6.5930/1
Hardware Architecture for Deep Learning

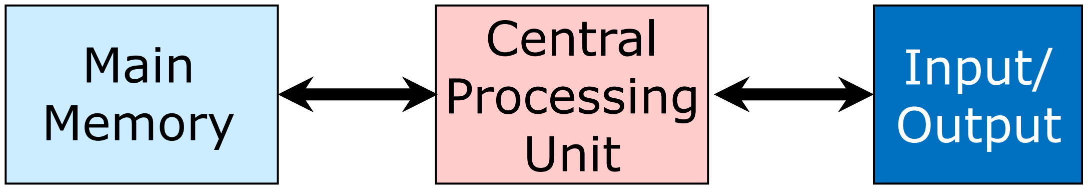
Computer Architecture Basics I - Pipelining

Zoey Song
MIT

Legacy slides adapted from 6.191/6.5900

The von Neumann Model

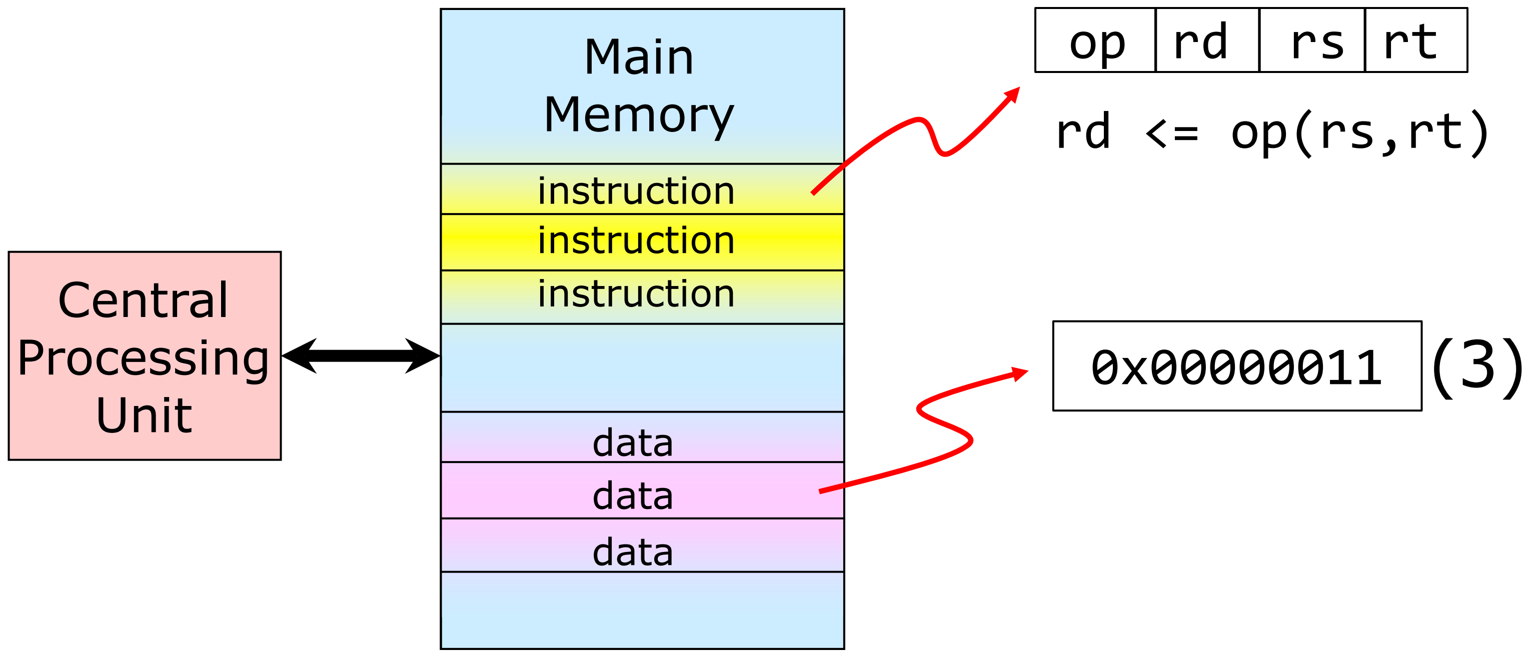
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



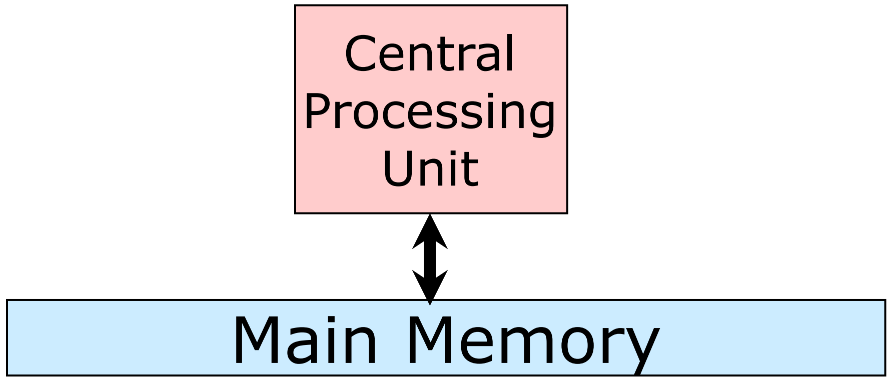
  - Main memory holds programs and their data
  - Central processing unit accesses and processes memory values
  - Input/output devices to communicate with the outside world

Key Idea: Stored-Program Computer

- Express program as a sequence of coded instructions
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

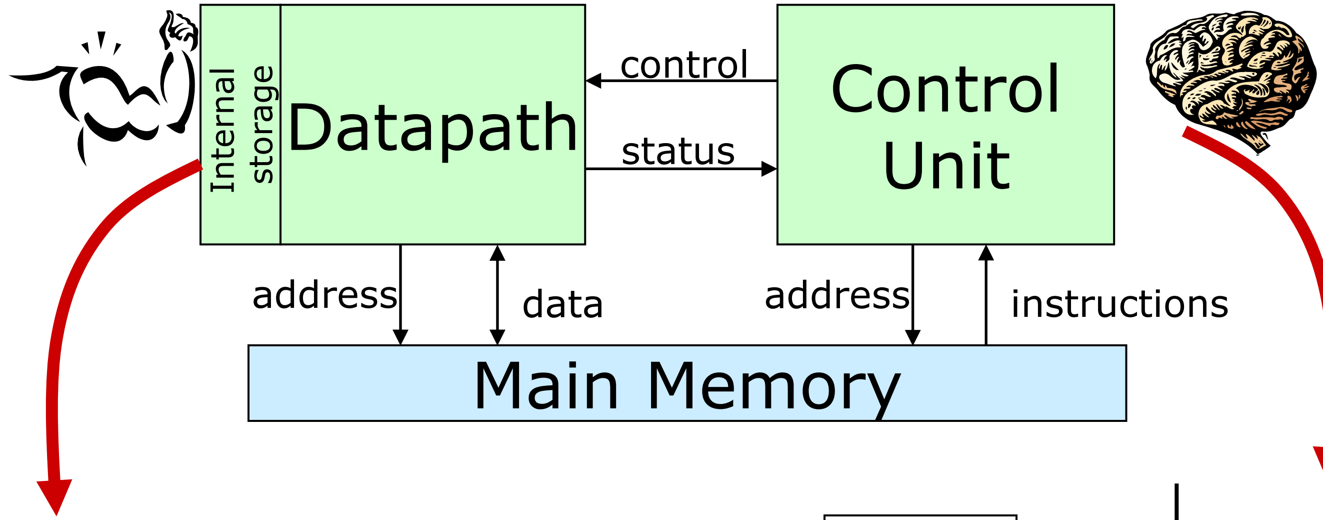


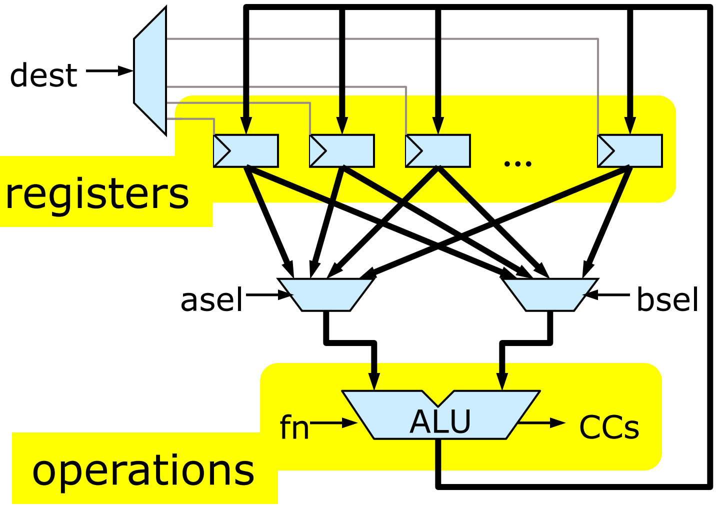
Anatomy of a von Neumann Computer

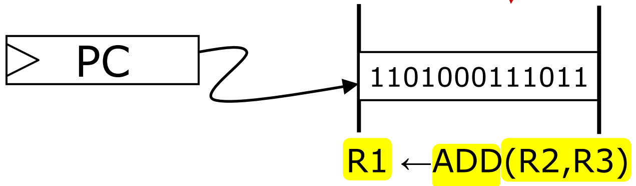


How does CPU distinguish between instructions and data?

