
099-
100-   -- transmits data to codec DAC
101-   tx_dac:
102-   PROCESS(clk,reset,chan_on,subcycle_cntr,bit_cntr,dac_shfreg)
103-   BEGIN
104-     IF(clk'EVENT AND clk='1') THEN
105-       IF(reset=yes) THEN
106-         dac_shfreg <= (OTHERS=>'0');
107-         dac_empty <= yes;
108-       ELSIF(chan_sel=yes AND wr=yes) THEN
109-         dac_shfreg <= dac_in;
110-       ELSIF(chan_on=yes AND subcycle_cntr=1) THEN
111-         IF(bit_cntr=0) THEN
112-           dac_empty <= no;
113-         ELSIF(bit_cntr<dac_width) THEN
114-           dac_shfreg <= dac_shfreg(dac_width-2 DOWNTO 0) & '0';
115-         ELSIF(bit_cntr=dac_width) THEN
116-           dac_empty <= yes;
117-           dac_shfreg <= dac_shfreg(dac_width-2 DOWNTO 0) & '0';
118-         END IF;
119-       END IF;
120-     END IF;
121-   END PROCESS;
122-
123-   -- output the serial data to the SDIN pin of the codec DAC
124-   sdin <= dac_shfreg(dac_width-1) WHEN chan_on=yes ELSE '0';
125-
126-   -- handle writing of DAC data from codec interface
127-   dac_wr_nxt <= yes WHEN (dac_empty=yes AND chan_sel=yes AND wr=yes) OR
128-                           (dac_empty=yes AND dac_wr=yes)
129-             ELSE no;
130-   write_dac:
131-   PROCESS(clk,reset,dac_wr_nxt)
132-   BEGIN
133-     IF(clk'EVENT AND clk='1') THEN
134-       IF(reset=yes) THEN
135-         dac_wr <= no;
136-       ELSE
137-         dac_wr <= dac_wr_nxt;
138-       END IF;
139-     END IF;
140-   END PROCESS;
141-   -- DAC is ready for writing if reg is empty and not written yet
142-   dac_in_rdy_int <= yes WHEN dac_empty=yes AND dac_wr=no ELSE no;
143-   dac_in_rdy <= dac_in_rdy_int;
144-
145-   -- detect and signal underflow of data to the codec DAC channels
146-   detect_dac_underrun:
147-   PROCESS(clk,reset,bit_cntr,chan_on,dac_in_rdy_int)
148-   BEGIN
149-     IF(clk'EVENT AND clk='1') THEN
150-       IF(reset=yes) THEN
151-         dac_underrun <= no;
152-       ELSIF(bit_cntr=1 AND chan_on=yes AND dac_in_rdy_int=yes) THEN
153-         dac_underrun <= yes;
154-       END IF;
155-     END IF;
156-   END PROCESS;
157- END channel_arch;
```

The VHDL code for the top-level module that combines the clock generator module with two channel modules is detailed in **Listing 26**. The inputs and outputs of the top-level module as defined in the entity declaration are as follows:

clk: This is the main clock input which is typically the 12 MHz clock from the XS Board.

reset: This input synchronously resets the two channel modules and the clock generator.

lrsel: This input selects either the right or left channel for parallel read or write operations.

rd: A high level on this input outputs the value stored in the selected shift register connected to the ADC.

wr: A high level on this input writes a new value into the selected shift register connected to the DAC.

ladc_out, radc_out: The bits stored in the left and right ADC shift registers are read out in parallel through these outputs..

ldac_in, rdac_in: The DAC shift registers are loaded in parallel with bits passed through these inputs.

ladc_out_rdy, rdac_out_rdy: These outputs go high after all the bits have been shifted from the codec into the left or right ADC shift register, respectively.

adc_overrun: This output goes high if new serial data is shifted into either the left or right ADC shift register before the old contents have been read out through the parallel outputs.

ldac_in_rdy, rdac_in_rdy: These outputs go high after all the bits in the left or right DAC shift register have been shifted over to the codec, respectively.

dac_underrun: This output goes high if either the left or right DAC shift register starts shifting data over to the codec before it has been written through the parallel inputs.

mclk: This output is the master clock for the codec chip.

sclk: This output is the clock for synchronizing serial data transfers between the FPLD and the codec.

lrck: This output controls the activation of the left and right channel circuitry in the codec.

sdin: The serial data stream for the codec DAC is shifted out through this output.

sdout: The serial data stream from the codec ADC is shifted in through this input.

Within the main body of the top-level module architecture section, the following modules are instantiated:

u0: One clock generator module is instantiated. It receives the 12 MHz clock as an input and generates the master clock, left/right clock, and serial shift clock for the codec. It also outputs the position of the current bit in the serial stream and the current cycle within each bit period.

u_left: The module which handles the left channel of the codec is instantiated. This module is activated during one half of the left/right clock period. It is selected for reading or writing by the left/right selection input.

u_right: The module which handles the right channel of the codec is instantiated. This module is activated during the other half of the left/right clock period. It is selected for reading and writing by the opposite polarity of the left/right selection input.

Lines 129—130: The overrun and underrun error indicators for the total codec interface are formed by the logical-OR of the associated error outputs of the left and right channel modules. Thus an error is reported if either channel reports an error.

Line 134: The serial data stream that is transmitted to the codec chip is selected from the output data stream of the currently-active channel module.

Listing 26: VHDL code for the top-level codec interface module.

