Implementation of 32-bit 5-stage pipelined RISC-V processor

by

Usama Ayub
2018-EE-124

Advisor:
Mr. Umer Shahid

FALL 2021

Department of Electrical Engineering
University of Engineering and Technology, Lahore

ABSTRACT

I have designed and implemented a small RISC-V processor that has a five-stage pipeline with CSR support for interrupts. The processor's instruction memory is written using the Verilog `readmemh` function. The processor supports edge-triggered interrupts with the help of RISC-V CSRs. It supports RISC-V R, I, B, J and S type instructions.

ACKNOWLEDGMENTS

I thank my teacher Prof. Muhammad Tahir, who taught me Computer Architecture and my Computer Architecture Lab instructor Mr. Umer Shahid, who provided all the necessary guidance and resources for this project. I also thank my classmate Salman Jamil, who provided me with much needed guidance for implementing interrupts and the CSR module.

STATEMENT OF ORGINALITY

It is stated that this CEP presented in this course consists of my own ideas and hands-on working. The contributions and ideas from others have been duly acknowledged and cited in the dissertation. This complete report and code are written by me.

Usama Ayub

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

1. INTRODUCTION

The processor was developed in stages. First, the complete datapath was developed and tested thoroughly. Then, the controller was developed and a complete single cycle processor was made. This processor was then tested and implemented on the Nexys A7 board. Then, five pipeline stages were introduced and the pipelined processor was tested with the help of hazard-free code. The hazard controller was then designed and added to the processor and the processor was tested with assembly code that had all three types of data hazards i.e., WAW (Write After Write), WAR (Write After Read) and RAW (Read After Write) hazards and control hazards. Finally, support for interrupts and RISC-V CSRs was added and the complete processor was tested thoroughly.

1.1. Design Description

The processor I have designed and implemented features a five-stage pipeline. It has a hazard controller for handling data and control hazards. The design has no structural hazards and is based on the Harvard architecture. It has a CSR module for handling interrupts, although I have not implemented any privilege levels. The processor supports R, S, B, I and J type RISC-V assembly instructions.

1.2. Potential Hazards

Of the three types of hazards i.e., structural, data and control hazards, there are no structural hazards in my design. The data memory is separate from the instruction memory, so instructions and data can be accessed simultaneously without any structural hazard. However, data and control hazards can arise in the processor and to deal with

these hazards, a hazard control unit has been implemented. This unit is responsible for stalling and clearing certain pipeline stages in the event of a hazard. The complete details about this unit are given in section 2.6

1.3. Datapath Explanation

A description of the complete data path is given below:

