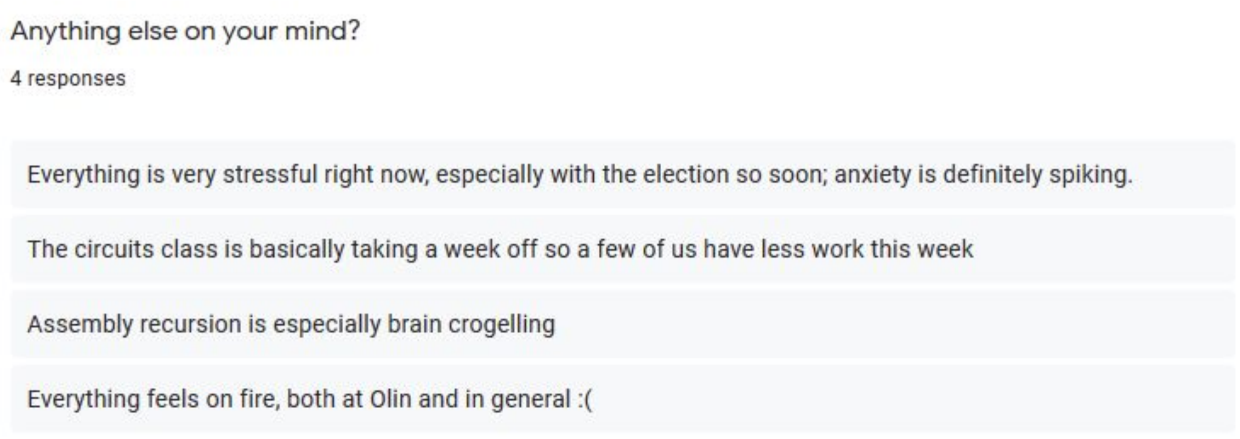
0x0F - Pipelining Hazards

ENGR 3410: Computer Architecture

Jon Tse

Fall 2020

Feedback - Top of Mind



Anything else on your mind?

4 responses

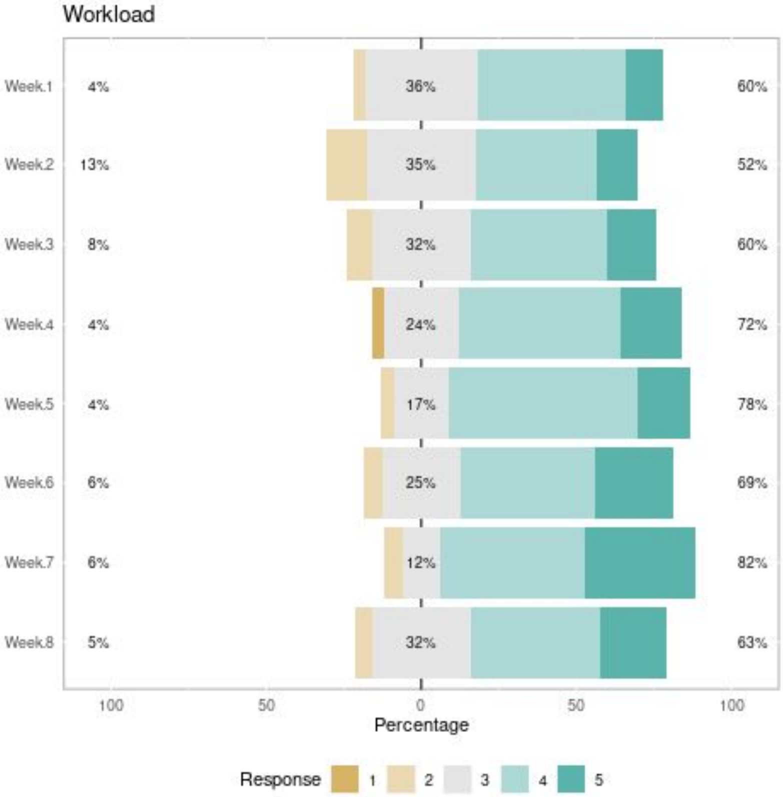
Everything is very stressful right now, especially with the election so soon; anxiety is definitely spiking.

The circuits class is basically taking a week off so a few of us have less work this week

Assembly recursion is especially brain crogelling

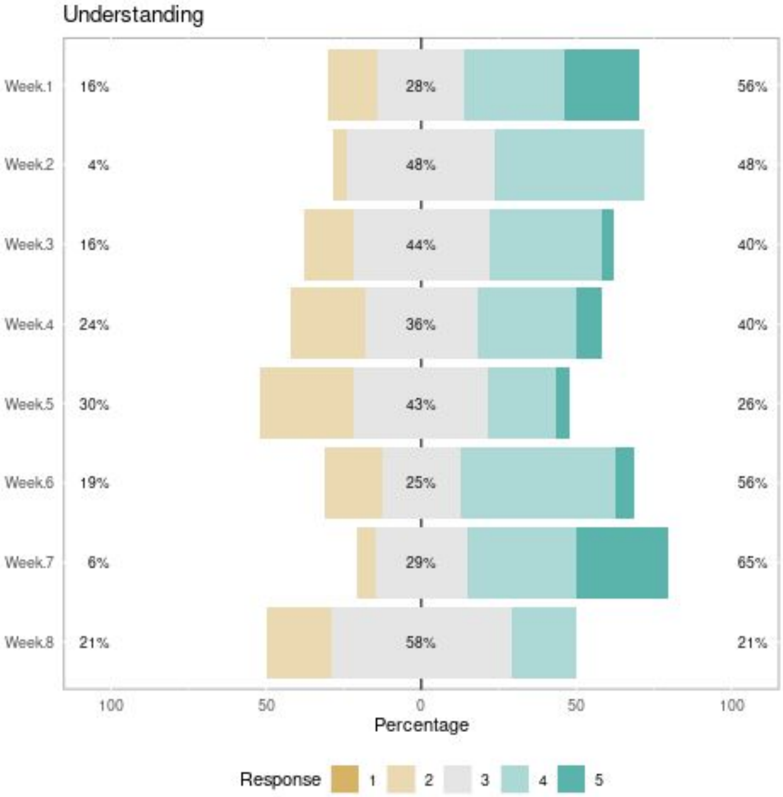
Everything feels on fire, both at Olin and in general :(

Feedback - Workload

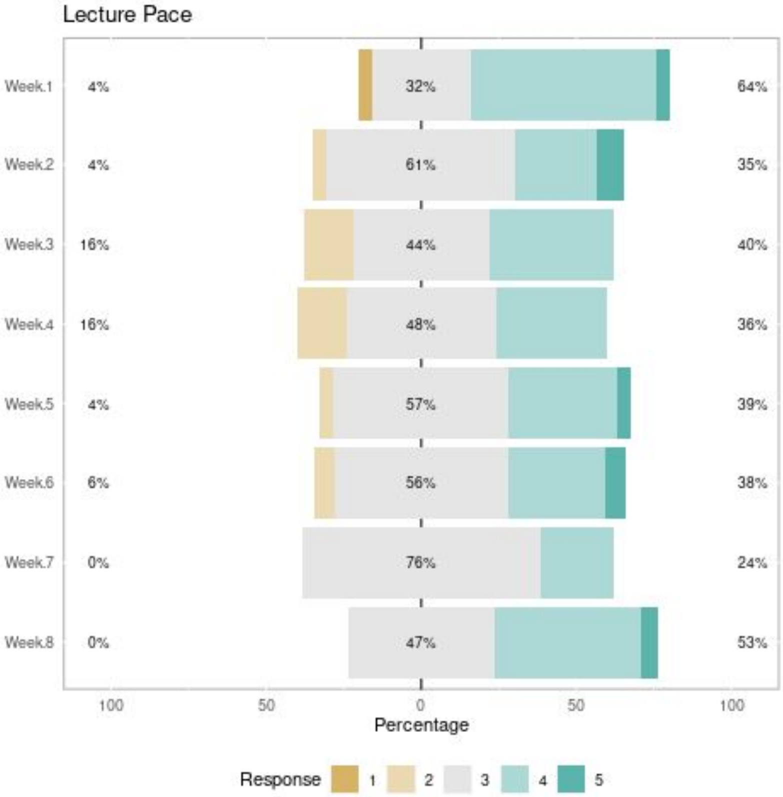


Circuits on break, likely explains reduced workload.

