Pipelined Processors

Processor Performance

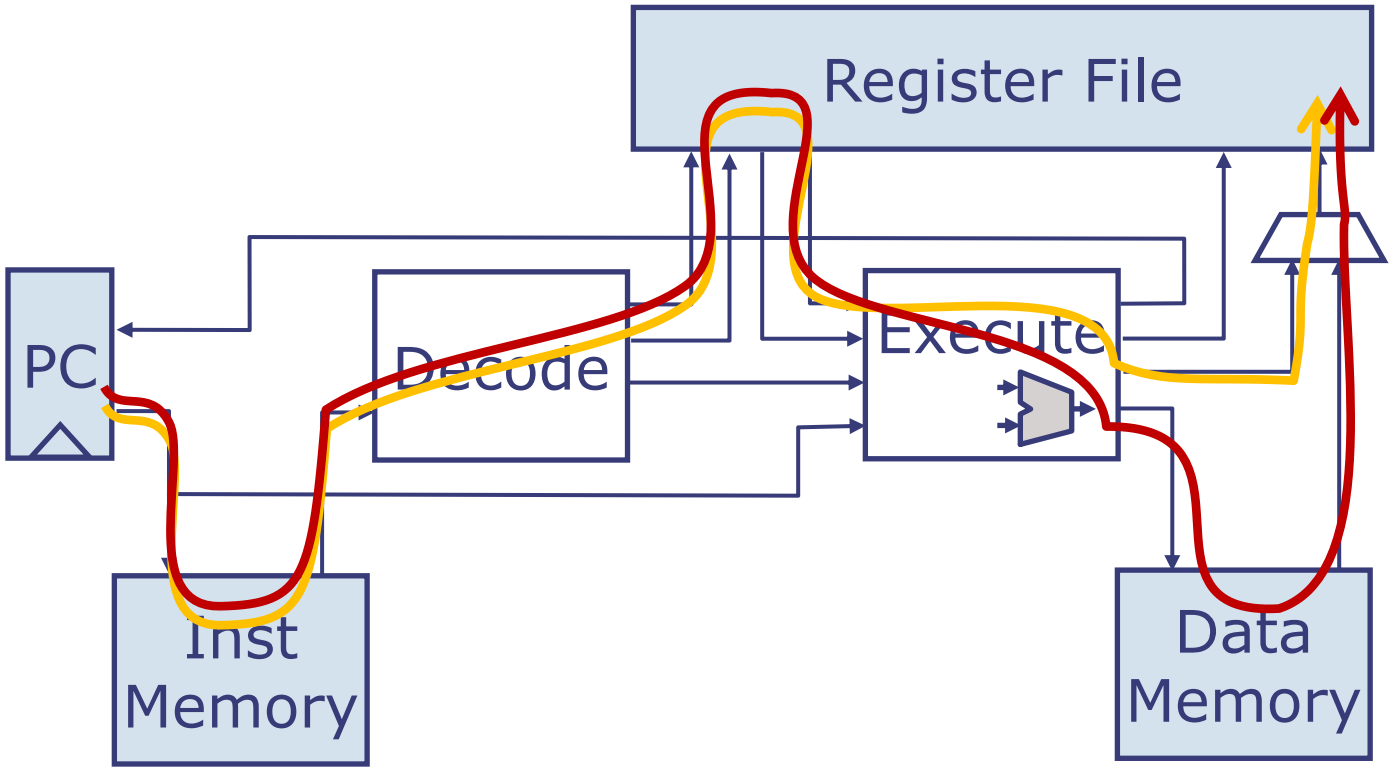
- "Iron Law" of performance:

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

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

- Options to reduce execution time:
  - Executed instructions ↓ (work/instruction ↑)
  - Cycles per instruction (CPI) ↓
  - Cycle time ↓ (frequency ↑)

Single-Cycle Processor Performance

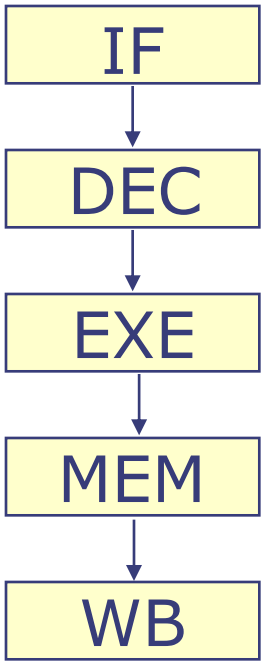


- CPI = 1
- t_{CLK} = Longest path for any instruction

$$t_{CLK} \approx t_{IMEM} + t_{DEC} + t_{RF} + t_{EXE} + t_{DMEM} + t_{WB}$$ *Slow!*

Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce t_{CLK}
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI ≈ 1.0
- We'll study the classic 5-stage pipeline:



Instruction Fetch stage: Maintains PC, fetches instruction and passes it to

Decode & Read Registers stage: Decodes instruction and reads source operands from register file, passes them to

