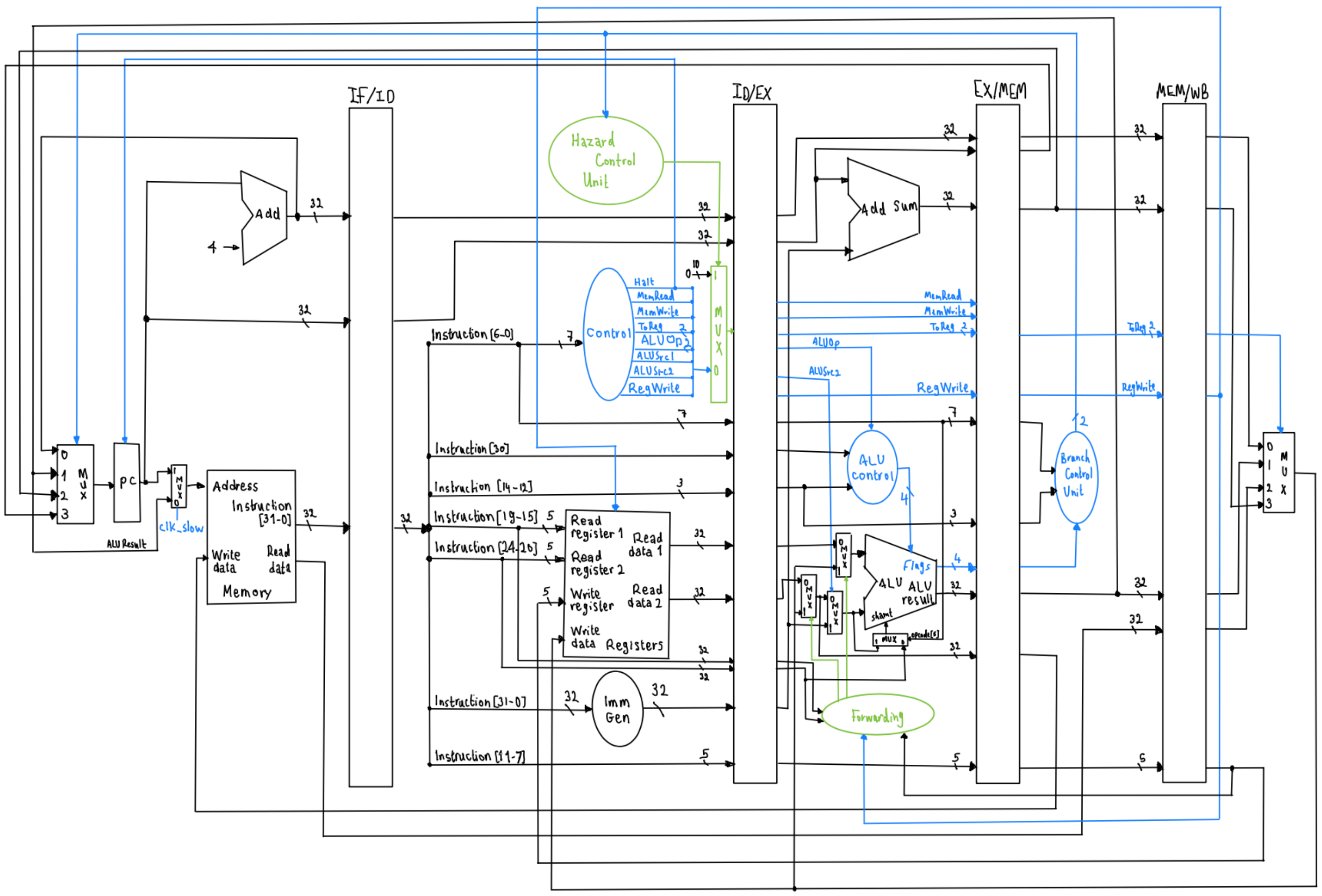
Project 1: MS3 & MS4: Pipelined Datapath for all RV-32I Instructions

Mina Ashraf Gamil 900182973
Sherif Hisham Gabr 900183120

# Report

- <u>Final Datapath:</u>



- <u>Design Decisions:</u>

  - The datapath is now pipelined. All relevant wires of the instruction are propagated with the instruction to five stages: IF, ID, EX, MEM, WB. Each stage performs different tasks. Its main purpose is to increase instruction throughput by executing several instructions simultaneously in different stages.
  - Since we are now working with a pipeline, the write back stage should write its data to the corresponding register to its instruction; in order to achieve this, the

destination register as well as the WB controls are propagated along the pipeline till the end.

- A single-ported memory module is implemented, which contains the instructions and any data the program requires. A single-ported memory introduces structural hazards because the memory module is accessed while fetching instructions (IF stage) and while reading/storing data (MEM stage). To eliminate the structural hazard, the fetching of instructions is to be done on a slower clock. The slower clock has twice the period of the normal clock, so fetching occurs every 2 normal clock cycles. In that case, there is a maximum of 3 instructions simultaneously in the pipeline, which also means that now two instructions will access the memory at the same instance.
  The input address to the memory is a MUX that selects the PC when the slow clock is one and selects the address computed from instruction when the slow clock is 0. The memory has 2 outputs, which are the instruction and the data. When fetching the instruction, it will output the instruction otherwise it will output NOP. Likewise, when reading data, it will output the data otherwise it will output 0. It depends on clock slow and MemRead respectively and both are never 1 at the same time.

- Forwarding is used to eliminate data hazards between successive instructions. Since fetching is done every 2 clock cycles. Forwarding occurs only if the preceding instruction writes to a register being used in the current instruction. The preceding instruction will be in the WB stage by the time the current instruction is in EX stage. Therefore the condition to forward :
  ```
  MEM/WB.rd != 0 &&
  MEM/WB.RegWrite &&
  (ID/EX.rs1 == MEM/WB.rd || ID/EX.rs2 == MEM/WB.rd)
  ```
  The forwarding selects the inputs to the ALU and propagates the value to be stored in memory. If the condition to forward is true, then one of the inputs to the ALU will be write_data of the preceding instruction depending on rs1 and rs2 (ForwardA or ForwardB, respectively).

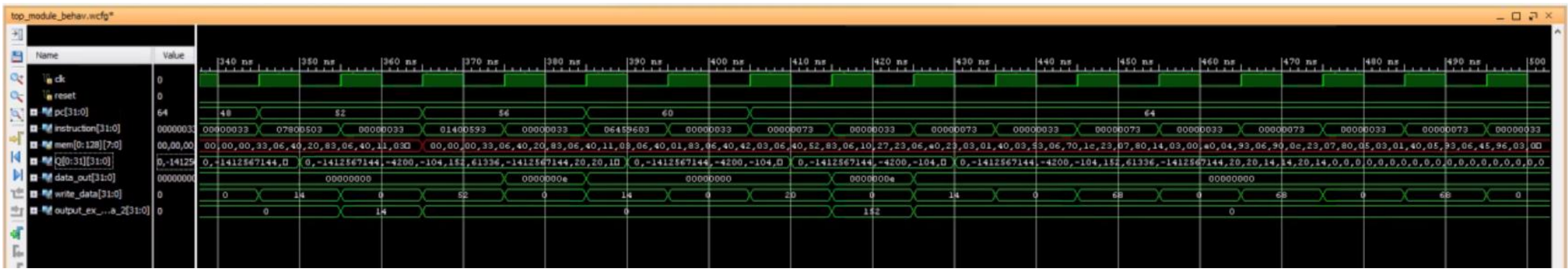
- Flushing is used to eliminate control hazards caused by branch and JAL/JALR instructions. The outcome of a branch instruction is decided in the MEM stage, which means that one instruction will be fetched already and in the ID stage. So if the branch turns out to be taken, then it must be flushed. Therefore, the Hazard Control Unit takes as input the PC selection and if the PC selection is anything but PC+4 (00), then a flag is set to flush the instruction. Flushing the instruction simply means setting all their control lines to 0. Therefore, a MUX is used to select between the instruction's control lines and 0 depending on the flag outputted by the Hazard Control Unit. The output of this MUX is sent to the next stage in the pipeline. Also note that, JAL/JALR instructions are the same as branch instructions, however, it is known that it is always taken, therefore it always flushes the instruction.