1.3.1. Highlighted Datapath for R-type Instructions

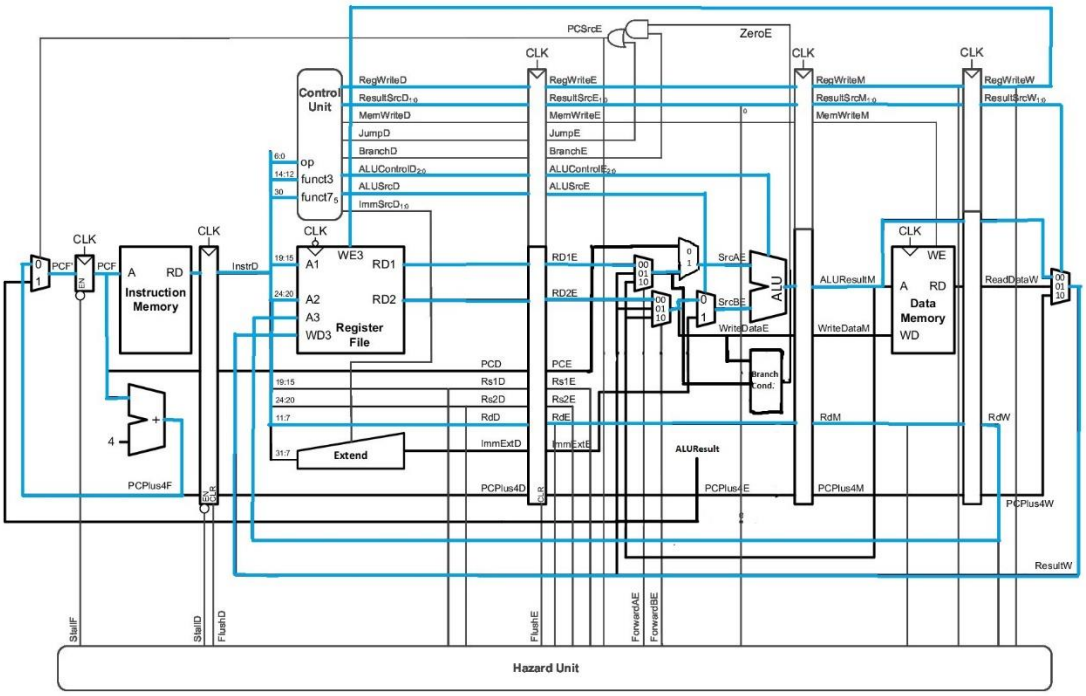


Figure 1:Highlighted datapath for R-type instructions

1.3.2. Highlighted Datapath for I-type Instructions

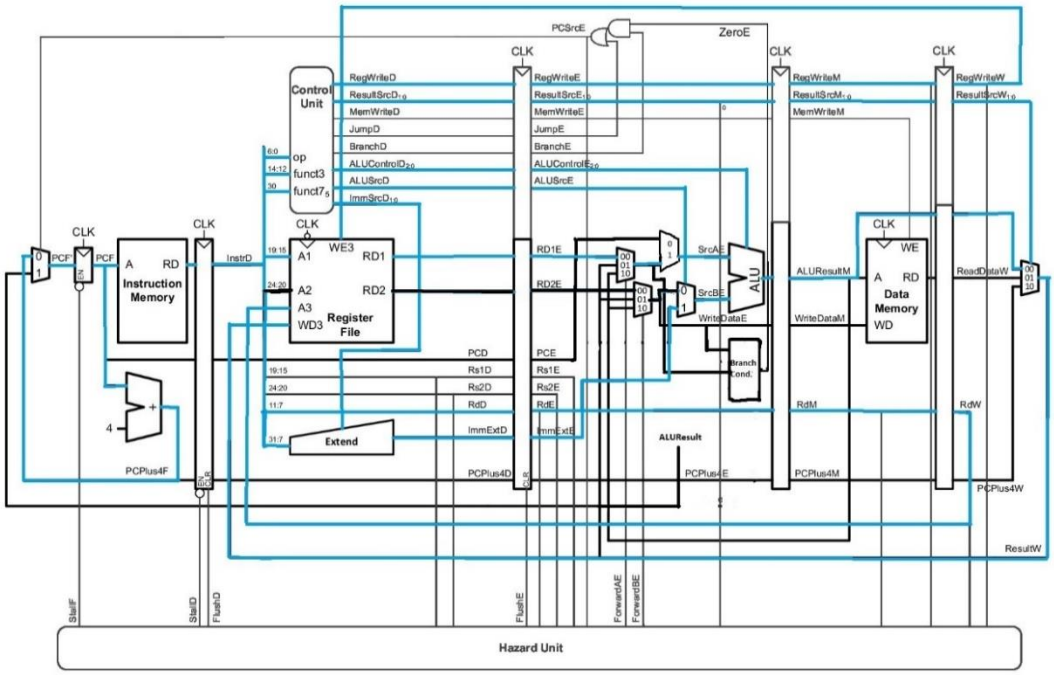


Figure 2:Highlighted datapath for I-type instructions

1.3.3. Highlighted Datapath for S-type Instructions

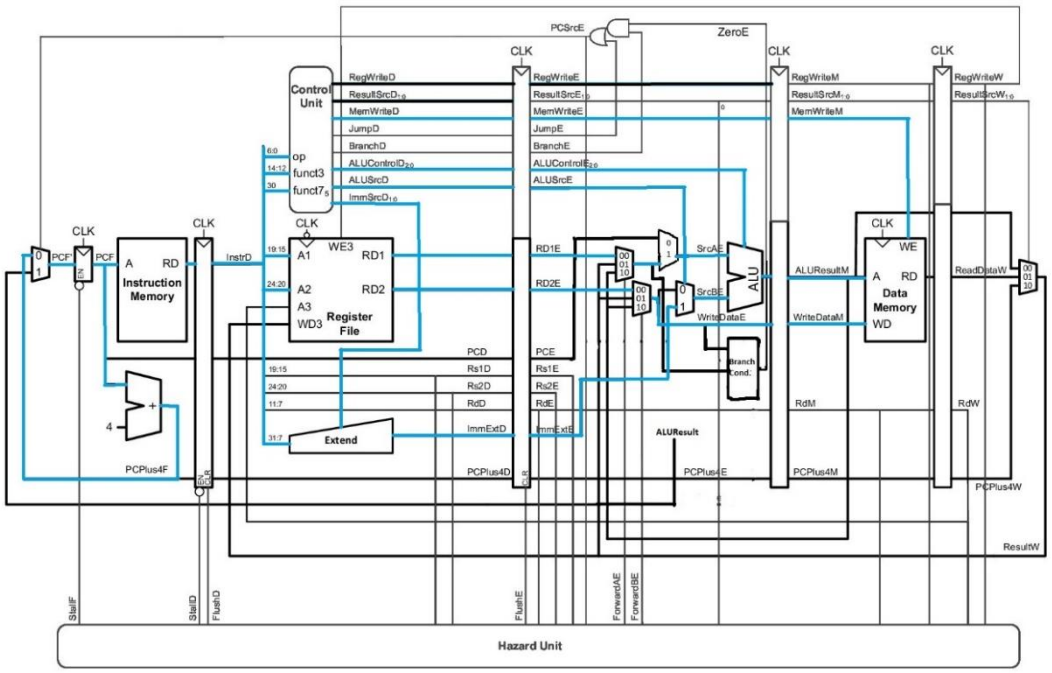


Figure 3:Highlighted datapath for S-type instructions

1.3.4. Highlighted Datapath for B-type Instructions

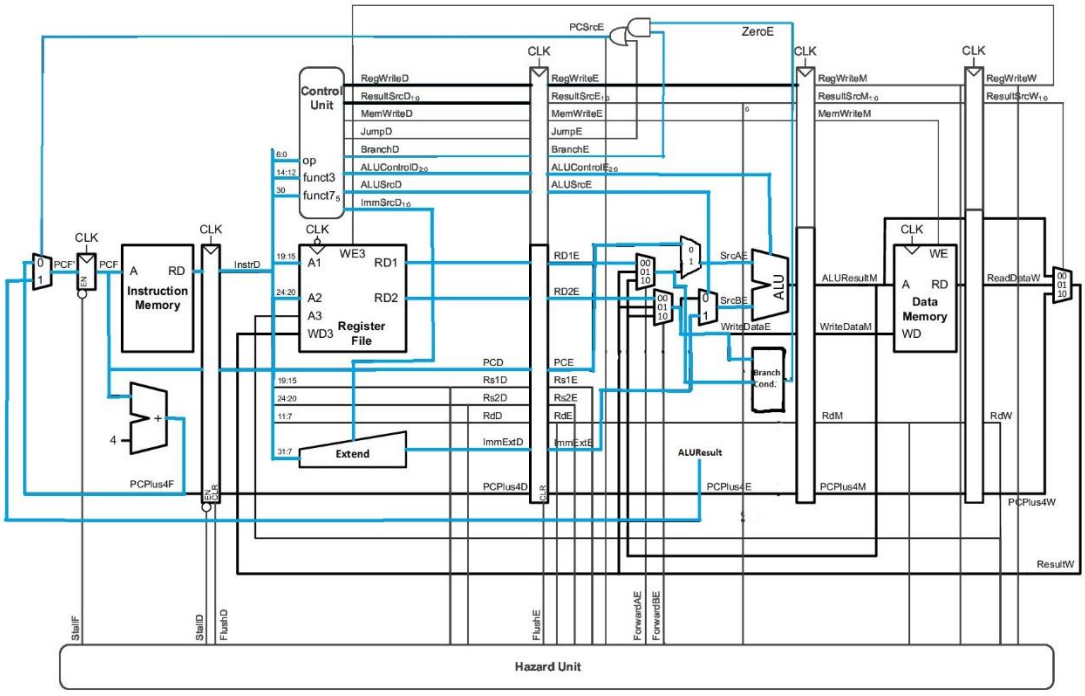


Figure 4:Highlighted datapath for B-type instructions

1.3.5. Highlighted Datapath for J-type Instructions

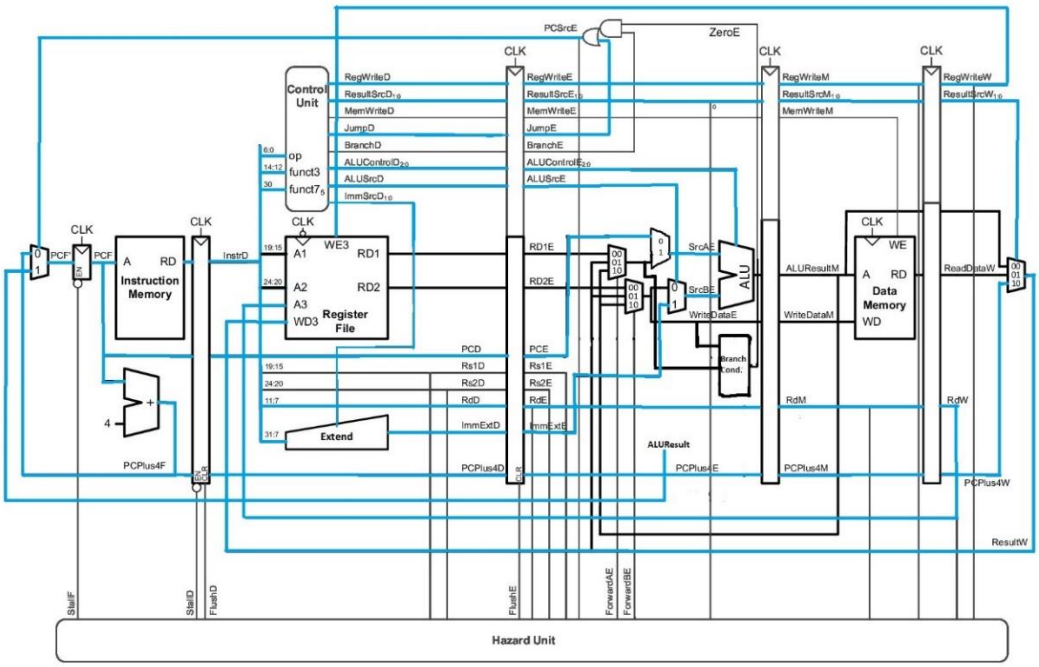


Figure 5:Highlighted datapath for J-type instructions

1.3.6. Highlighted Datapath for Hazards

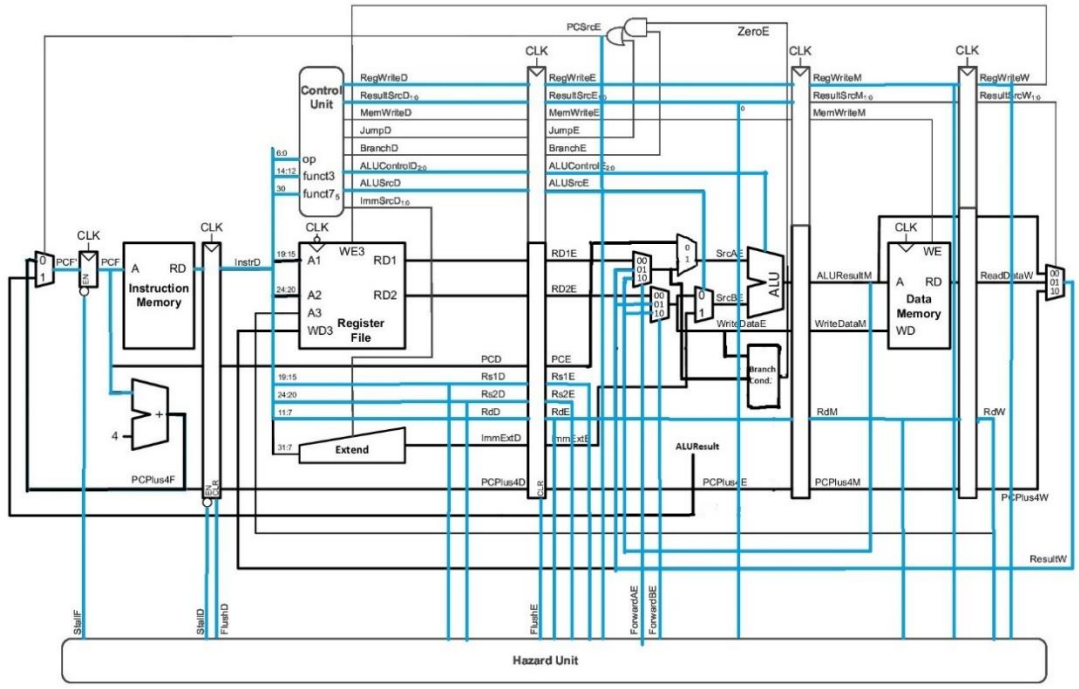


Figure 6: Highlighted datapath for Hazards

1.3.7. Highlighted Datapath for CSR Instructions

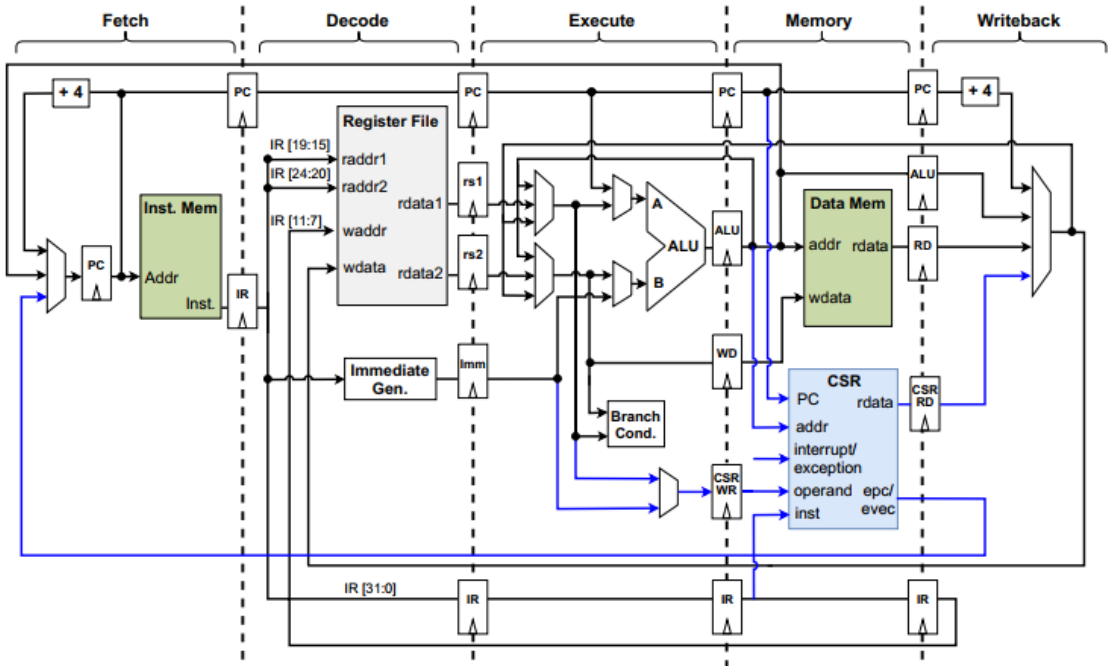


Figure 7: Highlighted datapath for CSR instructions

NOTE:

The branch cond. module is defined as `CompareAndBranch` module in the processor code. This module also receives the `opcode` and `func3` from the decode stage. But for simplicity, these inputs have not been shown in the above datapaths and they have also not been shown in the decode pipeline stage.

2. Modules

A description of the processor RTL is given below:

2.1. The Top module

The top module has seven input ports and three output ports. The input ports include the reset and clock inputs. As the project is used to compute the GCD of two numbers, the user can input two 8-bit unsigned numbers using a 4-bit num input port. In order to input a number, the user will first use a selection input port (`SelNum`) to select either the first or the second number, and then use another selection input port (`SelDigit`) to select whether to input the top or bottom four bits of the number. Once both the inputs are complete, the user will turn ON the done input port and the processor will compute the GCD it will be displayed on two