```
001- LIBRARY IEEE,codec;
002- USE IEEE.STD_LOGIC_1164.ALL;
003- USE IEEE.std_logic_unsigned.ALL;
004- USE codec.codec.ALL;
005-
006- ENTITY codec_intfc IS
007-   GENERIC
008-   (
009-     dac_width: POSITIVE := 20;
010-     adc_width: POSITIVE := 20;
011-     channel_duration: POSITIVE := 128  -- must be 128, 192, or 256
012-   );
013-   PORT
014-   (
015-     -- interface I/O signals
016-     clk: IN STD_LOGIC;    -- clock input
017-     reset: IN STD_LOGIC;  -- synchronous active-high reset
018-     lrsel: IN STD_LOGIC;  -- select the left/right chan for rd/wr
019-     rd: IN STD_LOGIC;     -- read from the codec ADC
020-     wr: IN STD_LOGIC;     -- write to the codec DAC
021-     ladc_out: OUT STD_LOGIC_VECTOR(adc_width-1 DOWNTO 0); -- left ADC
022-     radc_out: OUT STD_LOGIC_VECTOR(adc_width-1 DOWNTO 0); -- right ADC
023-     ldac_in: IN STD_LOGIC_VECTOR(dac_width-1 DOWNTO 0);   -- left DAC
024-     rdac_in: IN STD_LOGIC_VECTOR(dac_width-1 DOWNTO 0);   -- right DAC
025-     ladc_out_rdy: OUT STD_LOGIC; -- left ADC output is ready to be read
026-     radc_out_rdy: OUT STD_LOGIC; -- right ADC output is ready to be read
027-     adc_overrun: OUT STD_LOGIC;  -- ADC data overwritten before read
028-     ldac_in_rdy: OUT STD_LOGIC;  -- left DAC input ready to be written
029-     rdac_in_rdy: OUT STD_LOGIC;  -- right DAC input ready to be written
030-     dac_underrun: OUT STD_LOGIC; -- DAC not written to in time
031-     -- codec chip I/O signals
032-     mclk: OUT STD_LOGIC;  -- master clock output to codec
```

```
033-     sclk: OUT STD_LOGIC;  -- serial data clock to codec
034-     lrck: OUT STD_LOGIC;  -- left/right codec channel select
035-     sdin: OUT STD_LOGIC;  -- serial output to codec DAC
036-     sdout: IN STD_LOGIC    -- serial input from codec ADC
037-   );
038- END codec_intfc;
039-
040- ARCHITECTURE codec_intfc_arch OF codec_intfc IS
041- SIGNAL lrck_int: STD_LOGIC;  -- internal left/right codec channel select
042- SIGNAL bit_cntr: STD_LOGIC_VECTOR(5 DOWNTO 0);
043- SIGNAL subcycle_cntr: STD_LOGIC_VECTOR(1 DOWNTO 0);
044- SIGNAL lsdin: STD_LOGIC;
045- SIGNAL rsdin: STD_LOGIC;
046- SIGNAL ladc_overrun: STD_LOGIC;
047- SIGNAL radc_overrun: STD_LOGIC;
048- SIGNAL ldac_underrun: STD_LOGIC;
049- SIGNAL rdac_underrun: STD_LOGIC;
050- SIGNAL lchan_sel: STD_LOGIC;
051- SIGNAL rchan_sel: STD_LOGIC;
052- SIGNAL lchan_on: STD_LOGIC;
053- SIGNAL rchan_on: STD_LOGIC;
054- BEGIN
055-
056-   u0: clkgen
057-       GENERIC MAP
058-       (
059-         channel_duration=>channel_duration
060-       )
061-       PORT MAP
062-       (
063-         clk=>clk,
064-         reset=>reset,
065-         mclk=>mclk,
066-         sclk=>sclk,
067-         lrck=>lrck_int,
068-         bit_cntr=>bit_cntr,
069-         subcycle_cntr=>subcycle_cntr
070-       );
071-   lrck <= lrck_int;
072-
073-   lchan_sel <= yes WHEN lrsel=left ELSE no;
074-   lchan_on <= yes WHEN lrck_int=left ELSE no;
075-   u_left: channel
076-       GENERIC MAP
077-       (
078-         dac_width=>dac_width,
079-         adc_width=>adc_width
080-       )
081-       PORT MAP
082-       (
083-         clk=>clk,
084-         reset=>reset,
085-         chan_on=>lchan_on,
086-         bit_cntr=>bit_cntr,
087-         subcycle_cntr=>subcycle_cntr,
088-         chan_sel=>lchan_sel,
089-         rd=>rd,
090-         wr=>wr,
091-         adc_out=>ladc_out,
092-         dac_in=>ldac_in,
```

```
093-         adc_out_rdy=>ladc_out_rdy,
094-         adc_overrun=>ladc_overrun,
095-         dac_in_rdy=>ldac_in_rdy,
096-         dac_underrun=>ldac_underrun,
097-         sdin=>lsdin,
098-         sdout=>sdout
099-       );
100-
101-   rchan_sel <= yes WHEN lrsel=right ELSE no;
102-   rchan_on <= yes WHEN lrck_int=right ELSE no;
103-   u_right: channel
104-       GENERIC MAP
105-       (
106-         dac_width=>dac_width,
107-         adc_width=>adc_width
108-       )
109-       PORT MAP
110-       (
111-         clk=>clk,
112-         reset=>reset,
113-         chan_on=>rchan_on,
114-         bit_cntr=>bit_cntr,
115-         subcycle_cntr=>subcycle_cntr,
116-         chan_sel=>rchan_sel,
117-         rd=>rd,
118-         wr=>wr,
119-         adc_out=>radc_out,
120-         dac_in=>rdac_in,
121-         adc_out_rdy=>radc_out_rdy,
122-         adc_overrun=>radc_overrun,
123-         dac_in_rdy=>rdac_in_rdy,
124-         dac_underrun=>rdac_underrun,
125-         sdin=>rsdin,
126-         sdout=>sdout
127-       );
128-
129- dac_underrun <= yes WHEN ldac_underrun=yes OR rdac_underrun=yes ELSE no;
130-  adc_overrun <= yes WHEN ladc_overrun=yes OR radc_overrun=yes ELSE no;
131-
132-   -- generates the serial data output to the SDIN pin of the
133-   -- codec DAC depending on which channel is active
134-   sdin <= lsdin WHEN lrck_int=left ELSE rsdin;
135-
136- END codec_intfc_arch;
```

The interfaces to these three modules are placed into the package shown in

Listing 27. (The I/O declarations in the COMPONENT constructs have been removed for the sake of brevity.) The declarations for the constants used in these modules are also included in the package.

Listing 27 : VHDL code for the codec package.

```
001- LIBRARY IEEE;
002- USE IEEE.STD_LOGIC_1164.ALL;
003- USE IEEE.std_logic_unsigned.ALL;
004-
005- PACKAGE codec IS
006-   CONSTANT yes: STD_LOGIC := '1';
007-   CONSTANT no: STD_LOGIC := '0';
008-   CONSTANT ready: STD_LOGIC := '1';
009-   CONSTANT overrun: STD_LOGIC := '1';
010-   CONSTANT underrun: STD_LOGIC := '1';
011-   CONSTANT left: STD_LOGIC := '0';
012-   CONSTANT right: STD_LOGIC := '1';
013-
014-   COMPONENT clkgen
015-     GENERIC
016-     (
017-       ...
018-     );
019-     PORT
020-     (
021-       ...
022-     );
023-   END COMPONENT;
024-
025-   COMPONENT channel
026-     GENERIC
027-     (
028-       ...
029-     );
030-     PORT
031-     (
032-       ...
033-     );
034-   END COMPONENT;
035-
036-   COMPONENT codec_intfc
037-     GENERIC
038-     (
039-       ...
040-     );
041-     PORT
042-     (
043-       ...
