The Music Project - Final Report

The Music Group

Tristan Wentworth, Alex Jarnutowski, Dayton Lindsay, Ben Holzem, Ilhan Bok, and Sam Bitter

University of Wisconsin-Madison

University of Wisconsin-Madison, Spring 2020

Table of Contents

1. Overview and Motivation

The goal of this project is to build an audio harmonizer using the DE1-SoC FPGA board by Altera. We will use either the board's line in or microphone in lines to play a note, voice, or other music to the board. We will then process the audio signal and output the harmonized note through the board's line out. Using the switches on the board we can configure the number of harmonized octaves and chords of those octaves included in the output. To achieve this goal we will be implementing a custom processor and additional hardware control blocks. We will also build a compiler for our pseudo-language FindTune, as well as an assembler to convert into machine level instructions.

Beyond achieving basic functionality of our assembler, compiler, and audio processor, we have two additional software goals. First, we will use the VGA on the board to display harmonized sound waves to the monitor. Second, we will develop AutoTune software to adjust input notes to equal intended notes and to output the modified sound. Feasibility of both of these goals will be based on time remaining after basic functionality is implemented.

This project was chosen because it gave our group the direction we felt would be the most interesting to create a whole architecture around. Beyond that the project involved pieces that we recognized from our everyday life, but were unfamiliar with the process behind which they worked. It also would have been extremely fun to play with had we been able to use it on the board.

2. High-Level Architecture

Our hardware is divided into a number of modules, the top level of which is called "main". Outside of the CPU we have an I2C Controller, audio input buffer, and audio output buffer. Within our CPU, we have five pipelined stages: Fetch, Decode, Execute, Memory, Write-Back. The CPU also includes modules that operate outside of the pipelined stages including the Branching unit, Forwarding unit, Hazard detection unit, and memory. We will also interface with the CODEC, which already exists on our DE1-SoC board. All units colored black will be located on the FPGA chip.

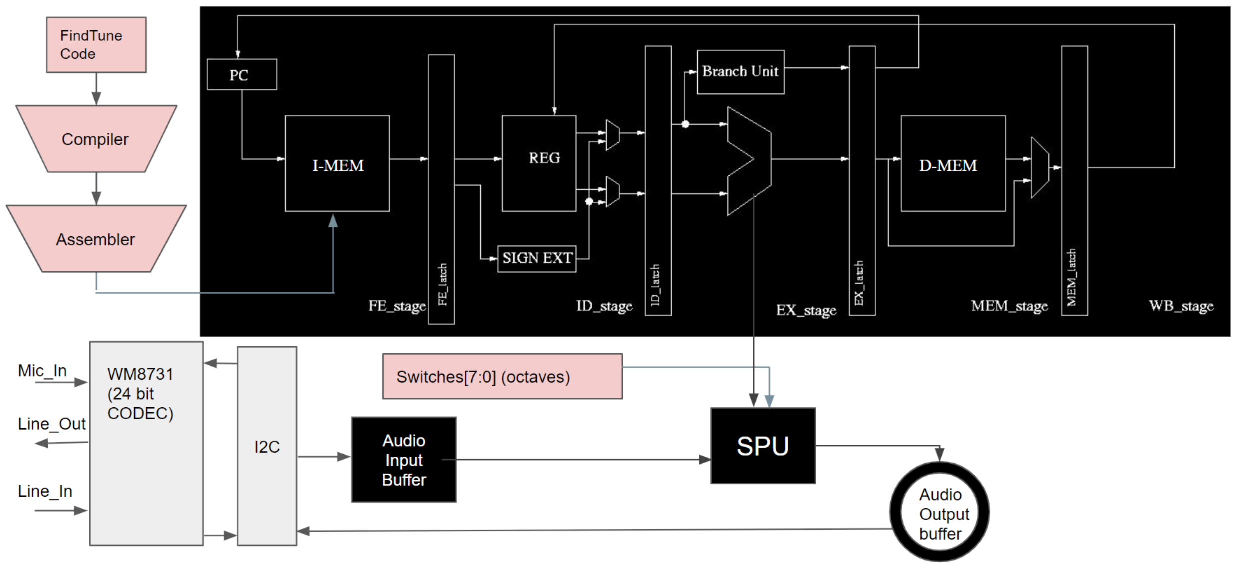


Figure 1: System level diagram of The Music Group's processor and external hardware modules.

2.1. I2C Controller

The I2C controller will communicate with the CODEC on the FPGA via I2C bus. This controller will fetch new audio data from the CODEC and send modified audio content back to the CODEC after it has been processed by our CPU. The received audio data will be sent to an audio input buffer, where the CPU can acquire new audio samples. The I2C controller will get the processed audio samples from an audio output buffer. When the I2C controller receives a signal from the audio output buffer alerting new processed data, it will prepare to send that data back to the CODEC. Our I2C controller will be programmed to operate the CODEC at 48kHz.

2.2. Audio Input & Output Buffer

The Audio Input Buffer will take in audio data from the I2C controller and hold it until and the data is requested by the CPU. The Audio Output Buffer will take in processed data from the CPU and output it to the I2C controller. It will also output a signal to alert the I2C Controller to new processed data. Both buffers will operate as first in, first out buffers.

2.3. Signal Processing Unit (SPU)

The SPU interfaces with the audio input and output buffer, reading in 16-bit sound samples and shifting their pitch up or down by several octaves producing a harmonized sound. The algorithms used to accomplish this task include PSOLA and STFT/ISTFT computation and interpolation, both of which are detailed in the CPU microarchitecture section of this document.

2.4. CPU

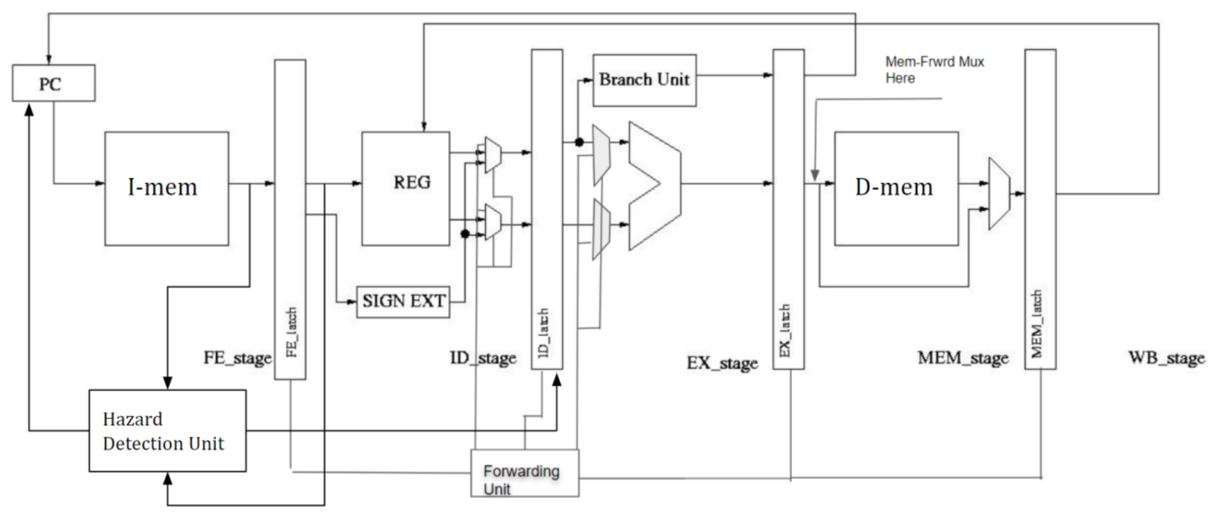


Figure 2. Block Diagram of CPU

In our project, we designed a five-stage pipelined processor with data forwarding and branch prediction. Above is a diagram of the microarchitecture in our current design iteration.

2.4.1. Registers

The register file will hold 8 16-bit registers. Registers will operate as write-before-read registers. Register descriptions are as follows:

- ❏ R0 will be immutable and hold the value zero
- ❏ R1 will be the Program Counter and hold the address of the next instruction
- ❏ R7 will be a link register and hold the address of the expected instruction for branch prediction
- ❏ R2-R6 will be 5 free registers used for short term storage of audio input and output data and computation

3. Instruction Set Architecture

Opcode	Instruction	Explanation
10000	**LD, R1, R2**	**Loads the value from R2 into R1**
10001	**LDR, R1, [R2]**	**Load value from memory at the address stored in R2[15:0]. For specific registers, we will instead interface with the audio input/output buffer**
01000	**LL, R1, imm**	**Load 8 bit immediate into the bits 0-7 of R1**
01001	**LH, R1, imm**	**Loads 8 bit immediate into bits 16-23 of R1**
10010	**ST, R1, [R2]**	**Store value in R1 into the address specified by the value in R2**
00001	**BRE**	**Branches if the equal flag is set**
00010	**BRG**	**Branches if the pos flag is set (R1 or imm is higher than R1)**
01010	**CMP, R1, imm**	**Compares R1 with imm and sets flags (imm - R1)** **R1 > imm => Neg Flag** **R1 < imm => Pos Flag** **R1 == imm => 0 Flag**
10100	**CMPR, R1, R2**	**Compares R1 with R2 and sets flags (R2 - R1)** **R1 > R2 => Neg Flag**

		R1 < R2 => Pos Flag **R1 == R2 => 0 Flag**
01011	**ADD, R1, imm**	**Adds value of R1 and immediate and places the result into R1**
11001	**ADDR, R1, R1, R2**	**Adds the value in R1 and the value in R2 and places the result into R1**
01100	**SUB, R1, imm**	**Subtracts the immediate from R1 and places the result in R1**
11010	**SUBR, R1, R1, R2**	**Subtracts the value in R2 from the value in R1 and places the result into R1**
01101	**SHL, R1, imm**	**Shifts the value in R1 left by the value specified in the 8 bit immediate**
01110	**SHR, R1, imm**	**Shift the value in R1 right by the value specified in the 8 bit immediate**
11000	**AND, R1, R1, R2**	**ANDs the values in R1 and R2 and places the result into R1**
10011	**NOT, R1**	**Flips all of the bits in R1**
11011	**OR, R1, R1, R2**	**ORs the values in R1 and R2 and places the result into R1**
11100	**XOR, R1, R1, R2**	**XORs the values in R1 and R2 and places the result into R1**
00000	**NOOP**	**Performs no operation**
11111	**HLT**	**Halts processes**

3.1.1. Condition Codes

Mnemonic Extension	**Meaning**	**Condition flag state**
EQ	Equal	Z == 1
GT	Greater than	N == 1
LT	Less than	P == 1
OV	Overflow	C == 1
SV	Signed Overflow	V == 1

3.1.2. Addressing Modes

Register Indirect (value in register is effective address of data) - Used in our memory (LDR and ST) instructions. Our instructions are only 16 bits long so in order to effectively be able to address through our memory the addresses are first stored in a 16 bit register instead of being used directly in a memory instruction.