Anatomy of a von Neumann Computer





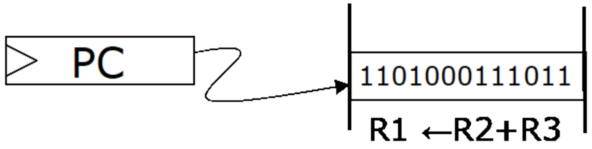


- Instructions coded as binary data
- Program Counter or PC: Address of the instruction to be executed
- Decode Instructions for Datapath

Instructions

- Instructions are the fundamental unit of work

- Each instruction specifies:
  - An operation or opcode to be performed
  - Source operands and destination for the result



- In a von Neumann machine, instructions are executed sequentially
  - By default, the next PC is current PC + size of current instruction (e.g., PC + 4)
  - Except for branch instruction

```
      loop: addi x12, x12, -1
            sub x14, x15, x16
PC   ➡      bne x12, x0, loop
PC+4 ➡      and x16, x17, x18
            xor x19, x20, x21
            …
```

ALU Instructions

$$\underbrace{\texttt{0000000}}_{\text{funct7}}\overbrace{\texttt{00001}}^{\text{rs2}}\overbrace{\texttt{00010}}^{\text{rs1}}\underbrace{\texttt{000}}_{\text{funct3}}\overbrace{\texttt{00011}}^{\text{rd}}\overbrace{\texttt{0110011}}^{\text{opcode}}$$

- What RISC-V instruction is represented by these 32 bits?
- Reference manual specifies the fields as follows:
  - `opcode = 0110011` => Which operation?
  - `funct3 = 000` => More specific info on op (ADD)
  - `funct7 = 0000000`
  - `rd = 00011` => x3
  - `rs1 = 00010` => x2
  - `rs2 = 00001` => x1

ADD x3, x2, x1

ALU Instructions

Instruction	Description	Execution
ADD rd, rs1, rs2	Add	reg[rd] <= reg[rs1] + reg[rs2]
SLL rd, rs1, rs2	Shift Left Logical	reg[rd] <= reg[rs1] << reg[rs2]
SLT rd, rs1, rs2	Set if < (Signed)	reg[rd] <= (reg[rs1] $<_s$ reg[rs2]) ? 1 : 0
…		

Grouped in a category called OP with fields (AluFunc, rd, rs1, rs2)

Instruction	Description	Execution
ADDI rd, rs1, immI	Add Immediate	reg[rd] <= reg[rs1] + immI
SLLI rd, rs1, immI	Shift Left Logical Immediate	reg[rd] <= reg[rs1] << immI
SLTI rd, rs1, immI	Set if < Immediate (Signed)	reg[rd] <= (reg[rs1] $<_s$ immI) ? 1 : 0
…		

Grouped in a category called OPIMM with fields (AluFunc, rd, rs1, immI)

Load and Store Instructions

Instruction	Description	Execution
LW rd, immI(rs1)	Load Word	reg[rd] <= mem[reg[rs1] + immI]
SW rs2, immS(rs1)	Store Word	mem[reg[rs1] + immS] <= reg[rs2]

LW and SW need to access memory for execution and thus, are required to compute an effective memory address

Branch Instructions

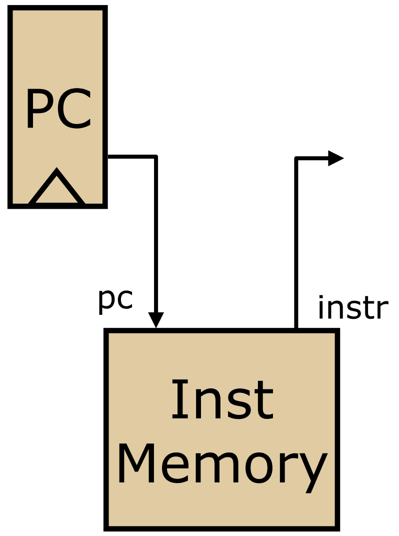
differ only in the aluBr operation they perform

Instruction	Description	Execution
BEQ rs1, rs2, immB	Branch =	pc <= (reg[rs1] == reg[rs2]) ? pc + immB : pc + 4
BNE rs1, rs2, immB	Branch !=	pc <= (reg[rs1] != reg[rs2]) ? pc + immB : pc + 4
BLT rs1, rs2, immB	Branch < (Signed)	pc <= (reg[rs1] $<_s$ reg[rs2]) ? pc + immB : pc + 4
BGE rs1, rs2, immB	Branch ≥ (Signed)	pc <= (reg[rs1] $\geq_s$ reg[rs2]) ? pc + immB : pc + 4
BLTU rs1, rs2, immB	Branch < (Unsigned)	pc <= (reg[rs1] $<_u$ reg[rs2]) ? pc + immB : pc + 4
BGEU rs1, rs2, immB	Branch ≥ (Unsigned)	pc <= (reg[rs1] $\geq_u$ reg[rs2]) ? pc + immB : pc + 4

These instructions are grouped in a category called BRANCH with fields (brFunc, rs1, rs2, immB)