044-     );
045-   END COMPONENT;
046- END PACKAGE;
```

Once the codec interface module is completed and packaged, we can use it in an application. The simplest use is to have the FPLD accept the left and right stereo inputs from the codec ADCs and loop these back to the codec DACs so they can output the stereo signals.

The VHDL code for the loopback application is detailed in **Listing 29**. The inputs and outputs of the loopback design are as follows:

clk: This is the 12 MHz clock from the XS Board.

reset: A high level on this input synchronously resets the codec interface module. The reset input is driven from the parallel port of the PC.

mclk: This output is the master clock for the codec chip.

lrck: This output controls the activation of the left and right channel circuitry in the codec and the codec interface.

sclk: This output is the clock for synchronizing serial data transfers between the FPLD and the codec.

sdout: The serial data stream from the codec ADCs are shifted in through this input.

sdin: The serial data stream for the codec DACs are shifted out through this output.

The following modules and processes are placed within the main body of the loopback application:

u0: This is the instantiation of the codec interface module. Note that the ADC output buses of this module are connected back to the DAC input buses on lines 43—46.

loop: This process controls the reading of each ADC and the writing of the value back to the associated DAC. For example, if the output of the left channel ADC is ready to be read and the left channel DAC is ready to be written, then the left channel is selected and the read and write control lines are asserted. This reads the data from the ADC shift register and writes it into the DAC shift register during a single clock cycle. Then the ADC and DAC registers will no longer be ready for reading or writing so the read and write signals will be deasserted.

Listing 29: VHDL code for a design that uses the codec interface module to do loopback.

```
001- LIBRARY IEEE,codec;
002- USE IEEE.STD_LOGIC_1164.ALL;
003- USE codec.codec.ALL;
004-
005- ENTITY loopback IS
006-   PORT
007-   (
008-     clk: IN STD_LOGIC;   -- 12 MHz clock
009-     rst: IN STD_LOGIC;   -- active-high reset
010-     mclk: OUT STD_LOGIC; -- master clock to codec
011-     lrck: OUT STD_LOGIC; -- left/right clock to codec
```

```
012-     sclk: OUT STD_LOGIC; -- serial data shift clock to codec
013-     sdout: IN STD_LOGIC; -- serial data from codec ADCs
014-     sdin: OUT STD_LOGIC; -- serial data to codec DACs
015-     s: OUT STD_LOGIC_VECTOR(1 DOWNTO 0) –- LED segments
016-   );
017- END loopback;
018-
019- ARCHITECTURE loopback_arch OF loopback IS
020- SIGNAL lrsel,rd,wr: STD_LOGIC;
021- SIGNAL left_channel,right_channel: STD_LOGIC_VECTOR(7 DOWNTO 0);
022- SIGNAL ldac_in_rdy,rdac_in_rdy: STD_LOGIC;
023- SIGNAL ladc_out_rdy,radc_out_rdy: STD_LOGIC;
024- BEGIN
025-   u0: codec_intfc
026-         GENERIC MAP
027-         (
028-           adc_width=>20,
029-           dac_width=>20
030-         )
031-         PORT MAP
032-         (
033-           clk=>clk,
034-           reset=>rst,
035-           mclk=>mclk,
036-           sclk=>sclk,
037-           lrck=>lrck,
038-           sdout=>sdout,
039-           sdin=>sdin,
040-           lrsel=>lrsel,
041-           rd=>rd,
042-           wr=>wr,
043-           ladc_out=>left_channel,   -- loop the left channel ADC
044-           ldac_in=>left_channel,    -- to the left channel DAC
045-           radc_out=>right_channel,  -- loop the right channel ADC
046-           rdac_in=>right_channel,   -- to the right channel DAC
047-           ladc_out_rdy=>ladc_out_rdy,
048-           radc_out_rdy=>radc_out_rdy,
049-           ldac_in_rdy=>ldac_in_rdy,
050-           rdac_in_rdy=>rdac_in_rdy,
051-           dac_underrun=>s(0),  -- connect underrun and overrun
052-           adc_overrun=>s(1)    -- error indicators to LEDs
053-         );
054-
055-   loop: PROCESS(ldac_in_rdy,ladc_out_rdy,rdac_in_rdy,radc_out_rdy)
056-   BEGIN
057-     IF(ladc_out_rdy=yes AND ldac_in_rdy=yes) THEN
058-       lrsel<=left;  -- loopback the left channel
059-       rd<=yes;      -- assert the read and
060-       wr<=yes;      -- write control signals
061-     ELSIF(radc_out_rdy=yes AND rdac_in_rdy=yes) THEN
062-       lrsel<=right; -- loopback the right channel
063-       rd<=yes;      -- assert the read and
064-       wr<=yes;      -- write control signals
065-     ELSE
066-       lrsel<=left;  -- default channel selection
067-       rd<=no;       -- but don’t read or
068-       wr<=no;       -- write the registers
069-     END IF;
070-   END PROCESS;
071- END loopback_arch;
```

Listing 30: XS40 UCF file for the stereo signal loopback application.

```
001- net clk     loc=p13;
002- net rst     loc=p44;
003- net sdout   loc=p6;
004- net mclk    loc=p9;
005- net lrck    loc=p66;
006- net sdin    loc=p70;
007- net sclk    loc=p77;
008- net s<0>    loc=p25;
009- net s<1>    loc=p26;
```

Listing 31: XS95 UCF file for the stereo signal loopback application.

```
001- net clk     loc = p9
002- net rst     loc = p46
003- net sdout   loc = p5
004- net mclk    loc = p11
005- net lrck    loc = p66
006- net sdin    loc = p71
007- net sclk    loc = p72
008- net s<0>    loc = p21
009- net s<1>    loc = p23
```

The steps for compiling and testing the design using an XS40 combined with an XStend Board are as follows:

- Synthesize the VHDL code in the LOOP40\LOOPBACK.VHD for an XC4005XL FPGA.
- Compile the synthesized netlist using the LOOP40.UCF constraint file (**Listing 30**).
- Mount an XS40 Board in the XStend Board and attach the downloading cable from the XS40 to the PC parallel port. Apply 9VDC though jack J9 of the XS40. Remove the shunts from jumpers J4, J7, and J8 to disable the LEDs. Place a shunt on jumper J17 so the codec serial output data stream can reach the FPLD. Set all the DIP switches to the OPEN position.
- Connect a stereo audio source (such as a CD player) to jack J9. Then plug a set of stereo mini-headphones into jack J10.
- Download the LOOP40.BIT file into the XS40/XStend combination with the command: `XSLOAD LOOP40.BIT`.
