

# Pipelined Processors

## Data and Control Hazards

Quiz 2 on 11/9

Let us know about conflicts

# Reminder: Processor Performance

---

- “Iron Law” of performance:

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

$$\text{Perf} = \frac{1}{\text{Time}}$$

- 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	<b>NOP</b>	<b>NOP</b>	<b>NOP</b>	xor	sub
MEM				addi	<b>NOP</b>	<b>NOP</b>	<b>NOP</b>	xor
WB					addi	<b>NOP</b>	<b>NOP</b>	<b>NOP</b>


 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

# Load-To-Use Stalls

- Bypassing cannot eliminate load delays because their data is not available until the WB stage

- Bypassing from WB still saves a cycle:

*lw* x11, 0(x10)

*xor* x13, x11, x12

*sub* x17, x15, x16

*xori* x19, x18, 0xF

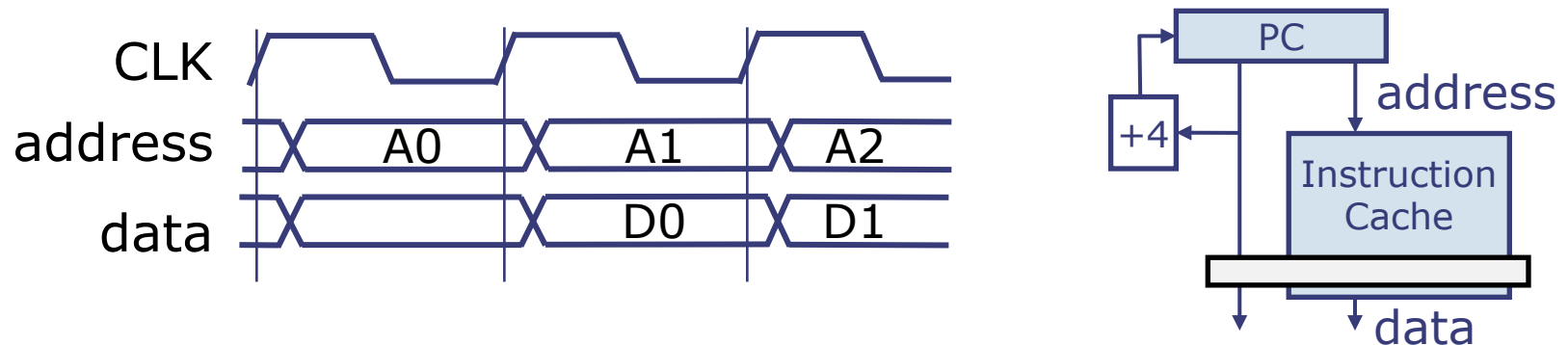
	1	2	3	4	5	6	7	8
IF	<i>lw</i>	<i>xor</i>	<i>sub</i>	<i>sub</i>	<i>sub</i>	<i>xori</i>		
DEC		<i>lw</i>	<i>xor</i>	<i>xor</i>	<i>xor</i>	<i>sub</i>	<i>xori</i>	
EXE			<i>lw</i>	<b>NOP</b>	<b>NOP</b>	<i>xor</i>	<i>sub</i>	<i>xori</i>
MEM				<i>lw</i>	<b>NOP</b>	<b>NOP</b>	<i>xor</i>	<i>sub</i>
WB					<i>lw</i>	<b>NOP</b>	<b>NOP</b>	<i>xor</i>

