CPEN311
University of British Columbia

Lab 2: Simple iPod (FSMs, Flash Memory and Keyboard)

In this lab we will build Finite State Machines (FSM) to read the Flash memory on the card. The Flash contains sound samples of a song that we need to move to the audio D/A (Digital to Analog converter) in order to listen to them. Additionally, we will learn how to interface with the keyboard. This is essentially the same thing that your iPod does - play songs from FLASH into an audio device, controlled by a keyboard (on a touchscreen).

Interfacing to electronic components, and in particular reading of a memory, are fundamental things in any hardware system, so this laboratory is extremely important and is quite representative of many things that hardware designers do in the real world.

Please read all these instructions before beginning.

Part 1: Programming the on-board Flash Memory

You first have to program the song onto the Flash memory of your board. This only needs to be done once, as it remain in the Flash memory, even after the board is powered off.

If you have a DE2:

1. Run the "DE2_ControlPanel.exe" application (that either came in the CD with your DE2 or you can download it from the course website under "Additional Files").

2. The application should program the DE2 board and then connect to it. This takes about 10-30 seconds.

3. Click the ‘Memory’ section, change the “Memory Type” to “FLASH (200000h WORDS, 4MB)”.

4. Click “Chip Erase (40 sec)” to erase the flash memory.

5. In the “Sequential Write” section, set ‘Address’ to 00000000, leave ‘Length’ alone, and check off ‘File Length’.

6. Click “Write a File to Memory”, browse to the “american_hero_song.hex” file, and the writing process will begin. This may take a few minutes.

7. After it is complete, power off then power on your DE2 board.

Troubleshooting:

Unfortunately the “DE2_ControlPanel.exe” application from Terasic is not very reliable. Sometimes it has problems connecting to the card and other times it has problems programming the flash. Your TA will try to help you with any problems during the programming process. Troubleshooting steps include:

1. If you get a card connection error, make sure you have the latest version of the program, which is 2.0.3, so download DE2_ControlPanel_v2.0.3.zip from the course website, in the "Additional Files" section of the table in the lab page.

2. If you get a programming error which is something similar to "hex file is invalid", try to program the file “american_hero_song.bin” (which is the same song just in binary file format)

3. It may be necessary to install Quartus 14.1 for this to work, but before you do that seek a TA's help.

If you have a DE1-SoC board:

1. Launch the Programmer tool.

2. Click ‘Auto-Detect’. If you get a popup box titled ‘Select Device’, then choose ‘5CSEMA5’.

3. If you get a popup asking “Do you want to update the Programmer’s device list, overwriting an existing settings?”, choose ‘Yes’.

4. Right-click the ‘5CSEMA5’ device and choose ‘Edit=>Change File”, and select the “song.jic” file.

5. A new entry will be added to the programmer list for the device “EPCS128”, which is the flash memory chip. For this entry, check the ‘Erase’ box and click ‘Start’. Wait for it to complete.

6. Next, for the same device check the “Program/Configure” box and click ‘Start’. The song will now be programmed into the flash memory.

Part 2: Taking a look at the solution and template and understanding what you need to do

1. Download the ".sof" solution for this lab from the website.

2. **Make sure the card is shut down.** Attach a keyboard to the PS/2 connection. Attach earphones or speakers to the "Line Out" connection of the card. **Always make sure that the card is turned off before connecting or disconnecting the keyboard!**

4. Now turn on the card and load the solution SOF file to the card using Quartus.

5. Select the "Information Console" mode of the LCD. That is, put SW[17] = 0. (There is no LCD for the DE1-SoC board, so skip to step 7).

6. Now press several keys. You will see the letters written on the LCD screen (in the top line). Remember that the LCD is updated every 1 second or so, so there may be a short delay until you see the letters on the LCD.

7. Connect speakers or headphones to the audio output of the card. Press the "E" and listen. To stop the music press "D". You can press "D" and "E" several times if you want to observe the effect.

