EECS 151/251A Discussion 6

03/16/2018

Announcements

- Homework 6 solutions posted
- Midterm 2 next Thursday in class slot with extra time (like Midterm 1)
- Catch-up discussion next week?

Midterm coverage

You are responsible for material covered in homeworks, labs, discussions

Delay, wires, power, etc (lectures 9 - 12)

- Review Taehwan’s discussion (5) for summary and overview of topics
- Understand homework questions, answers, and concepts
- Midterm questions won’t be as hard

Microprocessor design, instructions, logic elements (lectures 13 - 17)

Processor vocab + basics

ISA, ALU, Immediate, Byte, Half-word, Word

Instruction fetch, instruction decode, ALU op(eration), Memory op, Write back

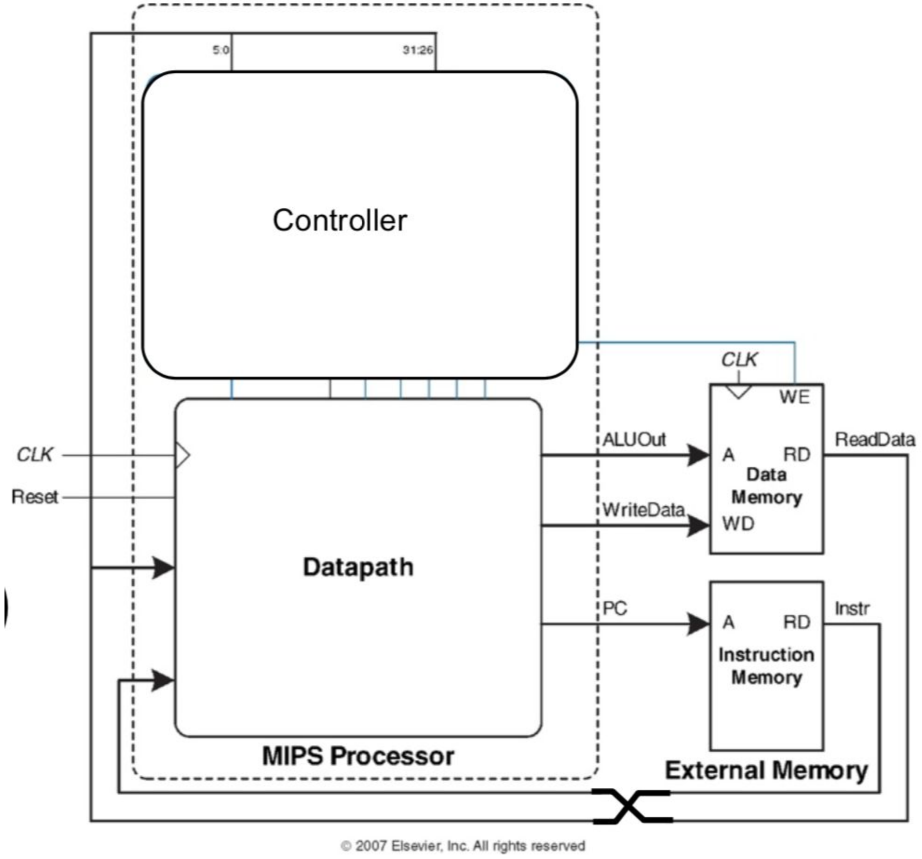
Processor vocab + basics

Always at least:

datapath + controller + memory

Number of execution steps (stages) is flexible. Typical RISC-V:

1. Instruction fetch
2. Instruction decode
3. ALU op(eration)
4. Memory op
5. Write back



Data Hazards

Named after what should happen *if nothing goes wrong:*

- Read-after-Write (RAW)
- Write-after-Read (WAR)
- Write-after-Write (WAW)
- What about Read-after-read (RAR) ?

```
R2 <- R1 + R3
R1 <- R4 + R5
```

```
R1 <- R2 + R3
R4 <- R1 + R2
```

```
R1 <- R2 + R3
R4 <- R5 + R6
R1 <- R6 + R7
```

Other hazards

Control

- When branching, processor does not know what the next instruction will be until some time later
- What should it run next?

Structural

- When things don’t fit in places at the same time
- e.g. if memory writes take a long time and we can’t issue multiple simultaneous requests

Question time

You are given the assembly code shown below, and it is run on a RISC-V processor with a typical 5 stage pipeline (instruction fetch — decode — execute — memory access — writeback).

```
add x1, x2, x3
add x5, x4, x1
```

(a) Explain what kind of hazard is present in the code above, and how many extra cycles are introduced if the processor has no data-forwarding. Assume all state elements in the processor are positive edge triggered.

