The American University in Cairo

Computer Architecture

Project 1 (MS4)

Pipelined scaler Implementation of RISC-V With hazards resolved

Ahmed Osama Hassan

900171850

Mohamed Ashraf Hemdan

900171923

Saturday, July 11, 2020

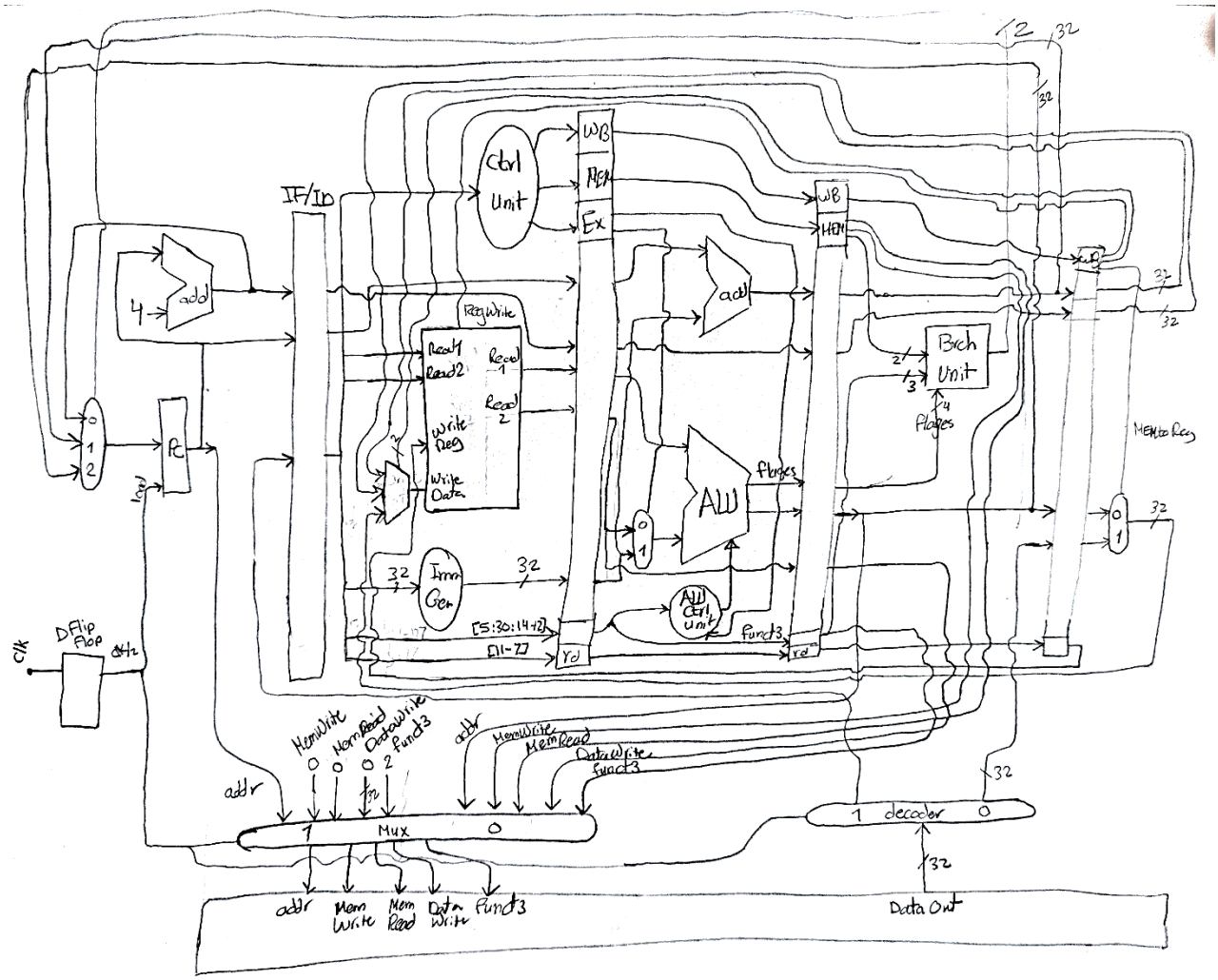
Submitted to:

Dr. Cherif Salama

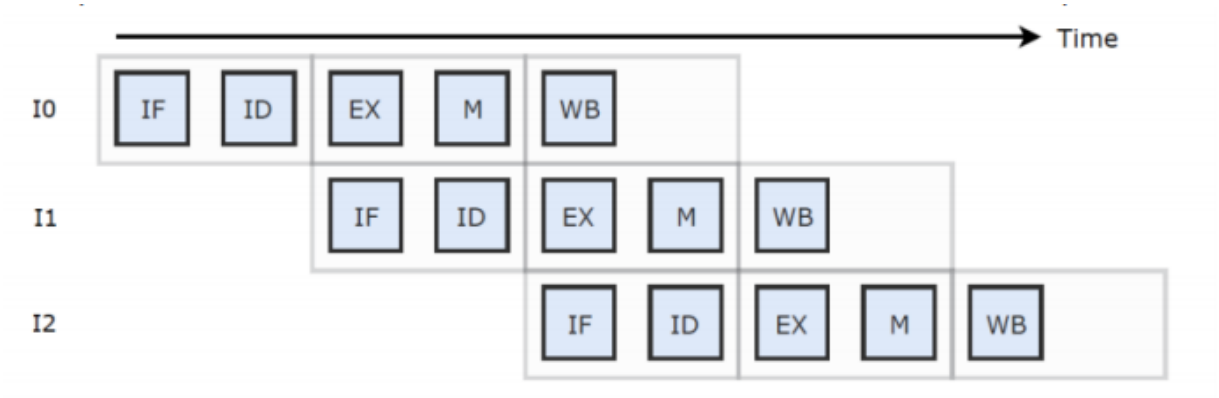
We are trying to resolve some of the hazards introduced by pipelining the single cycle design along with some additional changes

Previous design:

The design implemented before supports pipelining but cannot resolve the data dependency problems along with some control hazards. They were solved by adding 3 nope instructions between each consecutive instruction.



