

Computer Systems Organization

CS2.201

Lecture 1

Based on chapter 1 from Computer Systems by
Randal E. Bryant and David R. O'Hallaron

Introduction

- Welcome to CSO Class
- Text Book:
- Computer Systems A Programmers Perspective (Third Edition) by Randal E. Bryant and David R. O'Hallaron
- Please prepare notes during lectures – slides may not all be accessible

A Rough Grading Structure (some changes)

- Assignments: 20%
- Quizzes: 10-15%
 - Surprise quizzes are a possibility if need arises
- Mid Exam: 20%
- Final Exam: 30%
- Lab exam: 10-12%
- Notes, Attendance, Participation: 5-10% (TBD)
- Cheating, Attendance and Ethics related issues can incur 100% penalty

Overview of topics we may cover

- Introduction (1)
- Computer Arithmetic (2.1 – 2.3)
- Assembly language programming (3.4 – 3.7)
- Processor architecture and design (4.1 – 4.5)
- Memory Hierarchy (6.1 - 6.4)
- Optional:
 - System calls Intro to process control (8.2 – 8.4)
 - Virtual memory high level overview

Basic Course Goal

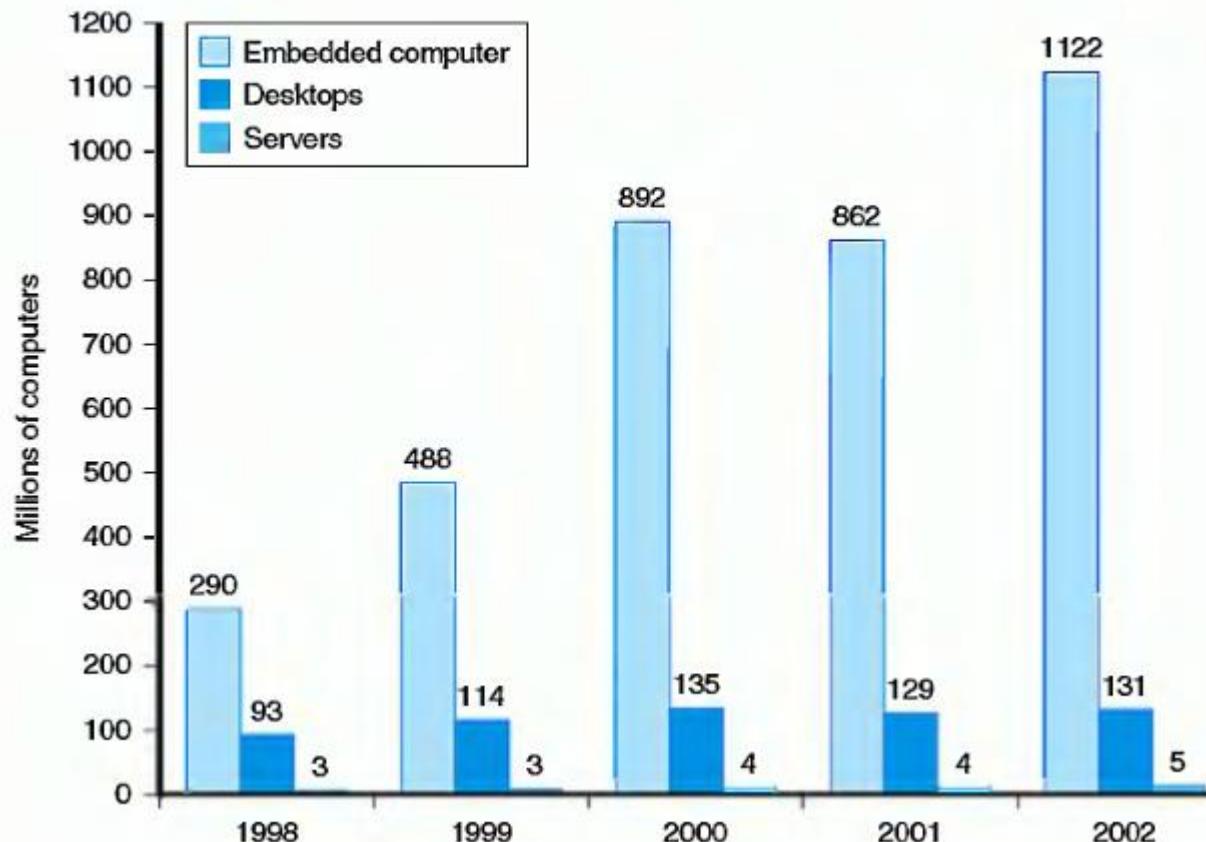
- **Course Goal:** To study the anatomy of a typical Computer System.
- Well, what is a typical computer system?
 - Desktops, Laptops, Notebooks
- How about Server Machines? In what way they are different from Desktops/Laptops?
 - Desktops run a user-friendly **OS** and desktop applications to facilitate desktop-oriented tasks.
 - Server manages all network resources and are often dedicated (performs no other task besides server tasks).
Servers are engineered to manage, store, send and process data 24-hours a day, has to be more reliable than a desktop computer and offers a variety of features/hardware.

Basic Course Goal

- Operating System (OS): OS is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers. It is an interface between a computer user and computer hardware.
- How about Embedded Computers lying inside Cell Phones, Automobiles, Airplanes, Set Top Boxes, Televisions etc. ?
 - A general purpose computer system can be programmed to perform a large number of tasks. Embedded systems are designed to perform a small number of tasks efficiently.

Sales Distribution

Source: H&P-3 (Hennesy & Patterson, 3rd Edition)



Number of distinct processors sold between 1998 and 2002.

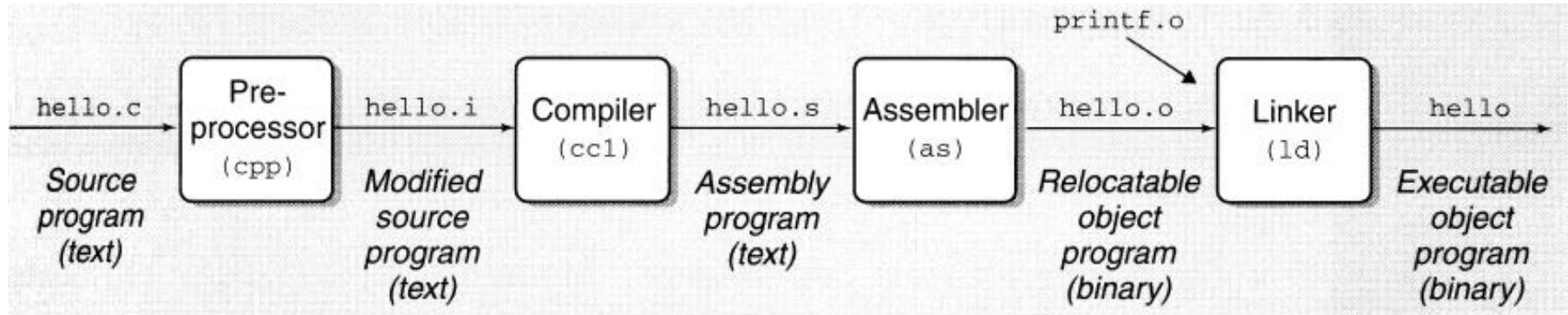
The hello world program

- Source program or source file
- Organized as 8 bit chunks called bytes
- ASCII (American Standard Code for Information Interchange) representation below
 - Files that exclusively containing ASCII chars are called text files - others called binary files

```
# i n c l u d e <sp> < > s t d :: o p e n ( " h e l l o . w o r l d " , " w " ) ; o u t p u t < > " H E L L O , W O R L D ! " ; o u t p u t < > " \ n " ; c l o s e ( " h e l l o . w o r l d " ) ;
```

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	()	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	("	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	")	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

Typical Compilation Sequence



- Steps during compilation of the program
 - Preprocessing phase
 - Compilation phase
 - Assembly phase
 - Linking phase

Typical Compilation Sequence

- Preprocessing phase: Modifies original C program according to directives of # character
 - Read contents of stdio.h and insert into program text. Results in another program with .i affix
- Compilation phase: Translates .i into .s containing assembly language program
 - Useful since it provides a common output language for different compilers for different high level languages e.g., C and Fortran compilers generate files in same assembly language
- Assembly phase: Translates .s into machine language instructions – packages into relocatable object program and outputs .o.

Typical Compilation Sequence

- Linking Phase: printf function resides in a separate precompiled object file called print.o which needs to be merged into hello.o. Linker handles the merging and creates hello [an executable file ready to be loaded into memory and executed by the system]
- Understanding compilation systems helps with
 - Optimizing program performance e.g., switch vs. if-else, overhead incurred by function call etc.
 - Understanding link-time errors which are hard to catch e.g., linker says cannot resolve a reference, difference between static and global variable ...
 - Avoiding security holes e.g., buffer overflow vulnerabilities ...

Buffer Overflow Vulnerability

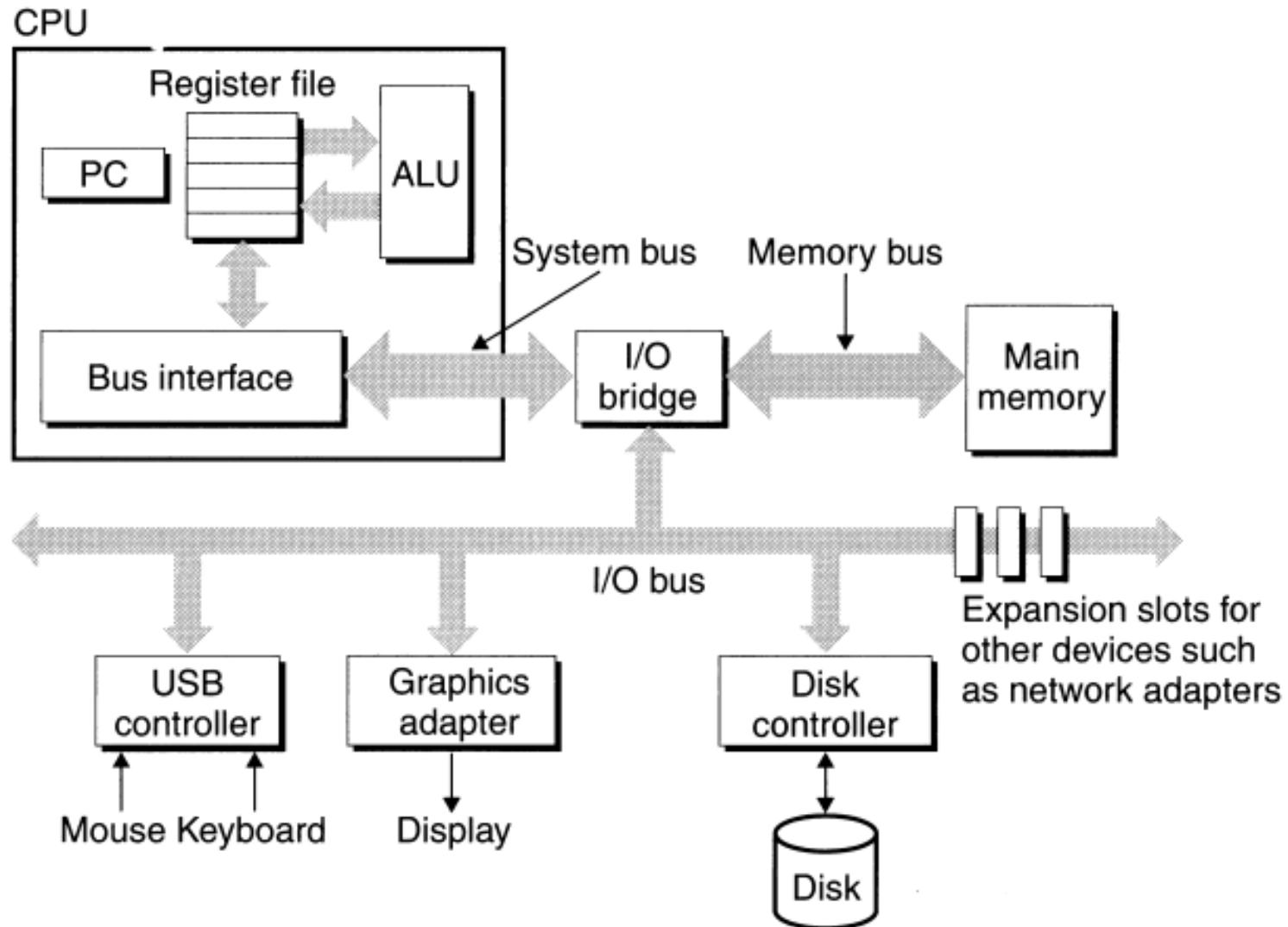
- Better understanding helps to avoid issues
- Code below uses `gets()` function to read an arbitrary amount of data into a stack buffer. Because there is no way to limit the amount of data read by this function, safety of the code depends on the user to always enter fewer than `BUFSIZE` characters.

...

```
char buf[BUFSIZE];
gets(buf);
```

...

Typical Hardware Organization of a System



Source: RB&DO -1 (Randal E. Bryant & David O'Hallaron, 1st Ed)

Hardware Organization of System

- Buses: Collection of electric conduits that carry bytes of information between components. Carry fixed size chunks of bytes called words (called word size) e.g., 4 bytes or 8 bytes word
- I/O devices: Systems connection to the external world e.g., keyboard, mouse, display and disk drive/disk
 - I/O devices connected to bus by either a controller (chipset in the device itself or on motherboard) or adapter (card that plugs into a slot on motherboard)

Hardware Organization of System

- Processor repeatedly executes instruction pointed by the program counter and updates the PC to point to next instruction (which may not be in next memory address)
- Processor operates according to a very simple instruction execution model, defined by its instruction set architecture
- Few simple operations that revolve around main memory, the register file and the arithmetic/logic unit (ALU).
- Register file is a small storage device that consist of a collection of word size registers each with its own unique name (Accumulator, Memory Address Register, Memory Data Register ... for fast retrieval of data)
- ALU computes new data and address values

Hardware Organization of System

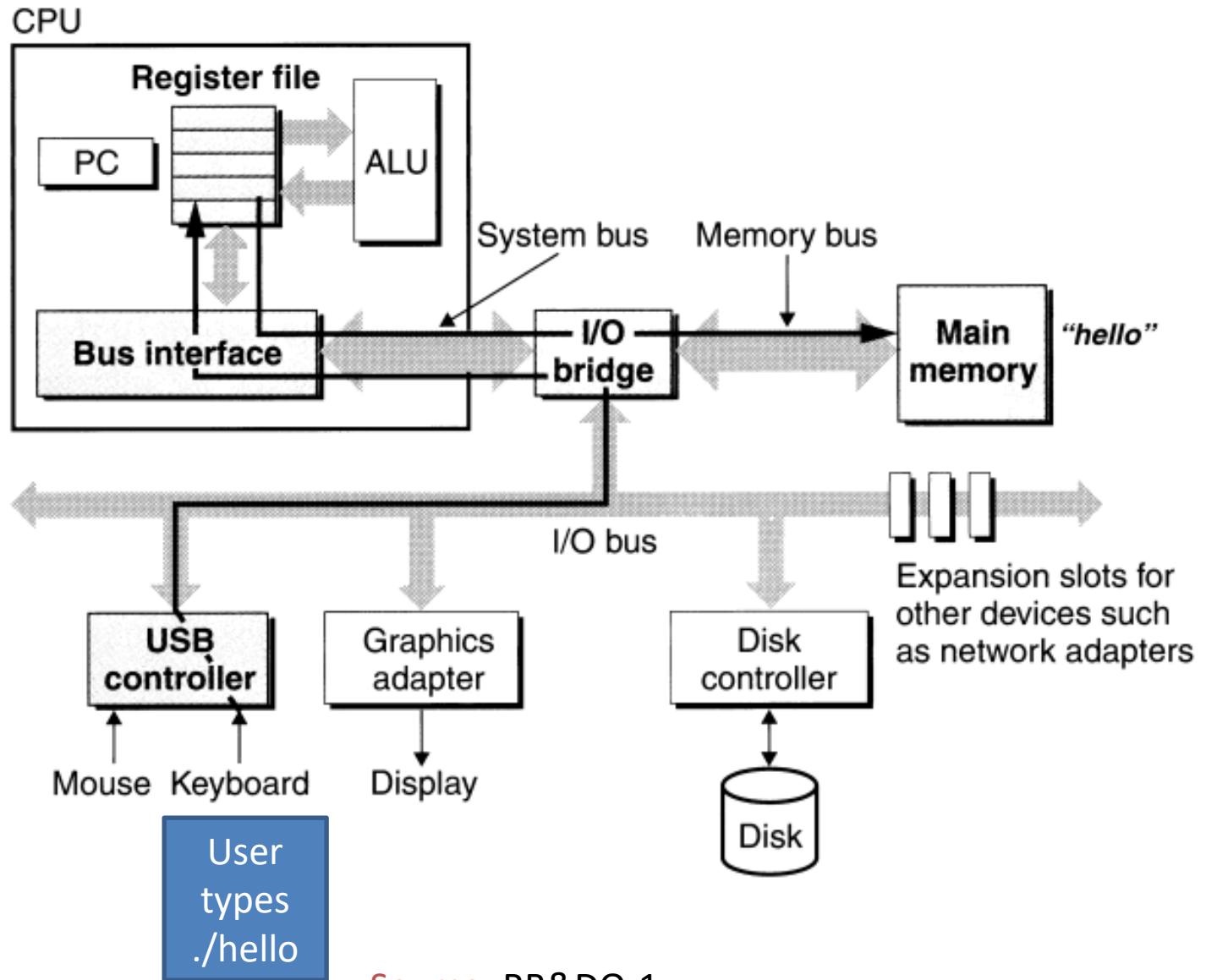
- Main memory: Temporary storage device that holds both a program and data it manipulates
 - Collection of DRAM (Dynamic Random Access Memory) chips
 - Memory organized as linear array of bytes, each with its own unique address starting at 0
- Processor: CPU is the engine that interprets (or executes) instructions stored in main memory. At its core is a word size storage device (or register) called the program counter (PC).
 - At any point, PC points to or contains address of some machine language instruction in main memory

Hardware Organization of System

– Some operations carried out by CPU:

- Load: Copy a byte or a word from main memory into a register, overwriting previous contents of the register
- Store: Copy a byte or word from a register to a location in main memory, overwriting the previous contents of that location
- Operate: Copy the contents of two registers to the ALU, perform an arithmetic operation on the two words and store the result in a register, overwriting its previous contents
- Jump: Extract a word from the instruction set itself and copy the word into the PC, overwriting the previous value of the PC

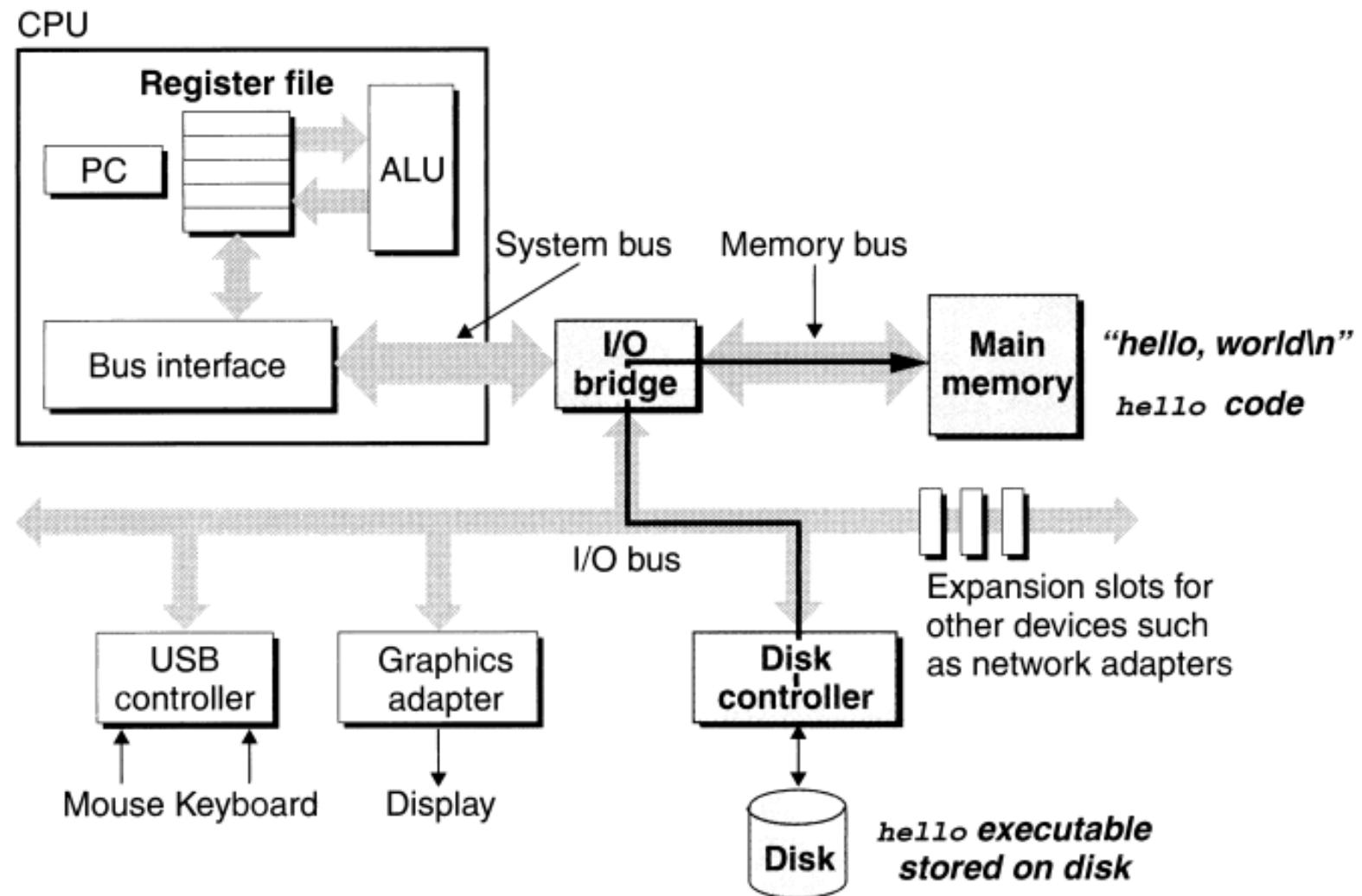
Running the “Hello World” Program



Sequence of steps

- Shell program waiting for inputs
- As we type the characters ./hello, shell program reads each one into a register and then stores it in memory. Once enter key is hit, shell knows we have finished typing.

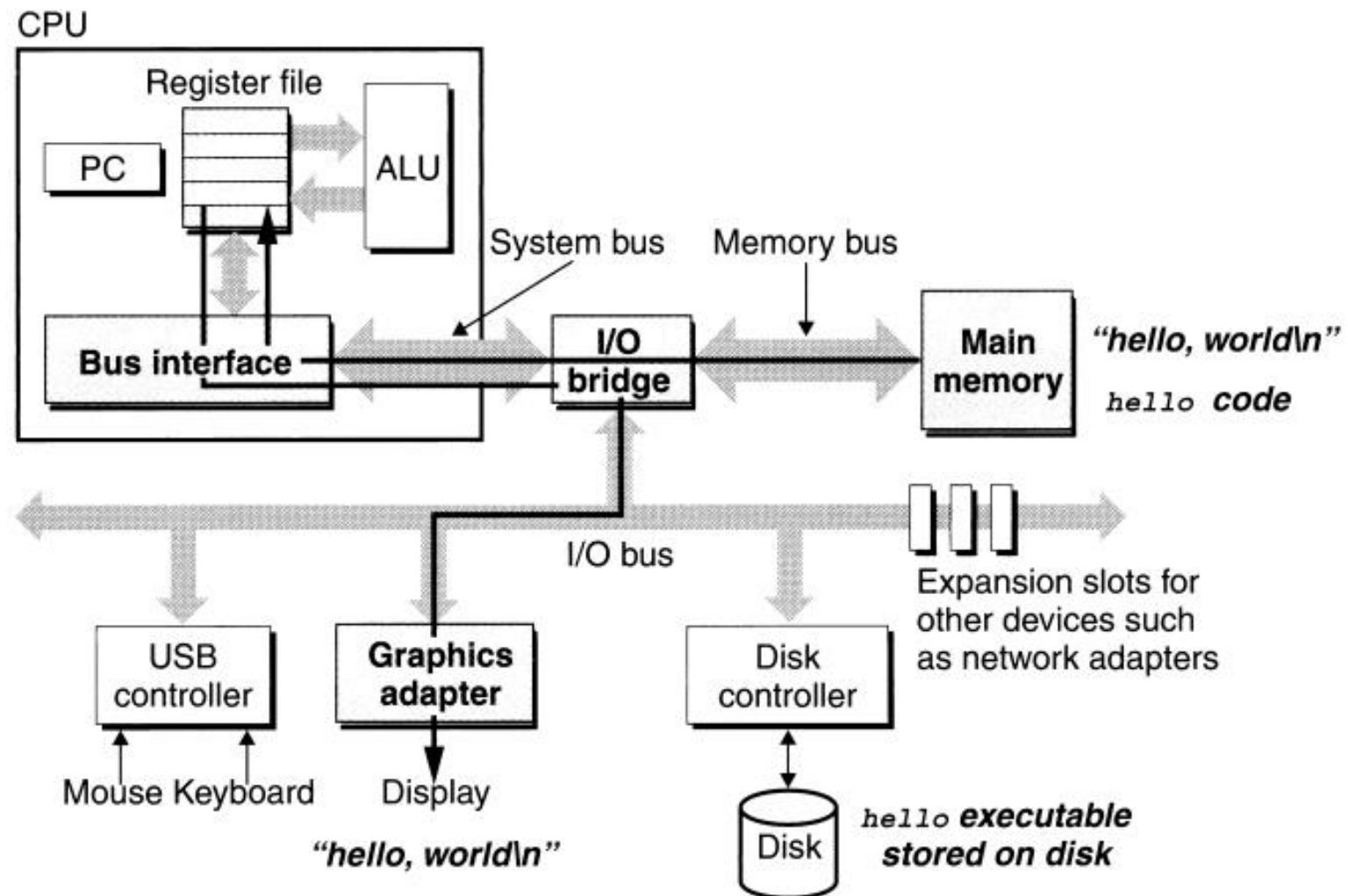
Running the “Hello World” Program



Sequence of steps

- Shell then loads the executable hello file by executing a sequence of instructions that copies the code and data in hello object file from disk to main memory

Running the “Hello World” Program



Sequence of steps

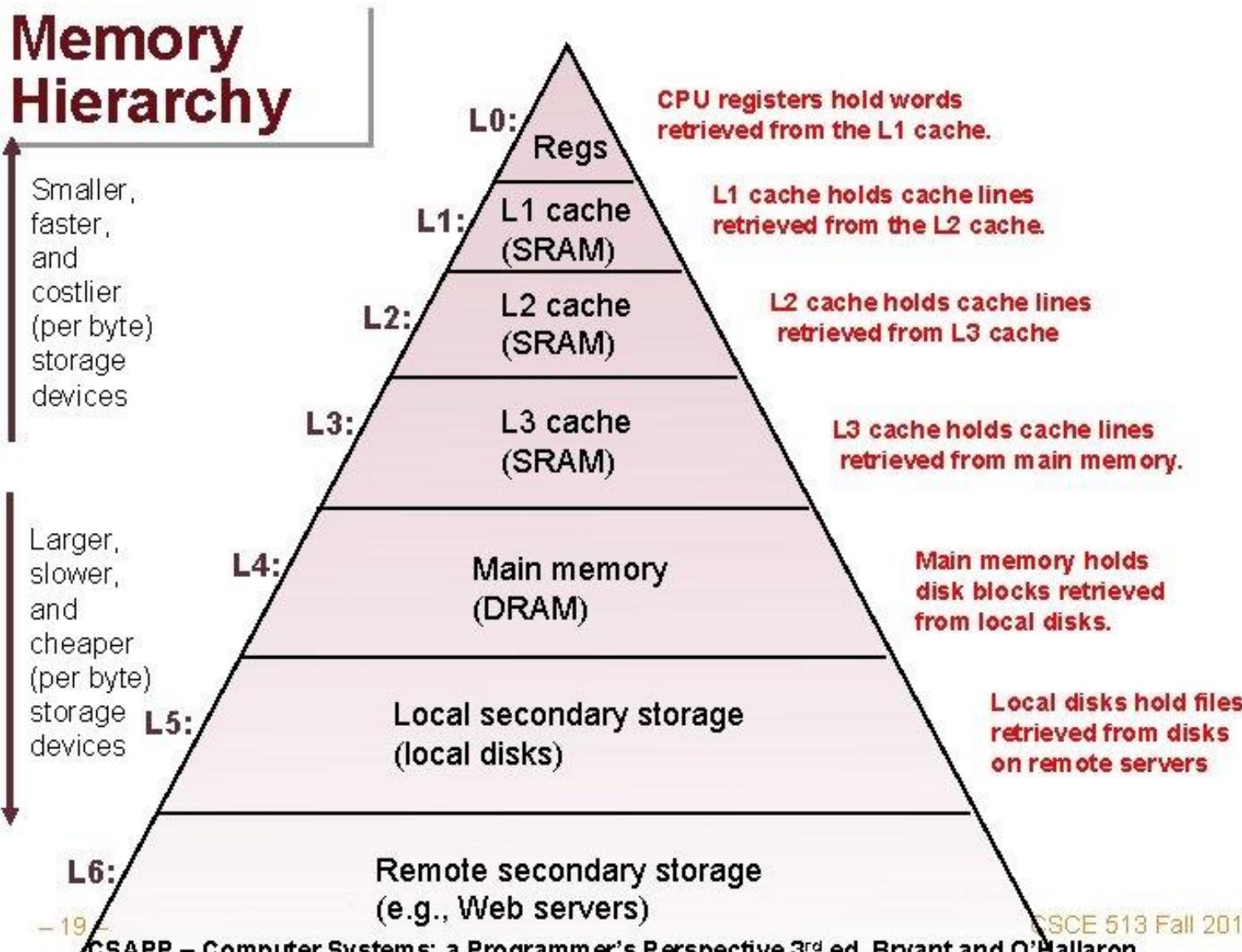
- Once loaded in memory, CPU begins executing the machine language instructions in hello program's main routine. These instructions copy the bytes in the hello, world\n string from memory to the register file and from there to the display device.

Cache Memory

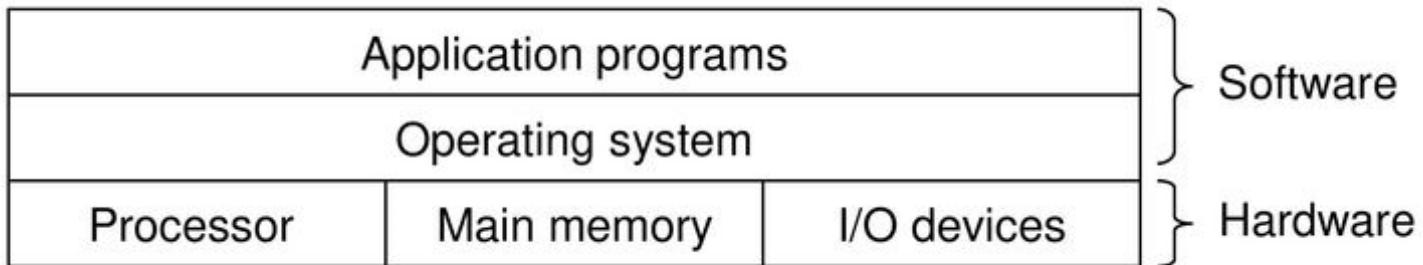
- Disk drive is typically 1000 times larger than main memory but it takes processor 10,000,000 times longer to read a word from disk than from memory
- Similarly, register file stores only a few 100 bytes of information - however processor can read data 100 times faster than from memory
- As semiconductor technology advances, the processor-memory gap continues to increase, hence cache memories become important
- Application programmers can exploit cache memories to improve performance by an order of magnitude

Memory hierarchy

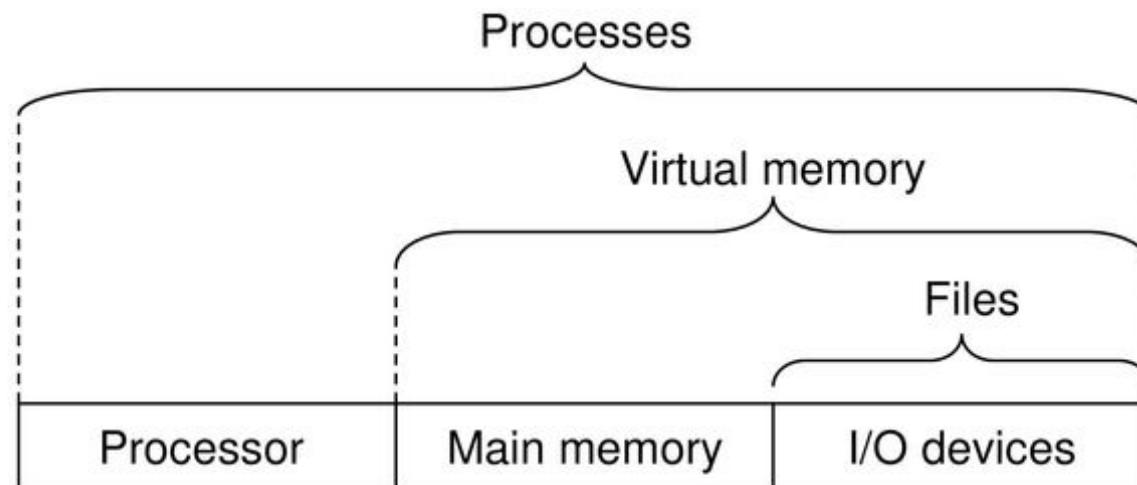
Memory Hierarchy



OS



Layered view of a computer system



Abstractions provided by an OS

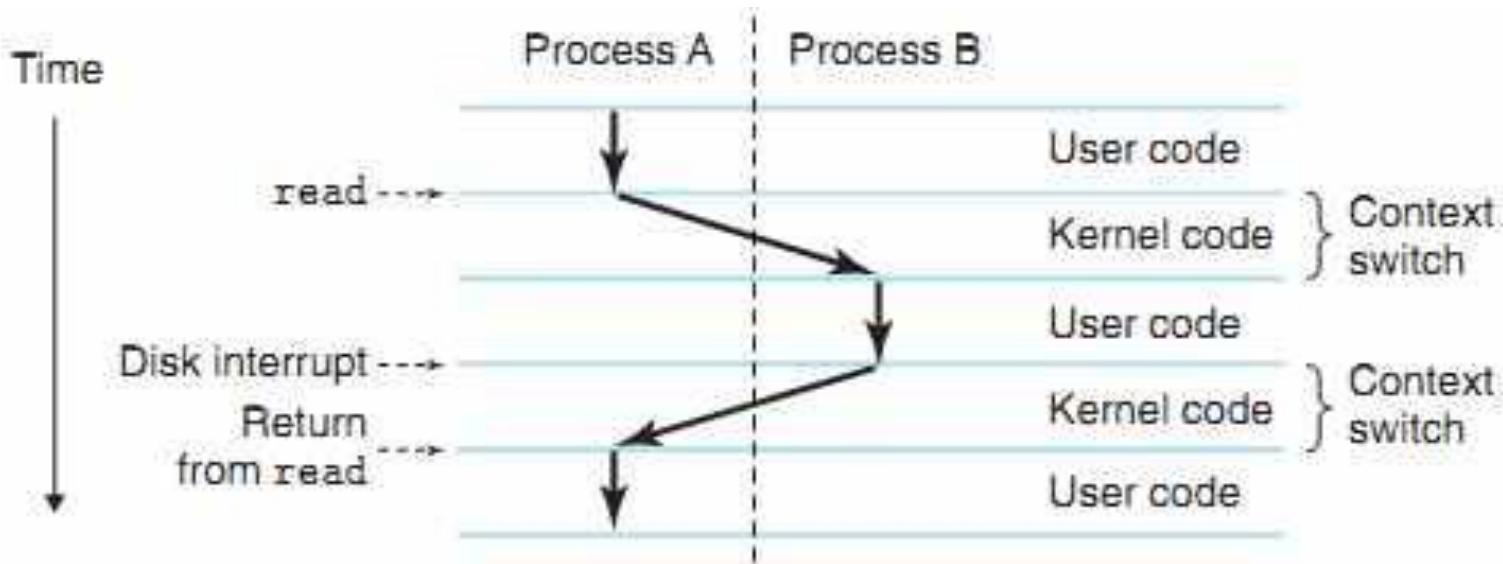
OS

- Layer interposed between application program and the h/w. 2 primary purposes:
 - Protect h/w from misuse by any application
 - Provide applications with simple and uniform mechanisms for manipulating complicated and wildly different low level h/w devices
- Achieves the goals via fundamental abstractions namely processes, virtual memory and files. Each are abstractions for:
 - Files for I/O devices
 - Virtual memory for main memory and disk I/O devices
 - Processes for the processor, main memory, I/O devices

Processes

- Process is OS's abstraction for a running program
- Provides illusion that it is the only program running on the system
- Multiple processes can run concurrently i.e., instruction of one process can be interleaved with another process – achieved via context switching
- State information of process is called context (includes values of the PC, register file and contents in main memory).
- OS performs context switch by saving the context of current process, restoring the context of new process and then passing control to the new process.

Processes



- When hello is run, invokes special function called system call to pass control to OS
- OS saves shell's context, creates a new hello process
- Transition between processes is handled by OS Kernel (portion of OS code always resident in memory)

Threads

- A process can consist of multiple execution units called threads (run in context of the process and share the same code and global data)
- Can allow concurrency in processes, easier to share data and more efficient than multiple processes
- Threads provide a way to improve application performance (e.g., network or web servers) through parallelism and represent a software approach to improving performance

Virtual Memory

- Virtual memory provides each process with illusion that it has exclusive use of main memory
- Each process has same uniform view of memory called virtual address space
- Virtual address space seen by each process consists of a number of well defined areas each with specific purposes
 - Program code and data, Heap, Shared libraries, Stack etc.

Files

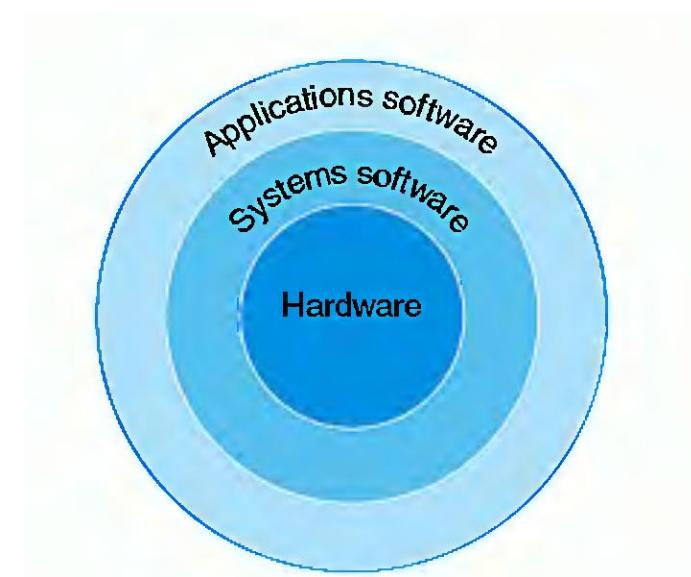
- Sequence of bytes
- Every I/O device incl. disks, keyboards, displays and networks modeled as a file
- All I/O in system performed by reading/writing files
- Provides a uniform view of the varied devices
 - Will therefore run on different systems

Networking related

- Modern systems linked to other systems by networks
- Network can be viewed as another I/O device from/to which data is copied/sent e.g., email, messaging, World Wide Web, ...

Computer System = Hardware + System Software + Application Software

Source: H&P-3 (Hennesy & Patterson, 3rd Edition)



System Software: Operating System, Device Drivers, Loaders, Linkers, Compilers, Assemblers,

Application Software: Web browsers, user-specific applications,

Amdahl's law

- Captures effectiveness of improving one part of the system – when we speedup one part of the system effect on overall system depends on how significant this part was
- Suppose an application needs time T_{old} . One part needs α of this time (i.e., $\alpha * T_{\text{old}}$) and we improve performance by factor k .
- $T_{\text{new}} = (1 - \alpha) * T_{\text{old}} + (\alpha * T_{\text{old}})/k$
- Speedup $s = T_{\text{old}}/T_{\text{new}} = 1/((1 - \alpha) + \alpha/k)$
- If $\alpha = .6$ and we sped up the component 3 times, $s = 1/((1 - .6) + .6/3) = 1.67$
- Insight: Need to speed up large fraction of system

Other details

- Concurrency is concept of system with multiple, simultaneous activities and parallelism refers to use of concurrency to make a system run faster
 - Used/implemented in different ways for significant performance improvements e.g., multithreading, multiprocessing etc.
- Abstraction is one of the important concept in computer science
 - Different programming languages provide different levels of abstraction and support
 - Instruction set architecture provides abstraction of actual processor hardware [machine code behaves as if it were executed on a processor that performs just one instruction at a time while underlying h/w is far more elaborate, executing multiple instructions in parallel]

Other details

- On OS side, files are an abstraction for I/O devices, virtual memory for program memory and processes as abstraction for running program.
- A virtual machine provides abstraction of an entire computer including the OS, processor and the programs.

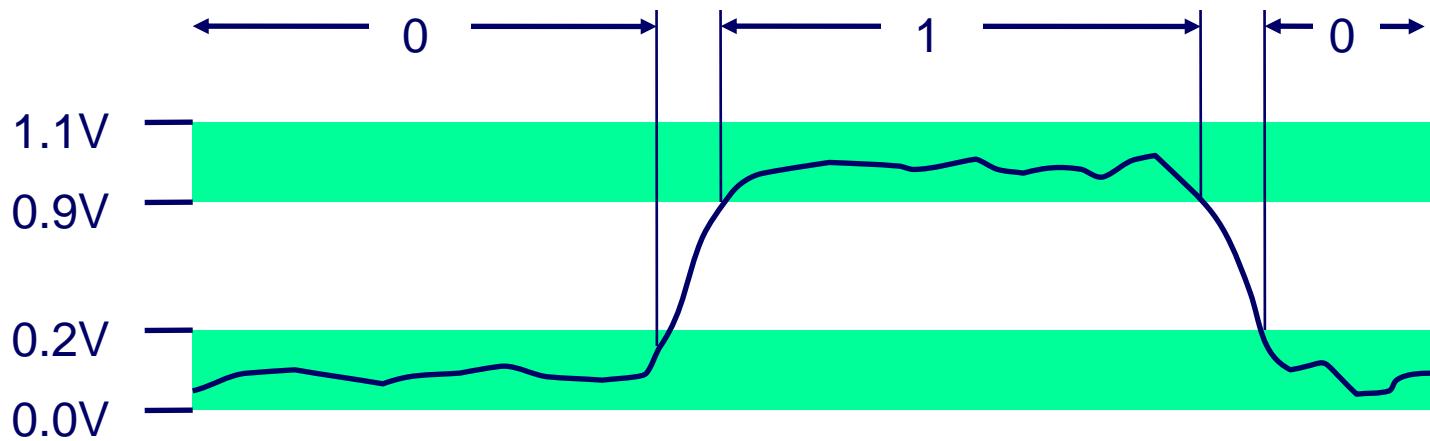
Computer Systems Organization

Topic 2

Based on chapter 2 from Computer Systems
by Randal E. Bryant and David R. O'Hallaron

Everything is bits

- Each bit is 0 or 1 (binary digits)
- Form basis of digital revolution
- Why bits? Electronic Implementation
 - Storing/performing computations is simple/reliable
 - Easy to store with bistable elements
 - Reliably transmitted on noisy and inaccurate wires



Everything is bits

- Single bit may not be useful but bit patterns do (groups of bits)
- 3 important representations of numbers
 - Unsigned encodings based on binary notation
 - Two's complement encoding to represent signed integers
 - Floating point encoding are base-2 version for representing real numbers
- Limited number of bits to encode numbers can have surprising effects e.g., $200 * 300 * 400 * 500$ yields -884901888 (32 bit representation)

Binary representation

- Base 2 Number Representation
 - Represent 15213_{10} as 11101101101101_2
 - Represent 1.20_{10} as
 $1.0011001100110011[0011]\dots_2$
 - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

How to convert

- $11 = (1011)_2 = 2^3 * 1 + 2^2 * 0 + 2^1 * 1 + 2^0 * 1$
 - $11/2 = 5 \text{ (1)}$
 - $5/2 = 2 \text{ (1)}$
 - $2/2 = 1 \text{ (0)}$
 - $1/2 = 0 \text{ (1)}$
-
- $12 = (1100)_2 = 2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 0$
 - $12/2 = 6(0)$
 - $6/2 = 3(0)$
 - $3/2 = 1(1)$
 - $1/2 = 0(1)$

How to convert (fraction)

- Convert 0.8125
- $.8125 * 2 = 1.6250$ (1)
- $.6250 * 2 = 1.250$ (1)
- $.250 * 2 = 0.5$ (0)
- $.5 * 2 = 1.0$ (1)
- $0 * 2 = 0$ (0)
- Soln: 0.11010
- Converting back: $1 * 2^{-1} + 1 * 2^{-2} + 0 + 1 * 2^{-3} + 0 = .5 + .25 + 0 + .0625 + 0 = 0.8125$

Encoding Byte Values

- Byte = 8 bits
 - Binary 00000000_2 to 11111111_2
 - Decimal: 0_{10} to 255_{10}
 - Hexadecimal 00_{16} to FF_{16}
 - Base 16 number representation
 - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
 - Write $FA1D37B_{16}$ in C as
 - `0xFA1D37B`
 - `0xfa1d37b`

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Hex Notation

- $314156 = 19634.16 + 12$
- $19634 = 1227.16 + 2$
- $1227 = 76.16 + 11$
- $76 = 4.16 + 12$
- $4 = 0.16 + 4$
- Hence 0x4CB2C
- What is binary and hex notation for 158 ?
- What is $0x605c + 0x5$?
- Please practice...

Data sizes

- Each computer has word size indicating nominal size of pointer data
- Virtual address is encoded by such a word, hence word size determines size of virtual address space
- For machine with w -bit word size, virtual address can range from 0 to $(2^w)-1$, giving program access to at most 2^w bytes
- 32 bit word size limits virtual address space to 4GB [$4 * 10^9$ bytes] while 64 bit leads to 16 exabytes i.e., $1.84 * 10^{19}$ bytes
- Most 64-bit machines can run programs compiled to use for 32-bit machines i.e., backward compatible
- 32 bit vs. 64 bit programs [rather than machine distinction lies in how the program is compiled]

Typical Data Representations in C

C Data Type	Typical 32-bit	Typical 64-bit
char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
long double	–	–
pointer	4	8

Most data types encode signed values unless prefixed by `unsigned`

Exception for `char` – need to declare `signed char`

Addressing and Byte Ordering

- In almost all machines, a multi-byte object stored as a contiguous sequence of bytes
 - E.g., variable x of type int has address 0x100 i.e., address expression &x is 0x100. Assuming int is 4 bytes, x would be stored in location 0x100, 0x101, 0x102 and 0x103.
- Two notations – big endian vs. little endian
- Number 0x01234567 stored as {01}{23}{45}{67} in big endian notation while stored as {67}{45}{23}{01} in little endian notation for the addresses 0x100, 0x101, 0x102, 0x103.
- Most intel compatible machines are little endian while most IBM/Oracle machines are big endian

Representing code

- Consider the following C function:

```
int sum(int x, int y) {  
    Return x + y;  
}
```

- Following machine code generated when compiled
- Linux 32: 55 89 e5 8b 45 0c 03 45 08 c9 c3
- Windows: 55 89 e5 8b 45 0c 03 45 08 5d c3
- Instruction codings can be different – binary code is seldom portable across combinations of machines + OS
- From machine perspective – program is simply a sequence of bytes and has no/minimal information of original source program

Boolean Algebra

- Developed by George Boole in 19th Century
 - Algebraic representation of logic
 - Encode “True” as 1 and “False” as 0

And

■ $A \& B = 1$ when both $A=1$ and $B=1$

&	0	1
0	0	0
1	0	1

Or

■ $A | B = 1$ when either $A=1$ or $B=1$

	0	1
0	0	1
1	1	1

Not

■ $\sim A = 1$ when $A=0$

\sim	
0	1
1	0

Exclusive-Or (Xor)

■ $A ^ B = 1$ when either $A=1$ or $B=1$, but not both

\wedge	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on Bit Vectors (strings of 0's and 1's of fixed length w) - operations applied bitwise

$$\begin{array}{rcl} \begin{array}{c} 01101001 \\ \& 01010101 \end{array} & \begin{array}{c} 01101001 \\ | 01010101 \end{array} & \begin{array}{c} 01101001 \\ ^ 01010101 \end{array} \\ \hline \begin{array}{c} 01000001 \\ 01111101 \end{array} & \begin{array}{c} 01111101 \\ 00111100 \end{array} & \begin{array}{c} 10101010 \\ 00111100 \end{array} \end{array}$$

- All of the Properties of Boolean Algebra Apply
- Let a and b denote bit vectors $[a_{w-1}, a_{w-2}, \dots, a_0]$ and $[b_{w-1}, b_{w-2}, \dots, b_0]$.
- $a \& b$ would be a bit vector of length w , where the i th element would be $a_i \& b_i$

Representing & Manipulating Sets

- Representation
 - Can encode any subset $\{0, \dots, w-1\}$ with a bit vector $[a_{w-1}, a_{w-2}, \dots, a_0]$
 - represents subsets of Width w
 - $a_j = 1$ if $j \in A$
 - 01101001 $\{0, 3, 5, 6\}$
 - **76543210**
 - 01010101 $\{0, 2, 4, 6\}$
 - **76543210**
- Operations
 - & Intersection 01000001 $\{0, 6\}$
 - | Union 01111101 $\{0, 2, 3, 4, 5, 6\}$
 - ^ Symmetric difference 00111100 $\{2, 3, 4, 5\}$
 - ~ Complement 10101010 $\{1, 3, 5, 7\}$

Bit-Level Operations in C

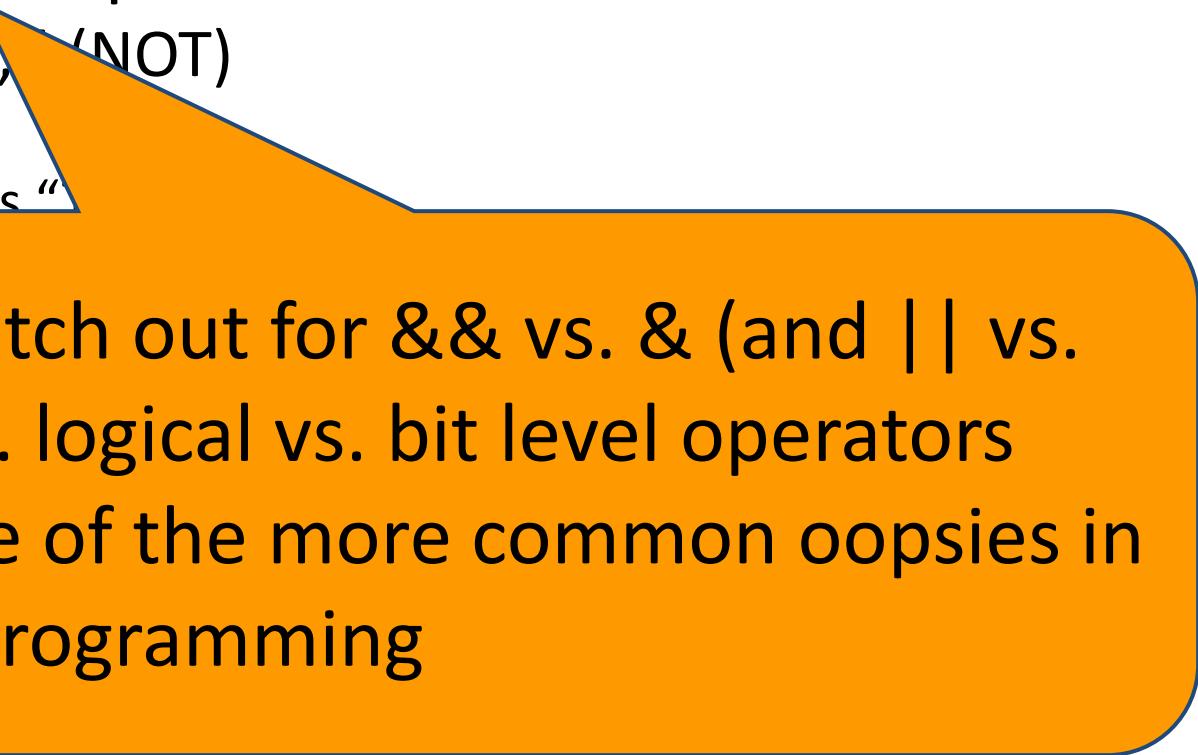
- Operations & (AND), | (OR), ~ (NOT), ^ (Exclusive-OR) Available in C
 - Apply to any “integral” data type
 - long, int, short, char, unsigned
 - View arguments as bit vectors
- Examples (Char data type)
 - ~0x41 is 0xBE
 - ~01000001₂ is 10111110₂
 - ~0x00 is 0xFF
 - ~00000000₂ is 11111111₂
 - 0x69 & 0x55 is 0x41
 - 01101001₂ & 01010101₂ is 01000001₂
 - 0x69 | 0x55 is 0x7D
 - 01101001₂ | 01010101₂ is 01111101₂

Contrast: Logic Operations in C

- Contrast to Logical Operators

- `&&` (AND), `||` (OR), `!` (NOT)

- View 0 as “False”
 - Anything nonzero as “True”
 - Always return a value
 - Early termination



Watch out for `&&` vs. `&` (and `||` vs. `|`)... logical vs. bit level operators
one of the more common oopsies in
C programming

- Examples (choose one)

- `!0x41` is `0x00`
 - `!0x00` is `0x01`
 - `!!0x41` is `0x01`
 - `0x69 && 0x55` is `0x00`
 - `0x69 || 0x55` is `0x01`
 - `p && *p` - avoids null pointer access since logical operators do not evaluate second argument if result can be determined with first
 - Similarly `a && 5/a` will never cause a division by 0

Shift Operations

- **Left Shift:** $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- **Right Shift:** $x \gg y$
 - Shift bit-vector **x** right **y** positions
 - Throw away extra bits on right
 - Logical shift
 - Fill with 0's on left
 - Arithmetic shift
 - Replicate most significant bit on left
- **Undefined Behavior**
 - Shift amount < 0 or \geq word size

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Integer Representations: Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x = 15213;  
short int y = -15213;
```

Sign Bit

- C short 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Sign Bit
 - For 2's complement, most significant bit indicates sign
 - 0 for nonnegative
 - 1 for negative

Two's-complement Encoding Example

x =	15213:	00111011	01101101
y =	-15213:	11000100	10010011

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		15213	-15213	

Numeric Ranges

- Unsigned Values
 - $UMin = 0$
000...0
 - $UMax = 2^w - 1$
111...1
- Two's Complement Values
 - $TMin = -2^{w-1}$
100...0
 - $TMax = 2^{w-1} - 1$
011...1
- Other Values
 - Minus 1
111...1

Values for $w = 16$ bits

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- Observations
 - $|TMin| = TMax + 1$
 - Asymmetric range
 - $UMax = 2 * TMax + 1$
- C Programming
 - `#include <limits.h>`
 - Declares constants, e.g.,
 - `ULONG_MAX`
 - `LONG_MAX`
 - `LONG_MIN`
 - Values platform specific

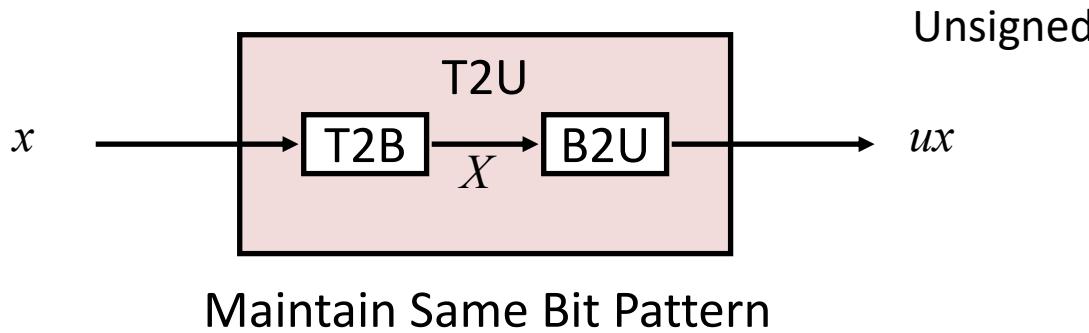
Unsigned & Signed Numeric Values

X	$B2U(X)$	$B2T(X)$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

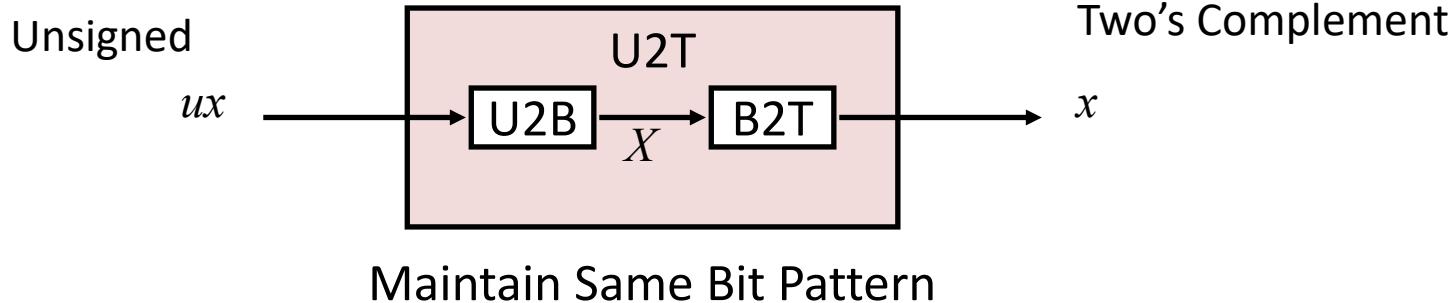
- Equivalence
 - Same encodings for nonnegative values
- Uniqueness
 - Every bit pattern represents unique integer value
 - Each representable integer has unique bit encoding
- \Rightarrow Can Invert Mappings
 - $U2B(x) = B2U^{-1}(x)$
 - Bit pattern for unsigned integer
 - $T2B(x) = B2T^{-1}(x)$
 - Bit pattern for two's comp integer

Mapping Between Signed & Unsigned

Two's Complement



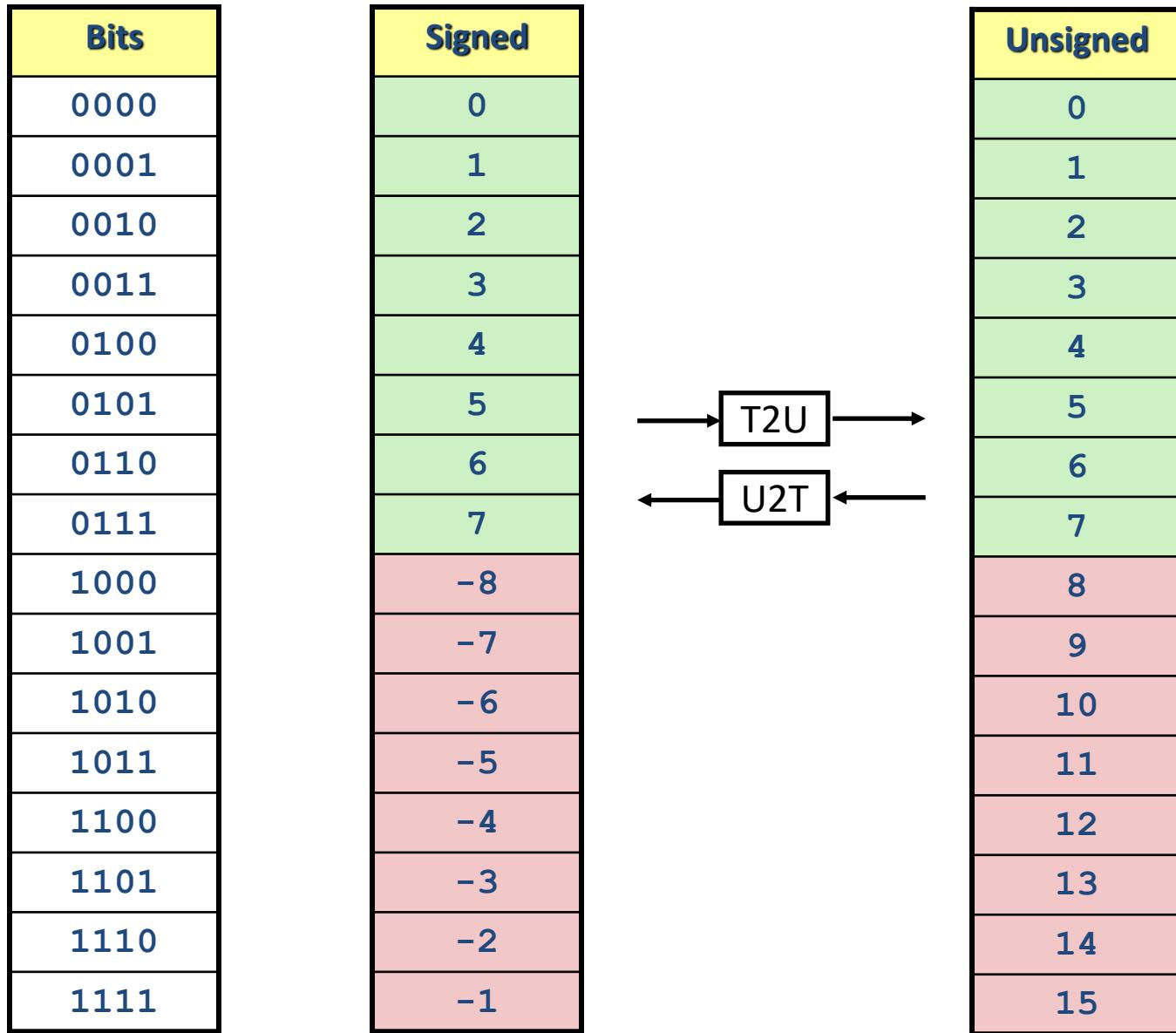
Unsigned



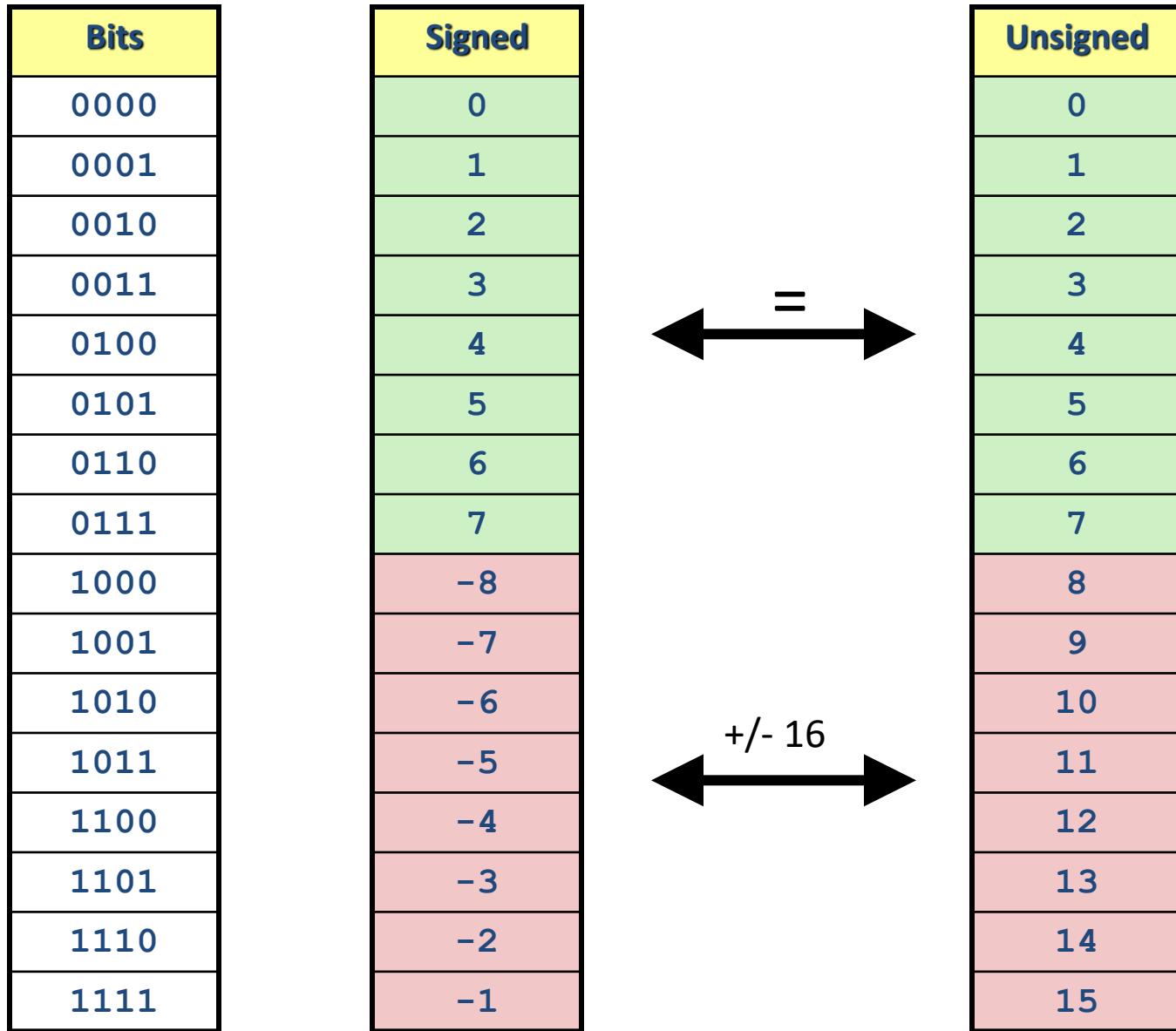
Maintain Same Bit Pattern

- Mappings between unsigned and two's complement numbers:
Keep bit representations and reinterpret

Mapping Signed ↔ Unsigned

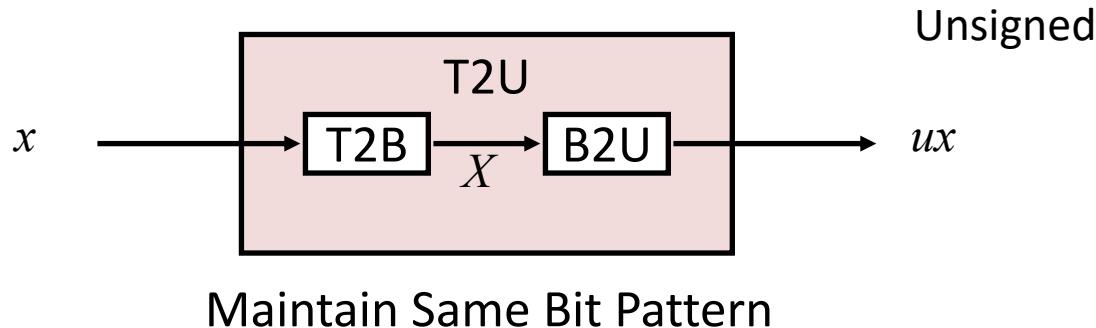


Mapping Signed ↔ Unsigned

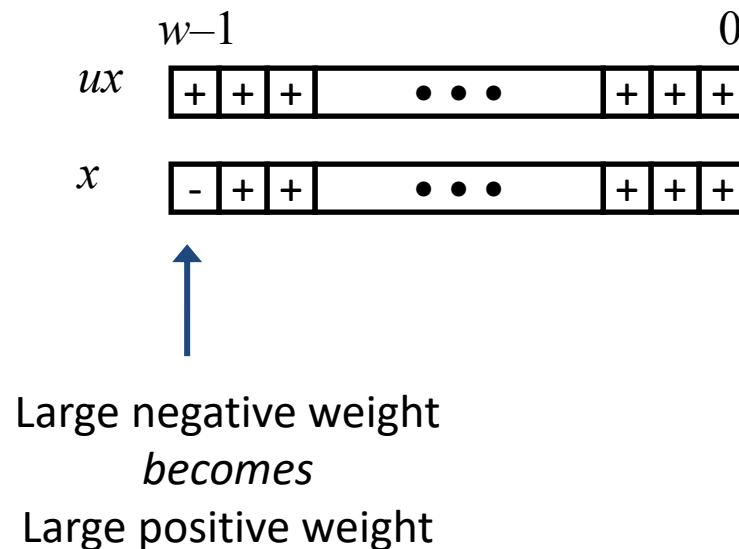


Relation between Signed & Unsigned

Two's Complement



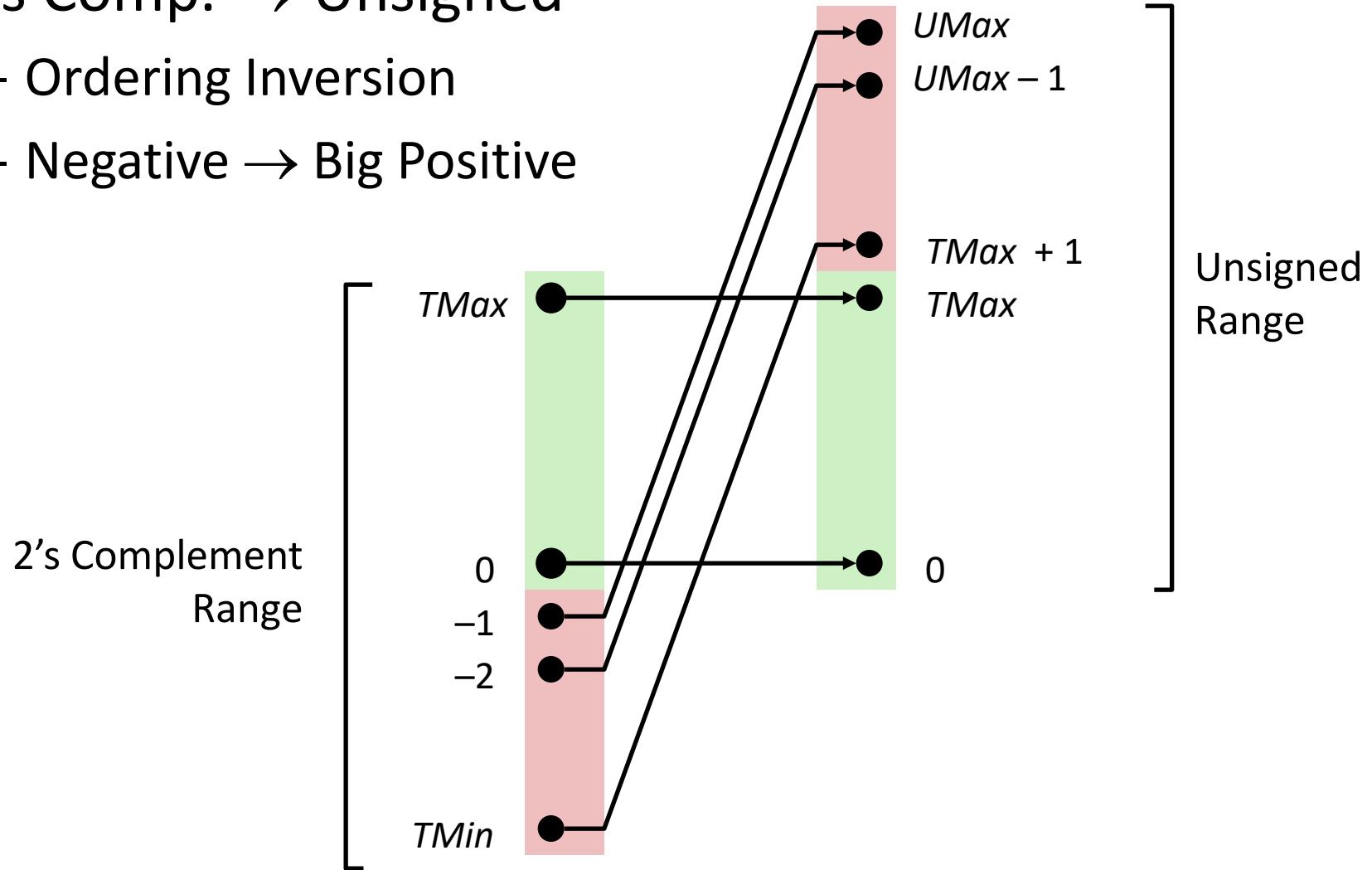
Maintain Same Bit Pattern



Add 2^w if $x < 0$,
otherwise remains same

Conversion Visualized

- 2's Comp. \rightarrow Unsigned
 - Ordering Inversion
 - Negative \rightarrow Big Positive



Signed vs. Unsigned in C

- Constants
 - By default are considered to be signed integers
 - Unsigned if have “U” as suffix
0U, 4294967259U
- Casting
 - Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```
 - Implicit casting also occurs via assignments and procedure calls