*lw* data available

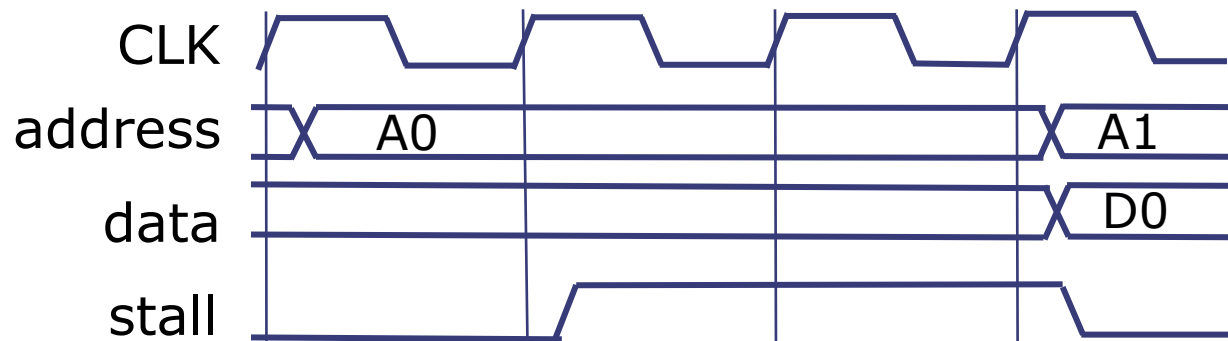
x11 updated

# 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

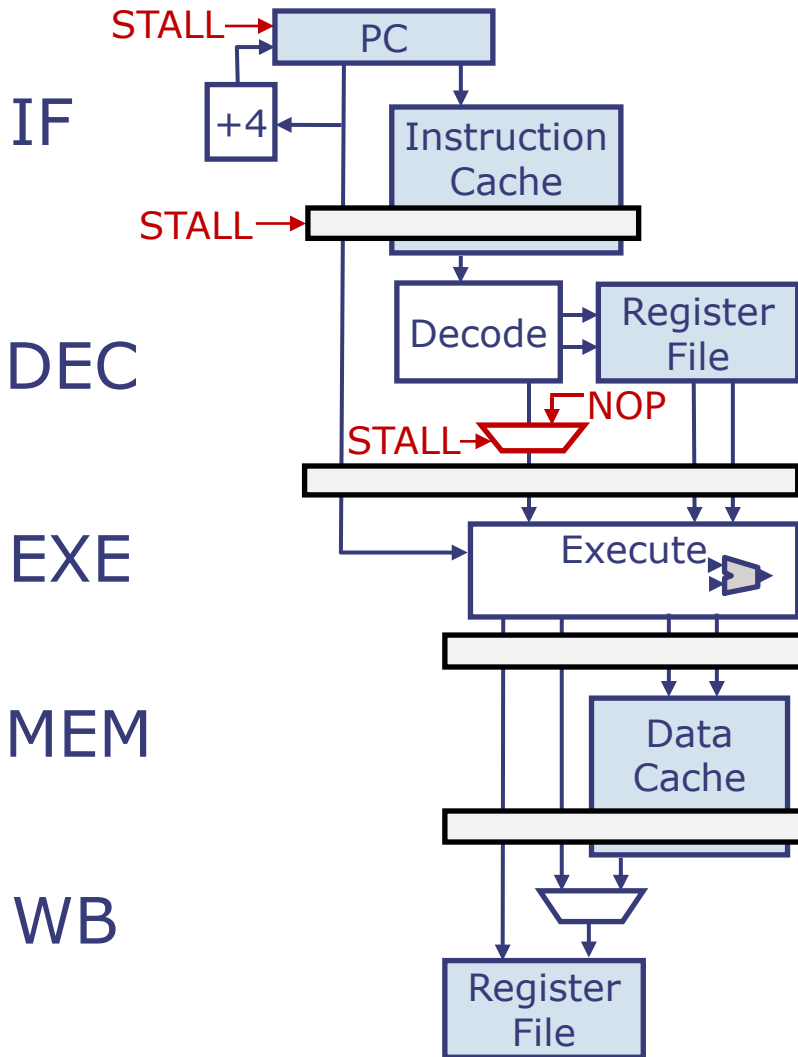
	1	2	3	4	5	6	7	8
IF	addi	xor	sub	sub	sub	sub	xori	
DEC		addi	xor	xor	xor	xor	sub	xori
EXE			addi	<b>NOP</b>	<b>NOP</b>	<b>NOP</b>	xor	sub
MEM				addi	<b>NOP</b>	<b>NOP</b>	<b>NOP</b>	xor
WB					addi	<b>NOP</b>	<b>NOP</b>	<b>NOP</b>