seven segment displays using time multiplexing by using the `Anode_Activate` and `LED_out` output ports. The numbers input by the user will also be displayed on the seven segment displays. The `switch` input port is for interrupts and on every positive edge on this port, the `LED` output is toggled. All of the modules that will be described below have been instantiated in the top module and they have been interconnected along with muxes in this module. The source code for this processor is available here. I have made a separate processor that is a general-purpose RISC-V processor that has only clock, reset and switch (for interrupt) inputs and an LED output. Its source code is available here.

2.2. CPU Control Unit

The Control unit has only three input ports: `opcode, func7` and `func3.` This module generates the control signals for all the other modules. It first determines the type

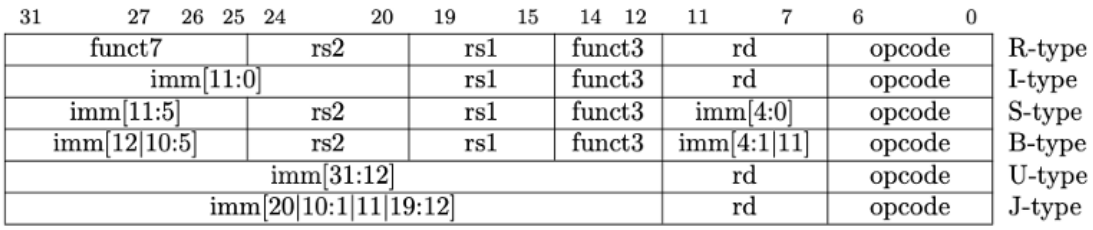


Figure 8: RISC-V instruction encoding formats

of instruction based on the `opcode.` Once the instruction type is determined, all the control signals can be set based on the instruction type except for `ALU_Sel.` For instance, the `RegWrite` signal will be set only for instructions that write to the register file. As can be seen in Figure 8, only the R, I, U and J type instructions write to the register file as these instructions have an rd or destination register. So, the Verilog code for the `RegWrite` signal is as under:

```
assign RegWrite = (instr_type == RTYPE || instr_type ==ITYPE || instr_type == JTYPE ||
instr_type == UTYPE);
```

As for `ALU_Sel,` this output signal decides what operation the ALU will perform. This operation is uniquely determined by `opcode, func7` and `func3.` For instance, add and sub have the same `opcode` and `func3`; and differ only with respect to `func7.`

2.3. Memory Module

The data memory module has a memory of size 2 KB. The memory is word addressable and currently supports only word loads and stores. Memory is read asynchronously. However, writes and reset are both synchronous.