Execute stage: Performs indicated operation in ALU, passes result to

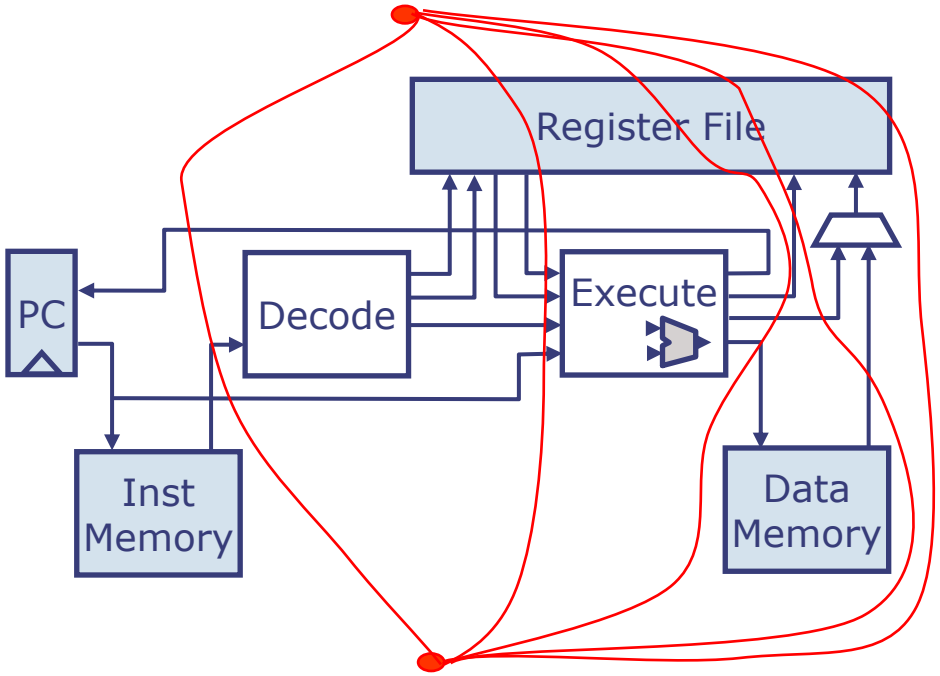
Memory stage: If it's a load, use input as the address, pass read data (or ALU result if not a load) to

Write-Back stage: writes result back into register file.

$$t_{CLK} = \max\{t_{IF}, t_{DEC}, t_{EXE}, t_{MEM}, t_{WB}\}$$

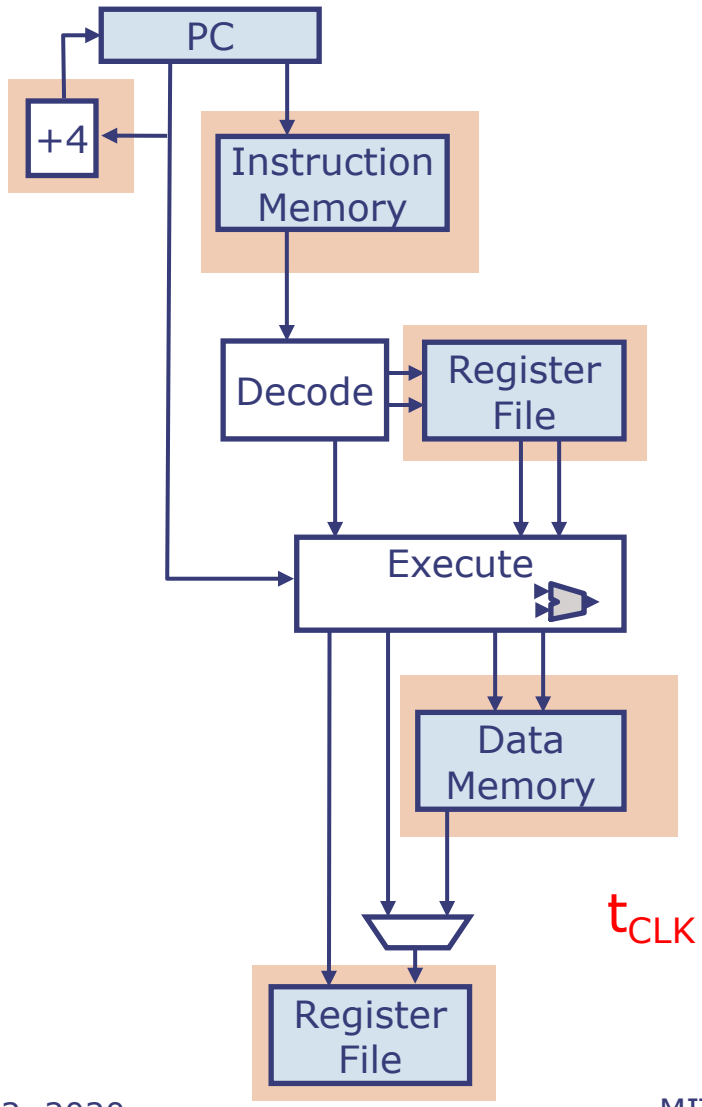
Why isn't this a 20-minute lecture?