Single-Cycle RISC-V Processor

Fetch inst

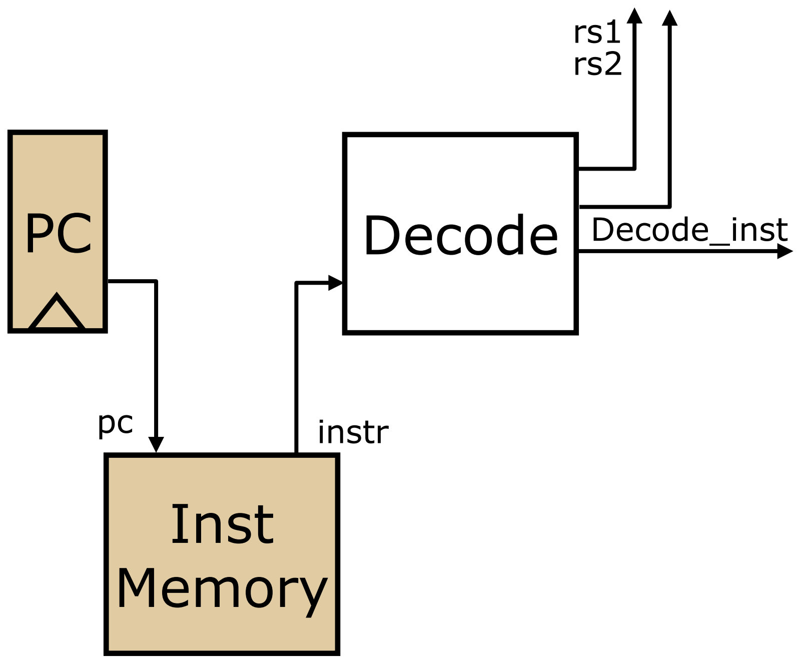


1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

Single-Cycle RISC-V Processor

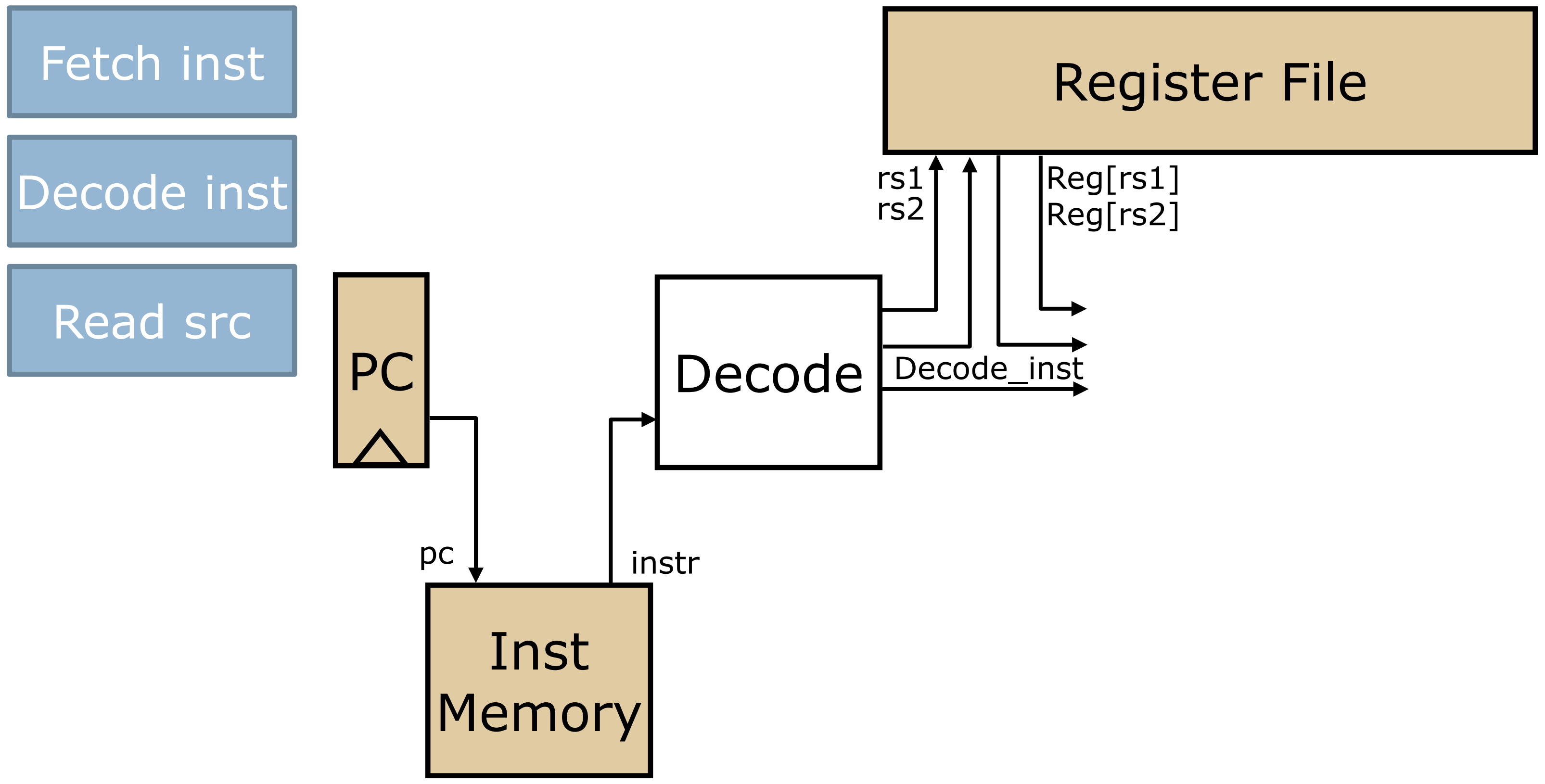
Fetch inst

Decode inst



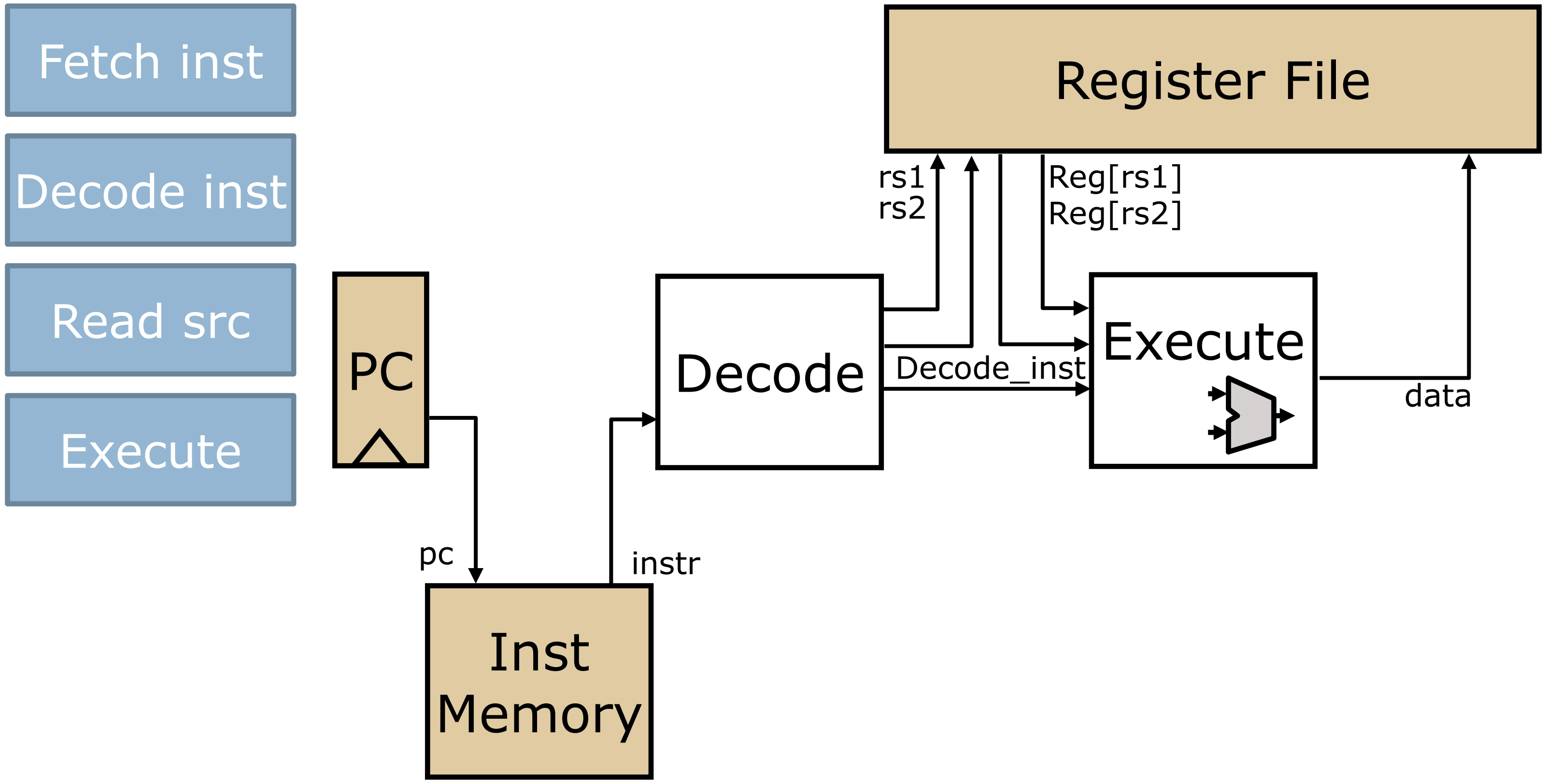
1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

Single-Cycle RISC-V Processor



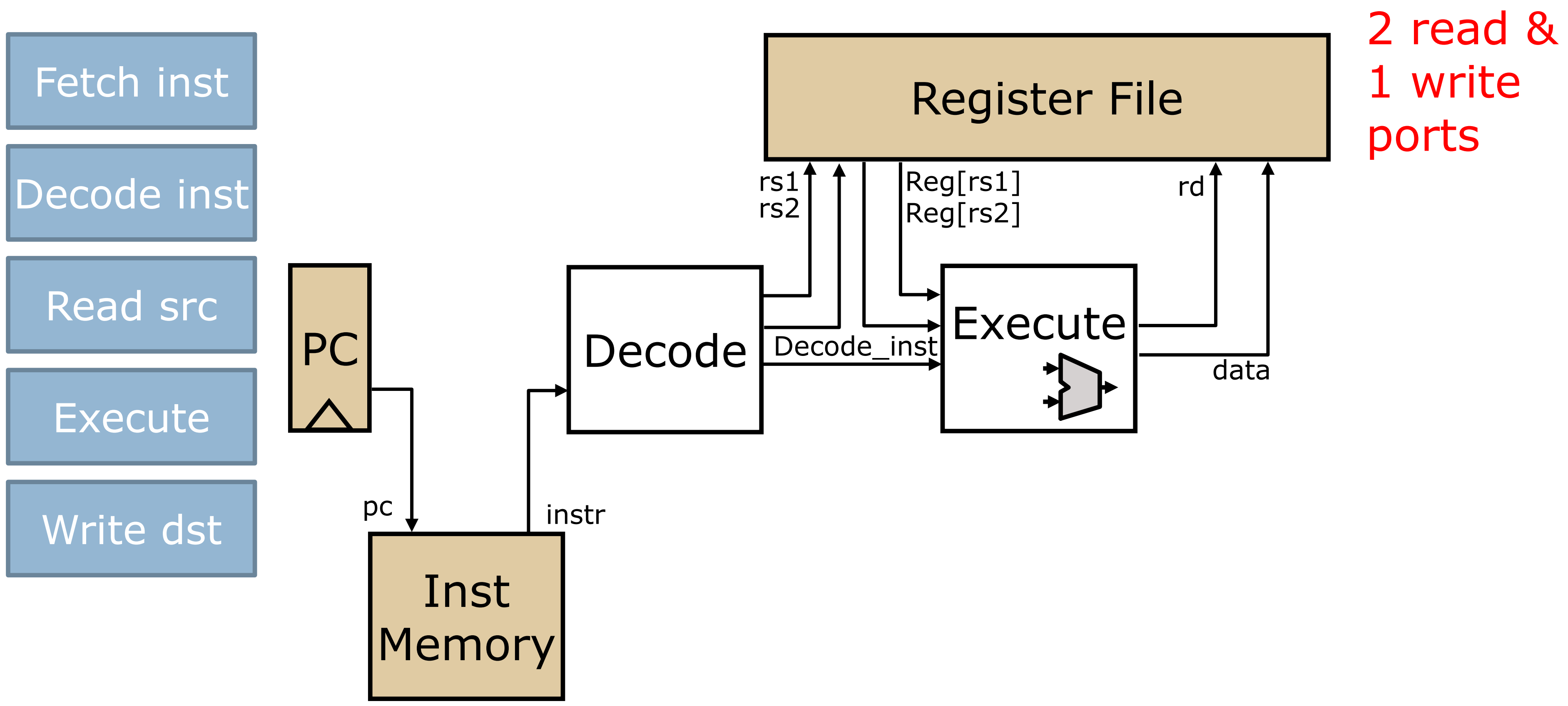
1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

