CSCE3301

Computer Architecture

Project 1 – MS3 & MS4

Spring 2022

April 14, 2022

By:

Omar Elayat – 900182568

Bishoy Sabry - 900183106

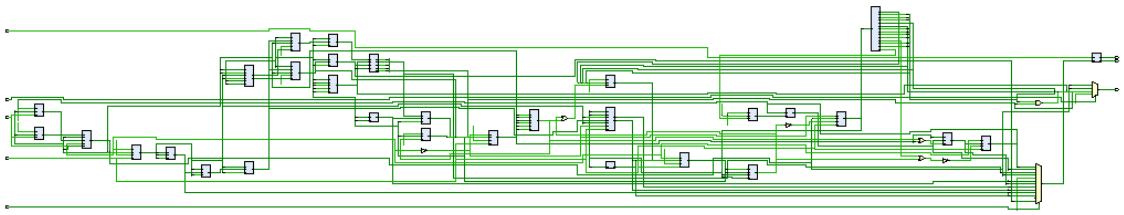
Contents

Introduction

This milestone implements a single cycle RISCV processor that supports all 40 user-level instructions. The implementation was done in Verilog, and was debugged and tested on Vivado software. The instructions were divided into six format groups with similar functionality, and each format was designed separately using tailored Verilog modules. In this milestone, we have assembled and tested the whole processor, and the results were recorded and verified.

This milestone implements a 5 stage pipelining RISCV processor using a single ported byte addressable memory for both data and instructions. The processor was modified to handle all kinds of hazards introduced with the implementation of the pipeline. These hazards include: Structural hazards, Data hazards, and Control hazards. The processor was debugged and tested on Vivado software. In this milestone, we have assembled and thoroughly tested the whole processor, and the results were recorded and verified.

Full Schematic



Pipelined implementation

The processor is divided into five unbalanced stages, which are: Fetching, Decoding, Executing, Memory, and Write back. The stages are created by separating the processor blocks by four flip flops that intermediates each two consecutive stages. The pipelined implementation introduced some hazards that were handled later on in the milestone. All the 40 user level instructions were tested on the pipelined processor in scenarios that doesn't cause any of the three aforementioned hazards, and the results were recorded and compared to the results obtained from venus online RISCV simulator, and RARs RISCV simulator.

Fetching stage (IF)

In this stage, the instruction is fetched from the instruction memory, the next PC is calculated, the target address is calculated and stored in the PC counter.

Implementation

There is no Stage flip flop in the start of this stage as it's the first stage in the pipelines. The input is the PC target address, and the output is the fetched instruction, the PC and the PC+4.

Stage modules

1- PC (Program Counter)
   It was modeled as a simple 32-bit array of D-Flip-flops

2- Adders
   n-bit Ripple carry adders modeled as a series of full carry adders.

3- Multiplexers (Muxs)
   Normal 2x1 n-bits multiplexers

4- Main Memory
   It is a single-ported byte addressable memory that holds both the data and the instructions. It is implemented as 2D array of flip-flops that can be accessed through control signals.

Reasons behind choosing these modules in this stage

This stage is concerned with only extracting the instructions from the main memory, and computing the PC. Thus, no need to include any further components that computes any further functionality

Decoding stage (ID)

In this stage, the register file is read, the control unit sets the values of the selection lines, the immediate generator extracts the immediate constant – if any- from the instruction. Also, the hazard detection unit detects the load-use hazard in this stage.

Implementation

The stage is separated from the fetching stage with a 96 bits flip-flop (IF_ID FF) from which the stage receives its inputs from. The outputs are forwarded to the next stage through another intermediate flip-flop (ID_EX FF).

Stage modules

1- Control unit
   A simple combinational circuit that enables/disables select lines based on a k-map built on the instruction's opcodes

2- Register file
   Implemented as an array of registers, where each register is 32- flip-flops representing 32 bits.

3- Immediate generator
   Implemented as combinational circuit that extends the immediate provided according to the type of the instruction being implemented.

4- Multiplexers (Muxs)
   Normal 2x1 n-bits multiplexers

5- Hazards detection unit
   A combinational circuit that checks some conditions that ensures that the instruction being executed doesn't require stalling to accurately execute.

Reasons behind choosing these modules in this stage

This stage is concerned with only decoding the previously fetched instruction into a set of selection lines that controls the flow of the bits through the rest of the data path, thus the control unit is included. Also, the register file is included to read the data stored in the register for further execution along with the immediate constants decoded by the immediate generator. The hazard detection unit is also included to issue one bubble in the pipeline if needed. No further functionality was required from this stage

Executing stage (EX)

In this stage, the register file is read, the control unit sets the values of the selection lines, the immediate generator extracts the immediate constant – if any- from the instruction, and

Implementation

The stage is separated from the decoding stage with a 197 bits flip-flop (ID_EX FF) from which the stage receives its inputs from, including the control signals of all the next stages. The outputs are forwarded to the next stage through another intermediate flip-flop (EX_MEM FF).

Stage modules

1- ALU control Unit
   It's a smaller control unit that was designed to receive a control signal from the main control unit in addition to specific bits from the instructions machine code. It performs combinational operations to produce a 4-bit ALU function selector signal that chooses a specific function to be operated by the ALU.

2- Adders
   n-bit Ripple carry adders modeled as a series of full carry adders.

3- ALU
   The Arithmetic logical unit is a combinational circuit that performs arithmetic, logical and shifting operations on the input data based on the incoming selection lines. It was modeled as a huge LUT of operations mapped on the select signals.

4- Forwarding unit
   Implemented as a combinational circuit that detects data hazards between the instruction being currently executed and the previous instruction. It then resolves this hazard by forwarding data between stages.

Reasons behind choosing these modules in this stage

This staging is considered the most important stage in the RISCV pipelined processor, as the instruction is actually executed in this stage. The MUXs are used to select which exact data to be input to the ALU based on the values of the selection lines already calculated in the previous stage. The ALU control unit produce a 4-bit ALU function selector signal that chooses a specific function to be operated by the ALU. Finally, the forwarding unit is used to input forwarded data from previous instructions to the ALU to avoid data hazards. No further functionality was required from this stage.

Memory stage (MEM)

In this stage, the branch decision is taken, and the main memory is accessed again for storing or loading data.

Implementation

The stage is separated from the decoding stage with a 183 bits flip-flop (EX_MEM FF) from which the stage receives its inputs from, including the control signals of the next stage (write back stage). The outputs are forwarded to the next stage through another intermediate flip-flop (MEM_WB FF).

Stage modules

1- Multiplexers (Muxs)
   Normal 2x1 n-bits multiplexers.

2- Main Memory
   It is a single-ported byte addressable memory that holds both the data and the instructions. It is implemented as 2D array of flip-flops that can be accessed through control signals.

3- Branch control unit
   After the ALU compares the two inputs, this unit takes the sign flag, overflow flag, carry flag, and zero flag signals. It enables/disables the branch operation by computing a control signal based on the input flags and main control unit input.

Reasons behind choosing these modules in this stage

This staging reads and writes data to the memory. And evaluates the branch instructions in the branch control unit. No further functionality is required from this stage.

Write back stage (WB)

In this stage, the executed data is propagated to the register file to be written.

Implementation

The stage is separated from the decoding stage with a 137 bits flip-flop (MEM_WB FF) from which the stage receives its inputs from. The outputs are either written in the register file or flushed.

Stage modules

1- Multiplexers (Muxs)
   Normal 2x1 n-bits multiplexers.
2- Register file
   Implemented as an array of registers, where each register is 32- flip-flops representing 32 bits.

Reasons behind choosing these modules in this stage

This stage whole purpose is to write the executed/memory-loaded data to the register file. So, it is composed of only a set of multiplexers that propagates the data to the register file according to the selection lines values.

