The American University in Cairo

FemtoRV32ic

Project 1 – Computer Architecture

Hunter CPU: Final report

Habiba Gamal 900151007

Ali El-Said 900150264

Ahmed Wael Fahmy 900160127

1. Overall Design:

1.1. Design Description:

This RISC-V implementation has several constraints like:

1) the register file is implemented as a dual ported memory instead of triple ported memory.

2) the instruction and the data share the same memory.

These constraints introduced structural hazards. As the memory needs to be accessed for fetching instructions in the fetch stage, and for, reading or writing data in the MEM stage.

Moreover, the register file needs to be accessed to read the data of rs1 and rs2 in the Decode stage, and to write data in rd in the WB stage.

These structural hazards made us implement the RISC-V as a 3-stage pipeline CPU.

The first stage is: fetch and decode (referred to as: F_D)

The second stage is: execute and memory (referred to as: EX_MEM)

The third stage is: register file (RF) write back (referred to as: WB)

Each stage lasts for 2 clock cycles.

This is why we introduced a new signal called “Alternating Signal”. This signal is the clock divider by 2 of the normal clock. This alternating signal alternates every 2 clock cycles. In other words, it alternates every STAGE.

We have only 2 pipeline registers. the first one is after the first stage (referred to as EX). The second one is after the second stage (referred to as WB).

By this, all structural hazards are eliminated.

1.2. Potential Hazards:

Data Hazards:

RAW: In order to fix this hazard, we implemented a forwarding unit that forwards the data that will be written to the register file, before it is actually written. This is multiplexed with rs1 and rs2 to get ALU operand 1 and ALU operand 2.

WAW: This hazard does not exist in our implementation because of in-order fetching and execution.

WAR: This hazard does not exist in our implementation because of in-order fetching and execution.

Control Hazards:

The branch prediction of our implementation is static and it is always not taken. When the branch is taken, the instruction that was fetched is flushed.

1.3. Changes Made Since Last Submission:

1) Addition of CSRs like mepc, mie, mip, mtimecmp, mtime, mcycle, minstret:
   The CSRs are added in a separate module called CSR.

2) Supporting system instructions like CSRRW, CSRRC, CSRRS, CSRRWI, CSRRCI, CSRRSI, mret, EBREAK and ECALL:
   Supporting CSRR instructions resulted in the addition of an input to the multiplexer that selects what gets written to the register file, as the output of the CSR module can now be written to the register file.

   Reading and writing to the CSRs using a CSRR instruction is treated in our datapath pretty much like reading and writing to the register file. In the decode stage, we read the data of the CSR that will be written to the register file, and in the WB stage, the CSR gets written to with the relevant input. For CSRRW, the content of rs1 is written, for CSRRWI, the immediate is written. For CSRRS, CSRRSI, CSRRC, CSRRCI, what gets written to the CSR is computed using a certain combinational logic. All these different options for the input to the CSR are multiplexed and the selection line is outputted from the control unit depending on which CSRR instruction it is.

   Some CSRs are written to without the need of an instruction and some CSRs are read only and are written to without a CSRR instruction at all. More details about CSR module are in section 3.5.

3) Supporting NMI, 8 external I/O interrupts (IRQ) and timer interrupt:
   As for supporting EBREAK, ECALL, timer interrupt, NMI and IRQ, it is treated as a branch taken or a jump. When they occur, the pc is updated with the handler address and the sequential instruction that was fetched will be flushed. EBREAK and ECALL handlers are jumped to in the execution stage.

4) The memory was previously a part of the datapath, now we implemented it as a module outside of the datapath, but in the top module (environment.v). More details are in section 3.1.

1.4. Area and Delay of CPU:

Wire Load: none

Gates: 12045 (25.0 %)

Cap: 31.5 ff (0.7 %)

Area: 302659.00 (97.0 %)

Delay: 6694.24 ps (3.2 %)

2. Datapath:

2.1. Highlighted Datapath for R instructions:

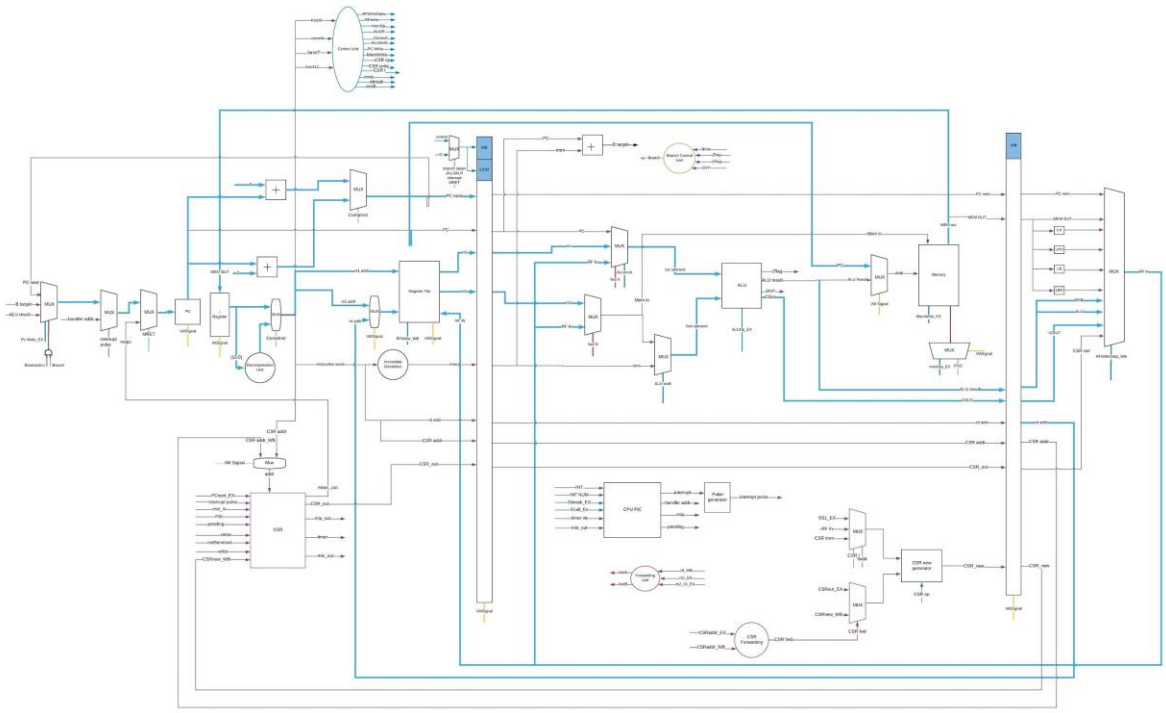

Figure 1: Highlighted Block diagram for R instructions (compressed and uncompressed)

Instructions covered:

add, sub, sll, slt, sltu, xor, srl, sra, or, and

c.sub, c.xor, c.or, c.and, c.mv, c.li, c.add

2.2. Highlighted Datapath for I instructions:

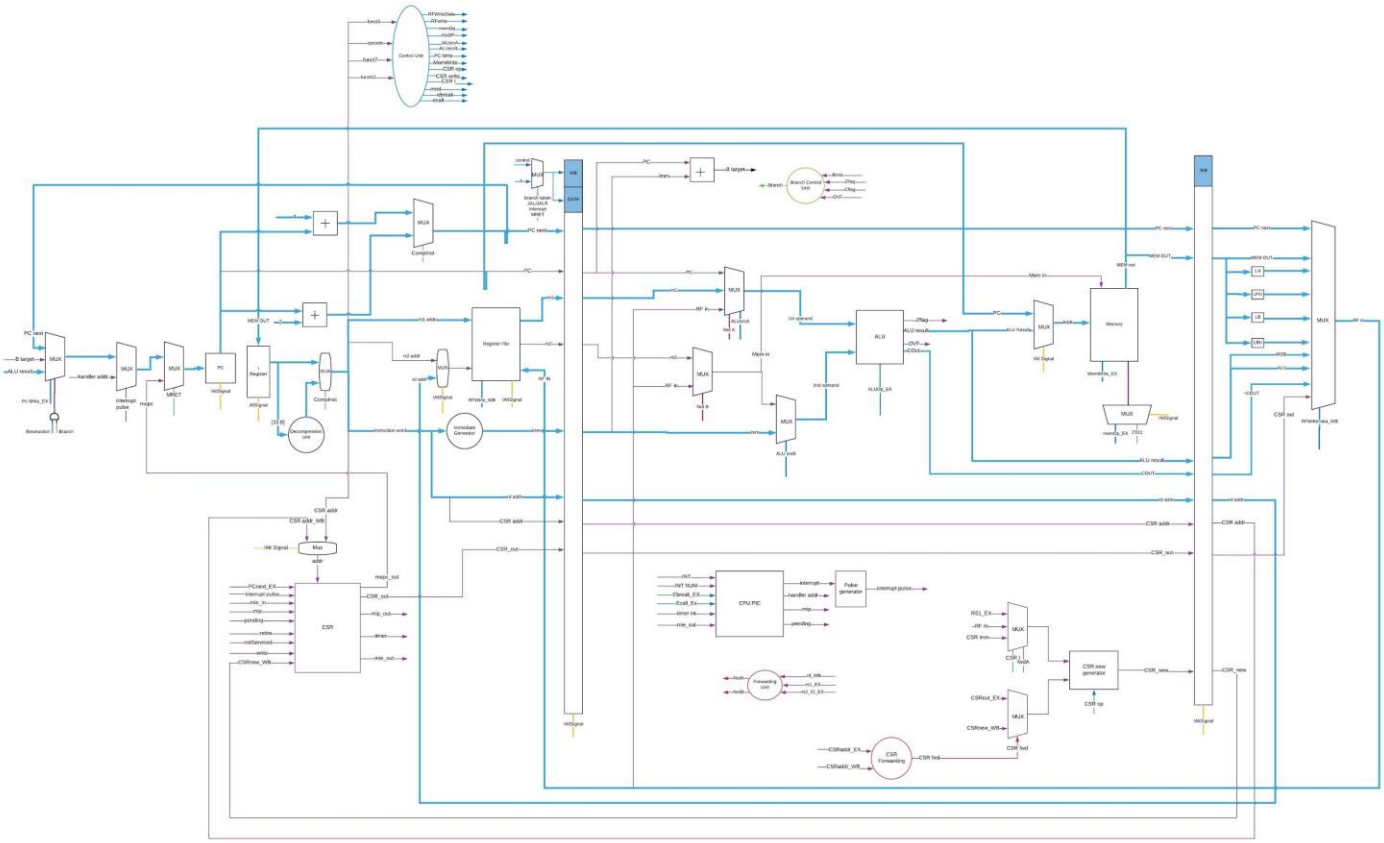

Figure 2: Highlighted Block diagram for I instructions (compressed and uncompressed)