Single-Cycle RISC-V Processor



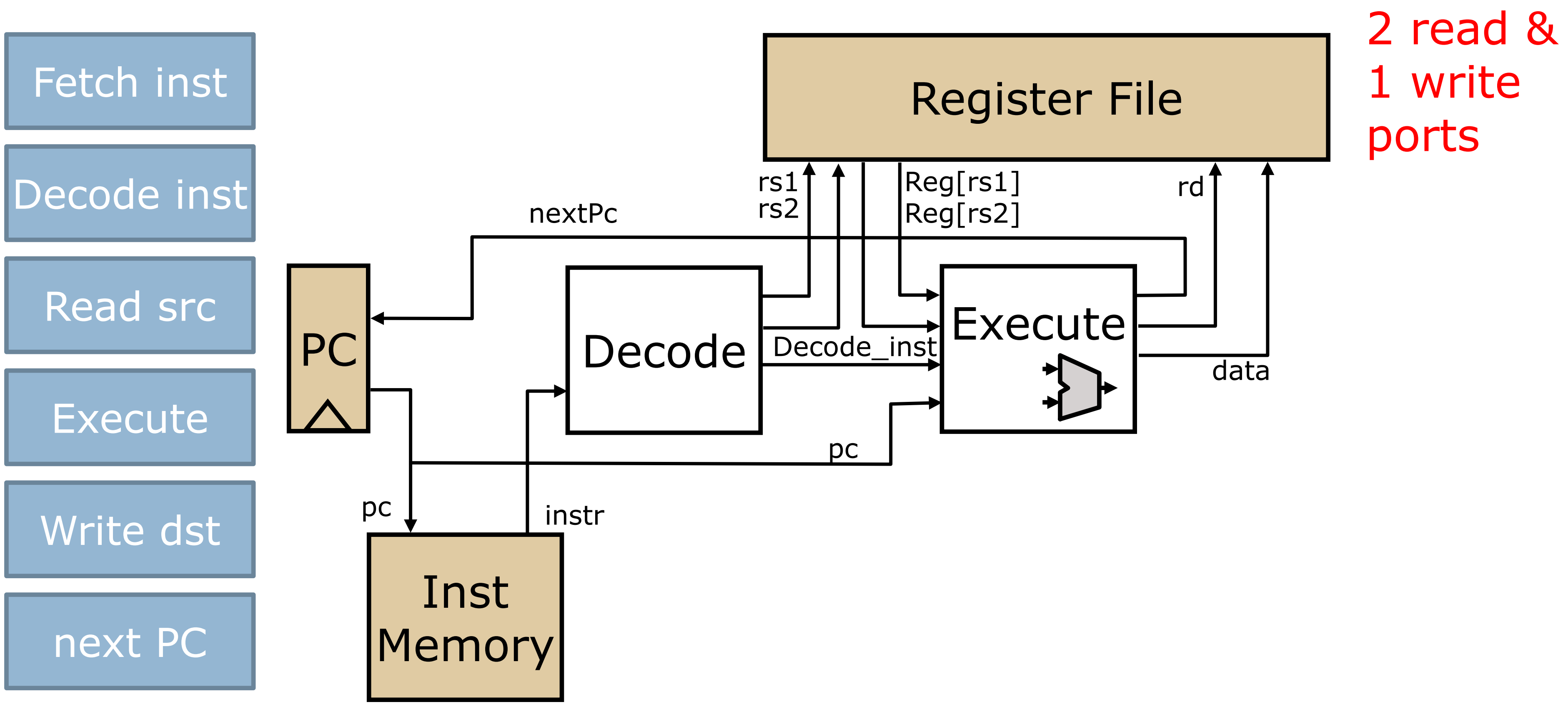
1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

Single-Cycle RISC-V Processor



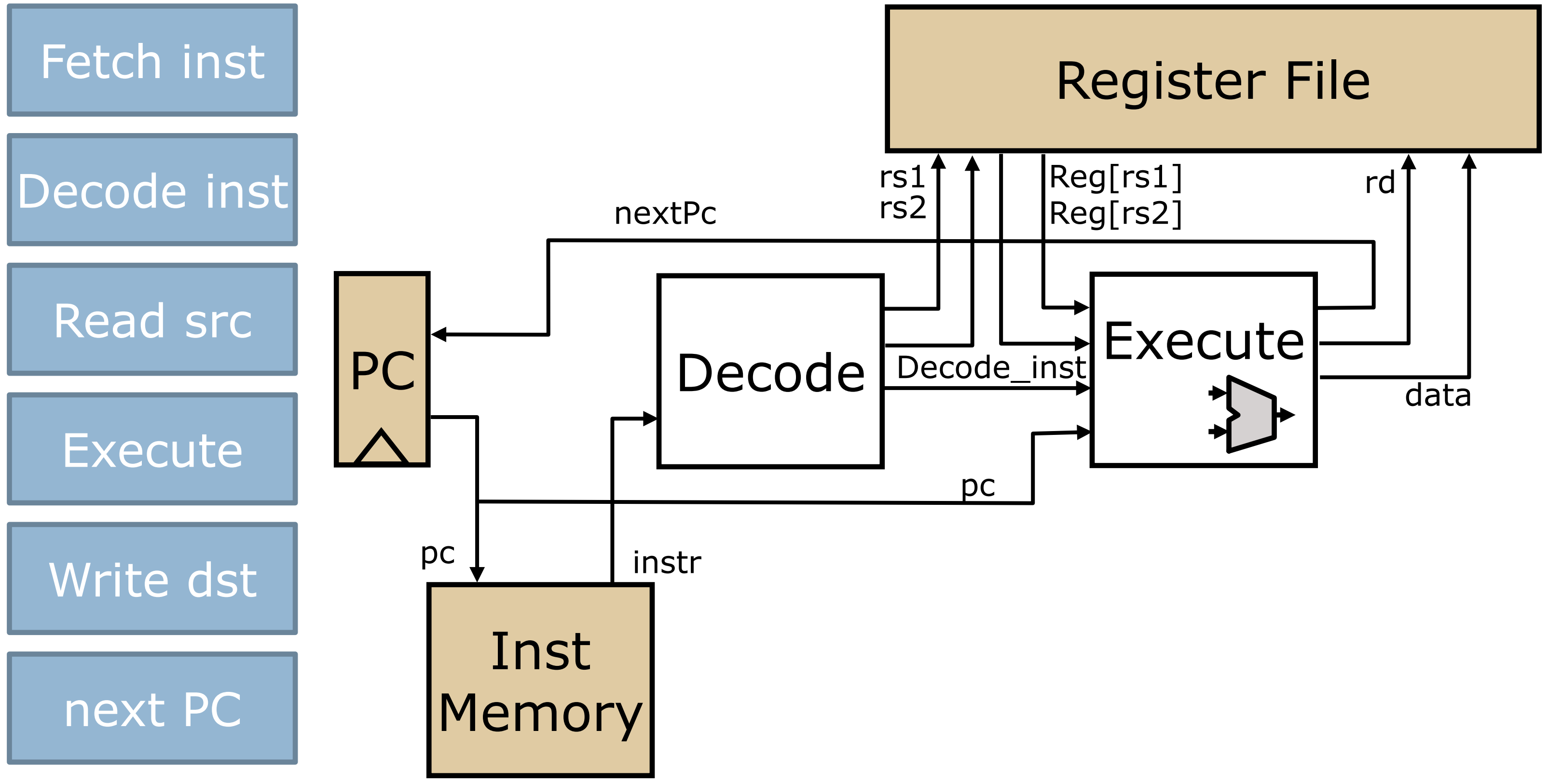
1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