8. Play with KEY0 and KEY1 on the FPGA card. Note how the speed of the song changes. When you press KEY2, the speed will return to normal speed.

9. Play with the "B" (for "Backwards") and "F" (for "Forward"). The song will be heard in the normal direction or vice versa, depending on the direction chosen.

10. When you press "R", the song will restart (if the direction of the song is "B" (backwards) you will hear the song from its end to its beginning).

11. What is happening is that there are audio samples within a Flash memory on the card. There is a FSM that reads these values and it sends them to the audio D/A to generate sound.

You have to design the content of the FPGA to comply with the behavior of the solution. To complete the lab, follow the steps below.

12. Download and open the template for this lab from the website.

13. Open the Quartus project in the template (the "QPF" file).

14. Please compile the project and verify that you can load the card.

15. Now press several keys. You will see the letters written on the LCD screen. Remember that the LCD is updated every 1 second so there may be a short delay until you see the letters on the LCD. (If you have a DE1-SoC, there is no LCD so you can skip this step; however when you write your code using the template, use lcd_scope.stp (provided) and SignalTap instead of the LCD - it behaves the same as in

Lab1. Using lcd_scope.stp you can see the letters from the keyboard and you can use this for debugging).

16. Open the file "simple_ipod_solution.v". Look inside this file Write_Kbd_To_Scope_LCD modules, Kbd_ctrl and key2ascii. These modules are already written and work well, no need to change them. However, I recommend that you enter the modules to see how they work. Overall, Kbd_ctrl manages the interface with the keyboard, and outputs the variable "kbd_scan_code", which is converted by the module key2ascii to an ASCII code (which is the same code of the parameters "character_A", "character_B" etc., that you already know). This ASCII code is used by the module Write_Kbd_To_Scope_LCD to write the letters in the LCD. The module Write_Kbd_To_Scope_LCD is interesting because it contains an FSM - try to understand how it works. To understand the interface with the keyboard, you can look at the DE2 or DE1-SoC manual and read chapter 9 in Chu's book.

Part 3: Adding your code to the template

Do the following:

1. **Preparation**:

   The sound samples are **16-bit samples**, and there are 0x200000 samples in the Flash memory. You have to read these values and get them to audio. To do this, you must first learn about the Flash memory and how it is connected to the FPGA. Do the following things:

   On the DE2 board, you will be interfacing directly with the Flash chip. So you should familiarize yourself with the operation of this chip:

   a. The Flash memory is of type S29GL032N whose datasheet S29GL032N90TFI040.pdf can be found under the datasheets subdirectory in the DE2. Look at the file S29GL032N90TFI040.pdf, especially sections 4, 5, Table 15.1 and Figure 15.2. In order to read data from the Flash, you need to generate waveforms like those in Figure 15.2.

b. Look at the DE2 Schematic (DE2-V22.pdf), especially page 19 of it, to see how the FLASH is connected.

c. Your FSM(s) have to handle the various interface signals to the FLASH: FL_DQ, FL_ADDR, FL_WE_N, FL_RST_N, FL_OE_N, and FL_CE_N.

d. Since we will only be reading from the FLASH, you can simply do the following in your code, so that you can concentrate on controlling the important signals:

```
assign FL_WE_N        = 1'b1;
assign FL_RST_N      =  1'b1;
```

On the DE1-SoC board, you will be interfacing with an Altera flash controller, which I have included in the project. You will have to learn how to interface with this controller.

a. The Flash controller is described in Section 39 of the "ug_embedded_ip.pdf" document, titled "Altera Generic Quad SPI Controller". Look at table 39-1, which describes the interface signals. Aside from Clock and Reset, there are two sets of signals – one for accessing control and status registers, and one for accessing the data of the flash. You should not need to access the control and status registers, so you can ignore these signals. You should read the descriptions about the "avl_mem_*" signals.

