CS330 - Computer Organization and Assembly Language Programming

Lecture 23

Review

Professor : Mahmut Unan – UAB CS

Final Exam

- CS330-2F

Tuesday, April 26

4:45 PM – 6:45 PM

Cumulative Exam

Hello World !

A typical *C program* basically consists of the following parts;

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comment

```
#include <stdio.h>
int main()
{
 /* the first program in CS330 */
 printf("hello, world\n");
 return 0;
}
```

Information is Bits + Context

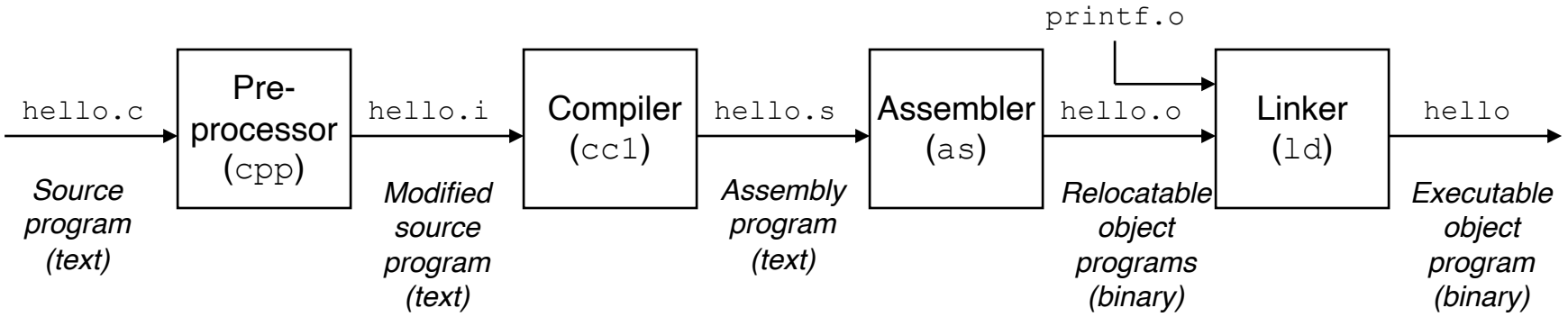
- "hello.c" is a source code
  - Sequence of bits (0 or 1)
  - 8-bit data chunks are called bytes
  - Each byte represents some text character in the program

#	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

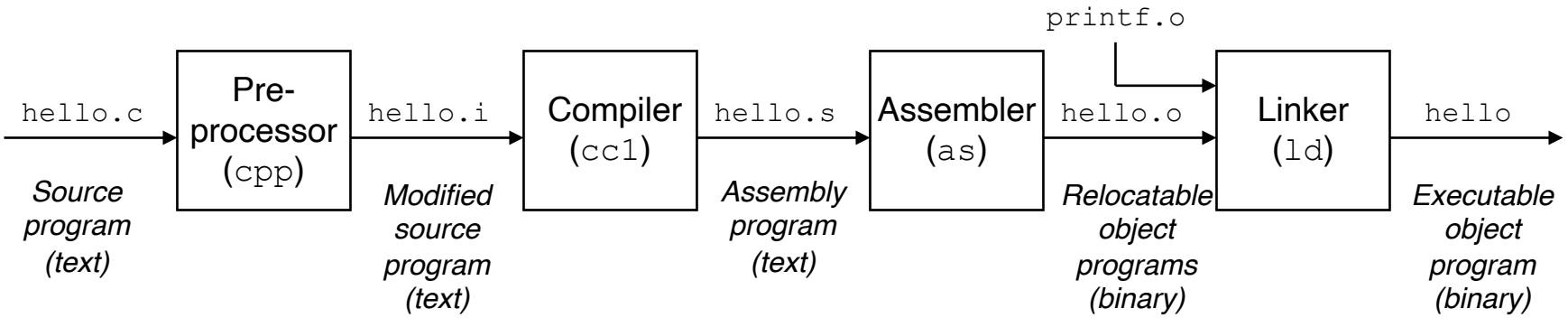
Figure 1.2 **The ASCII text representation of `hello.c`.**

Programs translated by other programs

- `unix> gcc –o hello hello.c`



Compilation System



- Pre-processing
  - E.g., `#include<stdio.h>` is inserted into hello.i
- Compilation (.s)
  - Each statement is an assembly language program
- Assembly (.o)
  - A binary file whose bytes encode mach. language instructions
- Linking
  - Get printf() which resides in a separate precompiled object file

Data Measurement Chart

Data Measurement	Size
Bit	Single Binary Digit (1 or 0)
Byte	8 bits
Kilobyte (KB)	1,024 Bytes
Megabyte (MB)	1,024 Kilobytes
Gigabyte (GB)	1,024 Megabytes
Terabyte (TB)	1,024 Gigabytes
Petabyte (PB)	1,024 Terabytes
Exabyte (EB)	1,024 Petabytes

The Decimal System

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers
- For example the number 83 means eight tens plus three:

$$83 = (8 * 10) + 3$$

- The number 4728 means four thousands, seven hundreds, two tens, plus eight:

$$4728 = (4 * 1000) + (7 * 100) + (2 * 10) + 8$$

- The decimal system is said to have a ***base***, or ***radix***, of 10. This means that each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position:

$$83 = (8 * 10^1) + (3 * 10^0)$$

$$4728 = (4 * 10^3) + (7 * 10^2) + (2 * 10^1) + (8 * 10^0)$$

The Binary System

- Only two digits, 1 and 0
- Represented to the base 2
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_2 = 0_{10}$$
$$1_2 = 1_{10}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

$$10_2 = (1 * 2^1) + (0 * 2^0) = 2_{10}$$
$$11_2 = (1 * 2^1) + (1 * 2^0) = 3_{10}$$
$$100_2 = (1 * 2^2) + (0 * 2^1) + (0 * 2^0) = 4_{10}$$

For representing numbers in base 2, there are two possible digits (0, 1) in which column values are a power of two:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	**1**	**1**	**0**	**0**	**0**	**1**	**1**

0 + 64 + 32 + 0 + 0 + 0 + 2 + 1 = **99**

Although values represented in base 2 are significantly longer than those in base 10, **binary representation is used in digital computing because of the resulting simplicity of hardware design**

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

1100 1001 0111 1011 → 0xC97B

Machine Words

- Machine has “word size”
  - Nominal size of integer-valued data
  - More importantly – a virtual address is encoded by such a word
    - Hence, it determines max size of virtual address space
  - Most current machines are 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Newer systems are 64 bits (8 bytes)
    - Potentially address ≈ 1.84 X 1019 bytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

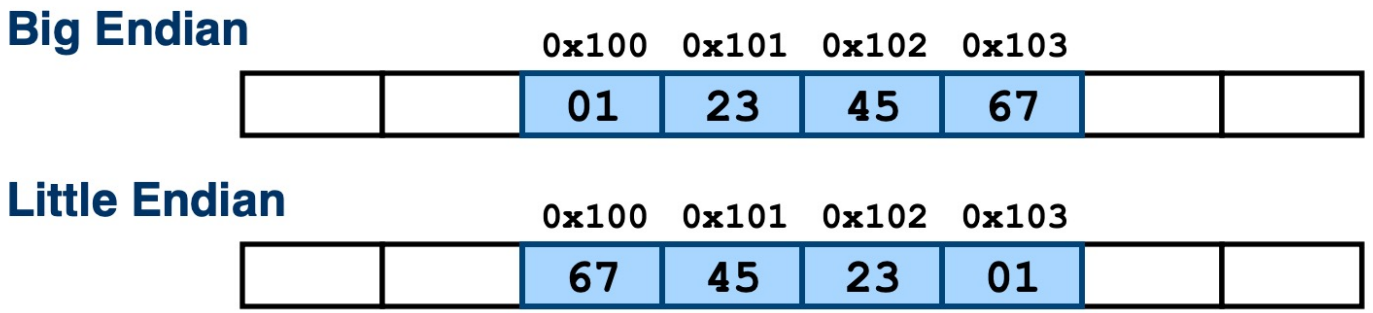
Data Sizes

- Each computer has a word size
  - For a machine with w-bit word size
  - The virtual address can range from 0 to 2^w-1
  - The program access to at most 2^w bytes

- 32 bit vs 64 bit

Byte Ordering

- How to order bytes within multi-byte word in memory
- Conventions
  - (most) Sun's, IBMs are "Big Endian" machines
    - Least significant byte has highest address (comes last)
  - (most) Intel's are "Little Endian" machines
    - Least significant byte has lowest address (comes first)
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100 0x100 0x101



http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Boolean Variables and Operations

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - ⟨{0,1}, |, &, ~, 0, 1⟩
  - | is “sum” operation, & is “product” operation
  - ~ is “complement” operation (not additive inverse)
  - 0 is identity for sum, 1 is identity for product
- Makes use of variables and operations
  - Are logical
  - A variable may take on the value 1 (TRUE) or 0 (FALSE)
  - Basic logical operations are AND, OR, XOR and NOT

Boolean Variables and Operations / 2

- AND
  - Yields true (binary value 1) if and only if both of its operands are true
  - In the absence of parentheses the AND operation takes precedence over the OR operation
  - When no ambiguity will occur the AND operation is represented by simple concatenation instead of the dot operator
- OR
  - Yields true if either or both of its operands are true
- NOT
  - Inverts the value of its operand

Table: Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \cdot Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \cdot Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs (A, B, . . .)

Operation	Expression	Output = 1 if
AND	$A \cdot B \cdot \ldots$	All of the set {A, B, …} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, …} are 1.
NAND	$\overline{A \cdot B \cdot \ldots}$	Any of the set {A, B, …} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, …} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, …} contains an odd number of ones.

Table: Basic Identities of Boolean Algebra

Basic Postulates		
$A \bullet B = B \bullet A$	$A + B = B + A$	Commutative Laws
$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$	$A + (B \bullet C) = (A + B) \bullet (A + C)$	Distributive Laws
$1 \bullet A = A$	$0 + A = A$	Identity Elements
$A \bullet \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \bullet A = 0$	$1 + A = 1$	
$A \bullet A = A$	$A + A = A$	
$A \bullet (B \bullet C) = (A \bullet B) \bullet C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \bullet B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \bullet \overline{B}$	DeMorgan's Theorem

Exercise 1

Evaluate the following expression when A =0, B =1, and C= 1

$F= B + \overline{C}A + B\overline{A} + A\overline{B}$

$1 + 0 + 1 + 0$

$1 + 1 = 1$

Simplify the following functions;

$F= AB + BC+\overline{B}C$

$F = A +\overline{A}\ B$

General Boolean Algebras

- Boolean operations can be extended to work on bit vectors
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

- All of the properties of Boolean algebra apply
- Now, Boolean |, & and ~ correspond to set union, intersection and complement

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Bit-Level Operations in C

- Operations &, |, ~, ^ Available in C
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - `~0x41 → 0xBE`
    - $\sim01000001_2 \rightarrow 10111110_2$
  - `~0x00 → 0xFF`
    - $\sim00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55 → 0x41`
    - $01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$
  - `0x69 | 0x55 → 0x7D`
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - **`&&, ||, !`**
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- Examples (char data type)
  - `!0x41 → 0x00`
  - `!0x00 → 0x01`
  - `!!0x41 → 0x01`
  - `0x69 && 0x55 → 0x01`
  - `0x69 || 0x55 → 0x01`
  - `p && *p` (avoids null pointer access)

Watch out for && vs. & (and || vs. |)…
one of the more common oopsies in C programming

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> 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
    - Useful in two's compliment
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Unsigned Encodings

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

(Binary To Unsigned)