- ECALL is used as a terminating instruction, where it halts the PC from being updated. Its main purpose is to be used at the end of any program to stop the PC from fetching even after there are no more instructions. When an ECALL instruction is executed, it is determined in the ID stage and sets the PC load to 0 which means the PC cannot be updated anymore.

## Testing and simulation of all instructions

**Note:** all testing programs written here are traced with the correct values that should appear if the program was executed correctly. All the following screenshots show 100% match between the expected results and the actual output of the simulation which proves that the processor is functioning as expected. Corner test case that include forwarding, hazard handling, flushing, multiple branches, double data hazards, and much more were tested, and they are all working fine.

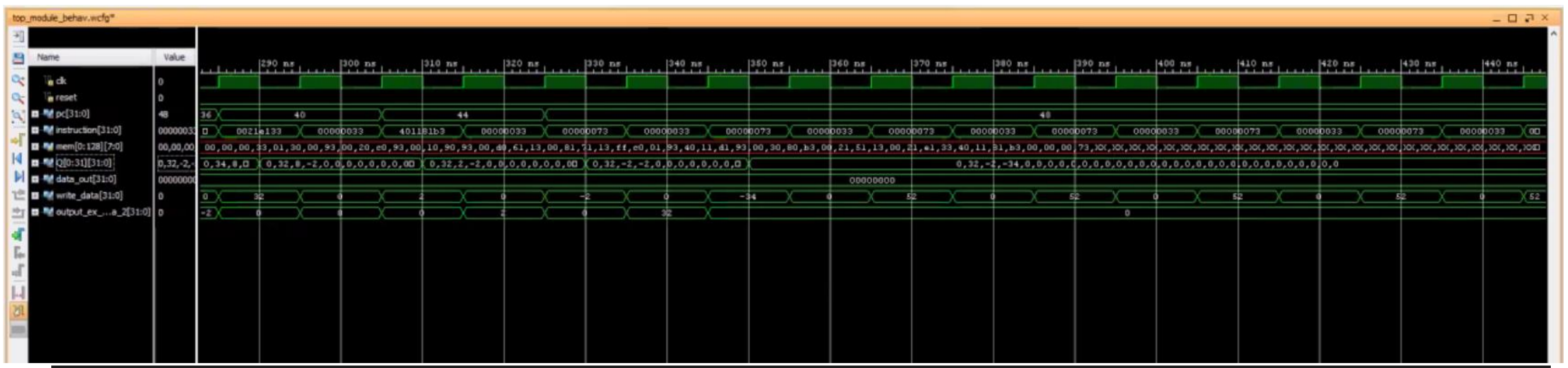
### Load and store instructions



```
#Loading and Storing Instructions

lw      x1, 100(x0)     #10101011_11001101_11101111_10011000 => -1412567144

lh      x2, 100(x0)     #11111111_11111111_11101111_10011000 => -4200

lb      x3, 100(x0)     #11111111_11111111_11111111_10011000 => -104

lbu     x4, 100(x0)     #00000000_00000000_00000000_10011000 => 152

lhu     x5, 100(x0)     #00000000_00000000_11101111_10011000 => 61336


sw  x1, 110(x0)

lw  x6, 110(x0) #10101011_11001101_11101111_10011000 => -1412567144


addi    x7, x0, 20

sh  x7, 120(x0)         #forwarding

lh  x8, 120(x0)


addi    x9, x0, 14
```

```
sb   x9, 120(x0)           #forwarding

lb   x10, 120(x0)


addi    x11, x0, 20

lh   x12, 100(x11)           #forwarding

# FINAL VALUES

#        x1       ---> -1412567144

#        x2       ---> -4200

#        x3       ---> -104

#        x4       ---> 152

#        x5       ---> 61336

#        x6       ---> -1412567144

#        x7       ---> 20

#        x8       ---> 20

#        x9       ---> 14

#        x10      ---> 14

#        x11      ---> 20

#        x12      ---> 14
```

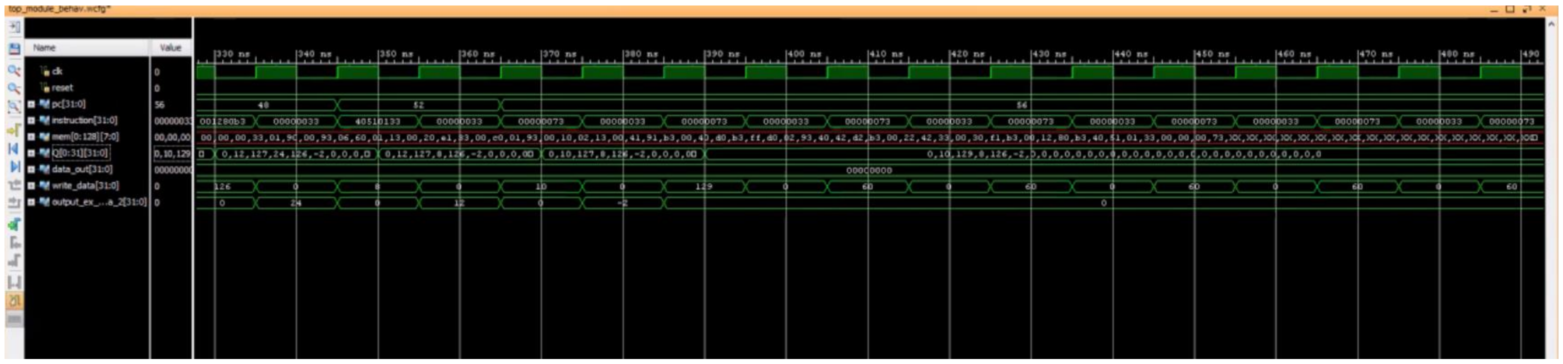
## ALU operations 1



```
#ALU Operation instructions (with hazards)

addi     x1, x0, 19   #x1 = 10011 => 19

xori     x1, x1, 2    #x1 = 10001 => 17                   (forwarding)
```

```
slli    x1, x1, 1   #x1 = 00000000_00000000_00000000_00100010 => 34     (forwarding)


ori x2, x0, 13  #x2 = 01101 => 13

andi    x2, x2, 8   #x2 = 01000 => 8                        (forwarding)


addi    x3, x0, -4  #x3 = 11111111_11111111_11111111_11111100 => -4

srai    x3, x3, 1   #x3 = 11111111_11111111_11111111_11111110 => -2     (forwarding)

add x1, x1, x3  #x1 = 00000000_00000000_00000000_00100000 => 32     (forwarding)

srli    x2, x2, 2   #x2 = 00000000_00000000_00000000_00000010 => 2

or  x2, x3, x2  #x2 = 111111111_11111111_11111111_1111110 => -2     (forwarding)

sub x3, x3, x1  #x3 = 11111111_11111111_11111111_11011110 => -34

ecall

# FINAL VALUES

#   x1  ---> 32

#   x2  ---> -2

#   x3  ---> -34
```

