

# Welcome to ICS(2)

# Outline

---

- Teaching staffs
- Grading

# Instructor

---



臧斌宇

Email: [byzang@fudan.edu.cn](mailto:byzang@fudan.edu.cn)

Office phone: 13917124245

陈榕

Email: [rongchen@sjtu.edu.cn](mailto:rongchen@sjtu.edu.cn)

Office phone: 13616861826



# Teaching Assistants

感谢 侯君

刘宇宸 (LYC)

Email: [liuyuchen@sjtu.edu.cn](mailto:liuyuchen@sjtu.edu.cn)

闵师达 (MSD)

Email: [minshistar@163.com](mailto:minshistar@163.com)

成立 (CL)

Email: [lewischeng0204@gmail.com](mailto:lewischeng0204@gmail.com)

曾福山 (ZFS)

Email: [chafee.hello@gmail.com<sup>4</sup>](mailto:chafee.hello@gmail.com)

# Grading

---

- Exams(60%)
  - Mid term (20%)
  - Final (40%)
  - All exams are open books/open notes.
- Labs (35%)
  - 4 labs (7-10% each)
- Home work(5%)

# Context

---

## ICS-1: basic

- ✓ data representations
- ✓ machine-level rep.
- ✓ y86
- ✓ linking
- ✓ memory allocation
- ✓ memory hierarchy

## ICS-2: advance

- processor architecture
- program optimization
- exceptional control flow
- system-level I/O & networking
- concurrent programming
- virtual memory

# Processor Architecture

# Topics

---

- Review y86 instruction set architecture
- Logic design
- Hardware Control Language HCL
- Suggested Reading: 4.1, 4.2

# Goal

---

- Understand basic computer organization
  - Instruction set architecture
- Deeply explore the CPU working mechanism
  - How the instruction is executed
- Help you programming
  - Fully understand how computer is organized and works will help you write more stable and efficient code

# Instruction Set Architecture #1

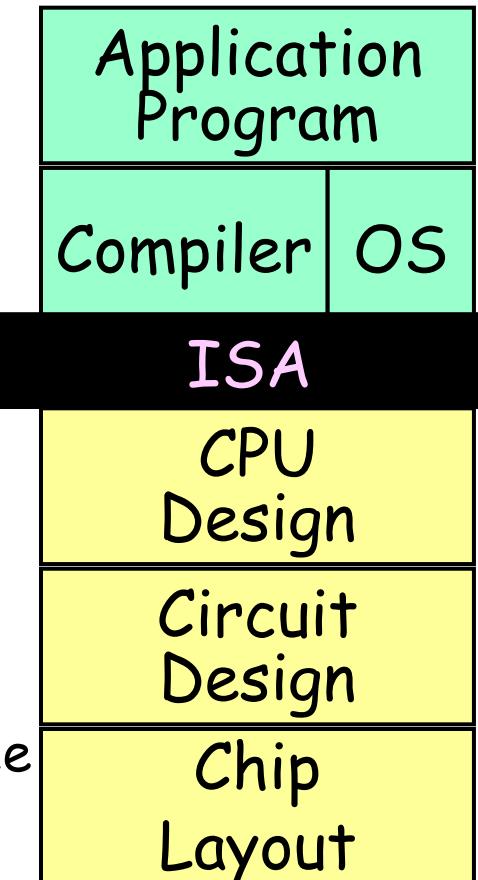
---

- What is it ?
  - Assemble Language Abstraction
  - Machine Language Abstraction
- What does it provide?
  - An abstraction of the real computer, hide the details of implementation
    - The syntax of computer instructions
    - The semantics of instructions
    - The execution model
    - Programmer-visible computer status

# Instruction Set Architecture #2

---

- Assembly Language View
  - Processor state
    - Registers, memory, ...
  - Instructions
    - addl, movl, leal, ...
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



# Instruction Set Architecture #3

---

- ISA define the processor family
  - Two main kind: RISC and CISC
    - RISC: SPARC, MIPS, PowerPC, ARM
    - CISC: X86 (or called IA32)
- Under same ISA, there are many different processors
  - From different manufacturers
    - X86 from Intel, AMD and VIA
  - Different models
    - 8086, 80386, Pentium, Pentium 4

# Y86 Processor State

---

- Program Registers
  - Same 8 as with IA32. Each 32 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - OF: Overflow ZF: Zero SF: Negative
- Program Counter
  - Indicates address of instruction
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

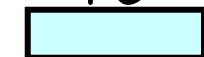
Program registers

%eax	%esi
%ecx	%edi
%edx	%esp
%ebx	%ebp

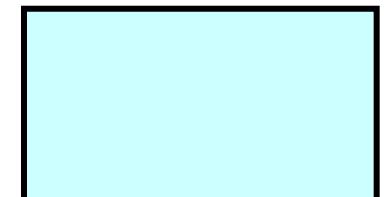
Condition codes



PC



Memory



# Y86 Instructions

- Format (P259)
  - 1--6 bytes of info. read from memory
    - Can determine inst. length from 1st byte
    - Not as many inst. types, and simpler encoding than with IA32
  - Each accesses and modifies some part(s) of the program state

length		Byte_0	Byte_1	Byte_2	Byte_3	Byte_4	Byte_5
1	nop	0	0				
1	halt	1	0				
2	rrmovl rA,rB	2	0	rA	rB		
6	irrmovl V,rB	3	0	rA	rB	V	
6	rmmovl rA,D(rB)	4	0	rA	rB	Dest	
6	mrmovl D(rB),rA	5	0	rA	rB	Dest	
2	opl rA,rB	6	fn	rA	rB		
2	addl rA,rB	6	0	rA	rB		
2	subl rA,rB	6	1	rA	rB		
2	andl rA,rB	6	2	rA	rB		
2	xorl rA,rB	6	3	rA	rB		
5	jxx Dest	7	fn	Dest			
5	jmp Dest	7	0	Dest			
5	jle Dest	7	1	Dest			
5	jl Dest	7	2	Dest			
5	je Dest	7	3	Dest			
5	jne Dest	7	4	Dest			
5	jge Dest	7	5	Dest			
5	jg Dest	7	6	Dest			
5	call Dest	8	0	Dest			
1	ret	9	0				
2	pushl rA	A	0	rA	8		
2	popl rA	B	0	rA	8		

# Encoding Registers

---

- Each register has 4-bit ID

%eax	0
%ecx	1
%edx	2
%ebx	3

%esi	6
%edi	7
%esp	4
%ebp	5

- Same encoding as in IA32, but IA32 using only 3-bit ID
- Register ID “F” indicates “no register”
  - Will use this in our hardware design in multiple places

# Logical Design & HCL

# Logic Design

---

- Digital circuit
  - What is digital circuit?
  - Know what a CPU will base on?
- Hardware Control Language (HCL)
  - A simple and functional language to describe our CPU implementation
  - Syntax like C

# Category of Circuit

---

- Analog Circuit
  - Use all the range of Signal
  - Most part is amplifier
  - Hard to model and automatic design
  - Use transistor and capacitance as basis
  - We will not discuss it here

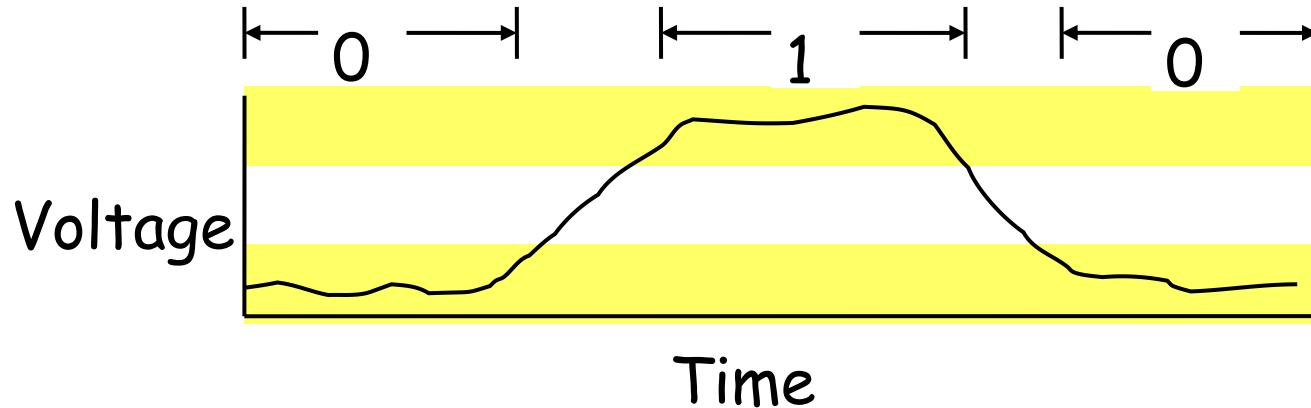
# Category of Circuit

---

- Digital Circuit
  - Has only two values, 0 and 1
  - Easy to model and design
  - Use true table and other tools to analyze
  - Use gate as the basis
  - The voltage of 1 is differ in different kind circuit.
    - E.g. TTL circuit using 5 voltage as 1

# Digital Signals

---



- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

# Overview of Logic Design

---

- Fundamental Hardware Requirements
  - Communication
    - How to get values from one place to another
  - Computation
  - Storage (Memory)
  - Clock Signal

# Overview of Logic Design

---

- Bits are Our Friends
  - Everything expressed in terms of values 0 and 1
  - Communication
    - Low or high voltage on wire
  - Computation
    - Compute Boolean functions
  - Storage
  - Clock Signal

# Category of Digital Circuit

---

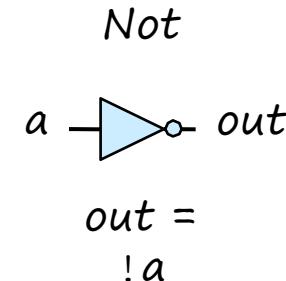
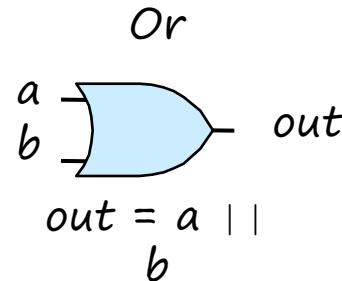
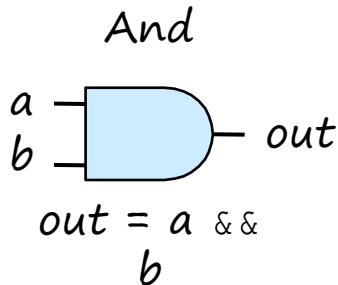
- Combinational Circuit
  - Without memory. So the circuit can't have state.  
Any same input will get the same output at any time.
  - Needn't clock signal
  - Typical application: ALU

# Category of Digital Circuit

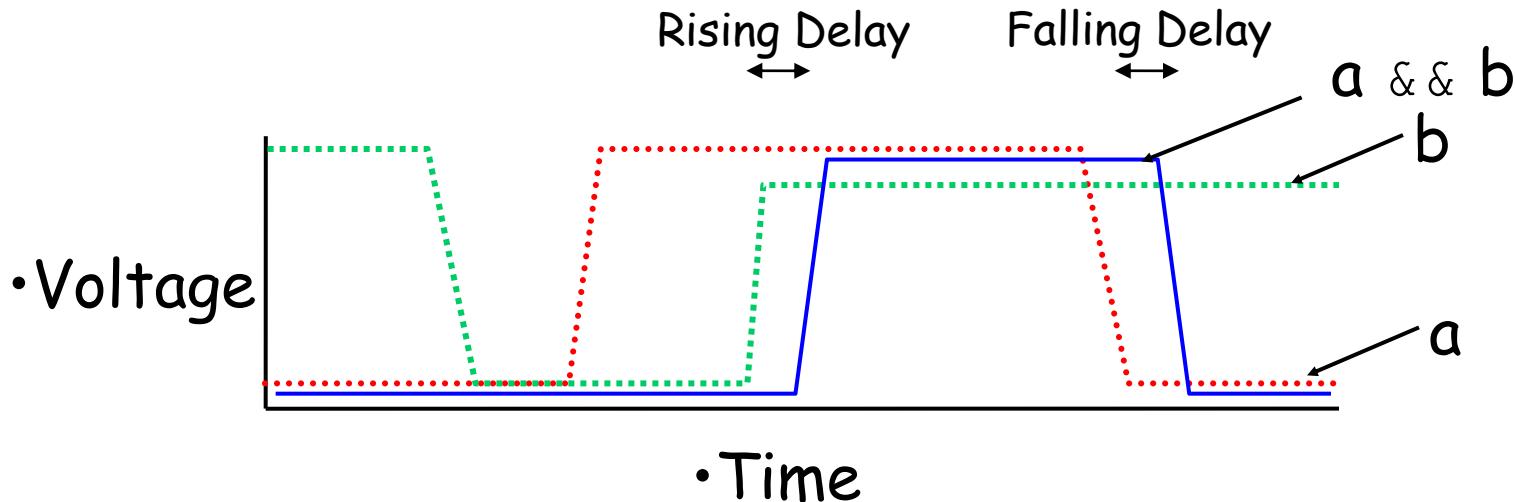
---

- Sequential Circuit
  - = Combinational circuit + memory and clock signal
  - Have state. Two same inputs may not generate the same output.
  - Use clock signal to control the run of circuit.
  - Typical application: CPU

# Computing with Logic Gates

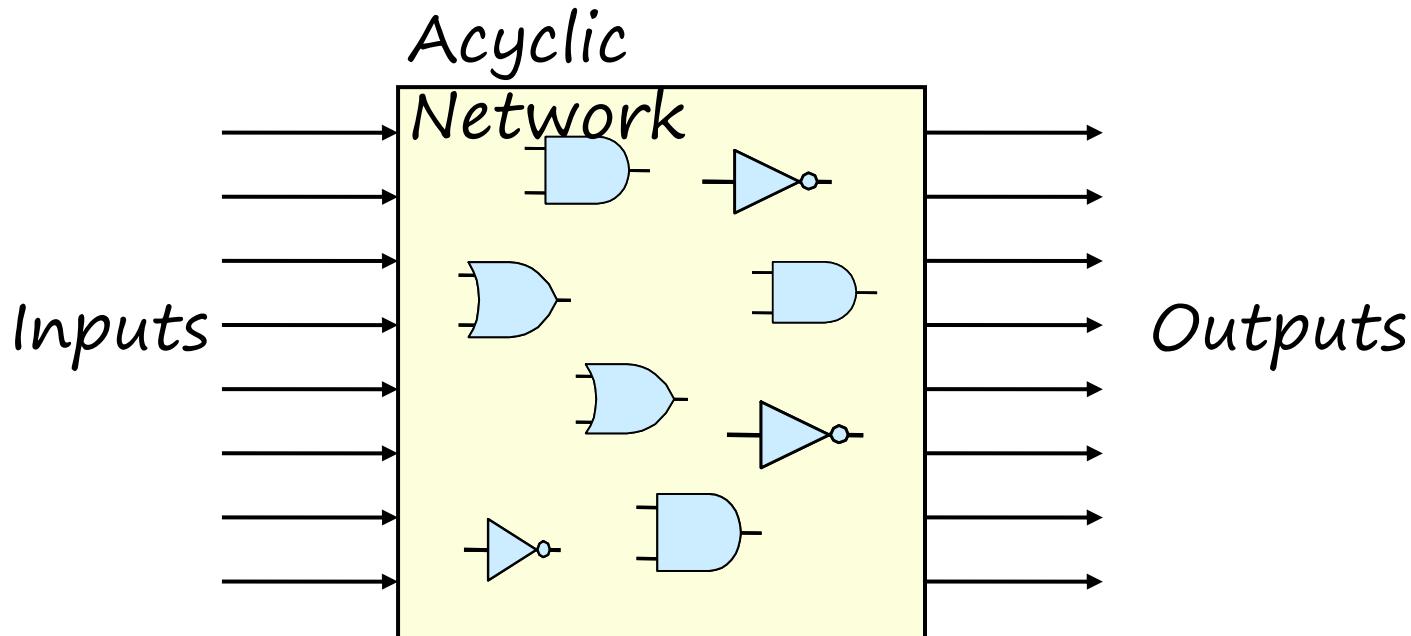


- Outputs are Boolean functions of inputs
- Not an assignment operation, just give the circuit a name
- Respond continuously to changes in inputs
  - With some, small delay



# Combinational Circuits

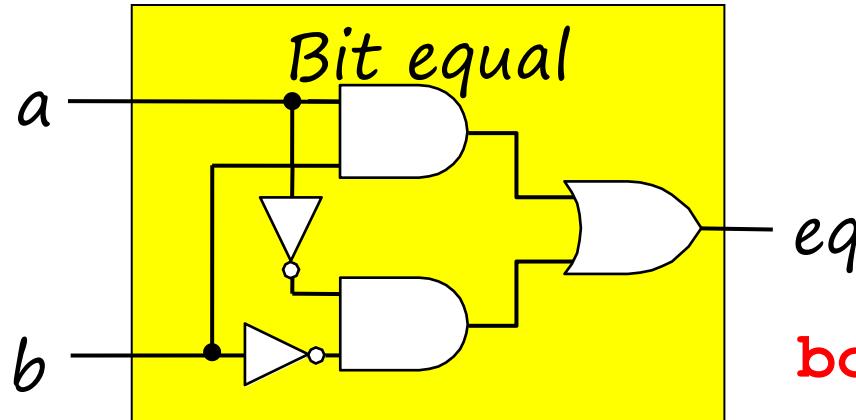
---



- Acyclic Network of Logic Gates
  - Continuously responds to changes on inputs
  - Outputs become (after some delay) Boolean functions of inputs

# Bit Equality

---

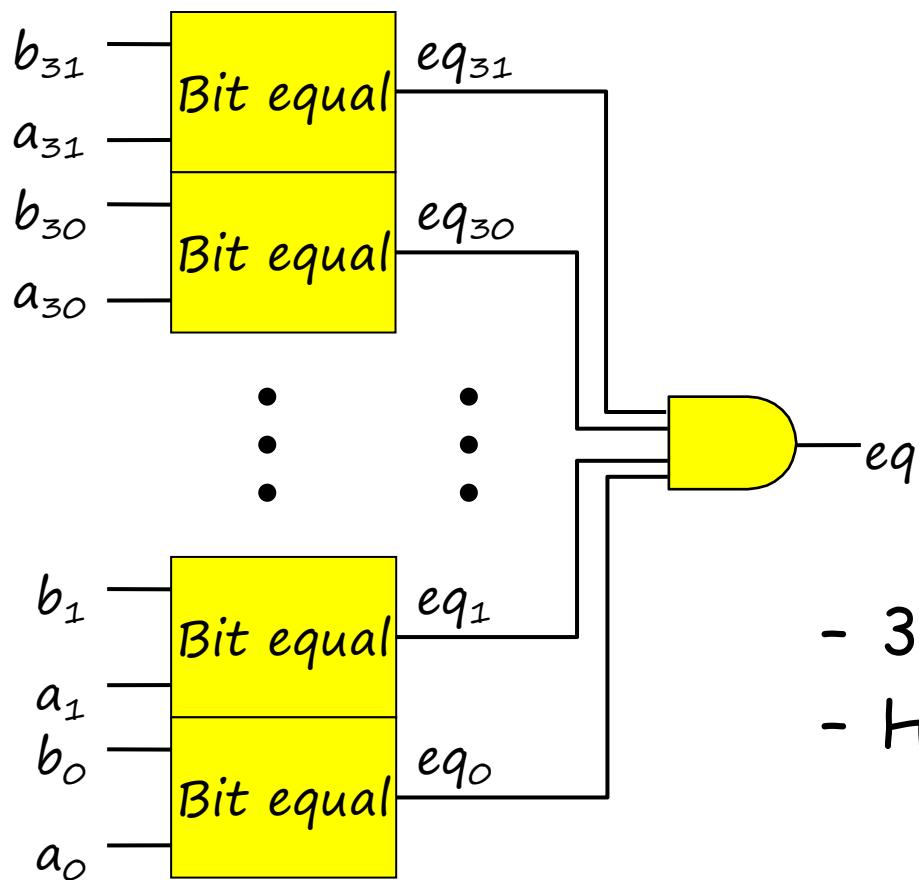


HCL Expression

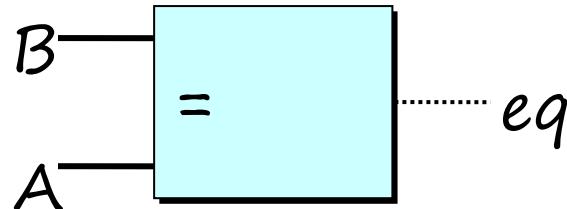
`bool eq = (a&&b) || (!a&&!b)`

- Generate 1 if  $a$  and  $b$  are equal
- Hardware Control Language (HCL)
  - Very simple hardware description language
    - Boolean operations have syntax similar to C logical operations
  - We'll use it to describe control logic for processors

# Word Equality



Word-Level Representation



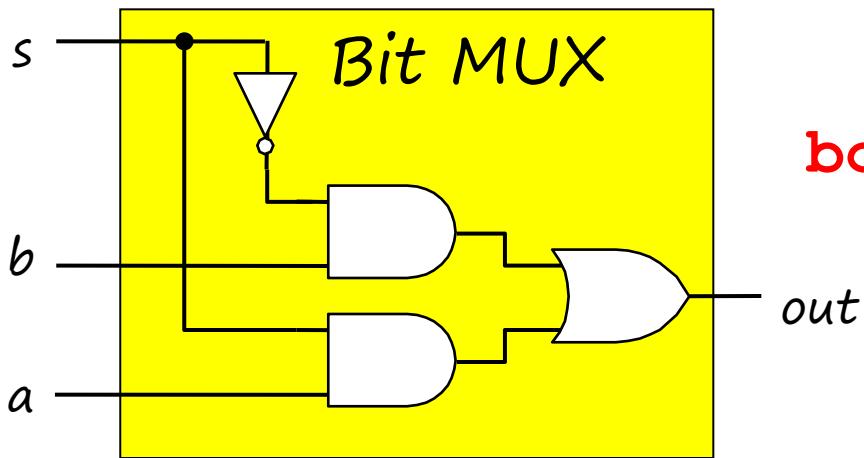
HCL Representation

**bool Eq = (A == B)**

- 32-bit word size
- HCL representation
  - Equality operation
  - Generates Boolean value

# Bit-Level Multiplexor

---

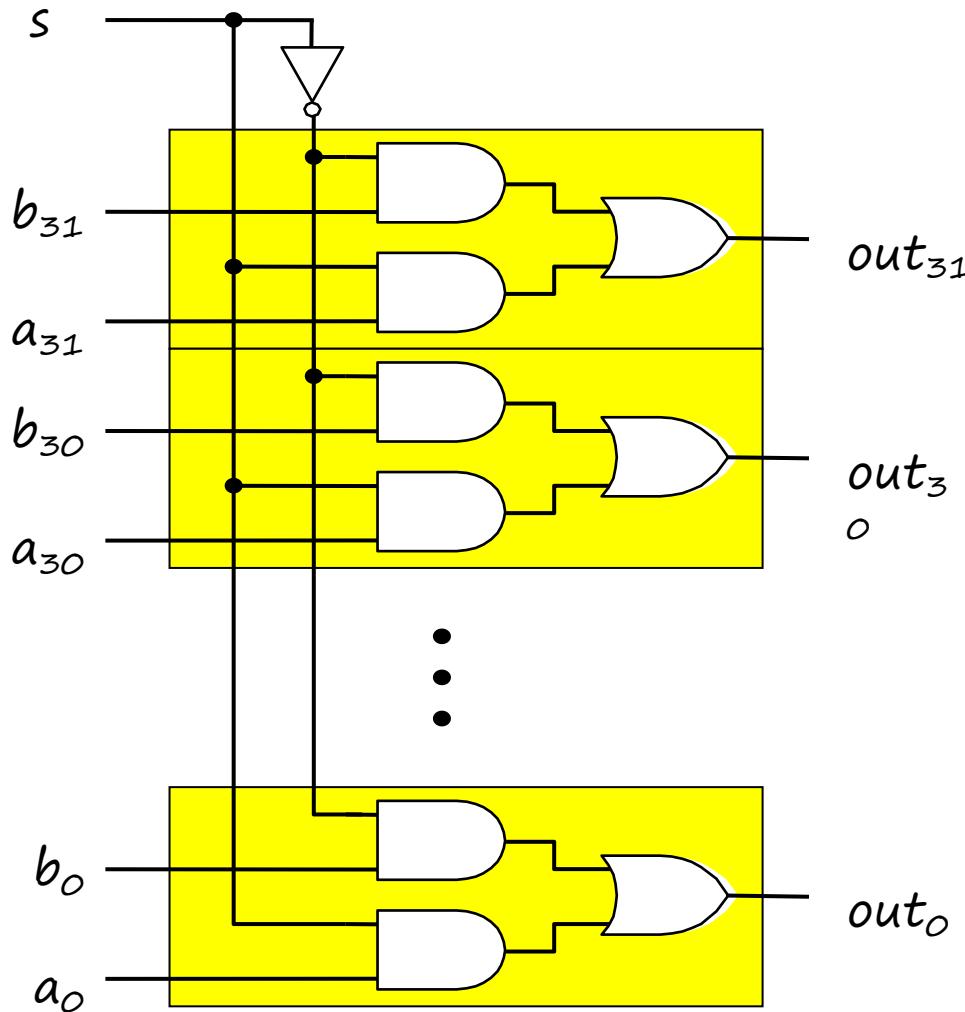


HCL Expression

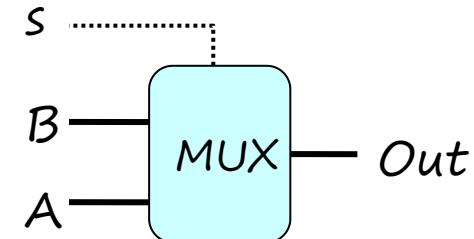
`bool out = (s&&a) || (!s&&b)`

- Control signal  $s$
- Data signals  $a$  and  $b$
- Output  $a$  when  $s=1$ ,  $b$  when  $s=0$
- Its name: MUX
- Usage: Select one signal from a couple of signals

# Word Multiplexor



Word-Level Representation



HCL Representation ?

# Word Multiplexor

---

- HCL Representation
  - Select input word A or B depending on control signal s
  - HCL representation
    - Case expression
    - Series of test : value pairs (Don't require mutually)
    - Output value for first successful test

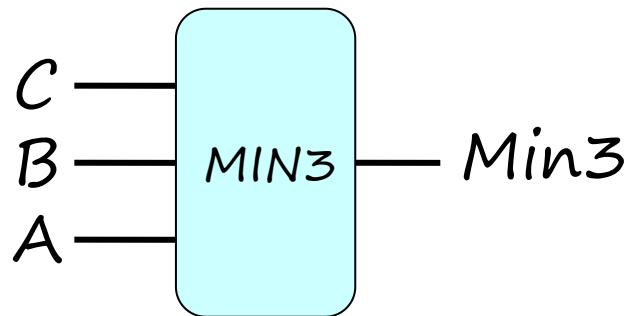
```
int Out = [
    s : A;
    1 : B;
];
```

default case

# HCL Word-Level Examples

---

Minimum of 3 Words



HCL Representation

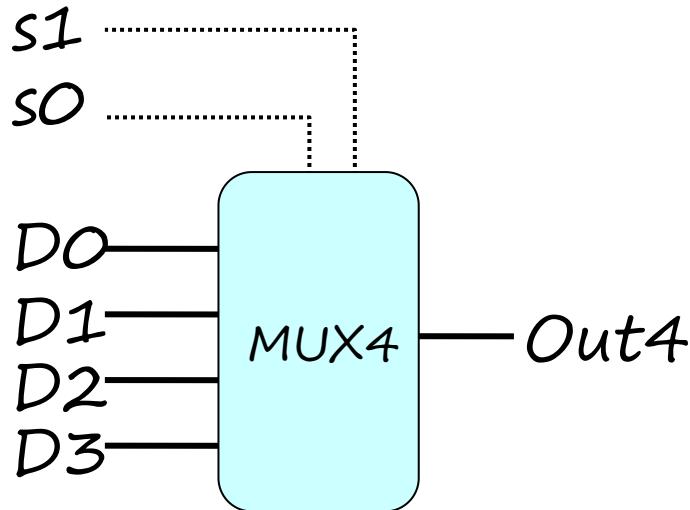
```
int Min3 = [
    A < B && A < C : A;
    B < A && B < C : B;
    1
];
;
```

- Find minimum of three input words
- HCL case expression
- Final case guarantees match

# HCL Word-Level Examples

---

4-Way Multiplexor



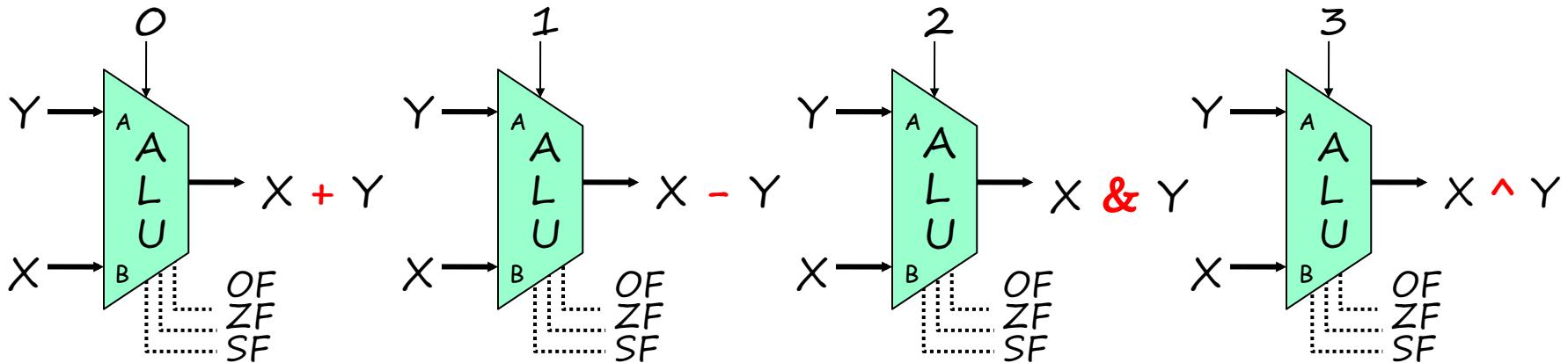
HCL Representation

```
int Out4 = [
    !s1&&!s0: D0;
    !s1        : D1;
    !s0        : D2;
    1          : D3;
];
```

- Select one of 4 inputs based on two control bits
- HCL case expression
- Simplify tests by assuming sequential matching

# Arithmetic Logic Unit

---

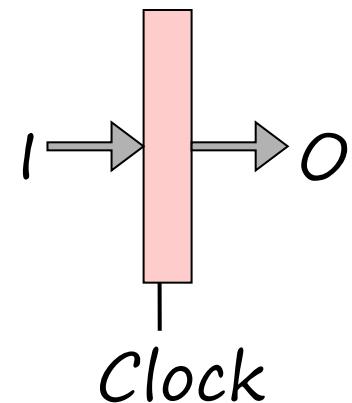


- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86
- Also computes values for condition codes
- We will use it as a basic component for our CPU

# Storage

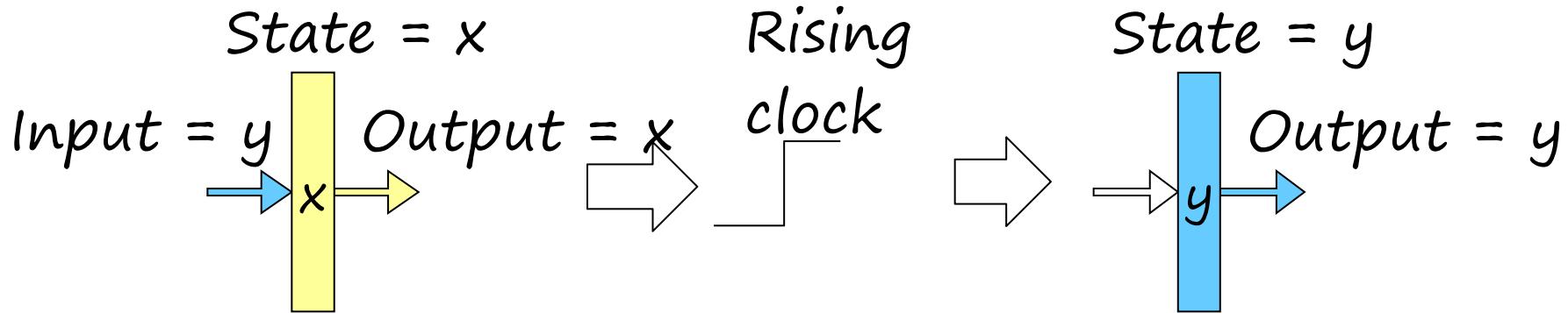
---

- Clocked Registers
  - e.g. Program Counter(PC), Condition Codes(CC)
  - Hold single words or bits
  - Loaded as clock rises
  - Not “program registers”



# Register Operation

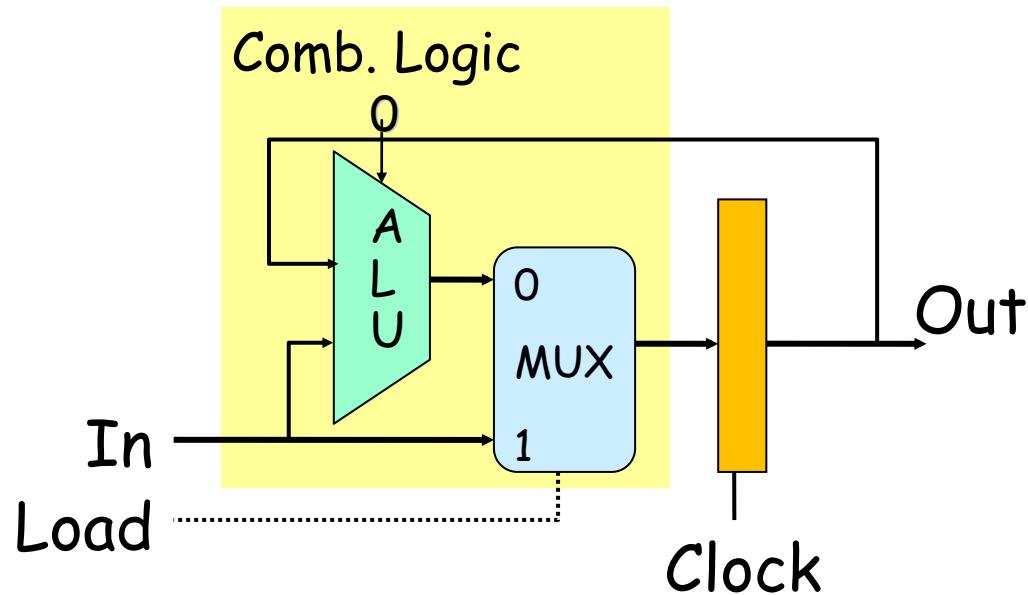
---



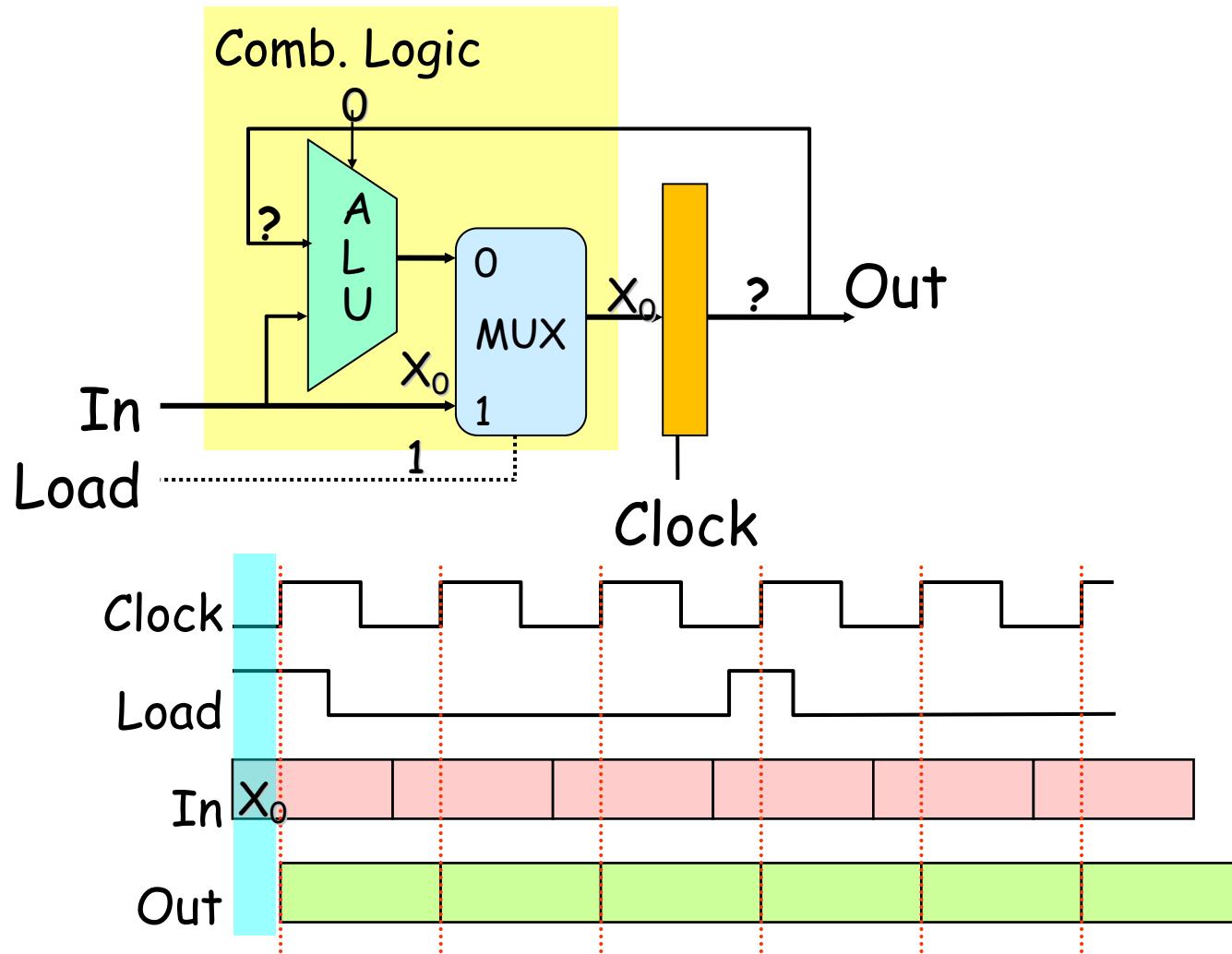
- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

# State Machine Example

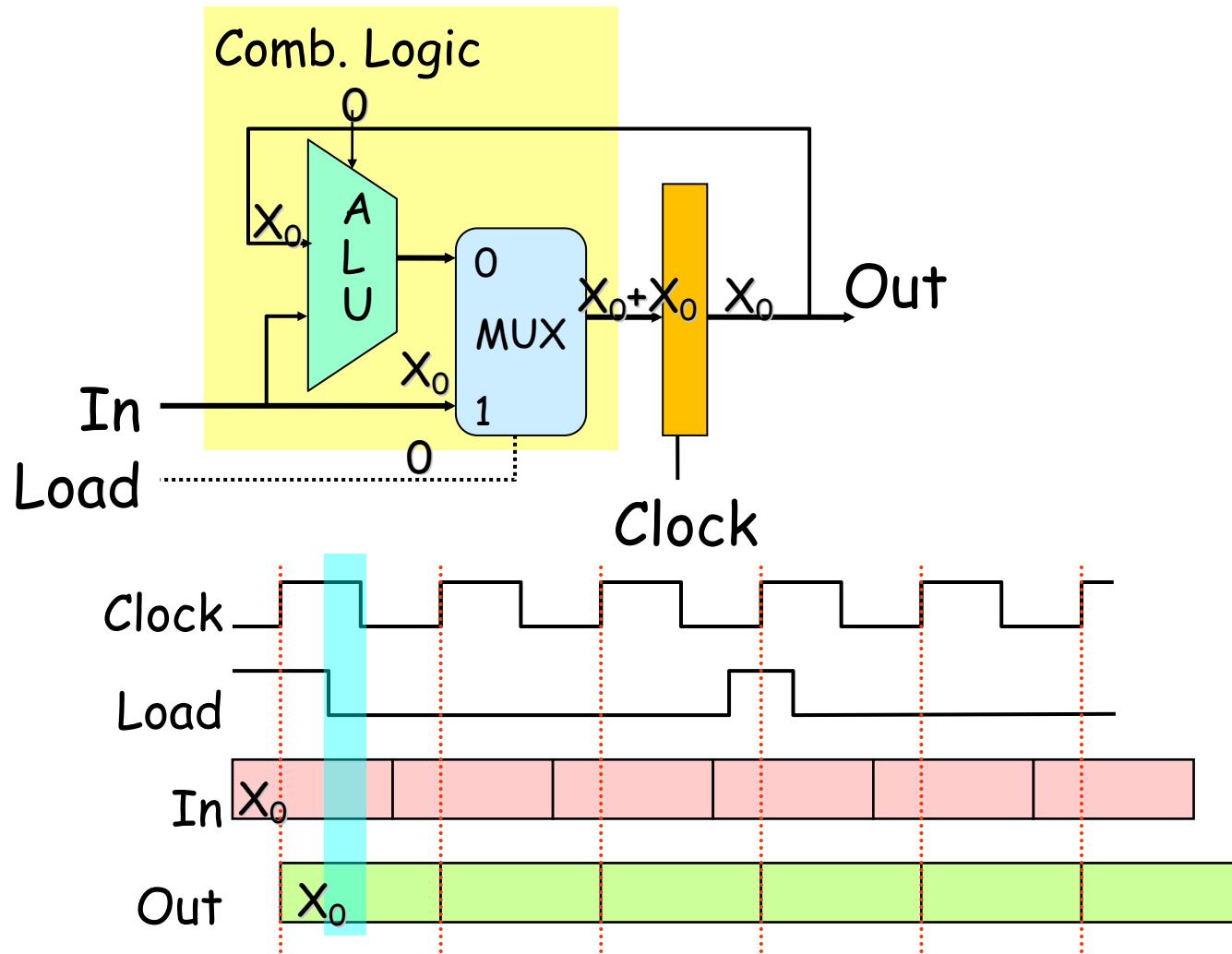
---



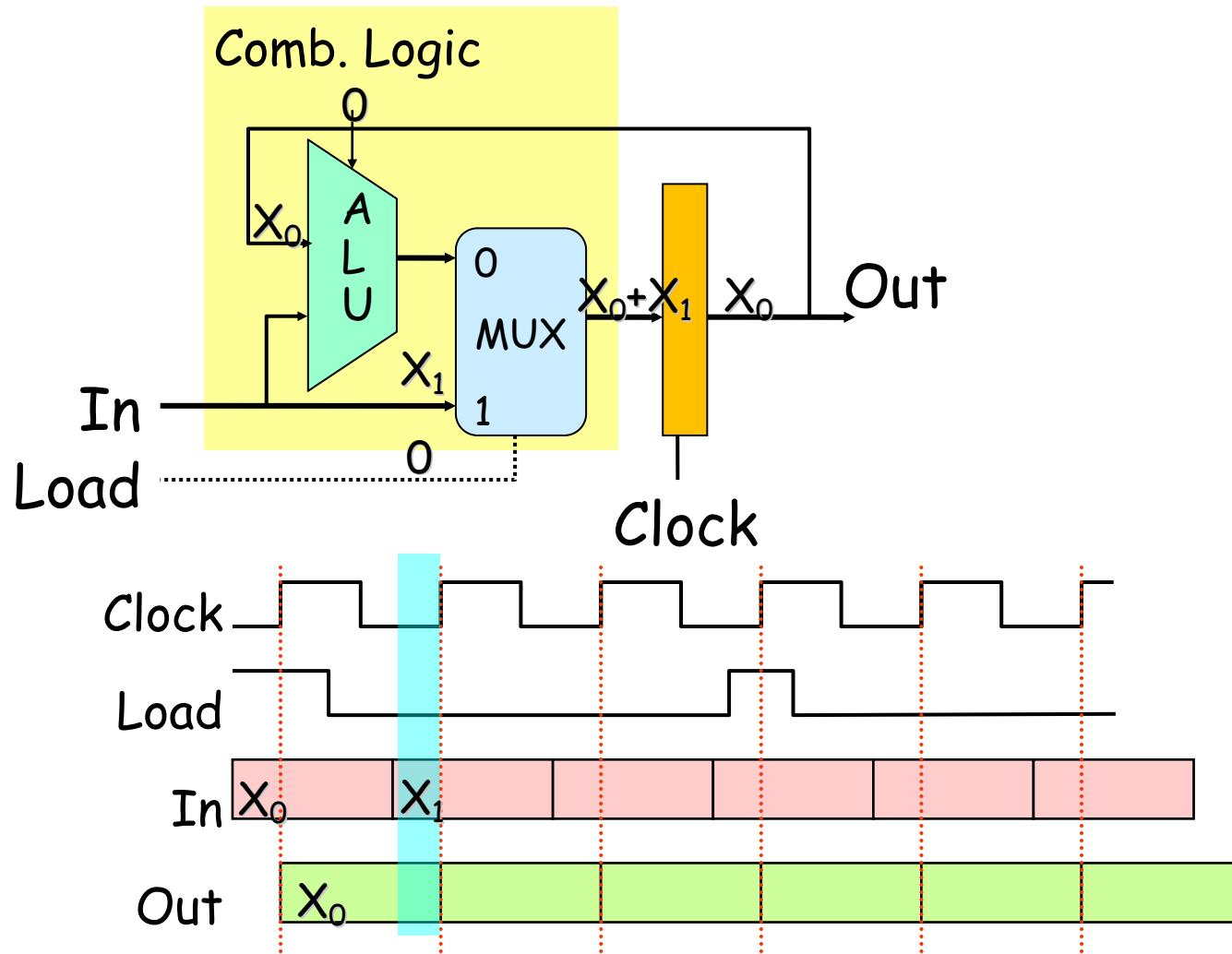
# State Machine Example



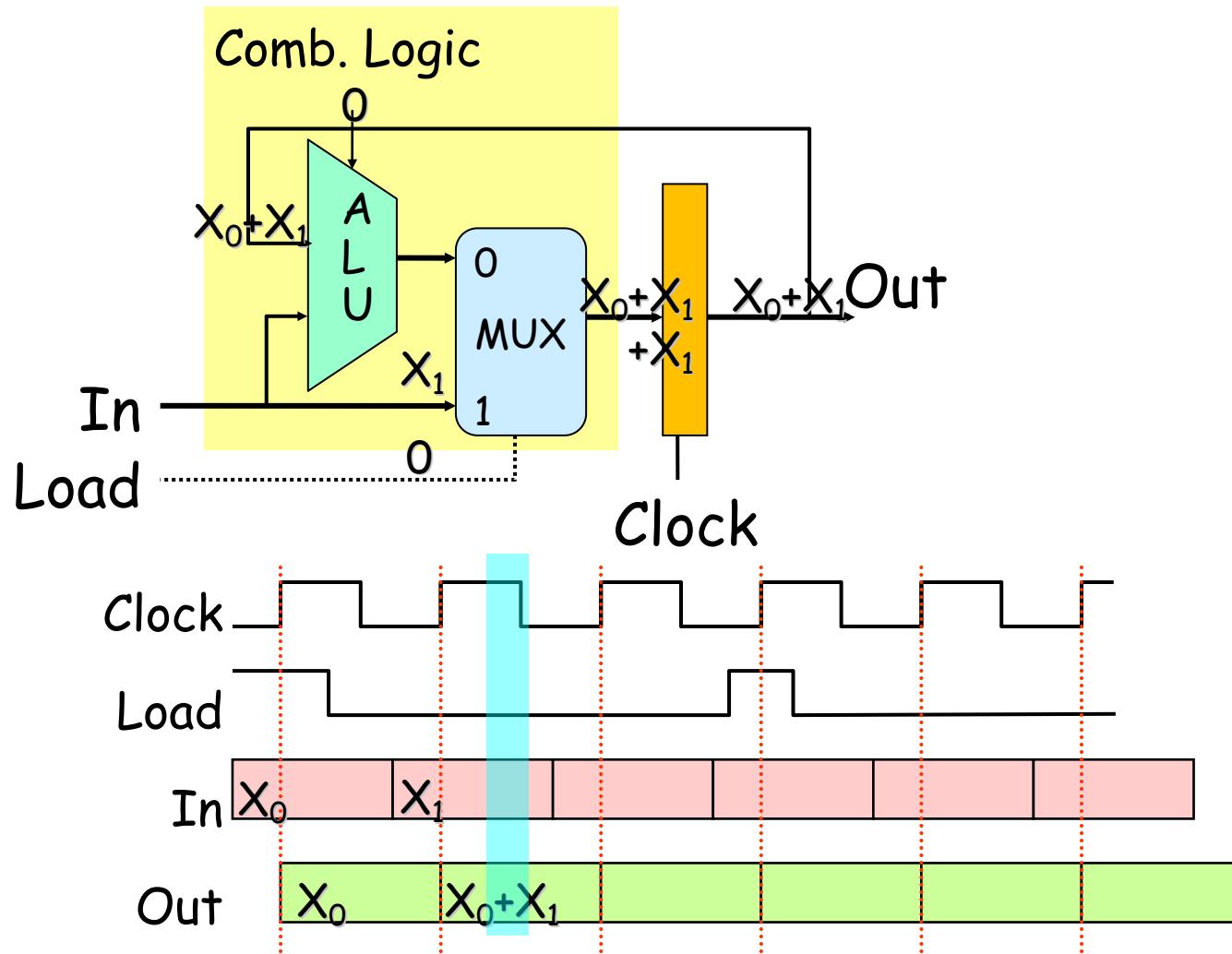
# State Machine Example



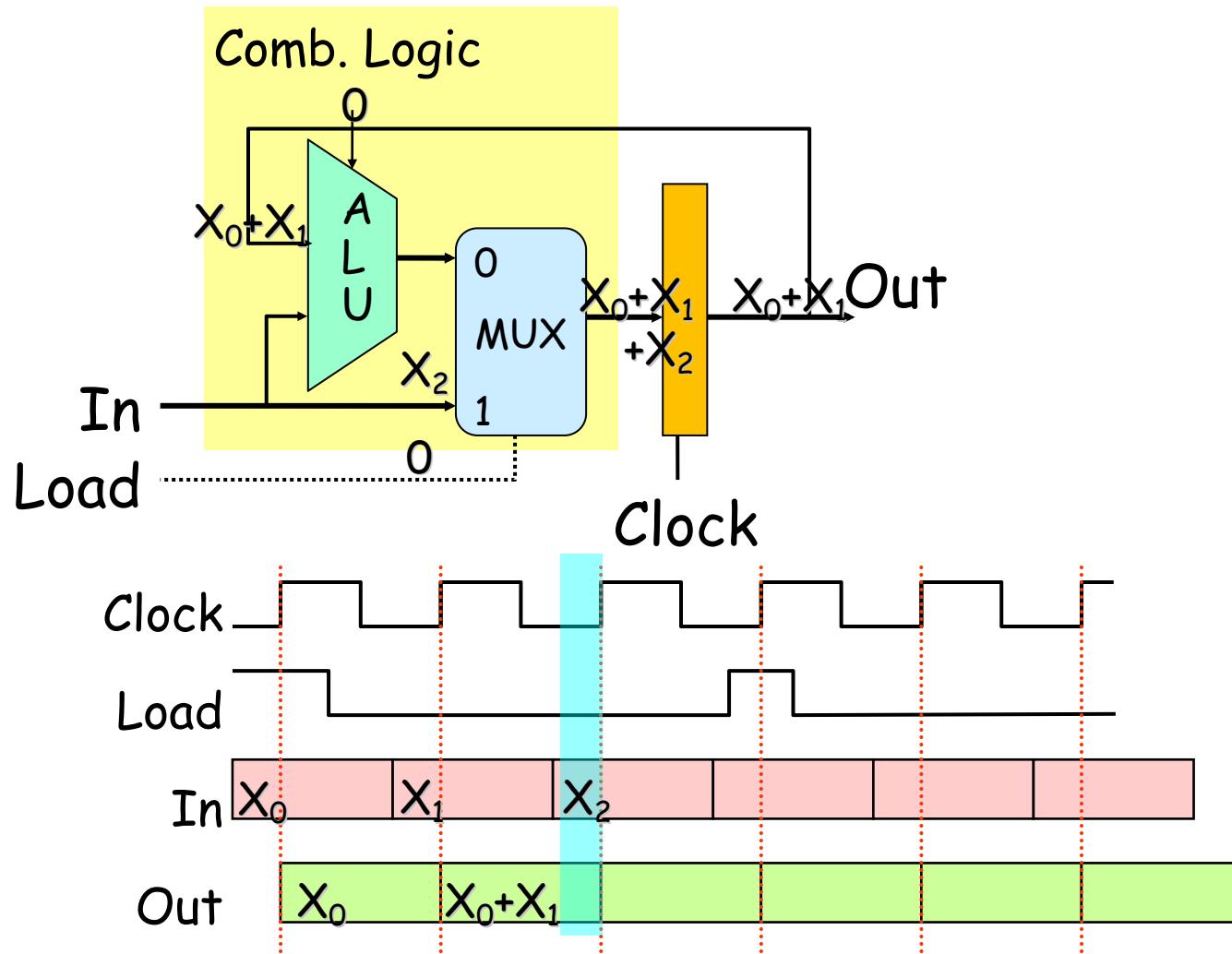
# State Machine Example



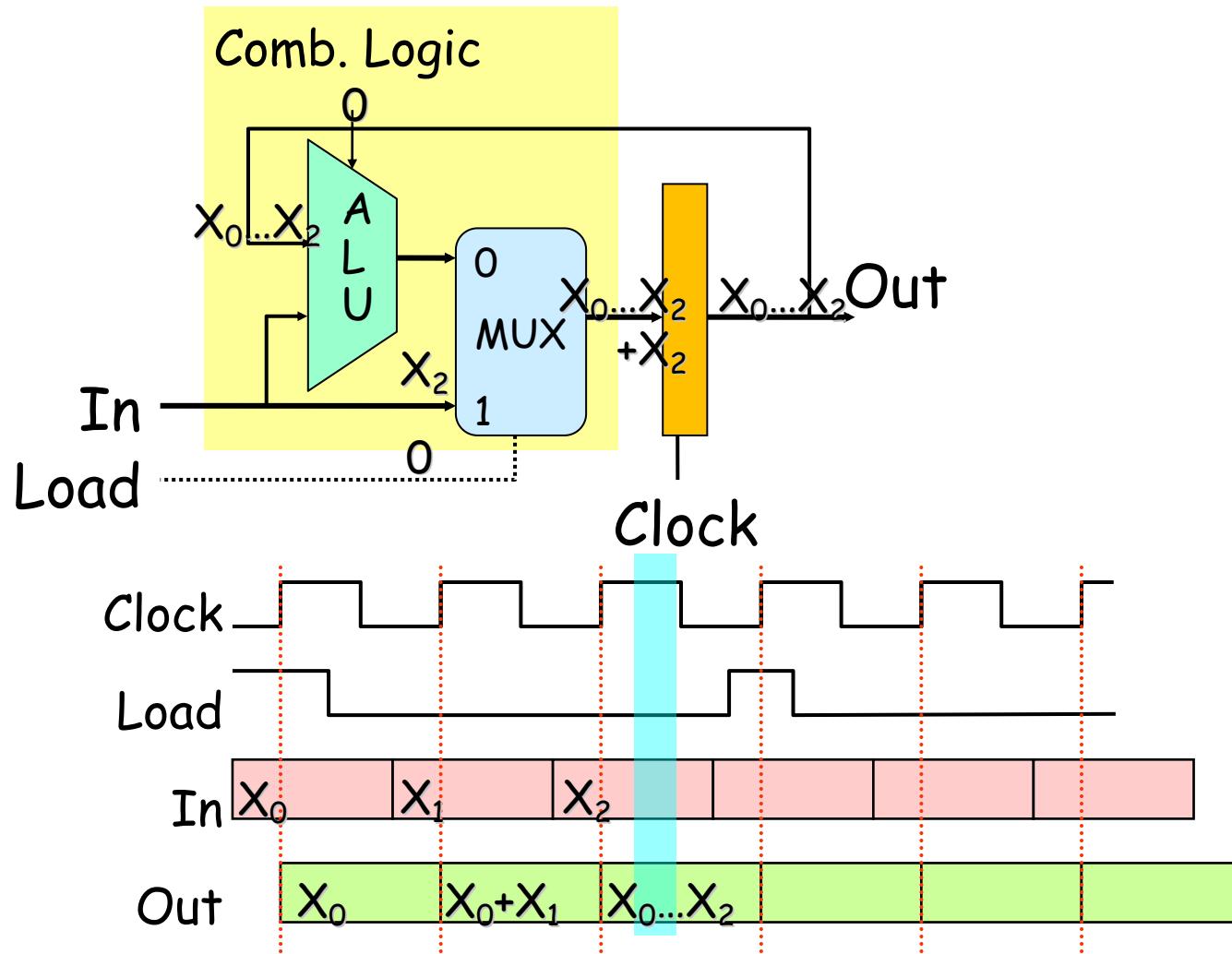
# State Machine Example



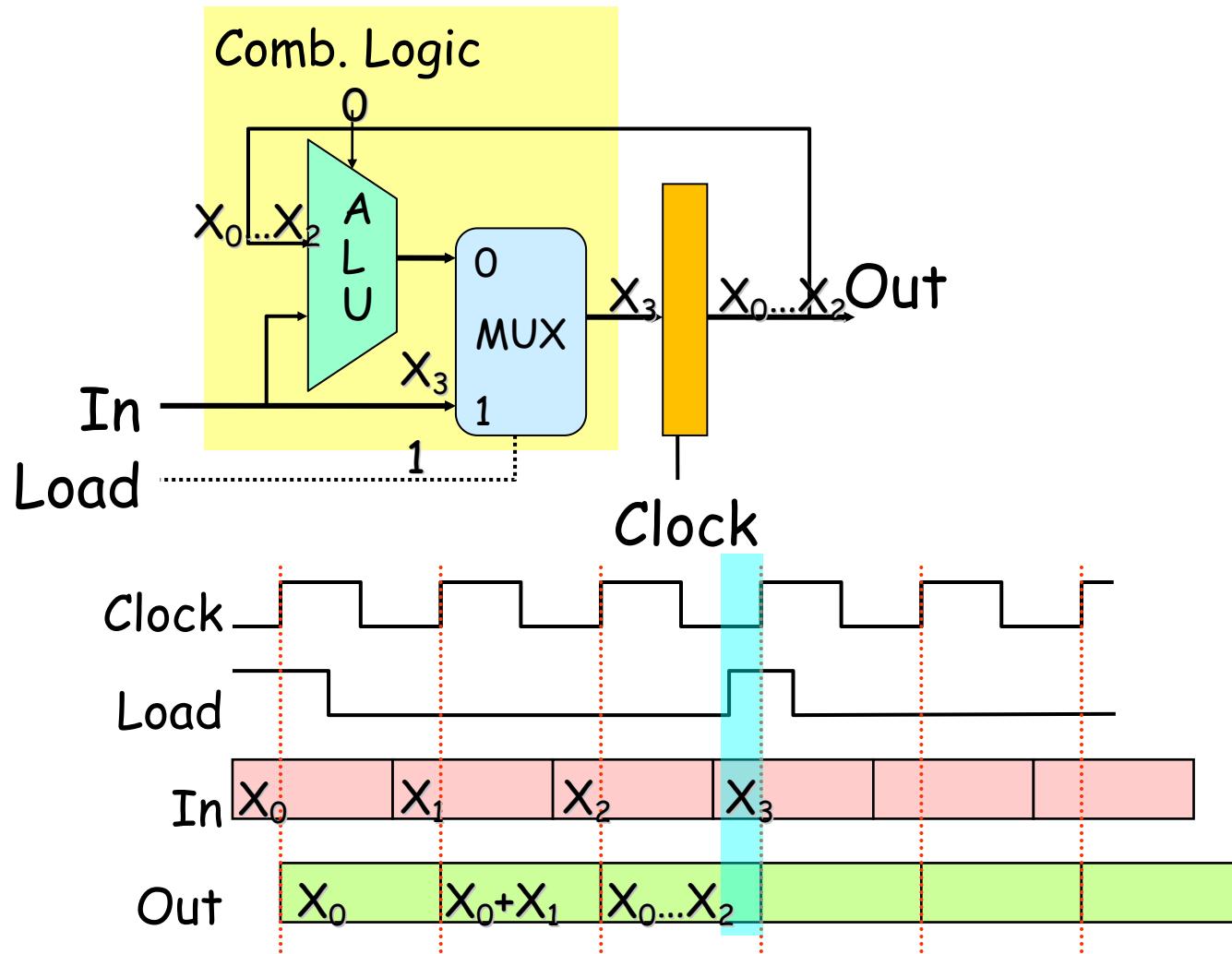
# State Machine Example



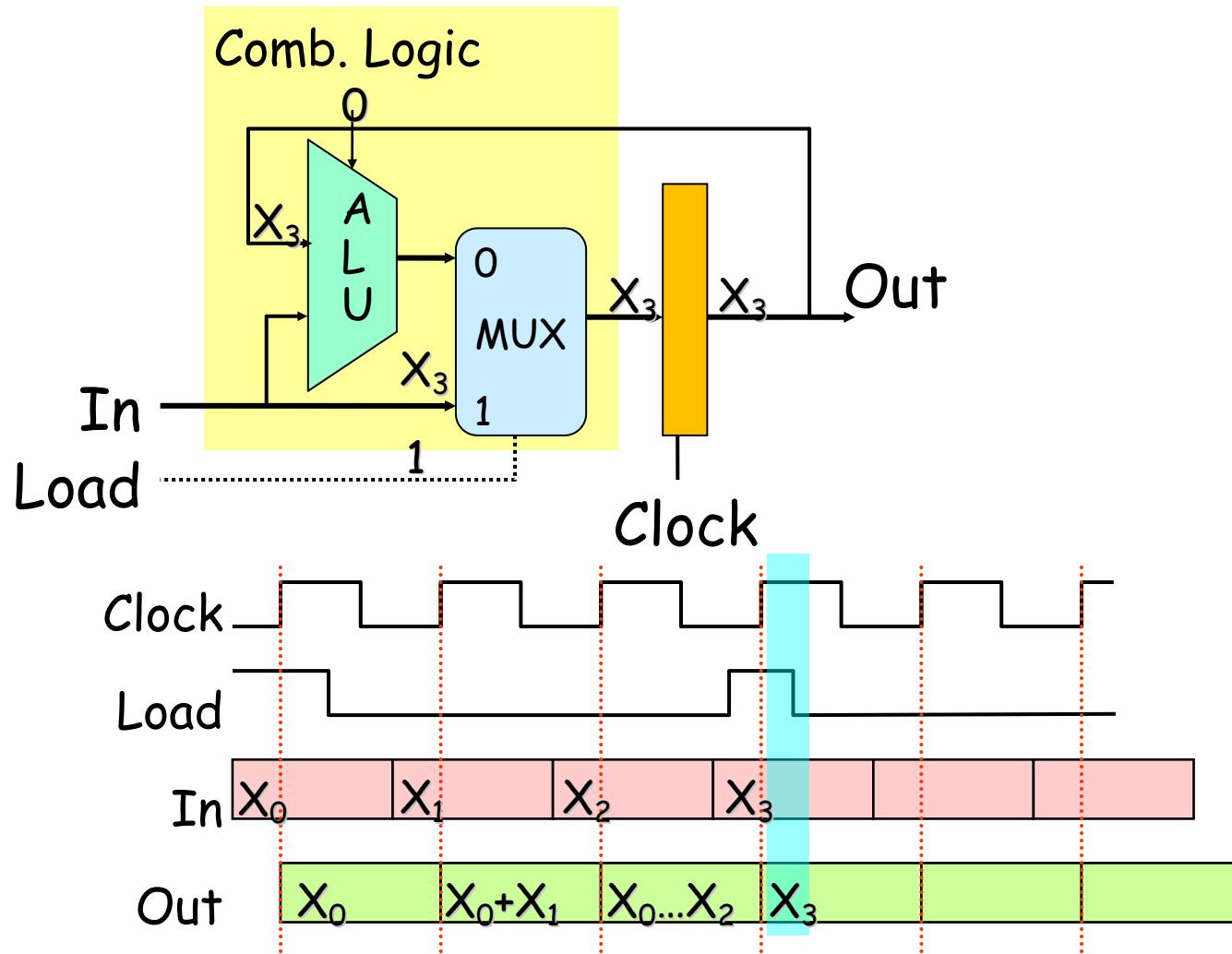
# State Machine Example



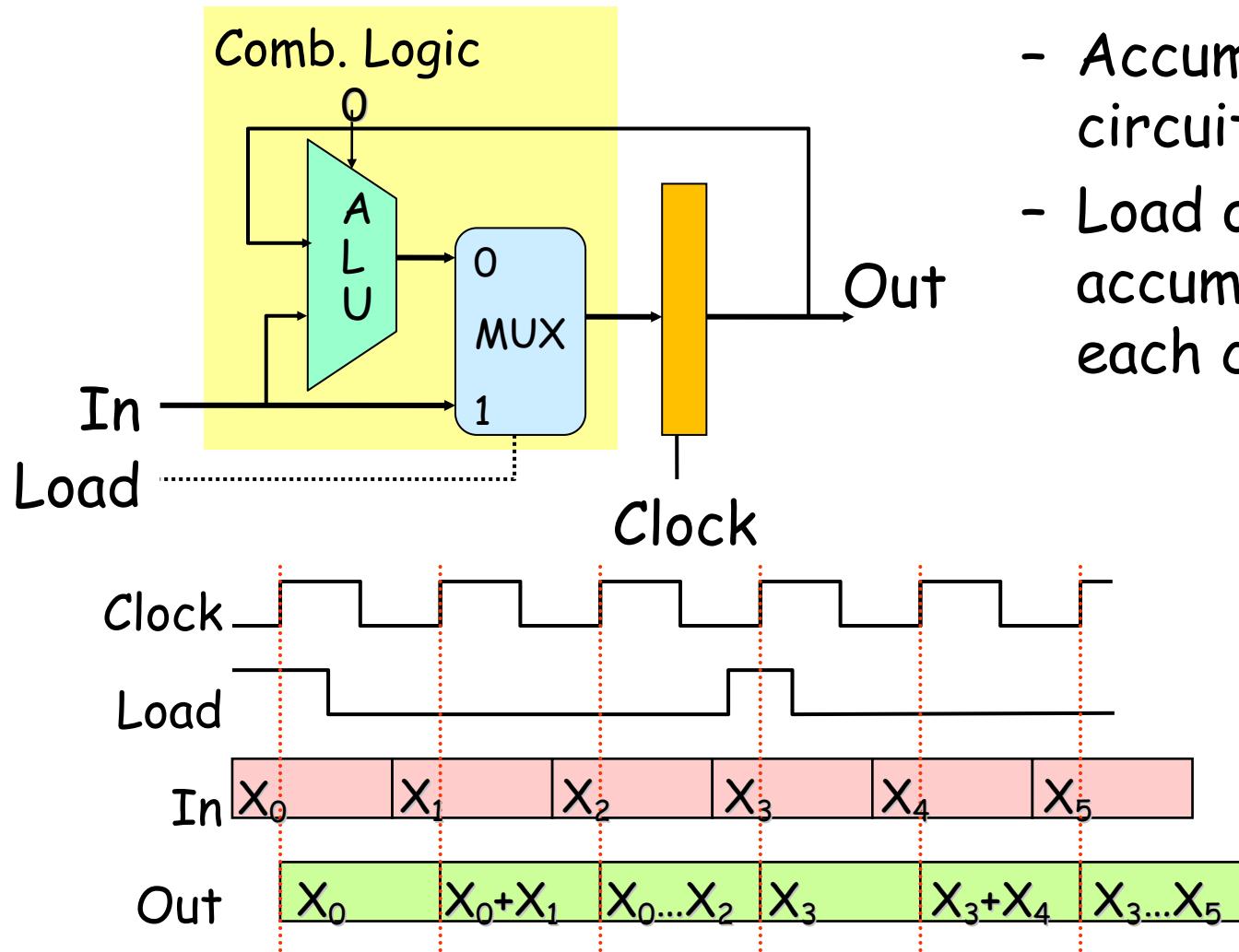
# State Machine Example



# State Machine Example



# State Machine Example



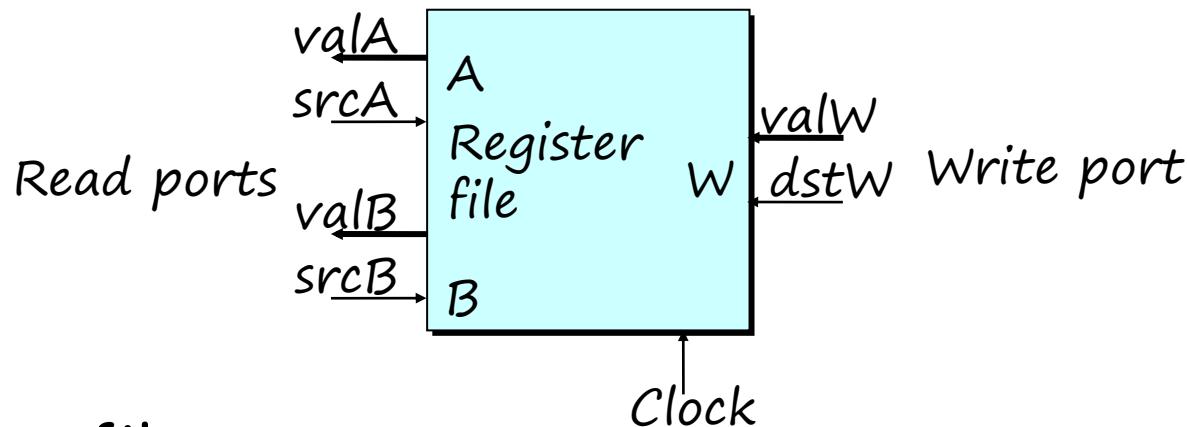
# Storage

---

- Random-access memories
  - e.g. Register File, Memory
  - Hold multiple words
    - Address input specifies which word to read or write
  - Possible multiple read or write ports
  - Read word when address input changes
  - Write word as clock rises

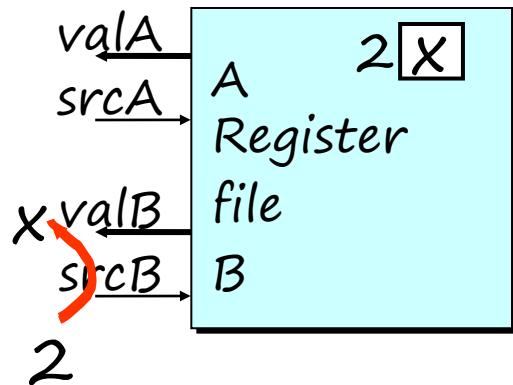
# Register File

---

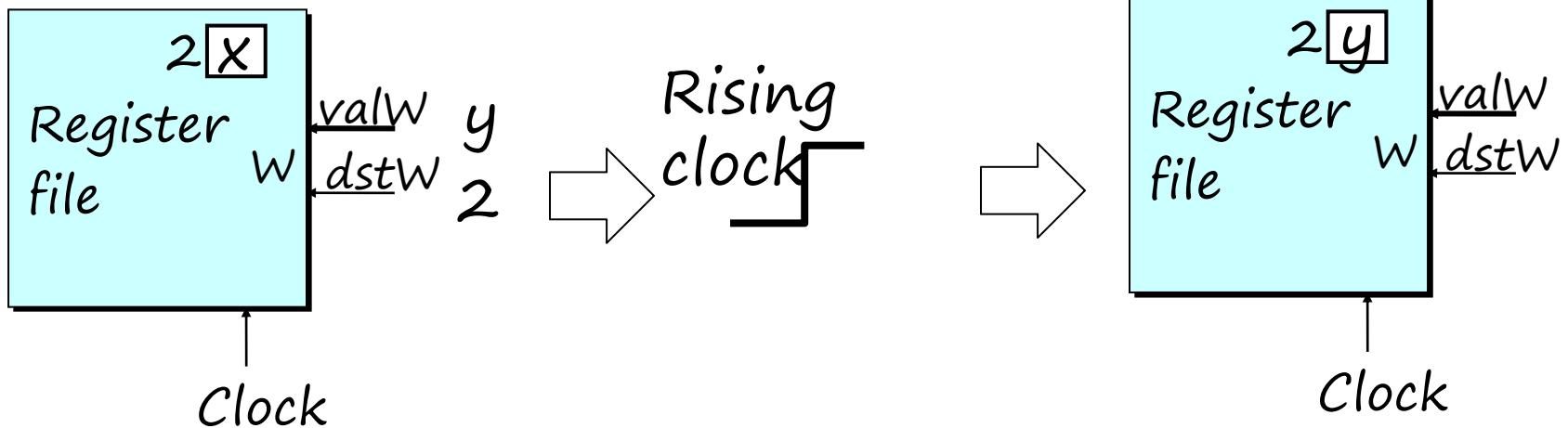


- **Register file**
  - Holds values of program registers
  - $\%eax$ ,  $\%esp$ , etc.
  - Register identifier serves as address
    - ID "F" implies no read or write performed
- **Multiple Ports**
  - Can read and/or write multiple words in one cycle
    - Each has separate address and data input/output

# Register File Timing

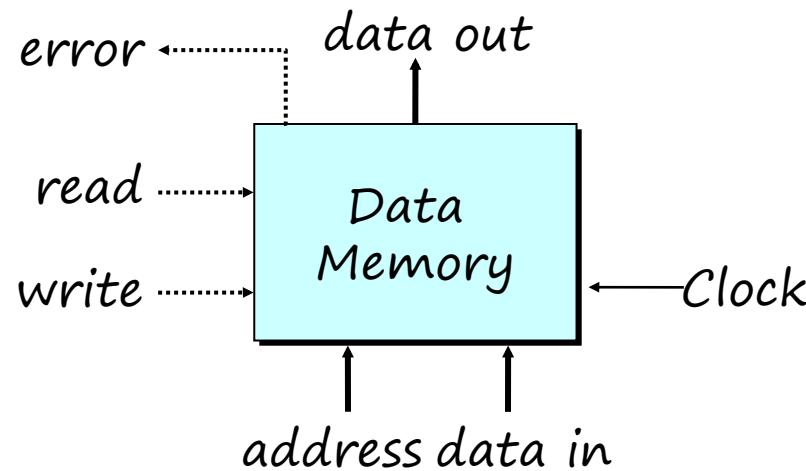


- **Reading**
  - Like combinational logic
  - Output data generated based on input address (After some delay)
- **Writing**
  - Like register
  - Update only as clock rises



# Memory

---



- **Memory**
  - Holds program data and instructions
- **Ports**
  - A single address input
  - A data input for writing, and a data output for reading
  - Error signal means invalid address

# Summary

---

- Computation
  - Performed by combinational logic
  - Computes “Boolean” functions
  - Continuously reacts to input changes

# Summary

---

- Storage
  - Clocked Registers
    - Hold single words (e.g., PC, CC)
    - Loaded as clock rises
  - Random-access memories
    - Hold multiple words (e.g., Register File, Memory)
    - Possible multiple read or write ports
    - Read word when address input changes
    - Write word as clock rises

# **Sequential CPU Implementation**

# Outline

---

- SEQ timing
- Organizing Processing into Stages
- Suggested Reading 4.3.1 ~ 4.3.3

# What is SEQ ?

---

- SEQ: Sequential Processor
  - Y86 processor
  - Process a complete instruction in each cycle
- What we have and we need?
  - Combinational logic and storage
  - ALU, Register File, Memory, PC, and CC

# SEQ Components

---

Combinational  
Logic



Clocked Register



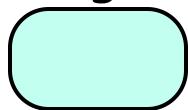
Memory



# SEQ Components

---

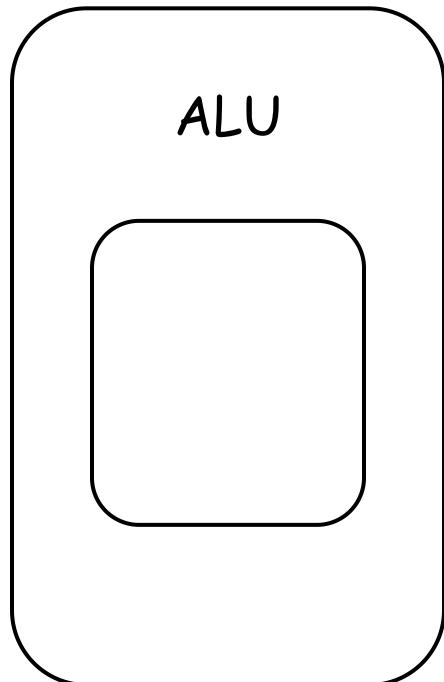
Combinational  
Logic



Clocked Register



Memory



# SEQ Components

---

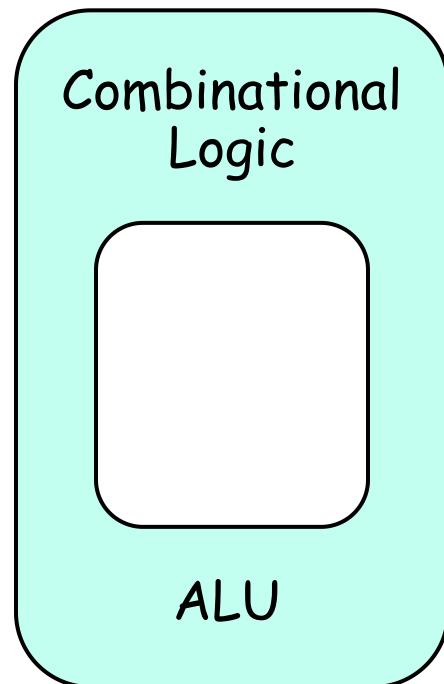
Combinational  
Logic



Clocked Register



Memory



# SEQ Components

---

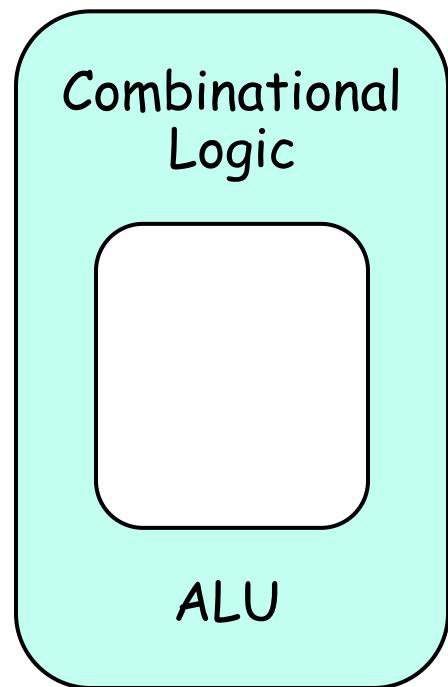
Combinational  
Logic



Clocked Register



Memory



Memory

Register  
File

# SEQ Components

---

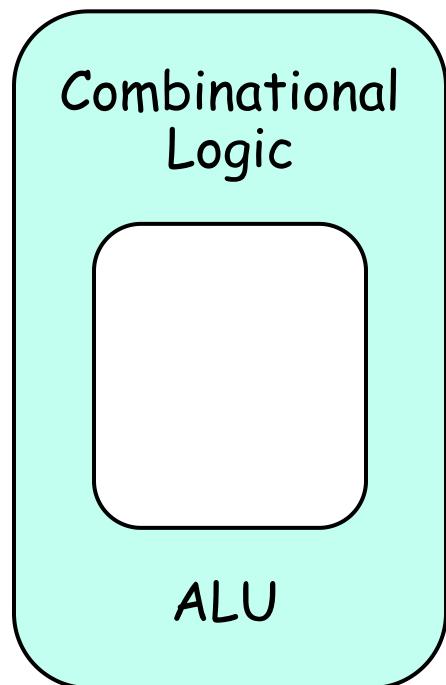
Combinational  
Logic



Clocked Register



Memory



Memory

Register  
File

# SEQ Components

---

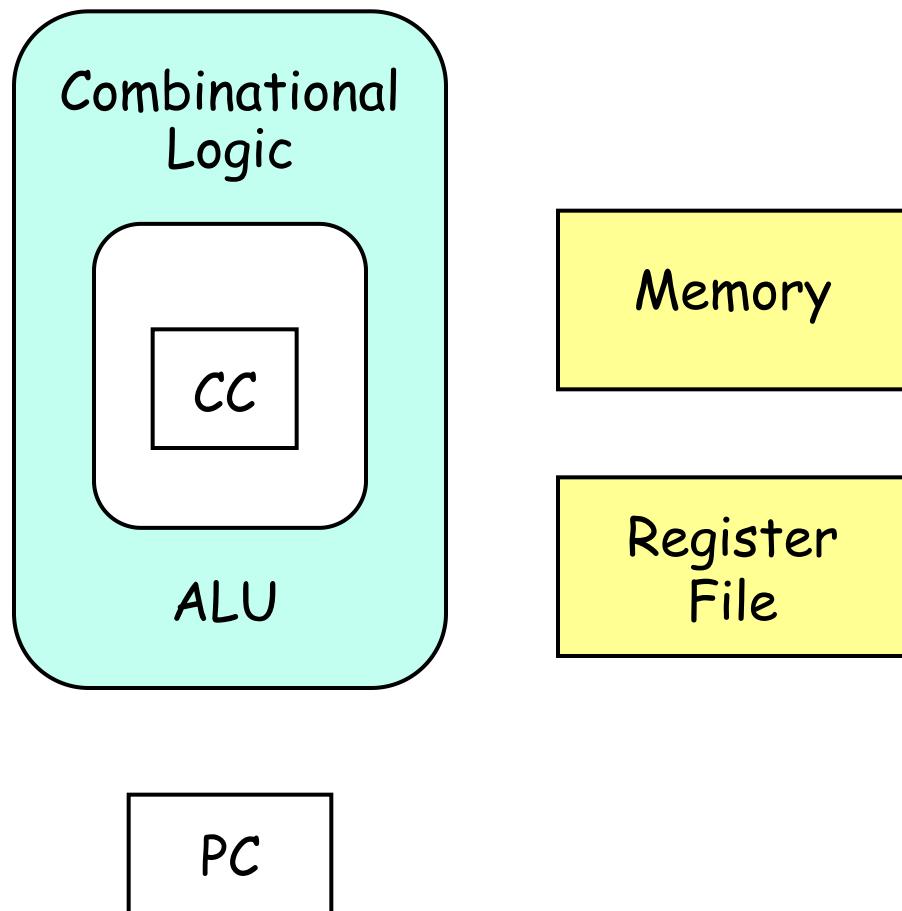
Combinational Logic



Clocked Register



Memory



# SEQ Components

---

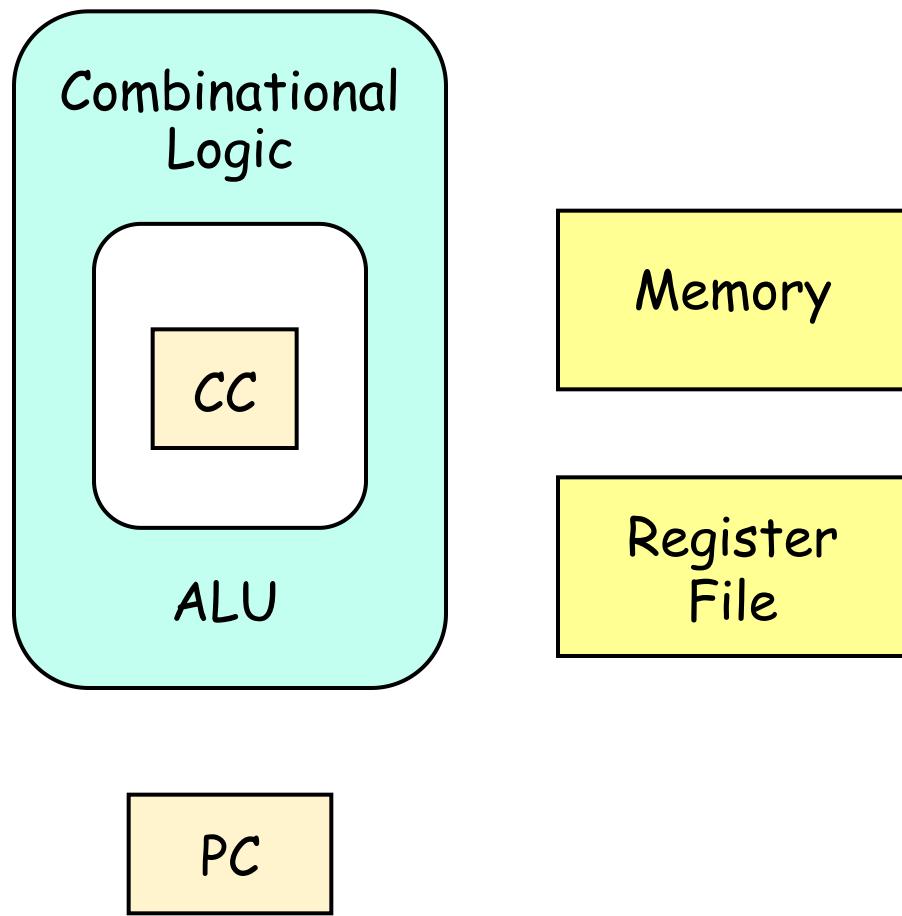
Combinational Logic



Clocked Register



Memory



# SEQ Components

---

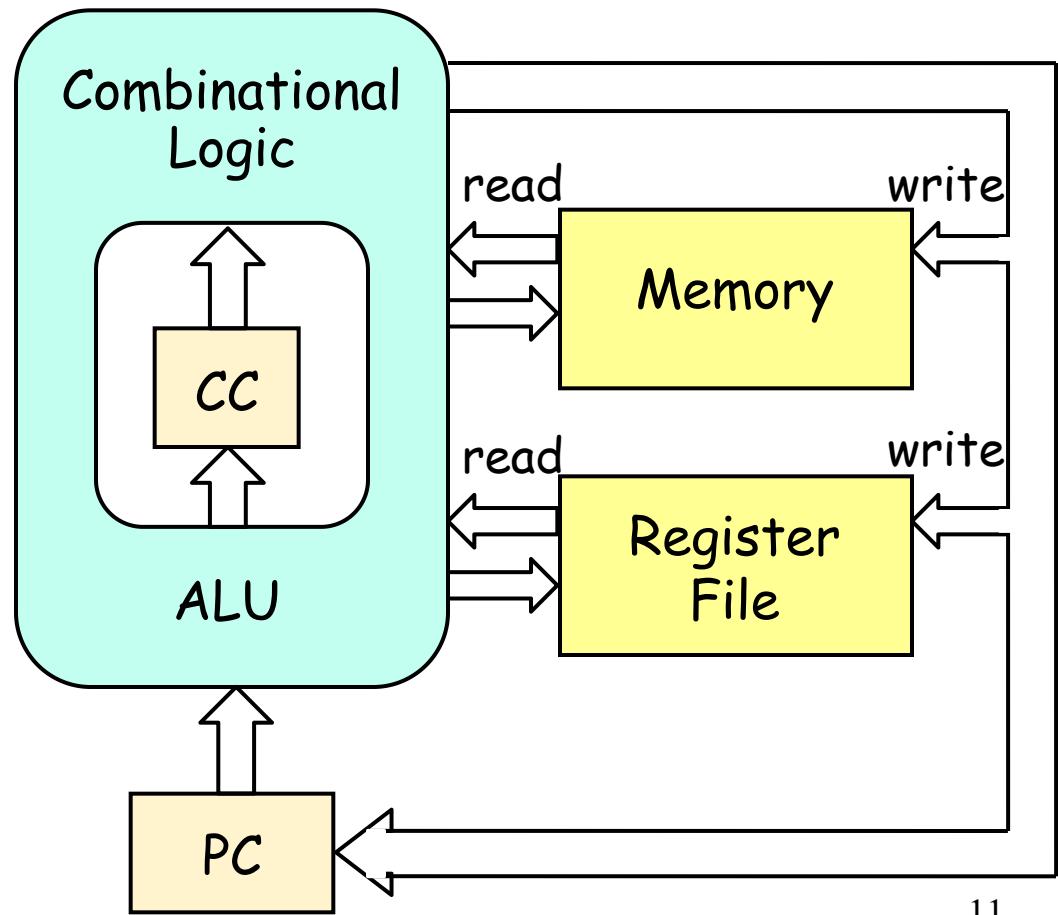
Combinational Logic



Clocked Register



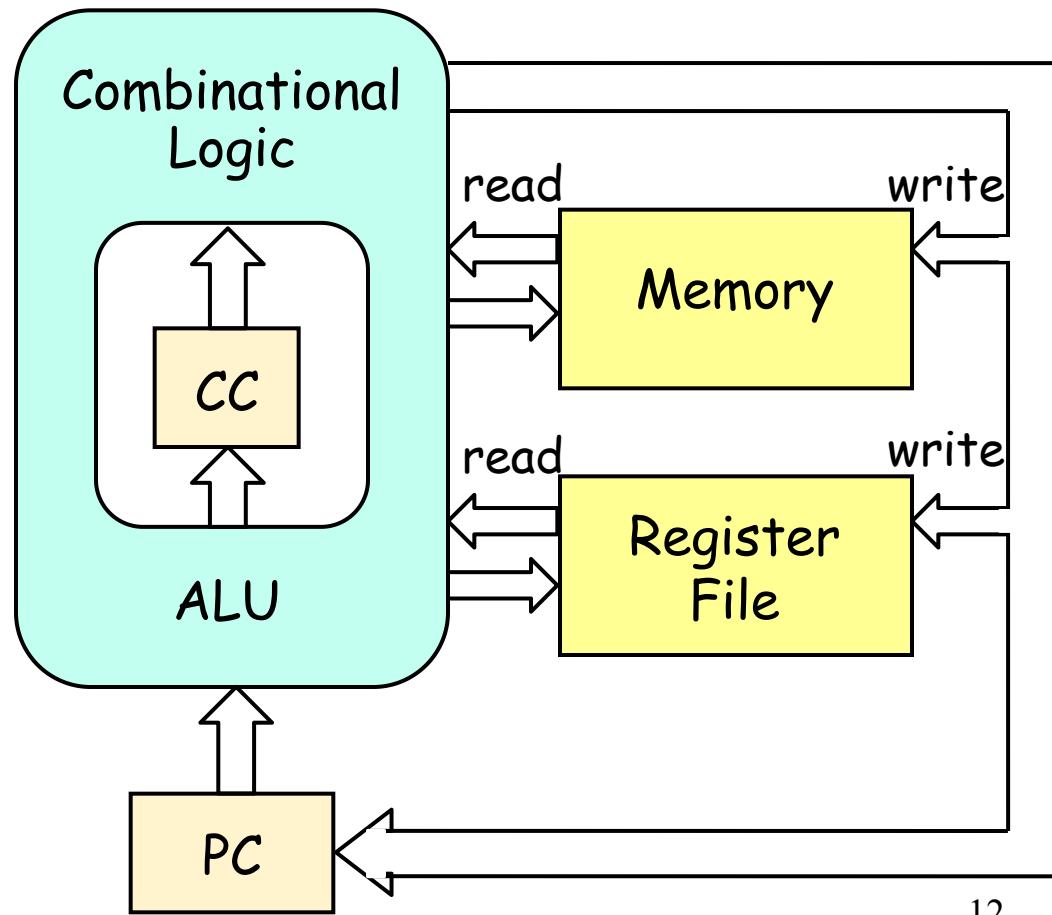
Memory



# SEQ Components

---

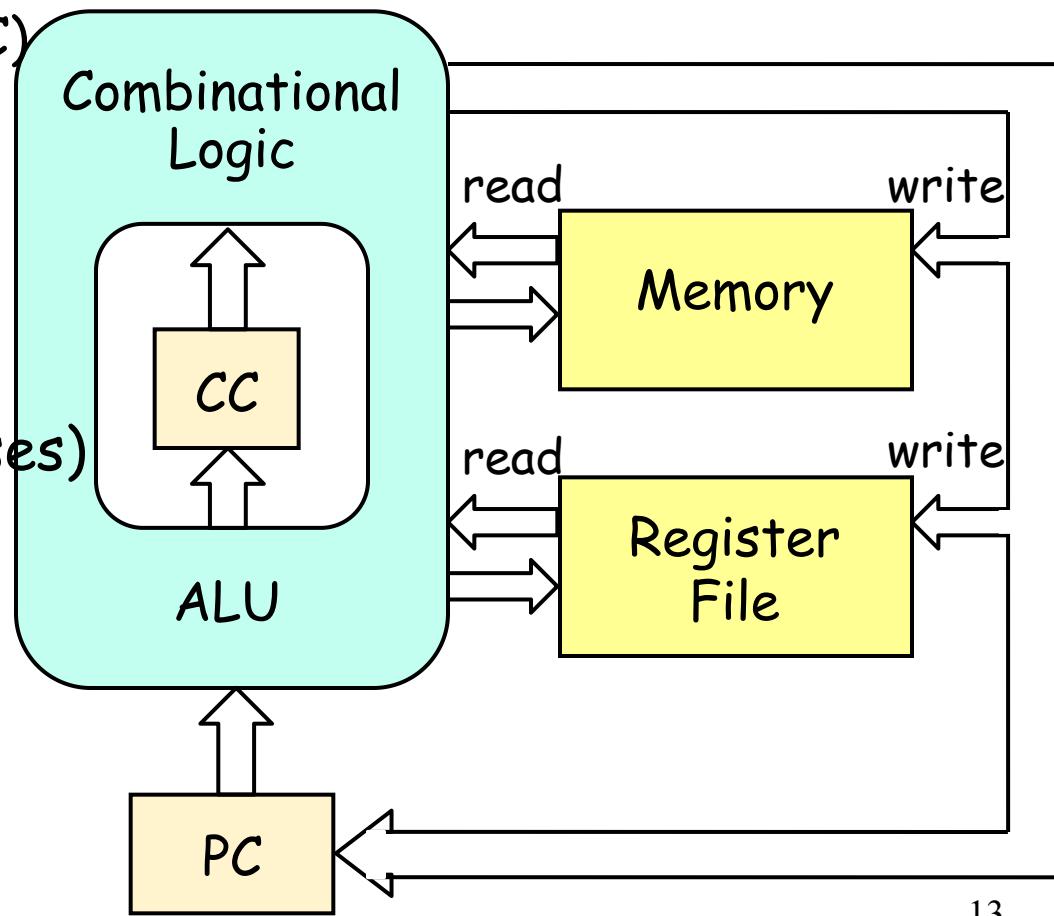
- Combinational Logic
  - ALU
  - Control logic
  - Memory reads
    - Inst. memory
    - Data memory
    - Register file



# SEQ Components

- Sequential Logic
  - Program counter (PC)
  - Condition code (CC)
  - Register File
  - Memories
  - a single clock signal

(All updated as clock rises)



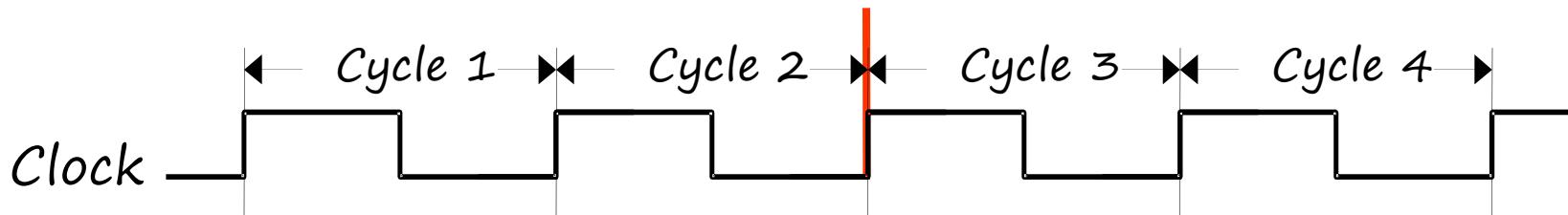
# Process Instruction on SEQ

---

- SEQ
  - Process a complete instruction in each cycle
  - A single clock controls all stat
    - Load CC and new PC
    - Write register file and memory

The processor never needs to read back the state updated by an instruction in order to complete the processing of this instruction.

# Beginning of Cycle 3



Cycle 1: 0x000: `irmovl $0x100,%ebx` # %ebx←0x100

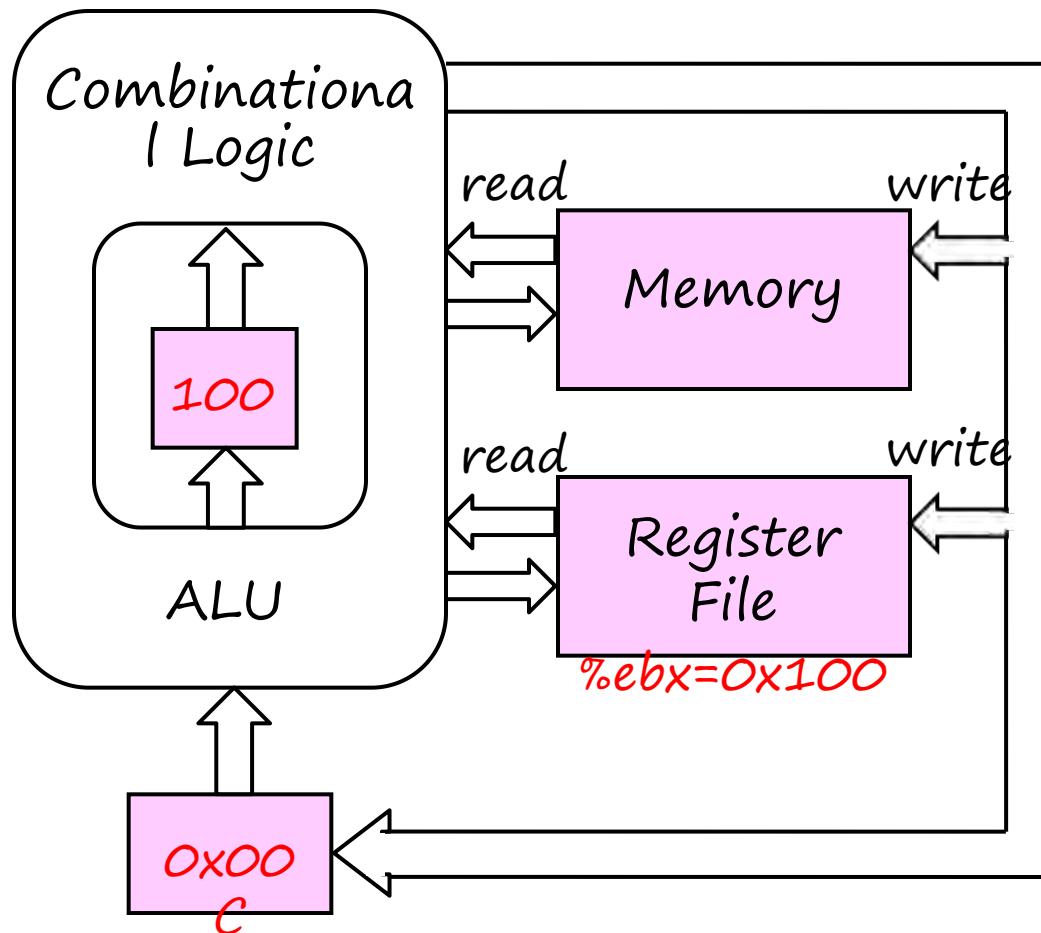
Cycle 2: 0x006: `irmovl $0x200,%edx` # %edx←0x200

Cycle 3: 0x00c: `addl %edx,%ebx` # %ebx←0x300 CC←000

Cycle 4: 0x00e: `je dest` # Not taken

# Beginning of Cycle 3

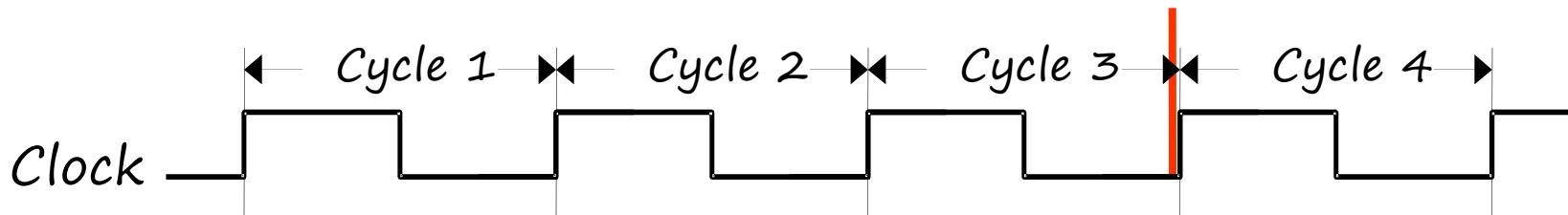
Cycle 2: 0x006: `irmovl $0x200,%edx # %edx←0x200`



- state set according to 2nd instruction
  - PC: 0x00C
  - CC: 100 (assume)
  - %edx: 0x200
  - %ebx: 0x100
- combinational logic starting to react to state changes

# End of Cycle 3

---



Cycle 1: 0x000: `irmovl $0x100,%ebx` # %ebx←0x100

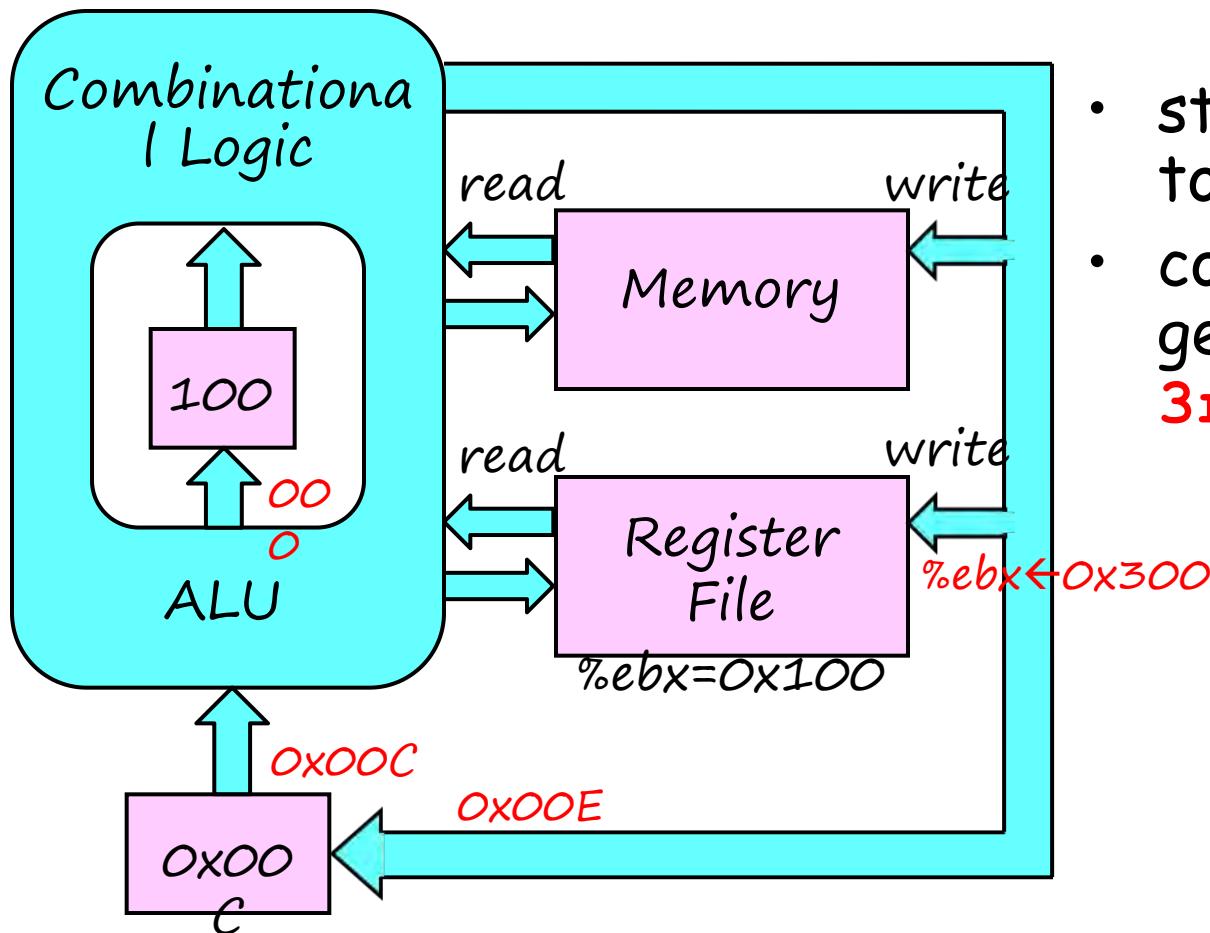
Cycle 2: 0x006: `irmovl $0x200,%edx` # %edx←0x200

Cycle 3: 0x00c: `addl %edx,%ebx` # %ebx←0x300 CC←000

Cycle 4: 0x00e: `je dest` # Not taken

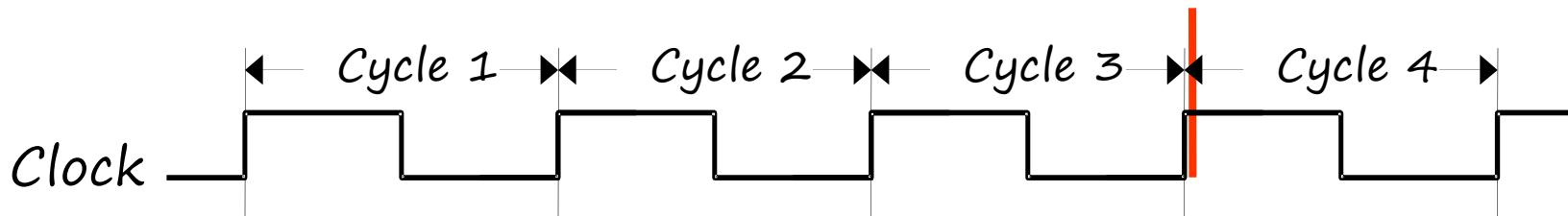
# End of Cycle 3

Cycle 3: 0x00c: addl %edx,%ebx # %ebx←0x300 cc←000



- state set according to **2nd instruction**
- combinational logic generates results for **3rd instruction**

# Beginning of Cycle 4



Cycle 1: 0x000: `irmovl $0x100,%ebx` #  $\%ebx \leftarrow 0x100$

Cycle 2: 0x006: `irmovl $0x200,%edx` #  $\%edx \leftarrow 0x200$

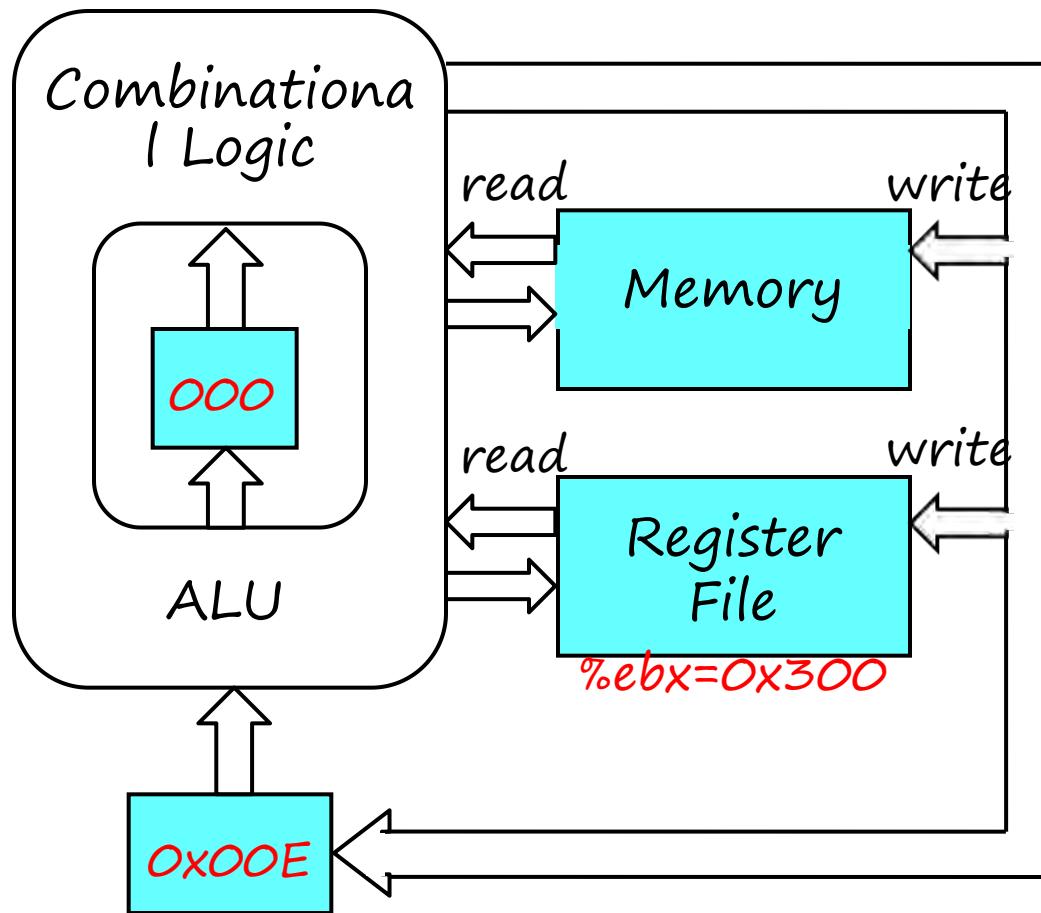
Cycle 3: 0x00c: `addl %edx,%ebx` #  $\%ebx \leftarrow 0x300$  CC $\leftarrow 000$

Cycle 4: 0x00e: `je dest` # Not taken

# Beginning of Cycle 4

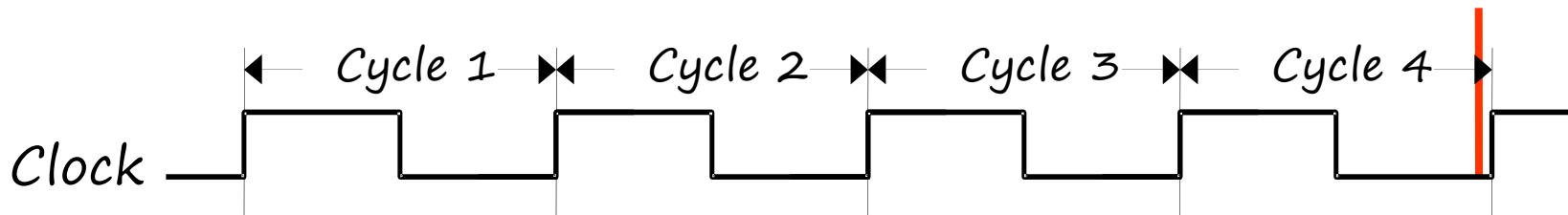
Cycle 4: 0x00e: je dest

# Not taken



- state set according to 3rd instruction
  - PC: 0x00E
  - CC: 000
  - %edx: 0x200
  - %ebx: 0x300
- combinational logic starting to react to state changes

# End of Cycle 4



Cycle 1: 0x000: `irmovl $0x100,%ebx` # %ebx←0x100

Cycle 2: 0x006: `irmovl $0x200,%edx` # %edx←0x200

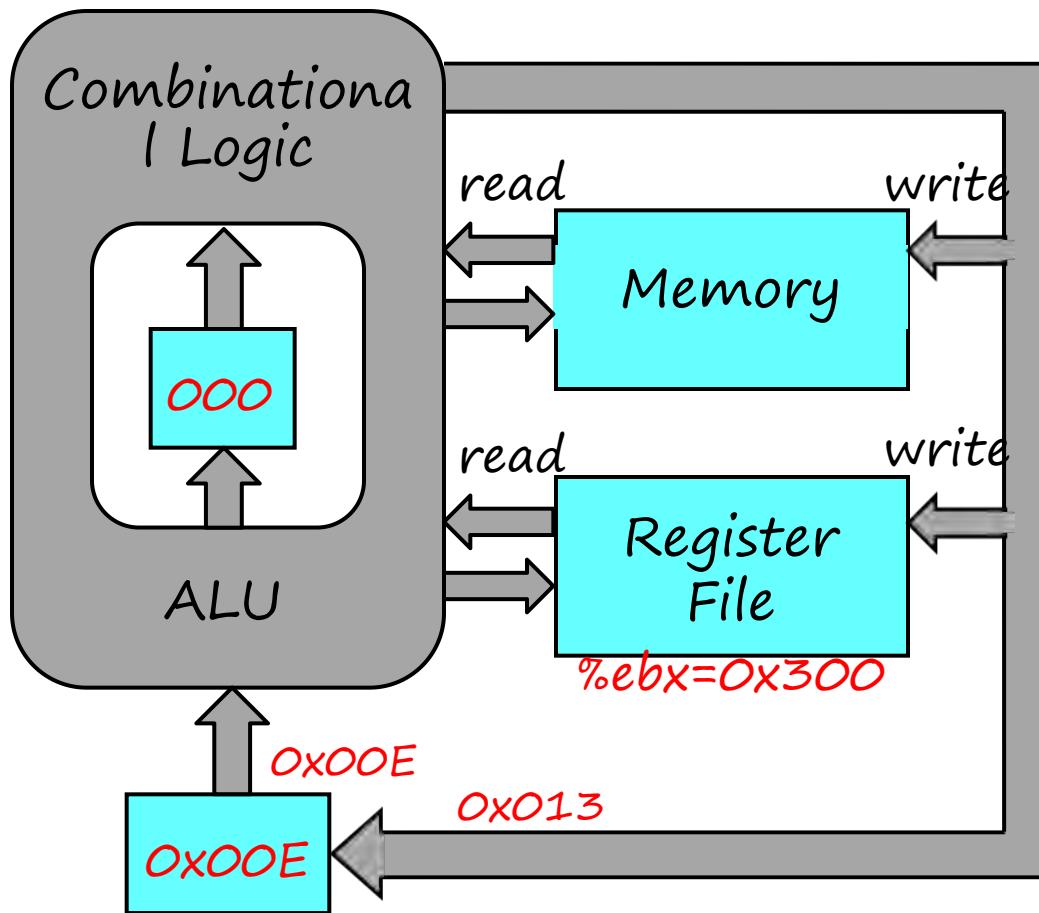
Cycle 3: 0x00c: `addl %edx,%ebx` # %ebx←0x300 CC←000

Cycle 4: 0x00e: `je dest` # Not taken

# End of Cycle 4

Cycle 4: 0x00e: je dest

# Not taken



- state set according to **3rd instruction**
- combinational logic generates results for **4th instruction**

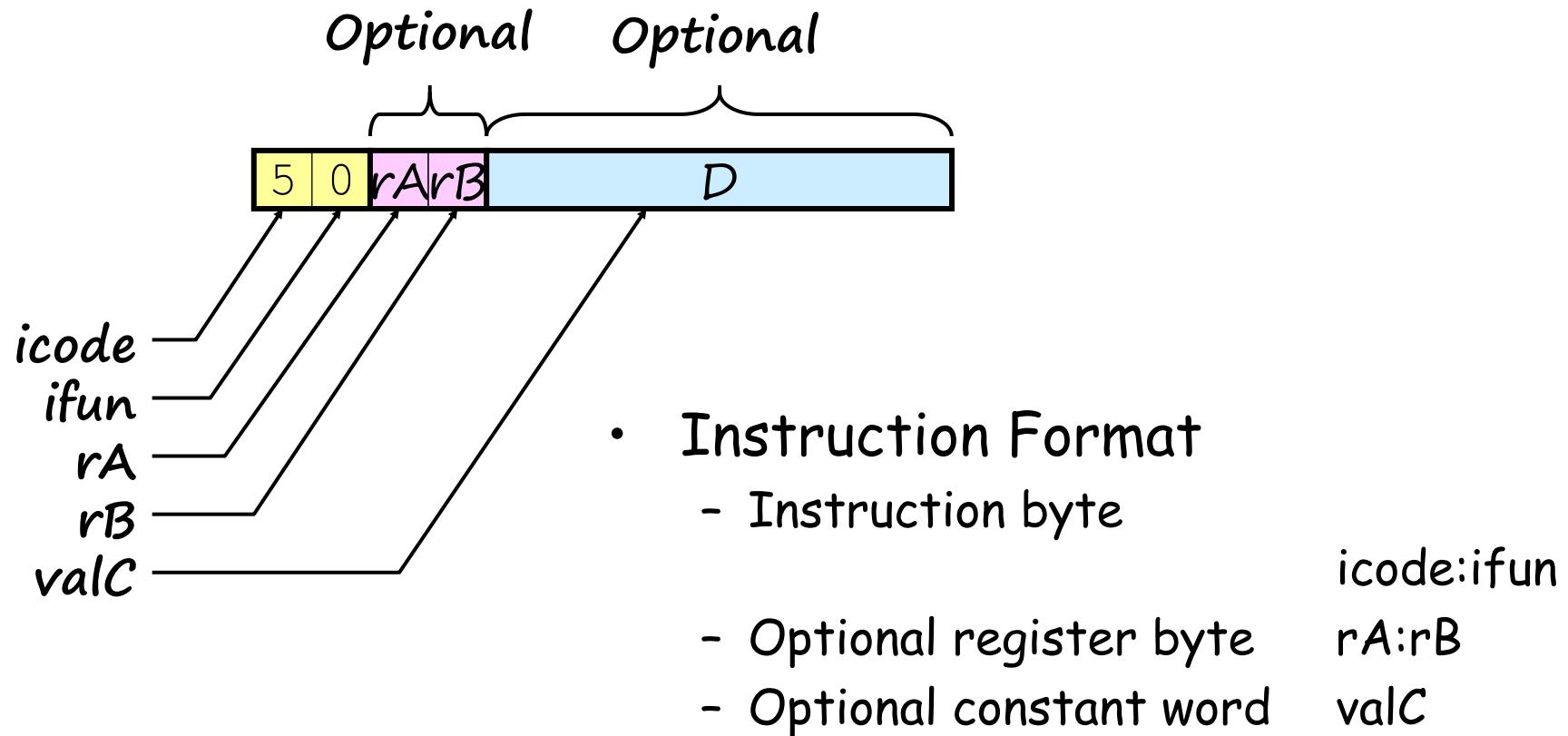
# How to design SEQ ?

---

- Naïve Design: One-by-one
  - Straightforward, but waste
- Advanced Design: Multi-stages
  - Formulate instruction execution as sequence of simple steps (stages)
    - Like: function for programming
  - Best use of hardware
  - Challenge: use same general form for all instructions

# Y86 Instruction Decoding

---



# Y86 Instruction Set

Byte

0 1 2 3 4 5

nop

0	0
---	---

halt

1	0
---	---

rrmovl rA, rB

2	0	rA	rB
---	---	----	----

irmovl V, rB

3	0	F	rB	V
---	---	---	----	---

rmmovl rA, D(rB)

4	0	rA	rB	D
---	---	----	----	---

mrmovl D(rB), rA

5	0	rA	rB	D
---	---	----	----	---

OPl rA, rB

6	f	n	rA	rB
---	---	---	----	----

addl 6|0  
subl 6|1  
andl 6|2  
xorl 6|3

# Y86 Instruction Set

Byte

jXX Dest

Byte	0	1	2	3	4	5	
jXX Dest	7	fn					jmp 7 0

call Dest

call Dest	8	0					jle 7 1
-----------	---	---	--	--	--	--	---------

ret

ret	9	0					jl 7 2
-----	---	---	--	--	--	--	--------

pushl rA

pushl rA	A	0	rA	F			je 7 3
----------	---	---	----	---	--	--	--------

popl rA

popl rA	B	0	rA	F			jne 7 4
---------	---	---	----	---	--	--	---------

							jge 7 5
--	--	--	--	--	--	--	---------

							jg 7 6
--	--	--	--	--	--	--	--------

# Lab5: Y86sim

example: pushl %edx

- 1: get icode: a
- get ifun: 0
- 2: get regA: reg[%edx]
- get regB: reg[F]
- 3: addr: reg[%esp]-4
- 4: set mem: reg[%edx]
- 5: set reg: reg[%esp]
- 6: pc: pc + 2

```
int nexti(argc, argv)
{
    /* get code and function */
    /* get register and immediate */

    /* execute */
    switch(icode) {
        /* alu + r/w mem + w reg */
        pc = next_pc;
    }

    /* do alu */
    long_t compute_alu(op, regA, regB) {}

    /*set memory */
    get_mem_val(m, addr) {}

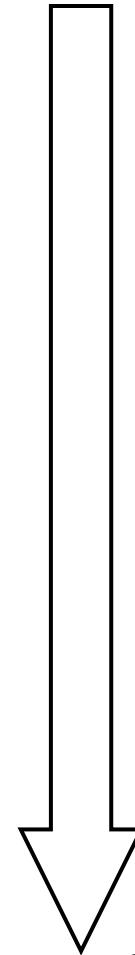
    /*set memory */
    set_mem_val(m, addr, val) {}

    /* set regs */
    set_reg_val(rf, id, val) {}
```

# Instruction Execution Stages

---

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



PC

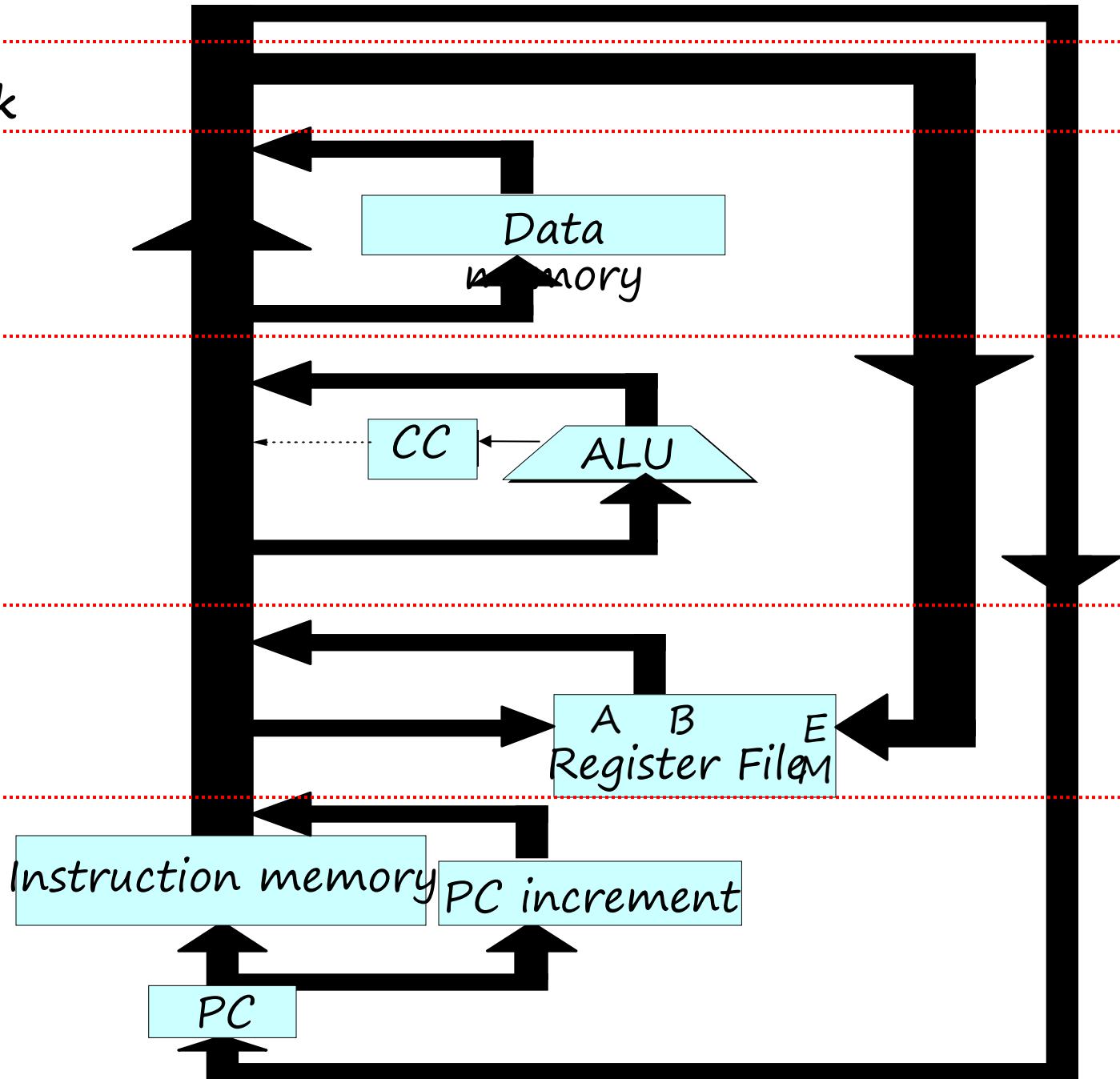
Write back

Memory

Execute

Decode

Fetch



# SEQ Hardware Structure

---

- Instruction Flow
  - Read instruction at address specified by PC: 1
  - Process through stages: 2 → 5
  - Update program counter: 6

# Executing arith./log. operation

---

opl rA, rB

6	fn	rA	rB
---	----	----	----

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set Condition Codes
- Memory
  - Do nothing
- Write back
  - Update register rB
- PC Update
  - Increment PC by 2

# Stage Computation: arith./log. ops

---

	$OPI\ rA, rB$	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$  $valP \leftarrow PC+2$	Read instruction Read register byte Compute next PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Read operand A Read operand B
Execute	$valE \leftarrow valB\ ifun$ Set CC	Perform ALU operation Set condition code register
Memory		
Write back	$R[rB] \leftarrow valE$	Write back result
PC update	$PC \leftarrow valP$	Update PC

# Stage Computation: arith./log. ops

	OPI rA, rB	subl %edx, %ebx   0x00C:
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$  $\text{valP} \leftarrow PC+2$	$\text{icode:ifun} \leftarrow M_1[0x00C] =$ $rA:rB \leftarrow M_1[0x00C+1] =$  $\text{valP} \leftarrow 0x00C+2 = 0x00E$
Decode	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$	$\text{valA} \leftarrow R[%edx] = 9$ $\text{valB} \leftarrow R[%ebx] = 21$
Execute	$\text{valE} \leftarrow \text{valB ifun}$ Set CC	$\text{valE} \leftarrow 21 - 9 = 12$ $ZF \leftarrow 0, SF \leftarrow 0, OF \leftarrow 0$
Memory		
Write back	$R[rB] \leftarrow \text{valE}$	$R[%ebx] \leftarrow \text{valE} = 12$
PC update	$PC \leftarrow \text{valP}$	$PC \leftarrow \text{valP} = 0x00E$

PC

Write back

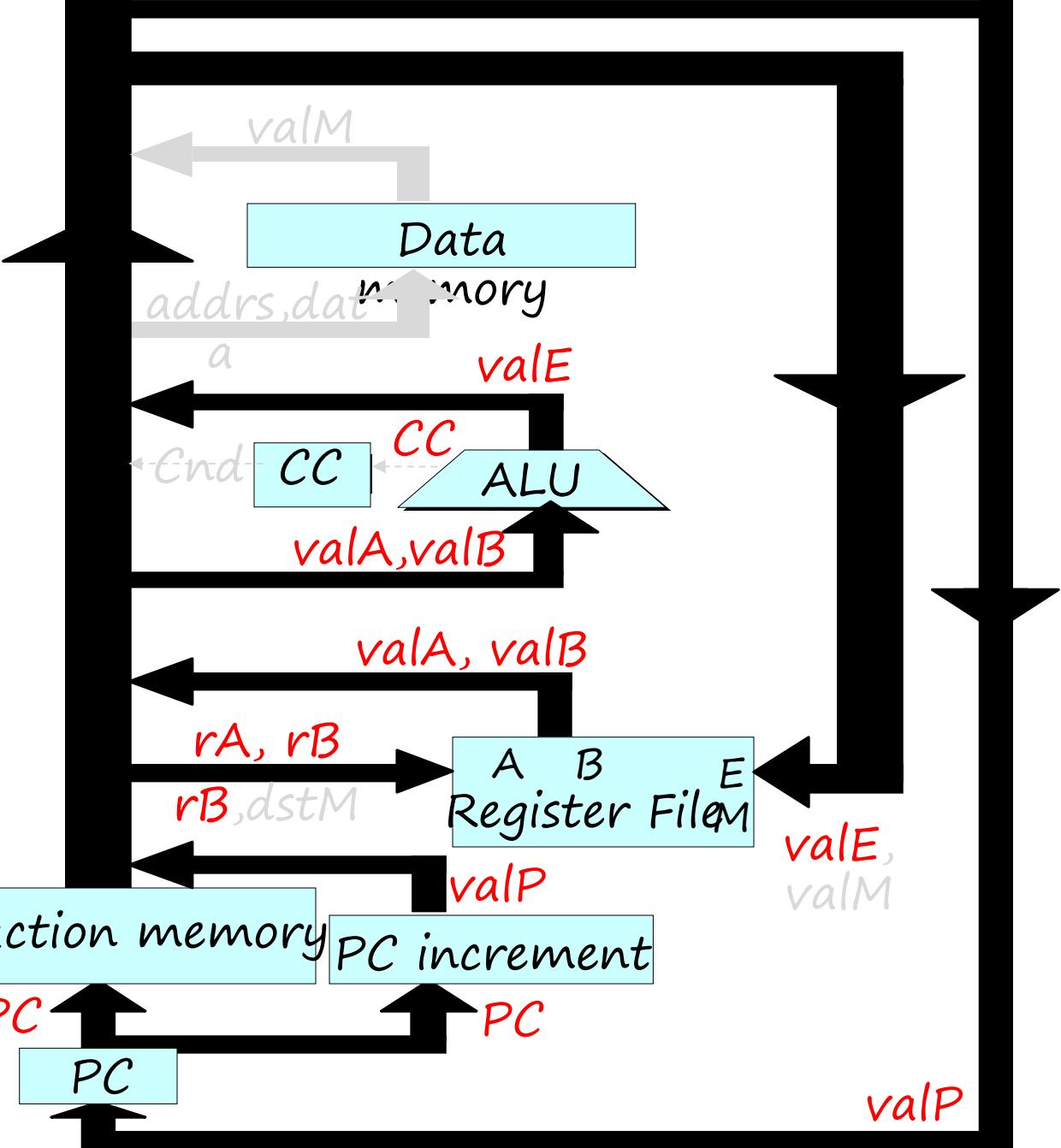
Memory

Execute

Decode

icode:ifun,  
rA:rB

Fetch<sub>IC</sub>



# Executing rrmovl

---

rrmovl rA, rB

2	0	rA	rB
---	---	----	----

- Fetch
  - Read 2 bytes
- Decode
  - Read operand register rA
- Execute
  - Do nothing
- Memory
  - Do nothing
- Write back
  - Update register rB
- PC Update
  - Increment PC by 2

# Stage Computation: rrmovl

	rrmovl rA, rB	
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$	Read instruction byte Read register byte
	$\text{valP} \leftarrow PC+2$	Compute next PC
Decode	$\text{valA} \leftarrow R[rA]$	Read operand A
Execute	$\text{valE} \leftarrow O + \text{valA}$	Perform ALU operation <b>(generalization)</b>
Memory		
Write back	$R[rB] \leftarrow \text{valE}$	Write back result
PC update	$PC \leftarrow \text{valP}$	Update PC

PC

Write back

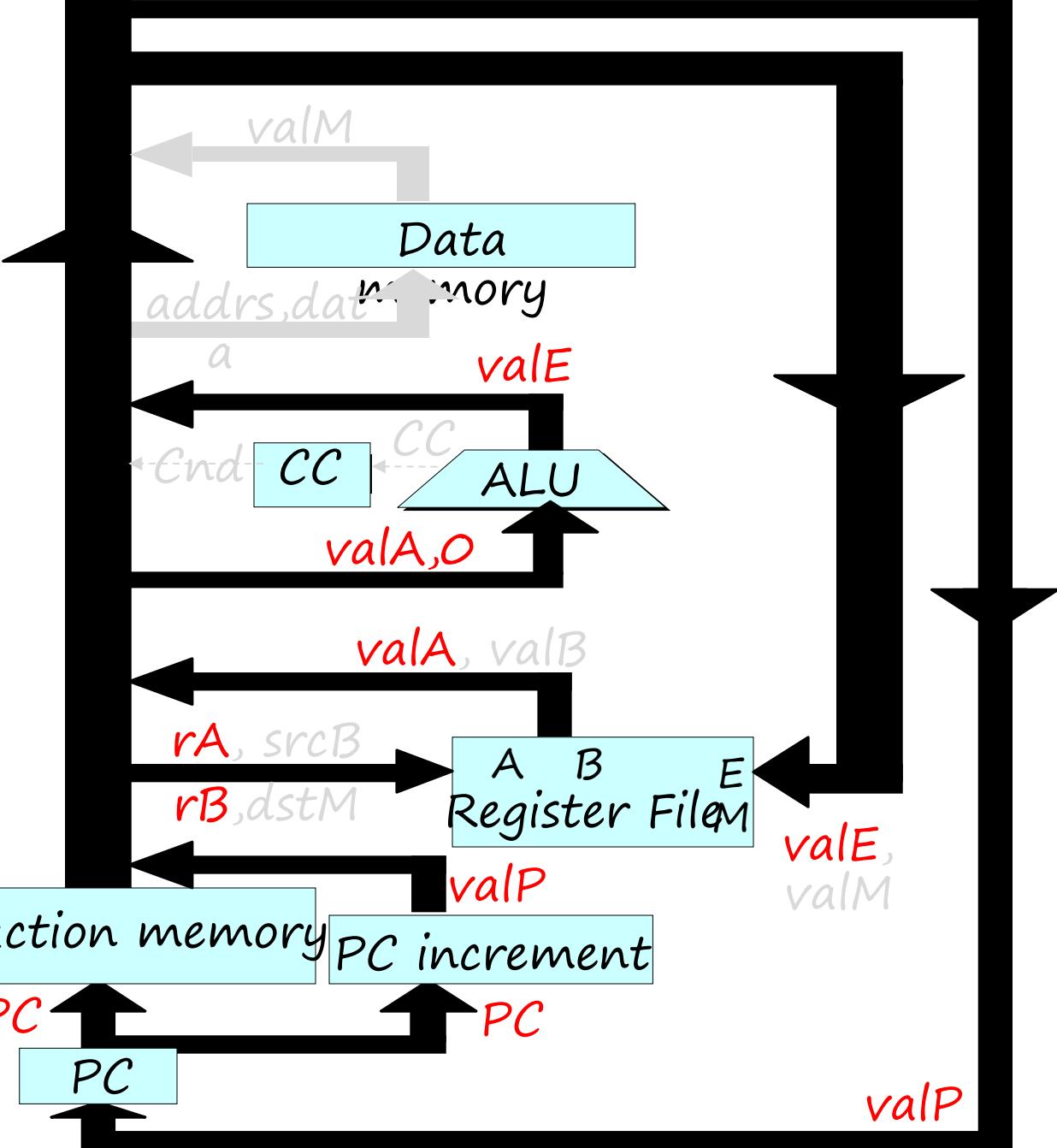
Memory

Execute

Decode

icode:ifun,  
rA:rB

Fetch<sub>IC</sub>



# Executing irmovl

---

irmovl V, rB

3	0	F	rB
---	---	---	----

V

- Fetch
  - Read 6 bytes
- Decode
  - Do nothing
- Execute
  - Do nothing
- Memory
  - Do nothing
- Write back
  - Update register rB
- PC Update
  - Increment PC by 6

# Stage Computation: irmovl

	irmovl rA, rB	
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $\text{valC} \leftarrow M_4[PC+2]$ $\text{valP} \leftarrow PC+6$	Read instruction byte Read register byte Read constant value Compute next PC
Decode		
Execute	$\text{valE} \leftarrow O + \text{valC}$	Perform ALU operation <span style="color:red">(generalization)</span>
Memory		
Write back	$R[rB] \leftarrow \text{valE}$	Write back result
PC update	$PC \leftarrow \text{valP}$	Update PC

PC

Write back

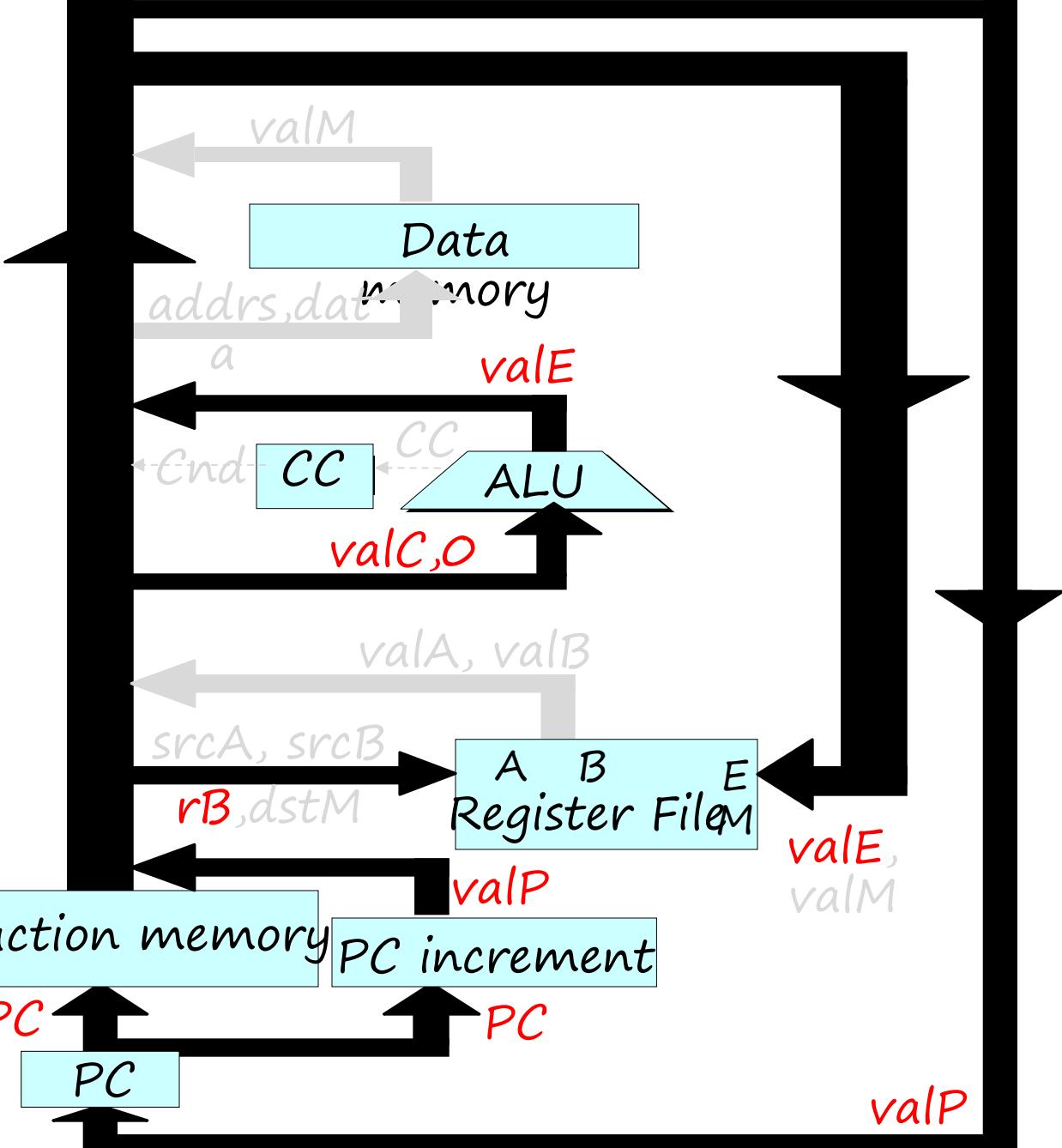
Memory

Execute

Decode

icode:ifun,  
rA:rB

Fetch



# Executing `rmmovl`

---

`rmmovl rA, D(rB)` 

- Fetch
  - Read 6 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 6

# Stage Computation: rmmovl

rmmovl rA, D(rB)	
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $\text{valC} \leftarrow M_4[PC+2]$ $\text{valP} \leftarrow PC+6$
Decode	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$
Memory	$M_4[\text{valE}] \leftarrow \text{valA}$
Write back	
PC update	$PC \leftarrow \text{valP}$

- Use ALU for address computation

PC

Write back

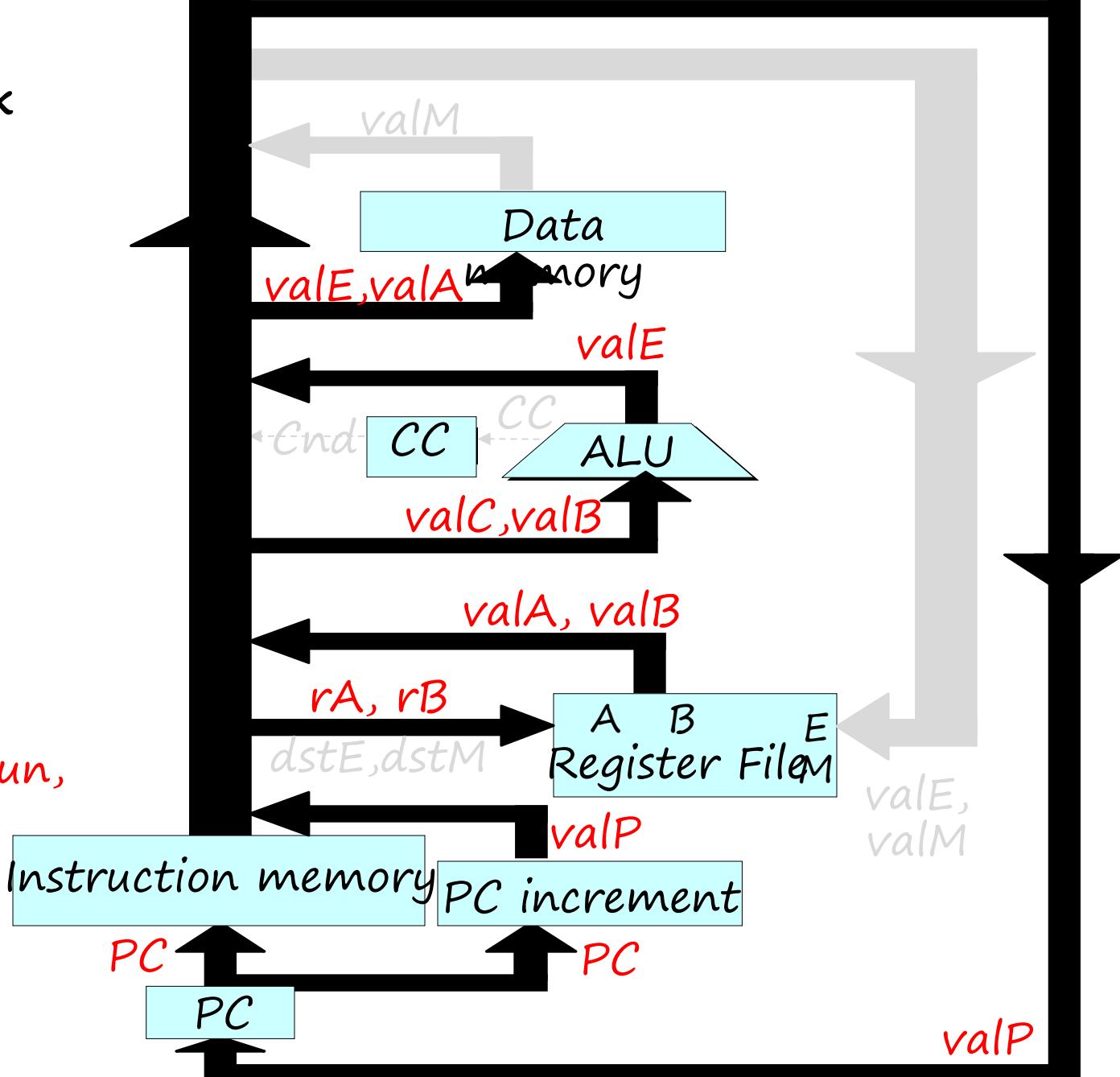
Memory

Execute

Decode

icode:ifun,  
rA:rB

Fetch



# Executing mrmovl

---

mrmovl	D(rB),rA	5	0	rA	rB		D
--------	----------	---	---	----	----	--	---

- Fetch
  - Read 6 bytes
- Decode
  - Read operand register rB
- Execute
  - Compute effective address
- Memory
  - Read from memory
- Write back
  - Update register rA
- PC Update
  - Increment PC by 6

# Stage Computation: mrmovl

	mrmovl D(rB) , rA	
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $\text{valC} \leftarrow M_4[PC+2]$ $\text{valP} \leftarrow PC+6$	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	$\text{valB} \leftarrow R[rB]$	Read operand B
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
Memory	$\text{valM} \leftarrow M_4[\text{valE}]$	Read data from memory
Write		Update register rA
back update	$R[rA] \leftarrow \text{valM}$	Update PC
update	$PC \leftarrow \text{valP}$	

- Use ALU for address computation

PC

Write back

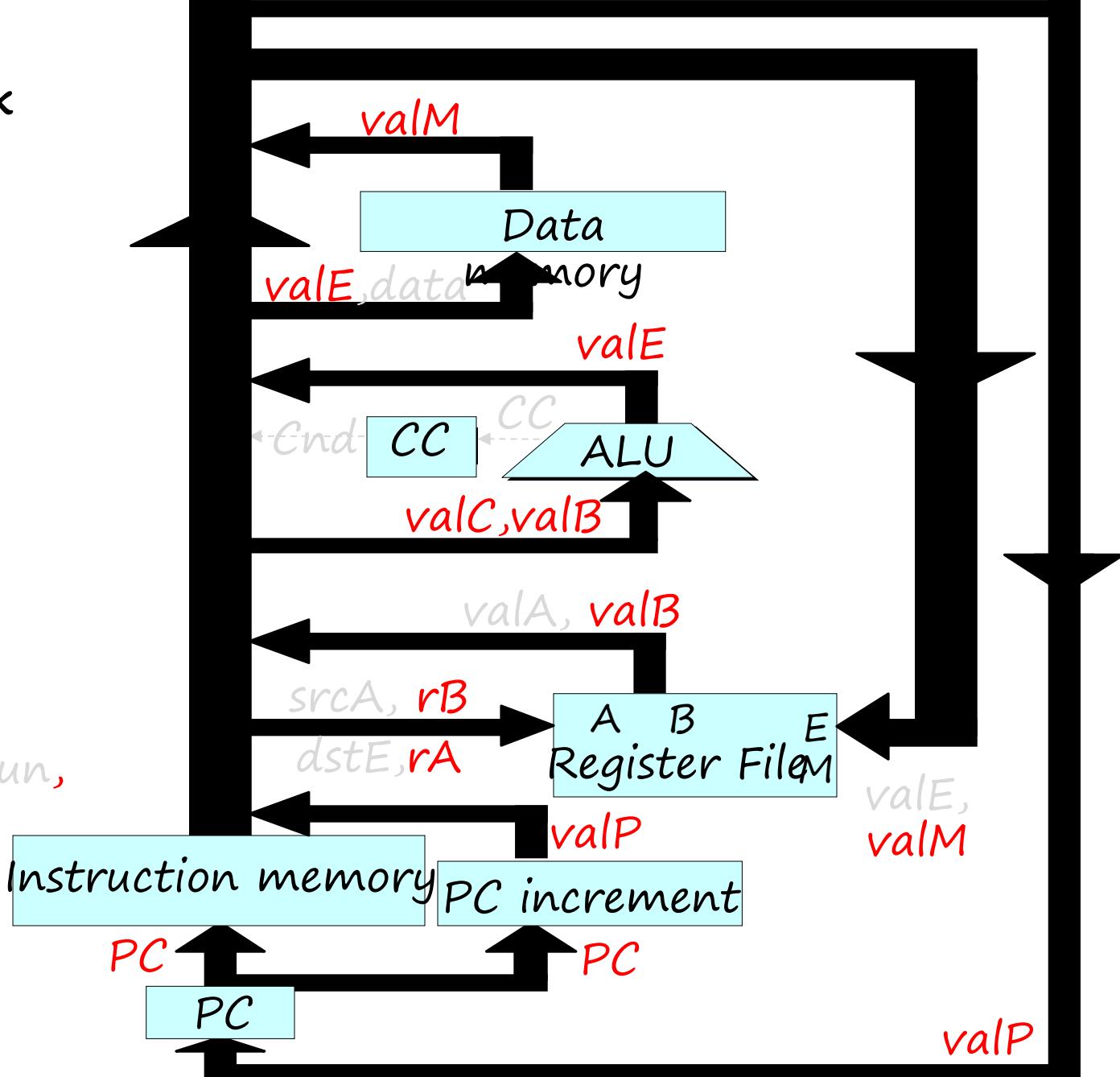
Memory

Execute

Decode

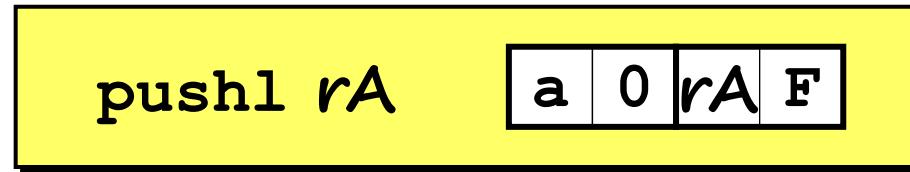
icode:ifun,  
rA:rB

Fetch



# Executing pushl

---



- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer and register rA
- Execute
  - Decrement stack pointer by 4
- Memory
  - Store valA at the address of new stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Increment PC by 2

# Stage Computation: pushl

	<code>pushl rA</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$	Read instruction byte Read register byte
	$valP \leftarrow PC+2$	Compute next PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[\%esp]$	Read valA Read stack pointer
Execute	$valE \leftarrow valB + (-4)$	Decrement stack pointer
Memory	$M_4[valE] \leftarrow valA$	Store to stack
Write back	$R[\%esp] \leftarrow valE$	Update stack pointer
PC update	$PC \leftarrow valP$	Update PC

- Use ALU to decrement stack pointer

PC

Write back

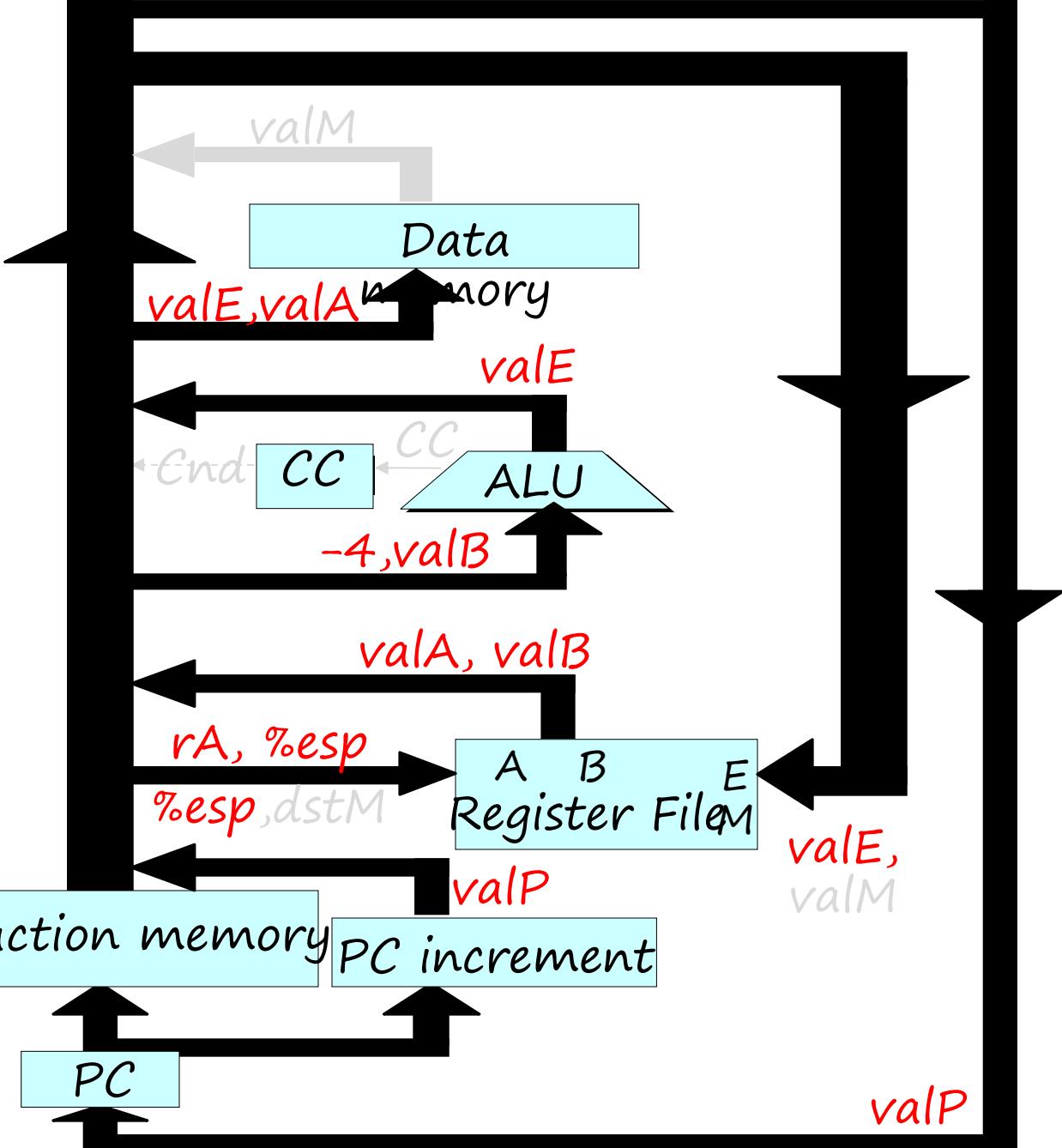
Memory

Execute

Decode

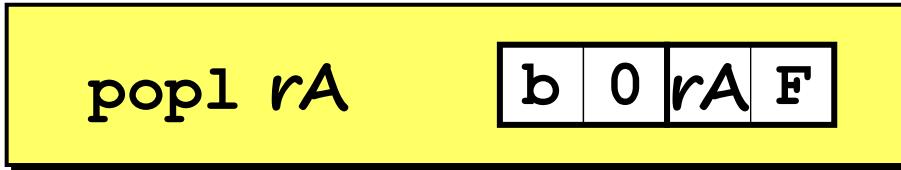
icode:ifun,  
rA:rB

Fetch<sub>IC</sub>



# Executing popl

---



- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read from old stack pointer
- Write back
  - Update stack pointer
  - Update register rA
- PC Update
  - Increment PC by 2<sub>50</sub>

# Stage Computation: popl

popl rA	
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $\text{valP} \leftarrow PC+2$
Decode	$\text{valA} \leftarrow R[\%esp]$ $\text{valB} \leftarrow R[\%esp]$
Execute	$\text{valE} \leftarrow \text{valB} + 4$
Memory	$\text{valM} \leftarrow M_4[\text{valA}]$
Write back	$R[\%esp] \leftarrow \text{valE}$ $R[rA] \leftarrow \text{valM}$
PC update	$PC \leftarrow \text{valP}$

Read instruction byte  
Read register byte

Compute next PC  
Read stack pointer  
Read stack pointer  
Increment stack pointer

Read from stack  
Update stack pointer  
Write back result

Update PC  
update two registers  
(special: rA=%esp)

- Use ALU to increment stack pointer

PC

Write back

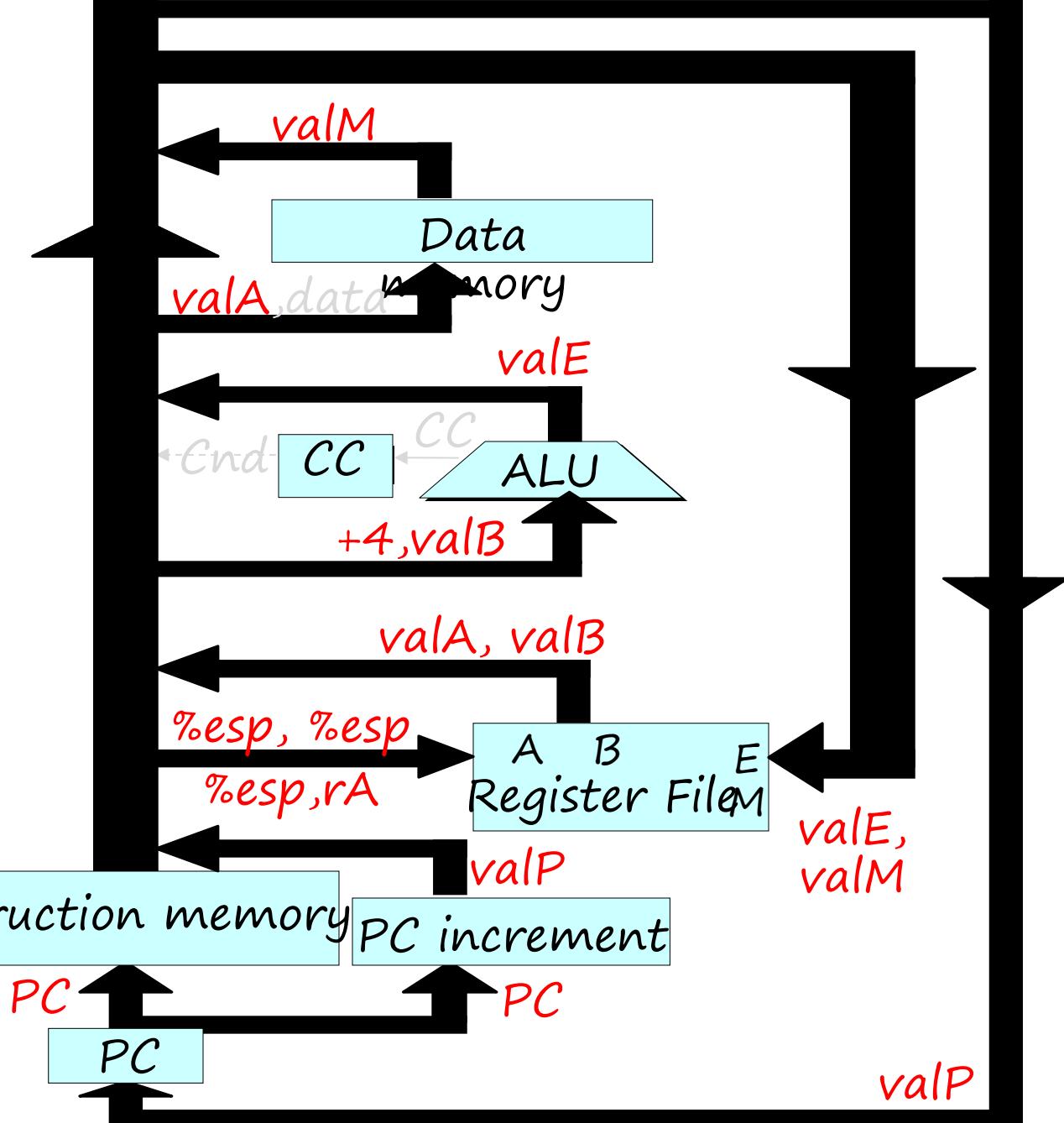
Memory

Execute

Decode

icode:ifun,  
rA:rB

Fetch



# Executing Jumps

---



# Executing Jumps

---

- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

# Stage Computation: Jumps

	jXX Dest
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $\text{valC} \leftarrow M_4[PC+1]$ $\text{valP} \leftarrow PC+5$
Decode	
Execute	$\text{Cnd} \leftarrow \text{Cond(CC,ifun)}$
Memory	
Write	
back PC update	$PC \leftarrow \boxed{\text{Cnd} ? \text{valC} : \text{valP}}$

Read instruction byte

Read destination address through address

Compute both addresses

Take branch?