```
tx = ux; /* cast to signed */
uy = ty; /* cast to unsigned */
```

Casting Surprises

- Expression Evaluation
 - If there is a mix of unsigned and signed in single expression,
signed values implicitly cast to unsigned
 - Including comparison operations <, >, ==, <=, >=
 - Examples for $W = 32$: **TMIN = -2,147,483,648 , TMAX = 2,147,483,647**

Casting Surprises

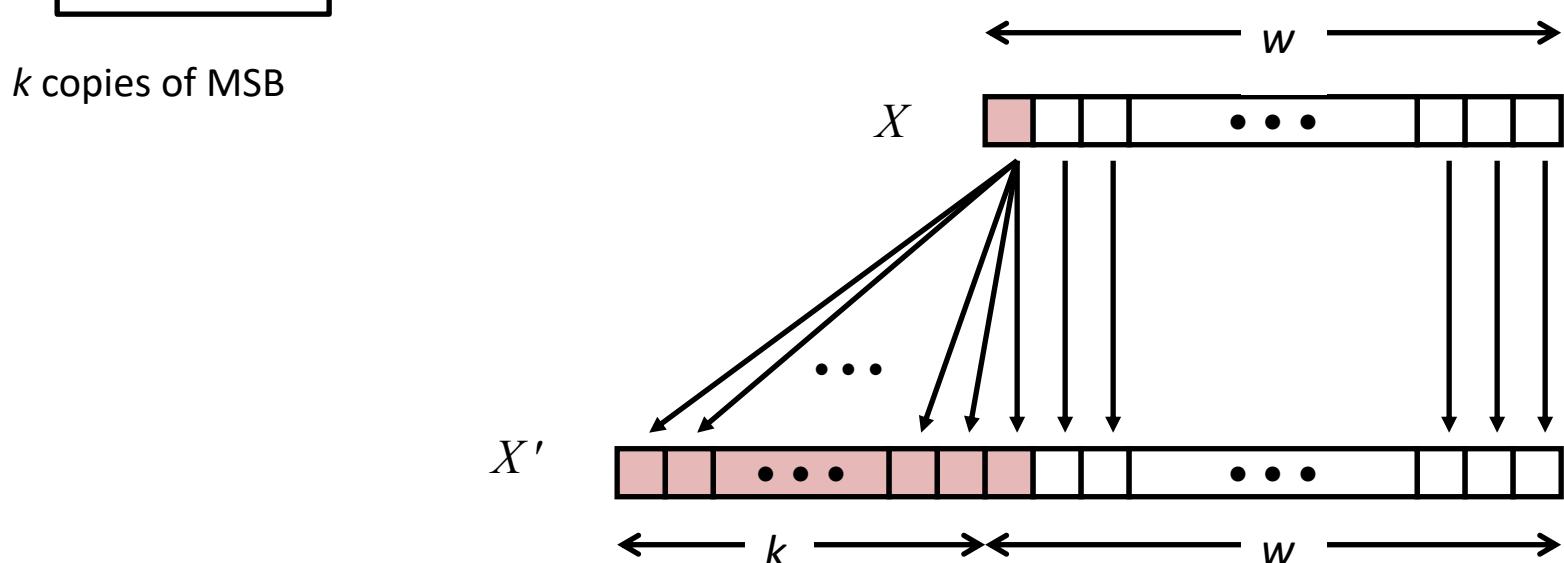
Constant ₁	Constant ₂	Relation	Evaluation
0	0U	==	Unsigned
-1	0	<	Signed
-1	0U	>	Unsigned
2147483647	-2147483647-1	>	Signed
2147483647U	-2147483647-1	<	Unsigned
-1	-2	>	Signed
(unsigned)-1	-2	>	Unsigned
2147483647	2147483648U	<	Unsigned
2147483647	(int) 2147483648U	>	signed

Summary: Casting Signed \leftrightarrow Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w
- Expression containing signed and unsigned int
 - int is cast to unsigned!!

Sign Extension

- Task:
 - Given w -bit signed integer x
 - Convert it to $w+k$ -bit integer with same value
- Rule:
 - Make k copies of sign bit:
 - $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \dots, x_0$



Sign Extension Example

```
short int x = 15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Truncation of number

- When truncating a w bit number to a k-bit number, we drop the high order $w-k$ bits
- Result: $x' = x \bmod 2^k$
- While similar property holds for two's-complement, it converts the most significant bit into a sign bit

```
int x = 53191
short sx = (short) x /* -12345 */
int y = sx;           /* -12345 */
```

Truncation of number

- Summary:
- $B2U_k([x_{k-1}, x_{k-2}, \dots, x_0]) = B2U_k([x_{w-1}, x_{w-2}, \dots, x_0]) \text{ mod } 2^k$
- $B2T_k([x_{k-1}, x_{k-2}, \dots, x_0]) = U2T_k(B2U([x_{w-1}, x_{w-2}, \dots, x_0]) \text{ mod } 2^k)$

Summary: Expanding, Truncating: Basic Rules

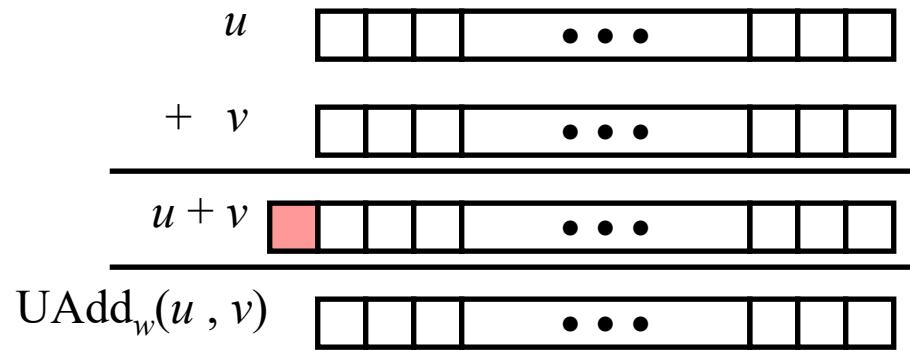
- Expanding (e.g., short int to int)
 - Unsigned: zeros added
 - Signed: sign extension
 - Both yield expected result
- Truncating (e.g., unsigned to unsigned short)
 - Unsigned/signed: bits are truncated
 - Result reinterpreted
 - Unsigned: mod operation
 - Signed: similar to mod
 - For small numbers yields expected behavior

Unsigned Addition

Operands: w bits

True Sum: $w+1$ bits

Discard Carry: w bits

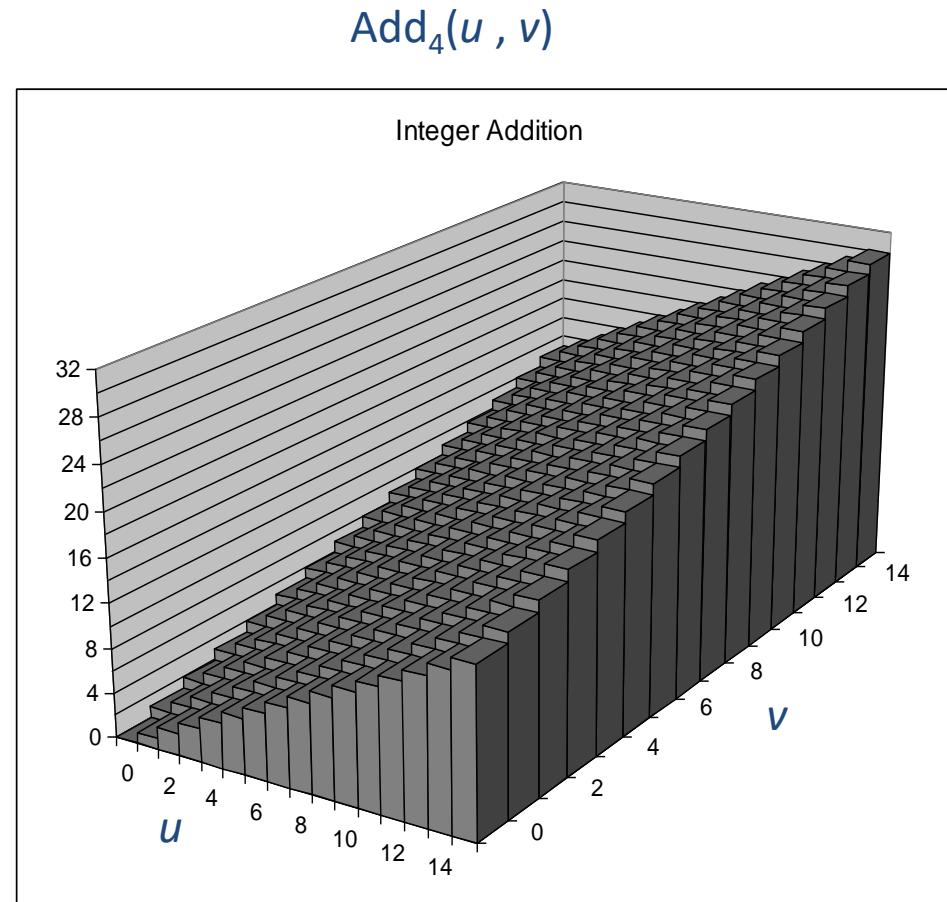


- Standard Addition Function
 - Ignores carry output
- Implements Modular Arithmetic

$$s = UAdd_w(u, v) = u + v \bmod 2^w$$

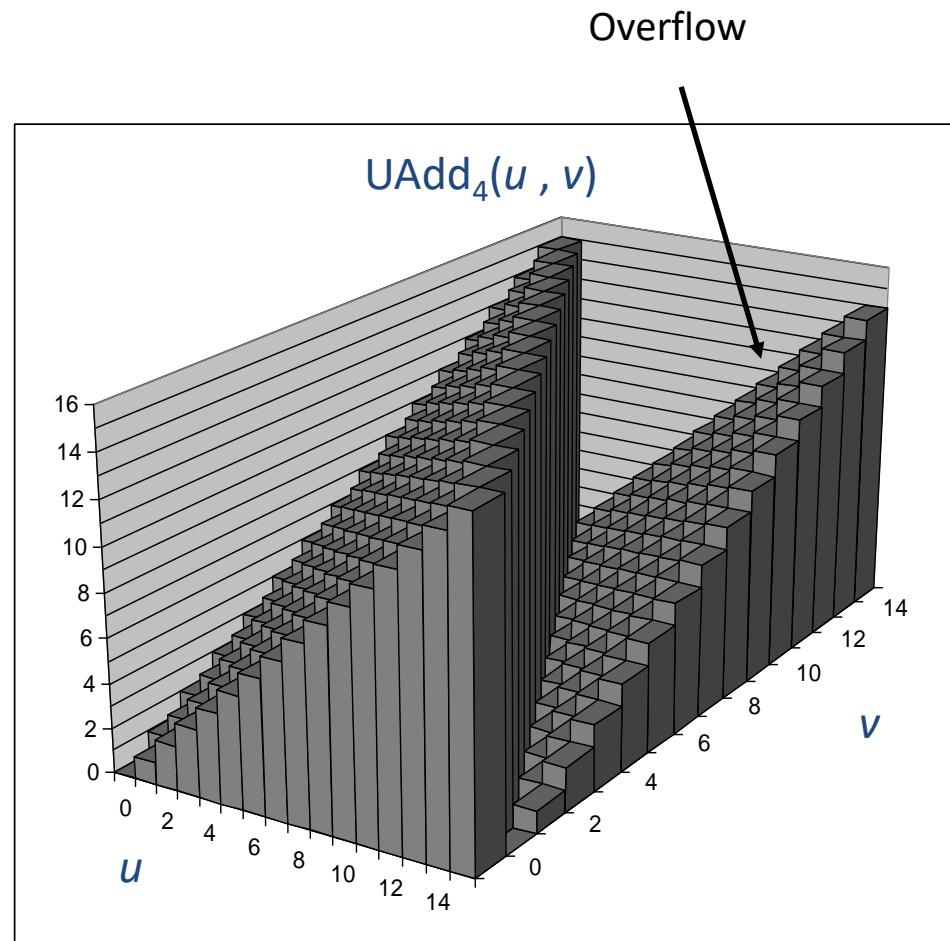
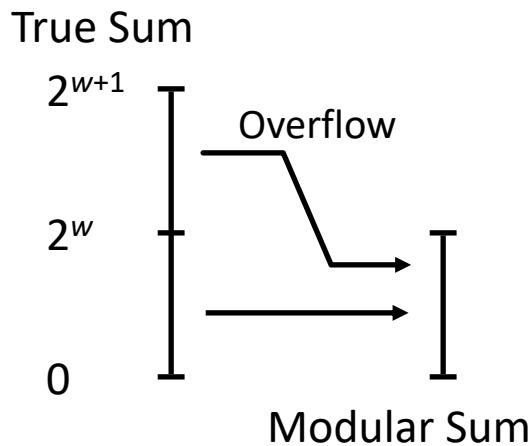
Visualizing (Mathematical) Integer Addition

- Integer Addition
 - 4-bit integers u , v
 - Compute true sum $\text{Add}_4(u, v)$
 - Values increase linearly with u and v
 - Forms planar surface



Visualizing Unsigned Addition

- Wraps Around
 - If true sum $\geq 2^w$
 - At most once
 - Decrement by 2^w

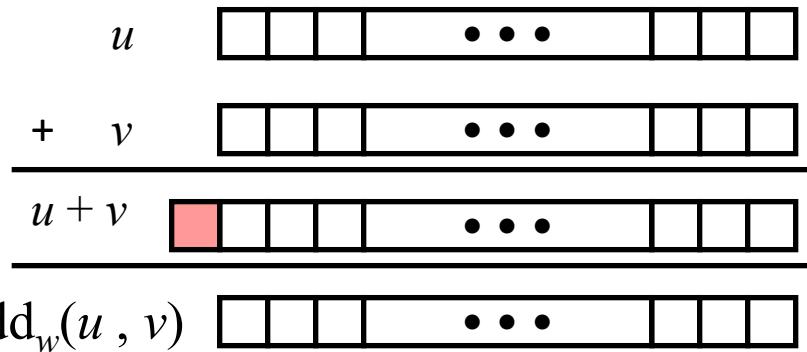


Two's Complement Addition

Operands: w bits

True Sum: $w+1$ bits

Discard Carry: w bits



- TAdd and UAdd have Identical Bit-Level Behavior

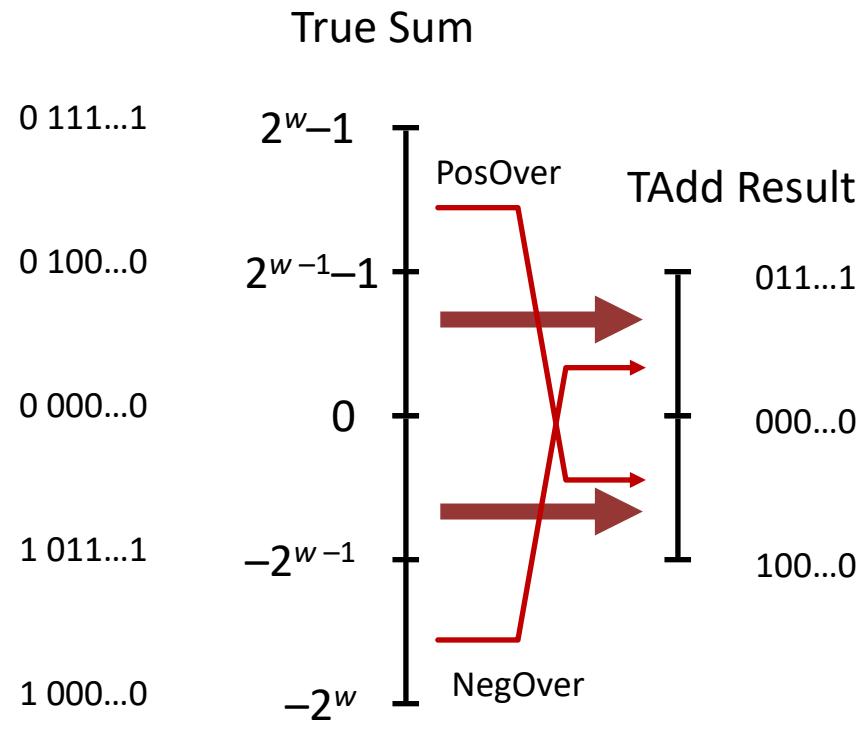
- Signed vs. unsigned addition in C:

```
int s, t, u, v;  
s = (int) ((unsigned) u + (unsigned) v);  
t = u + v
```

- Will give $s == t$

TAdd Overflow

- **Functionality**
 - True sum requires $w+1$ bits
 - Drop off MSB
 - Treat remaining bits as 2's comp. integer

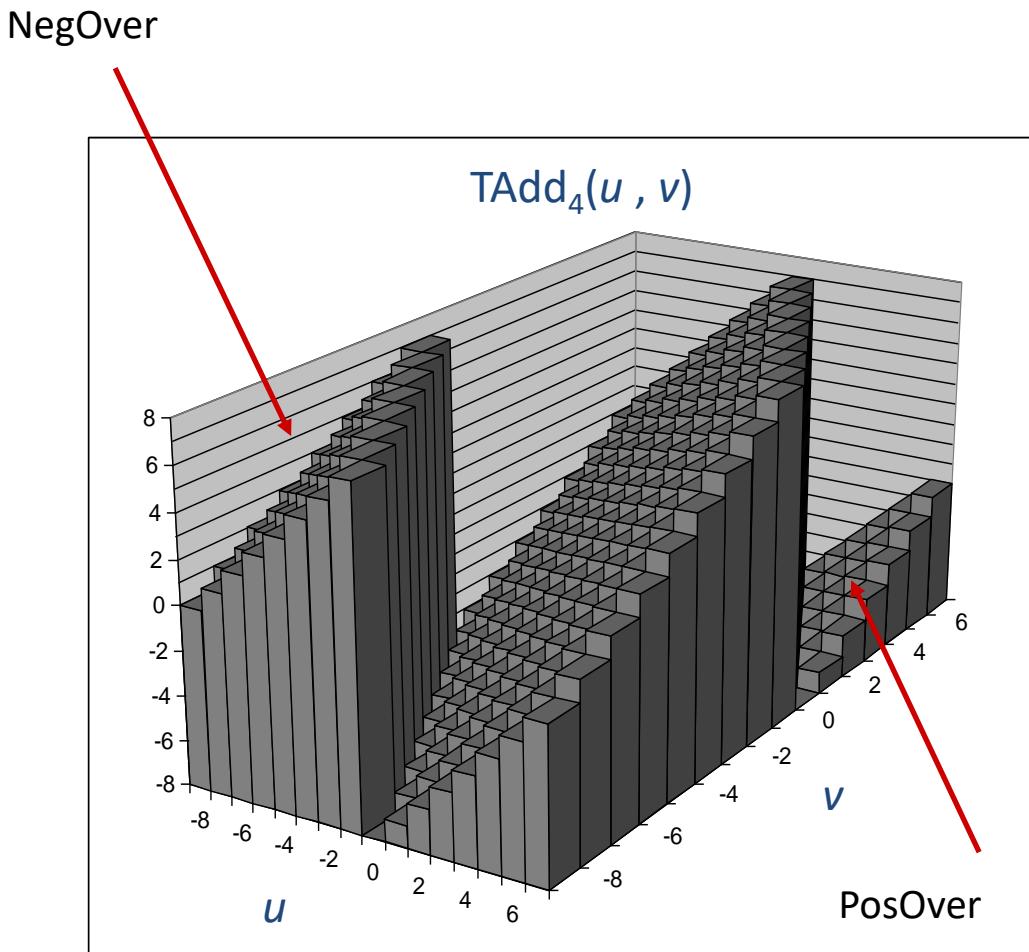


Two's Complement Addition

- In summary, subtract 2^w if positive overflow
- Add 2^w if negative overflow
- No changes if $2^{w-1} \leq \text{sum} < 2^w - 1$
- For $w = 4$ bits,
- $-8 [1000] + -5 [1011] = -13 [10011] = 3 [0011]$
- $5 [0101] + 5 [0101] = 10 [01010] = -6 [1010]$

Visualizing 2's Complement Addition

- Values
 - 4-bit two's comp.
 - Range from -8 to +7
- Wraps Around
 - If $\text{sum} \geq 2^{w-1}$
 - Becomes negative
 - At most once
 - If $\text{sum} < -2^{w-1}$
 - Becomes positive
 - At most once



Two's Complement Negation

- Complement the bits, increment the result by 1
- 0101 [5] → 1010 [-6] → 1011 [-5]
- 1000 [-8] → 0111 [7] → 1000 [-8]
- ...

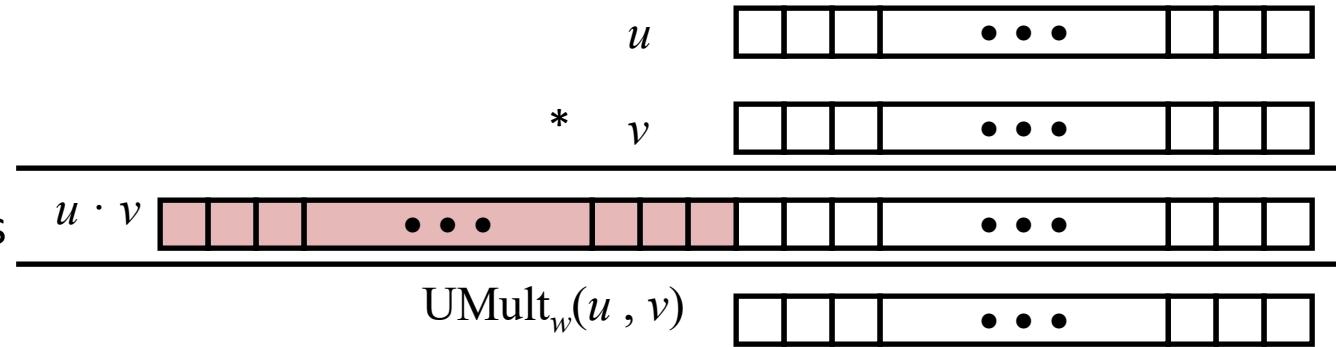
Multiplication

- Goal: Computing Product of w -bit numbers x, y
 - Either signed or unsigned
- But, exact results can be bigger than w bits
 - Unsigned: up to $2w$ bits
 - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
 - Two's complement min (negative): Up to $2w-1$ bits
 - Result range: $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
 - Two's complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$
 - Result range: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- So, maintaining exact results...
 - would need to keep expanding word size with each product computed
 - is done in software, if needed
 - e.g., by “arbitrary precision” arithmetic packages

Unsigned Multiplication in C

Operands: w bits

Discard w bits: w bits



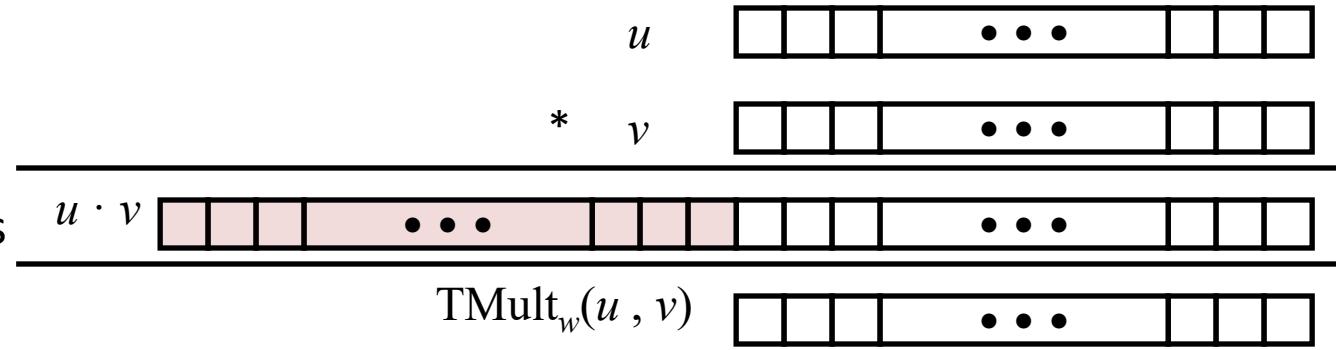
- Standard Multiplication Function
 - Ignores high order w bits
- Implements Modular Arithmetic
$$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$$

Signed Multiplication in C

Operands: w bits

True Product: $2w$ bits

Discard w bits: w bits



- Standard Multiplication Function
 - Ignores high order w bits
 - Some of which are different for signed vs. unsigned multiplication
 - Lower bits are the same

Example

- Unsigned: $5 [101] * 3 [011] = 15 [01111] \rightarrow 7 [111]$ Truncated
 - 101
 - 011
 - 101
 - 101
 - 000

- 01111

Example

- Two's C: -3 [101] * 3 [011] = -9 [110111] → -1 [111]
Truncated
- Need to sign extend and then multiply
- 111101
- 000011
- 111101
- 111101
- 000000
- 000000
- 000000
- 000000
- -----
- 000101110111

Power-of-2 Multiply with Shift

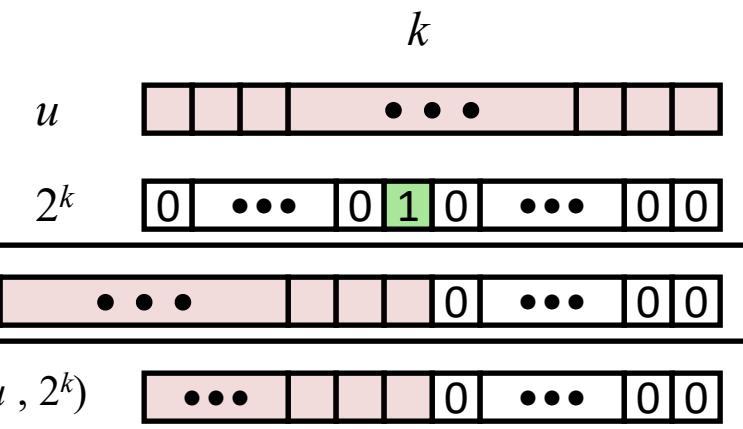
- Operation

- $u \ll k$ gives $u * 2^k$
- Both signed and unsigned

Operands: w bits

True Product: $w+k$ bits

Discard k bits: w bits



- Examples

- $u \ll 3 == u * 8$

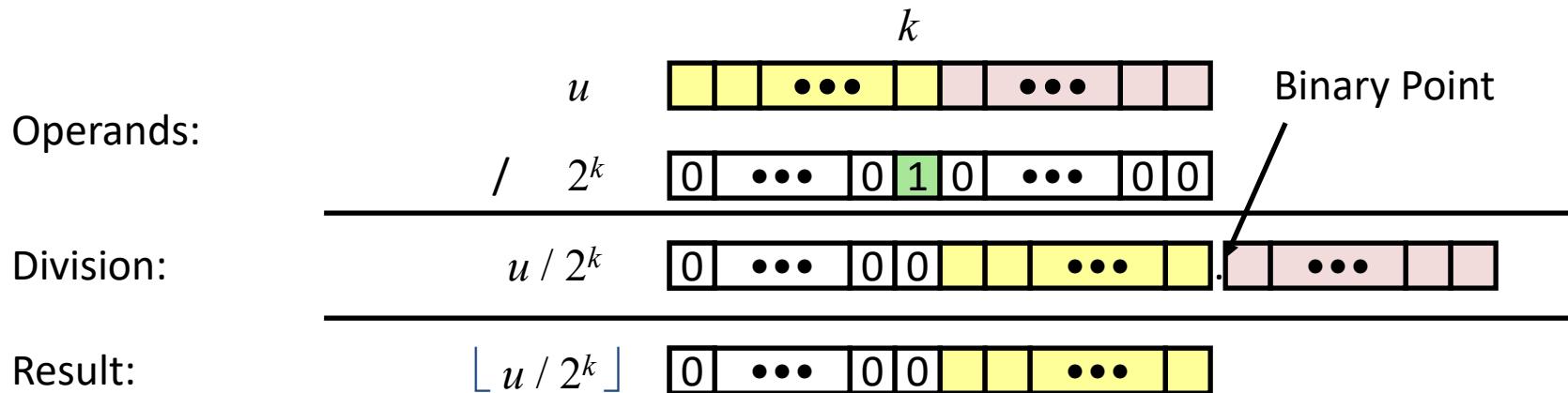
- $(u \ll 5) - (u \ll 3) == u * 24$

- Most machines shift and add faster than multiply

- Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
 - $u \gg k$ gives $\lfloor u / 2^k \rfloor$
 - Uses logical shift



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

Arithmetic: Basic Rules

- Addition:
 - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
 - Unsigned: addition mod 2^w
 - Mathematical addition + possible subtraction of 2^w
 - Signed: modified addition mod 2^w (result in proper range)
 - Mathematical addition + possible addition or subtraction of 2^w
- Multiplication:
 - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
 - Unsigned: multiplication mod 2^w
 - Signed: modified multiplication mod 2^w (result in proper range)

Using Unsigned

- *Don't* use without understanding implications
 - Easy to make mistakes

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

- Can be very subtle

```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i-DELTA >= 0; i-= DELTA)  
    . . .
```


Computer Systems Organization

Topic 3

Based on chapter 3 from Computer Systems
by Randal E. Bryant and David R. O'Hallaron

Historical Perspective: Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
 - Backwards compatible up until 8086, introduced in 1978
 - Added more features as time goes on
- x86 is a family of complex instruction set computer (CISC) instruction set architectures
 - Initially developed by Intel based on the Intel 8086 microprocessor and its 8088 variant
 - 8086 was introduced in 1978 as a fully 16-bit extension of Intel's 8-bit 8080 microprocessor
 - The term "x86" came into being because the names of several successors to Intel's 8086 processor end in "86", including the 80816, 80286, 80386 and 80486 processors.

Intel x86 Processors

- As of 2022, most desktop computers, laptops and game consoles (exception of Nintendo Switch) sold are based on the x86 architecture family, while mobile categories such as smartphones or tablets are dominated by ARM processor developed by Advance RISC Machines (ARM).
- At the high end, x86 continues to dominate compute intensive workstation and cloud computing segments, while the fastest supercomputer in 2020 was ARM-based, with the top 4 no longer x86-based in that year.
- Wiki page provides list of X86 CPUs

Intel x86 Processors (wiki page)

- 16 bit: 8086, 8088, 80186, 80286
- 32 bit Intel: i386 (80386), i486 (80486), Pentium, Pentium Pro, Pentium 2 (or II), Pentium 3 (or III), Older versions of the Pentium 4, Pentium M, Core, Older Xeon, Mobile versions of Intel Atom, Older Celeron
- 64 bit Intel: Newer Prescott Pentium 4, Pentium D, Core 2, Core i3, i5, i7, and i9, Newer Atom, Pentium dual core, Newer Celeron, Newer Xeon
- 32 bit AMD: AMD386, AMD486, AMD586, Am5x86-P75, K5, K6/K6-II/K6-III, Athlon, Athlon XP, Duron, Sempron, Geode

Intel x86 Processors (wiki page)

- 64 bit AMD: Opteron, Athlon 64, Phenom, Phenom 2, FX, Sempron, APU A4/A6/A8/A10/A12, APU Athlon, APU Sempron, Ryzen, Epyc
- Other processors: Cyrix 386/486S/DLC, 5x86, 6x86, MII, MIII (32 bit), IDT Winchip (32 bit), Rise (32 bit), NXGen (32 bit), Via C3 and C7 (32 bit), Via Nano (64 bit)

Instruction set

- An instruction set is a group of commands for a CPU in machine language. The term can refer to all possible instructions for a CPU or a subset of instructions to enhance its performance in certain situations. Some instructions are simple read, write and move commands that direct data to different hardware.
- A complex instruction set computer (CISC) is a computer architecture in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.
 - Term retroactively coined in contrast to reduced instruction set computer (RISC) and has become umbrella term for everything that is not RISC
 - Most RISC designs use uniform instruction length for almost all instructions, and employ strictly separate load and store instructions.

Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
• 8086	1978	29K	5-10
			<ul style="list-style-type: none">• First 16-bit Intel processor. Basis for IBM PC & DOS• 1MB address space
• 386	1985	275K	16-33
			<ul style="list-style-type: none">• First 32 bit Intel processor , referred to as IA32• Added “flat addressing”, capable of running Unix
• Pentium 4E	2004	125M	2800-3800
			<ul style="list-style-type: none">• First 64-bit Intel x86 processor, referred to as x86-64
• Core 2	2006	291M	1060-3500
			<ul style="list-style-type: none">• First multi-core Intel processor
• Core i7	2008	731M	1700-3900
			<ul style="list-style-type: none">• Four cores

Address space vs. Memory space

- Address used by programmer is a virtual address - set of such addresses generated by the programs as they reference instructions and data is the address space. Generally, address space is larger than the memory space.
- An address in main memory is called a location or physical address. Set of such locations is called the memory space. Generally, the memory space is smaller than the address space.
- For example if,
 - Address space = 24 bits
 - Memory space = 16 bits
 - # of address space possible = $2^{24} = 16M$
 - # of memory spaces possible = $2^{16} = 64K$

CPU Clock Speed

- Performance of CPU (“brain” of your PC) has a major impact on the speed at which programs load and how smoothly they run.
- Few different ways to measure processor performance - clock speed (also “clock rate” or “frequency”) is one of the most significant. A higher clock speed in general means a faster CPU. However, many other factors come into play.
- CPU processes many instructions (low-level calculations like arithmetic) from different programs every second. The clock speed measures the number of cycles CPU executes per second, measured in GHz (gigahertz).
- A “cycle” is technically a pulse synchronized by an internal oscillator, but for our purposes, they’re a basic unit that helps understand a CPU’s speed.

CPU Clock Speed

- A CPU with a clock speed of 3.2 GHz executes 3.2 billion cycles per second (older CPUs had speeds measured in megahertz, or millions of cycles per second). Sometimes, multiple instructions are completed in a single clock cycle; in other cases, one instruction might be handled over multiple clock cycles.
- Since different CPU designs handle instructions differently, it's best to compare clock speeds within the same CPU brand and generation e.g., a CPU with a higher clock speed from 5 years ago might be outperformed by a new CPU with a lower clock speed, as newer architecture may deal with instructions more efficiently.
- Within the same generation of CPUs, a processor with a higher clock speed will generally outperform a processor with a lower clock speed across many applications. In general, CPU clock speed is a good indicator of your processor's performance.

Intel x86 Processors, cont.

- Machine Evolution (in terms of # of transistors)

• 386	1985	0.3M
• Pentium	1993	3.1M
• Pentium/MMX	1997	4.5M
• PentiumPro	1995	6.5M
• Pentium III	1999	8.2M
• Pentium 4	2001	42M
• Core 2 Duo	2006	291M
• Core i7	2008	731M

- Added Features

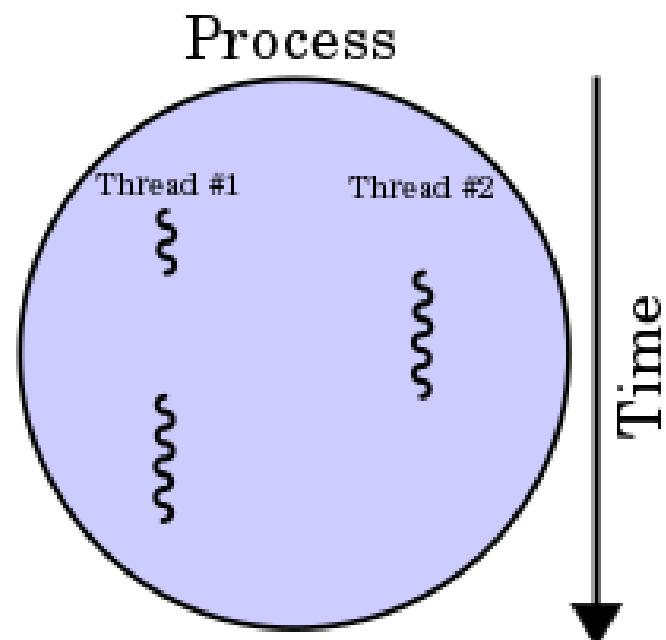
- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

Multi core processor

- A multi-core processor is a computer processor with two or more separate processing units, called cores, each of which reads and executes program instructions. The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run instructions on separate cores at the same time.
- Multithreading is the ability of a CPU (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. This approach differs from multiprocessing since the threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB).

Multithreading

- Multiprocessing systems include multiple complete processing units while multithreading aims to increase utilization of a single core by using thread-level parallelism, as well as instruction-level parallelism. Since the two techniques are complementary, they are combined in nearly all modern systems architectures.



x86 Clones: Advanced Micro Devices(AMD)

- Historically
 - AMD has followed just behind Intel
 - A little bit slower, a lot cheaper
- Then
 - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
 - Built Opteron: tough competitor to Pentium 4
 - Developed x86-64, their own extension to 64 bits
- In recent Years Intel got its act together and leads the world in semiconductor technology

Intel's 64-Bit History

- 2001: Intel Attempts Radical Shift from IA32 to IA64 (IA is Intel Architecture)
 - Totally different architecture (Itanium)
 - Executes IA32 code only as legacy
 - Performance disappointing
- 2003: AMD Steps in with Evolutionary Solution
 - x86-64 (now called “AMD64”)
- Intel Felt Obligated to Focus on IA64
 - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
 - Extended Memory 64-bit Technology
 - Almost identical to x86-64!
- All except low-end x86 processors support x86-64
 - But, lots of code still runs in 32-bit mode

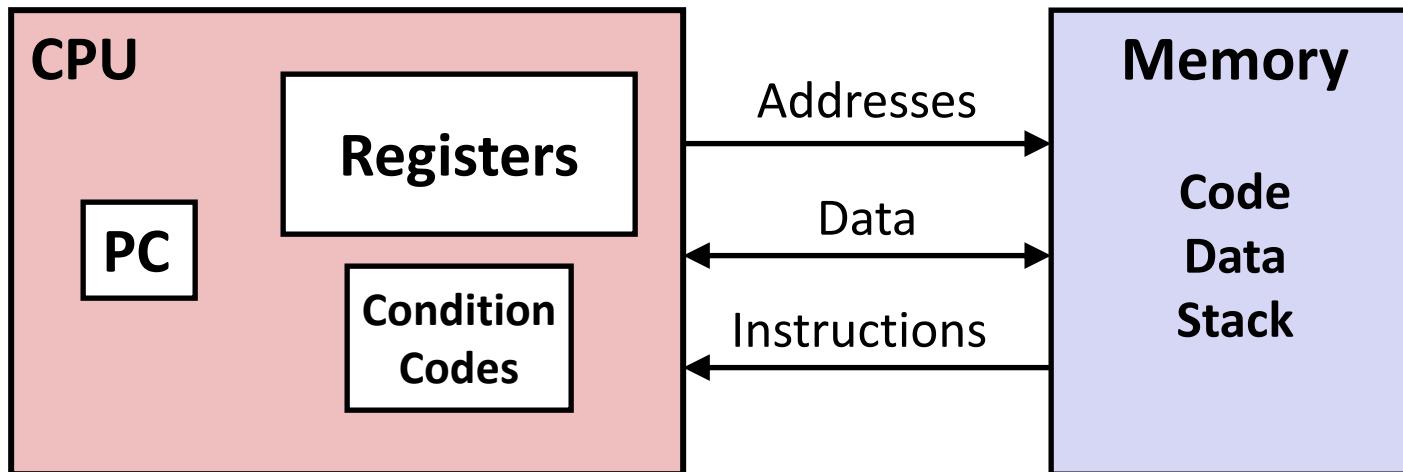
Definitions

- **Architecture (also ISA: instruction set architecture):** ISA is part of the abstract model of a computer that defines how the CPU is controlled by the software. It is a structure of commands and operations used by software to communicate with hardware. Acts as an interface between the hardware and software, specifying both what the processor is capable of doing as well as how it gets done e.g., x86 instruction set
 - ISA provides the only way through which a user is able to interact with the hardware - can be viewed as a programmer's manual because it's the portion of the machine that's visible to the assembly language programmer, the compiler writer, and the application programmer.
 - ISA defines the supported data types, the registers, how the hardware manages main memory, key features (such as virtual memory) etc. ISA can be extended by adding instructions or other capabilities, or by adding support for larger addresses and data values.

Definitions

- **Microarchitecture** (also called computer organization and sometimes abbreviated as μ arch or uarch) is a hardware implementation of an ISA (Instruction Set Architecture) i.e., it is the hardware circuitry that implements one particular ISA
 - A given ISA maybe implemented with different microarchitectures
 - Multiple CPU models may be designed for a particular microarchitecture. For this reason, a microarchitecture is sometimes referred to as a "family" or "generation" of CPU. For example, Intel Kaby Lake (7th generation) and Coffee Lake (8th generation) are separate microarchitectures, each with a "family" of compatible CPUs.
- **Computer architecture** is the combination of microarchitecture and instruction set architecture.
 - Code Forms include Machine Code (byte-level programs that a processor executes) and Assembly Code (text representation of machine code)

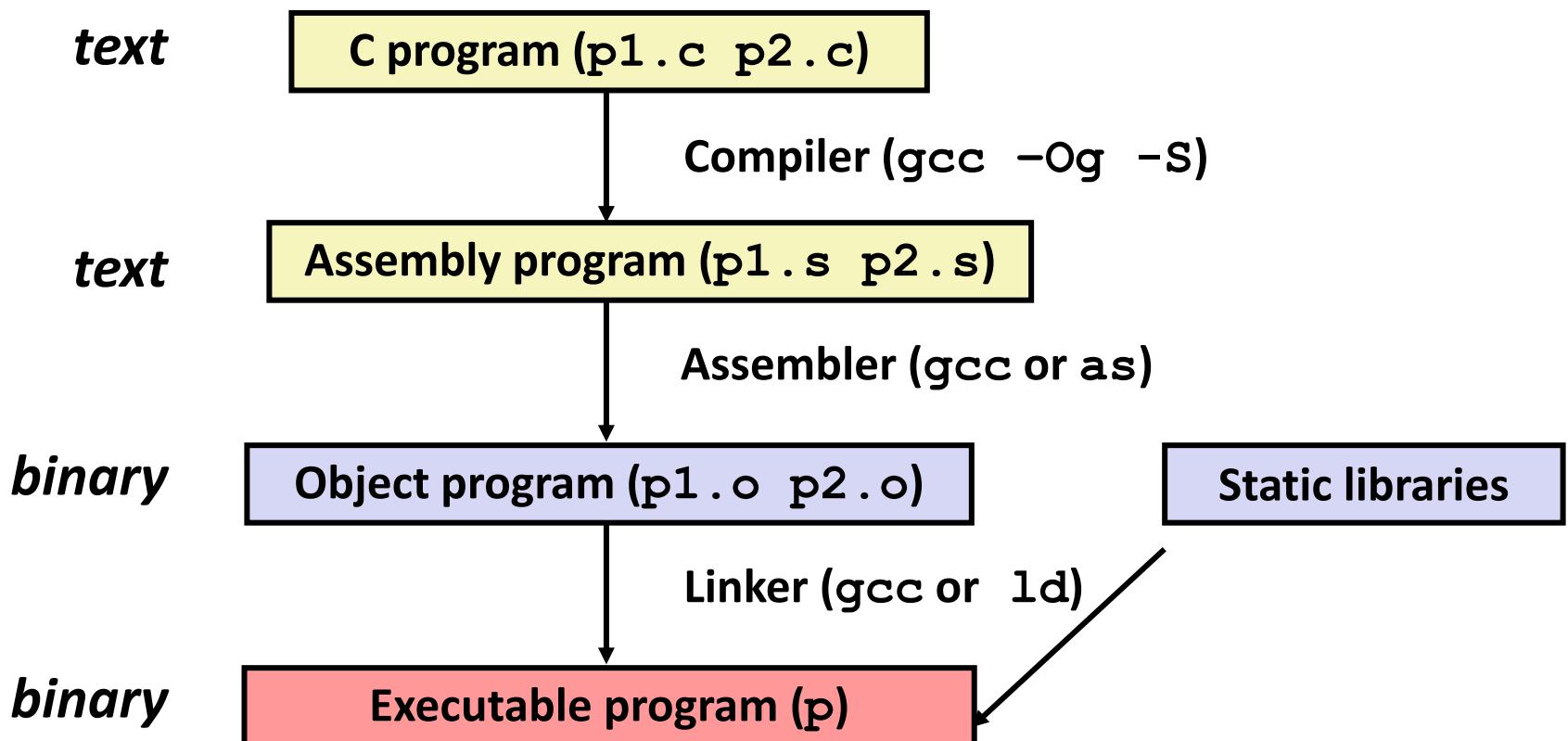
Assembly/Machine Code View of Processor State



- **PC: Program counter**
 - Address of next instruction
 - Called “RIP” (x86-64) [Register for Instruction Pointer]
- **Register file**
 - Holds addresses (pointers) or integer data
 - Heavily used program data
- **Condition code registers**
 - Store status information about most recent arithmetic or logical operation
 - Used for conditional branching
- **Memory**
 - Byte addressable array
 - Code and user data
 - Stack to support procedures

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og -o p p1.c p2.c`
 - Use basic optimizations (`-Og`) [New to recent versions of GCC]
 - Put resulting binary in file `p`



Compiling Into Assembly

C Code (mstore.c)

```
long mult2(long, long);

void multstore(long x, long y,
               long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
multstore:
    pushq  %rbx
    movq   %rdx, %rbx
    call   mult2
    movq   %rax, (%rbx)
    popq   %rbx
    ret
```

Obtained with command

```
gcc -Og -S mstore.c
```

Produces assembly file `mstore.s`

Warning: Can get very different results on different machines due to different versions of gcc and different compiler settings.

Assembly Description

- Assembly code file mstore.s contains various declarations, including lines from previous slide
- Each line of the code corresponds to a single machine instruction
- For example, pushq instruction indicates that the contents of the register %rbx should be pushed onto the program stack
- All information about local variable names or data types has been stripped away

Assembly Characteristics: Data Types

- “Integer” data of 1 (char), 2 (short), 4 (int), or 8 (long, char *) bytes
 - Data values
 - Addresses (untyped pointers)
- Floating point data of 4 (float), 8 (double), or 10 (long double) bytes
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
 - **Just contiguously allocated bytes in memory**

Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Transfer control
 - Unconditional jumps to/from procedures
 - Conditional branches

Object Code

Embedded within the object code `mstore.o`

`0x0400595:`

`0x53`

`0x48`

`0x89`

`0xd3`

`0xe8`

`0x00`

`0x00`

`0x00`

`0x00`

`0x48`

`0x89`

`0x03`

`0x5b`

`0xc3`

- Total of 14 bytes
- Each instruction 1 to 5 bytes each
- Starts at address `0x0400595`

- Assembler

- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files
- `gcc -Og -c mstore.c` (both compile and assemble)

- Linker

- Resolves references between files
- Combines with static run-time libraries
 - E.g., code for `malloc`, `printf`
- Some libraries are *dynamically linked*
 - Linking occurs when program begins execution

Machine Instruction Example

```
*dest = t;
```

- **C Code**

- Store value **t** where designated by **dest**

```
movq %rax, (%rbx)
```

- **Assembly**

- Move 8-byte value to memory
 - Quad words in x86-64 parlance
- Operands:
 - **t:** Register **%rax**
 - **dest:** Register **%rbx**
 - ***dest:** Memory **M[%rbx]**

```
0x40059e: 48 89 03
```

- **Object Code**

- 3-byte instruction
- Stored at address **0x40059e**

Disassembling Object Code

Disassembled

```
0000000000400595 <multstore>:  
 400595: 53                      push    %rbx  
 400596: 48 89 d3                mov     %rdx,%rbx  
 400599: e8 00 00 00 00          callq   400590 <mult2>  
 40059e: 48 89 03                mov     %rax,(%rbx)  
 4005a1: 5b                      pop    %rbx  
 4005a2: c3                      retq
```

- **Disassembler**
 - `objdump -d mstore.o`
 - Useful tool for examining object code
 - Produces approximate rendition of assembly code based purely on the byte sequences in the machine code file
 - Can be run on either a `.out` (complete executable) or `.o` file

Object Code

1. 0000000000000000 <multstore>:		
Offset	Byte value	Equivalent assembly
2.	0: 53	pushq %rbx
3.	1: 48 89 d3	movq %rdx, %rbx
4.	4: e8 00 00 00 00	call mult2
5.	9: 48 89 03	movq %rax, (%rbx)
6.	c: 5b	popq %rbx
7.	d: c3	ret

- X86-64 instructions can range in length from 1 to 15 bytes
- Encoding is done so commonly used and those with fewer operands require a smaller number of bytes
- Disassembler determines assembly code based purely on byte sequences (does not require access to source/assembly)

Assembly Basics: x86-64 Integer Registers

%rax	%eax
------	------

%r8	%r8d
-----	------

%rbx	%ebx
------	------

%r9	%r9d
-----	------

%rcx	%ecx
------	------

%r10	%r10d
------	-------

%rdx	%edx
------	------

%r11	%r11d
------	-------

%rsi	%esi
------	------

%r12	%r12d
------	-------

%rdi	%edi
------	------

%r13	%r13d
------	-------

%rsp	%esp
------	------

%r14	%r14d
------	-------

%rbp	%ebp
------	------

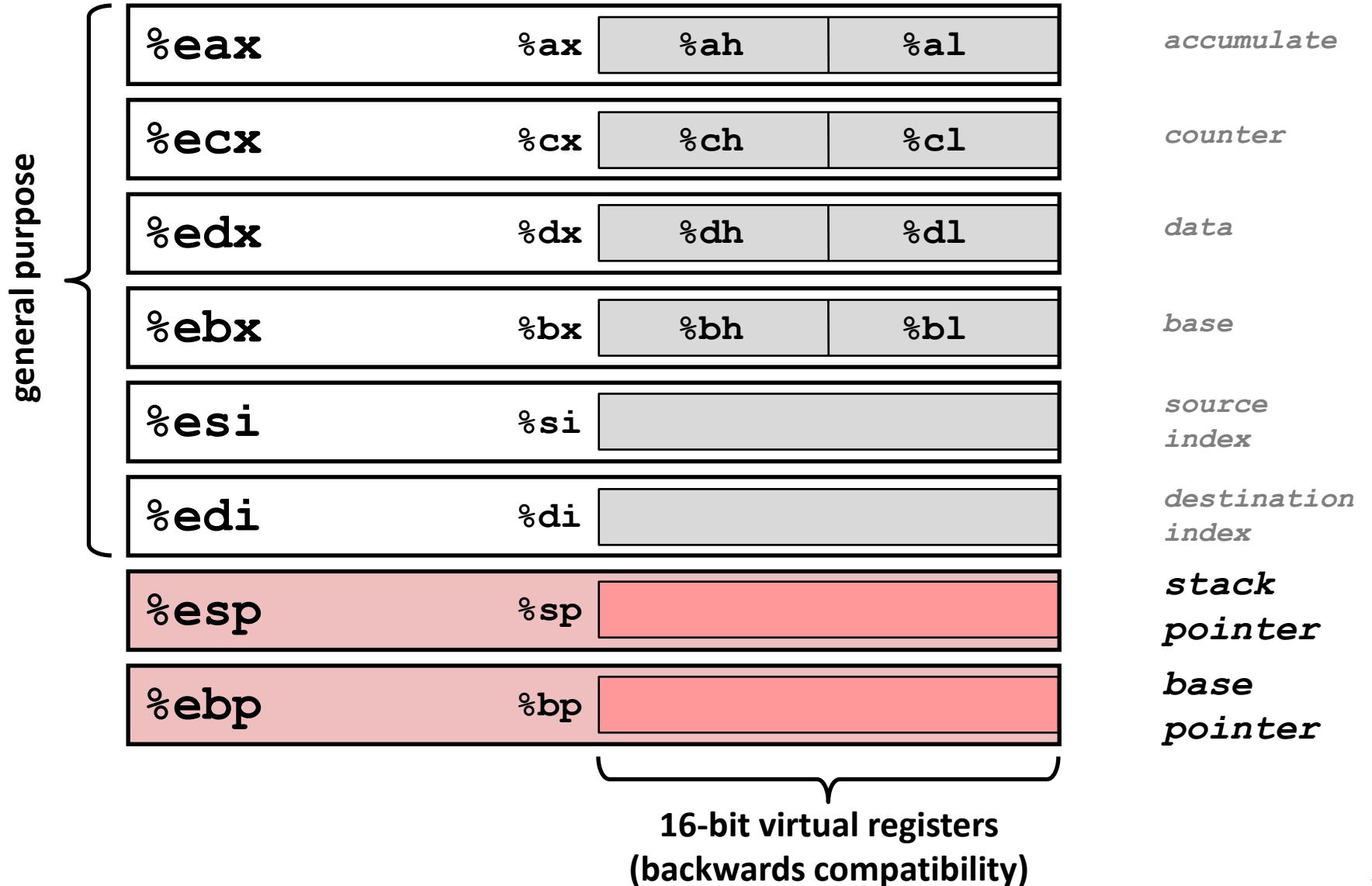
%r15	%r15d
------	-------

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Please refer page 216, table 3.2

What do they stand for ?

- 16 general purpose registers storing 64 bit values
- Store integer data as well as pointers
- Original 8086 had eight 16 bit registers %ax to %bp
- For IA32, registers expanded to 32 bit labeled %eax to %ebp
- For x86-64, the original eight registers were expanded to 64 bits, labeled %rax to %rbp. In addition, 8 new registers were added %r8 through %r15
- Stack pointer %rsp used to indicate end position in run time stack
- Other registers more flexible in usage

Some History: IA32 Registers



Moving Data

- Moving Data
 - **movq Source, Dest:**
- Operand Types
 - *Immediate*: Constant integer data
 - Example: **\$0x400, \$-533**
 - Like C constant, but prefixed with '\$'
 - Encoded with 1, 2, or 4 bytes
 - *Register*: One of the 16 integer registers
 - Example: **%rax, %r13**
 - But **%rsp** reserved for special use
 - *Memory*: 8 consecutive bytes of memory at address given by register
 - Simplest example: **(%rax)**
 - Various other “address modes”
- Source values can be constants or read from registers or memory.
Results can be stored in registers or memory.

%rax

%rcx

%rdx

%rbx

%rsi

%rdi

%rsp

%rbp

%rN

movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	<i>Imm</i>	<i>Reg</i>	movq \$0x4,%rax	temp = 0x4;
		<i>Mem</i>	movq \$-147,(%rax)	*p = -147;
	<i>Reg</i>	<i>Reg</i>	movq %rax,%rdx	temp2 = temp1;
	<i>Reg</i>	<i>Mem</i>	movq %rax,(%rdx)	*p = temp;
	<i>Mem</i>	<i>Reg</i>	movq (%rax),%rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]

- Register R specifies memory address
 - Pointer dereferencing in C

```
movq (%rcx), %rax
```

- Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
 - Constant displacement D specifies offset

```
movq 8(%rbp), %rdx
```

Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

movq	(%rdi), %rax
movq	(%rsi), %rdx
movq	%rdx, (%rdi)
movq	%rax, (%rsi)
ret	

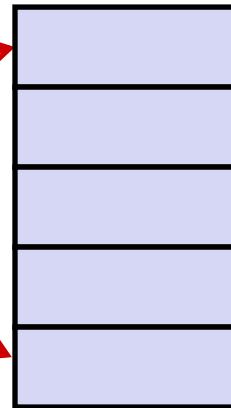
Understanding Swap()

```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Registers

%rdi	
%rsi	
%rax	
%rdx	

Memory



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Understanding Swap()

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

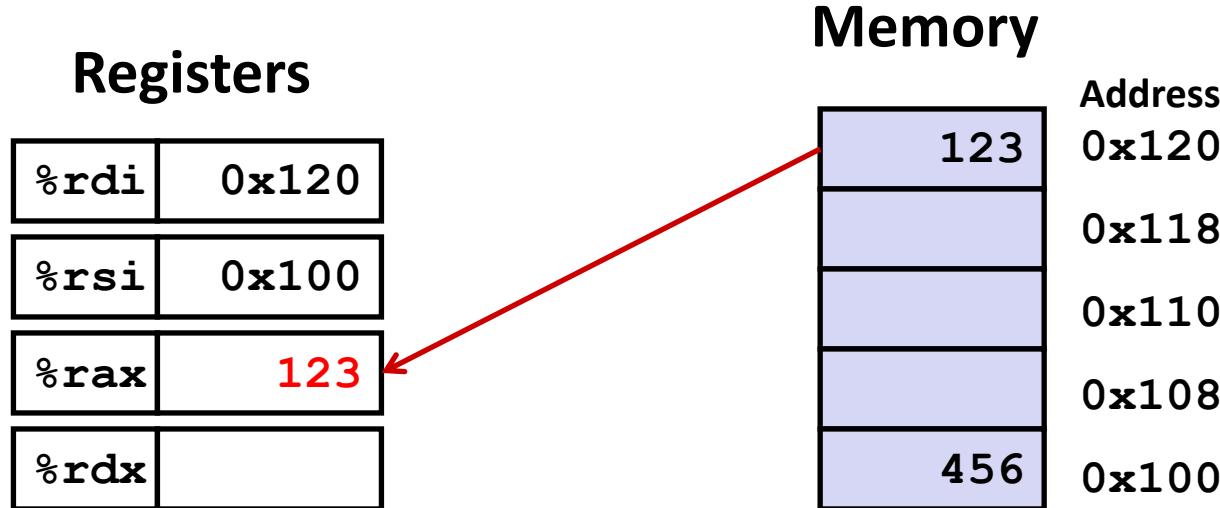
Memory

123	Address 0x120
	0x118
	0x110
	0x108
456	0x100

swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

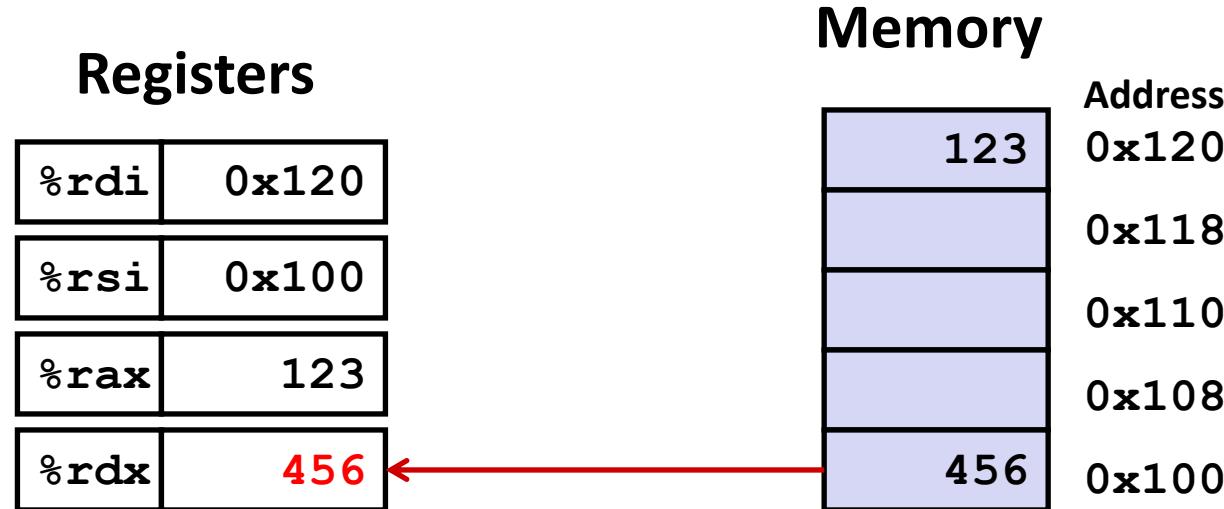
Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

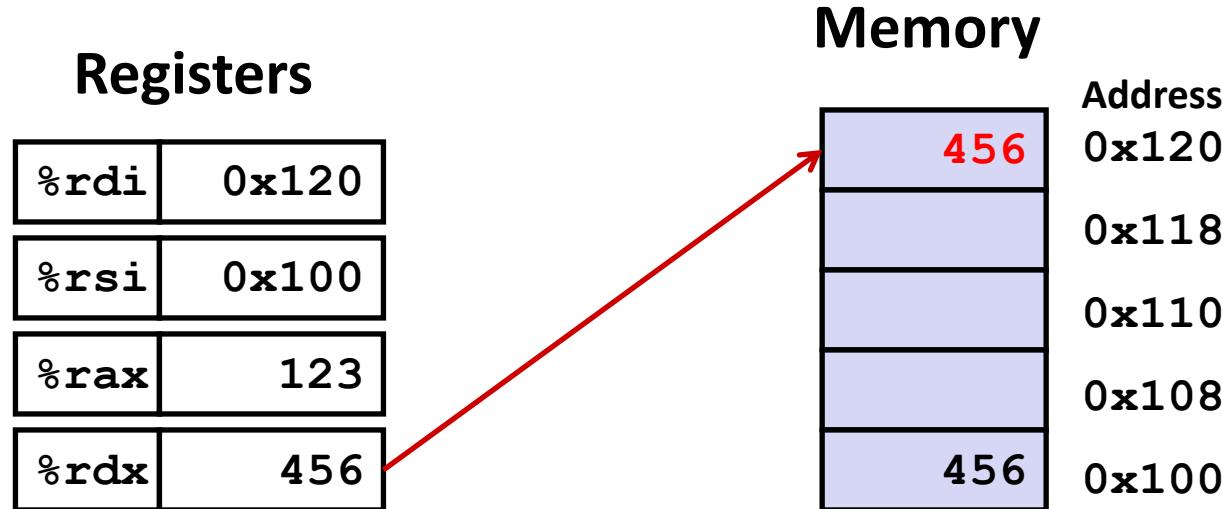
Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

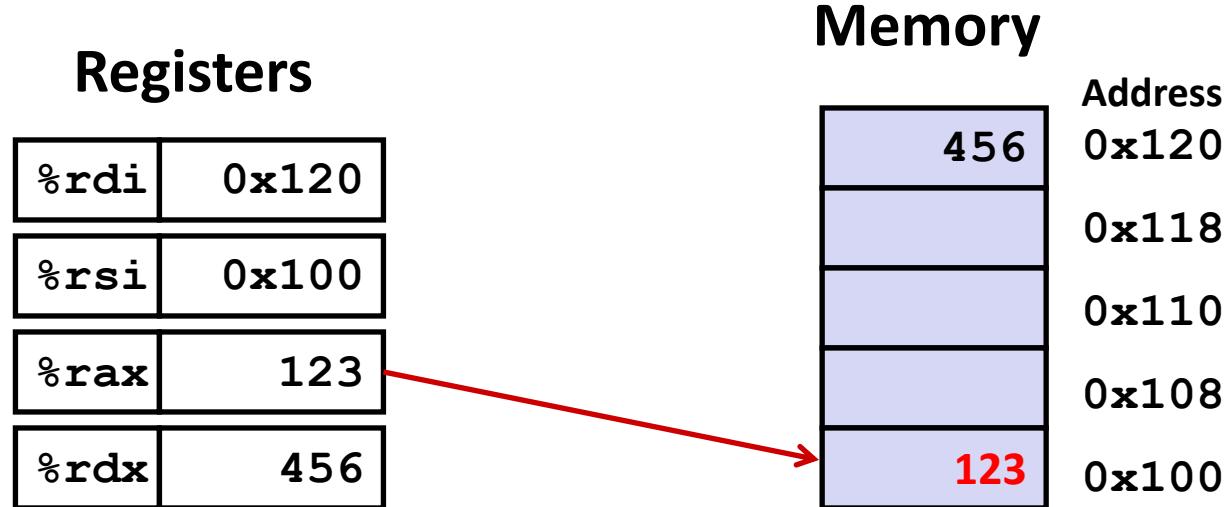
Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Simple Memory Addressing Modes

- Normal **(R)** **Mem[Reg[R]]**

- Register R specifies memory address
 - Pointer dereferencing in C

```
movq (%rcx), %rax
```

- Displacement **D(R)** **Mem[Reg[R]+D]**

- Register R specifies start of memory region
 - Constant displacement D specifies offset

```
movq 8(%rbp), %rdx
```

Complete Memory Addressing Modes

- Most General Form
- $D(Rb,Ri,S)$ $\text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+ D]$
 - D: Constant “displacement” of 1, 2, or 4 bytes
 - Rb: Base register: Any of 16 integer registers
 - Ri: Index register: Any, except for `%rsp`
 - S: Scaling factor S must be 1, 2, 4, or 8
- Different Forms
- (Rb,Ri) $\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]$
- $D(Rb,Ri)$ $\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]$
- (Rb,Ri,S) $\text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]$

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Zero extending data movement

- Two classes of data movement instructions when copying a smaller source value to a larger destination
 - Move with zero extension
 - Move with sign extension
- Move with zero extension MOVZ
 - movzbw → Byte to Word where Word is 2 bytes
 - movzbl → Byte to Double word
 - movzwl → Word to Double word
 - movzbq → Byte to Quad word
 - movzwq → Word to Quad word

Sign extending data movement

- Move with sign extension MOVS
 - movsbw → Byte to Word
 - movsbl → Byte to Double word
 - movswl → Word to Double word
 - movsbq → Byte to Quad word
 - movswq → Word to Quad word
 - movslq → Double word to Quad word

Arithmetic and Logical Operations

- Operations divided into 4 groups
 - load effective address (leaq), unary, binary and shifts
 - Operations given as instruction classes since they can have variants with different operand sizes (only leaq has no other size variant)
 - E.g., ADD consists of four additional instructions: addb, addw, addl, addq [bytes, word, double, quad]
- Leaq variant of movq – reads from memory to register
 - First operand appears to be a memory reference but instead of reading from the designated location, the instruction copies the effective address to the destination
 - Useful to generate pointer for later memory references and to compactly describe common arithmetic operations

Address Computation Instruction

- `leaq Src, Dst`
 - *Src* is address mode expression
 - Set *Dst* to address denoted by expression
- Uses
 - Computing addresses without a memory reference
 - E.g., translation of `p = &x[i];`
 - Computing arithmetic expressions of the form $x + k*y$
 - $k = 1, 2, 4, \text{ or } 8$
- Example: `leaq 7(%rdx,%rdx,4), %rax`
 - Will set register `%rax` to $5x+7$, if `%rdx` contains value x

Address Computation Instruction

■ Example

```
long m12(long x)
{
    return x*12;
}
```

Upon compiling:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax           # return t<<2
```

Some Arithmetic Operations

- Two Operand Instructions:

- **Format** **Computation**

• addq	<i>Src,Dest</i>	Dest = Dest + Src
• subq	<i>Src,Dest</i>	Dest = Dest – Src
• imulq	<i>Src,Dest</i>	Dest = Dest * Src
• salq	<i>Src,Dest</i>	Dest = Dest << Src
• sarq	<i>Src,Dest</i>	Dest = Dest >> Src
• shrq	<i>Src,Dest</i>	Dest = Dest >> Src
• xorq	<i>Src,Dest</i>	Dest = Dest ^ Src
• andq	<i>Src,Dest</i>	Dest = Dest & Src
• orq	<i>Src,Dest</i>	Dest = Dest Src

*Arithmetic right shift
(fill with sign bit - sarq)*
*Logical right shift
(fill with zeroes - shrq)*

- Watch out for the argument order

Signed vs. Unsigned

- No distinction between signed and unsigned data in the operations (why?)
- Operations have similar bit level behavior
 - Difference appears in interpretation of outcome
 - E.g., $5 [0101] + 5 [0101] = 10 [01010]$
 - Using 4 bits: 10[1010] with unsigned vs. -6 [1010] with signed
 - Operation is same but interpretation of outcome differs
 - Interpretation depends on the need
- Only right shifting requires instructions that differentiate between signed vs. unsigned data