Register Mode (address of register is effective address of data) - Used in many of our ALU operations. Fast operations using immediates and registers.

Implied Mode (data has no effective address) - Used for our branch and special instructions. No data is needed in the instruction.

Immediate Addressing Mode (data is present in address in instruction) - Used in our 8-bit memory operations (LL, LM, LH).

3.1.2.1. Immediate

Immediates are specified by a # in front of the number. The assembler will autocorrect equivalent instructions i.e ADD R1, R1, #-1 will become SUB R1, R1, #1 because -1 is not possible to represent using our encoding.

Syntax: #<immediate>

Example: CMP, R0, #7

Encoding:

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**

immediate_8

3.1.3. Instruction Descriptions

3.1.3.1 LDR - Load from register

Syntax:

LD, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)

Rs = Rt

Flags updated:

Does not update flags.

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	0	0	0	1	Rs			0	Rt			0	0	0	0

Usage and Examples:

The LD instruction is used to load values from register to register

LD R1, R2 Loads the value of R2 into R1

3.1.3.2 LD - Load from memory

Syntax:

LD, <Rs>,[<Rt>]

Pseudo Code: (Can be in C/Verilog)

Rs = value at addr <Rt>

Flags updated:

Does not update flags.

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	0	0	0	0	Rs			0	Rt			0	0	0	0

Usage and Examples:

The LD instruction is used to load values from memory to a register

LD R1, [R2] Loads the value at the address in R1 to the register R1

3.1.3.3 LL - Load low

Syntax:

LL, <Rs>,<8-bit Imm>

Pseudo Code: (Can be in C/Verilog)

Rs [0:7] = <8-bit Imm>

Flags updated:
Does not update flags.

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	Rs			Immediate							

Usage and Examples:
The LL instruction is used to load an immediate value into the lower 8 bits (0-7) of a register
LL, R1, 11111111 Loads 11111111 into bits 0-7 of R1

3.1.3.5 LH - Load high

Syntax:
LH, <Rs>,<8-bit Imm>

Pseudo Code: (Can be in C/Verilog)
Rs [16:23] = <8-bit Imm>

Flags updated:
Does not update flags.

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rs			Immediate							

Usage and Examples:
The LH instruction is used to load an immediate value into the upper 8 bits (16-23) of a register
LH R1, 11111111 Loads 11111111 into bits 8-15 of R1
Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

3.1.3.6 ST - Store

Syntax:
ST, <Rs>,[<Rt>]

Pseudo Code: (Can be in C/Verilog)
[<Rt>] = Rs

Flags updated:
Does not update flags.

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rs			0	Rt			0	0	0	0

Usage and Examples:

The ST instruction is used to store the value from a register into the address specified by the value of another register

ST, R1, [R2] Stores the value of R1 into the address in R2

3.1.3.7 BRE - Branch if equal

Syntax:

BRE

Pseudo Code: (Can be in C/Verilog)

```
if (z flag high)
        goto address in <R7>
```

Flags updated:

Does not update flags.

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Usage and Examples:

The BRE instruction branches to the address in register 7 if the z flag is high

BRE Branches to the address in R7 if the z flag is high

3.1.3.8 BRG - Branch if greater than

Syntax:

BRG

Pseudo Code: (Can be in C/Verilog)

```
if (p flag high)
        goto address in <R7>
```

Flags updated:

Does not update flags.

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Usage and Examples:

The BRG instruction branches to the address in register 7 if the p flag is high

BRG Branches to the address in R7 if the p flag is high

3.1.3.9 CMP - Compare with immediate

Syntax:

CMP, <Rs>, <8-bit Imm>

Pseudo Code: (Can be in C/Verilog)

```
if (8-bit Imm greater than Rs)
        set n flag
if (8-bit Imm> less than Rs)
        set p flag
if (8-bit Imm is equal to Rs)
        set z flag
```

Flags updated:

N, P, Z

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	1	0	1	0	Rs			Immediate							

Usage and Examples:

The CMP instruction is used to compare the value in a register with an immediate

CMP R1, 11111111 Compares R1 and 11111111 and sets the flag accordingly

3.1.3.10 CMPR - Compare with register

Syntax:

CMPR, <Rs>, <Rt>

Pseudo Code: (Can be in C/Verilog)

```
if (Rt greater than Rs)
        set n flag
if (Rt less than Rs)
        set p flag
if (Rt is equal to Rs)
        set z flag
```

Flags updated:

N, P, Z

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	0	1	0	0	Rs			0	Rt			0	0	0	0

Usage and Examples:

The CMPR instruction is used to compare the value in a register with a value in another register
CMPR R1, R2 Compares the values in R1 and R2 and sets flags accordingly

3.1.3.11 ADD - Add immediate

Syntax:
ADD, <Rs>, <8-bit Imm>

Pseudo Code: (Can be in C/Verilog)
Rs = Rt + 8-bit Imm

Flags updated:
N, P, Z, C

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	1	0	1	1	Rs			Immediate							

Usage and Examples:
The ADD instruction is used to add the value inside a register and an immediate and place the result back into the register
ADD R1, 11111111 Adds 11111111 and R1 and places the result into R1

3.1.3.12 ADDR - Add register

Syntax:
ADDR, <Rs>, <Rt>, <Rd>

Pseudo Code: (Can be in C/Verilog)
Rs = Rt + Rd

Flags updated:
N, P, Z, C

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	0	0	1	Rs			0	Rt				Rd		

Usage and Examples:
The ADDR instruction is used to add the value inside a register and a value inside another register and place the value in a register

ADDR R1, R1, R2 Adds the values of R1 and R2 and places the result into R1

3.1.3.13 SUB - Subtract immediate

Syntax:

SUB, <Rs>, <Rt>, <8-bit Imm>

Pseudo Code: (Can be in C/Verilog)

Rs = Rt - 8-bit Imm

Flags updated:

N, P, Z, C

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	1	1	0	0	Rs			Immediate							

Usage and Examples:

The SUB instruction is used to subtract the value inside a register and an immediate and place the result back into the register

SUB R1, 11111111 Subtracts 11111111 from R1 and places the value into R1

3.1.3.14 SUBR - Subtract register

Syntax:

SUBR, <Rs>, <Rt>, <Rd>

Pseudo Code: (Can be in C/Verilog)

Rs = Rt - Rd

Flags updated:

N, P, Z, C

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	0	1	0	Rs			0	Rt				Rd		

Usage and Examples:

The SUBR instruction is used to subtract the value inside a register and a value inside another register and place the value in a register

SUBBR R1, R1, R2 Subtracts the value in R2 from R1 and places the result into R1

3.1.3.15 SHL - Arithmetic shift left

Syntax:

SHL, <Rs>, <8-bit Imm>

Pseudo Code: (Can be in C/Verilog)

```
Rs = Rs * 2 ^ (8-bit Imm)
```

Flags updated:

Z, C (updated as an error flag in case of overflow)

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	1	1	0	1	Rs			Immediate							

Usage and Examples:

The SHL instruction is used to shift the value ina register left by a given immediate

SHL R1, 00000001 Shifts the value in R1 left by 1 bit

3.1.3.16 SHR - Arithmetic shift right

Syntax:

SHR, <Rs>, <8-bit Imm>

Pseudo Code: (Can be in C/Verilog)

```
Rs = Rs / 2 ^ (8-bit Imm)
```

Flags updated:

Z, C (updated as an error flag in case of overflow)

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	1	1	1	0	Rs			Immediate							

Usage and Examples:

The SHR instruction is used to shift the value ina register right by a given immediate

SHR R1, 00000001 Shifts the value in R1 right by 1 bit

3.1.3.17 AND - And

Syntax:

AND, <Rs>, <Rt>, <Rd>

Pseudo Code: (Can be in C/Verilog)

Rs = Rt & Rd

Flags updated:
Does not update flags.

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	0	0	0	Rs			0	Rt				Rd		

Usage and Examples:
The AND instruction is used to AND values in two registers and place the result in a register
AND R1, R1, R2: Bitwise ands the values in R1 and R2 and places the result in R1

3.1.3.18 NOT - Not

Syntax:
NOT,<Rs>

Pseudo Code: (Can be in C/Verilog)
Rs = !Rs

Flags updated:
Does not update flags.

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	0	0	1	1	Rs			0	0	0	0	0	0	0	0

Usage and Examples:
The NOT instruction is used to flip all of the bits in a register
NOT R1: Flips all values in R1

3.1.3.19 OR - Or

Syntax:
OR, <Rs>, <Rt>, <Rd>

Pseudo Code: (Can be in C/Verilog)
Rs = Rt | Rd

Flags updated:
Does not update flags.

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	0	1	1	Rs			0	Rt			0	Rd		

Usage and Examples:
OR: takes two operands and does a bitwise or operation and then stores that value.
OR R1, R1, R2: bitwise ORs the values in R1 and R2 and places the result in R1

3.1.3.20 XOR - Xor

Syntax:
XOR, <Rs>, <Rt>, <Rd>

Pseudo Code: (Can be in C/Verilog)
Rs = Rt xor Rd

Flags updated:
Does not update flags.