We know how to pipeline combinational circuits, what's the big deal?



- Processor has state: PC, Register file, Memories
- There are loops we cannot break!
  - To compute the next PC
  - To write result into the register file
- Can't produce a well-formed pipeline with what we know so far…

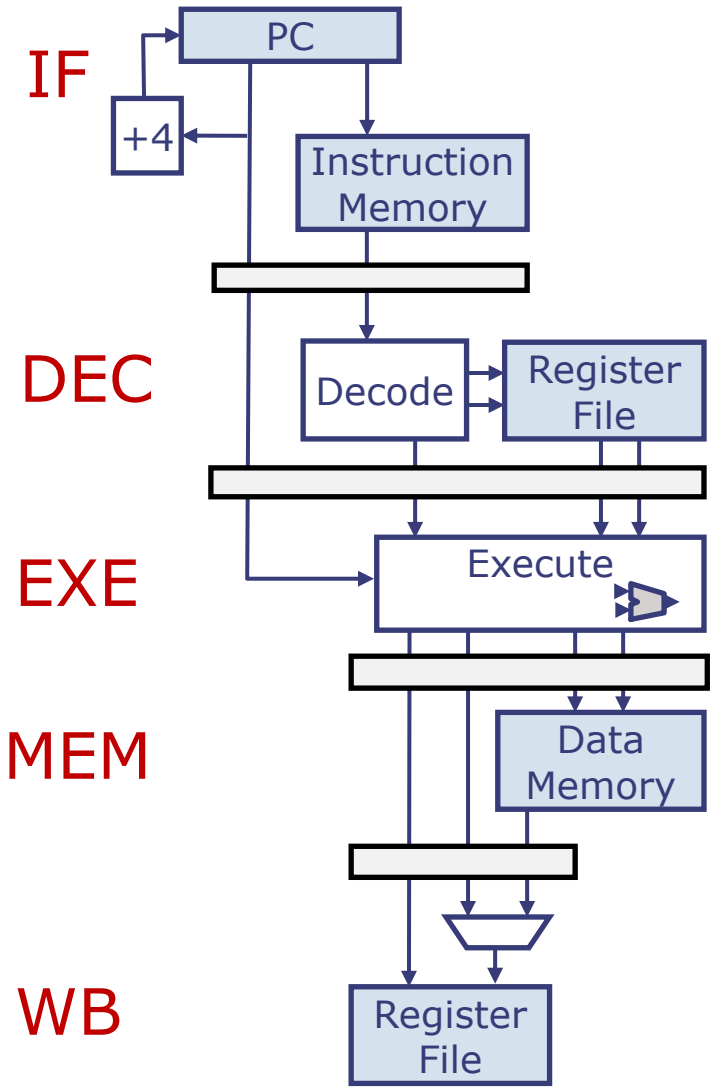
Simplified Single-Cycle Datapath



- For now, nextPC = PC+4 (no branches or jumps)
- Same register file appears twice in the diagram
  - Top: reads are combinational
  - Bottom: writes are clocked
- Magic memories
  - Loads are combinational
  - Return data in same clock cycle

