CENG-4303 HDL Design of Microprocessors

Final Design Document

Zachary Walden

Table Of Contents

1. Introduction **5**

2. Requirements **5**

3. Instruction Set Architecture **6**

3.1 No Operation 6
3.2 Increment 6
3.3 Decrement 6
3.4 Add 6
3.5 Subtract 7
3.6 Compare 7
3.7 Multiply 7
3.8 And 7
3.9 Or 7
3.10 Shift Right 7
3.11 Shift Left 8
3.12 Complement 8
3.13 Invert 8
3.14 Load 8
3.15 Load Immediate 8
3.16 Load Program Memory 9
3.17 Store 9
3.18 Move 9
3.19 In 9
3.20 Out 9
3.21 Jump 9
3.22 Branch if Carry Set 10
3.23 Branch if Carry Clear 10
3.24 Branch if Equal 10
3.25 Branch if Not Equal 10
3.26 Branch if Negative 10
3.27 Branch if Positive 10
3.28 Call 11
3.29 Return 11
3.30 Return From Interrupt 11
3.31 Push 11
3.32 Pop 11
3.33 Halt 11
3.34 Opcode Table 12
3.35 Errata 12
3.36 SFR Register Map 12

3.36.1 Stack Pointer Low (SPL) 12
3.36.2 Stack Pointer High (SPH) 12
3.36.3 X Pointer Low (XL) 12
3.36.4 X Pointer High (XH) 12
3.36.5 Y Pointer Low (YL) 12
3.36.6 Y Pointer High (YH) 13
3.36.7 Z Pointer Low (ZL) 13
3.36.8 Z Pointer High (ZH) 13
3.36.9 Timer One Control Register (T1CR) 13
3.36.10 Call Stack Pointer (CSP) 13
3.36.11 LED (LED) 13
3.36.12 Interrupt Controller Control Register (ICCR) 13
3.36.13 General Interrupt Control Register (GICR) 13
3.36.14 Timer Compare Byte 0 (TCB0) 13
3.36.15 Timer Compare Byte 1 (TCB1) 13
3.36.16 Timer Compare Byte 2 (TCB2) 14
3.36.17 Timer Compare Byte 3 (TCB3) 14
3.36.18 Timer Compare Byte 4 (TCB4) 14
3.36.19 Timer Compare Byte 5 (TCB5) 14
3.36.20 Timer Compare Byte 6 (TCB6) 14
3.36.21 Timer Compare Byte 7 (TCB7) 14
3.36.22 Port B Out (PBOUT) 14
3.36.23 Port A Out (PAOUT) 14
3.36.24 Port A In (PAIN) 14
The value stored in this register is read in from 8 pins external to the FPGA. It is used for general purpose input. It is read in on any cycle that an SFR is not written to. 14
3.36.25 Timer Byte 0 (TB0) 14
3.36.26 Timer Byte 1 (TB1) 14
3.36.27 Timer Byte 2 (TB2) 15
3.36.28 Timer Byte 3 (TB3) 15
3.36.29 Timer Byte 4 (TB4) 15
3.36.30 Timer Byte 5 (TB5) 15
3.36.31 Timer Byte 6 (TB6) 15
3.36.32 Timer Byte 7 (TB7) 15
3.37 Memory Pointer Guide 15
3.37.1 Stack Pointer 15
3.37.2 X Pointer 15
3.37.3 Y Pointer 15
3.37.4 Z Pointer 16

4. Microarchitecture 16
4.1 Memory Architecture 16

4.2 Pipeline 16
4.3 Diagrams 21

5. Design Conclusion 21

6. Modules & Testbenches 22
6.1 SOC 22
6.1.1 Control 30
6.1.1.1 Hazard Control Unit 30
6.1.1.2 Interrupt Controller 40
6.1.2 Memory I/O 50
6.1.3 Memory 52
6.1.3.1 Program Memory 54
6.1.3.2 Main Memory 54
6.1.3.3 Frame Buffer 54
6.1.3.4 Call Stack 54
6.1.4 Peripherals 54
6.1.3.1 Timer 54
6.1.3.2 VGA Controller 57
6.2 Pipeline 66
6.2.0 Datapath 67
6.2.1 Fetch Stage 78
6.2.1.1 Program Counter 79
6.2.1.2 Program Counter Input Selection Mux 82
6.2.2 IF/ID Register 84
6.2.3 Decode Stage 87
6.2.3.1 Instruction Decode Logic 89
6.2.3.2 Branch Resolution Logic 106
6.2.3.3 ALU Forwarding Logic 110
6.2.3.4 ID/EX Data Input Selection Mux 140
6.2.4 ID/EX Register 141
6.2.5 Execute Stage 144
6.2.5.1 Alu Input Selection Mux 147
6.2.5.2 Alu 150
6.2.5.2.1 Adder 153
6.2.5.2.2 Multiplier 162
6.2.5.2.3 Bit Shifter 163
6.2.5.2.4 Bitwise Logic Unit 165
6.2.5.3 Memory Forwarding Logic 166
6.2.5.4 EX/MEM Data Input Selection Mux 190
6.2.6 EX/MEM Register 193
6.2.7 Memory Stage 196

6.2.7.1 Special Function Register File 199
6.2.7.2 Special Function Register File Input Selection Mux 207
6.2.7.3 Memory Store Data Selection Mux 209
6.2.7.4 Memory Address Selection Mux 211
6.2.7.5 MEM/WB Data Input Selection Mux 212
6.2.8 MEM/WB Register 216
6.2.9 Instruction Word Selection Mux 218
6.2.10 Register File 219

7. Assembler **221**
7.1 zwriscassemble 224
7.2 bin2coe 224

8. Appendices **238**
8.1 A: Opcode Table 240
8.2 B: Microarchitecture Diagram 241
8.3 C: Machine Cycle Diagram 242
8.4 D: Final Test Program 243

1. Introduction

My processor utilizes a 5 stage pipeline to increase the performance of the processor. This was done mainly because, I wanted to design something more complex to challenge myself and reinforce advanced topics in computer architecture. This resulted in a full System on Chip that includes a VGA controller that outputs a 160x120 progressive signal at 12-Bits per pixel and uses a framebuffer tightly coupled to the core. A 64-Bit timer for simple performance evaluation and future plans for PWM modules. My design also contains an interrupt controller for I/O and ensuring that the framebuffer is only written to during the vertical blanking period for prevention of screen tearing. Delving into the core, there are five pipline stages: fetch decode, execute, memory, and writeback. The core is mostly bypassed due to a few oversights on my part when writing the pipeline hazard detection and avoidance modules. Pipelining allows for significantly higher clock frequencies than a single cycle design while maintaining the effective execution latency of the single cycle design.

2. Requirements

The requirements for this project are to design an 8-Bit microprocessor with a 16-Bit address bus using the Verilog Hardware Description Language. That Implements the following instructions and condition codes.

Instruction Name	Mnemonic	Opcode	Operandi (dst, src)
NOP	NOP	00h	----------
ADD	ADD	01h	reg, reg or imm
SUBTRACT	SUB	02h	reg, reg or imm
MULTIPLY	MUL	03h	reg, reg or imm
AND	AND	04h	reg, reg or imm
OR	OR	05h	reg, reg or imm
SHIFT RIGHT	SHR	06h	reg, reg
SHIFT LEFT	SHL	07h	reg, reg
COMPLEMENT	CMP	08h	reg, reg
LOAD	LD	10h	reg, mem or imm
STORE	STR	11h	mem, reg
MOVE	MOV	12h	mem, mem
JUMP	JMP	0Fh	mem (inst), cc
HALT	HLT	1Fh	----------

Condition Code	Abbreviation	Numeric Designation
Always	A	00
Carry	C	01
Zero	Z	10
Negative	N	11

As someone who is interested in making a career out of Verilog, I decided to rearchitect the processor completely as a challenge. My new design will implement the given instructions, except those that operate on operands from memory as that violates the RISC philosophy.

3. Instruction Set Architecture

3.1 No Operation

No Operation - NOP

Simply wastes 1 instruction cycle. This will be the same number of clock cycles as the quickest instruction that actually does something. The effective cycle latency is 1 Cycle.

3.2 Increment

Increment - INC Rx

This instruction increments the given register. The effective cycle latency is 1 Cycle.

Rx <- Rx + 1

3.3 Decrement

Decrement - DEC Rx

This instruction decrements the given register. The effective cycle latency is 1 Cycle.

Rx <- Rx - 1

3.4 Add

Add - ADD Rx, Ry - ADDI Rx, k

This instruction either adds two registers or an immediate value to a register. The effective cycle latency is 1 Cycle.

Rx <- Rx + Ry

Rx <- Rx + k

3.5 Subtract

Subtract - SUB Rx, Ry - SUBI Rx, k

This instruction subtracts two registers or a register and an immediate value. The effective cycle latency is 1 Cycle.

Rx <- Rx - Ry

Rx <- Rx - k

3.6 Compare

Compare - CP Rx, Ry - CPI Rx, k

This instruction subtracts the second operand from the first, but does not store the result. Thus only relevant flags are produced for any subsequent control flow instruction to make use of. The effective cycle latency is 1 Cycle.

3.7 Multiply

Multiply - MUL Rx, Ry - MULI Rx, Rx1, k

This instruction multiplies either two registers or a register and an immediate value. The effective cycle latency is 1 Cycle.

Rx, Ry <- Rx * Ry

Rx, Ry <- Rx * k

3.8 And

And - AND Rx, Ry - ANDI Rx, k

This instruction performs a bitwise and between either two registers or a register and an immediate value. The effective cycle latency is 1 Cycle.

Rx <- Rx & Ry

Rx <- Rx & k

3.9 Or

Or - OR Rx, Ry - ORI Rx, k

This instruction performs a bitwise or between either two registers or a register and an immediate value. The effective cycle latency is 1 Cycle.

Rx <- Rx | Ry

Rx <- Rx | k

3.10 Shift Right

Right Shift - SHR Rx

This instruction shifts the source register right by a single bit storing the result in the destination register. A zero is shifted into the high bit, which is not shifted into the carry flag. The effective cycle latency is 1 Cycle.

Rx <- Rx >> 1

3.11 Shift Left

Left Shift - SHL Rx

This instruction shifts the source register left by a single bit storing the result in the destination register. A zero is shifted into the lowbit, which is not shifted into the carry flag. The effective cycle latency is 1 Cycle.

Rx <- Rx << 1

3.12 Complement

Complement - COM Rx

This instruction will take the two's complement of a given register. The effective cycle latency is 1 Cycle.

Rx <- ~Rx + 1

3.13 Invert

Invert - INV Rx

This instruction inverts all the bits in a given register. The effective cycle latency is 1 Cycle.

Rx <- ~Rx

3.14 Load

Load - LD Rx, (addr) - LDFB Rx, Ry (addr)

This instruction loads a value from a specific memory address into a register or register pair. The effective cycle latency is 1 Cycle.

Rx <- (ptr)

Rx, Ry <- (ptr) (For LDFB 12 bits are loaded, the top byte is zero extended to 16-Bits)

3.15 Load Immediate

Load Immediate - LDI Rx, k

This instruction stores an 8-Bit immediate data value into the specified register. The effective cycle latency is 1 Cycle.

Rx <- k

3.16 Load Program Memory

Load From Program Memory - LPM Rx, Z

This instruction loads the contents of program memory pointed by the value in the Z register and writes it to register Rx. The effective cycle latency is 1 Cycle.

Rx <- (ptr)

3.17 Store

Store - ST (addr), Rx - STFB (addr) Rx, Ry

This instruction stores the value in the source register into the destination register. The effective cycle latency is 1 Cycle.

(addr) <- Rx

(addr) <- Rx, Ry

3.18 Move

Move - MOVR Rx, Ry - MOV (addr0), (addr1)

This instruction moves a value between two addresses in data memory or two registers. The effective cycle latency is 1 Cycle.

Rx <- Ry

(addr0) <- (addr1)

3.19 In

In - IN Rx, SFRy

This instruction reads in a value from one of the 32 special function registers into a general purpose register. The effective cycle latency is 1 Cycle.

Rx <- SFRx

3.20 Out

OUT - OUT SFRx, Rx

This instruction stores the values in a specified GPR into the specified SFR. The effective cycle latency is 1 Cycle.

SFRx <- Rx

3.21 Jump

Jump - JMP apma (absolute program memory address)

This instruction changes the program counter to the absolute program memory address provided in an immediate fashion. The effective cycle latency is 2 Cycles.

PC <- apma

3.22 Branch if Carry Set

Branch If Carry Set - BRCS apma

This instruction sets the program counter to the specified absolute program memory address if the carry flag is set. The effective cycle latency is 2 Cycles.

Carry flag ? PC <- apma : PC <- PC

3.23 Branch if Carry Clear

Branch If Carry Clear - BRCC apma

This instruction sets the program counter to the specified absolute program memory address if the carry flag is cleared. The effective cycle latency is 2 Cycles.

Carry flag ? PC <- PC : PC <- apma

3.24 Branch if Equal

Branch If Equal - BREQ apma

This instruction sets the program counter to the specified absolute program memory address if the zero flag is set. The effective cycle latency is 2 Cycles.

Z flag ? PC <- apma : PC <- PC

3.25 Branch if Not Equal

Branch If Not Equal - BRNE apma

This instruction sets the program counter to the specified absolute program memory address if the zero flag is cleared. The effective cycle latency is 2 Cycles.

Z flag ? PC <- PC : PC <- apma

3.26 Branch if Negative

Branch If Negative - BRNQ apma

This instruction sets the program counter to the specified absolute program memory address if the negative flag is set. The effective cycle latency is 2 Cycles.

Negative flag ? PC <- apma : PC <- PC

3.27 Branch if Positive

Branch If Positive - BRPS apma

This instruction sets the program counter to the specified absolute program memory address if the negative flag is cleared. The effective cycle latency is 2 Cycles.

Negative flag ? PC <- PC : PC <- apma

3.28 Call

Call - CALL apma

This instruction pushes the address of the next instruction onto the stack and then loads the absolute program memory address into the Program Counter. The effective cycle latency is 2 Cycles.

(CSP) <- PC + 1
CSP <- CSP + 1
PC <- apma

3.29 Return

Return - RET

This instruction pops the previously pushed return address into the program counter. The effective cycle latency is 5 Cycles.

PC <- (CSP)
CSP <- CSP - 1

3.30 Return From Interrupt

Return From Interrupt - RETI

This instruction is identical to return except it alerts the interrupt controller that the program has left its interrupt service routine. This allows for nested interrupts. The effective cycle latency is 5 Cycles.

PC <- (CSP)
CSP <- CSP - 1

3.31 Push

Push - PUSH Rx

This instruction pushes the contents of register Rx onto the stack. The effective cycle latency is 1 Cycle.

(SP) <- Rx
SP <- SP - 1 (8-bit registers)

3.32 Pop

Pop - POP Rx

This instruction pops the contents of memory pointed to by the stack pointer and places that value in register Rx. The effective cycle latency is 1 Cycle.

Rx <- (SP)
SP <- SP + 1

3.33 Halt

Halt - HLT

This instruction halts the processor. Only a reset or interrupt can restore the cpu to operation. The effective cycle latency is 1 Cycle.

3.34 Opcode Table

All information regarding the encoding of each instruction may be found in the opcode table found in Appendix A.

3.35 Errata

In the case of instructions that require two register operands, if each of the two instructions preceding it have exactly 1 unique dependency for the originally mentioned instruction, the hazard detection and forwarding units will not detect the hazard and thus the chance for incorrect execution is almost guaranteed unless two NOPS are inserted between the final dependent instruction and the two preceding hazardous writes.

Any pointer increment or decrement does not take effect until the next cycle. Thus, it is advised that the programmer not write sequential memory instructions with increments on the same pointer. They should either add a single NOP, or, if they can perform interleaving of independent memory accesses.3.36.1 Stack Pointer Low (SPL)

3.36 SFR Register Map

3.36.1 Stack Pointer Low (SPL)

Low byte of the 16-Bit stack pointer.

3.36.2 Stack Pointer High (SPH)

High Byte of the 16-Bit stack pointer.

3.36.3 X Pointer Low (XL)

Low byte of the 16-Bit X pointer.

3.36.4 X Pointer High (XH)

High byte of the 16-Bit X pointer.

3.36.5 Y Pointer Low (YL)

Low byte of the 16-Bit Y pointer.

3.36.6 Y Pointer High (YH)

High byte of the 16-Bit Y pointer.

3.36.7 Z Pointer Low (ZL)

Low byte of the 16-Bit Z pointer.

3.36.8 Z Pointer High (ZH)

High byte of the 16-Bit Z pointer.

3.36.9 Timer One Control Register (T1CR)

Bit <0> is a timer enable bit. If it is set, the timer will increment on every clock cycle. Bit <1> is a clear bit. If set, it will clear the value in the timer.

3.36.10 Call Stack Pointer (CSP)

8-Bit call stack pointer. The user can set this, but it does not actually matter so long as no function calls go more than 256 deep.

3.36.11 LED (LED)

This register is directly connected to LED on the board of the Arty S7.

3.36.12 Interrupt Controller Control Register (ICCR)

This register controls the operation of the interrupt controller. If bit <0> is set, the interrupt controller will accept interrupts, otherwise it will ignore any detected interrupt conditions.

3.36.13 General Interrupt Control Register (GICR)

This register is used as a mask register. If bit <0> is set, the interrupt controller will look for interrupt conditions on the vblank interrupt line. If bit <1> is set, the interrupt controller will look for interrupt conditions coming from an illegal opcode exception. If bit <2> is set, the interrupt controller will look for interrupt conditions on the timer compare match line.

3.36.14 Timer Compare Byte 0 (TCB0)

Byte 0 of the value that is compared to the timer value in the timer. The full value is 64-Bits.

3.36.15 Timer Compare Byte 1 (TCB1)

Byte 1 of the value that is compared to the timer value in the timer. The full value is 64-Bits.

3.36.16 Timer Compare Byte 2 (TCB2)

Byte 2 of the value that is compared to the timer value in the timer. The full value is 64-Bits.

3.36.17 Timer Compare Byte 3 (TCB3)

Byte 3 of the value that is compared to the timer value in the timer. The full value is 64-Bits.

3.36.18 Timer Compare Byte 4 (TCB4)

Byte 4 of the value that is compared to the timer value in the timer. The full value is 64-Bits.

3.36.19 Timer Compare Byte 5 (TCB5)

Byte 5 of the value that is compared to the timer value in the timer. The full value is 64-Bits.

3.36.20 Timer Compare Byte 6 (TCB6)

Byte 6 of the value that is compared to the timer value in the timer. The full value is 64-Bits.

3.36.21 Timer Compare Byte 7 (TCB7)

Byte 7 of the value that is compared to the timer value in the timer. The full value is 64-Bits.

3.36.22 Port B Out (PBOUT)

The value stored in this register is output onto 8 pins external to the FPGA. It is used for general purpose output.

3.36.23 Port A Out (PAOUT)

The value stored in this register is output onto 8 pins external to the FPGA. It is used for general purpose output.

3.36.24 Port A In (PAIN)

The value stored in this register is read in from 8 pins external to the FPGA. It is used for general purpose input. It is read in on any cycle that an SFR is not written to.

3.36.25 Timer Byte 0 (TB0)

This register is Byte 0 of the timer’s value. It is read in on any cycle that an SFR is not written to.

3.36.26 Timer Byte 1 (TB1)

This register is Byte 1 of the timer’s value. It is read in on any cycle that an SFR is not written to.

3.36.27 Timer Byte 2 (TB2)

This register is Byte 2 of the timer's value. It is read in on any cycle that an SFR is not written to.

3.36.28 Timer Byte 3 (TB3)

This register is Byte 3 of the timer's value. It is read in on any cycle that an SFR is not written to.

3.36.29 Timer Byte 4 (TB4)

This register is Byte 4 of the timer's value. It is read in on any cycle that an SFR is not written to.

3.36.30 Timer Byte 5 (TB5)

This register is Byte 5 of the timer's value. It is read in on any cycle that an SFR is not written to.

3.36.31 Timer Byte 6 (TB6)

This register is Byte 6 of the timer's value. It is read in on any cycle that an SFR is not written to.

3.36.32 Timer Byte 7 (TB7)

This register is Byte 7 of the timer's value. It is read in on any cycle that an SFR is not written to.

3.37 Memory Pointer Guide

3.37.1 Stack Pointer

This pointer is used only by push and pop instructions. The programmer should intialize it to point to the end of ram at the beginning of their program by loading in 0xFF to the high and low byte.

3.37.2 X Pointer

The X pointer, is a general purpose memory pointer. It may be used to address program memory, main memory, or the framebuffer. After using it you may increment it by using this syntax: X+, rather than this syntax X.

3.37.3 Y Pointer

The Y pointer, is a general purpose memory pointer. It may be used to address program memory, main memory, or the framebuffer. After using it you may increment it by using this syntax: Y+, rather than this syntax Y.

3.37.4 Z Pointer

The Z pointer, is a general purpose memory pointer. It may be used to address program memory, main memory, or the framebuffer. After using it you may increment it by using this syntax: Z+, rather than this syntax Z.

4. Microarchitecture

4.1 Memory Architecture

My processor has a Harvard memory architecture. What that means is that the data and instruction memories are split. In my case, I have one ROM that stores the instructions and any constant data baked into a program. The other parts of the memory structure are the Framebuffer and Main memory. These are both RAM's that are used for two separate things, which I will dive into shortly. All memories can complete a read and write within a single CPU cycle. This is achieved by running the memory clock at exactly twice the frequency of the core. This happens to be the number one performance bottleneck in this architecture.

Program Memory:

The program will be loaded into an FPGA block ram at program time then accessed by the pipeline. This memory is a dual ported 64 kB BRAM. The fetch port will have a 32 bit data bus and a 14 bit address bus. The memory access unit port will have an 8 bit data bus and a 16 bit address bus.

Data Memory:

64 kB of RAM. This is stored on the FPGA. It has a single access port with a 16-bit address bus and an 8-bit data bus. This can only be accessed by the cpu.

Call Stack:

The call stack is a small block ram that stores return addresses. These are 14-bit words. In total, this memory can store up to 256 of those return addresses. Thus it is indexed by an 8-but stack pointer register. Because this memory will only store the call stack, the pointer register shall count up from address 0 which will be clocked in on system reset.

Frame Buffer:

This is a dual ported memory that stores exactly enough data to store a single 160x120 frame with 12-bit per pixel color data. This will be read by the VGA controller and drawn to a display. It will be read and written to by the cpu using load and store instructions.

4.2 Pipeline

My processor utilizes a 5-stage execution pipeline to increase total instruction throughput, and thus overall performance relative to a sequential or single cycle machine. The

five stages are, in this order, Fetch, Decode, Execute, Memory, and Writeback. A special register, called a pipeline register, separates each stage of the pipeline. These registers store every control signal, or data value needed for an instruction to execute properly in the next stage. With that said, I will go into more detailed descriptions of each pipeline stage.

Functional Units.

1. Instruction Word Selection Multiplexor. This module allows the hazard control unit to select between either the output of the fetch port of program memory or its own 32-Bit instruction bus as inputs into the first pipeline register, IF/ID. This feature is used for coordinating stalls and interrupts. Say we need to stall the fetch stage for a cycle, we do not want whatever instruction was being fetched from program memory by the program counter that cycle to enter the pipeline. So, by asserting this multiplexor's select signal, the hazard control unit can insert a nop into the pipeline. It can also insert a call instruction in the case of an interrupt.
2. Register File. This module is a 32 entry 8-Bit register file. It has two read ports, as well as two write ports. This is a reasonably sized register file allowing for longer programs that do not have to touch memory for full operation, improving speed of hand optimized assembly and potentially compiled code without the need to implement the complex hardware algorithms for register renaming.

Stage 1: Fetch

The fetch stage is one of the simpler stages. It only has two modules within it. The program counter, and a multiplexor, controlled by the cpu control state machine, selects the next value of the program counter. There are four: the current value incremented, a return address, a branch target address, or an interrupt vector address. The program counter can also be stalled. In this case it simply keeps the same value.

Functional Units.

1. Program Counter Input Selection Multiplexor. This module is controlled by the pipeline hazard control state machine. Effectively, it is used so that the hazard control unit can insert an arbitrary instruction into the pipeline when the need arises in case of control flow instructions (CALL), interrupts (CALL), or pipeline hazards (NOP).
2. Program Counter. This module is used as the address for the fetch port of program memory. TThe value stored in it is the address, in program memory of the next instruction to be put into the pipeline.It outputs the its current value + 1 for propagating down the pipeline for any potential call instructions it may fetch(return address), that value is also sent to the program counter input selection mux, and is the default value to be placed at the output of that multiplexor. It has a stall signal that may be asserted by the hazard control unit in

order to keep the value stored in the program counter the same across two cycles, effectively stalling the pipeline.

Stage 2: Decode

This stage translates instruction words into operands and control signals to be propagated through the pipeline. It also checks for data dependencies in relation to arithmetic operations and either stalls, or forwards operands from the execute stage or memory stage by generating control signals based on register addresses in subsequent pipeline registers. It also checks the alu flags when a control flow instruction is encountered to know whether to stall on a taken branch instruction or allow the instruction fetched in the branch's decode cycle to propagate through the pipeline. Register file reads also occur in this stage. All register reads are clocked on the negative edge of the clock. This allows writes and reads to the same register to occur within the same clock cycle. This ia a hard requirement for a pipelined processor.

Functional Units.

1. Decode Logic. This module looks at the current instruction and generates any control signal for it that has no potential to cause a pipeline hazard. It also looks for special instructions like return, halt, or an illegal opcode, and alerts the hazard unit and interrupt controller of the events.
2. Branch Resolution Logic. This simply looks at the whether the instruction in IF/ID is a Jump or Branch instruction, based on bits [10:8] in the instruction word and the alu result flags from the operation occurring during that same cycle, determines whether to take a branch. If the branch is taken, It alerts the Hazard Unit incurring a stall.
3. ALU FOrwarding Logic. This module Looks at the instructions in IF/ID, ID/EX, and EX/MEM and determines if any data dependencies exist and either stalls to wait for a load result, or simply choose the selection signals for the alu input selection mux.
4. ID/EX Selection Mux. This determines whether immediate data values, or the register file read values are latched into ID/EX on the next clock cycle.

Stage 3: Execute

The execute stage has two major functions. First it has the Arithmetic Logic Unit which performs all the calculations of the CPU. Second it checks for data hazards related to writes to either memory or the special function register file. If a hazard is found, multiplexor control signals are generated that ensure that the sequential model of execution is presented to the programmer.

Functional Units.

1. Alu Input Mux. This mux can send 1 of 5 data values into each operand port of the alu. ID/EX top and bottom to the top and bottom operand respectively. Both operand ports can take both data values stored in EX/MEM, and MEM/WB.
2. ALU. This module contains a multiplier, bit shifter, adder/subtractor, and bitwise logic unit. It uses a common result bus with the low 2-bits of the instruction word effectively

encoding a selection signal to generate individual output enable signals to each of the four functional units within the alu, ensuring no bus conflicts occur.

3. Memory forwarding logic. This module looks at the instructions in the ID/EX, EXX/MEM, and MEM/WB pipeline registers and decides whether or not data needs to be forwarded to ensure correct execution.
4. EX/MEM Data Input Selection Multiplexor. This module simply selects between the ID/EX data values and the alu result bytes for latching into the data of EX/MEM.

Stage 4: Memory Access

This stage handles all data related memory accesses This is done by sending the memory write enable signal along with a one hot vector that selects which memory interface is being used. This includes tasks such as ensuring that the correct data is sent to the memory blocks depending on specific streams of instructions and their data dependencies. This is done using three multiplexers in this stage. One to ensure that the data being written to the Special Function Register File is correct. One to ensure the data being written to memory is correct. And, finally a multiplexor to ensure that the correct data is latched into the MEM/WB register on the next positive edge of the clock. IT also contains the Special function register file which is used for controlling I/O operations such as stack pointers, and the three architectural memory pointers, X, Y & Z. A multiplexor controlled by bit in the instruction word will control which of the four pointer options will be presented to the memory blocks. Other valuesthe sfr stores are the LED output register, the 64-bit timer value, a 64-bit value for triggering timer compare matches, PWM?, and registers to control the timer, interrupt controller, and which interrupts are enabled. All forwarding capable multiplexes in this stage are controlled by

Functional Units.

1. Special Function Register File:32 8-Bit registers. This contains 4 16-bit pointer registers X, Y, Z, and the stack pointer. It also contains the call stack pointer. The two stack pointers will be incrementable as well as decrementable. It will also contain I/O registers (input and output will be separate) along with peripheral control registers to control timers
2. SFR File Input Multiplexer
   This mux is used to forward either the bottom data value of the ex/mem pipeline or the current data values in MEM/WB and the previous data Values into MEM/WB.
3. Memory Address Multiplexer
   Potential address inputs: X, Y & Z pointers, Stack Pointer. It termines which to use based on bits 18, and 18 in the instruction word stored in EX/MEM.
4. Memory Data Multiplexer
   Potential inputs: EX/MEM top & bottom data, MEM/WB top & bottom, MEM/WB Time - 1 Top & Bottom.
5. MEM/WB register data input Multiplexer.

Potential inputs: EX/MEM top & bottom data, SFR read data, MEM/WB top & bottom, and MEM/WB Time - 1

Stage 5: Writeback

This stage is quite simple relative to the Execute and Memory stages. It simply sends the requisite data, address, and write enable signals to the general purpose register file, along with a return address popped off of the call stack for jumping too upon completion of a return instruction.

Functional Units.

None

Pipeline Registers:

1. IF/ID. This module stores the selected instruction word (hazard unit || program memory), and the address of the instruction directly following it in memory.
2. ID/EX. This module stores the instruction word previously in IF/ID, and the address of the instruction directly following it in memory. It also stores all control signals generated in the decode pipeline stage (Alu Top & Bottom Operand Select, Memory Write Enable, Main Memory, Program Memory, Call Stack , and Frame Buffer Enable, Memory Pointer Control Signals, (these are double latched to ensure the increments or decrements occur after the address has been used, Call Stack & Stack Address Sel (Ensure correct values are pushed and popped), EX/MEM Data Input Select, Register File Write Enable, SFR File Write & Read Enable), along with the data operands.
3. EX/MEM. This module stores the instruction word previously in EX/MEM, and the address of the instruction directly following it in memory. It also stores all control signals generated in the decode pipeline stage (Memory Write Enable, Main Memory, Program Memory, Call Stack, and Frame Buffer Enable, Memory Pointer Control Signals, Call Stack & Stack Address Sel (Ensure correct values are pushed and popped), EX/MEM Data Input Select, Register File Write Enable, SFR File Write & Read Enable), along with the data operands.
4. MEM/WB. This module stores the instruction word previously in EX/MEM, and the address read out of the call stack. It also stores the data outputs of the memory stage and the values previously stored in itself.

Control:

1. Hazard Control Unit. This module is the control state machine for the entire pipeline. It handles stall requests coming from the decode stage. It handles which of the four next addresses gets loaded into the program counter. It handles interrupts, and return instruction sequencing. It can stall any of three pipeline registers, the program counter, IF/ID, or ID/EX. It handles the insertion of a call to an interrupt vector address when the interrupt controller asserts its interrupt signal.

2. Interrupt Controller. This module contains edge detectors for interrupt signals. And, when one asserts will trigger the two state machines to put the controller into an interrupted state. The controller will wait for the hazard unit to acknowledge the interrupt, by checking the state variable for the hazard control unit until it enters its interrupt state. It also ensures that the hazard unit gets the correct vector table address for the interrupt that was asserted.

4.3 Diagrams

For the microarchitecture diagram, see Appendix B, for the Machine Cycle Diagram see Appendix C

5. Design Conclusion

Using the modern FPGA rather than the old lattice part allows a much more integrated fast system. Based on static timing analysis, the highest clock speed this design would reach is approximately 50 MHz. Most instructions will work well over a hundred MHz, but due to a misguided design decision, any branch occurring directly after an arithmetic instruction creates a massive time constraint as it forces the branch resolution logic data outputs to be valid before the negative edge of the clock to ensure the hazard unit sequences a taken branch correctly. Because, I designed the Branch Resolution logic to simply look at the flags asynchronously in the decode stage, while the arithmetic instruction executes in the execute stage, I made it to where every arithmetic operation has be be finished before the negative edge of the clock. This greatly increases cycle time in order to ensure correct operation of branches following arithmetic. I made this decision because it reduced the taken branch penalty to a single cycle, while allowing not taken branches to have no penalty. There are two solutions to this problem. The first, which I would choose, is to move the branch resolution logic to the execute stage and read from the flags register. This has one disadvantage in making the penalty for a taken branch 2 cycles rather than 1 a 50% increase on the effective latency of that instruction, but granting a much lower cycle time. To facilitate this change some slight modifications must be made to the hazard control unit for sequencing the a taken branch and invalidating the two instructions fetch directly after the taken branch. The second option, which is more time and resource intensive, is to add a second ALU into the decode stage and reject the use of a condition code register all together. This would match RISC-V's implementation of branch instructions. It would leave taken branches with a penalty with a penalty of two cycles because two register values would have to be read, operated on, than the results would have to be examined, a lengthy process with high latency. This solution would work well on deeper pipelines that split decode into multiple stages.

The second big issue with my design that limits clock speed is how I access memory. My design must run the memory at twice the frequency of the core to ensure single cycle latency for reads. This means that I limited in core frequency by the fundamental limit of the memory blocks on the Spartan 7 FPGA of around 300 MHz. To improve clock speed each stage that access memory would have to be pipelined, bringing the total depth of the integer pipeline to 7 stages. It would also increase the taken branch penalty to 3 cycles, a serious limiter of performance.

Adding the extra memory stages would not be a performance issue as a new address would be able to be issued to the memory unit every cycle, results would not be valid for one extra cycle but the throughput would remain identical to the current implementation ignoring taken branches which would invalidate three instructions rather than just one. Moving to the data memory access stages, unlike the fetch port, we would not lose any throughput as accesses will still be pipelined. Although arithmetic stalls would incur a two cycle penalty rather than one. Overall, these changes would reduce Instructions Per Clock, but the frequency increases greatly outweigh that loss, conservatively increasing the performance of the core by 50%.

I would highly recommend the Xilinx hardware. The simplicity of having a single board with no external wires that can greatly outperform the older system is excellent. On top of that, the solutions are significantly cheaper. I think the school should should move to the use of Digilent's Basys 3 board which uses a slightly smaller then my Spartan 7 50, Artix 7 35. The basys has a VGA connector built in, a 4 digit 7-segment display, tons of switches, and four pmod connectors, allowing for great flexibility in projects for digital labs. The one downside is the sheer size of the Vivado toolchain which comes in at around 60 GB, and requires 100 GB to be free in order to install it. Overall, switching from the old Lattice parts to Xilinx would bring our students into contact with tools they are significantly more likely to encounter in their future careers.

To wrap up, I am glad I went with the pipelined design. It presented my with a significant challenge, while also spicing up my semester. I would call this project a success, though, I wish I had written more demos to show off my VGA controller, but alas. That is what the summer is for. I may even write a C compiler backend. Thanks for offering the class Professor Maher.

6. Modules & Testbenches

A Note on Simulation Waveforms:
I decided not to embed simulation waveforms directly into this document for a myriad of reasons. Those reasons being, the module and testbench code already consumed 230 pages, I could not find a program to nicely automate the formatting of the vcd files into 8.5x11 inch paper format(i.e. They would look bad), thirdly it would take an inordinate amount of time to screenshot and align them all. Instead, GTKWave offers a file format, .gtkw, to save which signals from a vcd to show. So, I went through every simulation file, pulled up every valid signal and saved those to a .gtkw file. I then organized the vcd's and their respective gtkw's into a hierarchical directory structure where each folder either contains submodule folders and that modules vcd and gtkw, or just the gtkw. To use the gtkw, simply open its respective vcd file in GTKWave then click the file menu and then click "Read Save File" and select the correct .gtkw and you will be presented with the appropriate signals.