Single-Cycle RISC-V Processor



1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

Single-Cycle RISC-V Processor



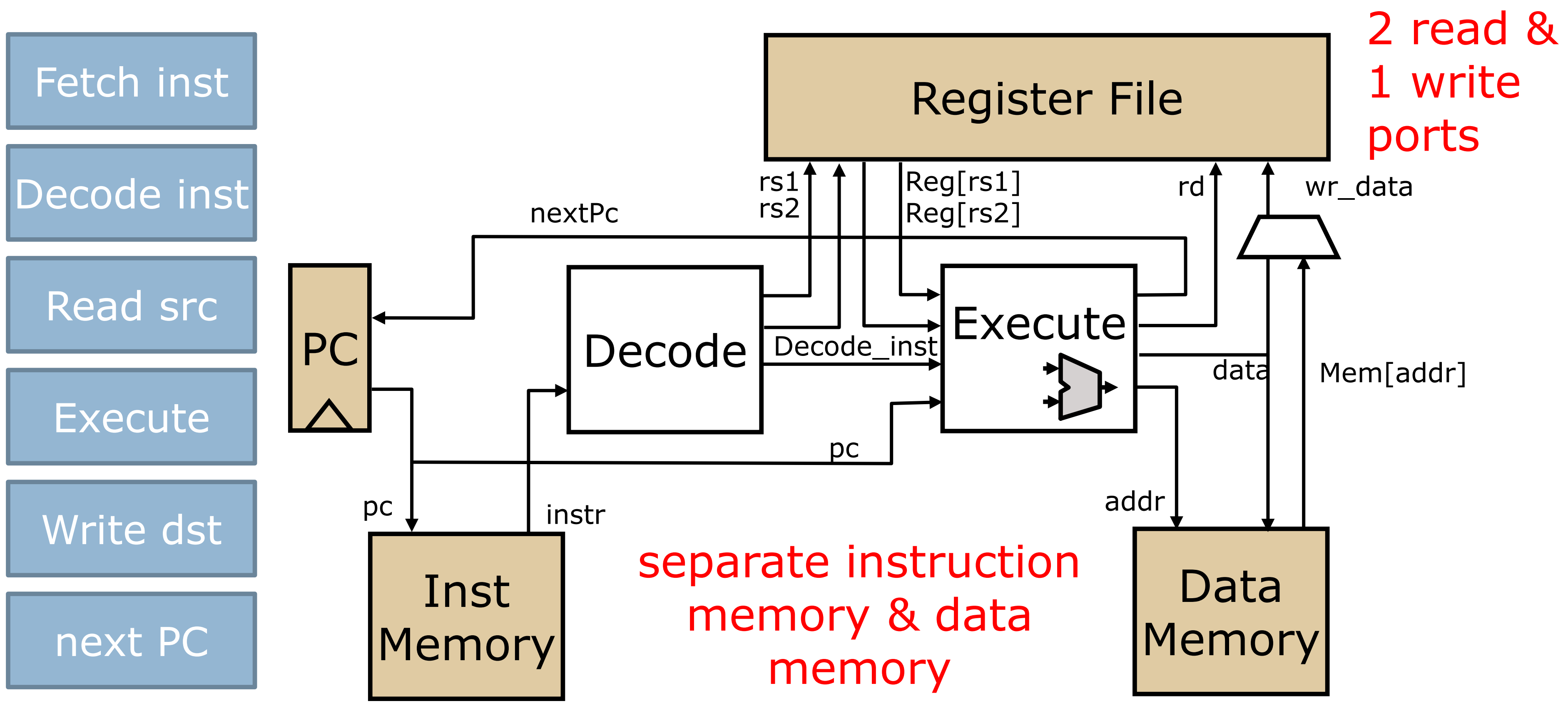
2 read & 1 write ports

1. ALU instructions
2. **Load & store instructions**
3. Branch & jump instructions

`LD rd, (rs1)`

reg[rd] <= mem[reg[rs1]]

Single-Cycle RISC-V Processor



1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

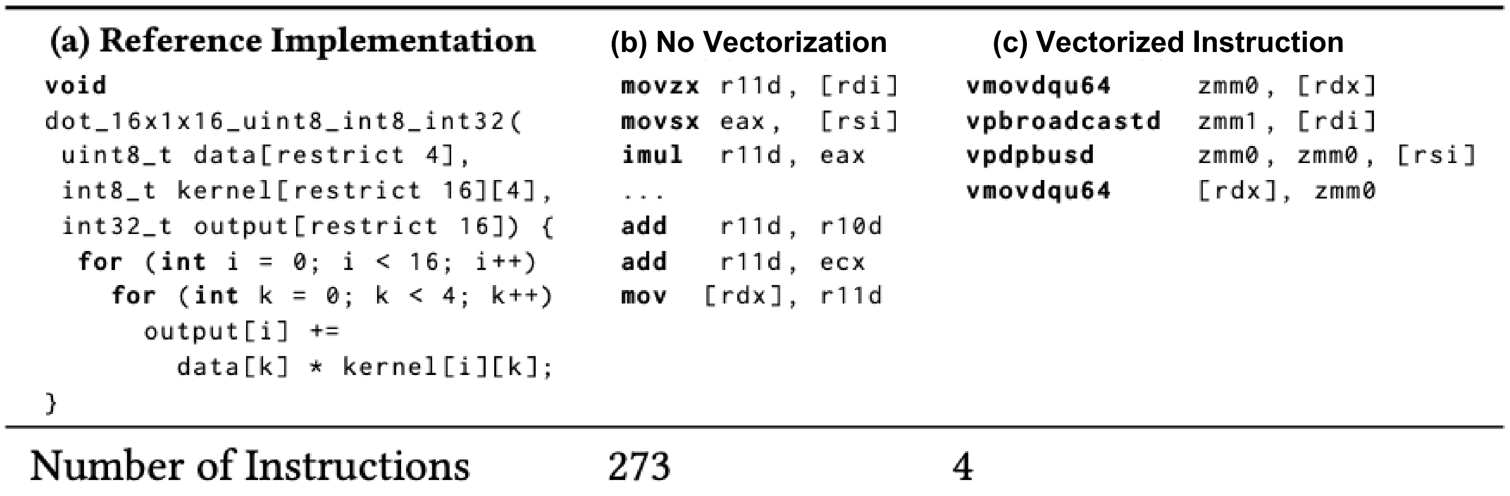
Processor Performance

- "Iron Law" of performance:

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

$$\frac{\text{Program}}{\text{Instruction}}$$

Instruction Set



(a) Reference Implementation

```
void
dot_16x1x16_uint8_int8_int32(
 uint8_t data[restrict 4],
 int8_t kernel[restrict 16][4],
 int32_t output[restrict 16]) {
  for (int i = 0; i < 16; i++)
    for (int k = 0; k < 4; k++)
      output[i] +=
        data[k] * kernel[i][k];
}
```

(b) No Vectorization

```
movzx r11d, [rdi]
movsx eax,  [rsi]
imul  r11d, eax
...
add   r11d, r10d
add   r11d, ecx
mov  [rdx], r11d
```

(c) Vectorized Instruction

```
vmovdqu64     zmm0, [rdx]
vpbroadcastd  zmm1, [rdi]
vpdpbusd      zmm0, zmm0, [rsi]
vmovdqu64     [rdx], zmm0
```

	(b) No Vectorization	(c) Vectorized Instruction
Number of Instructions	273	4

[Reference] VeGen: A Vectorizer Generator for SIMD and Beyond (Chen et al.)

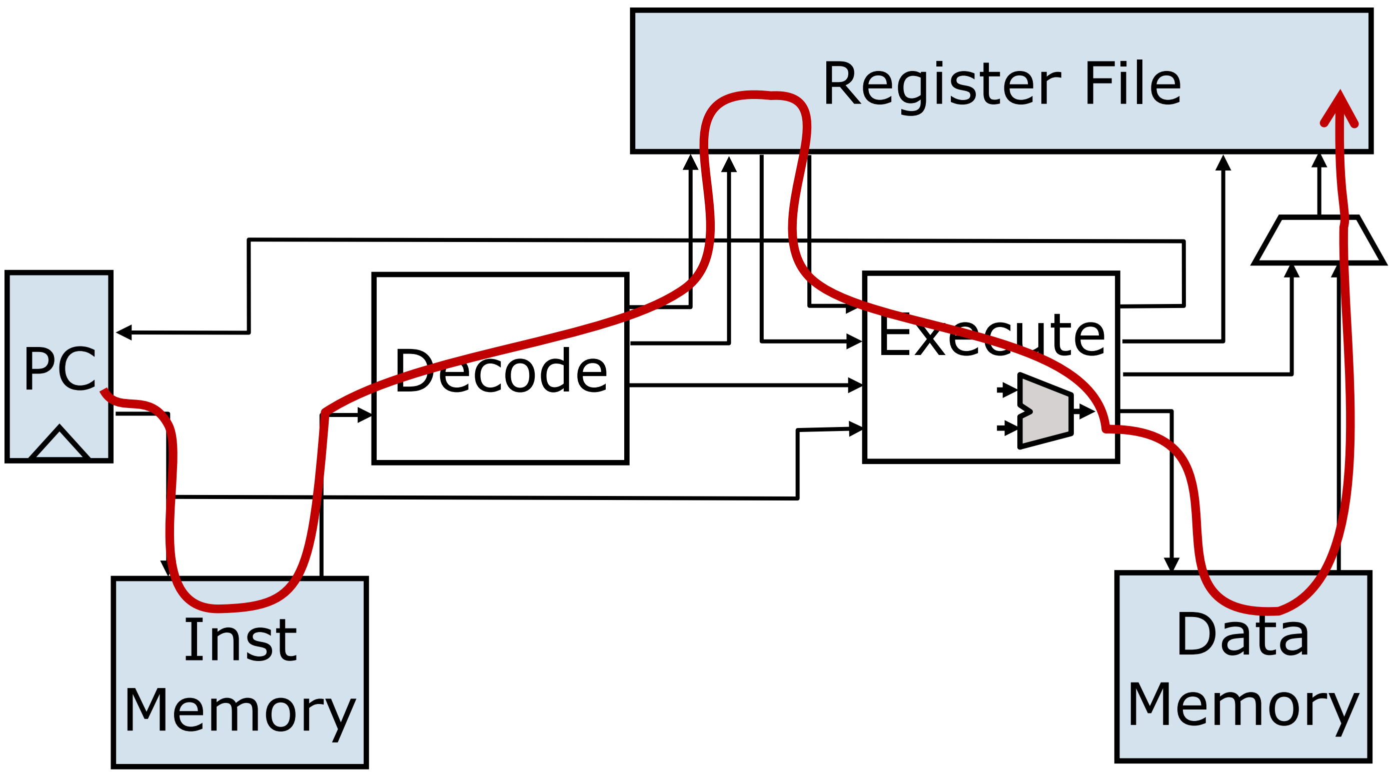
Processor Performance

- "Iron Law" of performance:

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

$\frac{\text{Program}}{\text{Instruction}}$	Instruction Set
$\frac{\text{Instruction}}{\text{Cycle}}$	Microarchitecture Design
$\frac{\text{Cycle}}{\text{Time}}$	Technology, circuits

Single-Cycle Processor Performance



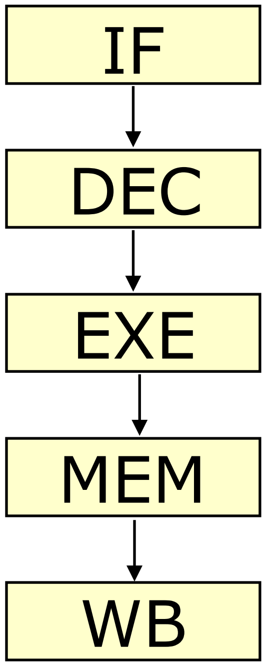
- IPC (Instruction Per Cycle) = 1
- t_{CLK} = Longest path for any instruction