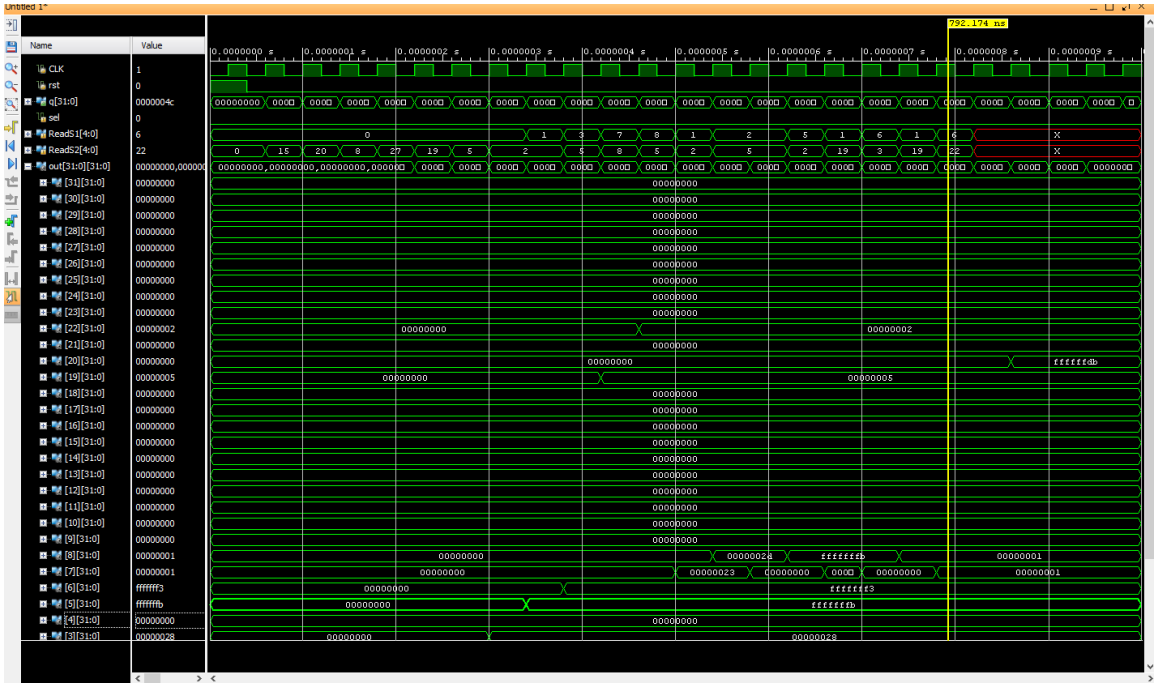
Testing

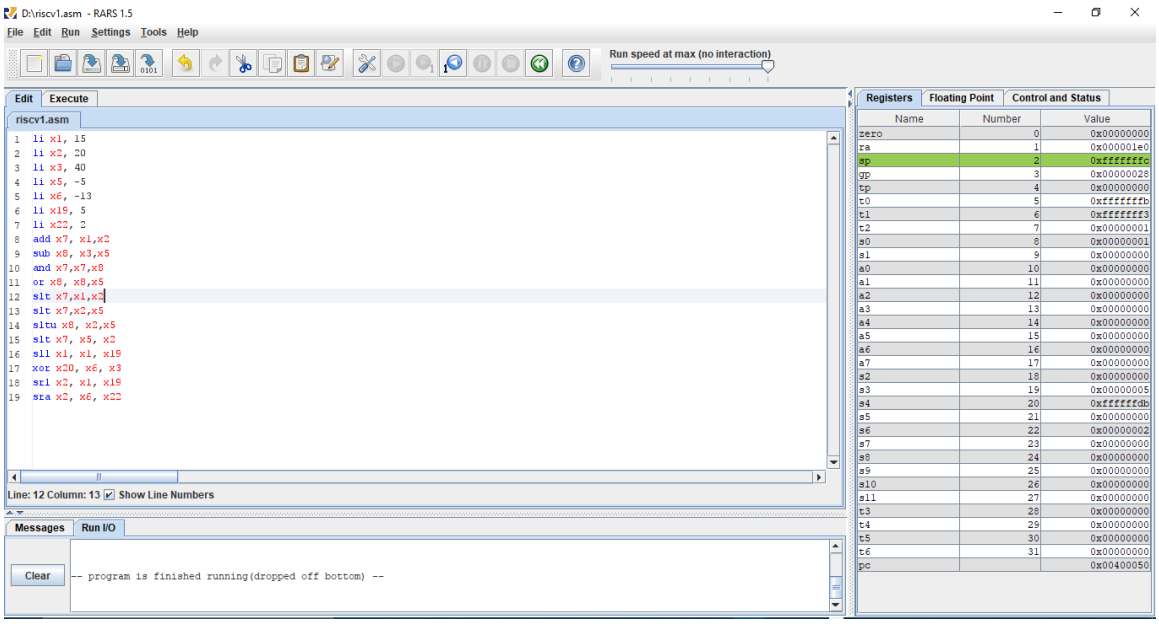
To test all the 40 level instructions, 6 programs were written and tested in the pipelined processor. Each program contains a mixture of instructions from all types but mainly focuses on a certain format out of the six. For each file, the result was recorded and compared with RARS RISCV simulator, and Venus RISCV simulator.

R-Format testing:

The processor was tested using the attached “R_program.txt” program. The result was compared with RARS RISCV simulator, and Venus RISCV simulator.

Results:





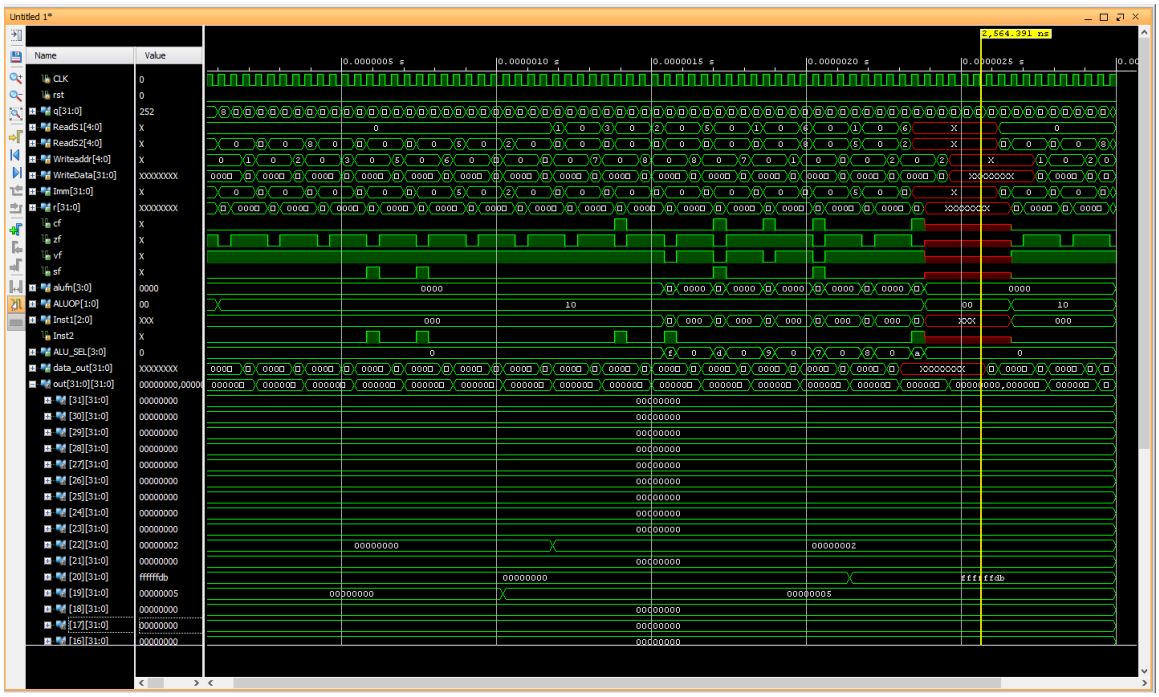
Analysis:

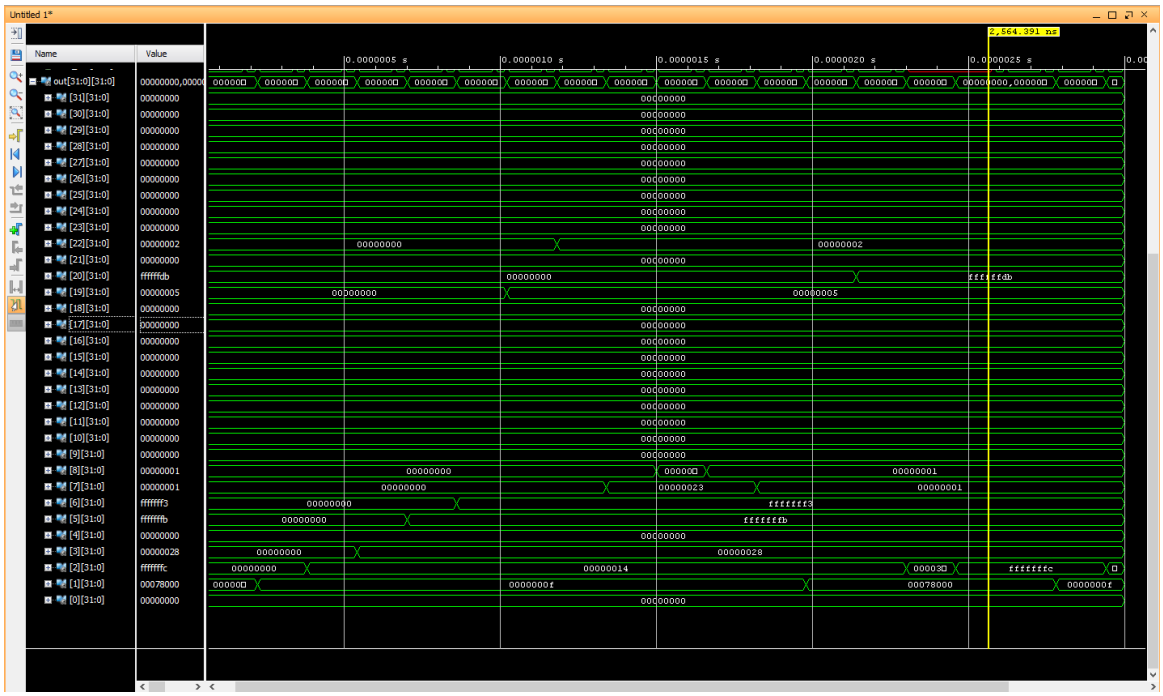
After ignoring the hazards with nops, all the instructions were executed as expected, with no detected errors. The pipelined processor can successfully perform all supported RISCV R-Format instructions.

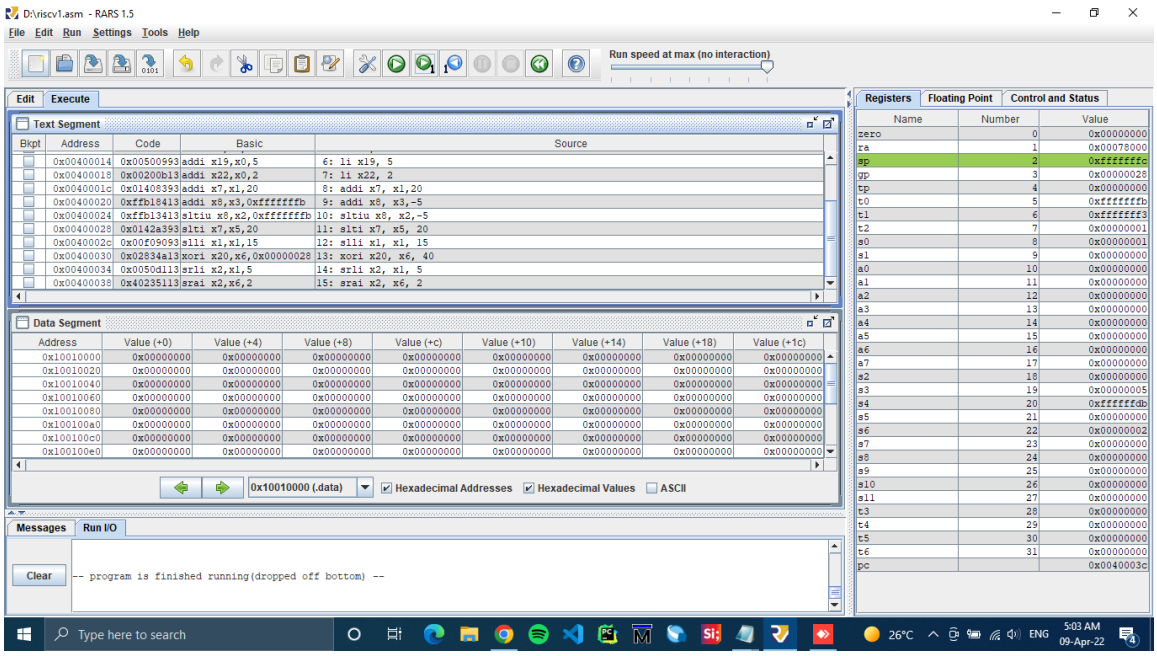
I-Format testing:

The Program was tested using the attached “I_program.txt” program. The result was compared with RARS RISCV simulator, and Venus RISCV simulator.

Results:







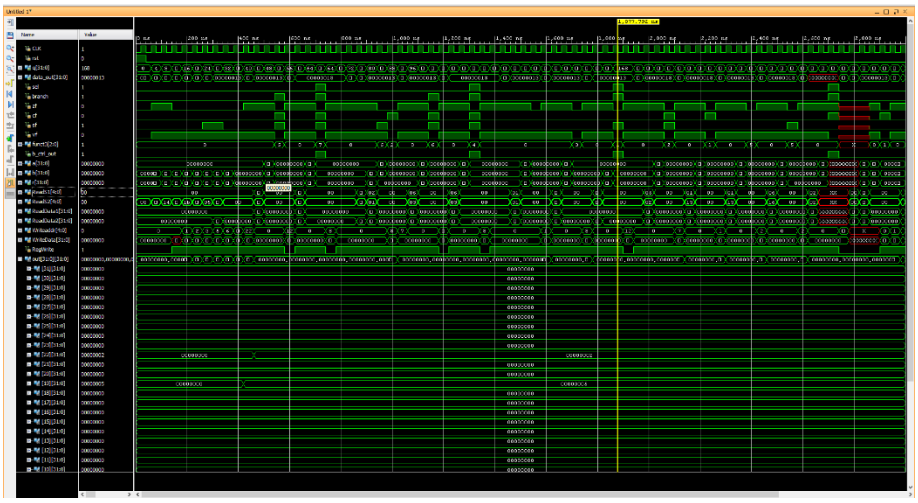
Analysis:

After ignoring the hazards with nops, all the instructions were executed as expected, with no detected errors. The pipelined processor can successfully perform all supported RISCV I-Format instructions.

B-Format testing:

The processor was tested using the attached “branch_test.txt” program. The result was compared with RARS RISCV simulator, and Venus RISCV simulator.

Results:



Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x0040001c	0x002083b3	add x7,x1,x2	8: l1:add x7, x1,x2
	0x00400020	0x40518433	sub x8,x3,x5	9: sub x8, x3,x5
	0x00400024	0x0083f3b3	and x7,x7,x8	10: and x7,x7,x8
	0x00400028	0x00235663	bge x6,x2,0x0000000c	11: bge x6,x2,l3 #not taken
	0x0040002c	0x00237463	bgeu x6,x2,0x00000008	12: bgeu x6,x2,l3 #taken
	0x00400030	0x00546433	or x8,x8,x5	13: or x8, x8,x5
	0x00400034	0x0020a3b3	slt x7,x1,x2	14: l3:slt x7,x1,x2
	0x00400038	0x005123b3	slt x7,x2,x5	15: slt x7,x2,x5
	0x0040003c	0x00336863	bltu x6,x3,0x00000010	16: bltu x6,x3,l4 #not taken
	0x00400040	0x00334663	blt x6,x3,0x0000000c	17: blt x6,x3,l4 #taken

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Messages | Run I/O

Clear

-- program is finished running (dropped off bottom) --

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x0000000f
sp	2	0xfffffffc
gp	3	0x00000028
tp	4	0x00000000
t0	5	0xfffffffb
t1	6	0xfffffff3
t2	7	0x00000009
s0	8	0x0000002d
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000005
s4	20	0xffffffdb
s5	21	0x00000000
s6	22	0x00000002
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400028

