Zachary Seymour
CS 520–03
Homework 1
September 30, 2013

> **1** Before we start this code sequence, assume that the registers R1 through R6 contain, respectively, the values of 100, 200, 300, 400, 500 and 600. Assume further that the instructions are 4 Bytes wide, the memory is byte-addressable (that is, each byte in the memory has a unique address) and the address of the first `LOAD` instruction is 5000. This means that the `ADD` instruction is stored at address 5004, the SUB at address 5008 and so on. The `LOAD` and `STORE` write and retrieve one 4-Byte wide memory word, whose effective address is computed by the `LOAD` and `STORE`. Assume that the contents of the memory words targeted by the first and second `LOAD` instructions are 3205 and 6409, respectively.
> Assume that this code fragment is executed on a five-stage in-order pipeline with data forwarding, as we discussed in class. Determine how many cycles would it take to execute this code and for each cycle, show the following: contents of the register file, data at the ALU inputs, data at the ALU output.

This code, considering the delay for the `LOAD` instruction, would require 10 cycles to execute.

Cycle 1

R1	100
R2	200
R3	300
R4	400
R5	500
R6	600

- ALU inputs at Cycle 1: Unknown
- ALU output at Cycle 1: Unknown

Cycle 2

R1	100
R2	200
R3	300
R4	400
R5	500
R6	600

- ALU inputs at Cycle 2: Unknown
- ALU output at Cycle 2: Unknown

Cycle 3

R1	100
R2	200
R3	300
R4	400
R5	500
R6	600

- ALU inputs at Cycle 3: Unknown
- ALU output at Cycle 3: Unknown

Cycle 4

R1	100
R2	200
R3	300
R4	400
R5	500
R6	600

- ALU inputs at Cycle 4: Unknown
- ALU output at Cycle 4: Unknown

Cycle 5

R1	100
R2	200
R3	300
R4	400
R5	500
R6	600

- ALU inputs at Cycle 5: R5 = 100 and R1=3205
- ALU output at Cycle 5: Unknown

Cycle 6

R1	3205
R2	200
R3	300
R4	400
R5	500
R6	600

- ALU inputs at Cycle 6: R4 = 3305 and R6=600
- ALU output at Cycle 6: 3305

Cycle 7

R1	3205
R2	200
R3	300
R4	400
R5	500
R6	600

- ALU inputs at Cycle 7: None
- ALU output at Cycle 7: $3305 - 600 = 2705$

Cycle 8

R1	3205
R2	200
R3	300
R4	3305
R5	500
R6	600

- ALU inputs at Cycle 8: None
- ALU output at Cycle 8: None

Cycle 9

R1	3205
R2	200
R3	300
R4	2705
R5	500
R6	600

- ALU inputs at Cycle 9: None
- ALU output at Cycle 9: None

Cycle 10

R1	3205
R2	200
R3	300
R4	2705
R5	500
R6	600

- ALU inputs at Cycle 10: None
- ALU output at Cycle 10: None

Cycle 11

R1	3205
R2	200
R3	300
R4	6409
R5	500
R6	600

- ALU inputs at Cycle 11: None
- ALU output at Cycle 11: None

2 Consider the execution of the following code fragment on a 5-stage in-order pipelined processor augmented with data forwarding logic. To avoid pipeline resource conflicts, assume that every instruction goes through every pipeline stage even if some stages may not be required for them. For example, the arithmetic instructions still go through the memory stage, where their generated results are simply moved from the output latch of the ALU stage to the output latch of the memory stage. Also assume that the execution stage for each instruction takes a single cycle.

2a Identify all of the data dependencies in this code.

The second argument of the first `ADD` instruction is dependent on the result from the `SUB`. The location accessed for the `STORE` is dependent on that produced by the first `ADD`, and the value loaded into R3 is dependent on the `STORE` before it. Finally, the last `ADD` must wait for the load to complete for its first operand.

2b Which of these dependencies would impact the correctness of this program if data forwarding was not implemented and no other mechanism to handle data dependencies was in place?

The dependency on `SUB` from the first `ADD` would impact the correctness, because `ADD` would be fetching the value from R4 before `SUB` had written the new correct value. Also, without hazard detection or the like, the second `ADD` would not have the correct first parameter without waiting on `LOAD` to complete execution.

2c How many cycles does it take to execute the above code? Explain your answer.

The above code requires 10 cycles to complete. In particular, with data forwarding, the first four instructions complete in normal pipelined fashion. With hazard detection, a bubble must be introduced on the final `ADD` instruction to wait for `LOAD` to complete, so it must wait one additional instruction longer until the value is written to the register file, resulting in 10 total cycles.

2d For *each processing cycle*, show which registers are read and written.