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

Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce t_{CLK}
  - Each instruction executes over multiple cycles

- 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

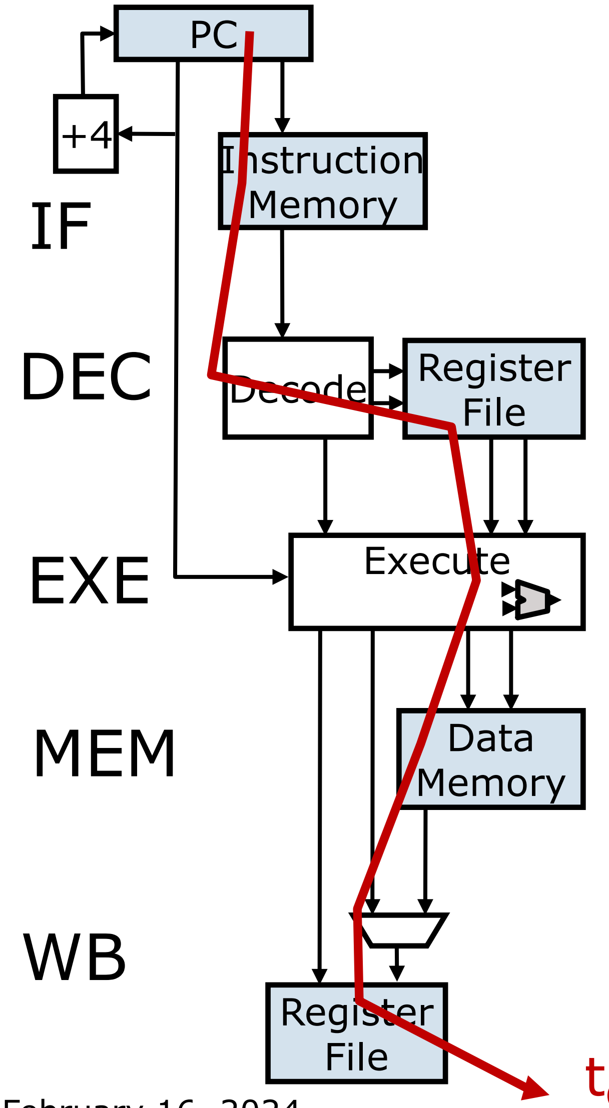
Execute stage: Performs indicated operation in ALU, passes result to

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.

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

Example: Non-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
```

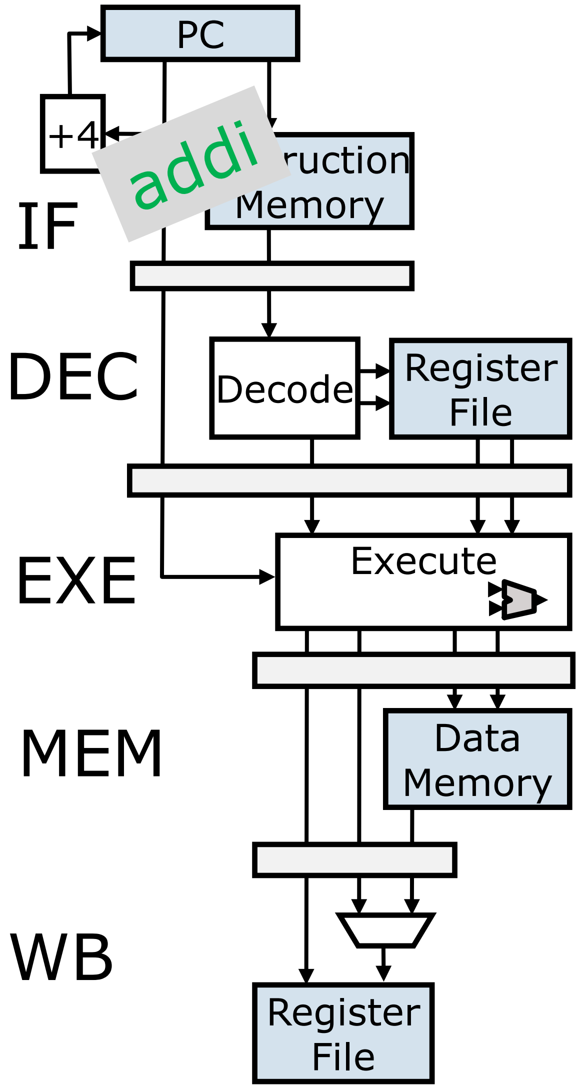
Cycles →

Stages

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

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

Example: Pipelined Execution



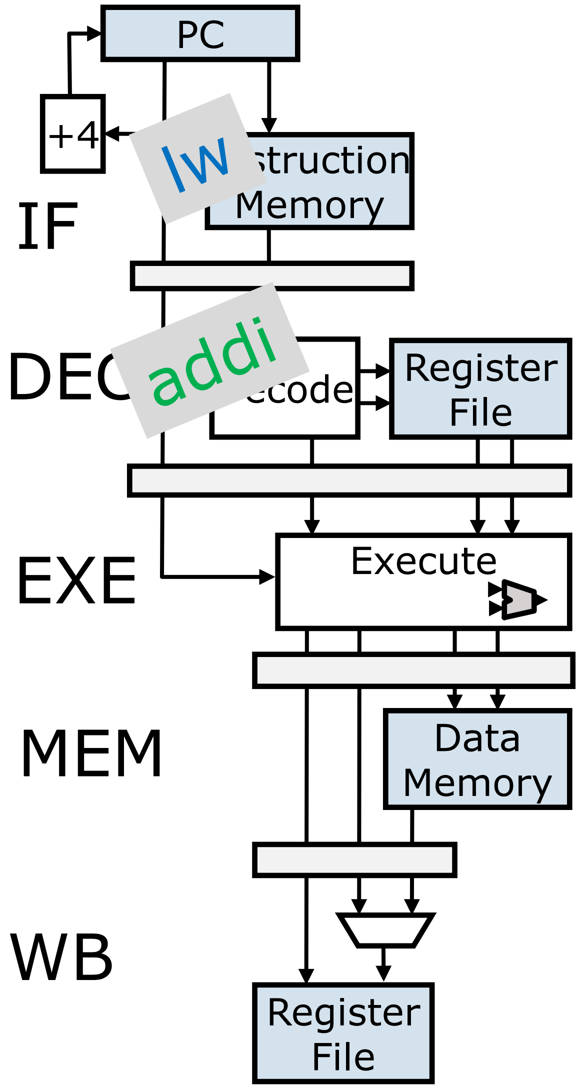
```
addi x11, x10, 2
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
```

Cycles →

Stages

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

Example: Pipelined Execution



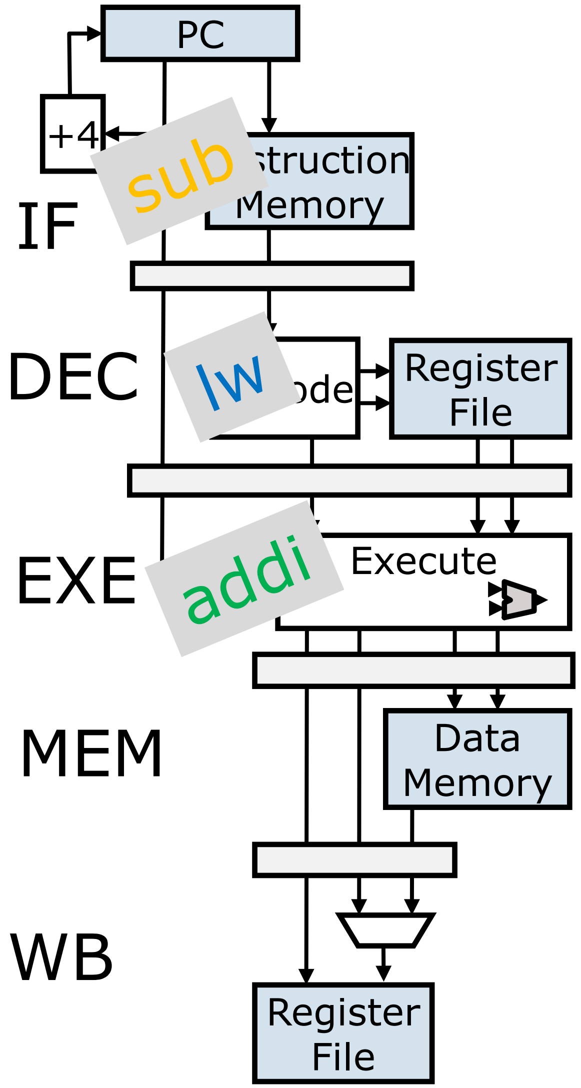
```
addi x11, x10, 2
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
```

Cycles →

Stages

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

Example: Pipelined Execution



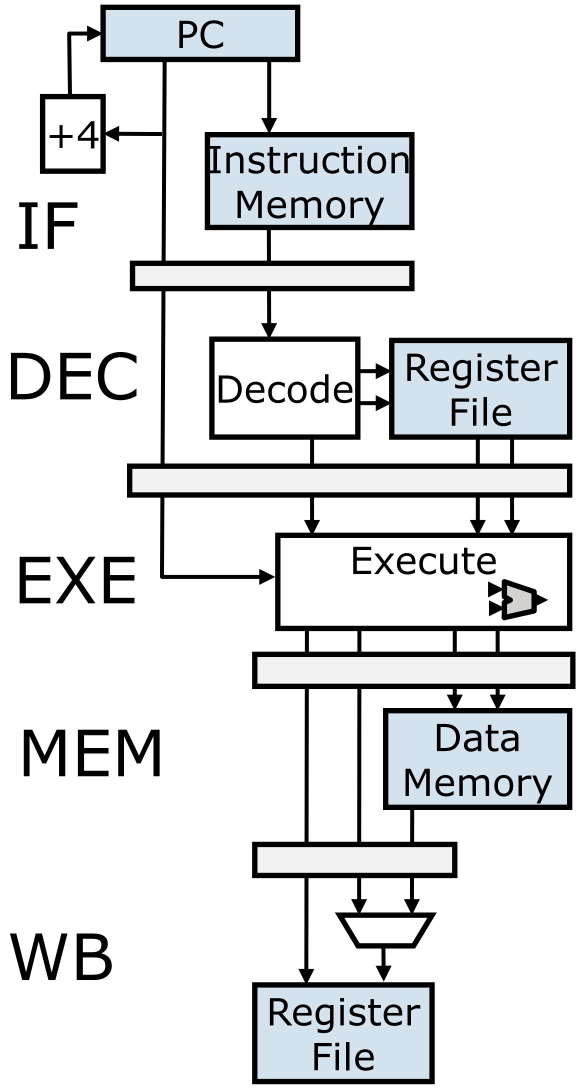
```
addi x11, x10, 2
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
```

Cycles →

Stages

	1	2	3	4	5	6
IF	addi	lw	sub			
DEC		addi	lw			
EXE			addi			
MEM						
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
```