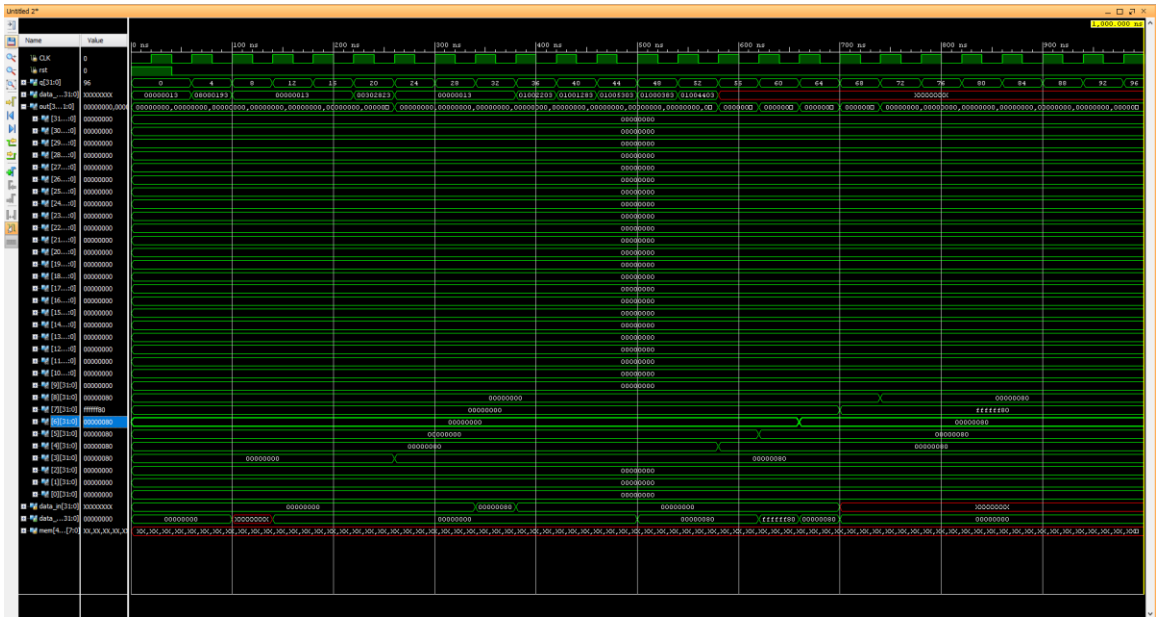
Analysis:

After ignoring the hazards with nops, all the instructions were executed as expected, with no detected errors. The pipelined processor can successfully perform all supported RISCV B-Format instructions.

Load/Store testing:

The processor was tested using the attached “machine_code.txt” program. The result was compared with RARS RISCV simulator, and Venus RISCV simulator.

Results:



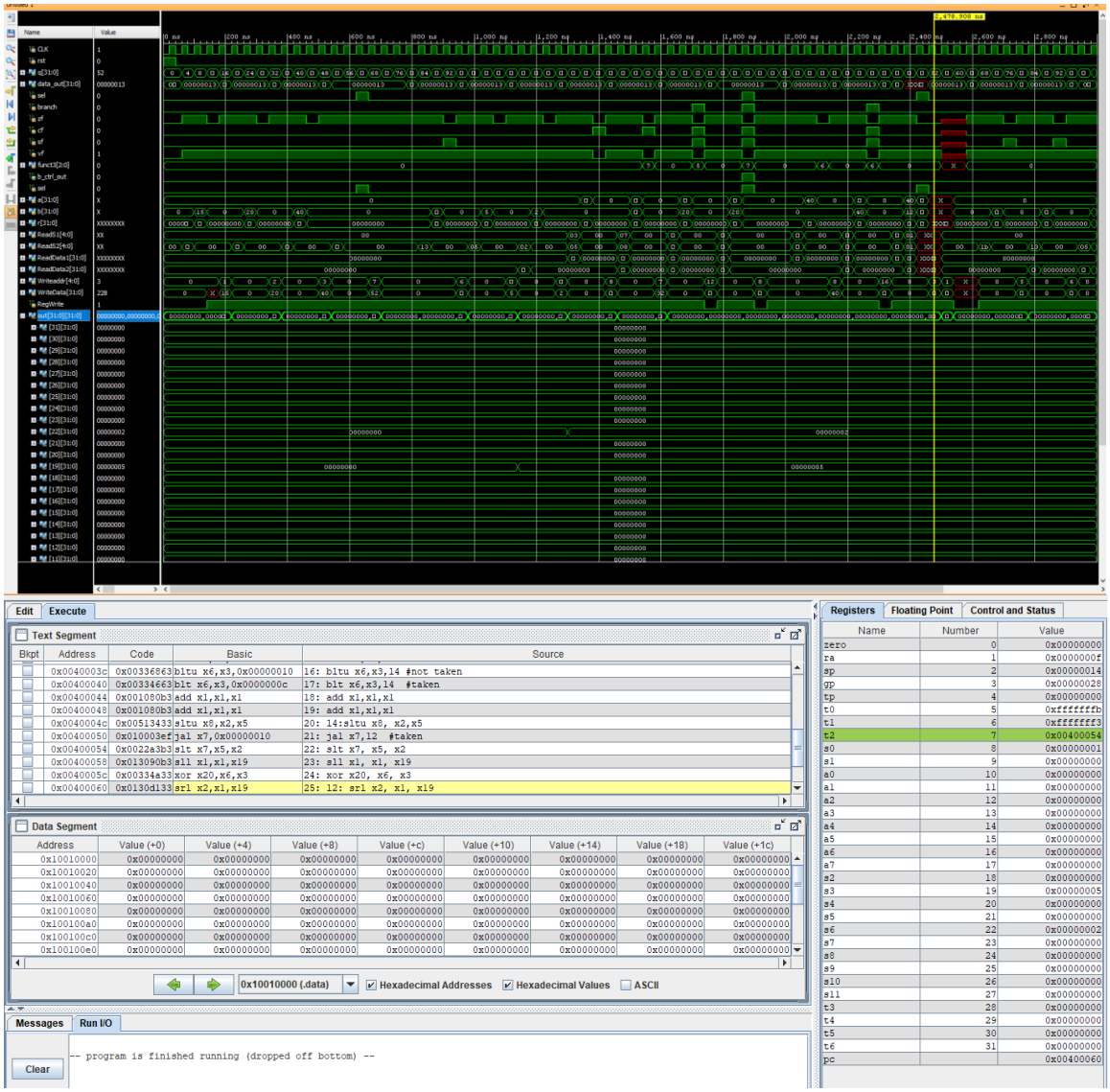
Analysis:

After ignoring the hazards with nops, all the instructions were executed as expected, with no detected errors. The pipelined processor can successfully perform all supported RISCV Load/Store instructions.

UJ-Format testing:

The processor was tested using the attached “machine_code.txt” program. The result was compared with RARS RISCV simulator, and Venus RISCV simulator.

Results:



Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x0040003c	0x00336863	bltu x6,x3,0x00000010	16: bltu x6,x3,14 #not taken
	0x00400040	0x00334663	blt x6,x3,0x0000000c	17: blt x6,x3,14 #taken
	0x00400044	0x001080b3	add x1,x1,x1	18: add x1,x1,x1
	0x00400048	0x001080b3	add x1,x1,x1	19: add x1,x1,x1
	0x0040004c	0x00513433	sltu x8,x2,x5	20: 14:sltu x8, x2,x5
	0x00400050	0x010003ef	jal x7,0x00000010	21: jal x7,12 #taken
	0x00400054	0x0022a3b3	slt x7,x5,x2	22: slt x7, x5, x2
	0x00400058	0x013090b3	sll x1,x1,x19	23: sll x1, x1, x19
	0x0040005c	0x00334a33	xor x20,x6,x3	24: xor x20, x6, x3
	0x00400060	0x0130d133	srl x2,x1,x19	25: l2: srl x2, x1, x19

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Messages | Run I/O

Clear

-- program is finished running (dropped off bottom) --

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x0000000f
sp	2	0x00000014
gp	3	0x00000028
tp	4	0x00000000
t0	5	0xfffffffb
t1	6	0xfffffff3
t2	7	0x00400054
s0	8	0x00000001
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000005
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000002
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400060