e.g. B2U ([1011]) = $1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11$

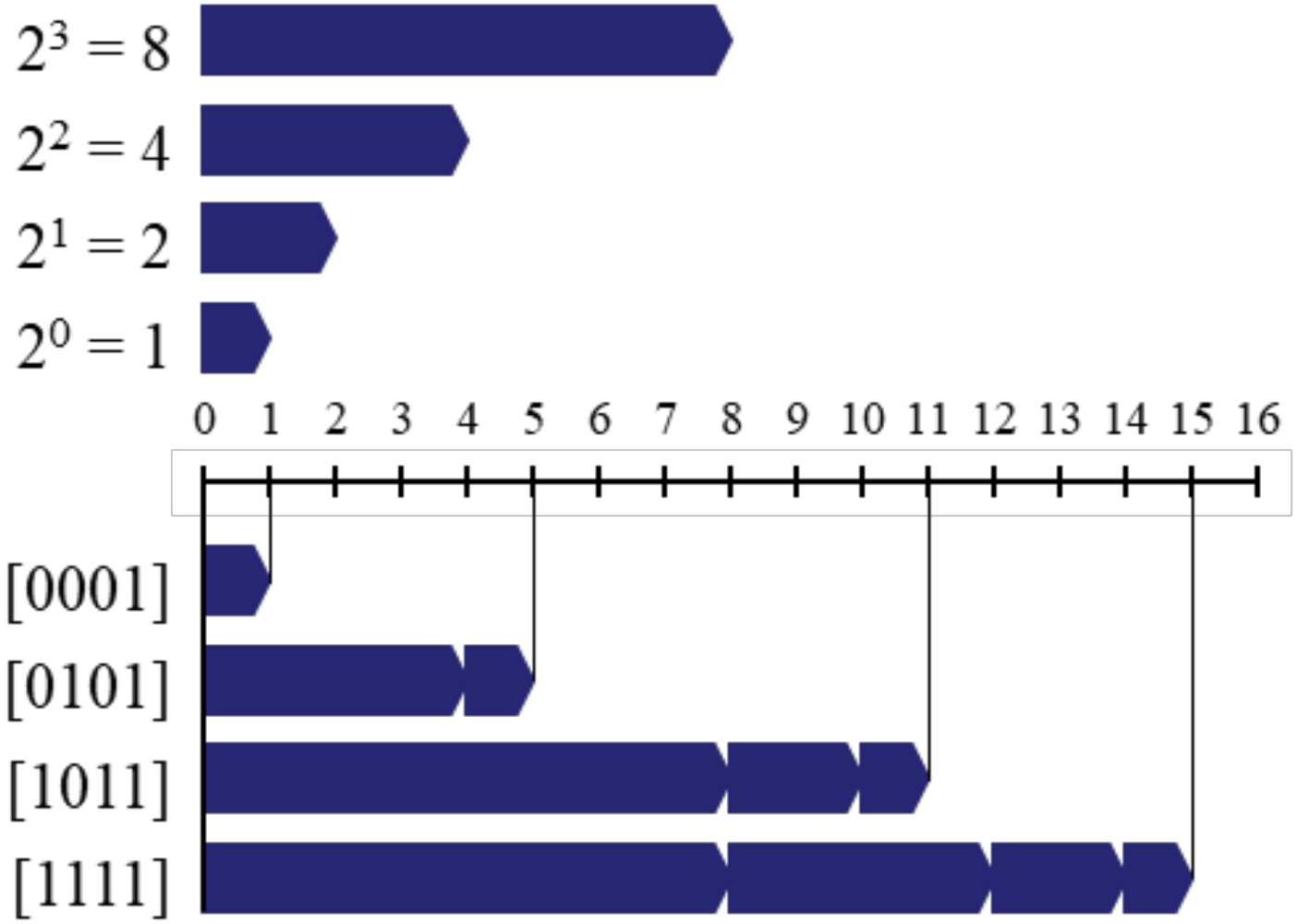
– C `short` 2 bytes long

```
short int x = 15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101

http://www.aqualab.cs.northwestern.edu/component/attachments/download/168

Examples


$2^3 = 8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[0001]
[0101]
[1011]
[1111]

Two's Complement

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

Sign Bit

- e.g. B2T ([1011]) = $-1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = -5$
- C `short` 2 bytes long

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

	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

http://www.aqualab.cs.northwestern.edu/component/attachments/download/168

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

	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**

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

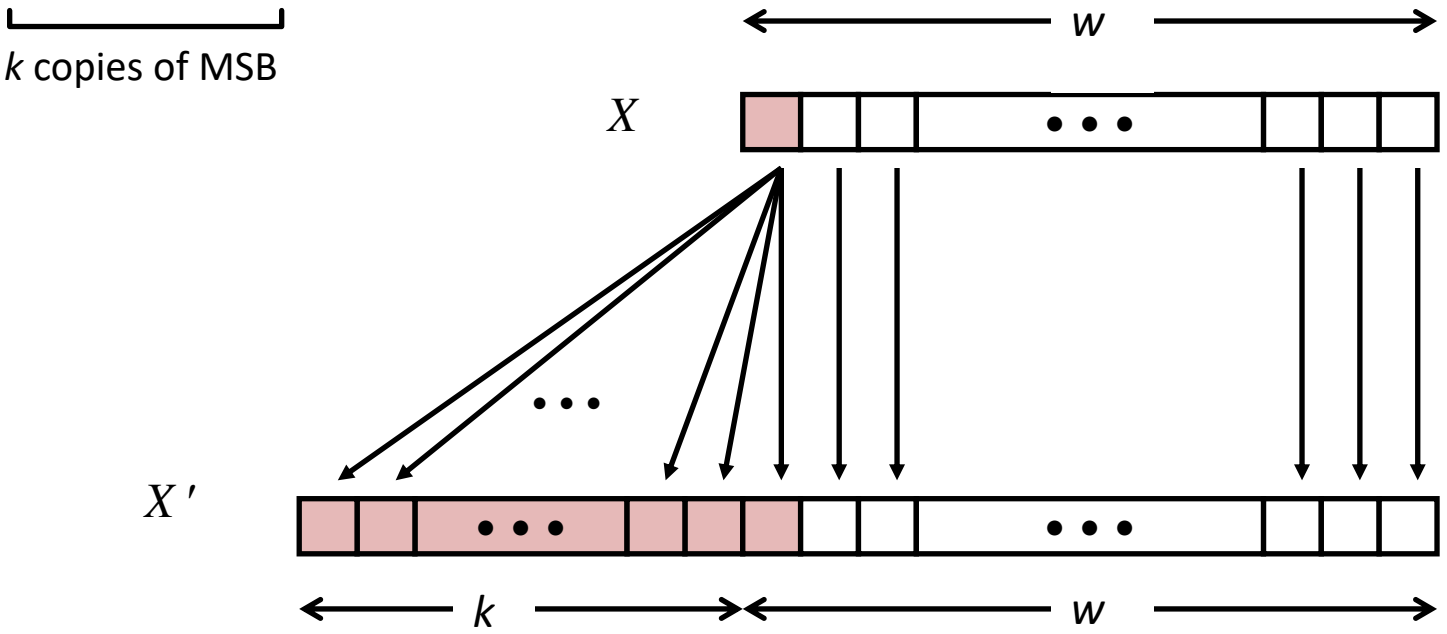
    ```
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```

  - Implicit casting also occurs via assignments and procedure calls

    ```
    tx = ux;
    uy = ty;
    ```

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:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



Truncating Numbers

- Reduce the number of bits representing the number
- Truncating *w-bit* number to a *k* bit number, we drop the high order *w-k* bits
  - Can alter its value
    - A form of overflow

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

OVERFLOW RULE:

- If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Unsigned Binary Multiplication

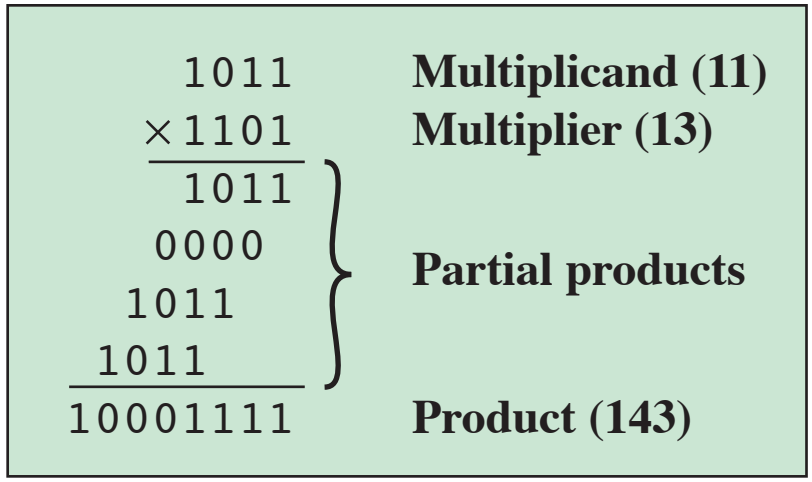
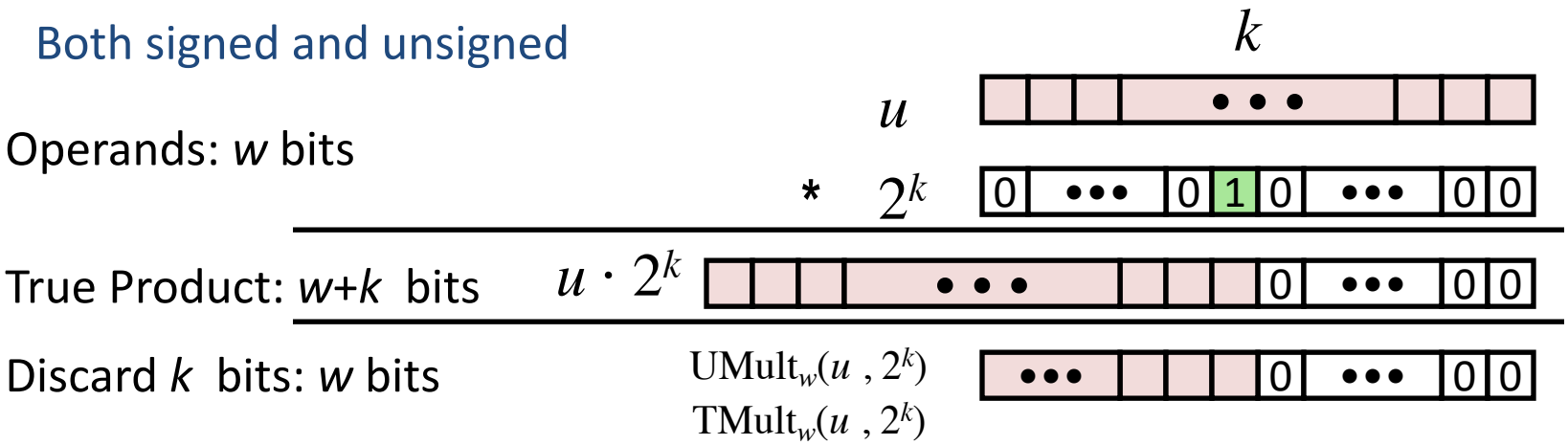


Figure 10.7 Multiplication of Unsigned Binary Integers

Power-of-2 Multiply with Shift

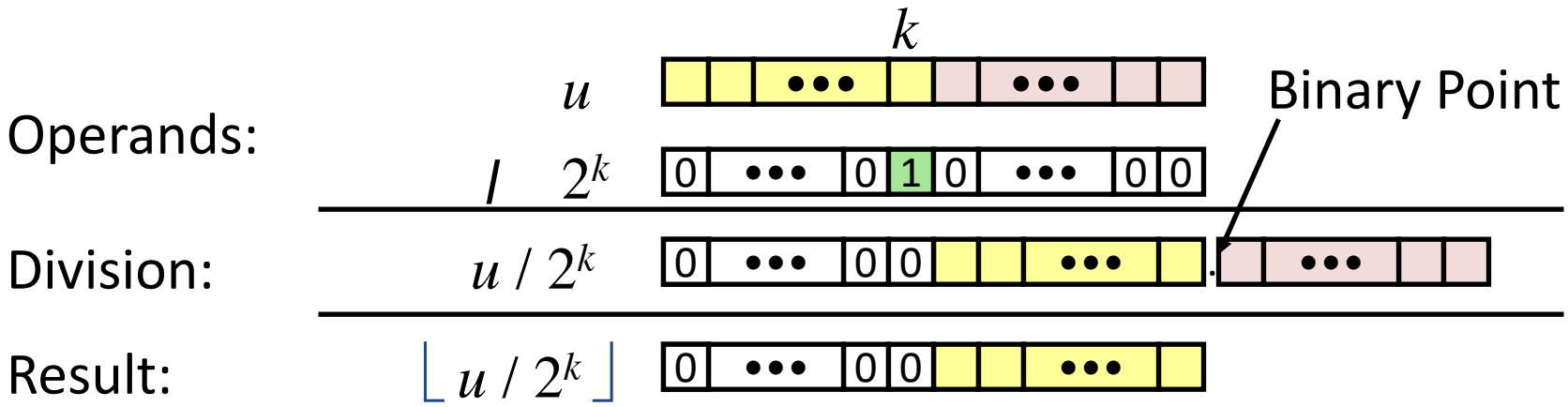
- **Operation**
  - `u << k` gives `u * 2^k`
  - Both signed and unsigned



- **Examples**
  - `u << 3 == u * 8`
  - `(u << 5) - (u << 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 >> 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

IEEE Floating Point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Floating Point Representation

- Numerical Form:

$$(-1)^s \, M \;\; 2^E$$

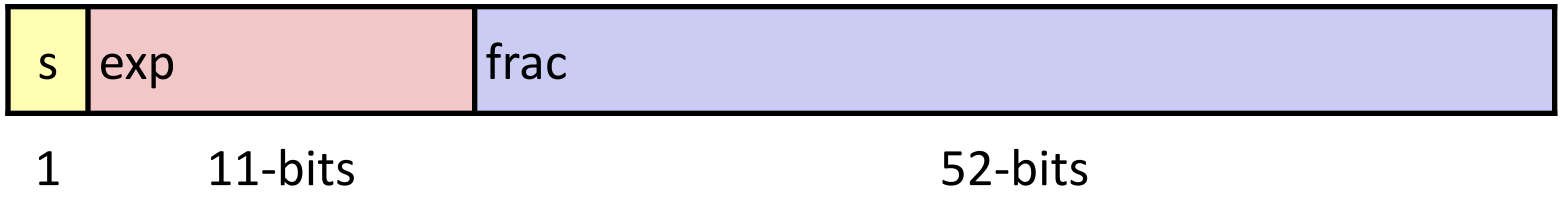
  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

- Encoding
  - MSB S is sign bit s
  - exp field encodes E (but is not equal to E)
  - frac field encodes M (but is not equal to M)

Precision options

- Single precision: 32 bits

- Double precision: 64 bits



- The value encoded by a given bit representation can be divided into three different cases, depending on the value of **exp**.
- Case 1: Normalized Values
- Case 2: Denormalized Values
- Case 3: Special Values

Rounding

- **Rounding Modes (illustrate with $ rounding)**

	$1.40	**$1.60**	**$1.50**	**$2.50**	**–$1.50**
– Towards zero	$1	$1	$1	$2	–$1
– Round down (−∞)	$1	$1	$1	$2	–$2
– Round up (+∞)	$2	$2	$2	$3	–$1
– Nearest Even (default)	$1	$2	$2	$2	–$2

Our Coverage

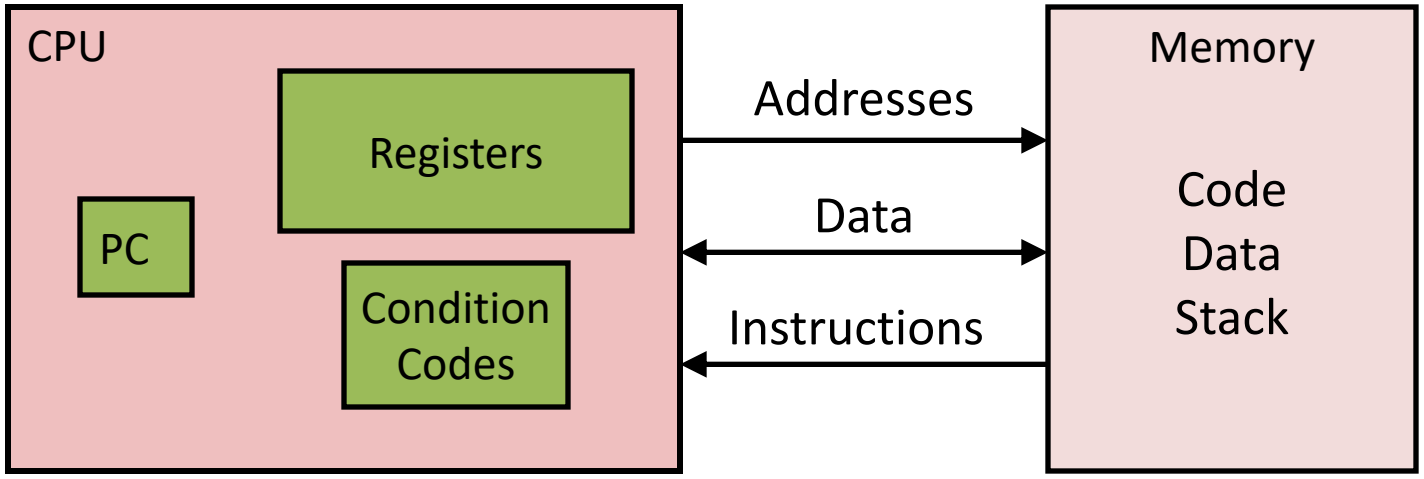
- IA32
  - The traditional x86
  - For ???: RIP, Fall 2018
- x86-64
  - The standard
  - `moat.cis.uab.edu`
  - `gcc hello.c`
  - `gcc –m64 hello.c`
- Presentation
  - Book covers x86-64
  - So, we will cover x86-64

• C, assembly, machine code

Definitions

- Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones

Assembly/Machine Code View



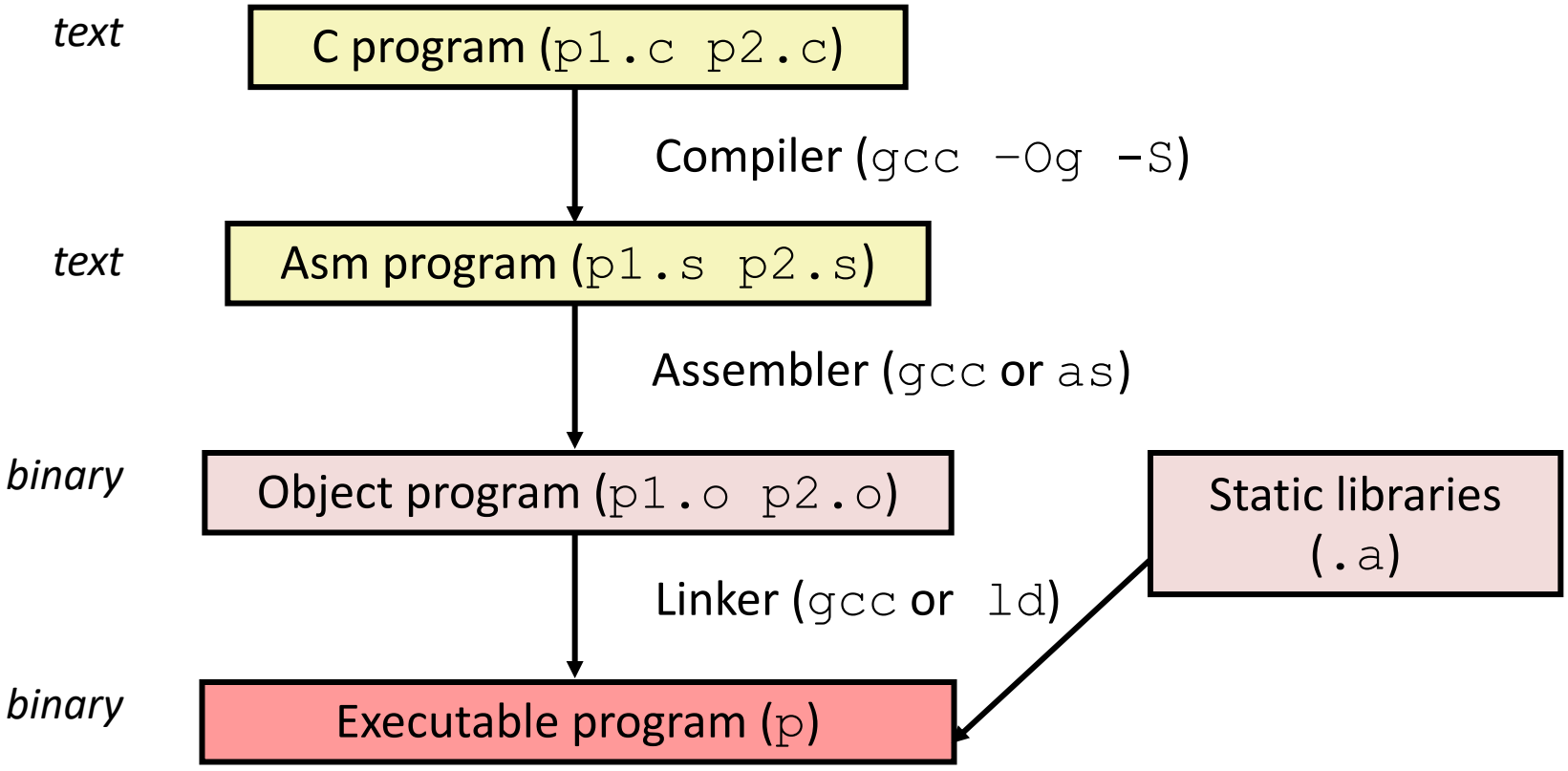
Programmer-Visible State

- **PC: Program counter**
  - Indicates the address of next instruction
  - Called "%rip" (x86-64)
- **Register file**
  - Heavily used program data
  - 16 named locations storing 64bit values
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

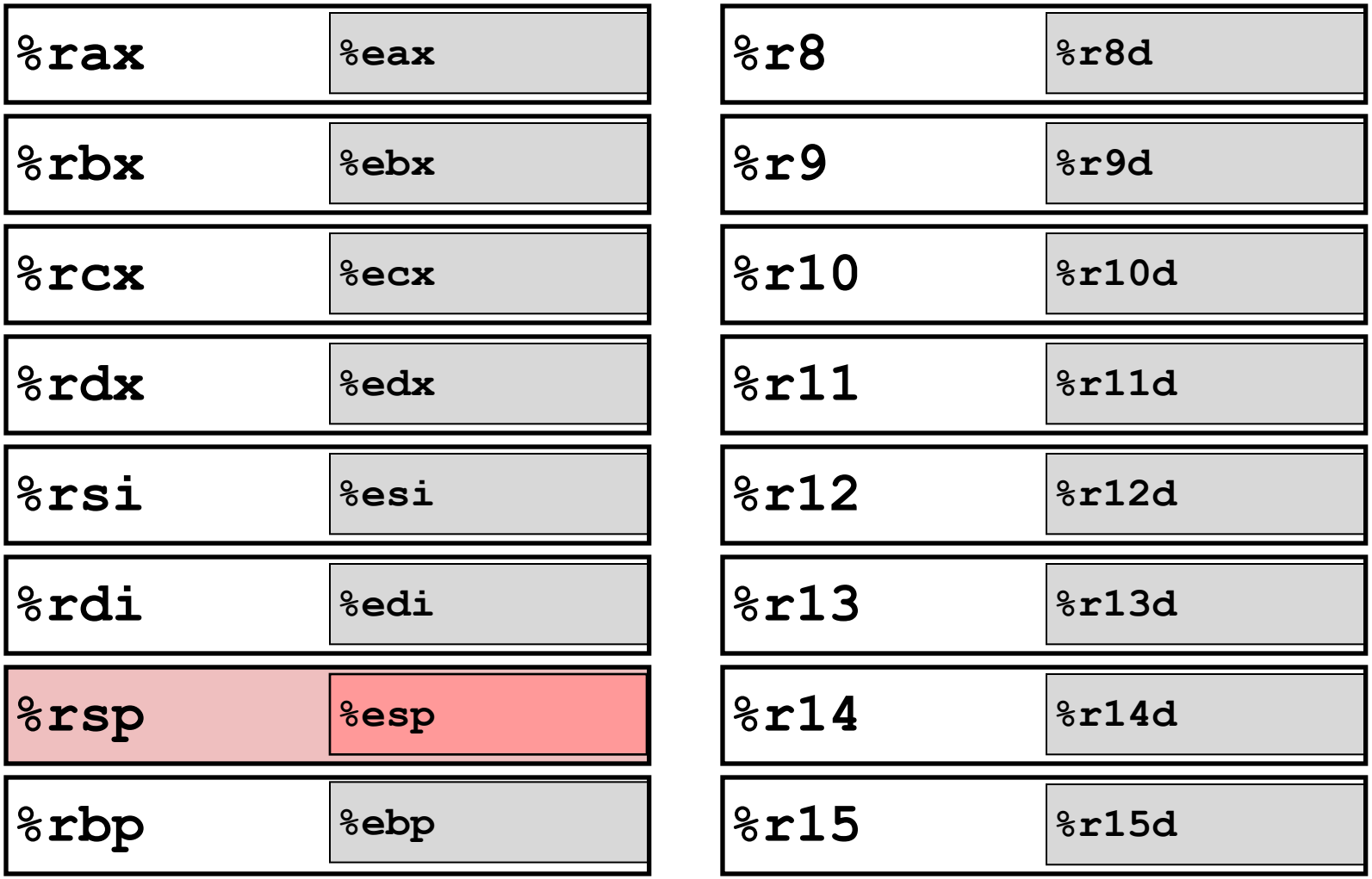
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Turning C into Object Code

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



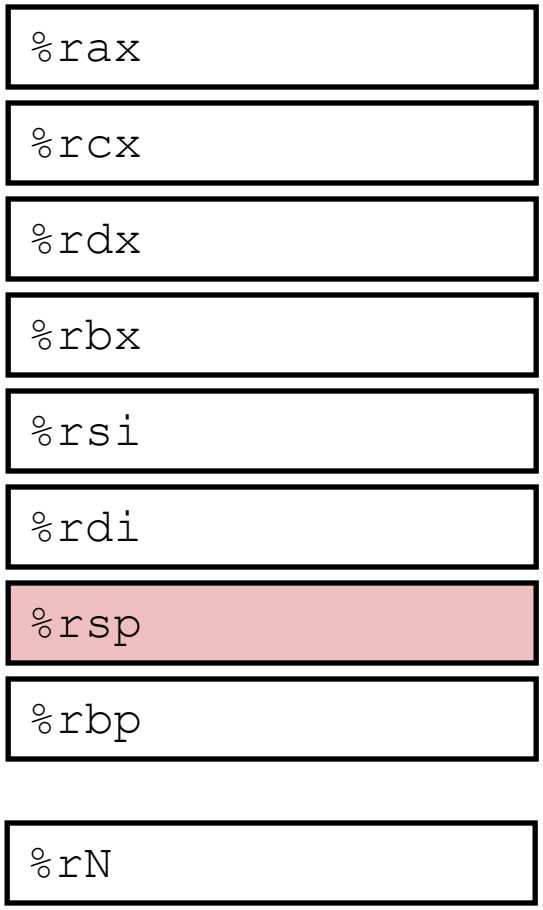
x86-64 Integer Registers



– Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

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 16 integer registers
    - Example: **`%rax`**, **`%r13`**
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other "address modes"



`movq` Operand Combinations

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

Cannot do memory-memory transfer with a single instruction

Pushing and Popping Stack Data

- Stack plays a vital role in the handling of procedure calls.
- Stack = a data structure where values can be added or deleted → according to last in first out discipline

- Add data to stack → push
- Remove data from stack → pop

- The stack pointer %rsp holds the address of the top stack element.
- Pushing a quad word value onto the stack involves first decrementing the stack pointer by 8 and then writing the value at the new top of stack address.
- `pushq %rbp` is equal to

```
subq &8, %rsp
movq %rbp, (%rsp)
```

Address Computation Instruction

- **`leaq`** Source, Destination
  - Source is address mode expression
  - Set Destination to address denoted by expression

  leaq S, D D ← &S Load effective address

- 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, or 8
- Example

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

Converted to ASM by compiler:

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

ADD

`ADDQ  %rbx,  %rax`

- adds %rbx to %rax, and overwrites the result in %rax

`ADDQ  &8,  %rsp`

adds 8 to the stack pointer %rsp, (incrementing)

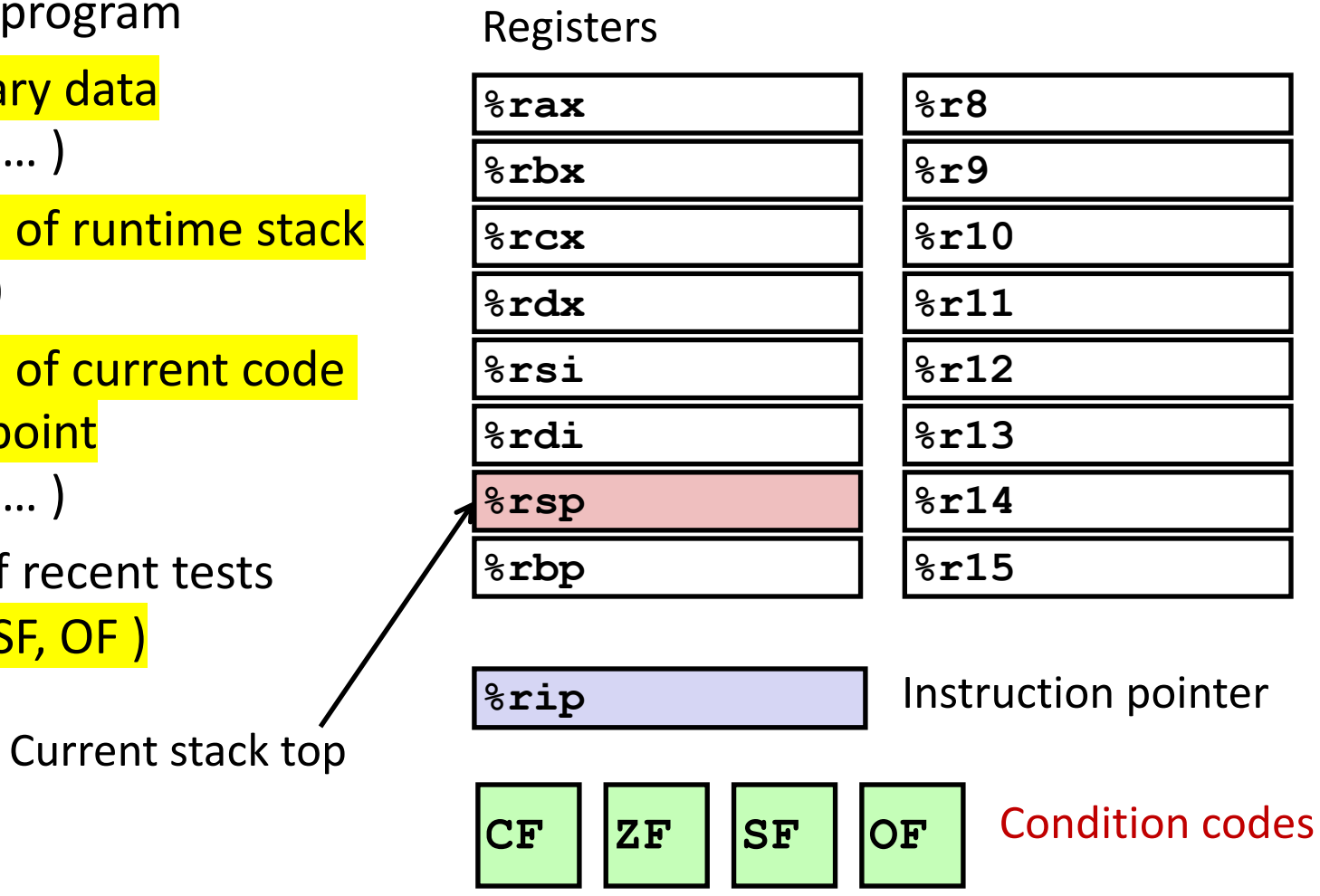
Some Arithmetic Operations

- Two Operand Instructions:

Format		Computation
`addq`	Src,Dest	Dest = Dest + Src
`subq`	Src,Dest	Dest = Dest – Src
`imulq`	Src,Dest	Dest = Dest * Src
`salq`	Src,Dest	Dest = Dest << Src Also called shlq
`sarq`	Src,Dest	Dest = Dest >> Src Arithmetic
`shrq`	Src,Dest	Dest = Dest >> Src Logical
`xorq`	Src,Dest	Dest = Dest ^ Src
`andq`	Src,Dest	Dest = Dest & Src
`orq`	Src,Dest	Dest = Dest | Src

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)



Condition Codes (Explicit Setting: Compare)

- **Explicit Setting 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)
```

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  - –**`testq`** Src2, Src1
    - **`testq b,a`** like computing **`a&b`** without setting destination
  - –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`**

Conditional Branching: 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)

“For” Loop Form

General Form

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

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

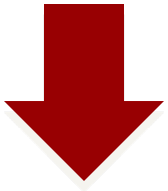
Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

“For” Loop → While Loop

For Version

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

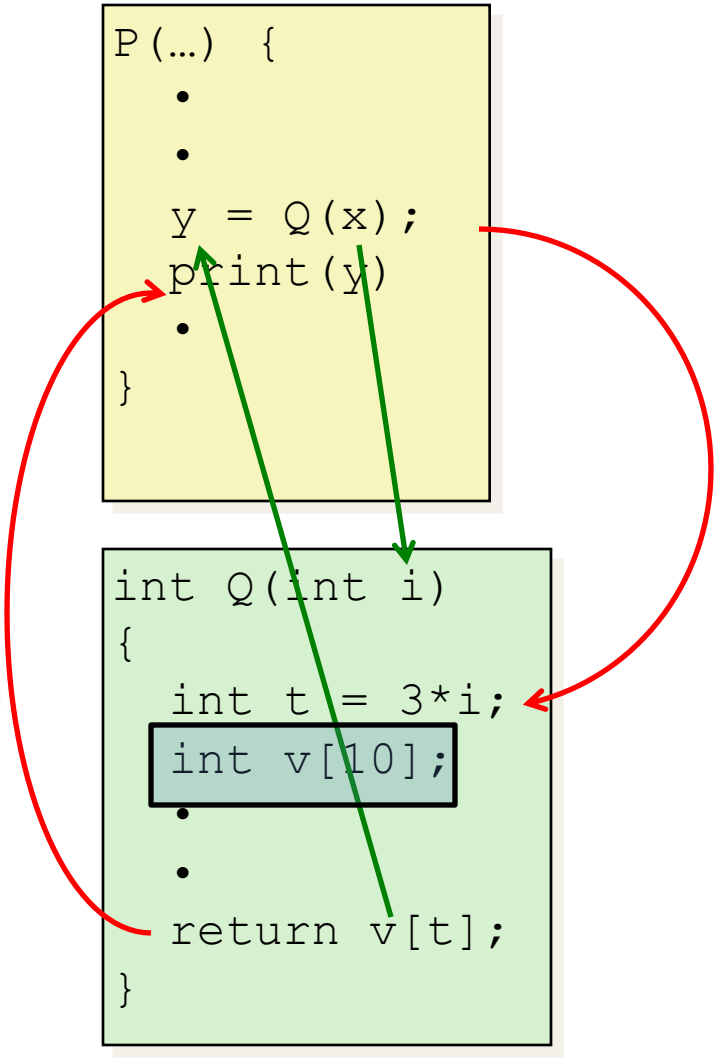


While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

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 implementation of a procedure uses only those mechanisms required**



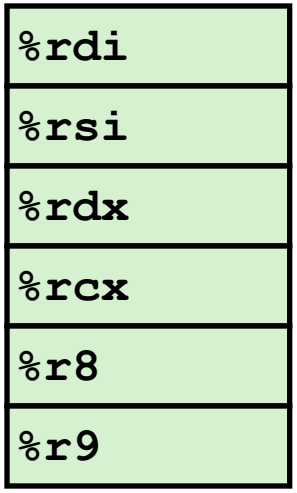
Procedure Control Flow

- 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

Passing data : Procedure Data Flow

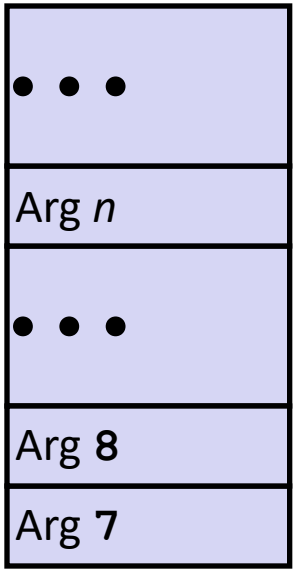
Registers

- First 6 arguments



- Return value

Stack



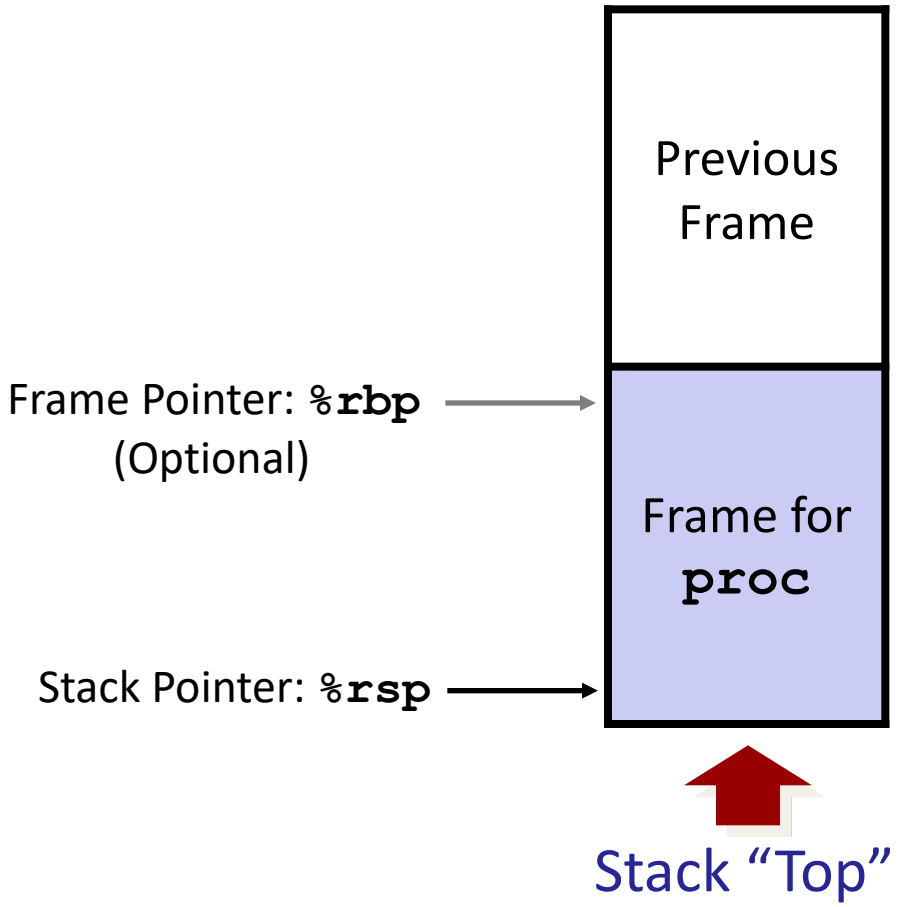
- Only allocate stack space when needed

Managing local data : 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 discipline**
  - 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