Update PC

Chose based on CC and ifun

PC

Write back

Memory

Execute

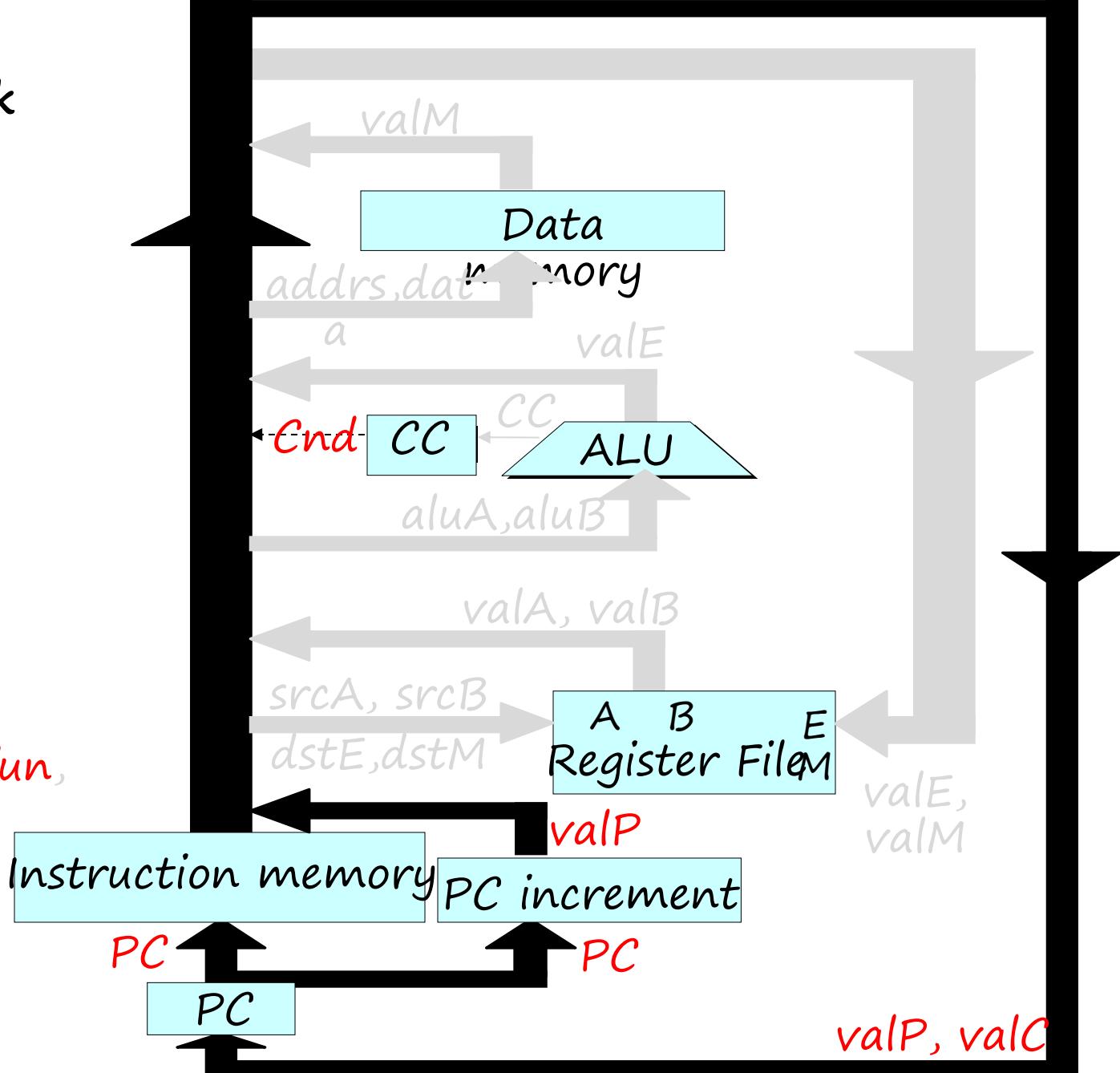
Decode

icode:ifun,

rA:rB

Fetch

valIC



# Executing call

---

call Dest

8	0
---	---

Dest

return:

xx	xx
----	----

target:

xx	xx
----	----

# Executing call

---

- Fetch
  - Read 5 bytes
  - Increment PC by 5
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 4
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

# Stage Computation: call

	call Dest icode:ifun $\leftarrow M_1[PC]$	
Fetch	$valC \leftarrow M_4[PC+1]$ $valP \leftarrow PC+5$	Read instruction byte Read destination address Compute return point
Decode	$valB \leftarrow R[\%esp]$	Read stack pointer
Execute	$valE \leftarrow valB + -4$	Decrement stack pointer
Memory	$M_4[valE] \leftarrow valP$	Write return value on <del>update</del> stack pointer
Write back	$R[\%esp] \leftarrow valE$	Store incremental PC
PC update	$PC \leftarrow valC$	Set PC to destination

- Use ALU to decrement stack pointer

PC

Write back

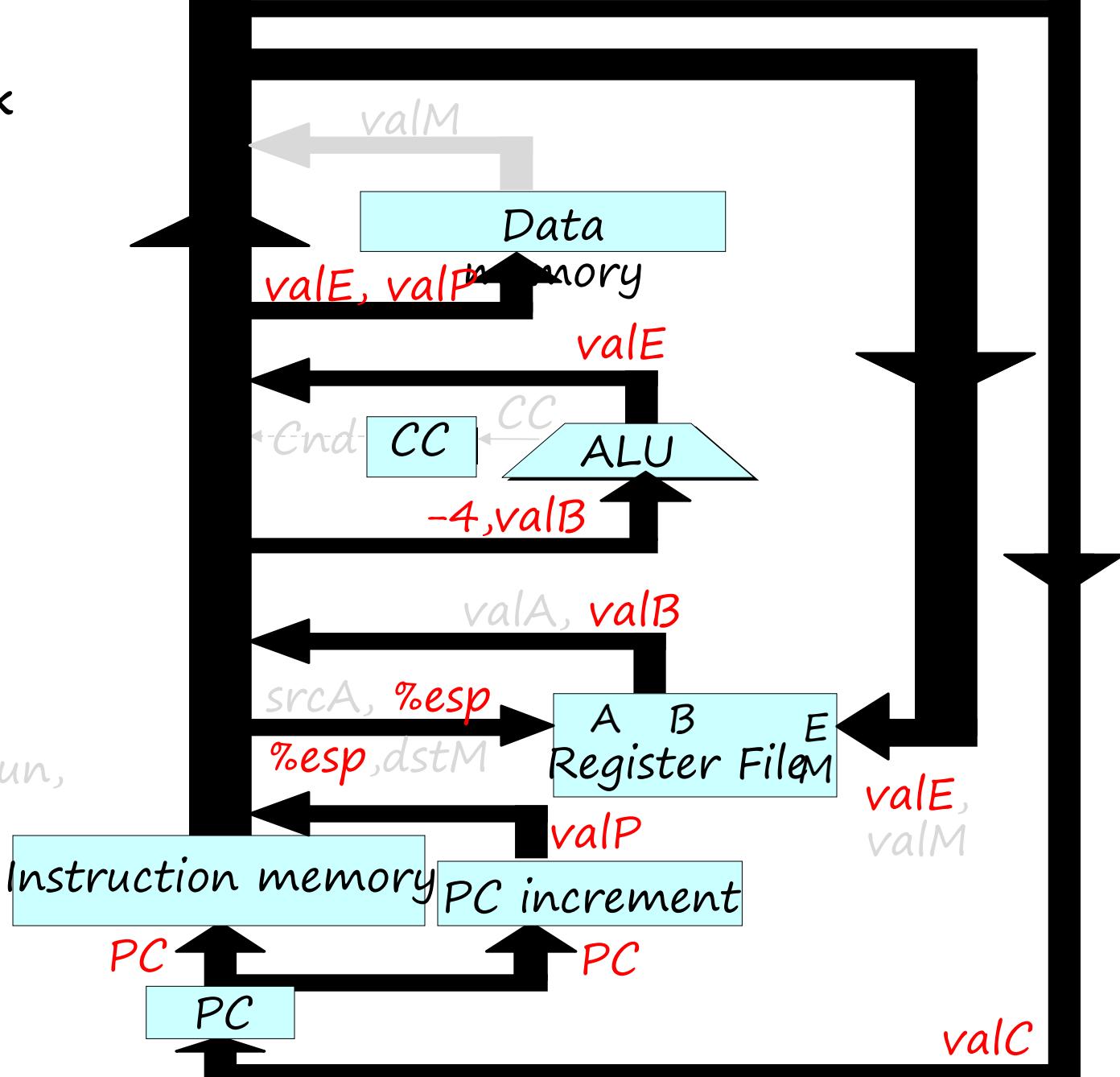
Memory

Execute

Decode

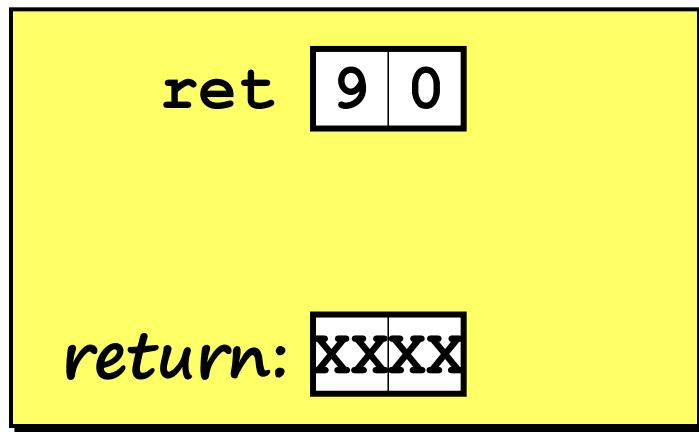
icode:ifun,  
rA:rB

Fetch



# Executing ret

---



# Executing ret

---

- Fetch
  - Read 1 byte
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 4
- Memory
  - Read return address from old stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to return address

# Stage Computation: ret

	ret icode:ifun $\leftarrow M_1[PC]$	
Fetch		Read instruction byte
Decode	$valA \leftarrow R[\%esp]$ $valB \leftarrow R[\%esp]$	Read operand stack Reinteroperand stack Point to current stack pointer
Execute	$valE \leftarrow valB + 4$	
Memory	$valM \leftarrow M_4[valA]$	Read return address
Write back	$R[\%esp] \leftarrow valE$	Update stack pointer
PC update	$PC \leftarrow valM$	Set PC to return address

- Use ALU to increment stack pointer

PC

Write back

Memory

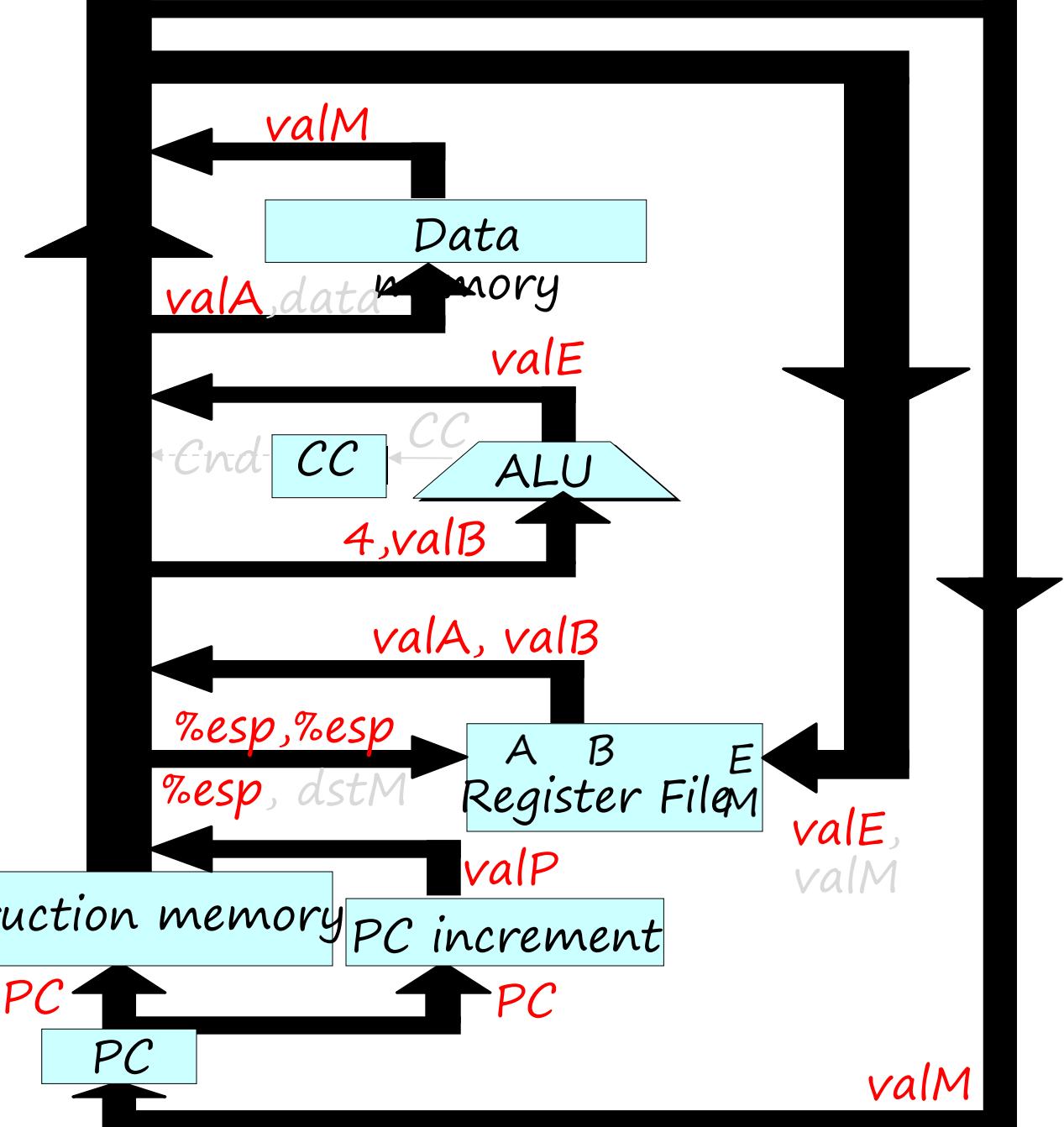
Execute

Decode

icode:ifun,

rA:rB

Fetch



# Executing cmovXX

---

cmovXX rA, rB

2	fn	rA	rB
---	----	----	----

- Fetch
  - Read 2 bytes
- Decode
  - Read operand register rA
- Execute
  - Determine whether to move
- Memory
  - Do nothing
- Write back
  - Update register rB if move taken
- PC Update
  - Increment PC by 2

# Stage Computation: cmovXX

---

	$\text{cmovXX } rA, rB$	
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$  $\text{valP} \leftarrow PC+2$	Read instruction byte Read register byte  Compute next PC
Decode	$\text{valA} \leftarrow R[rA]$	Read operand A
Execute	$\text{valE} \leftarrow O + \text{valA}$ $\text{Cnd} \leftarrow \text{Cond}(CC, \text{ifun})$	Perform ALU operation Take move?
Memory		
Write back	$\text{Cnd} ? R[rB] \leftarrow \text{valE} : -$	Write back result if move taken
PC update	$PC \leftarrow \text{valP}$	Update PC

PC

Write back

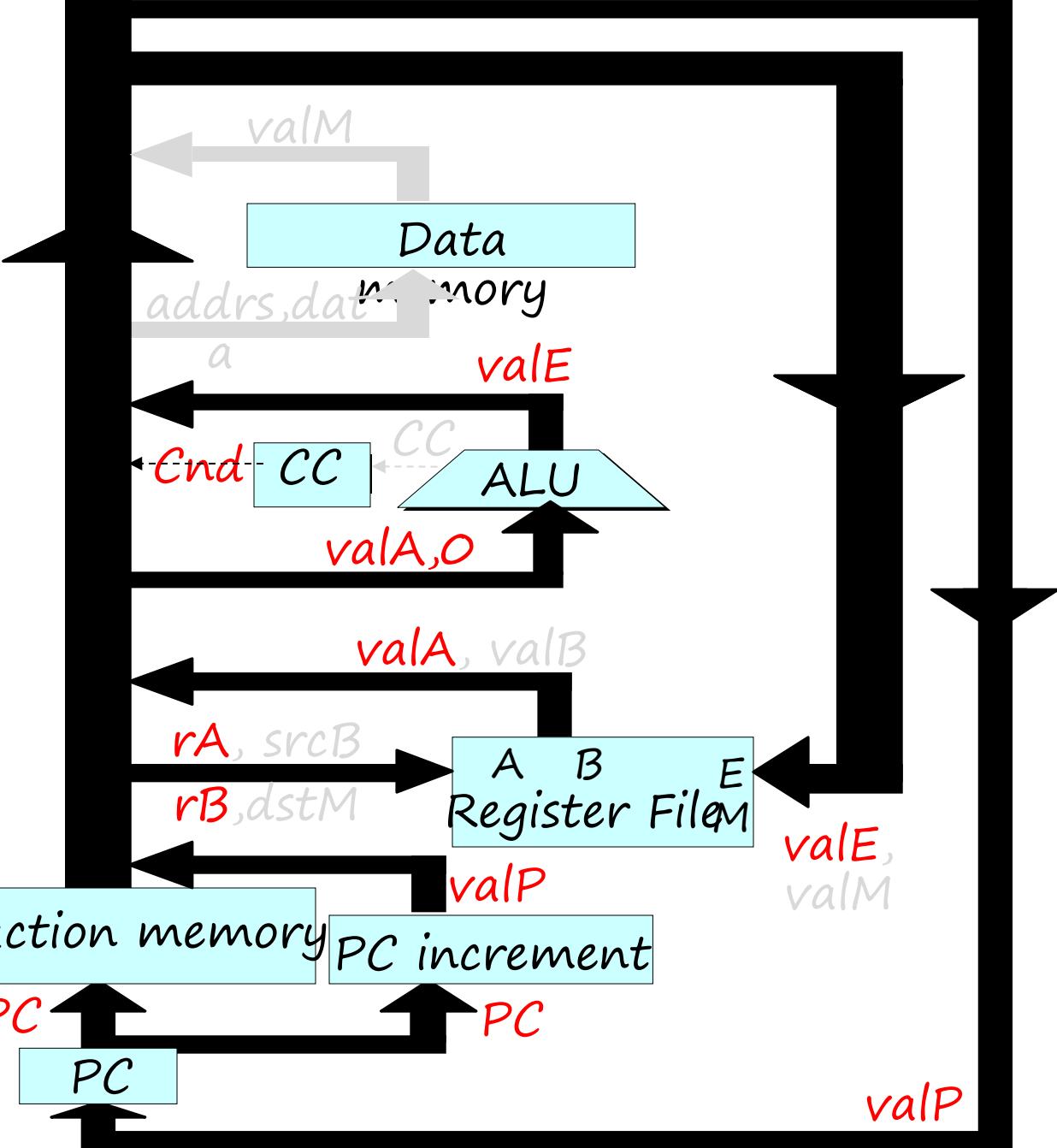
Memory

Execute

Decode

icode:ifun,  
rA:rB

Fetch<sup>IC</sup>



# Computation Steps

---

		OP1 rA, rB	
Fetch	icode,ifun	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	rA,rB	$rA:rB \leftarrow M_1[PC+1]$	Read register byte
	valC		Read constant word
	valP	$valP \leftarrow PC+2$	Compute next PC
Decode	valA, srcA	$valA \leftarrow R[rA]$	Read operand A
	valB, srcB	$valB \leftarrow R[rB]$	Read operand B
Execute	valE	$valE \leftarrow valB \text{ OP } valA$	Perform ALU operation
	Cond code	Set CC	Set condition code reg.
Memory	valM		Read/Write Memory
Write Back	dstE	$R[rB] \leftarrow valE$	Write back ALU result
	dstM		Write back Memory
PC update	PC	$PC \leftarrow valP$	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

# Computation Steps

		OP1 rA, rB	call Dest
Fetch	icode,ifun	$icode:ifun \leftarrow M_1[PC]$	$icode:ifun \leftarrow M_1[PC]$
	rA,rB	$rA:rB \leftarrow M_1[PC+1]$	
	valC		$valC \leftarrow M_4[PC+1]$
	valP	$valP \leftarrow PC+2$	$valP \leftarrow PC+5$
Decode	valA, srcA	$valA \leftarrow R[rA]$	
	valB, srcB	$valB \leftarrow R[rB]$	$valB \leftarrow R[%esp]$
Execute	valE	$valE \leftarrow valB \text{ OP } valA$	$valE \leftarrow valB + -4$
	Cond code	Set CC	
Memory	valM		$M_4[valE] \leftarrow valP$
Write Back	dstE	$R[rB] \leftarrow valE$	$R[%esp] \leftarrow valE$
	dstM		
PC update	PC	$PC \leftarrow valP$	$PC \leftarrow valC$

- All instructions follow same general pattern
- Differ in what gets computed on each step

PC

Write back

Memory

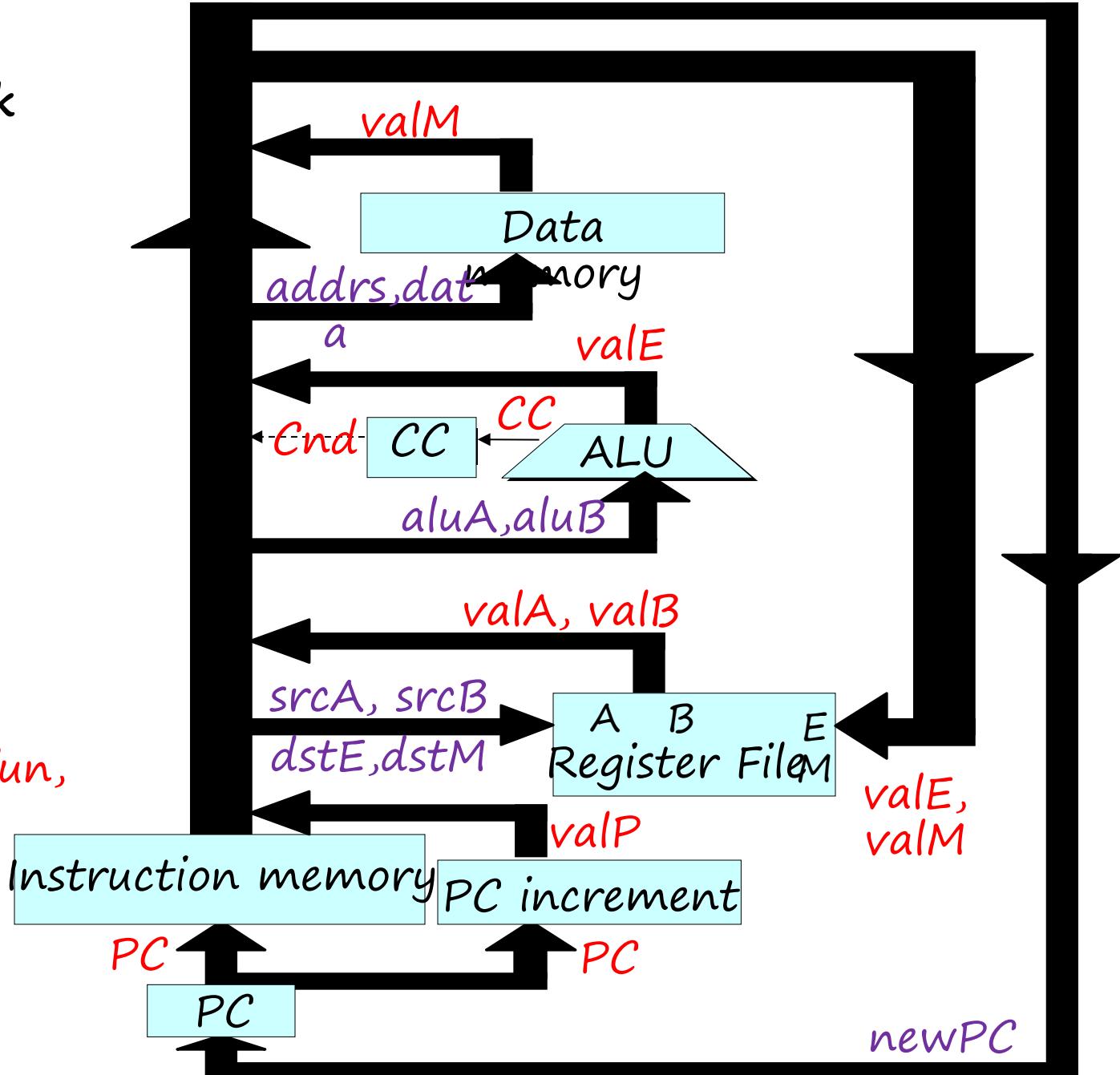
Execute

Decode

icode:ifun,  
rA:rB

Fetch

valIC



# Determinate Values

---

## Fetch

PC	Program Count
icode	Instr. Code
ifun	Instr. Function
rA	Instr. Register A
rB	Instr. Register B
valC	Instr. Constant
valP	Incremented PC

## Decode

valA	Register value A
valB	Register value B

## Execute

valE	ALU result
CC	Condition Code
Cnd	Condition Flag

## Memory

valM Value from Memory

## Write back

valE	ALU result
valM	Value from Memory

# Indeterminate Values

## Decode

srcA location of valA

rA, %esp

srcB location of valB

rB, %esp

## Execute

aluA input A of ALU

valA, valC, +4, -4

aluB input B of ALU

valB, 0

## Memory

addr address of memory

valA, valE

data data of Memory

valA, valP

## Write back

dstE dest. of ALU result

rB, %esp

dstM dest of Memory

rA

## PC

newPC next of PC

valP, valC, valM

PC

Write back

Memory

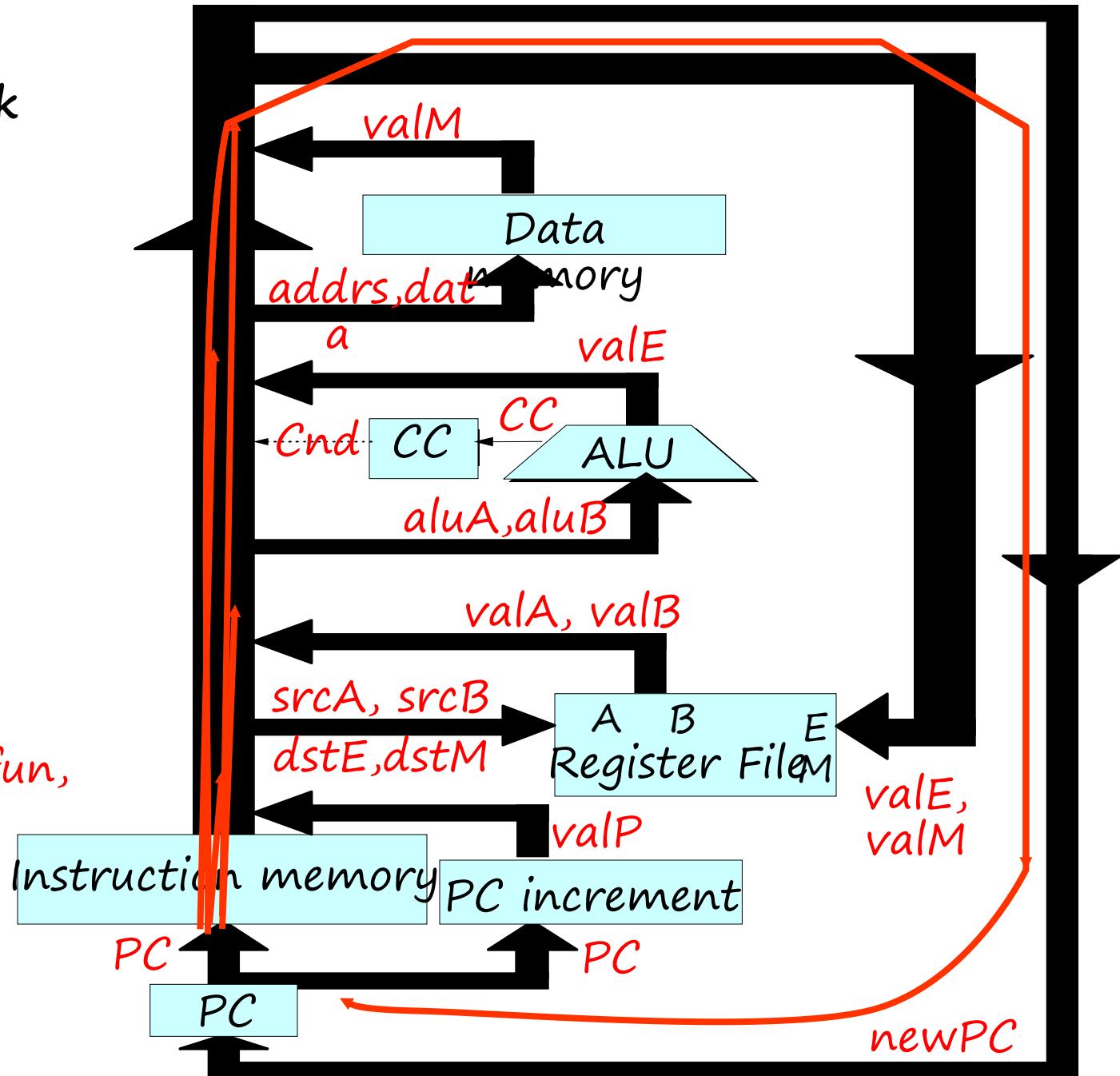
Execute

Decode

icode:ifun,  
rA:rB

Fetch

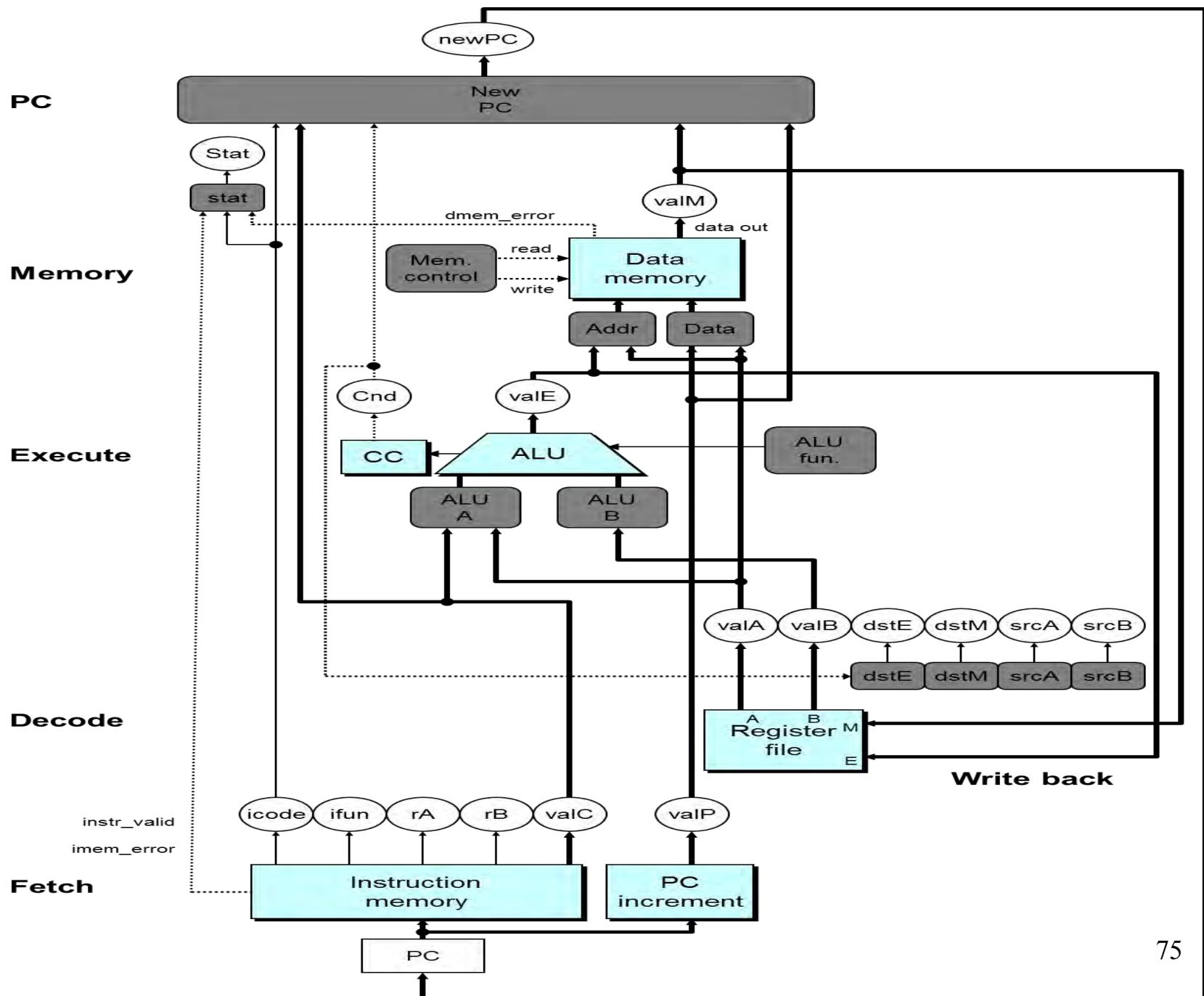
hIC



# SEQ Semantics

---

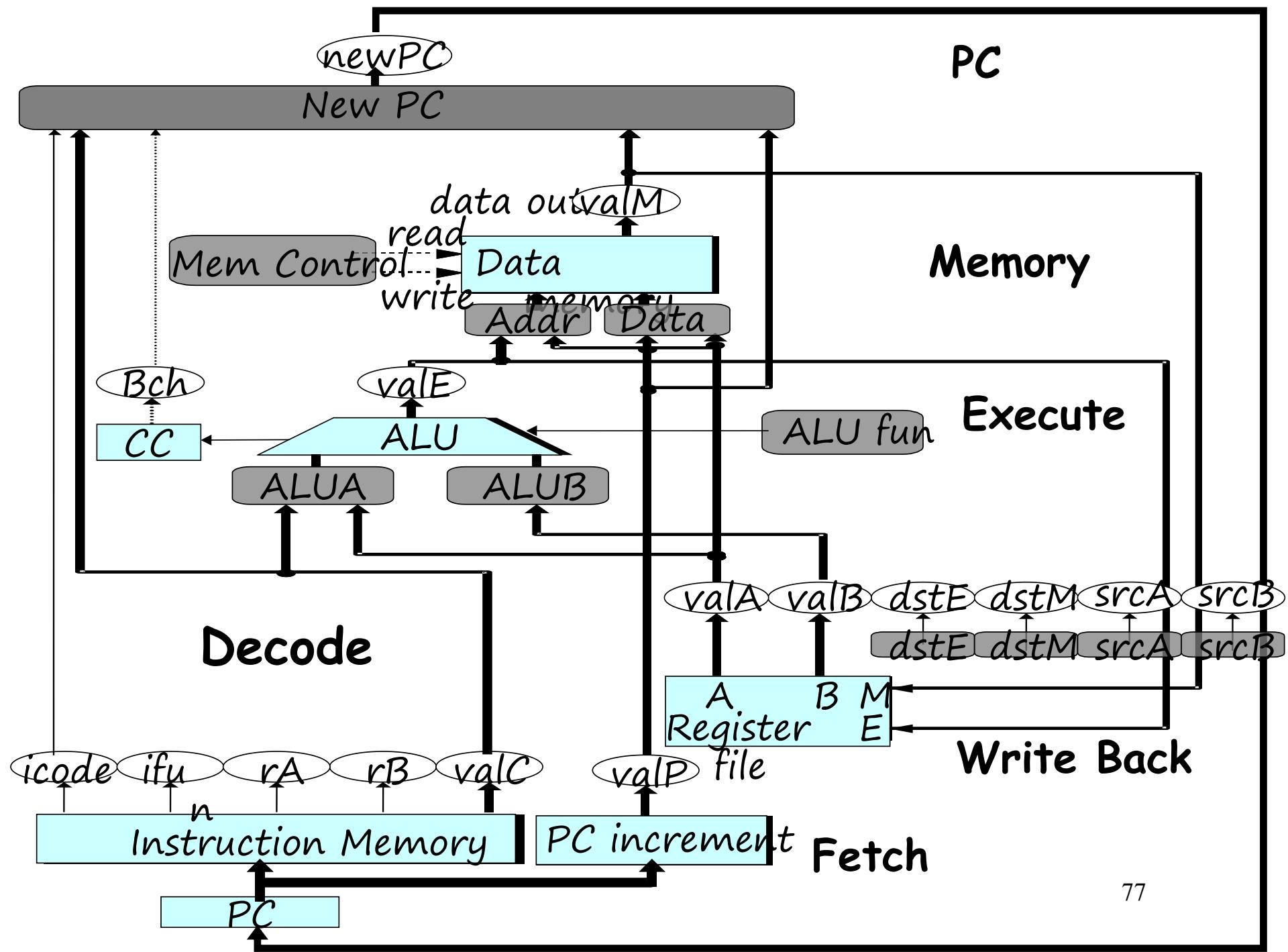
- Achieve the same effect as a sequential execution of the assignment shown in the tables of Figures 4.18 to 4.21
  - Though all of the state updates occur simultaneously at the clock rises to the next cycle.
  - A problem: `popl %esp` need to sequentially write two registers. So the register file control logic must process it.



# SEQ Summary

---

- Sequential Processor (SEQ)
  - Combinational logic + Sequential logic
  - Instruction executes in sequential
  - Every Instruction finished in one cycle.
  - Express every instruction as series of simple steps
    - Fetch, Decode, Execute, Memory, Write Back, PC



# SEQ CPU Implementation

# Outline

---

- SEQ Implementation
- Suggested Reading 4.3.2, 4.3.4

# What we will discuss today?

---

- The implementation of a sequential CPU (SEQ)
  - Every Instruction finished in one cycle.
  - Instruction executes in sequential
  - No two instruction execute in parallel or overlap

# SEQ Hardware Structure

---

- Stages
  - Fetch: Read instruction from memory
  - Decode: Read program registers
  - Execute: Compute value or address
  - Memory: Read or write data
  - Write Back: Write program registers
  - PC: Update program counter

PC

Write back

Memory

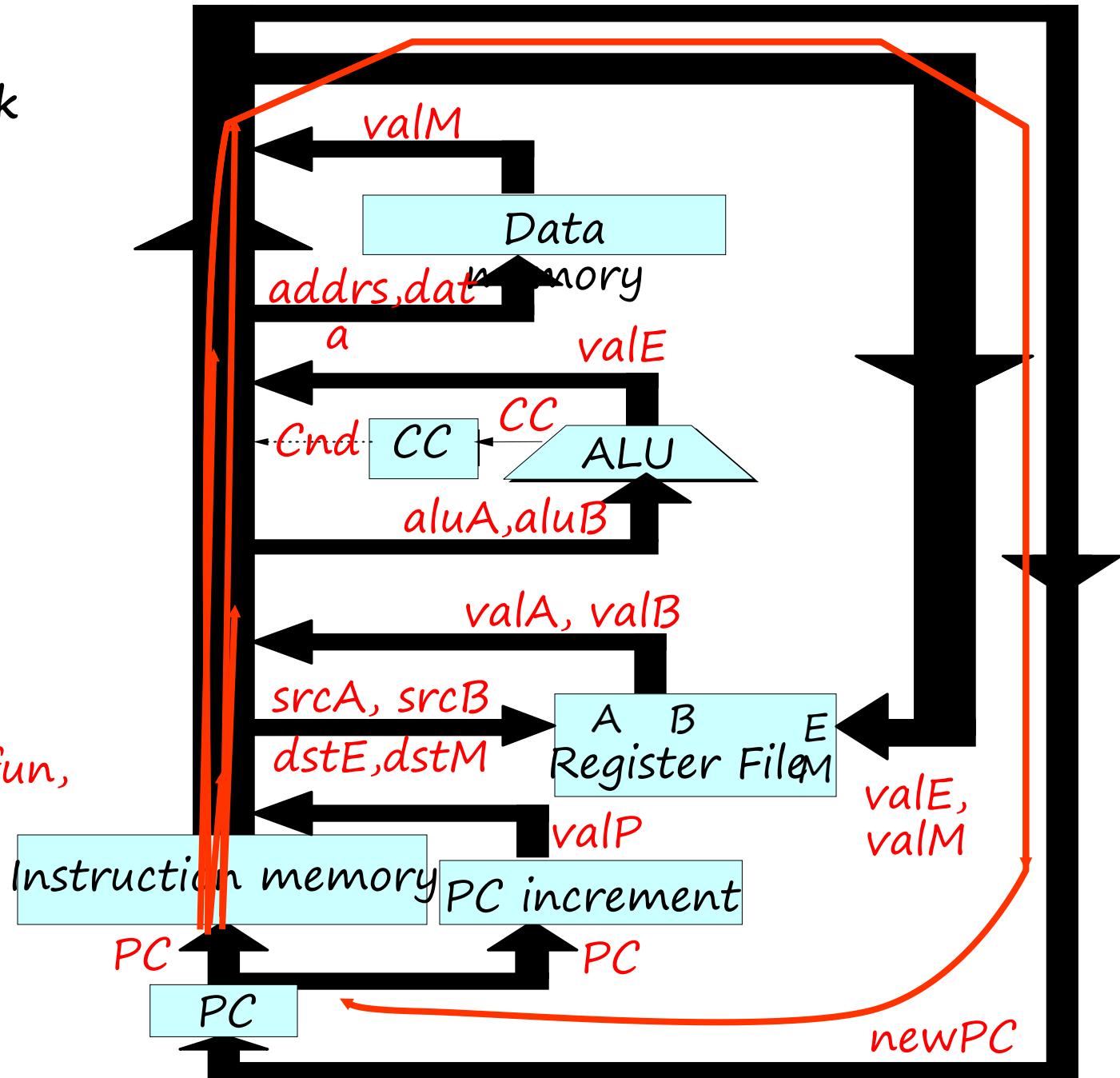
Execute

Decode

icode:ifun,  
rA:rB

Fetch

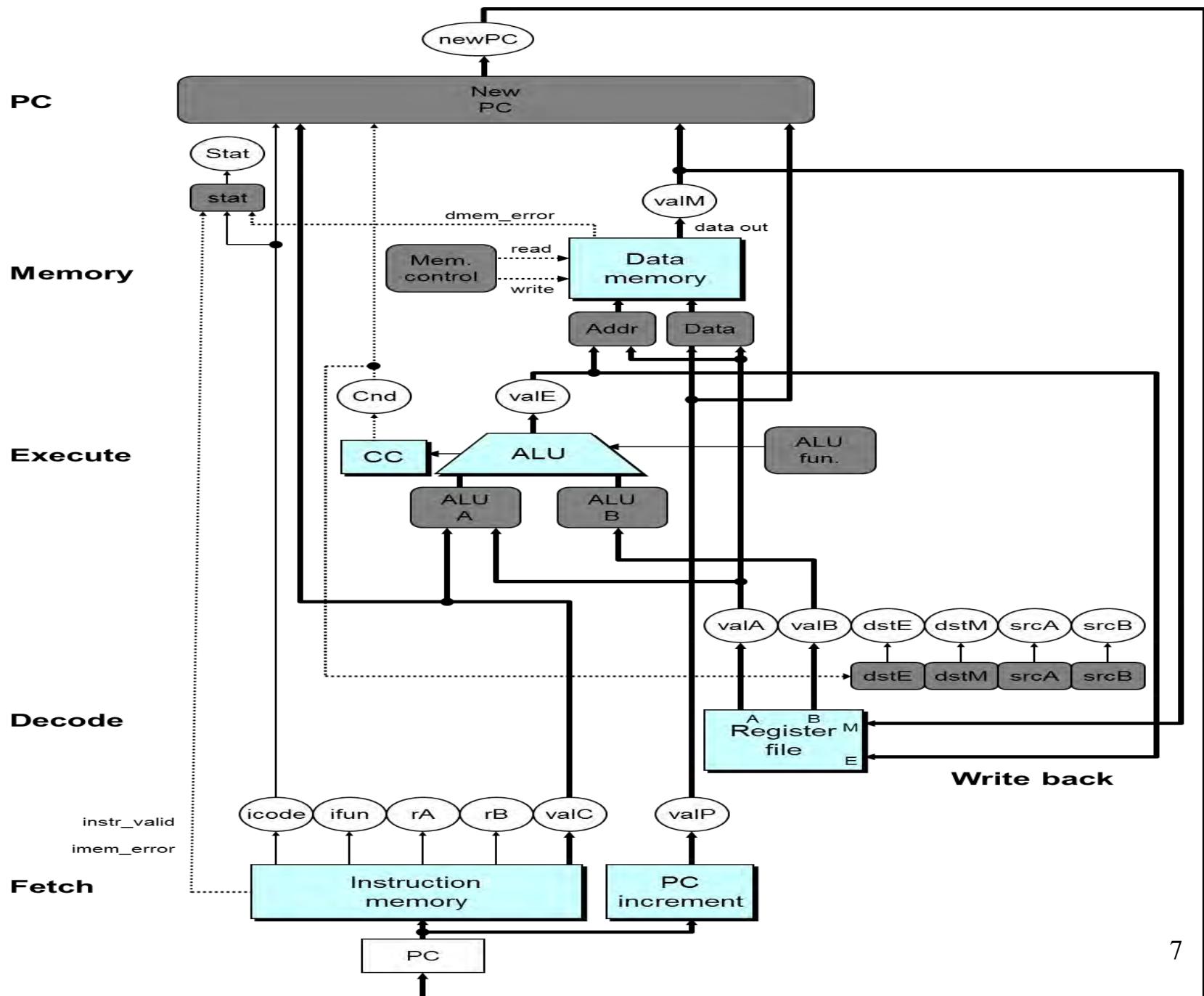
hIC



# Difference between semantics and implementation

---

- ISA
  - Every stage may update some states, these updates occur sequentially
- SEQ
  - All the state update operations occur simultaneously at clock rising

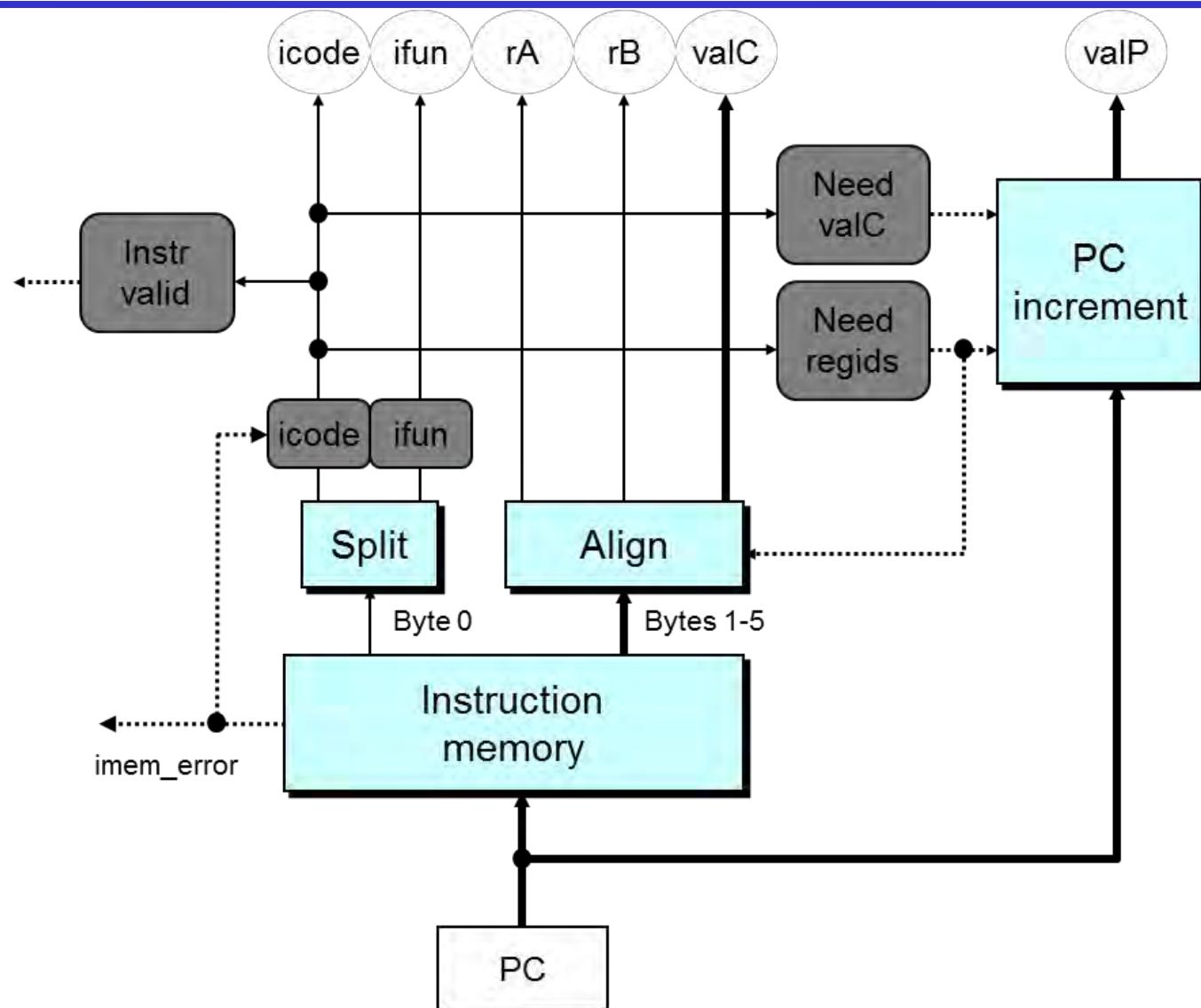


# SEQ Hardware

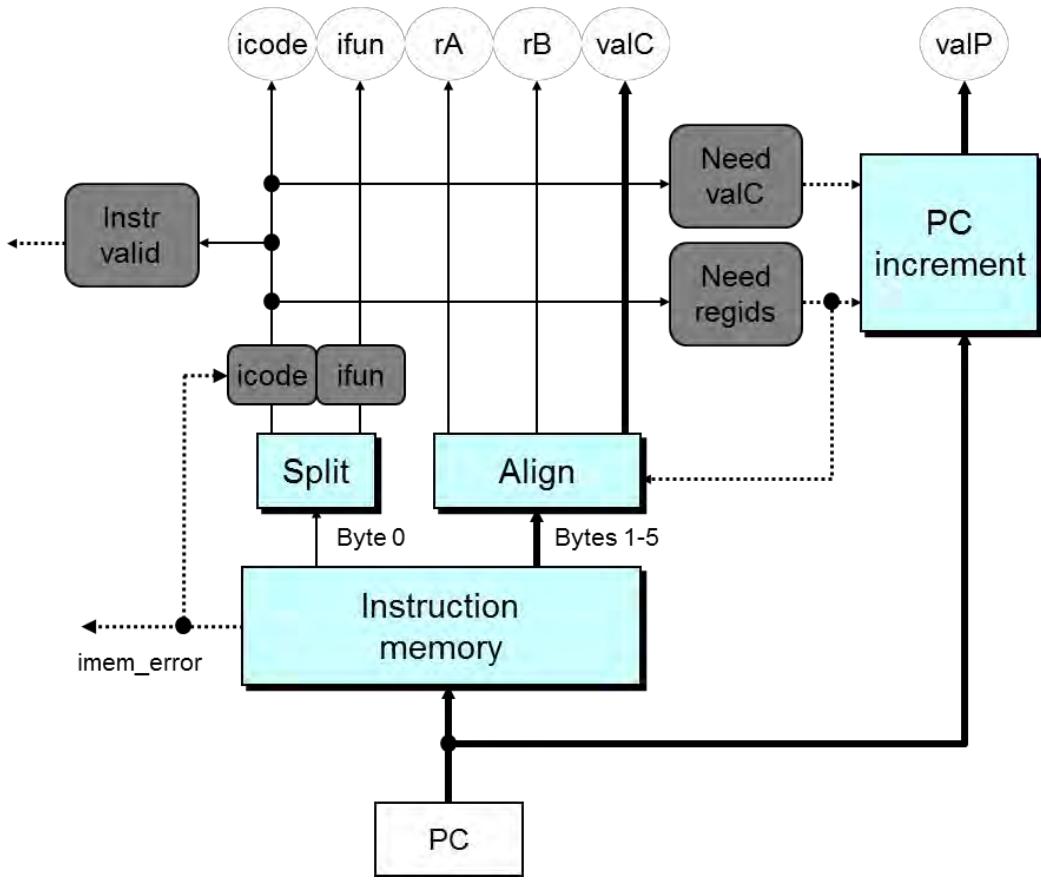
---

- Blue boxes:  predesigned hardware blocks
  - e.g., memories, ALU
- Gray boxes:  control logic
  - Describe in HCL
- White ovals:  labels for signals
- Thick lines:  32-bit word values
- Thin lines:  4-8 bit values
- Dotted lines:  1-bit values

# Fetch Logic

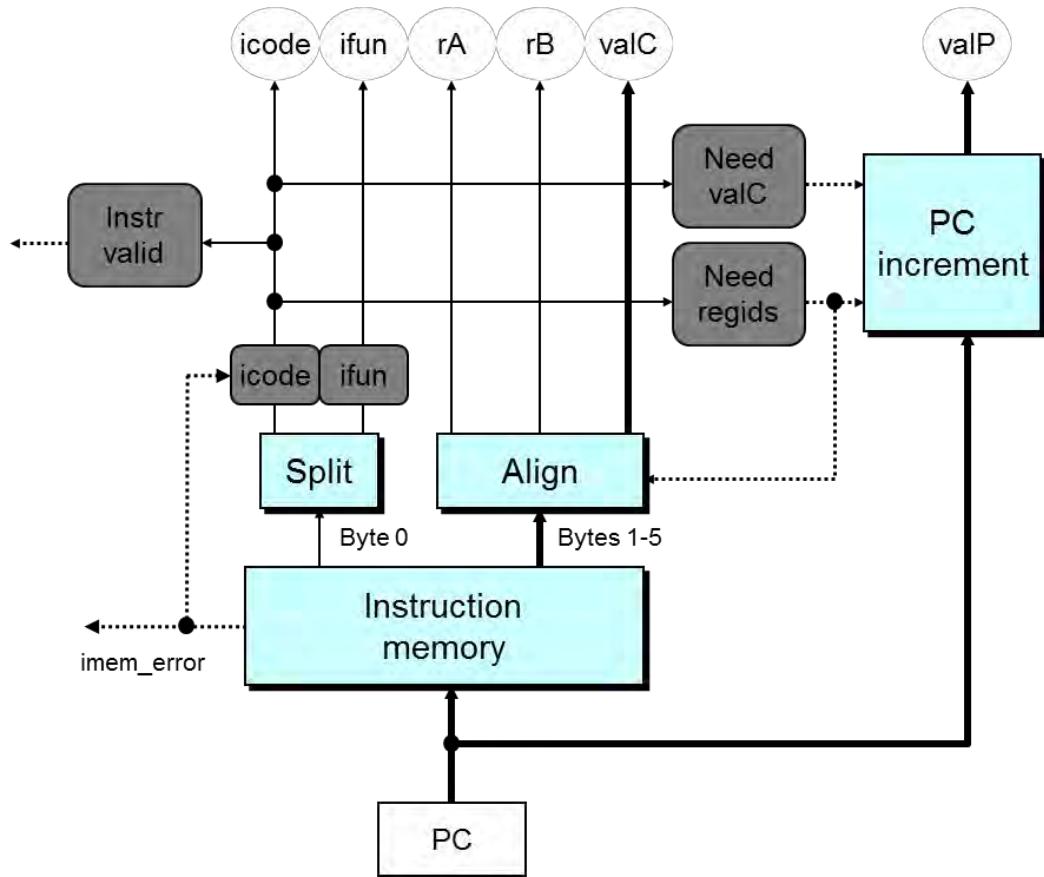


# Fetch Logic



- Predefined Blocks
  - PC: Register containing PC
  - Instruction memory: Read 6 bytes (PC to PC+5)
  - Split: Divide instruction byte into icode and ifun
  - Align: Get fields for rA, rB, and valC

# Fetch Logic



- ## Control Blocks

- Instr. Valid: Is this instruction valid?
- Need regids: Does this instruction have a register bytes?
- Need valC: Does this instruction have a constant word?
- icode|ifun: Set to "nop" if imem\_error

# Some Macros

---

Name	Value	Meaning
INOP	0	Code for <code>nop</code> instruction
IHALT	1	Code for <code>halt</code> instruction
IRRMOVL	2	Code for <code>rrmovl</code> instruction
IIRMOVL	3	Code for <code>irmovl</code> instruction
IRMMOVL	4	Code for <code>rmmovl</code> instruction
IMRMOVL	5	Code for <code>mrmovl</code> instruction
IOPL	6	Code for integer op instructions
IJXX	7	Code for jump instructions
.....	.....	.....
IPOPL	B	Code for <code>popl</code> instruction

# Some Macros

---

Name	Value	Meaning
RESP	6	Register ID for %esp
RNONE	F	Indicates no register file access
ALUADD	0	Function for addition operation
SAOK	1	Status code for normal operations
SADR	2	Status code for address exception
SINS	3	Status code for illegal instruction Exception
SHLT	4	Status code for halt

nop	<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0			
0	0					
halt	<table border="1"><tr><td>1</td><td>0</td></tr></table>	1	0			
1	0					
rrmovl rA, rB	<table border="1"><tr><td>2</td><td>0</td><td>rA</td><td>rB</td></tr></table>	2	0	rA	rB	
2	0	rA	rB			
irmovl V, rB	<table border="1"><tr><td>3</td><td>0</td><td>8</td><td>rB</td><td>V</td></tr></table>	3	0	8	rB	V
3	0	8	rB	V		
rmmovl rA, D(rB)	<table border="1"><tr><td>4</td><td>0</td><td>rA</td><td>rB</td><td>D</td></tr></table>	4	0	rA	rB	D
4	0	rA	rB	D		
mrmovl D(rB), rA	<table border="1"><tr><td>5</td><td>0</td><td>rA</td><td>rB</td><td>D</td></tr></table>	5	0	rA	rB	D
5	0	rA	rB	D		
OPl rA, rB	<table border="1"><tr><td>6</td><td>fn</td><td>rA</td><td>rB</td></tr></table>	6	fn	rA	rB	
6	fn	rA	rB			
jXX Dest	<table border="1"><tr><td>7</td><td>fn</td><td>Dest</td></tr></table>	7	fn	Dest		
7	fn	Dest				
call Dest	<table border="1"><tr><td>8</td><td>0</td><td>Dest</td></tr></table>	8	0	Dest		
8	0	Dest				
ret	<table border="1"><tr><td>9</td><td>0</td></tr></table>	9	0			
9	0					
pushl rA	<table border="1"><tr><td>A</td><td>0</td><td>rA</td><td>8</td></tr></table>	A	0	rA	8	
A	0	rA	8			
popl rA	<table border="1"><tr><td>B</td><td>0</td><td>rA</td><td>8</td></tr></table>	B	0	rA	8	
B	0	rA	8			

need\_regs

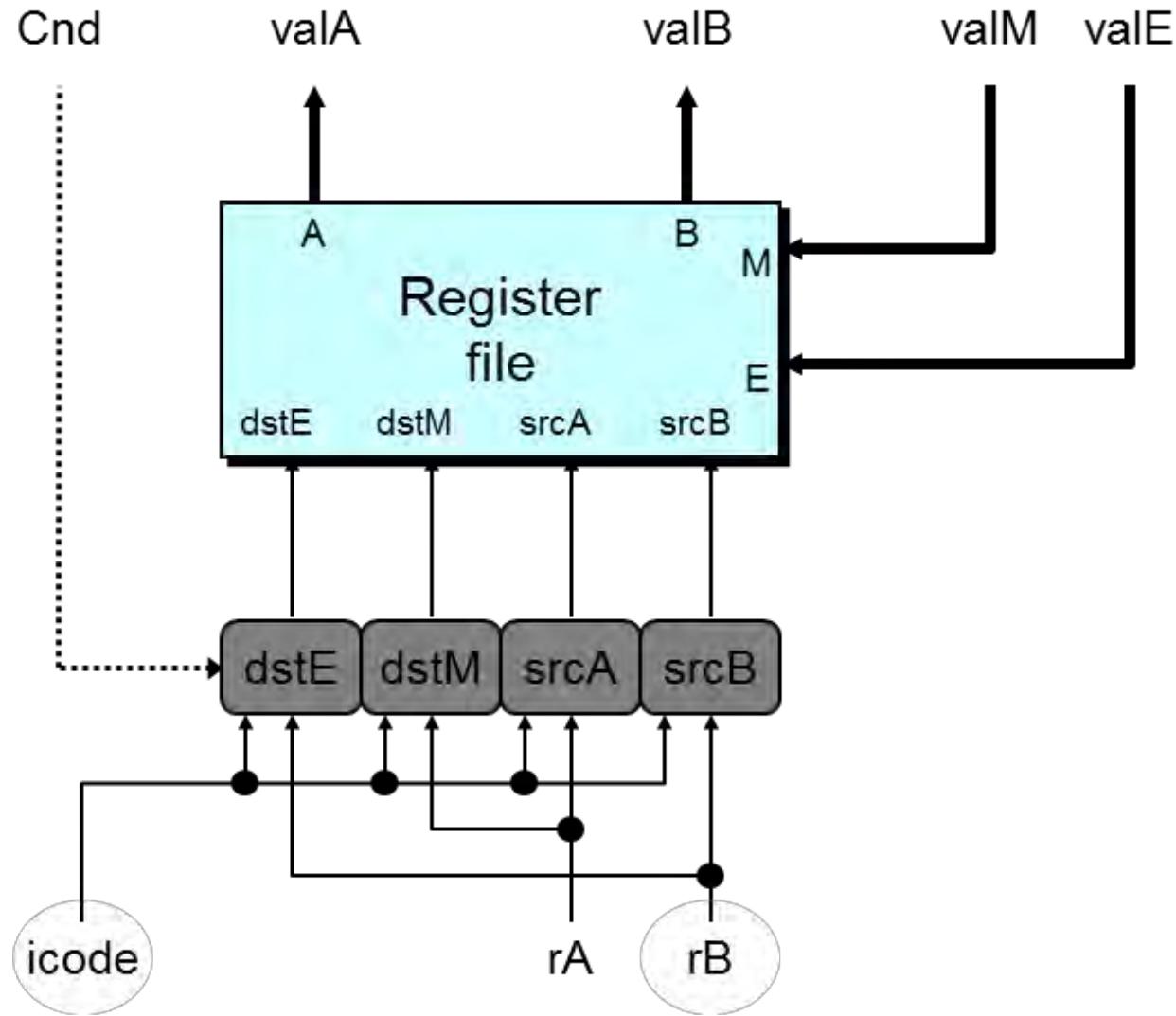
# Fetch Control Logic

---

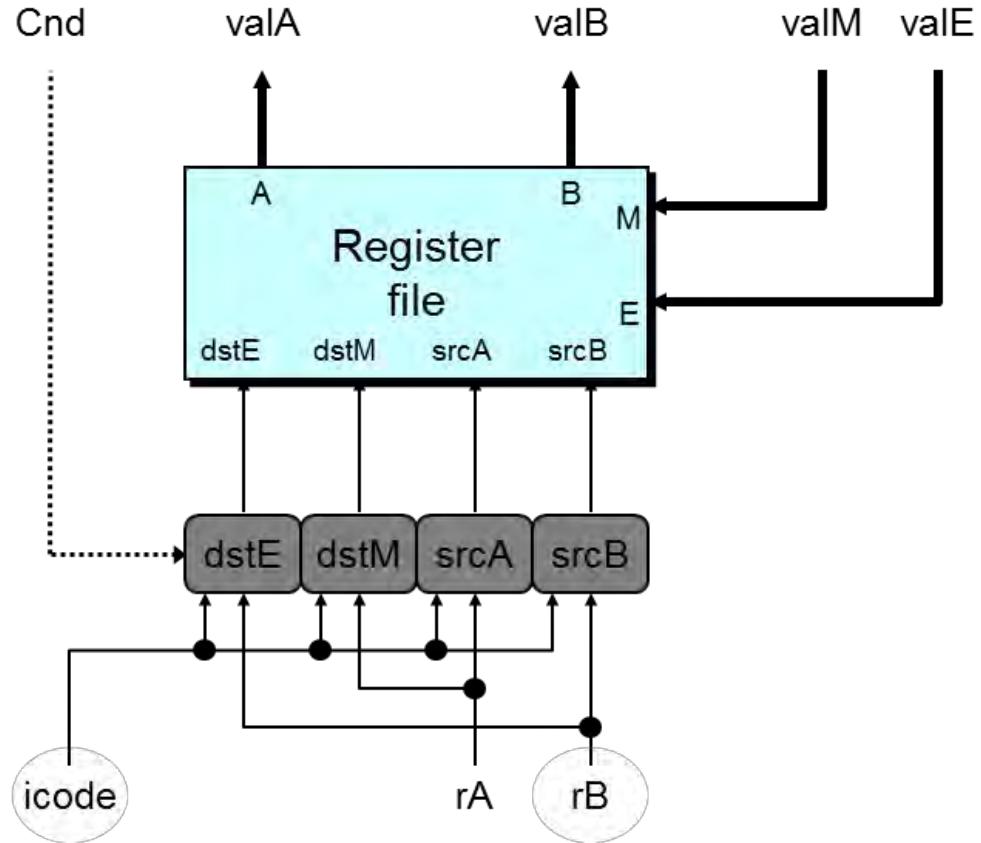
```
bool need_regs = icode in
{ IRRMOVL, IIRMOVL, IRMMOVL,
IMRMOVL, IOPL, IPUSHL, IPOPL } ;

bool instr_valid = icode in
{ INOP, IHALT, IRRMOVL, IIRMOVL,
IRMMOVL, IMRMOVL, IOPL, IJXX,
ICALL, IRET, IPUSHL, IPOPL } ;
```

# Decode & Write-Back Logic

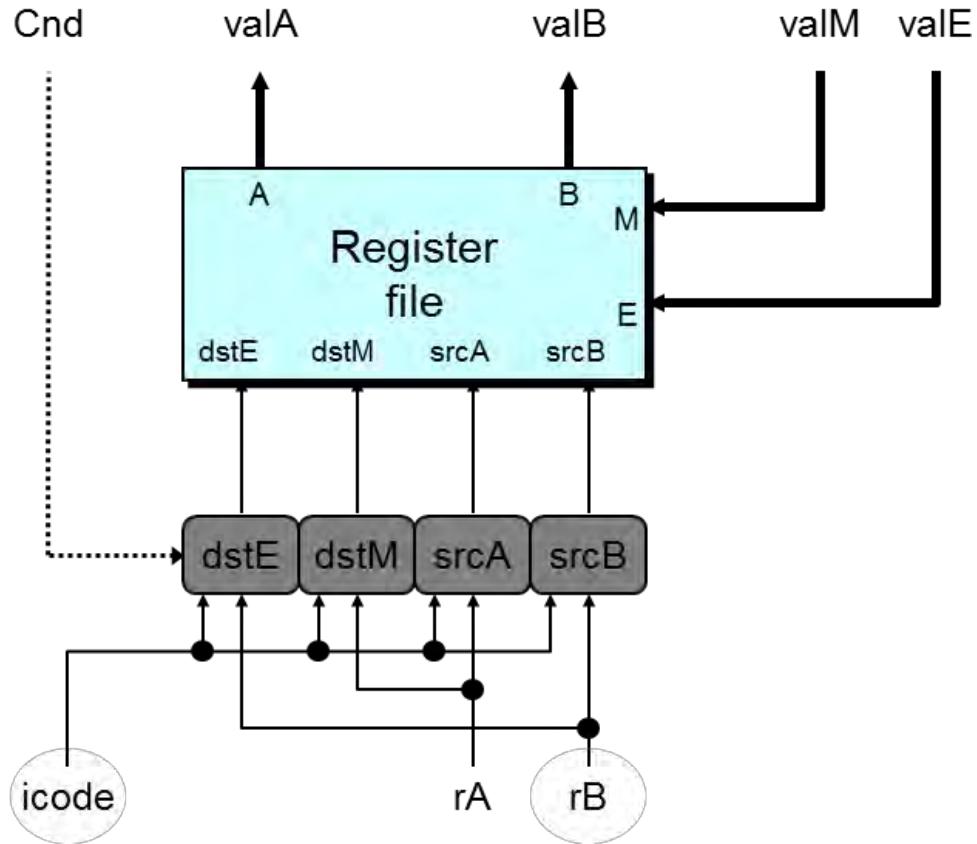


# Decode & Write Back Logic



- Predefined Blocks
  - Registers File
    - Read ports A, B & Write ports E, M
    - Addresses are register IDs or F (no access)

# Decode & Write Back Logic



## • Control Logic

- srcA: read port address for valA {rA, %esp}
- srcB: read port address for valB {rB, %esp}
- dstE: write port address for valE {rB, %esp}
- dstM: write port address for valM {rA}
- Cnd: used to decide whether set valE (cmovXX)<sub>18</sub>

# A Source

---

	op1 rA, rB	
Decode	valA $\leftarrow R[rA]$	Read operand A
	rmmovl rA, D(rB)	
Decode	valA $\leftarrow R[rA]$	Read operand A
	pop1 rA	
Decode	valA $\leftarrow R[\%esp]$	Read stack pointer
	jXX Dest	
Decode		No operand
	call Dest	
Decode		No operand
	ret	
Decode	valA $\leftarrow R[\%esp]$	Read stack pointer

# A Source

---

```
int srcA = [
    icode in { IRRMOVL, IRMMOVL,
                IOPL, IPUSHL } : rA;
    icode in { IPOPL, IRET } : RESP;
    1 : RNONE; # Don't need register
];
```

# E Destination

---

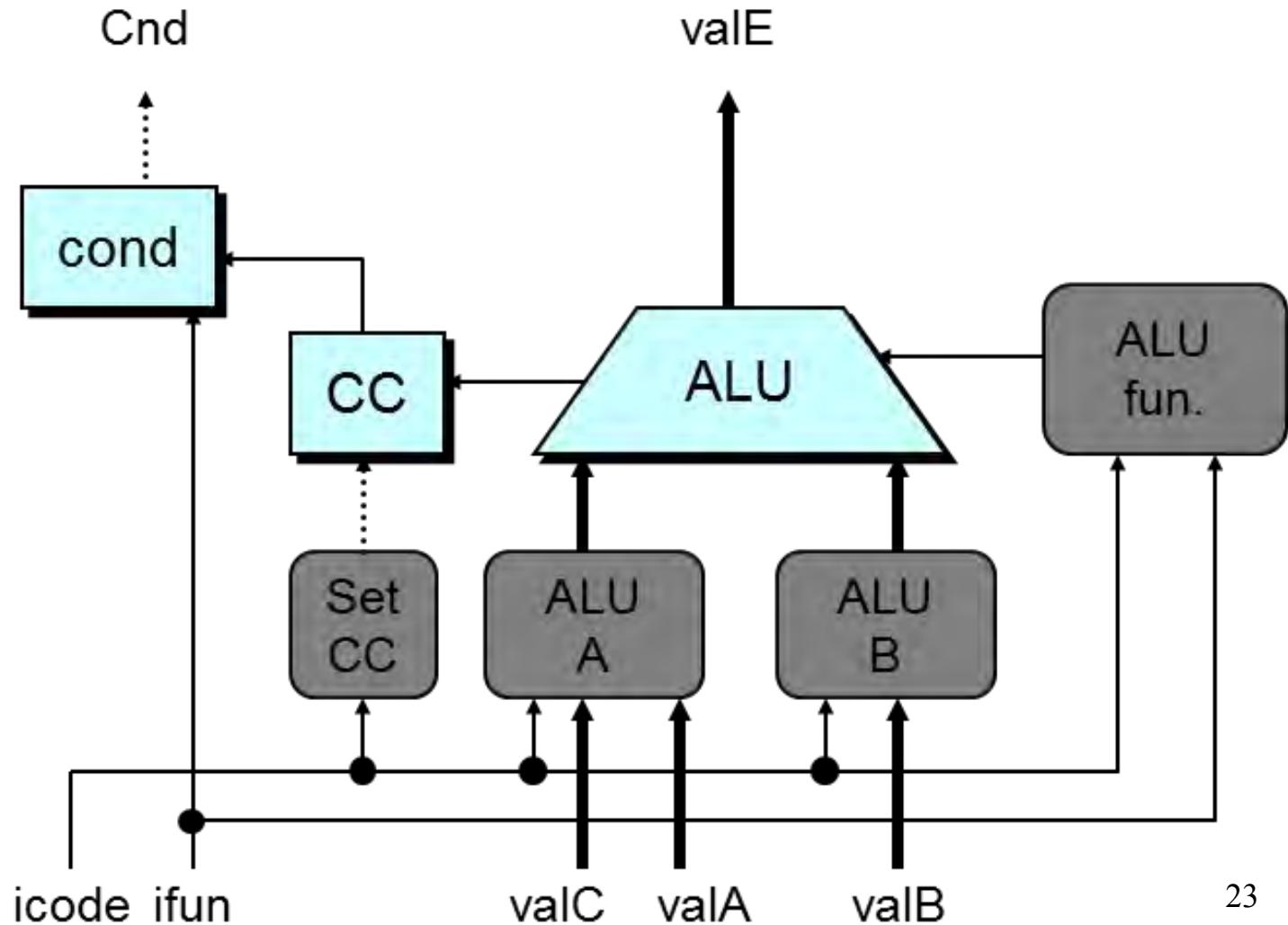
	opl rA, rB	
Write-back	$R[rB] \leftarrow valE$	Write back result
	rmmovl rA, D(rB)	
Write-back		None
	popl rA	
Write-back	$R[%esp] \leftarrow valE$	Update stack pointer
	jXX Dest	
Write-back		None
	call Dest	
Write-back	$R[%esp] \leftarrow valE$	Update stack pointer
	ret	
Write-back	$R[%esp] \leftarrow valE$	Update stack pointer

# E Destination

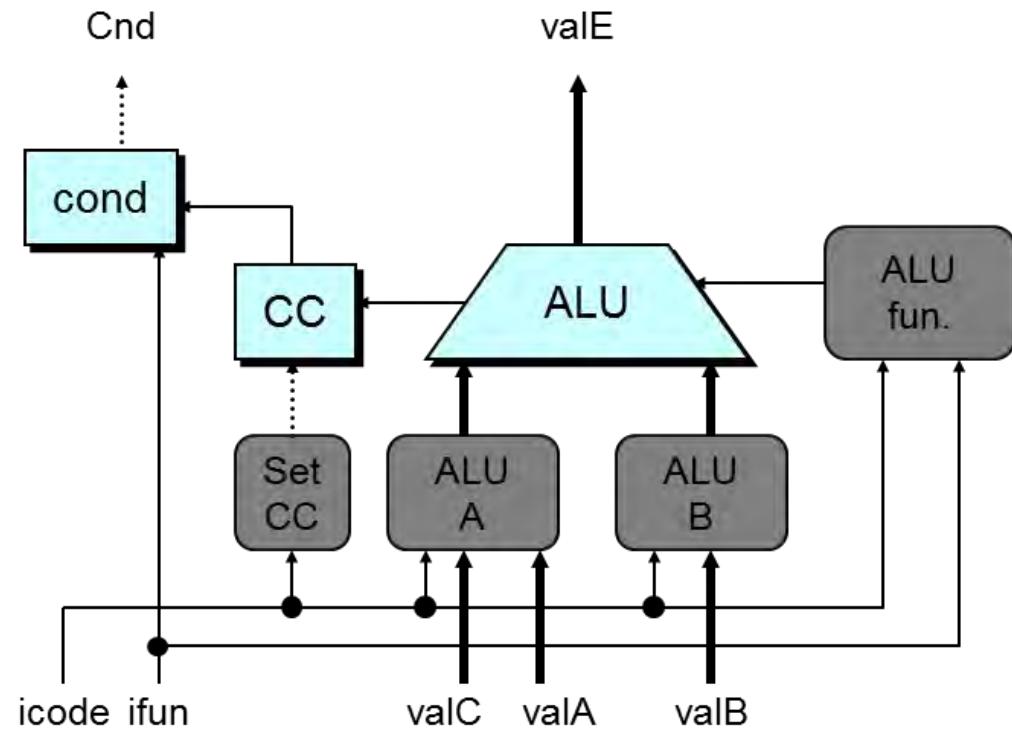
---

```
int dstE = [
    icode in { IRRMOVL, IIRMOVL,
                IOPL} : rB;
    icode in { IPUSHL, IPOPL, ICALL,
                IRET } : RESP;
    1 : RNONE;  # Don't need register
];
// not consider cmovXX
```

# Execute Logic

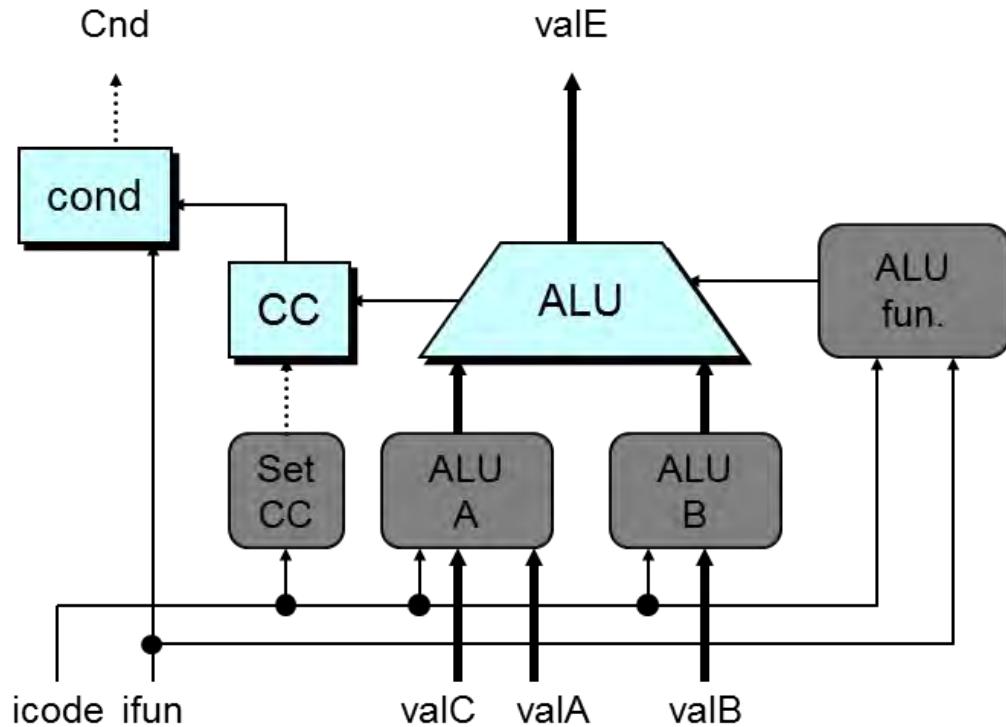


# Execute Logic



- Predefined Blocks
  - ALU: implements 4 required functions and generate condition code values
  - CC: register with 3 condition code bits
  - cond: computes condition flag

# Execute Logic



- Control Logical
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU { $\text{valA}$ ,  $\text{valC}$ , +4, -4}
  - ALU B: Input B to ALU { $\text{valB}$ , 0}
  - ALU fun: What function should ALU compute?

# ALU A Input

	OP1 rA, rB	
Execute	$\text{valE} \leftarrow \text{valB OP valA}$	Perform ALU operation
	rmmovl rA, D(rB)	
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
	popl rA	
Execute	$\text{valE} \leftarrow \text{valB} + 4$	Increment stack pointer
	jxx Dest	
Execute	call Dest	No operation
Execute	$\text{valE} \leftarrow \text{valB} + -4$	Decrement stack pointer
	ret	
Execute	$\text{valE} \leftarrow \text{valB} + 4$	Increment stack pointer

## ALU A Input

---

```
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL }
                                : valC;
    icode in { ICALL, IPUSHL } : -4;
    icode in { IRET, IPOPL }   : 4;
    # Other instructions don't need ALU
];
```

# ALU Operation

---

	op1 rA, rB	
Execute	valE $\leftarrow$ valB $OP$ valA	Perform ALU operation
	rmmovl rA, D(rB)	
Execute	valE $\leftarrow$ valB + valC	Compute effective address
	popl rA	
Execute	valE $\leftarrow$ valB + 4	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	valE $\leftarrow$ valB + -4	Decrement stack pointer
	ret	
Execute	valE $\leftarrow$ valB + 4	Increment stack pointer

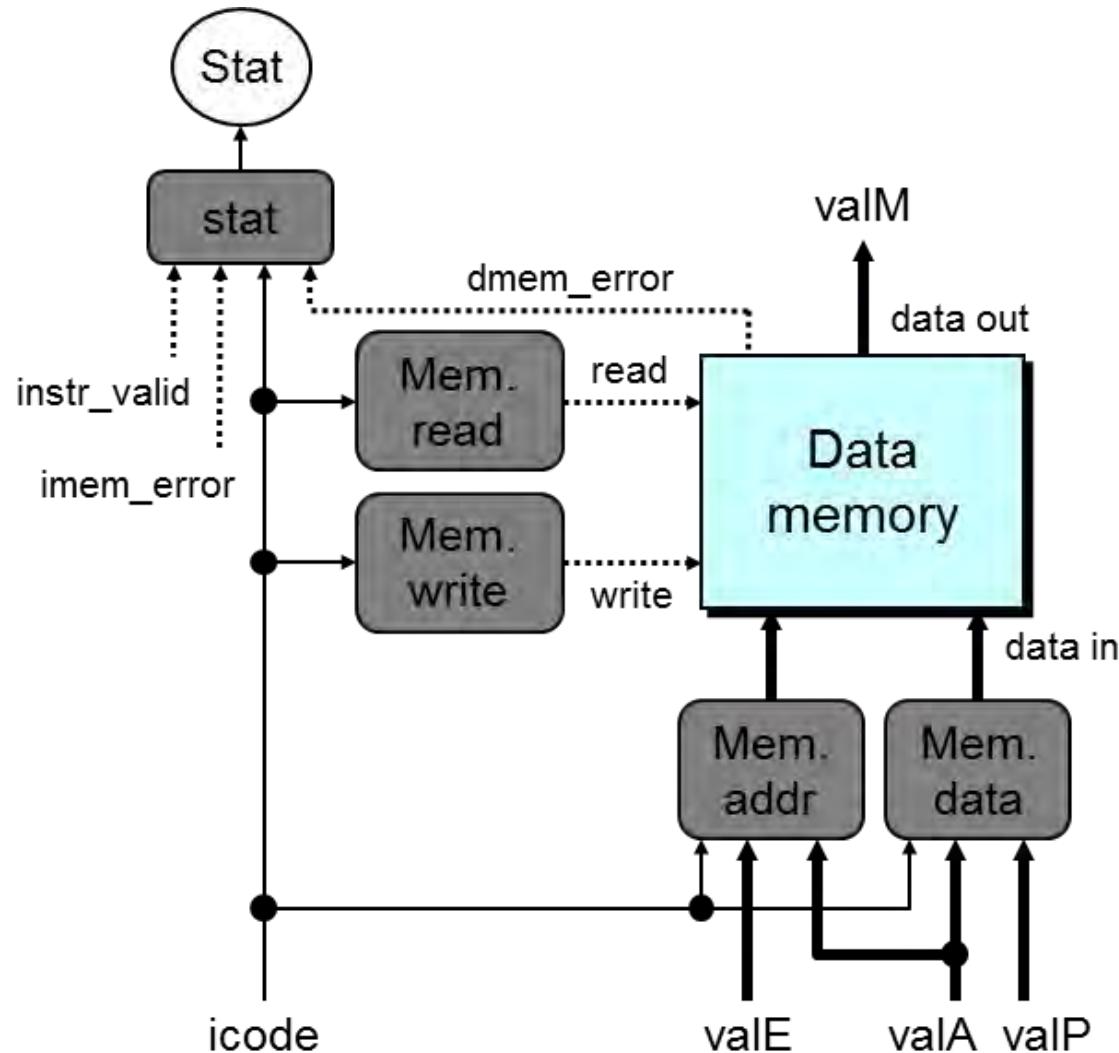
# ALU Operation / Condition Set

---

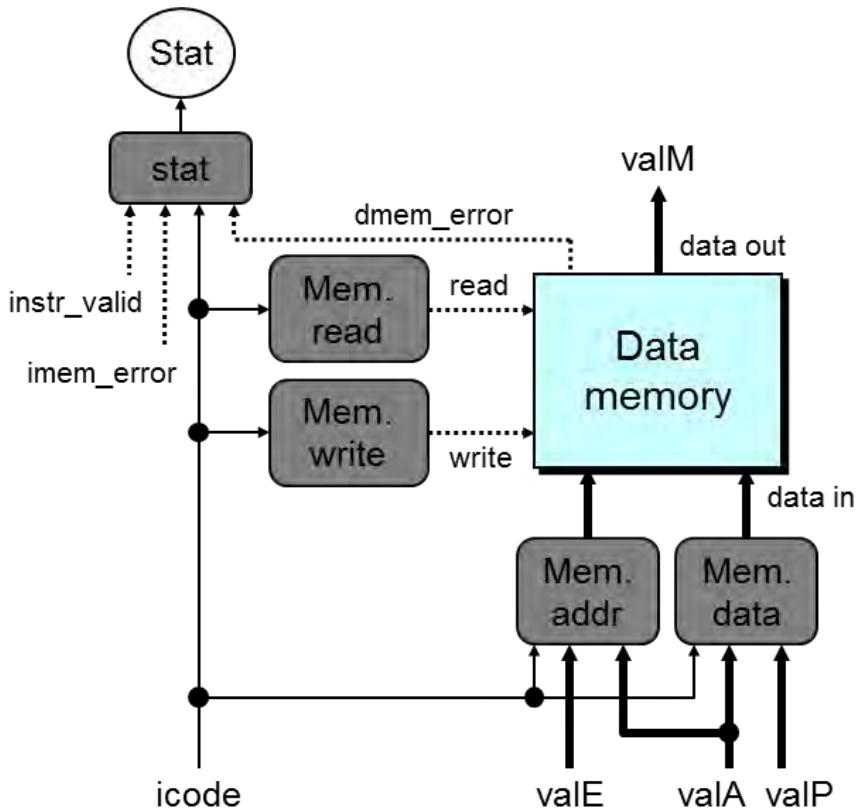
```
int alufun = [  
    icode == IOPL : ifun;  
    1 : ALUADD;  
];
```

```
Bool set_cc = icode in { IOPL };
```

# Memory Logic

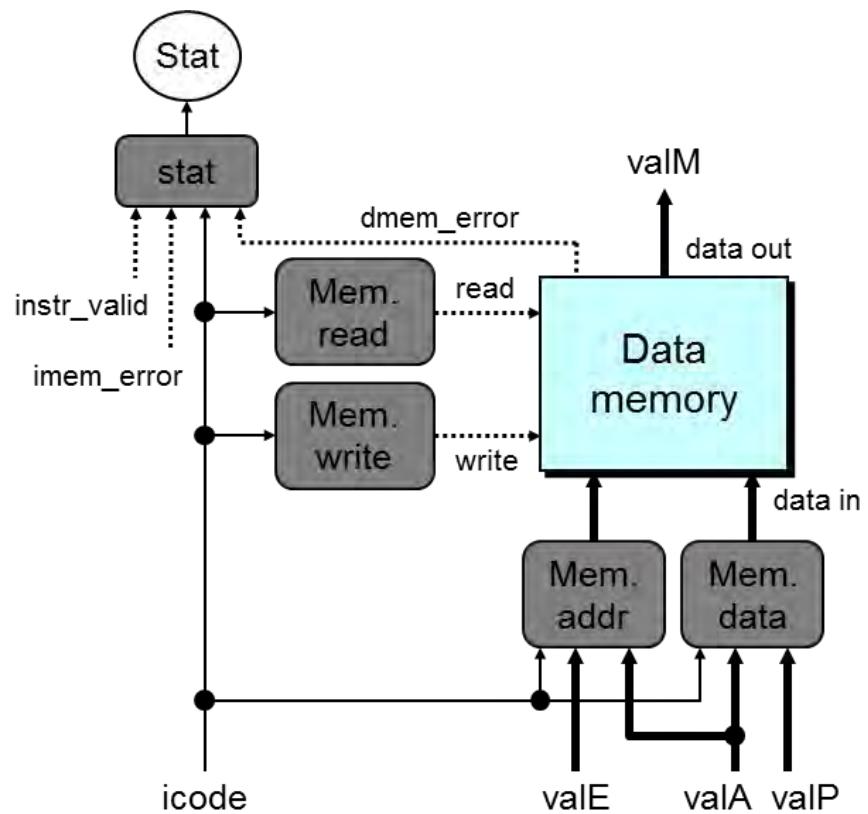


# Memory Logic



- Predefined Blocks
  - Memory: Reads or writes memory word

# Memory Logic



- Control Logical
  - Mem. read: should word be read?
  - Mem. write: should word be written?
  - Mem. addr.: Select address {valA, valE}
  - Mem. data.: Select data {valA, valP}

# Memory Address

---

	op1 rA, rB	
Memory		No operation
Memory	rmmovl rA, D(rB) $M_4[\text{valE}] \leftarrow \text{valA}$	Write value to memory
Memory	pop1 rA $\text{valM} \leftarrow M_4[\text{valA}]$	Read from stack
Memory	jxx Dest	No operation
Memory	call Dest $M_4[\text{valE}] \leftarrow \text{valP}$	Write return value on stack
Memory	ret $\text{valM} \leftarrow M_4[\text{valA}]$	Read return address

# Memory Address

---

```
int mem_addr = [
    icode in { IRMMOVL, IPUSHL,
                ICALL, IMRMOVL } : valE;
    icode in { IPOPL, IRET } : valA;
    # Other instructions don't need address
];
```

# Memory Read

	opl rA, rB	
Memory		No operation
Memory	rmmovl rA, D(rB)	
Memory	$M_4[valE] \leftarrow valA$	Write value to memory
	popl rA	
Memory	$valM \leftarrow M_4[valA]$	Read from stack
	jxx Dest	
Memory		No operation
	call Dest	
Memory	$M_4[valE] \leftarrow valP$	Write return value on stack
	ret	
Memory	$valM \leftarrow M_4[valA]$	Read return address

# Memory Read/Write

---

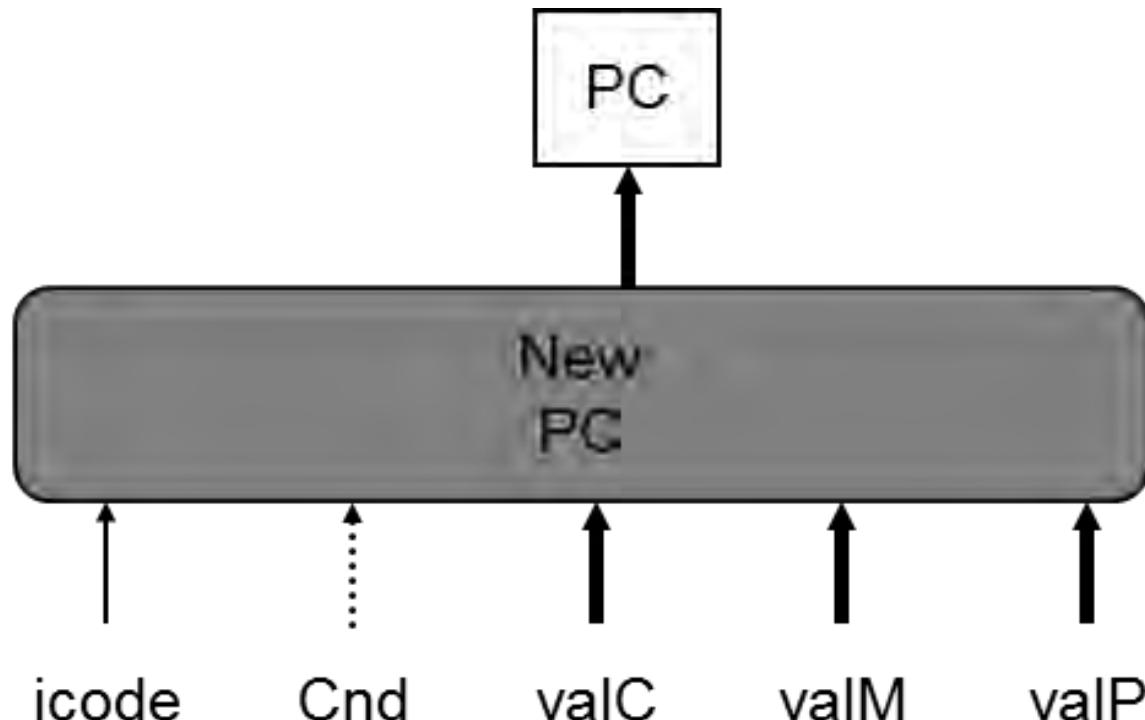
```
bool mem_read = icode in { IMRMOVL,  
IPOPL, IRET } ;
```

```
bool mem_write = icode in { IRMMOVL,  
IPUSHL, ICALL } ;
```

# PC Update Logic

---

- New PC
  - Select next value of PC



# PC Update

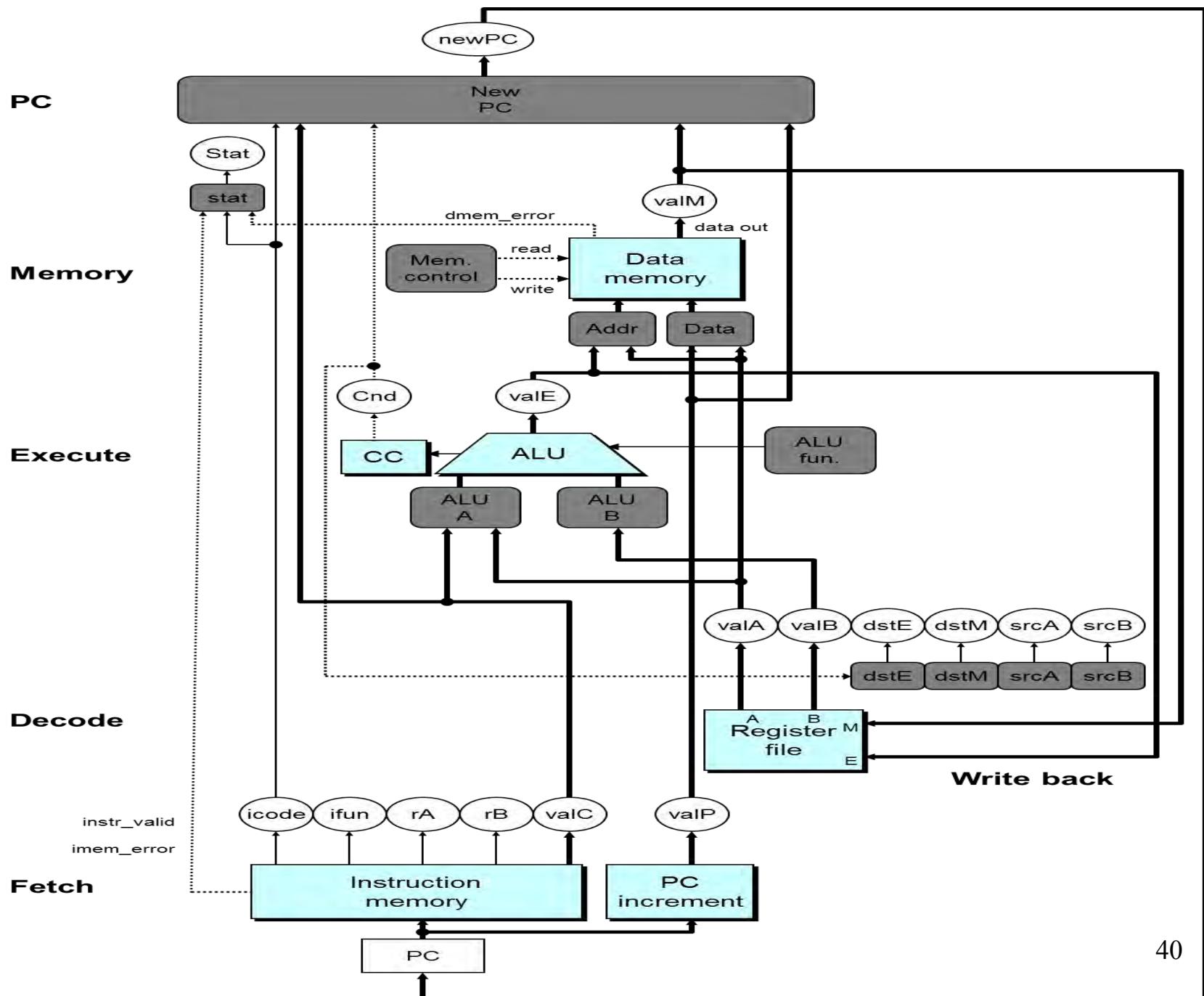
---

	OPl rA, rB	
PC update	PC $\leftarrow$ valP	Update PC
	rmmovl rA, D(rB)	
PC update	PC $\leftarrow$ valP	Update PC
	popl rA	
PC update	PC $\leftarrow$ valP	Update PC
	jXX Dest	
PC update	PC $\leftarrow$ Bch ? valC : valP	Update PC
	call Dest	
PC update	PC $\leftarrow$ valC	Set PC to destination
	ret	
PC update	PC $\leftarrow$ valM	Set PC to return address

# PC Update

---

```
int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Cnd : valC;
    icode == IRET : valM;
    1 : valP;
];
```



# SEQ Summary

---

- Implementation
  - Express every instruction as series of simple steps
  - Follow same general flow for each instruction type
  - Assemble registers, memories, predesigned combinational blocks
  - Connect with control logic

# SEQ Summary

---

- Limitations
  - Too slow to be practical
  - In one cycle, must propagate through instruction memory, register file, ALU, and data memory
  - Would need to run clock very slowly
  - Hardware units only active for fraction of clock cycle

# Principles of Pipeline

# Outline

---

- General Principles of Pipelining
  - Goal
  - Difficulties
- Suggested Reading 4.4

# Problem of SEQ

---

- Too slow
  - Too many tasks needed to finish in one clock cycle
  - Signals need long time to propagate through all of the stages
  - The clock must run slowly enough
- Does not make good use of hardware units
  - Every unit is active for part of the total clock cycle

# Real-World Pipelines: Car Washes

---

Sequential



Pipelined



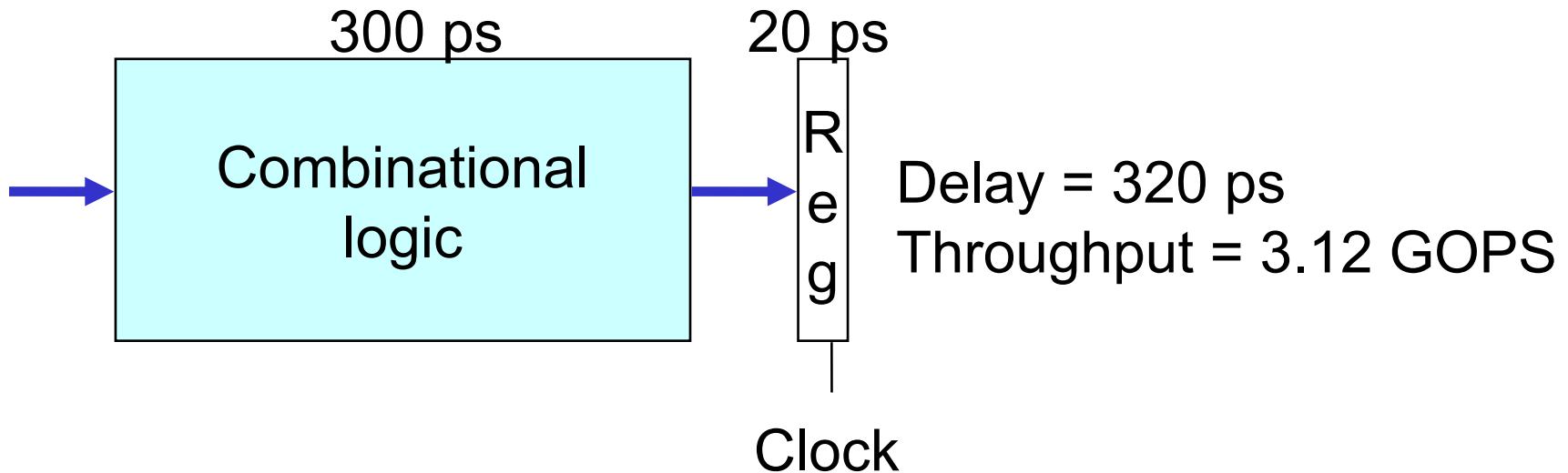
Parallel



- Idea

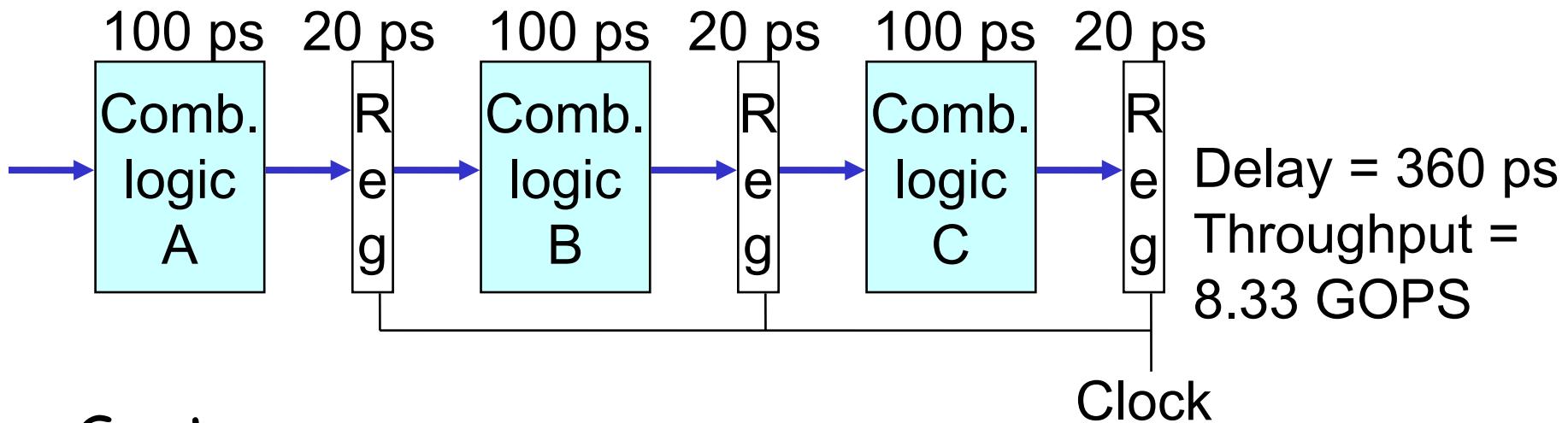
- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

# Computational Example



- System
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Clock must have clock cycle of at least 320 ps

# 3-Way Pipelined Version

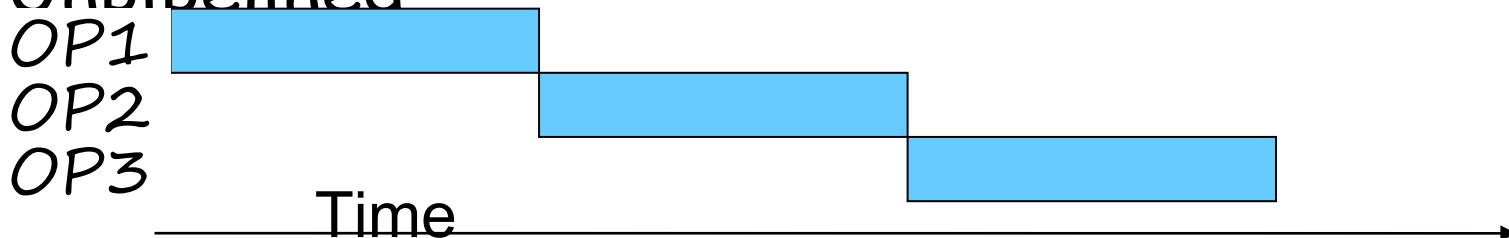


- System
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

# Pipeline Diagrams

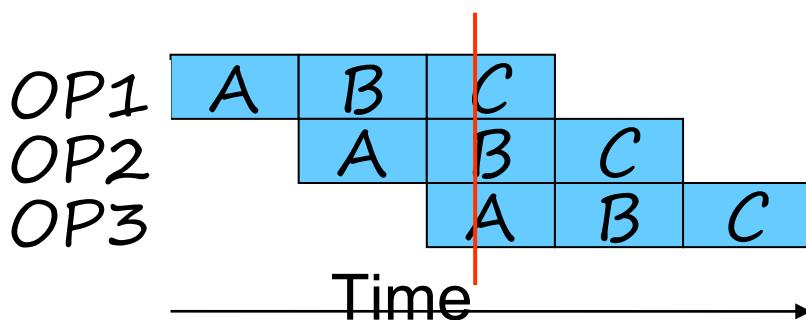
---

- Unpipelined



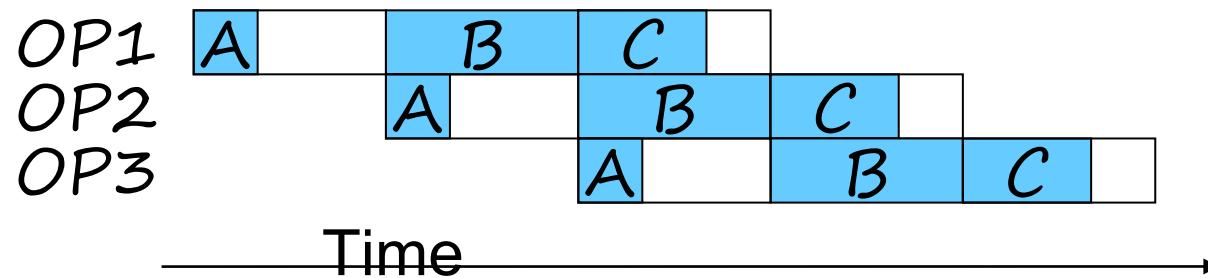
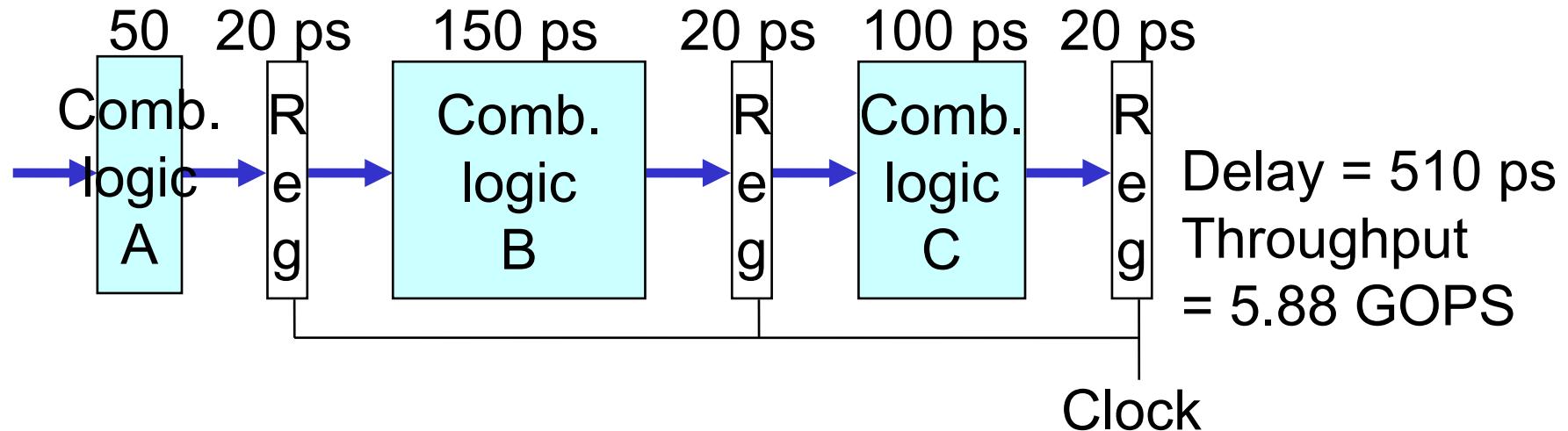
- Cannot start new operation until previous one completes

- 3-Way Pipelined



- Up to 3 operations in process simultaneously

# Limitations: Nonuniform Delays

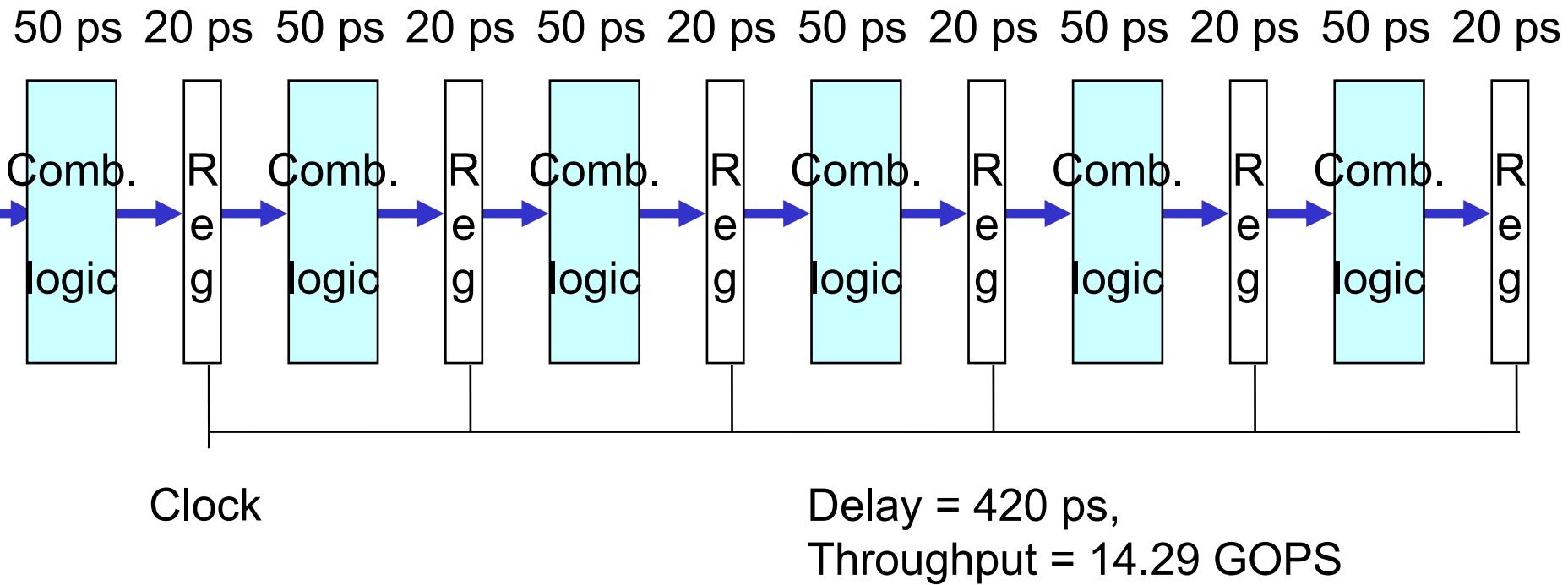


# Limitations: Nonuniform Delays

---

- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

# Limitations: Register Overhead

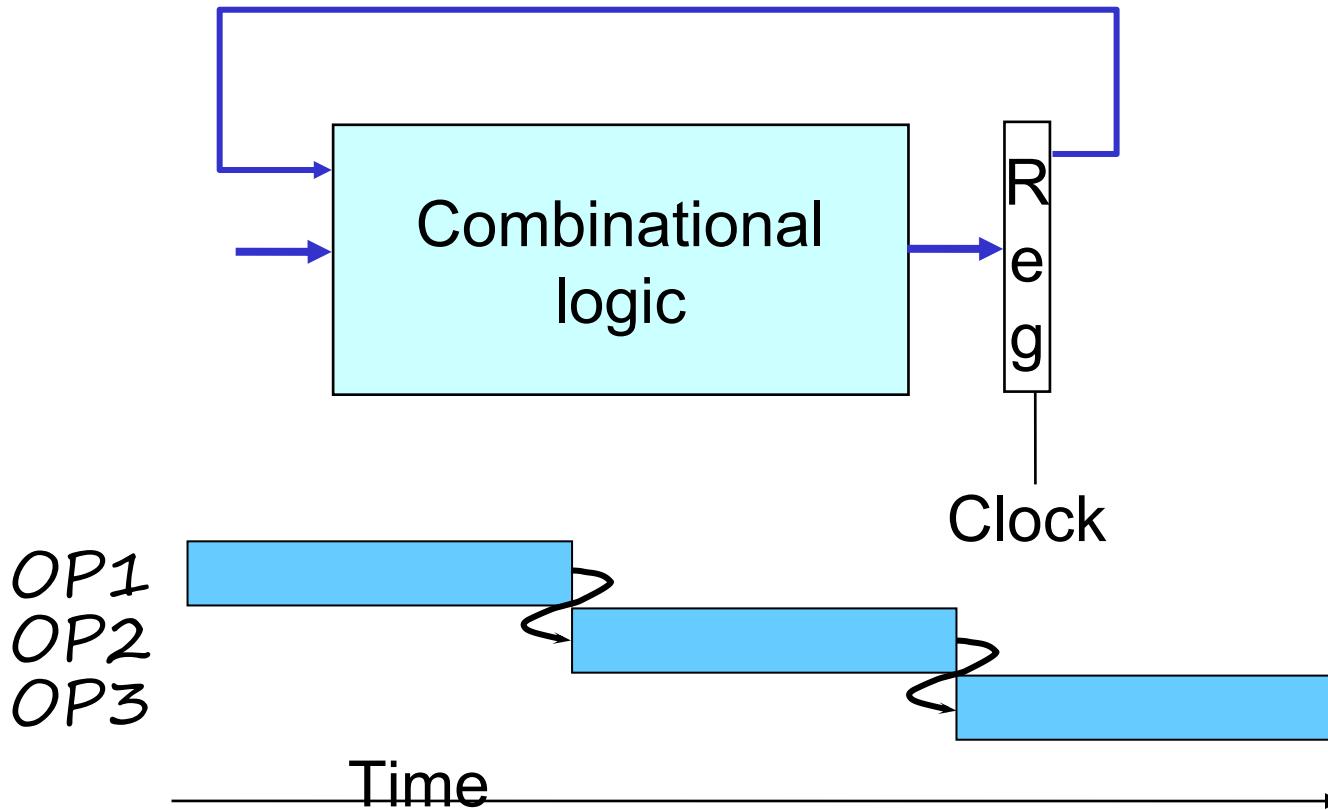


# Limitations: Register Overhead

---

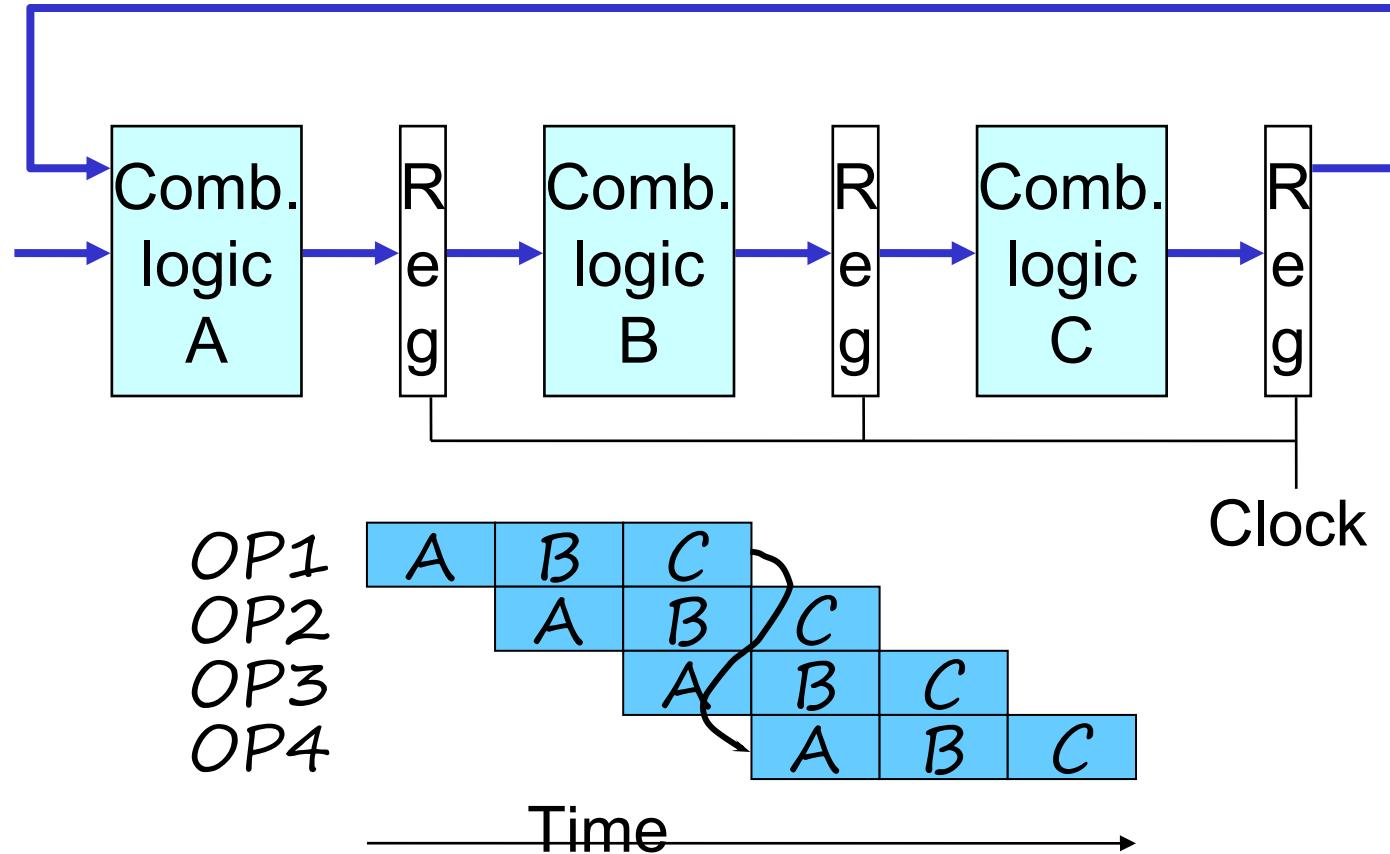
- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

# Data Dependencies



- System
  - Each operation depends on result from preceding one

# Data Hazards

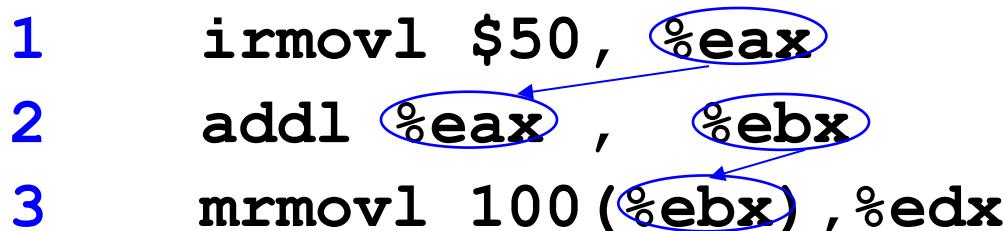


# Data Hazards

---

- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

```
1  irmovl $50, %eax
2  addl %eax, %ebx
3  mrmovl 100(%ebx), %edx
```



# SEQ+ CPU Implementation

# Outline

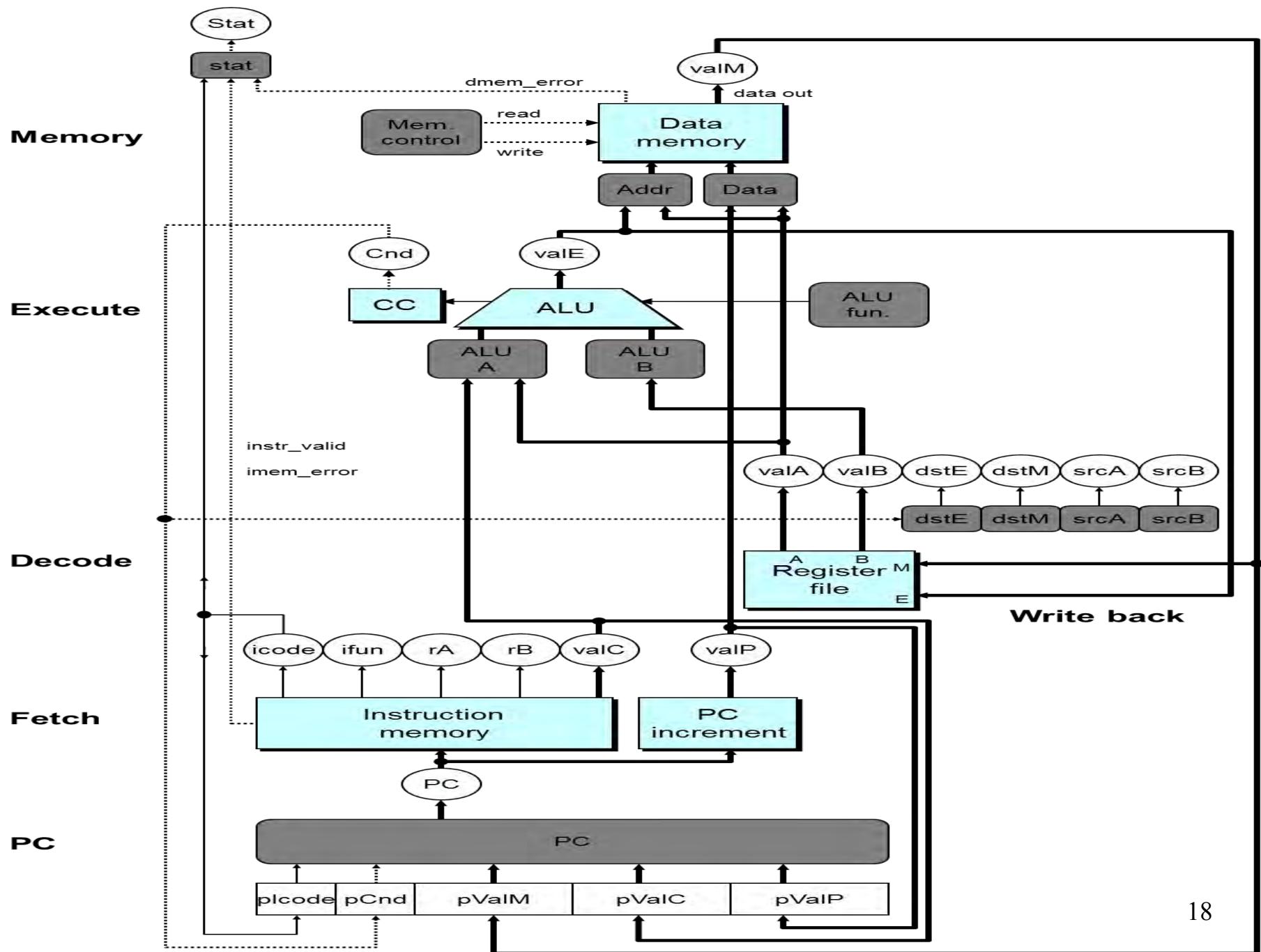
---

- SEQ+ Implementations
- Suggested Reading 4.5.1

# SEQ Hardware vs. SEQ+ Hardware

---

- SEQ Hardware
  - Stages occur in sequence
  - One operation in process at a time
- SEQ+ Hardware
  - Still sequential implementation
  - Reorder PC stage to put at beginning



# SEQ+ Hardware

---

- PC Stage
  - Task is to select PC for current instruction
  - Based on results computed by previous instruction
- Processor State
  - PC is no longer stored in register
  - But, can determine PC based on other stored information

# PC Computation

---

```
int pc= [
    picode == ICALL : pValC;
    picode == IJXX && pBch : pValC;
    Picode == IRET : pValM;
    1 : pValP;
];
```

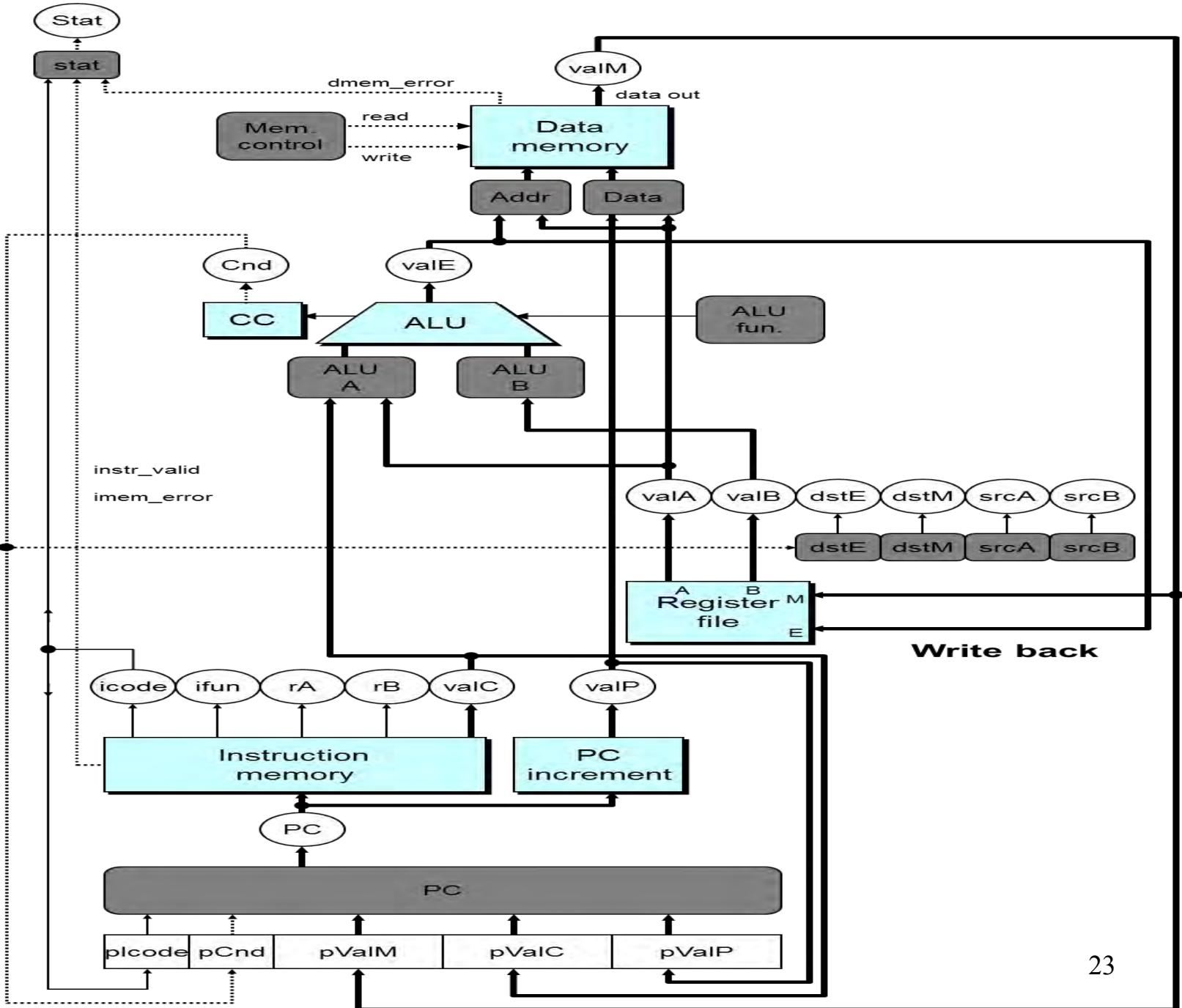
# Naïve Pipelined Implementation

# Outline

---

- Naïve PIPE Implementation
- Pipeline Hazards
- Suggested Reading 4.5.2 ~ 4.5.5

## Memory



# Pipeline Stages

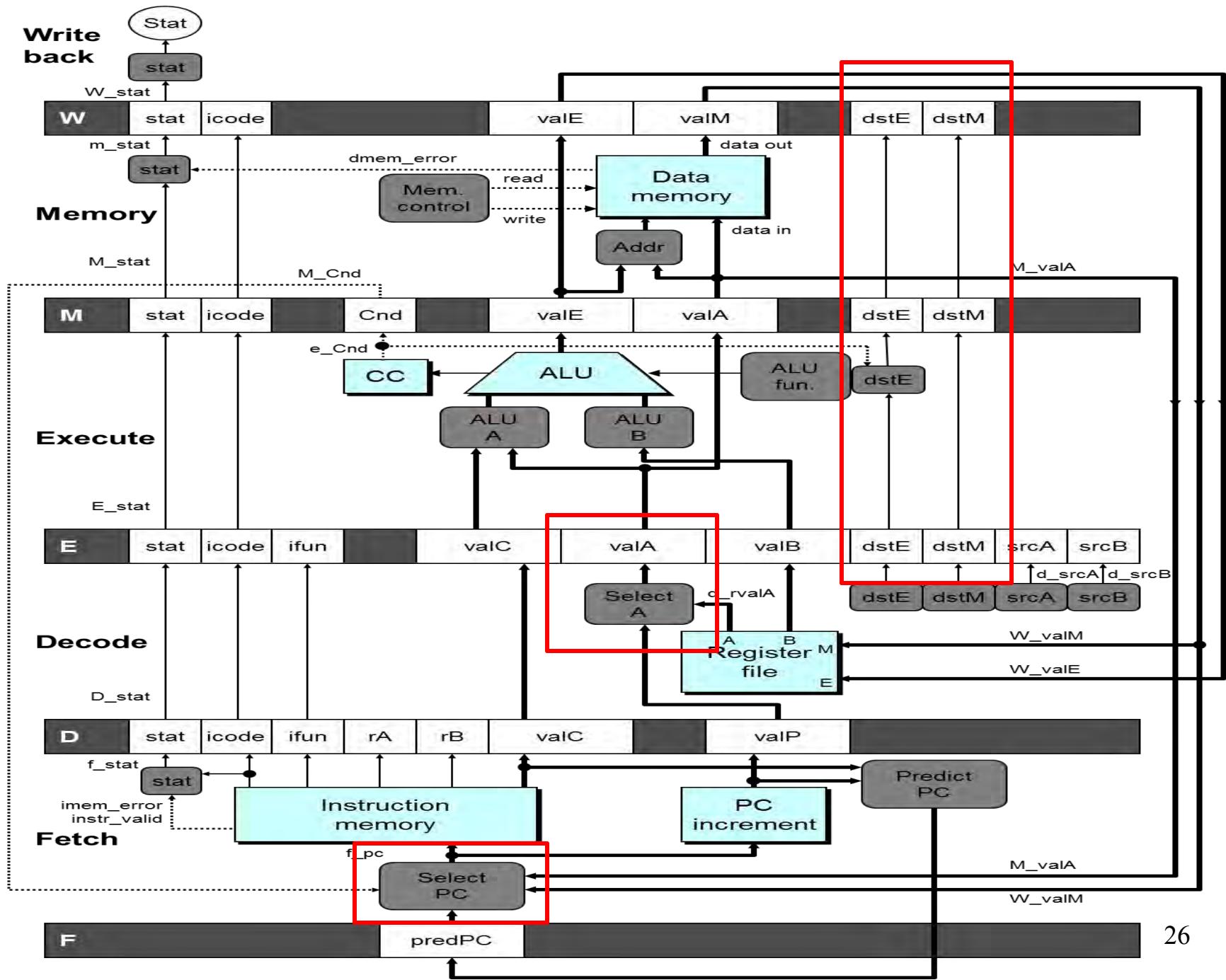
---

- Fetch
  - Select current PC
  - Read instruction
  - Compute incremented PC
- Decode
  - Read program registers
- Execute
  - Operate ALU
- Memory
  - Read or write data memory
- Write Back
  - Update register file

# PIPE- Hardware

---

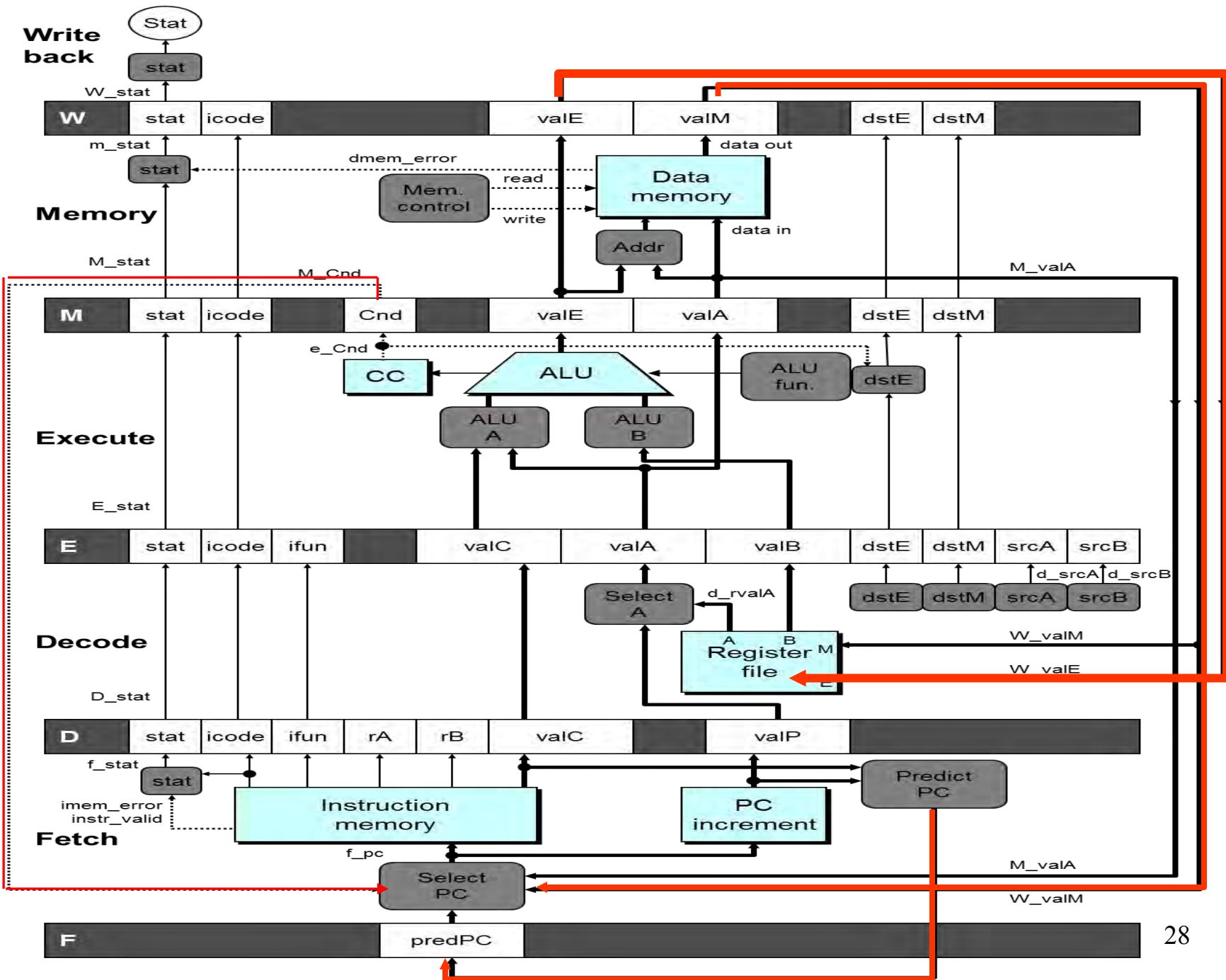
- Pipeline registers hold intermediate values from instruction execution
- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode



# Feedback Paths

---

- Predicted PC
  - Guess value of next PC
  - Branch information
    - Jump taken/not-taken
    - Fall-through or target address
  - Return point
    - Read from memory
- Register updates
  - To register file write ports

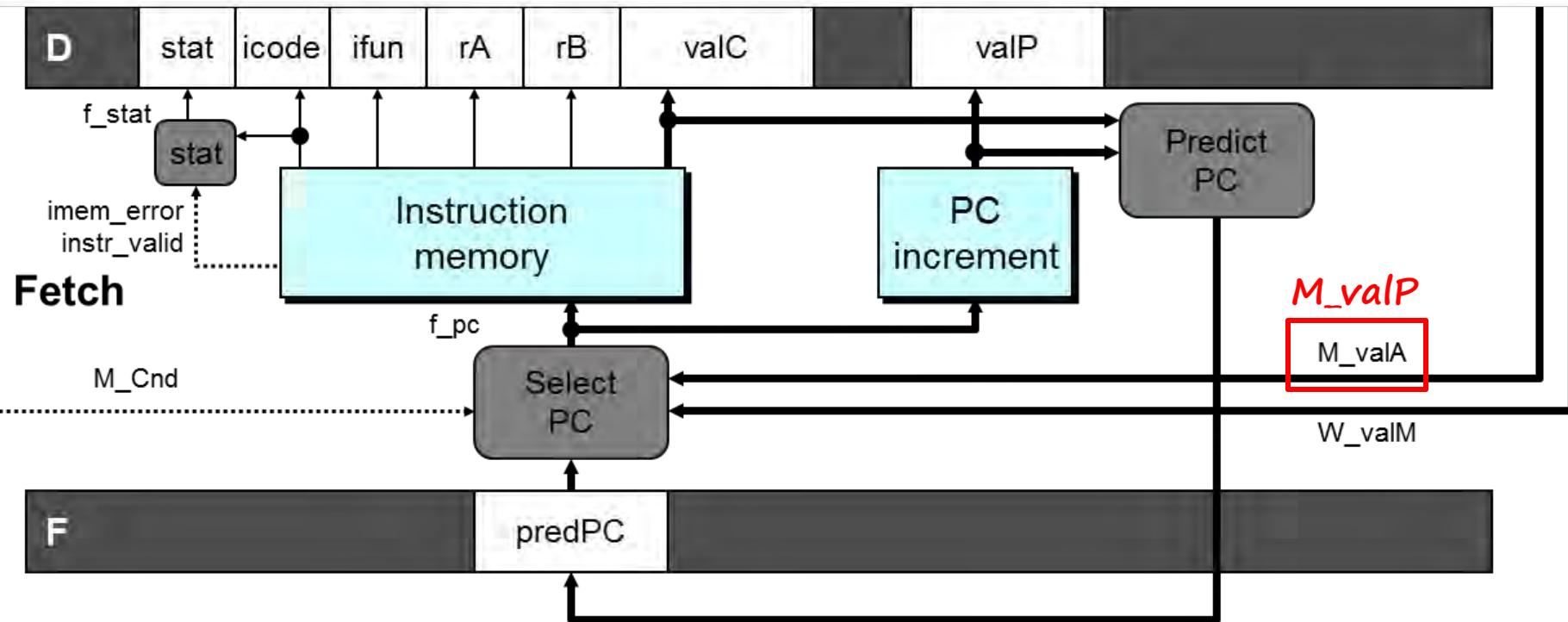


# Predicting the PC

---

- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

# Predicting the PC



# Our Prediction Strategy

---

- Instructions that Don't Transfer Control
  - Predict next PC to be valP
  - Always reliable
- Call and Unconditional Jumps
  - Predict next PC to be valC (destination)
  - Always reliable

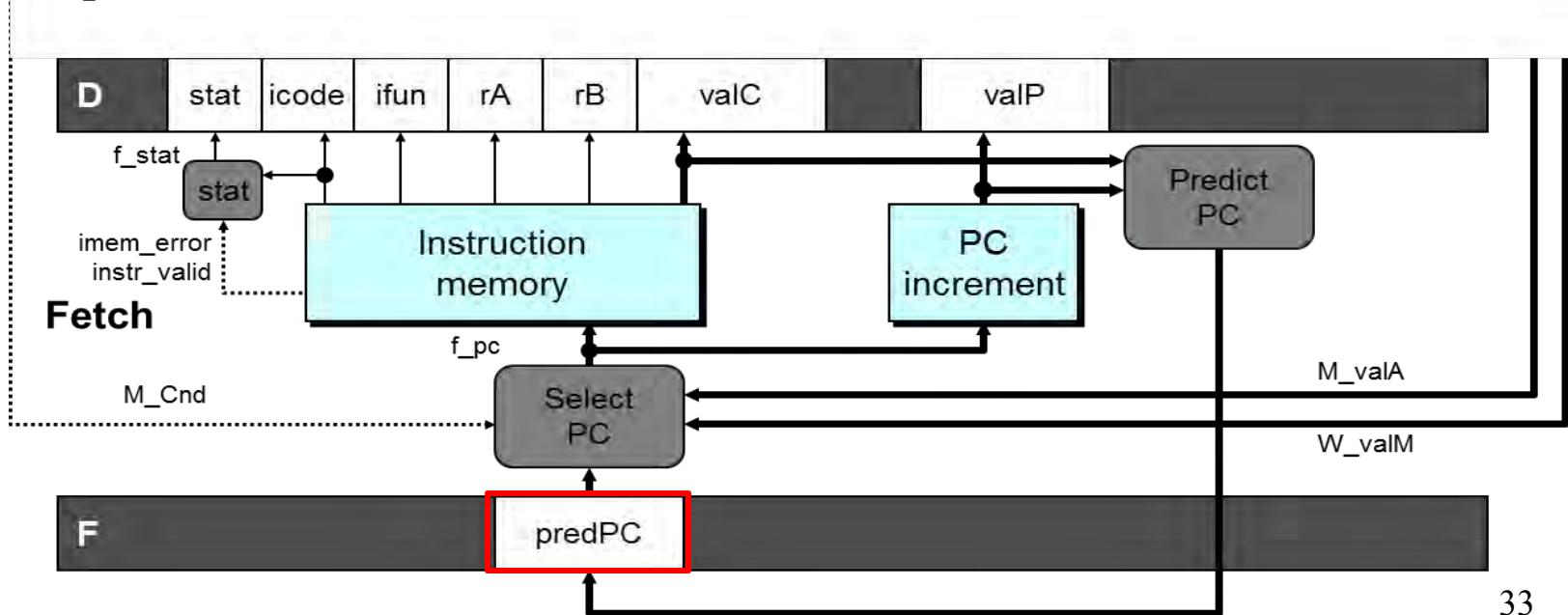
# Our Prediction Strategy

---

- Conditional Jumps
  - Predict next PC to be valC (destination)
  - Only correct if branch is taken
    - Typically right 60% of time
    - Recovery: M\_Cnd and M\_valA (valP: next PC)
- Return Instruction
  - Don't try to predict

# Select PC

```
Int F_predPC = [  
    f_icode in {IJXX, ICALL} : f_valC;  
    1: f_valP;  
];
```



# Recovering from PC Misprediction

---

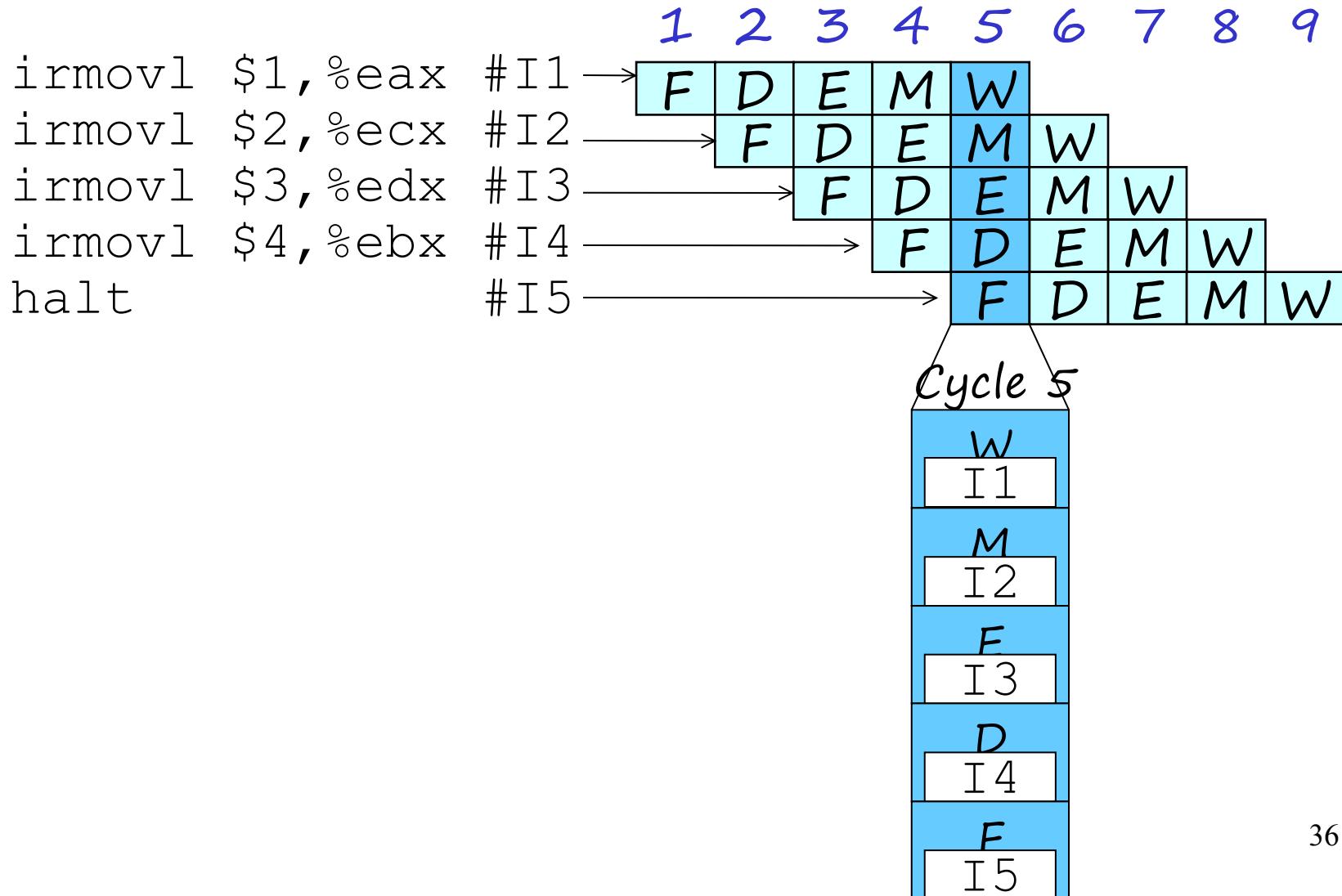
- Mispredicted Jump
  - Will see branch flag once instruction reaches Memory stage
  - Can get fall-through PC from valA (valP)
- Return Instruction
  - Will get return PC when ret reaches write-back stage

# Select PC

---

```
int f_PC = [  
#mispredicted branch. Fetch at incremented PC  
    M_icode == IJXX && !M_Cnd : M_valA;  
#completion of RET instruciton  
    W_icode == IRET : W_valM;  
#default: Use predicted value of PC  
    1: F_predPC  
] ;
```

# Pipeline Demonstration



# Avoiding Data Hazard

# Outline

---

- Instruction Stalling
- Data Forwarding
- Suggested Reading 4.5.6, 4.5.7

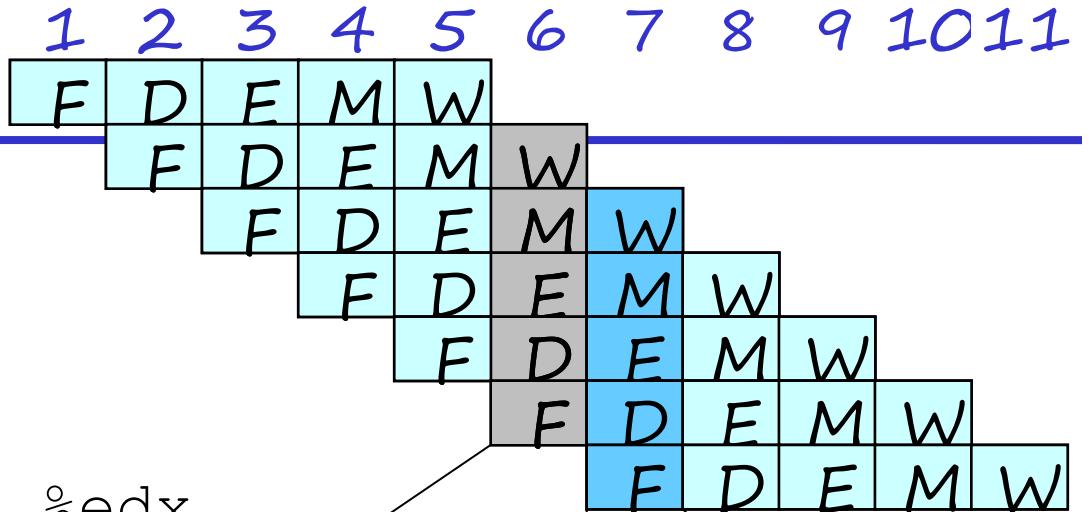
# Data Dependencies in Processors

---

```
1    irmovl $50, %eax
2    addl %eax, %ebx
3    mrmovl 100(%ebx), %edx
```

- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

# Data Dependencies: 3 Nop's



# demo-h3.ys

0x000:irmovl \$10,%edx

0x006:irmovl \$3,%eax

0x00c:nop

0x00d:nop

0x00e:nop

0x00f:addl %edx,%eax

0x011:halt

Cycle 6

Write Back

R[%eax] ← 3

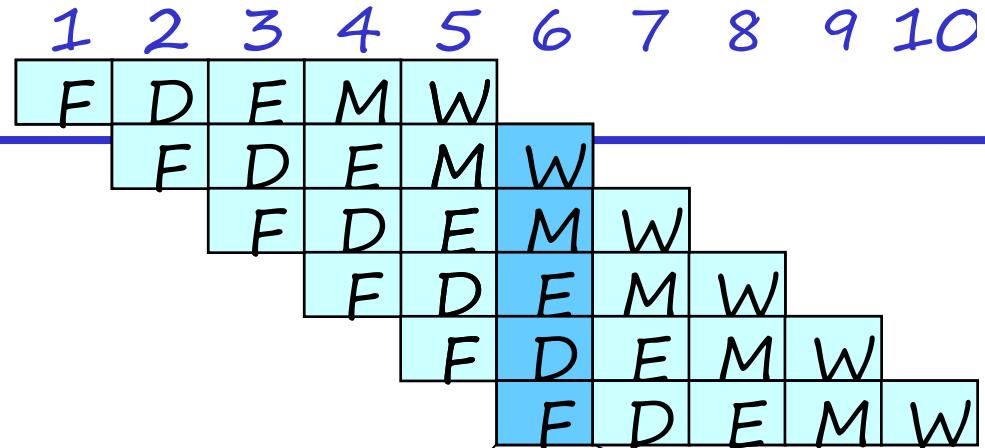
Cycle 7

Decode

val ← R[%edx] = 10

val ← R[%eax] = 3

# Data Dependencies: 2 Nop's



```
# demo-h2.ys
```

```
0x000:irmovl $10,%edx
```

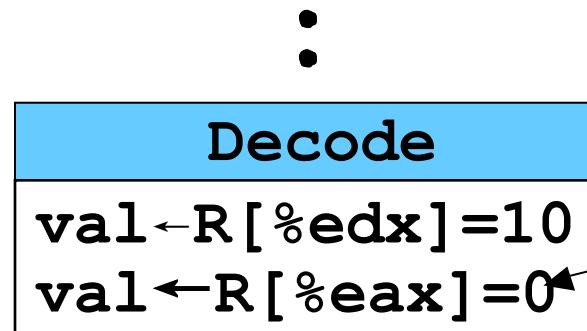
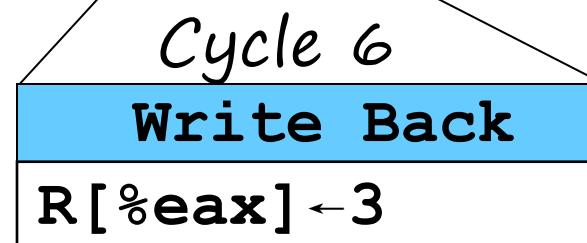
```
0x006:irmovl $3,%eax
```

```
0x00c:nop
```

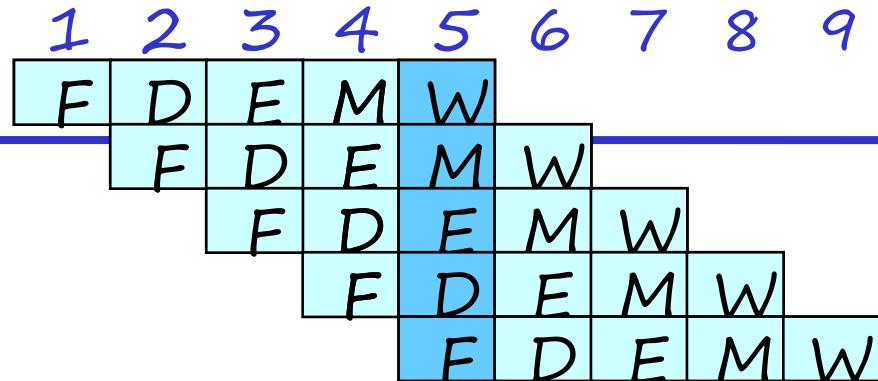
```
0x00d:nop
```

```
0x00e:addl %edx,%eax
```

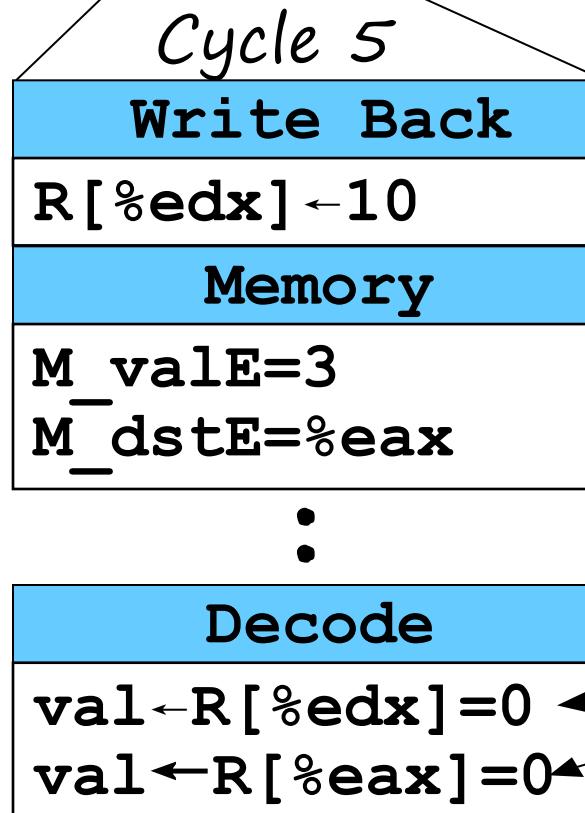
```
0x010:halt
```



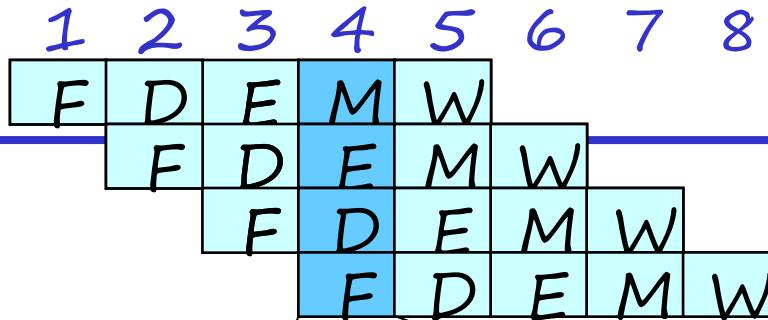
# Data Dependencies: 1 Nop



```
# demo-h1.ys
0x000:irmovl $10,%edx
0x006:irmovl $3,%eax
0x00c:nop
0x00d:addl %edx,%eax
0x00f:halt
```