This design was capable only of fetching instructions every two cycles since we use one memory only and therefore it needs to be accessed one time in a cycle one time for fetching instruction and another one for data memory accessing as shown below



Design Challenges

1- Data dependencies
For any two consecutive instructions of Read After Write (RAW) dependency, the pipeline must stall for 3 cycles. The previous design does not even support stalling.

2- Structural Hazards (as a bonus) *
Because we use one data memory. It is required to fetch every two cycles. However, we want to make our processor runs normally and fetches every cycle. This introduced a structural hazard as we want to use the memory for two purposes at the same time

3- Control Hazards
Branch instructions requires the processor to flush the following 3 instructions in case the branch is taken. This was not implemented in the previous design.

4- EBreak instruction
This instruction was supposed to be included so that the program can terminate properly without fetching non-initialized memory words or even non instructions.

Design Changes and solutions:

1- Forwarding between instructions
For RAW dependency, forwarding enables instructions within the pipeline to forward the ALU result for example to the input of the ALU in the next cycle or the output of the memory in the MEM/WB register to the input of the ALU. This change solved all data dependencies except the load use ones. The using instruction must stall for one cycle at least to enable forwarding from MEM/WB data memory out to the input of the ALU. This solved in the next design solution

2- Hazard Detection Unit
This unit detects the hazard introduced by the load use data dependency. It did some comparisons to make sure it's a load use dependency and then sends a stall signal to the IF_ID registers and the pc to not load a new instruction.

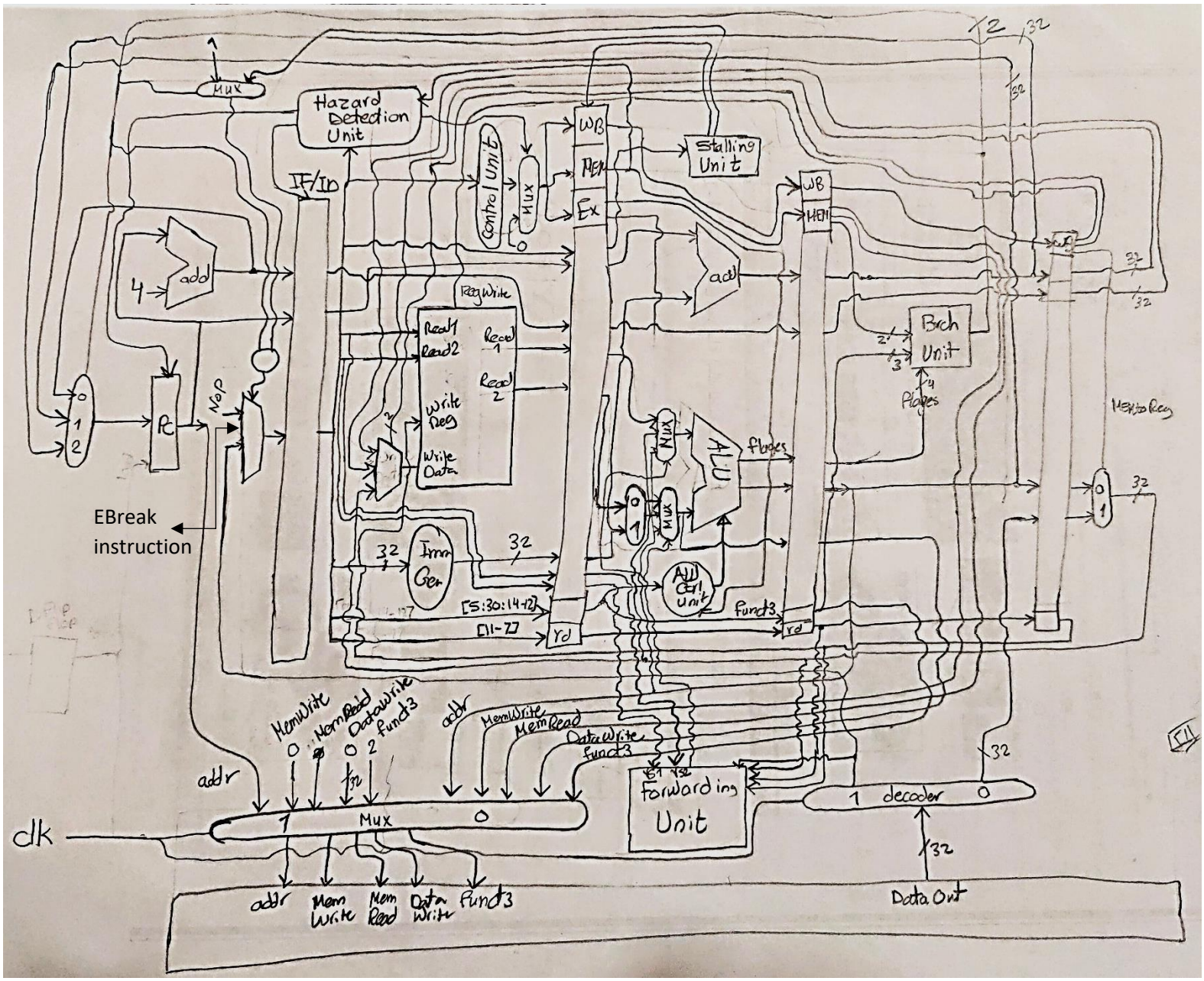
3- Data Memory changes
Data memory is designed to work for both purposes (instruction fetch, data memory usage). Hence, we divide the cycle into two so that in the first half cycle, the instruction gets fetch. In the next half cycle, the data memory is accessed the same as the register file (WB, ID). The main changes are removing the D flip flop and connecting the clk to the mux. Also, we set the