2.4. Register File

The register file consists of 32 registers and the size of each register is 32 bits. The register `x0` is hardwired to zero. This module has two read ports and one write port. Correspondingly, there are two read address ports and one write address port. The read and write ports are 32-bit wide and the address ports are 5-bit wide. The register file read is asynchronous. However, the register file is written at negative edges of the clock and is written only when the `RegWrite` input is HIGH. In addition to these ports, this module also has two more outputs. One is the `GCD` output port of width 8-bits which is hardwired to `x8` as the final GCD computed by the code in the instruction memory is written to `x8` and `x9`. Second is the `LED` output port which is a single bit and is hardwired to the zeroth bit of `x11` as the interrupt service routine toggles the `x11` register every time it is executed. Thus, the state of LED on the board is toggled at every positive edge of the switch.

2.5. ALU

The ALU module performs various arithmetic and logic operations. This module has two input ports and one output port. All of these ports are 32-bit wide. The `ALU_Sel` input provided by the Control Unit decides what operation the ALU will perform.

2.6. Hazard Controller

This module is responsible for detecting and resolving hazards.

2.6.1. Solving Data Hazards

Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction because its result can then be forwarded to the Execute stage of the next instruction. Forwarding has been explained in section 2.7. However, the `lw` instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the `lw` instruction has a two-cycle latency because a dependent instruction cannot use its result until two cycles later. A solution is to stall the pipeline, holding up operation until the data is available. Figure 9 shows stalling the dependent instruction (`and`) in the Decode stage. `and` enters the Decode stage in cycle 3 and stalls there through cycle 4. The subsequent instruction (`or`) must remain in the Fetch stage during both cycles as well because the Decode stage is full. In cycle 5, the result can be forwarded from the Writeback stage of `lw` to the Execute stage of `and`. Also, in cycle 5, source `s7` of the `or` instruction is read directly from the register file, with no need for forwarding. Note that

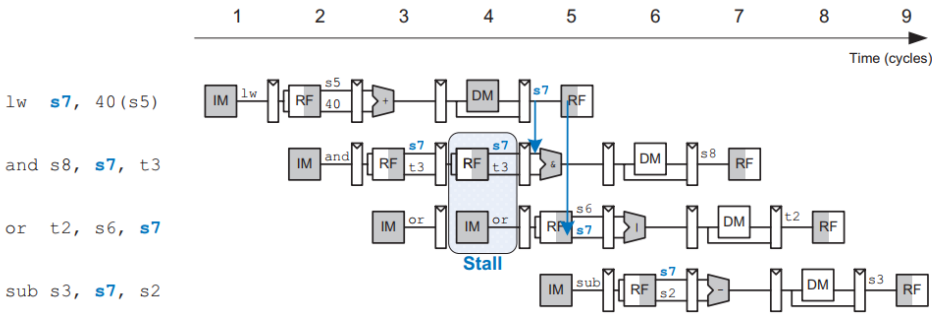


Figure 9: Abstract pipeline diagram illustrating stall to solve hazards

the Execute stage is unused in cycle 4. Likewise, Memory is unused in cycle 5 and Writeback is unused in cycle 6. This unused stage propagating through the pipeline is called a bubble, which behaves like a `nop` instruction. The bubble is introduced by zeroing out the Execute stage control signals during a Decode stage stall so that the bubble performs no action and changes no architectural state.

In order for the Hazard Unit to stall the pipeline, the following conditions must be met:

1. A load instruction is in the Execute stage.
2. The load's destination register matches the source operands of the instruction in the Decode stage.

Stalls are supported by adding enable inputs (`EN`) to the Fetch and Decode pipeline registers and a synchronous reset/clear (`CLR`) input to the Execute pipeline register. When a load word (`lw`) stall occurs, `StallD` and `StallF` are asserted to force the Decode and Fetch stage pipeline registers to retain their existing values. `FlushE` is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble. The Hazard Unit `lwStall` (load word stall) signal indicates when the pipeline should be stalled due to a load word dependency. Whenever `lwStall` is TRUE, all of the stall and flush signals are asserted. Hence, the logic to compute the stalls and flushes is:

$$lwStall = ResultSrcE_0 \;\&\; ((Rs1D == RdE) \mid (Rs2D == RdE))$$

$$StallF = StallD = FlushE = lwStall$$

2.6.2. Solving Control Hazards

For the case of branches, the processor will always predict that the branch will not be taken and continues executing the subsequent instructions. If the branch should have

been taken, then the two instructions following the branch must be flushed (discarded) by clearing the pipeline registers for those instructions. An example is shown in illustrated in Figure 10.

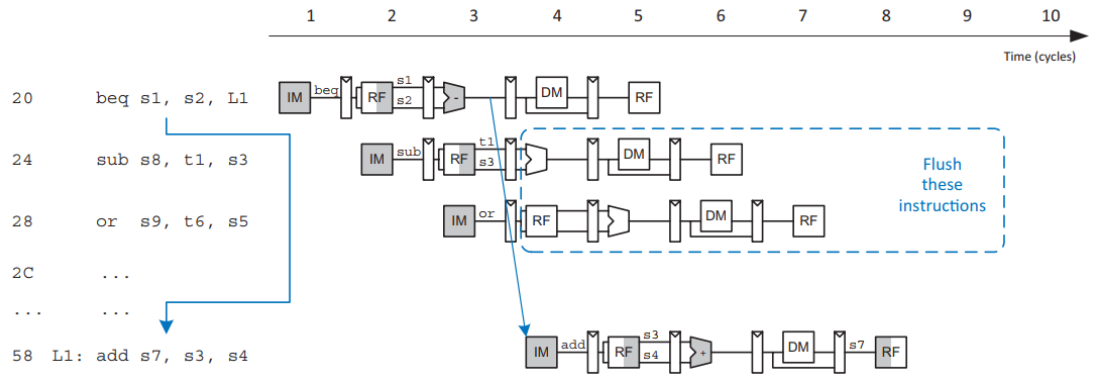


Figure 10: Abstract pipeline diagram illustrating flushing when a branch is taken