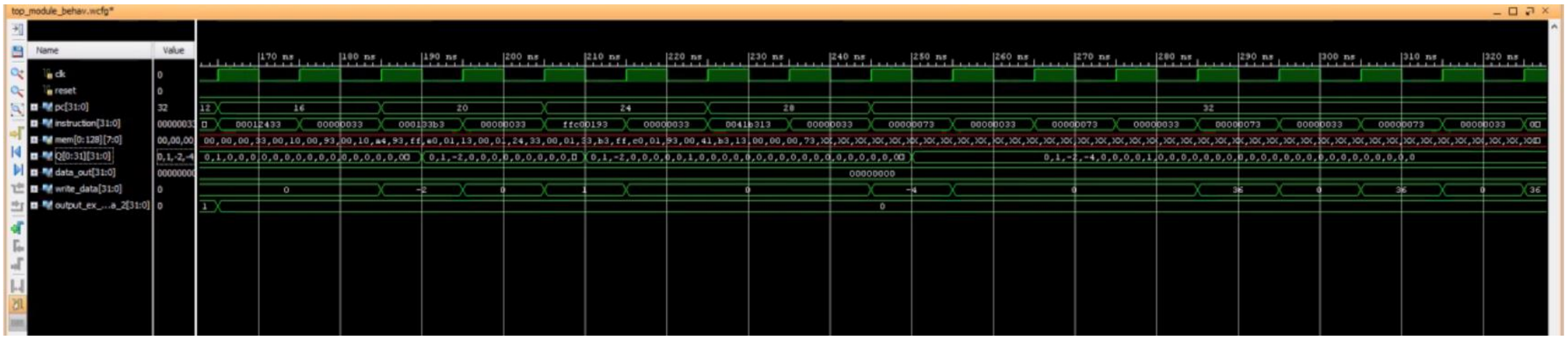
## Alu operations 2



#some ALU Operation instructions (with hazards)

```
addi    x1, x0, 25  #x1 = 00000000_00000000_00000000_00011001 => 25

addi    x2, x0, 102 #x2 = 00000000_00000000_00000000_01100110 => 102

or  x2, x1, x2  #x2 = 00000000_00000000_00000000_01111111 => 127    (forwarding)



addi    x3, x0, 12  #x3 = 00000000_00000000_00000000_00001100 => 12

addi    x4, x0, 1   #x4 = 00000000_00000000_00000000_00000001 => 1

sll x3, x3, x4  #x3 = 00000000_00000000_00000000_00011000 => 24     (forwarding)

srl x1, x1, x4  #x1 = 00000000_00000000_00000000_00001100 => 12



addi    x5, x0, -3  #x5 = 11111111_11111111_11111111_11111101 => -3

sra x5, x5, x4  #x5 = 11111111_11111111_11111111_11111110 => -2     (forwarding)

xor x4, x4, x2  #x4 = 00000000_00000000_00000000_01111110 => 126

and x3, x1, x3  #x3 = 00000000_00000000_00000000_00001000 => 8

add x1, x5, x1  #x1 = 00000000_00000000_00000000_00001010 => 10

sub x2, x2, x5  #x2 = 00000000_00000000_00000000_10000001 => 129

ecall

# FINAL VALUES

#   x1  ---> 10

#   x2  ---> 129

#   x3  ---> 8

#   x4  ---> 126

#   x5  ---> -2
```

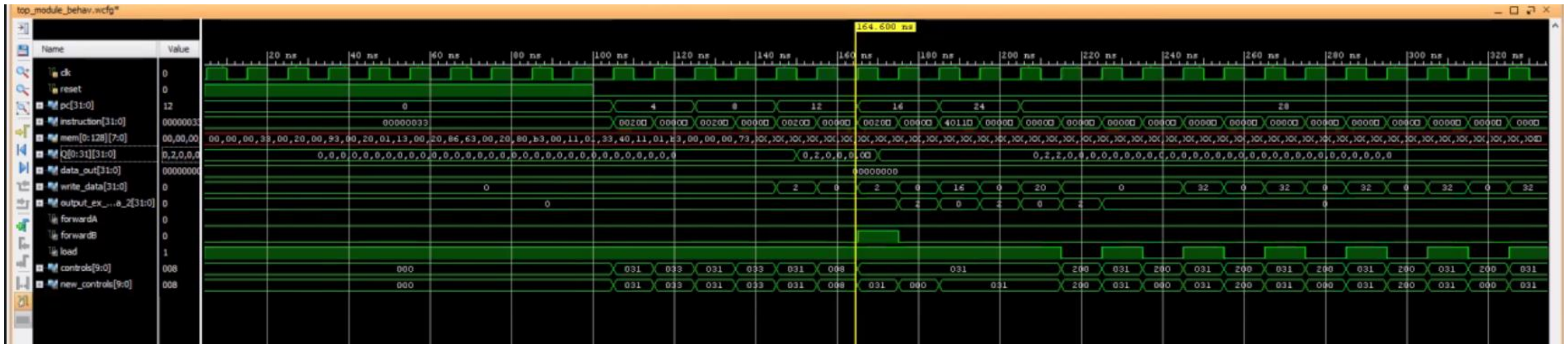
## Alu operations 3



| Name | Value |
|---|---|
| clk | 0 |
| reset | 0 |
| pc[31:0] | 32 |
| instruction[31:0] | 00000033 |
| mem[0:128][7:0] | 00,00,00 |
| Q[0:31][31:0] | 0,1,-2,-4 |
| data_out[31:0] | 00000000 |
| write_data[31:0] | 0 |
| output_ex_...a_2[31:0] | 0 |

```
#SLT instructions (with hazards)

addi    x1, x0, 1   #x1 = 00001 => 1

slti    x9, x1, 1   #x9 = 00000 => 0                        (forwarding)

addi    x2, x0, -2  #x2 = 11111111_11111111_11111111_11111110 => -2

slt     x8, x2, x0  #x8 = 00000000_00000000_00000000_00000001 => 1      (forwarding)

sltu    x7, x2, x0  #x7 = 00000000_00000000_00000000_00000000 => 0


addi    x3, x0, -4  #x3 = 11111111_11111111_11111111_11111100 => -4

sltiu   x6, x3, 4   #x6 = 00000000_00000000_00000000_00000000 => 0      (forwarding)

ecall

#FINAL VALUES

#   x1  ---> 1

#   x2  ---> -2

#   x3  ---> -4

#   x6  ---> 0

#   x7  ---> 0

#   x8  ---> 1

#   x9  ---> 0
```