Some Arithmetic Operations

- One Operand Instructions

- incq $Dest$ $Dest = Dest + 1$
- decq $Dest$ $Dest = Dest - 1$
- negq $Dest$ $Dest = -Dest$
- notq $Dest$ $Dest = \sim Dest$

Arithmetic Expression Example

```
long arith  
(long x, long y, long z)  
{  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    return rval;  
}
```

arith:

```
    leaq    (%rdi,%rsi), %rax  
    addq    %rdx, %rax  
    leaq    (%rsi,%rsi,2), %rdx  
    salq    $4, %rdx  
    leaq    4(%rdi,%rdx), %rcx  
    imulq   %rcx, %rax  
    ret
```

Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
 - But, only used once

Arithmetic Expression Example

```
long arith  
(long x, long y, long z)  
{  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    return rval;  
}
```

arith:

```
leaq    (%rdi,%rsi), %rax    # t1  
addq    %rdx, %rax          # t2  
leaq    (%rsi,%rsi,2), %rdx  
salq    $4, %rdx            # t4  
leaq    4(%rdi,%rdx), %rcx  # t5  
imulq   %rcx, %rax          # rval  
ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Computer Systems Organization

Topic 3 Contd.

Based on chapter 3 from Computer Systems
by Randal E. Bryant and David R. O'Hallaron

Mechanisms in Procedures

■ Passing control

- To beginning of procedure code
- Back to return point

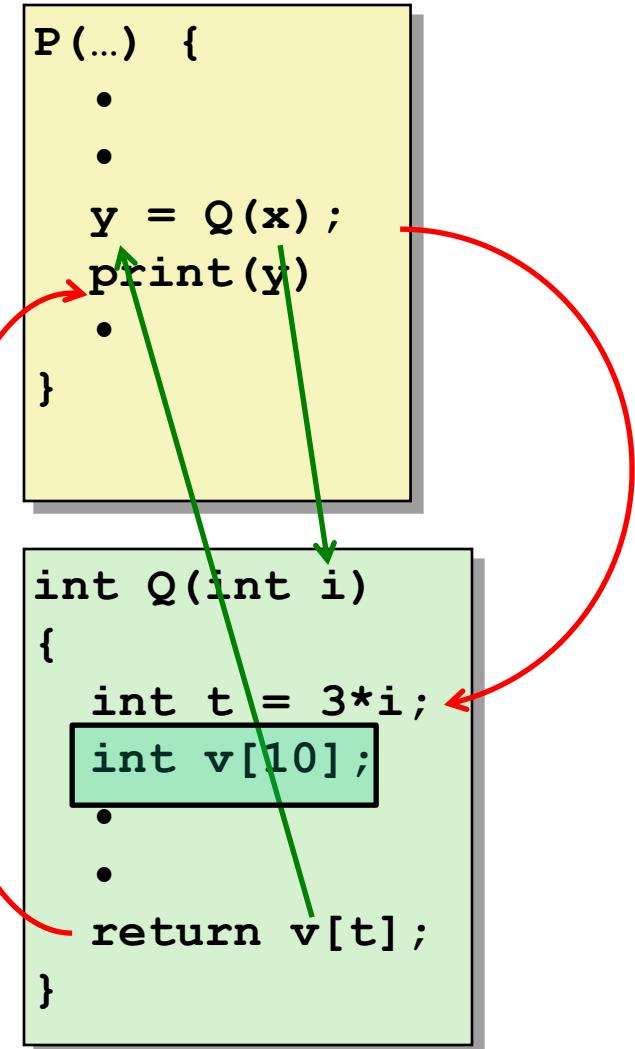
■ Passing data

- Procedure arguments
- Return value

■ Memory management

- Allocate during procedure execution
- Deallocate upon return

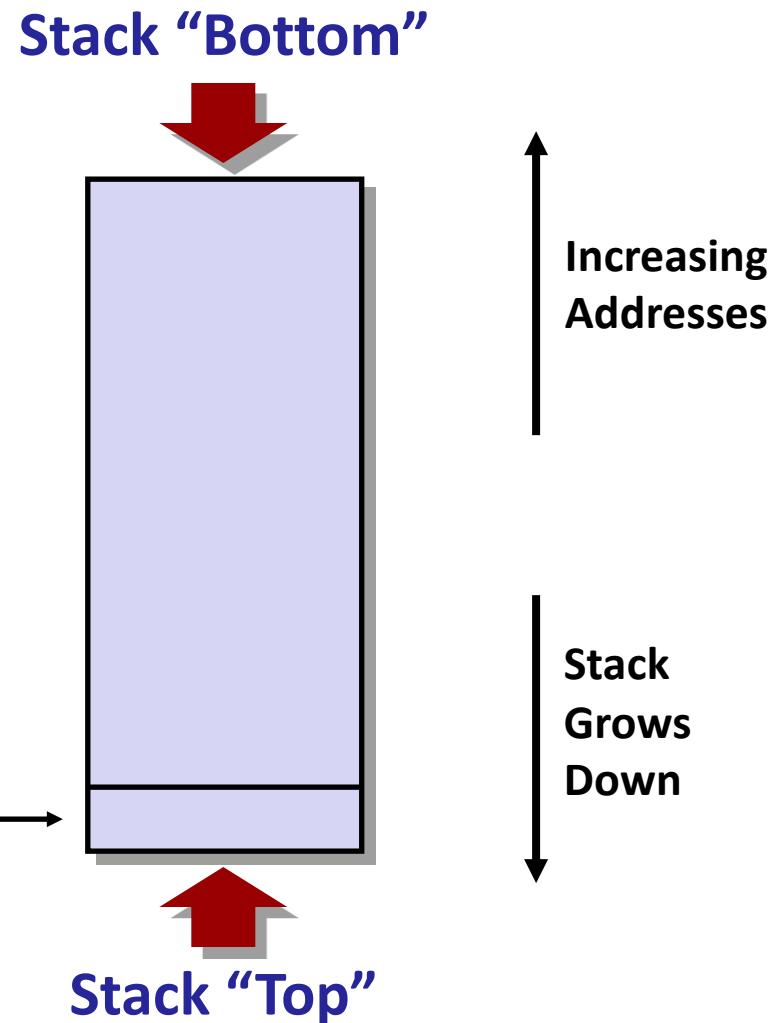
■ Mechanisms all implemented with machine instructions



x86-64 Stack

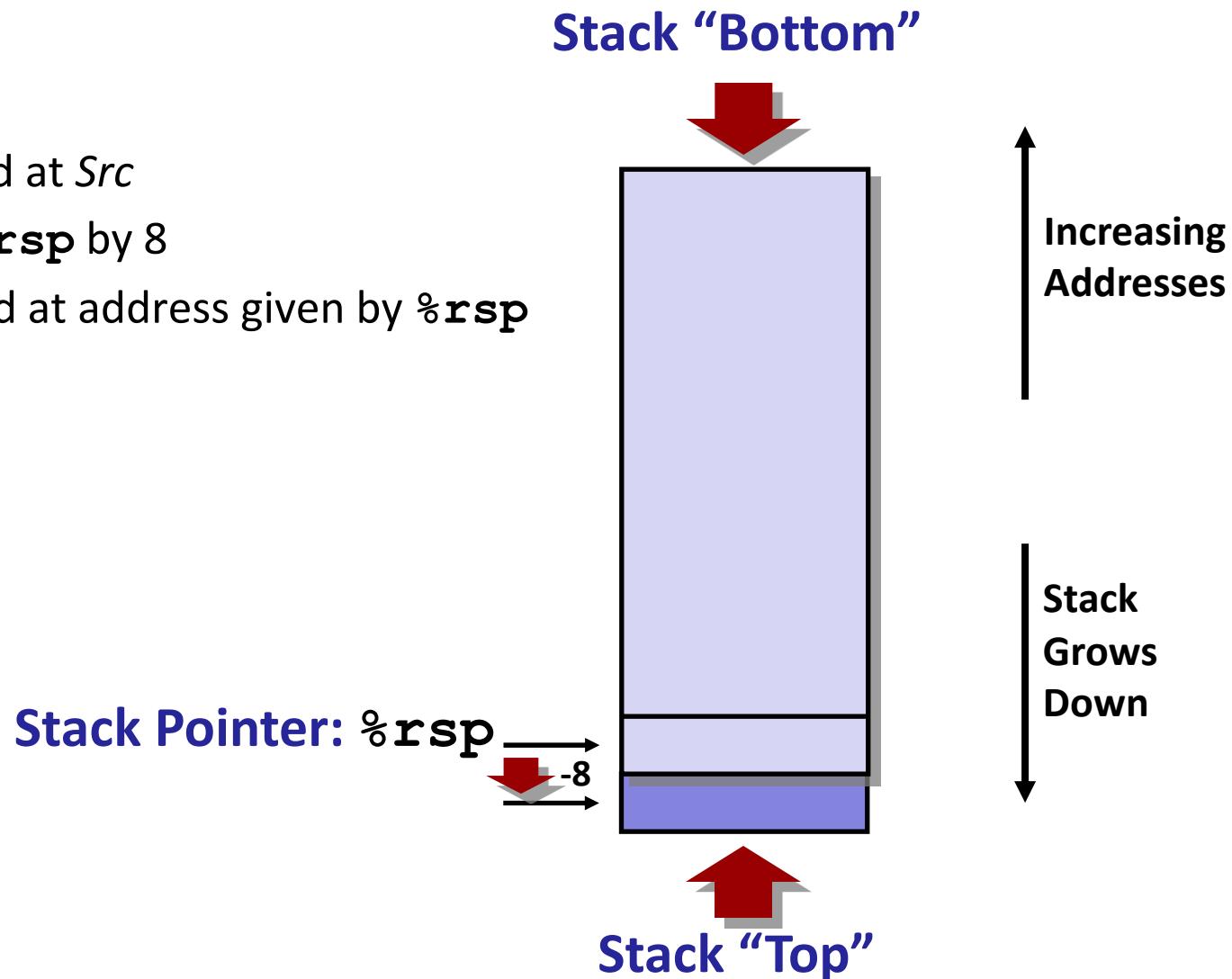
- Region of memory managed with stack
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
 - address of “top” element

Stack Pointer: `%rsp` →



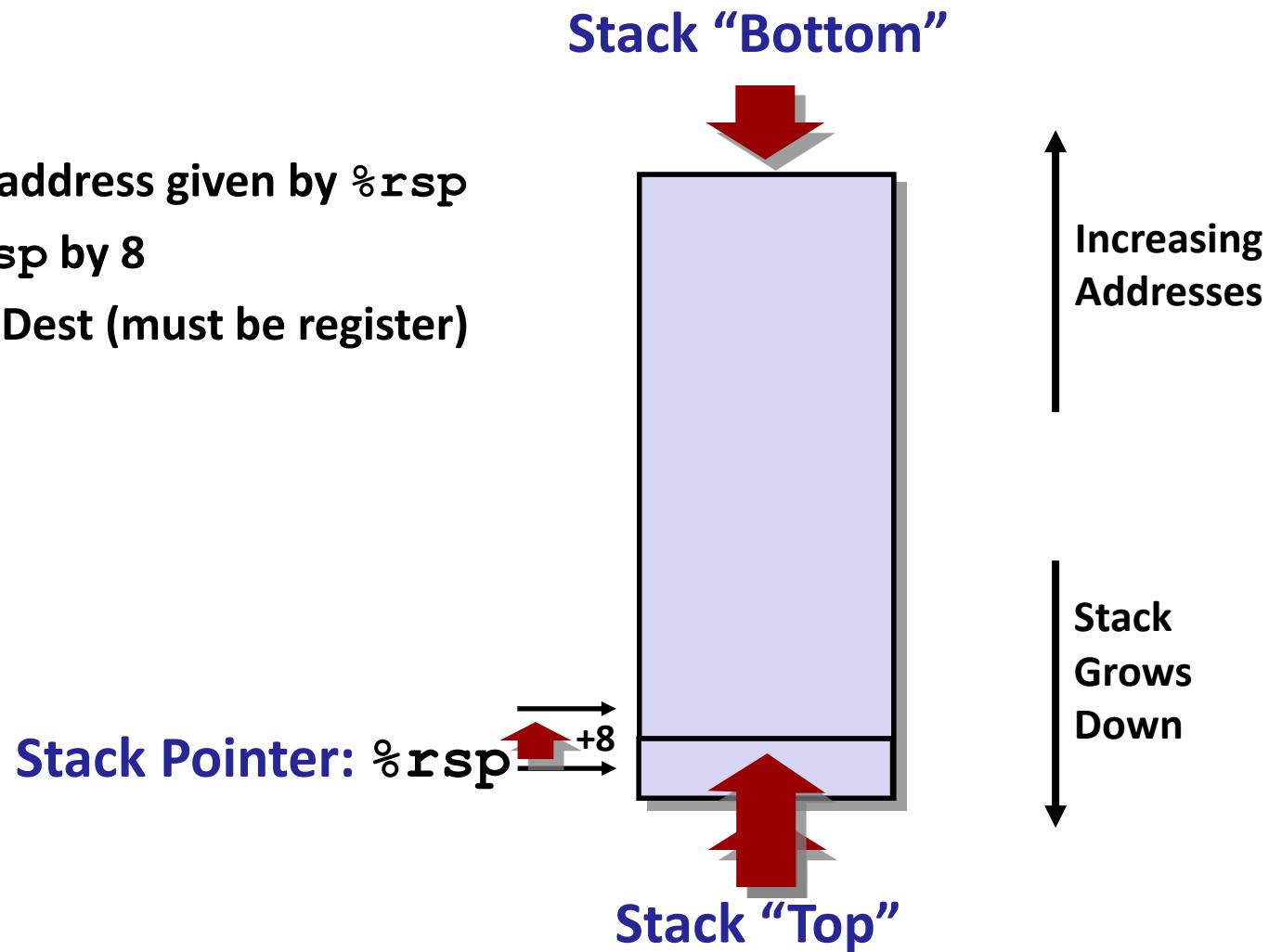
x86-64 Stack: Push

- **pushq *Src***
 - Fetch operand at *Src*
 - Decrement **%rsp** by 8
 - Write operand at address given by **%rsp**



x86-64 Stack: Pop

- **popq Dest**
 - Read value at address given by `%rsp`
 - Increment `%rsp` by 8
 - Store value at Dest (must be register)



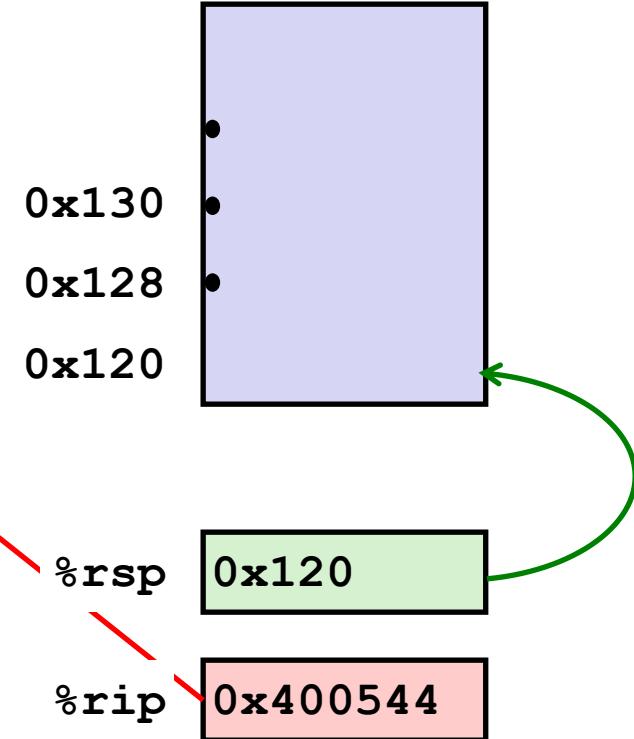
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** call label
 - Push return address on stack
 - Jump to *label*
- Return address:
 - Address of the next instruction right after call
 - Example from disassembly
- **Procedure return:** ret
 - Pop address from stack
 - Jump to address

Control Flow Example #1

```
0000000000400540 <multstore>:  
•  
•  
400544: callq  400550 <mult2>  
400549: mov     %rax, (%rbx)  
•  
•
```

```
0000000000400550 <mult2>:  
400550:  mov     %rdi,%rax  
•  
•  
400557:  retq
```



`%rsp` stack pointer
`%rip` program counter

Control Flow Example #2

```
0000000000400540 <multstore>:
```

```
•  
•  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←
```

0x130

0x128

0x120

0x118

0x400549

%rsp 0x118

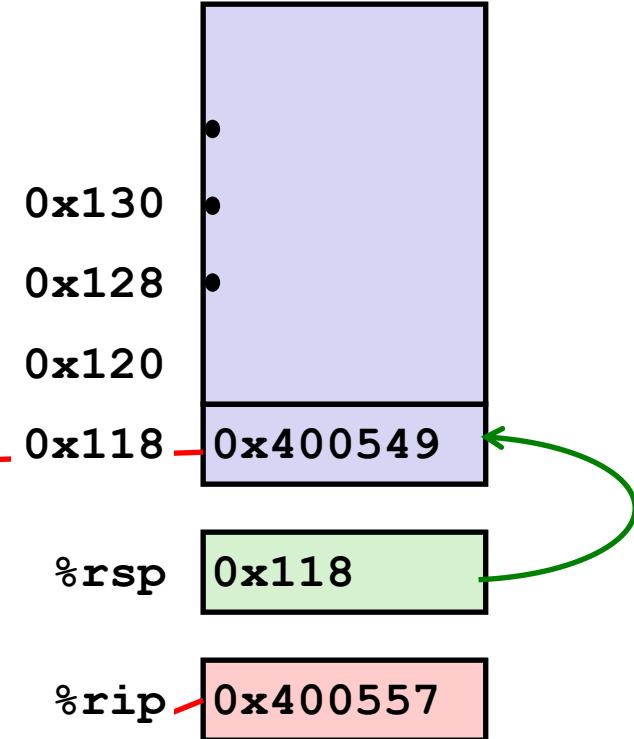
%rip 0x400550

```
0000000000400550 <mult2>:
```

```
400550: mov    %rdi,%rax ←  
•  
•  
400557: retq
```

Control Flow Example #3

```
0000000000400540 <multstore>:  
•  
•  
400544: callq  400550 <mult2>  
400549: mov     %rax, (%rbx) ←  
•  
•
```

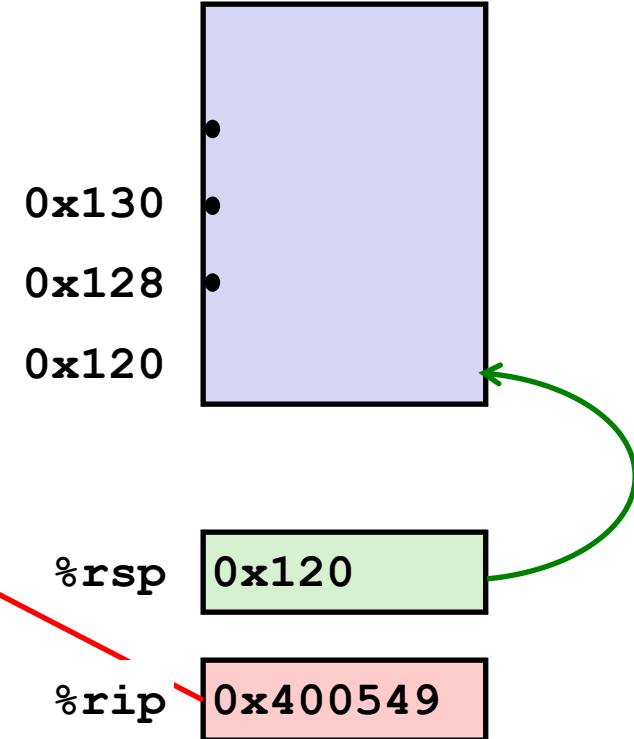


```
0000000000400550 <mult2>:  
400550:  mov     %rdi,%rax  
•  
•  
400557:  retq ←
```

Control Flow Example #4

```
0000000000400540 <multstore>:  
•  
•  
400544: callq  400550 <mult2>  
400549: mov     %rax, (%rbx) ←  
•  
•
```

```
0000000000400550 <mult2>:  
400550:  mov     %rdi,%rax  
•  
•  
400557:  retq
```



Procedure Data Flow

Registers

- First 6 arguments



Stack



- Return value



- Only allocate stack space when needed

Registers %rbx, %rbp and %r12-r15 are **callee-save registers**, meaning that they are saved across function calls.

Data Flow Examples (Disassembled code)

```
void multstore
    (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
# x in %rdi, y in %rsi, dest in %rdx
...
400541: mov    %rdx,%rbx          # Save dest
400544: callq  400550 <mult2>    # mult2(x,y)
# t in %rax
400549: mov    %rax,(%rbx)      # Save at dest
...
```

```
long mult2
    (long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
# a in %rdi, b in %rsi
400550: mov    %rdi,%rax        # a
400553: imul   %rsi,%rax        # a * b
# s in %rax
400557: retq
```

Stack-Based Languages

- **Languages that support recursion**
 - e.g., C, Pascal, Java
 - Code must be “*Reentrant*”
 - Multiple simultaneous instantiations of single procedure
 - Need some place to store state of each instantiation
 - Arguments
 - Local variables
 - Return pointer
- **Stack based model**
 - State for given procedure needed for limited time
 - From when called to when return
 - Callee returns before caller does
- **Stack allocated in *Frames***
 - state for single procedure instantiation

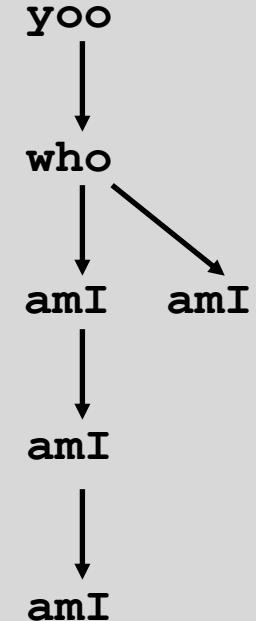
Call Chain Example

```
yoo(...)  
{  
    •  
    •  
    who();  
    •  
    •  
}
```

```
who(...)  
{  
    • • •  
    amI();  
    • • •  
    amI();  
    • • •  
}
```

```
amI(...)  
{  
    •  
    •  
    amI();  
    •  
    •  
}
```

Example
Call Chain



Procedure `amI()` is recursive

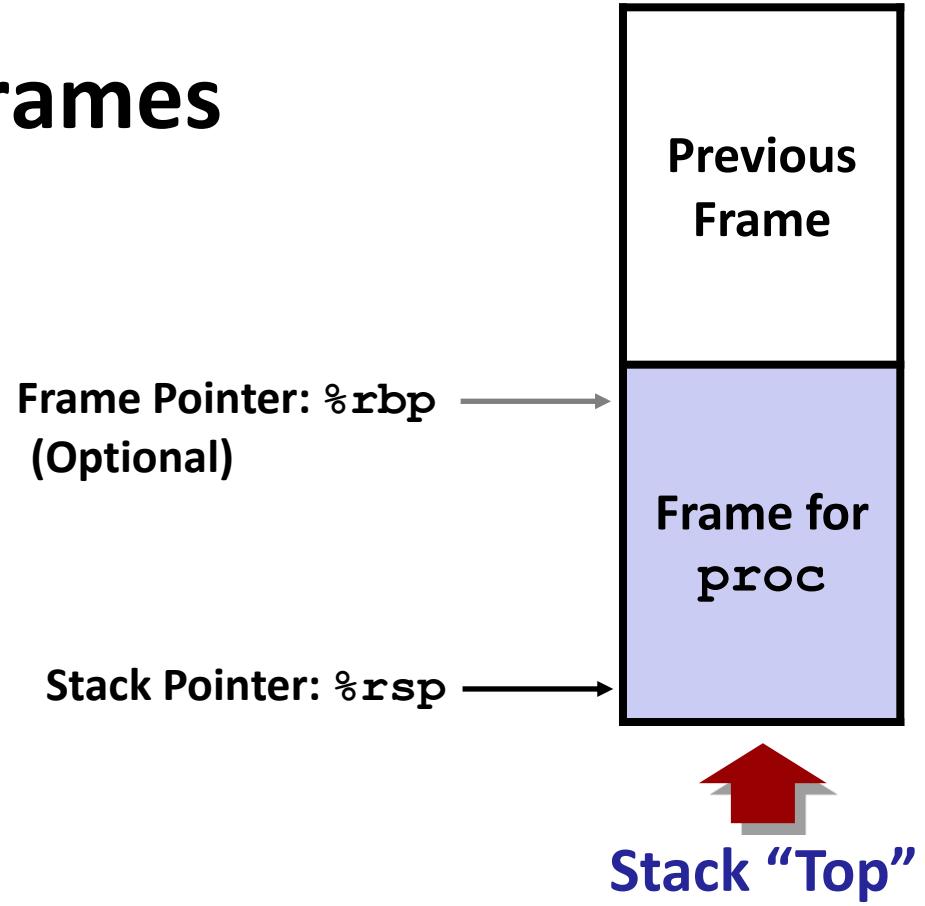
Stack Frames

- **Contents**

- Return information
- Local storage (if needed)
- Temporary space (if needed)

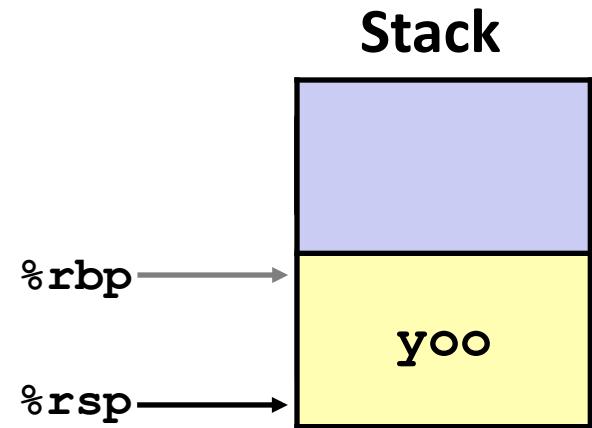
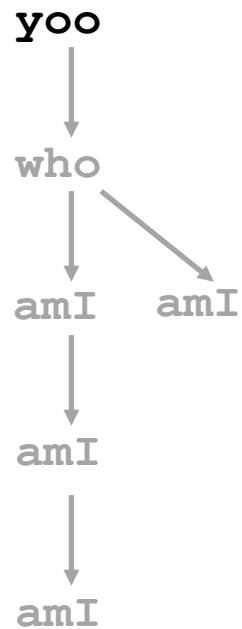
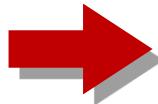
- **Management**

- Space allocated when enter procedure
 - “Set-up” code
 - Includes push by **call** instruction
- Deallocated when return
 - “Finish” code
 - Includes pop by **ret** instruction

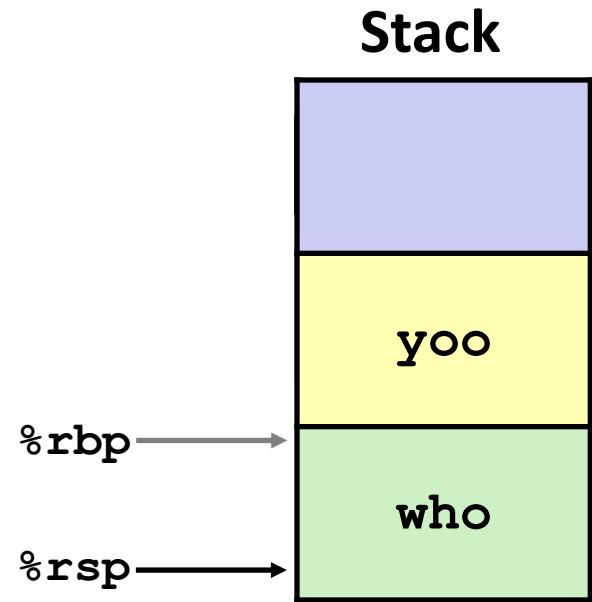
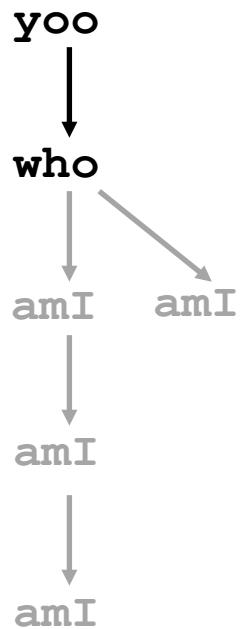
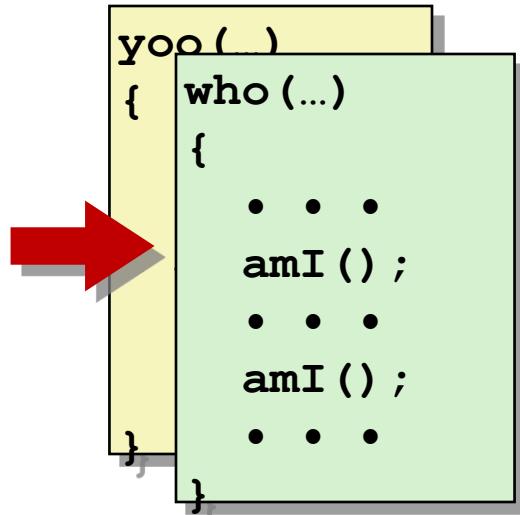


Example

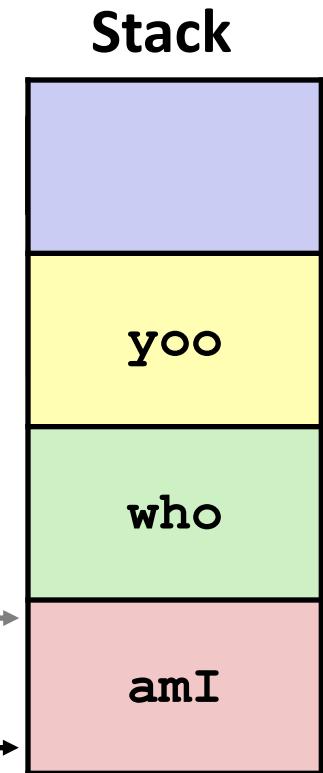
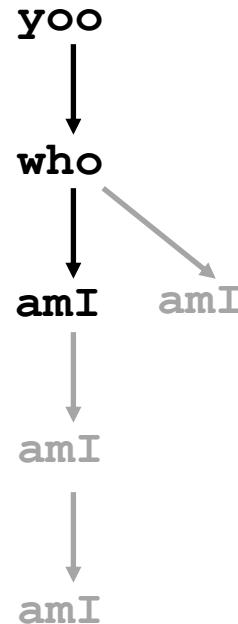
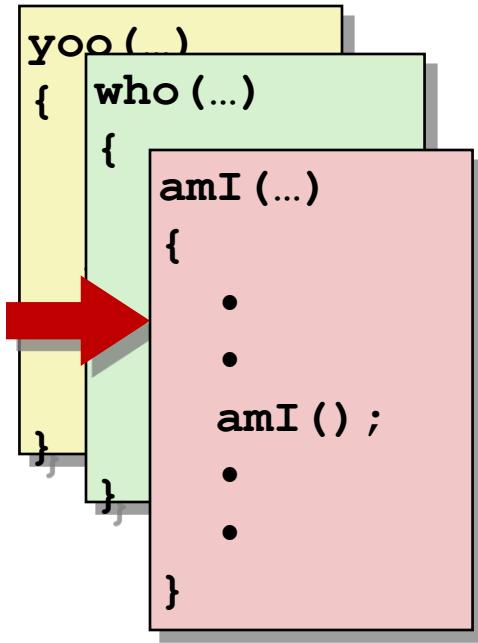
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



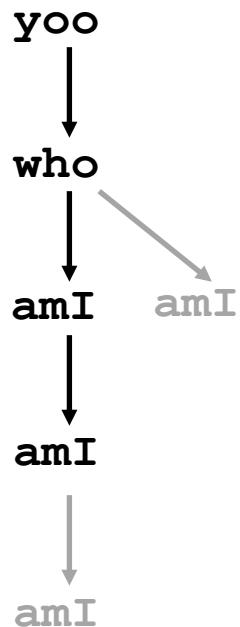
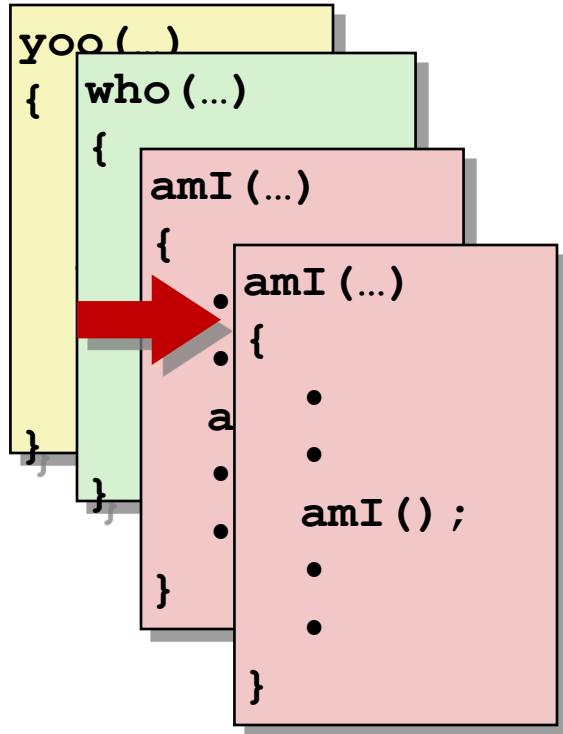
Example



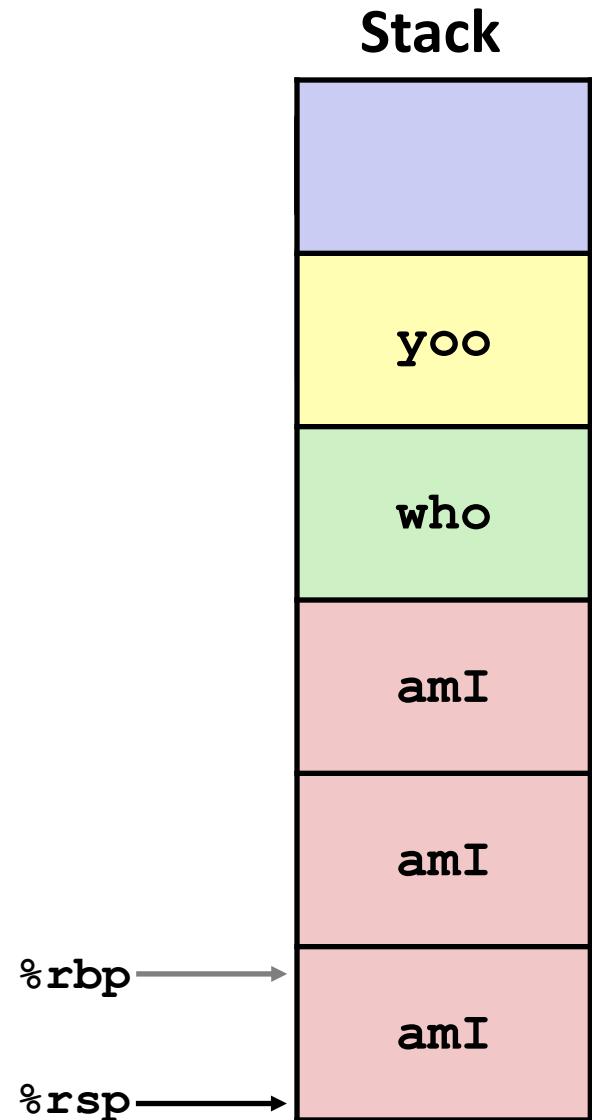
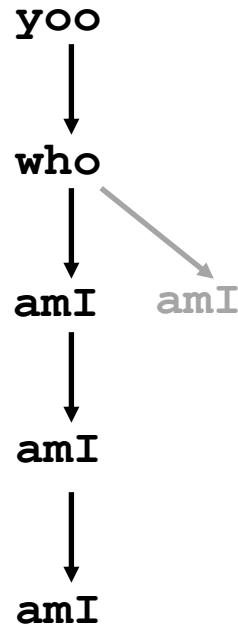
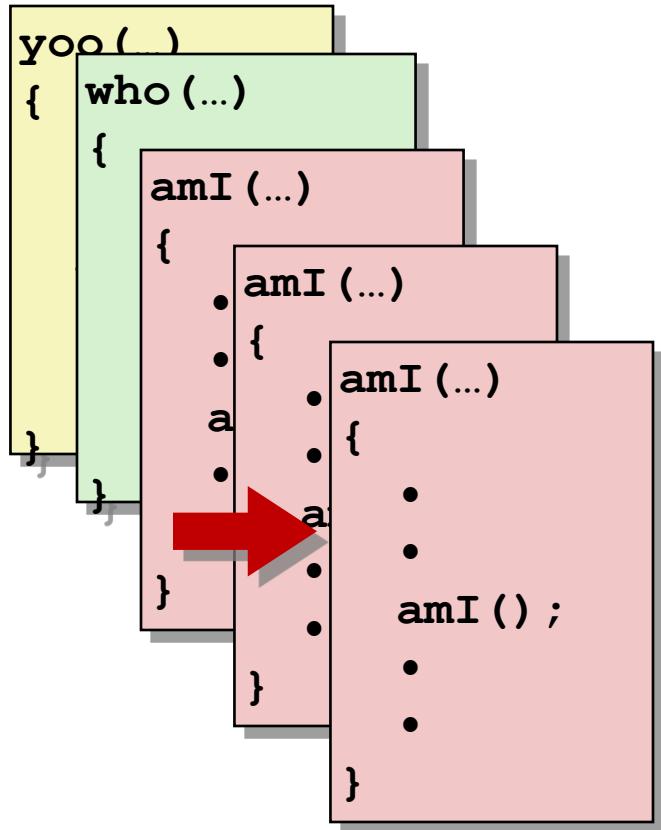
Example



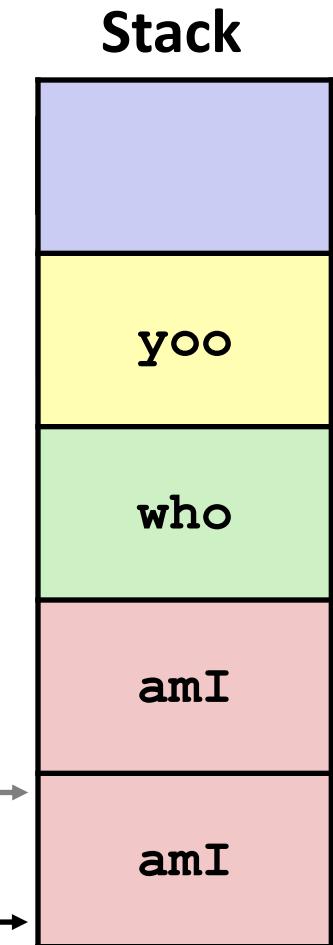
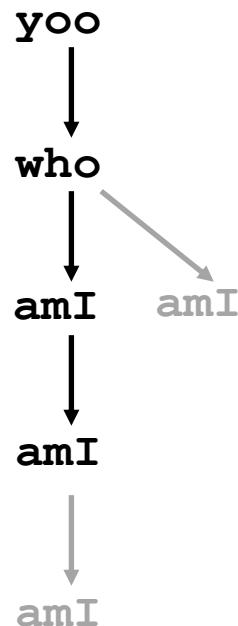
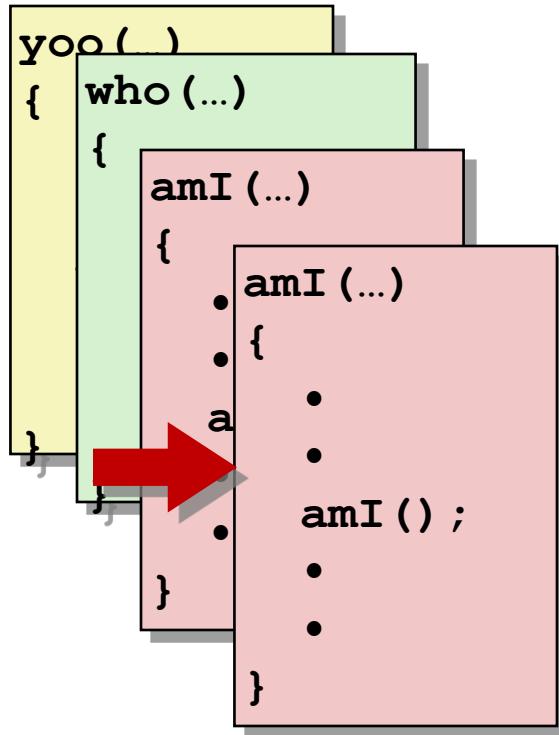
Example



Example



Example

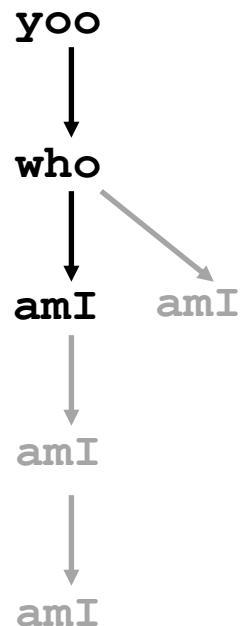


Example

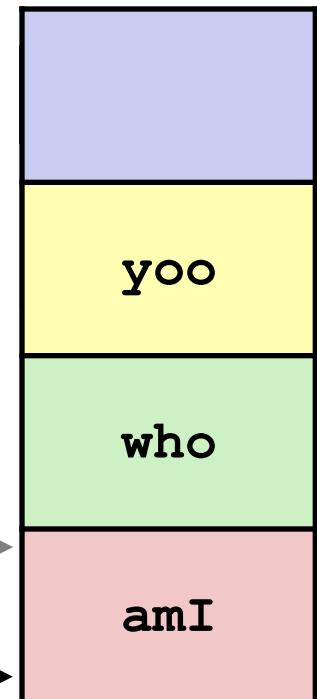
The diagram illustrates nested function scopes using three colored boxes and a red arrow:

- Outer Scope (Yellow Box):** Contains the text "yoo(...)" and a closing brace "}".
- Middle Scope (Green Box):** Contains the text "who(...)" and a closing brace "}".
- Inner Scope (Pink Box):** Contains the text "amI(...)" followed by a brace "{", four bullet points (•), the text "amI();", another brace "{", and finally a closing brace "}".

A large red arrow points from the outer scope towards the middle scope, indicating the flow of execution or the nesting of functions.



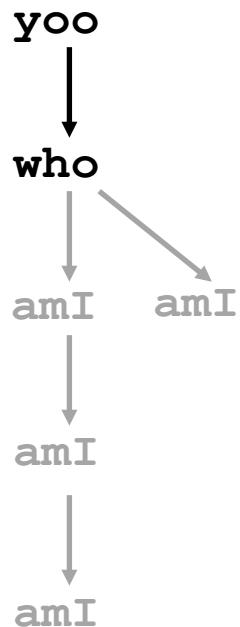
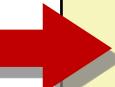
Stack



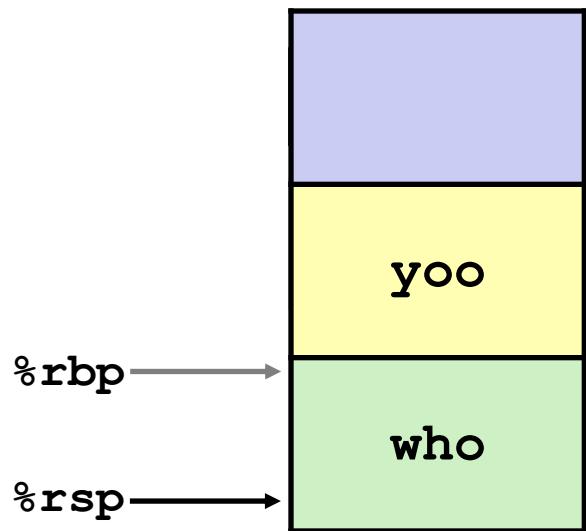
Example

```
yoo()
{
    who(...)

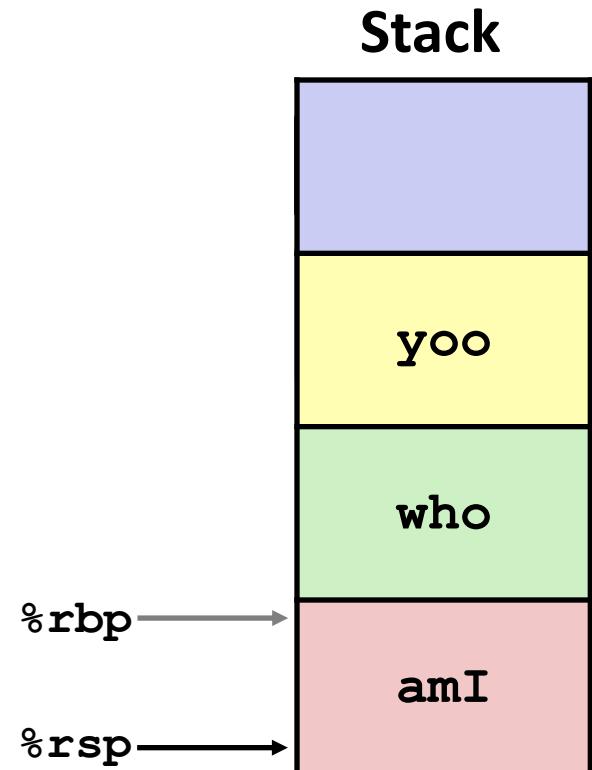
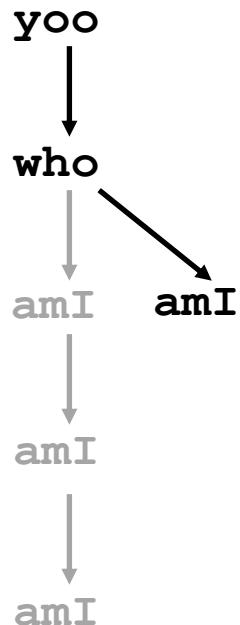
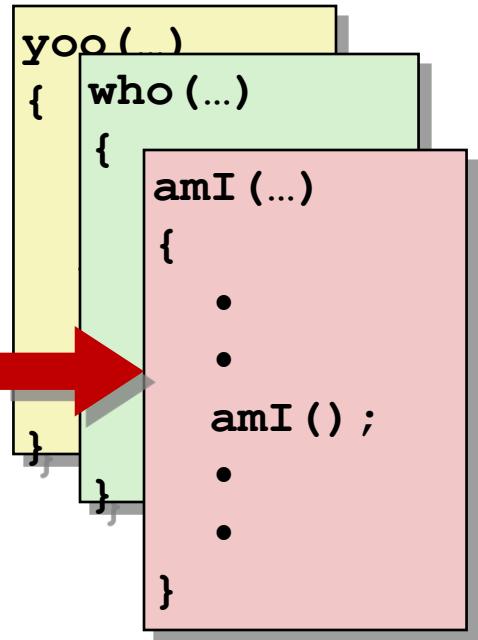
    {
        . . .
        amI();
        . . .
        amI();
        . . .
    }
}
```



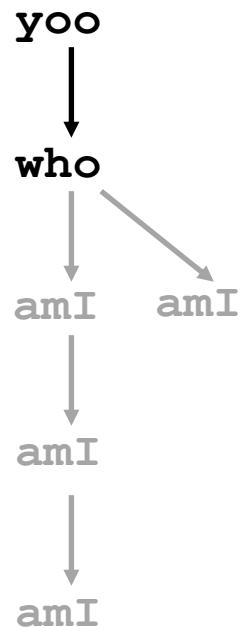
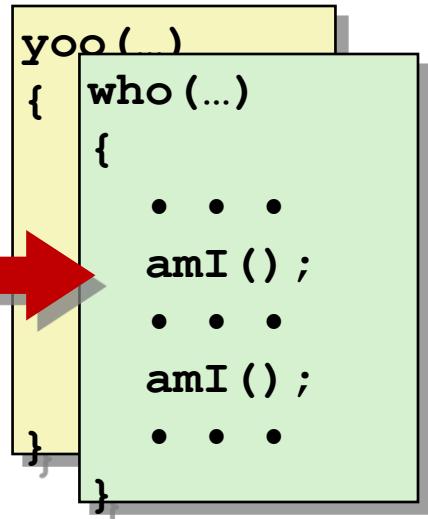
Stack



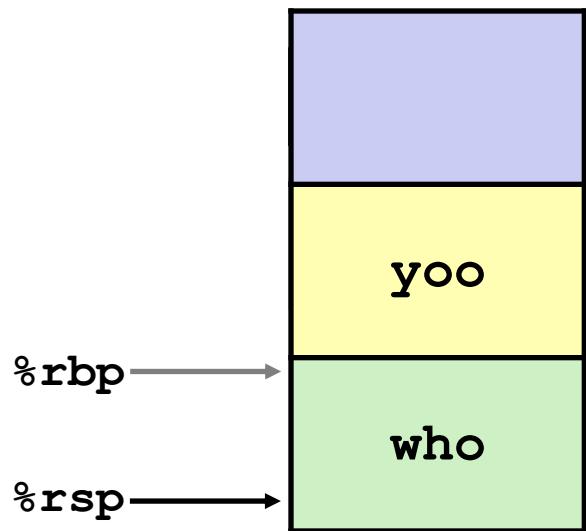
Example



Example

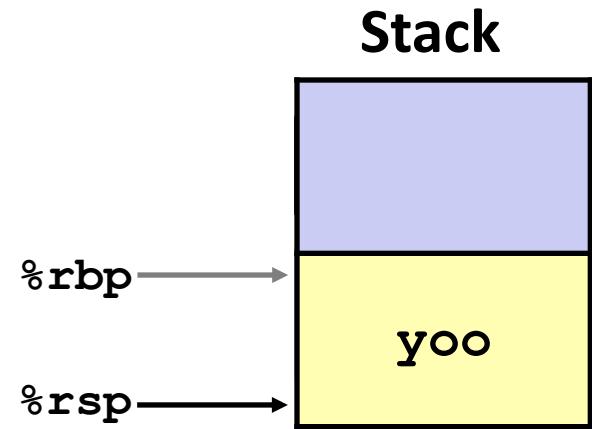
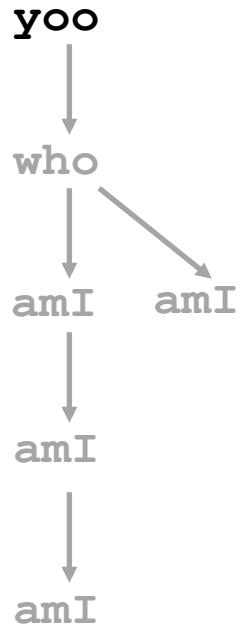


Stack



Example

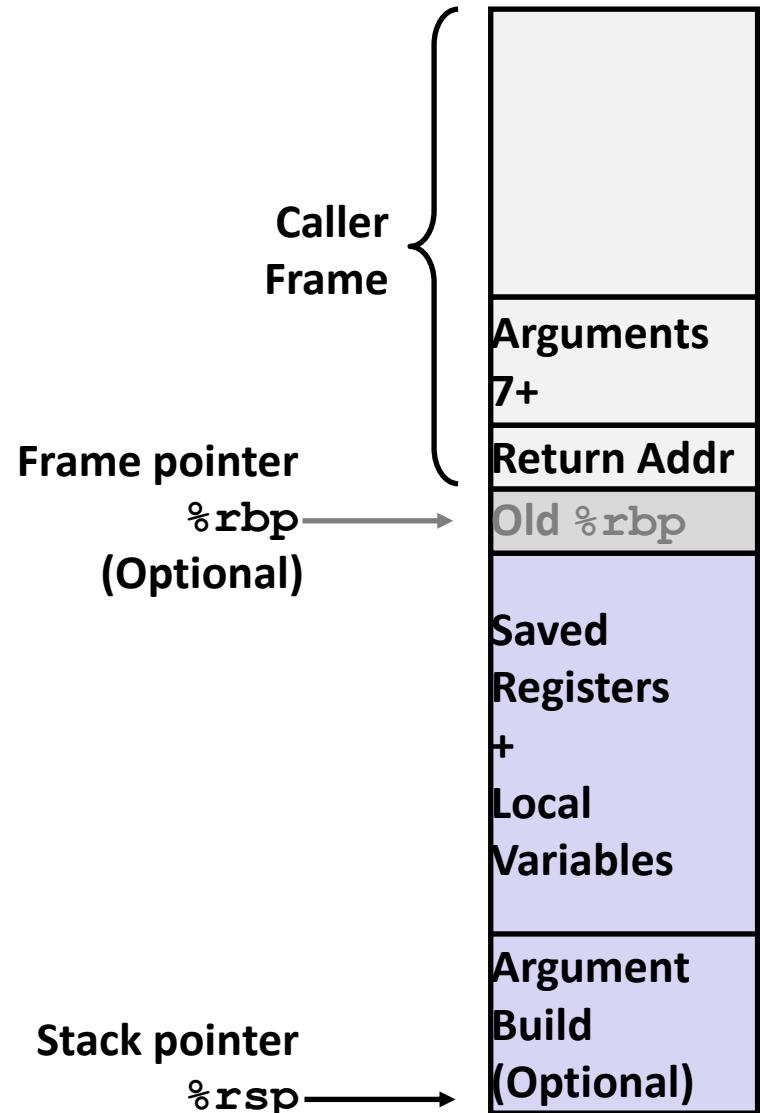
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



x86-64 Stack Frame

- **Current Stack Frame (“Top” to Bottom)**

- “Argument build:”
Parameters for function about to call
- Local variables
If can’t keep in registers
- Saved register context
- Old frame pointer (optional)



- **Caller Stack Frame**

- Return address
 - Pushed by **call** instruction
- Arguments for this call

Example: incr

```
long incr(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

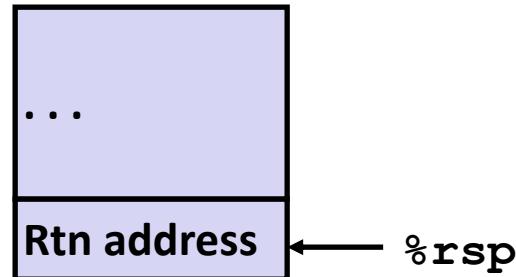
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
%rdi	Argument p
%rsi	Argument val , y
%rax	x , Return value

Example: Calling `incr` #1

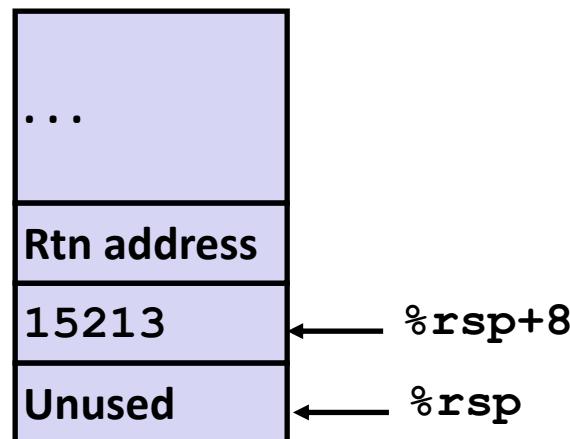
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

Initial Stack Structure



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Resulting Stack Structure



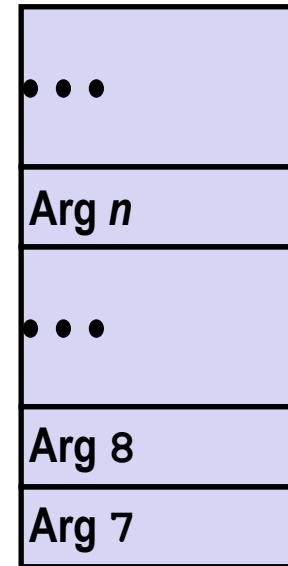
Procedure Data Flow

Registers

- First 6 arguments



Stack



- Return value



- Only allocate stack space when needed

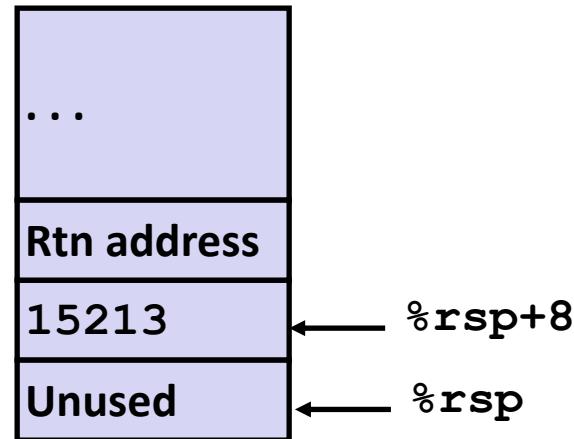
Registers %rbx, %rbp and %r12-r15 are **callee-save registers**, meaning that they are saved across function calls.

Example: Calling incr #2

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure



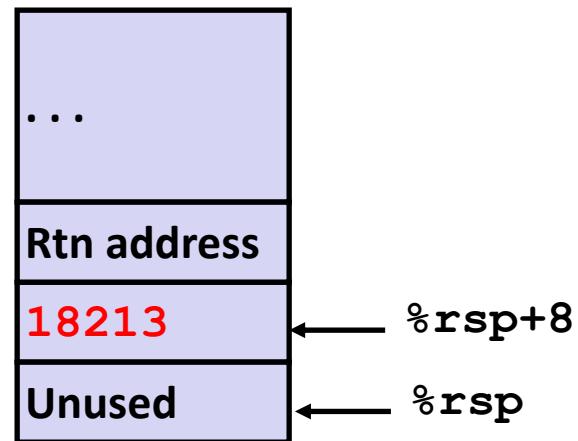
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #3

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure

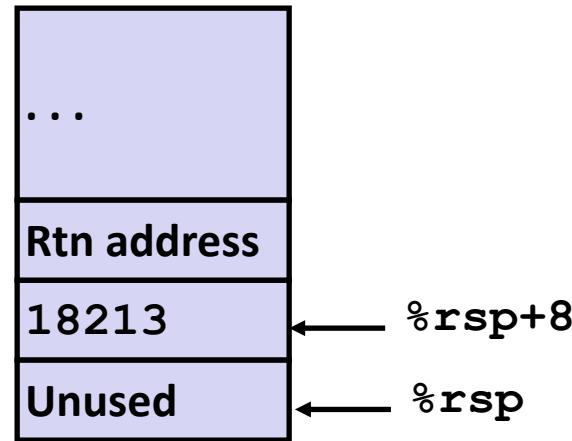


Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

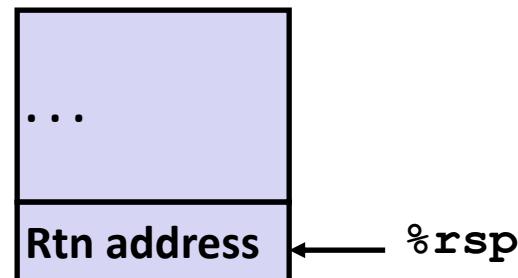
Stack Structure



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Register	Use(s)
%rax	Return value

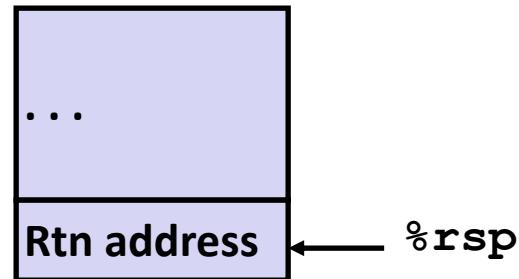
Updated Stack Structure



Example: Calling `incr` #5

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

Updated Stack Structure



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Register	Use(s)
%rax	Return value

Final Stack Structure

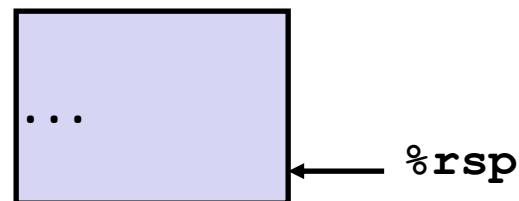


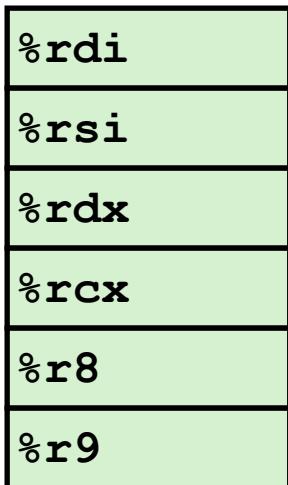
Figure 3.31 Example of procedure definition and call

-

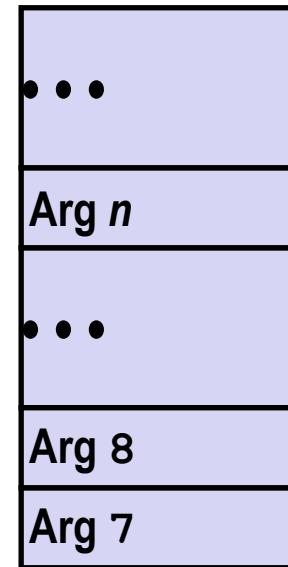
Procedure Data Flow

Registers

- First 6 arguments



Stack



- Return value



- Only allocate stack space when needed

Registers %rbx, %rbp and %r12–r15 are **callee-save registers**, meaning that they are saved across function calls.

Register Saving Conventions

- Set of registers act as a single resource shared by all procedures
- When a **caller** procedure calls another procedure (called **callee**), the callee does not overwrite some register values
- X86-64 adopts a uniform set of conventions for register usage that must be respected by all procedures
- When a procedure **P** calls procedure **Q**, **Q** must preserve the values of **callee-saved** registers.
 - This is so they have same values when Q returns to P as they did when Q was called.
 - Q preserves a register value by not changing it at all or by pushing the original value on the stack, altering it and then popping the old value from the stack before returning.
 - Pushing of register values has the effect of creating the portion of stack frame labeled “Saved registers”

Register Saving Conventions

- All other registers except for the stack pointer %rsp are classified as **caller-saved** registers
 - Can be modified by any function
 - The calling function P has to first save the data before it makes a call to another function Q

Register Saving Conventions

- When procedure **yoo** calls **who**:
 - **yoo** is the *caller*
 - **who** is the *callee*
- Can register be used for temporary storage?

```
yoo:  
    • • •  
    movq $15213, %rdx  
    call who  
    addq %rdx, %rax  
    • • •  
    ret
```

```
who:  
    • • •  
    subq $18213, %rdx  
    • • •  
    ret
```

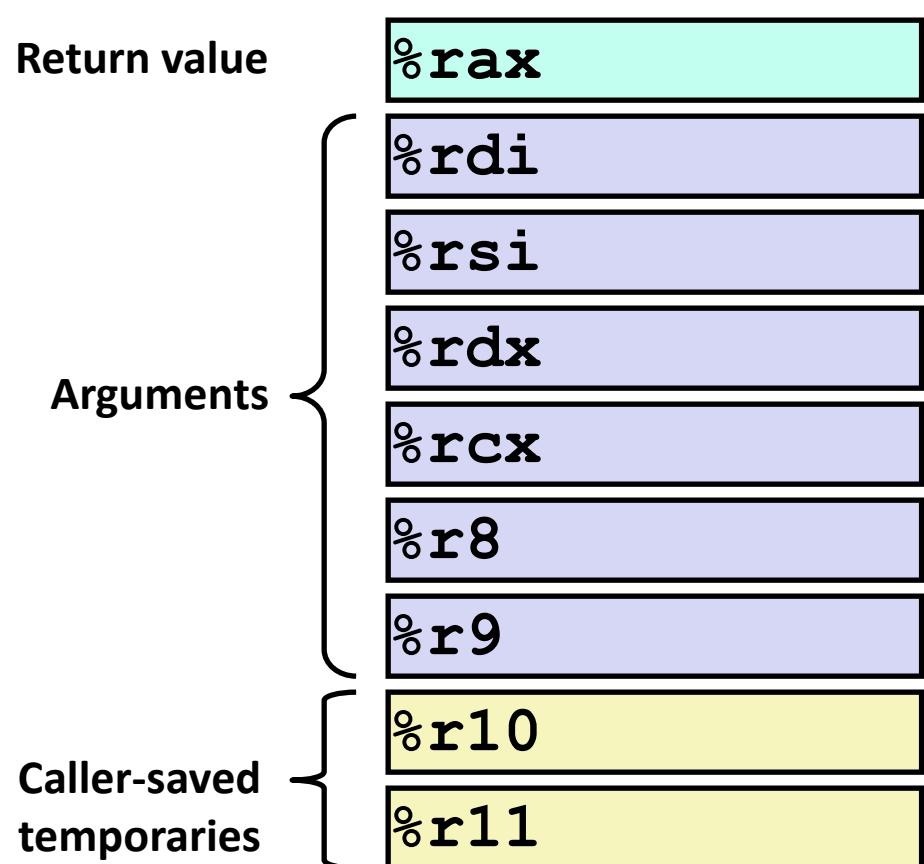
- Contents of register **%rdx** overwritten by **who**
- This could be trouble - need some coordination

Register Saving Conventions

- When procedure **yoo** calls **who**:
 - **yoo** is the *caller*
 - **who** is the *callee*
- Can register be used for temporary storage?
- Conventions
 - ***“Caller Saved”***
 - Caller saves temporary values in its frame before the call
 - ***“Callee Saved”***
 - Callee saves temporary values in its frame before using
 - Callee restores them before returning to caller

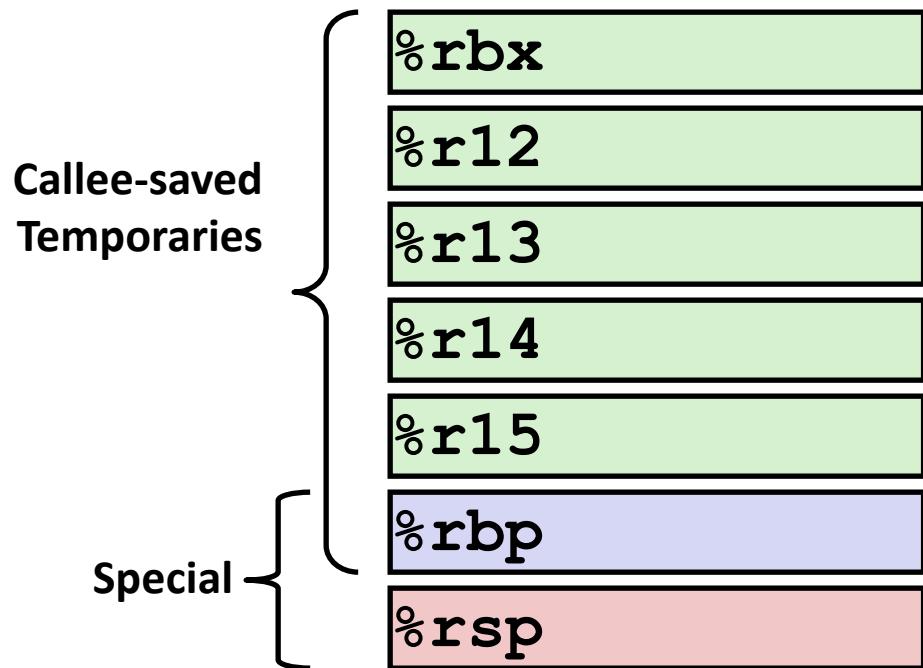
x86-64 Register Usage

- **%rax**
 - Return value
 - Caller-saved
 - Can be modified by procedure
- **%rdi, ..., %r9**
 - Arguments
 - Also caller-saved
 - Can be modified by procedure
- **%r10, %r11**
 - Caller-saved
 - Can be modified by procedure



x86-64 Register Usage

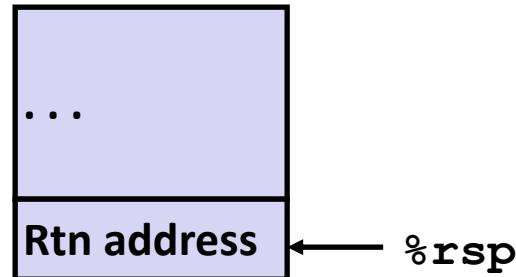
- **%rbx, %r12, %r13, %r14, %r15**
 - Callee-saved
 - Callee must save & restore
- **%rbp**
 - Callee-saved
 - Callee must save & restore
 - May be used as frame pointer
- **%rsp**
 - Special form of callee save
 - Restored to original value upon exit from procedure



Callee-Saved Example #1

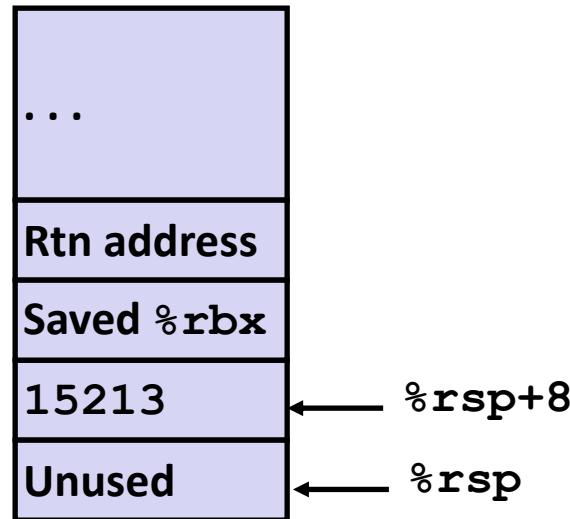
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

Initial Stack Structure



```
call_incr2:  
    pushq  %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```

Resulting Stack Structure

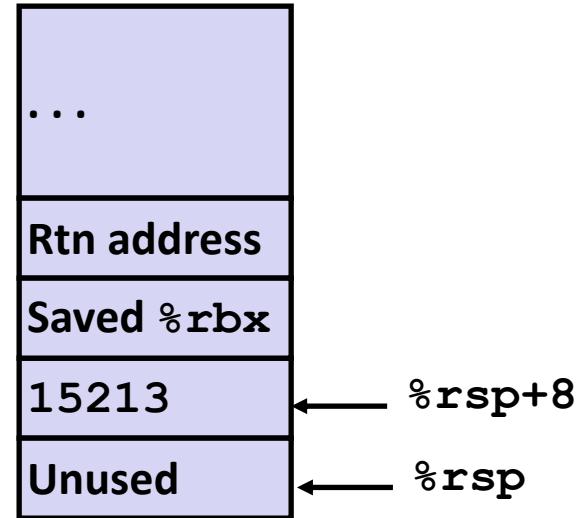


Callee-Saved Example #2

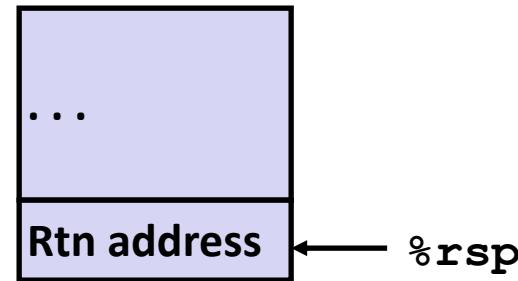
Resulting Stack Structure

```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
call_incr2:  
    pushq  %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```



Pre-return Stack Structure



In summary

- **Call** instruction pushes the return address onto the stack and transfers control to a procedure.
- **Ret** instruction pops the return address off the stack and returns control to that location.