Stack Frames

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

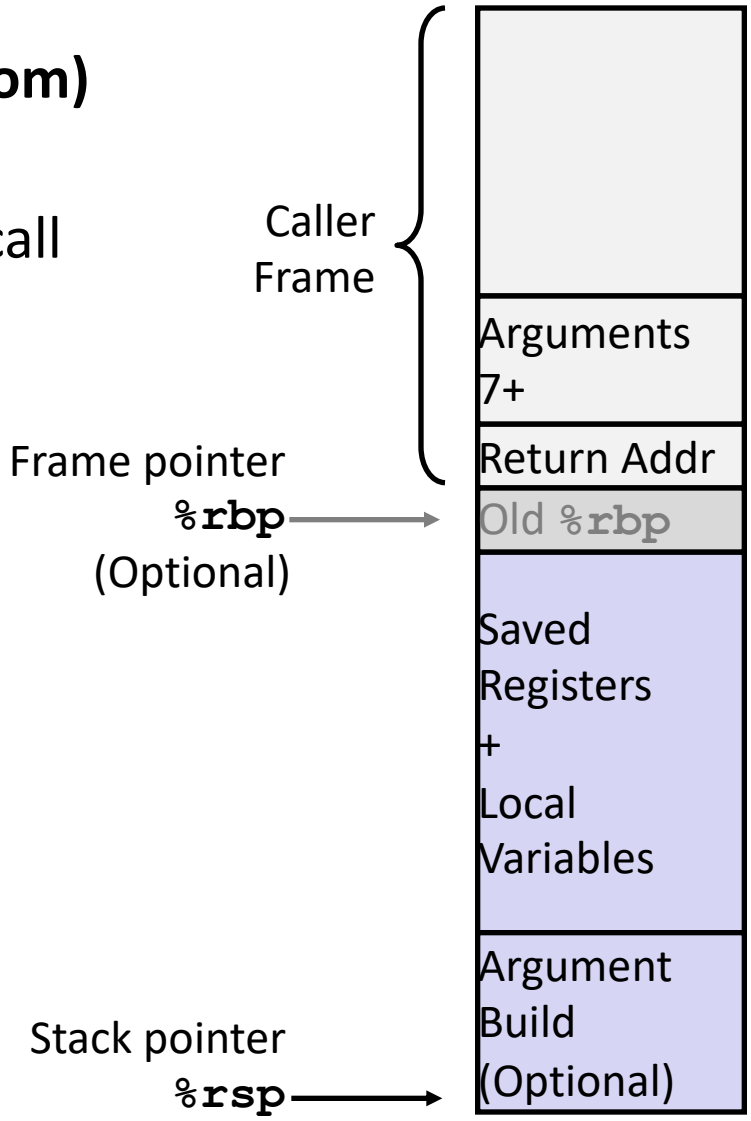


- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction

x86-64/Linux 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



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
    - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - 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

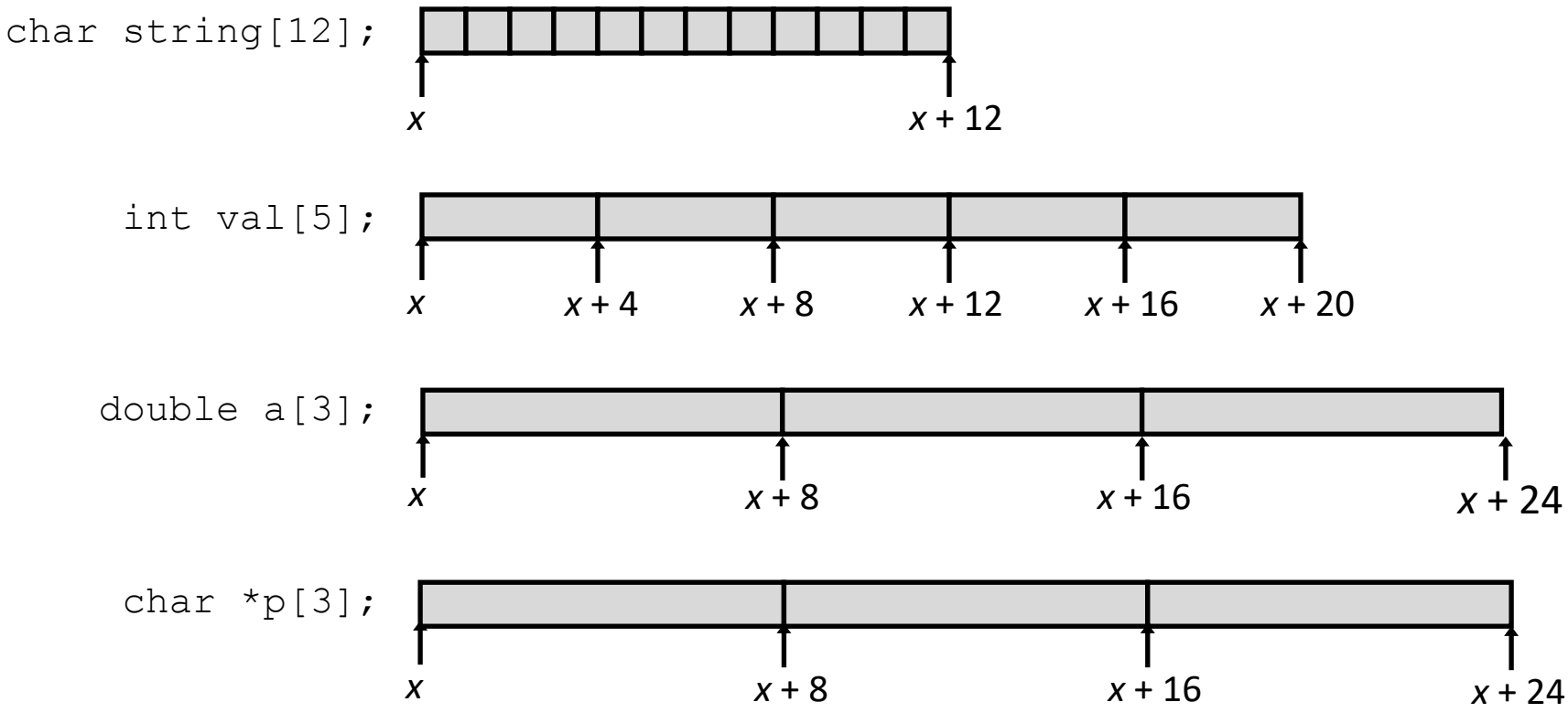
Array Allocation

- Basic Principle

  T **`A[`***L***`];`**

  – Array of data type *T* and length *L*

  – Contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes in memory



recall pointers

- The unary operators & and * allow generation and dereferencing of pointers
- An expression Expr → denoting some object;
  - &Expr is a pointer giving the address of the object
- An expression AExpr denoting an address;
  - *AExpr gives the value at that address
- Thus;
  - Expr = *&AExpr

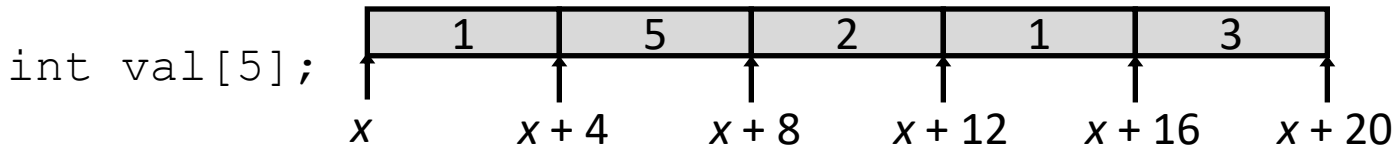
Array Access

- Basic Principle

  T **A[***L***];**

  – Array of data type *T* and length *L*

  – Identifier **A** can be used as a pointer to array element 0: Type *T**



- Reference Type Value

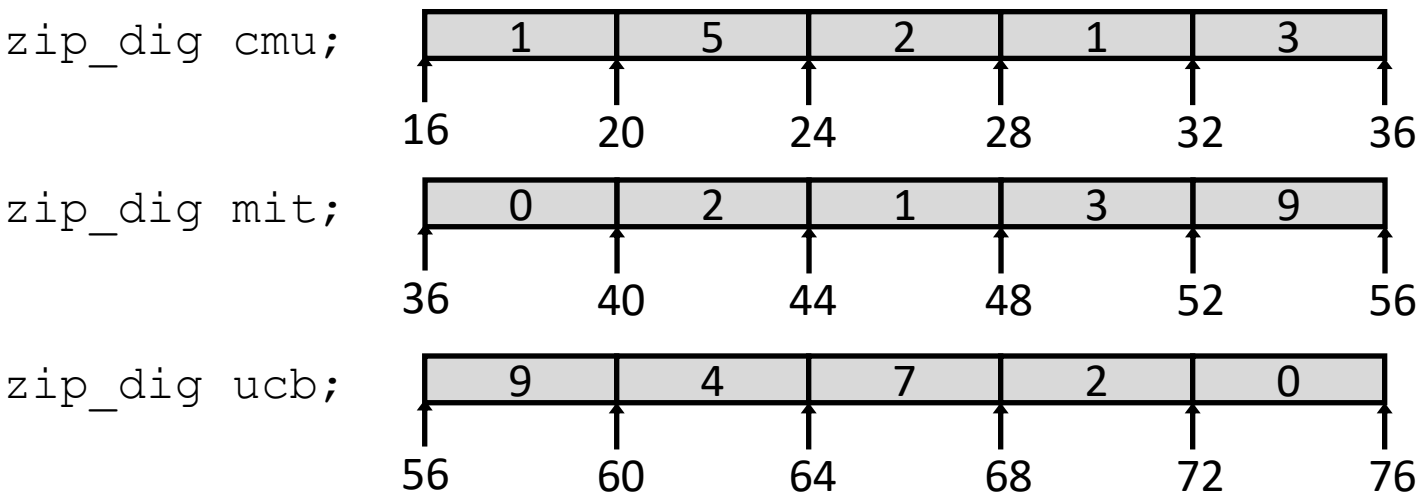
Reference	Type	Value
val[4]	**int**	3
val	**int ***	*x*
val+1	**int ***	*x* + 4
&val[2]	**int ***	*x* + 8
val[5]	**int**	??
***(val+1)**	**int**	5
val + *i*	**int ***	*x* + 4 *i*

- The memory referencing instruction of x86_64 are designed to simplify array access.
- For ex: suppose E is an array of values of type int and we wish to evaluate E[i], where the **address of E** is stored in register **%rdx** and **i** is stored in register **%rcx**. Then the instruction is:
  - movl (%rdx, %rcx,4), %eax
- which performs the address computation xE+4i, read that memory location, and copy the result to register %eax

Array Example

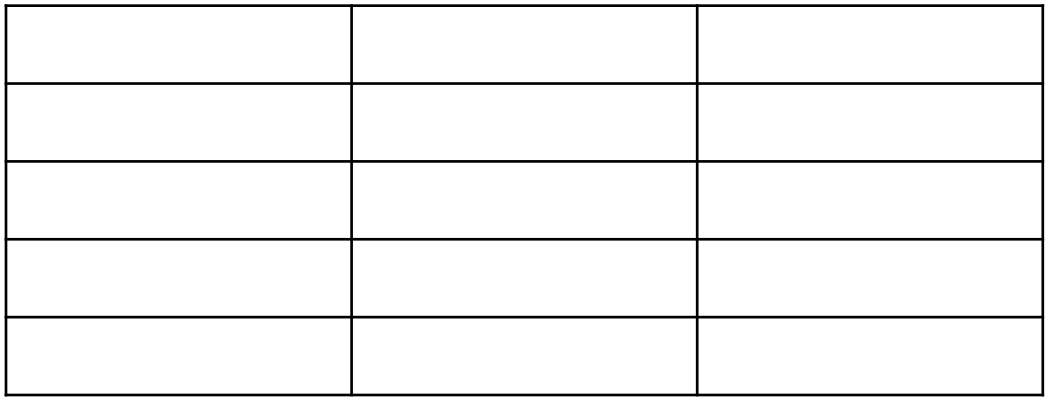
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Multidimensional Arrays



```
int A[5] [3];
```

is equivalent to the declaration;

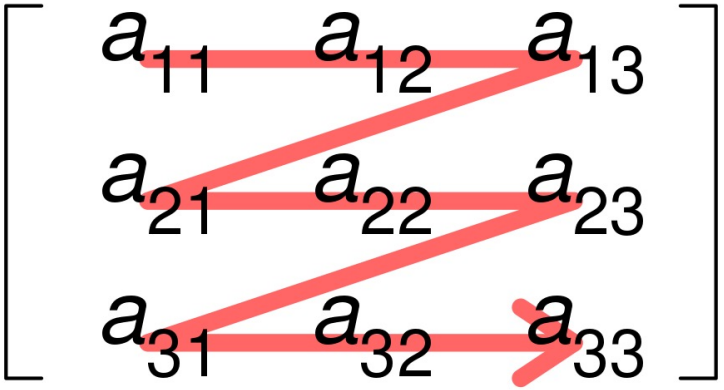
```
typedef int row3_t;
row3_t A[5];
```

Storing in the memory

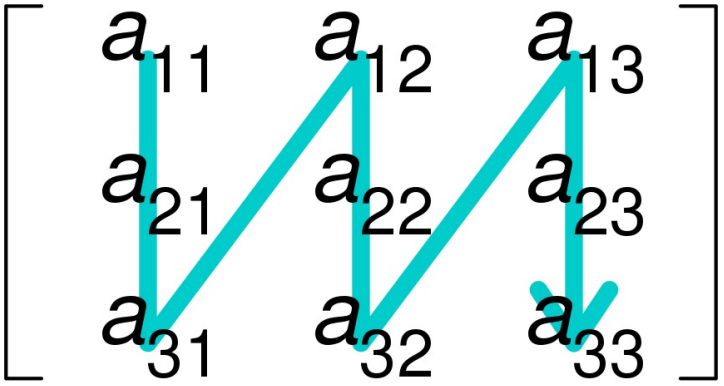
- The array elements are ordered in memory in ***row-major*** order;
  - all elements of row 0 (A[0][i])
  - followed by all elements of row 1 (A[1][i])
  - ….
  - …
- To access elements of multidimensional arrays, the compiler generates code to compute the offset of the desired element and then uses one of the mov instructions with the start of the array as the base address.

Row Major – Column Major

Row-major order



Column-major order



https://en.wikipedia.org/wiki/Row-_and_column-major_order#/media/File:Row_and_column_major_order.svg

C - Structures

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

https://www.tutorialspoint.com/cprogramming/c_structures.htm

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
struct [structure tag] {

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

https://www.tutorialspoint.com/cprogramming/c_structures.htm

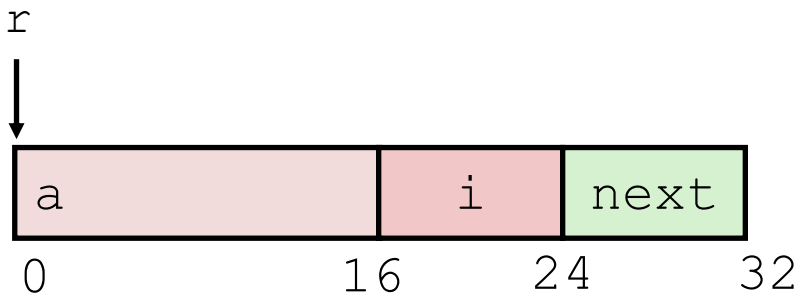
The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure –

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

https://www.tutorialspoint.com/cprogramming/c_structures.htm

Structure Representation

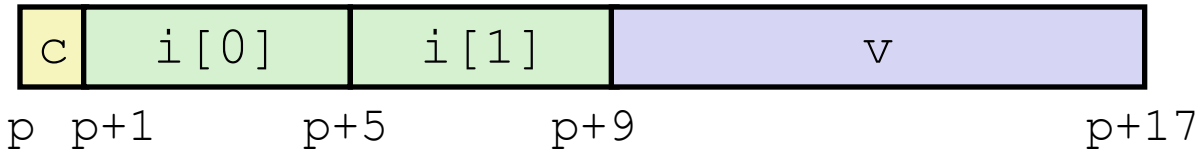
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

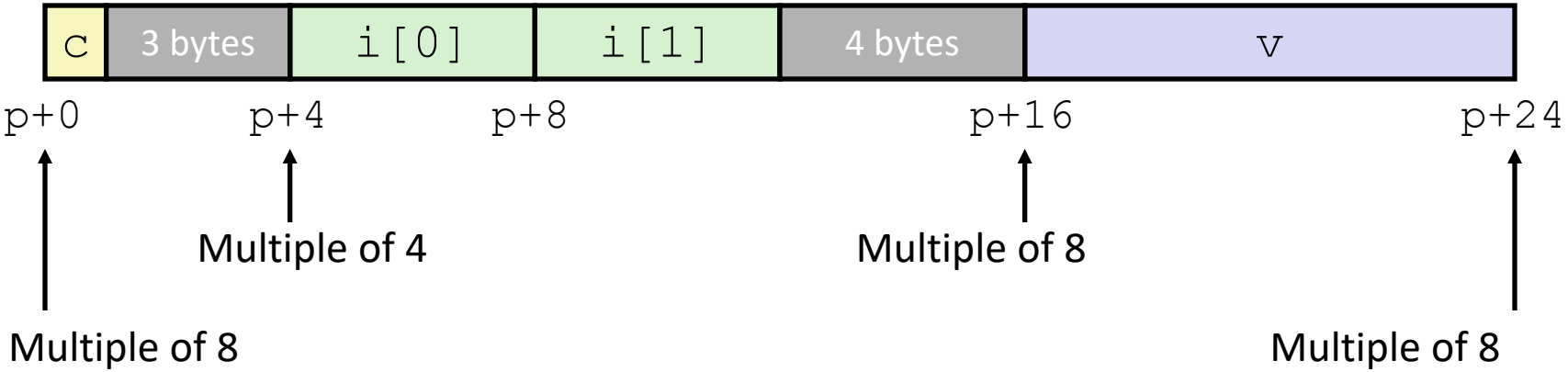
Structures & Alignment

- Unaligned Data




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



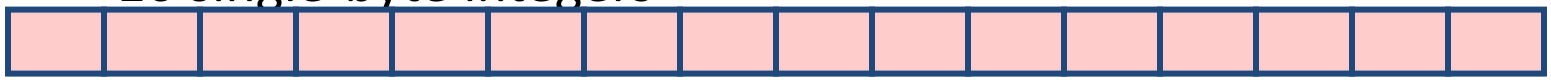
Alignment Principles

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

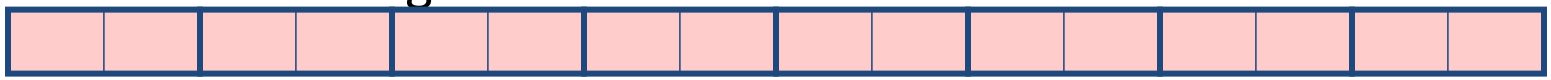
Programming with SSE3

XMM Registers

- 16 total, each 16 bytes
- 16 single-byte integers



- 8 16-bit integers



- 4 32-bit integers

- 4 single-precision floats

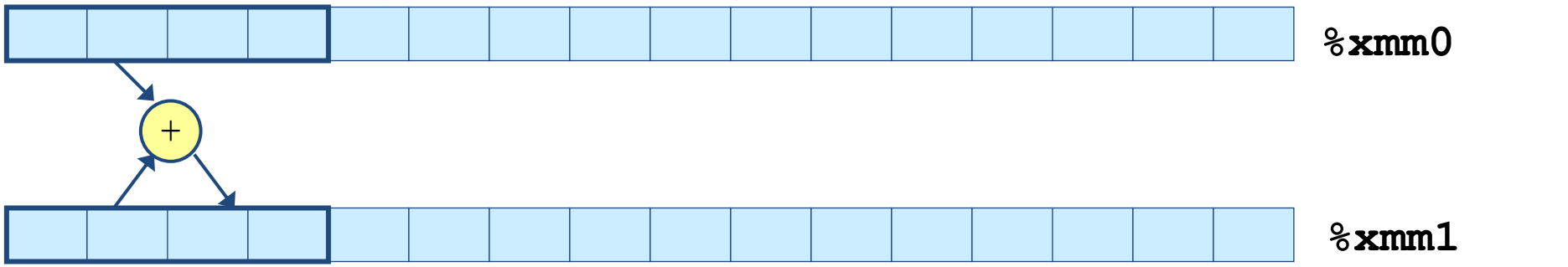
- 2 double-precision floats

- 1 single-precision float

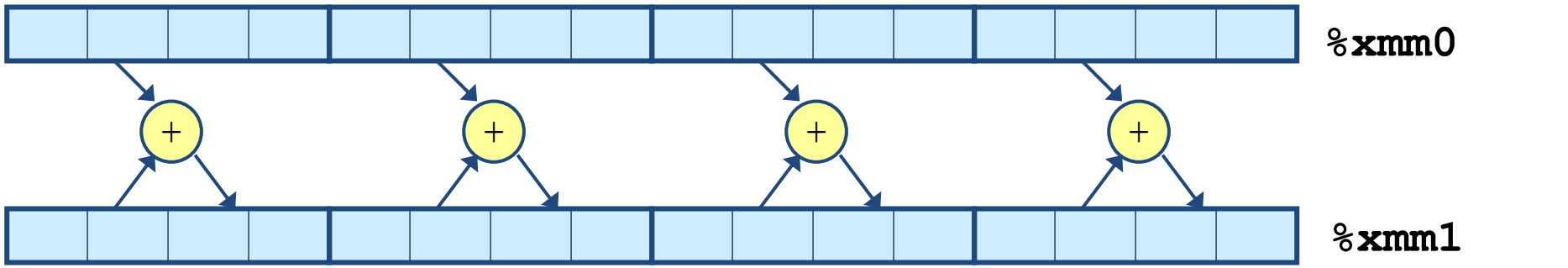
- 1 double-precision float

Scalar & SIMD Operations

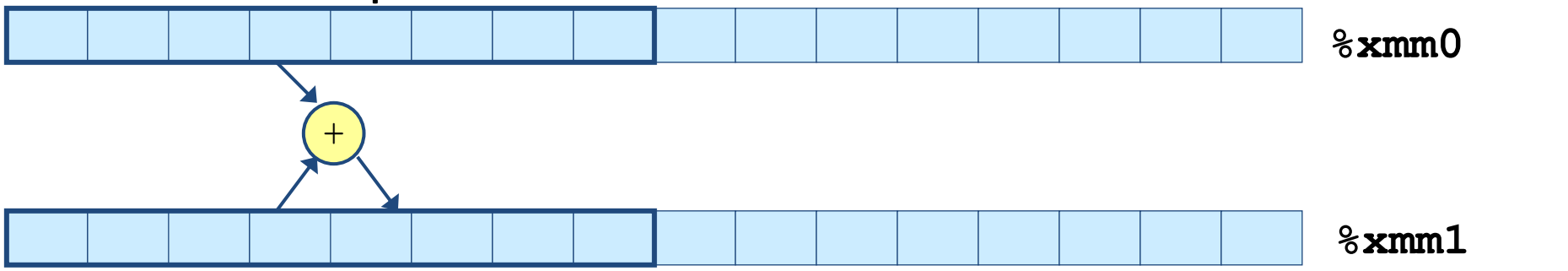
- Scalar Operations: Single Precision `addss %xmm0,%xmm1`