Instructions covered:

addi, slti, sltiu, xori, ori, andi, slli, srli, srai, lb, lh, lhu, lbu, lw, jalr

c.addi4spn, c.lw, c.addi, c.addi16sp, c.srli, c.srai, c.andi, c.slli, c.nop, c.jalr, c.jr

2.3. Highlighted Datapath for S instructions:

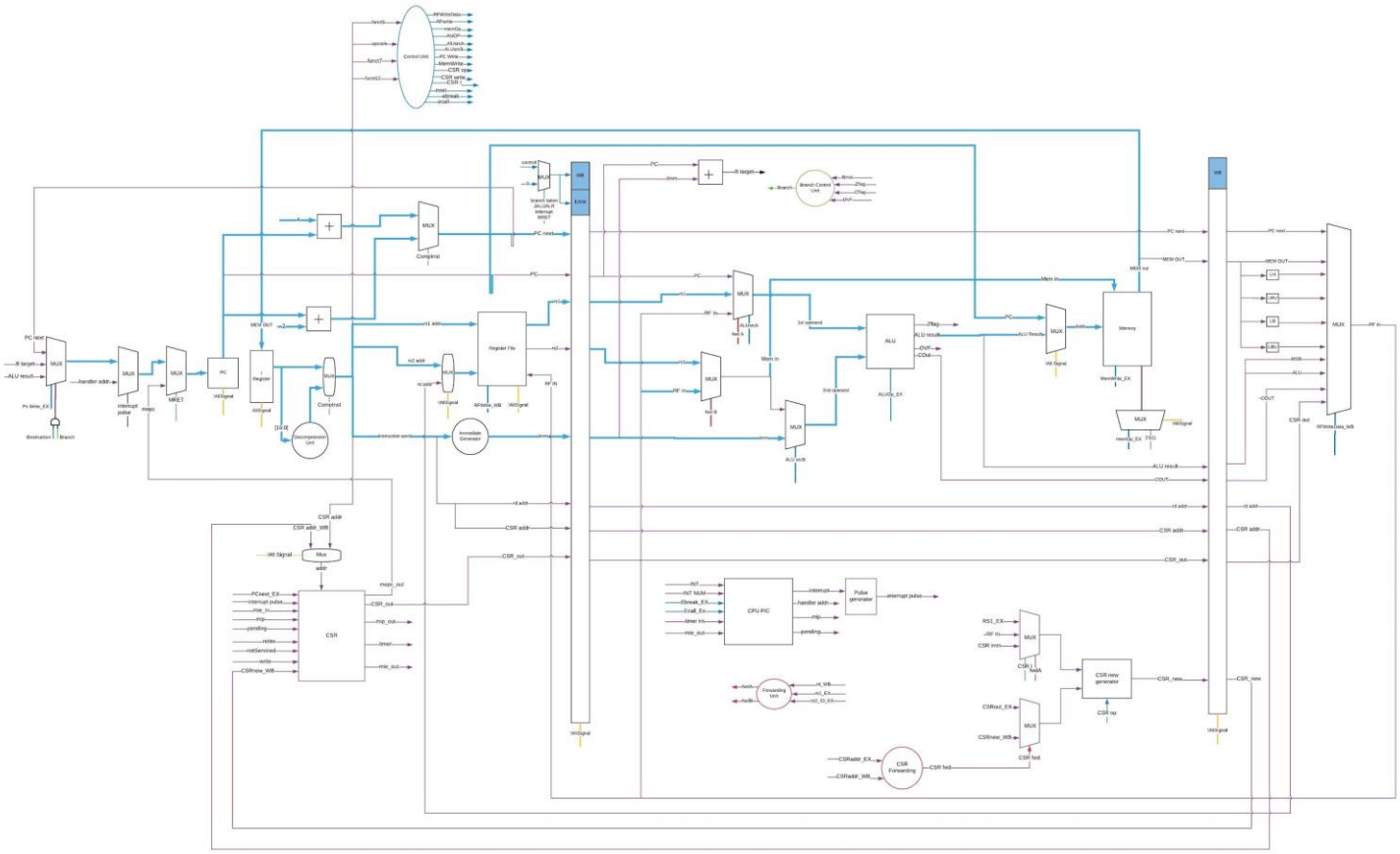

Figure 3: Highlighted Datapath for S instructions

Instructions Covered:

sb, sh, sw

c.sw

2.4. Highlighted Datapath for B instructions:

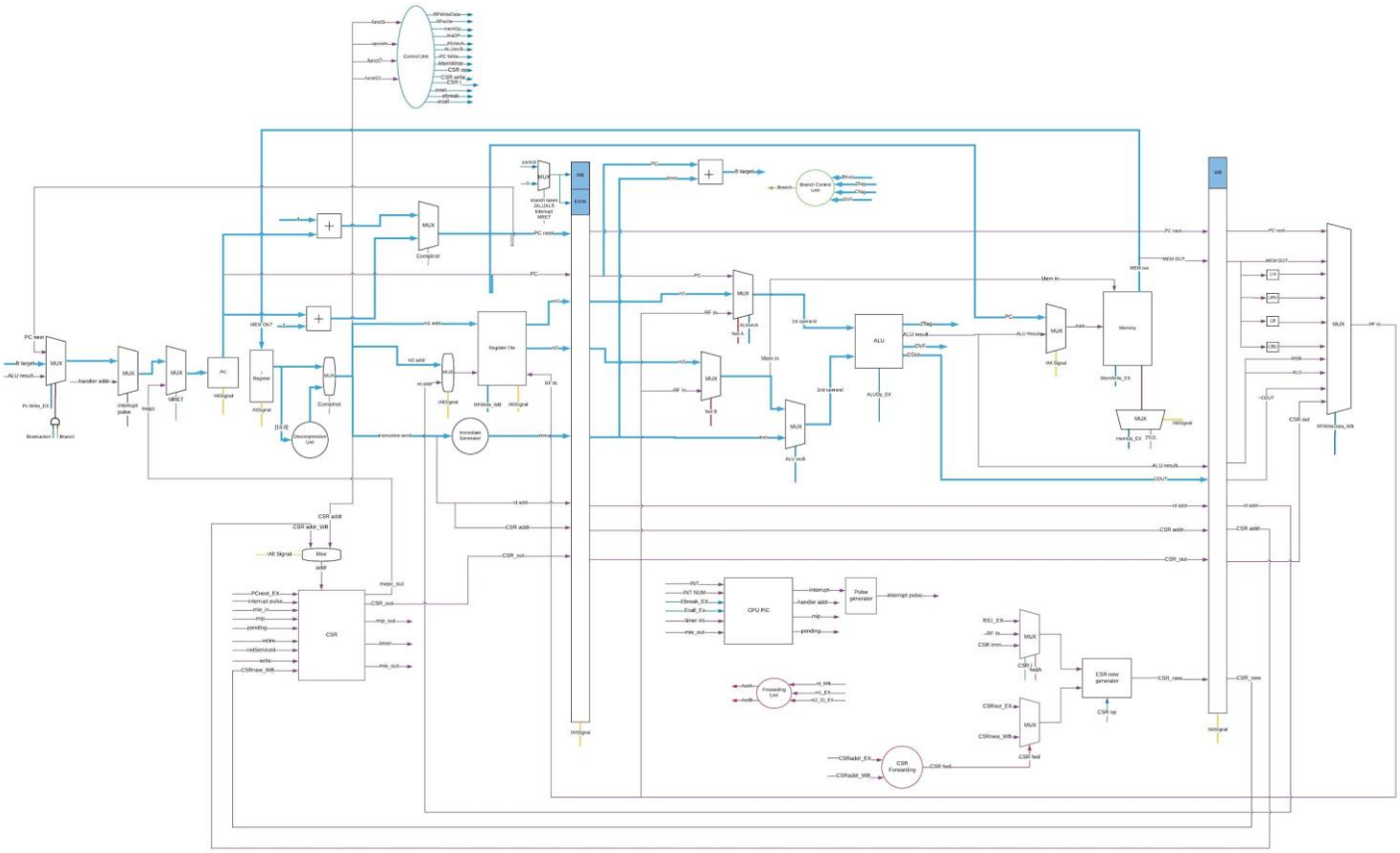

Figure 4: Highlighted Block Diagram for B instructions

Instructions Covered:

beq, bne, blt, bge, bltu, bgeu

c.beqz, c.bnez

2.5. Highlighted Datapath for U instructions:

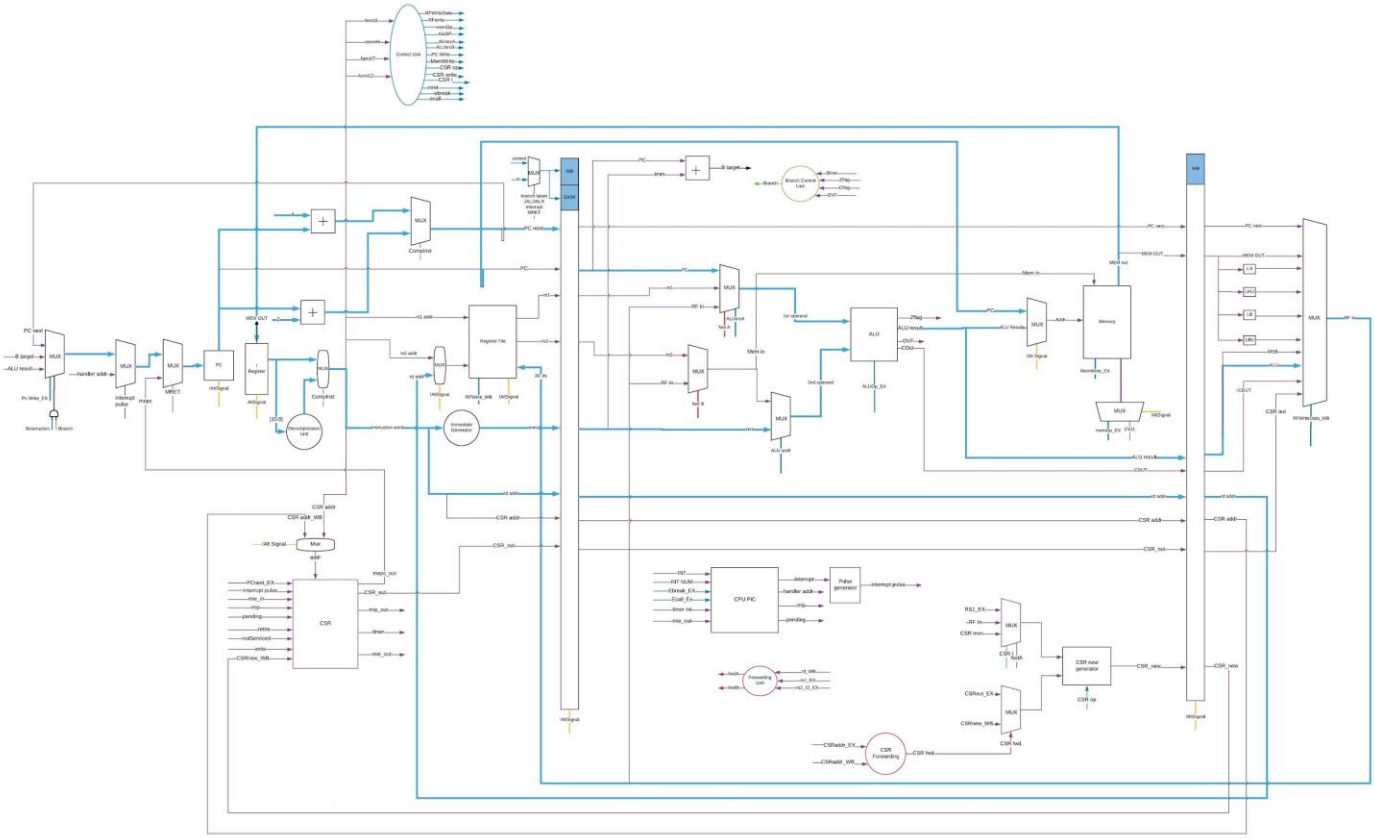

Figure 5: Highlighted datapath for U instructions

Instructions Covered:

lui, auipc

c.lui

2.6. Highlighted Datapath for J instructions:

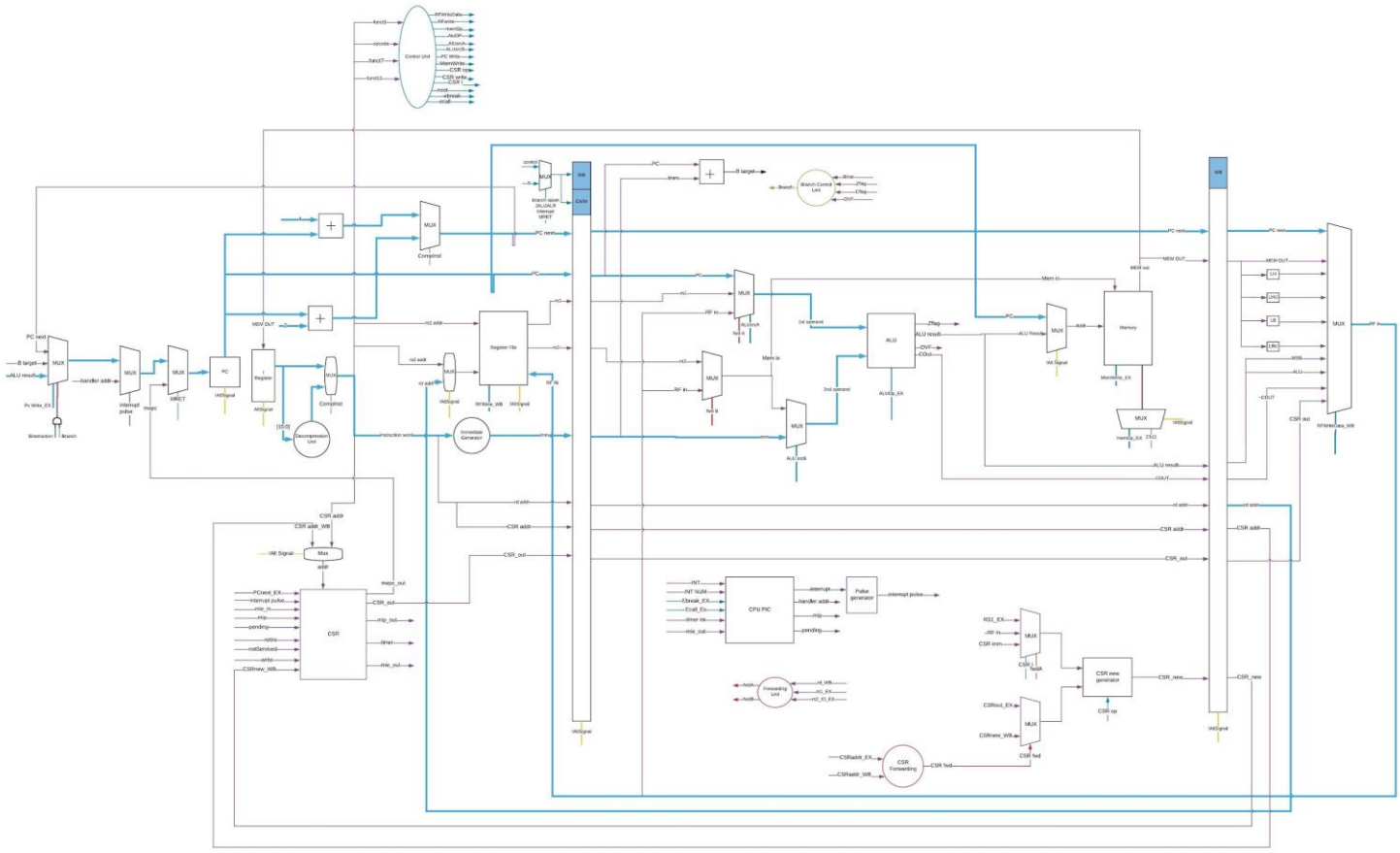

Figure 6: Highlighted Datapath for J instructions

Instructions Covered:

jal

c.j, c.jal

2.7. Highlighted Datapath for System instructions:

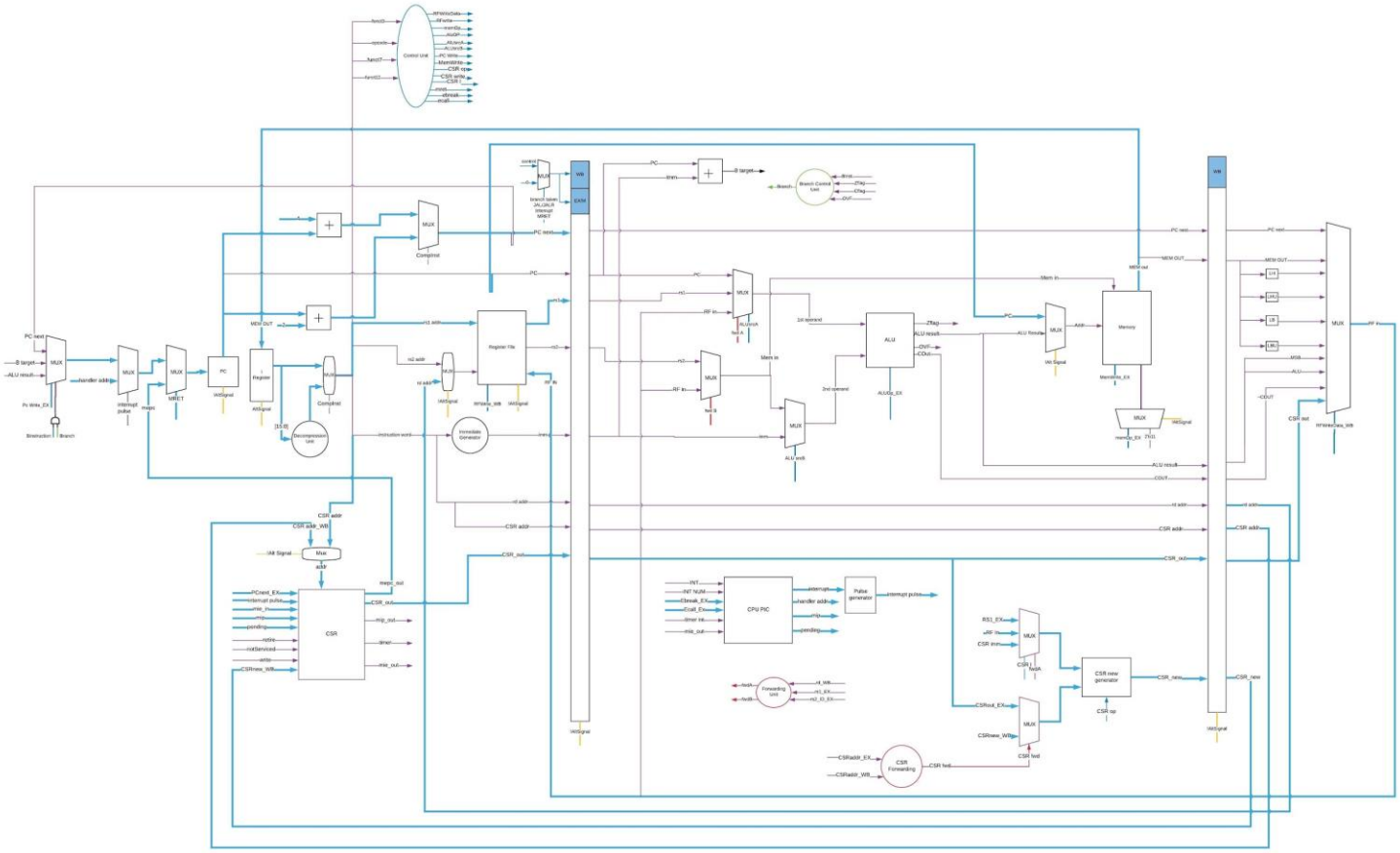

Figure 7: Highlighted datapath for system instructions

Instructions covered:

csrrw, csrrs, csrrc, csrrwi, csrrsi, csrrci, ebreak, ecall, mret

c.ebreak

2.8. Whole Datapath:

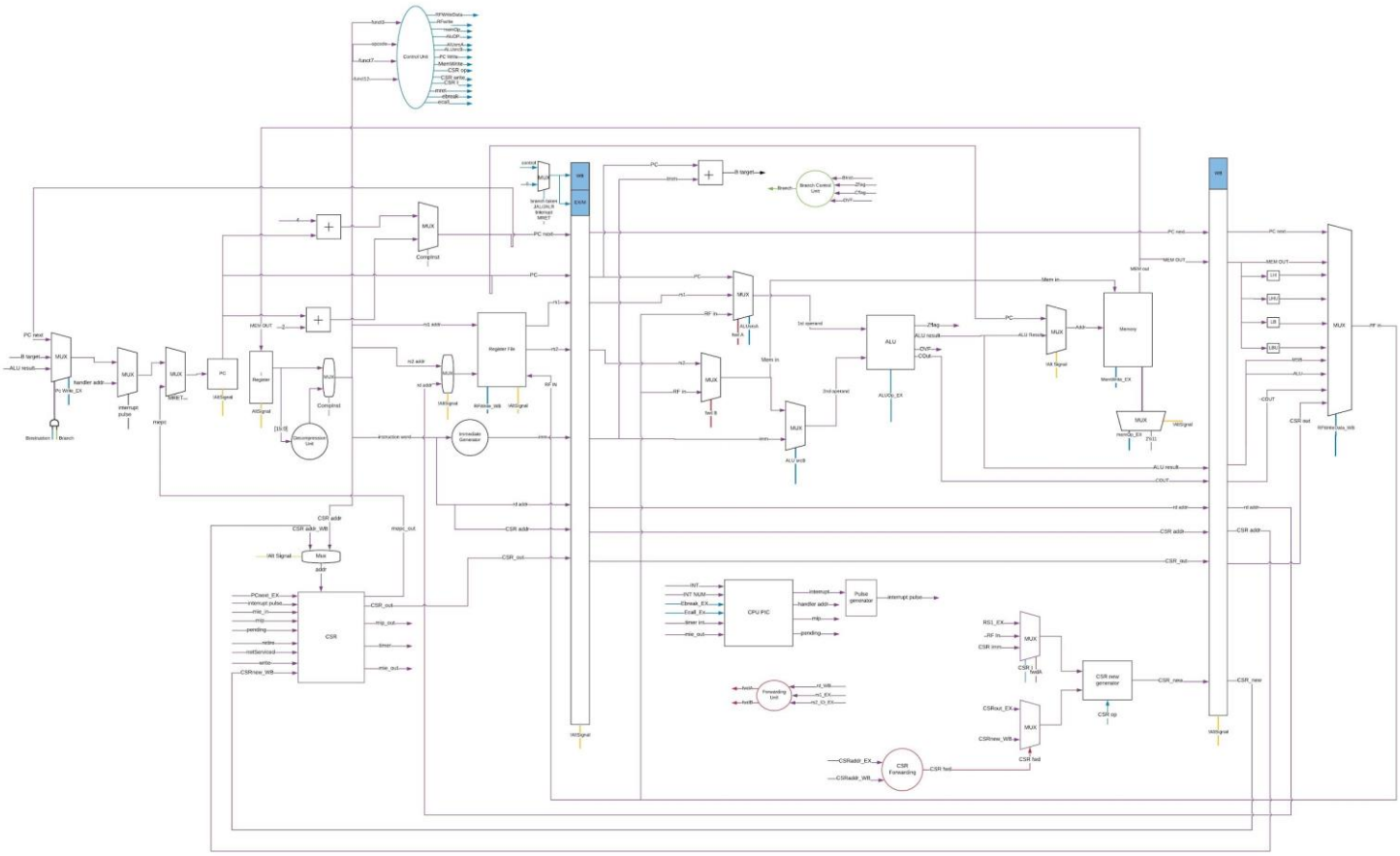

Figure 8: Datapath for all instructions

3. Modules:

3.1. Top Module:

● Module Name:

environment.v

● How It Works:

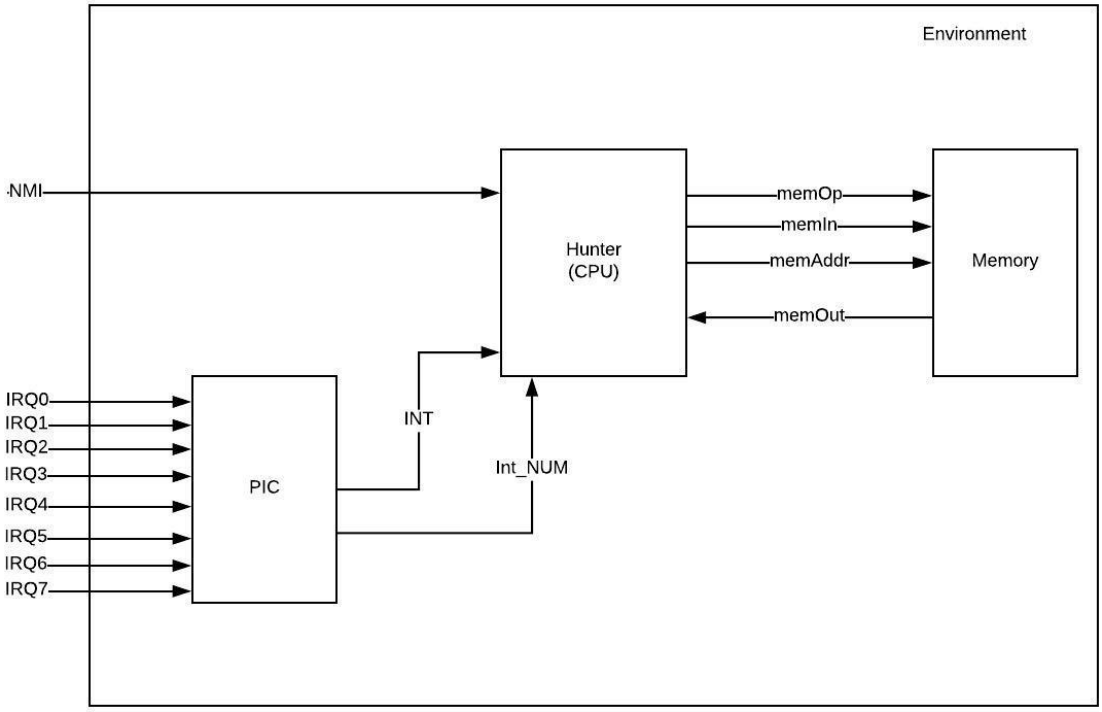

Figure 9: Block diagram for environment

This is the top module that encompases the PIC, datapath for the CPU, and memory.

The three components are able to interface with each other inside this module.

● Design Decisions, Debates and Rationale:

This module simulates the environment that our CPU might be inside. Hence, it contains a PIC that handles the interrupts that are external to our CPU and it has a signal that represents the non-maskable interrupts. In addition, it contains the memory module that represents our main instruction and data memory that the CPU utilizes.

At first, the memory module was inside the CPU module, however in the end when we were in the synthesis step, we found that it was better to remove the memory module from inside the datapath module so we can be able to synthesise it alone without the dependencies on the memory module, especially that the memory module generates an insane amount of area when synthesising.

● Ports:

Port Name	Type	Function
clk	Input	Main clock
rst	Input	Global reset
NMI	Input	Signal to indicate a non-maskable interrupt
IRQ	Input	Bus of wires that represents any external interrupts that our CPU needs to handle
hunterOut	Output	Dummy output propagated from the datapath to enable us to successfully synthesis and implement our design on Vivado

3.2.CPU:

● Module Name:

hunter_RV32.v

● How It Works:

This module is the whole datapath of our CPU. It is the biggest module that holds all the logic inside Hunter.