There is a read-after-write data hazard present, involving the value written to the $x1$ register by the first instruction. We can draw a pipeline diagram to analyze this situation:

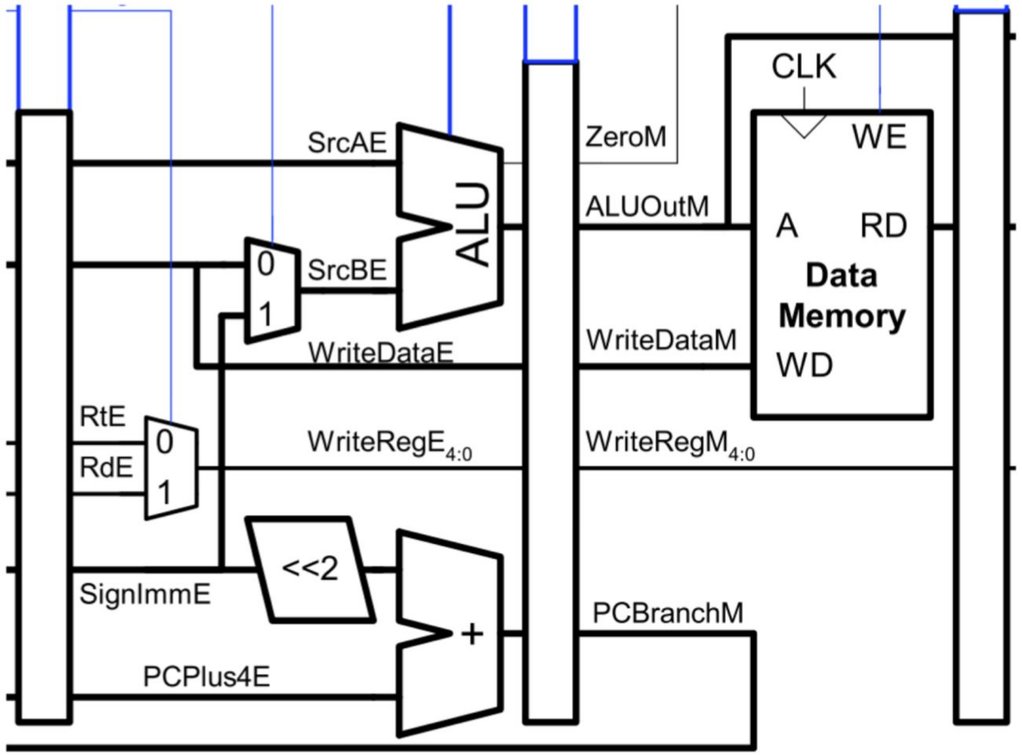
Cycle	1	2	3	4	5	6	7	8
Inst 1	IF	D	EX	M	WB			
Inst 2		IF	D	EX	EX	EX	M	WB

Since the result of the first instruction isn't available to the next instruction until after it has been written to the regfile (at the end of cycle 5), the second instruction has to stall until it is available.

2 extra cycles are needed due to the data hazard.

(b) Assuming the register file is written on the positive edge of the clock, how many cycles would the two instructions in total take? From the pipeline diagram above, it can be seen that 8 total cycles are required.

(c) To improve the performance, forwarding is implemented such that the inputs to the ALU can be pulled from the memory access stage. Now, how many extra cycles are caused by the hazard? How would you modify the part of the datapath shown below for this to be possible? Name any new control signals you have added.



Now the data needed by instruction 2 is available in cycle 4 and can be forwarded immediately.

Cycle	1	2	3	4	5	6	7	8
Inst 1	IF	D	EX	M	WB			
Inst 2		IF	D	EX	M	WB		

No extra cycles are caused by this hazard now.

We can modify the datapath by routing `ALUOutM` from the memory access stage to 2 muxes that drive `SrcAE` and `SrcBE`. The muxes are controlled by one control signal each and if the control signal is 1, `ALUOutM` is forwarded to the respective ALU inputs. We can call these control signals `ALUFwdA` and `ALUFwdB`.

(d) Now consider this set of instructions:

```
lw x1, 0(x2)
add x5, x4, x1
```

Assuming an asynchronous read data memory, how many cycles will this set of instructions take to execute? Identify any data hazards.

There is a data hazard concerning register `x1` similar to part (a). As a result these instructions will take 8 cycles to execute without any additional forwarding paths.

(e) If you could add another forwarding path from the output `RD` of the data memory, how many cycles will these instructions take to execute? What could be a disadvantage of forwarding from the output of the data memory versus from the pipeline register clocking `RD`?

