ECE 3710 – Fall 2014
Instructor: Dr. Myers
T.A: Trevor Hill

Project Documentation

By Andy Gilbert, Calli Clark, Tom Becnel, and Amandeep Gill

This document explains the design and implementation of our project titled “Audio Jacked”. A Spartan 6 Field Programmable Gate Array (FPGA) board was programmed for an audio filter application via a touchscreen interface. This document is broken down into the following sections:

1. **ISA –** description of instruction set architecture
2. **Datapath –** description of processor datapath
3. **Memory –** description of memory layout with memory map diagram
4. **Controller –** description of processor controller with a finite state machine (FSM) diagram
5. **I/O –** description of all external inputs and outputs and their protocols
6. **Application code –** description of application code and structure of the software
7. **Assembler Users Guide –** complete guide to run instructions or application code through our assembler
8. **Application User’s Guide –** complete guide for the user to run our audio filter application including I/O interface instructions

1. ISA

The design of our ISA closely followed that of the RISC processor with 4 bit op codes and 4 bit destination and sources registers. We included all baseline instructions and highly recommended instructions expect ADDUI and ADDCI. Because our application code was anticipated to be simple, mainly a series of load and store instructions to collect data from a microphone and output it to a speaker, we concluded it was not necessary to ever utilize an add immediate unsigned or with carry. In addition, we decided to make our instructions 32 bits to compensate for the large amount of data we anticipated storing from a microphone into memory. The following tables show our instruction encoding. The different instruction types included: R-type, I-type, Special, load/store immediate, set conditional, jump conditional, jump conditional immediate, and branch. The jump and link (JAL) instruction has bits 27:24 designated for Rlink, the return address. We designed this register to always be 15. The most important aspect of this instruction encoding is the 24 bit immediate value in the I-type instruction. This design allows registers to hold large values, thus large memory addresses.

R-Type Instruction				
Instruction [31:28]	Instruction [27:24]	Instruction [23:20]	Instruction [19:16]	Instruction [15:0]
OpCode [0000]	Dst Reg	Function	Src Reg	Shamt

I-Type Instruction		
Instruction [31:28]	Instruction [27:24]	Instruction [23:0]
OpCode	Dst Reg	Immediate Value

Special Instruction				
Instruction [31:28]	Instruction [27:24]	Instruction [23:20]	Instruction [19:16]	Instruction[15:0]
OpCode [0100]	Rdst	Special Type/cond	Raddr/Rsrc	Unused

Load/Store Immediate			
Instruction [31:28]	Instruction [27:24]	Instruction [23:16]	Instruction [15:0]
OpCode	Rdst/Rsrc	immediate	-

Scond Instruction					
Instruction [31:28]	Instruction [27:24]	Instruction [23:20]	Instruction [19:16]	Instruction [15:12]	Instruction [11:0]
OpCode [0100]	Rdst	1101	-	cond	-

Branch Condition Instruction				
Instruction [31:28]	Instruction [27:24]	Instruction [23:20]	Instruction [19:16]	Instruction[15:0]
OpCode [1100]	R1	Condition	R2	Displacement

Jump Condition Instruction					
Instruction [31:28]	Instruction [27:24]	Instruction [23:20]	Instruction [19:16]	Instruction [15:12]	Instruction [11:0]
OpCode [0100]	-	1100	Rtarget	cond	-

Jump Condition Immediate Instruction					
Instruction [31:28]	Instruction [27:24]	Instruction [23:20]	Instruction [19:16]	Instruction [15:12]	Instruction [11:0]
OpCode [0100]	R1	1001	R2	cond	immediate

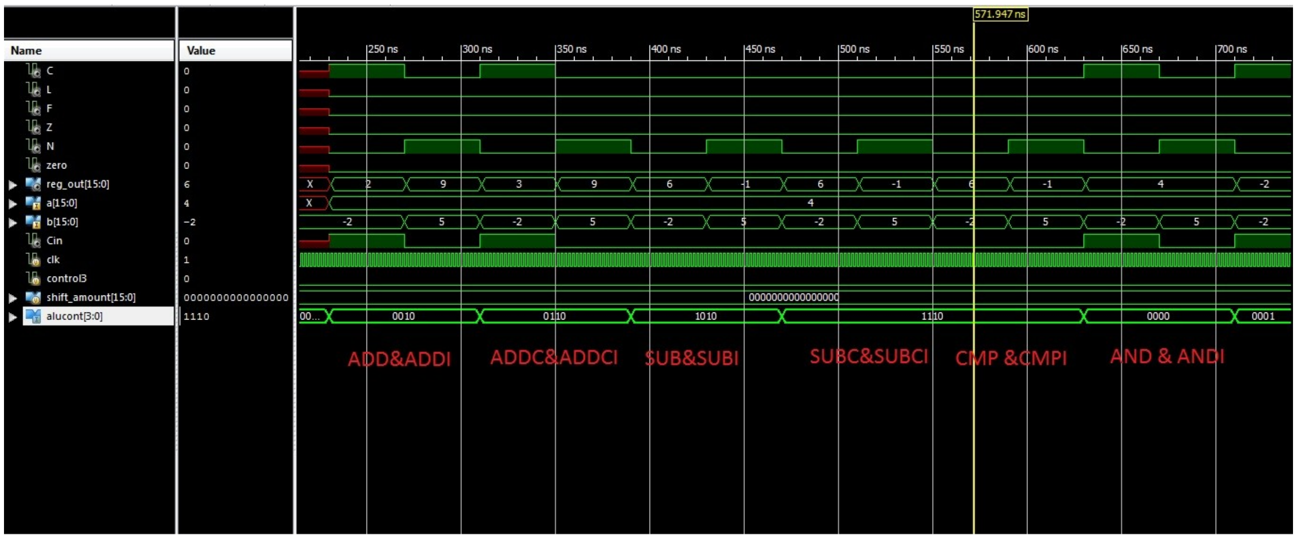
The instruction op codes were also very similar that of the RISC architecture (excluding the nonessential instructions, and ADDUI/ADDCI). Jump register (JR) was also added to the special type instructions.