● Design Decisions, Debates and Rationale:

(Discussed in detail in Section 1)

● Ports:

Port Name	Type	Function
clk	Input	main clock
rst	Input	global reset
NMI	Input	non-maskable external interrupt signal
INT	Input	external interrupt signal
INT_NUM	Input	signal that specifies which external interrupt
memOut	Input	output of memory module
memInOUT	Output	write data to memory module
memOpOUT	Output	memory operation 00: read 01: sh 10: sb 11: sw
memAddrOUT	Output	memory address
hunterOut	Output	dummy output that was propagated for synthesis purposes

● Area and Delay:

WireLoad: none

Gates: 12045 (25.0 %)

Cap: 31.5 ff (0.7 %)

Area: 302659.00 (97.0 %)

Delay: 6694.24 ps (3.2 %)

3.3.Memory:

● Module Name:

memory.v

● How It Works:

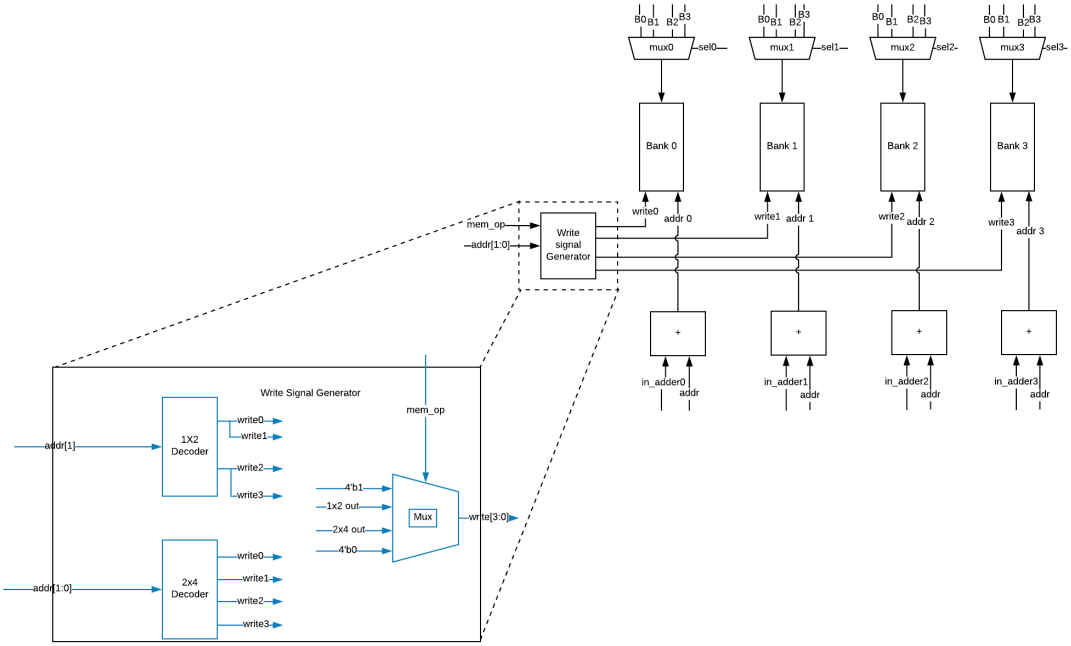

Figure 10: Block diagram of the memory

This memory design supports both unaligned reads and writes.

This memory design is little endian.

This design proved helpful when supporting compressed instructions as it allowed reading unaligned words from the memory. This was crucial as compressed instructions are often half word aligned.

In order to support unaligned word reads: 4 adders were added to compute the address that will be used to access each byte in the memory. If the word is aligned, all 4 bytes will be read from the same address (same word). However, if the word is unaligned, we might need to read 2 bytes from the word addressed by the input address and 2 bytes from the following word. This is taken care of using the 4 adders added at the address input of each memory bank.

The concatenation of outputs of the memory banks differs if the word is unaligned. This is taken care of. For example if 2 bytes are read for the first memory word and 2 bytes are read from the following memory word, the concatenation of the output will be {bank

1, bank 0, bank 3, bank 2}. Our memory design supports all kinds of unaligned word fetching, not just on half-word boundaries.

As for unaligned word writes, the address is handled in the same way mentioned above. The multiplexers before the memory banks control the DATA IN for each bank. If the word is aligned, byte 0 will get into bank 0, byte 1 into bank 1, byte 2 into bank 2 and byte 3 into bank 3. However, if the word is half-word aligned, byte 0 will go into bank 2, byte 1 will go into bank 3, byte 2 will go into bank 0 and byte 3 will go into bank 1. Again, our memory design takes care of all kinds of unalignment, not just on half-word boundaries and not just in store word, but in store half- word as well.

● Design Decisions and Debates:

We were debating 2 design decisions.

The first option is: The memory can either support unaligned words reads or writes and then we wouldn't need to stall the pipeline if we are fetching an unaligned instruction word.

The second option is: The memory won't support unaligned word reads and if we face the case when we were to read unaligned instruction word, we would flush the pipeline.

We chose the second option. Thus, our memory support unaligned word reads and writes.

● Design Rationale:

We chose this design because in the RISC-V manual it was stated that "unaligned memory reads and writes are supported but not recommended." This is why we supported them to exactly mimic what is in the manual. Moreover, by this design, we eliminate the need to support exceptions caused by reading or writing from an unaligned memory location. The support of these exceptions is not required in the scope of our project, and by our design, these exceptions are non-existent as our memory design is flexible.

● Ports:

Port Name	Type	Function
clk	Input	clock to the module
rst	Input	reset to the module
memp_op	Input	control signal to the memory that dictates the memory operation. 00: read 01: sh

		10: sb 11: sw
dataIn	Input	Input, Data to be stored in the memory location
addr	Input	address for the memory location for reads and writes
dataOut	Output	data read from the memory

3.4.Register File:

● Module Name:

registerFile.v

● How It Works:

The register file has as an input rs1 address, rs2 address, rd address, input data, RF write (control signal) and clock

The register file outputs are rs1 data out and rs2 data out.

The register file reads the contents of rs1 and rs2 using rs1 address and rs2 address, respectively and puts the outputs in rs1 data out and rs2 data out, respectively. It also, write input data into rd address when RF write is 1.

The register file's interface acts as a triple ported memory although in implementation, it is a dual ported memory. This is handled inside the register file module itself.

The first port of the dual ported memory will be used as a read only for the sake of our design, so the write signal will always be 0, the dataIn will always be 0, the address will always be rs1 address and the dataOut will always be rs1 data out.

As for the other port, the dataOut will always be rs2 data out. The address will shuffle between rs2 address and rd address depending on whether we are in the write back stage or decode stage. This is controlled by the alternating signal (the clk divided by half). The write signal will shuffle between reading (in the decode stage) and the RF write control signal, again depending on the stage, so this is also controlled by the alternating signal. This is clarified in the diagram below.

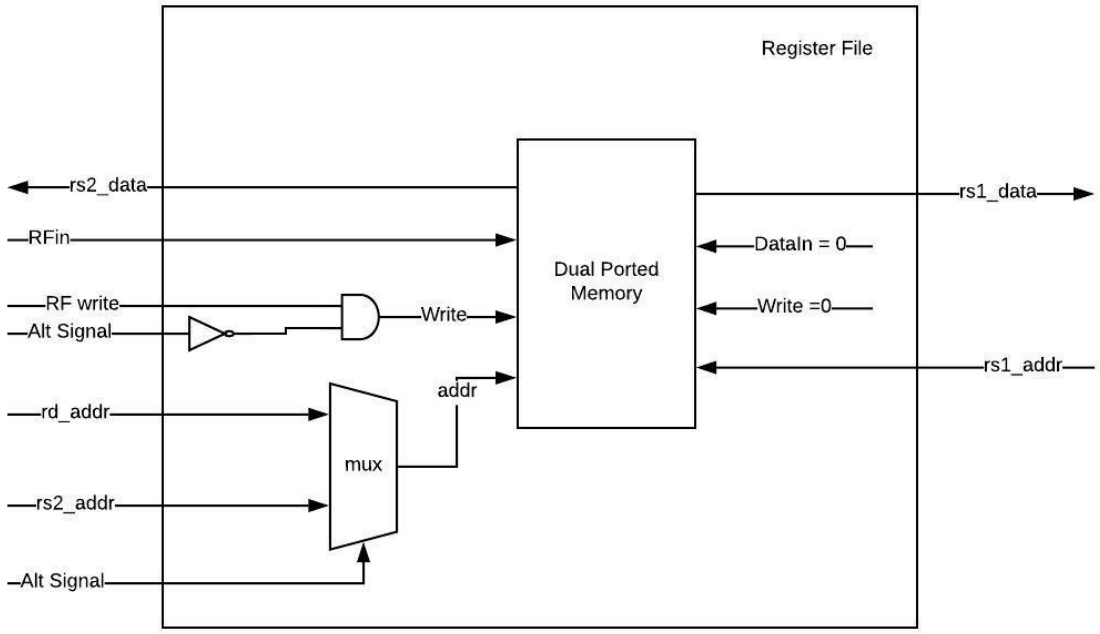

Figure 11: Block diagram for the Register File

● Design Decisions and Debates:

We debated whether the register file's interface should be that of the dual ported memory and the multiplexers and the combinational logic needed to make it function as the register file that we are familiar with should be implemented in Hunter (the datapath module). However, at the end, we agreed that the register file interface will be the one we are used to and all the combinational logic needed to implement it as a dual ported memory and not triple ported memory will be handled inside the module (as shown in the diagram above) for simplicity in the Hunter module.

● Design Rationale:

We believe that this design makes the idea simpler and more understandable. This makes the code more readable in Hunter module and simplified matter for the team member who was responsible for writing the Hunter module.

