Embedded System Design

Project Report

Internet Radio

Yingjian Gu yg2154@columbia.edu
Qiutao Yu qy2104@columbia.edu
Chun-Chuen Li cl2222@columbia.edu
Imran Quyyum iq2101@columbia.edu

1. Project Overview

We have implement an internet radio server using the XESS XSB-300E board. In this project we have been able to access as much as possible peripherals on the FPGA board, and learned more from the embeded system design course. We do not focus on a real functional complex internet server software in this project.

We implemented the live broadcast radio with configurable advertisement insertion. We captured the audio through the audio codec chip, and playback the broadcast content for monitoring through the codec chip and speaker. We used the UDP protocol to streaming out the audio through the ethernet controller. We could insert advitisement to the broadcast under user control.. In order to implement the configurable features, we used the VGA with menu display, and user input control from UART channel, so that the user input can be used to change the mode. We provided the SRAM, and Flash memory interface for the data storage. The SRAM is used to save the codec serial data during the capturing, also it is used as the software code running. The Flash is used to save the advertisement sound data. The SDRAM was developed, but we could not make it work with intensive audio data access.

The following is the system diagram,

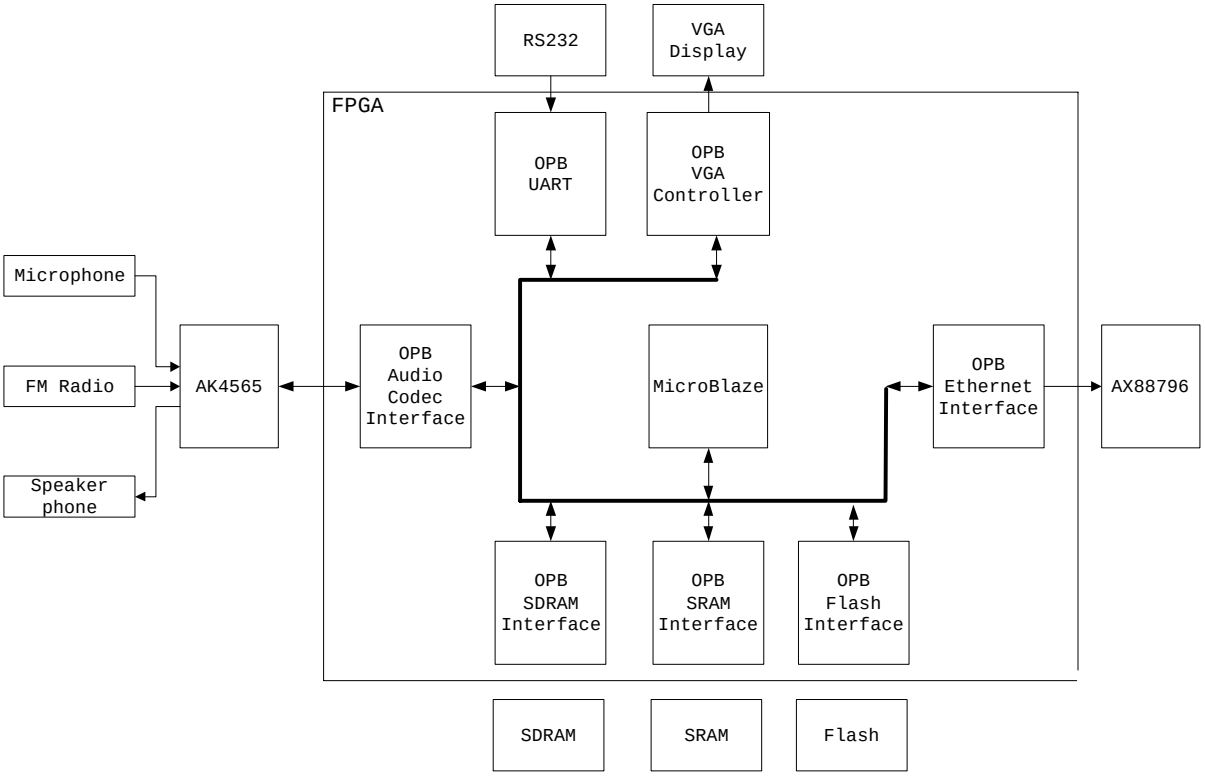
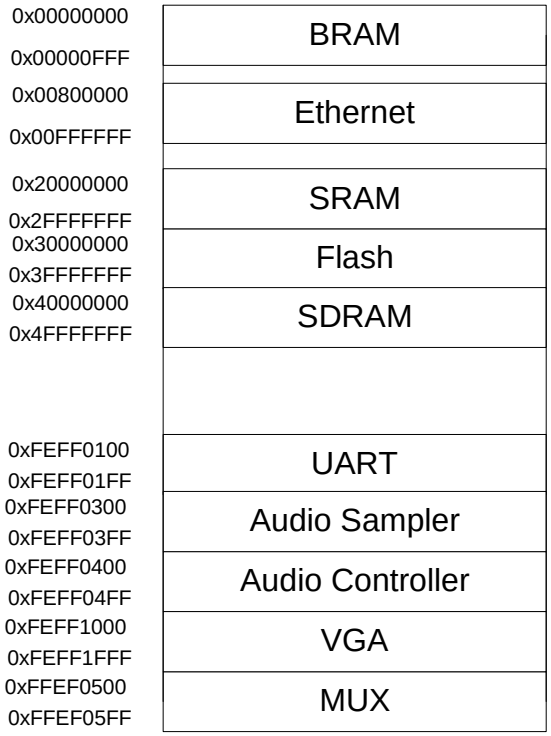


Figure 1 System Diagram

2. Project Design

2.1 System Memory Map

There are multiple pheriherals, such as the BRAM, SRAM, Flash, SDRAM, VGA, Ethernet, Audio Sampler, Audio Controller, UART, Interrupt controller and MUX, being used in the project design. The memory address map is defined in the mhs file. The following memory map is created in the OPB memory space,

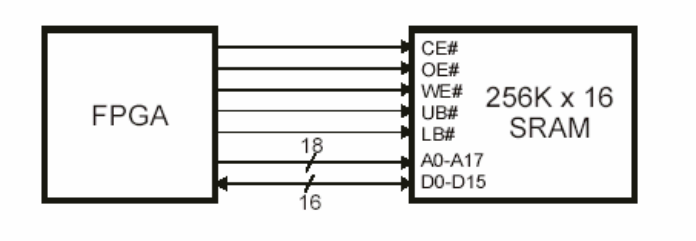


System Memory Mapping

2.2 SRAM Controller Design

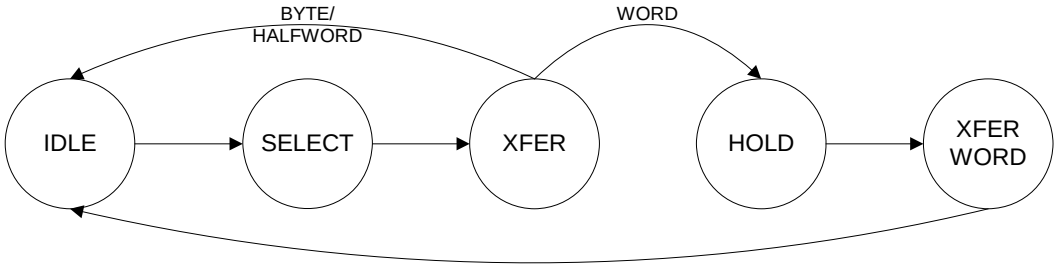
The SRAM memory is used to run the application code by the processor. It is also used as the audio data buffer. The BRAM size is very limited in terms of the program code size, and we were not able to fit the application to the BRAM. So the SRAM interface is required to run application.

The SRAM interface will handle the transfer from 32 bit OPB data bus to 16 bit SRAM data bus. All the processor access modes are supported, such as the byte mode, halfword mode and word mode. All the 512KB memory space will be accessible from the processor.

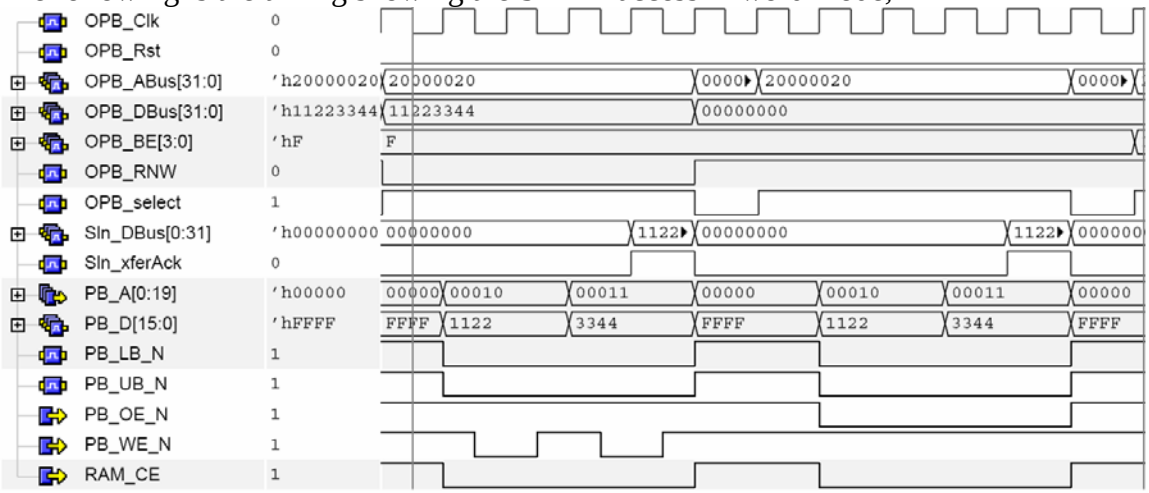


SRAM/FPGA connection

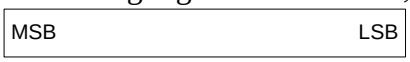
The SRAM controller will do byte mode and halfword mode in one SRAM access cycle, and word access in two SRAM access cycle. The following is the state machine of the SRAM controller,



The following is the timing showing the SRAM access in word mode,

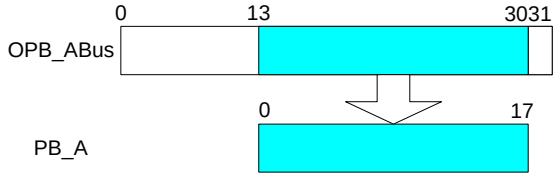


We are using big Endian notation, which has the following format:

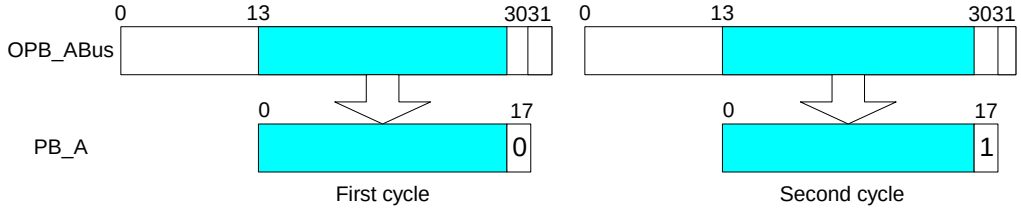


For example, the address bit 0 will the highest bit A17 on the SRAM chip.

The OPB bus has 32-bit address bus, while the SRAM has 18-bit address bus. The following shows the mapping between the OPB and SRAM in byte and halfword mode,

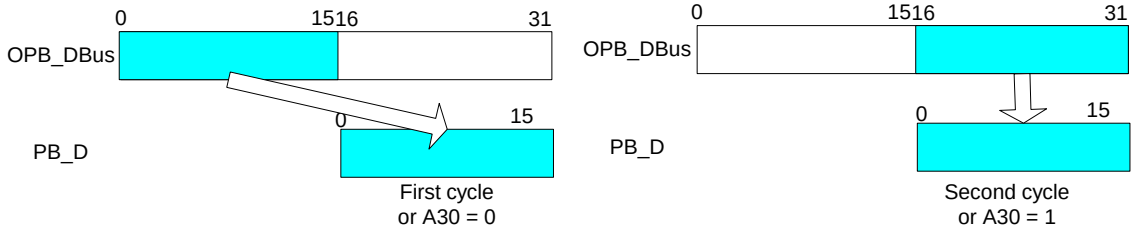


In the word mode, we will drive the lowest bit address of the SRAM as 0 in the first cycle, and drive it as 1 in the second cycle.

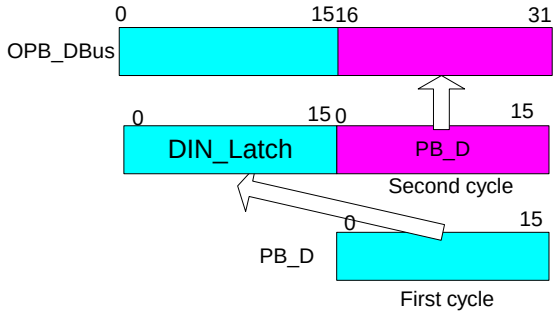


The OPB bus has 32-bit data bus, while the SRAM has 16-bit data bus. The data bus mappings are different depending on the access modes.

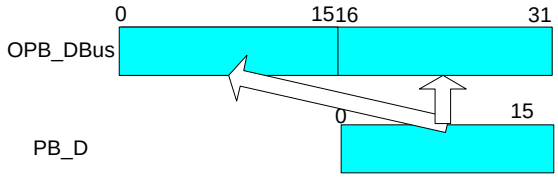
In the write mode, we have similar mapping for word, halfword and byte access. In the word mode, we select the OPB_DBus higher 16 bit(0 to 15) to the SRAM data bus in the first cycle. Then we select the OPB_DBus lower 16 bit(16 to 31) to the SRAM data bus in the second cycle.
In the halfword or byte mode, we select the OPB_DBus higher 16 bit(0 to 15) to the SRAM data bus when the A30 is 0. We select the OPB_DBus lower 16 bit(16 to 31) to the SRAM data bus when the A30 is 1. This is not really necessary, since the MicroBlaze processor seems to provide write data mirroring.



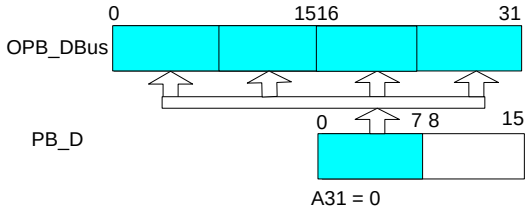
In the word mode read, we need to latch and save the first 16-bit read back data in a buffer DIN_LATCH. Then we drive this data together with the second 16-bit read back data to the OPB_DBus.

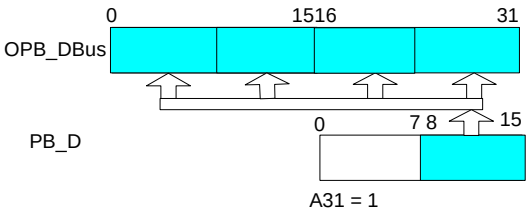


In the halfword mode read, we always drive the SRAM data output to both of the OPB_DBus higher 16 bit and lower 16 bit.

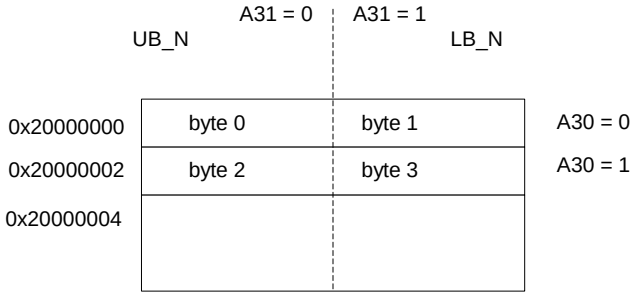


In the byte mode read, when the A31 is 0, we drive the OPB_DBus with SRAM higher byte (PB_D bit 0 to 7). When the A31 is 1, we drive the OPB_DBus with SRAM lower byte (PB_D bit 8 to 15).





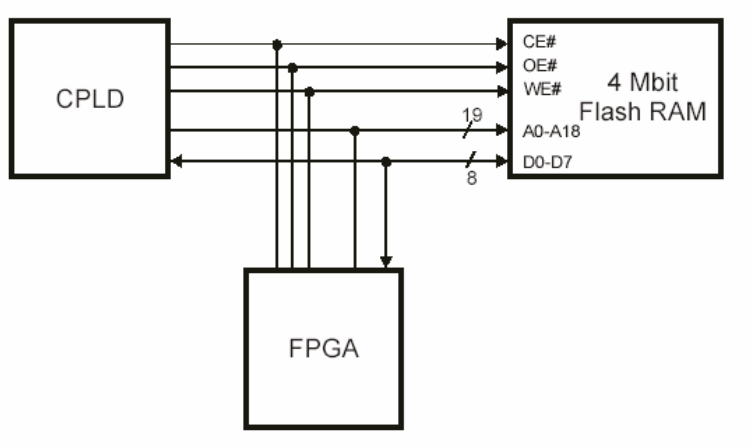
The following is the diagram shows the SRAM data and address in different access mode(byte, halfword and word),



The LB_N and UB_N signal is generated in the following way,
In the read operation, we drive them as active low for all the cases, since read will not destroy data any way.
In the word or halfword mode write, we drive them as active low, since both byte lane will be written.
In the byte mode, we drive them according to A31 address bit. When A31 is 1, we drive LB_N active low. When A31 is 0, we drive UB_N active low.

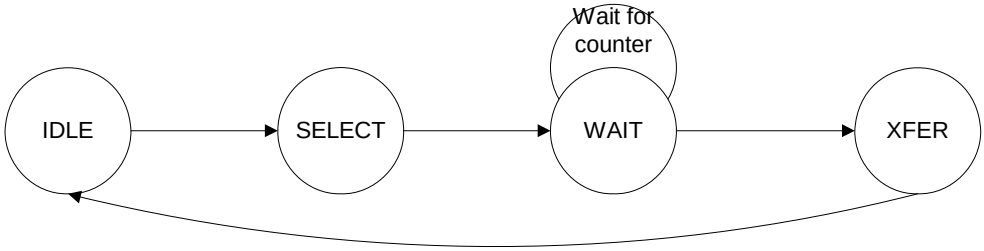
2.3 Flash Controller

The Flash memory is used to store the audio data. The Flash controller will provide byte mode access to the processor. All the 512KB memory space will be accessible from the processor.

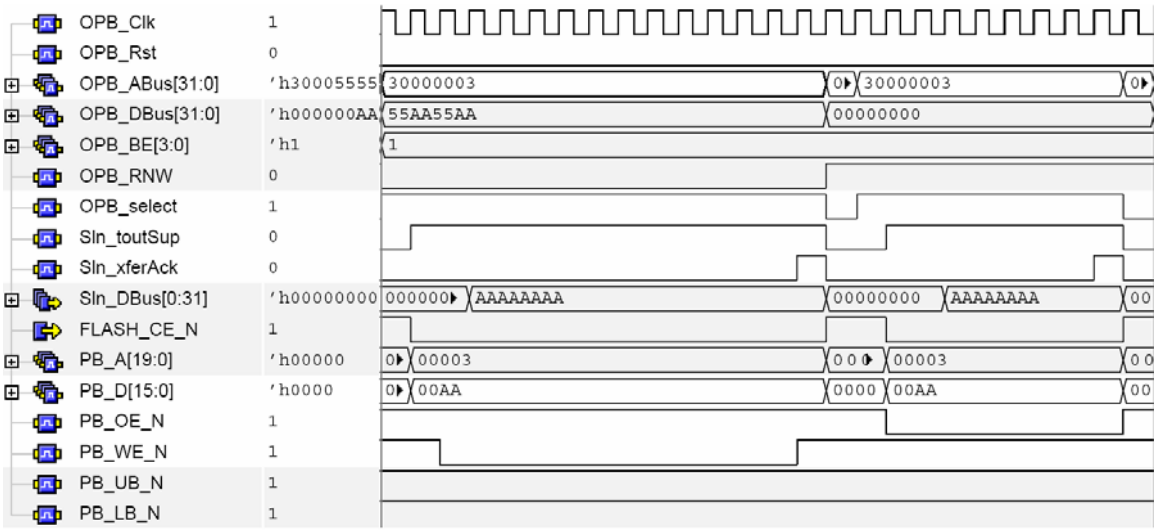


Flash/FPGA Connection

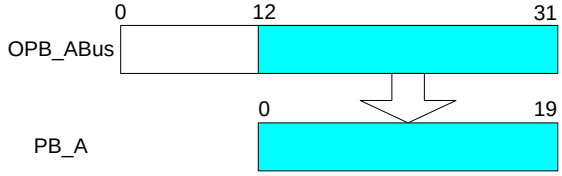
The Flash controller generates read and write timing to the Flash memory. The Flash access time is much slower than the SRAM. We had used a counter to measure the wait time during the Flash access. The following is the Flash access state machine,



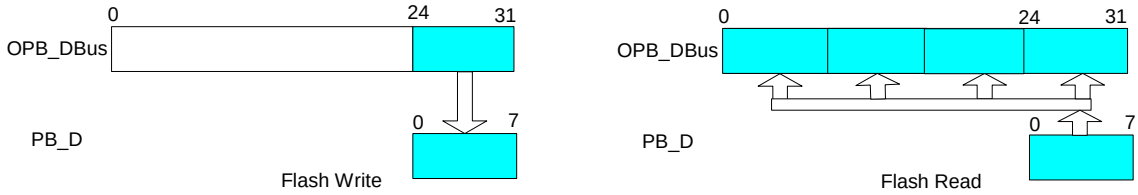
The following is the timing diagram showing the Flash access,



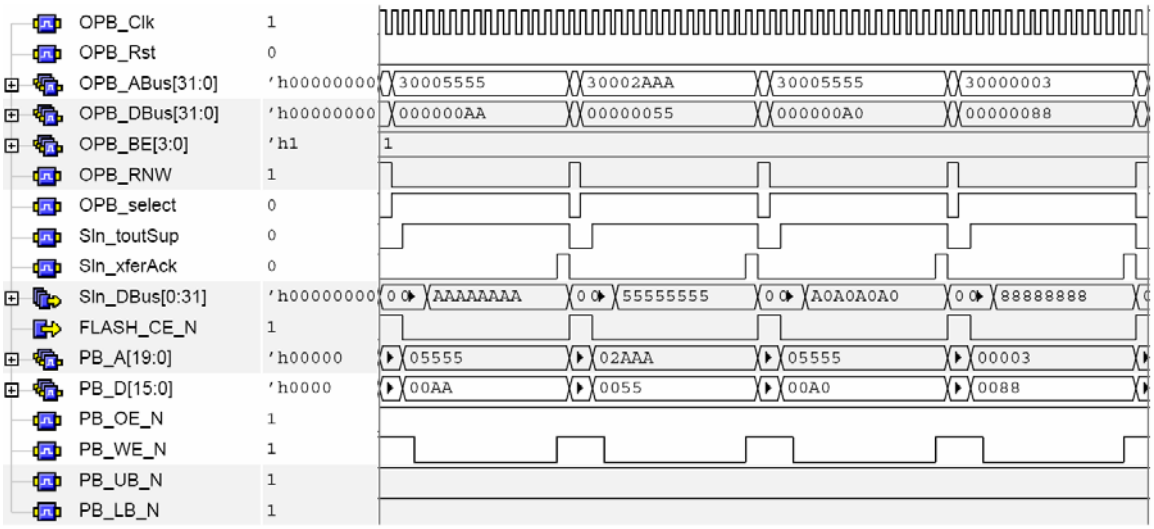
The Flash Address is mapped into byte address, and the following is the mapping:



The Flash data access is always in byte mode, and the following is the data mapping:



The Flash requires a special writing sequence for the erase and program operation. We had developed subroutine to support these operations. The following diagram shows the Flash programming cycles:



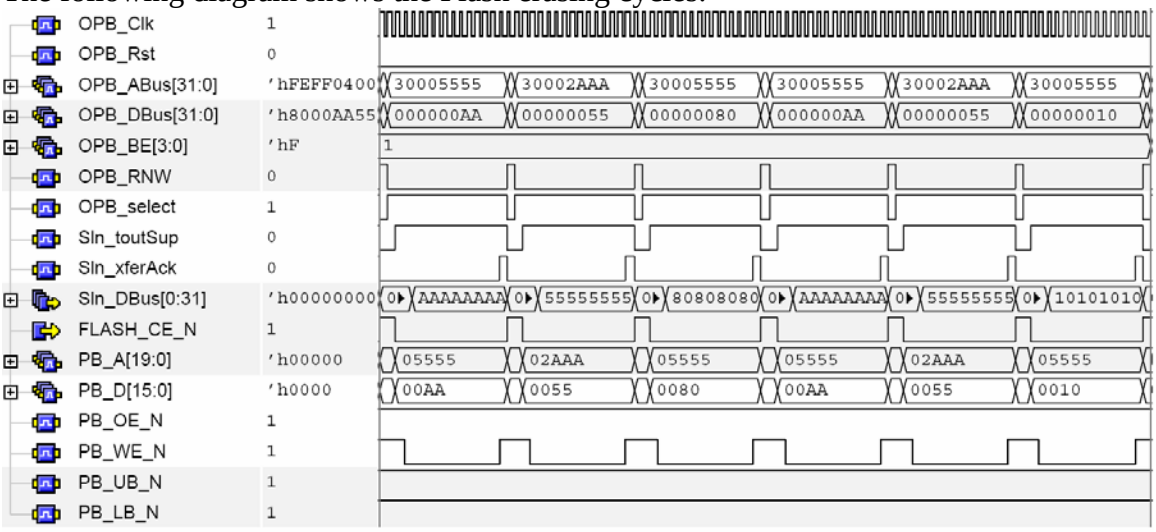
Flash Programming Timing

The Flash program subroutine takes the address and data as inputs, and then it does the following writing sequence,

```
FlashWrite(0x30005555, 0xAA);
FlashWrite(0x30002AAA, 0x55);
FlashWrite(0x30005555, 0xA0);
FlashWrite(address, data);
```

Then it will do the polling of the Flash status by calling the FlashPolling subroutine.

The following diagram shows the Flash erasing cycles:



Flash Erasing Timing

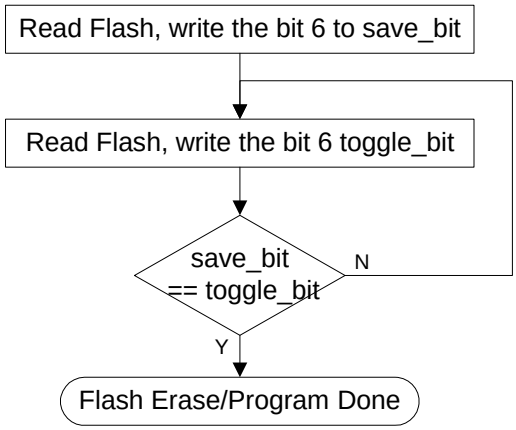
The Flash chip erase subroutine does the following writing sequence,

```
FlashWrite(0x30005555, 0xAA);
FlashWrite(0x30002AAA, 0x55);
FlashWrite(0x30005555, 0x80);
FlashWrite(0x30005555, 0xAA);
```

FlashWrite(0x30002AAA, 0x55);
FlashWrite(0x30005555, 0x10);

Then it will do the polling of the Flash status by calling the FlashPolling subroutine.

The Flash erase and program operation takes long time to finish. During the Flash internal non volatile operation, there are toggling provided on the Flash data bus. We developed polling routine, and check the Flash status. The following shows the polling routine:



Flash polling

2.4 SDRAM Controller

The on board SDRAM provides larger memory space. We had developed SDRAM controller to provide the software SDRAM access.
The SDRAM controller works fine in memory test routine, however it fails in the extensive audio data application.

The SDRAM interface will be designed such that the SDRAM is accessed in a similar way as the SRAM. The refresh operation of the SDRAM will be done by the SDRAM controller itself. The SDRAM controller provides simple read and write operation to the OPB bus.

