# ECE 570
High-Performance Computer Architecture
Solutions Set #1
Winter 2014

[25 pts]

1- Consider the following piece of code written using MIPS instruction set:

| | | | Pipelined | Non-pipelined |
|---|---|---|---|---|
| Initialize: | DADD | R15, R0, R0 | 1 | 4 |
| | DADDI | R17, R15, #404 | 1 | 4 |
| Loop: | BEQ | R15, R17, Exit | 1 | 3 |
| | LW | R8, 2000(R15) | 1 | 5 |
| | DADD | R8, R8, R16 | 1 | 4 |
| | SW | R8, 1500(R15) | 1 | 4 |
| | DADDI | R15, R15, 4 | 1 | 4 |
| | J | Loop | 1 | 3 |
| Exit: | | | | |

Determine how much faster it would be to run the above code on a 5-stage pipeline compared to a non-pipelined datapath. Ignoring the pipeline cold-start delay and each instruction in the pipelined datapath requires 1 cycle to execute. On the other hand, the number of cycles required to execute an instruction in the non-pipelined datapath is indicated above. Assume both datapaths are running at a clock rate of 250 MHz.

(a) What is the CPU time required to execute the above code in a pipeline datapath?

The initialization phase requires 3 cycles (DADD, DADDI, and BEQ). The body of the loop consists of 6 instructions (BEQ to J). Since each loop is incremented by 4 and the limit is 404, the loop is executed 101 times. Therefore, the total number of cycles required is 3+(6*101)=609 cycles. The CPU time is given as

$$\text{CPUtime} = 609 \text{ cycles} \times (250 \times 10^{6} \text{ cycles/sec})^{-1} = 2.44 \times 10^{-6} \text{ sec}$$

(b) What is the average CPI for the above code running on the non-pipelined datapath?

| | | | |
|---|---|---|---|
| LW | => | 101/609 | @ 5 cycles |
| SW, DADD, DADDI | => | (303+2)/609 | @ 4 cycles |
| BEQ, J | => | (101+102)/609 | @ 3 cycles |

Therefore,

$$\text{CPI} = \frac{101}{609} \times 5 + \frac{305}{609} \times 4 + \frac{203}{609} \times 3 = 3.83$$

(c) How much faster is the pipeline version compared to the non-pipelined version?

Since,

$$\text{CPUtime} = 3.83 \text{ cycles/inst.} \times 609 \text{ insts.} \times (250 \times 10^{6} \text{ cycles/sec})^{-1} = 9.33 \times 10^{-6} \text{ sec}$$

pipelined version is 9.33/2.44 = 3.82 times faster.

[25 pts]

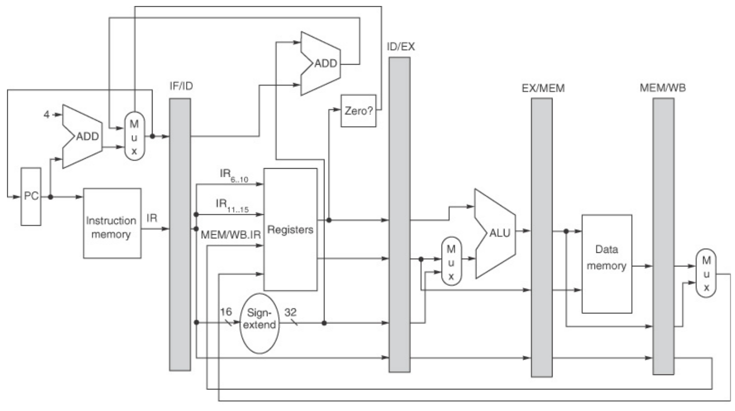
2- Consider the following two code examples where *ALU-store forwarding* will benefit on the datapath shown below.

Code 1 Code 2

```
…                          …
DADDI   R1, R1, #1         DADDI   R1, R1, #1
SW R1, 0(R3)               DSUB    R4, R3, R2
…                          SW R1, 0(R3)
                           …
```

(a) Show and explain why the datapath with the normal forwarding or bypassing (not shown) would not properly execute the above code sequences.