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	1	0	0	Rs			0	Rt			0	Rd		

Usage and Examples:
XOR takes two operands and does a bitwise xor operation and then stores that value.
XOR R1, R2, R3: XORs R2 and R3 and stores the result in R1

3.1.3.21 NOOP - No operation

Syntax:
NOOP

Pseudo Code: (Can be in C/Verilog)
() - processor holds for a cycle

Flags updated:
Does not update flags.

Encoding:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Usage and Examples:

NOOP instruction holds all values currently in the processor for a cycle. Used for flushing and timing.

NOOP: do nothing

3.1.3.22 HLT - Halt execution

Syntax:

HLT

Pseudo Code: (Can be in C/Verilog)

stop

Flags updated:

Does not update flags.

Encoding:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

Usage and Examples:

The HLT instruction is used to immediately stop the processor's execution. It is used for testing and error purposes.

HLT: stop processor

4. Microarchitecture

4.1. I2C Controller

This module will interface with the audio CODEC, the audio input buffer, and the audio output buffer. This controls the CODEC configuration and takes audio samples from the CODEC and sends them toward the processor for modification. A majority of I2C commands to the CODEC will be sent when we initialize our program. This will send commands over I2C bus to the CODEC to set the correct configuration mode, volume level, and sound select.

Since we will no longer be able to use the board for our project, we will create a CODEC module that contains all of the ports we would be using in our design. This module will be used for testing purposes to ensure correct behavior of the I2C module. More details of how the CODEC module will be designed and tested will be coming in the testing review document.

Our I2C module will have an always block containing a set of case statements that check our i2c send counter. Depending on the count value, we will know what information to send over the I2C bus. We always send 3 bytes of information when we issue a command. The first byte represents the slave address we would like to write to. While we are no longer using the on-board CODEC, we will still issue this address byte so that future developers could adapt this to a board. The next two bytes represent the command we will send. A helper module named clock_500 communicates with our I2C controller to send the proper command based on an internal state machine.

Signal Name	I/O	Source/Target (blocks)	Description
clk	I	main	
CLOCK	I	clock_500	Clock for issuing I2C commands; generated by helper
rst_n	I	main	Active low reset
new_cpu_data	I	Audio Output Buffer	alert from the Audio Output buffer telling us we have new data in the buffer to be processed
Aud_Output_Data [15:0]	I	Audio Output Buffer	retrieve processed data from the audio output buffer to be sent back to the CODEC
ack	O	Audio Output Buffer	tells the output buffer that we have received the new data from the buffer, and that the buffer can shift out that data
Audio_Input_Data [15:0]	O	Audio Input Buffer	holds the new audio input data to be sent to the audio input buffer
new_aud_sample	O	Audio Input Buffer	alerts the Audio Input Buffer when a new audio sample has been received from the CODEC
I2C_MODE	O	CODEC	Used to configure the CODEC. We will be configuring it for master mode
I2C_SDAT	O	CODEC	Line out to the CODEC SDIN. used to issue commands to the CODEC
I2C_DATA[23:0]	I	clock_500	Receive command data for transfer to CODEC from helper
I2C_CLOCK	O	clock_500	clock is driven by our helper module
ADC_DAT	I	CODEC	line into our controller from the CODEC holding the audio data from the CODEC
DAC_DAT	O	CODEC	line out from our controller the delivers DAC data
DACLRC	IO	CODEC	line controlling the left channel/right channel alignment for the DAC on theCODEC
ADCLRC	IO	CODEC	line controlling the left channel/right channel alignment for the ADC on the CODEC
BCLK	IO	CODEC	digital audio bit clock
END	O	clock_500	Alerts the helper we have completed a command issue
GO	I	clock_500	Alerts the controller that a command needs to be issued

4.1.1 clock_500

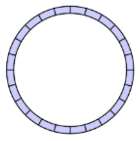
This module helps maintain the clock of the I2C controller and also issues 3 byte data commands to the I2c controller to be used as output. The base design was provided in the FPGA demonstrations directory. An additional state machine in this module will control which commands to issue on program startup. If additional features such as volume adjustment were to be implemented, a new state and corresponding command could be added into this module.

Signal Name	I/O	Source/Target (blocks)	Description
clk	I	main	-
reset	I	main	-

END	I	i2c	This alerts the helper that we have completed our command send
CLOCK_500	O	i2c	Clock generated for the i2c module
DATA[23:0]	O	i2c	Our 3 byte data containing (SlaveAddr(8 bit), Command(16 bit))
GO	O	i2c	Alert to the i2c controller that we need to send a command
CLOCK_2	O	CODEC	Clock for the CODEC module, runs at half normal clock speed

4.2. Audio Input Buffer

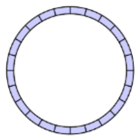
This module will interface with the I2C controller and the CPU. This is a circular buffer capable of reading and writing to BRAM. The buffer gathers data sampled by the CODEC when enable is high. Once the buffer is full, and the CPU is ready to collect data, the buffer passes data to the CPU. This buffer requires 2048 x 16 bits of memory.



Signal Name	I/O	Source/Target (blocks)	Description
clk	I	main	50 MHz
rst_n	I	main	Active low
enable	I	main	Fill samples from I2C/Codec
Aud_Data_In[16:0]	I	I2C	Audio data sample
Aud_Data_Out[16:0]	O	12C	Harmonized audio data
full	O	CPU	Tells CPU it has a data sample ready
collect	I	CPU	Cpu alerts buffer that it is ready for data

4.3. Audio Output Buffer

This module will interface with the I2C Controller and the CPU. This is a circular buffer capable of reading and writing to BRAM. The buffer gathers data processed by the CPU that will be re-output by the CODEC. This buffer requires 2048 x 16 bits of memory.



Signal Name	I/0	Source/Target	Description
clk	I	main	50 MHz
rst_n	I	main	Active low
Aud_Data[15:0]	I	CPU	Processed audio data
Buffer_Data[15:0]	O	I2C	Processed audio data ready to be played
new_data	O	I2C	Alerts I2C that there is new data ready to be played
ack	I	I2C	Acknowledges data received from I2C, shift data
data_ready	I	CPU	Alerts buffer that cpu has new data to be added

4.4. CPU

Below is a block diagram overview of the CPU for our project, which is a standard five-stage pipeline (with data forwarding between the MEM, EX, and ID stages) and one-level branch prediction.

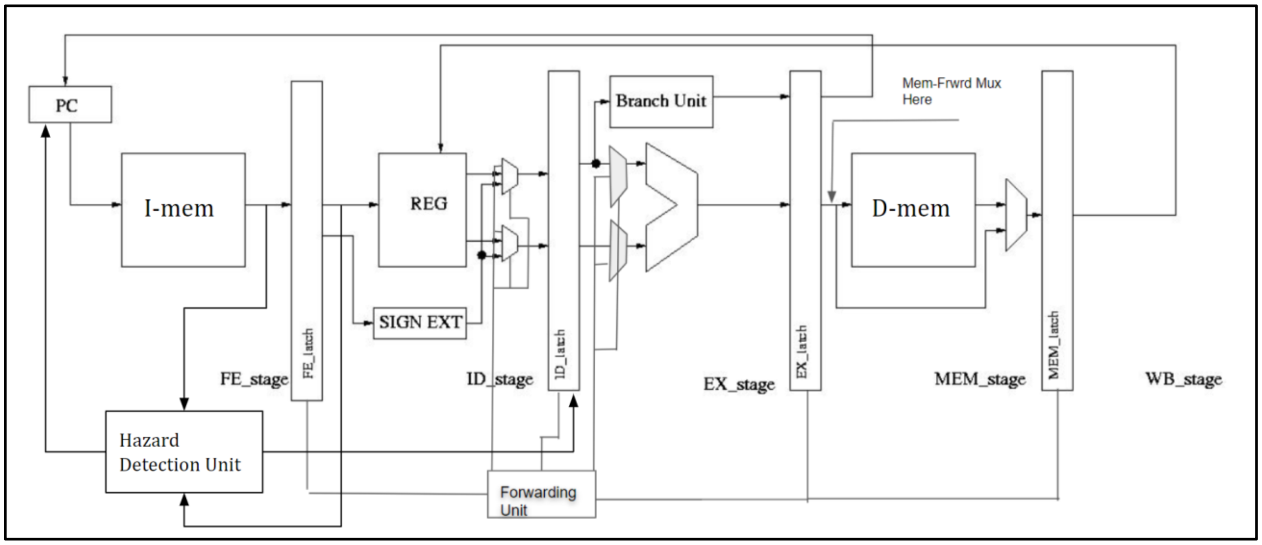


Figure 3: Block diagram of The Music Group's CPU showing distinct stages and modules

4.4.1. Instruction Fetch

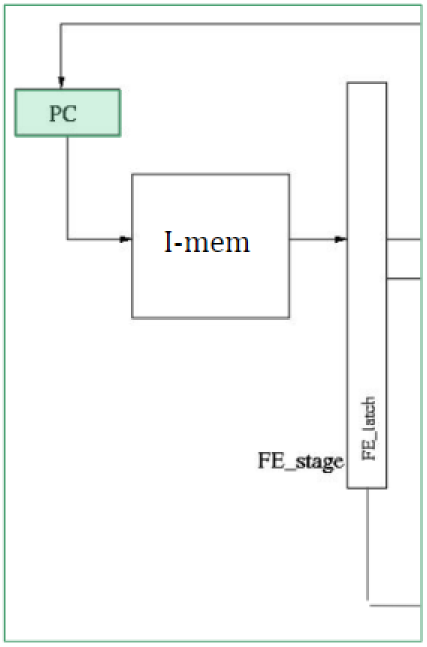


Figure 4: Block diagram of part of The Music Group's CPU showing only the FETCH stage

The instruction fetch (IF) stage reads the value of the program counter (PC) and outputs the corresponding instruction.