memory to read whenever MemRead is set to on. However, for writing, the memory writes at the negative edge.

4- Addition of Stalling Unit

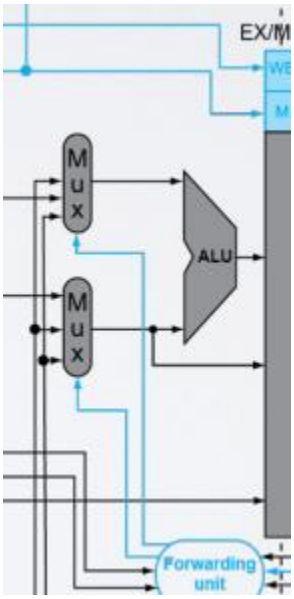
This unit is responsible for stalling the processor in case an Ebreak instruction enters the EX stage. However, this introduces another control hazard in case of previous branch instruction especially the case of taken branch. This unit along with the mux on the upper left makes sure the

New Data Path:



*The indicated circle component beside the pc indicates some logical operations done on the coming signals to act as a select signal for the instruction to be passed

Due to the limited space here we have on this paper, it's too difficult to spot the input wires of the ALU MUXs. However, it's the same wiring as the one mentioned in the slides for forwarding. It was difficult for us to make it clear here. Here is a close snapshot of this area



Testing Procedures:

We used several test programs to test our processor.

First Test:

We tested the new instructions like JAL, immediate instructions along with other branch instructions

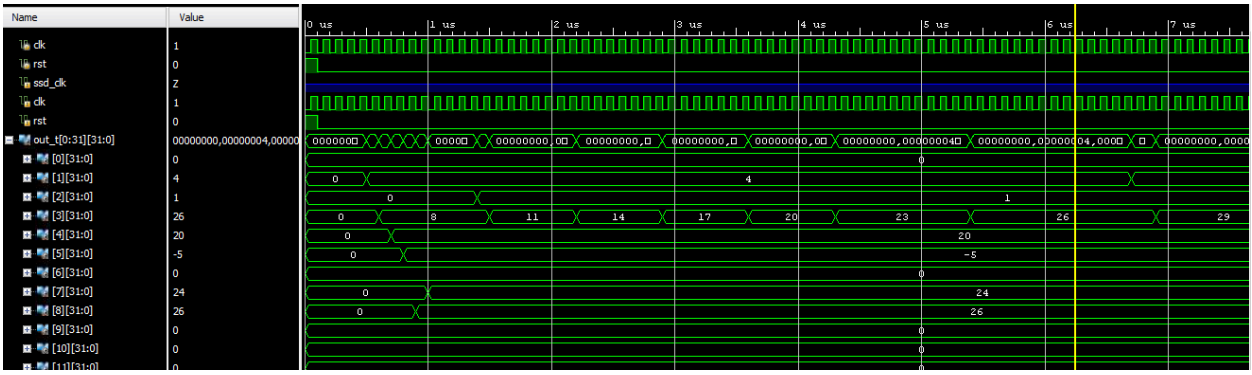
Assemply	Binary	Expectations
addi x1, x0, 4	00000000010000000000000010010011	X1 = 4
addi x3, x1, 4	00000000010000001000000110010011	X3 = 8
addi x4, x0, 20	00000001010000000000001000010011	X4 = 20
addi x5, x0, -5	11111111101100000000001010010011	X5 = -5
addi x8, x0, 26	00000001101000000000010000010011	X8 = 26
jal x7, function	00000000110000000000001111101111	X7 = 24
addi x1, x1, 1	00000000000100001000000010010011	----invalid instruction
addi x1, x1, 2	00000000001000001000000010010011	----invalid instruction
function: addi x2, x0, 1	00000000000100000000000100010011	X2 = 1
bigger_than: addi x3, x3, 3	00000000001100011000000110010011	X3 = 11
blt x4, x3, less_than	00000000001100100100011001100011	Skipped first time
bge x5, x3, bigger_than	11111110001100101101110011100011	Skipped first time
bgeu x5, x3, bigger_than	11111110001100101111101011100011	---all of the rest are loops which will
less_than: beq x3, x8, end	00000000100000011000010001100011	be explained below

bne x1, x2, function end: jalr x0, x7, 4	11111110001000001001010011100011 00000000010000111000000001100111	

The loop mentioned above first branches at bgeu instruction until the value of x3 reaches a number bigger than 20 so that it can branch at the blt instruction. Once it branches, it checks the x3 and x8. In the first time, they are not equal. Therefore, the program jumps to function again which is indicated below on the screen shots by (BNE → function(address of 32)). Then, x3 is incremented again until it reaches the number 26. Once it reaches this number, it can jump to the end label (jalr instruction) which jumps to x7 +4 (addi x1, x1, 2). This is indicated below in the screen shots by the yellow comment (JALR) and change in the x1 value from 4 to 6 at the same period the x3 is 26.