- Release the reset on the loopback circuit with the command `XSPORT 0`.
- Start the CD player and listen to the result with the headphones.

The steps for compiling and testing the design using an XS95 combined with an XStend Board are as follows:

- Synthesize the VHDL code in the LOOP95\LOOP.VHD for an XC95108 CPLD.

- Compile the synthesized netlist using the LOOP95.UCF constraint file (**Listing 31**).

- Generate an SVF file for the design.

- Mount an XS95 Board in the XStend Board and attach the downloading cable from the XS95 to the PC parallel port. Apply 9VDC though jack J9 of the XS95. Remove the shunts from jumpers J4, J7, and J8 to disable the LEDs. Place a shunt on jumper J17 so the codec serial output data stream can reach the FPLD. Set all the DIP switches to the OPEN position.

- Connect a stereo audio source (such as a CD player) to jack J9. Then plug a set of stereo mini-headphones into jack J10.

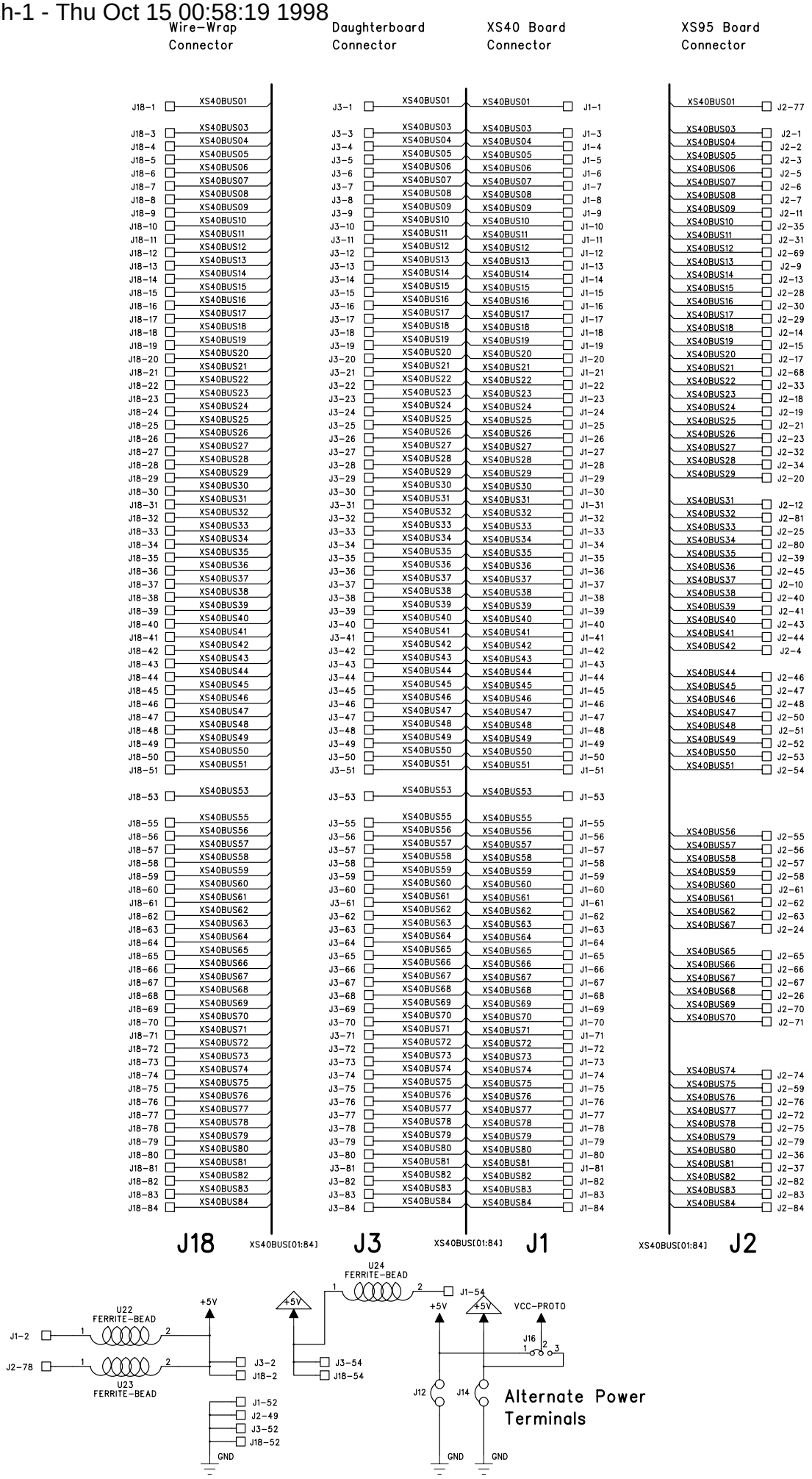
- Download the LOOP95.BIT file into the XS95/XStend combination with the command: `XSLOAD LOOP95.BIT`.