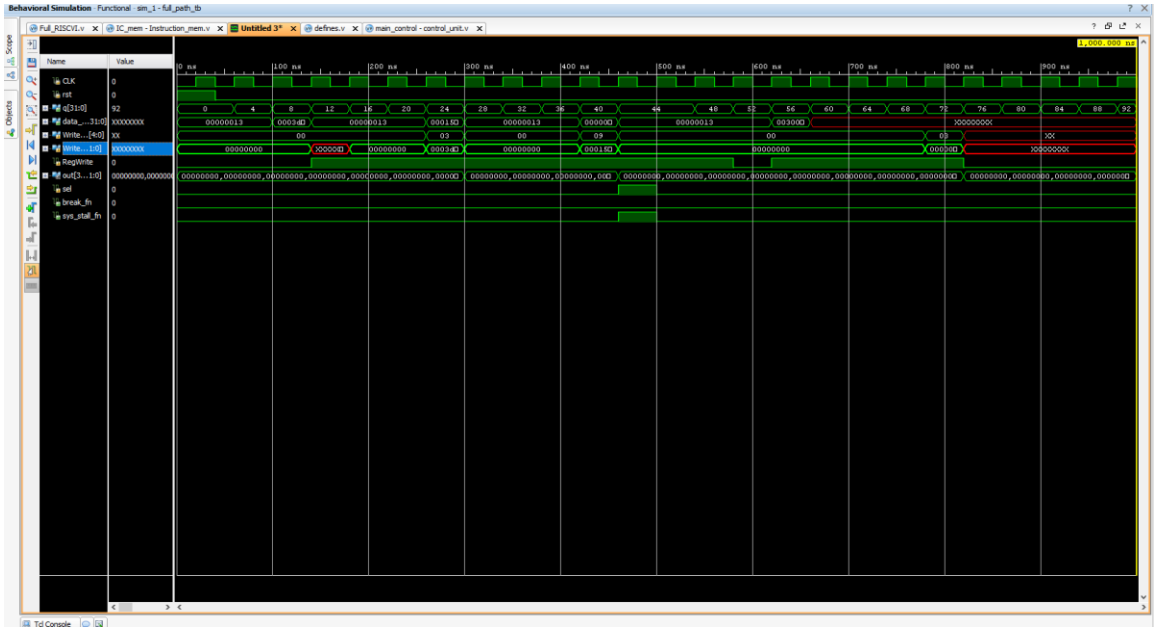
Analysis:

After ignoring the hazards with nops, all the instructions were executed as expected, with no detected errors. The pipelined processor can successfully perform all supported RISCV UJ-Format instructions.

Auipc/lui/sysfunctions testing:

The processor was tested using the attached “machine_code.txt” program. The result was compared with RARS RISCV simulator, and Venus RISCV simulator.

Results:



Analysis:

After ignoring the hazards with nops, all the instructions were executed as expected, with no detected errors. The pipelined processor can successfully perform all supported RISCV 40 level instructions.

Hazards

This section deals with the handling of the hazards introduced with the pipelined implementation of the RISCVI processor.

Structural hazards:

Definition:

Happens when the main memory is accessed from two different instructions at the same time. Solving this hazard is done by make each two consecutive stages work with opposite edge trigger. For example, the fetch, execute and write back stages are positive edge triggered, while the decode and memory stages are negative edge triggered. Thus, the memory and fetch stages will never be simultaneous.

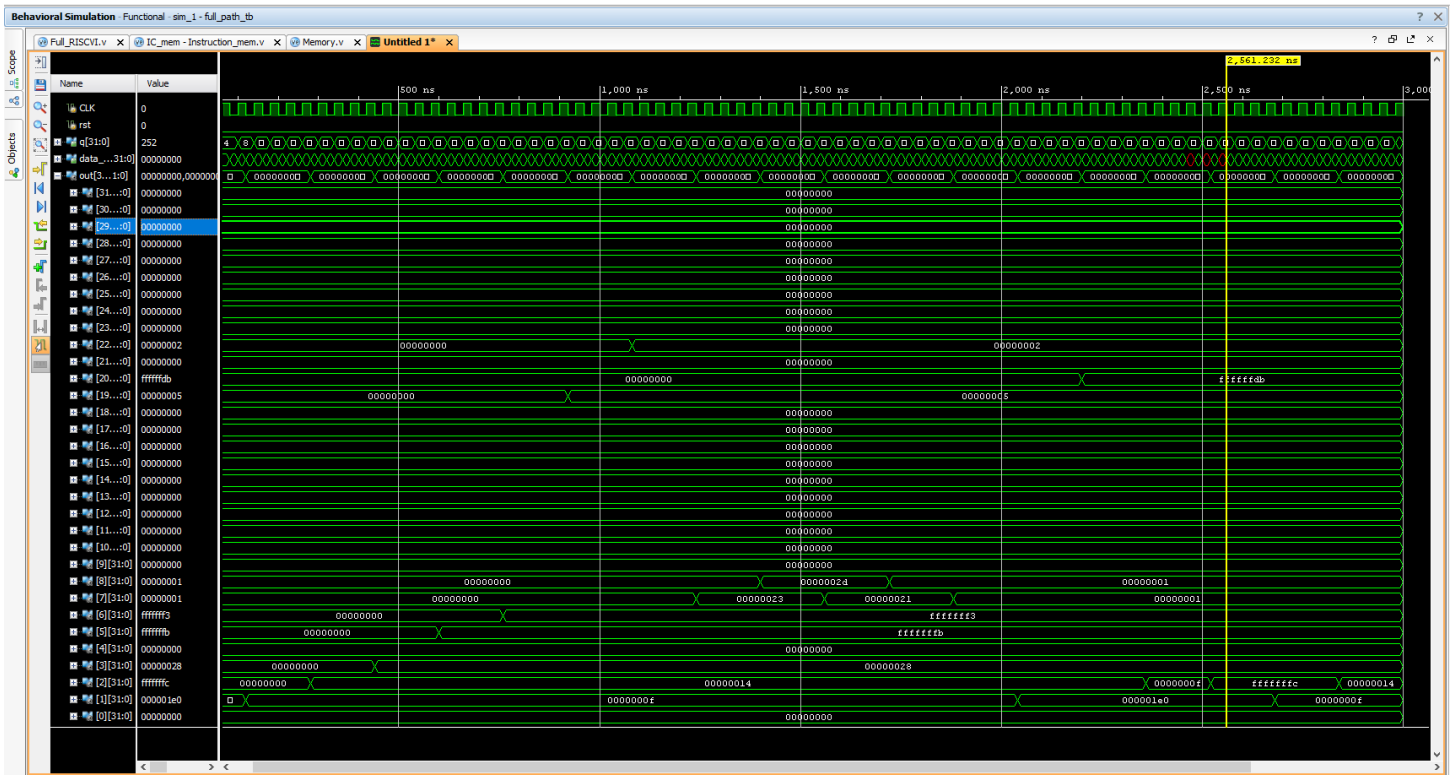
Testing: the following code are put to test the hazard detection unit.

```
addi x3, x0, 128
addi x1, x0, 256
sw x3, 16(x1)
lw x4, 16(x1)
lh x5, 16(x1)
lhu x6,16(x1)
lb x7, 16(x1)
lbu x8, 16(x1)
```

Starting from the third instruction, the processor accesses the memory for fetching and loading and stalling with each clock cycle. Thus, creating the hazard.

Results:

As it appears in the following figure, the instructions are fetched with the positive edge of the clock and the memory accesses are done in the negative edges


Behavioral Simulation - Functional - sim_1 - full_path_tb
Full_RISCVI.v
IC_mem - Instruction_mem.v
Memory.v
Untitled 1*
Name
Value
CLK
rst
q[31:0]
data_...31:0]
out[3...1:0]
2,561.232 ns
500 ns
1,000 ns
1,500 ns
2,000 ns
2,500 ns

Data Hazards:

Data hazards in in-order instructions execution can happen due to:

i. Read After Write (RAW) data dependency:

Definition:

Happens when the current instruction reads data from register being updated in the preceding instruction. Solving this hazard needs forwarding unit to detect the RAW dependency and forward the ALU result to the inputs of the ALU upon need before updating the register value in the register file.

