Pipelined Processors
Data and Control Hazards

Good luck on quiz 2!

Reminder: Processor Performance

- "Iron Law" of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}} \qquad \text{Perf} = \frac{1}{\text{Time}}$$

- Pipelining Goals:
  - Lower CPI: Keep CPI as close to 1 as possible
  - Lower cycle time since each pipeline stage does less work than a single cycle processor.

Reminder: Pipelining with Data Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
  - Simple, wastes cycles, higher CPI

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
  - More expensive, lower CPI
  - Still needs stalls when result is produced after EXE stage
  - Can trade off having fewer bypasses with stalling more often

Resolving Data Hazards by Stalling

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

```
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

Stall

	1	2	3	4	5	6	7	8
IF	addi	xor	sub	*sub*	*sub*	*sub*	xori	
DEC		addi	xor	*xor*	*xor*	*xor*	sub	xori
EXE			addi	**NOP**	**NOP**	**NOP**	xor	sub
MEM				addi	**NOP**	**NOP**	**NOP**	xor
WB					addi	**NOP**	**NOP**	**NOP**

x11 updated

Stalls increase CPI!

Resolving Data Hazards by Bypassing

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

```
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

- addi writes to x11 at the end of cycle 5… but the result is produced during cycle 3, at the EXE stage!

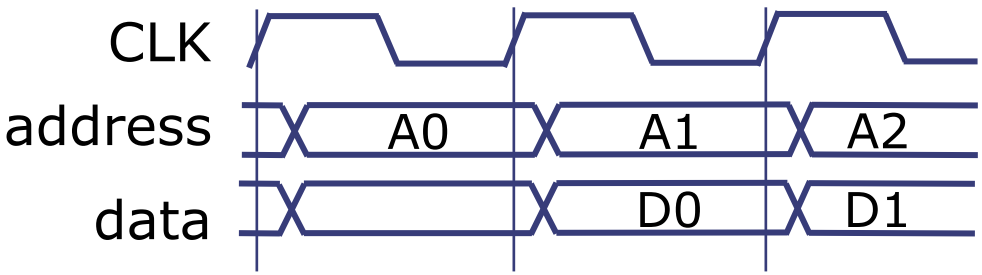
	1	2	3	4	5
IF	addi	xor	sub	xori	
DEC		addi	xor	sub	xori
EXE			addi	xor	sub
MEM				addi	xor
WB					addi