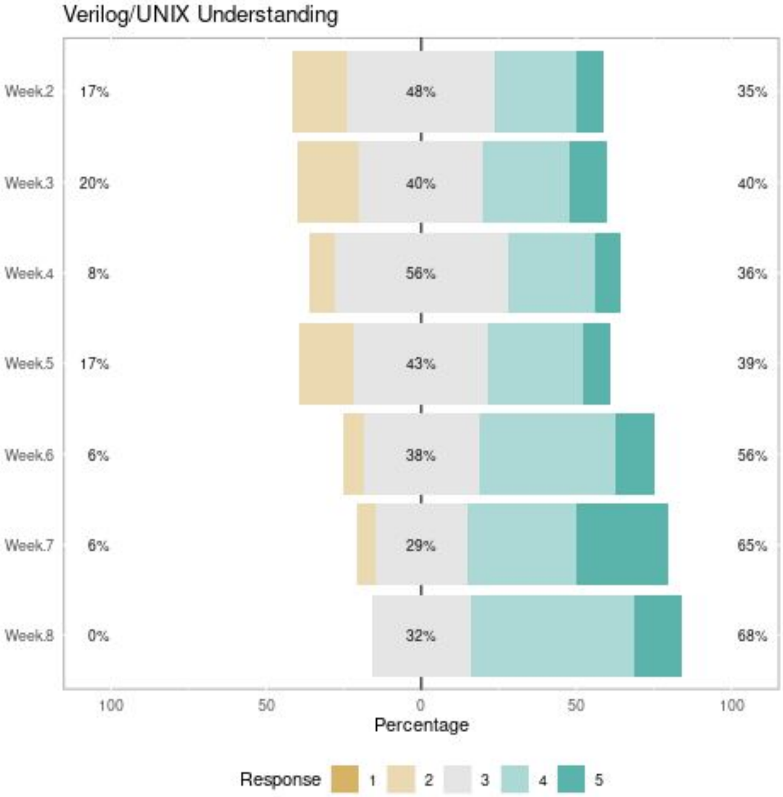
Feedback - Understanding



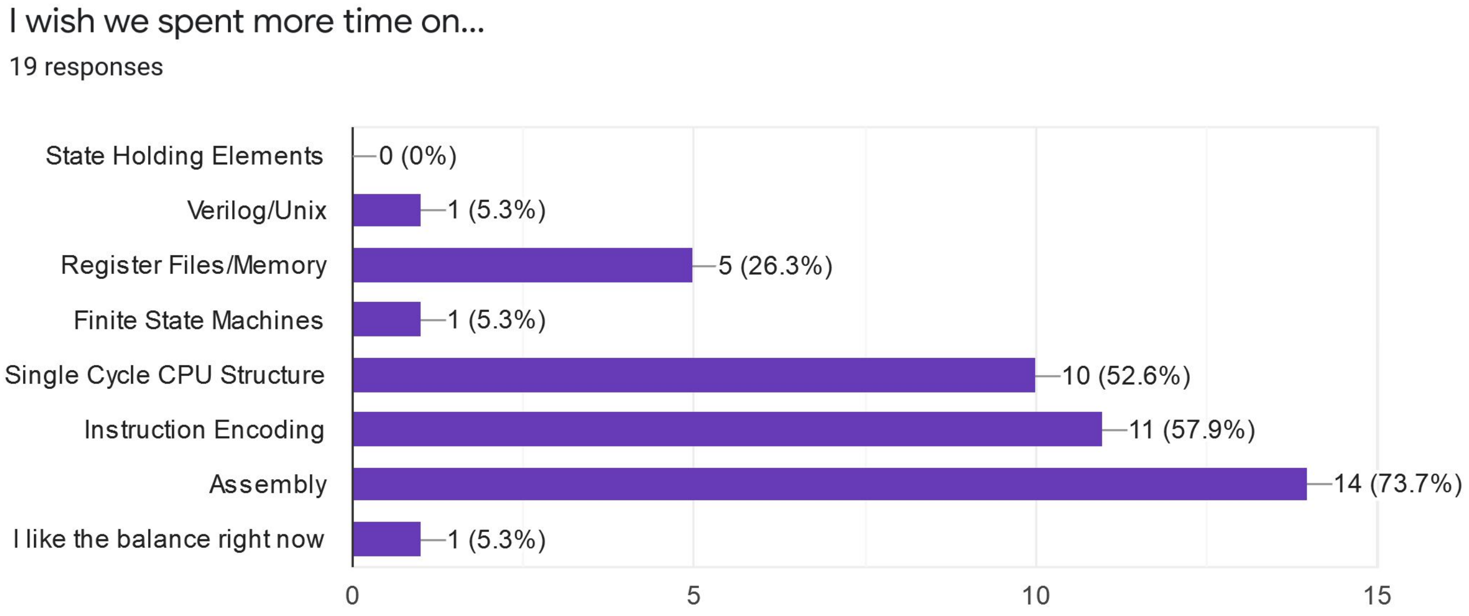
Feedback - Pace



Feedback - Tools



Feedback - More Time



I wish we spent more time on...

19 responses

Lecture on Memory coming.
More time on CPU structure.
Encoding/Assembly

Midterm

- Average Hours Spent: 6.5
- Average Transistors: 913 (max 1130, min 650)
- I waived all late penalties (only late by a few hours, not going to punish you for staying up)
- Common Pitfalls
  - States cycle when button is held
  - Bug in LUT
  - Flops connected wrong

Final Project Plan

- Please finalize groups this week

- Based on groups, I’ll publish a schedule to meet with me either 11/12 or 11/17 during class time

- Reach out now if you have concerns/questions/want ideas for the project

Today

- Discuss Hazards of Pipelining

MIPS Code Encoding Formats

- All instructions encoded in 32 bits
- Register (R-type) instructions

rd = rs op rt

1. 2.

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 09 08 07 06	05 04 03 02 01 00
OP	RS	RT	RD	SHAMT	FUNCT

(OP = 0,16-20)

- Immediate (I-type) instructions

rt = rs op imm

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
OP	RS	RT	16 bit Address/Immediate

(OP = any but 0,2,3,16-20)

- Jump (J-type) instructions

31 30 29 28 27 26	25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
OP	26 bit Address

(OP = 2,3)