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   $1, %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

```
    rep; ret
```

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   $1, %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

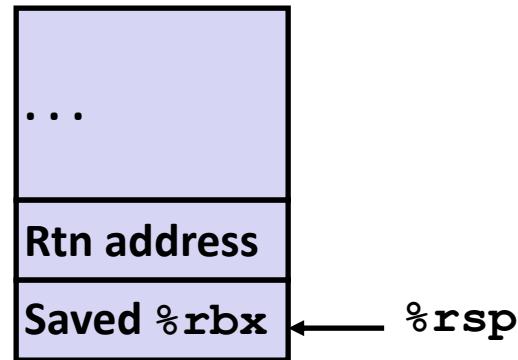
pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   $1, %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

```
    rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    $1, %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

```
    rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    $1, %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

rep; ret

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   $1, %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

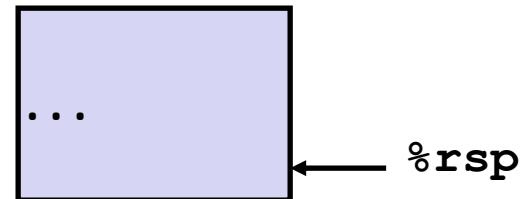
pcount_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   $1, %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret

Register	Use(s)	Type
%rax	Return value	Return value



Observations About Recursion

- Handled Without Special Consideration
 - Stack frames mean that each function call has private storage
 - Saved registers & local variables
 - Saved return pointer
 - Register saving conventions prevent one function call from corrupting another's data
 - Stack discipline follows call / return pattern
 - If P calls Q, then Q returns before P
 - Last-In, First-Out
- Also works for mutual recursion
 - P calls Q; Q calls P

Computer Systems Organization

Topic 3 Contd.

Based on chapter 3 from Computer Systems
by Randal E. Bryant and David R. O'Hallaron

Processor State (x86-64, Partial)

- Information about currently executing program
 - Temporary data (`%rax`, ...)
 - Location of runtime stack (`%rsp`)
 - Location of current code control point (`%rip`, ...)
 - Status of recent tests (CF, ZF, SF, OF)

Current stack top

Registers

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip` Program Counter or
Instruction pointer

CF

ZF

SF

OF

Condition codes

Condition Codes (Implicit Setting)

- Single bit registers
 - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
 - **ZF** Zero Flag **OF** Overflow Flag (for signed)
- Implicitly set (think of it as side effect) by arithmetic operations
 - Example: **addq Src, Dest** $\leftrightarrow t = a+b$
 - **CF set** if carry out from most significant bit (unsigned overflow)
 - **ZF set** if $t == 0$
 - **SF set** if $t < 0$ (as signed)
 - **OF set** if two's-complement (signed) overflow
 $(a>0 \ \&\& \ b>0 \ \&\& \ t<0) \ || \ (a<0 \ \&\& \ b<0 \ \&\& \ t>=0)$
- Not set by **leaq** instruction (since intended to be used in address computations)

Example

- Carry flag enables numbers larger than a single ALU width to be added by carrying a binary digit
- Unsigned addition: $1111 + 0111 = 10110$, CF = 1
- Unsigned addition: $0111 + 0001 = 1000$, CF = 0
- Borrow flag for subtraction if borrow value (represented using Carry flag)
- Unsigned subtraction: $0000 - 0001 = 1111$, CF = 1
- Unsigned subtraction: $1000 - 0001 = 0111$, CF = 0

Example

- Overflow flag is set when the most significant bit (i.e., sign bit) is changed by adding two numbers with the same sign (or subtracting two numbers with opposite signs). Overflow cannot occur when the sign of two addition operands are different.
- Signed addition: $1111 + 1000 = 10111$, CF = 1, SF = 0, OF = 1
- Signed addition: $0111 + 0111 = 1110$, CF = 0, SF = 1, OF = 1
- Signed addition: $1111 + 1111 = 11110$, CF = 1, SF = 1, OF = 0
- Overflow flag is meaningless for unsigned numbers and normally ignored. It is set for signed numbers so the program can be aware of the problem and mitigate or signal an error.
- Most instruction sets do not distinguish between signed and unsigned operands. Generate both (signed) overflow and (unsigned) carry flags on every operation, and allow to pick later whichever is of interest.

Compare Instruction

- Explicit setting of conditional code by Compare instruction
 - `cmpq Src2, Src1`
 - `cmpq b, a` like computing $a - b$ without setting destination
 - **CF set** if carry out from most significant bit (used for unsigned comparisons)
 - **ZF set** if $a == b$
 - **SF set** if $(a - b) < 0$ (as signed)
 - **OF set** if two's-complement (signed) overflow
$$(a>0 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \ \|\ (a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0)$$

Example

- Unsigned subtraction: $0000 - 0001 = 1111$, CF = 1
- Unsigned subtraction: $1000 - 0001 = 0111$, CF = 0
- Signed subtraction: $0001 - 1000 = 1001$, CF = 1, SF = 1, OF = 1
- Signed subtraction: $0011 - 1100 = 0111$, CF = 1, SF = 0, OF = 0
- Signed subtraction: $1000 - 0001 = 0111$, CF = 0, SF = 0, OF = 1
- Signed subtraction: $1001 - 0001 = 1000$, CF = 0, SF = 1, OF = 0

Explicitly Setting Condition Codes: Test

- Explicit setting of conditional codes by Test instruction
 - `testq Src2, Src1`
 - `testq b, a` like computing `a&b` without setting destination (performs bitwise AND)
 - Sets condition codes based on value of `Src1 & Src2`
 - Useful to have one of the operands be a mask
 - **ZF set when $a \& b == 0$**
 - **SF set when $a \& b < 0$**

Example

- `testq %rax %rax` sets ZF or SF – hence used to test whether a value is negative, zero or positive
- One of the values can be a mask e.g., test the last bit
 - Mask would be 00..01 – If ZF set, last bit is 0
 - In general, can test nth bit or a subset of the expression in general

SET Instructions

- Rather than reading the conditional codes directly, set a single byte to 0 or 1 depending on some combination of the condition codes.
 - SET instructions are useful to model this
- For conditional codes set using `cmpq` i.e., $t = a-b$
 - If a , b and t are integers represented in 2's complement form
 - Consider `sete` or “set when equal” - when $a == b$, $t = 0$ and hence zero flag indicates equality

SET Instructions

- Consider setl or “set when less”
 - When no overflow occurs (OF set to 0), we will have $a < b$ when $a-b < 0$ indicated by having SF set to 1. Similarly, we will have $a \geq b$ when $a-b \geq 0$ indicated by having SF set to 0
 - When overflow occurs, we will have $a < b$ when $a-b > 0$ (negative overflow) and $a > b$ when $a-b < 0$ (positive overflow)
 - cannot have overflow when $a = b$
 - In summary , when OF is set to 1, we will have $a < b$ only if SF is set to 0
 - Combining the EXCLUSIVE-OR of the overflow and sign bits provides a test for whether $a < b$
 - Signed comparison tests are based on combinations of SF, CF, OF and ZF

SET Instructions

- For unsigned comparisons of variables a and b, for $t = a-b$, carry flag will be set by CMP instruction when $a-b < 0$ (uses combinations of carry and zero flags)
- Machine code does not distinguish between signed and unsigned values since many arithmetic operations have the same bit level behavior for unsigned and 2's complement arithmetic.
- Some circumstances can need handling of signed vs. unsigned operations e.g., right shifts [Sign extend for Arithmetic (or Signed) Shift while 0 extend for Logical Shift]

Reading Condition Codes

- SetX Instructions
 - Set low-order byte of destination to 0 or 1 based on combinations of condition codes - does not alter remaining 7 bytes

SetX	Condition	Description
sete D	$D \leftarrow ZF$	Equal / Zero
setne D	$D \leftarrow \sim ZF$	Not Equal / Not Zero
sets D	$D \leftarrow SF$	Negative
setns D	$D \leftarrow \sim SF$	Nonnegative
setg D	$D \leftarrow \sim(SF \wedge OF) \& \sim ZF$	Greater (Signed)
setge D	$D \leftarrow \sim(SF \wedge OF)$	Greater or Equal (Signed)
setl D	$D \leftarrow (SF \wedge OF)$	Less (Signed)
setle D	$D \leftarrow (SF \wedge OF) ZF$	Less or Equal (Signed)
seta D	$D \leftarrow \sim CF \& \sim ZF$	Above (unsigned >)
setb D	$D \leftarrow CF$	Below (unsigned <)

x86-64 Integer Registers

%rax	%al	
%rbx	%bl	
%rcx	%cl	
%rdx	%dl	
%rsi	%sil	
%rdi	%dil	
%rsp	%spl	
%rbp	%bp1	
%r8		%r8b
%r9		%r9b
%r10		%r10b
%r11		%r11b
%r12		%r12b
%r13		%r13b
%r14		%r14b
%r15		%r15b

- Can reference low-order byte

Reading Condition Codes

- SetX Instructions:
 - Set single byte based on combination of condition codes
 - One of addressable byte registers
 - Does not alter remaining bytes
 - Typically use **movzbl** to finish job
 - Move zero-extended byte to double word
 - Set upper 32 bits to 0

```
int gt (long x, long y)
{
    return x > y;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
cmpq    %rsi, %rdi    # Compare y:x
setg    %al             # Set when >
movzbl  %al, %eax      # Zero rest of %eax (and %rax)
ret
```

Details on ret

- By convention, %rax is used to store a function's return value, if it exists and is no more than 64 bits long.
- Registers %rbx, %rbp, and %r12-r15 are callee-save registers, meaning that they are saved across function calls.
- Additionally, %rdi, %rsi, %rdx, %rcx, %r8, and %r9 are used to pass the first six integer or pointer parameters to called functions.

Conditional Branches: Jumping

- **jX Instructions**

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF) ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Conditional Branch Example (Old Style)

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

absdiff:

```
    cmpq    %rsi, %rdi    # y:x
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:      # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Expressing with Goto Code

- C allows goto statement
- Jump to position designated by label

```
long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
ntest = !Test;  
if (ntest) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

- Conditional Move Instructions
 - Instruction supports:
 - if (Test) Dest \leftarrow Src
 - Supported in post-1995 x86 processors
 - GCC tries to use them
 - But, only when known to be safe
- Why?
 - Branches are very disruptive to instruction flow through pipelines
 - Conditional moves do not require control transfer

C Code

```
val = Test  
    ? Then_Expr  
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

Conditional Move Example

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq    %rdi, %rax    # x
    subq    %rsi, %rax    # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx    # eval = y-x
    cmpq    %rsi, %rdi    # y:x
    cmovle %rdx, %rax    # if <=, result = eval
    ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Loops: “Do-While” Loop Example

C Code

```
long fact_do
(long n) {
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

Goto Version

```
long fact_do_goto
(long x) {
    long result = 1;
loop:
    result *= n;
    n = n-1;
    if(n > 1) goto loop;
    return result;
}
```

- Compute n factorial
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

```
long fact_goto
(long x) {
    long result = 1;
loop:
    result *= n;
    n = n-1;
    if(n > 1) goto loop;
    return result;
}
```

Register	Use(s)
%rdi	n
%rax	result

```
        movl    $1, %eax      # result = 1
.L2:                           # loop:
        imulq   %rdi, %rax
        subq    $1, %rdi      # Decrement n
        cmpq    $1, %rdi      # Compare n:1
        jg     .L2             # if >, goto loop
        rep; ret
```

General “Do-While” Translation

C Code

```
do  
  Body  
  while (Test);
```

Goto Version

```
loop:  
  Body  
  if (Test)  
    goto loop
```

- **Body:** {
 Statement₁;
 Statement₂;
 ...
 Statement_n;
}

General “While” Translation #1

- “Jump-to-middle” translation

While version

```
while (Test)
    Body
```



Goto Version

```
goto test;
loop:
    Body
test:
    if (Test)
        goto loop;
done:
```

While Loop Example #1

C Code

```
long fact_while
(long n) {
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

Jump to Middle

```
long fact_while_jtm_goto
(long n) {
    long result = 1;
    goto test;
loop:
    result *= n;
    n = n-1;
test:
    if(n > 1) goto loop;
    return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General “While” Translation #2

While version

```
while (Test)
    Body
```



Do-While Version

```
if (!Test)
    goto done;
do
    Body
    while (Test);
done:
```

- “Do-while” translation
- Used with -O1 (higher level of optimization in GCC)

Goto Version

```
if (!Test)
    goto done;
loop:
    Body
    if (Test)
        goto loop;
done:
```



While Loop Example #2

C Code

```
long fact_while
(long n) {
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

Do-While (or Guarded Do) Version

```
long fact_while_gd_goto
(long n) {
    long result = 1;
    if (n <= 1) goto done;
loop:
    result *= n;
    n = n-1;
    if(n != 1) goto loop;
done:
    return result;
}
```

- Compare to do-while version of function
- Initial condition guards entrance to loop

“For” Loop Form

General Form

```
for (Init; Test; Update)  
    Body
```

```
long fact_for  
    (long n)  
{  
    long i;  
    long result = 1;  
    for (i = 2; i <= n; i++)  
        result *= i;  
    return result;  
}
```

Init

```
i = 2
```

Test

```
i <= n
```

Update

```
i++
```

Body

```
result *= i;
```

“For” Loop → While Loop

For Version

```
for (Init; Test; Update)
```

Body



While Version

```
Init;
```

```
while (Test) {
```

Body

Update;

```
}
```

For-While Conversion

Init

```
i = 2
```

Test

```
i <= n
```

Update

```
i++
```

Body

```
result *= i
```

```
long fact_for_while  
(long n)  
{  
    long i = 2;  
    long result = 1;  
    while (i <= n)  
    {  
        result *= i;  
        i++;  
    }  
    return result;  
}
```

“For” Loop Do-While Conversion

C Code

```
long fact_for (long n)
{
    long i;
    long result = 1;
    for (i = 2; i <= n; i++)
    {
        result *= i;
    }
    return result;
}
```

Goto Version

```
long fact_for_jm_goto
(long n) {
    long i = 2;
    long result = 1;
    goto test;
loop:
    result *= i;
    i++;
test:
    if (i <= n)
        goto loop;
    return result;
}
```

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement: An example

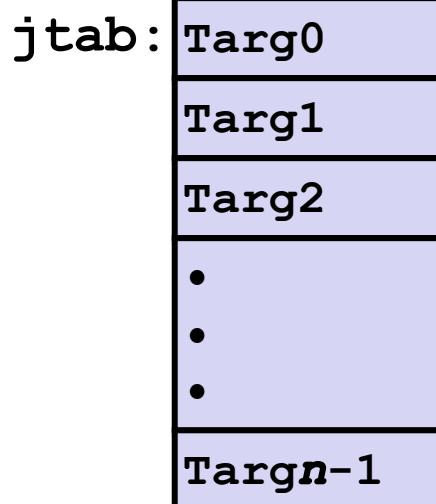
- Multiple case labels
 - Here: 5 & 6
- Fall through cases
 - Here: 2
- Missing cases
 - Here: 4

Jump Table Structure

Switch Form

```
switch(x) {  
    case val_0:  
        Block 0  
    case val_1:  
        Block 1  
        . . .  
    case val_{n-1}:  
        Block n-1  
}
```

Jump Table



Jump Targets

Targ0:

Code Block 0

Targ1:

Code Block 1

Targ2:

Code Block 2

•
•
•

Targ{n-1}:

Code Block n-1

Translation (Extended C)

```
goto *JTab[x];
```

Jump Table jt

- Array where entry i is the address of a code segment implementing the action the program should take when the switch index equals i
- Advantage of using jt is the time taken to perform the switch is independent of the number of switch cases
- Jt used when ≥ 4 cases

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

`switch_eg:`

```
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja     .L8 # if x > 6
    jmp    * .L4(,%rdi,8)
```

What range of values
takes default?

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Note that **w** not
initialized here

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi      # x:6
    ja     .L8          # Use default
    jmp    * .L4(,%rdi,8) # goto *JTab[x]
```

Jump table

```
.section  .rodata
.align 8   Align address
to multiple of 8
.L4:
    .quad   .L8  # x = 0
    .quad   .L3  # x = 1
    .quad   .L5  # x = 2
    .quad   .L9  # x = 3
    .quad   .L8  # x = 4
    .quad   .L7  # x = 5
    .quad   .L7  # x = 6
```

Indirect
jump



Assembly Setup Explanation

- Table Structure
 - Each target requires 8 bytes
 - Base address at `.L4`
- Jumping
 - **Direct:** `jmp .L8`
 - Jump target is denoted by label `.L8`
 - **Indirect:** `jmp * .L4(,%rdi,8)`
 - Start of jump table: `.L4`
 - Must scale by factor of 8 (addresses are 8 bytes)
 - Fetch target from effective Address `.L4 + x*8`
 - Only for $0 \leq x \leq 6$

Jump table

```
.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

Jump Table

Jump table

```
.section    .rodata
.align 8
.L4:
.quad      .L8  # x = 0
.quad      .L3  # x = 1
.quad      .L5  # x = 2
.quad      .L9  # x = 3
.quad      .L8  # x = 4
.quad      .L7  # x = 5
.quad      .L7  # x = 6
```

```
switch(x) {
    case 1:          // .L3
        w = y*z;
        break;
    case 2:          // .L5
        w = y/z;
        /* Fall Through */
    case 3:          // .L9
        w += z;
        break;
    case 5:
    case 6:          // .L7
        w -= z;
        break;
    default:         // .L8
        w = 2;
}
```

Code Blocks ($x == 1$)

```
switch(x) {  
    case 1:          // .L3  
        w = y*z;  
        break;  
    . . .  
}
```

```
.L3:  
    movq    %rsi, %rax  # y  
    imulq   %rdx, %rax  # y*z  
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Handling Fall-Through

```
long w = 1;  
.  
.  
switch(x) {  
.  
.case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
.  
.  
}
```

```
case 2:  
    w = y/z;  
    goto merge;
```

```
case 3:  
    w = 1;  
  
merge:  
    w += z;
```

Code Blocks ($x == 2$, $x == 3$)

```
long w = 1;  
.  
.  
switch(x) {  
.  
. . .  
case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
.  
.  
}
```

```
.L5:          # Case 2  
    movq    %rsi, %rax  
    cqto  
    idivq   %rcx      # y/z  
    jmp     .L6        # goto merge  
.L9:          # Case 3  
    movl    $1, %eax    # w = 1  
.L6:          # merge:  
    addq    %rcx, %rax # w += z  
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rcx	Argument z
%rax	Return value

Code Blocks

- cqto: sign extend rax to rdx:rax
- idivq S:
 - Signed divide %rdx:%rax by S
 - Quotient stored in %rax
 - Remainder stored in %rdx
- Figure 3.12 of book

Code Blocks ($x == 5$, $x == 6$, default)

```
switch(x) {  
    . . .  
    case 5: // .L7  
    case 6: // .L7  
        w -= z;  
        break;  
    default: // .L8  
        w = 2;  
}
```

```
.L7:                      # Case 5,6  
    movl $1, %eax      # w = 1  
    subq %rcx, %rax   # w -= z  
    ret  
.L8:                      # Default:  
    movl $2, %eax      # 2  
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rcx	Argument z
%rax	Return value

Summarizing

- C Control
 - if-then-else
 - do-while
 - while, for
 - switch
- Assembler Control
 - Conditional jump
 - Conditional move
 - Indirect jump (via jump tables)
 - Compiler generates code sequence to implement more complex control
- Standard Techniques
 - Loops converted to do-while or jump-to-middle form
 - Large switch statements use jump tables
 - Sparse switch statements may use decision trees (if-elseif-elseif-else)

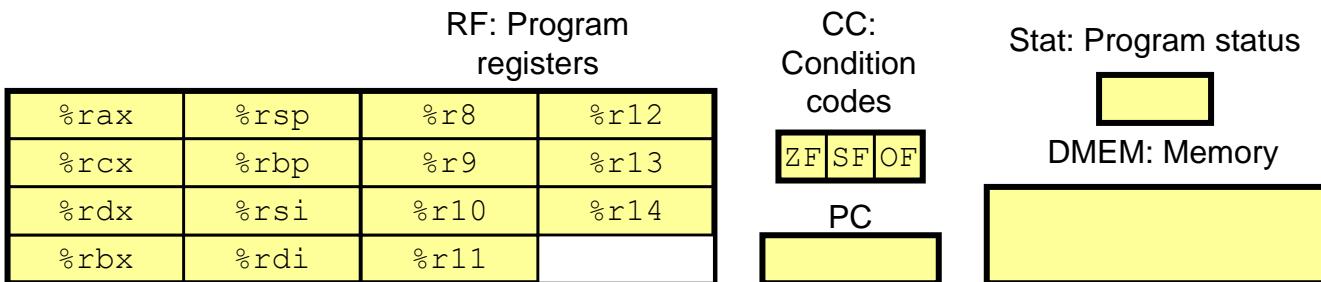
Computer Systems Organization (CS2.201)

Y86-64: INSTRUCTION SET ARCHITECTURE

Deepak Gangadharan
Computer Systems Group (CSG), IIIT Hyderabad

Slide Contents: Based on materials from text books and other public sources

X86-64 Processor State



- Program Registers
 - 15 registers (omit %r15). Each 64 bits
- Condition Codes
 - Single-bit flags set by arithmetic or logical instructions
 - ZF: Zero
 - SF:Negative
 - OF: Overflow
- Program Counter
 - Indicates address of next instruction
- Program Status
 - Indicates either normal operation or some error condition
- Memory
 - Byte-addressable storage array
 - Words stored in little-endian byte order

Y86-64 Instructions

- Subset of x86-64 instruction set
 - includes only 8-byte integer operations
 - fewer addressing modes (second index register and scaling not supported)
 - No transfer of immediate data to memory
 - smaller set of operations

Y86-64 Instruction Set #1

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F	The register order in encoding here is correct - Verified					

The register order
in encoding here is
correct - Verified

X86-64 Instructions

Format

- 1–10 bytes of information read from memory
 - Can determine instruction length from first byte
 - Not as many instruction types, and simpler encoding than with x86-64
- Each accesses and modifies some part(s) of the program state

Y86-64 Instruction Set #2

Byte	0	1	2	3	4	5	6	7	
halt	0	0							rmmovq 2 0
nop	1	0							cmove 2 1
cmovXX rA, rB	2	fn	rA	rB					cmovl 2 2
irmovq V, rB	3	0	F	rB	V				cmovne 2 3
rmmovq rA, D(rB)	4	0	rA	rB	D				cmovge 2 4
mrmovq D(rB), rA	5	0	rA	rB	D				cmovg 2 5
OPq rA, rB	6	fn	rA	rB					
jXX Dest	7	fn			Dest				
call Dest	8	0			Dest				
ret	9	0							
pushq rA	A	0	rA	F					
popq rA	B	0	rA	F					

Y86-64 Instruction Set #3

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D	addq	6	0		
OPq rA, rB	6	fn	rA	rB		subq	6	1		
jXX Dest	7	fn			Dest	andq	6	2		
call Dest	8	0			Dest	xorq	6	3		
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Y86-64 Instruction Set #4

Byte	0	1	2	3	4	5	6	7	
halt	0	0							jmp [7] [0]
nop	1	0							jle [7] [1]
cmoveXX rA, rB	2	fn	rA	rB					jl [7] [2]
irmovq V, rB	3	0	F	rB	V				je [7] [3]
rmmovq rA, D(rB)	4	0	rA	rB	D				jne [7] [4]
mrmovq D(rB), rA	5	0	rA	rB	D				jge [7] [5]
OPq rA, rB	6	fn	rA	rB					jk [7] [6]
jXX Dest	7	fn			Dest				
call Dest	8	0			Dest				
ret	9	0							
pushq rA	A	0	rA	F					
popq rA	B	0	rA	F					

Encoding Registers

Each register has 4-bit ID

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7
%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

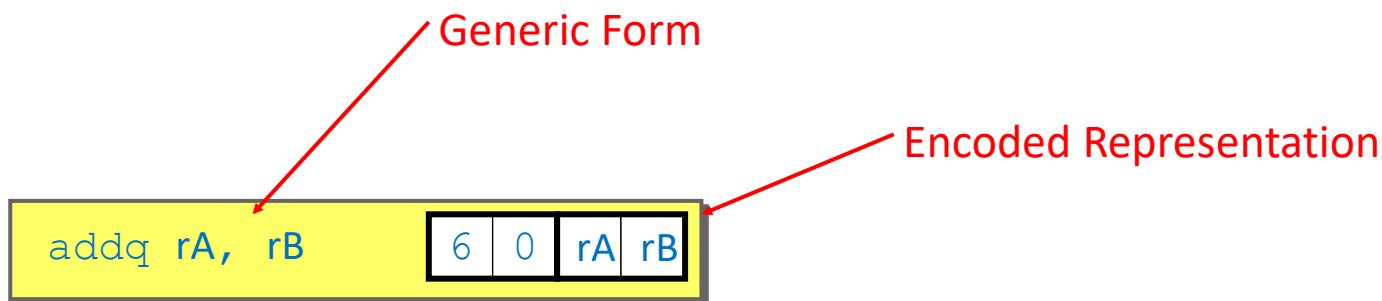
- Same encoding as in x86-64

Register ID 15 (0xF) indicates “no register”

- Will use this in our hardware design in multiple places

Instruction Example

Addition Instruction



- Add value in register rA to that in register rB
 - Store result in register rB
 - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., `addq %rax, %rsi` Encoding: 60 06
- Two-byte encoding
 - First indicates instruction type
 - Second gives source and destination registers

Arithmetic and Logical Operations

Instruction Code Function Code

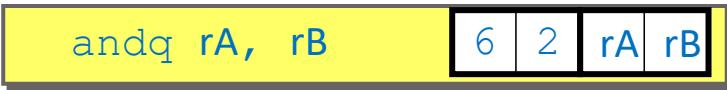
Add



Subtract (rA from rB)



And



Exclusive-Or



- Refer to generically as “OPq”
- Encodings differ only by “function code”
 - Low-order 4 bits in first instruction word
- Set condition codes as side effect

Move Operations

rrmovq rA, rB	<table border="1"><tr><td>2</td><td>0</td><td>rA</td><td>rB</td></tr></table>	2	0	rA	rB	Register → Register
2	0	rA	rB			
irmovq V, rB	<table border="1"><tr><td>3</td><td>0</td><td>F</td><td>rB</td></tr></table>	3	0	F	rB	V
3	0	F	rB			
rmmovq rA, D(rB)	<table border="1"><tr><td>4</td><td>0</td><td>rA</td><td>rB</td></tr></table>	4	0	rA	rB	D
4	0	rA	rB			
mrmovq D(rB), rA	<table border="1"><tr><td>5</td><td>0</td><td>rA</td><td>rB</td></tr></table>	5	0	rA	rB	D
5	0	rA	rB			

- Like the x86-64 movq instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

X86-64

```
movq $0xabcd, %rdx
```

Y86-64

```
irmovq $0xabcd, %rdx
```

Encoding:

```
30 F2 cd ab 00 00 00 00 00 00
```

```
movq %rsp, %rbx
```

```
rrmovq %rsp, %rbx
```

Encoding:

```
20 43
```

```
movq -12(%rbp),%rcx
```

```
mrmovq -12(%rbp),%rcx
```

Encoding:

```
50 15 f4 ff ff ff ff ff ff ff ff
```

```
movq %rsi,0x41c(%rsp)
```

```
rmmovq %rsi,0x41c(%rsp)
```

Encoding:

```
40 64 1c 04 00 00 00 00 00 00
```

Conditional Move Instructions

Move Unconditionally

`rrmovq rA, rB`

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

Move When Less or Equal

`cmovele rA, rB`

2	1	rA	rB
---	---	----	----

Move When Less

`cmovl rA, rB`

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

Move When Equal

`cmove rA, rB`

2	3	rA	rB
---	---	----	----

Move When Not Equal

`cmovne rA, rB`

2	4	rA	rB
---	---	----	----

Move When Greater or Equal

`cmovge rA, rB`

2	5	rA	rB
---	---	----	----

Move When Greater

`cmovg rA, rB`

2	6	rA	rB
---	---	----	----

- Refer to generically as “`cmovXX`”
- Encodings differ only by “function code”
- Based on values of condition codes
- Variants of `rrmovq` instruction
 - (Conditionally) copy value from source to destination register

Jump Instructions

Jump (Conditionally)



- Refer to generically as “j XX”
- Encodings differ only by “function code” fn
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
 - Unlike PC-relative addressing seen in x86-64

Jump Instructions

Jump Unconditionally



Jump When Less or Equal



Jump When Less



Jump When Equal



Jump When Not Equal



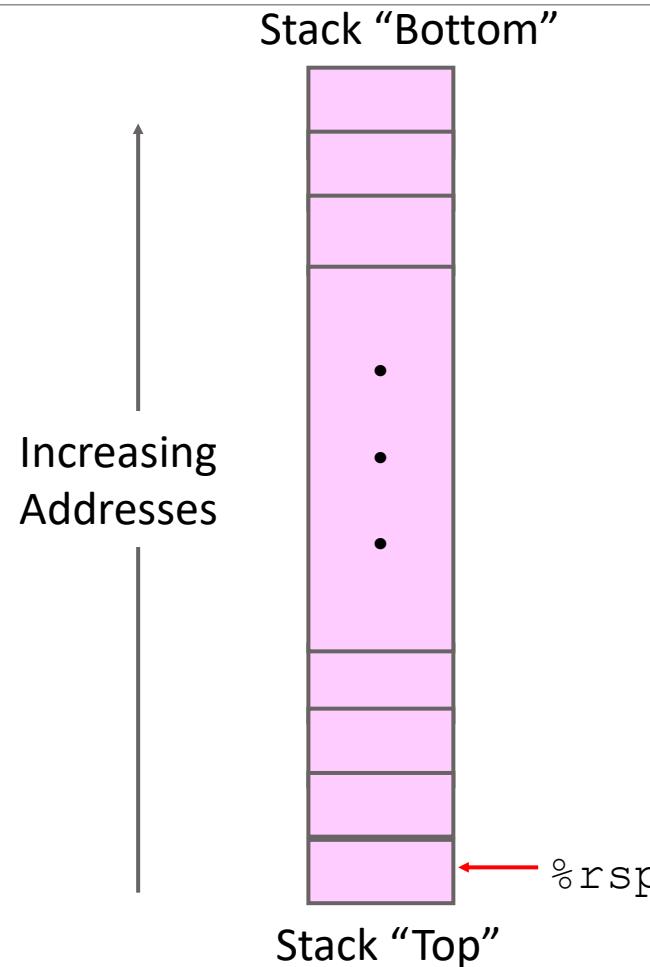
Jump When Greater or Equal



Jump When Greater



Y86-64 Program Stack



- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by `%rsp`
 - Address of top stack element
- Stack grows toward lower addresses
 - Top element is at highest address in the stack
 - When pushing, must first decrement stack pointer
 - After popping, increment stack pointer

Stack Operations



- Decrement `%rsp` by 8
- Store word from `rA` to memory at `%rsp`
- Like x86-64



- Read word from memory at `%rsp`
- Save in `rA`
- Increment `%rsp` by 8
- Like x86-64

Subroutine Call and Return



- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64



- Pop value from stack
- Use as address for next instruction
- Like x86-64

Miscellaneous Instructions



- Don't do anything



- Stop executing instructions
- x86-64 has comparable instruction, but can't execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

Mnemonic	Code
AOK	1

- Normal operation

Mnemonic	Code
HLT	2

- Halt instruction encountered

Mnemonic	Code
ADR	3

- Bad address (either instruction or data) encountered

Mnemonic	Code
INS	4

- Invalid instruction encountered

Desired Behavior

- If AOK, keep going
- Otherwise, stop program execution

Y86-64 program

```
1 long sum(long *start, long count)
2 {
3     long sum = 0;
4     while (count) {
5         sum += *start;
6         start++;
7         count--;
8     }
9     return sum;
10 }
```

x86-64 code

```
long sum(long *start, long count)
start in %rdi, count in %rsi
sum:                                sum = 0
    movl $0, %eax
    jmp .L2
.L3:                                Goto test
    addq (%rdi), %rax
    addq $8, %rdi
    subq $1, %rsi
    .L2:                                loop:
        testq %rsi, %rsi
        jne .L3
    rep; ret
```

Y86-64 code

```
long sum(long *start, long count)
start in %rdi, count in %rsi
sum:                                sum = 0
    irmovq $8,%r8
    irmovq $1,%r9
    xorq %rax,%rax
    andq %rsi,%rsi
    jmp test
loop:                                Goto test
    mrmovq (%rdi),%r10
    addq %r10,%rax
    addq %r8,%rdi
    subq %r9,%rsi
    test:                                test:
        jne loop
    ret
```

Summary

Y86-64 Instruction Set Architecture

- Similar state and instructions as x86-64
- Simpler encodings
- Somewhere between CISC and RISC

How Important is ISA Design?

- Less now than before
 - With enough hardware, can make almost anything go fast

Thank You!

Computer Systems Organization (CS2.201)

PROCESSOR ARCHITECTURE DESIGN – SEQUENTIAL (SECTION 4.3)

Deepak Gangadharan
Computer Systems Group (CSG), IIIT Hyderabad

Slide Contents: Adapted from slides by Randal Bryant

Preliminaries

CPU time = Number of instructions × Clocks per instruction (CPI) × Clock cycle time

$$\text{Clock rate} = \frac{1}{\text{Clock cycle time}}$$

Factors affecting the above parameters:

Clock rate – hardware technology and organization

CPI – organization, ISA and compiler technology

Instruction count – ISA and compiler technology

Sequential Y86-64 Implementation

Sequential Y86-64 implementation

- Let us call the processor SEQ (for sequential processor)
- On each clock cycle, SEQ performs all the steps required to process a complete instruction
- **Result: Very long cycle time and low clock rate**
- **Goal: Improve the sequential implementation by understanding the problems with it**

Sequential Y86 Instruction Stages

Each instruction sequentially goes through following common stages:

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write-back
6. PC update

The processor loops indefinitely, performing the functions in each stage unless any exception condition occurs.

Sequential Y86 Instruction Stages

Why common stages for all instructions?

- Very simple and uniform structure is important when designing hardware → To reduce the footprint of logic on the chip
- One way to minimize complexity is by sharing hardware as much as possible among instructions
- Cost of duplicating block of logic in hardware is much higher than the cost of having multiple copies of code in software

SEQ stages

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

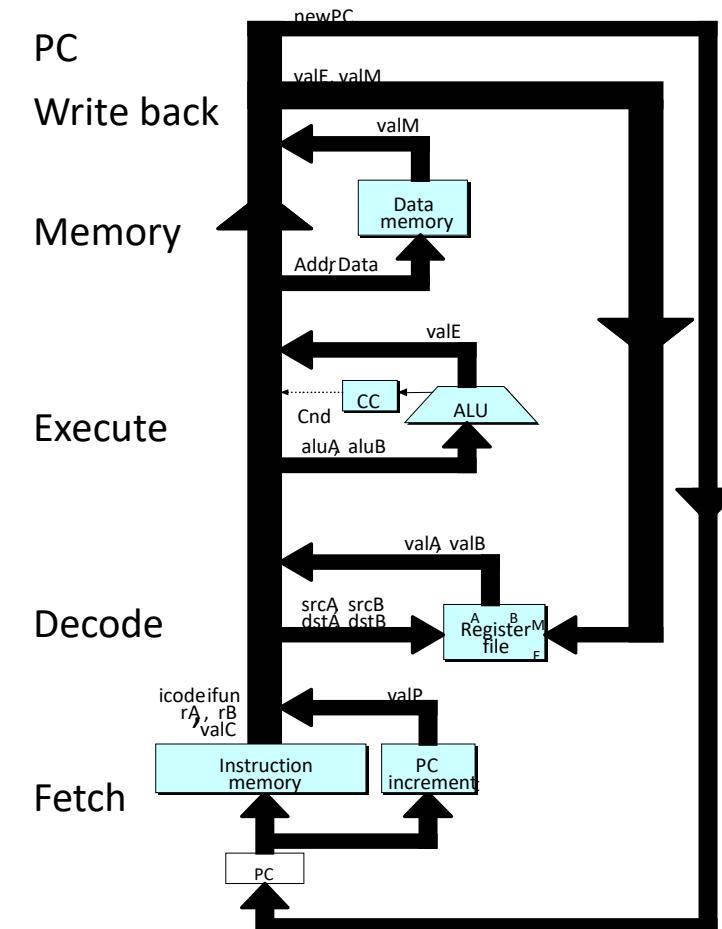
- Read or write data

Write Back

- Write program registers

PC

- Update program counter



SEQ stages

Fetch:

- Reads bytes of an instruction from memory using the PC value as address → Extracts the two 4-bit portions of instruction specifier byte referred to as **icode** and **ifun**
- Possibly fetches the register specifier byte giving one or both of the register operand specifiers rA and rB
- Also possibly fetches an 8-byte constant word valC → Computes valP as the address of the next instruction in the sequence , i.e. $\text{valP} = \text{PC} + \text{length of fetched instruction}$

Decode:

- Reads up to two operands from the register file giving values valA and/or valB
- For some instructions, it reads register %rsp

SEQ stages

Execute:

- ALU either performs operation given by ifun, computes effective address of a memory reference, or increments or decrements the stack pointer. Resulting value → valE
- Condition codes are possibly set
- For a jump instruction, tests condition code and branch condition (referred to by ifun) to determine if branch should be taken or not

SEQ stages

Memory:

- May read or write data from/to memory respectively. Value read referred to as valM.

Write back:

- Writes up to two results to the register file

PC Update:

- PC is set to address of next instruction or valP

Executing Arithmetic/Logic Operation



Fetch

- Read 2 bytes

Decode

- Read operand registers

Execute

- Perform operation
- Set condition codes

Memory

- Do nothing

Write back

- Update register

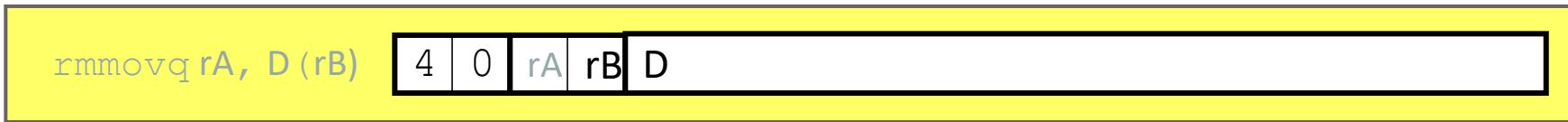
PC Update

- Increment PC by 2

Stage Computation: Arithmetic/Logic Operations

	OPq rA, rB	
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	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Read operand A Read operand B
Execute	$valE \leftarrow valB \text{ OP } valA$ 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

Executing rmmovq



Fetch

- Read 10 bytes

Memory

- Write to memory

Decode

- Read operand registers

Write back

- Do nothing

Execute

- Compute effective address

PC Update

- Increment PC by 10

Stage Computation: `rmmovq`

<code>rmmovq rA, D(rB)</code>	
Fetch	icode:ifun $\leftarrow M_1[PC]$
	$rA:rB \leftarrow M_1[PC+1]$
	$valC \leftarrow M_8[PC+2]$
	$valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$
	$valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$
Memory	$M_8[valE] \leftarrow valA$
Write back	
PC update	$PC \leftarrow valP$

- Use ALU for address computation

Executing popq



Fetch

- Read 2 bytes

Decode

- Read stack pointer

Execute

- Increment stack pointer by 8

Memory

- Read from old stack pointer

Write back

- Update stack pointer
- Write result to register

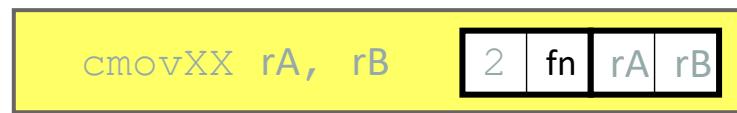
PC Update

- Increment PC by 2

Stage Computation: popq

	popq rA icode:ifun $\leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valP \leftarrow PC+2$	Read instruction byte Read register byte Compute next PC	<ul style="list-style-type: none"> Use ALU to increment stack pointer Must update two registers <ul style="list-style-type: none"> Popped value New stack pointer
Fetch		Read stack pointer Read stack pointer	
Decode	$valA \leftarrow R[\%rsp]$ $valB \leftarrow R[\%rsp]$		
Execute	$valE \leftarrow valB + 8$	Increment stack pointer	
Memory	$valM \leftarrow M_8[valA]$	Read from stack	
Write back	$R[\%rsp] \leftarrow valE$ $R[rA] \leftarrow valM$	Update stack pointer Write back result	
PC update	$PC \leftarrow valP$	Update PC	

Executing Conditional Moves



Fetch

- Read 2 bytes

Decode

- Read operand registers

Execute

- If !cnd, then set destination register to 0xF

Memory

- Do nothing

Write back

- Update register (or not)

PC Update

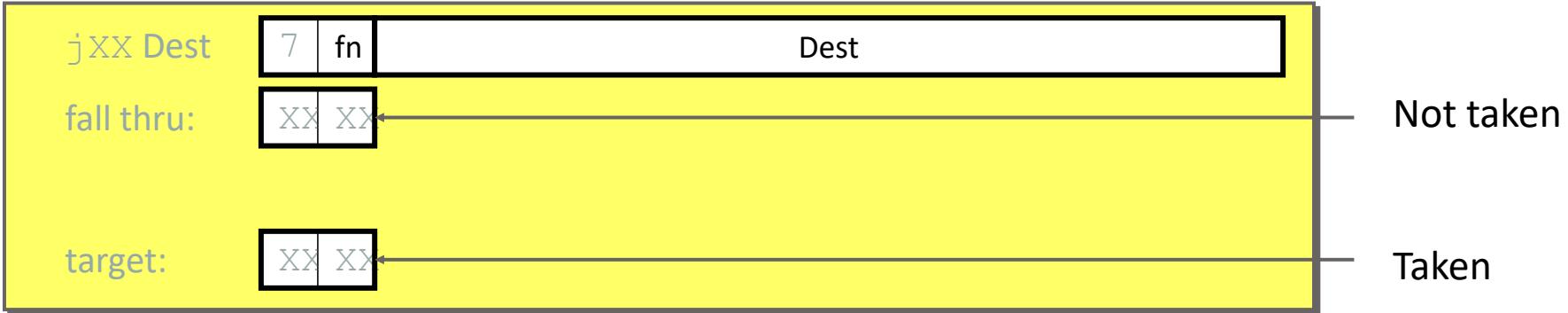
- Increment PC by 2

Stage Computation: Cond. Move

	cmoveXX rA, rB	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $rA:rB \leftarrow M_1[\text{PC+1}]$ $\text{valP} \leftarrow \text{PC+2}$	Read instruction byte Read register byte Compute next PC
Decode	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow 0$	Read operand A
Execute	$\text{valE} \leftarrow \text{valB} + \text{valA}$ $\text{If } ! \text{Cond(CC,ifun)} \text{ } rB \leftarrow 0xF$	Pass valA through ALU (Disable register update)
Memory		
Write back	$R[rB] \leftarrow \text{valE}$	Write back result
PC update	$\text{PC} \leftarrow \text{valP}$	Update PC

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
 - If condition codes & move condition indicate no move

Executing Jumps



Fetch

- Read 9 bytes
- Increment PC by 9

Decode

- Do nothing

Execute

- Determine whether to take branch based on jump condition and condition codes

Memory

- Do nothing

Write back

- Do nothing

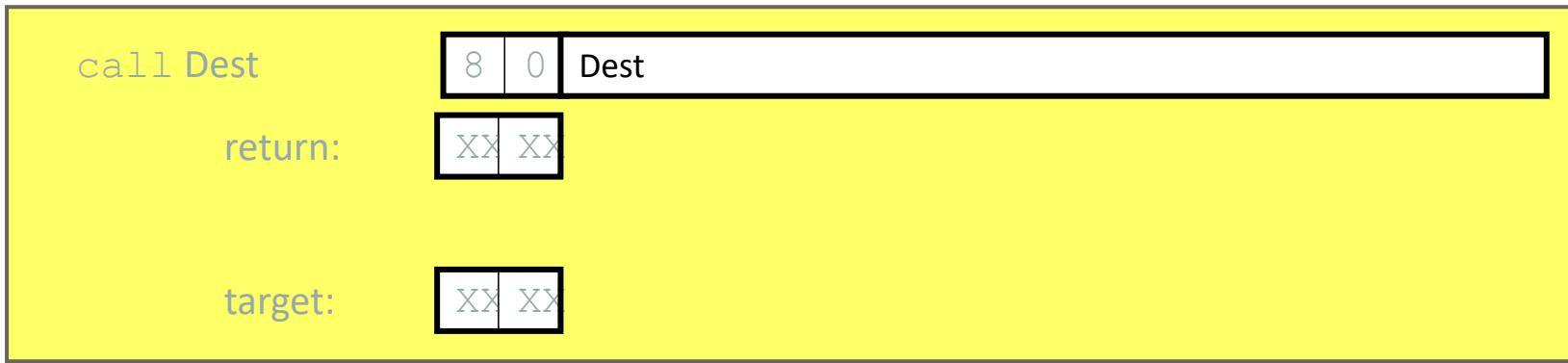
PC Update

- Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest		
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+9$	Read instruction byte Read destination address Fall through address	<ul style="list-style-type: none">◦ Compute both addresses◦ Choose based on setting of condition codes and branch condition
Decode			
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	Take branch?	
Memory			
Write back			
PC update	$\text{PC} \leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	Update PC	

Executing call



Fetch

- Read 9 bytes
- Increment PC by 9

Decode

- Read stack pointer

Execute

- Decrement stack pointer by 8

Memory

- Write incremented PC to new value of stack pointer

Write back

- Update stack pointer

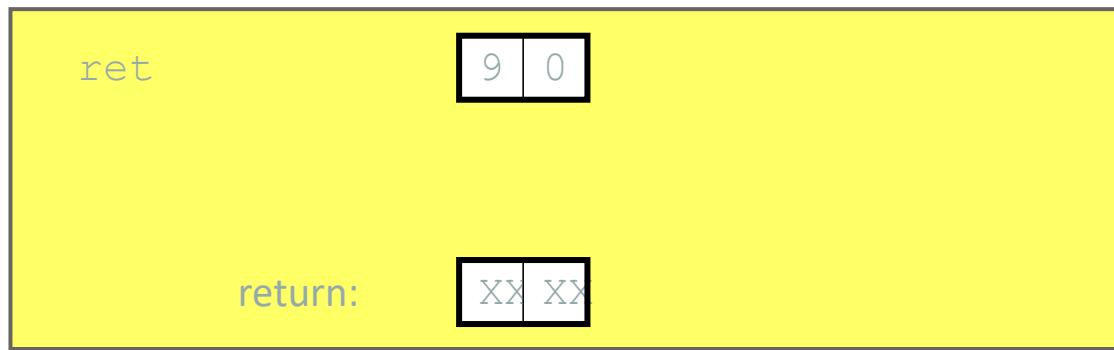
PC Update

- Set PC to Dest

Stage Computation: call

call Dest			
Fetch	$icode:ifun \leftarrow M_1[PC]$ $valC \leftarrow M_8[PC+1]$ $valP \leftarrow PC+9$	Read instruction byte Read destination address Compute return point	<ul style="list-style-type: none">◦ Use ALU to decrement stack pointer◦ Store incremented PC
Decode	$valB \leftarrow R[\%rsp]$	Read stack pointer	
Execute	$valE \leftarrow valB + -8$	Decrement stack pointer	
Memory	$M_8[valE] \leftarrow valP$	Write return value on stack	
Write back	$R[\%rsp] \leftarrow valE$	Update stack pointer	
PC update	$PC \leftarrow valC$	Set PC to destination	

Executing ret



Fetch

- Read 1 byte

Decode

- Read stack pointer

Execute

- Increment stack pointer by 8

Memory

- Read return address from old stack pointer

Write back

- Update stack pointer

PC Update

- Set PC to return address

Stage Computation: ret

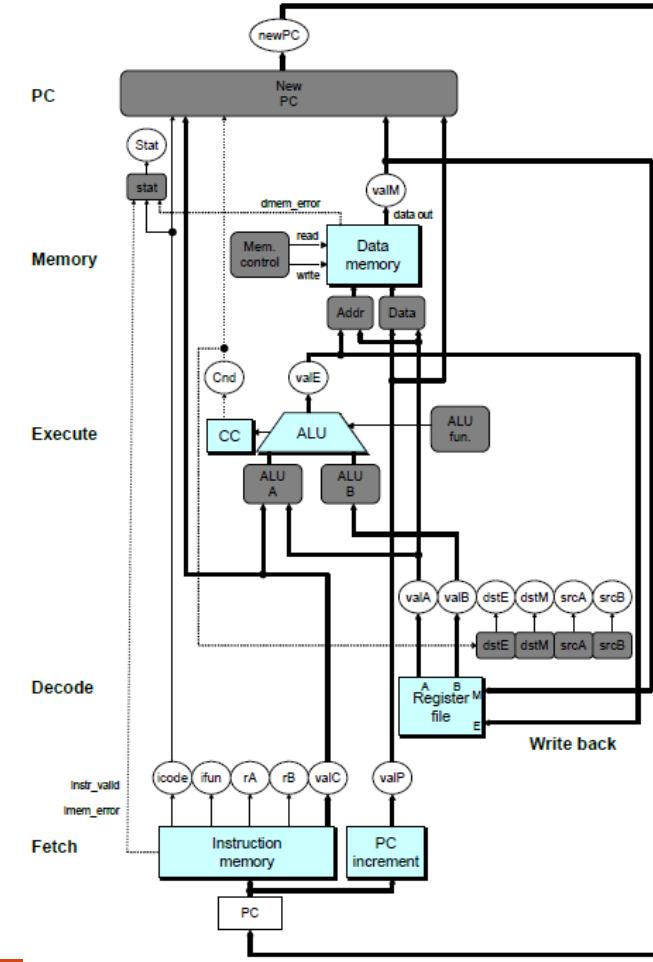
	ret	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
Decode	$\text{valA} \leftarrow R[\% \text{rsp}]$ $\text{valB} \leftarrow R[\% \text{rsp}]$	Read operand stack pointer Read operand stack pointer
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Read return address
Write back	$R[\% \text{rsp}] \leftarrow \text{valE}$	Update stack pointer
PC update	$\text{PC} \leftarrow \text{valM}$	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

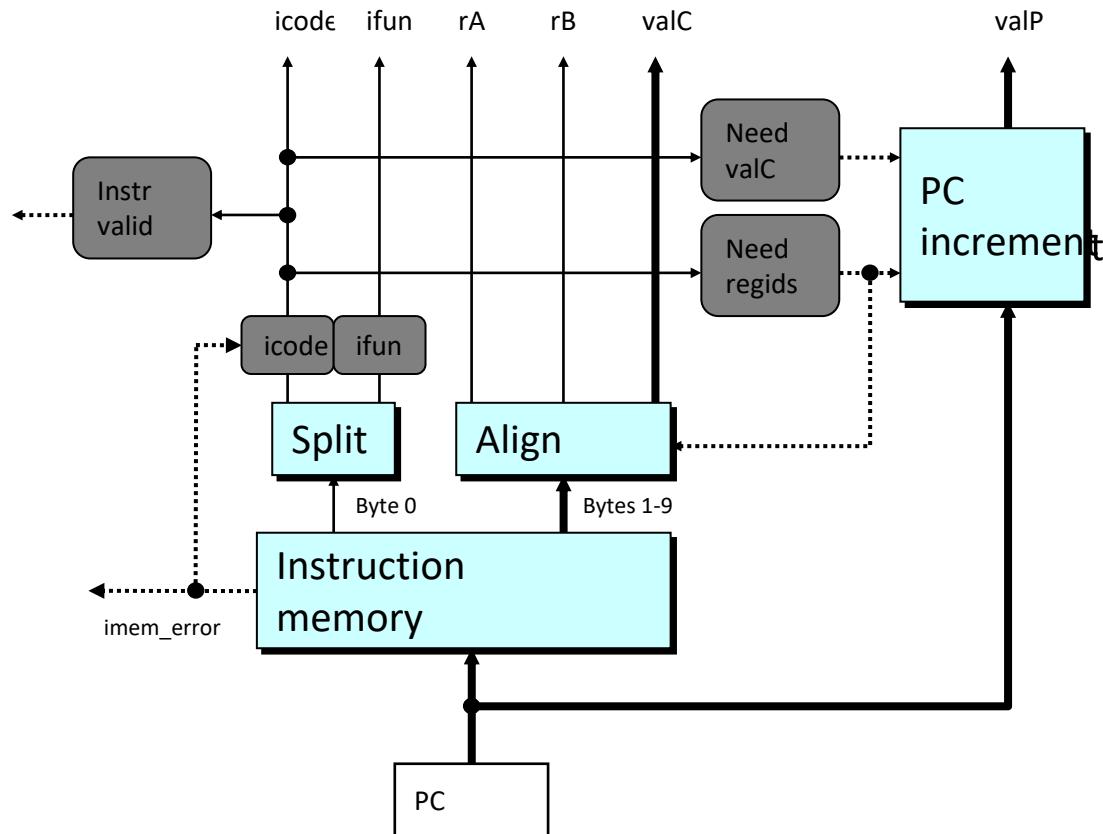
SEQ Hardware

Key

- Blue boxes: predefined hardware blocks
 - E.g., memories, ALU
- Gray boxes: control logic
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



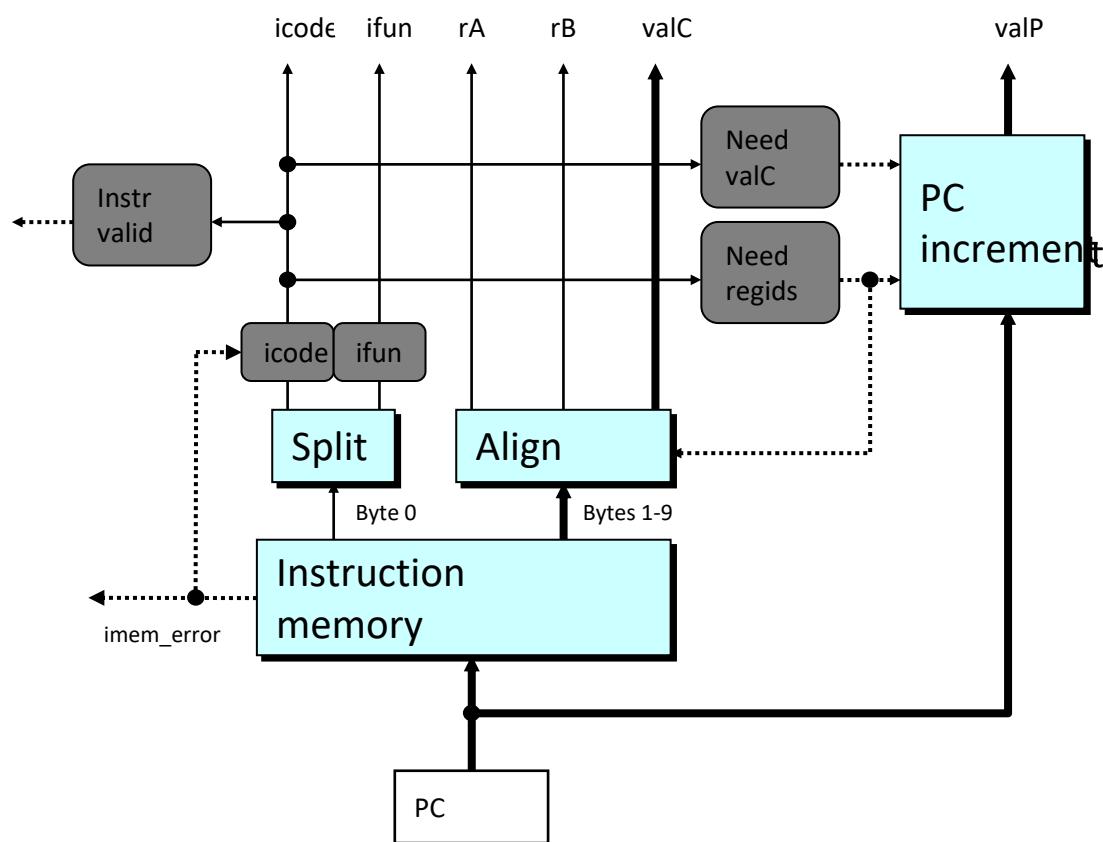
Fetch Logic



Predefined Blocks

- PC: Register containing PC
- Instruction memory: Read 10 bytes (PC to PC+9)
 - Signal invalid address
- Split: Divide instruction byte into iCode and ifun
- Align: Get fields for rA, rB, and valC

Fetch Logic



Control Logic

- Instr. Valid: Is this instruction valid?
- iCode, ifun: Generate no-op if invalid address
- Need regids: Does this instruction have a register byte?
- Need valC: Does this instruction have a constant word?

Decode Logic

Register File

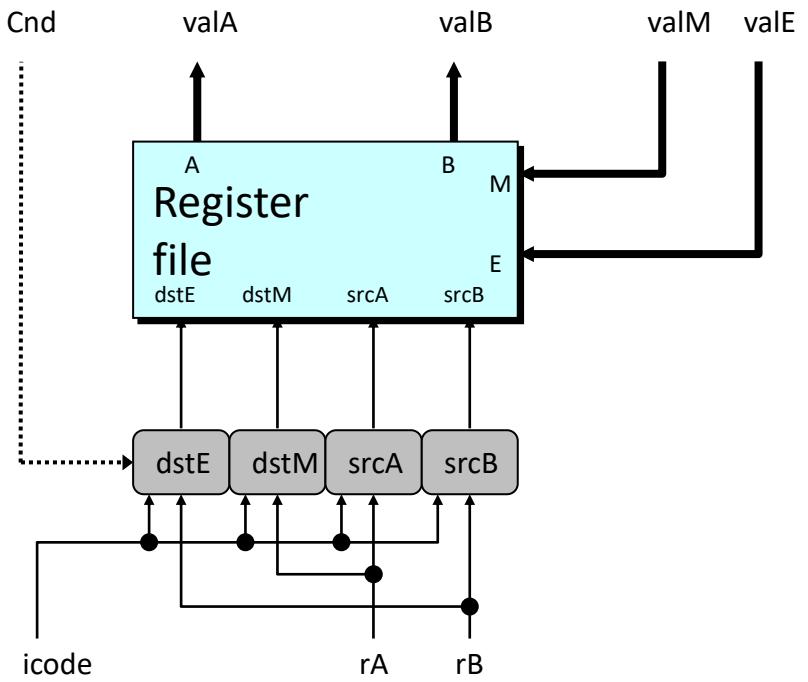
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

Control Logic

- srcA, srcB: read port addresses
- dstE, dstM: write port addresses

Signals

- Cnd: Indicate whether or not to perform conditional move
 - Computed in Execute stage



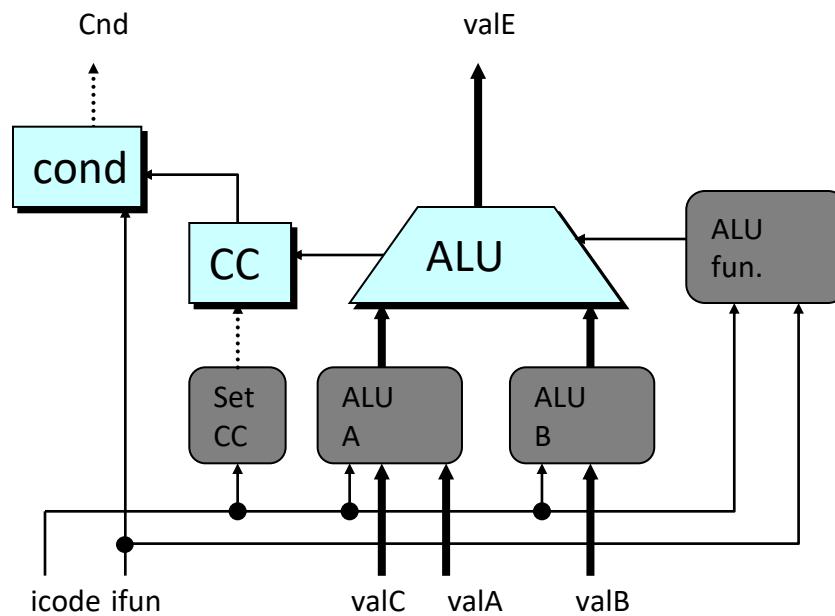
Execute Logic

Units

- ALU
 - Implements 4 required functions
 - Generates condition code values
- CC
 - Register with 3 condition code bits
- cond
 - Computes conditional jump/move flag

Control Logic

- Set CC: Should condition code register be loaded?
- ALU A: Input A to ALU
- ALU B: Input B to ALU
- ALU fun: What function should ALU compute?