6.1 SOC

The following code is the top module of the system that I implement in Xilinx's Vivado toolchain, and then load onto the board. Thus it contains black boxes for clock generation, and

for the memory units. Otherwise it is essentially identical to the cpu.v used for simulation that strips out all the black boxes and instead emulates the memory units in its python testbench.
Code:

```
/*
Module - SOC (System On Chip)
Author - Zach Walden
Last Changed - 3/28/22
Description - This Module is the Top Module for the entire System On chip.
Parameters -
*/

module soc(
      input clk100M,
      input nreset_sel,
      input [7:0] prta_in,
      output [7:0] led,
      output hsync,
      output vsync,
      output [3:0] red,
      output [3:0] green,
      output [3:0] blue,
      output [7:0] prta_out,
      output [7:0] prtb_out
);

      //Instantiate Clock Generation Modules
      wire mem_clk;
      wire core_clk;
      wire vga_clk;
      wire vga_mem_clk;
      wire ila_clk;

      clk_gen sys_clk_gen(
            // Clock out ports
            .core_clk(core_clk),     // output core_clk
            .mem_clk(mem_clk),     // output mem_clk
            .ila_clk(ila_clk),
            // Clock in ports
            .clk_in1(clk100M)
      );          // input clk_in1

      vga_pix_clk_gen vga_clk_gen(
            // Clock out ports
```

```
        .vga_clk(vga_clk),     // output vga_clk
        .vga_mem_clk(vga_mem_clk),     // output vga_mem_clk
        // Clock in ports
        .clk_in1(ila_clk)
    );


    wire [11:0] data_in;
    wire [15:0] addr_in;

    //Instantiate Memory Blocks
    wire [13:0] prog_mem_addra;
    wire [31:0] prog_mem_douta;
    wire [7:0] prog_mem_doutb;
    program_memory prog_mem(
        .clka(mem_clk),    // input wire clka
        .addra(prog_mem_addra),  // input wire [13 : 0] addra
        .douta(prog_mem_douta),  // output wire [31 : 0] douta
        .clkb(mem_clk),    // input wire clkb
        .addrb(addr_in),  // input wire [15 : 0] addrb
        .doutb(prog_mem_doutb)  // output wire [7 : 0] doutb
    );

    wire fb_wena;
    wire [11:0] fb_douta;
    wire fb_wenb = 0;
    wire [14:0] fb_addrb;
    wire [11:0] fb_dinb = 0;
    wire [11:0] fb_doutb;

    frame_buffer frame_buf(
        .clka(mem_clk),    // input wire clka
        .wea(fb_wena),      // input wire [0 : 0] wea
        .addra(addr_in[14:0]),  // input wire [14 : 0] addra
        .dina(data_in),    // input wire [11 : 0] dina
        .douta(fb_douta),  // output wire [11 : 0] douta
        .clkb(vga_mem_clk),    // input wire clkb
        .web(fb_wenb),      // input wire [0 : 0] web
        .addrb(fb_addrb),  // input wire [14 : 0] addrb
        .dinb(fb_dinb),    // input wire [11 : 0] dinb
        .doutb(fb_doutb)  // output wire [11 : 0] doutb
    );
```

```
    wire call_stk_wen;
    wire [7:0] call_stk_addr;
    wire [13:0] call_stk_din;
    wire [13:0] call_stk_dout;

    call_stack call_stk(
          //.clock(mem_clk),    // input wire clka
          //.nreset(nreset),
          .clka(mem_clk),
          .wea(call_stk_wen),      // input wire [0 : 0] wea
          .addra(call_stk_addr),  // input wire [7 : 0] addra
          .dina(call_stk_din),    // input wire [13 : 0] dina
          .douta(call_stk_dout)  // output wire [13 : 0] douta
      );

    wire main_mem_wen;
    wire [7:0] main_mem_dout;
    main_memory main_mem(
          .clka(mem_clk),    // input wire clka
          .wea(main_mem_wen),      // input wire [0 : 0] wea
          .addra(addr_in),  // input wire [15 : 0] addra
          .dina(data_in[7:0]),    // input wire [7 : 0] dina
          .douta(main_mem_dout)  // output wire [7 : 0] douta
    );

    wire call_stk_en;
    wire main_mem_en;
    wire fb_en;
    wire prog_mem_en;
    wire mem_wen;
    //wire [13:0] call_stk_data_in;
    wire [11:0] data_out;
    //Instantiate Memory I/O Buffer
    memory_io mem_buffer(
          //BEGIN interface with memory pipeline stage.
          .call_stk_en(call_stk_en),
          .main_mem_en(main_mem_en),
          .prog_mem_en(prog_mem_en),
          .fb_en(fb_en),
          .mem_wen(mem_wen),
          .data_out(data_out),
          //BEGIN Interface with the call stack.
          .call_stk_wen(call_stk_wen),
```

```
        //BEGIN interface with main memory
        .main_mem_wen(main_mem_wen),
        .main_mem_dout(main_mem_dout),
        //BEGIN interface with program memory.
        .prog_mem_doutb(prog_mem_doutb),
        //BEGIN interface with the framebuffer.
        .frame_buf_wena(fb_wena),
        .frame_buf_douta(fb_douta)
  );

  reg nreset = 0;
  always @ (posedge core_clk)
begin
    if(nreset_sel == 1'b1)
    begin
        nreset <= 0;
    end
    else
    begin
        nreset <= 1;
    end
end

  wire [71:0] sfr_input;
  wire [111:0] sfr_output;
  assign sfr_input[7:0] = prta_in;
  assign prta_out = sfr_output[103:96];
  assign prtb_out = sfr_output[95:88];

wire mem_wb_reti_bit;
  wire interrupt;
  wire [13:0] interrupt_vector_address ;

  wire [3:0] hazard_control_unit_state;
  wire illegal_opcode_exception;
  wire vblank_int;
  wire timer_compare_match;
  //Instantiate Interrupt Controller
  interrupt_controller int_controller(
        .clock(core_clk),
        .nreset(nreset),
        //BEIGN Interupt Signals
        .vblank_int(vblank_int),
```

```
        .illegal_opcode_exception(illegal_opcode_exception),
        .timer_compare_match(),
        //BEIGN Interface with Hazard Control Unit
        .hazard_unit_state(hazard_control_unit_state),
        .interrupt(interrupt),
        .int_vec_addr(interrupt_vector_address),
        //BEGIN Interface with SFR Conntrol Registers
        .control_reg(sfr_output[15:8]),
        .mask_reg(sfr_output[23:16])
    );

    wire ret;
    wire halt;
    wire fetch_stl_req;
    wire dec_stl_req;
    wire take_branch_target;

    wire stall_fetch;
    wire stall_decode;
    wire [3:0] prog_cntr_load_sel;
    wire inst_word_sel;
    wire [31:0] hazard_instruction;
    wire [13:0] hazard_int_addr;

    //Instantiate Hazard Control Unit
    hazard_control_unit hazard_unit(
        .clock(core_clk),
        .nreset(nreset),
        //Inputs
        .ret(ret),
        .halt(halt),
        .fetch_stl_req(fetch_stl_req),
        .dec_stl_req(dec_stl_req),
        .take_branch_target(take_branch_target),
        .interrupt(interrupt),
        .interrupt_vector_address(interrupt_vector_address),
        //Outputs
        .stall_fetch(stall_fetch),
        .stall_decode(stall_decode),
        .prog_cntr_load_sel(prog_cntr_load_sel),
        .inst_word_sel(inst_word_sel),
        .new_inst_word(hazard_instruction),
        .prog_cntr_int_addr(hazard_int_addr),
```

```
        //State
        .control_state(hazard_control_unit_state)
    );

    //Instantiate 64-Bit Timer
    timer hardware_timer(
        .clock(core_clk),
        .nreset(nreset),
        .control_reg(sfr_output[111:104]),
        .timer_compare_value(sfr_output[87:24]),
        .timer_compare_match(timer_compare_match),
        .timer_value(sfr_input[71:8])
    );

    wire [15:0] memwb_data;
    wire [7:0] mem_wb_opcode;
    wire [1:0] reg_file_wen;

    //Instantiate Data Path
    datapath pipeline(
        .clock(core_clk),
        .nreset(nreset),
        //ila interface
        .mem_wb_opcode(mem_wb_opcode),
        .mem_wb_reti_bit(mem_wb_reti_bit),
        .memwb_data(memwb_data),
        .reg_file_wen_ext(reg_file_wen),
        //Memory Interface
        .prog_cntr_val(prog_mem_addra),
        .mem_fetch_instruction(prog_mem_douta),
        .main_mem_en(main_mem_en),
        .prog_mem_en(prog_mem_en),
        .fb_en(fb_en),
        .call_stk_en(call_stk_en),
        .mem_wen(mem_wen),
        .mem_addr(addr_in),
        .call_stk_addr(call_stk_addr),
        .write_data(data_in),
        .read_data(data_out),
        .call_stk_write_data(call_stk_din),
        .call_stk_read_data(call_stk_dout),
        //Hazard Unit Interface
        .stall_fetch(stall_fetch),
```

```
            .stall_decode(stall_decode),
            .hazard_prog_cntr_sel(prog_cntr_load_sel),
            .inst_word_sel(inst_word_sel),
            .hazard_inst_word(hazard_instruction),
            .prog_cntr_int_addr(hazard_int_addr),
            .stall_fetch_req(fetch_stl_req),
            .stall_decode_req(dec_stl_req),
            .halt(halt),
            .take_branch_target(take_branch_target),
            .illegal_opcode_exception(illegal_opcode_exception),
            //To Interrupt Controller
            .return_in_pipeline(ret),
            //SFR I/O Interface
            .sfr_file_in(sfr_input),
            .sfr_file_out(sfr_output)
      );

      assign led = sfr_output[7:0];
    wire trig_ack;
ila_0 ila(
      .clk(ila_clk), // input wire clk


      .trig_in(nreset_sel),// input wire trig_in
      .trig_in_ack(trig_ack),// output wire trig_in_ack
      .probe0(prog_mem_addra), // input wire [13:0]  probe0
      .probe1(prog_mem_douta), // input wire [31:0]  probe1
      .probe2(mem_wen), // input wire [0:0]  probe2
      .probe3(call_stk_en), // input wire [0:0]  probe3
      .probe4(main_mem_en), // input wire [0:0]  probe4
      .probe5(fb_en), // input wire [0:0]  probe5
      .probe6(prog_mem_en), // input wire [0:0]  probe6
      .probe7(addr_in), // input wire [15:0]  probe7
      .probe8(data_in), // input wire [11:0]  probe8
      .probe9(led), // input wire [7:0]  probe9
      .probe10(memwb_data), // input wire [15:0]  probe10
      .probe11(reg_file_wen), // input wire [1:0]  probe11
      .probe12(mem_wb_opcode), // input wire [7:0]  probe12
      .probe13(nreset), // input wire [0:0]  probe13
      .probe14(core_clk) // input wire [0:0]  probe14
);

    wire vga_nreset = 1;
```

```
      //Instantiate VGA Controller
      wire [11:0] pixel_switch;
      vga_controller vga(
            .clock(vga_clk),
            .nreset(vga_nreset),
            .pixel_data(fb_doutb),
            .pixel_addr(fb_addrb),
            .pixel(pixel_switch),
            .h_sync(hsync),
            .v_sync(vsync),
            .v_blank_interupt(vblank_int)
      );

      assign red = pixel_switch[11:8];
      assign green = pixel_switch[7:4];
      assign blue = pixel_switch[3:0];

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("soc.vcd");
  $dumpvars (0,) soc);
  #1;
end
`endif
*/
endmodule
```

6.1.1 Control

6.1.1.1 Hazard Control Unit

Code:

```
/*
Module - Hazard Control Unit.
Author - Zach Walden
Last Changed - 3/28/22, 4/4/22, 4/14/22
Description - This Module is a finite state machine that controls the state
of operation of the pipeline.
Parameters -
```

```
*/

module hazard_control_unit(
	input clock,
	input nreset,
	//Inputs
	input ret,
	input halt,
	input fetch_stl_req,
	input dec_stl_req,
	input take_branch_target,
	input interrupt,
	input [13:0] interrupt_vector_address,
	//Outputs
	output reg stall_fetch = 0,
	output reg stall_decode = 0,
	output reg [3:0] prog_cntr_load_sel = 0,
	output reg inst_word_sel = 0,
	output reg [31:0] new_inst_word = 0,
	output reg [13:0] prog_cntr_int_addr = 0,
	//State
	output [3:0] control_state
);

	reg [3:0] state = 0, next_state = 0;

	always @ (negedge clock)
	begin
		if(nreset == 1'b0)
		begin
			state <= 0;
		end
		else
		begin
			state <= next_state;
		end
	end

	always @ (*)
	begin
		case(state)
			//Normal 4'b0000
			4'h0 :
```

```
                begin
                      //Outputs
                      stall_fetch <= 1'b0;
                      stall_decode <= 1'b0;
                      prog_cntr_load_sel <= 4'b0010;
                      inst_word_sel <= 1'b0;
                      new_inst_word <= 32'h00000000;
                      prog_cntr_int_addr <= 14'h0000;
                      if(ret == 1'b1)
                      begin
                            next_state <= 4'b0101;
                      end
                      else if(halt == 1'b1)
                      begin
                            next_state <= 4'b0001;
                      end
                      else if(take_branch_target == 1'b1)
                      begin
                            next_state <= 4'b1001;
                      end
                      else if(fetch_stl_req == 1'b1)
                      begin
                            next_state <= 4'b0011;
                      end
                      else if(dec_stl_req == 1'b1)
                      begin
                            next_state <= 4'b0100;
                      end
                      else if(interrupt == 1'b1)
                      begin
                            next_state <= 4'b0010;
                      end
                      else
                      begin
                            next_state <= 4'b0000;
                      end
                end
                //Halt 4'b0001
                4'h1 :
                begin
                      //Outputs
                      stall_fetch <= 1'b1;
                      stall_decode <= 1'b0;
```

```
                prog_cntr_load_sel <= 4'b0010;
                inst_word_sel <= 1'b1;
                new_inst_word <= 32'h00000000;
                prog_cntr_int_addr <= 14'h0000;
                if(interrupt == 1'b1)
                begin
                        next_state <= 4'b0010;
                end
                else
                begin
                        next_state <= 4'b0001;
                end
        end
        //Interrupt 4'b0010
        4'h2 :
        begin
                //Outputs
                stall_fetch <= 1'b0;
                stall_decode <= 1'b0;
                prog_cntr_load_sel <= 4'b0100;
                inst_word_sel <= 1'b1;
                new_inst_word[31:18] <= interrupt_vector_address;
                new_inst_word[17:0] <= 18'h00042;
                prog_cntr_int_addr <= interrupt_vector_address;
                //Go To Normal
                next_state <= 4'b0000;
        end
        //Stall Fetch 4'b0011
        4'h3 :
        begin
                //Outputs
                stall_fetch <= 1'b1;
                stall_decode <= 1'b0;
                prog_cntr_load_sel <= 4'b0010;
                inst_word_sel <= 1'b1;
                new_inst_word <= 32'h00000000;
                prog_cntr_int_addr <= 14'h0000;

                if(interrupt == 1'b1)
                begin
                        next_state <= 4'b0010;
                end
                else
```

```
		begin
			next_state <= 4'b0000;
		end
	end
	//Stall Decode 4'b0100
	4'h4 :
	begin
		//Outputs
		stall_fetch <= 1'b1;
		stall_decode <= 1'b1;
		prog_cntr_load_sel <= 4'b0010;
		inst_word_sel <= 1'b0;
		new_inst_word <= 32'h00000000;
		prog_cntr_int_addr <= 14'h0000;
		if(interrupt == 1'b1)
		begin
			next_state <= 4'b0010;
		end
		else
		begin
			next_state <= 4'b0000;
		end
	end
	//Return 1 4'b0101
	4'h5 :
	begin
		//Outputs
		stall_fetch <= 1'b1;
		stall_decode <= 1'b0;
		prog_cntr_load_sel <= 4'b0010;
		inst_word_sel <= 1'b1;
		new_inst_word <= 32'h00000000;
		prog_cntr_int_addr <= 14'h0000;
		//To ret 2
		next_state <= 4'b0110;
	end
	//Return 2 4'b0110
	4'h6 :
	begin
		//Outputs
		stall_fetch <= 1'b1;
		stall_decode <= 1'b0;
		prog_cntr_load_sel <= 4'b0010;
```

```
            inst_word_sel <= 1'b1;
            new_inst_word <= 32'h00000000;
            prog_cntr_int_addr <= 14'h0000;
            //To Return 3
            next_state <= 4'b0111;
      end
      //Return 3 4'b111
      4'h7 :
      begin
            //Outputs
            stall_fetch <= 1'b1;
            stall_decode <= 1'b0;
            prog_cntr_load_sel <= 4'b0010;
            inst_word_sel <= 1'b1;
            new_inst_word <= 32'h00000000;
            prog_cntr_int_addr <= 14'h0000;
            //To Normal
            next_state <= 4'b1000;
      end
      //Return 4 4'b1000
      4'h8 :
      begin
            //Outputs
            stall_fetch <= 1'b0;
            stall_decode <= 1'b0;
            prog_cntr_load_sel <= 4'b1000;
            inst_word_sel <= 1'b1;
            new_inst_word <= 32'h00000000;
            prog_cntr_int_addr <= 14'h0000;
            //To Normal
            next_state <= 4'b0000;
      end
      //Take Branch Target 4'b1001
      4'h9 :
      begin
            //Outputs
            stall_fetch <= 1'b0;
            stall_decode <= 1'b0;
            prog_cntr_load_sel <= 4'b0001;
            inst_word_sel <= 1'b1;
            new_inst_word <= 32'h00000000;
            prog_cntr_int_addr <= 14'h0000;
            //To Normal
```

```
                        next_state <= 4'b0000;
                end
                //Default
                default
                begin
                        //Outputs
                        stall_fetch <= 1'b0;
                        stall_decode <= 1'b0;
                        prog_cntr_load_sel <= 4'b0010;
                        inst_word_sel <= 1'b0;
                        new_inst_word <= 32'h00000000;
                        prog_cntr_int_addr <= 14'h0000;
                        //To Normal
                        next_state <= 4'b0000;
                end
            endcase
      end

      assign control_state = state;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("hazard_control_unit.vcd");
  $dumpvars (0, hazard_control_unit);
  #1;
end
`endif
*/
endmodule
```

Testbench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_hazard_control_unit(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())
```

```
    dut.nreset.value = 1
    dut.ret.value = 0
    dut.halt.value = 0
    dut.fetch_stl_req.value = 0
    dut.dec_stl_req.value = 0
    dut.interrupt.value = 0
    dut.interrupt_vector_address.value = 0x3FFF
    dut.take_branch_target.value = 0

    await RisingEdge(dut.clock)
    await RisingEdge(dut.clock)

    assert dut.stall_fetch.value == 0
    assert dut.stall_decode.value == 0, f"Normal Signal Bad"
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    assert dut.inst_word_sel.value == 0, f"Normal Operation signals not
correct."
    assert dut.control_state.value == 0, f"State not normal"

    #Test The ret sequence. and its immunity to interrupts.
    dut.ret.value = 1

    await RisingEdge(dut.clock)

    dut.ret.value = 0
    dut.interrupt.value = 1

    assert dut.control_state.value == 5, f"State Not Return 1"
    assert dut.stall_fetch.value == 1, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Return 1 signals not correct."

    await RisingEdge(dut.clock)
    assert dut.control_state.value == 6, f"State Not Return 2"
    assert dut.stall_fetch.value == 1, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    dut.inst_word_sel.value == 1, f"Return 2 signals not correct."

    await RisingEdge(dut.clock)
    assert dut.control_state.value == 7, f"State Not Return 3"
```

```
    assert dut.stall_fetch.value == 1, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 8, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Return 3 signals not correct."

    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 0, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    assert dut.inst_word_sel.value == 0, f"Normal Operation signals not
correct."
    assert dut.control_state.value == 0, f"State not normal"

    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 0, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 4, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Bad"
    assert dut.prog_cntr_int_addr.value == 0x3FFF
    assert dut.new_inst_word.value == 0xFFFC0042, f"Interrupt signals not
correct."
    assert dut.control_state.value == 2, f"State not interrupt"

    #Test Halt
    dut.interrupt.value = 0
    dut.halt.value = 1
    await RisingEdge(dut.clock)
    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 1, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Bad"
    assert dut.prog_cntr_int_addr.value == 0, f"Bad"
    assert dut.new_inst_word.value == 0, f"Halt signals not correct."
    assert dut.control_state.value == 1, f"State not Halt"

    dut.halt.value = 0

    dut.fetch_stl_req.value = 1

    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 1, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
```

```
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Bad"
    assert dut.prog_cntr_int_addr.value == 0, f"Bad"
    assert dut.new_inst_word.value == 0, f"Halt signals not correct."
    assert dut.control_state.value == 1, f"State not Halt"

    dut.fetch_stl_req.value = 0
    dut.dec_stl_req.value = 1

    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 1, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Bad"
    assert dut.prog_cntr_int_addr.value == 0, f"Bad"
    assert dut.new_inst_word.value == 0, f"Halt signals not correct."
    assert dut.control_state.value == 1, f"State not Halt"

    dut.dec_stl_req.value = 0
    dut.ret.value = 1

    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 1, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Bad"
    assert dut.prog_cntr_int_addr.value == 0, f"Bad"
    assert dut.new_inst_word.value == 0, f"Halt signals not correct."
    assert dut.control_state.value == 1, f"State not Halt"

    dut.ret.value = 0
    dut.interrupt.value = 1

    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 0, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 4, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Bad"
    assert dut.prog_cntr_int_addr.value == 0x3FFF
    assert dut.new_inst_word.value == 0xFFFC0042, f"Interrupt signals not
correct."
    assert dut.control_state.value == 2, f"State not interrupt"
```

```
    #Test Take Branch Target
    dut.interrupt.value = 0
    await RisingEdge(dut.clock)

    dut.take_branch_target.value = 1

    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 0, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 1, f"Bad"
    assert dut.inst_word_sel.value == 1, f"Bad"
    assert dut.prog_cntr_int_addr.value == 0
    assert dut.new_inst_word.value == 0, f"Interrupt signals not correct."
    assert dut.control_state.value == 8, f"State not Take Branch Target"

    dut.take_branch_target.value = 0

    await RisingEdge(dut.clock)
    assert dut.stall_fetch.value == 0, f"Bad"
    assert dut.stall_decode.value == 0, f"Bad"
    assert dut.prog_cntr_load_sel.value == 2, f"Bad"
    assert dut.inst_word_sel.value == 0, f"Bad"
    assert dut.prog_cntr_int_addr.value == 0
    assert dut.new_inst_word.value == 0, f"Interrupt signals not correct."
    assert dut.control_state.value == 0, f"State not Normal"
```

6.1.1.2 Interrupt Controller

Code:

```
/*

Module -

Author - Zach Walden

Last Changed -

Description -

Parameters -

*/
```

```
module interrupt_controller(
      input clock,
      input nreset,
      //BEIGN Interupt Signals
      input vblank_int,
      input illegal_opcode_exception,
      input timer_compare_match,
      //BEIGN Interface with Hazard Control Unit
      input [3:0] hazard_unit_state,
      output reg interrupt = 0,
      output reg [13:0] int_vec_addr = 0,
      //BEGIN Interface with SFR Conntrol Registers
      input [7:0] control_reg,
      input [7:0] mask_reg
);

      reg vblankint_t = 0;
      reg vblankint_tm1 = 0;

      reg ioeint_t = 0;
      reg ioeint_tm1 = 0;
```

```
reg tcmint_t = 0;
reg tcmint_tm1 = 0;

reg state = 0;

always @ (posedge clock)
begin
      if(nreset == 1'b0)
      begin
            //Zero out edge detect registers
            ioeint_tm1 <= 0;
            ioeint_t <= 0;

            vblankint_tm1 <= 0;
            vblankint_t <= 0;

            tcmint_t <= 0;
            tcmint_tm1 <= 0;

            interrupt <= 0;
            int_vec_addr <= 0;
      end
      else
      begin
```

```
                  ioeint_tm1 = ioeint_t;
                  ioeint_t = illegal_opcode_exception;

                  vblankint_tm1 = vblankint_t;
                  vblankint_t = vblank_int;

                  if(control_reg[0] == 1'b1 && state == 1'b0)
                  begin
                        //Do interrupts.
                        //Always Rising Edge
                        if(vblankint_t == 1'b1 && vblankint_tm1 == 1'b0 &&
mask_reg[0] == 1'b1)
                        begin
                              //Vblank Interrut
                              state <= 1;
                              interrupt <= 1;
                              int_vec_addr <= 14'h0001;
                        end
                        //Always Rising Edge
                        else if(ioeint_t == 1'b1 && ioeint_tm1 == 1'b0 &&
mask_reg[1] == 1'b1)
                        begin
                              //Illegal Opcode Exception
                              state <= 1;
```

```
                              interrupt <= 1;

                              int_vec_addr <= 14'h0002;

                        end

                        //Always Rising Edge

                        else if(tcmint_t == 1'b1 && tcmint_t == 1'b0 &&
mask_reg[2] == 1'b1)

                        begin

                              //Illegal Opcode Exception

                              state <= 1;

                              interrupt <= 1;

                              int_vec_addr <= 14'h0003;

                        end

                        else

                        begin

                              //No Interrupts

                              state <= 0;

                              interrupt <= 0;

                              int_vec_addr <= 14'h0000;

                        end

                  end

                  else if(control_reg[0] == 1'b1 && state == 1'b1 &&
hazard_unit_state != 4'b0010)
```

```
                begin
                        //Keep signals the same.
                        state <= state;
                        interrupt <= interrupt;
                        int_vec_addr <= int_vec_addr;
                end
                else
                begin
                        //No do Interrupts.
                        state <= 0;
                        interrupt <= 0;
                        int_vec_addr <= 14'h0000;
                end
            end
      end


/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("interrupt_controller.vcd");
  $dumpvars (0, interrupt_controller);
```

```
  #1;

end

`endif

*/

endmodule/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/

module interrupt_controller(
      input clock,
      input nreset,
      //BEIGN Interupt Signals
      input vblank_int,
      input illegal_opcode_exception,
      input timer_compare_match,
      //BEIGN Interface with Hazard Control Unit
      input [3:0] hazard_unit_state,
      output reg interrupt = 0,
      output reg [13:0] int_vec_addr = 0,
      //BEGIN Interface with SFR Conntrol Registers
      input [7:0] control_reg,
      input [7:0] mask_reg
);

      reg vblankint_t = 0;
      reg vblankint_tm1 = 0;

      reg ioeint_t = 0;
      reg ioeint_tm1 = 0;

      reg tcmint_t = 0;
      reg tcmint_tm1 = 0;

      reg state = 0;
```

```
	always @ (posedge clock)
	begin
		if(nreset == 1'b0)
		begin
			//Zero out edge detect registers
			ioeint_tm1 <= 0;
			ioeint_t <= 0;

			vblankint_tm1 <= 0;
			vblankint_t <= 0;

			tcmint_t <= 0;
			tcmint_tm1 <= 0;

			interrupt <= 0;
			int_vec_addr <= 0;
		end
		else
		begin
			ioeint_tm1 = ioeint_t;
			ioeint_t = illegal_opcode_exception;

			vblankint_tm1 = vblankint_t;
			vblankint_t = vblank_int;

			if(control_reg[0] == 1'b1 && state == 1'b0)
			begin
				//Do interrupts.
				//Always Rising Edge
				if(vblankint_t == 1'b1 && vblankint_tm1 == 1'b0 &&
mask_reg[0] == 1'b1)
				begin
					//Vblank Interrut
					state <= 1;
					interrupt <= 1;
					int_vec_addr <= 14'h0001;
				end
				//Always Rising Edge
				else if(ioeint_t == 1'b1 && ioeint_tm1 == 1'b0 &&
mask_reg[1] == 1'b1)
				begin
					//Illegal Opcode Exception
```

```
                              state <= 1;
                              interrupt <= 1;
                              int_vec_addr <= 14'h0002;
                        end
                        //Always Rising Edge
                        else if(tcmint_t == 1'b1 && tcmint_t == 1'b0 &&
mask_reg[2] == 1'b1)
                        begin
                              //Illegal Opcode Exception
                              state <= 1;
                              interrupt <= 1;
                              int_vec_addr <= 14'h0003;
                        end
                        else
                        begin
                              //No Interrupts
                              state <= 0;
                              interrupt <= 0;
                              int_vec_addr <= 14'h0000;
                        end
                  end
                  else if(control_reg[0] == 1'b1 && state == 1'b1 &&
hazard_unit_state != 4'b0010)
                  begin
                        //Keep signals the same.
                        state <= state;
                        interrupt <= interrupt;
                        int_vec_addr <= int_vec_addr;
                  end
                  else
                  begin
                        //No do Interrupts.
                        state <= 0;
                        interrupt <= 0;
                        int_vec_addr <= 14'h0000;
                  end
            end
      end


/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
```

```
initial begin
  $dumpfile ("interrupt_controller.vcd");
  $dumpvars (0, interrupt_controller);
  #1;
end
`endif
*/
endmodule
```

Testbench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_interrupt_controller(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.nreset.value = 0
    dut.hazard_unit_state.value = 0
    dut.sfr_output.value = 1

    await FallingEdge(dut.clock)

    dut.nreset.value = 1

    dut.vblank_int.value = 0
    dut.illegal_opcode_exception.value = 0

    await FallingEdge(dut.clock)

    dut.vblank_int.value = 1
    dut.illegal_opcode_exception.value = 0

    await FallingEdge(dut.clock)
    await FallingEdge(dut.clock)
    assert dut.interrupt.value == 1, f"Failed"
    dut.hazard_unit_state.value = 2
    await FallingEdge(dut.clock)
    assert dut.interrupt.value == 0, f"Failed after ack"
```

```
    dut.hazard_unit_state.value = 0

    dut.vblank_int.value = 0
    dut.illegal_opcode_exception.value = 0

    await FallingEdge(dut.clock)

    dut.vblank_int.value = 0
    dut.illegal_opcode_exception.value = 1

    await FallingEdge(dut.clock)
    assert dut.interrupt.value == 1, f"Failed"
    dut.hazard_unit_state.value = 2

    await FallingEdge(dut.clock)
    assert dut.interrupt.value == 0, f"Failed"
```

6.1.2 Memory I/O

Code:

```
/*
Module - Memory I/O
Author - Zach Walden
Last Changed - 2/28/22, 4/2/22
Description - Multiplexes the memory interfaces onto a single bus for the
memory pipeline stage. Abstracts
Parameters -
*/

module memory_io(
      //input clock,
      //BEGIN interface with memory pipeline stage.
      input call_stk_en,
      input main_mem_en,
      input prog_mem_en,
      input fb_en,
      input mem_wen,
      output reg [11:0] data_out,
      //BEGIN Interface with the call stack.
      output call_stk_wen,
      //BEGIN interface with main memory
```

```
      output main_mem_wen,
      input [7:0] main_mem_dout,
      //BEGIN interface with program memory.
      input [7:0] prog_mem_doutb,   //Read Only
      //BEGIN interface with the framebuffer.
      output frame_buf_wena,
      input [11:0] frame_buf_douta
);

      assign call_stk_wen = mem_wen & call_stk_en;

      assign main_mem_wen = mem_wen & main_mem_en;
      assign frame_buf_wena = mem_wen & fb_en;

      always @ (*)
      begin
            if(main_mem_en == 1'b1)
            begin
                  data_out[7:0] <= main_mem_dout;
            end
            else if(prog_mem_en == 1'b1)
            begin
                  data_out[7:0] <= prog_mem_doutb;
            end
            else if(fb_en == 1'b1)
            begin
                  data_out[7:0] <= frame_buf_douta[7:0];
            end
            else
            begin
                  data_out[7:0] <= 8'h00;
            end
      end

      always @ (*)
      begin
            if(fb_en == 1'b1)
            begin
                  data_out[11:8] <= frame_buf_douta[11:8];
            end
            else
            begin
                  data_out[11:8] <= 4'h0;
```

```
            end
        end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("memory_io.vcd");
  $dumpvars (0, memory_io);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_memory_io(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.mem_wen.value = 0

    dut.call_stk_en.value = 1
    #Read Call Stack
    dut.call_stk_data_in.value = 0x3FFF
    dut.call_stk_addr_in.value = 0xFF
    dut.call_stk_dout.value = 0x2AA
    await FallingEdge(dut.clock)
    assert dut.call_stk_addr.value == 0xFF, f"addr ass. failed."
    assert dut.call_stk_wen.value == 0, f"call stack wen ass. failed"
    assert dut.call_stk_data_out.value == 0x2AA, f"call stack read failed"

    #write to call stack
    dut.mem_wen.value = 1
    await FallingEdge(dut.clock)
    assert dut.call_stk_wen.value == 1, f"call stack write failed."
```

```
    assert dut.call_stk_data_in == 0x3FFF, f"data not properly propagating"

    dut.call_stk_en.value = 0

    #Write too the multiplexed memories. Main -> Framebuffer (Check that
only the correct wen is 1)
    dut.data_in.value = 0xFFF
    dut.addr_in.value = 0x1FFFF

    #write to main memory
    dut.main_mem_en.value = 1
    await FallingEdge(dut.clock)
    assert dut.main_mem_wen.value == 1, f"main memory write failed."
    assert dut.main_mem_addr.value == 0x0FFFF, f"Address assignment failed
for main mem"
    assert dut.main_mem_din.value == 0xFF, f"Data ass. failed, main mem"
    dut.main_mem_en.value = 0

    #write to the framebuffer.
    dut.fb_en.value = 1
    await FallingEdge(dut.clock)
    assert dut.frame_buf_wena.value == 1, f"FB write failed"
    assert dut.frame_buf_addra.value == 0x1FFFF, f"FB address ass. failed"
    assert dut.frame_buf_dina.value == 0xFFF, f"FB write data ass. failed"
    dut.fb_en.value = 0


    #read from the three multiplexed memories. Main -> Prog -> Framebuffer
    dut.mem_wen.value = 0
    dut.main_mem_dout.value = 0xFF
    dut.prog_mem_doutb.value = 0xEE
    dut.frame_buf_douta.value = 0xDDD

    #read from main memory
    dut.main_mem_en.value = 1
    await FallingEdge(dut.clock)
    assert dut.main_mem_wen.value == 0, f"main mem read failed"
    assert dut.data_out.value == 0x0FF, f"main mem data out multiplexing
failed"
    dut.main_mem_en.value = 0

    #read from program memory
    dut.prog_mem_en.value = 1
```

```
    await FallingEdge(dut.clock)
    assert dut.prog_mem_addrb.value == 0x0FFFF, f"program memory address
assignment ffailed"
    assert dut.data_out.value == 0x0EE, f"program memory data multiplex
failed"
    dut.prog_mem_en.value = 0

    #read from the framebuffer
    dut.fb_en.value = 1
    await FallingEdge(dut.clock)
    assert dut.frame_buf_wena.value == 0, f"FB read failed"
    assert dut.data_out.value == 0xDDD, f"FB data multiplex Failed"
    dut.fb_en.value = 0

    #read from the instruction fetch port of program memory.
    dut.prog_mem_addr_fetch.value = 0x2AA
    dut.prog_mem_douta.value = 0xFEFEFEFE
    await FallingEdge(dut.clock)
    assert dut.prog_mem_addra.value == 0x2AA, f"Fetch prot address assign
failed"
    assert dut.prog_mem_instruction.value == 0xFEFEFEFE, f"Fetch port data
assignment failed"
```

6.1.3 Memory

6.1.3.1 Program Memory

6.1.3.2 Main Memory

6.1.3.3 Frame Buffer

6.1.3.4 Call Stack

6.1.4 Peripherals

6.1.3.1 Timer

Code:

```
/*
Module - Timer
Author - Zach Walden
```

```
Last Changed - 4/16/22
Description - 64-Bit Timer with an enable bit along with a clear bit.
Synchronous Active Low Reset.
Parameters -
*/

module timer(
      input clock,
      input nreset,
      input [7:0] control_reg, //<7> ,<6> ,<5> ,<4> ,<3> ,<2> ,<1> Clear 1
= Reset, 0 = Normal,<0> Timer Enable 1 = run, 0 = stop
      input [63:0] timer_compare_value,
      output reg timer_compare_match,
      output reg [63:0] timer_value = 0
);

      wire [63:0] timer_inc;

      always @ (posedge clock)
      begin
            if(nreset == 1'b0 || control_reg[1] == 1'b1)
            begin
                  timer_value <= 0;
            end
            else if(control_reg[0] == 1'b1)
            begin
                  timer_value <= timer_inc;
            end
            else
            begin
                timer_value <= timer_value;
            end
            if(timer_value == timer_compare_value)
            begin
                  timer_compare_match <= 1'b1;
            end
            else
            begin
                  timer_compare_match <= 1'b0;
            end
      end

      assign timer_inc = timer_value + 1;
```

```
/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("timer.vcd");
  $dumpvars (0, timer);
  #1;
end
`endif
*/
endmodule
```

Testbench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_timer(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.nreset.value = 1
    dut.control_reg.value = 1

    for i in range(200):
        await FallingEdge(dut.clock)

    assert dut.timer_value != 0, f"Timer Not counting"

    dut.control_reg.value = 0
    dut.nreset.value = 0
    await FallingEdge(dut.clock)
    assert dut.timer_value.value == 0, "Reset Failed"

    dut.control_reg.value = 1
    dut.nreset.value = 1

    cnt = 1
```

```
    for i in range(200):
        await FallingEdge(dut.clock)
        print(hex(cnt))
        assert dut.timer_value.value == cnt, "Timer not Counting
synchrnously"
        cnt += 1

    dut.control_reg.value = 2
    await FallingEdge(dut.clock)
    assert dut.timer_value.value == 0, "Clear Bit failed"
```

6.1.3.2 VGA Controller

VGA Controller Module:

Code:

```
/*
Module - VGA Controller
Author - Zach Walden
Last Changed - 3/13/22
Description - This Module reads through each value in the frame buffer and
displays them using proper vga signalling.
Parameters -
*/

module vga_controller(
      input clock,
      input nreset,
      input [11:0] pixel_data,            //Output of the framebuffer unit
port b.
      output [14:0] pixel_addr,           //Framebuffer Port B address.
      output reg [11:0] pixel = 0,
      output h_sync,
      output v_sync,
      output v_blank_interupt
);

      reg [14:0] memory_addr = 0;

      reg [14:0] row_addr_cache = 0;

      //reg row_cnt = 0;
      reg [1:0] row_cnt = 0;
```

```
wire [14:0] mem_addr_inc;
wire [1:0] row_cnt_inc;

wire hblank;
wire hsync;
wire row_done;
//Instantiate Horizontal Counter
horiz_cntr horizontal_cntr(
      .clock(clock),
      .nreset(nreset),
      .hsync(hsync),
      .hblank(hblank),
      .row_done(row_done)
);

wire vblank;
wire vsync;
wire frame_done;
//Instantiate Vertical Counter
vert_cntr vertical_cnter(
      .row_done(row_done),
      .nreset(nreset),
      .vsync(vsync),
      .vblank(vblank),
      .frm_done(frame_done)
);

always @ (posedge clock)
begin
      if(vblank == 1'b1 || frame_done == 1'b1)
      begin
            memory_addr <= 0;
            row_addr_cache <= 0;
      end
      else if(row_done == 1'b1 && vblank == 1'b0)
      begin
            //row_cnt = ~row_cnt;
            //if(row_cnt == 1'b1)
            row_cnt = row_cnt_inc;
            if(row_cnt == 2'b11)
            begin
                  memory_addr <= row_addr_cache;
```

```
                end
                else if(row_cnt == 2'b10)
                begin
                    memory_addr <= row_addr_cache;
                end
            else if(row_cnt == 2'b01)
            begin
                memory_addr <= row_addr_cache;
            end
                else
                begin
                      row_addr_cache <= memory_addr;
                end
            end
            else if(hblank == 1'b1)
            begin

            end
            else
            begin
                  memory_addr <= mem_addr_inc;
            end

            if(hblank == 1'b0 && vblank == 1'b0)
            begin
                  pixel <= pixel_data;
            end
            else
            begin
                  pixel <= 12'h000;
            end
      end

      assign v_blank_interupt = vblank;

      assign mem_addr_inc = memory_addr + 1;
      assign row_cnt_inc = row_cnt + 1;

      assign pixel_addr = memory_addr;
      assign h_sync = hsync;
      assign v_sync = vsync;