At the positive edge of the clock, if there was no branch, the PC is incremented, the instruction found in the I-mem and then output as curr_instr and loaded into the FE latch. If there was a branch instruction the branch_pc is loaded into the I-mem instead to find the correct instruction and the output is still found in curr_instr. If the signal rst_n goes low during the positive edge of the clock, then the values are reset.

Signal Name	I/0	Source/Target (blocks)	Description
clk	I	main	50Mhz
rst_n	I	main	Active low
branch_pc	I	EX_reg	New address in case of branch
curr_instr	O	FE_reg	Bitwise current instruction
is_branch	O	ID_reg	Enables branch prediction

4.4.2. Instruction Decode

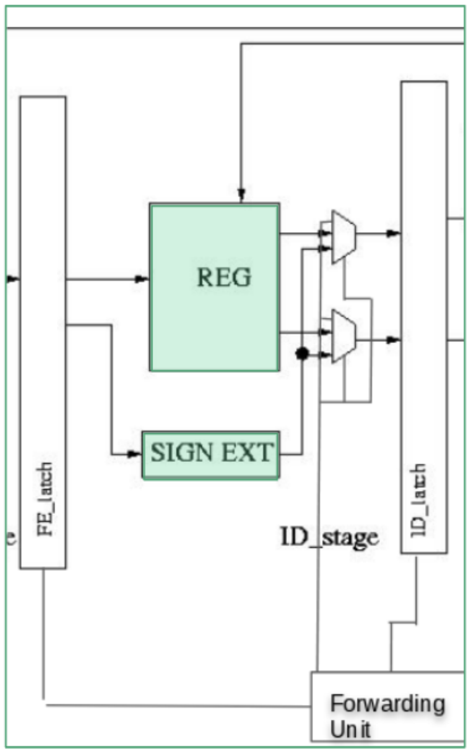


Figure 5: Block diagram of part of The Music Group's CPU showing only the DECODE stage

The instruction decode (ID) stage breaks down and processes the incoming instruction word, sign extending immediates and reading register values from the register file.

At the positive edge of the clock, the input values are flopped from the FE and MEM latches, and the output values are loaded into the ID latch. The curr_instr is input into the Register File, along with the r1, r2 (the registers that are potentially being read from) new_reg and the new_register_val the register being written to and the v0'alue being written there. The register file outputs r1_val, and r2_val which are the values being held in the specified registers. Immediates are input into the sign ext block where the values are sign extended to 8 bits. If the signal rst_n is low during the positive edge of the clock, then the values are reset.

Signal Name	I/0	Source/Target (blocks)	Description
clk	I	main	50Mhz
rst_n	I	main	Active low
new_reg	I	MEM_reg	Register to write back to
new_reg_val	I	MEM_reg	New register value to be written back
curr_instr	I	FE_reg	Bitwise current instruction
to_sign_ext	I	FE_reg	Immediate to sign extend
r1	I	FE_reg	R1 to read from
r1_val	O	ID_reg	R1 value read out from RF
r2	I	FE_reg	R2 to read from
r2_val	O	ID_reg	R2 value read out from RF

4.4.2.1 Register File

The register file (RF) can read two registers at once or write one register at once. All register values from R1 to R7 are stored in the RF.

Signal Name	I/0	Source/Target (blocks)	Description
clk	I	main	50Mhz
rst_n	I	main	Active low
new_reg	I	MEM_reg	Register to write back to
new_reg_val	I	MEM_reg	New register value to be written back
r1	I	FE_reg	R1 to read from
r1_val	O	ID_reg	R1 value read out from RF
r2	I	FE_reg	R2 to read from
r2_val	O	ID_reg	R2 value read out from RF

4.4.3. Execute

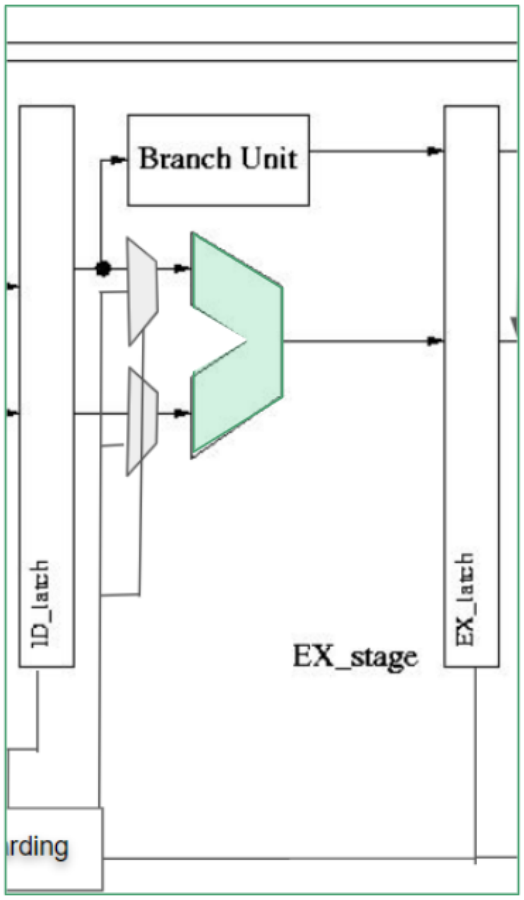


Figure 6: Block diagram of part of The Music Group's CPU showing only the EXECUTE stage

The execute (EX) stage does a lot of the work in the CPU. To ensure the audio input rate is accommodated, there are numerous multipliers and adders within each STFT unit. The EX stage uses the STFT and ISTFT algorithms to interpolate waveforms between time frames.

If the SPU is processing sound data, then its inputs will consist of the current audio signal amplitude x(n) and the complex number coefficients read from the hardcoded cosine and sine tables. At this time, the execute stage will assert the collect signal to enable data to be fed in by input audio buffer. Additional logic exists in the SPU for the purpose of computing basic logical or arithmetic information to allow the FindTune high-level code to modify the state of the processor as necessary.

After being passed through a windowing function to remove discontinuities, the processed sound data is passed through the STFT unit, which scale the real and imaginary parts of the number, perform fixed-point multiplication, shift the numbers to normalize them, then accumulate (take the summation) of the real and imaginary parts before taking their magnitude and storing it in a shift register. The ISTFT units then perform the same operation thus inverting the STFT and producing a sound wave with modified pitch across multiple octaves.

The three lower octaves and the nearest high octave are pitch-scaled simply by taking the frames and placing them closer together or further apart in time. All seven octaves are averaged and sent to the audio output buffer.

Signal Name	I/0	Source/Target (blocks)	Description
clk	I	main	50Mhz
rst_n	I	main	Active low
alu_in1	I	ID_reg	First ALU operand (possibly convolved x(n))
alu_in2	I	ID_reg	Second ALU operand (possibly exp(-j2□kn/N))
alu_out	O	EX_reg	Output of STFT computation (can be buffered)
branch_out	O	EX_reg	Target branch address (if taken)
collect	O	Audio input buffer	Alerts audio input buffer that it is ready for data
data_ready	O	Audio output buffer	Alerts audio output buffer that data is ready

4.4.3.1 STFT Unit

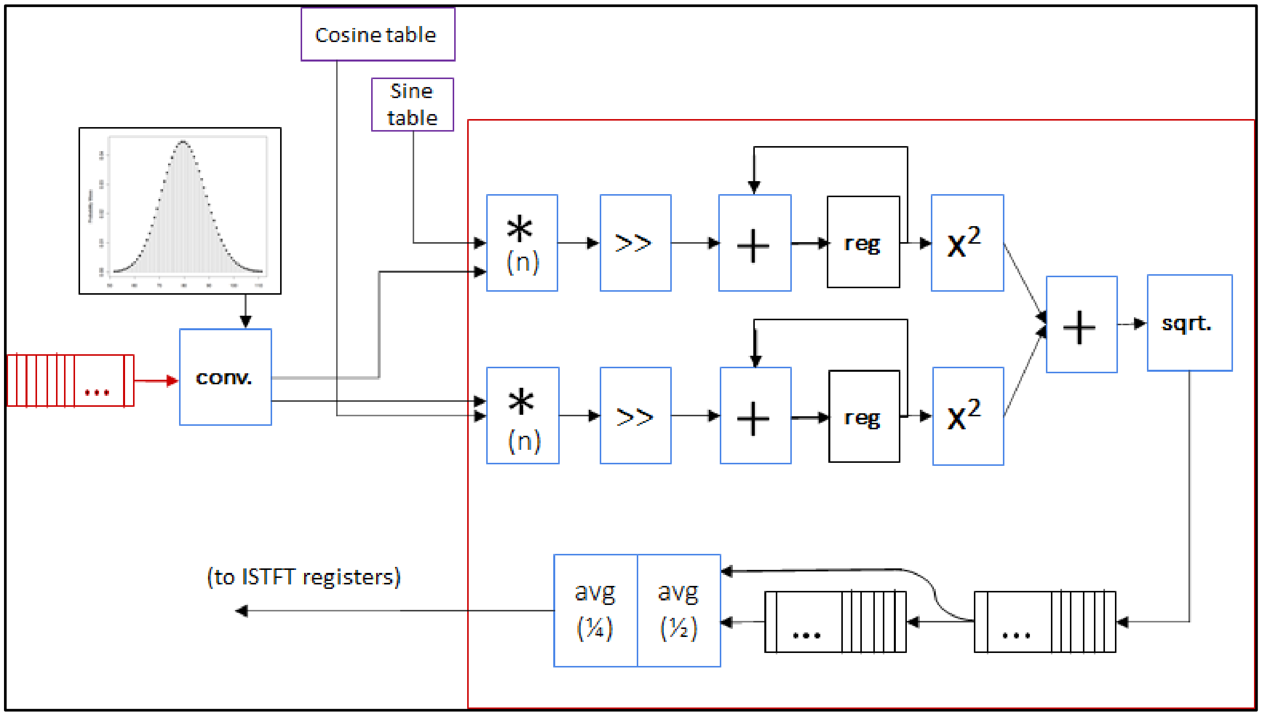


Figure 7: Block diagram of the SPU showing the STFT unit (in red) and surrounding logic

As audio inputs are read from the audio input buffer in chunks of 2048, they are fed into the single STFT unit (in red). The cosine and sine tables have precomputed values for the full range of audible frequencies. The SPU must process three clusters of interpolated STFT coefficients for the two highest octaves in parallel to meet the audio input/CODEC processing rate (which could be lowered if necessary). Interpolation after re-expansion replaces the STFT unit for all other octaves. Two buffers bridge the STFT and ISTFT units, serving to cache the two most recent frames in order to perform coefficient interpolation properly.