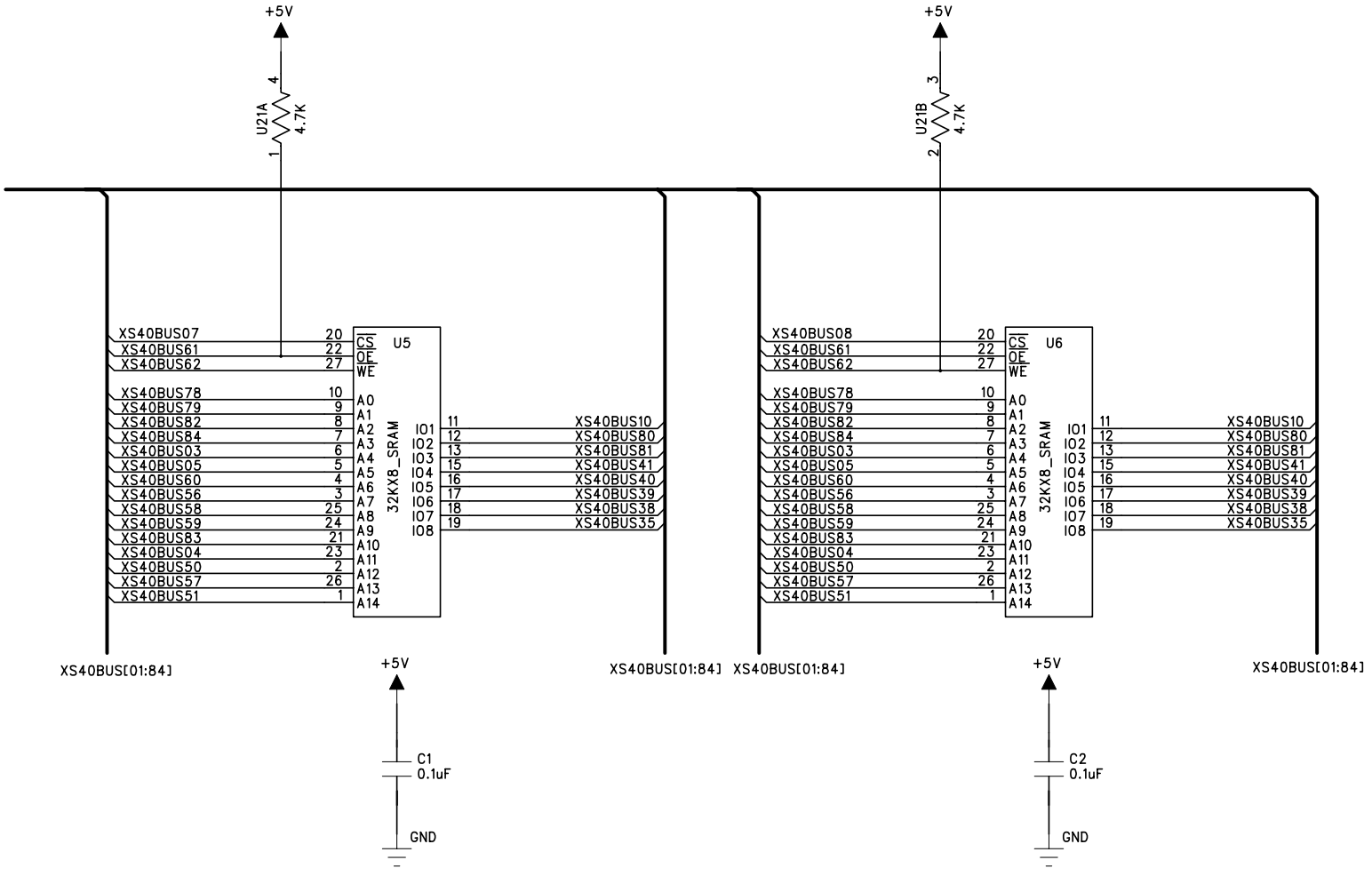
- Release the reset on the loopback circuit with the command `XSPORT 0`.

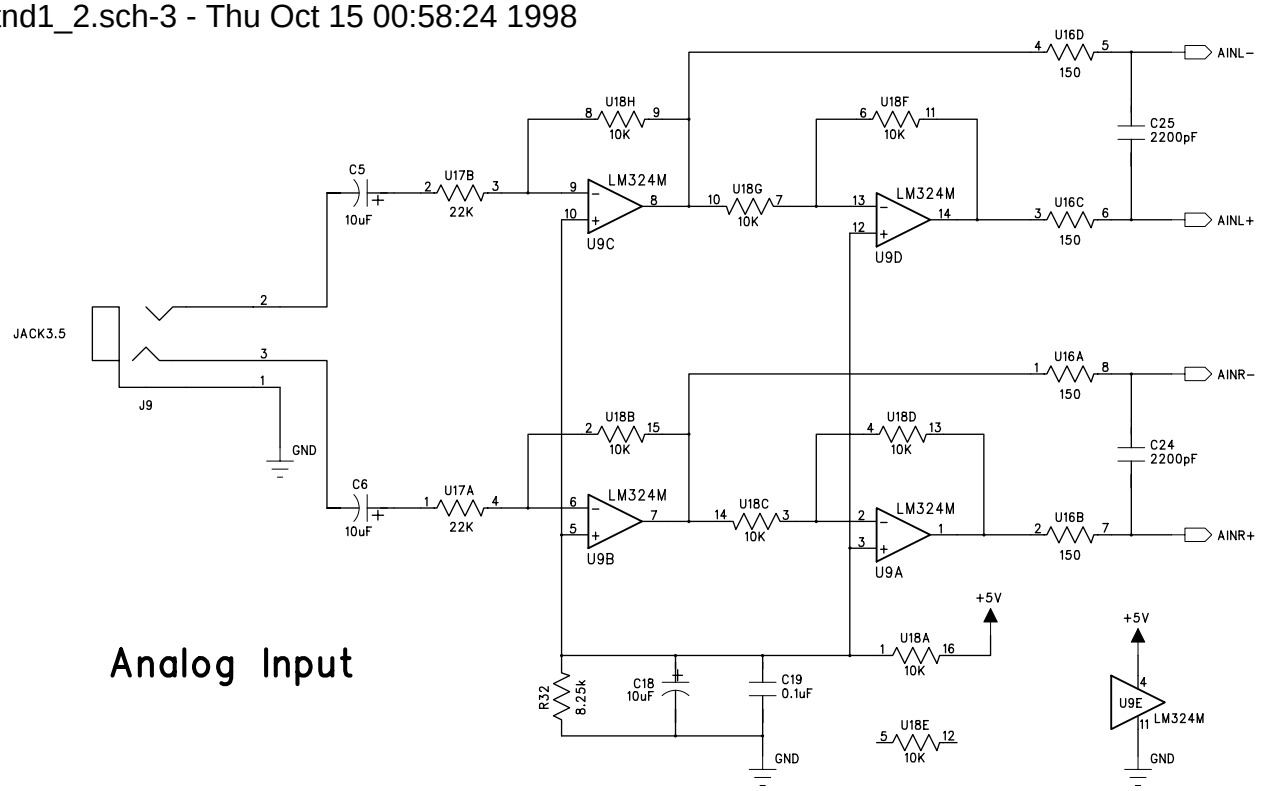
- Start the CD player and listen to the result with the headphones.