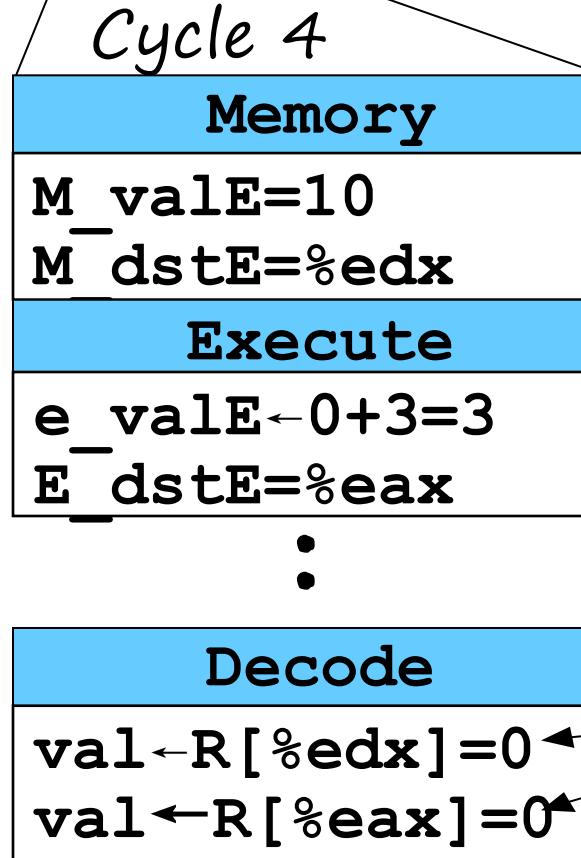


# Data Dependencies: No Nop



```
# demo-h0.ys
```

```
0x000:irmovl $10,%edx  
0x006:irmovl $3,%eax  
0x00c:addl %edx,%eax  
0x00e:halt
```



# Classes of Data Hazards

---

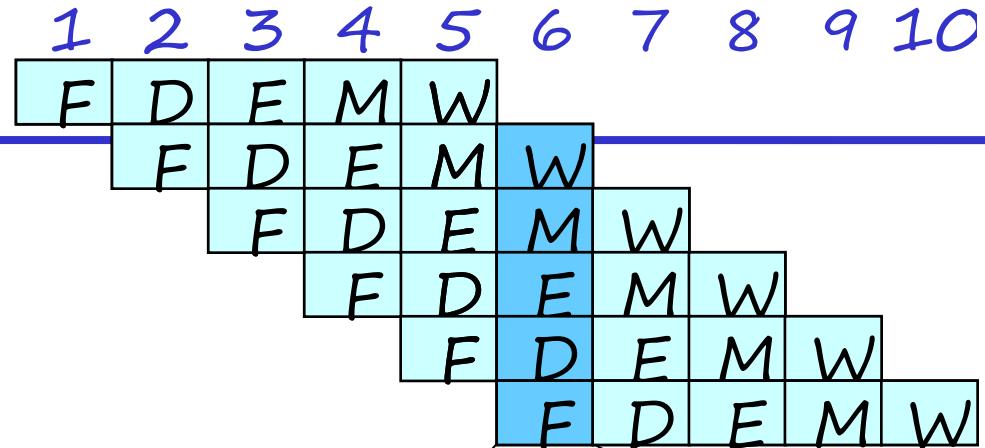
Hazards can potentially occur when one instruction updates part of the program state that is read by a later instruction

# Classes of Data Hazards

---

- Program states:
  - Program registers (identified)
  - Condition codes (**No hazards**)
    - Both written and read in the execute stage.
  - Program counter (section 4.5.11)
    - Conflicts between updating and reading PC cause control hazards (e.g. mispredicted branches and `ret`)
  - Memory
    - Both written and read in the memory stage.
    - Without self-modified code, **no hazards**.
  - Status register (section 4.5.9)
    - Orderly halt when an exception occurs

# Data Dependencies: 2 Nop's



```
# demo-h2.ys
```

```
0x000:irmovl $10,%edx
```

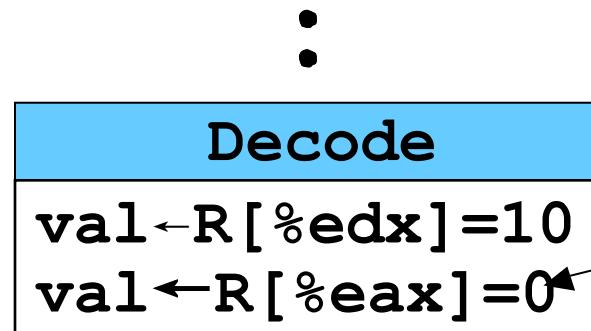
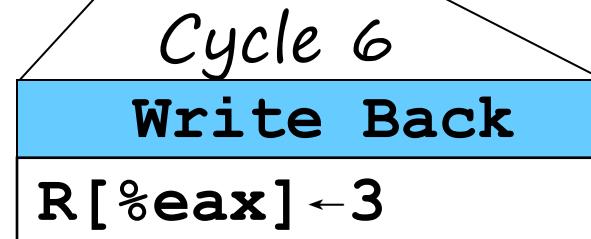
```
0x006:irmovl $3,%eax
```

```
0x00c:nop
```

```
0x00d:nop
```

```
0x00e:addl %edx,%eax
```

```
0x010:halt
```



# Data Dependencies: 2 Nop's

```
# demo-h2.ys
```

```
0x000:irmovl $10,%edx
```

```
0x006:irmovl $3,%eax
```

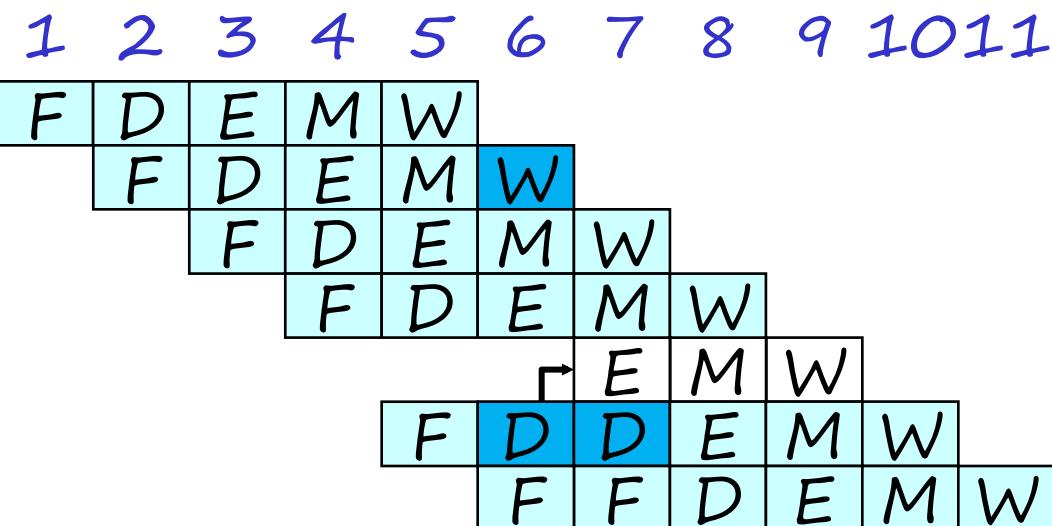
```
0x00c:nop
```

```
0x00d:nop
```

*bubble*

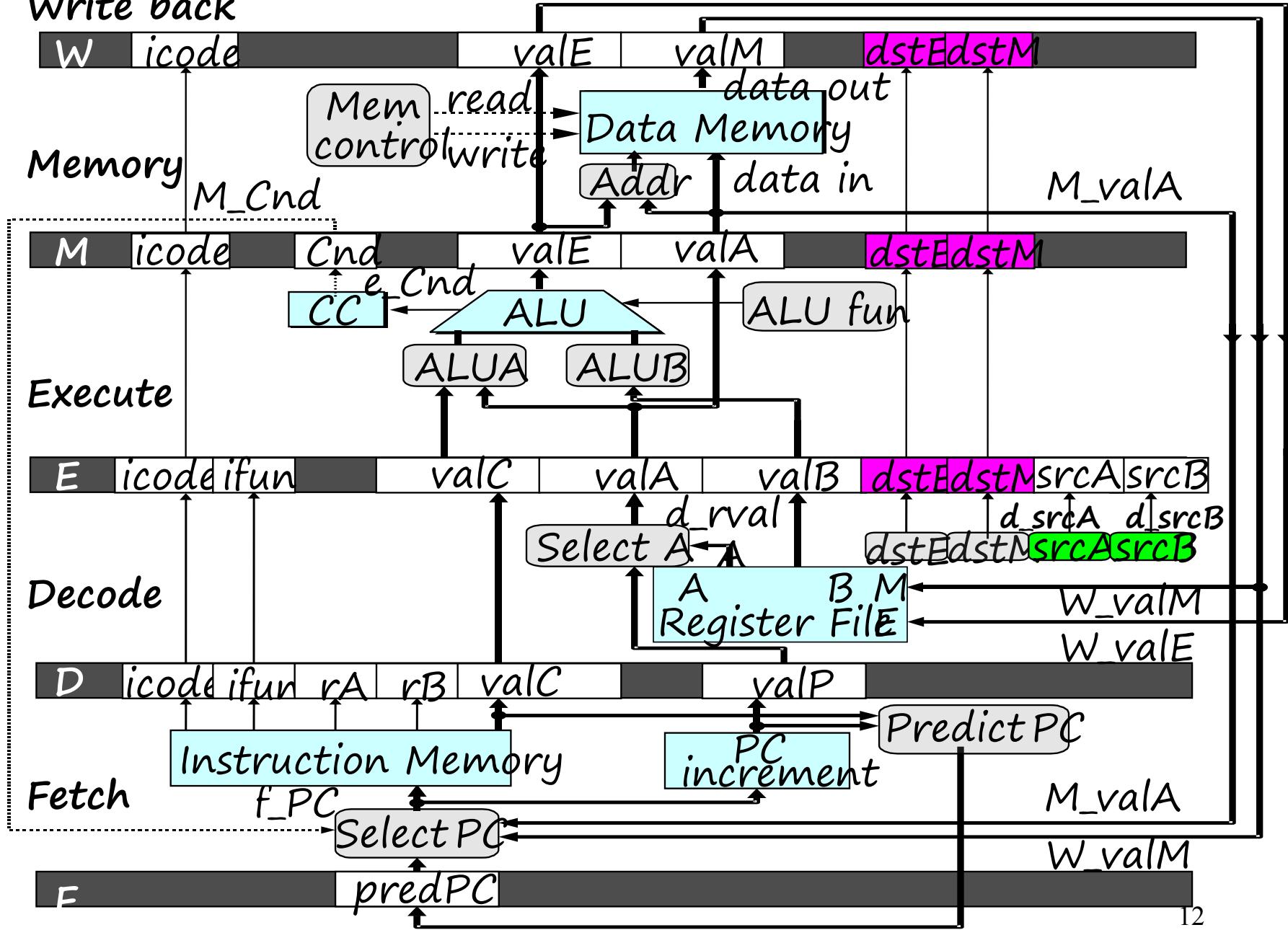
```
0x00e:addl %edx,%eax
```

```
0x010:halt
```



- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject **nop** into execute stage

# Write back



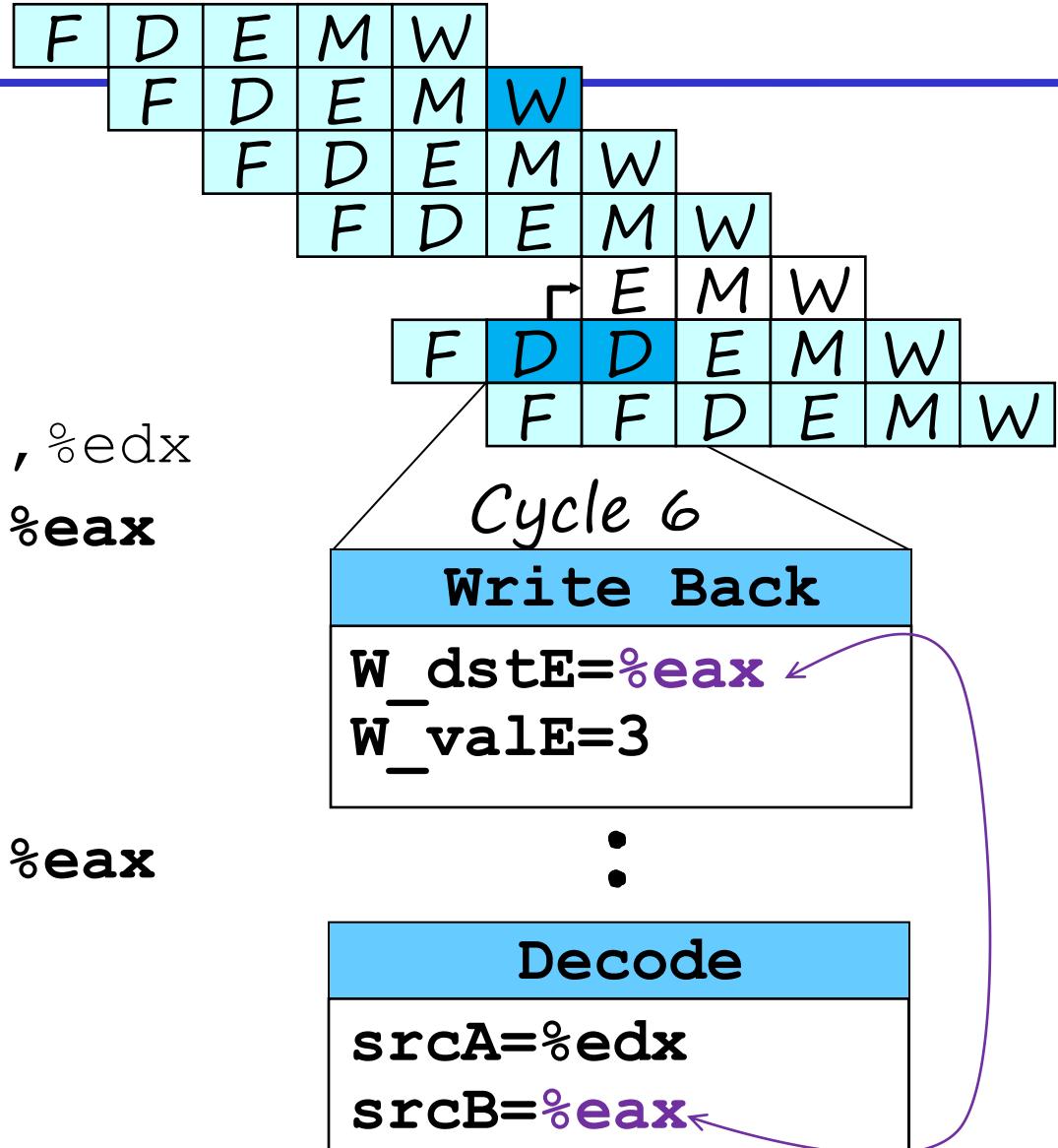
# Stall Condition

---

- Source Registers
  - srcA and srcB of current instruction in decode stage
- Destination Registers
  - dstE and dstM fields
  - Instructions in execute, memory, and write-back stages
- Condition
  - $\text{srcA} == \text{dstE}$  or  $\text{srcA} == \text{dstM}$
  - $\text{srcB} == \text{dstE}$  or  $\text{srcB} == \text{dstM}$
- Special Case
  - Don't stall for register ID F
    - Indicates absence of register operand

# Data Dependencies: 2 Nop's

1 2 3 4 5 6 7 8 9 10 11



# demo-h2.ys

0x000:irmovl \$10,%edx

0x006:irmovl \$3,%eax

0x00c:nop

0x00d:nop

**bubble**

0x00e:addl %edx,%eax

0x010:halt

# Stalling X3

# demo-h0.ys

0x000:irmovl \$10,%edx

0x006:irmovl \$3,%eax

bubble

bubble

bubble

0x00c:addl %edx,%eax

0x0e:halt

Cycle 4

Execute

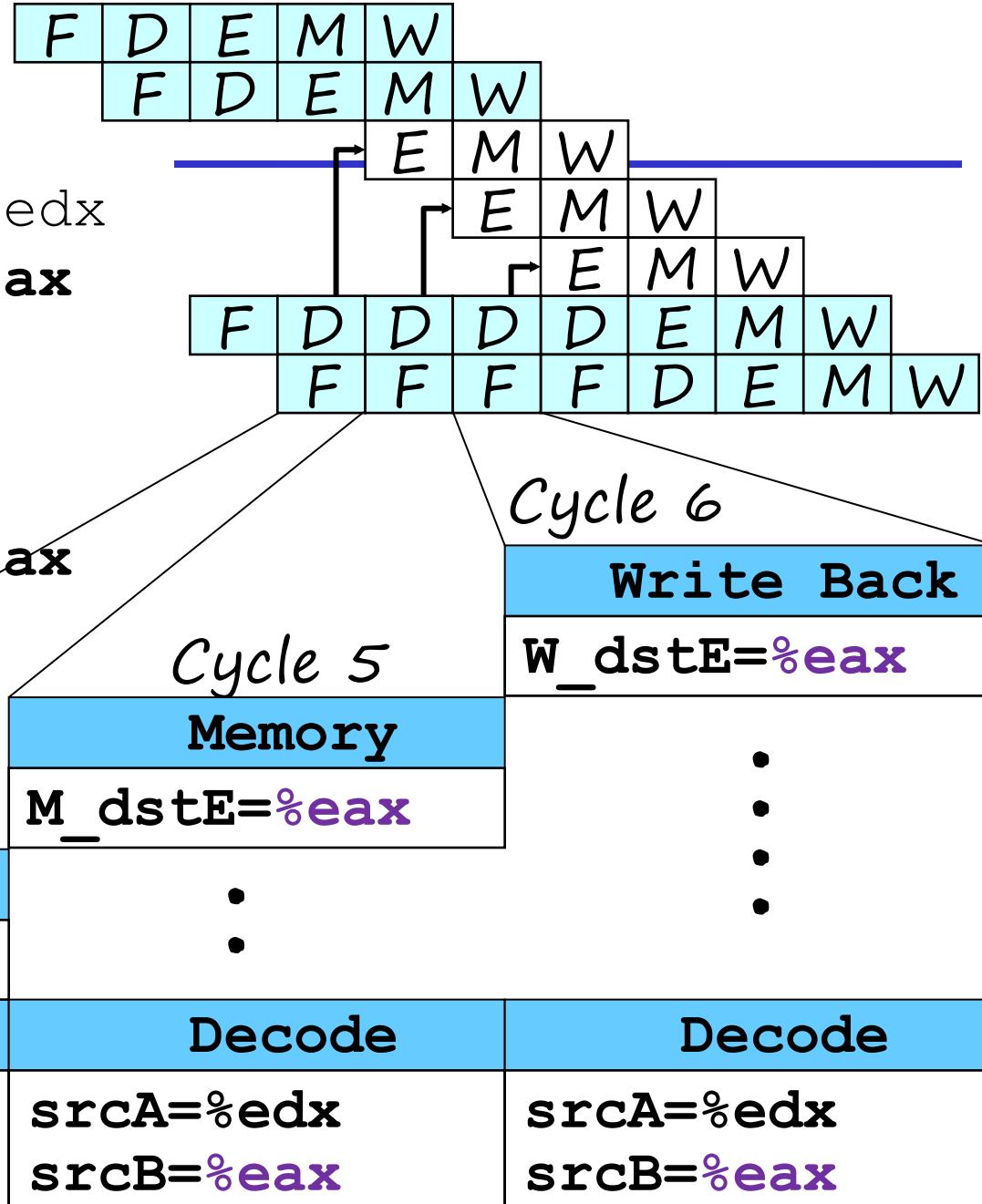
E\_dstE=%eax

Decode

srcA=%edx

srcB=%eax

1 2 3 4 5 6 7 8 9 10 11



# What Happens When Stalling?

```
# demo-h0.ys
```

```
0x000: irmovl $10,%edx  
0x006: irmovl $3,%eax  
0x00c: addl %edx,%eax  
0x00e: halt
```

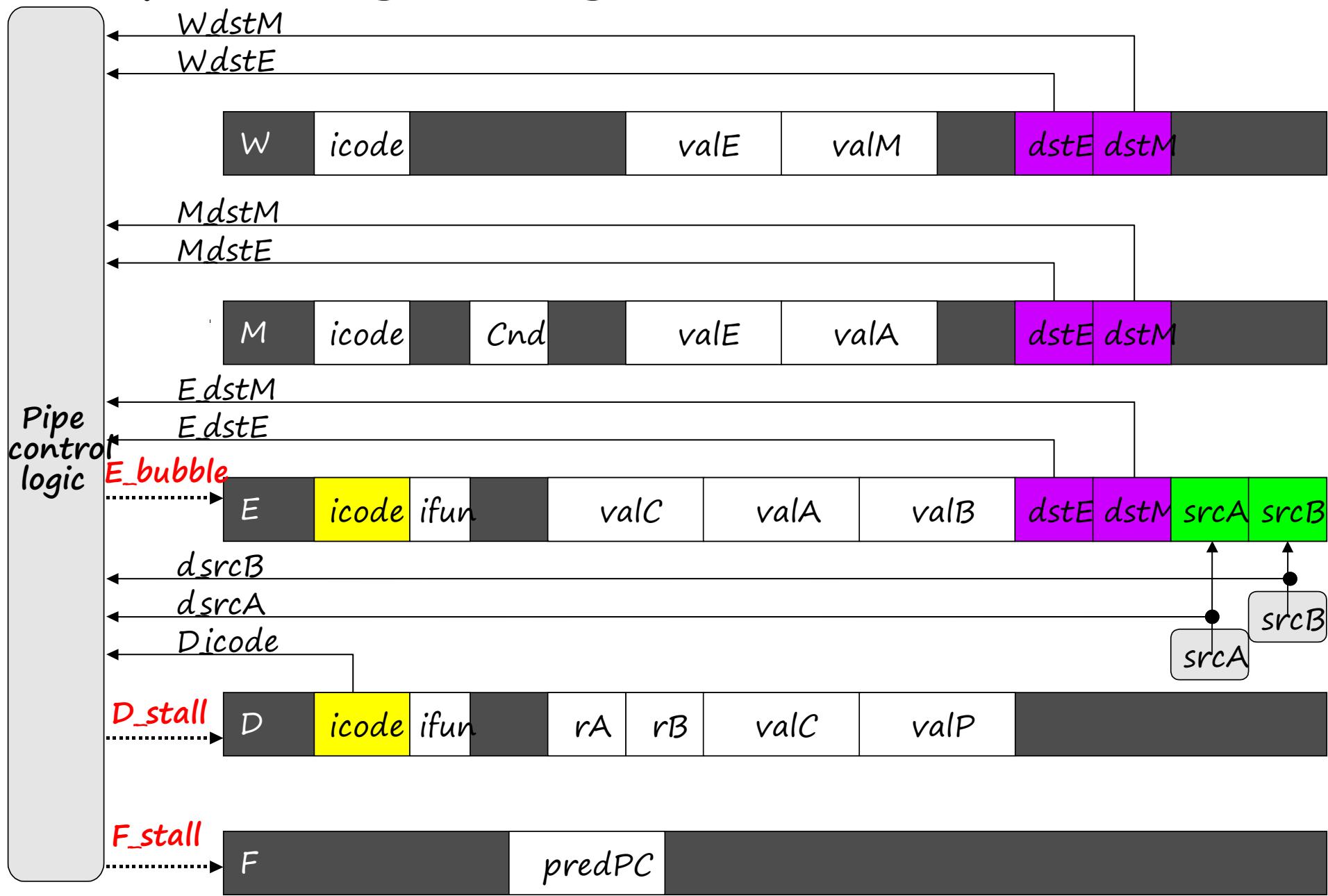
- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage

Cycle 8

Write Back	bubble
Memory	bubble
Execute	0x00c: addl %edx,%eax
Decode	0x00e: halt
Fetch	

- Like dynamically generated nop's
- Move through later stages

# Implementing Stalling



# Implementing Stalling

---

- Pipeline Control
  - Combinational logic detects stall condition
  - Sets mode signals for how pipeline registers should update

# Initial Version of Pipeline Control

---

```
bool F_stall =
    d_srcA == E_dstE || d_srcA == E_dstM ||
    d_srcA == M_dstE || d_srcA == M_dstM ||
    d_srcA == W_dstE || d_srcA == W_dstM;

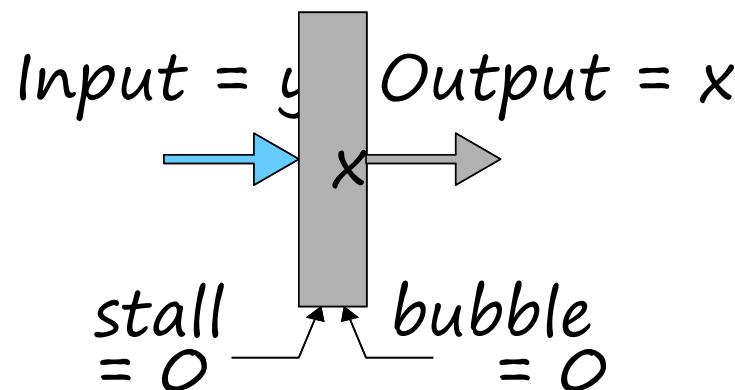
bool D_stall =
    d_srcA == E_dstE || d_srcA == E_dstM ||
    d_srcA == M_dstE || d_srcA == M_dstM ||
    d_srcA == W_dstE || d_srcA == W_dstM;

bool E_bubble =
    d_srcA == E_dstE || d_srcA == E_dstM ||
    d_srcA == M_dstE || d_srcA == M_dstM ||
    d_srcA == W_dstE || d_srcA == W_dstM;
```

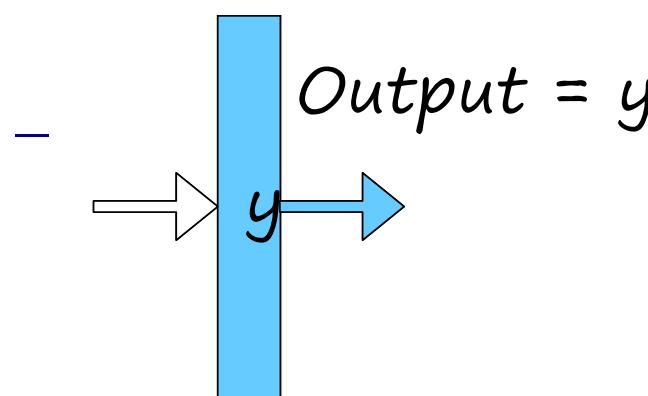
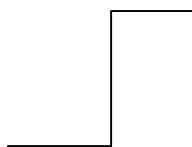
# Pipeline Register Modes

---

Normal

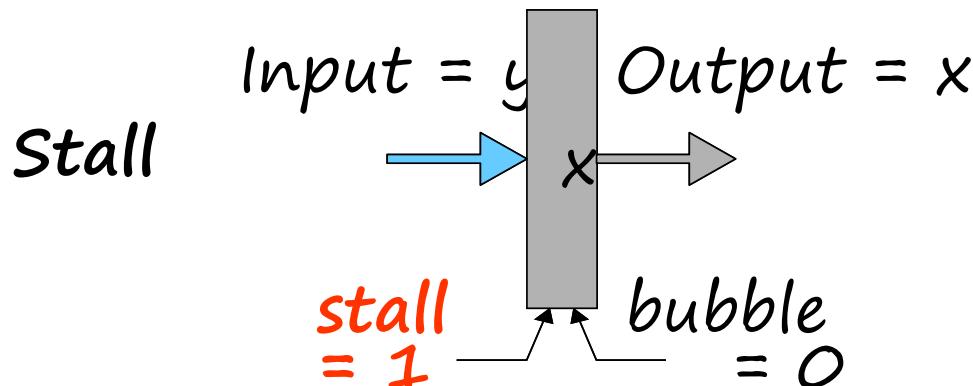


Rising  
clock

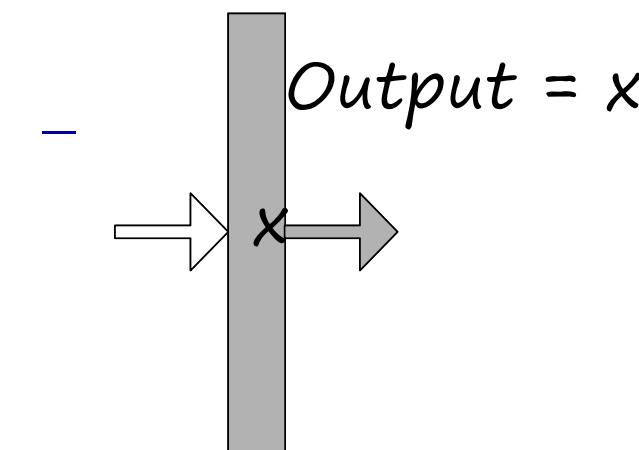
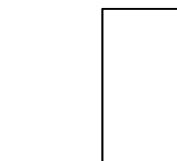


# Pipeline Register Modes

---

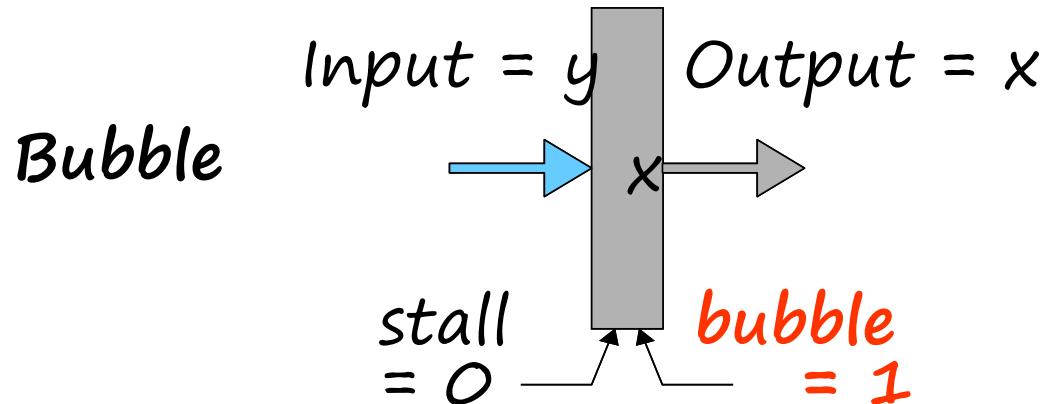


Rising  
clock

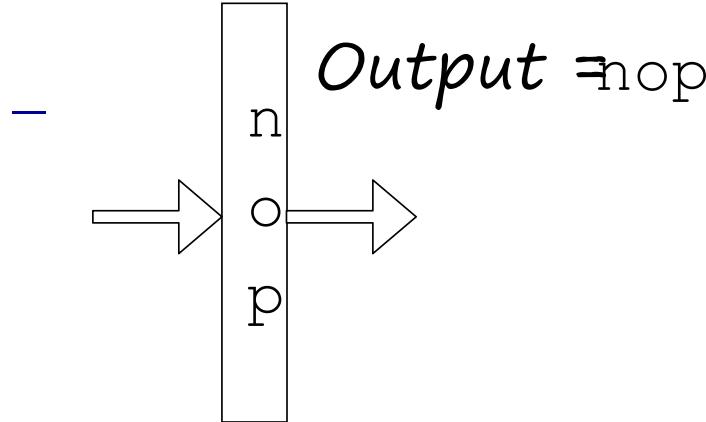
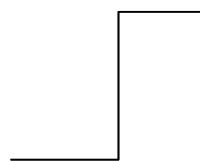


# Pipeline Register Modes

---



Rising  
clock



# Data Forwarding

---

- Observation
  - Value generated in execute or memory stage
- Trick
  - Pass value directly from generating instruction to decode stage
  - Needs to be available at end of decode stage

# Data Dependencies: 2 Nop's

# demo-h2.ys

0x000:irmovl \$10,%edx

0x006:irmovl \$3,%eax

0x00c:nop

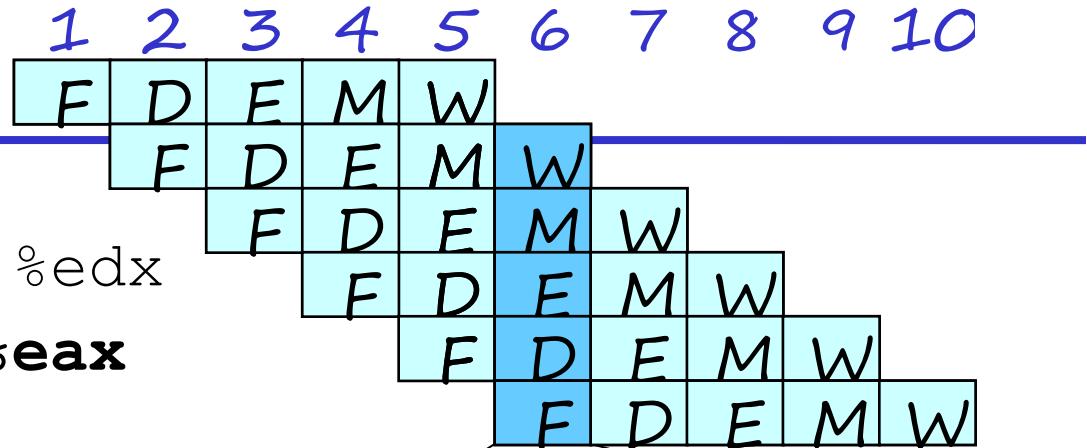
0x00d:nop

**bubble**

0x00e:addl %edx,%eax

0x010:halt

- irmovl in W stage
- Destination value in W pipeline register
- Forward as valB for D stage



Cycle 6

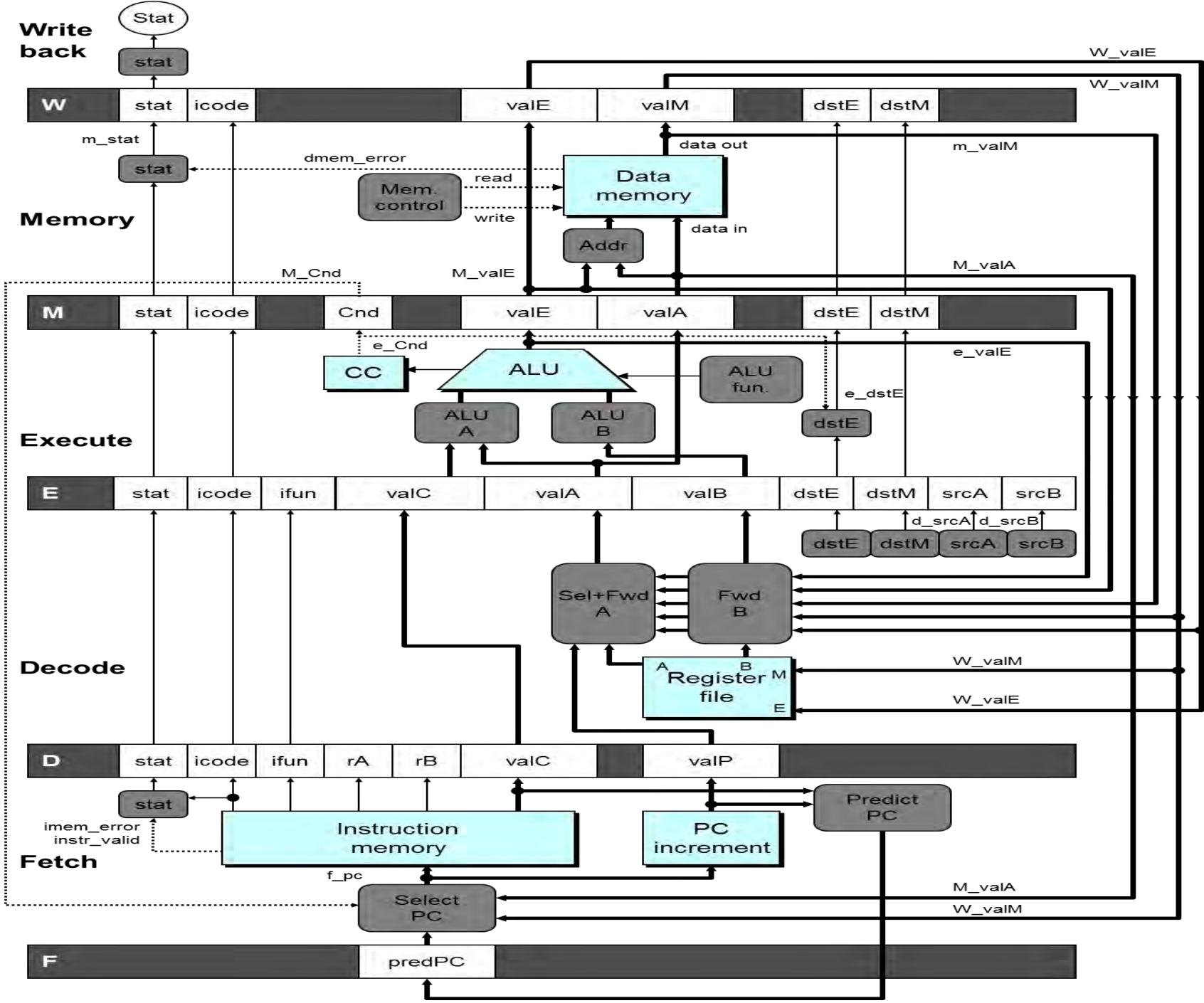
Write Back

$W_{dstE} = \%eax$	$R[\%eax] \leftarrow 3$
$W_{valE} = 3$	

:

Decode

$srcA = \%edx$	$val \leftarrow R[\%edx] = 10$
$srcB = \%eax$	$val \leftarrow W_{valE} = 3$



# Bypass Paths

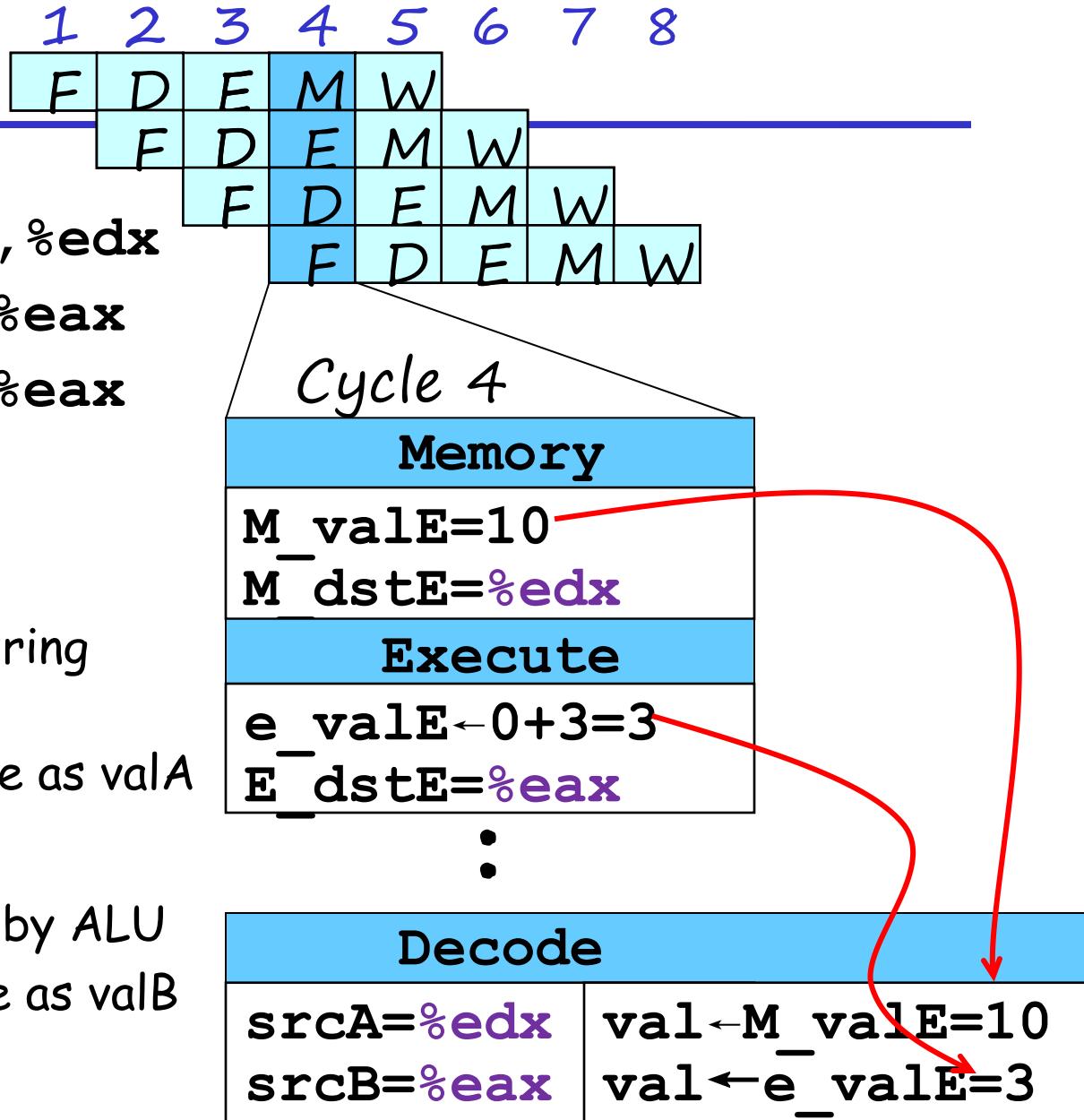
---

- Decode Stage
  - Forwarding logic selects valA and valB
  - Normally from register file
  - Forwarding: get valA or valB from later pipeline stage
- Forwarding Sources
  - Execute: valE
  - Memory: valE, valM
  - Write back: valE, valM

# Data Dependencies: No Nop

```
# demo-h0.ys
0x000:irmovl $10,%edx
0x006:irmovl $3,%eax
0x00c:addl %edx,%eax
0x00e:halt
```

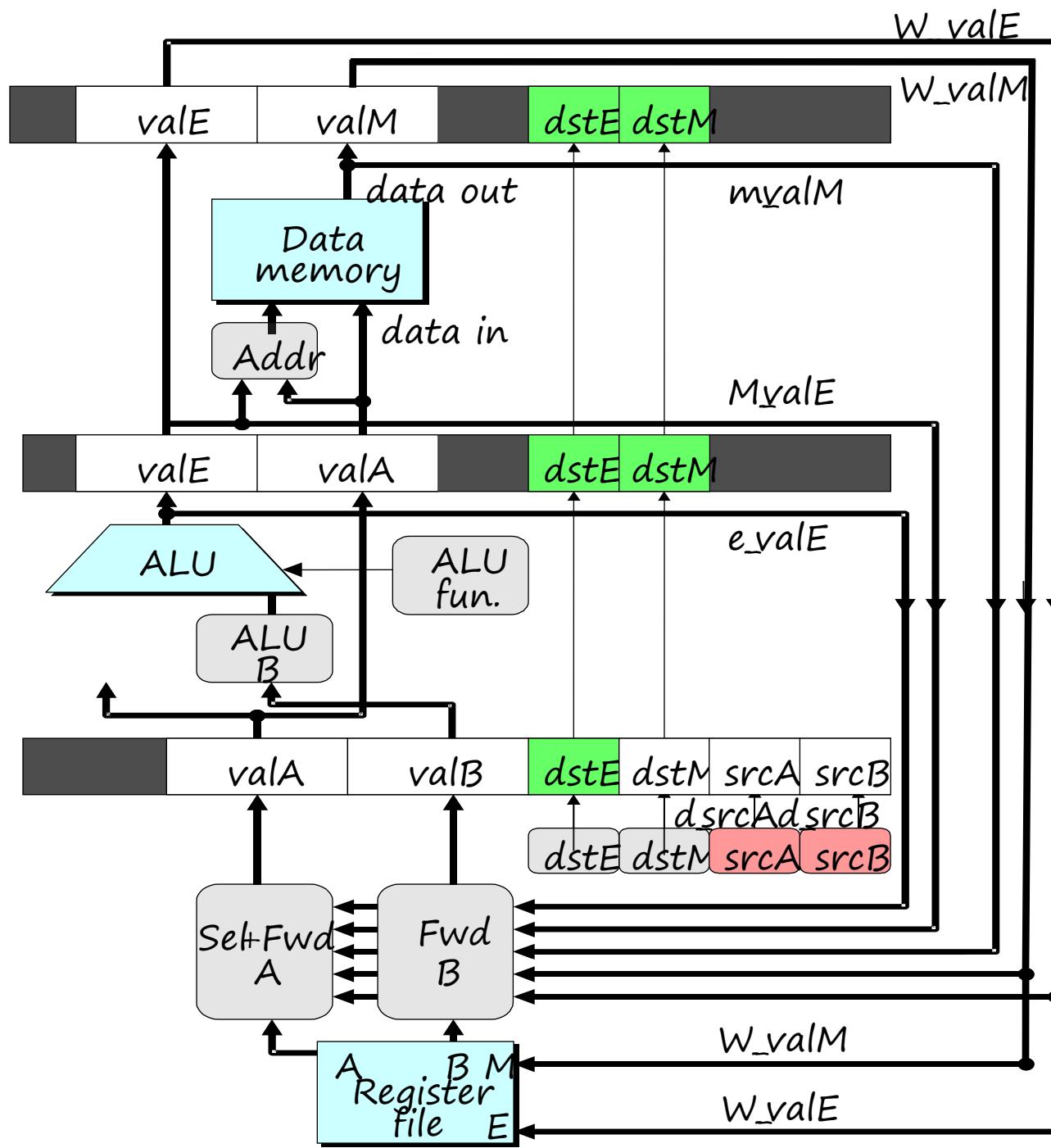
- Register **%edx**
  - Generated by ALU during previous cycle
  - Forward from M stage as valA
- Register **%eax**
  - Value just generated by ALU
  - Forward from E stage as valB



# Implementing Forwarding

---

- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage



# Implementing Forwarding

---

```
## What should be the A value?  
int new_E_valA = [  
    # Use incremented PC  
    D_icode in { ICALL, IJXX } : D_valP;  
    # Forward valE from execute  
    d_srcA == E_dstE : e_valE;  
    # Forward valM from memory  
    d_srcA == M_dstM : m_valM;  
    # Forward valE from memory  
    d_srcA == M_dstE : M_valE;  
    # Forward valM from write back  
    d_srcA == W_dstM : W_valM;  
    # Forward valE from write back  
    d_srcA == W_dstE : W_valE;  
    # Use value read from register file  
    1 : d_rvalA;  
];
```

# Limitation of Forwarding

```
#demo-luh.ys
```

```
0x000: irmovl $128, %edx
0x006: irmovl $3, %ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10, %ebx
```

**#Load %eax**

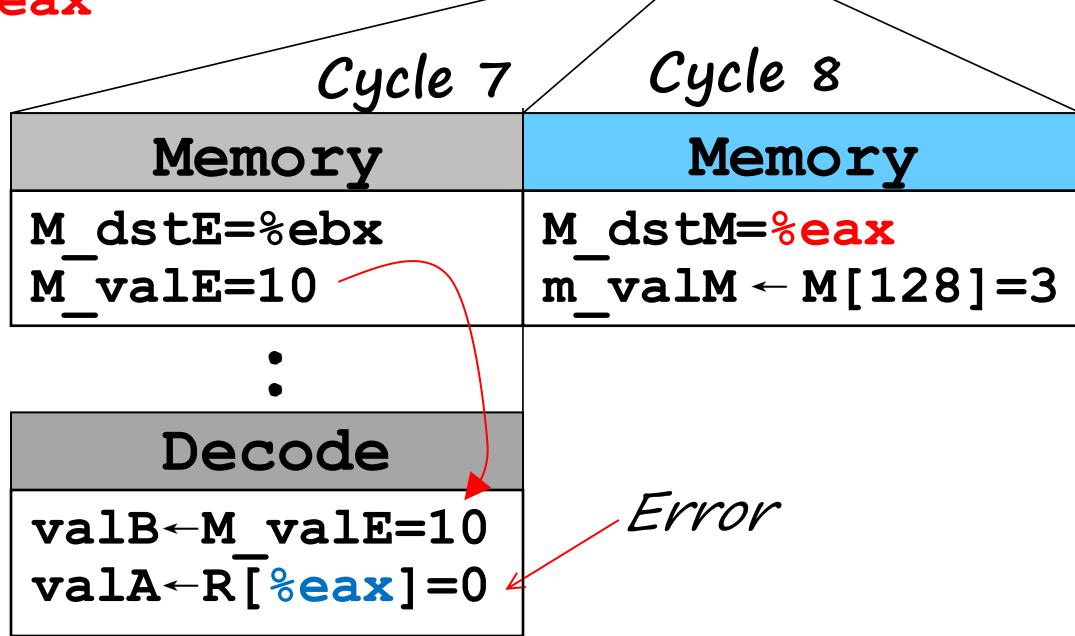
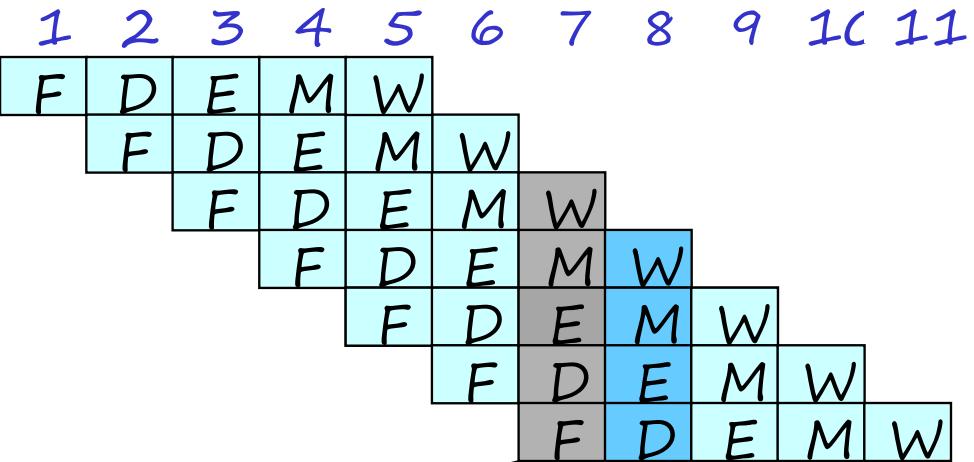
```
0x018: mrmovl 0(%edx), %eax
```

**#Use %eax**

```
0x10e: addl %ebx, %eax
```

```
0x020: halt
```

- Load-use dependency
  - Value needed by end of D stage in cycle 7
  - Value read from memory in M stage of cycle 8



# Avoiding Load/Use Hazard

#demo-luh.ys

```
0x000: irmovl $128, %edx
0x006: irmovl $3, %ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10, %ebx
```

**#Load %eax**

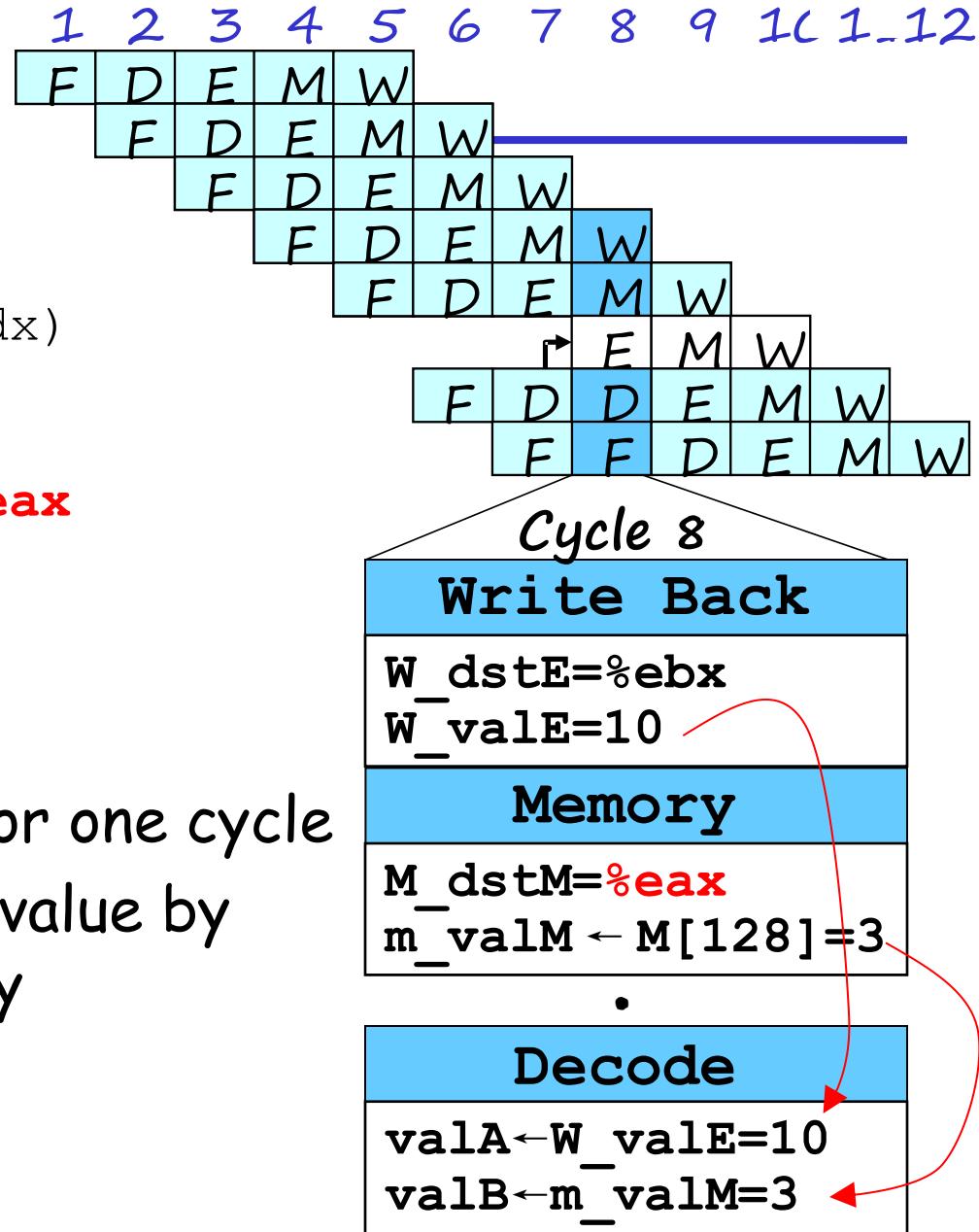
```
0x018: mrmovl 0(%edx), %eax
```

**#Use %eax**

```
0x10e: addl %ebx, %eax
```

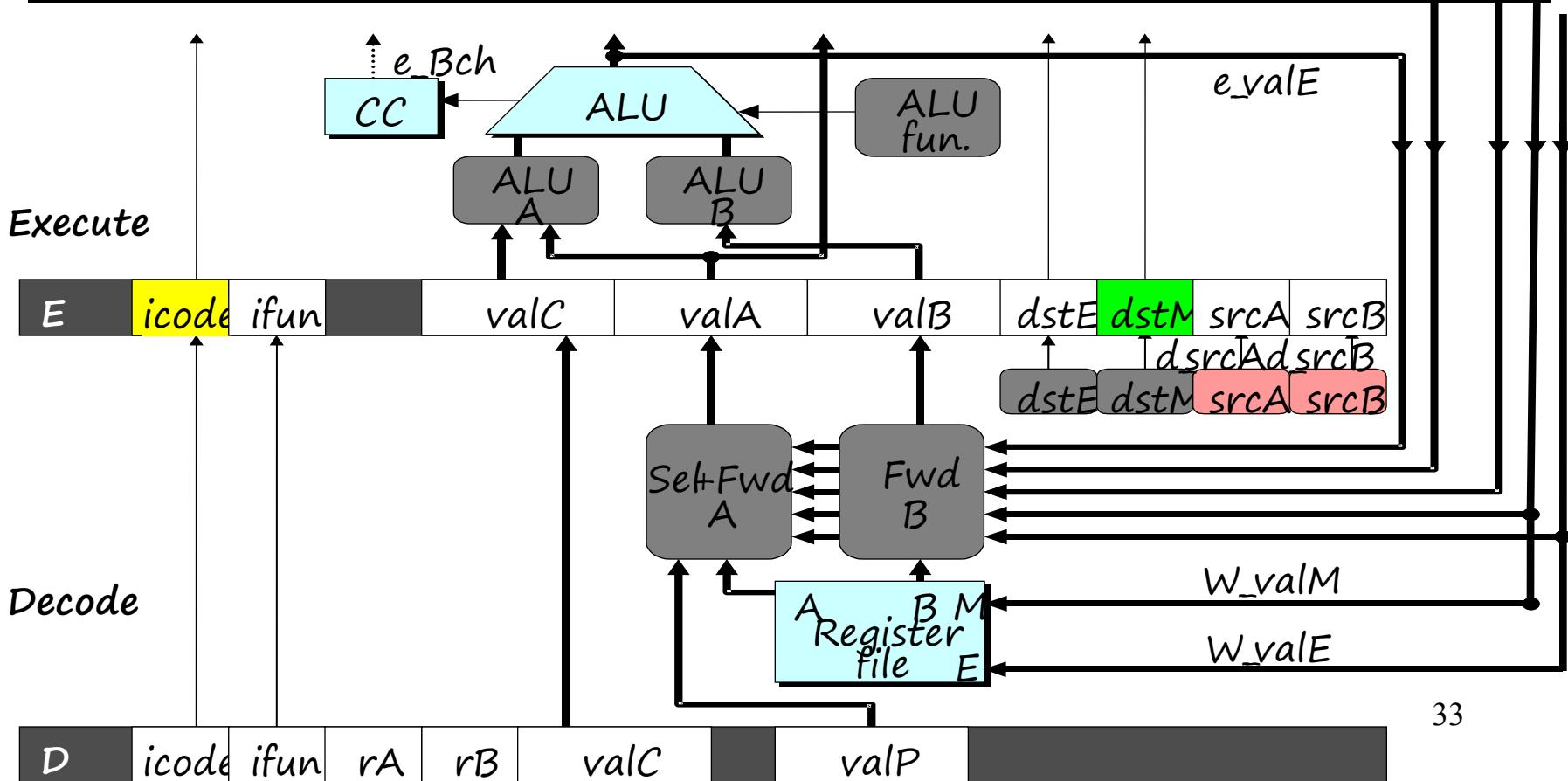
```
0x020: halt
```

- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory



# Detecting Load/Use Hazard

Condition	Trigger
Load/Use Hazard	$E\_icode \in \{ IMRMOVL, IPOPL \} \& \& E\_dstM \in \{ d\_srcA, d\_srcB \}$



# Control for Load/Use Hazard

#demo-luh.ys

```
0x000: irmovl $128, %edx
0x006: irmovl $3, %ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10, %ebx
```

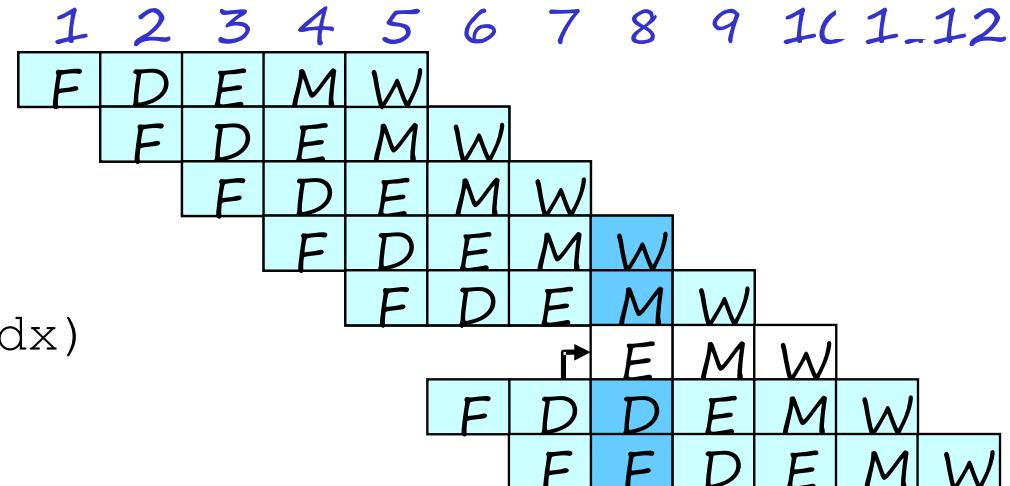
**#Load %eax**

**0x018: mrmovl 0(%edx), %eax**

**#Use %eax**

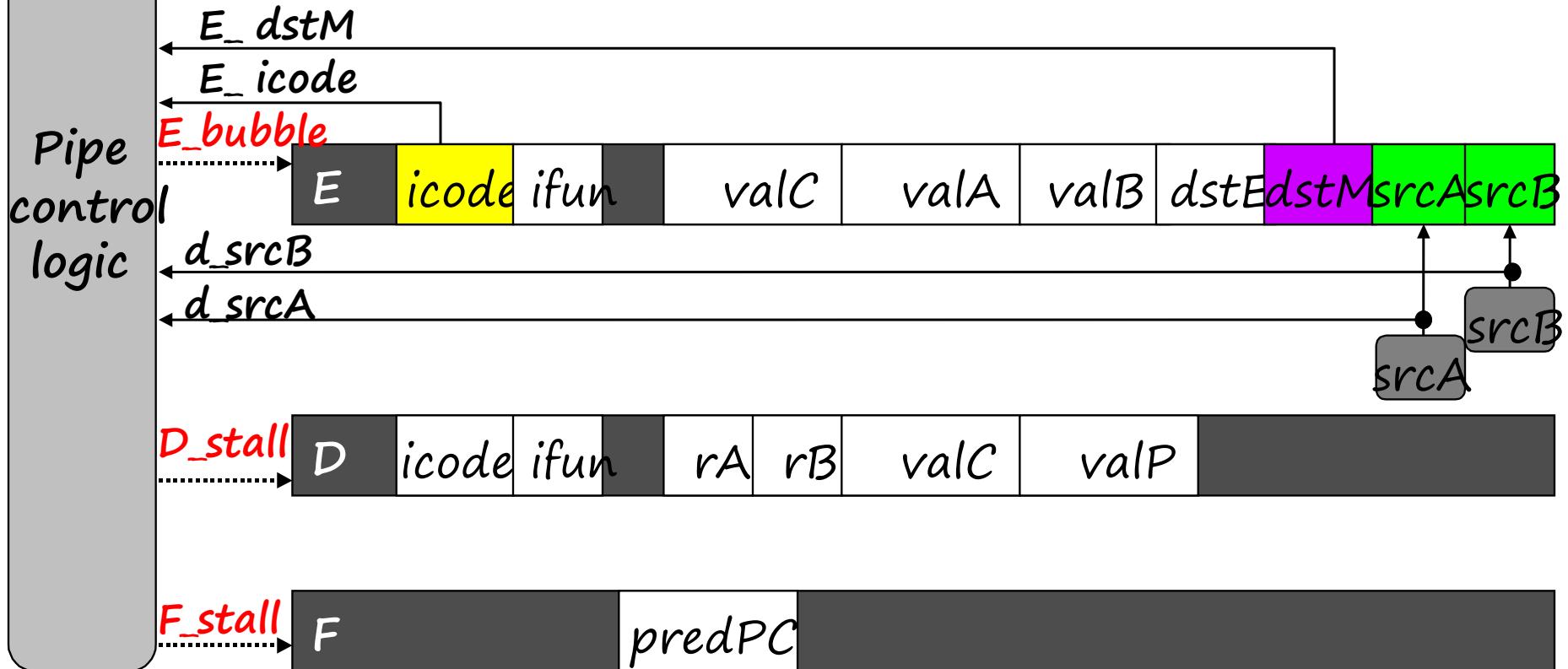
**0x10e: addl %ebx, %eax**

0x020: halt



- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal



# Pipelined Implementation

# Outline

---

- Handle Control Hazard
- Special cases
- Suggested Reading 4.5

# Control Dependence

---

- Example:

loop:

```
    subl %edx, %ebx  
    jne targ  
    irmovl $10, %edx  
    jmp loop
```

targ:

```
    halt
```

- The `jne` instruction creates a control dependency
  - Which instruction will be executed?

# Branch Misprediction Example

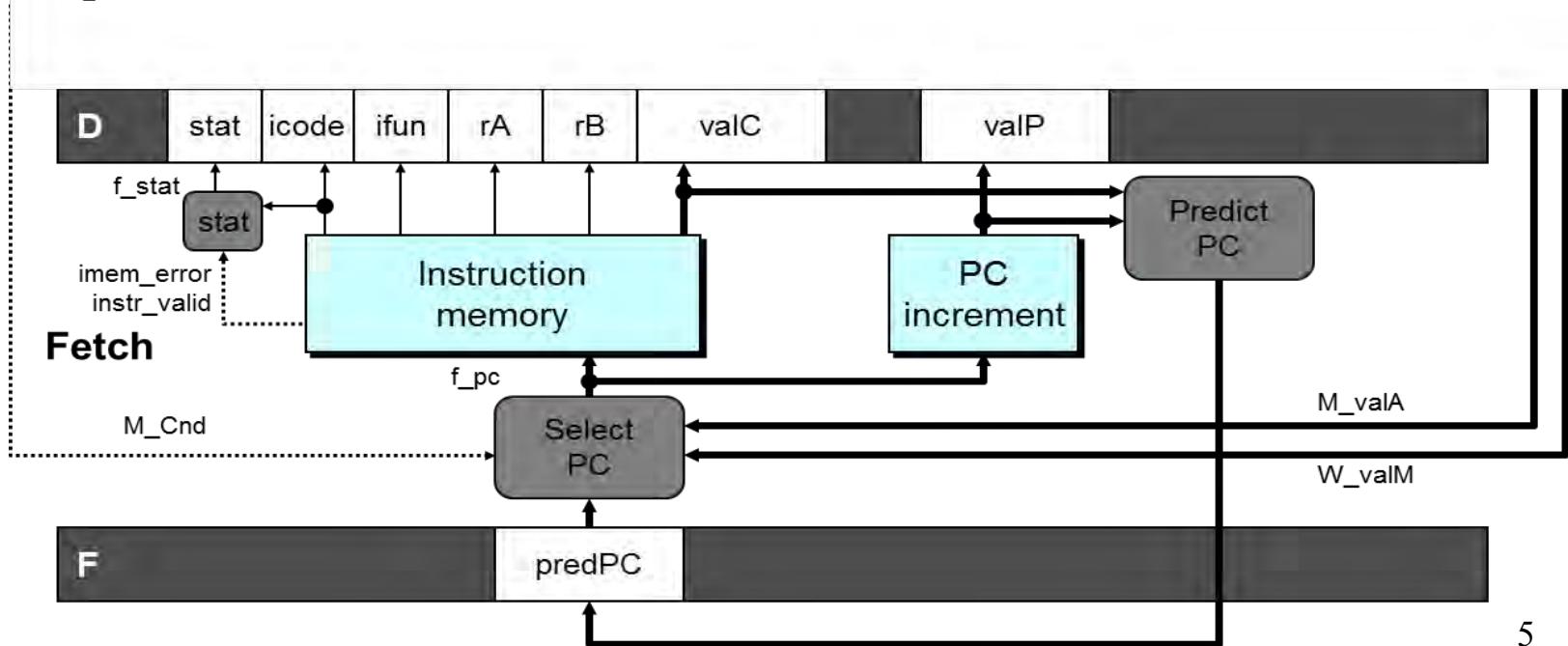
---

```
#demo-j.ys
0x000: xorl %eax,%eax
0x002: jne t          # Not taken
0x007: irmovl $1, %eax # Fall through
0x00d: nop
0x00e: nop
0x00f: nop
0x010: halt
0x011: t: irmovl $3, %edx # Target (Should not execute)
0x017: irmovl $4, %ecx # Should not execute
0x01d: irmovl $5, %edx # Should not execute
```

-Should only execute first 7 instructions

# Select PC

```
int F_predPC = [  
    f_icode in {IJXX, ICALL} : f_valC;  
    1: f_valP;  
];
```

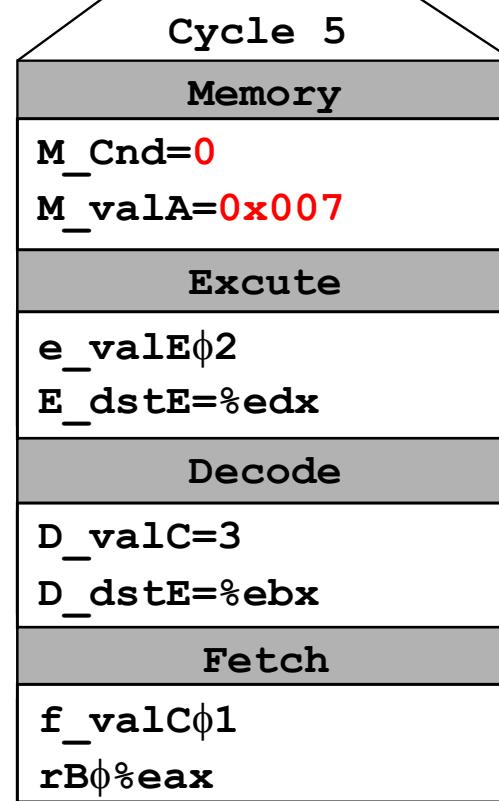


# Branch Misprediction Trace

	1	2	3	4	5	6	7	8	9
1: 0x000: xorl %eax, %eax	F	D	E	M	W				
2: 0x002: jne T # Not taken	F	D	E	M	W				
3: 0x00e: irmovl \$2, %edx # T	F	D	E	M	W				
4: 0x014: irmovl \$3, %ebx # T+1	F	D	E	M	W				
5: 0x007: irmovl \$1, %eax # Fall Through	F	D	E	M	W				

```
# demo-j.ys
0x000: xorl %eax, %eax
0x002: jne T # Not taken
0x007: irmovl $1, %eax
0x00d: halt
0x00e: T:irmovl $2, %edx
0x014: irmovl $3, %ebx
0x01a: halt
```

Incorrectly execute two instructions at branch target



# Select PC

---

```
int f_PC = [  
#mispredicted branch. Fetch at incremented PC  
    M_icode == IJXX && !M_Cnd : M_valA;  
#completion of RET instruciton  
    W_icode == IRET : W_valM;  
#default: Use predicted value of PC  
    1: F_predPC  
] ;
```

# Return Example

```
#demo-ret.ys
0x000:    irmovl Stack,%esp    # Intialize stack pointer
0x006:    nop                  # Avoid hazard on %esp
0x007:    nop
0x008:    nop
0x009:    call p                # Procedure call
0x00e:    irmovl $5,%esi      # Return point
0x014:    halt
0x020: .pos 0x20
0x020: p:    nop              # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovl $1,%eax      # Should not be executed
0x02a:    irmovl $2,%ecx      # Should not be executed
0x030:    irmovl $3,%edx      # Should not be executed
0x036:    irmovl $4,%ebx      # Should not be executed
0x100: .pos 0x100
0x100: Stack:                 # Stack: Stack pointer
```

- Require lots of nops to avoid data hazards

# Incorrect Return Example

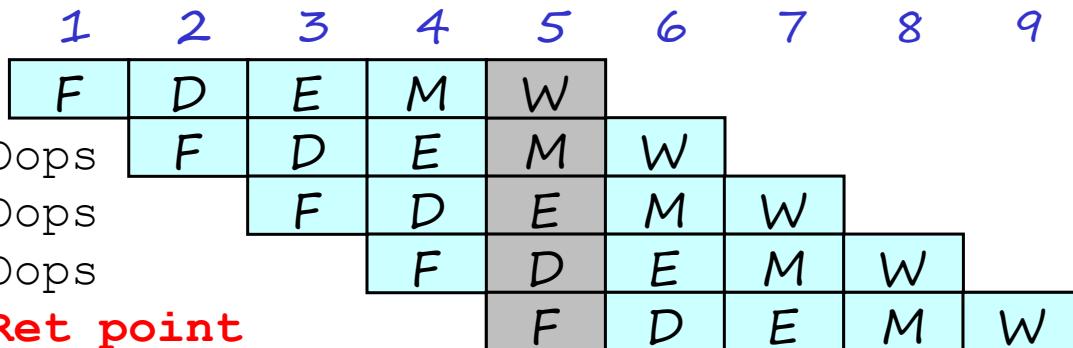
1: 0x023: ret

2: 0x024: irmovl \$1,%eax #Oops

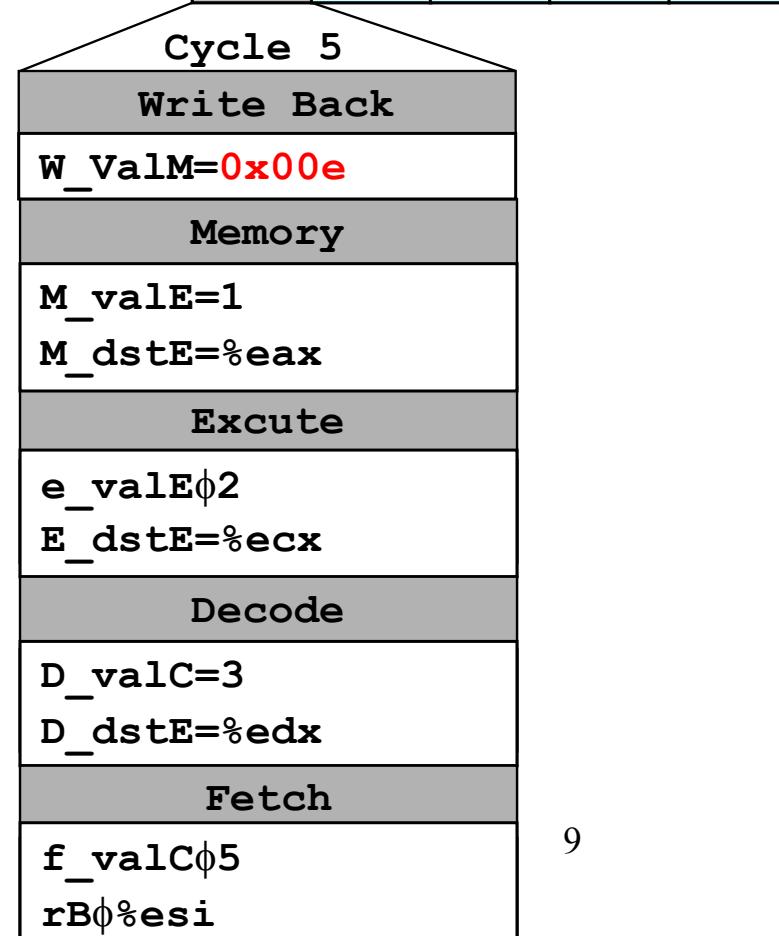
3: 0x02a: irmovl \$2,%ecx #Oops

4: 0x030: irmovl \$3,%edx #Oops

5: 0x00e: irmovl \$5,%esi #Ret point



Incorrectly execute 3 instructions  
following ret



# Handling Misprediction

---

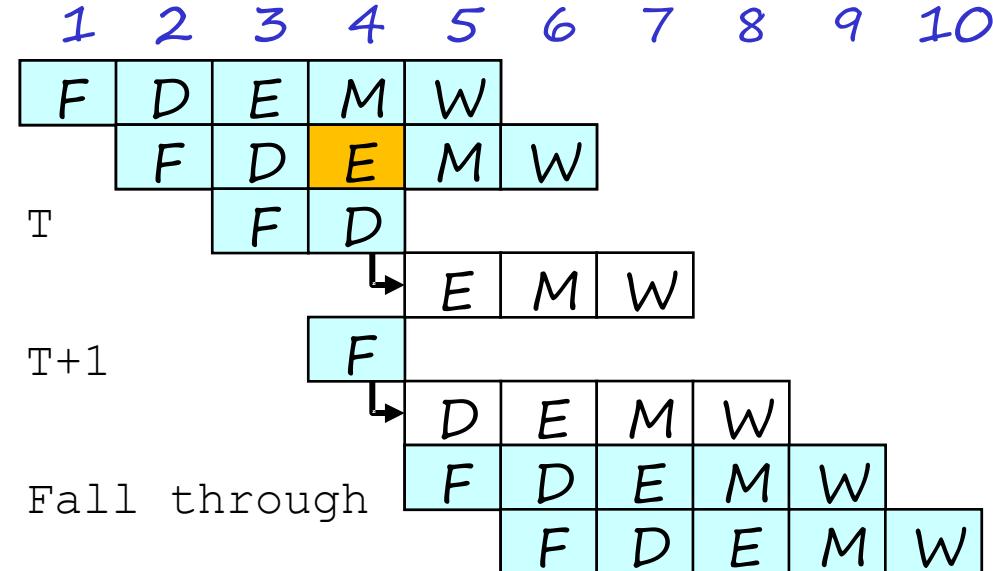
```
# demo-j.ys
0x000: xorl %eax,%eax
0x002: jne T # Not taken
0x007: irmovl $1, %eax
0x00d: halt
0x00e: T:irmovl $2, %edx
0x014: irmovl $3, %ebx
0x01a: halt
```

```
# demo-j2.ys
0x000: xorl %eax,%eax
0x002: jne T # Not taken
0x007: irmovl $1, %eax
0x00d: halt
0x00e: T:nop
0x00f: nop
0x010: irmovl $2, %edx
0x016: irmovl $3, %ebx
0x01c: halt
```

# Handling Misprediction

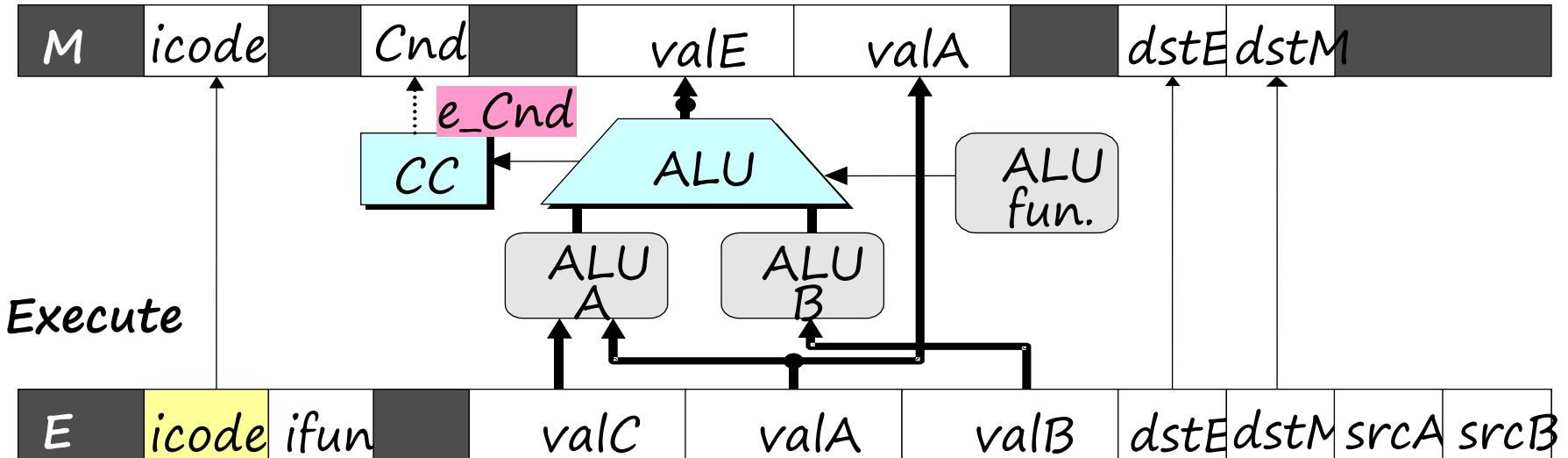
#demo-j.ys

```
1: 0x000: xorl %eax, %eax
2: 0x002: jne T #Not Taken
3: 0x00e: irmovl $2, %edx # T
4:      bubble
5: 0x014: irmovl $3, %ebx # T+1
6:      bubble
7: 0x007: irmovl $1, %eax # Fall through
8: 0x00d: halt
```



- Predict branch as taken
  - Fetch 2 instructions at target
- Cancel when mispredicted
  - Detect branch not-taken in execute stage
  - On following cycle, replace instructions in execute and decode by bubbles
  - No side effects have occurred yet

# Detecting Mispredicted Branch



Condition	Trigger
Mispredicted Branch	$E\_icode = IJXX \& !e\_Cnd$

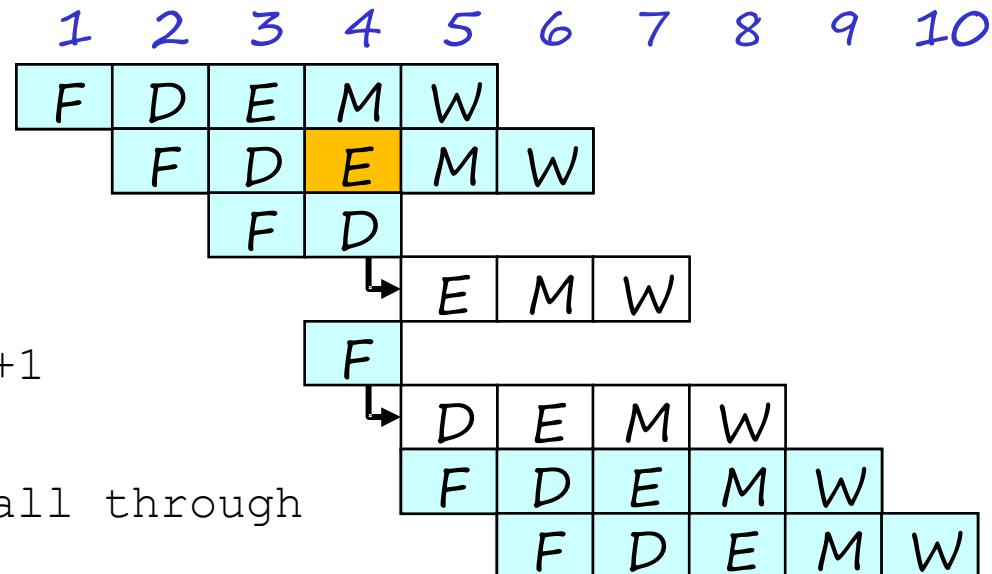
# Control for Misprediction

#demo-j.ys

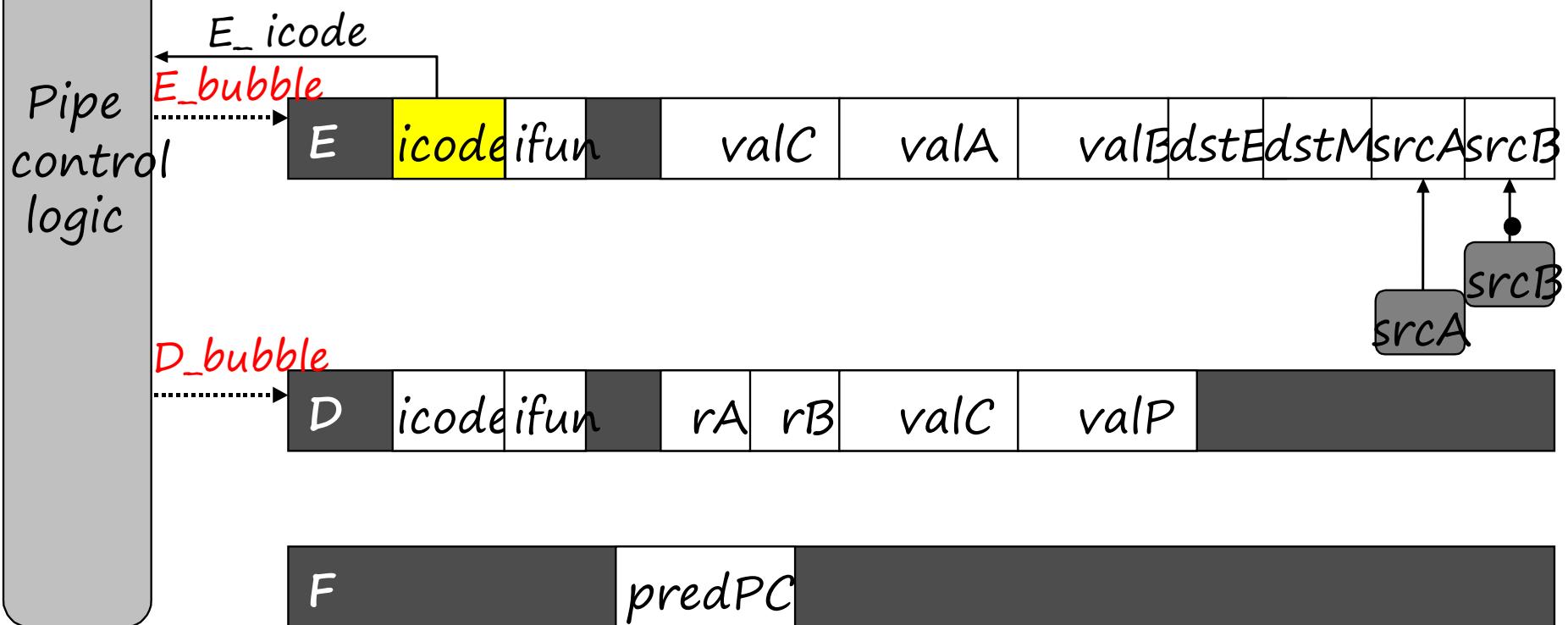
```

1: 0x000: xorl %eax, %eax
2: 0x002: jne T #Not Taken
3: 0x00e: irmovl $2, %edx # T
4:          bubble
5: 0x014: irmovl $3, %ebx # T+1
6:          bubble
7: 0x007: irmovl $1, %eax # Fall through
8: 0x00d: halt

```



Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal



# Return Example

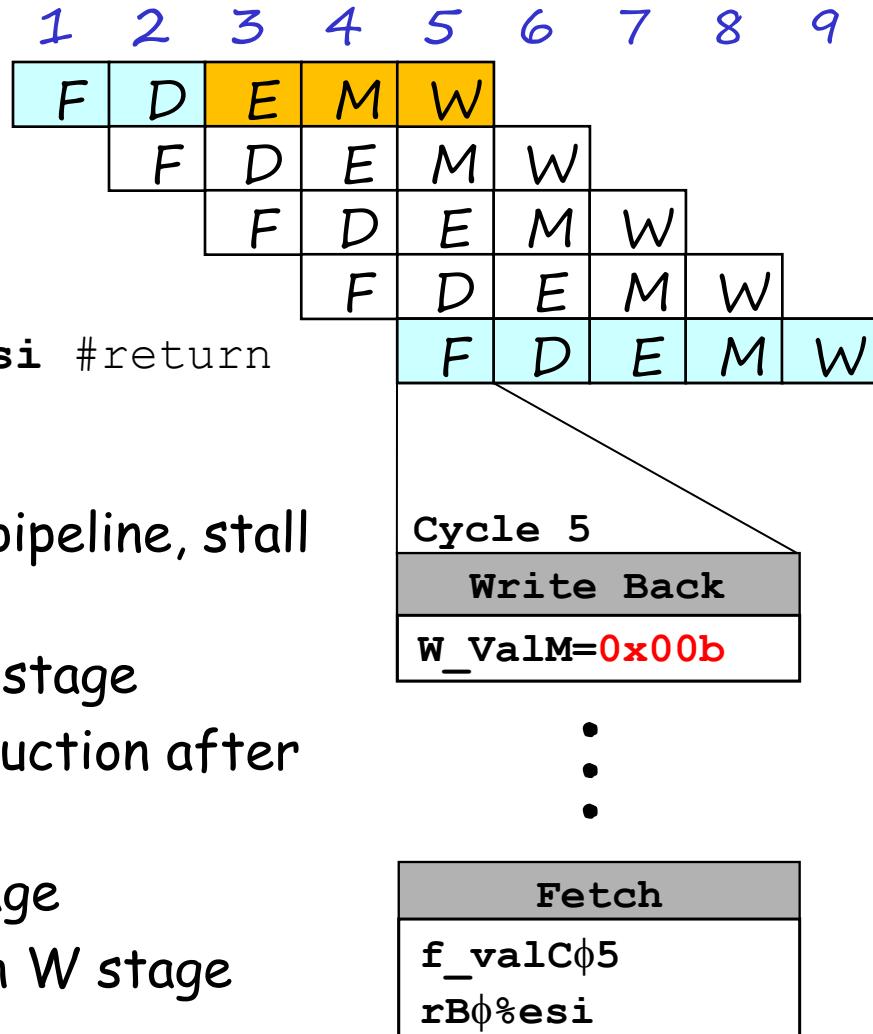
```
#demo-retB.ys
0x000:    irmovl Stack,%esp    # Intialize stack pointer
0x006:    call p                # Procedure call
0x00b:    irmovl $5,%esi      # Return point
0x011:    halt
0x020: .pos 0x20
0x020: p: irmovl $-1,%edi    # procedure
0x026:    ret
0x027:    irmovl $1,%eax      # Should not be executed
0x02d:    irmovl $2,%ecx      # Should not be executed
0x033:    irmovl $3,%edx      # Should not be executed
0x039:    irmovl $4,%ebx      # Should not be executed
0x100: .pos 0x100
0x100: Stack:                 # Stack: Stack pointer
```

- Previously executed three additional instructions

# Correct Return Example

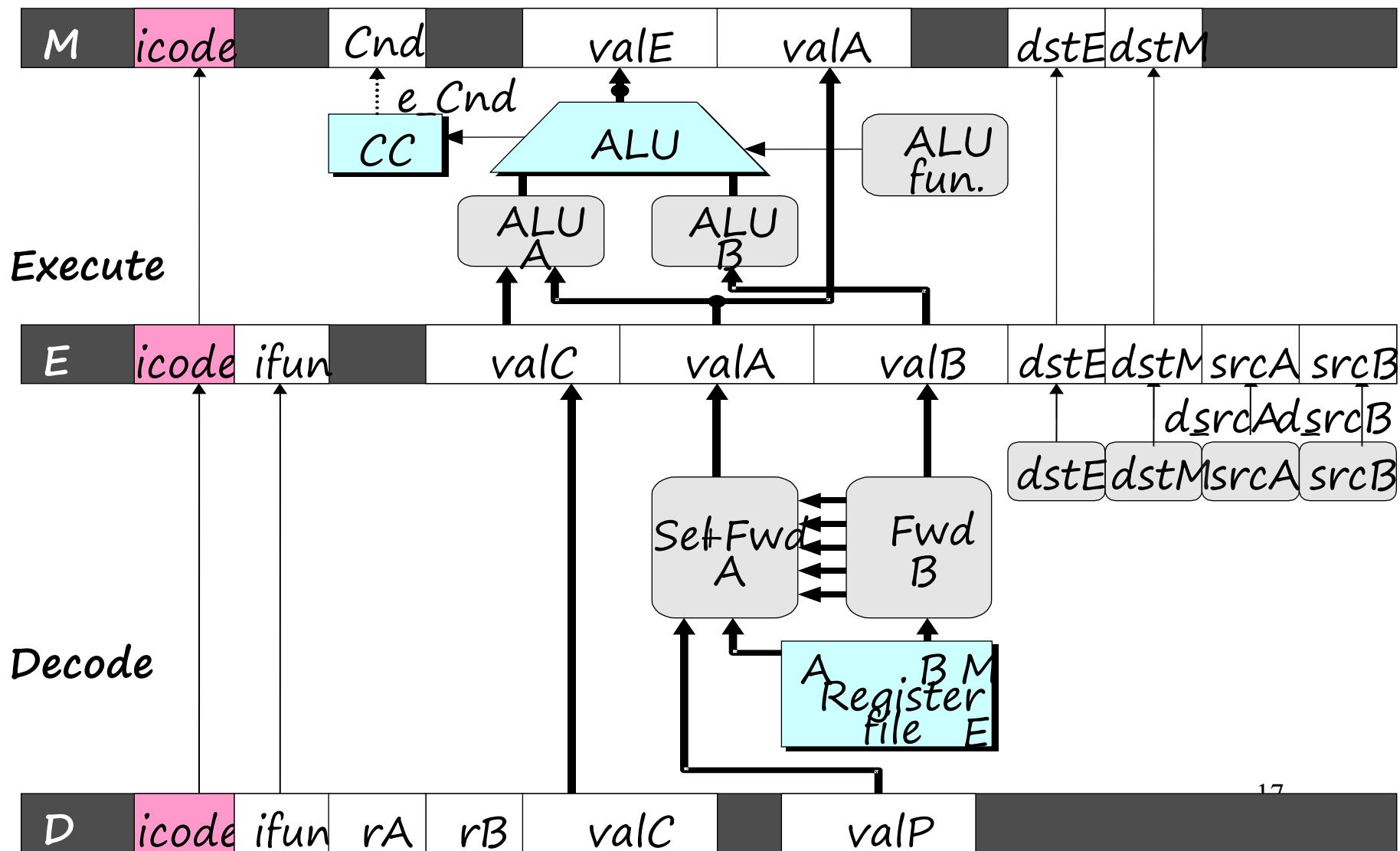
```
#demo retb
```

```
1: 0x026: ret
2:      bubble
3:      bubble
4:      bubble
5: 0x00b: irmovl $5, %esi #return
```



- As `ret` passes through pipeline, stall at F stage
  - While in D, E, and M stage
  - fetch the same instruction after `ret` 3 times.
- Inject bubble into D stage
- Release stall when reach W stage

# Detecting Return



# Control for Return

---

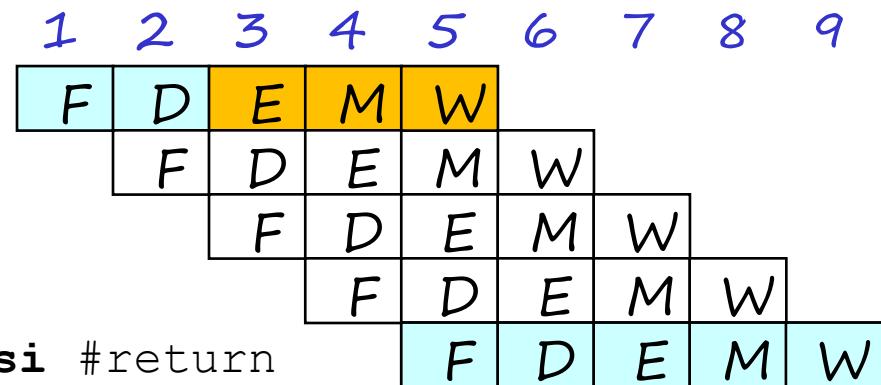
Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }

#demo\_retb

```

1: 0x026: ret
2:      bubble
3:      bubble
4:      bubble
5: 0x00b: irmovl $5, %esi #return

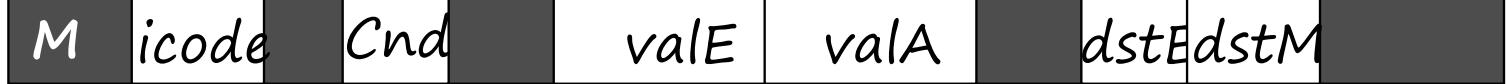
```



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal



$M\_icode$



$E\_icode$



Pipe  
control  
logic

$D\_icode$

$D\_bubble$



$F\_stall$



# Control Cases

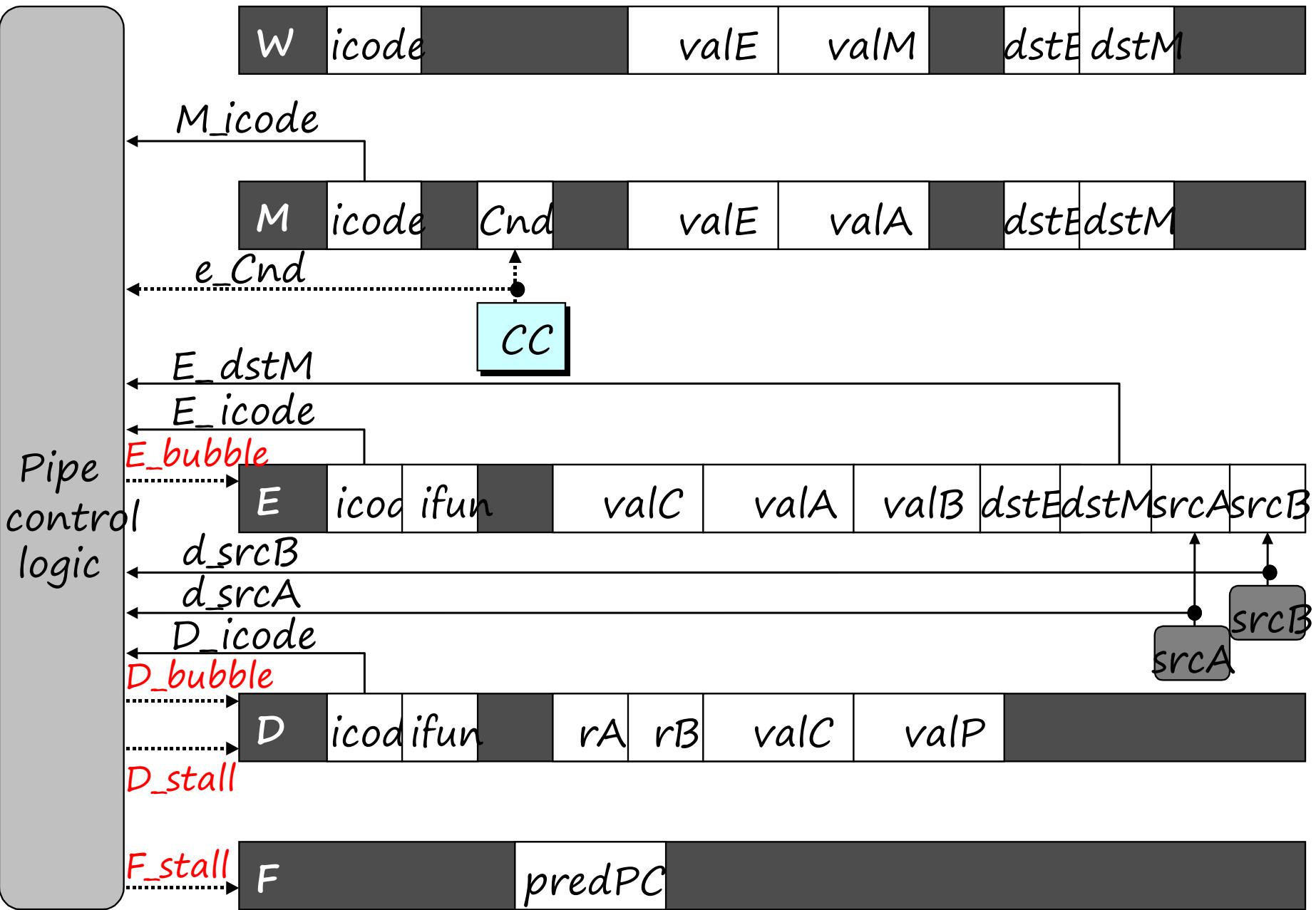
---

- Detection

Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Cnd

- Action

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal



# Implementing Pipeline Control

---

- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

# Initial Version of Pipeline Control

---

```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
    E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch
    while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode } ;

bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
    E_dstM in { d_srcA, d_srcB } ;
```

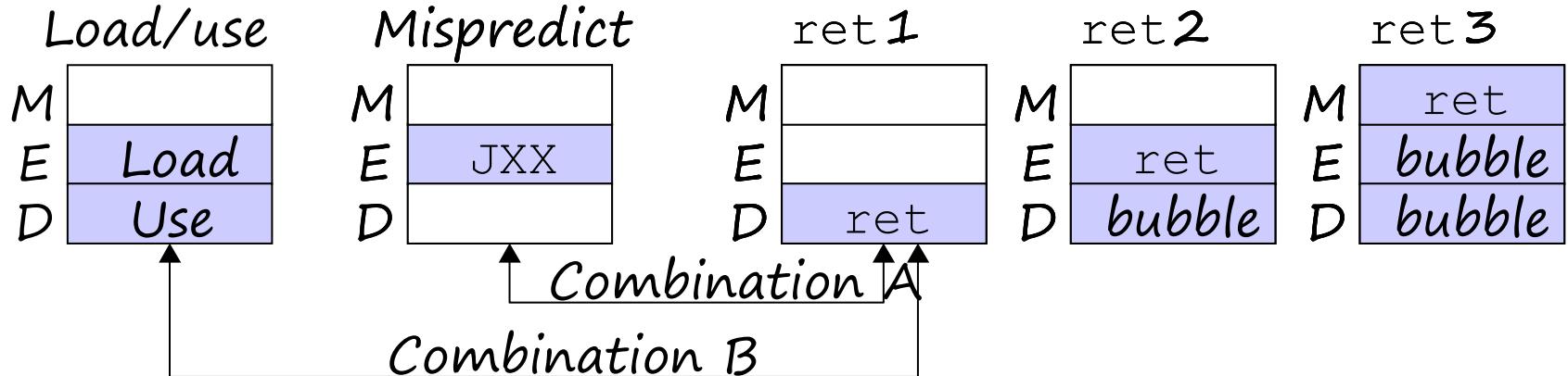
# Initial Version of Pipeline Control

---

```
bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Stalling at fetch
    while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

```
bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
    E_dstM in { d_srcA, d_srcB};
```

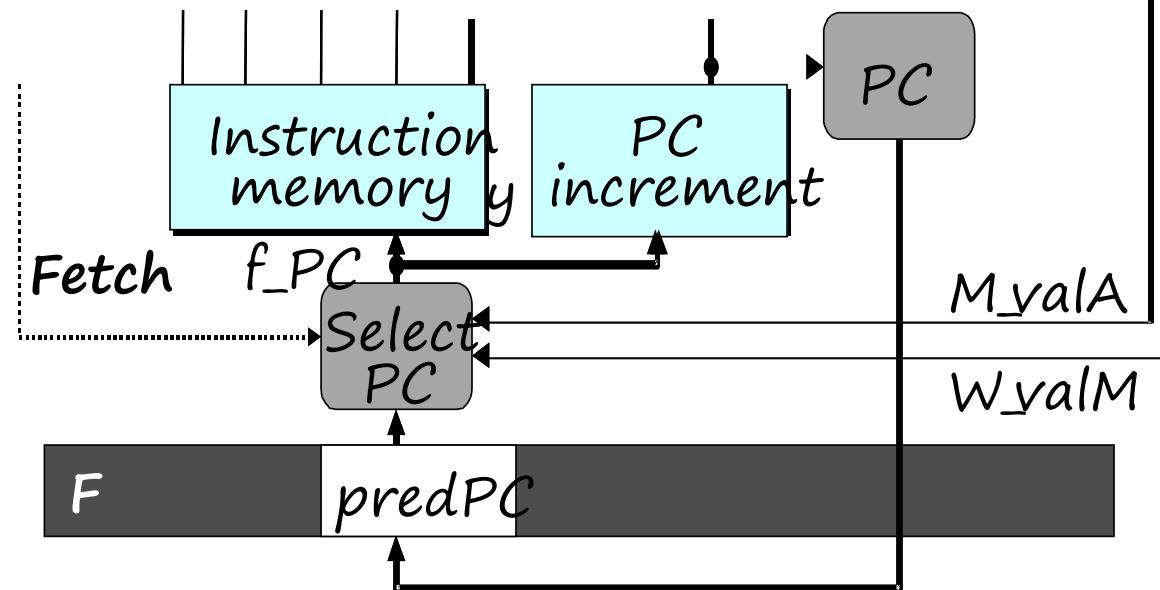
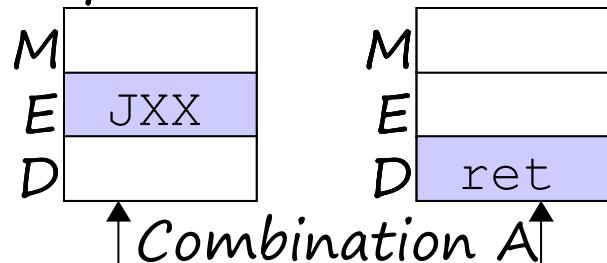
# Control Combinations



- Special cases that can arise on same clock cycle
- Combination A
  - Not-taken branch
  - ret instruction at branch target
- Combination B
  - Instruction that reads from memory to %esp
  - Followed by ret instruction

# Control Combination A

Mispredict



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

# Control Combination A

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

- Should handle as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M\_valA anyhow

# Control Combination B



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble + stall	bubble	normal	normal

- Would attempt to bubble and stall pipeline register D
- Signaled by processor as pipeline error

# Handling Control Combination B

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	stall	bubble	normal	normal

- Load/use hazard should get priority
- ret instruction should be held in decode stage for additional cycle

# Corrected Pipeline Control Logic

---

```
bool D_bubble =  
    # Mispredicted branch  
    (E_icode == IJXX && !e_Cnd) ||  
    # Stalling at fetch  
    while ret passes through pipeline  
        IRET in { D_icode, E_icode, M_icode }  
        # but not condition for a load/use hazard  
        && !(E_icode in { IMRMOVL, IPOPL }  
            && E_dstM in { d_srcA, d_srcB });
```

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	stall	bubble	normal	normal

# Pipeline Summary

---

- Data Hazards
  - Most handled by forwarding
    - No performance penalty
  - Load/use hazard requires one cycle stall
- Control Hazards
  - Cancel instructions when detect mispredicted branch
    - Two clock cycles wasted
  - Stall fetch stage while `ret` passes through pipeline
    - Three clock cycles wasted

# Pipeline Summary

---

- Control Combinations
  - Must analyze carefully
  - First version had subtle bug
    - Only arises with unusual instruction combination

# Exception Handling

# Outline

---

- Exception Handling
- Suggested Reading 4.5.9

# Exceptions

---

- Condition: instruction encounter an error condition
- Deal flow:
  - Break the program flow
  - Invoke the exception handler provided by OS.
  - (Maybe) continue the program flow
    - E.g. page fault exception

# Exceptions

- Conditions under which processor cannot continue normal operation
- Causes
  - Halt instruction (Current)
  - Bad address for instruction or data (Previous)
  - Invalid instruction (Previous)
- Typical Desired Action
  - Complete some instructions
    - Either current or previous (depends on exception type)
  - Discard others
  - Call exception handler
    - Like an unexpected procedure call

# Exception Examples

---

- Detect in Fetch Stage

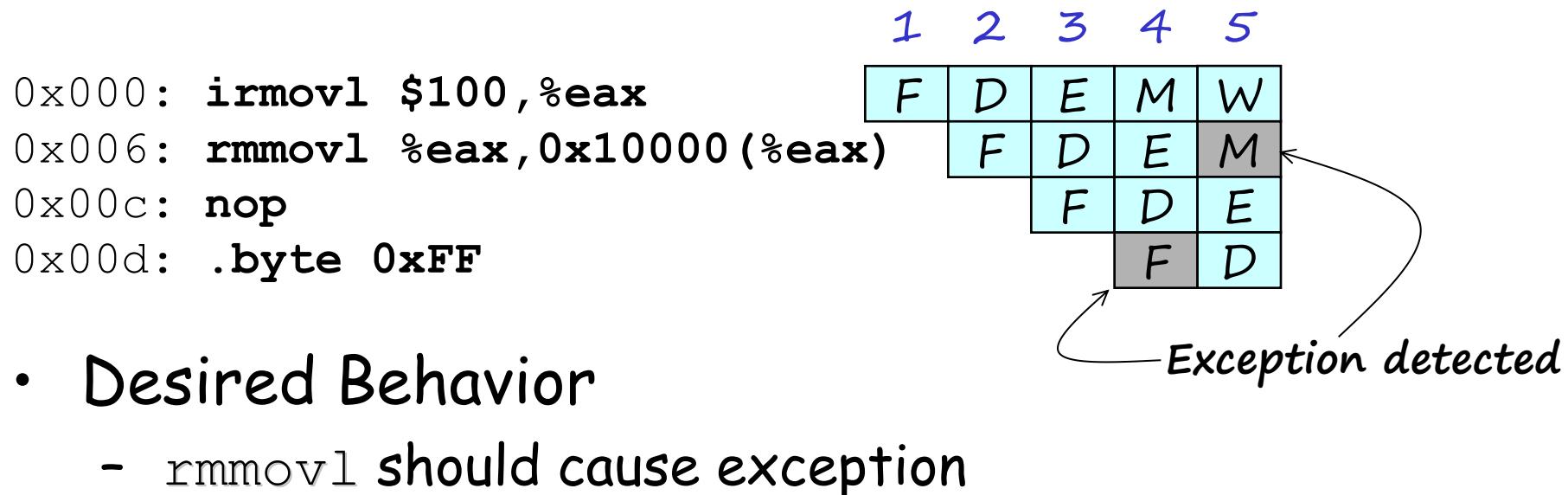
```
jmp $-1          # Invalid jump target  
.byte 0xFF      # Invalid instruction code  
halt            # Halt instruction
```

- Detect in Memory Stage

```
irmovl $100,%eax  
rmmovl %eax,0x10000(%eax) # invalid address
```

# Exceptions in Pipeline Processor #1

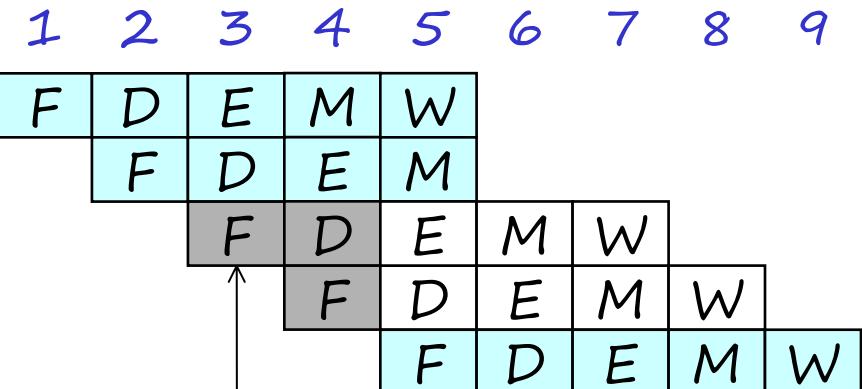
```
# demo-excl.ys
irmovl $100,%eax
rmmovl %eax,0x10000(%eax) # Invalid address
nop
.byte 0xFF # Invalid instruction code
```



# Exceptions in Pipeline Processor #2

```
# demo-exc2.ys
0x000: xorl %eax,%eax      # Set condition codes
0x002: jne t                  # Not taken
0x007: irmovl $1,%eax
0x00d: irmovl $2,%edx
0x013: halt
0x014: t: .byte 0xFF         # Target
```

0x000: xorl %eax,%eax  
0x002: jne t  
0x014: t: .byte 0xFF  
**0x???: (I'm lost!)**  
0x007: irmovl \$1,%eax



- Desired Behavior
  - No exception should occur

Exception  
detected

# Maintaining Exception Ordering

W	stat	icode		valE	valM	dstEdstm	
M	stat	icode	Cnd	valE	valA	dstEdstm	
E	stat	icode	ifun	valC	valA	valB	dstEdstNsraSrcAsrcB
D	stat	icode	ifun	rA	rB	valC	valP
F	predPC						

- Add exception status field to pipeline registers
- Fetch stage sets to either "AOK", "ADR" (when bad fetch address), "HTL" (halt instruction) or "INS" (illegal instruction)
- Decode & Execute pass values through
- Memory either passes through or sets to "ADR"
- Exception triggered only when instruction hits Write-back

# Exception Handling Logic

```
int Stat = [
    # SBUB in earlier stages indicates bubble
    W_stat == SBUB : SAOK;
    1 : W_stat;
];
```

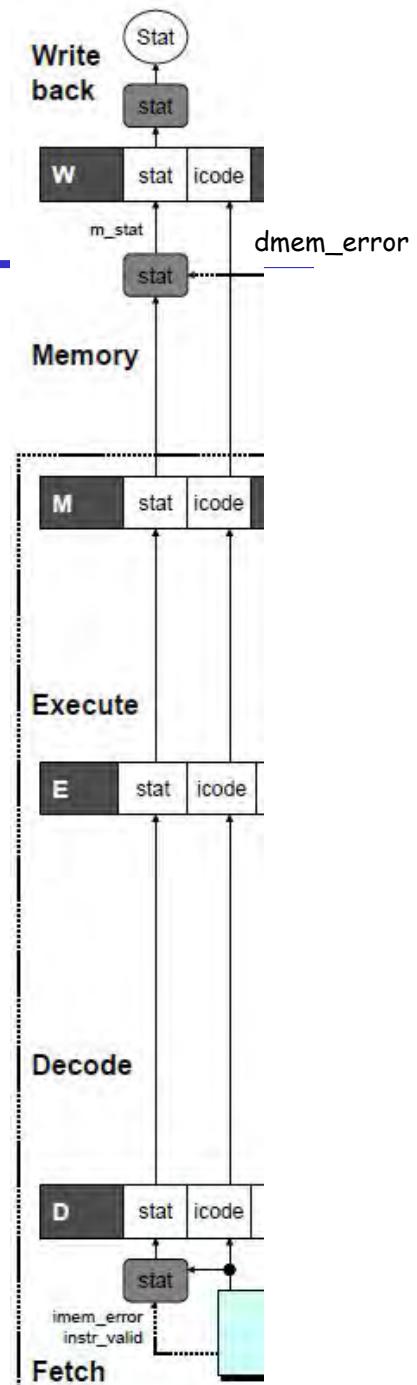
Write-back Stage

```
# Update the status
int m_stat = [
    dmem_error : SADR;
    1 : M_stat;
];
```

Memory Stage

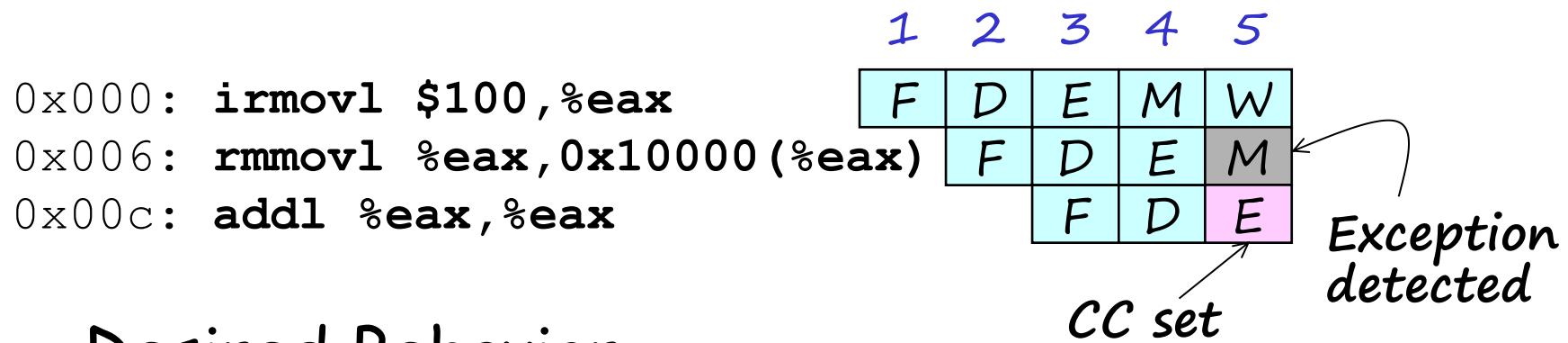
```
# Determine status code for fetched instruction
int f_stat = [
    imem_error: SADR;
    !instr_valid : SINS;
    f_icode == IHALT : SHLT;
    1 : SAOK;
];
```

Fetch Stage



# Side Effects in Pipeline Processor

```
# demo-exc3.ys
irmovl $100,%eax
rmmovl %eax,0x10000(%eax) # invalid address
addl %eax,%eax           # Sets condition codes
```



- Desired Behavior
  - rmmovl should cause exception
  - No following instruction should have any effect

# Avoiding Side Effects

---

- Presence of Exception Should Disable State Update
  - When detect exception in Memory stage
    - Disable condition code setting in Execute stage
    - Must happen in same clock cycle
  - When exception passes to Write-back stage
    - Disable memory write in Memory stage
    - Disable condition code (CC) setting in Execute stage
- Implementation
  - Hardwired into the design of the PIPE simulator
  - You have no control over this

# Avoiding Side Effects

---

- Presence of Exception Should Disable State Update
  - Invalid instructions are converted to pipeline bubbles
    - Except have stat indicating exception status
  - Data memory will not write to invalid address
  - Prevent invalid update of condition codes
    - Detect exception in memory stage
    - Disable condition code setting in execute
    - Must happen in same clock cycle

# Avoiding Side Effects

---

- Presence of Exception Should Disable State Update
  - Handling exception in final stages
    - When detect exception in memory stage
      - Start injecting bubbles into memory stage on next cycle
    - When detect exception in write-back stage
      - Stall excepting instruction
  - Included in HCL code

# Control Logic for State Changes

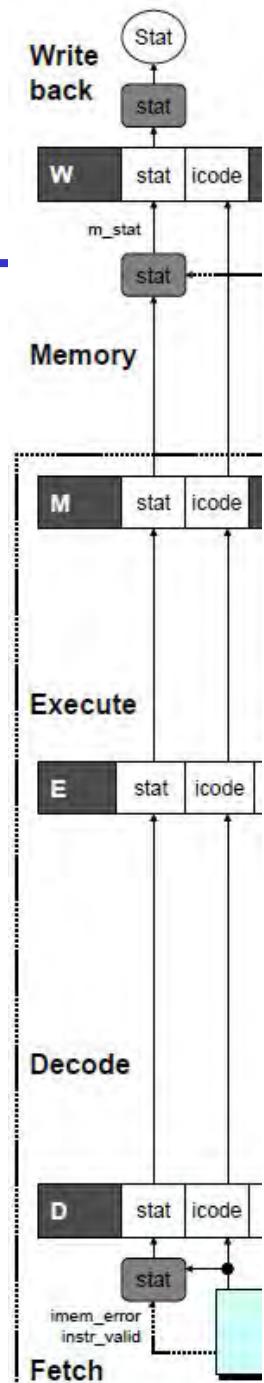
## Setting Condition Codes

```
# Should the condition codes be updated?  
bool set_cc = (E_icode == IOPL)  
    # State changes only during normal operation  
    && !m_stat in { SADR, SINS, SHLT }  
    && !W_stat in { SADR, SINS, SHLT };
```

## Stage Control

- Also controls updating of memory

```
# Start injecting bubbles as soon as exception passes  
through memory stage  
bool M_bubble = m_stat in { SADR, SINS, SHLT }  
    || W_stat in { SADR, SINS, SHLT };  
  
# Stall pipeline register W when exception encountered  
bool W_stall = W_stat in { SADR, SINS, SHLT };
```



# Rest of Exception Handling

---

- Calling Exception Handler
  - Push PC onto stack
    - Either PC of faulting instruction or of next instruction
    - Usually pass through pipeline along with exception status
  - Jump to handler address
    - Usually fixed address
    - Defined as part of ISA
- Implementation
  - Haven't tried it yet!

# Processor Summary

---

- Design Technique
  - Create uniform framework for all instructions
    - Want to share hardware among instructions
  - Connect standard logic blocks with bits of control logic
- Operation
  - State held in memories and clocked registers
  - Computation done by combinational logic
  - Clocking of registers/memories sufficient to control overall behavior
- Enhancing Performance
  - Pipelining increases throughput and improves resource utilization
  - Must make sure maintains ISA behavior

# Performance of Processor

# Outline

---

- Performance Analysis
- Suggested Reading 4.5.12

# Performance Metrics

---

- Clock rate
  - Measured in Gigahertz
  - Function of stage partitioning and circuit design
    - Keep amount of work per stage small
- Rate at which instructions executed
  - CPI: cycles per instruction
  - On average, how many clock cycles does each instruction require?
  - Function of pipeline design and benchmark programs
    - E.g., how frequently are branches mispredicted?

# CPI for PIPE

---

- $CPI \approx 1.0$ 
  - Fetch instruction each clock cycle
  - Effectively process new instruction almost every cycle
    - Although each individual instruction has latency of 5 cycles
- $CPI > 1.0$ 
  - Sometimes must stall or cancel branches

# CPI for PIPE

---

- Computing CPI
  - C: clock cycles
  - I: instructions executed to completion
  - B: bubbles injected ( $C = I + B$ )
$$CPI = C/I = (I+B)/I = 1.0 + B/I$$
  - Factor  $B/I$  represents average penalty due to bubbles

## CPI for PIPE (Cont.)

- LP: Penalty due to load/use hazard stalling Typical Values
    - Fraction of instructions that are loads 0.25
    - Fraction of load instructions requiring stall 0.20
    - Number of bubbles injected each time 1
$$\Rightarrow LP = 0.25 * 0.20 * 1 = 0.05$$
  - MP: Penalty due to mispredicted branches
    - Fraction of instructions that are cond. jumps 0.20
    - Fraction of cond. jumps mispredicted 0.40
    - Number of bubbles injected each time 2
$$\Rightarrow MP = 0.20 * 0.40 * 2 = 0.16$$

## CPI for PIPE (Cont.)

---

- RP: Penalty due to `ret` instructions *Typical Values*
  - Fraction of instructions that are returns 0.02
  - Number of bubbles injected each time 3
$$\Rightarrow RP = 0.02 * 3 = 0.06$$
- Net effect of penalties  $0.05 + 0.16 + 0.06 = 0.27$ 
$$\Rightarrow CPI = 1.27 \quad (\text{Not bad!})$$

$$B/I = LP + MP + RP$$

# **Modern Processors**

# Outline

---

- Understanding Modern Processor
  - Super-scalar
  - Out-of -order execution
- Suggested reading
  - 5.7

# Review

---

- Machine-Independent Optimization
  - Eliminating loop inefficiencies
  - Reducing procedure calls
  - Eliminating unneeded memory references

# Review

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;

    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

```
void combine4(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t x = IDENT;

    for (i = 0; i < length; i++)
        x = x OP data[i];
    *dest = x;
}
```

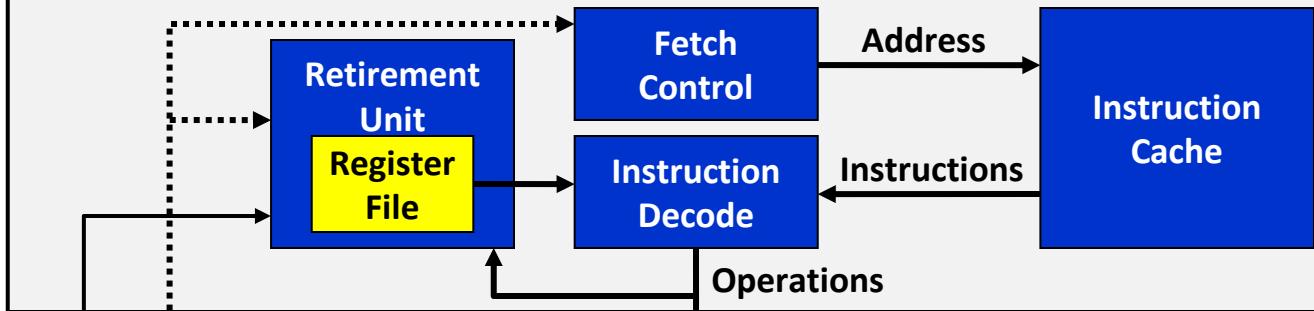
Function	Integer		Floating Point		
	+	*	+	F*	D*
combine1	12	12	12	12	13
combine4	2	3	3	4	5

# Modern Processor

---

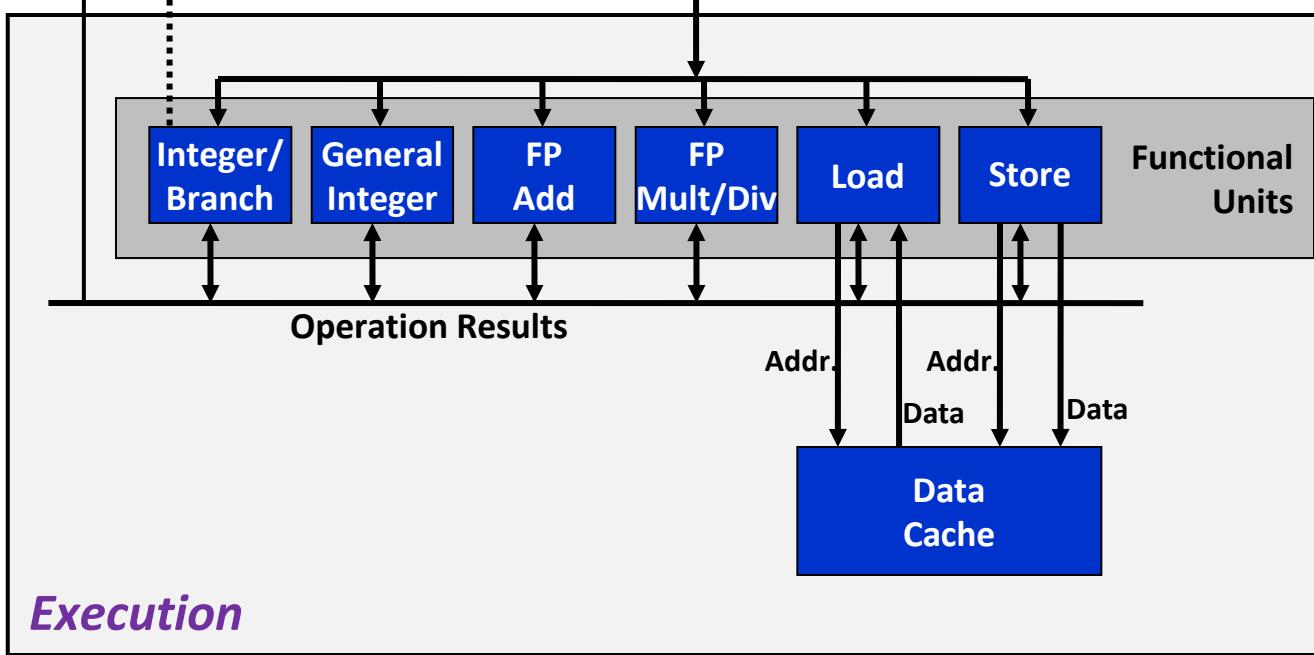
- Superscalar
  - Perform multiple operations on every clock cycle
  - Instruction level parallelism
- Out-of-order execution
  - The order in which the instructions execute need not correspond to their ordering in the assembly program

## *Instruction Control*



Register Updates

Prediction OK?



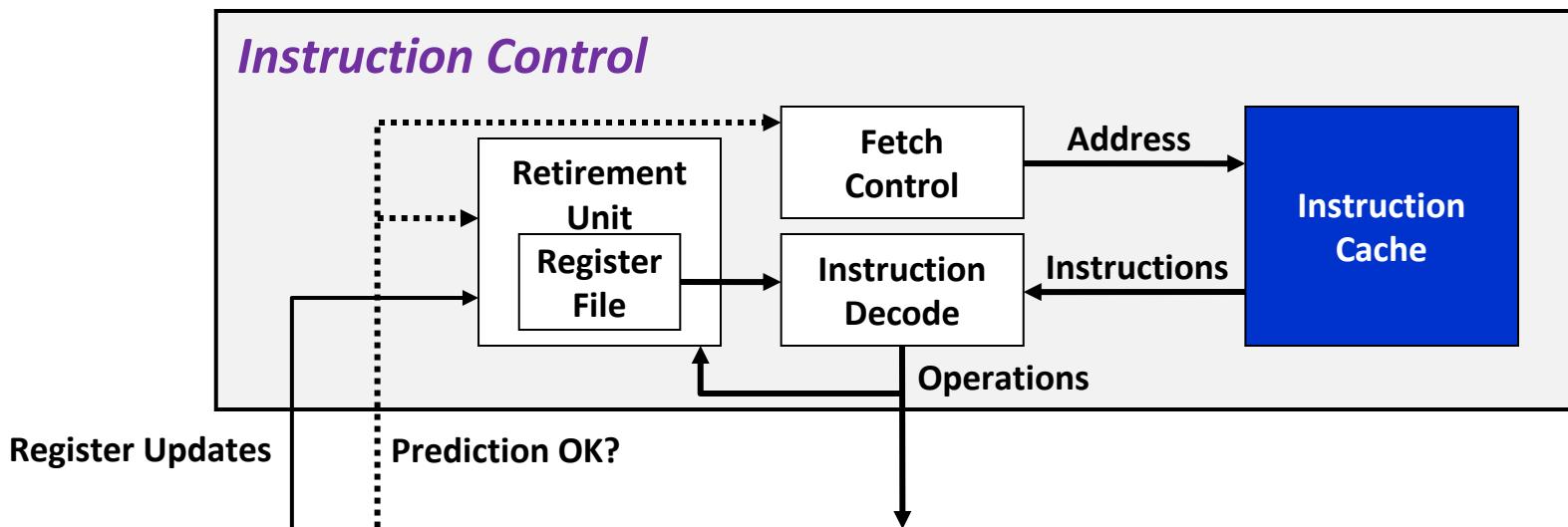
# Modern Processor

---

- Two main parts
  - Instruction Control Unit (ICU)
    - Responsible for reading a sequence of instructions from memory
    - Generating from above instructions a set of primitive operations to perform on program data
  - Execution Unit (EU)
    - Execute these operations

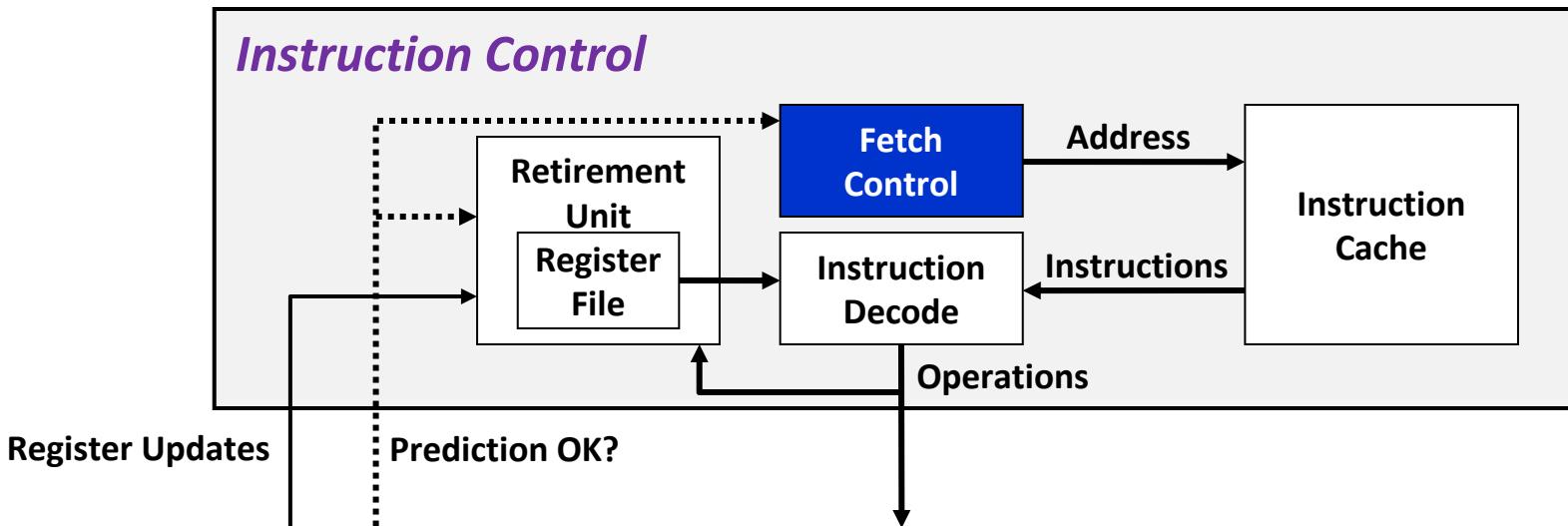
# Instruction Control Unit

- Instruction Cache
  - A special, high speed memory containing the most recently accessed instructions.



# Instruction Control Unit

- Fetch Control
  - Fetches ahead of currently accessed instructions
    - enough time to decode instructions and send decoded operations down to the EU



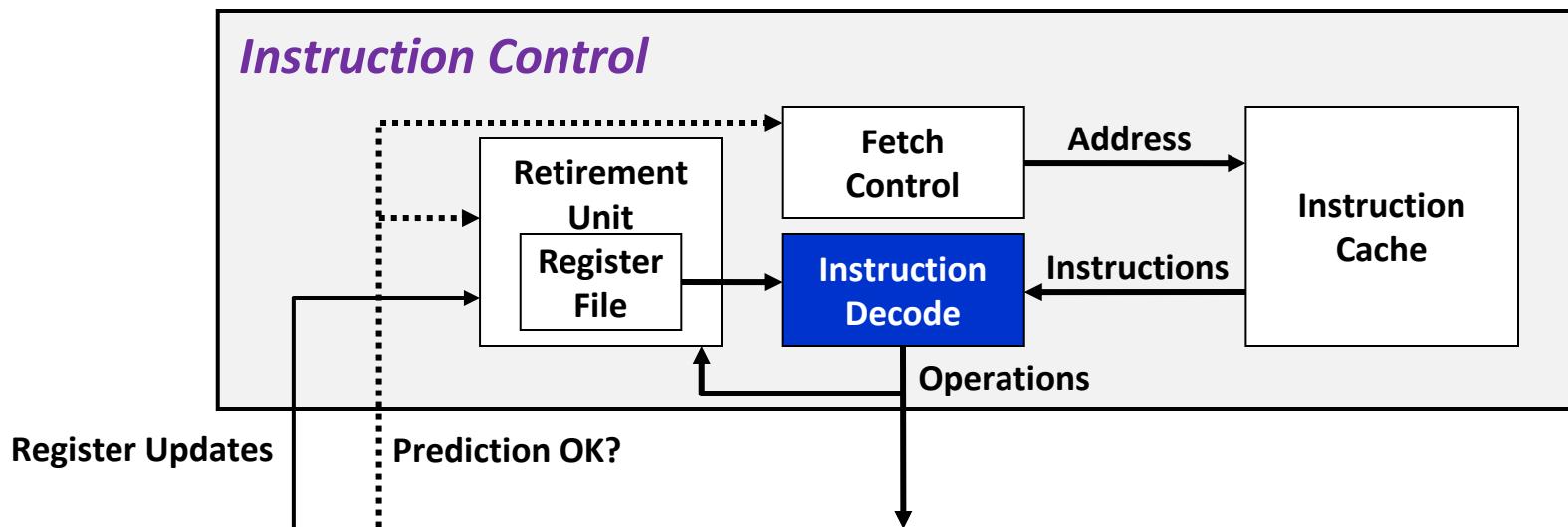
# Fetch Control

---

- Branch Predication
  - Branch taken or fall through
  - Guess whether branch is taken or not
- Speculative Execution
  - Fetch, decode and execute only according to the branch prediction
  - Before the branch predication has been determined whether or not

# Instruction Control Unit

- Instruction Decoding Logic
  - Take actual program instructions



# Instruction Control Unit

---

- Instruction Decoding Logic
  - Take actual program instructions
  - Converts them into a set of primitive operations
    - An instruction can be decoded into a variable number of operations
  - Each primitive operation performs some simple task
    - Simple arithmetic, Load, Store

addl %eax, 4(%edx)

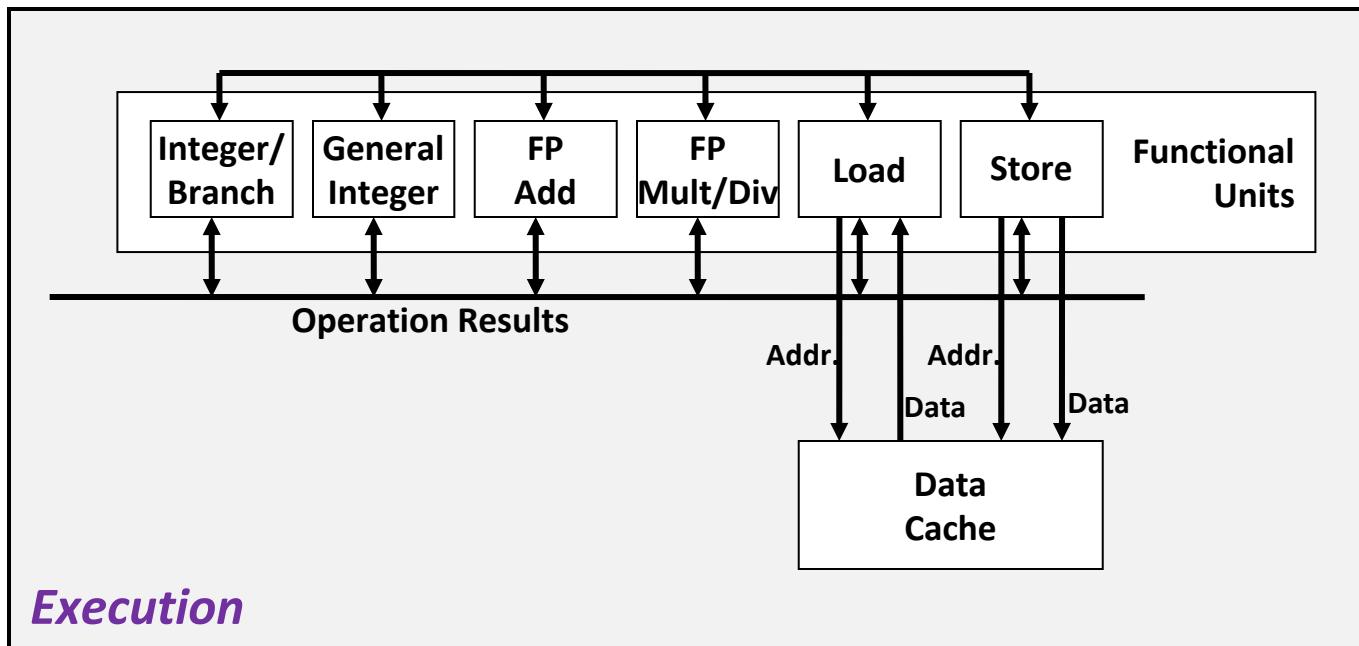
⇒

load 4(%edx) → t1  
addl %eax, t1 → t2  
store t2, 4(%edx)

- Register renaming

# Execution Unit

- Multi-functional Units
  - Receive operations from ICU
  - Execute a number of operations on each clock cycle
  - Handle specific types of operations



# Multi-functional Units

---

- Multiple Instructions Can Execute in Parallel
  - Nehalem CPU (Core i7)
    - 1 load, with address computation
    - 1 store, with address computation
    - 2 simple integer (one may be branch)
    - 1 complex integer (multiply/divide)
    - 1 FP Multiply
    - 1 FP Add

# Multi-functional Units

---

- Some Instructions Take > 1 Cycle, but Can be Pipelined

Nehalem (Core i7)

<u>Instruction</u>	<u>Latency</u>	<u>Cycles/Issue</u>
<b>Integer Add</b>	1	0.33
<b>Integer Multiply</b>	3	1
Integer/Long Divide	11--21	5--13
<b>Single/Double FP Add</b>	3	1
<b>Single/Double FP Multiply</b>	4/5	1
Single/Double FP Divide	10--23	6--19

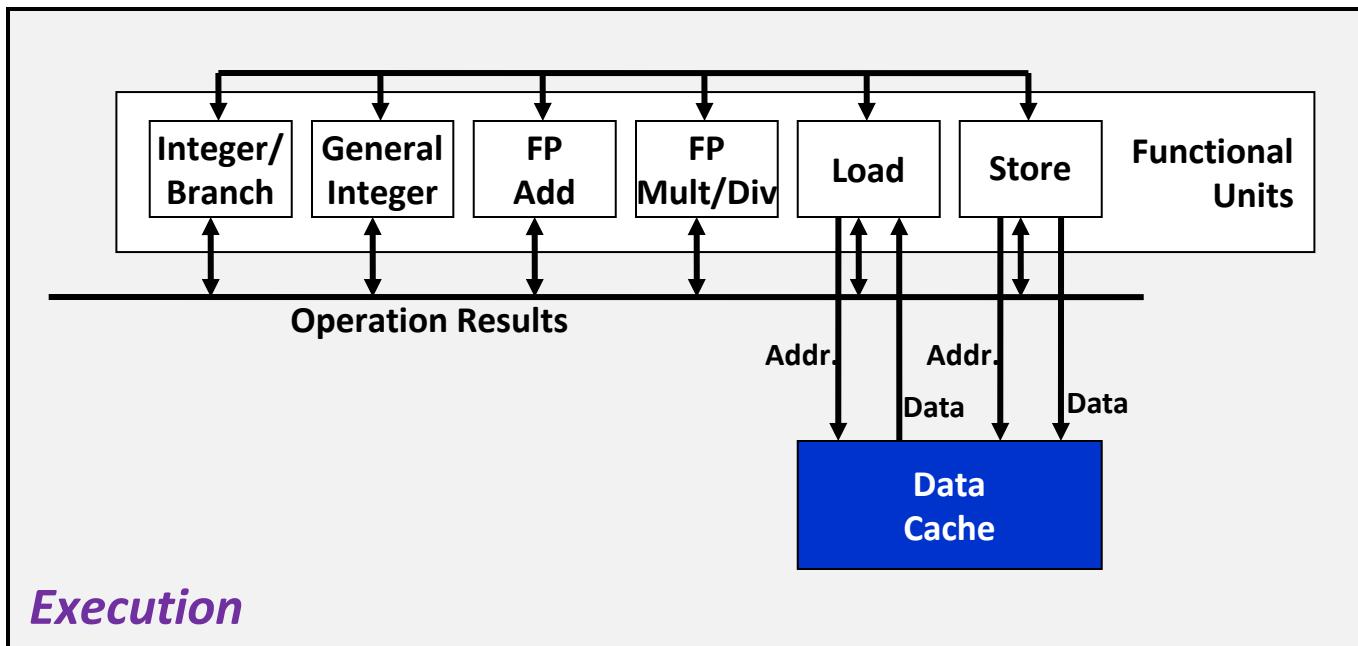
# Execution Unit

---

- Operation is dispatched to one of multi-functional units, whenever
  - All the operands of an operation are ready
  - Suitable functional units are available
- Execution results are passed among functional units

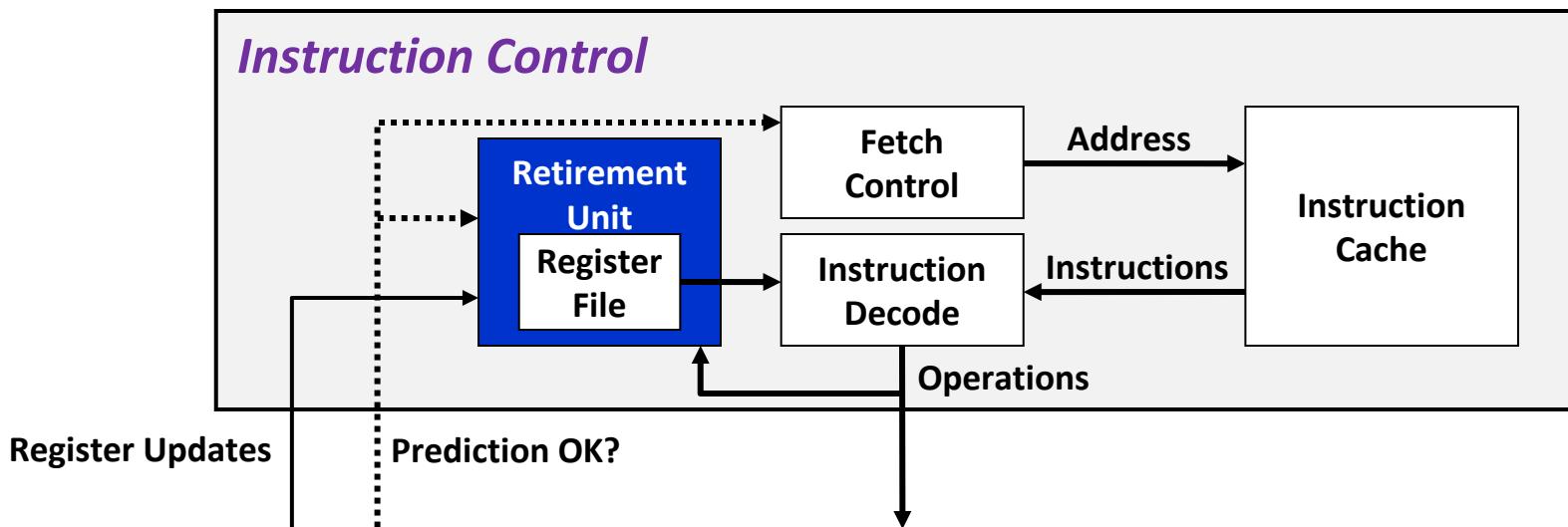
# Execution Unit

- Data Cache
  - Load and store units access memory via data cache
  - A high speed memory containing the most recently accessed data values



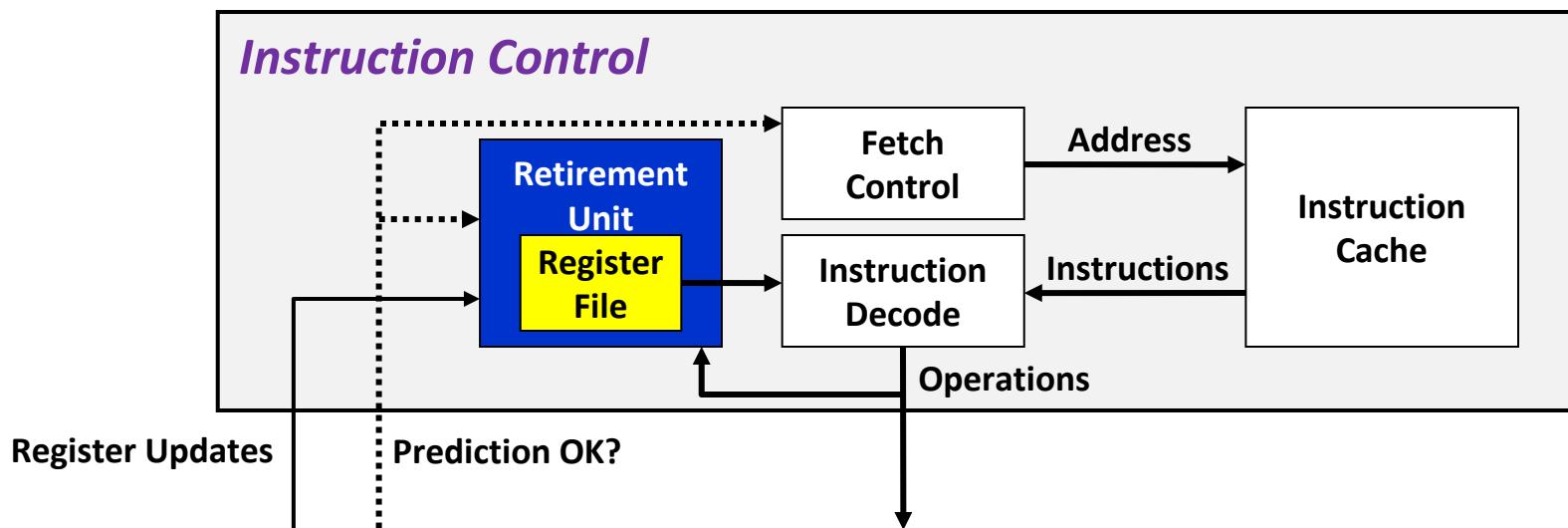
# Instruction Control Unit

- Retirement Unit
  - Keep track of the ongoing processing
  - Obey the sequential semantics of the machine-level program (misprediction & exception)



# Instruction Control Unit

- Register File
  - Integer, floating-point and other registers
  - Controlled by Retirement Unit



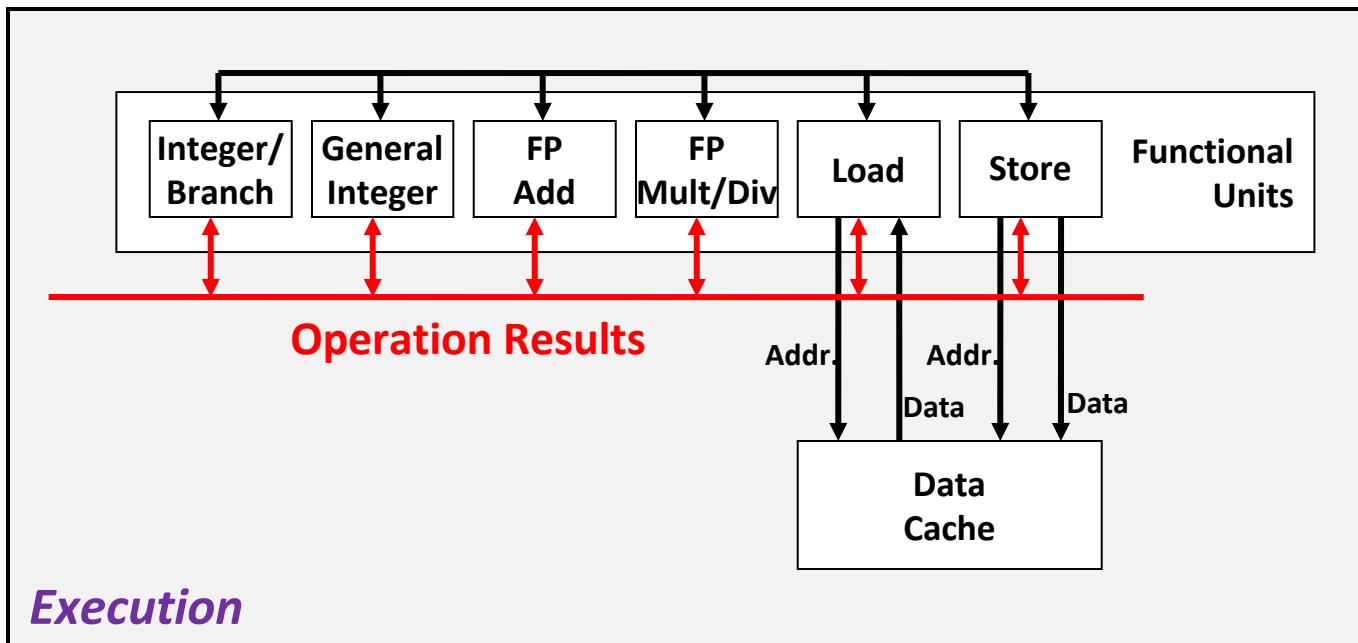
# Instruction Control Unit

---

- Instruction Retired/Flushed
  - Place instructions into a first-in, first-out queue
  - Retired: any updates to the registers being made
    - Operations of the instruction have completed
    - Any branch prediction to the instruction are confirmed correctly
  - Flushed: discard any results have been computed
    - Some branch prediction was mispredicted
    - Mispredictions can't alter the program state

# Execution Unit

- Operation Results
  - Functional units can send results directly to each other
  - An elaborate form of data forwarding techniques



# Execution Unit

---

- Register Renaming
  - Values passed directly from producer to consumers
  - A tag  $t$  is generated to the result of the operation
    - E.g. %ecx.0, %ecx.1
  - Renaming table
    - Maintain the association between program register  $r$  and tag  $t$  for an operation that will update this register

# Data-Flow Graphs

---

- Data-Flow Graphs
  - Visualize how the data dependencies in a program dictate its performance
  - Example: combine4 (`data_t` = float, `OP` = `*`)

```
void combine4(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t x = IDENT;

    for (i = 0; i < length; i++)
        x = x OP data[i];
    *dest = x;
}
```

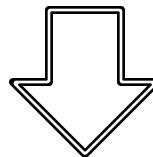
# Translation Example

.L488:

```
mulss (%rax,%rdx,4),%xmm0  
addq $1, %rdx  
cmpq %rdx,%rbp  
jg .L488
```

# Loop:

```
# t *= data[i]  
# Increment i  
# Compare length:i  
# if > goto Loop
```



.L488:

```
mulss (%rax,%rdx,4),%xmm0  
  
addq $1, %rdx  
cmpq %rdx,%rbp  
jg .L488
```

```
load (%rax,%rdx.0,4) → t.1  
mulq t.1, %xmm0.0      → %xmm0.1  
addq $1, %rdx.0         → %rdx.1  
cmpq %rdx.1, %rbp       → cc.1  
jg-taken cc.1
```

# Understanding Translation Example

---

```
mulss (%rax,%rdx,4),%xmm0
```

```
load (%rax,%rdx.0,4) → t.1  
mulq t.1, %xmm0.0 → %xmm0.1
```

- Split into two operations
  - Load reads from memory to generate temporary result t.1
  - Multiply operation just operates on registers

# Understanding Translation Example

---

```
mulss (%rax,%rdx,4),%xmm0
```

```
load (%rax,%rdx.0,4) → t.1  
mulq t.1, %xmm0.0 → %xmm0.1
```

- Operands

- Registers %rax does not change in loop
- Values will be retrieved from register file during decoding

# Understanding Translation Example

---

```
mulss (%rax,%rdx,4),%xmm0
```

```
load (%rax,%rdx.0,4) → t.1  
mulq t.1, %xmm0.0 → %xmm0.1
```

- Operands

- Register %xmm0 changes on every iteration
- Uniquely identify different versions as
  - %xmm0.0, %xmm0.1, %xmm0.2, ...
- Register renaming
  - Values passed directly from producer to consumers

# Understanding Translation Example

---

```
addq $1, %rdx
```

```
addq $1, %rdx.0
```

```
→ %rdx.1
```

- Register %rdx changes on each iteration
- Renamed as %rdx.0, %rdx.1, %rdx.2, ...

# Understanding Translation Example

---

