RISC-V Pipelined Processor
Milestone 3
Report

Omar Miniesy 900202087
Ziad Miniesy 900202283

Overview

In this milestone the main objective was to implement the 5 stage pipelined datapath for the RISC-V 40-instruction set. To achieve this, we had first to introduce the 4 pipeline registers, then we implemented a forwarding unit that forwards the operands through each stage, and finally we had to implement the flushing system. This project also used a single memory for both instructions and data, which prevented the need to implement a hazard detection unit.

Pipeline Registers

IF/ID: This register is directly after the fetching stage. As mentioned, a single memory was implemented, so there is no instruction memory and data memory. The fetched instruction is stored in this register. Since fetching takes one clock cycle, this allows us to fetch the next instruction and decode the current instruction at the same time, since the IF/ID register sends the opcode as well the operands to the decoding stage and stores the control signals decoded in the ID stage to the ID/EX register.

ID/EX: This register, as previously mentioned, takes the input from the ID stage. The PC, control signals, register file data, immediate value and the funct3 and funct7 of the instructions. Similar to the previous stage, this allows decoding of the next instruction and passing those values to the execution stage.

EX/MEM: This is the third pipeline register which is responsible for storing the outputs of the ALU as well as memory addresses and control signals needed for memory access. This data is passed on to the MEM stage for memory access, which can include loading data from memory, storing data to memory, or modifying data in memory. Finally, this allows the processor to start executing the next instruction without having a problem in changing the values of the ALU outputs since they are already stored in the EX/MEM register and can be used in the MEM stage.

MEM/WB: This is the final pipeline register which is responsible for storing the outputs of the memory stage which are to be written back to the register file. It also keeps track of the control signals to be able to differentiate between different instruction types.

Forwarding Unit

Forwarding was implemented to avoid data hazards. Data hazards occur when an instruction uses a value in a register that has not been yet written back to the register file by the previous instruction, ending up producing a wrong value. However, since the outputs of the ALU are stored in the EX/MEM register, they can be used directly without waiting for the data to be written back.

The same case happens when trying to access a memory location that is still not updated yet. Data to be written to the memory is stored in the MEM/WB reister, so the forwarding unit forwards this data from the register to the ALU. To achieve this, we created 2 wires forwardA and forwardB, one for each ALU input. Those wires act as flags that are used as select lines for 2 multiplexers placed before the ALU. Those select lines are set to a certain value if the destination register of an instruction is used as a source register for the following instruction.

Flushing

Flushing is used in the case of branching or jumping. Since we assume The need for branching and jumping is not determined before the execution stage, at this moment already an instruction has entered our pipeline because we follow a branch not taken prediction, so this instruction needs to be flushed. Flushing is implemented by setting the control signals in the ID/EX register to zeroes. A multiplexer after the PC was also added to choose between the instruction or a NOP. This allowed us to use branching and jumping instructions without having to add NOPs after them.

Implementing Single Memory

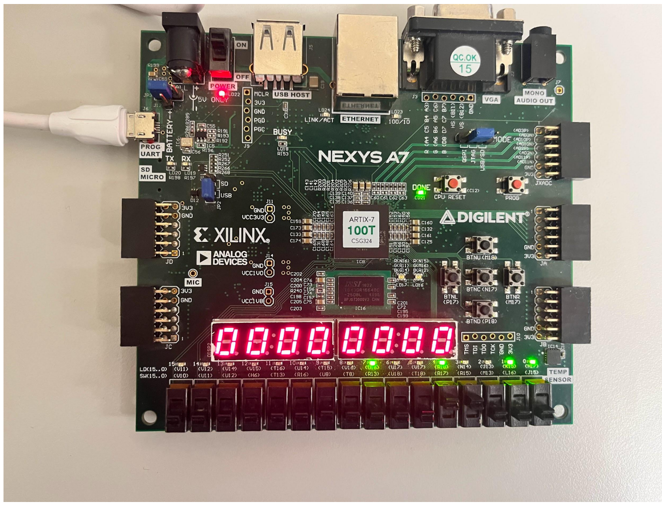
To implement 2 ported single memory for our processor, we had to modify the clock passed to our pipeline registers to alternate between each stage, so IF/ID and EX/MEM took the negation of the clock, while the ID/EX and MEM/WB took the clock. This was mainly to prevent reading and writing simultaneously. We then added a decoder to either assign the value read from the memory to a data memory wire or assign it to an instruction memory wire based on the clock. Finally, we added a multiplexer before the memory to either read data or read instructions. Loading instructions was implemented the same way load word instruction was implemented, since an instruction is actually a word.

