13 Pipelined CPU Design with Data Forwarding

13.1 Objectives

After completing this lab, you will:

- Learn how to design a pipelined CPU
- Learn the different types of pipeline hazards
- Implement Forwarding to handle data hazards
- Verify the correct operation of your pipelined CPU design

13.2 Pipeline Data Path

The single-cycle data path design can be pipelined into a 5-stage pipeline by introducing registers at the end of each stage as shown in Figure 13.1.

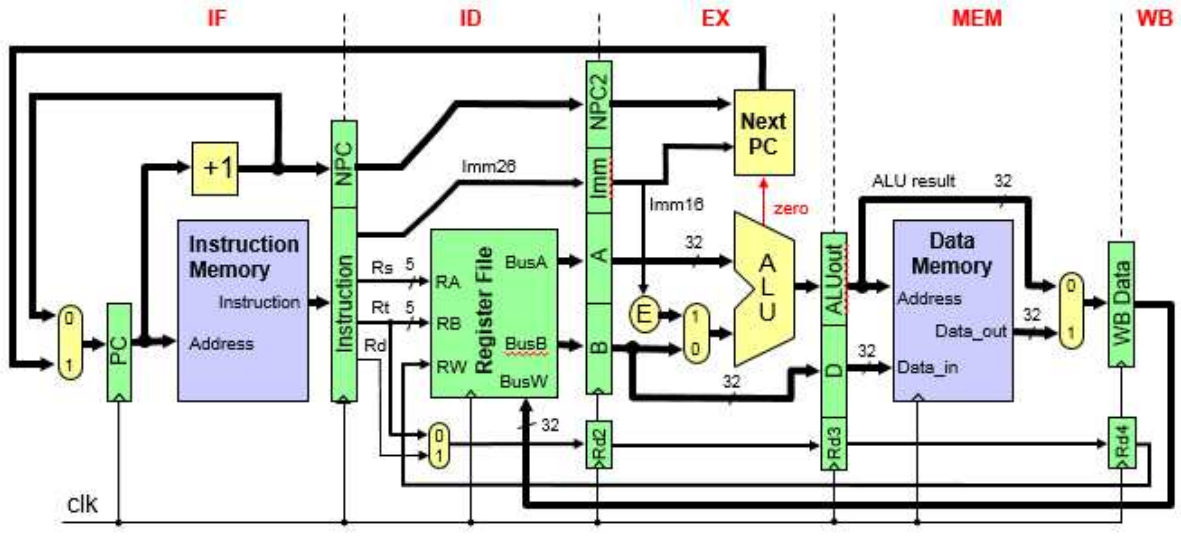

Figure 13.1: Pipelined Data Path.

It should be observed that the destination register number is also pipelined by saving it across stages as the writing of the content of the register is done at stage 5. In addition, the incremented value of PC is also pipelined across stage 2 and 3 as it is need by the Next PC block.

13.3 Pipelined Control

The control signals are generated by the control unit in the second stage (i.e. ID state). In order to pipeline the control unit, we need to save all the control signals needed by the later stages in registers. For example, the control signals ExtOp, ALUSrc, ALUCtrl, J, Beq, Bne, MemRead, MemWrite, Memtoreg and RegWrite need to be saved in the register separating stage 2 and stage 3. However, only the control signals MemRead, MemWrite, Memtoreg and RegWrite are saved in the register separating stages 3 and 4 as the remaining control signals are used in stage 3 and are not needed in stages 4 and 5. The pipelined data path and control unit CPU is shown in Figure 13.2.