```
cmpq %rdx,%rbp
```

```
cmpq %rdx.1, %rbp → cc.1
```

- Condition codes are treated similar to registers
- Assign tag to define connection between producer and consumer

# Understanding Translation Example

---

jg .L488

jg-taken cc.1

- Instruction control unit determines destination of jump
- Predicts whether target will be taken
- Starts fetching instruction at predicted destination

# Understanding Translation Example

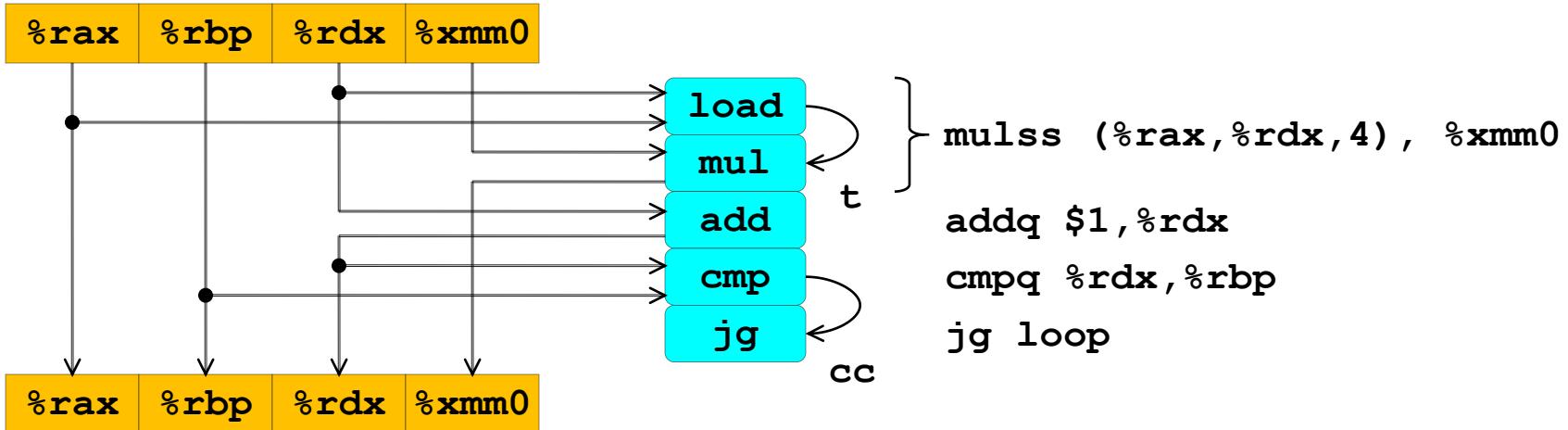
---

jg .L488

jg-taken cc.1

- Execution unit simply checks whether or not prediction was OK
- If not, it signals instruction control unit
  - Instruction control unit then “invalidates” any operations generated from misfetched instructions
  - Begins fetching and decoding instructions at correct target

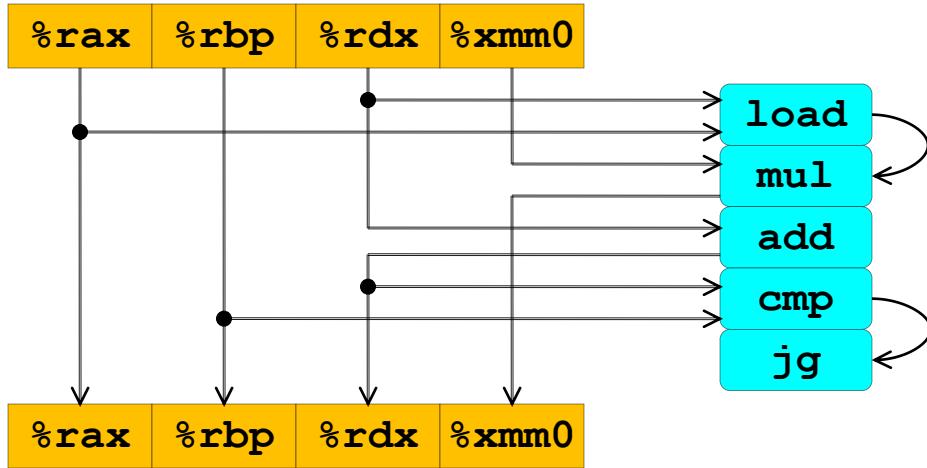
# Graphical Representation



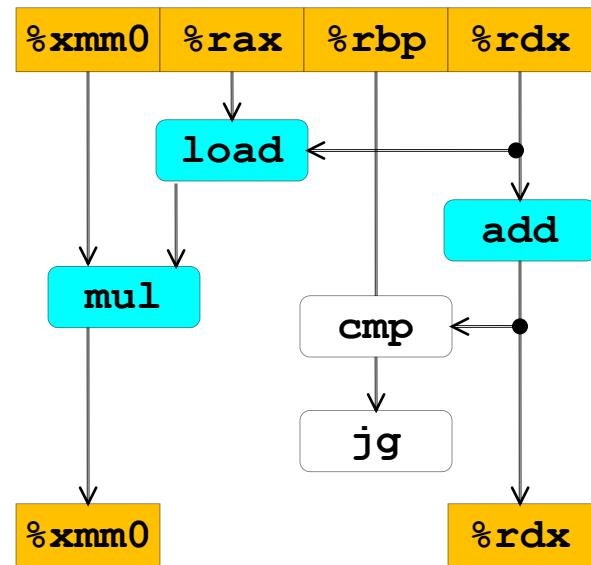
```
load (%rax,%rdx.0,4) → t.1
mulq t.1, %xmm0.0      → %xmm0.1
addq $1, %rdx.0         → %rdx.1
cmpq %rdx.1, %rbp       → cc.1
jg-taken cc.1
```

- Registers
  - read-only: %rax, %rcx
  - write-only: -
  - Loop: %rdx, %xmm0
  - Local: t, cc

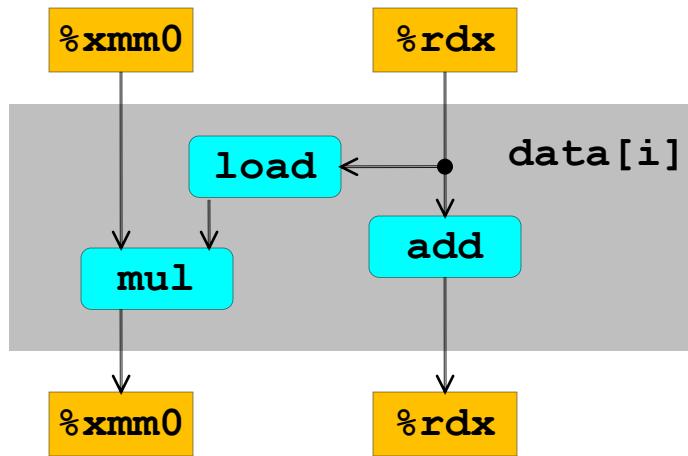
# Refinement of Graphical Representation



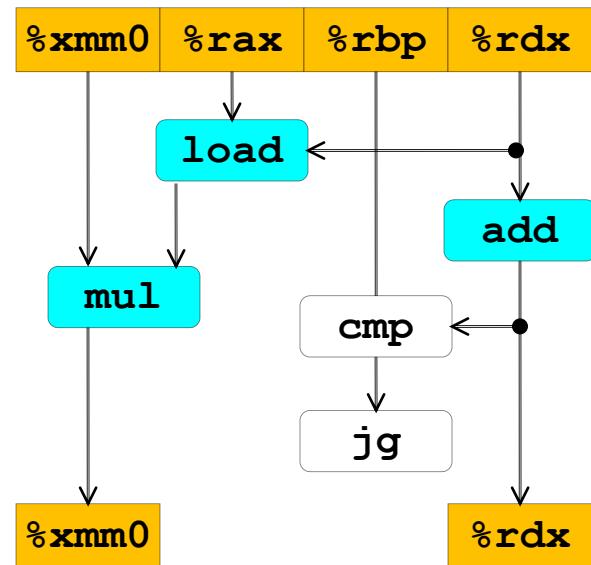
## Data Dependencies



# Refinement of Graphical Representation

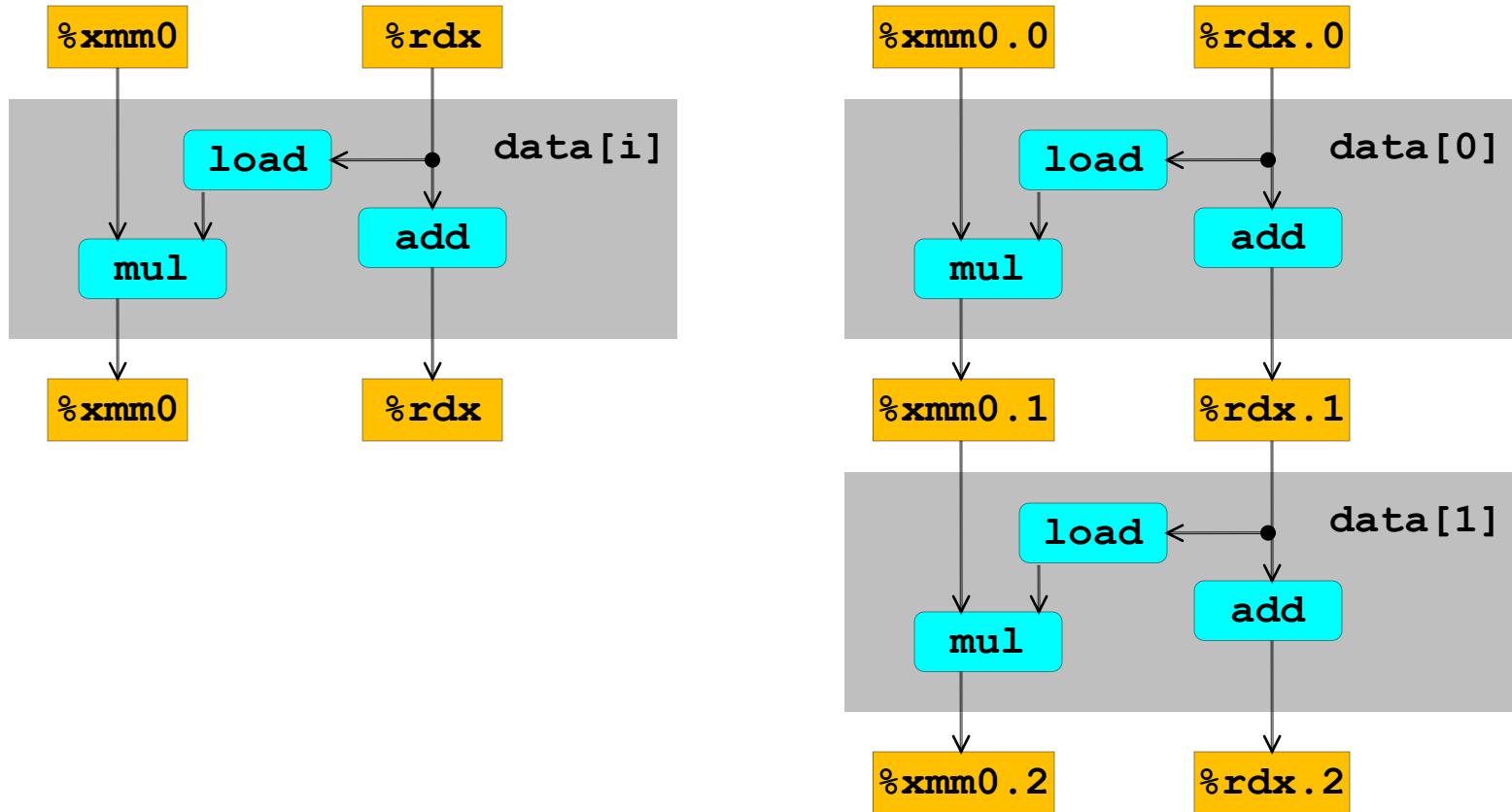


## Data Dependencies

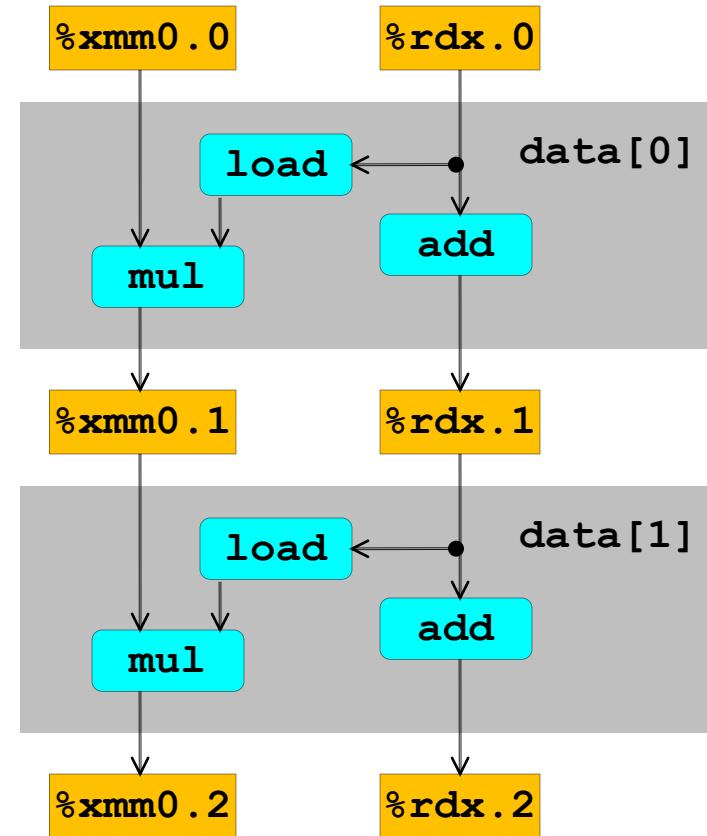
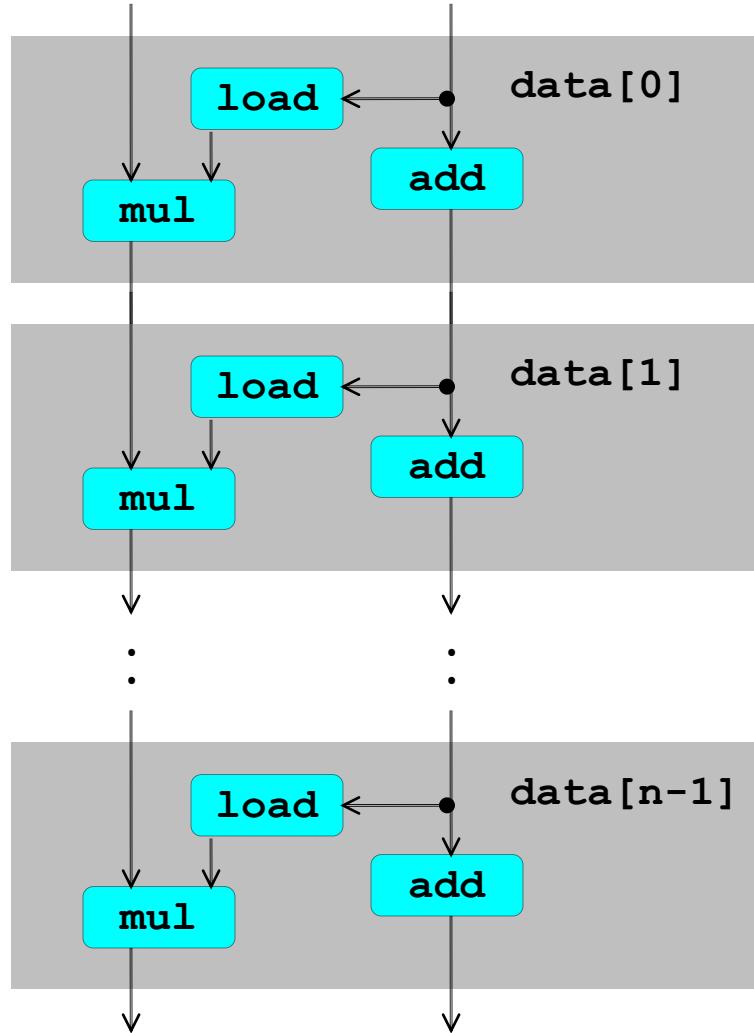


# Refinement of Graphical Representation

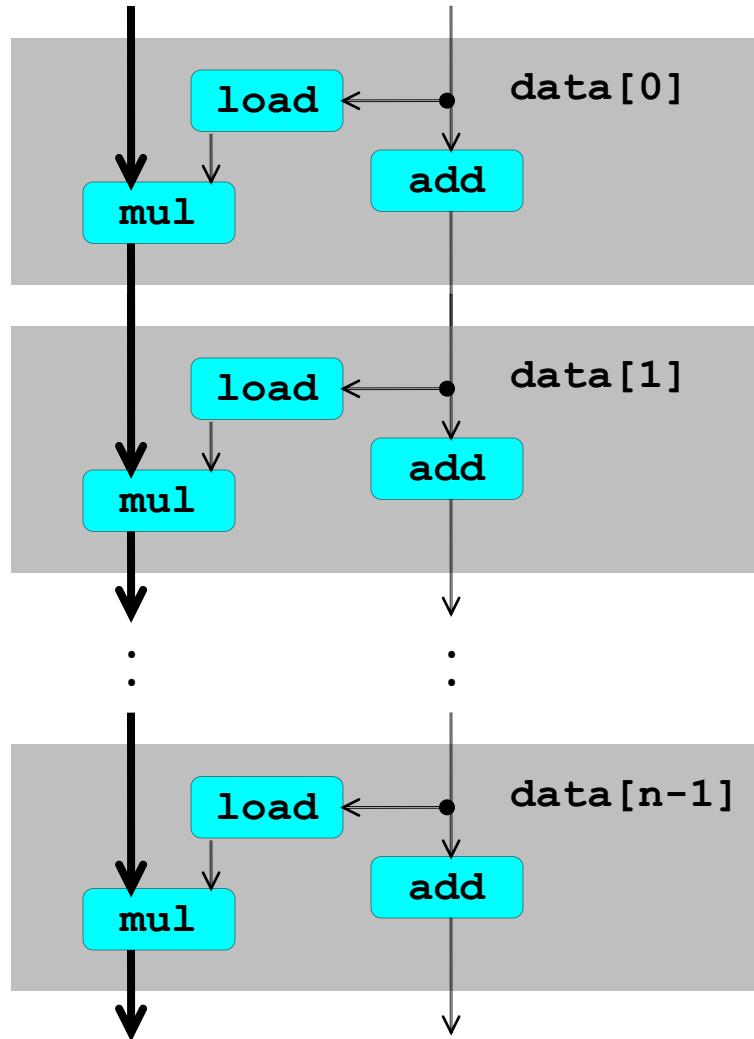
---



# Refinement of Graphical Representation



# Refinement of Graphical Representation



- Two chains of data dependencies
  - Update  $x$  by `mul`
  - Update  $i$  by `add`
- Critical path
  - Latency of `mul` is 4
  - Latency of `add` is 1

The latency of `combine4` is 4

Function	Integer		Floating Point		
	+	*	+	F*	D*
combine1	12	12	12	12	13
combine4	2	3	3	4	5

# Performance-limiting Critical Path

Function	Integer		Floating Point		
	+	*	+	F*	D*
combine1	12	12	12	12	13
combine4	2	3	3	4	5

Nehalem (Core i7)

<u>Instruction</u>	<u>Latency</u>	<u>Cycles/Issue</u>
Integer Add	<u>1</u>	0.33
Integer Multiply	3	1
Integer/Long Divide	11--21	5--13
Single/Double FP Add	3	1
Single/Double FP Multiply	4/5	1
Single/Double FP Divide	10--23	6--19

# Other Performance Factors

---

- Data-flow representation provide only a lower bound
  - e.g. Integer addition, CPE = 2.0
  - Total number of functional units available
  - The number of data values can be passed among functional units
- Next step
  - Enhance instruction-level parallelism
  - Goal: CPEs close to 1.0

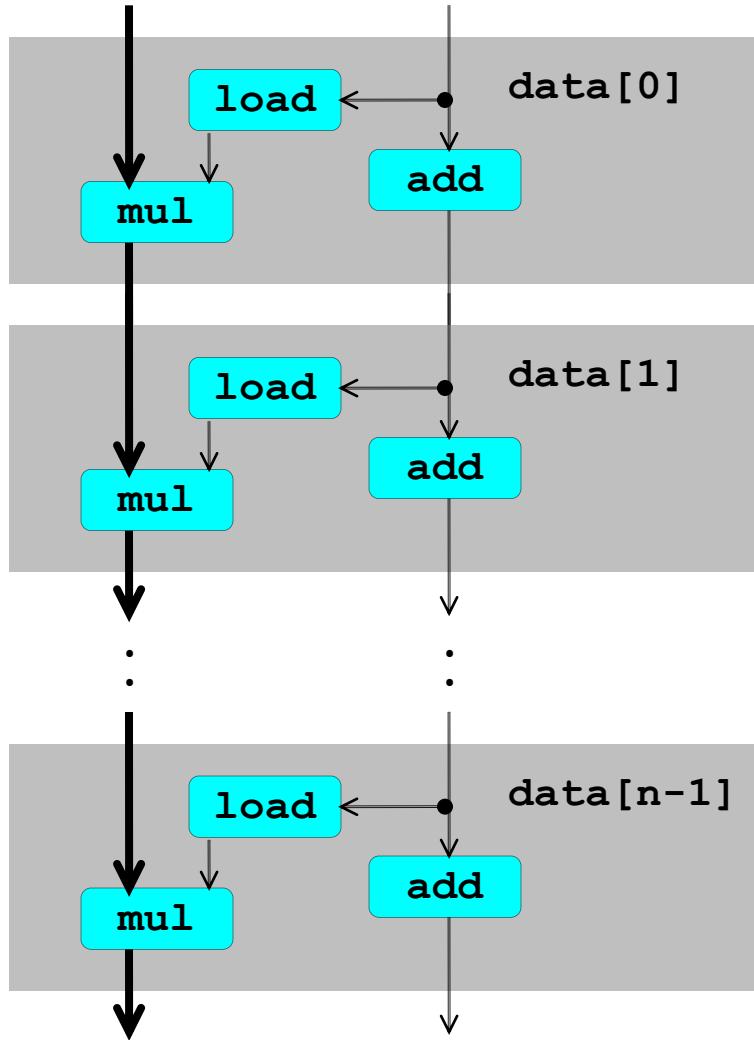
# More Code Optimization

# Outline

---

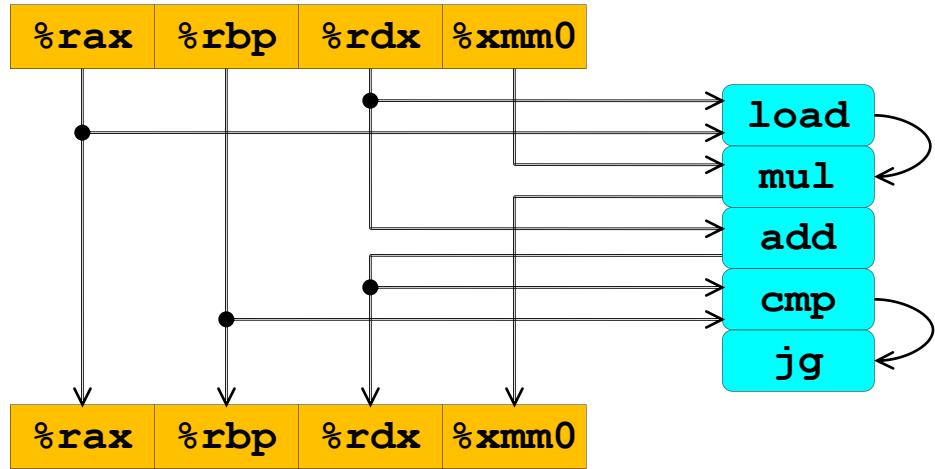
- More Code Optimization techniques
- Optimization Limiting Factors
- Memory Performance
- Suggested reading
  - 5.8 ~ 5.12

# Review



```
void combine4(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t x = IDENT;

    for (i = 0; i < length; i++)
        x = x OP data[i];
    *dest = x;
}
```



# Review

---

Function	Integer		Floating Point		
	+	*	+	F*	D*
combine1	12	12	12	12	13
combine4	2	3	3	4	5
Latency	1	3	3	4	5
Throughput	1	1	1	1	1

## Nehalem (Core i7)

<u>Instruction</u>	<u>Latency</u>	<u>Cycles/Issue</u>
Integer Add	1	0.33
Integer Multiply	3	1
Integer/Long Divide	11--21	5--13
Single/Double FP Add	3	1
Single/Double FP Multiply	4/5	1
Single/Double FP Divide	10--23	6--19

# Loop Unrolling

---

```
void combine5(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int limit = length - 2;
    int *data = get_vec_start(v);
    int x = IDENT;

    /* combine 3 elements at a time */
    for (i = 0; i < limit; i+=3)
        x = x OPER data[i] OPER data[i+1] OPER data[i+2];

    /* finish any remaining elements */
    for (; i < length; i++)
        x = x OPER data[i];

    *dest = x;
}
```

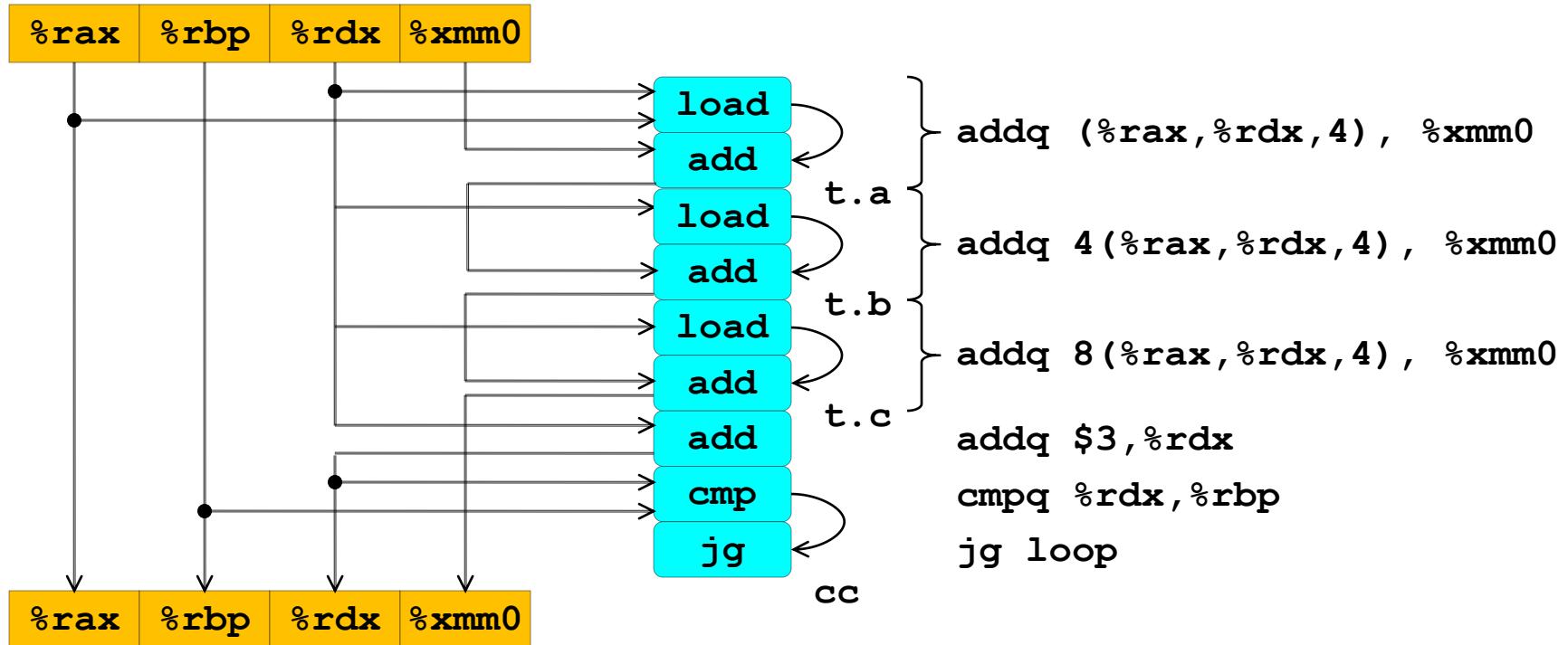
# Translation

---

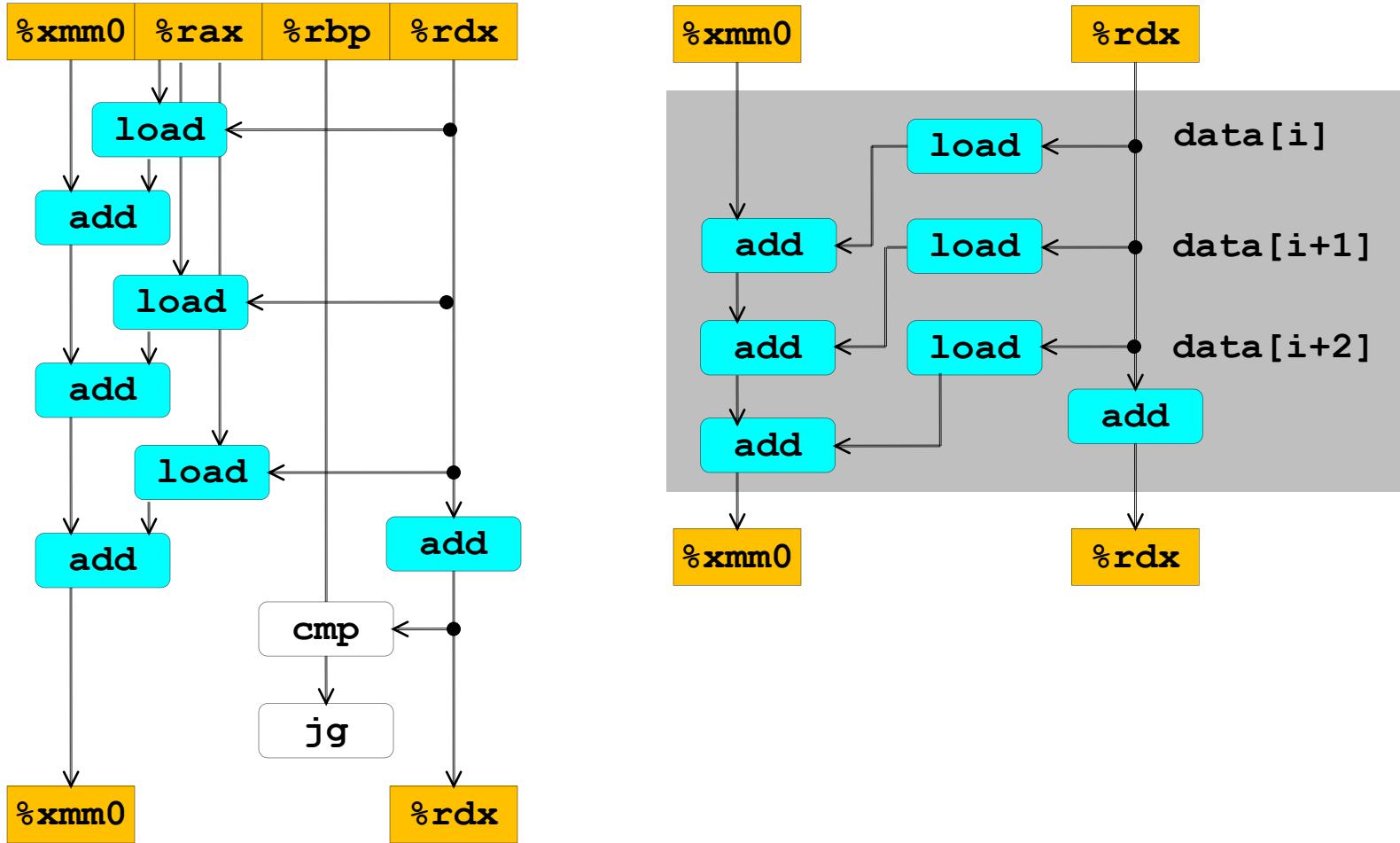
- Loads can pipeline, since don't have dependencies
- Only one set of loop control operations

load (%rax,%rdx.0,4)	→ t.1a
addq t.1a, %rcx.0c	→ %rcx.1a
load 4(%rax,%rdx.0,4)	→ t.1b
addq t.1b, %rcx.1a	→ %rcx.1b
load 8(%rax,%rdx.0,4)	→ t.1c
addq t.1c, %rcx.1b	→ %rcx.1c
addq \$3,%rdx.0	→ %rdx.1
cmpq %rdx.1, %rbp	→ cc.1
jg-taken cc.1	

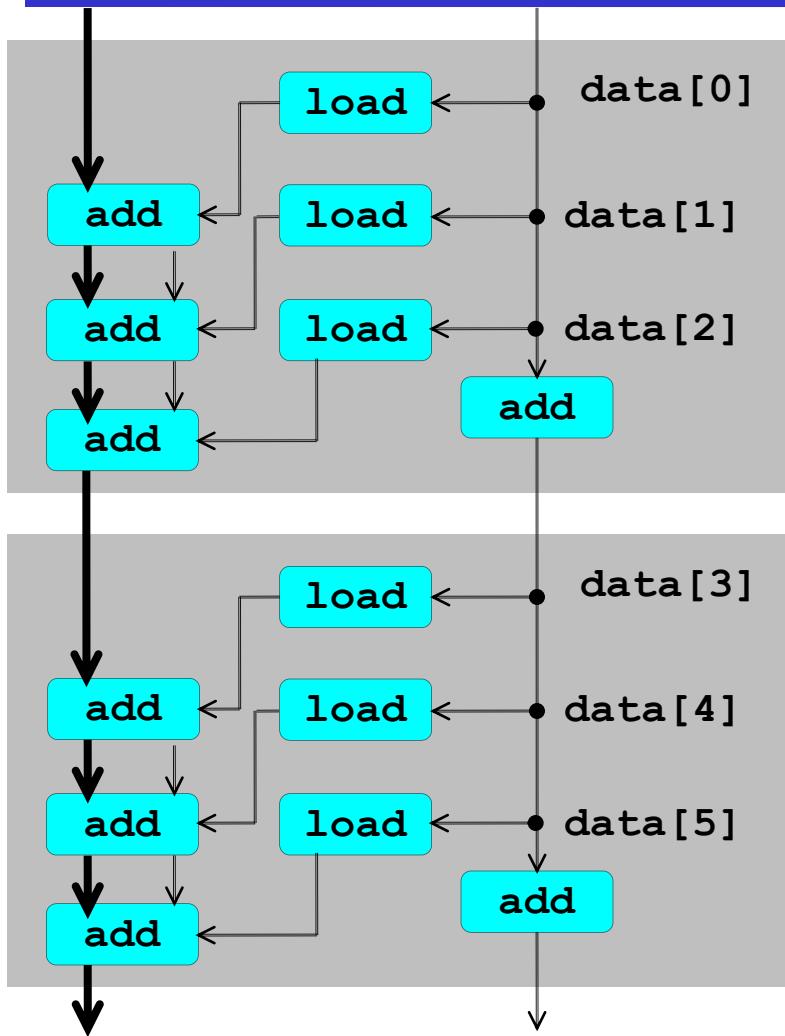
# Graphical Representation



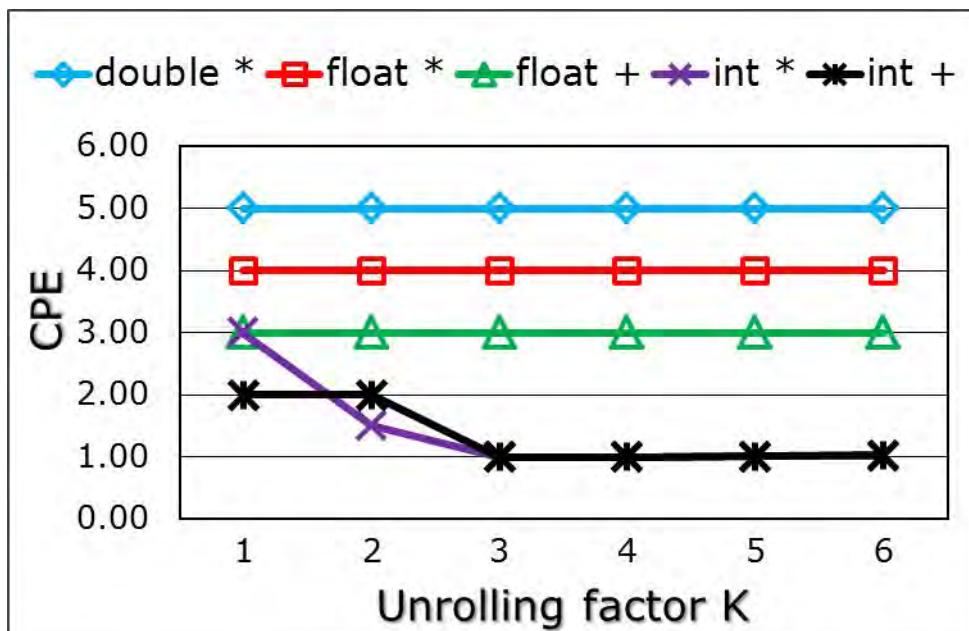
# Graphical Representation



# Graphical Representation



Function	Integer		Floating Point		
	+	*	+	F*	D*
combine4	2	3	3	4	5
Combine5*2	2	1.5	3	4	5
combine5*3	1	1	3	4	5
Latency	1	3	3	4	5
Throughput	1	1	1	1	1

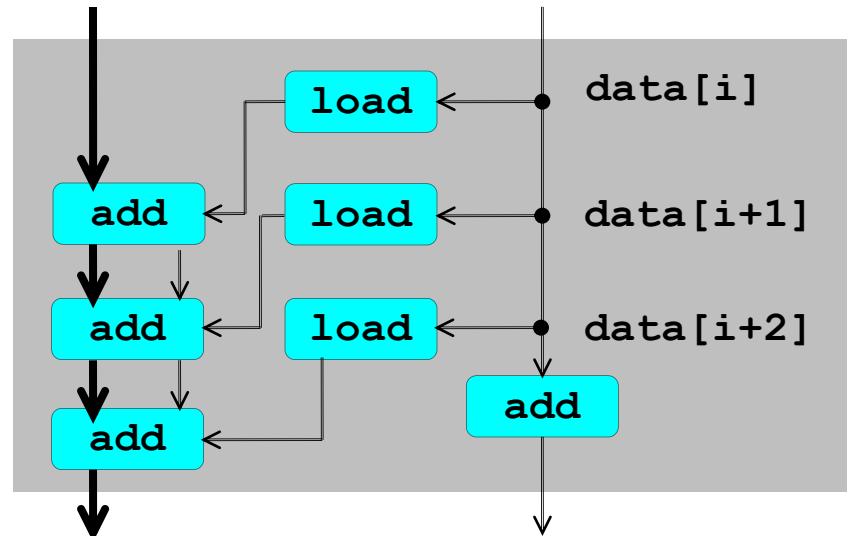
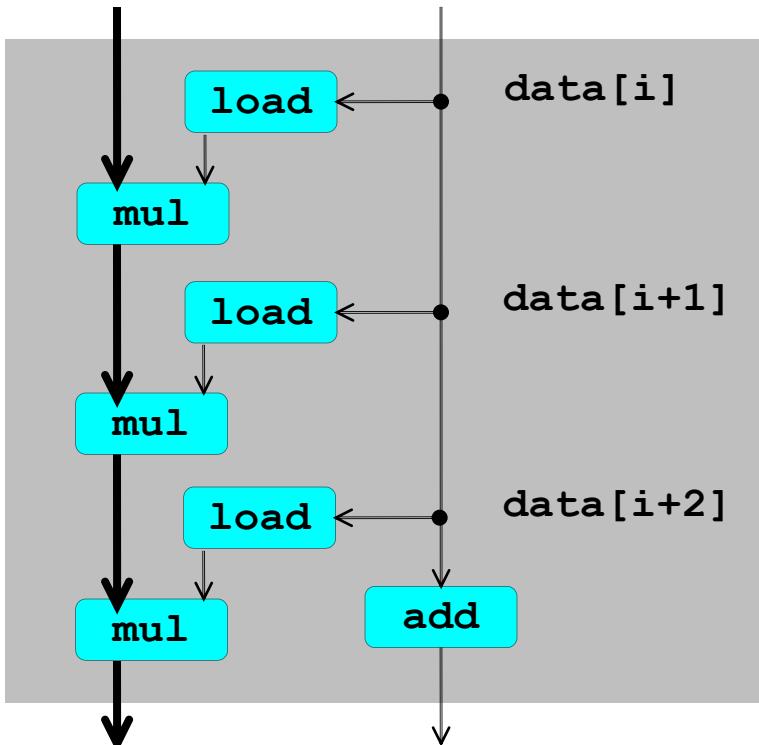


# Loop Unrolling

---

- Improve performance
  - Reduces the number of operations  
(e.g. loop indexing and conditional branching)
  - Transform the code to reduce the number of operations in the critical paths
- CPEs for both integer ops improve
- But for both floating-point ops do not

# Effect of Unrolling



- **Critical path**
  - Latency of integer add is 1
  - Latency of FP multiplex is 4

# Effect of Unrolling

---

- Only helps integer sum for our examples
  - Other cases constrained by functional unit latencies
- Effect is nonlinear with degree of unrolling
  - Many subtle effects determine exact scheduling of operations

# Enhance Parallelism

---

- Multiple Accumulators
  - Accumulate in two different sums
    - Can be performed simultaneously
  - Combine at end

# Multiple Accumulator

```
void combine6(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v), limit = length-1;
    int *data = get_vec_start(v);
    int x0 = IDENT, x1 = IDENT;

    /* combine 2 elements at a time */
    for (i = 0; i < limit; i+=2){
        x0 = x0 OPER data[i];
        x1 = x1 OPER data[i+1];
    }

    /* finish any remaining elements */
    for (; i < length; i++)
        x0 = x0 OPER data[i];

    *dest = x0 OPER x1;
}
```

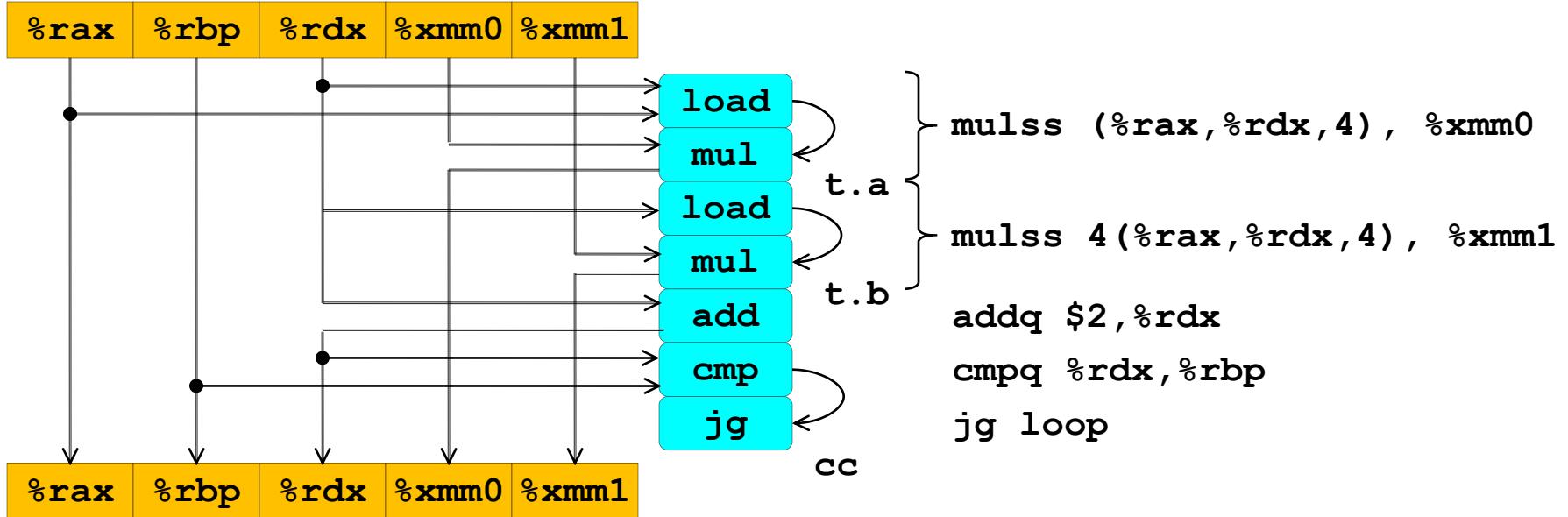
# Translation

---

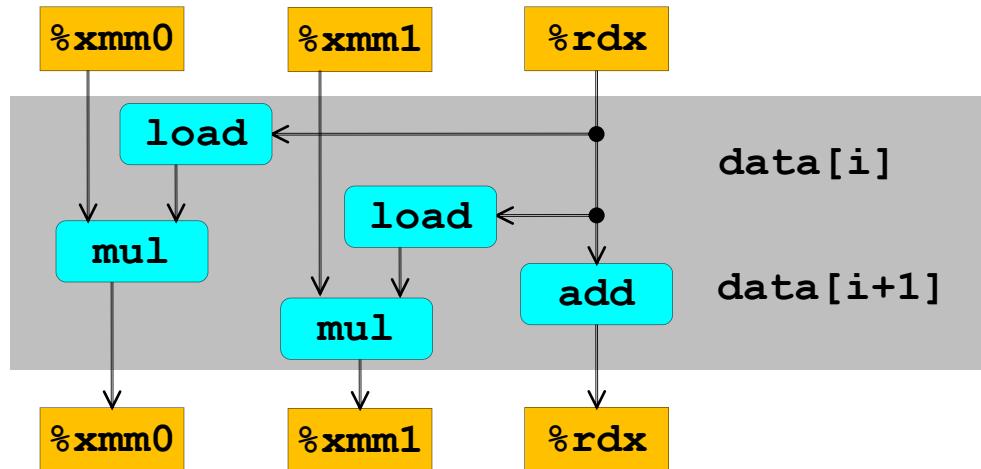
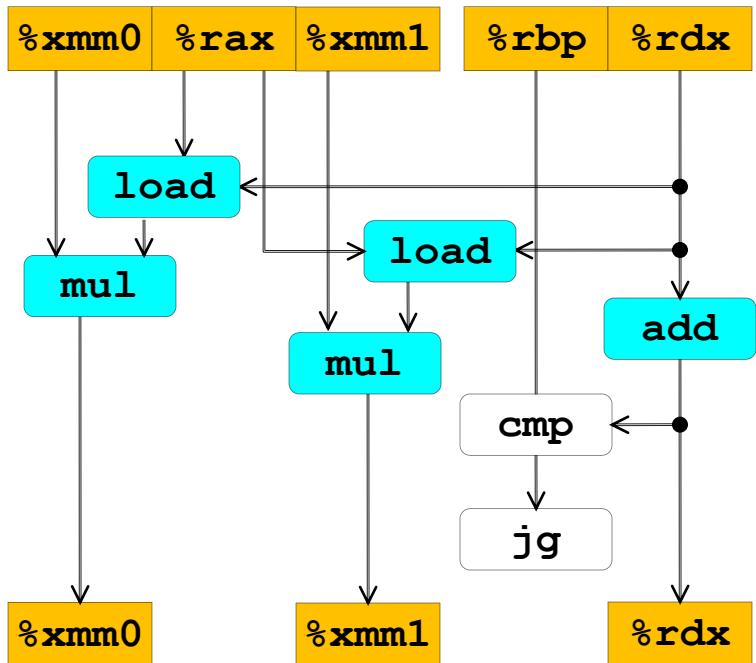
- Two multiplies within loop no longer have data dependency
- Allows them to pipeline

```
load (%rax,%rdx.0,4)    → t.1a
mulq t.1a, %xmm0.0       → %xmm0.1
load 4(%rax,%rdx.0,4)   → t.1b
mulq t.1b, %xmm1.0       → %xmm1.1
addq $2,%rdx.0           → %rdx.1
cmpq %rdx.1,%rbp         → cc.1
jg-taken cc.1
```

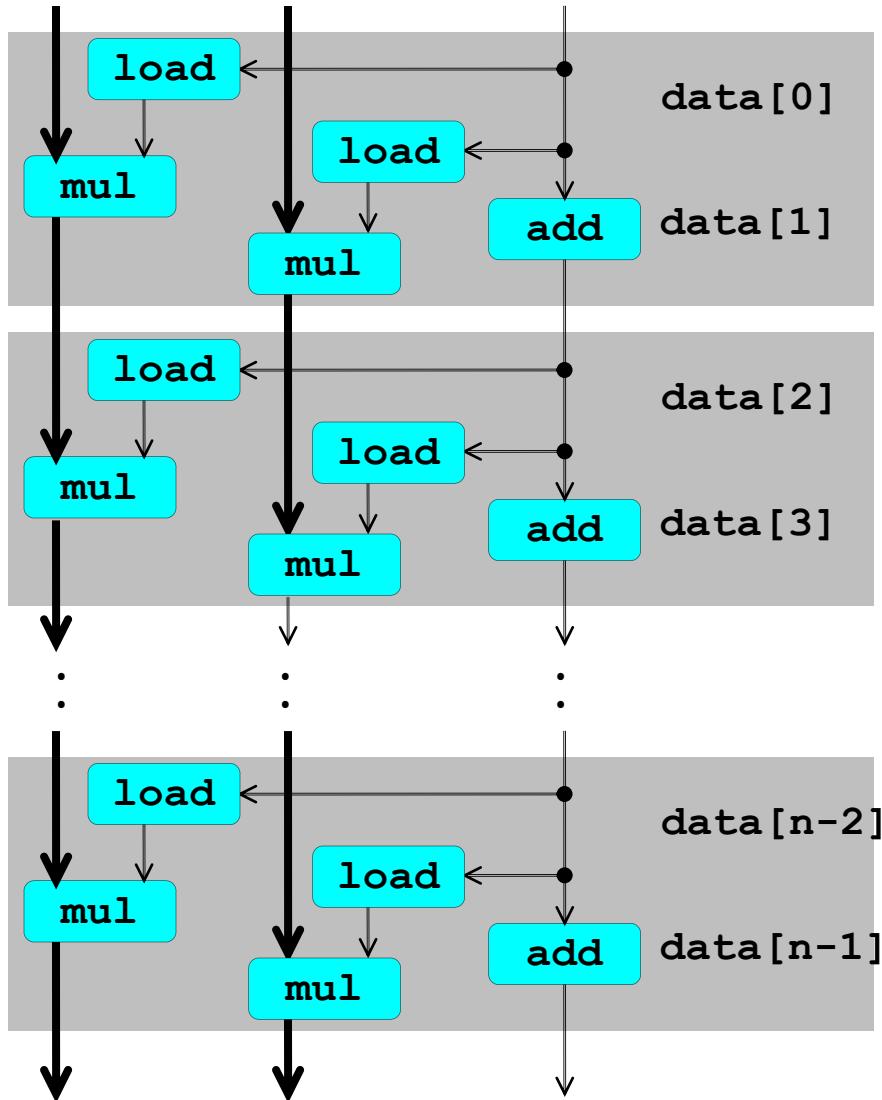
# Graphical Representation



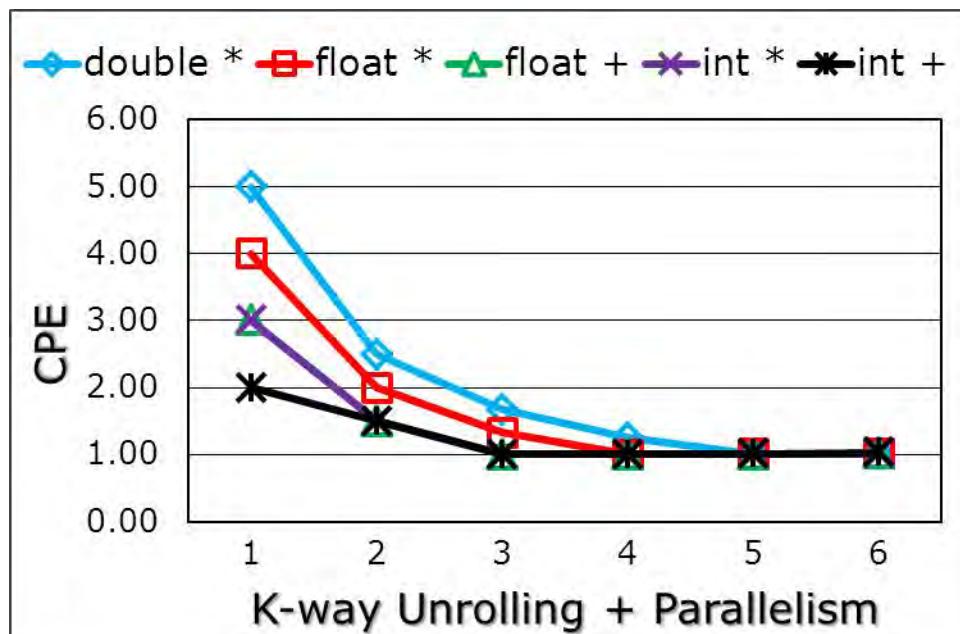
# Graphical Representation



# Graphical Representation



	Integer		Floating Point		
Function	+	*	+	F*	D*
combine4	2	3	3	4	5
Combine5*2	2	1.5	3	4	5
Combine6 (U*2, P*2)	1.5	1.5	1.5	2	2.5
Latency	1	3	3	4	5
Throughput	1	1	1	1	1



# Enhance Parallelism

---

- Multiple Accumulators
  - Accumulate in two different sums
    - Can be performed simultaneously
  - Combine at end
- Re-association Transformation
  - Exploits property that integer addition & multiplication are associative & commutative
  - FP addition & multiplication not associative, but transformation usually acceptable

# Re-association Transformation

```
void combine7(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v), limit = length-1;
    int *data = get_vec_start(v);
    int x = IDENT;

    /* combine 2 elements at a time */
    for (i = 0; i < limit; i+=2){
        x = x OPER (data[i] OPER data[i+1]);
    }

    /* finish any remaining elements */
    for (; i < length; i++)
        x0 = x0 OPER data[i];

    *dest = x0 OPER x1;
}
```

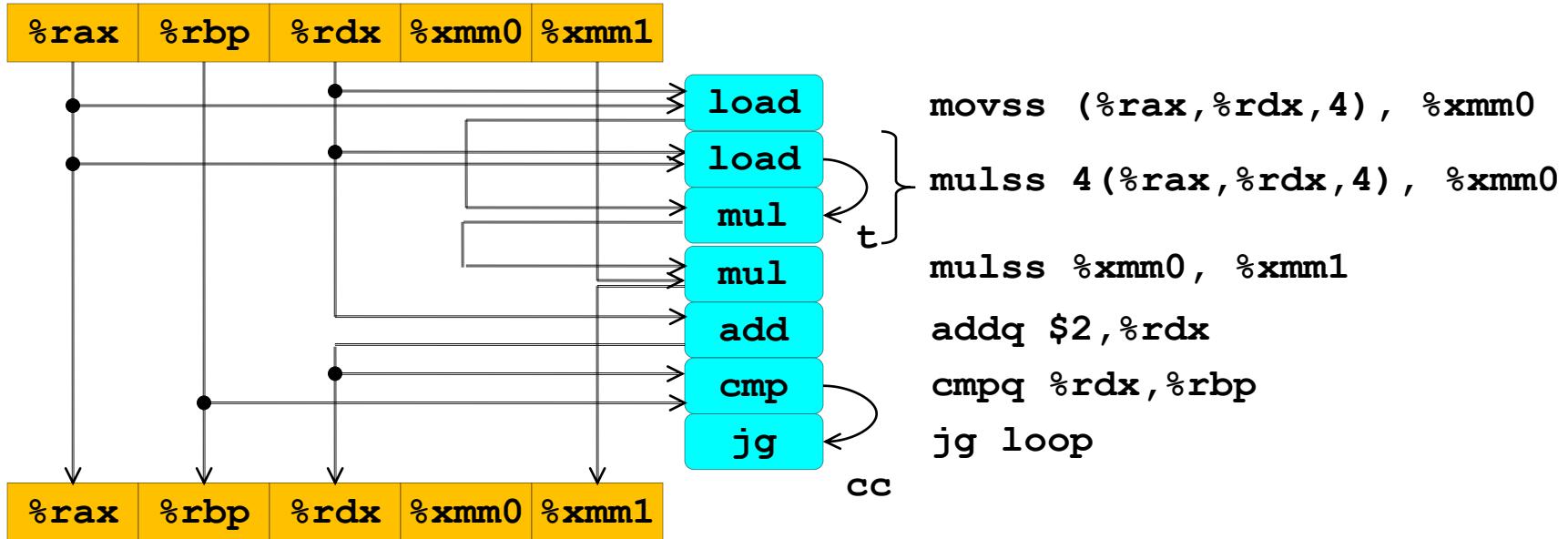
# Translation

---

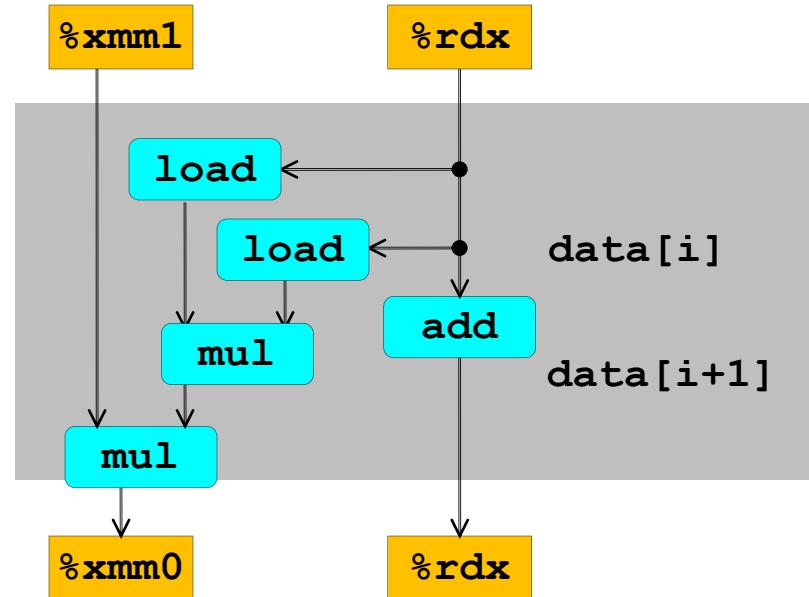
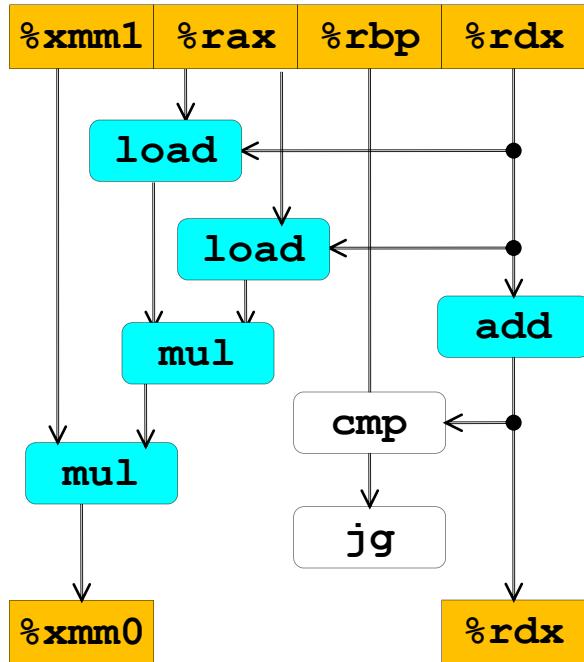
- Two multiplies within loop no longer have data dependency
- Allows them to pipeline

```
load (%rax,%rdx.0,4)    → xmm0.0
load 4(%rax,%rdx.0,4)   → t.1
mulq t.1, %xmm0.0        → xmm0.1
mulq %xmm0.1, %xmm1.0   → xmm1.1
addq $2,%rdx.0           → %rdx.1
cmpq %rdx.1,%rbp         → cc.1
jg-taken cc.1
```

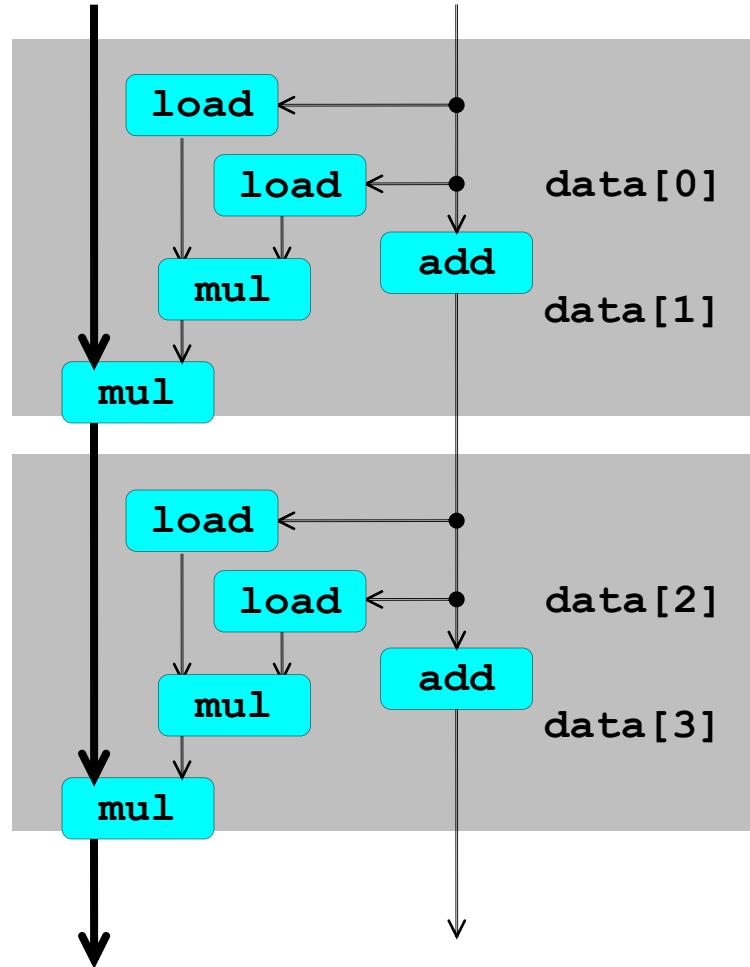
# Graphical Representation



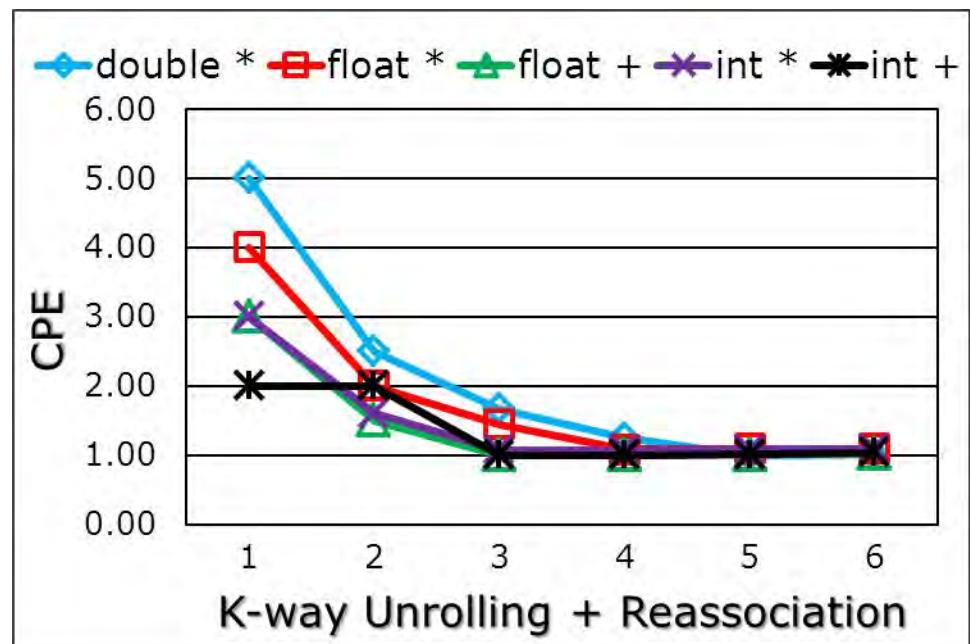
# Graphical Representation



# Graphical Representation



Function	Integer		Floating Point		
	+	*	+	F*	D*
combine6 (U*2,P*2)	1.5	1.5	1.5	2	2.5
combine7 (U*2,RT)	2	1.5	1.5	2	2.5
Latency	1	3	3	4	5
Throughput	1	1	1	1	1



# Summary of Results

---

- Optimization Results for Combining
  - Archive a CPE close to 1.0 for all combinations
  - Performance improvement of over 10X

## Machine Indep. Opt

- Eliminating loop inefficiencies
- Reducing procedure calls
- Eliminating unneeded memory references

## Machine dep. Opt

- Loop Unrolling
- Multiple Accumulator
- Reassociation

Function	Integer		Floating Point		
	+	*	+	F*	D*
combine1	12	12	12	12	13
combine6 (U*2,P*2)	1.5	1.5	1.5	2	2.5
combine6 (U*5,P*5)	1	1	1	1	1
Latency	1	3	3	4	5
Throughput	1	1	1	1	1

# Optimization Limiting Factors

---

- Register spilling
  - Only 6 registers available (IA32)
  - Using stack(memory) as storage

Machine	Degree of Unrolling					
	1	2	3	4	5	6
IA32	2.12	1.76	1.45	1.39	1.90	1.99
X86-64	2.00	1.50	1.00	1.00	1.01	1.00

# Example

**IA32 code.** Unroll **x5**, accumulate **x5**

**data\_t** = int, **OP** = +, **i** in **%edx**, **data** in **%eax**, limit at **%ebp-20**

.L291:

imull (%eax,%edx,4), %ecx  
**movl -16(%ebp), %ebx**  
imull 4(%eax,%edx,4), %ebx  
**movl %ebx, -16(%ebp)**  
imull 8(%eax,%edx,4), %edi  
imull 12(%eax,%edx,4), %esi  
**movl -28(%ebp), %ebx**  
imull 16(%eax,%edx,4), %ebx  
**movl %ebx, -28(%ebp)**  
addl \$5, %edx  
cmpl %edx, -20(%ebp)  
jg

loop:

x0 = x0 \* data[i]  
Get x1  
x1 = x1 \* data[i+1]  
Store x1  
x2 = x2 \* data[i+2]  
x3 = x3 \* data[i+3]  
Get x4  
x4 = x4 \* daa[i+4]  
Store x4  
i+= 5  
Compare limit:i  
If >, goto loop

# Optimization Limiting Factors

---

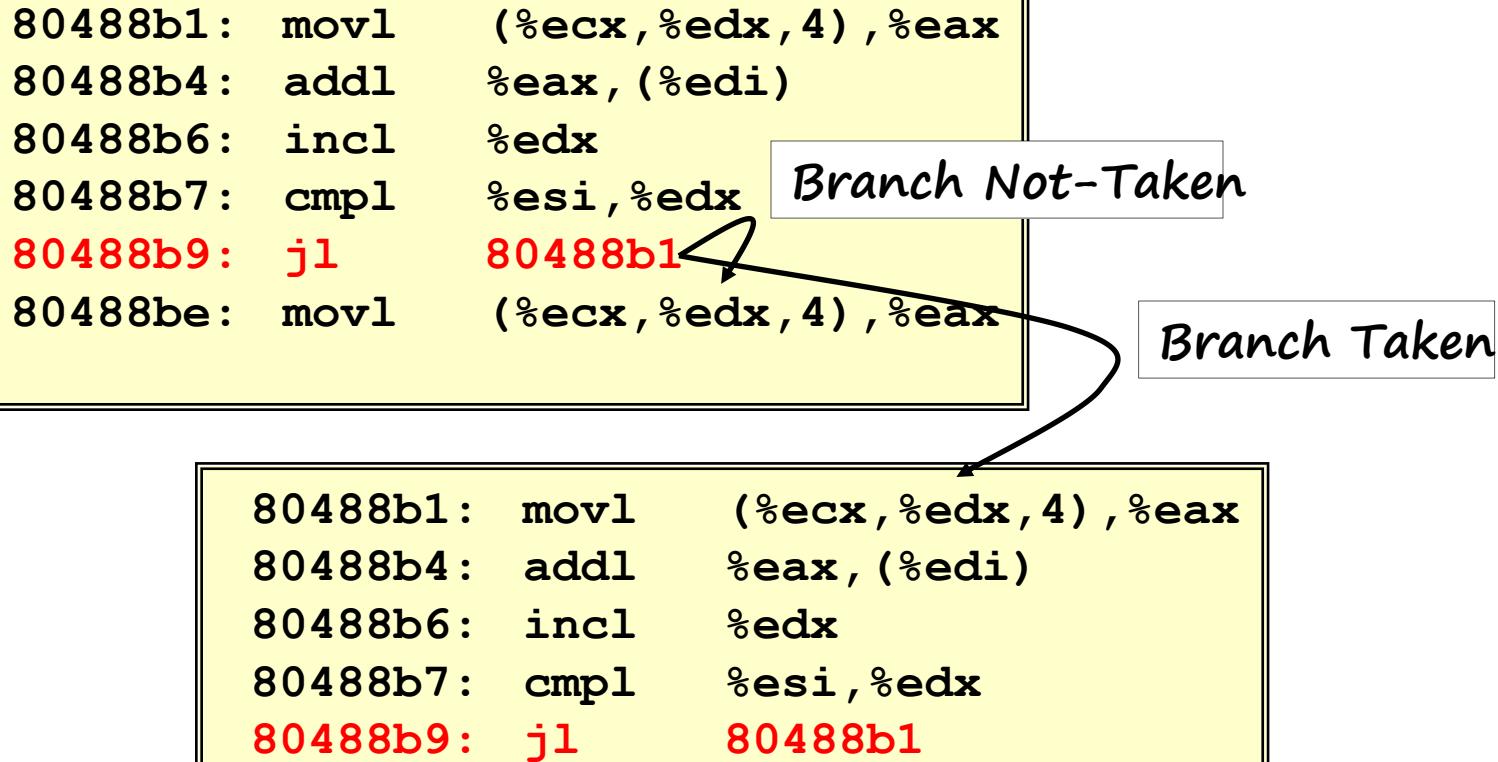
- Register Spilling
  - Only 6 registers available (IA32)
  - Using stack(memory) as storage
- Branch Prediction
  - Speculative Execution
  - Misprediction Penalties

# Branch Prediction

---

- Challenges
  - Instruction Control Unit must work well ahead of Exec. Unit
  - To generate enough operations to keep EU busy
- Speculative Execution
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don't actually modify register or memory data

# Example: Loop



# Branch Prediction Through Loop

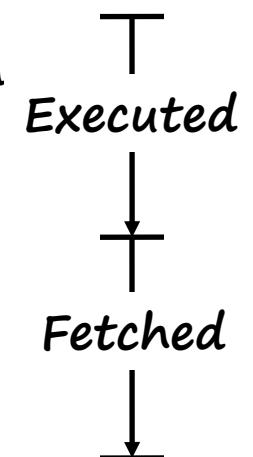
80488b1:	movl	(%ecx,%edx,4),%eax	
80488b4:	addl	%eax,(%edi)	
80488b6:	incl	%edx	
80488b7:	cmpl	%esi,%edx	$i = 98$
80488b9:	jl	80488b1	
80488b1:	movl	(%ecx,%edx,4),%eax	
80488b4:	addl	%eax,(%edi)	
80488b6:	incl	%edx	
80488b7:	cmpl	%esi,%edx	$i = 99$
80488b9:	jl	80488b1	
80488b1:	movl	(%ecx,%edx,4),%eax	
80488b4:	addl	%eax,(%edi)	
80488b6:	incl	%edx	
80488b7:	cmpl	%esi,%edx	$i = 100$
80488b9:	jl	80488b1	
80488b1:	movl	(%ecx,%edx,4),%eax	
80488b4:	addl	%eax,(%edi)	
80488b6:	incl	%edx	
80488b7:	cmpl	%esi,%edx	$i = 101$
80488b9:	jl	80488b1	

Assume vector length = 100

Predict Taken (OK)

Predict Taken  
(Oops)

Read  
invalid  
location



# Branch Misprediction Invalidation

80488b1:	movl	(%ecx,%edx,4),%eax
80488b4:	addl	%eax,(%edi)
80488b6:	incl	%edx
80488b7:	cmpl	%esi,%edx
80488b9:	jl	80488b1
		i = 98
80488b1:	movl	(%ecx,%edx,4),%eax
80488b4:	addl	%eax,(%edi)
80488b6:	incl	%edx
80488b7:	cmpl	%esi,%edx
80488b9:	jl	80488b1
		i = 99
80488b1:	movl	(%ecx,%edx,4),%eax
80488b4:	addl	%eax,(%edi)
80488b6:	incl	%edx
80488b7:	cmpl	%esi,%edx
80488b9:	jl	80488b1
		i = 100
80488b1:	movl	(%ecx,%edx,4),%eax
80488b4:	addl	%eax,(%edi)
80488b6:	incl	%edx
80488b7:	cmpl	%esi,%edx
80488b9:	jl	80488b1
		i = 101

Assume vector length = 100

Predict Taken (OK)

Predict Taken  
(Oops)

Invalidate

# Branch Misprediction Invalidation

80488b1:	movl	(%ecx,%edx,4),%eax
80488b4:	addl	%eax,(%edi)
80488b6:	incl	%edx
80488b7:	cmpl	%esi,%edx
80488b9:	jl	80488b1
		<i>i = 98</i>
80488b1:	movl	(%ecx,%edx,4),%eax
80488b4:	addl	%eax,(%edi)
80488b6:	incl	%edx
80488b7:	cmpl	%esi,%edx
80488b9:	jl	80488b1
80488be:	movl	(%ecx,%edx,4),%eax
		<i>i = 99</i>

*Assume vector length = 100*

Predict Taken (OK)

Definitely not taken

# Branch Misprediction Recovery

---

- Performance Cost
  - Misprediction on Core i7 wastes ~44 clock-cycle
  - Don't be overly concerned about predictable branches
    - E.g. loop-closing branches would typically be predicted as being taken, and only incur a misprediction penalty on the **last** time around

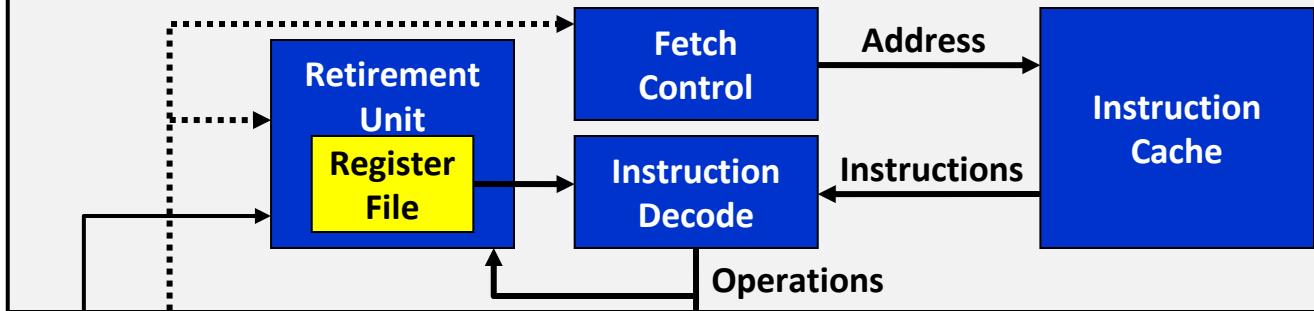
# Write Suitable Code

```
void minmax2(int a[],  
             int b[], int n)  
{  
    int i;  
    for (i = 0; i < n; i++) {  
        int min = a[i] < b[i] ? a[i] : b[i];  
        int max = a[i] < b[i] ? b[i] : a[i];  
        a[i] = min;  
        b[i] = max;  
    }  
}
```

```
void minmax1(int a[],  
             int b[], int n)  
{  
    int i;  
    for (i = 0; i < n; i++) {  
        if (a[i] > b[i]) {  
            int t = a[i];  
            a[i] = b[i];  
            b[i] = t;  
        }  
    }  
}
```

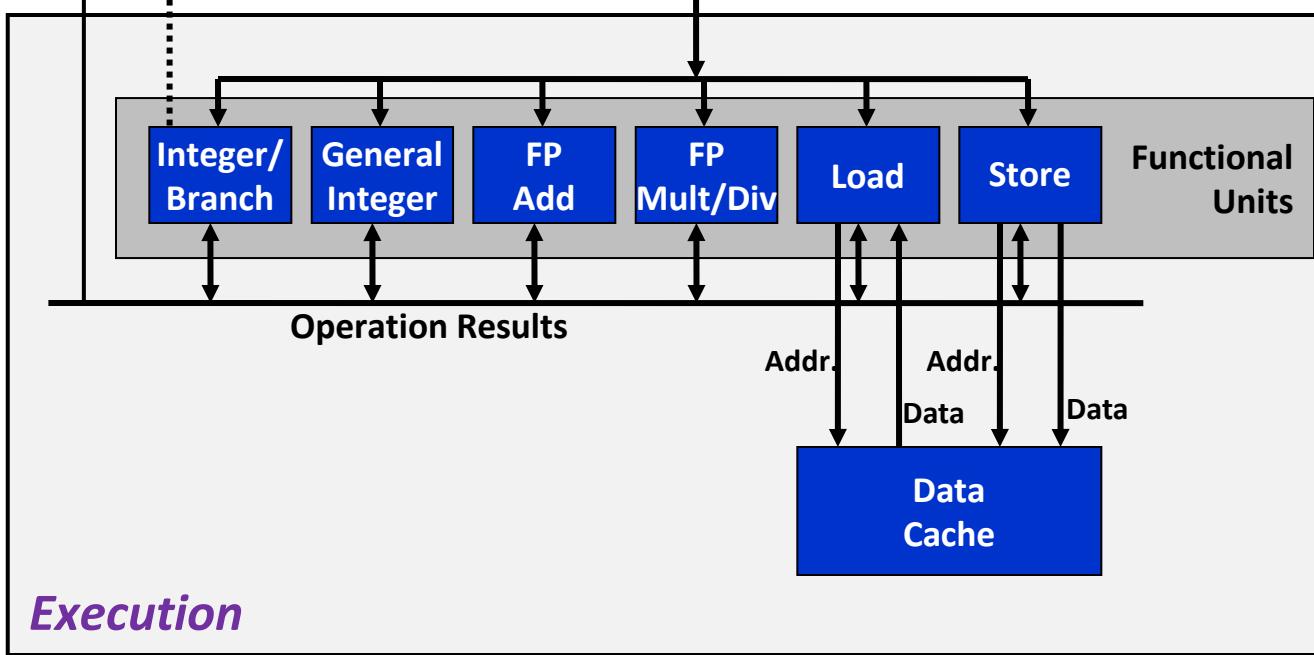
Function	random	predictable
minmax1	14.5	3.0-4.0
minmax2	5.0	5.0

## *Instruction Control*



Register Updates

Prediction OK?



# Load Performance

- load unit can only initiate one load operation every clock cycle (Issue=1.0)

```
typedef struct ELE {  
    struct ELE *next ;  
    int data ;  
} list_ele, *list_ptr ;  
  
int list_len(list_ptr ls) {  
    int len = 0 ;  
    while (ls) {  
        len++ ;  
        ls = ls->next;  
    }  
    return len ;  
}
```

```
len in %eax, ls in %rdi  
.L11:  
    addl $1, %eax  
    movq (%rdi), %rdi  
    testq %rdi, %rdi  
    jne .L11
```

Function	CPE
list_len	4.0
load latency	4.0

# Store Performance

---

- store unit can only initiate one store operation every clock cycle (Issue=1.0)

```
void array_clear_4(int *dest, int
    n)
{
    int i;
    int limit = n-3;
    for (i = 0; i < limit; i+=4) {
        dest[i] = 0;
        dest[i+1] = 0;
        dest[i+2] = 0;
        dest[i+3] = 0;
    }
    for ( ; i < n; i++)
        dest[i] = 0;
}
```

Function	CPE
array_clear_4	1.0

# Store Performance

```
void write_read(int *src,
                int *dest, int n)

{
    int cnt = n;
    int val = 0;

    while (cnt--) {
        *dest = val;
        val = (*src)+1;
    }
}
```

Function	CPE
Example A	2.0
Example B	6.0

**Example A:**  
`write_read(&a[0], &a[1], 3)`

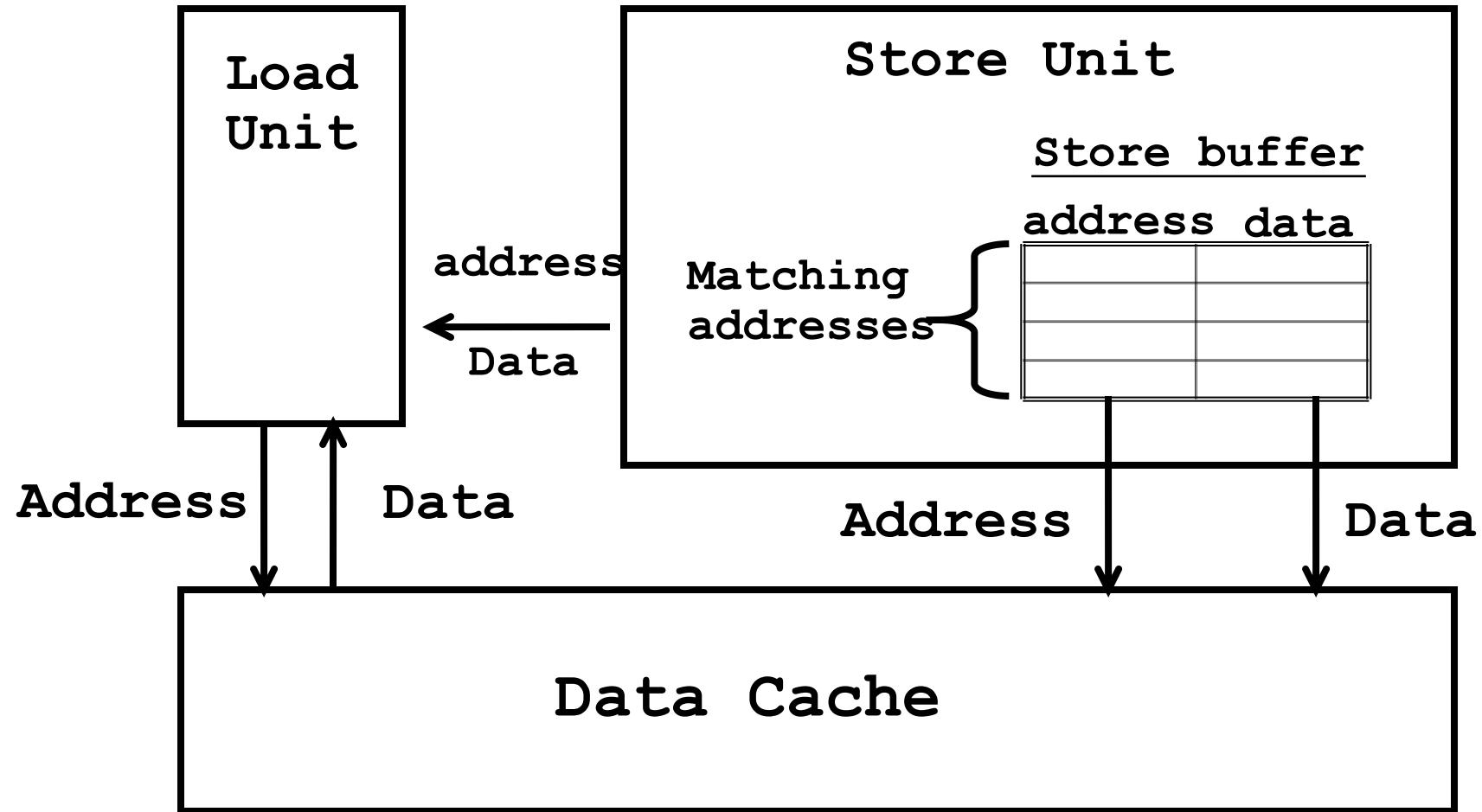
	initial		initial		initial		initial	
cnt	3		2		1		0	
a	-10	17	-10	0	-10	-9	-10	-9
val	0		-9		-9		-9	

**Example B:**  
`write_read(&a[0], &a[0], 3)`

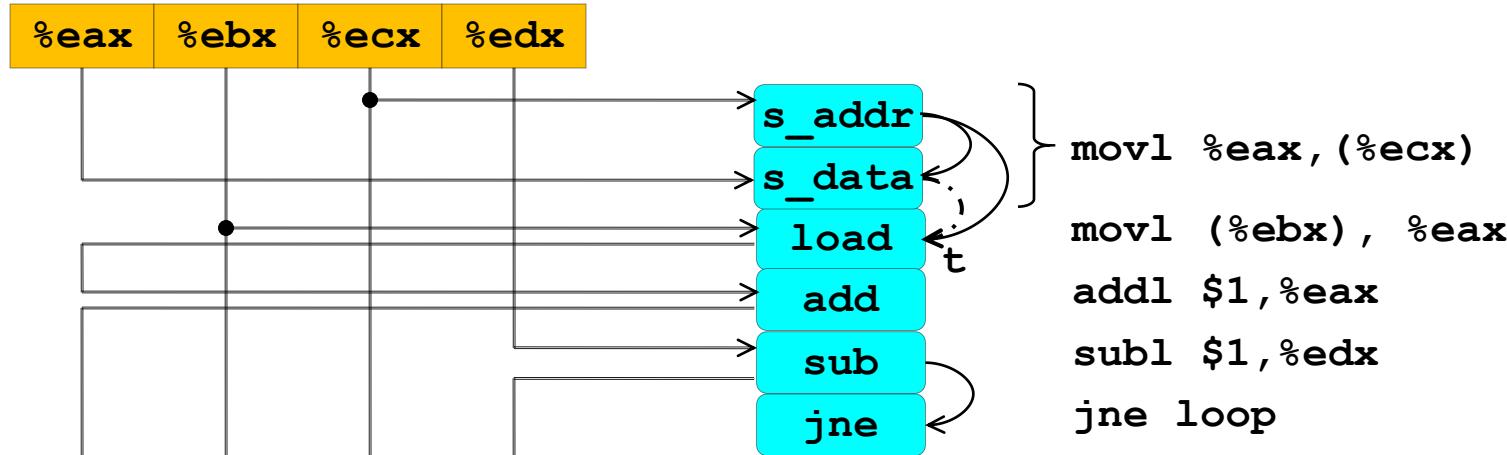
	initial		initial		initial		initial	
cnt	3		2		1		0	
a	-10	17	0	17	1	17	2	17
val	0		1		2		3	

# Load and Store Units

---



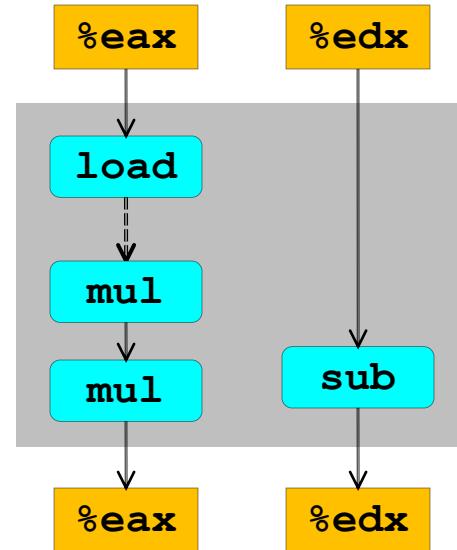
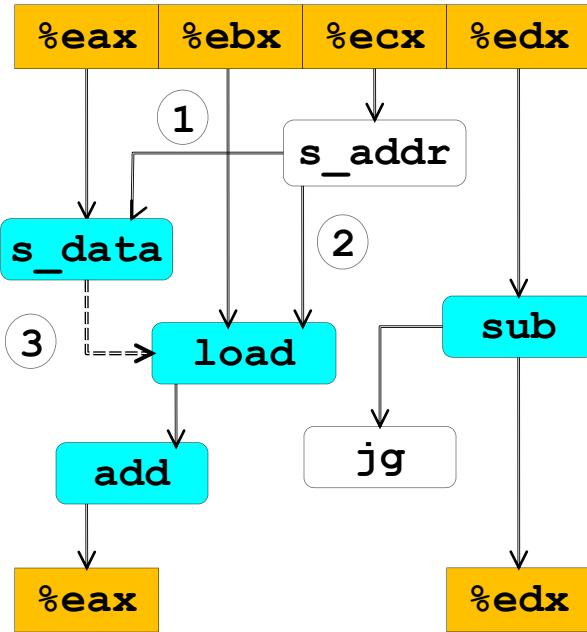
# Graphical Representation



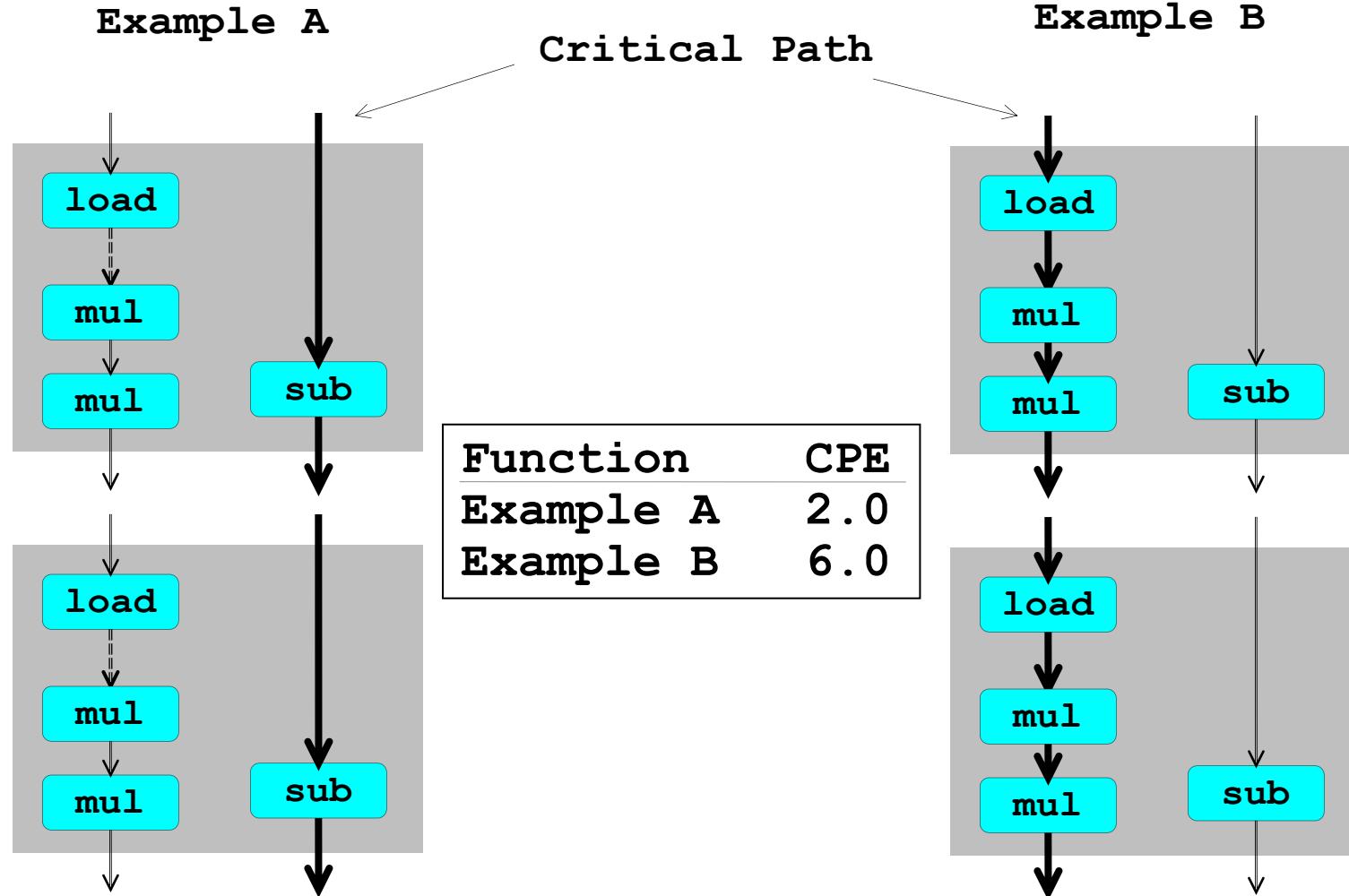
```
//inner-loop
while (cnt--) {
    *dest = val;
    val = (*src)+1;
}
```

# Graphical Representation

---



# Graphical Representation



# Getting High Performance

---

- Good compiler and flags
- Don't do anything stupid
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:  
procedure calls & memory references
    - Look carefully at innermost loops
- Tune code for machine
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly

# Exceptional Control Flow I

# Outline

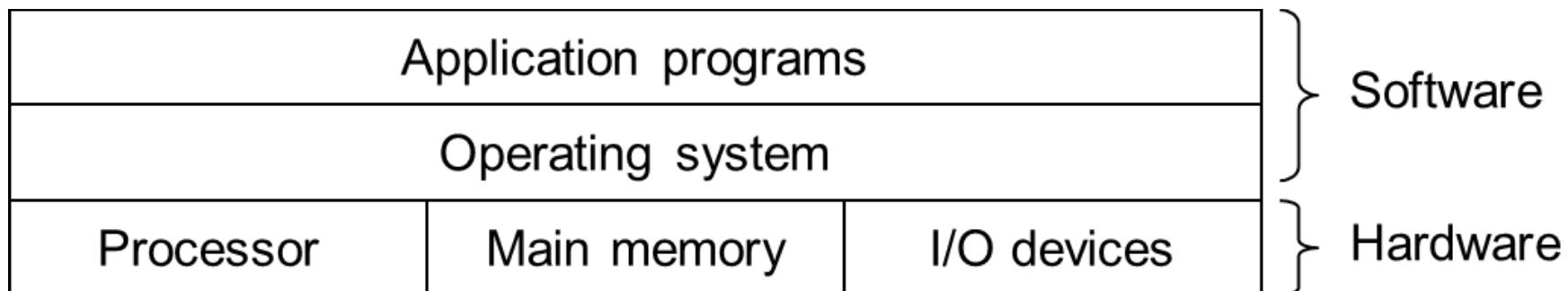
---

- Operating Systems
- Processes
- Virtual Memory
- Suggested reading 1.7, 8.2.1, 8.2.2, 8.2.3

# Operating Systems

---

- Contemporary Applications never access hardware resources directly
- They rely on the services provided by the operating system



# Operating System

---

- Is a program that manages the computer hardware
- Also provides a basis for application programs
- And acts as an intermediary between a user of a computer and the computer hardware

# Two primary purposes

---

- Protect hardware from misuse by applications
- Provide applications with simple and uniform mechanisms for manipulating hardware devices

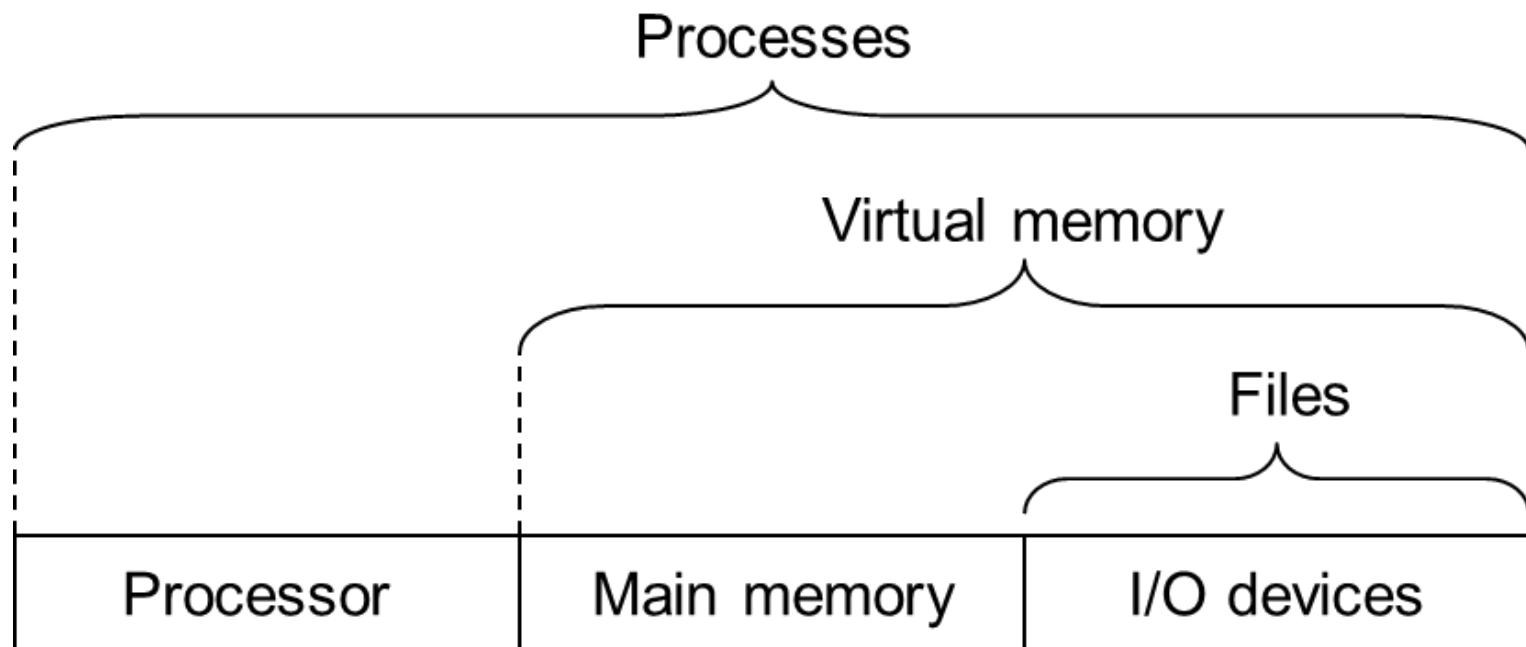
# Operating Systems

---

- Fundamental abstractions:
  - Process
  - Virtual memory
  - files

# Abstractions Provided by OS

---



# Process: phenomenon

---

- When a program runs on a modern system,
  - the operating system provides the illusion
    - that the program is the only one running on the system

# Process: phenomenon

---

- The program appears to have exclusive use of all the processor, main memory, and I/O devices
- The program appears to execute the instructions in the program, one after the other, without interruption
- The code and data of the program appear to be the only objects in the system's memory

# Processes

---

- Def: A *process* is an instance of a running program.
- Process is one of the most profound ideas in computer science.
- Each program in the system runs in the context of some process

# When a New Process is Created

---

- Each time a user runs a program
  - by typing the name of an executable object file to the shell,
- the shell creates a new process
- and then runs the executable object file
  - in the context of this new process

# When a New Process is Created

---

- Application programs can also
  - create new processes
  - and run
    - either their own
    - or other application
  - in the context of new process

# Processes

---

- Process provides each program with two key abstractions:
  - Logical control flow
    - gives each program the illusion that it has exclusive use of the CPU.
  - Private address space
    - gives each program the illusion that has exclusive use of main memory

# Processes

---

- How is this illusion maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

# Process

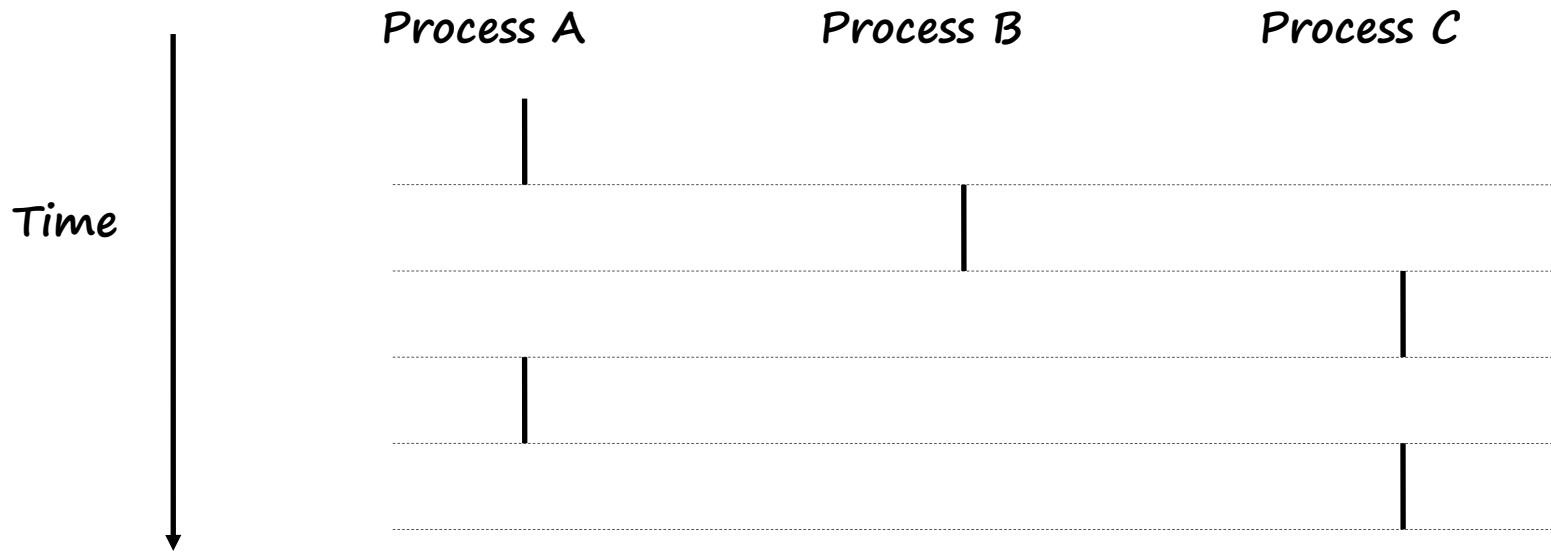
---

- Concurrency
  - Multiple processes can run concurrently
  - The instructions of one process are interleaved with the instructions of another process

# Logical control flows

---

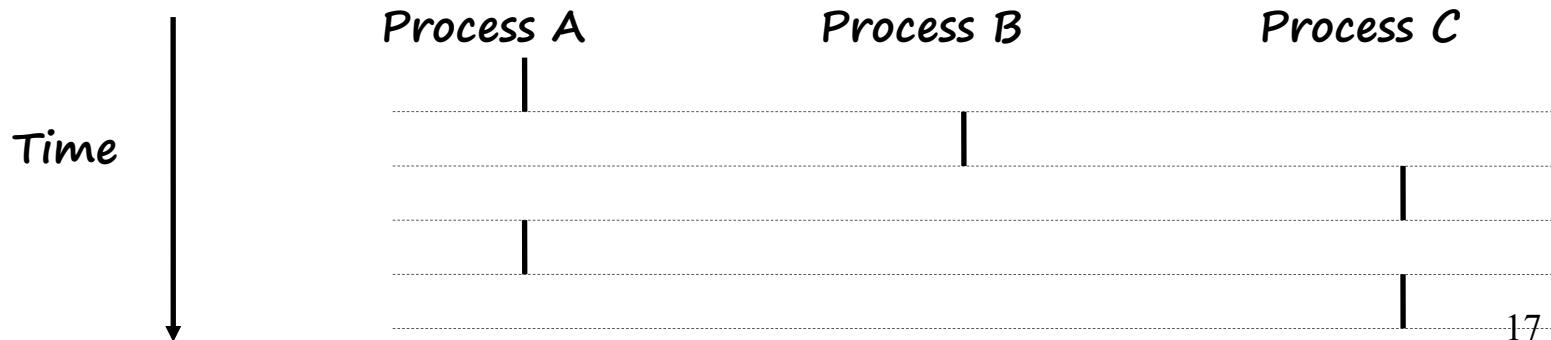
- Each process has its own logical control flow
  - does not affect the states of any other processes
- Preempted
- Time slice



# Concurrent processes

---

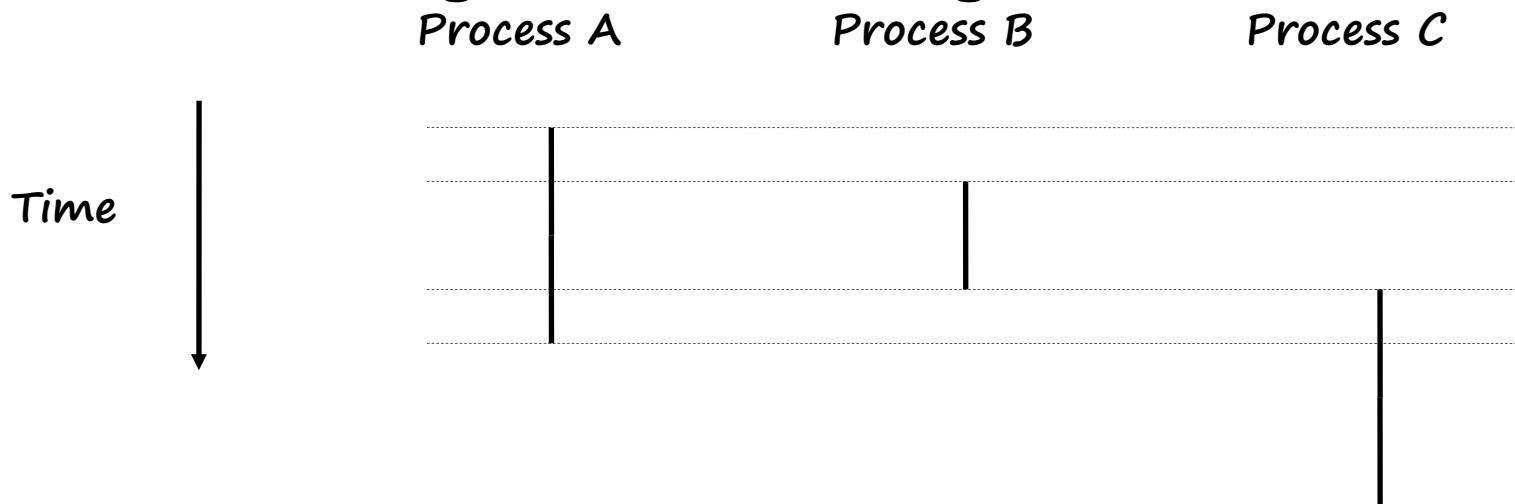
- Two processes run concurrently
  - are concurrent, if their flows overlap in time.
- Otherwise, they are sequential.
- Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C



# User view of concurrent processes

---

- Control flows for concurrent processes are physically disjoint in time.
- However, we can think of concurrent processes are running in parallel with each other.
- Multitasking or time slicing



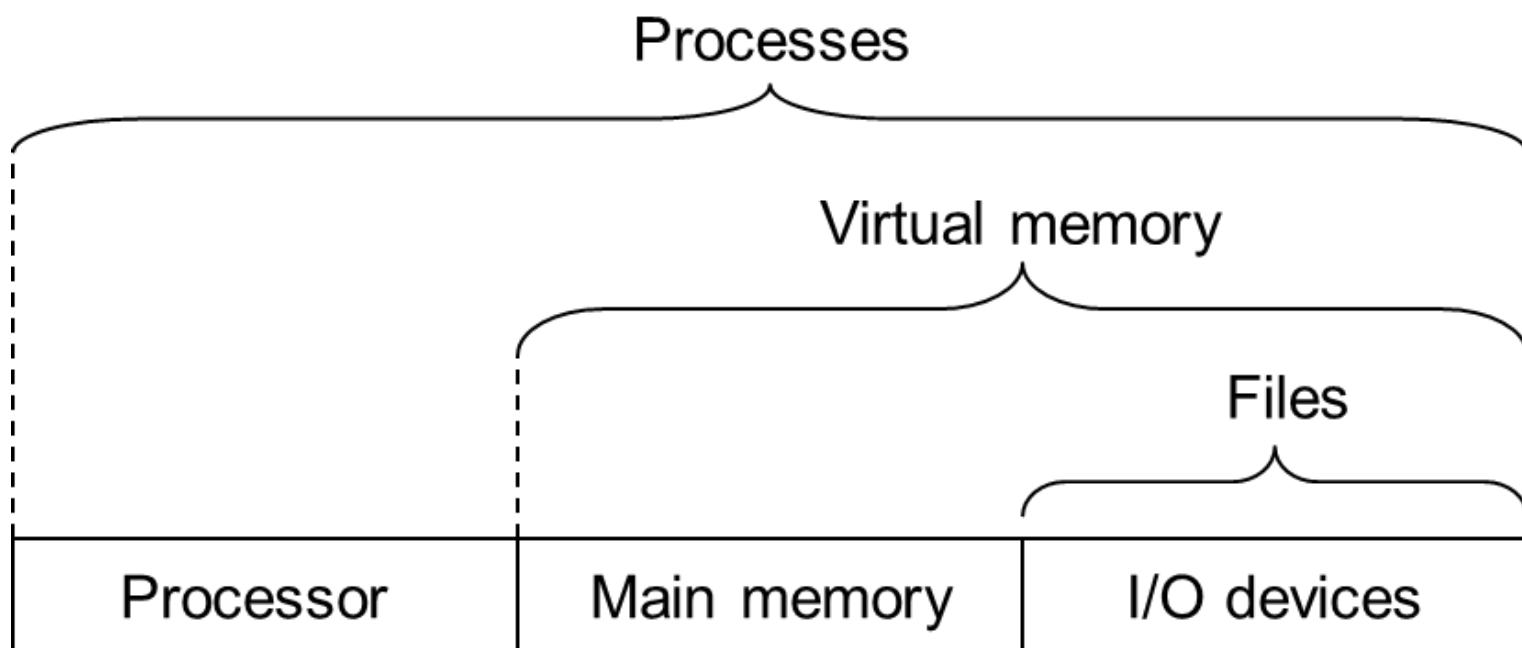
# Contexts of Processes

---

- The context consists of the state that the program needs to run correctly
- This state includes
  - the program's code and data stored in memory
  - its stack
  - the contents of its general-purpose registers
  - its program counter
  - environment variables
  - and the set of open file descriptors

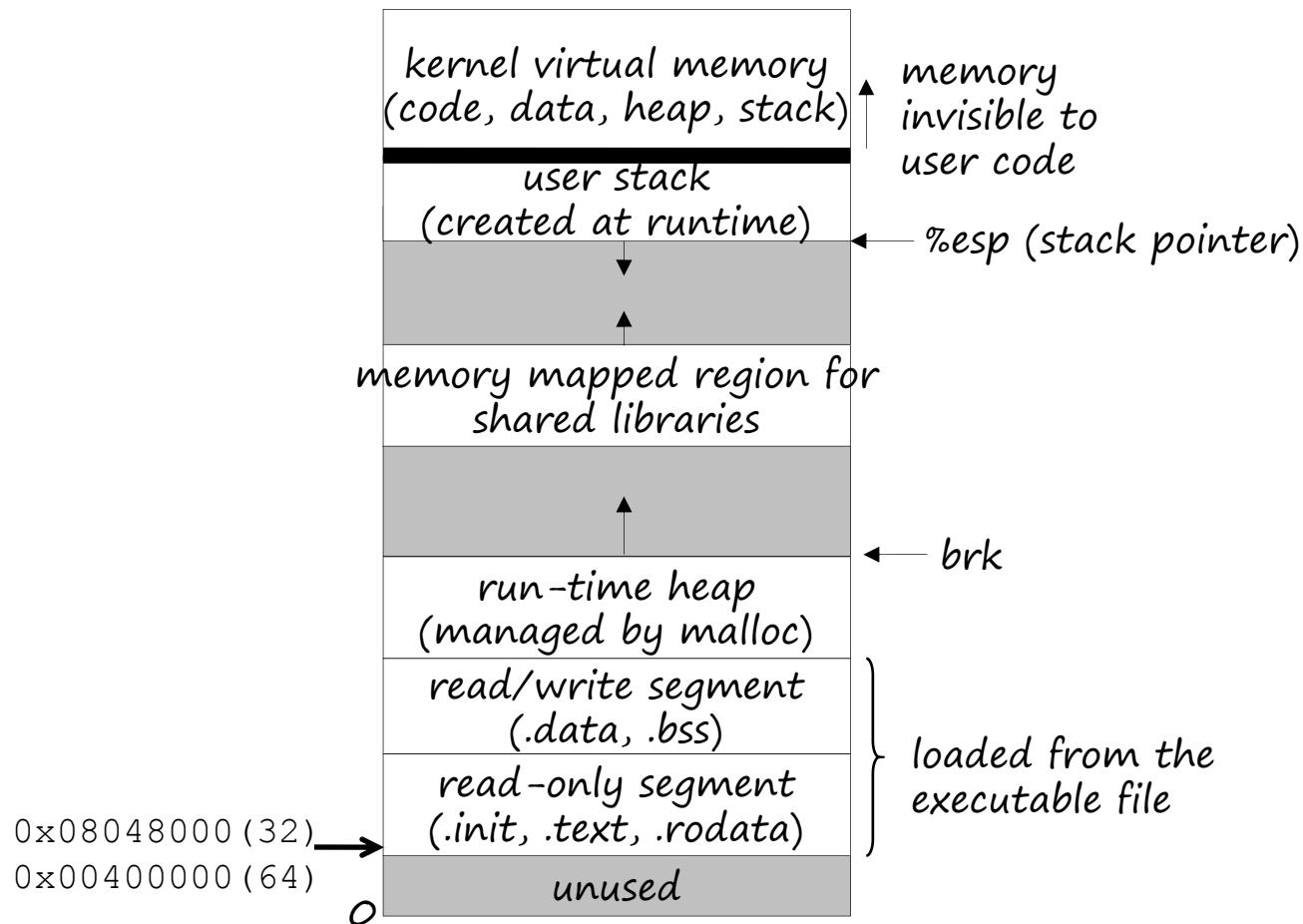
# Abstractions Provided by OS

---



# Private address spaces

- Each process has its own private address space



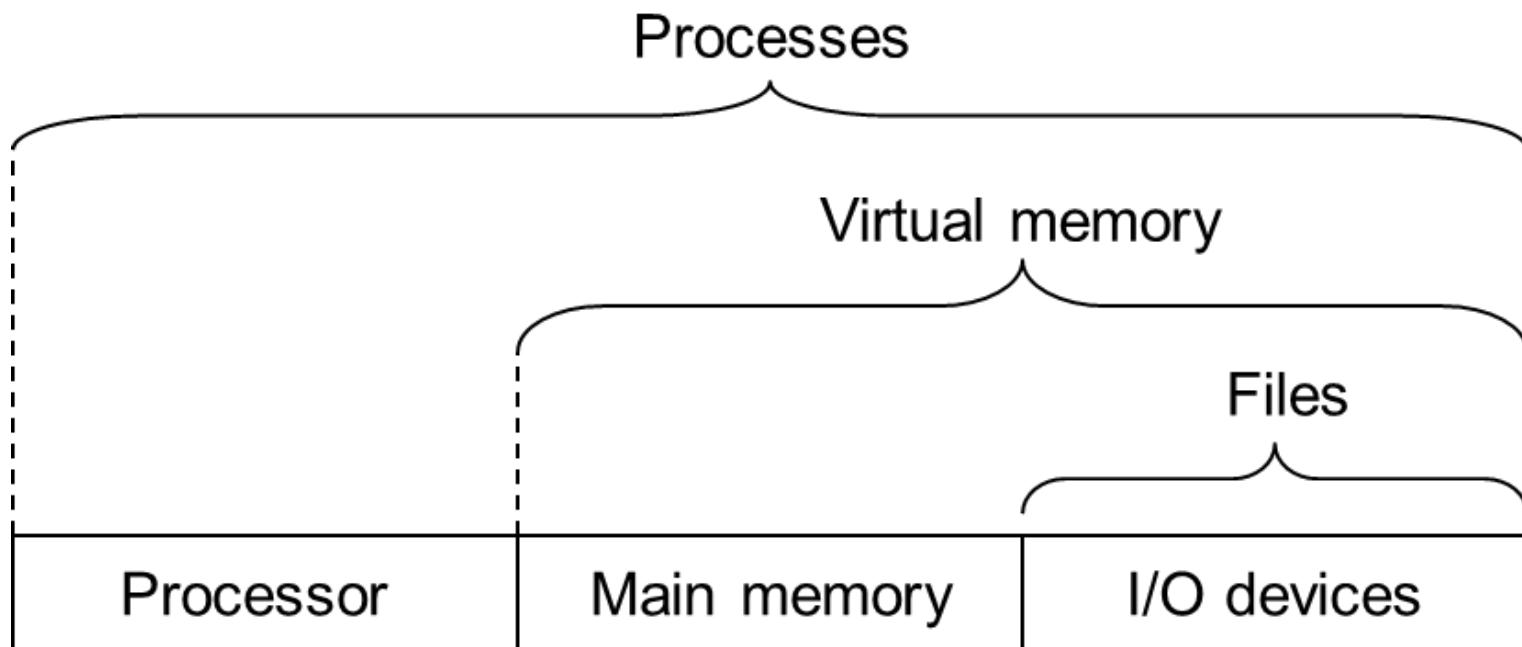
# Virtual Memory

---

- An abstraction
  - Each process appears to have exclusive use of the main memory
  - Virtual address space
    - Program code and data (global variables)
    - Heap
    - Shared libraries (standard and math libraries)
    - Stack (function calls)
    - Kernel (operating system)
  - Hardware supports to translate the virtual address

# Abstractions Provided by OS

---



# Files

---

- A sequence of bytes
- Each I/O device is modeled as a file
- Unix I/O
  - using a small set of system calls reading and writing files
  - All input and output in the system is performed by Unix I/O

# Outline

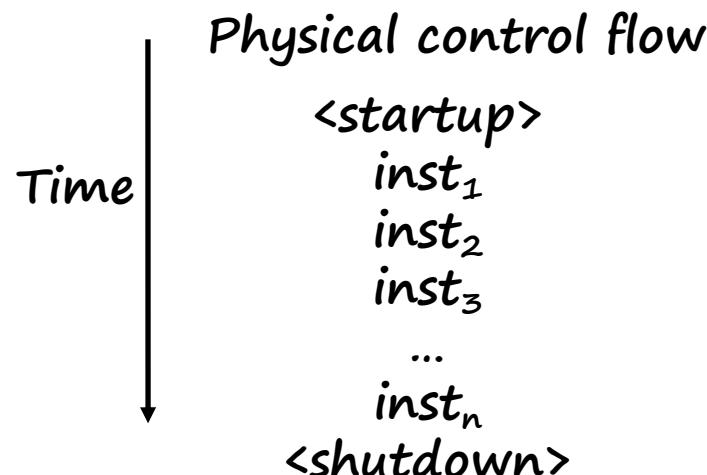
---

- Exceptions
- Suggested reading 8.1

# Control flow

---

- From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time.
- This sequence is the system's *physical control flow* (or *flow of control*).



# Altering the Control Flow

---

- We've discussed two mechanisms for changing the control flow:
  - Jumps and branches
  - Call and return using the stack discipline.
  - Both react to changes in program state.
- How can we change the control flows among processes (multitasking) ?

# Altering the Control Flow

---

- Other needs for the CPU to react to changes in system state
  - data arrives from a disk or a network adapter.
  - instruction divides by zero
  - user hits `ctl-c` at the keyboard
- System needs mechanisms for “exceptional control flow”

# Exceptions for Y86(review)

---

- Conditions under which pipeline cannot continue normal operation
- Causes
  - Halt instruction (Current)
  - Bad address for instruction or data (Previous)
  - Invalid instruction (Previous)
  - Pipeline control error (Previous)

# Exceptions for Y86(review)

---

- Desired Action
  - Complete some instructions
    - Either current or previous (depends on exception type)
  - Discard others
  - Call exception handler
    - Like an unexpected procedure call

# Exceptional control flow

---

- Mechanisms for exceptional control flow exists at all levels of a computer system.
- Low level mechanism:
  - exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Implemented as a combination of both hardware and OS software

# Exceptional control flow

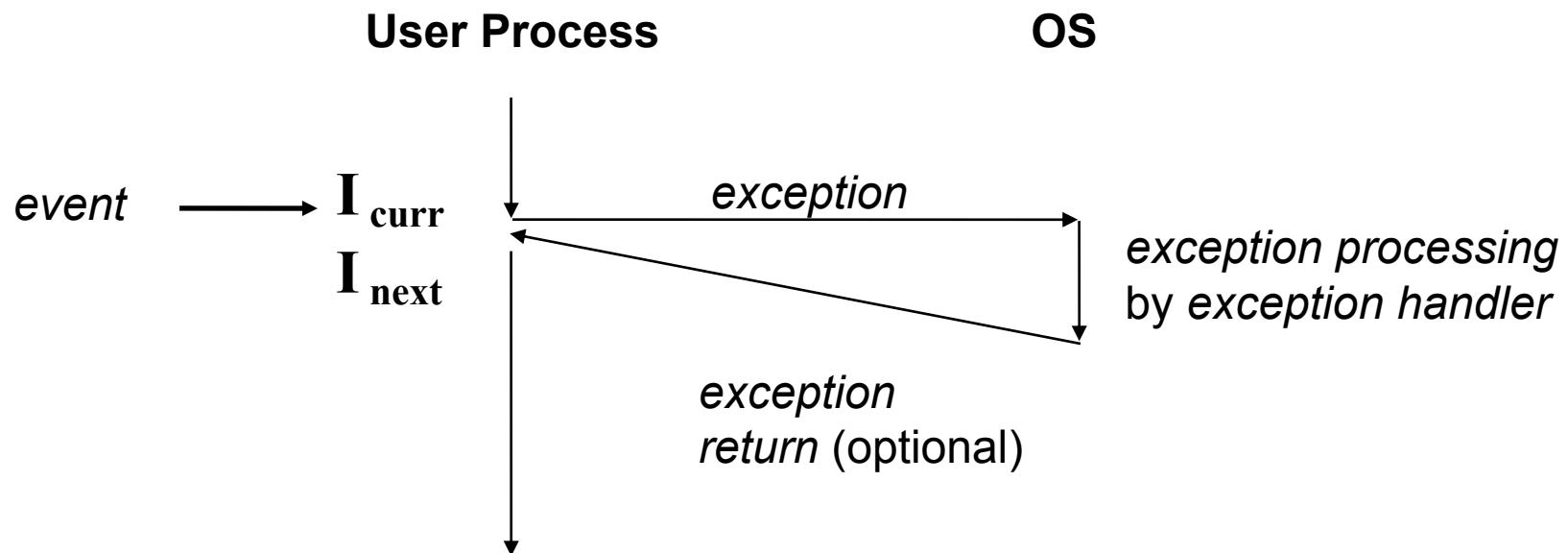
---

- Higher level mechanisms:
  - process context switch (OS level)
  - signals
  - nonlocal jumps (`setjmp/longjmp`)
  - Implemented by either:
    - OS software (context switch and signals).
    - C language runtime library: nonlocal jumps.

# Exceptions

---

- An exception is a transfer of control to the OS in response to some event (i.e., change in processor state)



# Event

---

- Significant changes in the processor 's state
- Encoded in various bits and signals inside the processor
- It may be related to the execution of the current instruction
  - Page fault, arithmetic overflow
- It may be unrelated to the execution of the current instruction
  - A system timer goes off,
  - An I/O request completes

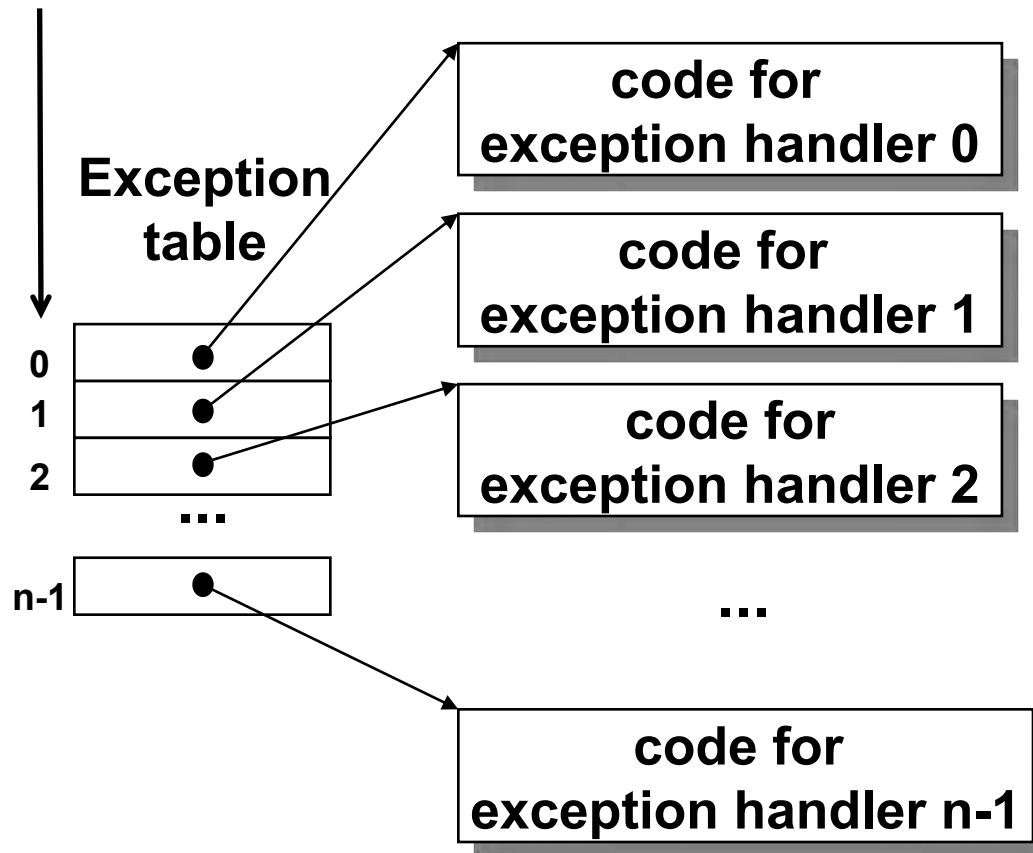
# Exceptions

---

- When the exception handler finishes processing:
  1. The handler returns control to the current instruction  $I_{curr}$
  2. The handler returns control to the next instruction  $I_{next}$
  3. The handler aborts the interrupted program

# Exception Table

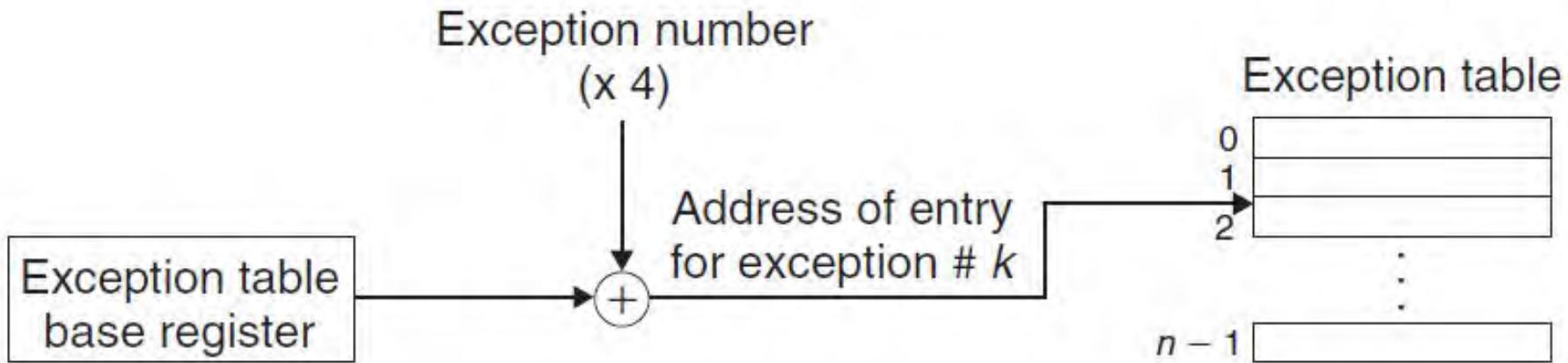
Exception  
numbers



1. Each type of event has a unique exception number  $k$
2. Exception table entry  $k$  points to a function (exception handler).
3. Handler  $k$  is called each time exception  $k$  occurs.

# Exception Table

---



# Exception Handler

---

- The processor pushes a return address on the stack the return address is
  - either the current instruction
  - or the next instruction
- The processor also pushes some additional processor state onto the stack
  - will be necessary to restart the interrupted program when the handler returns
    - e.g. the current condition codes

# Exception Handler

---

- All of these items are pushed onto the **kernel's stack**
  - rather than onto the user's stack
  - If control is being transferred from a user program to the kernel
- Exception handlers run in kernel mode
  - means they have complete access to all system resources

# Exceptional Control Flow II

# Outline

---

- Classes of Exceptions
- Kernel Mode and User Mode
- Context Switch
- System Calls and Error Handling
- Process Control
- Suggested reading 8.1.4, 8.2, 8.3, 8.4

# Asynchronous exceptions (interrupts)

---

- Caused by events (changes in state) external to the processor
  - Indicated by setting the processor's interrupt pin
  - handler returns to "next" instruction.

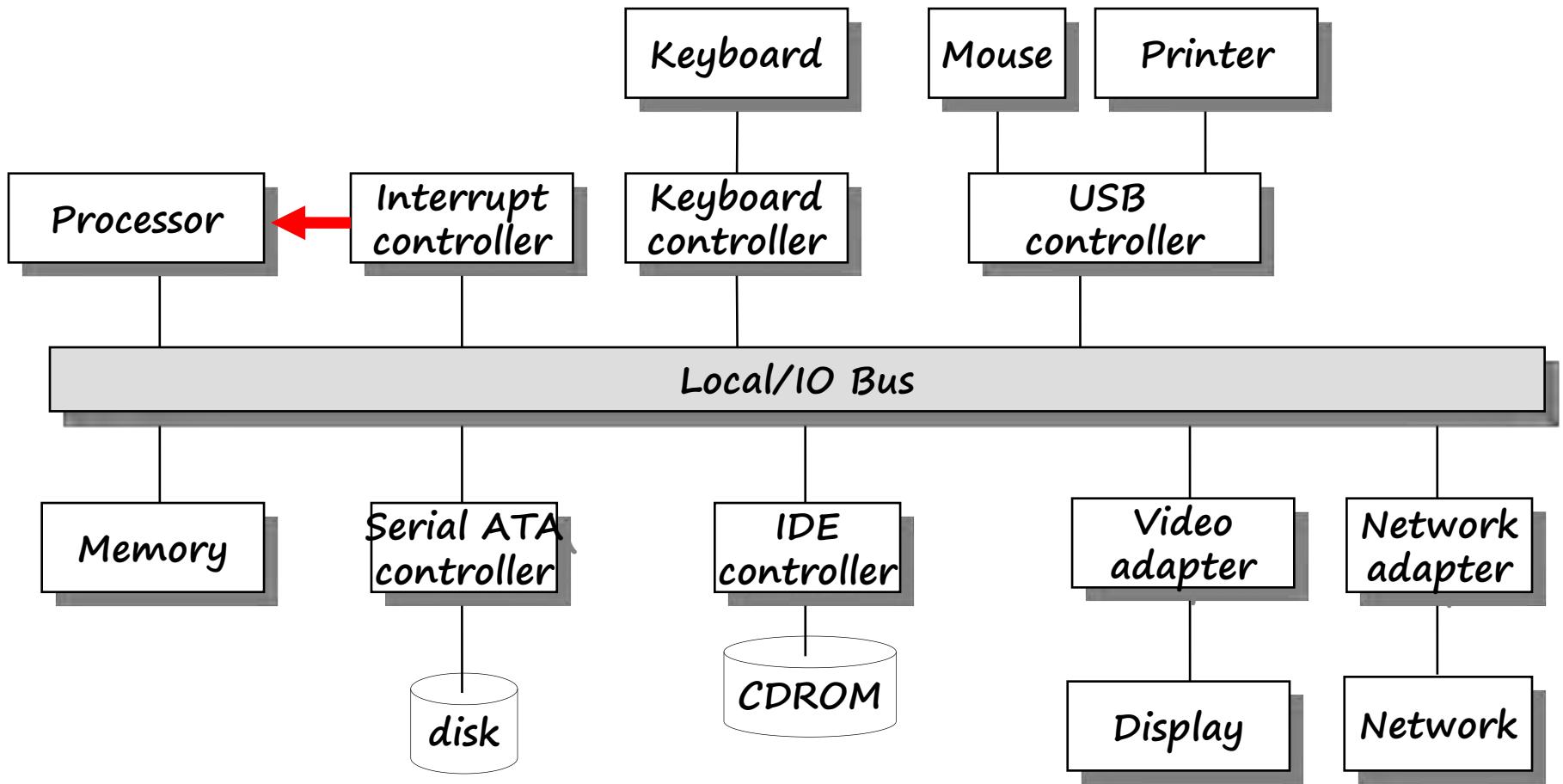
# Asynchronous exceptions (interrupts)

---

- Examples:
  - I/O interrupts
    - hitting `ctl-c` at the keyboard
    - arrival of a packet from a network
    - arrival of a data sector from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting `ctl-alt-delete` on a PC

# System context for exceptions

---



# Synchronous exceptions

---

- Caused by events that occur as a result of executing an instruction
  - e.g. faulty instruction
- Traps
  - Intentional
  - returns control to “next” instruction
  - Examples: **system calls**, breakpoint traps

# Synchronous exceptions

---

- Faults
  - Unintentional but possibly recoverable
  - either re-executes faulting ("current") instruction or aborts.
  - Examples: **page faults** (recoverable), protection faults (unrecoverable)
- Aborts
  - Unintentional and unrecoverable
  - aborts current program
  - Examples: **parity error**, machine check

# Synchronous exceptions

---

- Exceptions in IA32 systems

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–127	OS-defined exceptions	Interrupt or trap
128 (0x80)	System call	Trap
129–255	OS-defined exceptions	Interrupt or trap

# Fault Example #1

---

- Memory Reference
  - User writes to memory location
  - That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7: c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```

# Fault Example #1

---

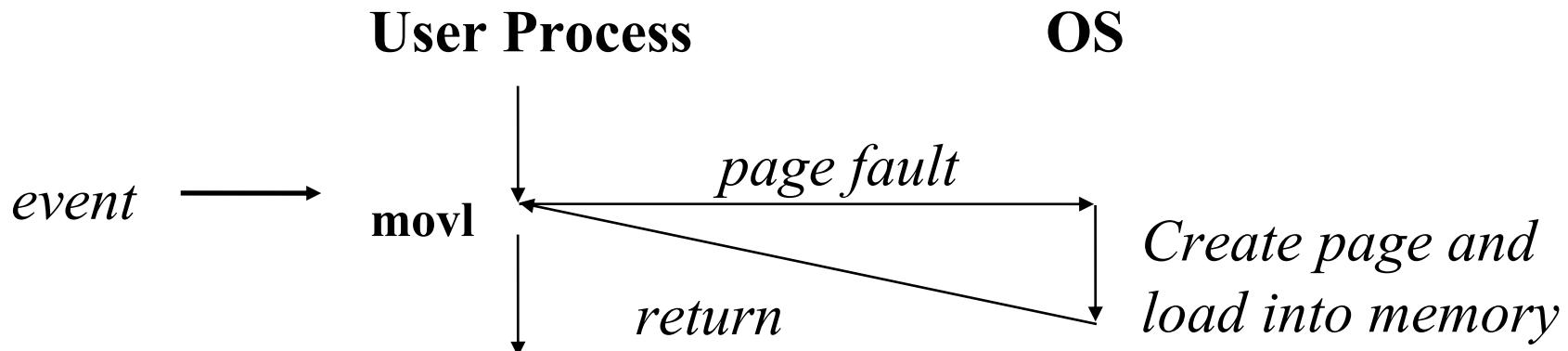
- Memory Reference
  - Page fault handler must load page into physical memory
  - Returns to faulting instruction
  - Successful on second try

# Fault Example #1

---

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7: c7 05 10 9d 04 08 0d  movl    $0xd,0x8049d10
```



# Fault Example #2

---

- Memory Reference
  - User writes to memory location
  - Address is not valid

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7: c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```

## Fault Example #2

---

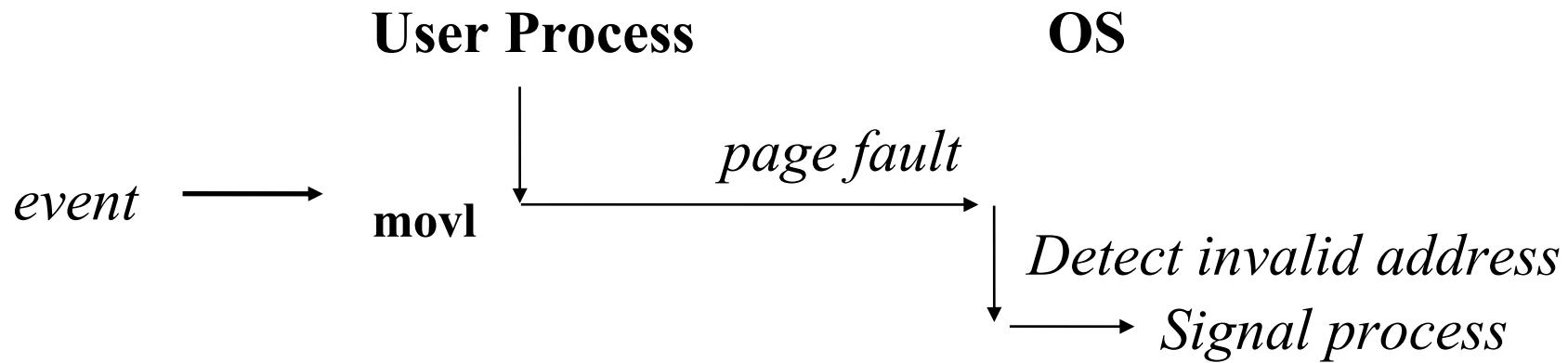
- Memory Reference
  - Page fault handler detects invalid address
  - Sends SIGSEG signal to user process
  - User process exits with "segmentation fault"

# Fault Example #2

---

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7: c7 05 60 e3 04 08 0d  movl    $0xd,0x804e360
```



# Synchronous exceptions

---

- Traps
  - intentional exception
  - A result of executing an instruction.
  - Examples: read a file (read), create process (fork)
- System Calls
  - A procedure-like interface between user programs and operating systems
  - Controlled access to kernel services

# System Calls

---

- System calls in IA32 systems

Number	Name	Description	Number	Name	Description
1	exit	Terminate process	27	alarm	Set signal delivery alarm clock
2	fork	Create new process	29	pause	Suspend process until signal arrives
3	read	Read file	37	kill	Send signal to another process
4	write	Write file	48	signal	Install signal handler
5	open	Open file	63	dup2	Copy file descriptor
6	close	Close file	64	getppid	Get parent's process ID
7	waitpid	Wait for child to terminate	65	getpgrp	Get process group
11	execve	Load and run program	67	sigaction	Install portable signal handler
19	lseek	Go to file offset	90	mmap	Map memory page to file
20	getpid	Get process ID	106	stat	Get information about file

# System Call Example

---

```
# hello world
1 int main()
2 {
3     write(1, "hello, world\n", 13);
4     exit(0);
5 }
```

# System Call Example

---

```
1 .section .data
2 string:
3     .ascii "hello, world\n"
4 string_end:
5     .equ len, string_end - string
6 .section .text
7 .globl main
8 main:
```

# System Call Example

---

*First, call write(1, "hello, world\n", 13)*

9      movl \$4, %eax	<i>System call number 4</i>
10     movl \$1, %ebx	<i>stdout has descriptor 1</i>
11     movl \$string, %ecx	<i>Hello world string</i>
12     movl \$len, %edx	<i>String length</i>
13     int \$0x80	<i>System call code</i>

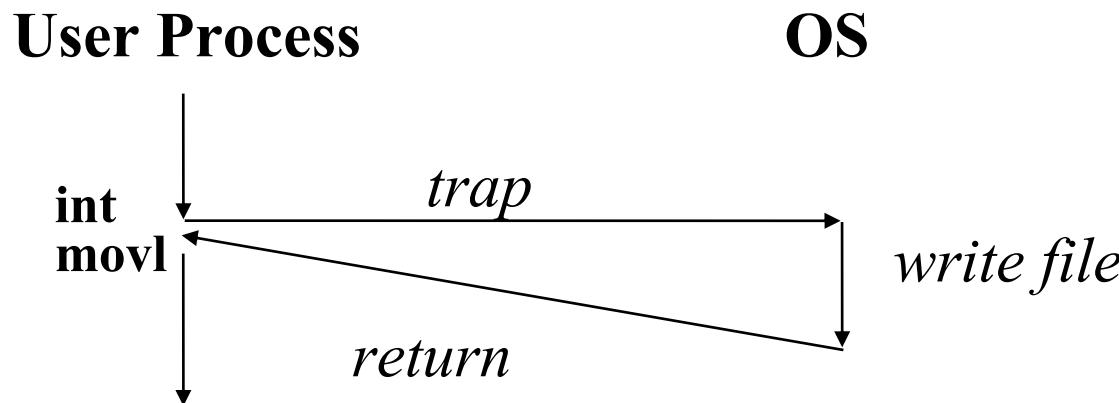
*Next, call exit(0)*

14     movl \$1, %eax	<i>System call number 0</i>
15     movl \$0, %ebx	<i>Argument is 0</i>
16     int \$0x80	<i>System call code</i>

# Trap Example

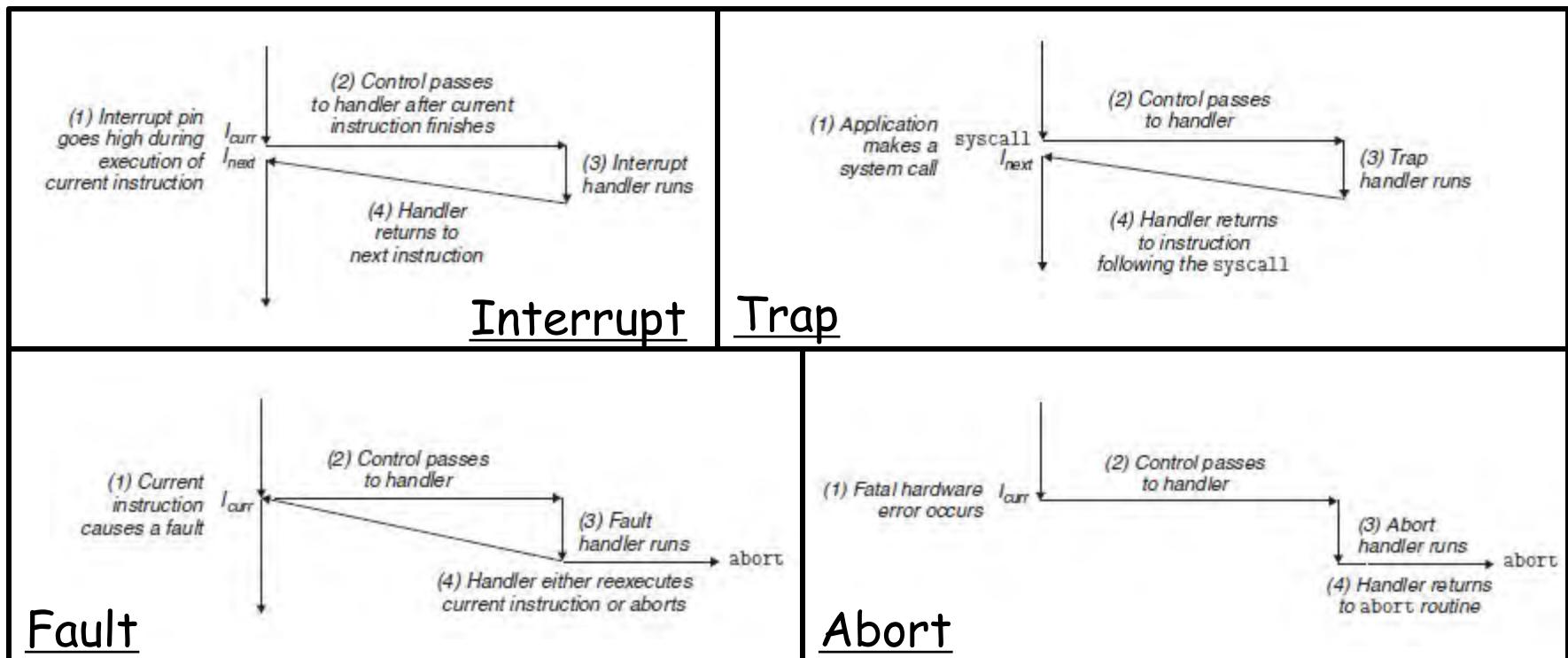
---

- Writing a File
  - OS must find the file, write the string to it
  - Returns integer whether it is succeeded



# Exceptions

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

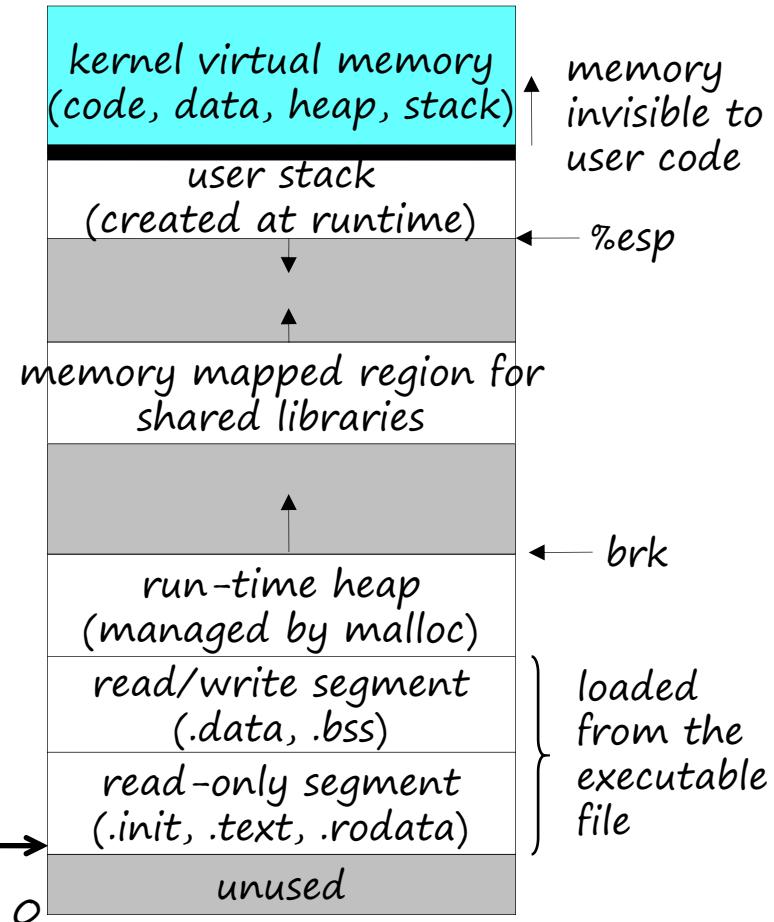


# *Kernel and User Mode*

# User and Kernel Modes

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process

0x08048000 (32)  
0x00400000 (64)



# Protection

---

- Restricts an application
  - to execute some instructions (privileged instruction)
  - to access some portions of the address space
- In some control registers there typically is a mode bit
  - Kernel (supervisor) mode if the bit is set
  - User mode if the bit is clear

# User and Kernel Modes

---

- Process running in kernel mode can
  - execute **any** instructions in the instruction set
  - can access **any** memory locations in the system
- Process running in user mode can
  - **neither** execute privileged instructions
  - **nor** directly reference code or data in the kernel area of the address space
  - only do above indirectly via system call interface

# User and Kernel Modes

---

- A process running application code
  - initially in user mode
  - changes the user mode to kernel mode
    - When the **exception** occurs and control passes to the exception handler
  - changes the kernel mode to user mode
    - When control returns to the application code
- The only way for the process to change from user mode to kernel mode is via exception

# Implementing the Process Abstraction

---

- Requires close cooperation between both
  - the low-level hardware and
  - the operating system software

# **Context Switches**

# Context switching

---

- The kernel maintains a context for each process
- The context is the state that the kernel needs to restart a preempted process
- Context contains
  - Value of PC, register file, status registers, user's stack, kernel's stack
  - Kernel data structures
    - Process table, page table, file table

# Context switching

---

- The mechanism that performs the multitasking (time slicing)
- Higher-level form of exceptional control flow
  - built on top of the lower-level exception mechanism

# Context switching

---

- The scheduler (a chunk of kernel code) does scheduling as following
  - Decides whether to preempt the current process during the execution of a process
  - Selects a previously preempted process (scheduled process) to restart
  - Preempts the current process
    - **Saves** the context of the current process
  - Restarts the scheduled process
    - **Restores** the saved context of the scheduled process
    - Passes the control to this newly restored process

# Three states of a process

---

- Running
  - The process is either **executing** on the CPU
  - or is **waiting** to be executed
  - and will eventually be scheduled.
- Stopped
  - The execution of the process is **suspended**
  - and will not be scheduled.
- Terminated
  - The process is **stopped** permanently.

# Three states of a process

---

- A running process becomes stop
  - By receiving a **signal** such as
    - SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU
- A stopped process becomes running
  - By receiving a **SIGCONT signal**  
(a signal is a form of software interrupt)

# Three states of a process

---

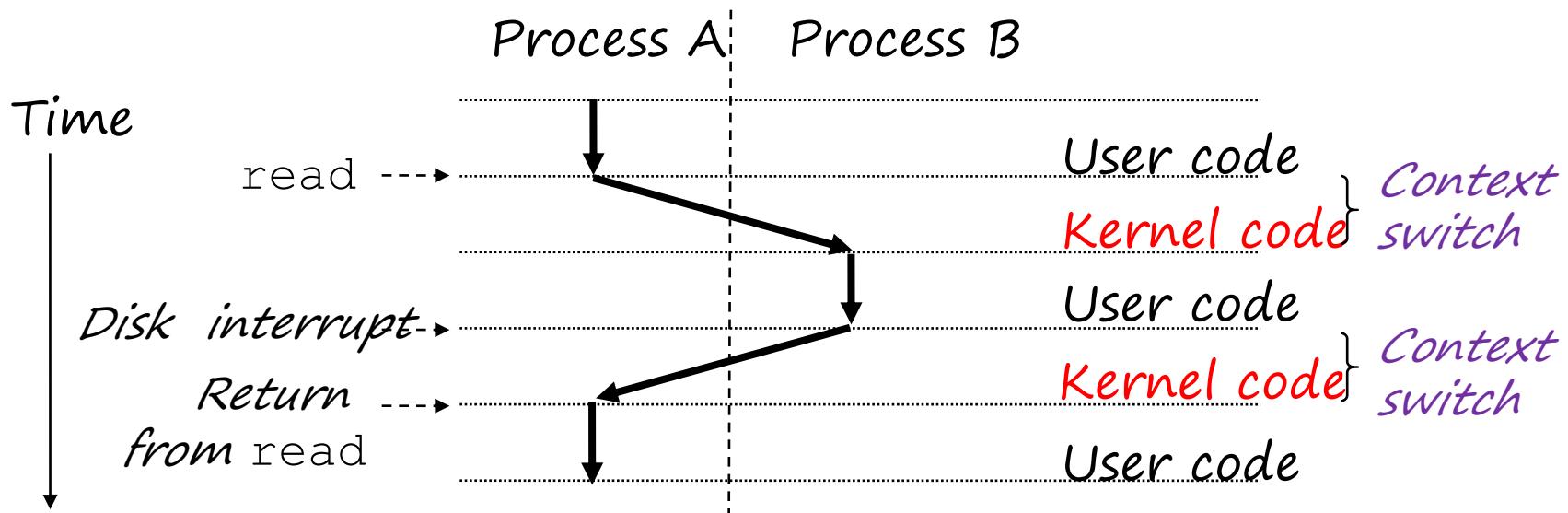
- A process becomes terminated
  - receiving a **signal** whose default action is to terminate the process
  - returning from the main routine
  - calling the `exit` function

# Context switching

---

- When does the context switch happen?
  - The kernel is executing a system call on behalf of the user such as
    - `read`, `sleep` , etc. which will cause the calling process **blocked**
    - Even if a system call does not block, the kernel can decide to perform a context switch rather than return control to the calling process
  - As a result of an interrupt
    - Timer interrupt

# Context switching



# System Call Error Handling

---

- Unix system-level functions encounter an error
  - typically return -1
  - set the global integer variable `errno`
    - to indicate what went wrong

# System Call Error Handling

---

```
1 if ((pid = fork()) < 0) {  
•     fprintf(stderr, "fork error: %s\n",  
•             strerror(errno));  
3     exit(0);  
4 }
```

---

```
1 void unix_error(char *msg) /* unix-style error */  
2 {  
•     fprintf(stderr, "%s: %s\n", msg, strerror(errno));  
4     exit(0);  
5 }  
  
1 if ((pid = fork()) < 0)  
2     unix_error("fork error");
```

# System Call Error Handling

---

```
1 pid_t Fork(void)
2 {
3     pid_t pid;
4
5     if ((pid = fork()) < 0)
6         unix_error("Fork error");
7     return pid;
8 }
```

# Process Control

# Obtaining Process ID's

---

- Process ID
  - PID
  - Each process has a unique positive PID
- `Getpid()`
  - Returns the PID of the calling process
- `Getppid()`
  - Returns the PID of its parent
    - The process that created the calling process

# Obtaining Process ID's

---

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void);
pid_t getppid(void);
```

returns: PID of either the caller or the parent

- The `getpid` and `getppid` routines return an integer value of type `pid_t`
- `pid_t`
  - Defined in `types.h` as an `int` on Linux systems

# Exit Function

---

```
#include <stdlib.h>
void exit(int status);
```

this function does not return

- The `exit` function terminates the process with an `exit status` of `status`.

# Fork Function

---

- A parent process
  - creates a new running child process
  - by calling the `fork` function

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);
```

Returns: 0 to child, PID of child to parent, -1 on error

# Fork Function

---

- The newly created child process is
  - **almost**, but not quite, identical to the parent.
- The parent and the newly created child have different PIDs.

# Fork Function

---

- The child
  - gets an identical (but separate) copy of the parent's user-level virtual address space
    - including the **text**, **data**, and **bss** segments, **heap**, and user **stack**.
  - also gets identical copies of any of the parent's open **file descriptors**
    - Which means the child can read and write any files that were open in the parent when it called **fork**.

# Fork Function

---

- Called once
  - In the parent process
- Returns twice:
  - In the parent process
    - Return the PID of the child
  - In the newly created child process.
    - Return 0

# Fork Function

---

- The return value provides an unambiguous way
  - whether the program is executing in the parent or the child.

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6     int x = 1;
7
8     pid = Fork();
9     if (pid == 0) { /* child */
10         printf("child : x=%d\n", ++x);
11         exit(0);
12     }
13
14     /* parent */
15     printf("parent: x=%d\n", --x);
16     exit(0);
17 }
```

# Fork Function

---

- Call once, return twice.
  - As mentioned before
  - This is fairly straightforward for programs that create a single child.
  - Programs with multiple instances of `fork`
    - can be confusing
    - need to be reasoned about carefully

# Fork Function

---

- Concurrent execution
  - The parent and the child are separate processes that run concurrently.
  - The instructions in their logical control flows can be interleaved by the kernel in an arbitrary way.
  - We can never make assumptions about the interleaving of the instructions in different processes.

# Fork Function

---

- Duplicate but separate address spaces
  - Immediately after the `fork` function returned in each process, the address space of each process is identical.
    - Local variable `x` has a value of 1 in both the parent and the child when the `fork` function returns in line 8.

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6     int x = 1;
7
8     pid = Fork();
9     if (pid == 0) { /* child */
10         printf("child : x=%d\n", ++x);
11         exit(0);
12     }
13
14     /* parent */
15     printf("parent: x=%d\n", --x);
16     exit(0);
17 }
```

# Fork Function

---

- Duplicate but separate address spaces.
  - The parent and the child are separate processes
    - they each have their own **private** address spaces.
    - Any subsequent changes that a parent or child makes to ~~x~~
      - private
      - not reflected in the memory of the other process.

# Fork Function

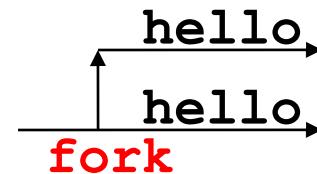
---

- Duplicate but separate address spaces
  - The variable `x` has different values in the parent and child when they call their respective `printf` statements.
- Shared files
  - Like `stdout`
  - Communniation between child and parent

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     printf("hello!\n");
7     exit(0);
8 }
```



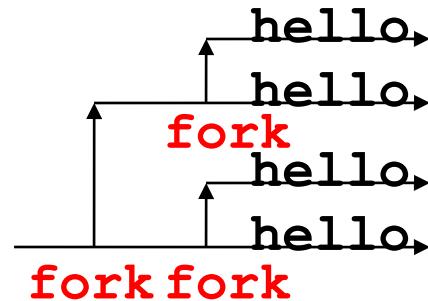
(a) Calls `fork` once.

(b) Prints two output lines.

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     printf("hello!\n");
8     exit(0);
9 }
```

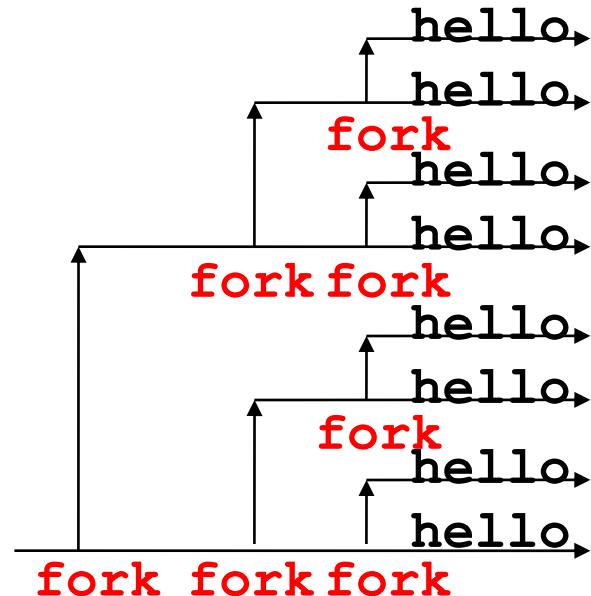


- (c) Calls **fork twice**.      (d) Prints **four output lines**.

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     Fork();
8     printf("hello!\n");
9     exit(0);
10 }
```



- (e) Calls fork **three** times. (f) Prints **eight** output lines.

# Exceptional Control Flow III

# Outline

---

- Creating Processes
- Reaping Child Processes
- Putting Processes to Sleep
- Loading and Running Programs
- Suggested reading 8.4

# Fork Function

---

- A parent process
  - creates a new running child process
  - by calling the `fork` function

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);
```

Returns: 0 to child, PID of child to parent, -1 on error

# Fork Function

---

- The newly created child process is
  - **almost**, but not quite, identical to the parent.
- The parent and the newly created child have different PIDs.

# Fork Function

---

- The child
  - gets an identical (but separate) copy of the parent's user-level virtual address space
    - including the **text**, **data**, and **bss** segments, **heap**, and user **stack**.
  - also gets identical copies of any of the parent's open **file descriptors**
    - Which means the child can read and write any files that were open in the parent when it called **fork**.

# Fork Function

---

- Called once
  - In the parent process
- Returns twice:
  - In the parent process
    - Return the PID of the child
  - In the newly created child process.
    - Return 0

# Fork Function

---

- The return value provides an unambiguous way
  - whether the program is executing in the parent or the child.

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6     int x = 1;
7
8     pid = Fork();
9     if (pid == 0) { /* child */
10         printf("child : x=%d\n", ++x);
11         exit(0);
12     }
13
14     /* parent */
15     printf("parent: x=%d\n", --x);
16     exit(0);
17 }
```

# Fork Function

---

- Call once, return twice.
  - As mentioned before
  - This is fairly straightforward for programs that create a single child.
  - Programs with multiple instances of `fork`
    - can be confusing
    - need to be reasoned about carefully

# Fork Function

---

- Concurrent execution
  - The parent and the child are separate processes that run concurrently.
  - The instructions in their logical control flows can be interleaved by the kernel in an arbitrary way.
  - We can never make assumptions about the interleaving of the instructions in different processes.

# Fork Function

---

- Duplicate but separate address spaces
  - Immediately after the `fork` function returned in each process, the address space of each process is identical.
    - Local variable `x` has a value of 1 in both the parent and the child when the `fork` function returns in line 8.

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6     int x = 1;
7
8     pid = Fork();
9     if (pid == 0) { /* child */
10         printf("child : x=%d\n", ++x);
11         exit(0);
12     }
13
14     /* parent */
15     printf("parent: x=%d\n", --x);
16     exit(0);
17 }
```

# Fork Function

---

- Duplicate but separate address spaces.
  - The parent and the child are separate processes
    - they each have their own **private** address spaces.
    - Any subsequent changes that a parent or child makes to ~~x~~
      - private
      - not reflected in the memory of the other process.

# Fork Function

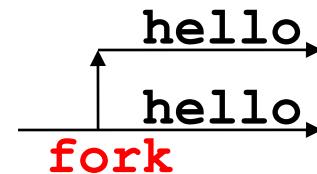
---

- Duplicate but separate address spaces
  - The variable `x` has different values in the parent and child when they call their respective `printf` statements.
- Shared files
  - Like `stdout`
  - Communniation between child and parent

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     printf("hello!\n");
7     exit(0);
8 }
```



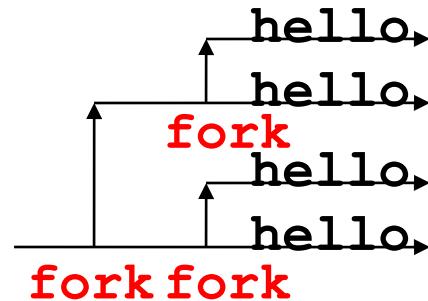
(a) Calls `fork` once.

(b) Prints two output lines.