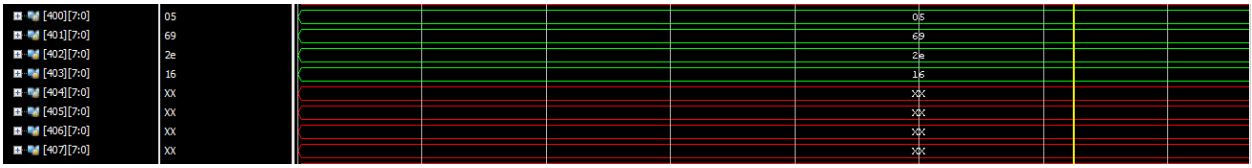
Simulation results:

registers

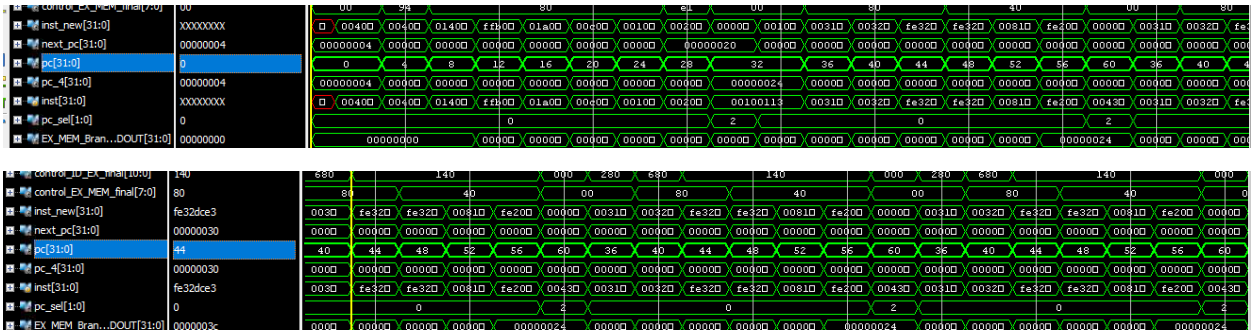


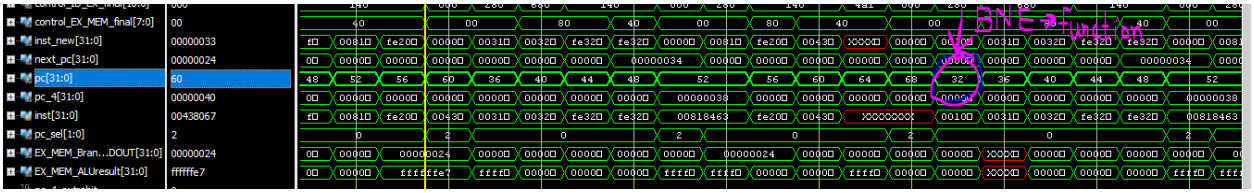
The register values were as expected. It first intili

Memory

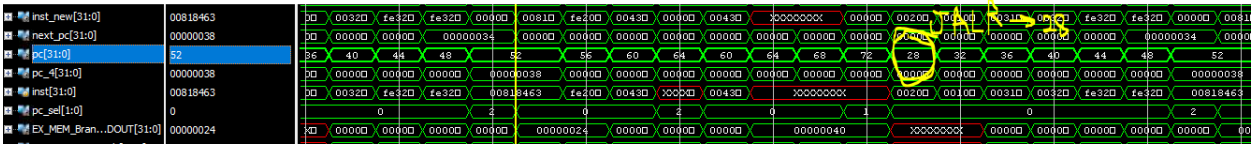


Others





This is the jump to the function label by the BNE instruction



This is the jump by the JALR instruction (x7 +4 = 24 +4 = 28)

Second test:

We tested the rest of the instructions (shift instructions, load upper immediate)