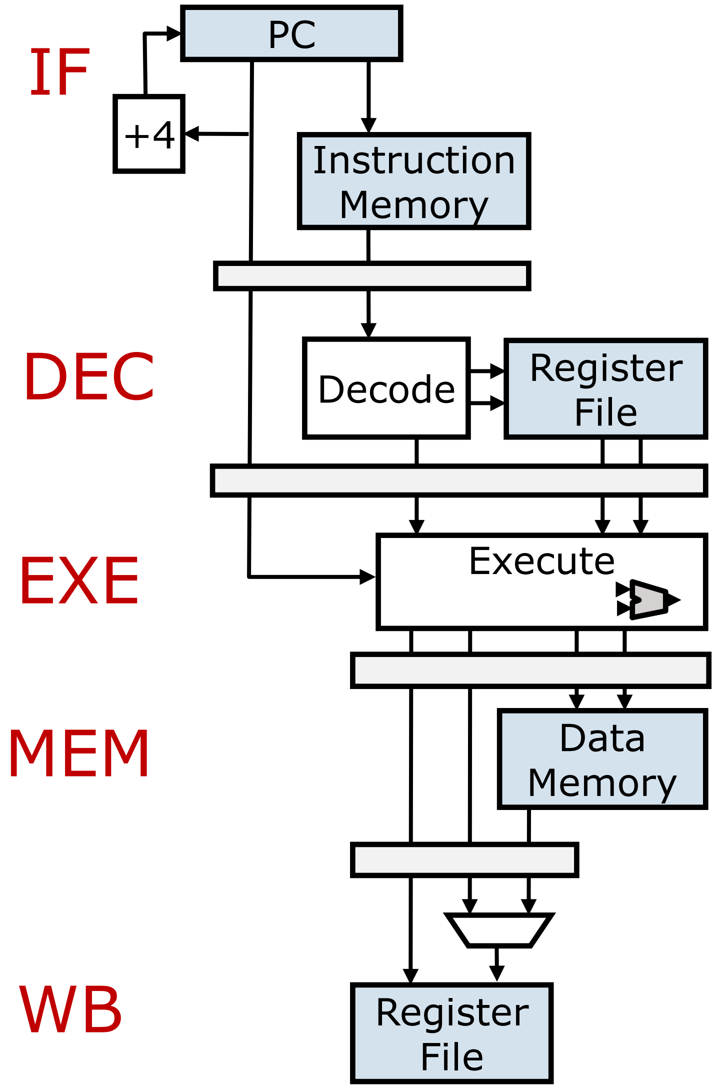
Cycles →

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

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

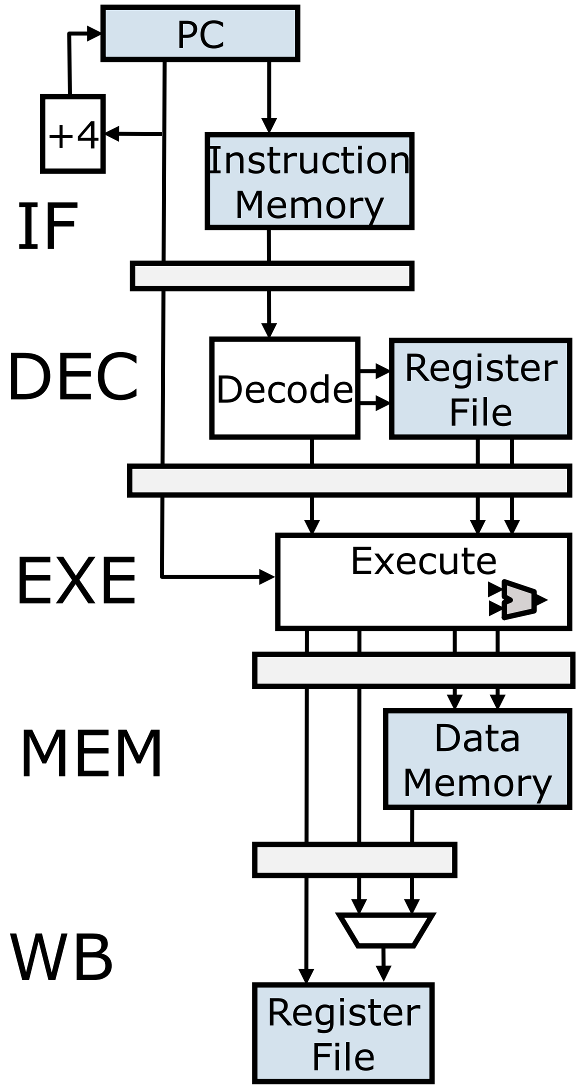
Classic 5-Stage Pipelined Datapath



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

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

Pipeline Hazard



```
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

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	$\boxed{\text{xor}}$	sub	xori	
EXE			addi	xor	sub	xori
MEM				addi	xor	sub
WB					$\boxed{\text{addi}}$	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)

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass (Data hazard). Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate (Control hazard)
  - 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
```

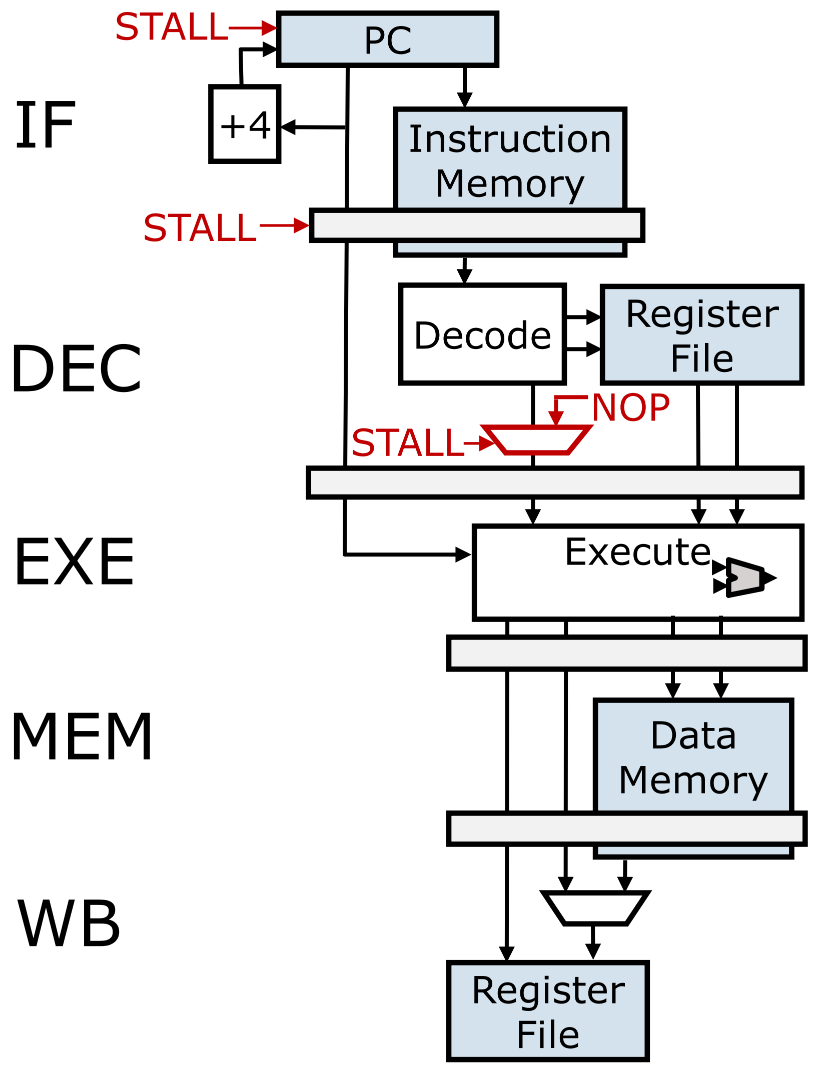
Stall

	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 decrease IPC!