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     printf("hello!\n");
8     exit(0);
9 }
```

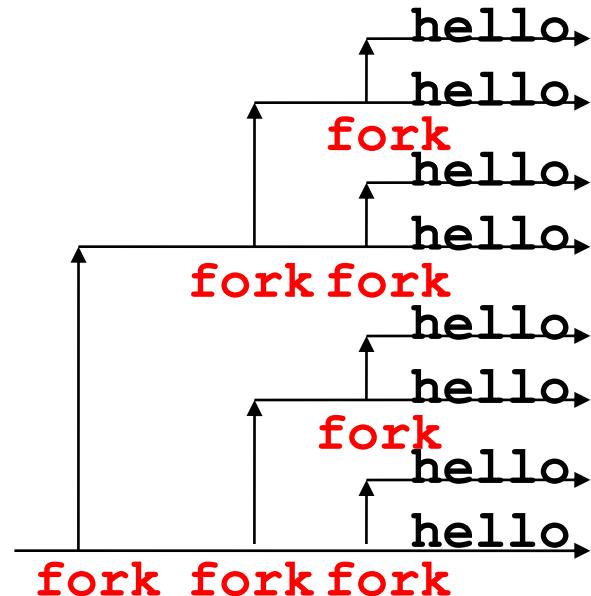


- (c) Calls **fork twice**.      (d) Prints **four output lines**.

# Fork Function

---

```
1 #include "csapp.h"
2
3 int main()
4 {
5     Fork();
6     Fork();
7     Fork();
8     printf("hello!\n");
9     exit(0);
10 }
```



- (e) Calls `fork` **three** times. (f) Prints **eight** output lines.

# Zombie

---

- The kernel does not remove a terminated process from the system immediately after the process terminates
- The process is kept around in a terminated state until it is reaped by its parent.

# Zombie

---

- When the parent reaps the terminated child
  - the kernel passes the child's exit status to the parent,
  - and then discards the terminated process, at which point it ceases to exist.
- A terminated process that has not yet been reaped is called a **zombie**.

# Zombie

---

- If the parent process
  - terminates without reaping its zombie children,
  - the kernel arranges for the **init** process to reap them.
- The **init** process
  - has a PID of 1
  - and is created by the kernel during system initialization.

# Zombie

---

- Long-running programs such as shells or servers should always reap their zombie children.
- Even though zombies are not running, they still consume system memory resources.

# Wait\_pid Function

---

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait(int *status);
```

Returns: PID of child if OK, 0 (if WNOHANG) or -1 on error

# Wait\_pid Function

---

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **pid > 0**
  - The wait set is the singleton child process
    - whose process ID is equal to **pid**
- **pid = -1**
  - The wait set consists of all of the parent's child processes

# Wait\_pid Function

---

- The `waitpid` function
  - If the calling process has no children
    - returns -1
    - sets `errno` to `ECHILD`
  - If interrupted by a signal
    - returns -1
    - sets `errno` to `EINTR`

# Wait\_pid Function

---

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **options = 0**
  - waitpid suspends execution of the calling process until a child process in its wait-set terminates
  - If a process in the wait set has already terminated at the time of the call, then waitpid returns immediately
  - waitpid returns the PID of the terminated child that caused waitpid to return
  - the terminated child is removed from the system

# Wait\_pid Function

---

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- options
  - WNOHANG: Return immediately (with a return value of 0 ) if none of the child processes in the waiting set has terminated yet
  - WUNTRACED: Suspended execution of the calling process until a process terminated or stopped child that caused the return

# Wait\_pid Function

---

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- options
  - WUNTRACED | WNOHANG : Suspended execution of the calling process until a process terminated or stopped child that caused the return. Also, return immediately (with a return value of 0 ) if none of the child processes in the wait-set is terminated or stopped

# Wait\_pid Function

---

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Checking the exit status of a reaped child
  - The `status` argument is non-NULL
  - `waitpid` encodes status information about the child
    - that caused the return in the `status` argument.
  - The `wait.h` include file defines several macros
    - for interpreting the `status` argument

# Wait\_pid Function

---

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **WIFEXITED (status)**
  - Returns true if the child terminated normally
    - via a call to `exit` or a return.
- **WEXITSTATUS (status)**
  - Returns the exit status of a normally terminated child.
  - This status is only defined if **WIFEXITED** returned true.

# Wait\_pid Function

---

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **WIFSIGNALED(status)**
  - Returns true if the child process terminated because of a signal that was not caught
- **WTERMSIG(status)**
  - Returns the number of the signal that caused the child process to terminate.
  - This status is only defined if **WIFSIGNALED** returned true.

# Wait\_pid Function

---

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **WIFSTOPPED(status)**
  - Returns true if the child that caused the return is currently stopped.
- **WSTOPSIG(status)**
  - Returns the number of the signal that caused the child to stop.
  - This status is only defined if **WIFSTOPPED** returned true.

# Wait\_pid Function

---

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6     int status, i;
7     pid_t pid;
8
9     /* Parent creates N children */
10    for (i = 0; i < N; i++)
11        if ((pid = Fork()) == 0) /* child */
12            exit(100+i);
13
```

# Wait\_pid Function

---

```
14 /* Parent reaps N chds. in no particular order */
15 while ((pid = waitpid(-1, &status, 0)) > 0) {
16     if (WIFEXITED(status))
17         printf("child %d terminated normally with exit
18                 status=%d\n", pid, WEXITSTATUS(status));
19     else
20         printf("child %d terminated abnormally\n", pid);
21 }
22
23 /* The only normal term. is if there no more chds. */
24 if (errno != ECHILD)
25     unix_error("waitpid error");
26
27 exit(0);
28}
```

# Wait\_pid Function

---

```
unix>./waitpid1
```

```
child 22966 terminated normally with exit status=100
```

```
child 22967 terminated normally with exit status=101
```

# Wait\_pid Function

---

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6     int status, i;
7     pid_t pid[N+1], rtpid;
8
9     /* Parent creates N children */
10    for (i = 0; i < N; i++)
11        if ((pid[i] = Fork()) == 0) /* Child */
12            exit(100+i);
13
14    /* Parent reaps N children in order */
15    i = 0;
```

# Wait\_pid Function

---

```
15 while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {  
16     if (WIFEXITED(status))  
18         printf("child %d terminated normally with exit  
19             status=%d\n", retpid, WEXITSTATUS(status));  
20     else  
21         printf("child %d terminated abnormally\n", retpid);  
22 }  
23  
24 /* The only normal term. is if there are no more chds. */  
25 if (errno != ECHILD)  
26     unix_error("waitpid error");  
27  
28 exit(0);  
29}
```

# Putting Process to Sleep

---

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
int pause(void);
```

Returns: seconds left to sleep  
Always returns -1

- Suspends a process for some period of time.
- Returns zero if the requested amount of time has elapsed
- Returns the number of seconds still left to sleep otherwise.

# Loading and Running

---

```
#include <unistd.h>
int execve(const char *filename, const char *argv[],
            const char *envp[]);

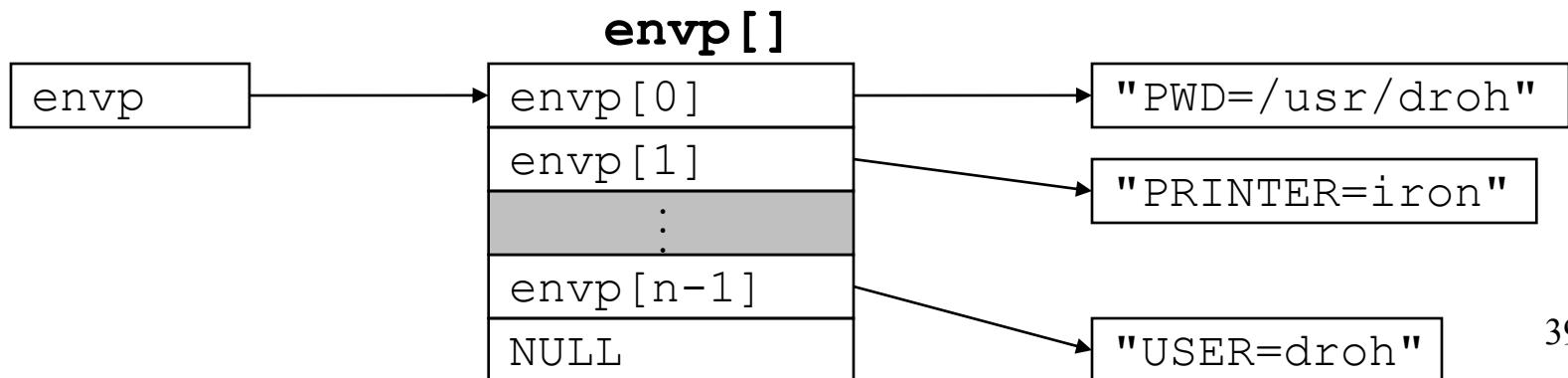
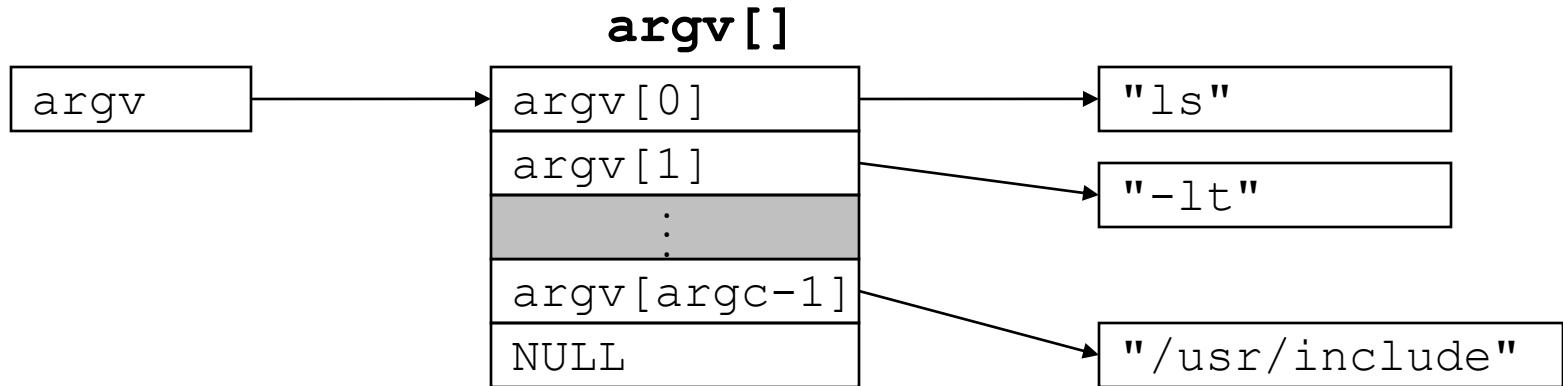
```

does not return if OK, returns -1 on error

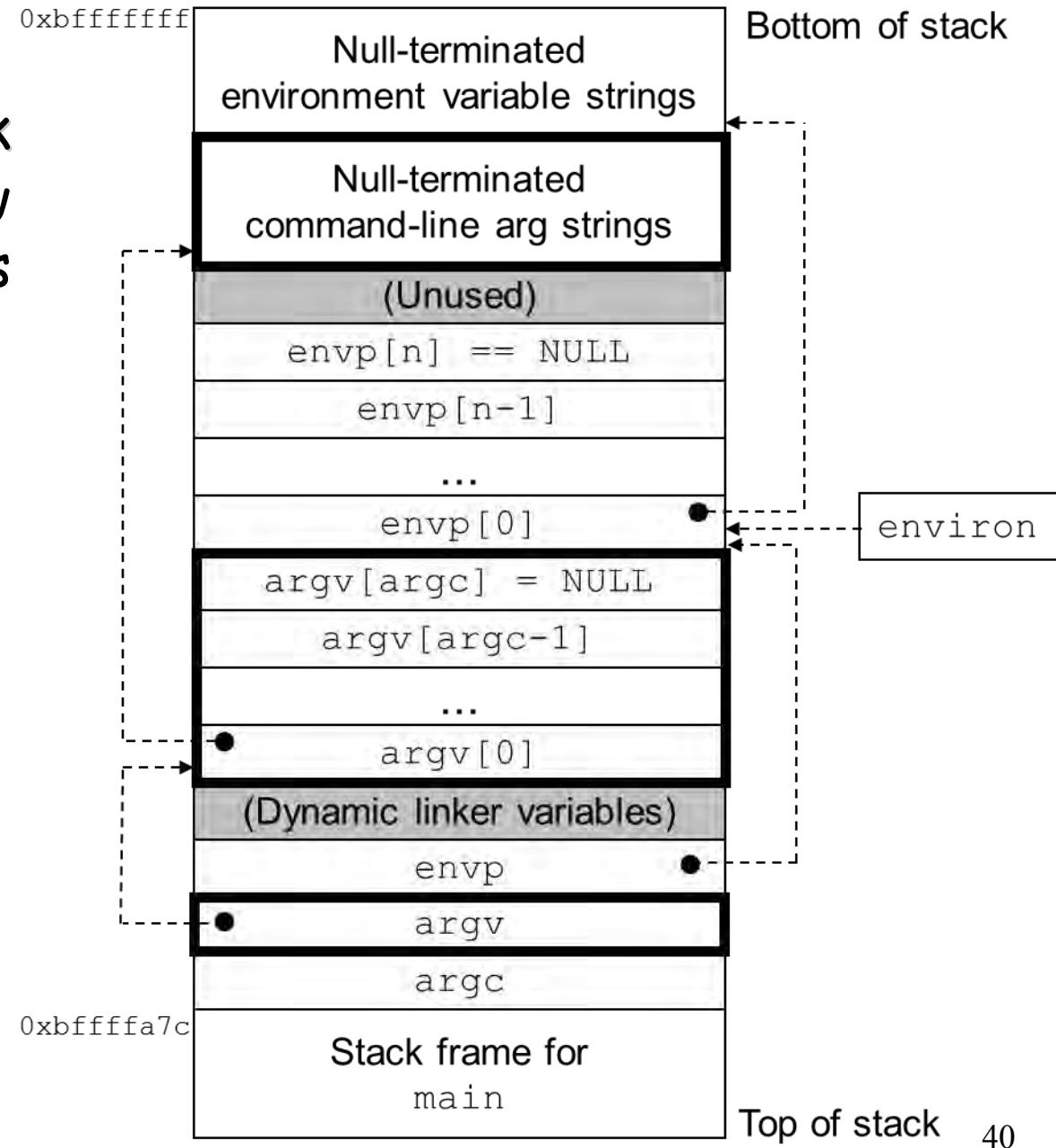
- Loads and runs
  - the executable object file `filename`
  - with the argument list `argv`
  - the environment variable list `envp`.
  - returns to the calling program only if there is an error
    - such as not being able to find `filename`.
- The `Execve` is called once and never returns<sub>38</sub>

# Loading and Running

```
int execve(const char *filename, const char **argv[],  
          const char **envp[]);
```



# The user stack when a new program starts



# Loading and Running

---

```
#include <unistd.h>
```

```
char *getenv(const char *name) ;
```

Returns: ptr to name if exists, NULL if no match.

```
int setenv(const char *name, const char *newvalue,  
          int overwrite) ;
```

Returns: 0 on success, -1 on error.

```
void unsetenv(const char *name) ;
```

Returns: nothing.

# Unix Shell

---

- An interactive application-level program that
  - runs other program on behalf of the user
- Variants: sh, csh, tcsh, ksh, bash
- Performs a sequence of read/evaluate steps and terminate
  - Read: reads a command line from the user
  - Evaluate: parses the command line and runs programs on behalf of the user

```
/* The main routine for a simple shell program */
1 #include "csapp.h"
2 #define MAXARGS 128
3
4 /* function prototypes */
5 void eval(char* cmdline);
6 int parseline(const char *cmdline, char **argv);
7 int builtin_command(char **argv);
8
```

```
9 int main()
10 {
11     char cmdline[MAXLINE]; /* command line */
12
13     while (1) {
14         /* read */
15         printf("> ");
16         Fgets(cmdline, MAXLINE, stdin);
17         if (feof(stdin))
18             exit(0);
19
20         /* evaluate */
21         eval(cmdline);
22     }
23 }
```

```
1  /* eval - evaluate a command line */
2  void eval(char *cmdline)
3  {
4      char *argv[MAXARGS]; /* argv for execve() */
5      char buf[MAXLINE]; /* holds modified cmd line */
6      int bg; /* should the job run in bg or fg? */
7      pid_t pid; /* process id */
8
9      strcpy(buf, cmdline);
10     bg = parseline(buf, argv);
11     if (argv[0] == NULL)
12         return; /* ignore empty lines */
13 }
```

# Parsing command line

---

- Parse the space separated command-line arguments
- Builds the `argv` vector
  - which will eventually be passed to `execve`

```
1 /* parse the cmd line and build the argv array */
2 int parseline(const char *cmdline, char **argv)
3 {
4     char *delim; /* first space delimiter */
5     int argc; /* number of args */
6     int bg; /* background job? */
7
8     buf[strlen(buf)-1] = ' ';
9     /* replace trailing '\n' with space */
10    while (*buf && (*buf == ' ')) /* ignore spaces */
11        buf++;
12
13
```

```
12  /* build the argv list */
13  argc = 0;
14  while ((delim = strchr(buf, ' '))) {
15      argv[argc++] = buf;
16      *delim = '\0'; /* replace space with '\0'*/
17      buf = delim + 1;
18      while (*buf && (*buf == ' ')) /* ignore spaces */
19          buf++;
20  }
21  argv[argc] = NULL; /* set the end of argv list */
22
23  if (argc == 0) /* ignore blank line */
24      return 1;
25
26  /* should the job run in the background? */
27  if ((bg = (*argv[argc-1] == '&')) != 0)
28      argv[--argc] = NULL;
29  return bg;
30 }
```

# Parsing command line

---

- The first argument is assumed to be
  - either the name of a built-in shell command
    - that is interpreted immediately
  - or an executable object file
    - that will be loaded and run in the context of a new child process

```
14  if (!builtin_command(argv)) {
15      if ((pid = Fork()) == 0) { /* execute in child */
16          if (execve(argv[0], argv, environ) < 0) {
17              printf("%s: Command not found.\n", argv[0]);
18              exit(0);
19      }
20  }
```

```
14 if (!builtin_command(argv)) {
15     if ((pid = Fork()) == 0) { /* execute in child */
16         if (execve(argv[0], argv, environ) < 0) {
17             printf("%s: Command not found.\n", argv[0]);
18             exit(0);
19         }
20     }
21 }
```

---

```
34 /* if 1st arg is a builtin command,
   run it and return true */
35 int builtin_command(char **argv)
36 {
37     if (!strcmp(argv[0], "quit")) /* quit command */
38         exit(0);
39     if (!strcmp(argv[0], "&")) /* ignore singleton & */
40         return 1;
41     return 0; /* not a builtin command */
42 }
```

# Foreground and Background

---

- If the last argument is a "&" character
  - parseline returns 1
  - indicating the program should be executed in the background  
(the shell returns to the top of the loop and waits for the next command)
- Otherwise
  - parseline returns 0
  - indicating the program should be run in the foreground  
(the shell waits for it to complete)

```
14     if (!builtin_command(argv)) {
15         if ((pid = Fork()) == 0) { /* execute in child */
16             if (execve(argv[0], argv, environ) < 0) {
17                 printf("%s: Command not found.\n", argv[0]);
18                 exit(0);
19             }
20         }
21     }
22
23     /* parent waits for foreground job to terminate */
24     if (!bg) { /* foreground */
25         int status;
26         if (waitpid(pid, &status, 0) < 0)
27             unix_error("waitfg: waitpid error");
28     }
29     else
30         printf("%d %s", pid, cmdline);
31
32 }
```

# Exceptional Control Flow IV

# Outline

---

- Signals
  - Signal Terminology
  - Sending Signals
  - Receiving Signals
- Suggested reading 8.5.1~8.5.3

# Signals

---

- Unix signal
  - A higher-level software form of exception
  - That allows processes and the kernel to interrupt other processes

# The kill Program

---

- Kill program
  - is located in directory `/bin/`
  - sends an arbitrary signal to another process

e.g. `unix> kill -9 15213`

- sends signal 9 (SIGKILL) to process 15213.

# Signals Type

---

- Signal is a message that notifies a process
  - an event of some type has occurred in the system.
- Each type corresponds to some kind of system event
  - Low-level **hardware** exceptions
    - are processed by the kernel's exception handlers would not normally be visible to user processes.
    - Signals provide a mechanism for exposing the occurrence of such exceptions to user processes.
  - Higher-level **software** events
    - in the kernel
    - in other user processes

# Hardware Events

---

- SIGFPE signal (number 8)
  - If a process attempts to divide by zero, then the kernel sends it a SIGFPE signal
- SIGILL signal (number 4)
  - If a process executes an illegal instruction, the kernel sends it a SIGILL signal.
- SIGSEGV signal (number 11)
  - If a process makes an illegal memory reference, the kernel sends it a SIGSEGV signal.

# Software Events

---

- SIGINT signal (number 2)
  - While a process is running in the foreground
  - if you type a ctrl-c
  - then the kernel sends a SIGINT signal to the process
- SIGKILL signal (number 9)
  - A process can forcibly terminate another process
  - by sending it a SIGKILL signal
- SIGCHLD signal (number 17)
  - When a child process terminates or stops,
  - the kernel sends a SIGCHLD signal to the parent.

# Linux Signals

---

> **man signal**

Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core (1)	Trace trap
6	SIGABRT	Terminate and dump core (1)	Abort signal from abort function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core (1)	Floating point exception
9	SIGKILL	Terminate (2)	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core (1)	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from alarm function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT (2)	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal

# Signal Terminology

---

- Two steps to transfer a signal to a destination process
  - Sending a signal
  - Receiving a signal

# Sending a signal

---

- The kernel
  - **sends (delivers)** a signal to a destination process
    - by updating some state in the context of the destination process.
- The signal is delivered for one of two reasons
  - The kernel has detected a system event
    - Divide by zero, termination of a child process
  - A process has invoked the **kill** function
    - to explicitly request the kernel to send a signal to the destination process.
    - A process can send a signal to itself.

# Receiving a signal

---

- A destination process **receives** a signal
  - when it is forced by the kernel to **react** in some way to the delivery of the signal
- The process can
  - Ignore the signal
  - Terminate
  - Catch the signal
    - by executing a user-level function called a **signal handler**

# Pending Signal

---

- Pending signal
  - A signal that has been sent but not yet received
- Only one
  - At any point in time,
  - there can be at most one pending signal of a particular type
- Not queued
  - If a process has a pending signal of type k
  - then any subsequent signals of type k sent to that process are not queued
  - they are simply discarded.
- A pending signal is received at most once.

# Blocking a Signal

---

- A process can selectively **block** the receipt of certain signals.
- When a signal is blocked
  - it can be delivered
  - but the resulting pending signal will not be received
  - until the process unblocks the signal

# Internal Data Structures

---

- For each process, the kernel maintains
  - the set of pending signals in the **pending bit vector**
  - the set of blocked signals in the **blocked bit vector**
- The kernel sets bit  $k$  in **pending**
  - whenever a signal of type  $k$  is delivered
- The kernel clears bit  $k$  in **pending**
  - whenever a signal of type  $k$  is received.

# Process Groups

---

- To send signals to processes
  - Unix systems provide a number of mechanisms for sending signals to processes
  - All of them rely on the notion of a **process group**
- Every process belongs to exactly one process group
  - which is identified by a positive integer (process group ID).
- By default, a child process belongs to the same process group as its parent.

# Process Groups

```
#include <unistd.h>
pid_t getpgrp(void) ;
                                returns: process group ID of calling process
#include <unistd.h>
pid_t setpgid(pid_t pid, pid_t pgid) ;
                                returns: 0 on success, -1 on error
```

- Changes the process group of process `pid` to `pgid`
- If `pid` is zero, the PID of the current process is used
- If `pgid` is zero, the PID of the process specified by `pid` is used for the process group ID

# setpgid Function

---

e.g. **setpgid(0, 0);**

- Suppose process 15213 is the calling process,
- then this function call
  - creates a new process group whose process group ID is 15213
  - adds process 15213 to this new group.

# The kill Program

---

- Kill program
  - is located in directory `/bin/`
  - sends an arbitrary signal to another process

e.g. `unix> kill -9 15213`

- sends signal 9 (SIGKILL) to process 15213.

# The kill Program

---

- Sending signals to process group by kill
  - Using a negative PID
  - Which causes the signal to be sent to every process in process group PID

e.g. unix> kill -9 **-15213**

- sends a SIGKILL signal to every process in process group 15213.

# Sending Signals from the Keyboard

---

- Job
    - An abstraction used by Unix shells
    - To represent the processes
      - that are created as a result of evaluating a single cmdline
    - At any point, there is at most one foreground job and zero or more background jobs
- e.g. `unix> ls | sort`
- a foreground job consisting of two processes connected by a pipe

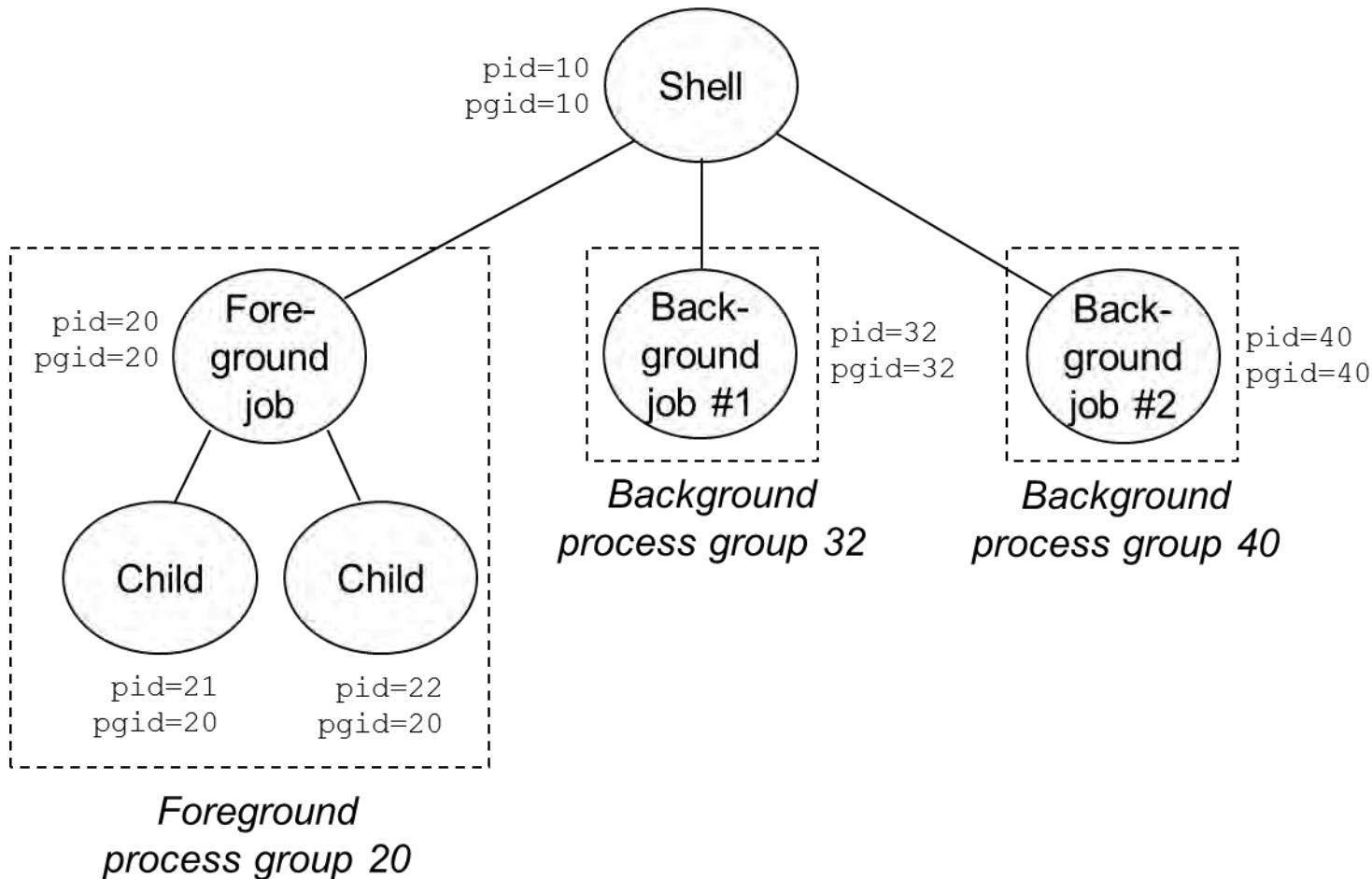
# Sending Signals from the Keyboard

---

- The shell creates a separate process group for each job
  - Typically, the process group ID is taken from one of the parent processes in the job

# Sending Signals from the Keyboard

---



# Sending Signals from the Keyboard

---

- Typing **ctrl-c** at the keyboard
  - causes a SIGINT signal to be sent to the shell
  - The shell catches the signal
  - Then sends a SIGINT to every process in the foreground process group
  - The result is to terminate the foreground job (default)

# Sending Signals from the Keyboard

---

- Typing `crtl-z`
  - sends a SIGTSTP signal to the shell
  - The shell catches the signal
  - Sends a SIGTSTP signal to every process in the foreground process group
  - The result is to stop (suspend) the foreground job (default)

# Sending Signals with kill Function

---

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

returns: 0 if OK, -1 on error

- If **pid** is greater than zero,
  - then the **kill** function sends signal number **sig** to process **pid**
- If **pid** is less than zero
  - then **kill** sends signal **sig** to every process in process group **abs(pid)**

```
1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6
7     /* child sleeps until SIGKILL signal received */
8     if ((pid = Fork()) == 0) {
9         Pause(); /* wait for a signal to arrive */
10        printf("control should never reach here!\n");
11        exit(0);
12    }
13
14    /* parent sends a SIGKILL signal to a child */
15    Kill(pid, SIGKILL);
16    exit(0);
17 }
```

# Receiving a Signal

---

- Preconditions
  - When the kernel is returning from an exception handler
  - and is ready to pass control to process p

# Receiving a Signal

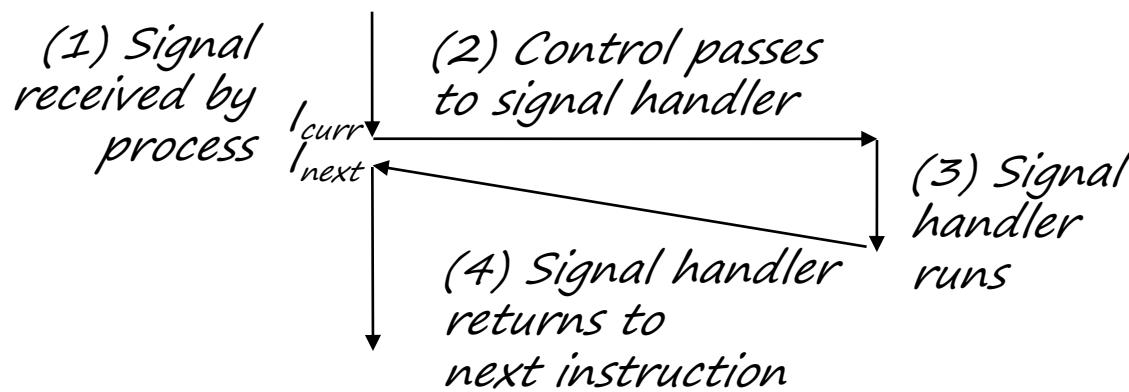
---

- Actions
  - kernel checks the set of unblocked pending signals (**pending** & **~blocked**)
  - If this set is empty (the usual case)
    - then the kernel passes control to the next instruction ( $I_{next}$ ) in the logical control flow of p.
  - However, if the set is nonempty
    - then the kernel chooses some signal k in the set (typically the smallest k)
    - and forces p to receive signal k.

# Receiving a Signal

---

- The receipt of the signal triggers some action by the process.
- Once the process completes the action, then control passes back to the next instruction ( $I_{next}$ ) in the logical control flow of  $p$ .



# Receiving a Signal

---

- Each signal type has a predefined default action, which is one of the following:
  - The process terminates.
  - The process terminates and dumps core.
  - The process stops until restarted by a SIGCONT signal.
  - The process ignores the signal.

# Receiving a Signal

---

- SIGKILL
  - The default action is to terminate the receiving process
- SIGCHLD
  - The default action is to ignore the signal.

# Receiving a Signal

---

- A process can modify the default action associated with a signal
  - by using the `signal` function
  - The only exceptions are `SIGSTOP` and `SIGKILL`, whose default actions cannot be changed.

# Signal Function

```
#include <signal.h>
typedef void handler_t(int)
handler_t *signal(int signum, handler_t *handler)
```

returns: ptr to previous handler if OK,  
`SIG_ERR` on error (does not set `errno`)

- Three ways to change default actions:
  - If `handler` is `SIG_IGN`, then signals of type `signum` are ignored.
  - If `handler` is `SIG_DFL`, then the action for signals of type `signum` reverts to the default action.
  - Otherwise, change action to `handler` (called signal handler)

# Signal Function

---

- Installing the handler
  - Changing the default action by passing the address of a handler to the `signal` function
- Catching the signal
  - The invocation of the handler
- Handling the signal
  - The execution of the handler

```
1 #include "csapp.h"
2
3 void handler(int sig) /* SIGINT handler */
4 {
5     printf("Caught SIGINT\n");
6     exit(0);
7 }
8
9 int main()
10 {
11     /* Install the SIGINT handler */
12     if (signal(SIGINT, handler) == SIG_ERR)
13         unix_error("signal error");
14
15     pause(); /* wait for the receipt of a signal */
16
17     exit(0);
18 }
```

# Signal Function

---

- Handling the signal
  - When a process catches a signal of type k,
  - the handler installed for signal k is invoked
    - with a single integer argument set to k.
  - When the handler executes its **return** statement,
    - control (usually) passes back to the instruction in the control flow
      - where the process was interrupted by the receipt of the signal.
      - in some systems, interrupted system calls return immediately with an error.

# Signal Function

---

- The same handler function can be used to catch different types of signals
  - By setting different signal numbers as the argument to the signal handler

# Sending Signals With the `alarm` Function

---

```
#include <unistd.h>
unsigned int alarm(unsigned int secs);
```

returns: remaining secs of previous alarm, or 0 if no previous alarm

- Arranges for the kernel to send a `SIGALRM` signal to the calling process in `secs` seconds.
- If `secs` is zero
  - then no new alarm is scheduled.
- The call to `alarm`
  - cancels any pending alarms
  - returns the number of seconds remaining until any pending alarm was due to be delivered
  - returns 0 if there were no pending alarms.

```
1 #include "csapp.h"
2 void handler(int sig)
3 {
4     static int beeps = 0;
5     printf("BEEP\n");
6     if (++beeps < 5)
7         Alarm(1); /* next SIGALRM will be delivered in 1s */
8     else {
9         printf("BOOM!\n");
10        exit(0);
11    }
12 }
13 int main()
14 {
15     Signal(SIGALRM, handler); /* install SIGALRM handler */
16     Alarm(1); /* next SIGALRM will be delivered in 1s */
17
18     /* signal handler returns control here each time */
19     while (1);
20     exit(0);
21 }
```

```
linux> ./alarm
```

BEEP

BEEP

BEEP

BEEP

BEEP

BOOM!

# Exceptional Control Flow VI

# Outline

---

- Signal
  - Signal Handling Issues
  - Portable Signal Handling
  - Explicitly Blocking Signals
  - Synchronizing Flows to Avoid Nasty Concurrency Bugs
- Nonlocal Jump
- Suggested reading 8.5, 8.6

# Signal Handling Issues

---

- Pending signals can be blocked
  - Unix signal handlers typically block pending signals of the type currently being processed by the handler
- Pending signals are not queued
  - There can be at most one pending signal of any particular type
  - then the second same type of pending signal is simply discarded

# Signal Handling Issues

---

- System calls can be interrupted
  - Slow system calls
    - `read`, `wait`, and `accept`
  - On some systems
    - slow system calls that are interrupted when a handler catches a signal do not resume when the signal handler returns
    - but instead return immediately to the user with an error condition and `errno` set to `EINTR`

```
14 int main()
15 {
16     int i, n;    char buf[MAXBUF];
17     if (signal(SIGCHLD, handler1) == SIG_ERR)
18         unix_error("signal error");
19
20     /* parent creates children */
21     for (i = 0; i < 3; i++) {
22         if (Fork() == 0) {
23             printf("Hello from child %d\n", (int)getpid());
24             Sleep(1);
25             exit(0);
26         }
27     }
28     /* parent waits for term. input and then processes it */
29     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
30         unix_error("read");
31
32     printf("Parent processing input\n");
33     while (1) ;
34
35     exit(0);
36 }
```

```
1 #include "csapp.h"
2
3 void handler1(int sig)
4 {
5     pid_t pid;
6
7     if ((pid = waitpid(-1, NULL, 0)) < 0)
8         unix_error("waitpid error");
9     printf("Handler reaped child %d\n", (int)pid);
10    Sleep(2);
11    return;
12 }
13
```

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
<cr>
Parent processing input
<ctrl-z>
Suspended
```

```
linux> ps
PID TTY STAT TIME COMMAND
...
10319 p5 T 0:03 signal1
10321 p5 Z 0:00 signal1 <zombie>
10323 p5 R 0:00 ps
```

```
1 #include "csapp.h"
2
3 void handler2(int sig)
4 {
5     pid_t pid;
6
7     while ((pid = waitpid(-1, NULL, 0)) > 0)
8         printf("Handler reaped child %d\n", (int)pid);
9     if (errno != ECHILD)
10        unix_error("waitpid error");
11     Sleep(2);
12     return;
13 }
```

linux> **./signal2**

Hello from child 10378  
Hello from child 10379  
Hello from child 10380  
Handler reaped child 10379  
Handler reaped child 10378  
Handler reaped child 10380  
<cr>  
Parent processing input

solaris> **./signal2**

Hello from child 18906  
Hello from child 18907  
Hello from child 18908  
Handler reaped child 18906  
Handler reaped child 18908  
Handler reaped child 18907  
**read: Interrupted system call**

```
15 int main()
16 {
17     int i, n;    char buf[MAXBUF];
18     if (signal(SIGCHLD, handler2) == SIG_ERR)
19         unix_error("signal error");
20
21     /* parent creates children */
22     for (i = 0; i < 3; i++) {
23         if (Fork() == 0) {
24             printf("Hello from child %d\n", (int)getpid());
25             Sleep(1);
26             exit(0);
27         }
28     }
29     /* Manually restart the read if it is interrupted */
30     while ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
31         if (errno != EINTR)
32             unix_error("read");
33
34     printf("Parent processing input\n");
35     while (1) ;
36     exit(0);
37 }
```

```
solaris> ./signals3
```

```
Hello from child 19571
```

```
Hello from child 19572
```

```
Hello from child 19573
```

```
Handler reaped child 19571
```

```
Handler reaped child 19572
```

```
Handler reaped child 19573
```

**<cr>**

Parent processing input

# sigaction Function

---

```
#include <signal.h>

int sigaction(int signum,
              struct sigaction *act,
              struct sigaction *oldact);
                                         returns: 0 if OK, -1 on error

struct sigaction {
    void (*sa_handler)();           /* addr of signal handler,
                                     or SIG_IGN, or SIG_DFL */
    sigset_t sa_mask; /* additional signals to block */
    int sa_flags;                /* signal options */
} ;
```

```
1 handler_t *Signal(int signum, handler_t *handler)
2 {
3     struct sigaction action, old_action;
4
5     action.sa_handler = handler;
6     /* block sigs of type being handled */
7     sigemptyset(&action.sa_mask);
8     /* restart syscalls if possible */
9     action.sa_flags = SA_RESTART;
10
11    if (sigaction(signum, &action, &old_action) < 0)
12        unix_error("Signal error");
13    return (old_action.sa_handler);
14 }
```

# Signal Function

---

- Only signals of the type currently being processed by the handler are blocked
- As with all signal implementations, signals are not queued
- Interrupted system calls are automatically restarted whenever possible

# Signal Function

---

- Once the signal handler is installed, it remains installed until `Signal` is called with a `handler` argument of either `SIG_IGN` or `SIG_DFL`

# Explicitly Blocking Signals

---

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set,
                 sigset_t *oldset)
int sigemptyset(sigset_t *set) ;
int sigfillset(sigset_t *set) ;
int sigaddset(sigset_t *set, int signum) ;
int sigdelset(sigset_t *set, int signum) ;

                                         Return: 0 if OK, -1 on error

Int sigismember(const sigset_t *set, int signum) ;
                                         Returns: 1 if member, 0 if not, -1 on error
```

# Explicitly Blocking Signals

---

- `sigprocmask`
  - Changes the set of currently blocked signals
  - `SIG_BLOCK`: add the signals in set to blocked  
(`blocked = blocked | set`)
  - `SIG_UNBLOCK`: Remove the signals in set from blocked  
(`blocked = blocked & ~set`)
  - `SIG_SETMASK`: `blocked = set`
  - If `oldset` is non-NULL, the previous value of the blocked bit vector is stored in `oldset`

# Explicitly Blocking Signals

---

- Signal set
  - Set manipulations are provided by the rest of the functions
    - `sigemptyset`
    - `sigfillset`
    - `sigaddset`
    - `sigdelset`

# Nasty Concurrency Bugs

---

```
10 int main(int argc, char **argv)
11 {
12     int pid;
14     Signal(SIGCHLD, handler);
15     initjobs(); /* initialize the job list */
16
17     while(1) {
18         /*child process */
19         if ((pid = Fork()) == 0)
20             Execve("/bin/ls", argv, NULL);
21
22         /* parent process */
23         addjob(pid); /* Add the child to the job list */
24     }
25     exit(0)
26 }
```

# Nasty Concurrency Bugs

---

```
1 void handler(int sig)
2 {
3     pid_t pid ;
4     /*reap a zombie child */
5     while ( (pid = waitpid(-1, NULL, 0) > 0)) > 0)
6         /* delete the child from the job list */
7         deletejob(pid) ;
8
9 }
```

# Nasty Concurrency Bugs

---

- Incorrectly sequence of events
  - Parent executes `fork` and kernel schedules the newly created child
  - Child terminates and kernel delivers a `SIGCHLD` signal to parent
  - Parent receives the `SIGCHLD` signal and runs the signal handler
  - Signal handler calls `deletejob`
  - Parent return from `fork` and calls `addjob`

```
1 void handler(int sig)
2 {
3     pid_t pid ;
4     /*reap a zombie child */
5     while ( (pid = waitpid(-1, NULL, 0) > 0)) > 0)
6         deletejob(pid); /* delete the child from the job list */
7     if (errno != ECHILD)
8         unix_error("waitpid error") ;
9
10 int main(int argc, char **argv)
11 {
12     int pid;
13     sigset_t mask;
14
15     Signal(SIGCHLD, handler);
16     initjobs(); /* initialize the job list */
17
```

```
18     while(1) {  
19         Sigemptyset(&mask);  
20         Sigaddset(&mask, SIGCHLD);  
21         Sigprocmask(SIG_BLOCK, &mask, NULL);  
22                                         Block SIGCHLD  
23         /*child process */  
24         if ((pid = Fork()) == 0) {  
25             Sigprocmask(SIG_UNBLOCK, &mask, NULL);  
26             Execve("/bin/ls", argv, NULL);  
27         }  
28  
29         /* parent process */  
30         addjob(pid); /* Add the child to the job list */  
31         Sigprocmask(SIG_UNBLOCK, &mask, NULL);  
32     }  
33     exit(0)  
34 }
```

# Nonlocal Jumps

# Nonlocal Jumps

---

- Nonlocal Jumps
  - transfers control directly from one function to another currently executing function
  - without having to go through the normal call-and-return sequence
  - C provides a form of user-level exceptional control flow

# Nonlocal Jumps

---

- Nonlocal jumps are provided by
  - the `setjmp` function
    - Saves the current stack context in the `env` buffer, for later use by `longjmp`, returns 0
  - the `longjmp` function
    - restores the stack context from the `env` buffer
    - triggers a return from the most recent `setjmp` call that initialized `env`
    - The `setjmp` then returns with the nonzero return value `retval`.

# Nonlocal Jumps

---

```
#include <setjmp.h>
```

```
int setjmp(jmp_buf env) ;
```

Returns: 0 from setjmp,  
nonzero from long jumps

```
#include <setjmp.h>
```

```
void longjmp(jmp_buf env, int retval);
```

Never returns

```
1 #include "csapp.h"
2
3 jmp_buf buf;
4
5 int error1 = 0;
6 int error2 = 1;
7
8 void foo(void) , bar(void) ;
9
10 int main()
11 {
12     int rc;
13
14     rc = setjmp(buf);
15     if (rc == 0)
16         foo();
17     else if (rc == 1)
18         printf("Detected an error1 condition in foo\n");
19     else if (rc == 2)
20         printf("Detected an error2 condition in foo\n");
21     else
22         printf("Unknown error condition in foo\n");
23     exit(0);
24 }
```

```
25
26 /* deeply nested function foo */
27 void foo(void)
28 {
29     if (error1)
30         longjmp(buf, 1);
31     bar();
32 }
33
34 void bar(void)
35 {
36     if (error2)
37         longjmp(buf, 2);
38 }
```

# Nonlocal Jumps

---

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savesigs);
>Returns: 0 from sigsetjmp,
nonzero from siglongjumps

#include <setjmp.h>
void siglongjmp(sigjmp_buf env, int retval);
>Never returns
```

```
1 #include "csapp.h"
2 sigjmp_buf buf;
3
4 void handler(int sig) { siglongjmp(buf, 1); }
5
6 int main()
7 {
8     Signal(SIGINT, handler);
9
10    if (!sigsetjmp(buf, 1))
11        printf("starting\n");
12    else
13        printf("restarting\n");
14
15    while(1) {
16        Sleep(1);
17        printf("processing...\\n");
18    }
19    exit(0);
20 }
```

# System-Level I/O

# Outline

---

- Unix I/O
- Reading File Metadata
- Sharing Files & I/O redirection
- Robust I/O
- Standard I/O
- Suggested Reading
  - 10.1~10.3, 10.5~10.7, 10.4, 10.8

# Why Unix I/O

---

- Input/Output
  - A process of copying data between main memory and external devices
- Standard I/O library
  - User level or high level I/O functions

# Why Unix I/O

---

- Unix I/O
  - I/O functions provided by kernel
  - Helps you understand other system concepts
    - I/O vs. process
  - Sometimes, it is the only choice

# Unix I/O

---

- A Unix file is a sequence of  $m$  bytes
  - $B_0, B_1, \dots, B_k, \dots, B_m$
- All I/O devices are modeled as files
  - such as networks, disks, and terminals
  - allows Unix to export a simple, low-level application interface, known as Unix I/O,

# Unix I/O

---

- All input and output is performed by reading and writing the appropriate files
  - Enables all input and output to be performed in a uniform and consistent way.

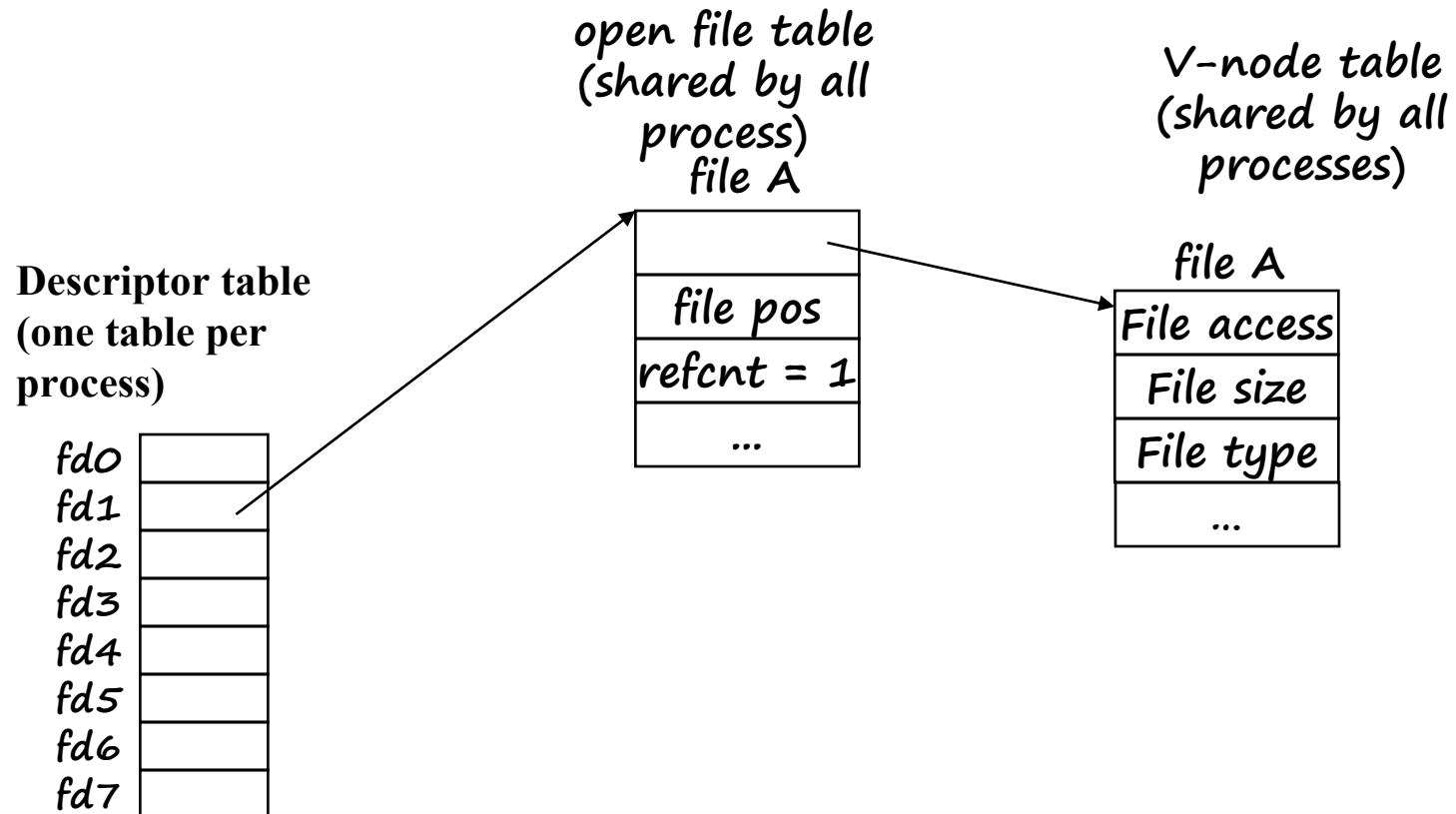
# Open Files

---

- An application announces its intention to access an I/O device
  - The kernel opens the corresponding file
  - The kernel returns a small non-negative integer, descriptor
    - Identifies the file in all subsequent operations on the file
  - The kernel keeps track of all information about the open file
    - Maintains a file position  $k$ , initially 0, for each open file

# Kernel Data Structures for Files

---



# Descriptor table

---

- Each process has its own separate descriptor table
  - whose entries are indexed by the process's open file descriptors
  - each open descriptor entry points to an entry in the file table

# File table

---

- The set of open files is represented by a file table
- File table is shared by all processes
- Each file table entry consists
  - the current file position
  - a reference count of the number of descriptor entries that currently point to it
  - a pointer to an entry in the v-node table
  - the kernel deletes a file table entry when its reference count becomes zero

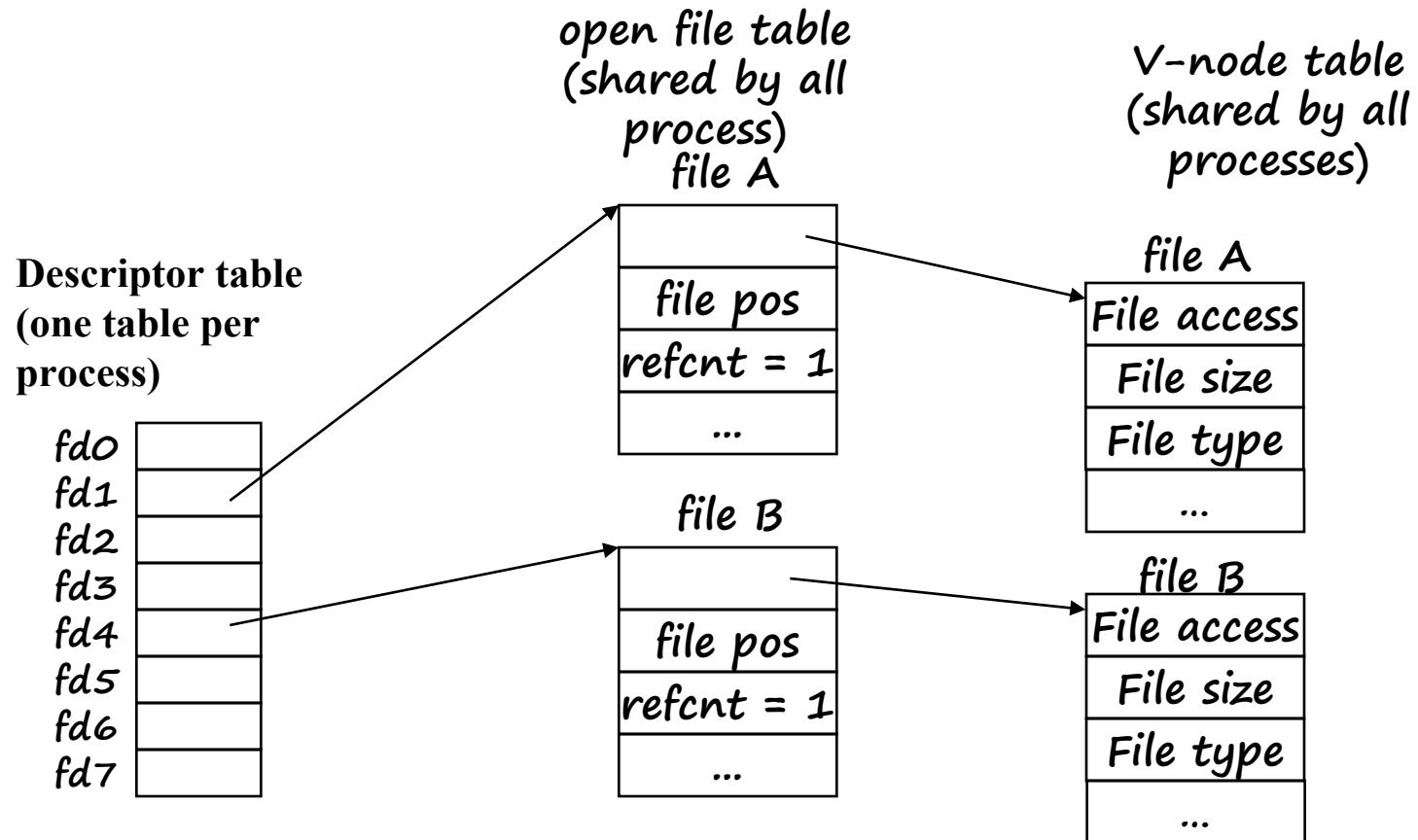
## V-node table

---

- V-node table is shared by all processes
- Each entry contains most of the information of a file

# Kernel Data Structures for Files

---



# Open Files

---

- An application announces its intention to access an I/O device
  - The application keeps track of only the descriptor
  - An application can set the current file position explicitly by performing a `seek` operation

# Open Files

---

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(char *filename, int flags, mode_t mode);
```

Returns: new file descriptor if OK, -1 on error

## flags

- O\_RDONLY, O\_WRONLY, O\_RDWR (must have one)
- O\_CREAT, O\_TRUNC, O\_APPEND (optional)

## mode

- S\_IRUSR, S\_IWUSR, S\_IXUSR
- S\_IRGRP, S\_IWGRP, S\_IXGRP
- S\_IROTH, S\_IWOTH, S\_IXOTH

# Open Files

---

`umask()`: set mask of process

```
#define DEF_MODE S_IRUSR | S_IWUSR | \
                 S_IRGRP | S_IWGRP | \
                 S_IROTH | S_IWOTH
#define DEF_UMASK S_IWGRP | S_IWOTH
```

```
umask(DEF_UMASK);
fd = open ("foot.txt",
           O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
```

# Close Files

---

- close the file
  - Do not access the file
  - The kernel action
    - The kernel frees the structures it created when the file was opened
    - The kernel restores the descriptor to a pool of available descriptors
      - The next file that is opened is guaranteed to receive the smallest available descriptor in the pool

# Close Files

---

- close the file
  - Default actions when terminate
    - The kernel closes all open files
    - The kernel frees their memory resources

```
#include <unistd.h>
```

```
int close(int fd) ;
```

Returns: zero if OK, -1 on error

# Reading and Writing Files

---

- A read operation
  - Copies  $m > 0$  bytes from the file to memory
    - starting at the current file position  $k$
    - incrementing  $k$  by  $m$
  - If the total bytes from  $k$  to the end of file is less than  $m$ , it triggers a condition known as end-of-file (EOF)
    - Can be detected by the application
    - There is no explicit "EOF character" at the end of a file

# Reading and Writing Files

---

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

    returns: number of bytes read if OK,
              0 on EOF, -1 on error

ssize_t write(int fd, const void *buf, size_t count);

    returns: number of bytes written if OK,
              -1 on error
```

# Reading and Writing Files

---

```
1 #include "csapp.h"
2
3 int main(void)
4 {
5     char c;
6
7     /* copy stdin to stdout, one byte at a time */
8     while(Read(STDIN_FILENO, &c, 1) != 0)
9         Write(STDOUT_FILENO, &c, 1);
10    exit(0);
11 }
```

STDIN\_FILENO(0), STDOUT\_FILENO(1), STDERR\_FILENO(2)

# Reading and Writing Files

---

- `ssize_t` vs. `size_t`
- Short count
  - Read and write transfer fewer bytes than the application requests
    - Encounter EOF on reads
    - Reading text lines from a terminal
    - Reading and writing network sockets

# Reading File Metadata

---

```
#include <unistd.h>

#include <sys/stat.h>

int stat(const char *filename, struct stat *buf);

int fstat(int fd, struct stat *buf) ;

returns: 0 if OK, -1 on error
```

# Reading File Metadata

---

```
/* file info returned by the stat function */
struct stat {
    dev_t st_dev;                      /* device */
    ino_t st_ino;                      /* inode */
    mode_t st_mode;                    /* protection and file type */
    nlink_t st_nlink;                 /* number of hard links */
    uid_t st_uid;                     /* user ID of owner */
    gid_t st_gid;                     /* group ID of owner */
    dev_t st_rdev;                    /* device type (if inode device) */
    off_t st_size;                    /* total size, in bytes */
    unsigned long st_blksize;         /* blocksize for filesystem I/O */
    unsigned long st_blocks;          /* number of blocks allocated */
    time_t st_atime;                  /* time of last access */
    time_t st_mtime;                  /* time of last modification */
    time_t st_ctime;                  /* time of last change */
};
```

```
int main(int argc, char **argv)
{
    struct stat stat ;
    char *type, *readok ;
    Stat(argv[1], &stat) ;
    if (_S_ISREG(stat.st_mode)) /* Determine file type */
        type = "regular" ;
    else if (_S_ISDIR(stat.st_mode))
        type = "directory" ;
    else
        type = "other" ;

    if ((stat.mode & S_IRUSR)) /* check read access */
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok) ;
    exit(0) ;
}
```

`S_ISREG()` Is this a regular file?  
`S_ISDIR()` Is this a directory file?

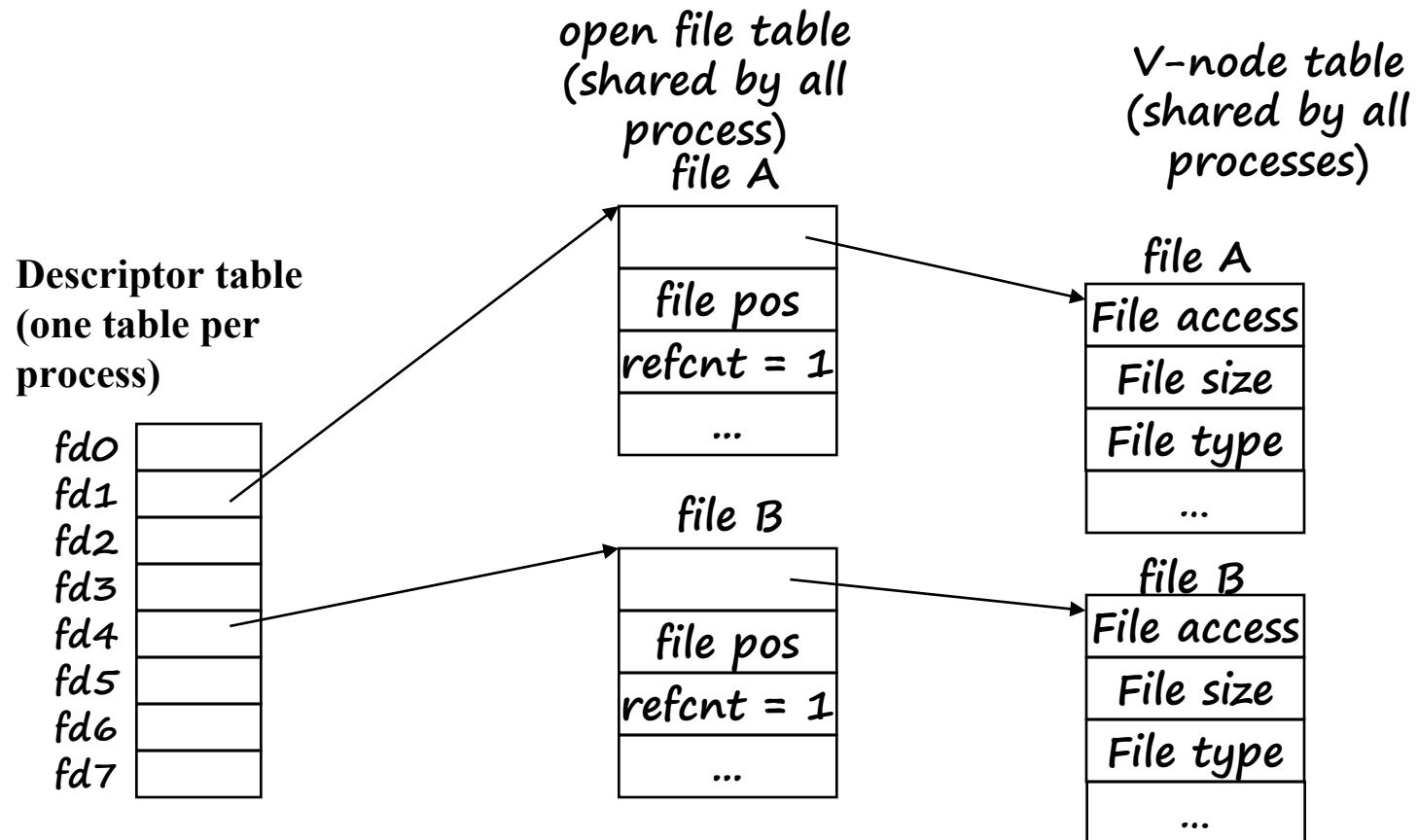
# V-Node table

---

- Shared by all processes
- Each entry contains most of the information in the stat structure, such as
  - st\_mode
  - st\_size

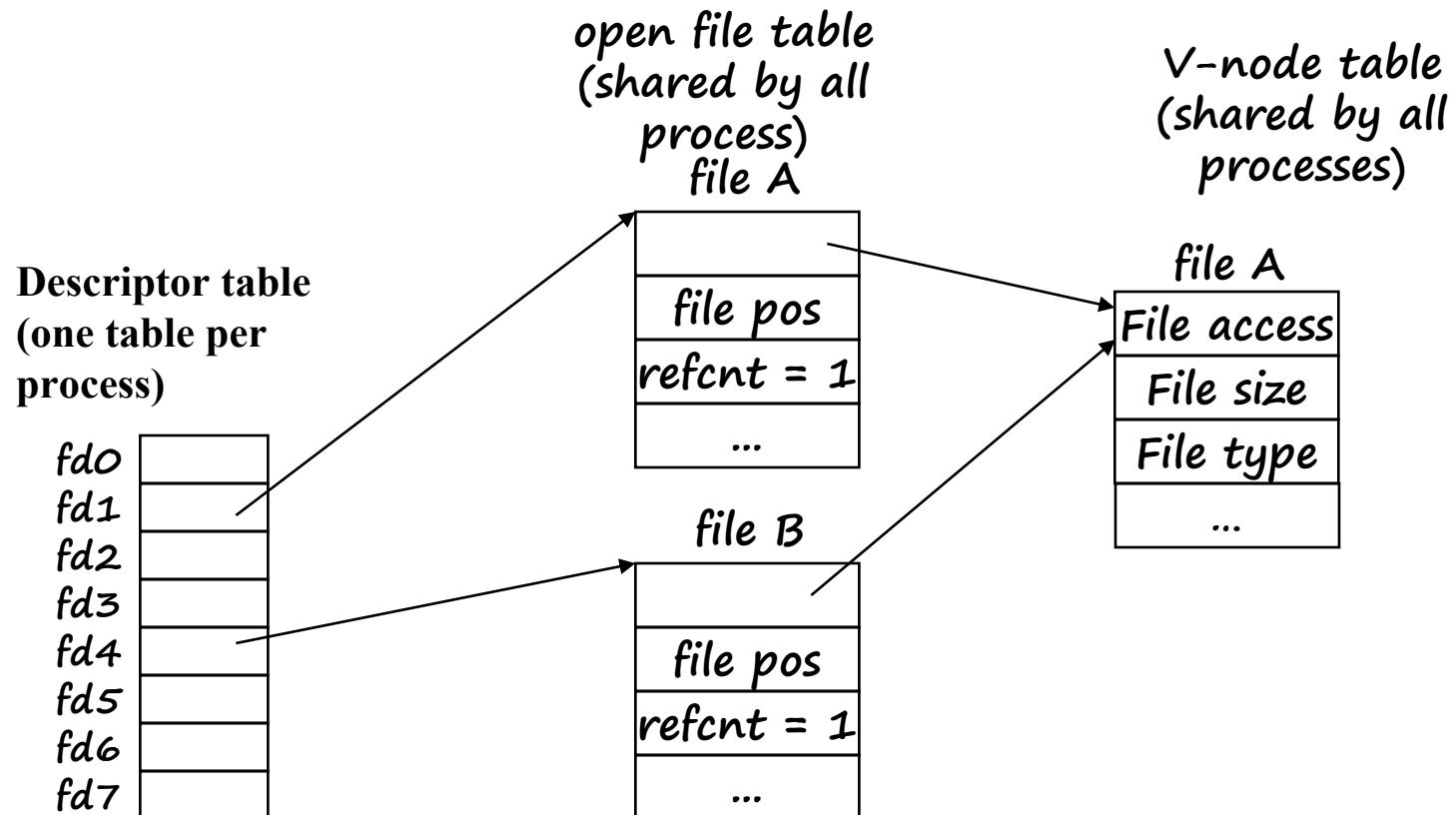
# Sharing Files

---



# Sharing Files

---



# Sharing Files

---

```
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;

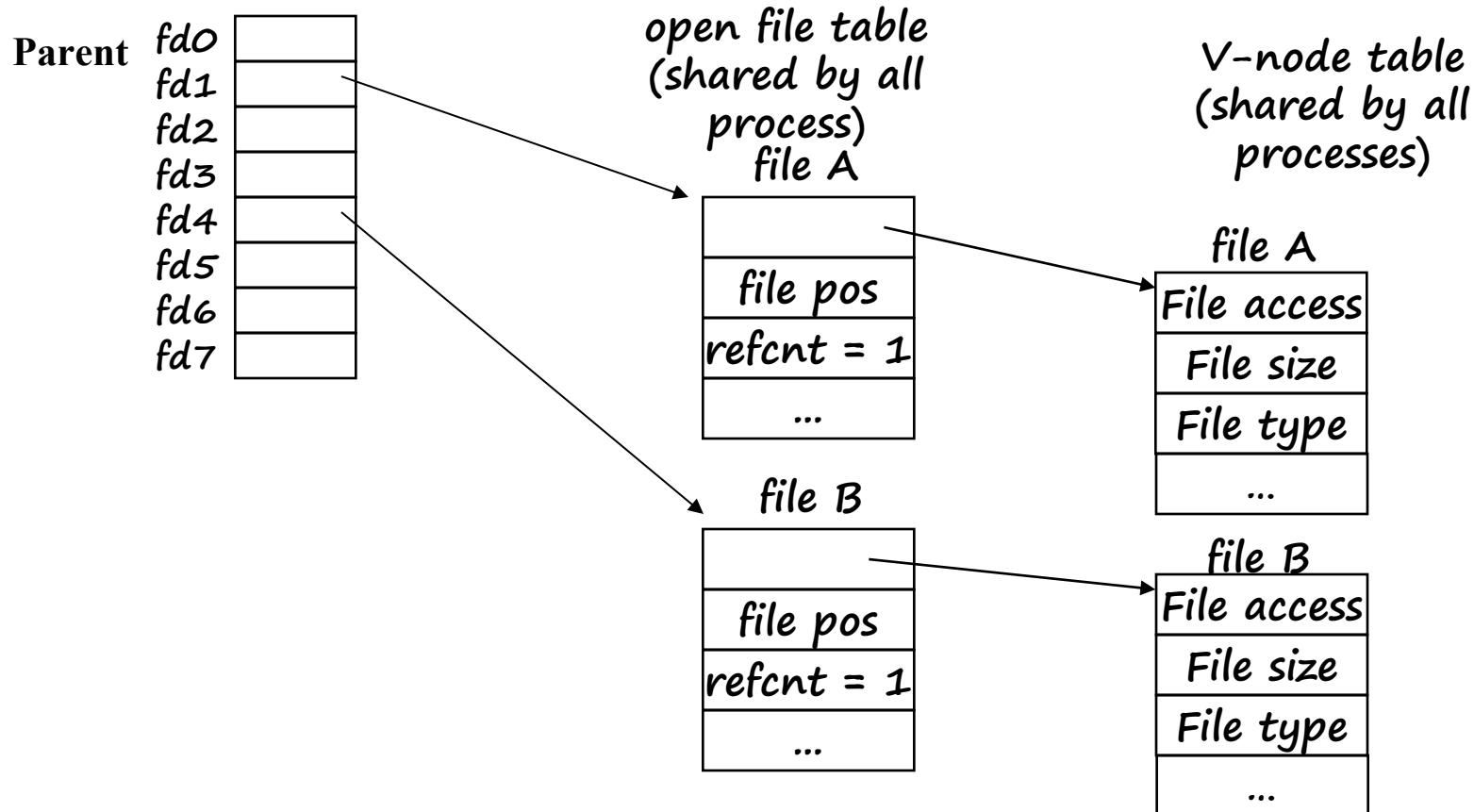
    fd1 = open("foobar.txt", O_RDONLY, 0) ;
    fd2 = open("foobar.txt", O_RDONLY, 0) ;
    read(fd1, &c, 1) ;
    read(fd2, &c, 1) ;
    printf("c = %c\n", c) ;
    exit(0)
}
```

**foobar.txt**

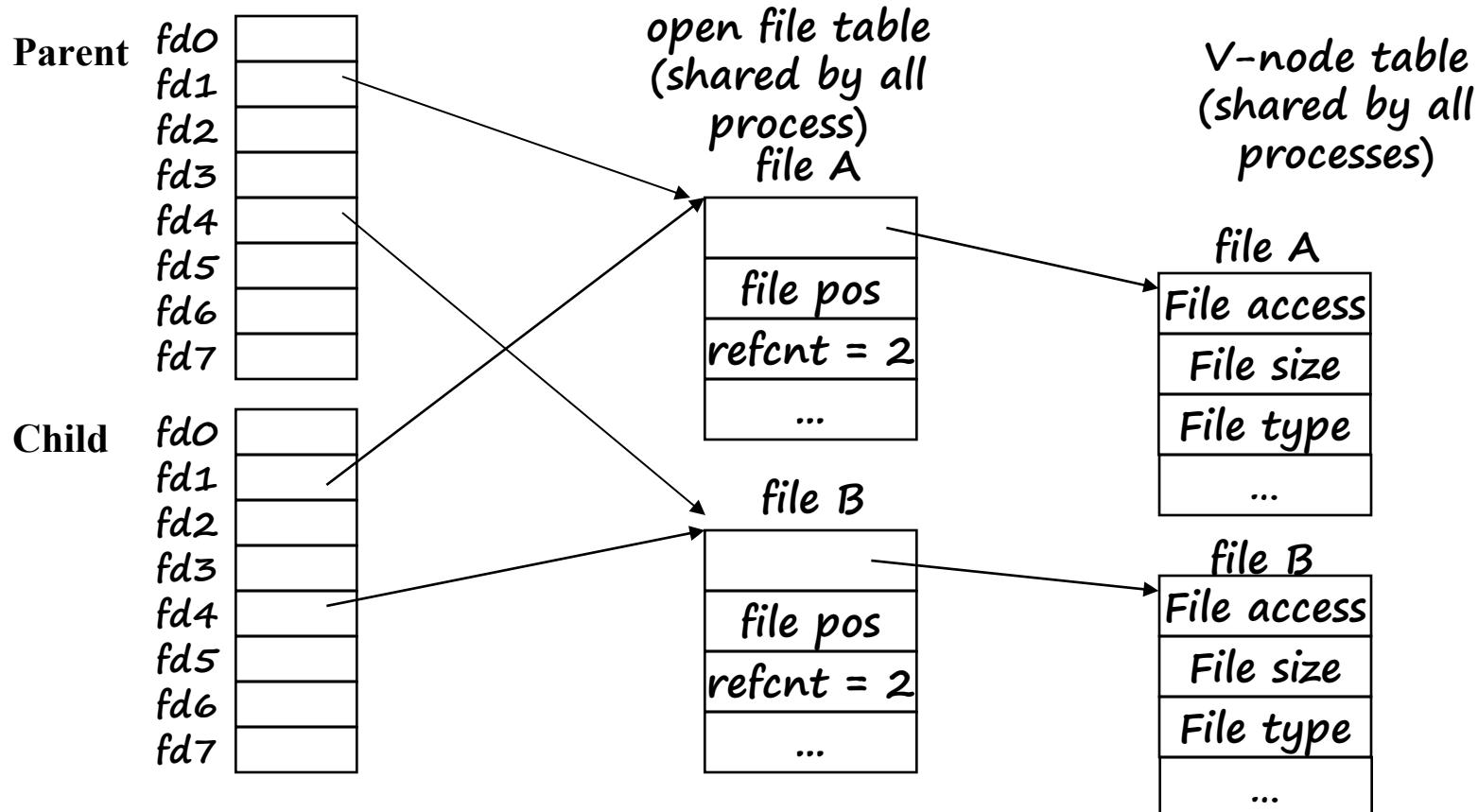
foobar

# Sharing Files between Parent and Child

---



# Sharing Files between Parent and Child



# Sharing Files between Parent and Child

---

```
#include "csapp.h"

int main()
{
    int fd;
    char c;

    fd = open("foobar.txt", O_RDONLY, 0) ;
    if (fork() == 0 ) {
        read(fd, &c, 1) ;
        exit(0) ;
    }
    wait(NULL) ;
    read(fd, &c, 1) ;
    printf("c = %c\n", c) ;
    exit(0)
}
```

**foobar.txt**

foobar

# I/O Redirection

---

unix > ls > foo.txt

- dup2 copies entries in the per-process file descriptor table.

```
#include <unistd.h>

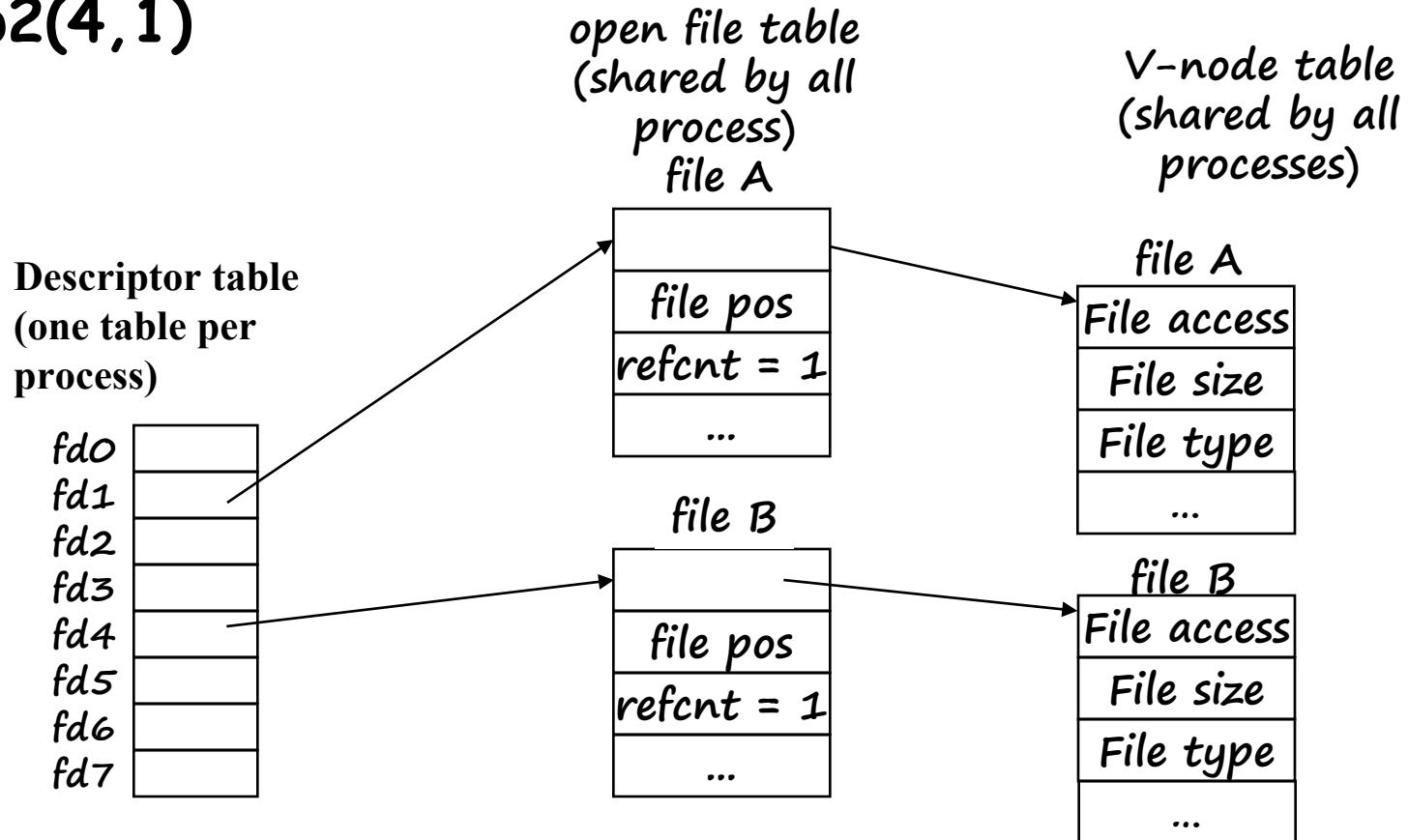
int dup2(int oldfd, int newfd);
    returns: nonnegative descriptor if OK,
              -1 on error
```

- `dup2 (oldfd, newfd)` overwrites the entry in the per-process file table for `newfd` with the entry for `oldfd`.

# Redirection

---

dup2(4,1)



# Redirection

---

dup2(4, 1)

Descriptor table  
(one table per process)

fd0	
fd1	
fd2	
fd3	
fd4	
fd5	
fd6	
fd7	

open file table  
(shared by all processes)  
file A

file pos
refcnt = 0
...

file B

file pos
refcnt = 2
...

V-node table  
(shared by all processes)

file A
File access
File size
File type

file B
File access
File size
File type

# Redirection

---

```
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;

    fd1 = open("foobar.txt", O_RDONLY, 0) ;
    fd2 = open("foobar.txt", O_RDONLY, 0) ;
    read(fd2, &c, 1) ;
    dup2(fd2, fd1) ;
    read(fd1, &c, 1) ;
    printf("c = %c\n", c) ;
    exit(0)
}
```

**foobar.txt**

foobar

# The RIO Package

---

- RIO is a set of wrappers that provide efficient and **robust I/O** in apps, such as network programs that are subject to short counts
- RIO provides two different kinds of functions
  - Unbuffered input and output of binary data
    - `rio_readn` and `rio_writen`
  - Buffered input of binary data and text lines
    - `rio_readlineb` and `rio_readnb`

# Robust I/O

---

- Unbuffered Input and Output
  - Transfer data directly between memory and a file, with **no application-level buffering**

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t count);
ssize_t rio_writen(int fd, void *usrbuf, size_t count);

    return: number of bytes read (0 if EOF) or written,
            -1 on error
```

```
1 ssize_t rio_readn(int fd, void *buf, size_t count)
2 {
3     size_t nleft = count;
4     ssize_t nread;
5     char *ptr = buf;
6
7     while (nleft > 0) {
8         if ((nread = read(fd, ptr, nleft)) < 0) {
9             if (errno == EINTR)
10                 nread = 0; /* and call read() again */
11             else
12                 return -1; /* errno set by read() */
13         }
14         else if (nread == 0)
15             break; /* EOF */
16         nleft -= nread;
17         ptr += nread;
18     }
19     return (count - nleft); /* return >= 0 */
20 }
```

```
1 ssize_t rio_writen(int fd, const void *buf, size_t count)
2 {
3     size_t nleft = count;
4     ssize_t nwritten;
5     const char *ptr = buf;
6
7     while (nleft > 0) {
8         if ((nwritten = write(fd, ptr, nleft)) <= 0) {
9             if (errno == EINTR)
10                 nwritten = 0; /* and call write() again */
11             else
12                 return -1; /* errno set by write() */
13         }
14         nleft -= nwritten;
15         ptr += nwritten;
16     }
17     return count;
18 }
```

# Buffered I/O: Motivation

---

- Applications often read/write one character at a time
  - `getc`, `putc`, `ungetc`
  - `gets`, `fgets`
    - Read line of text on character at a time, stopping at newline
- Implementing as Unix I/O calls expensive
  - `read` and `write` require Unix kernel calls
    - > 10,000 clock cycles

# Buffered I/O: Motivation

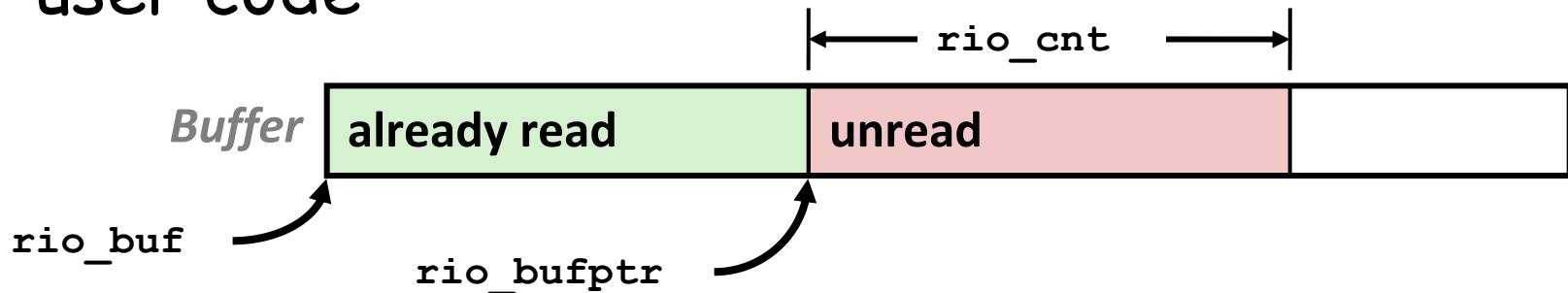
---

- Solution: Buffered read
  - Use Unix `read` to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty



# Buffered I/O: Implementation

- For reading from file
- File has associated buffer to hold bytes that have been read from file but not yet read by user code

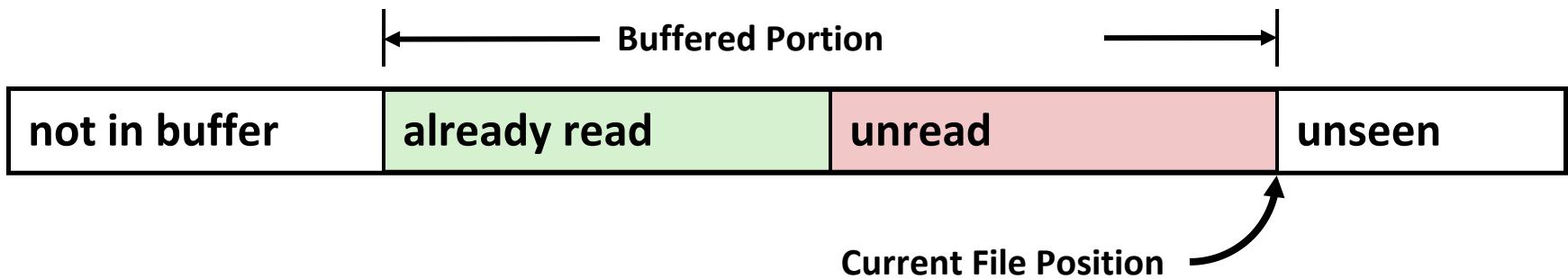


```
typedef struct {
    int rio_fd;                      /* descriptor for this internal buf */
    int rio_cnt;                     /* unread bytes in internal buf */
    char *rio_bufptr;                /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE];       /* internal buffer */
} rio_t;
```

# Buffered I/O: Implementation

---

- Layered on Unix file:



# Robust I/O

---

- Buffered Input and Output
  - Efficiently read text lines and binary data from a file whose contents are cached in **an application-level buffer**

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd) ;
ssize_t rio_readlineb(rio_t *rp,
                      void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp,
                   void *usrbuf, size_t maxlen);

returns: number of bytes read (0 if EOF), -1 on error
```

# Robust I/O

---

```
#define RIO_BUFSIZE 8192
typedef struct {
    int rio_fd;
    int rio_cnt;
    char *rio_bufptr;
    char rio_buf[RIO_BUFSIZE];
} rio_t;

void rio_readinitb(rio_t *rp, int fd)
{
    rp->rio_fd = fd ;
    rp->rio_cnt = 0 ;
    rp->rio_bufptr = rp->rio_buf ;
}
```

# Robust I/O

---

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while ((n = Rio_readlineb(
        &rio, buf, MAXLINE) ) != 0 )
        Rio_writen(STDOUT_FILENO, buf, n);
}
```

```
1 static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
2 {
3     int cnt = 0;
4
5     while (rp->rio_cnt <= 0) { /* refill if buf is empty */
6         rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
7                             sizeof(rp->rio_buf));
8         if (rp->rio_cnt < 0) {
9             if (errno != EINTR)
10                 return -1;
11         }
12         else if (rp->rio_cnt == 0) /* EOF */
13             return 0;
14     else
15         rp->rio_bufptr = rp->rio_buf; /* reset buffer ptr */
16 }
17
```

```
18  /* Copy min(n, rp->rio_cnt) bytes
       from internal buf to user buf */
19  cnt = n ;
20  if ( rp->rio_cnt < n)
21      cnt = rp->rio_cnt ;
22  memcpy(usrbuf, rp->rio_bufptr, cnt) ;
23  rp->rio_buffer += cnt ;
24  rp->rio_cnt -= cnt ;
25  return cnt ;
26 }
```

```
1 ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
2 {
3     size_t nleft = n;    ssize_t nread ;
4     char *bufp = usrbuf;
5     while (nleft > 0) {
6         if ((nread = rio_read(rp, bufp, nleft)) < 0) {
7             if (errno = EINTR)
8                 /* interrupted by sig handler return */
9                 nread = 0;
10            else
11                return -1;
12        }
13        else if (nread == 0)
14            break;
15        nleft -= nread;
16        bufp += nread;
17    }
18    return (n - nleft);
19 }
```

```
1 ssize_t rio_readlineb (rio_t *rp,
2                         void *usrbuf, size_t maxlen)
3 {
4     int n, rc;
5     char c, *bufp = usrbuf;
6     for (n=1; n < maxlen; n++) {
7         if ((rc = rio_read(rp, &c, 1)) == 1) {
8             *bufp++ = c;
9             if (c == '\n')
10                break;
11            } else if (rc == 0) {
12                if (n== 1)
13                    return 0; /* EOF, no data read */
14                else
15                    break;
16            } else
17                return -1; /* error */
18        }
19        *bufp = 0 ;
20    return n ;
```

# Standard I/O

---

- The C standard library (`libc.so`) contains a collection of higher-level standard I/O functions
- Examples of standard I/O functions:
  - Opening and closing files (`fopen` and `fclose`)
  - Reading and writing bytes (`fread` and `fwrite`)
  - Reading and writing text lines (`fgets` and `fputs`)
  - Formatted reading and writing (`fscanf` and `fprintf`)

# Standard I/O

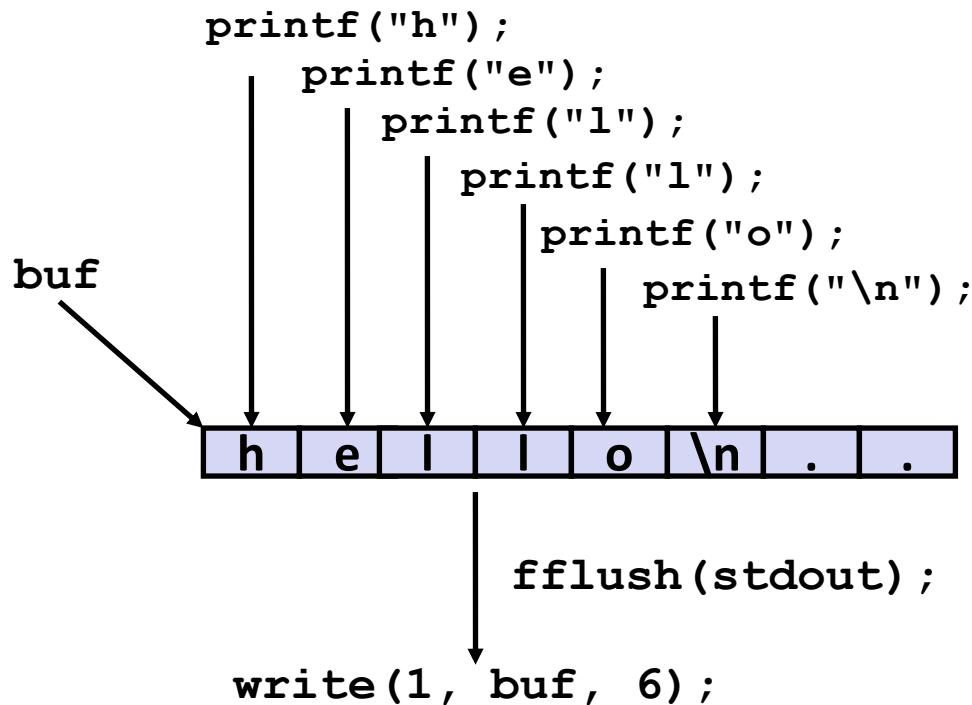
---

- Standard I/O models open files as **streams**
  - Abstraction for a file descriptor and a buffer in memory.
  - Similar to buffered RIO
- C programs begin life with three open streams (defined in `stdio.h`)
  - `stdin` (standard input `fd=0`)
  - `stdout` (standard output `fd=1`)
  - `stderr` (standard error `fd=2`)

# Buffering in Standard I/O

---

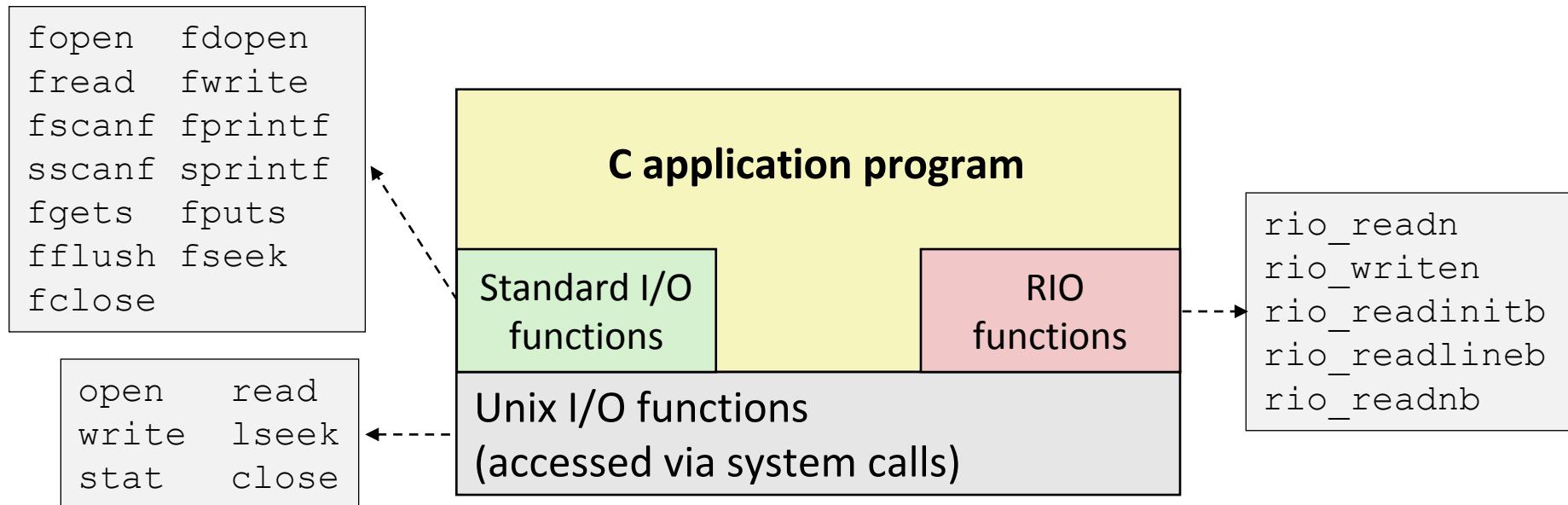
- Standard I/O functions use buffered I/O



- Buffer flushed to output fd on "\n" or fflush() call

# Unix I/O, Standard I/O, and Robust I/O

- Standard I/O and RIO are implemented using low-level Unix I/O



- Which ones should you use in your programs?

# Pros and Cons of Unix I/O

---

- Pros
  - Unix I/O is the most general and lowest overhead form of I/O.
    - All other I/O packages are implemented using Unix I/O functions.
  - Unix I/O provides functions for accessing file metadata.
  - Unix I/O functions are async-signal-safe and can be used safely in signal handlers.

# Pros and Cons of Unix I/O

---

- Cons
  - Dealing with short counts is tricky and error prone.
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone.
  - Both of these issues are addressed by the standard I/O and RIO packages.

# Pros and Cons of Standard I/O

---

- Pros:
  - Buffering increases efficiency by decreasing the number of `read` and `write` system calls
  - Short counts are handled automatically

# Pros and Cons of Standard I/O

---

- **Cons:**
  - Provides no function for accessing file metadata
  - Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers.
  - Standard I/O is not appropriate for input and output on network sockets
    - There are poorly documented restrictions on streams that interact badly with restrictions on sockets

# Choosing I/O Functions

---

- General rule: use the highest-level I/O functions you can
  - Many C programmers are able to do all of their work using the standard I/O functions

# Choosing I/O Functions

---

- When to use standard I/O
  - When working with disk or terminal files
- When to use raw Unix I/O
  - Inside signal handlers, because Unix I/O is async-signal-safe.
  - In rare cases when you need absolute highest performance.
- When to use RIO
  - When you are reading and writing network sockets.
  - Avoid using standard I/O on sockets.

# Networking Programming

# Outline

---

- Client-Server Programming Model
- Networks
- Global IP Internet
- Suggested Reading:
  - 11.1~11.3

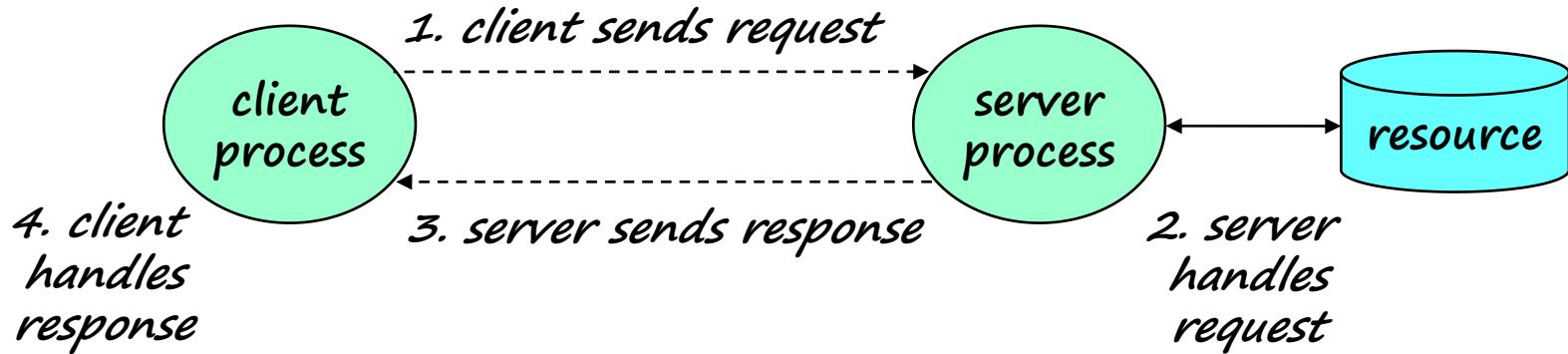
# A client-server transaction

---

- Most network applications are based on the client-server model
  - A **server** process and one or more **client** processes
  - Server manages some **resource**
  - Server provides **service** by manipulating resource for **clients**
  - Server activated by **request** from client

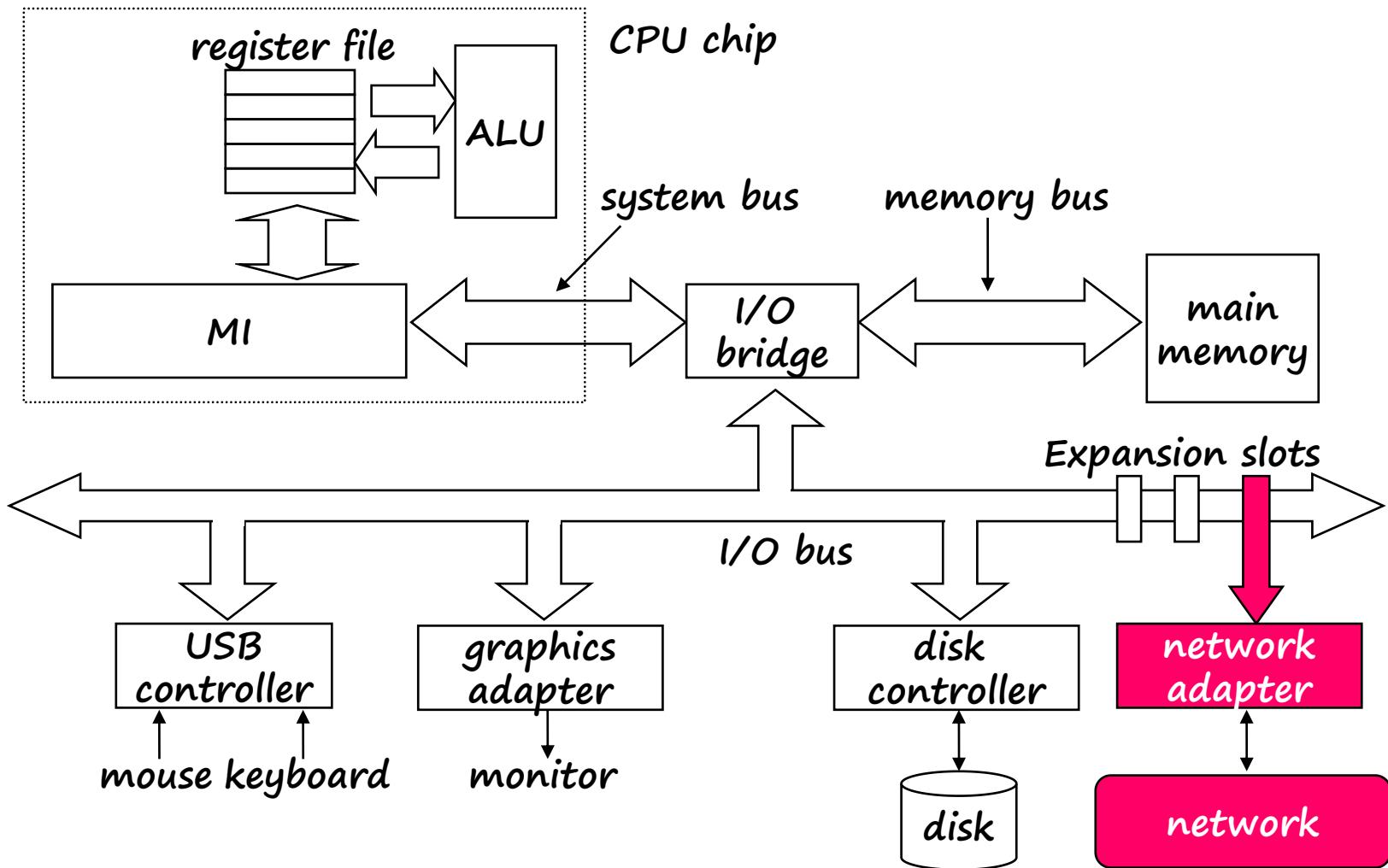
# A client-server transaction

---



Note: clients and servers are processes running on hosts  
(can be the same or different hosts).

# Hardware organization of a network host



# Computer networks

---

- A **network** is a hierarchical system of boxes and wires organized by geographical proximity
  - SAN (System Area Network) spans cluster or machine room
    - Switched Ethernet, Quadric QSW
  - LAN (Local Area Network) spans a building or campus
    - Ethernet is most prominent example
  - WAN (Wide-Area Network) spans country or world
    - typically high-speed point-to-point phone lines

# Computer networks

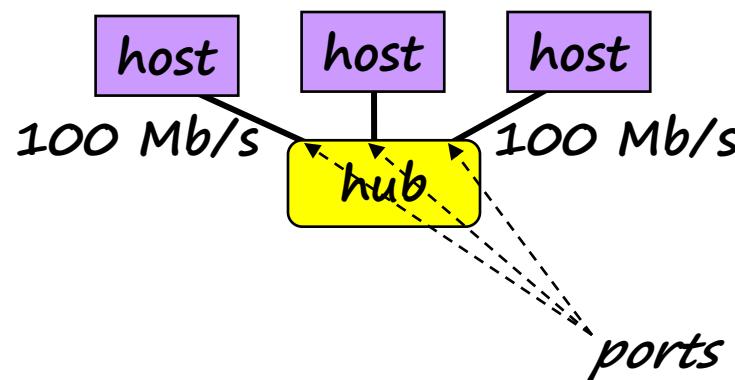
---

- An internetwork (**internet**) is an interconnected set of networks
  - The Global IP Internet is the most famous example of an internetwork
- Let's see how we would build an internetwork from the ground up

# Lowest level: Ethernet segment

---

- Ethernet segment consists of a collection of **hosts** connected by wires (twisted pairs) to a **hub**
- Spans room or floor in a building.



# Lowest level: Ethernet segment

---

- Operation
  - Each **Ethernet adapter** has a unique 48-bit address
    - E.g. 00:16:**ea**:**e3**:**54**:**e6**
  - Hosts send bits to any other host in chunks called **frames**.
  - **Hub** slavishly copies each bit from each port to every other port
    - Every host sees every bit
    - NOTE: Hubs are on their way out

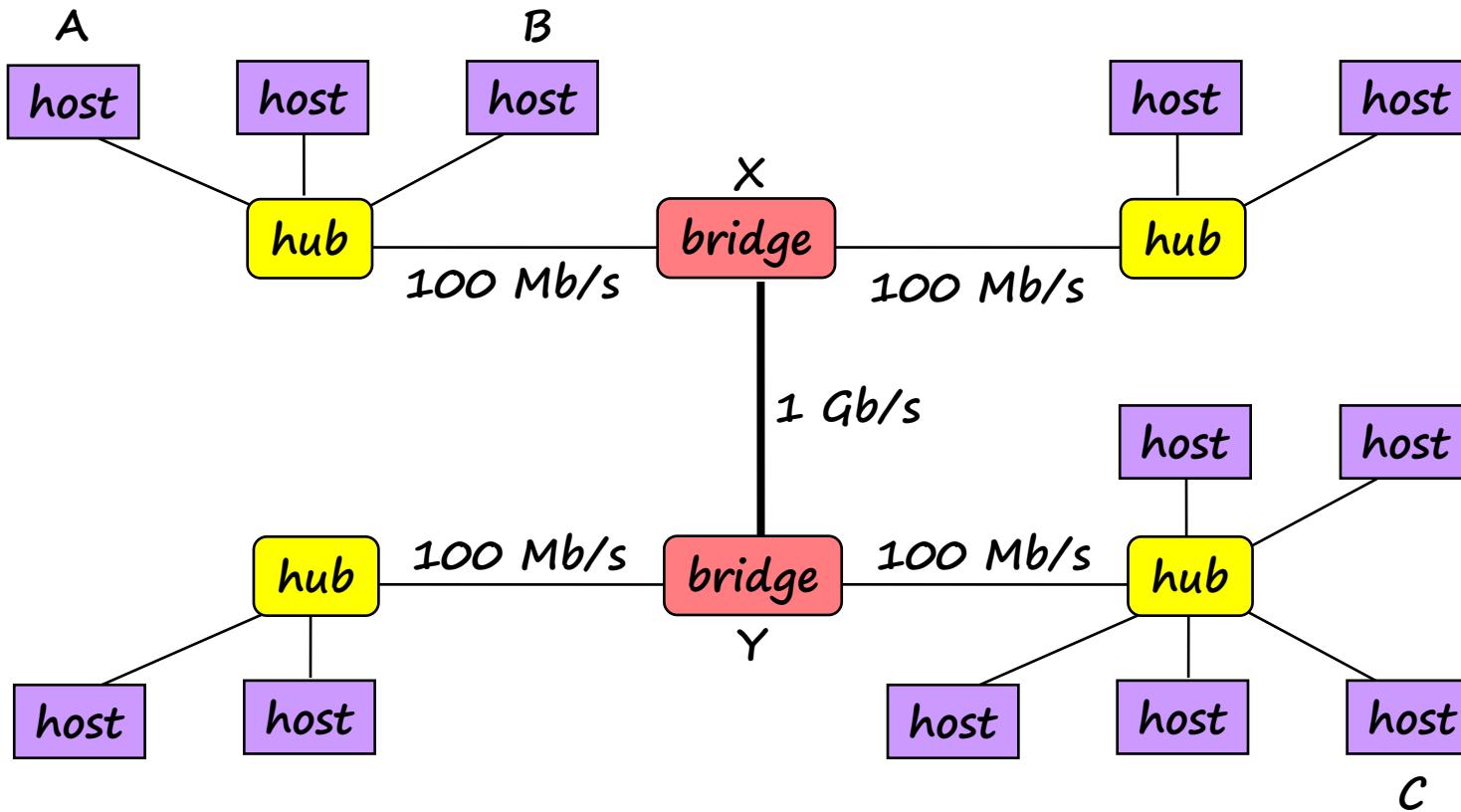
# Next level: Bridged Ethernet segment

---

- Spans building or campus
- Bridges cleverly learn which hosts are reachable from which ports and then selectively copy frames from port to port

# Next level: Bridged Ethernet segment

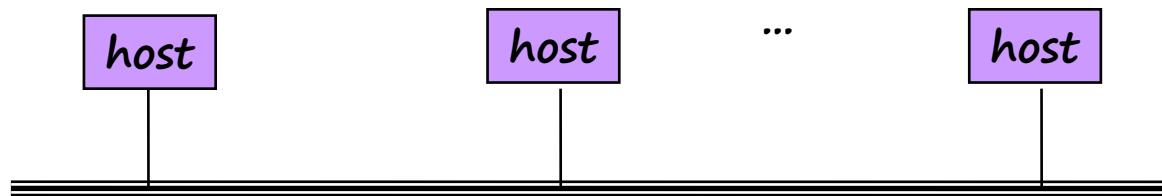
---



# Conceptual view of LANs

---

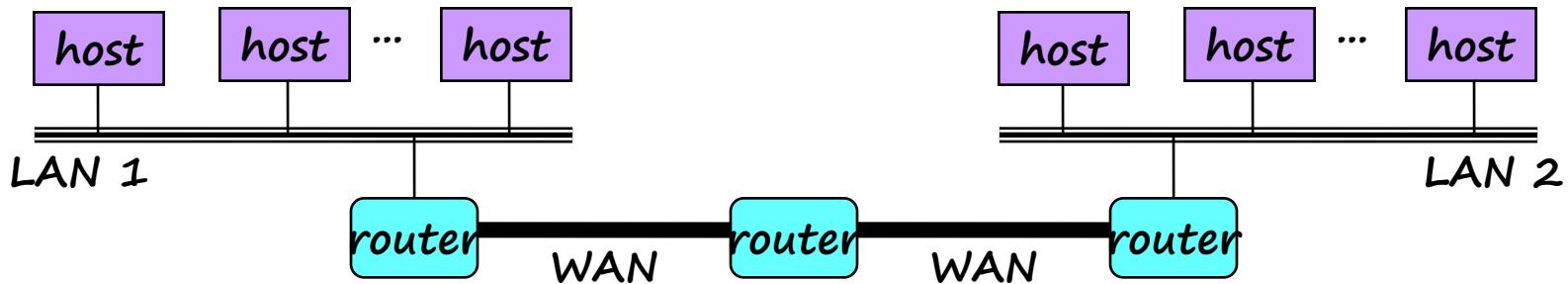
- For simplicity, hubs, bridges, and wires are often shown as a collection of hosts attached to a single wire:



# Next level: internets

---

- Multiple incompatible LANs can be physically connected by specialized computers called **routers**
- The connected networks are called an **internet**

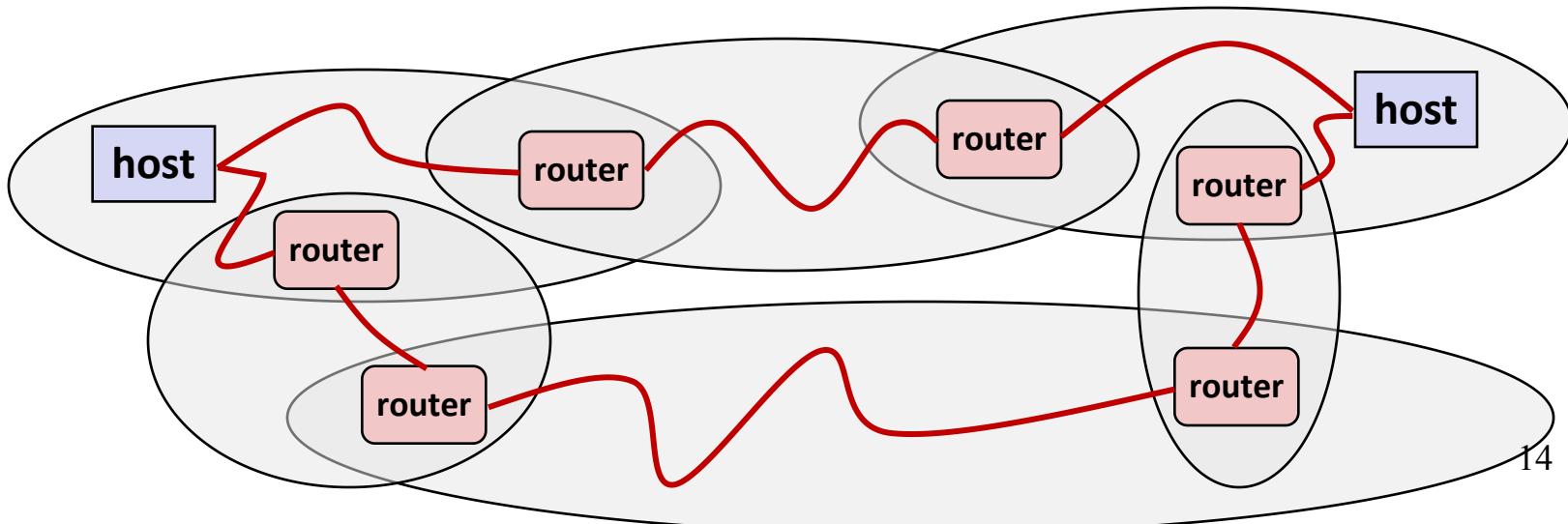


LAN 1 and LAN 2 might be completely different, totally incompatible LANs (e.g. Ethernet, Wifi, DSL)

# Logical Structure of an internet

---

- Ad hoc interconnection of networks
  - No particular topology
  - Vastly different router & link capacities
- Send packets from source to destination by hopping through networks
  - Router forms bridge from one network to another



# The notion of an internet protocol

---

- How is it possible to send bits across incompatible LANs and WANs?
- Solution:
  - **protocol software** running on each host and router smoothens out the differences between the different networks

# The notion of an internet protocol

---

- Implements an **internet protocol** (i.e., set of rules)
  - governs how hosts and routers should cooperate when they transfer data from network to network
  - **TCP/IP** is the protocol for the global IP Internet.

# What does an internet protocol do?

---

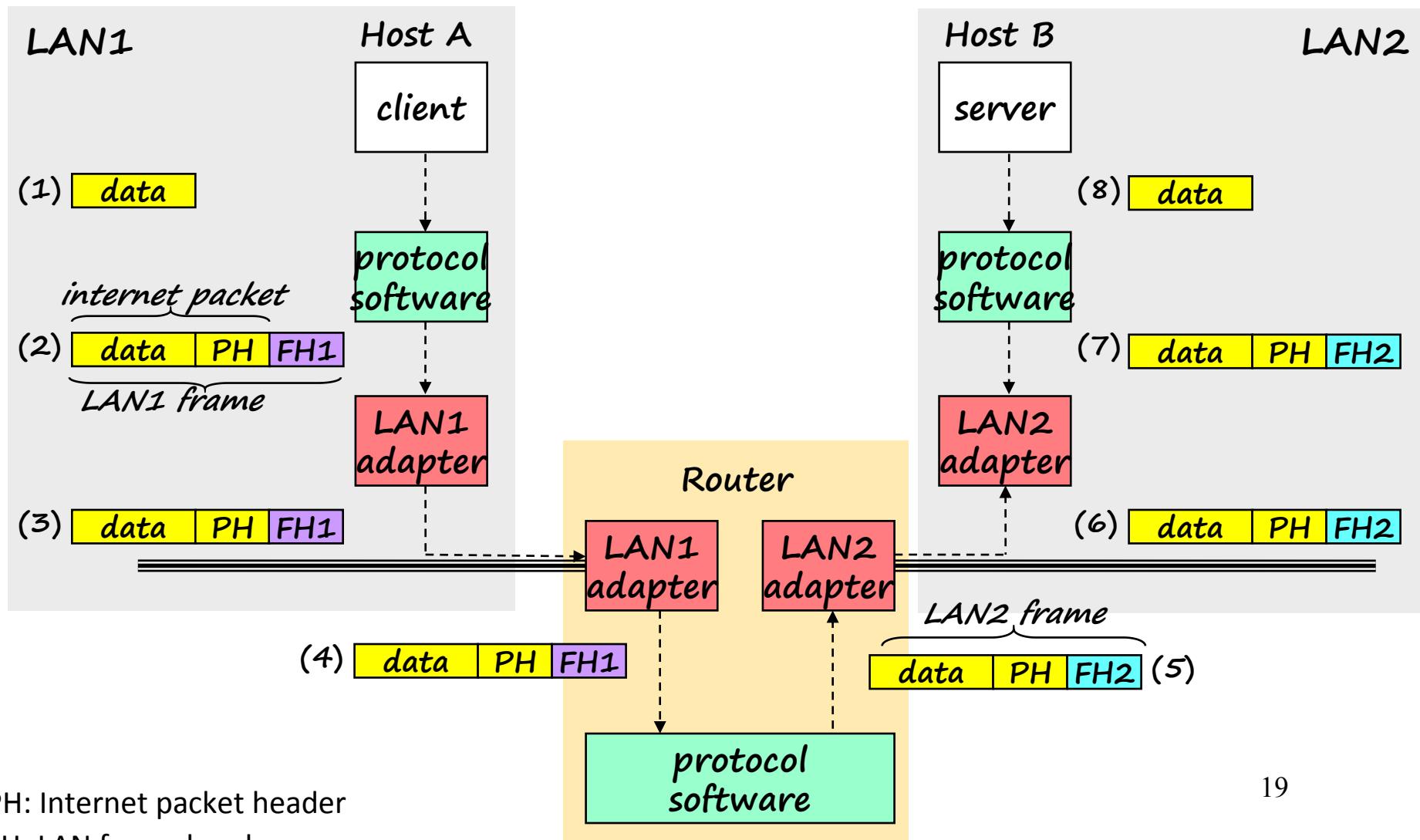
- Provides a **naming scheme**
  - The internet protocol defines a uniform format for **host addresses**
  - Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it.

# What does an internet protocol do?

---

- Provide a **delivery mechanism**
  - The internet protocol defines a standard transfer unit (**packet**)
  - Packet consists of **header** and **payload**
    - header: contains info such as packet size, source and destination addresses.
    - payload: contains data bits sent from source host.

# Transferring data over an internet



# Other issues

---

- We are glossing over a number of important questions:
  - What if different networks have different maximum frame sizes? (segmentation)
  - How do routers know where to forward frames?
  - How are routers informed when the network topology changes?
  - What if packets get lost?

# Other issues

---

- These questions form the heart of the area of computer systems known as **computer networking**

# Global IP Internet

---

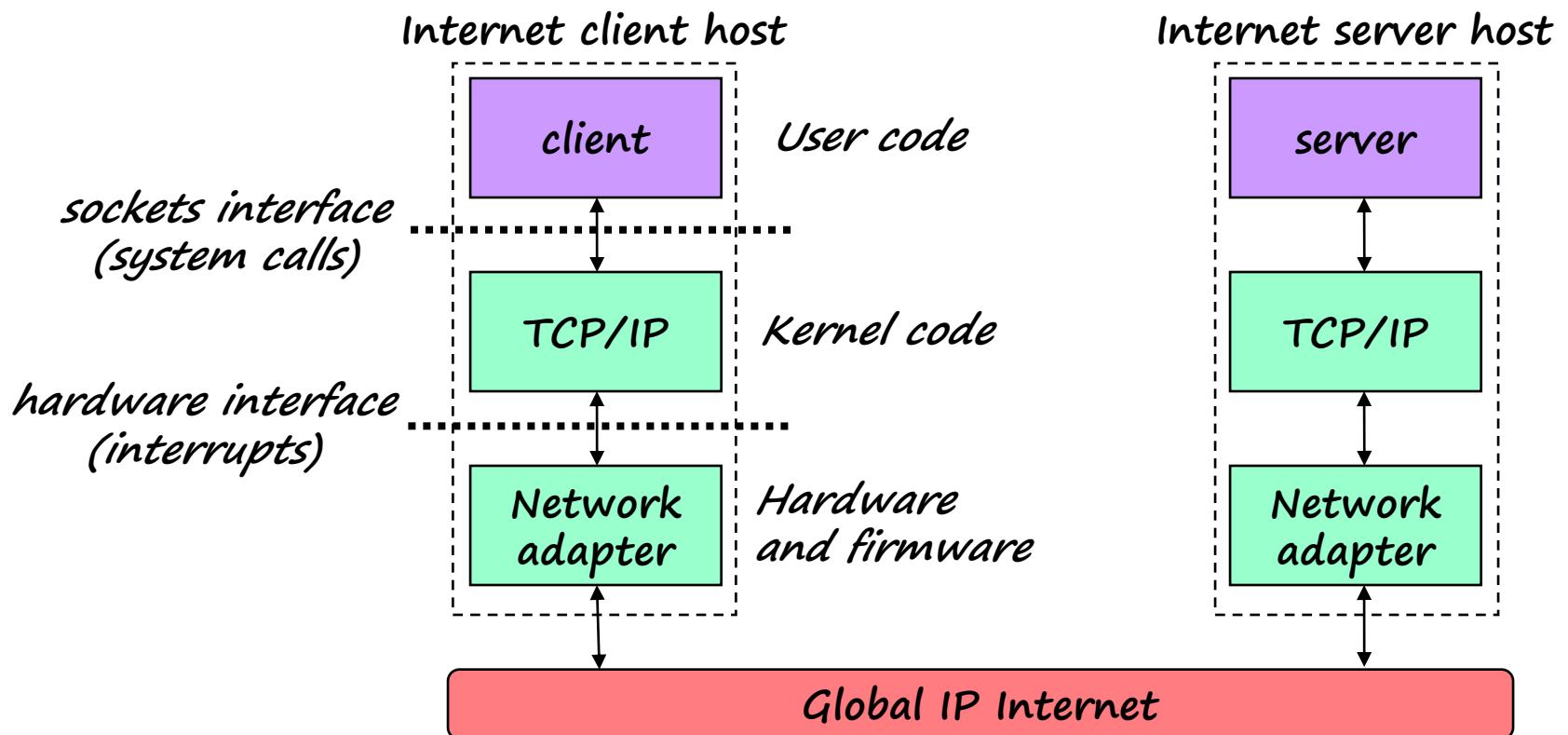
- Most famous example of an internet.
- Based on the TCP/IP protocol family.
  - **IP** (Internet protocol) :
    - provides basic **naming scheme** and unreliable **delivery capability** of packets (datagrams) from host-to-host.

# Global IP Internet

---

- Based on the TCP/IP protocol family.
  - **UDP** (Unreliable Datagram Protocol)
    - uses IP to provide **unreliable datagram delivery** from process-to-process.
  - **TCP** (Transmission Control Protocol)
    - uses IP to provide **reliable byte streams** (like files) from process-to-process.
- Accessed via a mix of Unix file I/O and functions from the Berkeley **sockets interface**

# Hardware and software organization of an Internet application



# Programmer's view of the Internet

---

- Hosts are mapped to a set of 32-bit **IP addresses**
  - 202.120.40.188 (ICS server)
- The set of IP addresses is mapped to a set of identifiers called Internet **domain names**
  - 202.120.40.188 is mapped to ipads.se.sjtu.edu.cn
- A process on one host communicates with a process on another host over a **connection**

# Dotted decimal notation

---

- By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period
  - IP address **0xca7828bc** = 202.120.40.188

# Dotted decimal notation

---

- Functions for converting between binary IP addresses and dotted decimal strings:
  - `inet_aton`: converts a dotted decimal string to an IP address in network byte order.
  - `inet_ntoa`: converts an IP address in network byte order to its corresponding dotted decimal string.
  - “n” denotes network representation. “a” denotes application representation.

# IP Addresses

---

- 32-bit IP addresses are stored in an **IP address struct**
  - IP addresses are always stored in memory in network byte order (**big-endian** byte order)

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian)
*/
```

};

## Useful network byte-order conversion functions

**htonl**: convert uint32\_t from host to network byte order

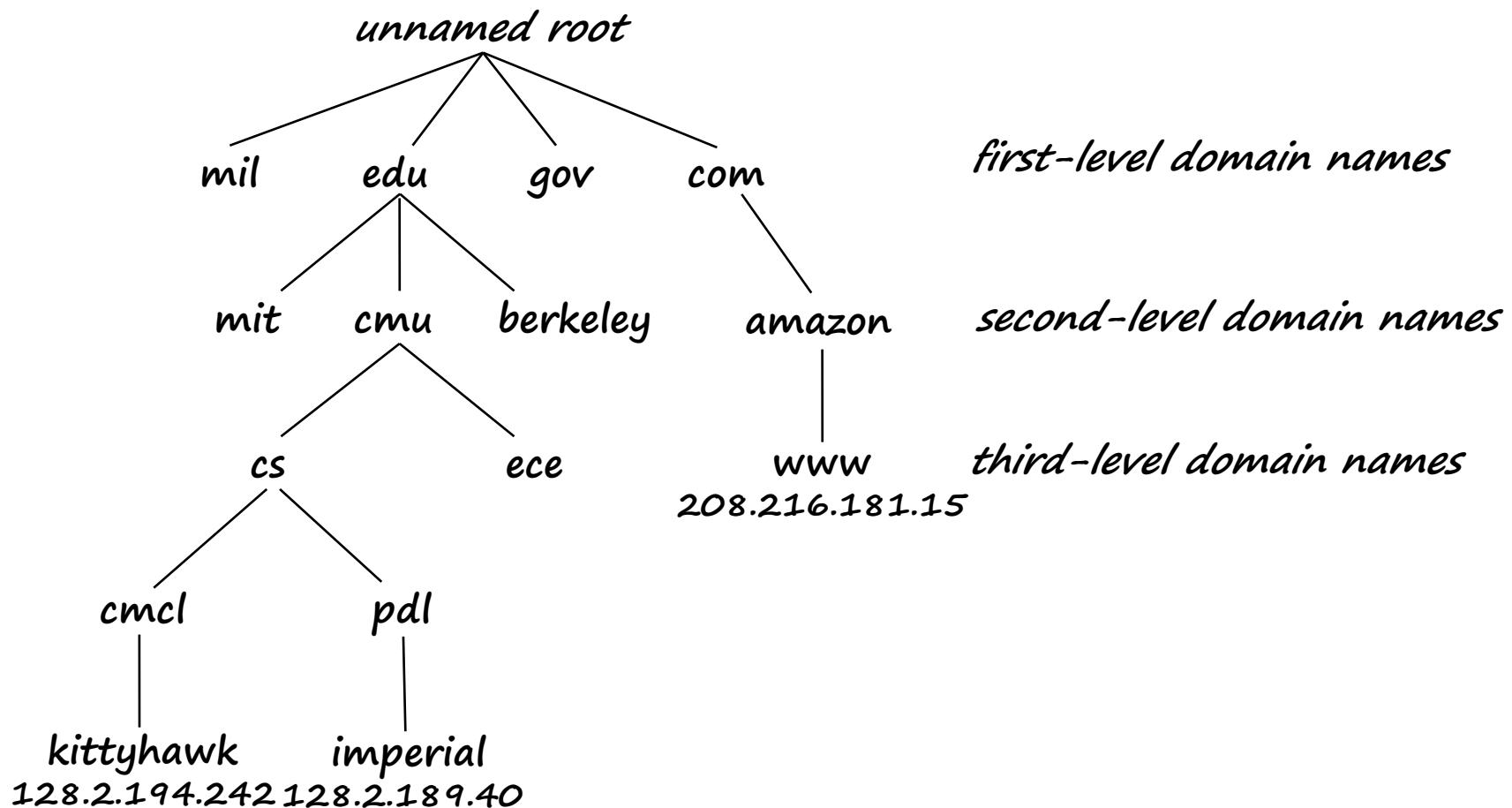
**htons**: convert uint16\_t from host to network byte order

**ntohl**: convert uint32\_t from network to host byte order

**ntohs**: convert uint16\_t from network to host byte order

# Internet Domain Names

---



# Domain Naming System (DNS)

---

- The Internet maintains a **mapping** between IP addresses and domain names in a huge distributed database called **DNS**
- Conceptually, we can think of the DNS database as being millions of **host entry structures**

# Domain Naming System (DNS)

---

```
/* DNS host entry structure */
struct hostent {
    /* official domain name of host */
    char *h_name;
    /* null-terminated array of domain names */
    char **h_aliases;
    /* host address type (AF_INET) */
    int h_addrtype;
    /* length of an address, in bytes */
    int h_length;
    /* null-terminated array of in_addr structs*/
    char **h_addr_list;
};
```

# Properties of DNS host entries

---

- Each host entry is an equivalence class of domain names and IP addresses
- Each host has a locally defined domain name **localhost** which always maps to the loopback address **127.0.0.1**

# Properties of DNS host entries

---

- Different kinds of mappings are possible:
  - **1-1**: mapping between domain name and IP address:
    - `ipads.se.sjtu.edu.cn` maps to `202.120.40.188`
  - **M-1**: Multiple domain names mapped to the same IP address
    - `google.com.cn` and `google.cn` both map to `74.125.128.160`

# Properties of DNS host entries

---

- Different kinds of mappings are possible:
  - **M-N:** Multiple domain names mapped to multiple IP addresses:
    - `baidu.com` and `baidu.com.cn` map to three different IP addresses
  - **1-?:** Some valid domain name don't map to any IP address:
    - `ics.se.sjtu.edu.cn`

# Domain Naming System (DNS)

---

- Functions for retrieving host entries from DNS:
  - **gethostbyname**: query key is a DNS domain name
  - **gethostbyaddr**: query key is a an IP address

# A program that queries DNS

---

```
int main(int argc, char **argv) { /* argv[1] is a domain name */
    char **pp;                                /* or dotted decimal IP addr */
    struct in_addr addr;
    struct hostent *hostp;

    if (inet_aton(argv[1], &addr) != 0)
        hostp = gethostbyaddr((const char *)&addr, sizeof(addr),
                               AF_INET);
    else
        hostp = gethostbyname(argv[1]);
    printf("official hostname: %s\n", hostp->h_name);

    for (pp = hostp->h_aliases; *pp != NULL; pp++)
        printf("alias: %s\n", *pp);

    for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
        addr.s_addr = *((unsigned int *)*pp);
        printf("address: %s\n", inet_ntoa(addr));
    }
}
```

# Using DNS program

---

```
$ ./dns ipads.se.sjtu.edu.cn
```

```
official hostname: ipads.se.sjtu.edu.cn
```

```
address: 202.120.40.188
```

```
$ ./dns baidu.com
```

```
official hostname: baidu.com
```

```
address: 123.125.114.144
```

```
address: 220.181.111.85
```

```
address: 220.181.111.86
```

# Internet connections

---

- Clients and servers communicate by sending streams of bytes of connections
  - point-to-point, full-duplex, and reliable
- A **socket** is an endpoint of a connection
  - Socket address is an **IPaddress:port** pair

# Internet connections

---

- A **port** is a 16-bit integer that identifies a process:
  - ephemeral port: assigned automatically on client when client makes a connection request
  - well-known port: associated with some service provided by a server (e.g., port 80 is associated with Web servers)

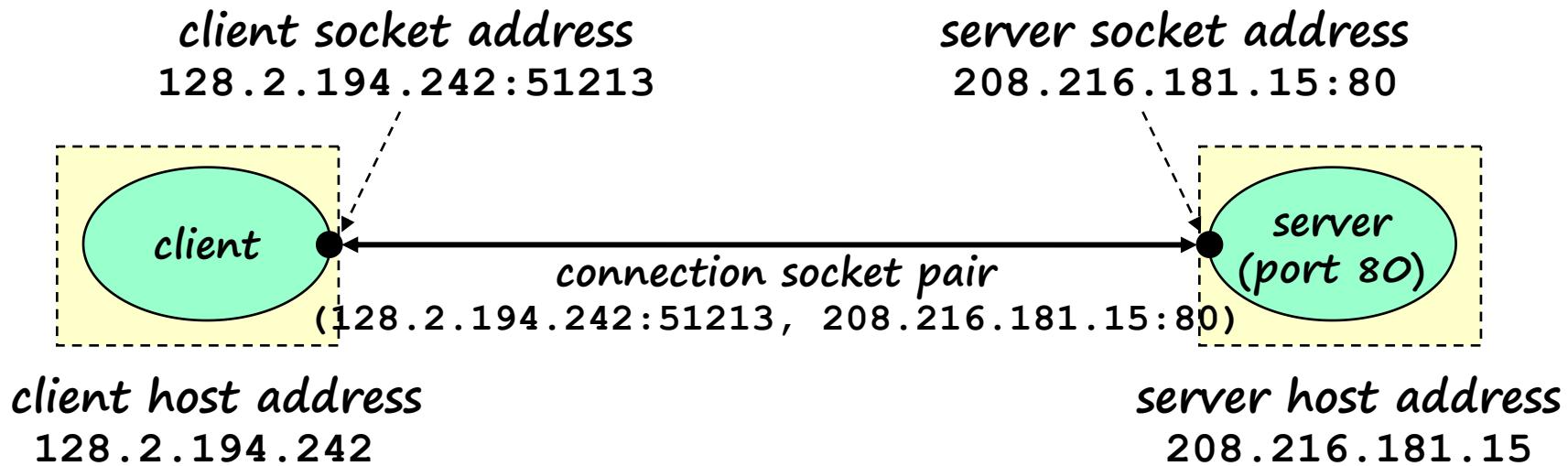
# Internet connections

---

- A connection is uniquely identified by the socket addresses of its endpoints (**socket pair**)
  - `(cliaddr:cliport, servaddr:servport)`

# Putting it all together: Anatomy of an Internet connection

---



# Networking Programming

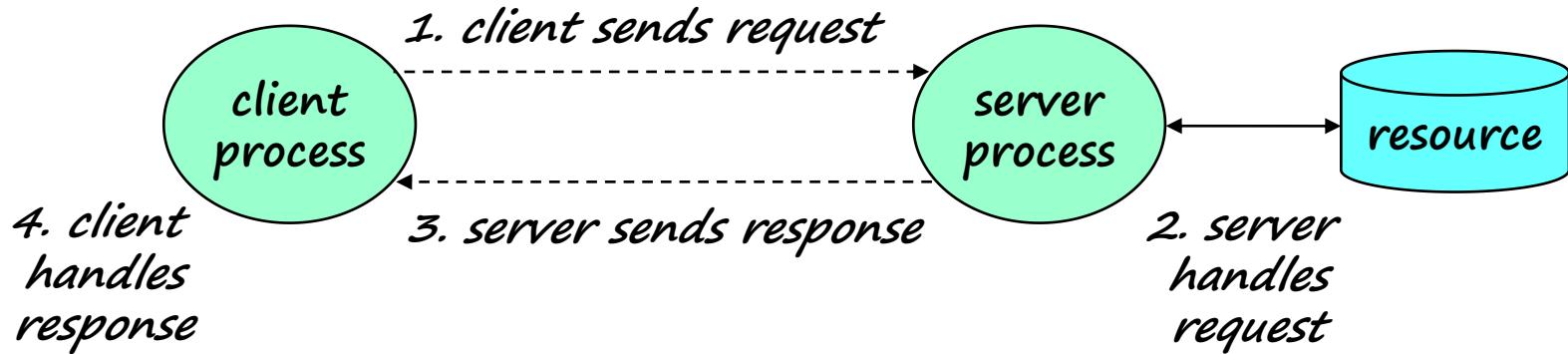
# Outline

---

- Sockets Interface
  - Functions
  - Echo Client and Server
- Suggested Reading:
  - 11.4

# A client-server transaction

---



Note: clients and servers are processes running on hosts  
(can be the same or different hosts).

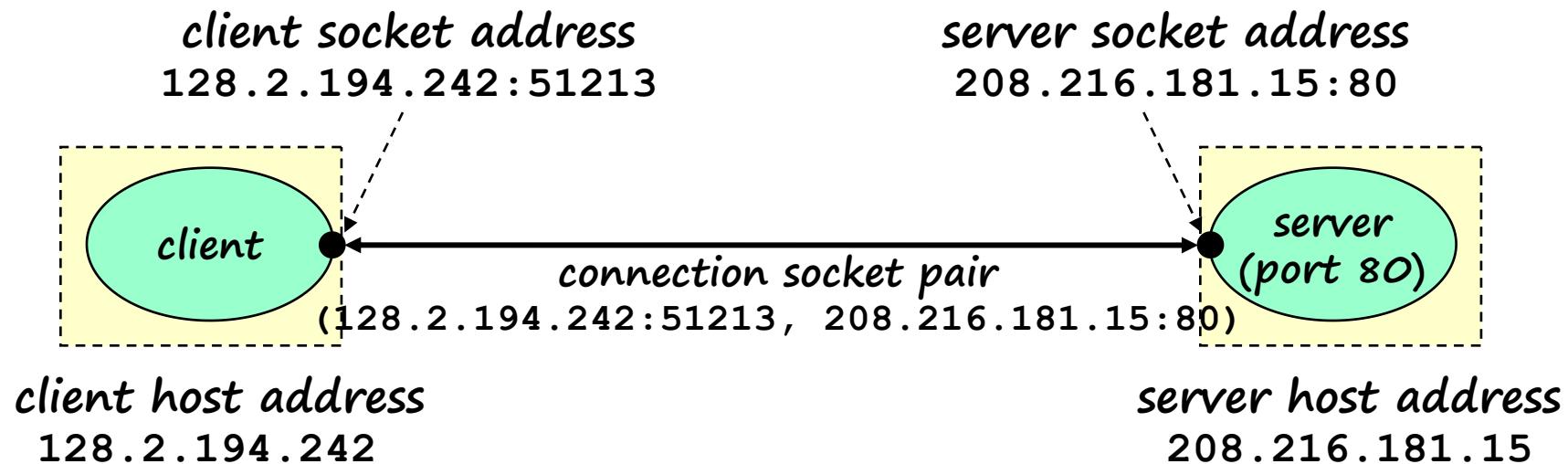
# Internet connections

---

- Clients and servers communicate by sending streams of bytes of connections
  - point-to-point, full-duplex, and reliable
- A **socket** is an endpoint of a **connection**
  - Socket address is an **IPaddress:port** pair

# Putting it all together: Anatomy of an Internet connection

---



# Clients

---

- Examples of client programs
  - Web browsers, ftp, telnet, ssh
- How does the client find the server?
  - The address of the server process has two parts:  
**IP-address:port**
    - The **IP address** is a unique 32-bit positive integer that identifies the host (adapter).
      - dotted decimal form: 0x8002C2F2 = 128.2.194.242
    - The **port** is 16-bit positive integer associated with a service (and thus a server process) on that machine

# Servers

---

- Servers are long-running processes (**daemons**)
  - Created at boot-time (typically) by the `init` process (process 1)
  - Run continuously until the machine is turned off
- A machine that runs a server process is also often referred to as a "server".

# Example: Echo Client and Server

---

On Client

```
server> ./echoserveri 15213  
client> echoclient greatwhite.ics.cs.cmu.edu 15213
```

On Server

```
server connected to BRYANT-  
TP4.VLSI.CS.CMU.EDU (128.2.213.29)
```

```
type: hello there
```

```
server received 12 bytes
```

```
echo: HELLO THERE
```

```
type: ^D
```

```
Connection closed
```

# Berkeley Sockets Interface

---

- Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols
- Provides a user-level interface to the network
- Underlying basis for all Internet applications
- Based on client/server programming model

# What is a socket?

---

- A **socket** is a descriptor that lets an application read/write from/to the network.
  - Key idea: Unix uses the same abstraction for both file I/O and network I/O

# What is a socket?

---

- Clients and servers communicate with each other by reading from and writing to **socket descriptors**
  - Using regular Unix **read** and **write** I/O functions
- The main difference between file I/O and socket I/O is how the application “opens” the socket descriptors

# Key data structures

---

- Internet-style sockets are characterized by
  - a 32-bit IP address and a port
- Defined in `/usr/include/netinet/in.h`

```
/* Internet address */
struct in_addr {
    unsigned int s_addr; /* 32-bit IP address */
};
```

# Key data structures

---

```
/* Internet style socket address */
struct sockaddr_in {
    unsigned short int sin_family;
                    /*Address family (AF_INET) */
    unsigned short int sin_port;      /*Port number*/
    struct in_addr sin_addr;          /*IP address*/
    unsigned char sin_zero[8]; /*Pad to "sockaddr"*/
};
```

```
/* Generic socket address structure */
struct sockaddr {
    unsigned short int sin_family;
                    /*Protocol family (AF_INET)*/
    char sa_data[14];           /*address data*/
};
```

# Key data structures

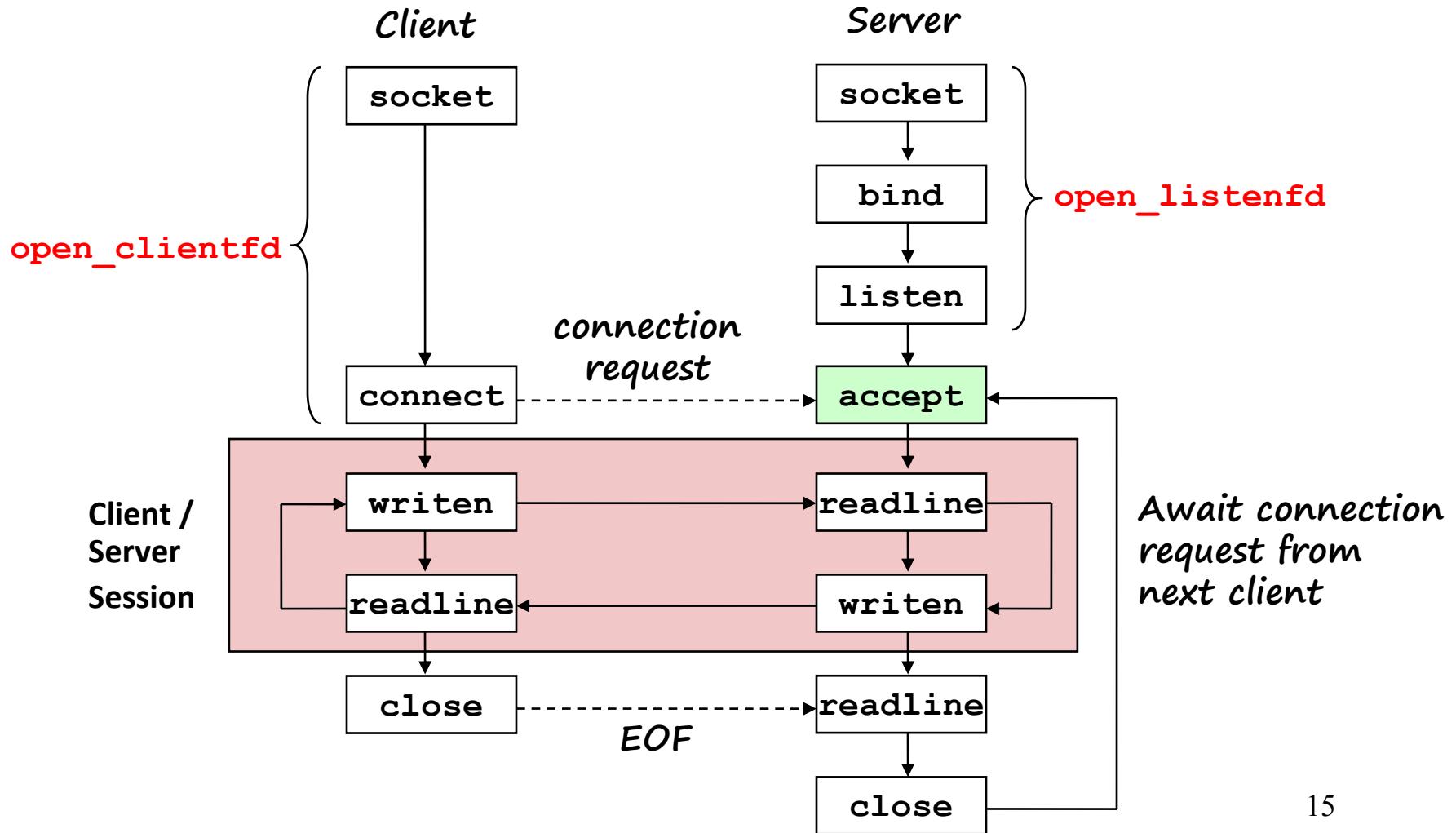
---

- Defined in `/usr/include/netdb.h`

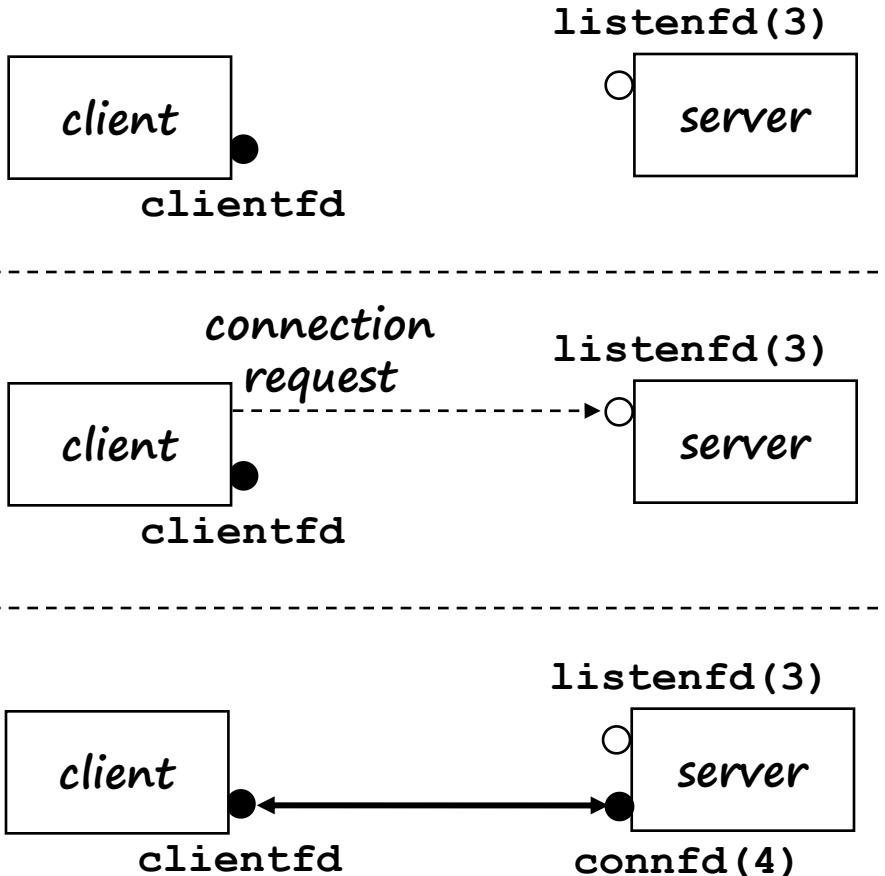
```
/* Domain Name Service (DNS) host entry */

struct hostent {
    char      *h_name;      /* official name of host */
    char      **h_aliases;   /* alias list */
    int       h_addrtype;   /* host address type */
    int       h_length;     /* length of address */
    char      **h_addr_list; /*list of addresses */
}
```

# Overview of the Sockets Interface



# accept() illustrated



1. *Server blocks in accept, waiting for connection request on listening descriptor `listenfd`.*
2. *Client makes connection request by calling and blocking in `connect`.*
3. *Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`.*

# Echo client

```
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;

    if (argc != 3) {
        fprintf(stderr,"usage:%s <host> <port>\n",argv[0]);
        exit(0);
    }
    host = argv[1]; port = atoi(argv[2]);
    clientfd = open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readline(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
}
```

# Echo client: open\_clientfd()

```
int open_clientfd(char *hostname, int port)
{
    int clientfd;
    struct hostent *hp;
    struct sockaddr_in serveraddr;

    clientfd = Socket(AF_INET, SOCK_STREAM, 0);

    /* fill in the server's IP address and port */
    hp = Gethostbyname(hostname);
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *)hp->h_addr,
          (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
    serveraddr.sin_port = htons(port);

    /* establish a connection with the server */
    Connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr));

    return clientfd;
}
```

## Echo client: `open_clientfd()` (socket)

---

- The client creates a **socket** that will serve as the endpoint of an Internet (AF\_INET) **connection** (SOCK\_STREAM).
  - `socket()` returns an integer socket descriptor.

```
int clientfd; /* socket descriptor */  
  
clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

## Echo client: open\_clientfd()

---

- The client builds the server's Internet address.

```
struct hostent *hp;                      /* DNS host entry */
struct sockaddr_in serveraddr; /* server's IP address */

/* fill in the server's IP address and port */
hp = Gethostbyname(hostname);
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family_ = AF_INET;
bcopy((char *)hp->h_addr,
      (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
serveraddr.sin_port = htons(port);
```

## Echo client: `open_clientfd()` (`connect`)

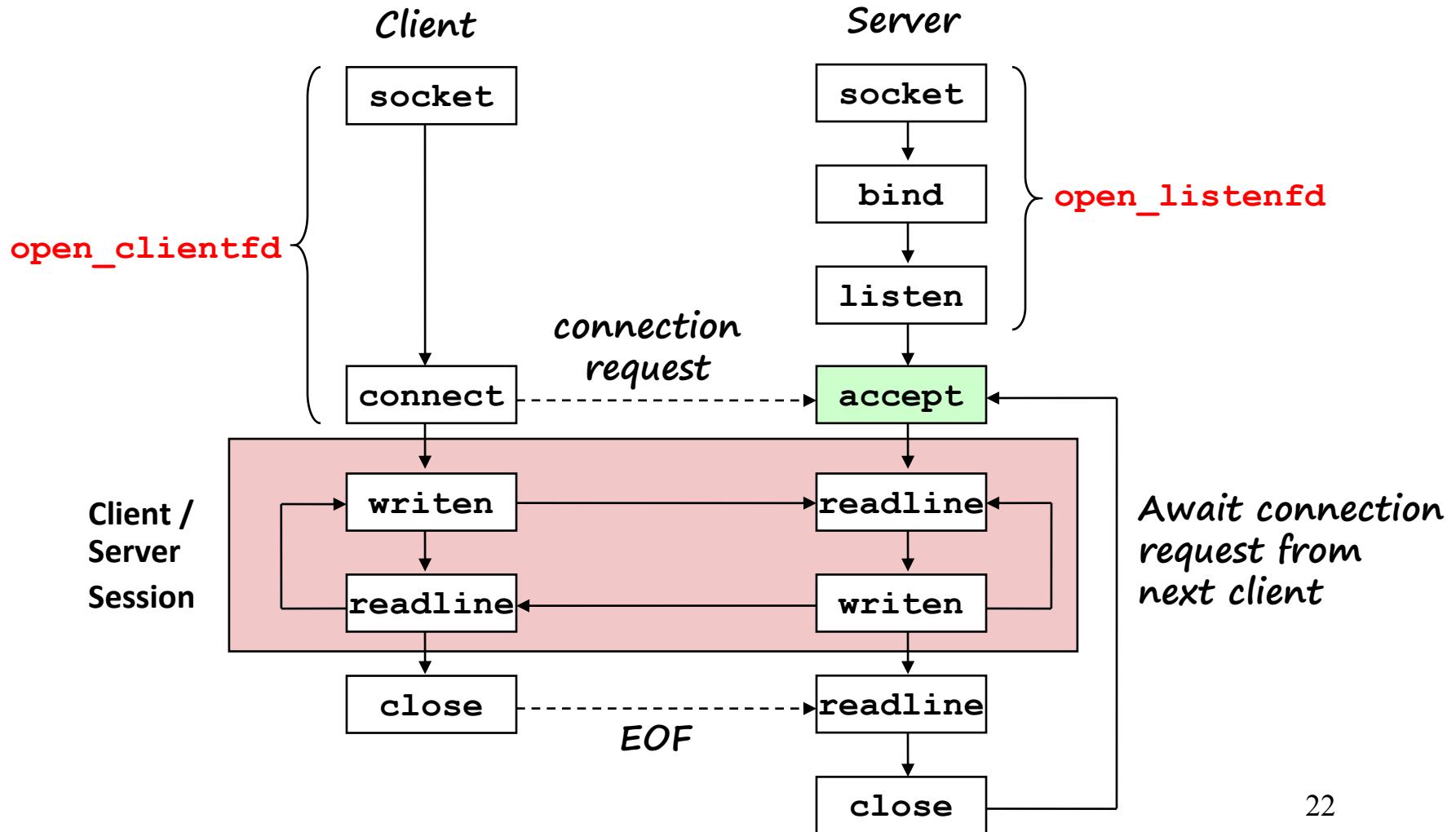
---

- The client creates a connection with the server
  - The client process suspends (blocks) until the connection is created with the server.
  - At this point the client is ready to begin exchanging messages with the server via Unix I/O calls on the descriptor `clientfd`.

```
int clientfd;                                /* socket descriptor */
struct sockaddr_in serveraddr;                /* server address */

/* establish a connection with the server */
Connect(clientfd, (SA*)&serveraddr, sizeof(serveraddr));
```

# Overview of the Sockets Interface



# Echo server

---

```
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;

    port = atoi(argv[1]); /* the server listens on
                           a port passed on the command line */
    listenfd = open_listenfd(port);
```

# Echo server

---

```
while (1) {  
    clientlen = sizeof(clientaddr);  
    connfd = Accept(listenfd,  
                    (SA *)&clientaddr, &clientlen);  
    hp = Gethostbyaddr((const char *)  
                        &clientaddr.sin_addr.s_addr,  
                        sizeof(struct in_addr), AF_INET);  
    haddrp = inet_ntoa(clientaddr.sin_addr);  
    printf("server connected to %s (%s)\n",  
          hp->h_name, haddrp);  
    echo(connfd);  
    Close(connfd);  
}  
}
```

# Echo server: open\_listenfd()

---

```
int open_listenfd(int port)
{
    int listenfd;
    int optval;
    struct sockaddr_in serveraddr;

    /* create a socket descriptor */
    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    /* eliminates "Address already in use"
       error from bind. */
    optval = 1;
    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
               (const void *)&optval , sizeof(int));
```

# Echo server: open\_listenfd() (cont)

---

```
/* listenfd will be an endpoint for all requests
   to port on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port =
    htons((unsigned short)port);
Bind(listenfd,
      (SA *)&serveraddr, sizeof(serveraddr));
/* make it a listening socket ready to accept
   connection requests */
Listen(listenfd, LISTENQ);
return listenfd;
}
```

# Echo server: open\_listenfd() (setsockopt)

---

- The socket can be given some attributes.

```
/* eliminates "Address already in use" error
   from bind. */
optval = 1;
Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
            (const void *)&optval , sizeof(int));
```

## Echo server: `open_listenfd()` (`setsockopt`)

---

- Handy trick that allows us to rerun the server immediately after we kill it.
  - Otherwise we would have to wait about 15 secs.
  - Eliminates "Address already in use" error from `bind()`.
  - Strongly suggest you do this for all your servers to simplify debugging.

## Echo server: open\_listenfd()

---

- Next, we initialize the socket with the server's Internet address (IP address and port)

```
/* server's socket addr */
struct sockaddr_in serveraddr;

/* listenfd will be an endpoint for all requests
   to port on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons((unsigned short)port);
```

## Echo server: `open_listenfd()` (initialize)

---

- IP addr and port stored in network  
(big-endian) byte order
  - `htonl()` converts longs from host byte order to network byte order.
  - `htons()` converts shorts from host byte order to network byte order.

## Echo server: open\_listenfd() (bind)

---

- **bind()** associates the socket with the socket address we just created.

```
int listenfd;                                /* listening socket */

struct sockaddr_in serveraddr; /*server's socket addr*/

/* listenfd will be an endpoint for all requests
   to port on any IP address for this host */

Bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr));
```

## Echo server: open\_listenfd (listen)

---

- `listen()` indicates that this socket will accept connection (`connect`) requests from clients.

```
int listenfd;                                /* listening socket */  
  
/* make listenfd it a server-side listening socket  
ready to accept connection requests from clients */  
Listen(listenfd, LISTENQ);
```

- We're finally ready to enter the main server loop that accepts and processes client connection requests.

# Echo server: main loop

---

- The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

```
main() {  
    /* create and configure the listening socket */  
  
    while(1) {  
        /* Accept(): wait for a connection request */  
        /* echo(): read/echo input line from client */  
        /* Close(): close the connection */  
    }  
}
```

# Echo server: accept()

---

- `accept()` blocks waiting for a connection request

```
int listenfd; /* listening descriptor */
int connfd;   /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = Accept(listenfd,
                (SA *)&clientaddr, &clientlen);
```

## Echo server: accept()

---

- `accept()` returns a connected socket descriptor (`connfd`) with the same properties as the listening descriptor (`listenfd`)
  - Returns when connection between client and server is complete.
  - All I/O with the client will be done via the connected socket.
- `accept()` also fills in client's address.

# Echo server: identifying the client

---

- The server can determine the domain name and IP address of the client.

```
struct hostent *hp; /* pointer to DNS host entry */
char *haddrp; /* pointer to dotted decimal string */

hp = Gethostbyaddr((const char *)
    &clientaddr.sin_addr.s_addr,
    sizeof(clientaddr.sin_addr.s_addr), AF_INET);
haddrp = inet_ntoa(clientaddr.sin_addr);
printf("server connected to %s (%s)\n",
    hp->h_name, haddrp);
```

## Echo server: echo()

---

- The server uses Unix I/O to read and echo text lines until EOF (end-of-file) is encountered.
  - EOF notification caused by client calling `close(clientfd)` .
  - NOTE: EOF is a condition, not a data byte.

# Echo server: echo()

---

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio

    Rio_readinitb(&rio, connfd)
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) !=0) {
        printf("server received %d bytes\n", n);
        Rio_writen(connfd, buf, n);
    }
}
```

# Networking Programming

# Outline

---

- Web servers
  - HTTP Protocol
  - Web Content
  - CGI
- The Tiny Web Server
- Suggested Reading:
  - 11.5~11.6

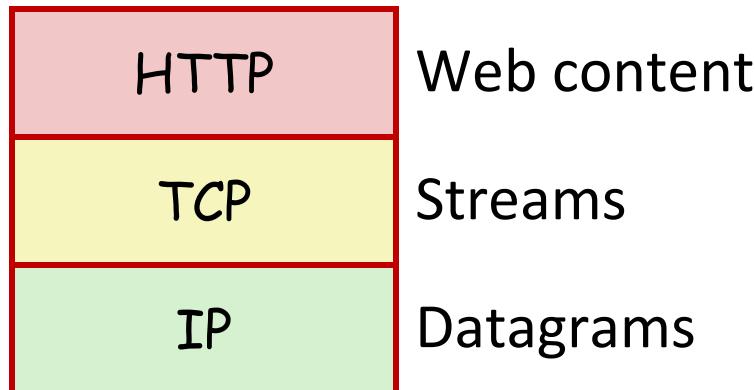
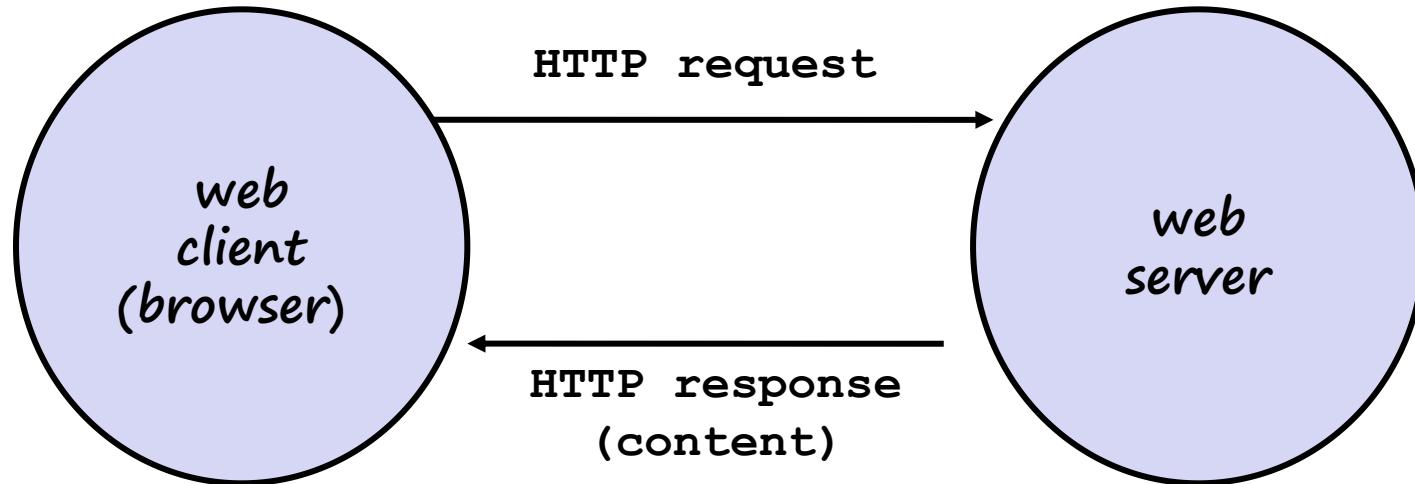
# Web servers

---

- Clients and servers communicate using the HyperText Transfer Protocol (HTTP)
  - Client and server establish TCP connection
  - Client requests content
  - Server responds with requested content
  - Client and server close connection (usually)
- Current version is HTTP/1.1
  - RFC 2616, June, 1999.