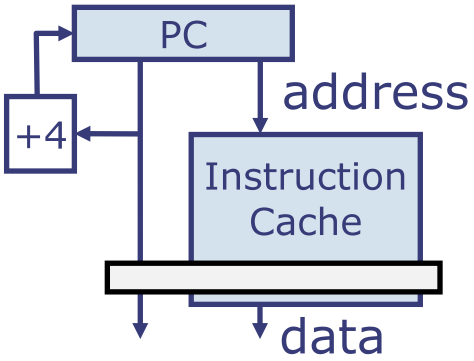
addi result computed

x11 updated

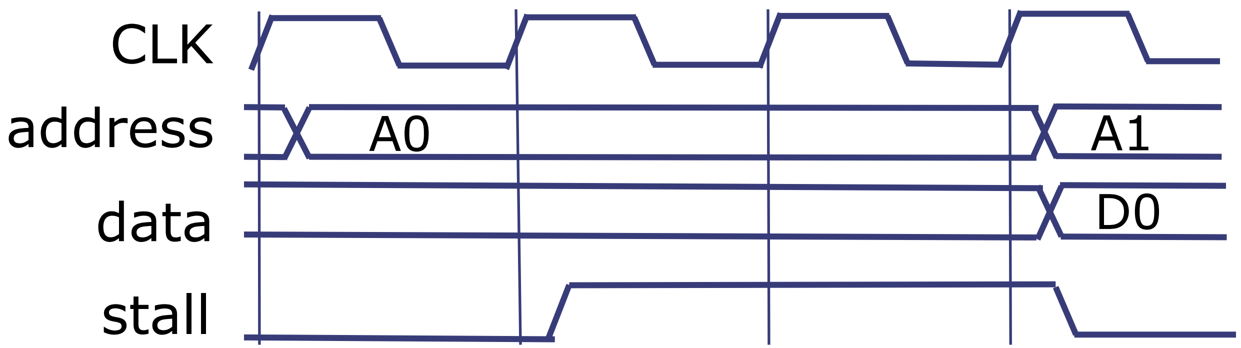
Variable Memory Response Time

- Timing of clocked read assuming cache hit (returns data by next clock cycle)





- Timing of clocked read on cache miss. The cache will produce a stall signal, telling the pipeline to wait until the memory responds.



Handling Instruction Cache Miss by Stalling

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

```
addi x9, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

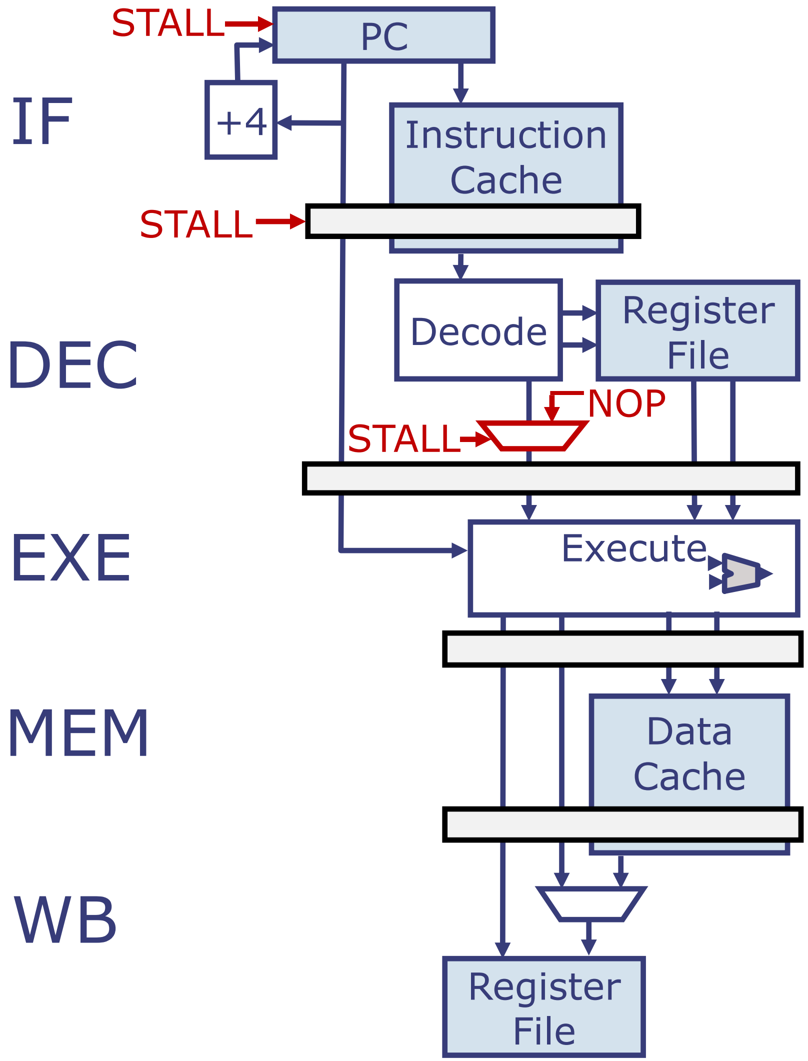
Stall

	1	2	3	4	5	6	7	8
IF	addi	*xor*	*sub*	*sub*	*sub*	sub	xori	
DEC		addi	*xor*	*xor*	xor	xor	sub	xori
EXE			addi	**NOP**	**NOP**	**NOP**	xor	sub
MEM				addi	**NOP**	**NOP**	**NOP**	xor
WB					addi	**NOP**	**NOP**	**NOP**

Instruction cache hasn't responded to fetch of xor

Instruction cache returns xor instruction
Begins fetch of sub

Stall Logic for Instruction Cache Miss



- STALL==1
  - Disables PC and IF pipeline register
  - Instruction cache keeps working to fetch data from memory
  - Injects NOP instruction into EXE stage

- Control logic sets STALL=1 if instruction cache misses (in addition to setting it when a data hazard exists.)

Resolving Data Cache Miss by Stalling

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