b. The "avl_mem_*" signals are part of an Avalon Memory Mapped bus interface. Avalon is a bus system created by Altera and used by many of the Altera IP cores. You can read about the protocol in the "mnl_avalon_spec.pdf" document. This document is quite long, so you should focus on Table 3-1 provides detailed describes of the different signals in the bus. In order to read from the Flash controller, you need to generate waveforms like those in Figure 3-5. Although the example shown includes pipelined reads (new addresses provided every cycle), your read operations will be more simple, and you should only initiate a single read request and then wait for the data to return.

c. Your FSM(s) have to handle the various interface signals to the Flash controller. The inputs to the controller are flash_mem_write, flash_mem_burstcount, flash_mem_read,

flash_mem_address, flash_mem_writedata, and flash_mem_byteenable. The outputs are flash_mem_waitrequest, flash_mem_readdata, and flash_mem_readdatavalid.

d. Since we will only be reading from the FLASH, you can simply do the following in your code, so that you can concentrate on controlling the important signals:

```
assign flash_mem_write = 1'b0;
assign flash_mem_writedata = 32'b0;
```

Also, since we will not be using the burst capabilities of the flash controller, you can set:

```
assign flash_mem_burstcount = 6'b000001;
```

2. **Implementation**: To implement the algorithm do the following things:

a. Write an FSM to cyclically read the Flash. **Use CLOCK_50 for the FSM clock**.

For DE2 users, notice that in Table 15.1 there are several times that correspond to waveforms in Figure 15.2. These are timing constraints that must be met (assume that the "Speed Option" is 110). Remember that **minimum** times are **minimum** times, if we wait a little more time, it doesn't hurt.

For DE1-SoC users, remember that Figure 3-5 shows the timing of a read operation. Although you don't have to wait a specified amount of time, you must observe both the waitrequest and readdatavalid signals.

b. The data in the FLASH is 16-bit audio data. The even addresses are the lower (least significant) bytes, whereas the odd addresses are the higher (most significant) bytes.

For DE2 users, the address signal is a byte index in the memory, and the data signal is 8 bits wide. So the first audio sample will be at address 1 and address 0, and the last sample will be at 0x1FFFFF and 0x1FFFFE.

For DE1-SoC users, the address signal is a 32-bit word index in the memory, and the data signal is 32 bits wide. So the first audio sample will be in the lower-half (i.e. bits 15:0) of the data at address 0, and the last sample will be in the upper half (i.e. bits 31:16) of the data at address 0x7FFFF.

c. You have to read a new value from the FLASH and send it to the audio every 0.045 milliseconds, i.e the sampling rate of the song, which you must reproduce, is 22,000Hz = 1/0.045 milliseconds. To do this, use a frequency divider to divide the 27 MHz clock which comes from the input pin TD_CLK27 in order to generate a frequency of 22KHz. Each rising edge of this 22KHz clock will serve as a stimulus for your FSM which tells it to read a new value from the Flash and give it to the audio D/A converter. Remember the 22KHz clock is not synchronized with the FSM clock of **50 MHz,** so in order to generate the stimulus for your FSMs, you have to detect the edge of the 22KHz clock with the methods we learned in class. There is a reason we are using the input pin TD_CLK27 as the clock for the frequency divider that generates the 22 KHz clock. This will provide a truly asynchronous 22KHz clock as compared to the 50 MHz FSM clock, and will expose setup/hold vulnerabilities that your design may have (before the TA finds them, so that is a good idea!). Had we generated the 22KHz clock as a division of the 50 MHz FSM clock, even though the 22KHz clock is a different clock domain than the 50 MHz clock, Quartus is usually smart enough to compensate for that and since the clocks are related even if your design is problematic it may work (but

not all tools are as smart as Quartus; it may fail in another manufacturer's devices). So using the TD_CLK27 pin is a good verification that your design is actually correct and handles a truly asynchronous stimulus correctly.