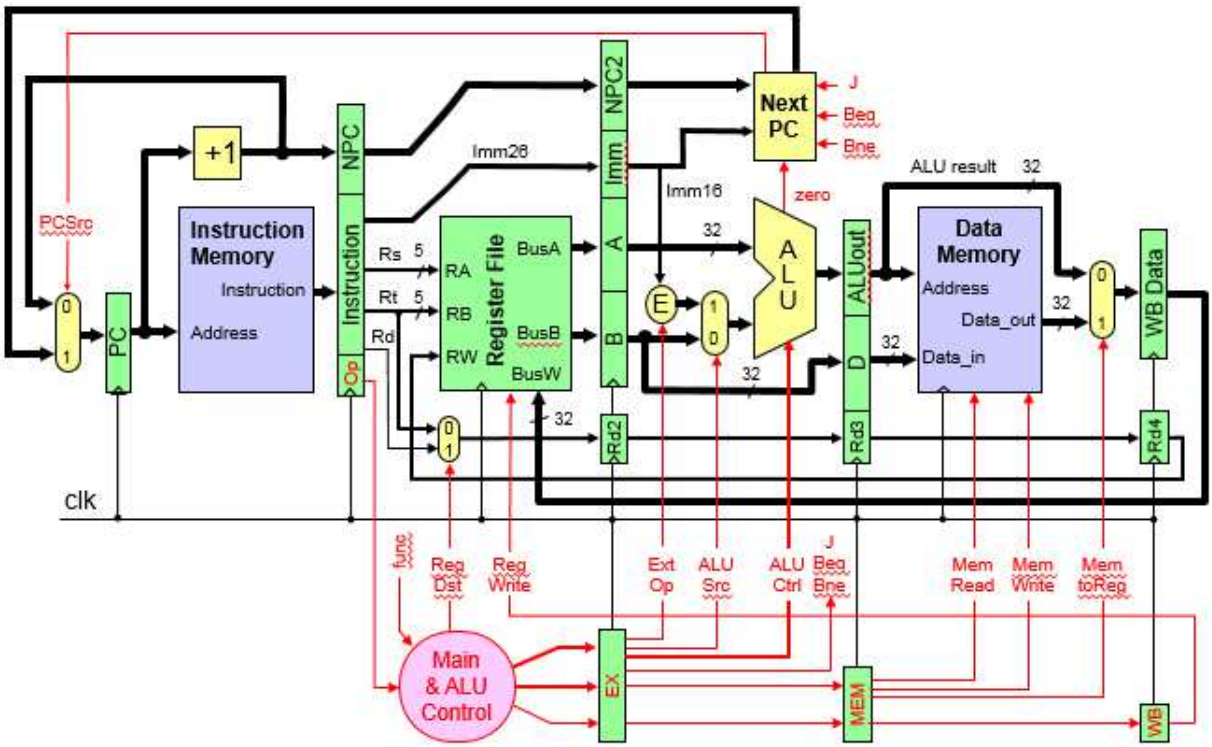

Figure 13.2: Pipeline Data Path and Control logic.

13.4 Pipeline Hazards

Hazards are situations that would cause incorrect execution if next instruction were launched during its designated clock cycle. Hazards can be classified into three main types:

1. Structural hazards
    - ✧ Caused by resource contention

- ✧ Using same resource by two instructions during the same cycle

2. Data hazards
   - ✧ An instruction may compute a result needed by next instruction
   - ✧ Hardware can detect dependencies between instructions

3. Control hazards
   - ✧ Caused by instructions that change control flow (branches/jumps)
   - ✧ Delays in changing the flow of control

Hazards complicate pipeline control and limit performance. Dependency between instructions causes a data hazard. An example of a data hazard is Read After Write – RAW Hazard. An example of a RAW hazard is given below. Given two instructions *I* and *J*, where *I* comes before *J*. Instruction *J* should read an operand after it is written by *I*.

I: add $s1, $s2, $s3 # $s1 is written

J: sub $s4, $s1, $s3 # $s1 is read

The RAW Hazard occurs when *J* reads the operand before *I* writes it.

Figure 13.3 shows a sample MIPS program with several RAW hazards. The result of sub instruction is needed by the add, or, and, & sw instructions. The instructions add & or will read old value of $s2 from reg file as the value of $s has not been updated in the reg file yet. During CC5, $s2 is written at the end of the cycle, and thus the old value is read. Thus, from this example we can see that any dependency between an instruction and any of the three following instructions will cause a RAW hazard.