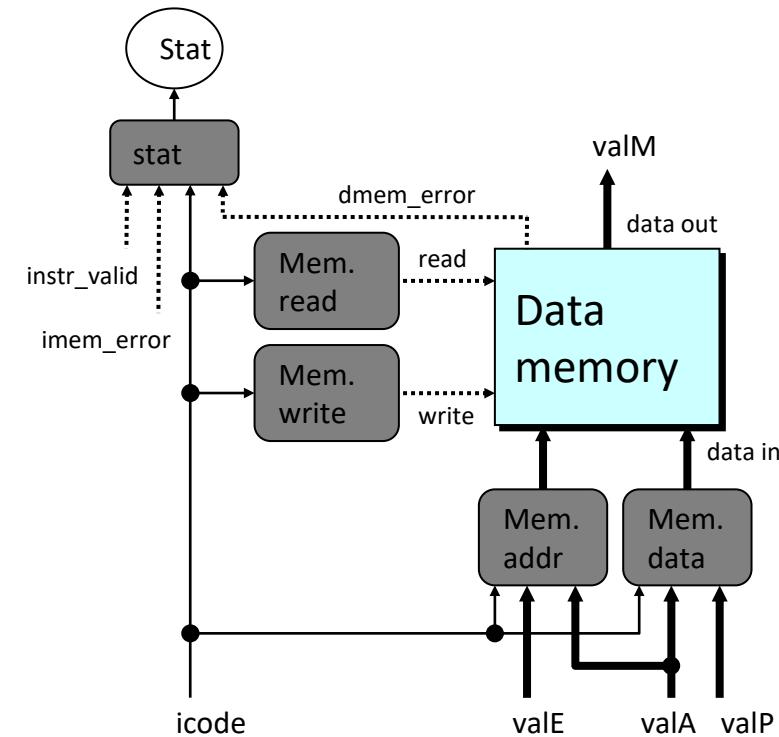
Memory Logic

Memory

- Reads or writes memory word

Control Logic

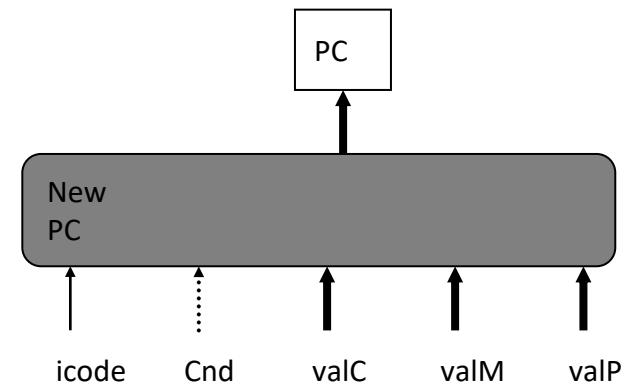
- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



PC Update Logic

New PC

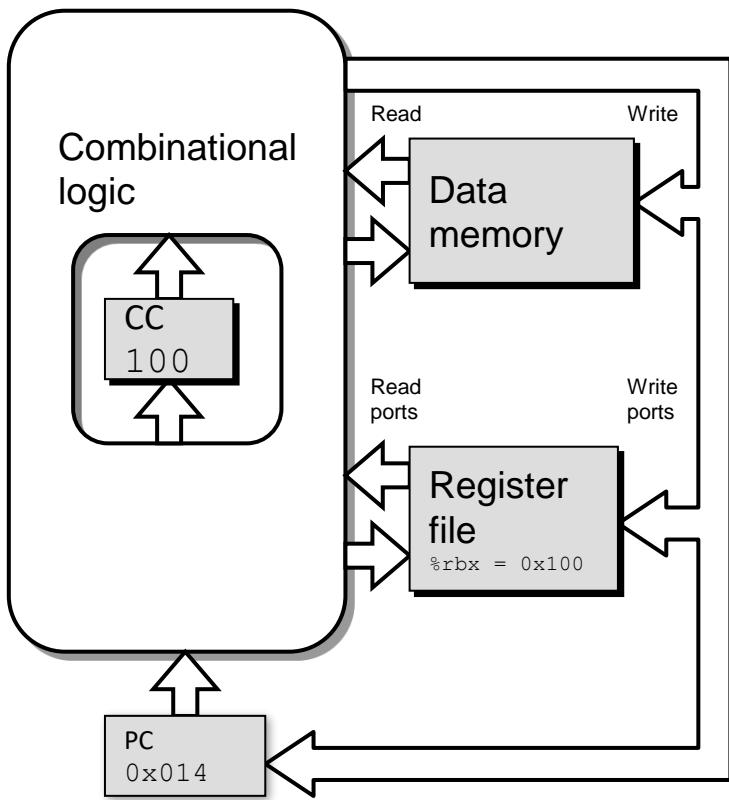
- Select next value of PC



PC Update

OPq rA, rB		
PC update	PC \leftarrow valP	Update PC
rmmovq rA, D(rB)		
PC update	PC \leftarrow valP	Update PC
popq rA		
PC update	PC \leftarrow valP	Update PC
jXX Dest		
PC update	PC \leftarrow Cnd ? 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

SEQ Operation



State

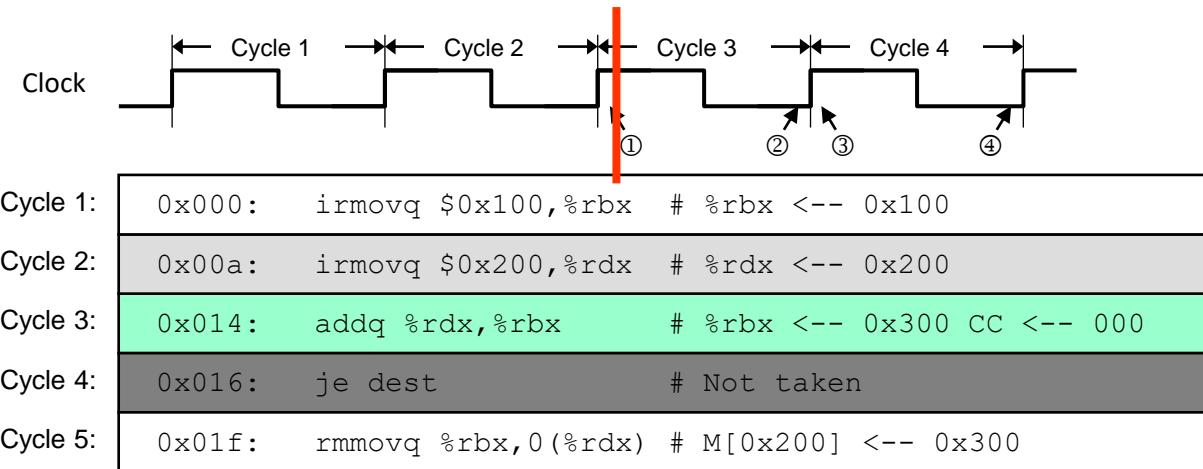
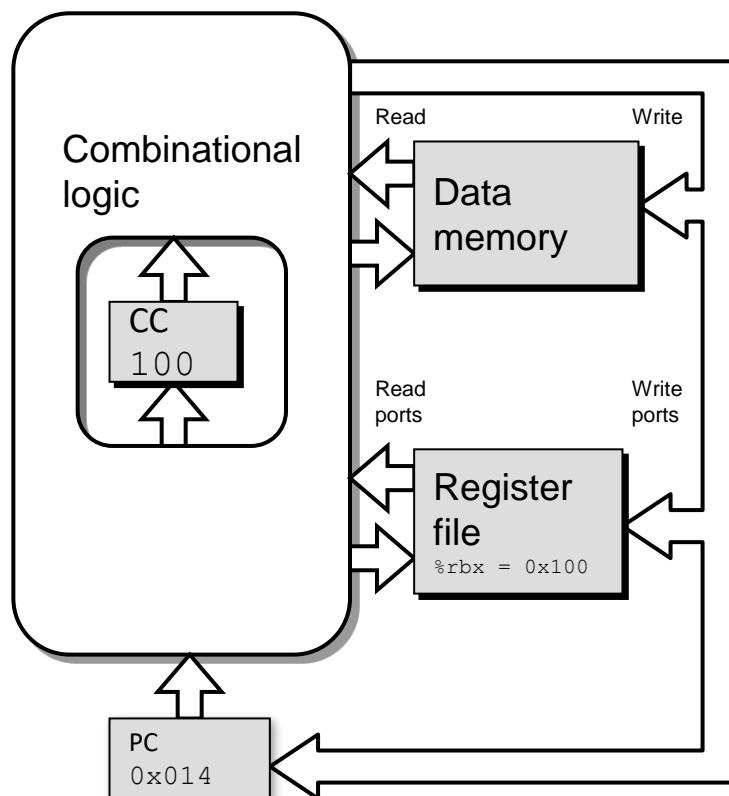
- PC register
- Cond. Code register
- Data memory
- Register file

All updated as clock rises

Combinational Logic

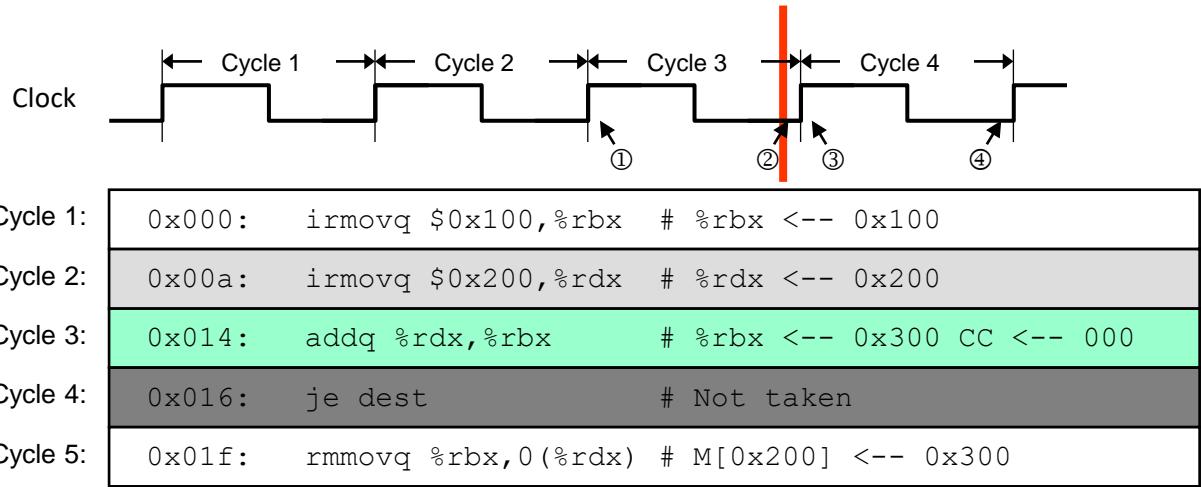
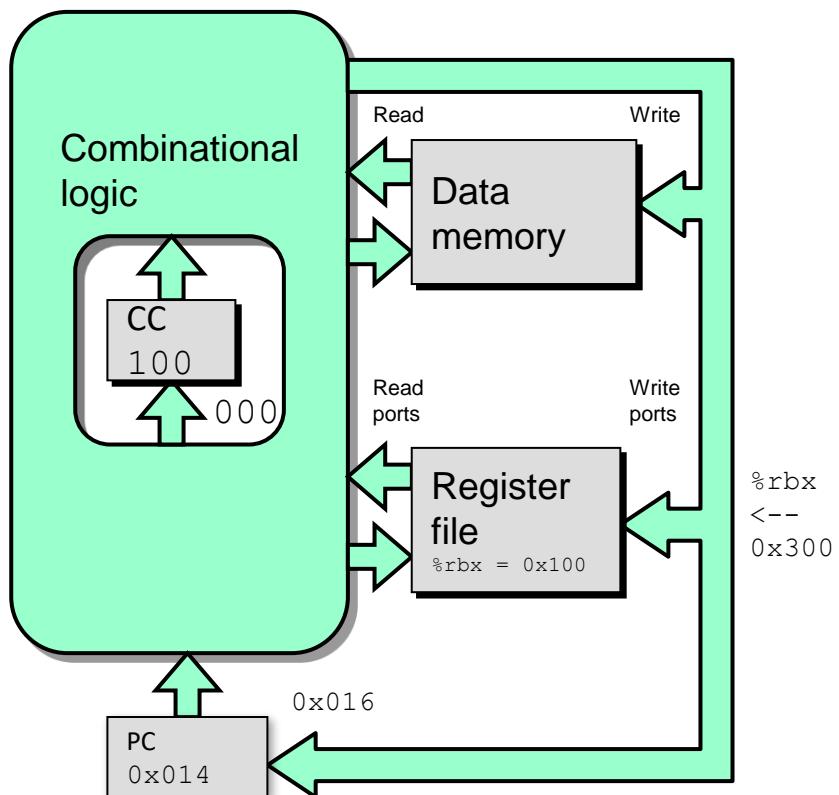
- ALU
- Control logic
- Memory reads
 - Instruction memory
 - Register file
 - Data memory

SEQ Operation #2



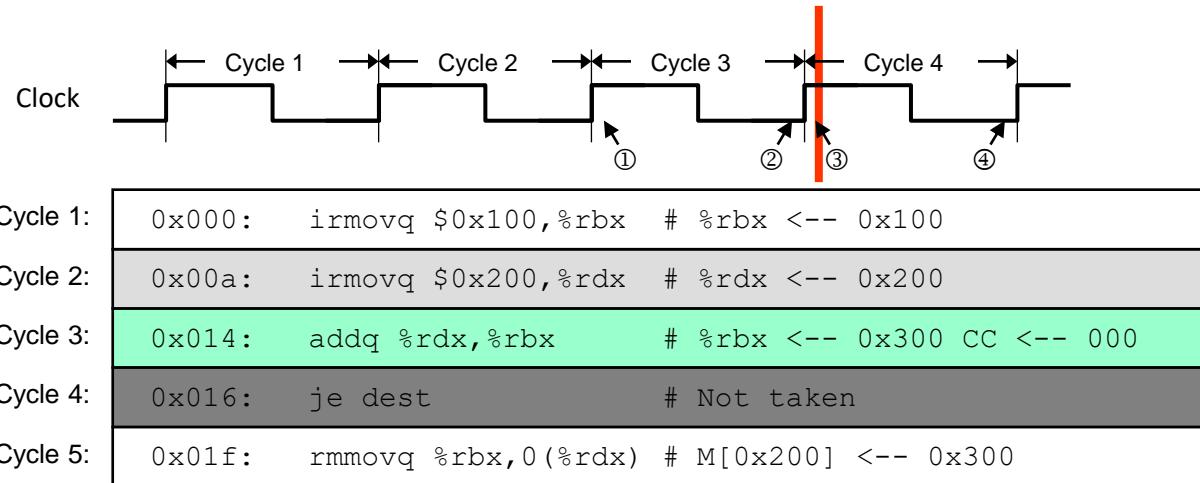
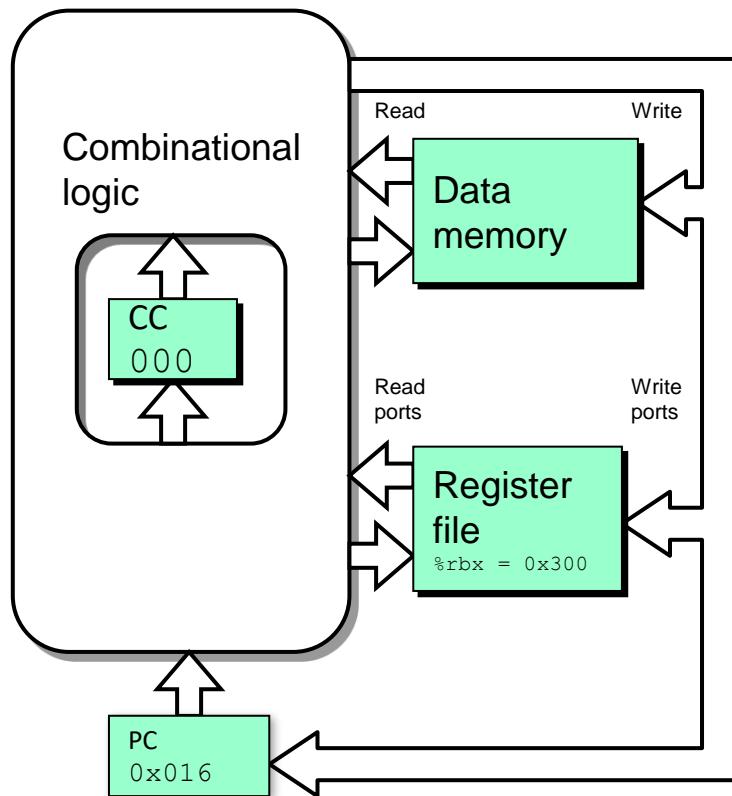
- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

SEQ Operation #3



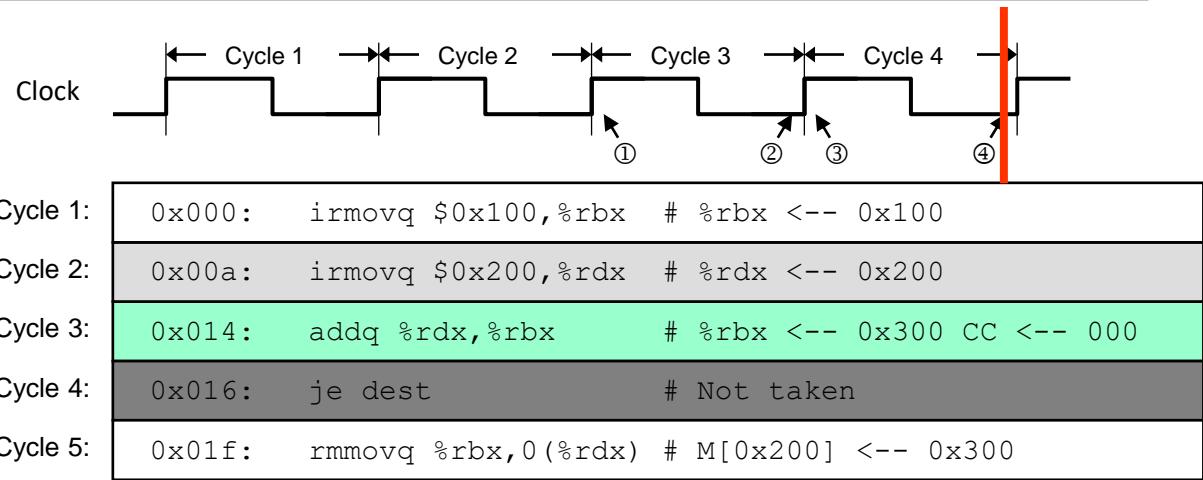
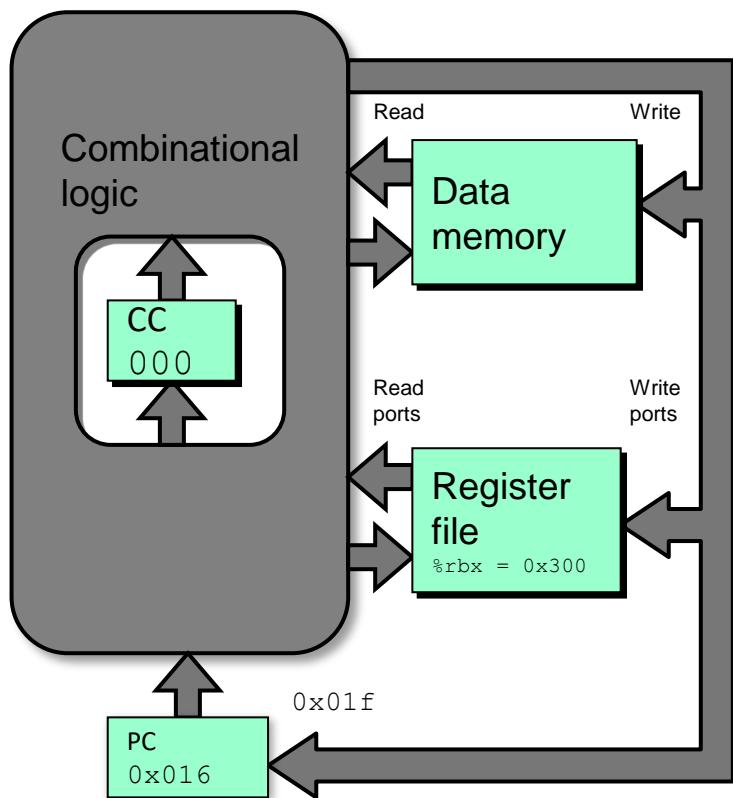
- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction

SEQ Operation #4



- state set according to addq instruction
- combinational logic starting to react to state changes

SEQ Operation #5



- state set according to `addq` instruction
- combinational logic generates results for `je` instruction

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

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

Thank You!

Computer Systems Organization (CS2.201)

LECTURE 8 & 9 & 10 - PROCESSOR ARCHITECTURE DESIGN : PIPELINING (SECTION
4.4 TILL 4.5.5)

Deepak Gangadharan
Computer Systems Group (CSG), IIIT Hyderabad

Slide Contents: Adapted from slides by Randal Bryant

Overview

General Principles of Pipelining

- Goal
- Difficulties

Creating a Pipelined Y86-64 Processor

- Rearranging SEQ
- Inserting pipeline registers
- Problems with data and control hazards

Real-World Pipelines: Car Washes

Sequential



Parallel



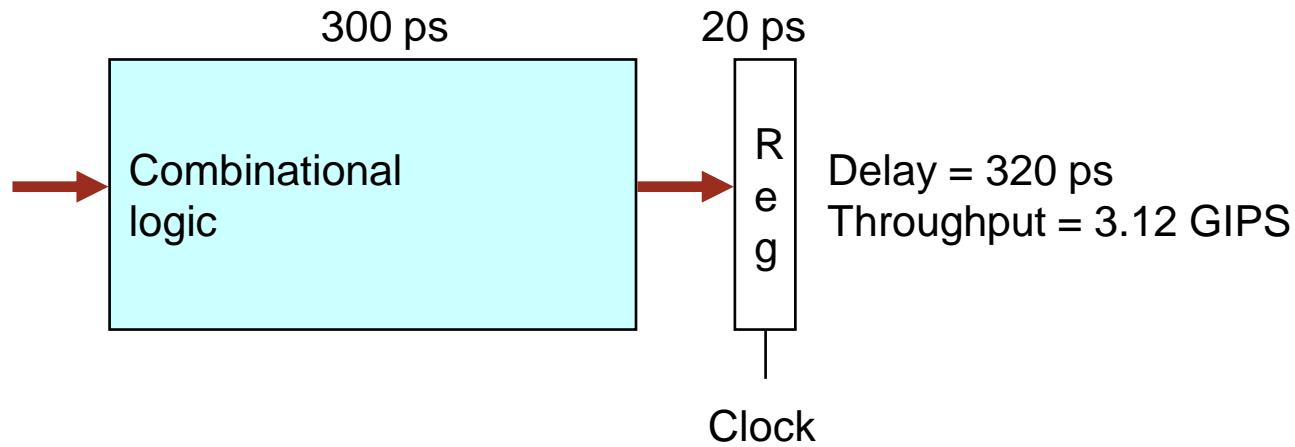
Idea

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

Pipelined



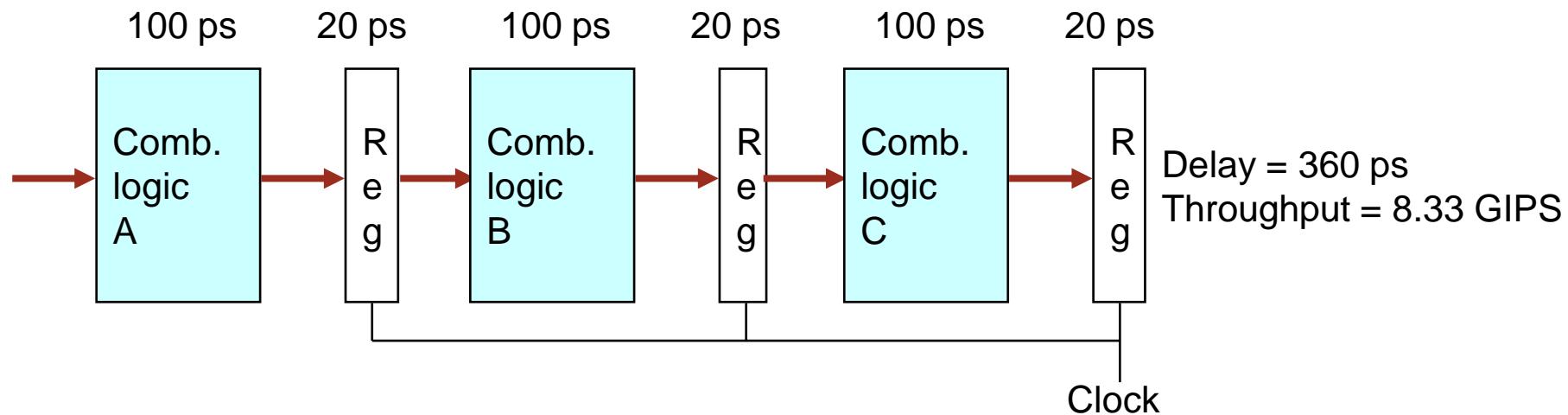
Computational Example



System

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

3-Way Pipelined Version

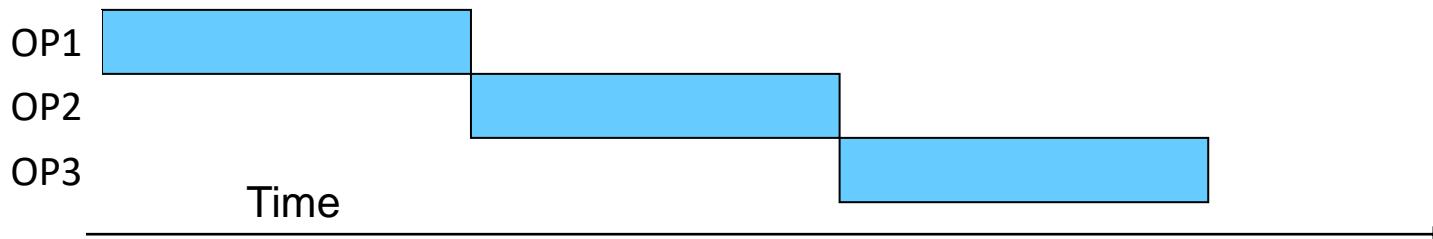


System

- 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

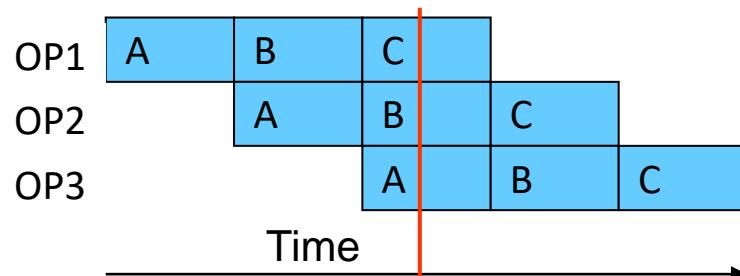
Pipeline Diagrams

Unpipelined



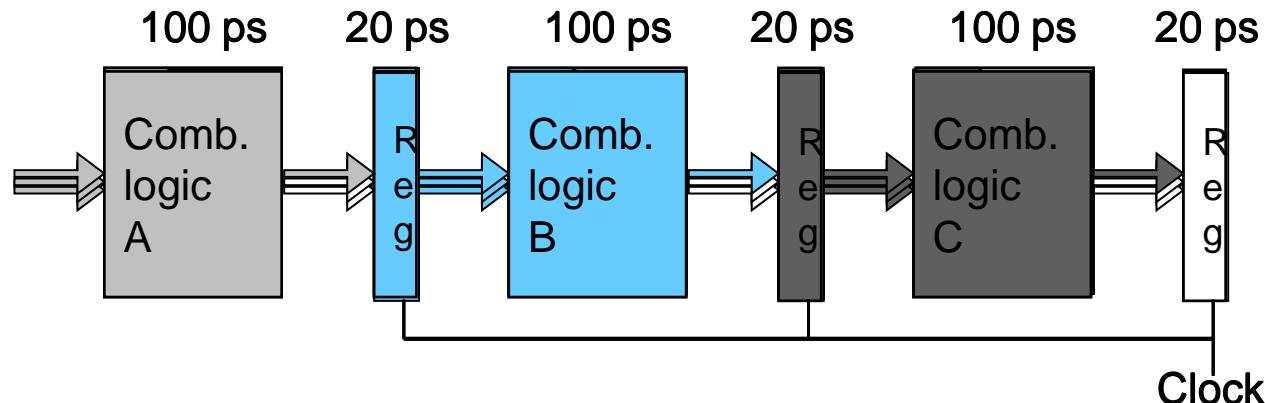
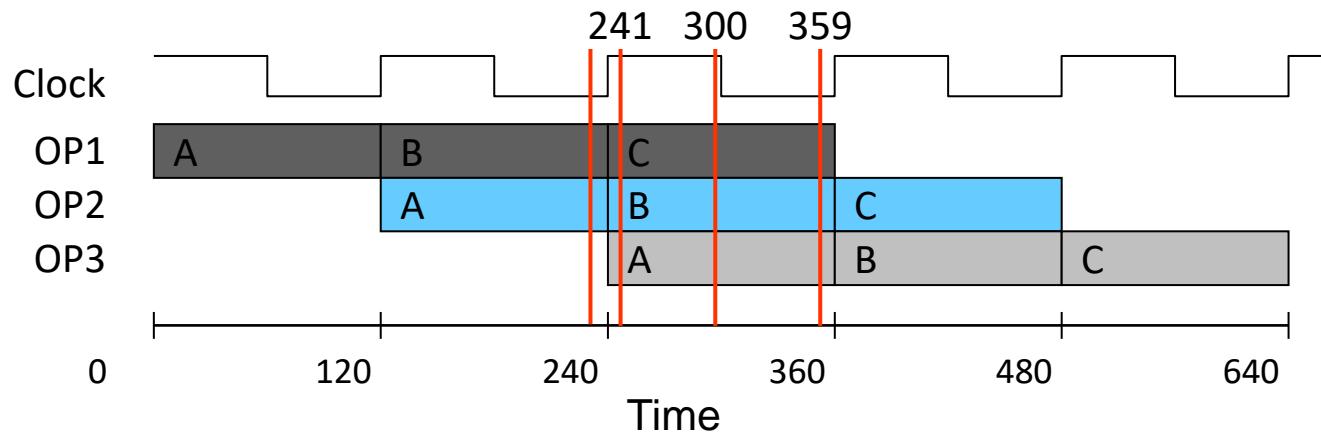
- Cannot start new operation until previous one completes

3-Way Pipelined

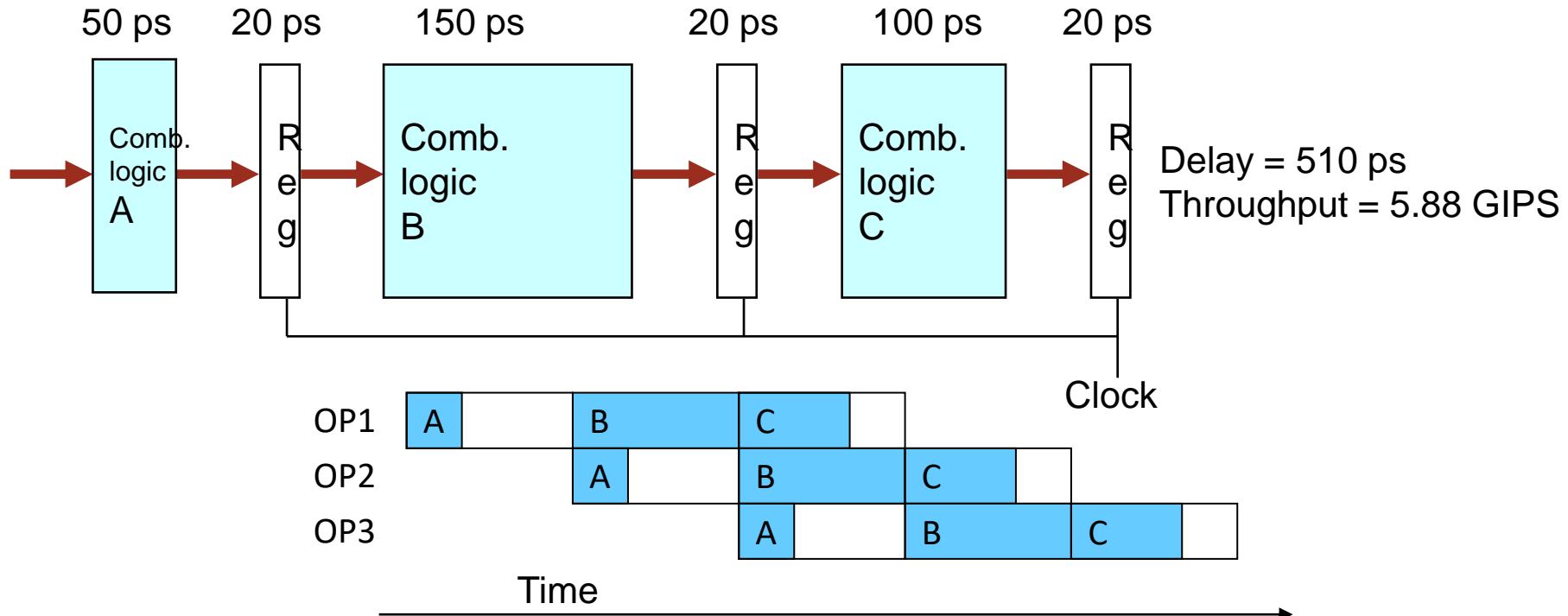


- Up to 3 operations in process simultaneously

Operating a Pipeline

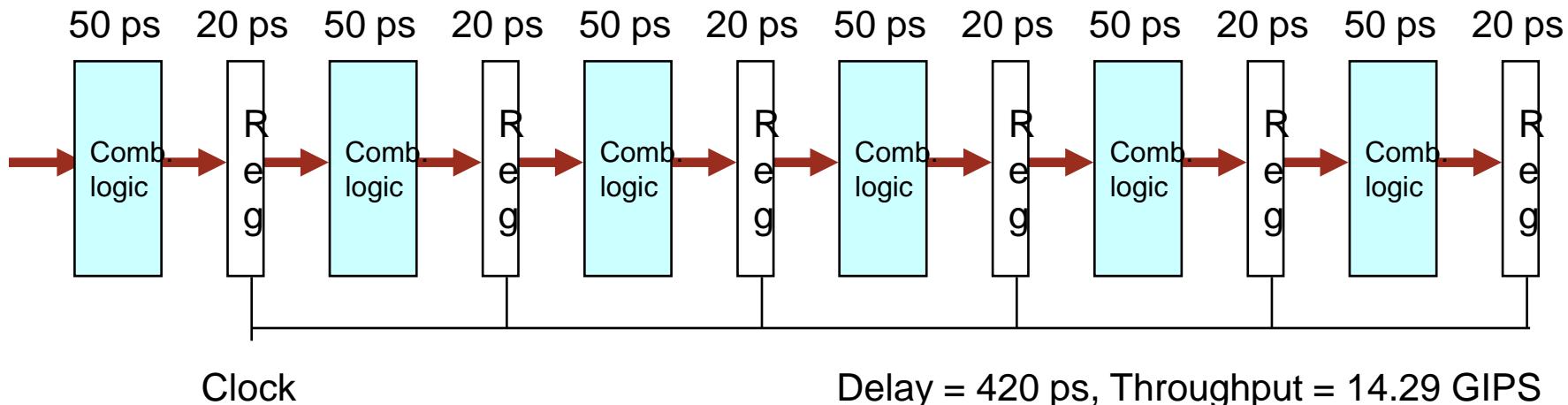


Limitations: Nonuniform Delays



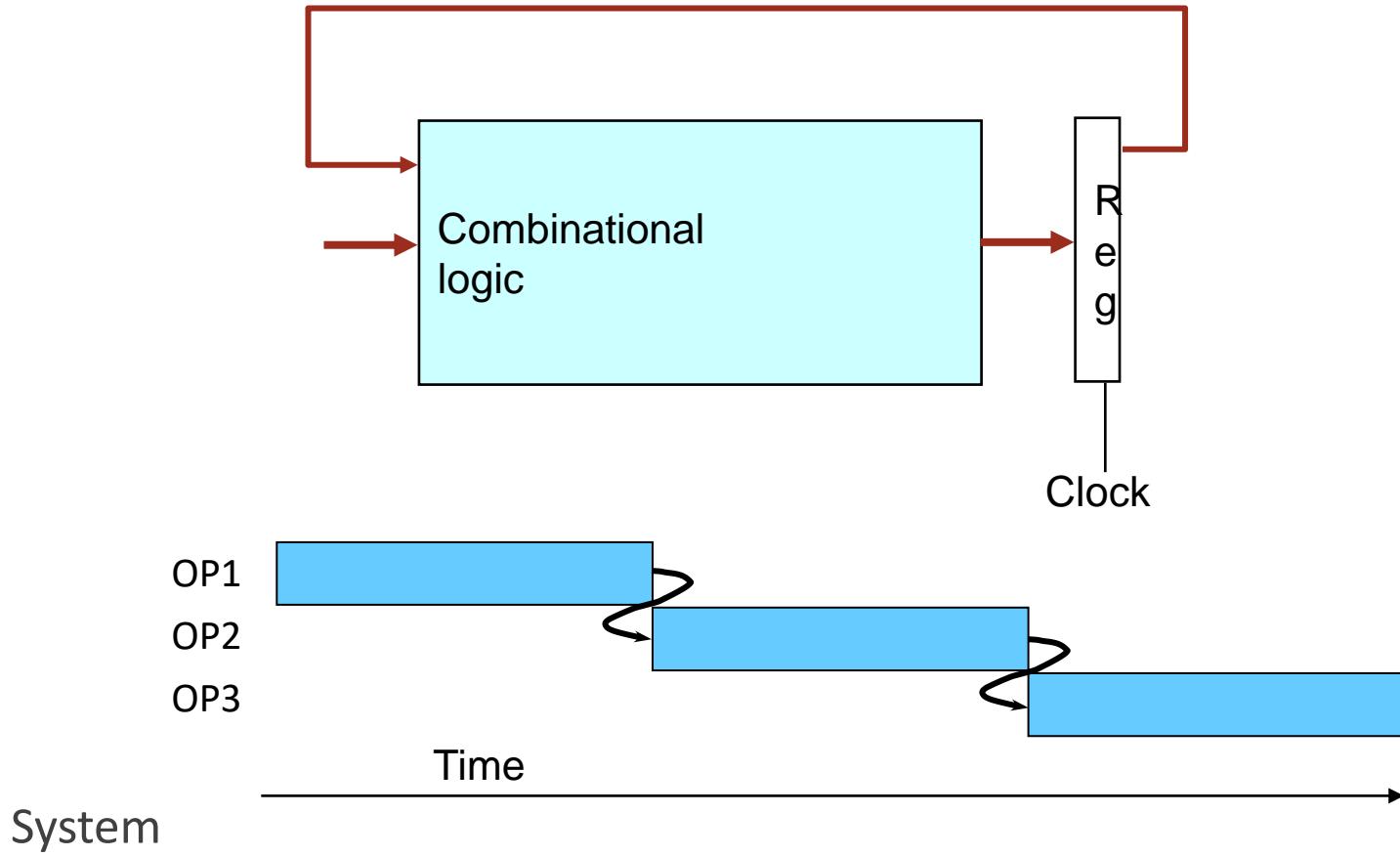
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead



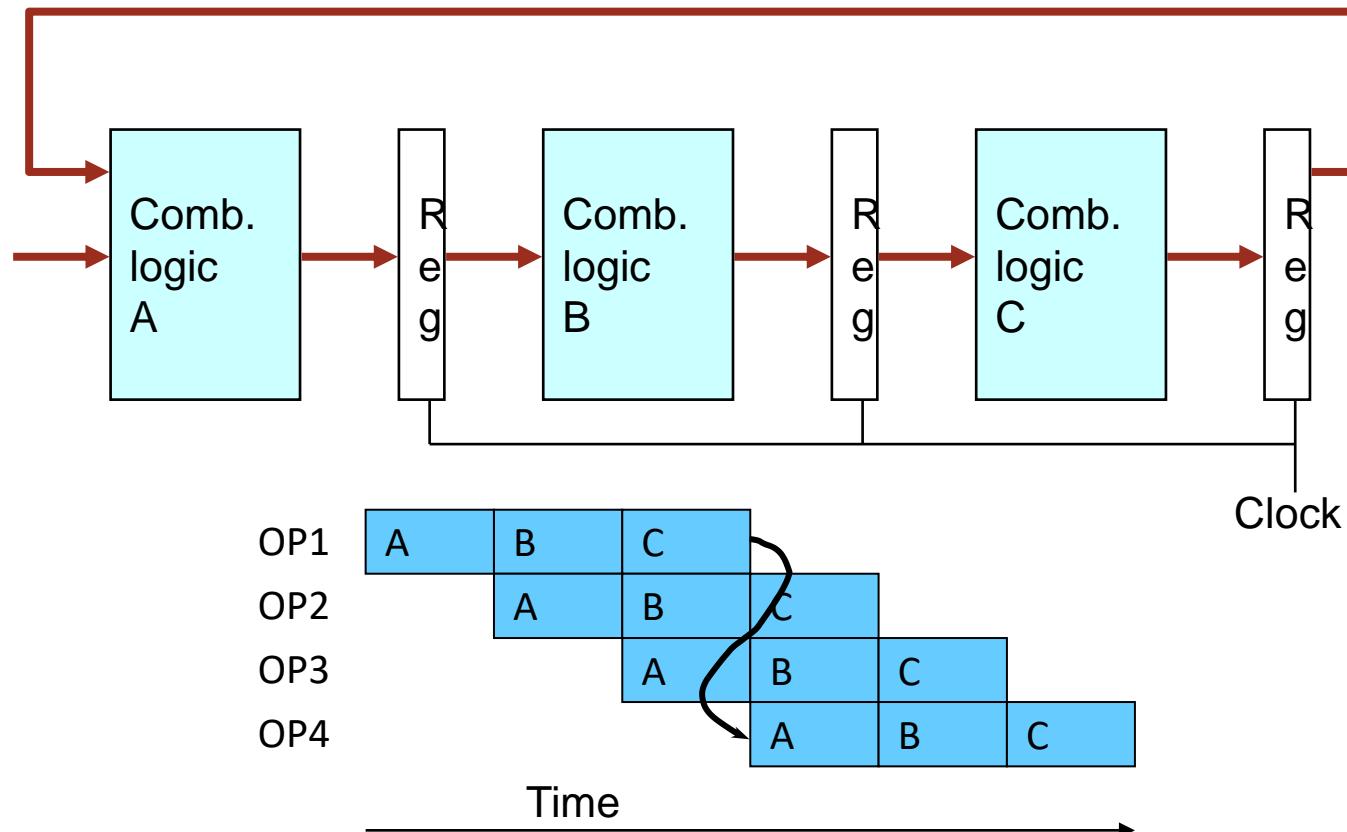
- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
 - 1-stage pipeline: 6.25%
 - 3-stage pipeline: 16.67%
 - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Data Dependencies



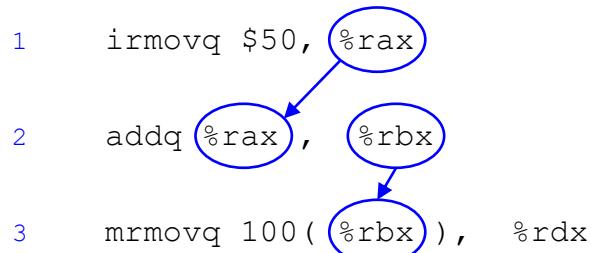
- Each operation depends on result from preceding one

Data Hazards



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

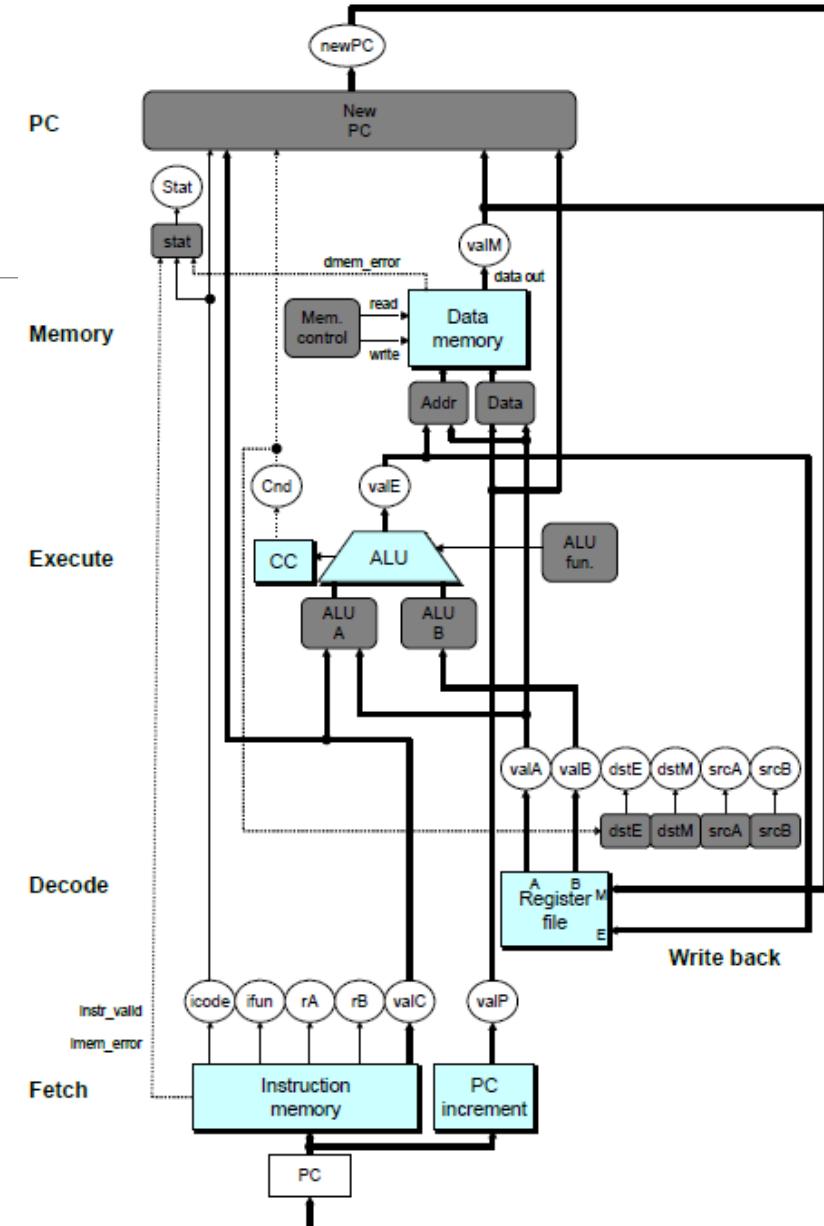
Data Dependencies in Processors



- 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

SEQ Hardware

- Stages occur in sequence
- One operation in process at a time



SEQ+ Hardware

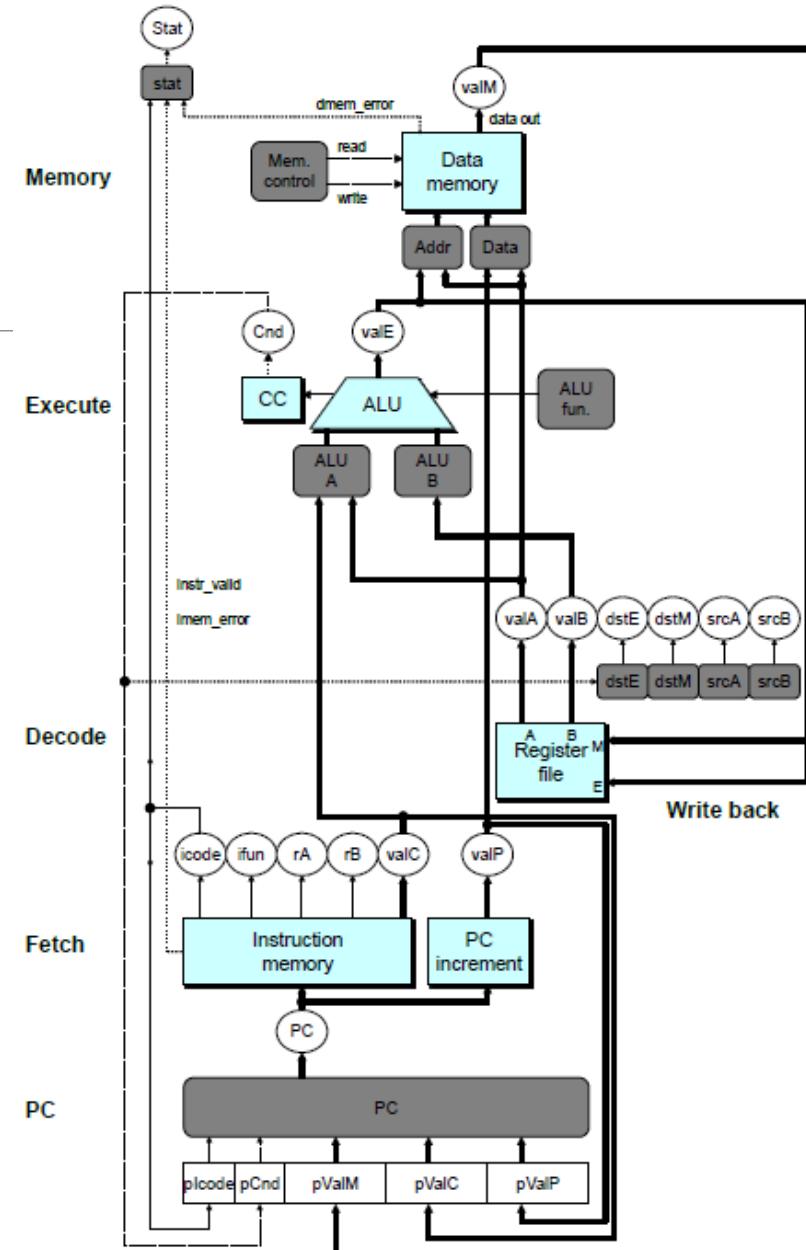
- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

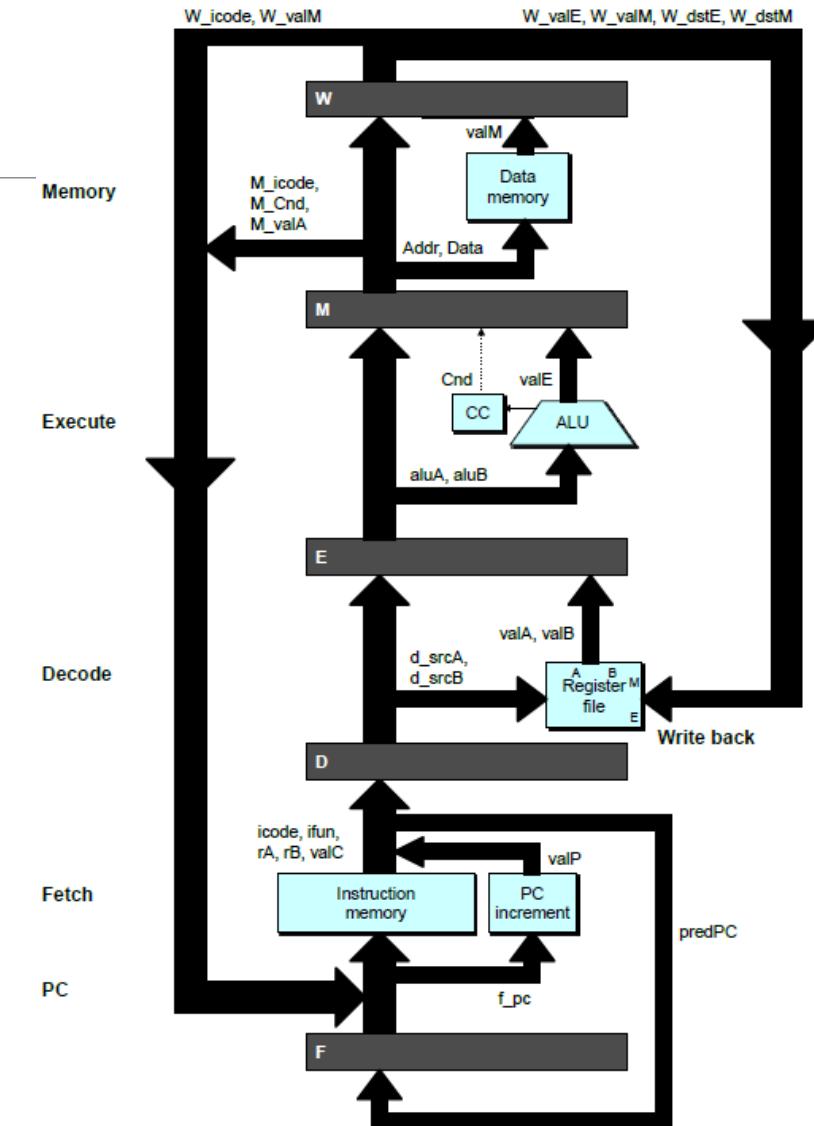
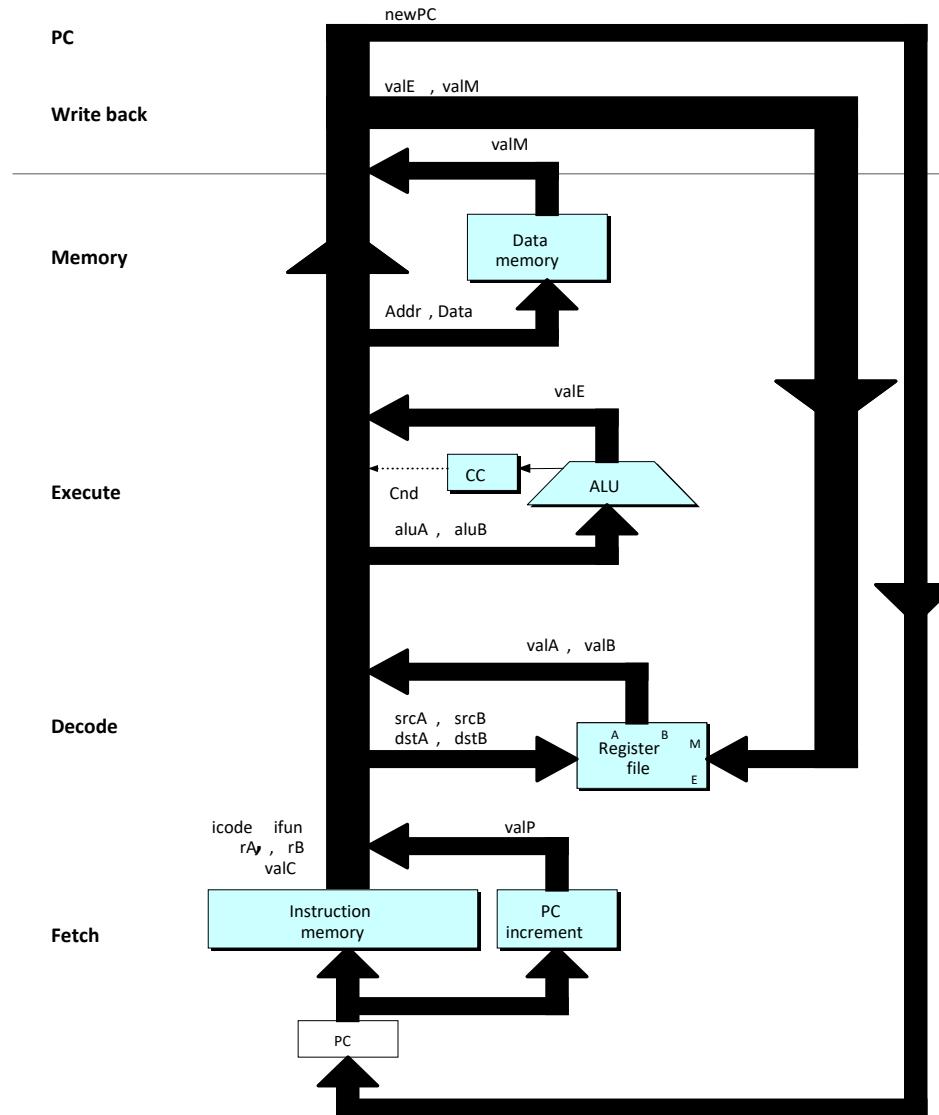
- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State

- PC is no longer stored in register
- But, can determine PC based on other stored information



Adding Pipeline Registers



Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

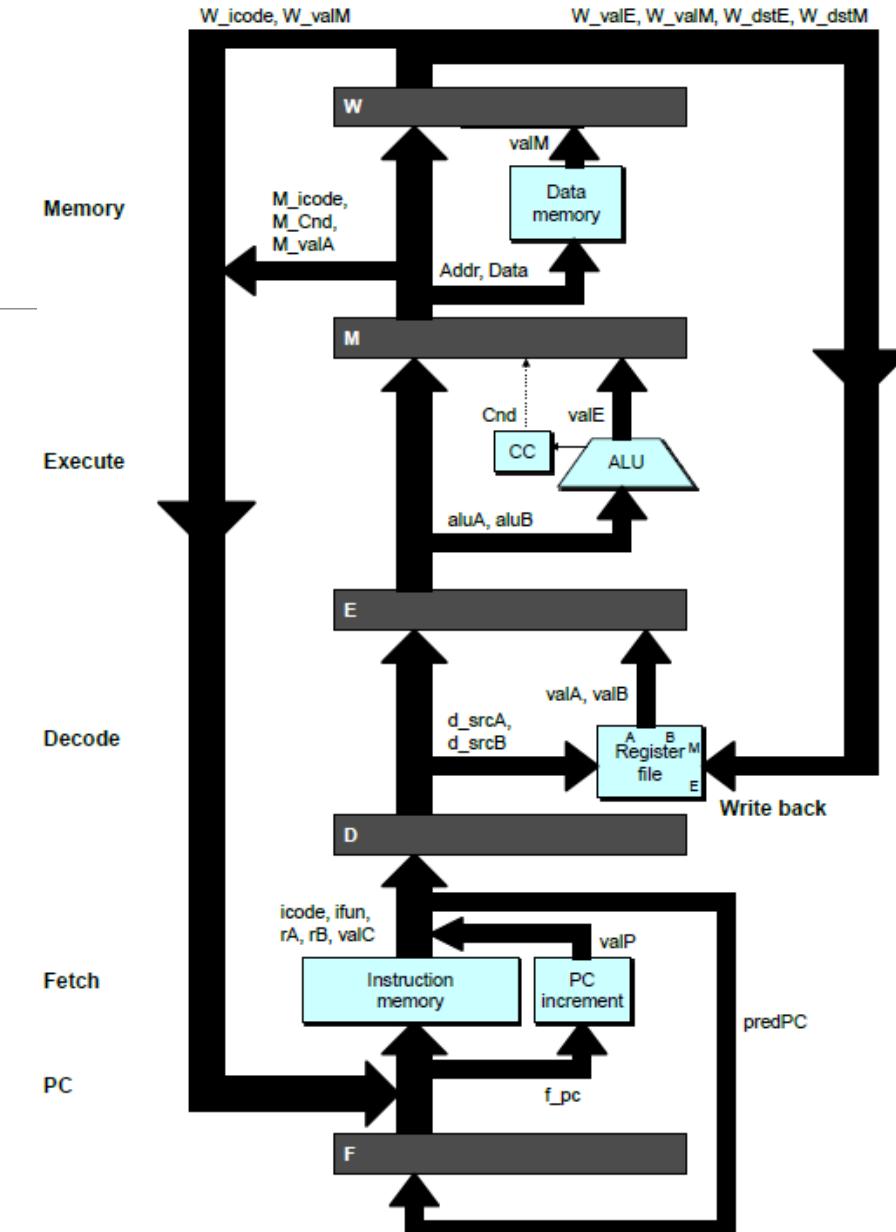
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file

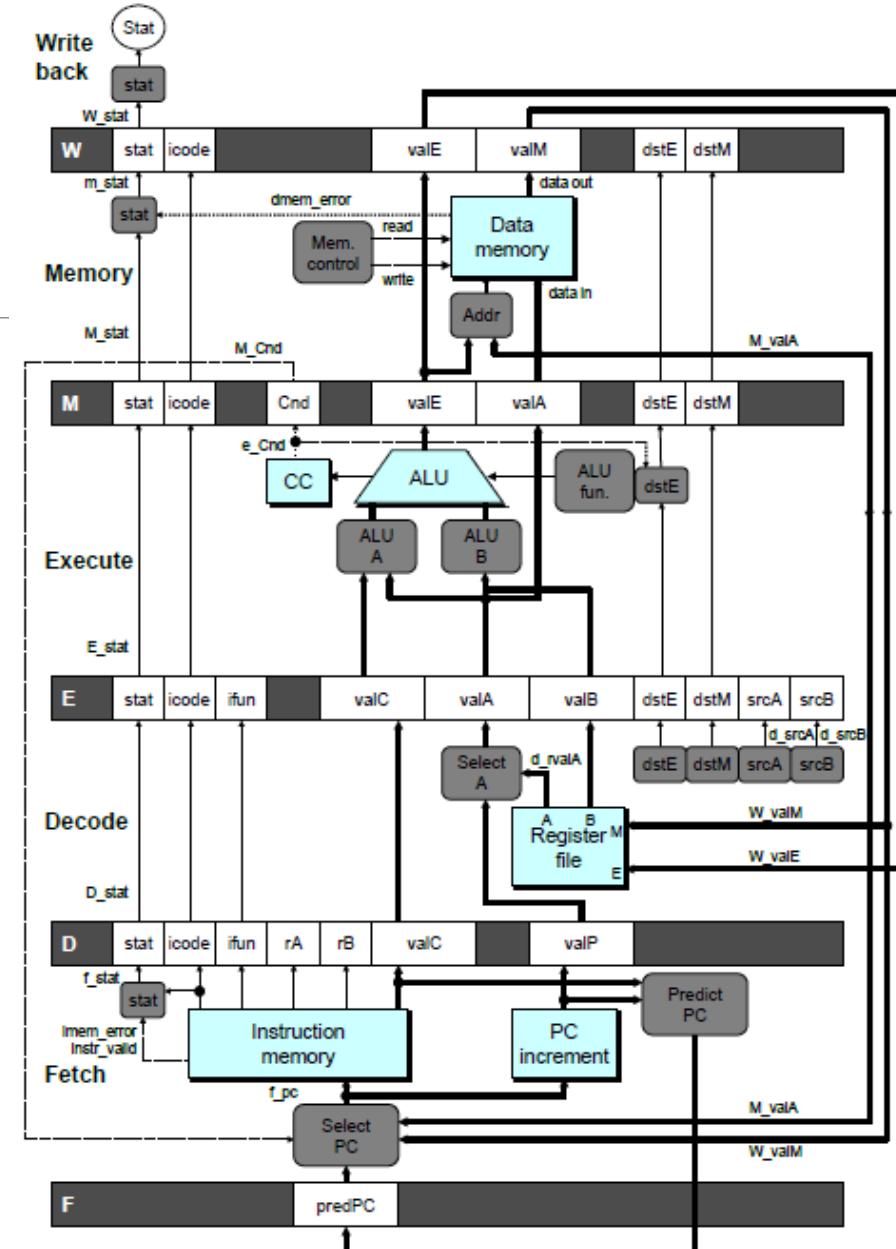


PIPE- Hardware

- 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



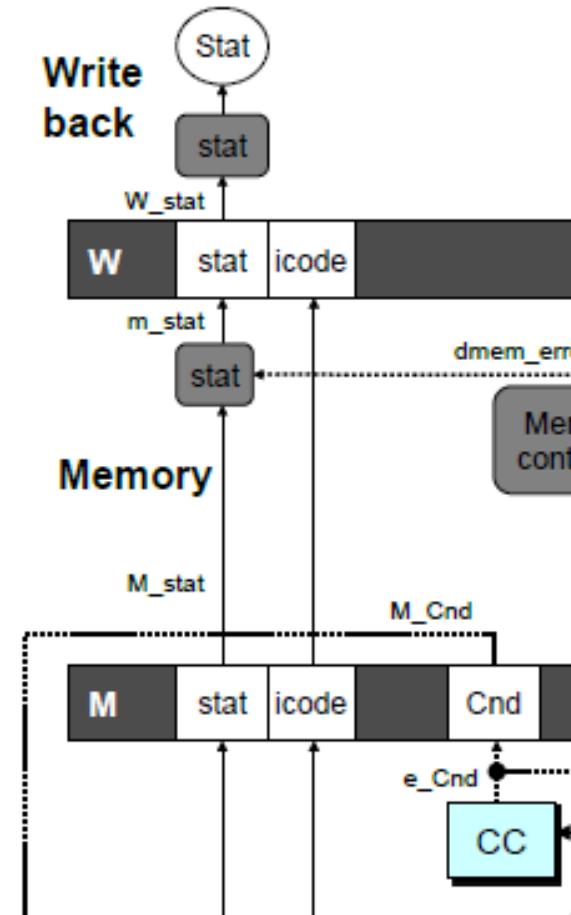
Signal Naming Conventions

S_Field

- Value of Field held in stage S pipeline register

s_Field

- Value of Field computed in stage S



Feedback Paths

Predicted PC

- Guess value of next PC

Branch information

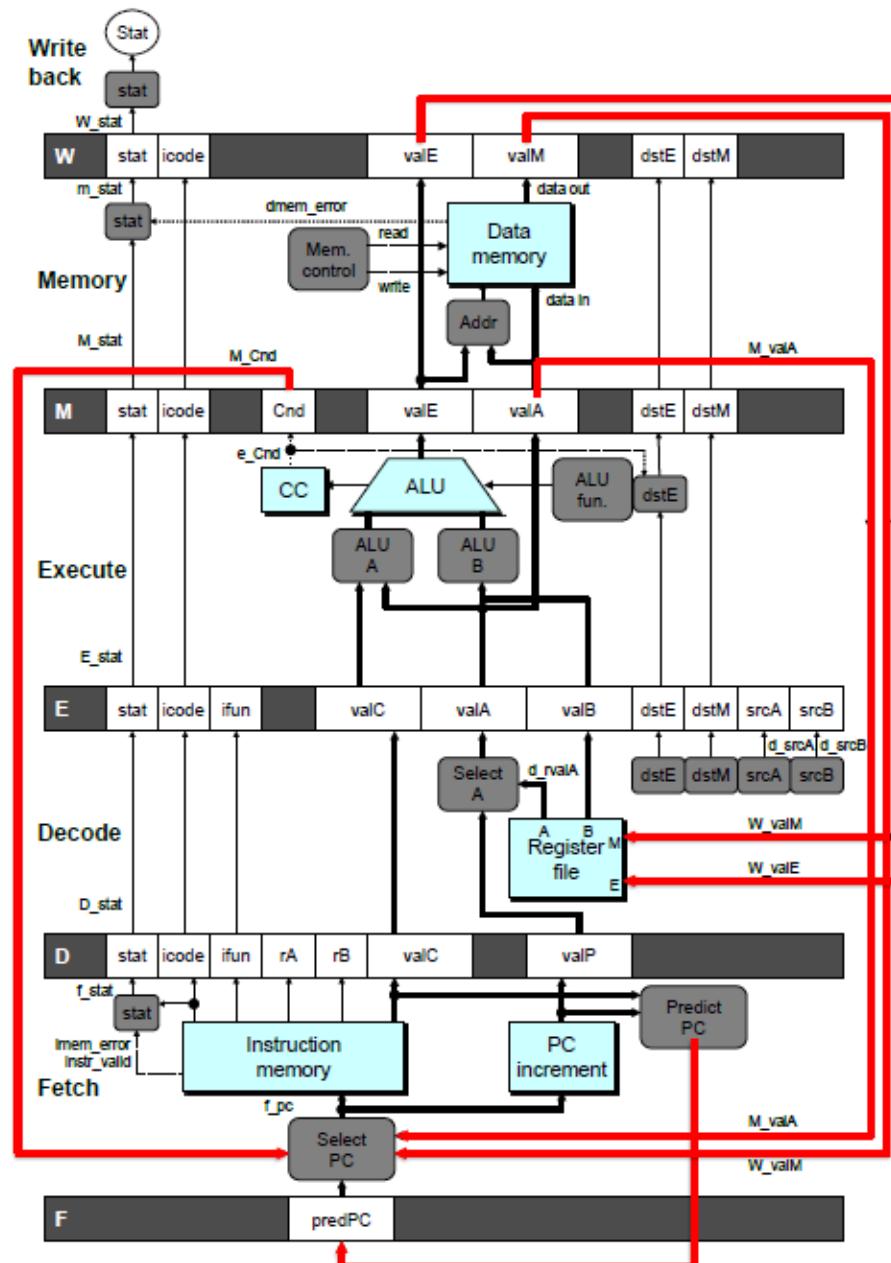
- Jump taken/not-taken
- Fall-through or target address

Return point

- Read from memory

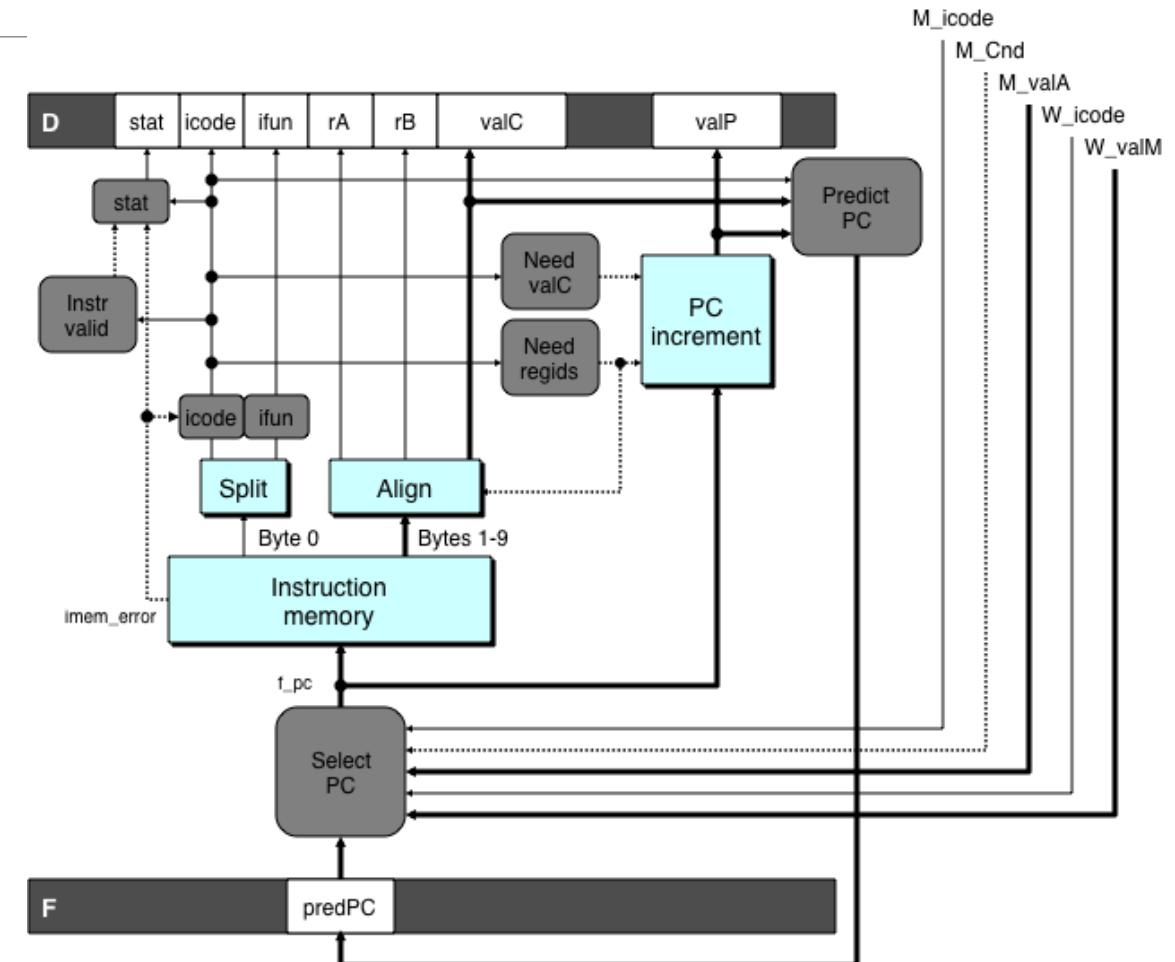
Register updates

- 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



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

Conditional Jumps

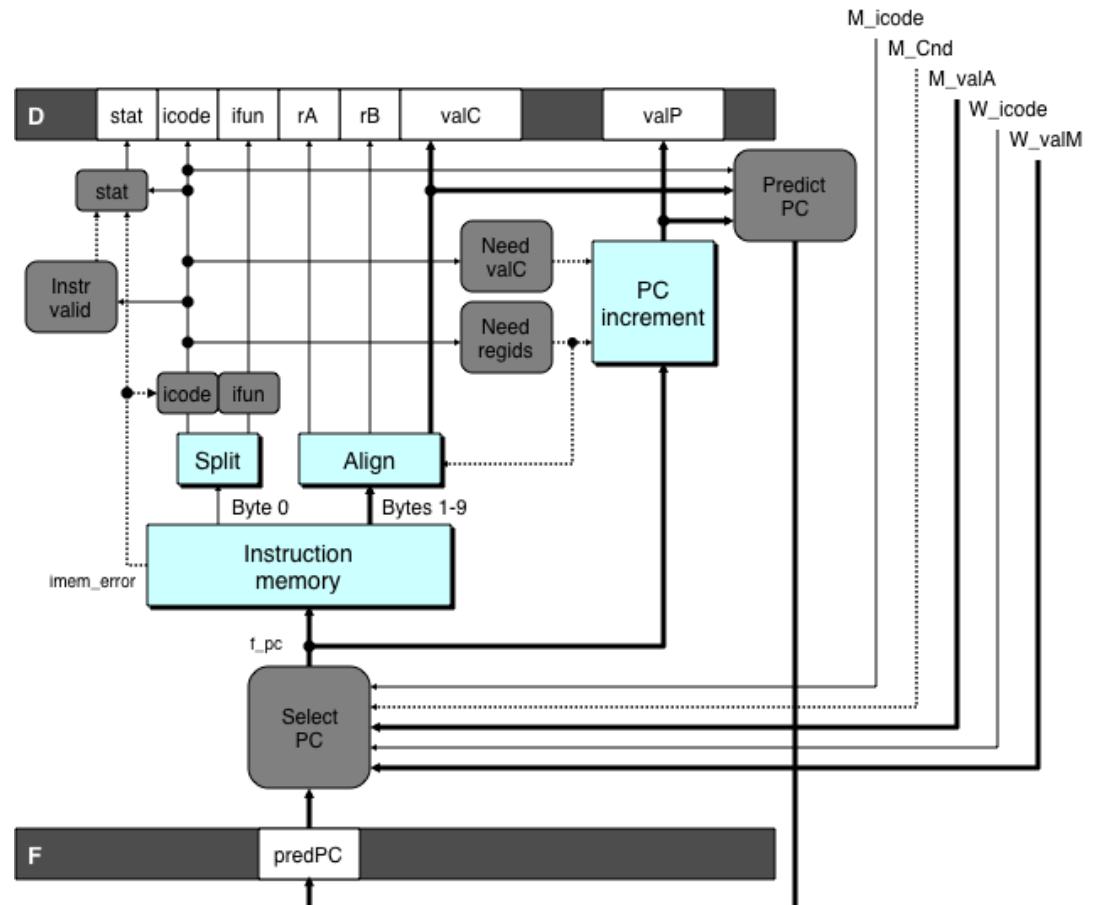
- Predict next PC to be valC (destination)
- Only correct if branch is taken
 - Typically right 60% of time

Return Instruction

- Don't try to predict

Recovering from PC Misprediction

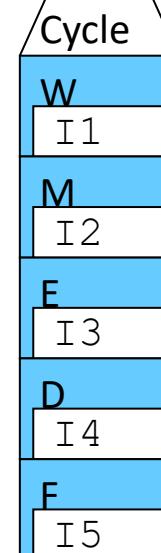
- Mispredicted Jump
 - Will see branch condition flag once instruction reaches memory stage
 - Can get fall-through PC from valA (value M_{valA})
- Return Instruction
 - Will get return PC when `ret` reaches write-back stage (W_{valM})