When a branch is taken, the subsequent two instructions must be flushed from the pipeline registers of the Decode and Execute stages. Thus, we add a synchronous clear input (`CLR`) to the Decode pipeline register and add the `FlushD` output to the Hazard Unit. (When `CLR` = 1, the register contents are cleared, that is, become 0.) When a branch is taken (indicated by `PCSrcE` being 1), `FlushD` and `FlushE` must be asserted to flush the Decode and Execute pipeline registers. The flushes are now calculated as:

$$FlushD = PCSrcE$$

$$FlushE = lwStall \mid PCSrcE$$

2.7. Forwarding Unit

This unit resolves some data hazards by forwarding or bypassing a result from the Memory or Writeback stage to the Execute stage for an instruction that is dependent on that result. For this purpose, multiplexers are added in front of the ALU to select its operands from the register file or the Memory or Writeback stage. The control signals for

the forwarding multiplexers are computed by the Hazard Unit to choose operands from the register file or from the results in the Memory or Writeback stage. The hazard detection unit receives the two source registers from the instruction in the Execute stage, and the destination registers from the instructions in the Memory and Writeback stages. It also receives the `RegWrite` signals from the Memory and Writeback stages to know whether the destination register will actually be written (e.g., the `sw` and `beq` instructions do not write results to the register file and, hence, do not have their results forwarded). Thus, the forwarding logic for the A input of the ALU is given below:

if ((*Rs1E* == *RdM*) & *RegWriteM*) & (*Rs1E* != 0) then **// Forward from Memory stage**

ForwardAE = 10

else if ((*Rs1E* == *RdW*) & *RegWriteW*) & (*Rs1E* != 0) then **// Forward from Writeback stage**

ForwardAE = 01

else *ForwardAE* = 00 **// No forwarding (use RF output)**

The forwarding logic for the B input of ALU is identical to the above. [1]

2.8. CSR Controller

The CSR module is fairly simple. It has been placed in the memory stage in the pipeline. I have not implemented any privilege levels. This module receives an interrupt signal and then sends appropriate signals to the necessary modules so that the ISR's execution can start and it also deals with returning from the ISR to the normal execution. This module is explained in more detail in the next section.

2.9. CSR Control Signals

The input signals for the CSR module are `interrupt`, `mret` and `clock`. This module has to generate an interrupt pending signal that is used by the hazard controller. The interrupt pending signal goes HIGH at the positive edge of the interrupt (for positive edge triggered interrupt) and it goes LOW at the next positive clock edge as shown below:

```
always @(posedge interrupt)
    interrupt_pending = 1;
always @(posedge clock)
    interrupt_pending = 0;
```

The problem with using two always blocks for this purpose is that it will cause a multi-driven net error during implementation. So, for this purpose, I found a Verilog code on ChipVerify [2] that does exactly this job and is synthesizable.

Three tasks are performed whenever an interrupt arrives:

1. Interrupt pending signal goes HIGH
2. The `PC` in the Execute stage of the pipeline is saved as the return address after the ISR is executed
3. The `ePC` is set to the ISR address.

The `mret` signal is provided by the Control Unit whenever it encounters an `mret` instruction. When this flag goes HIGH, the CSR module will set `ePC` to the return address saved in step 2 and the priority mux must also be enabled. So, the logic for the enable input for the priority mux is as:

```
assign interrupt_PC_en = mret | interrupt_pending;
```

Thus, the three outputs of this module are `int_pending`, `int_pc_en` and `ePC`.

3. Tests

In this chapter, I will present the method I used for verification of my processor.

3.1. Simulation Tool Description

I have used the simulator provided by Xilinx Vivado for simulating and testing my processor at each stage of development.

3.2. Steps to run a basic modular test

The following steps were taken to test the working of my processor:

1. First, I wrote a comprehensive assembly code for the instruction memory for testing the processor. This code has all the types of hazards and covers all of the instructions that the processor supports.
2. This code was then simulated using some RISC-V simulation tool and the output was saved.
3. The code was then fed to the processor and its output was compared with the one obtained in step 2.

3.3. Test performed

One test I performed for verification of my processor's working is the `riscvtest.s` at page 464 of the book Digital Design and Computer Architecture RISC-V Edition. [1] This test is given in the Appendix. After running this test, it did write the value 25 to address 100. However, as my data memory is word addressable at the time, it wrote the value 25 or 0x19 to the address 25 as shown below:

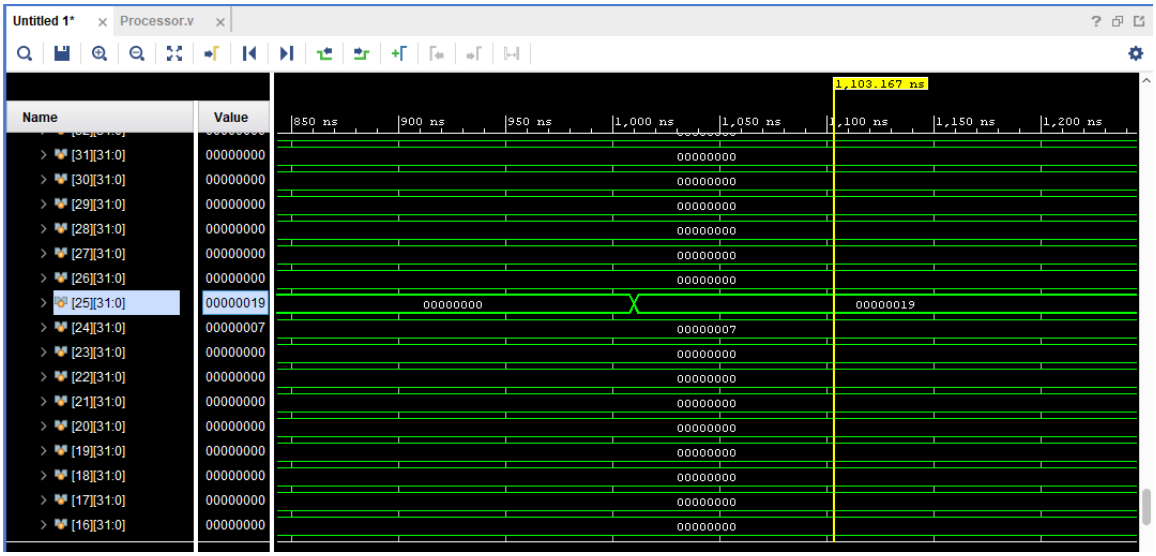


Figure 11: Data memory after running `riscvtest.s` on my processor

This same code is then run again; only this time, an interrupt is given via the switch input as shown below:

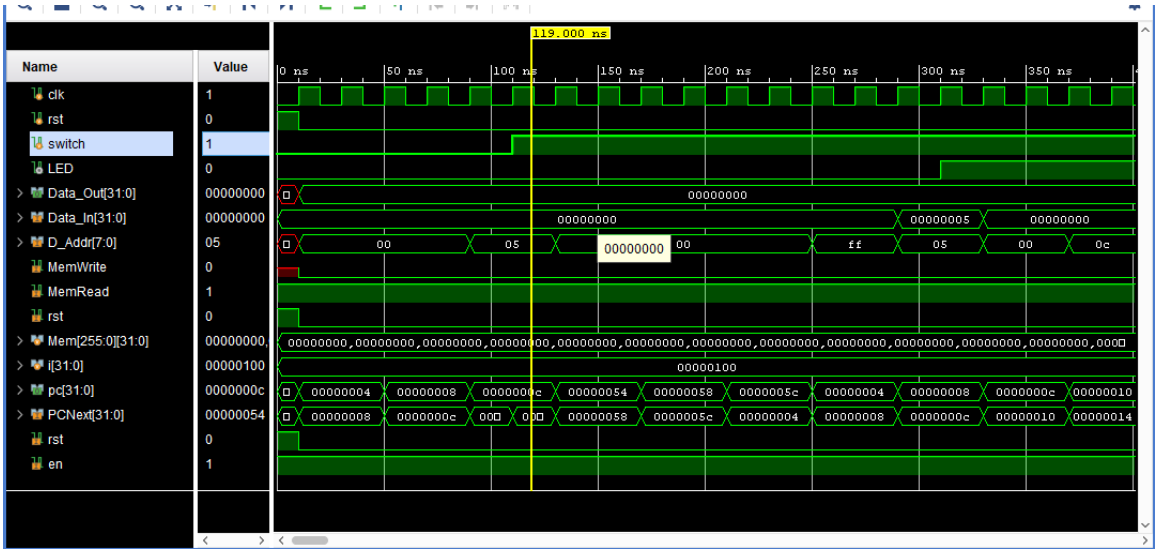


Figure 12: Giving interrupt using the switch

Note how the pc[31:0] goes to 0x54 which is the address of the ISR and as a result of the execution of the ISR, the LED output is toggled i.e., it goes from LOW to HIGH. After the ISR is executed, the pc[31:0] goes back to 0x4. So, the CSR module is working correctly. Moreover, the value 25 or 0x19 is also written to the data memory at the address 25 as shown below:

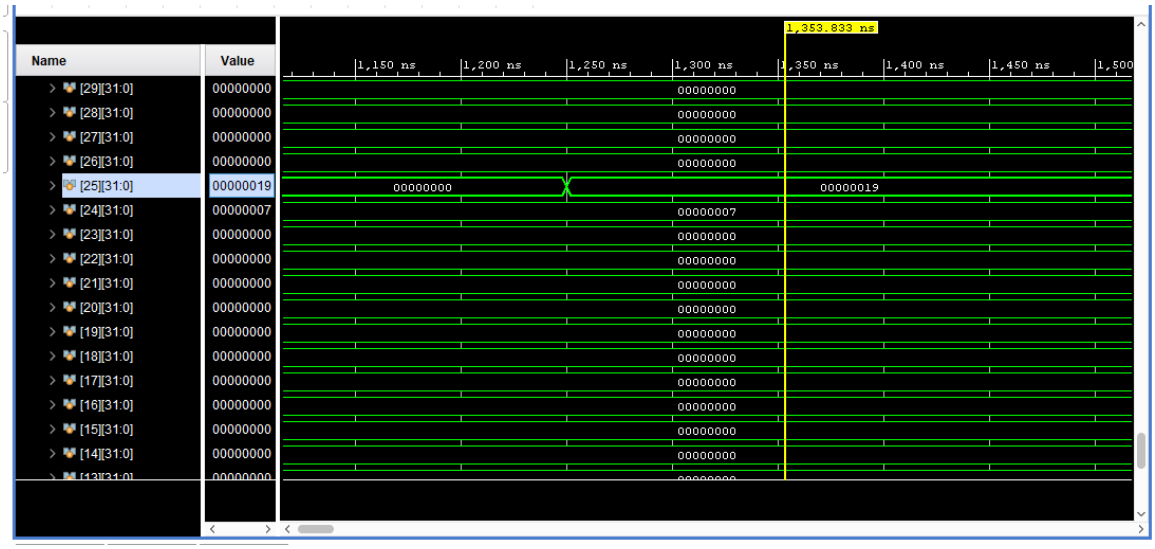


Figure 13: Data memory for testing CSR module

Another test is the one that Sir Umer uploaded on Piazza. The results of this test are shown below:

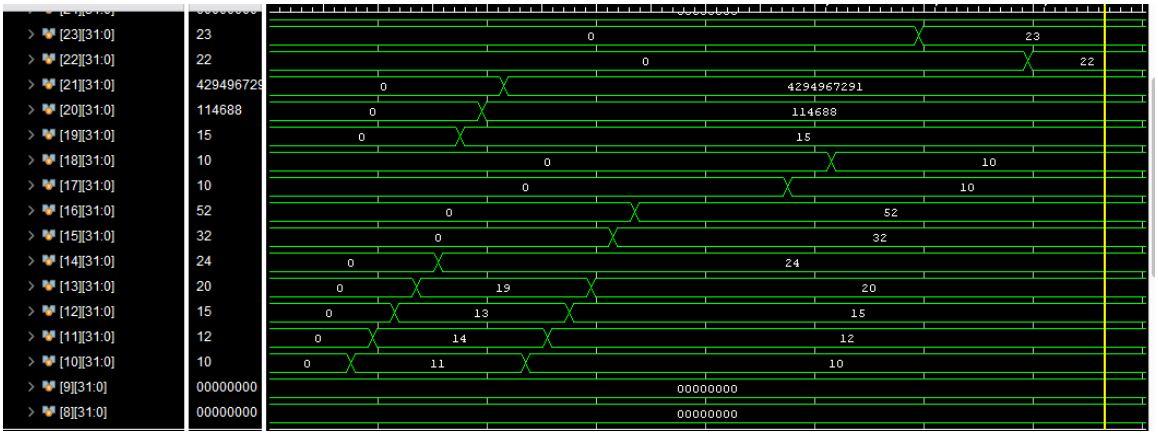
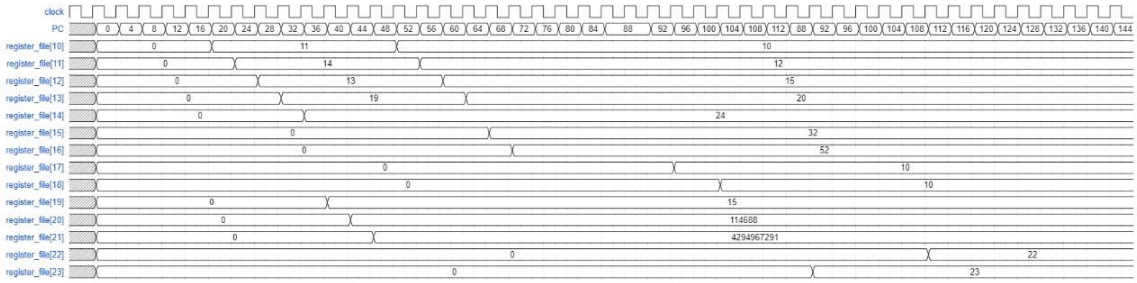


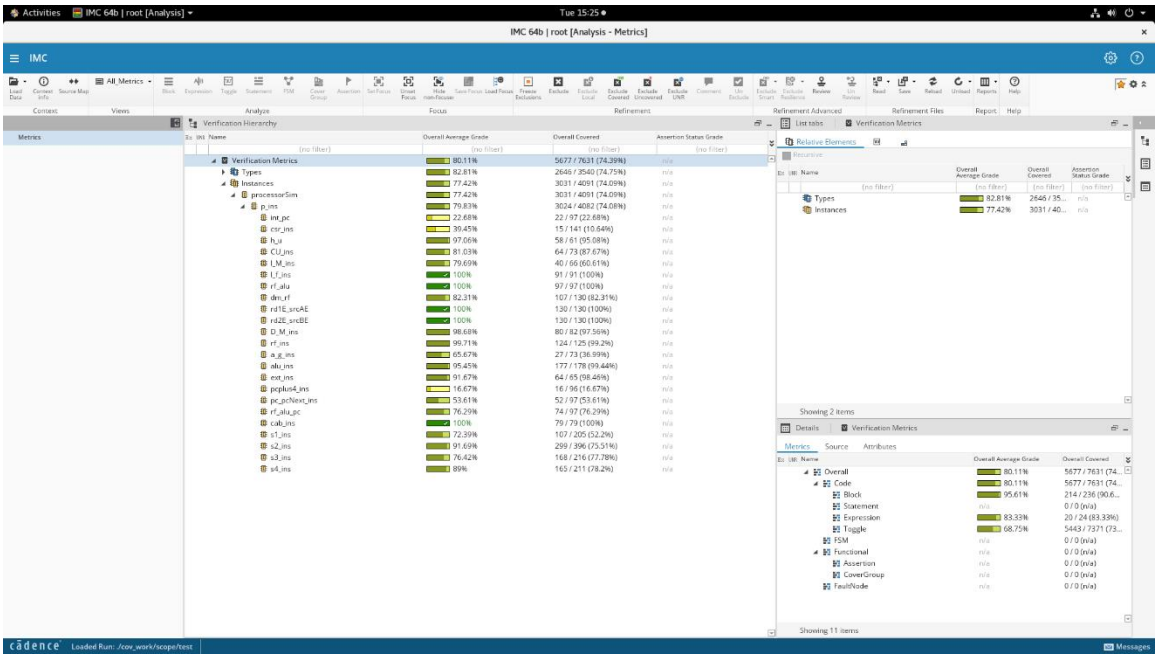
Figure 14: Results after running Test_Imem.mem