```
addi x9, x10, 2
lw x13, 0(x11)
sub x17, x15, x16
xori x19, x18, 0xF
ori x2, x1, 0x3
```

Stall

	1	2	3	4	5	6	7	8
IF	addi	lw	sub	xori	ori	*nextI*	*nextI*	nextI
DEC		addi	lw	sub	xori	*ori*	*ori*	ori
EXE			addi	lw	sub	*xori*	*xori*	xori
MEM				addi	lw	*sub*	*sub*	sub
WB					addi	*lw*	*lw*	lw

Data cache miss on lw request of cycle 5

lw completes

Control Hazards

Which instruction to fetch next?

- So far, we have only considered sequential execution where nextPC = PC + 4.
- Now, we will add support for branch and jump instructions.

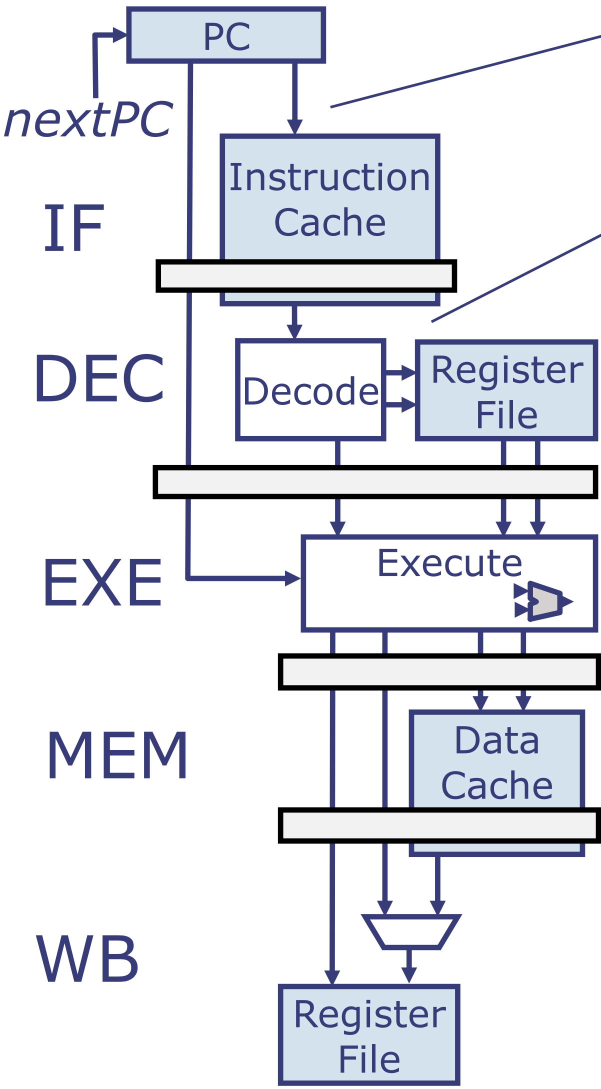
Control Hazards

- What do we need to compute nextPC?
  - We always need opcode to know how to compute nextPC

  - JAL: `nextPC = pc + immJ`
  - JALR: `nextPC = {(reg[rs1] + immI)[31:1], 1’b0}`
  - Branches: `nextPC = brFun(reg[rs1], reg[rs2])? pc + immB`
    `: pc + 4`
  - All other instructions: `nextPC = PC + 4`

- In what stage is nextPC available?
  - Depends on the pipeline and instruction type

Resolving Control Hazards



pc available in IF

opcode, imm available in DEC

operations on pc, imm, reg[rs1], reg[rs2] available in EXE

- *In what stage is nextPC available?*

JAL	EXE
JALR	EXE
Branches	EXE
Others	DEC

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

- Strategy 2: Bypass (aka Forward). Route data to the earlier pipeline stage as soon as it is calculated

- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

Resolving Control Hazards By Stalling

- Assume bne is taken in this example