/*
```

```
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("vga_controller.vcd");
  $dumpvars (0, vga_controller);
  #1;
end
`endif
*/
endmodule
```

Testbench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_vga_controller(dut):
    clock = Clock(dut.clock, 80, units="ns")
    cocotb.start_soon(clock.start())

    dut.pixel_data.value = 0xFFF
    dut.nreset.value = 1

    for i in range(4*76800):
        await FallingEdge(dut.clock)
```

Module horiz_cntr:

Code:

```
/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/

module horiz_cntr(
      input clock,
      input nreset,
```

```
      output reg hsync = 1,
      output reg hblank = 0,
      output row_done
);
/*
    9 bits are required for 320 x 240
      reg [8:0] value = 0;

      wire [8:0] value_inc;
*/
    // 8-bits for 160 x 120
    reg [7:0] value = 0;

      wire [7:0] value_inc;

      wire set_hblank;
      wire set_hsync;
      wire clear_hsync;

      always @ (posedge clock)
      begin
            if(nreset == 1'b0)
            begin
                  value <= 0;
                  hblank <= 0;
                  hsync <= 1;
            end
            else
            begin
                  if(row_done == 1'b1)
                  begin
                        hblank <= 0;
                        value <= 0;
                  end
                  else
                  begin
                        value <= value_inc;
                  end
                  if(set_hblank == 1'b1)
                  begin
                        hblank <= 1;
                  end
                  else
```

```
                begin

                end
                if(set_hsync == 1'b1)
                begin
                      hsync <= 0;
                end
                else
                begin

                end
                if(clear_hsync == 1'b1)
                begin
                      hsync <= 1;
                end
                else
                begin

                end
            end
      end

      assign value_inc = value + 1;

/*
    //Constants for 320 x 240
      assign set_hblank = value[0] & value[1] & value[2] & value[3] &
value[4] & value[5] & ~value[6] & ~value[7] & value[8];     //319 -> 159
//    assign set_hsync = value[0] & value[1] & value[2] & ~value[3] &
~value[4] & ~value[5] & value[6] & ~value[7] & value[8];    //327
//    assign clear_hsync = value[0] & value[1] & value[2] & ~value[3] &
value[4] & value[5] & value[6] & ~value[7] & value[8];      //375
//    assign row_done = value[0] & value[1] & value[2] & value[3] &
~value[4] & ~value[5] & ~value[6] & value[7] & value[8];    //399


//    assign set_hblank = ~value[0] & ~value[1] & ~value[2] & ~value[3] &
~value[4] & ~value[5] & value[6] & ~value[7] & value[8];    //320 -> 160
      assign set_hsync = ~value[0] & ~value[1] & ~value[2] & value[3] &
~value[4] & ~value[5] & value[6] & ~value[7] & value[8];    //328 -> 164
      assign clear_hsync = ~value[0] & ~value[1] & ~value[2] & value[3] &
value[4] & value[5] & value[6] & ~value[7] & value[8];      //376 -> 188
      assign row_done = ~value[0] & ~value[1] & ~value[2] & ~value[3] &
```

```
value[4] & ~value[5] & ~value[6] & value[7] & value[8];     //400 -> 200
*/
            assign set_hblank = value[0] & value[1] & value[2] & value[3] &
value[4] & ~value[5] & ~value[6] & value[7];    //319 -> 159
//    assign set_hsync = value[0] & value[1] & value[2] & ~value[3] &
~value[4] & ~value[5] & value[6] & ~value[7] & value[8];    //327
//    assign clear_hsync = value[0] & value[1] & value[2] & ~value[3] &
value[4] & value[5] & value[6] & ~value[7] & value[8];      //375
//    assign row_done = value[0] & value[1] & value[2] & value[3] &
~value[4] & ~value[5] & ~value[6] & value[7] & value[8];    //399


//    assign set_hblank = ~value[0] & ~value[1] & ~value[2] & ~value[3] &
~value[4] & ~value[5] & value[6] & ~value[7] & value[8];    //320 -> 160
      assign set_hsync = ~value[0] & ~value[1] & value[2] & ~value[3] &
~value[4] & value[5] & ~value[6] & value[7];    //328 -> 164
      assign clear_hsync = ~value[0] & ~value[1] & value[2] & value[3] &
value[4] & value[5] & ~value[6] & value[7];     //376 -> 188
      assign row_done = ~value[0] & ~value[1] & ~value[2] & value[3] &
~value[4] & ~value[5] & value[6] & value[7];    //400 -> 200


/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("horiz_cntr.vcd");
  $dumpvars (0, horiz_cntr);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_horiz_cntr(dut):
```

```
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    for i in range (410):
        await FallingEdge(dut.clock)
```

Module vert_cntr:

Code:

```
/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/

module vert_cntr(
      input row_done,
      input nreset,
      output reg vsync = 1,
      output reg vblank = 0,
      output frm_done
);

      reg [9:0] value = 0;

      wire [9:0] value_inc;

      wire set_vblank;
      wire set_vsync;
      wire clear_vsync;
      wire frame_done;

      always @ (negedge row_done)
      begin
            if(nreset == 1'b0)
            begin
                  value <= 0;
                  vblank <= 0;
                  vsync <= 1;
            end
            else
```

```
            begin
                  if(frame_done == 1'b1)
                  begin
                        vblank <= 0;
                        value <= 0;
                  end
                  else
                  begin
                        value <= value_inc;
                  end
                  if(set_vblank == 1'b1)
                  begin
                        vblank <= 1;
                  end
                  else
                  begin

                  end
                  if(set_vsync == 1'b1)
                  begin
                        vsync <= 0;
                  end
                  else
                  begin

                  end
                  if(clear_vsync == 1'b1)
                  begin
                        vsync <= 1;
                  end
                  else
                  begin

                  end
            end
      end

      assign frm_done = frame_done;

      assign value_inc = value + 1;

      assign set_vblank = value[0] & value[1] & value[2] & value[3] &
value[4] & ~value[5] & value[6] & value[7] & value[8] & ~value[9];
```

```
      //479
//    assign set_vsync = value[0] & ~value[1] & ~value[2] & value[3] &
~value[4] & value[5] & value[6] & value[7] & value[8] & ~value[9];
      //489
//    assign clear_vsync = value[0] & value[1] & ~value[2] & value[3] &
~value[4] & value[5] & value[6] & value[7] & value[8] & ~value[9];
      //491
//    assign frame_done = ~value[0] & ~value[1] & value[2] & value[3] &
~value[4] & ~value[5] & ~value[6] & ~value[7] & ~value[8] & value[9];
//524

//    assign set_vblank = ~value[0] & ~value[1] & ~value[2] & ~value[3] &
~value[4] & value[5] & value[6] & value[7] & value[8] & ~value[9];
      //480
      assign set_vsync = ~value[0] & value[1] & ~value[2] & value[3] &
~value[4] & value[5] & value[6] & value[7] & value[8] & ~value[9];
      //490
      assign clear_vsync = ~value[0] & ~value[1] & value[2] & value[3] &
~value[4] & value[5] & value[6] & value[7] & value[8] & ~value[9];
      //492
      assign frame_done = value[0] & ~value[1] & value[2] & value[3] &
~value[4] & ~value[5] & ~value[6] & ~value[7] & ~value[8] & value[9];
//525

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("vert_cntr.vcd");
  $dumpvars (0, vert_cntr);
  #1;
end
`endif
*/
endmodule
```

Testbench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge
```

```
@cocotb.test()
async def test_vert_cntr(dut):
    clock = Clock(dut.row_done, 10, units="ns")
    cocotb.start_soon(clock.start())

    for i in range(550):
        await FallingEdge(dut.row_done)
```

6.2 Pipeline

6.2.0 Datapath

Code:

```
/*
Module - Datapath
Author - Zach Walden
Last Changed - 3/28/22
Description - This module contains the CPU's Datapath
Parameters -
*/

module datapath(
      input clock,
      input nreset,
      //ila interface
      output [7:0] mem_wb_opcode,
      output mem_wb_reti_bit,
      output [15:0] memwb_data,
      output [1:0] reg_file_wen_ext,
      //Memory Interface
      output [13:0] prog_cntr_val,
      input [31:0] mem_fetch_instruction,
      output main_mem_en,
      output prog_mem_en,
      output fb_en,
      output call_stk_en,
      output mem_wen,
      output [15:0] mem_addr,
      output [7:0] call_stk_addr,
      output [11:0] write_data,
      input [11:0] read_data,
```

```
      output [13:0] call_stk_write_data,
      input [13:0] call_stk_read_data,
      //Hazard Unit Interface
      input stall_fetch,
      input stall_decode,
      input [3:0] hazard_prog_cntr_sel,
      input inst_word_sel,
      input [31:0] hazard_inst_word,
      input [13:0] prog_cntr_int_addr,
      output stall_fetch_req,
      output stall_decode_req,
      output halt,
      output take_branch_target,
      output illegal_opcode_exception,
      output return_in_pipeline,
      //SFR I/O Interface
      input [71:0] sfr_file_in,
      output [111:0] sfr_file_out
);

      wire [1:0] reg_file_wen;
      wire [1:0] reg_file_ren;
      wire [15:0] reg_file_rd_data;
      wire [15:0] reg_file_wr_data;
      wire [9:0] reg_file_rd_addr;
      wire [9:0] reg_file_wr_addr;
      //Istaniate Register File
      register_file reg_file(
            .clock(clock),
            .nreset(nreset),
            .wr_en(reg_file_wen),
            .rd_en(reg_file_ren),
            .wr_addr(reg_file_wr_addr),
            .rd_addr(reg_file_rd_addr),
            .data_in(reg_file_wr_data),
            .data_out(reg_file_rd_data)
      );

      wire [31:0] if_id_inst;
      //Instantiate Instruction Word Selection Mux
      inst_word_sel_mux fetch_inst_word_sel_mux(
            .sel(inst_word_sel),
            .mem_inst_word(mem_fetch_instruction),
```

```
        .hazard_unit_inst_word(hazard_inst_word),
        .inst_word_out(if_id_inst)
    );

    wire [13:0] fetch_return_address;
    wire [13:0] ret_addr_wb;
    //Instantiate Fetch Stage
    fetch fetch_stage(
        .clock(clock),
        .nreset(nreset),
        .stall(stall_fetch),
        .prog_mem_fetch_read_addr(prog_cntr_val),
        .prog_cntr_input_sel(hazard_prog_cntr_sel),
        .branch_target_address(if_id_inst_out[31:18]),
        .interrupt_branch_addr(prog_cntr_int_addr),
        .ret_addr_mem(ret_addr_wb),
        .ret_addr_out(fetch_return_address)
    );

    wire [31:0] if_id_inst_out;

    wire [13:0] if_id_ret_addr;
    wire stall_ifid;
    assign stall_ifid = stall_fetch & ~inst_word_sel;
    //Instantiate IF/ID Pipeline Register
    if_id if_id_register(
        .clock(clock),
        .nreset(nreset),
        .stall(stall_ifid),
        .instruction_in(if_id_inst),
        .instruction_out(if_id_inst_out),
        .return_addr_in(fetch_return_address),
        .return_addr_out(if_id_ret_addr)
    );

    assign reg_file_rd_addr = if_id_inst_out[17:8];

    wire [31:0] id_ex_instruction_out;
    wire [31:0] ex_mem_instruction_out;
    wire [31:0] mem_wb_instruction_out;

    wire [2:0] alu_flags;
    wire [4:0] alu_top_sel;
```

```
	wire [4:0] alu_bot_sel;

	wire [1:0] ex_mem_data_input_sel_dec;

	wire mem_wen_dec;

	wire main_mem_en_dec;
	wire fb_en_dec;
	wire call_stk_en_dec;
	wire prog_mem_en_dec;

	wire [6:0] mem_ptr_ctl_dec;

	wire [1:0] reg_file_wen_dec;
	wire [1:0] sfr_wren_dec;

	wire [15:0] id_ex_data;
	//Instantiate Decode Stage
	decode decode_stage(
		.clock(clock),
		.nreset(nreset),
		//BEGIN interface definition with General Purpose Register
File.
		.reg_file_ren(reg_file_ren),
		.reg_file_data_top(reg_file_rd_data[15:8]),
		.reg_file_data_bot(reg_file_rd_data[7:0]),
		//BEGIN Interface with IF/ID pipeline register
		.take_branch_target(take_branch_target),
		.instruction_word(if_id_inst_out),
		//BEGIN Interface with ID/EX pipeline register.
		.id_ex_instruction(id_ex_instruction_out),
		.ex_mem_instruction(ex_mem_instruction_out),
		.alu_flags(alu_flags),
		.alu_top_sel(alu_top_sel),
		.alu_bot_sel(alu_bot_sel),
		.ex_mem_data_input_sel(ex_mem_data_input_sel_dec),
		.mem_wen(mem_wen_decode),
		.main_memory_en(main_mem_en_dec),
		.fb_en(fb_en_dec),
		.call_stack_en(call_stk_en_dec),
		.prog_mem_en(prog_mem_en_dec),
		.mem_ptr_ctl(mem_ptr_ctl_dec),
		.reg_file_wen(reg_file_wen_dec),
```

```
        .sfr_file_wren(sfr_wren_dec),
        .id_ex_data_top(id_ex_data[15:8]),
        .id_ex_data_bot(id_ex_data[7:0]),
        //BEGIN interface to hazard unit.
        .stall_fetch(stall_fetch_req),
        .stall_decode(stall_decode_req),
        .halt(halt),
        .illegal_opcode_exception(illegal_opcode_exception),
        .return_in_pipeline(return_in_pipeline)
    );

    wire [4:0] alu_top_sel_out;
    wire [4:0] alu_bot_sel_out;

    wire mem_wen_id_ex;

    wire main_mem_en_idex;
    wire fb_en_idex;
    wire call_stk_en_idex;
    wire prog_mem_en_idex;

    wire [6:0] mem_ptr_ctl_idex;
    wire call_stk_addr_sel_idex;
    wire stk_addr_sel_idex;

    wire [1:0] ex_mem_data_input_sel_idex;

    wire [1:0] reg_file_wen_idex;
    wire [1:0] sfr_wren_idex;

    wire [15:0] id_ex_data_out;

    wire [13:0] ret_addr_idex;
    //Instantiate ID/EX Pipeline Register
    id_ex id_ex_register(
        .clock(clock),
        .nreset(nreset),
        .stall(stall_decode),
        .alu_top_select_in(alu_top_sel),
        .alu_top_select_out(alu_top_sel_out),
        .alu_bot_select_in(alu_bot_sel),
        .alu_bot_select_out(alu_bot_sel_out),
        .id_ex_top_in(id_ex_data[15:8]),
```

```
        .id_ex_top_out(id_ex_data_out[15:8]),
        .id_ex_bot_in(id_ex_data[7:0]),
        .id_ex_bot_out(id_ex_data_out[7:0]),
        .instruction_in(if_id_inst_out),
        .instruction_out(id_ex_instruction_out),
        .mem_wen_in(mem_wen_decode),
        .mem_wen_out(mem_wen_id_ex),
        .main_memory_enable_in(main_mem_en_dec),
        .main_memory_enable_out(main_mem_en_idex),
        .frame_buffer_enable_in(fb_en_dec),
        .frame_buffer_enable_out(fb_en_idex),
        .call_stack_enable_in(call_stk_en_dec),
        .call_stack_enable_out(call_stk_en_idex),
        .prog_mem_enable_in(prog_mem_en_dec),
        .prog_mem_enable_out(prog_mem_en_idex),
        .mem_ptr_ctl_in(mem_ptr_ctl_dec),
        .mem_ptr_ctl_out(mem_ptr_ctl_idex),
        .stk_addr_sel_in(mem_ptr_ctl_dec[1]),
        .stk_addr_sel_out(stk_addr_sel_idex),
        .call_stk_addr_sel_in(mem_ptr_ctl_dec[2]),
        .call_stk_addr_sel_out(call_stk_addr_sel_idex),
        .ex_mem_data_input_sel_in(ex_mem_data_input_sel_dec),
        .ex_mem_data_input_sel_out(ex_mem_data_input_sel_idex),
        .reg_file_wen_in(reg_file_wen_dec),
        .reg_file_wen_out(reg_file_wen_idex),
        .sfr_file_wren_in(sfr_wren_dec),
        .sfr_file_wren_out(sfr_wren_idex),
        .call_addr_in(if_id_ret_addr),
        .call_addr_out(ret_addr_idex)
    );

    wire [15:0] ex_mem_data_out;

    wire [15:0] execute_data_out;

    wire [4:0] sfr_input_sel_ex;
    wire [4:0] mem_str_data_sel_top_ex;
    wire [4:0] mem_str_data_sel_bot_ex;
    wire [3:0] mem_wb_data_sel_top_ex;
    wire [6:0] mem_wb_data_sel_bot_ex;
    //Instantiate Execute Stage
    execute execute_stage(
        .clock(clock),
```

```
            .nreset(nreset),
            //BEGIN interface with ID/EX pipeline register.
            .alu_top_sel(alu_top_sel_out),
            .alu_bot_sel(alu_bot_sel_out),
            .data_in_top(id_ex_data_out[15:8]),
            .data_in_bot(id_ex_data_out[7:0]),
            .instruction(id_ex_instruction_out),
            .ex_mem_data_input_sel(ex_mem_data_input_sel_idex),
            .flags_out(alu_flags),
            //BEGIN interface with EX/MEM pipeline register.
            .ex_mem_operation(ex_mem_instruction_out),
            .mem_wb_operation(mem_wb_instruction_out),
            .ex_mem_data_top(ex_mem_data_out[15:8]),
            .ex_mem_data_bot(ex_mem_data_out[7:0]),
            .mem_wb_data_top(mem_wb_data_out[15:8]), //Add this as a data
passtrough in EX/MEM register().
            .mem_wb_data_bot(mem_wb_data_out[7:0]),
            .data_out_top(execute_data_out[15:8]),
            .data_out_bot(execute_data_out[7:0]),
            .sfr_input_sel(sfr_input_sel_ex),
            .mem_str_data_sel_top(mem_str_data_sel_top_ex),
            .mem_str_data_sel_bot(mem_str_data_sel_bot_ex),
            .mem_wb_data_sel_top(mem_wb_data_sel_top_ex),
            .mem_wb_data_sel_bot(mem_wb_data_sel_bot_ex)
      );

      wire [6:0] mem_ptr_ctl_exmem;
      wire call_stk_addr_sel_exmem;
      wire stk_addr_sel_exmem;

      wire [1:0] reg_file_wen_exmem;
      wire [1:0] sfr_wren_exmem;

      wire [4:0] sfr_input_sel_exmem;
      wire [4:0] mem_str_data_sel_top_exmem;
      wire [4:0] mem_str_data_sel_bot_exmem;
      wire [3:0] mem_wb_data_sel_top_exmem;
      wire [6:0] mem_wb_data_sel_bot_exmem;
      //Instantiate EX/MEM Pipeline Register
      ex_mem ex_mem_register(
            .clock(clock),
            .nreset(nreset),
            .data_top_in(execute_data_out[15:8]),
```

```
        .data_top_out(ex_mem_data_out[15:8]),
        .data_bot_in(execute_data_out[7:0]),
        .data_bot_out(ex_mem_data_out[7:0]),
        .instruction_in(id_ex_instruction_out),
        .instruction_out(ex_mem_instruction_out),
        .mem_wen_in(mem_wen_id_ex),
        .mem_wen_out(mem_wen),
        .main_memory_enable_in(main_mem_en_idex),
        .main_memory_enable_out(main_mem_en),
        .frame_buffer_enable_in(fb_en_idex),
        .frame_buffer_enable_out(fb_en),
        .call_stack_enable_in(call_stk_en_idex),
        .call_stack_enable_out(call_stk_en),
        .prog_mem_enable_in(prog_mem_en_idex),
        .prog_mem_enable_out(prog_mem_en),
        .mem_ptr_ctl_in(mem_ptr_ctl_idex),
        .mem_ptr_ctl_out(mem_ptr_ctl_exmem),
        .call_stk_addr_sel_in(call_stk_addr_sel_idex),
        .call_stk_addr_sel_out(call_stk_addr_sel_exmem),
        .stk_addr_sel_in(stk_addr_sel_idex),
        .stk_addr_sel_out(stk_addr_sel_exmem),
        .mem_wb_data_sel_top_in(mem_wb_data_sel_top_ex),
        .mem_wb_data_sel_top_out(mem_wb_data_sel_top_exmem),
        .mem_wb_data_sel_bot_in(mem_wb_data_sel_bot_ex),
        .mem_wb_data_sel_bot_out(mem_wb_data_sel_bot_exmem),
        .sfr_file_input_sel_in(sfr_input_sel_ex),
        .sfr_file_input_sel_out(sfr_input_sel_exmem),
        .mem_str_data_sel_top_in(mem_str_data_sel_top_ex),
        .mem_str_data_sel_top_out(mem_str_data_sel_top_exmem),
        .mem_str_data_sel_bot_in(mem_str_data_sel_bot_ex),
        .mem_str_data_sel_bot_out(mem_str_data_sel_bot_exmem),
        .reg_file_wen_in(reg_file_wen_idex),
        .reg_file_wen_out(reg_file_wen_exmem),
        .sfr_file_wren_in(sfr_wren_idex),
        .sfr_file_wren_out(sfr_wren_exmem),
        .call_addr_in(ret_addr_idex),
        .call_addr_out(call_stk_write_data)
    );

    wire [15:0] memory_data_out;
    wire [15:0] mem_wb_tm1_data_out;
    //Instantiate Memory Stage
    memory memory_stage(
```

```
            .clock(clock),
            .nreset(nreset),
            //BEGIN interface with EX/MEM pipeline register
            .data_in_top(ex_mem_data_out[15:8]),
            .data_in_bot(ex_mem_data_out[7:0]),
            .instruction(ex_mem_instruction_out),
            .mem_wb_data_input_sel_top(mem_wb_data_sel_top_exmem),
            .mem_wb_data_input_sel_bot(mem_wb_data_sel_bot_exmem),
            .sfr_file_input_sel(sfr_input_sel_exmem),
            .mem_ptr_ctl(mem_ptr_ctl_exmem),
            .call_stk_addr_sel(call_stk_addr_sel_exmem),
            .stk_addr_sel(stk_addr_sel_exmem),
            .mem_str_data_input_sel_top(mem_str_data_sel_top_exmem),
            .mem_str_data_input_sel_bot(mem_str_data_sel_bot_exmem),
            .sfr_file_wren(sfr_wren_exmem),
            //BEGIN interface with MEM/WB pipeline register
            .data_out_top(memory_data_out[15:8]),
            .data_out_bot(memory_data_out[7:0]),
            .mem_wb_top(mem_wb_data_out[15:8]),
            .mem_wb_bot(mem_wb_data_out[7:0]),
            .mem_wb_tm1_top(mem_wb_tm1_data_out[15:8]),
            .mem_wb_tm1_bot(mem_wb_tm1_data_out[7:0]),
            //BEGIN interface with memory i/o unit
            .address(mem_addr),
            .call_stack_ptr(call_stk_addr),
            //.stack_ptr(),
            .mem_read_data(read_data),
            .mem_write_data(write_data),
            //BEGIN I/O interface
            .sfr_file_in(sfr_file_in),                          //These
are enormus.
            .sfr_file_out(sfr_file_out)
      );

      wire [15:0] mem_wb_data_out;
      //Instantiate MEM/WB Pipline Register
      mem_wb mem_wb_register(
            .clock(clock),
            .nreset(nreset),
            .data_top_in(memory_data_out[15:8]),
            .data_top_out(mem_wb_data_out[15:8]),
            .data_bot_in(memory_data_out[7:0]),
            .data_bot_out(mem_wb_data_out[7:0]),
```

```
            .data_tm1_top(mem_wb_tm1_data_out[15:8]),
            .data_tm1_bot(mem_wb_tm1_data_out[7:0]),
            .instruction_in(ex_mem_instruction_out),
            .instruction_out(mem_wb_instruction_out),
            .reg_file_wen_in(reg_file_wen_exmem),
            .reg_file_wen_out(reg_file_wen),
            .ret_addr_in(call_stk_read_data),                    //From
Call Stk
            .ret_addr_out(ret_addr_wb)
      );

      assign reg_file_wr_data = mem_wb_data_out;
      assign memwb_data = mem_wb_data_out;
      assign reg_file_wen_ext = reg_file_wen;

      assign reg_file_wr_addr = mem_wb_instruction_out[17:8];

      assign mem_wb_opcode = mem_wb_instruction_out[7:0];
      assign mem_wb_reti_bit = mem_wb_instruction_out[20];

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("datapath.vcd");
  $dumpvars (0, datapath);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_datapath(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())
```

```
#Open Test Program.coe

testProg = open("test.coe", "r")
lines = testProg.readlines()
#Define Program Memory
prog_mem = []
i = 2
for a in range(16384):
    if(a == 16383):
        prog_mem.append(int("0x" + (lines[i].split(";"))[0], 16))
        break
    prog_mem.append(int("0x" + (lines[i].split(","))[0], 16))
    i += 1

main_mem = []
#Deine Main Memory
for i in range(65536):
    main_mem.append(0)

call_stk = []
#Define The Call Stack
for i in range(256):
    call_stk.append(0)

frame_buf = []
#Define The Framebuffer
for i in range(19200):
    frame_buf.append(0)

#Begin testing
dut.nreset.value = 0
dut.mem_fetch_instruction.value = 0
dut.read_data.value = 0
dut.call_stk_read_data.value = 0
dut.stall_fetch.value = 0
dut.stall_decode.value = 0
dut.hazard_prog_cntr_sel.value = 1
dut.inst_word_sel.value = 0
dut.hazard_inst_word.value = 0
dut.prog_cntr_int_addr.value = 0
dut.sfr_file_in.value = 0
await FallingEdge(dut.clock)
```

```
    dut.nreset.value = 1

    for i in range(250):
        dut.mem_fetch_instruction.value = prog_mem[dut.prog_cntr_val.value]
        await FallingEdge(dut.clock)
```

6.2.1 Fetch Stage

Code:

```
/*
Module - Fetch Pipeline Stage
Author - Zach Walden
Last Changed - 2/21/22
Description - Thin wrapper around my program_counter module so as to meet
code structure conventions.
Parameters -
*/

module fetch(
      input clock,
      input nreset,
      input stall,
      output [13:0] prog_mem_fetch_read_addr,
      input [3:0] prog_cntr_input_sel,          //This signal will come from
the hazard unit.
      //Begin interface definition with the IF/ID pipeline register.
      input [13:0] branch_target_address,
      input [13:0] interrupt_branch_addr,
      input [13:0] ret_addr_mem,
      output [13:0] ret_addr_out
);

      wire [13:0] next_prog_cntr;
      wire [13:0] prog_cntr_load_val;
      //instantiate prog_cntr_load_sel_mux
      prog_cntr_input_sel_mux input_mux(
            //.clock(clock),
            .sel_signals(prog_cntr_input_sel),
            //.sel_signals(sel_prog_counter),
            .next_prog_cntr(next_prog_cntr),
            .branch_target_addr(branch_target_address),
```

```
            .int_branch_addr(interrupt_branch_addr),
            .ret_addr(ret_addr_mem),
            .prog_cntr_load_val(prog_cntr_load_val)
      );

      //Instantiate program_counter
      program_counter prog_cntr(
            .clock(clock),
            .nreset(nreset),
            .stall(stall),
            .load_value(prog_cntr_load_val),
            .next_prog_cntr(next_prog_cntr),
            .prog_mem_addr_fetch(prog_mem_fetch_read_addr)
      );

      assign ret_addr_out = next_prog_cntr;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("fetch.vcd");
  $dumpvars (0, fetch);
  #1;
end
`endif
*/
endmodule
```

6.2.1.1 Program Counter

Code:

```
/*
Module - Program Counter
Author - Zach Walden
Last Changed - 2/1/22, 4/1/22
Description - Program Counter, If fetch is not being stalled by the hazard
controller, it increments the address every cycle "fetching" the next
instruction. It is resettable. It also supports a parallel load feature.
The parallel load bus will be connected to the output of a multiplexor to
that, if a load is required, selects a new input from the decode pipeline
stage or from the output of the call stack.
```

```
Parameters - input stall - stall signal. This signal will prevent an
increment if asserted, thus "stalling" the fetch stage.
          input take_branch_target - This signal in combination with a
2X14 to 1 multiplexor to select the proper new program counter.
          input [13:0] branch_target - this is either the branch target
for a branch instruction in the adjacent decode pipeline stage or the
output of the call stack memory.
          output [13:0] prog_mem_addr_fetch - This is the address
presented to the fetch pipline stages port of the program memory ROM. The
outputs of that rom will be latched into the IF/ID pipeline register on the
next positive edge of the core clock. Memory has a full cycle of latency
before data is valid. Thus the memory will be run at at least a 10% higher
frequency clock with respect to the core to ensure that the output of the
memory is valid in time to ensure a single cycle for instruction fetch to
ensure that the pipeline remains as full as possible. This latency has been
tested on hardware using equal clocks which would work for pipelining, but
fetch would have to be stretched into a two stage cycle thus increasing the
branch penalty a cycle. I tested memory @110% of the core frequency and
data seemed to be valid at the next positive edge although the address
seemed to be held by the memory itself a little longer than would be
desired so more testing must be done to ensure that that would be stable. I
have also test the memory at double the frequency of the core and had no
issues.
*/

module program_counter(
      input clock,
      input nreset,
      input stall,
      input [13:0] load_value,
      output [13:0] next_prog_cntr,
      output [13:0] prog_mem_addr_fetch
);

      reg [13:0] value = 0;
      wire [13:0] next_value;

      always @ (posedge clock)
      begin
            if(nreset == 1'b0)
            begin
                  value <= 0;
            end
```

```
            else if(stall == 1'b0)
            begin
                  value <= load_value;
            end
            else
            begin
                  value <= value;
            end
      end

    assign next_value = value + 1;
    assign next_prog_cntr = next_value;
      assign prog_mem_addr_fetch = value;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("program_counter.vcd");
  $dumpvars (0, program_counter);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_program_counter(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.stall.value = 0

    #Test Parallel Load
    dut.nreset.value = 1
    dut.branch_target.value = 0x20
```

```
    dut.take_branch_target.value = 1
    await FallingEdge(dut.clock)
    assert dut.prog_mem_addr_fetch.value == 0x20, f"Parallel Load Failed"

    #Test That data on load bus is not latched without the
take_branch_target signal high
    #This test also verifies the supremacy of take_branch_target over
stall.
    dut.branch_target.value = 0x10
    dut.take_branch_target.value = 0
    await FallingEdge(dut.clock)
    assert dut.prog_mem_addr_fetch.value == 0x21, f"Parallel Load occured
when signal was low. or stall did not function properly addr =
{dut.prog_mem_addr_fetch.value}"

    #Test that an increment does not occur by pulling stall high.
    dut.stall.value = 1
    await FallingEdge(dut.clock)
    assert dut.prog_mem_addr_fetch.value == 0x21, f"Program Counter
Increment occured when stall was asserted"

    #Ensure control signal supremacy nreset > take_branch_target > stall
    #Care Must be taken, that on a cycle in which the program counter must
stall, the take_branch_target signal, must be low.
    dut.nreset.value = 0
    dut.stall.value = 1
    dut.take_branch_target.value = 1
    await FallingEdge(dut.clock)
    assert dut.prog_mem_addr_fetch == 0, f"Reset did not function, or it
did not have supremacy over stall and take_branch_target"
```

6.2.1.2 Program Counter Input Selection Mux

Code:

```
/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/
```

```
module prog_cntr_input_sel_mux(
      //input clock,
      input [3:0] sel_signals,
      input [13:0] next_prog_cntr,
      input [13:0] branch_target_addr,
      input [13:0] int_branch_addr,
      input [13:0] ret_addr,
      output reg [13:0] prog_cntr_load_val
);

      always @ (*)
      begin
            if(sel_signals == 4'b0001)
            begin
                  prog_cntr_load_val <= branch_target_addr;
            end
            else if(sel_signals == 4'b0010)
            begin
                  prog_cntr_load_val <= next_prog_cntr;
            end
            else if(sel_signals == 4'b0100)
            begin
                  prog_cntr_load_val <= int_branch_addr;
            end
            else if(sel_signals == 4'b1000)
            begin
                  prog_cntr_load_val <= ret_addr;
            end
            else
            begin
                  prog_cntr_load_val <= next_prog_cntr;
            end
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("prog_cntr_input_sel_mux.vcd");
  $dumpvars (0, prog_cntr_input_sel_mux);
  #1;
end
`endif
```

```
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_CHANGE(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.sel_signals.value = 1

    dut.next_prog_cntr.value = 32
    dut.branch_target_addr.value = 64
    dut.int_branch_addr.value = 128
    dut.ret_addr.value = 256

    await FallingEdge(dut.clock)
    assert dut.prog_cntr_load_val.value == 64, f"failed"

    dut.sel_signals.value = 2
    await FallingEdge(dut.clock)
    assert dut.prog_cntr_load_val.value == 32, f"failed"

    dut.sel_signals.value = 4
    await FallingEdge(dut.clock)
    assert dut.prog_cntr_load_val.value == 128, f"failed"

    dut.sel_signals.value = 8
    await FallingEdge(dut.clock)
    assert dut.prog_cntr_load_val.value == 256, f"failed"
```

6.2.2 IF/ID Register

Code:

```
/*
Module - Instruction Fetch/Instruction Decode Pipeline Register.
```

```
Author - Zach Walden
Last Changed - 2/12/22, 3/27/22
Description - This register simply stores the instruction word coming out
of the fetch port on the positive edge of every clock.
Parameters -      clock - System Clock.
            nreset - Active low system reset signal.
            [31:0] instruction_in - Instruction word input. This bus comes
directly from the program memory read port in the Instruction Fetch
pipeline stage.
            [31:0] instruction_out - Instruction word output going to the
decode pipeline stage.
*/

module if_id(
      input clock,
      input nreset,
      input stall,
      input [31:0] instruction_in,
      output reg [31:0] instruction_out = 0,
      input [13:0] return_addr_in,
      output reg [13:0] return_addr_out = 0
);

      always @ (posedge clock)
      begin
            if(nreset == 1'b0)
            begin
                  instruction_out <= 32'h00000000;

                  return_addr_out <= 0;
            end
            else if(stall == 1'b0)
            begin
                  instruction_out <= instruction_in;

                  return_addr_out <= return_addr_in;
            end
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
```

```
  $dumpfile ("if_id.vcd");
  $dumpvars (0, if_id);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_if_id(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.nreset.value = 1
    dut.stall.value = 0
    await FallingEdge(dut.clock)

    #Test clock in nreset = 1, stall = 0
    dut.instruction_in.value = 0xFFFFFFFF
    await FallingEdge(dut.clock)
    assert dut.instruction_out.value == 0xFFFFFFFF, f"Instruction clock in
failed"

    #test the take branch control signal passthrough from the Instruction
Decode pipeline stage.
    dut.take_branch_addr.value = 1
    await FallingEdge(dut.clock)
    assert dut.take_branch_addr_out == 1, f"Take brach passthrough failed"

    #Test stall, nreset = 1, stall = 1
    dut.stall.value = 1
    dut.instruction_in.value = 0xFFFFFFFF
    await FallingEdge(dut.clock)
    assert dut.instruction_out.value == 0, f"Stall Failed"

    #Test reset, nreset = 0 other signals do not matter.
    dut.nreset.value = 0
```

```
    dut.instruction_in.value = 0xFFFFFFFF
    await FallingEdge(dut.clock)
    assert dut.instruction_out.value == 0, f"Reset failed"
```

6.2.3 Decode Stage

Code:

```
/*
Module - Instruction Decode
Author - Zach Walden
Last Changed - 3/28/22
Description - Instruction Decode Pipeline Stage
Parameters -
*/

module decode(
      input clock,
      input nreset,
      //BEGIN interface definition with General Purpose Register File.
      output [1:0] reg_file_ren,
      input [7:0] reg_file_data_top,
      input [7:0] reg_file_data_bot,
      //BEGIN Interface with IF/ID pipeline register
      output take_branch_target,
      input [31:0] instruction_word,
      //BEGIN Interface with ID/EX pipeline register.
      input [31:0] id_ex_instruction,
      input [31:0] ex_mem_instruction,
      input [2:0] alu_flags,
      output [4:0] alu_top_sel,
      output [4:0] alu_bot_sel,
      output [1:0] ex_mem_data_input_sel,
      output mem_wen,
      output main_memory_en,
      output fb_en,
      output call_stack_en,
      output prog_mem_en,
      output [6:0] mem_ptr_ctl,
      output [1:0] reg_file_wen,
      output [1:0] sfr_file_wren,
      output [7:0] id_ex_data_top,
```

```
      output [7:0] id_ex_data_bot,
      //BEGIN interface to hazard unit.
      output stall_fetch,
      output stall_decode,
      output halt,
      output illegal_opcode_exception,
      output return_in_pipeline                              //This signal is
neccessary in order to communicate to the hazard unit that a return
instruction has entered the pipeline. Requiring fetch to stall until the
return address can be popped off of the call stack and loaded into the
program counter.
);

      wire idex_data_sel;
      //instantiate instruction decode unit.
      decode_logic inst_decoder(
            .instruction(instruction_word),
            .reg_file_ren(reg_file_ren),
            .id_ex_data_input_sel(idex_data_sel),
            .ex_mem_data_input_sel(ex_mem_data_input_sel),
            .main_memory_enable(main_memory_en),
            .frame_buffer_enable(fb_en),
            .call_stack_enable(call_stack_en),
            .prog_mem_enable(prog_mem_en),
            .mem_ptr_ctl(mem_ptr_ctl),
            .sfr_wren(sfr_file_wren),
            .mem_wen(mem_wen),
            .reg_file_wen(reg_file_wen),
            .return_in_pipeline(return_in_pipeline),
            .stall_fetch(stall_fetch),
            .illegal_opcode_exception(illegal_opcode_exception),
            .halt(halt)
      );