(b) Make the necessary modifications to the datapath shown below with normal forwarding or bypassing (not shown) to allow the code above to run with minimal stalling and show the execution timing for the modified design. Clearly explain your design.

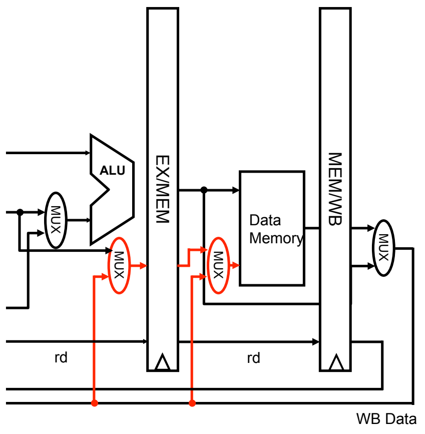


(a) The current datapath with normal forwarding and bypassing can only forward from either MEM or WB to EX. Thus, it cannot forward an ALU result to the MEM stage, and thus it has to forward through the register file by appropriately stalling SW. That is

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| DADDI R1, R1, #1 | IF | ID | EX | MEM | WB | | | |
| SW R1, 0(R3) | | IF | ID | stall | stall | EX | MEM | WB |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| DADDI R1, R1, #1 | IF | ID | EX | MEM | WB | | | |
| DSUB R4, R3, R2 | | IF | ID | EX | MEM | WB | | |
| SW R1, 0(R3) | | | IF | ID | stall | EX | MEM | WB |

(b) One way to solve this problem is to forward from the WB stage to the MEM stage by having a MUX at the input of Data Memory. This would solve the problem in example Code 1. In order to solve the problem in Code 2, Forwarding is performed from the MEM stage to the EX stage. These modifications are shown below.



Delay is not an issue since R1 is forwarded at the beginning of the WB stage to either the MEM stage or the EX stage. Thus, the forwarding can be done without additional stalls.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| DADDI R1, R1, #1 | IF | ID | EX | MEM | WB | | | |
| SW R1, 0(R3) | | IF | ID | EX | MEM | WB | | |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| DADDI R1, R1, #1 | IF | ID | EX | MEM | WB | | | |
| DSUB R4, R3, R2 | | IF | ID | EX | MEM | WB | | |
| SW R1, 0(R3) | | | IF | ID | EX | MEM | WB | |

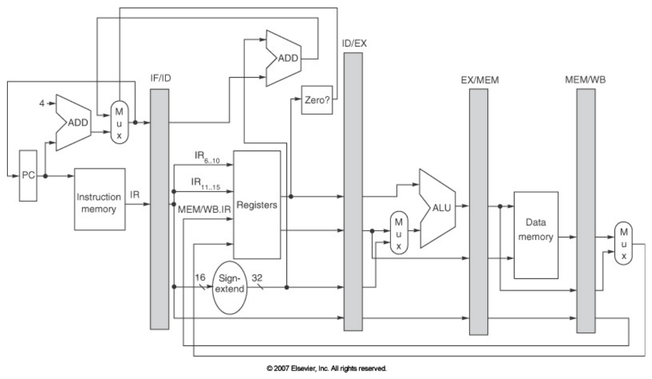
[25 pts]

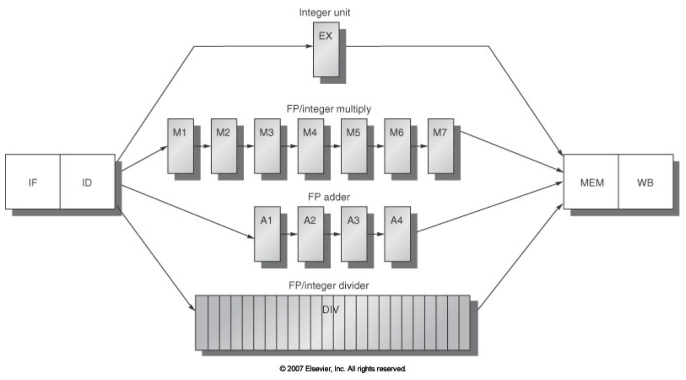
3- Consider the following piece of code executing on a 5-stage pipeline and the MIPS FP pipeline shown below with the *normal forwarding and bypassing hardware* (not shown):