# Web servers

---



# Web content

- Web servers return **content** to clients
    - content: a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type
  - Example MIME types
    - text/html HTML page
    - text/plain Unformatted text
    - application/postscript Postscript document
    - image/gif Binary image encoded in GIF format
    - image/jpg Binary image encoded in JPG format

# Static and dynamic content

---

- The content returned in HTTP responses can be either static or dynamic
  - Static content:
    - Content stored in files and retrieved in response to an HTTP request
    - Examples: HTML files, images, audio clips.
  - Dynamic content:
    - Content produced on-the-fly in response to an HTTP request
    - Example: content produced by a program executed by the server on behalf of the client.

# URLs

---

- Each file managed by a server has a unique name called a **URL** (Universal Resource Locator)
- URLs for static content:
  - identifies a file called `index.html`, managed by a Web server at `ipads.se.sjtu.edu.cn` that is listening on port 80.

`http://ipads.se.sjtu.edu.cn:80/courses/ics/index.html`

`http://ipads.se.sjtu.edu.cn:/courses/ics/index.html`

`http://ipads.se.sjtu.edu.cn/courses/ics/`

# URLs

---

- URLs for dynamic content:
  - Identifies an executable file called **adder**, managed by a Web server at `www.cs.cmu.edu` that is listening on port 8000, that should be called with two argument strings: 15000 and 213.

`http://www.cs.cmu.edu:8000/cgi-bin/adder?15000&213`

# How clients and servers use URLs

---

- Example URL:

`http://www.aol.com:80/index.html`

- Clients use prefix (`http://www.aol.com:80`) to infer:
  - What kind of server to contact (**Web server**)
  - Where the server is (**www.aol.com**)
  - What port it is listening on (**80**)

# How clients and servers use URLs

---

- Servers use suffix (/index.html) to:
  - Determine if request is for static or dynamic content.
    - No hard and fast rules for this.
    - Convention: executables reside in cgi-bin directory
  - Find file on filesystem
    - Initial "/" in suffix denotes **home directory** for requested content.
    - Minimal suffix is "/", which all servers expand to some default home page (e.g., index.html).

# Anatomy of an HTTP transaction

---

//Client: open connection to server

unix> telnet ipads.se.sjtu.edu.cn 80

//Telnet prints 3 lines to the terminal

Trying 202.120.40.88...

Connected to ipads.se.sjtu.edu.cn.

Escape character is '^]'.

//Client: request line

GET /courses/ics/index.shtml HTTP/1.1

//Client: required HTTP/1.1 HOST header

host: ipads.se.sjtu.edu.cn

//Client: empty line terminates headers

# Anatomy of an HTTP transaction

---

//Server: response line

**HTTP/1.1 200 OK**

//Server: followed by five response headers

**Server: nginx/1.0.4**

**Date: Thu, 29 Nov 2012 10:15:38 GMT**

//Server: expect HTML in the response body

**Content-Type: text/html**

//Server: expect 11,560 bytes in the resp body

**Content-Length: 11560**

...

//Server: empty line ("\r\n") terminates hdrs

# Anatomy of an HTTP transaction

---

```
//Server: first HTML line in response body
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html>
...
//Server: 292 lines of HTML not shown.
</html>
//Server: last HTML line in response body
//Server: closes connection
Connection closed by foreign host.
Client: closes connection and terminates
unix>
```

# HTTP Requests

---

- HTTP request is a **request line**, followed by zero or more **request headers**
- Request line: **<method> <uri> <version>**
  - **<version>** is HTTP version of request (HTTP/1.0 or HTTP/1.1)
  - **<uri>** is typically URL for proxies, URL suffix for servers.
    - A URL is a type of URI (Uniform Resource Identifier)
    - See <http://www.ietf.org/rfc/rfc2396.txt>
  - **<method>** is either GET, POST, OPTIONS, HEAD, PUT, DELETE, or TRACE.

# HTTP Requests (cont)

---

- HTTP methods:
  - **GET**: Retrieve static or dynamic content
    - Arguments for dynamic content are in URI
    - Workhorse method (99% of requests)
  - **POST**: Retrieve dynamic content
    - Arguments for dynamic content are in the request body
  - OPTIONS, HEAD, PUT, DELETE, TRACE
- Request headers: <**header name**>: <**header data**>
  - Provide additional information to the server
  - e.g. host: ipads.se.sjtu.edu.cn

# HTTP Responses

---

- HTTP response is a **response line** followed by zero or more **response headers**.
- Response line: **<version> <status code> <status msg>**
  - **<version>** is HTTP version of the response.
  - **<status code>** is numeric status.
  - **<status msg>** is corresponding English text.

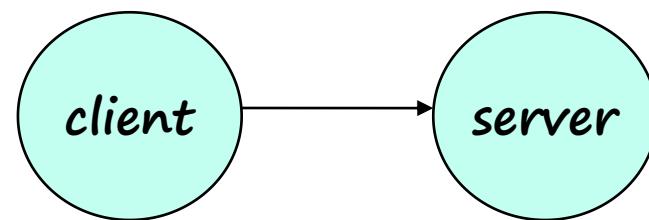
200 OK	Request was handled without error
301 Moved	Provide alternate URL
403 Forbidden	Server lacks permission to access file
404 Not found	Server couldn't find the file.
- Response headers: **<header name>: <header data>**
  - Provide additional information about response
  - **Content-Type**: MIME type of content in response body
  - **Content-Length**: Length of content in response body

# Serving dynamic content

---

- Client sends request to server
- If request URI contains the string “/cgi-bin”, then the server assumes that the request is for dynamic content.

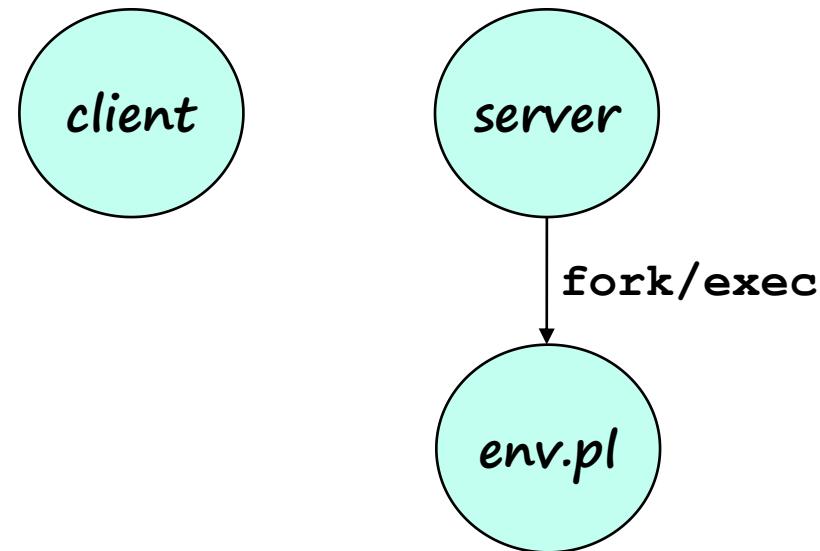
`GET /cgi-bin/env.pl HTTP/1.1`



# Serving dynamic content

---

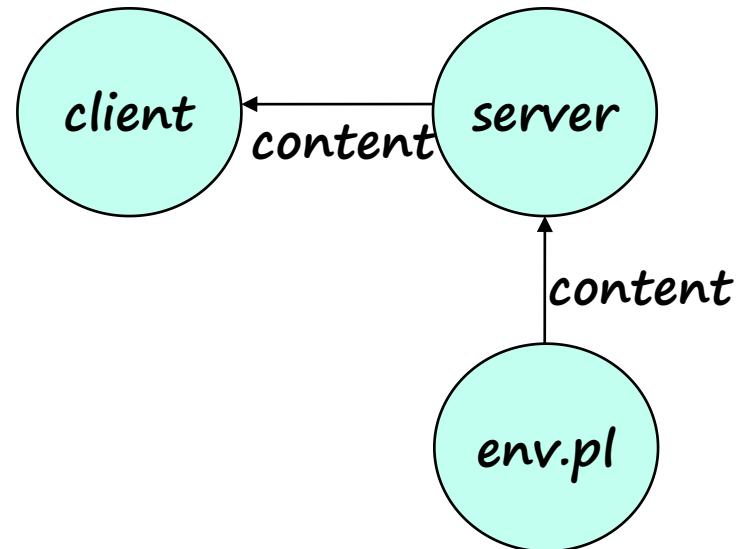
- The server creates a child process and runs the program identified by the URI in that process



# Serving dynamic content

---

- The child runs and generates the dynamic content.
- The server captures the content of the child and forwards it without modification to the client



# Serving dynamic content

---

- 1. How does the client pass program arguments to the server?
- 2. How does the server pass these arguments to the child?
- 3. How does the server pass other info relevant to the request to the child?

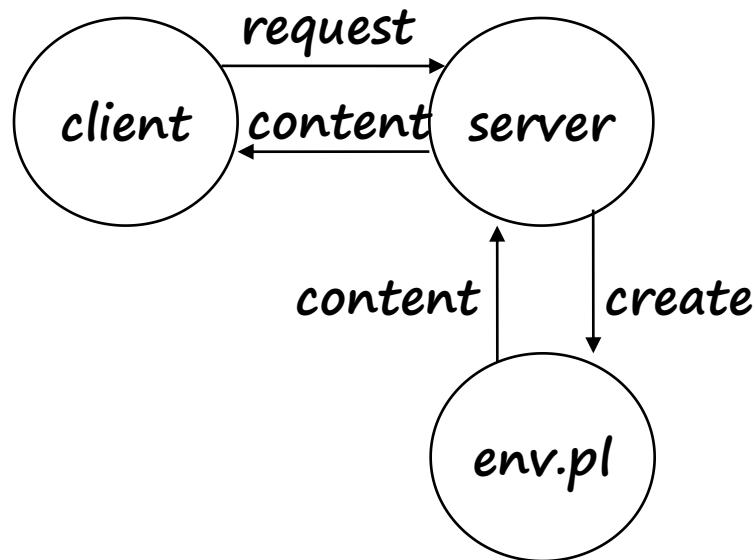
# Serving dynamic content

---

- 4. How does the server capture the content produced by the child?
- These issues are addressed by the Common Gateway Interface (CGI) specification.

# Serving dynamic content

---



# Serving dynamic content

---

- Question:
  - How does the client pass arguments to the server?
- Answer:
  - The arguments are appended to the URI
    - `http://add.com/cgi-bin/adder?1&2`
    - adder is the CGI program on the server that will do the addition.
    - argument list starts with “?”
    - arguments separated by “&”
    - spaces represented by “%20”

# Serving dynamic content

---

- Question:
  - How does the server pass these arguments to the child?
- Answer:
  - In environment variable QUERY\_STRING
    - a single string containing everything after the "?"
    - for add.com: QUERY\_STRING = "1&2"

# Serving dynamic content

---

```
if ((buf = getenv("QUERY_STRING")) != NULL) {  
    p = strchr(buf, '&');  
    *p = '\0';  
    strcpy(arg1, buf);  
    strcpy(arg2, p+1);  
    n1 = atoi(arg1);  
    n2 = atoi(arg2);  
}
```

# Serving dynamic content

---

- Question:
  - How does the server pass other info relevant to the request to the child?
- Answer:
  - in a collection of environment variables defined by the *CGI* spec.

# Some CGI environment variables

---

- Request-specific
  - QUERY\_STRING (**contains** GET args)
  - SERVER\_PORT
  - REQUEST\_METHOD (GET, POST, etc)
  - REMOTE\_HOST (**domain name of client**)
  - REMOTE\_ADDR (**IP address of client**)
  - CONTENT\_TYPE (**for** POST, **MIME type of the request body** )
  - CONTENT\_LENGTH (**for** POST, **length in bytes**)

# Serving dynamic content

---

- Question:
  - How does the server capture the content produced by the child?
- Answer:
  - The child generates its output on `stdout`. Server uses `dup2` to redirect `stdout` to its connected socket.
  - Notice that only the child knows the type and size of the content.
  - Thus the child (not the server) must generate the corresponding headers.

# Tiny Web Server

---

- Tiny Web server described in text
  - Tiny is a sequential Web server.
  - Serves static and dynamic content to real browsers.
    - text files, HTML files, GIF and JPEG images.
  - 226 lines of commented C code.
  - Not as complete or robust as a real web server

# Tiny Operation

---

- Read request from client
- Split into `method / uri / version`
  - If not GET, then return error
- If URI contains “`cgi-bin`” then serve dynamic content
  - (Would do wrong thing if had file “`abcgi.html`”)
  - Fork process to execute program
- Otherwise serve static content
  - Copy file to output

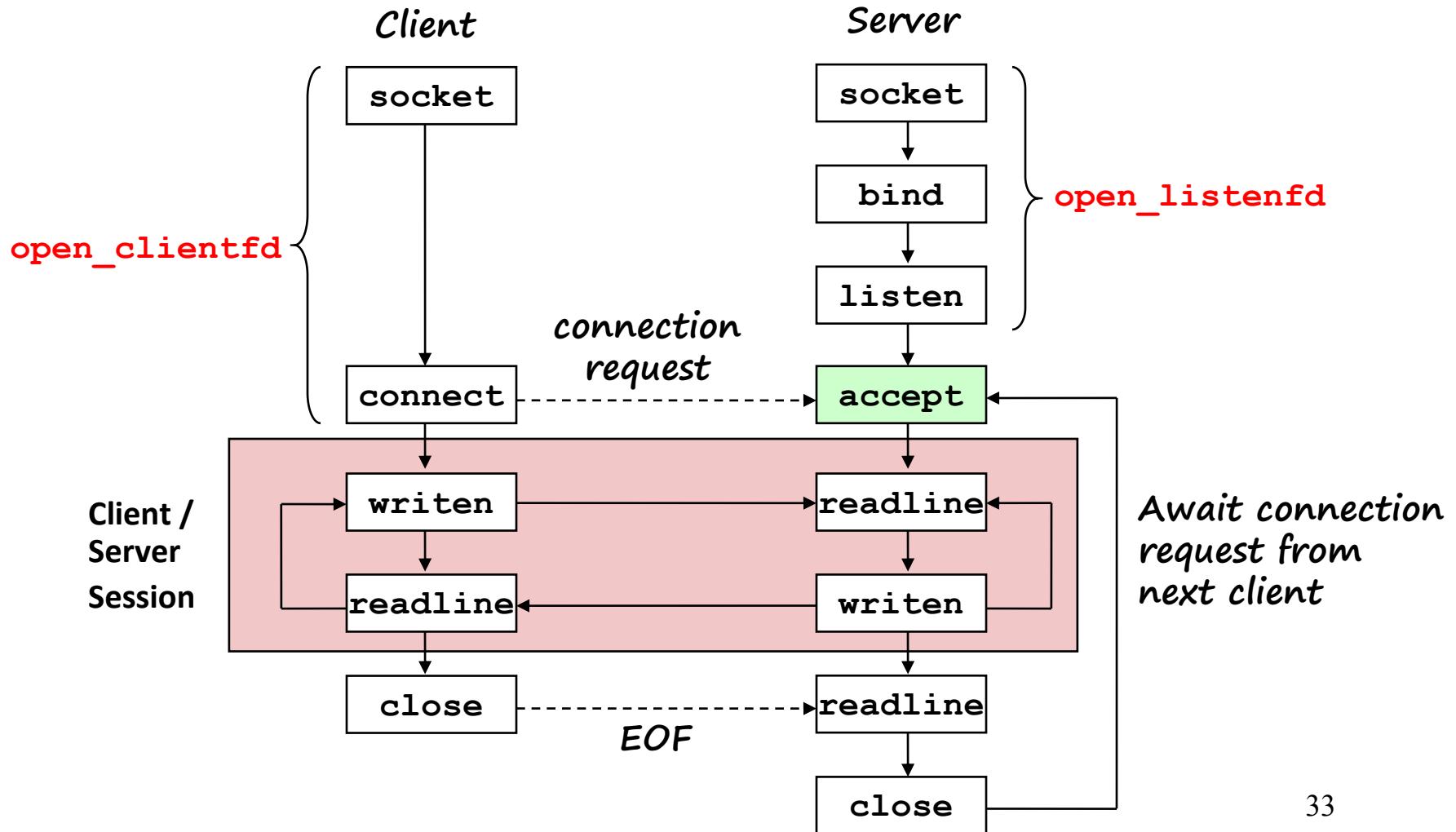
```
/* Domain Name Service (DNS) host entry */
struct hostent {
    char      *h_name;          /* official name of host */
    char      **h_aliases;       /* alias list */
    int       h_addrtype;        /* host address type */
    int       h_length;          /* length of address */
    char      **h_addr_list;     /* list of addresses */
};

/* Internet address */
struct in_addr {
    unsigned int s_addr;         /* 32-bit IP address */
};

/* Generic socket address structure
   (for connect, bind, and accept) */
struct sockaddr {
    unsigned short sa_family;   /* protocol family */
    char      sa_data[14];       /* address data */
};
```

```
/* Internet style socket address */
struct sockaddr_in {
    /* Address family (AF_INET) */
    unsigned short sin_family;
    unsigned short sin_port;      /* Port number */
    struct in_addr sin_addr;     /* IP address */
    /* Pad to sizeof "struct sockaddr" */
    unsigned char  sin_zero[8];
};
```

# Overview of the Sockets Interface



```
struct hostent *genhostbyname(const char *name) ;
struct hostent *genhostbyaddr(
                           const char *addr, int len,0) ;

/* client-side socket interface */
int socket(int domain, int type, int protocol) ;
int connect(int sockfd,
            struct sockaddr *serv_addr,
            int addr_len) ;

/* server-side socket interface */
int bind(int sockfd,
          struct sockaddr *my_addr,
          int addrlen) ;
int listen(int sockfd, int backlog) ;
int accept(int listenfd,
           struct sockaddr *addr,
           int addrlen) ;
```

# The Tiny Web Server

---

```
1  /*
2  * tiny.c - A simple HTTP/1.0 Web server that uses the
3  * GET method to serve static and dynamic content.
4  */
5 #include "csapp.h"
6
7 void doit(int fd);
8 void read_requesthdrs(int fd);
9 int parse_uri(char *uri, char *fname, char *cgiargs);
10 void serve_static(rio_t *rio, char *fname, int fsize);
11 void get_filetype(char *fname, char *ftype);
12 void serve_dynamic(rio_t *rio, char *fname, char *cgiargs);
13 void clienterror(rio_t *rio, char *cause, char *errnum,
14 char *shortmsg, char *longmsg);
```

# The Tiny Web Server

---

```
16 int main(int argc, char **argv)
17 {
18     int listenfd, connfd, port, clientlen;
19     struct sockaddr_in clientaddr;
20
21     /* check command line args */
22     if (argc != 2) {
23         fprintf(stderr,"usage: %s <port>\n", argv [0]);
24         exit(1);
25     }
26     port = atoi(argv[1]);
27 }
```

# The Tiny Web Server

---

```
28     listenfd = open_listenfd(port) ;
29     while (1) {
30         clientlen = sizeof(clientaddr) ;
31         connfd = Accept(listenfd,
32                         (SA *) &clientaddr, &clientlen) ;
33         doit(connfd) ;
34     }
35 }
```

# The Tiny Web Server

---

```
1 void doit(int fd)
2 {
3     int is_static;
4     struct stat sbuf;
5     char buf[MAXLINE], method[MAXLINE];
6     char uri[MAXLINE], version[MAXLINE];
7     char filename[MAXLINE], cgiargs[MAXLINE];
8     rio_t rio;
```

GET

/courses/ics

HTTP/1.1

# The Tiny Web Server

method

uri

version

```
9  /* read request line and headers */
10 Rio_readinitb(&rio, fd)
11 Rio_readlineb(&rio, buf, MAXLINE);
12 sscanf(buf, "%s %s %s\n", method, uri, version);
13 if (strcasecmp(method, "GET")) {
14     clienterror(fd, method, "501", "Not Implemented",
15                 "Tiny does not implement this method");
16     return;
17 }
18 read_requesthdrs(&rio);
19
```

# The Tiny Web Server

---

```
20  /* parse URI from GET request */
21  is_static = parse_uri(uri, filename, cgiargs);
22  if (stat(filename, &sbuf) < 0) {
23      clienterror(fd, filename, "404", "Not found",
24                  "Tiny couldn't find this file");
25      return;
26  }
27
28  if (is_static) { /* serve static content */
29      if (!(S_ISREG(sbuf.st_mode)) ||
30          !(S_IRUSR & sbuf.st_mode)) {
31          clienterror(fd, filename, "403",
32                      "Forbidden", "Tiny couldn't read the file");
33          return;
34      }
35      serve_static(&rio, filename, sbuf.st_size);40
36  }
```

# The Tiny Web Server

---

```
34 else /* serve dynamic content */  
35     if (!(S_ISREG(sbuf.st_mode)) ||  
           !(S_IXUSR & sbuf.st_mode)) {  
36         clienterror(fd, filename, "403", "Forbidden",  
37                     "Tiny couldn't run the CGI program");  
38         return;  
39     }  
40     serve_dynamic(&rio, filename, cgiargs);  
41 }  
42 }
```

```
1 void clienterror(rio_t *fd, char *cause, char *errnum,
2                               char *shortmsg, char *longmsg)
3 {
4     char buf[MAXLINE], body[MAXBUF];
5
6     /* build the HTTP response body */
7     sprintf(body, "<html><title>Tiny Error</title>");
8     sprintf(body, "%s<body bgcolor=\"ffffff\">\r\n", body);
9     sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
10    sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
11    sprintf(body, "%s<hr><em>Tiny Web server</em>\r\n", body);
12
13    /* print the HTTP response */
14    sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
15    Rio_writen(fd, buf, strlen(buf));
16    sprintf(buf, "Content-type: text/html\r\n");
17    Rio_writen(fd, buf, strlen(buf));
18    sprintf(buf, "Content-length: %d\r\n\r\n", strlen(body));
19    Rio_writen(fd, buf, strlen(buf));
20    Rio_writen(fd, body, strlen(body));
21}
```

# The Tiny Web Server

---

```
1 void read_requesthdrs(rio_t *fd)
2 {
3     char buf[MAXLINE];
4
5     Readline(fd, buf, MAXLINE);
6     while(strcmp(buf, "\r\n"))
7         Rio_readlineb(fd, buf, MAXLINE);
8     return;
9 }
```

```
1 int parse_uri(char *uri, char *filename, char *cgiargs)
2 {
3     char *ptr;
4
5     if (!strstr(uri, "cgi-bin")) { /* static content */
6         strcpy(cgiargs, "");
7         strcpy(filename, ".");
8         strcat(filename, uri);
9         if (uri[strlen(uri)-1] == '/')
10             strcat(filename, "home.html");
11         return 1;
12     }
13     else { /* dynamic content */
14         ptr = index(uri, '?');
15         if (ptr) {
16             strcpy(cgiargs, ptr+1);
17             *ptr = '\0';
18         } else
19             strcpy(cgiargs, "");
20         strcpy(filename, ".");
21         strcat(filename, uri);
22     }
23     return 0;
24 }
25}
```

```
1 void serve_static(int fd, char *filename, int filesize)
2 {
3     int srcfd;
4     char *srcp, ftype[MAXLINE], buf[MAXBUF];
5
6     /* send response headers to client */
7     get_filetype(filename, ftype);
8     sprintf(buf, "HTTP/1.0 200 OK\r\n");
9     sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
10    sprintf(buf, "%sContent-length: %d\n", buf, filesize);
11    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, ftype);
12    Rio_writen(fd, buf, strlen(buf));
13
14     /* send response body to client */
15     srcfd = Open(filename, O_RDONLY, 0);
16     srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
17     Close(srcfd);
18     Rio_writen(fd, srcp, filesize);
19     Munmap(srcp, filesize);
20 }
21
```

# The Tiny Web Server

---

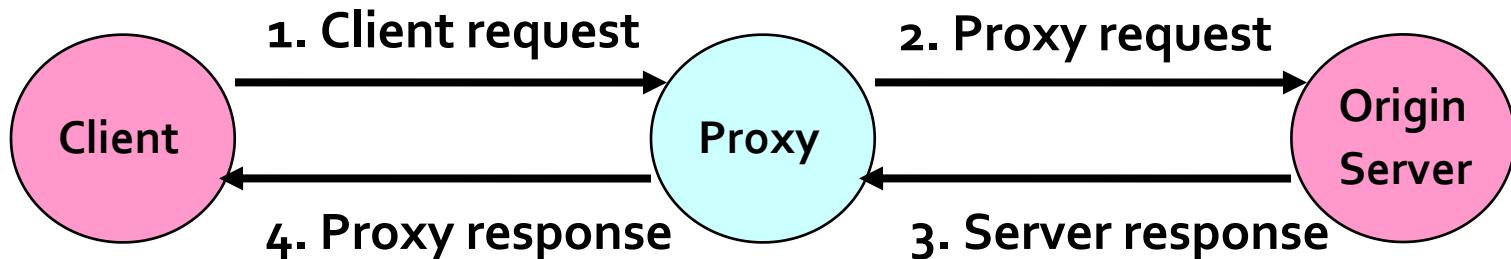
```
22 /*  
23 * get_filetype - derive file type from file name  
24 */  
25 void get_filetype(char *filename, char *filetype)  
26 {  
27     if (strstr(filename, ".html"))  
28         strcpy(filetype, "text/html");  
29     else if (strstr(filename, ".gif"))  
30         strcpy(filetype, "image/gif");  
31     else if (strstr(filename, ".jpg"))  
32         strcpy(filetype, "image/jpg");  
33     else  
34         strcpy(filetype, "text/plain");  
35 }
```

```
1 void serve_dynamic(int fd, char *filename, char *cgiargs)
2 {
3     char buf[MAXLINE];
4
5     /* return first part of HTTP response */
6     sprintf(buf, "HTTP/1.0 200 OK\r\n");
7     Rio_writen(fd, buf, strlen(buf));
8     sprintf(buf, "Server: Tiny Web Server\r\n");
9     Rio_writen(fd, buf, strlen(buf));
10
11    if (Fork() == 0) { /* child */
12        /* real server would set all CGI vars here */
13        setenv("QUERY_STRING", cgiargs, 1);
14        Dup2(fd, STDOUT_FILENO); /* redirect output to client*/
15        Execve(filename, NULL, environ); /* run CGI program */
16    }
17    Wait(NULL); /* parent reaps child */
18}
```

# Proxies

---

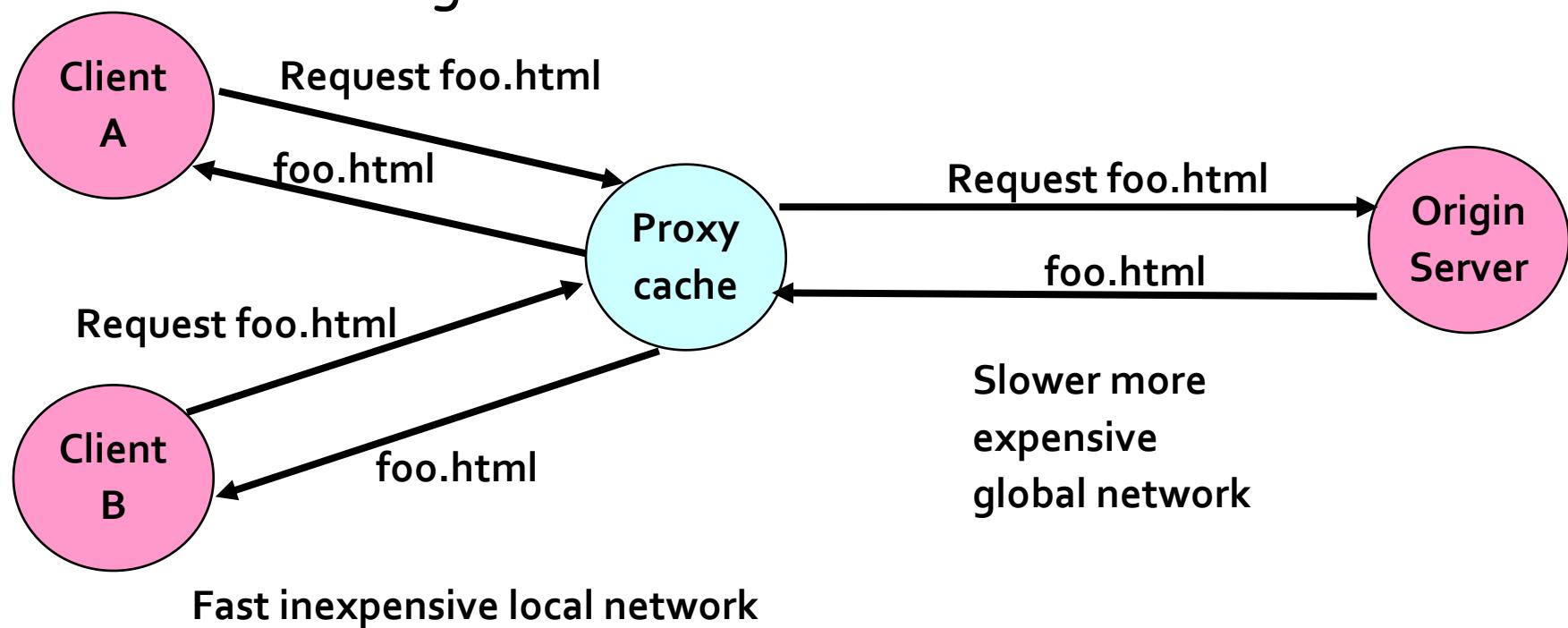
- A **proxy** is an intermediary between a client and an **origin server**.
  - To the client, the proxy acts like a server.
  - To the server, the proxy acts like a client.



# Why Proxies?

---

- Can perform useful functions as requests and responses pass by
  - Examples: Caching, logging, anonymization, filtering, transcoding



# Concurrent Programming

# Outline

---

- Topics:
  - Concurrent programming with processes
  - Concurrent programming with threads
  - Concurrent programming with I/O multiplexing
- Suggested reading:
  - 12.1~12.3

# Concurrency

---

- Concurrency is a general phenomenon
  - Hardware exception handlers
  - Processes
  - Unix signal handlers

# Concurrency

---

- Application-level concurrency is useful
  - Responding to asynchronous events
  - Computing in parallel on multiprocessor
  - Accessing slow I/O devices
  - Interacting with humans
  - Reducing latency by deferring work
  - Servicing multiple network clients

# Concurrency

---

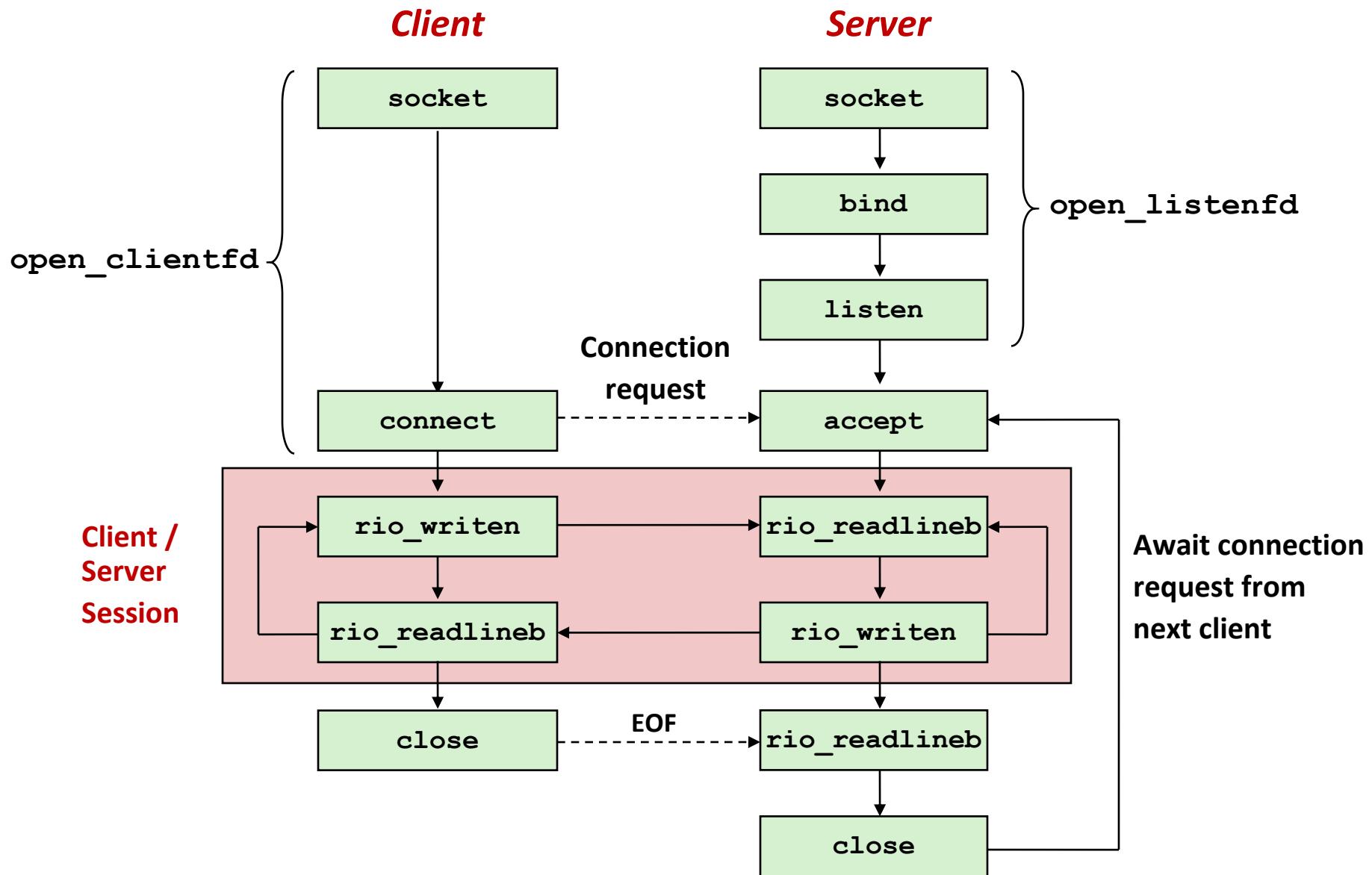
- Concurrent programming
  - Applications that use application-level concurrency
- Three basic approaches for concurrent programming
  - Processes
  - Threads
  - I/O multiplexing

# Concurrency

---

- Concurrent programming is hard
  - The human mind tends to be sequential
  - Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible
  - Classical problem classes of concurrent programs
    - Data races, deadlock, and livelock/starvation

# Iterative Echo Server



# Review: Iterative Echo Server

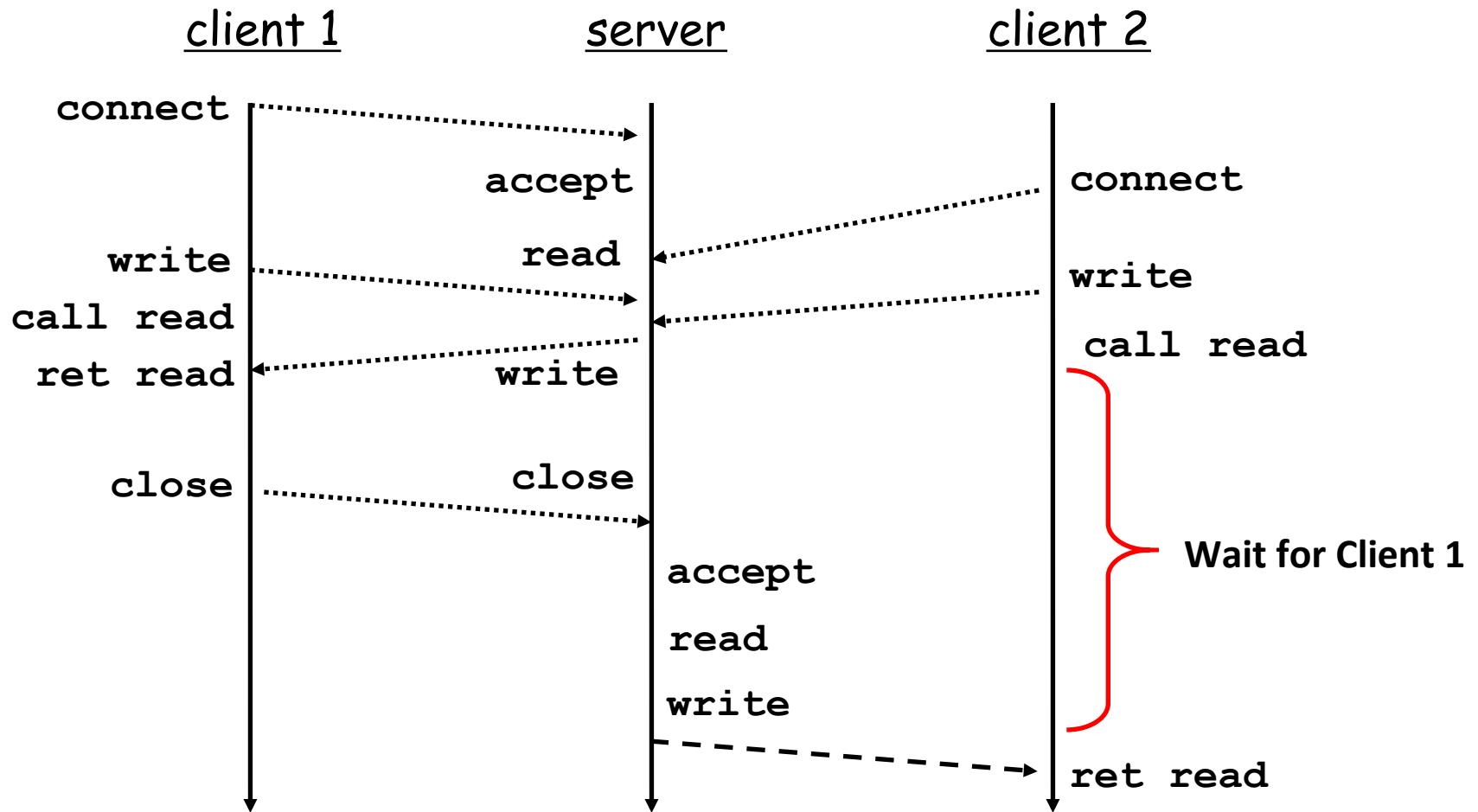
```
int main(int argc, char **argv)
{
    int listenfd, connfd, port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);

    listenfd = Open_listenfd(port);
    while (1) {
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

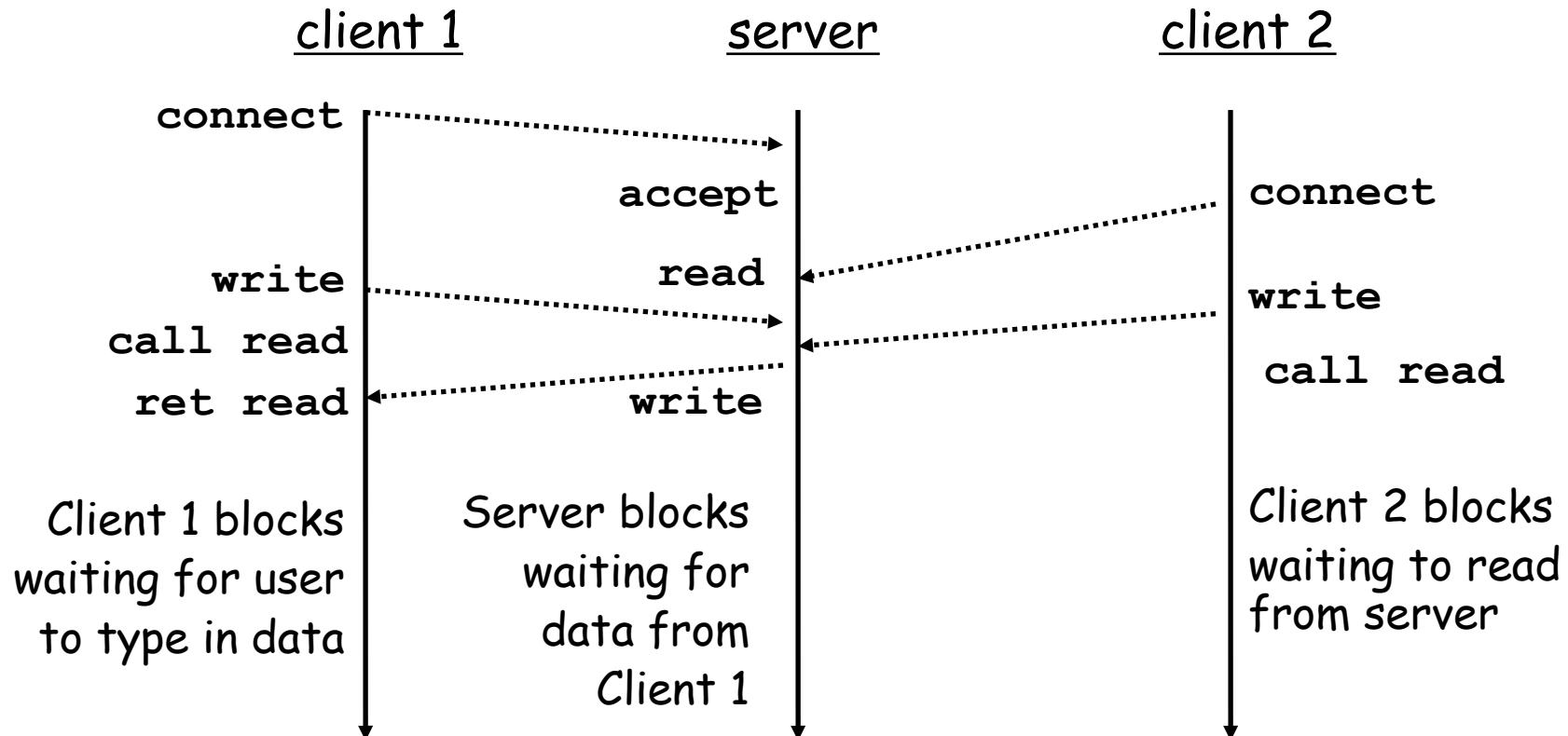
- Accept a connection request
- Handle echo requests until client terminates

# Iterative Servers

- Iterative servers process one request at a time

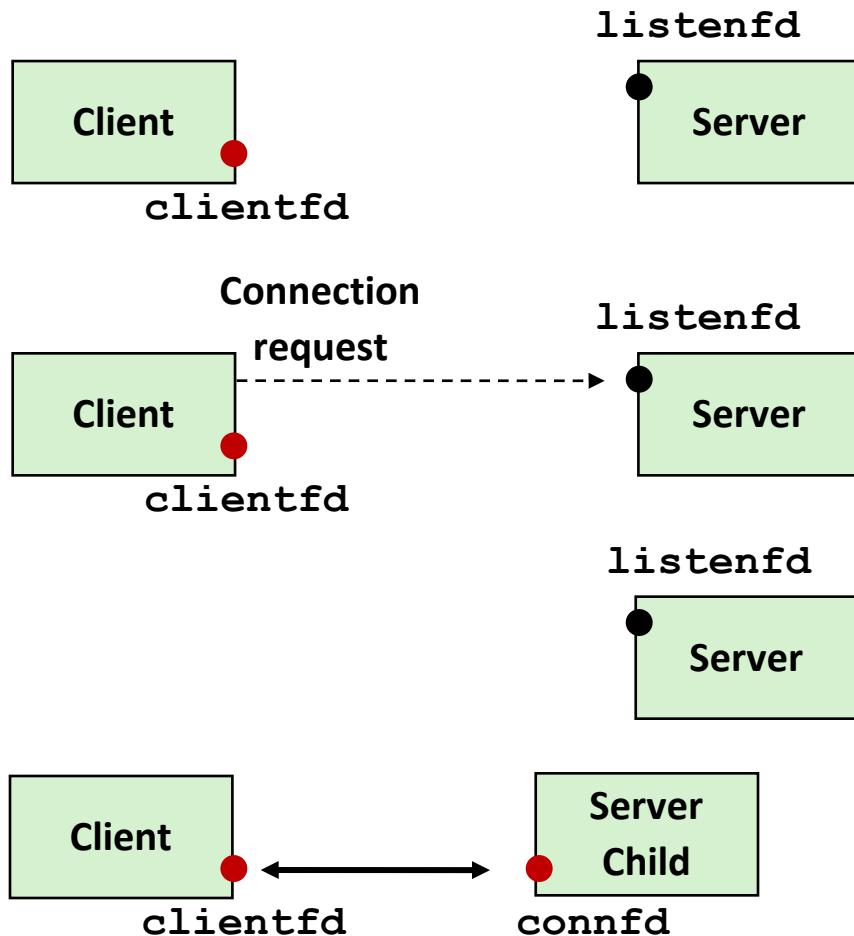


# Fundamental Flaw of Iterative Servers



- Solution: use **concurrent** servers instead
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

# Concurrent Programming with processes



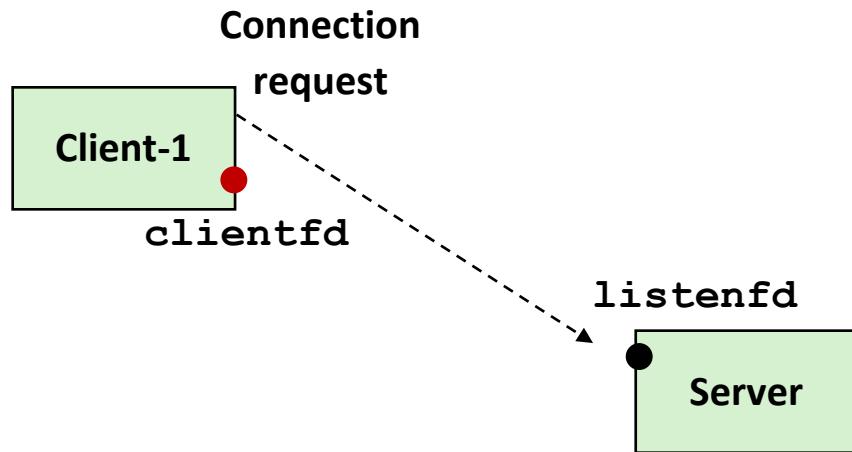
1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. Client makes connection request by calling and blocking in `connect`

3. Server returns `connfd` from `accept`. Forks child to handle client. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

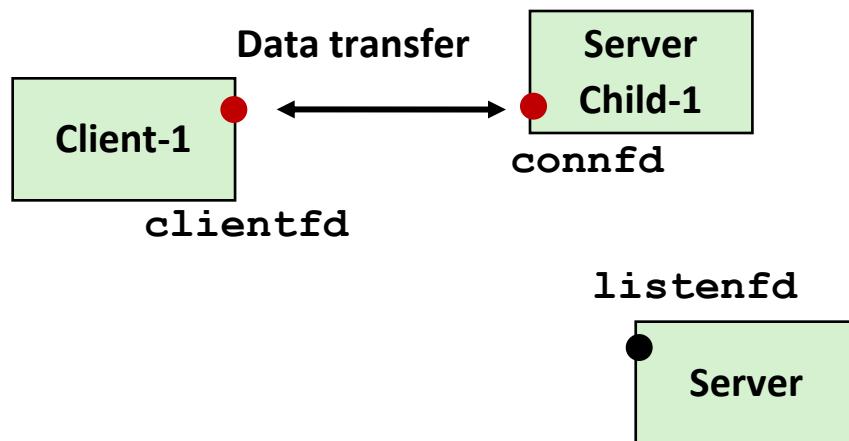
# Concurrent Programming with processes

---



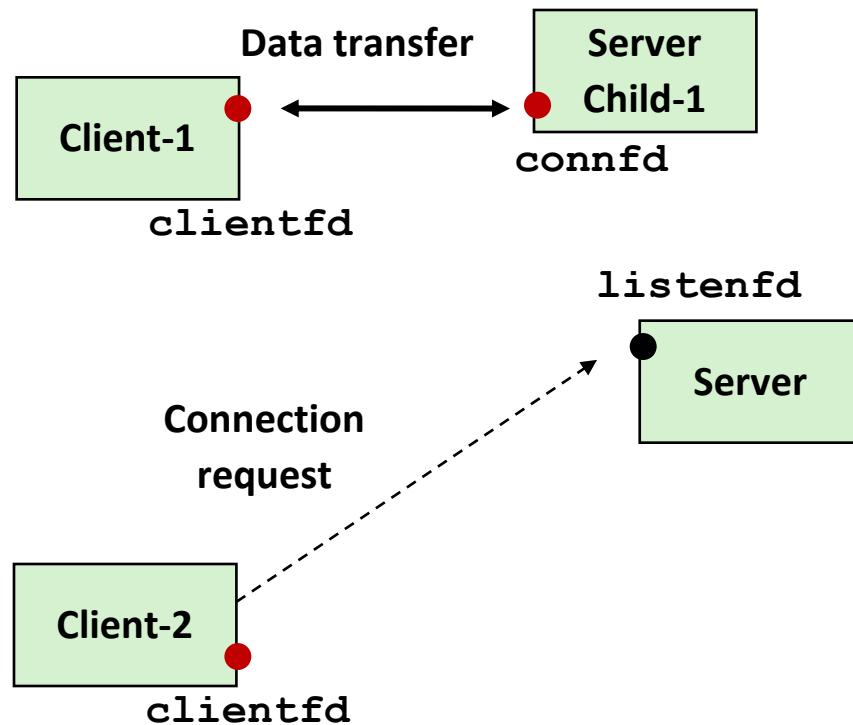
# Concurrent Programming with processes

---



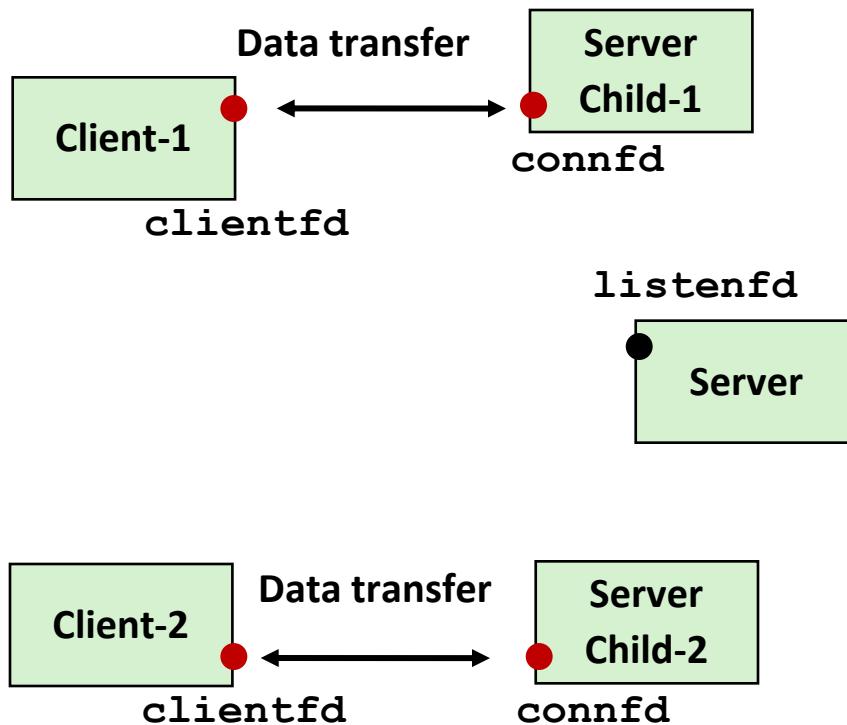
# Concurrent Programming with processes

---



# Concurrent Programming with processes

---



# Process-Based Concurrent Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd, port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(port);
    while (1) {
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);      /* Child services client */
            Close(connfd);    /* Child closes connection with client */
            exit(0);           /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

- Fork separate process for each client
- Does not allow any communication between different client handlers

# Concurrent Programming with Processes

---

- Kernel provides multiple control flows with separate address spaces.
- Standard Unix process control and signals.
- Explicit interprocess communication mechanism

# Implementation issues with process

---

- Server should restart accept call if it is interrupted by a transfer of control to the SIGCHLD handler
- Server must reap zombie children
  - to avoid fatal memory leak

```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) >
0)
        ;
    return;
}
```

# Implementation issues with process

---

- Server must close its copy of `connfd`.
  - Kernel keeps reference for each socket.
  - After fork, `refcnt(connfd) = 2`.
  - Connection will not be closed until `refcnt(connfd)=0`.

# Pros and cons of process

---

- + Handles multiple connections concurrently
- + Clean sharing model
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- + Simple and straightforward.

# Pros and cons of process

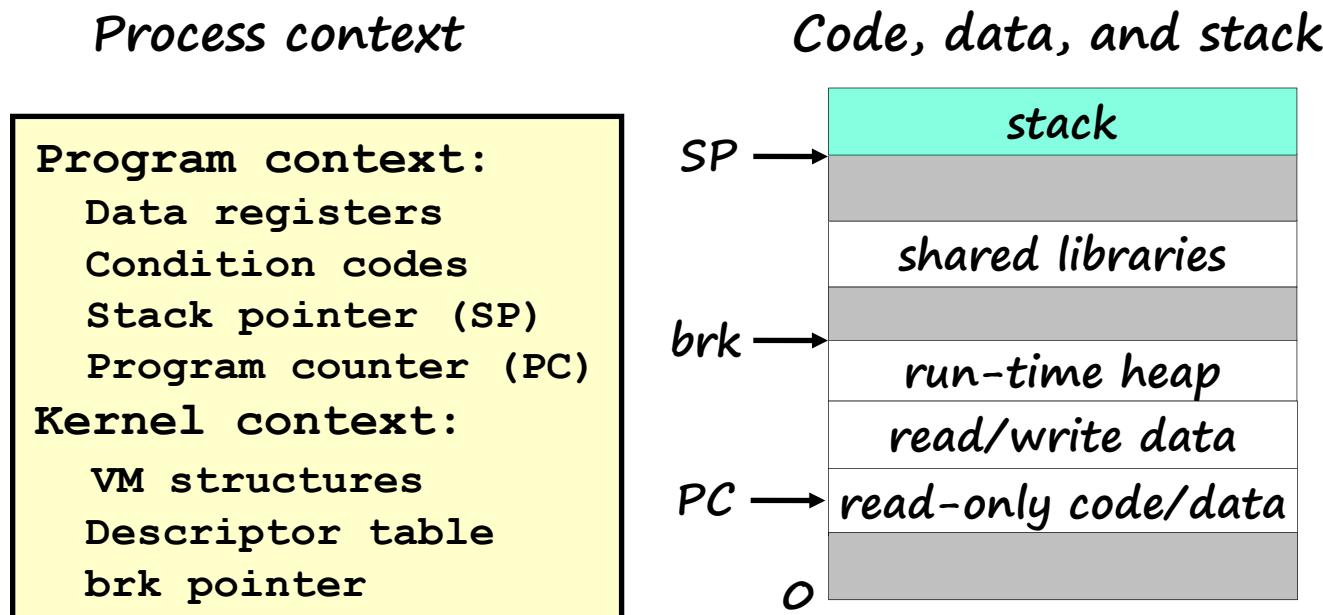
---

- - Additional overhead for process control.
- - Nontrivial to share data between processes.
  - Requires IPC (inter-process communication) mechanisms
    - FIFO's (named pipes), System V shared memory and semaphores

# Traditional view of a process

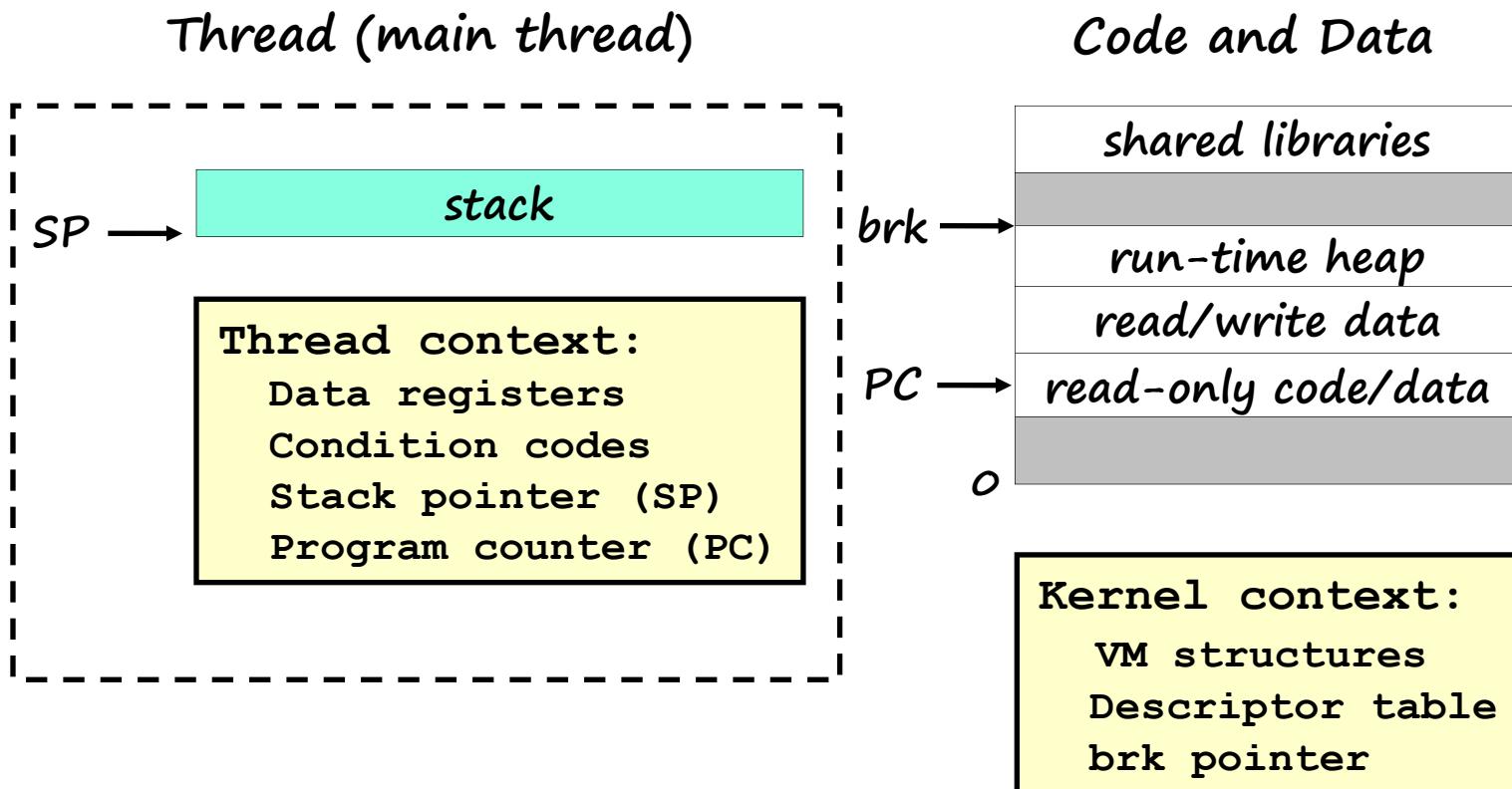
---

- Process = process context + code, data, and stack



# Alternate view of a process

- Process = thread + code, data, and kernel context



# A process with multiple threads

---

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow (sequence of PC values)
  - Each thread shares the same **code, data, and kernel context**
  - Each thread has its own **thread id (TID)**

# A process with multiple threads

---

Thread 1 (main thread)

stack 1

Thread 1 context:

Data registers

Condition codes

SP1

PC1

Shared code and data

shared libraries

run-time heap

read/write data

read-only code/data

Thread 2 (peer thread)

stack 2

Thread 2 context:

Data registers

Condition codes

SP2

PC2

Kernel context:

VM structures

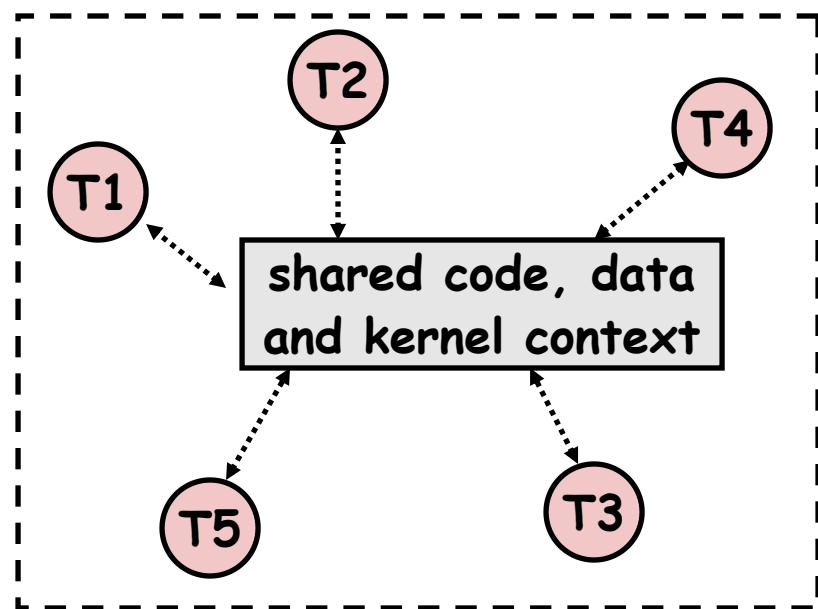
Descriptor table

brk pointer

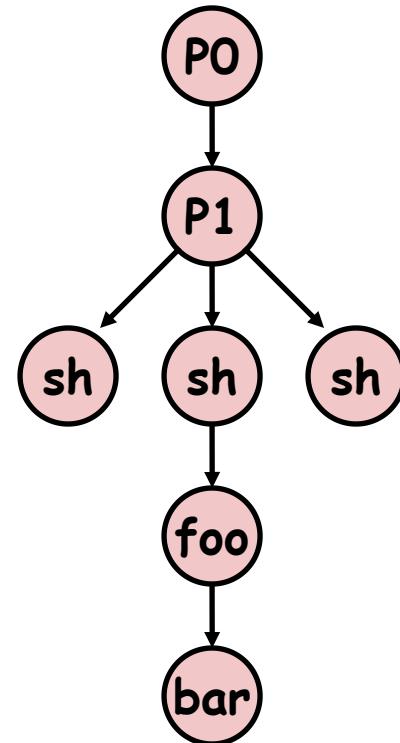
# Logical View of Threads

- Threads associated with process form a pool of peers
  - Unlike processes which form a tree hierarchy

Threads associated with process foo



Process hierarchy



# Threads vs. Processes

---

- How threads and processes are **similar**
  - Each has its own logical control flow
  - Each can run **concurrently** with others (possibly on different cores)
  - Each is context switched

# Threads vs. Processes

---

- How threads and processes are **different**
  - Threads share code and some data
    - Processes (typically) do not
  - Threads are somewhat **less expensive** than processes
    - Process control (creating and reaping) is twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

# Posix threads (Pthreads) interface

---

- Pthreads: Standard interface for ~60 functions
  - Manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create`
    - `pthread_join`
  - Determining your thread ID
    - `pthread_self`
  - Terminating threads
    - `pthread_cancel`
    - `pthread_exit`
    - `exit` [terminates all threads]
    - `return` [terminates current thread]

# The Pthreads "hello, world" Program

```
/* hello.c - Pthreads "hello, world" program */
#include "csapp.h"

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}

int main() {
    pthread_t tid;

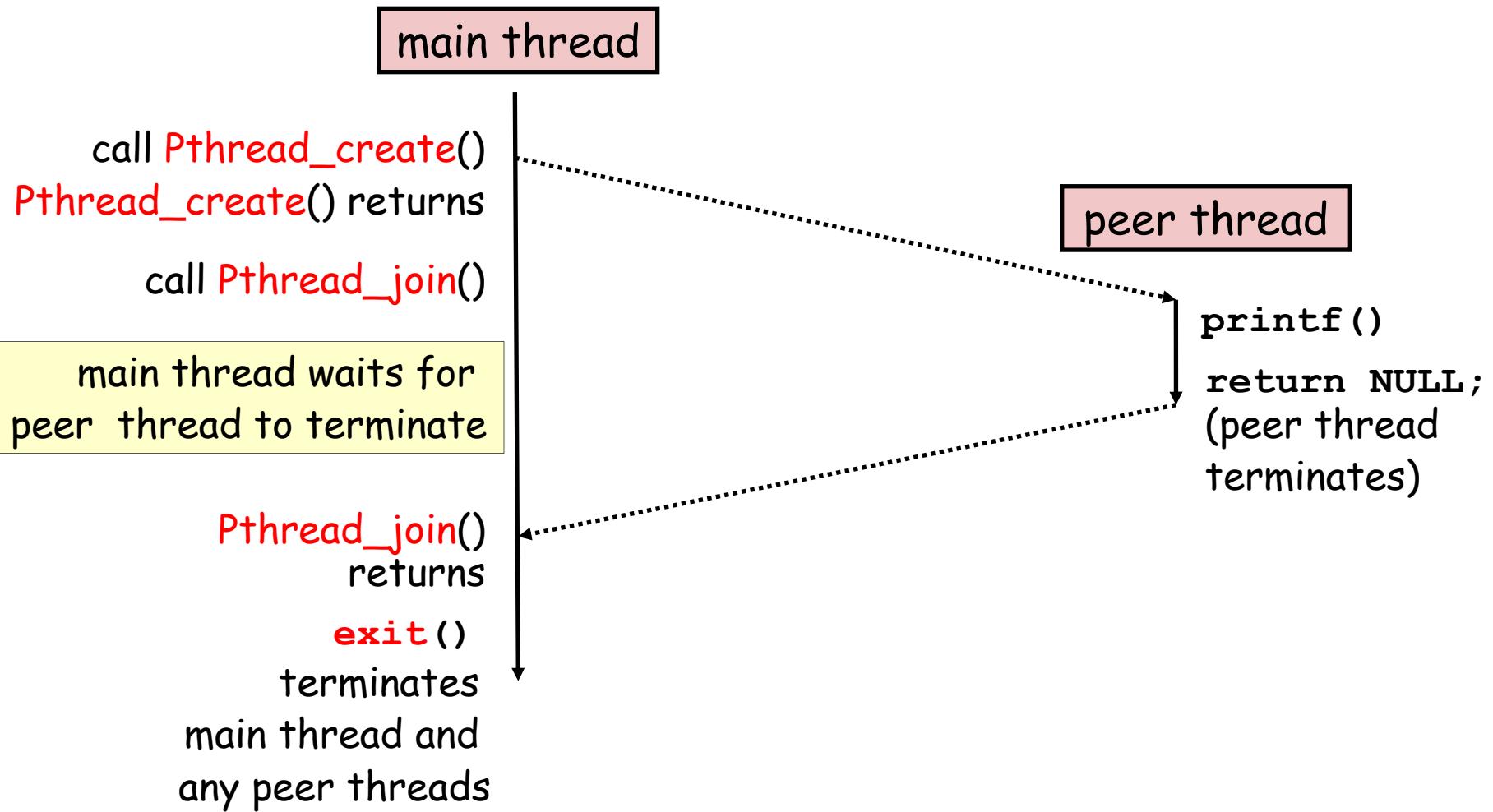
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

The diagram illustrates the arguments passed to the `Pthread_create` and `Pthread_join` functions. It shows three boxes with arrows pointing to specific parameters in the code:

- A light blue box labeled "Thread attributes (usually NULL)" has an arrow pointing to the first parameter of `Pthread_create`.
- A light blue box labeled "return value (void \*\*p)" has an arrow pointing to the second parameter of `Pthread_create`.
- A light blue box labeled "Thread arguments (void \*p)" has an arrow pointing to the third parameter of `Pthread_create`. Another arrow points from this box to the argument of `Pthread_join`.

# Execution of Threaded "hello, world"

---



# Thread-based concurrent server (cont)

```
int main(int argc, char **argv) {
    int listenfd, *connfdp, port, clientlen;
    struct sockaddr_in clientaddr;
    pthread_t tid;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd,
                          (SA *)&clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

# Thread-based concurrent server (cont)

---

```
/* thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);

    Pthread_detach(pthread_self());
    Free(vargp);

    echo_r(connfd); /* reentrant version of echo() */
    Close(connfd);
    return NULL;
}
```

# Issues with thread-based servers

---

- Must run “detached” to avoid memory leak.
  - At any point in time, a thread is either **joinable** or **detached**
  - joinable thread can be reaped and killed by other threads.
    - must be reaped (with **pthread\_join**) to free memory resources.

# Issues with thread-based servers

---

- Must run “detached” to avoid memory leak.
  - Detached thread cannot be reaped or killed by other threads.
    - resources are **automatically reaped on termination**.
  - Default state is joinable.
    - use `pthread_detach(pthread_self())` to make detached.

# Issues with thread-based servers

---

- Must be careful to avoid unintended sharing.
  - For example, what happens if we pass the address of `connfd` to the thread routine?
  - `Pthread_create(&tid, NULL, thread, (void *) &connfd);`

# Pros and cons of thread-based designs

---

- + Easy to **share** data structures between threads
  - e.g., logging information, file cache.
- + Threads are more **efficient** than processes
- - **Unintentional sharing** can introduce subtle and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & with private

# I/O multiplexing: select() function

---

```
#include <sys/select.h>

int select (int maxfd, fd_set *readset,
            NULL, NULL, NULL) ;
```

Return nonzero count of ready descriptors, -1 on error

# I/O multiplexing: select() function

---

- `select()`
  - Sleeps until one or more file descriptors in the set `readset` are ready for reading
  - Returns the number of ready descriptors and sets each bit of `readset` to indicate the ready status of its corresponding descriptor

# I/O multiplexing: select() function

---

- readset
  - bit vector (max `FD_SETSIZE` bits) that indicates membership in a **descriptor set**
  - if bit k is 1, then descriptor k is a member of the descriptor set
- maxfd
  - Cardinality of the readset
  - tests descriptors 0, 1, 2, . . . , `maxfd-1` for set membership

# Macros for manipulating set descriptors

---

```
/* clear all bits in fdset. */
void FD_ZERO(fd_set *fdset);

/* clear bit fd in fdset */
void FD_CLR(int fd, fd_set *fdset);

/* turn on bit fd in fdset */
void FD_SET(int fd, fd_set *fdset);

/* Is bit fd in fdset on? */
int FD_ISSET(int fd, *fdset);
```

# Concurrent Programming with I/O Multiplexing

---

```
#include "csapp.h"

int main(int argc, argv)
{
    int listenfd, connfd, port;
    socklen_t clientlen=sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    fd_set read_set, ready_set;
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);
    listenfd = Open_listenfd(port);
```

# Concurrent Programming with I/O Multiplexing

```
FD_ZERO(&read_set);
FD_SET(STDIN_FILENO, &read_set);
FD_SET(listenfd, &read_set);
while(1) {
    ready_set = read_set;
    Select(listenfd+1, &ready_set,
           NULL, NULL, NULL);
    if (FD_ISSET(STDIN_FILENO, &ready_set)
        /*read command line from stdin */
        command();
    if (FD_ISSET(listenfd, &ready_set)) {
        connfd = Accept(listenfd,
                         (SA *)&clientaddr, &clientlen);
        echo(connfd);
    }
}
```

# Concurrent Programming with I/O Multiplexing

```
void command(void)
{
    char buf[MAXLINE];
    if (!Fgets(buf, MAXLINE, stdin))
        exit(0); /* EOF */
    /*Process the input command */
    printf("%s", buf);
}
```

# Pros and Cons of I/O Multiplexing

---

- + One logical control flow.
- + Can single-step with a debugger.
- + No process or thread control overhead.
  - Design of choice for high-performance Web servers and search engines.
- - Significantly more **complex** to code than process- or thread-based designs.
- - Hard to provide fine-grained concurrency
  - E.g., our example will hang up with partial lines.
- - Cannot take advantage of multi-core
  - Single thread of control

# Approaches to Concurrency

---

- Processes
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients
- Threads
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug: event orderings not repeatable
- I/O Multiplexing
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core

# Concurrent Programming

# Outline

---

- Topics:
  - Shared variables
  - Synchronizing with semaphores
- Suggested reading:
  - 12.4~12.5

# Approaches to Concurrency

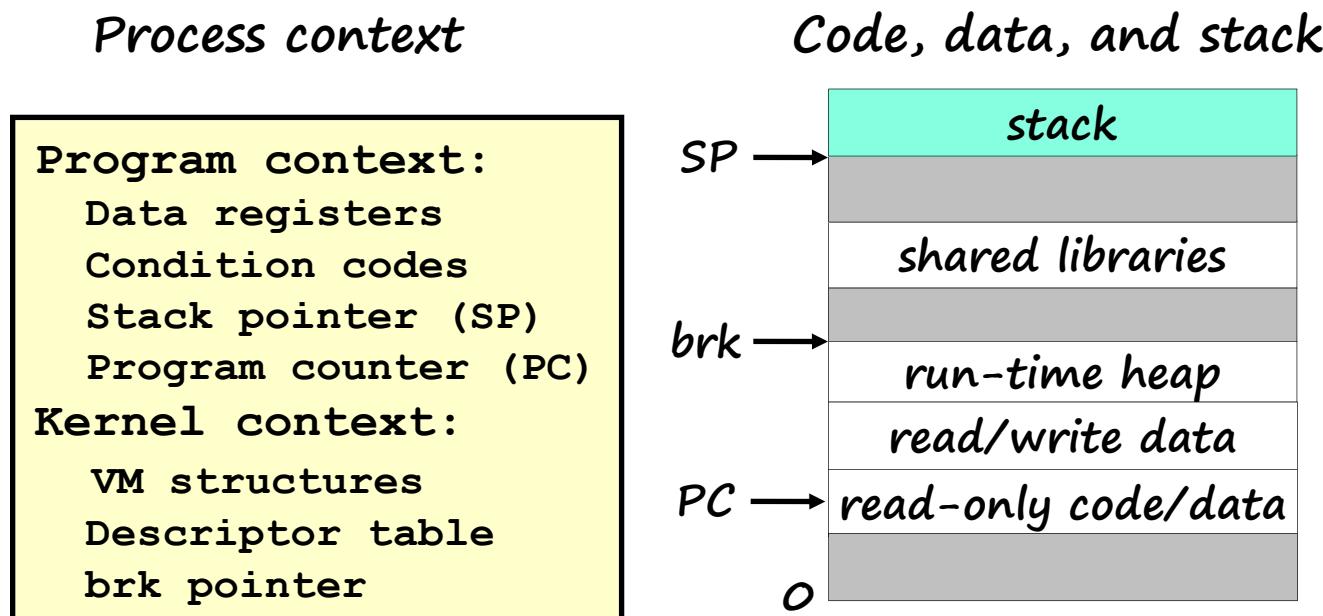
---

- Processes
  - Hard to share resources: easy to avoid unintended sharing
  - High overhead in adding/removing clients
- Threads
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug: event orderings not repeatable
- I/O Multiplexing
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core

# Traditional view of a process

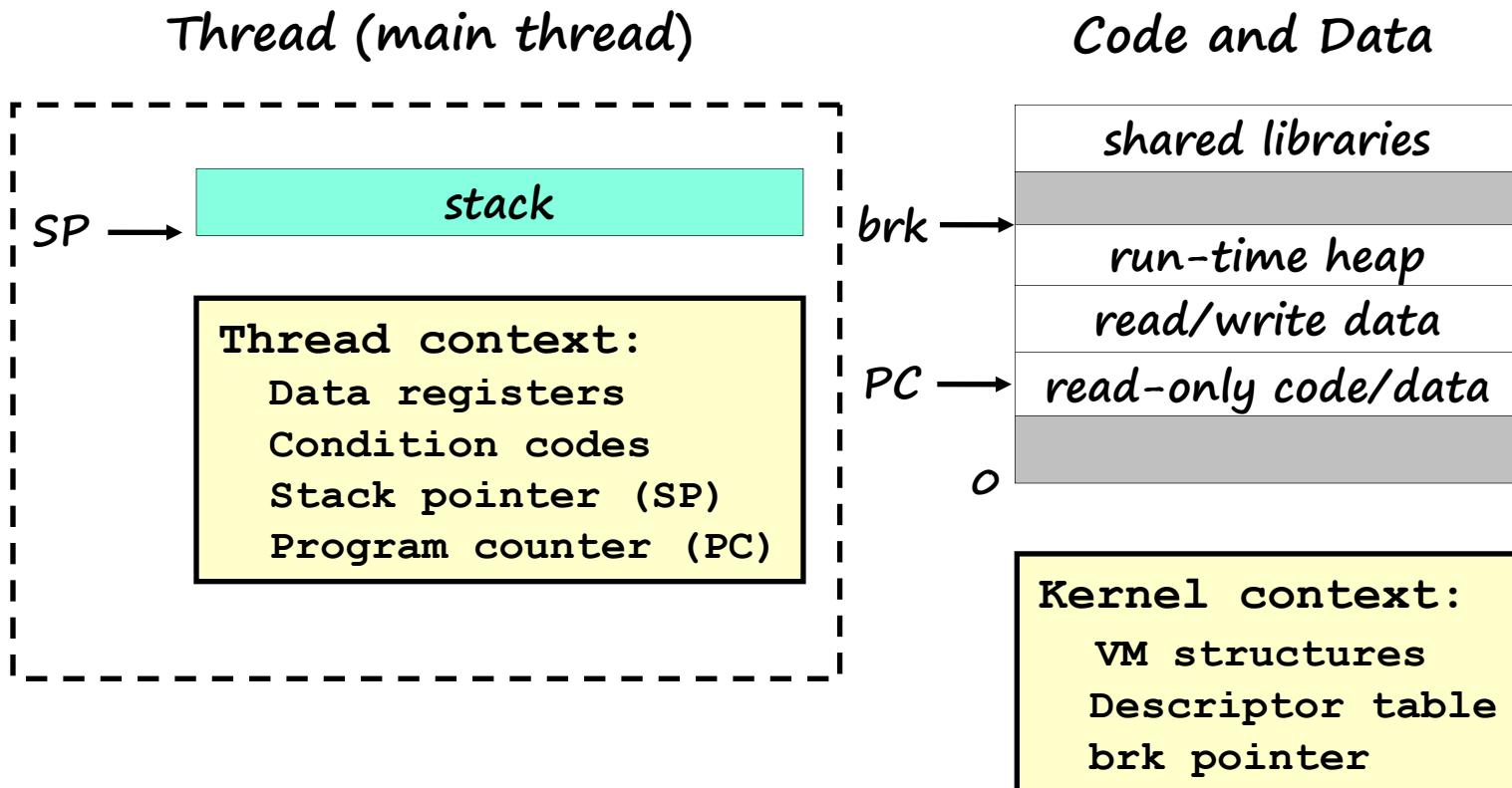
---

- Process = process context + code, data, and stack



# Alternate view of a process

- Process = thread + code, data, and kernel context



# A process with multiple threads

---

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow (sequence of PC values)
  - Each thread shares the same **code, data, and kernel context**
  - Each thread has its own **thread id (TID)**

# A process with multiple threads

---

Thread 1 (main thread)

stack 1

Thread 1 context:

Data registers

Condition codes

SP1

PC1

Shared code and data

shared libraries

run-time heap

read/write data

read-only code/data

Thread 2 (peer thread)

stack 2

Thread 2 context:

Data registers

Condition codes

SP2

PC2

Kernel context:

VM structures

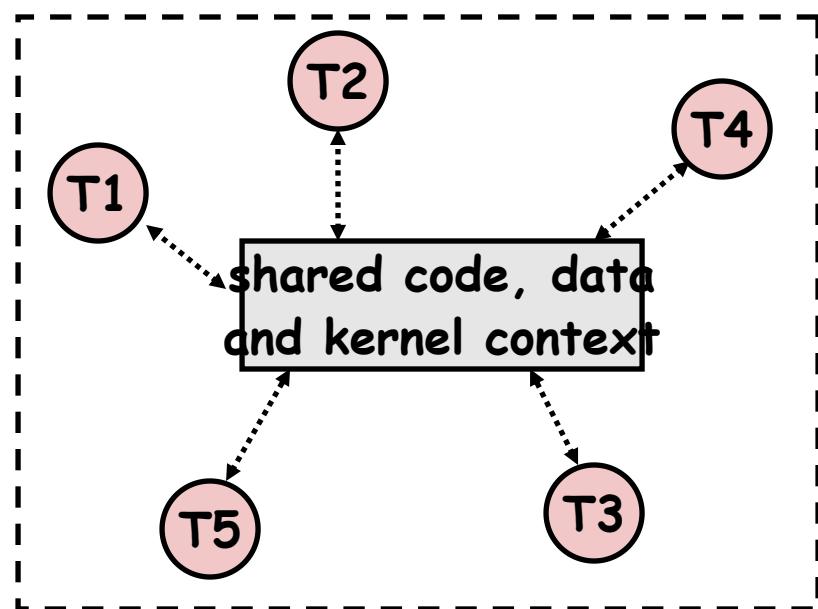
Descriptor table

brk pointer

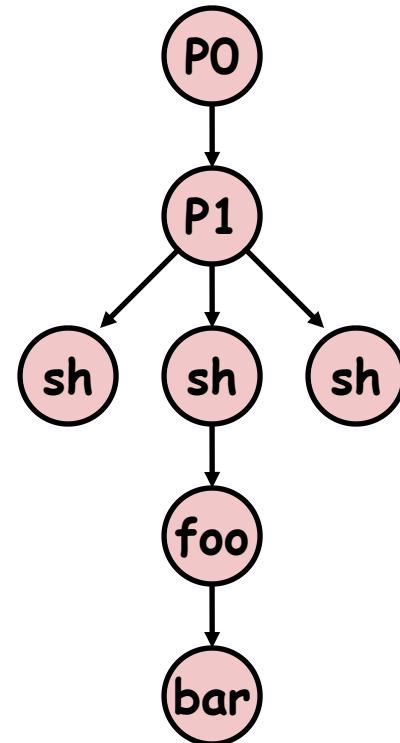
# Logical View of Threads

- Threads associated with process form a pool of peers
  - Unlike processes which form a tree hierarchy

Threads associated with process foo



Process hierarchy



# Pros and cons of thread-based designs

---

- + Easy to **share** data structures between threads
  - e.g., logging information, file cache.
- + Threads are more **efficient** than processes
- - **Unintentional sharing** can introduce subtle and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & with private

# Shared variables in threaded C programs

---

- Which variables in a C program are **shared** or **not?**
  - What is the memory model for threads?
  - How are instances of the variable mapped to memory?
  - How many threads reference each of these instances?

# Threads memory model

---

- Conceptual model:
  - Each thread runs in the context of a process.
  - Each thread has its own separate **thread context**
    - Thread ID, stack, stack pointer, program counter, condition codes, and general purpose registers
  - All threads share the remaining **process context**
    - Code, data, heap, and shared library segments of the process virtual address space
    - Open files and installed handlers

# Threads memory model

---

- Operationally, this model is not strictly enforced:
  - While register values are truly separate and protected....
  - Any thread can read and write the stack of any other thread.

*The **MISMATCH** between the **CONCEPTUAL** and **OPERATION MODEL** is a source of confusion and errors*

# Shared variable analysis

---

```
1 #include "csapp.h"
2 #define N 2
3 void *thread(void *vargp);
4
5 char **ptr; /* global variable */
6
```

# Shared variable analysis

---

```
7 int main()
8 {
9     int i;
10    pthread_t tid;
11    char *msgs[N] = {
12        "Hello from foo",
13        "Hello from bar"
14    };
15
16    ptr = msgs;
17    for (i = 0; i < N; i++)
18        Pthread_create(&tid, NULL, thread, (void *)i);
19    Pthread_exit(NULL);
20 }
```

```
1 #include "csapp.h"
2 #define N 2
3 void *thread(void *vargp);
4
5 char **ptr;
6 /* global variable */
```

# Shared variable analysis

---

```
21 void *thread(void *vargp)
22 {
23     int myid = (int)vargp;
24     static int cnt = 0;
25
26     printf("[%d] : %s (cnt=%d) \n", myid, ptr[myid], ++cnt);
27 }
```

# Mapping Variable Instances to Memory

---

- Global variables
  - Def: Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable
- Local variables
  - Def: Variable declared inside function without static attribute
  - Each thread stack contains one instance of each local variable
- Local static variables
  - Def: Variable declared inside function with the static attribute
  - Virtual memory contains exactly one instance of any local static variable.

# Shared variable analysis

---

- Which variables are shared?

Variable instance	Referenced by main thread?	Referenced by peer thread-0?	Referenced by peer thread-1?
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

# Shared variable analysis

---

- Answer: A variable  $x$  is shared iff multiple threads **reference** at least one instance of  $x$
- Thus:
  - `ptr`, `cnt`, and `msgs` are shared.
  - `i` and `myid` are NOT shared.

# Shared variable analysis

```
9 int main()
10 {
11     pthread_t tid1, tid2;
12
13     Pthread_create(&tid1, NULL, count, NULL);
14     Pthread_create(&tid2, NULL, count, NULL);
15     Pthread_join(tid1, NULL);
16     Pthread_join(tid2, NULL);
17
18     if (cnt != (unsigned)NITERS*2)
19         printf("BOOM! cnt=%d\n", cnt);
20     else
21         printf("OK cnt=%d\n", cnt);
22     exit(0);
23 }
```

```
1 #include "csapp.h"
2
3 #define NITERS 100000000
4 void *count(void *arg);
5
6 /* shared variable */
7 unsigned int cnt = 0;
8
```

# Shared variable analysis

---

```
24
25 /* thread routine */
26 void *count(void *arg)
27 {
28     int i;
29     for (i=0; i<NITERS; i++)
30         cnt++;
31     return NULL;
32 }
```

# Shared variable analysis

---

```
linux> badcnt  
BOOM! cnt=198841183
```

```
linux> badcnt  
BOOM! cnt=198261801
```

```
linux> badcnt  
BOOM! cnt=198269672
```

- `cnt` should be equal to 200,000,000.
- What went wrong?!

# Assembly code for counter loop

---

C code for thread i

```
for (i=0; i<NITERS; i++)
    cnt++;
```

# Assembly code for counter loop

## Asm code for thread i

```
Asm code for thread i

Head (Hi) { .L9:
    movl -4(%ebp), %eax #i:-4(%ebp)
    cmpl $99999999, %eax
    jle .L12
    jmp .L10
}-----.
Load cnt (Li) .L12:
Update cnt (Ui)     movl cnt, %eax      # Load
Store cnt (Si)      leal 1(%eax), %edx  # Update
                        movl %edx, cnt      # Store
}-----.
Tail (Ti) { .L11:
    movl -4(%ebp), %eax
    leal 1(%eax), %edx
    movl %edx, -4(%ebp)
    jmp .L9
}-----.
.L10:
```

# Concurrent execution

---

- Key idea: In general, any sequentially consistent interleaving is possible, but some are incorrect!
  - $I_i$  denotes that thread  $i$  executes instruction  $I$
  - $\%eax_i$  is the contents of  $\%eax$  in thread  $i$ 's context

# Concurrent execution

---

i (thread)	instr <sub>i</sub>	%eax <sub>1</sub>	%eax <sub>2</sub>	cnt
------------	--------------------	-------------------	-------------------	-----

1	H <sub>1</sub>	-	-	0
1	L <sub>1</sub>	0	-	0
1	U <sub>1</sub>	1	-	0
1	S <sub>1</sub>	1	-	1
2	H <sub>2</sub>	-	-	1
2	L <sub>2</sub>	-	1	1
2	U <sub>2</sub>	-	2	1
2	S <sub>2</sub>	-	2	2
2	T <sub>2</sub>	-	2	2
1	T <sub>1</sub>	1	-	2

OK

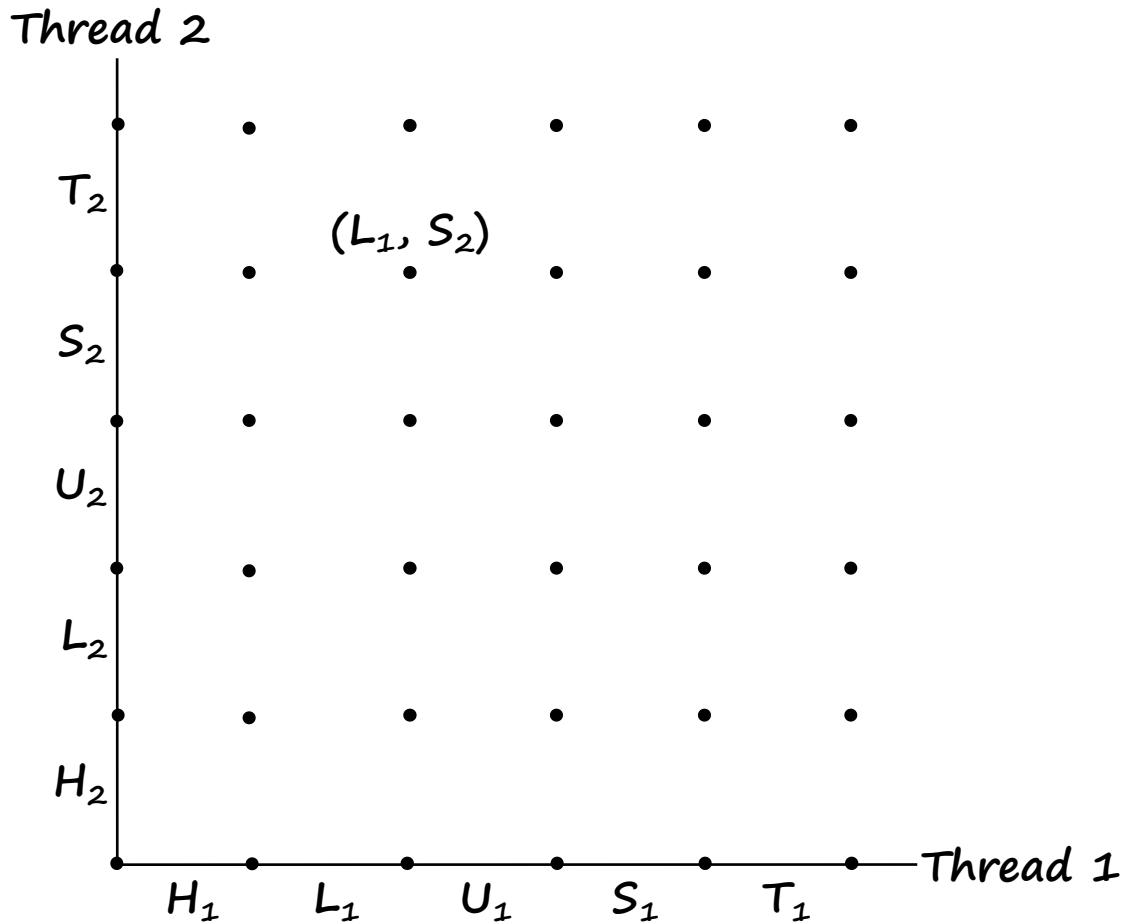
# Concurrent execution (cont)

---

- Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2.

i (thread)	instr <sub>i</sub>	%eax <sub>1</sub>	%eax <sub>2</sub>	cnt
1	H <sub>1</sub>	-	-	0
1	L <sub>1</sub>	0	-	0
1	U <sub>1</sub>	1	-	0
2	H <sub>2</sub>	-	-	0
2	L <sub>2</sub>	-	0	0
1	S <sub>1</sub>	1	-	1
1	T <sub>1</sub>	1	-	1
2	U <sub>2</sub>	-	1	1
2	S <sub>2</sub>	-	1	1
2	T <sub>2</sub>	-	1	1

# Progress graphs



A *progress graph* depicts the discrete **execution state space** of concurrent threads.

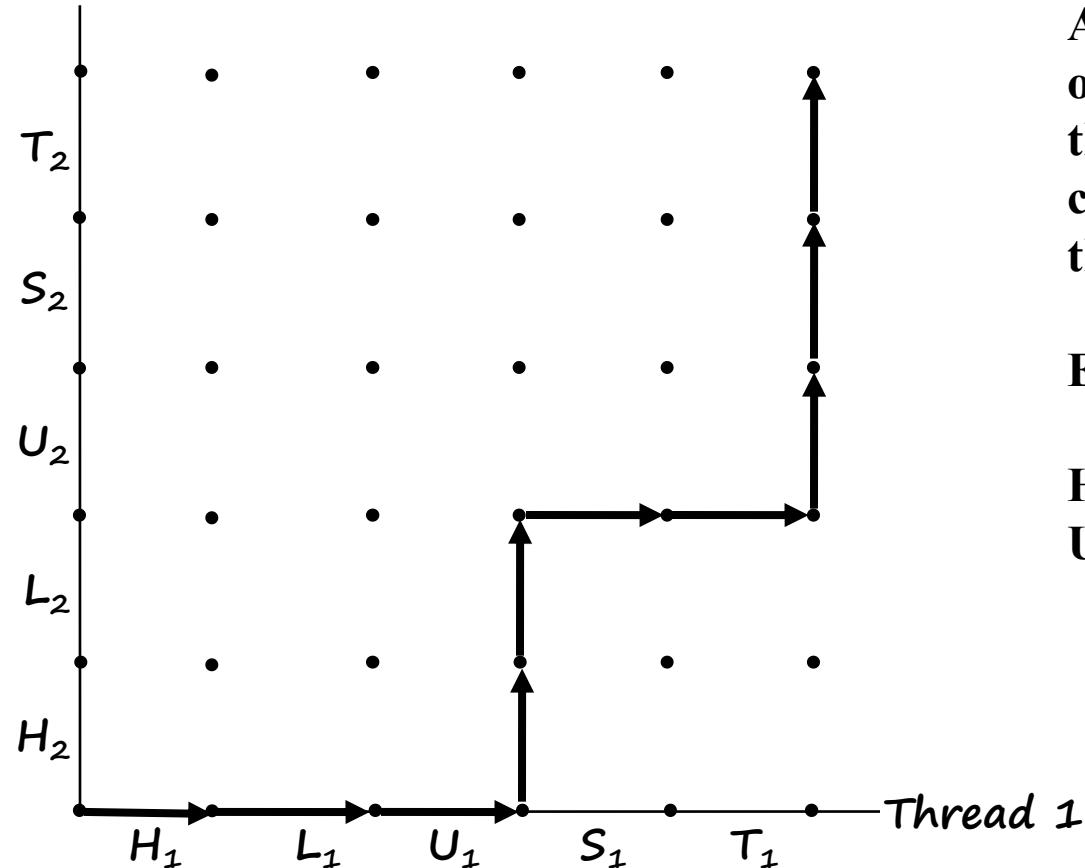
Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible **execution state** ( $Inst_1, Inst_2$ ).

E.g.,  $(L_1, S_2)$  denotes state where thread 1 has completed  $L_1$  and thread 2 has completed  $S_2$ .

# Trajectories in progress graphs

Thread 2

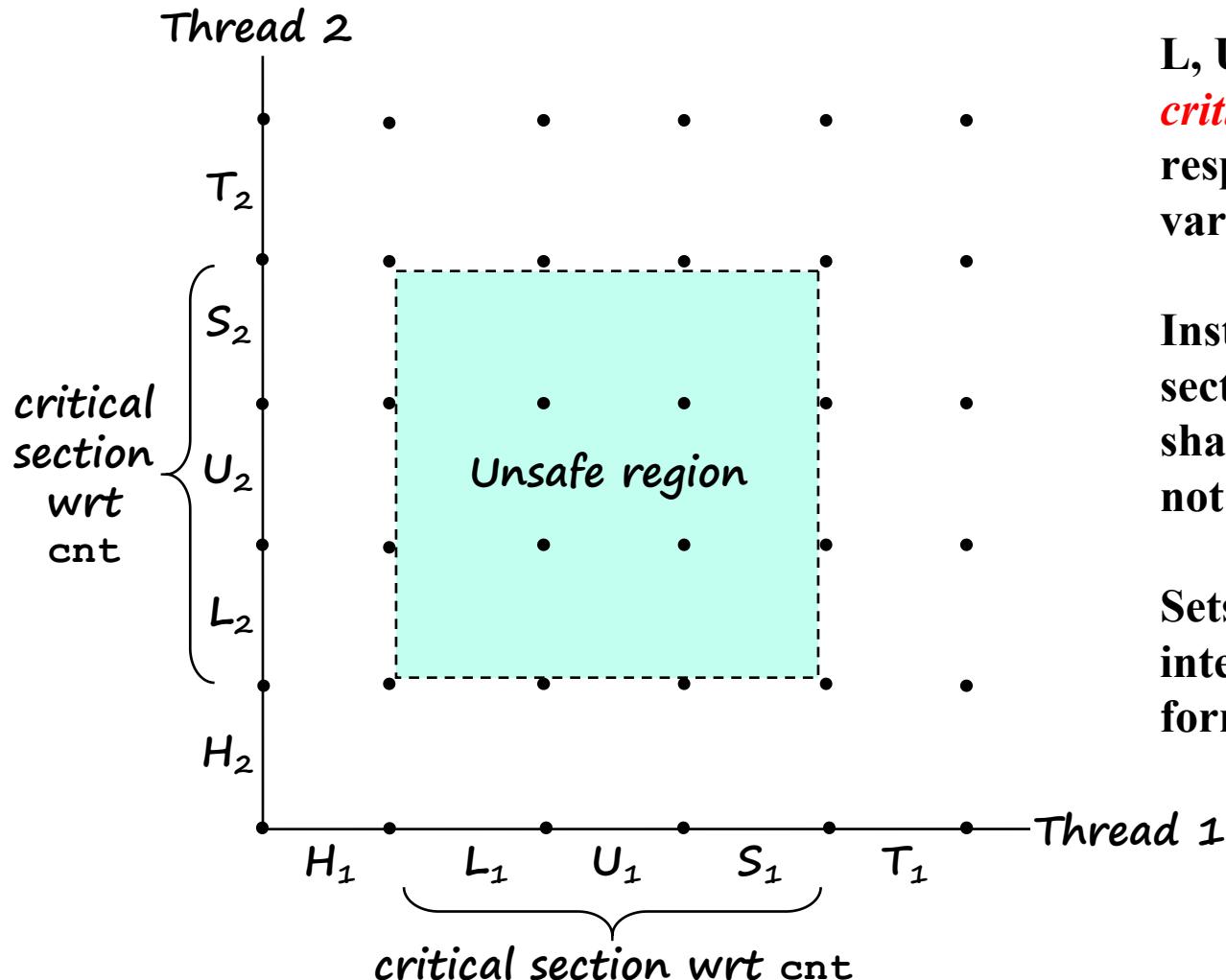


A **trajectory** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2$

# Critical sections and unsafe regions

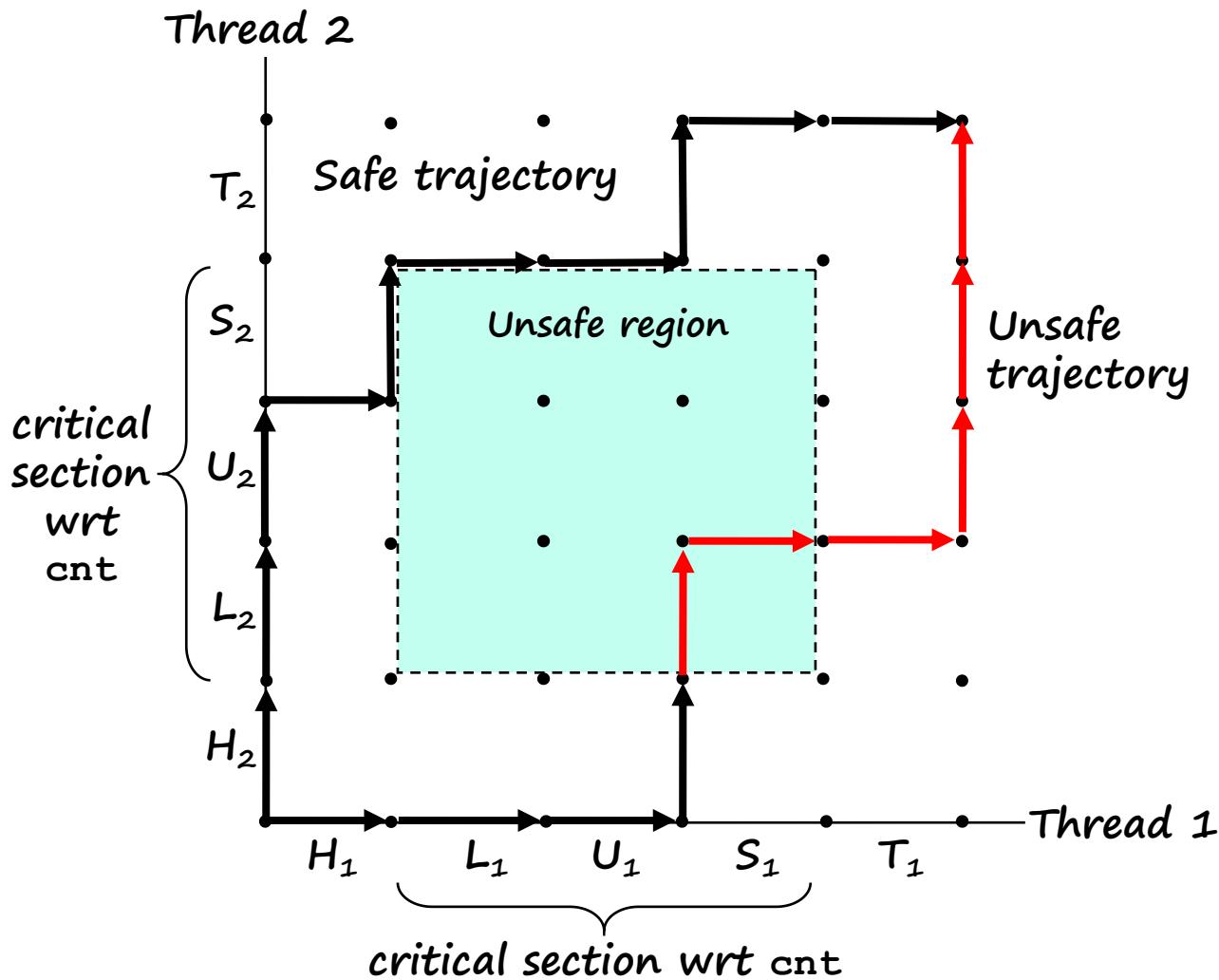


L, U, and S form a ***critical section*** with respect to the shared variable ***cnt***.

Instructions in critical sections (write to some shared variable) should not be **interleaved**.

Sets of states where such interleaving occurs form ***unsafe regions***.

# Safe and unsafe trajectories



*Def:* A trajectory is **safe** iff it doesn't touch any part of an unsafe region.

*Claim:* A trajectory is **correct** (write cnt) iff it is safe.

# Synchronizing with semaphores

---

- Dijkstra's P and V operations on **semaphores**
  - semaphore: non-negative integer synchronization variable.
    - **P(s)** : [ **while (s == 0) wait(); s--;** ]
      - Dutch for "Proberen" (test)
    - **V(s)** : [ **s++;** ]
      - Dutch for "Verhogen" (increment)

# Synchronizing with semaphores

---

- Dijkstra's P and V operations on **semaphores**
  - OS guarantees that operations between brackets [ ] are executed **indivisibly**.
    - Only one P or V operation at a time can modify s.
    - When while loop in P terminates, only that P can decrement s.
- Semaphore invariant: ( $s \geq 0$ )

# POSIX semaphores

---

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */

#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_wait */
```

# Sharing with POSIX semaphores

---

```
#include "csapp.h"
#define NITERS 10000000
unsigned int cnt; /* counter */
sem_t sem;          /* semaphore */

int main() {
    pthread_t tid1, tid2;
    Sem_init(&sem, 0, 1);
    /* create 2 threads and wait */
    ...
    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}
```

# Sharing with POSIX semaphores

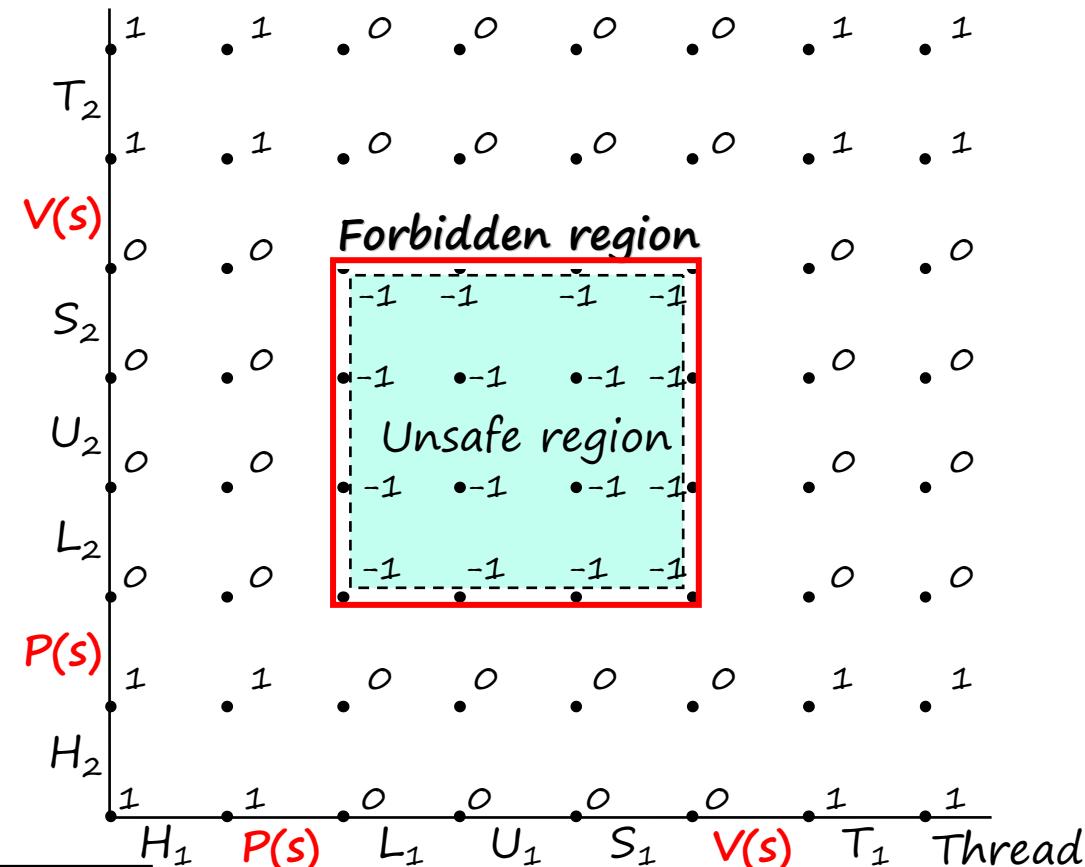
---

```
/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```

# Safe sharing with semaphores

Thread 2



Provide mutually exclusive access to shared variable by surrounding critical section with  $P$  and  $V$  operations on semaphore  $s$  (initially set to 1).

Semaphore invariant creates a forbidden region that encloses unsafe region and is never touched by any trajectory.

Initially  
 $s = 1$

# Concurrent Programming

# Outline

---

- Topics:
  - Producer-consumer problem
  - Readers-writers problem
  - Concurrent Server Based on Pthreading
  - Concurrent Issues
- Suggested reading:
  - 12.5~12.7

# Mutex

---

```
/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```

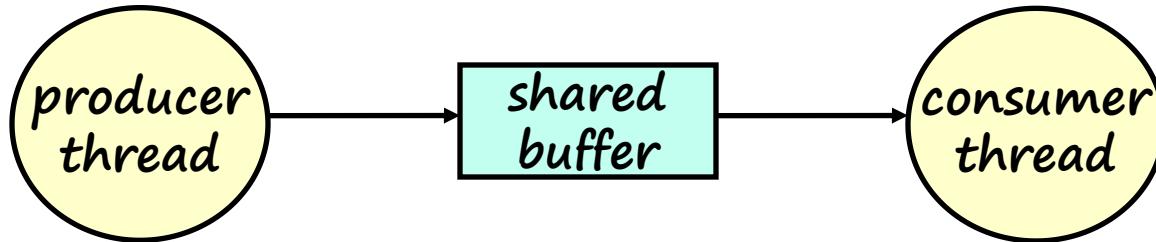
# Using Semaphores to Schedule Access to Shared Resources

---

- Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true
  - Use **counting semaphores** to keep track of resource state.
  - Use **binary semaphores** to notify other threads.
- Two classic examples:
  - The Producer-Consumer Problem
  - The Readers-Writers Problem

# Signaling with semaphores

---



- Common synchronization pattern:
  - Producer waits for empty slot, inserts item in buffer, and notifies consumer
  - Consumer waits for item, removes it from buffer, and notifies producer

# Producer-Consumer on 1-element Buffer

```
#define NITERS 5

int main() {
    pthread_t tid_producer;
    pthread_t tid_consumer;

    /* Initialize the semaphores */
    Sem_init(&shared.empty, 0, 1);
    Sem_init(&shared.full, 0, 0);
    /* Create threads and wait */
    Pthread_create(&tid_producer, NULL, producer, NULL);
    Pthread_create(&tid_consumer, NULL, consumer, NULL);
    Pthread_join(tid_producer, NULL);
    Pthread_join(tid_consumer, NULL);

    exit(0);
}

struct {
    int buf;
    sem_t full;
    sem_t empty;
} shared;
```

# Producer-Consumer on 1-element Buffer

Initially: `empty==1`, `full==0`

## Producer Thread

```
void *producer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Produce item */
        item = i;
        printf("produced %d\n",
item);

        /* Write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }

    return NULL;
}
```

## Consumer Thread

```
void *consumer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);

        /* Consume item */
        printf("consumed %d\n", item);
    }

    return NULL;
}
```

# Producer-Consumer on an $n$ -element Buffer

---

- Requires a mutex and two counting semaphores
  - **mutex**: enforces mutually exclusive access to the buffer
  - **slots**: counts the available slots in the buffer
  - **items**: counts the available items in the buffer
- Implemented using a shared buffer package called **sbuf**.

# sbuf Package - Declarations

---

```
struct {
    int *buf;          /* Buffer array */
    int n;            /* Maximum number of slots */
    int front;        /* buf[(front+1)%n] is first item */
    int rear;         /* buf[rear%n] is last item */
    sem_t mutex;      /* protects accesses to buf */
    sem_t slots;      /* Counts available slots */
    sem_t items;      /* Counts available items */
} sbuf_;
```

# sbuf Package - Implementation

---

```
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    /* Buffer holds max of n items */
    sp->n = n;
    /* Empty buffer iff front == rear */
    sp->front = sp->rear = 0;
    /* Binary semaphore for locking */
    Sem_init(&sp->mutex, 0, 1);
    /* Initially, buf has n empty slots */
    Sem_init(&sp->slots, 0, n);
    /* Initially, buf has zero data items */
    Sem_init(&sp->items, 0, 0);
}
```

# sbuf Package - Implementation

---

```
void sbuf_insert(sbuf_t *sp, int item)
{
    /* Wait for available slot */
    P(&sp->slots);
    /*Lock the buffer */
    P(&sp->mutex);
    /*Insert the item */
    sp->buf[ (++sp->rear) % (sp->n) ] = item;
    /* Unlock the buffer */
    V(&sp->mutex);
    /* Announce available items*/
    V(&sp->items);
}
```

# sbuf Package - Implementation

---

```
void sbuf_remove(sbuf_t *sp)
{
    int item;
    /* Wait for available item */
    P(&sp->items);
    /*Lock the buffer */
    P(&sp->mutex);
    /*Remove the item */
    item = sp->buf[ (++sp->front) % (sp->n) ];
    /* Unlock the buffer */
    V(&sp->mutex);
    /* Announce available slot*/
    V(&sp->slots);
    return item;
}
```

# Readers-Writers Problem

---

- Generalization of the mutual exclusion problem
- Problem statement:
  - Reader threads only read the object
  - Writer threads modify the object
  - Writers must have **exclusive** access to the object
  - Unlimited number of readers can access the object

# Readers-Writers Problem

---

- Occurs frequently in real systems, e.g.,
  - Online airline reservation system
  - Multithreaded caching Web proxy

# Variants of Readers-Writers

---

- First readers-writers problem  
(favors readers)
  - No reader should be kept waiting unless a writer has already been granted permission to use the object.
  - A reader that arrives after a waiting writer gets priority over the writer.

# Variants of Readers-Writers

---

- Second readers-writers problem (favors writers)
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting.
- Starvation (where a thread waits indefinitely) is possible in both cases.

# Solution to First Readers-Writers Problem

---

## Readers:

```
int readcnt;      /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

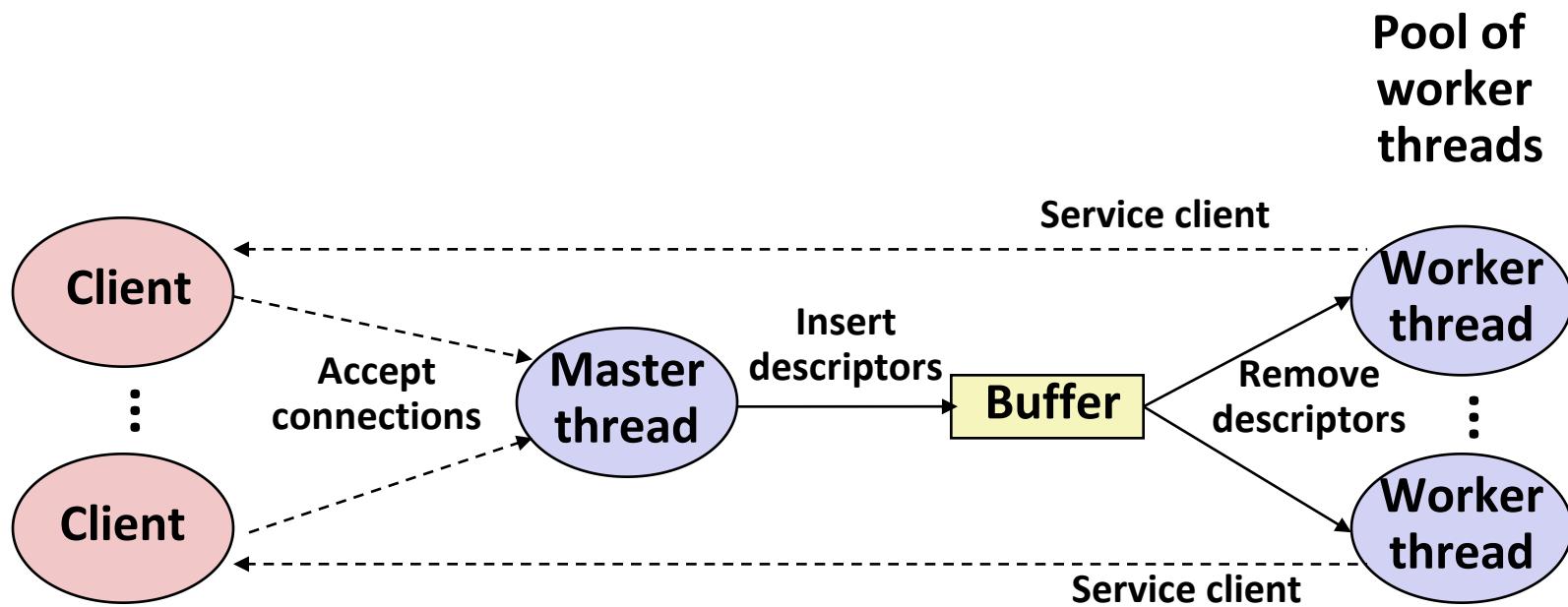
void reader(void) {
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

## Writers:

```
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

# Case Study: Prethreaded Concurrent Server

---



# Prethreading

---

```
#define NTHREADS    4
#define SBUFSIZE   16

/* shared buffer of connected descriptors */
sbuf_t sbuf ;

int main(int argc, char **argv)
{
    int i, listenfd, connfd, port;
    int clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    pthread_t tid;
```

# Prethreading

---

```
port = atoi(argv[0]);
sbuf_init(&sbuf, SBUFSIZE);
listenfd = open_listenfd(port);
/* Create worker threads */
for (i = 0; i < NTHREADS; i++)
    Pthread_create(&tid, NULL, thread, NULL);

while (1) {
    connfd = Accept (listenfd,
                    (SA *)&clientaddr, &clientlen);
    /* Insert connfd in buffer */
    sbuf_insert(&sbuf, connfd);
}
}
```

# Prethreading

---

```
void *thread(void *vargp)
{
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf);
        /* Remove connfd from buffer */
        echo_cnt(connfd);
        /* Service client */
        Close(connfd);
    }
}
```

# Prethreading

---

```
#include "csapp.h"

static int byte_cnt;      /* byte counter */
static sem_t mutex;       /* and the mutex that
protects it */

static void init_echo_cnt(void)
{
    Sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}
```

# Prethreading

---

```
void echo_cnt(int connfd)
{
    int n; char buf[MAXLINE]; rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;

    Pthread_once(&once, init_echo_cnt);
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        P(&mutex);
        byte_cnt += n;
        printf("thread %d rec. %d(%d) byte on fd %d\n",
               (int)pthread_self(), n, byte_cnt, connfd);
        V(&mutex);
        Rio_writen(connfd, buf, n);
    }
}
```

# Thread-safe functions

---

- Functions called from a thread must be **thread-safe**
- Definition: A function is thread-safe iff it will always produce correct results when called **repeatedly** from multiple concurrent threads.

# Thread-safe functions

---

- We identify four (non-disjoint) classes of thread-unsafe functions:
  - Class 1: Failing to protect shared variables.
  - Class 2: Relying on persistent state across invocations.
  - Class 3: Returning a pointer to a static variable.
  - Class 4: Calling thread-unsafe functions.

# Thread-unsafe functions

---

- Class 1: Failing to protect shared variables.
  - Fix: use Pthreads P/V operations.
  - Do not need to modify the calling code
  - Issue: synchronization operations will slow down code.

# Thread-unsafe functions (case 2)

---

- Class 2: Relying on persistent state across invocations

```
unsigned int next = 1;
/* rand - return pseudo-random int on 0..32767 */
int rand(void)
{
    next = next * 110351524 + 12345 ;
    return (unsigned int) ((next/65536) % 32768);
}
/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

# Thread-unsafe functions (case 2)

---

- Pass state as part of argument
  - and, thereby, eliminate static state

```
/* rand_r - return pseudo-random int on 0..32767 */
int rand_r(int *nextp)
{
    *nextp = *nextp * 110351524 + 12345 ;
    return (unsigned int) ((*nextp/65536) % 32768);
}
```

# Thread-safe functions

---

- Class 3: Returning a pointer to a static variable

```
struct hostent *gethostbyname(char name)
{
    static struct hostent host;
    <contact DNS and fill in h>
    return &host;
}
```

# Thread-safe functions

- Class 3: (Fixes)

```
hostp = Malloc(...);  
gethostbyname_r(name, hostp);
```

- Rewrite so caller passes pointer to struct
  - Issue: Requires changes in caller and callee
- “Lock-and-copy”
  - Issues: caller must free memory

```
struct hostent *gethostbyname_ts(char *name)  
{  
    struct hostent *p,*q = Malloc(...);  
    P(&mutex);  
    p = gethostbyname(name);  
    *q = *p;  
    V(&mutex);  
    return q;  
}
```

# Thread-safe functions

---

- Class 4: Calling thread-unsafe functions.
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the function so it calls only thread-safe functions ☺

# Reentrant functions

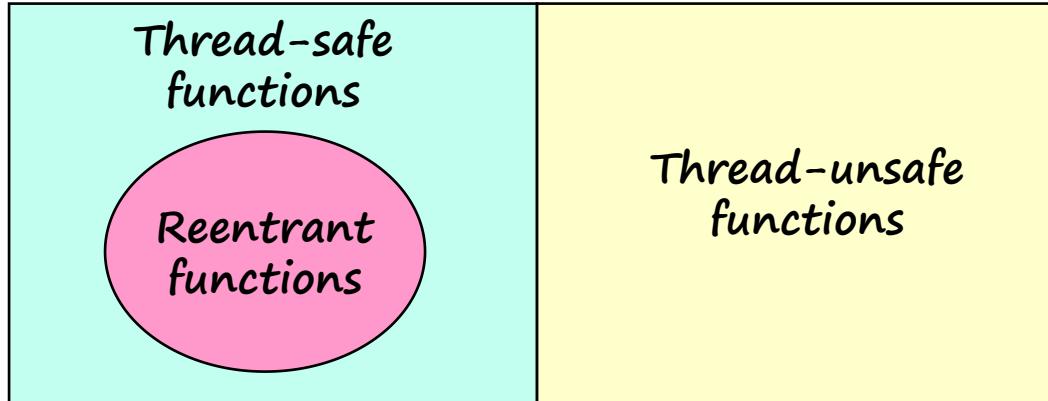
---

- A function is **reentrant** iff it accesses NO shared variables when called from multiple threads.
  - Reentrant functions are an important subset of thread-safe functions.
  - Require no synchronization operations.

# Reentrant functions

---

## All functions



NOTE: The fixes to Class 2 thread-unsafe functions require modifying the function to make it reentrant

# Thread-safe library functions

---

- All functions in the Standard C Library (at the back of your K&R text) are thread-safe.
- Most Unix system calls are thread-safe, with a few exceptions:

Thread-unsafe function	Class	Reentrant version
<code>rand</code>	2	<code>rand_r</code>
<code>Strtok</code>	2	<code>strtok_r</code>
<code>asctime</code>	3	<code>asctime_r</code>
<code>ctime</code>	3	<code>ctime_r</code>
<code>gethostbyaddr</code>	3	<code>gethostbyaddr_r</code>
<code>gethostbyname</code>	3	<code>gethostbyname_r</code>
<code>inet_ntoa</code>	3	(none)
<code>localtime</code>	3	<code>localtime_r</code>

## Issues: Races

---

- Correctness of a program depends on
  - one thread reaching point x in its control before
  - another thread reaches point y

# Issues: Races

```
#define N 4
int main()
{
    pthread_t tid[N];
    int i ;
    for ( i=0 ; i<N ; i++ )
        pthread_create(&tid[i] , NULL, thread, &i);
    for ( i=0 ; i<N ; i++ )
        pthread_join(tid[i] , NULL) ;
    exit(0) ;
}

/*thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp) ;
    printf("Hello from th. %d\n", myid);
    return NULL ;
}
```

# Issues: Races

---