      //instantiate branch_resolution_logic.
      branch_resolution_logic branch_resolver(
            //.clock(clock),
            .operation(instruction_word),
            .flags(alu_flags),
            .take_branch_target(take_branch_target)
      );

      //instantiate alu forwarding unit.
```

```
	alu_forwarding_logic forward_alu(
		//.clock(),
		.instruction(instruction_word),
		.ex_mem_instruction(ex_mem_instruction),
		.id_ex_instruction(id_ex_instruction),
		.alu_top_sel(alu_top_sel),
		.alu_bot_sel(alu_bot_sel),
		.stall_decode(stall_decode)
	);

	//instantiate ex_mem_data_input_mux
	id_ex_data_input_mux id_ex_mux(
		.sel_signal(idex_data_sel),
		.immeadiate_data(instruction_word[31:24]),
		.reg_file_top(reg_file_data_top),
		.reg_file_bot(reg_file_data_bot),
		.id_ex_data_input_top(id_ex_data_top),
		.id_ex_data_input_bot(id_ex_data_bot)
	);

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("decode.vcd");
  $dumpvars (0, decode);
  #1;
end
`endif
*/
endmodule
```

6.2.3.1 Instruction Decode Logic

Code:

```
/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/
```

```
module decode_logic(
      input [31:0] instruction,

      output reg [1:0] reg_file_ren = 0,
      output reg id_ex_data_input_sel = 0,

      output reg [1:0] ex_mem_data_input_sel = 0,

      output reg main_memory_enable = 0,
      output reg frame_buffer_enable = 0,
      output reg call_stack_enable = 0,
      output reg prog_mem_enable = 0,
      output reg [6:0] mem_ptr_ctl = 0,
      output reg [1:0] sfr_wren = 0,
      output reg mem_wen = 0,

      output reg [1:0] reg_file_wen = 0,

      output reg return_in_pipeline = 0,              //To the Hazard Unit
      output reg stall_fetch = 0,               //To the Hazard Unit
      output reg illegal_opcode_exception = 0,        //To the Interrupt
Contorller.
      output reg halt = 0                       //To the Hazard Unit
);

      always @ (*)
      begin
            case(instruction[7:0])
                  //NOP DONE
                  8'h00 :
                  begin
                        //All Zeros
                        reg_file_ren <= 2'b00;
                        id_ex_data_input_sel <= 1'b0;
                        ex_mem_data_input_sel <= 2'b00;
                        main_memory_enable <= 1'b0;
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;
```

```
                        reg_file_wen <= 2'b00;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Add Immeadiate, Increment, Decrement, Sub Immeadiate,
Complement, Invert, Compare Immeadiate   DONE
                  8'hBC :
                  begin
                        reg_file_ren <= 2'b01;              //Load
Destination Register Operand. This will be in ID/EX Data Top

                        id_ex_data_input_sel <= 1'b1;             //Select
Imemadiate
                        ex_mem_data_input_sel <= 2'b10;     //Select ALU
Result Bottom, the bottom bit does not matter in this case
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= 0;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Add, Subtract, Compare   DONE
                  8'h80 :
                  begin
                        reg_file_ren <= 2'b11;

                        id_ex_data_input_sel <= 1'b0;             //Select
Register File Output
```

```
                        ex_mem_data_input_sel <= 2'b10;     //Select ALU
Result Bottom, the bottom bit does not matter in this case
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= 0;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                end
                //Multiply DONE
                8'h8E :
                begin
                        reg_file_ren <= 2'b11;

                        id_ex_data_input_sel <= 1'b0;             //Select
Register File Output
                        ex_mem_data_input_sel <= 2'b11;     //Select ALU
Result Bottom & Top
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= instruction[21];

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
```

```
                        halt <= 1'b0;

                end
                //Mulitply Immeadiate   DONE
                8'h9E :
                begin
                        reg_file_ren <= 2'b01;

                        id_ex_data_input_sel <= 1'b1;             //Select
Imemadiate
                        ex_mem_data_input_sel <= 2'b11;     //Select ALU
Result Bottom & Top
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= instruction[21];

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                end
                //And, Or   DONE
                8'h97 :
                begin
                        reg_file_ren <= 2'b11;

                        id_ex_data_input_sel <= 1'b0;             //Select
Register File Output
                        ex_mem_data_input_sel <= 2'b10;     //Select ALU
Result Bottom, the bottom bit does not matter in this case
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
```

```
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= 0;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //And Immeadiate, Or Immeadiate   DONE
                  8'h9B :
                  begin
                        //Read destination operand from the register file,
load it into
                        reg_file_ren <= 2'b01;

                        id_ex_data_input_sel <= 1'b1;             //Select
Imemadiate
                        ex_mem_data_input_sel <= 2'b10;     //Select ALU
Result Bottom, the bottom bit does not matter in this case
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= 0;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Shift Right, Shift Left   DONE
                  8'hA5 :
                  begin
```

```
                        reg_file_ren <= 2'b01;

                        id_ex_data_input_sel <= 1'b0;             //Select
Register File Output
                        ex_mem_data_input_sel <= 2'b10;     //Select ALU
Result Bottom, the bottom bit does not matter in this case
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= 0;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Load, Load Framebuffer, Pop   DONE
                  8'hFB :
                  begin
                        reg_file_ren <= 2'b00;

                        id_ex_data_input_sel <= 1'b0;
                        ex_mem_data_input_sel <= 2'b00;
                        main_memory_enable <= instruction[20];
//Select Between The correct memory
                        frame_buffer_enable <= ~instruction[20];
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;

                        case(instruction[19:18])
                              //Stack Pointer, i.e. this instruction is a
Pop
                              2'b00 :
                              begin
                                    mem_ptr_ctl <= 7'b0000010;
            //Stack Pointer Increment
```

```
                        end
                        //X Pointer
                        2'b01 :
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b0010000;
        //X Pointer Post Increment
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                        //Y Pointer
                        2'b10 :
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b0100000;
        //Y Pointer Post Increment
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                        //Z Pointer
                        2'b11 :
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b1000000;
        //Z Pointer Post Increment
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                endcase

                sfr_wren <= 2'b00;
```

```
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                  //Write Load Result Bottom is the Write Result Bit is set
in the instruciton word.
                        reg_file_wen[1] <= (~instruction[20] &
instruction[21]);             //Write Load Result Top if this is a LDFB and
Wrtie Result is set.

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Store, Store Framebuffer, Push   DONE
                  8'hC6 :
                  begin
                        reg_file_ren[0] <= ~instruction[20];
                  //Read Register Address Bottom if a Load Framebuffer
                        reg_file_ren[1] <= 1'b1;
            //Read Register Address Top Always

                        id_ex_data_input_sel <= 1'b0;
                        ex_mem_data_input_sel <= 2'b00;
                        main_memory_enable <= instruction[20];
//Select Between The correct memory
                        frame_buffer_enable <= ~instruction[20];
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;

                        case(instruction[19:18])
                              //Stack Pointer, i.e. this instruction is a
Push
                              2'b00 :
                              begin
                                    mem_ptr_ctl <= 7'b0000001;
            //Stack Pointer Decrement
                              end
                              //X Pointer
                              2'b01 :
                              begin
                                    if(instruction[22] == 1'b1)
                                    begin
```

```
                                    mem_ptr_ctl <= 7'b0010000;
            //X Pointer Post Increment
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                        //Y Pointer
                        2'b10 :
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b0100000;
            //Y Pointer Post Increment
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                        //Z Pointer
                        2'b11 :
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b1000000;
            //Z Pointer Post Increment
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                  endcase

                  sfr_wren <= 2'b00;
                  mem_wen <= 1'b1;

                  reg_file_wen <= 2'b00;

                  return_in_pipeline <= 1'b0;
                  stall_fetch <= 1'b0;
```

```
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Load Immeadiate   DONE
                  8'hF8 :
                  begin
                        //This moves the immeadiate data into the
destination register.
                        reg_file_ren <= 2'b00;

                        id_ex_data_input_sel <= 1'b1;             //Select
Imemadiate
                        ex_mem_data_input_sel <= 2'b00;     //Select the two
data words in ID/EX to be placed in EX/MEM
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= 1'b0;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Move Register, In, Out   DONE
                  8'h9C :
                  begin
                        //OUT SFR Write Address is the "Top" address, IN
SFR Read Addres is "Bottom" Address
                        //This block
                        if(instruction[19:18] == 2'b00)
                        begin
                              //Move Register.
                              reg_file_ren <= 2'b10;        //Read Top
address from the register file.
                              sfr_wren <= 2'b00;
```

```
                              reg_file_wen <= 2'b01;              //Write
the read value to the bottom address. Data values are flipped coming out of
the register file, alu resultss are then reflipped. So, only storage data
need to be flipped.
                        end
                        else if(instruction[19:18] == 2'b10)
                        begin
                              //IN.
                              reg_file_ren <= 2'b00;        //Read Top
address from the register file.
                              sfr_wren <= 2'b10;
                              reg_file_wen <= 2'b01;              //Write
the read value to the bottom address. Data values are flipped coming out of
the register file, alu resultss are then reflipped. So, only storage data
need to be flipped.
                        end
                        else if(instruction[19:18] == 2'b01)
                        begin
                              //OUT.
                              reg_file_ren <= 2'b10;        //Read Top
address from the register file.
                              sfr_wren <= 2'b01;
                              reg_file_wen <= 2'b00;              //Write
the read value to the bottom address. Data values are flipped coming out of
the register file, alu resultss are then reflipped. So, only storage data
need to be flipped.
                        end
                        else
                        begin
                              //Should Never Happen, but if so do
something.
                              reg_file_ren <= 2'b00;        //Read Top
address from the register file.
                              sfr_wren <= 2'b00;
                              reg_file_wen <= 2'b00;              //Write
the read value to the bottom address. Data values are flipped coming out of
the register file, alu resultss are then reflipped. So, only storage data
need to be flipped.
                        end

                        id_ex_data_input_sel <= 1'b0;             //Select
the Reg file outputs.
                        ex_mem_data_input_sel <= 2'b00;     //Select the two
```

```
data words in ID/EX to be placed in EX/MEM
                        main_memory_enable <= 1'b0;         //No memory
Accesses, Load Store Architecture.
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        mem_wen <= 1'b0;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Control Flow Instructions
                  8'h38 :
                  begin
                        //All Zeros
                        reg_file_ren <= 2'b00;

                        id_ex_data_input_sel <= 1'b0;
                        ex_mem_data_input_sel <= 2'b00;
                        main_memory_enable <= 1'b0;
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;
                        reg_file_wen <= 2'b00;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Call   DONE
                  8'h42 :
                  begin
                        reg_file_ren <= 2'b00;

                        id_ex_data_input_sel <= 1'b0;
                        ex_mem_data_input_sel <= 2'b00;
```

```
                        main_memory_enable <= 1'b0;
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b1;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0001000;                //Call
Stack Pointer Increment
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b1;
                        reg_file_wen <= 2'b00;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b1;                      //This may
be unecessary, depends on logic delay and the ratio between memory clock
and core clock.
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Return, Return From Interrupt   DONE
                  8'h43 :
                  begin
                        //Invert instruction[20] signifies whether the
instruction is ret or reti
                        reg_file_ren <= 2'b00;

                        id_ex_data_input_sel <= 1'b0;
                        ex_mem_data_input_sel <= 2'b00;
                        main_memory_enable <= 1'b0;
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b1;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000100;                //Call
Stack Pointer Decrement
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;
                        reg_file_wen <= 2'b00;

                        return_in_pipeline <= 1'b1;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Load From Program Memory   DONE
                  8'hF9 :
```

```
                  begin
                        reg_file_ren <= 2'b00;

                        id_ex_data_input_sel <= 1'b0;
//Select The Register File Outputs. Does, not matter.
                        ex_mem_data_input_sel <= 2'b00;           //Select
ID/EX Data, does not matter.
                        main_memory_enable <= 1'b0;
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b1;

                        //Check for a post increment
                        if(instruction[19:18] == 2'b01)
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b0010000;          //X
Pointer Post Increment
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                        else if(instruction[19:18] == 2'b10)
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b0100000;          //Y
Pointer Post Increment
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                        else if(instruction[19:18] == 2'b11)
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b1000000;          //Z
Pointer Post Increment
```

```
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end
                        else
                        begin
                              if(instruction[22] == 1'b1)
                              begin
                                    mem_ptr_ctl <= 7'b0000000;          //Do
not modify the value of the stack pointer.
                              end
                              else
                              begin
                                    mem_ptr_ctl <= 7'b0000000;
                              end
                        end

                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;

                        reg_file_wen[0] <= instruction[21];
                        reg_file_wen[1] <= 0;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b0;
                  end
                  //Halt   DONE
                  8'h1F :
                  begin
                        //All Zeros
                        reg_file_ren <= 2'b00;

                        id_ex_data_input_sel <= 1'b0;
                        ex_mem_data_input_sel <= 2'b00;
                        main_memory_enable <= 1'b0;
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
```

```
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;
                        reg_file_wen <= 2'b00;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b0;
                        halt <= 1'b1;
                end
                //Default Case   DONE
                default
                begin
                        //Illegal Opcode Exception. This is very useful for
security. All other control signals are NOP'd
                        //All Zeros
                        reg_file_ren <= 2'b00;

                        id_ex_data_input_sel <= 1'b0;
                        ex_mem_data_input_sel <= 2'b00;
                        main_memory_enable <= 1'b0;
                        frame_buffer_enable <= 1'b0;
                        call_stack_enable <= 1'b0;
                        prog_mem_enable <= 1'b0;
                        mem_ptr_ctl <= 7'b0000000;
                        sfr_wren <= 2'b00;
                        mem_wen <= 1'b0;
                        reg_file_wen <= 2'b00;

                        return_in_pipeline <= 1'b0;
                        stall_fetch <= 1'b0;
                        illegal_opcode_exception <= 1'b1;
                        halt <= 1'b0;
                end
            endcase
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("decode_logic.vcd");
  $dumpvars (0, decode_logic);
  #1;
```

```
end
`endif
*/
endmodule
```

6.2.3.2 Branch Resolution Logic

Code:

```
/*
Module - Branch Resolution Logic
Author - Zach Walden
Last Changed - 4/1/22
Description - This Logic Resolves Whether a Branch Target Address should be
taken or not.
Parameters -
*/

module branch_resolution_logic(
	//input clock,
	input [31:0] operation,
	input [2:0] flags,
	output reg take_branch_target = 0
);

	always @ (*)
	begin
		if(operation[7:0] == 8'b00111000)
		begin
			//Carry Branches flags[2]
			if(operation[9:8] == 2'b01)
			begin
				if(flags[2] == 1'b1 && operation[10] == 1'b1)
				begin
					take_branch_target <= 1;
				end
				else if(flags[2] == 1'b0 && operation[10] == 1'b0)
				begin
					take_branch_target <= 1;
				end
				else
				begin
					take_branch_target <= 0;
```

```
                end
            end
            //Zero Branches flags[0]
            else if(operation[9:8] == 2'b10)
            begin
                if(flags[0] == 1'b1 && operation[10] == 1'b1)
                begin
                    take_branch_target <= 1;
                end
                else if(flags[0] == 1'b0 && operation[10] == 1'b0)
                begin
                    take_branch_target <= 1;
                end
                else
                begin
                    take_branch_target <= 0;
                end
            end
            //Negative Branches flags[1]
            else if(operation[9:8] == 2'b11)
            begin
                if(flags[1] == 1'b1 && operation[10] == 1'b1)
                begin
                    take_branch_target <= 1;
                end
                else if(flags[1] == 1'b0 && operation[10] == 1'b0)
                begin
                    take_branch_target <= 1;
                end
                else
                begin
                    take_branch_target <= 0;
                end
            end
            else
            begin
                take_branch_target <= 1;
            end
        end
        //If a Call Instruction
        else if(operation[7:0] == 8'h42)
        begin
            take_branch_target <= 1;
```

```
            end
            else
            begin
                  take_branch_target <= 0;
            end
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("branch_resolution_logic.vcd");
  $dumpvars (0, branch_resolution_logic);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_branch_resolution_logic(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    #Run Through Each Control Flow Opcode, Jump and Branches.

    #Jump Opcode = 0b00111 1 00
    dut.operation.value = 0x3C
    #Loop through each value for the flags registor 0-8 decimal and ensure
that take_branch_target is high.
    for i in range(8):
        dut.flags.value = i
        await FallingEdge(dut.clock)
        assert dut.take_branch_target.value == 1, f"Jump failed."

    #Branch If Carry Set Opcode = 0b00111 1 01
```

```
    dut.operation.value = 0x3D
    dut.flags.value = 0
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 0, f"BRCS Taken when carry
cleared"
    dut.flags.value = 4
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 1, f"BRCS Not Taken when carry
is set"

    #Branch If Carry Clear Opcode = 0b00111 0 01
    dut.operation.value = 0x39
    dut.flags.value = 4
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 0, f"BRCS Taken when carry set"
    dut.flags.value = 0
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 1, f"BRCS Not Taken when carry
is cleared"

    #Branch If Equal Opcode = 0b00111 1 10
    dut.operation.value = 0x3E
    dut.flags.value = 0
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 0, f"BREQ Taken when Zero flag
is cleared"
    dut.flags.value = 1
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 1, f"BREQ Not Taken when Zero
flag is set"

    #Branch If Not Equal Opcode = 0b00111 0 10
    dut.operation.value = 0x3A
    dut.flags.value = 1
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 0, f"BRNE Taken when Zero flag
is set"
    dut.flags.value = 0
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 1, f"BRNE Not Taken when Zero
flag is cleared"

    #Branch If Negative Opcode = 0b00111 1 11
```

```
    dut.operation.value = 0x3F
    dut.flags.value = 0
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 0, f"BRNG Taken when Negative
flag is cleared"
    dut.flags.value = 3
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 1, f"BRNG Not Taken when
Negative flag is set"

    #Branch If Positive Opcode = 0b00111 0 11
    dut.operation.value = 0x3B
    dut.flags.value = 3
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 0, f"BRPS Taken when Negative
flag is set"
    dut.flags.value = 0
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 1, f"BRPS Not Taken when
Negative flag is cleared"

    #Test if a branch is taken when a non branch operation is set
    dut.operation.value = 0
    dut.flags.value = 0
    await FallingEdge(dut.clock)
    assert dut.take_branch_target.value == 0, f"Branch Target Taken on a
non branch operation."
```

6.2.3.3 ALU Forwarding Logic

Code:

```
/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/

module alu_forwarding_logic(
            //input clock,
            input [31:0] instruction,
```

```
            input [31:0] ex_mem_instruction,
            input [31:0] id_ex_instruction,
            output reg [4:0] alu_top_sel = 0,
            output reg [4:0] alu_bot_sel = 0,
            output reg stall_decode = 0
);

      always @ (*)
      begin
            //Default Values
            alu_top_sel <= 5'b00001;
            alu_bot_sel <= 5'b00001;
            stall_decode <= 1'b0;
            case(instruction[7:0])
                  //Add Immeadiate, Increment, Decrement, Sub Immeadiate,
Complement, Invert, Compare Immeadiate, Or Immeadiate, And Immeadiate,
Mulitply Immeadiate, Shift Right, Shift Left   CASES DONE
                  8'hBC, 8'h9E, 8'h9B, 8'hA5 :
                  begin
                        //Check for dependent load that requires a stall.
                        //LD, LDFB
                        if(id_ex_instruction[7:0] == 8'hFB)
                        begin
                              //Check if a LDFB or not.
                              if(id_ex_instruction[20] == 1'b1)
                              begin
                                    //Normal Load
                                    if(instruction[12:8] ==
id_ex_instruction[12:8])
                                    begin
                                          //forawrd the load result bot to
the alu top, and STALL
                                          alu_top_sel <= 5'b10000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b1;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
```

```
                              end
                              else
                              begin
                                    //Load Frame Buffer
                                    if(instruction[12:8] ==
id_ex_instruction[12:8])
                                    begin
                                          //Forward load result bottom to
the alu top, STALL
                                          alu_top_sel <= 5'b10000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b1;
                                    end
                                    else if(instruction[12:8] ==
id_ex_instruction[17:13])
                                    begin
                                          //forward load result top to the
alu top, STALL
                                          alu_top_sel <= 5'b01000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b1;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                              end
                        end
                        //Load Immeadiate, This could never cause a stall
                        else if(id_ex_instruction[7:0] == 8'hF8)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward ex/mem data bottom on to alu
top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
```

```
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Load Program Memory
                        else if(id_ex_instruction[7:0] == 8'hF9)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //forward load result bottom to the alu
top, ,STALL
                                    alu_top_sel <= 5'b10000;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b1;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //In
                        else if( id_ex_instruction[7:0] == 8'h9C &&
id_ex_instruction[19:18] == 2'b10)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //forward load result bottom to the alu
top, STALL
                                    alu_top_sel <= 5'b10000;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b1;
                              end
                              else
                              begin
```

```
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //MOVR
                        else if( id_ex_instruction[7:0] == 8'h9C &&
id_ex_instruction[19:18] == 2'b00)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //forward ex/mem data bottom to the alu
top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Check For a potential dependent arithmetic
instruction.
                        //ADDI, SUBI, CPI, COM, INV
                        else if(id_ex_instruction[7:0] == 8'hBC &&
id_ex_instruction[21] == 1'b1)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
```

```
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //ADD, SUB, CP
                        else if(id_ex_instruction[7:0] == 8'h80 &&
id_ex_instruction[21] == 1'b1)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //MUL, MULI
                        else if(id_ex_instruction[7:0] == 8'h8E ||
id_ex_instruction[7:0] == 8'h9E)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu
top.
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if(instruction[12:8] ==
id_ex_instruction[17:13])
                              begin
                                    //Forward EX/MEM data top to alu top
```

```
                                    alu_top_sel <= 5'b00010;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //AND, OR
                        else if(id_ex_instruction[7:0] == 8'h97)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //ANDI, ORI
                        else if(id_ex_instruction[7:0] == 8'h9B)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
```

```
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //SHR, SHL
                        else if(id_ex_instruction[7:0] == 8'hA5)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Check for a potential dependent Load
                        //LD, LDFB
                        else if(ex_mem_instruction[7:0] == 8'hFB)
                        begin
                              //Check if a LDFB or not.
                              if(ex_mem_instruction[20] == 1'b1)
                              begin
                                    //Normal Load
                                    if(instruction[12:8] ==
ex_mem_instruction[12:8])
                                    begin
                                          //forawrd the load result bot to
the alu top
                                          alu_top_sel <= 5'b10000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
```

```
                                    else
                                    begin
                                          //No forward necessary
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                              end
                              else
                              begin
                                    //Load Frame Buffer
                                    if(instruction[12:8] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward load result bottom to
the alu top
                                          alu_top_sel <= 5'b10000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                                    else if(instruction[12:8] ==
ex_mem_instruction[17:13])
                                    begin
                                          //forward load result top to the
alu top
                                          alu_top_sel <= 5'b01000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                              end
                        end
                        //Load Immeadiate, This could never cause a stall
                        else if(ex_mem_instruction[7:0] == 8'hF8)
                        begin
                              if(instruction[12:8] ==
ex_mem_instruction[12:8])
```

```
                              begin
                                    //Forward mem/wb data bottom on to alu
top
                                    alu_top_sel <= 5'b10000;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Load Program Memory
                        else if(ex_mem_instruction[7:0] == 8'hF9)
                        begin
                              if(instruction[12:8] ==
ex_mem_instruction[12:8])
                              begin
                                    //forward load result bottom to the alu
top
                                    alu_top_sel <= 5'b10000;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //In, Move Register
                        else if( ex_mem_instruction[7:0] == 8'h9C &&
(ex_mem_instruction[19:18] == 2'b10 || ex_mem_instruction[19:18] == 2'b00))
                        begin
                              if(instruction[12:8] ==
ex_mem_instruction[12:8])
                              begin
                                    //forward load result bottom to the alu
```

```
top
                                    alu_top_sel <= 5'b10000;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //No Hazards
                        else
                        begin
                              alu_top_sel <= 5'b00001;
                              alu_bot_sel <= 5'b00001;
                              stall_decode <= 1'b0;
                        end

                  end
                  //Add, Subtract, Compare, MUL, AND, OR,
                  8'h80, 8'h8E, 8'h97 :
                  begin

                        //Check for dependent load that requires a stall.
                        //LD, LDFB
                        if(id_ex_instruction[7:0] == 8'hFB)
                        begin
                              //Check if a LDFB or not.
                              if(id_ex_instruction[20] == 1'b1)
                              begin
                                    //Normal Load
                                    if(instruction[12:8] ==
id_ex_instruction[12:8])
                                    begin
                                          //forawrd the load result bot to
the alu top. STALL
                                          alu_top_sel <= 5'b10000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b1;
                                    end
```

```
                                    else if(instruction[17:13] ==
id_ex_instruction[12:8])
                                    begin
                                          //Forward the load result bottom
to the alu bot. STALL
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b10000;
                                          stall_decode <= 1'b1;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                              end
                              else
                              begin
                                    //Load Frame Buffer
                                    if(instruction[12:8] ==
id_ex_instruction[12:8])
                                    begin
                                          if(instruction[17:13] ==
id_ex_instruction[17:13])
                                          begin
                                                //Forward both the top and
bottom load results to alu top and bottom STALL
                                                alu_top_sel <= 5'b10000;
                                                alu_bot_sel <= 5'b01000;
                                                stall_decode <= 1'b1;
                                          end
                                          else
                                          begin
                                                //Forward just the bottom
load result to the alu top. STALL
                                                alu_top_sel <= 5'b10000;
                                                alu_bot_sel <= 5'b00001;
                                                stall_decode <= 1'b1;
                                          end
                                    end
                                    else if(instruction[12:8] ==
id_ex_instruction[17:13])
```

```
                                    begin
                                          if(instruction[17:13] ==
id_ex_instruction[12:8])
                                          begin
                                                //Forward both the top and
bottom load results to alu top and bottom STALL
                                                alu_top_sel <= 5'b10000;
                                                alu_bot_sel <= 5'b01000;
                                                stall_decode <= 1'b1;
                                          end
                                          else
                                          begin
                                                //Forward just the the top
load result to the alu top STALL
                                                alu_top_sel <= 5'b01000;
                                                alu_bot_sel <= 5'b00001;
                                                stall_decode <= 1'b1;
                                          end
                                    end
                                    else if(instruction[17:13] ==
id_ex_instruction[12:8])
                                    begin
                                          //Forward Load result bot to the
alu bottom STALL
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b10000;
                                          stall_decode <= 1'b1;
                                    end
                                    else if(instruction[17:13] ==
id_ex_instruction[17:13])
                                    begin
                                          //forward load result top to the
alu bottom. STALL
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b01000;
                                          stall_decode <= 1'b1;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
```

```
                                    end
                              end
                        end
                        //Load Immeadiate, This could never cause a stall
                        else if(id_ex_instruction[7:0] == 8'hF8)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward ex/mem data bottom on to alu
top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if(instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward ex/mem data bottom to alu
bottom
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00100;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Load Program Memory
                        else if(id_ex_instruction[7:0] == 8'hF9)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //forward load result bot to the alu
top, STALL
                                    alu_top_sel <= 5'b10000;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b1;
```

```
                              end
                              else if(instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward load result top to alu bot,
STALL
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b01000;
                                    stall_decode <= 1'b1;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //In
                        else if( id_ex_instruction[7:0] == 8'h9C &&
id_ex_instruction[19:18] == 2'b10)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //forward mem/wb data bottom to the alu
top, STALL
                                    alu_top_sel <= 5'b10000;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b1;
                              end
                              else if (instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward mem/wb data bottom to alu
bottom, STALL
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b10000;
                                    stall_decode <= 1'b1;
                              end
                              else
                              begin
                                    //No forward necessary
```

```
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Move Register
                        else if( id_ex_instruction[7:0] == 8'h9C &&
id_ex_instruction[19:18] == 2'b00)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //forward ex/mem data bottom to the alu
top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if (instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward ex/mem data bottom to alu
bottom
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00100;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Check For a potential dependent arithmetic
instruction.
                        //ADDI, SUBI, CPI, COM, INV
                        else if(id_ex_instruction[7:0] == 8'hBC &&
id_ex_instruction[21] == 1'b1)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
```

```
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if(instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu
bottom.
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00100;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //ADD, SUB, CP
                        else if(id_ex_instruction[7:0] == 8'h80 &&
id_ex_instruction[21] == 1'b1)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if(instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu
bottom.
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00100;
                                    stall_decode <= 1'b0;
```

```
                              end
                              else
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //MUL, MULI
                        else if(id_ex_instruction[7:0] == 8'h8E ||
id_ex_instruction[7:0] == 8'h9E)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    if(instruction[17:13] ==
id_ex_instruction[17:13])
                                    begin
                                          //Forward both the top and bottom
ex/mem data values to alu top and bottom
                                          alu_top_sel <= 5'b00100;
                                          alu_bot_sel <= 5'b00010;
                                          stall_decode <= 1'b0;
                                    end
                                    else
                                    begin
                                          //Forward just the bottom ex/mem
data to the alu top.
                                          alu_top_sel <= 5'b00100;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                              end
                              else if(instruction[12:8] ==
id_ex_instruction[17:13])
                              begin
                                    if(instruction[17:13] ==
id_ex_instruction[12:8])
                                    begin
                                          //Forward both the top and bottom
ex/mem data values to alu top and bottom
                                          alu_top_sel <= 5'b00010;
```

```
                                    alu_bot_sel <= 5'b00100;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //Forward just the the top ex/mem
data value to the alu top
                                    alu_top_sel <= 5'b00010;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        else if(instruction[17:13] ==
id_ex_instruction[12:8])
                        begin
                              //Forward ex/mem data bot to the alu
bottom
                              alu_top_sel <= 5'b00001;
                              alu_bot_sel <= 5'b00001;
                              stall_decode <= 1'b0;
                        end
                        else if(instruction[17:13] ==
id_ex_instruction[17:13])
                        begin
                              //forward ex/mem data top to the alu
bottom
                        end
                        else
                        begin
                              //No forward necessary
                              alu_top_sel <= 5'b00001;
                              alu_bot_sel <= 5'b00001;
                              stall_decode <= 1'b0;
                        end
                  end
                  //AND, OR
                  else if(id_ex_instruction[7:0] == 8'h97)
                  begin
                        if(instruction[12:8] ==
id_ex_instruction[12:8])
                        begin
                              //Forward EX/MEM data bottom to alu top
                              alu_top_sel <= 5'b00100;
```

```
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if(instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu
bottom.
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00100;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //ANDI, ORI
                        else if(id_ex_instruction[7:0] == 8'h9B)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if(instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu
bottom.
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00100;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forwarding necessary
```

```
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //SHR, SHL
                        else if(id_ex_instruction[7:0] == 8'hA5)
                        begin
                              if(instruction[12:8] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu top
                                    alu_top_sel <= 5'b00100;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if(instruction[17:13] ==
id_ex_instruction[12:8])
                              begin
                                    //Forward EX/MEM data bottom to alu
bottom.
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00100;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forwarding necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Check for a potential dependent Load
                        //LD, LDFB
                        else if(ex_mem_instruction[7:0] == 8'hFB)
                        begin
                              //Check if a LDFB or not.
                              if(ex_mem_instruction[20] == 1'b1)
                              begin
                                    //Normal Load
                                    if(instruction[12:8] ==
ex_mem_instruction[12:8])
```

```
                                    begin
                                          //forawrd the load result bot to
the alu top.
                                          alu_top_sel <= 5'b10000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                                    else if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward the load result bottom
to the alu bot.
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b10000;
                                          stall_decode <= 1'b0;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                              end
                              else
                              begin
                                    //Load Frame Buffer
                                    if(instruction[12:8] ==
ex_mem_instruction[12:8])
                                    begin
                                          if(instruction[17:13] ==
ex_mem_instruction[17:13])
                                          begin
                                                //Forward both the top and
bottom load results to alu top and bottom
                                                alu_top_sel <= 5'b10000;
                                                alu_bot_sel <= 5'b01000;
                                                stall_decode <= 1'b0;
                                          end
                                          else
                                          begin
                                                //Forward just the bottom
load result to the alu top.
```

```
                                                alu_top_sel <= 5'b10000;
                                                alu_bot_sel <= 5'b00001;
                                                stall_decode <= 1'b0;
                                          end
                                    end
                                    else if(instruction[12:8] ==
ex_mem_instruction[17:13])
                                    begin
                                          if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                          begin
                                                //Forward both the top and
bottom load results to alu top and bottom
                                                alu_top_sel <= 5'b10000;
                                                alu_bot_sel <= 5'b01000;
                                                stall_decode <= 1'b0;
                                          end
                                          else
                                          begin
                                                //Forward just the the top
load result to the alu top
                                                alu_top_sel <= 5'b01000;
                                                alu_bot_sel <= 5'b00001;
                                                stall_decode <= 1'b0;
                                          end
                                    end
                                    else if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward Load result bot to the
alu bottom
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b10000;
                                          stall_decode <= 1'b0;
                                    end
                                    else if(instruction[17:13] ==
ex_mem_instruction[17:13])
                                    begin
                                          //forward load result top to the
alu bottom.
                                          alu_top_sel <= 5'b01000;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
```

```
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          alu_top_sel <= 5'b00001;
                                          alu_bot_sel <= 5'b00001;
                                          stall_decode <= 1'b0;
                                    end
                              end
                        end
                        //Load Immeadiate, This could never cause a stall
                        else if(ex_mem_instruction[7:0] == 8'hF8)
                        begin
                              if(instruction[12:8] ==
ex_mem_instruction[12:8])
                              begin
                                    //Forward mem/wb data bottom on to alu
top
                                    alu_top_sel <= 5'b10000;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                              else if(instruction[17:13] ==
ex_mem_instruction[12:8])
                              begin
                                    //Forward mem/wb data bottom to alu
bottom
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b10000;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //Load Program Memory
                        else if(ex_mem_instruction[7:0] == 8'hF9)
                        begin
                              if(instruction[12:8] ==
```

```
ex_mem_instruction[12:8])
                                begin
                                      //forward load result bottom to the alu
top
                                      alu_top_sel <= 5'b10000;
                                      alu_bot_sel <= 5'b00001;
                                      stall_decode <= 1'b0;
                                end
                                else if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                begin
                                      //Forward ex/mem data bottom to alu
bottom
                                      alu_top_sel <= 5'b00001;
                                      alu_bot_sel <= 5'b00100;
                                      stall_decode <= 1'b0;
                                end
                                else
                                begin
                                      //No forward necessary
                                      alu_top_sel <= 5'b00001;
                                      alu_bot_sel <= 5'b00001;
                                      stall_decode <= 1'b0;
                                end
                          end
                          //In, Move Register
                          else if( ex_mem_instruction[7:0] == 8'h9C &&
(ex_mem_instruction[19:18] == 2'b10 || ex_mem_instruction[19:18] == 2'b00))
                          begin
                                if(instruction[12:8] ==
ex_mem_instruction[12:8])
                                begin
                                      //forward mem_wb data bottom to the alu
top
                                      alu_top_sel <= 5'b10000;
                                      alu_bot_sel <= 5'b00001;
                                      stall_decode <= 1'b0;
                                end
                                else if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                begin
                                      //Forward mem/wb data bottom to alu
bottom
```

```
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b10000;
                                    stall_decode <= 1'b0;
                              end
                              else
                              begin
                                    //No forward necessary
                                    alu_top_sel <= 5'b00001;
                                    alu_bot_sel <= 5'b00001;
                                    stall_decode <= 1'b0;
                              end
                        end
                        //No Hazards
                        else
                        begin
                              alu_top_sel <= 5'b00001;
                              alu_bot_sel <= 5'b00001;
                              stall_decode <= 1'b0;
                        end
                  end
                  //Default Case
                  default
                  begin
                        alu_top_sel <= 5'b00001;
                        alu_bot_sel <= 5'b00001;
                        stall_decode <= 1'b0;
                  end
            endcase
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("alu_forwarding_logic.vcd");
  $dumpvars (0, alu_forwarding_logic);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_alu_forwarding_logic(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.instruction.value = 0x00000000
    dut.ex_mem_instruction.value = 0x00000000
    dut.id_ex_instruction.value = 0x00000000

    await FallingEdge(dut.clock)
    assert dut.alu_top_sel.value == 0x00001, f"ALU Top select not default"
    assert dut.alu_bot_sel.value == 0x00001, f"ALU Bottpm Select dot
default"
    assert dut.stall_decode.value == 0, f"Stall High when all NOP"

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x006C20FB #Top Addr is 0x1, bottom addr is 0x0

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x00242080

    await FallingEdge(dut.clock)
    assert dut.stall_decode.value == 1, f"Stall High when all NOP"

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = 0x00000000

    await FallingEdge(dut.clock)
    assert dut.alu_top_sel.value ==0x10, f"ALU Top select not default"
    assert dut.alu_bot_sel.value == 0x8, f"ALU Bottpm Select dot default"
    assert dut.stall_decode.value == 0, f"Stall High when all NOP"
```

```
    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x006C20FB

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0F2C20BC

    await FallingEdge(dut.clock)
    assert dut.stall_decode.value == 1, f"Dependent Load did not stall"

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = 0x00000000

    await FallingEdge(dut.clock)
    assert dut.alu_top_sel.value ==0x10, f"ALU Top select not default"
    assert dut.alu_bot_sel.value == 0x1, f"ALU Bottpm Select dot default"
    assert dut.stall_decode.value == 0, f"Stall High when all NOP"

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0
```

```
await FallingEdge(dut.clock)

#Shift the Instructions
dut.ex_mem_instruction.value = dut.id_ex_instruction.value
dut.id_ex_instruction.value = dut.instruction.value
dut.instruction.value = 0x0

await FallingEdge(dut.clock)

#Shift the Instructions
dut.ex_mem_instruction.value = dut.id_ex_instruction.value
dut.id_ex_instruction.value = dut.instruction.value
dut.instruction.value = 0x0

await FallingEdge(dut.clock)