```
loop:  addi x12, x11, -1
       sub x14, x15, x16
       bne x13, x0, loop
```

	1	2	3	4	5	6	7	8	9
IF	addi	**NOP**	sub	**NOP**	bne	**NOP**	**NOP**	addi	**NOP**
DEC		addi	**NOP**	sub	**NOP**	bne	**NOP**	**NOP**	addi
EXE			addi	**NOP**	sub	**NOP**	bne	**NOP**	**NOP**
MEM				addi	**NOP**	sub	**NOP**	bne	**NOP**
WB					addi	**NOP**	sub	**NOP**	bne

Opcode not known yet
nextPC unknown → Stall

Opcode = addi
nextPC = PC + 4

Opcode = bne
nextPC unknown (branch outcome in EXE) → Stall once more

CPI = 7 cycles / 3 instructions !

Might as well not pipeline…

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass (aka Forward). Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

Resolving Control Hazards with Speculation

```
loop: addi x12, x11, -1
      sub x14, x15, x16
      bne x13, x0, loop
      and x16, x17, x18
      xor x19, x20, x21
      …
```

- *What's a good guess for nextPC?* PC+4
- Assume bne is not taken in example

	1	2	3	4	5	6	7	8	9
IF	addi	sub	bne	and	xor				
DEC		addi	sub	bne	and	xor			
EXE			addi	sub	bne	and	xor		
MEM				addi	sub	bne	and	xor	
WB					addi	sub	bne	and	xor

Start fetching at PC+4 (and) but bne not resolved yet…

Guessed right, keep going

Resolving Control Hazards with Speculation