Signal Name	I/0	Source/Target (blocks)	Description
conv_audio_in	I	main	Current 16-bit audio signal (amplitude) convolved with a Hann window function
stft_quarter_in	O	ISTFT unit 2	STFT magnitude coefficient for current frequency bin weighted ¼:¾ towards the older sample
stft_half_in	O	ISTFT units 1 and 2	STFT magnitude coefficient for current frequency bin weighted ½:½ towards the older sample
stft_3quarters_in	O	ISTFT unit 2	STFT magnitude coefficient for current frequency bin weighted ¾:¼ towards the older sample
coeff_ready	O	Intermediate buffers	Alerts coefficient storage and interpolation registers that the next STFT coefficient is available for use

4.4.3.2 Interpolation

After the STFT has been performed for each of the lower octaves, the audio amplitudes and packets are interpolated and combined across frames as necessary. The analysis hopsize is 512 samples out of the 2048-large chunks, but the synthesis hopsize varies between octaves. Each new frame of audio contains only 512 *new* samples, and thus overlaps greatly with the previous frame.

Octave	**Synthesis Hopsize (in samples)**
-3	64
-2	128
-1	256
0	512 (unchanged)
+1	1024
+2	2048 (requires STFT/ISTFT)
+3	4096 (requires STFT/ISTFT)

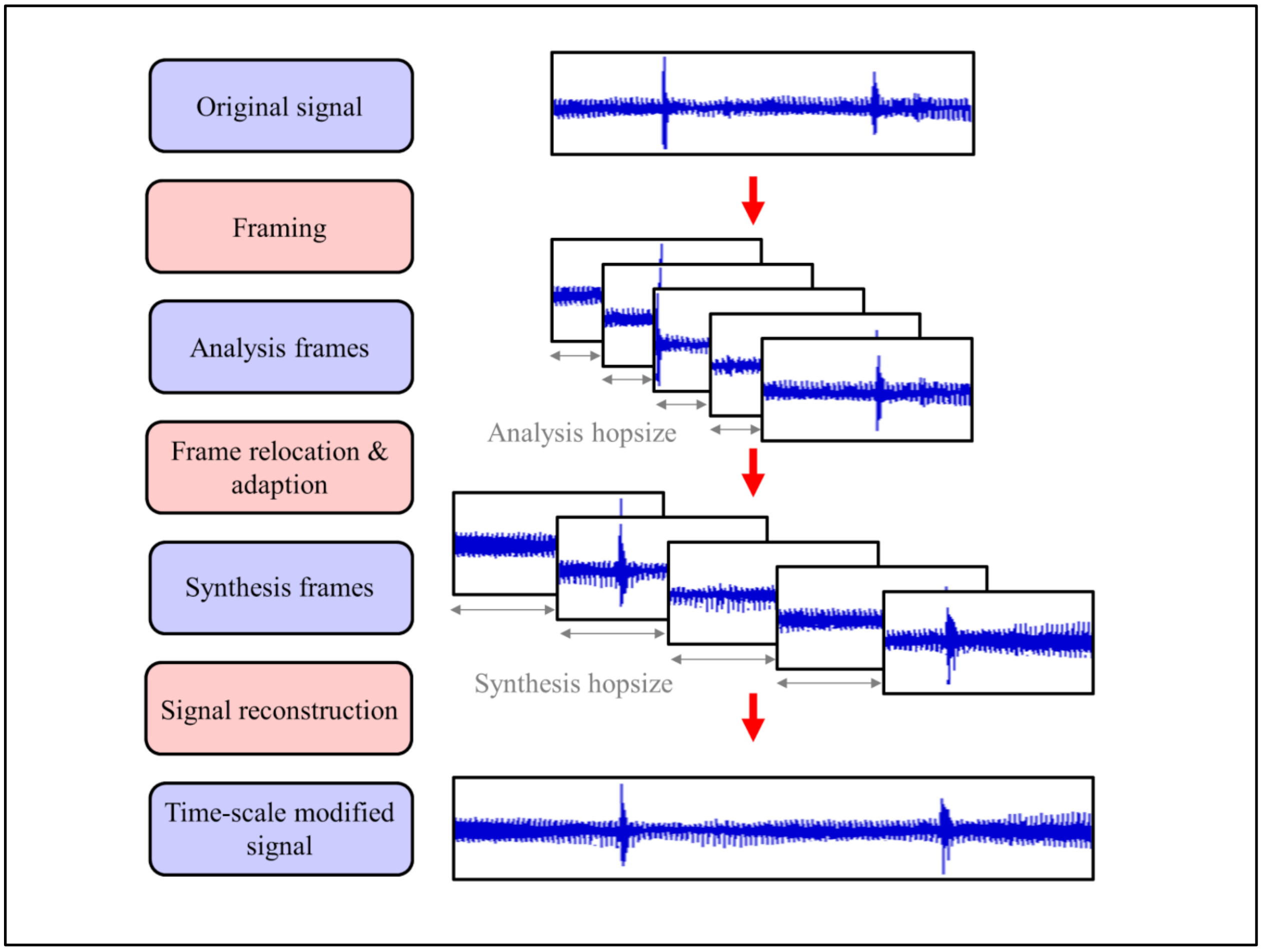


Figure 8: Visualization of how pitch shifting occurs in frame-based TSM

Source: https://en.wikipedia.org/wiki/Audio_time_stretching_and_pitch_scaling#/media/File:GeneralizedPrinciple_TSM.png

4.4.3.3 ISTFT Unit

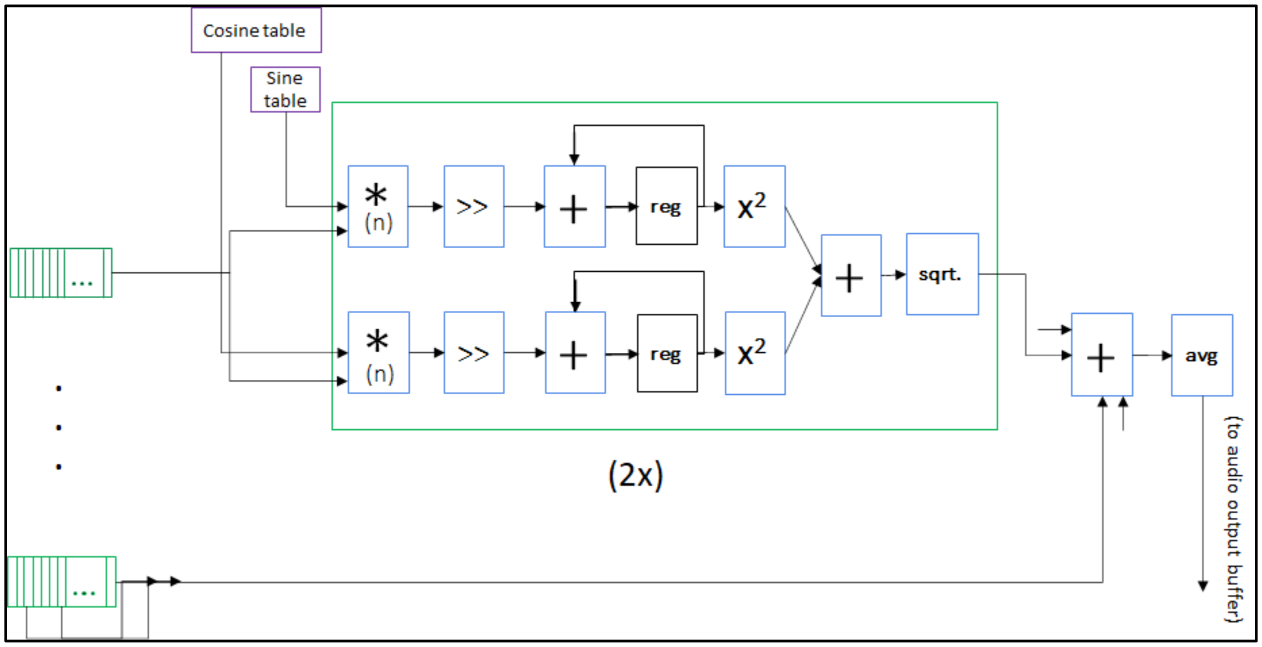


Figure 9: Block diagram of the ALU showing the ISTFT unit (in green)

The inverse STFT (ISTFT) is performed on a large number of output buffers and storage registers (dependent on final audio sampling rate and analysis hopsize) to restore the interpolated signal using the ISTFT unit (green). The output buffers and registers take in one windowed frame or coefficient at a time and discard the oldest frame. The ISTFT also requires specialized sine and cosine tables with precomputed values. For lower octaves, the ISTFT units are bypassed and the frames are added back together.