$$t_{CLK} \approx t_{IMEM} + t_{DEC} + t_{RF} + t_{EXE} + t_{DMEM} + t_{WB}$$

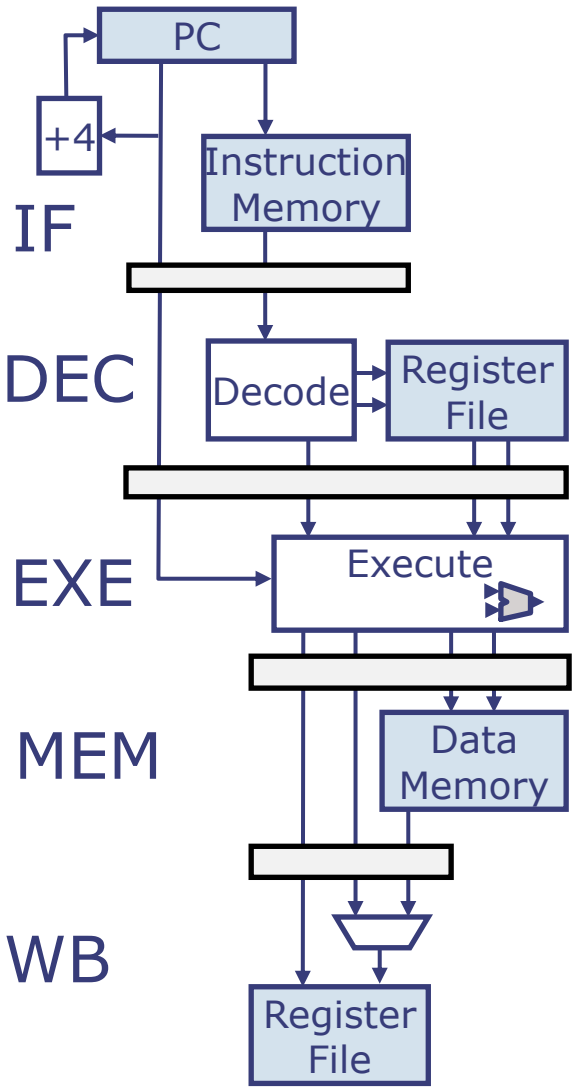
Classic 5-Stage Pipelined Datapath



- Pipeline registers separate different stages
- Each stage services one instruction per cycle

$t_{CLK} = \max\{t_{IF}, t_{DEC}, t_{EXE}, t_{MEM}, t_{WB}\}$

Example: Pipelined Execution




```
addi x11, x10, 2
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
```

When do register reads and writes happen?

Reads in DEC stage
Writes at end of WB stage

Cycles →

Stages

	1	2	3	4	5	6
IF	addi	lw	sub	xor	add	addi
DEC		addi	lw	sub	xor	add
EXE			addi	lw	sub	xor
MEM				addi	lw	sub
WB					addi	lw

Read x10

Write x11

CPI in Pipelined Processor

```
addi x11, x10, 2
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
```

Latency per instr = 5 * t_{CLK}

t_{CLK} = max{t_{IF}, t_{DEC}, t_{EXE}, t_{MEM}, t_{WB}} ↓

Throughput = 1/ t_{CLK} = 1 instr/cycle

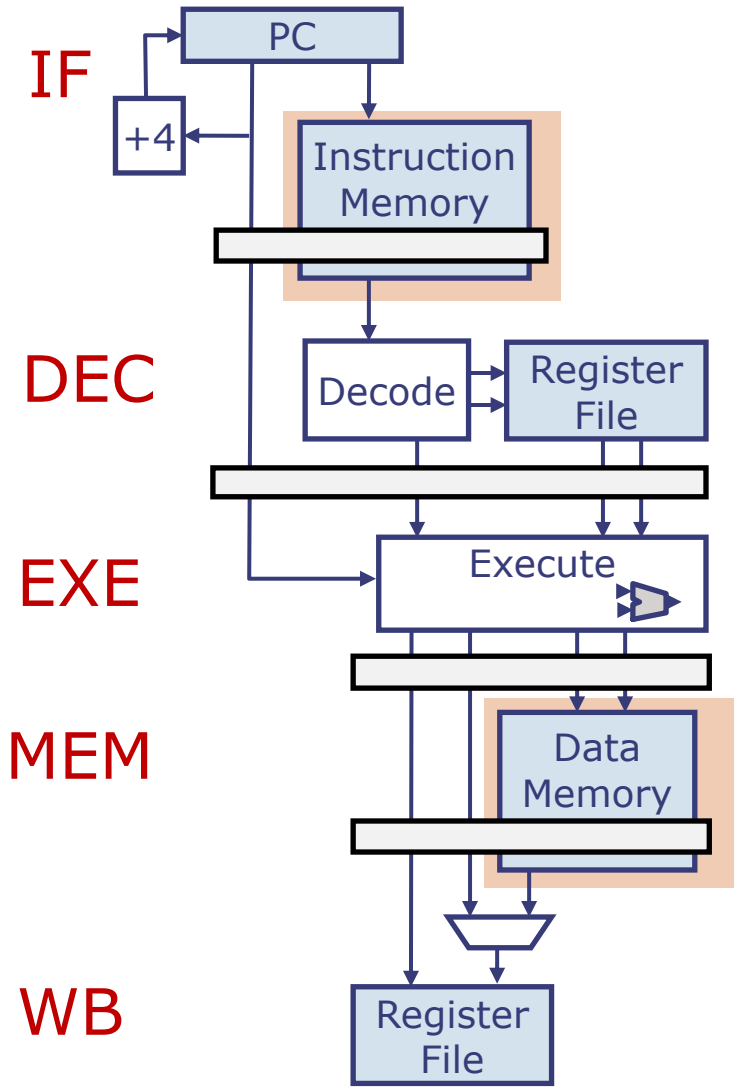
CPI = 1

Better Performance

Cycles →

Stages	1	2	3	4	5	6	7	8
IF	addi	lw	sub	xor	add	addi		
DEC		addi	lw	sub	xor	add	addi	
EXE			addi	lw	sub	xor	add	addi
MEM				addi	lw	sub	xor	add
WB					addi	lw	sub	xor