Testing: the following code are put to test the forwarding unit.

```
add x0, x0, x0
addi x1,x0,17
addi x2,x0,9
addi x3,x0,25
sw x1, 200(x0)
sw x2, 204(x0)
sw x3, 208(x0)
add x0, x0, x0
lw x1, 200(x0)
lw x2, 204(x0)
lw x3, 208(x0)
or x4, x1, x2
beq x4, x3, 16
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x3, x1, x2
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
```

add x5, x3, x2

sw x5, 12(x0)

lw x6, 12(x0)

and x7, x6, x1

sub x8, x1, x2

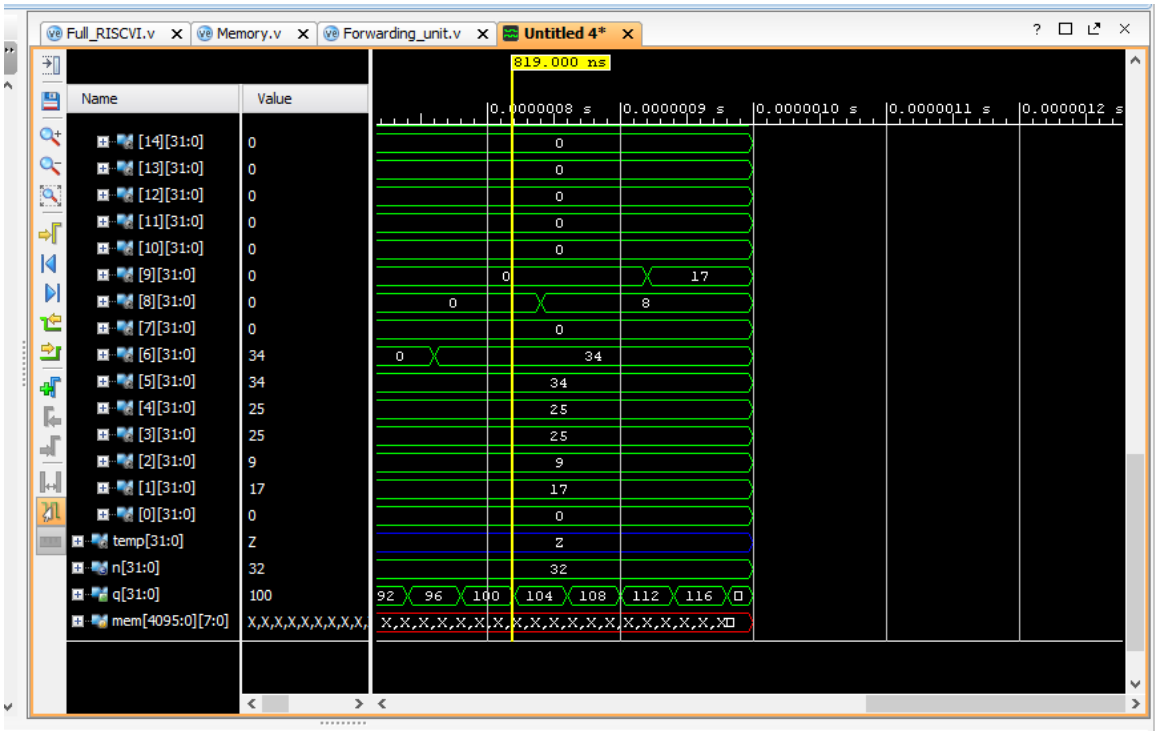
add x0, x1, x2

add x9, x0, x1

The bold instruction (**or x4, x1, x2)** will test the forwarding unit, returning the correct result in x4 means that the unit is working fine.

Results:

as it appears in the following figure, the x4 have the correct value.



ii. load-use dependency:

Definition:

Happens when the current instruction reads data from register being loaded from the memory in the preceding instruction. Solving this hazard needs hazard detection unit to detect the dependency and stall this instruction then forward the Memory result to the inputs of the ALU upon need before updating the register value in the register file.

Testing: the following code are put to test the hazard detection unit.

```
add x0, x0, x0
addi x1,x0,17
addi x2,x0,9
addi x3,x0,25
sw x1, 200(x0)
sw x2, 204(x0)
sw x3, 208(x0)
add x0, x0, x0
lw x1, 200(x0)
lw x2, 204(x0)
lw x3, 208(x0)
or x4, x1, x2
beq x4, x3, 16
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x3, x1, x2
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x5, x3, x2
sw x5, 12(x0)
lw x6, 12(x0)
```

and x7, x6, x1

sub x8, x1, x2

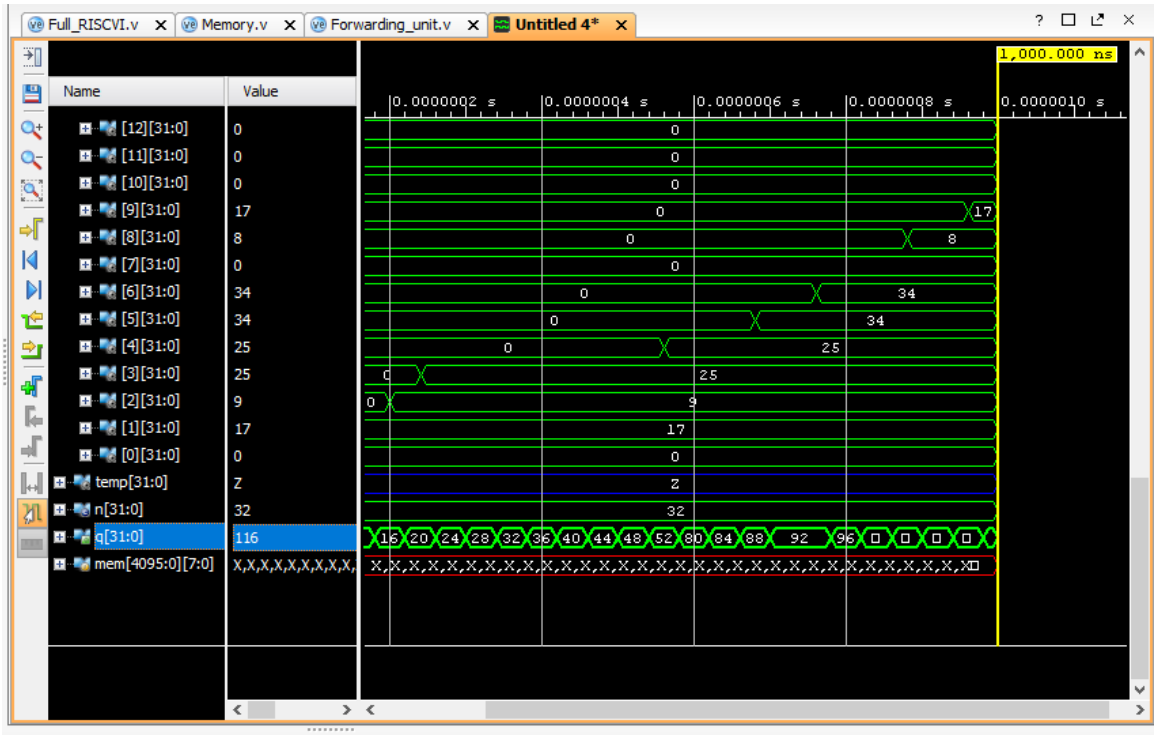
add x0, x1, x2

add x9, x0, x1

The bold instruction (**and x7, x6, x1)** will test the load-use dependency, stalling one cycle and returning the correct result in x7 means that the unit is working fine.

Results:

As it appears in the following figure, the x7 have the correct value. The PC is stalled at cycle 92.



Control Hazards:

This type of hazards occurs because the result of branch instruction decided in the memory stage after the next 3 instructions being fetched, decoded and executed in respectively. The solution is to detect each branch instruction and compare it with the branch predictor output and then flush the following 3 instructions.