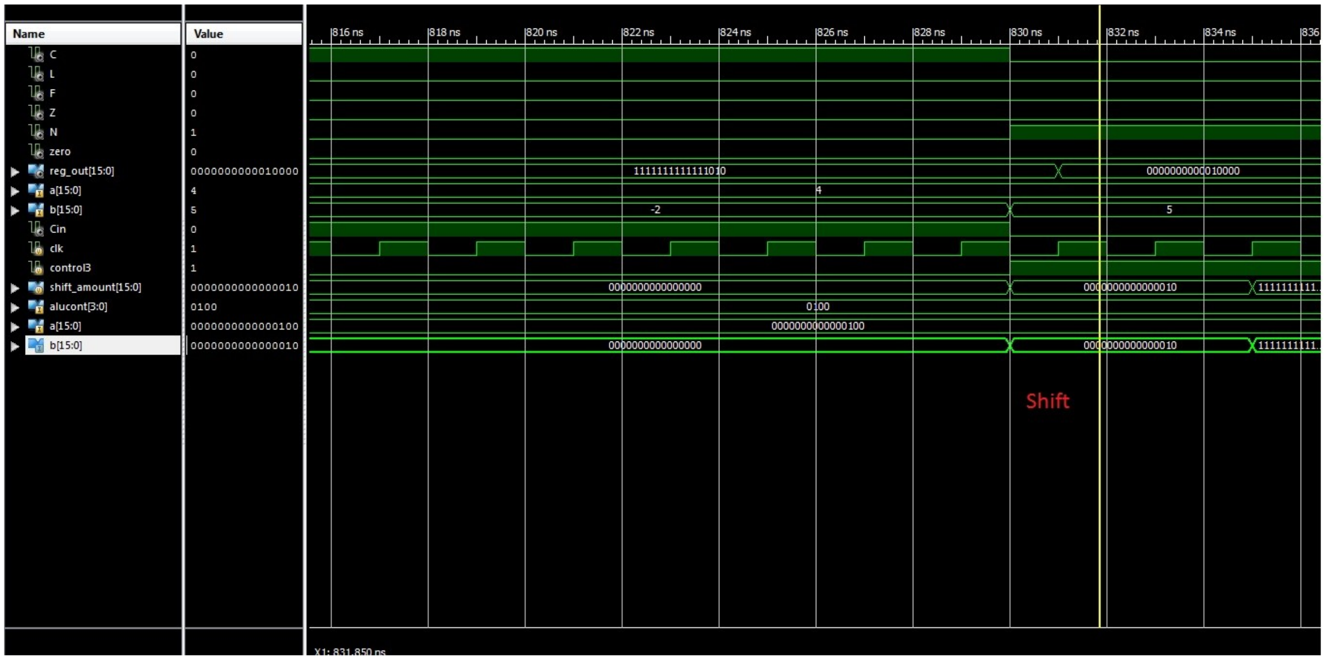
ALU

Our ALU and connected datapath consisted of a register file containing 16 32-bit registers, a central ALU which controlled the arithmetic, a shifter and an ALU output register. Two multiplexors were used to choose between the output from the ALU or shifter. The logical functions are controlled through the ALU controller. We utilized a total of 7 functions: add, multiply, subtract, compare, AND, OR, and XOR. The compare function does not output a value to be used in future computations but rather sets flags by comparing two registers. These flags are then used in conditional instructions and control the datapath. The flags we set are C, L, F, Z, and N. C is set high if there is a carry bit during addition. L is the ‘Low’ Flag, and is set HIGH when the source register is less than the destination register (for unsigned). F is the overflow flag, and is set according to overflow. This flag is not taken into account when doing any unsigned arithmetic, since it is only necessary when doing signed operations to make sure the sign is still correct. The flag Z is set when two registers are equal to each other. This is necessary for branch instructions. Flag N is set when the destination register destination operand is less than the source register only when they are both signed integers. This flag is needed for branch and jump conditional statements.

Our shifter utilized a logical and arithmetic shift left – determined from the controller. We implemented it in such a way the right shifts are achieved by the use of a negative number.

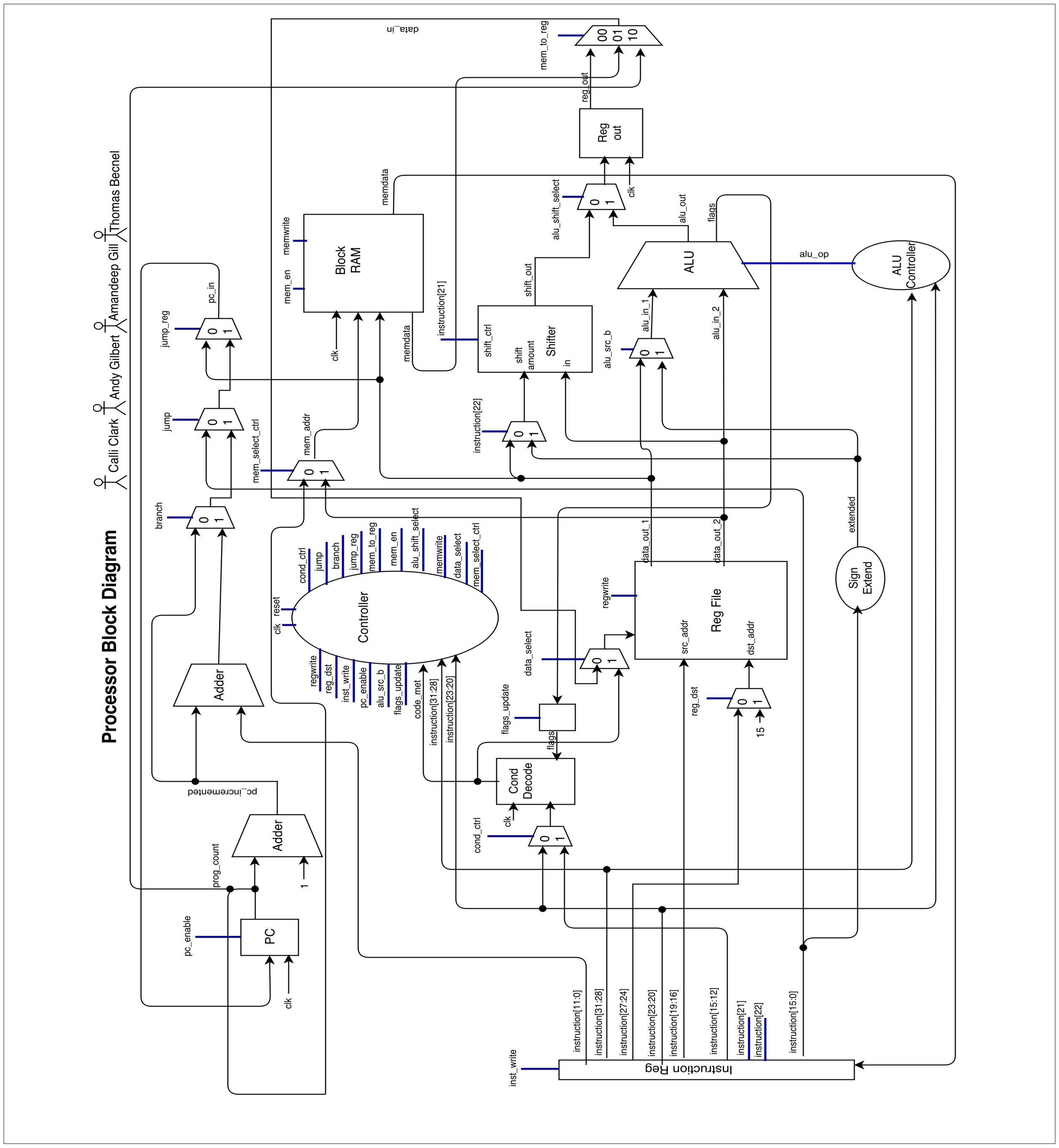
To test that our ALU worked we wrote a testbench that performed some operations and checked to make sure the flags were what they should be. Certain instructions such as jump and branch were not tested explicitly because if the flags were working then it follows that the ALU controller for jump and branch instructions would work as well. Functions tested explicitly were: ADD(I), SUB(I), SUBC(I), SUBU(I), AND(I), XOR(I), OR(I), LSH(I), and ASHU(I). The following waveforms show this simulated testbench.





2. **Datapath**

The data path that we designed had one central top module with most other modules instantiated inside. However, the data path was divided into several sections. These included: I/O, Memory, Controller, Condition Decode, Program Counter, Register, ALU and Shifter. These sections then interfaced with each other. Each of these sections is described in more detail below. The overall datapath is shown in the figure below.



I/O

This section included modules that controlled each of our I/O modules as well as an controller module to interface these with the processor. The first module was our microphone module which took in 12 bits of data at a time from the microphone and was controlled via a input start signal and an output done signal. The second module was our speaker module which controlled the speaker with I2S protocol. It took in 16 bits of data at a time and was controlled via a done signal which went high when it finished the current 16 bits. There was also a buffer module to hold this done signal high until the application had acknowledged it. There was also a module to interface with the VMOD Touch Screen. This module took in touch inputs from the screen and sent that to a menu controller module which decided which menu was currently active. It would then output this value to a glyph control module which would combine this with the current horizontal and vertical count to determine which glyph was currently active. A RGB Decode module then took the glyph address and sent it to the memory to get RGB values and output them to the screen. Finally, we had a I/O controller module which interfaced between the processor and memory (block ram and cellram), switches, leds, the speaker, and the microphone. Each I/O had a specific memory address associated with it so the processor would control these pieces by sending a load or store command to the address associated with the I/O it wanted to control. For more information on this please see the memory map section.