● Ports:

Port Name	Type	Function
clk	Input	clock signal

rst	Input	reset signal
rs1_addr	Input	address of rs1
rs2_addr	Input	address of rs2
rd_addr	Input	address of rd
rfWrite	Input	control signal (1 if write)
altSignal	Input	input clock divided by 2. Lasts for 2 normal clock cycles. Determines the stage we are in.
rfIN	Input	data to write in rd address if rfWrite = 1
rs1_data	Output	data read from rs1
rs2_data	Output	data read from rs2

● Area and Delay:

WireLoad: none

Gates: 5425 (29.5 %)

Cap: 32.0 ff (0.9 %)

Area: 144207.00 (96.5 %)

Delay: 1333.76 ps (24.0 %)

3.5.CSR:

● Module Name:

CSR.v

● How It Works:

The inputs clk, rst, CSR_new, write and addr, as well as the output CSR, make this module behave pretty much like a single ported memory. If write is 1, CSR_new is written to the CSR with the corresponding address. Else, CSR will be the read data from the CSR with the corresponding address.

However, the other inputs and outputs allow certain CSRs to be written to and read from without the need for an instruction. Among these are: mepc which we need to write to whenever interrupt is 1, mip which we need to update whenever pending is 1, mie which we need to update whenever we jump into a handler or return back from a handler (to enable and disable other interrupts).

Moreover, CSR has a set of counters which makes it different from the register file as we know it, in the sense that it is not implemented as a set of registers or a memory block. minstret, mcycle, mtime all needed to be implemented as counters.

The other CSR registers are not all of the same size, as with the case of regular register files or memories. mepc and mtimecmp are 32 bits, while mie is 4 bits and mip is 3 bits.

● Design Decisions and Debates:

We had a debate whether we should have a module including the CSRs like the register file or should we just implement them in the datapath. We chose to place them in a separate module as it is more organized although this lead to having several input and output ports (unlike the register file module which we wanted to mimic).

● Design Rationale:

We chose to implement the CSRs in a separate module, firstly: to make the changes in the datapath module simpler and minimal. Secondly: to make the testing easier by focusing the testing on one entity/ module. Thirdly, to make the code in the datapath more readable.

● Ports:

Port Name	Type	Function

clk	input	clock signal.
altSignal	input	clock divided by 2. It alternates every 2 clock cycles / every stage. Used to control the count of minstret
rst	input	reset signal
interrupt	input	a pulse that is generated when interrupt signal is 1
interrupt_EX	input	interrupt pulse from second pipeline register. Used to load mie with mie_in
pc	input	value of pc that should be written to mepc when interrupt is 1
pending	input	1 if an interrupt is pending
addr	input	address of the csr to be written to/read
CSR_new	input	data to be written to a csr
write	input	1 any csrw instruction (will write to a csr)
notServiced	input	it is 1 until service is done.
mie_in	input	value with which we want to update mie. It is 1 bit (global bit only)
mret	input	1 if mret is 1. Used to update mie
CSR	output	data read for the CSR whose address was provided
mip_out	output	output of mip. Used to indicate which interrupts are pending
mie_out	output	output of mie. Used to indicate which interrupts are enabled and which are disabled
timer_int	Output	1 when there is a timer interrupt
mepc_out	Output	output of mepc. Used when mret = 1 as pc gets loaded with the value of mepc

- Area and Delay:

WireLoad: none

Gates: 1544 (30.2 %)

Cap: 26.4 ff (0.5 %)

Area: 40207.00 (96.5 %)

Delay: 3803.69 ps (4.3 %)

3.6.ALU:

● Module Name:

alu.v

● How It Works:

The ALU (Arithmetic and Logic Unit) is a module that takes in two operands A and B, and performs an operation on them.

The type of operation that is performed is decided by the signal *op*.

It also outputs several flags that indicate whether:

1- The output is zero (zFlag)

2- An overflow has occurred (vFlag)

` 3- There is a carry out (cFlag)

● Design Decisions, Debates & Rationale:

At first, we planned to implement all the functions of the ALU structurally in Verilog from scratch. However, after we discovered that the softwares optimization of our design will most likely be better than our own structural implementation, we decided to implement most of the functions behaviorally. The only function that was implemented structurally was addition, and subsequently subtraction, because in the logic for the Overflow Flag we needed the intermediary carry outs of the addition operation so it was easier to use our already implemented ripple carry adder from the lab.

● Ports:

Port Name	Type	Function
A	Input	ALU operand 1 data
B	Input	ALU operand 2 data
op	Input	indicates what operation the ALU wi execute
out	Output	output of the ALU operation
zFlag	Output	zero flag, indicates if output is zero

cFlag	Output	carry out flag, indicates if there is a carry ou
vFlag	Output	overflow flag, indicates if operation result in overflow

● Area and Delay:

WireLoad: none

Gates: 1637 (16.7 %)

Cap: 35.1 ff (1.5 %)

Area: 42440.00 (95.1 %)

Delay: 3967.08 ps (4.0 %)

3.7.Internal Program Interrupt Controller:

● Module Name:

CPU_PIC.v

● How It Works:

This module is responsible for enabling and disabling the maskable interrupts (timer, ecall, external interrupts), as well as prioritising the interrupts as follows:

○ NMI
○ EBREAK
○ Timer interrupt
○ External interrupt
○ ECALL

This module produces an interrupt signal that remains high until an MRET instruction is found in the handler. The module outputs the appropriate handler address according to the aforementioned priorities, which is the address the PC jumps to.

● Design Decisions and Debates:

We initially handled detecting the end of a handler differently, which was done by comparing the last handler address to the PC. After we realised that this would produce inaccuracies in some applications, we changed the design to check for the MRET instruction within the handler. We also didn't know if we should include the NMI and the ebreak in the pending signal, yet we chose not to as we do not support nesting, and a NMI or a EBREAK cannot be handled later, as this would be logically incorrect as they cannot be disabled.

● Design Rationale:

We decided to use if-else statements to avoid latches, and it is a simple and efficient way to enforce the priorities required. The enable bits (both global and interrupt-specific) are anded with their corresponding input signal to produce a new signal that is only high when a non-disabled interrupt is incoming. the highest priority is checking for an MRET instruction as this indicates that we are exiting the handler, and that the interrupt has been serviced. The other statements are done in order of priority. The external handlers require

us to check on the hander number to jump to the correct address, which is done in a case statement.

● Ports:

Port Name	Type	Function
interrupt	Output	is high in case there is an enabled interrupt, low if there is none, or if we have returned from the handler.
HA	Output	The handler address that the PC needs to go to to handle the incoming interrupt
pending	Output	A signal to identify if there are any other interrupts “waiting”. It is the OR of the maskable interrupts.
mip	Output	the concatenation of the input maskable interrupts. This is needed in the mip register.
ebreak	Input	Ebreak signal Outputted from the handlerEndHandler module after being extended.
ecall	Input	Ecall signal Outputted from the handlerEndHandler module after being extended.
nmi	Input	Non Maskable input. a signal that is external to the CPU.for example, could be a thermometer signal that indicates overheating.
timer	Input	Outputted from the CSR module. Already in second pipeline stage.
INT	Input	Signal is high when any of the 8-bits of IRQ are high, otherwise is low.
INT_NUM	Input	The number external interrupt requestor, taking priority into account.
enables	Input	4 bit mie_out value representing the output of mie coming out of the CSR

		module.
mret	Input	pulsed signal of the mret coming out of the Control Unit.

● Area and Delay:

Gates:	27 (14.8 %)
Cap:	21.9 ff (1.9 %)
Area:	760.00 (96.3 %)
Delay:	379.09 ps (22.2 %)

3.8.External Program Interrupt Controller:

● Module Name:

PIC

● How It Works:

A module that handles the external interrupt requests. Takes the interrupt signals as an 8-bit input and outputs a one bit interrupt signal as an output, along with the corresponding interrupt number according to the required priorities.

● Design Rationale:

The module was done in if else statements to handle the priorities. the if there are no interrupts then the interrupt signal is low and the number is a “don’t care”.

● Ports:

Port Name	Type	Function
INT	output	Signal is high when any of the 8-bits of IRQ are high, otherwise is low.
INT_NUM	output	The number external interrupt requestor, taking priority into account.
IRQ	input	An 8-bit value with each bit representing an external “device”.

● Area and Delay:

Gates:	19 (26.3 %)
Cap:	1.9 ff (5.3 %)
Area:	474.00 (89.5 %)
Delay:	280.43 ps (47.4 %)

3.9.Forwarding Unit:

● Module Name:

forwardingUnit.v

● How It Works:

This unit handles the forwarding that might need to happen if two consecutive instruction write then read from the same register. The first instruction will not have finished writing back to the register file and therefore we need to forward its value to the following instruction for it to be able to use the correct value.

● Design Decisions, Debates and Rationale:

The conditions this module checks for are that the source register 1 or source register 2 in the current instruction versus the destination register in the previous instruction. If they are equal and if the instruction should write to the register file (indicated by MEM_WB_WRITE) and also that the instruction isn't writing to x0. Then in that case we forward the appropriate value to either operand 1 or 2 or both.

● Ports:

Port Name	Type	Function
MEM_WB_RD	Input	destination register address from the previous instruction
ID_EX_RS1	Input	source register 1 address from the current instruction
ID_EX_RS2	Input	source register 2 address from the current instruction
MEM_WB_WRITE	Input	signal that indicates if the register file should be written to by previous instruction
fwdA	Output	signal that indicates if source register 1 needs to be forwarded
fwdB	Output	signal that indicates if source register 2 needs to be forwarded

● Area and Delay:

Wire Load: none

Gates: 23 (4.3 %)

Cap: 27.5 ff (0.0 %)

Area: 925.00 (100.0 %)

Delay: 304.48 ps (26.1 %)

3.10.CSR Forwarding Unit:

● Module Name:

csrFwdUnit.v

● How It Works:

Handles the forwarding of a CSR value in the case of two successive CSR instructions that use the same CS register.