PC → A → instr → DEC ≡ ctrl signals

PC+4

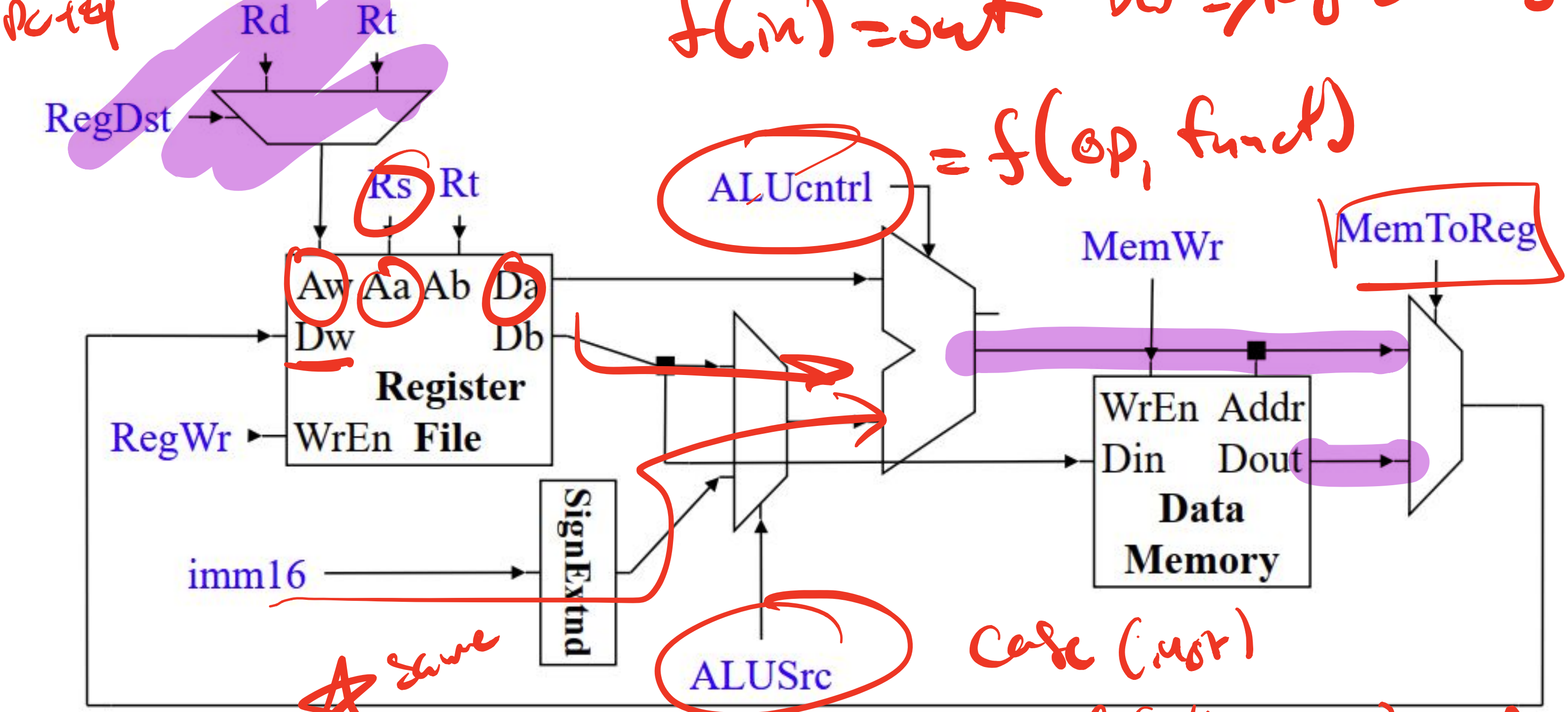
Single Cycle CPU

Da <= Reg[Aa]

Db <= Reg[Ab]

Dw => Reg[Aw]

f(in) = out



ALUcntrl = f(op, funct)

case (instr)

default : MemToReg = reg

lw : " = mem

A same

A → inst mem → B

Review

- Pipelining allows multiple instructions to be “in flight” in the data path at the same time

- **Temporal Parallelism** breaks instructions in to small tasks that run in multiple stages

- Potential Throughput Speedup = # Stages

- **Hazards** reduce these benefits
  - Can always be solved with No-Ops (we can do better)

Instructions In Flight

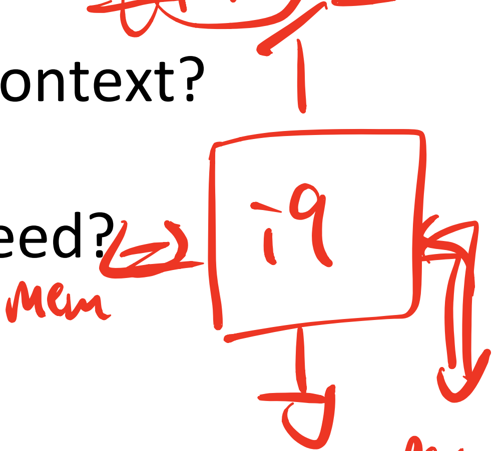
- What does “in flight” mean in this context?
- What state does each instruction need?
- Where is this state stored?

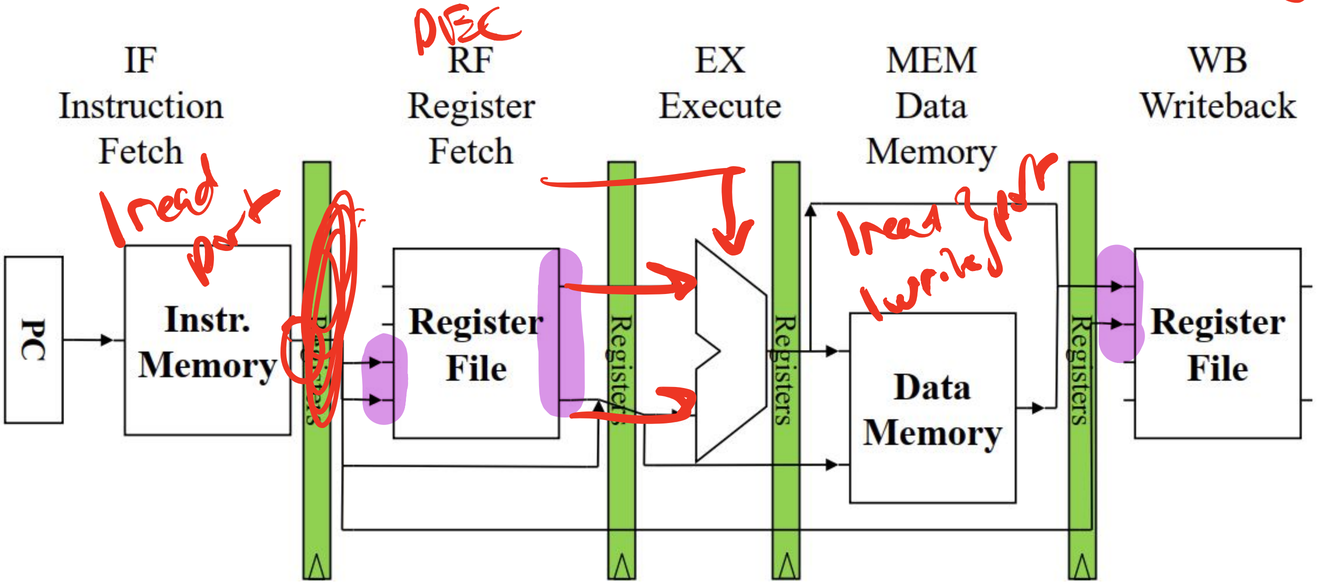
Instructions In Flight

Mem =

PCie =

- What does “in flight” mean in this context?
- What state does each instruction need?
- Where is this state stored?