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
    
```

	1	2	3	4	5	6	7	8
IF	addi	lw	sub	xori	ori	<i>nextI</i>	<i>nextI</i>	<i>nextI</i>
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

Stall

Data cache miss on  
lw request of cycle 5

lw completes

# Control Hazards

# Which instruction to fetch next?

---

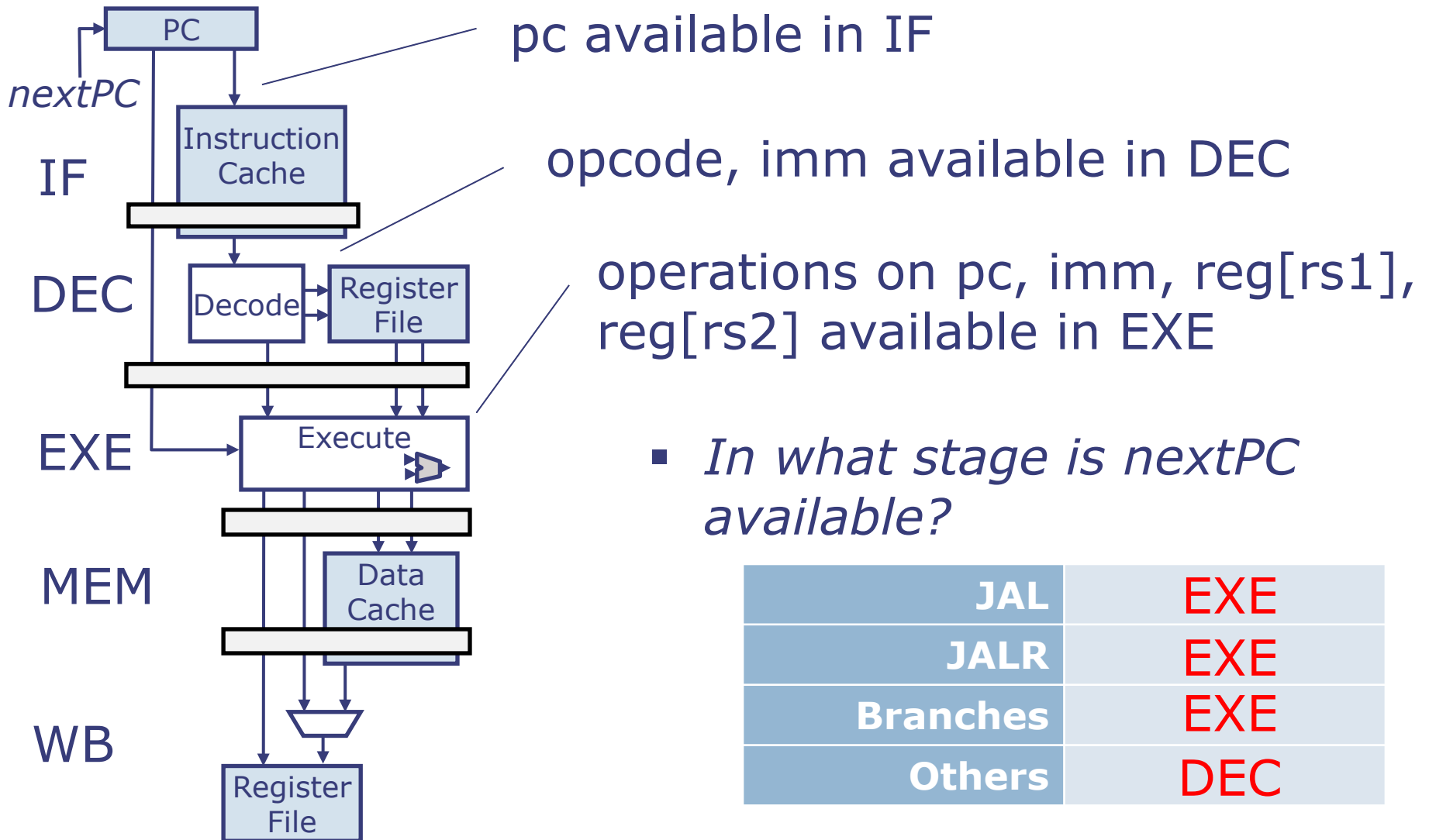
- So far, we have only considered sequential execution where  $\text{nextPC} = \text{PC} + 4$ .
- Now, we will add support for branch and jump instructions.

# Control Hazards

---

- What do we need to compute nextPC?
  - We always need **opcode** to know how to compute nextPC
  - JAL:  $\text{nextPC} = \text{pc} + \text{immJ}$
  - JALR:  $\text{nextPC} = \{(\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\}$
  - Branches:  $\text{nextPC} = \text{brFun}(\text{reg}[\text{rs1}], \text{reg}[\text{rs2}])? \text{pc} + \text{immB} : \text{pc} + 4$
  - All other instructions:  $\text{nextPC} = \text{PC} + 4$
- In what stage is nextPC available?
  - Depends on the pipeline and instruction type

# Resolving Control Hazards



# 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	<b>NOP</b>	sub	<b>NOP</b>	bne	<b>NOP</b>	<b>NOP</b>	addi	<b>NOP</b>
DEC		addi	<b>NOP</b>	sub	<b>NOP</b>	bne	<b>NOP</b>	<b>NOP</b>	addi
EXE			addi	<b>NOP</b>	sub	<b>NOP</b>	bne	<b>NOP</b>	<b>NOP</b>
MEM				addi	<b>NOP</b>	sub	<b>NOP</b>	bne	<b>NOP</b>
WB					addi	<b>NOP</b>	sub	<b>NOP</b>	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

- What's a good guess for nextPC? **PC+4**

