CSE 306 : Offline 4

8-bit MIPS Pipelined Execution

July 4, 2021

Lab Section : B2
Group 03

1705093
1705098
1705103
1705110
1705119

1 Introduction

In this assignment we implemented an 8-bit processor that supports pipelined datapath for a subset of MIPS instruction set.

What is Pipelining: In the single cycle implementation, only 1 instruction can be processed during a clock cycle. This has two problems:

- The clock cycle length becomes equal to the time required to process the most complex instruction.
- When one part of the circuitry is busy processing the instruction, other parts of the circuitry remain idle.

To solve these problems we use Pipelining where multiple instructions are overlapped in execution. This is done by dividing each instruction into fives stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execution and Address Calculation (EX)
4. Data Memory Access (MEM)
5. Write Back (WB)

Now, at any given time, the circuitry can process up to 5 different instructions since all the instructions can be in different stages and thus can be processed by different parts of the circuitry.

The length of the clock cycle is now equal to the maximum time needed to execute any single stage. Therefore, the length of each clock cycle decreases while the number of clock cycles needed for execution of each instruction increases to 5. The overall affect of this can slightly increase the latency, but improves the throughput of instruction execution significantly.

2 Block Diagram of Pipelined Datapath

We have designed the circuitry to handle r-type instructions and handle the following hazards:

- EX Hazard
- MEM Hazard
- Double Data Hazard

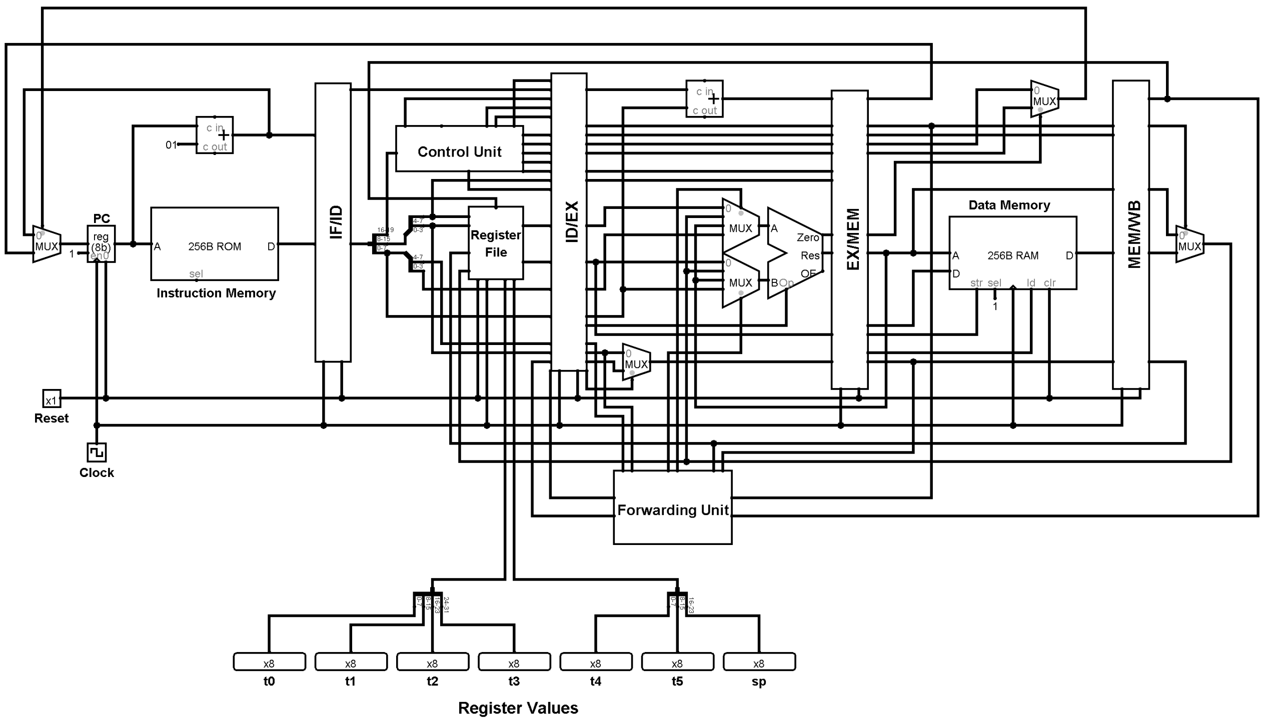


Figure 1: Pipelined MIPS Processor

However, since we have built on top of the design for the previous assignment, we have not removed some components that are not required for r-type instruction execution. In fact, our design handles i-type instructions as well.