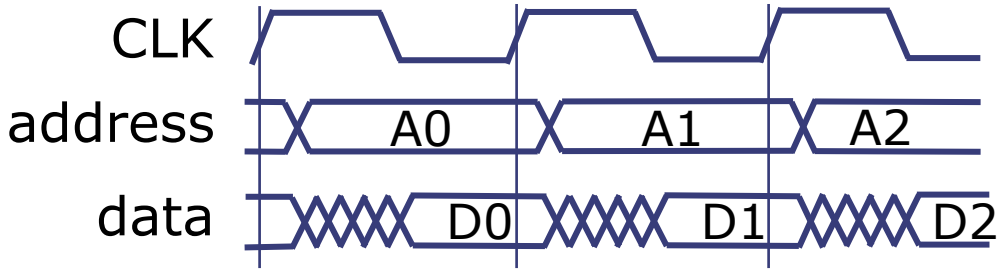
Classic 5-Stage RISC Pipeline with Clocked Memory Reads

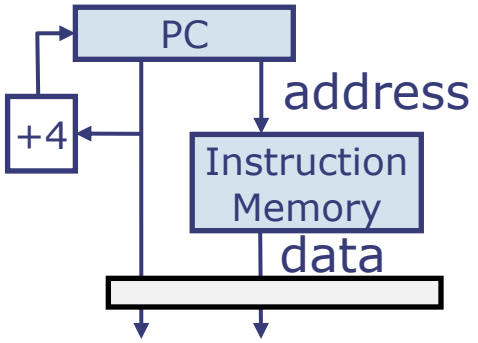


- Pipeline registers separate different stages
- Each stage services one instruction per cycle
- For now, nextPC = PC+4 (no branches or jumps)
- Instruction and data memories have clocked reads

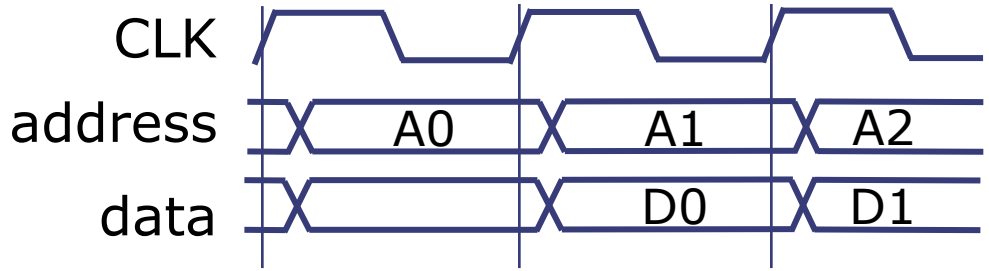
Clarification: Memories with Combinational vs. Clocked Reads

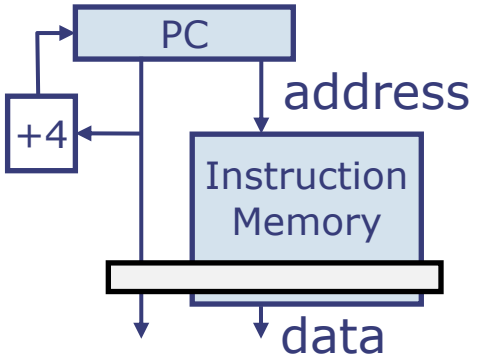
- Lab 6, instruction and data magic memories have combinational reads:
  reads complete in the same cycle





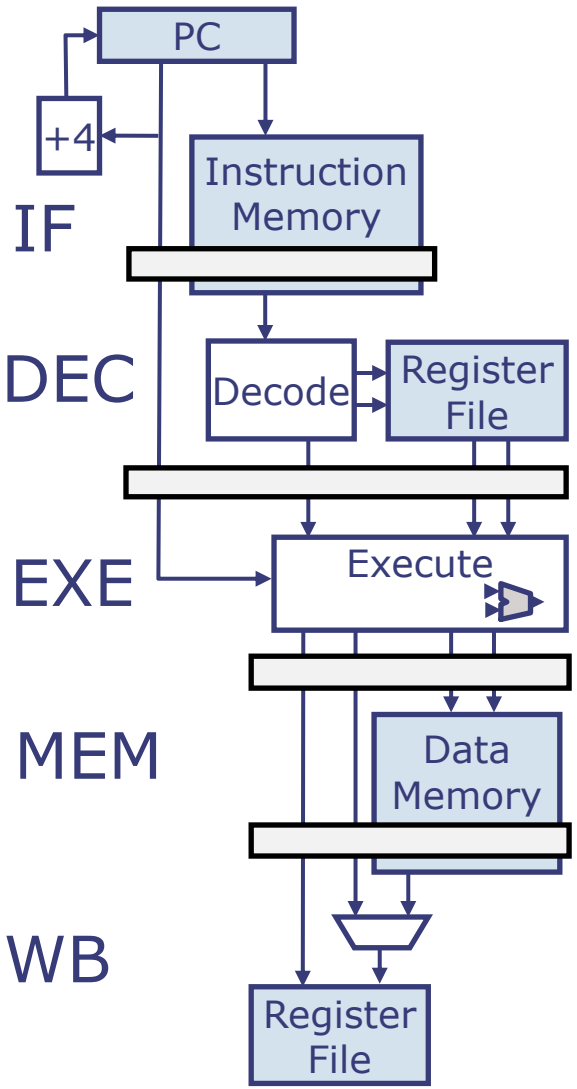
- Lab 7 and DP, we will use memories with clocked reads: read data is available at the beginning of the next cycle assuming cache hit





  - Clocked reads are more common
  - Clocked reads simplify some pipelining issues

Pipelined Execution




```
addi x11, x10, 2
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
```

When do register reads and writes happen?

