University of Jordan
Computer Engineering Department
CPE439: Computer Design Lab
Experiment 9: Resolving Control Hazards

Description

In the previous experiment, student worked on resolving one out of several cases where data dependencies between instructions may cause data hazards in pipelining. In this experiment, students have to modify their pipelining implementation to accommodate for a new type of pipelining hazards; namely, control hazards.

Control hazards arise when executing program flow control instructions such as beq, j, jr, and jal. When these instructions are being executed (stored in the IF/ID register), the processor is fetching the following instruction (at PC+4). However, when execution is over (the decoding of the flow instruction is over and it is stored in the ID/EX register), the fetched instruction (stored in IF/ID register) might not be correct for conditional flow instructions (beq) if the condition evaluates to true. In this case, the processor should have fetched the instruction pointed-to by the branch address. Similarly, for unconditional flow instructions (j, jr, and jal), the fetched instruction is always wrong since it has to be fetched from the jump address for j and jal instructions, and from the address contained in one of the registers for the jr instruction.

In order to resolve this hazard, the fetched instruction in both cases has to be removed (flushed) from the pipeline. This can be implemented by clearing or flushing the IF/ID register asynchronously after the instruction is stored and the hazard is detected. Note how this affects the performance of the pipeline since it wastes one cycle.

In this experiment, students have to resolve control hazard by designing a hazard detection hardware that is capable of determining the need for flushing the fetched instruction or not based on the type of the instruction in the decode stage, and then incorporate it within the pipelined implementation done in experiment 8, as shown in Figure 1.