#Shift the Instructions
dut.ex_mem_instruction.value = dut.id_ex_instruction.value
dut.id_ex_instruction.value = dut.instruction.value
dut.instruction.value = 0x0

await FallingEdge(dut.clock)

#Shift the Instructions
dut.ex_mem_instruction.value = dut.id_ex_instruction.value
dut.id_ex_instruction.value = dut.instruction.value
dut.instruction.value = 0x0

await FallingEdge(dut.clock)

#Shift the Instructions
dut.ex_mem_instruction.value = dut.id_ex_instruction.value
dut.id_ex_instruction.value = dut.instruction.value
dut.instruction.value = 0x0

await FallingEdge(dut.clock)

#Shift the Instructions
dut.ex_mem_instruction.value = dut.id_ex_instruction.value
dut.id_ex_instruction.value = dut.instruction.value
dut.instruction.value = 0x0
```

```
    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0

    await FallingEdge(dut.clock)

    #Shift the Instructions
    dut.ex_mem_instruction.value = dut.id_ex_instruction.value
    dut.id_ex_instruction.value = dut.instruction.value
    dut.instruction.value = 0x0

    await FallingEdge(dut.clock)
```

6.2.3.4 ID/EX Data Input Selection Mux

Code:

```
/*
Module - ID/EX Data Input Multiplexer
Author - Zach Walden
Last Changed - 2/26/22
Description - This multiplexer switches the bottom data output of the
decode stage between the top data of the register file, and the immeadiate
data stored in the instruction word.
Parameters -
*/

module id_ex_data_input_mux(
            input sel_signal,
            input [7:0] immeadiate_data,
            input [7:0] reg_file_top,
            input [7:0] reg_file_bot,
            output [7:0] id_ex_data_input_top,
            output reg [7:0] id_ex_data_input_bot
);

      always @ (*)
      begin
            if(sel_signal == 1'b1)
            begin
                  id_ex_data_input_bot <= immeadiate_data;
            end
            else
            begin
                  id_ex_data_input_bot <= reg_file_top;
            end
      end

      assign id_ex_data_input_top = reg_file_bot;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("id_ex_data_input_mux.vcd");
```

```
  $dumpvars (0, id_ex_data_input_mux);
  #1;
end
`endif
*/
endmodule
```

6.2.4 ID/EX Register

Code:

```
/*
Module - Instruction Decode/Execution Pipeline Register
Author - Zach Walden
Last Changed - 2/12/22, 3/27/22
Description - This register holds the necessary data to ensure that the
correct results exit the ALU
*/

module id_ex(
      input clock,                        //System Clock
      input nreset,                       //System Reset Signal
      input stall,
      input [4:0] alu_top_select_in,      //I/O for the alu top input
multiplexor selection signals. Consumed in execute
      output reg [4:0] alu_top_select_out = 0,
      input [4:0] alu_bot_select_in,           //I/O for the alu bottom
input multiplexor selection signals. Consumed in execute
      output reg [4:0] alu_bot_select_out = 0,
      input [7:0] id_ex_top_in,           //I/O for the top register file
operand read.
      output reg [7:0] id_ex_top_out = 0,
      input [7:0] id_ex_bot_in,           //I/O for the bottom register file
operand read.
      output reg [7:0] id_ex_bot_out = 0,
      input [31:0] instruction_in,        //I/O for the instruction word.
      output reg [31:0] instruction_out = 0,
      input mem_wen_in,                                 //Consumed in memory
      output reg mem_wen_out = 0,
      input main_memory_enable_in,                      //Consumed in memory
      output reg main_memory_enable_out = 0,
      input frame_buffer_enable_in,                          //Consumed in
```

```
memory
      output reg frame_buffer_enable_out = 0,
      input call_stack_enable_in,                        //Consumed in memory
      output reg call_stack_enable_out = 0,
      input prog_mem_enable_in,                          //Consumed in memory
      output reg prog_mem_enable_out = 0,
      input [6:0] mem_ptr_ctl_in,                        //Consumed in memory
      output reg [6:0] mem_ptr_ctl_out = 0,
      input call_stk_addr_sel_in,                        //Consumed in memory
      output reg call_stk_addr_sel_out = 0,
      input stk_addr_sel_in,                             //Consumed in memory
      output reg stk_addr_sel_out = 0,
      input [1:0] ex_mem_data_input_sel_in,                  //Consumed in
execute
      output reg [1:0] ex_mem_data_input_sel_out = 0,
      input [1:0] reg_file_wen_in,                       //Consumed in memory
      output reg [1:0] reg_file_wen_out = 0,
      input [1:0] sfr_file_wren_in,                          //Consumed in
memory
      output reg [1:0] sfr_file_wren_out = 0,
      input [13:0] call_addr_in,                         ///Consumed in memory.
      output reg [13:0] call_addr_out = 0
);

    reg [6:0] mem_ptr_ctl_signals = 0;

      always @ (posedge clock)
      begin
            if(nreset == 1'b0 || stall == 1'b1)
            begin
                  ex_mem_data_input_sel_out <= 0;

                  alu_top_select_out <= 0;
                  alu_bot_select_out <= 0;

                  id_ex_top_out <= 0;
                  id_ex_bot_out <= 0;

                  instruction_out <= 0;

                  mem_wen_out <= 0;

                  main_memory_enable_out <= 0;
```

```
            frame_buffer_enable_out <= 0;

            call_stack_enable_out <= 0;

            prog_mem_enable_out <= 0;

            mem_ptr_ctl_out <= 0;
            call_stk_addr_sel_out <= 0;
            stk_addr_sel_out <= 0;

            reg_file_wen_out <= 0;

            sfr_file_wren_out <= 0;

            call_addr_out <= 0;
      end
      else
      begin
            ex_mem_data_input_sel_out <= ex_mem_data_input_sel_in;

            alu_top_select_out <= alu_top_select_in;
            alu_bot_select_out <= alu_bot_select_in;

            id_ex_top_out <= id_ex_top_in;
            id_ex_bot_out <= id_ex_bot_in;

            instruction_out <= instruction_in;

            mem_wen_out <= mem_wen_in;

            main_memory_enable_out <= main_memory_enable_in;

            frame_buffer_enable_out <= frame_buffer_enable_in;

            call_stack_enable_out <= call_stack_enable_in;

            prog_mem_enable_out <= prog_mem_enable_in;

            mem_ptr_ctl_signals <= mem_ptr_ctl_in;
            mem_ptr_ctl_out <= mem_ptr_ctl_signals;
            call_stk_addr_sel_out <= call_stk_addr_sel_in;
            stk_addr_sel_out <= stk_addr_sel_in;
```

```
                reg_file_wen_out <= reg_file_wen_in;

                sfr_file_wren_out <= sfr_file_wren_in;

                call_addr_out <= call_addr_in;
            end
        end


/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("id_ex.vcd");
  $dumpvars (0, id_ex);
  #1;
end
`endif
*/
endmodule
```

6.2.5 Execute Stage

Code:

```
/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/

module execute(
      input clock,
      input nreset,
      //BEGIN interface with ID/EX pipeline register.
      input [4:0] alu_top_sel,
      input [4:0] alu_bot_sel,
      input [7:0] data_in_top,
      input [7:0] data_in_bot,
```

```
      input [31:0] instruction,
      input [1:0] ex_mem_data_input_sel,
      output [2:0] flags_out,
      //BEGIN interface with EX/MEM pipeline register.
      input [31:0] ex_mem_operation,
      input [31:0] mem_wb_operation,
      input [7:0] ex_mem_data_top,
      input [7:0] ex_mem_data_bot,
      input [7:0] mem_wb_data_top, //Add this as a data passtrough in
EX/MEM register.
      input [7:0] mem_wb_data_bot,
      output [7:0] data_out_top,
      output [7:0] data_out_bot,
      output [4:0] sfr_input_sel,
      output [4:0] mem_str_data_sel_top,
      output [4:0] mem_str_data_sel_bot,
      output [3:0] mem_wb_data_sel_top,
      output [6:0] mem_wb_data_sel_bot
);

      wire [7:0] alu_data_in_top;
      wire [7:0] alu_data_in_bot;
      wire [7:0] alu_data_top_inv;
      //instantiate alu input multiplexor
      alu_input_mux alu_mux(
            //.clock(clock),
            .alu_input_sel_top(alu_top_sel),
            .alu_input_sel_bot(alu_bot_sel),
            .id_ex_data_top(data_in_top),
            .id_ex_data_bot(data_in_bot),
            .ex_mem_top(ex_mem_data_top),
            .ex_mem_bot(ex_mem_data_bot),
            .mem_wb_top(mem_wb_data_top),
            .mem_wb_bot(mem_wb_data_bot),
            .alu_data_input_top(alu_data_in_top),
            .alu_data_input_bot(alu_data_in_bot)
      );

      assign alu_data_top_inv[0] = (instruction[20] & ~alu_data_in_top[0])
| (alu_data_in_top[0] & ~instruction[20]);
      assign alu_data_top_inv[1] = (instruction[20] & ~alu_data_in_top[1])
| (alu_data_in_top[1] & ~instruction[20]);
      assign alu_data_top_inv[2] = (instruction[20] & ~alu_data_in_top[2])
```

```
| (alu_data_in_top[2] & ~instruction[20]);
      assign alu_data_top_inv[3] = (instruction[20] & ~alu_data_in_top[3])
| (alu_data_in_top[3] & ~instruction[20]);
      assign alu_data_top_inv[4] = (instruction[20] & ~alu_data_in_top[4])
| (alu_data_in_top[4] & ~instruction[20]);
      assign alu_data_top_inv[5] = (instruction[20] & ~alu_data_in_top[5])
| (alu_data_in_top[5] & ~instruction[20]);
      assign alu_data_top_inv[6] = (instruction[20] & ~alu_data_in_top[6])
| (alu_data_in_top[6] & ~instruction[20]);
      assign alu_data_top_inv[7] = (instruction[20] & ~alu_data_in_top[7])
| (alu_data_in_top[7] & ~instruction[20]);

      wire [15:0] alu_out;
      wire [2:0] alu_flags_out;
      //instantiate alu
      alu alu(
            .clock(clock),
            .nreset(nreset),
            .alu_operation_select(instruction[18]),
            .alu_operation(instruction[1:0]),
            .top_operand(alu_data_top_inv),
            .bottom_operand(alu_data_in_bot),
            .alu_flags(alu_flags_out),
            .alu_out(alu_out)
      );
      assign flags_out = alu_flags_out;

      //instantiate EX/MEM data input multiplexor.
      ex_mem_data_input_mux ex_mem_input_mux(
            //.clock(clock),
            .sel_signals(ex_mem_data_input_sel),
            .id_ex_top(data_in_top),
            .id_ex_bot(data_in_bot),
            .alu_res_top(alu_out[15:8]),
            .alu_res_bot(alu_out[7:0]),
            .ex_data_out_top(data_out_top),
            .ex_data_out_bot(data_out_bot)
      );

      //instantiate memory forwarding stub.
      memory_forwarding_logic mem_frwd(
            //.clock(),
            .instruction(instruction),
```

```
            .ex_mem_instruction(ex_mem_operation),
            .mem_wb_instruction(mem_wb_operation),
            .sfr_input_sel(sfr_input_sel),
            .mem_wb_data_sel_top(mem_wb_data_sel_top),
            .mem_wb_data_sel_bot(mem_wb_data_sel_bot),
            .mem_write_data_sel_top(mem_str_data_sel_top),
            .mem_write_data_sel_bot(mem_str_data_sel_bot)
      );

      wire [2:0] cur_flags;
      //instantiate flags register.
      flags_register flags_reg(
            .clock(clock),
            .nreset(nreset),
            .new_flags(alu_flags_out),
            .cur_flags(cur_flags)
      );

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("execute.vcd");
  $dumpvars (0, execute);
  #1;
end
`endif
*/
endmodule
```

6.2.5.1 Alu Input Selection Mux

Code:

```
/*
Module - EX/MEM Data Input Multiplexor
Author - Zach Walden
Last Changed - 2/17/22
Description - This module multiplexes the register file data values in the
ID/EX register and the alu results into the data inputs of the EX/MEM
register
Parameters -
*/
```

```
module alu_input_mux(
      //input clock,
      input [4:0] alu_input_sel_top,
      input [4:0] alu_input_sel_bot,
      input [7:0] id_ex_data_top,
      input [7:0] id_ex_data_bot,
      input [7:0] ex_mem_top,
      input [7:0] ex_mem_bot,
      input [7:0] mem_wb_top,
      input [7:0] mem_wb_bot,
      output reg [7:0] alu_data_input_top,
      output reg [7:0] alu_data_input_bot
);

      always @ (*)
      begin
            if(alu_input_sel_top == 5'b00001)
            begin
                  alu_data_input_top <= id_ex_data_top;
            end
            else if(alu_input_sel_top == 5'b00010)
            begin
                  alu_data_input_top <= ex_mem_top;
            end
            else if(alu_input_sel_top == 5'b00100)
            begin
                  alu_data_input_top <= ex_mem_bot;
            end
            else if(alu_input_sel_top == 5'b01000)
            begin
                  alu_data_input_top <= mem_wb_top;
            end
            else if(alu_input_sel_top == 5'b10000)
            begin
                  alu_data_input_top <= mem_wb_bot;
            end
            else
            begin
                  alu_data_input_top <= 8'h00;
            end
      end
```

```
      always @ (*)
      begin
            if(alu_input_sel_bot == 5'b00001)
            begin
                  alu_data_input_bot <= id_ex_data_bot;
            end
            else if(alu_input_sel_bot == 5'b00010)
            begin
                  alu_data_input_bot <= ex_mem_top;
            end
            else if(alu_input_sel_bot == 5'b00100)
            begin
                  alu_data_input_bot <= ex_mem_bot;
            end
            else if(alu_input_sel_bot == 5'b01000)
            begin
                  alu_data_input_bot <= mem_wb_top;
            end
            else if(alu_input_sel_bot == 5'b10000)
            begin
                  alu_data_input_bot <= mem_wb_bot;
            end
            else
            begin
                  alu_data_input_bot <= 8'h00;
            end
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("alu_input_mux.vcd");
  $dumpvars (0, alu_input_mux);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
```

```
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_alu_input_mux(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    await FallingEdge(dut.clock)
    dut.id_ex_data_top.value = 0x0FF
    dut.id_ex_data_bot.value = 0x0FF
    dut.ex_mem_top.value = 0x0EE
    dut.ex_mem_bot.value = 0x0EF
    dut.mem_wb_top.value = 0x0DD
    dut.mem_wb_bot.value = 0x0DF

    dut.alu_input_sel_top.value = 1
    dut.alu_input_sel_bot.value = 1

    await FallingEdge(dut.clock)
    assert dut.alu_data_input_top.value == 0x0FF, f"Mux Failed on Reg File
inputs"
    assert dut.alu_data_input_bot.value == 0x0FF, f"Mux failed on reg file
inputs"

    dut.alu_input_sel_top.value = 2
    dut.alu_input_sel_bot.value = 2

    await FallingEdge(dut.clock)
    assert dut.alu_data_input_top.value == 0x0EE, f"Mux Failed on
ex_mem_top inputs"
    assert dut.alu_data_input_bot.value == 0x0EE, f"Mux failed on
ex_mem_top inputs"

    dut.alu_input_sel_top.value = 4
    dut.alu_input_sel_bot.value = 4

    await FallingEdge(dut.clock)
    assert dut.alu_data_input_top.value == 0x0EF, f"Mux Failed on
ex_mem_bot inputs"
    assert dut.alu_data_input_bot.value == 0x0EF, f"Mux failed on
ex_mem_bot inputs"
```

```
    dut.alu_input_sel_top.value = 8
    dut.alu_input_sel_bot.value = 8

    await FallingEdge(dut.clock)
    assert dut.alu_data_input_top.value == 0x0DD, f"Mux Failed on
mem_wb_top inputs"
    assert dut.alu_data_input_bot.value == 0x0DD, f"Mux failed on
mem_wb_top inputs"

    dut.alu_input_sel_top.value = 16
    dut.alu_input_sel_bot.value = 16

    await FallingEdge(dut.clock)
    assert dut.alu_data_input_top.value == 0x0DF, f"Mux Failed on
mem_wb_bot inputs"
    assert dut.alu_data_input_bot.value == 0x0DF, f"Mux failed on
mem_wb_bot inputs"
```

6.2.5.2 Alu

Code:

```
module alu(
      input clock,
      input nreset,
      input alu_operation_select,
      input [1:0] alu_operation,
      input [7:0] top_operand,
      input [7:0] bottom_operand,
      output [2:0] alu_flags,
      output [15:0] alu_out
      );

      wire [3:0] fu_oe;

      //Instantiate Adder, Multiplier, Shifter, & Bit Manipulator.
      adder add_unit(
            .clock(clock),
            .nreset(nreset),
            .add_sub(alu_operation_select),
            .oe(fu_oe[0]),
```

```
        .primary_operand(top_operand),
        .secondary_operand(bottom_operand),
        .flags(alu_flags),
        .result(alu_out[7:0])
        );

    multiplier multiply_unit(
        .clock(clock),
        .nreset(nreset),
        .oe(fu_oe[1]),
        .primary_operand(top_operand),
        .secondary_operand(bottom_operand),
        .flags(alu_flags),
        .mult_out(alu_out)
        );

    bit_shifter shifter(
        .clock(clock),
        .nreset(nreset),
        .oe(fu_oe[2]),
        .right_left(alu_operation_select),
        .primary_operand(top_operand),
        .flags(alu_flags),
        .result(alu_out[7:0])
        );

    bitwise_logic_unit bitwise(
        .clock(clock),
        .nreset(nreset),
        .oe(fu_oe[3]),
        .and_or(alu_operation_select),
        .primary_operand(top_operand),
        .secondary_operand(bottom_operand),
        .flags(alu_flags),
        .result(alu_out[7:0])
        );

    //adder oe.
    assign fu_oe[0] = ~alu_operation[0] & ~alu_operation[1];
    //multiplier oe
    assign fu_oe[1] = ~alu_operation[0] & alu_operation[1];
    //bit shifter oe
    assign fu_oe[2] = alu_operation[0] & ~alu_operation[1];
```

```
      //bitwise logic oe
      assign fu_oe[3] = alu_operation[0] & alu_operation[1];

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("alu.vcd");
  $dumpvars (0, alu);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_alu(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.nreset.value = 1

    await FallingEdge(dut.clock)

    #test Adder/Subtractor (tests for carry & negative & zero)
    dut.alu_operation.value = 0 & 0x3

    # add 1 to ff this should result in over flow leaving the result to be
zero with the carry and zero flags set.
    dut.top_operand.value = 0x0FF
    dut.bottom_operand.value = 0x1
    dut.alu_operation_select.value = 0x1 & 0x1 #select an addition
    await FallingEdge(dut.clock)
    assert dut.alu_out.value.integer & 0x00FF == 0, f"addition result
incorrect"
    assert dut.alu_flags.value == 0x5, f"carry & zero not set in adder
```

```
test"

    dut.top_operand.value = 0x07E
    dut.bottom_operand.value = 0x1
    await FallingEdge(dut.clock)
    assert dut.alu_out.value.integer & 0x00FF == 0x07F, f"addition result
incorrect"
    assert dut.alu_flags.value == 0x0, f"carry & zero not set in adder
test"

    #set bottom_operand to 0 to test the negative flag. carry and zero
should be cleared.
    dut.top_operand.value = 0x0FF
    dut.bottom_operand.value = 0x00
    await FallingEdge(dut.clock)
    assert dut.alu_out.value.integer & 0x00FF == 0xFF, f"addition result
incorrect"
    assert dut.alu_flags.value == 0x2, f"negative not set in adder test"

    #test signed operation -1 = 0xFF
    dut.top_operand.value = 0x0FF
    dut.bottom_operand.value = 0x0FF
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == -2 & 0x0FF, f"two's complement
addition value failed"
    assert dut.alu_flags.value == 6, f"bad flags in -1 + -1 addition test"

    #BEGIN SUBTRACTION TESTS
    #test a non overflowed subtraction.
    dut.alu_operation_select.value = 0x0
    dut.top_operand.value = 0x1
    dut.bottom_operand.value = 0x1
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0, f"subtract 1 from 1 failed to
produce 0"
    assert dut.alu_flags.value == 1, f"zero flag not set"

    #test that, when correct, no flags are set
    dut.top_operand.value = 0x0A
    dut.bottom_operand.value = 0x1
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 9, f"subtract 1 from 9 failed to
produce 0"
```

```
    assert dut.alu_flags.value == 0, f"flag set"

    #Test an overflowed subtraction
    dut.top_operand.value = 0x1
    dut.bottom_operand.value = 0x0FF
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0x2, f"subtract 255 from 1 failed
to produce -254 in 9 bit form."
    assert dut.alu_flags.value == 4, f"flags not properly set in overflowed
subtraction test."

    #Test a subtraction that produces a carry out, but does not overflow
out of the 8-bit signed range. In this case 127 - 143 which produces -16 in
2's comp ie 0xF0
    dut.top_operand.value = 0x7F
    dut.bottom_operand.value = 0x8F
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0xF0, f"subtract 143 from 127
failed to produce -16"
    assert dut.alu_flags.value == 6, f"flags not set properly in non
overflow, carried out subtraction"

    #Perform a non overflow subtraction that produces a carry and a
negative value.
    dut.top_operand.value = 0x7F
    dut.bottom_operand.value = 0xFF
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0x80, f"subtract 255 from 255
failed to produce 0"
    assert dut.alu_flags.value == 6, f"flags not set properly in non
overflow, carried out subtraction"
    await FallingEdge(dut.clock)

    #Test Multiplier (tests for negative & zero)
    dut.alu_operation.value = 2 & 0x3

    dut.top_operand.value = 0x0FF
    dut.bottom_operand.value = 0x0FF
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FFFF == (0x0FF * 0x0FF) & 0x0FFFF,
f"multiply 255*255 failed"
    assert dut.alu_flags.value == 2, f"flags not set properly in 255*255
multiplication"
```

```
    #test multiplication by a 2's complement number THIS TEST WILL FAIL AS
THE SIMULATOR DOES NOT INFER SIGNED MULTIPLICATION, HOWEVER, IT MAY WORK ON
HARDWARE AS MY FPGA HAS A 25x18 signed hardware multiplier in each of its
120 DSP slices.:w
    dut.top_operand.value = 0x01
    dut.bottom_operand.value = 0x0FF
    await FallingEdge(dut.clock)
    #assert dut.alu_out.value & 0x0FFFF == (0x01 * (-1 & 0x0FF)) & 0x0FFFF,
f"multiply 1*-1 failed"
    #assert dut.alu_flags.value == 2, f"flags not set properly in 1*-1
multiplication"

    #test the zero flag
    dut.top_operand.value = 0x0FF
    dut.bottom_operand.value = 0x0
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FFFF == 0, f"multiply 255*0 failed"
    assert dut.alu_flags.value == 1, f"flags not set properly in 255*0
multiplication"

    dut.top_operand.value = 0x00F
    dut.bottom_operand.value = 0x0F
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FFFF == 225 & 0x0FFFF, f"multiply 15*15
failed"
    assert dut.alu_flags.value == 0, f"flags not set properly in 15*15
multiplication"

    #Test Shifter (tests for negative(SHL) & zero)
    dut.alu_operation.value = 1 & 0x3

    #test Shift right. NOTE negative flag can never be high.
    dut.alu_operation_select.value = 1

    dut.top_operand.value = 0x0FF
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0x7F, f"right shift failed"
    assert dut.alu_flags.value == 0x4, f"carry not set when a 1 is shifted
out SHR"

    dut.top_operand.value = 0x0FE
    await FallingEdge(dut.clock)
```

```
    assert dut.alu_out.value & 0x0FF == 0x7F, f"right shift failed"
    assert dut.alu_flags.value == 0x0, f"carry not set when a 1 is shifted
out SHR"

    dut.top_operand.value = 0x1
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0, f"right shift failed"
    assert dut.alu_flags.value == 0x5, f"zero not set when a 1 is shifted
out SHR"

    dut.top_operand.value = 0x0
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0, f"right shift failed"
    assert dut.alu_flags.value == 0x1, f"zero not set when a 1 is shifted
out SHR"

    #test Shift left, negative and zero can never be simultaneously set.
    dut.alu_operation_select.value = 0

    dut.top_operand.value = 0xFF
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0x0FE, f"right shift failed"
    assert dut.alu_flags.value == 0x6, f"zero not set when a 1 is shifted
out SHR"

    dut.top_operand.value = 0x7F
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0x0FE, f"right shift failed"
    assert dut.alu_flags.value == 0x2, f"zero not set when a 1 is shifted
out SHR"

    dut.top_operand.value = 0x80
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0, f"right shift failed"
    assert dut.alu_flags.value == 0x5, f"zero not set when a 1 is shifted
out SHR"

    dut.top_operand.value = 0x0
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0, f"right shift failed"
    assert dut.alu_flags.value == 0x1, f"zero not set when a 1 is shifted
out SHR"
```

```
    #Test Bitwise Logic (Check negative and zero flags)
    dut.alu_operation.value = 3 & 0x3

    #Test bitwise and. Zero and Negative may never be high at the same
time.
    dut.alu_operation_select.value = 1

    dut.top_operand.value = 0x0FF
    dut.bottom_operand.value = 0x0F
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 255 & 0x0F, f"bitwise and failed"
    assert dut.alu_flags.value == 0, f"flags not set properly in bitwise
and"

    dut.top_operand.value = 0x0FF
    dut.bottom_operand.value = 0x0F0
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 255 & 0x0F0, f"bitwise and failed"
    assert dut.alu_flags.value == 2, f"flags not set properly in bitwise
and"

    dut.top_operand.value = 0x0FF
    dut.bottom_operand.value = 0x0
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 255 & 0x0, f"bitwise and failed"
    assert dut.alu_flags.value == 1, f"flags not set properly in bitwise
and"

    #Test bitwise or. Zero and Negative may never be high at the same time.
    dut.alu_operation_select.value = 0

    dut.top_operand.value = 0x0F0
    dut.bottom_operand.value = 0x0F
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0x0F0 | 0x0F, f"bitwise or failed"
    assert dut.alu_flags.value == 2, f"flags not set properly in bitwise
or"

    dut.top_operand.value = 0x0
    dut.bottom_operand.value = 0x0
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0, f"bitwise or failed"
    assert dut.alu_flags.value == 1, f"flags not set properly in bitwise
```

```
or"

    dut.top_operand.value = 0x02A
    dut.bottom_operand.value = 0x55
    await FallingEdge(dut.clock)
    assert dut.alu_out.value & 0x0FF == 0x7F, f"bitwise or failed"
    assert dut.alu_flags.value == 0, f"flags not set properly in bitwise
or"
```

6.2.5.2.1 Adder

Code:

```
/*
Module - Adder
Author - Zach Walden
Last Changed - 1/28/22
Description - Adds or subtracts two operands. This module sets all 3 flags.
Parameters - input clock - system clock
           input nreset - system reset, active low.
           input add_sub - 0 results in a subtraction 1 results in an
addition.
           input oe - 1 results in the devices data being output to the alu
output bus. 0 high impedance.
           input [7:0] primary_operand - speakes for itself.
           input [7:0] secondary_operand - speaks for itself.
           output [2:0] flags - carry, negative, and zero flags in order
from most to least significant.
           output [7:0] result - a-bit adder result.
*/

module adder(
      input clock,
      input nreset,
      input add_sub,
      input oe,
      input [7:0] primary_operand,
      input [7:0] secondary_operand,
      output [2:0] flags,
      output [7:0] result
);

      wire [2:0] flag_result;
```

```
      reg [8:0] value;

      always @ (*)
      begin
            if(add_sub == 1'b1)
            begin
                  value <= primary_operand + secondary_operand;
            end
            else if(add_sub == 1'b0)
            begin
                  value <= primary_operand - secondary_operand;
            end
      end

    assign flag_result[0] = (value[7:0] == 8'h00) ? 1'b1 : 1'b0;
    assign flag_result[1] = value[7] ? 1'b1 : 1'b0;
    assign flag_result[2] = value[8];

      assign flags = oe ? flag_result : 3'bzzz;
      assign result = oe ? value[7:0] : 8'hZZ;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("adder.vcd");
  $dumpvars (0, adder);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_adder(dut):
    clock = Clock(dut.clock, 10, units="ns")
```

```
    cocotb.start_soon(clock.start())

    dut.oe.value = 1;

    print("Starting Addition Test")
    dut.nreset.value = 1
    dut.add_sub.value = 1
    pri = 0
    sec = 1

    for i in range(256):
        dut.primary_operand.value = pri
        dut.secondary_operand.value = sec
        res = addition(pri, sec)
        await FallingEdge(dut.clock)
        assert dut.result.value == res, f"addition was unsuccessful with
these operands: pri = {pri} sec = {sec}"
        pri = pri + 1

    print("Starting Subtraction Test")
    sec = 0xF0
    pri = 0x00
    dut.add_sub.value = 0

    for i in range(256):
        dut.primary_operand.value = pri
        dut.secondary_operand.value = sec
        res = subtraction(pri, sec)
        await FallingEdge(dut.clock)
        assert dut.result.value == res, f"Subtraction failed with these
operands pri = {pri} sec = {sec}"
        pri = pri + 1

    #test what happens when reset and oe are asserted.




def addition(A, B):
    return (A + B) & 0xFF

def subtraction(A, B):
    return (A - B) & 0xFF
```

6.2.5.2.2 Multiplier

Code:

```
/*
Module - Multiplier
Author - Zach Walden
Last Changed - 1/24/22 10:02 PM
Description - This module multiplies two 8-bit operands and produces a
16-bit result. It also checks & set the zero and negative flag accordingly.
Parameters -      input [7:0] primary_operand - primary operand
            input [7:0] secondary_operand - secondary operand
            input enable - unit enable signal. This comes from the alu
operation decode logic, it it asyncronous and is assumed to be valid on the
rising edge of hte devices clock.
            output [2:0] flags - condition flags. This unit only accesses
the zero flag - flags[0], and the negative flag - flags[1].
            output [7:0] result_low - This is the least significant byte of
the result of multiplication.
            output [7:0] result_high - This is the most significant byte of
the multiplication.
*/

module multiplier(
      input clock,
      input nreset,
      input [7:0] primary_operand,
      input [7:0] secondary_operand,
      input oe,
      output [2:0] flags,
      output [15:0] mult_out
);

      reg [15:0] result;
      wire [2:0] flags_result;

      always @ (*)
      begin
            result <= primary_operand * secondary_operand;
      end

    assign flags_result[0] = (result == 16'h0000) ? 1'b1 : 1'b0;
    assign flags_result[1] = result[15] ? 1'b1 : 1'b0;
```

```
    assign flags_result[2] = 0;

      assign flags = oe ? flags_result : 3'bzzz;
      assign mult_out = oe ? result : 16'h00ZZ; //high byte is set to zero
for 2 reasons, potential metastability that should be eliminated by control
signals, and for my testbenches.

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("multiplier.vcd");
  $dumpvars (0, multiplier);
  #1;
end
`endif
*/
endmodule
```

6.2.5.2.3 Bit Shifter

Code:

```
/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/

module bit_shifter(
      input clock,
      input nreset,
      inout oe,
      input right_left,
      input [7:0] primary_operand,
      output [2:0] flags,
      output [7:0] result
);

    integer Index;
```

```
      wire [2:0] flags_value;
      reg [7:0] value;

      always @ (*)
      begin
            if(right_left == 1'b1)
            begin
                  //shift right
                  value[6:0] <= primary_operand[7:1];
                  value[7] <= 1'b0;
            end
            else if (right_left == 1'b0)
            begin
                  //shift left
                  value[7:1] <= primary_operand[6:0];
                  value[0] <= 1'b0;
            end
      end

      //assign the zero flag
    assign flags_value[0] = (value[7:0] == 8'h00) ? 1'b1 : 1'b0;
    //assign the negative flag
    assign flags_value[1] = value[7] ? 1'b1 : 1'b0;
    //if shift left carry flag == high bit, if shift right, carry flag =
low bit. both of the input operand.
    assign flags_value[2] = (primary_operand[7] & ~right_left) |
(primary_operand[0] & right_left);

      assign result = oe ? value : 8'hZZ;
      assign flags = oe ? flags_value : 3'bzzz;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("bit_shifter.vcd");
  $dumpvars (0, bit_shifter);
  #1;
end
`endif
*/
endmodule
```

6.2.5.2.4 Bitwise Logic Unit

Code:

```
/*
Module - Bitwise Logic Unit
Author - Zach Walden
Last Changed - 1/25/22 11:47 PM
Description -
Parameters -
*/

module bitwise_logic_unit(
            input clock,
            input nreset,
            input oe,
            input and_or,
            input [7:0] primary_operand,
            input [7:0] secondary_operand,
            output [2:0] flags,
            output [7:0] result
);

      reg [7:0] value;
      wire [2:0] flags_result;

      always @ (*)
      begin
            if(and_or == 1'b1)
            begin
                  value = primary_operand & secondary_operand;
            end
            else
            begin
                  value = primary_operand | secondary_operand;
            end
      end

    assign flags_result[0] = (value == 8'h00) ? 1'b1 : 1'b0;
    assign flags_result[1] = value[7] ? 1'b1 : 1'b0;
    assign flags_result[2] = 0;  //there will never be a carry set when
performing bitwise logic.
```

```
      assign result = oe ? value : 8'hZZ;
      assign flags = oe ? flags_result : 3'bzzz;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("bitwise_logic_unit.vcd");
  $dumpvars (0, bitwise_logic_unit);
  #1;
end
`endif
*/
endmodule
```

6.2.5.3 Memory Forwarding Logic

Code:

```
/*
Module - Memory Forwarding Logic
Author - Zach Walden
Last Changed - 3/21/22
Description - Checks for pipline hazards relating to stores in either the
memory banks or in the sfr.
Parameters -
*/

module memory_forwarding_logic(
      //input clock,
      input [31:0] instruction,
      input [31:0] ex_mem_instruction,
      input [31:0] mem_wb_instruction,
      output reg [4:0] sfr_input_sel,
      output reg [3:0] mem_wb_data_sel_top,
      output reg [6:0] mem_wb_data_sel_bot,
      output reg [4:0] mem_write_data_sel_top,
      output reg [4:0] mem_write_data_sel_bot
);

      always @ (*)
      begin
            case(instruction[7:0])
```

```
                  //Load, Load Framebuffer, Pop
                  8'hFB :
                  begin
                        sfr_input_sel <= 5'b00001;
                        mem_write_data_sel_top <= 5'b00001;
                        mem_write_data_sel_bot <= 5'b00001;
                        mem_wb_data_sel_top <= 4'b0010;           //Select
both load results for simplicity.
                        mem_wb_data_sel_bot <= 7'b0000100;
                  end
                  //Load Program Memory
                  8'hF9 :
                  begin
                        sfr_input_sel <= 5'b00001;
                        mem_write_data_sel_top <= 5'b00001;
                        mem_write_data_sel_bot <= 5'b00001;
                        mem_wb_data_sel_top <= 4'b0001;           //Select
load bottom result
                        mem_wb_data_sel_bot <= 7'b0000100;
                  end
                  //Store, Store Framebuffer, Push
                  8'hC6 :
                  begin
                        //Check if store or store framebuffer?
                        if(instruction[20] == 1'b1)
                        begin
                              //Normal Store SINGLE WRITE
                              //EX/MEM
                              //SINGLE WRITE INSTRUCTIONS, INC, DEC, ADD,
ADDI, SUB, SUBI, CP, CPI, AND, ANDI, OR, ORI, SHR, SHL, COM, INV, LD, POP,
LPM, MOVR, OUT
                              if((ex_mem_instruction[7:0] == 8'hBC) ||
(ex_mem_instruction[7:0] == 8'h80) || (ex_mem_instruction[7:0] == 8'h97) ||
(ex_mem_instruction[7:0] == 8'h9B) || (ex_mem_instruction[7:0] == 8'hA5) ||
(ex_mem_instruction[7:0] == 8'hFB && ex_mem_instruction[20] == 1'b1) ||
(ex_mem_instruction[7:0] == 8'hF9) || (ex_mem_instruction[7:0] == 8'hF8) ||
(ex_mem_instruction[7:0] == 8'h9C && ex_mem_instruction[19:18] == 2'b00) ||
(ex_mem_instruction[7:0] == 8'h9C && ex_mem_instruction[19:18] == 2'b10))
                              begin
                                    if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward MEM/WB data bottom to
```

```
mem str data bottom
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00100;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                                  else
                                  begin
                                        //No forward necessary
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                            end
                            //DOUBLE WRITE INSTRUCTIONS LDFB, MUL, MULI
                            else if((ex_mem_instruction[7:0] == 8'hFB &&
ex_mem_instruction[20] == 1'b0) || (ex_mem_instruction[7:0] ==  8'h8E) ||
(ex_mem_instruction[7:0] == 8'h9E))
                            begin
                                  if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                  begin
                                        //Forward mem_wb data bot to mem
str data bottom
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00100;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                                  else if(instruction[17:13] ==
```

```
ex_mem_instruction[17:13])
                                    begin
                                          //Forward mem_wb data top to mem
str data bottom
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00010;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                              end
                              //MEM/WB
                              else if((mem_wb_instruction[7:0] == 8'hBC) ||
(mem_wb_instruction[7:0] == 8'h80) || (mem_wb_instruction[7:0] == 8'h97) ||
(mem_wb_instruction[7:0] == 8'h9B) || (mem_wb_instruction[7:0] == 8'hA5) ||
(mem_wb_instruction[7:0] == 8'hFB && mem_wb_instruction[20] == 1'b1) ||
(mem_wb_instruction[7:0] == 8'hF9) || (mem_wb_instruction[7:0] == 8'hF8) ||
(mem_wb_instruction[7:0] == 8'h9C && mem_wb_instruction[19:18] == 2'b00) ||
(mem_wb_instruction[7:0] == 8'h9C && mem_wb_instruction[19:18] == 2'b10))
                              begin
                                    if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                    begin
                                          //Forward MEM/WB tm1 data bottom
to mem str data bottom
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
```