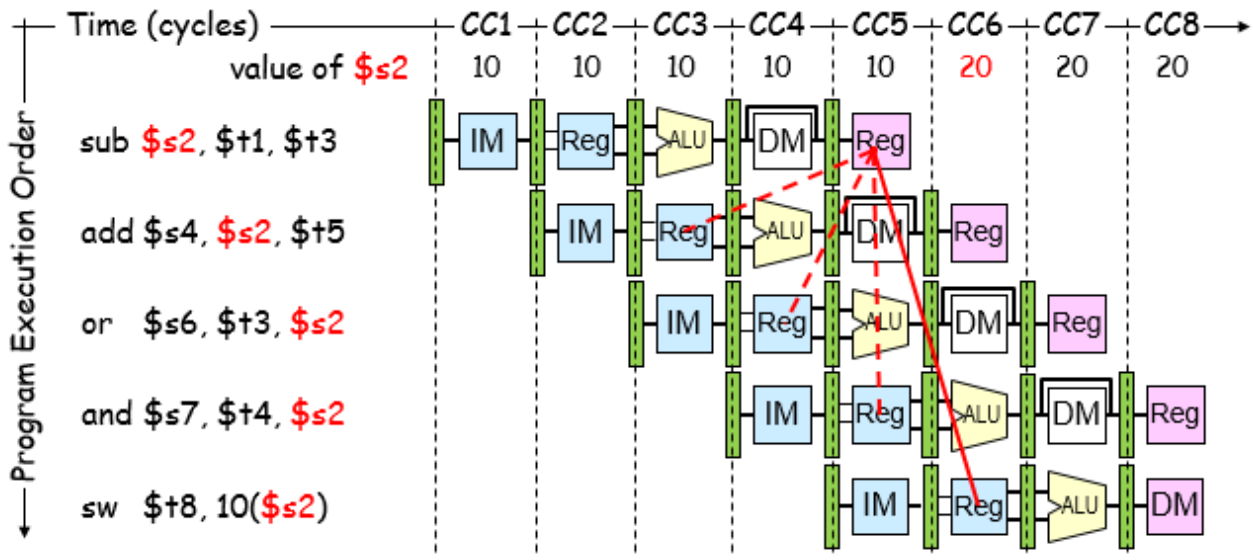

Figure 13.3: Example of RAW hazard.

13.5 Handling RAW Data Hazards

One way of handling RAW data hazards is to stall the pipeline until the required destination register is updated in the reg file. This requires freezing the execution of instructions that have such dependency for three clock cycles to all the reg file to be updated. Figure 13.4 illustrates an example of that. Due to the RAW dependency between the add and sub instructions, fetching the operands of the add instruction has to wait until register $s2 of the sub instruction is updated. This requires stalling the pipeline for three clock cycles from CC3 to CC5. Stall cycles delay the execution of the add instruction & fetching of the or instruction. The add instruction cannot read $s2 until beginning of CC6. The add instruction remains in the Instruction register until CC6 and the PC register is not modified until the beginning of CC6.

However, instead of stalling the pipeline and wasting clock cycles, RAW hazards can be handled by observing that the needed data is available in one of the stages 3 to 5 and can be used by forwarding it to stage 2 instead of waiting until the data is written to the reg file. This idea is illustrated in Figure 13.5.