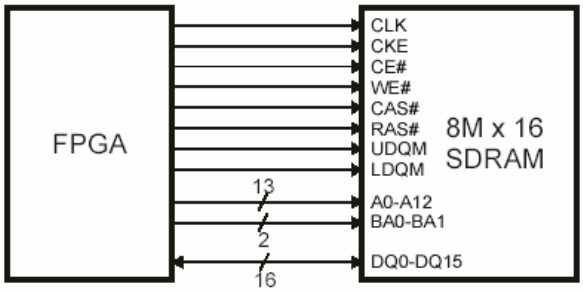
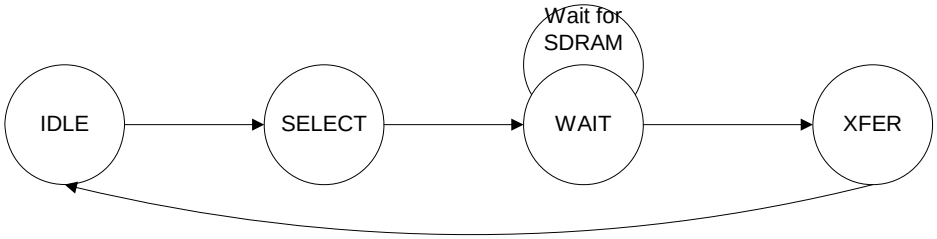
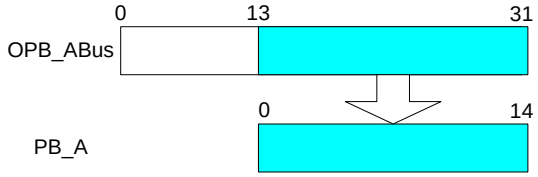


Figure 7 SDRAM/FPGA connection

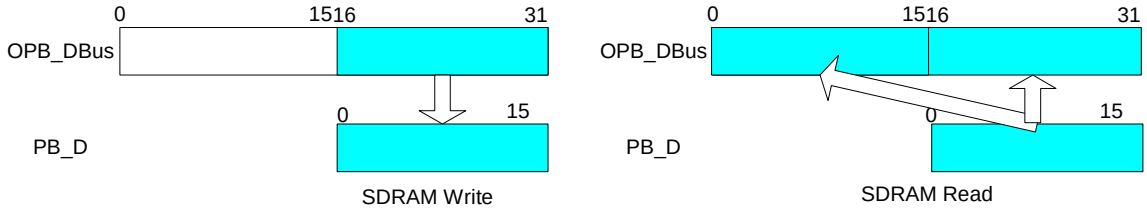
We had used the XESS SDRAM controller reference design, and the details are in the an-071205-xsbsdramcntl.pdf. We wrapped up the XESS SDRAM IP core, and implemented the OPB interface with our own logic.
The following is the OPB state machine of the SDRAM controller,



The SDRAM interface provides 16-bit mode access to the processor. The following is the address bus mapping:

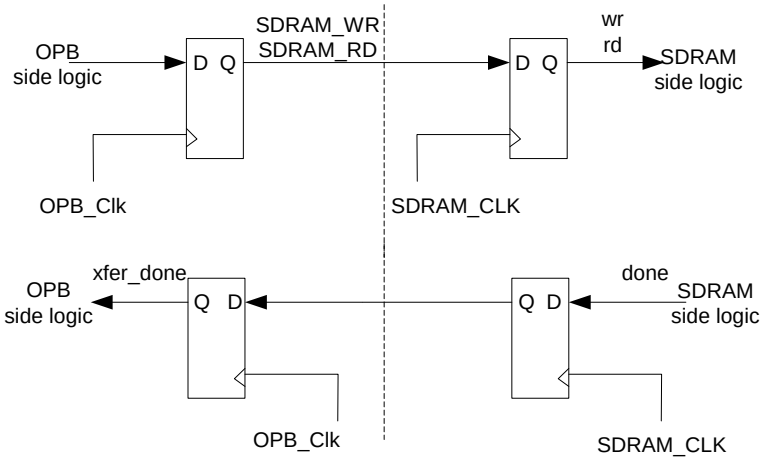


The following is the data bus mapping,

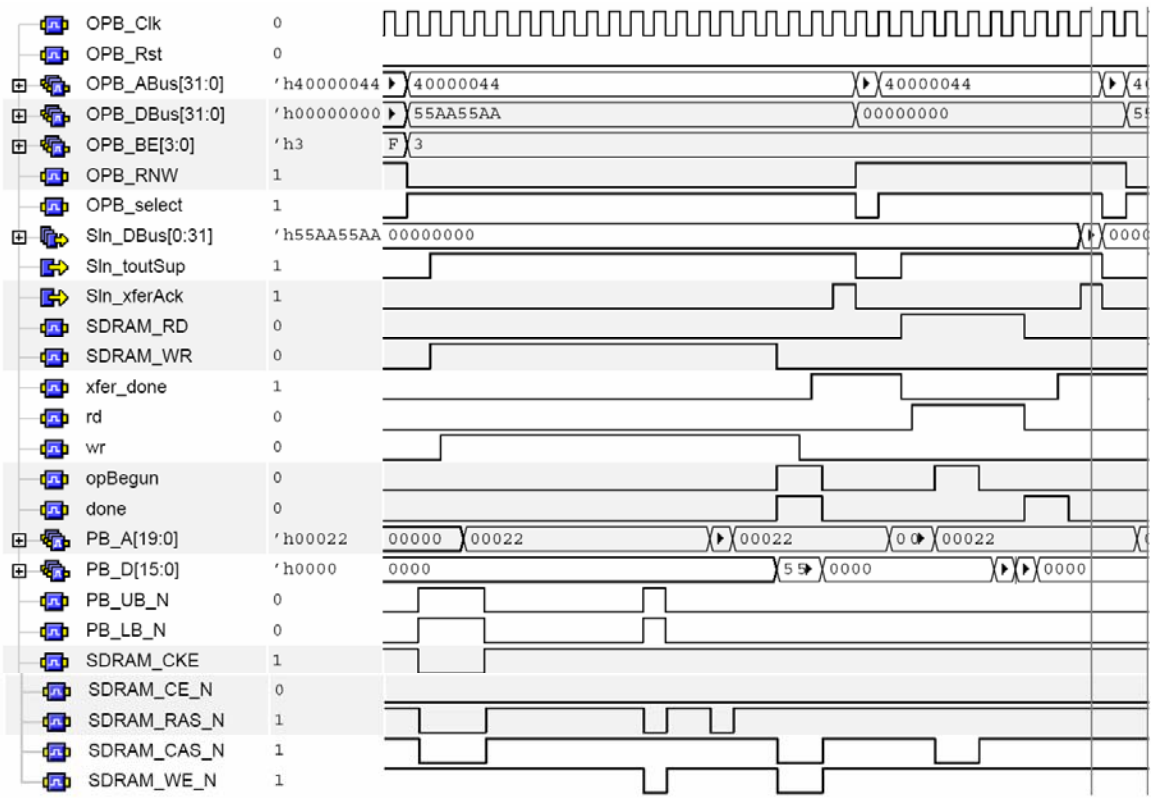


We provides halfword access for the SDRAM interface, so the C code will be looking like:
Unsigned short * SDRAM_BUFFER;
SDRAM_BUFFER[address] = value;

In the SDRAM controller, there are two global clock, OPB_Clk (50MHz) and SDRAM clock(66MHz). The SDRAM controller works across clock domain, and we have to provide synchronization between them. The OPB_Clk generated signal (OPB_Clk domain) will be latched by the SDRAM clock first, then it is used to drive the SDRAM IP(SDRAM clock domain). The SDRAM IP generated signal (SDRAM clock domain) will be latched by the OPB_Clk, then it is used to drive the OPB state machine.



The following is the SDRAM timing diagram from the XESS application note,

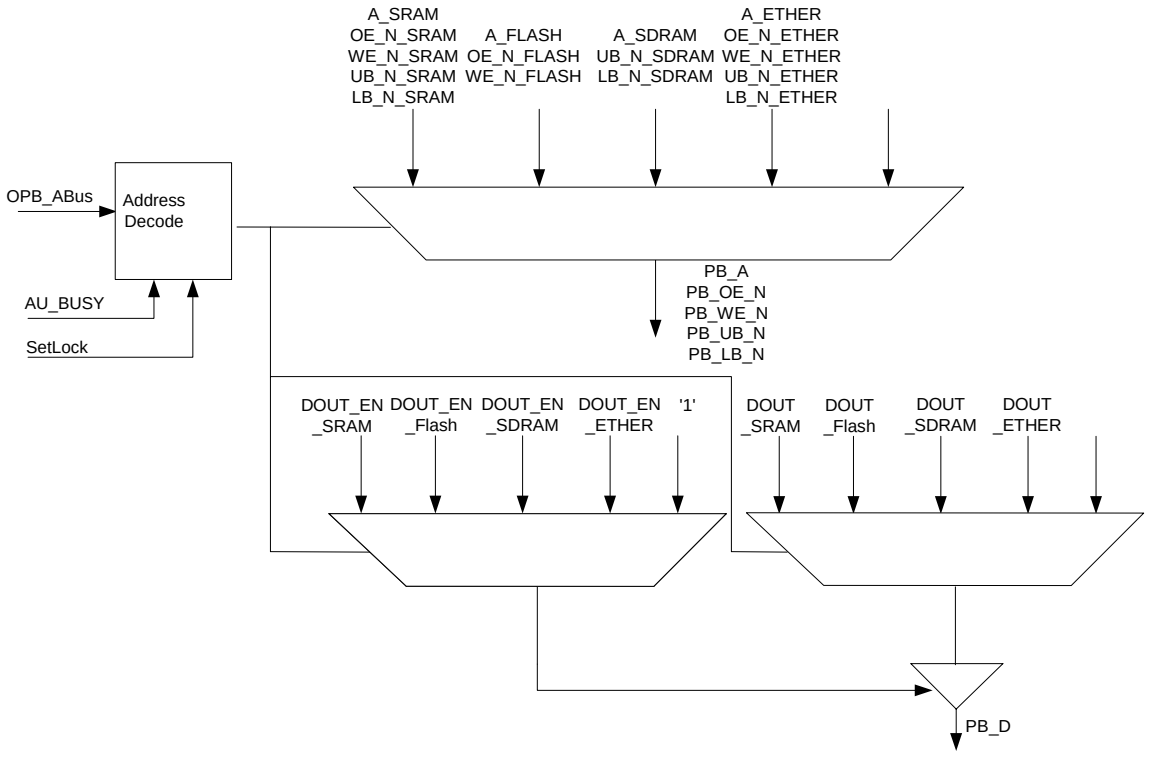


SDRAM Read/Write Timing

2.5 Peripheral Bus MUX

The FPGA board provides peripherals, such as the SRAM, Flash, SDRAM, Ethernet, and Audio, but all these peripherals share the same PB_A, PB_D, PB_LB_N, PB_UB_N, PB_OE_N, PB_WE_N bus signals, which are driven by the same sets of FPGA pins.

In order to access all these peripherals at the same time, we have developed peripheral bus MUX.
The MUX will consider the OPB bus address, and pick up the correct peripheral, then multiplex the address, data, and control signals to the FPGA pins.
The following shows the diagram of the MUX controller,



There are two special things in the MUX design, which is the SDRAM initialization and Audio Controller interface.

The SDRAM initialization needs dedicated PB_A bus access, so that the SDRAM refresh, precharge, and load mode register can be done. This process takes hundreds of micro second (us). We have to lock the PB bus to the SDRAM controller during this time. This locking is done by the hardware and software cooperation. The hardware provides a SDRAM control register inside the MUX module. The following is the bit map of this register at address 0xFFEF0500,

SDRAM CONTROL REGISTER

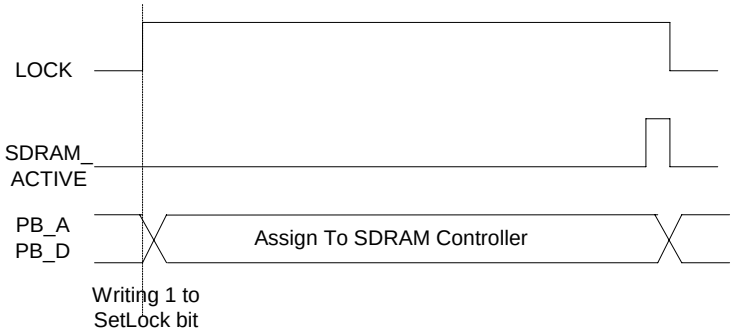
BIT 23	BIT 24	BIT 26	BIT 27	BIT 28	BIT 29	BIT 30	BIT 31
						SDRAM BUSY	SET LOCK

The InitializeSDRAM routine has been developed for this purpose. This routine will first set the LOCK as high,

```
preg = (volatile unsigned long * )SDRAM_CTL_REG_ADDR;
* preg = 1;
```

Then the MUX controller will assign the PB_A bus to the SDRAM controller, and the BUSY bit will be driven high. When the SDRAM initialization is done, the BUSY bit will be cleared. So the code checks for the BUSY bit for the status information,

```
while (status != 0)
{
   status = * preg;
}
```



The Audio controller also requires dedicated access during the serial download. The AUDIO_BUSY signal is created for this purpose. When the Audio controller is downloading, the MUX will lock the PB_B bus to the Audio controller.

2.6 OPB TimeOut

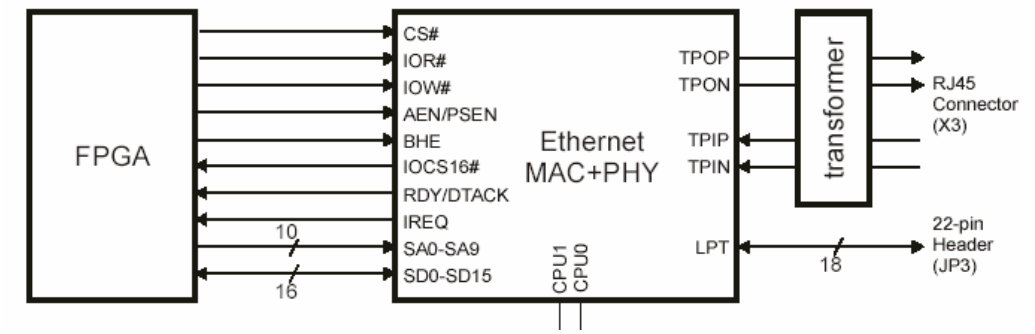
The OPB operation defines 16 clock cycle as the time out limit. Any operation that exceeds 16 clock cycles will get time out error.
In our application, the Flash requires long writing time, but it is still within 16 clock cycles.

However, there is no guarantee that the SDRAM read or write can be done within 16 clocks. Since the SDRAM might be in the refresh mode or precharge mode when it is accessed, then it takes longer time to finish the access.

We used the Sln_toutSup signal to suppress the time out condition on the OPB, so that we never get time out. We drive the Sln_toutSup signal high during the access, and drive it low when the access is done.

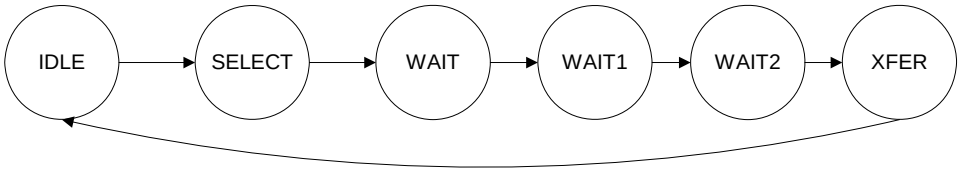
2.7 Ethernet Controller

The RTP protocol will be used for the audio streaming to the internet. The processor will take the payload data from the buffer, and construct the IP/UDP/RTP packet. The Ethernet Controller ASIX AX88796L will be initialized and used to transimit the audio data to the internet.



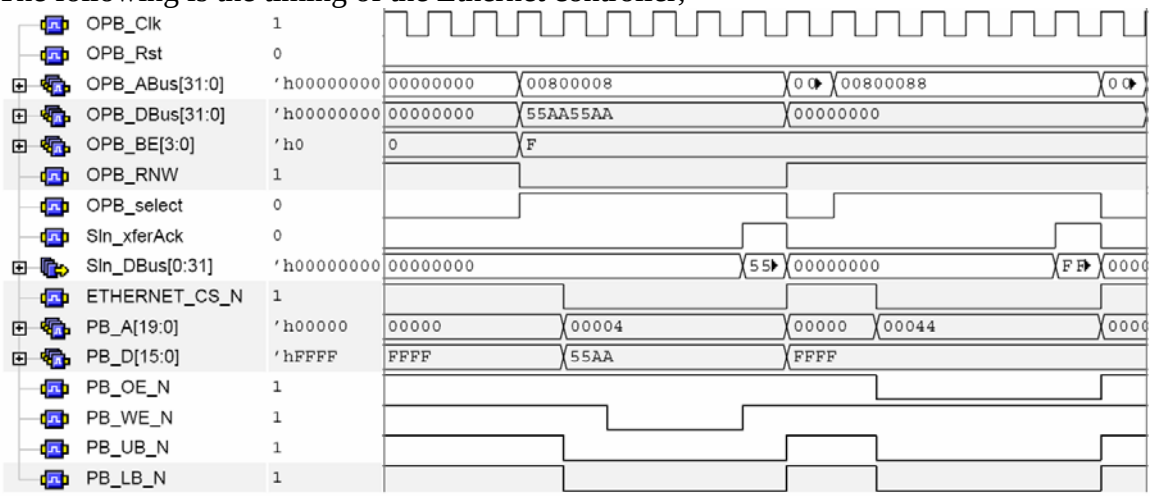
FPGA/Ethenet Connection

The Ethernet controller provides 16 bit interface to the processor. The following is the state machine of the Ethernet OPB interface,



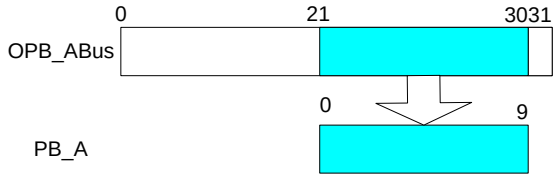
The Ethernet access provides longer wait time by additional WAIT states.

The following is the timing of the Ethernet controller,

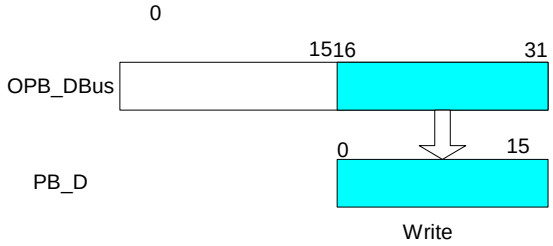


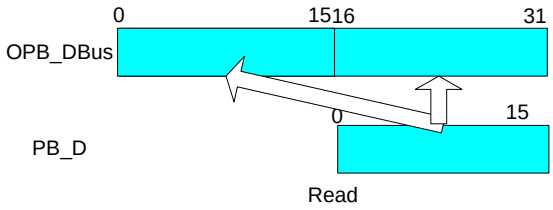
Ethernet Timing

The following is the address mapping between the OPB and PB bus,



The following is the data bus mapping,





The Ethernet controller has similar timing as the SRAM. The IO registers are mapped to halfword address in the OPB address space. So all the C accessible register address equals Ethernet chip defined address times two. The data bus width is fixed to be 16-bit, so that the access to the Ethernet is always halfword mode.

2.8 User Interface

The VGA display and UART is used to implement user interface. The operation of the server is controlled by the menu displayed on the VGA. The user select the menu command from the UART channel. Based on the user inputs, the operation of the server can be changed, such as switch between live broadcast and advertisement, record advertisement to Flash, etc.

The following is the menu configuration:

```
Live broadcast    -----------  Enable live broadcast
                         |
                         |----  Disable live broadcast
                         |
                         |----- Return


Advertisement Insertion ------ Start Ad insertion
                         |
                         |----- Return

Utility           ------------ Erase Flash buffer
                         |
                         |
                         |----- Record Ad to Flash
                         |
                         |
                         |----- Return
```

We use fixed address in the Flash to save the advertisement and back ground music. The size of the data is also fixed.

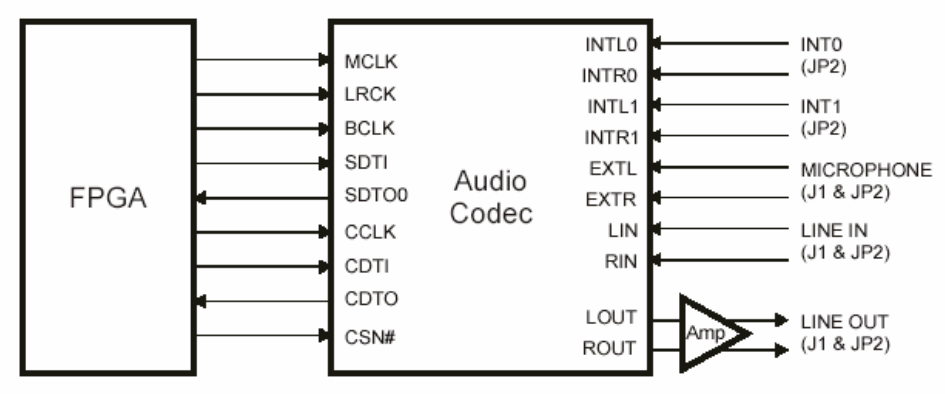
We use the text mode VGA. Because of time limit we didn't implement the graphics mode VGA for better visual effects.

The menu selection is controlled from the UART channel, and there are double right arrows pointing to the menu item which the user has chosen using up-down arrow key. The corresponding operation will be implemented after the user press enter.

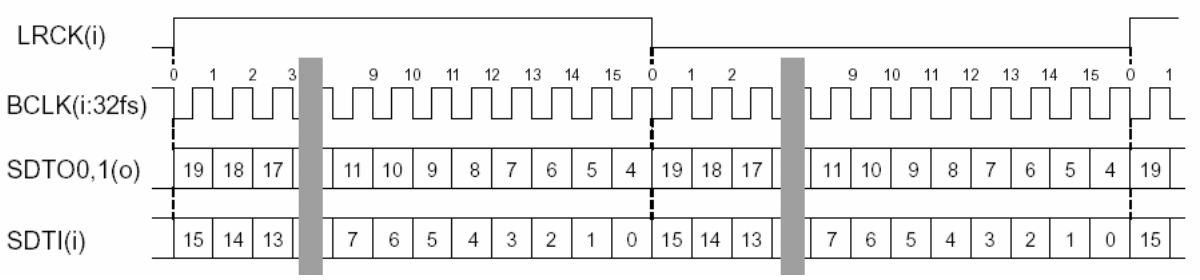
2.9 Live Broadcast

The audio input is taken from the AK4565 Audio Codec. The Line In is used to input sound for the broadcast. The audio codec interface is built in the FPGA, and serialized bit

stream is sent out to Ethernet controller onboard RAM. The header of packets is fixed. We send 32-bit sampler data to Ethernet as soon as one sampler interrupt occurs. Once the payload for one packet is filled. The Ethernet will transmit the constructed packet to Internet.



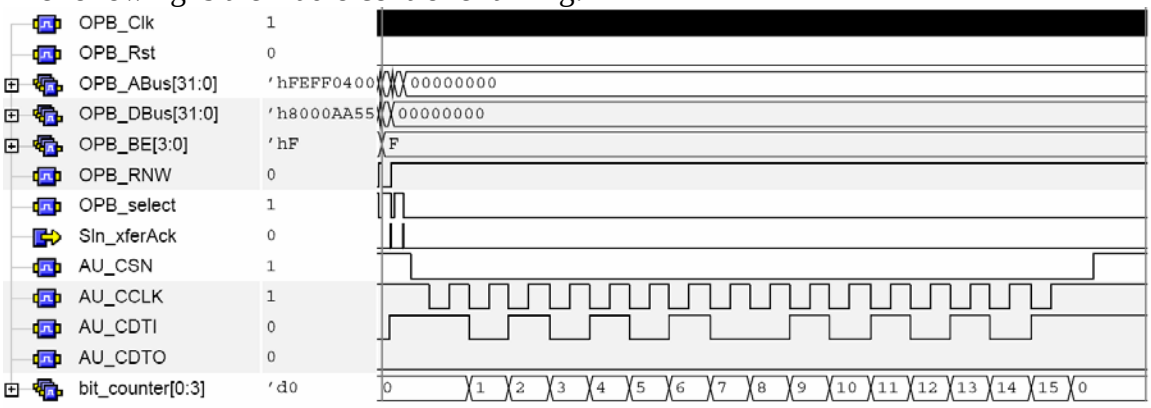
FPGA/Audio Codec Connection



Codec Timing Diagram

In order to configure the Audio Codec, we developed the Audio controller. It will handle the serial protocol between the FPGA and Codec. The FPGA starts the controller by writing to the Audio Control Register, then the OPB bus is switched to the Audio by the AU_BUSY signal to the MUX, and the serial data is downloaded to the Audio.

The following is the Audio controller timing:



2.10 Advertisement Insertion

The advertisement clip would be inserted to the broadcast. The source data for the advertisement is saved in the Flash memory in advance, and sent to the ethernet controller when the advertisement mode is enabled.
The Flash OPB interface need to be designed in the FPGA. The 8-bit Flash data bus to 32 bit processor data bus conversion will be done automatically. The Flash programming and erasing command will be started by the software, as well as the polling of the status.

2.11 Ad/Background Sound Recording

In order to facilitate the advertisement and background music feature, the line in will be sampled in the same way as the broadcasting, and the serialized codec data will be saved in the SRAM memory temporarily.
The size of the captured data will be in fixed length to simplify the design. The SRAM buffer is then programmed to the Flash memory, so that the contents are not lost after power down.

2.12 Packet Receving Debug Tools

We use several tools to test our audio packet sent out via Ethernet can be properly received.

Ethereal: sniffing packets on the network.
Rtpdump: rtpdump –F payload –t 0.5 –o test.pcm /3042
Dump packets from port 3042 to file test.pcm for 0.5 minutes
Cool Edit: play PCM audio file dumped by rtp tools
Mplayer: playing client
mplayer -demuxer rawaudio -rawaudio channels=2:rate=11800
rtp://128.59.144.169:3042

3. Conclusion

3.1 Who Did What And Lessons Learned

Yingjian Gu
I did the SRAM controller, Flash controller, MUX and SDRAM controller hardware design. I also did the Flash driver software, including the programming, erasing and polling routine. I did the SDRAM initialization software routine as well, and the memory access test code. The SRAM controller is verified by processor code running. The Flash controller is verified in the Audio application. The MUX is the key to implement and access all the peripherals in the same design. The SDRAM controller design works in test code, but fails in extensive data application as the Audio recording. I did not realize that the SDRAM controller had multiple clock domain issue, and had no synchronization logic between clock domains at the beginning. This caused the SDRAM controller did not work stable, and system hang up some time.

Qiutao Yu
I love embedded system design! I am really fascinated by the small XESS XSB-300E FPGA board. I am so gald that I took this course. Most of the system software design is done by me. First I spend countless hours trying to figure out the system principle of this project from last year's successfully implemented internet radio project. As I read their project report I realize this is really not an easy job. The guys of last year did very well. After I analyzed their C codes and well written report things become more and more clear. Basically our project should have improvement over last year so we add more function to the internet radio server built on the FPGA board.
Originally we plan to add background music to our live broadcast, create our own client software and use graphic VGA to display the user interface for better visual effects. We do not incorporate these to our final project because of time constraints. The BRAM is really too small for our project. When I design the menu interface to be displayed on VGA something really weird happens. There are just random characters on the VGA which is not expected and the program does not respond to any input from the uart. At first I do not realize this is because the length of the codes is out of the BRAM space. It is really painful when you debug your code and find even the sequence of two similar statements can have different consequence. I should have found out the reason much earlier. After I realized it is the problem of the BRAM limit I have to move most of our codes to SRAM which will lower down our program running speed such that we have to lower down the sampling frequence of the audio codes in order to the frequence of interrupt handling. We use the audio chip Line In as audio input and create our own audio controller but it does not work properly so I found a tool from XSTOOL provided by XESS to configure the audio chip with specified sampling frequency and input method first before the bit stream is downloaded to FPGA. I find a lot of fun when I successfully make my own program work after days of debugging.

Chun-Chuen Li
Teamwork is really, really important! There are times we missed meeting each other, and got a bit lazy. However, it was great we came back with new ideas toward the project. I learned to criticize my teammate's work and provided suggestion to have the C code run more efficient. Working in team also helps alleviate anxiety built from trying to understanding things on our own, as reading specifications and think "worldly" in this real world like project environment. I was trying to figure out how to vary the rate of the audio output by changing the MPlayer. Out of the blue, I found out an answer to it. The real world is vast, yet there is somewhere we can fetch for the facts. This is what I learned from this team project.

Imran Quyyum
I help with team to create the audio sampler OPB peripheral to sample the audio data in. The distinction between computer programming and computer engineering is hard to grasp until this course is taken. The programming style that the Intro. to Computer Science taught becomes less relevant and FSM diagrams and debugging soon top the list of things that makes you question sometimes whether engineering is really for you or not. But when the project actually works, it's an overwhelming feeling of joy and you will be glad that you took the challenge.

3.2 Future Work

There can be a lot of things to do for future enhancement of our project. It would be fine to create the client on our own not just the Mplayer we use. For this project we only broadcast out the rtp packets to a computer with destination IP and MAC address specified in the header of the packets. That the audio packets can be received anywhere with connection to Internet would be great. Also more function such as mixing the live broadcast with background music can be added. If the audio sampling frequency can be increased without affecting the execution of the main program the sound quality would be much improved.