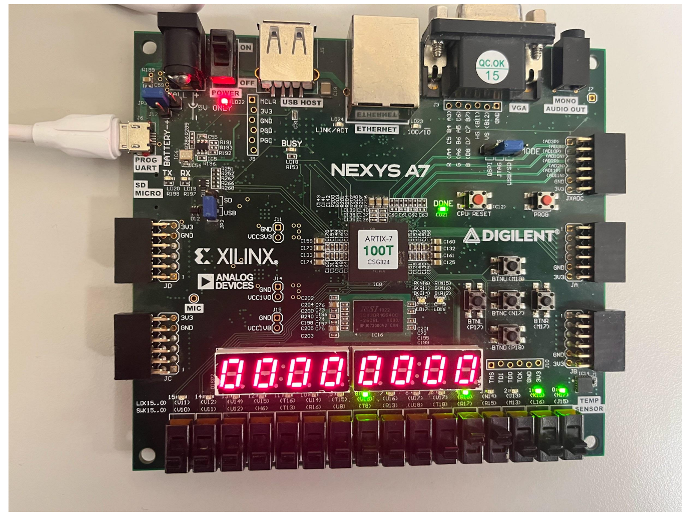
Bonus Features

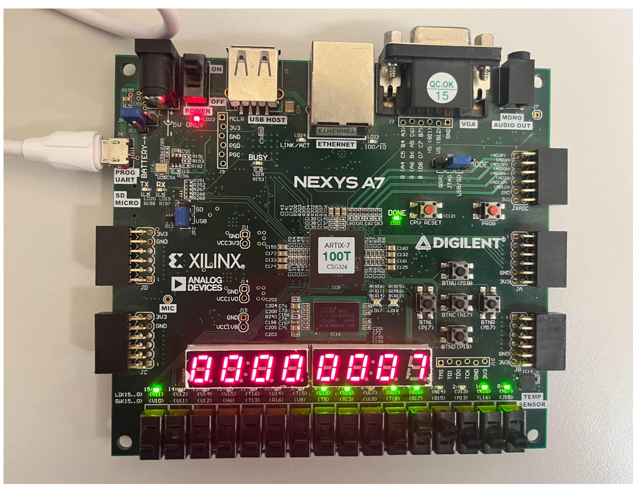
1) The first bonus feature implemented was supporting multiplication. To achieve this, we had to modify the ALU control and the ALU. We checked the 25th bit of the instruction to differentiate between multiplication and other rtype instructions. The ALU control then assigns a specific value for the alufn which is then used in the ALU to compute the suitable result.
2) The second bonus feature implemented was showing the results on the FPGA. Pictures of the results are shown below. The test program was as follows:

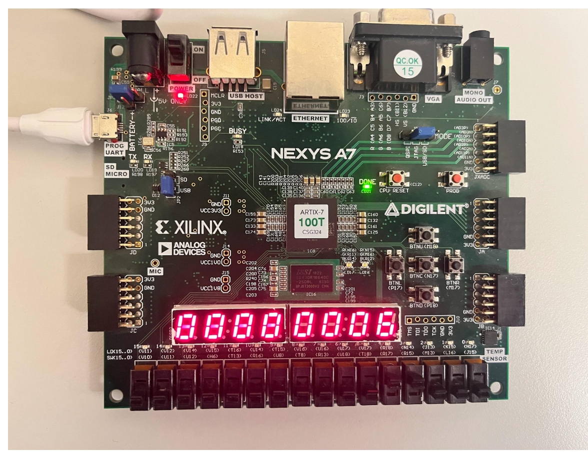
```
addi x1,x0,7
addi x2,x0,6
mul x3,x1,x2
```