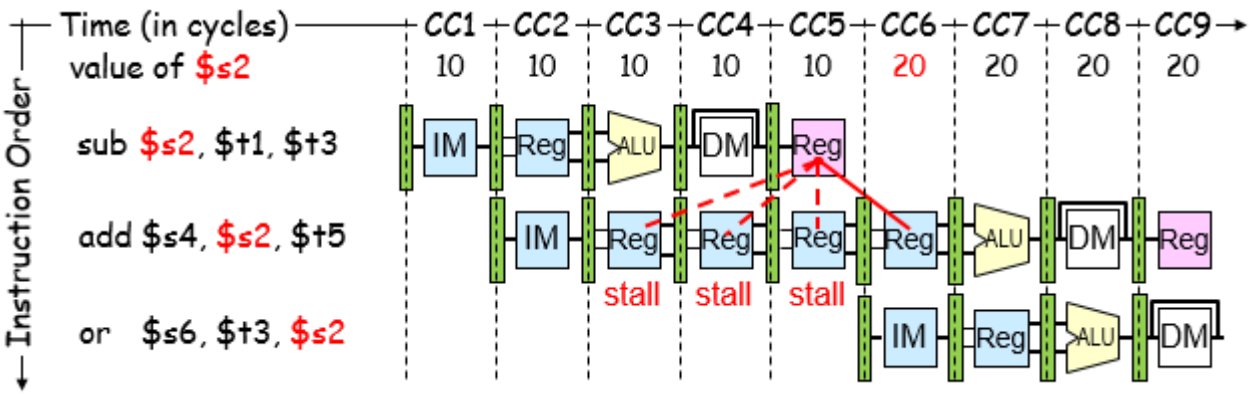

Figure 13.4: Pipeline stall due to RAW hazard.

The add instruction takes the content of $s2 from the ALU output. The or instruction takes the content of $s2 from the output of the DM stage. The and instruction takes the content of $s2 from the input of the reg file stage 5 and the content of $s6 from the ALU output at stage 2.

To implement forwarding, two multiplexers are added at the inputs of the A & B registers and data from ALU stage, MEM stage, and WB stage is fed back to these multiplexers as shown in Figure 13.6. Two signals ForwardA and ForwardB control forwarding as shown in Table 13.1.

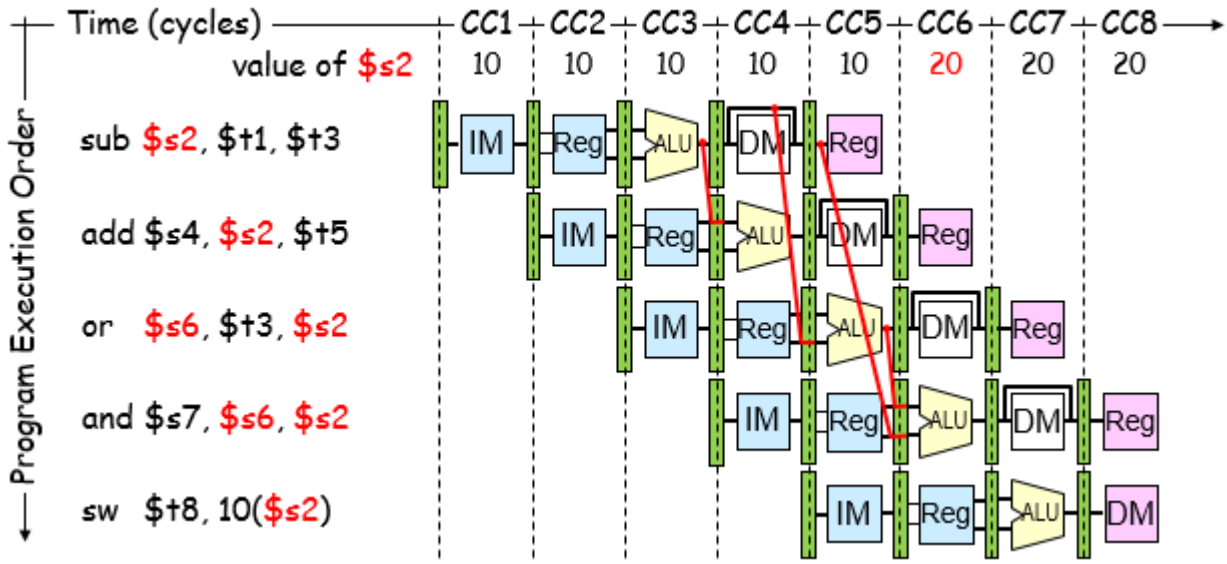

Figure 13.5: Example of data forwarding.