```
                                        mem_write_data_sel_bot <=
5'b10000;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                                  else
                                  begin
                                        //No forward necessary
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                            end
                            //DOUBLE WRITE INSTRUCTIONS LDFB, MUL, MULI
                            else if((mem_wb_instruction[7:0] == 8'hFB &&
mem_wb_instruction[20] == 1'b0) || (mem_wb_instruction[7:0] ==  8'h8E) ||
(mem_wb_instruction[7:0] == 8'h9E))
                            begin
                                  if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                  begin
                                        //Forward mem_wb tm1 data bot to
mem str data bottom
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b10000;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                                  else if(instruction[17:13] ==
mem_wb_instruction[17:13])
                                  begin
                                        //Forward mem_wb tm1 data top to
mem str data bottom
```

```
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b01000;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                                  else
                                  begin
                                        //No forward necessary
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                            end
                            else
                            begin
                                  //No forward necessary
                                  sfr_input_sel <= 5'b00001;
                                  mem_write_data_sel_top <= 5'b00001;
                                  mem_write_data_sel_bot <= 5'b00001;
                                  mem_wb_data_sel_top <= 4'b0001;
      //Simply pass EX/MEM through
                                  mem_wb_data_sel_bot <= 7'b0000010;
                            end
                      end
                      else
                      begin
                            //Store Framebuffer DOUBLE WRITE
                            //EX/MEM
                            //SINGLE WRITE INSTRUCTIONS, INC, DEC, ADD,
ADDI, SUB, SUBI, CP, CPI, AND, ANDI, OR, ORI, SHR, SHL, COM, INV, LD, POP,
LPM, MOVR, OUT
                            if((ex_mem_instruction[7:0] == 8'hBC) ||
(ex_mem_instruction[7:0] == 8'h80) || (ex_mem_instruction[7:0] == 8'h97) ||
(ex_mem_instruction[7:0] == 8'h9B) || (ex_mem_instruction[7:0] == 8'hA5) ||
```

```
(ex_mem_instruction[7:0] == 8'hFB && ex_mem_instruction[20] == 1'b1) ||
(ex_mem_instruction[7:0] == 8'hF9) || (ex_mem_instruction[7:0] == 8'hF8) ||
(ex_mem_instruction[7:0] == 8'h9C && ex_mem_instruction[19:18] == 2'b00) ||
(ex_mem_instruction[7:0] == 8'h9C && ex_mem_instruction[19:18] == 2'b10))
                              begin
                                    if(instruction[12:8] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward MEM/WB bottom to mem str
data bottom
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00100;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward MEM/WB bottom to mem str
data top
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00100;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else
                                    begin
                                          //No forwarding necessary
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
```

```
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                              end
                              //DOUBLE WRITE INSTRUCTIONS LDFB, MUL, MULI
                              else if((ex_mem_instruction[7:0] == 8'hFB &&
ex_mem_instruction[20] == 1'b0) || (ex_mem_instruction[7:0] ==  8'h8E) ||
(ex_mem_instruction[7:0] == 8'h9E))
                              begin
                                    if(instruction[12:8] ==
ex_mem_instruction[12:8])
                                    begin
                                          if(instruction[17:13] ==
ex_mem_instruction[17:13])
                                          begin
                                                //Forward MEM/WB data top &
bot to mem_str data top & bot
                                                sfr_input_sel <= 5'b00001;
                                                mem_write_data_sel_top <=
5'b00010;
                                                mem_write_data_sel_bot <=
5'b00100;
                                                mem_wb_data_sel_top <=
4'b0001;          //Simply pass EX/MEM through
                                                mem_wb_data_sel_bot <=
7'b0000010;
                                          end
                                          else
                                          begin
                                                //Forward MEM/WB bot to
mem_str data bot
                                                sfr_input_sel <= 5'b00001;
                                                mem_write_data_sel_top <=
5'b00001;
                                                mem_write_data_sel_bot <=
5'b00100;
                                                mem_wb_data_sel_top <=
4'b0001;          //Simply pass EX/MEM through
                                                mem_wb_data_sel_bot <=
7'b0000010;
                                          end
                                    end
                                    else if(instruction[12:8] ==
ex_mem_instruction[17:13])
```

```
                                    begin
                                          if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                          begin
                                                //Forward MEM/WB data top &
bot to mem_str data bot & top
                                                sfr_input_sel <= 5'b00001;
                                                mem_write_data_sel_top <=
5'b00100;
                                                mem_write_data_sel_bot <=
5'b00010;
                                                mem_wb_data_sel_top <=
4'b0001;          //Simply pass EX/MEM through
                                                mem_wb_data_sel_bot <=
7'b0000010;
                                          end
                                          else
                                          begin
                                                //Forward MEM/WB top to
mem_str data bot
                                                sfr_input_sel <= 5'b00001;
                                                mem_write_data_sel_top <=
5'b00001;
                                                mem_write_data_sel_bot <=
5'b00010;
                                                mem_wb_data_sel_top <=
4'b0001;          //Simply pass EX/MEM through
                                                mem_wb_data_sel_bot <=
7'b0000010;
                                          end
                                    end
                                    else if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward MEM/WB data bot to
mem_str data top
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00100;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
```

```
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else if(instruction[17:13] ==
ex_mem_instruction[17:13])
                                    begin
                                        //Forward MEM/WB data top to
mem_str data top
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00010;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else
                                    begin
                                        //No forward necessary
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                end
                                //MEM/WB
                                else if((mem_wb_instruction[7:0] == 8'hBC) ||
(mem_wb_instruction[7:0] == 8'h80) || (mem_wb_instruction[7:0] == 8'h97) ||
(mem_wb_instruction[7:0] == 8'h9B) || (mem_wb_instruction[7:0] == 8'hA5) ||
(mem_wb_instruction[7:0] == 8'hFB && mem_wb_instruction[20] == 1'b1) ||
(mem_wb_instruction[7:0] == 8'hF9) || (mem_wb_instruction[7:0] == 8'hF8) ||
(mem_wb_instruction[7:0] == 8'h9C && mem_wb_instruction[19:18] == 2'b00) ||
(mem_wb_instruction[7:0] == 8'h9C && mem_wb_instruction[19:18] == 2'b10))
                                begin
                                    if(instruction[12:8] ==
mem_wb_instruction[12:8])
                                    begin
                                        //Forward MEM/WB tm1 bottom to mem
str data bottom
```

```
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b10000;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;

                                    end
                                    else if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                    begin
                                        //Forward MEM/WB tm1 bottom to mem
str data top
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b10000;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else
                                    begin
                                        //No forwarding necessary
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                end
                                //DOUBLE WRITE INSTRUCTIONS LDFB, MUL, MULI
                                else if((mem_wb_instruction[7:0] == 8'hFB &&
mem_wb_instruction[20] == 1'b0) || (mem_wb_instruction[7:0] ==  8'h8E) ||
(mem_wb_instruction[7:0] == 8'h9E))
                                begin
                                        if(instruction[12:8] ==
```

```
mem_wb_instruction[12:8])
                                    begin
                                          if(instruction[17:13] ==
mem_wb_instruction[17:13])
                                          begin
                                                //Forward MEM/WB tm1 data
top & bot to mem_str data top & bot
                                                sfr_input_sel <= 5'b00001;
                                                mem_write_data_sel_top <=
5'b01000;
                                                mem_write_data_sel_bot <=
5'b10000;
                                                mem_wb_data_sel_top <=
4'b0001;          //Simply pass EX/MEM through
                                                mem_wb_data_sel_bot <=
7'b0000010;
                                          end
                                          else
                                          begin
                                                //Forward MEM/WB tm1 bot to
mem_str data bot
                                                sfr_input_sel <= 5'b00001;
                                                mem_write_data_sel_top <=
5'b00001;
                                                mem_write_data_sel_bot <=
5'b10000;
                                                mem_wb_data_sel_top <=
4'b0001;          //Simply pass EX/MEM through
                                                mem_wb_data_sel_bot <=
7'b0000010;
                                          end
                                    end
                                    else if(instruction[12:8] ==
mem_wb_instruction[17:13])
                                    begin
                                          if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                          begin
                                                //Forward MEM/WB tm1 data
top & bot to mem_str data bot & top
                                                sfr_input_sel <= 5'b00001;
                                                mem_write_data_sel_top <=
5'b10001;
```

```
                                                mem_write_data_sel_bot <=
5'b01000;
                                                mem_wb_data_sel_top <=
4'b0001;          //Simply pass EX/MEM through
                                                mem_wb_data_sel_bot <=
7'b0000010;
                                          end
                                          else
                                          begin
                                                //Forward MEM/WB tm1 top to
mem_str data bot
                                                sfr_input_sel <= 5'b00001;
                                                mem_write_data_sel_top <=
5'b00001;
                                                mem_write_data_sel_bot <=
5'b01000;
                                                mem_wb_data_sel_top <=
4'b0001;          //Simply pass EX/MEM through
                                                mem_wb_data_sel_bot <=
7'b0000010;
                                          end
                                    end
                                    else if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                    begin
                                          //Forward MEM/WB tm1 data bot to
mem_str data top
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b10000;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else if(instruction[17:13] ==
mem_wb_instruction[17:13])
                                    begin
                                          //Forward MEM/WB tm1  data top to
mem_str data top
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
```

```
5'b00001;
                                    mem_write_data_sel_bot <=
5'b10000;
                                    mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                    mem_wb_data_sel_bot <= 7'b0000010;
                                end
                                else
                                begin
                                    //No forward necessary
                                    sfr_input_sel <= 5'b00001;
                                    mem_write_data_sel_top <=
5'b00001;
                                    mem_write_data_sel_bot <=
5'b00001;
                                    mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                    mem_wb_data_sel_bot <= 7'b0000010;
                                end
                            end
                            else
                            begin
                                //No forward necessary
                                sfr_input_sel <= 5'b00001;
                                mem_write_data_sel_top <= 5'b00001;
                                mem_write_data_sel_bot <= 5'b00001;
                                mem_wb_data_sel_top <= 4'b0001;
      //Simply pass EX/MEM through
                                mem_wb_data_sel_bot <= 7'b0000010;
                            end
                        end
                    end
                    //MOVR, OUT, IN
                    8'h9C :
                    begin
                        //In
                        if(instruction[19:18] == 2'b10)
                        begin
                            //No forward necessary
                            sfr_input_sel <= 5'b00001;
                            mem_write_data_sel_top <= 5'b00001;
                            mem_write_data_sel_bot <= 5'b00001;
                            mem_wb_data_sel_top <= 4'b0001;
```

```
//Simply pass EX/MEM through
                              mem_wb_data_sel_bot <= 7'b0000001;
//Select SFR read data
                        end
                        //Out
                        else if(instruction[19:18] == 2'b01)
                        begin
                              //EX/MEM
                              //SINGLE WRITE INSTRUCTIONS, INC, DEC, ADD,
ADDI, SUB, SUBI, CP, CPI, AND, ANDI, OR, ORI, SHR, SHL, COM, INV, LD, POP,
LPM, MOVR, OUT
                              if((ex_mem_instruction[7:0] == 8'hBC) ||
(ex_mem_instruction[7:0] == 8'h80) || (ex_mem_instruction[7:0] == 8'h97) ||
(ex_mem_instruction[7:0] == 8'h9B) || (ex_mem_instruction[7:0] == 8'hA5) ||
(ex_mem_instruction[7:0] == 8'hFB && ex_mem_instruction[20] == 1'b1) ||
(ex_mem_instruction[7:0] == 8'hF9) || (ex_mem_instruction[7:0] == 8'hF8) ||
(ex_mem_instruction[7:0] == 8'h9C && ex_mem_instruction[19:18] == 2'b00) ||
(ex_mem_instruction[7:0] == 8'h9C && ex_mem_instruction[19:18] == 2'b10))
                              begin
                                    if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward MEM/WB data bottom to
sfr input
                                          sfr_input_sel <= 5'b00100;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
```

```
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                              end
                              //DOUBLE WRITE INSTRUCTIONS LDFB, MUL, MULI
                              else if((ex_mem_instruction[7:0] == 8'hFB &&
ex_mem_instruction[20] == 1'b0) || (ex_mem_instruction[7:0] ==  8'h8E) ||
(ex_mem_instruction[7:0] == 8'h9E))
                              begin
                                    if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward mem_wb data bot to sfr
input
                                          sfr_input_sel <= 5'b00100;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else if(instruction[17:13] ==
ex_mem_instruction[17:13])
                                    begin
                                          //Forward mem_wb data top to sfr
input
                                          sfr_input_sel <= 5'b00010;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
```

```
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                            end
                            //MEM/WB
                            else if((mem_wb_instruction[7:0] == 8'hBC) ||
(mem_wb_instruction[7:0] == 8'h80) || (mem_wb_instruction[7:0] == 8'h97) ||
(mem_wb_instruction[7:0] == 8'h9B) || (mem_wb_instruction[7:0] == 8'hA5) ||
(mem_wb_instruction[7:0] == 8'hFB && mem_wb_instruction[20] == 1'b1) ||
(mem_wb_instruction[7:0] == 8'hF9) || (mem_wb_instruction[7:0] == 8'hF8) ||
(mem_wb_instruction[7:0] == 8'h9C && mem_wb_instruction[19:18] == 2'b00) ||
(mem_wb_instruction[7:0] == 8'h9C && mem_wb_instruction[19:18] == 2'b10))
                            begin
                                  if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                  begin
                                        //Forward MEM/WB tm1 data bottom
to sfr input
                                        sfr_input_sel <= 5'b10000;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                                  else
                                  begin
                                        //No forward necessary
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                            end
                            //DOUBLE WRITE INSTRUCTIONS LDFB, MUL, MULI
```

```
                              else if((mem_wb_instruction[7:0] == 8'hFB &&
mem_wb_instruction[20] == 1'b0) || (mem_wb_instruction[7:0] ==  8'h8E) ||
(mem_wb_instruction[7:0] == 8'h9E))
                              begin
                                    if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                    begin
                                          //Forward mem_wb tm1 data bot to
sfr input
                                          sfr_input_sel <= 5'b10000;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else if(instruction[17:13] ==
mem_wb_instruction[17:13])
                                    begin
                                          //Forward mem_wb tm1 data top to
sfr input
                                          sfr_input_sel <= 5'b01000;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
```

```
                                    end
                              end
                              else
                              begin
                                    //No forward necessary
                                    sfr_input_sel <= 5'b00001;
                                    mem_write_data_sel_top <= 5'b00001;
                                    mem_write_data_sel_bot <= 5'b00001;
                                    mem_wb_data_sel_top <= 4'b0001;
      //Simply pass EX/MEM through
                                    mem_wb_data_sel_bot <= 7'b0000010;
                              end
                        end
                        //TODO This module needs to handle the creation of
the signal that controls the mem/wb data input mux. to do this, a control
signal will be added to this bus along with cases to handle the creation of
the signal for anny memory loads, and sfr reads. On top of that, the mem/wb
data input mux will need to be rewritten to handsle the mem/wb time minus 1
(tm1) values
                        //MOVR
                        else if(instruction[19:18] == 2'b00)
                        begin
                              //EX/MEM
                              //SINGLE WRITE INSTRUCTIONS, INC, DEC, ADD,
ADDI, SUB, SUBI, CP, CPI, AND, ANDI, OR, ORI, SHR, SHL, COM, INV, LD, LDI,
POP, LPM, MOVR, OUT
                              if((ex_mem_instruction[7:0] == 8'hBC) ||
(ex_mem_instruction[7:0] == 8'h80) || (ex_mem_instruction[7:0] == 8'h97) ||
(ex_mem_instruction[7:0] == 8'h9B) || (ex_mem_instruction[7:0] == 8'hA5) ||
(ex_mem_instruction[7:0] == 8'hFB && ex_mem_instruction[20] == 1'b1) ||
(ex_mem_instruction[7:0] == 8'hF9) || (ex_mem_instruction[7:0] == 8'hF8) ||
(ex_mem_instruction[7:0] == 8'h9C && ex_mem_instruction[19:18] == 2'b00) ||
(ex_mem_instruction[7:0] == 8'h9C && ex_mem_instruction[19:18] == 2'b10))
                              begin
                                    if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                    begin
                                          //Forward MEM/WB data bottom to
mem str data bottom
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
```

```
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0010000;
                                  end
                                  else
                                  begin
                                        //No forward necessary
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                            end
                            //DOUBLE WRITE INSTRUCTIONS LDFB, MUL, MULI
                            else if((ex_mem_instruction[7:0] == 8'hFB &&
ex_mem_instruction[20] == 1'b0) || (ex_mem_instruction[7:0] ==  8'h8E) ||
(ex_mem_instruction[7:0] == 8'h9E))
                            begin
                                  if(instruction[17:13] ==
ex_mem_instruction[12:8])
                                  begin
                                        //Forward mem_wb data bot to
mem/wb data bottom
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0010000;
                                  end
                                  else if(instruction[17:13] ==
ex_mem_instruction[17:13])
                                  begin
                                        //Forward mem_wb data top to mem
str data bottom
                                        sfr_input_sel <= 5'b00001;
```

```
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0001000;
                                  end
                                  else
                                  begin
                                        //No forward necessary
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b0000010;
                                  end
                              end
                              //MEM/WB
                              else if((mem_wb_instruction[7:0] == 8'hBC) ||
(mem_wb_instruction[7:0] == 8'h80) || (mem_wb_instruction[7:0] == 8'h97) ||
(mem_wb_instruction[7:0] == 8'h9B) || (mem_wb_instruction[7:0] == 8'hA5) ||
(mem_wb_instruction[7:0] == 8'hFB && mem_wb_instruction[20] == 1'b1) ||
(mem_wb_instruction[7:0] == 8'hF9) || (mem_wb_instruction[7:0] == 8'hF8) ||
(mem_wb_instruction[7:0] == 8'h9C && mem_wb_instruction[19:18] == 2'b00) ||
(mem_wb_instruction[7:0] == 8'h9C && mem_wb_instruction[19:18] == 2'b10))
                              begin
                                  if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                  begin
                                        //Forward MEM/WB tm1 data bottom
to mem/wb data bottom
                                        sfr_input_sel <= 5'b00001;
                                        mem_write_data_sel_top <=
5'b00001;
                                        mem_write_data_sel_bot <=
5'b00001;
                                        mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                        mem_wb_data_sel_bot <= 7'b1000000;
```

```
                                    end
                                    else
                                    begin
                                          //No forward necessary
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b0000010;
                                    end
                              end
                              //DOUBLE WRITE INSTRUCTIONS LDFB, MUL, MULI
                              else if((mem_wb_instruction[7:0] == 8'hFB &&
mem_wb_instruction[20] == 1'b0) || (mem_wb_instruction[7:0] ==  8'h8E) ||
(mem_wb_instruction[7:0] == 8'h9E))
                              begin
                                    if(instruction[17:13] ==
mem_wb_instruction[12:8])
                                    begin
                                          //Forward mem_wb tm1 data bot to
mem/wb data bottom
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
                                          mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                          mem_wb_data_sel_bot <= 7'b1000000;
                                    end
                                    else if(instruction[17:13] ==
mem_wb_instruction[17:13])
                                    begin
                                          //Forward mem_wb tm1 data top to
mem/wb data bottom
                                          sfr_input_sel <= 5'b00001;
                                          mem_write_data_sel_top <=
5'b00001;
                                          mem_write_data_sel_bot <=
5'b00001;
```

```
                                    mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                    mem_wb_data_sel_bot <= 7'b0100000;
                                end
                                else
                                begin
                                    //No forward necessary
                                    sfr_input_sel <= 5'b00001;
                                    mem_write_data_sel_top <=
5'b00001;
                                    mem_write_data_sel_bot <=
5'b00001;
                                    mem_wb_data_sel_top <= 4'b0001;
            //Simply pass EX/MEM through
                                    mem_wb_data_sel_bot <= 7'b0000010;
                                end
                            end
                            else
                            begin
                                //No forward necessary
                                sfr_input_sel <= 5'b00001;
                                mem_write_data_sel_top <= 5'b00001;
                                mem_write_data_sel_bot <= 5'b00001;
                                mem_wb_data_sel_top <= 4'b0001;
      //Simply pass EX/MEM through
                                mem_wb_data_sel_bot <= 7'b0000010;
                            end
                        end
                        else
                        begin
                            //No forward needed
                            sfr_input_sel <= 5'b00001;
                            mem_write_data_sel_top <= 5'b00001;
                            mem_write_data_sel_bot <= 5'b00001;
                            mem_wb_data_sel_top <= 4'b0001;
//Simply pass EX/MEM through
                            mem_wb_data_sel_bot <= 7'b0000010;
                        end
                    end
                    //Default Case
                    default
                    begin
                        //No forward needed
```

```
                        sfr_input_sel <= 5'b00001;
                        mem_write_data_sel_top <= 5'b00001;
                        mem_write_data_sel_bot <= 5'b00001;
                        mem_wb_data_sel_top <= 4'b0001;           //Simply
pass EX/MEM through
                        mem_wb_data_sel_bot <= 7'b0000010;
                end
            endcase
      end


/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("memory_forwarding_logic.vcd");
  $dumpvars (0,memory_forwarding_logic);
  #1;
end
`endif
*/
endmodule
```

6.2.5.4 EX/MEM Data Input Selection Mux

Code:

```
/*
Module - EX/MEM Data Input Multiplexor
Author - Zach Walden
Last Changed - 2/17/22
Description - This module multiplexes the register file data values in the
ID/EX register and the alu results into the data inputs of the EX/MEM
register
Parameters -
*/

module ex_mem_data_input_mux(
      //input clock,
      input [1:0] sel_signals,           //4 bits the low bit selects
between the alu result and ID/EX data top values,
      input [7:0] id_ex_top,
      input [7:0] id_ex_bot,
```

```
      input [7:0] alu_res_top,
      input [7:0] alu_res_bot,
      output reg [7:0] ex_data_out_top,
      output reg [7:0] ex_data_out_bot
);

      always @ (*)
      begin
            if(sel_signals[0] == 1'b1)
            begin
                  ex_data_out_top <= alu_res_top;
            end
            else
            begin
                  ex_data_out_top <= id_ex_top;
            end
      end

      always @ (*)
      begin
            if(sel_signals[1] == 1'b1)
            begin
                  ex_data_out_bot <= alu_res_bot;
            end
            else
            begin
                  ex_data_out_bot <= id_ex_bot;
            end
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("ex_mem_data_input_mux.vcd");
  $dumpvars (0, ex_mem_data_input_mux);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_ex_mem_data_input_mux(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    await FallingEdge(dut.clock)

    dut.reg_file_top.value = 0x0AA
    dut.reg_file_bot.value = 0x0AA
    dut.alu_res_top.value = 0x0FF
    dut.alu_res_bot.value = 0x0FF
    dut.immeadiate.value = 0x0EE

    dut.sel_signals.value = 2

    await FallingEdge(dut.clock)
    assert dut.ex_data_out_top.value == 0x0AA, f"Muxing Failed on the top
byte"
    assert dut.ex_data_out_bot.value == 0x0AA, f"Muxing Failed on the
bottom byte"

    await FallingEdge(dut.clock)
    dut.sel_signals.value = 5

    await FallingEdge(dut.clock)
    assert dut.ex_data_out_top.value == 0x0FF, f"Muxing Failed on the top
byte"
    assert dut.ex_data_out_bot.value == 0x0FF, f"Muxing Failed on the
bottom byte"

    await FallingEdge(dut.clock)
    dut.sel_signals.value = 9

    await FallingEdge(dut.clock)
    assert dut.ex_data_out_top.value == 0x0FF, f"Muxing Failed on the top
byte"
    assert dut.ex_data_out_bot.value == 0x0EE, f"Muxing Failed on the
bottom byte"
```

6.2.6 EX/MEM Register

Code:

```
/*
Module - Execution/Memory Pipeline Register
Author - Zach Walden
Last Changed - 2/12/22, 3/27/22
Description - This register holds the necessary data to ensure that the
correct results exit the memory pipeline stage.
*/

module ex_mem(
	input clock,				//System Clock
	input nreset,				//System Reset Signal
	input [7:0] data_top_in,		//I/O for the top register file
operand read.
	output reg [7:0] data_top_out = 0,
	input [7:0] data_bot_in,		//I/O for the bottom register file
operand read.
	output reg [7:0] data_bot_out = 0,
	input [31:0] instruction_in,		//I/O for the instruction word.
	output reg [31:0] instruction_out = 0,
	input mem_wen_in,
	output reg mem_wen_out,
	input main_memory_enable_in,		//BEGIN These Signals Are to be
sent to the memory i/o unit rather than the memory stage.//Consumed in
memory.
	output reg main_memory_enable_out = 0,
	input frame_buffer_enable_in,			//Consumed in memory.
	output reg frame_buffer_enable_out = 0,
	input call_stack_enable_in,		//Consumed in memory.
	output reg call_stack_enable_out = 0,		//END MEM/IO Signals
	input prog_mem_enable_in,		//Consumed in memory.
	output reg prog_mem_enable_out = 0,
	input [6:0] mem_ptr_ctl_in,			//Consumed in memory
	output reg [6:0] mem_ptr_ctl_out = 0,
	input call_stk_addr_sel_in,		//Consumed in memory
	output reg call_stk_addr_sel_out = 0,
	input stk_addr_sel_in,			//Consumed in memory
	output reg stk_addr_sel_out = 0,
	input [3:0] mem_wb_data_sel_top_in,	//Consumed in memory.
```

```
      output reg [3:0] mem_wb_data_sel_top_out = 0,
      input [6:0] mem_wb_data_sel_bot_in,       //Consumed in memory.
      output reg [6:0] mem_wb_data_sel_bot_out = 0,
      input [4:0] sfr_file_input_sel_in,        //Consumed in memory.
      output reg [4:0] sfr_file_input_sel_out = 0,
      input [4:0] mem_str_data_sel_top_in,      //this will only be used
when writing to the frame buffer.//Consumed in memory.
      output reg [4:0] mem_str_data_sel_top_out = 0,
      input [4:0] mem_str_data_sel_bot_in,      //Memory data in multiplexor
selion signals, as well as the comemnt above.//Consumed in memory.
      output reg [4:0] mem_str_data_sel_bot_out = 0,
      input [1:0] reg_file_wen_in,              //Consumed in writeback
      output reg [1:0] reg_file_wen_out = 0,
      input [1:0] sfr_file_wren_in,                   //Consumed in memory.
      output reg [1:0] sfr_file_wren_out = 0,
      input [13:0] call_addr_in,          //This Signal goes directly to the
call stack. //Consumed in memory.
      output reg [13:0] call_addr_out = 0
);

      always @ (posedge clock)
      begin
            if(nreset == 1'b0)
            begin
                  data_top_out <= 0;
                  data_bot_out <= 0;

                  instruction_out <= 0;

                  mem_wen_out <= 0;

                  main_memory_enable_out <= 0;
                  frame_buffer_enable_out <= 0;
                  call_stack_enable_out <= 0;
                  prog_mem_enable_out <= 0;

                  mem_ptr_ctl_out <= 0;
                  call_stk_addr_sel_out <= 0;
                  stk_addr_sel_out <= 0;

                  mem_wb_data_sel_top_out <= 0;
                  mem_wb_data_sel_bot_out <= 0;
```

```
            mem_str_data_sel_top_out <= 0;
            mem_str_data_sel_bot_out <= 0;

            sfr_file_input_sel_out <= 0;
            sfr_file_wren_out <= 0;

            reg_file_wen_out <= 0;

            call_addr_out <= 0;
        end
        else
        begin
            data_top_out <= data_top_in;
            data_bot_out <= data_bot_in;

            instruction_out <= instruction_in;

            mem_wen_out <= mem_wen_in;

            main_memory_enable_out <= main_memory_enable_in;
            frame_buffer_enable_out <= frame_buffer_enable_in;
            call_stack_enable_out <= call_stack_enable_in;
            prog_mem_enable_out <= prog_mem_enable_in;

            mem_ptr_ctl_out <= mem_ptr_ctl_in;
            call_stk_addr_sel_out <= call_stk_addr_sel_in;
            stk_addr_sel_out <= stk_addr_sel_in;

            mem_wb_data_sel_top_out <= mem_wb_data_sel_top_in;
            mem_wb_data_sel_bot_out <= mem_wb_data_sel_bot_in;

            mem_str_data_sel_top_out <= mem_str_data_sel_top_in;
            mem_str_data_sel_bot_out <= mem_str_data_sel_bot_in;

            sfr_file_input_sel_out <= sfr_file_input_sel_in;
            sfr_file_wren_out <= sfr_file_wren_in;

            reg_file_wen_out <= reg_file_wen_in;

            call_addr_out <= call_addr_in;
        end
    end
```

```
/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("ex_mem.vcd");
  $dumpvars (0, ex_mem);
  #1;
end
`endif
*/
endmodule
```

6.2.7 Memory Stage

Code:

```
/*
Module - Memory Pipeline Stage
Author - Zach Walden
Last Changed - 2/18/22, 3/28/22
Description - Memory Stage of the pipeline.
Parameters -
*/

module memory(
      input clock,
      input nreset,
      //BEGIN interface with EX/MEM pipeline register
      input [7:0] data_in_top,
      input [7:0] data_in_bot,
      input [31:0] instruction,
      input [3:0] mem_wb_data_input_sel_top,
      input [6:0] mem_wb_data_input_sel_bot,
      input [4:0] sfr_file_input_sel,
      input [6:0] mem_ptr_ctl,
      input call_stk_addr_sel,
      input stk_addr_sel,
      input [4:0] mem_str_data_input_sel_top,
      input [4:0] mem_str_data_input_sel_bot,
      input [1:0] sfr_file_wren,
      //BEGIN interface with MEM/WB pipeline register
```

```
      output [7:0] data_out_top,
      output [7:0] data_out_bot,
      input [7:0] mem_wb_top,
      input [7:0] mem_wb_bot,
      input [7:0] mem_wb_tm1_top,
      input [7:0] mem_wb_tm1_bot,
      //BEGIN interface with memory i/o unit
      output [15:0] address,
      output [7:0] call_stack_ptr,
      input [11:0] mem_read_data,         //This is assumed to be sign
extended.
      output [11:0] mem_write_data,
      //BEGIN I/O interface
      input [71:0] sfr_file_in,
      output [111:0] sfr_file_out
);

      wire [7:0] sfr_input;
      //instantiate sfr input mux
      sfr_sel_mux sfr_in_sel(
            //.clock(clock),
            .sel_signals(sfr_file_input_sel),
            .ex_mem_data_bot(data_in_bot),
            .mem_wb_data_top(mem_wb_top),
            .mem_wb_data_bot(mem_wb_bot),
            .mem_wb_tm1_data_top(mem_wb_tm1_top),
            .mem_wb_tm1_data_bot(mem_wb_tm1_bot),
            .sfr_data_input(sfr_input)
      );

      wire [7:0] sfr_output;
      wire [15:0] x_ptr;
      wire [15:0] y_ptr;
      wire [15:0] z_ptr;
      wire [15:0] stack_ptr;
      //instantiate SFR file
      sfr_file sf_reg_file(
            .clock(clock),
            .nreset(nreset),
            .mem_ptr_ctl_signals(mem_ptr_ctl),  //Add to control signal list
            .call_stk_addr_sel(call_stk_addr_sel),
            .stk_addr_sel(stk_addr_sel),
            .wren(sfr_file_wren),
```

```
            .wr_addr(instruction[12:8]),
            .write_data(sfr_input),
            .rd_addr(instruction[17:13]),
            .read_data(sfr_output),
            .stack_ptr(stack_ptr),
            .x_ptr(x_ptr),
            .y_ptr(y_ptr),
            .z_ptr(z_ptr),
            .call_stk_ptr(call_stack_ptr),
            .sfr_file_in(sfr_file_in),
            .sfr_file_out(sfr_file_out)
      );

      //instantiate memeory io data input mux
      mem_str_data_sel_mux mem_data_in_mux(
            //.clock(clock),
            .sel_signal_top(mem_str_data_input_sel_top),
            .sel_signal_bot(mem_str_data_input_sel_bot),
            .ex_mem_data_top(data_in_top),
            .ex_mem_data_bot(data_in_bot),
            .mem_wb_data_top(mem_wb_top),       //these signals must be
connected to the data outputs of the MEM/WB pipeline register.
            .mem_wb_data_bot(mem_wb_bot),
            .mem_wb_tm1_data_top(mem_wb_tm1_top),
            .mem_wb_tm1_data_bot(mem_wb_tm1_bot),
            .mem_data(mem_write_data)
      );

      //instantiate memeory address input mux
      mem_addr_sel_mux mem_addr_in_mux(
            //.clock(clock),
            .sel_signals(instruction[19:18]),
            .x_ptr(x_ptr),
            .y_ptr(y_ptr),
            .z_ptr(z_ptr),
            .stack_ptr(stack_ptr),
            .mem_addr(address)
      );

      //instantiate MEM/WB data input mux
      mem_wb_data_input_mux mem_wb_input_mux(
                  //.clock(clock),
                  .sel_signals_top(mem_wb_data_input_sel_top),
```

```
                .sel_signals_bot(mem_wb_data_input_sel_bot),
                .sfr_data(sfr_output),
                .ex_mem_data_top(data_in_top),
                .ex_mem_data_bot(data_in_bot),
                .ld_res_top(mem_read_data[11:8]),
                .ld_res_bot(mem_read_data[7:0]),
                .mem_wb_top(mem_wb_top),
                .mem_wb_bot(mem_wb_bot),
                .mem_wb_tm1_top(mem_wb_tm1_top),
                .mem_wb_tm1_bot(mem_wb_tm1_bot),
                .mem_data_out_top(data_out_top),
                .mem_data_out_bot(data_out_bot)
            );

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("memory.vcd");
  $dumpvars (0, memory);
  #1;
end
`endif
*/
endmodule
```

6.2.7.1 Special Function Register File

Code:

```
/* DONE
Module - Special Function Register File
Author - Zach Walden
Last Changed - 2/23/21, 4/16/22
Description - 32 x 8 special purpose register file. This module has a
single read and write port. What differentiates this module from the
general purpose register file is that every single register is exposed to
the entire peripheral system for reading at any time.This will be use for
memory addressing, interupt control, and peripheral control. These
functions, besides memory addressing, need not be defined during the design
of the main CPU.