Reads in DEC stage
Writes at end of WB stage

Cycles →

Stages

	1	2	3	4	5	6
IF	addi	lw	sub	xor	add	addi
DEC		addi	lw	sub	xor	add
EXE			addi	lw	sub	xor
MEM				addi	lw	sub
WB					addi	lw

Read x14

Write x13

Data Hazards

- Consider this instruction sequence:

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

	1	2	3	4	5	6
IF	addi	xor	sub	xori		
DEC		addi	xor	sub	xori	
EXE			addi	xor	sub	xori
MEM				addi	xor	sub
WB					addi	xor

- xor reads x11 on cycle 3, but addi does not update it until end of cycle 5 → x11 is stale!
- Pipeline must maintain correct behavior…

Pipeline Hazards

- Pipelining tries to overlap the execution of multiple instructions, but an instruction may depend on something produced by an earlier instruction
  - A data value → Data hazard
  - The program counter → Control hazard (branches, jumps, exceptions)

- Plan of attack:
  1. Design a 5-stage pipeline that works with sequences of independent instructions
  2. Handle data hazards
  3. Handle control hazards

Today: 1 and 2
Next time: 3

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 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
```

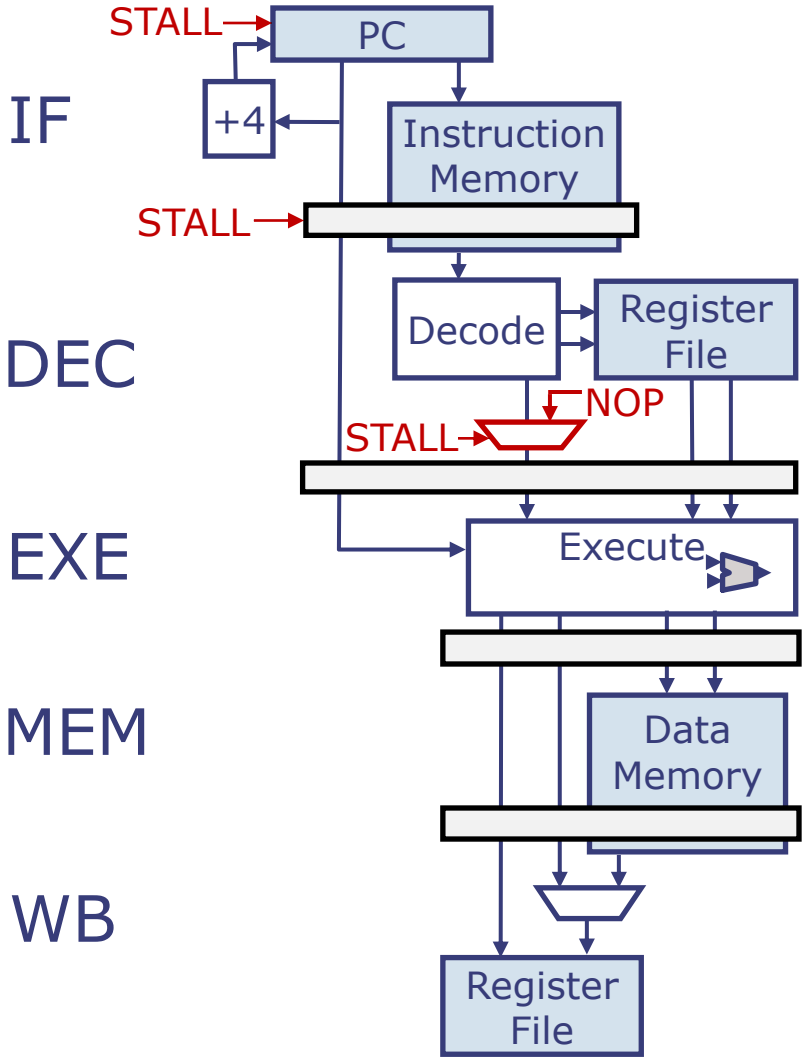
Stall

	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!

Stall Logic



- New STALL control signal
- STALL==1
  - Disables PC and IF pipeline register (freezing IF and DEC)
  - Injects NOP instruction into EXE stage
- NOP = No-operation, e.g., addi x0, x0, 0
- Control logic sets STALL=1 if source registers of instruction in DEC match destination register in EXE, MEM, or WB *(except when source is x0!)*

Data Hazards due to Loads & Stores

- Is there any possible data hazard in this instruction sequence?