- SIMD Operations: Single Precision `addps %xmm0,%xmm1`



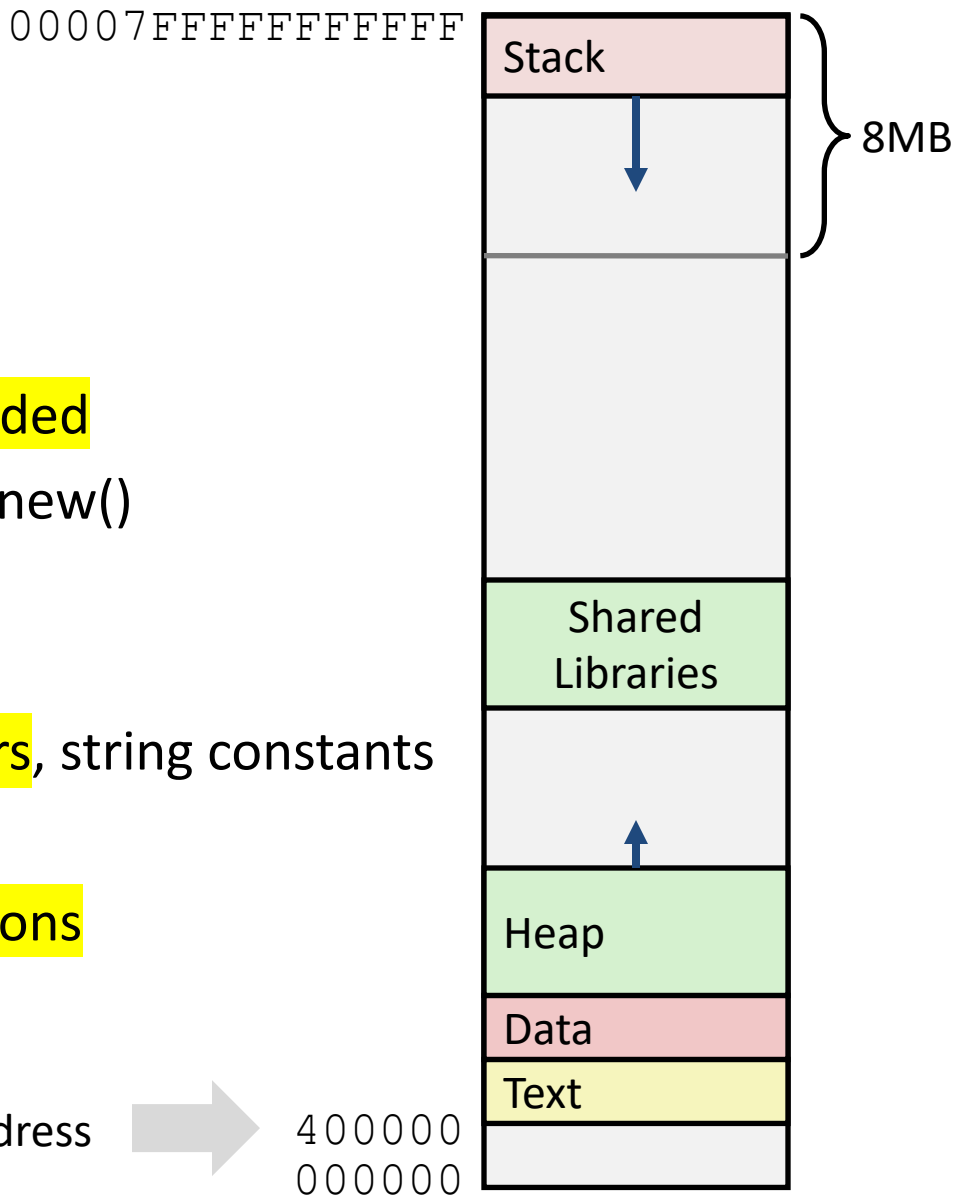
- Scalar Operations: Double Precision `addsd %xmm0,%xmm1`



x86-64 Linux Memory Layout

not drawn to scale

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



Such problems are a BIG deal

- Generally called a "buffer overflow"
  - when exceeding the memory size allocated for an array
- Why a big deal?
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- Most common form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

String Library Code

- Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

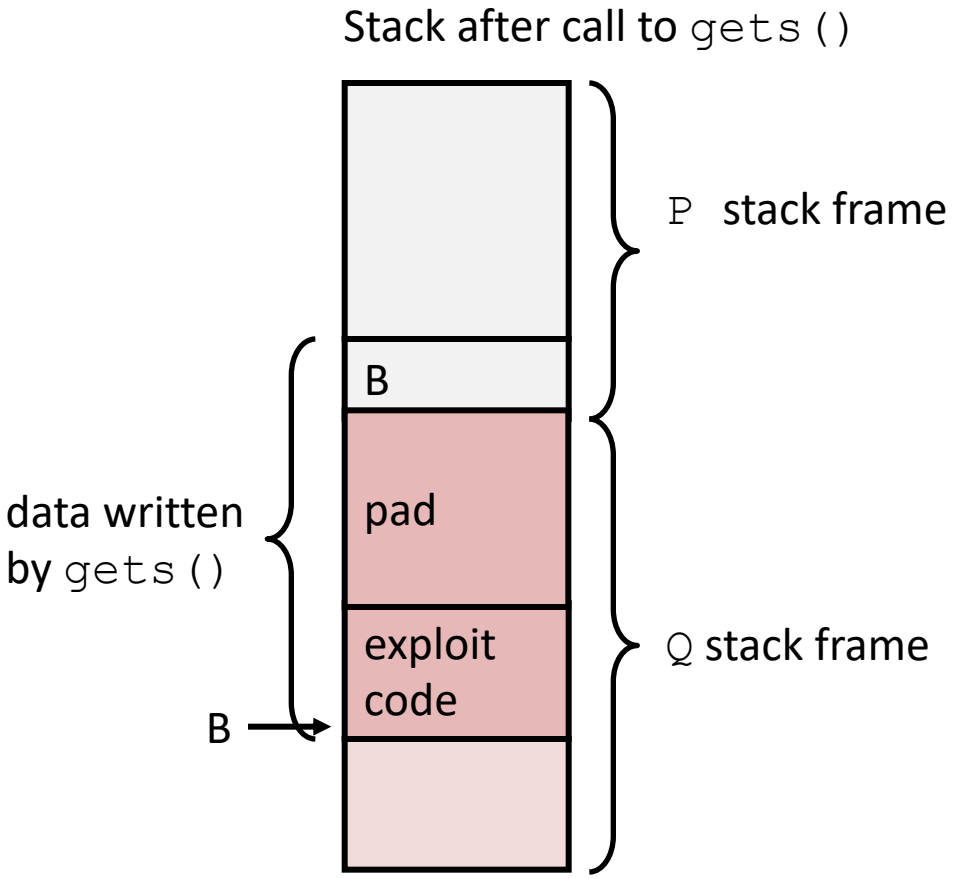
  - No way to specify limit on number of characters to read
- Similar problems with other library functions
  - **`strcpy`**, **`strcat`**: Copy strings of arbitrary length
  - **`scanf`**, **`fscanf`**, **`sscanf`**, when given **`%s`** conversion specification

Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes ret, will jump to exploit code

Exploits Based on Buffer Overflows

- *Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines*
- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (1988)
  - "IM wars" (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

- Lets talk about each…

Performance Realities

- ***There's more to performance than asymptotic complexity***
- **Constant factors matter too!**
  - You will learn in CS332, the performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files
- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
    a[n*i+j] = b[j];
}
```

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x  -->  x << 4`

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```



```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
leaq   1(%rsi), %rax  # i+1
leaq   -1(%rsi), %r8  # i-1
imulq  %rcx, %rsi     # i*n
imulq  %rcx, %rax     # (i+1)*n
imulq  %rcx, %r8      # (i-1)*n
addq   %rdx, %rsi     # i*n+j
addq   %rdx, %rax     # (i+1)*n+j
addq   %rdx, %r8      # (i-1)*n+j
```

```
imulq    %rcx, %rsi  # i*n
addq%rdx, %rsi  # i*n+j
movq%rsi, %rax  # i*n+j
subq%rcx, %rax  # i*n+j-n
leaq(%rsi,%rcx), %rcx # i*n+j+n
```

Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Optimization Blocker: Procedure Calls

- ***Why couldn't compiler move `strlen` out of inner loop?***
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`
- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations near them
- **Remedies:**
  - Use of inline functions
    - GCC does this with –O1
      - Within single file
  - Do your own code motion

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
    s++; length++;
    }
    lencnt += length;
    return length;
}
```

Memory Matters

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
    b[i] = 0;
    for (j = 0; j < n; j++)
        b[i] += a[i*n + j];
    }
}
```

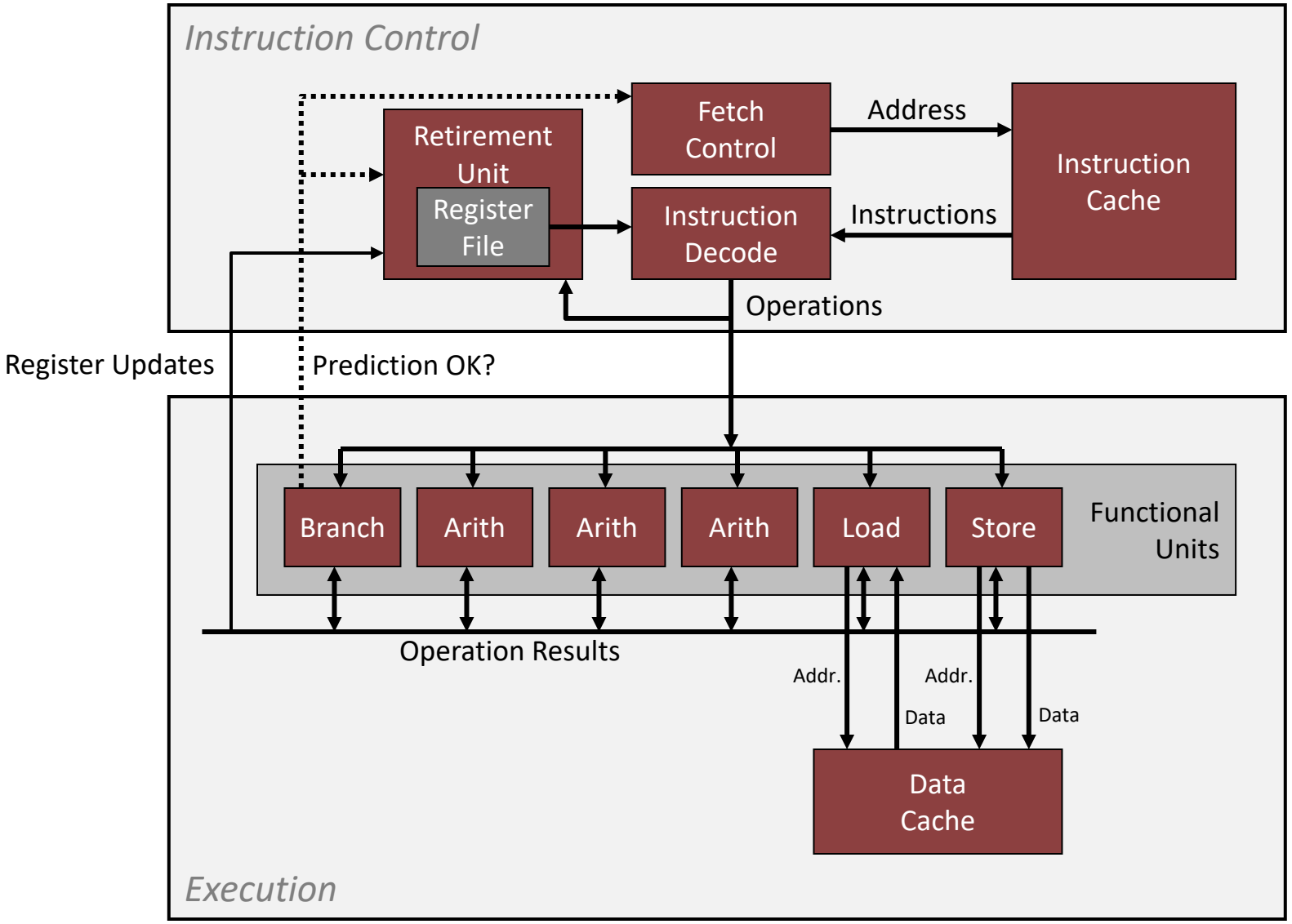
```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0   # FP load
        addsd   (%rdi), %xmm0      # FP add
        movsd   %xmm0, (%rsi,%rax,8)   # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

Modern CPU Design



Superscalar Processor

- Definition: A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

Loop Unrolling (2x1)

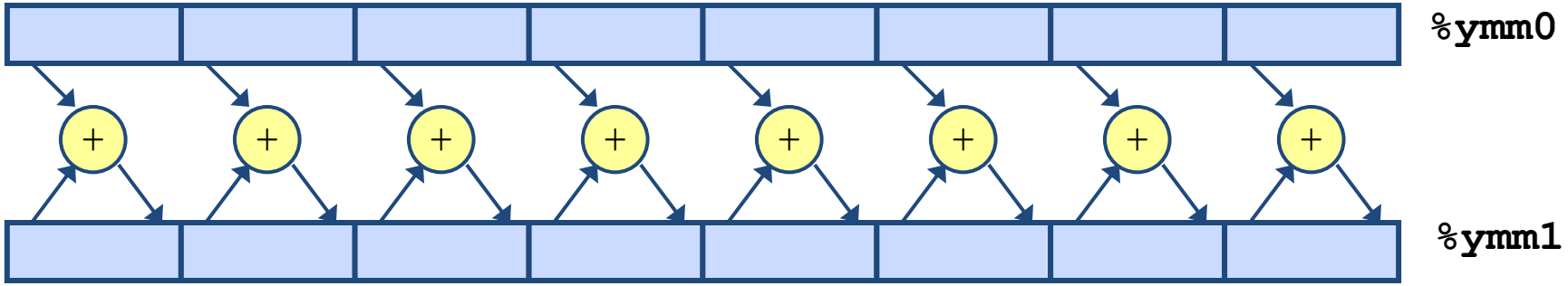
```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

SIMD Operations

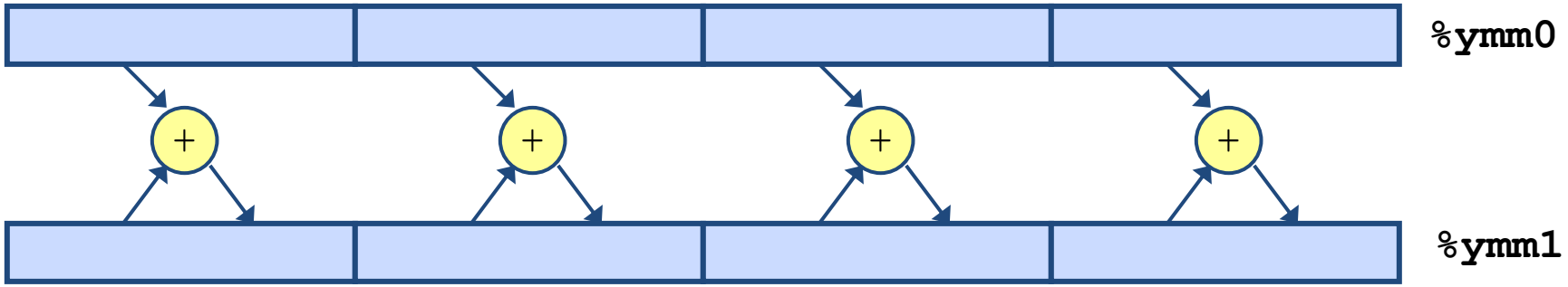
- SIMD Operations: Single Precision