4. Code Listings

4.1 Configuration Files

4.1.1 system.mhs

```
#-----------------------------------------------#
# CSEE 4840 Embedded System Design    #
#                                               #
#  Internet Radio                               #
#                                               #
#  Team Members:      Qiutao Yu                 #
#                     Yingjian Gu               #
#                     Chun-Chuen Li             #
#                      Imran Quyyum             #
#-----------------------------------------------#

# Parameters
 PARAMETER VERSION = 2.1.0


# Global Ports
PORT FPGA_CLK1 = FPGA_CLK1, DIR = IN
PORT RS232_TD = RS232_TD, DIR = OUT
PORT RS232_RD = RS232_RD, DIR = IN
#PORT RIGHT_LED = RIGHT_LED, DIR = OUT, VEC = [7:0]
#PORT LEFT_LED = LEFT_LED, DIR = OUT, VEC = [7:0]
#PORT BAR_LED = BAR_LED, DIR = OUT, VEC = [9:0]
PORT VIDOUT_CLK = VIDOUT_CLK, DIR = OUT
PORT VIDOUT_HSYNC_N = VIDOUT_HSYNC_N, DIR = OUT
PORT VIDOUT_VSYNC_N = VIDOUT_VSYNC_N, DIR = OUT
PORT VIDOUT_BLANK_N = VIDOUT_BLANK_N, DIR = OUT
PORT VIDOUT_RED = VIDOUT_RED, DIR = OUT, VEC = [9:0]
PORT VIDOUT_GREEN = VIDOUT_GREEN, DIR = OUT, VEC = [9:0]
PORT VIDOUT_BLUE = VIDOUT_BLUE, DIR = OUT, VEC = [9:0]

# PH bus ports
PORT PB_D = PB_D, DIR = INOUT, VEC=[15:0]
PORT PB_A = PB_A, DIR = OUT,   VEC=[19:0]
PORT PB_OE_N = PB_OE_N, DIR = OUT
PORT PB_WE_N = PB_WE_N, DIR = OUT
PORT PB_UB_N = PB_UB_N, DIR = OUT
PORT PB_LB_N = PB_LB_N, DIR = OUT
PORT RAM_CE_N = RAM_CE_N, DIR = OUT

# Ethernet Ports
```

```
PORT ETHERNET_CS_N = ETHERNET_CS_N, DIR = OUT
PORT ETHERNET_IOCS16_N = ETHERNET_IOCS16_N, DIR = IN
PORT ETHERNET_RDY = ETHERNET_RDY, DIR = IN
PORT ETHERNET_IREQ = ETHERNET_IREQ, DIR = IN

# Audio Codec Ports
PORT AU_CSN = AU_CSN, DIR=OUT
PORT AU_MCLK  = AU_MCLK, DIR=OUT
PORT AU_LRCK  = AU_LRCK, DIR=OUT
PORT AU_BCLK  = AU_BCLK, DIR=OUT
PORT AU_SDTI  = AU_SDTI, DIR=OUT
PORT AU_SDTO0 = AU_SDTO0, DIR=IN

#SDRAM Ports
PORT SDRAM_CKE = SDRAM_CKE , DIR = OUT
PORT SDRAM_CAS_N = SDRAM_CAS_N , DIR = OUT
PORT SDRAM_RAS_N = SDRAM_RAS_N , DIR = OUT
PORT SDRAM_CE_N = SDRAM_CE_N , DIR = OUT
PORT SDRAM_WE_N = SDRAM_WE_N , DIR = OUT
PORT SDRAM_CLK = SDRAM_CLK, DIR = IN

#Flash Ports
PORT FLASH_CE_N = FLASH_CE_N, DIR = OUT


# Hint: Put your peripheral first in this file so it will be analyzed
# first and will generate errors faster.
# BRAM example peripheral

BEGIN opb_pb_mux
 PARAMETER INSTANCE = pb_mux
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xFFEF0500
 PARAMETER C_HIGHADDR = 0xFFEF05FF
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT PB_D = PB_D
 PORT PB_A = PB_A
 PORT PB_OE_N = PB_OE_N
 PORT PB_WE_N = PB_WE_N
 PORT PB_UB_N = PB_UB_N
 PORT PB_LB_N = PB_LB_N

 PORT PB_DIN = PB_DIN
 PORT PB_DOUT_SRAM = PB_DOUT_SRAM
 PORT PB_DOUT_EN_SRAM = PB_DOUT_EN_SRAM
```

```
 PORT PB_A_SRAM = PB_A_SRAM
 PORT PB_OE_N_SRAM = PB_OE_N_SRAM
 PORT PB_WE_N_SRAM = PB_WE_N_SRAM
 PORT PB_UB_N_SRAM = PB_UB_N_SRAM
 PORT PB_LB_N_SRAM = PB_LB_N_SRAM

 PORT PB_DOUT_FLASH = PB_DOUT_FLASH
 PORT PB_DOUT_EN_FLASH = PB_DOUT_EN_FLASH
 PORT PB_A_FLASH = PB_A_FLASH
 PORT PB_OE_N_FLASH = PB_OE_N_FLASH
 PORT PB_WE_N_FLASH = PB_WE_N_FLASH

 PORT PB_DOUT_SDRAM = PB_DOUT_SDRAM
 PORT PB_DOUT_EN_SDRAM = PB_DOUT_EN_SDRAM
 PORT PB_A_SDRAM = PB_A_SDRAM
 PORT PB_UB_N_SDRAM = PB_UB_N_SDRAM
 PORT PB_LB_N_SDRAM = PB_LB_N_SDRAM

 PORT PB_DOUT_ETHER = PB_DOUT_ETHER
 PORT PB_DOUT_EN_ETHER = PB_DOUT_EN_ETHER
 PORT PB_A_ETHER = PB_A_ETHER
 PORT PB_OE_N_ETHER = PB_OE_N_ETHER
 PORT PB_WE_N_ETHER = PB_WE_N_ETHER
 PORT PB_UB_N_ETHER = PB_UB_N_ETHER
 PORT PB_LB_N_ETHER = PB_LB_N_ETHER

 PORT AU_BUSY = AU_BUSY
 PORT AU_CCLK = AU_CCLK
 PORT AU_CDTI = AU_CDTI

 PORT SDRAM_LOCK = SDRAM_LOCK
 PORT SDRAM_ACTIVE = SDRAM_ACTIVE

END

BEGIN opb_sram
 PARAMETER INSTANCE = sram
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x20000000
 PARAMETER C_HIGHADDR = 0x2FFFFFFF
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT PB_DIN   = PB_DIN
 PORT PB_DOUT  = PB_DOUT_SRAM
 PORT PB_DOUT_EN = PB_DOUT_EN_SRAM
 PORT PB_A     = PB_A_SRAM
```

```
 PORT RAM_CE   = RAM_CE_N
 PORT PB_OE_N    = PB_OE_N_SRAM
 PORT PB_WE_N    = PB_WE_N_SRAM
 PORT PB_UB_N    = PB_UB_N_SRAM
 PORT PB_LB_N    = PB_LB_N_SRAM
END

BEGIN opb_flash
 PARAMETER INSTANCE = flash
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x30000000
 PARAMETER C_HIGHADDR = 0x3FFFFFFF
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT PB_DIN   = PB_DIN
 PORT PB_DOUT  = PB_DOUT_FLASH
 PORT PB_DOUT_EN = PB_DOUT_EN_FLASH
 PORT PB_A     = PB_A_FLASH
 PORT PB_OE_N    = PB_OE_N_FLASH
 PORT PB_WE_N    = PB_WE_N_FLASH
 PORT FLASH_CE_N = FLASH_CE_N
END

BEGIN opb_sdram
 PARAMETER INSTANCE = sdram
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x40000000
 PARAMETER C_HIGHADDR = 0x4FFFFFFF
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT PB_DIN   = PB_DIN
 PORT PB_DOUT  = PB_DOUT_SDRAM
 PORT PB_DOUT_EN = PB_DOUT_EN_SDRAM
 PORT PB_A     = PB_A_SDRAM
 PORT SDRAM_CKE = SDRAM_CKE
 PORT SDRAM_CAS_N = SDRAM_CAS_N
 PORT SDRAM_RAS_N = SDRAM_RAS_N
 PORT SDRAM_CE_N = SDRAM_CE_N
 PORT SDRAM_WE_N = SDRAM_WE_N
 PORT PB_UB_N = PB_UB_N_SDRAM
 PORT PB_LB_N = PB_LB_N_SDRAM
 PORT SDRAM_LOCK = SDRAM_LOCK
 PORT SDRAM_ACTIVE = SDRAM_ACTIVE
 PORT SDRAM_CLK = SDRAM_CLK
END
```

```
# Useful trick: put the IP you are developing first in this list
# so that it is compiled (and therefore may fail) before anything else
# Video Controller
BEGIN opb_xsb300e_vga
 PARAMETER INSTANCE = vga
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xFEFF1000
 PARAMETER C_HIGHADDR = 0xFEFF1fff
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT pixel_clock = pixel_clock
 PORT VIDOUT_CLK = VIDOUT_CLK
 PORT VIDOUT_RED = VIDOUT_RED
 PORT VIDOUT_GREEN = VIDOUT_GREEN
 PORT VIDOUT_BLUE = VIDOUT_BLUE
 PORT VIDOUT_BLANK_N = VIDOUT_BLANK_N
 PORT VIDOUT_HSYNC_N = VIDOUT_HSYNC_N
 PORT VIDOUT_VSYNC_N = VIDOUT_VSYNC_N
END


# Ethernet peripheral

BEGIN opb_ethernet
 PARAMETER INSTANCE = ethernet_peripheral
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x00800000
 PARAMETER C_HIGHADDR = 0x00FFFFFF
 PORT OPB_CLK = sys_clk
 BUS_INTERFACE SOPB = myopb_bus

 PORT ETHERNET_CS_N = ETHERNET_CS_N
 PORT ETHERNET_RDY = ETHERNET_RDY
 PORT ETHERNET_IREQ = ETHERNET_IREQ
 PORT ETHERNET_IOCS16_N = ETHERNET_IOCS16_N

 PORT PB_DIN = PB_DIN
 PORT PB_DOUT = PB_DOUT_ETHER
 PORT PB_DOUT_EN = PB_DOUT_EN_ETHER
 PORT PB_A = PB_A_ETHER
 PORT IORD_N = PB_OE_N_ETHER
 PORT IOWR_N = PB_WE_N_ETHER
 PORT BHE_N = PB_UB_N_ETHER
 PORT AEN = PB_LB_N_ETHER

END
```

```
# Audio Sampler Peripheral
BEGIN opb_audio_sampler
 PARAMETER INSTANCE = audio_sampler
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xFEFF0300
 PARAMETER C_HIGHADDR = 0xFEFF03FF
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT AU_MCLK   = AU_MCLK
 PORT AU_LRCK   = AU_LRCK
 PORT AU_BCLK   = AU_BCLK
 PORT AU_SDTI   = AU_SDTI
 PORT AU_SDTO0  = AU_SDTO0
 PORT INTERRUPT = sampler_intr
END

# Audio Controller Peripheral
BEGIN opb_audio_controller
 PARAMETER INSTANCE = audio_controller
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xFEFF0400
 PARAMETER C_HIGHADDR = 0xFEFF04FF
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT AU_BUSY = AU_BUSY
 PORT AU_CSN = AU_CSN
 PORT AU_CCLK = AU_CCLK
 PORT AU_CDTI = AU_CDTI
 PORT PB_DIN = PB_DIN
END

# Interrupt controller for dealing with interrupts from the UART
BEGIN opb_intc
 PARAMETER INSTANCE = intc
 PARAMETER HW_VER = 1.00.c
 PARAMETER C_BASEADDR = 0xFFFF0000
 PARAMETER C_HIGHADDR = 0xFFFF00FF
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT Intr = uart_intr & sampler_intr
 PORT Irq = intr
END

# Block RAM for code and data is connected through two LMB busses
# to the Microblaze, which has two ports on it for just this reason.
```

```
# Data LMB bus
BEGIN lmb_v10
 PARAMETER INSTANCE = d_lmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = sys_clk
 PORT SYS_Rst = fpga_reset
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = lmb_data_controller
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00000FFF
 BUS_INTERFACE SLMB = d_lmb
 BUS_INTERFACE BRAM_PORT = conn_0
END

# Instruction LMB bus
BEGIN lmb_v10
 PARAMETER INSTANCE = i_lmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = sys_clk
 PORT SYS_Rst = fpga_reset
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = lmb_instruction_controller
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00000FFF
 BUS_INTERFACE SLMB = i_lmb
 BUS_INTERFACE BRAM_PORT = conn_1
END

# The actual block memory
BEGIN bram_block
 PARAMETER INSTANCE = bram
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = conn_0
 BUS_INTERFACE PORTB = conn_1
END

# Clock divider to make the whole thing run
BEGIN clkgen
 PARAMETER INSTANCE = clkgen_0
 PARAMETER HW_VER = 1.00.a
```

```
 PORT FPGA_CLK1 = FPGA_CLK1
 PORT sys_clk = sys_clk
 PORT pixel_clock = pixel_clock
 PORT fpga_reset = fpga_reset
END

# The OPB bus controller connected to the Microblaze
# All peripherals are connected to this
BEGIN opb_v20
 PARAMETER INSTANCE = myopb_bus
 PARAMETER HW_VER = 1.10.a
 PARAMETER C_DYNAM_PRIORITY = 0
 PARAMETER C_REG_GRANTS = 0
 PARAMETER C_PARK = 0
 PARAMETER C_PROC_INTRFCE = 0
 PARAMETER C_DEV_BLK_ID = 0
 PARAMETER C_DEV_MIR_ENABLE = 0
 PARAMETER C_BASEADDR = 0x0fff1000
 PARAMETER C_HIGHADDR = 0x0fff10ff
 PORT SYS_Rst = fpga_reset
 PORT OPB_Clk = sys_clk
END

# UART: Serial port hardware
BEGIN opb_uartlite
 PARAMETER INSTANCE = myuart
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_CLK_FREQ = 50_000_000
 PARAMETER C_USE_PARITY = 0
 PARAMETER C_BASEADDR = 0xFEFF0100
 PARAMETER C_HIGHADDR = 0xFEFF01FF
 BUS_INTERFACE SOPB = myopb_bus
 PORT OPB_Clk = sys_clk
 PORT Interrupt = uart_intr
 PORT RX = RS232_RD
 PORT TX = RS232_TD
END

# Very simple LED controller for the XESS XSB-300E board
#BEGIN opb_xsbleds
# PARAMETER INSTANCE = leds
# PARAMETER HW_VER = 1.00.a
# PARAMETER C_BASEADDR = 0xFEFF0200
# PARAMETER C_HIGHADDR = 0xFEFF02ff
# BUS_INTERFACE SOPB = myopb_bus
# PORT OPB_Clk = sys_clk
```

```
# PORT RIGHT_LED = RIGHT_LED
# PORT LEFT_LED = LEFT_LED
# PORT BAR_LED = BAR_LED
#END

# The main processor core
BEGIN microblaze
 PARAMETER INSTANCE = mymicroblaze
 PARAMETER HW_VER = 3.00.a
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_USE_ICACHE = 0
 BUS_INTERFACE DLMB = d_lmb
 BUS_INTERFACE ILMB = i_lmb
 BUS_INTERFACE DOPB = myopb_bus
 BUS_INTERFACE IOPB = myopb_bus
 PORT Clk = sys_clk
 PORT Reset = fpga_reset
 PORT Interrupt = intr
END
```

4.1.2 system.mss

```
#-----------------------------------------------#
# CSEE 4840 Embedded System Design    #
#                                               #
#  Internet Radio                               #
#                                               #
#  Team Members:      Qiutao Yu                 #
#                     Yingjian Gu               #
#                     Chun-Chuen Li             #
#                      Imran Quyyum             #
#-----------------------------------------------#

 PARAMETER VERSION = 2.2.0
 PARAMETER HW_SPEC_FILE = system.mhs


BEGIN OS
 PARAMETER PROC_INSTANCE = mymicroblaze
 PARAMETER OS_NAME = standalone
 PARAMETER OS_VER = 1.00.a
 PARAMETER STDIN = myuart
 PARAMETER STDOUT = myuart
END
```

```
BEGIN PROCESSOR
 PARAMETER HW_INSTANCE = mymicroblaze
 PARAMETER DRIVER_NAME = cpu
 PARAMETER DRIVER_VER = 1.00.a
END


BEGIN DRIVER
 PARAMETER HW_INSTANCE = myuart
 PARAMETER DRIVER_NAME = uartlite
 PARAMETER DRIVER_VER = 1.00.b
END

BEGIN DRIVER
 PARAMETER HW_INSTANCE = intc
 PARAMETER DRIVER_NAME = intc
 PARAMETER DRIVER_VER = 1.00.c
END

# Assign a null driver for hardware that doesn't need one,
# mostly to disable warnings
BEGIN DRIVER
 PARAMETER HW_INSTANCE = vga
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
END

BEGIN DRIVER
 PARAMETER HW_INSTANCE = lmb_data_controller
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
END

BEGIN DRIVER
 PARAMETER HW_INSTANCE = lmb_instruction_controller
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
END
```

4.1.3 system.ucf

```
net FPGA_CLK1 loc="p77";

net RS232_TD loc="p71";
net RS232_RD loc="p73";
```

```
net VIDOUT_CLK loc="p23";
net VIDOUT_BLANK_N loc="p24";
net VIDOUT_HSYNC_N loc="p8";
net VIDOUT_VSYNC_N loc="p7";

net VIDOUT_RED<0> loc="p41";
net VIDOUT_RED<1> loc="p40";
net VIDOUT_RED<2> loc="p36";
net VIDOUT_RED<3> loc="p35";
net VIDOUT_RED<4> loc="p34";
net VIDOUT_RED<5> loc="p33";
net VIDOUT_RED<6> loc="p31";
net VIDOUT_RED<7> loc="p30";
net VIDOUT_RED<8> loc="p29";
net VIDOUT_RED<9> loc="p27";

net VIDOUT_GREEN<0> loc="p9" ;
net VIDOUT_GREEN<1> loc="p10";
net VIDOUT_GREEN<2> loc="p11";
net VIDOUT_GREEN<3> loc="p15";
net VIDOUT_GREEN<4> loc="p16";
net VIDOUT_GREEN<5> loc="p17";
net VIDOUT_GREEN<6> loc="p18";
net VIDOUT_GREEN<7> loc="p20";
net VIDOUT_GREEN<8> loc="p21";
net VIDOUT_GREEN<9> loc="p22";

net VIDOUT_BLUE<0> loc="p42";
net VIDOUT_BLUE<1> loc="p43";
net VIDOUT_BLUE<2> loc="p44";
net VIDOUT_BLUE<3> loc="p45";
net VIDOUT_BLUE<4> loc="p46";
net VIDOUT_BLUE<5> loc="p47";
net VIDOUT_BLUE<6> loc="p48";
net VIDOUT_BLUE<7> loc="p49";
net VIDOUT_BLUE<8> loc="p55";
net VIDOUT_BLUE<9> loc="p56";


# Ethernet
net ETHERNET_CS_N loc="p82";
net ETHERNET_RDY loc="p81";
net ETHERNET_IREQ loc="p75";
net ETHERNET_IOCS16_N loc="p74";
```

```
# OPB_ETHERNET
net PB_OE_N  loc="p125";
net PB_WE_N  loc="p123";
net PB_UB_N  loc="p146";
net PB_LB_N  loc="p140";

net RAM_CE_N loc="p147";

# OPB_Data
net PB_D<0> loc="p153";
net PB_D<1> loc="p145";
net PB_D<2> loc="p141";
net PB_D<3> loc="p135";
net PB_D<4> loc="p126";
net PB_D<5> loc="p120";
net PB_D<6> loc="p116";
net PB_D<7> loc="p108";
net PB_D<8> loc="p127";
net PB_D<9> loc="p129";
net PB_D<10> loc="p132";
net PB_D<11> loc="p133";
net PB_D<12> loc="p134";
net PB_D<13> loc="p136";
net PB_D<14> loc="p138";
net PB_D<15> loc="p139";

#OPB_Address
net PB_A<0> loc="p83";
net PB_A<1> loc="p84";
net PB_A<2> loc="p86";
net PB_A<3> loc="p87";
net PB_A<4> loc="p88";
net PB_A<5> loc="p89";
net PB_A<6> loc="p93";
net PB_A<7> loc="p94";
net PB_A<8> loc="p100";
net PB_A<9> loc="p101";
net PB_A<10> loc="p102";
net PB_A<11> loc="p109";
net PB_A<12> loc="p110";
net PB_A<13> loc="p111";
net PB_A<14> loc="p112";
net PB_A<15> loc="p113";
net PB_A<16> loc="p114";
net PB_A<17> loc="p115";
net PB_A<17> loc="p115";
```

```
net PB_A<18> loc="p121";
net PB_A<19> loc="p122";

# Audio Codec
net AU_CSN   loc="p165";
net AU_MCLK  loc="p167";
net AU_LRCK  loc="p168";
net AU_BCLK  loc="p166";
net AU_SDTI  loc="p169";
net AU_SDTO0 loc="p173";

#SDRAM
net SDRAM_CKE loc = "p99";
net SDRAM_CAS_N loc = "p96";
net SDRAM_RAS_N loc = "p97";
net SDRAM_CE_N loc = "p98";
net SDRAM_WE_N loc = "p95";
net SDRAM_CLK loc = "p80";

#Flash
net FLASH_CE_N loc = "p57";

NET "SDRAM_CLK" TNM_NET = "SDRAM_CLK";
TIMESPEC "TS_SDRAM_CLK" = PERIOD "SDRAM_CLK" 10 ns HIGH 50 %;

NET "PB_A<10>" DRIVE = 24;
NET "PB_A<0>" DRIVE = 24;
NET "PB_A<1>" DRIVE = 24;
NET "PB_A<2>" DRIVE = 24;
NET "PB_A<3>" DRIVE = 24;
NET "PB_A<4>" DRIVE = 24;
NET "PB_A<5>" DRIVE = 24;
NET "PB_A<6>" DRIVE = 24;
NET "PB_A<7>" DRIVE = 24;
NET "PB_A<8>" DRIVE = 24;
NET "PB_A<9>" DRIVE = 24;
NET "PB_A<11>" DRIVE = 24;
NET "PB_A<12>" DRIVE = 24;
NET "PB_A<13>" DRIVE = 24;
NET "PB_A<14>" DRIVE = 24;
NET "PB_D<0>" DRIVE = 24;
NET "PB_D<1>" DRIVE = 24;
NET "PB_D<2>" DRIVE = 24;
NET "PB_D<3>" DRIVE = 24;
NET "PB_D<4>" DRIVE = 24;
NET "PB_D<5>" DRIVE = 24;
```

```
NET "PB_D<6>" DRIVE = 24;
NET "PB_D<7>" DRIVE = 24;
NET "PB_D<8>" DRIVE = 24;
NET "PB_D<9>" DRIVE = 24;
NET "PB_D<10>" DRIVE = 24;
NET "PB_D<11>" DRIVE = 24;
NET "PB_D<12>" DRIVE = 24;
NET "PB_D<13>" DRIVE = 24;
NET "PB_D<14>" DRIVE = 24;
NET "PB_D<15>" DRIVE = 24;
NET "PB_UB_N" DRIVE = 24;
NET "PB_LB_N" DRIVE = 24;
NET "SDRAM_CAS_N" DRIVE = 24;
NET "SDRAM_CE_N" DRIVE = 24;
NET "SDRAM_CKE" DRIVE = 24;
NET "SDRAM_RAS_N" DRIVE = 24;
NET "SDRAM_WE_N" DRIVE = 24;
NET "PB_A<0>" FAST;
NET "PB_A<1>" FAST;
NET "PB_A<2>" FAST;
NET "PB_A<3>" FAST;
NET "PB_A<4>" FAST;
NET "PB_A<5>" FAST;
NET "PB_A<6>" FAST;
NET "PB_A<7>" FAST;
NET "PB_A<8>" FAST;
NET "PB_A<9>" FAST;
NET "PB_A<10>" FAST;
NET "PB_A<11>" FAST;
NET "PB_A<12>" FAST;
NET "PB_A<13>" FAST;
NET "PB_A<14>" FAST;
NET "PB_D<0>" FAST;
NET "PB_D<1>" FAST;
NET "PB_D<2>" FAST;
NET "PB_D<3>" FAST;
NET "PB_D<4>" FAST;
NET "PB_D<5>" FAST;
NET "PB_D<6>" FAST;
NET "PB_D<7>" FAST;
NET "PB_D<8>" FAST;
NET "PB_D<9>" FAST;
NET "PB_D<10>" FAST;
NET "PB_D<11>" FAST;
NET "PB_D<12>" FAST;
NET "PB_D<13>" FAST;
```

```
NET "PB_D<14>" FAST;
NET "PB_D<15>" FAST;
NET "PB_UB_N" FAST;
NET "PB_LB_N" FAST;
NET "SDRAM_CAS_N" FAST;
NET "SDRAM_CE_N" FAST;
NET "SDRAM_CKE" FAST;
NET "SDRAM_RAS_N" FAST;
NET "SDRAM_WE_N" FAST;
```

4.1.4 Makefile

```
#-------------------------------------------------#
# CSEE 4840 Embedded System Design    #
#                                                 #
#  Internet Radio                                 #
#                                                 #
#  Team Members:      Qiutao Yu                   #
#                     Yingjian Gu                 #
#                     Chun-Chuen Li               #
#                      Imran Quyyum               #
#-------------------------------------------------#


SYSTEM = system

MICROBLAZE_OBJS = \
        c_source_files/audio.o \
        c_source_files/flash.o \
        c_source_files/main.o \
        c_source_files/menu.o \
        c_source_files/initialization.o \
        c_source_files/int.o \
        c_source_files/ether.o

LIBRARIES = mymicroblaze/lib/libxil.a

ELF_FILE = $(SYSTEM).elf

NETLIST = implementation/$(SYSTEM).ngc

# Bitstreams for the FPGA

FPGA_BITFILE = implementation/$(SYSTEM).bit
MERGED_BITFILE = implementation/download.bit
```

```
# Files to be downloaded to the SRAM

SRAM_BINFILE = implementation/sram.bin
SRAM_HEXFILE = implementation/sram.hex

MHSFILE = $(SYSTEM).mhs
MSSFILE = $(SYSTEM).mss

FPGA_ARCH = spartan2e
DEVICE = xc2s300epq208-6

LANGUAGE = vhdl
PLATGEN_OPTIONS = -p $(FPGA_ARCH) -lang $(LANGUAGE)
LIBGEN_OPTIONS = -p $(FPGA_ARCH) $(MICROBLAZE_LIBG_OPT)

# Paths for programs

XILINX = /usr/cad/xilinx/ise7.1i
ISEBINDIR = $(XILINX)/bin/lin
ISEENVCMDS = LD_LIBRARY_PATH=$(ISEBINDIR) XILINX=$(XILINX)
PATH=$(ISEBINDIR)

XILINX_EDK = /usr/cad/xilinx/edk7.1i

MICROBLAZE = /usr/cad/xilinx/edk7.1i/gnu/microblaze/lin
MBBINDIR = $(MICROBLAZE)/bin
XESSBINDIR = /usr/cad/xess/bin

# Executables

XST = $(ISEENVCMDS) $(ISEBINDIR)/xst
XFLOW = $(ISEENVCMDS) $(ISEBINDIR)/xflow
BITGEN = $(ISEENVCMDS) $(ISEBINDIR)/bitgen
DATA2MEM = $(ISEENVCMDS) $(ISEBINDIR)/data2mem
XSLOAD = $(XESSBINDIR)/xsload
XESS_BOARD = XSB-300E

MICROBLAZE_CC = $(MBBINDIR)/mb-gcc
MICROBLAZE_LD = $(MBBINDIR)/mb-ld
MICROBLAZE_CC_SIZE = $(MBBINDIR)/mb-size
MICROBLAZE_OBJCOPY = $(MBBINDIR)/mb-objcopy

# External Targets

all :
        @echo "Makefile to build a Microprocessor system :"
```

```
	@echo "Run make with any of the following targets"
	@echo "  make libs     : Configures the sw libraries for this system"
	@echo "  make program  : Compiles the program sources for all the processor
instances"
	@echo "  make netlist  : Generates the netlist for this system ($(SYSTEM))"
	@echo "  make bits     : Runs Implementation tools to generate the bitstream"
	@echo "  make init_bram: Initializes bitstream with BRAM data"
	@echo "  make download : Downloads the bitstream onto the board"
	@echo "  make netlistclean: Deletes netlist"
	@echo "  make hwclean  : Deletes implementation dir"
	@echo "  make libsclean: Deletes sw libraries"
	@echo "  make programclean: Deletes compiled ELF files"
	@echo "  make clean    : Deletes all generated files/directories"
	@echo " "
	@echo "  make <target> : (Default)"
	@echo "      Creates a Microprocessor system using default initializations"
	@echo "      specified for each processor in MSS file"


#bits : $(FPGA_BITFILE)

netlist : $(NETLIST)

libs : $(LIBRARIES)

program : $(ELF_FILE)

init_bram : $(MERGED_BITFILE)

clean : hwclean libsclean programclean
	rm -f bram_init.sh
	rm -f _impact.cmd

hwclean : netlistclean
	rm -rf implementation synthesis xst hdl
	rm -rf xst.srp $(SYSTEM).srp

netlistclean :
	rm -f $(FPGA_BITFILE) $(MERGED_BITFILE) \
	  $(NETLIST) implementation/$(SYSTEM)_bd.bmm

libsclean :
	rm -rf mymicroblaze/lib

programclean :
	rm -f $(ELF_FILE) $(SRAM_BITFILE) $(SRAM_HEXFILE)
```

```
#
# Software rules
#

MICROBLAZE_MODE = executable

# Assemble software libraries from the .mss and .mhs files

$(LIBRARIES) : $(MHSFILE) $(MSSFILE)
#	PATH=$$PATH:$(MBBINDIR) XILINX=$(XILINX)
XILINX_EDK=$(XILINX_EDK) \
#	 perl -I $(XILINX_EDK)/bin/nt/perl5lib $(XILINX_EDK)/bin/nt/libgen.pl \
#	  $(LIBGEN_OPTIONS) $(MSSFILE)
	@echo "Fetal ERROR! Library should be generated by the XPS GUI."

# Compilation

MICROBLAZE_CC_CFLAGS =
MICROBLAZE_CC_OPT = -O3 #-mxl-gp-opt
MICROBLAZE_CC_DEBUG_FLAG =#  -gstabs
MICROBLAZE_INCLUDES = -I./mymicroblaze/include/ # -I
MICROBLAZE_CFLAGS = \
	$(MICROBLAZE_CC_CFLAGS)\
	-mxl-barrel-shift \
	$(MICROBLAZE_CC_OPT) \
	$(MICROBLAZE_CC_DEBUG_FLAG) \
	$(MICROBLAZE_INCLUDES)

$(MICROBLAZE_OBJS) : %.o : %.c
	PATH=$(MBBINDIR) $(MICROBLAZE_CC) $(MICROBLAZE_CFLAGS) -c
$< -o $@

# Linking

# Uncomment the following to make linker print locations for everything
# MICROBLAZE_LD_FLAGS = -Wl,-M
MICROBLAZE_LINKER_SCRIPT = -Wl,-T -Wl,linkscript
MICROBLAZE_LIBPATH = -L./mymicroblaze/lib/
MICROBLAZE_CC_START_ADDR_FLAG= -Wl,-defsym -
Wl,_TEXT_START_ADDR=0x00000000
MICROBLAZE_CC_STACK_SIZE_FLAG=  -Wl,-defsym -Wl,_STACK_SIZE=0x200
MICROBLAZE_LFLAGS = \
	 -xl-mode-$(MICROBLAZE_MODE) \
	$(MICROBLAZE_LD_FLAGS) \
	$(MICROBLAZE_LINKER_SCRIPT) \
```

```
	$(MICROBLAZE_LIBPATH) \
	$(MICROBLAZE_CC_START_ADDR_FLAG) \
	$(MICROBLAZE_CC_STACK_SIZE_FLAG)
#MICROBLAZE_LDFLAGS = -T linker -lc -lm -L./mymicroblaze/lib/ -lxil -defsym
_TEXT_START_ADDR=0x00000000 -defsym _STACK_SIZE=0x200
MICROBLAZE_LDFLAGS1 = -T linker -lc -lm
MICROBLAZE_LDFLAGS2 = -L./mymicroblaze/lib/ -lxil -defsym
_TEXT_START_ADDR=0x00000000 -defsym _STACK_SIZE=0x200

#$(ELF_FILE) :  $(LIBRARIES) $(MICROBLAZE_OBJS)
$(ELF_FILE) : $(MICROBLAZE_OBJS)
	PATH=$(MBBINDIR) $(MICROBLAZE_LD) $(MICROBLAZE_LDFLAGS1)
$(MICROBLAZE_OBJS) -o $(ELF_FILE) $(MICROBLAZE_LDFLAGS2)
	$(MICROBLAZE_CC_SIZE) $(ELF_FILE)
	rm -f c_source_files/*.o
#
# Hardware rules
#

# Hardware compilation : optimize the netlist, place and route

#$(FPGA_BITFILE) : $(NETLIST) \
#		etc/fast_runtime.opt etc/bitgen.ut data/$(SYSTEM).ucf
#	cp -f etc/bitgen.ut implementation/
#	cp -f etc/fast_runtime.opt implementation/
#	cp -f data/$(SYSTEM).ucf implementation/$(SYSTEM).ucf
#	$(XFLOW) -wd implementation -p $(DEVICE) -implement fast_runtime.opt \
#		$(SYSTEM).ngc
#	cd implementation; $(BITGEN) -f bitgen.ut $(SYSTEM)

# Hardware assembly: Create the netlist from the .mhs file

#$(NETLIST) : $(MHSFILE)
#	XILINX=$(XILINX) XILINX_EDK=$(XILINX_EDK) \
#	perl -I $(XILINX_EDK)/bin/nt/perl5lib $(XILINX_EDK)/bin/nt/platgen.pl \
#	  $(PLATGEN_OPTIONS) -st xst $(MHSFILE)
#	perl synth_modules.pl < synthesis/xst.scr > xst.scr
#	$(XST) -ifn xst.scr
#	rm -r xst xst.scr
#	$(XST) -ifn synthesis/$(SYSTEM).scr

#
# Downloading
#

# Add software code to the FPGA bitfile
```

```
#$(MERGED_BITFILE) : $(FPGA_BITFILE) $(ELF_FILE)
$(MERGED_BITFILE) : $(ELF_FILE)
	bitinit system.mhs  -pe mymicroblaze system.elf  \
	-bt implementation/system.bit -o implementation/download.bit

# Create a .hex file with data for the SRAM

$(SRAM_HEXFILE) : $(ELF_FILE)
	$(MICROBLAZE_OBJCOPY) \
		-j .sram_text -j .sdata2 -j .sdata -j .rodata -j .data \
	-O binary $(ELF_FILE) $(SRAM_BINFILE)
	./bin2hex  -a 20000000 < $(SRAM_BINFILE) > $(SRAM_HEXFILE)

# Download the files to the target board

download : $(MERGED_BITFILE) $(SRAM_HEXFILE)
	$(XSLOAD) -ram -b $(XESS_BOARD) $(SRAM_HEXFILE)
	$(XSLOAD) -fpga -b $(XESS_BOARD) $(MERGED_BITFILE)
```