Pipeline Demonstration

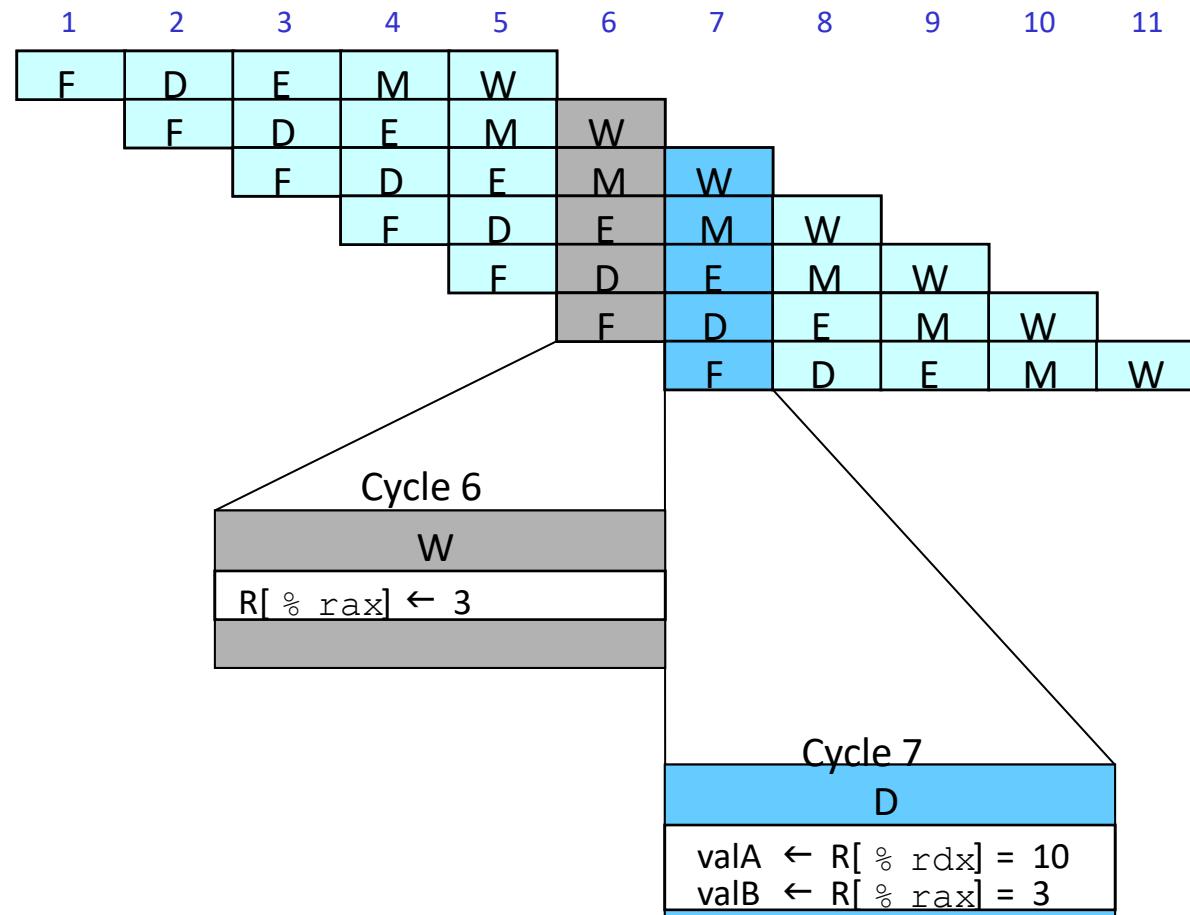
File: demo-basic.ys

			1	2	3	4	5	6	7	8	9
irmovq	\$1,%rax	#I1	F	D	E	M	W				
irmovq	\$2,%rcx	#I2		F	D	E	M	W			
irmovq	\$3,%rdx	#I3			F	D	E	M	W		
irmovq	\$4,%rbx	#I4				F	D	E	M	W	
halt		#I5					F	D	E	M	W



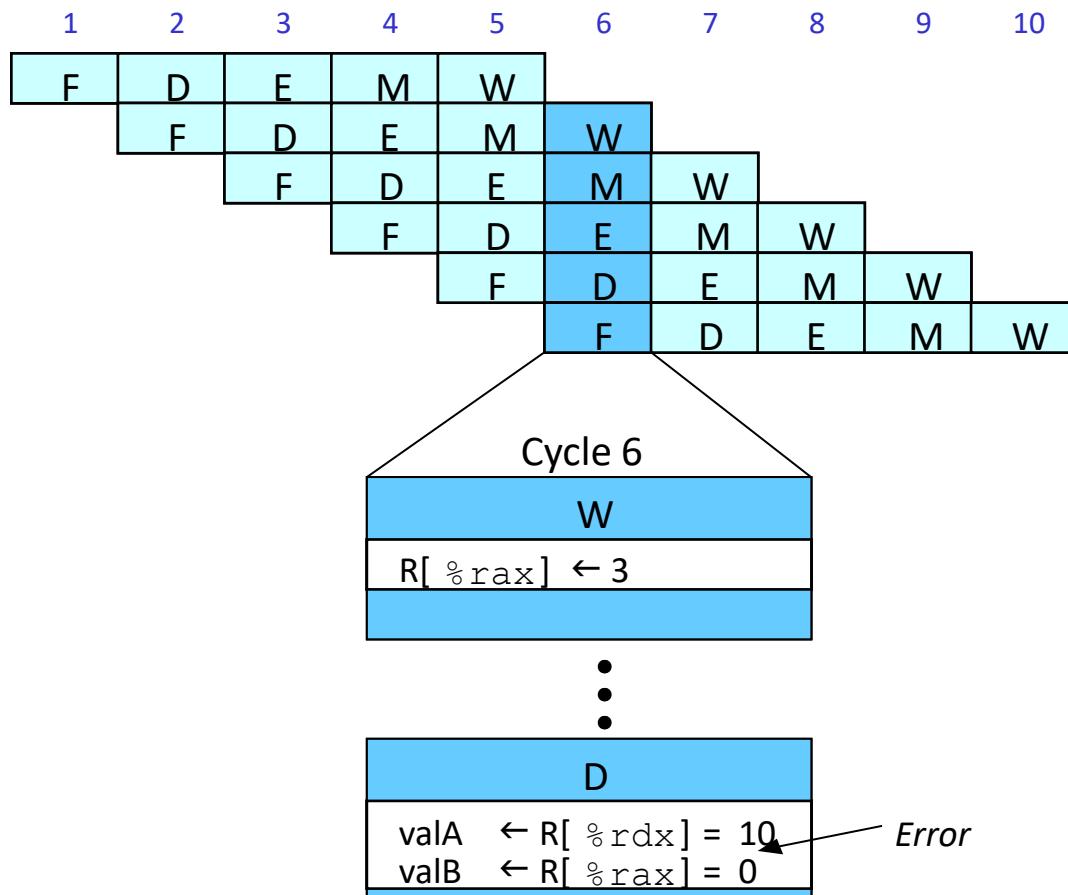
Data Dependencies: 3 Nop's

```
# demo-h3.ys
0x000:  irmovq $10,% rdx
0x00a:  irmovq $3,% rax
0x014:  nop
0x015:  nop
0x016:  nop
0x017:  addq % rdx,% rax
0x019:  halt
```



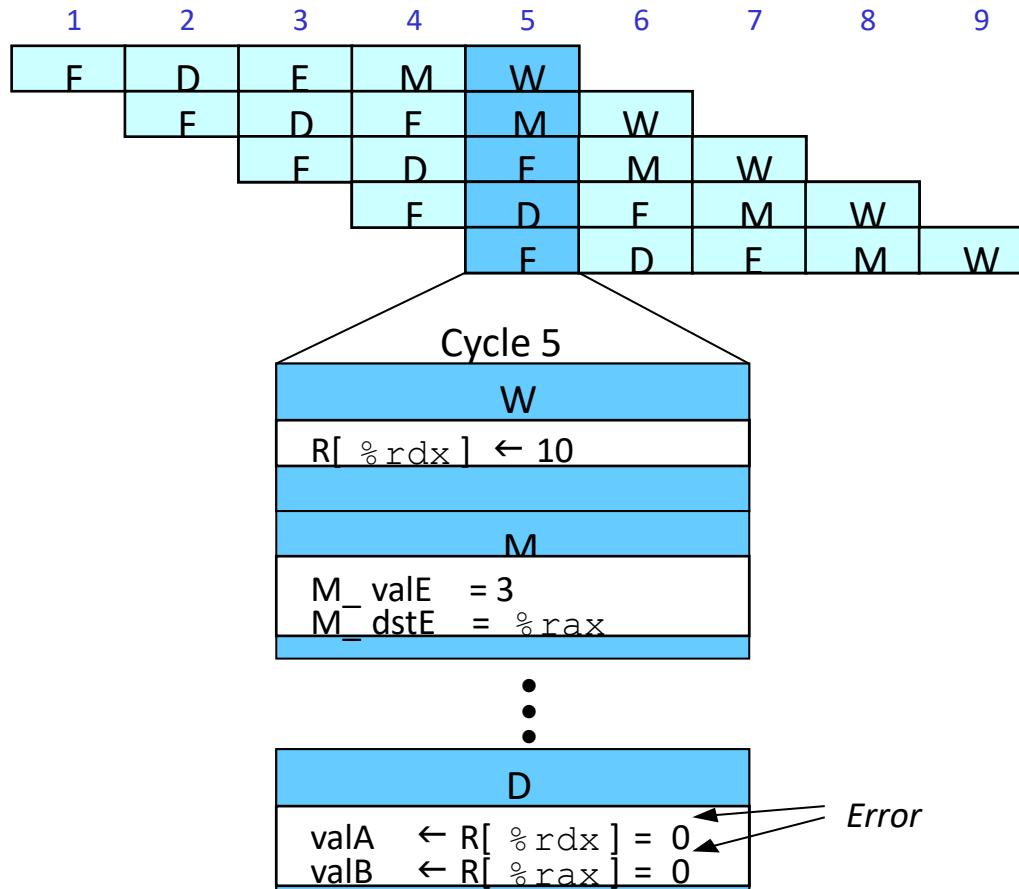
Data Dependencies: 2 Nop's

```
# demo-h2.ys
0x000:  irmovq $10,%rdx
0x00a:  irmovq $3,%rax
0x014:  nop
0x015:  nop
0x016:  addq %rdx,%rax
0x018:  halt
```



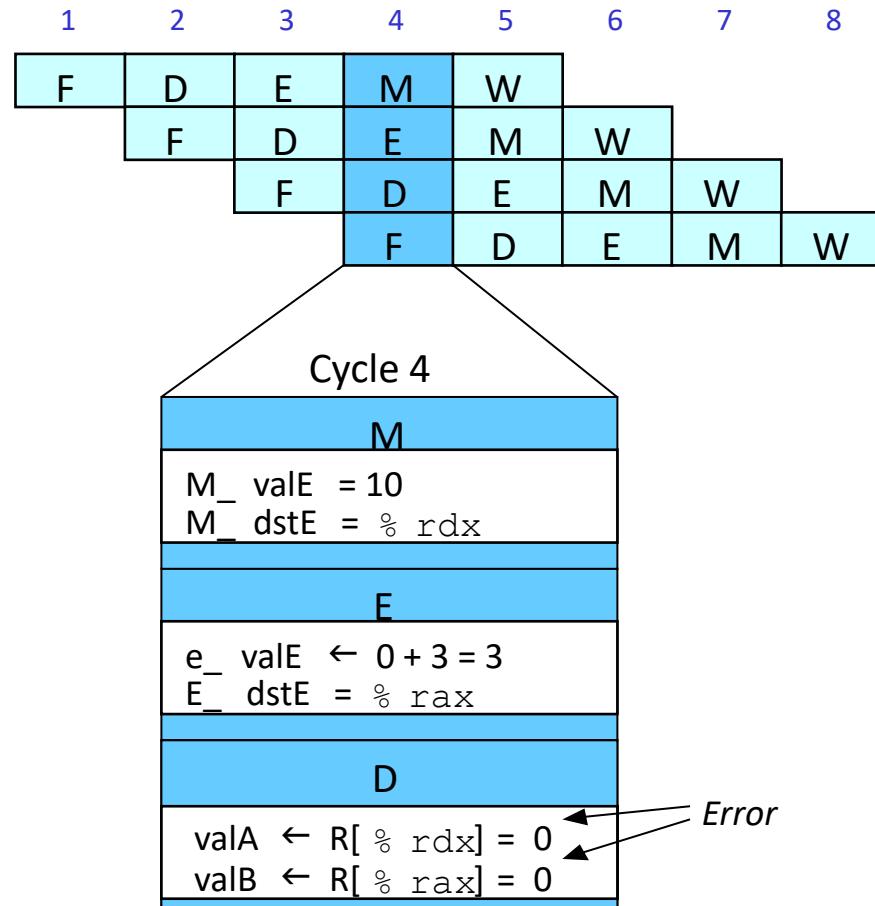
Data Dependencies: 1 Nop

```
# demo-h1.ys
0x000:  irmovq $10,%rdx
0x00a:  irmovq $3,%rax
0x014:  nop
0x015:  addq %rdx,%rax
0x017:  halt
```



Data Dependencies: No Nop

```
# demo-h0.ys
0x000:  irmovq $10,% rdx
0x00a:  irmovq $3,% rax
0x014:  addq % rdx,% rax
0x016:  halt
```

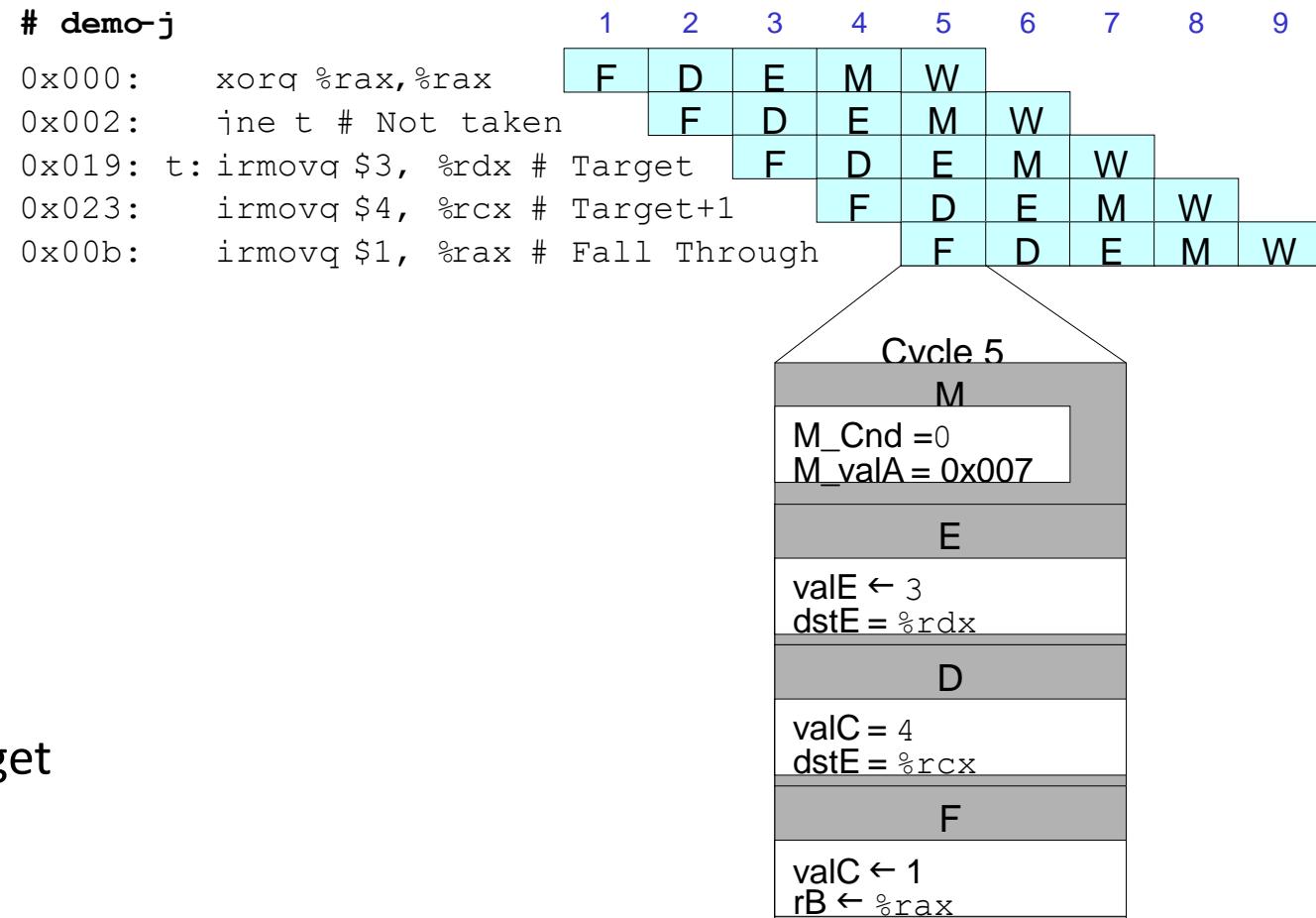


Branch Misprediction Example

demo-j.ys

```
0x000: xorq %rax,%rax
0x002: jne t          # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, %rdx # Target (Should not execute)
0x023: irmovq $4, %rcx # Should not execute
0x02d: irmovq $5, %rdx # Should not execute
```

Branch Misprediction Trace



- Incorrectly execute two instructions at branch target

Return Example

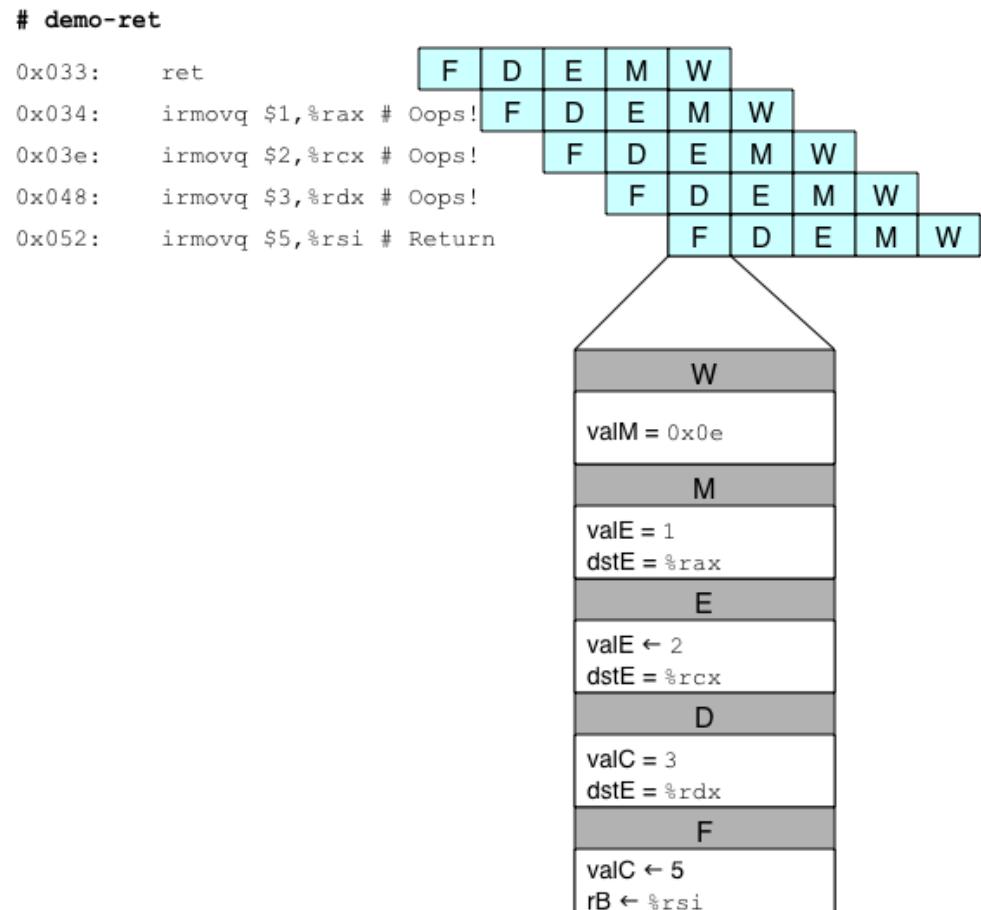
demo-ret.ys

```
0x000:    irmovq Stack,%rsp # Intialize stack pointer
0x00a:    nop                 # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p              # Procedure call
0x016:    irmovq $5,%rsi    # Return point
0x020:    halt
0x020: .pos 0x20
0x020: p:    nop             # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovq $1,%rax    # Should not be executed
0x02e:    irmovq $2,%rcx    # Should not be executed
0x038:    irmovq $3,%rdx    # Should not be executed
0x042:    irmovq $4,%rbx    # Should not be executed
0x100: .pos 0x100
0x100: Stack:               # Initial stack pointer
```

- Require lots of nops to avoid data hazards

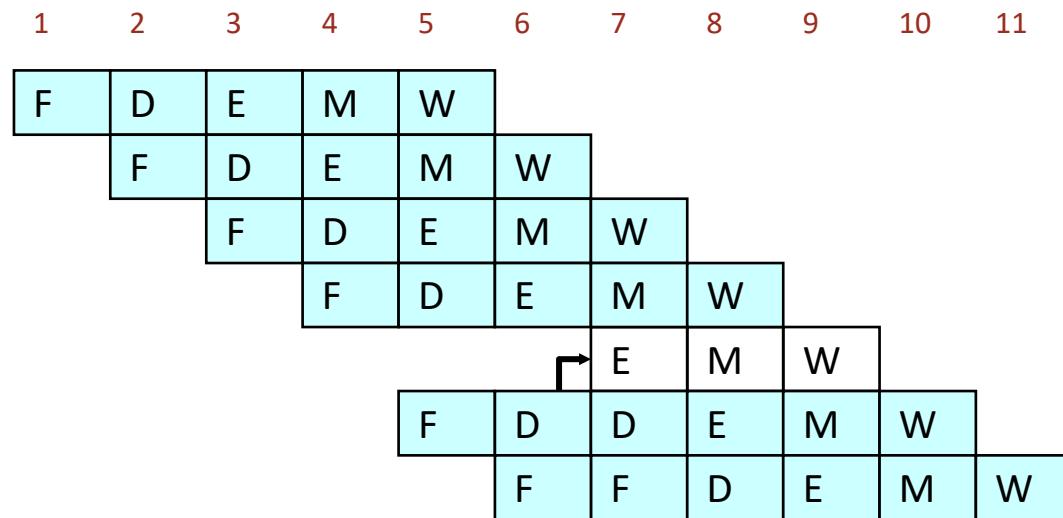
Incorrect Return Example

- Incorrectly execute 3 instructions following `ret`



Stalling for Data Dependencies

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
    bubble
0x016: addq %rdx,%rax
0x018: halt
```



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

Stall Condition

Source Registers

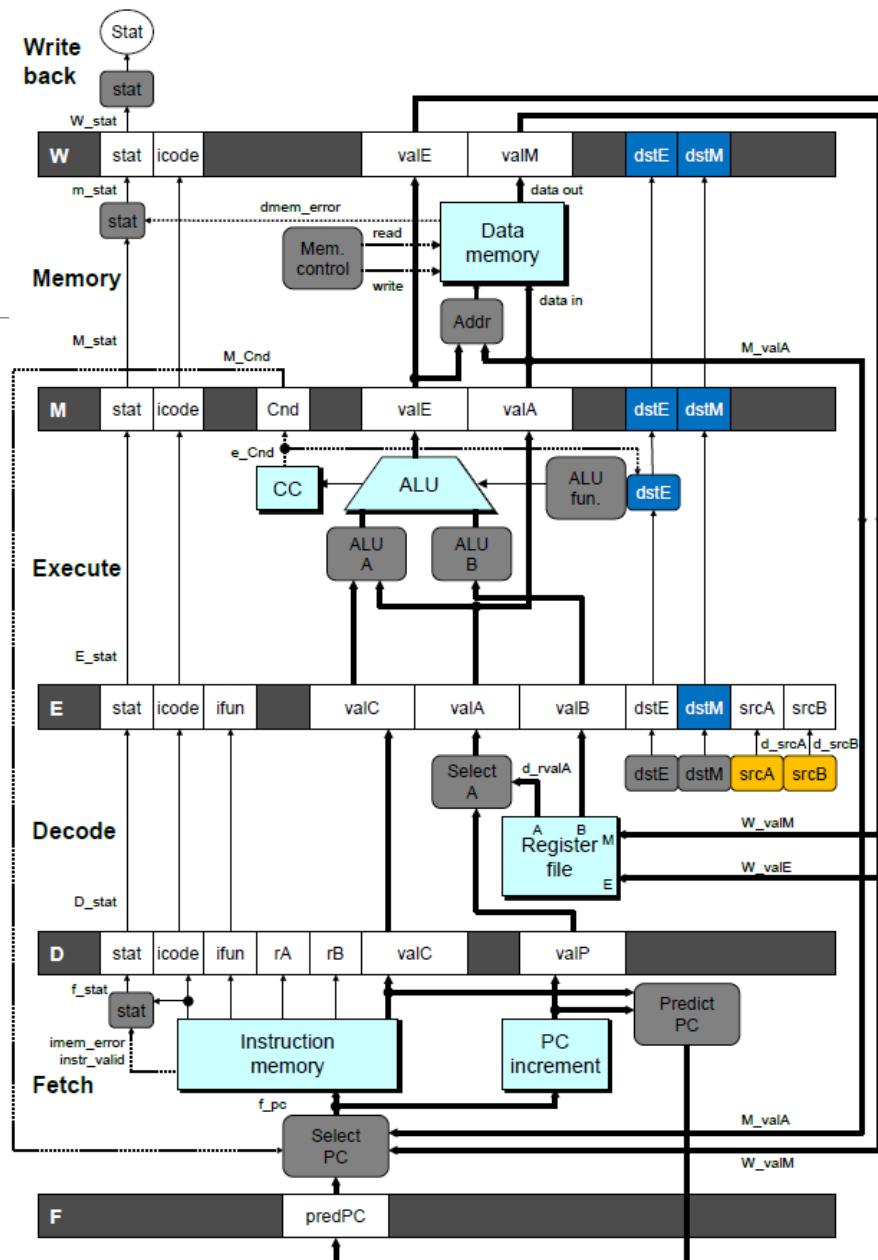
- srcA and srcB of current instruction in decode stage

Destination Registers

- dstE and dstM fields
- Instructions in execute, memory, and write-back stages

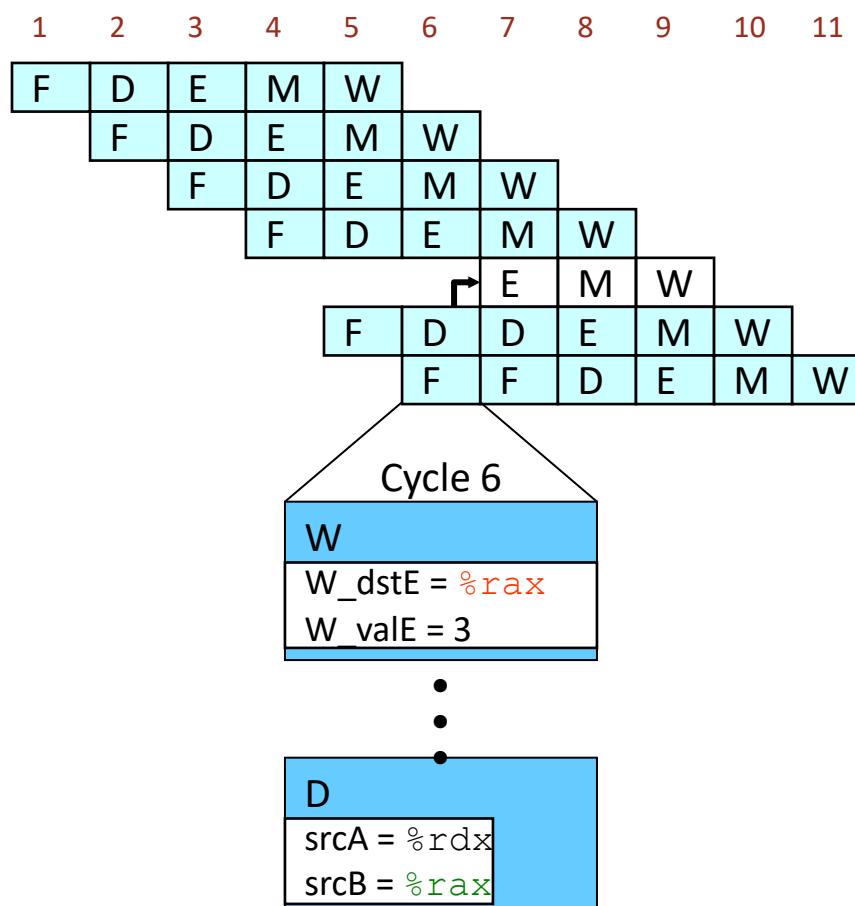
Special Case

- Don't stall for register ID 15 (0xF)
 - Indicates absence of register operand
 - Or failed cond. move



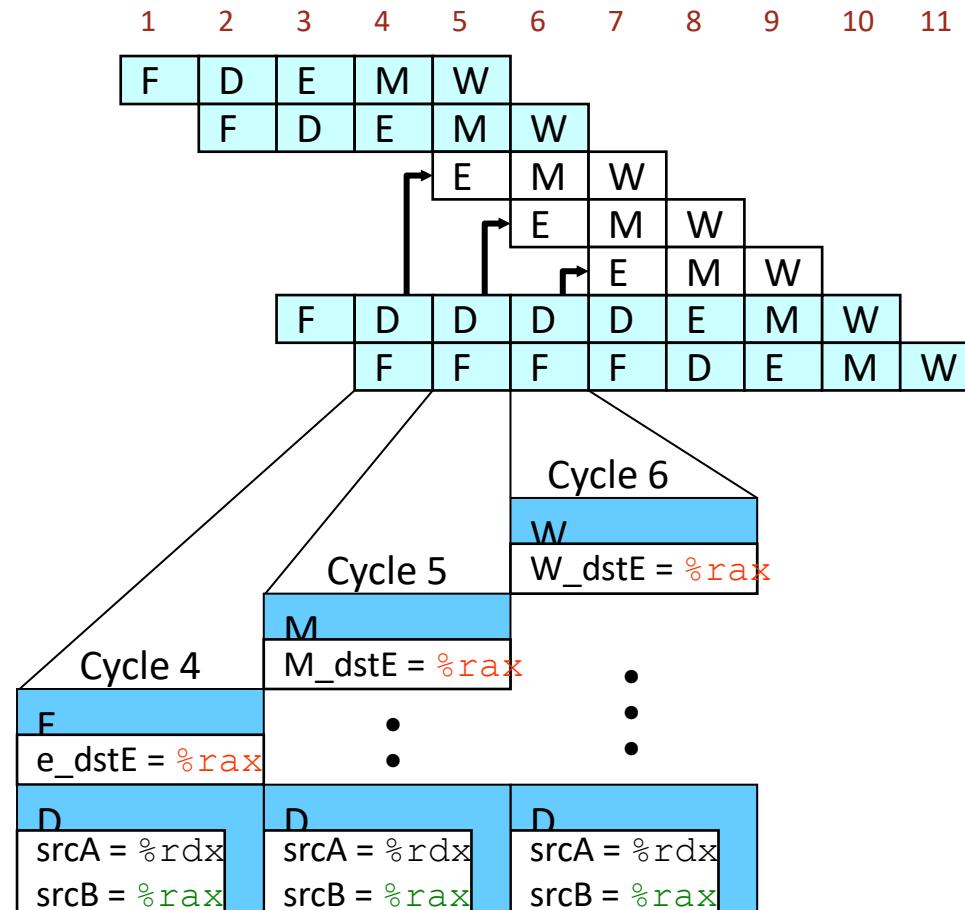
Detecting Stall Condition

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
    bubble
0x016: addq %rdx,%rax
0x018: halt
```



Stalling X3

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
bubble
bubble
bubble
0x014: addq %rdx,%rax
0x016: halt
```



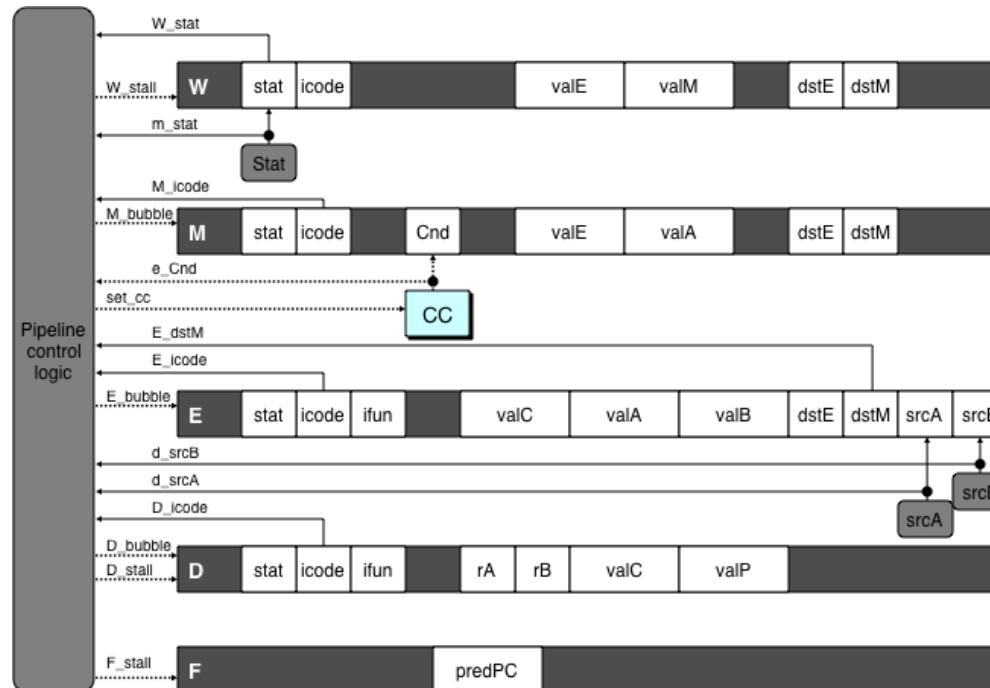
What Happens When Stalling?

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Cycle 8	
Write Back	<i>bubble</i>
Memory	<i>bubble</i>
Execute	0x014: addq %rdx,%rax
Decode	0x016: halt
Fetch	

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
 - Like dynamically generated nop's
 - Move through later stages

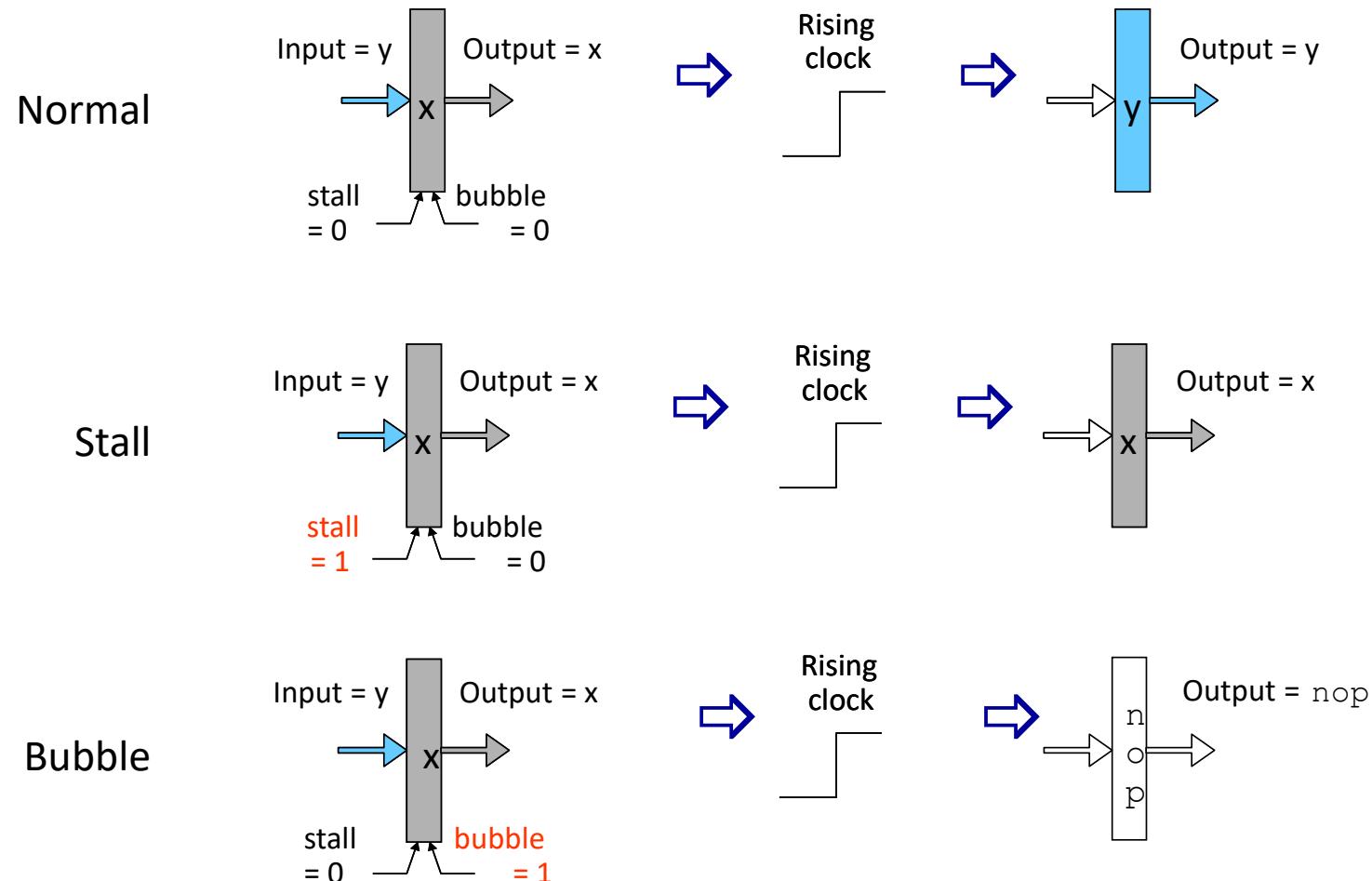
Implementing Stalling



Pipeline Control

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

Pipeline Register Modes



Data Forwarding

Naïve Pipeline

- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
 - Needs to be in register file at start of stage

Observation

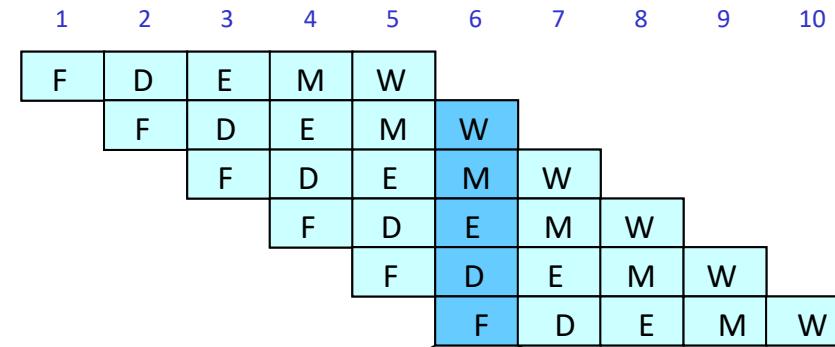
- Value generated in execute or memory stage

Trick

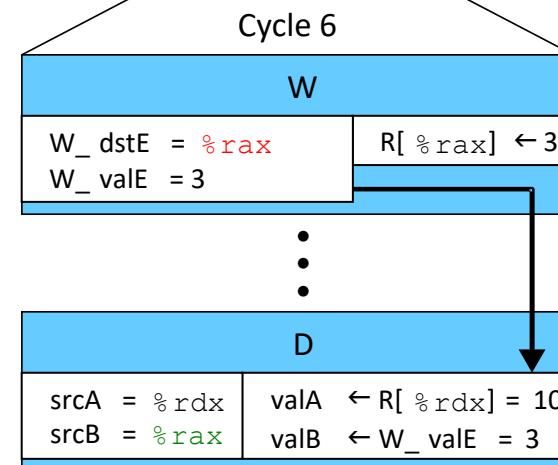
- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage

Data Forwarding Example

```
# demo-h2.ys
0x000:  irmovq $10,% rdx
0x00a:  irmovq $3,% rax
0x014:  nop
0x015:  nop
0x016:  addq % rdx,% rax
0x018:  halt
```



- `irmovq` in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage



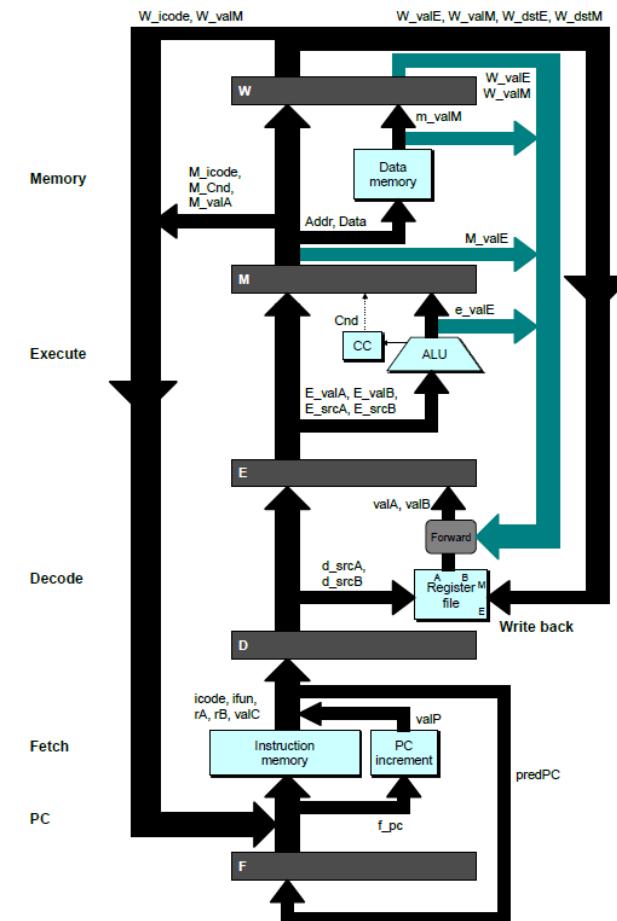
Bypass Paths

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 Forwarding Example #2

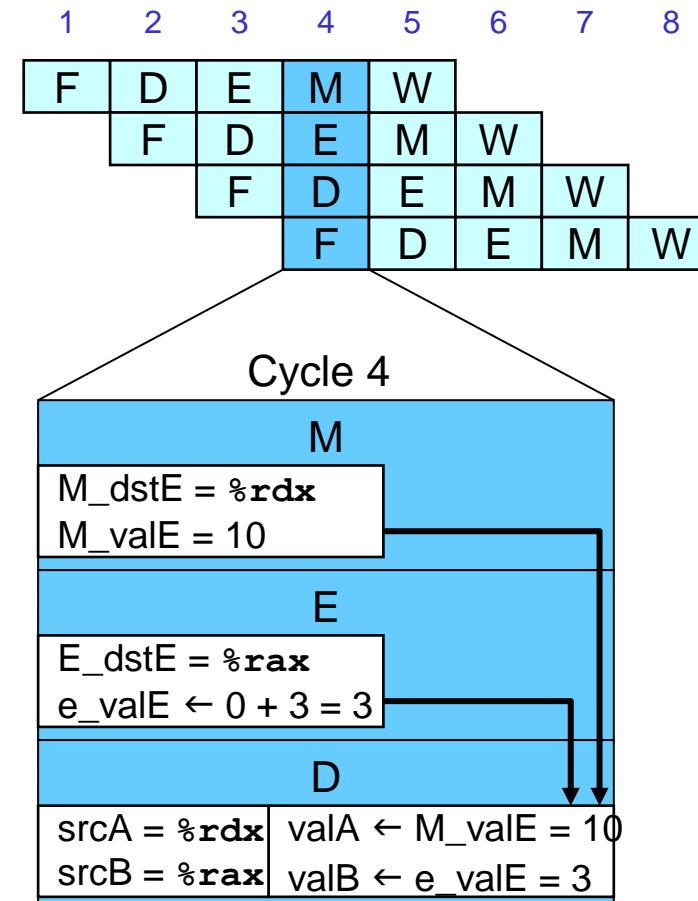
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Register %rdx

- Generated by ALU during previous cycle
- Forward from memory as valA

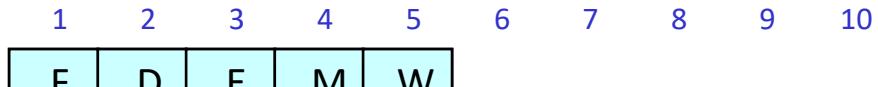
Register %rax

- Value just generated by ALU
- Forward from execute as valB



Forwarding Priority

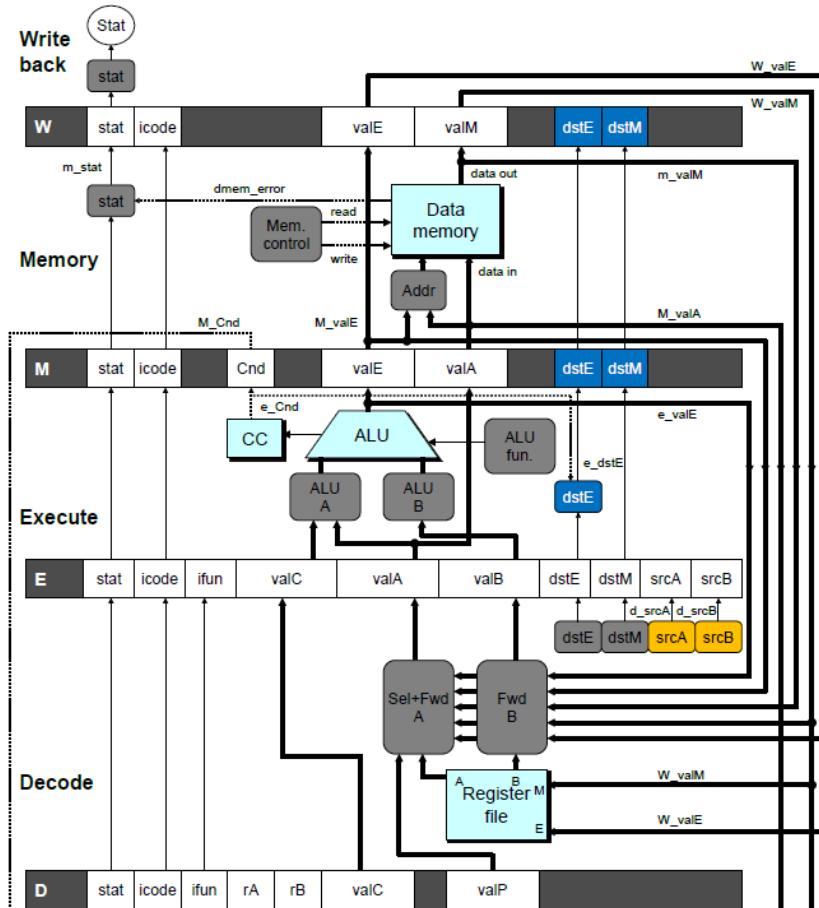
```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```



Multiple Forwarding Choices

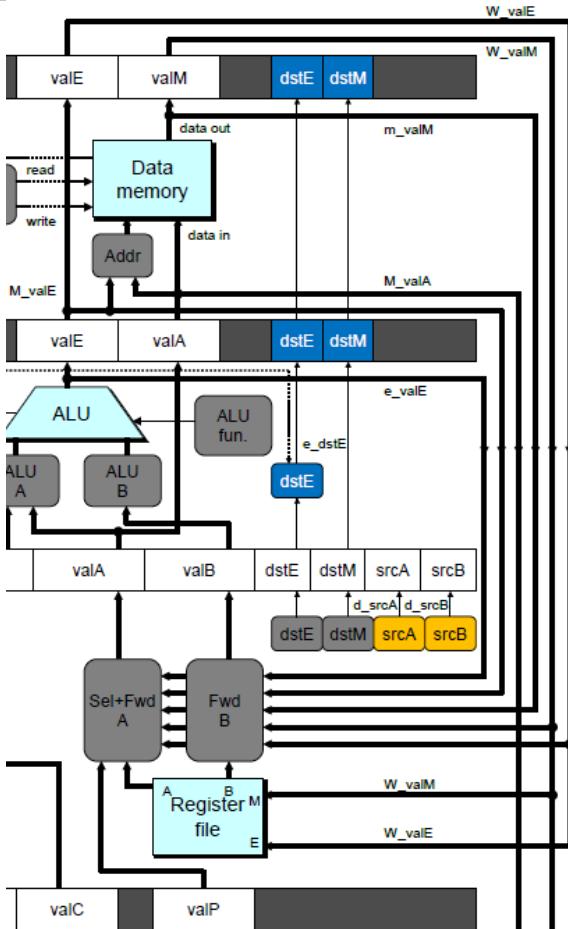
- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage

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



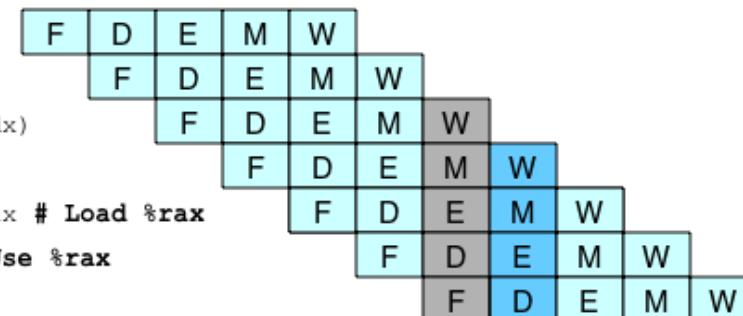
```
## What should be the A value?  
int d_valA = [  
    # Use incremented PC  
    D_icode in { ICALL, IJXX } : D_valP;  
    # Forward valE from execute  
    d_srcA == e_dstE : e_valE;  
    # Forward valM from memory  
    d_srcA == M_dstM : m_valM;  
    # Forward valE from memory  
    d_srcA == 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

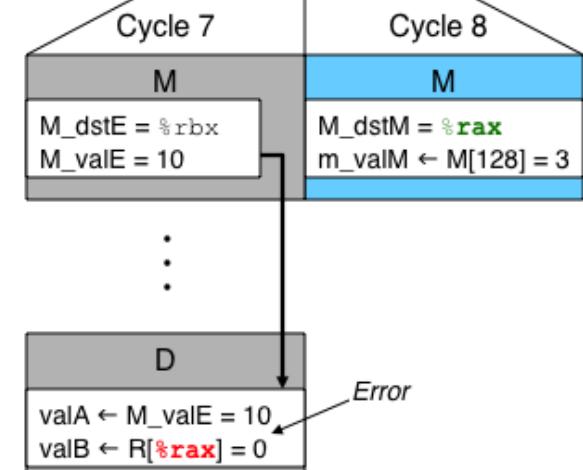
```
0x000: irmovq $128,%rdx
0x00a: irmovq $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```

1 2 3 4 5 6 7 8 9 10 11



Load-use dependency

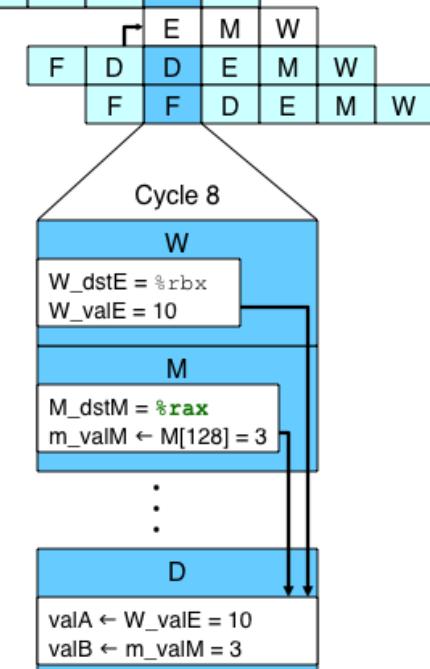
- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8



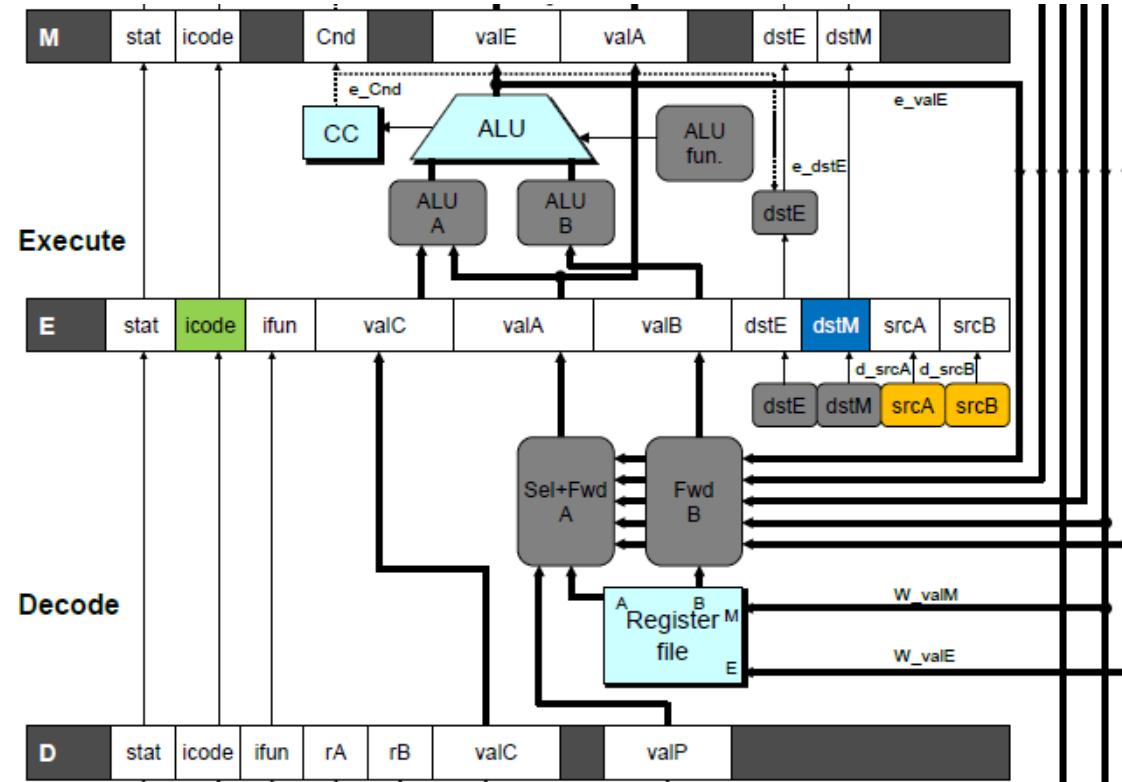
Avoiding Load/Use Hazard

# demo-luh.ys	1	2	3	4	5	6	7	8	9	10	11	12
0x000: irmovq \$128,%rdx	F	D	E	M	W							
0x00a: irmovq \$3,%rcx	F	D	E	M	W							
0x014: rmmovq %rcx, 0(%rdx)	F	D	E	M	W							
0x01e: irmovq \$10,%rbx	F	D	E	M	W							
0x028: mrmovq 0(%rdx),%rax # Load %rax	F	D	E	M	W							
<i>bubble</i>												
0x032: addq %rbx,%rax # Use %rax	F	D	E	M	W							
0x034: halt	F	F	D	E	M	W						

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



Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	<code>E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }</code>

Control for Load/Use Hazard

demo-luh.ys

	1	2	3	4	5	6	7	8	9	10	11	12
0x000: irmovq \$128,%rdx	F	D	E	M	W							
0x00a: irmovq \$3,%rcx		F	D	E	M	W						
0x014: rmmovq %rcx, 0(%rdx)			F	D	E	M	W					
0x01e: irmovq \$10,%ebx				F	D	E	M	W				
0x028: mrmovq 0(%rdx),%rax # Load %rax					F	D	E	M	W			
bubble												
0x032: addq %ebx,%rax # Use %rax						F	D	D	E	M	W	
0x034: halt							F	F	D	E	M	W

- 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

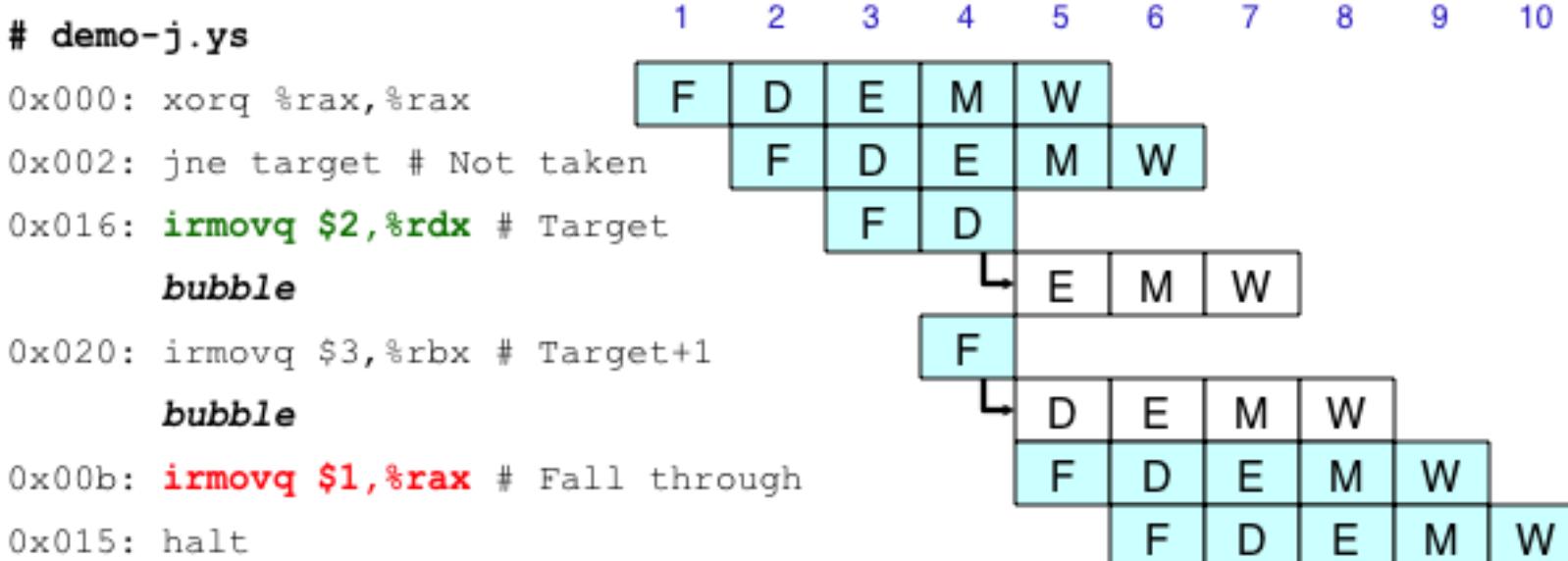
Branch Misprediction Example

demo-j.ys

```
0x000: xorq %rax,%rax
0x002: jne t          # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, %rdx # Target
0x023: irmovq $4, %rcx # Should not execute
0x02d: irmovq $5, %rdx # Should not execute
```

- Should only execute first 8 instructions

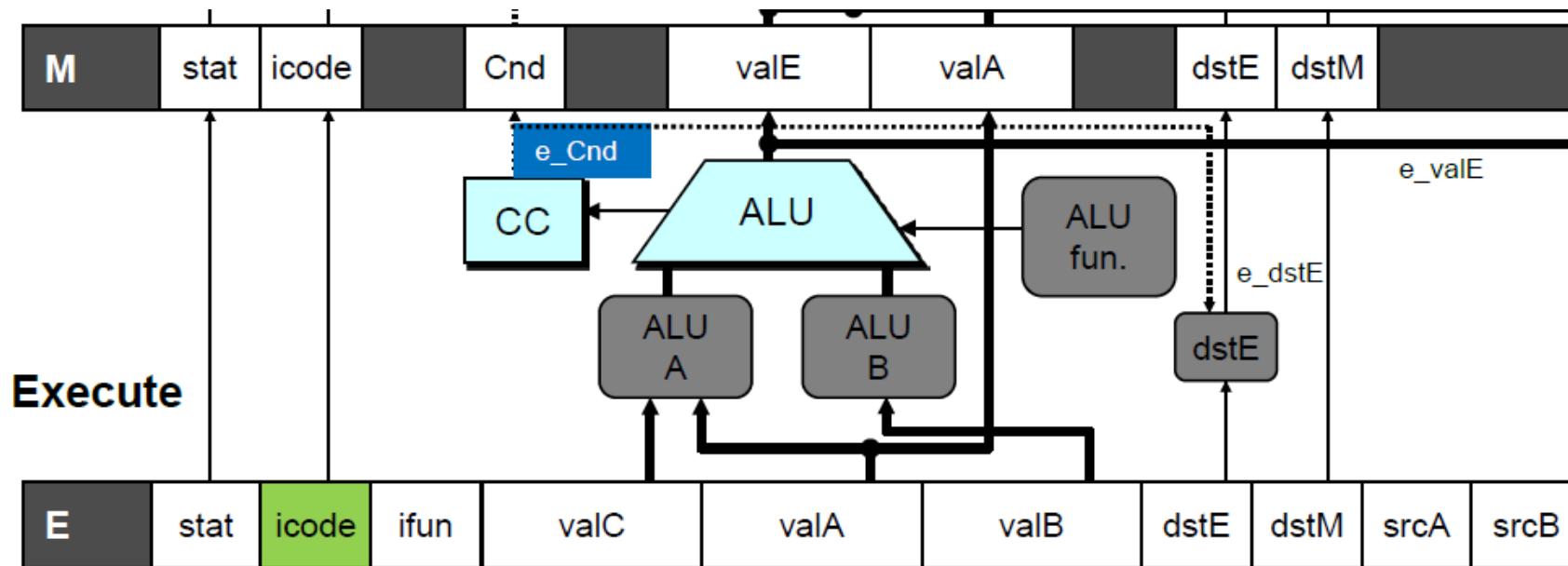
Handling Misprediction



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

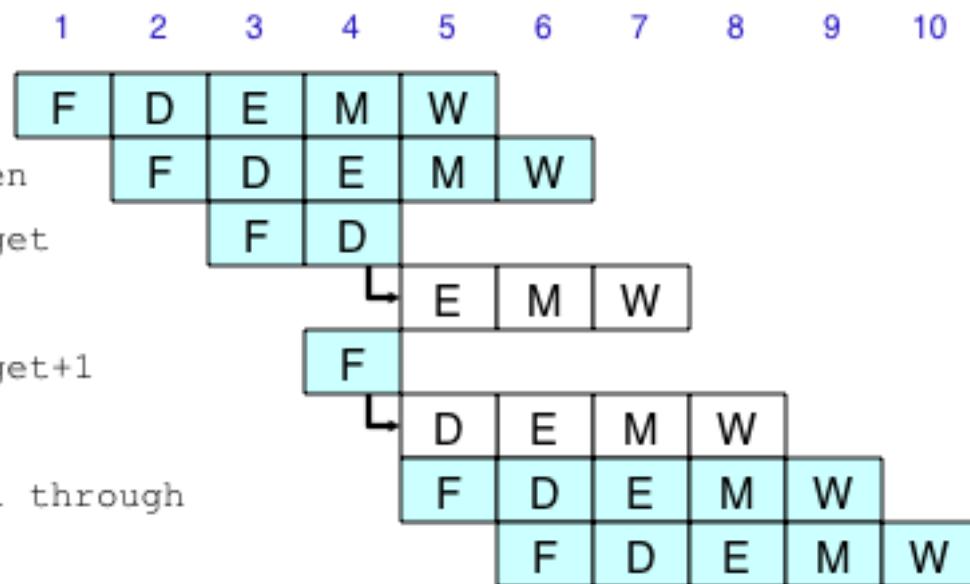
Detecting Mispredicted Branch



Control for Misprediction

```
# demo-j.ys
```

```
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
          bubble
0x020: irmovq $3,%rbx # Target+1
          bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



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

Return Example

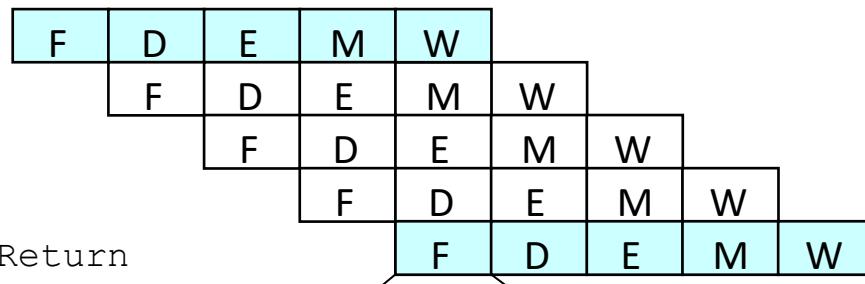
demo-retb.ys

```
0x000:    irmovq Stack,%rsp    # Intialize stack pointer
0x00a:    call p              # Procedure call
0x013:    irmovq $5,%rsi     # Return point
0x01d:    halt
0x020: .pos 0x20
0x020: p:    irmovq $-1,%rdi   # procedure
0x02a:    ret
0x02b:    irmovq $1,%rax     # Should not be executed
0x035:    irmovq $2,%rcx     # Should not be executed
0x03f:    irmovq $3,%rdx     # Should not be executed
0x049:    irmovq $4,%rbx     # Should not be executed
0x100: .pos 0x100
0x100: Stack:                # Stack: Stack pointer
```

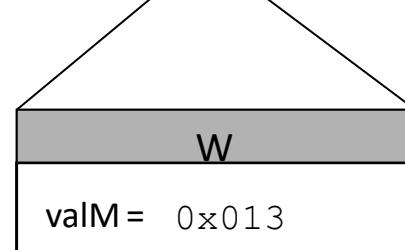
- Previously executed three additional instructions

Correct Return Example

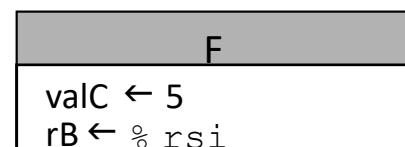
```
# demo- retb  
0x026:    ret  
           bubble  
           bubble  
           bubble  
0x013:    irmovq$5,%rsi # Return
```



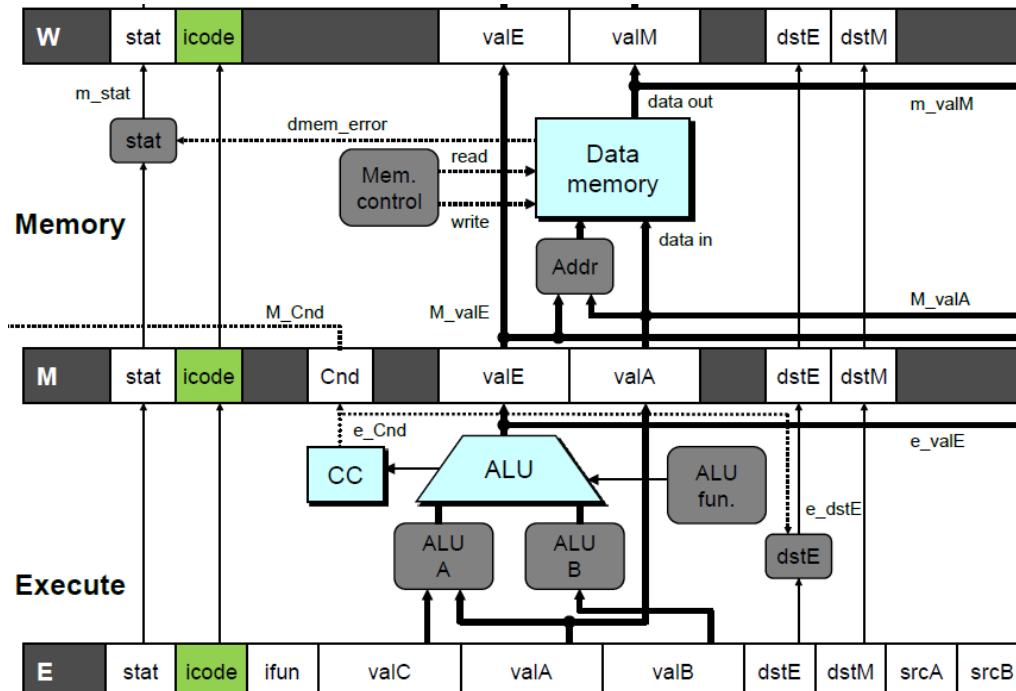
- As `ret` passes through pipeline, stall at fetch stage
 - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage



⋮



Detecting Return



Condition	Trigger
Processing <code>ret</code>	<code>IRET in { D_icode, E_icode, M_icode }</code>

Control for Return

```
# demo-retb
```

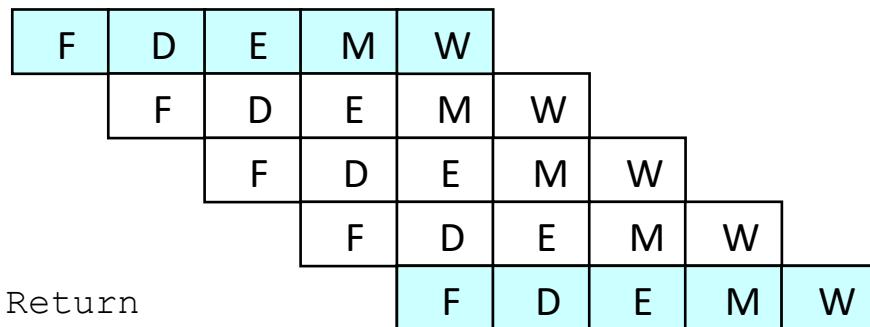
```
0x026:    ret
```

bubble

bubble

bubble

```
0x014:    irmovq $5,%rsi # Return
```



Condition	F	D	E	M	W
Processing <code>ret</code>	stall	bubble	normal	normal	normal

Initial Version of Pipeline Control

```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && 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 { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };
```



```
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 };
```



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

Thank You!

Computer Systems Organization (CS2.201)

MEMORY HIERARCHY (SECTION 6.1-6.3.1)

Deepak Gangadharan
Computer Systems Group (CSG), IIIT Hyderabad

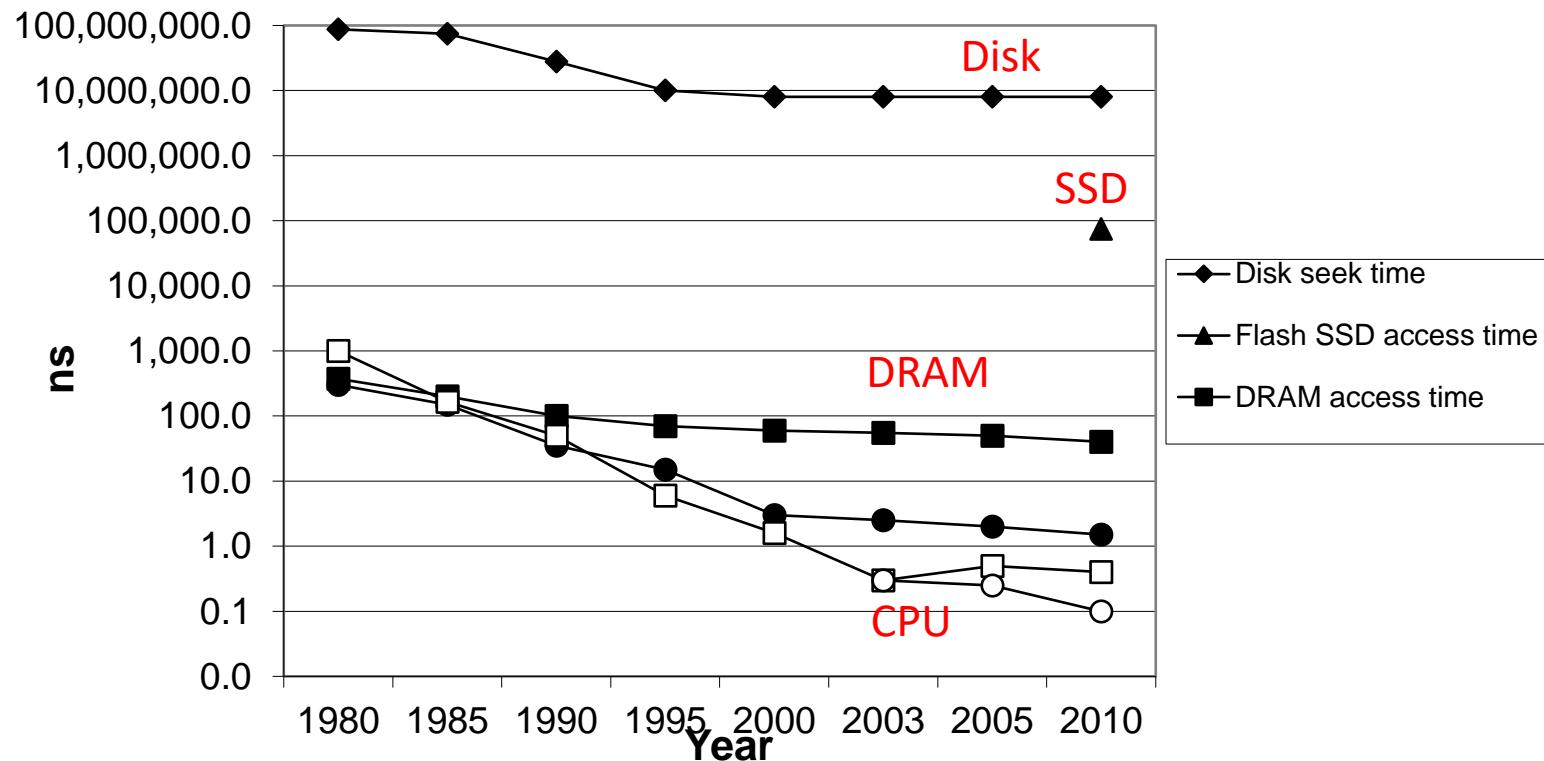
Slide Contents: Adapted from slides by Randal Bryant

Topics

- Locality of reference
- Caching in the memory hierarchy

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



Locality to the Rescue!

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as **locality**

Topics

Storage technologies and trends

Locality of reference

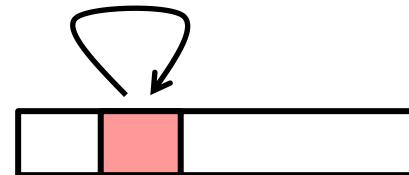
Caching in the memory hierarchy

Locality

Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

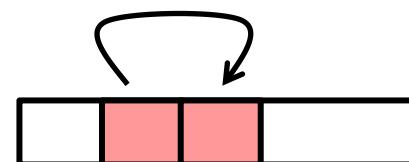
Temporal locality:

- Recently referenced items are likely to be referenced again in the near future



Spatial locality:

- Items with nearby addresses tend to be referenced close together in time



Locality Example

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

Data references

- Reference array elements in succession (stride-1 reference pattern). Spatial locality
- Reference variable `sum` each iteration. Temporal locality

Instruction references

- Reference instructions in sequence. Spatial locality
- Cycle through loop repeatedly. Temporal locality

Qualitative Estimates of Locality

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality with respect to array a ?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

Question: Does this function have good locality with respect to array a?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

Question: Can you permute the loops so that the function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int a[M] [N] [N] )
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k] [i] [j];
    return sum;
}
```

Memory Hierarchies

Some fundamental and enduring properties of hardware and software:

- Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
- The gap between CPU and main memory speed is widening.
- Well-written programs tend to exhibit good locality.