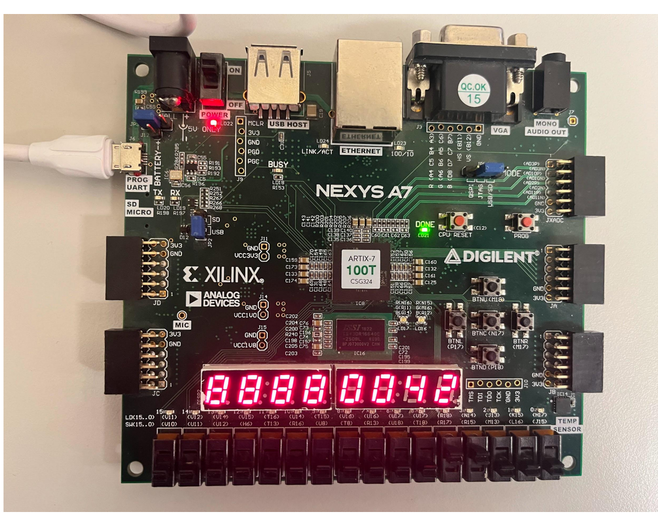
As shown below, the results should be 42. The constraints file for the FPGA is attached in the folder.

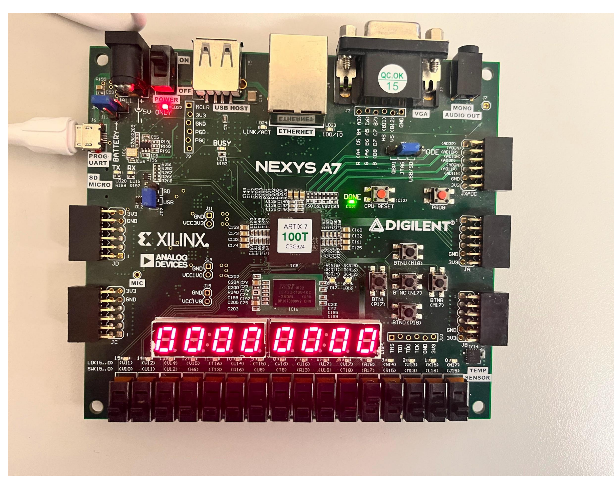












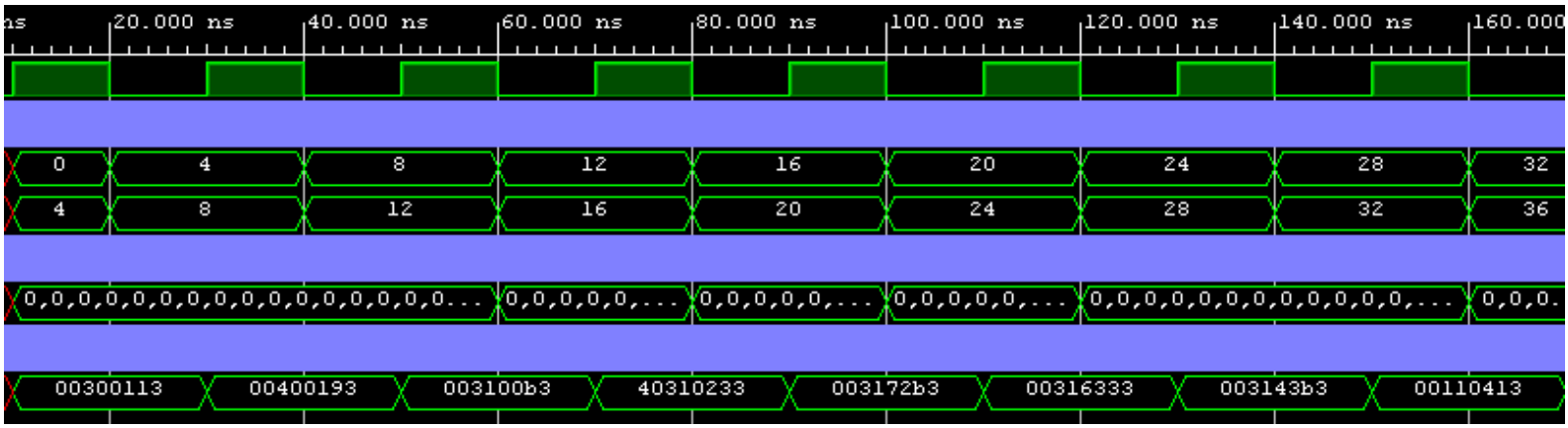
Test
Attached below is the waveform for this test:

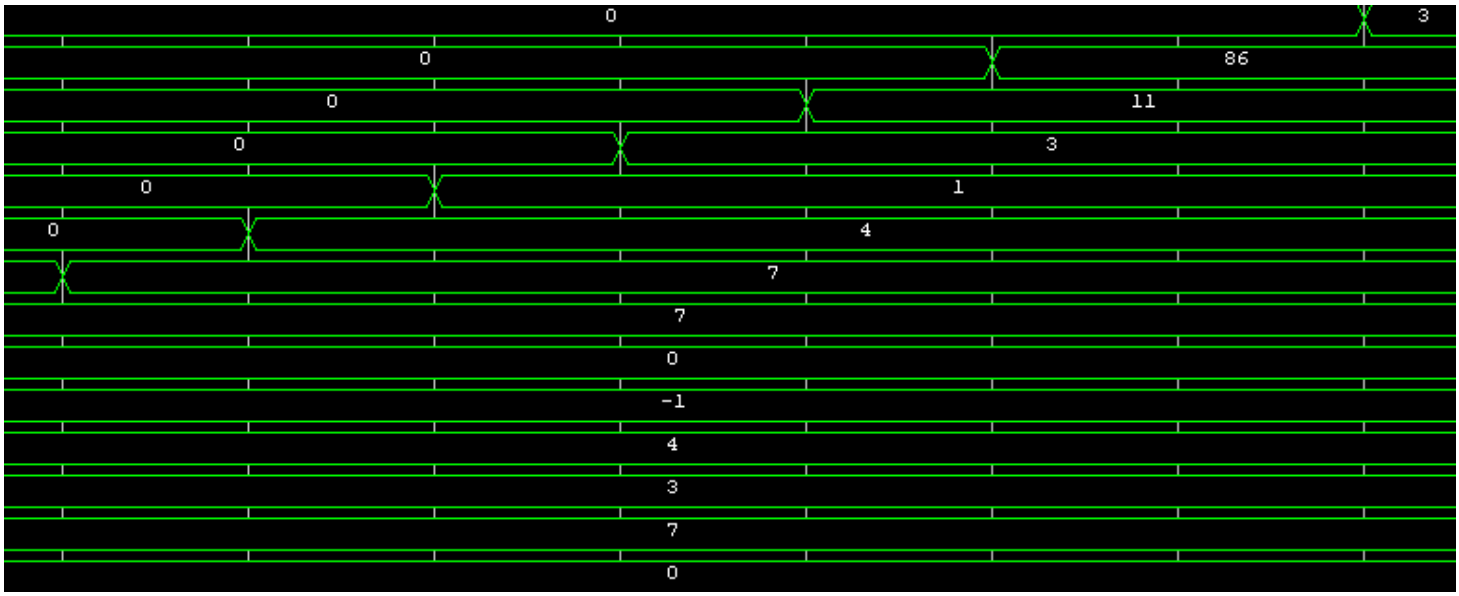
```
addi x2,x0,3
addi x3,x0,4


add x1, x2, x3
sub x4, x2, x3
and x5, x2, x3
or  x6, x2, x3
xor x7, x2, x3

addi x8, x2, 1
slti x9, x2, 5
andi x10, x2, 15
ori x11, x2, 10
xori x12, x2, 85

sw x2, 0(x3)
lw x13, 0(x3)

beq x2, x3, end
bne x2, x3, start
blt x2, x3, start
bge x2, x3, end

lui x14, 12
auipc x15, 24

jal x0, end

lui x16, 48

addi x16, x16, 20
jalr x17, x16, 0
end:addi x2, x2, 1
addi x2, x2, -3
start: addi x2, x2, 2
```