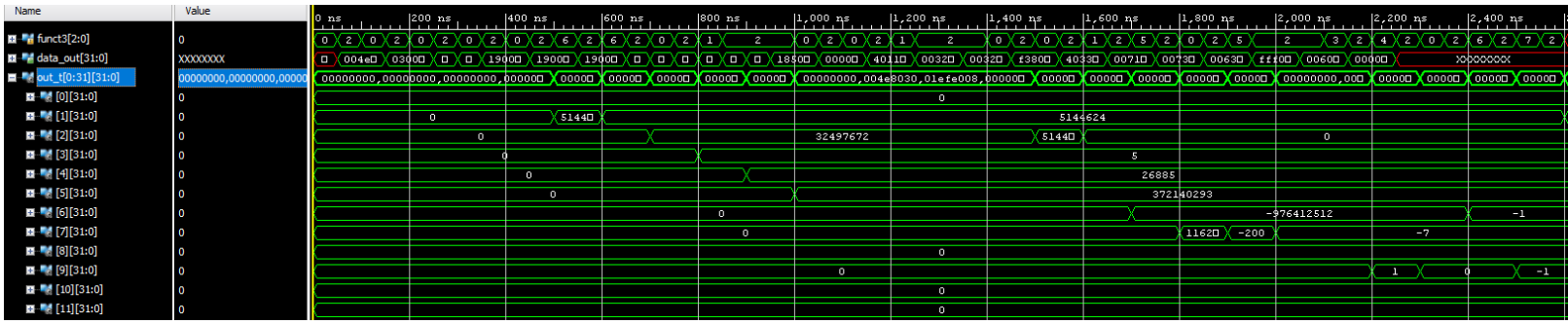
Assemply	Hexa	Expected Results
lui x1, 1256	004e80b7	
ori x1, x1, 48	0300e093	#(x1 = 5144624)
auipc x2, 7934	01efe117	#(x2 = 32497664+8 = 32497672)
lb x3, 400(x0)	19000183	#(x3 = 5)
lh x4, 400(x0)	19001203	#(x4 = 01101001_00000101 = 26885)
lw x5, 400(x0)	19002283	#(x5 = 00010110_00101110_01101001_00000101 = 372140293)
sb x5, 404(x0)	18500a23	#(mem[4] = 00000101 = 5)
sh x5, 408(x0)	18501c23	#(mem[8],[9] = 01101001_00000101 = 105, 5 == 26885)
sw x5, 412(x0)	18502e23	#(mem[12],[13],[14],[15] = 00010110_00101110_01101001_00000101 = 22,46,105,5==372140293)
add x2, x1, x0	00008133	#(x2 = x1 = 5144624)
sub x2, x2, x1	40110133	#(x2 = 0)
sll x6, x5, x3	00329333	#(x6 = 3318554784)
srl x7, x5, x3	0032d3b3	#(x7 = 11629384)
addi x7, x0, -200	f3800393	#(x7 = -200)
sra x7, x7, x3	4033d3b3	#(x7 = -7)
slt x8, x2, x7	00712433	#(x8 = 0)
sltu x9, x6, x7	007334b3	#(x9 = 1)
xor x9, x6, x6	006344b3	#(x9 = 0)
addi x6, x0, -1	fff00313	#(x6 = -1)
or x9, x1, x6	0060e4b3	#(x9 = -1)
and x1 ,x1 , x0	0000f0b3	#(x1 = 0)

#mem[400] = 8'd5	

#mem[401] = 8'd105 #mem[402] = 8'd46 #mem[403] = 8'd22	

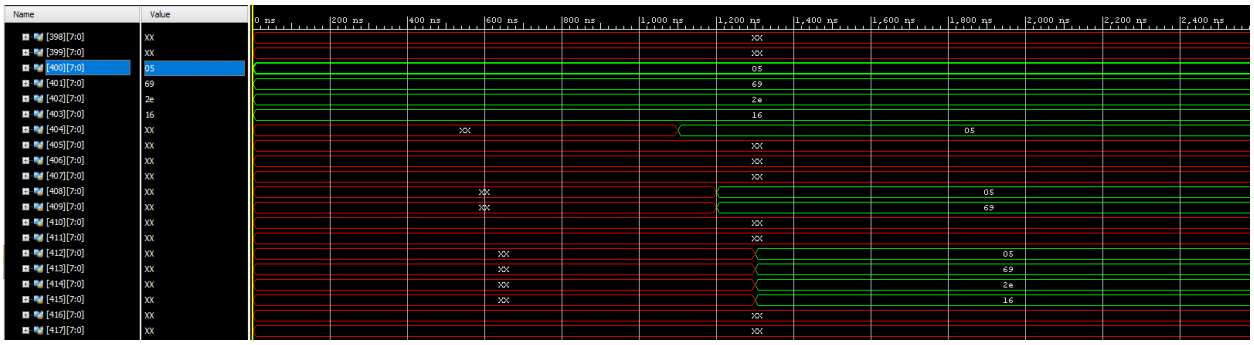
Simulation results:

Registers



The register values were as expected and as mentioned above in the table. Therefore, execution of instructions, fetching, decoding and writing to registers is working correctly.