d. Use a counter controlled inside or outside your FSM, or you may even want to write another FSM simply to control the address that you send to the FLASH (i.e. you might want to separate the generation of the address that goes to the FLASH and the FLASH waveform control into two different FSMs that communicate among them, as we learned in class).

e. Construct an interface between your FSM(s) and the ASCII code received from the keyboard to control your music. When the user presses "D" the music stops, the music starts playing again when the user presses the key "E", the music should go backwards when the user presses "B" and go forward when the user presses "F". Implementing the "R" key is optional and can be used to a earn 5% bonus.

f. Use the KEY0, KEY1, and KEY2 to control audio speed. Do this by changing the frequency that generates the stimulus to the FSM(s) that reads data from the FLASH (normal frequency is 22 KHz. Pressing KEY0 will increase the speed, pressing KEY1 will decrease the speed, and KEY2 will reset the speed to the normal speed of 22KHz). I have already done part of the work for you. The speed_up_event, speed_down_event, and speed_reset_event signals already handles talking to KEY0 to KEY2 and give you three signals that you can use. It is your job to use them in order to control the speed of the song.

Specific grading requirements for this lab

To receive full points for simulation, you need to simulate all FSMs that you write, and demonstrate/explain those simulations to the TA. Include annotated screenshots of the simulations in your lab report.

10% of the functionality grade for this lab will be flow charts (in English) of your FSMs, which should be included in your lab report. This basically should be a 10 point gift to you. But please do not just make the flow charts to get 10 easy points after you finish the lab. Make the flow charts at the beginning of work on the FSMs to clarify in your mind the algorithm of the FSMs that you want to design, and then follow the design method shown in class. Do not just start to write an FSM without first drawing the flow chart - it might seem superfluous but especially when you are just starting to design FSMs this is more important than perhaps you realize.

To receive full points for the SignalTap component of the lab, see the video "Multiple SignalTap Instances" in the column "Additional Files" in the course website's Labs page. The SignalTap file must properly instrument your state machine(s) and you should be able to show during the demo, via SignalTap, how your state machine(s) operate. Make sure to also include in your lab report annotated screenshots of your SignalTap waveforms (not the "LCD scope" waveforms, but rather those waveforms that show your state machine(s) operating), which clearly show that your state machines are working properly.

Bonus Points

There are 5 bonus points available for implementing the "R" key functionality. You can get another 5 bonus points as follows: actually, the data in the lab is 8 bits sampled at 44 kHz (not 16 bits sampled at 22 KHz). After I made the whole lab and assumed 16-bits 22 KHz, I realized that actually the song was sampled at 44 KHz and was 8-bit samples, so instead of redoing the entire lab template,

solution, and instructions, I am leaving this as a bonus. You can get 5 bonus points by implementing the design at 44 KHz and shoving 8-bit samples out to the audio (so every 8 bits of data is a new sample). Note that when reproduced in this way the 44-KHz song should sound normal, not fast (think about it).

Final Tips

If in doubt about what to do, remember that you can always load the solution SOF and see what it does.

Do not forget the rules of good design:

- Always design while thinking about the **hardware** implementation
- Use **simple** structures
- **Incremental** design and test
- **Modular design**
- Use the **RTL viewer**
- Verification and test: Use **simulations**, LEDs, 7-segments, SignalTap, LCD scope (or SignalTap equivalent)
- Write **Clean**, **Neat**, and **Legible** code
- Give **meaningful** names to variables, use **comments**
- Go over and understand the warnings given by Quartus during compilation
- The circuit should be correct by **design**, not just by simulation.
- The code should be **verifiable** by **inspection**
- Design in a **modular** fashion, always thinking about future **reutilization** of the module.
- **Re-use** proven modules, instead of re-inventing the wheel all the time

Good luck and have fun!