Note -
```

```
*/

module sfr_file(
      input clock,
      input nreset,
      input [6:0] mem_ptr_ctl_signals,    //<6> Z inc, <5> Y inc, <4> X inc,
<3> call stack ptr inc,<2> call stack ptr dec ,<1> stack_ptr inc,<0>
stack_ptr dec
      input call_stk_addr_sel,
      input stk_addr_sel,
      input [1:0] wren,
      input [4:0] wr_addr,
      input [7:0] write_data,
      input [4:0] rd_addr,
      output [7:0] read_data,
      output [15:0] stack_ptr,
      output [15:0] x_ptr,
      output [15:0] y_ptr,
      output [15:0] z_ptr,
      output [7:0] call_stk_ptr,
      input [71:0] sfr_file_in,
      output [111:0] sfr_file_out
);

      integer i;

      reg [7:0] sfr_array [0:31];
      reg [7:0] out_data = 0;

      wire [15:0] x_intermediate;
      assign x_intermediate[15:8] = sfr_array[3];
      assign x_intermediate[7:0] = sfr_array[2];
      wire [15:0] x_inc;

      wire [15:0] y_intermediate;
      assign y_intermediate[15:8] = sfr_array[5];
      assign y_intermediate[7:0] = sfr_array[4];
      wire [15:0] y_inc;

      wire [15:0] z_intermediate;
      assign z_intermediate[15:8] = sfr_array[7];
      assign z_intermediate[7:0] = sfr_array[6];
      wire [15:0] z_inc;
```

```
	wire [15:0] stk_ptr_intermediate;
	assign stk_ptr_intermediate[15:8] = sfr_array[1];
	assign stk_ptr_intermediate[7:0] = sfr_array[0];
	wire [15:0] stk_ptr_dec;
	wire [15:0] stk_ptr_inc;

	wire [7:0] call_stk_inc;
	wire [7:0] call_stk_dec;

	initial
	begin
		for(i=0;i<32;i=i+1)
		begin
			sfr_array[i] <= 0;
		end
	end


	always @ (negedge clock)
	begin
		if(nreset == 1'b0)
		begin
			for(i=0;i<32;i=i+1)
			begin
				sfr_array[i] <= 0;
			end
		end
		else
		begin
			//wren[0] is write enable
			if(wren[0] == 1'b1)
			begin
				sfr_array[wr_addr] <= write_data;
			end
			else
			begin
				if(mem_ptr_ctl_signals == 7'b0000010 && wren[0] ==
1'b0)
				begin
					sfr_array[0] <= stk_ptr_inc[7:0];
					sfr_array[1] <= stk_ptr_inc[15:8];
				end
```

```
                        else if(mem_ptr_ctl_signals == 7'b0000001 &&
wren[0] == 1'b0)
                        begin
                              sfr_array[0] <= stk_ptr_dec[7:0];
                              sfr_array[1] <= stk_ptr_dec[15:8];
                        end
                        else if(mem_ptr_ctl_signals == 7'b0000100 &&
wren[0] == 1'b0)
                        begin
                              sfr_array[9] <= call_stk_dec;
                        end
                        else if(mem_ptr_ctl_signals == 7'b0001000 &&
wren[0] == 1'b0)
                        begin
                              sfr_array[9] <= call_stk_inc;
                        end
                        else if(mem_ptr_ctl_signals == 7'b0010000 &&
wren[0] == 1'b0)
                        begin
                              sfr_array[2] <= x_inc[7:0];
                              sfr_array[3] <= x_inc[15:8];
                        end
                        else if(mem_ptr_ctl_signals == 7'b0100000 &&
wren[0] == 1'b0)
                        begin
                              sfr_array[4] <= y_inc[7:0];
                              sfr_array[5] <= y_inc[15:8];
                        end
                        else if(mem_ptr_ctl_signals == 7'b1000000 &&
wren[0] == 1'b0)
                        begin
                              sfr_array[6] <= z_inc[7:0];
                              sfr_array[7] <= z_inc[15:8];
                        end

                        //read in input values.
                        sfr_array[23] <= sfr_file_in[7:0];  //Input Port A
                        sfr_array[24] <= sfr_file_in[15:8];       //Timer
Byte 0
                        sfr_array[25] <= sfr_file_in[23:16];      //Timer
Byte 1
                        sfr_array[26] <= sfr_file_in[31:24];      //Timer
Byte 2
```

```
                        sfr_array[27] <= sfr_file_in[39:32];      //Timer
Byte 3
                        sfr_array[28] <= sfr_file_in[47:40];      //Timer
Byte 4
                        sfr_array[29] <= sfr_file_in[55:48];      //Timer
Byte 5
                        sfr_array[30] <= sfr_file_in[63:56];      //Timer
Byte 6
                        sfr_array[31] <= sfr_file_in[71:64];      //Timer
Byte 7


                end
                if(wren[1] == 1'b1)
                begin
                        out_data <= sfr_array[rd_addr];
                end
                else
                begin
                        out_data <= 8'h00;
                end
            end
      end

      //Memory Pointers
      assign stack_ptr[7:0] = stk_addr_sel ? stk_ptr_inc : sfr_array[0];
      assign stack_ptr[15:8] = stk_addr_sel ? stk_ptr_inc : sfr_array[1];
      assign x_ptr[7:0] = sfr_array[2];
      assign x_ptr[15:8] = sfr_array[3];
      assign y_ptr[7:0] = sfr_array[4];
      assign y_ptr[15:8] = sfr_array[5];
      assign z_ptr[7:0] = sfr_array[6];
      assign z_ptr[15:8] = sfr_array[7];

      assign x_inc = x_intermediate + 1;
      assign y_inc = y_intermediate + 1;
      assign z_inc = z_intermediate + 1;

      assign stk_ptr_inc = stk_ptr_intermediate + 1;
      assign stk_ptr_dec = stk_ptr_intermediate - 1;

      assign call_stk_inc = sfr_array[9] + 1;
      assign call_stk_dec = sfr_array[9] - 1;
```

```
      assign call_stk_ptr = call_stk_addr_sel ? call_stk_dec :
sfr_array[9];

      assign sfr_file_out[7:0] = sfr_array[10];       //External LED
Register
      //Interrupt Conntroller Control Register <0> Int Enable Flag
      assign sfr_file_out[15:8] = sfr_array[11];
      //GICR <2> Timer Compare Match Interrupt Mask, <1>
illegal_opcode_exception mask, <0> vblank_int mask
      assign sfr_file_out[23:16] = sfr_array[12];
      //Timer Compare Match Value Registers.
      assign sfr_file_out[31:24] = sfr_array[13];     //Timer Compare Byte 0
      assign sfr_file_out[40:32] = sfr_array[14];     //Timer Compare Byte 1
      assign sfr_file_out[48:41] = sfr_array[15];     //Timer Compare Byte 2
      assign sfr_file_out[56:49] = sfr_array[16];     //Timer Compare Byte 3
      assign sfr_file_out[63:57] = sfr_array[17];     //Timer Compare Byte 4
      assign sfr_file_out[71:64] = sfr_array[18];     //Timer Compare Byte 5
      assign sfr_file_out[79:72] = sfr_array[19];     //Timer Compare Byte 6
      assign sfr_file_out[87:80] = sfr_array[20];     //Timer Compare Byte 7
      //General Purpose I/O Port B
      assign sfr_file_out[95:88] = sfr_array[21];
      //General Purpuse I/O Port A
      assign sfr_file_out[103:96] = sfr_array[22];    //output
      assign sfr_file_out[111:104] = sfr_array[8];    //Timer Control
Register

      assign read_data = out_data;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("sfr_file.vcd");
  $dumpvars (0, sfr_file);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_sfr_file(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.nreset.value = 0
    await FallingEdge(dut.clock)
    dut.nreset.value = 1
    await FallingEdge(dut.clock)

    addr1 = 0

    dut.sfr_file_in.value = 0

    dut.wren.value = 2

    #loop through each register and write too it.
    for i in range(32):
        dut.rd_addr.value = addr1 & 0x1F
        await FallingEdge(dut.clock)
        assert dut.read_data.value == 0x0, f"reset failed"
        addr1 = addr1 + 1

    #NOTE It should be of note that, the way things are designed, there is
a 2 cycle latency between when a value appears on an input line and when it
can properly be read out into the processing pipeline. This is acceptable
behaviuor as I/O will be a much slower series of events in comparison to
CPU execution.

    #Test Input Functionality
    dut.sfr_file_in.value = 0x0000000FF
    await FallingEdge(dut.clock)
    dut.rd_addr.value = 28
    await FallingEdge(dut.clock)
    #await FallingEdge(dut.clock)
    #assert dut.read_data.value == 0x0FF, f"Input failed"

    dut.sfr_file_in.value = 0x00000FF00
```

```
    await FallingEdge(dut.clock)
    dut.rd_addr.value = 29
    await FallingEdge(dut.clock)
    #await FallingEdge(dut.clock)
    #assert dut.read_data.value == 0x0FF, f"Input failed"

    dut.sfr_file_in.value = 0x000FF0000
    await FallingEdge(dut.clock)
    dut.rd_addr.value = 30
    await FallingEdge(dut.clock)
    #await FallingEdge(dut.clock)
    #assert dut.read_data.value == 0x0FF, f"Input failed"

    dut.sfr_file_in.value = 0x0FF000000
    await FallingEdge(dut.clock)
    dut.rd_addr.value = 31
    await FallingEdge(dut.clock)
    #await FallingEdge(dut.clock)
    #assert dut.read_data.value == 0x0FF, f"Input failed"

    await FallingEdge(dut.clock)
    #await RisingEdge(dut.clock)

    #Test Writing in values.
    #This is a two cycle process.
    #Addresses 0-27

    for addr1 in range(28):
        dut.wren.value = 1
        dut.rd_addr.value = addr1
        dut.wr_addr.value = addr1
        dut.write_data.value = 0x0AA
        await FallingEdge(dut.clock)
        dut.wren.value = 2
        await FallingEdge(dut.clock)
        #await RisingEdge(dut.clock)
        #assert dut.read_data.value == 0x0AA, f"Write, and then read
failed."

    await FallingEdge(dut.clock)
    dut.wren.value = 0

    dut.mem_ptr_ctl_signals.value = 1
```

```
    await FallingEdge(dut.clock)
    #await RisingEdge(dut.clock)
    #assert dut.stack_ptr.value == 0x0AAAB

    dut.mem_ptr_ctl_signals.value = 2
    await FallingEdge(dut.clock)
    #await RisingEdge(dut.clock)
    #assert dut.stack_ptr.value == 0x0AAAA

    dut.mem_ptr_ctl_signals.value = 4
    await FallingEdge(dut.clock)
    dut.mem_ptr_ctl_signals.value = 0
    await FallingEdge(dut.clock)
    #assert dut.call_stk_ptr.value == 0x0AB

    dut.mem_ptr_ctl_signals.value = 8
    await FallingEdge(dut.clock)
    dut.mem_ptr_ctl_signals.value = 0
    await FallingEdge(dut.clock)
    #assert dut.call_stk_ptr.value == 0x0AA
```

6.2.7.2 Special Function Register File Input Selection Mux

Code:

```
/* //DONE
Module - Special Function Register File Input Selection Multiplexer
Author - Zach Walden
Last Changed - 2/23/22
Description - This multi
Parameters -
*/

module sfr_sel_mux(
      //input clock,
      input [4:0] sel_signals,
      input [7:0] ex_mem_data_bot,
      input [7:0] mem_wb_data_top,
      input [7:0] mem_wb_data_bot,
      input [7:0] mem_wb_tm1_data_top,
      input [7:0] mem_wb_tm1_data_bot,
      output reg [7:0] sfr_data_input
);
```

```
	always @ (*)
	begin
		if(sel_signals == 5'b00001)
		begin
			sfr_data_input <= ex_mem_data_bot;
		end
		else if(sel_signals == 5'b00010)
		begin
			sfr_data_input <= mem_wb_data_top;
		end
		else if(sel_signals == 5'b00100)
		begin
			sfr_data_input <= mem_wb_data_bot;
		end
		else if(sel_signals == 5'b01000)
		begin
			sfr_data_input <= mem_wb_tm1_data_top;
		end
		else if(sel_signals == 5'b10000)
		begin
			sfr_data_input <= mem_wb_tm1_data_bot;
		end
		else
		begin
			sfr_data_input <= 8'h00;
		end
	end

	//sel_signals[0] -> sfr_data_input ex_mem_data_bot, sel_signals[1] ->
sfr_data_input = mem_wb_data_top, sel_signals[2] -> sfr_data_input =
mem_wb_data_bot

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("sfr_sel_mux.vcd");
  $dumpvars (0, sfr_sel_mux);
  #1;
end
`endif
*/
```

```
endmodule
```

6.2.7.3 Memory Store Data Selection Mux

Code:

```
/* NEEDS WORK
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/

module mem_str_data_sel_mux(
      //input clock,
      input [4:0] sel_signal_top,
      input [4:0] sel_signal_bot,
      input [7:0] ex_mem_data_top,
      input [7:0] ex_mem_data_bot,
      input [7:0] mem_wb_data_top,
      input [7:0] mem_wb_data_bot,
      input [7:0] mem_wb_tm1_data_top,
      input [7:0] mem_wb_tm1_data_bot,
      output reg [11:0] mem_data
);

      always @ (*)
      begin
            if(sel_signal_top == 5'b00001)
            begin
                  mem_data[11:8] <= ex_mem_data_top[3:0];
            end
            else if(sel_signal_top == 5'b00010)
            begin
                  mem_data[11:8] <= mem_wb_data_top[3:0];
            end
            else if(sel_signal_top == 5'b00100)
            begin
                  mem_data[11:8] <= mem_wb_data_bot[3:0];
            end
            else if(sel_signal_top == 5'b01000)
            begin
```

```
                mem_data[11:8] <= mem_wb_tm1_data_top[3:0];
            end
            else if(sel_signal_top == 5'b10000)
            begin
                mem_data[11:8] <= mem_wb_tm1_data_bot[3:0];
            end
            else
            begin
                mem_data[11:8] <= 0;
            end
        end

        always @ (*)
        begin
            if(sel_signal_bot == 5'b00001)
            begin
                mem_data[7:0] <= ex_mem_data_bot;
            end
            else if(sel_signal_bot == 5'b00010)
            begin
                mem_data[7:0] <= mem_wb_data_top;
            end
            else if(sel_signal_bot == 5'b00100)
            begin
                mem_data[7:0] <= mem_wb_data_bot;
            end
            else if(sel_signal_bot == 5'b01000)
            begin
                mem_data[7:0] <= mem_wb_tm1_data_top;
            end
            else if(sel_signal_bot == 5'b10000)
            begin
                mem_data[7:0] <= mem_wb_tm1_data_bot;
            end
            else
            begin
                mem_data[7:0] <= 0;
            end
        end

        //sel_signal_bot[0] -> mem_data[7:0] = ex_mem_data_bot,
sel_signal_bot[1] -> mem_data[7:0] -> mem_wb_data_top, sel_signal_bot[2] ->
mem_wb_data_bot, sel_signal_bot[3] -> mem_data[11:8] ->
```

```
mem_wb_data_top[3:0], sel_signal_bot[4] -> mem_data[11:8] =
mem_wb_data_bot[3:0]
      //sel_signal_top[0] -> mem_data[11:8] = ex_mem_data_top[3:0],
sel_signal_top[1] -> mem_data[11:8] -> mem_wb_data_top[3:0],
sel_signal_top[2] -> mem_data[11:8] = mem_wb_data_bot[3:0],
sel_signal_top[3] -> mem_data[11:8] -> mem_wb_tm1_data_top[3:0],
sel_signal_top[4] -> mem_data[11:8] = mem_wb__tm1data_bot[3:0]

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("mem_str_data_sel_mux.vcd");
  $dumpvars (0, mem_str_data_sel_mux);
  #1;
end
`endif
*/
endmodule
```

6.2.7.4 Memory Address Selection Mux

Code:

```
/*
Module -
Author - Zach Walden
Last Changed -
Description -
Parameters -
*/

module mem_addr_sel_mux(
      //input clock,
      input [1:0] sel_signals,
      input [15:0] x_ptr,
      input [15:0] y_ptr,
      input [15:0] z_ptr,
      input [15:0] stack_ptr,
      output reg [15:0] mem_addr
);

      always @(*)
```

```
      begin
            if(sel_signals == 2'b00)
            begin
                  mem_addr <= stack_ptr;
            end
            else if(sel_signals == 2'b01)
            begin
                  mem_addr <= x_ptr;
            end
            else if(sel_signals == 2'b10)
            begin
                  mem_addr <= y_ptr;
            end
            else
            begin
                  mem_addr <= z_ptr;
            end
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("mem_addr_sel_mux.vcd");
  $dumpvars (0, mem_addr_sel_mux);
  #1;
end
`endif
*/
endmodule
```

6.2.7.5 MEM/WB Data Input Selection Mux

Code:

```
/* DONE
Module - MEM/WB Data Input Multiplexor
Author - Zach Walden
Last Changed - 2/18/22, 3/16/22, 3/24/22
Description - This module multiplexes the data values in the EX/MEM
register and the LD results into the data inputs of the MEM/WB register
Parameters -
*/
```

```
module mem_wb_data_input_mux(
      //input clock,
      input [3:0] sel_signals_top,
      input [6:0] sel_signals_bot,
      input [7:0] sfr_data,               //SFR data out will always go to
MEM/WB data top is it is selected.
      input [7:0] ex_mem_data_top,
      input [7:0] ex_mem_data_bot,
      input [3:0] ld_res_top,
      input [7:0] ld_res_bot,
      input [7:0] mem_wb_top,
      input [7:0] mem_wb_bot,
      input [7:0] mem_wb_tm1_top,
      input [7:0] mem_wb_tm1_bot,
      output reg [7:0] mem_data_out_top,
      output reg [7:0] mem_data_out_bot
);

      always @ (*)
      begin
            if(sel_signals_top == 4'b0001)
            begin
                  mem_data_out_top <= ex_mem_data_top;
            end
            else if(sel_signals_top == 4'b0010)
            begin
                  mem_data_out_top[3:0] <= ld_res_top;
                  mem_data_out_top[7:4] <= 4'h0;
            end
            else if(sel_signals_top == 4'b0100)
            begin
                  mem_data_out_top <= mem_wb_tm1_top;
            end
            else if(sel_signals_top == 4'b1000)
            begin
                  mem_data_out_top <= mem_wb_tm1_bot;
            end
            else
            begin
                  mem_data_out_top <= 8'h00;
            end
      end
```

```
      always @ (*)
      begin
            if(sel_signals_bot == 7'b0000001)
            begin
                  mem_data_out_bot <= sfr_data;
            end
            else if(sel_signals_bot == 7'b0000010)
            begin
                  mem_data_out_bot <= ex_mem_data_bot;
            end
            else if(sel_signals_bot == 7'b0000100)
            begin
                  mem_data_out_bot <= ld_res_bot;
            end
            else if(sel_signals_bot == 7'b0001000)
            begin
                  mem_data_out_bot <= mem_wb_top;
            end
            else if(sel_signals_bot == 7'b0010000)
            begin
                  mem_data_out_bot <= mem_wb_bot;
            end
            else if(sel_signals_bot == 7'b0100000)
            begin
                  mem_data_out_bot <= mem_wb_tm1_top;
            end
            else if(sel_signals_bot == 7'b1000000)
            begin
                  mem_data_out_bot <= mem_wb_tm1_bot;
            end
            else
            begin
                  mem_data_out_bot <= 8'h00;
            end
      end

      // <0> data_bot = sfr_data, <1> data_bot = ex/mem bottom , <2>
data_bot = ld_res_bot , <3> data_bot = mem_wb_top , <4> data_bot =
mem_wb_bot, <5> data_bot = mem_wb_tm1_top, <6> data_bot = mem_wb_tm1_bot

/*
// the "macro" to dump signals
```

```
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("ex_mem_data_input_mux.vcd");
  $dumpvars (0, ex_mem_data_input_mux);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_ex_mem_data_input_mux(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    await FallingEdge(dut.clock)

    dut.reg_file_top.value = 0x0AA
    dut.reg_file_bot.value = 0x0AA
    dut.alu_res_top.value = 0x0FF
    dut.alu_res_bot.value = 0x0FF

    dut.sel_signals.value = 0

    await FallingEdge(dut.clock)
    assert dut.ex_data_out_top.value == 0x0AA, f"Muxing Failed on the top
byte"
    assert dut.ex_data_out_bot.value == 0x0AA, f"Muxing Failed on the
bottom byte"

    await FallingEdge(dut.clock)
    dut.sel_signals.value = 3

    await FallingEdge(dut.clock)
    assert dut.ex_data_out_top.value == 0x0FF, f"Muxing Failed on the top
byte"
```

```
    assert dut.ex_data_out_bot.value == 0x0FF, f"Muxing Failed on the
bottom byte"
```

6.2.8 MEM/WB Register

Code:

```
/*
Module - Execution/Memory Pipeline Register
Author - Zach Walden
Last Changed - 2/12/22, 3/27/22
Description - This register holds the necessary data to ensure that the
correct results exit the memory pipeline stage.
*/

module mem_wb(
      input clock,                        //System Clock
      input nreset,                       //System Reset Signal
      input [7:0] data_top_in,            //I/O for the top register file
operand read.
      output reg [7:0] data_top_out,
      input [7:0] data_bot_in,            //I/O for the bottom register file
operand read.
      output reg [7:0] data_bot_out,
      output reg [7:0] data_tm1_top,
      output reg [7:0] data_tm1_bot,
      input [31:0] instruction_in,        //I/O for the instruction word.
      output reg [31:0] instruction_out,
      input [1:0] reg_file_wen_in,              //To the register file write
port.
      output reg [1:0] reg_file_wen_out,
      input [13:0] ret_addr_in,                 //Comes from the memory i/o
buffer
      output reg [13:0] ret_addr_out
);

      always @ (posedge clock)
      begin
            if(nreset == 1'b0)
            begin

                  data_top_out <= 0;
```

```
                  data_bot_out <= 0;

                  instruction_out <= 0;

                  reg_file_wen_out <= 0;

                  ret_addr_out <= 0;

                  data_tm1_top <= 0;
                  data_tm1_bot <= 0;
            end
            else
            begin
                  data_top_out <= data_top_in;
                  data_bot_out <= data_bot_in;

                  instruction_out <= instruction_in;

                  reg_file_wen_out <= reg_file_wen_in;

                  ret_addr_out <= ret_addr_in;

                  data_tm1_top <= data_top_out;
                  data_tm1_bot <= data_bot_out;
            end
      end



/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("mem_wb.vcd");
  $dumpvars (0, mem_wb);
  #1;
end
`endif
*/
endmodule
```

6.2.9 Instruction Word Selection Mux

Code:

```
/*
Module - Instruction Word Sel Mux
Author - Zach Walden
Last Changed - 3/31/22
Description - This mux allows the hazard_control_unit to insert either a
nop or call instruction into the pipeline when stalling the instruction
fetch stage.
Parameters -
*/

module inst_word_sel_mux(
      input sel,
      input [31:0] mem_inst_word,
      input [31:0] hazard_unit_inst_word,
      output reg [31:0] inst_word_out
);

      always @ (*)
      begin
            if(sel == 1'b1)
            begin
                  inst_word_out <= hazard_unit_inst_word;
            end
            else
            begin
                  inst_word_out <= mem_inst_word;
            end
      end

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("inst_word_sel_mux.vcd");
  $dumpvars (0, inst_word_sel_mux);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_inst_word_sel_mux(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.mem_inst_word.value = 0xFFFFFFFF
    dut.hazard_unit_inst_word.value = 0xAAAAAAAA

    dut.sel.value = 0
    #test taking memory instruction word.
    await FallingEdge(dut.clock)
    assert dut.inst_word_out.value == 0xFFFFFFFF, f"Taking Memory fetched
value failed"

    #test taking hazard unit inst word
    dut.sel.value = 1
    await FallingEdge(dut.clock)
    assert dut.inst_word_out.value == 0xAAAAAAAA, f"Taking Hazard value
failed"
```

6.2.10 Register File

```
Code:
/*
Module - Register File
Author - Zach Walden
Last Changed - 4/16/22
Description - 32 8-Bit Registers 2 reads and 2 writes each cycle
Parameters -
*/

module register_file(
      input clock,
      input nreset,
      input [1:0] wr_en,
```

```
	input [1:0] rd_en,
	input [9:0] wr_addr,
	input [9:0] rd_addr,
	input [15:0] data_in,
	output [15:0] data_out
);
	integer i;

	reg [7:0] reg_file [0:31];

	initial
	begin
		for(i=0;i<32;i=i+1)
		begin
			reg_file[i] = 0;
		end
	end

	always @ (negedge clock)
	begin
		//handle writes
		if(nreset == 1'b0)
		begin
			for(i=0;i<32;i=i+1)
			begin
				reg_file[i] <= 0;
			end
		end
		else
		begin
			if(wr_en[0] == 1'b1)
			begin
				reg_file[wr_addr[4:0]] <= data_in[7:0];
			end
			/*
			else
			begin
				reg_file[wr_addr[4:0]] <= reg_file[wr_addr[4:0]];
			end
			*/
			if(wr_en[1] == 1'b1)
			begin
				reg_file[wr_addr[9:5]] <= data_in[15:8];
```

```
                end
                /*
                else
                begin
                      reg_file[wr_addr[9:5]] <= reg_file[wr_addr[9:5]];
                end
                */
            end
      end

      assign data_out[7:0] = rd_en[0] ? reg_file[rd_addr[4:0]] : 8'h00;
      assign data_out[15:8] = rd_en[1] ? reg_file[rd_addr[9:5]] : 8'h00;

/*
// the "macro" to dump signals
`ifdef COCOTB_SIM
initial begin
  $dumpfile ("register_file.vcd");
  $dumpvars (0, register_file);
  #1;
end
`endif
*/
endmodule
```

Test Bench:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_register_file(dut):
    clock = Clock(dut.clock, 10, units="ns")
    cocotb.start_soon(clock.start())

    dut.nreset.value = 1

    j = 0xFFFF
    addr1 = 0
    addr2 = 31
    await RisingEdge(dut.clock)
```

```
    dut.nreset.value = 1
    dut.wr_en.value = 1
    dut.rd_en.value = 1
    dut.wr_addr.value = 0x000
    dut.rd_addr.value = 0x000

    #write 0-32 to all registers.
    #This test validates write on the positive edge & read on the negative
edge.:w

    for i in range(32):
        dut.data_in.value = (addr1 + 1) & 0x0FF
        k = (addr1 ) & 0x01F
        dut.wr_addr.value = (k & 0x1F)
        dut.rd_addr.value = (k & 0x1F)
        await RisingEdge(dut.clock)
        print("data_out = " + str(dut.data_out.value.integer) + " data_in =
" + str(dut.data_in.value.integer))
        assert dut.data_out.value == (addr1 + 1) & 0x0FF, f"write failed
data_out = {dut.data_out.value.integer}, data_in =
{dut.data_in.value.integer}"
        addr1 = (addr1 + 1) & 0x1F

    addr1 = 0
    #reset the register file. This should zero out every register
    dut.wr_en.value = 0
    dut.nreset.value = 0
    await FallingEdge(dut.clock)
    dut.nreset.value = 1
    await RisingEdge(dut.clock)
    #loop through every register to ensure that a full reset occured.
    for i in range(32):
        k = (addr1) & 0x1F
        dut.rd_addr.value = k
        await RisingEdge(dut.clock)
        assert dut.data_out.value == 0, f"reset failed"
        addr1 = (addr1 + 1) & 0x1F

    write_list = [0xEE, 0xFF, 0xDD, 0xCC, 0xBB, 0xAA, 0x99, 0x88]

    #flash reset again
    dut.nreset.value = 0
```

```
    await RisingEdge(dut.clock)
    dut.nreset.value = 1

    await RisingEdge(dut.clock)

    addr1 = 0
    j = 0
    dut.wr_en.value = j

    for i in range(4):
        k = (addr1 & 0x1F | ((addr1 + 1)) & 0x3E0) & 0x3FF
        dut.rd_addr.value = k
        dut.wr_addr.value = k
        l = ((write_list[addr1] & 0x0FF)|((write_list[addr1 + 1] << 8) &
0xFF00)) & 0x0FFF
        dut.data_in.value = l
        await RisingEdge(dut.clock)
        check_different_rd_wrs(j, write_list, l, addr1, dut)
        addr1 = (addr1 + 2) & 0x1F
        j = (j + 1) & 0x3
        dut.wr_en.value = j
        dut.rd_en.value = j



def check_different_rd_wrs(j, write_list, l, addr1, dut):
    if(j == 0):
        f = 0
        assert dut.data_out.value == f, f"No write failed data_out =
{dut.data_out.value}, f = {f}"
    elif(j == 1):
        f = write_list[addr1] & 0x0FF
        assert dut.data_out.value == f, f"Low write & read failed data_out
= {dut.data_out.value.integer}, f = {f}"
    elif(j == 2):
        f = (write_list[addr1 + 1] << 8) & 0x0FF00
        assert dut.data_out.value == f, f"High write & read failed data_out
= {dut.data_out.value.integer}, f = {f}"
    else:
        assert dut.data_out.value == l, f"Double write and read failed
data_out = {dut.data_out.value.integer}, l = {l}"
```

7. Assembler

7.1 zwriscassemble

Code:

```
#!/usr/bin/env python3

import sys

class Assembler():

    #Bitwose And Constants
    REG_LOW = 0x00001F00
    REG_HIGH = 0x0003E000
    MEM_PTR_ADDR = 0x000C0000
    POST_INC = 0x00400000
    IMMEADIATE = 0xFF000000
    ADDRESS = 0xFFFC0000
    MAKE_32 = 0xFFFFFFFF

    #Default Instruction Words.
    instructions = [    ("nop", 0x00000000),
                        ("inc", 0x012C00BC),
                        ("dec", 0xFF2C00BC),
                        ("com", 0x013400BC),
                        ("inv", 0x003400BC),
                        ("addi",0x002C00BC),
                        ("add", 0x00240080),
                        ("sub", 0x00200080),
                        ("subi",0x002800BC),
                        ("cp",  0x00000080),
                        ("cpi", 0x000800BC),
                        ("mul", 0x0020008E),
                        ("muli",0x0028009E),
                        ("and", 0x00240097),
                        ("andi",0x002C009B),
                        ("or",  0x00200097),
                        ("ori", 0x0028009B),
                        ("shr", 0x002400A5),
                        ("shl", 0x002000A5),
                        ("ld",  0x003000FB),
                        ("ldfb",0x002000FB),
```

```
                        ("pop", 0x003000FB),
                        ("ldi", 0x002800F8),
                        ("lpm", 0x002000F9),
                        ("str", 0x001000C6),
                        ("stfb",0x000000C6),
                        ("push",0x001000C6),
                        ("movr",0x0020009C),
                        ("mov", 0x00000000),
                        ("in",  0x0028009C),
                        ("out", 0x0024009C),
                        ("jmp",0x00000438),
                        ("brcs",0x00000538),
                        ("brcc",0x00000138),
                        ("breq",0x00000638),
                        ("brne",0x00000238),
                        ("brng",0x00000738),
                        ("brps",0x00000338),
                        ("call",0x00000042),
                        ("ret", 0x00000043),
                        ("reti",0x00100043),
                        ("hlt", 0x0000001F)]

    #Keywords
    gprs = [("r0", 0x00), ("r1", 0x01), ("r2", 0x02), ("r3", 0x03), ("r4",
0x04), ("r5", 0x05), ("r6", 0x06), ("r7", 0x07), ("r8", 0x08), ("r9",
0x09),("r10", 0x0A), ("r11", 0x0B), ("r12", 0x0C), ("r13", 0x0D), ("r14",
0x0E), ("r15", 0x0F), ("r16", 0x10), ("r17", 0x11), ("r18", 0x12), ("r19",
0x13), ("r20", 0x14), ("r21", 0x15), ("r22", 0x16), ("r23", 0x17), ("r24",
0x18), ("r25", 0x19), ("r26", 0x1A), ("r27", 0x1B), ("r28", 0x1C), ("r29",
0x1D), ("r30", 0x1E), ("r31", 0x1F)]

    sfrs = [("sph", 0x01), ("spl", 0x00), ("xh", 0x03), ("xl", 0x02),
("yh", 0x5), ("yl", 0x04), ("zh", 0x07), ("zl", 0x06), ("t1cr", 0x08),
("csp", 0x09), ("led", 0x0A), ("iccr", 0x0B), ("gicr", 0x0C), ("tcb0",
0x0D), ("tcb1", 0x0E), ("tcb2", 0x0F), ("tcb3", 0x10), ("tcb4", 0x11),
("tcb5", 0x12), ("tcb6", 0x13), ("tcb7", 0x14), ("pbout", 0x15), ("paout",
0x16), ("pain", 0x17), ("tb0", 0x18), ("tb1", 0x19), ("tb2", 0x1A), ("tb3",
0x1B), ("tb4", 0x1C), ("tb5", 0x1D), ("tb6", 0x1E), ("tb7", 0x1F)]
    ptrOptions = [("x", 0x01, 0x0), ("x+", 0x01, 0x1), ("y", 0x02, 0x0),
("y+", 0x02, 0x1), ("z", 0x03, 0x0), ("z+", 0x03, 0x1)]
    directives = [(".org", 0x1), (".inc", 0x1), (".equ", 0x1), (".dw",
0x1)]
```

```
    def __init__(self, asmLines, binaryFile):
        self.asmLines = asmLines
        self.binaryFile = binaryFile
        self.labels = []
        self.address = 0x0

    def loop(self):
        self.address = 0x0
        for line in self.asmLines:
            #Is line a comment?
            test = line.strip()
            if(len(test) != 0 and test[0] == ';'):
                pass
            #is line a label?
            elif(line[0] == ':'):
                #give label a memory address
                splitStr = line.split(':')
                splitStr = splitStr[1].split(';')
                splitStr = splitStr[0].split(' ')
                self.labels.append((splitStr[0], self.address))
            elif(test[0] == '.'):
                #If a .org change address to .org.
                pass
            elif(line == '\n'):
                pass
            else:
                inst = test.split(' ')
                if inst[0] == "MOV":
                    self.address += 0x2
                else:
                    self.address += 0x1
        print(self.labels)
        #Loop through to assemble
        self.address = 0x0
        #Re Loop through the lines to assemble the instructions.
        self.count = 1
        for line in self.asmLines:
            tmp = line.strip().split(' ')[0].lower()
            cont = True
            #is line a comment or blank?
            test = line.strip()
            if((len(test) != 0 and test[0] == ';') or line[0] == '\n'):
```

```
                cont = False
            #Is line an assembler directive?
            inList, item = self.isInList(tmp, self.directives)
            if(inList and cont):
                cont = False
                if item[0] == ".org":
                    self.address = int(line.split(' ')[1], 16)
                elif item[0] == ".inc":
                    pass
                elif item[0] == ".equ":
                    pass
                elif item[0] == ".dw":
                    pass
            #Is line a Label?
            if(cont and tmp[0] == ':'):
                cont = False
            #Line Must be an instruction
            inList, item = self.isInList(tmp, self.instructions)
            if(inList and cont):
                cont = False
                self.writeInstruction(item, line.split(' '),
self.binaryFile)
            self.count += 1

    def writeInstruction(self, item, line, binaryFile):
        instructionWord = 0x0
        #Instruction Switch
        if item[0] == "nop":
            #All zeros, already done
            instructionWord = item[1]
        elif item[0] == "inc":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW)
        elif item[0] == "dec":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
```

```
            regOperand = regOperand[1]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW)
        elif item[0] == "add":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "addi":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            immeadiate = int(line[2], 16)
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (immeadiate << 24 & self.IMMEADIATE)
        elif item[0] == "sub":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "subi":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            immeadiate = int(line[2], 16)
```

```
            immeadiate &= 0x0FF
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (immeadiate << 24 & self.IMMEADIATE)
        elif item[0] == "cp":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "cpi":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            immeadiate = int(line[2], 16)
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (immeadiate << 24 & self.IMMEADIATE)
        elif item[0] == "mul":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            assert regOperand2 != regOperand1, ("You cannot load both
halves of a frambebuffer value into the same register")
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "muli":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
```

```
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            assert regOperand2 != regOperand1, ("You cannot load both
halves of a frambebuffer value into the same register")
            immeadiate = int(line[3], 16)
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH) | (immeadiate << 24 &
self.IMMEADIATE)
        elif item[0] == "and":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "andi":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            immeadiate = int(line[2], 16)
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (immeadiate << 24 & self.IMMEADIATE)
        elif item[0] == "or":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            instructionWord = instructionWord | (regOperand1 << 8 &
```

```
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "ori":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            immeadiate = int(line[2], 16)
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (immeadiate << 24 & self.IMMEADIATE)
        elif item[0] == "shr":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW)
        elif item[0] == "shl":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW)
        elif item[0] == "com":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW)
        elif item[0] == "inv":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW)
        elif item[0] == "ld":
```

```
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            inList, item = self.isInList(line[2].lower(), self.ptrOptions)
            assert inList, (line[2] + " Is not a valid pointer option,
Line: " + str(self.count))
            ptrAddr = item[1]
            postInc = item[2]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (ptrAddr << 18 & self.MEM_PTR_ADDR) | (postInc << 22 &
self.POST_INC)
        elif item[0] == "ldfb":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            assert regOperand2 != regOperand1, ("You cannot load both
halves of a frambebuffer value into the same register")
            inList, item = self.isInList(line[3].lower(), self.ptrOptions)
            assert inList, (line[3] + " Is not a valid pointer option,
Line: " + str(self.count))
            ptrAddr = item[1]
            postInc = item[2]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH) | (ptrAddr << 18 &
self.MEM_PTR_ADDR) | (postInc << 22 & self.POST_INC)
        elif item[0] == "ldi":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            immeadiate = int(line[2], 16)
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (immeadiate << 24 & self.IMMEADIATE)
        elif item[0] == "str":
```

```
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            inList, item = self.isInList(line[1].lower(), self.ptrOptions)
            assert inList, (line[1] + " Is not a valid pointer option,
Line: " + str(self.count))
            ptrAddr = item[1]
            postInc = item[2]
            instructionWord = instructionWord | (regOperand << 13 &
self.REG_HIGH) | (ptrAddr << 18 & self.MEM_PTR_ADDR) | (postInc << 22 &
self.POST_INC)
        elif item[0] == "stfb":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[3].lower(), self.gprs)
            assert inList, (line[3] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            inList, item = self.isInList(line[1].lower(), self.ptrOptions)
            assert inList, (line[1] + " Is not a valid pointer option,
Line: " + str(self.count))
            ptrAddr = item[1]
            postInc = item[2]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH) | (ptrAddr << 18 &
self.MEM_PTR_ADDR) | (postInc << 22 & self.POST_INC)
        elif item[0] == "mov":
            #This instruction will overwrite R0
            inList, item = self.isInList("ld", self.instructions)
            instructionWord = item[1]
            inList, item = self.isInList(line[2].lower(), self.ptrOptions)
            assert inList, (line[2] + " Is not a valid pointer option,
Line: " + str(self.count))
            ptrAddr = item[1]
            postInc = item[2]
            instructionWord = instructionWord | (ptrAddr << 18 &
self.MEM_PTR_ADDR) | (postInc << 22 & self.POST_INC)
            #Write The First instruction
```

```
            self.writeInstructionWord(instructionWord & self.MAKE_32)
            #Prepare the Store
            inList, item = self.isInList(line[1].lower(), self.ptrOptions)
            assert inList, (line[1] + " Is not a valid pointer option,
Line: " + str(self.count))
            ptrAddr = item[1]
            postInc = item[2]
            inList, item = self.isInList("str", self.instructions)
            instructionWord = item[1]
            instructionWord = instructionWord | (ptrAddr << 18 &
self.MEM_PTR_ADDR) | (postInc << 22 & self.POST_INC)
        elif item[0] == "movr":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "in":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.sfrs)
            assert inList, (line[2] + " is not a valid register, Line: " +
str(self.count))
            regOperand2 = regOperand[1]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "out":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.sfrs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand1 = regOperand[1]
            inList, regOperand = self.isInList(line[2].lower(), self.gprs)
            assert inList, (line[2] + " is not a valid register, Line: " +
```

```
str(self.count))
            regOperand2 = regOperand[1]
            instructionWord = instructionWord | (regOperand1 << 8 &
self.REG_LOW) | (regOperand2 << 13 & self.REG_HIGH)
        elif item[0] == "jmp":
            instructionWord = item[1]
            inList, item = self.isInList(line[1], self.labels)
            assert inList, (line[1] + " Is not a valid label, Line: " +
str(self.count))
            address = item[1]
            instructionWord = instructionWord | (address << 18 &
self.ADDRESS)
        elif item[0] == "brcs":
            instructionWord = item[1]
            inList, item = self.isInList(line[1], self.labels)
            assert inList, (line[1] + " Is not a valid label, Line: " +
str(self.count))
            address = item[1]
            instructionWord = instructionWord | (address << 18 &
self.ADDRESS)
        elif item[0] == "brcc":
            instructionWord = item[1]
            inList, item = self.isInList(line[1], self.labels)
            assert inList, (line[1] + " Is not a valid label, Line: " +
str(self.count))
            address = item[1]
            instructionWord = instructionWord | (address << 18 &
self.ADDRESS)
        elif item[0] == "breq":
            instructionWord = item[1]
            inList, item = self.isInList(line[1], self.labels)
            assert inList, (line[1] + " Is not a valid label, Line: " +
str(self.count))
            address = item[1]
            instructionWord = instructionWord | (address << 18 &
self.ADDRESS)
        elif item[0] == "brne":
            instructionWord = item[1]
            inList, item = self.isInList(line[1], self.labels)
            assert inList, (line[1] + " Is not a valid label, Line: " +
str(self.count))
            address = item[1]
            instructionWord = instructionWord | (address << 18 &
```

```
self.ADDRESS)
        elif item[0] == "brng":
            instructionWord = item[1]
            inList, item = self.isInList(line[1], self.labels)
            assert inList, (line[1] + " Is not a valid label, Line: " +
str(self.count))
            address = item[1]
            instructionWord = instructionWord | (address << 18 &
self.ADDRESS)
        elif item[0] == "brps":
            instructionWord = item[1]
            inList, item = self.isInList(line[1], self.labels)
            assert inList, (line[1] + " Is not a valid label, Line: " +
str(self.count))
            address = item[1]
            instructionWord = instructionWord | (address << 18 &
self.ADDRESS)
        elif item[0] == "call":
            instructionWord = item[1]
            inList, item = self.isInList(line[1], self.labels)
            assert inList, (line[1] + " Is not a valid label, Line: " +
str(self.count))
            address = item[1]
            instructionWord = instructionWord | (address << 18 &
self.ADDRESS)
        elif item[0] == "ret":
            instructionWord = item[1]
        elif item[0] == "reti":
            instructionWord = item[1]
        elif item[0] == "push":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW)| (regOperand << 13 & self.REG_HIGH)
        elif item[0] == "pop":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
```

```
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (regOperand << 13 & self.REG_HIGH)
        elif item[0] == "lpm":
            instructionWord = item[1]
            inList, regOperand = self.isInList(line[1].lower(), self.gprs)
            assert inList, (line[1] + " is not a valid register, Line: " +
str(self.count))
            regOperand = regOperand[1]
            inList, item = self.isInList(line[2].lower(), self.ptrOptions)
            assert inList, (line[2] + " Is not a valid pointer option,
Line: " + str(self.count))
            ptrAddr = item[1]
            postInc = item[2]
            instructionWord = instructionWord | (regOperand << 8 &
self.REG_LOW) | (ptrAddr << 18 & self.MEM_PTR_ADDR) | (postInc << 22 &
self.POST_INC)
        elif item[0] == "hlt":
            #Nothing Needs to be done
            instructionWord = item[1]
        else:
            print("Not an Instruction")
        #write the instruction
        self.writeInstructionWord(instructionWord & self.MAKE_32)



    def writeInstructionWord(self, instructionWord):
        #seek to the location in the file
        self.binaryFile.seek(self.address, 0)
        self.binaryFile.write(instructionWord.to_bytes(32,
byteorder='little', signed=False))
        self.address += 0x4

    def isInList(self, key, list):
        key = key.strip()
        for item in list:
            tmp = item[0].strip()
            if key == tmp:
                return (True, item)
        return (None, None)

def main():
    if (len(sys.argv) != 2):
```

```
        print("Please provide the proper .asm file for compilation")

    asmFile = sys.argv[1]
    binaryFile = open(asmFile + ".bin", "wb")
    for i in range(16384):
        binaryFile.write((0x00000000).to_bytes(32, byteorder='little',
signed=False))

    asmFile = open(asmFile, "r")

    asmLines = asmFile.readlines()
    asmFile.close()

    assembler = Assembler(asmLines, binaryFile)
    assembler.loop()

    binaryFile.close()

if __name__ == "__main__":
    main()
```