These fundamental properties complement each other beautifully.

They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

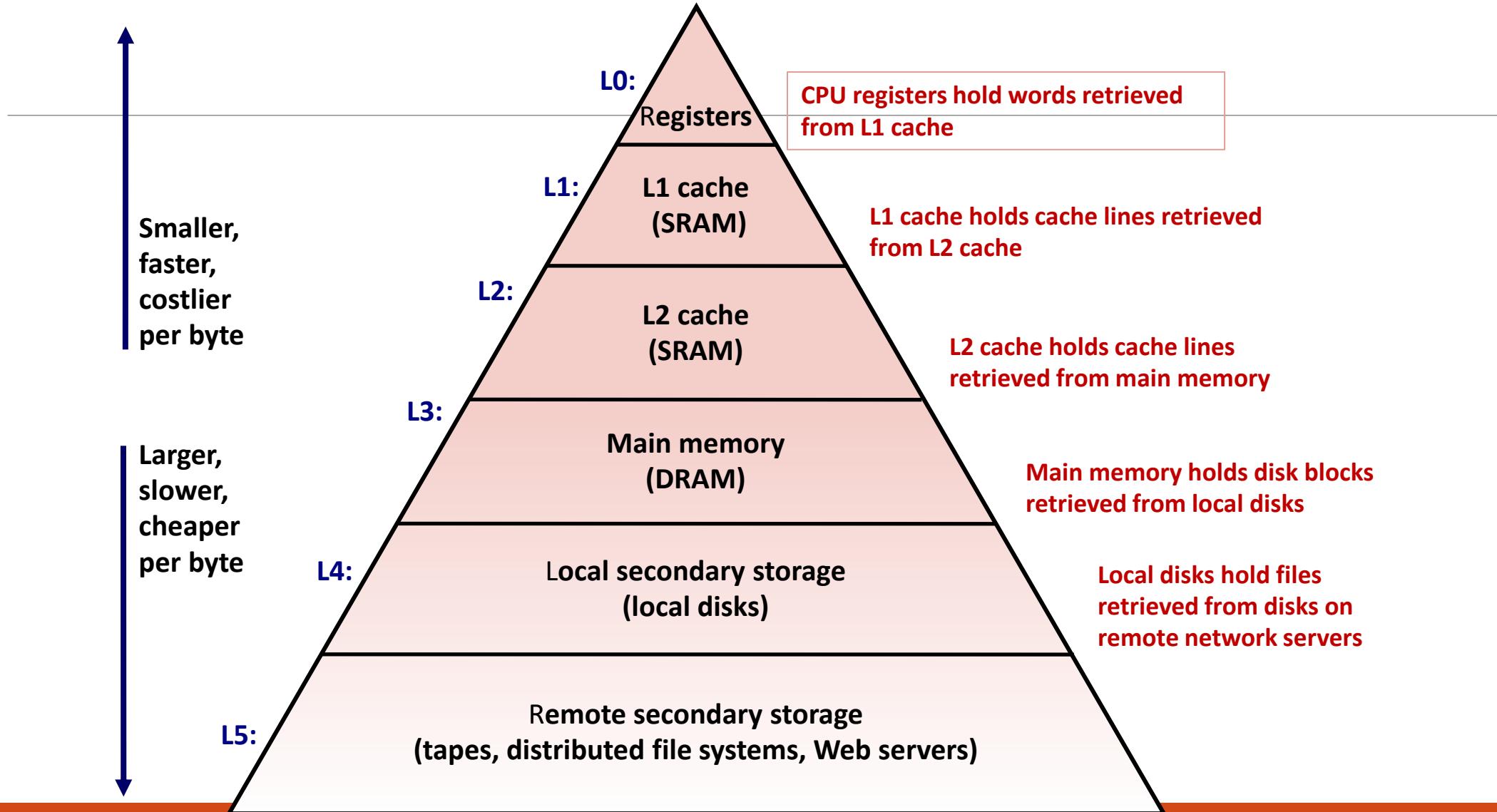
Today

Storage technologies and trends

Locality of reference

Caching in the memory hierarchy

An Example Memory Hierarchy



Caches

Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

Fundamental idea of a memory hierarchy:

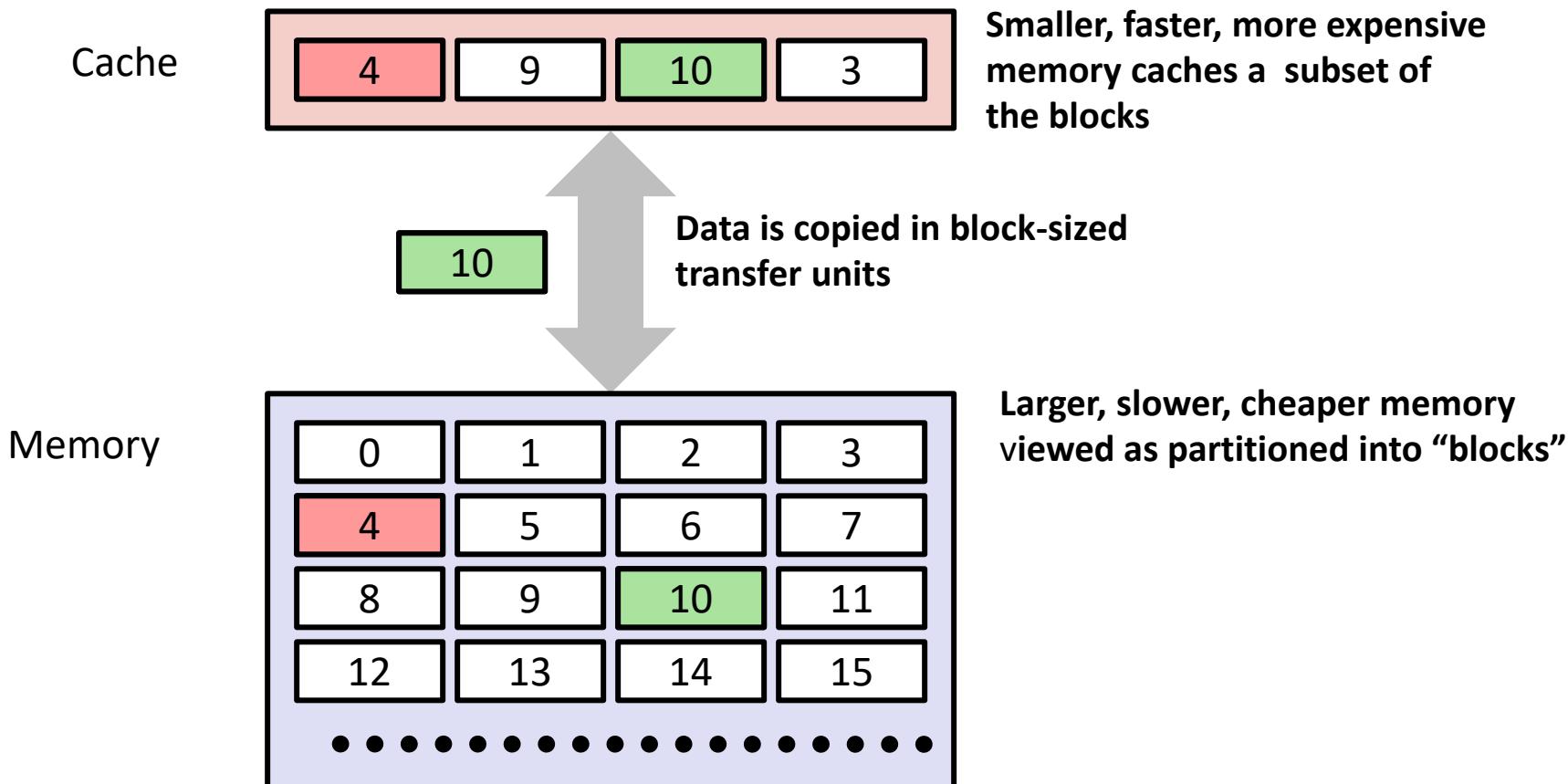
- For each k , the faster, smaller device at level k serves as a cache for the larger, slower device at level $k+1$.

Why do memory hierarchies work?

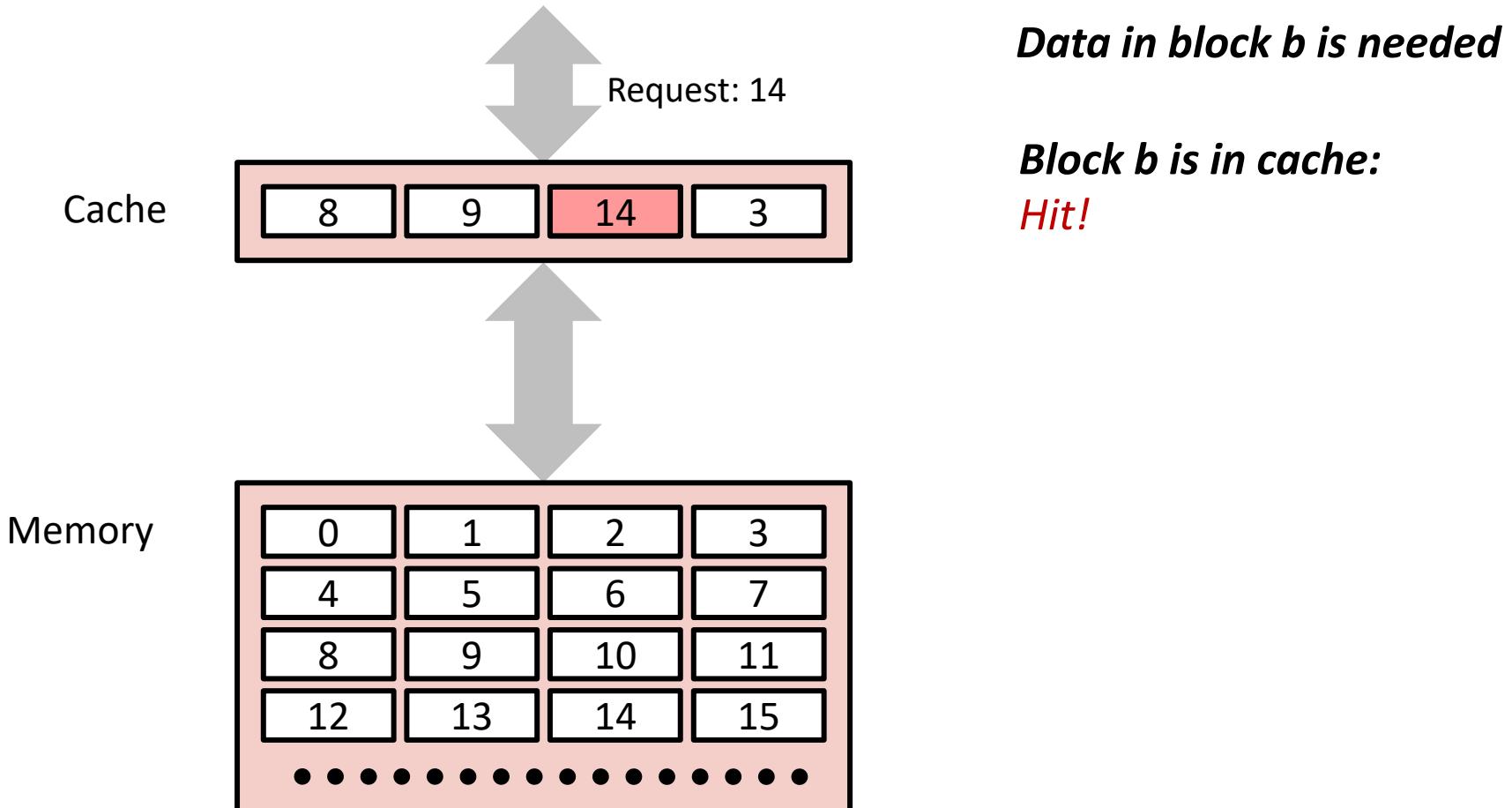
- Because of locality, programs tend to access the data at level k more often than they access the data at level $k+1$.
- Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.

Big Idea: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

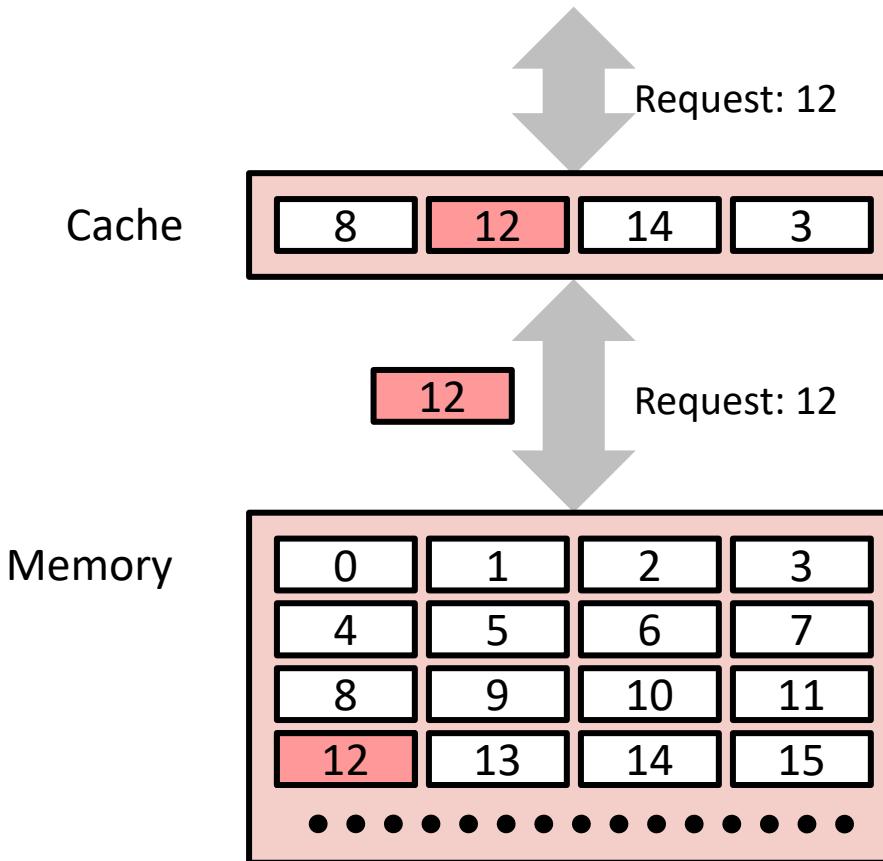
General Cache Concepts



General Cache Concepts: Hit



General Cache Concepts: Miss



Data in block b is needed

***Block b is not in cache:
Miss!***

***Block b is fetched from
memory***

Block b is stored in cache

- **Placement policy:**
determines where b goes
- **Replacement policy:**
determines which block gets evicted (victim)

General Caching Concepts: Types of Cache Misses

Cold (compulsory) miss

- Cold misses occur because the cache is empty.

Conflict miss

- Most caches limit blocks at level $k+1$ to a small subset (sometimes a singleton) of the block positions at level k .
 - E.g. Block i at level $k+1$ must be placed in block $(i \bmod 4)$ at level k .
- Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
 - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

Capacity miss

- Occurs when the set of active cache blocks (**working set**) is larger than the cache.

Topics

Cache memory organization and operation

Performance impact of caches

- Rearranging loops to improve spatial locality
- Using blocking to improve temporal locality

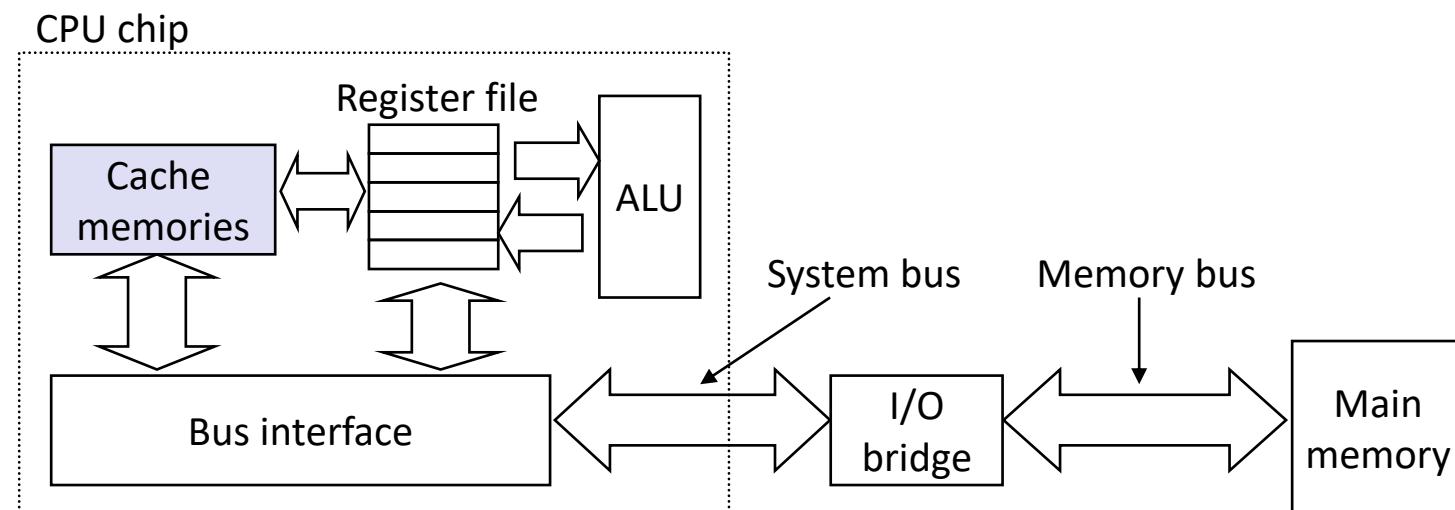
Cache Memories

Cache memories are small, fast SRAM-based memories managed automatically in hardware.

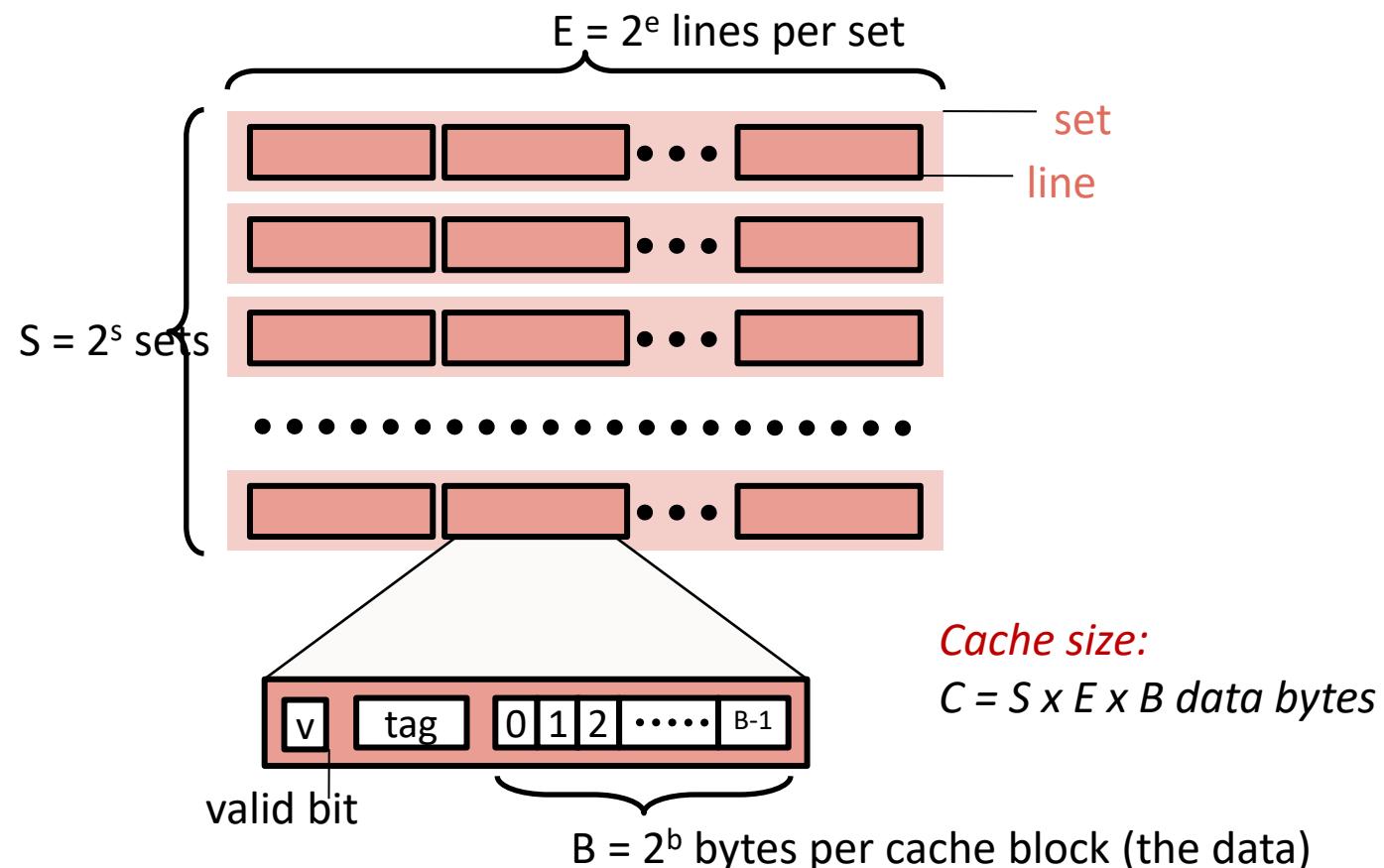
- Hold frequently accessed blocks of main memory

CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.

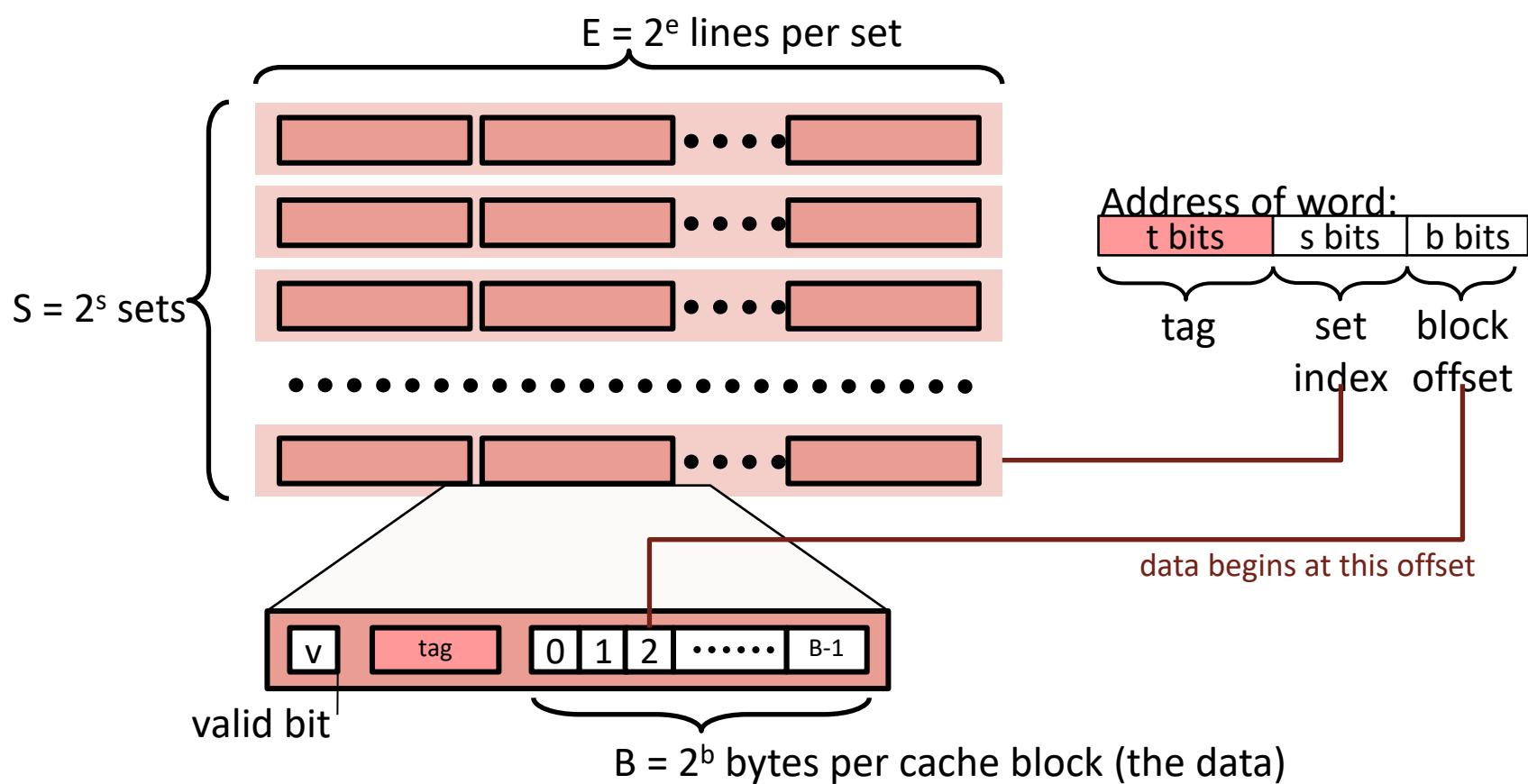
Typical system structure:



General Cache Organization (S, E, B)



Cache Read

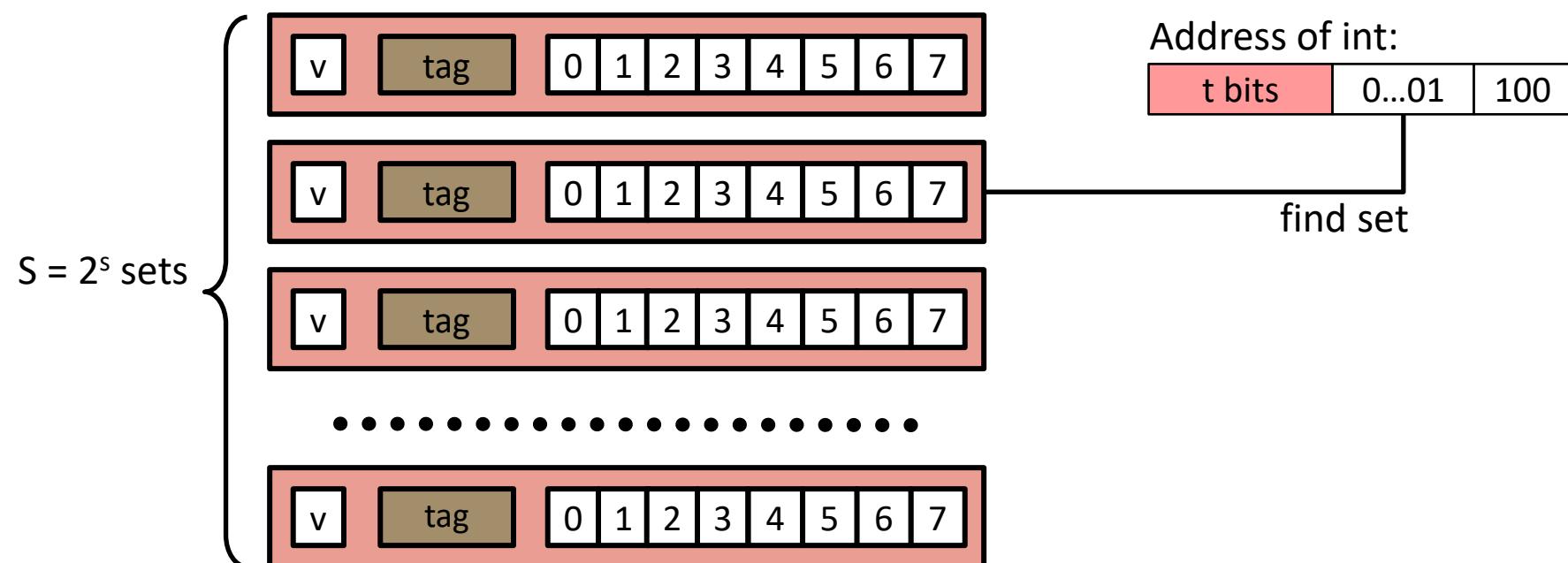


- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Example: Direct Mapped Cache ($E = 1$)

Direct mapped: One line per set

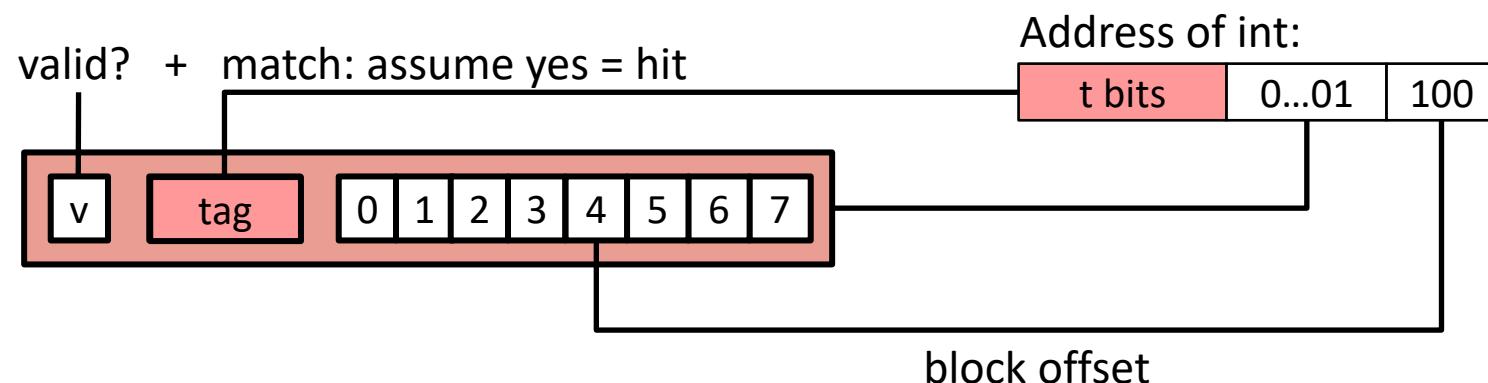
Assume: cache block size 8 bytes



Example: Direct Mapped Cache ($E = 1$)

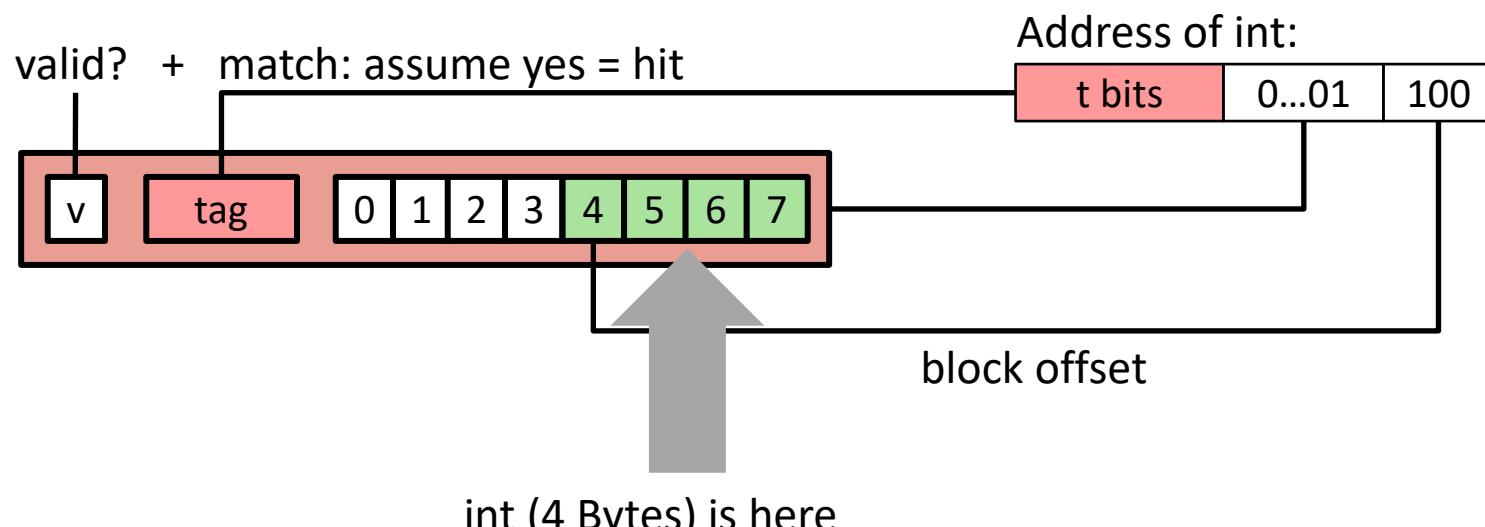
Direct mapped: One line per set

Assume: cache block size 8 bytes



Example: Direct Mapped Cache ($E = 1$)

Direct mapped: One line per set
Assume: cache block size 8 bytes



No match: old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

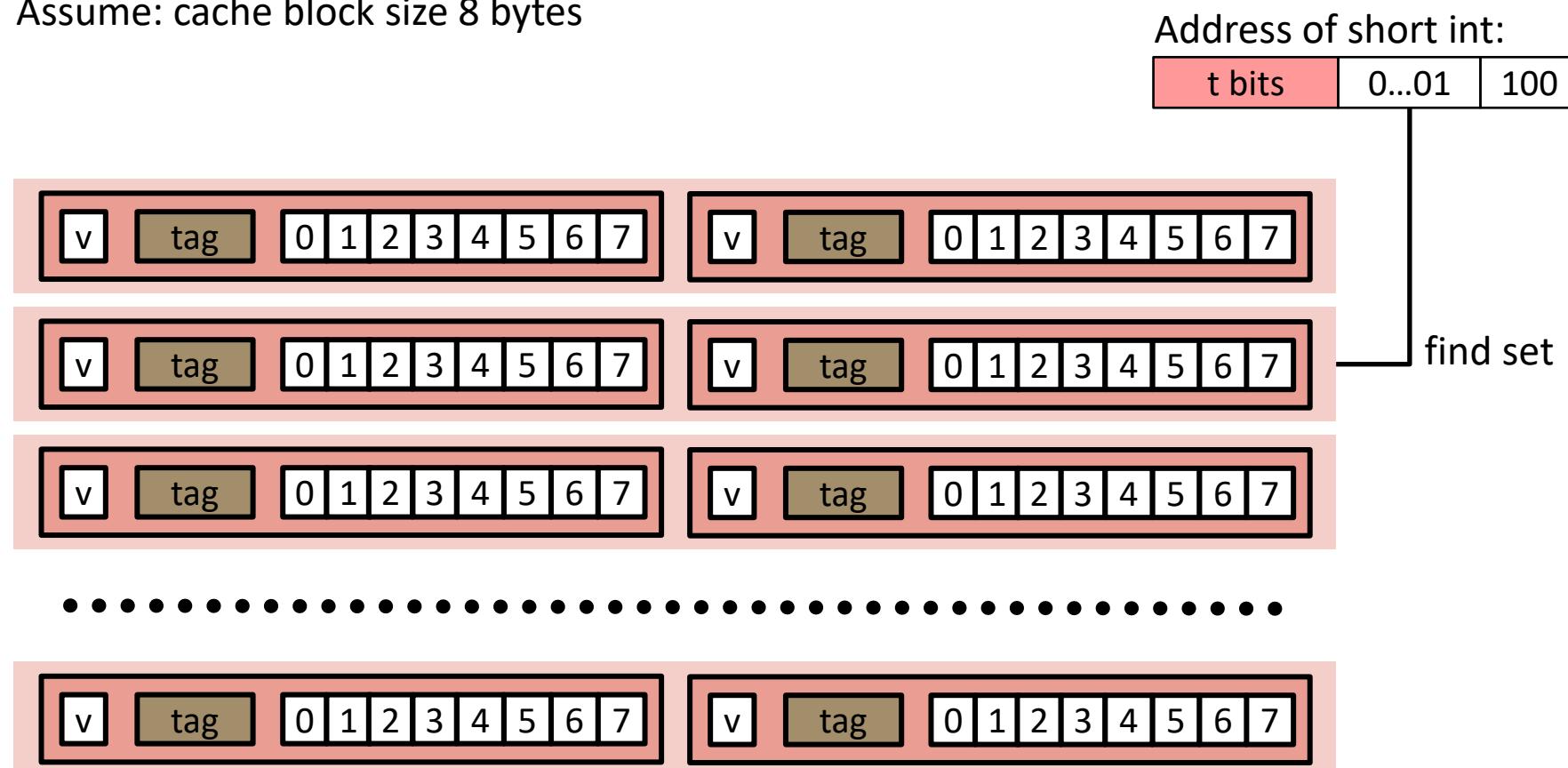
Address trace (reads, one byte per read):

- | | | |
|---|-----------------------|------|
| 0 | [0000] ₂ , | miss |
| 1 | [0001] ₂ , | hit |
| 7 | [0111] ₂ , | miss |
| 8 | [1000] ₂ , | miss |
| 0 | [0000] ₂ | miss |

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

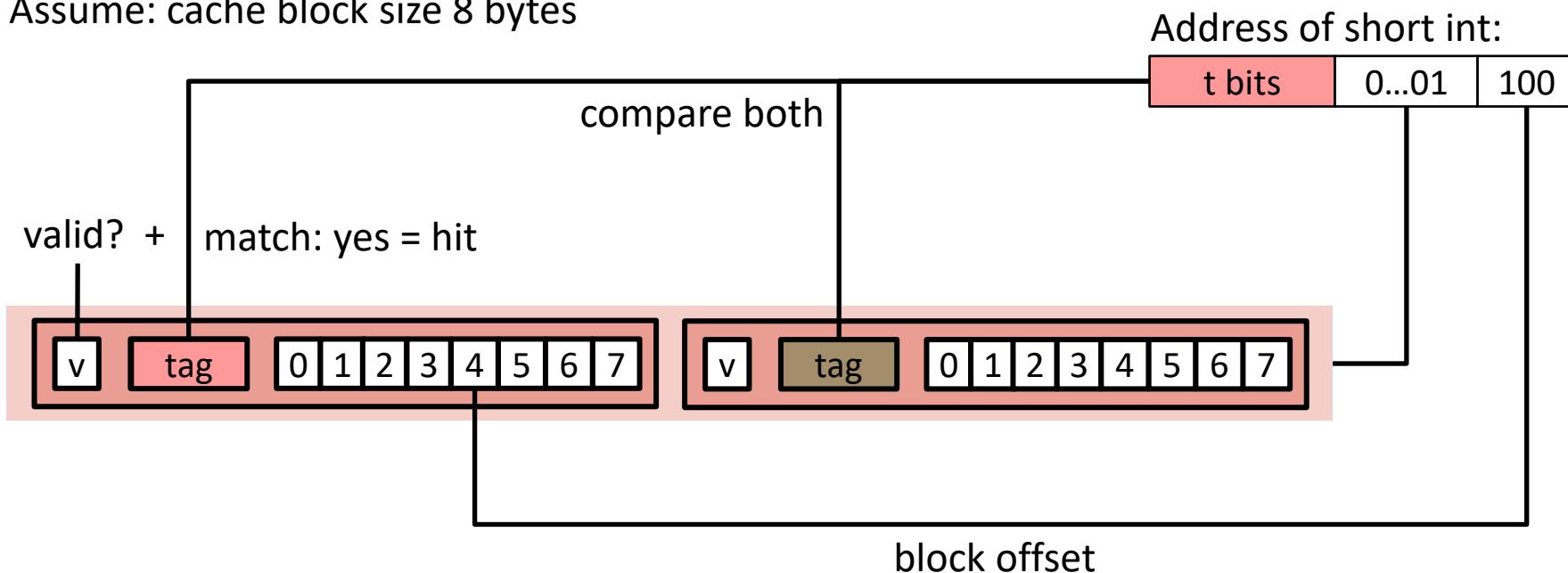
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

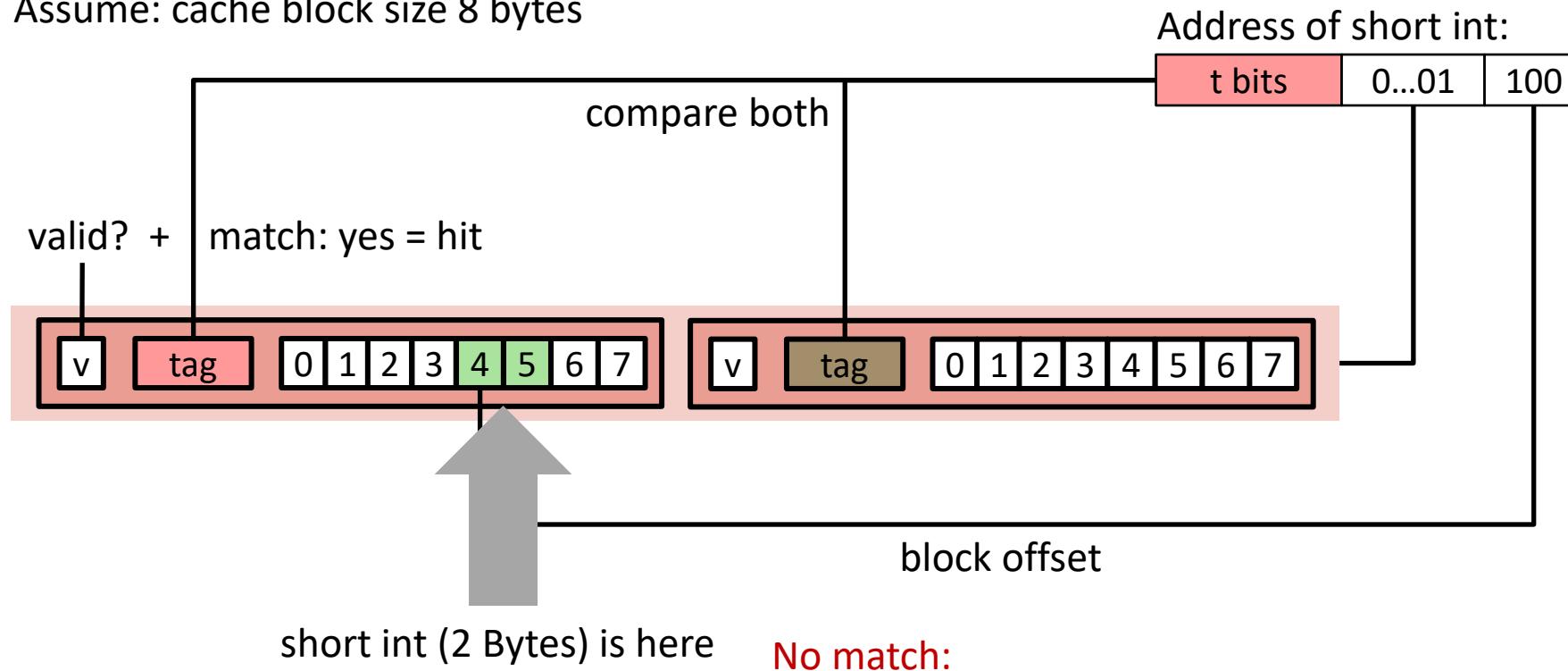
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

2-Way Set Associative Cache Simulation

$t=2$ $s=1$ $b=1$

xx	x	x
----	---	---

$M=16$ byte addresses, $B=2$ bytes/block,
 $S=2$ sets, $E=2$ blocks/set

Address trace (reads, one byte per read):

0	[000 <u>0</u> ₂],	miss
1	[000 <u>1</u> ₂],	hit
7	[01 <u>11</u> ₂],	miss
8	[10 <u>00</u> ₂],	miss
0	[00 <u>00</u> ₂]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

What about writes?

Multiple copies of data exist:

- L1, L2, Main Memory, Disk

What to do on a write-hit?

- **Write-through** (write immediately to memory)
- **Write-back** (defer write to memory until replacement of line)
 - Need a dirty bit (line different from memory or not)

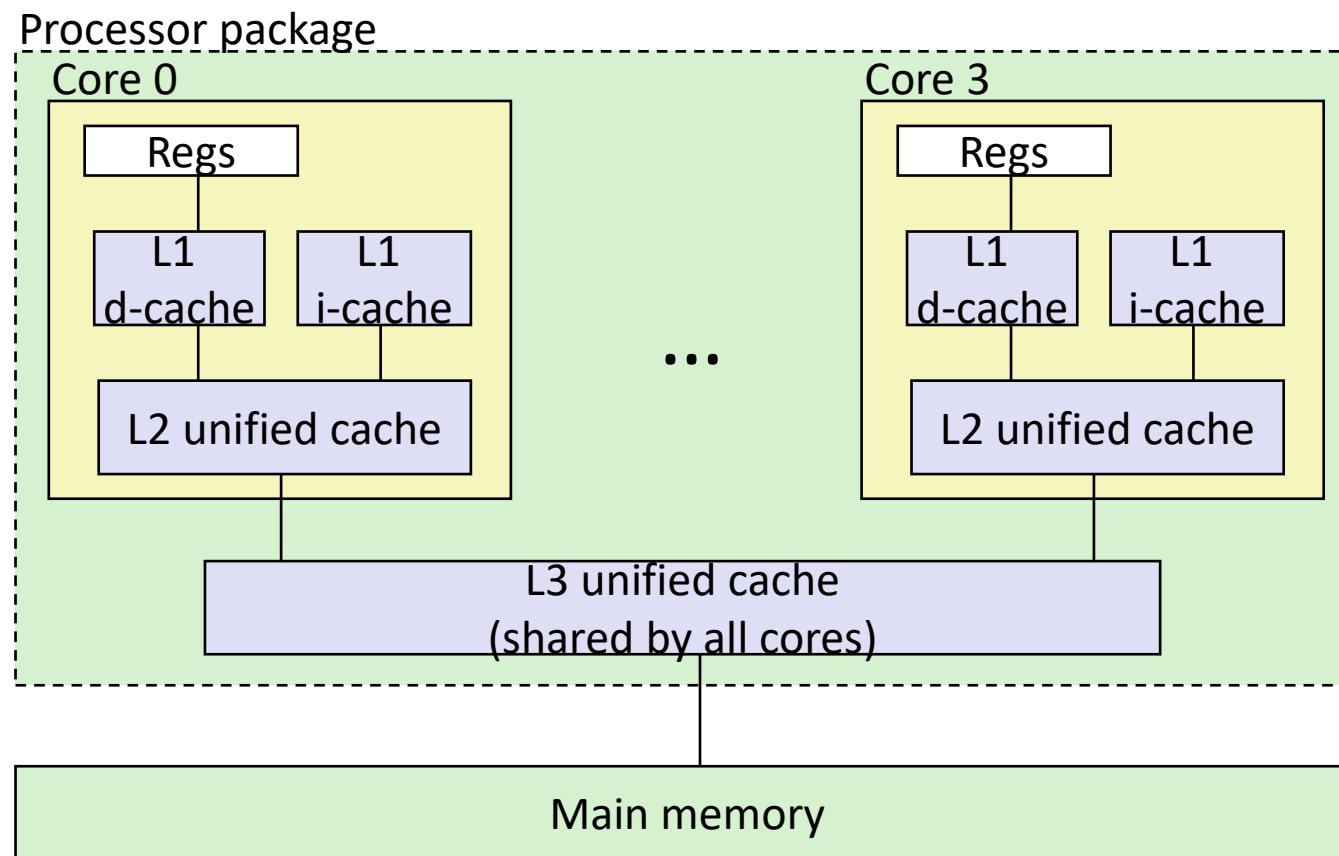
What to do on a write-miss?

- **Write-allocate** (load into cache, update line in cache)
 - Good if more writes to the location follow
- **No-write-allocate** (writes immediately to memory)

Typical

- Write-through + No-write-allocate
- **Write-back + Write-allocate**

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:

32 KB, 8-way,
Access: 4 cycles

L2 unified cache:

256 KB, 8-way,
Access: 10 cycles

L3 unified cache:

8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for
all caches.

Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses / accesses)
= $1 - \text{hit rate}$
- Typical numbers (in percentages):
 - 3-10% for L1
 - can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time

- Time to deliver a line in the cache to the processor
 - includes time to determine whether the line is in the cache
- Typical numbers:
 - 1-2 clock cycle for L1
 - 5-20 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
 - typically 50-200 cycles for main memory (Trend: increasing!)

Lets think about those numbers

- Huge difference between a hit and a miss
 - Could be 100x, if just L1 and main memory

Would you believe 99% hits is twice as good as 97%?

- Consider:
 - cache hit time of 1 cycle
 - miss penalty of 100 cycles
- Average access time:
 - 97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
 - 99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

This is why “miss rate” is used instead of “hit rate”

Writing Cache Friendly Code

Make the common case go fast

- Focus on the inner loops of the core functions

Minimize the misses in the inner loops

- Repeated references to variables are good (**temporal locality**)
- Stride-1 reference patterns are good (**spatial locality**)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.

Topics

Cache organization and operation

Performance impact of caches

- Rearranging loops to improve spatial locality
- Using blocking to improve temporal locality

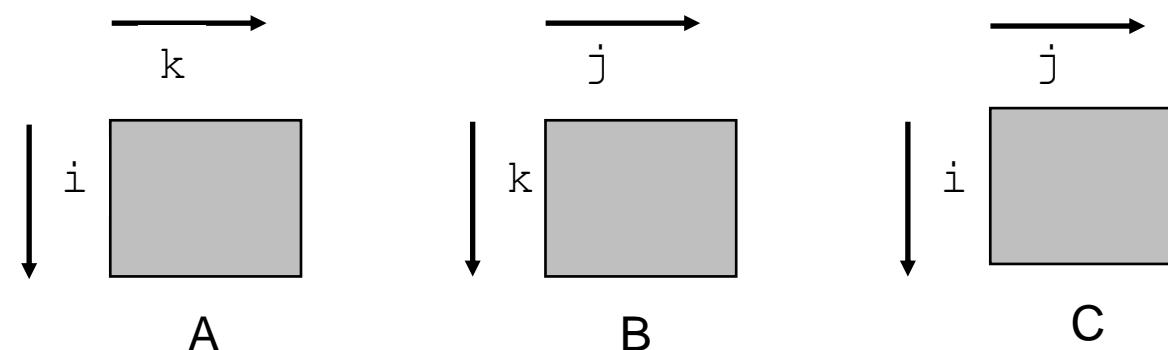
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = $32B$ (big enough for four 64-bit words)
- Matrix dimension (N) is very large
 - Approximate $1/N$ as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:

- Look at access pattern of inner loop



Matrix Multiplication Example

Description:

- Multiply $N \times N$ matrices
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination
 - but may be able to hold in register

```
/* ijk */  
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0; ←  
        for (k=0; k<n; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

Variable sum held in register

Layout of C Arrays in Memory (review)

C arrays allocated in row-major order

- each row in contiguous memory locations

Stepping through columns in one row:

- `for (i = 0; i < N; i++)
 sum += a[0][i];`
- accesses successive elements
- if block size (B) > 4 bytes, exploit spatial locality
 - compulsory miss rate = 4 bytes / B

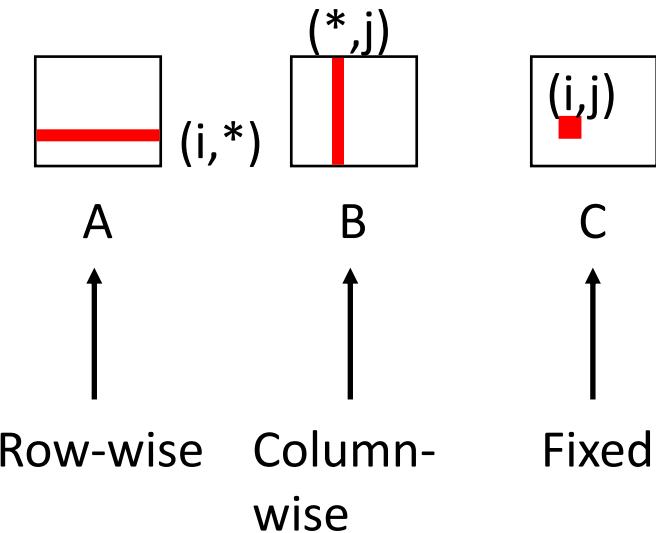
Stepping through rows in one column:

- `for (i = 0; i < n; i++)
 sum += a[i][0];`
- accesses distant elements
- no spatial locality!
 - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Inner loop:



Misses per inner loop iteration:

A
0.25

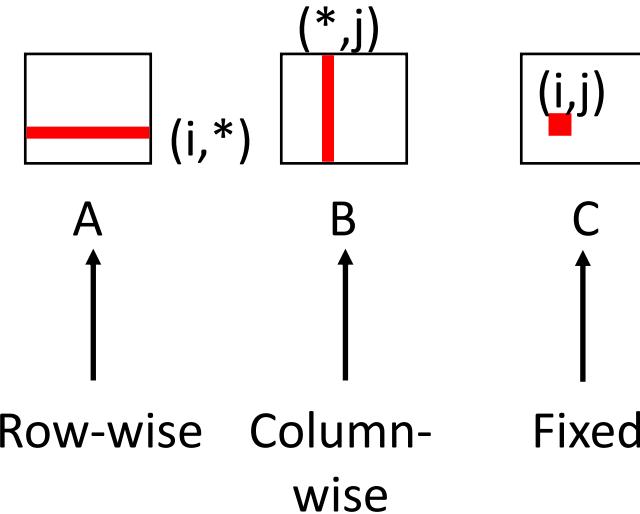
B
1.0

C
0.0

Matrix Multiplication (jik)

```
/* jik */  
for (j=0; j<n; j++) {  
    for (i=0; i<n; i++) {  
        sum = 0.0;  
        for (k=0; k<n; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum  
    }  
}
```

Inner loop:



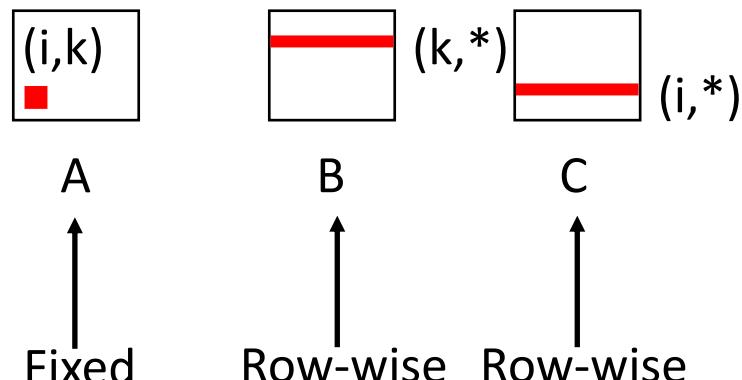
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */  
for (k=0; k<n; k++) {  
    for (i=0; i<n; i++) {  
        r = a[i][k];  
        for (j=0; j<n; j++)  
            c[i][j] += r * b[k][j];  
    }  
}
```

Inner loop:



Misses per inner loop iteration:

A
0.0

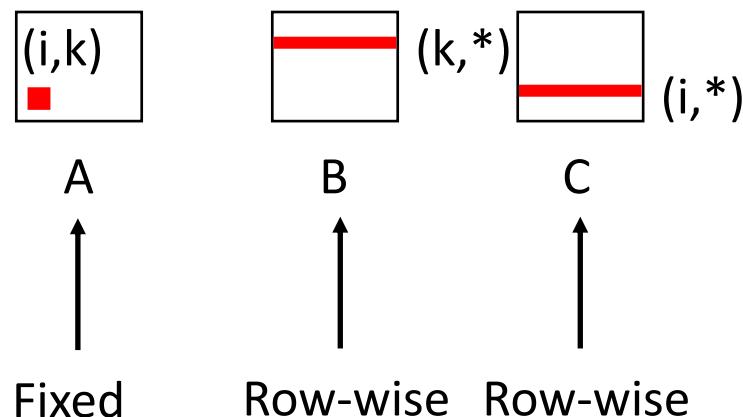
B
0.25

C
0.25

Matrix Multiplication (ikj)

```
/* ikj */  
for (i=0; i<n; i++) {  
    for (k=0; k<n; k++) {  
        r = a[i][k];  
        for (j=0; j<n; j++)  
            c[i][j] += r * b[k][j];  
    }  
}
```

Inner loop:



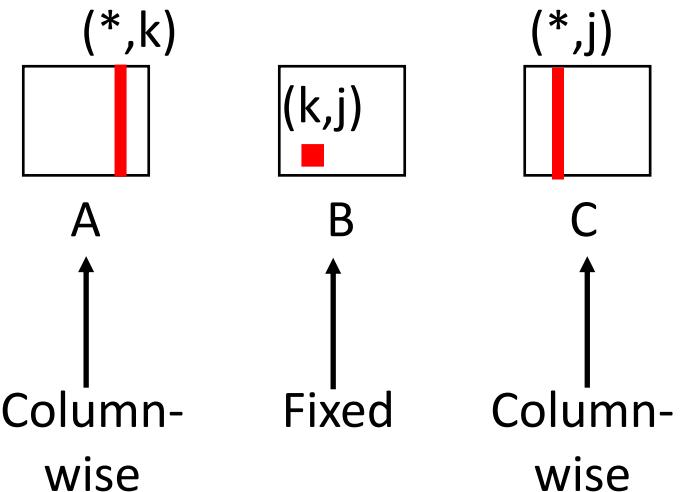
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Inner loop:



Misses per inner loop iteration:

A
1.0

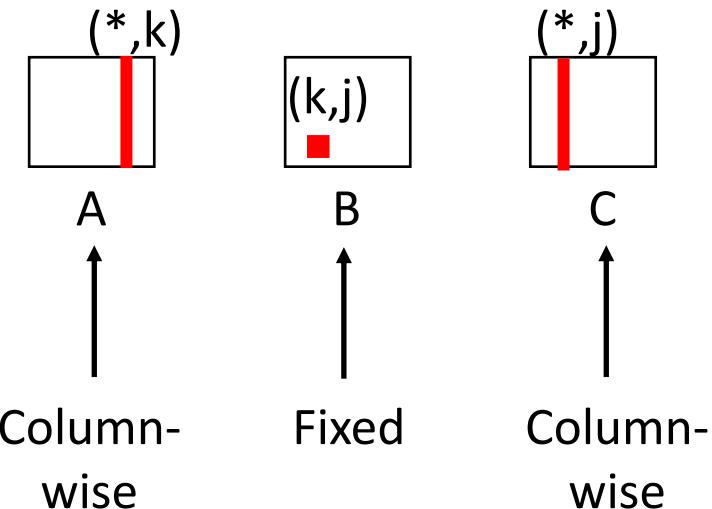
B
0.0

C
1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0;  
        for (k=0; k<n; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

```
for (j=0; j<n; j++) {  
    for (k=0; k<n; k++) {  
        r = b[k][j];  
        for (i=0; i<n; i++)  
            c[i][j] += a[i][k] * r;  
    }  
}
```

jki (& kji):

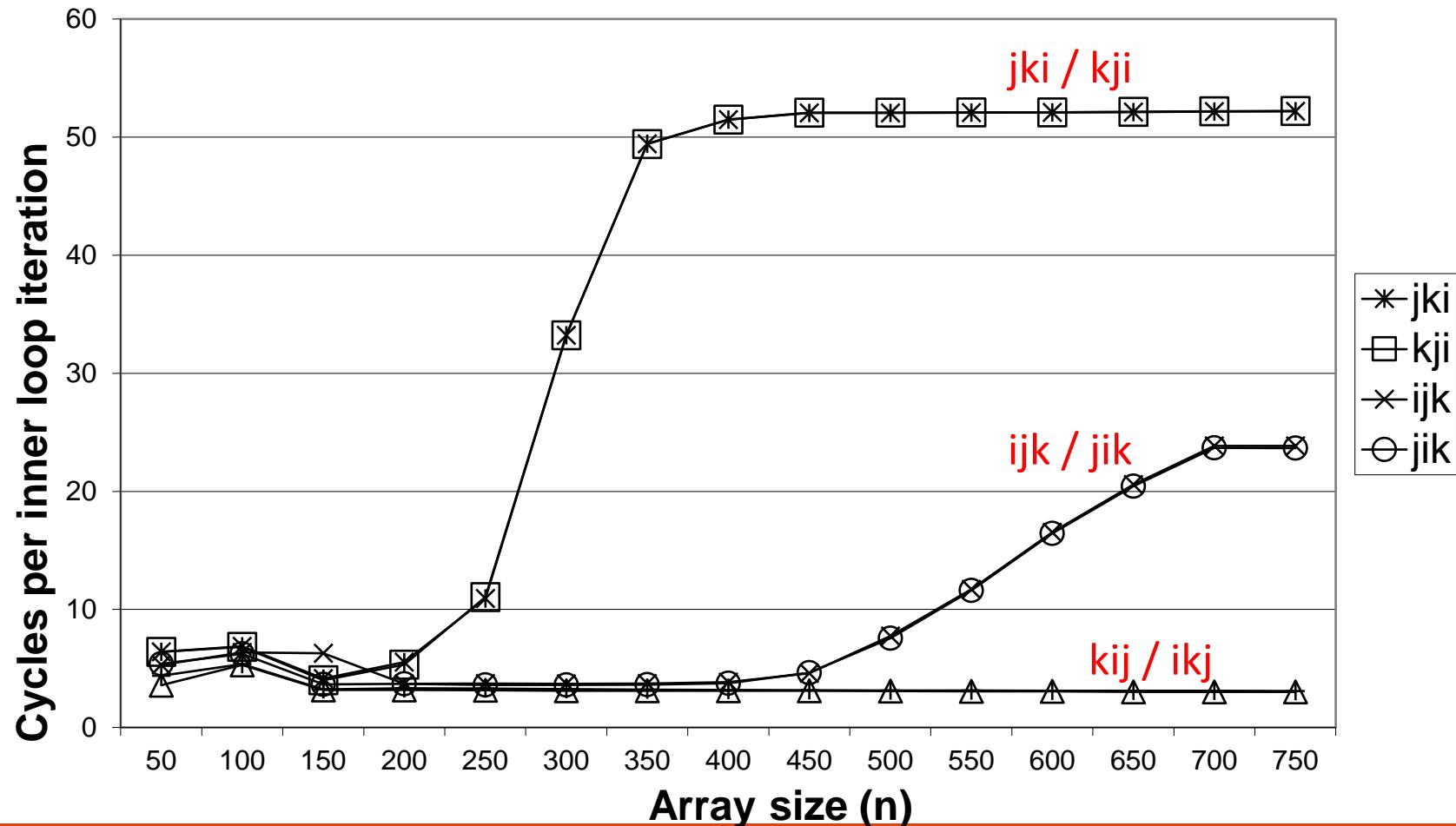
- 2 loads, 1 store
- misses/iter = 2.0

```
for (k=0; k<n; k++) {  
    for (i=0; i<n; i++) {  
        r = a[i][k];  
        for (j=0; j<n; j++)  
            c[i][j] += r * b[k][j];  
    }  
}
```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

Core i7 Matrix Multiply Performance



Topics

Cache organization and operation

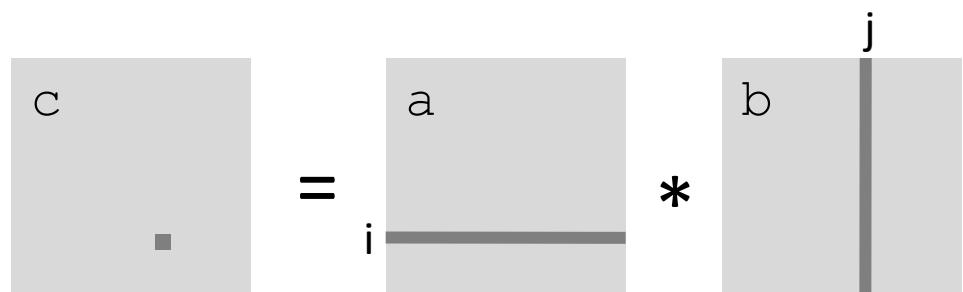
Performance impact of caches

- Rearranging loops to improve spatial locality
- Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```



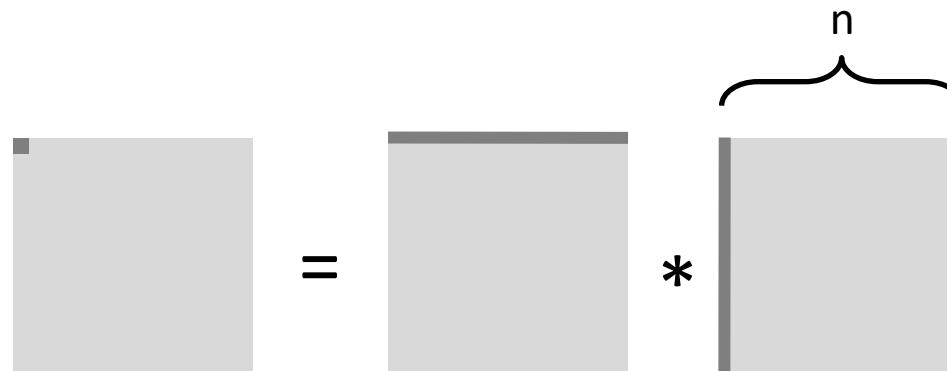
Cache Miss Analysis

Assume:

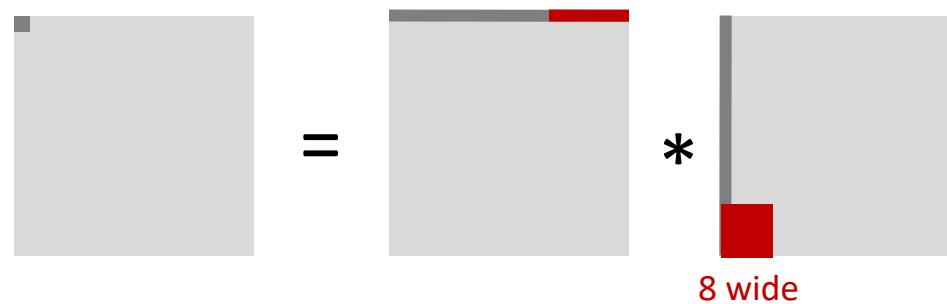
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)

First iteration:

- $n/8 + n = 9n/8$ misses



- Afterwards **in cache**:
(schematic)



Cache Miss Analysis

Assume:

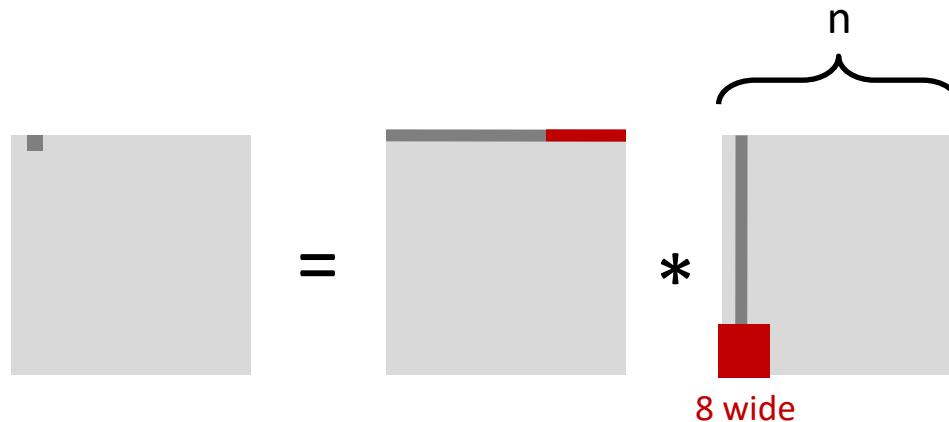
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)

Second iteration:

- Again:
 $n/8 + n = 9n/8$ misses

Total misses:

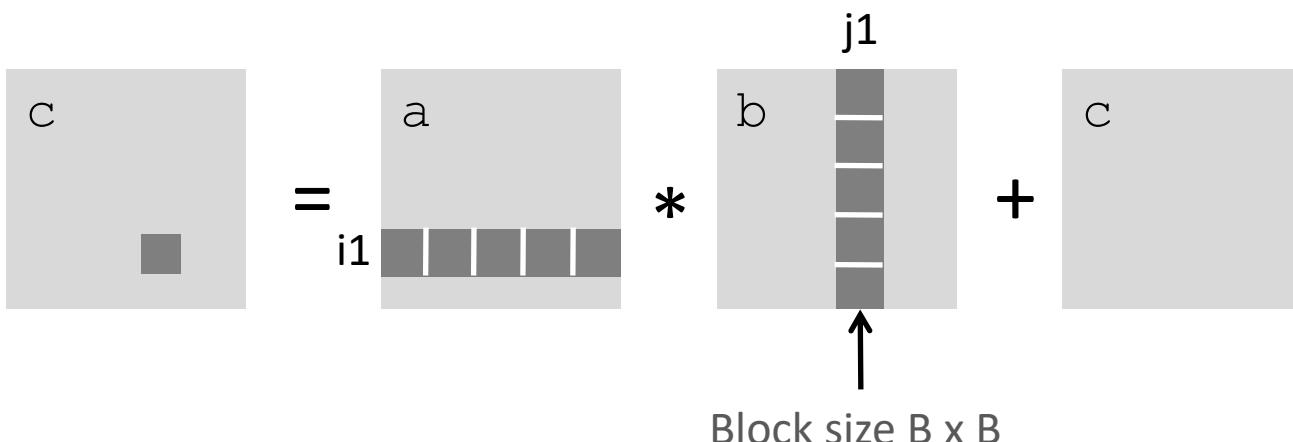
- $9n/8 * n^2 = (9/8) * n^3$



Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Cache Miss Analysis

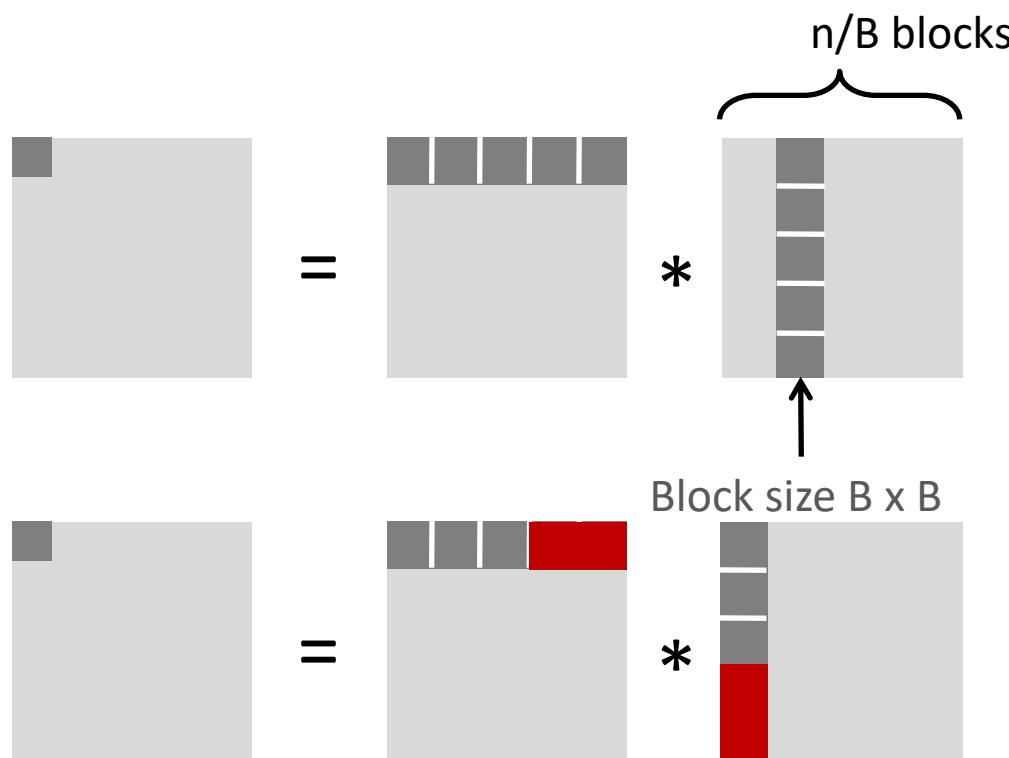
Assume:

- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)
- Three blocks fit into cache: $3B^2 < C$

First (block) iteration:

- $B^2/8$ misses for each block
- $2n/B * B^2/8 = nB/4$
(omitting matrix c)

- Afterwards in cache
(schematic)



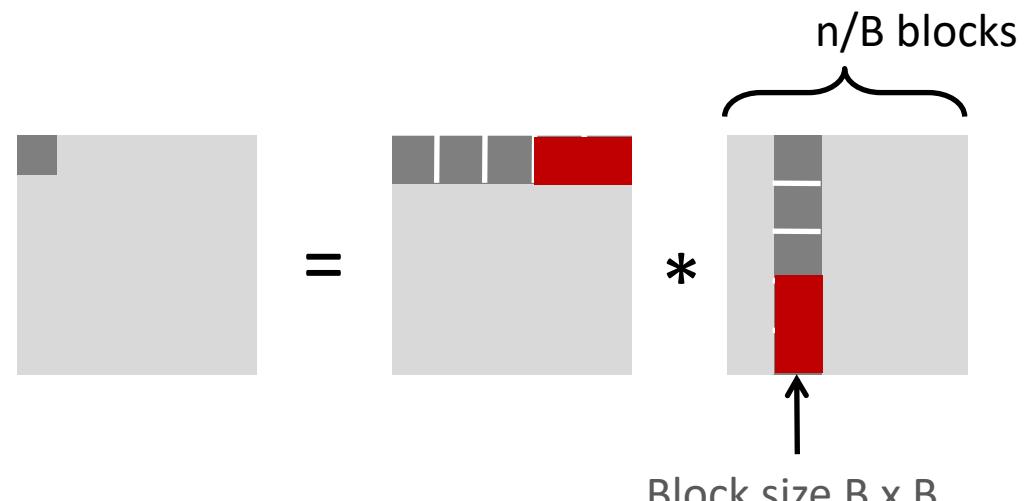
Cache Miss Analysis

Assume:

- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)
- Three blocks fit into cache: $3B^2 < C$

Second (block) iteration:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$



Total misses:

- $nB/4 * (n/B)^2 = n^3/(4B)$

Summary

No blocking: $(9/8) * n^3$

Blocking: $1/(4B) * n^3$

Suggest largest possible block size B, but limit $3B^2 < C$!