Cycle 1
- Read: None
- Written: None

Cycle 2
- Read: R2 and R5
- Written: None

Cycle 3
- Read: R2 and R4
- Written: None

Cycle 4
- Read: R4
- Written: None

Cycle 5
- Read: R4
- Written: R4

Cycle 6
- Read: R3 and R5
- Written: R4

Cycle 7
- Read: R3 and R5
- Written: None

Cycle 8
- Read: None
- Written: R3

Cycle 9
- Read: None
- Written: None

Cycle 10
- Read: None
- Written: R5

> **2e** For *each processing cycle*, list all pairs of registers that are being compared by the forwarding logic. Show how many of these comparisons result in a match. Also indicate if any of the matches are ignored for the purpose of controlling the MUXes in front of the ALU and explain why.

Cycle 1 Registers: None

Cycle 2 Registers: none

Cycle 3 Registers:

(a) R4
(b) R4
(c) R4
(d) R6

Both the first-third and second-third pairs match.

Cycle 4 Registers:

(a) —
(b) R4
(c) R4

(d) R3

The second and third match.

Cycle 5 Registers:

(a) —
(b) —
(c) R3
(d) R4

No matches.

Cycle 6 Registers: None

Cycle 7 Registers: None

Cycle 8 Registers: None (no future commands to check)

Cycle 9 Registers: None

Cycle 10 Registers: None

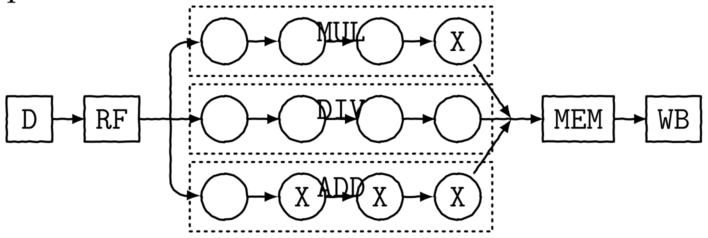
3 With the use of multiple function units, we can have instruction completing out-of-order. Consider the two following proposals for guaranteeing in-order completions in a modified version of a simple pipeline that has the three following types of function units:

- A multiplier pipelined into 3 stages, each with a delay of one cycle,
- A divider, pipelined into 4 stages, each with a delay of one cycle,
- A single stage integer unit for implementing all other arithmetic and logical operations,

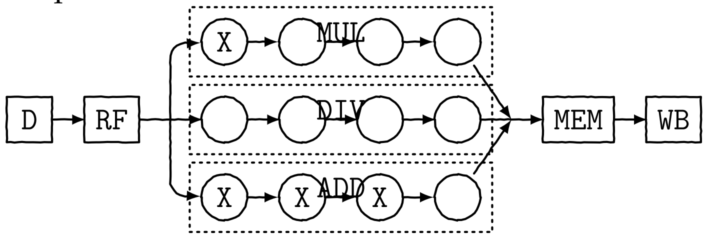
Following their transit through the function units, instructions still go through a single cycle MEM stage and a WB stage, as in the original pipeline. Suitable interlocking logic is present in the D/RF stage, that precedes these function units.
Proposal A: Add dummy delay stages, each with a delay of one cycle, at the end of the multiplier and integer FU to increase the combined number of stages that all instructions go through to 4 cycles (including the delays of the added dummy stages).
Proposal B: This is similar to Proposal A, but delay stages are added at the front of the integer FU and the multiplier.

3a Draw the stages, including the delay stages and the inter-stage connections. What is the instruction processing latency?

Proposal A:



Proposal B:



Where `X` designates a dummy stage. The latency is thus 6 cycles before a result is available for data forwarding.

3b Explain why both solutions ensure in-order updates to the registers.

Both proposals constrain the three function units to complete in exactly the same number of cycles, which insures the results of the operations reach the `MEM` stage in precisely the same order in which they were issued to the FU, resulting in in-order register updates.

3c Assuming that each proposal is implemented with forwarding logic in place, compare and contrast these two solutions.

The benefits of either proposal would, in part, depend on the typical load and instruction order the pipeline will receive. If either `MUL` or `ADD` are waiting on `DIV`, Proposal B seems more beneficial: during the dummy stages, these instructions would be stalled anyway, waiting for the result from `DIV`. This way, `ADD` would need only one additional cycle, though `MUL` would need 3.

On the other hand, with Proposal A, a `DIV` waiting on either `MUL` or `ADD` would be able to have its result forward more quickly, decreasing the latency of that dependency.

3d Estimate the total number of comparators needed in either design to forward register operand values to waiting instructions. `MUL`, `DIV`, other logical and arithmetic operations and `LOAD` all have at most 2 source registers each, while `STORE` can have up to 3 source registers. Do not consider forwarding from a `LOAD` to a `STORE`.

We will first need the four comparators originally discussed in the lecture on the forwarding unit. These will cover, for example, cases where `ADD` or `MUL` is `DIV` to finish execution and reach the `MEM` stage. Comparators will also be needed for inter-unit forwarding. For instances, an `ADD` or `MUL` may finish early and be able to stop a `DIV` from stalling and proceed with execution. This adds an additional 6 (two per FU). We will thus need approximate 10 comparators in our forwarding unit to properly dispatch values.