3 Block Diagrams and Size of Pipeline Registers

3.1 IF/ID

The diagram below shows the components of the IF/ID registers (Instruction Fetch/Instruction Decode). For each component the size is given in its corresponding box. **The total size of the IF/ID registers is 28 bit.**

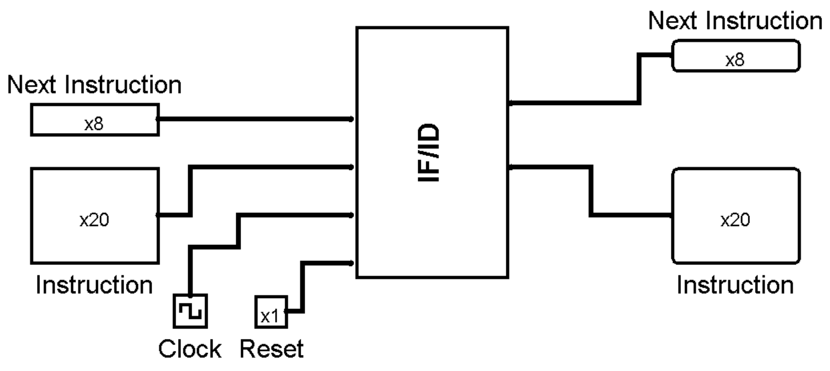


Figure 2: IF/ID Registers

3.2 ID/EX

The diagram below shows the components of the ID/EX registers (Instruction Decode/Execution). For each component the size is given in its corresponding box. **The total size of the ID/EX registers is 60 bit.**

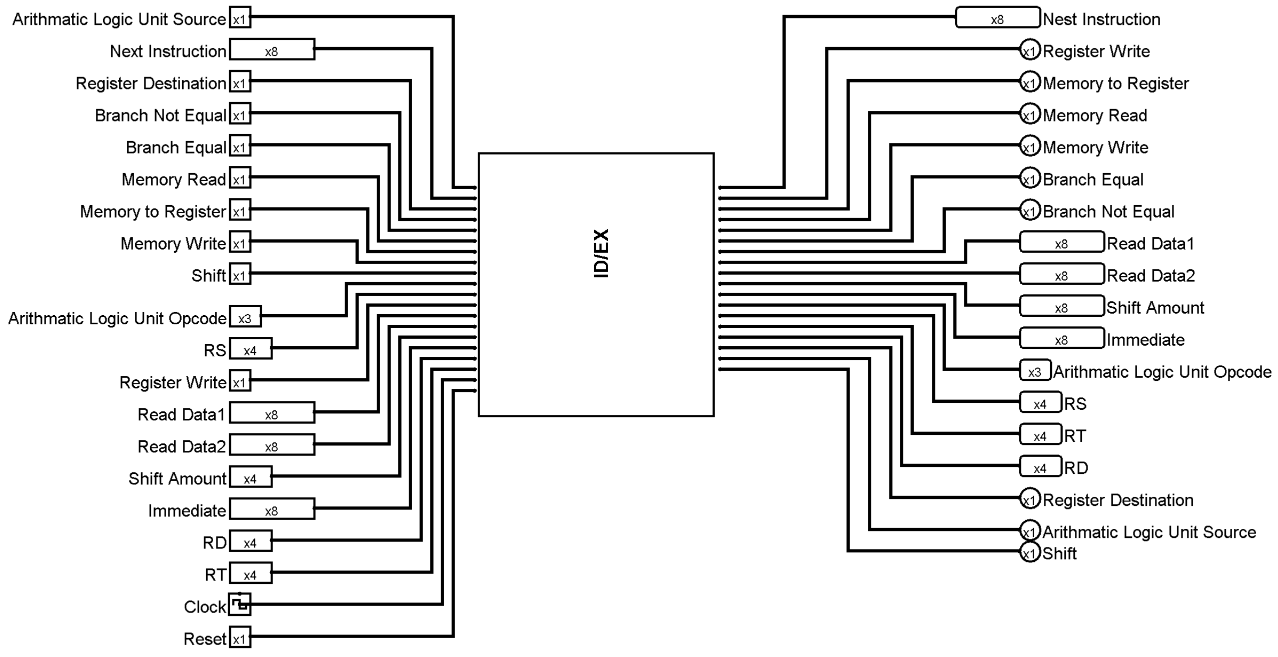


Figure 3: ID/EX Registers

3.3 EX/MEM

The diagram below shows the components of the EX/MEM registers (Execution/Memory). For each component the size is given in its corresponding box. **The total size of the EX/MEM registers is 35 bit.**