It should be observed that current instruction being decoded is in the Decode stage, the previous instruction is in the Execute stage, the second previous instruction is in the Memory stage and the third previous instruction in the Write Back stage. Thus, RAW data hazards detection conditions and the generation of the forwarding control signals can be done as follows:

If	((Rs != 0) and (Rs == Rd2) and (EX.RegWrite))	ForwardA ← 1
Else if	((Rs != 0) and (Rs == Rd3) and (MEM.RegWrite))	ForwardA ← 2
Else if	((Rs != 0) and (Rs == Rd4) and (WB.RegWrite))	ForwardA ← 3
Else		ForwardA ← 0

If	((Rt != 0) and (Rt == Rd2) and (EX.RegWrite))	ForwardB ← 1
Else if	((Rt != 0) and (Rt == Rd3) and (MEM.RegWrite))	ForwardB ← 2
Else if	((Rt != 0) and (Rt == Rd4) and (WB.RegWrite))	ForwardB ← 3
Else		ForwardB ← 0

The hazard detection and forwarding unit is shown in Figure 13.7.

Table 13.1: Data forwarding signals.

Signal	Explanation
ForwardA = 0	First ALU operand comes from register file = Value of (Rs)
ForwardA = 1	Forward result of previous instruction to A (from ALU stage)
ForwardA = 2	Forward result of 2nd previous instruction to A (from MEM stage)
ForwardA = 3	Forward result of 3rd previous instruction to A (from WB stage)
ForwardB = 0	Second ALU operand comes from register file = Value of (Rt)
ForwardB = 1	Forward result of previous instruction to B (from ALU stage)
ForwardB = 2	Forward result of 2nd previous instruction to B (from MEM stage)
ForwardB = 3	Forward result of 3rd previous instruction to B (from WB stage)

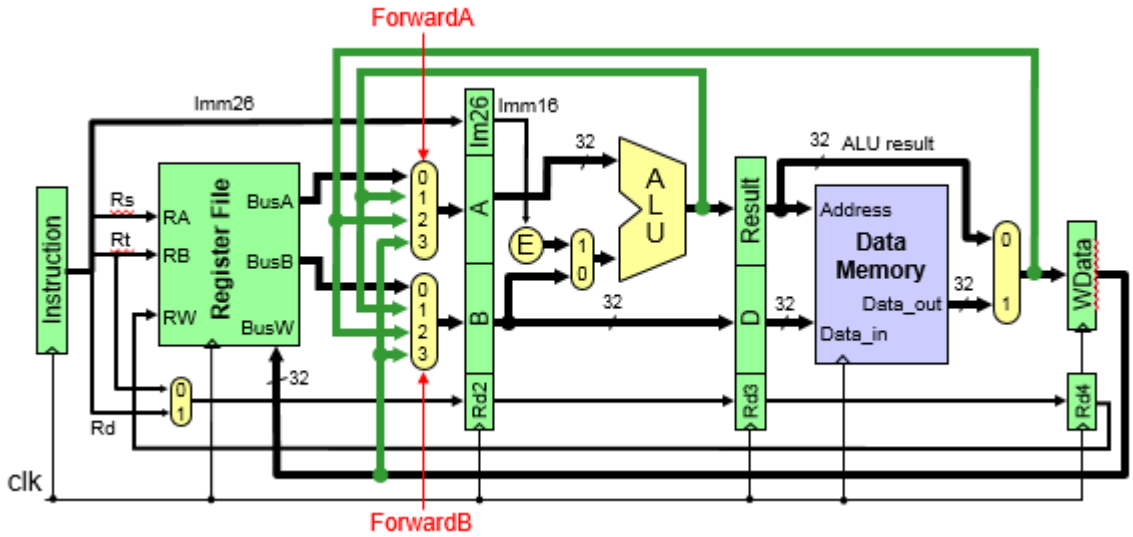

Figure 13.6: Implementation of data forwarding.