This Test_Imem file is given in the Appendix. Its expected output is as under:



This output is exactly the same as the one I have obtained which confirms that my processor is working correctly.

3.4. Coverage Report



The assembly code I used for the coverage test is given in the Appendix as `coverage.s` For the purpose of coverage, in order to get maximum coverage, I have used a data memory of size 16 bytes or 4 words only. The coverage code reads from and writes to all four memory locations multiple times in order to get maximum coverage.

3.5. Challenges and Limitations

I have faced many challenges during the development of my processor and the developed processor has many limitations as well. These are explained in the subsequent sections.

3.5.1. Challenges

A frequent challenge I faced during the development of this processor is the ambiguity I faced due to the discrepancies between the designs that Sir Umer was using in the lab and the designs that Sir Tahir was teaching in the class. Due to these differences, my processor is a combination of the processor explained in the Pipelined Control Section of the book Digital Design and Computer Architecture RISC-V Edition [1] and the described in Sir Tahir's class slides. Another challenge was the development of the CSR module as we had never written Verilog code for interrupts.

3.5.2. Limitations

One limitation of the developed processor is that it does not support U-type instructions. It also does not currently support CSR Read Write operations and does not have any privilege levels.

4. Bibliography

[1] S. Harris and D. Harris, Digital Design and Computer Architecture RISC-V Edition, Cambridge: Katey Birtcher, 2006.

[2] "Verilog Positive Edge Detector," ChipVerify, [Online]. Available: https://www.chipverify.com/verilog/verilog-positive-edge-detector. [Accessed 4 October 2021].

APPENDIX

```
# riscvtest.s
# Sarah.Harris@unlv.edu
# David_Harris@hmc.edu
# 27 Oct 2020
#
# Test the RISC-V processor:
#   add, sub, and, or, slt, addi, lw, sw, beq, jal
# If successful, it should write the value 25 to address 100
#       RISC-V Assembly         Description                Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                   0         00500113
        addi x3, x0, 12         # x3 = 12                  4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3        8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7        C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4      10        0041F2B3
        add  x5, x5, x4         # x5 = 4 + 7 = 11          14        004282B3
        beq  x5, x7, end        # shouldn't be taken       18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0        1C        0041A233
        beq  x4, x0, around     # should be taken          20        00020463
        addi x5, x0, 0          # shouldn't execute        24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5) = 1         28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12       2C        005203B3
        sub  x7, x7, x2         # x7 = (12 - 5) = 7        30        402383B3
        sw   x7, 84(x3)         # [96] = 7                 34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7            38        06002103
        add  x9, x2, x5         # x9 = (7 + 11) = 18       3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44   40        008001EF
        addi x2, x0, 1          # shouldn't execute        44        00100113
end:    add  x2, x2, x9         # x2 = (7 + 18) = 25       48        00910133
        sw   x2, 0x20(x3)       # [100] = 25               4C        0221A023
done:   beq  x2, x2, done       # infinite loop            50        00210063
```