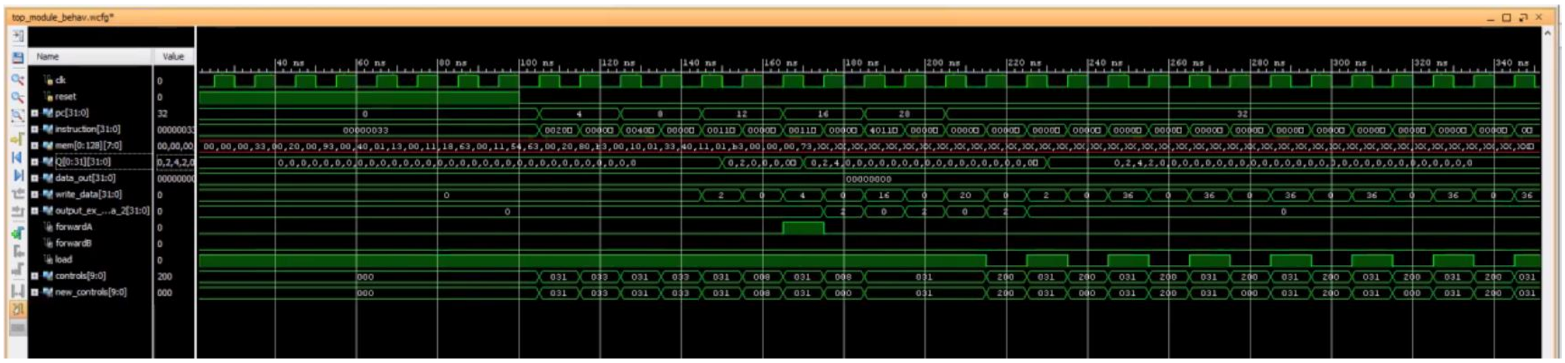
## Branch if Equal testing



```
#BEQ instruction
```

```
addi    x1, x0, 2        #x1 = 0010 => 2

addi    x2, x0, 2        #x2 = 0010 => 2

beq x1, x2, jump         #branch is taken        (forwarding)

add x1, x1, x2           #x1 = 0100 => 4     (flush instructions)

add x2, x2, x1           #x2 = 0110 => 6         (forwarding)


jump:

sub x3, x2, x1           #x3 = 0000 => 0     (if branch taken)

                #   = 0010 => 2     (if branch not taken)

ecall


#FINAL VALUES

#   x1  ---> 2

#   x2  ---> 2

#   x3  ---> 0
```

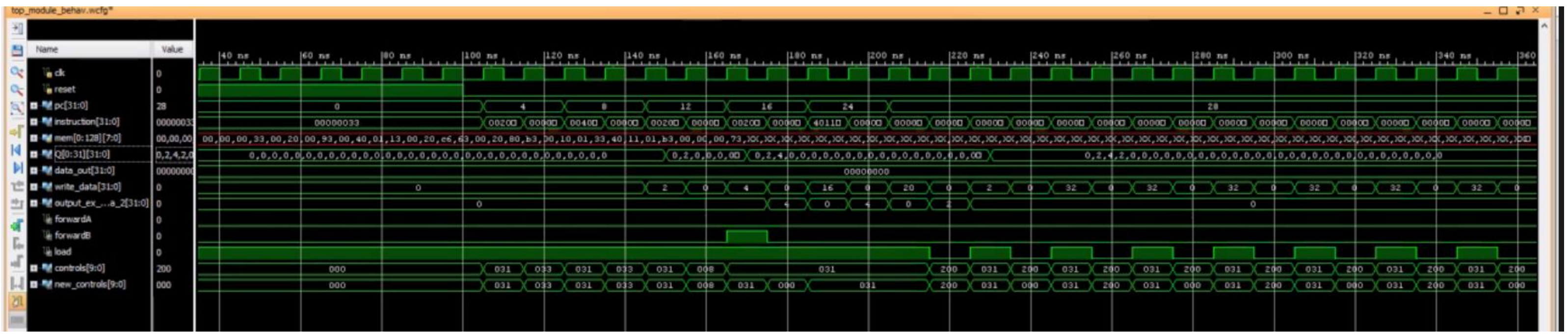
## BNE



‘

```
#BNE instruction

addi    x1, x0, 2        #x1 = 0010 => 2
addi    x2, x0, 4        #x2 = 0100 => 4

bne x2, x1, jump         #branch is taken        (forwarding)

    bge x2, x1, jump2            #(flush instructions)

    add x1, x1, x2           #x1 = 0110 => 6

    jump2:

    add x2, x0, x1           #x2 = 0110 => 6         (forwarding)


jump:


sub x3, x2, x1           #x3 = 0000 => 2      (if branch taken)

                #   = 0010 => 0      (if branch not taken)

ecall


#FINAL VALUES

#   x1  ---> 2

#   x2  ---> 4

#   x3  ---> 2
```