```
vaddsd %ymm0, %ymm1, %ymm1
```



- SIMD Operations: Double Precision

```
vaddpd %ymm0, %ymm1, %ymm1
```



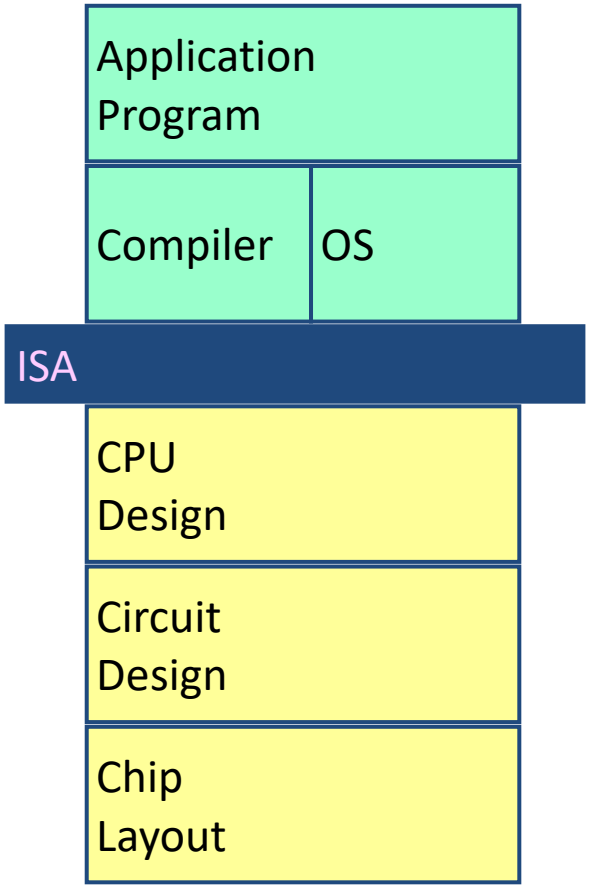
Getting High Performance

- Good compiler and flags
- Don’t do anything stupid
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers: procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

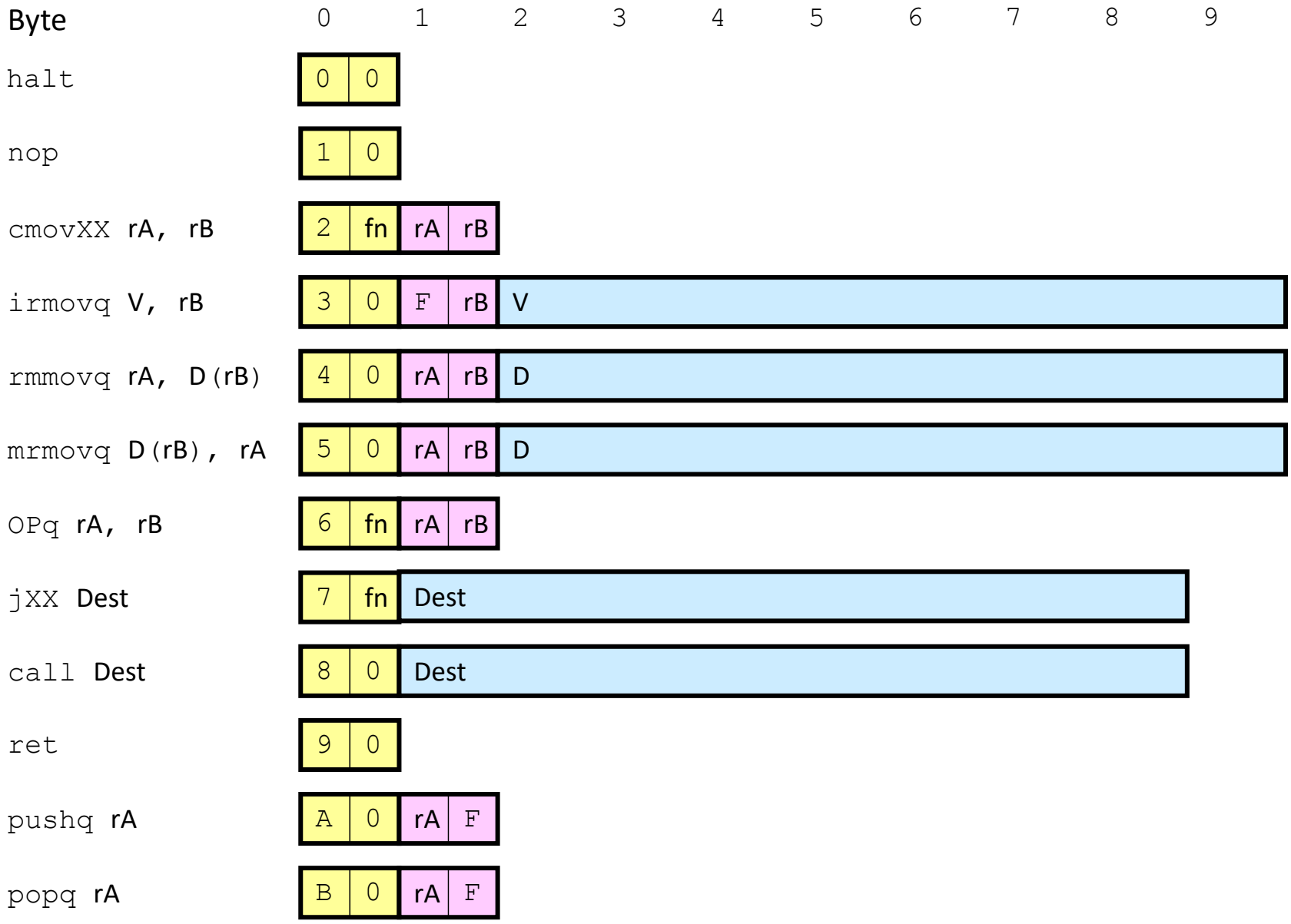
- Tune code for machine
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)

Instruction Set Architecture

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



Y86-64 Instruction Set #1



Y86-64 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

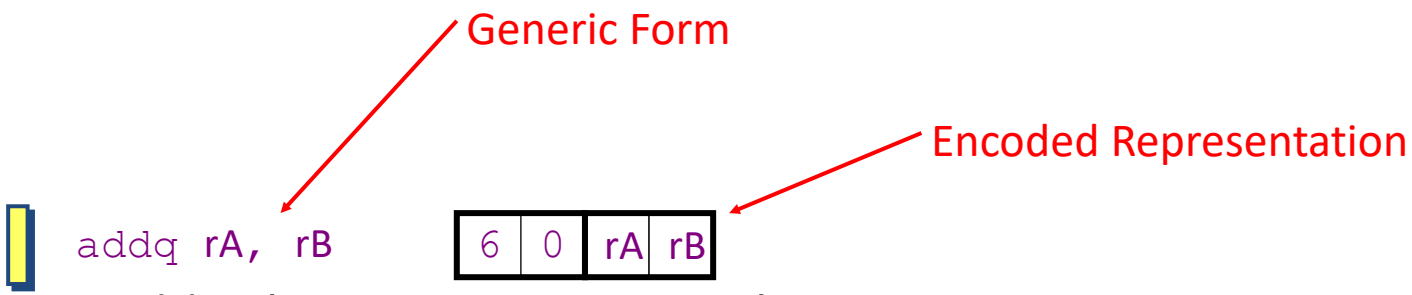
- Instruction encodings range between 1 and 10 bytes.
  - An instruction consists of a **1-byte instruction specifier, possibly a 1-byte register specifier, and possibly a 8-byte constant word**.

Instruction Encoding

- Each instruction requires between 1 and 10 bytes, depending on which fields are required.
- Every instruction has an initial byte identifying the instruction type. This byte is split into two 4-bit parts: the high-order, or code, part, and the low-order, or function, part.

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

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

CISC Instruction Sets

- Complex Instruction Set Computer
- IA32 is example

- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Arithmetic instructions can access memory
  - `addq %rax, 12(%rbx,%rcx,8)`
    - requires memory read and write
    - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions
- Philosophy
  - Add instructions to perform “typical” programming tasks

RISC Instruction Sets

  - Reduced Instruction Set Computer
  - Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)
- Fewer, simpler instructions
  - Might take more to get given task done
  - Can execute them with small and fast hardware
- Register-oriented instruction set
  - Many more (typically 32) registers
  - Use for arguments, return pointer, temporaries
- Only load and store instructions can access memory
  - Similar to Y86-64 `mrmovq` and `rmmovq`
- No Condition codes
  - Test instructions return 0/1 in register

CISC	Early RISC
A large number of instructions. The Intel document describing the complete set of instructions [28, 29] is over 1200 pages long.	Many fewer instructions. Typically less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
Variable-length encodings. IA32 instructions can range from 1 to 15 bytes.	Fixed-length encodings. Typically all instructions are encoded as 4 bytes.
Multiple formats for specifying operands. In IA32, a memory operand specifier can have many different combinations of displacement, base and index registers, and scale factors.	Simple addressing formats. Typically just base and displacement addressing.

Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by *load* instructions, reading from memory into a register, and *store* instructions, writing from a register to memory. This convention is referred to as a *load/store architecture*.
Implementation artifacts hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation artifacts exposed to machine-level programs. Some RISC machines prohibit particular instruction sequences and have jumps that do not take effect until the following instruction is executed. The compiler is given the task of optimizing performance within these constraints.
Condition codes. Special flags are set as a side effect of instructions and then used for conditional branch testing.	No condition codes. Instead, explicit test instructions store the test results in normal registers for use in conditional evaluation.
Stack-intensive procedure linkage. The stack is used for procedure arguments and return addresses.	Register-intensive procedure linkage. Registers are used for procedure arguments and return addresses. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.

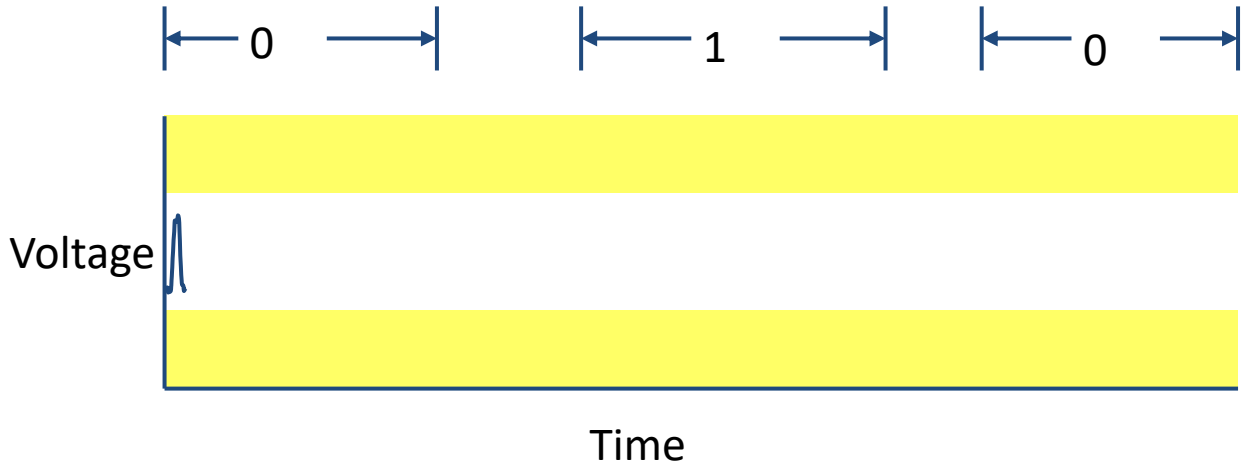
CISC vs. RISC

- Original Debate
  - Strong opinions!
  - CISC proponents---easy for compiler, fewer code bytes
  - RISC proponents---better for optimizing compilers, can make run fast with simple chip design
- Current Status
  - For desktop processors, choice of ISA not a technical issue
    - With enough hardware, can make anything run fast
    - Code compatibility more important
  - x86-64 adopted many RISC features
    - More registers; use them for argument passing
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power
    - Most cell phones use ARM processor

Overview of Logic Design

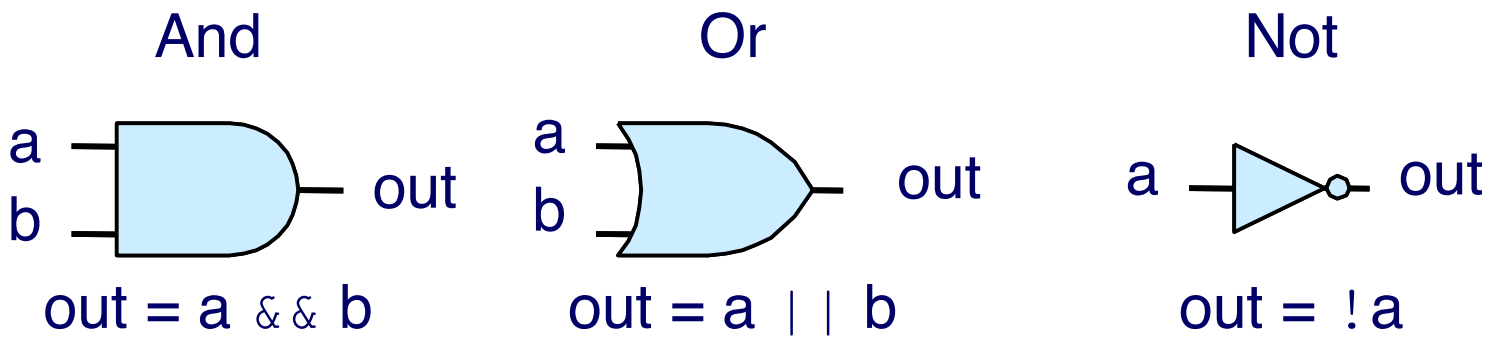
- Fundamental Hardware Requirements
  - Communication
    - How to get values from one place to another
  - Computation
  - Storage
- Bits are Our Friends
  - Everything expressed in terms of values 0 and 1
  - Communication
    - Low or high voltage on wire
  - Computation
    - Compute Boolean functions
  - Storage
    - Store bits of information

Digital Signals

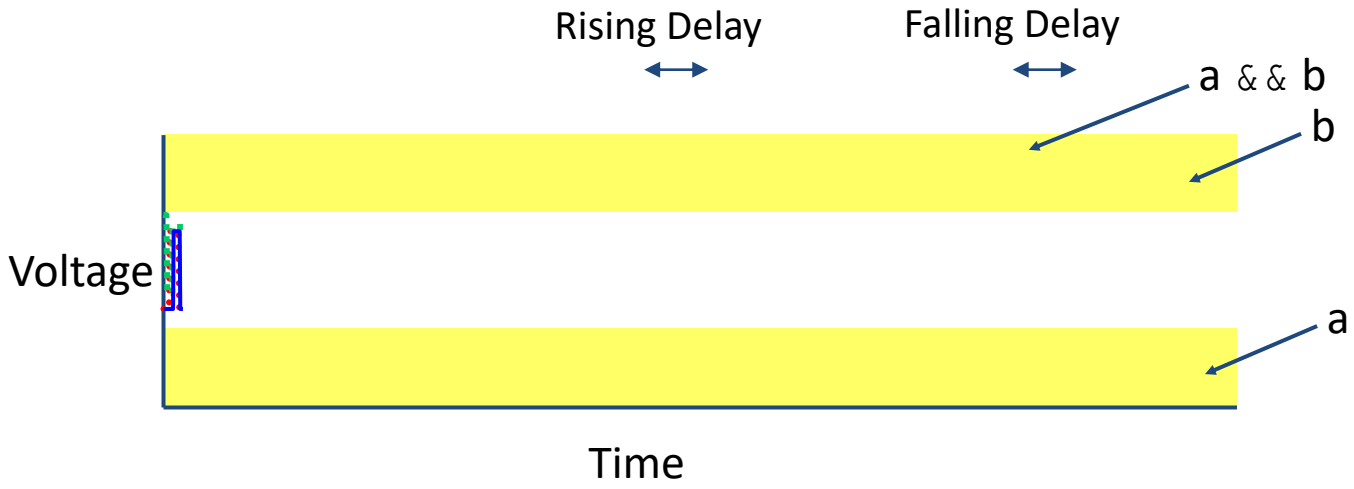


- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

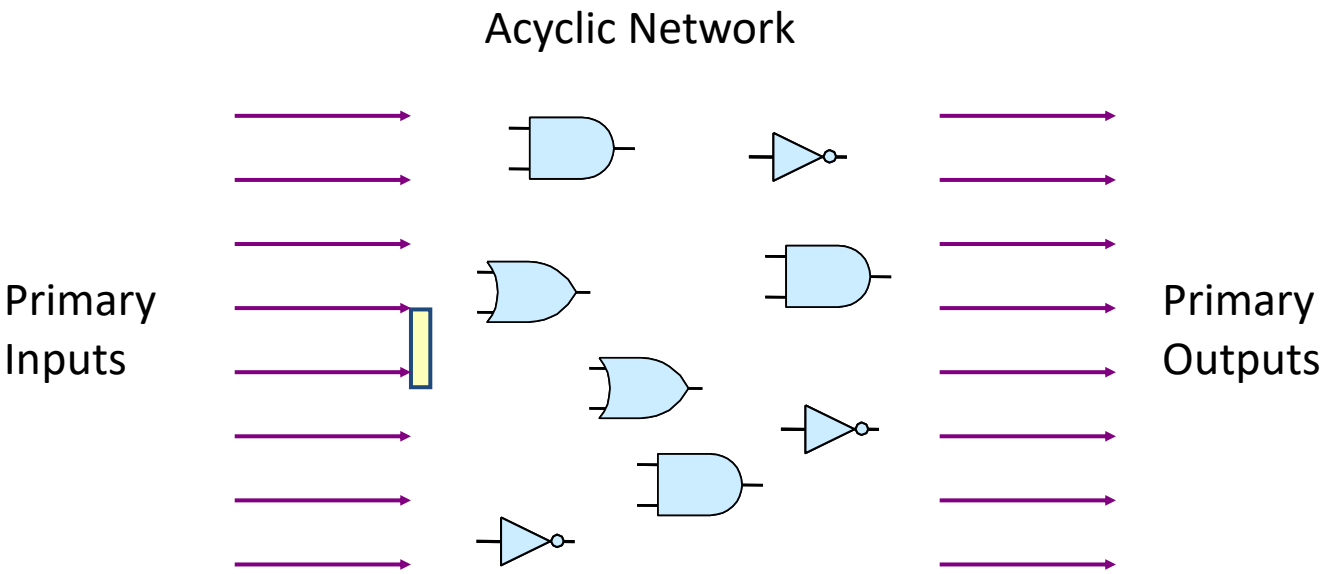
Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
  - With some, small delay

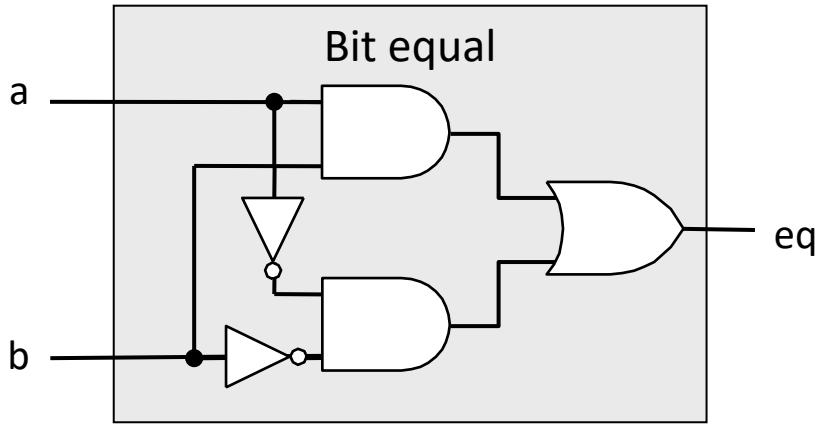


Combinational Circuits



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

Bit Equality

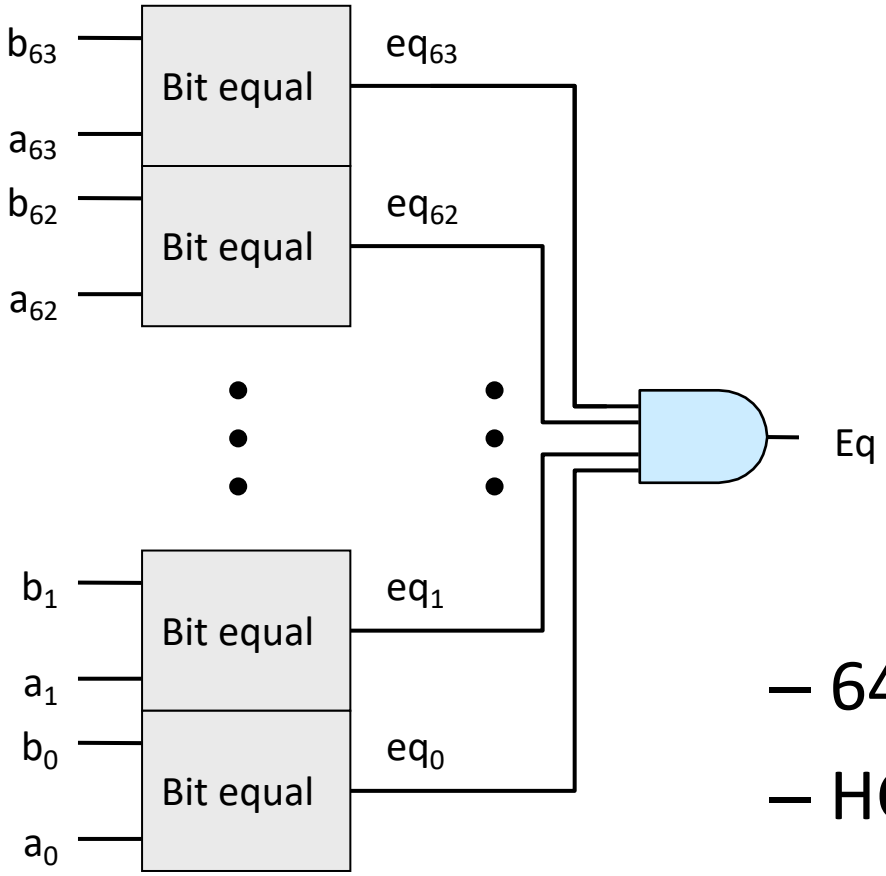


HCL Expression

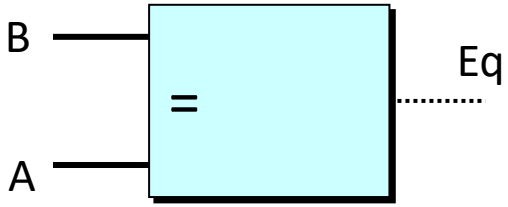
```
bool eq = (a&&b)||(!a&&!b)
```

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

Word Equality



Word-Level Representation

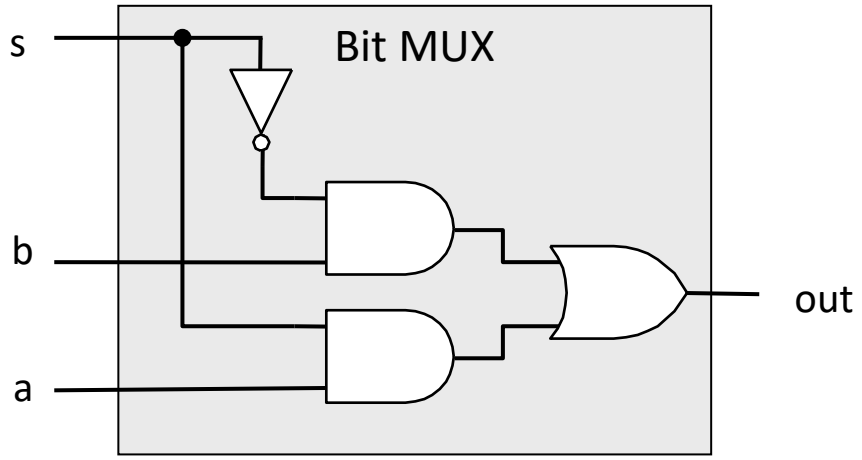


HCL Representation