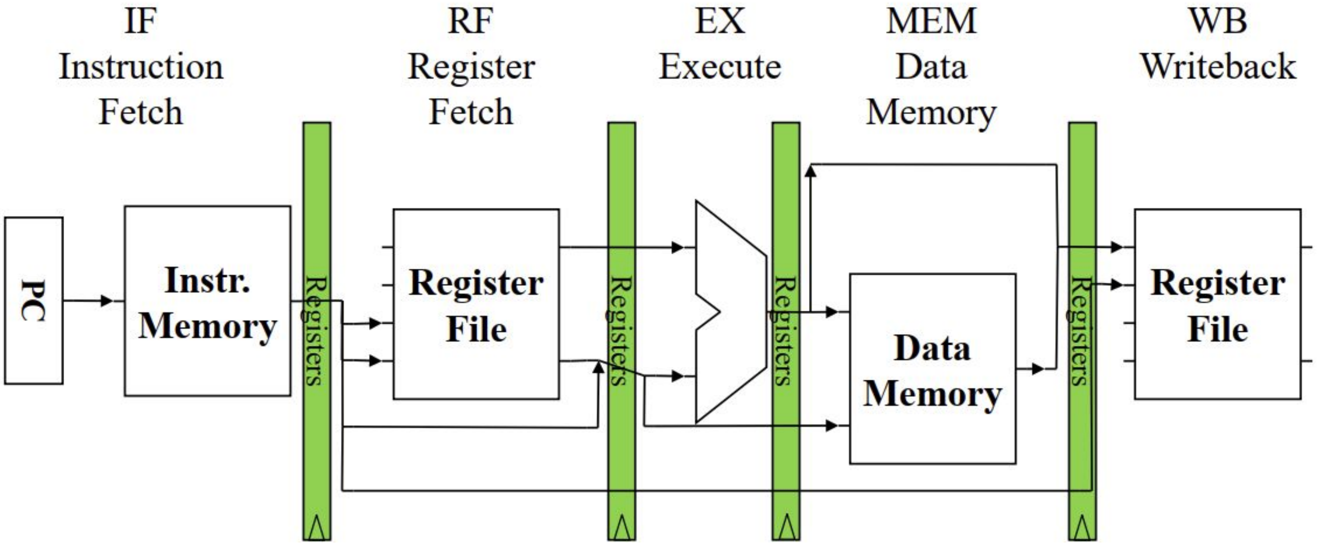




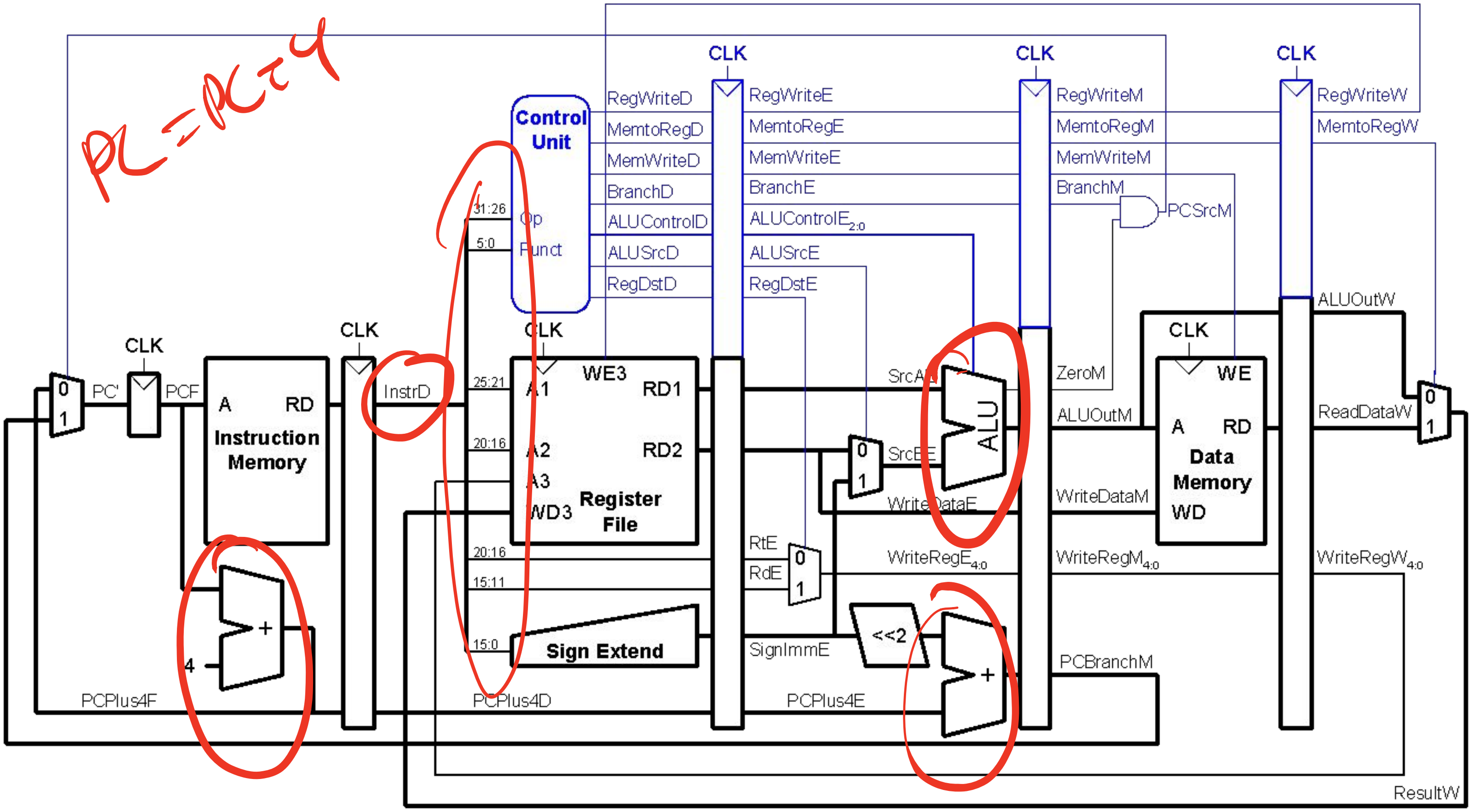
Instructions In Flight

- One instruction is in stage at a time
  - No “smearing” across stages

- Entire instruction state is in the stage’s registers



Pipelined CPU w/ Controls



PC=PC+4

Montek Singh, COMPS541

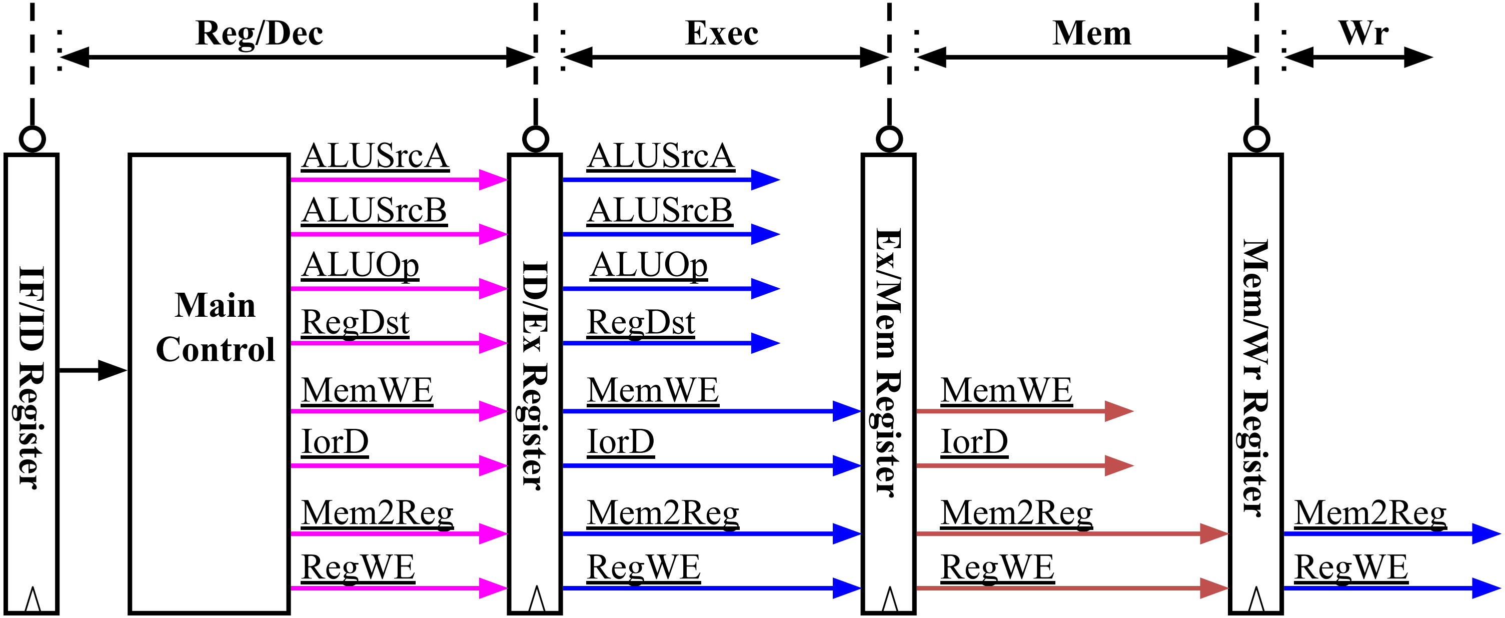
The Life and Death of State

- Control Signals are generated in the Decoder
  - Propagated until they are needed