Signal Name	I/0	Source/Target (blocks)	Description
stft_quarter_in	I	ISTFT unit 2	STFT magnitude coefficient for current frequency bin weighted ¼:¾ towards the older sample
stft_half_in	I	ISTFT units 1 and 2	STFT magnitude coefficient for current frequency bin weighted ½:½ towards the older sample
stft_3quarters_in	I	ISTFT unit 2	STFT magnitude coefficient for current frequency bin weighted ¾:¼ towards the older sample
audio_out	O	main	Harmonized 16-bit audio output

4.4.4. Memory Access

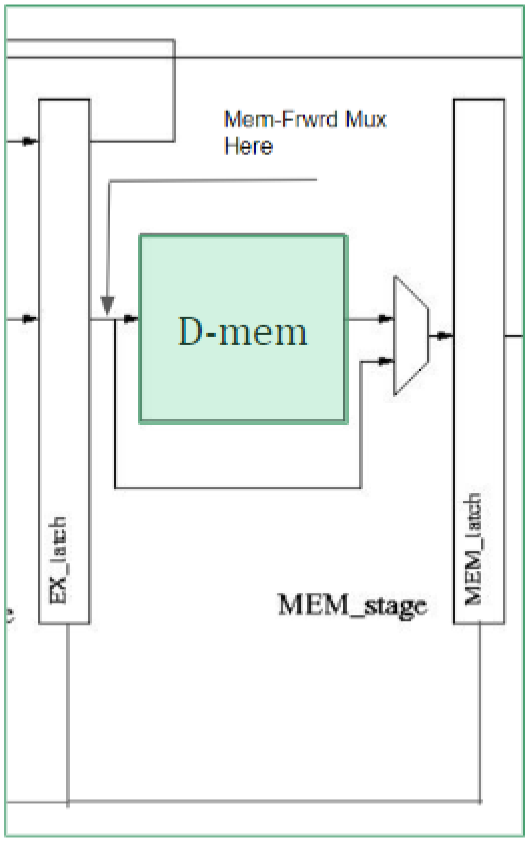


Figure 10: Block diagram of part of The Music Group's CPU showing only the MEMORY stage

The memory (MEM) stage reads or writes to memory as requested by the instructions.

At the positive edge of the clk signal the input values are flopped from the EX latch and the output values loaded into the MEM latch. The values flopped from the EX latch, dmem_data_in and dmem_addr_in are input into the D-Mem block. Mux_dmem_out chooses from the output of the D-mem and the dmem_data_in signals.

Signal Name	I/0	Source/Target (blocks)	Description
clk	I	main	50Mhz
rst_n	I	main	Active low
dmem_data_in	I	EX_reg	Value to write to memory
dmem_addr_in	I	EX_reg	Address to write to or read from
mux_dmem_out	O	MEM_reg	Either memory access result or ALU output

4.4.5. Register Writeback

This stage has no signals, although there is a wire that transfers data back to the register file.

4.4.6. Branch Prediction Unit

The branch prediction unit is a module that interfaces with the fetch and decode stages. It contains a branch-target buffer which is a table. The entries in the table contain the PC of branch instructions, and the predicted address, prediction bits. The BPU will take in curr_inst and check if it corresponds to an entry in the table. If it is not in the buffer, it checks if it is a branch instruction using the is_branch signal. If it is a branch instruction, it adds a new entry to the table. If it is not a branch instruction it continues normal execution. If the entry was found in the table, it checks the prediction bits. If the branch is predicted taken it outputs the address found in the entry. If it predicts not taken it follows normal program flow of PC + 4. When the branch is resolved, if mispredicted, it stalls the CPU and updates the prediction bits.

Signal Name	I/0	Source/Target (blocks)	Description
curr_inst	I	FE_reg	Current instruction
prev_PC	I	PC	Address of current branch instruction in decode
is_branch	I	ID_reg	Enables branch prediction
stall	O	CPU	Misprediction or cache miss
branch_out	O	EX_reg	Target branch address (if taken)

4.4.7. Forwarding Unit

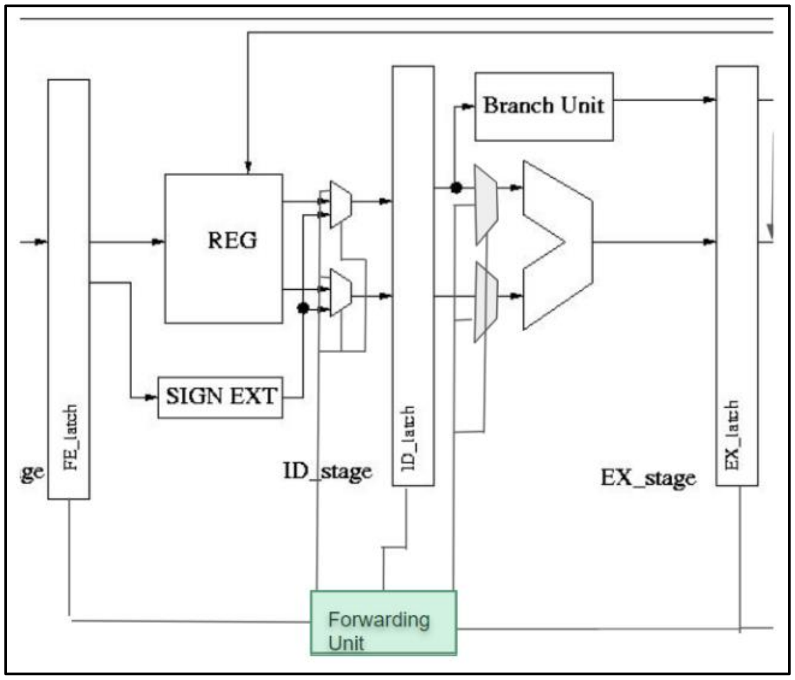


Figure 11: Block diagram of part of The Music Group's CPU showing only the range of the forwarding unit

The forwarding unit ensures efficient and correct operation of the CPU by allowing computation results to be available before writeback. Types of forwarding will include EX-to-EX, MEM-to-EX, and MEM-to-MEM, considering which registers from later pipeline stages are being used in earlier ones, and ensuring that the newest value from the latest pipeline stage is always used when possible.

Signal Name	I/0	Source/Target (blocks)	Description
FE_result	I	FE_reg	Result of FETCH stage
ID_result	I	ID_reg	Result of DECODE stage
EX_result	I	EX_reg	Result of EXECUTE stage
MEM_result	I	MEM_reg	Result of MEMORY stage
ID_fwd_mux	O	ID stage	Decide whether to use forwarded data
ID_fwd_data	O	ID stage	Forwarded data to use
EX_fwd_mux	O	EX stage	Decide whether to use forwarded data
EX_fwd_data	O	EX stage	Forwarded data to use

4.4.8. Hazard Detection Unit

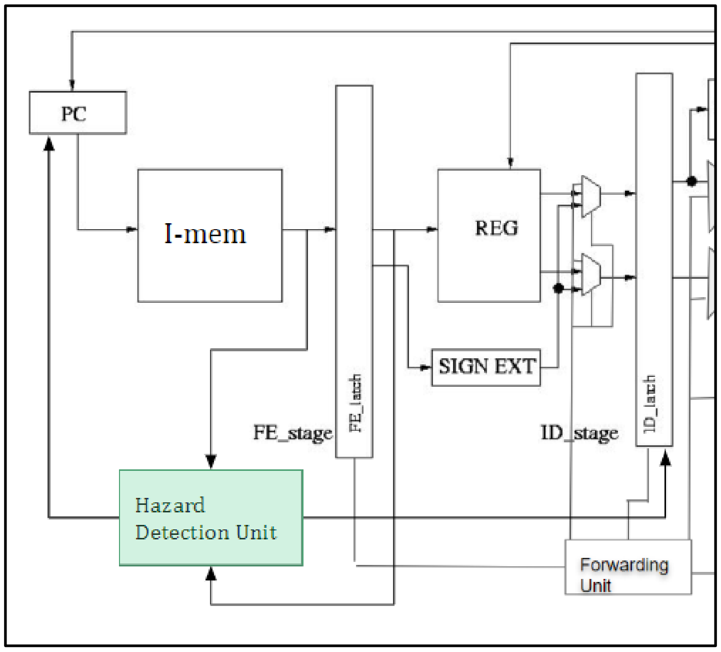


Figure 12: Block diagram of part of The Music Group's CPU showing only the range of the hazard detection unit

The hazard detection unit ensures that there is a load-to-use stall inserted after a memory read and an instruction immediately after that uses that read's result. If this situation is detected, it stalls IF and ID and inserts a NOP into the EX stage, making the result of the memory read available after a one-cycle delay.

5. Software

Our software consists of a compiler and assembler. These all work in conjunction to convert our high level language to machine code and make sure it is working as designed.

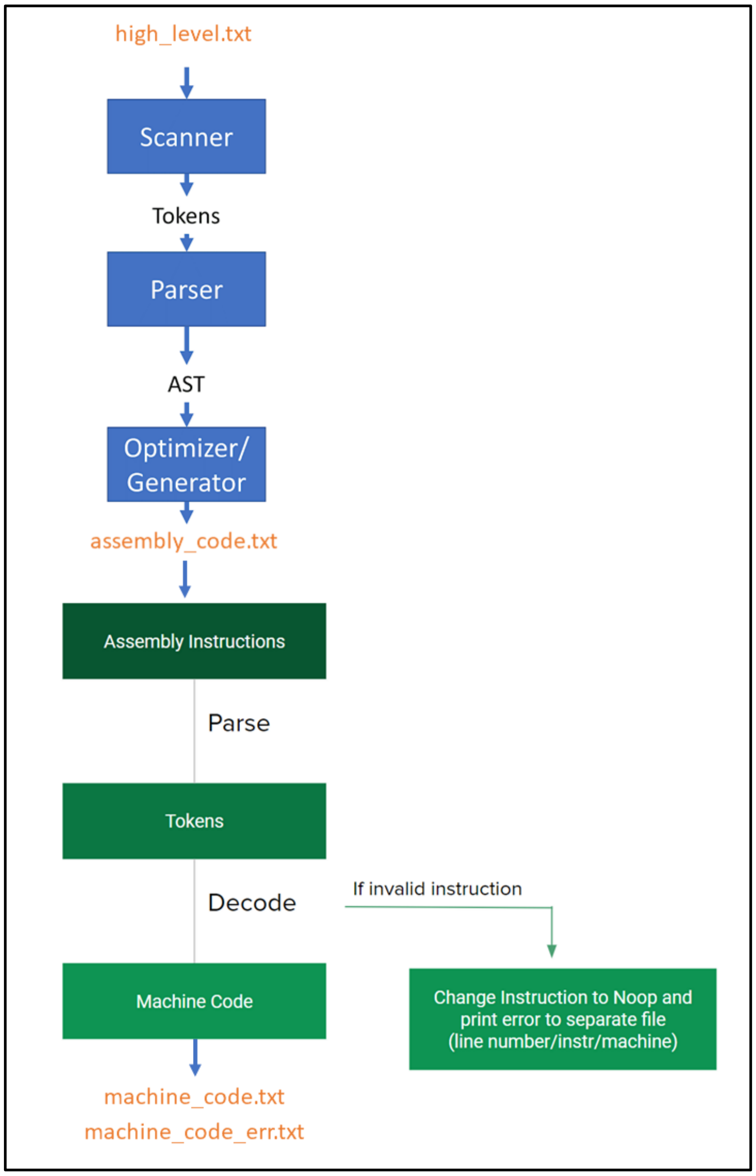


Figure 13: Flowchart outlining stages of code manipulation through the FindTune compiler and assembler