```
int main()
{
    pthread_t tid[N];
    int i, *ptr ;
    for ( i=0 ; i<N ; i++ ) {
        ptr = malloc(sizeof(int));
        *ptr = i ;
        pthread_create(&tid[i], NULL, thread, ptr);
    }
    for ( i=0 ; i<N ; i++ )
        pthread_join(tid[i], NULL) ;
    exit(0) ;
}
```

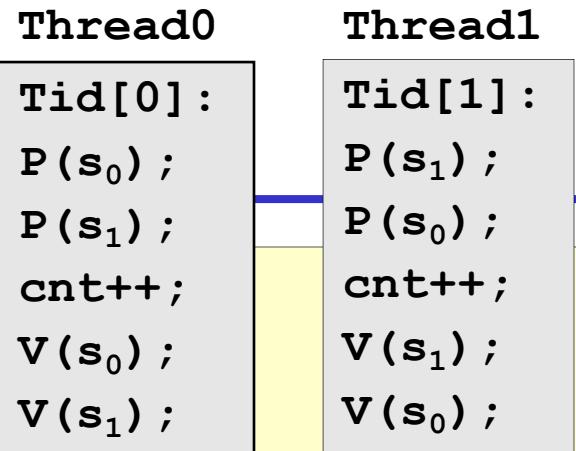
# Issues: Deadlock

---

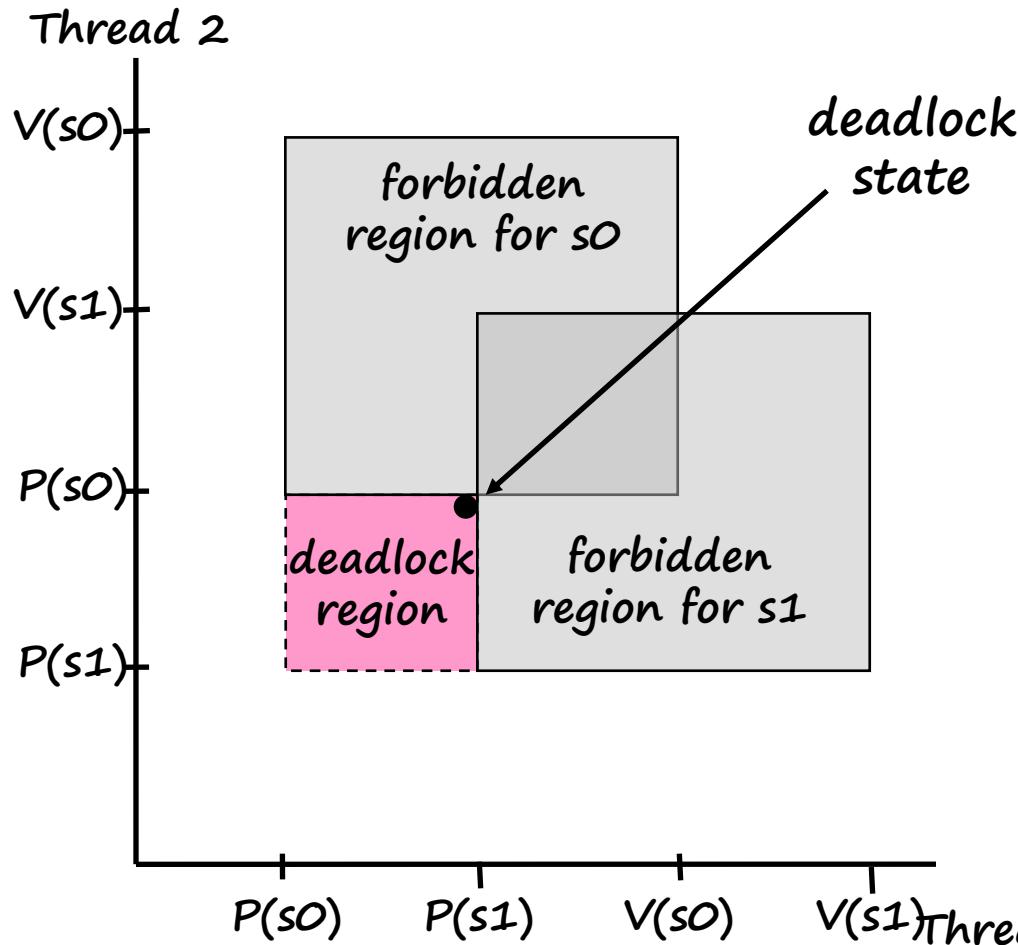
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*)0);
    Pthread_create(&tid[1], NULL, count, (void*)1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

# Issues: Deadlock

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```



# Issues: Deadlock



Initially,  $s=t=1$

Locking introduces the potential for **deadlock**: waiting for a condition that will never be true.

Any trajectory that enters the **deadlock region** will eventually reach the **deadlock state**, waiting for either  $s$  or  $t$  to become nonzero.

Other trajectories luck out and skirt the deadlock region.

# Threads Summary

---

- Threads provide another mechanism for writing concurrent programs
- Threads are growing in popularity
  - Somewhat cheaper than processes
  - Easy to share data between threads
- However, the ease of sharing has a cost:
  - Easy to introduce subtle synchronization errors
  - Tread carefully with threads!
- For more info:
  - D. Butenhof, "Programming with Posix Threads", Addison-Wesley, 1997

# Virtual Memory (I)

# Outline

---

- Physical and Virtual Addressing
- Address Spaces
- VM as a Tool for Caching
- VM as a Tool for Memory Management
- VM as a Tool for Memory Protection
- Suggested reading: 9.1~9.5

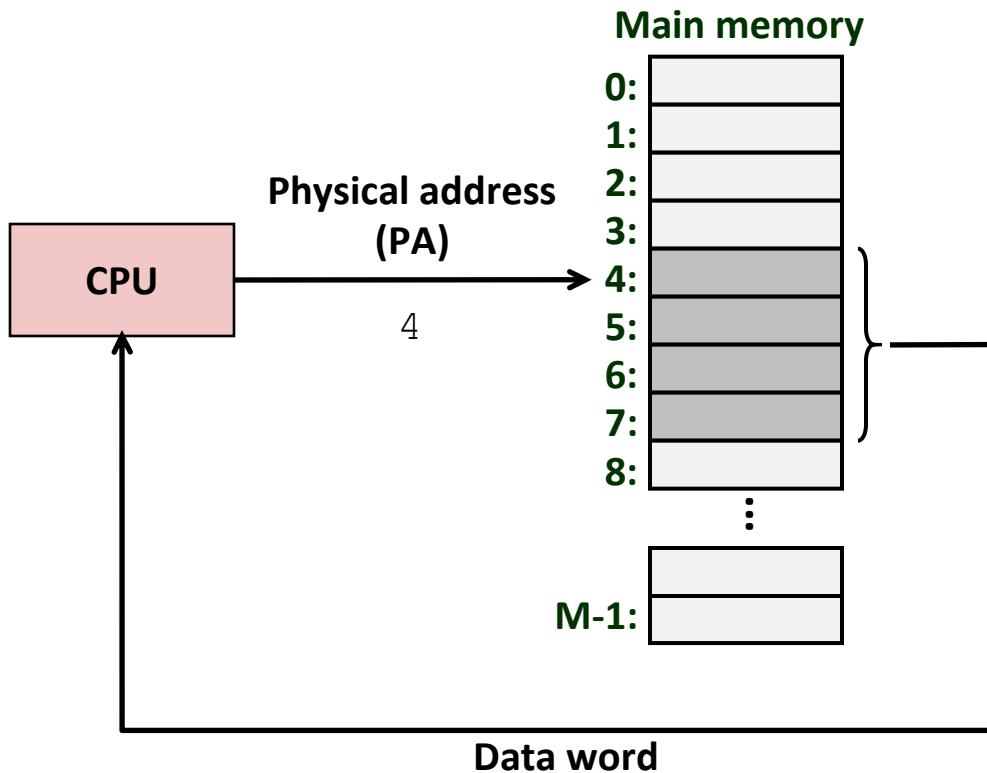
# Physical Addressing

---

- Attributes of the main memory
  - Organized as an array of  $M$  contiguous byte-sized cells
  - Each byte has a unique **physical address** (PA) started from 0
- physical addressing
  - A CPU use physical addresses to access memory
- Examples
  - Early PCs, DSP, embedded microcontrollers, and Cray supercomputers

# A System Using Physical Addressing

---



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

# Virtual Addressing

---

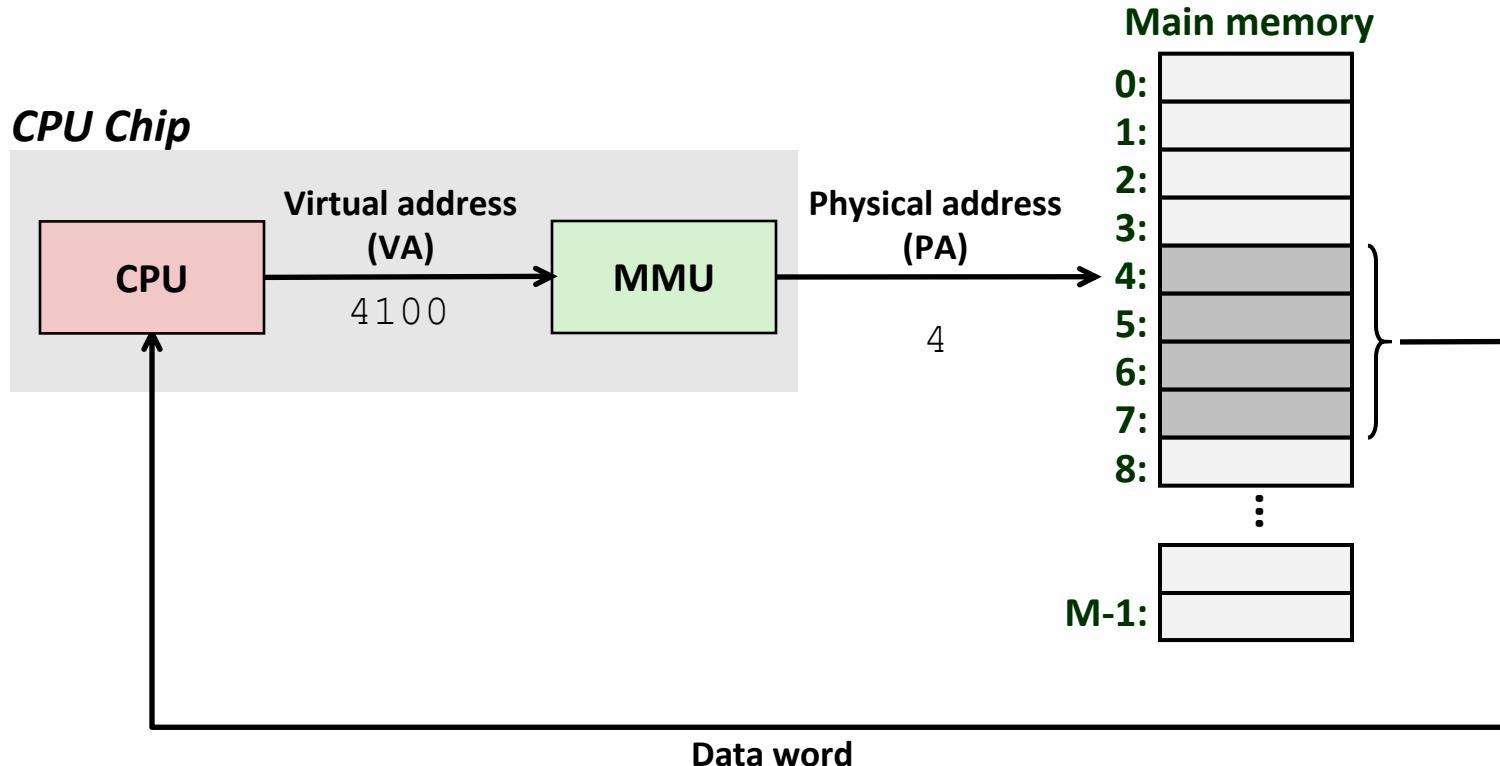
- Virtual addressing
  - the CPU accesses main memory by a **virtual address** (VA)
    - The virtual address is converted to the appropriate physical address

# Virtual Addressing

---

- Address translation
  - Converting a **virtual address** to a **physical address**
  - Requires close **cooperation** between the CPU hardware and the operating system
    - HW: the memory management unit (MMU)
      - Dedicated hardware on the CPU chip to translate virtual addresses on the fly
    - SW: A look-up table
      - Stored in main memory
      - Contents are managed by the operating system

# A System Using Virtual Addressing



- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science

# Address Space

---

- Address Space
  - An ordered set of nonnegative integer addresses
- Linear Space
  - The integers in the address space are consecutive
- N-bit address space

# Address Spaces

---

**Linear address space:** Ordered set of contiguous non-negative integer addresses:

$$\{0, 1, 2, 3 \dots\}$$

**Virtual address space:** Set of  $N = 2^n$  virtual addresses

$$\{0, 1, 2, 3, \dots, N-1\}$$

**Physical address space:** Set of  $M = 2^m$  physical addresses

$$\{0, 1, 2, 3, \dots, M-1\}$$

- Clean distinction between data (**bytes**) and their attributes (**addresses**)
- Each object can now have multiple addresses
- Every byte in main memory:  
one physical address, one (or more) virtual addresses

# Address Space

---

- $K=2^{10}$ (Kilo),  $M=2^{20}$ (Mega),  $G=2^{30}$ (Giga),  
 $T=2^{40}$ (Tera),  $P=2^{50}$ (Peta),  $E=2^{60}$ (Exa)

#virtual address bits (n)	#virtual address (N)	Largest possible virtual address
8	256	255
16	64K	64K-1
32	4G	4G-1
48	256T	256T-1
64	16E	16E-1

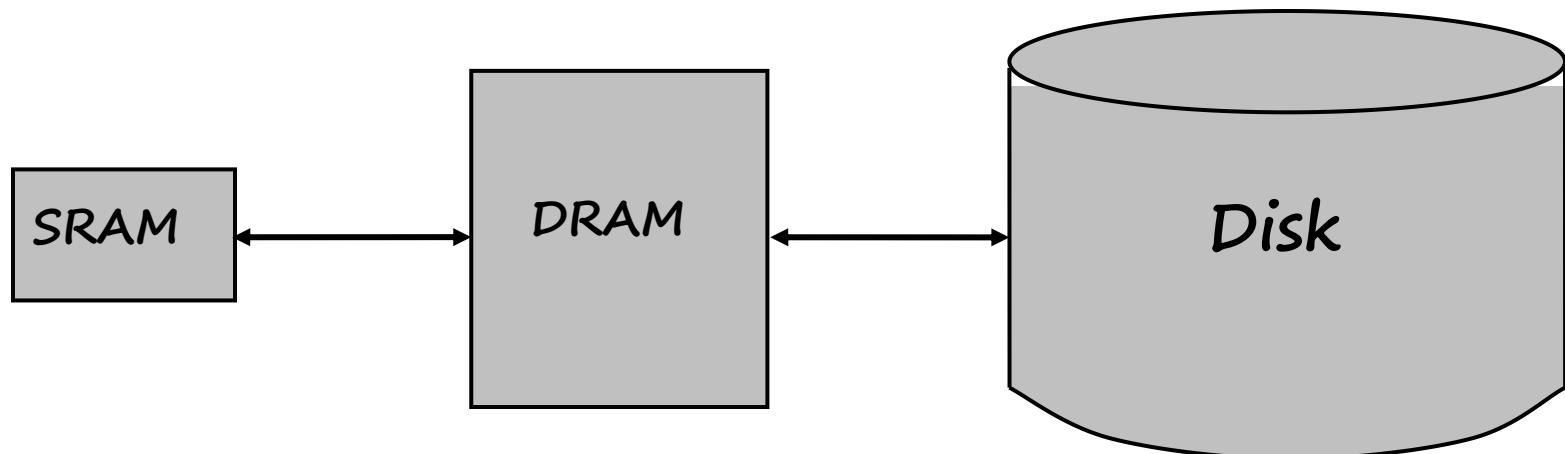
# Why Virtual Memory (VM)?

---

- Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
- Isolates address spaces
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information

# Using Main Memory as a Cache

---



# Using Main Memory as a Cache

---

- DRAM vs. disk is more extreme than SRAM vs. DRAM
  - Access latencies:
    - DRAM ~**10X** slower than SRAM
    - Disk ~**100,000X** slower than DRAM
  - Bottom line:
    - Design decisions made for DRAM caches driven by enormous cost of misses

# Design Considerations

---

- Line size? (Large vs. Small)
  - Large, since disk better at transferring large blocks
- Associativity? (Full vs. Direct)
  - Full, to minimize miss rate
- Write through or write back? (WB vs. WT)
  - Write back, since can't afford to perform small writes to disk

# Page

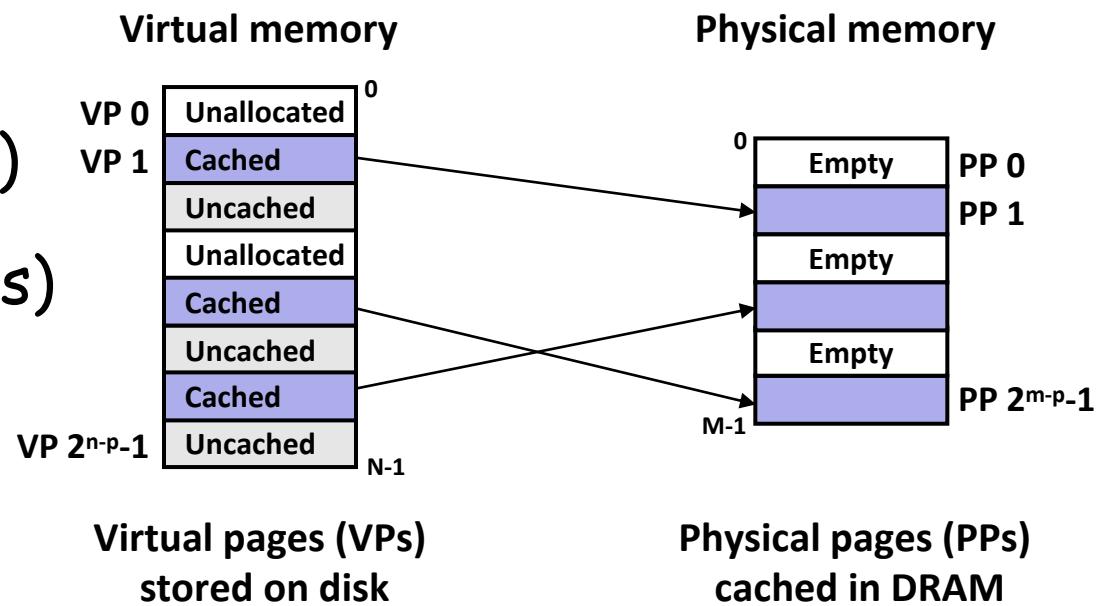
---

- Virtual memory
  - Organized as an **array** of contiguous byte-sized cells stored on disk conceptually.
  - Each byte has a unique **virtual address** that serves as an index into the array
  - The contents of the array on disk are **cached** in main memory

# Page

---

- The data on disk is partitioned into **blocks**
  - Serve as the transfer units between the disk and the main memory
  - virtual pages (VPs)
  - physical pages (PPs)
    - or page frames



# Page Attributes

---

- Unallocated:
  - Pages that have not yet been allocated (or created) by the VM system
  - Do not have any data associated with them
  - Do not occupy any space on disk.

# Page Attributes

---

- **Cached:**
  - Allocated pages that are currently cached in physical memory.
- **Uncached:**
  - Allocated pages that are not cached in physical memory.

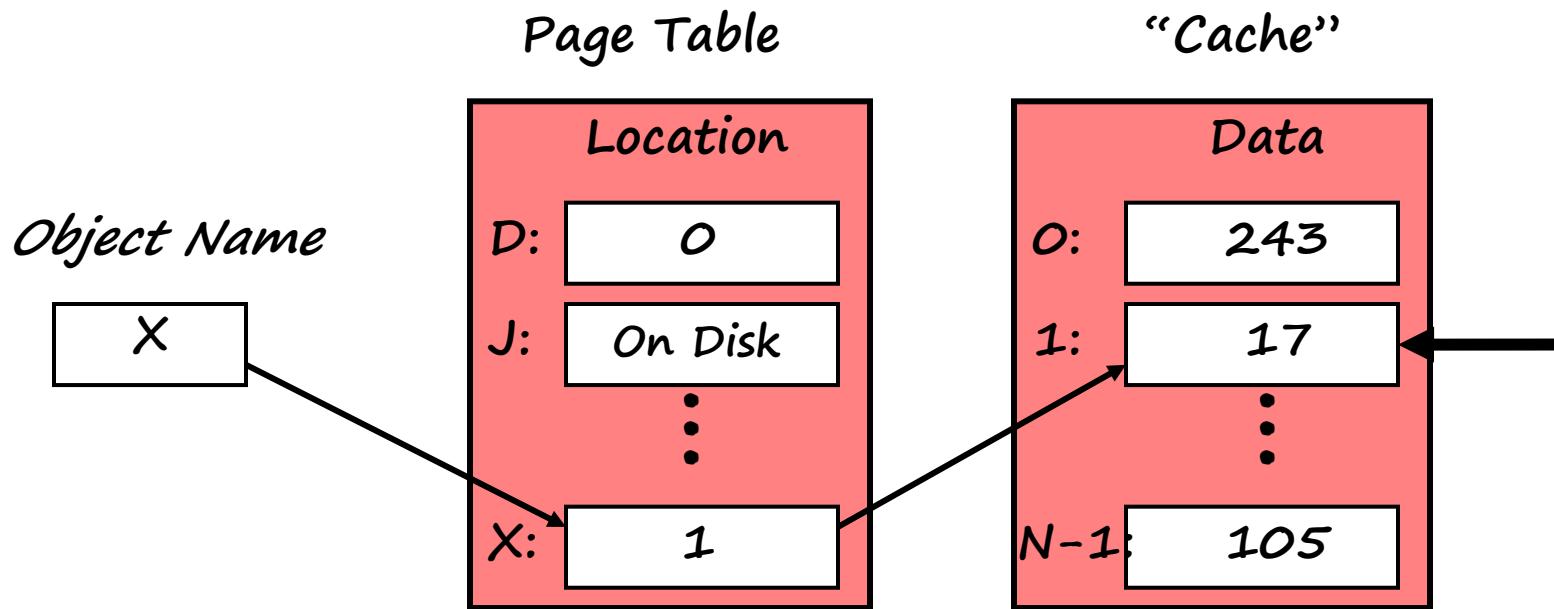
# Page Table

---

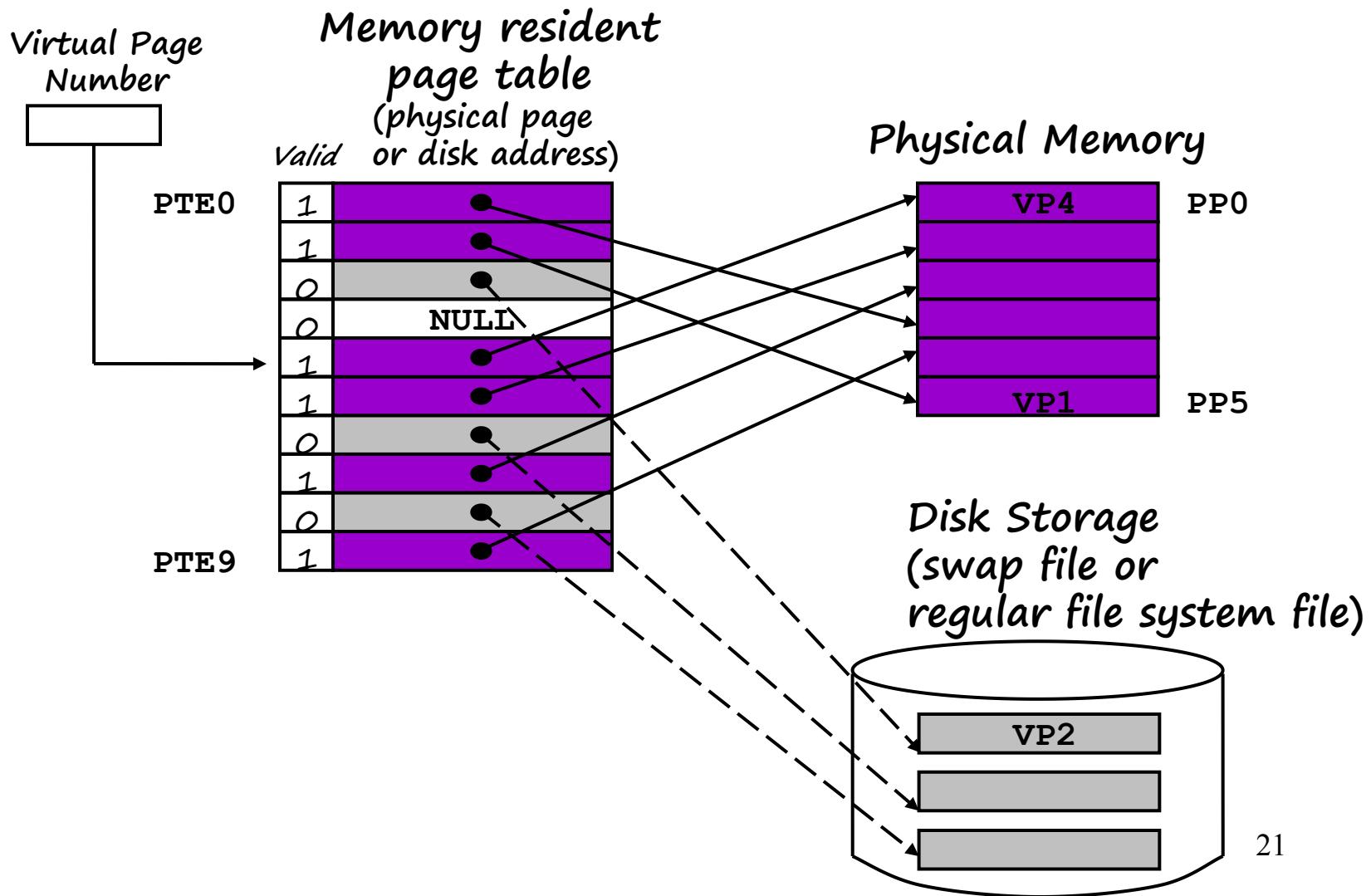
- Each allocate page of virtual memory has entry in **page table**
- **Mapping** from virtual pages to physical pages
  - From uncached form to cached form
- Page table entry even if page not in memory
  - Specifies disk address
- OS retrieves information

# Page Table

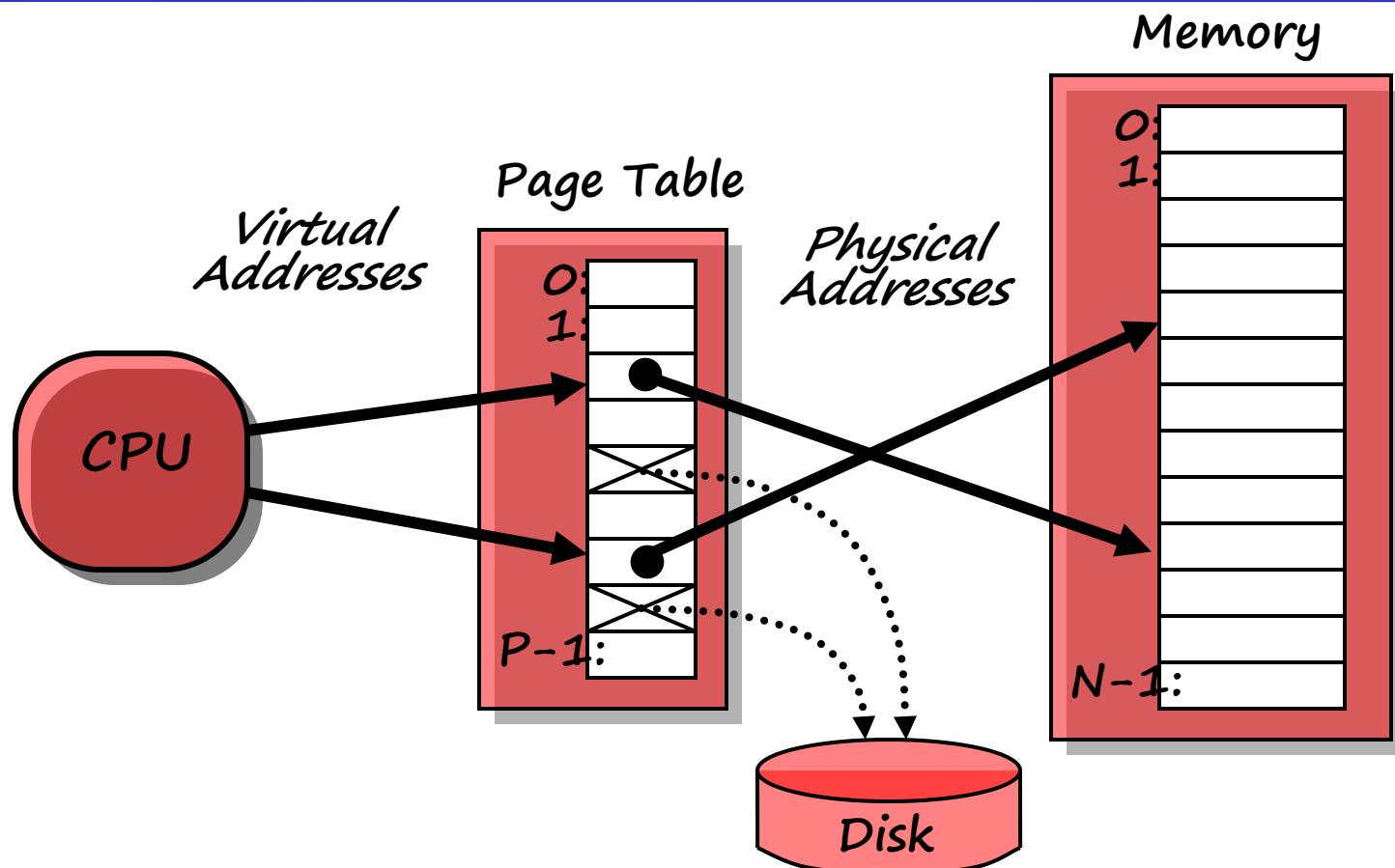
---



# Page Table



# Page Hits



Address Translation: Hardware converts *virtual addresses* to *physical addresses* via an OS-managed lookup table (*page table*)

# Page Faults

---

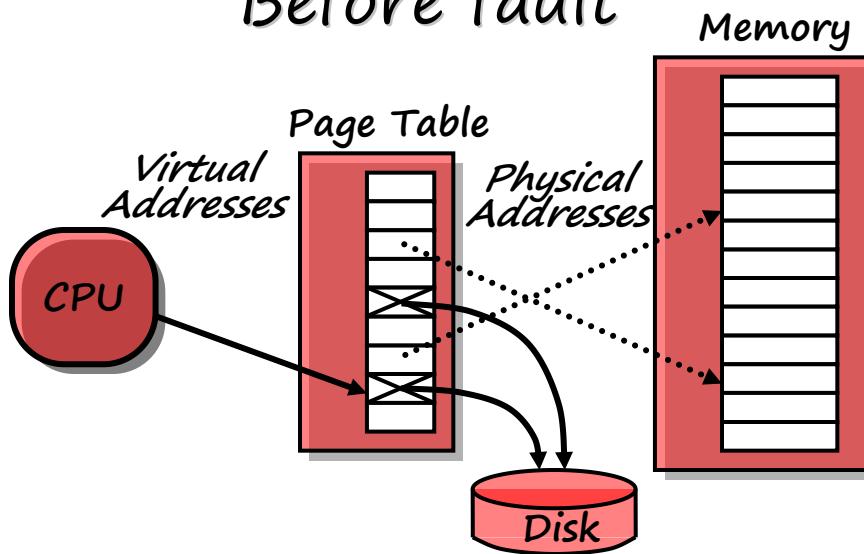
- Page table entry indicates virtual address **not in memory**
- OS exception handler invoked to move data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.

# Page Faults

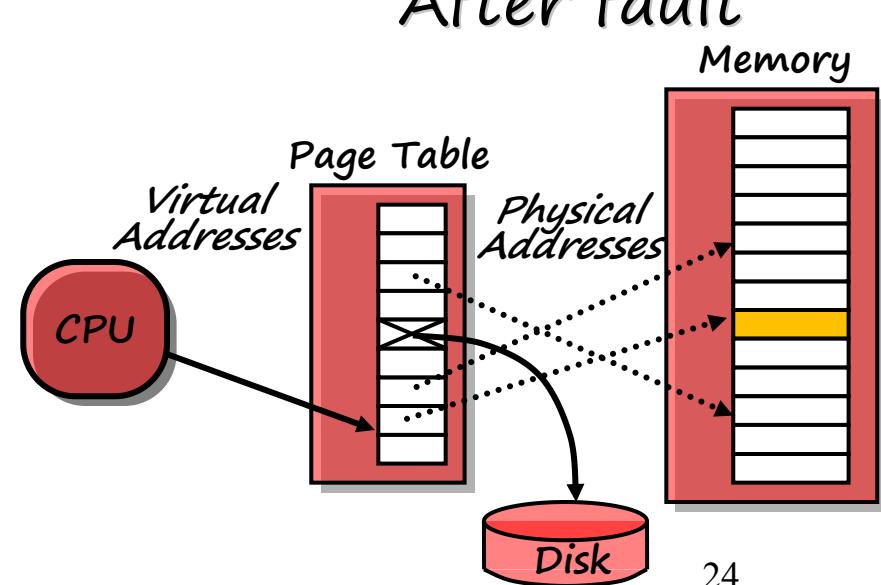
---

- Swapping or paging
- Swapped out or paged out
- Demand paging

Before fault

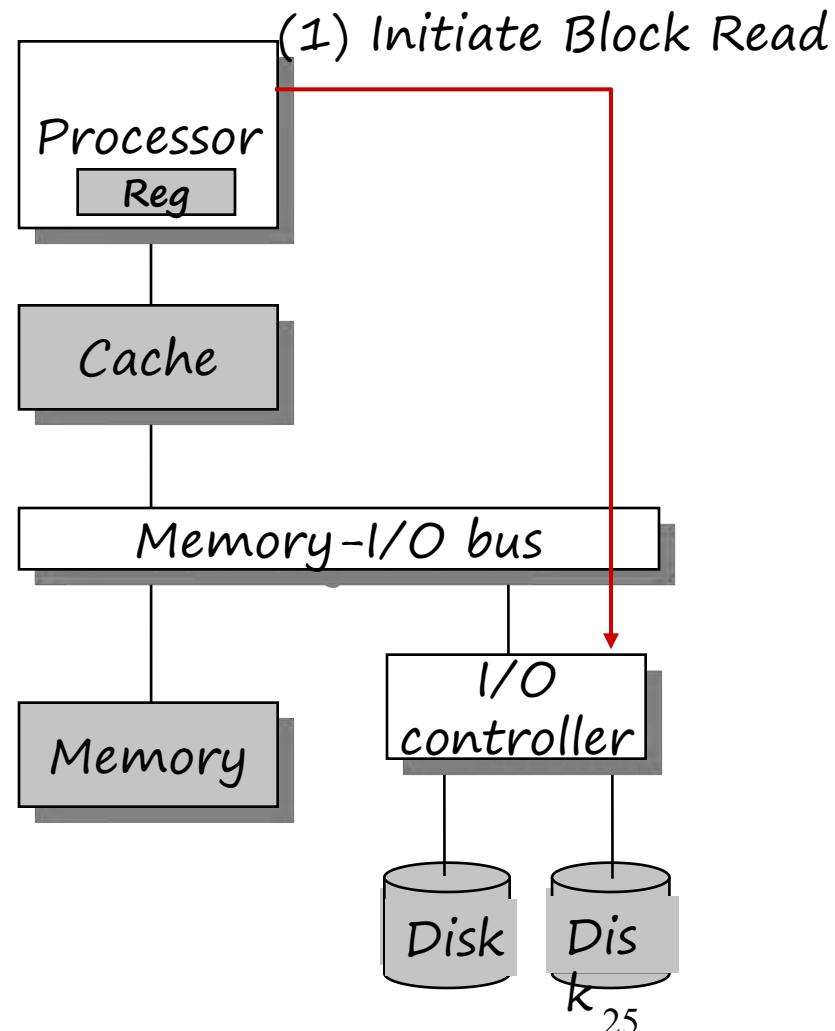


After fault



# Servicing a Page Fault

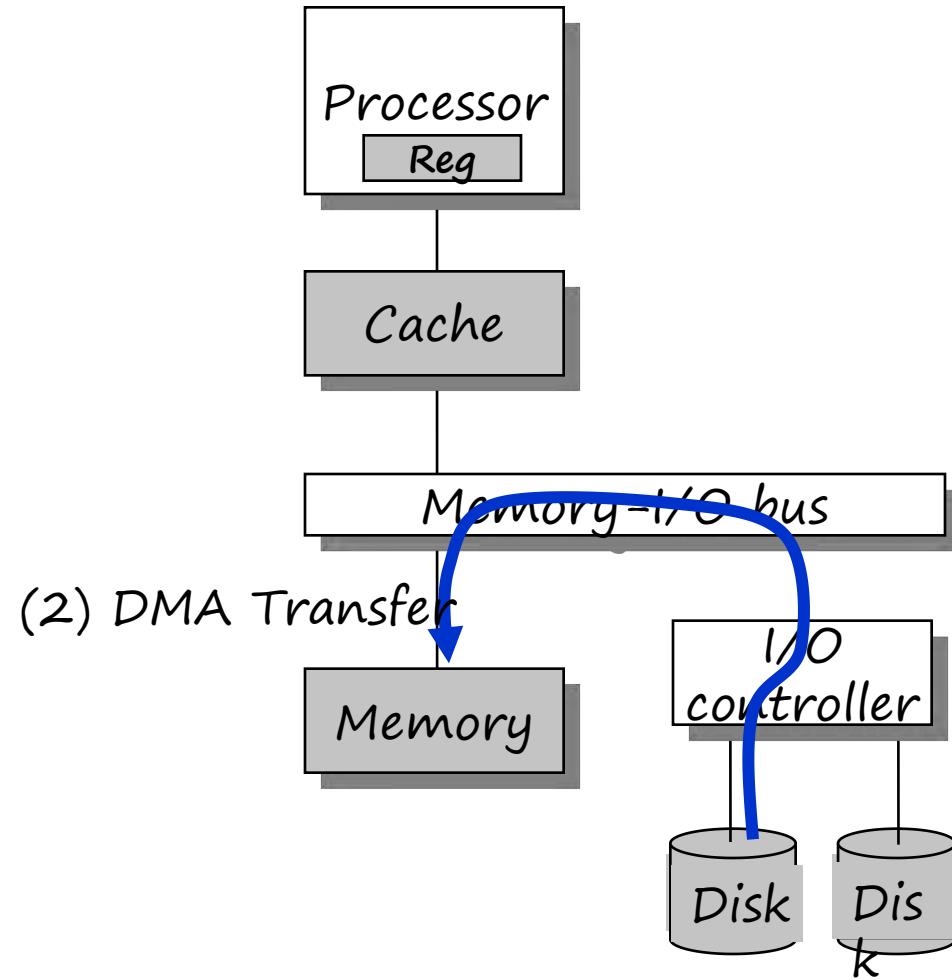
- Processor Signals Controller
  - Read block of length P starting at disk address X and store starting at memory address Y



# Servicing a Page Fault

---

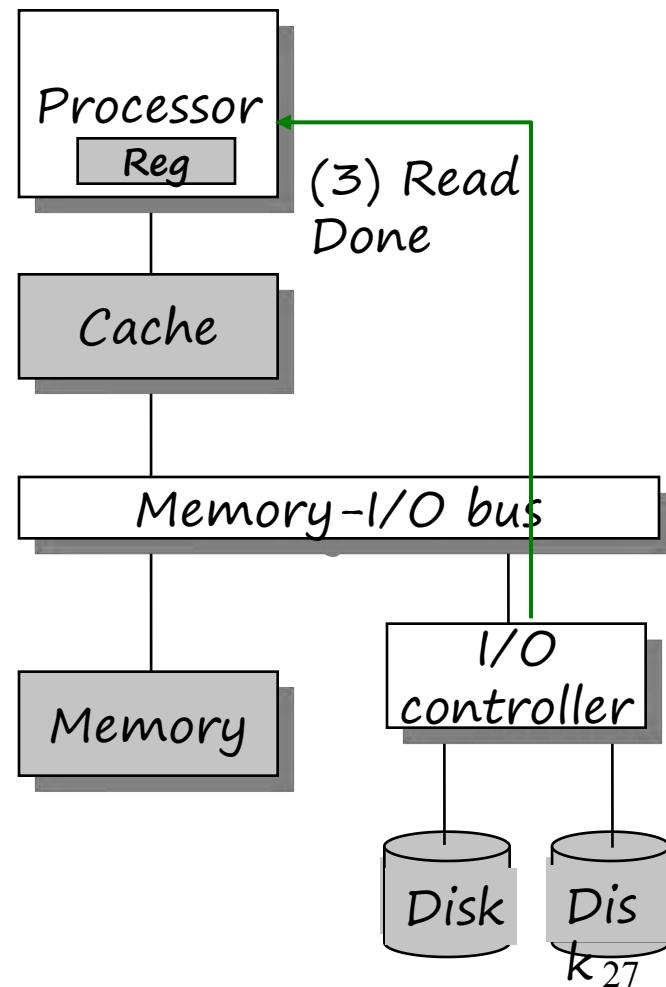
- Read Occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller



# Servicing a Page Fault

---

- I / O Controller Signals Completion
  - Interrupt processor
  - OS resumes suspended process



# Locality to the Rescue Again!

---

- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the **working set**
  - Programs with better **temporal** locality will have smaller working sets

# Locality to the Rescue Again!

---

- If (working set size < main memory size)
  - Good performance for one process after compulsory misses
- If ( SUM(working set sizes) > main memory size )
  - **Thrashing:** Performance meltdown where pages are swapped (copied) in and out continuously

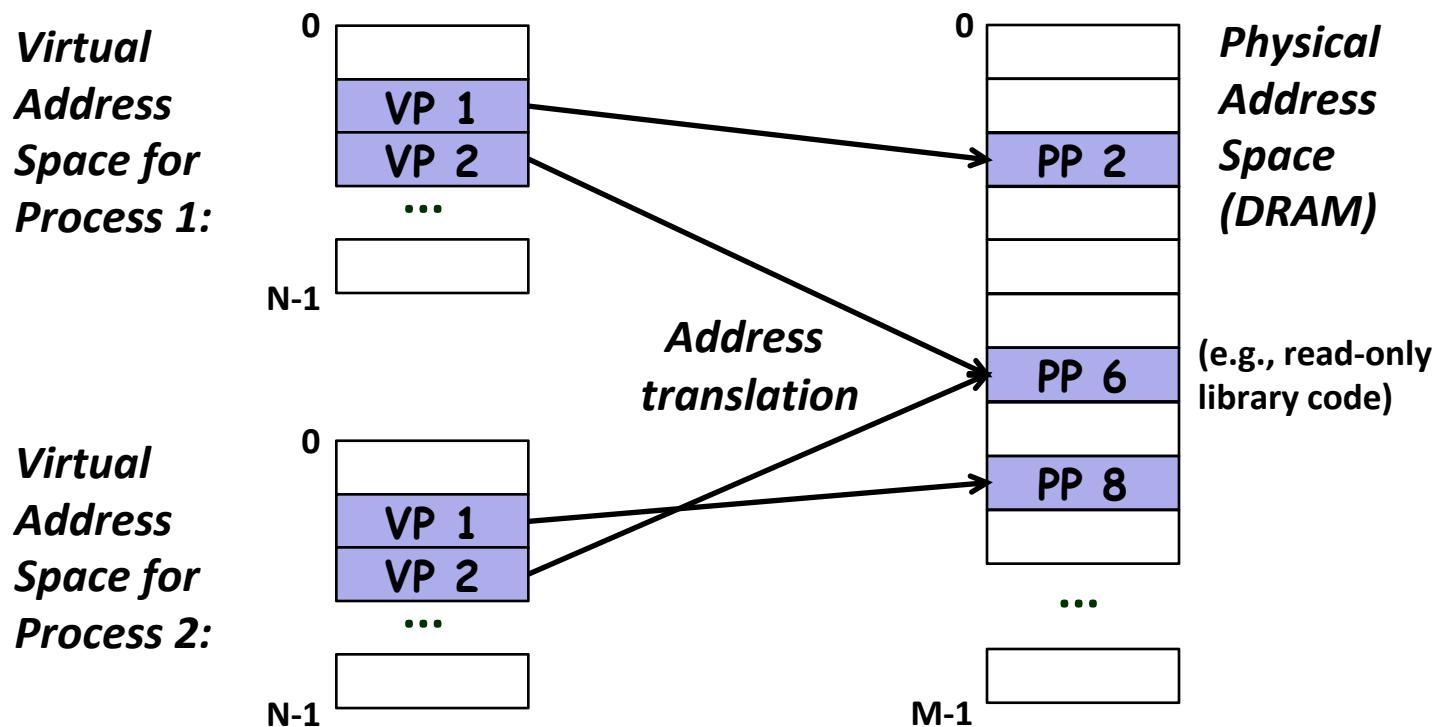
# Why Virtual Memory (VM)?

---

- Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
- Isolates address spaces
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information

# VM as a Tool for Memory Management

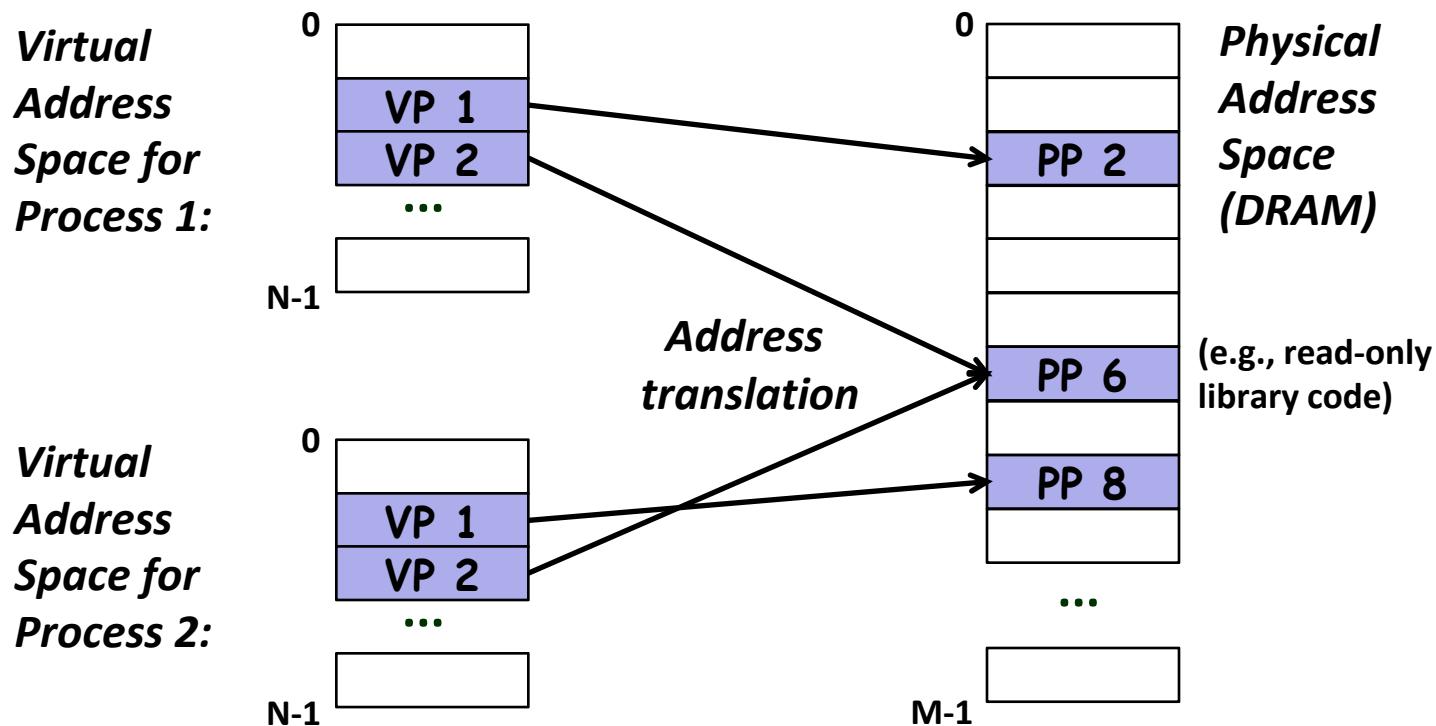
- Key idea: each process has its **own** virtual address space
  - It can view memory as a simple linear array



# VM as a Tool for Memory Management

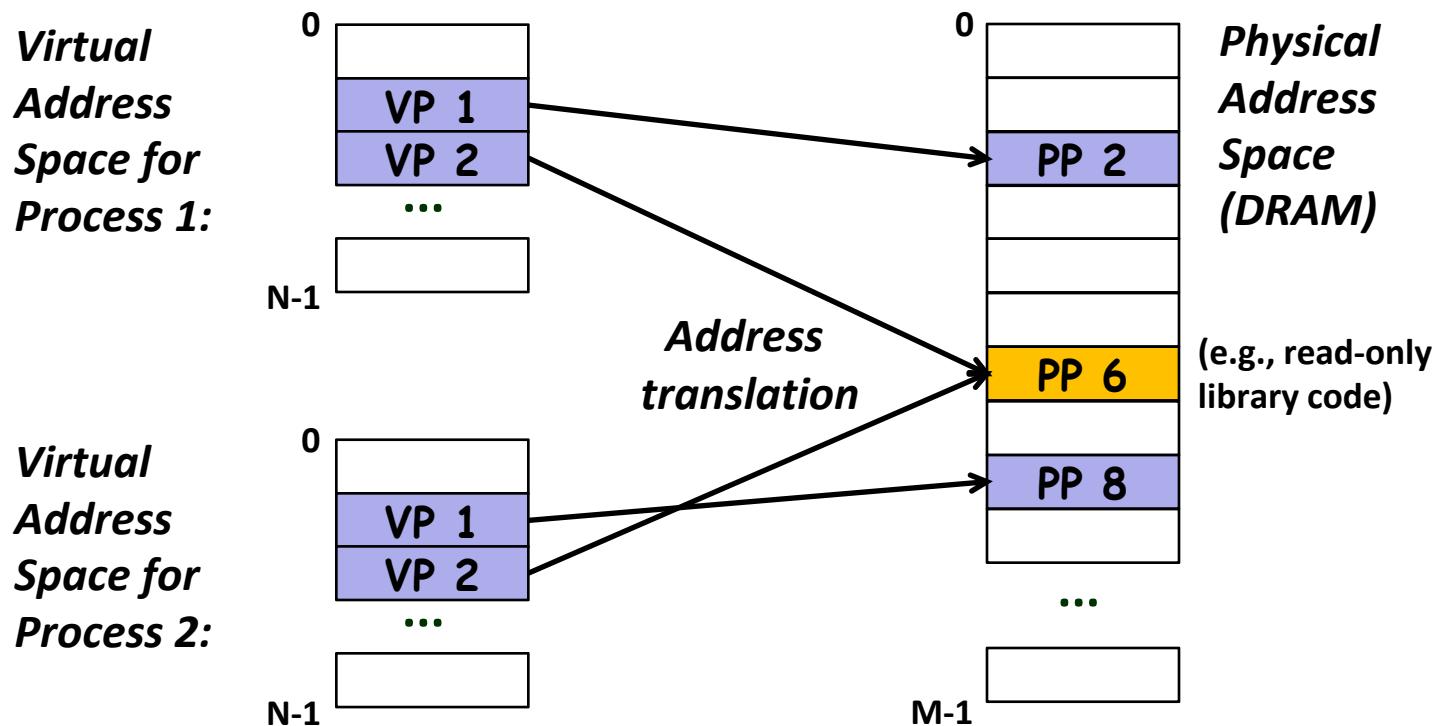
- Memory **allocation**

- Each virtual page can be mapped to any physical page
- A virtual page can be stored in different physical pages at different times



# VM as a Tool for Memory Management

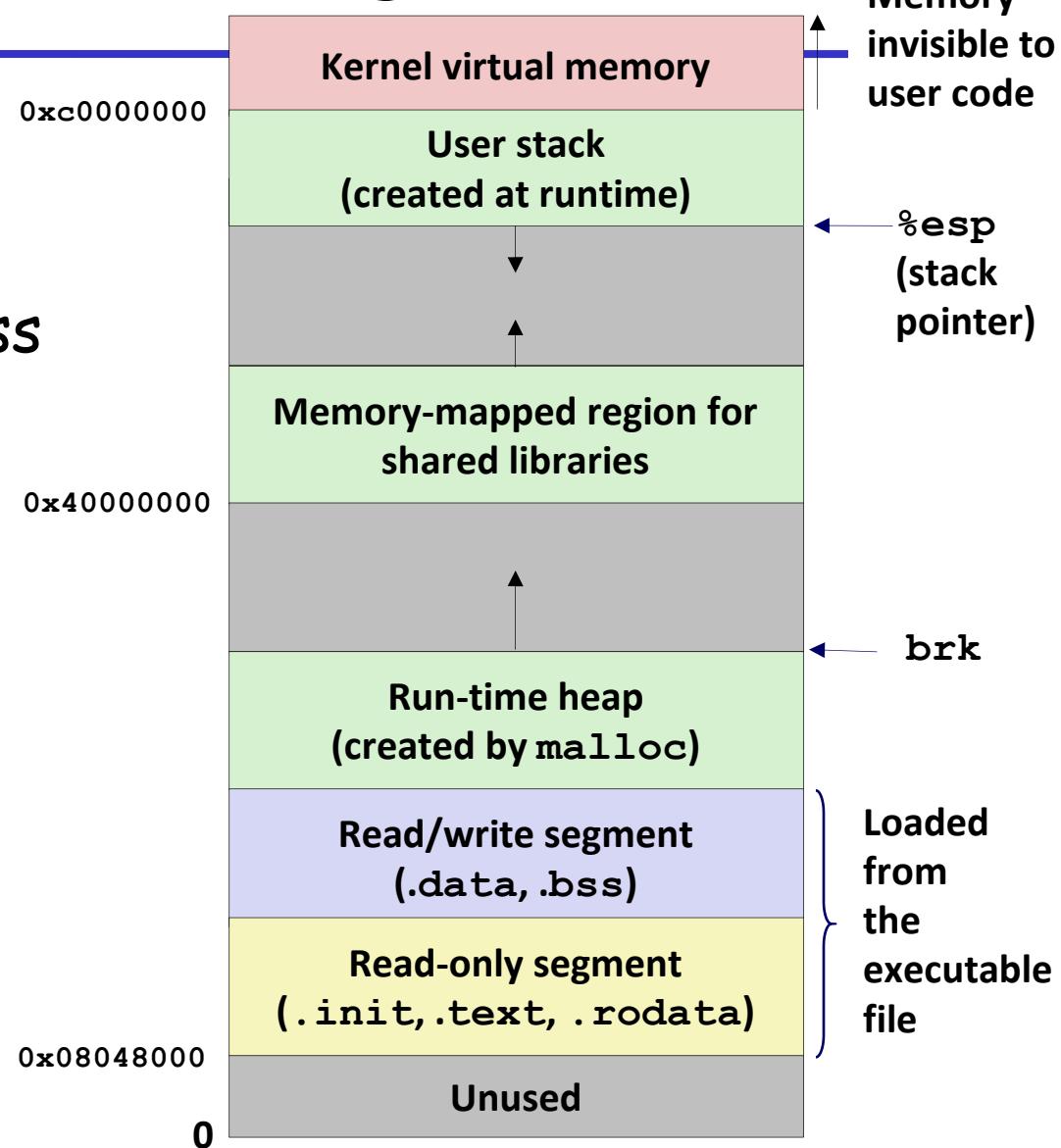
- **Sharing** code and data among processes
  - Map virtual pages to the same physical page (e.g. PP 6)



# Simplifying Linking and Loading

- **Linking**

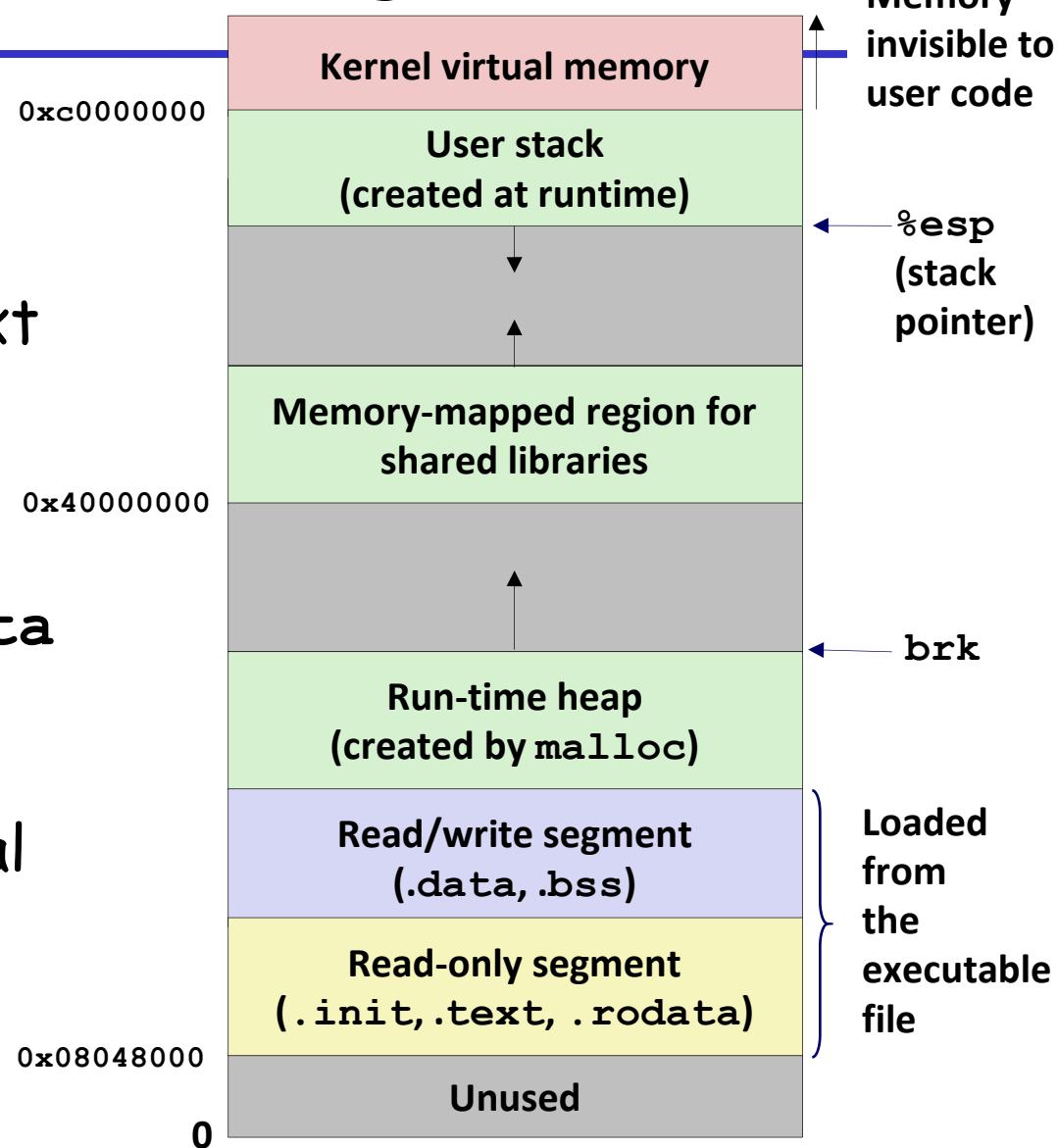
- Each program has **similar** virtual address space
- Code, stack, and shared libraries always start at the **same** address



# Simplifying Linking and Loading

- Loading

- `execve()` allocates virtual pages for `.text` and `.data` sections
  - = creates PTEs marked as **invalid**
- The `.text` and `.data` sections are copied, page by page, **on demand** by the virtual memory system



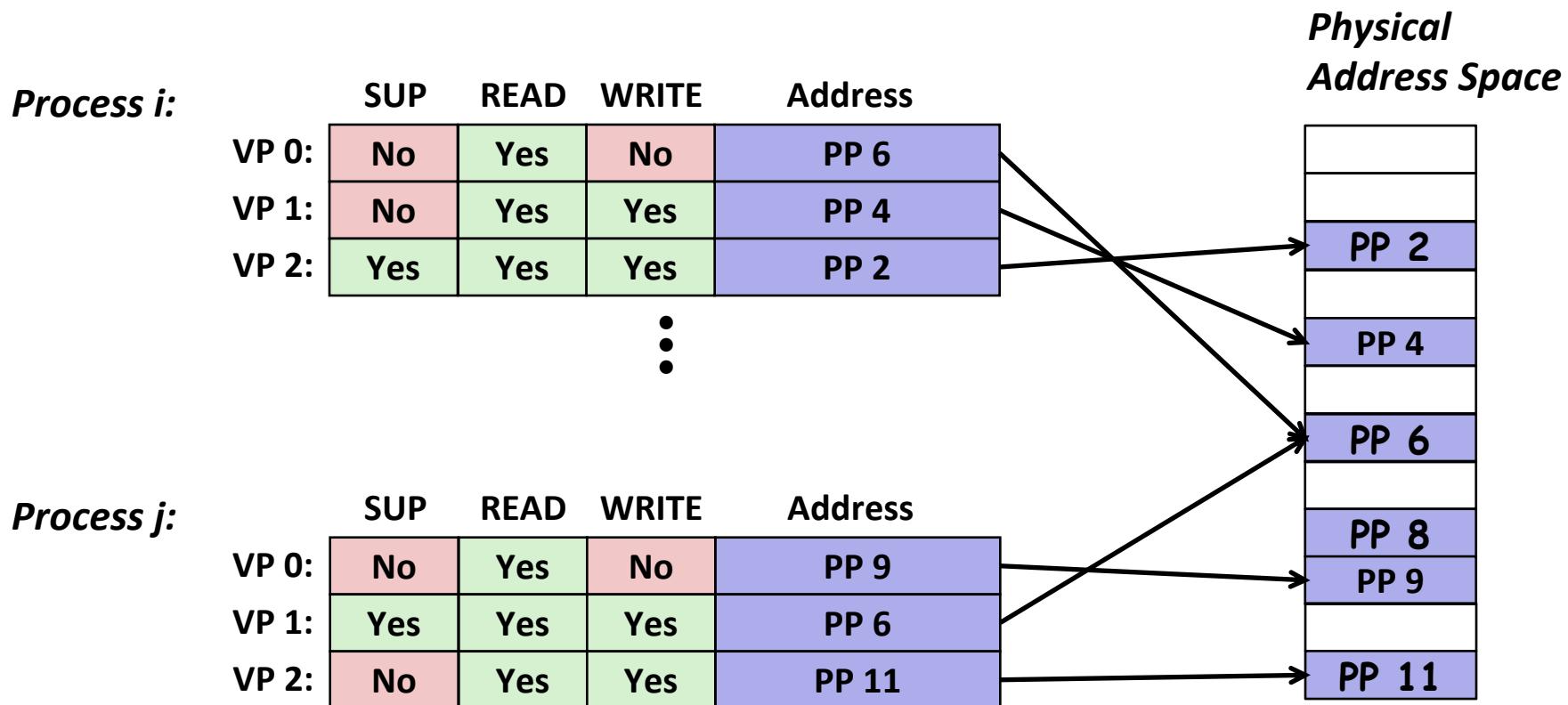
# Why Virtual Memory (VM)?

---

- Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
- Isolates address spaces
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information

# VM as a Tool for Memory Protection

- Extend PTEs with permission bits
  - The same physical page has different permission for different process



# VM as a Tool for Memory Protection

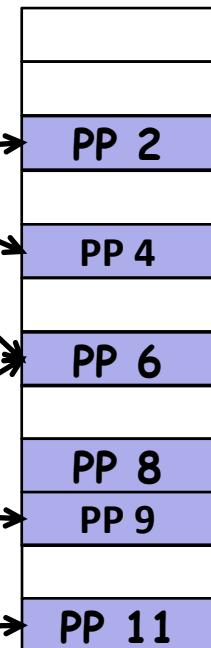
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)

Process i:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2

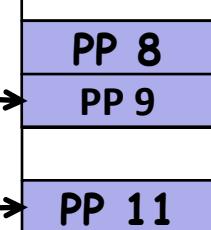
⋮

Physical Address Space



Process j:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11



# Virtual Memory

# Outline

---

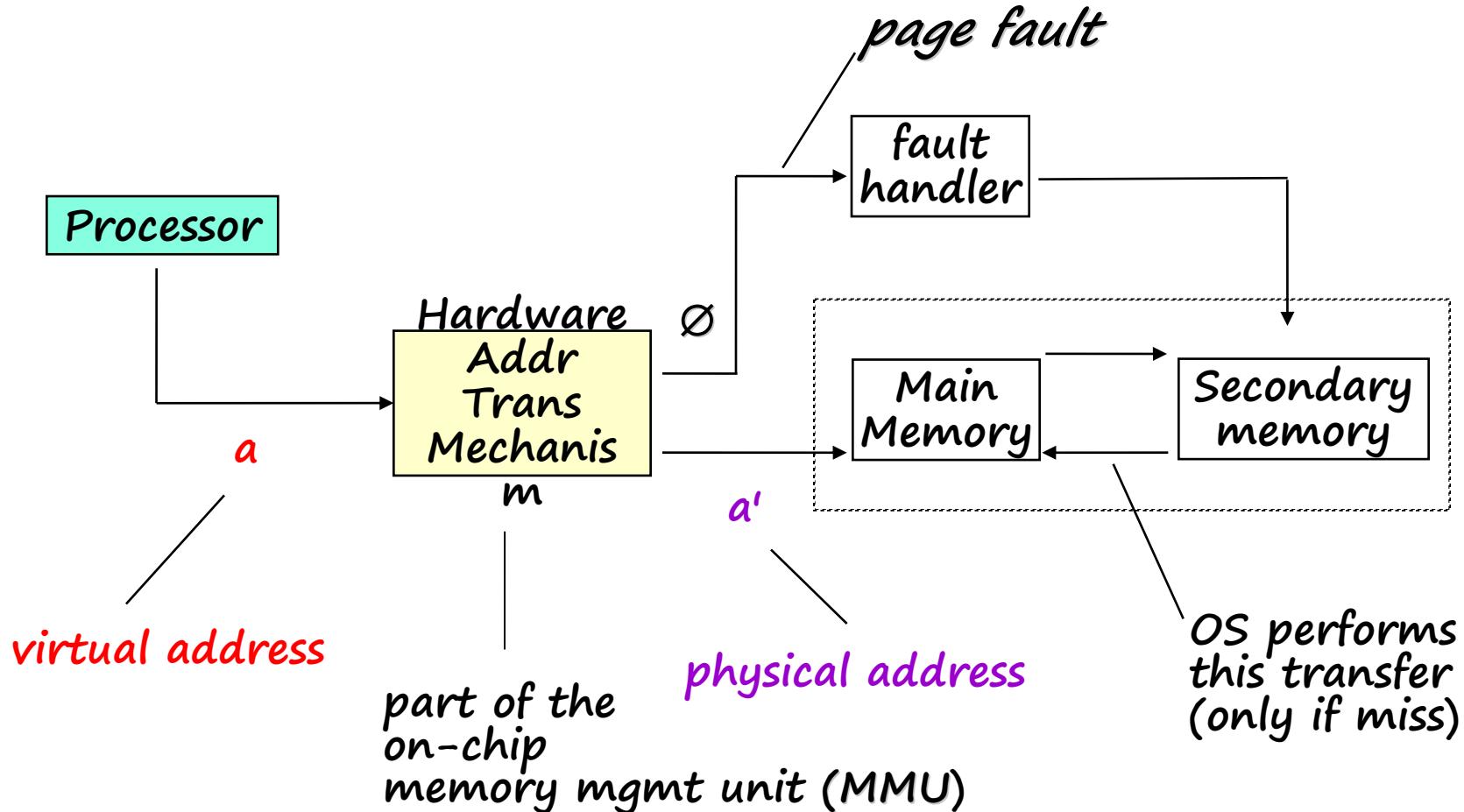
- Address translation
- Accelerating translation
  - with a TLB
  - Multilevel page tables
- Suggested reading: 10.6

# Address Translation

---

- $V = \{0, 1, \dots, N-1\}$  virtual address space
- $P = \{0, 1, \dots, M-1\}$  physical address space
- ~~$N > M$~~
- Address Translation
- MAP:  $V \rightarrow P \cup \{\emptyset\}$  address mapping function
- $\text{MAP}(a) = a'$  if data at virtual address  $a$  is present at physical address  $a'$  in  $P$   
 $= \emptyset$  if data at virtual address  $a$  is not present in  $P$  (invalid or on disk)

# Address Translation



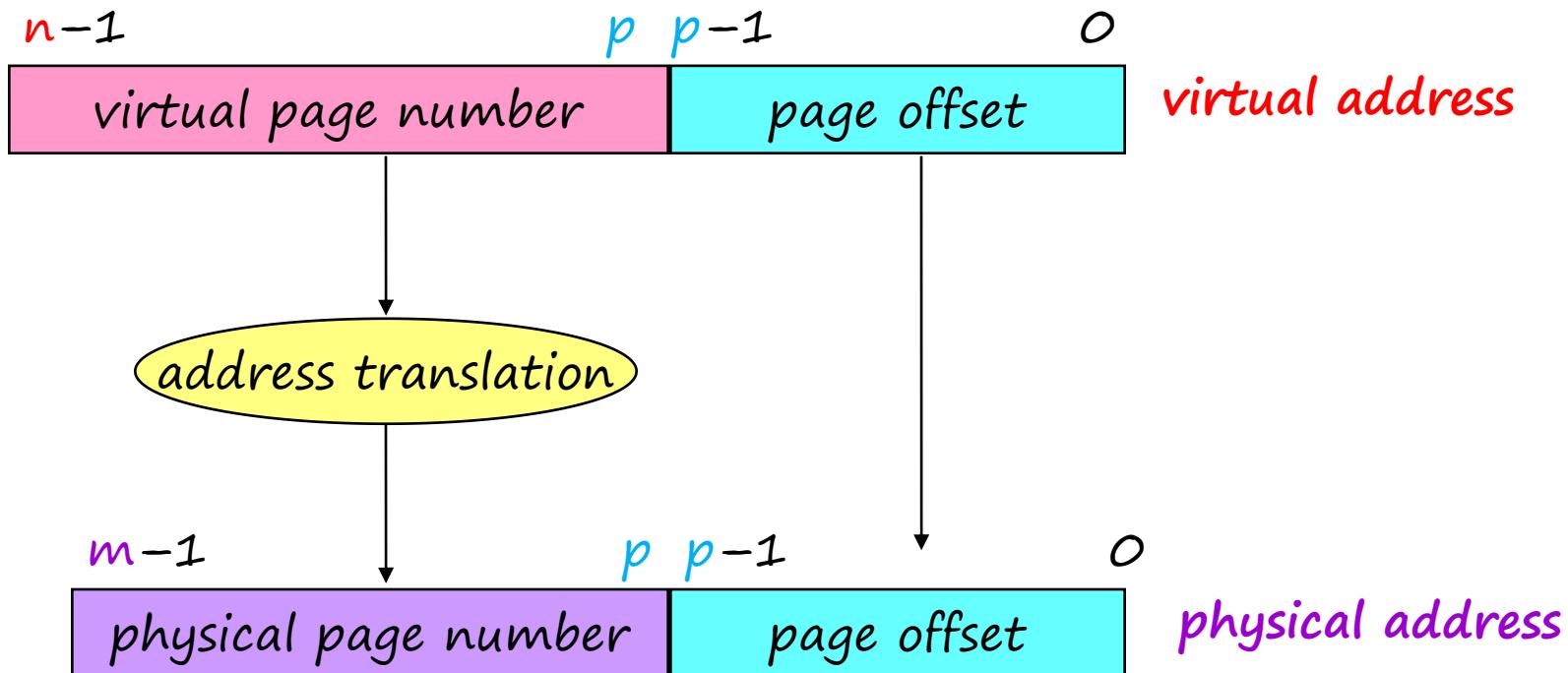
# Address Translation

---

- Basic Parameters
  - $N = 2^n$  = Virtual address limit
  - $M = 2^m$  = Physical address limit
  - $P = 2^p$  = page size (bytes).

# Address Translation

---



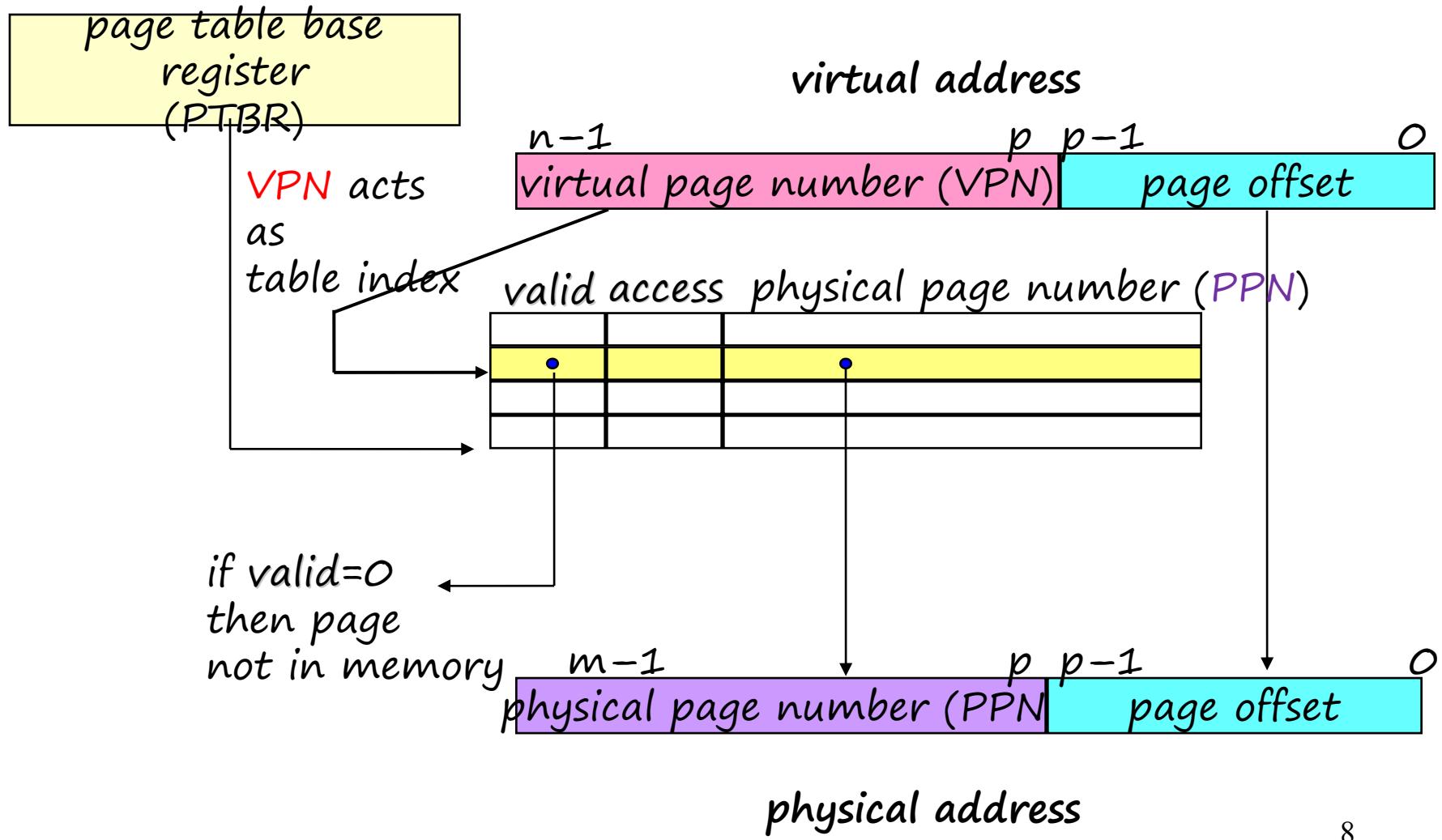
Notice that the page offset bits don't change as a result of translation

# Address Translation

---

- Basic Parameters
  - $N = 2^n$  = Virtual address limit
  - $M = 2^m$  = Physical address limit
  - $P = 2^p$  = page size (bytes).
- Components of the virtual address (VA)
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)
  - **PPN**: Physical page number

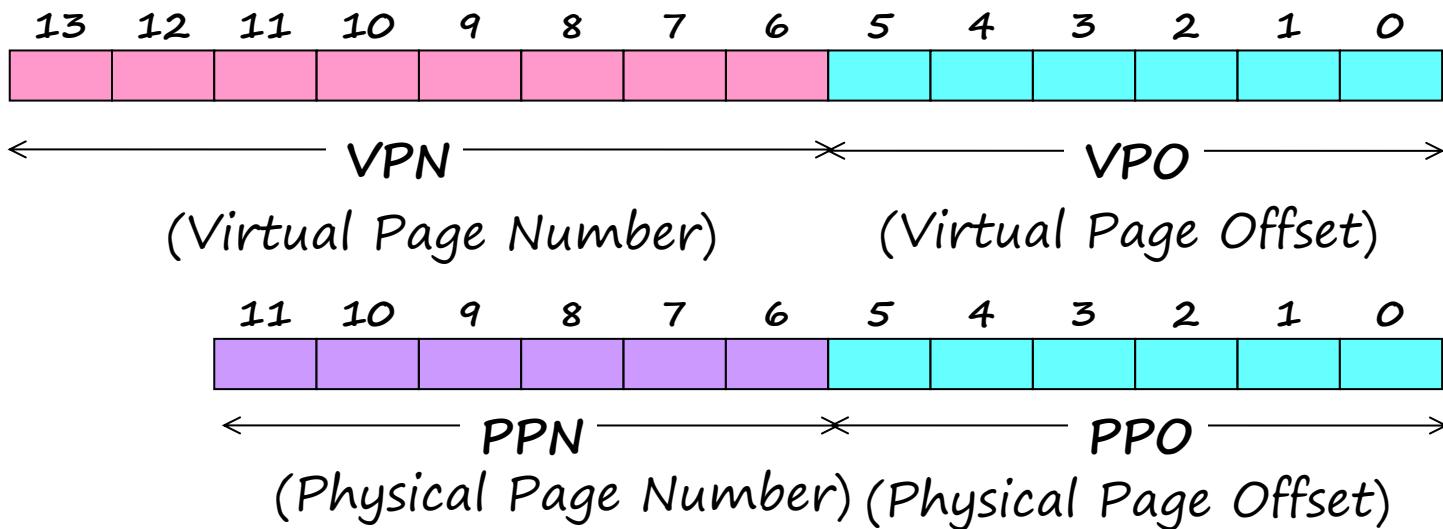
# Address Translation via Page Table



# Simple Memory System Example

---

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bits (6-bit)



# Simple Memory System Page Table

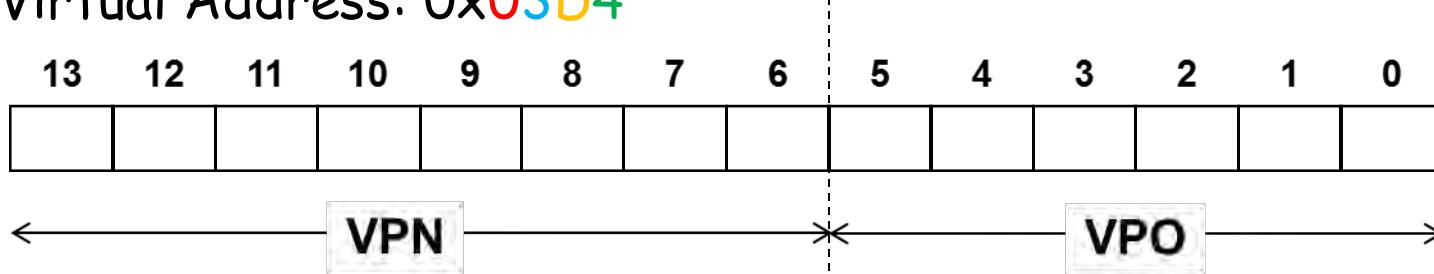
---

- Only show first 16 entries

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	-	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	-	0
04	-	0	0C	-	0
05	16	1	0D	2D	1
06	-	0	0E	11	1
07	-	0	0F	0D	1

# Address Translation Example

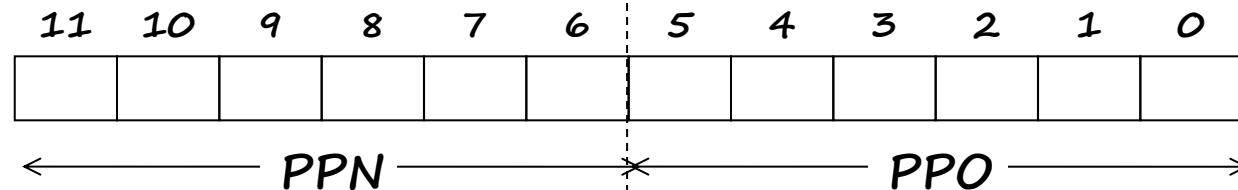
Virtual Address: 0x03D4



VPN:

VPO:

Page Fault? \_\_\_\_\_



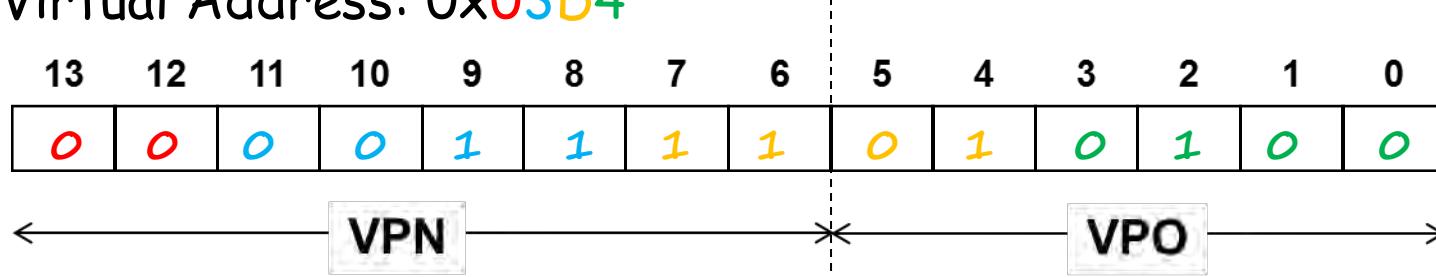
PPN:

PPO: \_\_\_\_\_

PA:

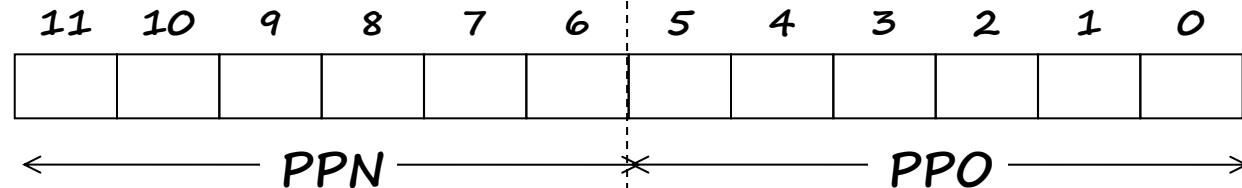
# Address Translation Example

Virtual Address: 0x03D4



VPN: 0x0f VPO: 0x14

Page Fault? \_\_\_\_\_



PPN:

PPO: 0x14

PA:

# Simple Memory System Page Table

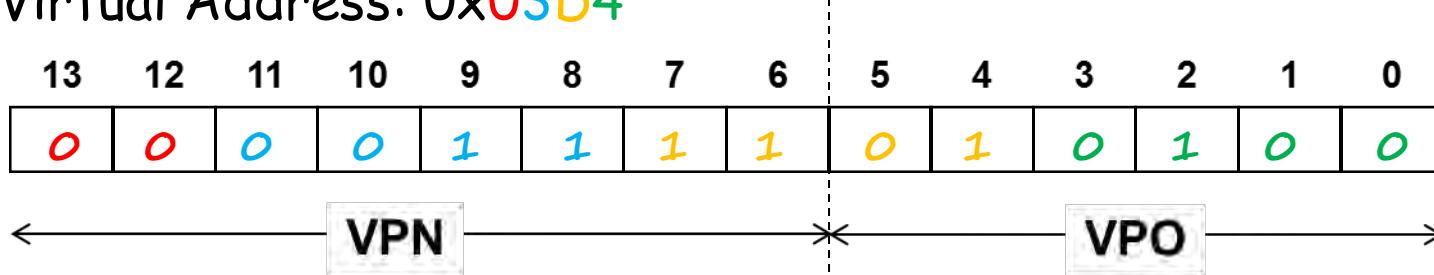
---

- Only show first 16 entries

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	-	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	-	0
04	-	0	0C	-	0
05	16	1	0D	2D	1
06	-	0	0E	11	1
07	-	0	0F	0D	1

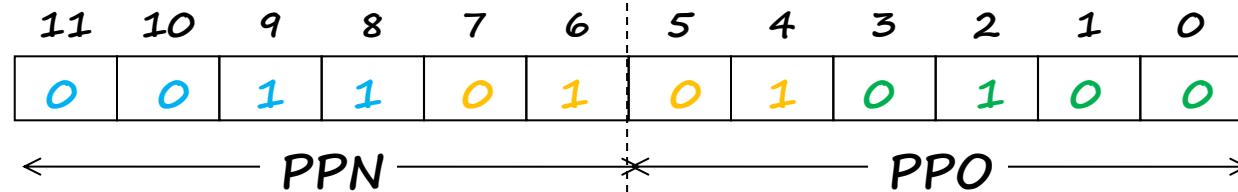
# Address Translation Example

Virtual Address: 0x03D4



VPN: 0x0f VPO: 0x14

Page Fault? No

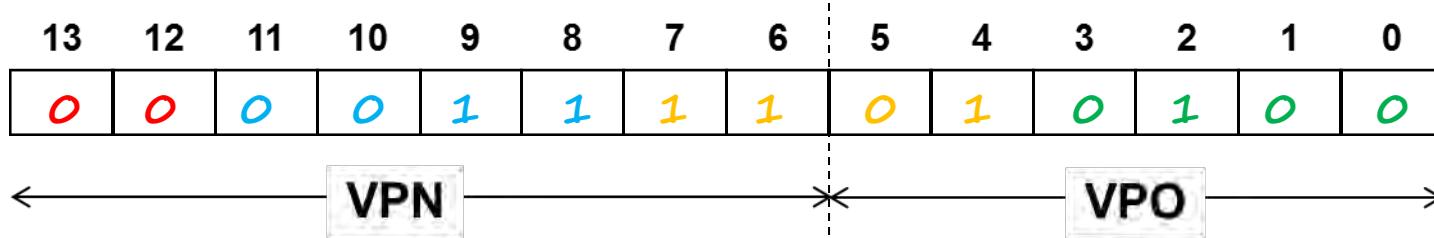


PPN: 0xD VPO: 0x14

PA: 0x

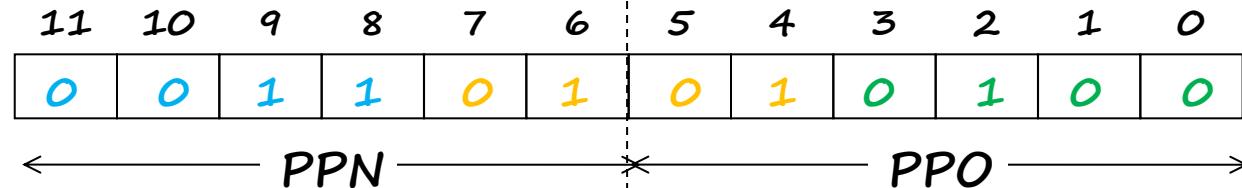
# Address Translation Example

Virtual Address: 0x03D4



VPN: 0x0f VPO: 0x14

Page Fault? No



PPN: 0x0D VPO: 0x14

PA: 0x354

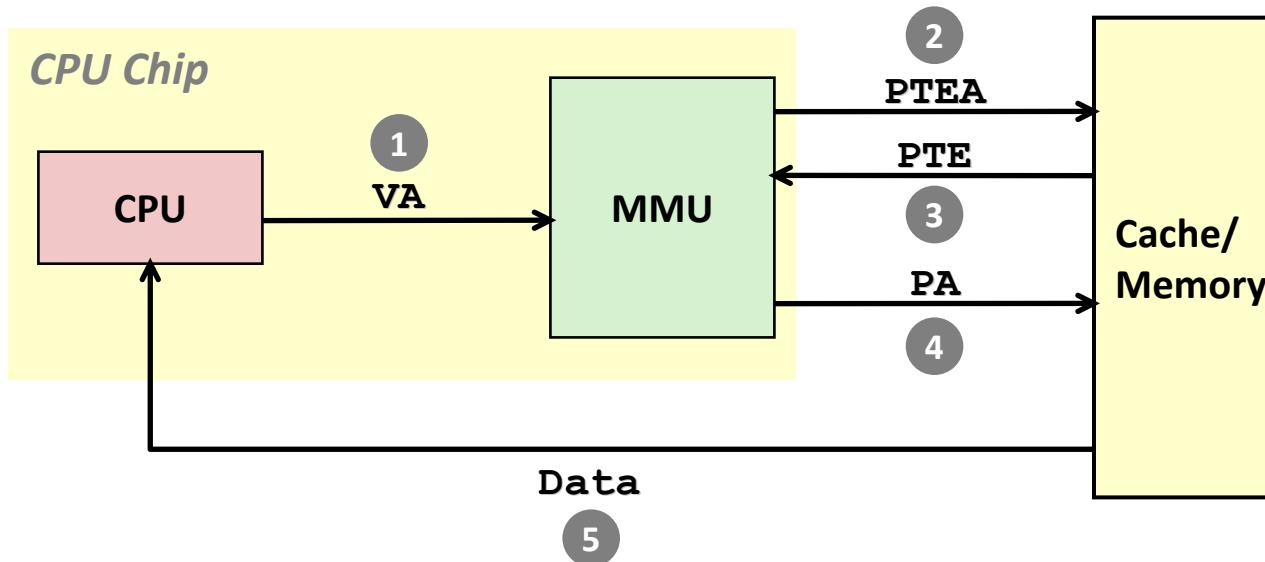
VA: virtual address

PTEA: page table entry address

PTE: page table entry

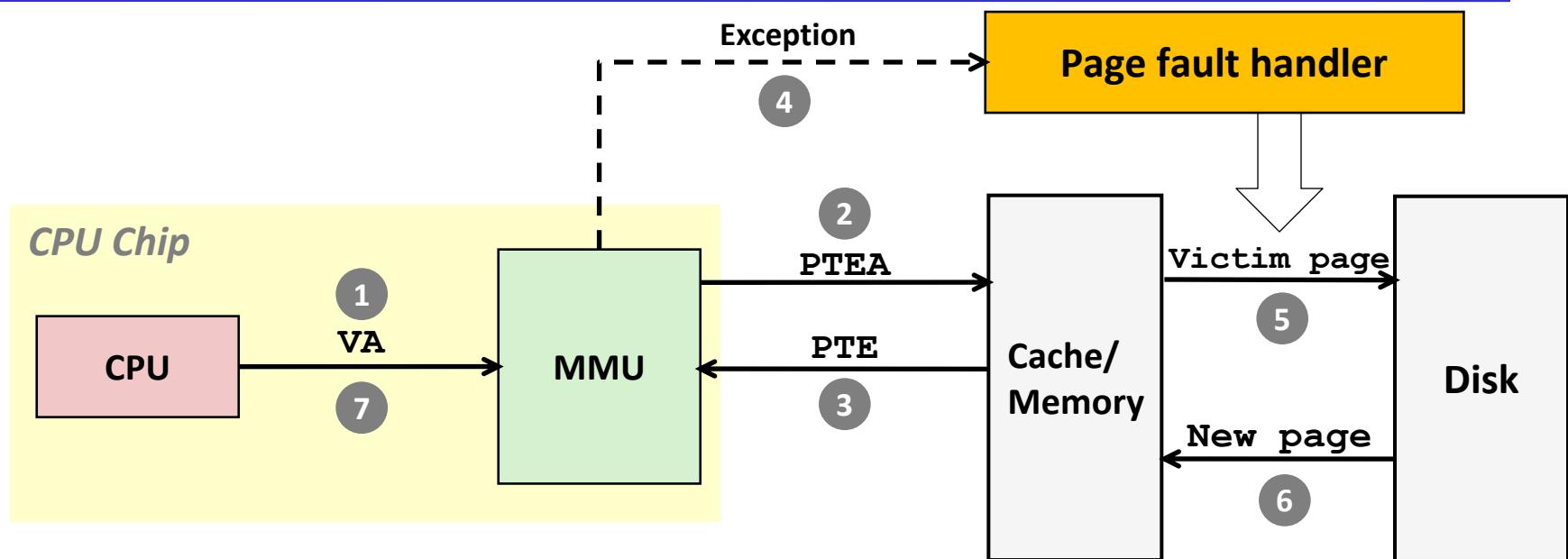
PA: physical address

# Page Hit



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

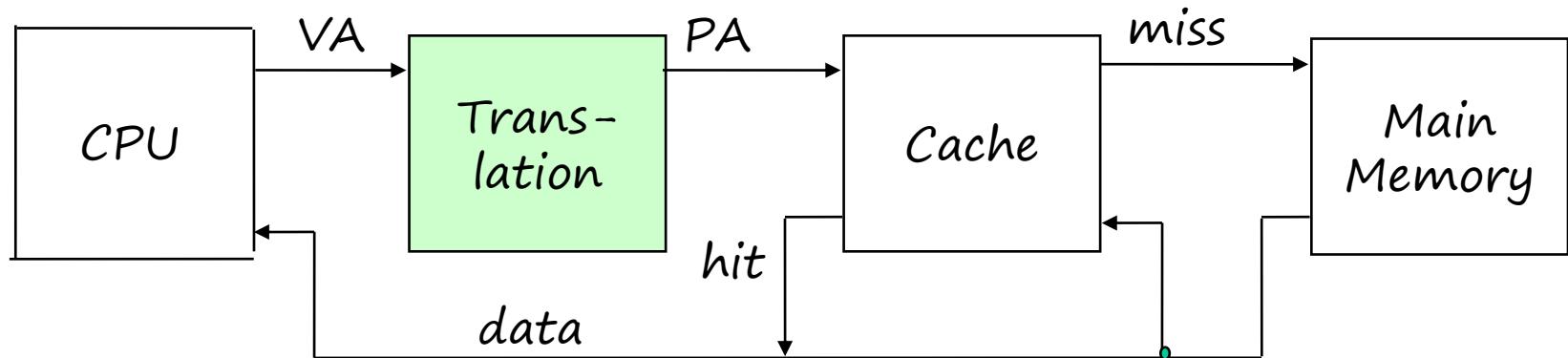
# Page Faults



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

# Integrating Caches and VM

---



# Integrating Caches and VM

---

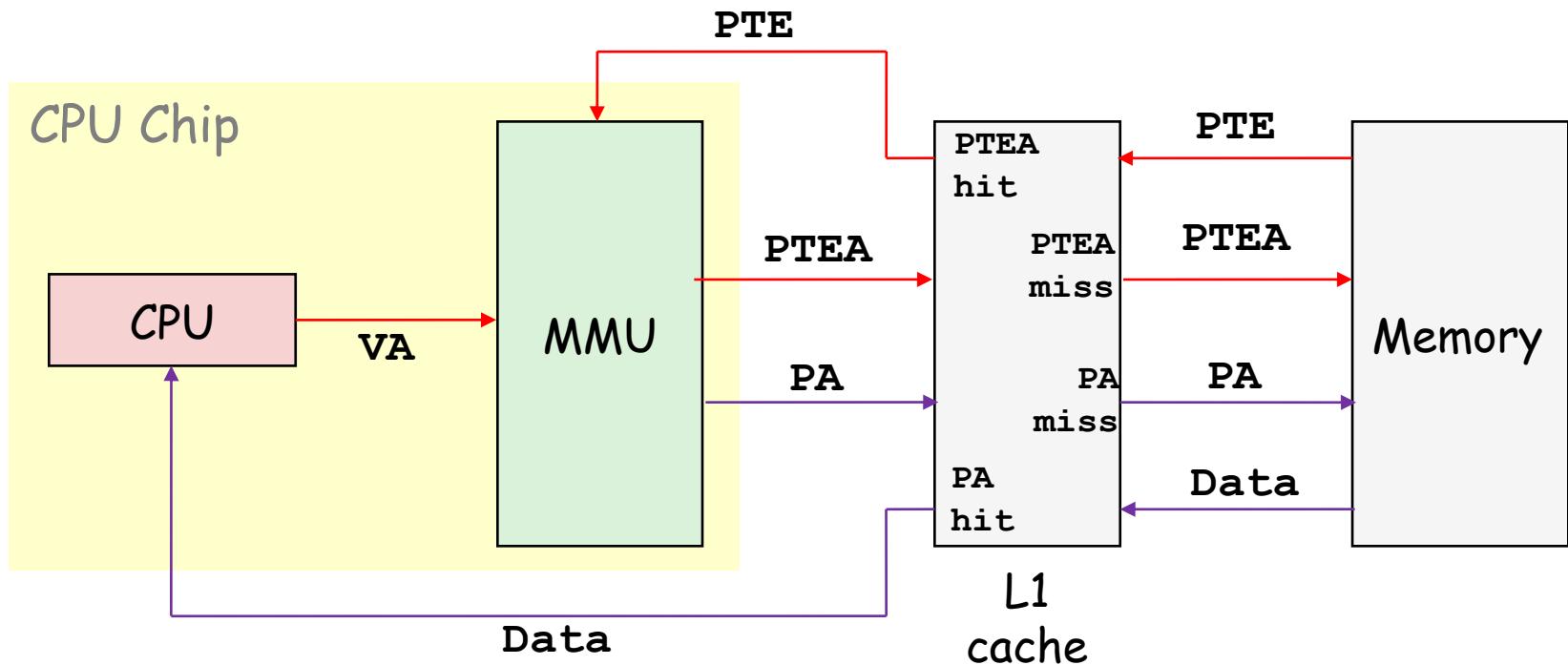
- Most Caches “Physically Addressed”
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time
  - Allows multiple processes to share pages
  - Cache doesn’t need to be concerned with protection issues
    - Access rights checked as part of address translation

# Integrating Caches and VM

---

- Perform Address Translation Before Cache Lookup
  - But this could involve a memory access itself (of the PTE)
  - Of course, page table entries can also become cached

# Integrating Caches and VM



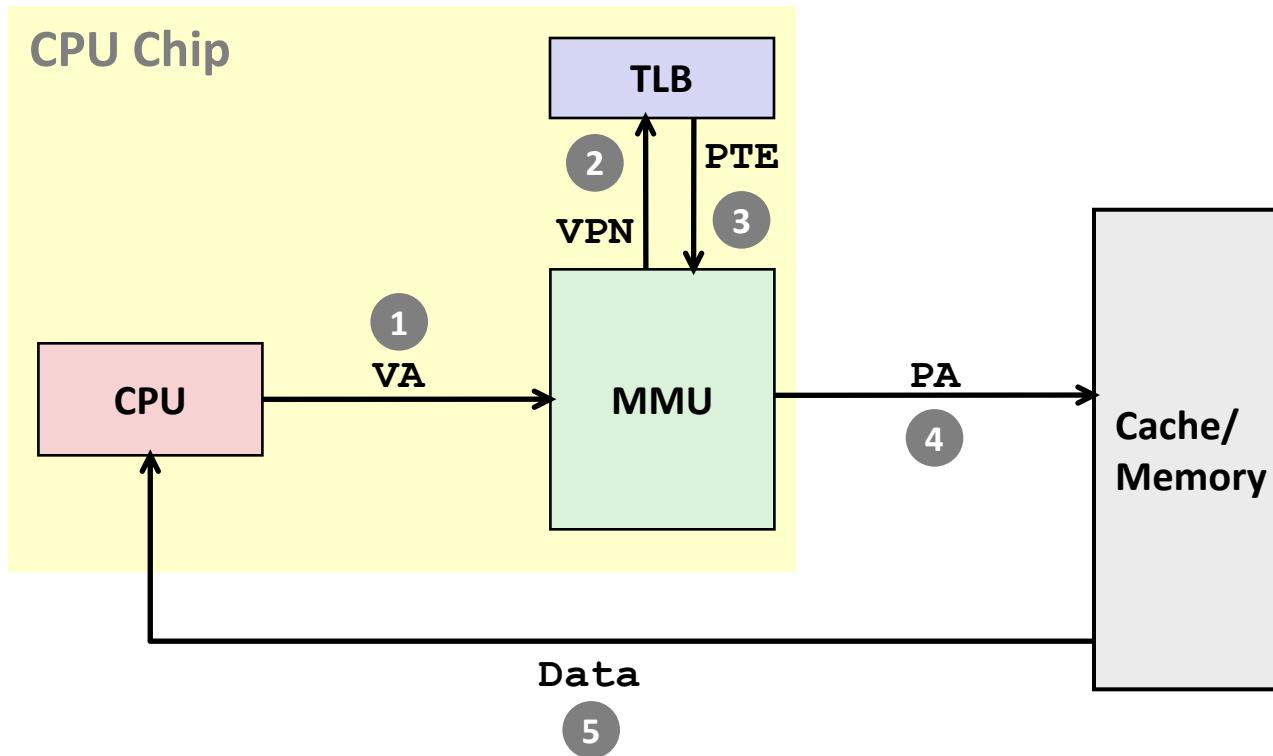
# Speeding up Translation with a TLB

---

- “Translation Lookaside Buffer” (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers

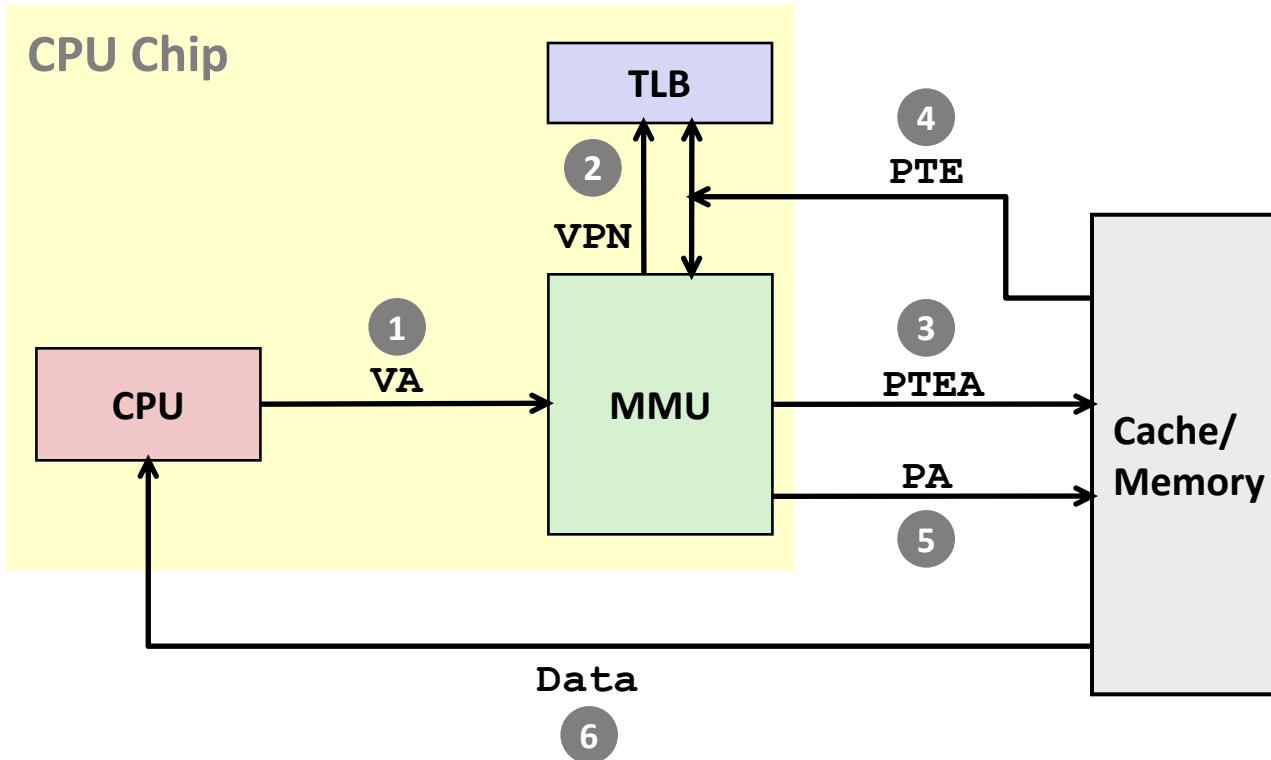
# TLB Hit

---



A TLB hit eliminates a memory access

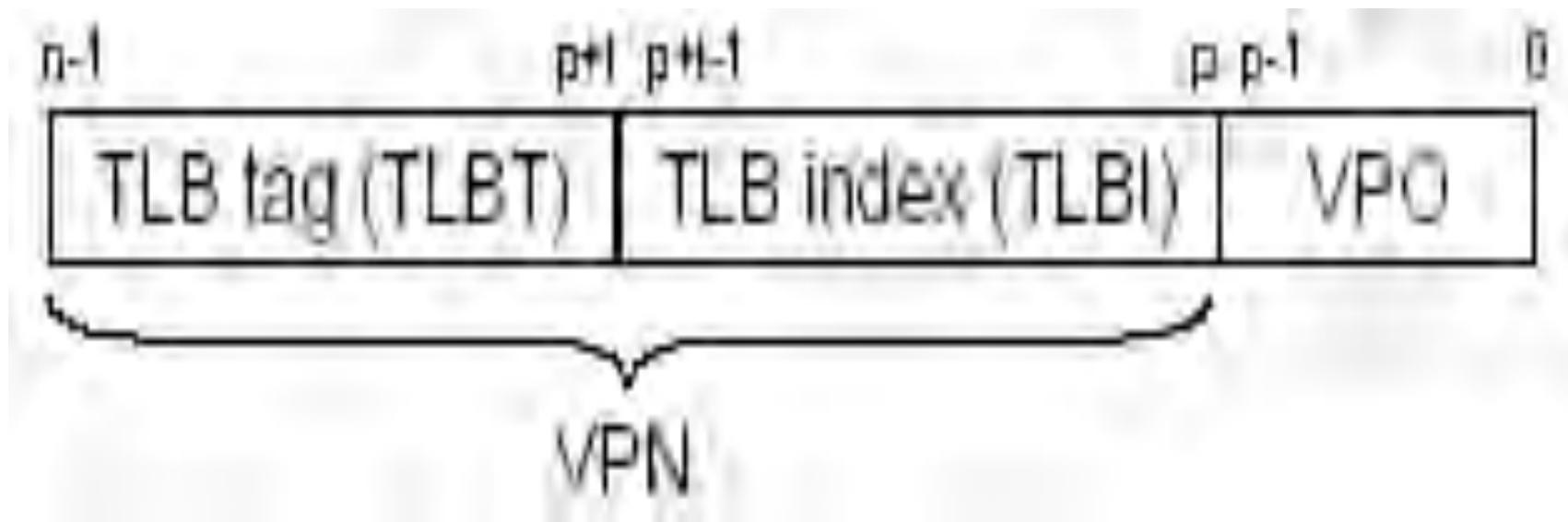
# TLB Miss



A TLB miss incurs an additional memory access (PTE)  
Fortunately, TLB misses are rare. Why?

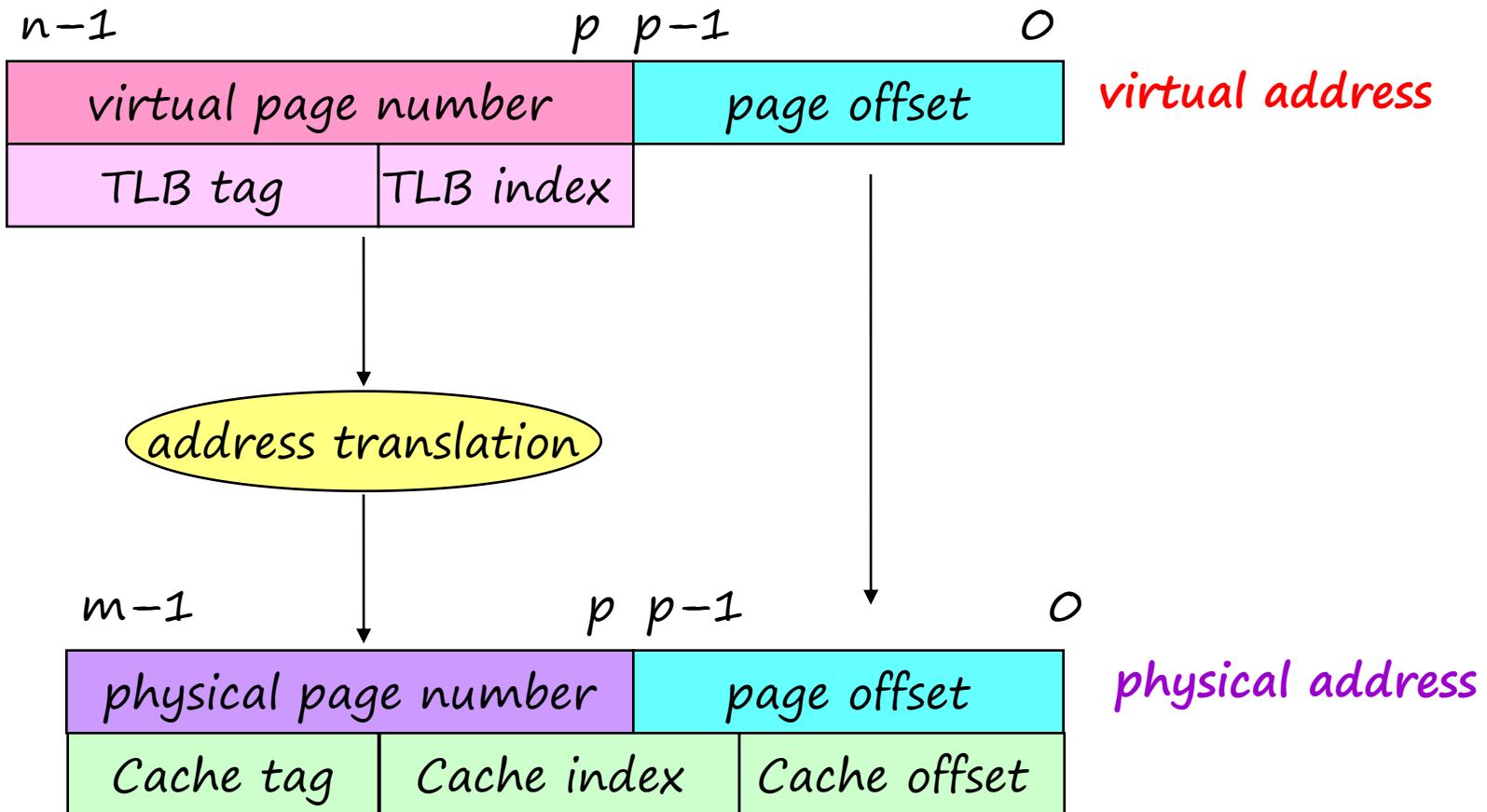
# Speeding up Translation with a TLB

---



# Address Translation

---

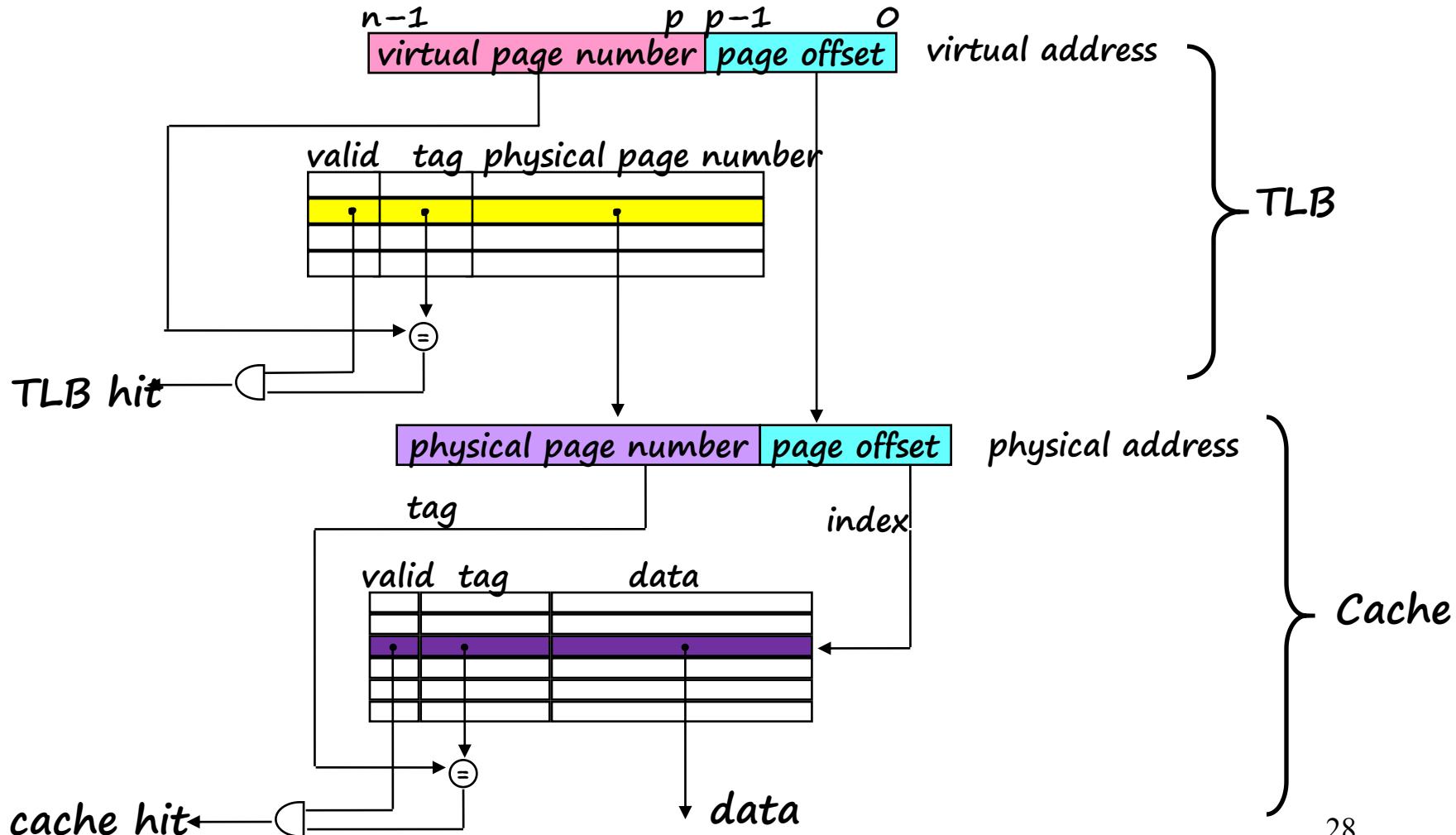


# Address Translation

---

- Components of the virtual address (VA)
  - VPO: Virtual page offset
  - VPN: Virtual page number
  - TLBI: TLB index
  - TLBT: TLB tag
- Components of the physical address (PA)
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number
  - CO: Byte offset within cache line
  - CI: Cache index
  - CT: Cache tag

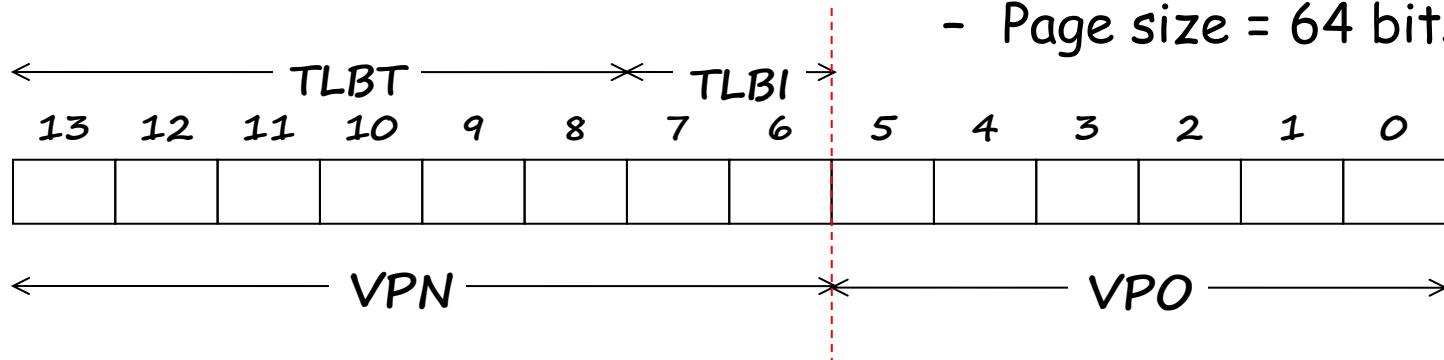
# Speeding up Translation with a TLB



# Simple Memory System TLB

---

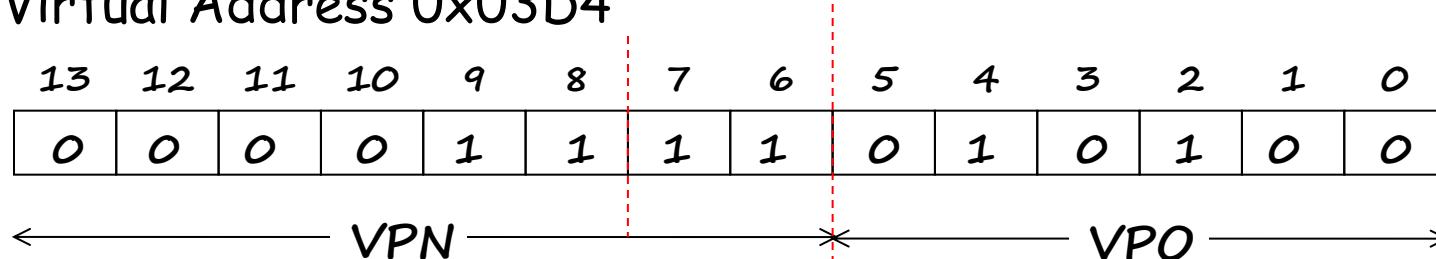
- TLB
  - 16 entries
  - 4-way associative (2-bit)
- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bits (6-bit)



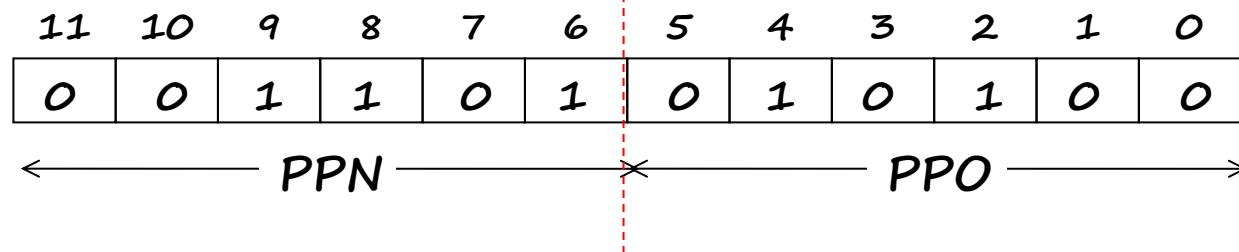
Set	Tag	PPN	Valid									
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

# Address Translation Example

Virtual Address 0x03D4



VPN: 0x0f TLBI: 0x03 TLBT: 0x03 TLB Hit? \_\_\_\_\_ Page Fault? \_\_\_\_\_



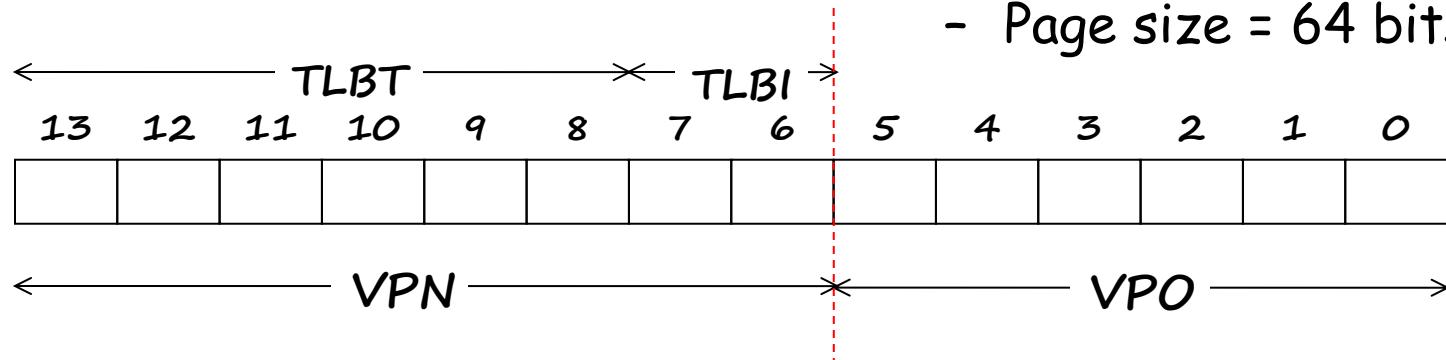
PPN:

PPO: \_\_\_\_\_

PA:

# Simple Memory System TLB

- TLB
  - 16 entries
  - 4-way associative (2-bit)
- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bits (6-bit)

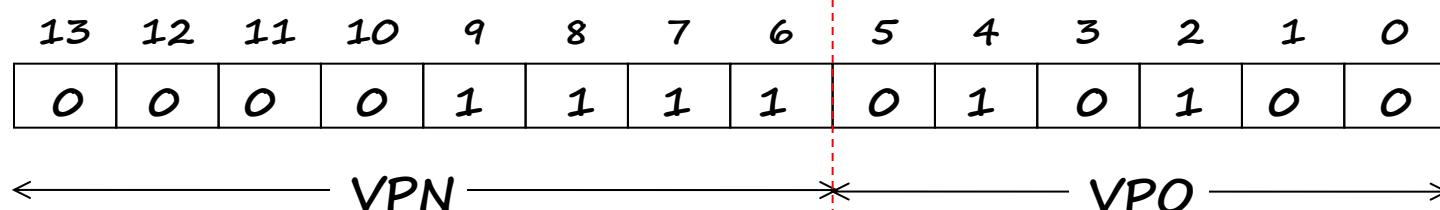


Set	Tag	PPN	Valid									
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

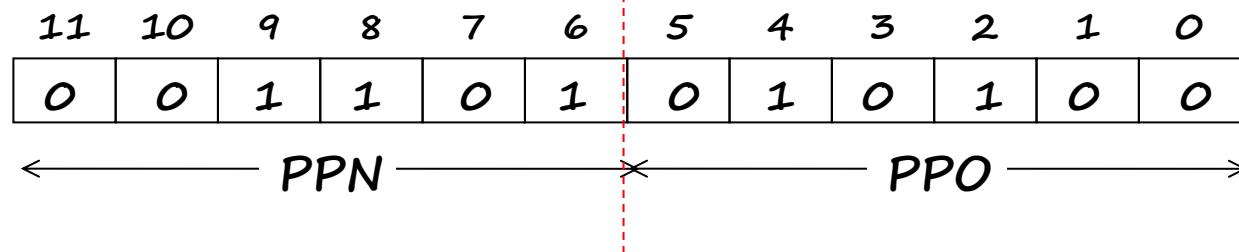
# Address Translation Example

---

Virtual Address 0x03D4



VPN: 0x0f TLBI: 0x03 TLBT: 0x03 TLB Hit? Yes Page Fault? No



PPN: 0x0D PPO: 0x14

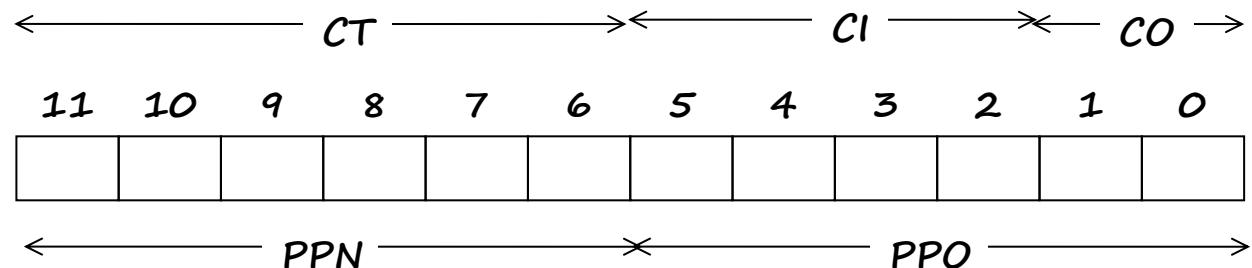
PA: 0x354

# Simple Memory System Cache

---

- Cache

- 16 lines
- 4-byte line size
- Direct mapped

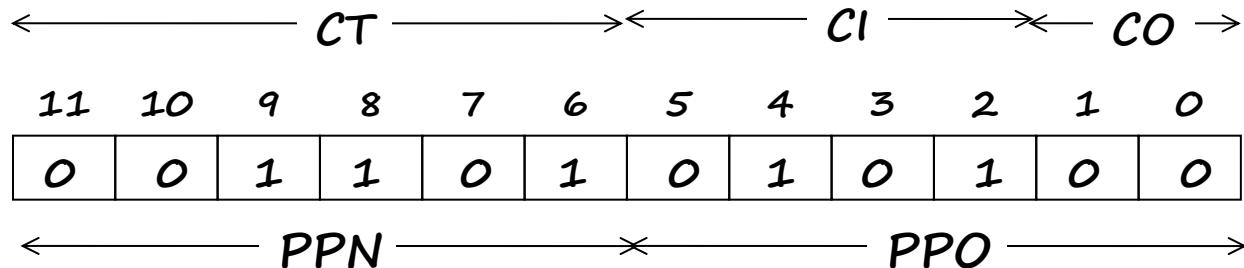


Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	-	-	-	-	9	2D	0	-	-	-	-
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	-	-	-	-	B	0B	0	-	-	-	-
4	32	1	43	6D	8F	09	C	12	0	-	-	-	-
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	-	-	-	-	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	-	-	-	-

# Address Translation Example

---

PA: 0x354

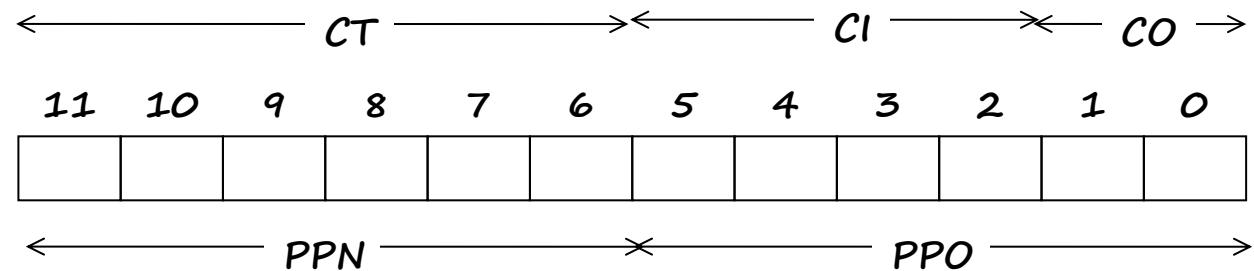


Offset: 0x0   CI: 0x05   CT: 0x0D   Hit?      Byte: 0x \_\_\_\_\_

# Simple Memory System Cache

- Cache

- 16 lines
- 4-byte line size
- Direct mapped

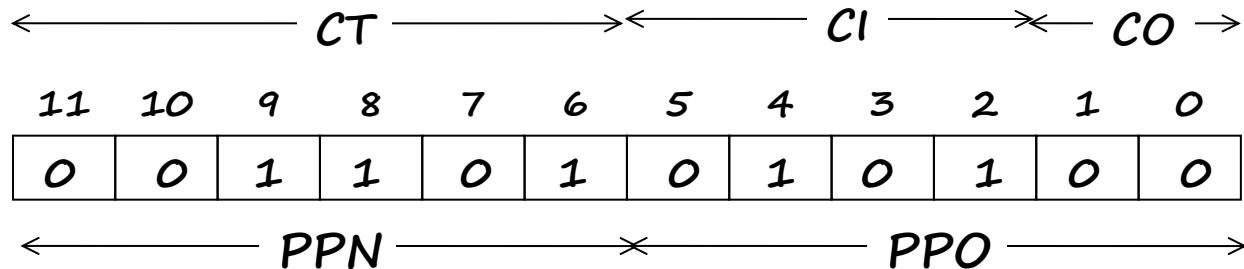


Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	-	-	-	-	9	2D	0	-	-	-	-
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	-	-	-	-	B	0B	0	-	-	-	-
4	32	1	43	6D	8F	09	C	12	0	-	-	-	-
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	-	-	-	-	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	-	-	-	-

# Address Translation Example

---

PA: 0x354



Offset: 0x0 CI: 0x05 CT: 0x0D Hit? **Yes** Byte: 0x36

# Multi-Level Page Tables

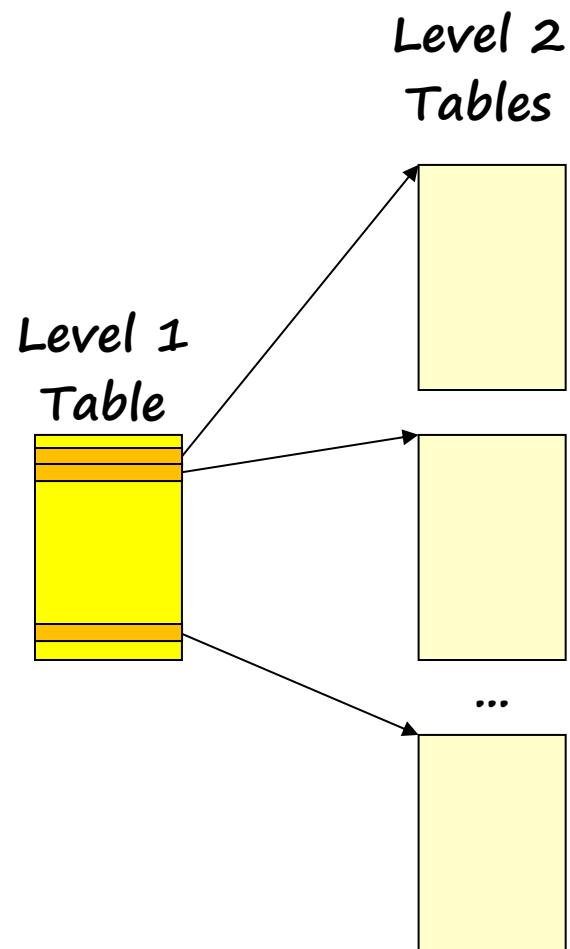
---

- Given:
  - X86: 32-bit address space  
4KB ( $2^{12}$ ) page size, 4-byte PTE
  - X86-64: 48-bit address space  
4KB ( $2^{12}$ ) page size, 8-byte PTE
- Problem:
  - X86: Would need a **4 MB** page table!
    - $2^{20} * 4$  bytes (20bit = 32bit - 12bit)
  - X86-64: Would need a **512 GB** page table!
    - $2^{30} * 8$  bytes (30bit = 48bit - 12bit)

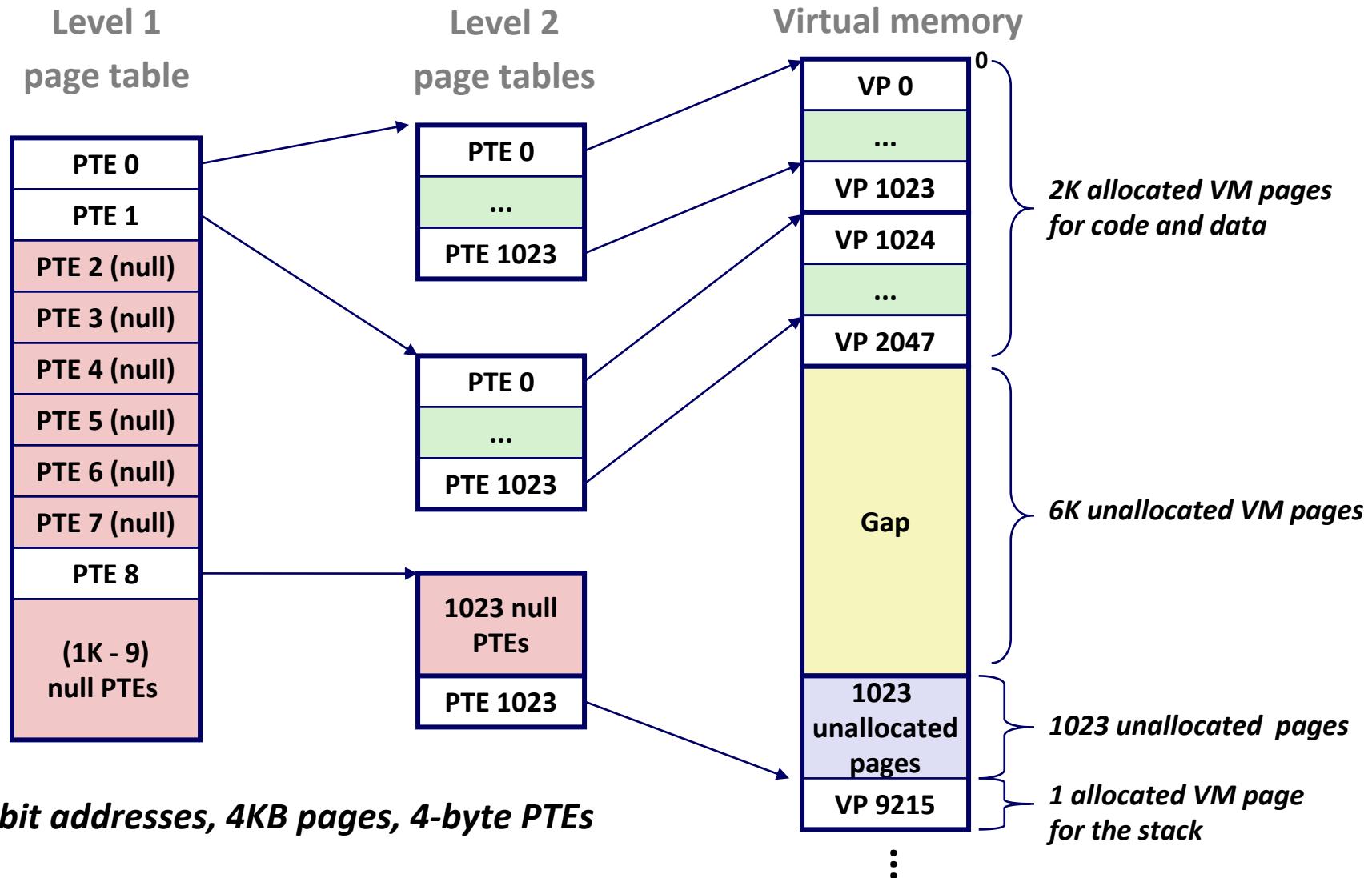
# Multi-Level Page Tables

---

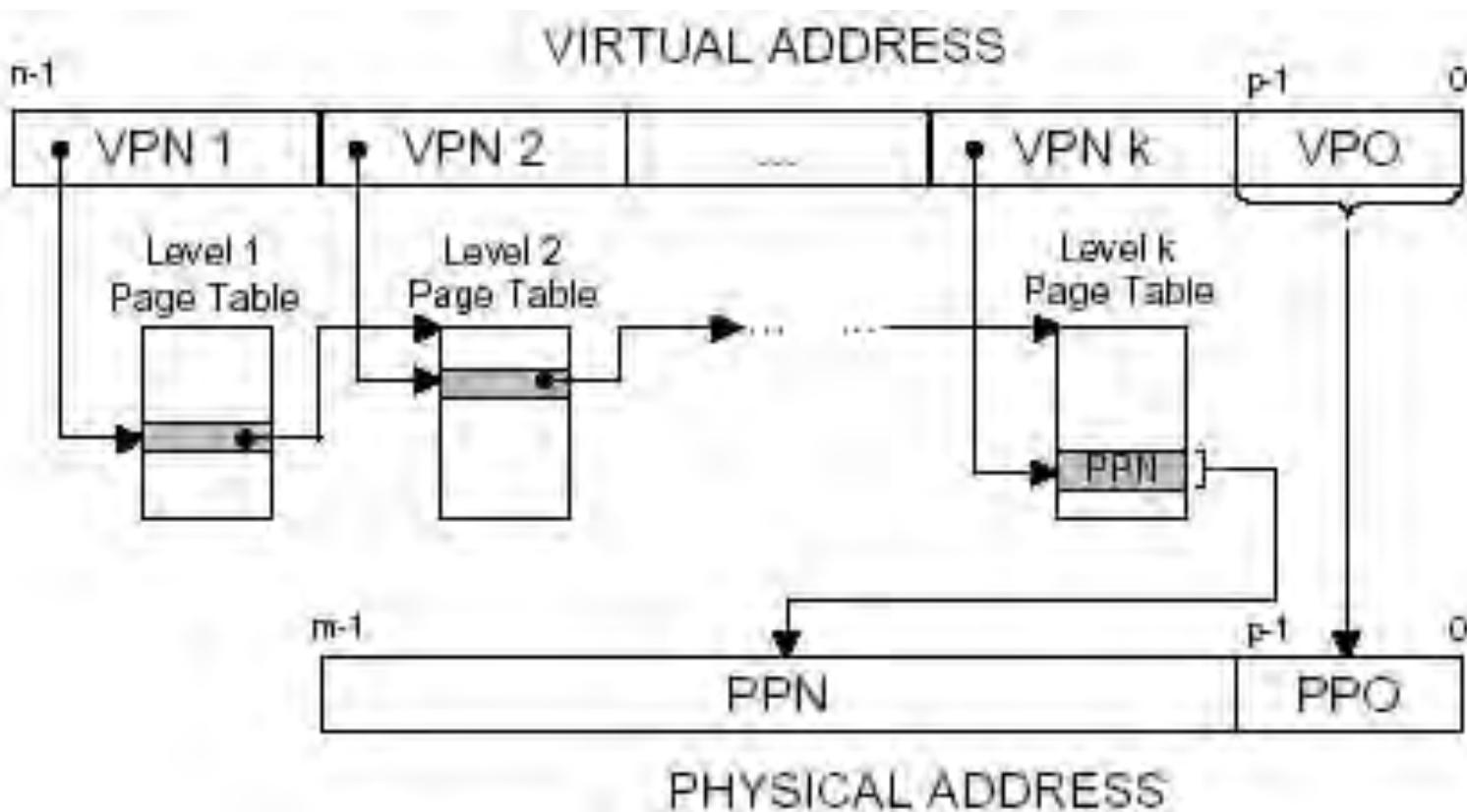
- Common solution
  - multi-level page tables
  - e.g., 2-level page table
    - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
    - Level 2 table: 1024 entries, each of which points to a page



# Multi-Level Page Tables



# Multi-Level Page Tables



# Summary

---

- Programmer's view of virtual memory
  - Each process has its own **private** linear address space
  - Cannot be corrupted by other processes
- System view of virtual memory
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions

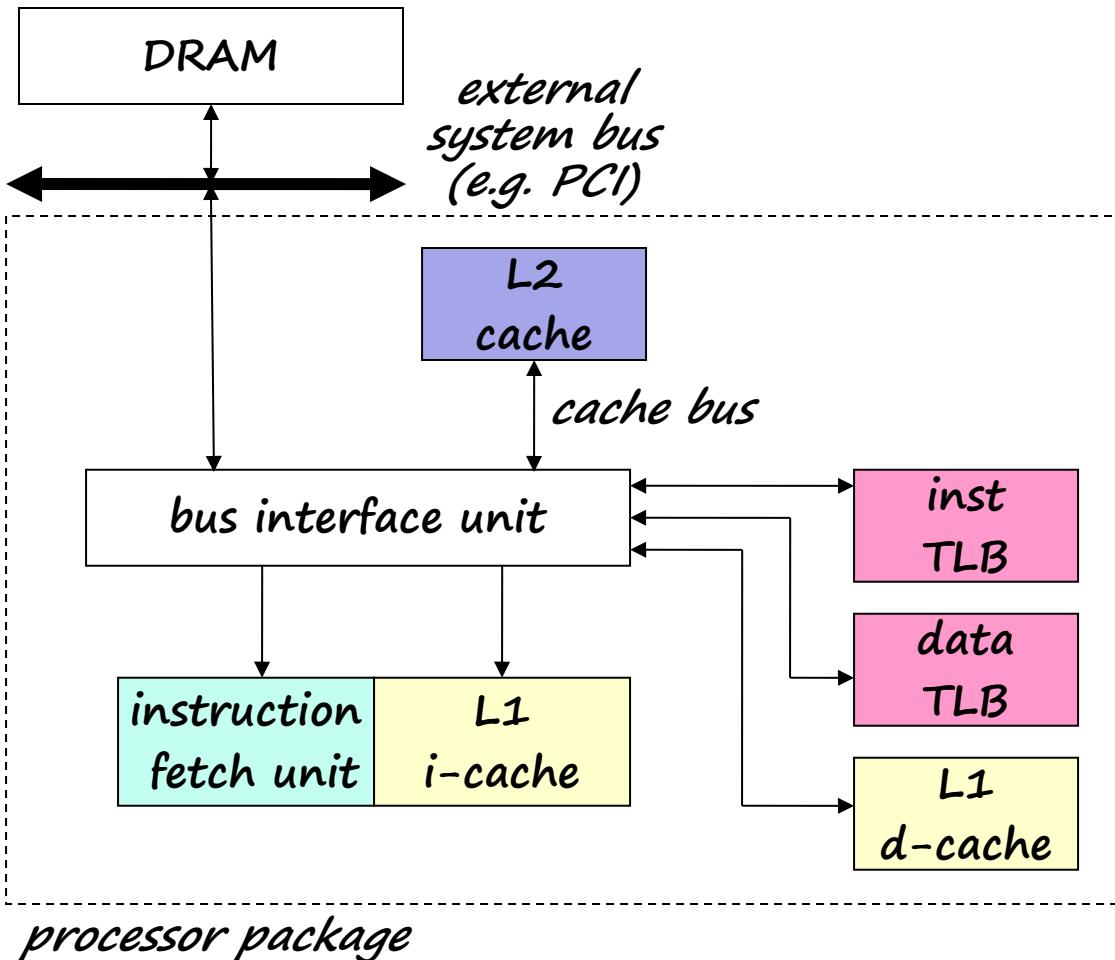
# Virtual Memory

# Outline

---

- Pentium/Linux Memory System
- Core i7
- Suggested reading: 9.7

# P6 Memory System



- 32 bit address space
- 4 KB page size
- L1, L2, and TLBs
  - 4-way set associative
- inst TLB
  - 32 entries
  - 8 sets
- data TLB
  - 64 entries
  - 16 sets
- L1 i-cache and d-cache
  - 16 KB
  - 32 B line size
  - 128 sets
- L2 cache
  - unified
  - 128 KB -- 2 MB
  - 32 B line size

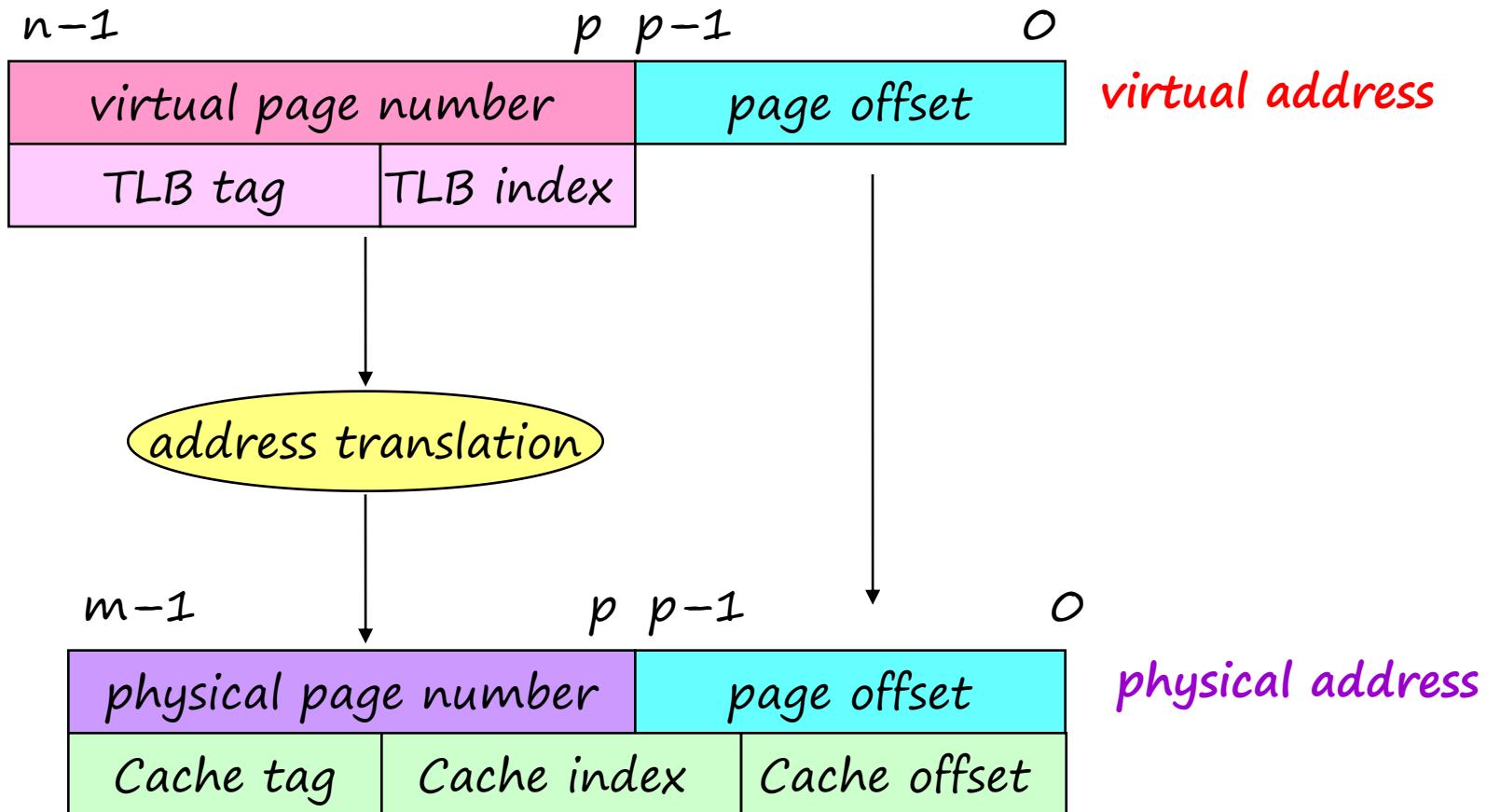
# Review of Address Translation

---

- Basic Parameters
  - $N = 2^n$  = Virtual address limit
  - $M = 2^m$  = Physical address limit
  - $P = 2^p$  = page size (bytes).
- Components of the virtual address (VA)
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)
  - **PPN**: Physical page number

# Review of Address Translation

---

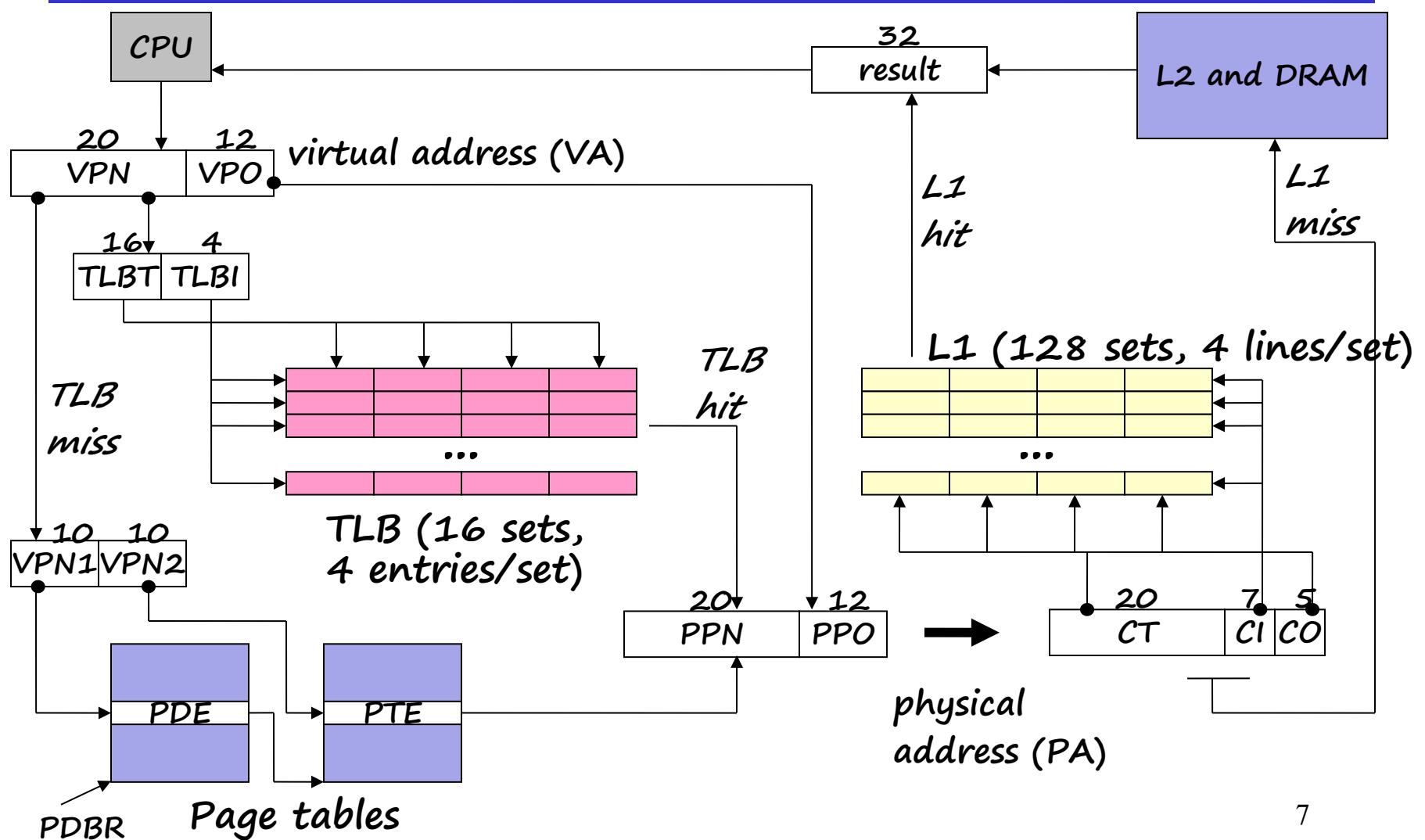


# Review of Address Translation

---

- Components of the virtual address (VA)
  - VPO: Virtual page offset
  - VPN: Virtual page number
  - TLBI: TLB index
  - TLBT: TLB tag
- Components of the physical address (PA)
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number
  - CO: Byte offset within cache line
  - CI: Cache index
  - CT: Cache tag

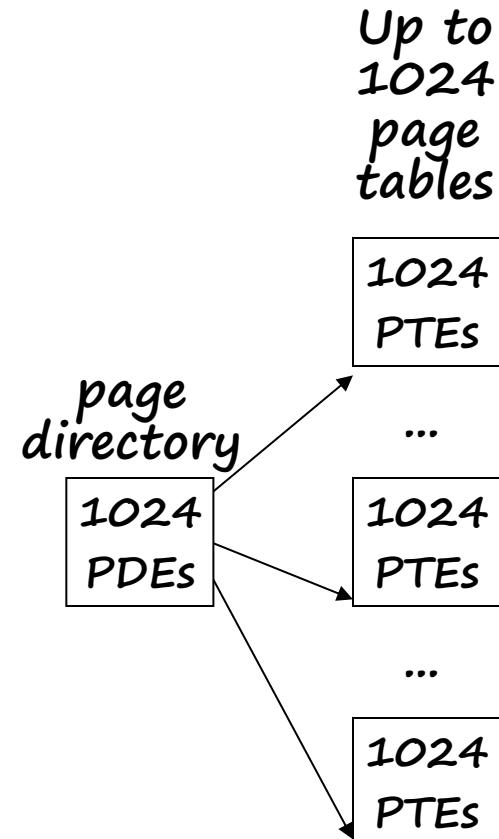
# P6 Address Translation



# P6 Page Table

---

- Page directory
  - 1024 4-byte page directory entries (PDEs) that point to page tables
  - one page directory per process.
  - page directory must be in memory when its process is running
  - always pointed to by PDBR
- Page tables:
  - 1024 4-byte page table entries (PTEs) that point to pages.
  - page tables can be paged in and out.



# P6 page directory entry (PDE)

31	1211	9	8	7	6	5	4	3	2	1	0
Page table physical base addr		Avail	G	PS		A	CD	WT	U/SR	WP=1	

Page table physical base address: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)

Avail: available for system programmers

G: global page (don't evict from TLB on task switch)

PS: page size 4K (0) or 4M (1)

A: accessed (set by MMU on reads and writes, cleared by software)

CD: cache disabled (1) or enabled (0)

WT: write-through or write-back cache policy for this page table

U/S: user or supervisor mode access

R/W: read-only or read-write access

31	P: page table is present in memory (1) or not (0)	1	0
	Available for OS (page table location in secondary storage)		P=0

# P6 page table entry (PTE)

31	1211	9	8	7	6	5	4	3	2	1	0
Page physical base address	Avail	G	0	D	A	CD	WT	U/SR	WP=1		

Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

Avail: available for system programmers

G: global page (don't evict from TLB on task switch)

D: dirty (set by MMU on writes)

A: accessed (set by MMU on reads and writes)

CD: cache disabled or enabled

WT: write-through or write-back cache policy for this page

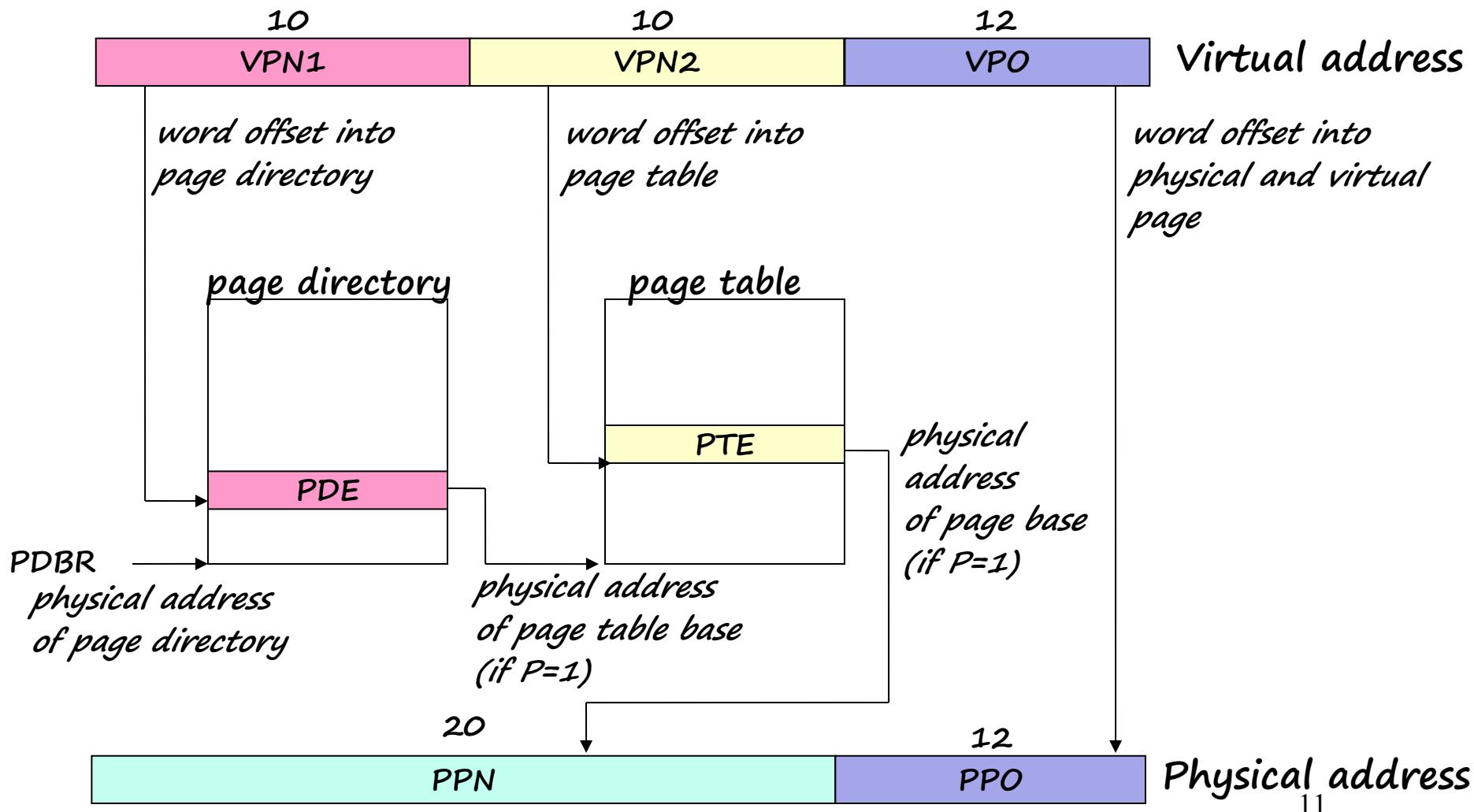
U/S: user/supervisor

R/W: read/write

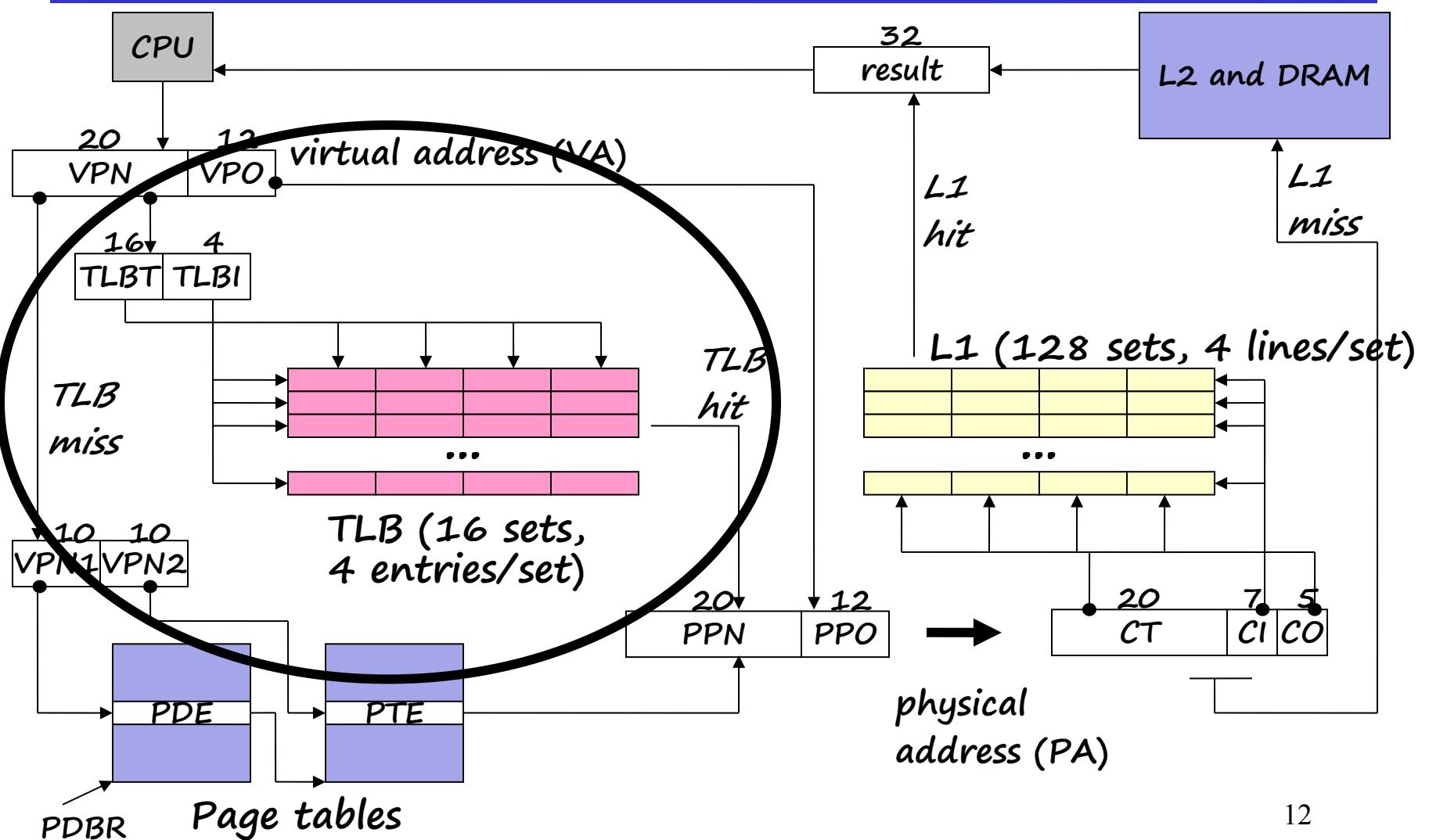
P: page is present in physical memory (1) or not (0)

31	1	0
Available for OS (page location in secondary storage)		P=0

# Page tables Translation



# P6 TLB Translation



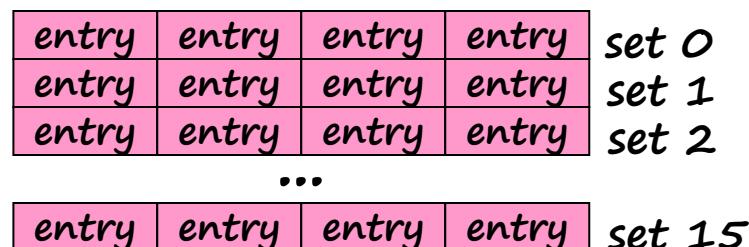
# P6 TLB

---

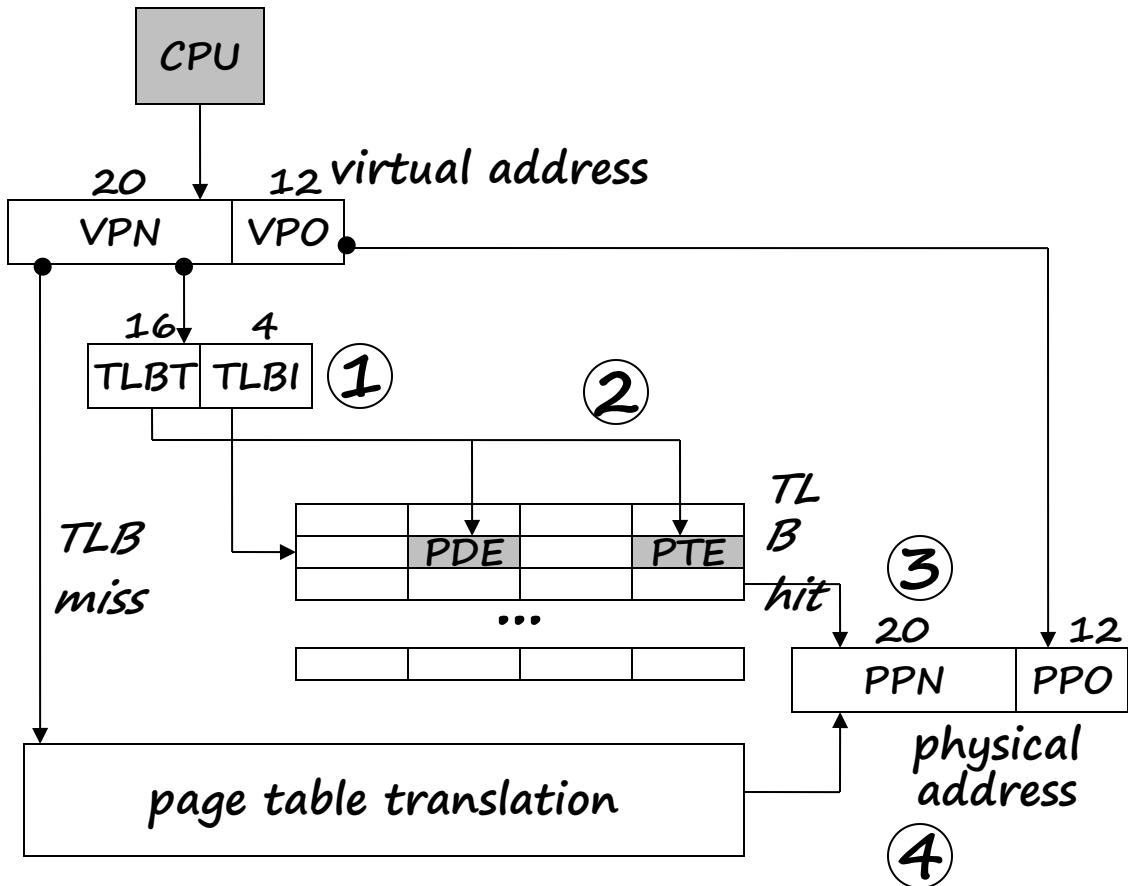
- TLB entry (not all documented, so this is speculative):

32	16	1	1
PDE/PTE	Tag	PD	V

- V: indicates a valid (1) or invalid (0) TLB entry
- PD: is this entry a PDE (1) or a PTE (0)?
- tag: disambiguates entries cached in the same set
- PDE/PTE: page directory or page table entry
- Structure of the data TLB:
  - 16 sets, 4 entries/set

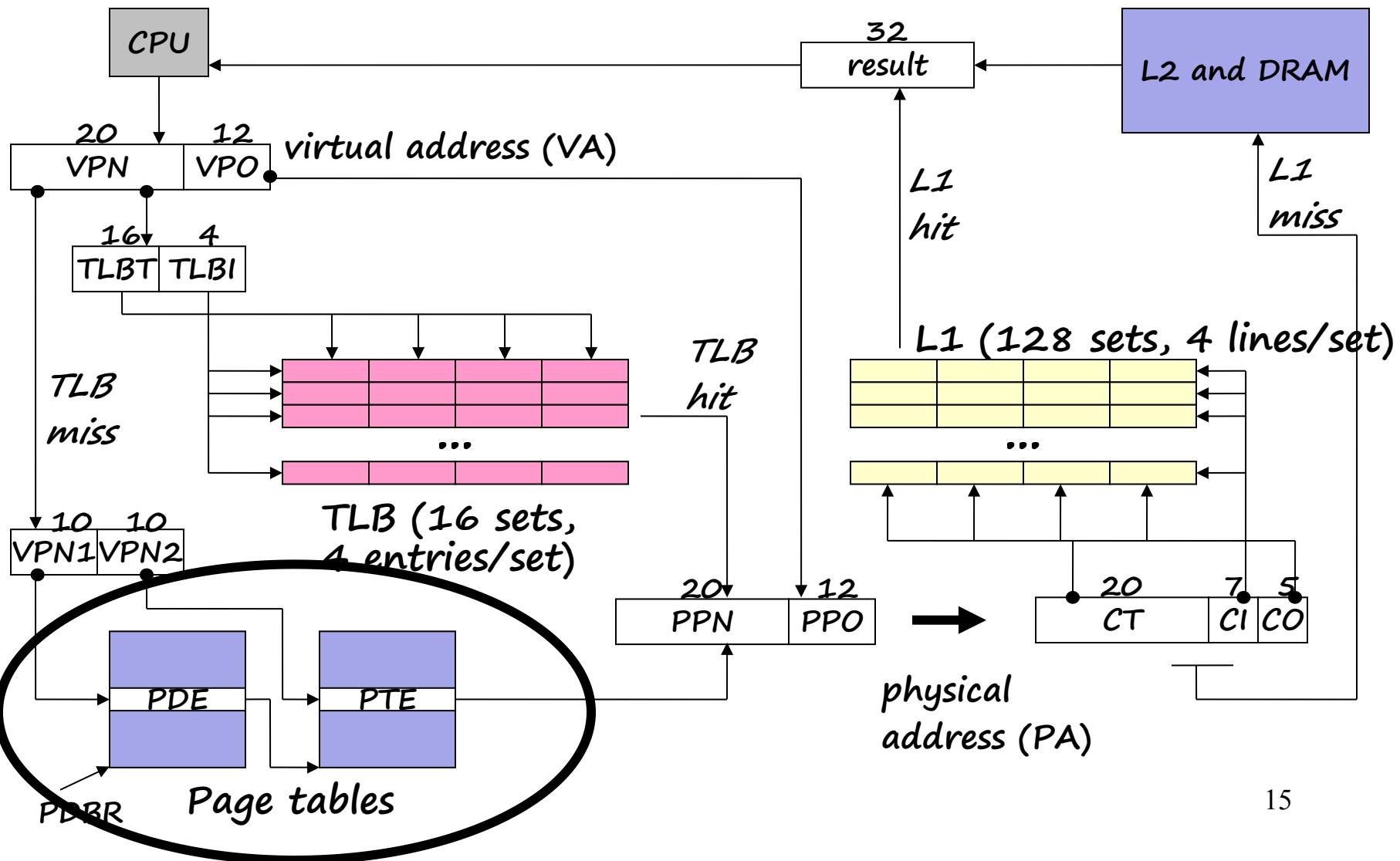


# Translating with the P6 TLB

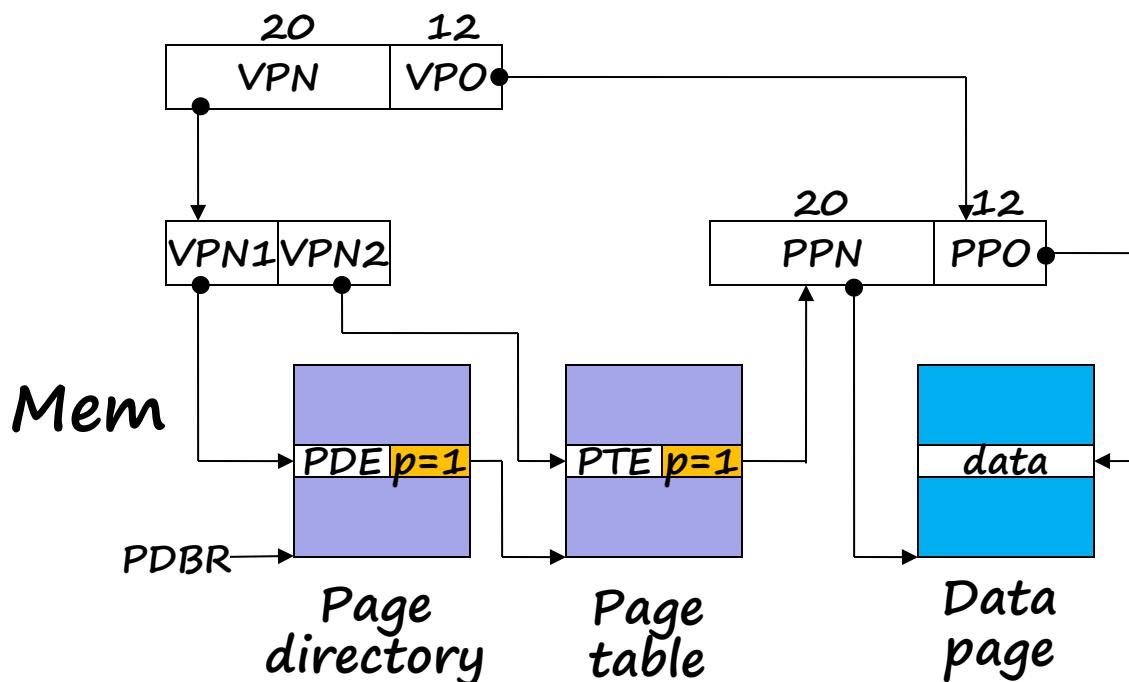


1. Partition VPN into TLBT and TLBI.
2. Is the PTE for VPN cached in set TLBI?
3. Yes: then build physical address.
4. No: then read PTE (and PDE if not cached) from memory and build physical address.

# P6 Page Table Translation



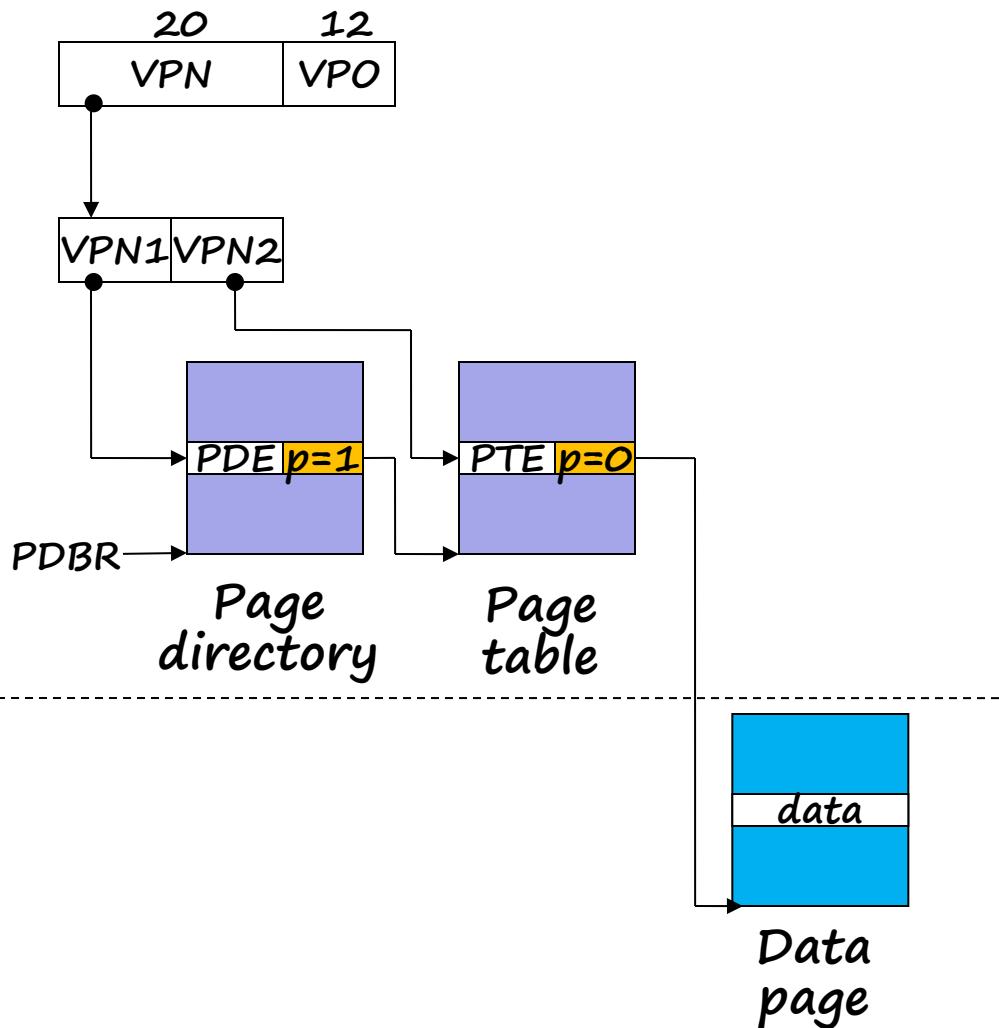
# Translating with the P6 page tables (case 1/1)



- Case 1/1: page table and page present.
- MMU Action:
  - MMU build physical address and fetch data word.
- OS action
  - none

# Translating with the P6 page tables (case 1/0)

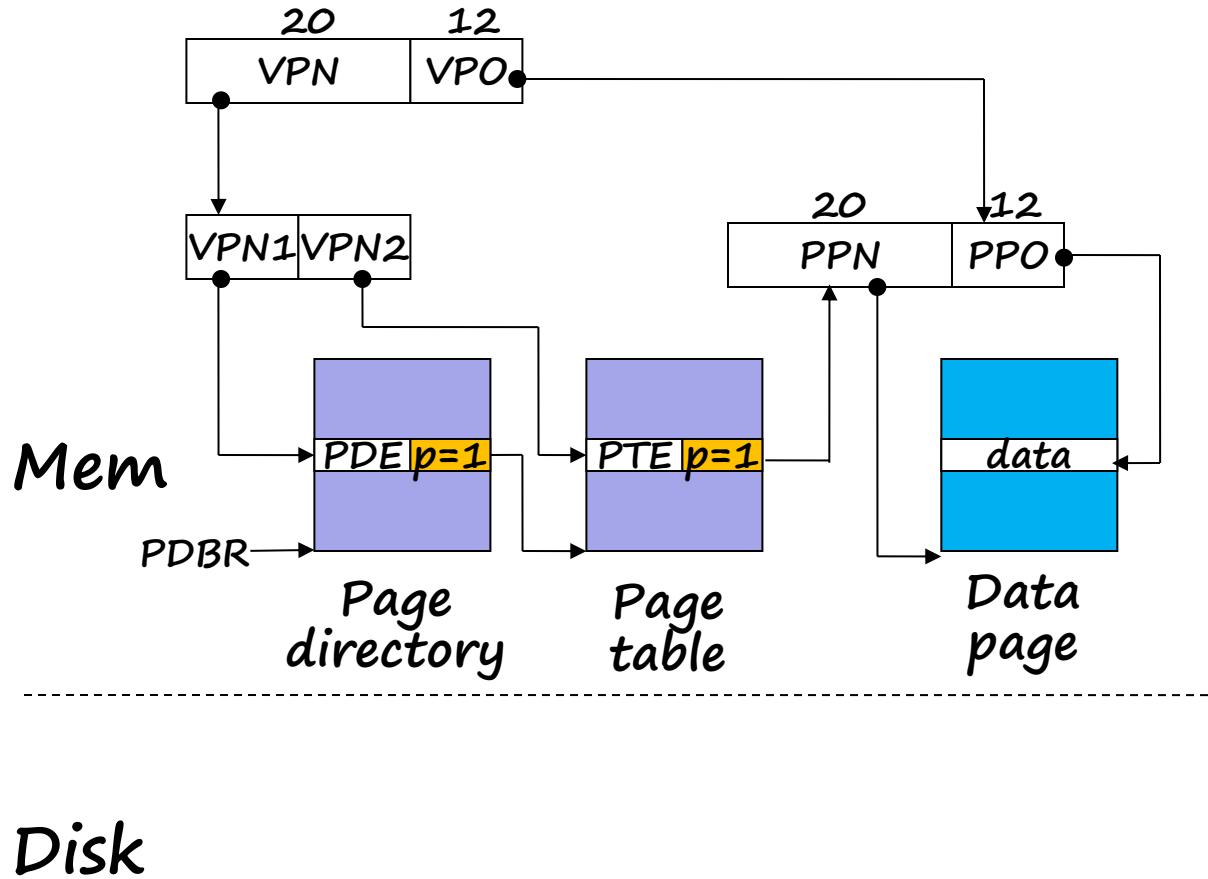
Mem



Disk

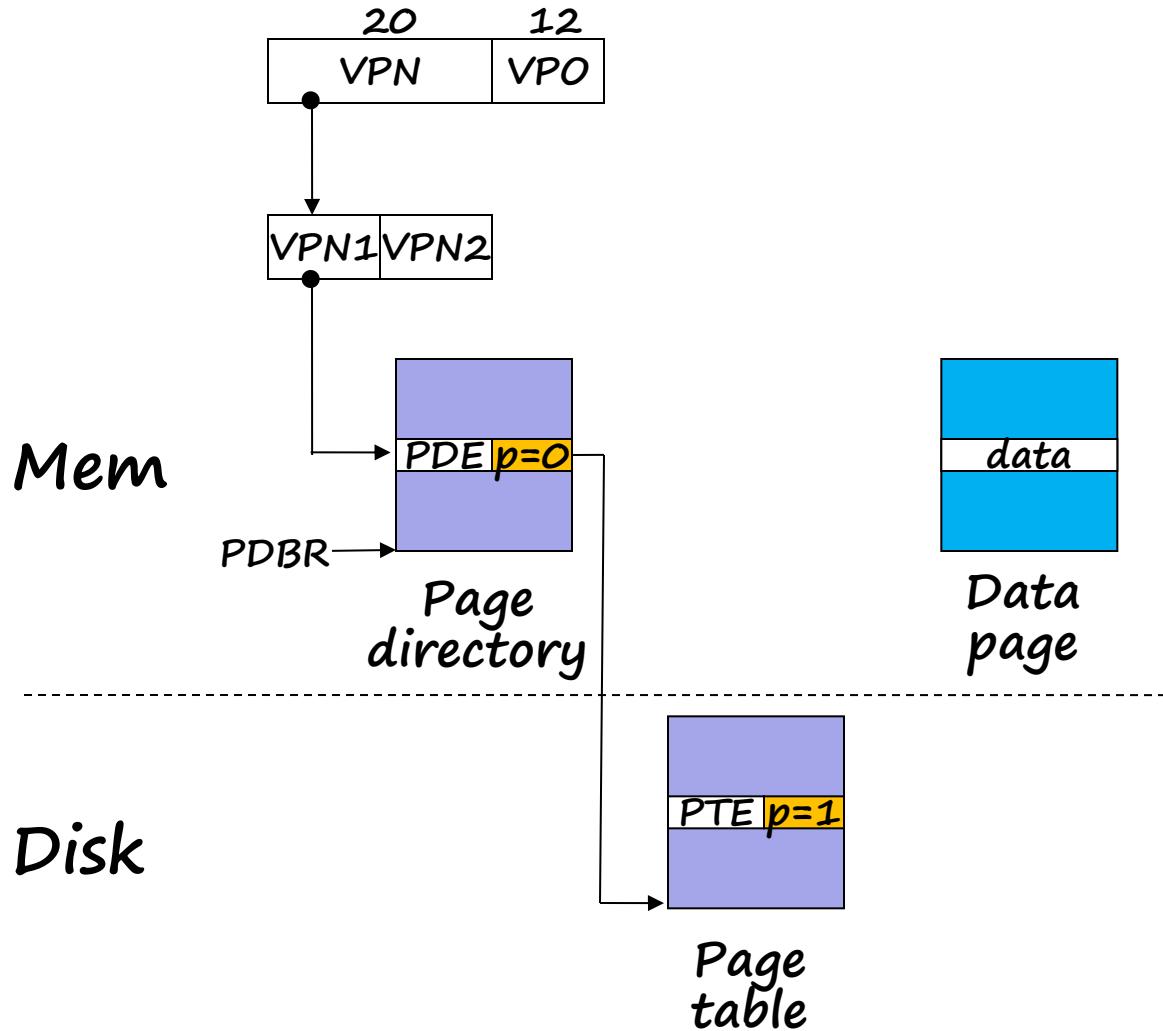
- Case 1/0: page table present but page missing.
- MMU Action:
  - page fault exception
  - handler receives the following args:
    - VA that caused fault
    - fault caused by non-present page or page-level protection violation
    - read/write
    - user/supervisor

# Translating with the P6 page tables (case 1/0, cont)



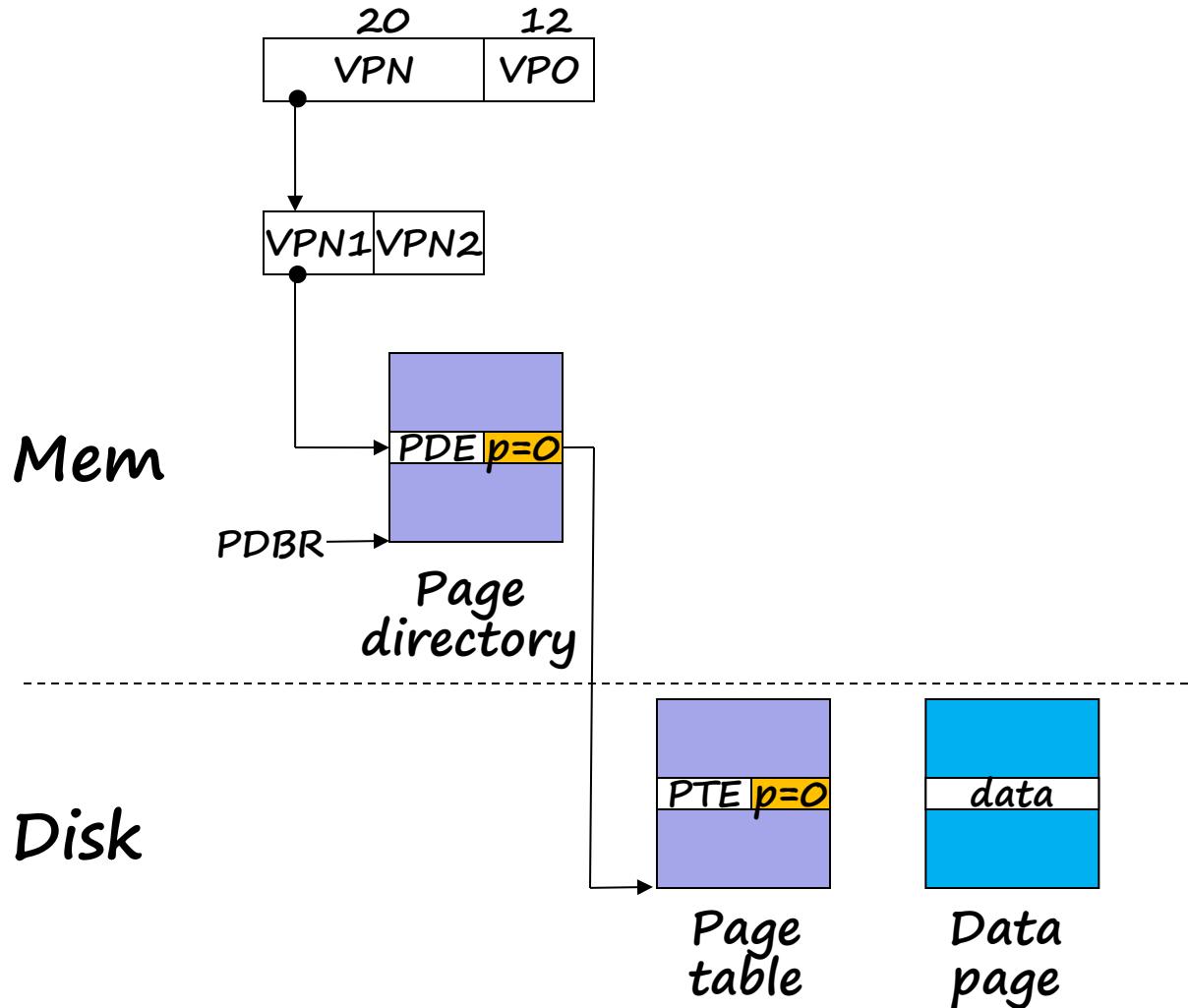
- OS Action:
  - Check for a legal virtual address.
  - Read PTE through PDE.
  - Find free physical page (swapping out current page if necessary)
  - Read virtual page from disk and copy to virtual page
  - Restart faulting instruction by returning from exception handler.

# Translating with the P6 page tables (case 0/1)



- Case 0/1: page table missing but page present.
- Introduces consistency issue.
  - potentially every page out requires update of disk page table.
- Linux **disallows** this
  - if a page table is swapped out, then swap out its data pages too.

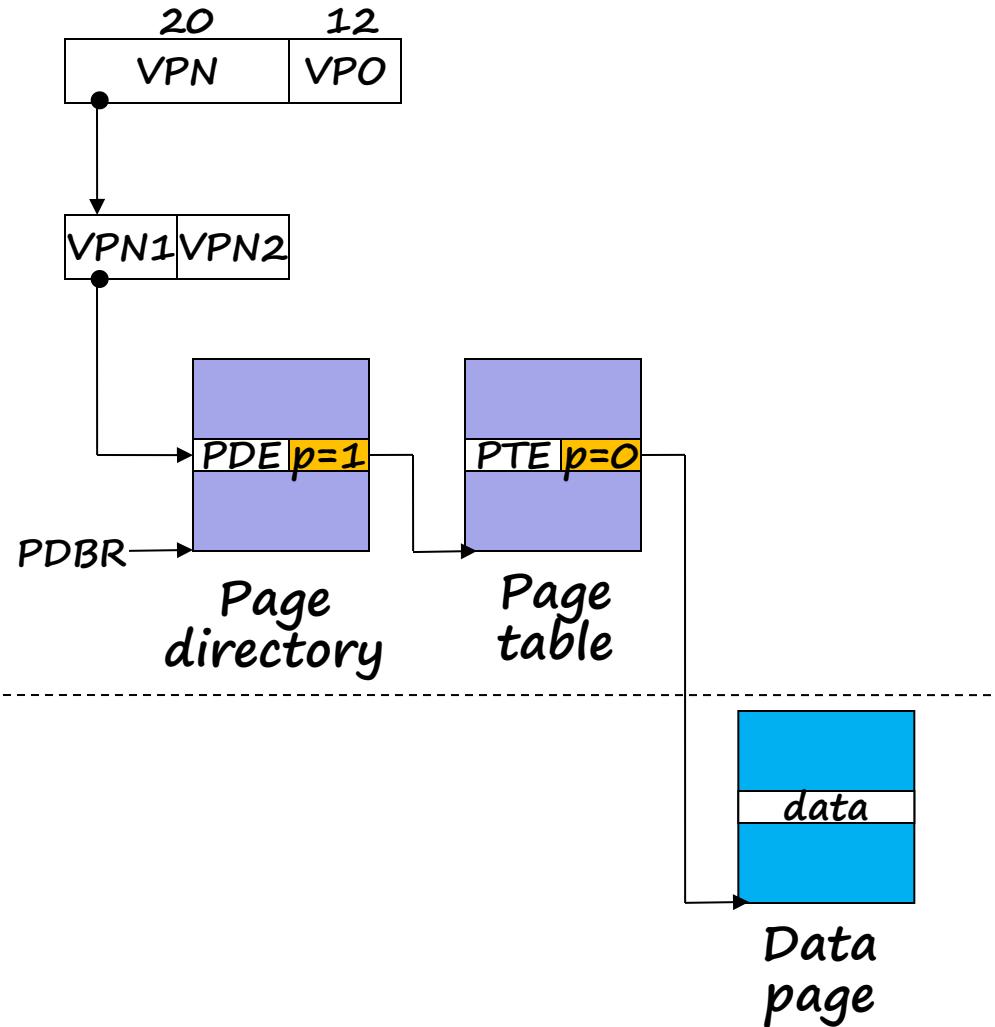
# Translating with the P6 page tables (case 0/0)



- Case 0/0: page table and page missing.
- MMU Action:
  - page fault exception

# Translating with the P6 page tables (case 0/0, cont)

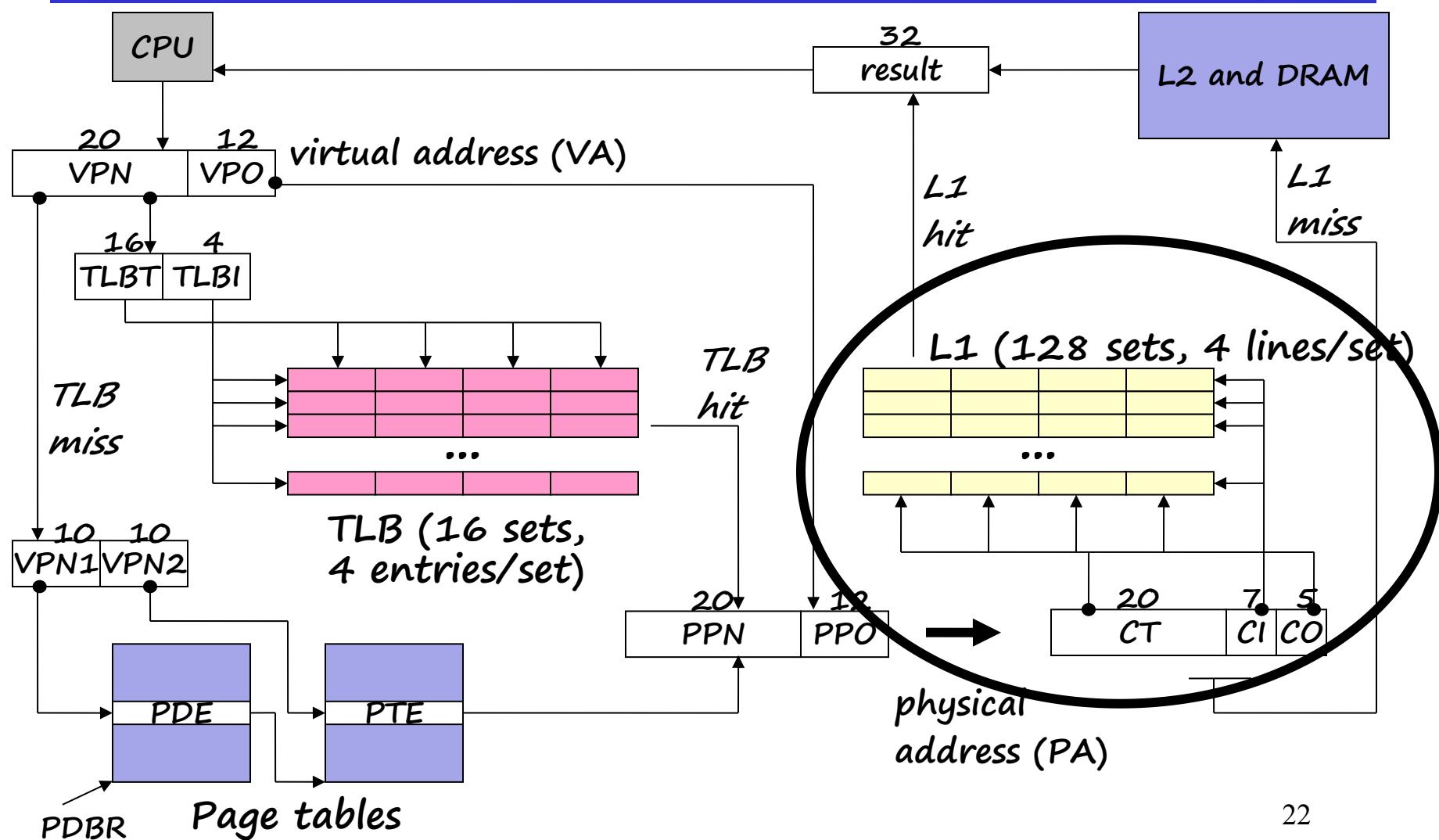
Mem



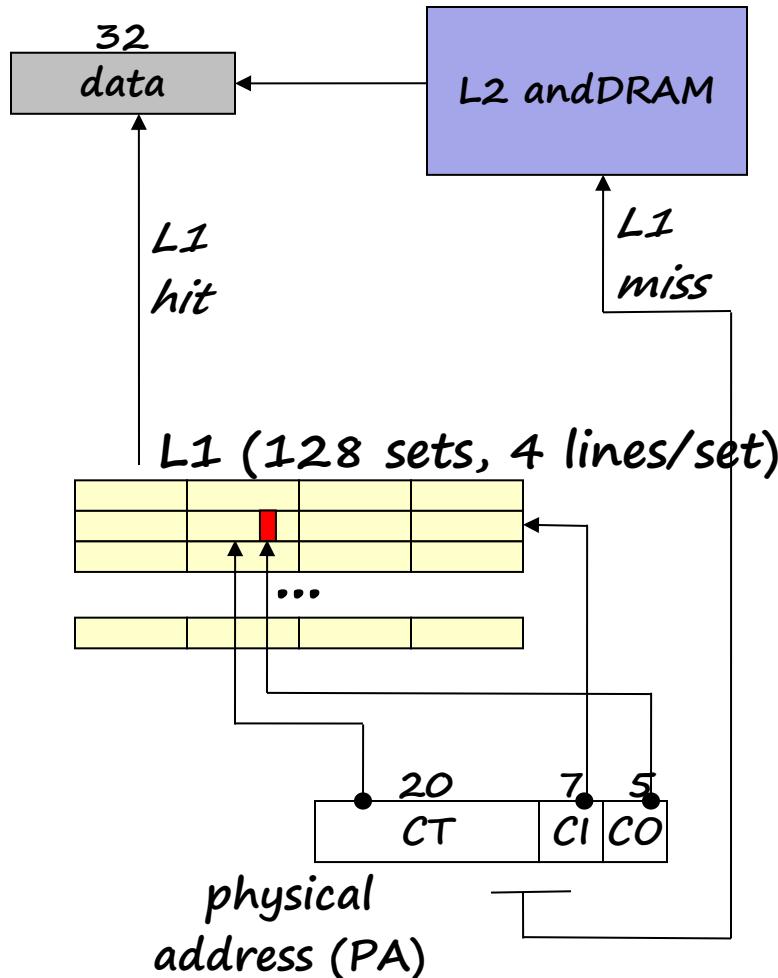
- OS action:
  - swap in page table.
  - restart faulting instruction by returning from handler.
- Like case 1/0 from here on.

Disk

# P6 L1 Cache Access



# L1 cache access



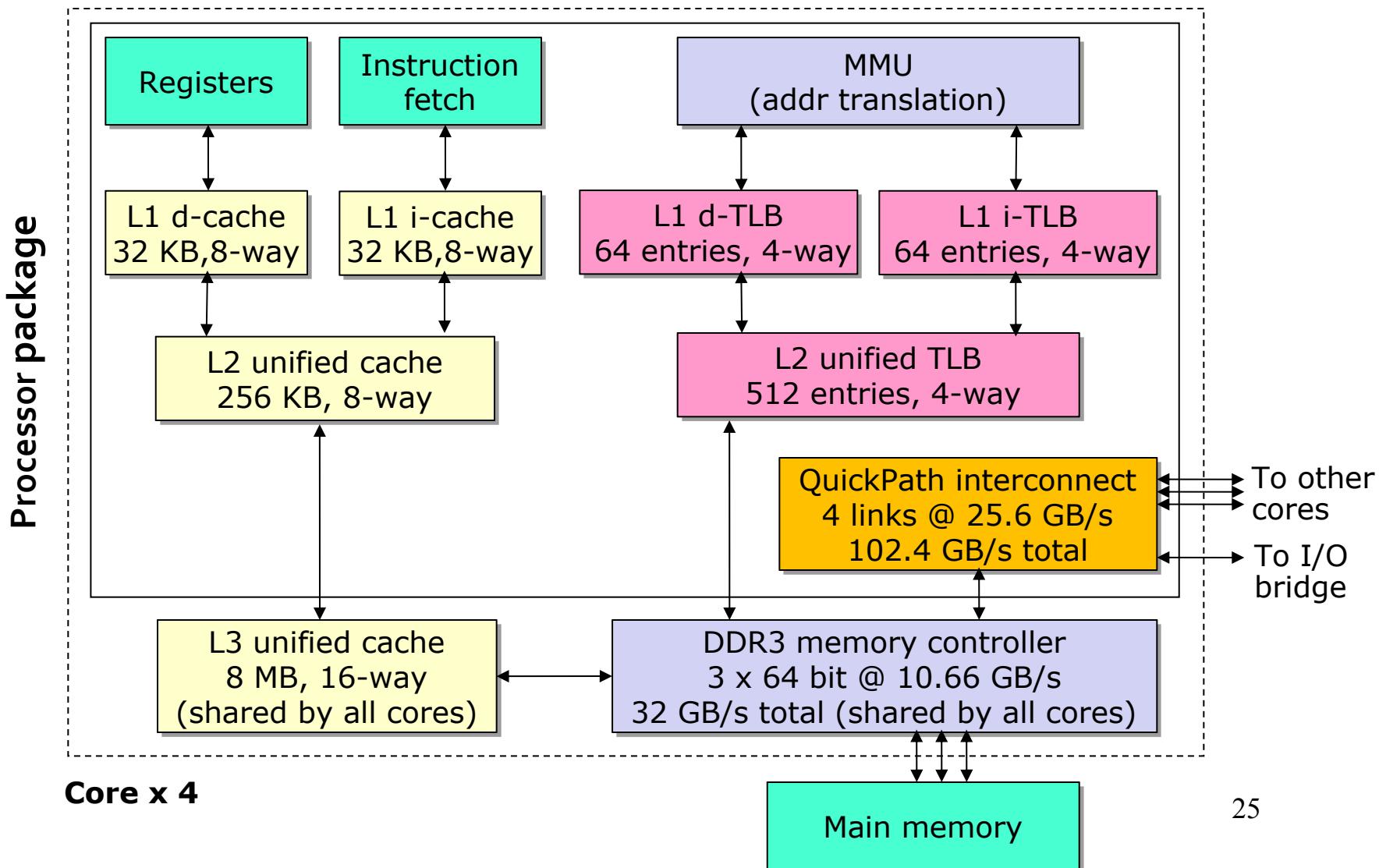
- Partition physical address into CO, CI, and CT.
- Use CT to determine if line containing word at address PA is cached in set CI.
- If no: check L2.
- If yes: extract word at byte offset CO and return to processor.

# Core i7 Summary

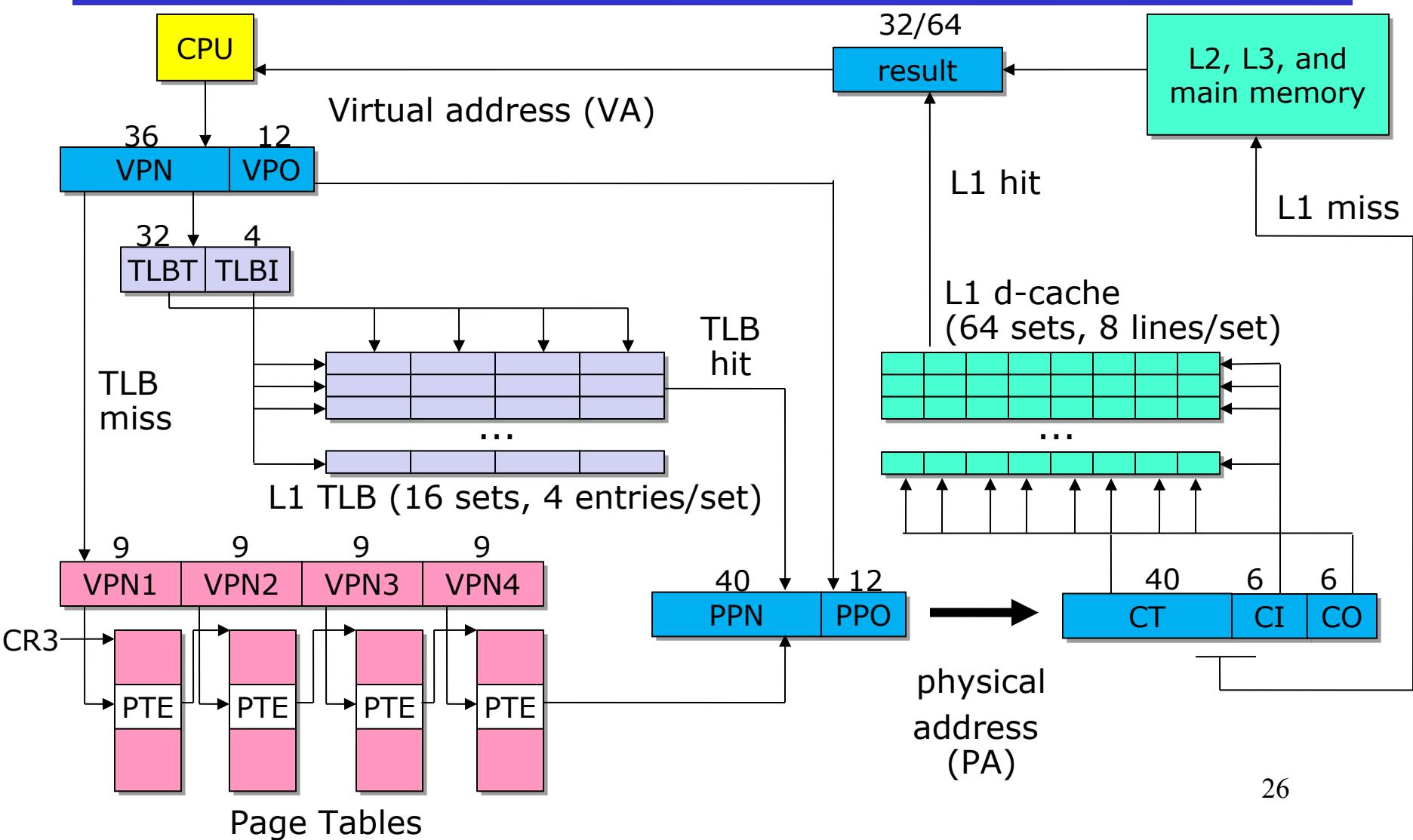
---

- Core i7
  - The 64-bit Nehalem microarchitecture
  - Current support 48-bit (256 TB) virtual address space and 52-bit (4 PB) physical address space
  - Compatibility mode support 32-bit (4 GB) virtual and physical address spaces

# Core i7 Memory System



# Core i7 Address Translation



# Level 1, Level 2 and Level 3 Page Table Entry

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base addr				Unused	G	PS		A	CD	WT	U/S	R/W	P=1
Available for OS (page table location in secondary storage)															P=0

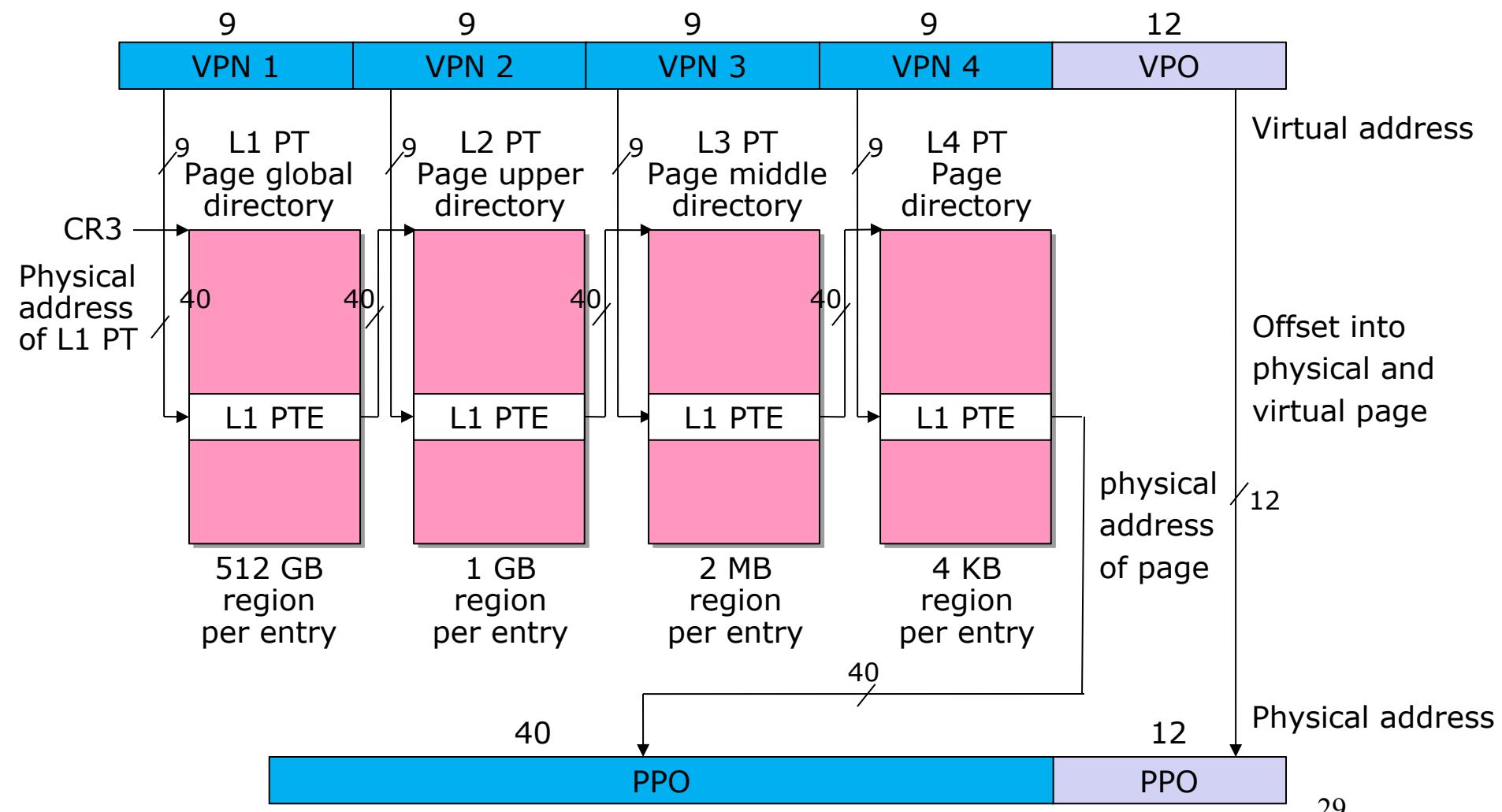
- XD                  Disable or enable instruction fetches
- Base addr        40 most significant bits of base address of child page table (forces page tables to be 4KB aligned)
- G                  global page (don't evict from TLB on task switch)
- PS                Page size either 4K(0) or 2M(1) (defined for Level 1 PTEs only)
- A                Reference bit (set by MMU on reads and writes, cleared by software)
- CD               Cache disabled(1) or enabled(0) for child page table
- WT               Write-through or write-back cache policy
- U/S              User or supervisor(kernel) mode access permission
- R/W             Read-only or read-write access permission
- P                Child page table present in memory(1) or not(0)

# Level 4 Page Table Entry

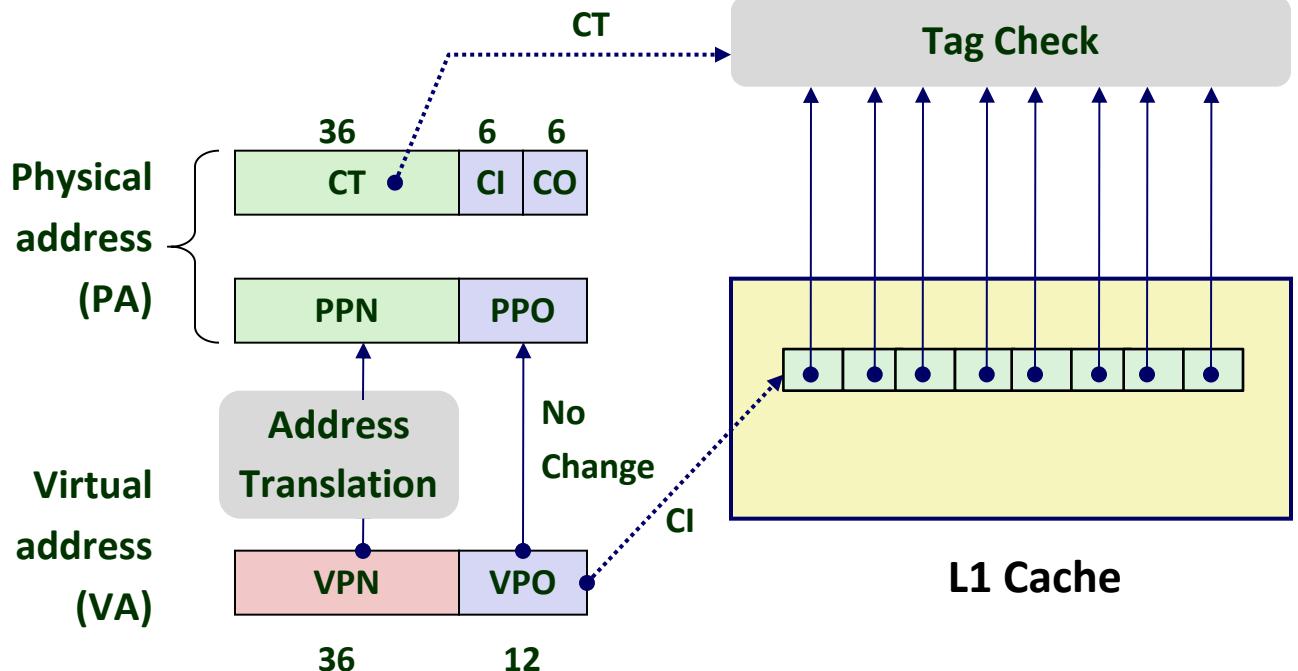
63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base addr				Unused	G		D	A	CD	WT	U/S	R/W	P=1
Available for OS (page table location in secondary storage)														P=0	

- XD                  Disable or enable instruction fetches
- Base addr        40 most significant bits of base address of child page table (forces page tables to be 4KB aligned)
- G                  global page (don't evict from TLB on task switch)
- D                  Dirty bit (Set by MMU on writes, cleared by software)
- A                  Reference bit (set by MMU on reads and writes, cleared by software)
- CD                 Cache disabled(1) or enabled(0) for child page table
- WT                 Write-through or write-back cache policy
- U/S                User or supervisor(kernel) mode access permission
- R/W                Read-only or read-write access permission
- P                  Child page table present in memory(1) or not(0)

# Page tables Translation



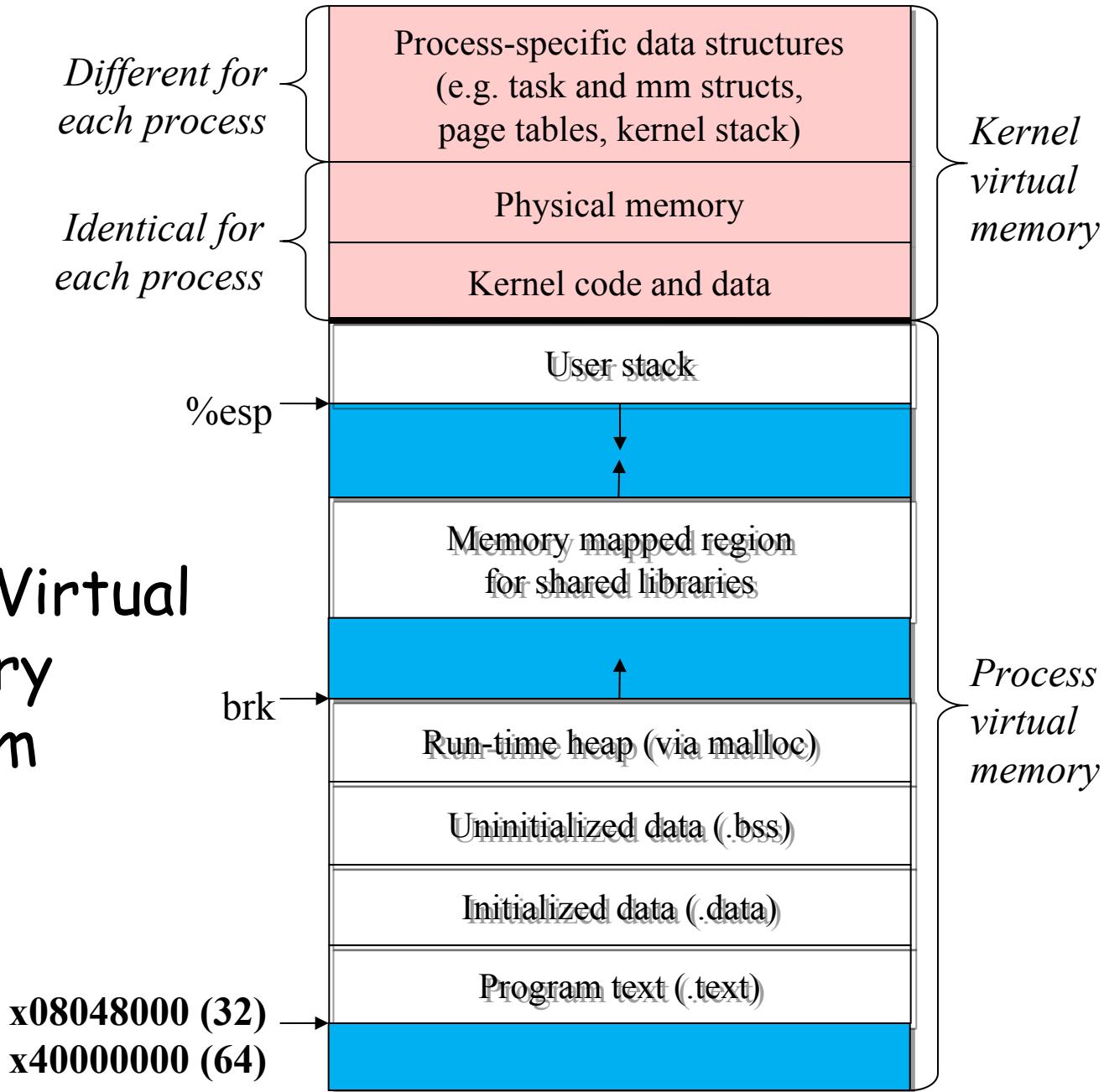
# Cute Trick for Speeding Up L1 Access



- **Observation**

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible

# Linux Virtual Memory System



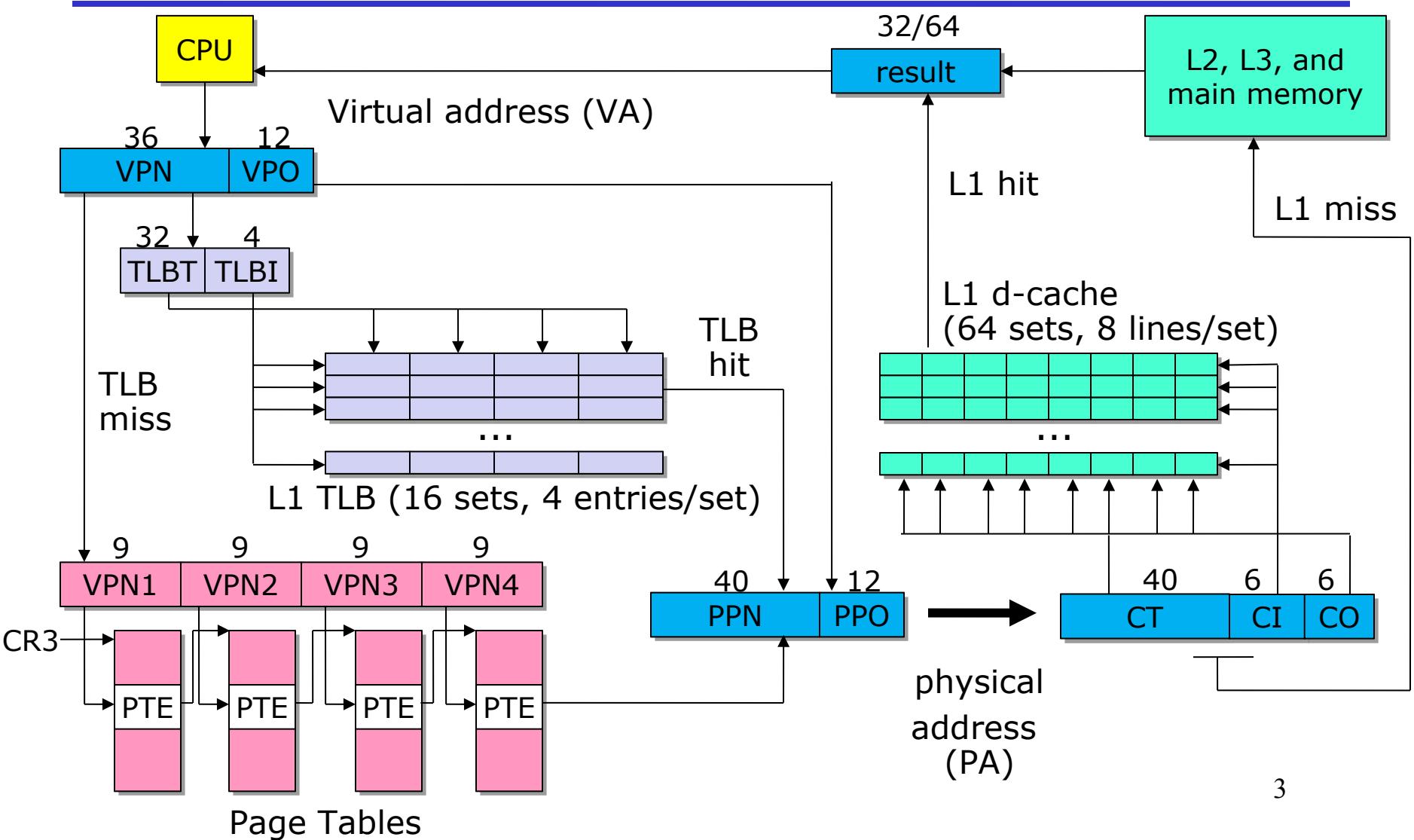
# Virtual Memory

# Outline

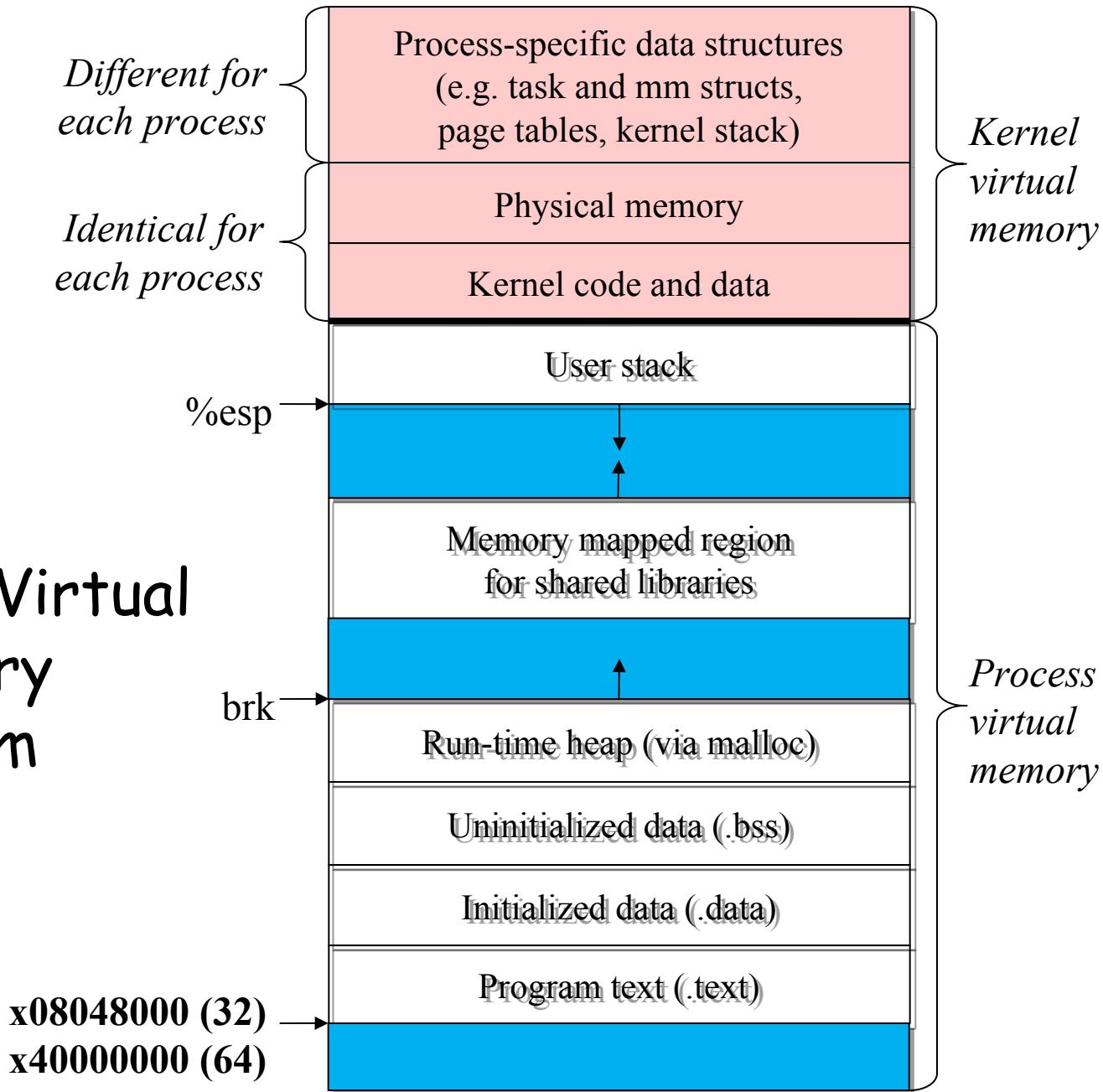
---

- Memory Mapping
- Suggested reading: 10.7, 10.8

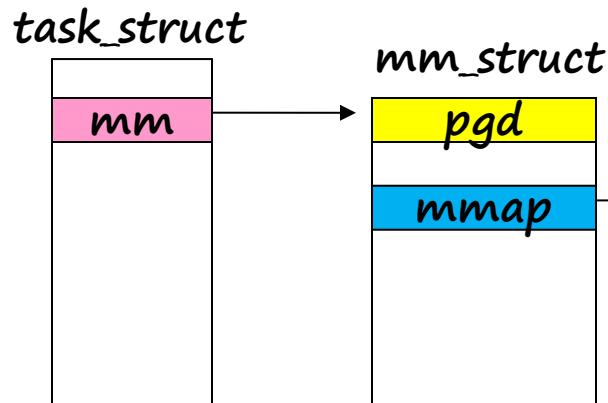
# Address Translation



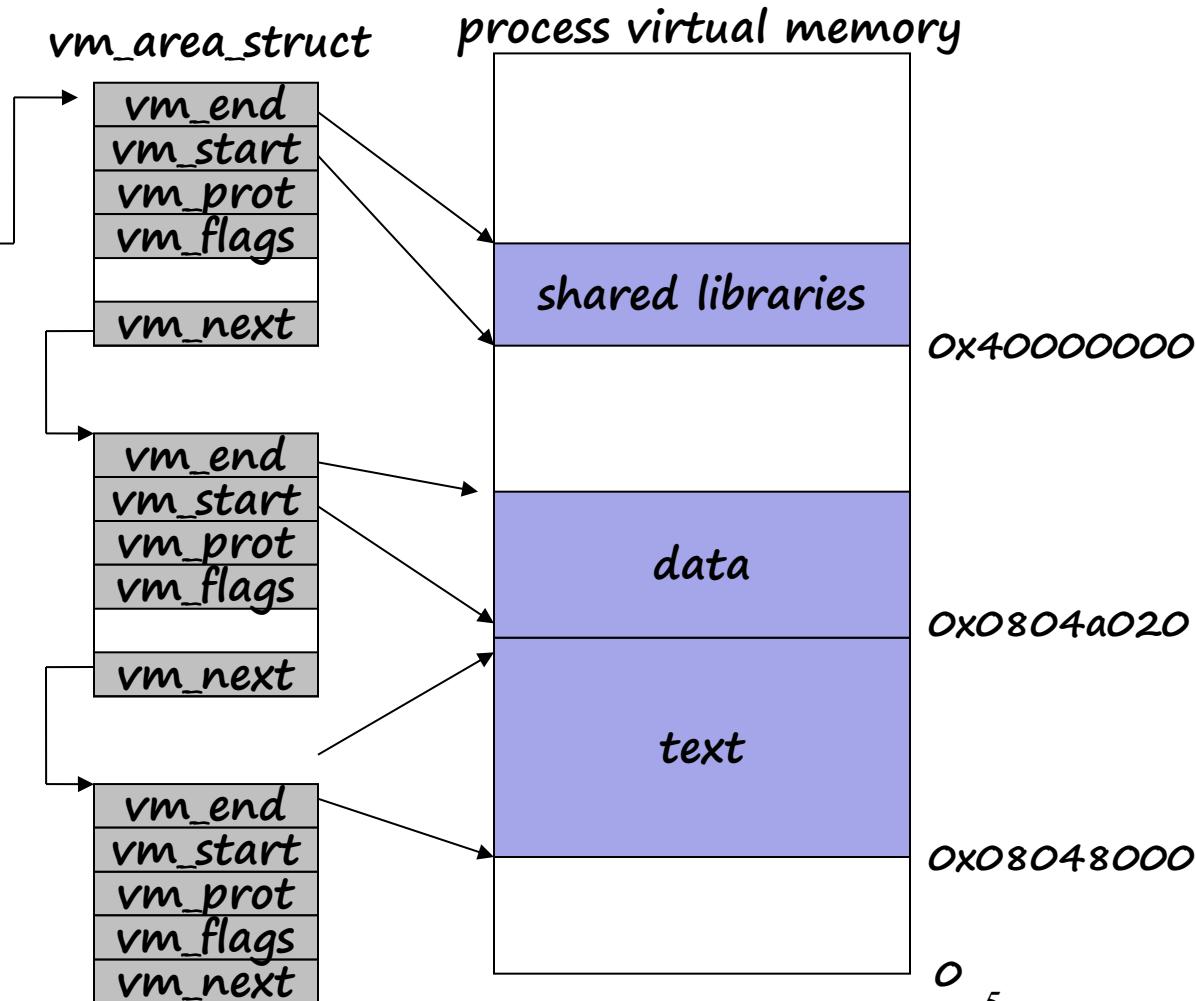
# Linux Virtual Memory System



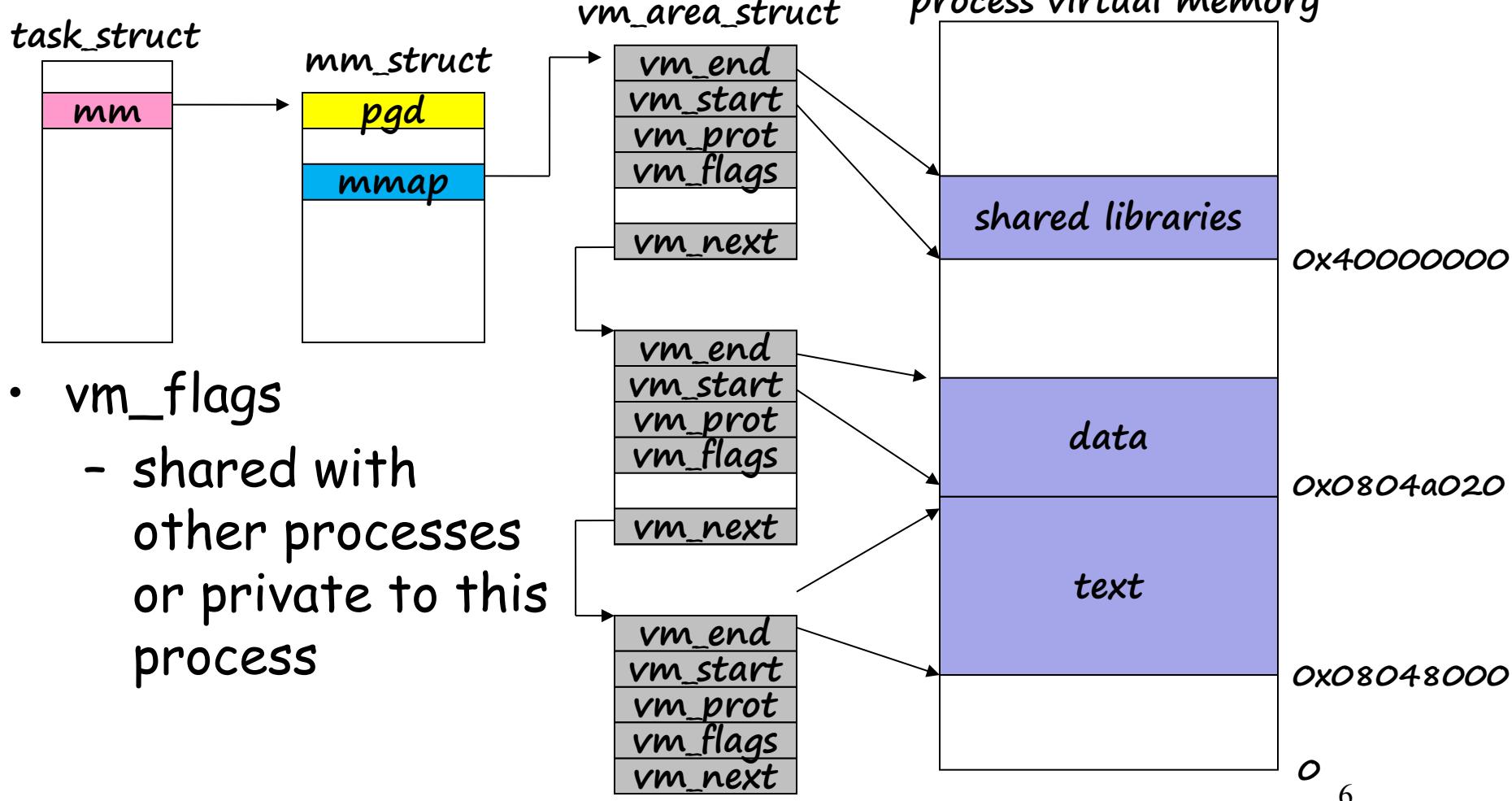
# Linux organizes VM as a collection of “areas”



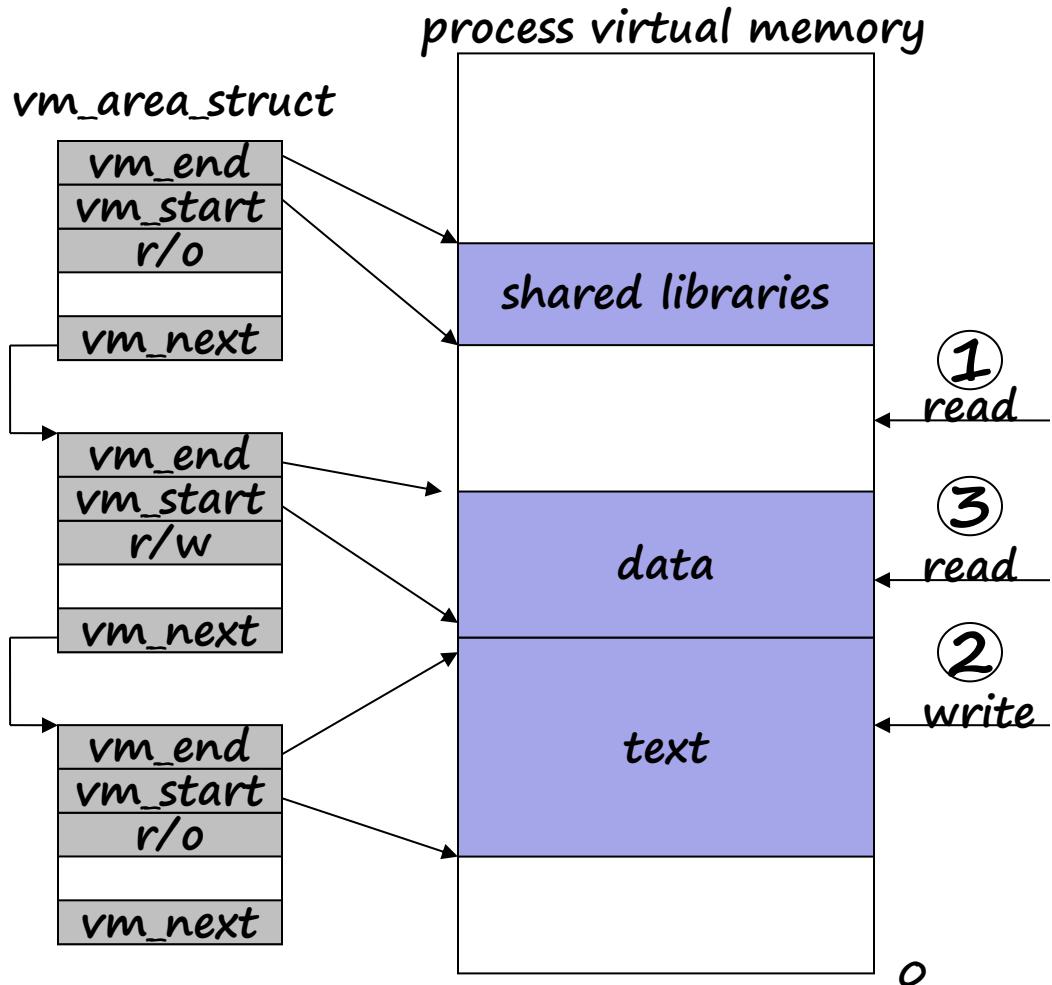
- **pgd:**
  - page directory address
- **vm\_prot:**
  - read/write permissions for this area



# Linux organizes VM as a collection of “areas”

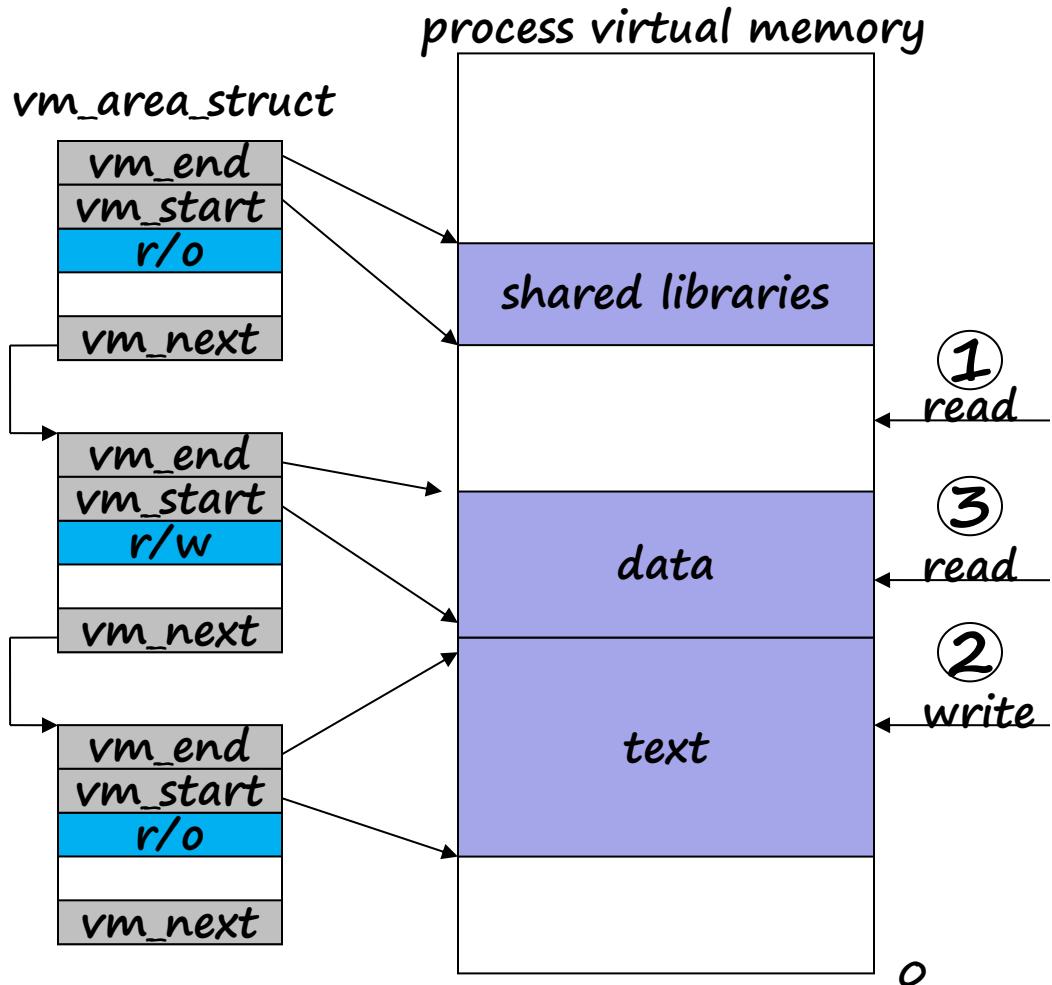


# Linux page fault handling



- Is the VA legal?
  - i.e. is it in an area defined by a **vm\_area\_struct**?
  - if not then signal segmentation violation (e.g. (1))

# Linux page fault handling



- Is the operation legal?
  - i.e., can the process read/write this area?
  - if not then signal protection violation (e.g., (2))
- If OK, handle fault
  - e.g., (3)

# Memory mapping

---

- Creation of new VM area done via “**memory mapping**”
  - create new **vm\_area\_struct** and **page tables** for area
- Area can be backed by (i.e., get its initial values from):
  - regular file on disk (e.g., an executable object file)
    - initial page bytes come from a section of a file

# Memory mapping

---

- Area can be backed by (i.e., get its initial values from)
  - anonymous file (e.g. nothing)
    - First fault will allocate a physical page full of 0's (demand-zero page)
    - Once the page is written to (dirtied), it is like any other page

# Memory mapping

---

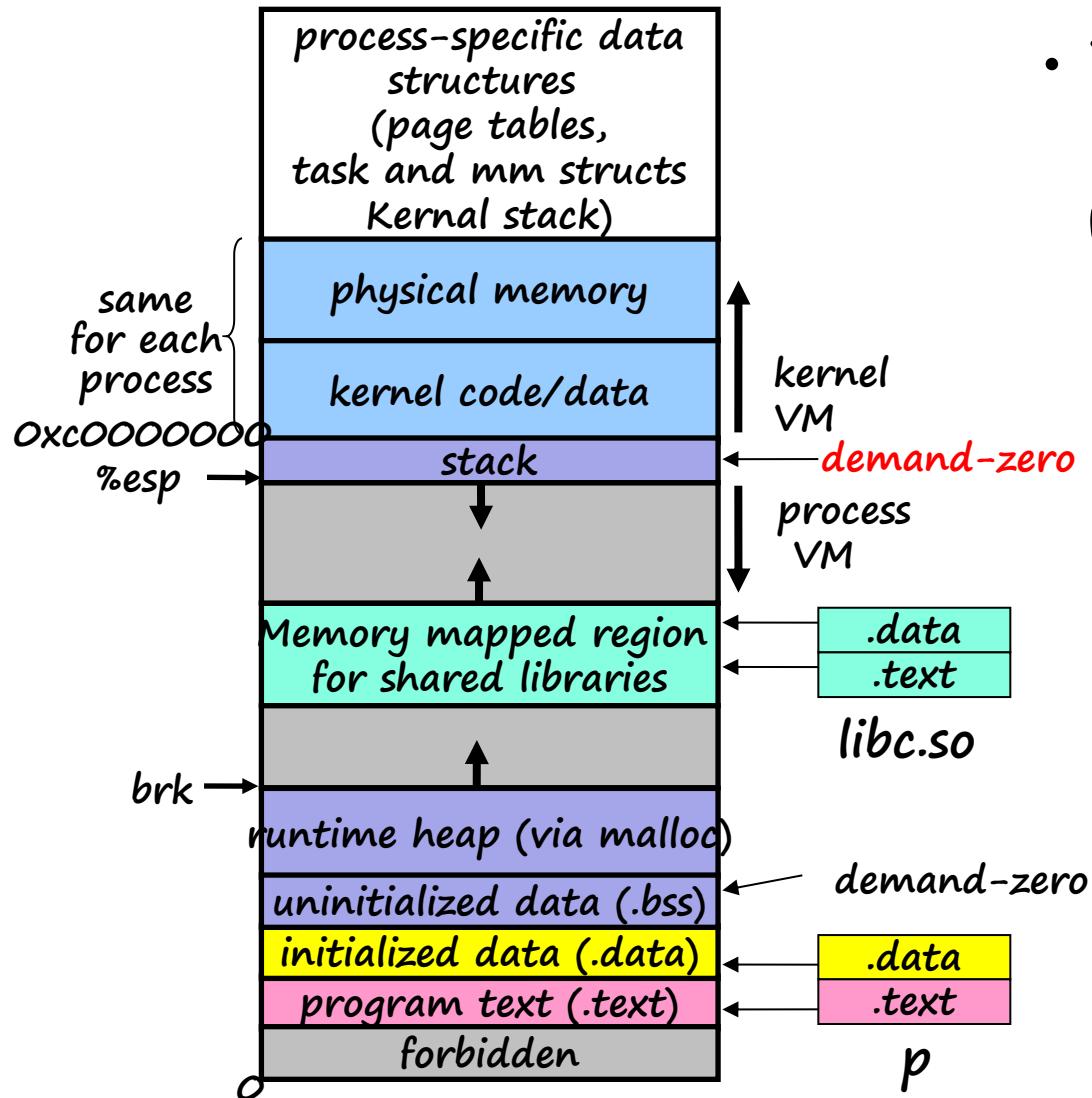
- Dirty pages are copied back and forth between memory and a special **swap file**.

# Demand Paging

---

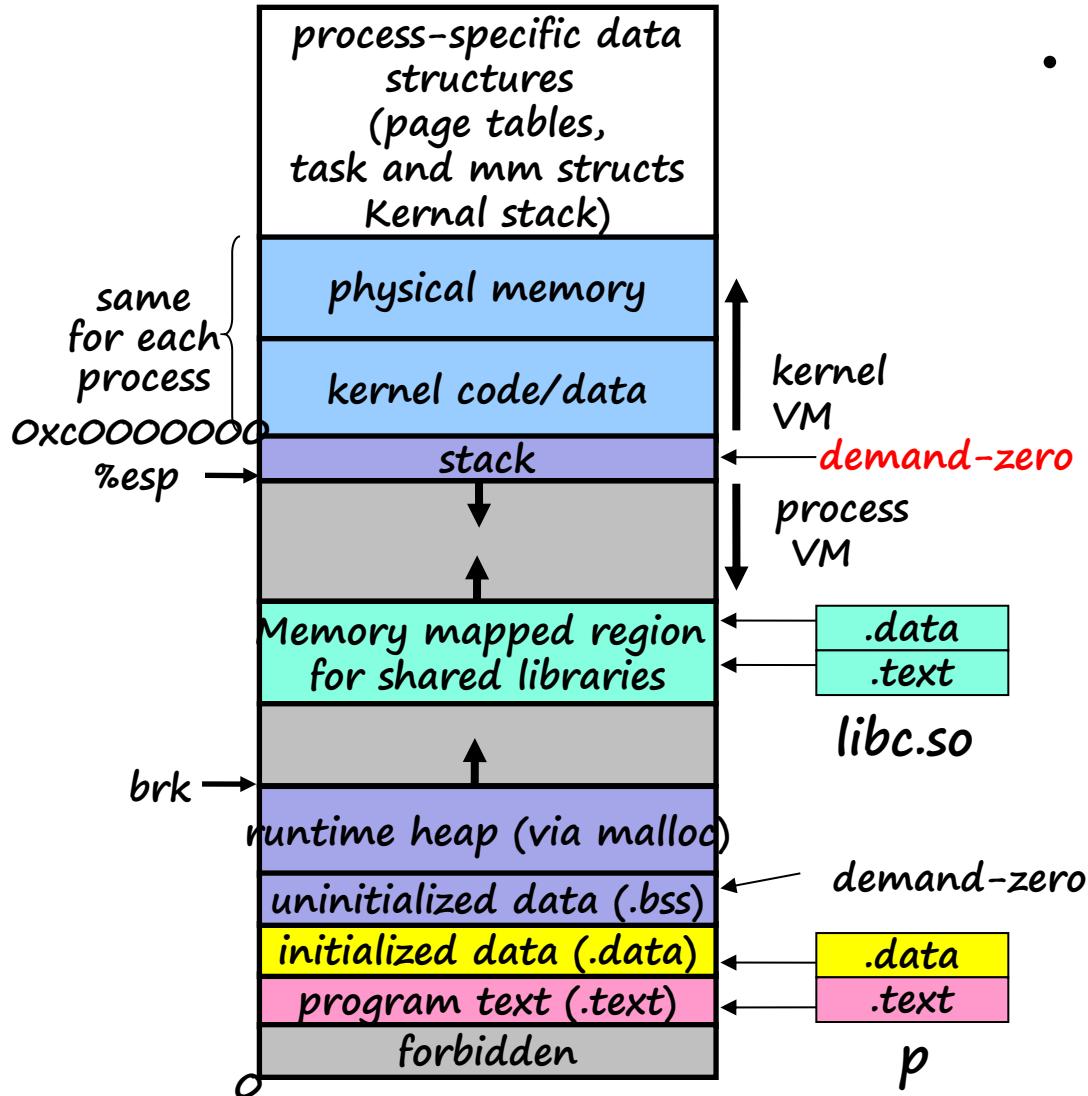
- Key point: no virtual pages are copied into physical memory until they are referenced!
  - known as “**demand paging**”
  - crucial for time and space efficiency

# Exec() revisited



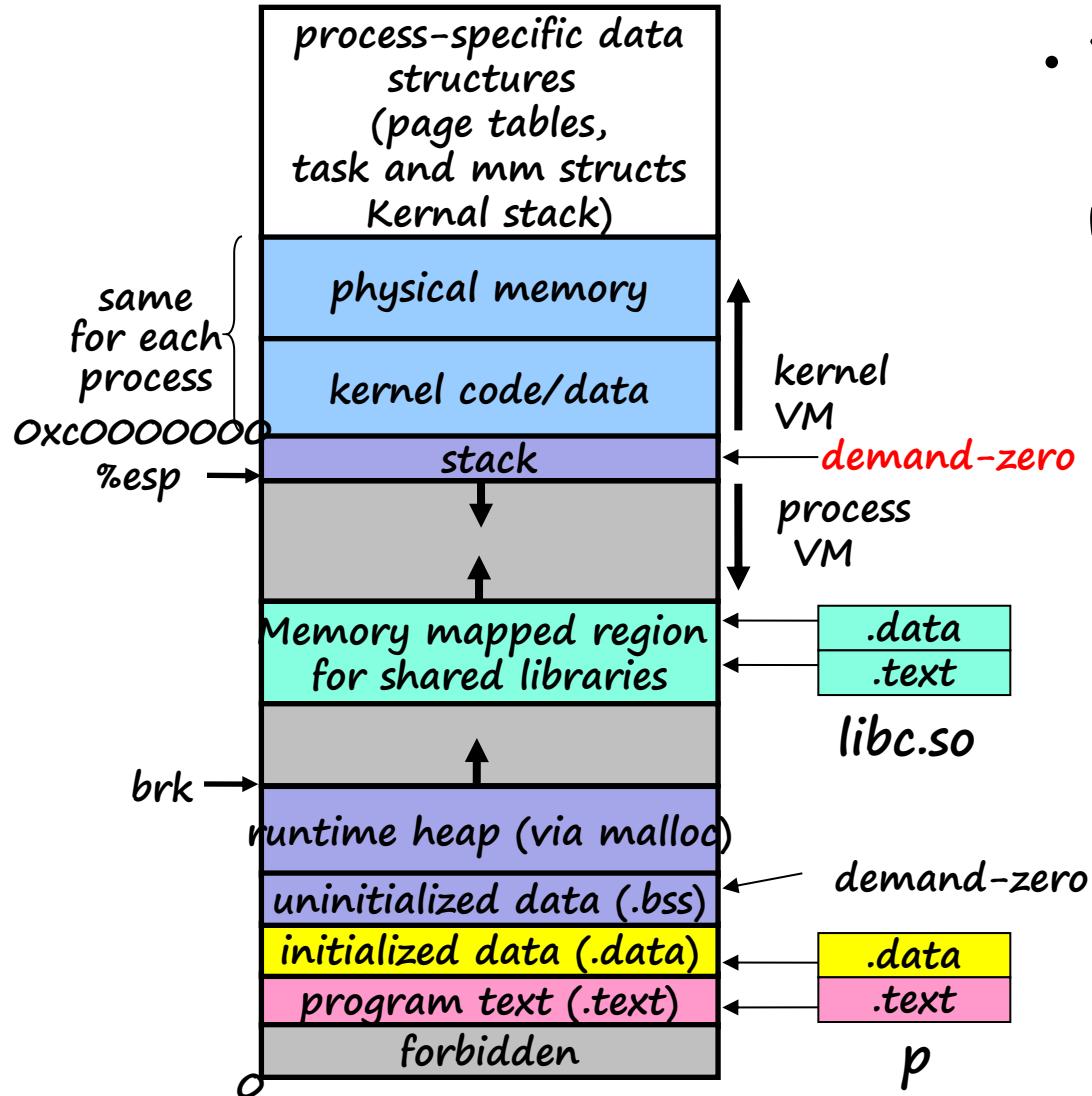
- To run a new program `p` in the current process using `exec()`:
  - Free all **vm\_area\_structs** and **page tables** for old areas.

# Exec() revisited



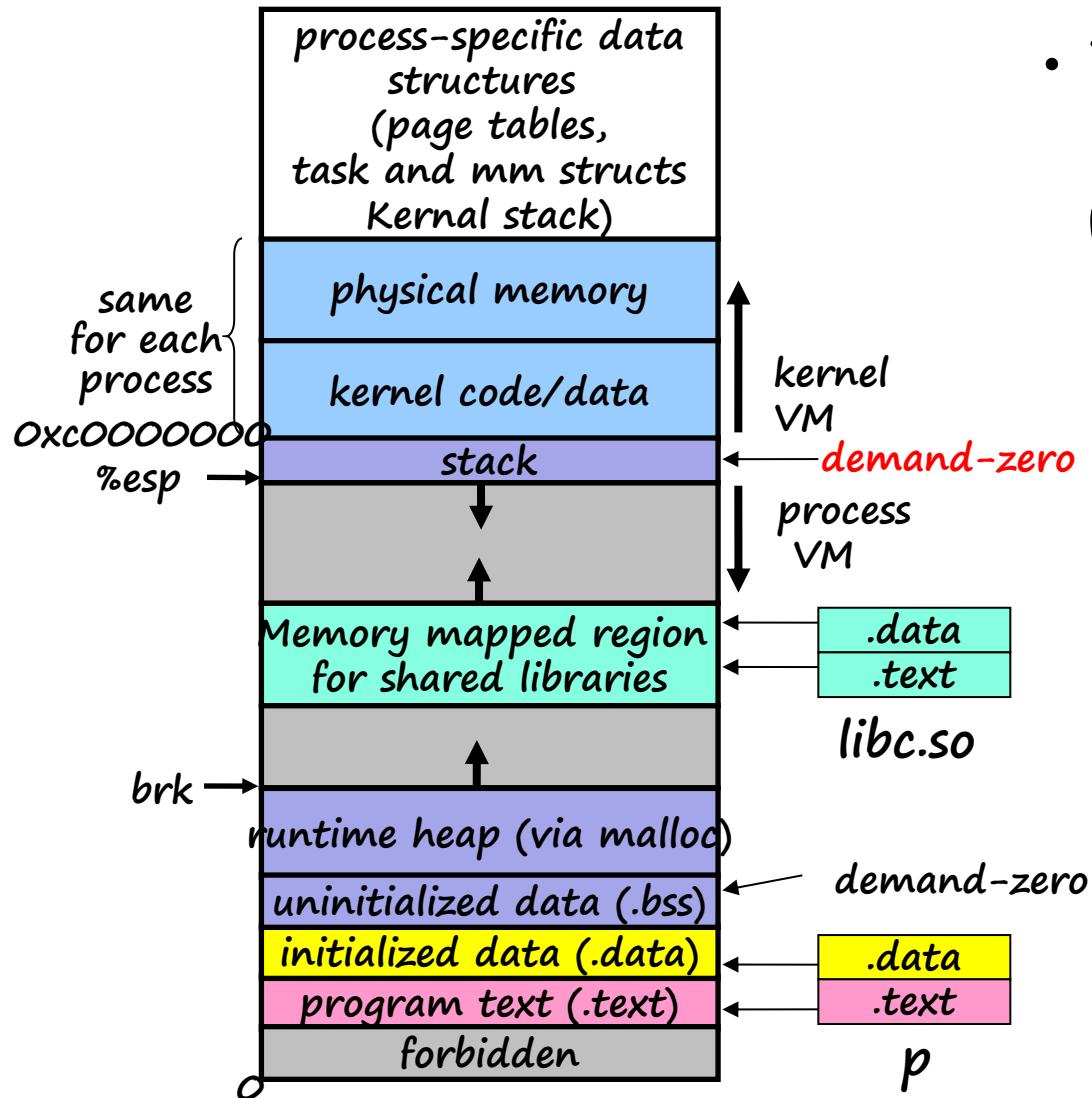
- To run a new program **p** in the current process using **exec()**:
  - create new **vm\_area\_structs** and **page tables** for new areas.
  - stack, bss, data, text, shared libs.

# Exec() revisited



- To run a new program *p* in the current process using `exec()`:
  - create new `vm_area_structs` and page tables for new areas.
  - **text** and **data** backed by ELF executable object file.
  - **bss** and **stack** initialized to zero.

# Exec() revisited



- To run a new program *p* in the current process using `exec()`:
  - set PC to entry point in *.text*
  - Linux will swap in code and data pages **as needed**.

# User-level memory mapping

---