- Branch is predicted not taken and BNEZ is taken.
- Contention for the WB stage is resolved in the ID stage.
- *There is ALU-store forwarding*, i.e., outputs from ALU can be forwarded from (beginning of) the WB stage to the input of the Data Memory in the MEM stage.
- *There is ALU-branch forwarding,* i.e., the pipeline is appropriately stalled and the outputof EX stage is forwarded from (beginning of) the MEM stage to the test for zero unit (i.e., "Zero?") in the ID stage.

Clearly show and explain the timing of this instruction sequence (one iteration of the loop plus the L.D instruction for the next iteration).

```
Loop: L.D       F0, 0(R2)
      MUL.D     F0, F0, F4
      L.D       F2, 0(R3)
      ADD.D     F2, F0, F2
      S.D       F2, 0(R1)
      DADDUI    R2, R2, #8
      DADDUI    R3, R3, #8
      DSUBU     R5, R4, R2
      BNEZ      R5, Loop
```





| Instruction | Clock Cycle | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| `L.D F0,0(R2)` | F | D | E | M | W | | | | | | | | | | | | | | | | | | | | | | | | | |
| `MUL.D F0,F0,F4` | | F | D | s | E | E | E | E | E | E | E | M | W | | | | | | | | | | | | | | | | | |
| `L.D F2,0(R3)` | | | F | s | D | E | M | W | | | | | | | | | | | | | | | | | | | | | | |
| `ADD.D F2,F0,F2` | | | | | F | D | s | s | s | s | s | E | E | E | E | M | W | | | | | | | | | | | | | |
| `S.D F2, 0(R1)` | | | | | | F | s | s | s | s | s | D | s | s | s | E | M | W | | | | | | | | | | | | |
| `DADDUI R2,R2,#8` | | | | | | | | | | | | F | s | s | s | D | E | M | W | | | | | | | | | | | |
| `DADDUI R3,R3,#8` | | | | | | | | | | | | | | | | F | D | E | M | W | | | | | | | | | | |
| `DSUBU R5,R4,R2` | | | | | | | | | | | | | | | | | F | D | E | M | W | | | | | | | | | |
| `BNEZ R5,Loop` | | | | | | | | | | | | | | | | | | F | D | s | E | M | W | | | | | | | |
| `L.D F0,0(R2)` | | | | | | | | | | | | | | | | | | | F | D | F | D | E | M | F | | | | | |

Branch resolved

Stall needed for loading of F0 followed by its immediate use

Stalled to forward F0

Since there is ALU-store forwarding, S.D need to be stalled until F2 can be forwarded from WB to MEM stages.

Since there is branch forwarding, R5 can be forwarded from MEM to EX stages.

Fetch resumes from the correct path

[25 pts]

4- (a) Consider the code sequence shown in Figure 2.35 of the text. What is the performance (in cycles per loop iteration) of the code? Assume a single pipeline machine with the following assumptions: (a) There are multiple FUs and results can be immediately forwarded from one execution unit to another, or to itself; (b) the only reason and execution pipeline would stall is to observe a true dependence.

```
Loop:   LD F2, 0(Rx) {1 + 3}                  1
        <stall>                               2
        <stall>                               3
        <stall>                               4
        MULTD F2, F0, F2 {1 + 4}              5
        <stall>                               6
        <stall>                               7
        <stall>                               8
        <stall>                               9
        DIVD F8, F2, F0 {1 + 10}              10
        LD F4, 0(Ry) {1 + 3}                  11
        <stall due to LD latency>             12
        <stall due to LD latency>             13
        <stall due to LD latency>             14
        ADDD F4, F0, F4 {1 + 2}               15
        <stall due to DIVD latency>           16
        <stall due to DIVD latency>           17
        ADDI Rx, Rx, #8 {1}                   18
        SUB R20, R4, Rx {1}                   19
        <stall due to DIVD latency>           20
        SD F4, 0(Ry) {1 + 1}                  21
        ADDD F10, F8, F2 {1 + 2}              22
        ADDI Ry, Ry, #8 {1}                   23
        BNZ R20, Loop {1 + 1}                 24
        <stall due to BNZ>                    25
        ____
        25 cycles per loop iteration
```