coverage.s

```
addi x0, x0, 1601
addi x1, x0, -209
addi x2, x0, 1541
addi x3, x0, 80
addi x4, x0, 237
addi x5, x0, 553
addi x6, x0, 1046
addi x7, x0, 1688
addi x8, x0, -1444
addi x9, x0, -10
addi x10, x0, -1218
addi x11, x0, 551
addi x12, x0, 1576
addi x13, x0, -1221
addi x14, x0, -1868
addi x15, x0, -610
addi x16, x0, -1009
addi x17, x0, 1119
addi x18, x0, 780
addi x19, x0, 1065
addi x20, x0, -2014
addi x21, x0, -1751
addi x22, x0, 713
addi x23, x0, -1656
addi x24, x0, -673
addi x25, x0, 759
addi x26, x0, -1195
addi x27, x0, -604
addi x28, x0, 1084
addi x29, x0, 1475
addi x30, x0, 433
addi x31, x0, -1694

     add x0, x0, x1
     add x1, x1, x2
```

```
add x2, x2, x3
add x3, x3, x4
add x4, x4, x5
add x5, x5, x6
add x6, x6, x7
add x7, x7, x8
add x8, x8, x9
add x9, x9, x10
add x10, x10, x11
add x11, x11, x12
add x12, x12, x13
add x13, x13, x14
add x14, x14, x15
add x15, x15, x16
add x16, x16, x17
add x17, x17, x18
add x18, x18, x19
add x19, x19, x20
add x20, x20, x21
add x21, x21, x22
add x22, x22, x23
add x23, x23, x24
add x24, x24, x25
add x25, x25, x26
add x26, x26, x27
add x27, x27, x28
add x28, x28, x29
add x29, x29, x30
add x30, x30, x31
add x31, x31, x0

sub x0, x0, x1
sub x1, x1, x2
sub x2, x2, x3
sub x3, x3, x4
sub x4, x4, x5
sub x5, x5, x6
sub x6, x6, x7
sub x7, x7, x8
sub x8, x8, x9
sub x9, x9, x10
sub x10, x10, x11
sub x11, x11, x12
sub x12, x12, x13
sub x13, x13, x14
```

```
sub x14, x14, x15
sub x15, x15, x16
sub x16, x16, x17
sub x17, x17, x18
sub x18, x18, x19
sub x19, x19, x20
sub x20, x20, x21
sub x21, x21, x22
sub x22, x22, x23
sub x23, x23, x24
sub x24, x24, x25
sub x25, x25, x26
sub x26, x26, x27
sub x27, x27, x28
sub x28, x28, x29
sub x29, x29, x30
sub x30, x30, x31
sub x31, x31, x0


or x0, x0, x1
or x1, x1, x2
or x2, x2, x3
or x3, x3, x4
or x4, x4, x5
or x5, x5, x6
or x6, x6, x7
or x7, x7, x8
or x8, x8, x9
or x9, x9, x10
or x10, x10, x11
or x11, x11, x12
or x12, x12, x13
or x13, x13, x14
or x14, x14, x15
or x15, x15, x16
or x16, x16, x17
or x17, x17, x18
or x18, x18, x19
or x19, x19, x20
or x20, x20, x21
or x21, x21, x22
or x22, x22, x23
or x23, x23, x24
or x24, x24, x25
```

```
or x25, x25, x26
or x26, x26, x27
or x27, x27, x28
or x28, x28, x29
or x29, x29, x30
or x30, x30, x31
or x31, x31, x0

xor x0, x0, x1
xor x1, x1, x2
xor x2, x2, x3
xor x3, x3, x4
xor x4, x4, x5
xor x5, x5, x6
xor x6, x6, x7
xor x7, x7, x8
xor x8, x8, x9
xor x9, x9, x10
xor x10, x10, x11
xor x11, x11, x12
xor x12, x12, x13
xor x13, x13, x14
xor x14, x14, x15
xor x15, x15, x16
xor x16, x16, x17
xor x17, x17, x18
xor x18, x18, x19
xor x19, x19, x20
xor x20, x20, x21
xor x21, x21, x22
xor x22, x22, x23
xor x23, x23, x24
xor x24, x24, x25
xor x25, x25, x26
xor x26, x26, x27
xor x27, x27, x28
xor x28, x28, x29
xor x29, x29, x30
xor x30, x30, x31
xor x31, x31, x0

sra x0, x0, x1
sra x1, x1, x2
sra x2, x2, x3
sra x3, x3, x4
```

```
sra x4, x4, x5
sra x5, x5, x6
sra x6, x6, x7
sra x7, x7, x8
sra x8, x8, x9
sra x9, x9, x10
sra x10, x10, x11
sra x11, x11, x12
sra x12, x12, x13
sra x13, x13, x14
sra x14, x14, x15
sra x15, x15, x16
sra x16, x16, x17
sra x17, x17, x18
sra x18, x18, x19
sra x19, x19, x20
sra x20, x20, x21
sra x21, x21, x22
sra x22, x22, x23
sra x23, x23, x24
sra x24, x24, x25
sra x25, x25, x26
sra x26, x26, x27
sra x27, x27, x28
sra x28, x28, x29
sra x29, x29, x30
sra x30, x30, x31
sra x31, x31, x0

srl x0, x0, x1
srl x1, x1, x2
srl x2, x2, x3
srl x3, x3, x4
srl x4, x4, x5
srl x5, x5, x6
srl x6, x6, x7
srl x7, x7, x8
srl x8, x8, x9
srl x9, x9, x10
srl x10, x10, x11
srl x11, x11, x12
srl x12, x12, x13
srl x13, x13, x14
srl x14, x14, x15
srl x15, x15, x16
```

```
srl x16, x16, x17
srl x17, x17, x18
srl x18, x18, x19
srl x19, x19, x20
srl x20, x20, x21
srl x21, x21, x22
srl x22, x22, x23
srl x23, x23, x24
srl x24, x24, x25
srl x25, x25, x26
srl x26, x26, x27
srl x27, x27, x28
srl x28, x28, x29
srl x29, x29, x30
srl x30, x30, x31
srl x31, x31, x0

sll x0, x0, x1
sll x1, x1, x2
sll x2, x2, x3
sll x3, x3, x4
sll x4, x4, x5
sll x5, x5, x6
sll x6, x6, x7
sll x7, x7, x8
sll x8, x8, x9
sll x9, x9, x10
sll x10, x10, x11
sll x11, x11, x12
sll x12, x12, x13
sll x13, x13, x14
sll x14, x14, x15
sll x15, x15, x16
sll x16, x16, x17
sll x17, x17, x18
sll x18, x18, x19
sll x19, x19, x20
sll x20, x20, x21
sll x21, x21, x22
sll x22, x22, x23
sll x23, x23, x24
sll x24, x24, x25
sll x25, x25, x26
sll x26, x26, x27
sll x27, x27, x28
```

```
sll x28, x28, x29
sll x29, x29, x30
sll x30, x30, x31
sll x31, x31, x0

addi x0, x0, 1
addi x1, x1, 1
addi x2, x2, 1
addi x3, x3, 1
addi x4, x4, 1
addi x5, x5, 1
addi x6, x6, 1
addi x7, x7, 1
addi x8, x8, 1
addi x9, x9, 1
addi x10, x10, 1
addi x11, x11, 1
addi x12, x12, 1
addi x13, x13, 1
addi x14, x14, 1
addi x15, x15, 1
addi x16, x16, 1
addi x17, x17, 1
addi x18, x18, 1
addi x19, x19, 1
addi x20, x20, 1
addi x21, x21, 1
addi x22, x22, 1
addi x23, x23, 1
addi x24, x24, 1
addi x25, x25, 1
addi x26, x26, 1
addi x27, x27, 1
addi x28, x28, 1
addi x29, x29, 1
addi x30, x30, 1
addi x31, x31, 1


#alu testing
add x1,x2,x3
add x2,x31,x5

or x4,x3,x2
or x1,x31,x2
```

```
ori x16,x7,24
ori x17,x8,-24

and x5,x3,x2
and x6,x31,x2

andi x18,x16,29
andi x19,x18,-27

xor x7,x5,x0
xor x8,x31,x1

xori x20,x17,23
xori x21,x18,-14

slt x9, x7,x5
slt x10, x5,x7

slti x22,x11,34
slti x23,x10,-29

sra x13,x31,x1
sra x14,x15,x1

srai x24,x2,24
srai x25,x18,2

slli x28,x7,11
slli x29,x10,4

srli x13,x1,11
srli x14,x17,7

sw x2, 0(x0)
sw x3, 4(x0)
sw x4, 8(x0)
sw x5, 12(x0)

lw x8, 8(x0)
lw x9, 12(x0)
lw x31, 0(x0)
lw x20, 4(x0)

sw x7, 0(x0)
```

```
      sw x9, 4(x0)
      sw x3, 8(x0)
      sw x11, 12(x0)

      lw x12, 8(x0)
      lw x21, 12(x0)
      lw x1, 0(x0)
      lw x7, 4(x0)

      sw x7, 0(x0)
      sw x24, 4(x0)
      sw x17, 8(x0)
      sw x19, 12(x0)

      lw x18, 8(x0)
      lw x13, 12(x0)
      lw x6, 0(x0)
      lw x10, 4(x0)

      addi x1,x0,1
      addi x2,x0,2
      beq x0,x21, failed_branch
      beq x0,x0, branch
      addi x2,x0, 7
      addi x3,x0, 8
      addi x4,x0, 9
branch:
      addi x4, x0, 4
      bne x0,x0, failed_branch
      bne x1,x0, branch2

      addi x5,x0, 11
      addi x6,x0, 12
      addi x7,x0, 13
branch2:
      addi x5,x0,5
      blt x5,x0, failed_branch
      blt x0,x5, branch3

      addi x12,x0, 7
      addi x13,x0, 8
      addi x14,x0, 9

branch3:
      addi x6,x0,6
```

```
      bge x0,x6, failed_branch
      bge x6,x0, continue

failed_branch:
      addi x10, x0,10
      j failed_branch

continue:
      addi x13,x4,1
      jal x0, jump_ahead
      addi x19,x0, 19
      addi x20,x0, 20
jump_ahead:
      addi x27,x0, 27
      addi x28,x0, 28
stop:
      j stop
addi x1,x0,1
addi x2,x1,1
```

Test_Imem.mem

```
00b00513
00e00593
00d00613
01300693
01800713
00b569b3
00c59a33
40e68ab3
00a00513
00c00593
00f00613
01400693
00d587b3
00f68833
00a02623
00000013
00000013
00000013
00000013
00000013
00c02883
01100933
01700663
016b0b13
00000663
01700b93
fe0008e3
00000013
```

VITA

My name is Usama Ayub and I am currently an Electrical Engineering student at the University of Engineering and Technology, Lahore. I passed my FSc exam in the year 2017 with 981 marks out of 1100. I did my matriculation in 2015 with 1008 marks out of 1100. I shall finish my Electrical Engineering degree this year (2022) with specialization in Computer.