```

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

- 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...

Gussed right, keep going

# Resolving Control Hazards with Speculation

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

```

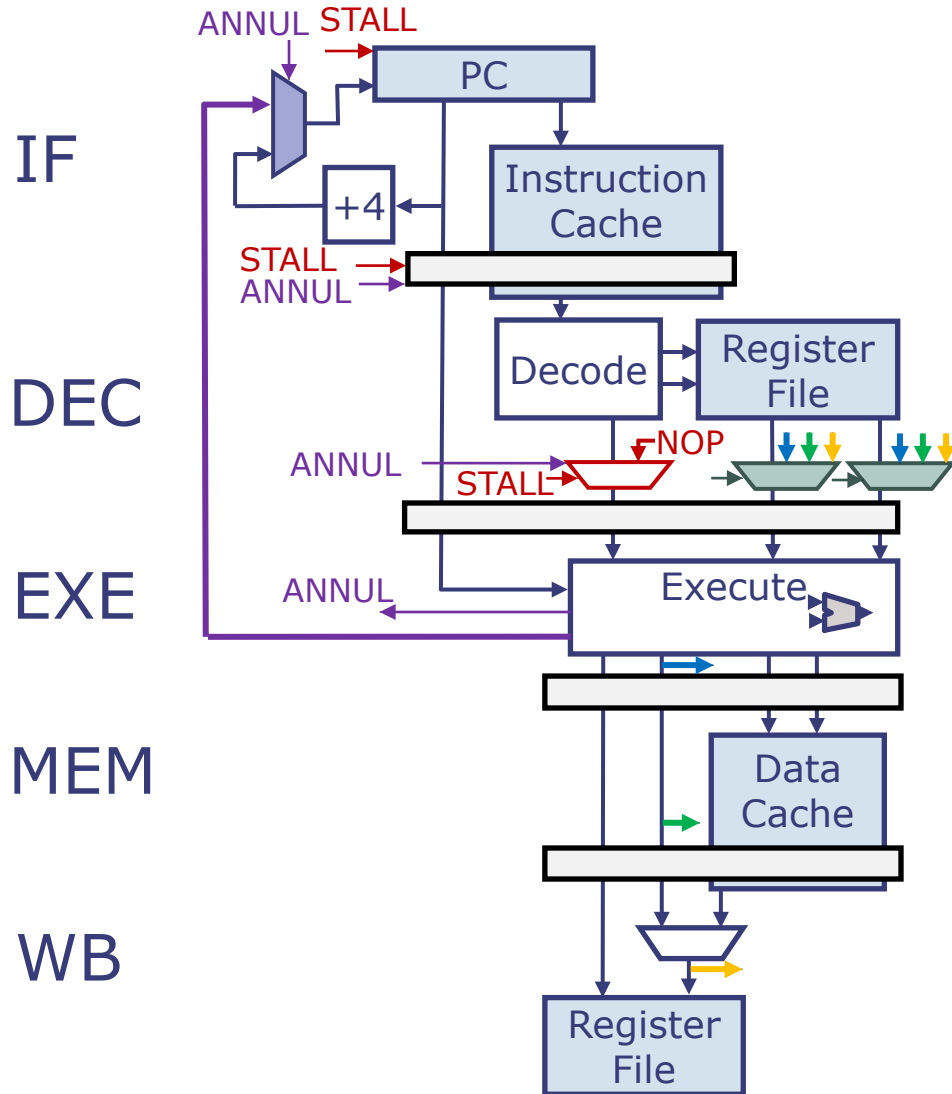
loop:  addi x12, x11, -1
       sub x14, x15, x16
       bne x13, x0, loop
       and x16, x17, x18
       xor x19, x20, x21
       ...
    
```