```
bool Eq = (A == B)
```

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

Bit-Level Multiplexor

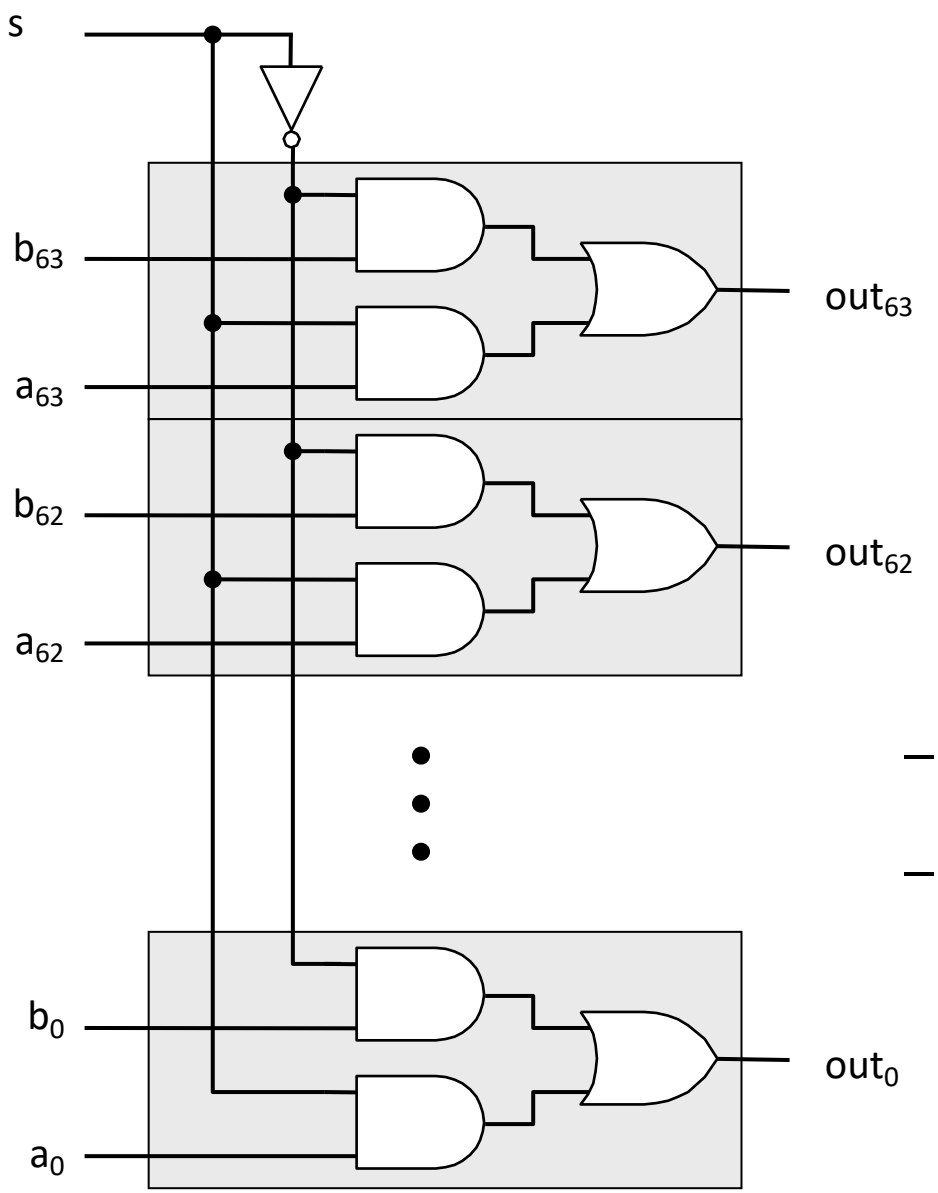


HCL Expression

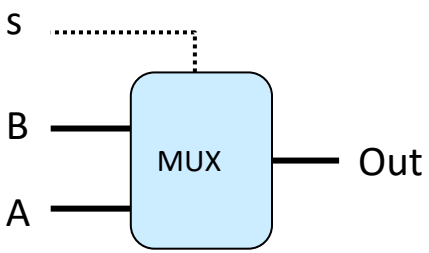
```
bool out = (s&&a)||(!s&&b)
```

- Control signal s
- Data signals a and b
- Output a when s=1, b when s=0

Word Multiplexor



Word-Level Representation

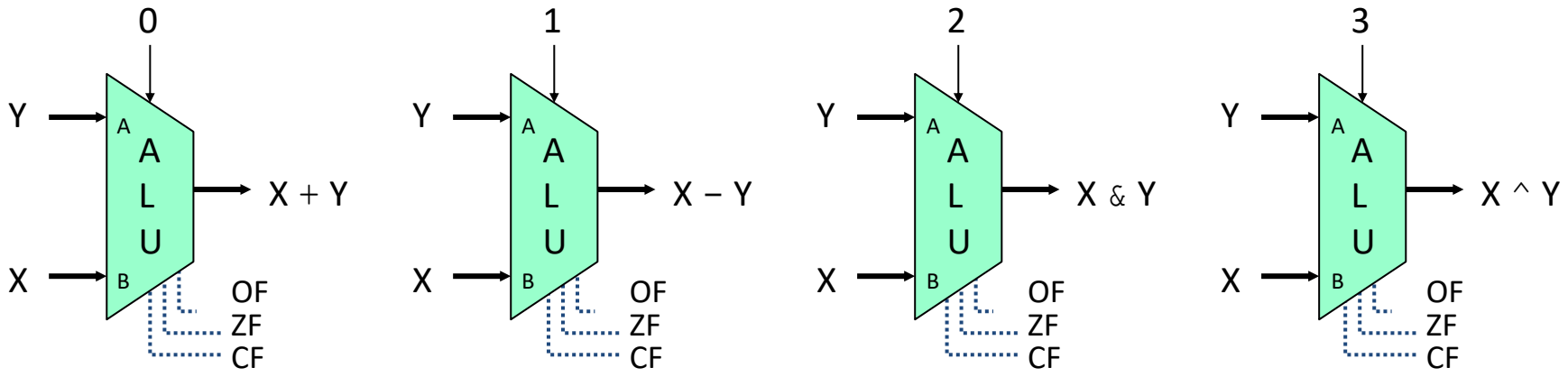


HCL Representation

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

- Select input word A or B depending on control signal s
- HCL representation
  - Case expression
  - Series of test : value pairs
  - Output value for first successful test

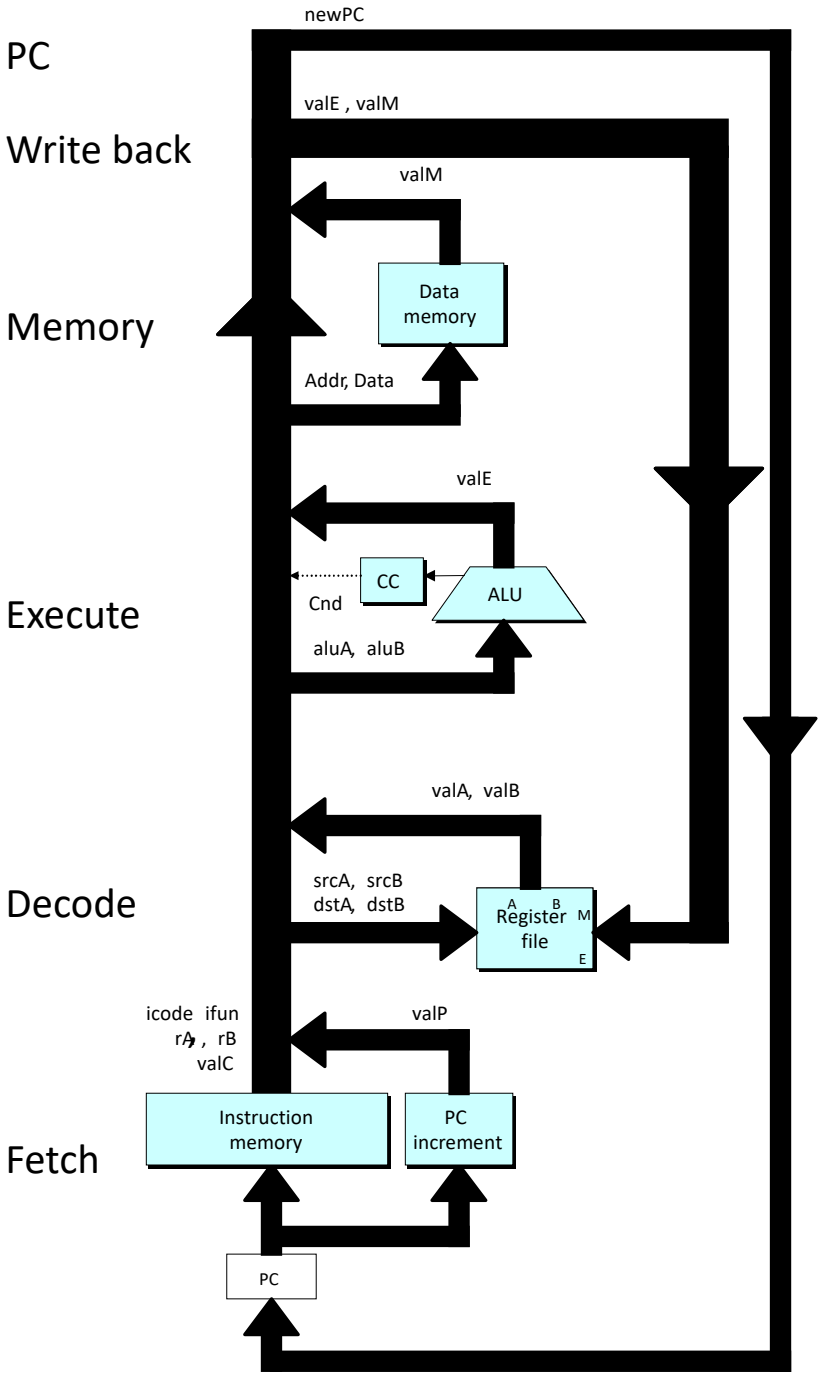
Arithmetic Logic Unit



- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

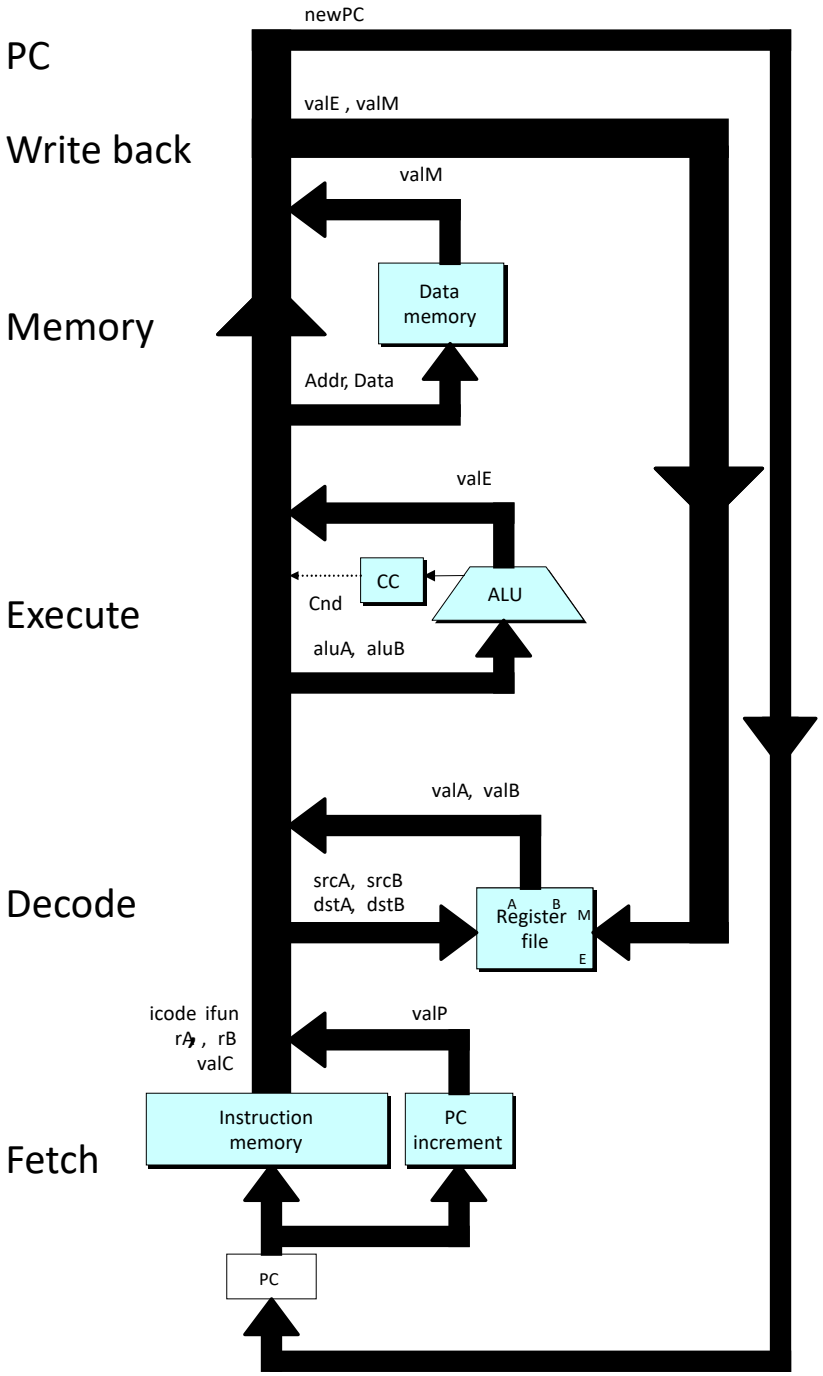
SEQ Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter



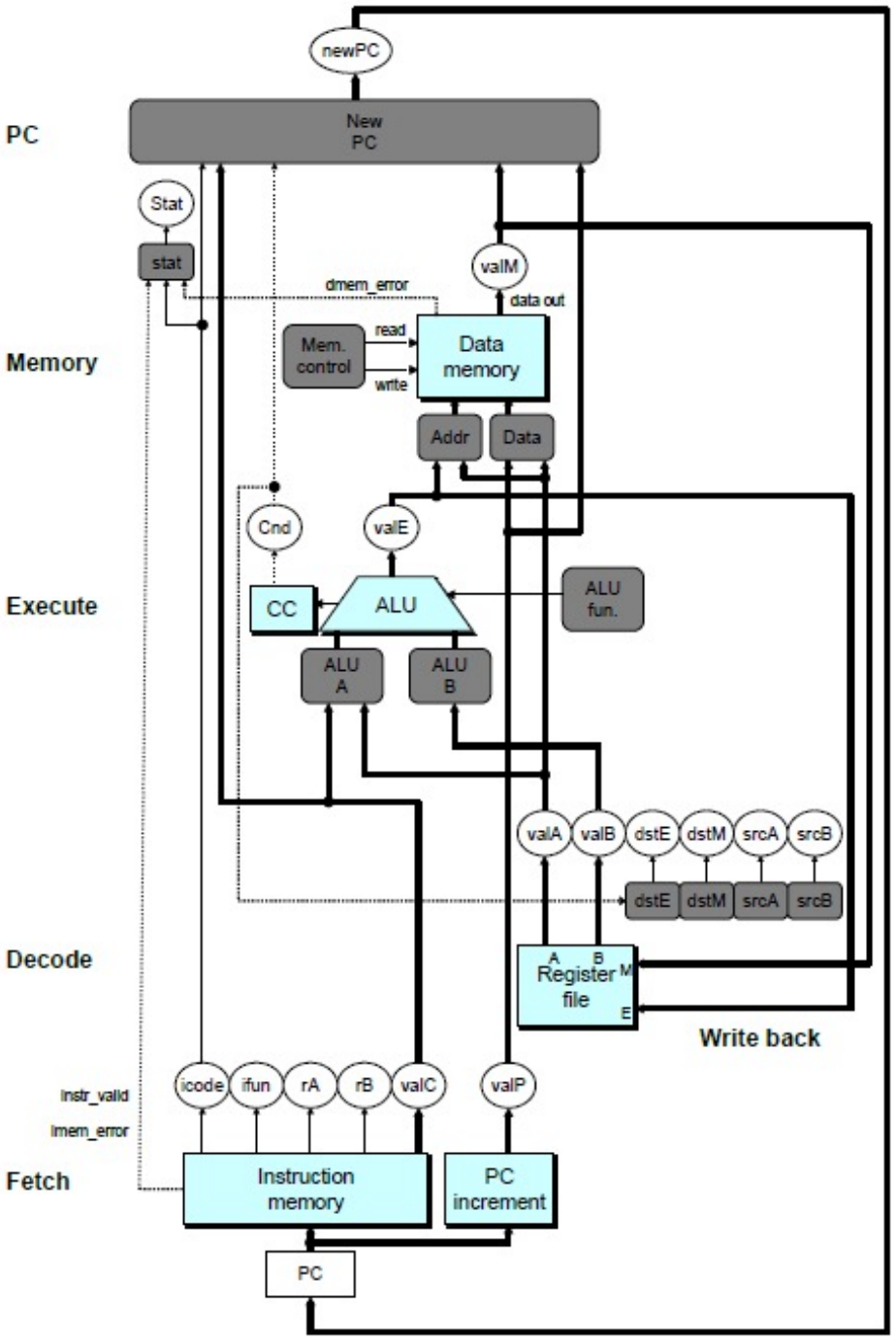
SEQ Stages

- 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 Hardware

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



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

Real-World Pipelines: Car Washes

Sequential

Parallel

Pipelined

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

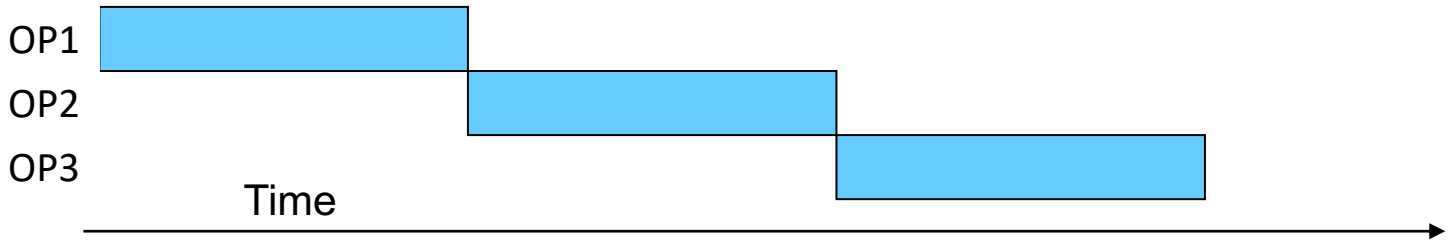
Pipelining

Pipelining is running multiple *stages* of the same *process* in parallel in a way that efficiently uses all the available hardware while respecting the dependencies of each stage upon the previous stages.
Latency is the time to complete a single task.
Throughput is the rate at which tasks complete.

http://15418.courses.cs.cmu.edu/spring2013/article/17

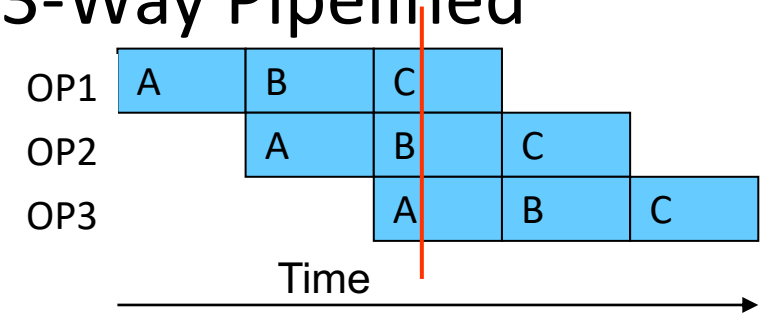
Pipeline Diagrams

- Unpipelined



  - Cannot start new operation until previous one completes

- 3-Way Pipelined



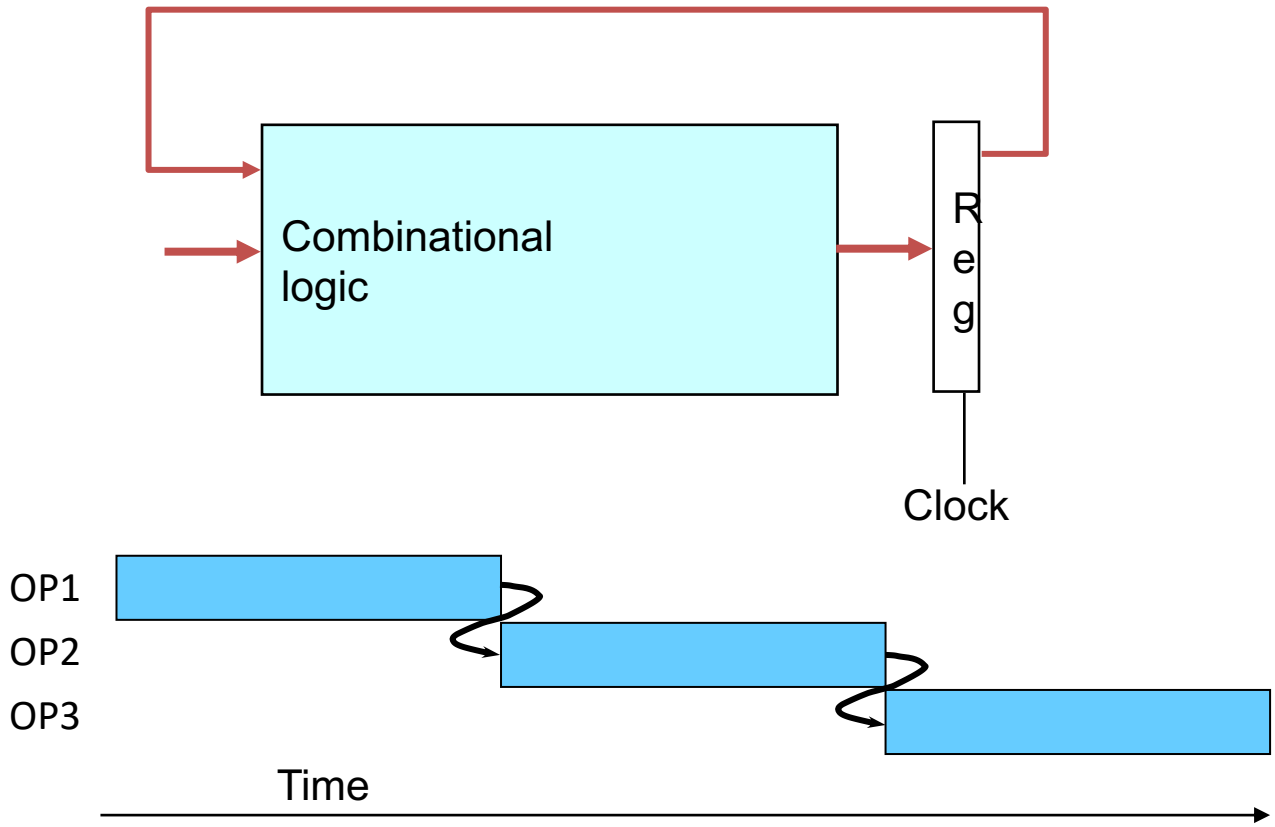
  - Up to 3 operations in process simultaneously

Hazards

- There are three classes of hazards:
- **Structural Hazards.** They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- **Data Hazards.** They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- **Control Hazards.** They arise from the pipelining of branches and other instructions that change the PC.

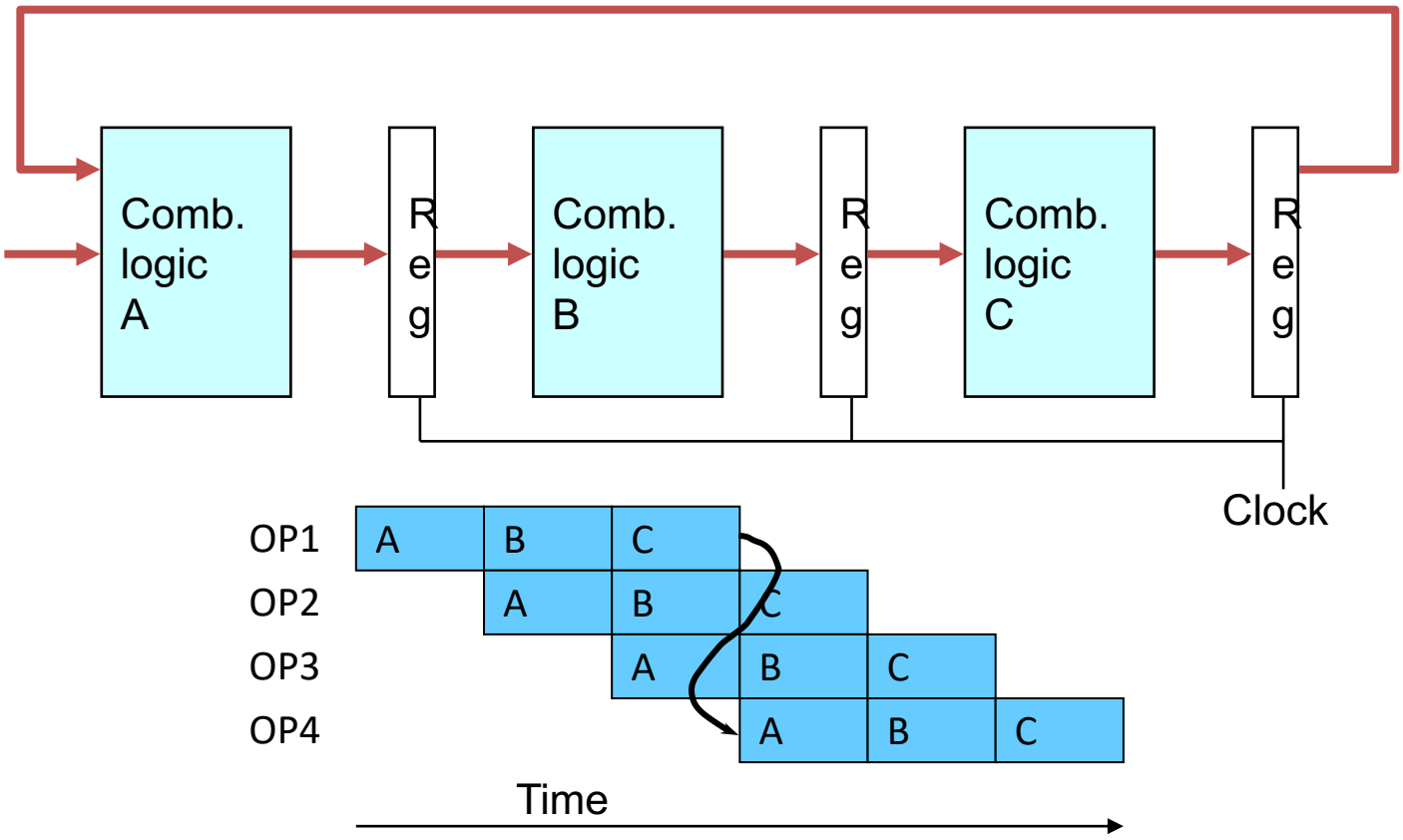
http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/hazards.html

Data Dependencies



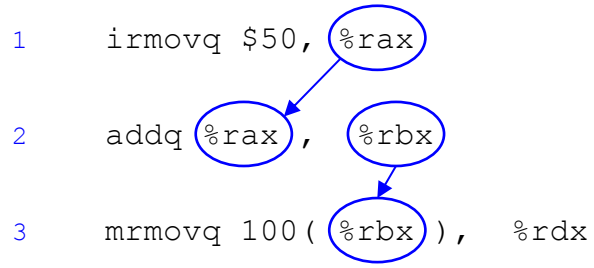
- System
  - Each operation depends on result from preceding one

Data Hazards



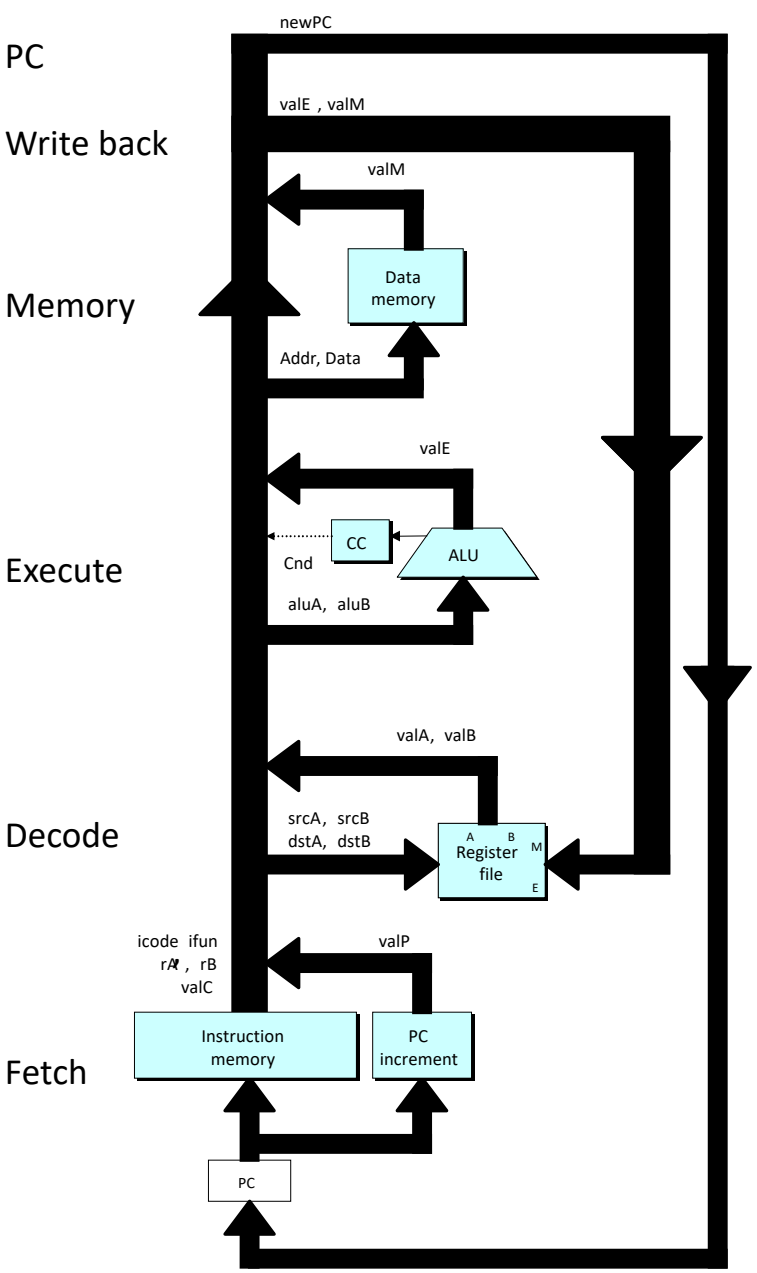
- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

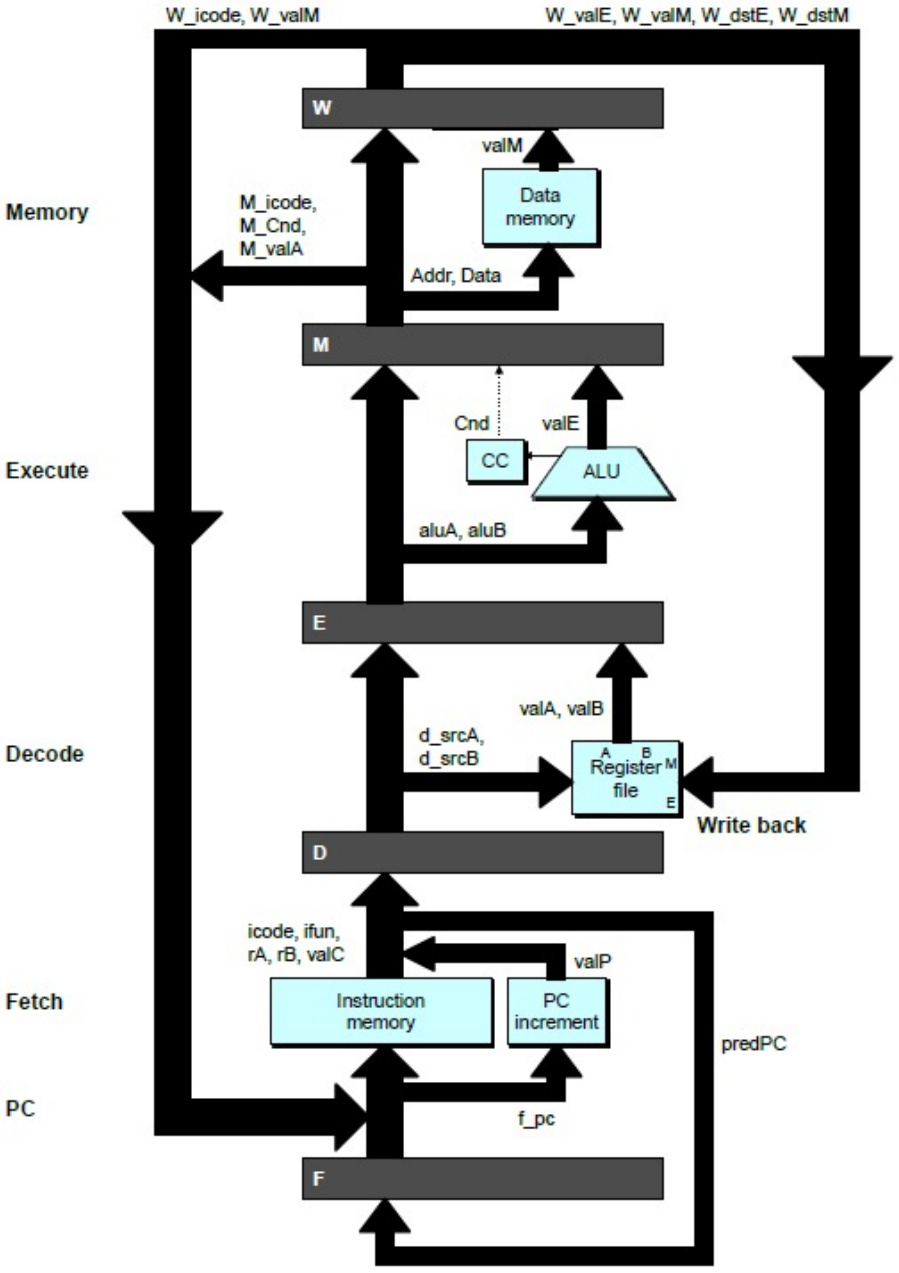
Data Dependencies in Processors



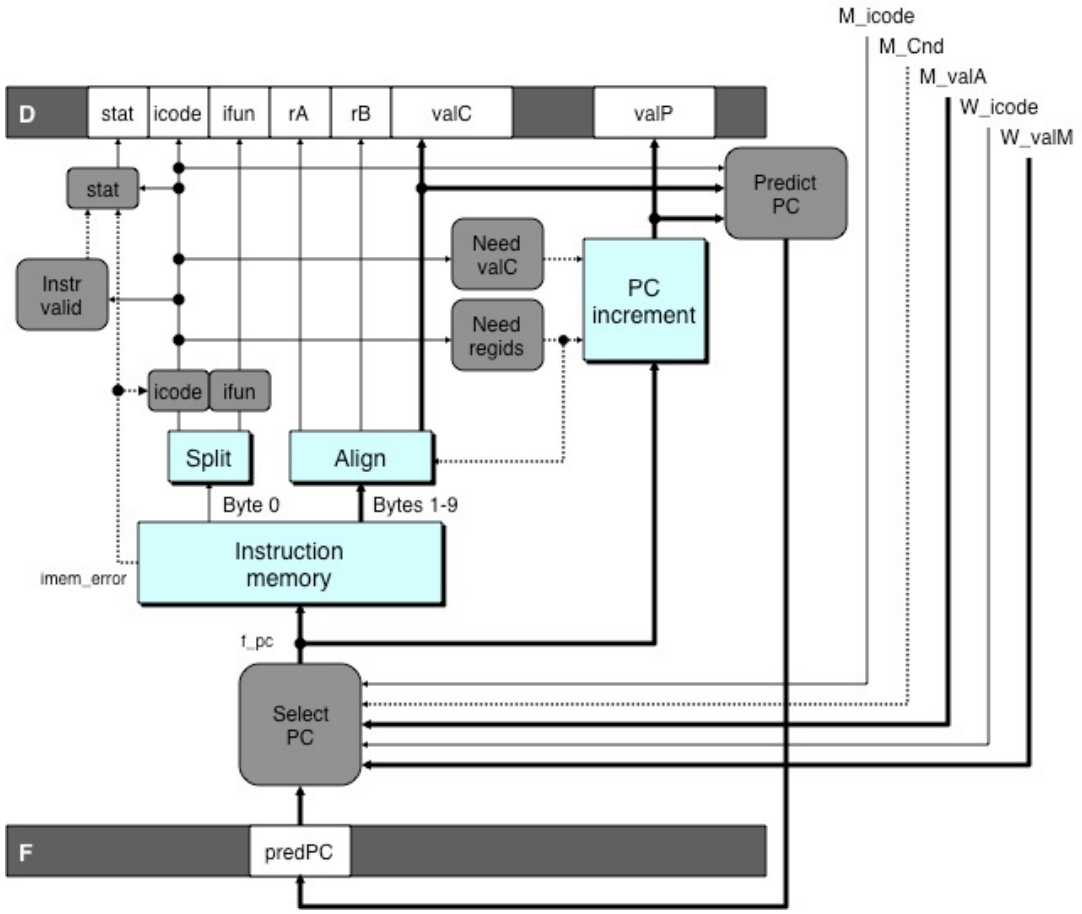
- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

Adding Pipeline Registers





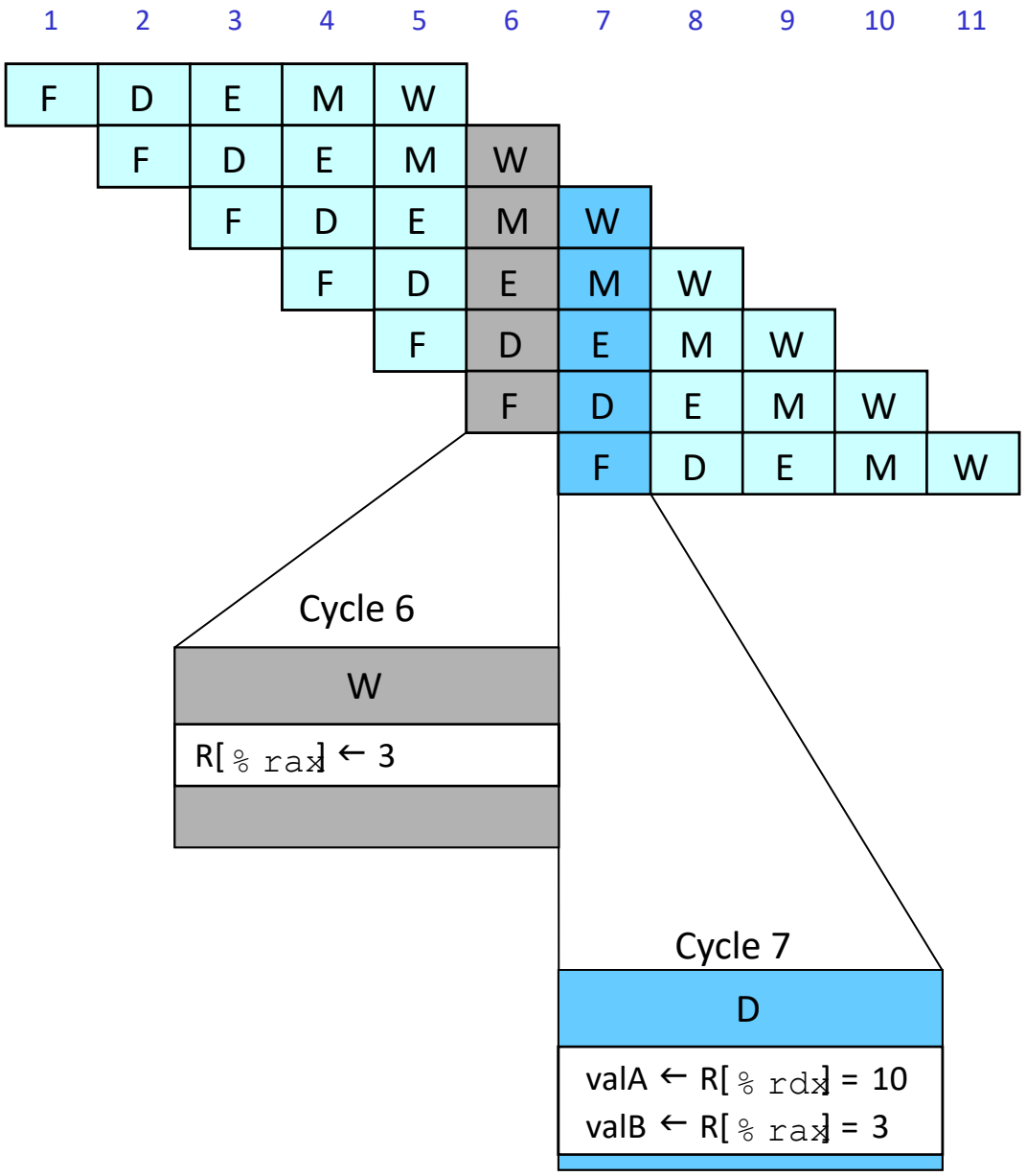
Predicting the PC



- 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

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
```