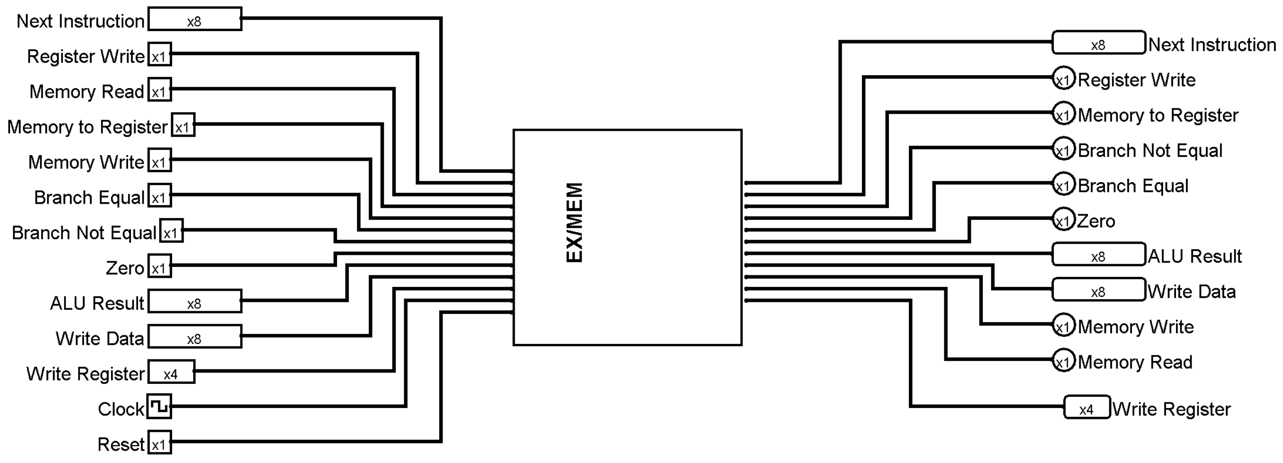


Figure 4: EX/MEM Registers

3.4 MEM/WB

The diagram below shows the components of the MEM/WB registers (Memory/Write Back). For each component the size is given in its corresponding box. **The total size of the MEM/WB registers is 22 bit.**

