

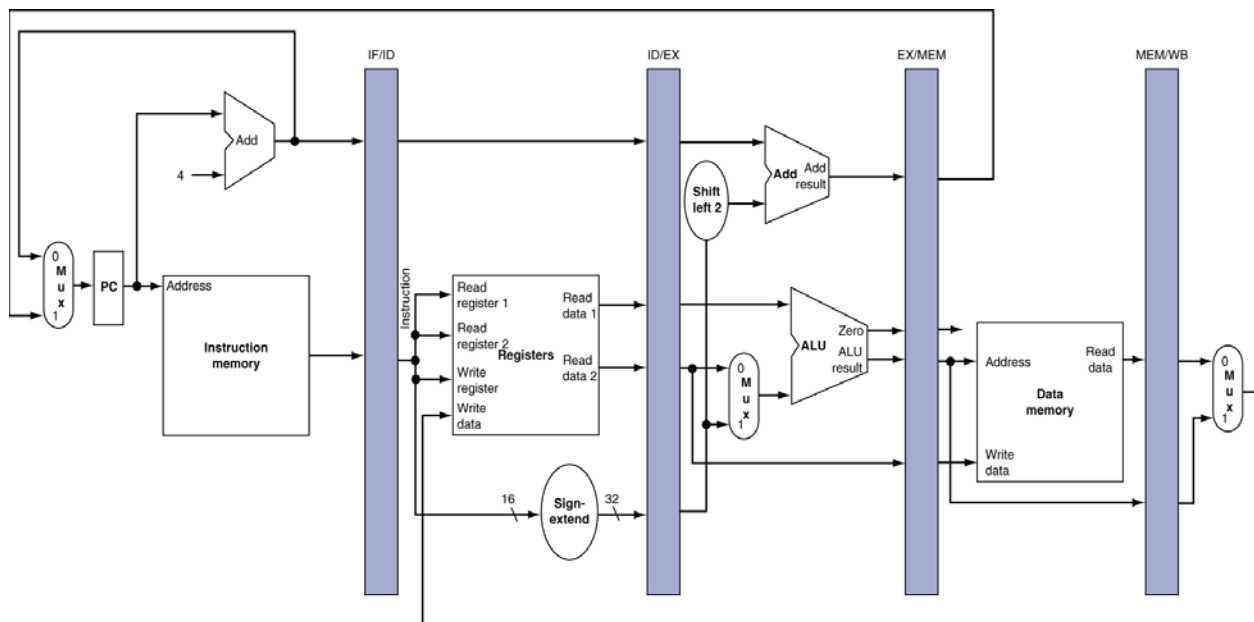
B443 Computer Architecture, Spring 2018
Project 2: Pipelined Processor Simulator

Released: Saturday, March 3, 2018

Due: Saturday, March 24, 2018 11:59 pm

Pipelined Processor Simulator

In this project you will implement a pipelined MIPS simulator in Java. The block diagram of the pipelined MIPS is shown below.



The stages of the pipelined processor are:

- IF, Instruction fetch
- ID, Instruction Decode
- EXE, ALU execution
- MEM, Data memory access
- WB, Register write back

The solution should be an extension of the Project 1:

- Use the same GUI (with few minor additions).
- Processor stages (IF, ID, EXE, MEM and WB) produced in the Project 1, are reused to produce respective pipelined stages.

In addition to the simulation of the pipelined processor execution, you should compare the following properties between your Project 1 and Project 2 solutions and **include the results in your final submission**:

- Instruction (number of instructions executed)
- Cycles (number of cycles since the beginning of program execution)
- IPC (number of instructions per cycle)

The "Print→IPC" menu option of your simulator can print these values to the console.

The simulator should detect data hazards and use forwarding and stalls to solve the hazards. The provided code already handles load-use and control hazards.

In the lectures we explained forwarding mechanism, which efficiently solves the data hazards between first and second and between first and third instructions. An example of the data hazard between first and second instruction is:

```
sub  $2, $1, $3
and  $12, $2, $5
```

This is type 1 data hazard, detected by checking:

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

An example of the data hazard between first and third instruction is:

```
sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
```

This is type 2 data hazard, detected by checking:

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

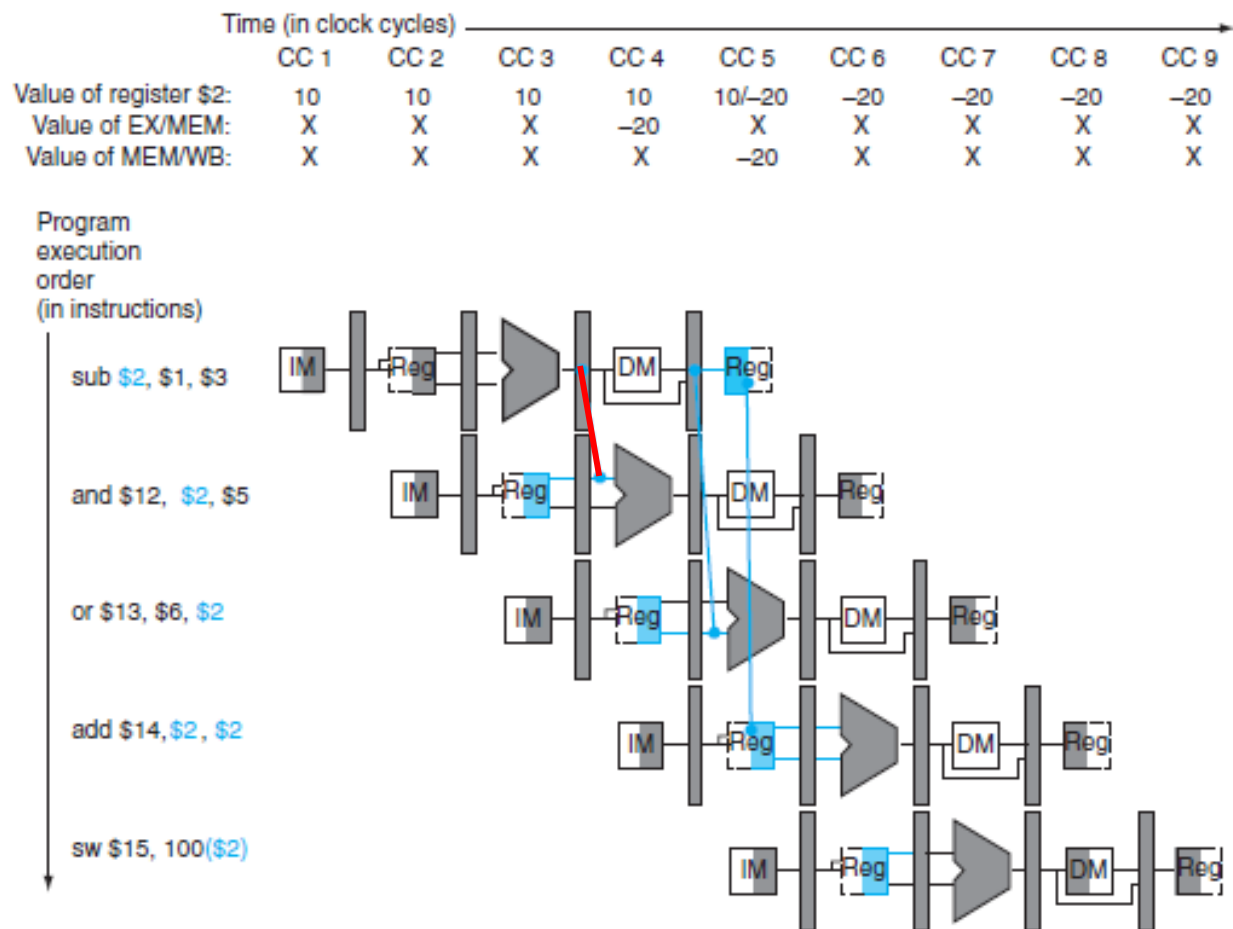
Here, we use the notation:

ID/EX: the pipeline register between the ID and EX stages

EX/MEM: the pipeline register between the EX and MEM stages

In the lecture, we explained that both type 1 (between first and second instructions) and type 2 (between first and third instructions) are solved without stalling the pipeline.

The solution for the type 1 data hazard requires forwarding directly from EXE stage of the first instruction to the EXE stage of the second instructions, indicated by the red line in the following diagram.

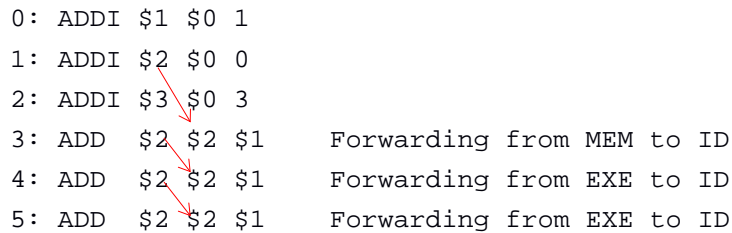


Test your implementation with the following program with type 2 data dependencies which are solved with forwarding but no stalls.

0: ADDI \$1 \$0 1	
1: ADDI \$2 \$0 0	
2: ADDI \$3 \$0 3	
3: ADD \$2 \$2 \$1	Forwarding from EXE (ALU) to ID, and from MEM to ID
4: ADD \$3 \$3 \$1	Forwarding from EXE (ALU) to ID
5: ADD \$2 \$2 \$1	Forwarding from EXE (ALU) to ID
6: ADD \$3 \$3 \$1	Forwarding from EXE (ALU) to ID
7: ADD \$2 \$2 \$1	Forwarding from EXE (ALU) to ID
8: ADD \$3 \$3 \$1	Forwarding from EXE (ALU) to ID

Test your implementation with the following program with type 1 data dependencies which are solved with forwarding and no stalls.

```
0: ADDI $1 $0 1
1: ADDI $2 $0 0
2: ADDI $3 $0 3
3: ADD $2 $2 $1    Forwarding from MEM to ID
4: ADD $2 $2 $1    Forwarding from EXE to ID
5: ADD $2 $2 $1    Forwarding from EXE to ID
```



This assignment is accompanied with:

- Executable pipelined simulator which you are supposed to design and implement.
- Source code for Project 1 solution.
- Project 2 partial solution (project_2.jar).

Partial solution: Hints and explanations

The explanation of each Java class in the project can be found in the description of Project 1. The following files contain sections marked “TODO” that must be filled in:

- **Control.java:** Unlike Project 1, where only one stage’s “work” method was invoked each cycle, for this project, we will invoke every stage’s “work” method each cycle. Hint: The “workListenerList” contains the list of objects implementing the “WorkListener” interface.
- **StageIF.java:** This stage can be stalled if we need to insert a bubble. Write code to check for a stall and do nothing if the stage is stalled.
- **StageID.java:** We will check for data hazards in this stage and indicate a stall if a data hazard is detected (“testDataHazards” static method). Rather than checking the registers in use by other stages, this stage “remembers” the last two instructions decoded for simplicity in the “last” attribute.

In the case of a stall, the “stageState” stores the state of the ID stage until the next cycle.

- **StageEXE.java:** The “getRegisterValue” static method performs forwarding if it is needed; otherwise register values are read directly from the register file.