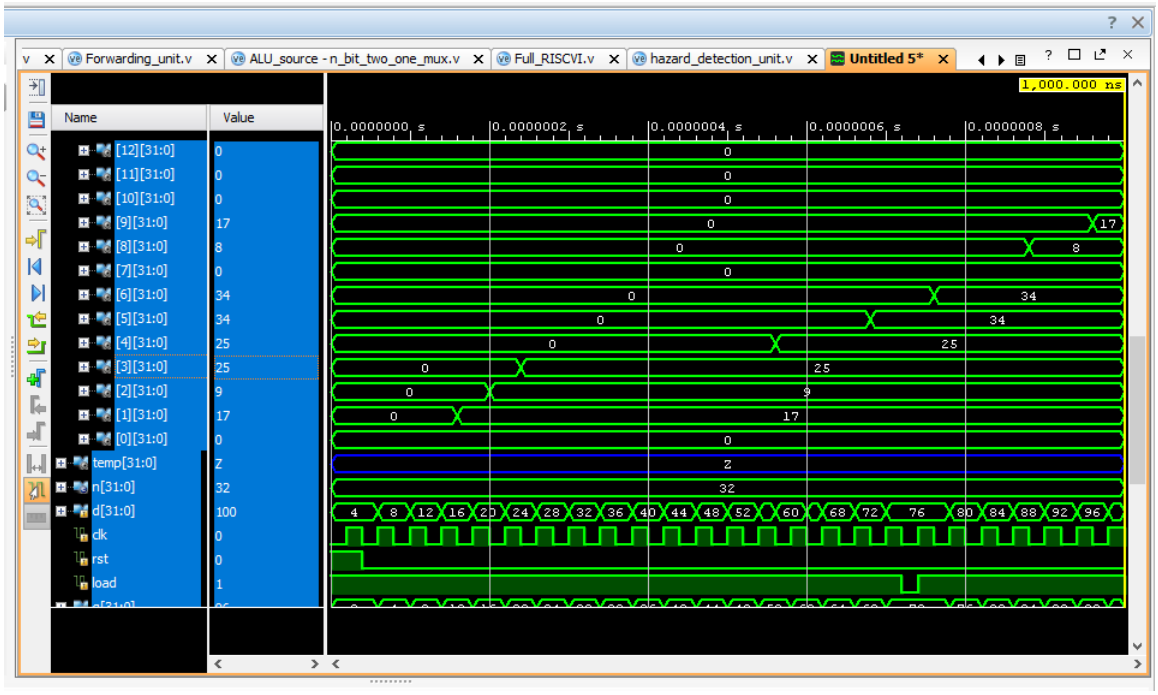
Testing: the following code are put to test the control hazard detection unit.

```
addi x1,x0,17
addi x2,x0,9
addi x3,x0,25
sw x1, 200(x0)
sw x2, 204(x0)
sw x3, 208(x0)
add x0, x0, x0
lw x1, 200(x0)
lw x2, 204(x0)
lw x3, 208(x0)
or x4, x1, x2
beq x4, x3, l1
add x10, x3, x0
add x3, x1, x2
l1:add x5, x3, x2
sw x5, 12(x0)
lw x6, 12(x0)
and x7, x6, x1
sub x8, x1, x2
add x0, x1, x2
add x9, x0, x1
```

The bold instructions (**add x10, x3, x0 & add x3, x1, x2)** will test the control, not executing these instructions means that the unit is working fine.

Results:

As it appears in the following figure, the x10 and x3 not changed in value.



Bonus feature

Integer Multiplication and division (M) extension added to the supported instructions. The newly supported instructions are:

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Instruction Format for M-extension Instructions

Testing: the following code are put to test the M extension instructions.

```
addi x1,x0,17
addi x2,x0,9
addi x3,x0,25
sw x1, 200(x0)
sw x2, 204(x0)
sw x3, 208(x0)
add x0, x0, x0
lw x1, 200(x0)
lw x2, 204(x0)
lw x3, 208(x0)
or x4, x1, x2
beq x4, x3, l1
add x10, x3, x0
add x3, x1, x2
l1:add x5, x3, x2
sw x5, 12(x0)
lw x6, 12(x0)
```

and x7, x6, x1

sub x8, x1, x2

add x0, x1, x2

add x9, x0, x1

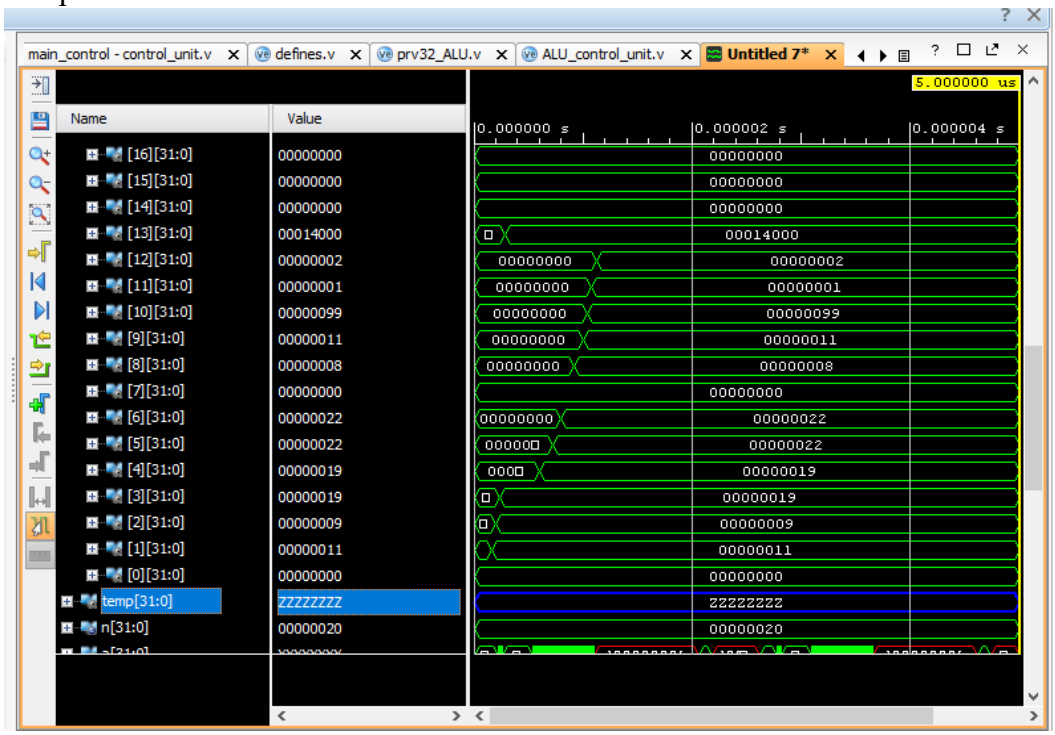
mul x10,x1,x2

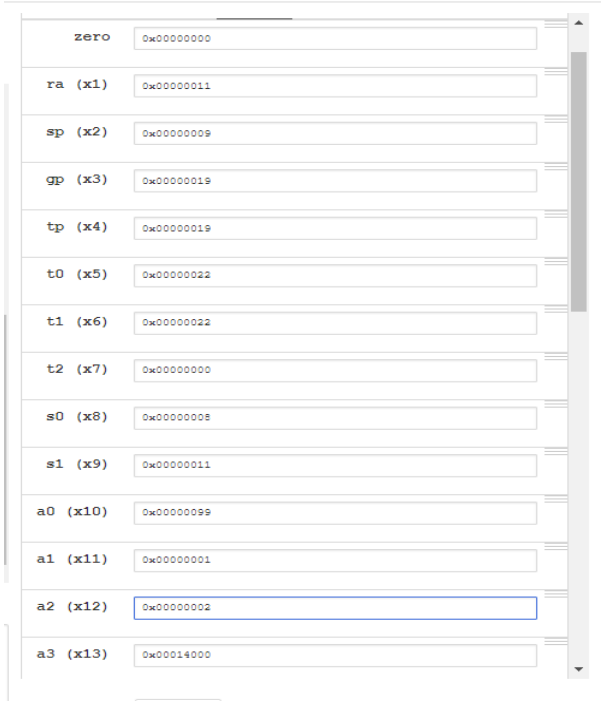
mulh x11,x1,x3

div x12,x3,x2

The last 3 instructions are put to test M instructions In the middle of a program. Correct results put in x10,x11,x12 means that the extension is working fine.

Results: as it appears in the following figure, the x10, x11, x12 have the correct values compared with Venus's simulator values




zero 0x00000000
ra (x1) 0x00000011
sp (x2) 0x00000009
gp (x3) 0x00000019
tp (x4) 0x00000019
t0 (x5) 0x00000022
t1 (x6) 0x00000022
t2 (x7) 0x00000000
s0 (x8) 0x00000008
s1 (x9) 0x00000011
a0 (x10) 0x00000099
a1 (x11) 0x00000001
a2 (x12) 0x00000002
a3 (x13) 0x00014000

FPGA Implementation and testing

The project is Synthesized, Implemented and Bitstream generated and put on the FPGA and Tested.

The following code is tested to ensure that the same results appear in the FPGA and the simulator.