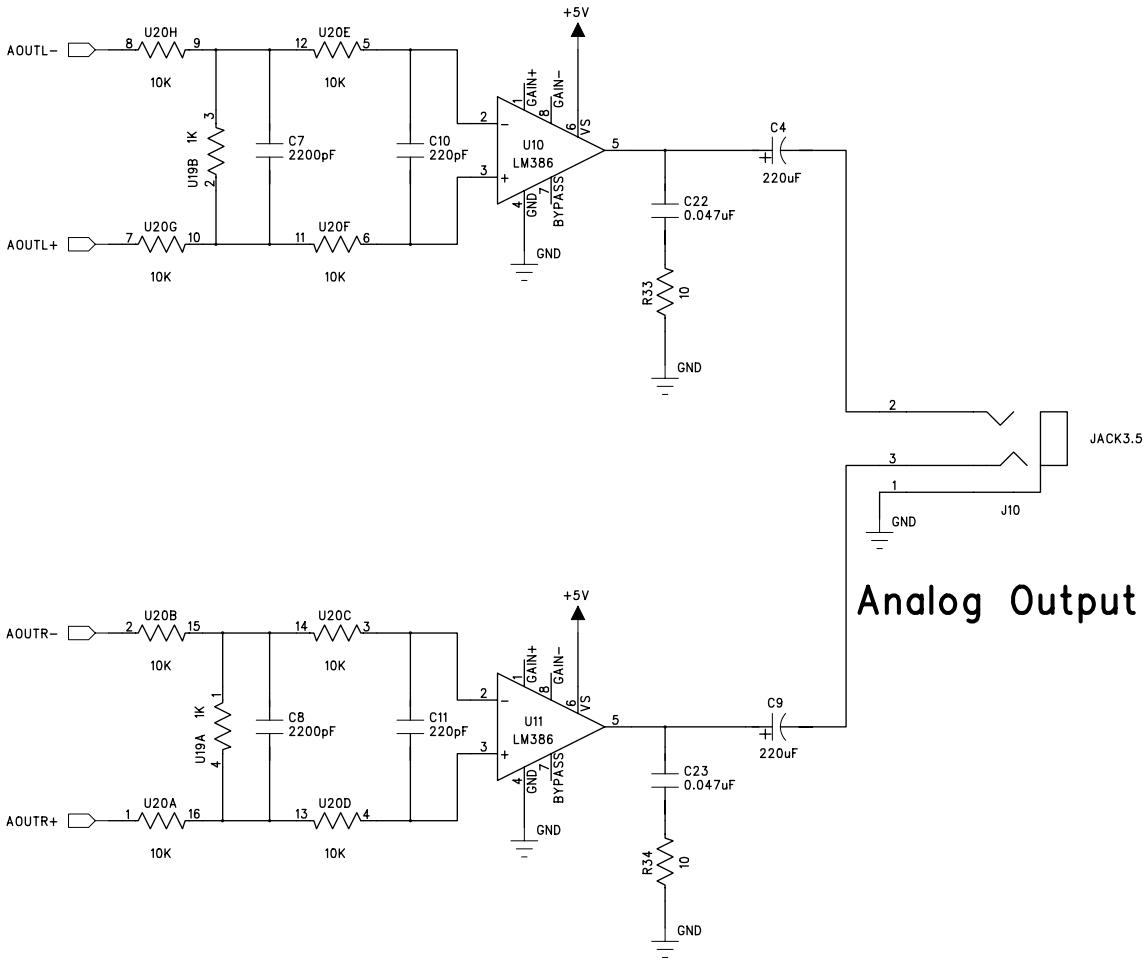
5 XStend V1.2 Schematics

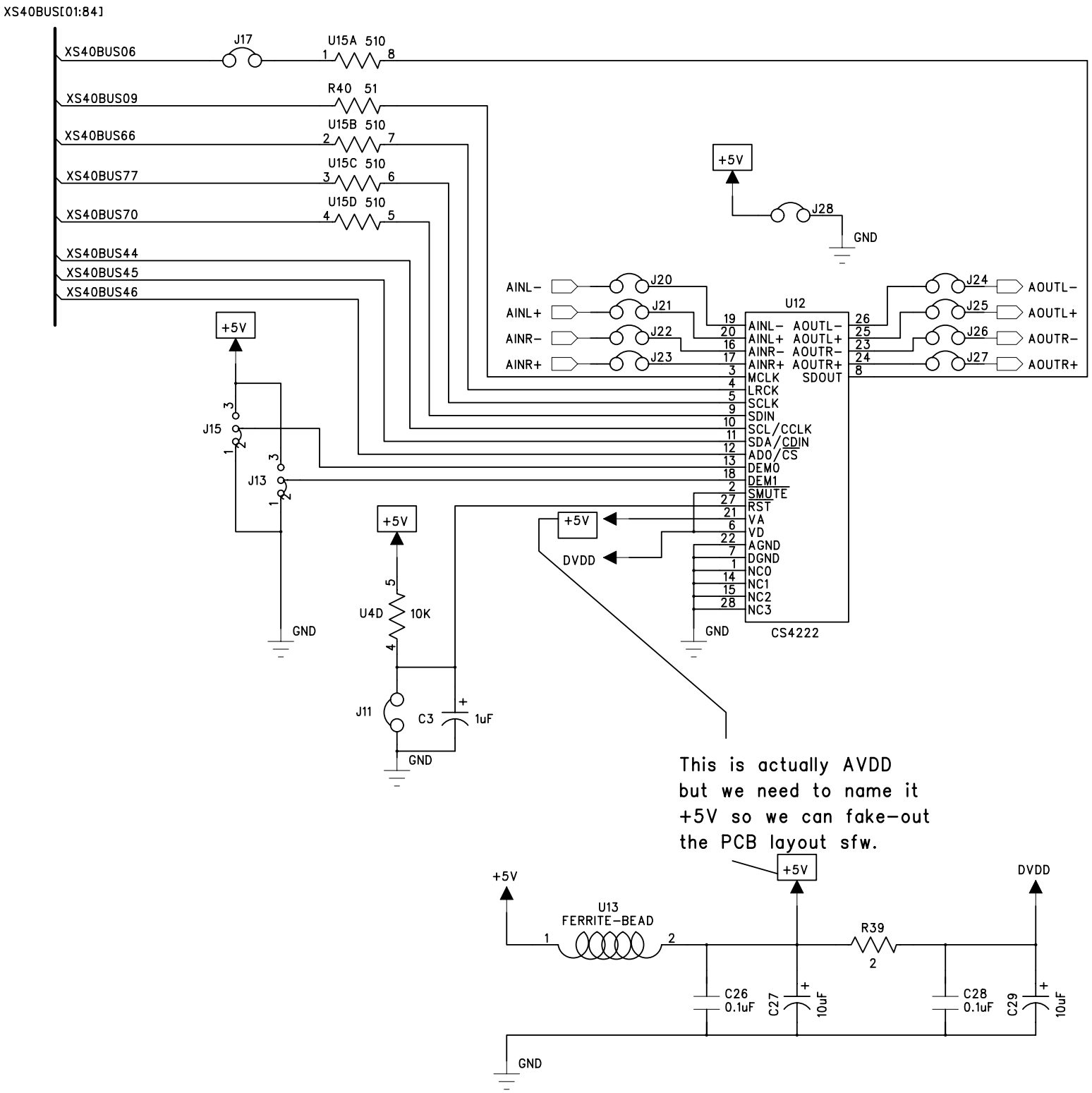
The detailed schematics for the XStend Board are on the following pages.