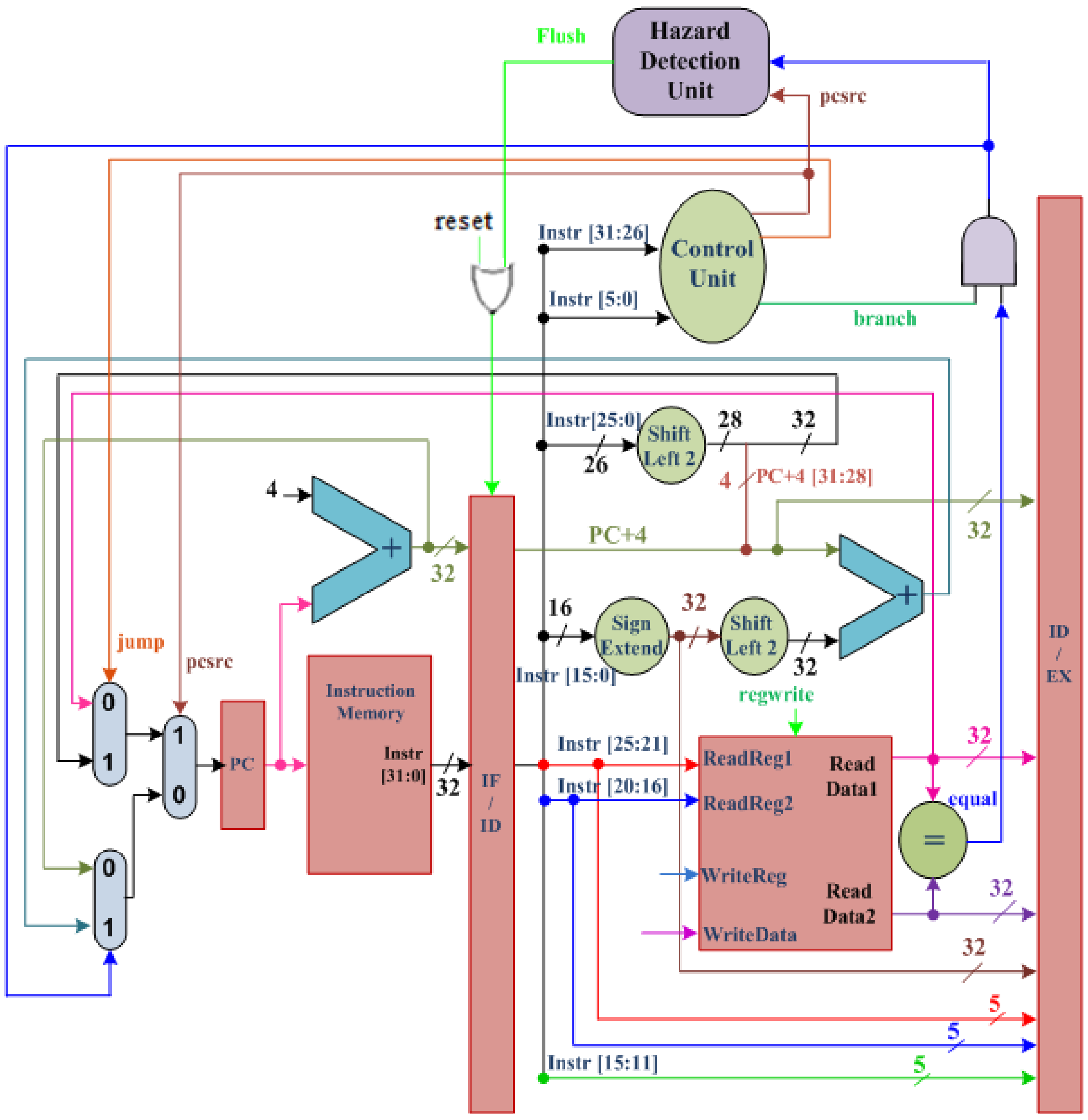
Procedure

1) ***(Prelab.)*** **Hazard Detection Unit**
   You need to build the hazard detection unit structurally. Your module should use the following template:

```
module HazardDetectionUnit(Flush, pcsrc, takenbranch);
  output Flush;
  input pcsrc, takenbranch;

 // implementation details are left to the student
endmodule
```

2) **The processor module**
   You need to modify the pipelined processor module by **adding the hazard detection unit** and **make the needed modifications**.



Hazard Detection Unit
Flush
pcsrc
reset
Instr [31:26]
Control Unit
Instr [5:0]
branch
Instr[25:0]
Shift Left 2
28
32
26
4
PC+4 [31:28]
4
PC+4
32
32
16
Sign Extend
32
Shift Left 2
32
Instr [15:0]
jump
pcsrc
0
1
1
0
0
1
PC
Instruction Memory
Instr [31:0]
32
IF / ID
regwrite
Instr [25:21]
Instr [20:16]
ReadReg1
ReadReg2
WriteReg
WriteData
Read Data1
Read Data2
32
equal
=
32
32
5
5
Instr [15:11]
5
ID / EX

Figure 1

Testing

- ***(Prelab.)*** Test your design for the pipelined processor by filling the instruction memory by the instruction sequence shown in Table 1.

Table 1. The Content of the Instruction Memory

Address	Instruction		Machine Code
00	LW	R1, 4(R0)	8C010004h
01	LW	R2, 12(R0)	
02	LW	R3, 20(R0)	
03	LW	R4, 28(R0)	
04	NAND	R5, R1, R3	
05	NORI	R6, R5, 1023	
06	SUB	R8, R4, R2	
07	JAL	11	
08	XOR	R7, R5, R6	
09	SW	R7, 8(R0)	
10	J	19	
11	ADDI	R8, R8, 2	
12	SW	R5, 4(R0)	
13	SW	R6, 24(R0)	
14	BEQ	R8, R3, -4	
15	SUB	R9, R8, R3	
16	JR	R31	
17	OR	R10, R7, R9	
18	SLT	R11, R9, R4	

- ***(Prelab.)*** Next, write a Verilog **test module** to test your processor module
  - ***Your timing diagram* should contain the following signals:**
    a) PC (The output of the program counter).
    b) The output of IFID register.
    c) The output for the registers R5, R6, R7, R8, R9, R10, R11.
    d) The output of forwarding unit (ForwardA, ForwardB).
    e) The input and the output of hazard detection unit.

  - ***Calculate number of cycles needed to execute the above code.***