- Data Signals are generated later
  - e.g. Reg File Reads, ALU Result

- Signals stop propagating when they are no longer needed

Pipelined Control



State Check

- Annotate control signals on the 5 stage CPU
  - Spawn Point, Usage(s), Cull Point
  - Width

	Width	IF/ID	ID/EX	EX/MEM	MEM/WB
Read Reg Addrs					
Read Reg Data A					
Read Reg Data B					
Write Reg Addr					
Write Reg Data					
ALU Cntl					
ALU Src					
RegWrite					
MemWrite					
ALU Result					
ALU Zero					

State Check

- Annotate control signals on the 5 stage CPU
  - Spawn Point, Usage(s), Cull Point
  - Width

	Width	IF/ID	ID/EX	EX/MEM	MEM/WB
Read Reg Addrs (2)	10		▨	▨	▨
Read Reg Data A	32	▨		▨	▨
Read Reg Data B	32	▨			▨
Write Reg Addr	5				
Write Reg Data	32	▨	▨	▨	
ALU Cntl	5			▨	▨
ALU Src	1			▨	▨
RegWrite	1		▨	▨	▨
MemWrite	1				▨
ALU Result	32	▨	▨		
ALU Zero	1	▨	▨		▨

Jumping and Branching

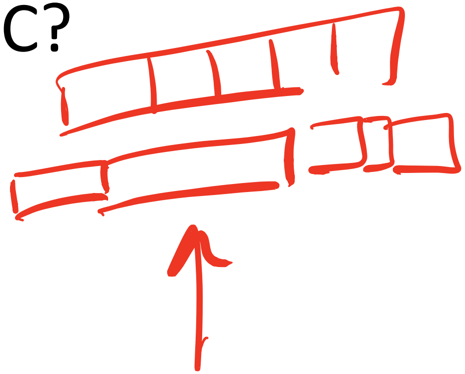
- When does Jump update PC?

- Is this ok?

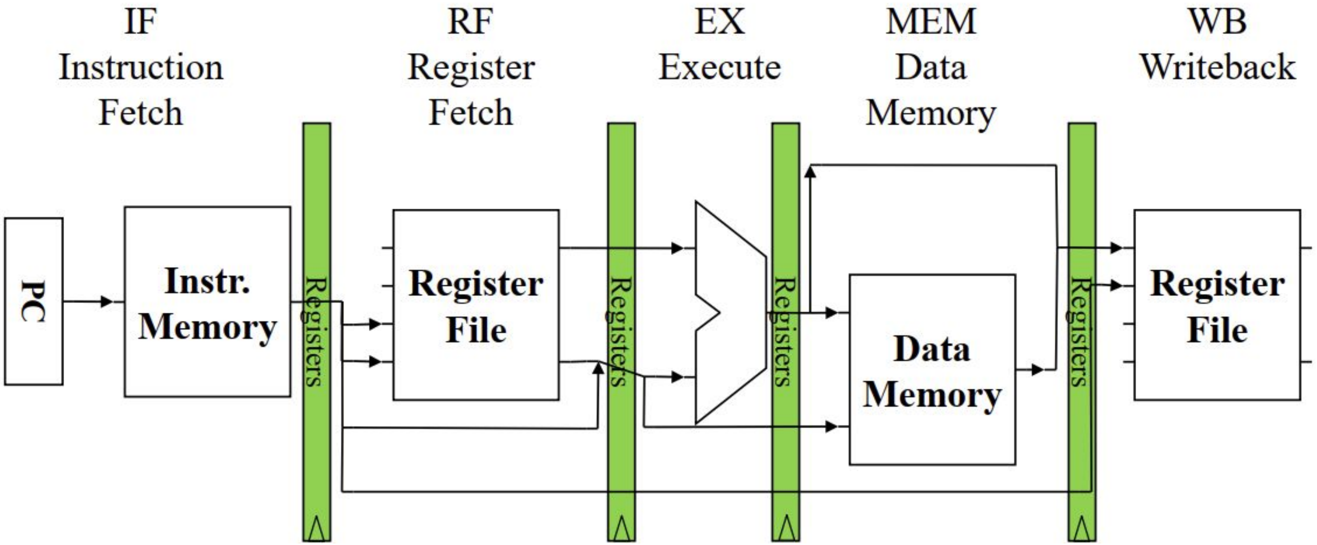
- Can we do better?

Jumping and Branching

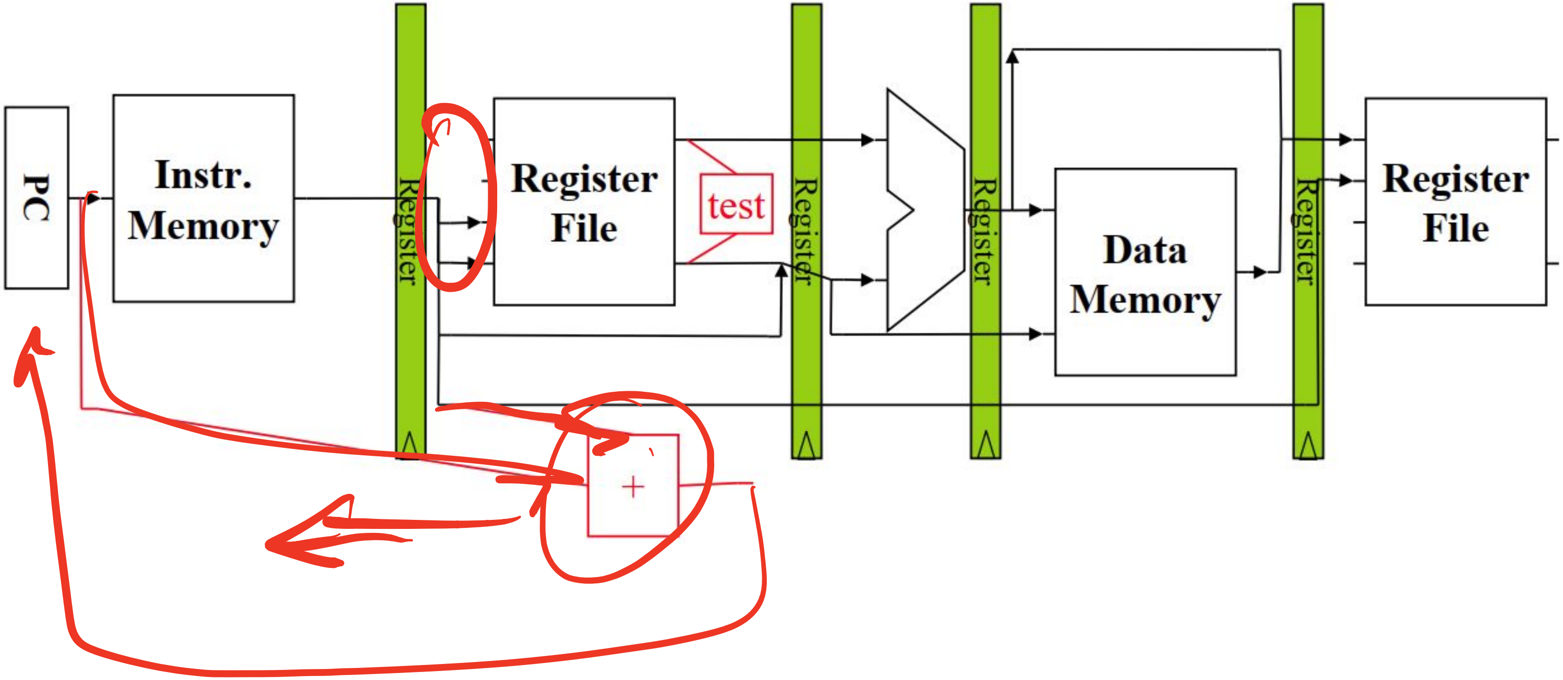
- When does Jump update PC?
  - In Instruction Decode
- Is this ok?
  - ☹
- Can we do better?
  - Push to Instruction Fetch?
- A **Control Hazard** is when the wrong instruction gets executed because IFetch Fail