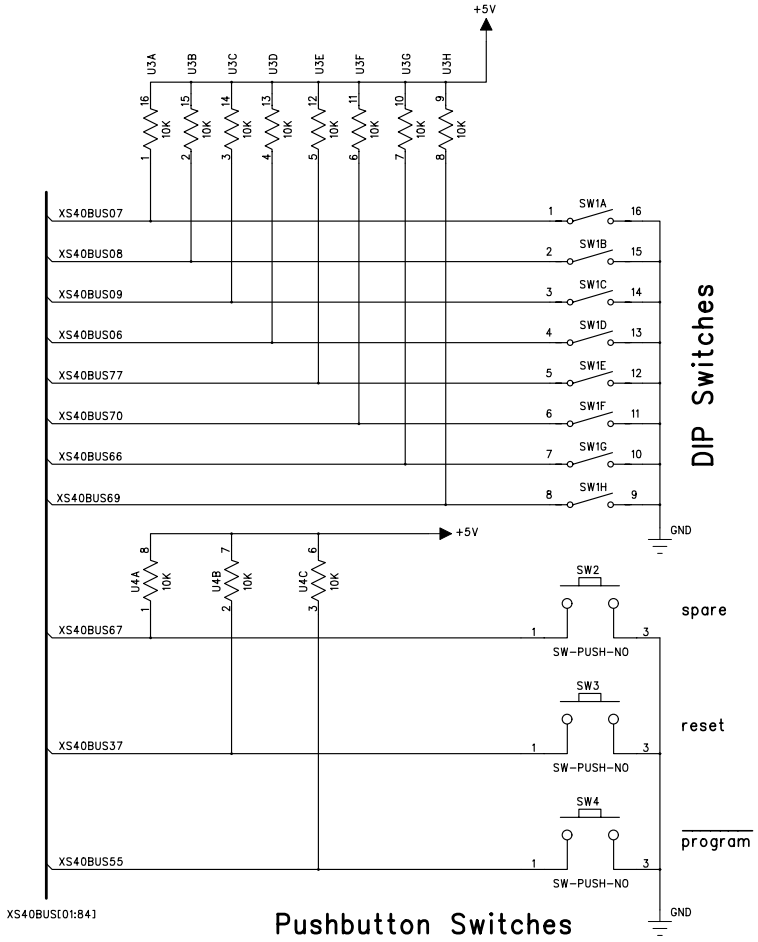

Wire-Wrap Connector
Daughterboard Connector
XS40 Board Connector
XS95 Board Connector
J18
XS40BUS[01:84]
J3
XS40BUS[01:84]
J1
XS40BUS[01:84]
J2
U22 FERRITE-BEAD
U23 FERRITE-BEAD
U24 FERRITE-BEAD
+5V
VCC-PROTO
GND
J16
J12
J14
Alternate Power Terminals

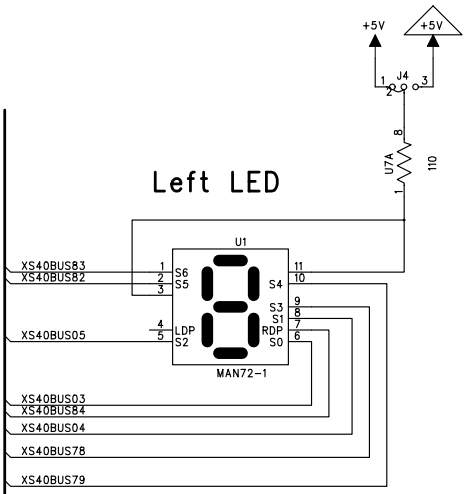

+5V
U21A
4.7K
+5V
U21B
4.7K
U5
U6
32KX8_SRAM
XS40BUS07
XS40BUS61
XS40BUS62
XS40BUS78
XS40BUS79
XS40BUS82
XS40BUS84
XS40BUS03
XS40BUS05
XS40BUS60
XS40BUS56
XS40BUS58
XS40BUS59
XS40BUS83
XS40BUS04
XS40BUS50
XS40BUS57
XS40BUS51
XS40BUS08
XS40BUS10
XS40BUS80
XS40BUS81
XS40BUS41
XS40BUS40
XS40BUS39
XS40BUS38
XS40BUS35
XS40BUS[01:84]
C1
0.1uF
C2
0.1uF
GND

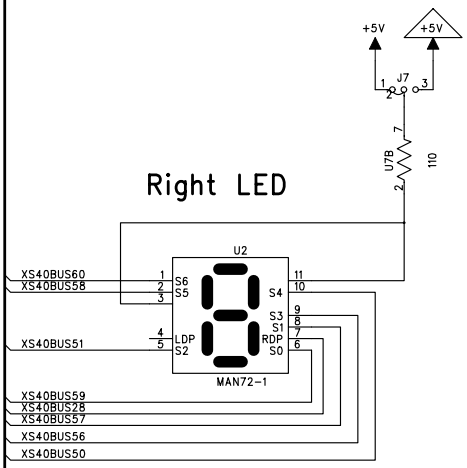

U16D
150
AINL-
C25
2200pF
U18H
10K
U18F
10K
C5
10uF
U17B
22K
LM324M
U9C
U18G
10K
LM324M
U9D
U16C
150
AINL+
JACK3.5
J9
GND
U16A
150
AINR-
U18B
10K
U18D
10K
C24
2200pF
C6
10uF
U17A
22K
LM324M
U9B
U18C
10K
LM324M
U9A
U16B
150
AINR+
+5V
U18A
10K
R32
8.25k
C18
10uF
C19
0.1uF
U18E
10K
+5V
U9E
LM324M
GND
Analog Input


AOUTL-
U20H
10K
U20E
10K
U19B
1K
C7
2200pF
C10
220pF
AOUTL+
U20G
10K
U20F
10K
+5V
U10
LM386
GAIN+
GAIN-
VS
GND
BYPASS
C4
220uF
C22
0.047uF
R33
10
GND
JACK3.5
J10
GND
AOUTR-
U20B
10K
U20C
10K
U19A
1K
C8
2200pF
C11
220pF
AOUTR+
U20A
10K
U20D
10K
+5V
U11
LM386
GAIN+
GAIN-
VS
GND
BYPASS
C9
220uF
C23
0.047uF
R34
10
GND
Analog Output

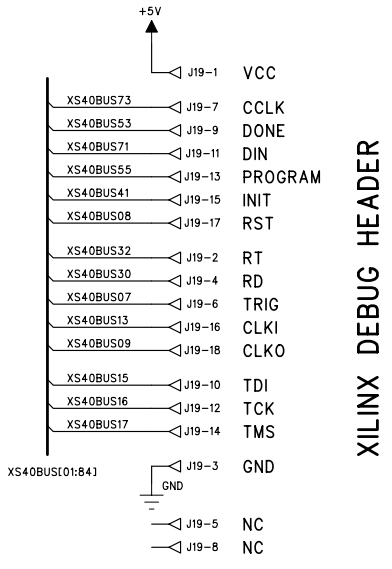

XS40BUS[01:84]
XS40BUS06
XS40BUS09
XS40BUS66
XS40BUS77
XS40BUS70
XS40BUS44
XS40BUS45
XS40BUS46
J17
U15A 510
R40 51
U15B 510
U15C 510
U15D 510
+5V
J28
GND
AINL−
AINL+
AINR−
AINR+
J20
J21
J22
J23
U12
MCLK
LRCK
SCLK
SDIN
SCL/CCLK
SDA/CDIN
AD0/CS
DEM0
DEM1
SMUTE
RST
VA
VD
AGND
DGND
NC0
NC1
NC2
NC3
AOUTL−
AOUTL+
AOUTR−
AOUTR+
SDOUT
J24
J25
J26
J27
CS4222
J15
J13
U4D 10K
J11
C3 1uF
DVDD
This is actually AVDD
but we need to name it
+5V so we can fake−out
the PCB layout sfw.