Memory

This section included the Block RAM module, the module to control Cellular RAM, various logic for signals in these modules, the instruction register, and a multiplexer to decide where the memory address should come from. Our Block RAM had 10 bits of address space with 32 bit data so it was 4096 kBytes large. The first segment was used to store our instruction memory, the next part was used for glyphs, while the last part was open for the processor to use. Meanwhile, our Cellular RAM had 23 bits of address space, but only stored 16 bit data so it had 16 Mbytes of space. This was an asynchronous controller, but required a delay of about 7 or 8 clock cycles (depending on whether it was a read or write command) to perform an operation. The processor would go into a stall state while these commands were being executed. The instruction register was hooked directly to Block RAM, but had a control signal so it would only update at the appropriate times. The multiplexer selects where the memory address comes from. It can either come from the program count (for gettting the next instruction), the registers (for loading from or storing to an address in a register), an immediate value (for loading or storing from an address in the instruction), or the VGA module (when we are setting the VGA values).

Controller

This section included the main controller which set the main control signals for all of the multiplexers, enabled registers, and other modules in the datapath. It also included two counters. The first ensured that the processor would do nothing for 20000 clock cycles leading up to the initialization to allow the cell ram time to turn on before the processor tried to access it. The second ensured that the processor did not try to access the cellular ram two times within 40 clock cycles. This was to give cell ram enough time in between reads and writes.

Condition Decode

This section was used to determine if the condition was met for the instructions that were conditional. It took the output flags from the ALU and compared them to the conditional code to determine whether or not the condition was met. This section also included an enabled register. This was used to keep the flags the same for those set of instructions that would compare a condition in the first instruction and then act on the result in the next one.

Program Counter

The program counter that we designed was based on our entire system being 32 bits. The PC is incremented by one to grab the next instruction. We implemented a separate adder for the program counter rather than going through the ALU. The new value of the PC then passes through the add offset which is used for our branch instruction. The lower 16 bits of the instruction (our offset) is sign extended and then is either added or subtracted from the PC. Both of these values are passed into a mux to decide what should be the new PC. After the branch option, there is a jump option. The output of the first mux and the last 24 bits of our instruction (memory address of the jump) are muxed together. This is controlled by a signal from our controller. The last mux is for the jump register. The first input is the output of the second mux and the second input is a register that holds the value of the PC before the jump (from JAL). The PC feeds into instruction memory to grab the next instruction. The Program count section contains these adders and muxes as well as an enabled register to hold the current program count and a sign extender module. The PC register is enabled so that it will only be updated once every instruction cycle. The sign extension module is separate from the sign extender used in the ALU section because in this case we only want to sign extend the last 16 bits of the instruction (the section used for the branch value). The sign extension module takes in one input, and gives one output. The input is the first 16 bits (0-15) of the instruction. The 16th bit is going to be extended 16 bits to the left to create a 32 bit number. The only place where this occurs in our data path is for the second input into the add offset function for branch instructions.

Register File

This section contains our registers, and a mux to determine where the input to the register file should come from. The register file is just 16 32-bit registers that have read asynchronously with an enabled synchronous write. This mux picks which data is passed to the registers. It can either come from the alu or shift (in the case of a R Type or I Type instruction) or I/O data (in the case of a load or store), or the program count (if we are jumping and linking), or code met (if we are storing the result of the condition check).

ALU and Shifter

This section contains the ALU, the shifter, an ALU controller, and various muxes to determine where the inputs to the ALU and shifter should come from. The shifter can either take in a register value or an immediate value and shifts by a given immediate value from the instruction. It can either perform an Arithmetic shift or a Logical Shift. The ALU controller decodes the instruction into an operation which is passed to the ALU. The ALU will act on this instruction. The ALU can add, multiply, subtract, AND, OR, XOR, or compare. It will compare no matter what the operation is and output the results into the flags. This is described in greater depth in the ISA section. One of the ALU inputs needs to come from a register while the other one can come from either a register (R-Type instruction) or an sign-extended immediate (I-Type instruction). In this case the sign extension module takes in 24 bits and sign extends this an additional 8 bits to put out a 32 bit value. There is one final mux to decide whether the output should come from the ALU or shifter. Once again, all of these mux control signals come from the controller.

3. Memory

We used two separate memories Block RAM and Cellular RAM. Cellular RAM was added because we needed more memory for our audio recordings. We were sampling at 16 bits at 48 kHz per second and were recording for up to 20 seconds which requires 1.92 Mbytes. Our Block Ram was only 4096 kBytes and thus could not handle this data. However, Block Ram was used to store our instructions as well as our VGA. Cellular RAM was only used for our recording space.

For our memory we had 24 bits of address space. This was because since we had 32 bit instructions we could use 24 bits to assign a value to a register with an add immediate instruction. More than this could have been achieved with various add and multiply instructions, but 24 bits was all that was necessary. The top 8 bits were then used to differentiate between our different I/O components. If the top bit was a 1 then it went to cellular RAM. This was because the other 23 bits were necessary to address within cellular RAM. If the top bit was a 0 then the next 7 bits were used. See the following table for more information. If all 8 bits were 0s then the data was stored/loaded in Block RAM and the bottom 10 bits of the address were used to address within Block RAM.

Name	**Memory Address (top 8 bits)**	**Notes:**
	Input/Output (Memory)	
Cellular RAM	8'b1xxxxxxx	If the top bit was a 1 then the data was automatically stored/loaded in cell ram regardless of what the rest of the bits were
Block RAM	8'b00000000	If all bits were zeros then the data was loaded/stored in block ram and the bottom 10 bits of the address were used to address within Block RAM
	Inputs	
Speaker Done	8'b01000001	This was used to tell the application if the speaker was done playing the current data.
Microphone	8'b00100000	This was used to load the current mic data in from the microphone module

Mic Done	8'b00100001	This was used to tell the application that the mic had finished collecting a full sample.
Menu and Filters	8'b00001000	This was used to tell the application the current menu and the current active filter (if in the 3rd menu)
Switches	8'b00000010	This was used to tell the application the current switch settings.
	Outputs	
Speaker	8'b01000000	This was used to set up the next speaker value in the buffer
Speaker Buffer	8'b01001000	This was used by the application to set the value actually going to the speaker
Speaker Reset	8'b01001001	This was used by the application to signify that the speaker value had been updated
LEDs	8'b00010000	This was used by the application the set the LEDs.

To allow easy and intuitive access to these components via the application two new commands were put in place. These were load immediate (li) and store immediate (si) which took as inputs the register that they were loading to or storing from and and 8 bit binary value which indicated the correct component. This was used for all I/O except Block RAM and Cell RAM which required the full 24 bits.