4.2 C Code

4.2.1 audio.h

```
#define XPAR_AUDIO_CONTROLLER_BASEADDR 0xFEFF0400
#define XPAR_AUDIO_SAMPLER_BASEADDR 0xFEFF0300
// The subroutine to enable the Audio
unsigned long InitializeAudio (unsigned long command );
```

4.2.2 ether.h

```
#include "xio.h"
#include "xbasic_types.h"

#define dprint print

#ifndef BYTE
#define BYTE unsigned char
#endif
#ifndef WORD
#define WORD unsigned short
#endif

// define the size of a packet

#define MAC_HEADER_SIZE 14
#define IP_HEADER_SIZE 20
```

```
#define UDP_HEADER_SIZE 8
#define RTP_HEADER_SIZE 12

#define HEADER_SIZE
(MAC_HEADER_SIZE+IP_HEADER_SIZE+UDP_HEADER_SIZE+RTP_HEADER_
SIZE)

#define IP_LENGTH (PACKET_SIZE-MAC_HEADER_SIZE)
#define UDP_LENGTH (IP_LENGTH-IP_HEADER_SIZE)

#define IP_CHECKSUM_OFFSET 24
#define UDP_CHECKSUM_OFFSET 40
#define RTP_SEQNUM_OFFSET 44
#define RTP_TIMESTAMP_OFFSET 46

#define DATA_CHUNK_SIZE 16
#define DATA_CHUNKS_IN_PACKET 90

#define PAYLOAD_SIZE (DATA_CHUNKS_IN_PACKET*DATA_CHUNK_SIZE)
#define PACKET_SIZE (PAYLOAD_SIZE+HEADER_SIZE)

#define PACKET_BUFFERS 1


// NE2000 definitions
#define NIC_BASE (0x00800400) // Base I/O address of the NIC card
#define DATAPORT (0x10*2)
#define NE_RESET (0x1f*2)

// NIC page0 register offsets
#define CMDR    (0x00*2) // command register for read & write
#define PSTART  (0x01*2) // page start register for write
#define PSTOP   (0x02*2) // page stop register for write
#define BNRY    (0x03*2) // boundary reg for rd and wr
#define TPSR    (0x04*2) // tx start page start reg for wr
#define TBCR0   (0x05*2) // tx byte count 0 reg for wr
#define TBCR1   (0x06*2) // tx byte count 1 reg for wr
#define ISR     (0x07*2) // interrupt status reg for rd and wr
#define RSAR0   (0x08*2) // low byte of remote start addr
#define RSAR1   (0x09*2) // hi byte of remote start addr
#define RBCR0   (0x0A*2) // remote byte count reg 0 for wr
#define RBCR1   (0x0B*2) // remote byte count reg 1 for wr
#define RCR     (0x0C*2) // rx configuration reg for wr
#define TCR     (0x0D*2) // tx configuration reg for wr
#define DCR     (0x0E*2) // data configuration reg for wr
#define IMR     (0x0F*2) // interrupt mask reg for wr
```

```
// NIC page 1 register offsets
#define PAR0    (0x01*2) // physical addr reg 0 for rd and wr
#define CURR    (0x07*2) // current page reg for rd and wr
#define MAR0    (0x08*2) // multicast addr reg 0 for rd and WR

// Buffer Length and Field Definition Info
#define TXSTART  0x41            // Tx buffer start page
#define TXPAGES  8               // Pages for Tx buffer
#define RXSTART  (TXSTART+TXPAGES) // Rx buffer start page
#define RXSTOP   0x7e            // Rx buffer end page for word mode


#define BASE_ADDR (XPAR_ETHERNET_PERIPHERAL_BASEADDR+0x400)
// macros for reading and writting registers
#define outnic(addr, data) XIo_Out16(NIC_BASE+addr, data)
#define innic(addr) (XIo_In16(NIC_BASE+addr))

#define PACKET_END_ADDRESS PACKET_START_ADDRESS+PACKET_SIZE
#define PACKET_START_ADDRESS  (TXSTART << 8)

#define PAYLOAD_START_ADDRESS
PACKET_START_ADDRESS+HEADER_SIZE;
#define UDP_CHECKSUM_ADDRESS
PACKET_START_ADDRESS+UDP_CHECKSUM_OFFSET
#define RTP_SEQNUM_ADDRESS
PACKET_START_ADDRESS+RTP_SEQNUM_OFFSET


/* function prototypes, external interface. */
BYTE init();
BYTE output_sample(WORD *sample);
int diagnostics();
void wait(int mult);


/* symbolic names for various register bits */

#define ISR_PTX  0x02 // packet transmitted with no error
#define ISR_RXE  0x04 // recieve error
#define ISR_RDC  0x40 // Remote DMA Successful

#define CR_STOP  0x01
#define CR_START 0x02
#define CR_TXP   0x04
#define CR_READ  0x08
```

```
#define CR_WRITE 0x10
#define CR_ABORT 0x20
#define CR_COMPLETE 0x20
#define CR_PAGE0 0x00
#define CR_PAGE1 0x40
```

4.2.3 flash.h

```
#define FLASH_BASE_ADDR 0x30000000

// Flash program subroutine
void FlashProgram (unsigned long address, unsigned char data);

// Chip Erase subroutine
void FlashErase (void);
```

4.2.4 menu.h

```
#define FLASH_BASE_ADDR 0x30000000

// Flash program subroutine
void FlashProgram (unsigned long address, unsigned char data);

// Chip Erase subroutine
void FlashErase (void);
```

4.2.5 flash.c

```
#include "xsbmemory.h"


// The Flash polling routine checks the Flash response
// Returns when the program or erase operation is complete
int FlashPolling(void)
{
   volatile unsigned char * p;
   volatile unsigned char toggle_bit, save_bit;
   unsigned char data;
   int i;
   int loop, null;

   p = (volatile unsigned char * ) 0x30000000;

   save_bit = *p;
   save_bit = save_bit & 0x40;
   //putnum (save_bit);
```

```
    for (;;)
    {
       toggle_bit = *p;
       //putnum (toggle_bit);

       toggle_bit = toggle_bit & 0x40;
       //putnum (toggle_bit);
       //print ("\r\n");

       if (save_bit == toggle_bit)
          return ;
       save_bit = toggle_bit;
    }

    return null;

}

// The Flash write routine generate the writing timing to the Flash
int FlashWrite(unsigned long address, unsigned char data)
{
    volatile unsigned char * p;
    int i;
    int loop, null;

    p = (volatile unsigned char * ) address;
    * p = data;

    null = 0;
    // Delay for some time before the next write
    for ( loop = 1; loop < 5; loop ++)
    {
       null = null + data;
    }

    return null;
}


// The Program routine program 1 byte to the specified address
void FlashProgram (unsigned long address, unsigned char data)
{

    FlashWrite(0x30005555, 0xAA);
    FlashWrite(0x30002AAA, 0x55);
    FlashWrite(0x30005555, 0xA0);
```

```
    FlashWrite(address, data);

    FlashPolling();
}

// Chip Erase 6 cycles
// 5555 AA 2AAA 55 5555 80 5555 AA 2AAA 55 5555 10

void FlashErase (void)
{
    FlashWrite(0x30005555, 0xAA);
    FlashWrite(0x30002AAA, 0x55);
    FlashWrite(0x30005555, 0x80);
    FlashWrite(0x30005555, 0xAA);
    FlashWrite(0x30002AAA, 0x55);
    FlashWrite(0x30005555, 0x10);

    FlashPolling();
}
```

4.2.6 audio.c

```
#include "xbasic_types.h"
#include "xio.h"

#include "xintc_l.h"
#include "xuartlite_l.h"
#include "xparameters.h"

#include "audio.h"


unsigned long InitializeAudio (unsigned long command )
{
    unsigned short * p;

    volatile unsigned long * preg;
    unsigned long status;


    preg = (volatile unsigned long * )XPAR_AUDIO_CONTROLLER_BASEADDR;
    * preg = command | 0x80000000;

    status = 0x80000000;

    while (status != 0)
    {
```

```
        status = (* preg) & 0x80000000;
    }

    status = *preg;

    return status;
}
```

4.2.7 ether.c

```
/*----------------------------
# CSEE 4840 Embedded System Design
#
# Most of the codes are taken from Internet Radio 2005
#
----------------------------*/

#include "ether.h"

#include "xparameters.h"
#include "xbasic_types.h"
#include "xio.h"
#include "xintc_l.h"

/* function prototypes, internal helpers */
static BYTE send(WORD *data, WORD addr, WORD dmalen);
static BYTE transmit(WORD sendlen);

/* note that at one point we had lots more functions.  the code
   indicates where they were. Many of them were flattened for code
   space reasons (the compiler had some inlining issues, possibly due
   to out smashing the stack). */

/* static data */

/* stored without the one's complement */
/* note: ip checksum can actually be constant!! */
/* all the pseudo headers and most of the headers on the udp_checksum
   can be calculated in advance.  we just need to ask ethereal what it
   should be :).  since time_stamp and sequence number are increments,
   we can just add (with carries) these increments to the new initial
   values each new packet.  so all we need is the update stuff in append. */
/* static unsigned long udp_checksum; */

/* the real note: udp checksum can actually be (and now is) 0.  This
   means that it is ignored.  We originally wrote code to calculate a
```

```
   proper udp checksum (it is still in the code, just commented out).
   we later commented it out and decided to use 0 due to code space
   limitations.  (this also allowed us to put back in diagnostics).

   note to future people: smashing your stack is a bad idea.
 */
static WORD cur_payload_addr;

struct {
  WORD padding;
  WORD sequence_number;
  unsigned long time_stamp;
} dynamic_data;

/* #define UDP_INIT_CHECKSUM (~((WORD)(0xaa78))); */
/* #define UDP_INIT_CHECKSUM (~((WORD)(0xb15d))); */

static BYTE header[HEADER_SIZE] = {
  /* create the static portion of the packets */
  /* MAC header (14 bytes) */
  /*   Destination MAC address (6 bytes): fixed */
  /* 0-5 * 0x00,0x0C, 0xF1, 0x73, 0x4C, 0x9C,
   */
   /* Laptop MAC */
     0x00,0x01,0x6C,0xC9,0xD7,0x2B,
  /*   Source MAC address (6 bytes): fixed */
  /* 6 */ 0x00,
  /* 7 */ 0x0D,
  /* 8 */ 0x60,
  /* 9 */ 0x7F,
  /* 10 */0xF9,
  /* 11 */0xAF,
  /*    Length (2 bytes): fixed */
  /* used as a type*/
  /* 12 */ 0x08,
  /* 13 */ 0x00,
  /*  IP header (20 bytes) */
  /*    Version (4 bits): fixed */
  /*    IHL (4 bits): fixed */
  /* 14 */ 0x45,
  /*    TOS (1 byte): fixed */
  /* 15 */ 0x00,
  /*    Total Length (2 bytes): fixed */
  /* 16 */ IP_LENGTH>>8,
  /* 17 */ IP_LENGTH & (0xff),
  /*    Identification (2 bytes): fixed */
```

```
/* 18 */ 0x00,
/* 19 */ 0x00,
/*    Flags (3 bits): fixed */
/*    Fragment Offset (13 bits): fixed */
/* 20 */ 0x00,
/* 21 */ 0x00,
/*    TTL (1 byte): fixed */
/* for now, this should die on contact*/
/* 22 */ 0x04,
/*    Protocol (1 byte): fixed */
/* UDP=17 */
/* 23 */ 0x11,
/*    Header Checksum (2 bytes): updated. see update_ip_checksum */
 // Micro4
   0x8A,0x63,
/*    Source IP (4 bytes): fixed */
/* 26 */ 0x80,
/* 27 */ 0x3B,
/* 28 */ 0x95,
/* 29 */ 0xA2,
/*    Destination IP (4 bytes): fixed */
// Micro4
   128,59,144,169,
///* 30 */ 128,
///* 31 */ 59,
///* 32 */ 144,
///* 33 */ 168,
/* UDP header (6 Bytes) */
/*   Source Port (2 bytes): fixed */
/* 34 */ 0x0B,
/* 35 */ 0xE2,
/*   Destination Port (2 bytes): fixed */
  /* in decimal: 3042 */
/* 36 */ 0x0B,
/* 37 */ 0xE2,
/*   Length (2 bytes): fixed */
/* 38 */ UDP_LENGTH>>8,
/* 39 */ UDP_LENGTH & 0xff,
/*   Checksum (2 bytes): updated */
/* 40 */ 0,
/* 41 */ 0,
/* RTP header (12 bytes) */
/*   Version (2 bits): fixed */
/* 10 */
/*   Padding (1 bit): fixed */
/* 0 */
```

```
  /*   Extension (1 bit): fixed */
  /* 0 */
  /*   CSRC count (4 bits): fixed */
  /* 0 */
  /* 42 */ 0x80,
  /*   Marker bit (1 bit): fixed */
  /* 0 */
  /*   Payload type (7 bits): fixed */
  /* L16=10 (2 channel( (see
     http://www.networksorcery.com/enp/protocol/rtp.htm) */
  /* 43 */ 0x0a,
  /*   Sequence number (16 bits): updated */
  /* 44 */ 0x00,
  /* 45 */ 0x00,
  /*   Time stamp (32 bits): updated */
  /* 46 */ 0x00,
  /* 47 */ 0x00,
  /* 48 */ 0x00,
  /* 49 */ 0x00,
  /*   SSRC (32 bits): fixed */
  /* should be a random value: I don't think this is what is meant :)
     -- since we don't change our source transport address and don't
     handle multiple synchronization source within the same RTP
     channel, this should not be a problem. */
  /* 50 */ 0x42,
  /* 51 */ 0x42,
  /* 52 */ 0x42,
  /* 53 */ 0x42
  /*   CSRC list (0 bits): fixed */
  /* we are evil and don't give proper attribution to sources. */
};


/* a simple wait function that burns clock cycles. */
void wait(int mult){
  volatile int j=0, i=1000000;
  for(; i > 0; --i) {
    for(; mult > 0; --mult) {
      // a smart compiler with aggresive inliner should unroll this
      ++j;
    }
  }
}

/*
   INITIALIZATION
```

```
*/

/* Diagnostics based on page 31 of the Ethernet Controller manual
   Write on two pages first and then read to avoid being fooled by
   data latched in both write and read.
*/

int diagnostics() {
  //  outnic(CMDR, 0x21);              // stop AX88796
  wait(10);

  outnic(CMDR,0x61);
  outnic(PSTART, 0x4E);

  outnic(CMDR,0x21);
  outnic(PSTOP, 0x3E);

  outnic(CMDR,0x61);
  if(innic(PSTART) != 0x4e)
    return 1;

  outnic(CMDR,0x21);               // switch to page 0
  if(innic(PSTOP) != 0x3e)
    return 2;

  if(innic(0x16*2) != 0x15)
    return 3;

  if(innic(0x12*2) != 0x0c)
    return 4;

  if(innic(0x13*2) != 0x12)
    return 5;

  return 0;
}

/* initializes the ethernet card and our data structures. sets up the
   static part of the packet header in memory. */
BYTE init() {
  int ret;

  outnic(NE_RESET, innic(NE_RESET));  // trigger a reset
  wait(2);
  if ((innic(ISR) & 0x80) == 0)       // Report if failed
    {
```

```
    //print("  Ethernet card failed to reset!\r\n");
    //print("Ethernet NIC not present or not initializing correctly\r\n");
    return 1;
   }
 else
   {
    //dprint("Ethernet card reset successful!\r\n");
    /*      ether_reset(); */ /* Reset Ethernet card, */

    /* Write 21h to Command Register to abort current DMA operations. */
    outnic(CMDR, 0x21);
    /* Wait 2 milliseconds (timout for inter-frame gap timer) */
    wait(10);
    /* Write 01h to Data Control Register to enable 16-bit word transfers. */
    outnic(DCR, 0x01);
    /* Write 00h to both Remote Byte Count Registers to zero out DMA counter. */
    outnic(RBCR0, 0x00);
    outnic(RBCR1, 0x00);
    /* Write 00h to Interrupt Mask Register to mask interrupts. */
    outnic(IMR, 0x00);
    /* Write ffh to Interrupt Status Register to clear interrupt flags */
    outnic(ISR, 0xff);
    /* Write 20h to Receive Configuration Register to put NIC in monitor mode */
    outnic(RCR, 0x20);
    /* Write 02h to Transmit Configuration Register to put NIC in loop-back mode */
    outnic(TCR, 0x02);
    /* Set RX Start and Stop, Boundary, and TX start page. */
    outnic(PSTART, RXSTART);
    outnic(PSTOP, RXSTOP);
    outnic(BNRY, (BYTE)(RXSTOP-1));
    outnic(TPSR, TXSTART);
    /* Reset interrupt mask and flags. */
    outnic(ISR, 0xFF);   // clear interrupt status register
    outnic(IMR, 0x00);   // Mask completion irq
    /* Write 22h to Command Register to start NIC */
    outnic(CMDR, 0x22);
    /* Write 00h to Transmit Configuration Register to set normal transmit operation */
    outnic(TCR, 0x00);

    //dprint("Ethernet card intialization complete!\r\n");
   }

/* init_packet_setup(); */
dynamic_data.sequence_number = 0;
dynamic_data.time_stamp = 0;
```

```
/*   udp_checksum = UDP_INIT_CHECKSUM; */
  cur_payload_addr = PAYLOAD_START_ADDRESS;


  /* now lets initialize the buffers. */
  return send((WORD *)header, PACKET_START_ADDRESS, 54);//HEADER_SIZE);
}


/* actually transmits a (filled in) packet. */
static BYTE transmit(WORD sendlen)
{
  int j;
  outnic(TPSR, TXSTART); // set Transmit Page Start Register
  outnic(TBCR0, (sendlen&0xff)); // set Transmit Byte Count
  outnic(TBCR1, (sendlen>>8));

  outnic(CMDR, CR_COMPLETE|CR_TXP); // start transmission
  j=1000;
  while(j-->0 && !(innic(ISR) & ISR_PTX));
  if(!j){
    return 1;
  }

  outnic(ISR,0xFF);
  return 0;
}


/* low level function that does dma to send data to card. */
/* based on code from Josh's group. */
static BYTE send(WORD *data, WORD addr, WORD dmalen){
  int i, j, h;
  WORD counter;
  WORD word;

  counter = dmalen>>1;

  outnic(RSAR0, (addr&0xff)); // set DMA starting address
  outnic(RSAR1, (addr>>8));

  outnic(ISR, 0xFF); // clear ISR

  outnic(RBCR0, (dmalen&0xff)); // set Remote DMA Byte Count
  outnic(RBCR1, (dmalen>>8));
```

```
  outnic(CMDR, CR_WRITE|CR_START); // start the DMA write - 0x12

  // change order of MS/LS since DMA
  // writes LS byte in 15-8, and MS byte in 7-0
  for(i=0; i<counter; i++){
    word = (data[i]<<8)|(data[i]>>8);
    outnic(DATAPORT, word);
  }

  if(!(innic(ISR) & ISR_RDC)){
    //    print("Data - DMA did not finish\r\n");
    return 1;
  }

  outnic(CMDR,CR_COMPLETE);

  return 0;
}

/* external interface */
/* takes a 2 word sample, and sends it out.  sends packet if needed */
/* 0 return is success */
BYTE output_sample(WORD *sample) {
  BYTE ret = 0;
/*   WORD check; */

  /*   ret = append(sample); */
   ret = send(sample, cur_payload_addr, 4);

   /* if(ret)
      return ret; */
  cur_payload_addr+=4;
  /* update checksums */

/*   udp_checksum += *(sample+1); */
/*   udp_checksum += *(sample); */
/*   while (udp_checksum>>16) */
/*     udp_checksum = (udp_checksum & 0xffff) + (udp_checksum >> 16); */

  if(cur_payload_addr >= PACKET_END_ADDRESS) {
    /* finalize_packet() */
    /* take the ones complement, as well as shoving it into a word */
/*     while (udp_checksum>>16) */
/*       udp_checksum = (udp_checksum & 0xffff) + (udp_checksum >> 16); */

/*     check = ~((WORD)(udp_checksum&0xffff)); */
```

```
/*     check = 0; */
/*     ret = send(&check, UDP_CHECKSUM_ADDRESS, 2); */
/*     if(ret) { */
/*       print("Error writing udp checksum\r\n"); */
/*       return ret; */
/*     } */
    ret = send(((WORD *)&dynamic_data)+1, RTP_SEQNUM_ADDRESS, 6);

    if(ret) {
      //print("Error writing rtp header data\r\n");
    }

    if(ret) {
      //print("Error finalizing packet. dropping.\r\n");
    } else {
      ret = transmit(PACKET_SIZE);
      //   print("probable success transmitting packet. happy! happy!\r\n");
    }

    /*     next_packet_setup(); */
    ++dynamic_data.sequence_number;
    dynamic_data.time_stamp += 20;

/*     udp_checksum = dynamic_data.sequence_number + (unsigned
short)((dynamic_data.time_stamp) >> 16) + (unsigned
short)((dynamic_data.time_stamp)&0xffff); */
/*     udp_checksum += UDP_INIT_CHECKSUM; */
/*     while (udp_checksum>>16) */
/*       udp_checksum = (udp_checksum & 0xffff) + (udp_checksum >> 16); */
    cur_payload_addr=PAYLOAD_START_ADDRESS;
  }

  return ret;
}
```

4.2.8 initialization.c

```
#include "xbasic_types.h"
#include "xio.h"

#include "xintc_l.h"
#include "xuartlite_l.h"
#include "xparameters.h"

#include "flash.h"
#include "sdram.h"
#include "audio.h"
```

```
#include "menu.h"

void main_init()
{
  int ret;
    ////////////////////////////////////////////////////////////////////////
  // The Initialization and Self test
  ////////////////////////////////////////////////////////////////////////

  //print("\r\n");
  //print(" Running Ethernet diagnostics\r\n");

  if (ret=diagnostics()) {
    //print(" Diagnostics failed. internal error number ");

    //print("\r\n");

  } else {
    //print(" Diagnostics successful.\r\n");
  }

  if(init()) {
    //print(" Ethernet NIC not present or not initializing correctly\r\n");
    //return 1;
  }

  //print(" Ethernet initialization done!\r\n");


  //print ( " Initialize Audio now......\r\n");
  //print ( " Attention: Audio initialization NOT done yet.\r\n");

  //print ( "\r\n");

  ////////////////////////////////////////////////////////////////////////
  // To be done
  // Example of Audio initialize subroutine calling
  //InitializeAudio(0x8000E008);   // Select LineIN
  ////////////////////////////////////////////////////////////////////////


  //print ( "\r\n");
  //print("Reading audio register 0");
  //print ( "\r\n");
  //print("The data is:  ");
  //putnum(InitializeAudio(0x8000C000));
```

```
  //print ( "\r\n");

  //print("\r\n\r\n Entering main loop now...\r\n");
}
```

4.2.9 int.c

```
#include "xparameters.h"
#include "xbasic_types.h"
#include "xio.h"
#include "xintc_l.h"
#include "xuartlite_l.h"

extern volatile short uart_interrupt_count;
extern char uart_character;
extern volatile unsigned short new_data;

void audio_sampler_handler(void *callback)
{
  // microblaze_disable_interrupts();
  new_data=1;
  //microblaze_enable_interrupts();
}

/*
 * Interrupt service routine for the UART
 */
void uart_handler(void *callback)
{
  Xuint32 IsrStatus;

  Xuint8 incoming_character;

  /* Check the ISR status register so we can identify the interrupt source */
  IsrStatus = XIo_In32(XPAR_MYUART_BASEADDR +
XUL_STATUS_REG_OFFSET);

  if ((IsrStatus & (XUL_SR_RX_FIFO_FULL |
XUL_SR_RX_FIFO_VALID_DATA)) != 0) {
    /* The input FIFO contains data: read it */
    incoming_character =
      (Xuint8) XIo_In32( XPAR_MYUART_BASEADDR + XUL_RX_FIFO_OFFSET );

    uart_character = incoming_character;
    uart_interrupt_count=1;
  }
```

```
 if ((IsrStatus & XUL_SR_TX_FIFO_EMPTY) != 0) {
  /* The output FIFO is empty: we can send another character */
 }

}
```

4.2.10 menu.c

```
#include "xbasic_types.h"
#include "xio.h"
#include "xintc_l.h"
#include "xuartlite_l.h"
#include "xparameters.h"
#include "flash.h"
#include "menu.h"

void FlashOut();
void FlashRecord();
volatile int j=1;
volatile int k=5;
volatile int flag=0;
extern volatile int out_enable;

// Write a text string on the display
void put_string(char *string, int row, int column)
{
 char *p = &(CHAR(row, column));
 while (*p++ = *string++)
  ;
}

void putMenu()
{
 char space_array[26]="                         ";
 volatile unsigned char *p;
 for ( p = &(CHAR(0,0)) ; p < &(CHAR(29,80)) ; ++p)              //clear VGA screen
   *p = ' ';
 put_string("INTERNET RADIO", 3, 30);                            //display menu items
 put_string("1. Live Broadcast            ", 5, 27);
 put_string("2. Advertisement Insertion    ", 7,27 );
 put_string("3. Utility                   ", 9, 27);
 put_string(space_array, 11, 27);
 put_string(space_array, 13, 27);
 put_string(space_array, 15, 27);
 put_string("  ", k, 24);
 put_string(">>", 5, 24);
 j=1;   //j save the menu item number
```

```
  k=5;   //k save the row position for ">>"
}

void vga_menu(char buf)
{

  char space_array[26]="                        ";
  volatile unsigned char *p;

  if (buf==0xd)                              //Enter key
       {

            switch (j)
              {
              case 1:
                put_string("Enable live broadcast    ", 5, 27);
                put_string("Disable live broadcast   ", 7, 27);
                put_string("Return                   ", 9, 27);
                put_string("  ", k, 24);
                k=5;
                put_string(">>", 5, 24);
                j=4;
                break;

              case 2:
                put_string("Start Ad insertion       ", 5, 27);
                put_string("Return                   ", 7, 27);
                put_string(space_array, 9, 27);
                put_string("  ", k, 24);
                k=5;
                put_string(">>", 5, 24);
                j=7;
                break;

              case 3:
                put_string("Erase Flash buffer       ", 5, 27);
                put_string("Record Ad to Flash       ", 7, 27);
                put_string("Return                   ", 9, 27);
                put_string("  ", k, 24);
                put_string(">>", 5, 24);
                k=5;
                j=9;
                break;

              case 4:
                out_enable=1;
```

```
            break;

          case 5:
            out_enable=0;
            break;

          case 6:
            putMenu();
            break;

          case 7:
            for ( p = &(CHAR(25,20)) ; p < &(CHAR(29,80)) ; ++p) *p = ' ';
            put_string("Playing advertisement now.....", 25, 25);
            FlashOut();
            put_string("Ad insertion completed", 27, 25);
            break;

          case 8:
            putMenu();
            break;

          case 9:
            for ( p = &(CHAR(25,20)) ; p < &(CHAR(29,80)) ; ++p) *p = ' ';
         put_string("Start erasing flash.....", 25, 25);
            FlashErase();
            put_string("Flash is empty now", 27, 25);
            break;

          case 10:                                                    //record sound
to flash

            for ( p = &(CHAR(25,20)) ; p < &(CHAR(29,80)) ; ++p) *p = ' ';
            put_string("Erase the entire Flash chip now.....", 25, 20);
            FlashErase();
            put_string("Record now.....", 26, 20);
            FlashRecord();
            put_string("Record to Flash successfully", 27, 20);
            break;

          case 11:
            putMenu();
            break;
          }

      }
                else  if (buf==0x1b || flag==1)       //Up or Down arrow key
```

```
	{
	   flag=1;
			if(buf==0x41)
				{
					if(j!=1 && j!=4 && j!=7 && j!=9) { j--;  put_string("  ", k,
24);  k=k-2; put_string(">>", k, 24);}
					flag=0;
				}
			if(buf==0x42)
				{
					if(j!=3 && j!=6 && j!=8 && j!=11) { j++; put_string("  ",
k, 24); k=k+2; put_string(">>", k, 24);}
					flag=0;
				}
	}

 }
```