If the address of the previous instructions CSR is equal to that of the current instruction, then forward the value of the previous instruction and use it instead of the value currently read from the CSR.

● Design Decisions, Debates and Rationale:

It was a fairly simple module that checked for the above mentioned situation, that the addresses of any two consecutive CSR instructions are equal, and then provides the the control signal *fwdCSR* that indicates whether we will use the currently read value or the one from the preceding instruction that hasn't been written yet.

● Ports:

Port Name	Type	Function
csrAddr_EX	Input	The CSR address from execution stage. It is used to check if forwarding is needed.
csrAddr_WB	Input	The CSR address from the writeback stage. It is used to check if forwarding is needed.
fwdCSR	Output	1 if forwardin is needed.

● Area and Delay:

Wire Load: none

Gates: 34 (23.5 %)

Cap: 24.2 ff (6.6 %)

Area: 1072.00 (85.3 %)

Delay: 332.95 ps (26.5 %)

3.11.Branch Control Unit:

● Module Name:

branchControlUnit.v

● How It Works:

The module has several inputs like the zero flag, overflow flag, cout and most significant bit of the ALU result. Another input is function 3 of the instruction word. Function 3 differentiates between the different branch instructions. The other inputs determine if branch should be taken or not according to branch instruction.

The output is a single bit that is 0 if branch should not be taken and is 1 if branch should be taken.

● Design Decisions:

The module is implemented as a case statement to check if function matches which branch instruction, then the output, branch, gets 0 or 1 according to the conditions.

● Design Rationale:

Case statement was used for the simplicity of the code over structural coding and also because the modern compilers can make a better optimization for the hardware than the structural code we would implement.

● Ports:

Port Name	Type	Function
Zflag	Input	zero flag from ALU (1 if result is 0). To compute if branch should be taken.
OVF	Input	overflow flag from ALU (1 if overflow). To compute if branch should be taken.
ALU_MSB	Input	the most significant bit of the ALU result. To compute if branch should be taken.
ALU_COUT	Input	the carry out of the ALU result. To compute if branch should be taken.
func3	Input	function 3 of the instruction word. To differentiate between the different branch instructions

branch	Output	1 if branch should be taken

● Area and Delay:

Wire Load: none

Gates: 18 (16.7 %)

Cap: 24.3 ff (2.8 %)

Area: 494.00 (94.4 %)

Delay: 322.93 ps (44.4 %)

3.12.Keep Interrupt High Till Serviced:

● Module Name:

handlerEndHandler.v

● How It Works:

This module handles the specifics of the interrupt signals. It extends the signal until it is served. This finite state machine module takes the interrupts as inputs, along with the PC, mret, clock and reset and produces a signal that remains high while serving.

● Design Decisions and Debates:

This module was designed as a Finite State Machine (FSM) as it was simple to implement and fit the type of issue we wanted to solve. We initially designed this module to check the last PC address before we exit the handler, yet we we came to know that this is conceptually false as this could cause issues when jumping from within the handler to an address with another ebreak, ecall, or timer interrupt. We worked around the issue by performing the a check on the initial address and knowing that we have handled the interrupt through checking the MRET instruction.

● Design Rationale:

The module is designed as a Finite State Machine, with three states. State A is the initial state, we move to state B when we encounter an ebreak, ecall, or timer interrupt in their respective FSM's. We then move to state C when we enter their handlers. The handler for ebreak, ecall, and timer are checked separately. This is achieved by comparing the PC address with the known handler address. We finally return to state A if we find an MRET instruction, which is an indication that the interrupt is handled. If we are in state B or C in any of the three interrupts (ebreak, ecall, and timer) the output signals remain high.

● Ports:

Port Name	Type	Function

ecallSit	output	Signal remains high until an ecall interrupt is completely serviced, then returns to zero.
ebreakStay	output	Signal remains high until an ebreak interrupt is completely serviced, then returns to zero.
timerFetch	output	Signal remains high until a timer interrupt is completely serviced, then returns to zero.
clk	input	The CPU clock
rst	input	Reset signal
mret	input	Signal coming from the EX stage after being propagated. Outputted from the Control Unit.
ecall	input	Signal coming from the EX stage after being propagated. Outputted from the Control Unit.
ebreak	input	Signal coming from the EX stage after being propagated. Outputted from the Control Unit.
timer	input	Outputted from the CSR module. Already in second pipeline stage.
pc	input	The CPU's program counter.

● Area and Delay:

Wire load: None
Gates: 58 (10.3 %)
Cap: 18.4 ff (2.2 %)
Area: 1595.00 (94.8 %)
Delay: 534.83 ps (24.1 %)

3.13.Control Unit:

● Module Name:

controlUnit.v

● How It Works:

The control unit issues many control signals that organize how the CPU functions. The appropriate fields from the current instruction are taken as an input, which are the Opcode, function 3 , 7, and 12 fields.

● Design Decisions and Debates:

We were considering further optimizing the control unit, yet this was not feasible given the timeframe we had. This module was continuously changed throughout the different milestones.

● Design Rationale:

Multiple case statements and if conditions are used, as almost every instruction has a unique combination of control signals.

● Ports Discussed in Section 4.
● Area and Delay:

Gates:	107 (14.0 %)
Cap:	29.7 ff (2.8 %)
Area:	3027.00 (93.5 %)
Delay:	658.37 ps (30.8 %)

4. Control Signals:

Inst	opcode	funct3	funct7	memWrite	rfWrite	aluSrcA	aluSrcB	pcSrc	rfWriteData	ALU Operation	aluOp	memOp	pcWrite	bInst	ebreak	ecall	mret	csrImmInst	csrInstType	csrWrite
LUI	0110111	xxx	x	0	1	x	1	0	0111	lui	1000	00	0	0	0	0	0	0	xx	0
AUIPC	0010111	xxx	x	0	1	0	1	0	0111	add	0000	00	0	0	0	0	0	0	xx	0
JAL	1101111	xxx	x	0	1	0	1	1	0000	add	0000	00	1	0	0	0	0	0	xx	0
JALR	1100111	000	x	0	1	1	1	1	0000	add	0000	00	1	0	0	0	0	0	xx	0
BEQ	1100011	000	x	0	0	1	0	0	xxxx	sub	0001	00	0	1	0	0	0	0	xx	0
BNE	1100011	001	x	0	0	1	0	0	xxxx	sub	0001	00	0	1	0	0	0	0	xx	0
BLT	1100011	100	x	0	0	1	0	0	xxxx	sub	0001	00	0	1	0	0	0	0	xx	0
BGE	1100011	101	x	0	0	1	0	0	xxxx	sub	0001	00	0	1	0	0	0	0	xx	0
BLTU	1100011	110	x	0	0	1	0	0	xxxx	sub	0001	00	0	1	0	0	0	0	xx	0
BGEU	1100011	111	x	0	0	1	0	0	xxxx	sub	0001	00	0	1	0	0	0	0	xx	0
LB	0000011	000	x	0	1	1	1	0	0100	add	0000	00	0	0	0	0	0	0	xx	0
LH	0000011	001	x	0	1	1	1	0	0010	add	0000	00	0	0	0	0	0	0	xx	0
LW	0000011	010	x	0	1	1	1	0	0001	add	0000	00	0	0	0	0	0	0	xx	0
LBU	0000011	100	x	0	1	1	1	0	0101	add	0000	00	0	0	0	0	0	0	xx	0
LHU	0000011	101	x	0	1	1	1	0	0011	add	0000	00	0	0	0	0	0	0	xx	0
SB	0100011	000	x	1	0	1	1	0	xxxx	add	0000	10	0	0	0	0	0	0	xx	0
SH	0100011	001	x	1	0	1	1	0	xxxx	add	0000	01	0	0	0	0	0	0	xx	0
SW	0100011	010	x	1	0	1	1	0	xxxx	add	0000	11	0	0	0	0	0	0	xx	0
ADDI	0010011	000	x	0	1	1	1	0	0111	add	0000	00	0	0	0	0	0	0	xx	0
SLTI	0010011	010	x	0	1	1	1	0	0110	sub	0001	00	0	0	0	0	0	0	xx	0
SLTIU	0010011	011	x	0	1	1	1	0	1000	sub	0001	00	0	0	0	0	0	0	xx	0
XORI	0010011	100	x	0	1	1	1	0	0111	xor	0100	00	0	0	0	0	0	0	xx	0
ORI	0010011	110	x	0	1	1	1	0	0111	or	0011	00	0	0	0	0	0	0	xx	0
ANDI	0010011	111	x	0	1	1	1	0	0111	and	0010	00	0	0	0	0	0	0	xx	0
SLLI	0010011	001	0	0	1	1	1	0	0111	shift left	0101	00	0	0	0	0	0	0	xx	0
SRLI	0010011	101	0	0	1	1	1	0	0111	shift right l	0110	00	0	0	0	0	0	0	xx	0
SRAI	0010011	101	1	0	1	1	1	0	0111	shift right a	0111	00	0	0	0	0	0	0	xx	0
ADD	0110011	000	0	0	1	1	0	0	0111	add	0000	00	0	0	0	0	0	0	xx	0
SUB	0110011	000	1	0	1	1	0	0	0111	sub	0001	00	0	0	0	0	0	0	xx	0
SLL	0110011	001	0	0	1	1	0	0	0111	shift left	0101	00	0	0	0	0	0	0	xx	0
SLT	0110011	010	0	0	1	1	0	0	0110	sub	0001	00	0	0	0	0	0	0	xx	0
SLTU	0110011	011	0	0	1	1	0	0	1000	sub	0001	00	0	0	0	0	0	0	xx	0
XOR	0110011	100	0	0	1	1	0	0	0111	xor	0100	00	0	0	0	0	0	0	xx	0
SRL	0110011	101	0	0	1	1	0	0	0111	shift right l	0110	00	0	0	0	0	0	0	xx	0
SRA	0110011	101	1	0	1	1	0	0	0111	shift right a	0111	00	0	0	0	0	0	0	xx	0
OR	0110011	110	0	0	1	1	0	0	0111	or	0011	00	0	0	0	0	0	0	xx	0
AND	0110011	111	0	0	1	1	0	0	0111	and	0010	00	0	0	0	0	0	0	xx	0
Ebreak	1110011	000	0	0	0	0	0	0	0000	add	0000	00	0	0	if f12 == 12'h001	if f12 == 12'h000	if f12 == 12'h302	0	xx	0
Ecall	1110011	000	0	0	0	0	0	0	0000	add	0000	00	0	0	if f12 == 12'h001	if f12 == 12'h000	if f12 == 12'h302	0	xx	0
CSRRWI	1110011	101	x	0	0	0	0	0	1001	add	0000	00	0	0	0	0	0	1	00	1
CSRRSI	1110011	110	x	0	0	0	0	0	1001	add	0000	00	0	0	0	0	0	1	01	1
CSRRCI	1110011	111	x	0	0	0	0	0	1001	add	0000	00	0	0	0	0	0	1	10	1
CSRRW	1110011	001	x	0	0	0	0	0	1001	add	0000	00	0	0	0	0	0	0	00	1
CSRRS	1110011	010	x	0	0	0	0	0	1001	add	0000	00	0	0	0	0	0	0	01	1
CSRRC	1110011	011	x	0	0	0	0	0	1001	add	0000	00	0	0	0	0	0	0	10	1