Here is one possible scheduling. In order to minimize the number of required cycles, the critical instructions (i.e., ones that have RAW dependencies) are scheduled first (indicated by red arrows) based on their required latencies. Thus, you should have something similar up until cycle 20. For cycles 21 and 22, it does not matter whether ADDD or SD is scheduled first. Then, the only instructions that are left are the two ADDIs, SUB, and BNZ. Incrementing of Rx and Ry by 8 can be done any time as long as it is after the usage of Rx and Ry, which happens to be after the first LD and SD, respectively. Thus, ADDI Ry,Ry,#8 is scheduled after SD with 1 cycles latency in between the two instructions, and ADDI Rx,Rx,#8 and SUB R20, R4, Rx together are scheduled somewhere between cycles 12-14 or 16-20. This resulted in 25 cycles per iteration.

(b) Consider a multiple issue design. Reorder the instructions to improve the performance of the code in Figure 2.35. What is the performance (in cycles per loop iteration) of the code? Assume the two-pipeline machine with the following assumption: (a) Each pipeline is capable of beginning execution of one instruction per cycle and enough fetch/decode bandwidth in the front-end so that it will not stall the executions: (b) results can be immediately forwarded from one execution unit to another, or to itself; and (c) the only reason and execution pipeline would stall is to observe a true dependence.

| **Execution pipe 0** | **Execution pipe 1** | |
|---|---|---|
| Loop: LD F2, 0(Rx) {1 + 3} | LD F4, 0(Ry) {1 + 3} | 1 |
| <stall for LD latency> | <stall for LD latency> | 2 |
| <stall for LD latency> | <stall for LD latency> | 3 |
| <stall for LD latency> | <stall for LD latency> | 4 |
| MULTD F2, F0, F2 {1 + 4} | ADDD F4, F0, F4 {1 + 2} | 5 |
| <stall for MULTD latency> | <stall for ADDD latency> | 6 |
| <stall for MULTD latency> | <stall for ADDD latency> | 7 |
| <stall for MULTD latency> | SD F4, 0(Ry) | 8 |
| <stall for MULTD latency> | <nop> | 9 |
| DIVD F8, F2, F0 {1 + 10} | <nop> | 10 |
| <stall for DIVD latency> | <nop> | 11 |
| <stall for DIVD latency> | <nop> #ops: 11 | 12 |
| <stall for DIVD latency> | <nop> | 13 |
| <stall for DIVD latency> | <nop> | 14 |
| <stall for DIVD latency> | <nop> | 15 |
| <stall for DIVD latency> | <nop> | 16 |
| <stall for DIVD latency> | <nop> | 17 |
| <stall for DIVD latency> | <nop> | 18 |
| ADDI Rx,Rx,#8 | ADDI Ry,Ry,#8 | 19 |
| SUB R20,R4,Rx | <nop> | 20 |
| ADDD F10,F8,F2 | BNZ R20,Loop | 21 |
| <stall due to BNZ> | <stall due to BNZ> | 22 |

This is done in a similar manner as case (a) except there are two pipelines. Thus, two sequences of instructions are identified and scheduled to the pipelines. Instructions that update Rx are scheduled into slots that would have been stalls due to the RAW dependency on F8. Once pipeline 1 is scheduled, instruction that updated Ry can be scheduled into empty slots in pipeline 2. Finally, BNZ is scheduled onto pipeline 2 such that the total number of cycles does not increase.