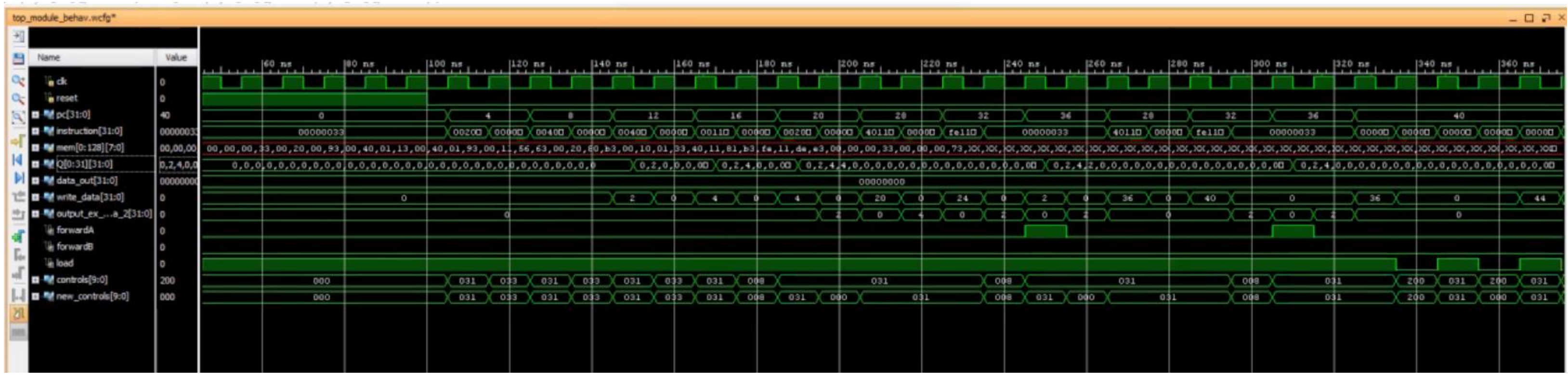
## BLT



```
#BLT instruction
```

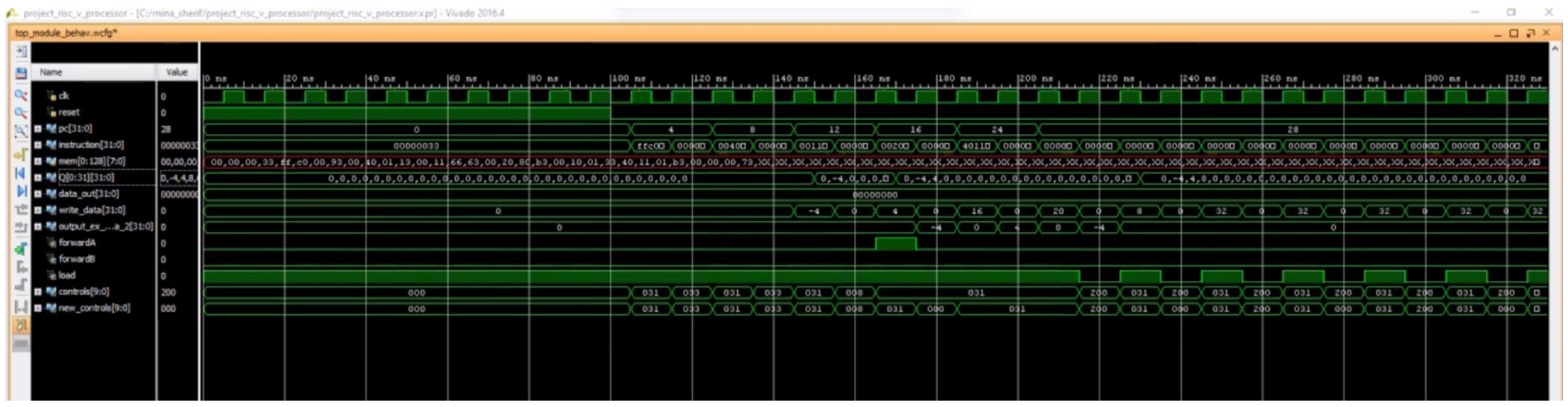
```
addi    x1, x0, 2        #x1 = 0010 => 2

addi    x2, x0, 4        #x2 = 0100 => 4

blt x1, x2, jump         #branch is taken        (forwarding)


add x1, x1, x2           #x1 = 0110 => 6     (flush instructions)

add x2, x0, x1           #x2 = 0110 => 6         (forwarding)


jump:


sub x3, x2, x1           #x3 = 0000 => 2     (if branch taken)

                #   = 0010 => 0     (if branch not taken)

ecall


#FINAL VALUES

#   x1  ---> 2

#   x2  ---> 4

#   x3  ---> 2
```

## BGE



```
#BGE instruction

addi    x1, x0, 2        #x1 = 0010 => 2

addi    x2, x0, 4        #x2 = 0100 => 4
addi    x3, x0, 4         #x3 = 0100 => 4
bge x2, x1, jump          #branch is taken        (forwarding)
add x1, x1, x2            #x1 = 0110 => 6     (flush instructions)
add x2, x0, x1            #x2 = 0110 => 6         (forwarding)
       jump:
sub     x3, x3, x1        #x3 = 0010 => 2     (if branch taken)
                #   = 1110 => -2     (if branch not taken)

                #x3 = 0000 => 0     (if 2nd branch taken)

bge x3, x1, jump          #branch taken if initial branch is taken
add x0, x0, x0
ecall

#FINAL VALUES
#   x1  ---> 2
#   x2  ---> 4
#   x3  ---> 0
```

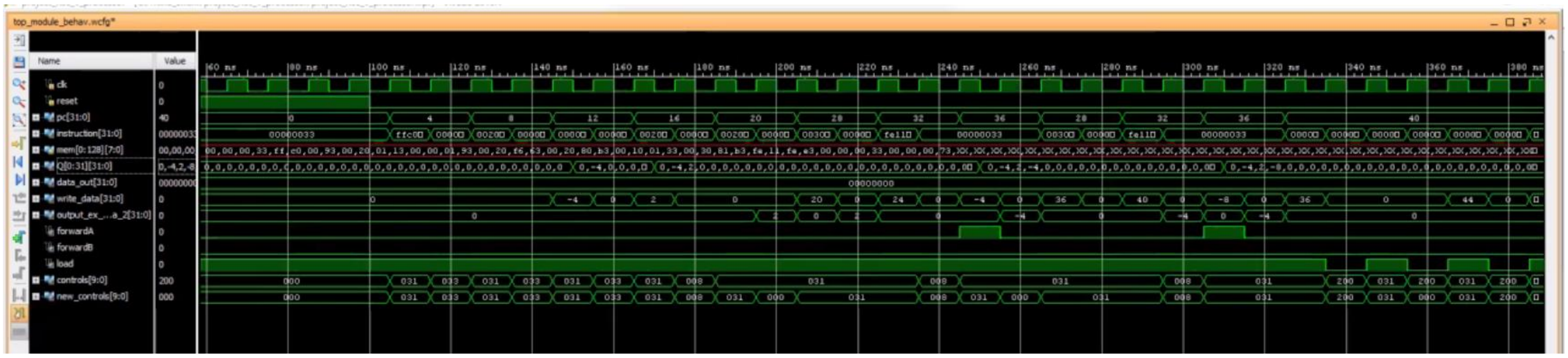
## BLTU



```
#BLTU instruction
```

```
addi    x1, x0, -4       #x1 = 1100 => -4

addi    x2, x0, 4        #x2 = 0100 => 4

bltu    x2, x1, jump         #branch is taken        (forwarding)


add x1, x1, x2           #x1 = 0000 => 0     (flush instructions)

add x2, x0, x1           #x2 = 0000 => 0         (forwarding)

jump:

sub x3, x2, x1           #x3 = 1000 => 8     (if branch taken)

                #   = 0000 => 0     (if branch not taken)

ecall


#FINAL VALUES

#   x1  ---> -4

#   x2  ---> 4

#   x3  ---> 8
```

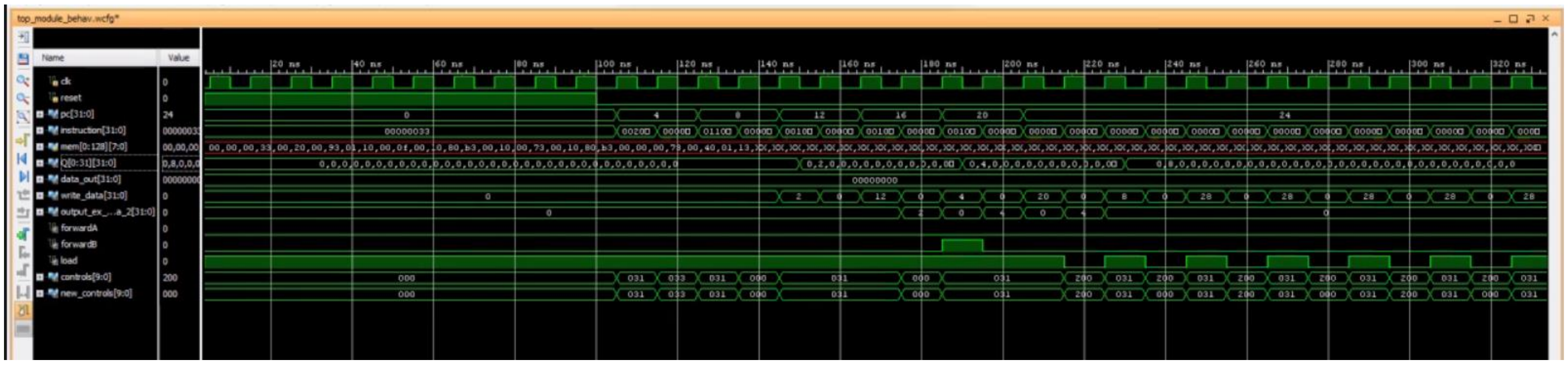
## BGEU



```
#BGEU instruction

addi    x1, x0, -4       #x1 = 1100 => -4

addi    x2, x0, 2        #x2 = 0010 => 2

addi    x3, x0, 0        #x3 = 0000 => 0
```