U13
FERRITE−BEAD
R39
C26 0.1uF
C27 10uF
C28 0.1uF
C29 10uF

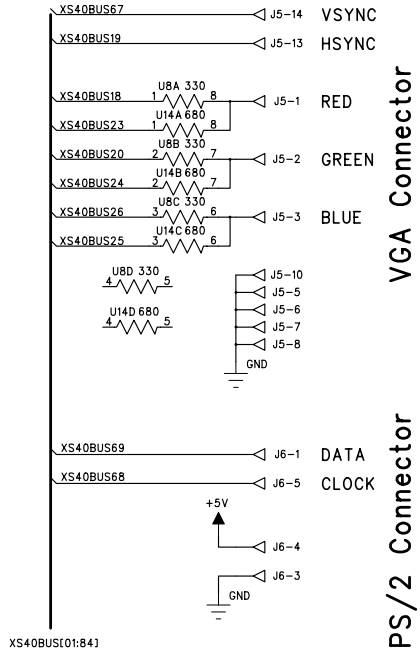

+5V
U3A 10K
U3B 10K
U3C 10K
U3D 10K
U3E 10K
U3F 10K
U3G 10K
U3H 10K
XS40BUS07
XS40BUS08
XS40BUS09
XS40BUS06
XS40BUS77
XS40BUS70
XS40BUS66
XS40BUS69
SW1A
SW1B
SW1C
SW1D
SW1E
SW1F
SW1G
SW1H
DIP Switches
GND
U4A 10K
U4B 10K
U4C 10K
XS40BUS67
SW2
SW-PUSH-NO
spare
XS40BUS37
SW3
reset
XS40BUS55
SW4
program
XS40BUS[01:84]
Pushbutton Switches

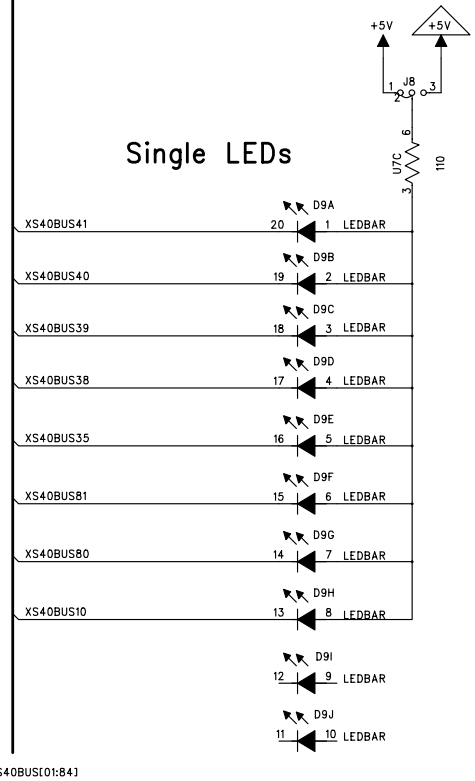

+5V
J4
U7A 110
Left LED
U1
XS40BUS83
XS40BUS82
XS40BUS05
XS40BUS03
XS40BUS84
XS40BUS04
XS40BUS78
XS40BUS79
S6
S5
S4
S3
S1
RDP
S0
LDP
S2
MAN72-1


+5V
J7
U7B 110
Right LED
U2
XS40BUS60
XS40BUS58
XS40BUS51
XS40BUS59
XS40BUS28
XS40BUS57
XS40BUS56
XS40BUS50
S6
S5
S4
S3
S1
RDP
S0
LDP
S2
MAN72-1


U7D 110


+5V
J19-1 VCC
XS40BUS73 J19-7 CCLK
XS40BUS53 J19-9 DONE
XS40BUS71 J19-11 DIN
XS40BUS55 J19-13 PROGRAM
XS40BUS41 J19-15 INIT
XS40BUS08 J19-17 RST
XS40BUS32 J19-2 RT
XS40BUS30 J19-4 RD
XS40BUS07 J19-6 TRIG
XS40BUS13 J19-16 CLKI
XS40BUS09 J19-18 CLKO
XS40BUS15 J19-10 TDI
XS40BUS16 J19-12 TCK
XS40BUS17 J19-14 TMS
XS40BUS[01:84]
J19-3 GND
GND
J19-5 NC
J19-8 NC
XILINX DEBUG HEADER


XS40BUS67
J5-14 VSYNC
XS40BUS19
J5-13 HSYNC
XS40BUS18
U8A 330
J5-1 RED
XS40BUS23
U14A 680
XS40BUS20
U8B 330
J5-2 GREEN
XS40BUS24
U14B 680
XS40BUS26
U8C 330
J5-3 BLUE
XS40BUS25
U14C 680
U8D 330
U14D 680
J5-10
J5-5
J5-6
J5-7
J5-8
GND
VGA Connector
XS40BUS69
J6-1 DATA
XS40BUS68
J6-5 CLOCK
+5V
J6-4
J6-3
GND
PS/2 Connector
XS40BUS[01:84]


+5V
J8
U7C 110
Single LEDs
XS40BUS41
XS40BUS40
XS40BUS39
XS40BUS38
XS40BUS35
XS40BUS81
XS40BUS80
XS40BUS10
D9A LEDBAR
D9B LEDBAR
D9C LEDBAR
D9D LEDBAR
D9E LEDBAR
D9F LEDBAR
D9G LEDBAR
D9H LEDBAR
D9I LEDBAR
D9J LEDBAR
XS40BUS[01:84]