4.2.11 main.c

```
#include "xbasic_types.h"
#include "xio.h"

#include "xintc_l.h"
#include "xuartlite_l.h"
#include "xparameters.h"

#include "ether.h"
#include "flash.h"
#include "audio.h"
#include "menu.h"


void main_init();                                   //function declaration
void audio_sampler_handler(void *callback);
void uart_handler(void *callback);

volatile short uart_interrupt_count=0;
char uart_character;
unsigned long uart_out;
volatile unsigned short new_data;
volatile int out_enable=0;

void FlashRecord()                                  //record audio to flash
{
  int i = 0;
  unsigned long sample_count = 0;
```

```
unsigned long max_count = 128000;
unsigned char byte;

while (sample_count < max_count)
{
   /* new_data is set in the interrupt handler when new data is ready. */
   if(new_data == 1) {

         new_data = 0;

         /* grab the data from the audio controller*/
         uart_out = XIo_In32(XPAR_AUDIO_SAMPLER_BASEADDR);
         /* the standard audio format is little endian, so we swap the bytes:
         we have 4 bytes, they get reordered from 1234 to 2143. */
         uart_out = ((uart_out >> 24 & 0xff) << 16) |
         ((uart_out >> 16 & 0xff) << 24) |
         (uart_out >> 8 & 0xff) |
         ((uart_out & 0xff) << 8);


     sample_count++;
     // Take 10 sample data from Audio, and save in SRAM
       // Save lower 16 bit
       //sram_buffer[2*sample_count] = sampler_out;
       // Save upper 16 bit
       //sram_buffer[2*sample_count+1] = sampler_out>>16;

  // write the 0th 8 bit
  //byte = flash_buffer[i*4+1];
  //word = word << 8 | byte;
  byte = uart_out & 0xff;
  FlashProgram(FLASH_BASE_ADDR+(sample_count<<2)+1, byte);

  // write the 1th 8 bit
  //byte = flash_buffer[i*4];
  //word = word << 8 | byte;
  byte = uart_out>>8;
  FlashProgram(FLASH_BASE_ADDR+(sample_count<<2), byte);

  // write the 2th 8 bit
  //byte = flash_buffer[i*4+3];
  //word = word << 8 | byte;
  byte = uart_out>>16;
  FlashProgram(FLASH_BASE_ADDR+(sample_count<<2)+3, byte);
```

```
        // write the 3th 8 bit
    //byte = flash_buffer[i*4+2];
    //word = byte;
    byte = uart_out>>24;
    FlashProgram(FLASH_BASE_ADDR+(sample_count<<2)+2, byte);
    }
  }
}

void FlashOut()                                    //playing audio saved in flash before
{
  int i;
  unsigned long max_count = 128000;
  volatile unsigned char *flash_buffer;
  unsigned char byte;
  unsigned long word;

  flash_buffer = (volatile unsigned char *) FLASH_BASE_ADDR;

  for (i = 0; i < max_count ; )
  {
  ////////////////////////////////////////
  // 32bit: 3/2/1/0
  ////////////////////////////////////////
    // read the 3th(highest) 8 bit

    if(new_data == 1) {
    i ++;
    new_data = 0;
    byte = flash_buffer[(i<<2)+2];
    word = byte;

    // read the 2th 8 bit
    byte = flash_buffer[(i<<2)+3];
    word = word << 8 | byte;

    // read the 1th 8 bit
    byte = flash_buffer[i<<2];
    word = word << 8 | byte;

    // read the 0th 8 bit
    byte = flash_buffer[(i<<2)+1];
    word = word << 8 | byte;
    output_sample((WORD *)(&word));
    }
  }
```

```
}


int main()
{
  uart_interrupt_count = 0;
  new_data = 0;

  ////////////////////////////////////////////////////////////////////////
  // Setup Interrupt
  ////////////////////////////////////////////////////////////////////////
  /* Enable UART and Audio interrupts and register the ISR */
  XIntc_RegisterHandler( XPAR_INTC_BASEADDR,
XPAR_INTC_MYUART_INTERRUPT_INTR,
                         (XInterruptHandler)uart_handler, (void *)0);
  XIntc_RegisterHandler(XPAR_INTC_SINGLE_BASEADDR,
XPAR_INTC_AUDIO_SAMPLER_INTERRUPT_INTR,
                         audio_sampler_handler, (void*)0);

  XIntc_mEnableIntr( XPAR_INTC_BASEADDR,
XPAR_MYUART_INTERRUPT_MASK |
XPAR_AUDIO_SAMPLER_INTERRUPT_MASK);
  //XIntc_mEnableIntr(XPAR_INTC_SINGLE_BASEADDR,
XPAR_AUDIO_SAMPLER_INTERRUPT_MASK);

  XIntc_mMasterEnable( XPAR_INTC_BASEADDR );
  XIntc_Out32(XPAR_INTC_BASEADDR + XIN_MER_OFFSET,
XIN_INT_MASTER_ENABLE_MASK);
  microblaze_enable_interrupts();
  XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);

  main_init();   //initialization
  putMenu();    //display menu on VGA

  while (1)
  {
     if ( uart_interrupt_count  != 0) {

           vga_menu(uart_character);              //call the vga_menu if there is input from
uart
        uart_interrupt_count = 0;

     }

     /* new_data is set in the interrupt handler when new data is ready. */
```

```
    if(new_data == 1) {

            new_data = 0;

            /* grab the data from the audio controller*/
            uart_out = XIo_In32(XPAR_AUDIO_SAMPLER_BASEADDR);
            /* the standard audio format is little endian, so we swap the bytes:
            we have 4 bytes, they get reordered from 1234 to 2143. */
            uart_out = ((uart_out >> 24 & 0xff) << 16) |
            ((uart_out >> 16 & 0xff) << 24) |
            (uart_out >> 8 & 0xff) |
            ((uart_out & 0xff) << 8);
            if(out_enable==0) uart_out=0x00000000;
            /* output the new sample to the ethernet card */
            output_sample((WORD *)(&uart_out));
   }
  }

  return 0;
}
```

4.3 VHDL Code

4.3.1 opb_sram.vhd

```
------------------------------------------------------------------------------
--
-- OPB SRAM controller
--
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity opb_sram is

  generic (
    C_OPB_AWIDTH : integer              := 32;
    C_OPB_DWIDTH : integer              := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"20000000";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"2FFFFFFF");

  port (
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
```

```
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;        -- Sequential Address
    Sln_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_errAck  : out std_logic;        -- (unused)
    Sln_retry   : out std_logic;        -- (unused)
    Sln_toutSup : out std_logic;        -- Timeout suppress
    Sln_xferAck : out std_logic;        -- Transfer acknowledge

    PB_DIN   : in std_logic_vector(0 to 15);      -- SRAM data bus
    PB_DOUT  : out std_logic_vector(0 to 15);     -- SRAM data bus output
    PB_DOUT_EN : out std_logic;                   -- SRAM data bus output enable
    PB_A   : out std_logic_vector (0 to 19);      -- SRAM address bus
    RAM_CE : out std_logic;                       -- SRAM chip select
    PB_OE_N : out std_logic;                      -- SRAM output enable
    PB_WE_N : out std_logic;                      -- SRAM write enable
    PB_UB_N : out std_logic;                      -- SRAM byte enable
    PB_LB_N : out std_logic                       -- SRAM byte enable
    );

end opb_sram;

architecture Behavioral of opb_sram is

  constant RAM_AWIDTH : integer := 18;  -- Number of address lines on the RAM
  constant RAM_DWIDTH : integer := 16;  -- Number of data lines on the RAM

  signal RNW : std_logic;
  signal RAM_DI: std_logic_vector(0 to 31);
  signal ABus : std_logic_vector(0 to RAM_AWIDTH-1);
  signal chip_select : std_logic;
  signal output_enable : std_logic;
  signal WE : std_logic;
  signal BE : std_logic_vector(0 to 3);
  signal A31: std_logic;
  signal A30R: std_logic;
  signal RAM_RD_DATA : std_logic_vector (0 to 31);
  signal RAM_RD_DATA_MUX : std_logic_vector (0 to 31);
  signal RAM_WR_DATA : std_logic_vector (0 to 31);

  signal CE : std_logic;          -- SRAM chip enalbe
  signal OE : std_logic;          -- SRAM output enalbe
  signal xfer_start: std_logic;    -- Transfer start
  signal xfer_end: std_logic;      -- Transfer end
  signal xfer_idle: std_logic;     -- No transfer pending
  signal obuf_oe: std_logic;       -- Data output buffer enable
```

```
  signal xfer_done: std_logic;     -- Transfer has been done
  signal byte_start: std_logic;
  signal word_address: std_logic;
  signal byte_mode: std_logic;
  signal halfword_mode: std_logic;
  signal word_mode: std_logic;
  signal A30 : std_logic;
  signal word_access: std_logic;
  signal word_latch: std_logic;
  signal set_a30: std_logic;
  signal DIN_Latch: std_logic_vector(0 to 15);

  -- Critical: Sln_xferAck is generated directly from state bit 0!
  constant STATE_BITS : integer := 3;
  constant ST_IDLE      : std_logic_vector(0 to STATE_BITS-1) := "000";
  constant ST_SELECTED  : std_logic_vector(0 to STATE_BITS-1) := "001";
  constant ST_WAIT      : std_logic_vector(0 to STATE_BITS-1) := "011";
  constant ST_XFER      : std_logic_vector(0 to STATE_BITS-1) := "111";
  constant ST_WORDINIT  : std_logic_vector(0 to STATE_BITS-1) := "010";
  constant ST_HOLD      : std_logic_vector(0 to STATE_BITS-1) := "100";
  constant ST_WAITWORD  : std_logic_vector(0 to STATE_BITS-1) := "101";
  constant ST_XFERWORD  : std_logic_vector(0 to STATE_BITS-1) := "110";

  signal present_state, next_state : std_logic_vector(0 to STATE_BITS-1);

begin

  register_opb_inputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      RAM_DI <= (others => '0');
      ABus <= (others => '0');
      RNW <= '0';
      A31 <= '0';
      A30R <= '0';
      BE <= (others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
      RAM_DI <= OPB_DBus(0 to C_OPB_DWIDTH-1);
      if ( word_address = '1') then
        ABus <= OPB_ABus(C_OPB_AWIDTH-2-(RAM_AWIDTH-1) to
C_OPB_AWIDTH-3) & '1';
      else
        ABus <= OPB_ABus(C_OPB_AWIDTH-2-(RAM_AWIDTH-1) to
C_OPB_AWIDTH-2);
      end if ;
      A31 <= OPB_ABus(31);
```

```
      A30R <= OPB_ABus(30);
      if (xfer_idle = '1') then
        BE <= OPB_BE(0 to 3);
        RNW <= OPB_RNW;
      end if;
    end if;
  end process register_opb_inputs;

--  register_opb_outputs: process (OPB_Clk, OPB_Rst)
--  begin
--    if OPB_Rst = '1' then
--      Sln_DBus <= (others => '0');
--    elsif OPB_Clk'event and OPB_Clk = '1' then
----     if (output_enable = '1' and RNW = '1')  then
--     if (output_enable = '1')  then
--       Sln_DBus <= RAM_RD_DATA_MUX;
--     else
--       Sln_DBus <= (others => '0');
--     end if;
--    end if;
--  end process register_opb_outputs;

  process (RAM_RD_DATA_MUX, output_enable)
  begin
    if (output_enable = '1')  then
      Sln_DBus <= RAM_RD_DATA_MUX;
    else
      Sln_DBus <= (others => '0');
    end if;
  end process;

  -- Unused outputs
  Sln_errAck  <= '0';
  Sln_retry   <= '0';
  Sln_toutSup <= '0';
  --Sln_DBus(RAM_DWIDTH to C_OPB_DWIDTH-1) <= (others => '0');

  chip_select <=
    '1' when OPB_select = '1' and
      OPB_ABus(0 to C_OPB_AWIDTH-2-RAM_AWIDTH) =
      C_BASEADDR(0 to C_OPB_AWIDTH-2-RAM_AWIDTH) else
    '0';

  Sln_xferAck <= xfer_done;

  -- Sequential part of the FSM
```

```
fsm_seq : process(OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    present_state <= ST_IDLE;
  elsif OPB_Clk'event and OPB_Clk = '1' then
    present_state <= next_state;
  end if;
end process fsm_seq;

-- Combinational part of the FSM
fsm_comb : process(present_state, chip_select, OPB_Select, word_mode)
begin
    case present_state is
      when ST_IDLE =>
        if chip_select = '1' then
          next_state <= ST_SELECTED;
        else
          next_state <= ST_IDLE;
        end if;

      when ST_SELECTED =>
        if OPB_Select = '1' then
           next_state <= ST_XFER;
        else
           next_state <= ST_IDLE;
        end if;

      -- State encoding is critical here: xfer must only be true here
      when ST_XFER =>
        if word_mode = '1' then
           next_state <= ST_HOLD;
        else
            next_state <= ST_IDLE;
        end if;

      when ST_HOLD =>
        if OPB_Select = '1' then
           next_state <= ST_XFERWORD;
        else
          next_state <= ST_IDLE;
        end if;

      when ST_XFERWORD =>
          next_state <= ST_IDLE;
      when others =>
        next_state <= ST_IDLE;
```

```
    end case;

end process fsm_comb;

fsm_output: process (present_state, word_mode)
begin
   output_enable <= '0';
   xfer_start <= '0';
   byte_start <= '0';
   xfer_end <= '0';
   xfer_idle <= '0';
   xfer_done <= '0';
   word_latch <= '0';
   word_address <= '0';
   set_a30 <= '0';
   case present_state is
      when ST_IDLE =>
         xfer_idle <= '1';
         byte_start <= '1';
      when ST_SELECTED =>
         xfer_start <= '1';
      when ST_XFER =>
         output_enable <= not word_mode;
         word_latch <= '1';
         xfer_end <= not word_mode;
         xfer_done <= not word_mode;
         set_a30 <= word_mode;
         word_address <= word_mode;
      when ST_HOLD =>
         xfer_start <= '1';
         word_address <= '1';
      when ST_XFERWORD =>
         output_enable <= '1';
         xfer_end <= '1';
         xfer_done <= '1';

      when others =>
         output_enable <= '0';
         byte_start <= '0';
         xfer_start <= '0';
         xfer_end <= '0';
         xfer_idle <= '0';
         set_a30 <= '0';
         word_latch <= '0';
         word_address <= '0';
   end case;
```

```
end process fsm_output;

-- The following process generates the SRAM OE output signal
sram_oe_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    OE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( next_state = ST_SELECTED) then
      OE <= not OPB_RNW;
    elsif ( next_state = ST_IDLE) then
      OE <= '1';
    end if;
  end if;
end process sram_oe_reg;

-- The following process generates the SRAM chip enable output signal
sram_ce_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    CE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( next_state = ST_SELECTED) then
      CE <= '0';
    elsif ( next_state = ST_IDLE) then
      CE <= '1';
    end if;
  end if;
end process sram_ce_reg;

-- The following process generates the SRAM WE output signal
sram_we_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    WE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '0' then
    if ( xfer_start = '1' and chip_select = '1' ) then
      WE <= RNW;
    else
      WE <= '1';
    end if;
  end if;
end process sram_we_reg;

-- Detect the current access mode
process (BE)
```

```
begin
  byte_mode <= '0';
  halfword_mode <= '1';
  word_mode <= '0';
  case BE is
    when "0001" =>
      byte_mode <= '1';
      halfword_mode <= '0';
      word_mode <= '0';
    when "0010" =>
      byte_mode <= '1';
      halfword_mode <= '0';
      word_mode <= '0';
    when "0100" =>
      byte_mode <= '1';
      halfword_mode <= '0';
      word_mode <= '0';
    when "1000" =>
      byte_mode <= '1';
      halfword_mode <= '0';
      word_mode <= '0';
    when "0011" =>
      byte_mode <= '0';
      halfword_mode <= '1';
      word_mode <= '0';
    when "1100" =>
      byte_mode <= '0';
      halfword_mode <= '1';
      word_mode <= '0';
    when "1111" =>
      byte_mode <= '0';
      halfword_mode <= '0';
      word_mode <= '1';
    when others =>
      byte_mode <= '0';
      halfword_mode <= '1';
      word_mode <= '0';
 end case;
end process;

process ( OPB_Clk, OPB_Rst)
begin
  if (OPB_Rst = '1') then
    DIN_Latch <= (others => '0');
  elsif ( OPB_Clk 'event and OPB_Clk = '1') then
    if ( word_latch = '1') then
```

```
        DIN_Latch <= PB_DIN;
      end if;
    end if;
  end process;

  process (OPB_Clk, OPB_Rst)
  begin
    if (OPB_Rst = '1') then
      word_access <= '0';
    elsif (OPB_Clk 'event and OPB_Clk = '1') then
      if (byte_start = '1' or xfer_idle = '1' or xfer_end = '1' ) then
          word_access <= '0';
      elsif ( set_a30 = '1' and word_mode = '1') then
          word_access <= '1';
      end if;
    end if;
  end process;

  --

  --RAM_WR_DATA <= RAM_DI;
  process ( RAM_DI, word_access, word_mode, A30R)
  begin
    if ( word_mode = '0') then
      if ( A30R = '1') then
        RAM_WR_DATA(16 to 31) <= RAM_DI(16 to 31);
      else
        RAM_WR_DATA(16 to 31) <= RAM_DI(0 to 15);
      end if;
    else
      if (word_access = '1') then
        RAM_WR_DATA(16 to 31) <= RAM_DI(16 to 31);
      else
        RAM_WR_DATA(16 to 31) <= RAM_DI(0 to 15);
      end if;
    end if;
  end process;

 --RAM_RD_DATA <= DIN_Latch & PB_DIN;

  process ( PB_DIN, DIN_Latch, A31, byte_mode)
  begin
    if ( A31 = '0' and byte_mode = '1') then
      RAM_RD_DATA(24 to 31) <= PB_DIN(0 to 7);
    else
      RAM_RD_DATA(24 to 31) <= PB_DIN(8 to 15);
```

```
    end if;

    RAM_RD_DATA(16 to 23) <= PB_DIN(0 to 7) ;
    RAM_RD_DATA(0 to 15) <= DIN_Latch ;
  end process;

  process ( RAM_RD_DATA, byte_mode, halfword_mode)
  begin
    if ( byte_mode = '1') then
      RAM_RD_DATA_MUX <= RAM_RD_DATA(24 to 31) & RAM_RD_DATA(24
to 31) & RAM_RD_DATA(24 to 31) & RAM_RD_DATA(24 to 31);
    elsif (halfword_mode = '1' ) then
      RAM_RD_DATA_MUX <= RAM_RD_DATA(16 to 31) & RAM_RD_DATA(16
to 31);
    else
      RAM_RD_DATA_MUX <=  RAM_RD_DATA;
    end if;
  end process;


  ----

  A30 <= ABus(RAM_AWIDTH-1) or word_access;

  process (A31, CE, byte_mode, RNW)
  begin
    if ( RNW = '0') then
      if ( byte_mode = '1') then
        if ( A31 = '1') then
          PB_UB_N <= '1';
          PB_LB_N <= CE;
        else
          PB_UB_N <= CE;
          PB_LB_N <= '1';
        end if;
      else
        PB_UB_N <= CE;
        PB_LB_N <= CE;
      end if;
   else
     PB_UB_N <= CE;
     PB_LB_N <= CE;
   end if;
  end process;

  -- The following assign the SRAM output signals
```

```
  PB_OE_N <= OE;
  PB_WE_N <= WE;
  RAM_CE <= CE;
  --PB_UB_N <= CE;
  --PB_LB_N <= CE;
  PB_A <= "00" & ABus(0 to RAM_AWIDTH-1);  -- & A30;
  PB_DOUT <= RAM_WR_DATA(16 to 31);
  PB_DOUT_EN <=  (not CE) and OE;


end Behavioral;
```

4.3.2 opb_flash.vhd

```
------------------------------------------------------------------------------
--
-- OPB Flash controller
--
--
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity opb_flash is

  generic (
    C_OPB_AWIDTH : integer                  := 32;
    C_OPB_DWIDTH : integer                  := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"30000000";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"3FFFFFFF");

  port (
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;        -- Sequential Address
    Sln_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_errAck  : out std_logic;        -- (unused)
    Sln_retry   : out std_logic;        -- (unused)
    Sln_toutSup : out std_logic;        -- Timeout suppress
```

```
    Sln_xferAck : out std_logic;        -- Transfer acknowledge

    PB_DIN   : in std_logic_vector(0 to 15);      -- FLASH data bus
    PB_DOUT  : out std_logic_vector(0 to 15);     -- FLASH data bus output
    PB_DOUT_EN : out std_logic;                   -- FLASH data bus output enable
    PB_A   : out std_logic_vector (0 to 19);      -- FLASH address bus
    PB_OE_N  : out std_logic;                     -- FLASH output enable
    PB_WE_N  : out std_logic;                     -- FLASH write enable
    FLASH_CE_N  : out std_logic                       -- FLASH byte enable
    );

end opb_flash;

architecture Behavioral of opb_flash is

  constant FLASH_AWIDTH : integer := 19;  -- Number of address lines on the FLASH
  constant FLASH_DWIDTH : integer := 8;  -- Number of data lines on the FLASH

  signal RNW : std_logic;
  signal FLASH_DI: std_logic_vector(0 to 31);
  signal ABus : std_logic_vector(0 to FLASH_AWIDTH-1);
  signal chip_select : std_logic;
  signal output_enable : std_logic;
  signal WE : std_logic;
  signal BE : std_logic_vector(0 to 3);
  signal A30A31: std_logic_vector (0 to 1);
  signal FLASH_RD_DATA : std_logic_vector (0 to 31);
  signal FLASH_WR_DATA : std_logic_vector (0 to 31);
  signal TOSUP : std_logic;

  signal CE : std_logic;          -- FLASH chip enalbe
  signal OE : std_logic;          -- FLASH output enalbe
  signal xfer_start: std_logic;    -- Transfer start
  signal xfer_end: std_logic;      -- Transfer end
  signal xfer_idle: std_logic;     -- No transfer pending
  signal obuf_oe: std_logic;       -- Data output buffer enable
  signal xfer_done: std_logic;     -- Transfer has been done
  signal wait_expire: std_logic;
  signal wait_counter: std_logic_vector ( 0 to 3) ;

  -- Critical: Sln_xferAck is generated directly from state bit 0!
  constant STATE_BITS : integer := 3;
  constant ST_IDLE      : std_logic_vector(0 to STATE_BITS-1) := "000";
  constant ST_SELECTED  : std_logic_vector(0 to STATE_BITS-1) := "001";
  constant ST_WAIT      : std_logic_vector(0 to STATE_BITS-1) := "011";
  constant ST_XFER      : std_logic_vector(0 to STATE_BITS-1) := "111";
```

```
  constant ST_HOLD      : std_logic_vector(0 to STATE_BITS-1) := "100";
  constant ST_WAITWORD  : std_logic_vector(0 to STATE_BITS-1) := "101";
  constant ST_XFERWORD  : std_logic_vector(0 to STATE_BITS-1) := "110";

  signal present_state, next_state : std_logic_vector(0 to STATE_BITS-1);

begin

  register_opb_inputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      FLASH_DI <= (others => '0');
      ABus <= (others => '0');
      RNW <= '0';
      A30A31 <= (others => '0');
      BE <= (others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
      FLASH_DI <= OPB_DBus(0 to C_OPB_DWIDTH-1);
      ABus <= OPB_ABus(C_OPB_AWIDTH-1-(FLASH_AWIDTH-1) to
C_OPB_AWIDTH-1);
      RNW <= OPB_RNW;
      A30A31 <= OPB_ABus(30 to 31);
      BE <= OPB_BE(0 to 3);
    end if;
  end process register_opb_inputs;

  register_opb_outputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      Sln_DBus <= (others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
--      if (output_enable = '1' and RNW = '1')  then
      if (output_enable = '1' )  then
        Sln_DBus <= FLASH_RD_DATA;
      else
        Sln_DBus <= (others => '0');
      end if;
    end if;
  end process register_opb_outputs;

  -- Unused outputs
  Sln_errAck  <= '0';
  Sln_retry   <= '0';
  Sln_toutSup <= TOSUP;
  --Sln_DBus(FLASH_DWIDTH to C_OPB_DWIDTH-1) <= (others => '0');
```

```
chip_select <=
  '1' when OPB_select = '1' and
      OPB_ABus(0 to C_OPB_AWIDTH-2-FLASH_AWIDTH) =
      C_BASEADDR(0 to C_OPB_AWIDTH-2-FLASH_AWIDTH) else
  '0';

Sln_xferAck <= xfer_end;

-- Sequential part of the FSM
fsm_seq : process(OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    present_state <= ST_IDLE;
  elsif OPB_Clk'event and OPB_Clk = '1' then
    present_state <= next_state;
  end if;
end process fsm_seq;

-- Combinational part of the FSM
fsm_comb : process(present_state, chip_select, OPB_Select , wait_expire)
begin
    case present_state is
      when ST_IDLE =>
        if chip_select = '1' then
          next_state <= ST_SELECTED;
        else
          next_state <= ST_IDLE;
        end if;

      when ST_SELECTED =>
        if OPB_Select = '1' then
           next_state <= ST_WAIT;
        else
          next_state <= ST_IDLE;
        end if;

      when ST_WAIT =>
        if OPB_Select = '1' then
          if ( wait_expire = '1' ) then
            next_state <= ST_XFER;
          else
            next_state <= ST_WAIT;
          end if;
        else
          next_state <= ST_IDLE;
        end if;
```

```
      -- State encoding is critical here: xfer must only be true here
      when ST_XFER =>
        next_state <= ST_IDLE;

      when others =>
        next_state <= ST_IDLE;
    end case;

end process fsm_comb;

fsm_output: process (present_state)
begin
  output_enable <= '0';
  xfer_start <= '0';
  xfer_end <= '0';
  xfer_idle <= '0';
  case present_state is
    when ST_IDLE =>
      xfer_idle <= '1';
    when ST_SELECTED =>
      xfer_start <= '1';
    when ST_WAIT =>
      output_enable <= '1';
    when ST_XFER =>
      xfer_end <= '1';
    when others =>
      output_enable <= '0';
      xfer_start <= '0';
      xfer_end <= '0';
      xfer_idle <= '0';
  end case;
end process fsm_output;

-- The following process generates the SFLASH OE output signal
flash_oe_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    OE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( chip_select = '1' and xfer_idle = '1') then
      OE <= not OPB_RNW;
    elsif ( xfer_end = '1') then
      OE <= '1';
    end if;
  end if;
```

```
end process flash_oe_reg;

-- The following process generates the flash chip enable output signal
flash_ce_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    CE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( chip_select = '1' and xfer_idle = '1') then
      CE <= '0';
    elsif ( xfer_end = '1') then
      CE <= '1';
    end if;
  end if;
end process flash_ce_reg;


process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    TOSUP <= '0';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( chip_select = '1' and xfer_idle = '1') then
      TOSUP <= '1';
    elsif ( xfer_end = '1') then
      TOSUP <= '0';
    end if;
  end if;
end process;

-- The following process generates the flash WE output signal
flash_we_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    WE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( xfer_start = '1') then
      WE <= RNW;
    elsif (wait_expire = '1') then
      WE <= '1';
    end if;
  end if;
end process flash_we_reg;

-- Wait counter
process (OPB_Clk, OPB_Rst)
```

```
  begin
    if ( OPB_Rst = '1') then
      wait_counter <= (others => '0');
    elsif OPB_Clk 'event and OPB_Clk = '1' then
      if ( xfer_start = '1' or xfer_idle = '1' ) then
        wait_counter <= "0000";
      else
        wait_counter <= wait_counter + '1';
      end if;
    end if;
  end process;

  process ( wait_counter, RNW)
  begin
   if (RNW = '1') then
     if ( wait_counter = "0101"   ) then
       wait_expire <= '1';
     else
       wait_expire <= '0';
     end if;
   else
     if ( wait_counter = "1011"   ) then
       wait_expire <= '1';
     else
       wait_expire <= '0';
     end if;
   end if;
  end process;

  --
  FLASH_WR_DATA <= FLASH_DI ;
  FLASH_RD_DATA <= PB_DIN(8 to 15) & PB_DIN(8 to 15) & PB_DIN(8 to 15) &
PB_DIN(8 to 15);

  -- The following assign the flash output signals
  PB_OE_N <=  OE;
  PB_WE_N <= WE;
  FLASH_CE_N <= CE;
  PB_A <= '0' & ABus;
  PB_DOUT <= "00000000" & FLASH_WR_DATA(24 to 31);
  --FLASH_DO <= FLASH_RD_DATA;
  PB_DOUT_EN <= not CE and OE;


end Behavioral;
```