```
loop: addi x12, x11, -1
      sub x14, x15, x16
      bne x13, x0, loop
      and x16, x17, x18
      xor x19, x20, x21
      …
```

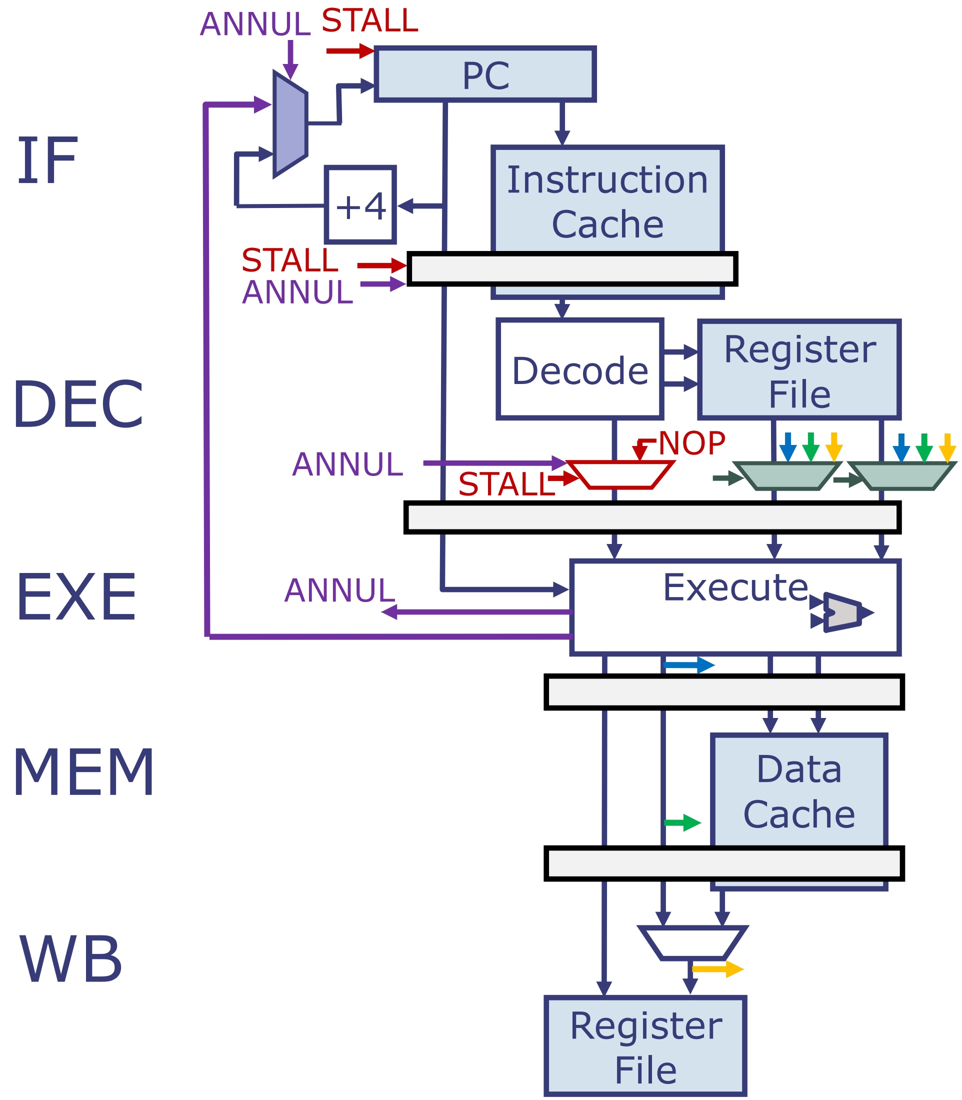
- *What's a good guess for nextPC?* PC+4
- Assume bne is taken in example

	1	2	3	4	5	6	7	8	9
IF	addi	sub	bne	and	xor	addi	sub	bne	and
DEC		addi	sub	bne	and	**NOP**	addi	sub	bne
EXE			addi	sub	bne	**NOP**	**NOP**	addi	sub
MEM				addi	sub	bne	**NOP**	**NOP**	addi
WB					addi	sub	bne	**NOP**	**NOP**

Start fetching at PC+4 (and) but bne not resolved yet …

Guessed wrong, annul and & xor and restart fetching at loop

Speculation Logic



- When EXE finds a jump or taken branch, it supplies nextPC and sets ANNUL==1
  - Writes NOPs in IF/DEC and DEC/EXE pipeline registers, annulling instructions currently in IF and DEC stages (called branch annulment)
  - Loads the branch or jump target into PC register

Interaction Between Stalling and Speculation

- Suppose that, on the same cycle,
  - EXE wants to annul DEC and IF due to a control hazard
  - DEC wants to stall due to a data hazard
- Example: Assume bne is taken

```
loop:  addi x12, x11, -1
       lw x14, 0(x15)
       bne x13, x0, loop
       and x16, x14, x18
       xor x19, x20, x21
```

	1	2	3	4	5
IF	addi	lw	bne	and	xor
DEC		addi	lw	bne	and
EXE			addi	lw	bne
MEM				addi	lw
WB					addi

bne wants to annul; and wants to stall

- *Which should take precedence, ANNUL or STALL?*

  ANNUL, because it comes from an earlier instruction

Putting It All Together

- Let's see an example with stalls, bypassing, and (mis)speculation
- Assume bne is taken once, then not taken