Reason for dramatic difference:

- Matrix multiplication has inherent temporal locality:
 - Input data: $3n^2$, computation $2n^3$
 - Every array elements used $O(n)$ times!
- But program has to be written properly

Concluding Observations

Programmer can optimize for cache performance

- How data structures are organized
- How data are accessed
 - Nested loop structure
 - Blocking is a general technique

All systems favor “cache friendly code”

- Getting absolute optimum performance is very platform specific
 - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
 - Keep working set reasonably small (temporal locality)
 - Use small strides (spatial locality)

Thank You!

Computer Systems Organization (CS2.201)

VIRTUAL MEMORY (SECTION 9.1-9.3 AND 9.6)

Deepak Gangadharan
Computer Systems Group (CSG), IIIT Hyderabad

Slide Contents: Adapted from slides by Randal Bryant

Processes

Definition: A *process* is an instance of a running program.

- One of the most profound ideas in computer science
- Not the same as “program” or “processor”

Process provides each program with two key abstractions:

- Logical control flow
 - Each program seems to have exclusive use of the CPU
- Private virtual address space
 - Each program seems to have exclusive use of main memory

How are these Illusions maintained?

- Process executions interleaved (multitasking) or run on separate cores
- Address spaces managed by virtual memory system

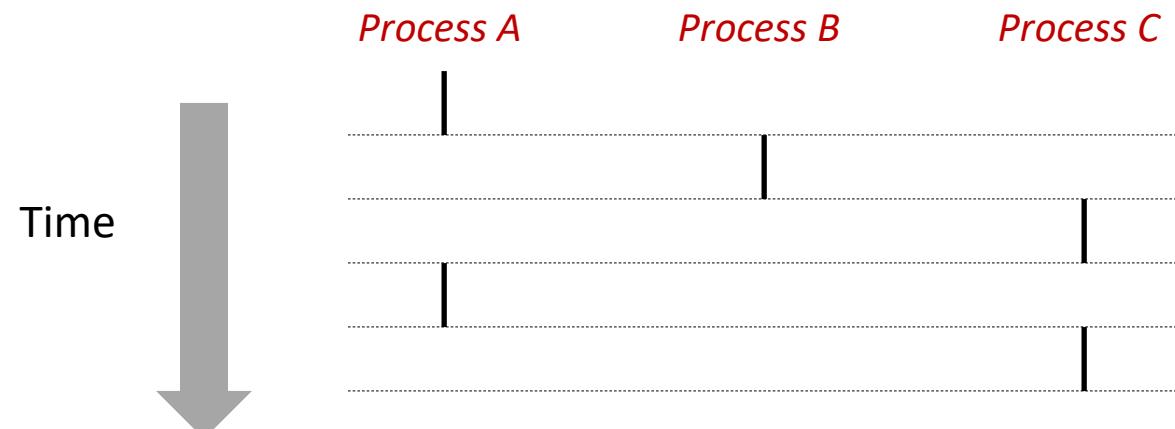
Concurrent Processes

Two processes *run concurrently* (are concurrent) if their flows overlap in time

Otherwise, they are *sequential*

Examples (running on single core):

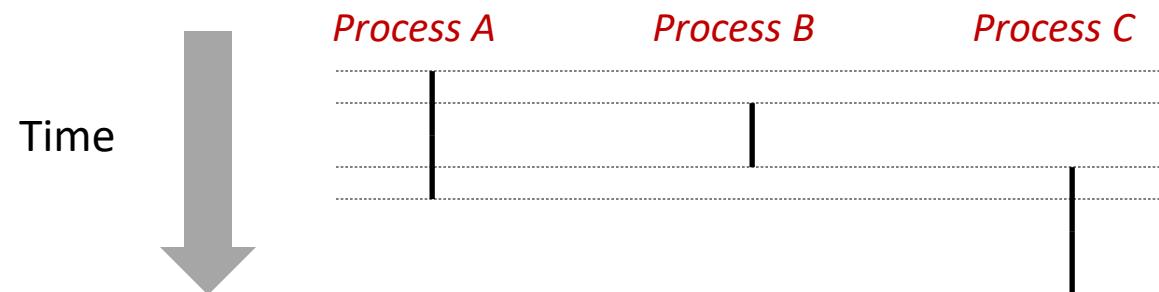
- Concurrent: A & B, A & C
- Sequential: B & C



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



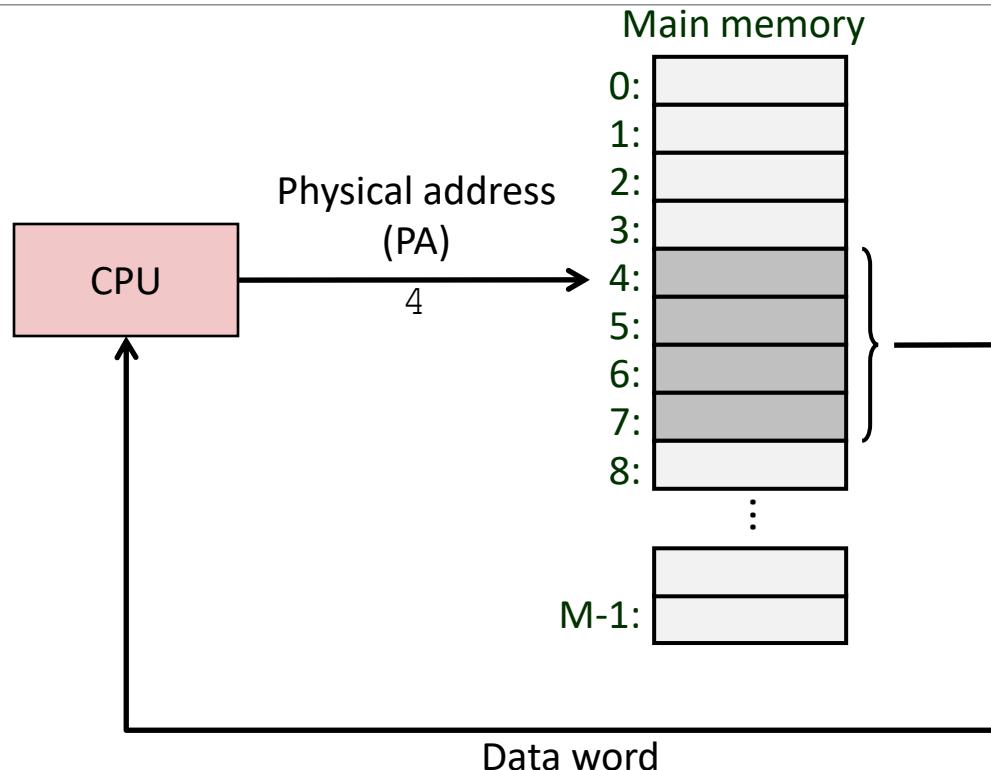
Topics

Address spaces

VM as a tool for caching

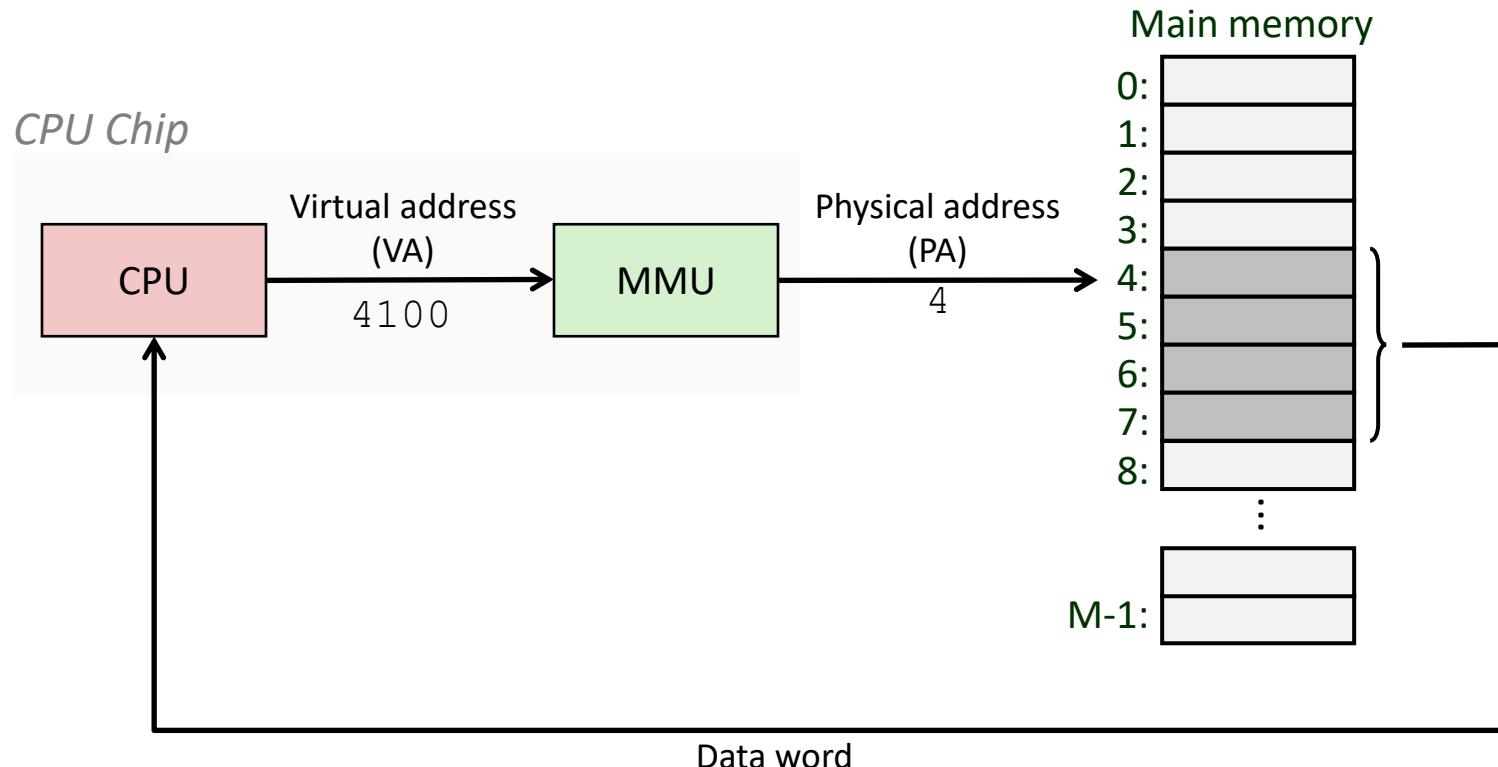
Address translation

A System Using Physical Addressing



Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



Used in all modern servers, desktops, and laptops

One of the great ideas in computer science

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

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

Topics

Address spaces

VM as a tool for caching

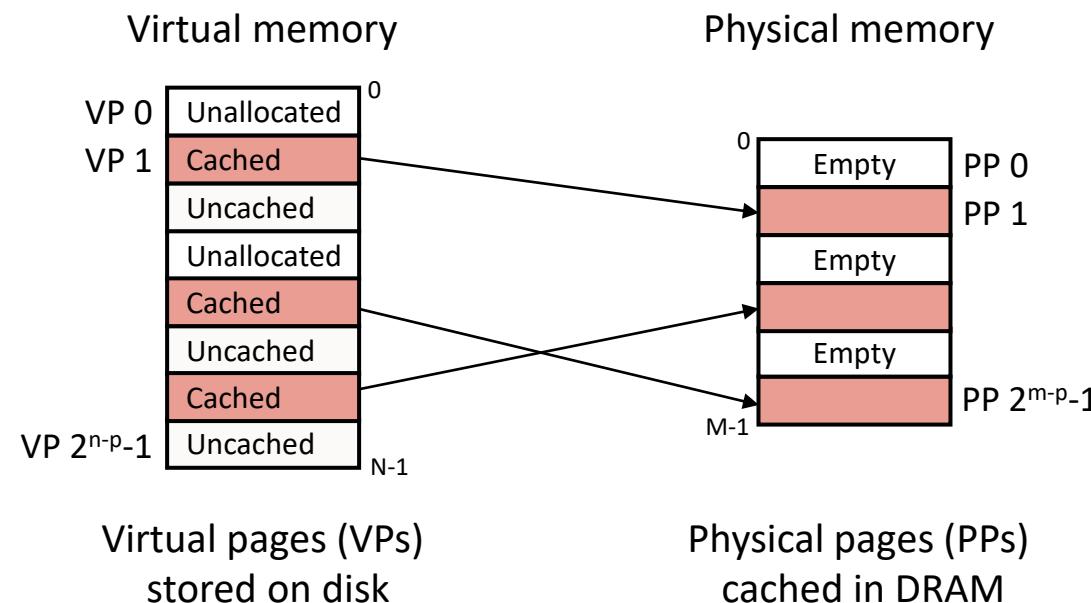
Address translation

VM as a Tool for Caching

Virtual memory is an array of N contiguous bytes stored on disk.

The contents of the array on disk are cached in *physical memory (DRAM cache)*

- These cache blocks are called *pages* (size is $P = 2^p$ bytes)



DRAM Cache Organization

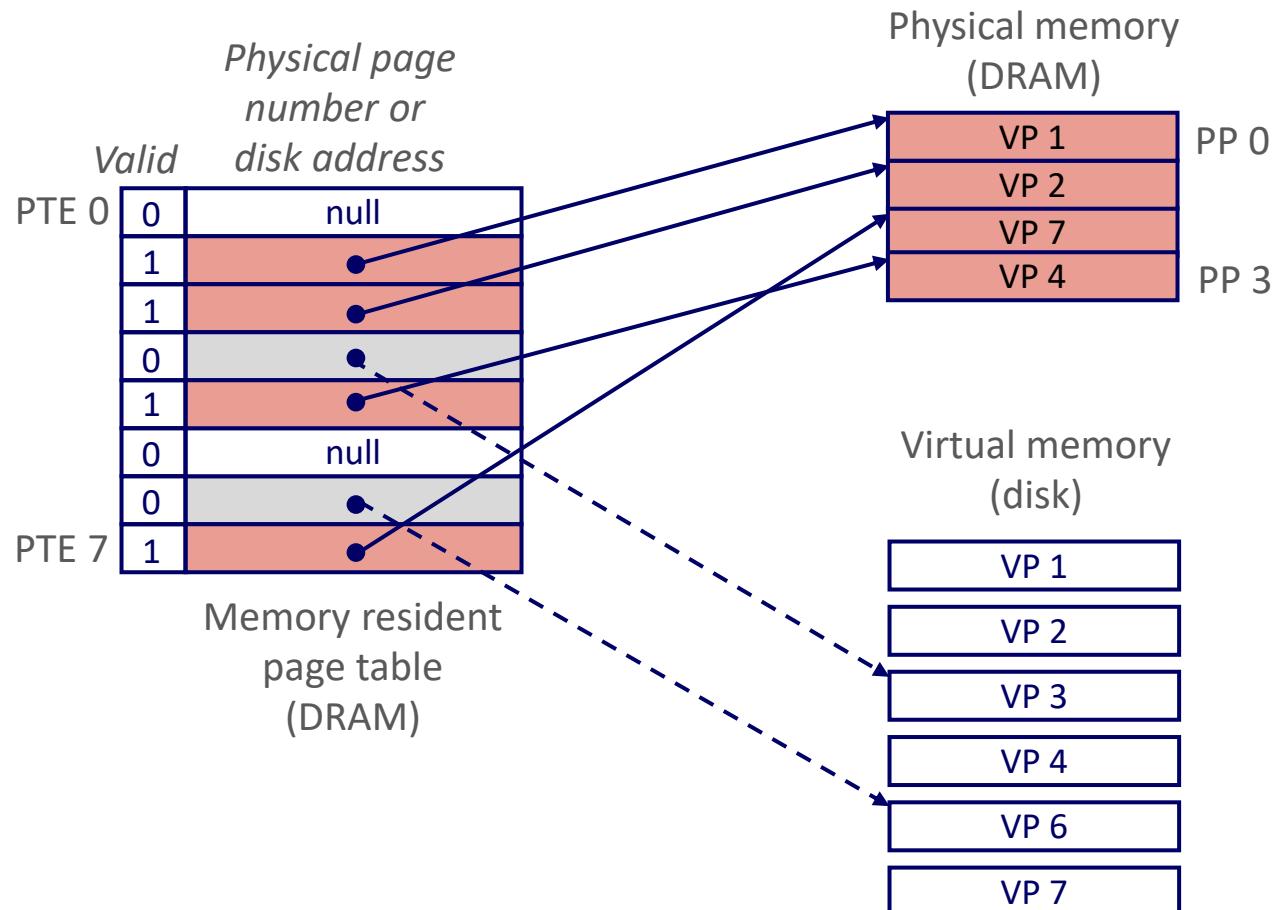
DRAM cache organization driven by the enormous miss penalty

- DRAM is about **10x** slower than SRAM
- Disk is about **10,000x** slower than DRAM

Consequences

- Large page (block) size: typically 4-8 KB, sometimes 4 MB
- Fully associative
 - Any VP can be placed in any PP
 - Requires a “large” mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms
 - Too complicated and open-ended to be implemented in hardware
- Write-back rather than write-through

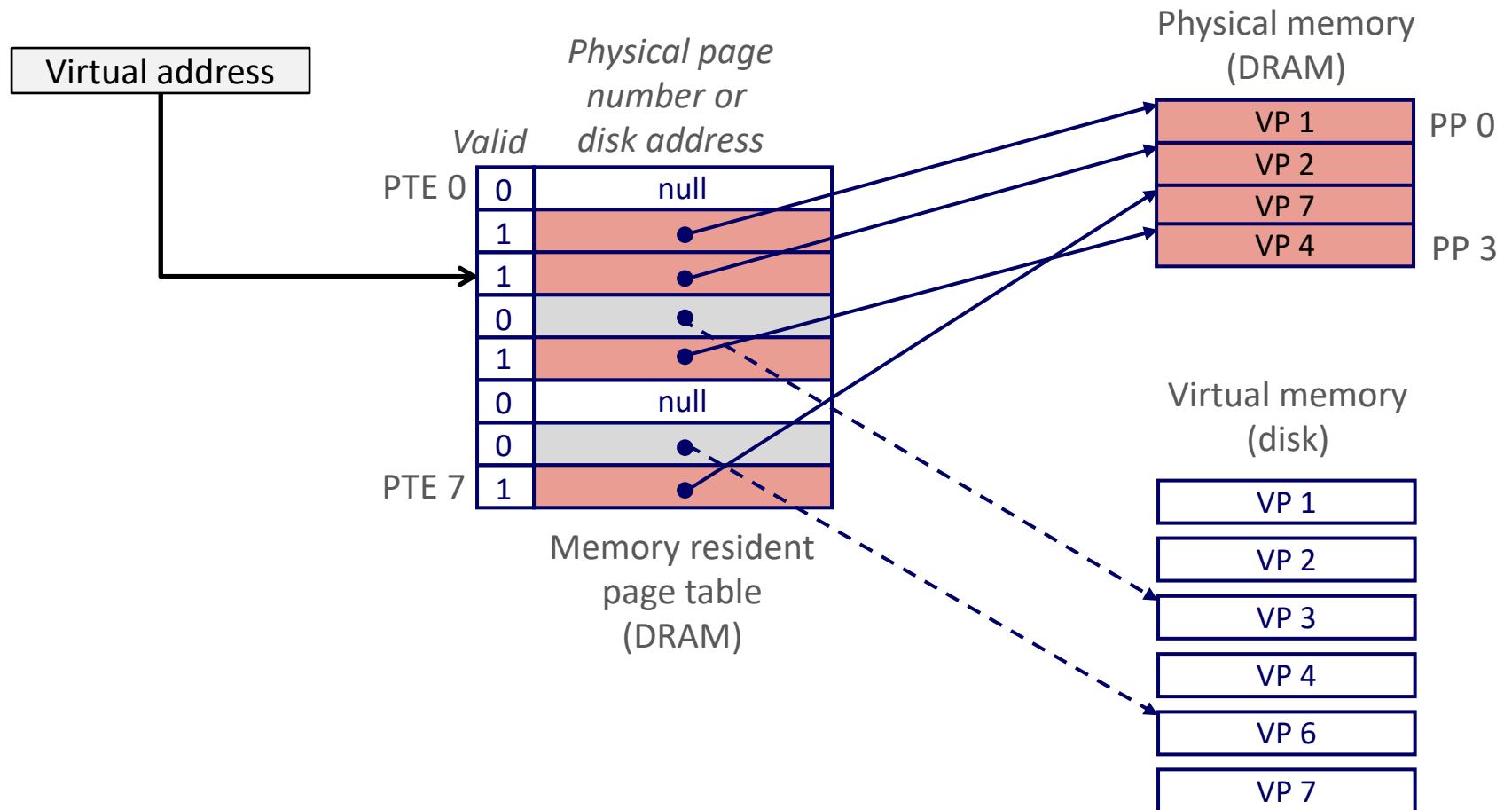
Page Tables



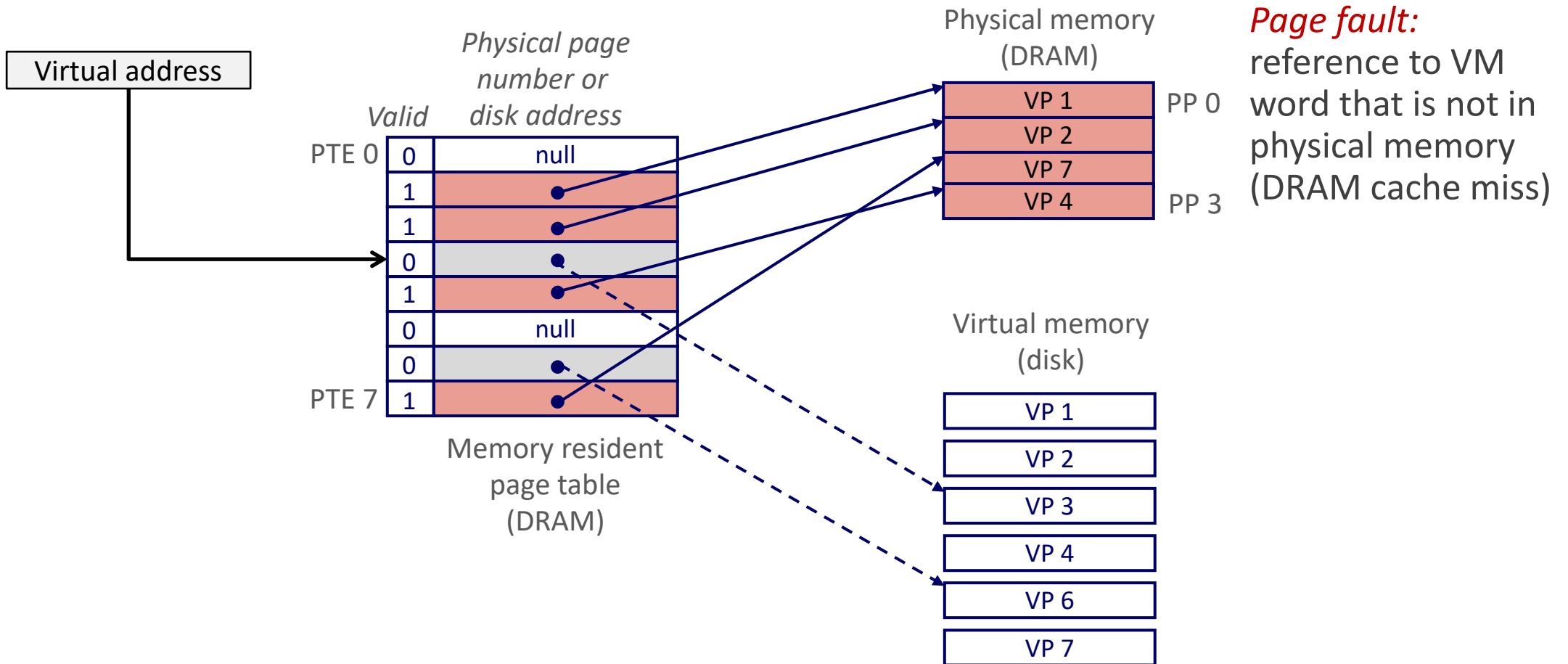
A **page table** is an array of page table entries (PTEs) that maps virtual pages to physical pages.

- Per-process kernel data structure in DRAM

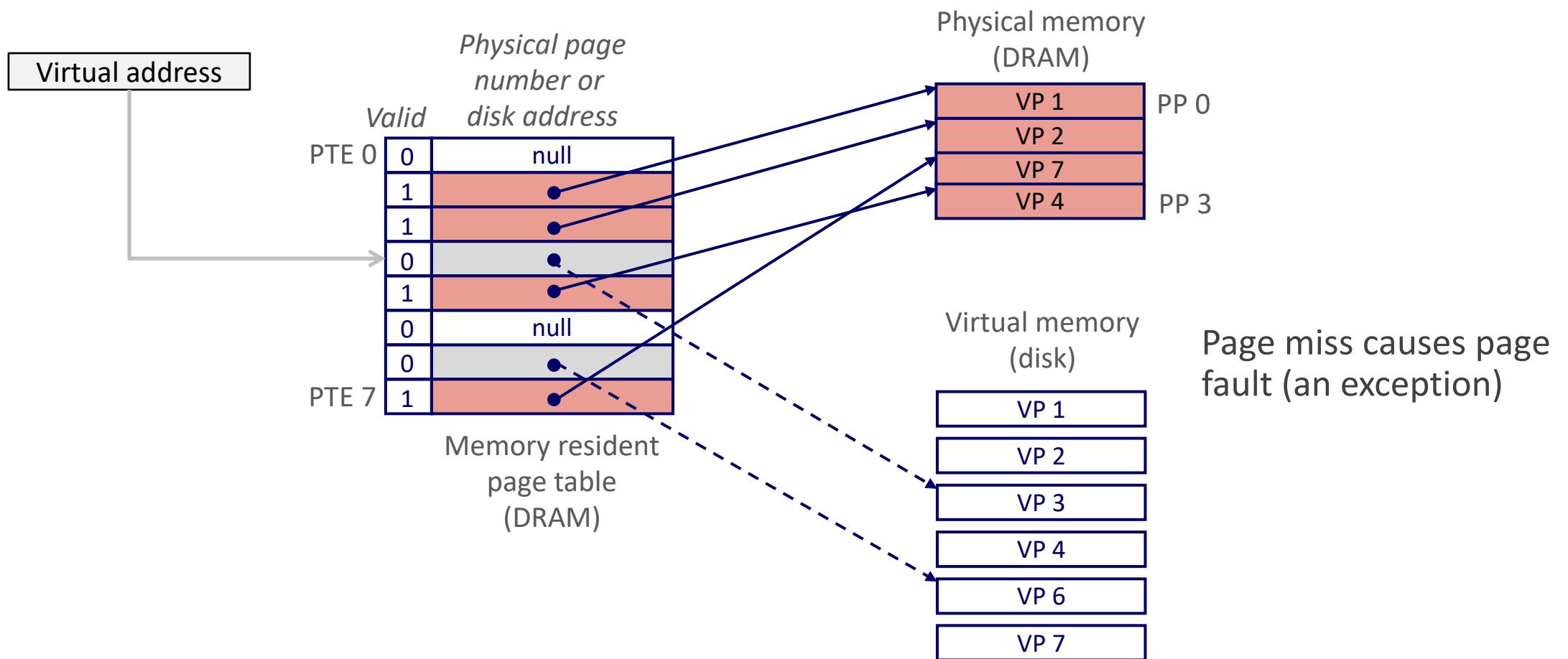
Page Hit



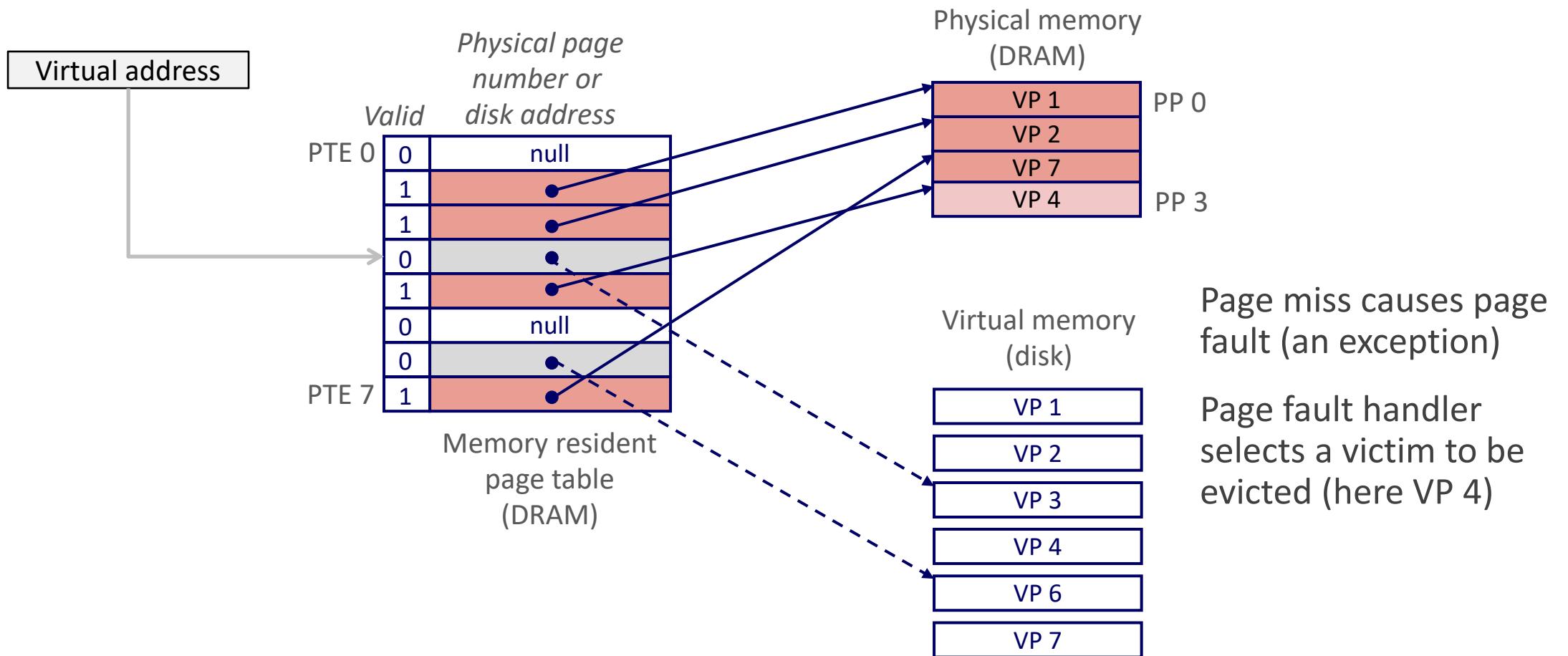
Page Fault



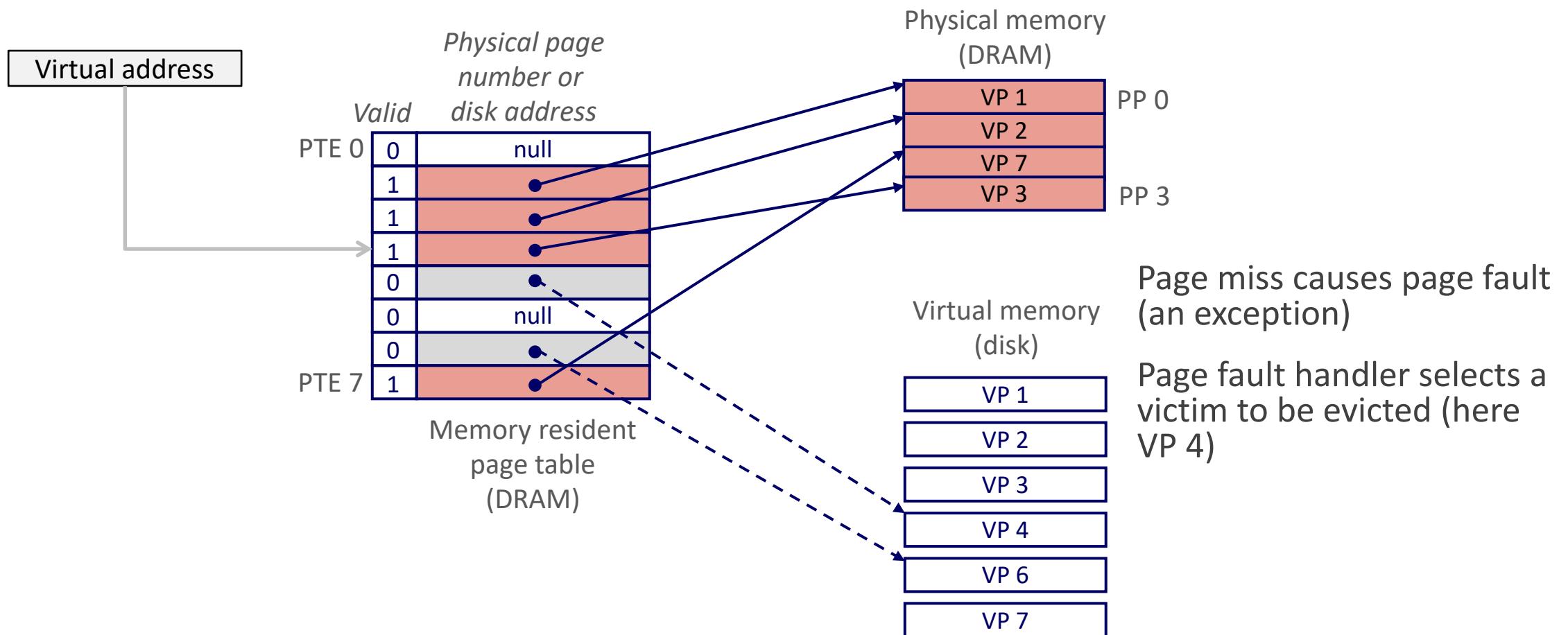
Handling Page Fault



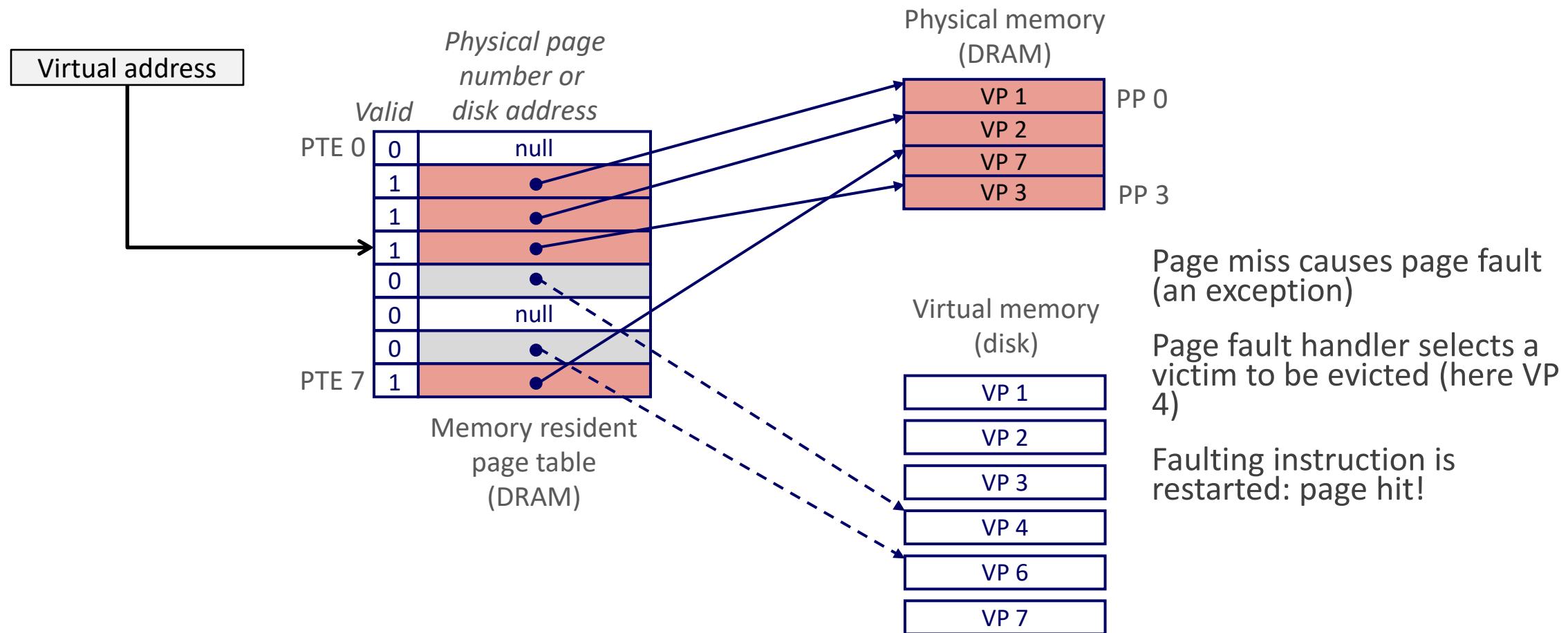
Handling Page Fault



Handling Page Fault



Handling Page Fault



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

If (working set size < main memory size)

- Good performance for one process after compulsory misses

If ($\text{SUM}(\text{working set sizes}) > \text{main memory size}$)

- *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

Topics

Address spaces

VM as a tool for caching

Address translation

VM Address Translation

Virtual Address Space

- $V = \{0, 1, \dots, N-1\}$

Physical Address Space

- $P = \{0, 1, \dots, M-1\}$

Address Translation

- **MAP:** $V \rightarrow P \cup \{\emptyset\}$
- For virtual address a :
 - $MAP(a) = a'$ if data at virtual address a is at physical address a' in P
 - $MAP(a) = \emptyset$ if data at virtual address a is not in physical memory
 - Either invalid or stored on disk

Summary of Address Translation Symbols

Basic Parameters

- **N = 2^n** : Number of addresses in virtual address space
- **M = 2^m** : Number of addresses in physical address space
- **P = 2^p** : Page size (bytes)

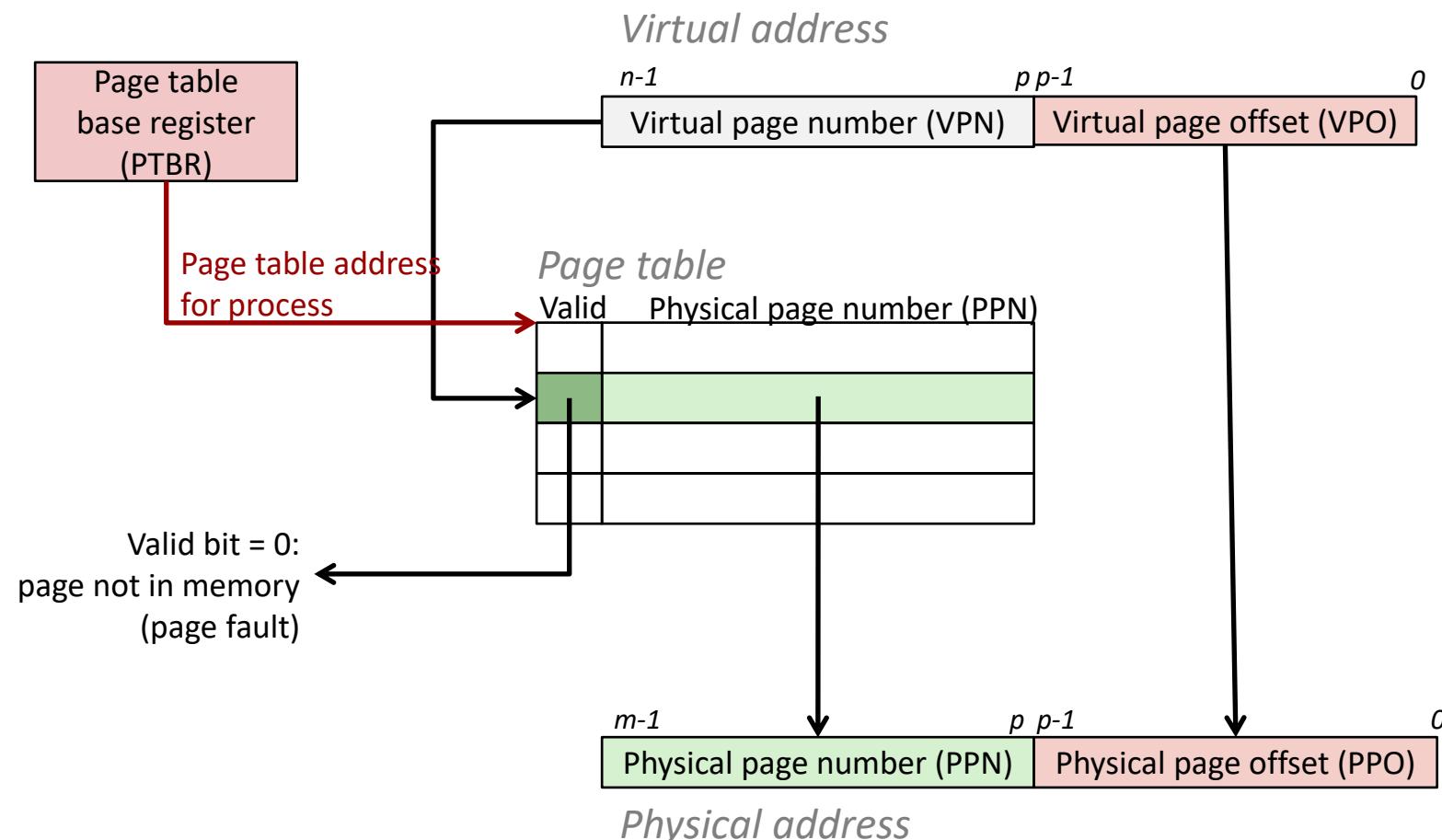
Components of the virtual address (VA)

- **TLBI**: TLB index
- **TLBT**: TLB tag
- **VPO**: Virtual page offset
- **VPN**: Virtual page number

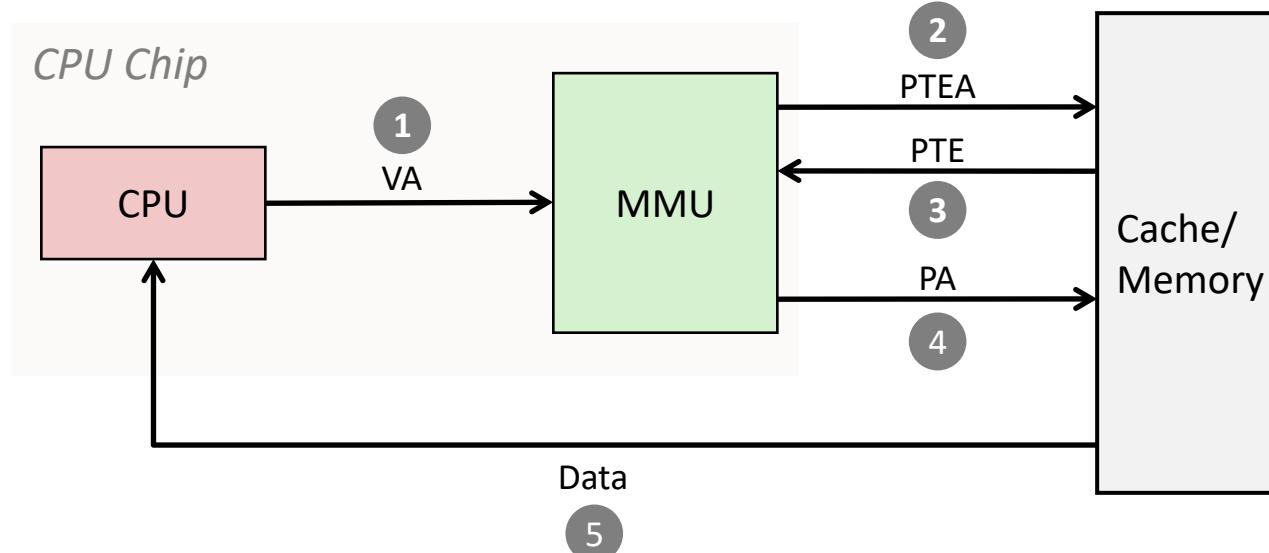
Components of the physical address (PA)

- **PPO**: Physical page offset (same as VPO)
- **PPN**: Physical page number
- **CO**: Byte offset within cache line
- **CI**: Cache index
- **CT**: Cache tag

Address Translation With a Page Table

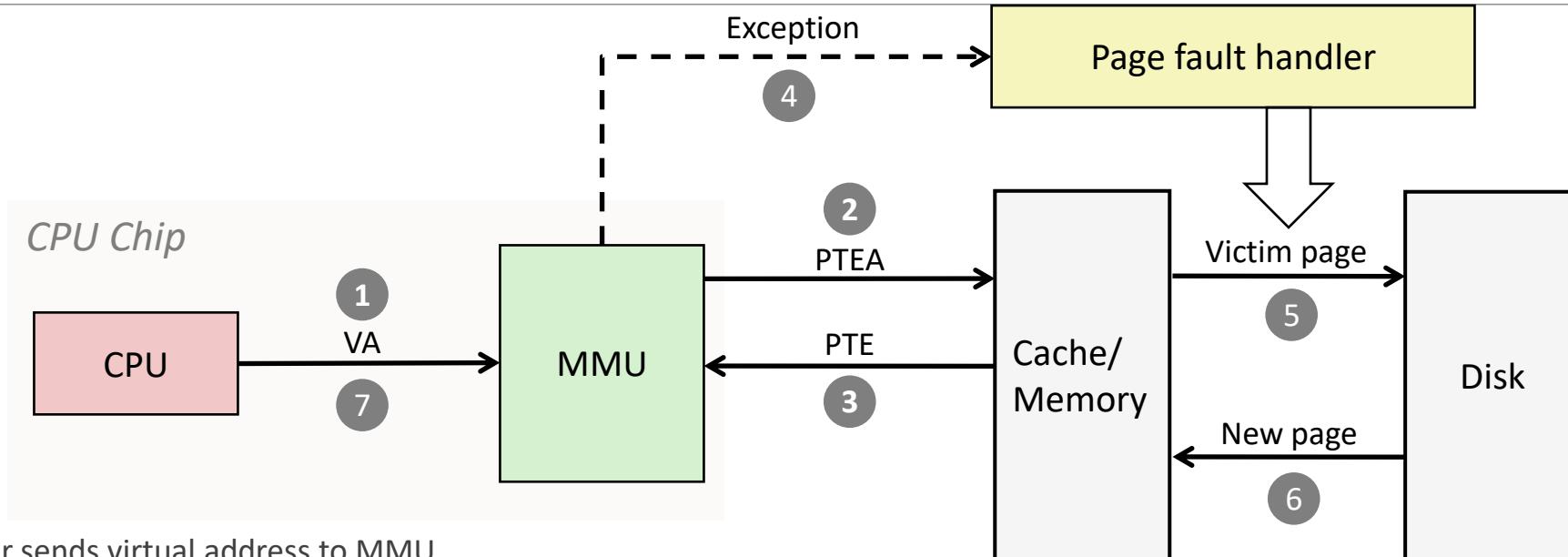


Address Translation: Page Hit



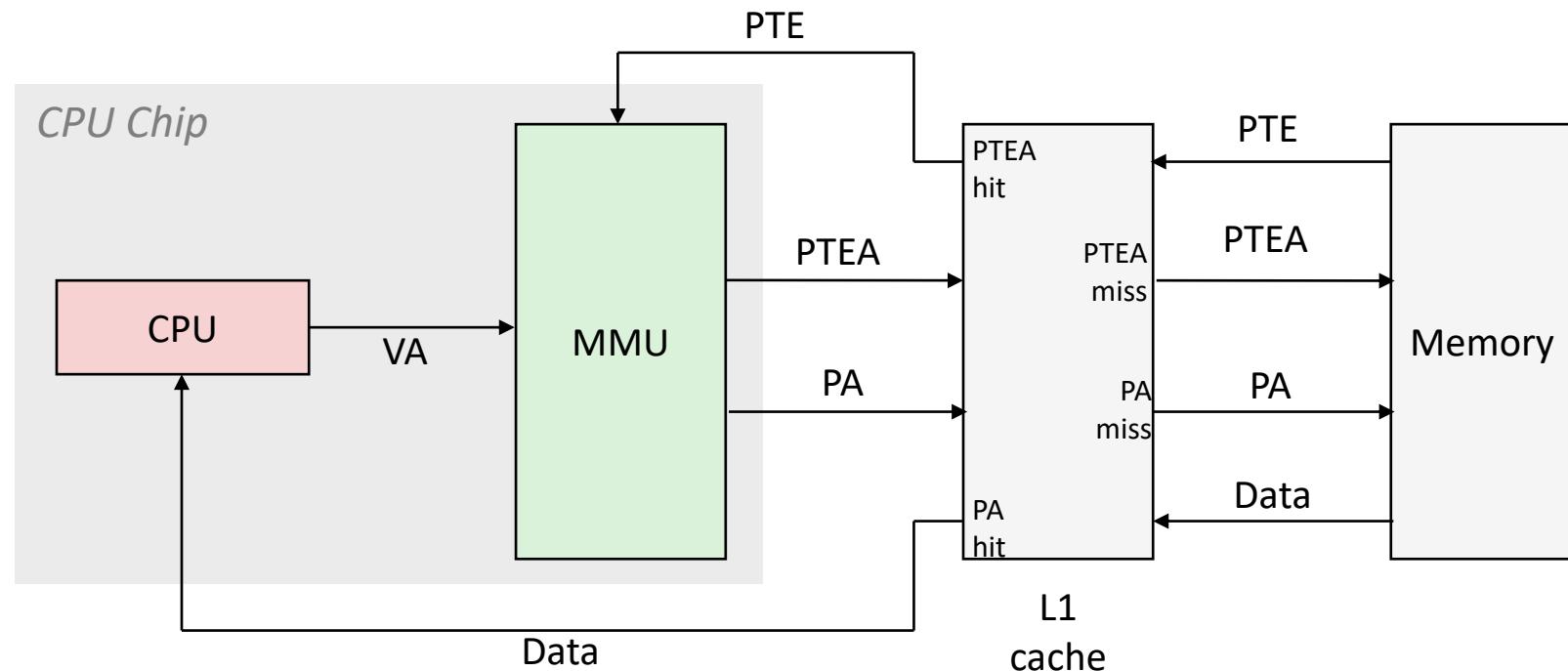
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

Address Translation: Page Fault



- 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 VM and Cache



VA: *virtual address*, PA: *physical address*, PTE: *page table entry*, PTEA = PTE address

Speeding up Translation with a TLB

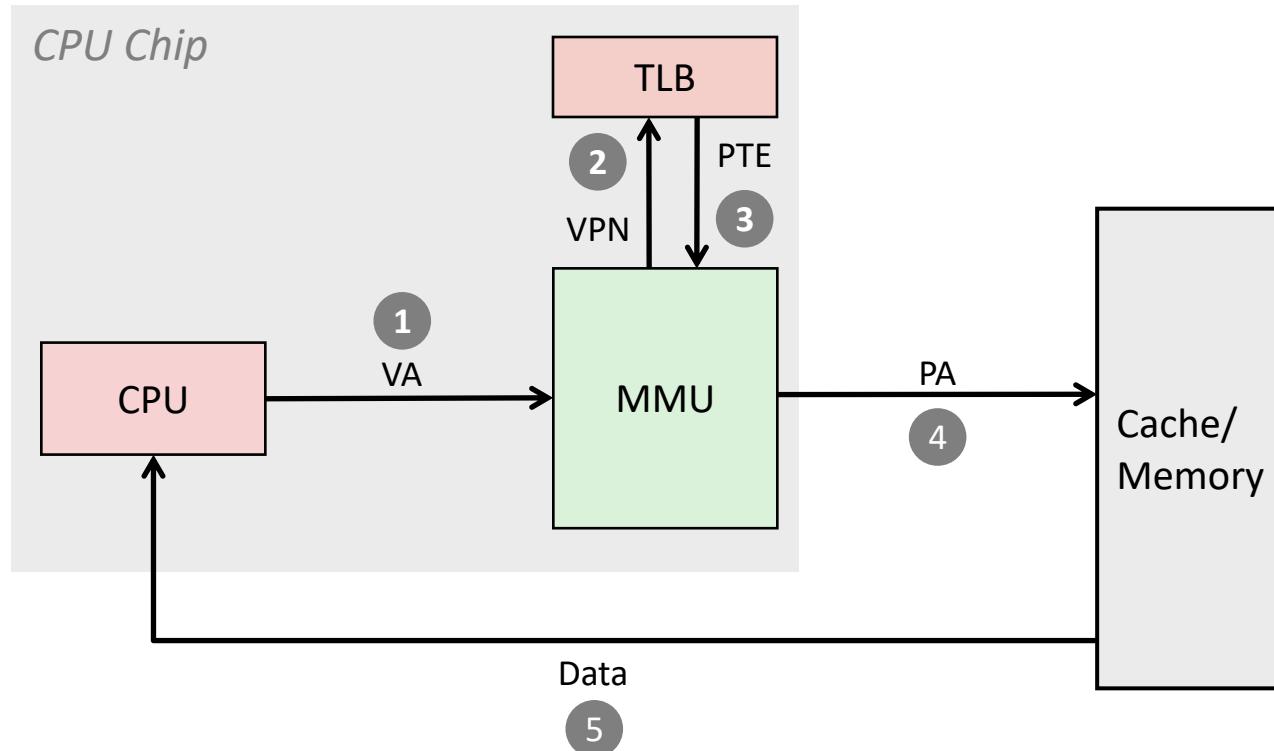
Page table entries (PTEs) are cached in L1 like any other memory word

- PTEs may be evicted by other data references
- PTE hit still requires a small L1 delay

Solution: *Translation Lookaside Buffer* (TLB)

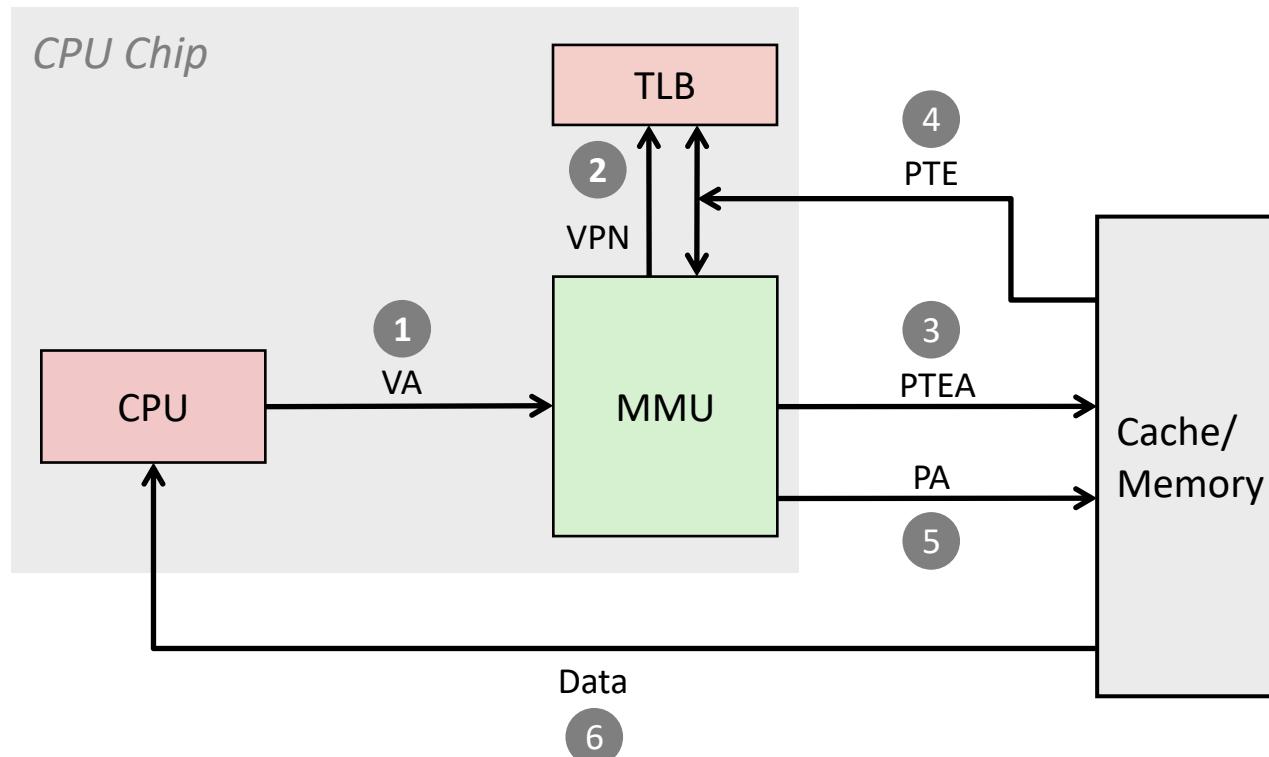
- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

TLB Hit



A TLB hit eliminates a memory access

TLB Miss



A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?

Multi-Level Page Tables

Suppose:

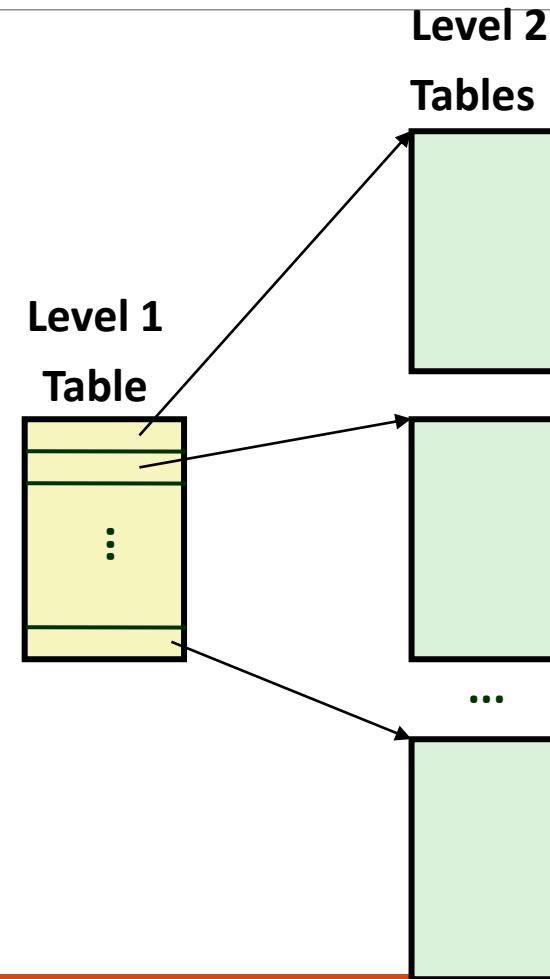
- 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE

Problem:

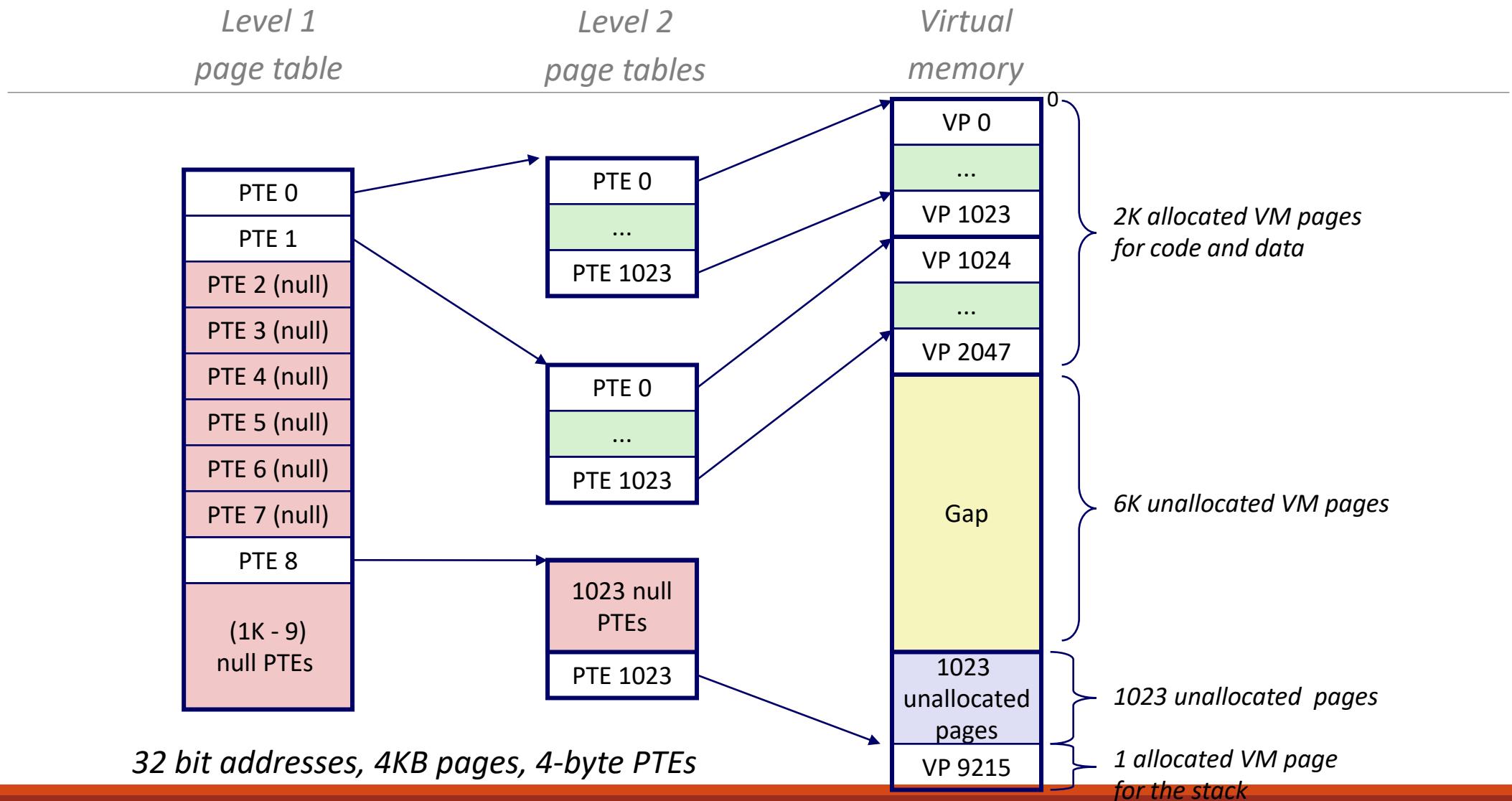
- Would need a 512 GB page table!
 - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes

Common solution:

- Multi-level page tables
- Example: 2-level page table
 - Level 1 table: each PTE points to a page table (always memory resident)
 - Level 2 table: each PTE points to a page (paged in and out like any other data)



A Two-Level Page Table Hierarchy



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

Thank You!