Random-Access Memory (RAM)

- Key features
  - RAM is traditionally packaged as a chip.
  - Basic storage unit is normally a cell (one bit per cell).
  - Multiple RAM chips form a memory.

- RAM comes in two varieties:
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)

SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

- **SRAM**
  - More complex than Dram, require 4 to 6 transistor per bit
  - More expensive (each cell is more complex)
  - Faster than DRAM
  - Used in small – fast memories
  - Stores each bit in bistable memory cell(each cell is implemented with a six-transistor ciruit)
- **DRAM**
  - Simple, Slower, Cheaper
  - Needs to be refreshed constantly
  - Used in main memories and frame buffers associated with the graphic cards
  - Stores each bit as charge on a capacitor

Nonvolatile Memories

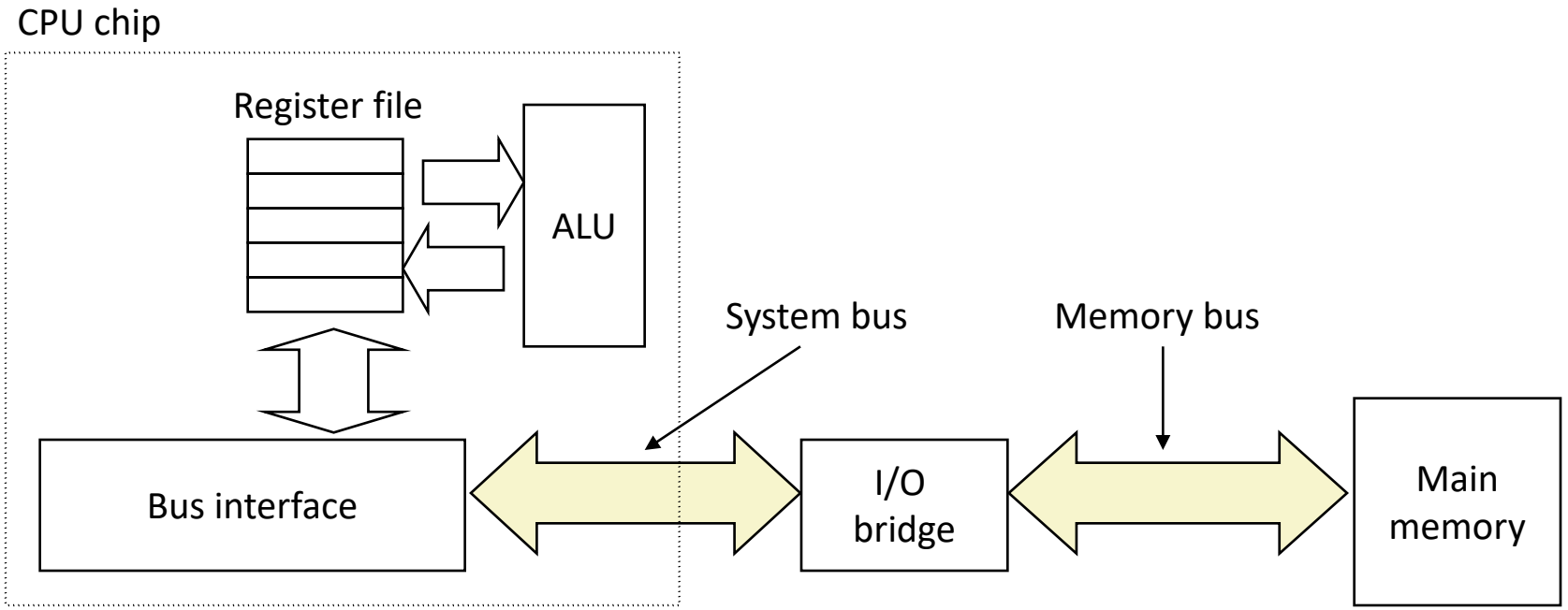
- DRAM and SRAM are **volatile** memories
  - Lose information if powered off.
- **Nonvolatile** memories retain value even if powered off
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Erasable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically erasable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 erasing

Nonvolatile Memories

- Uses for Nonvolatile Memories
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,…)
  - Disk caches

Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



What's Inside A Disk Drive?

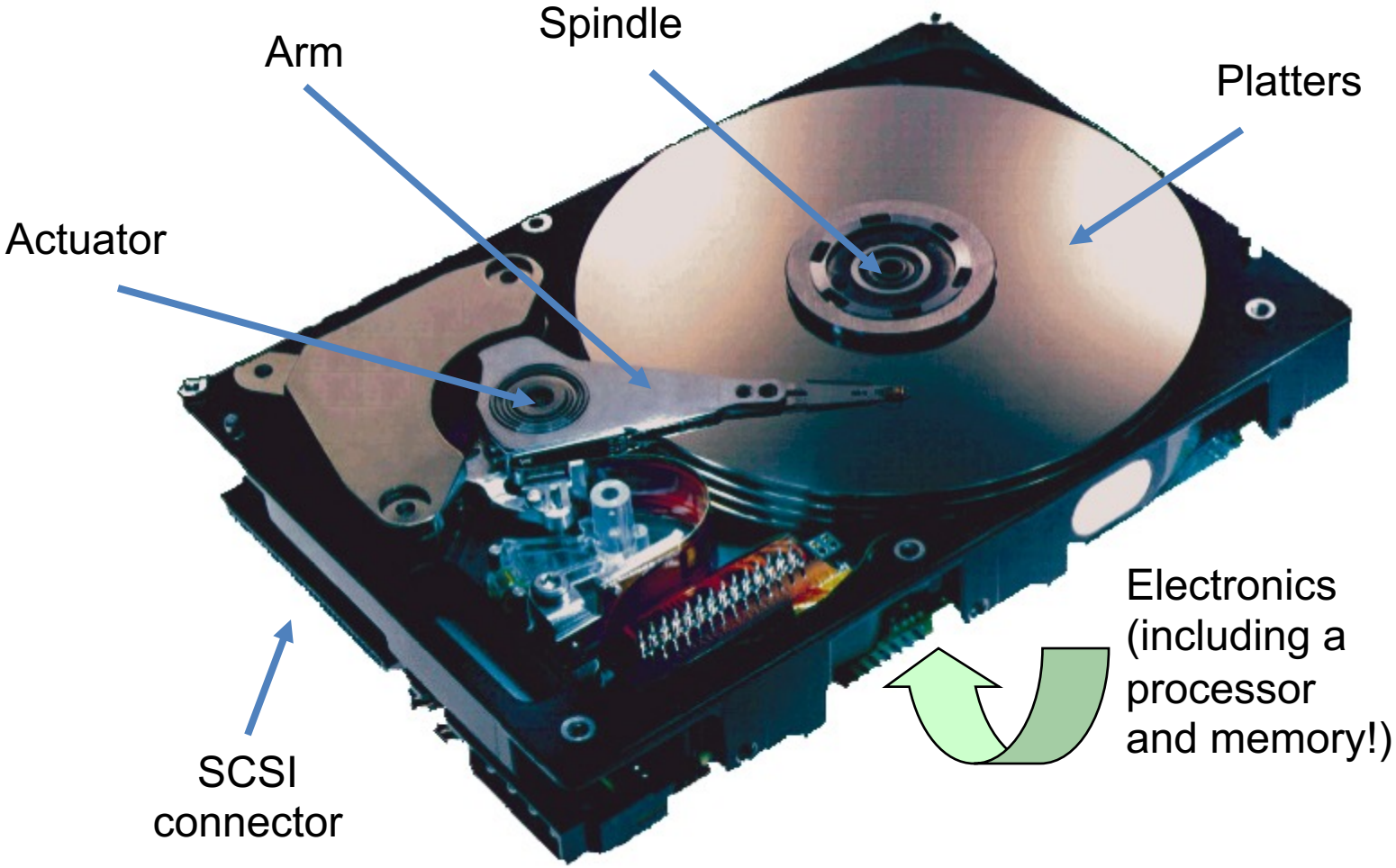
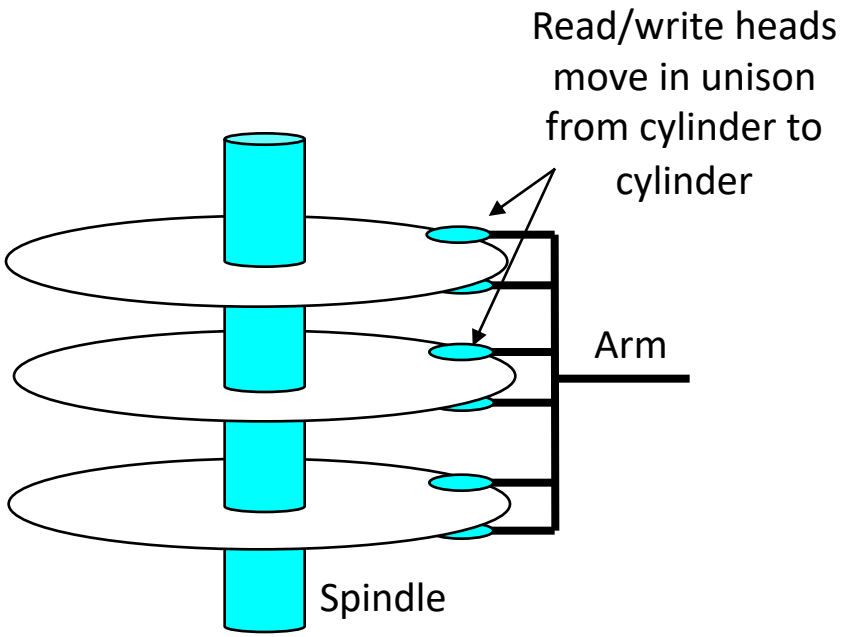
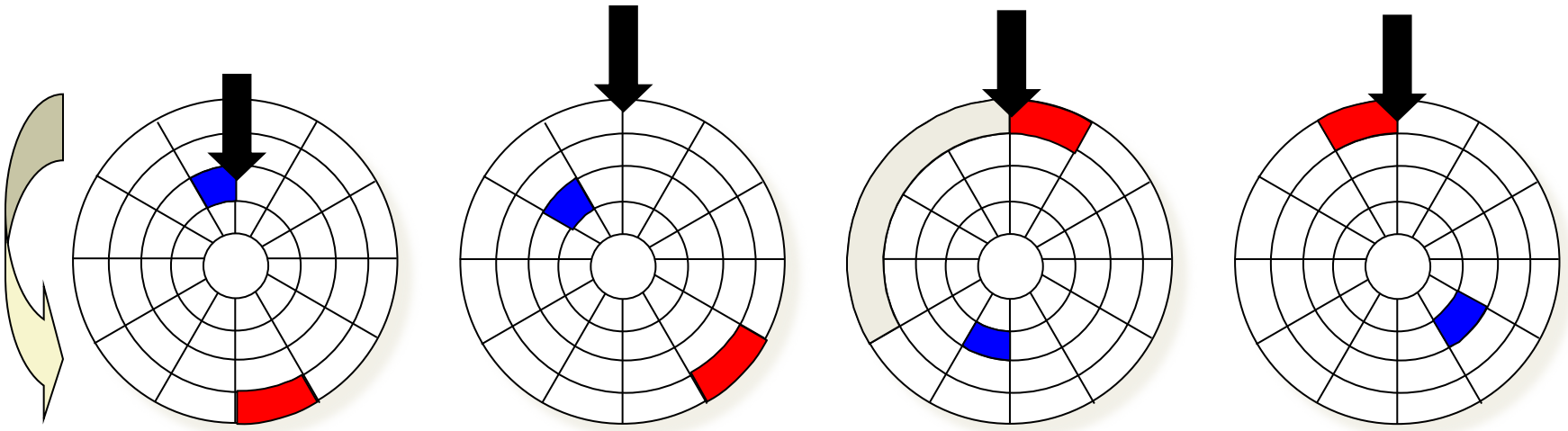


Image courtesy of Seagate Technology

Disk Operation (Multi-Platter View)



Disk Access – Service Time Components



After BLUE read

Seek for RED

Rotational latency

After RED read

Data transfer

Seek

Rotational latency

Data transfer

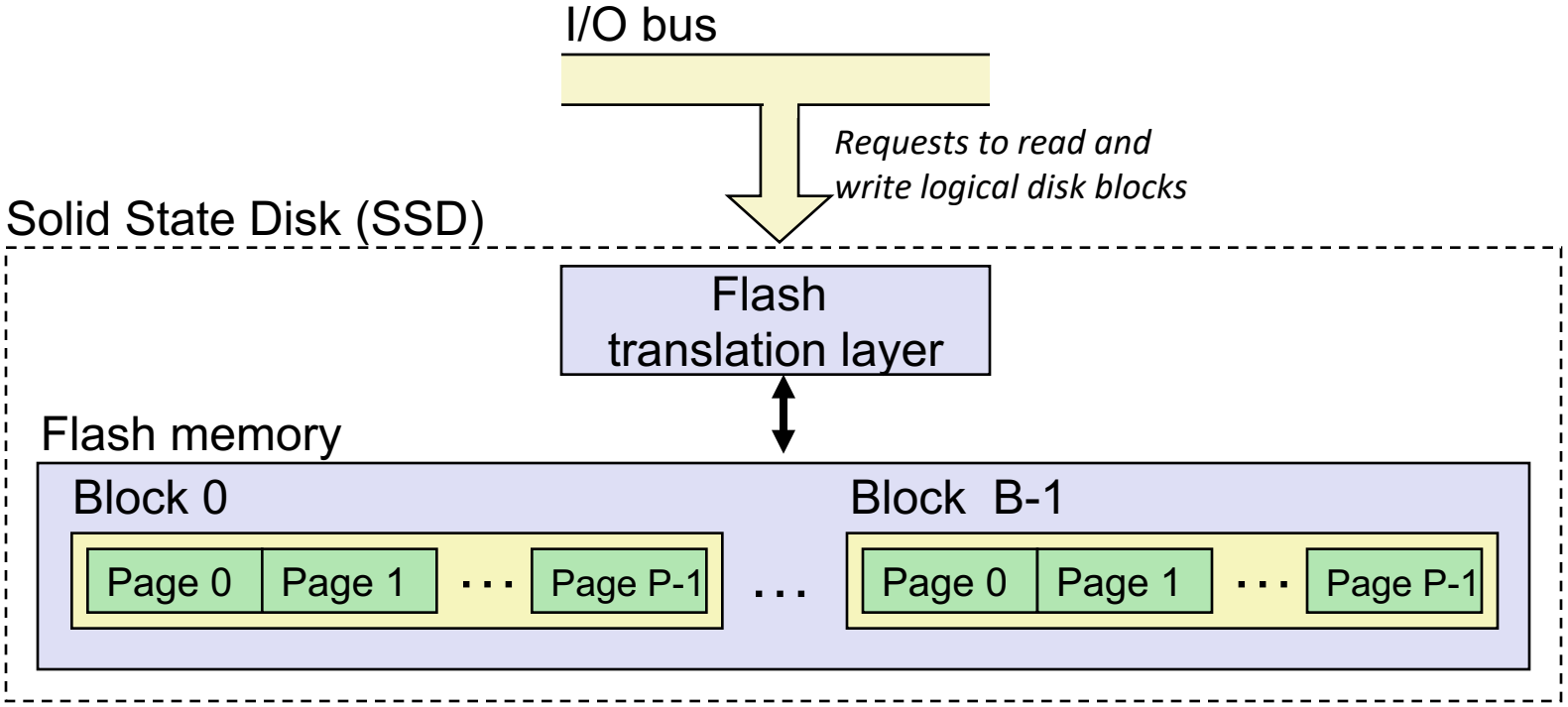
Disk Access Time

- **Average time to access some target sector approximated by :**
  - **Taccess = Tavg seek + Tavg rotation + Tavg transfer**
- Seek time (Tavg seek)
  - Time to position heads over cylinder containing target sector.
  - Typical Tavg seek is 3—9 ms
- Rotational latency (Tavg rotation)
  - Time waiting for first bit of target sector to pass under r/w head.
  - Tavg rotation = 1/2 x 1/RPMs x 60 sec/1 min
  - Typical Tavg rotation = 7200 RPMs
- Transfer time (Tavg transfer)
  - Time to read the bits in the target sector.
  - Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min.

Disk Access Time Example

- Given:
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.
- Derived:
  - Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM x 1/400 secs/track x 1000 ms/sec = 0.02 ms
  - Taccess = 9 ms + 4 ms + 0.02 ms
- **Important points:**
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - **Disk is about 40,000 times slower than SRAM,**
    - **2,500 times slower then DRAM.**

Solid State Disks (SSDs)



- Pages: 512KB to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.

SSD Performance Characteristics

Sequential read tput	550 MB/s	Sequential write tput	470 MB/s
Random read tput	365 MB/s	Random write tput	303 MB/s
Avg seq read time	50 us	Avg seq write time	60 us

- Sequential access faster than random access
  - Common theme in the memory hierarchy
- Random writes are somewhat slower
  - Erasing a block takes a long time (~1 ms)
  - Modifying a block page requires all other pages to be copied to new block
  - In earlier SSDs, the read/write gap was much larger.

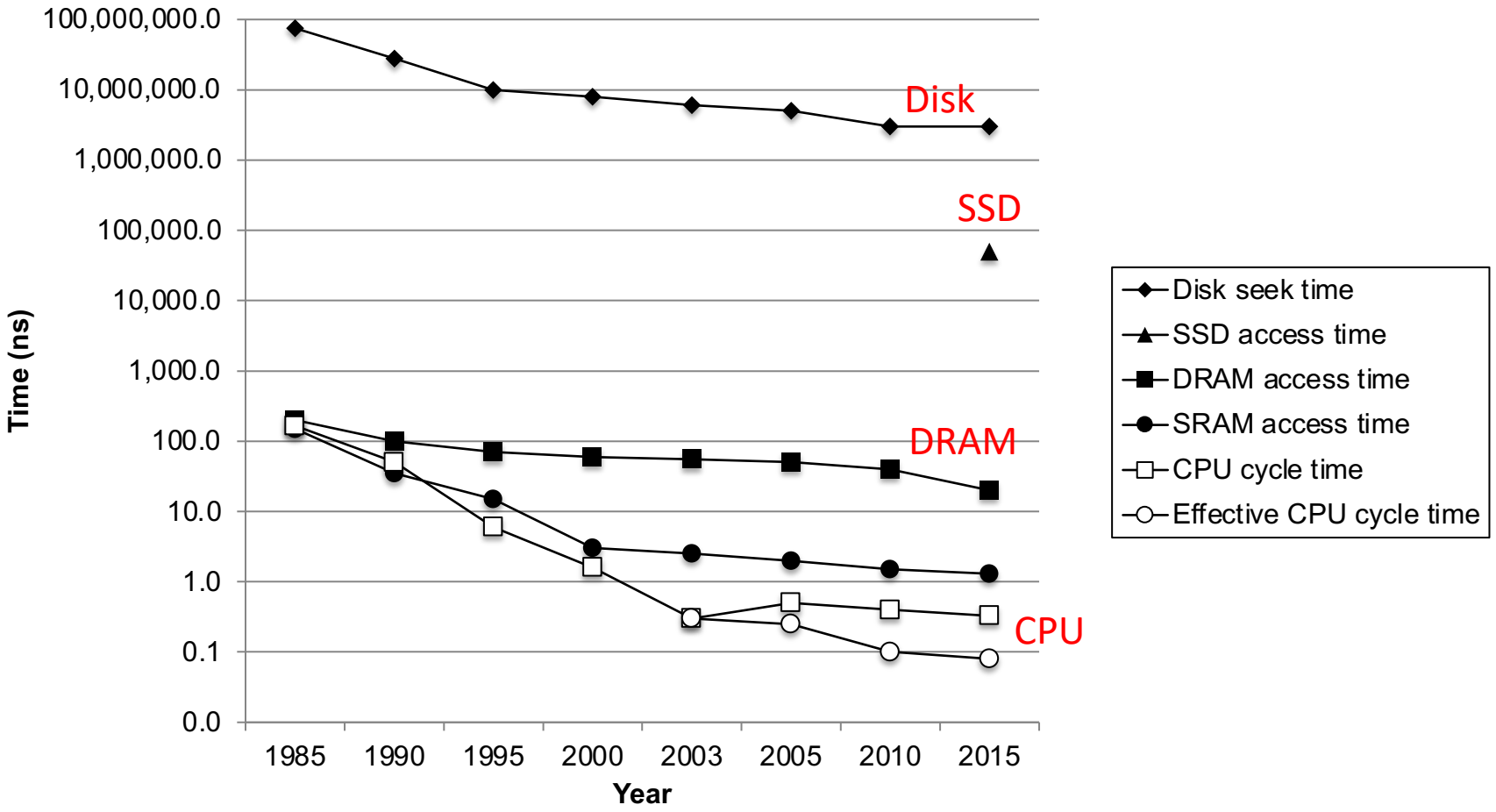
Source: Intel SSD 730 product specification.