4.3.3 opb_mux.vhd

```
------------------------------------------------------------------------------
--
-- OPB Bus Multiplexer
--
--
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity opb_pb_mux is

  generic (
    C_OPB_AWIDTH : integer                := 32;
    C_OPB_DWIDTH : integer                := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"FFEF0500";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"FFFF05FF");

  port (
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;        -- Sequential Address
    Sln_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_errAck  : out std_logic;        -- (unused)
    Sln_retry   : out std_logic;        -- (unused)
    Sln_toutSup : out std_logic;        -- Timeout suppress
    Sln_xferAck : out std_logic;        -- Transfer acknowledge

    PB_D   : inout std_logic_vector(0 to 15);      -- SRAM data bus
    PB_A   : out std_logic_vector (0 to 19);      -- SRAM address bus
    PB_OE_N : out std_logic;                    -- SRAM output enable
    PB_WE_N : out std_logic;                     -- SRAM write enable
    PB_UB_N : out std_logic;                    -- SRAM byte enable
    PB_LB_N : out std_logic;                    -- SRAM byte enable
    PB_DIN  : out std_logic_vector (0 to 15);     -- SRAM data bus


    PB_DOUT_SRAM   : in std_logic_vector(0 to 15);
    PB_DOUT_EN_SRAM   : in std_logic;
```

```
    PB_A_SRAM   : in std_logic_vector(0 to 19);
    PB_OE_N_SRAM   : in std_logic;
    PB_WE_N_SRAM   : in std_logic;
    PB_UB_N_SRAM   : in std_logic;
    PB_LB_N_SRAM   : in std_logic;

    PB_DOUT_FLASH   : in std_logic_vector(0 to 15);
    PB_DOUT_EN_FLASH   : in std_logic;
    PB_A_FLASH   : in std_logic_vector(0 to 19);
    PB_OE_N_FLASH   : in std_logic;
    PB_WE_N_FLASH   : in std_logic;

    PB_DOUT_SDRAM   : in std_logic_vector(0 to 15);
    PB_DOUT_EN_SDRAM   : in std_logic;
    PB_A_SDRAM   : in std_logic_vector(0 to 19);
    PB_UB_N_SDRAM   : in std_logic;
    PB_LB_N_SDRAM   : in std_logic;

    PB_DOUT_ETHER   : in std_logic_vector(0 to 15);
    PB_DOUT_EN_ETHER   : in std_logic;
    PB_A_ETHER   : in std_logic_vector(0 to 19);
    PB_OE_N_ETHER   : in std_logic;
    PB_WE_N_ETHER   : in std_logic;
    PB_UB_N_ETHER   : in std_logic;
    PB_LB_N_ETHER   : in std_logic;

    AU_BUSY   : in std_logic;
    AU_CCLK   : in std_logic;
    AU_CDTI   : in std_logic;
    --AU_CDTO   : out std_logic;

    SDRAM_LOCK : out std_logic;
    SDRAM_ACTIVE : in std_logic
    );

end opb_pb_mux;

architecture Behavioral of opb_pb_mux is

  component IOBUF_F_24
    port(
      O  : out   std_ulogic;
      IO : inout std_ulogic;
      I  : in    std_ulogic;
      T  : in    std_ulogic
      );
```

```
  end component;

  ----- component OBUF_F_24              -----
  component OBUF_F_24
    port(
      O : out std_ulogic;
      I : in  std_ulogic
      );
  end component;


  signal current_slave : std_logic_vector (0 to 2) ;
  signal Abus : std_logic_vector (0 to 8);
  signal PB_DOUT : std_logic_vector(0 to 15);
  signal PB_DOUT_EN : std_logic;

  signal SetLock : std_logic;
  signal Lock : std_logic;
  signal RNW : std_logic;
  signal chip_select : std_logic;
  signal output_enable : std_logic;

  signal xfer_start: std_logic;    -- Transfer start
  signal xfer_end: std_logic;      -- Transfer end
  signal xfer_idle: std_logic;     -- No transfer pending
  signal obuf_oe: std_logic;       -- Data output buffer enable
  signal xfer_done: std_logic;     -- Transfer has been done
  signal oe_audio: std_logic;

  -- Critical: Sln_xferAck is generated directly from state bit 0!
  constant STATE_BITS : integer := 3;
  constant ST_IDLE      : std_logic_vector(0 to STATE_BITS-1) := "000";
  constant ST_SELECTED  : std_logic_vector(0 to STATE_BITS-1) := "001";
  constant ST_WAIT      : std_logic_vector(0 to STATE_BITS-1) := "011";
  constant ST_XFER      : std_logic_vector(0 to STATE_BITS-1) := "111";

  signal present_state, next_state : std_logic_vector(0 to STATE_BITS-1);

begin

  register_opb_inputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      SetLock <=  '0';
      RNW <= '0';
    elsif OPB_Clk'event and OPB_Clk = '1' then
```

```
    SetLock <= OPB_DBus(C_OPB_DWIDTH-1);
    RNW <= OPB_RNW;
  end if;
end process register_opb_inputs;

register_opb_outputs: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    Sln_DBus <= (others => '0');
  elsif OPB_Clk'event and OPB_Clk = '1' then
   if (output_enable = '1' and RNW = '1')  then
      Sln_DBus <= "0000000000000000" & "00000000" & "000000" & Lock & '0';
   else
     Sln_DBus <= (others => '0');
   end if;
  end if;
end process register_opb_outputs;

-- Unused outputs
Sln_errAck  <= '0';
Sln_retry   <= '0';
Sln_toutSup <= '0';

chip_select <=
  '1' when OPB_select = '1' and
      OPB_ABus(0 to C_OPB_AWIDTH-1) = C_BASEADDR(0 to C_OPB_AWIDTH-1)
else
  '0';

Sln_xferAck <= present_state(0);

-- Sequential part of the FSM
fsm_seq : process(OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    present_state <= ST_IDLE;
  elsif OPB_Clk'event and OPB_Clk = '1' then
    present_state <= next_state;
  end if;
end process fsm_seq;

-- Combinational part of the FSM
fsm_comb : process(present_state, chip_select, OPB_Select)
begin
   case present_state is
     when ST_IDLE =>
```

```
        if chip_select = '1' then
          next_state <= ST_SELECTED;
        else
          next_state <= ST_IDLE;
        end if;
      when ST_SELECTED =>
        if OPB_Select = '1' then
           next_state <= ST_WAIT;
        else
          next_state <= ST_IDLE;
        end if;
      when ST_WAIT =>
        if OPB_Select = '1' then
          next_state <= ST_XFER;
        else
          next_state <= ST_IDLE;
        end if;
      -- State encoding is critical here: xfer must only be true here
      when ST_XFER =>
        next_state <= ST_IDLE;
      when others =>
        next_state <= ST_IDLE;
    end case;

end process fsm_comb;

process (present_state)
begin
  if ( present_state = ST_WAIT) then
     output_enable <= '1';
  else
     output_enable <= '0';
  end if;
end process;

process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    Lock <= '0';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( present_state = ST_WAIT and SetLock = '1' ) then
      Lock <= '1';
    elsif (SDRAM_ACTIVE = '1') then
      Lock <= '0';
    end if;
  end if;
```

```
  end process;

  process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      SDRAM_LOCK <= '0';
    elsif OPB_Clk 'event and OPB_Clk = '1' then
      if ( Lock = '1' ) then
        SDRAM_LOCK <= '1';
      end if;
    end if;
  end process;


--  PB_DIN <= PB_D;

--  process (PB_DOUT, PB_DOUT_EN) begin
--    if ( PB_DOUT_EN = '1') then
--      PB_D <= PB_DOUT;
--    else
--      PB_D <= "ZZZZZZZZZZZZZZZZ";
--    end if;
--  end process;

  Abus <= OPB_ABus ( 0 to 8);

  process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      current_slave <= "000";
    elsif OPB_Clk 'event and OPB_Clk = '1' then
      if ( Lock = '1') then
         current_slave <= "010";
      elsif ( AU_BUSY = '1') then
         current_slave <= "100";
      elsif ( OPB_select = '1') then
        case Abus is
          when "001000000" =>  -- 20000000
             current_slave <= "000";
          when "001100000" =>  -- 30000000
             current_slave <= "001";
          when "010000000" =>  -- 40000000
             current_slave <= "010";
          when "000000001" =>  -- 00800000
             current_slave <= "011";
          when "111111111" =>  -- FFEF0000
```

```
            current_slave <= "100";
          when others =>
            current_slave <= "010";
        end case;
      end if;
    end if;
end process;

--AU_CDTO <= PB_D(13);
--AU_CDTO <= '0';

process (     PB_DOUT_SRAM,
              PB_DOUT_EN_SRAM,
              PB_A_SRAM,
              PB_OE_N_SRAM,
              PB_WE_N_SRAM,
              PB_UB_N_SRAM,
              PB_LB_N_SRAM,

              PB_DOUT_FLASH,
              PB_DOUT_EN_FLASH,
              PB_A_FLASH,
              PB_OE_N_FLASH,
              PB_WE_N_FLASH,

              PB_DOUT_SDRAM,
              PB_DOUT_EN_SDRAM,
              PB_A_SDRAM,
              PB_UB_N_SDRAM,
              PB_LB_N_SDRAM,

              PB_DOUT_ETHER,
              PB_DOUT_EN_ETHER,
              PB_A_ETHER,
              PB_OE_N_ETHER,
              PB_WE_N_ETHER,
              PB_UB_N_ETHER,
              PB_LB_N_ETHER,

              AU_CCLK,
              AU_CDTI   )
begin
  case current_slave is
    when "000" =>
      PB_A      <= PB_A_SRAM;
      PB_OE_N   <= PB_OE_N_SRAM;
```

```
  PB_WE_N    <= PB_WE_N_SRAM;
  PB_UB_N    <= PB_UB_N_SRAM;
  PB_LB_N    <= PB_LB_N_SRAM;
  PB_DOUT    <= PB_DOUT_SRAM;
  PB_DOUT_EN <= PB_DOUT_EN_SRAM;
when "001" =>
  PB_A       <= PB_A_FLASH;
  PB_OE_N    <= PB_OE_N_FLASH;
  PB_WE_N    <= PB_WE_N_FLASH;
  PB_UB_N    <= '1';
  PB_LB_N    <= '1';
  PB_DOUT    <= PB_DOUT_FLASH;
  PB_DOUT_EN <= PB_DOUT_EN_FLASH;
when "010" =>
  PB_A       <= PB_A_SDRAM;
  PB_OE_N    <= '1';
  PB_WE_N    <= '1';
  PB_UB_N    <= PB_UB_N_SDRAM;
  PB_LB_N    <= PB_LB_N_SDRAM;
  PB_DOUT    <= PB_DOUT_SDRAM;
  PB_DOUT_EN <= PB_DOUT_EN_SDRAM;
when "011" =>
  PB_A       <= PB_A_ETHER;
  PB_OE_N    <= PB_OE_N_ETHER;
  PB_WE_N    <= PB_WE_N_ETHER;
  PB_UB_N    <= PB_UB_N_ETHER;
  PB_LB_N    <= PB_LB_N_ETHER;
  PB_DOUT    <= PB_DOUT_ETHER;
  PB_DOUT_EN <= PB_DOUT_EN_ETHER;

when "100" =>
  PB_A       <= (others => '0') ;
  PB_OE_N    <= '1';
  PB_WE_N    <= '1';
  PB_UB_N    <= '1';
  PB_LB_N    <= '1';
  PB_DOUT    <= "00000000000000" & AU_CDTI & AU_CCLK;
  PB_DOUT_EN <= '1';

when others =>
  PB_A       <= PB_A_SDRAM;
  PB_OE_N    <= '1';
  PB_WE_N    <= '1';
  PB_UB_N    <= PB_UB_N_SDRAM;
  PB_LB_N    <= PB_LB_N_SDRAM;
  PB_DOUT    <= PB_DOUT_SDRAM;
```

```
      PB_DOUT_EN <= PB_DOUT_EN_SDRAM;
  end case;
end process;


obuf_oe <= not PB_DOUT_EN;

-- Audio CDTO must be input mode in case of audio controller transfer
process ( PB_DOUT_EN , AU_BUSY)
begin
  if ( AU_BUSY = '1') then
    oe_audio <= '1';
  else
    oe_audio <= not PB_DOUT_EN;
  end if;
end process;

-- The following assign the data pads
data0_pad: IOBUF_F_24 port map (
O => PB_DIN(0),
IO => PB_D(0),
I => PB_DOUT(0),
T => obuf_oe
);

data1_pad: IOBUF_F_24 port map (
O => PB_DIN(1),
IO => PB_D(1),
I => PB_DOUT(1),
T => obuf_oe
);

data2_pad: IOBUF_F_24 port map (
O => PB_DIN(2),
IO => PB_D(2),
I => PB_DOUT(2),
T => obuf_oe
);

data3_pad: IOBUF_F_24 port map (
O => PB_DIN(3),
IO => PB_D(3),
I => PB_DOUT(3),
T => obuf_oe
);
```

```
data4_pad: IOBUF_F_24 port map (
O => PB_DIN(4),
IO => PB_D(4),
I => PB_DOUT(4),
T => obuf_oe
);

data5_pad: IOBUF_F_24 port map (
O => PB_DIN(5),
IO => PB_D(5),
I => PB_DOUT(5),
T => obuf_oe
);

data6_pad: IOBUF_F_24 port map (
O => PB_DIN(6),
IO => PB_D(6),
I => PB_DOUT(6),
T => obuf_oe
);

data7_pad: IOBUF_F_24 port map (
O => PB_DIN(7),
IO => PB_D(7),
I => PB_DOUT(7),
T => obuf_oe
);

data8_pad: IOBUF_F_24 port map (
O => PB_DIN(8),
IO => PB_D(8),
I => PB_DOUT(8),
T => obuf_oe
);

data9_pad: IOBUF_F_24 port map (
O => PB_DIN(9),
IO => PB_D(9),
I => PB_DOUT(9),
T => obuf_oe
);

data10_pad: IOBUF_F_24 port map (
O => PB_DIN(10),
IO => PB_D(10),
I => PB_DOUT(10),
```

```
 T => obuf_oe
 );

 data11_pad: IOBUF_F_24 port map (
 O => PB_DIN(11),
 IO => PB_D(11),
 I => PB_DOUT(11),
 T => obuf_oe
 );

 data12_pad: IOBUF_F_24 port map (
 O => PB_DIN(12),
 IO => PB_D(12),
 I => PB_DOUT(12),
 T => obuf_oe
 );

 data13_pad: IOBUF_F_24 port map (
 O => PB_DIN(13),
 IO => PB_D(13),
 I => PB_DOUT(13),
 T => oe_audio
 );

 data14_pad: IOBUF_F_24 port map (
 O => PB_DIN(14),
 IO => PB_D(14),
 I => PB_DOUT(14),
 T => obuf_oe
 );

 data15_pad: IOBUF_F_24 port map (
 O => PB_DIN(15),
 IO => PB_D(15),
 I => PB_DOUT(15),
 T => obuf_oe
 );

end Behavioral;
```