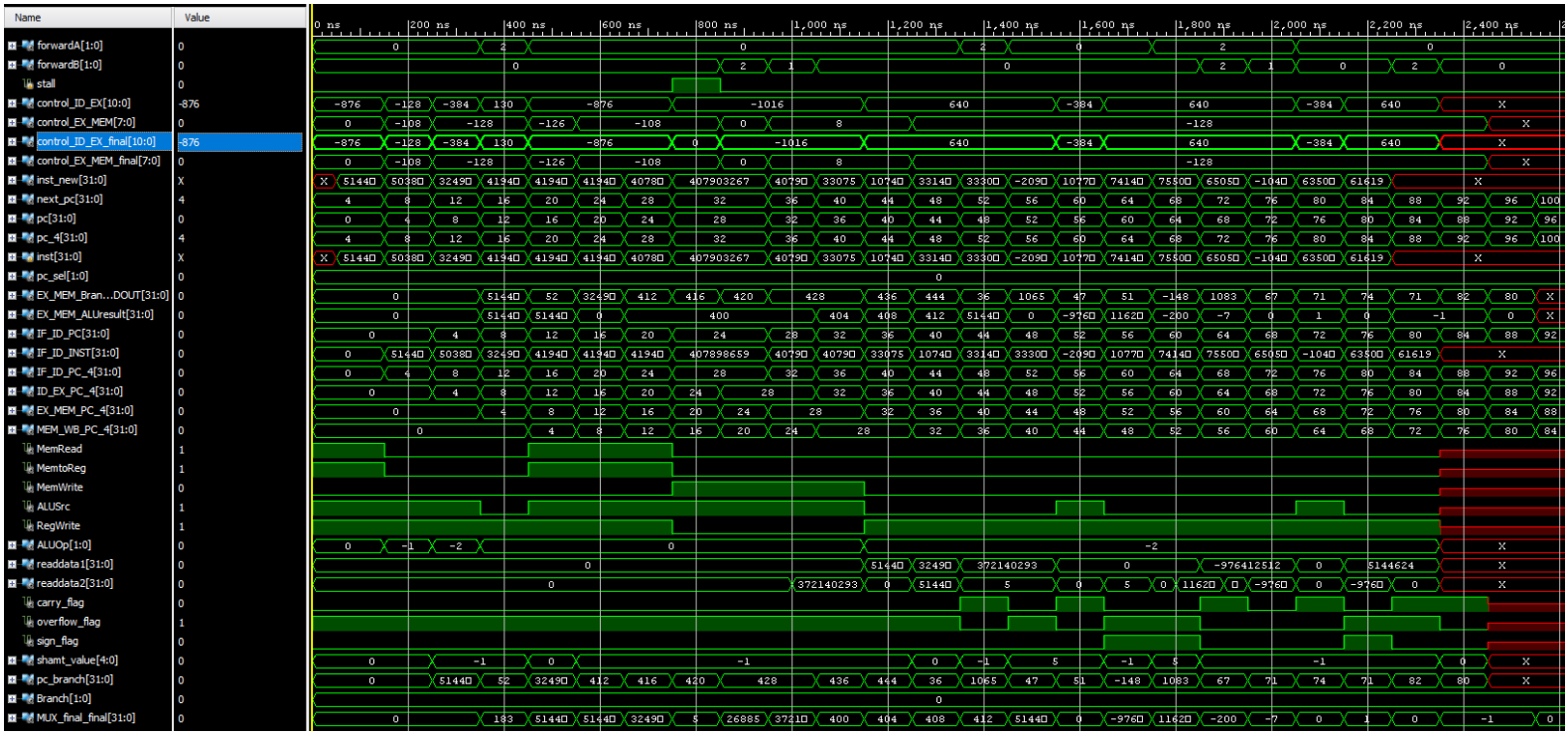
Memory



Here are the 4 things written to the memory (sb, sh, sw). These values were expected, the same values as indicated in the table. As there is a load use dependency between sb and lw (the previous instruction). This indicates that the instruction thrashing is working correctly along with the forwarding unit.