Jumping and Branching



Jumping and Branching



Add hardware -> Update PC after RegFetch/Decode

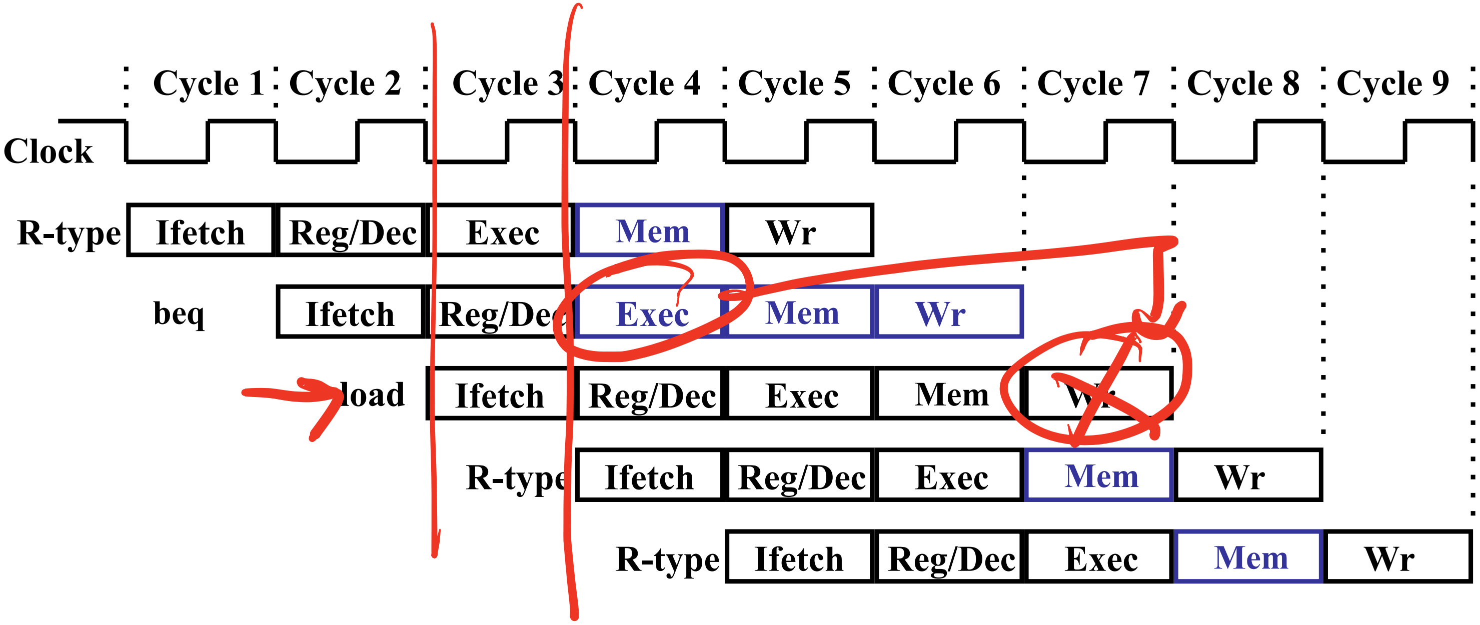
Branch is still a Hazard

R
beq
load
R
R

- PC is updated at the end of Reg/Dec

$rs == $rt

- What does this do to this sample program?

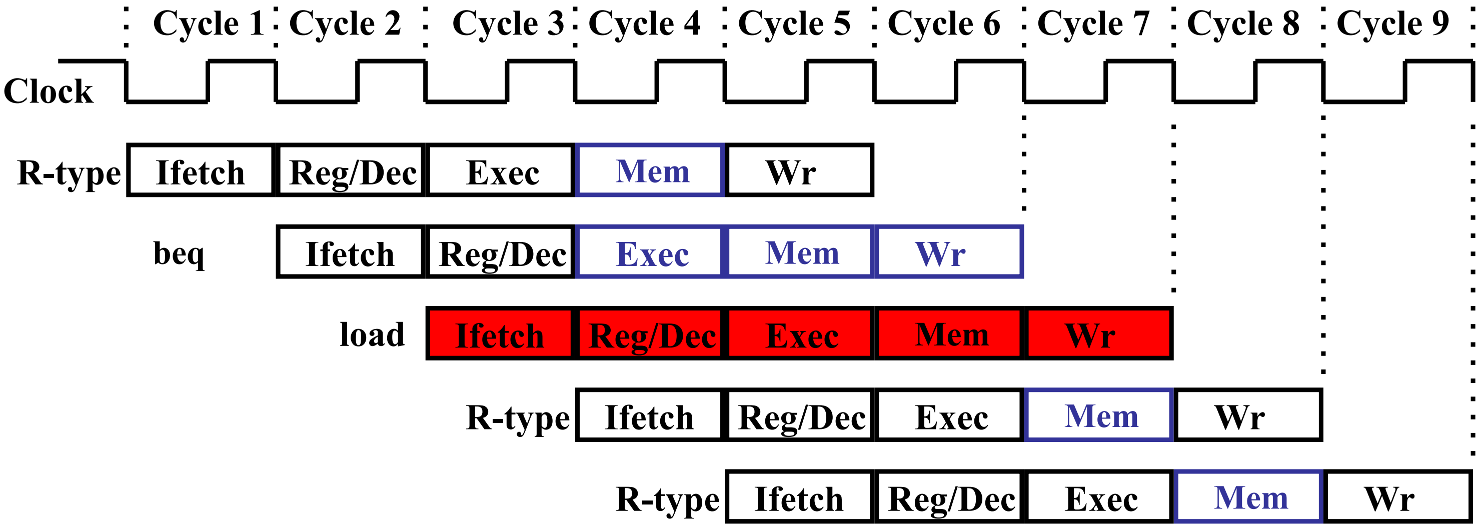


	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr				
beq		Ifetch	Reg/Dec	Exec	Mem	Wr			
load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
R-type					Ifetch	Reg/Dec	Exec	Mem	Wr

Branch is still a Hazard

- PC is updated at the end of Reg/Dec

- What does this do to this sample program?



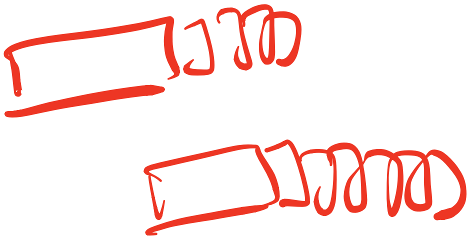
What to do?

- LW is sneaking in past the branch!!

- How can we solve this problem?