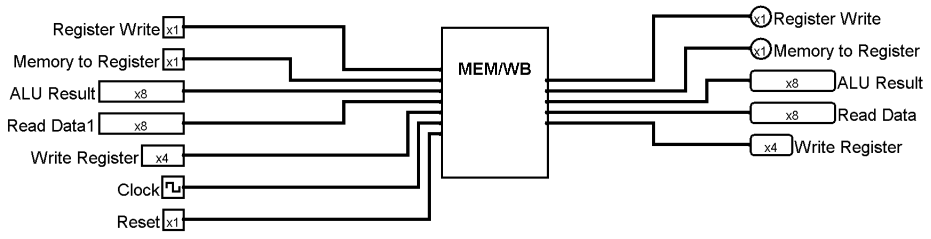


Figure 5: MEM/WB Registers

4 Mechanism and Block Diagram of Forwarding Unit

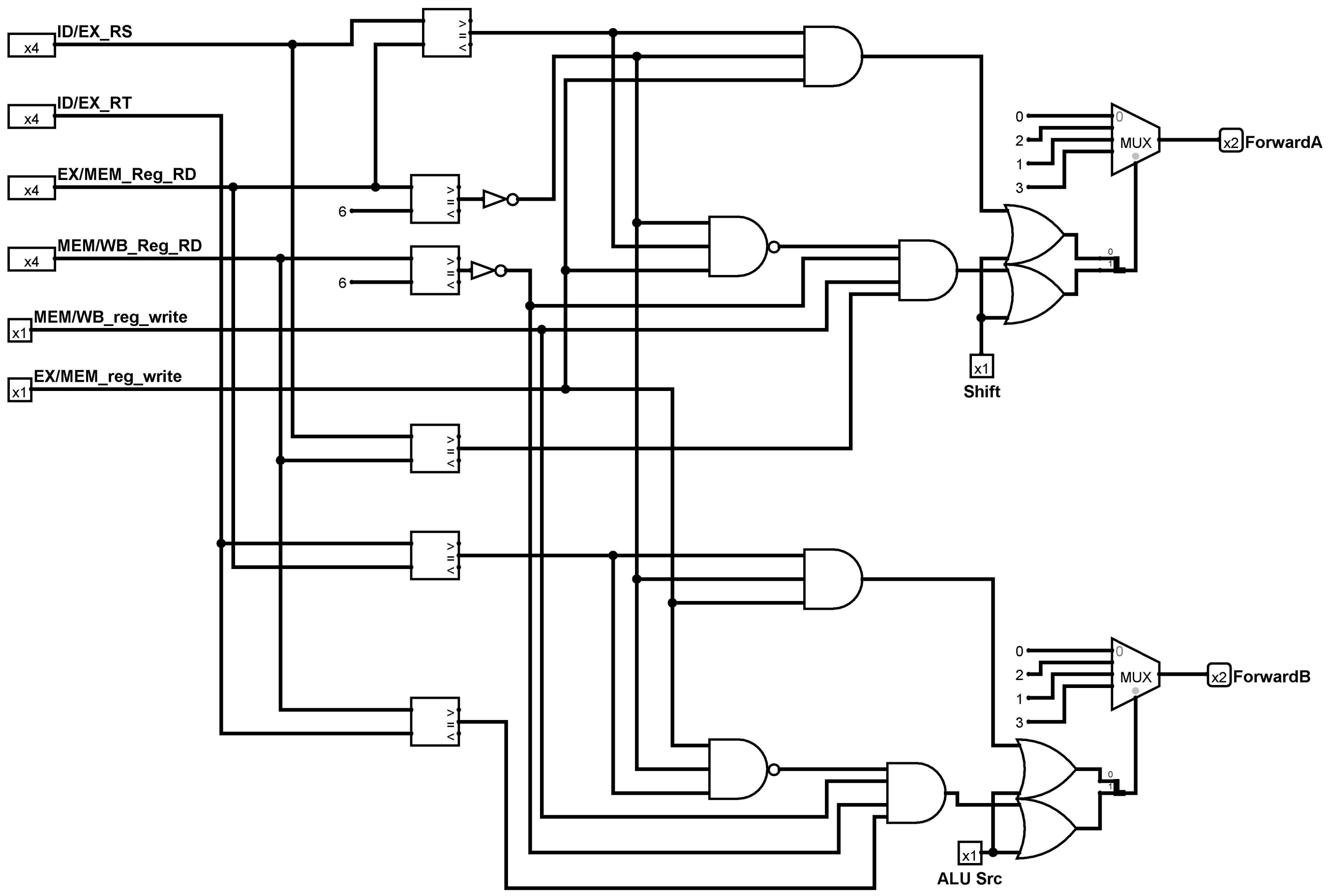


Figure 6: Forwarding Unit Block Diagram

The forwarding unit plays a major role in the pipeline process. Suppose we have instructions like: *add* $\$1, \$1, \$2$ and then *sub* $\$3, \$1, \$0$. Since, multiple instructions are running simultaneously, we need to be able to grab the updated value of $1 register as soon as add execution in the ALU occurs.

An instruction of such a type depends on completion of data access by a previous instruction. The forwarding unit controls what data from one stage of instruction execution will be a feedback or input to the next dependent instructions.

Detecting What to Forward

Mux Control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from register file
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from the data memory or an earlier ALU result
ForwardA = 11	SHIFT	The first ALU operand is from shift
ForwardB = 00	ID/EX	The second ALU operand comes from register file
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from the data memory or an earlier ALU result
ForwardB = 11	Immediate	The second ALU operand is an Immediate

We implemented the pipline stages so that the following forwarding output bits select what source should the ALU operands come from. Each operand may come from any of the following:

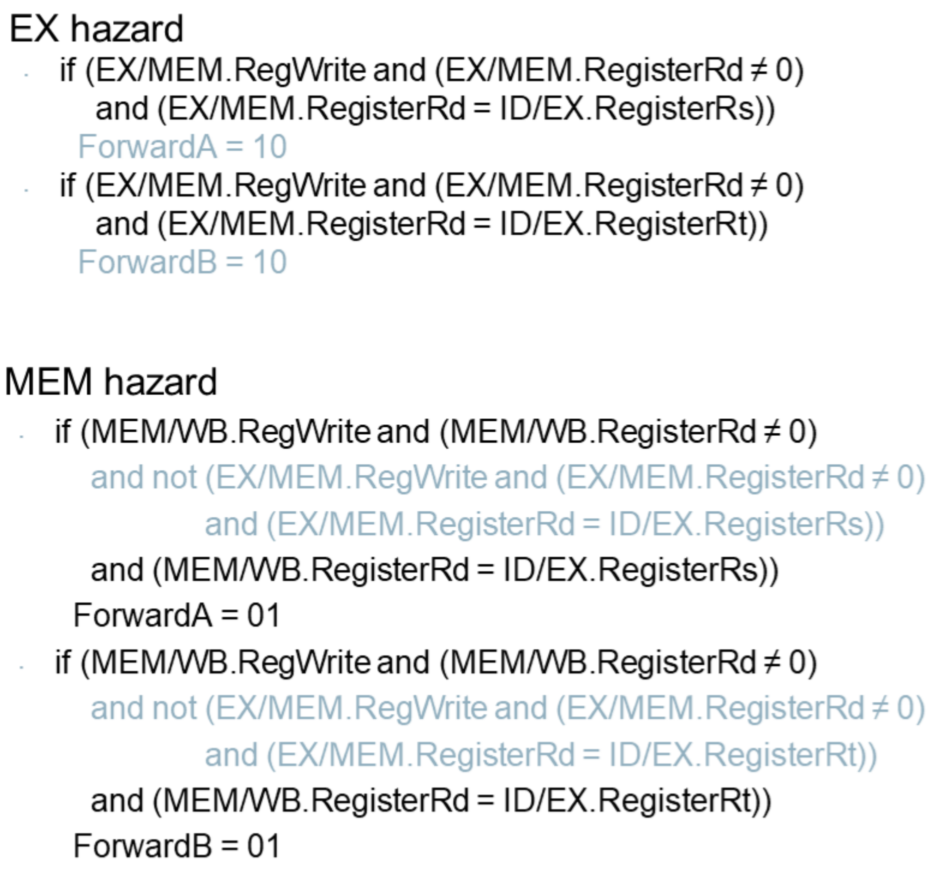
- register file(ID/EX)
- from the previous ALU result(EX/MEM)
- from data memory bank or even earlier ALU result
- from shift (for 1st operand) or immediate (for 2nd operand)

4:1 MUXs outside the forwarding unit are fed with these selection bits to select any of the above operands as shown in the table above.

Detecting When to Forward

Consider the previous example again: *add* $\$1, \$0, \$2$ and then *sub* $\$3, \$1, \$0$. Here when add instruction is performed and the value is of \$1 is saved in EX/MEM, and then forwarded to the sub instruction.

Note that, for this case, if the destination register of the first instruction(EX/MEM.RegisterRd) equals the source register in the next instruction(ID/EX.RegisterRs), we need to forward the value in EX/MEM.RegisterRd to the ALU.



EX hazard

```
· if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
     and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
   ForwardA = 10
· if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
     and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
   ForwardB = 10
```

MEM hazard

```
· if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
     and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
                and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
     and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
   ForwardA = 01
· if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
     and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
                and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
     and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
   ForwardB = 01
```

Figure 7: The conditions for forwarding

The conditions above are all checked in the forwarding unit. Registers are matched using 4-bit comparators; the equal output is used. Non-zero values are checked using same types of comparators with zero(0x4) as the other input and then inverting the output. Together, non-zero checking and equal checking are passed through AND/NAND gates and fed to a 4:1 MUX for each forward bit generation. Since, we are implementing for R-type instructions, we added support for shift and immediate using OR-gates. These use the forward bits-11 which were previously unused.

5 Simulation Platform

Logisim-2.7.10

6 Discussion

- Since at the start of execution the value of all the registers is 0, we would have to manually set the values of the registers to test the r-type instructions. Manually setting the register values seemed inconvenient, so we kept support for i-type instructions as well.
- We designing we tried to keep our design simple and efficient. However, after completing the design we did not perform any separate check for opportunities to minimize IC count since IC minimization was not a requirement for this assignment.