Stall Logic



- New STALL control signal
- STALL==1
  - Freezes PC and IF pipeline
  - Injects NOP into EXE stage
- NOP = No-operation

Resolving Data Hazards by Bypassing

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

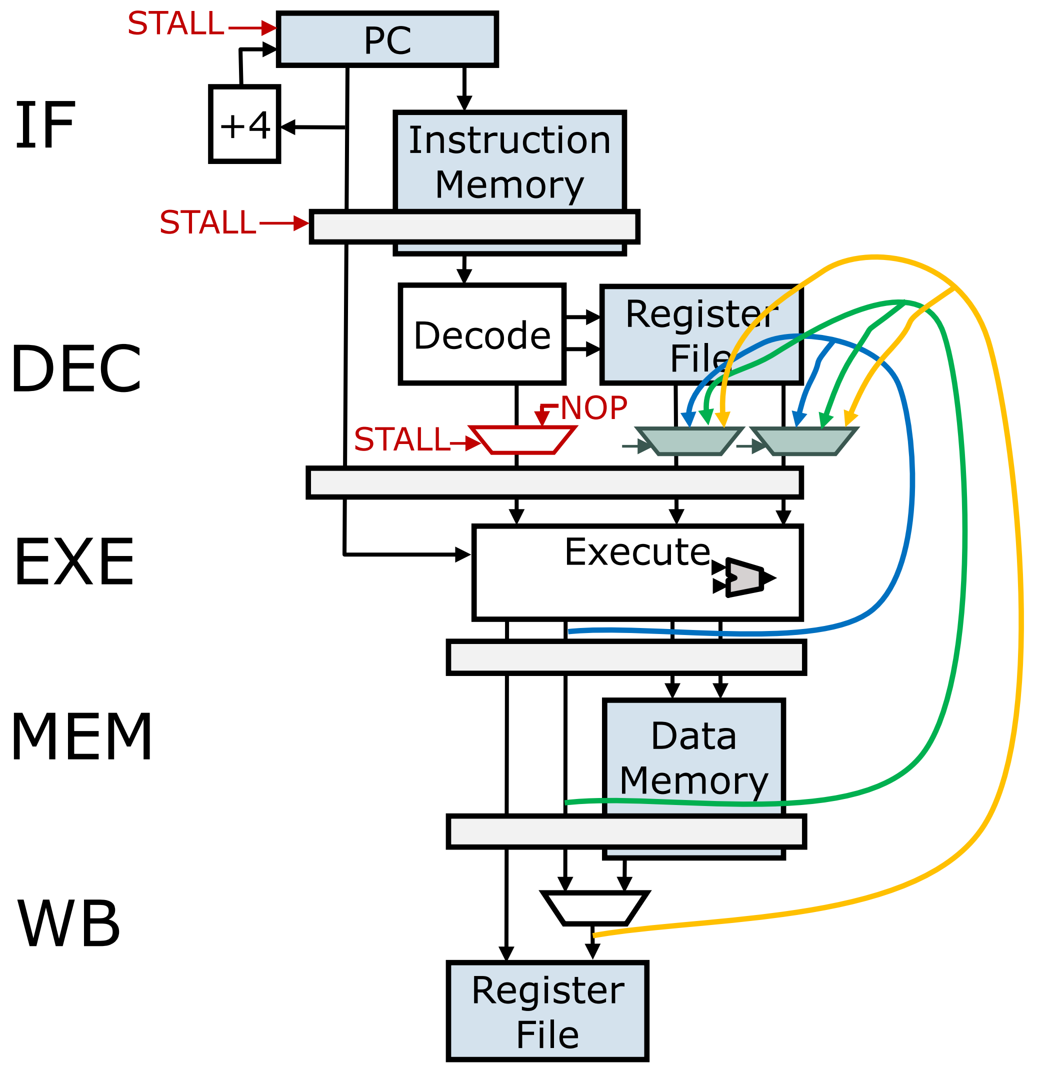
```
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

- 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 matches source register of instruction in DEC

Resolving Hazards

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

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

- Strategy 3: Speculate (Control hazard)
  - Guess a value and continue executing anyway
  - Two cases can happen
    - Correct Guess → do nothing 🙂
    - Wrong Guess → kill & restart with correct value 🙁

Resolving Control Hazards with Speculation

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

```
loop:      addi x12, x12, -1
           sub x14, x15, x16
PC    ➡    bne x12, x0, loop
PC+4  ➡    and x16, x17, x18
           xor x19, x20, x21
           …
```

```
for (int i=100; i>=0; i--){
   …
}
```

Resolving Control Hazards with Speculation

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

- Assume nextPC = PC+4

```
loop: addi x12, x11, -1
      sub x14, x15, x16
      bne x12, 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				
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 (**x12==x0**)

Resolving Control Hazards with Speculation

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

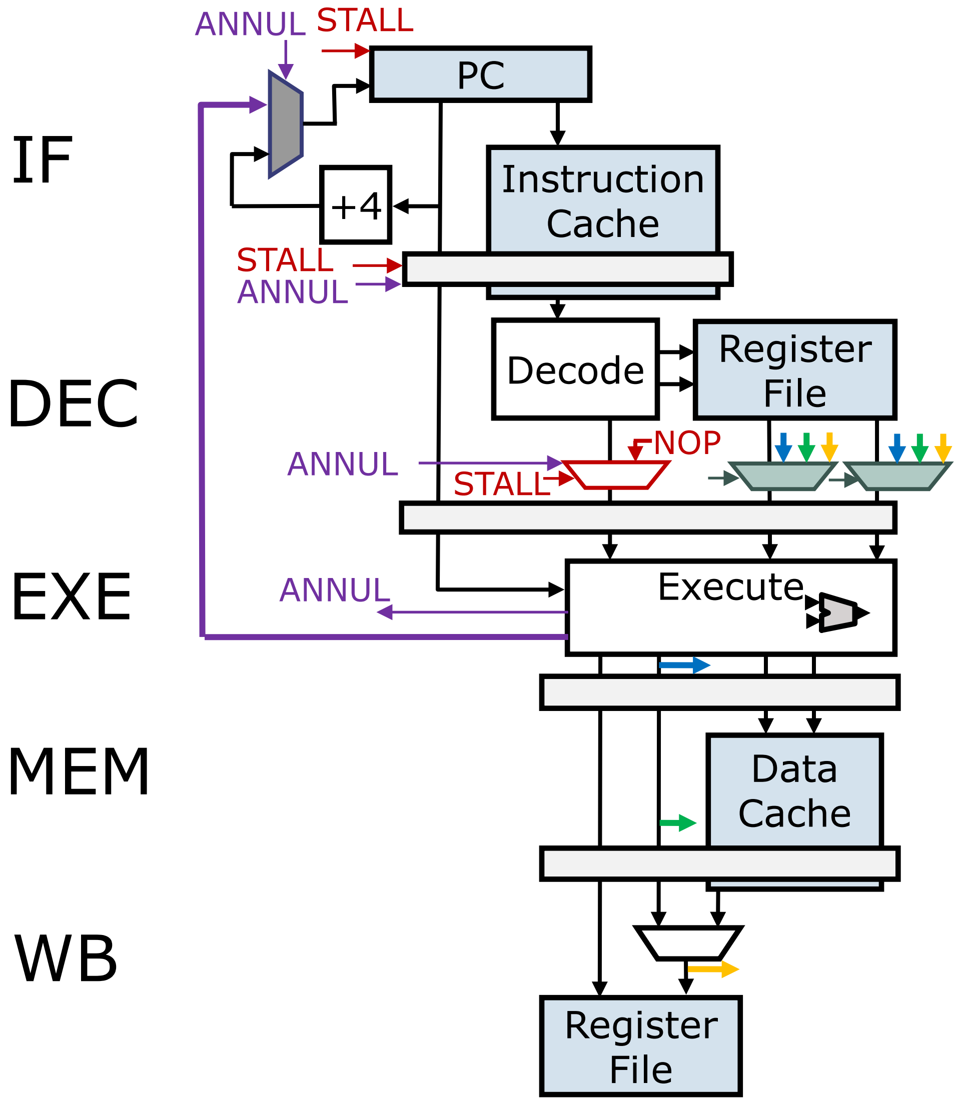
```
loop: addi x12, x11, -1
      sub x14, x15, x16
      bne x12, x0, loop
      and x16, x17, x18
      xor x19, x20, x21
      …
```

	1	2	3	4	5	6	7	8	9
IF	addi	sub	bne	and	xor	addi	sub	bne	and
DEC		addi	sub	bne	and	**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, kill and & xor and restart fetching at loop(addi)

Speculation Logic



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

Summary of solutions to hazards

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

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

$$\frac{\text{Instruction}}{\text{Cycle}}$$ Microarchitecture

Summary

- Processor state
  - Registers (including PC)
  - Memory
- Instruction set – means of updating state
  - Compute
  - Memory access
  - Control
- Basic implementation: single-cycle RISC-V processor
- Pipelining boosts throughput, but introduces hazards
  - Solutions to hazards: stall, bypass, and speculate