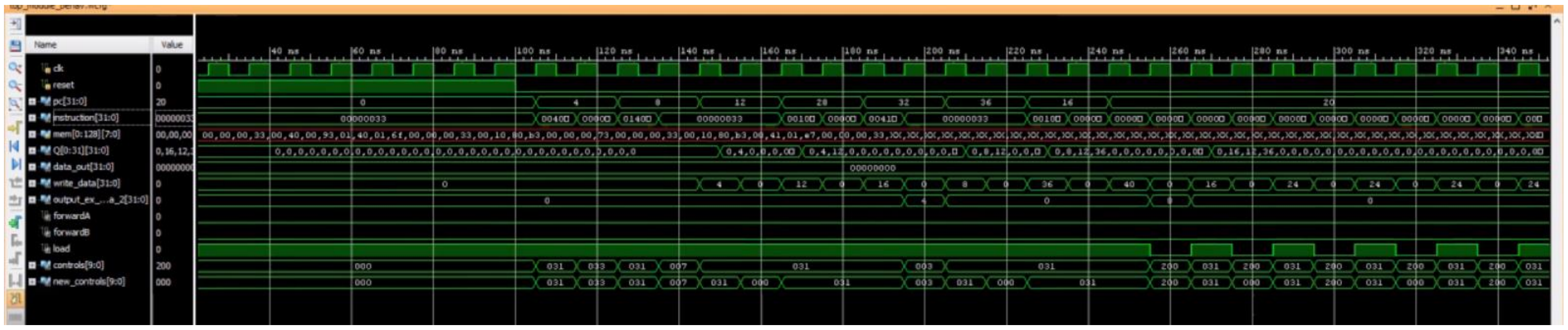
```
bgeu    x1, x2, jump         #branch is taken        (forwarding)


add x1, x1, x2          #x1 = 1110 => -2    (flush instructions)

add x2, x0, x1          #x2 = 1110 => -2            (forwarding)


jump

add     x3, x1, x3      #x3 = 1100 => -4    (if branch taken)

                #   = 1110 => -2     (if branch not taken)


                #x3 = 1000 => -8     (if 2nd branch taken)


bgeu    x3, x1, jump         #branch taken if initial branch is taken

add x0, x0, x0

ecall


#FINAL VALUES

#   x1   ---> -4

#   x2   ---> 2

#   x3   ---> -8
```

## Fence ebreak ecall



```
addi x1, x0, 2

fence 1, 1

add x1, x1, x1

ebreak

add x1, x1, x1

ecall

addi x2, x0, 4


# FINAL VALUES

#   x1  ---> 8

#   x2  ---> 0
```

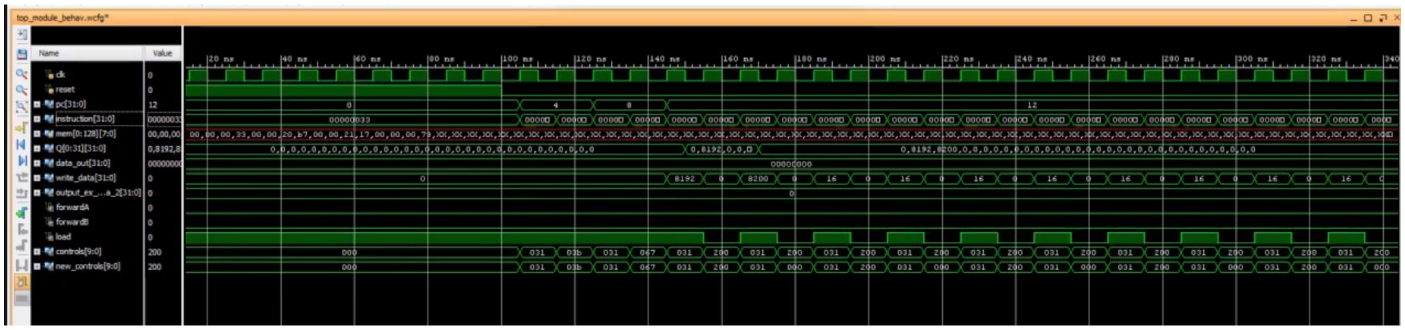
## Jalr and jal



```
#JAL isntruction


addi    x1, x0, 4

jal     x2, jump

add x0, x0, x0

add x1, x1, x1

ecall
```

```
add x0, x0, x0
jump:
add     x1, x1, x1
jalr    x3, x2, 4

add x0, x0, x0
# FINAL VALUES
#   x1  ---> 16
#   x2  ---> 12
#   x3  ---> 36
```

## LUI and AUIPC



```
#LUI and AUIPC instructions
lui     x1, 2
auipc   x2, 2
ecall
# FINAL VALUES
#   x1  ---> 10_000000000000 => 8192
#   x2  ---> 10_000000001000 => 8200
```

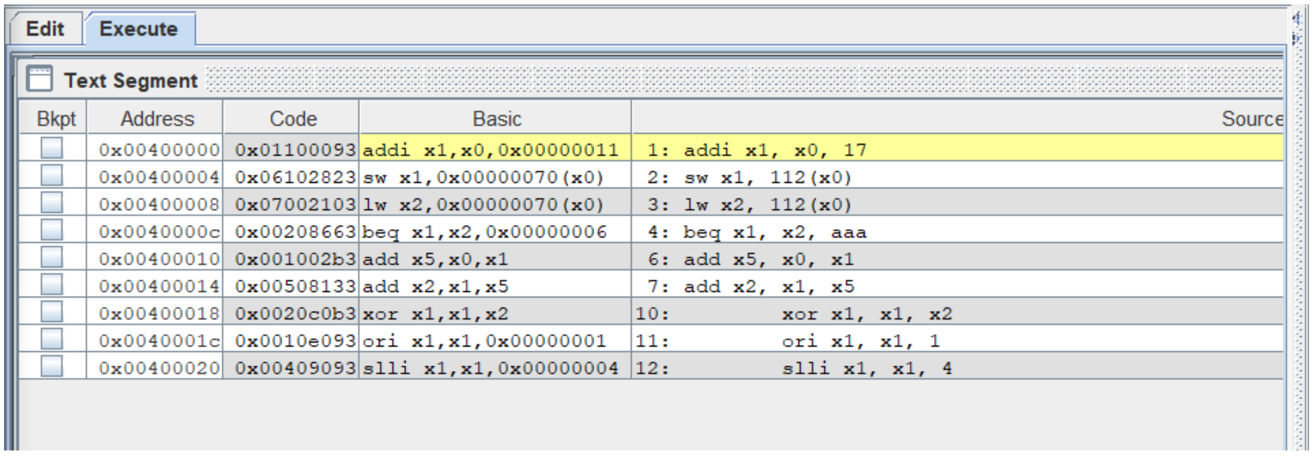
FPGA testing:

There was a mismatch between the simulation and the fpga synthesis testing. We have contacted Dr. Cherif and he sent this article

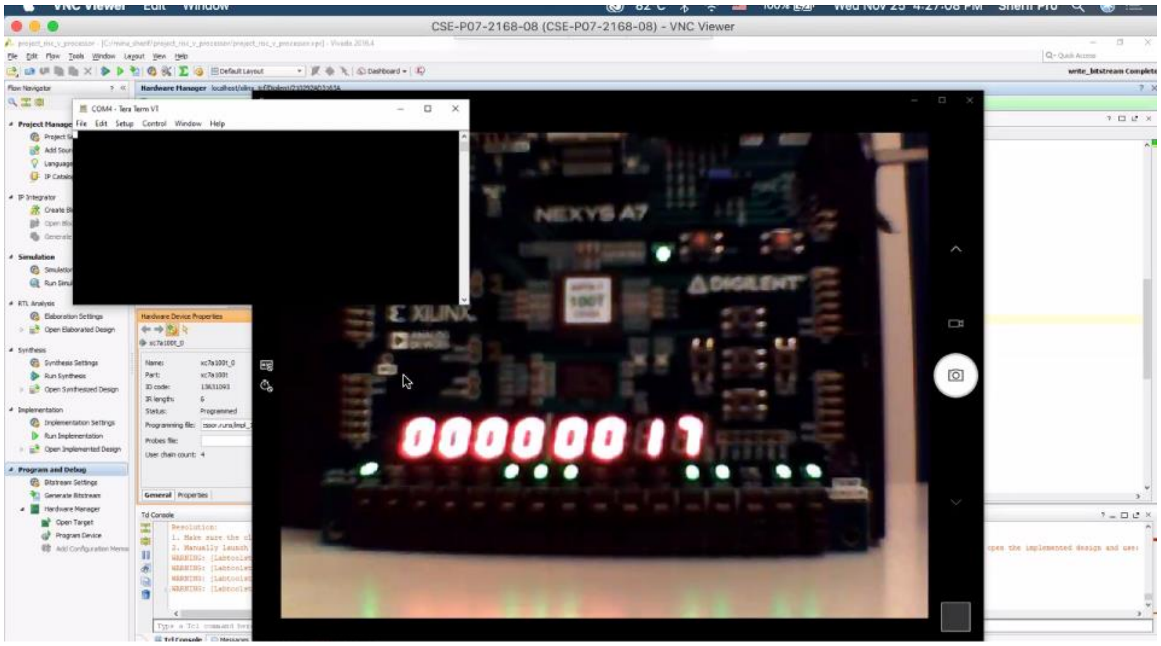
https://zipcpu.com/blog/2018/08/04/sim-mismatch.html

it explained that there might be a lot of causes for this error including the mismatching clocks. Since we use a manual input to advance the clock, we concluded that this might be the source of the error as the article suggests. It is worth noting that all supported instructions work 100% correct in the simulation.

We wrote a very simple program to show results on the FPGA

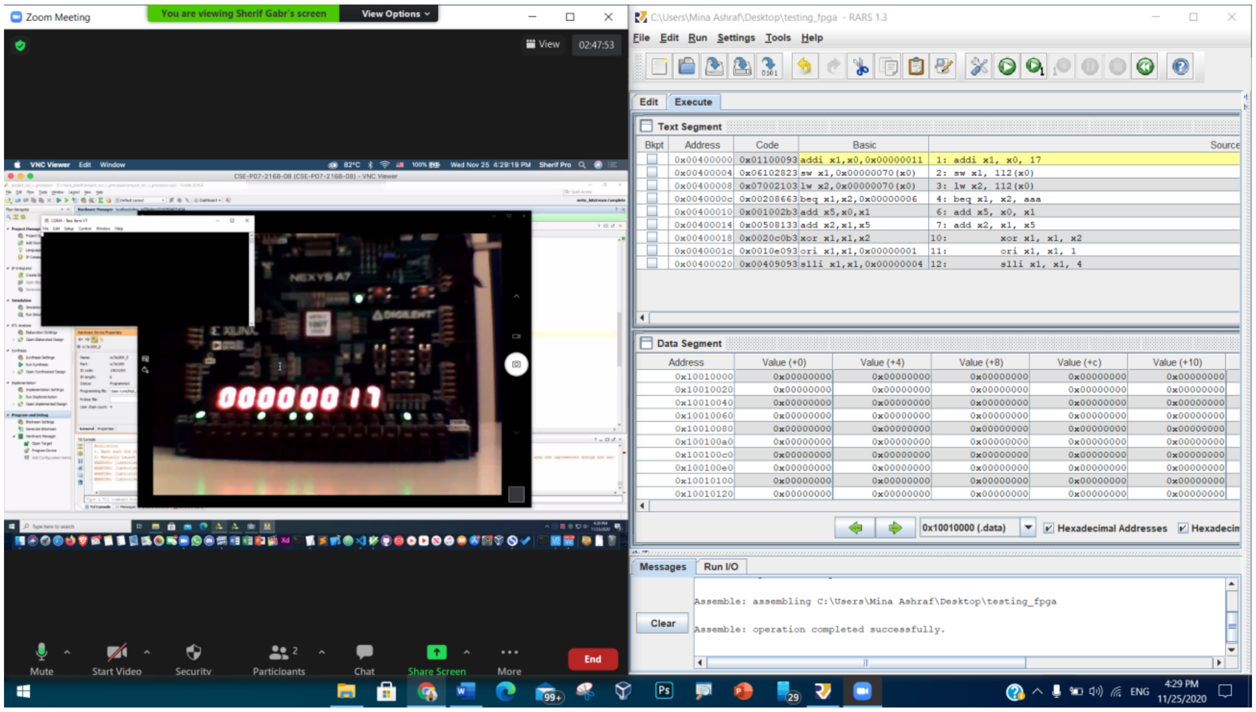


| Bkpt | Address | Code | Basic | Source |
|---|---|---|---|---|
| | 0x00400000 | 0x01100093 | addi x1,x0,0x00000011 | 1: addi x1, x0, 17 |
| | 0x00400004 | 0x06102823 | sw x1,0x00000070(x0) | 2: sw x1, 112(x0) |
| | 0x00400008 | 0x07002103 | lw x2,0x00000070(x0) | 3: lw x2, 112(x0) |
| | 0x0040000c | 0x00208663 | beq x1,x2,0x00000006 | 4: beq x1, x2, aaa |
| | 0x00400010 | 0x001002b3 | add x5,x0,x1 | 6: add x5, x0, x1 |
| | 0x00400014 | 0x00508133 | add x2,x1,x5 | 7: add x2, x1, x5 |
| | 0x00400018 | 0x0020c0b3 | xor x1,x1,x2 | 10: xor x1, x1, x2 |
| | 0x0040001c | 0x0010e093 | ori x1,x1,0x00000001 | 11: ori x1, x1, 1 |
| | 0x00400020 | 0x00409093 | slli x1,x1,0x00000004 | 12: slli x1, x1, 4 |



Here you can notice the SSD is displaying 17, this is the immediate value that is fed to the second input of the ALU.

ssdSel was 0111 which selects immGenOut which is 17.