Therefore, the memory stage is working correctly

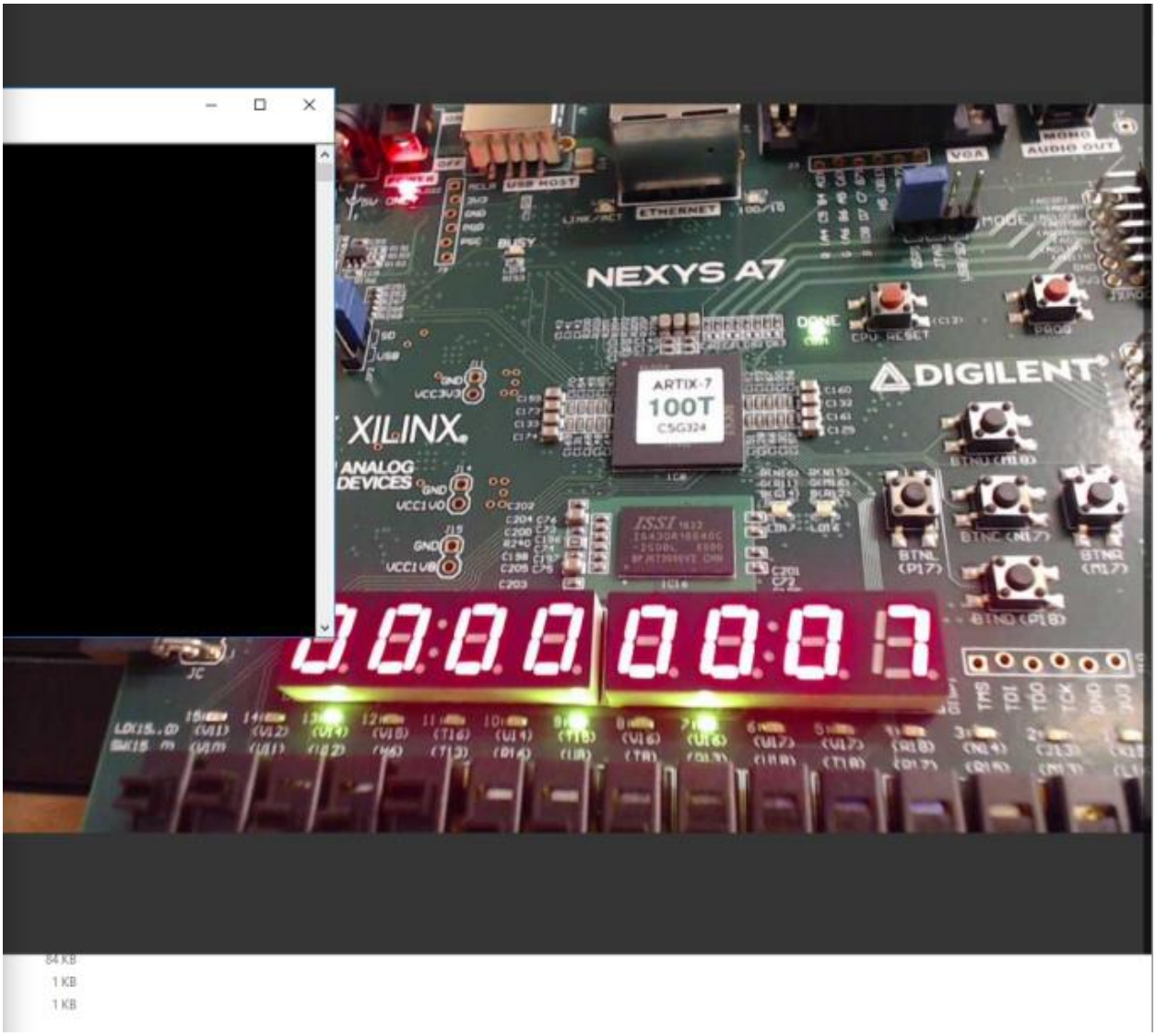
Others



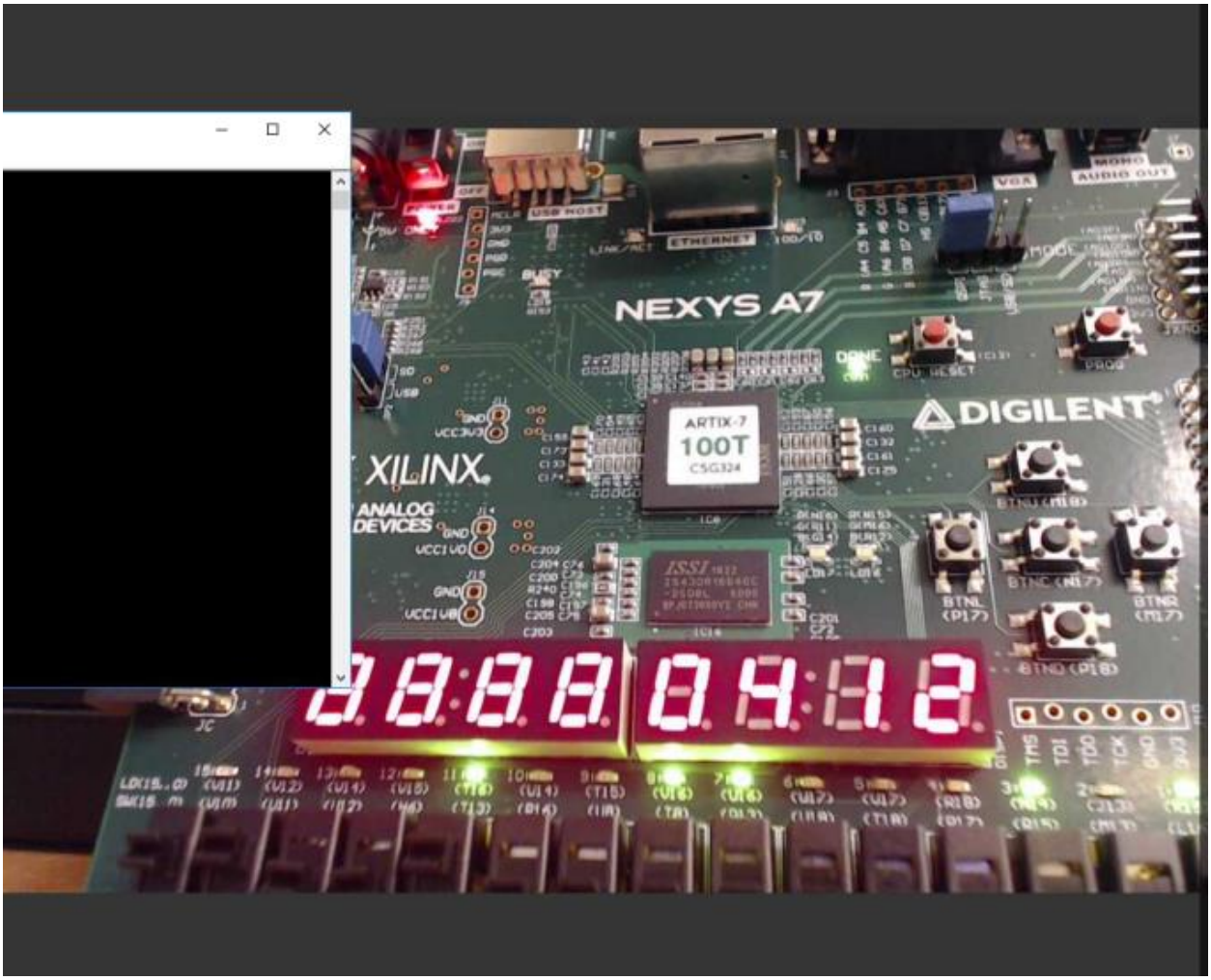
Here we can see the stall signal (third one) and the pc (tenth one). We noticed that program stalled at instruction 28 which is the 8th instruction which is the sh instruction. This was the stall needed for the load use data dependency between (lw and sb) instructions as they are using the same register x5. This indicates that load use hazards are handled correctly.