CPEN311
University of British Columbia

Lab 3: Add a strength meter to the simple iPod (Embedded Processors, Signal Processing/ Basic Filtering (Averaging))

In this laboratory, you will enhance your "simple iPod" of the previous lab in order to add an LED strength meter that will show the strength of the audio signal. You will do this using an embedded PicoBlaze processor. This will serve as an example of how embedded processors are used in practice and will close the gap between hardware and software, i.e. between this course and many other courses in your curriculum. This will also be an example of Digital Signal Processing using an embedded processor (we will do real-time averaging, which is a basic form of filtering). This lab represents the final step in our journey from transistors to software.

Start off from you solution to Lab 2, and from the PicoBlaze in-class activity. You will have to essentially merge these two. There is hence no template for this laboratory. You **MUST** use the Picoblaze processor in the manner described below, even though probably you could do this lab purely using logic, since the point of this lab is to practice the usage of embedded processors. You need to do the following:

Starting off from your "iPod" from Lab 2, instantiate a PicoBlaze processor to do the following things:

i. Your main program will toggle LEDR[0] every 1 second (trivial).

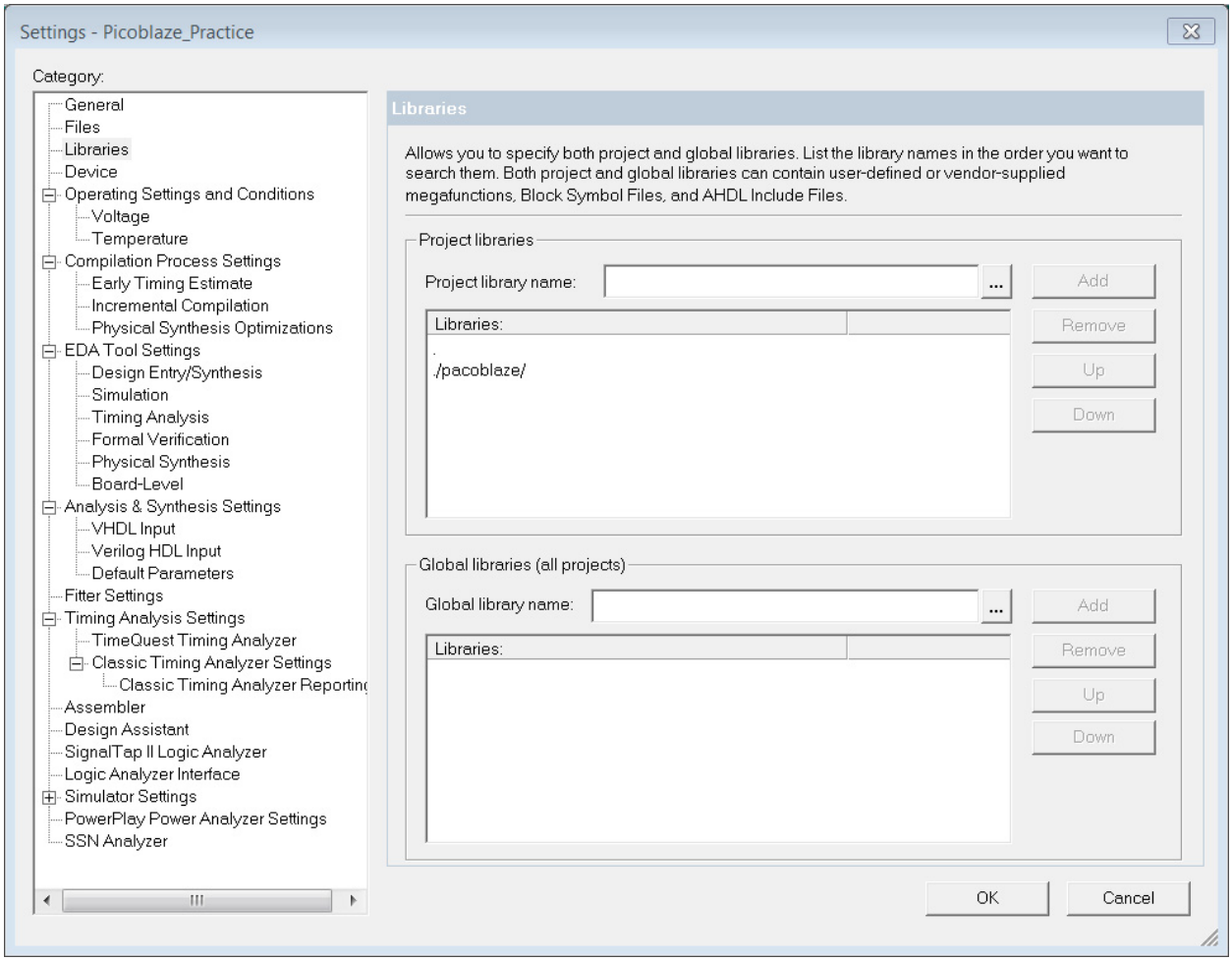
ii. The interrupt routine will be activated each time a new value is read from the Flash memory. Each value is a sound sample, each sample has its "intensity", or absolute value. The interrupt will accumulate (=sum) 256 of these absolute values (a value each time the interrupt is called), and the interrupt routine will divide this sum by 256 every 256-th interrupt. This is essentially an **averaging filter** operation, i.e. we are averaging every 256 absolute values of samples. Division by 256, because that is the power of 2, can be done very simply by discarding $\log_2(256)$ bits from the sum.