Expected Results

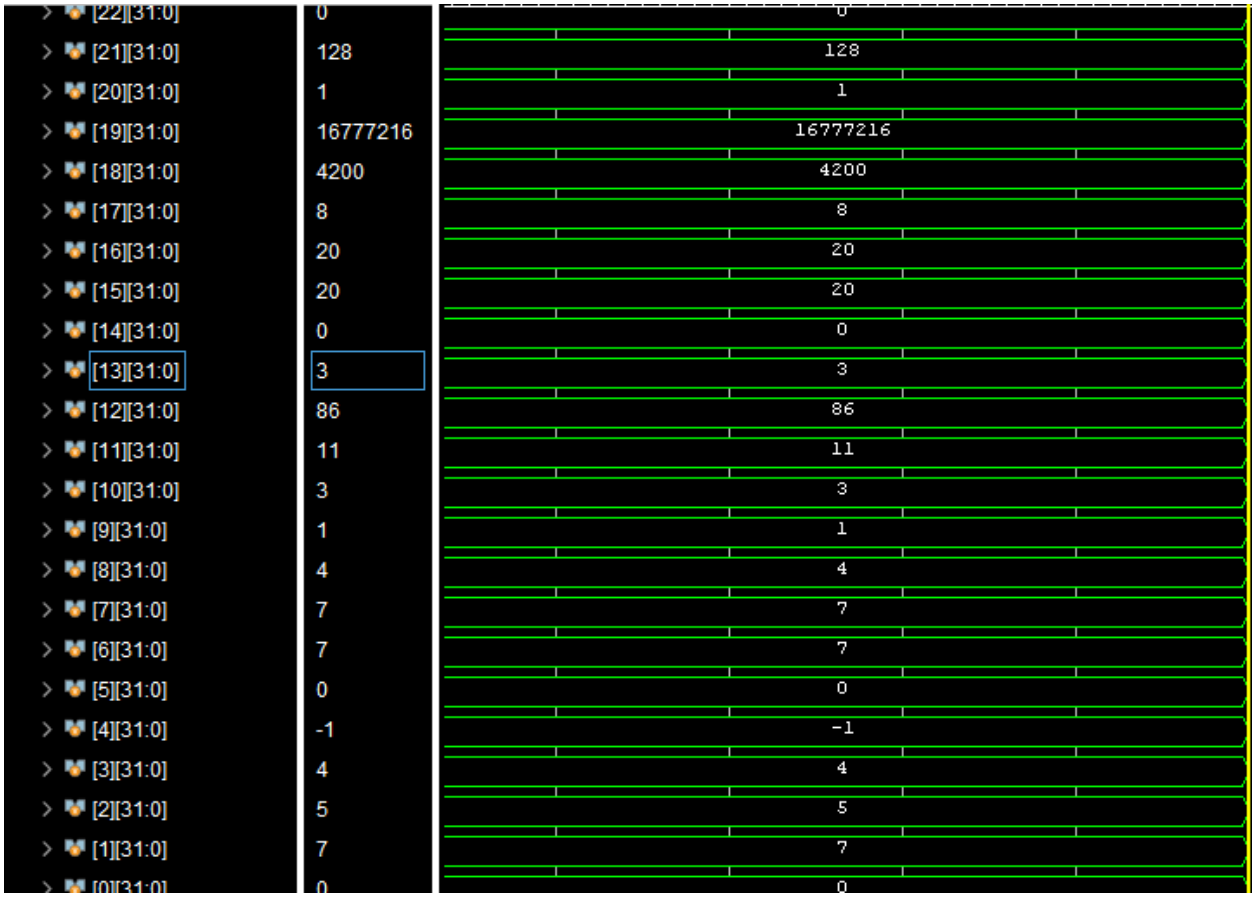
zero	0
ra (x1)	7
sp (x2)	5
gp (x3)	4
tp (x4)	-1
t0 (x5)	0
t1 (x6)	7
t2 (x7)	7
s0 (x8)	4
s1 (x9)	1
a0 (x10)	3
a1 (x11)	11
a2 (x12)	86
a3 (x13)	3
a4 (x14)	0
a5 (x15)	20
a6 (x16)	20
a7 (x17)	8
s2 (x18)	4200
s3 (x19)	16777216
s4 (x20)	1
s5 (x21)	128
s6 (x22)	0

Waveform


20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000
0 4 8 12 16 20 24 28 32
4 8 12 16 20 24 28 32 36
00300113 00400193 003100b3 40310233 003172b3 00316333 003143b3 00110413


0 3
0 86
0 11
0 3
0 1
0 4
7
7
0
-1
4
3
7
0

Contents of the Register File



Schematic

A clearer version of the schematic can be found in the submitted folder, inside the report folder.