operation	code
add	0000
sub	0001
and	0010
or	0011
xor	0100
shift left	0101
shift right logical	0110
shift right arithmetic	0111
shift right arithmetic immediate	1001
lui	1000

Control Signal	Function
memWrite	is High in operations that require writing to the memory (SB,SH,SW). This signal is used to control the memory, and is propagated through the first pipeline register.
rfWrite	This signal is used to control writing into the register file. This is propagated through the two pipeline registers before being inputted to the register file. Cases where it is high are listed in the table above.
aluSrcA	Selects between the PC and RS1 values to be inputted into the CPU. low takes the PC while high takes the value in RS1.

aluSrcB	Selects between RS2 (forwarded or unforwarded) and the Immediate value coming out of the Immediate Generator. low takes the value in RS2, while high takes the Immediate.
pcWrite	Controls the value being inputted into the PC MUX. Selects between PC next (low) and the other possible signals (High). It is only high in JAL and JALR instructions. This signal is propagated and taken from the EX stage.
memOp	This signal controls the way the memory is stored. 10 indicates a SB, 01 a SH and 11 a SW. this is used to avoid overwriting needed data.
rfWriteData	This signal is propagated and used to control the large MUX in the last phase. It consists of 4bits to allow us to select between the 10 possible inputs to the multiplexor.
aluOp	As shown in the table above, aluOp encodes the type of operation the ALU needs to execute. The four bits encode the 10 possible combinations.
bInst	This signal is inputted to the branch control unit for processing, and is anded with the branch signal to select between the other possible signals mentioned in the PCWrite. if the output of the anding is 1 the ALU result is taken.
ebreak	This signal is propagated till the EX stage then is used as an input to the “CPU PIC” where it is used to indicate an ebreak interrupt.The signal is high only if funct 12 is 0x001 and the instruction is an ebreak or ecall
ecall	This signal is propagated till the EX stage then is used as an input to the “CPU PIC” where it is used to indicate an ecall interrupt.It is high only if funct 12 is 0x000 and the instruction is an ebreak or ecall

csrImmInst	Used to control the CSR forwarding, and is propagated until the EX stage. Signal is high only in case of an immediate CSR instruction
csrInstType	This signal differentiates between the three different types of CSR instructions (disregarding immediates).
csrWrite	high in all CSR instructions.
mret	Used to generate a pulse and to control the handler end.is high only if funct 12 is 0x302 and the instruction is an ebreak or ecall

5. Tests:

5.1. Steps to run tests:

* In case you will start by writing an assembly program, start by step 1.

* In case you will run our ready made machine code, start by step 5.

* Pay close attention to step 4. Our memory is implemented as little endian and will work correctly if the hexadecimal machine code is inputted in the familiar order with the most significant byte of the word written on the right.

Step 1: Write an assembly program

Step 2: Disassemble the assembly program into machine code using gnu tool chain

Step 3: objcopy the binary machine code into hexadecimal

Step 4: Change the endianity of the hexadecimal machine code. GNU toolchain generate the hexadecimal machine code with the least significant byte on the right and the most significant byte on the left. Flip this. Make the most significant byte on the left and least significant byte on the right

Step 5: Upload the flipped hexadecimal machine to CloudV in the folder hex files

Step 6: Change the name of the file in the readmemh function in the reset block of the module named “memory.v”

Step 7: Start simulating

5.2. Implemented Test Cases:

1) Milestone 2: In order to test our datapath for milestone 3, we tried small testcases that we wrote and disassembled using Oak, and then to have a deeper test, we used testcase1 and testcase2 on blackboard. These can be found in the folder testcases in sub-folder milestone 2.

2) Milestone 3: In order to test our datapath for milestone 3, we tried small programs that as a whole tested each and every compressed instruction. 1

program was a mix of both compressed and uncompressed instructions, and testcase1 and testcase2 on blackboard were tested again to make sure they are still working correctly. These can be found in the folder testcases in sub-folder milestone 3.

3) Milestone 4: In order to test our datapath for milestone 4, we inserted dummy instructions in the interrupt handlers. In each instruction handler there are c.nop's, "li x31, immediate" and mret. x31 is only used by the instruction handler. The instruction handler does not use any registers that the main program uses. Moreover, each instruction handler loads a different immediate in x31 to make sure that the pc jumped to the correct handler.
   The files we used to test the datapath can be found in folder testcases in sub-folder milestone 4. The "handlers.s" includes the reset handler, ecall handler, ebreak handler and timer handler, so it does not change. The other files include IRQ handlers and the main program. The main program part is the only part that changes from 1 file to another. "mix.s" is the mix we made of compressed and uncompressed instructions when testing for milestone 3. "CSRtest2.s" includes ECALLs, EBREAKs, timer interrupts, as well as CSRR instructions. "mini.s" includes a very small program that mainly tests c.beqz, c.bnez, c.sw and c.lw. "testcase1.s" includes the same instructions as the one in testcase1 on blackboard, but the addresses for loads and stores are specially customized to load and store in empty locations away from the interrupt handlers and the main program.

5.3. Tests that Fail:

All the required instructions are supported and all tests work correctly.

6. Challenges and Limitations:

6.1. Challenges:

While working on the project, we were faced with several challenges.

Firstly, for deliverable 2, our memory design took a lot of thinking in order to come up with a correct design that supports unaligned reads and writes.

Secondly, for deliverable 3, thinking of all the different possibilities that can happen when fetching instructions after supporting compressed instructions, as well as, doing changes to the datapath that will make the support of compressed instructions simpler and systematic (whether the instruction is compressed or not) to be able to implement it in hardware.

Thirdly, for deliverable 3, when we had to submit the third deliverable, we had not learnt about gnu toolchain yet. This made us convert compressed assembly programs into binary machine code then into hexadecimal machine code by hand.

Fourthly, for deliverable 4, we needed a module to detect whether a certain interrupt is serviced or not, to be able to keep the interrupt 1 until it service is done. At first, we assumed that we knew the addresses where each interrupt ends, which the address for the mret instruction, and we implemented the module as such. It keeps the specific interrupt 1 until pc hits the address of its mret. Turns out, we cannot use this assumption as we can never be sure where the mret is because the interrupt handler block can include a jump to somewhere outside the handler. This made us implement this module as a finite state machine. More details about this module are in section 3.12.

Finally, for deliverable 4, when we synthesized and implemented our CPU on Vivado, the utilization report produced an error message because our design used up more LUTs than there is on the board. We, then, learnt after deep researching and analyzing that the problem was caused by the memory. This made us remove the memory from the CPU and it is now implemented outside of the CPU in the environment module.

6.2. Limitations:

Our implementation does not support nested interrupt handling. Therefore, if a higher priority interrupt that cannot be disabled (like NMI) became 1 when we are in the

handler of another interrupt, the NMI will have to wait until the service of the handler which the CPU is currently doing, is done, then NMI will be serviced.

We synthesized our datapath and there were no latches.

7. Future work:

If we had more time we would've:

Supported more control and status registers.

Supported nested interrupts.

Support the handling of a lot more exceptions like overflow exception and unknown instruction exception.

Support the memory interface that makes the memory transaction takes 2 phases.

8. Appendix:

Test Case Name	Deliverable No.	Feature									
		R Instr	I Instr	Branch Instr	Memory Instr	JALR	AUIPC	LUI	CSR	Interrupts	Compressed Instr
handlers.s	4		Y			Y	Y		Y	Y	Y
csr2.s	4	Y	Y						Y	Y	Y
mix.s	4	Y	Y			Y			Y		Y
case1_D4.s	4	Y	Y	Y	Y		Y	Y		Y	Y
mini.s	4	Y	Y	Y	Y						Y
compressed1.s	3	Y	Y								Y
compressed2.s	3		Y			Y					Y
compressed3.s	3		Y			Y					Y
compressed4.s	3	Y	Y	Y	Y			Y			Y
compressed5.s	3		Y		Y						Y
compressed6.s	3	Y	Y								Y
case1.s	2	Y	Y	Y	Y	Y	Y	Y			
case2.s	2	Y	Y	Y		Y					