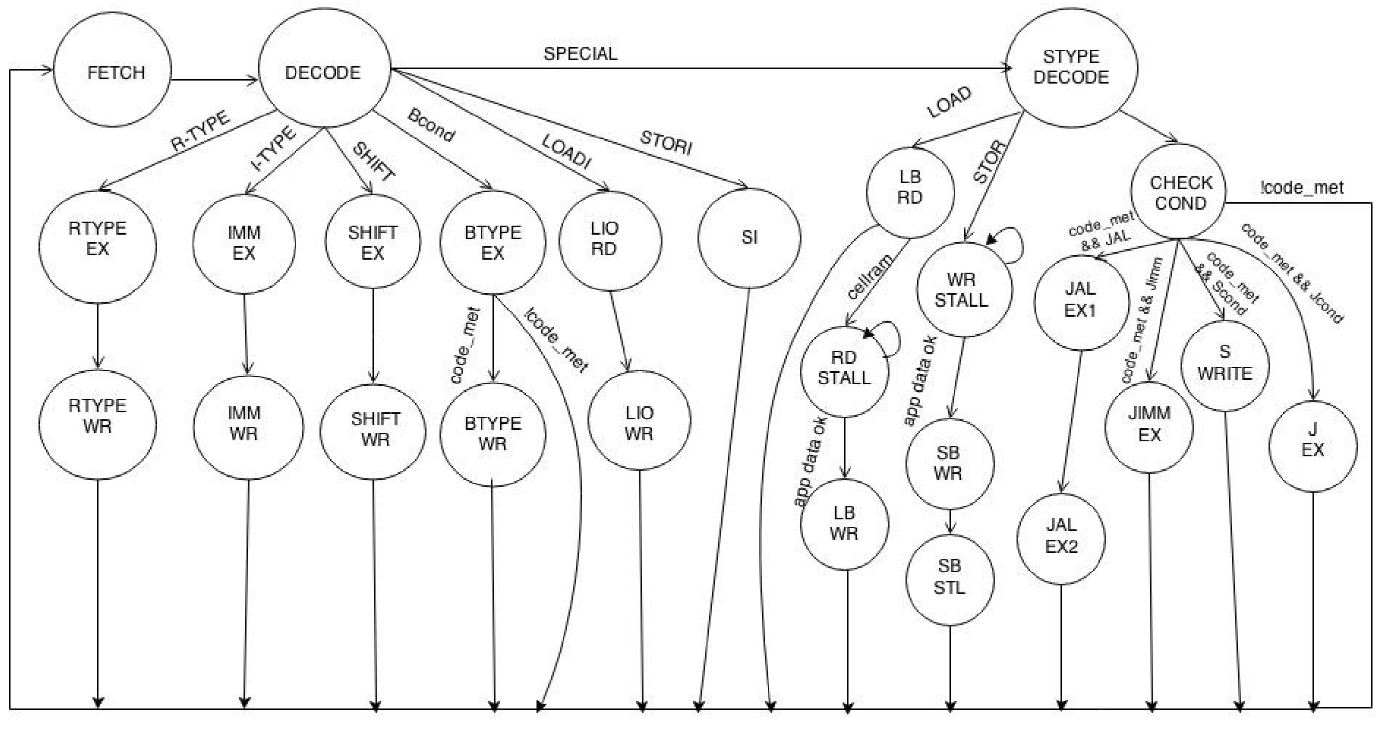
4. Controller

The controller was an essential part of our processor design because it allowed the proper timing and flow of the entire processor. It was important for control signals to be set in the correct states to ensure expected behavior. The FSM diagram below shows how we handled instruction decoding in our processor. The first state is used to fetch the appropriate instruction. Then the instruction is passed into a decoder which determines from the op code (4 MSB) which operation is going to take place. Based on this input the next state could either be R-type execution (RTYPE EX), I-type execution (IMM EX), shift (SHIFT EX), branch (BTYPE EX), load immediate (LIO RD), store immediate (STORI), or special type decode (STYPE DECODE).

R-type execution state is meant for register operations. Thus, two registers are used, one as a source register and one as a destination register. Both registers will be inputs to the ALU and the output of the ALU will be stored in the destination register. This register will then be stored in memory. For the I-type execution state, a destination reg and an immediate value are passed into the ALU. This output is then stored in the destination reg, which is then stored in memory in the I-type write state (IMM WR). The shift execution state shifts a register by a certain amount that is indicated by the instruction. Once this shift is completed, the value is then stored back in ot the original register and then is written in to memory in the shift write (SHIFT WR) state. Branch execution state contains the offset to determine how far down or up the PC will shift and if the conditional code is met from our flags, this new PC value is written in the branch write state (BTYPE WR). Load immediate instructions utilized two states for reading a value from memory and writing it into a register. Store immediate utilized one state to store the specified immediate value into a memory address. These instructions were added specifically to load and store from and to I/O.

The special type decode state determined whether a load, store, jump and link (JAL), set conditional, jump conditional or jump immediate conditional would be carried. For the special conditional instructions, a state was used to check the condition. If the condition was met, the execution and write states were moved to. If not, the next instruction would be fetched. Jump and link has two states JAL ex1 and JAL ex2. JAL ex1 takes the current program counter and stores that in register 15 and the second execution jumps to the specified address, updating the program counter. Load and store instructions used a read stall (for load) and write stall (for store) to ensure the cellular RAM data was ready. In addition, the load operation has two states LBRD (load byte read) and LBWR (load byte write). The LBRD reads a value from memory where LBWR stores that value into a register. The store operation is the reversal of this operation, SBRD (store byte read) reads a value from a register and SBWR (store byte write) stores this value into memory. Both of the above instructions know which registers are needed for the reads and writes because the registers are provided in the instruction.

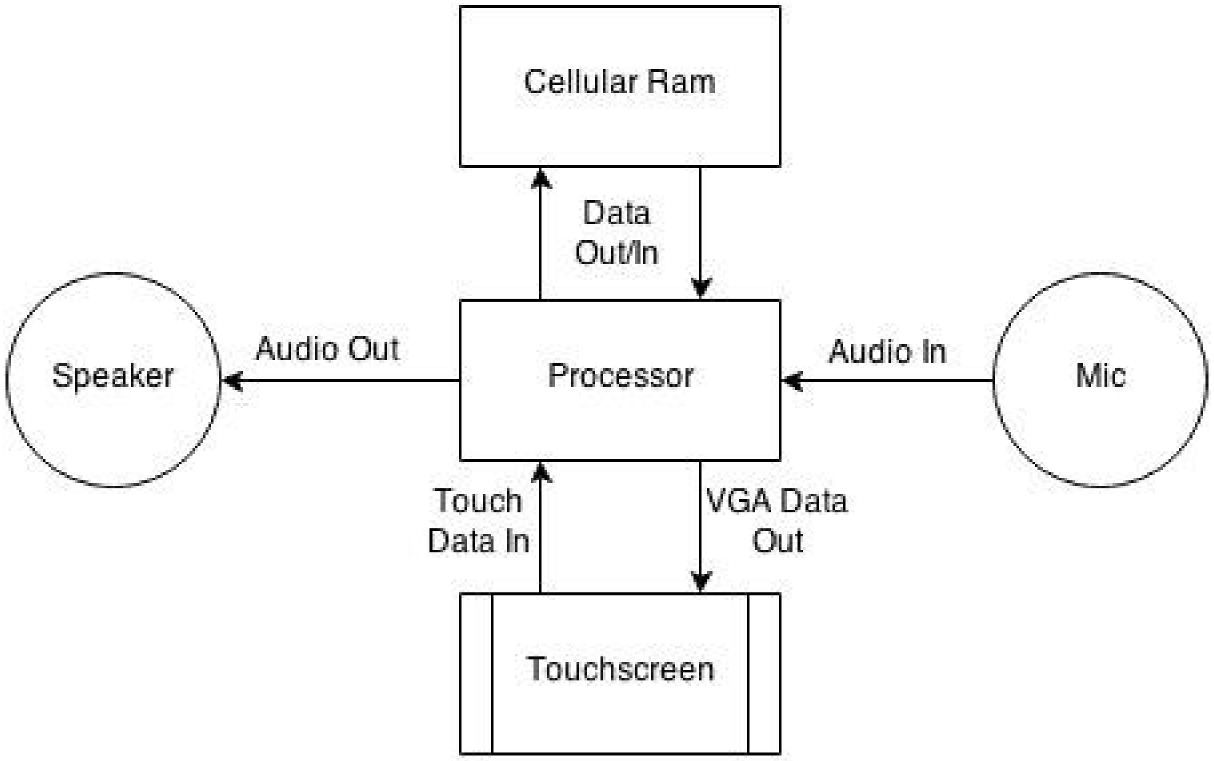
FSM



The controller used five inputs for the process of decoding instructions: clock, reset, a signal from the conditional decoder (code_met), the four bit op-code (bits [31:28]), and a four bit instruction that is used by special types and branch instructions (bits [23:20]). The states of the FSM can be seen in the above diagram. Also, it depicts the changes in between states and what requirements are needed to move to the next state. If nothing is listed as a transition, then the FSM moves to the next state on the next clock edge. Control signals were assigned to these states.