```
loop: addi x12, x11, -1
      lw x14, 0(x15)
      bne x13, x0, loop
      and x16, x14, x18
      xor x19, x20, x21
```

	1	2	3	4	5	6	7	8	9	10	11	12
IF	addi	lw	bne	and	xor	addi	lw	bne	and	xor	*xor*	
DEC		addi	lw	bne	and	**NOP**	addi	lw	bne	and	*and*	xor
EXE			addi	lw	bne	**NOP**	**NOP**	addi	lw	bne	**NOP**	and
MEM				addi	lw	bne	**NOP**	**NOP**	addi	lw	bne	**NOP**
WB					addi	lw	bne	**NOP**	**NOP**	addi	lw	bne

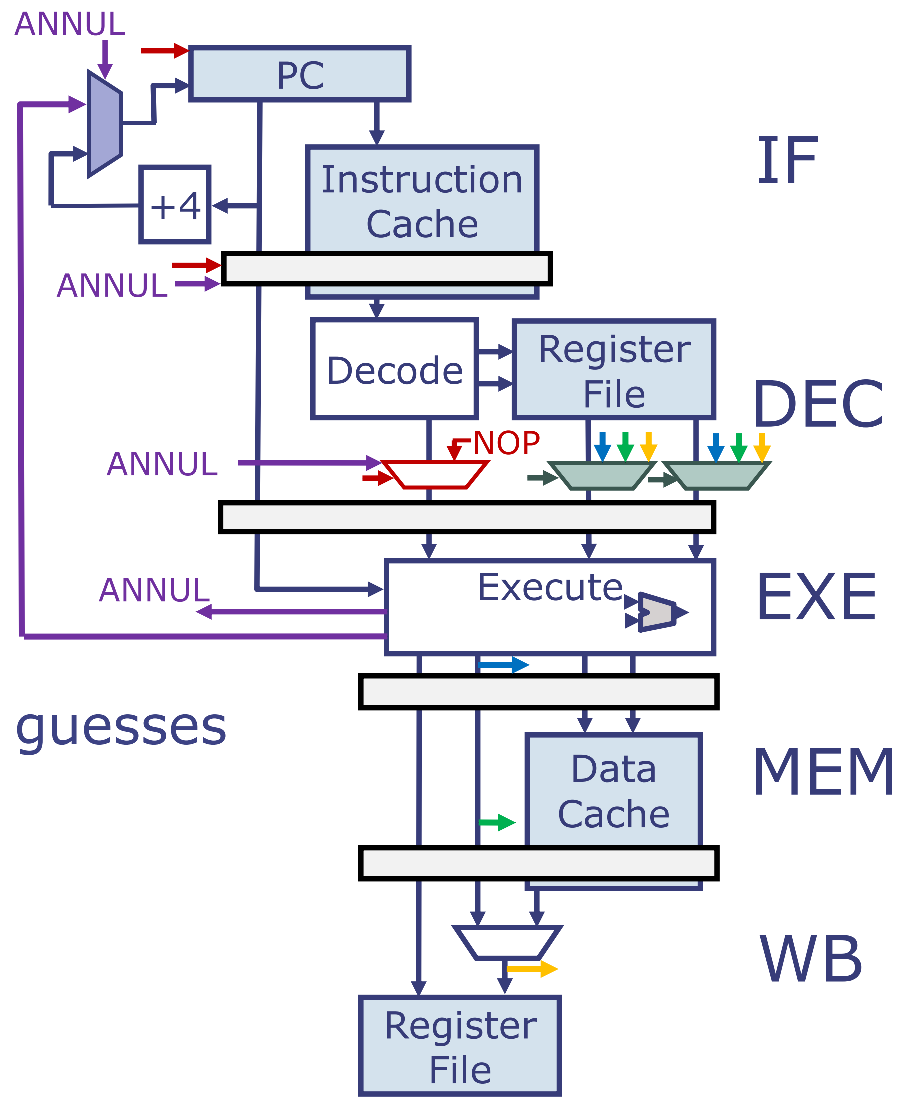
bne taken, annuls and and xor

and stalls on x14

lw value bypassed

Summary

- Stalling can address all pipeline hazards
  - Simple, but hurts CPI
- Bypassing improves CPI on data hazards
- Speculation improves CPI on control hazards
  - Speculation works only when it's easy to make good guesses



Thank you!

Next lecture: Synchronization