```
sw x11, 8(x10)
lw x13, 4(x12)
…
```

```
mem[reg[10] + 8] ← reg[11]
reg[13] ← mem[reg[12] + 4]
```

- What if reg[10] + 8 == reg[12] + 4?

- With single cycle memory, no hazards through memory because stores complete in one cycle, but this is not true in general
  - In general, load & store hazards may be solved in the pipeline or in the memory system

Load and Store with Single Cycle Memory

reg[10] + 8 == reg[12] + 4

X X

```
sw x11, 8(x10)
lw x13, 4(x12)
…
```

	1	2	3	4	5	6
IF	sw	lw				
DEC		sw	lw			
EXE			sw	lw		
MEM				sw	lw	
WB					sw	lw

Write mem[X]

Read mem[X]

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
```

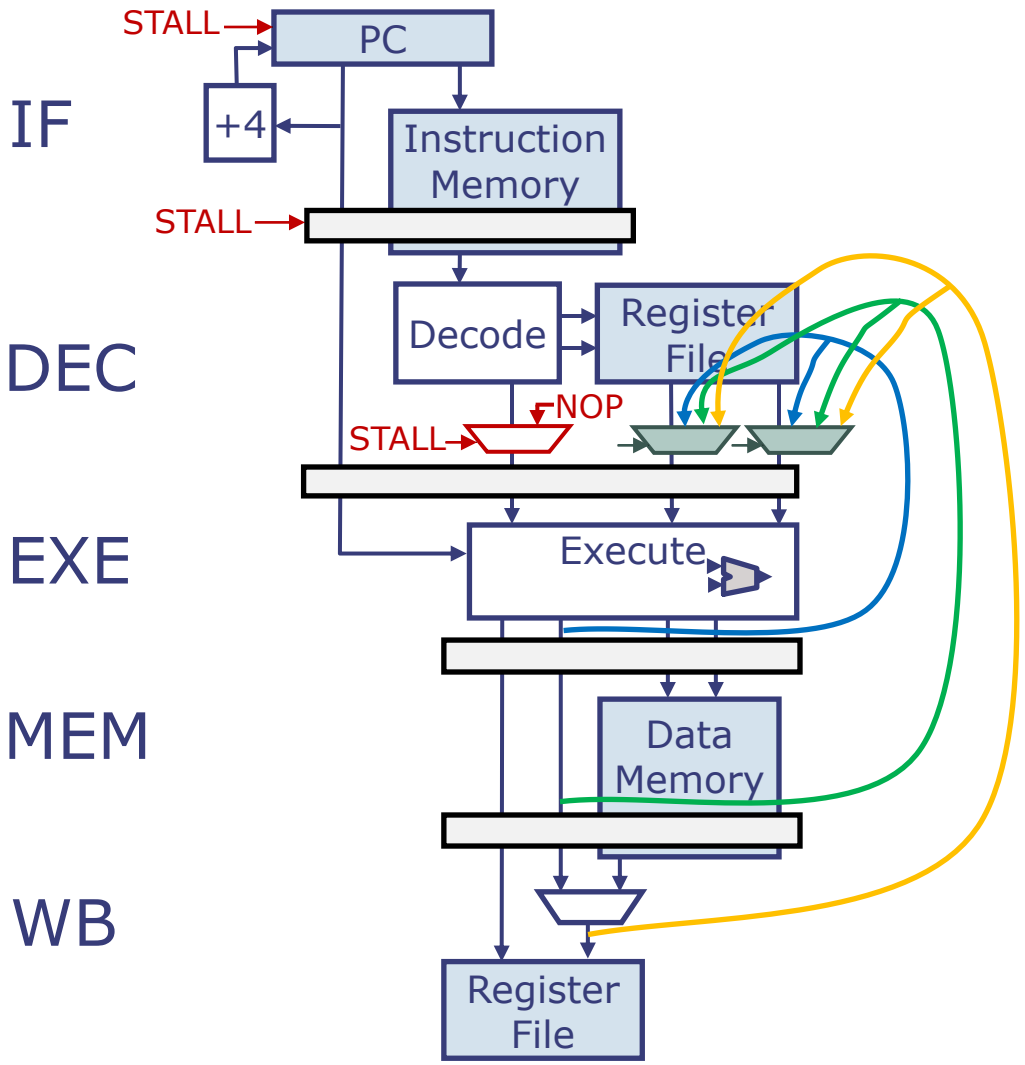
- addi writes to x11 at the end of cycle 5… but the result is produced during cycle 3, at the EXE stage!