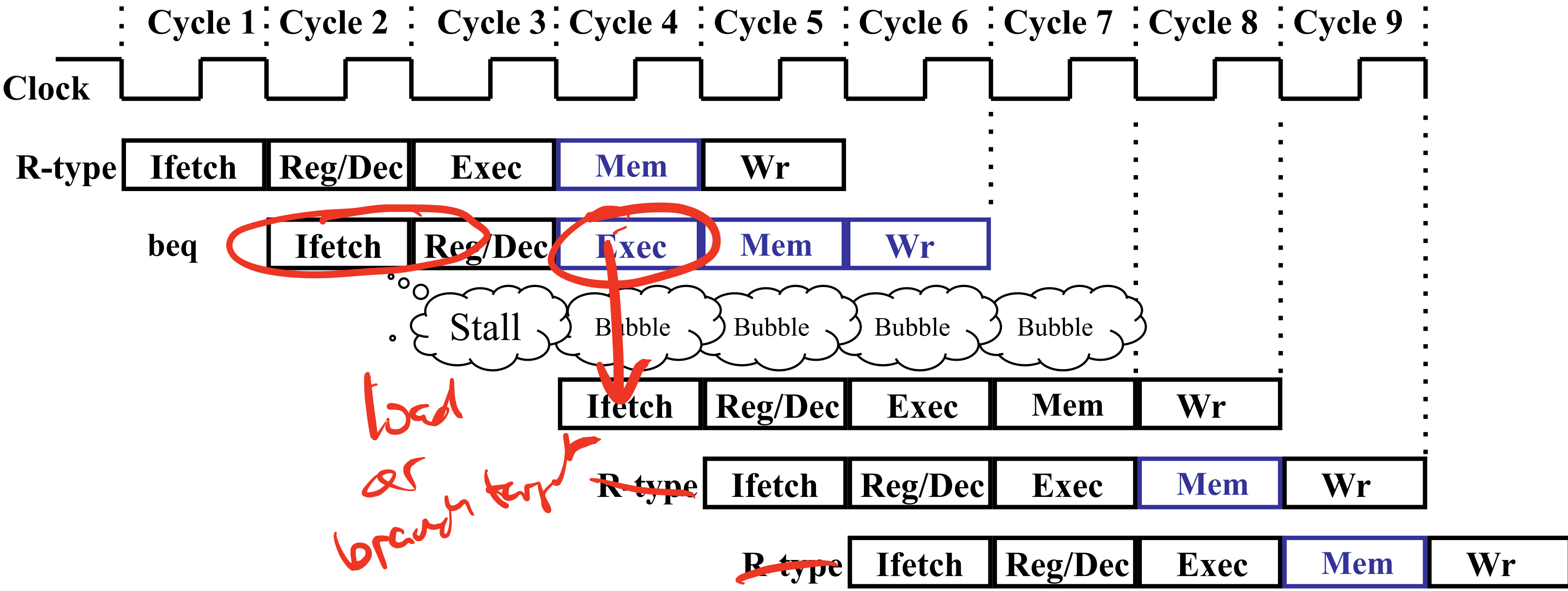
Control Hazard Solution: Go Faster

- Push the branch hardware in to the first cycle
- IF cycle takes **MUCH** longer now
- Poor Balancing
- Slows down all the other instructions!
- Common in “shallow” pipelines



Control Hazard Solution: Stall

- Delay Fetch/Decoding the next instruction
- What is the impact on performance?



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	
Clock										
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr					
beq		Ifetch	Reg/Dec	Exec	Mem	Wr				
			Stall	Bubble	Bubble	Bubble	Bubble			
load or branch target				Ifetch	Reg/Dec	Exec	Mem	Wr		
~~R-type~~					Ifetch	Reg/Dec	Exec	Mem	Wr	
~~R-type~~						Ifetch	Reg/Dec	Exec	Mem	Wr

Control Hazard Solution: Embrace It

- Re-define not as a hazard, but as a feature!
- Compiler moves an instruction in to the “Branch Delay Slot”
- Common in embedded / DSP processors
  - Total control over instruction set / compiler / etc

Control Hazard Solution: Guess&Check

- Easier to beg forgiveness than ask permission
  - Make an assumption, execute accordingly
  - If it was wrong, squash the **speculative** instructions

- Known as **Branch Prediction**

Branch Prediction

- How do we pick which way to go?

- Invent a scheme, apply it to example code
  - How many did you get right?
  - Does the nature of the code matter?
  - Does the nature of the inputs matter?

- How would this be implemented in HW?

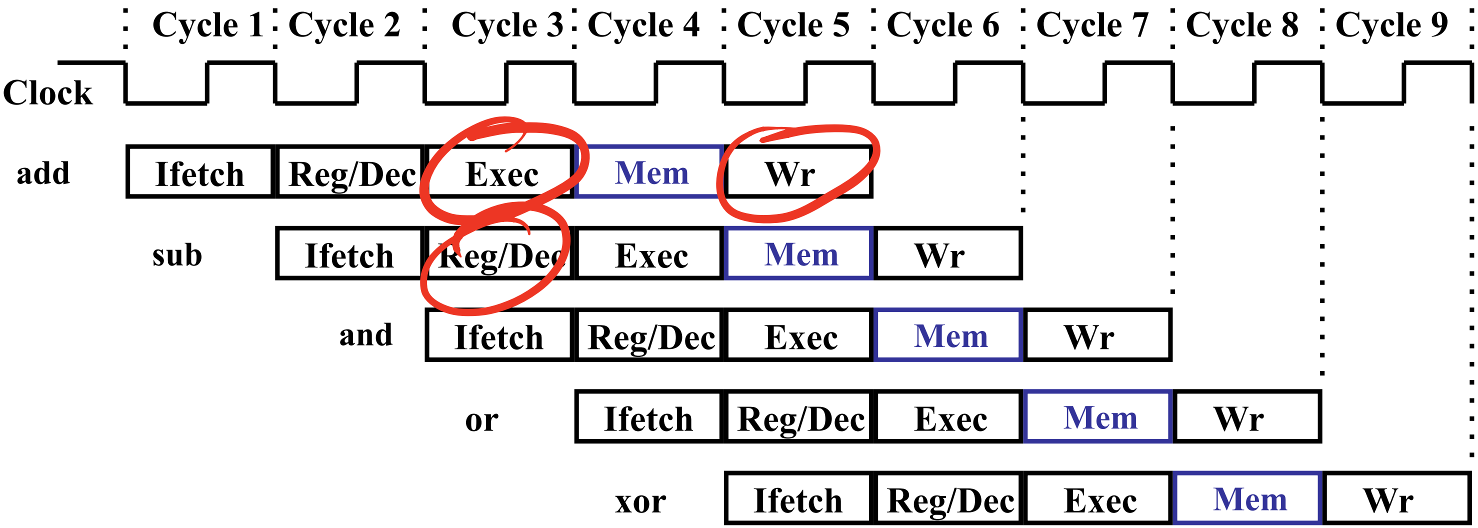
Data Hazards

- What happens with the following code?

```
add $t0, $t1, $t2
sub $t3, $t0, $t4
and $t5, $t0, $t7
or $t8, $t0, $s0
xor $s1, $t0, $s2
```

t0 = t1 + t2

t3 = t0 - t4

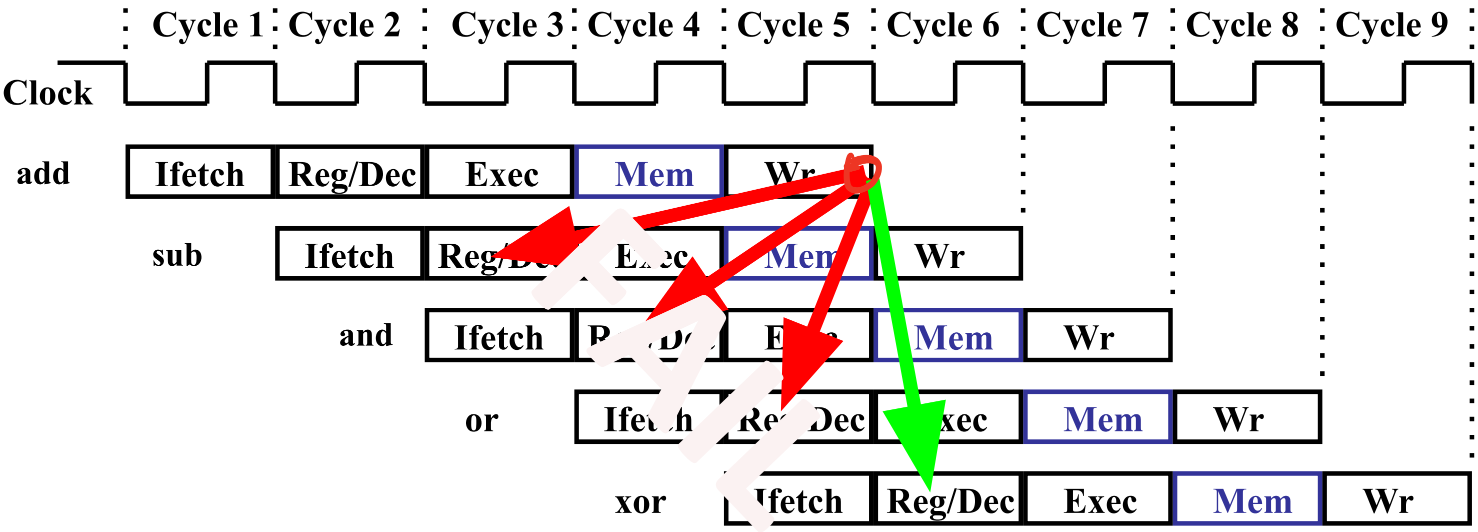


	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
add	Ifetch	Reg/Dec	Exec	Mem	Wr				
sub		Ifetch	Reg/Dec	Exec	Mem	Wr			
and			Ifetch	Reg/Dec	Exec	Mem	Wr		
or				Ifetch	Reg/Dec	Exec	Mem	Wr	
xor					Ifetch	Reg/Dec	Exec	Mem	Wr