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

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	<b>NOP</b>	addi	lw	bne	and	and	xor
EXE			addi	lw	bne	<b>NOP</b>	<b>NOP</b>	addi	lw	bne	<b>NOP</b>	and
MEM				addi	lw	bne	<b>NOP</b>	<b>NOP</b>	addi	lw	bne	<b>NOP</b>
WB					addi	lw	bne	<b>NOP</b>	<b>NOP</b>	addi	lw	bne

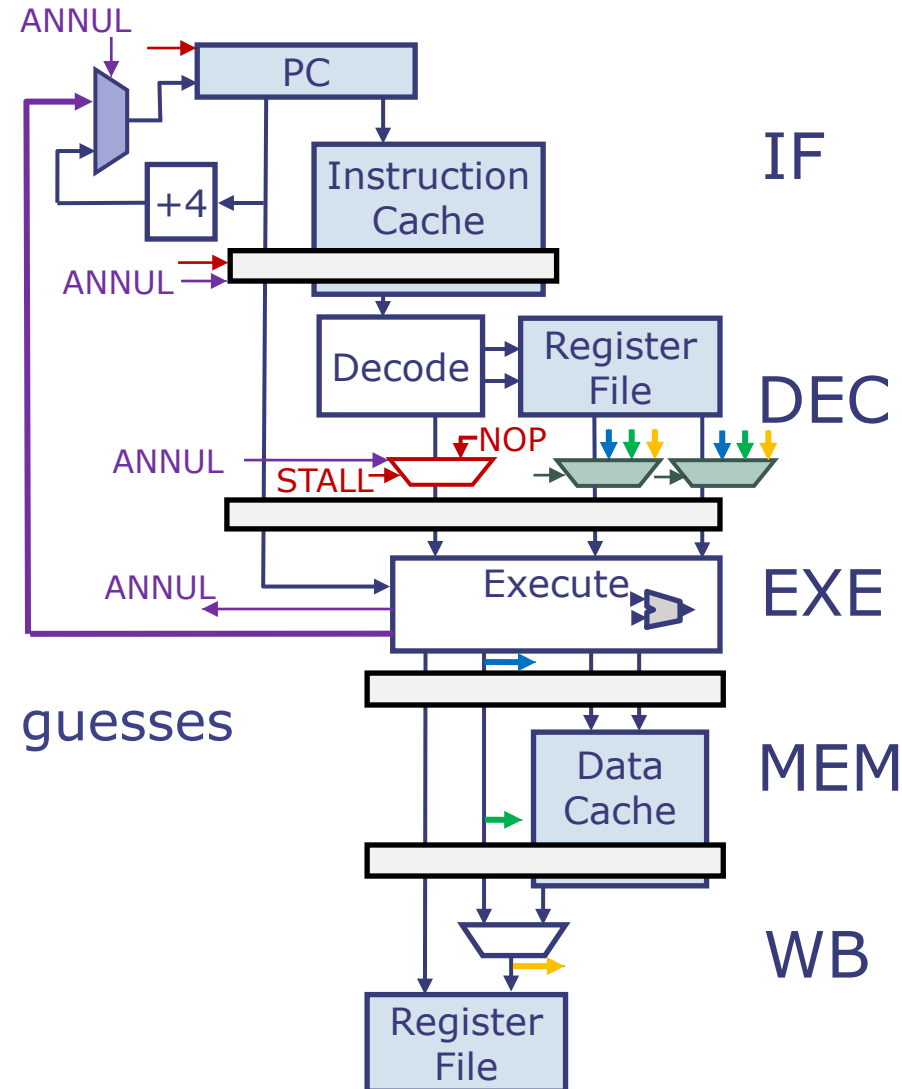
`bne` taken, annuls `and` and `xor`

`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: Operating Systems*