4.3.4 opb_sdram.vhd

```
--------------------------------------------------------------------------------
--
-- OPB SDRAM controller
--
--
```

```
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
--library IEEE, UNISIM;
--use IEEE.std_logic_1164.all;
--use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
--use WORK.common.all;

entity opb_sdram is

  generic (
    C_OPB_AWIDTH : integer                := 32;
    C_OPB_DWIDTH : integer                := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"40000000";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"4FFFFFFF");

  port (
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;        -- Sequential Address
    Sln_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_errAck  : out std_logic;        -- (unused)
    Sln_retry   : out std_logic;        -- (unused)
    Sln_toutSup : out std_logic;        -- Timeout suppress
    Sln_xferAck : out std_logic;        -- Transfer acknowledge

    SDRAM_CLK : in std_logic;
    SDRAM_LOCK : in std_logic;
    SDRAM_ACTIVE : out std_logic;
    PB_DIN   : in std_logic_vector(0 to 15);      -- SDRAM data bus
    PB_DOUT  : out std_logic_vector(0 to 15);     -- SDRAM data bus output
    PB_DOUT_EN : out std_logic;                   -- SDRAM data bus output enable
    PB_A   : out std_logic_vector (0 to 19);      -- SDRAM address bus
    PB_UB_N : out std_logic;                      -- SDRAM DQM
    PB_LB_N : out std_logic;                      -- SDRAM DQM
    SDRAM_CKE : out std_logic;                    -- SDRAM Clock enable
    SDRAM_CAS_N : out std_logic;                  -- SDRAM CAS
    SDRAM_RAS_N : out std_logic;                  -- SDRAM RAS
```

```
   SDRAM_CE_N : out std_logic;                -- SDRAM chip enable
   SDRAM_WE_N : out std_logic                 -- SDRAM write enable
   );

end opb_sdram;

architecture Behavioral of opb_sdram is

  component IBUFG
    port(
      O : out std_ulogic;

      I : in  std_ulogic
      );
  end component;

  constant SDRAM_AWIDTH : integer := 23;  -- Number of address lines on the RAM
  constant SDRAM_DWIDTH : integer := 16;  -- Number of data lines on the RAM
  constant FREQ    :     natural := 67_000;  -- operating frequency in KHz
  constant IN_PHASE  :  boolean := true;  -- SDRAM and controller work on same or
opposite clock edge
  constant PIPE_EN   :  boolean := false;  -- if true, enable pipelined read operations
  constant MAX_NOP   :  natural := 10000;  -- number of NOPs before entering self-
refresh
  constant MULTIPLE_ACTIVE_ROWS :     boolean := false;  -- if true, allow an active
row in each bank
  constant DATA_WIDTH :     natural := 16;  -- host & SDRAM data width
  constant NROWS :     natural := 4096;  -- number of rows in SDRAM array
  constant NCOLS :     natural := 512;  -- number of columns in SDRAM array
  constant HADDR_WIDTH  :     natural := 23;  -- host-side address width
  constant SADDR_WIDTH  :     natural := 12;  -- SDRAM-side address width

  signal RNW : std_logic;
  signal SDRAM_DI: std_logic_vector(0 to 31);
  signal ABus : std_logic_vector(0 to SDRAM_AWIDTH-1);
  signal chip_select : std_logic;
  signal output_enable : std_logic;
  signal WE : std_logic;
  signal BE : std_logic_vector(0 to 3);
  signal A30A31: std_logic_vector (0 to 1);
  signal SDRAM_RD_DATA : std_logic_vector (0 to 31);
  signal SDRAM_WR_DATA : std_logic_vector (0 to 31);

  signal CE : std_logic;          -- SDRAM chip enalbe
  signal OE : std_logic;          -- SDRAM output enalbe
  signal SDRAM_RD: std_logic;
```

```
  signal SDRAM_WR: std_logic;
  signal RD_RST: std_logic;
  signal WR_RST: std_logic;
  signal SDRAM_DATA_EN: std_logic;
  signal done_lat: std_logic;
  signal xfer_start: std_logic;    -- Transfer start
  signal xfer_end: std_logic;      -- Transfer end
  signal xfer_idle: std_logic;     -- No transfer pending
  signal obuf_oe: std_logic;       -- Data output buffer enable
  signal xfer_done: std_logic;     -- Transfer has been done
  signal xfer_done0: std_logic;
  signal xfer_done1: std_logic;
  signal TOSUP: std_logic;
  signal timeout_cnt: std_logic_vector ( 0 to 5) ;
  signal timeout : std_logic;


  -- Critical: Sln_xferAck is generated directly from state bit 0!
  constant STATE_BITS : integer := 3;
  constant ST_IDLE      : std_logic_vector(0 to STATE_BITS-1) := "000";
  constant ST_SELECTED  : std_logic_vector(0 to STATE_BITS-1) := "001";
  constant ST_WAIT      : std_logic_vector(0 to STATE_BITS-1) := "011";
  constant ST_XFER      : std_logic_vector(0 to STATE_BITS-1) := "111";
  constant ST_HOLD      : std_logic_vector(0 to STATE_BITS-1) := "100";
  constant ST_WAITWORD  : std_logic_vector(0 to STATE_BITS-1) := "101";
  constant ST_XFERWORD  : std_logic_vector(0 to STATE_BITS-1) := "110";

  signal present_state, next_state : std_logic_vector(0 to STATE_BITS-1);

  -- SDRAM controller signals
  signal clk          : std_logic;  -- master clock
  signal lock         : std_logic;  -- true if clock is stable
  signal rst          : std_logic;  -- reset
  signal rd           : std_logic;  -- initiate read operation
  signal wr           : std_logic;  -- initiate write operation
  signal earlyOpBegun : std_logic;  -- read/write/self-refresh op has begun (async)
  signal opBegun      : std_logic;  -- read/write/self-refresh op has begun (clocked)
  signal rdPending    : std_logic;  -- true if read operation(s) are still in the pipeline
  signal done         : std_logic;  -- read or write operation is done
  signal rdDone       : std_logic;  -- read operation is done and data is available
  signal hAddr        : std_logic_vector(HADDR_WIDTH-1 downto 0);  -- address from
host to SDRAM
  signal hDIn         : std_logic_vector(DATA_WIDTH-1 downto 0);  -- data from host
to SDRAM
  signal hDOut        : std_logic_vector(DATA_WIDTH-1 downto 0);  -- data from
SDRAM to host
```

```
  signal status       : std_logic_vector(3 downto 0);  -- diagnostic status of the FSM

    -- SDRAM side
  signal cke     : std_logic;           -- clock-enable to SDRAM
  signal ce_n    : std_logic;           -- chip-select to SDRAM
  signal ras_n   : std_logic;           -- SDRAM row address strobe
  signal cas_n   : std_logic;           -- SDRAM column address strobe
  signal we_n    : std_logic;           -- SDRAM write enable
  signal ba      : std_logic_vector(1 downto 0);  -- SDRAM bank address
  signal sAddr   : std_logic_vector(SADDR_WIDTH-1 downto 0);  -- SDRAM
row/column address
  signal sDIn    : std_logic_vector(DATA_WIDTH-1 downto 0);  -- data from SDRAM
  signal sDOut   : std_logic_vector(DATA_WIDTH-1 downto 0);  -- data to SDRAM
  signal sDOutEn : std_logic;           -- true if data is output to SDRAM on sDOut
  signal dqmh    : std_logic;           -- enable upper-byte of SDRAM databus if true
  signal dqml    : std_logic;           -- enable lower-byte of SDRAM databus if true

  ---------------------------------------------------------------------------
  -- The following signals are copied from the XESS design
  ---------------------------------------------------------------------------

  constant OUTPUT : std_logic := '1';   -- direction of dataflow w.r.t. this controller
  constant INPUT  : std_logic := '0';
  constant NOP    : std_logic := '0';   -- no operation
  constant READ   : std_logic := '1';   -- read operation
  constant WRITE  : std_logic := '1';   -- write operation

  -- SDRAM timing parameters
  constant Tinit : natural := 200;      -- min initialization interval (us)
  constant Tras  : natural := 45;       -- min interval between active to precharge commands
(ns)
  constant Trcd  : natural := 20;       -- min interval between active and R/W commands
(ns)
  constant Tref  : natural := 64_000_000;  -- maximum refresh interval (ns)
  constant Trfc  : natural := 66;       -- duration of refresh operation (ns)
  constant Trp   : natural := 20;       -- min precharge command duration (ns)
  constant Twr   : natural := 15;       -- write recovery time (ns)
  constant Txsr  : natural := 75;       -- exit self-refresh time (ns)

  -- SDRAM timing parameters converted into clock cycles (based on FREQ)
  constant NORM        : natural := 1_000_000;  -- normalize ns * KHz
  constant INIT_CYCLES : natural := 1+((Tinit*FREQ)/1000);  -- SDRAM power-on
initialization interval
  constant RAS_CYCLES  : natural := 1+((Tras*FREQ)/NORM);  -- active-to-precharge
interval
```

```
  constant RCD_CYCLES  : natural := 1+((Trcd*FREQ)/NORM);  -- active-to-R/W
interval
  constant REF_CYCLES  : natural := 1+(((Tref/NROWS)*FREQ)/NORM);  -- interval
between row refreshes
  constant RFC_CYCLES  : natural := 1+((Trfc*FREQ)/NORM);  -- refresh operation
interval
  constant RP_CYCLES   : natural := 1+((Trp*FREQ)/NORM);  -- precharge operation
interval
  constant WR_CYCLES   : natural := 1+((Twr*FREQ)/NORM);  -- write recovery time
  constant XSR_CYCLES  : natural := 1+((Txsr*FREQ)/NORM);  -- exit self-refresh
time
  constant MODE_CYCLES : natural := 2;  -- mode register setup time
  constant CAS_CYCLES  : natural := 3;  -- CAS latency
  constant RFSH_OPS    : natural := 8;  -- number of refresh operations needed to init
SDRAM

  -- timer registers that count down times for various SDRAM operations
  signal timer_r, timer_x       : natural range 0 to INIT_CYCLES;  -- current SDRAM op
time
  signal rasTimer_r, rasTimer_x : natural range 0 to RAS_CYCLES;  -- active-to-
precharge time
  signal wrTimer_r, wrTimer_x   : natural range 0 to WR_CYCLES;  -- write-to-
precharge time
  signal refTimer_r, refTimer_x : natural range 0 to REF_CYCLES;  -- time between row
refreshes
  signal rfshCntr_r, rfshCntr_x : natural range 0 to NROWS;  -- counts refreshes that are
neede
  signal nopCntr_r, nopCntr_x   : natural range 0 to MAX_NOP;  -- counts consecutive
NOP operations

  signal doSelfRfsh : std_logic;        -- active when the NOP counter hits zero and self-
refresh can start

  -- states of the SDRAM controller state machine
  type cntlState is (
    INITWAIT,                    -- initialization - waiting for power-on initialization to
complete
    INITPCHG,                    -- initialization - initial precharge of SDRAM banks
    INITSETMODE,                 -- initialization - set SDRAM mode
    INITRFSH,                    -- initialization - do initial refreshes
    RW,                          -- read/write/refresh the SDRAM
    ACTIVATE,                    -- open a row of the SDRAM for reading/writing
    REFRESHROW,                  -- refresh a row of the SDRAM
    SELFREFRESH                  -- keep SDRAM in self-refresh mode with CKE low
    );
  signal state_r, state_x : cntlState;  -- state register and next state
```

```
  -- commands that are sent to the SDRAM to make it perform certain operations
  -- commands use these SDRAM input pins (ce_n,ras_n,cas_n,we_n,dqmh,dqml)
  subtype sdramCmd is std_logic_vector(5 downto 0);
  constant NOP_CMD    : sdramCmd := "011100";
  constant ACTIVE_CMD : sdramCmd := "001100";
  constant READ_CMD   : sdramCmd := "010100";
  constant WRITE_CMD  : sdramCmd := "010000";
  constant PCHG_CMD   : sdramCmd := "001011";
  constant MODE_CMD   : sdramCmd := "000011";
  constant RFSH_CMD   : sdramCmd := "000111";

  -- SDRAM mode register
  -- the SDRAM is placed in a non-burst mode (burst length = 1) with a 3-cycle CAS
  subtype sdramMode is std_logic_vector(11 downto 0);
  constant MODE : sdramMode := "00" & "0" & "00" & "011" & "0" & "000";

  -- the host address is decomposed into these sets of SDRAM address components
  --constant ROW_LEN : natural := log2(NROWS);  -- number of row address bits
  --constant COL_LEN : natural := log2(NCOLS);  -- number of column address bits
  constant ROW_LEN : natural := 12;  -- number of row address bits
  constant COL_LEN : natural := 9;  -- number of column address bits

  signal   bank    : std_logic_vector(ba'range);  -- bank address bits
  signal   row     : std_logic_vector(ROW_LEN - 1 downto 0);  -- row address within bank
  signal   col     : std_logic_vector(sAddr'range);  -- column address within row

  -- registers that store the currently active row in each bank of the SDRAM
  --constant NUM_ACTIVE_ROWS           : integer :=
int_select(MULTIPLE_ACTIVE_ROWS = false, 1, 2**ba'length);
  constant NUM_ACTIVE_ROWS           : integer := 1;
  type activeRowType is array(0 to NUM_ACTIVE_ROWS-1) of
std_logic_vector(row'range);
  signal   activeRow_r, activeRow_x   : activeRowType;
  signal   activeFlag_r, activeFlag_x : std_logic_vector(0 to NUM_ACTIVE_ROWS-1);
-- indicates that some row in a bank is active
  signal   bankIndex                  : natural range 0 to NUM_ACTIVE_ROWS-1;  -- bank
address bits
  signal   activeBank_r, activeBank_x : std_logic_vector(ba'range);  -- indicates the bank
with the active row
  signal   doActivate                 : std_logic;  -- indicates when a new row in a bank needs
to be activated

  -- there is a command bit embedded within the SDRAM column address
  constant CMDBIT_POS    : natural   := 10;  -- position of command bit
```

```
  constant AUTO_PCHG_ON  : std_logic := '1';  -- CMDBIT value to auto-precharge the
bank
  constant AUTO_PCHG_OFF : std_logic := '0';  -- CMDBIT value to disable auto-
precharge
  constant ONE_BANK      : std_logic := '0';  -- CMDBIT value to select one bank
  constant ALL_BANKS     : std_logic := '1';  -- CMDBIT value to select all banks

  -- status signals that indicate when certain operations are in progress
  signal wrInProgress       : std_logic;  -- write operation in progress
  signal rdInProgress       : std_logic;  -- read operation in progress
  signal activateInProgress : std_logic;  -- row activation is in progress

  -- these registers track the progress of read and write operations
  signal rdPipeline_r, rdPipeline_x : std_logic_vector(CAS_CYCLES+1 downto 0);  --
pipeline of read ops in progress
  signal wrPipeline_r, wrPipeline_x : std_logic_vector(0 downto 0);  -- pipeline of write
ops (only need 1 cycle)

  -- registered outputs to host
  signal opBegun_r, opBegun_x            : std_logic;  -- true when SDRAM read or write
operation is started
  signal hDOut_r, hDOut_x                : std_logic_vector(hDOut'range);  -- holds data read
from SDRAM and sent to the host
  signal hDOutOppPhase_r, hDOutOppPhase_x : std_logic_vector(hDOut'range);  --
holds data read from SDRAM   on opposite clock edge

  -- registered outputs to SDRAM
  signal cke_r, cke_x          : std_logic;  -- clock enable
  signal cmd_r, cmd_x          : sdramCmd;  -- SDRAM command bits
  signal ba_r, ba_x            : std_logic_vector(ba'range);  -- SDRAM bank address bits
  signal sAddr_r, sAddr_x      : std_logic_vector(sAddr'range);  -- SDRAM row/column
address
  signal sData_r, sData_x      : std_logic_vector(sDOut'range);  -- SDRAM out databus
  signal sDataDir_r, sDataDir_x : std_logic;  -- SDRAM databus direction control bit

begin

  register_opb_inputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      SDRAM_DI <= (others => '0');
      ABus <= (others => '0');
      RNW <= '0';
      A30A31 <= (others => '0');
      BE <= (others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
```

```
    SDRAM_DI <= OPB_DBus(0 to C_OPB_DWIDTH-1);
    ABus <= OPB_ABus(C_OPB_AWIDTH-2-(SDRAM_AWIDTH-1) to
C_OPB_AWIDTH-2);
    RNW <= OPB_RNW;
    A30A31 <= OPB_ABus(30 to 31);
    BE <= OPB_BE(0 to 3);
  end if;
 end process register_opb_inputs;

 process (output_enable, SDRAM_RD_DATA)
 begin
   if (output_enable = '1')  then
     Sln_DBus <= SDRAM_RD_DATA;
   else
     Sln_DBus <= (others => '0');
   end if;
 end process;

 -- Unused outputs
 Sln_errAck  <= '0';
 Sln_retry   <= '0';
 Sln_toutSup <= TOSUP;
 --Sln_DBus(SDRAM_DWIDTH to C_OPB_DWIDTH-1) <= (others => '0');

 chip_select <=
  '1' when OPB_select = '1' and
     OPB_ABus(0 to C_OPB_AWIDTH-2-SDRAM_AWIDTH) =
     C_BASEADDR(0 to C_OPB_AWIDTH-2-SDRAM_AWIDTH) else
  '0';

 Sln_xferAck <= xfer_end;

 -- Sequential part of the FSM
 fsm_seq : process(OPB_Clk, OPB_Rst)
 begin
  if OPB_Rst = '1' then
    present_state <= ST_IDLE;
  elsif OPB_Clk'event and OPB_Clk = '1' then
    present_state <= next_state;
  end if;
 end process fsm_seq;

 -- Combinational part of the FSM
 fsm_comb : process(present_state, chip_select, OPB_Select, xfer_done)
 begin
    case present_state is
```

```
      when ST_IDLE =>
        if chip_select = '1' then
          next_state <= ST_SELECTED;
        else
          next_state <= ST_IDLE;
        end if;

      when ST_SELECTED =>
        if OPB_Select = '1' then
           next_state <= ST_WAIT;
        else
          next_state <= ST_IDLE;
        end if;

      when ST_WAIT =>
        if OPB_Select = '1' then
          if ( xfer_done = '1' ) then
            next_state <= ST_XFER;
          else
            next_state <= ST_WAIT;
          end if;
        else
          next_state <= ST_IDLE;
        end if;

      -- State encoding is critical here: xfer must only be true here
      when ST_XFER =>
        next_state <= ST_IDLE;

      when others =>
        next_state <= ST_IDLE;
    end case;

end process fsm_comb;

fsm_output: process (present_state)
begin
  output_enable <= '0';
  xfer_start <= '0';
  xfer_end <= '0';
  xfer_idle <= '0';
  case present_state is
    when ST_IDLE =>
      xfer_idle <= '1';
    when ST_SELECTED =>
      xfer_start <= '1';
```

```
      when ST_WAIT =>
        --output_enable <= '1';
      when ST_XFER =>
        output_enable <= '1';
        xfer_end <= '1';
      when others =>
        output_enable <= '0';
        xfer_start <= '0';
        xfer_end <= '0';
        xfer_idle <= '0';
  end case;
end process fsm_output;

RD_RST <= OPB_Rst or done;

process (OPB_Clk, RD_RST)
begin
  if RD_RST = '1' then
    SDRAM_RD <= '0';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( chip_select = '1' and xfer_idle = '1') then
      SDRAM_RD <=  OPB_RNW;
    elsif ( xfer_end = '1') then
      SDRAM_RD <= '0';
    end if;
  end if;
end process;

WR_RST <= OPB_Rst or opBegun;

process (OPB_Clk, WR_RST)
begin
  if WR_RST = '1' then
    SDRAM_WR <= '0';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( chip_select = '1' and xfer_idle = '1') then
      SDRAM_WR <= not OPB_RNW;
    elsif ( xfer_end = '1') then
      SDRAM_WR <= '0';
    end if;
  end if;
end process;


process (OPB_Clk, OPB_Rst)
begin
```

```
  if OPB_Rst = '1' then
    TOSUP <= '0';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( chip_select = '1' and xfer_idle = '1') then
      TOSUP <= '1';
    elsif ( xfer_end = '1') then
      TOSUP <= '0';
    end if;
  end if;
end process;

process (OPB_Rst, OPB_Clk)
begin
  if ( OPB_Rst = '1') then
    xfer_done <= '0';
  elsif (OPB_Clk'event and OPB_Clk = '1') then
    if (timeout = '1') then
      xfer_done <= '1';
    elsif (xfer_start = '1') then
      xfer_done <= '0';
    else
      xfer_done <= xfer_done1;
    end if;
  end if;
end process;

process (done, OPB_Clk)
begin
  if ( done = '1') then
    xfer_done0 <= '1';
  elsif (OPB_Clk'event and OPB_Clk = '1') then
    if (xfer_start = '1' or xfer_idle = '1') then
      xfer_done0 <= '0';
    end if;
  end if;
end process;

process (OPB_Rst, OPB_Clk)
begin
  if ( OPB_Rst = '1') then
    xfer_done1 <= '0';
  elsif (OPB_Clk'event and OPB_Clk = '1') then
    if (xfer_start = '1'  or xfer_idle = '1') then
      xfer_done1 <= '0';
    else
      xfer_done1 <= xfer_done0;
```

```
    end if;
  end if;
end process;

process (timeout_cnt)
begin
        if ( timeout_cnt = "111111") then
                timeout <= '1';
        else
                timeout <= '0';
        end if;
end process;

process (OPB_Clk, OPB_Rst)
begin
        if ( OPB_Rst = '1') then
          timeout_cnt <= "000000";
        elsif ( OPB_Clk 'event and OPB_Clk = '1') then
                if ( xfer_start = '1' or xfer_idle = '1' ) then
                  timeout_cnt <= "000000";
                else
                  timeout_cnt <= timeout_cnt + '1';
                end if;
        end if;
end process;

process (state_r )
begin
  if (state_r =  RW ) then
    SDRAM_ACTIVE <= '1';
  else
    SDRAM_ACTIVE <= '0';
  end if;
end process;

-- The following assign the SDRAM output signals
--SDRAM_DO <= SDRAM_RD_DATA;

process( clk)
begin
  if ( clk'event and clk = '1') then
    if ( SDRAM_DATA_EN = '1') then
      SDRAM_RD_DATA <= hDOut& hDOut;
    end if;
  end if;
end process;
```

```
  process( clk)
  begin
    if ( clk'event and clk = '1') then
      done_lat <= done;
    end if;
  end process;

  SDRAM_DATA_EN <= not done_lat and done;

--  clk <= SDRAM_CLK;
 clkibuf : IBUFG port map (
  I => SDRAM_CLK,
  O => clk
  );

  lock <= SDRAM_LOCK;

  rst <= OPB_Rst;
  hAddr <= ABus;
  hDin <= SDRAM_DI(16 to 31);
  sDin <= PB_DIN;

  PB_DOUT <= sDOut;
  PB_DOUT_EN <= sDOutEn;
  PB_A <= "000000" & ba & sAddr;
  PB_UB_N <= dqmh;
  PB_LB_N <= dqml;
  SDRAM_CKE <= cke;
  SDRAM_CAS_N <= cas_n;
  SDRAM_RAS_N <= ras_n;
  SDRAM_CE_N <= ce_n;
  SDRAM_WE_N <= we_n;

  process(OPB_Rst, clk)
  begin
    if ( OPB_Rst = '1') then
      wr <= '0';
    elsif ( clk'event and clk = '1') then
      if ( opBegun = '1') then
        wr <= '0';
      elsif ( SDRAM_WR = '1') then
        wr <= '1';
      end if;
    end if;
```

```
  end process;

  process(RD_RST, clk)
  begin
    if ( RD_RST = '1') then
      rd <= '0';
    elsif ( clk'event and clk = '1') then
--       if ( OpBegun = '1') then
--         rd <= '0';
--       els
       if ( SDRAM_RD = '1') then
         rd <= '1';
       end if;
    end if;

  end process;

--
**********************************************************************
**
--
**********************************************************************
**
--**                                                **
--**  The following SDRAM controller is taken from XESS design          **
--**                                                **
--
**********************************************************************
**
--
**********************************************************************
**

-- Company : XESS Corp.
-- Engineer : Dave Vanden Bout
-- Creation Date : 05/17/2005
-- Copyright : 2005, XESS Corp
-- Tool Versions : WebPACK 6.3.03i
--
-- Description:
-- SDRAM controller
--
-- Revision:
-- 1.4.0
--
-- Additional Comments:
```

```
-- 1.4.0:
-- Added generic parameter to enable/disable independent active rows in each bank.
-- 1.3.0:
-- Modified to allow independently active rows in each bank.
-- 1.2.0:
-- Modified to allow pipelining of read/write operations.
-- 1.1.0:
-- Initial release.
--
-- License:
-- This code can be freely distributed and modified as long as
-- this header is not removed.
----------------------------------------------------------------------


  ------------------------------------------------------------
  -- attach some internal signals to the I/O ports
  ------------------------------------------------------------

  -- attach registered SDRAM control signals to SDRAM input pins
  (ce_n, ras_n, cas_n, we_n, dqmh, dqml) <= cmd_r;  -- SDRAM operation control bits
  cke                             <= cke_r;  -- SDRAM clock enable
  ba                              <= ba_r;  -- SDRAM bank address
  sAddr                             <= sAddr_r;  -- SDRAM address
  sDOut                              <= sData_r;  -- SDRAM output data bus
  sDOutEn                              <= '1' when sDataDir_r = OUTPUT else '0';  -- output
databus enable

  -- attach some port signals
  hDOut   <= hDOut_r;               -- data back to host
  opBegun <= opBegun_r;               -- true if requested operation has begun


  ------------------------------------------------------------
  -- compute the next state and outputs
  ------------------------------------------------------------

  combinatorial : process(rd, wr, hAddr, hDIn, hDOut_r, sDIn, state_r, opBegun_x,
                          activeFlag_r, activeRow_r, rdPipeline_r, wrPipeline_r,
                          hDOutOppPhase_r, nopCntr_r, lock, rfshCntr_r, timer_r, rasTimer_r,
                          wrTimer_r, refTimer_r, cmd_r, cke_r)
  begin

    ------------------------------------------------------------
    -- setup default values for signals
    ------------------------------------------------------------
```

```
    opBegun_x    <= '0';               -- no operations have begun
    earlyOpBegun <= opBegun_x;
    cke_x        <= '1';               -- enable SDRAM clock
    cmd_x        <= NOP_CMD;           -- set SDRAM command to no-operation
    sDataDir_x   <= INPUT;             -- accept data from the SDRAM
    sData_x      <= hDIn(sData_x'range);  -- output data from host to SDRAM
    state_x      <= state_r;           -- reload these registers and flags
    activeFlag_x <= activeFlag_r;      --             with their existing values
    activeRow_x  <= activeRow_r;
    activeBank_x <= activeBank_r;
    rfshCntr_x   <= rfshCntr_r;

    ------------------------------------------------------------
    -- setup default value for the SDRAM address
    ------------------------------------------------------------

    -- extract bank field from host address
    ba_x                <= hAddr(ba'length + ROW_LEN + COL_LEN - 1 downto
ROW_LEN + COL_LEN);
    if MULTIPLE_ACTIVE_ROWS = true then
      bank              <= (others => '0');
      bankIndex             <= CONV_INTEGER(ba_x);
    else
      bank              <= ba_x;
      bankIndex             <= 0;
    end if;
    -- extract row, column fields from host address
    row                 <= hAddr(ROW_LEN + COL_LEN - 1 downto COL_LEN);
    -- extend column (if needed) until it is as large as the (SDRAM address bus - 1)
    col                 <= (others => '0');  -- set it to all zeroes
    col(COL_LEN-1 downto 0) <= hAddr(COL_LEN-1 downto 0);
    -- by default, set SDRAM address to the column address with interspersed
    -- command bit set to disable auto-precharge
    sAddr_x             <= col(col'high-1 downto CMDBIT_POS) & AUTO_PCHG_OFF
                       & col(CMDBIT_POS-1 downto 0);

    ------------------------------------------------------------
    -- manage the read and write operation pipelines
    ------------------------------------------------------------

    -- determine if read operations are in progress by the presence of
    -- READ flags in the read pipeline
    if rdPipeline_r(rdPipeline_r'high downto 1) /= 0 then
      rdInProgress <= '1';
    else
```

```
      rdInProgress <= '0';
    end if;
    rdPending      <= rdInProgress;     -- tell the host if read operations are in progress

    -- enter NOPs into the read and write pipeline shift registers by default
    rdPipeline_x    <= NOP & rdPipeline_r(rdPipeline_r'high downto 1);
    wrPipeline_x(0) <= NOP;

    -- transfer data from SDRAM to the host data register if a read flag has exited the
pipeline
    -- (the transfer occurs 1 cycle before we tell the host the read operation is done)
    if rdPipeline_r(1) = READ then
      hDOutOppPhase_x <= sDIn(hDOut'range);  -- gets value on the SDRAM databus on
the opposite phase
      if IN_PHASE then
        -- get the SDRAM data for the host directly from the SDRAM if the controller and
SDRAM are in-phase
        hDOut_x       <= sDIn(hDOut'range);
      else
        -- otherwise get the SDRAM data that was gathered on the previous opposite clock
edge
        hDOut_x       <= hDOutOppPhase_r(hDOut'range);
      end if;
    else
      -- retain contents of host data registers if no data from the SDRAM has arrived yet
      hDOutOppPhase_x <= hDOutOppPhase_r;
      hDOut_x         <= hDOut_r;
    end if;

    done   <= rdPipeline_r(0) or wrPipeline_r(0);  -- a read or write operation is done
    rdDone <= rdPipeline_r(0);          -- SDRAM data available when a READ flag exits
the pipeline

    ------------------------------------------------------------
    -- manage row activation
    ------------------------------------------------------------

    -- request a row activation operation if the row of the current address
    -- does not match the currently active row in the bank, or if no row
    -- in the bank is currently active
    if (bank /= activeBank_r) or (row /= activeRow_r(bankIndex)) or
(activeFlag_r(bankIndex) = '0') then
      doActivate <= '1';
    else
      doActivate <= '0';
    end if;
```

```
   ------------------------------------------------------------
   -- manage self-refresh
   ------------------------------------------------------------

   -- enter self-refresh if neither a read or write is requested for MAX_NOP consecutive
cycles.
   if (rd = '1') or (wr = '1') then
     -- any read or write resets NOP counter and exits self-refresh state
     nopCntr_x  <= 0;
     doSelfRfsh <= '0';
   elsif nopCntr_r /= MAX_NOP then
     -- increment NOP counter whenever there is no read or write operation
     nopCntr_x  <= nopCntr_r + 1;
     doSelfRfsh <= '0';
   else
     -- start self-refresh when counter hits maximum NOP count and leave counter
unchanged
     nopCntr_x  <= nopCntr_r;
     doSelfRfsh <= '1';
   end if;

   ------------------------------------------------------------
   -- update the timers
   ------------------------------------------------------------

   -- row activation timer
   if rasTimer_r /= 0 then
     -- decrement a non-zero timer and set the flag
     -- to indicate the row activation is still inprogress
     rasTimer_x        <= rasTimer_r - 1;
     activateInProgress <= '1';
   else
     -- on timeout, keep the timer at zero     and reset the flag
     -- to indicate the row activation operation is done
     rasTimer_x        <= rasTimer_r;
     activateInProgress <= '0';
   end if;

   -- write operation timer
   if wrTimer_r /= 0 then
     -- decrement a non-zero timer and set the flag
     -- to indicate the write operation is still inprogress
     wrTimer_x    <= wrTimer_r - 1;
     wrInPRogress <= '1';
   else
```

```
      -- on timeout, keep the timer at zero and reset the flag that
      -- indicates a write operation is in progress
      wrTimer_x    <= wrTimer_r;
      wrInPRogress <= '0';
    end if;

    -- refresh timer
    if refTimer_r /= 0 then
      refTimer_x <= refTimer_r - 1;
    else
      -- on timeout, reload the timer with the interval between row refreshes
      -- and increment the counter for the number of row refreshes that are needed
      refTimer_x <= REF_CYCLES;
      rfshCntr_x <= rfshCntr_r + 1;
    end if;

    -- main timer for sequencing SDRAM operations
    if timer_r /= 0 then
      -- decrement the timer and do nothing else since the previous operation has not
completed yet.
      timer_x <= timer_r - 1;
      status  <= "0000";
    else
      -- the previous operation has completed once the timer hits zero
      timer_x <= timer_r;              -- by default, leave the timer at zero

      ------------------------------------------------------------
      -- compute the next state and outputs
      ------------------------------------------------------------
      case state_r is

        ------------------------------------------------------------
        -- let clock stabilize and then wait for the SDRAM to initialize
        ------------------------------------------------------------
        when INITWAIT =>
          if lock = '1' then
                                -- wait for SDRAM power-on initialization once the clock is
stable
            timer_x <= INIT_CYCLES;     -- set timer for initialization duration
            state_x <= INITPCHG;
          else
                                -- disable SDRAM clock and return to this state if the clock is
not stable
                                -- this insures the clock is stable before enabling the SDRAM
                                -- it also insures a clean startup if the SDRAM is currently in
self-refresh mode
```

```
          cke_x   <= '0';
        end if;
        status    <= "0001";

        ------------------------------------------------------------
        -- precharge all SDRAM banks after power-on initialization
        ------------------------------------------------------------
      when INITPCHG =>
        cmd_x             <= PCHG_CMD;
        sAddr_x(CMDBIT_POS) <= ALL_BANKS;  -- precharge all banks
        timer_x           <= RP_CYCLES;  -- set timer for precharge operation duration
        rfshCntr_x         <= RFSH_OPS;  -- set counter for refresh ops needed after
precharge
        state_x           <= INITRFSH;
        status            <= "0010";

        ------------------------------------------------------------
        -- refresh the SDRAM a number of times after initial precharge
        ------------------------------------------------------------
      when INITRFSH =>
        cmd_x      <= RFSH_CMD;
        timer_x    <= RFC_CYCLES;     -- set timer to refresh operation duration
        rfshCntr_x <= rfshCntr_r - 1;  -- decrement refresh operation counter
        if rfshCntr_r = 1 then
          state_x  <= INITSETMODE;    -- set the SDRAM mode once all refresh ops are
done
        end if;
        status     <= "0011";

        ------------------------------------------------------------
        -- set the mode register of the SDRAM
        ------------------------------------------------------------
      when INITSETMODE =>
        cmd_x   <= MODE_CMD;
        sAddr_x <= MODE;             -- output mode register bits on the SDRAM address
bits
        timer_x <= MODE_CYCLES;       -- set timer for mode setting operation duration
        state_x <= RW;
        status  <= "0100";

        ------------------------------------------------------------
        -- process read/write/refresh operations after initialization is done
        ------------------------------------------------------------
      when RW                              =>
        ------------------------------------------------------------
        -- highest priority operation: row refresh
```

```
        -- do a refresh operation if the refresh counter is non-zero
        ------------------------------------------------------------
        if rfshCntr_r /= 0 then
                        -- wait for any row activations, writes or reads to finish before
doing a precharge
          if (activateInProgress = '0') and (wrInProgress = '0') and (rdInProgress = '0') then
            cmd_x                <= PCHG_CMD;  -- initiate precharge of the SDRAM
            sAddr_x(CMDBIT_POS)      <= ALL_BANKS;  -- precharge all banks
            timer_x              <= RP_CYCLES;  -- set timer for this operation
            activeFlag_x           <= (others => '0');  -- all rows are inactive after a
precharge operation
            state_x              <= REFRESHROW;  -- refresh the SDRAM after the
precharge
          end if;
          status                 <= "0101";
          ------------------------------------------------------------
          -- do a host-initiated read operation
          ------------------------------------------------------------
        elsif rd = '1' then
                                        -- Wait one clock cycle if the bank address has just
changed and each bank has its own active row.
          -- This gives extra time for the row activation circuitry.
          if (ba_x = ba_r) or (MULTIPLE_ACTIVE_ROWS=false) then
                        -- activate a new row if the current read is outside the active row
or bank
            if doActivate = '1' then
                        -- activate new row only if all previous activations, writes, reads
are done
              if (activateInProgress = '0') and (wrInProgress = '0') and (rdInProgress = '0')
then
                cmd_x            <= PCHG_CMD;  -- initiate precharge of the SDRAM
                sAddr_x(CMDBIT_POS)   <= ONE_BANK;  -- precharge this bank
                timer_x          <= RP_CYCLES;  -- set timer for this operation
                activeFlag_x(bankIndex) <= '0';  -- rows in this bank are inactive after a
precharge operation
                state_x          <= ACTIVATE;  -- activate the new row after the precharge
is done
              end if;
                        -- read from the currently active row if no previous read
operation
                        -- is in progress or if pipeline reads are enabled
                        -- we can always initiate a read even if a write is already in
progress
            elsif (rdInProgress = '0') or PIPE_EN then
              cmd_x              <= READ_CMD;  -- initiate a read of the SDRAM
```

```
                        -- insert a flag into the pipeline shift register that will exit the
end
                        -- of the shift register when the data from the SDRAM is
available
            rdPipeline_x          <= READ & rdPipeline_r(rdPipeline_r'high downto 1);
            opBegun_x             <= '1';  -- tell the host the requested operation has begun
          end if;
        end if;
        status                  <= "0110";
        ------------------------------------------------------------
        -- do a host-initiated write operation
        ------------------------------------------------------------
      elsif wr = '1' then
                                        -- Wait one clock cycle if the bank address has just
changed and each bank has its own active row.
        -- This gives extra time for the row activation circuitry.
        if (ba_x = ba_r) or (MULTIPLE_ACTIVE_ROWS=false) then
                        -- activate a new row if the current write is outside the active
row or bank
          if doActivate = '1' then
                        -- activate new row only if all previous activations, writes, reads
are done
            if (activateInProgress = '0') and (wrInProgress = '0') and (rdInProgress = '0')
then
              cmd_x             <= PCHG_CMD;  -- initiate precharge of the SDRAM
              sAddr_x(CMDBIT_POS)   <= ONE_BANK;  -- precharge this bank
              timer_x           <= RP_CYCLES;  -- set timer for this operation
              activeFlag_x(bankIndex) <= '0';  -- rows in this bank are inactive after a
precharge operation
              state_x           <= ACTIVATE;  -- activate the new row after the precharge
is done
            end if;
                        -- write to the currently active row if no previous read operations
are in progress
          elsif rdInProgress = '0' then
            cmd_x               <= WRITE_CMD;  -- initiate the write operation
            sDataDir_x          <= OUTPUT;  -- turn on drivers to send data to SDRAM
                        -- set timer so precharge doesn't occur too soon after write
operation
            wrTimer_x           <= WR_CYCLES;
                        -- insert a flag into the 1-bit pipeline shift register that will exit
on the
                        -- next cycle.  The write into SDRAM is not actually done by
that time, but
                        -- this doesn't matter to the host
            wrPipeline_x(0)     <= WRITE;
```

```
            opBegun_x               <= '1';  -- tell the host the requested operation has begun
          end if;
        end if;
        status                    <= "0111";
        -------------------------------------------------------
        -- do a host-initiated self-refresh operation
        -------------------------------------------------------
      elsif doSelfRfsh = '1' then
                              -- wait until all previous activations, writes, reads are done
        if (activateInProgress = '0') and (wrInProgress = '0') and (rdInProgress = '0') then
          cmd_x                   <= PCHG_CMD;  -- initiate precharge of the SDRAM
          sAddr_x(CMDBIT_POS)        <= ALL_BANKS;  -- precharge all banks
          timer_x                 <= RP_CYCLES;  -- set timer for this operation
          activeFlag_x              <= (others => '0');  -- all rows are inactive after a
precharge operation
          state_x                 <= SELFREFRESH;  -- self-refresh the SDRAM after the
precharge
        end if;
        status                    <= "1000";
        -------------------------------------------------------
        -- no operation
        -------------------------------------------------------
      else
        state_x                   <= RW;  -- continue to look for SDRAM operations to
execute
        status                    <= "1001";
      end if;

      -------------------------------------------------------
      -- activate a row of the SDRAM
      -------------------------------------------------------
    when ACTIVATE                     =>
      cmd_x                 <= ACTIVE_CMD;
      sAddr_x               <= (others => '0');  -- output the address for the row to be
activated
      sAddr_x(row'range)      <= row;
      activeBank_x            <= bank;
      activeRow_x(bankIndex)  <= row;  -- store the new active SDRAM row address
      activeFlag_x(bankIndex) <= '1';  -- the SDRAM is now active
      rasTimer_x              <= RAS_CYCLES;  -- minimum time before another
precharge can occur
      timer_x               <= RCD_CYCLES;  -- minimum time before a read/write
operation can occur
      state_x               <= RW;  -- return to do read/write operation that initiated this
activation
      status                <= "1010";
```

```
        ------------------------------------------------------------
        -- refresh a row of the SDRAM
        ------------------------------------------------------------
      when REFRESHROW =>
        cmd_x      <= RFSH_CMD;
        timer_x    <= RFC_CYCLES;     -- refresh operation interval
        rfshCntr_x <= rfshCntr_r - 1;  -- decrement the number of needed row refreshes
        state_x    <= RW;             -- process more SDRAM operations after refresh is done
        status     <= "1011";

        ------------------------------------------------------------
        -- place the SDRAM into self-refresh and keep it there until further notice
        ------------------------------------------------------------
      when SELFREFRESH           =>
        if (doSelfRfsh = '1') or (lock = '0') then
                              -- keep the SDRAM in self-refresh mode as long as requested
and until there is a stable clock
          cmd_x       <= RFSH_CMD;   -- output the refresh command; this is only needed
on the first clock cycle
          cke_x       <= '0';        -- disable the SDRAM clock
        else
                              -- else exit self-refresh mode and start processing read and write
operations
          cke_x       <= '1';       -- restart the SDRAM clock
          rfshCntr_x   <= 0;         -- no refreshes are needed immediately after leaving self-
refresh
          activeFlag_x <= (others => '0');  -- self-refresh deactivates all rows
          timer_x      <= XSR_CYCLES;  -- wait this long until read and write operations
can resume
          state_x      <= RW;
        end if;
        status        <= "1100";

        ------------------------------------------------------------
        -- unknown state
        ------------------------------------------------------------
      when others =>
        state_x <= INITWAIT;         -- reset state if in erroneous state
        status  <= "1101";

    end case;
  end if;
end process combinatorial;
```

```
------------------------------------------------------------
-- update registers on the appropriate clock edge
------------------------------------------------------------

update : process(rst, clk)
begin

  if rst = '1' then
    -- asynchronous reset
    state_r      <= INITWAIT;
    activeFlag_r <= (others => '0');
    rfshCntr_r   <= 0;
    timer_r      <= 0;
    refTimer_r   <= REF_CYCLES;
    rasTimer_r   <= 0;
    wrTimer_r    <= 0;
    nopCntr_r    <= 0;
    opBegun_r    <= '0';
    rdPipeline_r <= (others => '0');
    wrPipeline_r <= (others => '0');
    cke_r        <= '0';
    cmd_r        <= NOP_CMD;
    ba_r         <= (others => '0');
    sAddr_r      <= (others => '0');
    sData_r      <= (others => '0');
    sDataDir_r   <= INPUT;
    hDOut_r      <= (others => '0');
  elsif rising_edge(clk) then
    state_r      <= state_x;
    activeBank_r <= activeBank_x;
    activeRow_r  <= activeRow_x;
    activeFlag_r <= activeFlag_x;
    rfshCntr_r   <= rfshCntr_x;
    timer_r      <= timer_x;
    refTimer_r   <= refTimer_x;
    rasTimer_r   <= rasTimer_x;
    wrTimer_r    <= wrTimer_x;
    nopCntr_r    <= nopCntr_x;
    opBegun_r    <= opBegun_x;
    rdPipeline_r <= rdPipeline_x;
    wrPipeline_r <= wrPipeline_x;
    cke_r        <= cke_x;
    cmd_r        <= cmd_x;
    ba_r         <= ba_x;
    sAddr_r      <= sAddr_x;
    sData_r      <= sData_x;
```

```
    sDataDir_r   <= sDataDir_x;
    hDOut_r      <= hDOut_x;
   end if;

   -- the register that gets data from the SDRAM and holds it for the host
   -- is clocked on the opposite edge.  We don't use this register if IN_PHASE=TRUE.
   if rst = '1' then
    hDOutOppPhase_r <= (others => '0');
   elsif falling_edge(clk) then
    hDOutOppPhase_r <= hDOutOppPhase_x;
   end if;

  end process update;

end Behavioral;
```

4.3.5 opb_Ethernet.vhd

```
------------------------------
--
------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity opb_ethernet is
  generic (
    C_OPB_AWIDTH : integer                := 32;
    C_OPB_DWIDTH : integer                := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"00800000";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"00FFFFFF");

  port (
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;        -- Sequential Address
    Sln_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_errAck  : out std_logic;        -- (unused)
    Sln_retry   : out std_logic;        -- (unused)
```

```
    Sln_toutSup : out std_logic;        -- Timeout suppress
    Sln_xferAck : out std_logic;        -- Transfer acknowledge

    PB_DIN   : in std_logic_vector(0 to 15);      -- ETHER data bus
    PB_DOUT  : out std_logic_vector(0 to 15);     -- ETHER data bus output
    PB_DOUT_EN : out std_logic;                   -- ETHER data bus output enable
    PB_A : out std_logic_vector (0 to 19);
    IORD_N : out std_logic;
    IOWR_N : out std_logic;
    AEN : out std_logic;
    BHE_N : out std_logic;
    ETHERNET_CS_N : out std_logic;
    ETHERNET_RDY : in std_logic;
    ETHERNET_IREQ : in std_logic;
    ETHERNET_IOCS16_N : in std_logic
          );
end opb_ethernet;

architecture Behavioral of opb_ethernet is


  constant ETHER_AWIDTH : integer := 10;  -- Number of address lines on the ETHER
  constant ETHER_DWIDTH : integer := 16;  -- Number of data lines on the ETHER

  signal RNW : std_logic;
  signal ETHER_DI: std_logic_vector(0 to 31);
  signal ABus : std_logic_vector(0 to ETHER_AWIDTH-1);
  signal chip_select : std_logic;
  signal output_enable : std_logic;
  signal TE : std_logic;
  signal BE : std_logic_vector(0 to 3);
  signal A30: std_logic;
  signal ETHER_RD_DATA : std_logic_vector (0 to 31);
  signal ETHER_WR_DATA : std_logic_vector (0 to 31);

  signal CE : std_logic;          -- ETHER chip enalbe
  signal OE : std_logic;          -- ETHER output enalbe
  signal xfer_start: std_logic;    -- Transfer start
  signal xfer_end: std_logic;      -- Transfer end
  signal xfer_idle: std_logic;     -- No transfer pending
  signal obuf_oe: std_logic;       -- Data output buffer enable
  signal xfer_done: std_logic;     -- Transfer has been done
  signal data_latch: std_logic;

  -- Critical: Sln_xferAck is generated directly from state bit 0!
  constant STATE_BITS : integer := 3;
```

```
  constant ST_IDLE      : std_logic_vector(0 to STATE_BITS-1) := "000";
  constant ST_SELECTED  : std_logic_vector(0 to STATE_BITS-1) := "001";
  constant ST_WAIT      : std_logic_vector(0 to STATE_BITS-1) := "011";
  constant ST_XFER      : std_logic_vector(0 to STATE_BITS-1) := "111";
  constant ST_WAIT1     : std_logic_vector(0 to STATE_BITS-1) := "100";
  constant ST_WAIT2     : std_logic_vector(0 to STATE_BITS-1) := "101";

  signal present_state, next_state : std_logic_vector(0 to STATE_BITS-1);

begin

  register_opb_inputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      ETHER_DI <= (others => '0');
      ABus <= (others => '0');
      RNW <= '0';
      A30 <=  '0';
      BE <= (others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
      ETHER_DI <= OPB_DBus(0 to C_OPB_DWIDTH-1);
      ABus <= OPB_ABus(C_OPB_AWIDTH-2-(ETHER_AWIDTH-1) to
C_OPB_AWIDTH-2);
      RNW <= OPB_RNW;
      A30 <= OPB_ABus(30);
      BE <= OPB_BE(0 to 3);
    end if;
  end process register_opb_inputs;

  register_opb_outputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      Sln_DBus <= (others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
--      if ( output_enable = '1' and RNW = '1') then
     if ( output_enable = '1') then
        Sln_DBus <= ETHER_RD_DATA;
     else
       Sln_DBus <= (others => '0');
     end if;
    end if;
  end process register_opb_outputs;

  -- Unused outputs
  Sln_errAck  <= '0';
  Sln_retry   <= '0';
```

```
Sln_toutSup <= '0';
--Sln_DBus(ETHER_DWIDTH to C_OPB_DWIDTH-1) <= (others => '0');

chip_select <=
  '1' when OPB_select = '1' and
      OPB_ABus(0 to C_OPB_AWIDTH-2-ETHER_AWIDTH) =
      C_BASEADDR(0 to C_OPB_AWIDTH-2-ETHER_AWIDTH) else
  '0';

Sln_xferAck <= xfer_done;

-- Sequential part of the FSM
fsm_seq : process(OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    present_state <= ST_IDLE;
  elsif OPB_Clk'event and OPB_Clk = '1' then
    present_state <= next_state;
  end if;
end process fsm_seq;

-- Combinational part of the FSM
fsm_comb : process(present_state, chip_select, OPB_Select)
begin
    case present_state is
      when ST_IDLE =>
        if chip_select = '1' then
          next_state <= ST_SELECTED;
        else
          next_state <= ST_IDLE;
        end if;

      when ST_SELECTED =>
        if OPB_Select = '1' then
            next_state <= ST_WAIT;
        else
          next_state <= ST_IDLE;
        end if;

      when ST_WAIT =>
        if OPB_Select = '1' then
          next_state <= ST_WAIT1;
        else
          next_state <= ST_IDLE;
        end if;
      when ST_WAIT1 =>
```

```
          if OPB_Select = '1' then
            next_state <= ST_WAIT2;
          else
            next_state <= ST_IDLE;
          end if;
        when ST_WAIT2 =>
            if OPB_Select = '1' then
              next_state <= ST_XFER;
            else
              next_state <= ST_IDLE;
            end if;

        -- State encoding is critical here: xfer must only be true here
        when ST_XFER =>
          next_state <= ST_IDLE;

        when others =>
          next_state <= ST_IDLE;
      end case;

  end process fsm_comb;

  fsm_output: process (present_state)
  begin
    output_enable <= '0';
    xfer_start <= '0';
    xfer_end <= '0';
    xfer_idle <= '0';
    xfer_done <= '0';
    case present_state is
      when ST_IDLE =>
        xfer_idle <= '1';
      when ST_SELECTED =>
        xfer_start <= '1';
      when ST_WAIT2 =>
        output_enable <= '1';
      when ST_XFER =>
        xfer_end <= '1';
        xfer_done <= '1';
      when others =>
        output_enable <= '0';
        xfer_start <= '0';
        xfer_end <= '0';
        xfer_idle <= '0';
    end case;
  end process fsm_output;
```

```
-- The following process generates the OE output signal
ether_oe_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    OE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( chip_select = '1' and xfer_idle = '1') then
      OE <= not OPB_RNW;
    elsif ( xfer_end = '1') then
      OE <= '1';
    end if;
  end if;
end process ether_oe_reg;

-- The following process generates the chip enable output signal
ether_ce_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    CE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( chip_select = '1' and xfer_idle = '1') then
      CE <= '0';
    elsif ( xfer_end = '1') then
      CE <= '1';
    end if;
  end if;
end process ether_ce_reg;

-- The following process generates the TE output signal
ether_we_reg: process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    TE <= '1';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( xfer_start = '1') then
      TE <= '0';
    elsif (output_enable = '1') then
      TE <= '1';
    end if;
  end if;
end process ether_we_reg;

--

ETHER_RD_DATA <= PB_DIN & PB_DIN;
```

```
 --  ETHER_WR_DATA <= ETHER_DI;
  process ( ETHER_DI, A30)
  begin
       if ( A30 = '1') then
         ETHER_WR_DATA (16 to 31) <= ETHER_DI(16 to 31);
       else
         ETHER_WR_DATA (16 to 31) <= ETHER_DI(0 to 15);
       end if;
  end process;

  IORD_N <= OE;

  -- The following assign the output signals
  process ( RNW, TE)
  begin
    if ( RNW = '1') then
--       IORD_N <= TE;
       IOWR_N <= '1';
    else
--       IORD_N <= '1';
       IOWR_N <= TE;
    end if;
  end process;

--  process ( BE )
--  begin
--     if ( BE = "1100" or BE = "0011" ) then
--       BHE_N <= '0';
--     else
--       BHE_N <= '1';
--     end if;
--  end process;

  BHE_N <= CE;

  ETHERNET_CS_N <= CE;
  AEN <= CE;
  PB_A <= "0000000000" & ABus;
  PB_DOUT <= ETHER_WR_DATA(16 to 31);
  PB_DOUT_EN <= (not CE) and OE;


end Behavioral;
```