	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

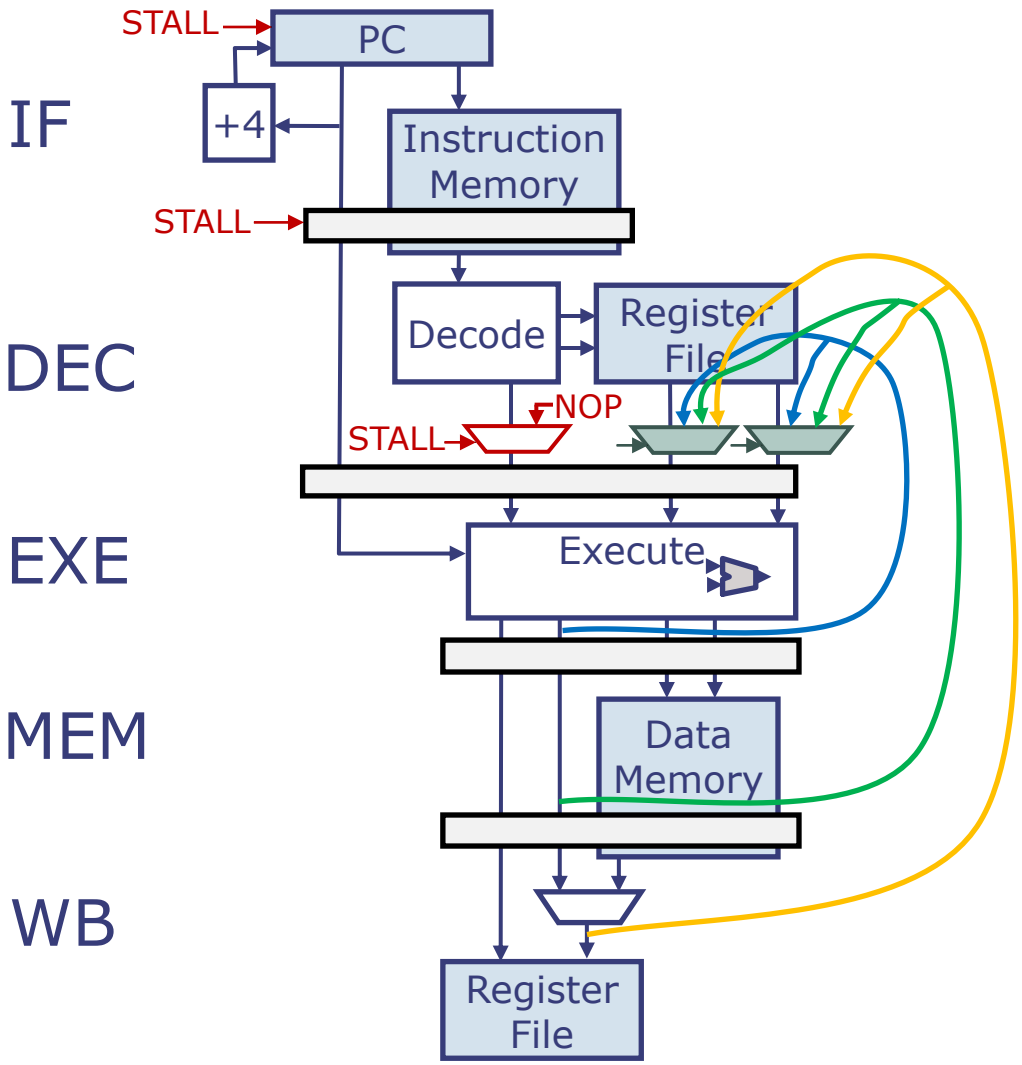
x11 updated

Bypass Logic



- Add bypass muxes to DEC outputs
- Route EXE, MEM, WB outputs to mux inputs
- Bypass value if destination register of instruction in EXE, MEM, or WB matches source register of instruction in DEC
  - No bypass for x0
- If multiple matches, use value from most recent instruction (EXE > MEM > WB)

Full vs. Partial Bypassing



- Bypasses are expensive
  - Lots of wires & large muxes
  - May affect clock cycle time…
- But full bypassing is not needed! We can always stall
  - e.g., just bypass from EXE and WB
- With a fully bypassed pipeline, do we still need the stall signal?

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	lw	xor	sub	*sub*	*sub*	xori		
DEC		lw	xor	*xor*	*xor*	sub	xori	
EXE			lw	**NOP**	**NOP**	xor	sub	xori
MEM				lw	**NOP**	**NOP**	xor	sub
WB					lw	**NOP**	**NOP**	xor

lw data available

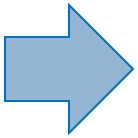
x11 updated

Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away
- Example:

```
lw x11, 0(x10)
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

2 stalls (w/ bypasses)



```
lw x11, 0(x10)
sub x17, x15, x16
xori x19, x18, 0xF
xor x13, x11, x12
```

No stalls

- Only works well when compiler can find independent instructions to move around!

Summary: 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 use fewer bypasses & stall more often
- More pipeline stages → More frequent data hazards
  - Lower t_{CLK}, but higher CPI

Thank you!

Next lecture: Control Hazards