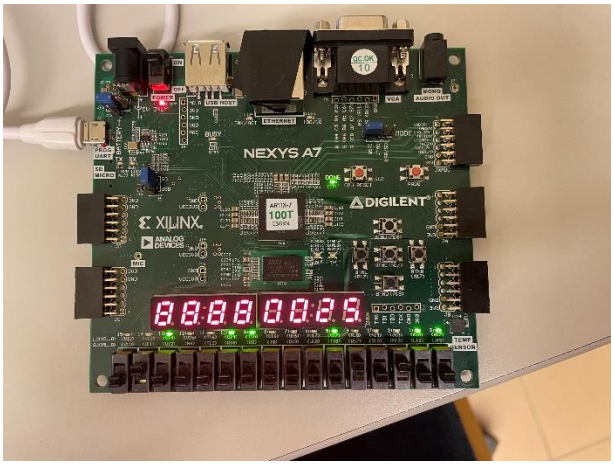
```
add x0, x0, x0
addi x1,x0,17
addi x2,x0,9
addi x3,x0,25
sw x1, 200(x0)
sw x2, 204(x0)
sw x3, 208(x0)
add x0, x0, x0
lw x1, 200(x0)
lw x2, 204(x0)
lw x3, 208(x0)
or x4, x1, x2          <<<<<<<INSTRUCTION MONITORED>>>>>>>>
beq x4, x3, 16
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x3, x1, x2
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x5, x3, x2
sw x5, 12(x0)
lw x6, 12(x0)
and x7, x6, x1
sub x8, x1, x2
add x0, x1, x2
```

add x9, x0, x1

Results: as it appears in the following figures, the leds and 7-segment driver generate correct results.

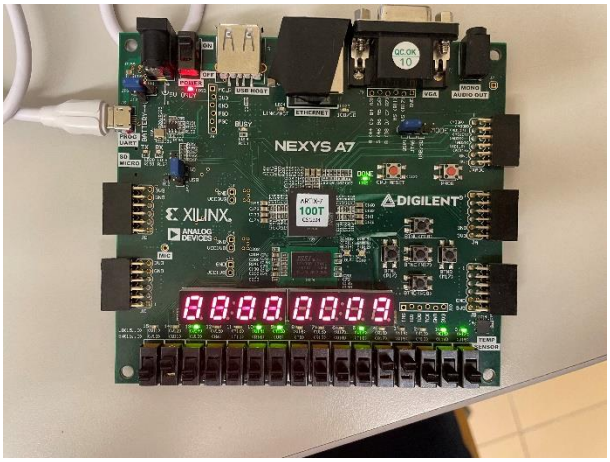
LedSel = 00

ssdSel = 1010



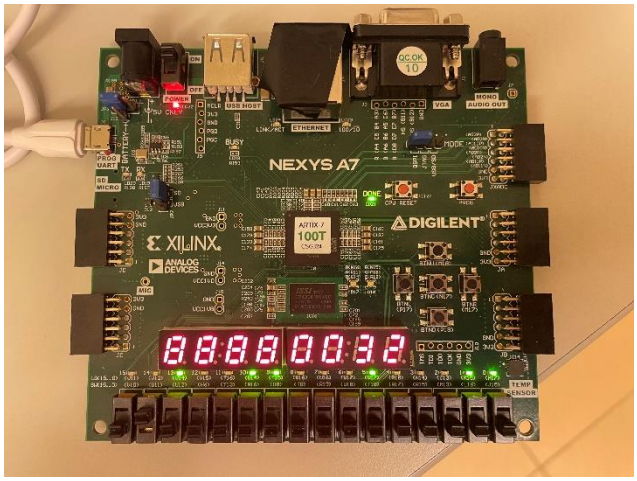
LedSel = 00

ssdSel = 0101



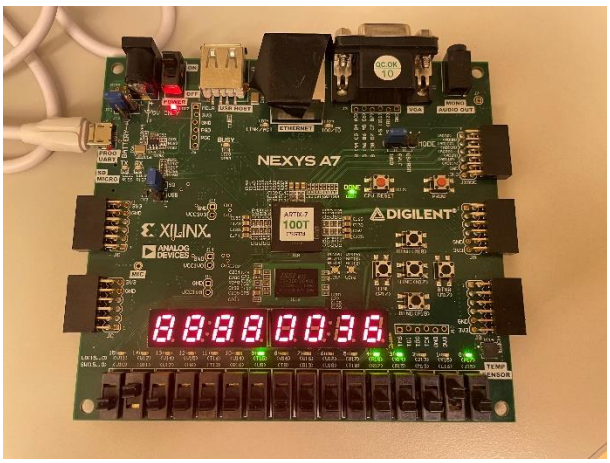
LedSel = 00

ssdSel = 0000



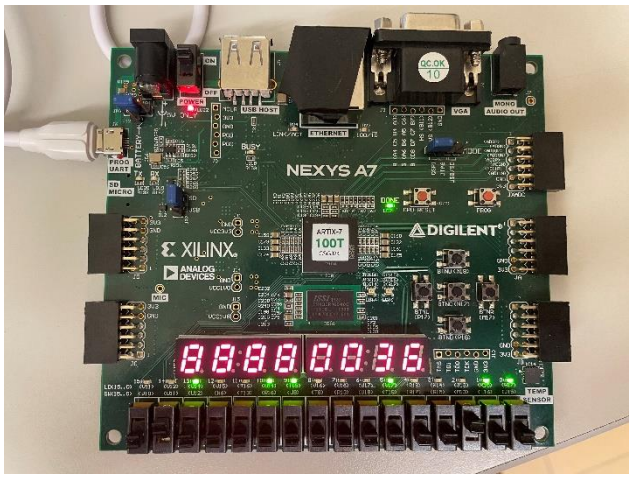
LedSel = 01

ssdSel = 0001



LedSel = 00

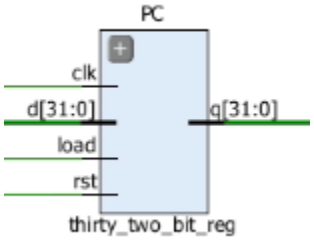
ssdSel = 0001



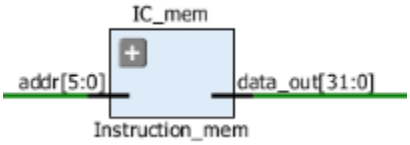
Conclusion

This milestone implements a single cycle RISCV processor that supports all 40 user-level instructions. The implementation was done in Verilog, and was debugged and tested on Vivado software. The instructions were divided into six format groups with similar functionality, and each format was designed separately using tailored Verilog modules. In this milestone, we have assembled and tested the whole processor on the FPGA, and the results were recorded and verified.

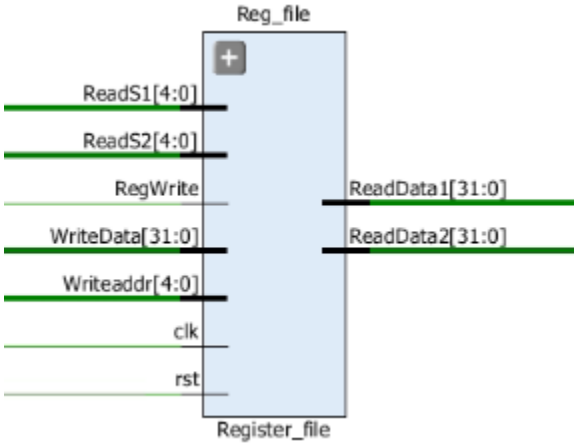
Appendix



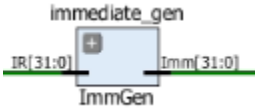
Program Counter



Instructions memory

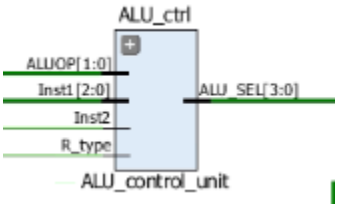


Register File

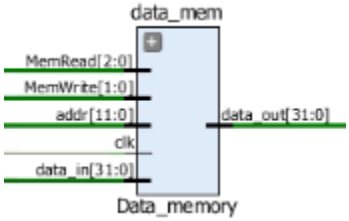


Immediate generator

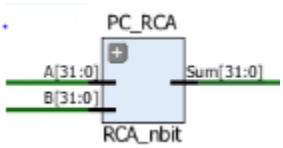
ALU



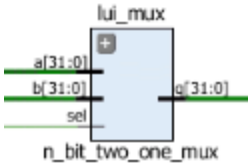
ALU control unit



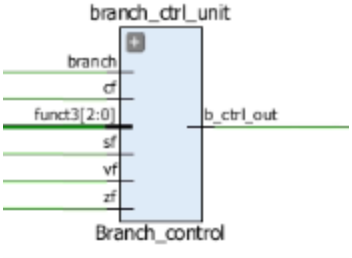
Data memory



RCA



2x1 Mux



Branch control unit