5. I/O

Our project used 5 different inputs and outputs. These were a microphone input, speaker output, touch input, VGA output, and Cellular RAM inputs and outputs. This is shown in the figure below. Each of our I/O components is discussed in more detail below.



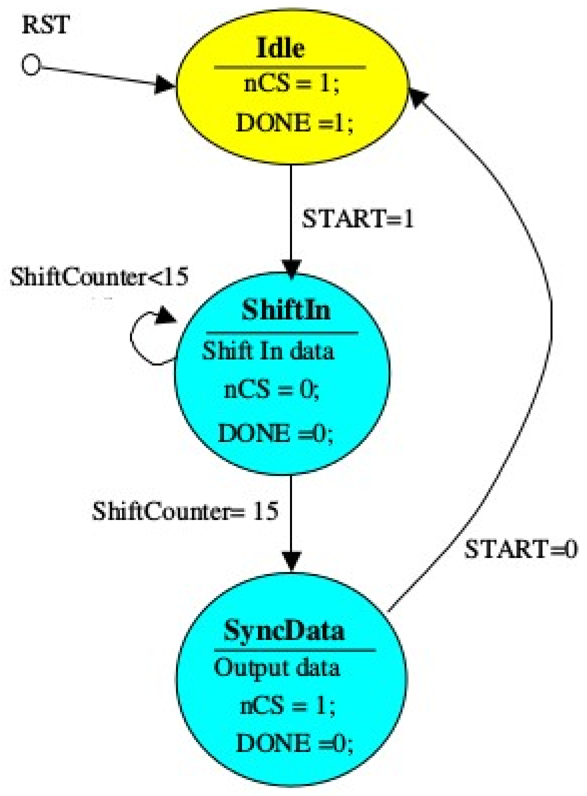
Microphone

We selected a Digilent Pmod Microphone for this project. This microphone was selected because it could easily interface with our FPGA Board. The microphone communicated with the board serially and with the board in packets of 12 bits. The signals for the microphone are shown in the table below.

Name	I/O	Size	Function
CLK	I	1	The system clock

RST	I	1	Universal reset
SDATA	I	1	The serial data from the microphone
SCLK	O	1	Clock sent to the chips on the board
NCS	O	1	Chip select, enables the chips on the board. Must be low when the data is generated. Immediately after data is generated this is driven high.
DATA	O	12	Sent to processor, the most recent data
START	I	1	Used to tell the microphone to start recording
DONE	O	1	Used to tell the processor to stop recording

The microphone uses the FSM in the following figure to control the microphone via the start and done signals. Start is set high to start the recording. The module then serially shifts in 16 bits of data. However, the four leading bits are always zeros. The start signal must then be driven low again before the module will go back to the Idle state and signify that it is done. Therefore the processor can interact with the microphone just via the START and DONE signals and does not need to actually control the chips.



All of these signals went to the I/O controller module so they could be interfaced with and accessed via the application. However, there were some shortcuts put in so every signal didn't have to be set. If a store was attempted to cellular RAM then the I/O controller would automatically set START high. To start START low again the processor would store to the mic address (the actual value being stored didn't matter since it wasn't used). The processor could then load from the mic done address to see if the mic had completed operation. Therefore a typical recording sequence would have the following form:

1. Load from mic – Get the microphone data into a register
2. Store to cellular RAM – Store the newly received data into memory.
3. Store to the microphone – Set START low again in preparation for the next loop
4. Check the value of DONE – If it is high then go to step 1
5. Else go back to step 4.

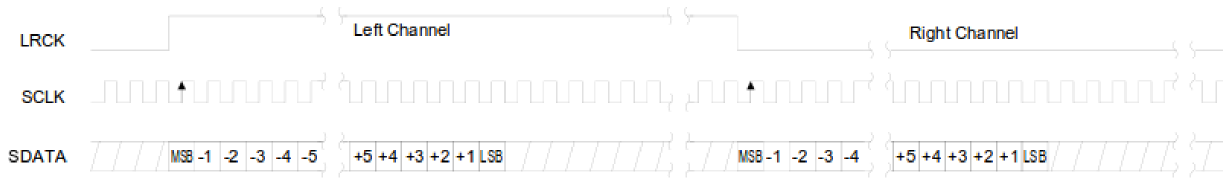
Of course, the very first time through this loop there will be no mic data because START has not been set high yet. However, for every successive iteration of the loop there will be a value ready to store.

Speaker

We again chose to use a Digilent audio jack in our project because we knew it would interface well with the board. The module that we chose required I2S protocol to talk to the speaker chip. The signals for the speaker are shown in the table below.

Name	**I/O**	**Size**	**Function**
CLK	I	1	Master clock
DATA_L	I	16	Data to left speaker (signed)
DATA_R	I	16	Data to right speaker (signed)
Accepted	O	1	Strobes when data_l and data_r are latched. Acts as DONE signal for speaker
I2S_SD	O	1	Serial data to chip
I2S_LRCLK	O	1	Decides whether data is going to left or right speaker
I2S_SCLK	O	1	Clock that serial data is output on
I2S_MCLK	O	1	Master clock to chip

In our case we wanted the same data going to the left and right speakers so DATA_L and DATA_R were both hooked to one common DATA input. It is important to note that this data must be in two's complement signed form. Based on the data this module will set the serial data high or low. The timing for this sequence is shown in the figure below.



Taken from CS4344 Datasheet:
http://www.cirrus.com/en/pubs/proDatasheet/CS4344-45-48_F2.pdf

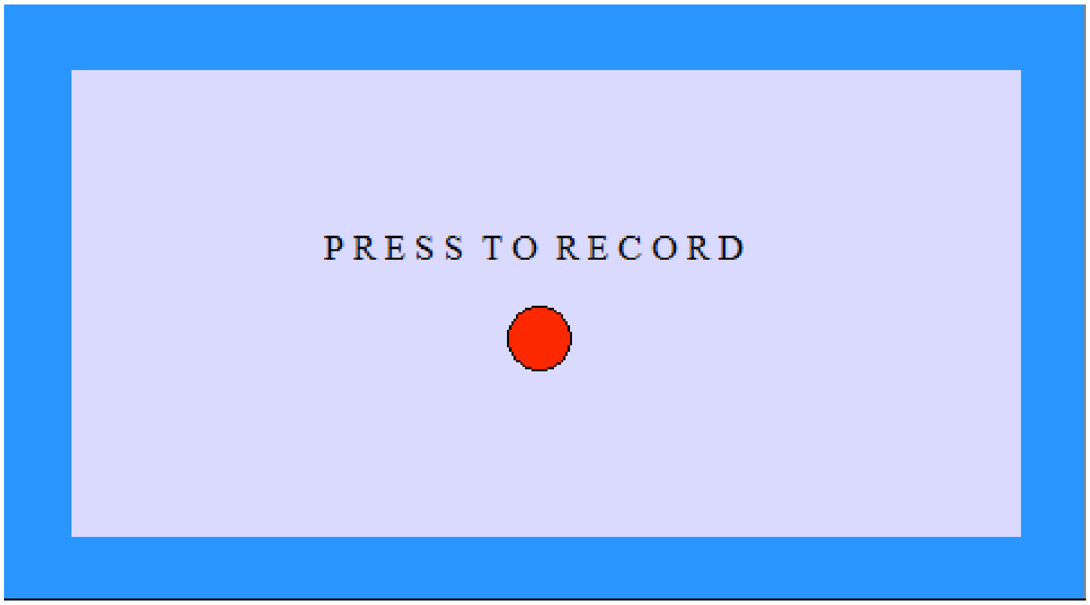
It is important to note that the data is shifted in on the negative edge of SCLK. This occurs so that the data can correctly be sampled on the positive edge of the SCLK. After that the processor can interact with the speaker in much the same way as the microphone except that no start signal is required. However, an additional module is required to hold the DONE signal high until the processor has acknowledged that it has been received because this module will only hold it high for one clock cycle. Therefore a buffer module was designed to do this. Speaker operation is also slightly more complicated because there is a delay in receiving data from cellular RAM. However, we want data to be instantly available whenever the speaker is ready for new data. Therefore a speaker buffer register was implemented to hold the data ready for the speaker. As soon as that data was sent to the speaker the processor requested a new value from Cellular RAM. Again some shortcuts were implemented in the load and store commands to the speaker so that signals that would be set anyway did not need multiple instructions. Therefore, speaker operation should use the following steps:

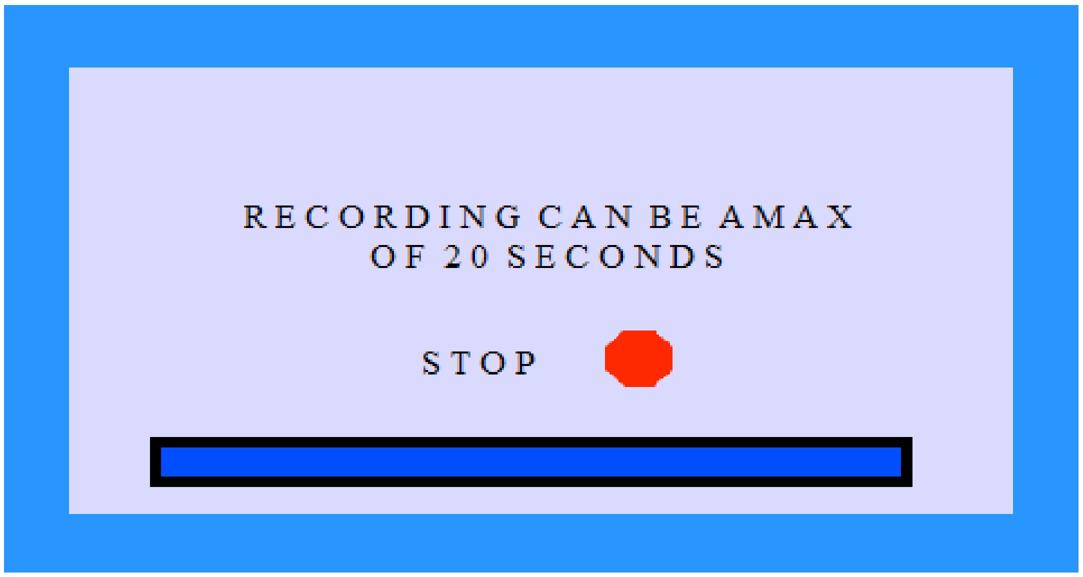
1. Store to the speaker – This has the effect of getting the current value stored in the buffer register and sending it to the speaker. It also sends an acknowledgement signal signal to the buffer module telling it that the DONE signal has been received and can be set low again.
2. Load data from the next cellram address – Gets the next speaker data
3. Store this data into the speaker buffer – Gets the next speaker data ready for playback. This command also resets the acknowledgement signal to the buffer low again.
4. Check if speaker update is high again – This means that the speaker is ready for new data. Go back to step 1.
5. Else go back to step 4.

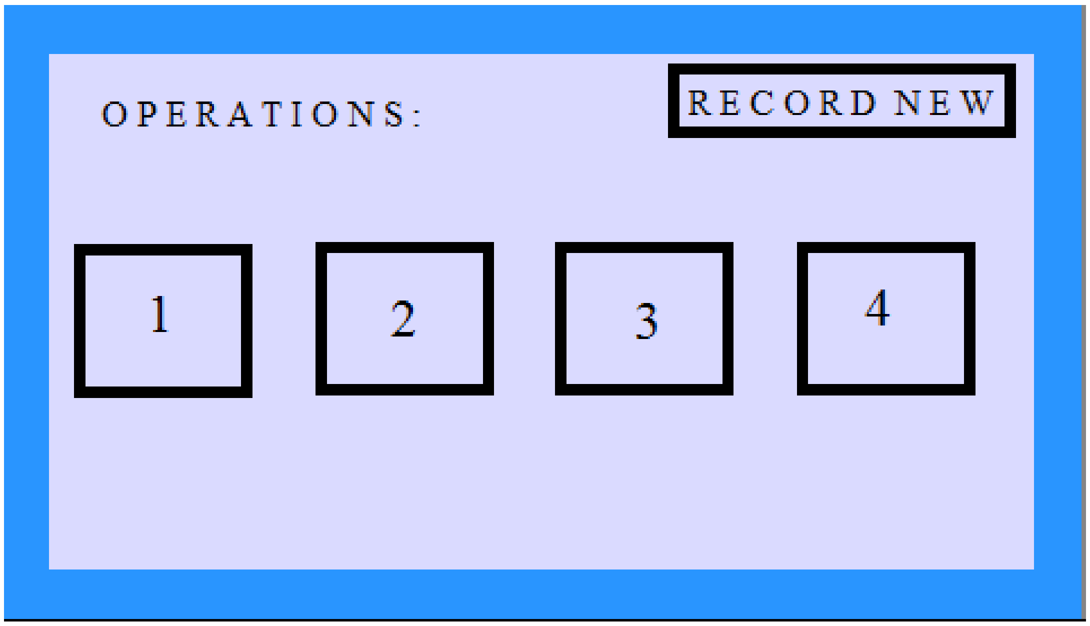
Again, the first time through the loop there will be no value in speaker buffer to load into the speaker. However, every other time through the loop a value will be ready.

VGA and Touch

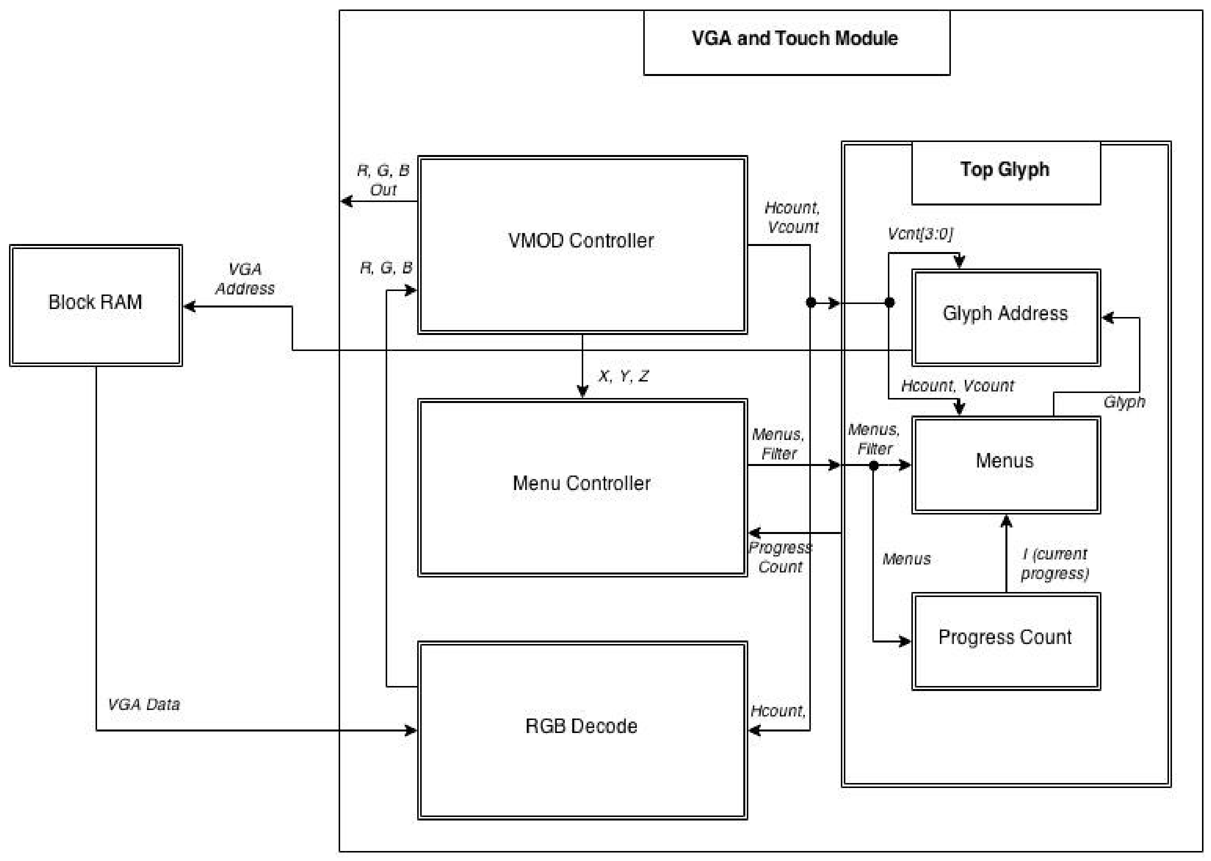
For our VGA and Touchscreen components we again chose a Digilent Board because we knew that it could interface with our board. The VGA and Touch components were stored in one common module. These components were handled exclusively by hardware and were not touched by our application at all. This was implemented because we only had 3 different menus. Additionally, we wanted to ensure that our processor was not too busy handling VGA and Touch inputs that it did not have time to correctly record and playback audio. The VGA design for our applications consists of three menus. The first menu is where the recording from a microphone will first be captured. This progresses to a second menu which shows the recording in progress. The third menu offers four different operations to be performed on the recording. A new recording can be captured when finished by pushing a button that loops back to first menu. The menus progress by using a 480x272 resolution touchscreen, pressing in the allocated button locations. The menu designs can be seen below:




RECORDING CAN BE AMAX
OF 20 SECONDS



The design for our VGA and Touch module can be seen in the figure below.



For our VGA design we used 16x16 glyphs. This meant that the top 6 bits of Hcount and Vcount were used to determine which glyph we were in while the bottom 4 bits were used to determine where we were within each glyph. Our VGA design was interesting because we still used glyphs despite designing it in hardware. For example, for each menu the screen was hard-coded in the form of what glyph needed to go where. These glyphs were then stored in Block RAM and accessed via an address from the RGB module. In order to not interfere with the processor the VGA controller was only allowed to access Block RAM during the decode state when it was guaranteed that the processor would not need it. Therefore the RGB decode module had two different VGA buffers. It would read the current RGB values from the current buffer using the bottom bits of Hcount. Meanwhile it would be filling in the other buffer during each decode state. When it reached the end of one buffer it would switch to the other one and the process would repeat.

The VMOD Controller was the main module that received the serial data from the touchscreen and output the RGB values to the screen. The X, Y, and Z were taken from here (Z was used to indicate how hard the screen was being pressed) coordinates were outputted to the Menu Controller module. The Menu Controller took these inputs and output a value that indicated what the current Menu was. It is important to note that a menu change required that Z was decreased and then increased again (the screen was touched and then released). This increased the reliability of our application. The Menus module took the current Hcount and Vcount from the VMOD Controller module as well as the current menus and filters. It would then output the correct glyph based on a combination of these. This glyph was sent to the glyph address module. The glyph address module used the glyph and the current Vcount to output an address to the Block RAM. This was not too difficult since we had 32 bits in each Block RAM address and 2 bits per pixel which meant that each address stored 1 row of 1 glyph of pixel information. In the diagram this is oversimplified as this is of course not the only input to the Block RAM address but actually goes through a mux. The Block RAM would then output the current RGB values to the RGB decode module. Since we only used 4 different colors each pixel was saved using only 2 bits. However, our screen was capable of 24 bit color meaning that these 2 bits had to be transformed into 8 bits each for R, G, and B. These RGB value were then sent back into the VMOD Controller which would output them to the screen. The Progress Count module was used to set the current value of the progress bar during the second menu.

6. Application code

The application code was written to playback filtered audio based on touch input from the user. To do this we had our I/O controller send out a series of signals and load them into specified memory address to identify if the user was recording, in playback mode or idling. The addresses and their values were used in the application code to branch to the correct label. We used what menu we were in as the control signal. Menu 1 indicates we are waiting for the user to record; we are idling, nothing should be playing back or getting loaded at this point. Menu 2 indicates the user is recording, we want the microphone data stored in cellular RAM. Note we are only storing every 40th microphone sample because the microphone rate is 40 times faster than the speaker rate. This was written as follows:

```
firstmenu:  and r13 zero         # filler
            and r4 zero          # reset values
            and r10 zero
            addi r4 8388608
            li r6 00001000        # check value of menus
            beq r6 r1 recording  # if the menu changes to 2 go to recording
            j firstmenu           # else check again

recording:        and r13 zero
                  li r8 00100000        # get the value of the mic
                  addi r10 1            #increment number of samples
                  beq r10 r11 store     # only store every 40th value
donestoring:      and r13 zero
                  si r12 00100000       #r12 should be irrelevant
                  li r6 00001000        # check value of menus
                  beq r6 r2 playback    # if the menu changes go to playback
hold:             and r13 zero
                  li r7 00100001        # get the value of mic done
                  beq r7 r1 recording
                  j hold
```

Menu 3 indicates the mic data has been stored and we're ready to playback to a speaker. Before there is any output to the speaker though, the user has to press which filtered audio they want to hear. The value of the control signal 'filter' (0 – 4) is concatenated with the value of menu to branch to the correct label in the assembly code (refer to the application code below).

```
playback:   and r13 zero
            li r6 00001000           # get value of menu and filter
            and r12 zero             # set r12 to zero for first comparison
            eq r6 r12 firstmenu      # go back to first menu if menu is 0n
            addi r12 6               # set to 6 for next comparison
            beq r6 r12 filteruno     # if first filter go there
            addi r12 4               # set to 10 for next comparison
            beq r6 r12 filterdos     # go to second filter if true
            addi r12 4               # set to 14 for next comparison
            beq r6 r12 filtertres    # go to third filter if true
            addi r12 4               # set to 18 for next comparison
            beq r6 r12 filterquatro # go to fourth filter if true
            and r14 zero            # reset leds
            addi r14 16             # set leds value
            si r14 00000100         # update leds
            j playback              # else do the checks again
```

Here is where the filter operations are computed and output to the speaker. For filter 1 – original playback – the data from memory was stored directly onto the speaker. For filter 2 – fast playback – the value in every other memory address was output to the speaker to only grab half the samples and speed up the audio. We also implemented different rates other than double for fast playback. For filter 3 – slow playback – the value in the same memory address was output to the speaker twice before moving to the next address as to slow the audio down. Filter 4 – reverse playback – started at the very last address of the recording and decremented the address value to store the reverse recording to the speaker.

A reset label at the very end of the application code was used to stop outputting audio to the speaker, reset the register storing the current memory address to the starting memory address and go check to see if another filter was pressed by the user.

7. Assembler User Guide

1. Download 'Assembler.jar' file and put into new folder
2. Write assembly code using format described in 'Assembly Documentation'
3. Save assembly code with '.txt' extension and save in same folder as 'Assembler.jar'
4. Run and compile assembly code to create machine code and hexidecimal represenatation file of machine code
   - I. Open Command Prompt or Terminal
   - II. 'cd' into directory containing 'Assembler.jar' and assembly code
   - III. Type following line into Command Prompt/Terminal to run Assembler:

```
java –jar Assembler.jar
```

   - IV. At this point user will prompted to input names of assembly code, desired name for machine code file, and desired name for the hexidecimal representation file. DO NOT include '.txt' or other extension names here. Assembler automatically adds '.txt' to file. Machine code and hexidecimal representation files will be created or overwritten in the same directory.