Data Hazards

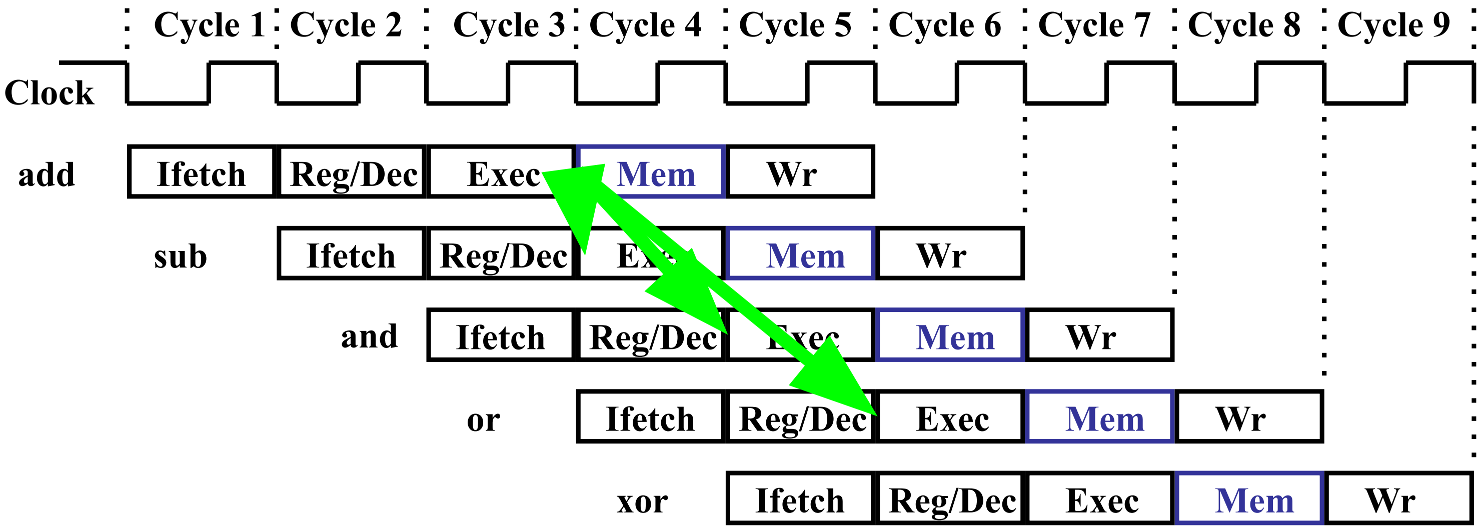
- What happens with the following code?

```
add $t0, $t1, $t2
sub $t3, $t0, $t4
and $t5, $t0, $t7
or $t8, $t0, $s0
xor $s1, $t0, $s2
```



Data Hazards: Forwarding

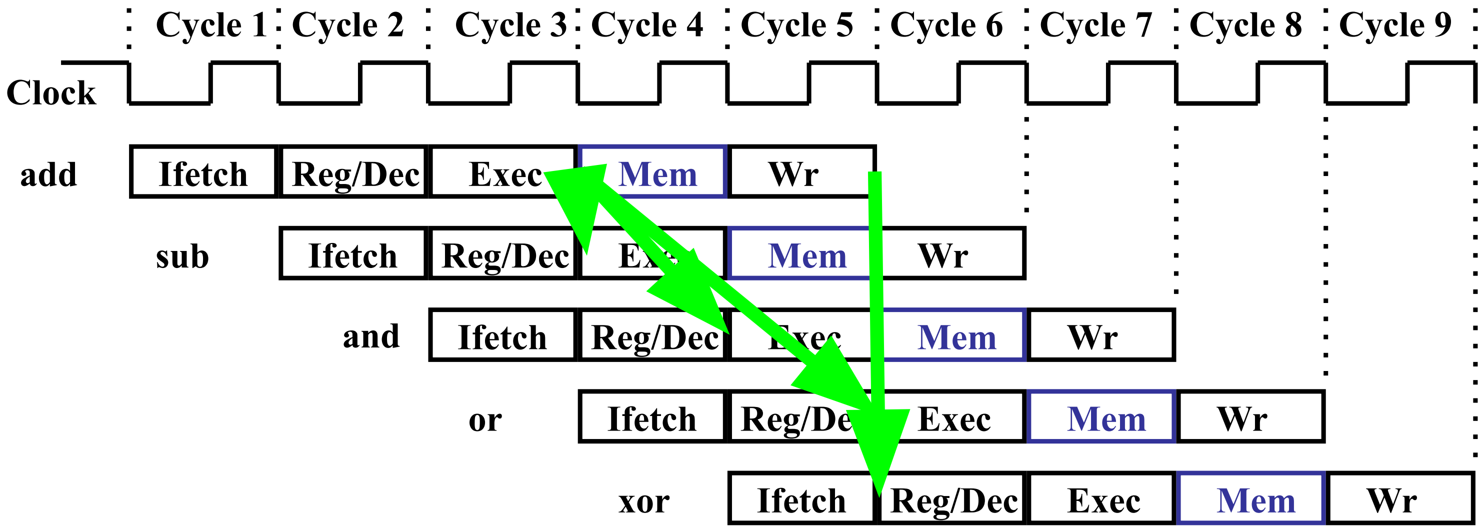
- Result isn’t committed until Writeback!
  - … but is available after Execute
  - … and really only needed in time for Execute



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
add	Ifetch	Reg/Dec	Exec	Mem	Wr				
sub		Ifetch	Reg/Dec	Exec	Mem	Wr			
and			Ifetch	Reg/Dec	Exec	Mem	Wr		
or				Ifetch	Reg/Dec	Exec	Mem	Wr	
xor					Ifetch	Reg/Dec	Exec	Mem	Wr

Data Hazards: Forwarding

- Result isn't committed until Writeback!
  - … but is available after Execute
  - … and really only needed in time for Execute



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
add	Ifetch	Reg/Dec	Exec	Mem	Wr				
sub		Ifetch	Reg/Dec	Exec	Mem	Wr			
and			Ifetch	Reg/Dec	Exec	Mem	Wr		
or				Ifetch	Reg/Dec	Exec	Mem	Wr	
xor					Ifetch	Reg/Dec	Exec	Mem	Wr

Potential Solution: Fix in Software?

- Ensure that dependent instructions do not follow each other “too closely”

- Problem: Requires detailed knowledge of microarchitecture/pipeline stages

  Not portable. Defeats the purpose of ISA. (e.g. restrictions not needed on single-cycle CPU)

Helpful Technique: Loop Unrolling

```
int num_positive(int[] sensor_values){
   for(i =0; i< length; i+=2)
      if(sensor_values[i] >0)
         numA += 1;
      if(sensor_values[i+1] >0)
         numB += 1;
   return numA+numB;
   }
```

Data Hazards: Forwarding

- Allows immediate use of a result

- Requires logic to track where things are

- Try implementing forwarding in HW
  - What new registers are needed?
  - New Muxes?
  - Control logic?
  - Can you forward with LW?