7.2 bin2coe

Code:

```
#!/usr/bin/env python3

import sys
from hashlib import new
import struct
import cv2
import numpy

def main():
    if (len(sys.argv) != 2):
        print("Please provide the proper .bin file for compilation")

    binFile = open(sys.argv[1], "rb")
    programName = (sys.argv[1].split('.'))[0]
    coeFile = open(programName + ".coe", "w")
```

```
coeFile.write("memory_initialization_radix=16;\nmemory_initialization_vecto
r=\n")

    for i in range(16384):
        word = struct.unpack('<I', binFile.read(4))[0]
        fileString = (hex(word).split('x'))[1]
        if i == 16383:
            coeFile.write(fileString + ";")
        else:
            coeFile.write(fileString + ",\n")

    coeFile.close()
    binFile.close()

if __name__ == "__main__":
    main()
```

8. Appendices

8.1 A: Opcode Table

Instruction Name	Nnemonic	QuickNotes	0xOP	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
				Opcode								dst/bot (alu top) reg					src/top (alu bot) reg					OS	Im.	inv	WR	PI		Immeadiate Data							
				FU op						ma	rf																								
No Operation	NOP	Do Nothing	0x00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Increment	INC	same as ADDI	0xBC	0	0	1	1	1	1	0	1	g	g	g	g	g	x	x	x	x	x	1	1	0	1	x	x	1	0	0	0	0	0	0	0
Decrement	DEC	same as ADDI	0xBC	0	0	1	1	1	1	0	1	g	g	g	g	g	x	x	x	x	x	1	1	0	1	x	x	1	1	1	1	1	1	1	1
Add	ADD		0x80	0	0	0	0	0	0	0	1	g	g	g	g	g	g	g	g	g	g	1	0	0	1	x	x	x	x	x	x	x	x	x	x
Add Immeadiate	ADDI		0xBC	0	0	1	1	1	1	0	1	g	g	g	g	g	x	x	x	x	x	1	1	0	1	x	x	d	d	d	d	d	d	d	d
Subtract	SUB	Same as ADD	0x80	0	0	0	0	0	0	0	1	g	g	g	g	g	g	g	g	g	g	0	0	0	1	x	x	x	x	x	x	x	x	x	x
Subtract Immeadiate	SUBI	Same as ADDI	0xBC	0	0	1	1	1	1	0	1	g	g	g	g	g	x	x	x	x	x	0	1	0	1	x	x	d	d	d	d	d	d	d	d
Compare	CP	Same as ADD	0x80	0	0	0	0	0	0	0	1	g	g	g	g	g	g	g	g	g	g	0	0	0	0	x	x	x	x	x	x	x	x	x	x
Compare Immeadiate	CPI	Same as ADDI	0xBC	0	0	1	1	1	1	0	1	g	g	g	g	g	x	x	x	x	x	0	1	0	0	x	x	d	d	d	d	d	d	d	d
Multiply	MUL		0x8E	0	1	1	1	0	0	0	1	g	g	g	g	g	g	g	g	g	g	0	0	0	1	x	x	x	x	x	x	x	x	x	x
Multiply Immeadiate	MULI		0x9E	0	1	1	1	1	0	0	1	g	g	g	g	g	g	g	g	g	g	0	1	0	1	x	x	d	d	d	d	d	d	d	d
And	AND		0x97	1	1	1	0	1	0	0	1	g	g	g	g	g	g	g	g	g	g	1	0	0	1	x	x	x	x	x	x	x	x	x	x
And Immeadiate	ANDI		0x9B	1	1	0	1	1	0	0	1	g	g	g	g	g	x	x	x	x	x	1	1	0	1	x	x	d	d	d	d	d	d	d	d
Or	OR	Same as AND	0x97	1	1	1	0	1	0	0	1	g	g	g	g	g	g	g	g	g	g	0	0	0	1	x	x	x	x	x	x	x	x	x	x
Or Immeadiate	ORI	Same as ANDI	0x9B	1	1	0	1	1	0	0	1	g	g	g	g	g	x	x	x	x	x	0	1	0	1	x	x	d	d	d	d	d	d	d	d
Right Shift	SHR	src = dst	0xA5	1	0	1	0	0	1	0	1	g	g	g	g	g	g	g	g	g	g	1	0	0	1	x	x	x	x	x	x	x	x	x	x
Left Shift	SHL	src = dst, Same as SHR	0xA5	1	0	1	0	0	1	0	1	g	g	g	g	g	g	g	g	g	g	0	0	0	1	x	x	x	x	x	x	x	x	x	x
Complement	COM	ADDI Pseudo Alias	0xBC	0	0	1	1	1	1	0	1	g	g	g	g	g	x	x	x	x	x	1	0	1	1	x	x	1	0	0	0	0	0	0	0
Invert	INV	alias of COM.	0xBC	0	0	1	1	1	1	0	1	g	g	g	g	g	x	x	x	x	x	1	0	1	1	x	x	0	0	0	0	0	0	0	0
Load	LD	PI set for post increment of mem reg.	0xFB	1	1	0	1	1	1	1	1	g	g	g	g	g	g	g	g	g	g	m	m	1	1	i	x	x	x	x	x	x	x	x	x
Load Framebuffer	LDFB	LD Alias. Inv Selects Mem Interface	0xFB	1	1	0	1	1	1	1	1	g	g	g	g	g	g	g	g	g	g	m	m	0	1	i	x	x	x	x	x	x	x	x	x
Load Immeadiate	LDI		0xF8	0	0	0	1	1	1	1	1	g	g	g	g	g	g	g	g	g	g	x	1	x	1	x	x	d	d	d	d	d	d	d	d
Store	STR	PI set for post increment of mem reg.	0xC6	0	1	1	0	0	0	1	1	g	g	g	g	g	g	g	g	g	g	m	m	1	0	i	x	x	x	x	x	x	x	x	x
Store FrameBuffer	STFB	STR Alias	0xC6	0	1	1	0	0	0	1	1	g	g	g	g	g	g	g	g	g	g	m	m	0	0	i	x	x	x	x	x	x	x	x	x
Move	MOV	LD -> STR Assm. Alias	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	an	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Move Register	MOVR	Pseudo Aliased as IN & OUT	0x9C	0	0	1	1	1	0	0	1	g	g	g	g	g	g	g	g	g	g	0	0	x	1	x	x	x	x	x	x	x	x	x	x
In	IN	LD SFR into GPR	0x9C	0	0	1	1	1	0	0	1	g	g	g	g	g	s	s	s	s	s	0	1	x	1	x	x	x	x	x	x	x	x	x	x
Out	OUT	LD GPR into SFR	0x9C	0	0	1	1	1	0	0	1	s	s	s	s	s	g	g	g	g	g	1	0	x	1	x	x	x	x	x	x	x	x	x	x
Jump	JMP	Low 2 Bits == CC, Bit 2 = set/!clr	0x38	0	0	0	1	1	1	0	0	0	0	1	x	x	x	x	x	x	x	a	a	a	a	a	a	a	a	a	a	a	a	a	a
Branch If Carry Set	BRCS	JMP Pseudo Alias	0x38	0	0	0	1	1	1	0	0	1	0	1	x	x	x	x	x	x	x	a	a	a	a	a	a	a	a	a	a	a	a	a	a
Branch If Carry Clear	BRCC	JMP Pseudo Alias	0x38	0	0	0	1	1	1	0	0	1	0	0	x	x	x	x	x	x	x	a	a	a	a	a	a	a	a	a	a	a	a	a	a
Branch If Equal	BREQ	JMP Pseudo Alias	0x38	0	0	0	1	1	1	0	0	0	1	1	x	x	x	x	x	x	x	a	a	a	a	a	a	a	a	a	a	a	a	a	a
Branch If Not Equal	BRNE	JMP Pseudo Alias	0x38	0	0	0	1	1	1	0	0	0	1	0	x	x	x	x	x	x	x	a	a	a	a	a	a	a	a	a	a	a	a	a	a
Branch If Negative	BRNG	JMP Pseudo Alias	0x38	0	0	0	1	1	1	0	0	1	1	1	x	x	x	x	x	x	x	a	a	a	a	a	a	a	a	a	a	a	a	a	a
Branch If Positive	BRPS	JMP Pseudo Alias	0x38	0	0	0	1	1	1	0	0	1	1	0	x	x	x	x	x	x	x	a	a	a	a	a	a	a	a	a	a	a	a	a	a
Call	CALL		0x42	0	1	0	0	0	0	1	0	x	x	x	x	x	x	x	x	x	x	a	a	a	a	a	a	a	a	a	a	a	a	a	a
Return	RET	INV bit is diff. with RETI	0x43	1	1	0	0	0	0	1	0	x	x	x	x	x	x	x	x	x	x	x	x	0	x	x	x	x	x	x	x	x	x	x	x
Return From Interrupt	RETI		0x43	1	1	0	0	0	0	1	0	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x	x	x	x	x	x	x	x
Push	PUSH	STR Alias	0xC6	0	1	1	0	0	0	1	1	g	g	g	g	g	g	g	g	g	g	0	0	1	0	x	x	x	x	x	x	x	x	x	x
Pop	POP	LD Alias	0xFB	1	1	0	1	1	1	1	1	g	g	g	g	g	g	g	g	g	g	0	0	1	1	x	x	x	x	x	x	x	x	x	x
Load From Program Memory	LPM		0xF9	1	0	0	1	1	1	1	1	g	g	g	g	g	x	x	x	x	x	m	m	x	1	i	x	x	x	x	x	x	x	x	x
Halt	HLT		0x1F	1	1	1	1	1	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Legend:

g = General Purpose Register Address

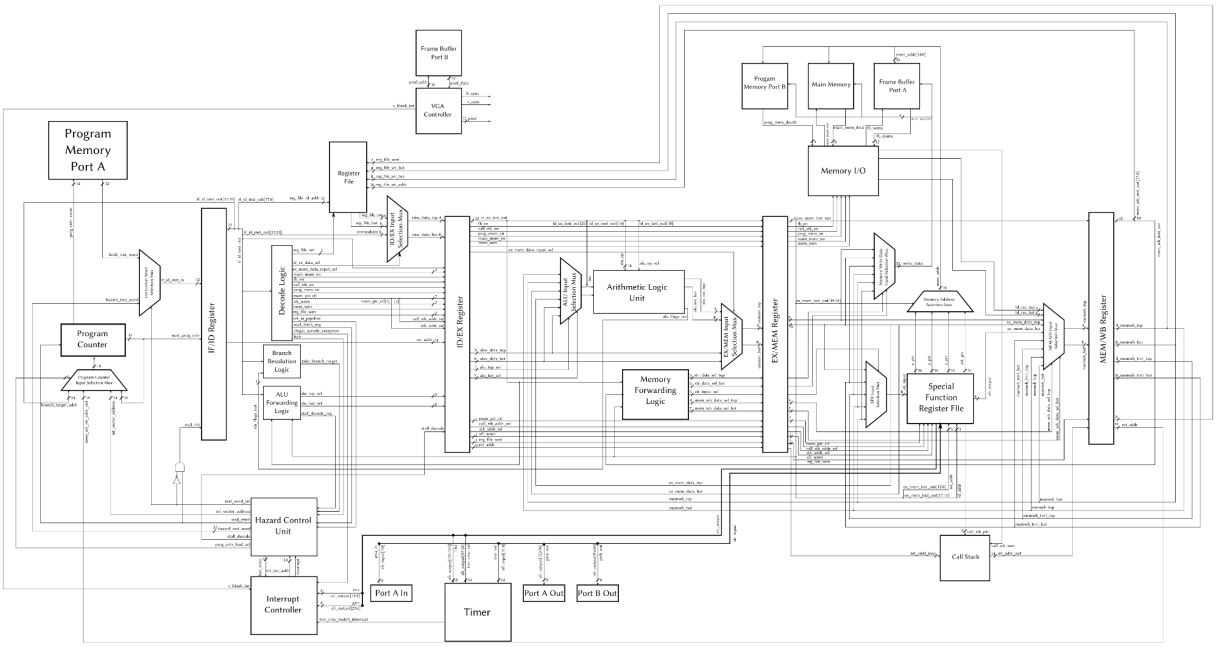
s = Special Function Register Address

d = Immeadiate Data

a = Program Memory Address

m = Memory Pointer Address

8.2 B: Microarchitecture Diagram



The full version is available at my [github](#).

8.3 C: Machine Cycle Diagram

Instruction	Operation	Cycle 0 (Fetch)	Cycle 1 (Decode)	Cycle 2 (ALU)	Cycle 3 (Memory Access)	Cycle 4 (Writeback)
NOP	Nothing	Nothing	Nothing	Nothing	Nothing	Nothing
INC	Increment Register	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode Instruction. Fetch Operand From Register File Set Alu Enable Flag. Set Immeadiate Data Flag. Latch IW, and Above Flags	Perform Operation Encoded In I.W. Forward Result & Dst Reg. Addr. To Decode Stage.	Nothing	Write Result.
DEC	Decrement Register	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode Instruction. Fetch Operand From Register File Set Alu Enable Flag. Set Immeadiate Data Flag. Latch IW, and Above Flags Into Dec/ALU Pipeline Register	Perform Operation Encoded In I.W. Forward Result & Dst Reg. Addr. To Decode Stage.	Nothing	Write Result.
ADD	Add Two Registers	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
ADDI	Add Register & Imm.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode Instruction. Fetch Operand From Register File Set Alu Enable Flag. Set Immeadiate Data Flag. Latch IW, and Above Flags Into Dec/ALU Pipeline Register	Perform Operation Encoded In I.W. Forward Result & Dst Reg. Addr. To Decode Stage.	Nothing	Write Result.
SUB	Subtract Two Registers	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
SUBI	Subtract Reg. & Imm.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode Instruction. Fetch Operand From Register File Set Alu Enable Flag. Set Immeadiate Data Flag. Latch IW, and Above Flags Into Dec/ALU Pipeline Register	Perform Operation Encoded In I.W. Forward Result & Dst Reg. Addr. To Decode Stage.	Nothing	Write Result.
MUL	Multiply Two Registers	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
MULI	Multiply Reg & Imm.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
AND	B.W. And Two Regs.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
ANDI	B.W. And Reg & Imm	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
OR	B.W. Or Two Regs.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
ORI	B.W. Or Reg & Imm	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
SHR	Shift Register Right	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
SHL	Shift Register Left	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode, Check for forward	Mux alu input, execute, forward result	Nothing	Write Result.
COM	Two's Complement Reg	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode Instruction. Fetch Operand From Register File Set Alu Enable Flag. Set Immeadiate Data Flag. Latch IW, and Above Flags Into Dec/ALU Pipeline Register	Invert Operand Bits Perform Operation Encoded In I.W. Forward Result & Dst Reg. Addr. To Decode Stage.	Nothing	Write Result.
INV	Invert Register	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Decode Instruction. Fetch Operand From Register File Set Alu Enable Flag. Set Immeadiate Data Flag. Latch IW, and Above Flags Into Dec/ALU Pipeline Register	Invert Operand Bits Perform Operation Encoded In I.W. Forward Result & Dst Reg. Addr. To Decode Stage.	Nothing	Write Result.
LD	Load Reg from Mem.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Read Memory	Write Result.
LDI	Load Imm. Into Reg.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Read Memory	Write Result.
LPM	Load Reg From P.M.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Read Memory	Write Result.
STR	Store Reg. To Mem.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Write Memory	
MOV	Move 1 Mem. Val To Diff. Addr.	Not Applicable	Not Applicable	Not Applicable	Not Applicable	Not Applicable
MOVR	Move A Reg. Value To Diff Reg.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Not Applicable	Write Result.
JMP	Jump To Given Addr	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Not Applicable	Not Applicable
BREQ	If Zero Set, Jump To Given Addr	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Not Applicable	Not Applicable
BRNE	If Zero Clr, Jump To Given Addr	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Not Applicable	Not Applicable
CALL	Push PC++ To Stack & Jump To Given Addr	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Push Next Address to Call Stack Increment Call Stack Pointer	Not Applicable
RET	Pop Stack To P.C. INC S.P.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Pop Address To Program Counter Decrement Call Stack Pointer	Not Applicable
RETI	Pop Stack To P.C., INC S.P., Dec Intrpt. Cntr.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Pop Address To Program Counter Decrement Call Stack Pointer	Not Applicable
PUSH	Push Reg. To Stack, Dec S.P.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Push Next Address to Call Stack Increment Call Stack Pointer	Not Applicable
POP	Pop Stack To Reg, Inc S.P.	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Read Memory Pointed to by SP Increment SP	Write Result.
HLT	Halt Execution Until Reset	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Not Applicable	Not Applicable
TINT	Toggle Interrupt Enable Flag	Get Word From Program Memory Latch Word Into Fet./Dec/. Pipeline Register.	Not Applicable	Not Applicable	Write Result.	Nothing

8.4 D: Final Test Program

Final Test Code:

```
;Reset Interrupt Vector
      JMP INIT
;V-Blank Interrupt
      JMP halt_execution
;Illegal Opcode Exception
      JMP MAIN

:INIT
      ;Initialize The Stack Pointer
      LDI R0 0xFF
      OUT SPH R0
      OUT SPL R0

      ;Initialize The Call Stack Pointer.
      LDI R0 0x00
      OUT CSP R0

      LDI R31 0x00 ;Set up sucess flag
      LDI R30 0x00 ;This register will be used to track which test the
program may fail on.

      LDI R0 0xFF
      OUT T1CR R0

:MAIN
      CALL TEST_INC ;LED = 0x01
      CALL TEST_DEC ;LED = 0x02
      CALL TEST_ADD ;LED = 0x03
      CALL TEST_ADDI ;LED = 0x04
      CALL TEST_SUB ;LED = 0x05
      CALL TEST_SUBI ;LED = 0x06
      CALL TEST_CP ;LED = 0x07
      CALL TEST_CPI ;LED = 0x08
      CALL TEST_MUL ;LED = 0x09
      CALL TEST_MULI ;LED = 0x0A
      CALL TEST_AND ;LED = 0x0B
      CALL TEST_ANDI ;LED = 0x0C
      CALL TEST_OR  ;LED = 0x0D
      CALL TEST_ORI ;LED = 0x0E
      CALL TEST_SHL ;LED = 0x0F
```

```
      CALL TEST_SHR ;LED = 0x10
      CALL TEST_COM ;LED = 0x11
      CALL TEST_INV ;LED = 0x12
      CALL TEST_LDI ;LED = 0x13
      CALL TEST_LPM ;LED = 0x14
      CALL TEST_STR_LD ;LED = 0x15
      CALL TEST_STFB_LDFB ;LED = 0x16
      CALL TEST_PUSH_POP ;LED = 0x17
      CALL TEST_MOV ;LED = 0x18
      CALL TEST_MOVR ;LED = 0x19
      CALL TEST_OUT_IN ;LED = 0x1A
      CALL TEST_JMP ;LED = 0x1B
      CALL TEST_BRCS ;LED = 0x1C
      CALL TEST_BRCC ;LED = 0x1D
      CALL TEST_BREQ ;LED = 0x1E
      CALL TEST_BRNE ;LED = 0x1F
      CALL TEST_BRNG ;LED = 0x20
      CALL TEST_BRPS ;LED = 0x21

:vga_out
      LDI R0 0x00
      OUT T1CR R0
      CALL WRITE_VGA
:halt_execution
      OUT LED R30
      ;Halt
      HLT

;Test Increment Instruction
:TEST_INC
      LDI R0 0x00
      INC R0
      CPI R0 0x01
      BREQ test_inc_ret
      CALL TEST_FAILED
:test_inc_ret
      INC R30
      OUT LED R30
      RET

;Test Decrement Instruction
:TEST_DEC
      LDI R0 0x01
```

```
      DEC R0
      BREQ test_dec_ret
      CALL TEST_FAILED
:test_dec_ret
      INC R30
      OUT LED R30
      RET

;Test Addition Instruction.
:TEST_ADD
      LDI R0 0x5A
      LDI R1 0xA5
      NOP ;Prevent Data Hazard Side Effect
      NOP ;Prevent Data Hazard Side Effect
      ADD R1 R0
      CPI R1 0xFF
      BREQ test_add_ret
      CALL TEST_FAILED
:test_add_ret
      INC R30
      OUT LED R30
      RET

;Test Add Immeadiate Instruction
:TEST_ADDI
      LDI R0 0x5A
      ADDI R0 0xA5
      CPI R0 0xFF
      BREQ test_addi_ret
      CALL TEST_FAILED
:test_addi_ret
      INC R30
      OUT LED R30
      RET

;Test Subtract Instruction
:TEST_SUB
      LDI R0 0xFF
      LDI R1 0x0F
      NOP ;Prevent Data Hazard Side Effect
      NOP ;Prevent Data Hazard Side Effect
      SUB R0 R1
      CPI R0 0xF0
```

```
      BREQ test_sub_ret
      CALL TEST_FAILED
:test_sub_ret
      INC R30
      OUT LED R30
      RET

;Test Immeadiate Subtraction
:TEST_SUBI
      LDI R0 0xFF
      SUBI R0 0x0F
      CPI R0 0xF0
      BREQ test_subi_ret
      CALL TEST_FAILED
:test_subi_ret
      INC R30
      OUT LED R30
      RET

;Test Register to Register Compare
:TEST_CP
      LDI R0 0xFF
      LDI R1 0xF0
      NOP ;Prevent Data Hazard Side Effect
      NOP ;Prevent Data Hazard Side Effect
      CP R0 R1
      BRNE test_cp_2
      CALL TEST_FAILED
:test_cp_2
      LDI R0 0xFF
      LDI R1 0xFF
      NOP ;Prevent Data Hazard Side Effect
      NOP ;Prevent Data Hazard Side Effect
      CP R0 R1
      BREQ test_cp_ret
      CALL TEST_FAILED
:test_cp_ret
      INC R30
      OUT LED R30
      RET

;Test Compare Immeadiate Instruction
:TEST_CPI
```

```
      LDI R0 0xFF
      CPI R0 0xF0
      BRNE test_cpi_2
      CALL TEST_FAILED
:test_cpi_2
      LDI R0 0xFF
      CPI R0 0xFF
      BREQ test_cpi_ret
      CALL TEST_FAILED
:test_cpi_ret
      INC R30
      OUT LED R30
      RET

;Test Multiply Instruction
:TEST_MUL
      LDI R0 0xFF
      LDI R1 0x03
      NOP ;Prevent Data Hazard Side Effect
      NOP ;Prevent Data Hazard Side Effect
      MUL R0 R1
      CPI R0 0xFD
      BREQ test_mul_chk_hb
      CALL TEST_FAILED
:test_mul_chk_hb
      CPI R1 0x02
      BREQ test_mul_ret
      CALL TEST_FAILED
:test_mul_ret
      INC R30
      OUT LED R30
      RET

;Test Multiply Immeadiate Instruction
:TEST_MULI
      LDI R0 0xFF
      MULI R0 R1 0x03
      CPI R0 0xFD
      BREQ test_mul_chk_hb
      CALL TEST_FAILED
:test_mul_chk_hb
      CPI R1 0x02
      BREQ test_mul_ret
```

```
      CALL TEST_FAILED
:test_mul_ret
      INC R30
      OUT LED R30
      RET

;Test And Immeadiate Instruction
:TEST_AND
      LDI R0 0xFF
      LDI R1 0x0F
      NOP ;Prevent Data Hazard Side Effect
      NOP ;Prevent Data Hazard Side Effect
      AND R1 R0
      CPI R1 0x0F
      BREQ test_and_ret
      CALL TEST_FAILED
:test_and_ret
      INC R30
      OUT LED R30
      RET

;Test Immeadiate Bitwise And Instruction
:TEST_ANDI
      LDI R0 0xFF
      ANDI R0 0x0F
      CPI R0 0x0F
      BREQ test_andi_ret
      CALL TEST_FAILED
:test_andi_ret
      INC R30
      OUT LED R30
      RET

;Test Or Instruction
:TEST_OR
      LDI R0 0xF0
      LDI R1 0x0F
      NOP ;Prevent Data Hazard Side Effect
      NOP ;Prevent Data Hazard Side Effect
      OR R1 R0
      CPI R1 0xFF
      BREQ test_or_ret
      CALL TEST_FAILED
```

```
:test_or_ret
      INC R30
      OUT LED R30
      RET

;Test Or Immeadiate Instruction
:TEST_ORI
      LDI R0 0xF0
      ORI R1 0x0F
      CPI R1 0xFF
      BREQ test_or_ret
      CALL TEST_FAILED
:test_or_ret
      INC R30
      OUT LED R30
      RET

;Test Shift Right Instruction
:TEST_SHR
      LDI R0 0xFF
      SHR R0
      CPI R0 0x7F
      BREQ test_shr_ret
      CALL TEST_FAILED
:test_shr_ret
      INC R30
      OUT LED R30
      RET

;Test Shift Left Instruction
:TEST_SHL
      LDI R0 0xFF
      SHL R0
      CPI R0 0xFE
      BREQ test_shl_ret
      CALL TEST_FAILED
:test_shl_ret
      INC R30
      OUT LED R30
      RET

;Test 2's Complement Instruction
:TEST_COM
```

```
      LDI R0 0x01
      COM R0
      CPI R0 0xFF
      BREQ test_com_ret
      CALL TEST_FAILED
:test_com_ret
      INC R30
      OUT LED R30
      RET

;Test Bit Inversion Instruction
:TEST_INV
      LDI R0 0x00
      INV R0
      CPI R0 0xFF
      BREQ test_inv_ret
      CALL TEST_FAILED
:test_inv_ret
      INC R30
      OUT LED R30
      RET

;Load Immeadiate Instruction Test
:TEST_LDI
      LDI R0 0x4B
      NOP ;These two nops are not neccessary for correct execution, but by
placing them after the immeadiate load, it will insure that the value of R0
read from the register file is compared to the imemadiate constant in the
CPI Instruction.
      NOP
      CPI R0 0x4B
      BREQ test_ldi_ret
      CALL TEST_FAILED
:test_ldi_ret
      INC R30
      OUT LED R30
      RET

:TEST_LPM
      ;Initialize Memory Pointer to point at the opcode of the first
instruciton of this program (JMP)
      LDI R0 0x00
      OUT ZH R0
```

```
      OUT ZL R0
      LPM R1 Z ;This and the next instruction will cause a pipeline stall
and then a forwarding of the bottom data word in MEM/WB to the top/primary
input of the alu.
      CPI R1 0x38
      BREQ test_lpm_ret
      CALL TEST_FAILED
:test_lpm_ret
      INC R30
      OUT LED R30
      RET

;This Tests LD, STR and each memory Pointer and Its post Increments.
:TEST_STR_LD
      ;Initialize Memory Pointers
      LDI R0 0x0F
      OUT XH R0
      OUT XL R0
      LDI R10 0x0D
      OUT YH R0
      OUT YL R10
      LDI R11 0x0B
      OUT ZH R0
      OUT ZL R11
      LDI R1 0x72
      LDI R2 0x71
      LDI R3 0x70
      STR X+ R1
      STR Y+ R2
      STR Z+ R3
      STR X R1
      STR Y R2
      STR Z R3
      ;Check Pointer Post Increments
      IN R4 XL
      IN R5 YL
      IN R6 ZL
      CPI R4 0x10
      BREQ yl_inc_test
      CALL TEST_FAILED
:yl_inc_test
      CPI R5 0x0E
      BREQ zl_inc_test
```

```
      CALL TEST_FAILED
:zl_inc_test
      CPI R6 0x0C
      BREQ load_test_begin
      CALL TEST_FAILED
:load_test_begin
      ;Initialize Memory Pointers
      LDI R0 0x0F
      LDI R11 0x0B
      OUT ZH R0
      OUT ZL R11
      LD R4 Z+
      CP R4 R3
      BREQ test_ld_1
      CALL TEST_FAILED
:test_ld_1
      LD R4 Z+
      CP R4 R3
      BREQ test_ld_2
      CALL TEST_FAILED
:test_ld_2
      LD R4 Z+
      CP R4 R2
      BREQ test_ld_3
      CALL TEST_FAILED
:test_ld_3
      LD R4 Z+
      CP R4 R2
      BREQ test_ld_4
      CALL TEST_FAILED
:test_ld_4
      LD R4 Z+
      CP R4 R1
      BREQ test_ld_5
      CALL TEST_FAILED
:test_ld_5
      LD R4 Z+
      CP R4 R1
      BREQ test_str_ld_ret
      CALL TEST_FAILED
:test_str_ld_ret
      INC R30
      OUT LED R30
```

```
      RET

;Test Load & Store to Framebuffer
:TEST_STFB_LDFB
      ;Initialize register with a write value.
      LDI R1 0xFF
      LDI R2 0xFF
      OUT LED R1
      OUT LED R2
      ;Initialize Memory Pointer
      LDI R0 0xAA
      OUT ZH R0
      OUT ZL R0
      ;STFB The Data
      STFB Z R1 R2
      ;LDFB The Data
      LDFB R3 R4 Z
      CP R3 R2
      BREQ test_ldfb_h
      CALL TEST_FAILED
:test_ldfb_h
      CPI R4 0x0F ; Framebuffer is only 12 bits wide thus top nibble get
zeroed out when reading in.
      BREQ test_stfb_ldfb_ret
      CALL TEST_FAILED
:test_stfb_ldfb_ret
      INC R30
      OUT LED R30
      RET

;Test Push and Pop
:TEST_PUSH_POP
      ;Initialize Stack Pointer
      LDI R0 0xFF
      OUT SPH R0
      OUT SPL R0
      LDI R1 0x56
      PUSH R1
      ;Check Stack Pointer Decrement
      NOP ;Nop For safety
      IN R1 SPL
      CPI R1 0xFE
      BREQ test_pop
```

```
      CALL TEST_FAILED
:test_pop
      POP R1
      CPI R1 0x56 ;Stall and forward should occur here.
      BREQ test_push_pop_ptr_inc
      CALL TEST_FAILED
:test_push_pop_ptr_inc
      IN R1 SPL
      CPI R1 0xFF
      BREQ test_push_pop_ret
      CALL TEST_FAILED
:test_push_pop_ret
      INC R30
      OUT LED R30
      RET

;Test the move assembler alias this is implemented as a LD R0 ptr1 STR ptr2
R0, thus destroying the contents of R0
:TEST_MOV
      ;Initialize Memory Pointers
      LDI R0 0xAA
      OUT ZH R0
      OUT ZL R0
      LDI R1 0xBB
      OUT YH R1
      OUT YL R1
      STR Z R1
      MOV Y Z ;Move the value pointed to by Z to the memory pointed to by Y
      LD R0 Y
      CP R0 R1 ;This should stall then forward.
      BREQ test_mov_ret
      CALL TEST_FAILED
:test_mov_ret
      INC R30
      OUT LED R30
      RET

;Test Moving of a register value
:TEST_MOVR
      LDI R0 0x37
      MOVR R1 R0
      CP R1 R0
      BREQ test_movr_ret
```

```
      CALL TEST_FAILED
:test_movr_ret
      INC R30
      OUT LED R30
      RET

;Test in and out instructions
:TEST_OUT_IN
      LDI R0 0xA3
      OUT LED R0
      IN R1 LED
      CP R1 R0
      BREQ test_out_in_ret
      CALL TEST_FAILED
:test_out_in_ret
      INC R30
      OUT LED R30
      RET

;Test jump instruciton
:TEST_JMP
      JMP test_jmp_ret
      CALL TEST_FAILED
:test_jmp_ret
      INC R30
      OUT LED R30
      RET

;Test Branch if Carry Set
:TEST_BRCS
      LDI R0 0x01
      ADDI R0 0xFF
      BRCS test_brcs_1
:brcs_2
      CALL TEST_FAILED
      JMP test_brcs_ret
:test_brcs_1
      LDI R0 0x00
      ADDI R0 0x0A
      BRCS brcs_2
:test_brcs_ret
      INC R30
      OUT LED R30
```

```
      RET

;Test Branch if Carry Clear
:TEST_BRCC
      LDI R0 0x00
      ADDI R0 0x0A
      BRCC test_brcc_1
:brcc_2
      CALL TEST_FAILED
      JMP test_brcc_ret
:test_brcc_1
      LDI R0 0x01
      ADDI R0 0xFF
      BRCC brcc_2
:test_brcc_ret
      INC R30
      OUT LED R30
      RET

;Test Branch if Equal i.e. Zero flag = 1 for taken, 0 for not taken.
:TEST_BREQ
      LDI R0 0xFF
      ADDI R0 0x01
      BREQ test_breq_1
:test_breq_2
      CALL TEST_FAILED
      JMP test_breq_ret
:test_breq_1
      LDI R0 0x0A
      ADDI R0 0x01
      BREQ test_breq_2
:test_breq_ret
      INC R30
      OUT LED R30
      RET

;Test Branch if Not Equal i.e. Zero flag = 1 for not taken, 0 for taken.
:TEST_BRNE
      LDI R0 0x0A
      ADDI R0 0x01
      BRNE test_brne_1
:test_brne_2
      CALL TEST_FAILED
```

```
      JMP test_brne_ret
:test_brne_1
      LDI R0 0xFF
      ADDI R0 0x01
      BRNE test_brne_2
:test_brne_ret
      INC R30
      OUT LED R30
      RET

;Test Branch if Negative i.e. Negative Flag = 1 for taken, 0 for not taken.
:TEST_BRNG
      LDI R0 0x7F
      ADDI R0 0x01
      BRNG test_brng_1
:test_brng_2
      CALL TEST_FAILED
      JMP test_brng_ret
:test_brng_1
      LDI R0 0x7E
      ADDI R0 0x01
      BRNG test_brng_2
:test_brng_ret
      INC R30
      OUT LED R30
      RET

;Test Branch if Positive i.e. Negative Flag = 0 for taken, 1 for not taken.
:TEST_BRPS
      LDI R0 0x7E
      ADDI R0 0x01
      BRPS test_brps_1
:test_brps_2
      CALL TEST_FAILED
      JMP test_brps_ret
:test_brps_1
      LDI R0 0x7F
      ADDI R0 0x01
      BRPS test_brps_2
:test_brps_ret
      INC R30
      OUT LED R30
      RET
```

```
;If any of the tests fails, this subroutine is called and the global fail
flag is set thus, when the processor finishes execution, all red will be
written to the framebuffer rather than all green.
:TEST_FAILED
      LDI R31 0xFF
      JMP vga_out

;These Do not get their Own tests as the program puts these instructions
through their paces
:TEST_CALL
:TEST_RET
:TEST_RETI
:TEST_HLT

:WRITE_VGA
      ;Initialize Memory Pointer
      LDI R0 0x00
      OUT ZH R0
      OUT ZL R0
      CPI R31 0x00
      BRNE green
;Red
      LDI R4 0x00
      LDI R5 0x0F
      JMP write_fb
:green
      LDI R4 0xF0
      LDI R5 0x00
:write_fb
      LDI R0 0x80
:fb_lop_2
      LDI R1 0x96
:fb_lop_1
      STFB Z+ R4 R5
      DEC R1
      BRNE fb_lop_1
      DEC R0
      BRNE fb_lop_2
      RET
```