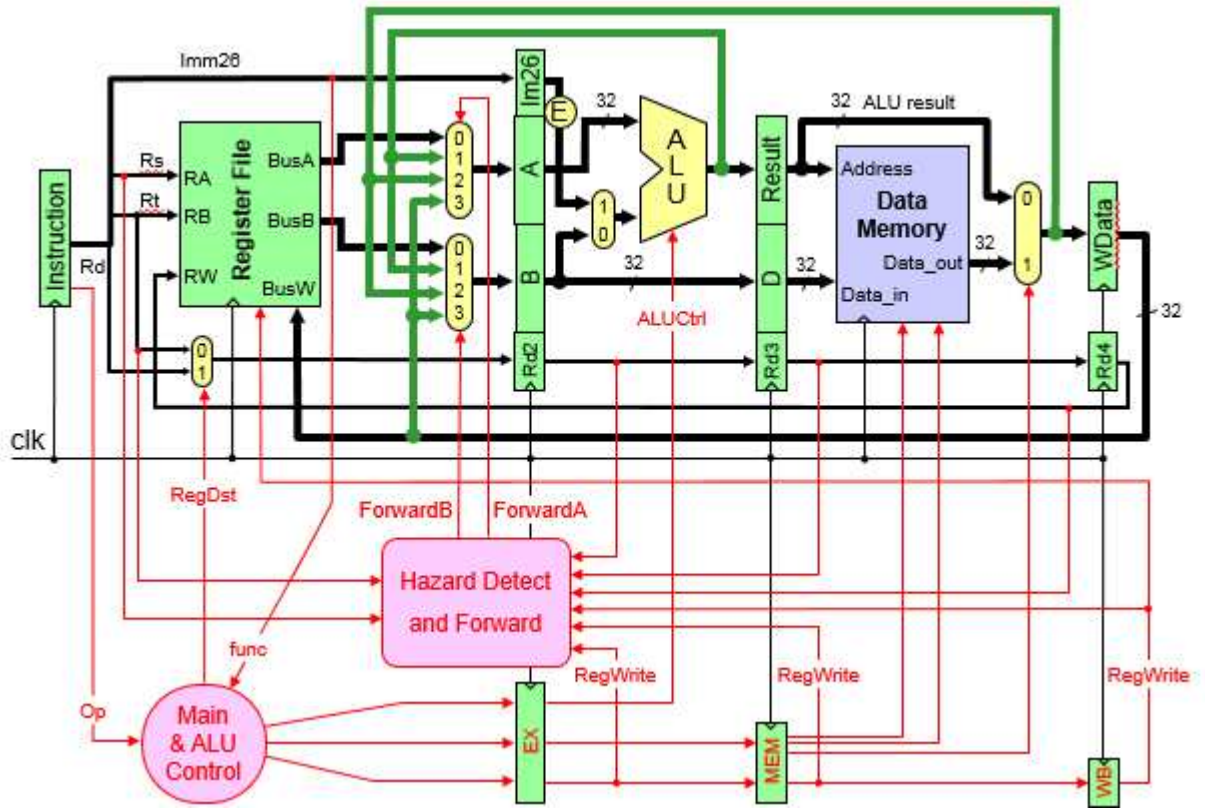

Figure 13.7: Hazard detection and forwarding unit.

13.6 In-Lab Tasks

1. Implement RAW hazard detection and the forwarding unit in your pipelined CPU design.
2. Add pipeline registers to your data path CPU design.
3. Add pipeline registers to pipeline the control signals in your CPU design.
4. Verify the correctness of your pipelined CPU design by executing the following instruction sequence:

```
ori   $s1, $0, 1
addi  $s2, $0, 2
xor   $s3, $s3, $s3
andi  $s4, $0, $0
addi  $s5, $s1, 5
add   $s6, $s1, $s2
sub   $s7, $s1, $s2
```

How many clock cycles your pipelined CPU takes to execute this program?

5. Add the two multiplexers needed to implement data forwarding.
6. Implement the forwarding unit and add it to your pipelined CPU.
7. Verify the correctness of your pipelined CPU design including data forwarding unit by executing the following instruction sequence:

```
ori   $s1, $0, 1
addi  $s2, $0, 2
ori   $s3, $0, 3
sub   $s4, $s3, $s1
add   $s5, $s4, $s2
or    $s6, $s4 $s5
and   $s7, $s3, $s4
sw    $s7, 10($s4)
```