NOTE:

Output to Command Prompt/Terminal will be numbered lines of assembly code. If there is no error, the last two lines will say that the two files were created successfully. If there is an error, the last line of assembly code that was able to compile in Assembler will be the last line of output to CP/Terminal. The line proceeding this will be the line that includes the error.

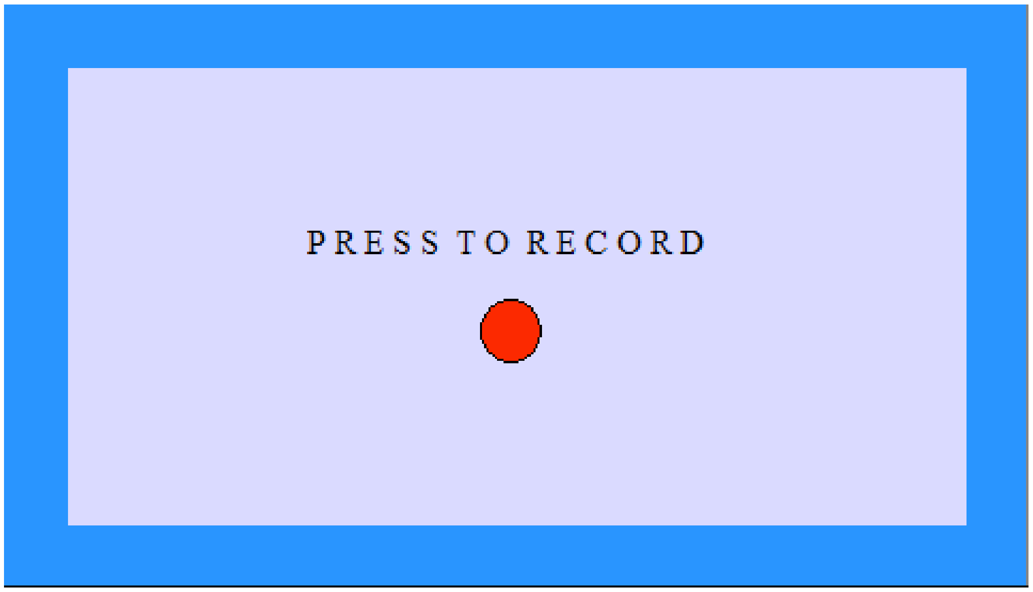
If CP/Terminal says that machine code was written but the hexidecimal code could not be written, then one of the lines of machine code was not 32 bits. Go into the machine code file andfind the line number that isn't 32 bits or has a 'null' value in it. Match that line number with the line number from the output into CP/Terminal to find the line with the error.

8. Application User's Guide:

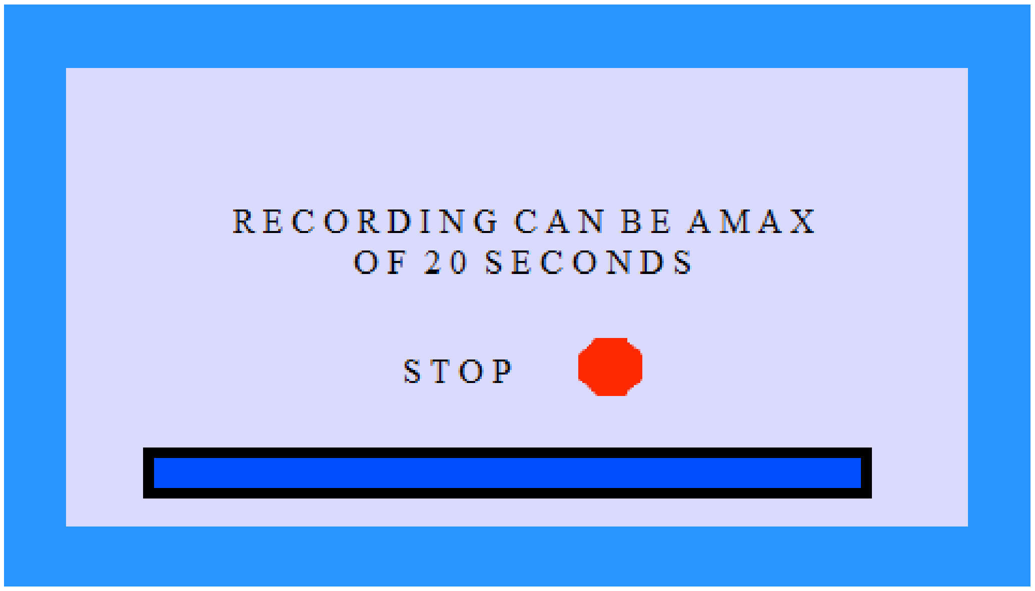
This audio filter application consists of a series of three menus displayed through a touchscreen. Up to 20 seconds of an audio recording can be stored using a microphone. Once the audio is recording, there are four playback features: play original recording, play recording sped up (x2), play recording slowed down (x2), and play recording in reverse.

The touchscreen is pressure sensitive and advances to the next screen when pressed firmly in the correct location. The microphone connected to PMOD connector J1 on the FPGA board is used to record any audio from the user. The audio jack connected to PMOD connector J2 on the FPGA board can be connected to a headphone or speaker audio jack to hear the filtered audio. The following are steps to record and audio playback:

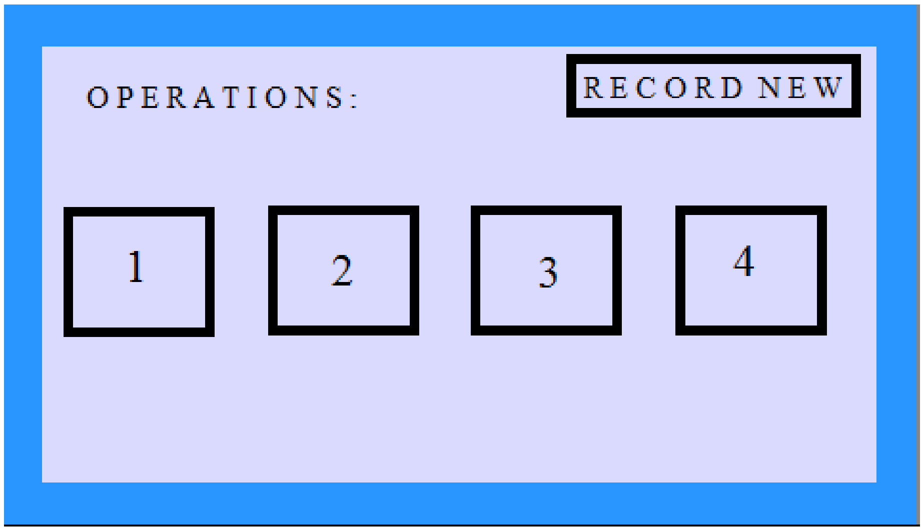
1. To start a recording, press the red button in the center of the screen. Make sure you press firmly and as close to the button as possible or the recording will not start. Refer to the image below.



2. The record button press advances to the next menu displaying the text "recording can be a max of 20 seconds" and a progress bar on a 20 second timer. Begin recording into the microphone. Any audio sample can be used in playback. Press the stop button firmly when you are done with the recording to advance to the next menu or wait until the progress bar finishes and the menu will automatically be updated. Refer to the image below for this second menu.



3. The third menu displays four boxes with the numbers 1 through 4 on them. These are the operations to hear the playback or filtered recording. Option number 1 plays the original recording, option number 2 plays a sped up version, option number 3 plays a slowed down version, and option number 4 plays the recording in reverse. Press any one of these buttons to hear the desired audio effect. A new recording can be started by pressing the button “record new” in the top right section of the screen and repeat steps 1-3. Refer to the image below for the third menu.



Important note: Once a playback operation is pressed in the third menu, the full recording must be played before a new filter operation can be pressed. The fifth LED on the FPGA board will turn green to indicate it is ready for a new filter operation.