iii. Every time we do this division by 256, the PicoBlaze interrupt routine should output the average value to the LEDG[7:0]. (If you have a DE1-SoC, use LEDR[9:2]) Note that you have to "fill" the LEDs from left to

right, i.e. make the LEDs light up to the value of the most significant binary digit of the average. For example, if the average of the absolute values is, in binary, 00101101, then since the highest bit that is "1" is bit #5 (where bit #0 is the LSB), the LEDs should be XXXXXX00 (where "X" is on and "0" is off). As always, look at what the solution does if you have any doubts.

iv. After each averaged value is output to the appropriate LEDs, the accumulator is set to 0 to prepare to average the next 256 values, and so on.

Note: When merging the lab2 template with the in-class activity, do not add all the "pacoblaze" files to the project, rather make sure that the pacoblaze directory is in the project search path (see image below) and Quartus will automatically find the files. The reason for this is that the pacoblaze directory contains some testbench files that are not synthesizeable and will cause errors in Quartus if included in the project. Furthermore, make sure to merge the Picoblaze in-class activity into the Lab2 template, not the other way around. The Lab2 template contains pins and device assignments and compiler assignments that you must keep for Lab3 to work properly.



Specific grading requirements for this lab

In this lab, you do not need to submit or demo simulations or SignalTap. The grade for functionality will incorporate the points for simulation and SignalTap. You should however, for your own benefit, simulate any modules that you write for this lab, and use SignalTap when necessary, as a matter of good practice.

If you do not use the Picoblaze/Pacoblaze for this lab, you will receive 0% in the lab.

If in doubt about what to do, remember that you can always load the solution SOF and see what it does.

Do not forget the rules of good design:

- Always design while thinking about the **hardware** implementation
- Use **simple** structures
- **Incremental** design and test
- **Modular design**
- Use the **RTL viewer**
- Verification and test: Use **simulations**, LEDs, 7-segments, SignalTap, LCD scope (or SignalTap equivalent)
- Write **Clean**, **Neat**, and **Legible** code
- Give **meaningful** names to variables, use **comments**
- Go over and understand the warnings given by Quartus during compilation
- The circuit should be correct by **design**, not just by simulation.
- The code should be **verifiable** by **inspection**
- Design in a **modular** fashion, always thinking about future **reutilization** of the module.
- **Re-use** proven modules, instead of re-inventing the wheel all the time

Good luck and have fun!