C:\Users\Mina Ashraf\Desktop\testing_fpga - RARS 1.3

Text Segment

| Bkpt | Address | Code | Basic | Source |
|---|---|---|---|---|
| | 0x00400000 | 0x01100093 | addi x1,x0,0x00000011 | 1: addi x1, x0, 17 |
| | 0x00400004 | 0x06102823 | sw x1,0x00000070(x0) | 2: sw x1, 112(x0) |
| | 0x00400008 | 0x07002103 | lw x2,0x00000070(x0) | 3: lw x2, 112(x0) |
| | 0x0040000c | 0x00208663 | beq x1,x2,0x00000006 | 4: beq x1, x2, aaa |
| | 0x00400010 | 0x001002b3 | add x5,x0,x1 | 6: add x5, x0, x1 |
| | 0x00400014 | 0x00508133 | add x2,x1,x5 | 7: add x2, x1, x5 |
| | 0x00400018 | 0x0020c0b3 | xor x1,x1,x2 | 10: xor x1, x1, x2 |
| | 0x0040001c | 0x0010e093 | ori x1,x1,0x00000001 | 11: ori x1, x1, 1 |
| | 0x00400020 | 0x00409093 | slli x1,x1,0x00000004 | 12: slli x1, x1, 4 |

Data Segment

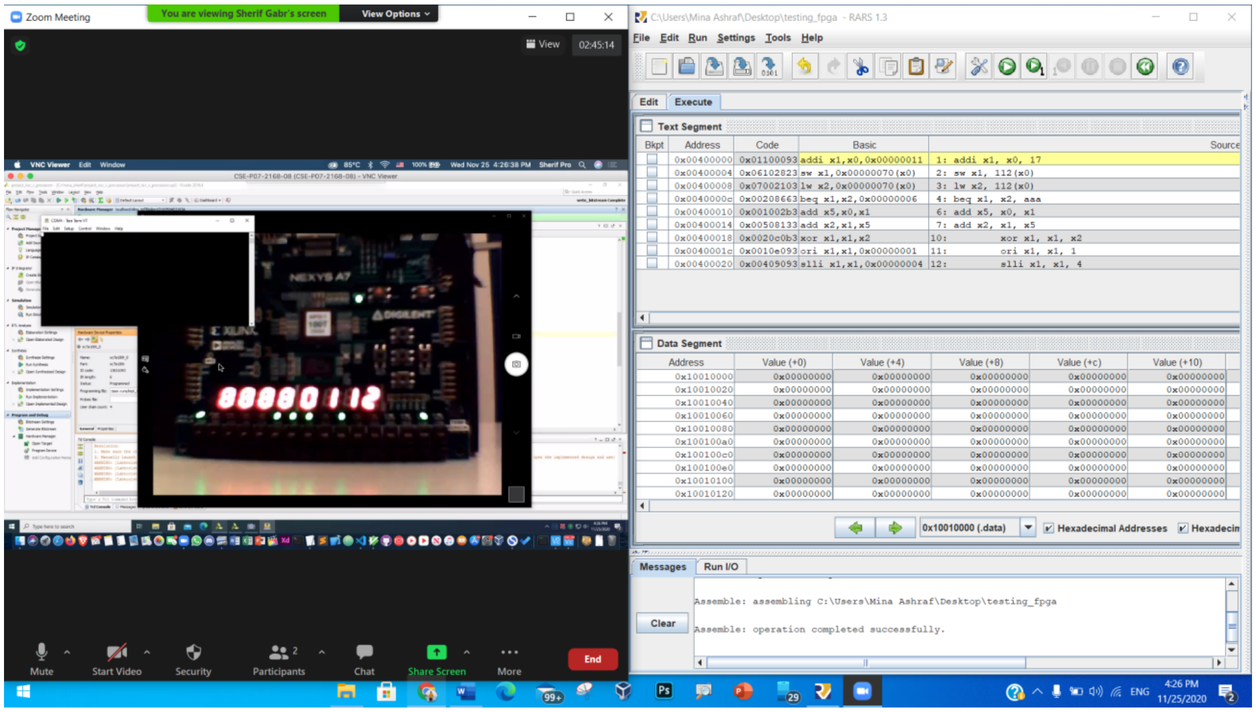
| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) |
|---|---|---|---|---|---|
| 0x10010000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010020 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010040 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010060 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010080 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100a0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100c0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100e0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010100 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010120 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |

0x10010000 (.data) ✓ Hexadecimal Addresses ✓ Hexadecim

```
Assemble: assembling C:\Users\Mina Ashraf\Desktop\testing_fpga

Assemble: operation completed successfully.
```

This is the write data of the Load word instruction,

ssdSel is 1011 which gets the write data of the Load word after 4 cycles from being fetched. It is showing 17 which proves that storing and loading are also working here.



Zoom Meeting
You are viewing Sherif Gabr's screen
View Options
02:45:14
88880 112
Mute
Start Video
Security
Participants
Chat
Share Screen
More
End
C:\Users\Mina Ashraf\Desktop\testing_fpga - RARS 1.3
File Edit Run Settings Tools Help
Edit
Execute

Text Segment

| Bkpt | Address | Code | Basic | Source |
|---|---|---|---|---|
|  | 0x00400000 | 0x01100093 | addi x1,x0,0x00000011 | 1: addi x1, x0, 17 |
|  | 0x00400004 | 0x06102823 | sw x1,0x00000070(x0) | 2: sw x1, 112(x0) |
|  | 0x00400008 | 0x07002103 | lw x2,0x00000070(x0) | 3: lw x2, 112(x0) |
|  | 0x0040000c | 0x00208663 | beq x1,x2,0x00000006 | 4: beq x1, x2, aaa |
|  | 0x00400010 | 0x001002b3 | add x5,x0,x1 | 6: add x5, x0, x1 |
|  | 0x00400014 | 0x00508133 | add x2,x1,x5 | 7: add x2, x1, x5 |
|  | 0x00400018 | 0x0020c0b3 | xor x1,x1,x2 | 10: xor x1, x1, x2 |
|  | 0x0040001c | 0x0010e093 | ori x1,x1,0x00000001 | 11: ori x1, x1, 1 |
|  | 0x00400020 | 0x00409093 | slli x1,x1,0x00000004 | 12: slli x1, x1, 4 |

Data Segment

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) |
|---|---|---|---|---|---|
| 0x10010000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010020 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010040 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010060 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010080 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100a0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100c0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100e0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010100 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010120 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |

0x10010000 (.data) ✓ Hexadecimal Addresses ✓ Hexadecim

Messages | Run I/O

```
Assemble: assembling C:\Users\Mina Ashraf\Desktop\testing_fpga

Assemble: operation completed successfully.
```

Immediate of store and load

This is the immediate offset of the store word instruction showing 112. SSDsel is 0111 which selects the immediate genout.

Branching didn't work on the FPGA due to the mismatch; however, they are all working fine on the simulation as shown above in the screenshots with the testing programs.