4.3.6 opb_audio_controller.vhd

```
------------------------------------------------------------------------------
--
-- Audio Controller OPB Peripheral
--
--
------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity opb_audio_controller is

  generic (
    C_OPB_AWIDTH : integer                  := 32;
    C_OPB_DWIDTH : integer                  := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"FEFF0400";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"FEFF04FF");

  port (
    -- OPB Input Signals
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;        -- Sequential Address

    -- OPB Output Signals
    Sln_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_errAck  : out std_logic;        -- (unused)
    Sln_retry   : out std_logic;        -- (unused)
    Sln_toutSup : out std_logic;        -- Timeout suppress
    Sln_xferAck : out std_logic;        -- Transfer acknowledge

    -- Audio IO Signals
    AU_BUSY     : out std_logic;        -- Audio interface busy
    AU_CSN      : out std_logic;        -- Audio Chip Select
    AU_CCLK     : out std_logic;        -- Audio Cntl Clock
    AU_CDTI     : out std_logic;        -- Audio Cntl Data In
    PB_DIN      : in  std_logic_vector(0 to 15)
    --AU_CDTO     : in  std_logic           -- Audio Cntl Data Out
```

```
    );
end opb_audio_controller;


architecture behavioral of opb_audio_controller is

  signal AU_CDTO: std_logic;
  signal Command : std_logic_vector(0 to 15);
  signal Status: std_logic_vector (0 to 7);
  signal Busy : std_logic;
  signal SetBusy : std_logic;
  signal RNW : std_logic;
  signal CmdLoad: std_logic;
  signal chip_select : std_logic;
  signal output_enable : std_logic;
  signal audio_done: std_logic;
  signal shift_enable: std_logic;
  signal capture_enable: std_logic;
  signal shift_mode: std_logic;
  signal rising: std_logic;
  signal falling: std_logic;
  signal clear: std_logic;
  signal cs_setup: std_logic;
  signal cs_hold: std_logic;
  signal baud_counter: std_logic_vector ( 0 to 3 );
  signal bit_counter: std_logic_vector (0 to 3);
  signal bit_counter_ov: std_logic;

  constant ST_AU_IDLE:  std_logic_vector( 0 to 2) := "000";
  constant ST_AU_START:  std_logic_vector( 0 to 2) := "001";
  constant ST_AU_SETUP:  std_logic_vector( 0 to 2) := "010";
  constant ST_AU_SHIFT:  std_logic_vector( 0 to 2) := "011";
  constant ST_AU_END:  std_logic_vector( 0 to 2) := "100";
  constant ST_AU_HOLD:  std_logic_vector( 0 to 2) := "101";
  signal au_current_state, au_next_state : std_logic_vector (0 to 2);

  signal xfer_start: std_logic;    -- Transfer start
  signal xfer_end: std_logic;      -- Transfer end
  signal xfer_idle: std_logic;     -- No transfer pending
  signal obuf_oe: std_logic;       -- Data output buffer enable
  signal xfer_done: std_logic;     -- Transfer has been done

  -- Critical: Sln_xferAck is generated directly from state bit 0!
  constant STATE_BITS : integer := 3;
  constant ST_IDLE      : std_logic_vector(0 to STATE_BITS-1) := "000";
  constant ST_SELECTED  : std_logic_vector(0 to STATE_BITS-1) := "001";
```

```
  constant ST_WAIT      : std_logic_vector(0 to STATE_BITS-1) := "011";
  constant ST_XFER      : std_logic_vector(0 to STATE_BITS-1) := "111";

  signal present_state, next_state : std_logic_vector(0 to STATE_BITS-1);

begin

  ------------------------------------------------------------------------
  -- OPB interface
  ------------------------------------------------------------------------
  register_opb_inputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      SetBusy <=  '0';
      RNW <= '0';
    elsif OPB_Clk'event and OPB_Clk = '1' then
      SetBusy <= OPB_DBus(0);
      RNW <= OPB_RNW;
    end if;
  end process register_opb_inputs;

  register_opb_outputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      Sln_DBus <= (others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
     if (output_enable = '1' and RNW = '1')  then
        Sln_DBus <= Busy & "0000000" & Status & "0000000000000000";
     else
       Sln_DBus <= (others => '0');
     end if;
    end if;
  end process register_opb_outputs;

  -- Unused outputs
  Sln_errAck  <= '0';
  Sln_retry   <= '0';
  Sln_toutSup <= '0';

  chip_select <=
    '1' when OPB_select = '1' and
        OPB_ABus(0 to C_OPB_AWIDTH-1) = C_BASEADDR(0 to C_OPB_AWIDTH-1)
else
    '0';

  Sln_xferAck <= present_state(0);
```

```
-- Sequential part of the FSM
fsm_seq : process(OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    present_state <= ST_IDLE;
  elsif OPB_Clk'event and OPB_Clk = '1' then
    present_state <= next_state;
  end if;
end process fsm_seq;

-- Combinational part of the FSM
fsm_comb : process(present_state, chip_select, OPB_Select)
begin
    case present_state is
      when ST_IDLE =>
        if chip_select = '1' then
          next_state <= ST_SELECTED;
        else
          next_state <= ST_IDLE;
        end if;
      when ST_SELECTED =>
        if OPB_Select = '1' then
           next_state <= ST_WAIT;
        else
          next_state <= ST_IDLE;
        end if;
      when ST_WAIT =>
        if OPB_Select = '1' then
          next_state <= ST_XFER;
        else
          next_state <= ST_IDLE;
        end if;
      -- State encoding is critical here: xfer must only be true here
      when ST_XFER =>
        next_state <= ST_IDLE;
      when others =>
        next_state <= ST_IDLE;
    end case;

end process fsm_comb;

process (present_state)
begin
   if ( present_state = ST_WAIT) then
      output_enable <= '1';
```

```
  else
     output_enable <= '0';
  end if;
end process;

-------------------------------------------------------------------------
-- Audio interface
-------------------------------------------------------------------------

AU_CDTO <= PB_DIN (13); -- PB_D2

process (present_state , SetBusy)
begin
  if ( present_state = ST_WAIT and SetBusy = '1' and RNW = '0' ) then
     CmdLoad <= '1';
  else
    CmdLoad <= '0';
  end if;
end process;

-- Audio controller is busy shifting out the command
-- Must be assigned the PH bus at this time
process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    Busy <= '0';
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( CmdLoad = '1' ) then
      Busy <= '1';
    elsif (audio_done = '1') then
      Busy <= '0';
    end if;
  end if;
end process;
AU_BUSY <= Busy;

-- Audio command register
process (OPB_Clk, OPB_Rst)
begin
  if OPB_Rst = '1' then
    Command <= (others => '0');
  elsif OPB_Clk 'event and OPB_Clk = '1' then
    if ( CmdLoad = '1' ) then
      Command <= OPB_DBus(16 to 31);
    elsif (shift_enable = '1') then
      Command <= Command( 1 to 15) & '0';
```

```
      end if;
    end if;
end process;

-- Audio controll data output
AU_CDTI <= Command (0);

-- Audio read back data
process (OPB_Clk, OPB_Rst)
begin
    if OPB_Rst = '1' then
      Status <= (others => '0');
    elsif OPB_Clk 'event and OPB_Clk = '1' then
      if (capture_enable = '1') then
        Status <= AU_CDTO & Status( 0 to 6) ;
      end if;
    end if;
end process;

-- Counter for the CCLK divider
-- 50MHz / 16
process (OPB_Clk, OPB_Rst)
begin
    if ( OPB_Rst = '1') then
      baud_counter <= "0000";
    elsif ( OPB_Clk 'event and OPB_Clk = '1') then
      if (clear = '1') then
        baud_counter <= "0000";
      else
        baud_counter <= baud_counter + '1';
      end if;
    end if;
end process;

-- Detect the time for the CCLK rising and falling
process ( baud_counter )
begin
    if ( baud_counter = "0111" ) then
      rising <= '1';
    else
      rising <= '0';
    end if;

    if ( baud_counter = "1111" ) then
      falling <= '1';
    else
```

```
      falling <= '0';
   end if;

end process;

-- Audio controller serial shift bit counter
process ( OPB_Clk, OPB_Rst)
begin
   if (OPB_Rst = '1' ) then
      bit_counter <= "0000";
   elsif ( OPB_Clk 'event and OPB_Clk = '1') then
      if ( clear = '1' or cs_setup = '1' ) then
         bit_counter <= "0000";
      elsif (falling = '1') then
         bit_counter <= bit_counter + '1';
      end if;
   end if;
end process;

process (bit_counter )
begin
   if ( bit_counter = "1111") then
      bit_counter_ov <= '1';
   else
      bit_counter_ov <= '0';
   end if;
end process;

-- Data output shift enable
shift_enable <= shift_mode and falling;

-- Data input shift enable
capture_enable <= shift_mode and rising;

-- Audio controller state machine
process ( OPB_Clk, OPB_Rst)
begin
   if ( OPB_Rst = '1') then
      au_current_state <= ST_AU_IDLE;
   elsif (OPB_Clk 'event and OPB_Clk = '1') then
      au_current_state <= au_next_state;
   end if;
end process;

-- Audio controller next state logic
process ( au_current_state, CmdLoad, rising, bit_counter_ov)
```

```
begin
  case au_current_state is
    when ST_AU_IDLE =>
      if ( CmdLoad = '1') then
        au_next_state <= ST_AU_START;
      else
        au_next_state <= ST_AU_IDLE;
      end if;
    when ST_AU_START =>
      if ( rising = '1' ) then
        au_next_state <= ST_AU_SETUP;
      else
        au_next_state <= ST_AU_START;
      end if;
    when ST_AU_SETUP =>
      if ( rising = '1' ) then
        au_next_state <= ST_AU_SHIFT;
      else
        au_next_state <= ST_AU_SETUP;
      end if;
    when ST_AU_SHIFT =>
      if ( rising = '1' and bit_counter_ov = '1') then
        au_next_state <= ST_AU_HOLD;
      else
        au_next_state <= ST_AU_SHIFT;
      end if;
    when ST_AU_HOLD =>
      if ( rising = '1' ) then
        au_next_state <= ST_AU_END;
      else
        au_next_state <= ST_AU_HOLD;
      end if;
    when ST_AU_END =>
      au_next_state <= ST_AU_IDLE;
    when others =>
      au_next_state <= ST_AU_IDLE;
  end case;

end process;

-- Audio controller control signal decode from state machine
process ( au_current_state, rising, bit_counter_ov)
begin
  shift_mode <= '0';
  clear <= '0';
  cs_setup <= '0';
```

```
  cs_hold <= '0';
  audio_done <= '0';
  case au_current_state is
    when ST_AU_IDLE =>
      shift_mode <= '0';
      clear <= '1';
      cs_setup <= '0';
      audio_done <= '0';
    when ST_AU_SETUP =>
      cs_setup <= '1';
    when ST_AU_SHIFT =>
      shift_mode <= '1';
    when ST_AU_HOLD =>
      cs_hold <= '1';
    when ST_AU_END =>
      audio_done <= '1';
    when others =>
      shift_mode <= '0';
      clear <= '0';
      cs_setup <= '0';
      audio_done <= '0';
      cs_hold <= '0';
  end case;
end process;

process ( OPB_Clk, OPB_Rst)
begin
  if ( OPB_Rst = '1') then
    AU_CSN <= '1';
  elsif (OPB_Clk 'event and OPB_Clk = '1') then
    if ( clear = '1' or audio_done = '1') then
      AU_CSN <= '1';
    elsif ( cs_setup = '1' ) then
      AU_CSN <= '0';
    end if ;
  end if;
end process;

process ( OPB_Clk, OPB_Rst)
begin
  if ( OPB_Rst = '1') then
    AU_CCLK <= '1';
  elsif (OPB_Clk 'event and OPB_Clk = '1') then
    if ( clear = '1' or cs_hold = '1' or rising = '1') then
      AU_CCLK <= '1';
    elsif ( falling = '1' and cs_hold = '0' ) then
```

```
        AU_CCLK <= '0';
      end if ;
    end if;
  end process;

end behavioral;
```

4.3.7 opb_audio_sampler.vhd

```
------------------------------
-- CSEE 4840 Embedded System Design - Audio Sampling OPB Peripheral
--
-- SOBA Server
--
--  Team Warriors: Avraham Shinnar  as1619@columbia.edu
--                 Benjamin Dweck  bjd2102@columbia.edu
--                 Oliver Irwin    omi3@columbia.edu
--                 Sean White      sw2061@columbia.edu
--
------------------------------

--------------------------------------------------------------------------
-- Use clock enable instead of C to do the latch_32
-- Xilinx generates global clock buffer for the C as clock
-- Which caused abused global clock buffer over flow
-- "Number of GCLKs:                  5 out of      4  125% (OVERMAPPED)"
-- 04/11/06


--------------------------------------------------------------------------
--
-- 32-bit Positive Edge Triggered Left Shift Register
-- with Serial In and Parallel Out
--
-- Used to receive samples serially from the audio codec
--
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity shift_32 is
  port(
    SI : in  std_logic;
    CK: in  std_logic;
    C  : in  std_logic;
    D  : out std_logic_vector(31 downto 0)
```

```
  );
end shift_32;

architecture arch of shift_32 is

  signal tmp : std_logic_vector(31 downto 0);
  signal EN, C_L : std_logic;

begin
  -- Add clock to generate a enable signal for the latch
  process (CK)
  begin
    if (CK 'event and CK = '1') then
      C_L <= C;
    end if;
  end process;
  EN <= not C_L and C;

  process (CK)
  begin
   if CK'event and CK='1' then
     if ( EN = '1') then
       tmp <= tmp(30 downto 0) & SI;
     end if;
   end if;
  end process;

  D <= tmp;

end arch;



--------------------------------------------------------------------------------
--
-- 32-bit Positive Edge Triggered Latch
-- with Asynchronous Reset
--
-- Used to receive samples serially from the audio codec
--
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity latch_32 is
```

```
  port(
    R : in  std_logic;
    CK: in  std_logic;
    C : in  std_logic;
    D : in  std_logic_vector(31 downto 0);
    Q : out std_logic_vector(31 downto 0)
  );
end latch_32;

architecture arch of latch_32 is

  signal tmp : std_logic_vector(31 downto 0);
  signal EN, C_L : std_logic;

begin

  -- Add clock to generate a enable signal for the latch
  process (CK, R)
  begin
    if ( R = '1') then
      C_L <= '0';
    elsif (CK 'event and CK = '1') then
      C_L <= C;
    end if;
  end process;
  EN <= not C_L and C;

  process (CK, R)
  begin
    if R='1' then
      tmp <= X"00000000";
    elsif CK'event and CK='1' then
      if ( EN = '1' ) then
        tmp <= D;
      end if;
    end if;
  end process;

  Q <= tmp;

end arch;



--------------------------------------------------------------------------
--
```

```
-- Audio Sampler Peripheral
--
------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity opb_audio_sampler is

  generic (
    C_OPB_AWIDTH : integer                := 32;
    C_OPB_DWIDTH : integer                := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"00000000";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"FFFFFFFF");

  port (
    -- OPB Input Signals
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;       -- Sequential Address

    -- OPB Output Signals
    AUS_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    AUS_errAck  : out std_logic;       -- (unused)
    AUS_retry   : out std_logic;       -- (unused)
    AUS_toutSup : out std_logic;       -- Timeout suppress
    AUS_xferAck : out std_logic;       -- Transfer acknowledge

    -- Interrupt
    Interrupt   : out std_logic;       -- Interrupt Signal

    -- Audio IO Signals
    AU_MCLK     : out std_logic;       -- Audio Chip Master Clock
    AU_LRCK     : out std_logic;       -- Audio Left/Right Channel Clock
    AU_BCLK     : out std_logic;       -- Audio Bit Clock
    AU_SDTI     : out std_logic;       -- Audio Data In (TO Codec)
    AU_SDTO0    : in  std_logic);      -- Audio Data Out 0 (FROM Codec)

end opb_audio_sampler;
```

```
architecture behavioral of opb_audio_sampler is

   -- Clock counter used to generate audio transmission clocks
   signal opb_clk_count : std_logic_vector(10 downto 0);

   -- Internal Utility Signals
   signal int_selected    : std_logic;    -- Decoded chip select signal
   signal int_aus_dbus_en : std_logic;    -- Enables output to AUS_DBus
   signal int_shift_out   : std_logic_vector(31 downto 0);    -- Output of shift-in register
   signal int_sample      : std_logic_vector(31 downto 0);    -- Last received sample

   -- Internal Buffer Signals
   signal int_interrupt : std_logic;
   signal int_au_mclk   : std_logic;
   signal int_au_bclk   : std_logic;
   signal int_au_lrck   : std_logic;
   signal int_au_sdto0  : std_logic;

   signal lrck_lat: std_logic;
   signal lrck_rising: std_logic;
   signal int_counter: std_logic_vector (0 to 2);

   -- State constants
   constant Idle : std_logic := '0';
   constant Xfer : std_logic := '1';

   signal sample_lat0      : std_logic_vector(31 downto 0);
   signal sample_lat1      : std_logic_vector(31 downto 0);
   signal sample_lat2      : std_logic_vector(31 downto 0);
   signal sample_out       : std_logic_vector(31 downto 0);
   signal sample_enable    : std_logic;
   signal sample_counter   : std_logic_vector(1 downto 0);
   signal present_state, next_state : std_logic;

   -- 32-bit Shift Left Register with Serial In and Parallel Out
   component shift_32 is
      port (
         SI : in  std_logic;
         CK: in  std_logic;
         C  : in  std_logic;
         D  : out std_logic_vector(31 downto 0));
   end component;

   -- 32-bit Latch with Asynchronous Reset
   component latch_32 is
      port (
```

```
        R : in  std_logic;
        C : in  std_logic;
        CK: in  std_logic;
        D : in  std_logic_vector(31 downto 0);
        Q : out std_logic_vector(31 downto 0));
    end component;

    -- Output buffer for FPGA outputs to audio codec
    component OBUF_F_8 is
      port (
        O : out std_ulogic;     -- Out to Audio codec
        I : in  std_ulogic);    -- In from Audio codec
    end component;

    component IBUF is
      port (
        I : in  std_logic;
        O : out std_logic);
    end component;

begin

    -- Receiving Shift Register
    shift_rcv : shift_32
        port map (SI => int_au_sdto0,
                  CK => OPB_Clk,
                  C  => int_au_bclk,
                  D  => int_shift_out);

    -- Sample Latch
    latch_sample : latch_32
        port map (R => OPB_Rst,
                  CK => OPB_Clk,
                  C => int_au_lrck,
                  D => int_shift_out,
                  Q => int_sample);

    -- AU_MCLK Buffer
    au_mclk_buff : OBUF_F_8
        port map (
          O => AU_MCLK,
          I => int_au_mclk);

    -- AU_BCLK Buffer
    au_bclk_buff : OBUF_F_8
        port map (
```

```
      O => AU_BCLK,
      I => int_au_bclk);

-- AU_LRCK Buffer
au_lrck_buff : OBUF_F_8
    port map (
      O => AU_LRCK,
      I => int_au_lrck);

-- AU_SDTI Buffer
au_sdti_buff : OBUF_F_8
    port map (
      O => AU_SDTI,
      I => int_au_sdto0);

-- AU_MCLK Buffer
au_sdto0_buff : IBUF
    port map (
      O => int_au_sdto0,
      I => AU_SDTO0);

-- Produce chip select signal by decoding OPB address and checking OPB_select
int_selected <= '1' when OPB_select = '1' and
                         OPB_ABus(0 to C_OPB_AWIDTH-9) =
                         C_BASEADDR(0 to C_OPB_AWIDTH-9)
                  else '0';


-- Tie unused OPB slave outputs
AUS_errAck <= '0';
AUS_retry <= '0';
AUS_toutSup <= '0';

-- Tie OPB ack to output enable signal
AUS_xferAck <= int_aus_dbus_en;

-- Tie unused audio output
--AU_SDTI <= '0';

-- Tie off-FPGA signals to internal buffer signals
INTERRUPT <= int_interrupt;

--int_au_sdto0 <= AU_SDTO0;

-- Generate Audio Transmission Clocks
int_au_mclk <= opb_clk_count(1);
```

```
    int_au_bclk <= opb_clk_count(4);
    int_au_lrck <= NOT opb_clk_count(9);

    opb_clk_count_proc : process(OPB_Clk, OPB_Rst)
    begin
       if OPB_Rst = '1' then
          opb_clk_count <= "00000000000";
       elsif OPB_Clk'event and OPB_Clk='1' then
          opb_clk_count <= opb_clk_count + 1;
       end if;
    end process opb_clk_count_proc;

    -- Gate output to AUS_DBus with int_aus_dbus_en
    gate_aus_dbus_output : process (OPB_Rst, OPB_RNW, int_aus_dbus_en)
    begin
       if OPB_Rst = '1' then
          AUS_DBus(0 to C_OPB_DWIDTH-1) <= (others => '0');
       else
         if int_aus_dbus_en = '1' then
           if ( OPB_ABus(29) = '1' ) then
             AUS_DBus(0 to C_OPB_DWIDTH-1) <= sample_out(23 downto 16) &
sample_out(31 downto 24) & sample_out(7 downto 0) & sample_out(15 downto 8);
           else
             AUS_DBus(0 to C_OPB_DWIDTH-1) <= sample_out(31 downto 0);
           end if;
         else
           AUS_DBus(0 to C_OPB_DWIDTH-1) <= (others => '0');
         end if;
       end if;
    end process gate_aus_dbus_output;

    -- Send interrupt pulse upon receiving sample
    -- SMALL HACK: Using int_au_mclk to reset the interrupt
--    send_interrupt : process (int_au_mclk, int_au_lrck, int_interrupt)
--    begin
--       if int_au_mclk='1' then
--          int_interrupt <= '0';
--       elsif int_au_lrck'event and int_au_lrck='1' then
--          int_interrupt <= '1';
--       end if;
--    end process send_interrupt;

   process(OPB_Clk, OPB_Rst)
    begin
       if OPB_Rst = '1' then
          lrck_lat <= '0';
```

```
    elsif OPB_Clk'event and OPB_Clk = '1' then
       lrck_lat <= int_au_lrck;
    end if;
end process;

lrck_rising <=   (not lrck_lat) and int_au_lrck;

process(OPB_Clk, OPB_Rst)
begin
    if OPB_Rst = '1' then
       int_interrupt <= '0';
    elsif OPB_Clk'event and OPB_Clk = '1' then
       if ( lrck_rising = '1' and sample_counter = "11")  then
          int_interrupt <= '1';
       elsif int_counter = "111" then
          int_interrupt <= '0';
       end if;
    end if;
end process;

process(OPB_Clk, OPB_Rst)
begin
    if OPB_Rst = '1' then
       int_counter <= "000";
    elsif OPB_Clk'event and OPB_Clk = '1' then
       if  lrck_rising = '1' then
          int_counter <= "000";
       else
          int_counter <= int_counter + '1';
       end if;
    end if;
end process;

process(OPB_Clk, OPB_Rst)
begin
    if OPB_Rst = '1' then
       sample_counter <= "00";
    elsif OPB_Clk'event and OPB_Clk = '1' then
       if  lrck_rising = '1' then
          sample_counter <= sample_counter + '1';
       end if;
    end if;
end process;


sample_enable <=  lrck_rising or int_aus_dbus_en;
```

```
sample_out <= sample_lat2;

process(OPB_Clk, OPB_Rst)
begin
  if ( OPB_Rst = '1') then
    sample_lat0 <= (others => '0');
    sample_lat1 <= (others => '0');
    sample_lat2 <= (others => '0');
  elsif (OPB_Clk 'event and OPB_Clk = '1') then
    if (sample_enable = '1') then
      sample_lat0 <= int_sample;
      sample_lat1 <= sample_lat0;
      sample_lat2 <= sample_lat1;
    end if;
  end if;
end process;

-- Sequential part of the FSM
fsm_seq : process(OPB_Clk, OPB_Rst)
begin
   if OPB_Rst = '1' then          -- Asynchronous reset to Idle state
      present_state <= Idle;
   elsif OPB_Clk'event and OPB_Clk = '1' then     -- Positive edge OPB_Clk
      present_state <= next_state;              -- advance to next state
   end if;
end process fsm_seq;

-- Combinational part of the FSM
fsm_comb : process (present_state, int_selected)
begin
   int_aus_dbus_en <= '0';

   case present_state is

      when Idle =>
         int_aus_dbus_en <= '0';
         if int_selected='1' then
            next_state <= Xfer;
         else
            next_state <= Idle;
         end if;

      when Xfer =>
         int_aus_dbus_en <= '1';
         next_state <= Idle;
```

```
          when others =>
             int_aus_dbus_en <= '0';
             next_state <= Idle;

       end case;
   end process fsm_comb;

end behavioral;
```

4.3.8 opb_xsb300e_vga.vhd

```
--------------------------------------------------------------------------------
--
-- Text-mode VGA controller for the XESS-300E
--
-- Uses an OPB interface, e.g., for use with the Microblaze soft core
--
-- Stephen A. Edwards
-- sedwards@cs.columbia.edu
--
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity opb_xsb300e_vga is

  generic (
    C_OPB_AWIDTH : integer              := 32;
    C_OPB_DWIDTH : integer              := 32;
    C_BASEADDR   : std_logic_vector(31 downto 0) := X"FEFF1000";
    C_HIGHADDR   : std_logic_vector(31 downto 0) := X"FEFF1FFF");

  port (
    OPB_Clk       : in std_logic;
    OPB_Rst       : in std_logic;

    -- OPB signals
    OPB_ABus      : in std_logic_vector (31 downto 0);
    OPB_BE        : in std_logic_vector (3 downto 0);
    OPB_DBus      : in std_logic_vector (31 downto 0);
    OPB_RNW       : in std_logic;
    OPB_select    : in std_logic;
    OPB_seqAddr   : in std_logic;
```

```
    VGA_DBus       : out std_logic_vector (31 downto 0);
    VGA_errAck     : out std_logic;
    VGA_retry      : out std_logic;
    VGA_toutSup    : out std_logic;
    VGA_xferAck    : out std_logic;

    Pixel_Clock    : in std_logic;

    -- Video signals
    VIDOUT_CLK     : out std_logic;
    VIDOUT_RED     : out std_logic_vector(9 downto 0);
    VIDOUT_GREEN   : out std_logic_vector(9 downto 0);
    VIDOUT_BLUE    : out std_logic_vector(9 downto 0);
    VIDOUT_BLANK_N : out std_logic;
    VIDOUT_HSYNC_N : out std_logic;
    VIDOUT_VSYNC_N : out std_logic);

end opb_xsb300e_vga;

architecture Behavioral of opb_xsb300e_vga is

  constant BASEADDR : std_logic_vector(31 downto 0) := X"FEFF1000";

  -- Video parameters

  constant HTOTAL : integer := 800;
  constant HSYNC : integer := 96;
  constant HBACK_PORCH : integer := 48;
  constant HACTIVE : integer := 640;
  constant HFRONT_PORCH : integer := 16;

  constant VTOTAL : integer := 525;
  constant VSYNC : integer := 2;
  constant VBACK_PORCH : integer := 33;
  constant VACTIVE : integer := 480;
  constant VFRONT_PORCH : integer := 10;

  -- 512 X 8 dual-ported Xilinx block RAM
  component RAMB4_S8_S8
    port (
      DOA   : out std_logic_vector (7 downto 0);
      ADDRA : in std_logic_vector (8 downto 0);
      CLKA  : in std_logic;
      DIA   : in std_logic_vector (7 downto 0);
      ENA   : in std_logic;
```

```
    RSTA  : in std_logic;
    WEA   : in std_logic;
    DOB   : out std_logic_vector (7 downto 0);
    ADDRB : in std_logic_vector (8 downto 0);
    CLKB  : in std_logic;
    DIB   : in std_logic_vector (7 downto 0);
    ENB   : in std_logic;
    RSTB  : in std_logic;
    WEB   : in std_logic);
end component;
```