We could again reduce the total cycles down to 6. However, directly feeding the output of the data memory to the execute stage will likely yield a long ciritical path which could un-balance the pipeline and negatively impact the max frequency of operation.

Next question

Imagine that you have a simple 3 stage in-order pipeline with the following stages:

1. Instruction fetch and decode
2. Execute
3. Write back

Registers are read in the first stage, and written to in the third stage.

```
add x0,x1,x2
sub x2,x3,x4
add x2,x3,x4
or  x3,x2,x0
and x4,x1,x0
xor x2,x1,x4
add x1,x2,x0
```

Assuming we can forward the result of an ALU operation from stage 3 to stage 2, how many cycles does the block of code take to execute?

```
add x0,x1,x2
sub x2,x3,x4
add x2,x3,x4
or  x3,x2,x0
and x4,x1,x0
xor x2,x1,x4
add x1,x2,x0
```

```
Cycle | IF  | EX  | WB  |
1     | add |  -  |  -  |
2     | sub | add |  -  |
3     | add | sub | add |
4     | or  | add | sub |
5     | and | or  | add |
6     | xor | and | or  |
7     | add | xor | and |
8     |  -  | add | xor |
9     |  -  |  -  | add |
```

9 cycles

What is the CPI?

9 cycles/7 instructions = 1.26 CPI

If the critical path is 1.2 ns, how long does the code take to execute?

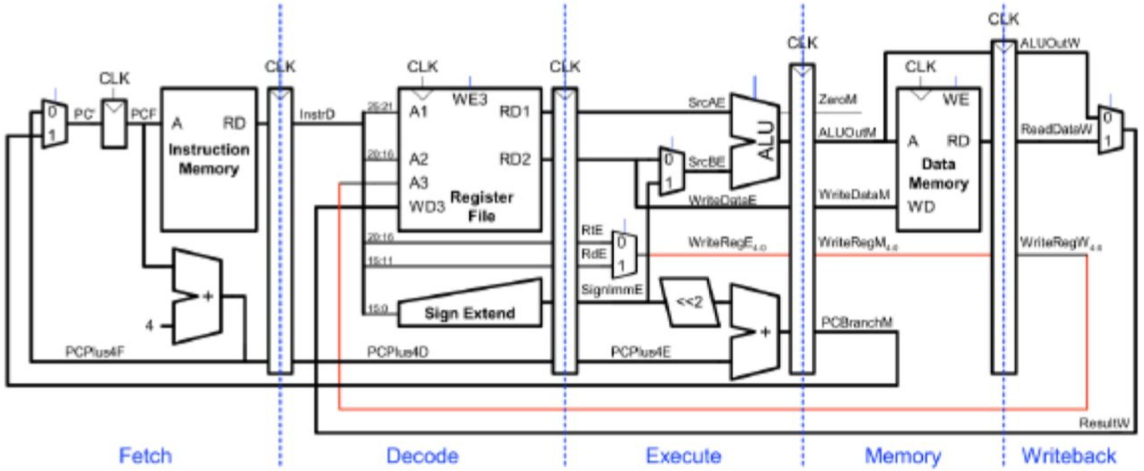
1.2 * 9 = 10.8 ns

Next question

You are to add a *store with postincrement* instruction to a 5-stage RISC-V processor pipeline. The instruction **swinc** updates the index register to point to the next memory word after completing the store. **swinc rt, imm(rs)** is equivalent to the following two instructions:

```
sw rt, imm(rs)
addi rs, rs, 4
```

(a) How would you modify the following datapath to accomodate this instruction? Try to add as little hardware as possible.



Since the main ALU is busy with calculating the store address (`imm + rs`), it can't be used for adding 4 to `rs`. However, the adder that normally computes the jump/branch address in the same stage is free during this instruction. Therefore, we can reuse that adder to implement this instruction by adding a mux to its input which is fed with the argument A of the ALU. We also need to modify the datapath to pass the output of that adder all the way through the pipeline, and add a mux to select it as `ResultW` during writeback.

(b) The following assembly program uses the new instruction; what hazard do you see? Modify the datapath you created in part (a) to eliminiate the stalls introduced.

```
swinc x3, 0x10(x2)
add x3, x2, x5
```

There is a data hazard from the first instruction writing `x2` to the second instruction using `x2` in the execute stage. To resolve this hazard, we need to add another forwarding path from the output of the pipeline register that holds the branch/jump adder's result to each of the ALU's inputs through a mux.

References

- Digital Design and Computer Architecture, David M. Harris & Sarah L. Harris
- Discussions from EECS151/251A Fall 2017, George Alexandrov
- Homeworks from EECS151/251A Fall 2016, Spring 2017, Fall 2017
- Wikipedia