FPGA Testing:

Stalling on load use dependency



This is the value of the pc where the pc gets stalled. The pc was stalled for two cycles. It gave 7 on two cycles. It did so because there we have a load use data dependency.



This is the address of the memory where we are storing the word in instruction (sw x5, 412(x0))

Third test:

We tested the ebreak, ecall, fence instructions

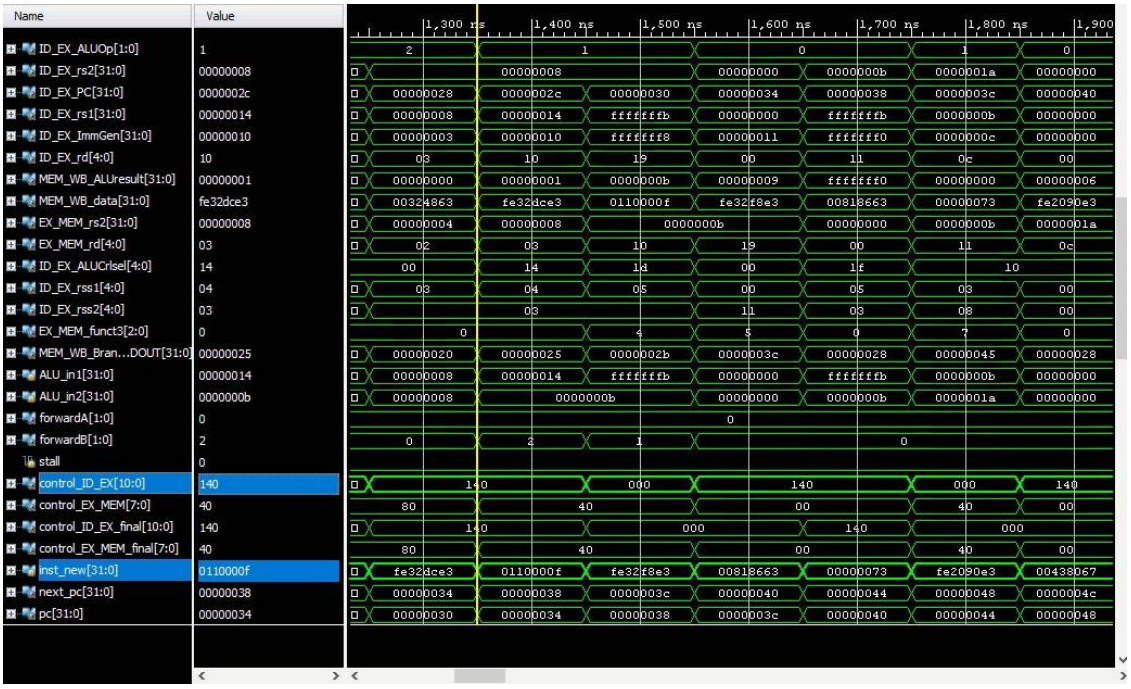
Assemply	Hexa	Expected Results

addi x1, x0, 4
addi x3, x1, 4
addi x4, x0, 20
addi x5, x0, -5
addi x8, x0, 26
jal x7, function #checking jal
addi x1, x1, 1
addi x1, x1, 2 # checking offset of jalr
# end of program 	
ebreak
function:
addi x2, x0, 1
	93 00 40 00
	93 81 40 00
	13 02 40 01
	93 02 B0 FF
	13 04 A0 01
	EF 03 00 01
	93 80 10 00
	93 80 20 00
	73 00 10 00
		The same as before except the nope instruction for the ecall and fence. Also the instruction will stop at the ebreak

bigger_than: addi x3, x3, 3 blt x4, x3, less_than #checking blt works correctly bge x5, x3, bigger_than # checking BGE vs BGEU fence 1, 1 bgeu x5, x3, bigger_than less_than: beq x3, x8, end #check beq ecall bne x1, x2, function #checking bne and backward jumps end: jalr x0, x7, 4 #cheking jalr	13 01 10 00 93 81 31 00 63 48 32 00 E3 DC 32 FE 0F 00 10 01 E3 F8 32 FE 63 86 81 00 73 00 00 00 E3 90 20 FE 67 80 43 00	

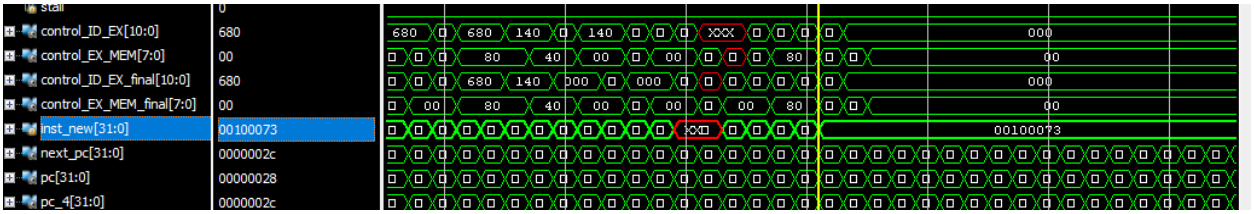
Simulation results:

Registers



Here, after one cycle of the ecall instruction (0110000f), all control signals (control ID_EX) had been set to zero

Instruction



It stopped at the ebreak instruction (00100073) as expected