5.1 Compiler

Our compiler takes in a text file containing code in our high level language, FindTune. The compiler converts FindTune into assembly language which is sent to the assembler in a different text file called *assembly_code.txt*. The text file contains one assembly instruction per line. The compiler is split into three stages: scanner, parser, and generator/optimizer. The scanner takes the high level instructions and breaks them into tokens which are sent to the parser. The parser takes the tokens and creates an AST to be given to the generator which transforms the AST into assembly language. After the assembly language has been created it is sent to the assembler.

The context-free grammar of FindTune is displayed below. FindTune is implemented using Python.

```
expr := if ( bool ) { expr } else { expr } | for var in ( math ,
math ) expr | fpga. action () | fpga.setOctaves( bitstring ) |
fpga.wait( int )

action := enableSample | disableSample | displayState |
enableGraphics | disableGraphics | freeze | unfreeze |
enableAutotune | disableAutotune

math := int | bitstring

bitstring := {bit}+

bit := 1 | 0

bool := (math == math) | (bool == bool) | (!bool) | (bool &&
bool) | (bool || bool)
```

Usage:

python3 compiler.py [-p *prefix*] *filename*

Arguments	Description
prefix	

	<user specified>	Filename prefix for all output files, including scanned tokens, the parsed AST, final assembly code, and errors
	<none>	Write only final code to *assembly_code.txt* and abort and print error if necessary
filename	Input text file that compiler will be reading in the FindTune language from	

Input:

A text file containing the *FindTune* instructions

Output:

A text file with the assembly instructions generated by the compiler, one instruction per line

(*assembly_code.txt*)

A folder containing the scanner output, parsed AST, and intermediate assembly code with comments, if folder name (i.e. prefix) specified.

5.2 Assembler

The assembler is also written in Python. It interfaces with the hardware, by taking in a file of assembly instructions from the compiler, transforming the assembly into machine code and storing that in an output file called *machine_code.mif*. Another text file called *machine_code_err.tx*t will be output that will be populated with any errors that the assembler comes across in the assembly instructions. This error file will not be used by any other module and is just for debugging purposes. The *machine_code.mif* file is given to the CPU and the machine instructions are stored in the Instruction Memory on the CPU and any global variables or constants are stored in the Data Memory on the CPU.

Usage:

python assembler.py *filename* [-o *filename*] [-e *filename*]

Arguments:

Arguments	Description
-o --output_file	Change the output filename to have the title of the given filename
-e --error_file	Change the error filename to have the title of the given filename
filename	Input file that the assembler will be reading in the assembly from

Input:

Text file of assembly instructions coming from the compiler, one instruction per line

(*assembly_code.txt*)

Output:

Mif file of the machine code for the processor, and text files for lines of machine code that cause errors.

(*machine_code.mif*), (*machine_code_err.txt*)

5.3 Memory

Instruction Memory

Addr	Value	Notes
0x0000	Instructions	--
…..	Instructions	--
0xFFFF	Instructions	--

Data Memory

Addr	Value	Notes
0x0000	Aud_Data_Out	read only, input audio
0x0001	Aud_Data	write only, output audio
0x0002	Data memory	--
…	Data memory	--
0xFFFF	Data memory	--

5.4 Applications

We proposed two features, both of which were stretch goals, that served to augment and enhance our design with regards to user interaction. In terms of implementation, the interface between the FPGA and our high-level software will probably be managed using either Quartus, some other FPGA management tool, specialized IP blocks, or a combination of these. Chances are the applications will require some type of device with minimal processing capability, for the sound wave storage and graphics rendering, and a modern laptop or desktop computer with sufficient memory, RAM, and CPU and GPU capacities would suffice.

1. Input Pitch Visualization
2. Sound-Based Display

5.4.1. Input Pitch Visualization

To verify and clearly demonstrate that our implementation can accurately detect and process pitches, as a stretch goal we will display, either in classic musical notation, as a note name, as a frequency, or some combination of all these features, the pitch being input into the DE1 board. A tentative example image output is shown below.

It is possible that the pitch of the sound may be inexact, or may cease to exist altogether. Therefore this display feature must accommodate these possibilities while also showcasing the efficacy of our pitch detection and processing.

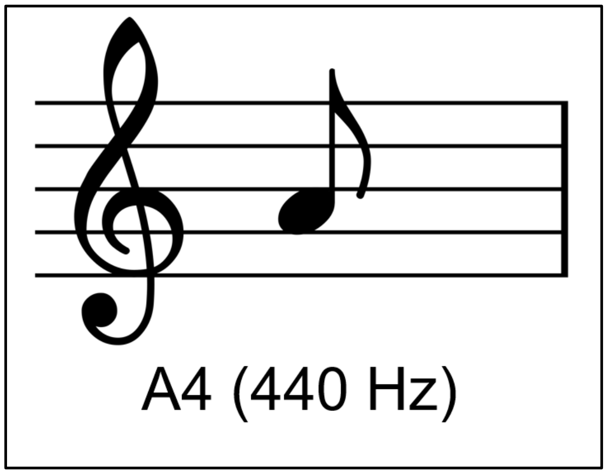


Figure 14. Example VGA note display idea

5.4.2. Sound-Based Display

Another way to verify that our hardware is functioning optimally would be to display some sound-related video to a computer monitor. This would be implemented in a way similar to the pitch visualization, except that the modules from Mini Project 2 (the DE1_SoC_CAMERA) would be reused with modification. The visualization could assume multiple forms, such as two waveforms, each representing the input (unmodified) and output (autotuned or harmonized) sound waves, or colorful patterns, shapes, or animations related to the sound.

6. Testing and Simulation

- L1: We began our testing with individual unit tests to ensure the smaller components of our project all worked on their own
- L2: We then moved to integration testing once partnered components passed their respective unit tests
- L3: Our testing finished with system tests of the hardware and software with as much integration as possible under remote conditions

What was tested

- Compiler (scanner, parser, generator)
- Assembler

- CPU (IF, ID, EX, MEM, WB)
    - ALU: ADD, SUB, AND, etc.
    - Signal Processing Unit (SPU): STFT, ISTFT, SQRT
- Other hardware modules (i2c, audio buffers, codec)
- Overall Input/Output

How it was tested - Compiler (L2) - Ben, Ilhan, Tristan

- "Crowdsourcing": distributed code to team and allow free usage
- Output/Store results of each compiler stage (scanner (L1), parser (L1), generator (L1) passes 1 and 2) to allow tracing back to source of error
- Preprogrammed (stored in .txt/.ft file) source of many different test cases, ran compiler and checked results (and that no errors were raised)
- Demo:
    - 1. Simple for loop with single if statement and directive
    - 2. Nested for loops with several cascading if statements, all boolean logic and directives
    - 3. FindTune code with a small non-obvious bug that makes it unable to compile

The scanner, parser and generator were all tested separately

When they passed the individual tests they were tested again together

Tests for

- Nested and unnested for loops
- Nested and unnested If and if else statements
- Compare operations in if statements
- All FPGA instructions

Test Case Details - Assembler L1 (Dayton, Tristan)

- Tested for completeness - all instructions
- Tested immediates - handling of immediates
- Tested for potential errors
    - Negative immediates
    - Out of bounds immediates
    - Incorrect instruction
    - Incorrect instruction format
    - Out of bounds register

- For correct cases have comparative code seen on the right

How it was tested - all software L3 (Ilhan, Ben, Dayton, Tristan)

After the two software components, compiler and assembler, passed their respective tests we tested them together using a simulator

This was used to ensure that the machine code that is being generated by the assembler from the compiler has correct grammar

How it was tested - SPU (STFT (L1)) - Ilhan

- Used directed random in Verilog, had golden model of STFT to compare coefficients to (using formula)
- MATLAB code to run STFT algorithm on sound sample and compare to STFT unit within a certain tolerance (due to sin/cos approximation table)
- Demo:
    - 1. STFT of sinusoid with T = 512 (quarter length of input buffer)
    - 2. STFT of two sinusoids with T = 64 (1/32 length) and T = 512 (quarter length)
    - 3. STFT of random noise

How it was tested - SPU (ISTFT (L1) & SQRT (L1)) - Ilhan

- aun through STFT unit, then ISTFT, see if we got back out the original signal again within a certain tolerance
- Demo:
    - All three cases for STFT run backwards through ISTFT to get original result
- For SQRT, Take x, square it, then take the square root to see if equal to original value
- Demo:
    - 1. Square root of 268,435,455 (2^28 - 1)
    - 2. Square root of 0
    - 3. Square root of 4
- For STFT unit, sinusoidal input was generated at certain notes (e.g. A4) and coefficients were examined
- Random values were also tested (less effective but better coverage)
- For ISTFT unit, had various fixed frequencies first fed through STFT unit

How it was tested - ALU (L1) - Ilhan

- Created SystemVerilog array of all ALU opcodes, iterated through each opcode and generated around 100 random test cases, verified that ALU output matched the golden model.
- Used $random and took upper and lower half as ALU operands, iterated through all opcodes
- Demo:
    - 1. ADD 0xAAAA and 0x5555 to get 0xFFFF
    - 2. AND 0xAAAA and 0x5555 to get 0x0000
    - 3. SHR 0x8000 by 0x000F to get 0x0001
    - 4. Any single operation of choosing

How it is tested - i2c controller/codec: L1 - Alex

- A codec module was created to mimic the behavior of the onboard codec.
- This module contains memory to store register data, we determined that the i2c controller properly sends a command to change a register.
- We also had memory to store sample audio data that our i2c controller can read from and separate memory to store processed audio data.
- Verilog testbench simulated input from the audio buffers to the i2c controller and the state of the codec and i2c modules matched the expected state.
- Initial unit test was designed for i2c controller and clock_500; additional integration testbench included codec module.