```
void *mmap(void *start, int len, int prot,  
           int flags, int fd, int offset)
```

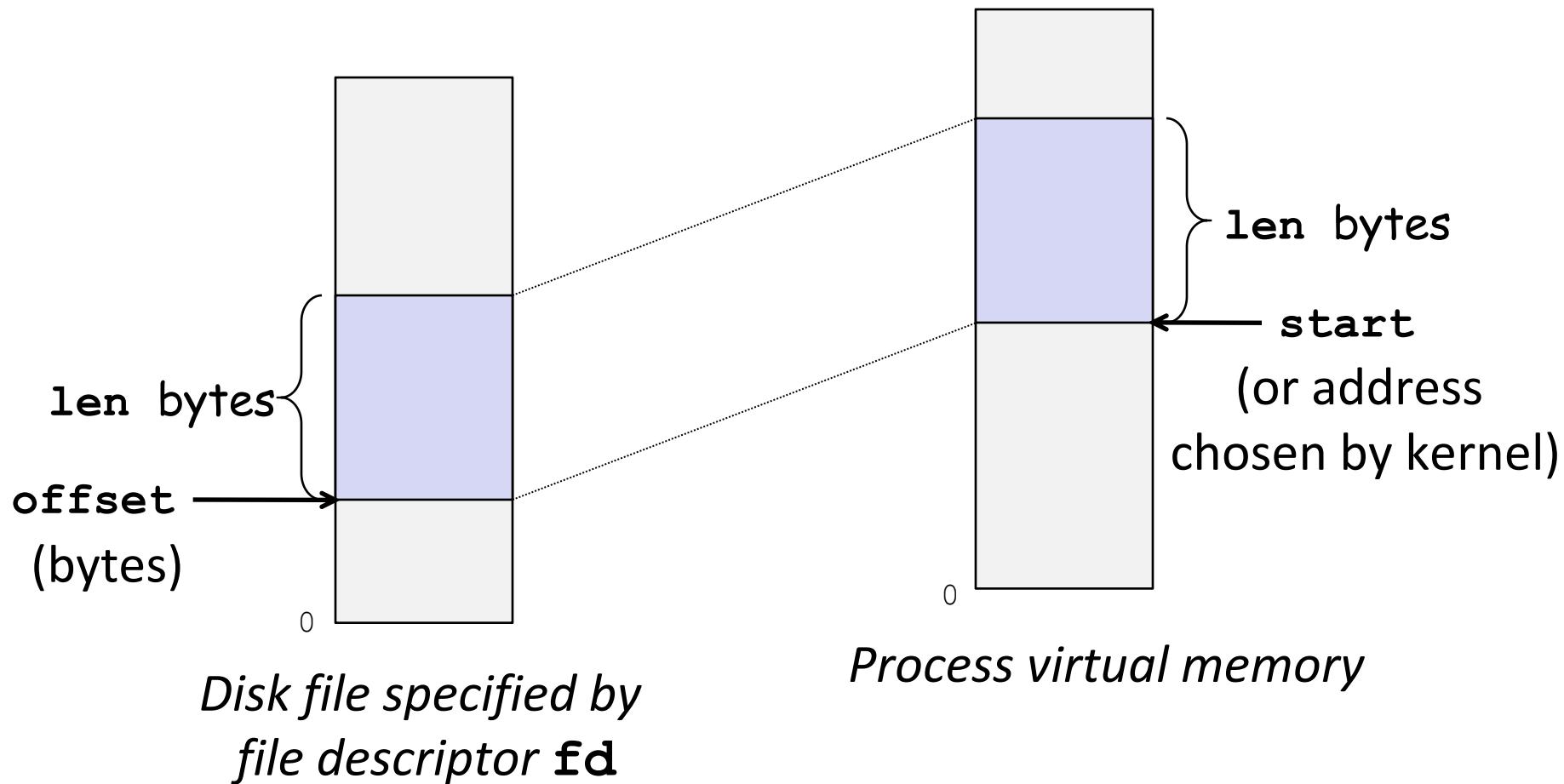
Map **len** bytes starting at offset **offset** of the file specified by file description **fd**, preferably at address **start** (usually 0 for don't care).

- **prot**: `MAP_READ`, `MAP_WRITE`
- **flags**: `MAP_PRIVATE`, `MAP_SHARED`

Return a pointer to the mapped area

# User-level memory mapping

```
void *mmap(void *start, int len,  
           int prot, int flags, int fd, int offset)
```



# User-level memory mapping

---

- Example: fast file copy
  - Useful for applications like Web servers that need to quickly copy files.
  - **mmap** allows file transfers without copying into user space.

# mmap() example: fast file copy

```
/*
 * mmapcopy - uses mmap to copy file fd to stdout
 */

void mmapcopy(int fd, int size)
{
    char *bufp;
    /* map the file to a new VM area */
    bufp = mmap(0, size, PROT_READ,
                MAP_PRIVATE, fd, 0);
    /* write the VM area to stdout */
    write(1, bufp, size);
    return ;
}
```

# mmap() example: fast file copy

---

```
int main(int argc, char **argv)
{
    struct stat stat;
    /* check for required command line argument */
    if ( argc != 2 ) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(0) ;
    }
    /* open the file and get its size*/
    fd = open(argv[1], O_RDONLY, 0);
    fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
}
```

# Fork() revisited

---

- To create a new process using **fork**:
  - make copies of the old process's **mm\_struct**, **vm\_area\_struct**, and **page tables**
    - Two processes are sharing all of their pages  
(At this point)
    - How to get separate spaces without copying all the virtual pages from one space to another?
      - "**Copy On Write**" (COW) technique.

# Shared Object

---

- Shared Object
  - An object which is mapped into an area of virtual memory of a process
  - Any writes that the process makes to that area are visible to any other processes that have also mapped the shared object into their virtual memory
  - The changes are also reflected in the original object on disk
- Shared Area
  - A virtual memory area that a shared object is mapped

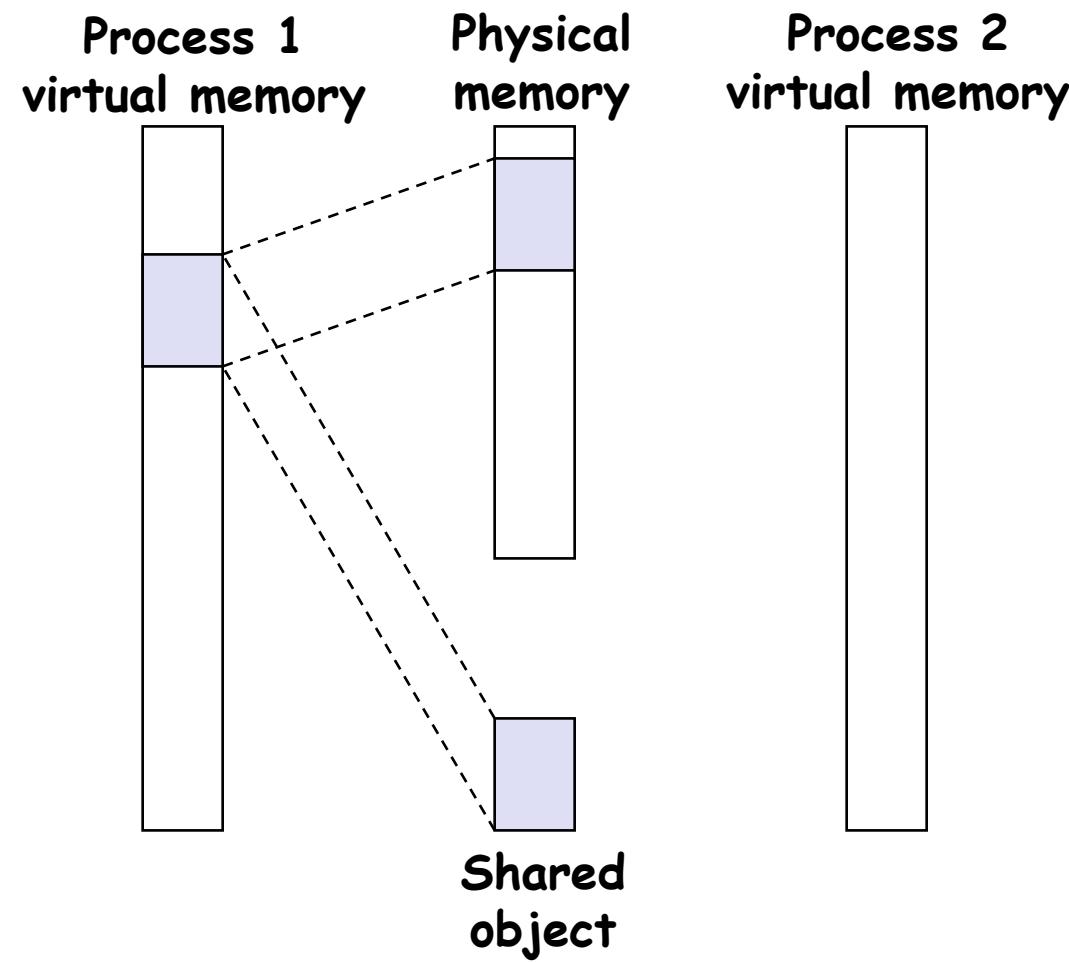
# Private object

---

- Private Object
  - As oppose to shared object
  - Changes made to an area mapped to a private object are not visible to other processes
  - Any writes that the process makes to the area are not reflected back to the object on disk.
- Private Area
  - Similar to shared area

# Sharing Revisited: Shared Objects

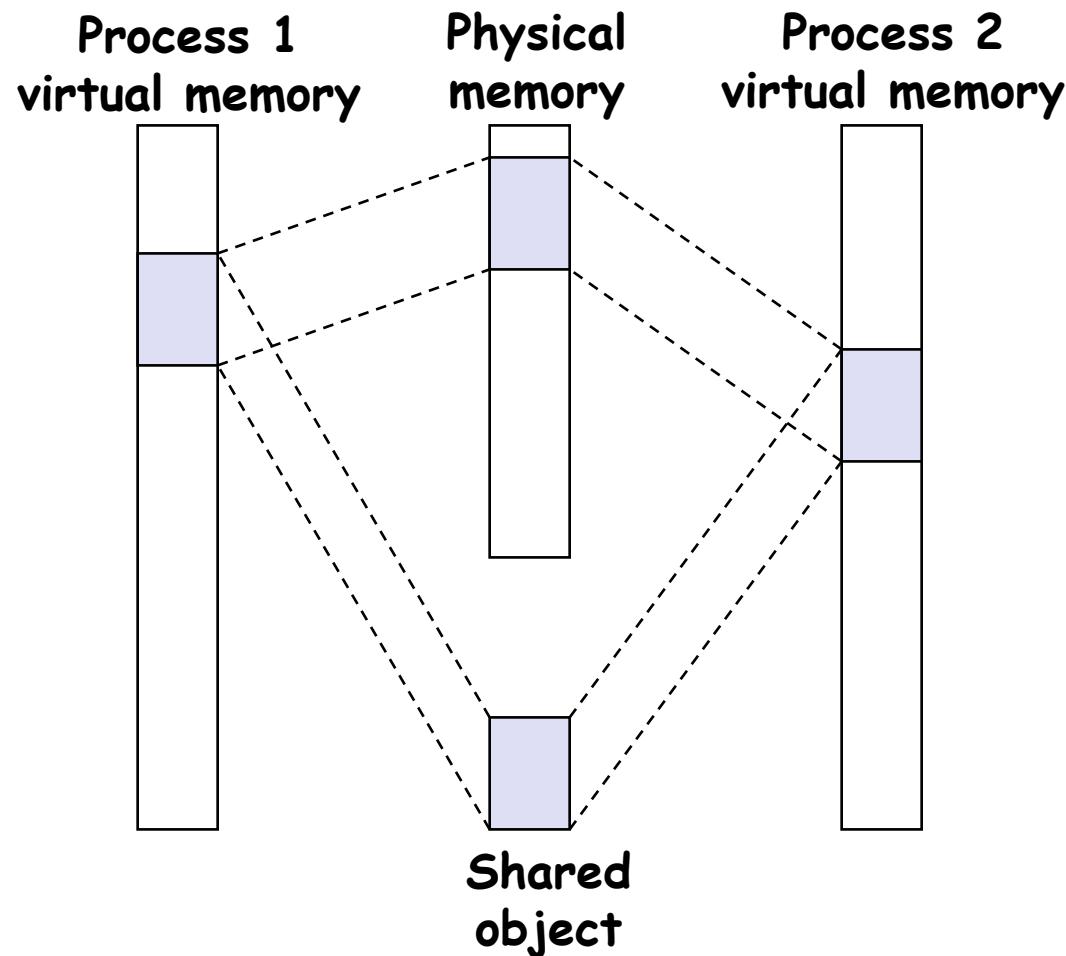
---



- Process 1 maps the shared object

# Sharing Revisited: Shared Objects

---



- Process 2 maps the shared object
- Notice how the virtual addresses can be **different**

# Copy-on-Write

---

- A private object begins life in exactly the same way as a shared object, with only one copy of the private object stored in physical memory.

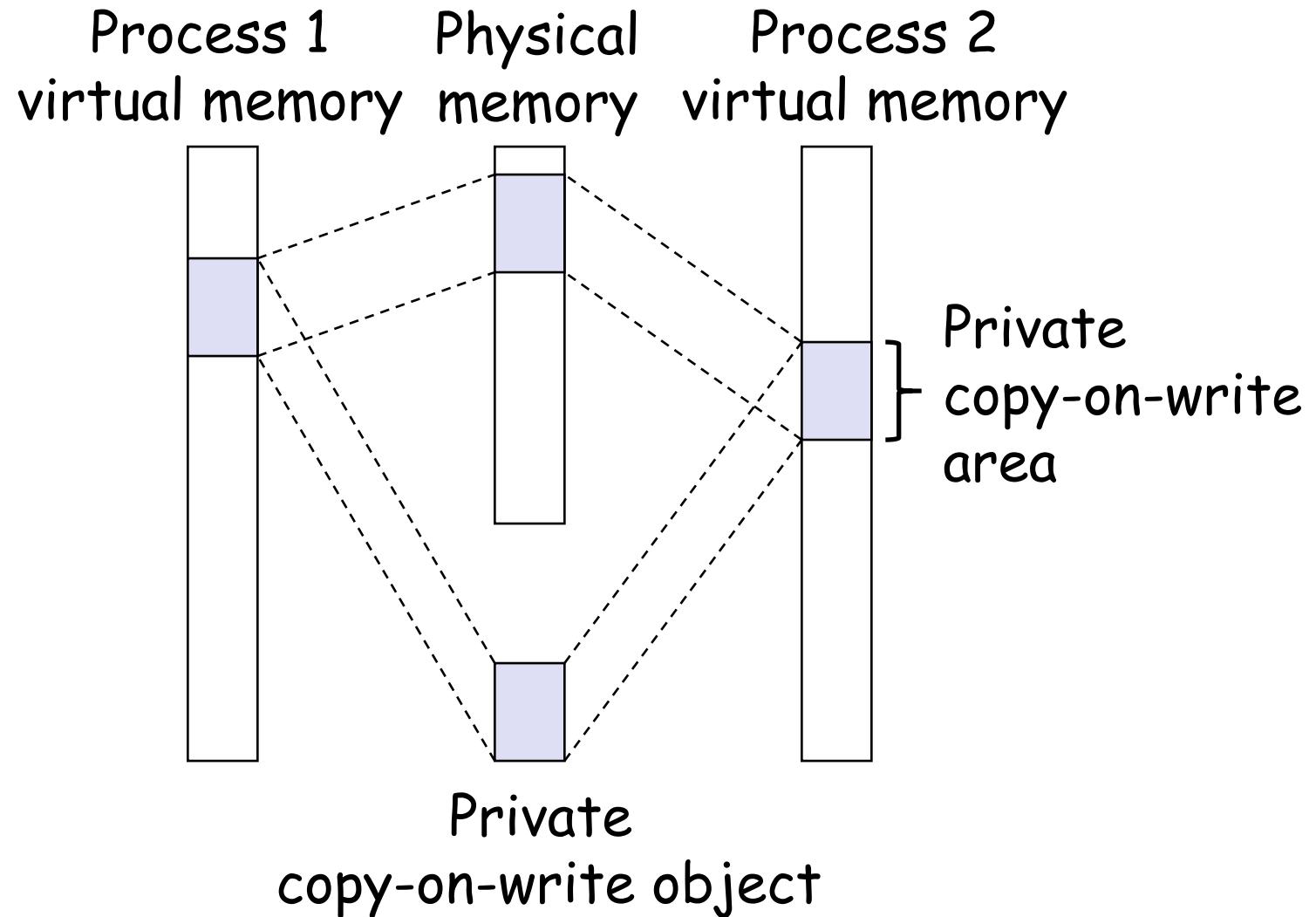
# Fork() revisited

---

- To create a new process using fork
  - Copy-On-Write
    - make pages of writeable areas **read-only**
    - flag **vm\_area\_struct** for these areas as **private** "copy-on-write".
    - writes by **either** process to these pages will cause **page faults**
      - fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.

# Sharing Revisited: Private COW Objects

---



# Copy-on-Write

---

- For each process that maps the private object
  - The page table entries for the corresponding private area are flagged as **read-only**
  - The area struct is flagged as **private copy-on-write**
  - So long as neither process attempts to write to its respective private area, they continue to share a single copy of the object in physical memory.

# Copy-on-Write

---

- For each process that maps the private object
  - As soon as a process attempts to **write** to some page in the private area, the write triggers a protection fault
  - The fault handler notices that the protection exception was caused by the process trying to write to a page in a private copy-on-write area

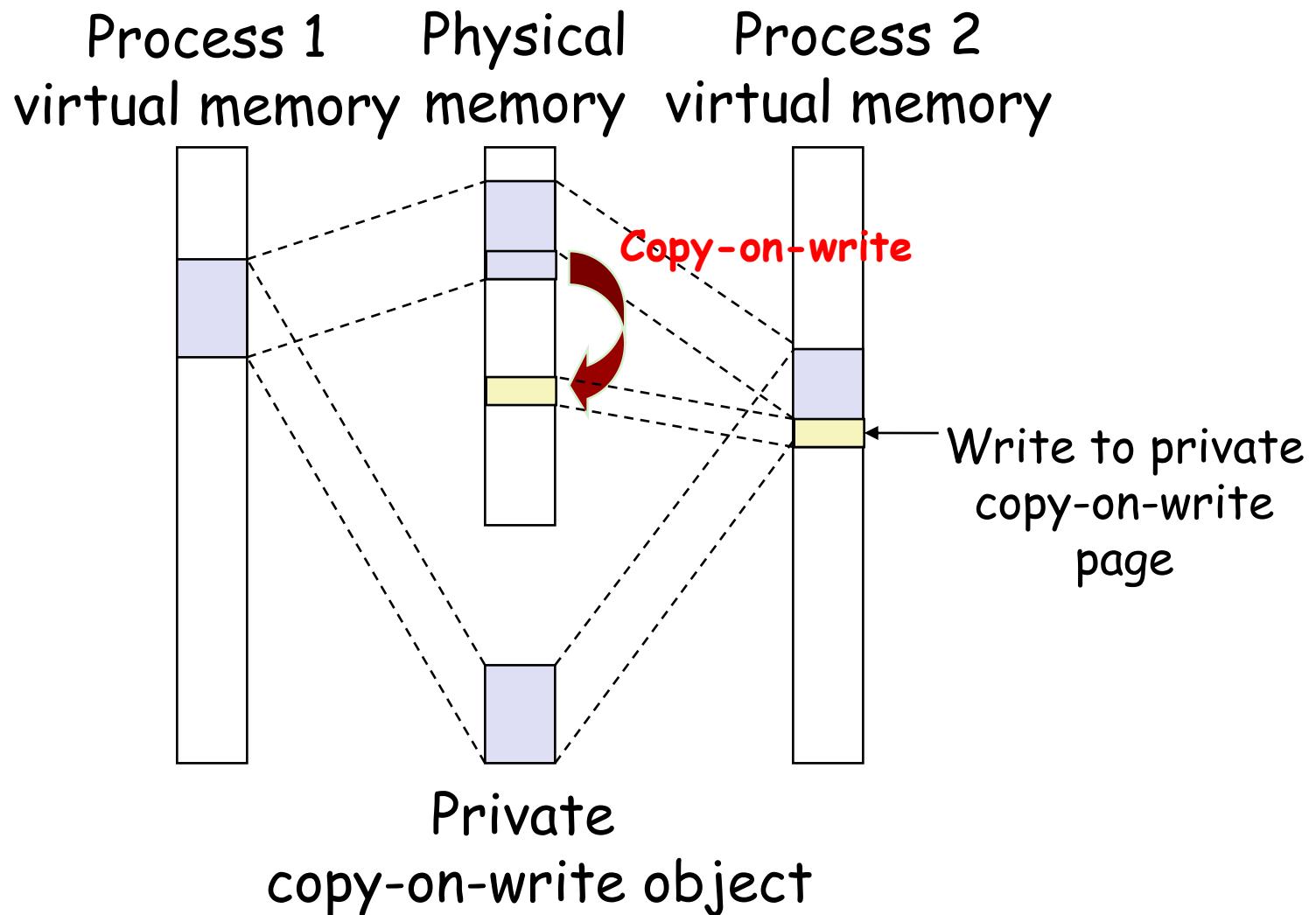
# Copy-on-Write

---

- For each process that maps the private object
  - The **fault handler**
    - Creates a new copy of the page in physical memory
    - Updates the page table entry to point to the new copy
    - Restores write permissions to the page

# Sharing Revisited: Private COW Objects

---



# Fork() revisited

---

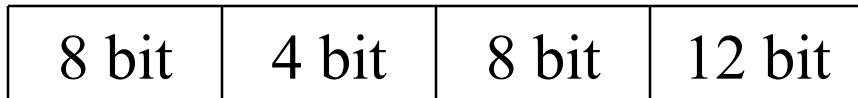
- To create a new process using fork:
  - Net result:
    - copies are deferred until absolutely necessary  
(i.e., when one of the processes tries to modify a shared page).

# Virtual Memory

## Case 1

---

- In a 32-bit machine we subdivide the virtual address into 4 pieces as follows:



- We use a 3-level page table, such that the first 7 bits are for the first level and so on.
- Physical addresses are 44 bits and there are 4 protection bits per page. (Byte Alignment)

## Case 1

---

- What is the page size in such system?
  - 4K ( $2^{12}$ )
- How much memory is consumed by the page table and wasted by internal fragmentation for a process that has 128K of memory starting at address 0?
  - #L3-entry:  $2^8 = 256$ , entry size: 5byte (44-12+4bits)
  - #L2-entry:  $2^4 = 16$ , entry size: 5byte (44-8+4bits)
  - #L1-entry:  $2^8 = 256$ , entry size: 6byte (44-4+4bits)

## Case 1

---

- How much memory is consumed by the page table and wasted by internal fragmentation for a process that has 128K of memory starting at address 0?
  - $128K = 4K * 32 \rightarrow 1*L1 + 1*L2 + 1*L3$
  - L3:  $256 * 5 = 1280$  Bytes
  - L2:  $16 * 5 = 80$  Bytes
  - L1:  $256 * 6 = 1536$  Bytes
  - Memory:  $1280 + 80 + 1536 = 2896$  Bytes
  - Internal:  $1*(5*8-36) + 1*(5*8-40) + 32*(6*8-44)$   
 $= 132$  bits

# Case 2

---

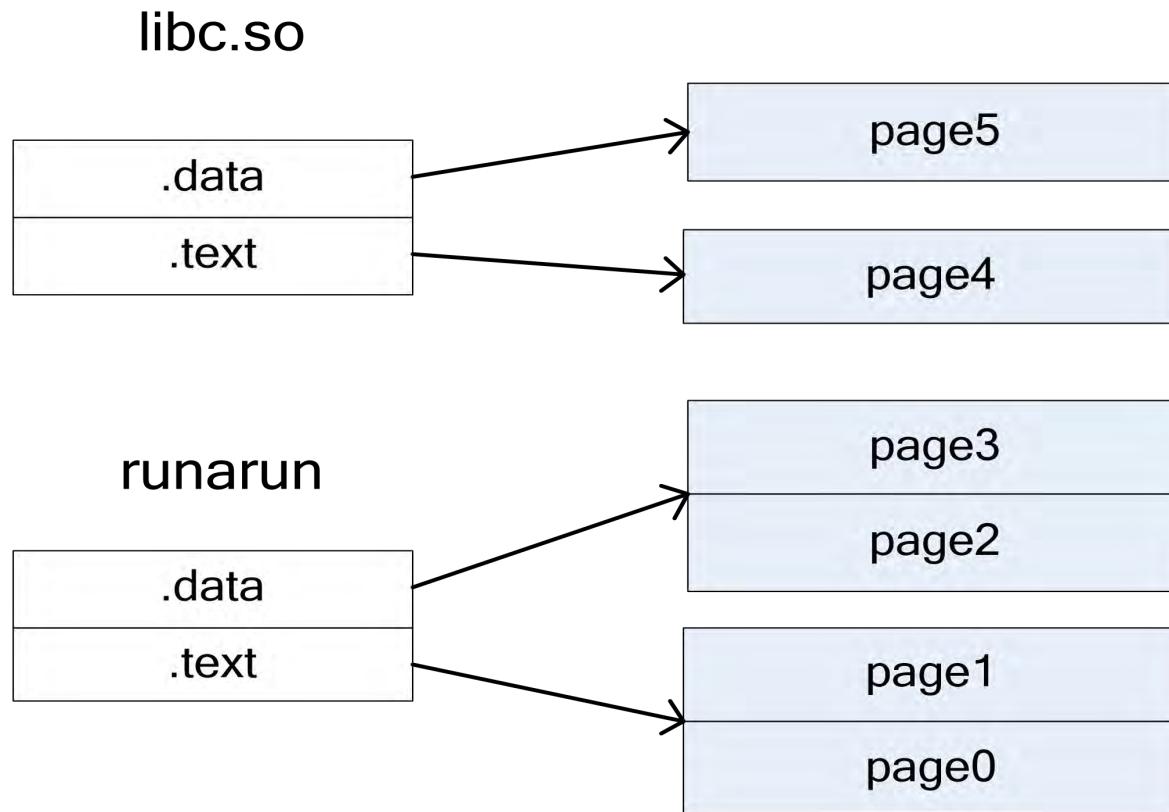
## Part 1

- Assume program “runarun” contains
  - 2 pages of code and
  - 2 pages of data, and
  - libc library contains 1 page of code and 1 page of data,
  - as shown in figure 1.
- Assume we have a machine with
  - 16 pages of physical memory, and
  - 64 pages of virtual memory.
  - User space takes 48 pages
  - kernel space takes 16 pages.

## Case 2

---

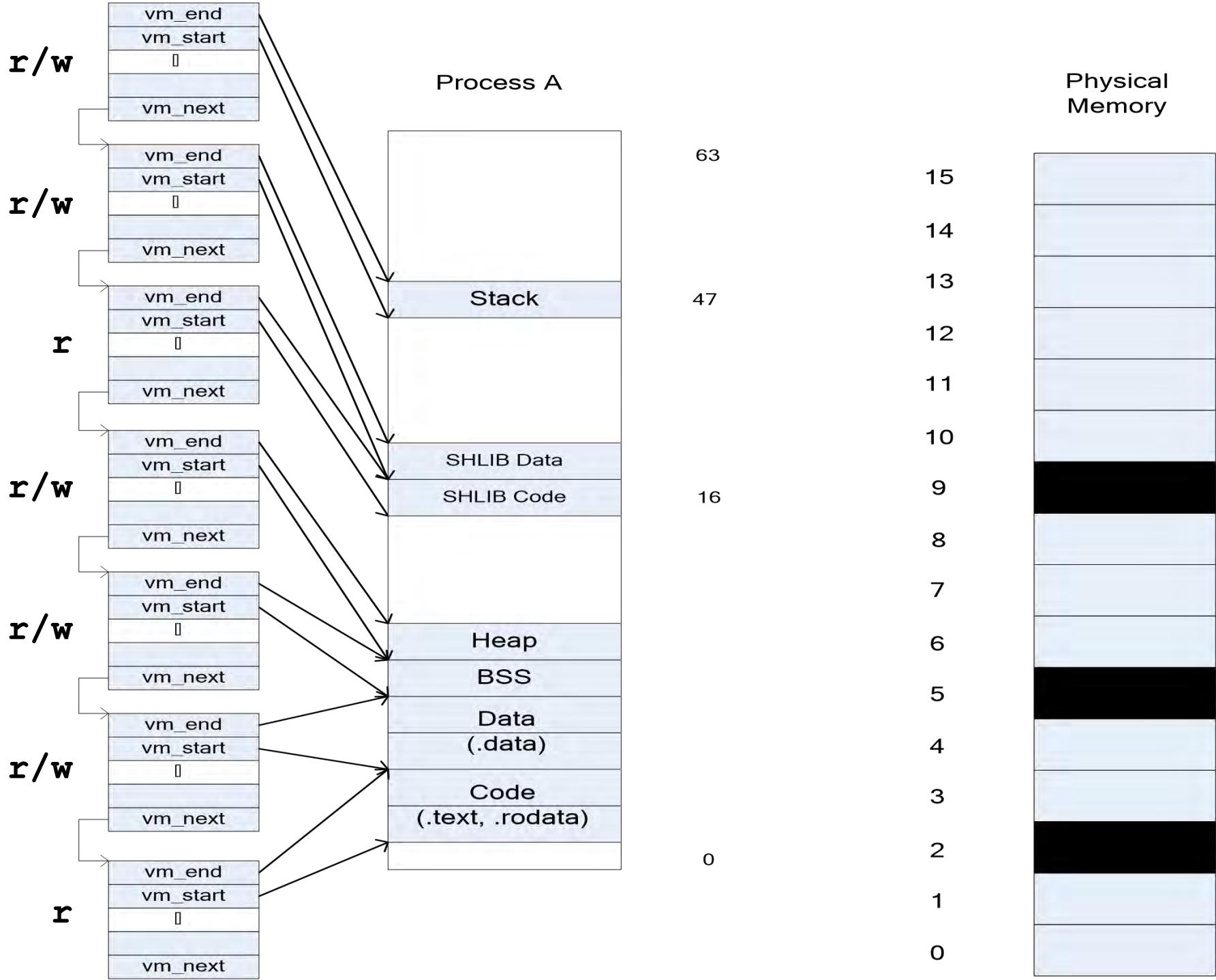
Figure 1



## Case 2

---

- The figure 2 shows
  - the runtime virtual memory layout of the program
  - the virtual memory areas
  - Each shadowed rectangle represents one page.
- Fill in the flags of virtual memory areas with "read-only" or "read/write"



# Case 2

---

## Part 2

- There are two types of page fault:
  - major and minor faults.
- major page faults occur
  - when data has to be read from disk
  - which is an expensive operation;
- minor page fault concerns
  - no disk operation,
  - the fault could be handled by
    - just memory read/write/copy operations.

# A sample from exam papers

---

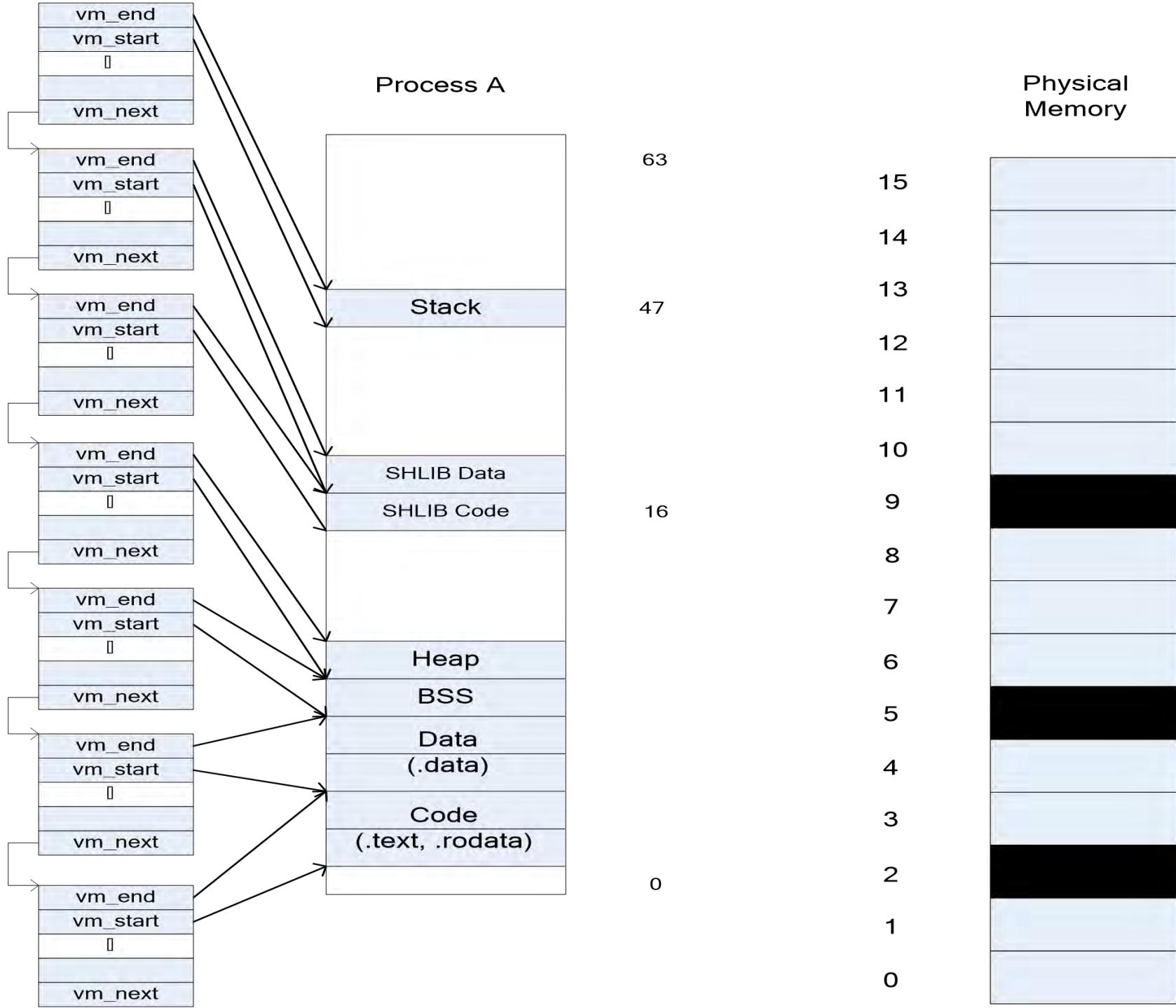
- During the lifetime of the program "runarun",
  - every of the code and data pages are visited.
- The process takes
  - one bss page,
  - one heap page
  - one stack page
- During the lifetime of the program, how many pages faults occurred?
  - Number of major page faults: text data
  - Number of minor page fault: bss heap stack

# A sample from exam papers

---

## Part 3

- Assume the system performs no page swapping.
- Before the program starts, physical memory looks like the table in Figure 2.
  - Blacked boxes (PPN:2,9) are already allocated.
- Parent process blocks itself and waits after forking, and continue after child exit



# Case 2

---

## Part 3

- Program binary and shared library are touched in the sequence of:
  - Parent process: <page4, page5, stack, page0, page2, bss, heap, **fork**, page1 >
  - Child Process: <stack, page1, page3, bss, page0, page2, heap>
  - Operations for data/bss/stack/heap are write
  - Don't consider reap

## Case 2

---

- A simple first-hit memory allocation policy is used.

```
slot_t *find_a_mem_slot()
{
    for(i = 0; i < 16; i++)
        if (slot_is_free)
            return slot[i];

    if (i == 16)
        return fail;
}
```

## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:

Page4

Page5

Stack

Page0

Page2

Bss

Heap

Fork

		Parent Process		Child Process	
Page4					
Page5	PPN : 3	stack	47	stack	47
Stack					
Page0	PPN : 1	page5		PPN : 1	
	PPN : 0	page4	16	PPN : 0	16
Page2					
Bss	PPN : 8	heap		PPN : 8	
	PPN : 7	bss		PPN : 7	
Heap	PPN : 6	page3		page3	
		page2		page2	
		page1		page1	
Fork	PPN : 4	page0	0	PPN : 4	0

## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:    Child:

Page4      Stack

Page5

**PPN : 3**

Stack

Page0

**PPN : 1**

Page2

**PPN : 0**

Bss

**PPN : 8**

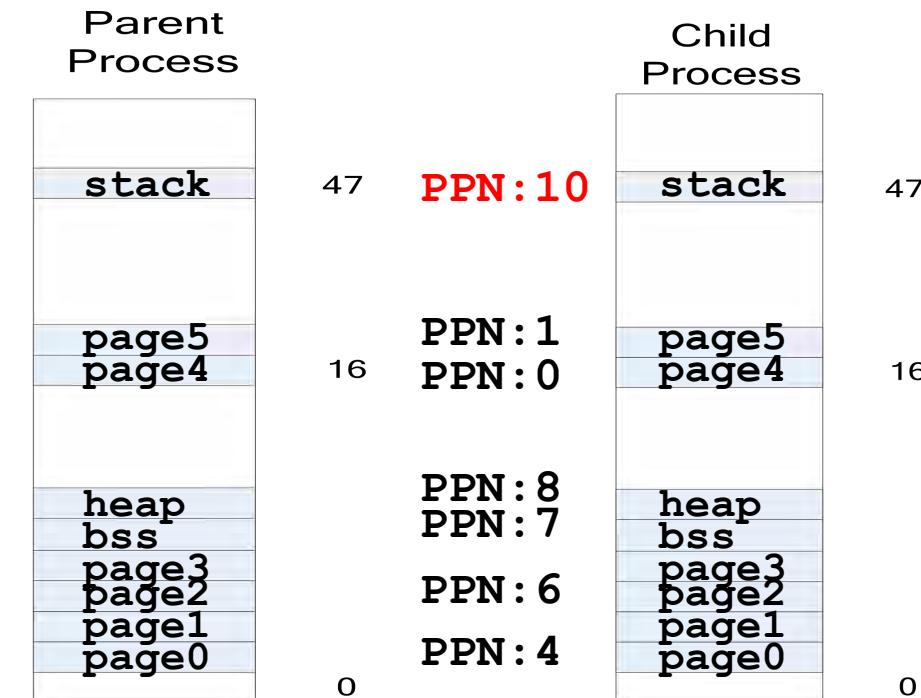
Heap

**PPN : 7**

Fork

**PPN : 6**

**PPN : 4**



## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:    Child:

Page4      Stack

Page5      Page1      **PPN : 3**

Stack

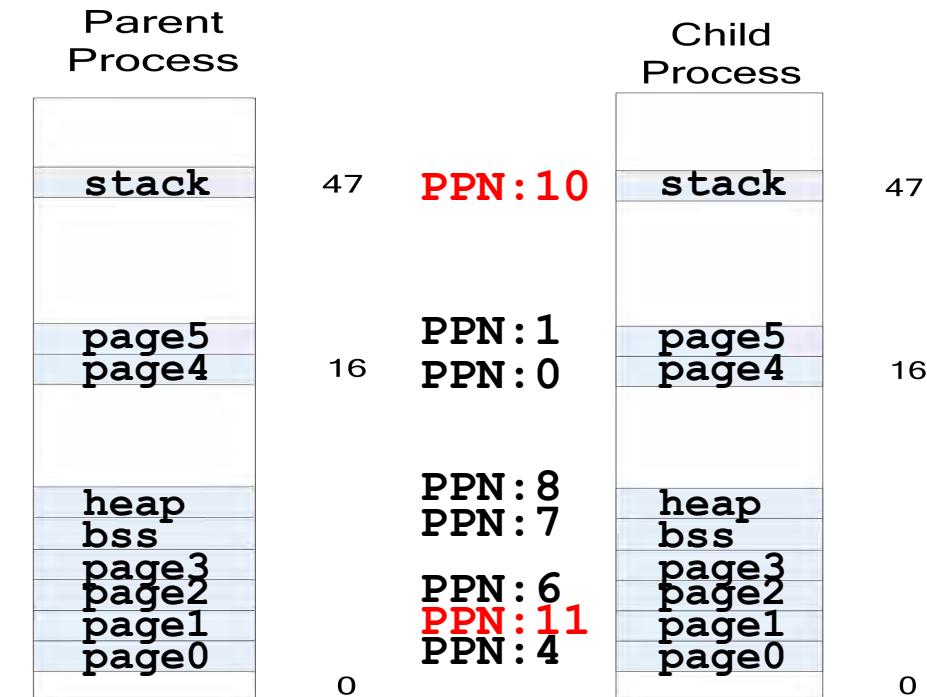
Page0      **PPN : 1**  
              **PPN : 0**

Page2

Bss      **PPN : 8**  
              **PPN : 7**

Heap      **PPN : 6**

Fork      **PPN : 4**



## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:    Child:

Page4      Stack

Page5      Page1      **PPN : 3**

Stack      Page3

Page0      **PPN : 1**

**PPN : 0**

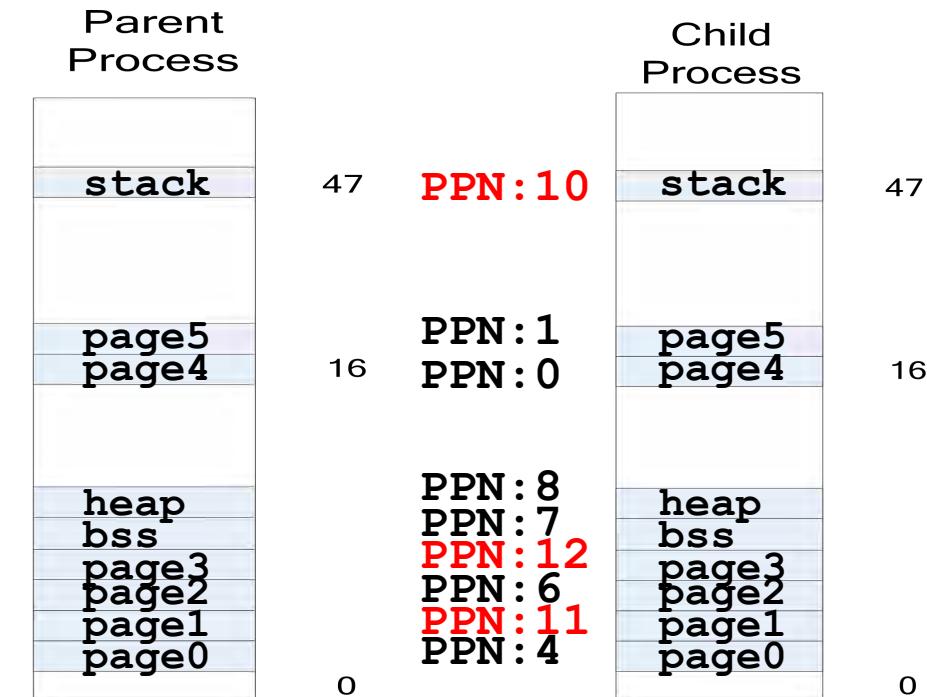
Page2

Bss      **PPN : 8**

**PPN : 7**

Heap      **PPN : 6**

Fork      **PPN : 4**



## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:    Child:

Page4      Stack

Page5      Page1      **PPN : 3**

Stack      Page3

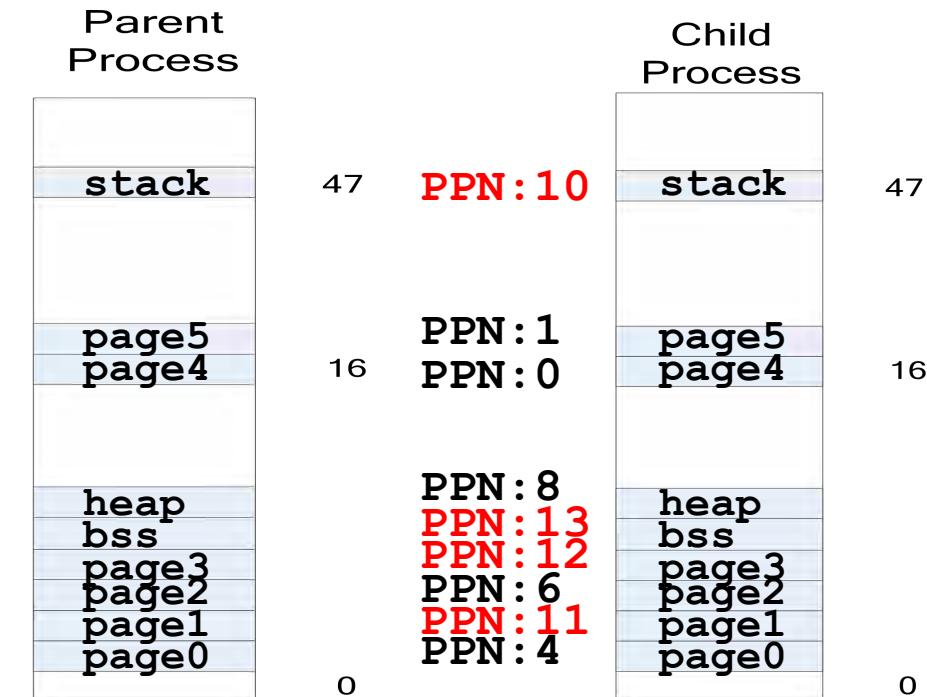
Page0      Bss      **PPN : 1**  
                       **PPN : 0**

Page2

Bss      **PPN : 8**  
                       **PPN : 7**

Heap      **PPN : 6**

Fork      **PPN : 4**



## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:    Child:

Page4      Stack

Page5      Page1      **PPN : 3**

Stack      Page3

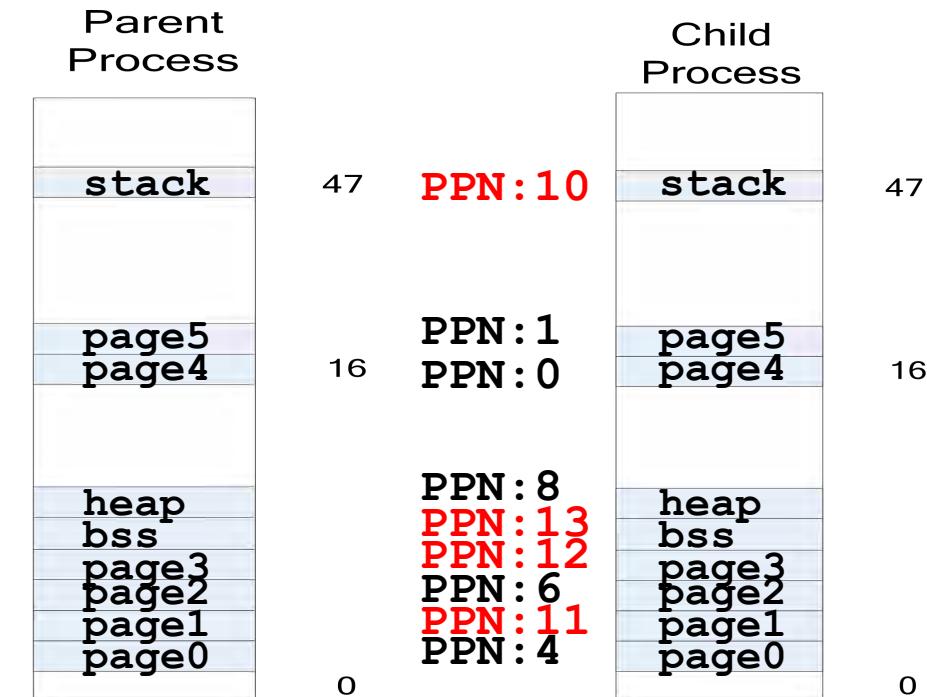
Page0      Bss      **PPN : 1**  
                   **PPN : 0**

Page2      Page0

Bss      **PPN : 8**  
                   **PPN : 7**

Heap      **PPN : 6**

Fork      **PPN : 4**



## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:    Child:

Page4      Stack

Page5      Page1      **PPN : 3**

Stack      Page3

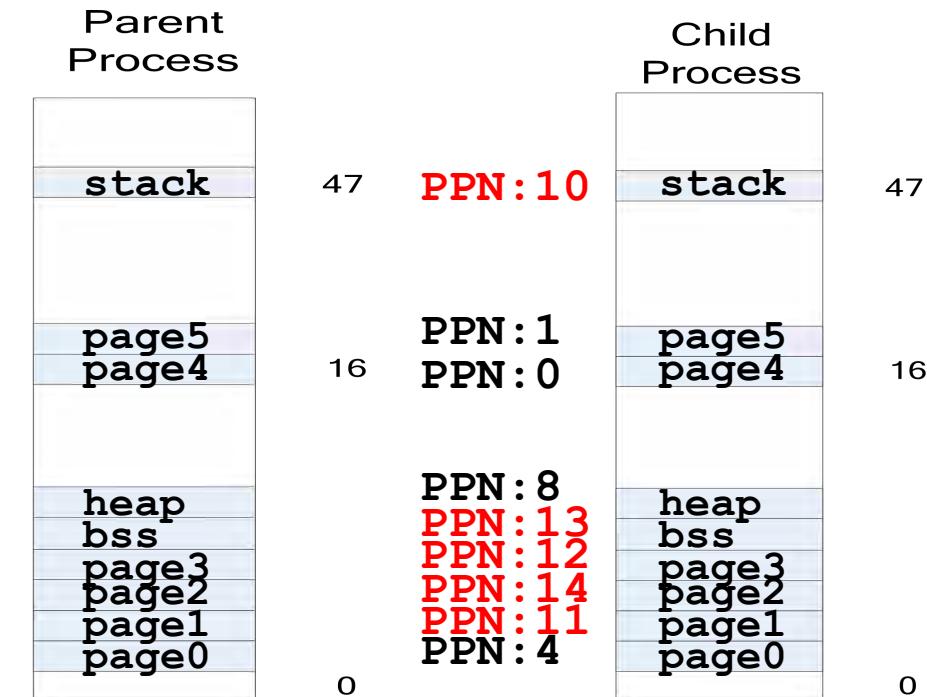
Page0      Bss      **PPN : 1**  
                   **PPN : 0**

Page2      Page0

Bss      Page2      **PPN : 8**  
                   **PPN : 7**

Heap           **PPN : 6**

Fork           **PPN : 4**



## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:    Child:

Page4      Stack

Page5      Page1      **PPN : 3**

Stack      Page3

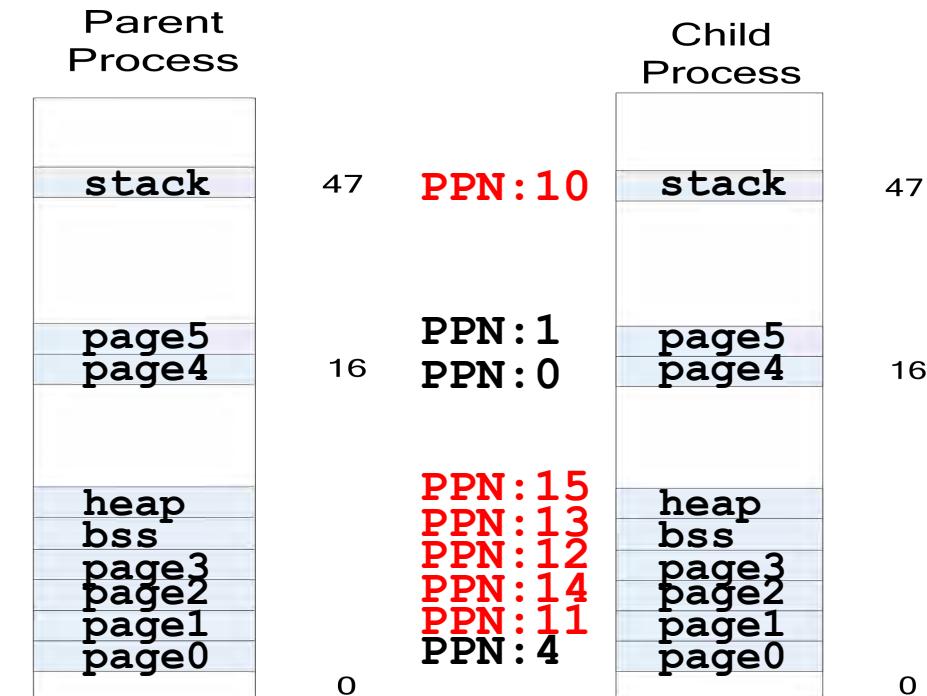
Page0      Bss      **PPN : 1**  
                   **PPN : 0**

Page2      Page0

Bss      Page2      **PPN : 8**  
                   **PPN : 7**

Heap      Heap      **PPN : 6**

Fork           **PPN : 4**



## Case 2

---

- Please fill in the flat page tables with PPN below for both the parent process and the child process just before they finished executing.

Parent:    Child:

Page4      Stack

Page5      Page1      **PPN : 3**

Stack      Page3

Page0      Bss      **PPN : 1**  
                   **PPN : 0**

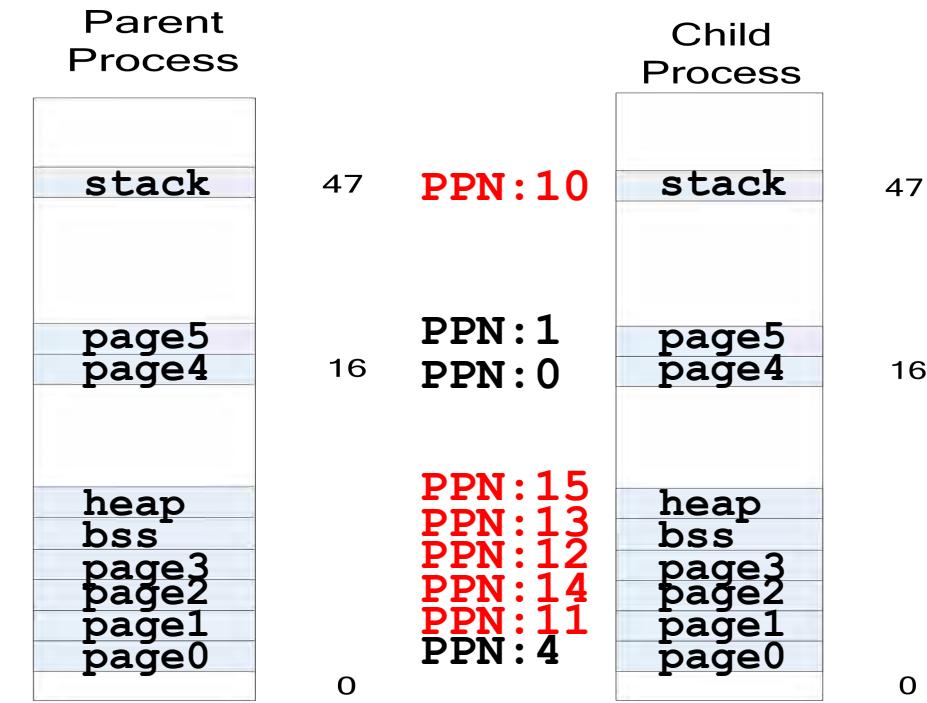
Page2      Page0

Bss      Page2      **PPN : 8**  
                   **PPN : 7**

Heap      Heap      **PPN : 6**

**Fork**      **PPN : 11**  
                   **PPN : 4**

Page1



## Case 3

---

```
do_page_fault()
{
    //page fault virtual address
    address = get_page_fault_addr();
    //the reason of the page fault
    type = get_page_fault_type();
    //find the vm area which the address falls in
    vma = find_vma(address);
    if (!vma)
        return fail;

    if (type == WRITE_PROTECT_FAULT)
        do_copy_on_write(); //not detailed here
```

## Case 3

---

```
/* page not present */
if (vma->type == FILE_PAGE) { //file-backed vm area
    //load file page into memory and
    //update the page table
    load_file();
    return success;
} else if (vma->type == ANONYMOUS_PAGE) {
    // anonymous vm area
    //allocate a zero page and update the page table
    alloc_zero_page();
    return success;
}
/* unknown type */
return fail;
}
```

## Case 3

---

```
find_vma(address)
{
    //get the list head of the vm area
    vma = get_head_vma();
    while(vma) {
        if (address > vma->vm_end)
            return NULL;
        if (address >= vma->vm_start
            && address <= vma->vm_end)
            return vma;
        vma = vma->vm_next;
    }
    return NULL;
}
```

## Case 3

---

- The page fault handler is buggy.
- If the stack grows down one page, the page fault handler is likely to fail.
- Why?

## Case 4

---

- The following program executes on Pentium/Linux

```
#define PAGE_SIZE 4 * 1024
#define BUF_SIZE 128*PAGE_SIZE

int main(void) {
    char *p = NULL;
    int i;
A:   p = mmap(0, 128 * PAGE_SIZE, PROT_READ |
            PROT_WRITE, MAP_PRIVATE | MAP_ANON, 0, 0);
    printf("buffer start: %p\n", p);
    for(i = 0; i < BUF_SIZE; i += PAGE_SIZE)
        p[i] = 1;
B:   munmap(p, BUF_SIZE);
    return;
}
```

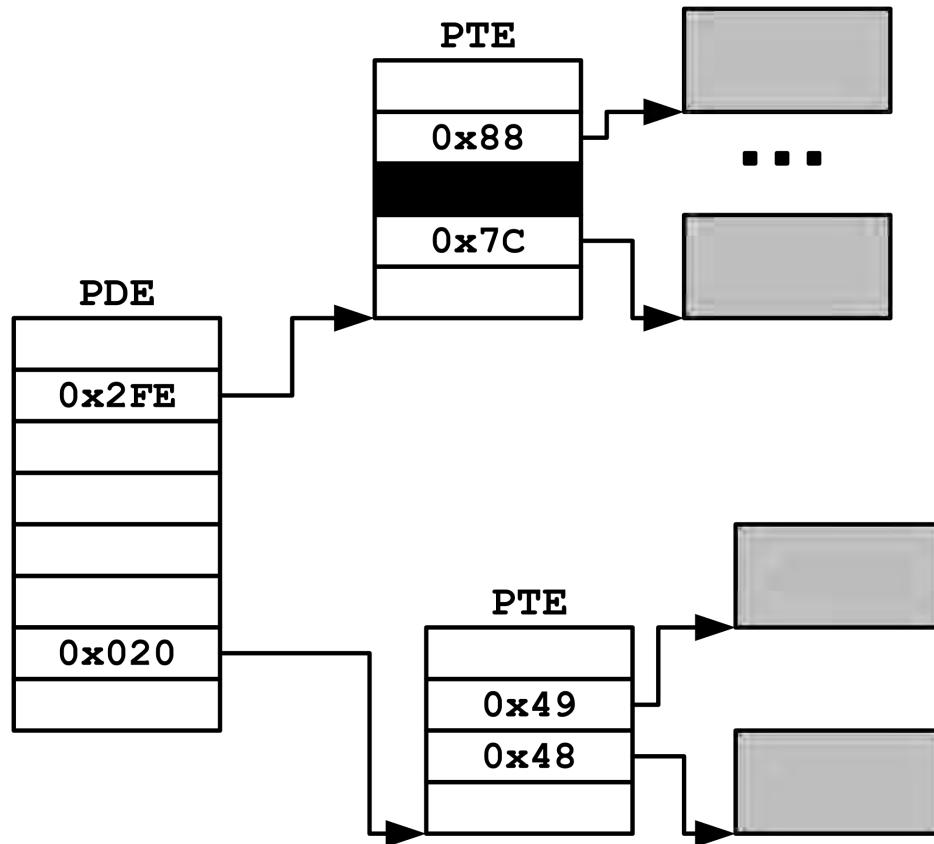
## Case 4

---

- When the program arrives at label A, the page table is like below. The number within block is the offset of PDE/PTE. The output of the function printf is “**buffer start: 0xb7bfd000**”.
- Please write the page table like above, when the program arrives at label B?
  - NOTE: the white block without number means empty PDE/PTE, the white block with number means a filled PDE/PTE, the black block means multiple filled PDE/PTEs, and the grey block means a page with some data/code

# Case 4

---



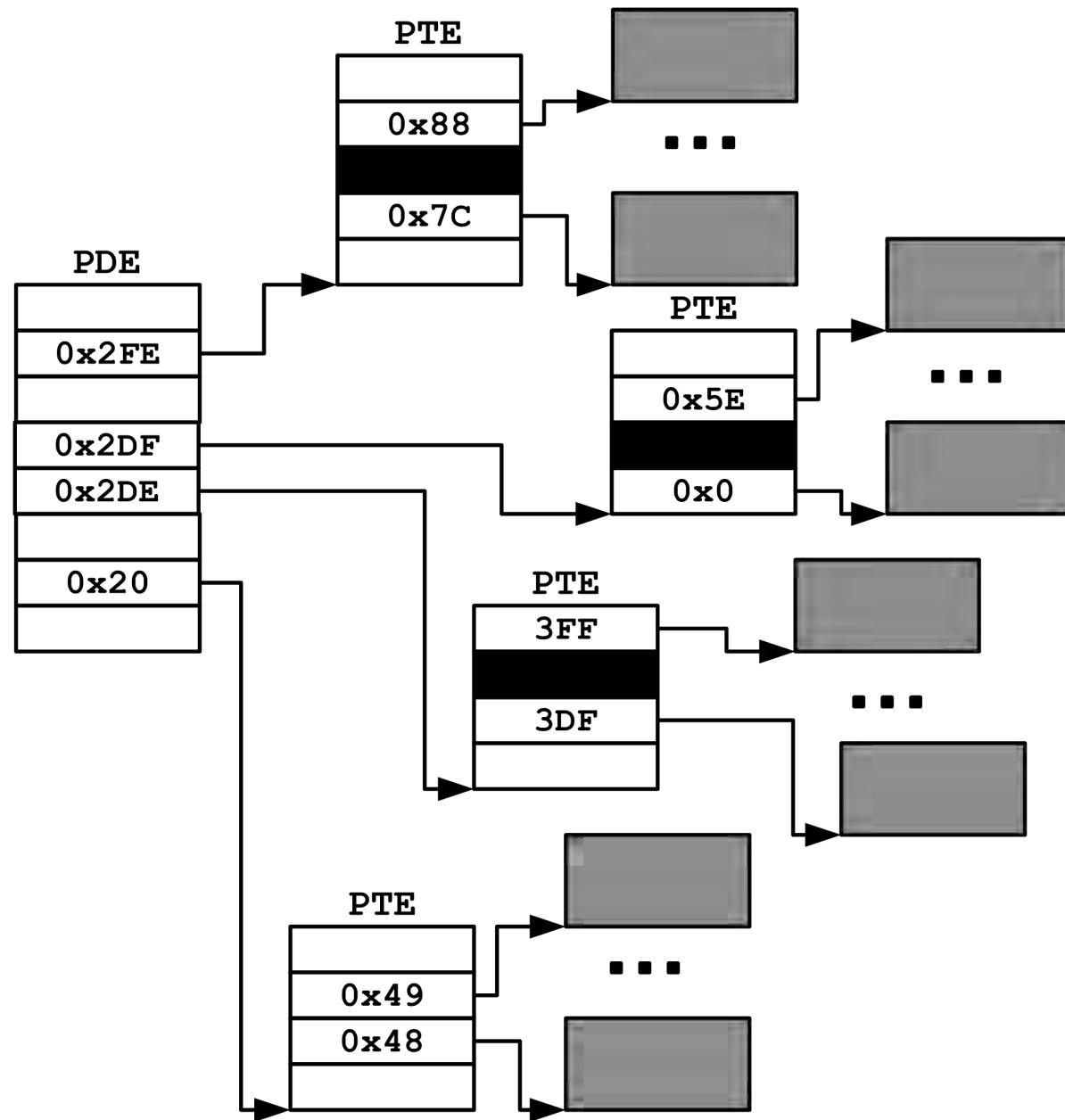
**Start**  
**0xb7bd000**

**L1: 2DE**  
**L2: 3DF**

**+128K**

**End**  
**0xb7c5f000**

**L1: 2DF**  
**L2: 05F**



# **CISC vs RISC**

# Y86

---

- Y86 Instruction Set Architecture
  - Similar state and instructions as IA32
  - Simpler encodings
  - Somewhere between CISC and RISC

# CISC

---

- CISC (pronounced “sisk”)
  - Complex Instruction Set Computer
  - Dominant style through mid-80's
- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions

# CISC

---

- Arithmetic instructions can access memory
  - addl %eax, 12(%ebx,%ecx,4)
    - requires memory read and write
    - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions

# CISC

---

- Philosophy
  - Add instructions to perform “typical” programming tasks

# RISC

---

- RISC (pronounced “risk”)
- Philosophy
  - generate efficient code for a much simpler form of instruction set
- Why not CISC ?
  - high-level instructions were very difficult to generate with a compiler and were seldom used



Search

TYPE HERE



## JOHN COCKE

United States – 1987

### CITATION

For significant contributions in the design and theory of compilers, the architecture of large systems and the development of reduced instruction set computers (RISC); for discovering and systematizing many fundamental transformations now used in optimizing compilers including reduction of operator strength, elimination of common subexpressions, register allocation, constant propagation, and dead code elimination.

## 约翰·科克

约翰·科克，又译为约翰·考克、约翰·寇克，生于美国北卡罗来纳州夏洛特，计算机科学家，在电脑架构及编译器最佳化技术方面有重大贡献，因此获得图灵奖。曾提出CYK算法。在他主导的IBM 801计划中，首次采用RISC架构，因此被称为RISC架构之父。[维基百科](#)

生于：1925年5月30日，美国北卡罗来纳州夏洛特（北卡罗来纳州）

逝于：2002年7月16日，美国纽约州瓦尔哈拉

教育背景：杜克大学（1953年），杜克大学（1946年）

所获奖项：图灵奖，约翰·冯诺依曼奖



更多图片

# CISC vs. RISC

---

## CISC

Variable-length encodings.

IA32 instructions can range from 1 to 15 bytes.

## RISC

Fixed-length encodings.

Typically all instructions are encoded as 4 bytes.

# CISC vs. RISC

---

## CISC

Multiple formats for specifying operands.  
e.g., a memory operand specifier can have many different combinations of displacement, base and index registers, and scale factors.

## RISC

Simple addressing formats.  
Typically just base and displacement addressing.

# CISC vs. RISC

---

## CISC

Arithmetic and logical operations can be applied to both **memory** and **register** operands.

## RISC

Arithmetic and logical operations only use **register** operands.

Memory referencing is only allowed by **load** and **store** instructions. This convention is referred to as a **load/store** architecture.

# CISC vs. RISC

---

## CISC

Implementation artifacts **hidden** from machine-level programs. The ISA provides a **clean** abstraction between programs and how they get executed.

## RISC

Implementation artifacts **exposed** to machine-level programs.  
(e.g., Some RISC machines prohibit particular instruction sequences and have jumps that do not take effect until the following instruction is executed. The compiler is given the task of optimizing performance within these constraints.)

# CISC vs. RISC

---

## CISC

*Condition codes.* Special flags are set as a side effect of instructions and then used for conditional branch testing.

## RISC

No condition codes. Instead, explicit test instructions store the test results in normal registers for use in conditional evaluation.

# CISC vs. RISC

---

## CISC

Stack-intensive procedure linkage. The stack is used for procedure **arguments** and **return addresses**.

## RISC

Register-intensive procedure linkage. Registers are used for procedure **arguments** and **return addresses**. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.

# Review Y86

---

- Y86 ISA studies both CISC and RISC
  - CISC: condition codes, variable-length instructions, and stack-intensive procedure linkages
  - RISC: load-store architecture and a regular encoding

# Controversy and Fuse

---

- Original Debate
  - Strong opinions!
  - CISC proponents: easy for compiler, fewer code bytes
  - RISC proponents: better for optimizing compilers, can make run fast with simple chip design

# Controversy and Fuse

---

Neither RISC nor CISC in their *purest* forms were better than designs that *incorporated* the best ideas of both

- RISC ← CISC
  - More instructions (multiple cycles to execute)
  - ~~Implementation artifacts exposed to machine level programs~~

# Controversy and Fuse

---

Neither RISC nor CISC in their *purest* forms were better than designs that *incorporated* the best ideas of both

- CISC ← RISC
  - Dynamically translate CISC instruction into a sequence of simpler, RISC-like operations
  - X86-64: more register, close to register-intensive procedure linkage, ...

# CISC vs. RISC

---

- Current Status
  - For desktop processors, choice of ISA not a technical issue
    - With enough hardware, can make anything run fast
    - Code compatibility more important
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power

# At glance of MIPS

---

- MIPS emphasizes
  - A simple load-store instruction set
  - Design for pipelining efficiency, including a fixed instruction set encoding
  - Efficiency as a compiler target

# At glance of MIPS

---

- Registers
  - 32 64-bit general-purpose registers
  - 32 floating-point registers
  - The value of R0 is always 0
- Data Type
  - 8-, 16-, 32- and 64-bit integer data
  - 32-, 64-bit floating point data

# At glance of MIPS

---

- Addressing Modes
  - immediate and displacement (offset)
  - 16-bits fields

# At glance of MIPS

---

- Instruction Format
  - 32-bit fixed instruction
  - 6-bit primary opcode
  - I-type, R-type and J-type

### I-type

<b>Opcode</b>	<b>rs</b>	<b>rt</b>	<b>imm</b>
---------------	-----------	-----------	------------

Encodes: Loads and stores of bytes, half words, words, double words. All immediates ( $rt \leftarrow rs \text{ op immediate}$ )

Conditional branch instructions (rs is register, rd unused)

Jump register, jump and link register

(rd=0, rs=destination, immediate=0)

### J-type

<b>Opcode</b>	<b>Offset added to PC</b>
---------------	---------------------------

Jump and jump and link  
Trap and return from exception

<b>Opcode</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>shamt</b>	<b>funct</b>
---------------	-----------	-----------	-----------	--------------	--------------

Register-register ALU operations:  $rd \leftarrow rs \text{ funct } rt$

Function encodes the data path operation: Add, Sub, . . .

Read/write special registers and moves

# At glance of MIPS

---

- Operations
  - Load and Store **I-type**

**LD R1 , 30 (R2)**

Regs [R1]  $\leftarrow$  Mem [30+Regs [R2]]

**SD R3 , 500 (R4)**

Mem [500+Regs [R4]]  $\leftarrow$  Regs [R3]

# At glance of MIPS

---

- Operations
  - Arithmetic/logical instructions **R-type**

**DADDU R1, R2, R3**

Regs [R1]  $\leftarrow$  Regs [R2] + Regs [R3]

**DADDIU R1, R2, #3**

Regs [R1]  $\leftarrow$  Regs [R2] + 3

**DSLL R1, R2, #5**

Regs [R1]  $\leftarrow$  Regs [R2]  $\ll$  5

**SLT R1, R2, R3**

Regs [R1]  $\leftarrow$  (Regs [R2] < Regs [R3]) ? 1 : 0

# At glance of MIPS

---

- Operations
  - Control Flow

<b>J-type</b>	<b>J name</b>	$PC_{36..63} \leftarrow \text{name}$
	<b>JAL name</b>	$\text{Regs}[R31] \leftarrow PC+8; PC_{36..63} \leftarrow \text{name}$ $((PC+4)-2^{27}) \leq \text{name} \leq ((PC+4)+2^{27})$
	<b>JR R3</b>	$PC \leftarrow \text{Regs}[R3]$
	<b>JALR R2</b>	$\text{Regs}[R31] \leftarrow PC+8; PC \leftarrow \text{Regs}[R3]$
<b>I-type</b>	<b>BEQZ R4 , name</b>	if $(\text{Regs}[R4]==0)$ $PC \leftarrow \text{name}$ $((PC+4)-2^{17}) \leq \text{name} \leq ((PC+4)+2^{17})$
<b>R-type</b>	<b>MOVZ R1 ,R2 ,R3</b>	if $(\text{Regs}[R3]==0)$ $\text{Regs}[R1] \leftarrow \text{Regs}[R2]$

# Thanks

---

# Branch Prediction on Pipeline

# Branch Behavior in Programs

---

- Based on SPEC benchmarks
  - Branches occur with a frequency of 14% to 16% in INT programs and 3% to 12% in FP programs.
  - About 75% of the branches are **forward** branches
  - 60% of **forward** branches are **taken**
  - 80% of **backward** branches are **taken**
  - 67% of all branches are **taken**

# Branch Behavior in Programs

---

Why are branches (especially backward branches) more likely to be **taken** than **not taken**?

# Dealing with Branch Stalls

---

- Approach 1: Stall until branch direction is clear
  - Simple
- Approach 2: Predict Branch **Not Taken**
  - Execute successor instructions in sequence  
(more efficient than Taken in deep pipeline)
  - PC+4 already calculated
  - 33% branches not taken on average

# Dealing with Branch Stalls

---

- Approach 3: Predict Branch Taken
  - 67% branches taken on average
  - But haven't yet calculated target address in deep pipeline
    - still incur a 1-cycle latency
  - Makes sense on machines where branch target is known before outcome (e.g., Y86)

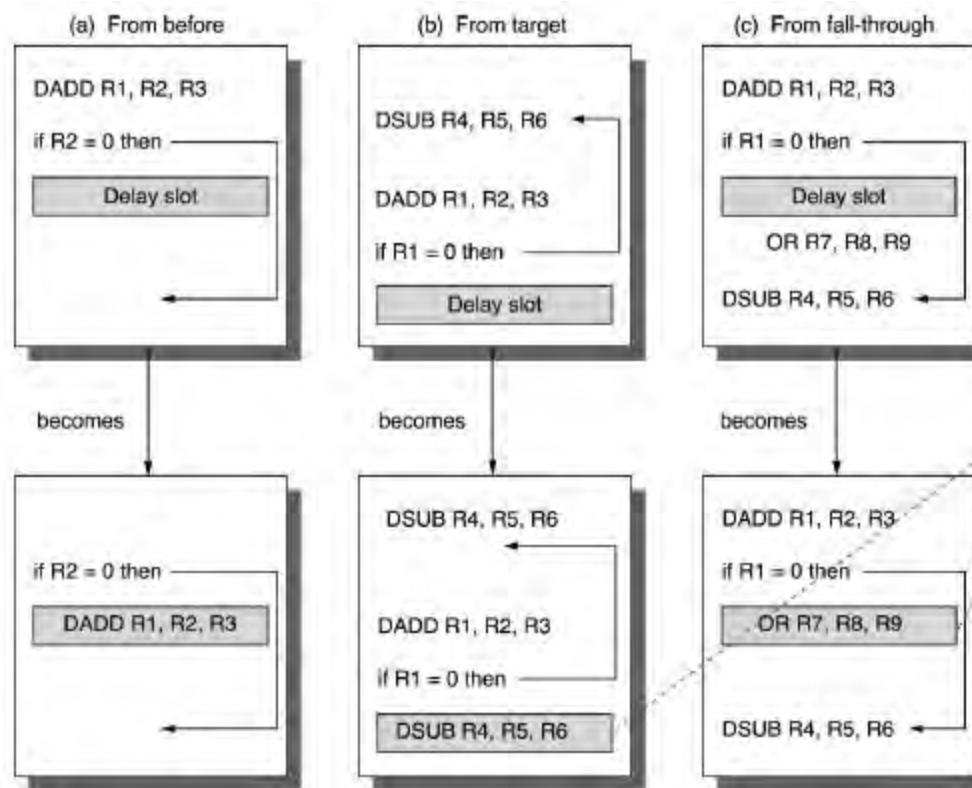
# Dealing with Branch Stalls

---

- Approach 4: **Delayed Branch**
  - Define branch to take place AFTER **n** following instructions
  - **Branch Delay Slots**
    - Instructions in the branch delay slot(s) get executed whether or not branch is taken
    - Heavily used in early RISC machines (e.g., MIPS)
    - Machines with deep pipelines require additional delay slots to avoid branch penalties

# Scheduling the Branch Delay Slot

- Where does the instruction for the delay slot come from?



Nullifying or  
canceling  
branches

- Converts delay slot instruction into a `nop`

# Importance of Avoiding Branch Stalls

---

- Crucial in modern microprocessors, which issue and execute multiple instructions every cycle
  - Need to have a steady stream of instructions to keep the hardware busy
  - Stalls due to control hazards dominate
- So far, we have looked at **static** schemes for reducing branch penalties

# Static vs. Dynamic

---

- **Static** schemes: same scheme applies to every branch instruction
  - **Dynamic** schemes: choose most appropriate scheme separately for each instruction
    - e.g., branches to top of loop // **Taken**
    - if (x == 0) return; // **Not Taken**
- Can "learn" appropriate scheme based on observed behavior

# Branch Predictor

---

- Two parts
  - Target Buffer: maps PC to taken target
  - Direction predictor: maps PC to taken/not taken
- What does it mean to "map PC" ?
  - Use some PC bits as index into an array of data items

# Mapping PCs

---

- If array of data has  $N$  entries
  - Need  $\log(N)$  bits to index it
- Which  $\log(N)$  bits to choose?
  - Least significant  $\log(N)$  after the least significant 2
  - Least significant change most often

# Mapping PCs

---

- What if two PCs have same pattern in that subset of bits (**aliasing**)
  - We get a nonsense target (intended for another PC)
  - That's OK, it's just a guess anyway, we can recover if it's wrong

# Branch Prediction (History) Buffer

---

- Small memory indexed by the low-order bits of the branch instruction
  - Stores **a single bit** of information: T or NT
  - Starts off as T, flips whenever a branch behaves opposite to prediction
  - Maintained by the **Fetch** stage
  - A correct prediction implies **no branch penalty**

# Branch Prediction (History) Buffer

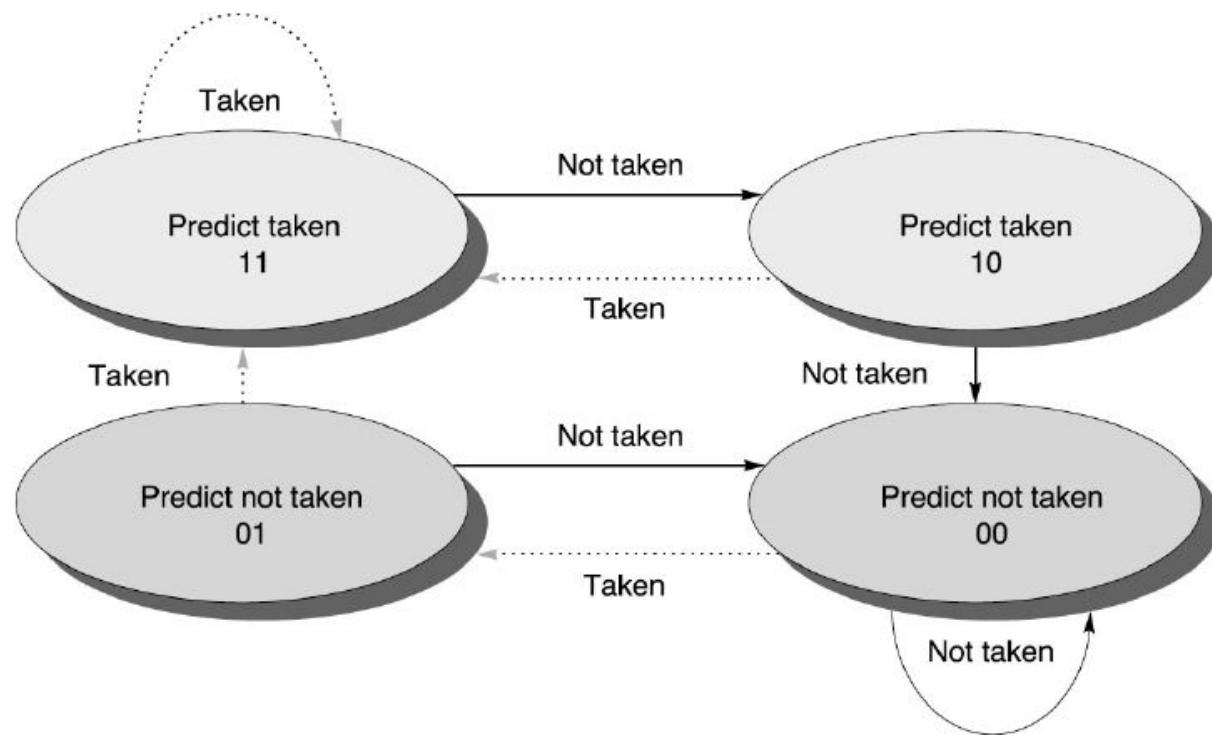
---

- Problems with this simple scheme
  - Does not do a good job of predicting “mostly-taken branches”
    - e.g., `for (i=0; i<10; i++) { ... }`
    - Repeated exec of the loop will result in **2** mispredictions
      - Last iteration flips T to NT
      - First iteration flips NT to T
    - Prediction accuracy of **80%**

# 2-bit Prediction Schemes

---

- Store 2 bits of info in branch history buffer
  - 1 misprediction per iteration if we start off in the (11) state



# Correlating Branch Predictors

---

- 2-bit predictor uses only the recent behavior of a single branch to predict its future behavior
- Is branch direction affected by more “global” properties?

```
if (aa == 2)          DSUBUI  R3, R1, #2
    aa = 0;
if (bb == 2)          BNEZ    R3, L1      ; branch b1
    bb = 0;
if (aa != bb)
{ ... }              DADD    R1, R0, R0
L1:                 DSUBUI  R3, R2, #2
                    BNEZ    R3, L2      ; branch b2
                    DADD    R2, R0, R0
L2:                 DSUBU   R3, R1, R2
                    BEQZ    R3, L3      ; branch b3
```

# Correlating Branch Predictors

---

```
if (aa == 2)          DSUBUI  R3, R1, #2
    aa = 0;
if (bb == 2)          DADD     R1, R0, R0
    bb = 0;
if (aa != bb)
{ ... }
L1:   DSUBUI  R3, R2, #2
      BNEZ    R3, L1      ; branch b1
      DADD     R2, R0, R0
L2:   DSUBU   R3, R1, R2
      BEQZ    R3, L3      ; branch b2
      BNEZ    R3, L2      ; branch b3
```

- Behavior of b3 is correlated with that of b1 and b2
  - if both b1 and b2 are NT, b3 will be T
- Can (how do) we predict such branches?

# (1,1) Correlating Predictor

---

```
if (d == 0)          BNEZ    R1, L1      ; branch b1
  d = 1;
if (d == 1)  L1:  DADDUI  R1, R0, #1
{ .. }           BNEZ    R3, L2      ; branch b2
...
L2:
```

	b1	b2
0	NT	NT
1	T	NT
2	T	T

- Extension: (m,n) Correlating Predictor
  - Use behavior of the last m branches to choose from among  $2^m$  n-bit predictors

# Thanks

---

# Loop Optimization

Oct. 2013

# Outline

- Why loop optimization
- Common techniques
- Blocking vs. Unrolling
- Blocking Example

# Why

- Loop: main bottleneck especially in Scientific Program(like image processing).
- Repeated execution: improve a little then Speedup a lot

# Definition

- Loop optimization can be viewed as the application of a sequence of specific loop transformations to the source code or intermediate representation
- Legality: result of the program should be preserved.

# Transformation Techniques

- According to Wikipedia, More than 10 items(just a list):
  - fission/distribution
  - fusion/combining
  - interchange/permutation
  - Inversion
  - loop-invariant code motion
  - Parallelization
  - Reversal
  - Scheduling
  - Skewing
  - software pipelining
  - splitting/peeling
  - Vectorization
  - Unswitching
- As to perflab, **Loop Unrolling** and **Loop Blocking** might be useful

# Loop Unrolling

- Mentioned a lot in class, which is also known as loop unwinding
- The overhead try to eliminate:
  - Pointer arithmetic
  - End of loop test

# Simple example

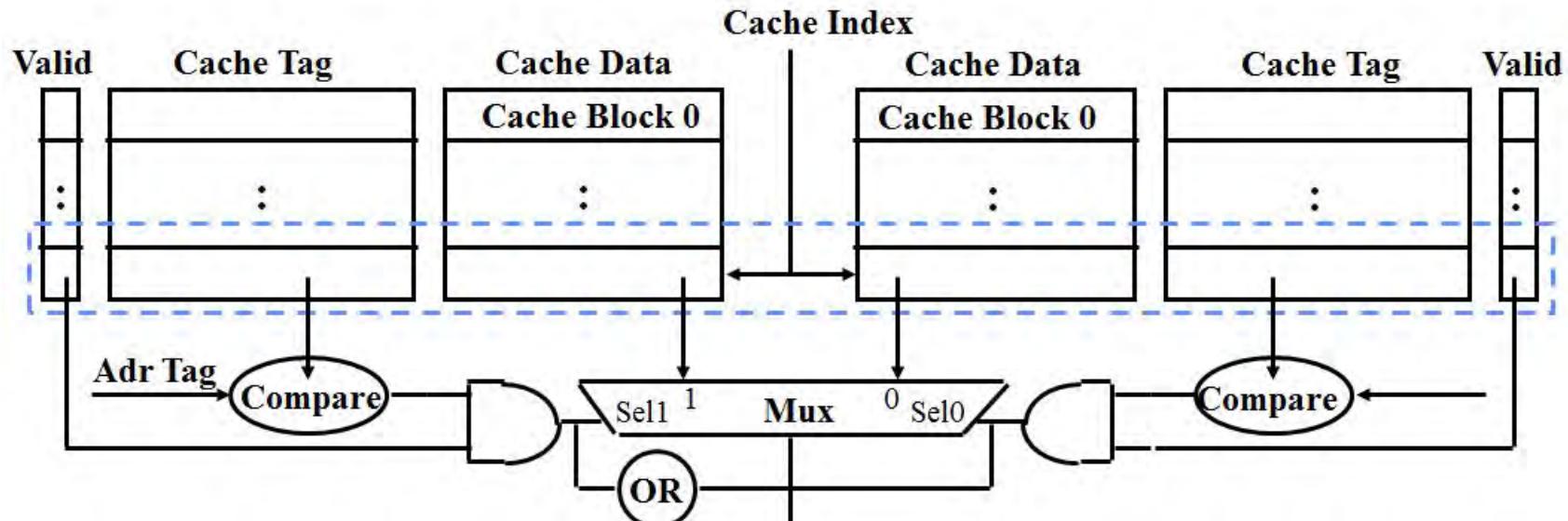
Normal loop	After loop unrolling
<pre>int x; for (x = 0; x &lt; 100; x++) {     delete(x); }</pre>	<pre>int x; for (x = 0; x &lt; 100; x+=5) {     delete(x);     delete(x+1);     delete(x+2);     delete(x+3);     delete(x+4); }</pre>

# Loop Blocking

- Also known as loop tiling.
- It partitions a loop's iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused
- Locality on loops

# Cache Brief Review

- Cache miss
- One Cache line each time
- Additionally: TLB miss



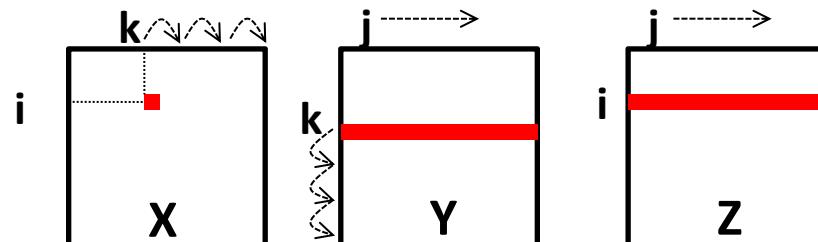
# Example

- Now We take Matrix multiplication as an example.
- You met it in Lecture “Cache Friendly Code” Last Semester
- This time, more specific.

- Two Matrix  $X * Y = Z$  each Matrix is  $N \times N$
- Look at the innermost loop. (Besides why register allocated first ? )

```

for(i = 0; i < N; i++)
    for(k = 0; k < N; K++)
        r = X(i,k); /*register allocated*/
        for(j = 0; j < N; j++)
            Z(i,j) += r*Y(k,j);
    
```



- If Cache is large enough ... everything will be fine!
  - Thanks to Prefetching, All Z and Y items will be reused.
- What if Cache can't hold one  $N \times N$  matrix?
  - Data Y would be replaced before reused
- What if Cache can't hold even a row in Z?
  - Z data in the cache can't be reused

- What is the worst case ?
- $2N^3 + N^2$  words of data need to be read from memory in  $N^3$  iterations

```
for(i = 0; i < N; i++)
    for(k = 0; k < N; K++)
        r = X(i,k); /*register allocated*/
        for(j = 0; j < N; j++)
            Z(i,j) += r*Y(k,j);
```

- Try blocking the matrix into small chunk, thus cache can hold that small chunk. Then loop locality will be back.

# Blocking code

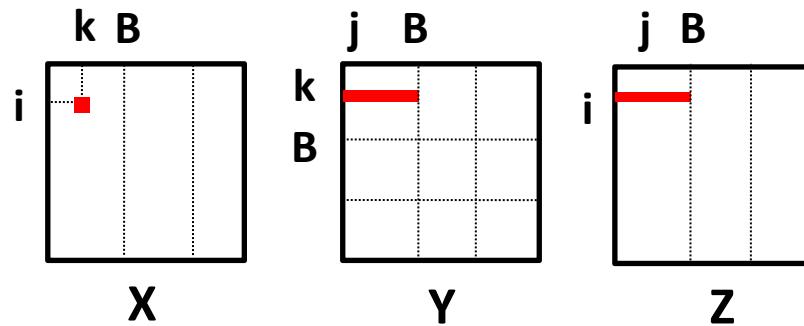
```
for(i = 0; i < N; i++)
    for(k = 0; k < N; K++)
        r = X(i,k); /*register allocated*/
        for(j = 0; j < N; j++)
            Z(i,j) += r*Y(k,j);
```

```
for(kk = 0; kk < N; KK+B)
    for(jj = 0; jj < N; jj+B)
        for(i = 0; i < N; i++)
            for(k = kk; k < min(kk+B-1,N); k++)
                r = X(i,k);
                for(j = jj; j < min(jj+B-1,N); j++)
                    Z(i,j) += r*Y(k,j);
```

```

for(kk = 0; kk < N; KK+B)
    for(jj = 0; jj < N; jj+B)
        for(i = 0; i < N; i++)
            for(k = kk; k < min(kk+B-1,N); k++)
                r = X(i,k);
                for(j = jj; j < min(jj+B-1,N); j++)
                    Z(i,j) += r*Y(k,j);

```



- $B \ll N$  (less than)
- $B$  is called blocking factor
- $B \times B$  submatrix of  $Y$  and a row of length  $B$  of  $Z$  can fit in cache. Thus called  $B \times B$  Blocking

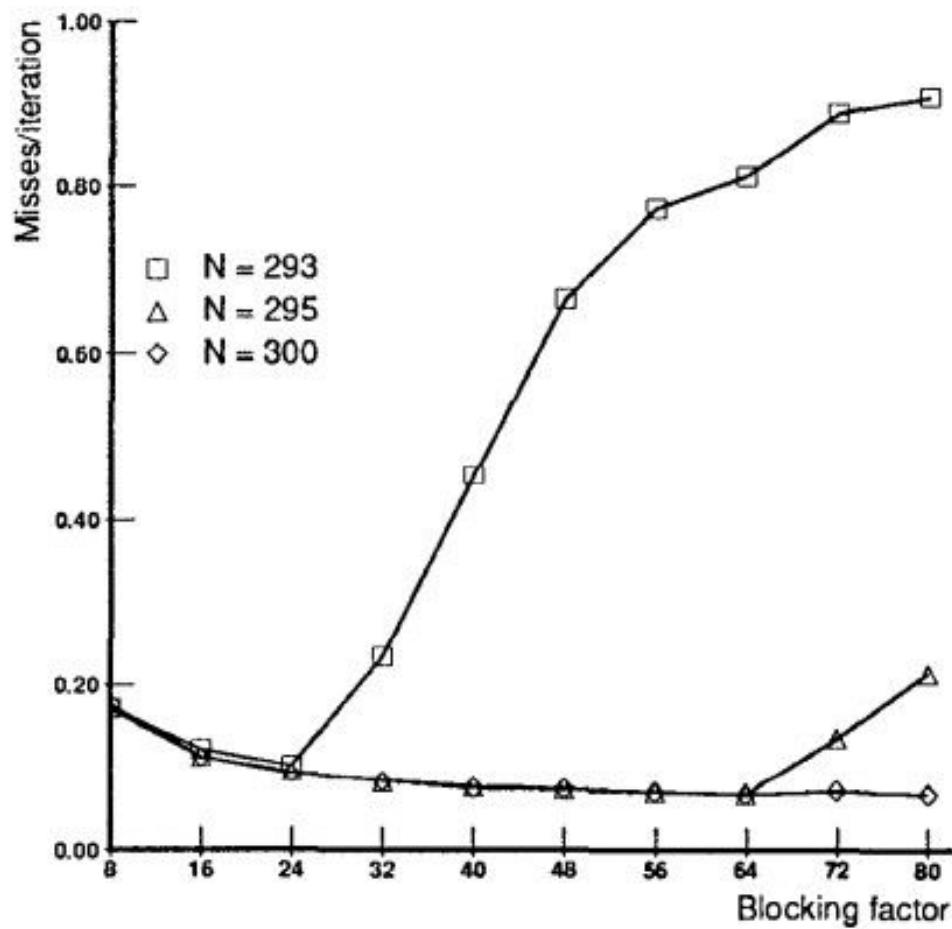
```

for(kk = 0; kk < N; KK+B)
    for(jj = 0; jj < N; jj+B)
        for(i = 0; i < N; i++)
            for(k = kk; k < min(kk+B-1,N); k++)
                r = X(i,k);
                for(j = jj; j < min(jj+B-1,N); j++)
                    Z(i,j) += r*Y(k,j);
    
```

- Thus  $2N^3 / B + N^2$  words accessed in main memory.
- Larger B, Larger performance gain?
  - Choose an appropriate Blocking factor, so that cache is fully occupied by data to be reused

- Yes, but not always.
- As to fully associative cache with LRU policy, it's right.
  - Cache fully used.
- In practice, caches are direct mapped or have at most a small degree of set associativity.
  - Map multiple rows of a matrix to the same cache line, making it infeasible to try to fully use cache.

- moreover, varies drastically with matrix size



# Lab note

- Hybrid method may be helpful
  - Loop unrolling and blocking together.
  - Function call can be inlined.
  - Other methods in previous wiki list might be helpful if you want higher and higher performance.  
( not suggested)

# Some facts

- Indeed, all these optimization jobs Can be done by Compiler. e.g. gcc flag LNO will do Loop Nested Optimization Job for you.
- Complex Algorithm used
- Manually code optimization sometimes do a better job ( you are smarter than the nerd compiler ^\_^)

- Loop Unrolling and Blocking will be involved in Midterm exam.
- Thanks!!!