Test case details - i2c controller - Alex

- Known audio sample values are hardcoded into codec memory and sent out serially once the activate command is issued by the i2c controller; expects correct audio sample at output to audio buffer
- Random cases not necessary since we expect any given input to end up in same memory location within codec unchanged
- We test to ensure acks are sent back from the codec module for each byte of data transmitted via i2c
- We also test that we reach the codec active state

L1 How is it tested - Audio Buffers - Sam

- Audio Input and Output buffer communication is tested with a ‘fake’ I2C and CPU
- I2C and CPU operate at deposit and collect samples at different frequencies, those frequencies are mimicked when filling and emptying audio buffers

L1 Test case details - Audio Buffers - Sam

- Audio buffers are tested by filling circular queue with numbers 1 through 2047
- Numbers are removed and reviewed to make sure all original samples are present and have been removed in the right order
- Intermittent pauses are introduced in enable and collect signals to test buffer ability to handle any pause in normal system flow

Integration Testing - Hardware

After testing correct functionality of all submodules, we tested integration of:

- Codec, I2C and audio buffers, L2 Sam and Alex
- Basic CPU functionality, L2, Sam, Alex, Ilhan
- CPU + branch prediction + forwarding functionality, L2 Sam, Alex, Ilhan
- Entire system functionality L3, see next slide

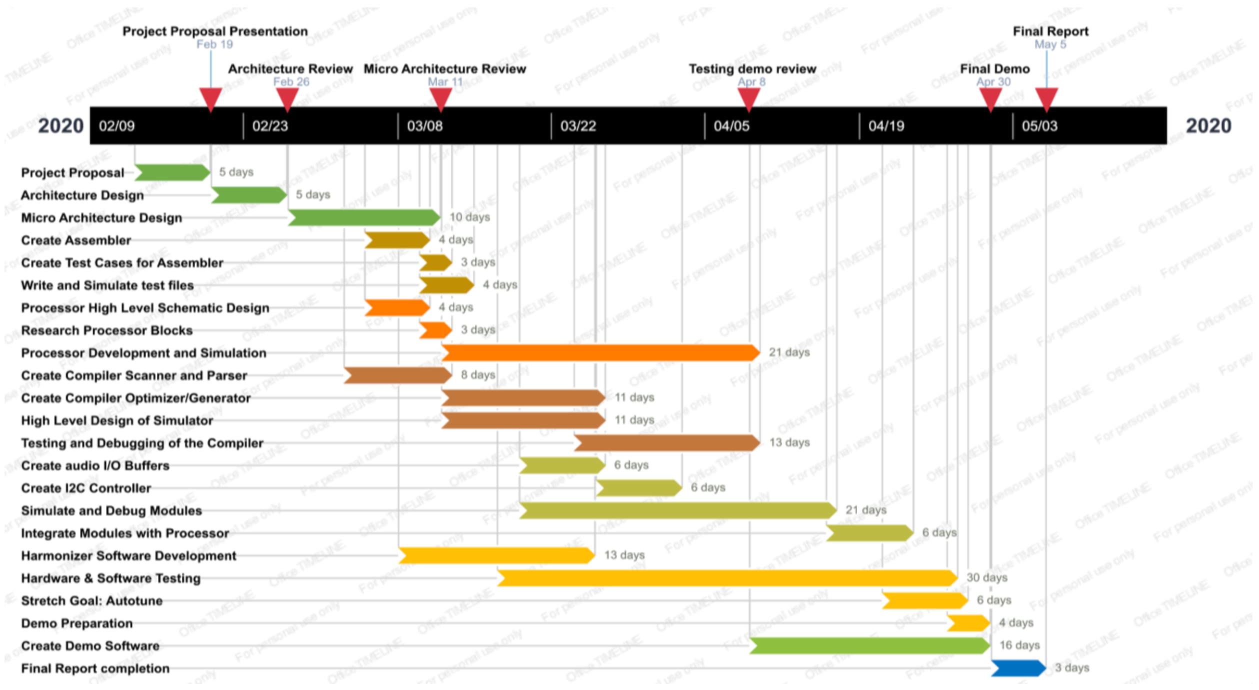
Testing the Output

- Use the simulated CODEC Module
  - Using the codec mimic that we will have created we can hard code in a known frequency, then store the output of our modules.
  - Can create test benches for this data using known inputs for the switches.
  - Essentially want to be able to input an MP3 file, and then have it output a different MP3 file
  - Can use MATLAB to convert the original file to binary then modulate as desired to have the comparison file

    MP3 to Binary File Conversion

    - filename = '1kHz_44100Hz_16bit_05sec.mp3';
    - fid = fopen(filename, 'r');
    - file_bits = fread(fid, [1 inf], 'bit1=>uint8');
    - fclose(fid);

7. Timeline



**The report was completed throughout the course of the project.

8. Challenges Faced and Lessons Learned

For the last half of this project, we were working from home and did not have access to the DE1-SOC board. This situation caused us to have to change the way in which we tested the board. The way in which we were able to validate our final product was not ideal but was as close as we could get to a test using the hardware. After testing and confirming the functionality of the software and hardware blocks, we tested the project using code generated by our compiler and assembler and were able to get the expected result from the waveform. Ideally we would be able to actually listen to the sounds generated and compare them with one generated with Matlab. This approach is a nice confirmation, however is not useful until we have completed the rigorous testing we already had done so it ended up not being that large of a factor. Another big effect that our lack of board had on our project was having to create a mock CODEC to be able to properly simulate our architecture. Both of these problems reinforced the importance of unit testing. Without spending time after creating a module to devise and create a thorough round of tests for it, debugging for the L2 and L3 tests would become difficult if not impossible.

One of the challenges specific to the I2c controller module and the codec module was mimicking the functionality of the codec so that our project could be modified to work on an actual fpga if we ever got access to a board. This involved reading through the datasheets of the onboard codec so that both the I2c protocol and the audio data transfer was implemented correctly.

Aside from access to the board, working online came with more sets of challenges. One of those that must always be considered when programming as a group is code version control. With many people working on different pieces of the project simultaneously it is vital that any changes that are made in order to fix a problem, work with the rest of the team's code. We utilized Github in order to help us easily share access to the project, and have the most up-to-date code from every person working on the project. Our workload was divided in such a way that there were not that many chances for this to become a huge issue.

Without the scheduled meetings every week, being able to see the team and talk about what everyone is currently working on, communication became even more important. Our team used Slack in order to stay connected, and this worked very well for us. The app allowed us to both communicate as a team, as subgroups, and also individually depending on what information we needed or were trying to communicate. I think in the future, continuing to have a set time each week for the group to meet may be something to try.

Another way we were able to stay organized as a team is through the use of our documentation. Even before we went online, the team used a number of Google Documents that contained information about each piece of our project. These documents became even more important after the switch because they became the first place people could check to find answers to questions they may have. I think that the way this project was organized with the many presentations and document turn ins was an extremely effective way for us to prioritize such complete documentation of each stage. This is something that I think everyone can take away from this group project and apply it to their future class and work assignments.

The effectiveness of the strategy we employed was clearly evident when it came time to integrate all hardware modules together. Although there was an unexpected outcome in Modelsim from putting together the i2c, audio input/output buffers (AIB/AOB), and SPU (explained below), the testing and validation each hardware team member did individually resulted in a fairly successful and straightforward hardware integration. Most of the pre-integration work happened on GitHub in parallel with communication through Slack and Skype, where a top level module was created and modified until the connections were correct. For the actual assembly, the only error in the entire hardware system was a minor difference in specification between the AIB and SPU on how to transfer audio samples. Once this error was found by the team and rectified, the entire system functioned as expected.

This semester provided a unique set of challenges for each group. Our group was able to overcome those and come out with a finished product that passed all the validation testing we created for it. While unable to get the satisfaction of putting our code on the board and using our project, The Music Group was still able to make this project successful.

9. Contract and Workload

I, _____________ agree to do the work I say I will and to let my teammates know if I am unable to. I will be reasonable about dividing up work, and I will treat my teammates fairly regardless of circumstance. I also agree that I will settle any issues that affect our group's work or that I may have with teammates with a civil discussion.

Ilhan Bok: Worked on STFT/ISTFT and PSOLA within signal processing unit, helped with compiler.
Tristan Wentworth: Worked on assembler, compiler, and output validation.
Sam Bitter: Built audio input and output buffers. Helped perform i2c/codec/audio buffers integration. Worked on CPU including register file, instruction memory and pipelining. Helped integrate forwarding.
Alex Jarnutowski: Worked on the I2c Controller module, clock_500 helper module, codec module, testbenches of i2c_codec and i2c_aud. Also worked on the memory section of our CPU, testbench for our top level design, and the top level verilog module.
Dayton Lindsay: Worked on the Assembler and helped with the compiler as well as software testing.
Ben Holzem: Worked on compiler and software testing

Signed:
Ilhan Bok
Tristan Wentworth
Sam Bitter
Alex Jarnutowski
Dayton Lindsay
Ben Holzem