SSD Tradeoffs vs Rotating Disks

- Advantages
  - No moving parts → faster, less power, more rugged
- Disadvantages
  - Have the potential to wear out
    - Mitigated by "wear leveling logic" in flash translation layer
    - E.g. Intel SSD 730 guarantees 128 petabyte (128×10^{15} bytes) of writes before they wear out
  - In 2015, about 30 times more expensive per byte
- Applications
  - MP3 players, smart phones, laptops
  - Beginning to appear in desktops and servers

The CPU-Memory Gap

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

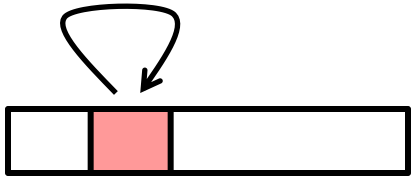


Locality to the Rescue!

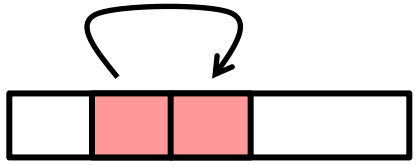
The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality

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



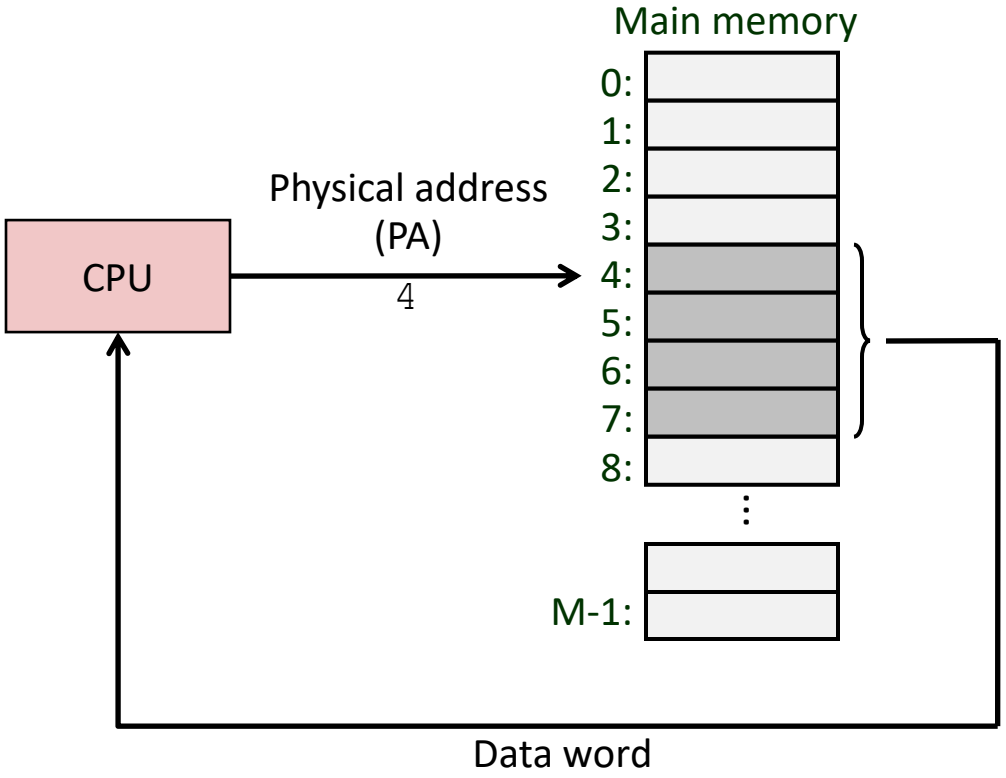
- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

Locality Example

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

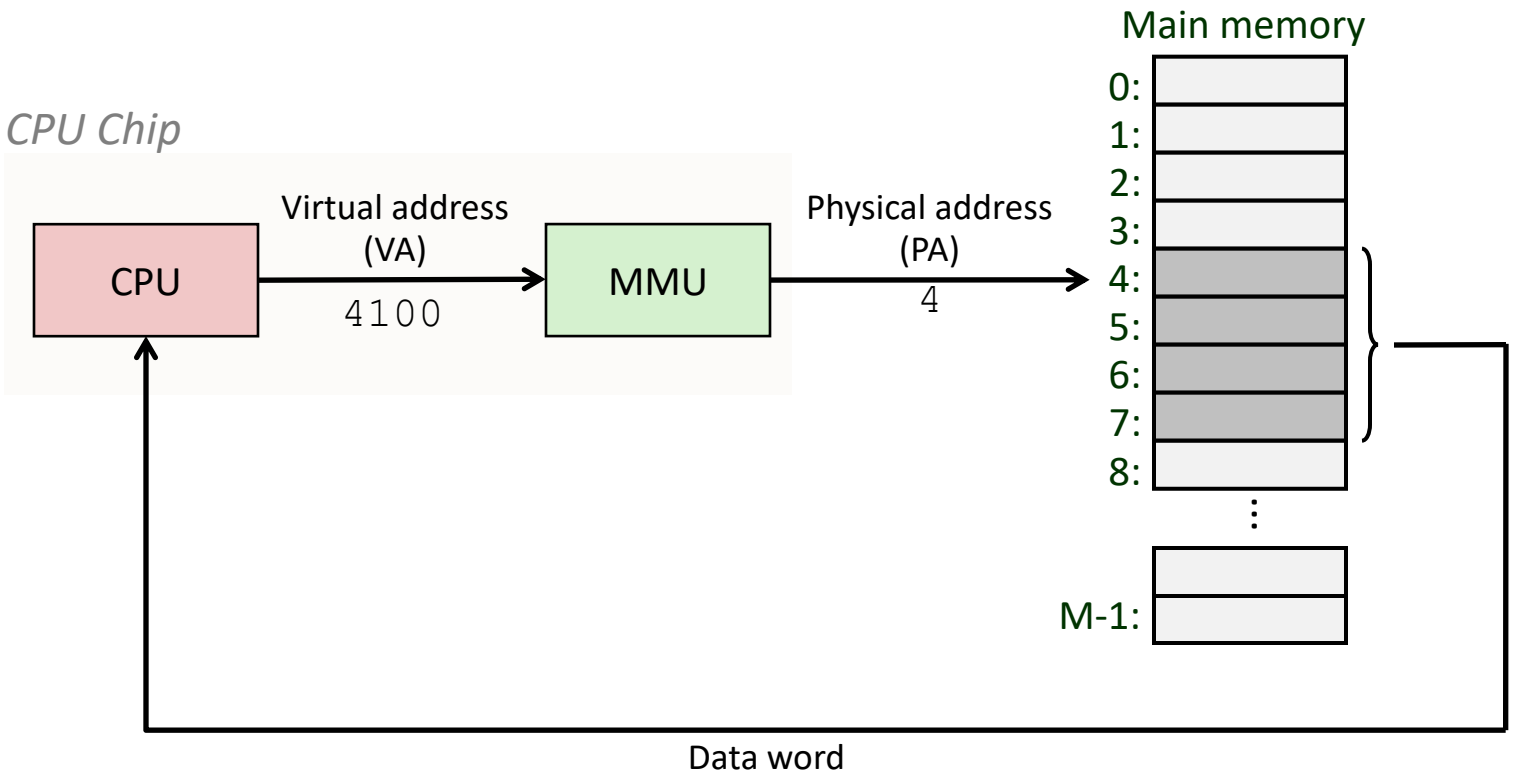
- Data references
  - Reference array elements in succession (stride-1 reference pattern). — Spatial locality
  - Reference variable `sum` each iteration. — Temporal locality
- Instruction references
  - Reference instructions in sequence. — Spatial locality
  - Cycle through loop repeatedly. — Temporal locality

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



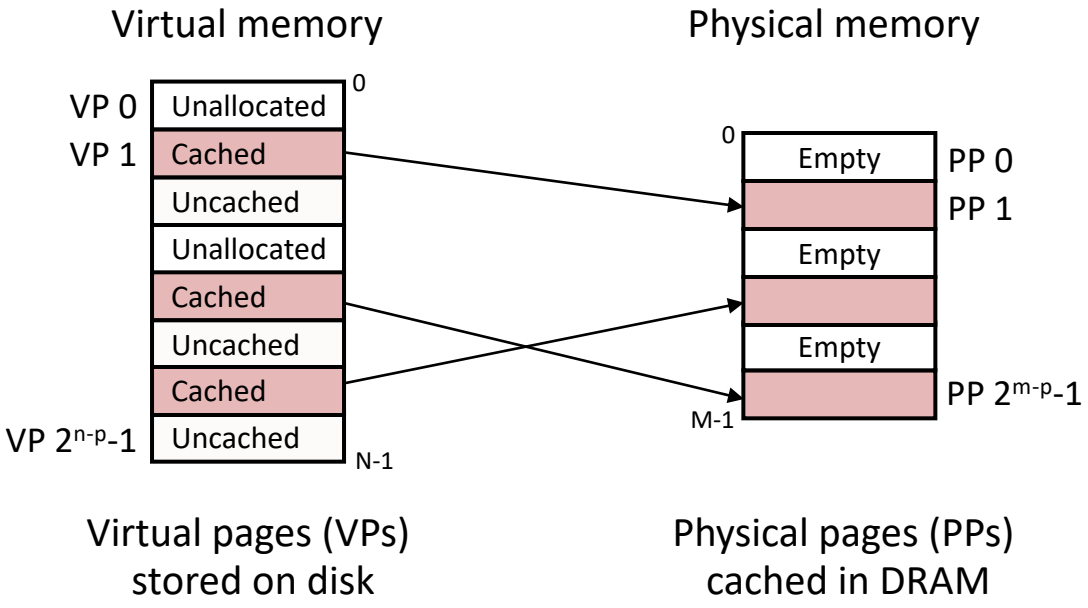
- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for 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 and code

VM as a Tool for Caching

- Conceptually, *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)



Locality to the Rescue Again!

- Virtual memory seems terribly inefficient, but it works because of locality.

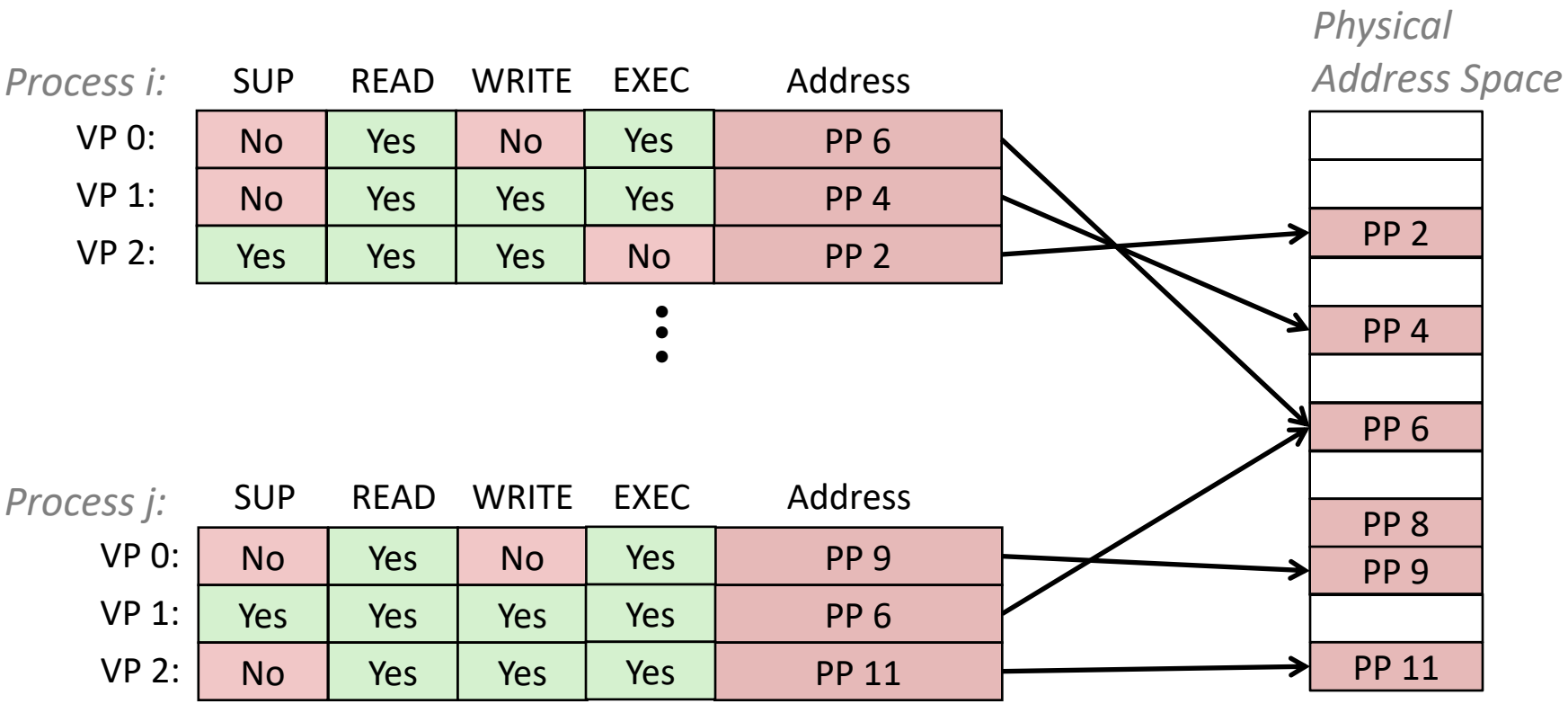
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets

- If (working set size < main memory size)
  - Good performance for one process after compulsory misses

- If (SUM(working set sizes) > main memory size)
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access



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

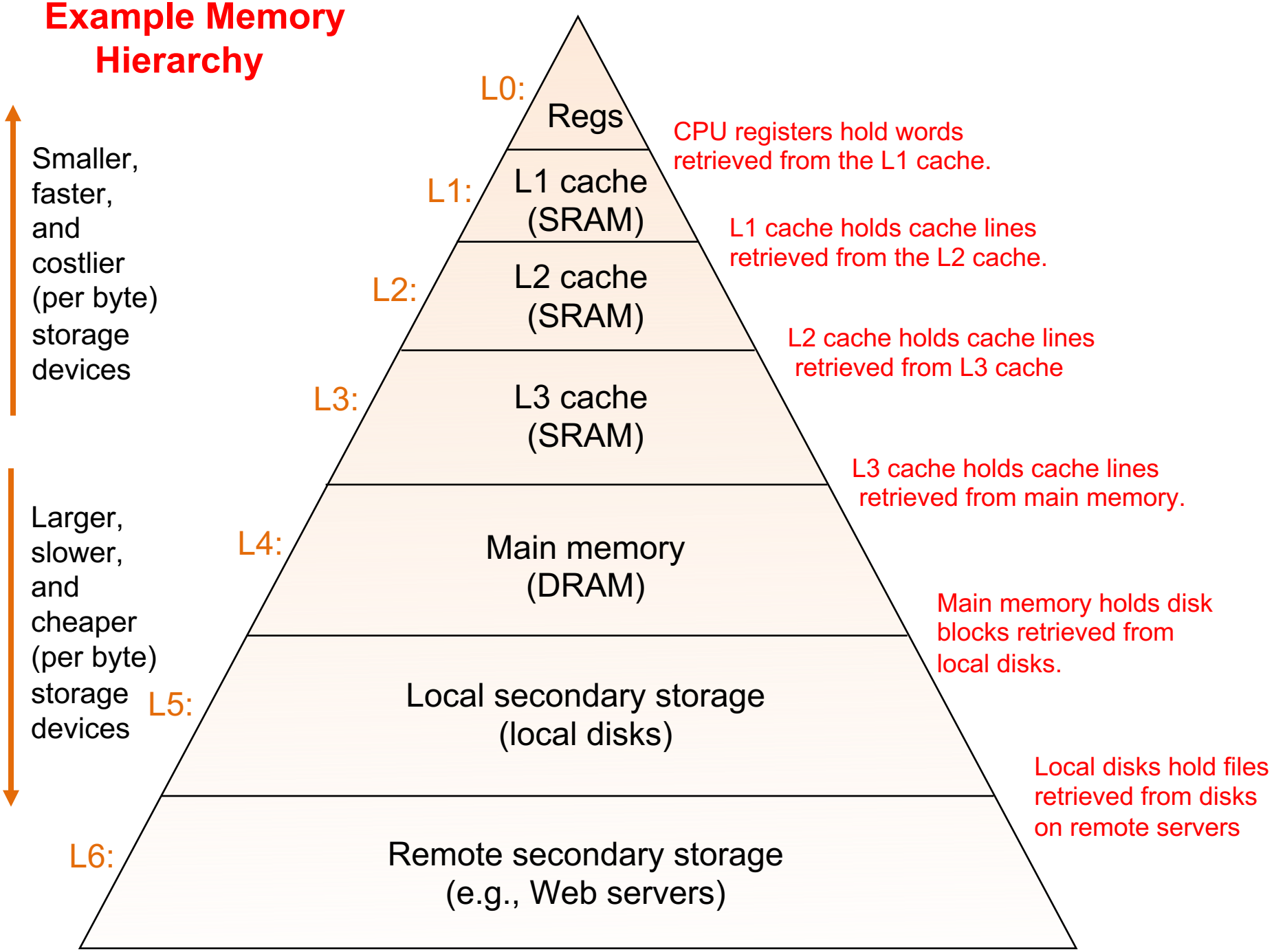
Memory Hierarchies

- 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.

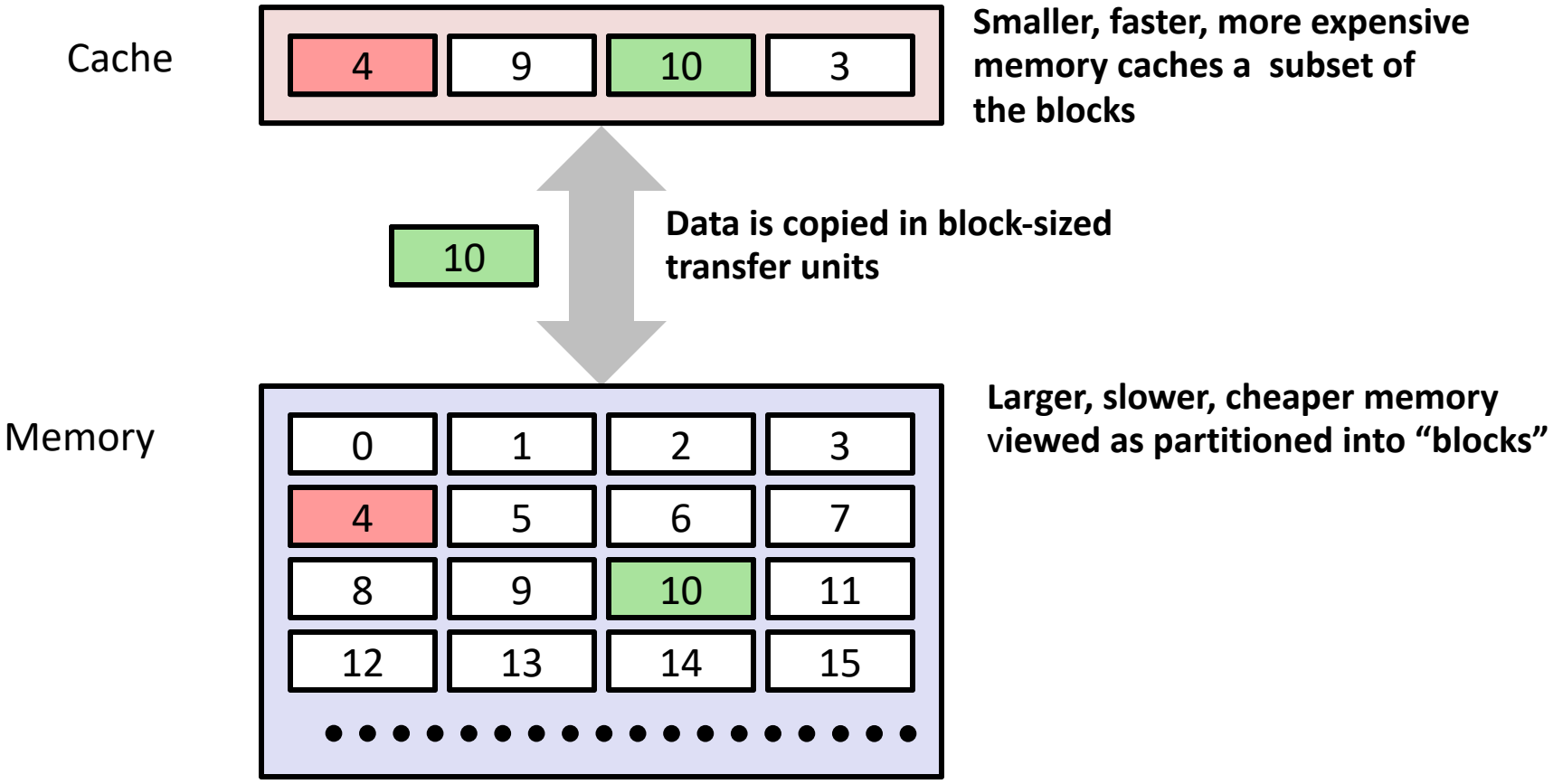
Example Memory Hierarchy



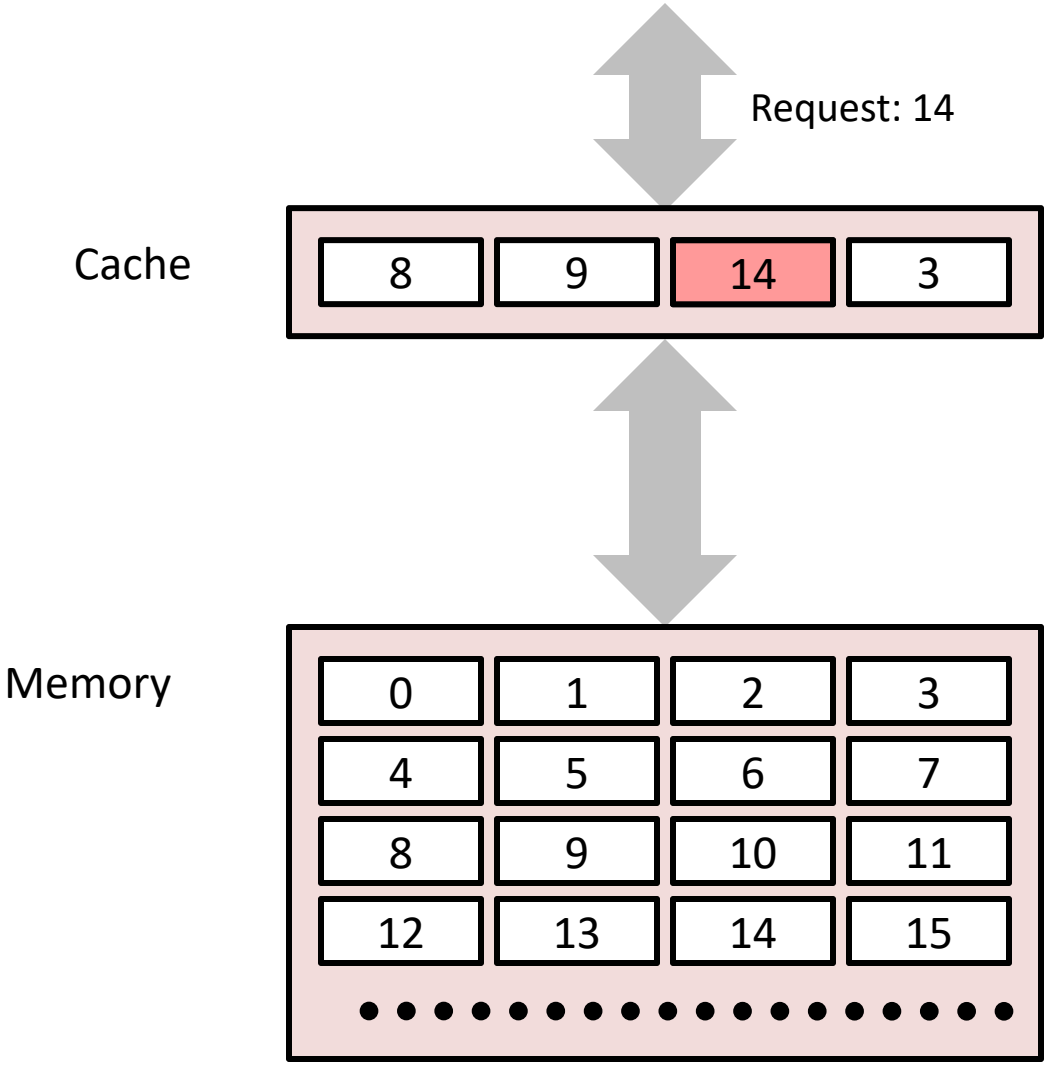
Caches

- *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



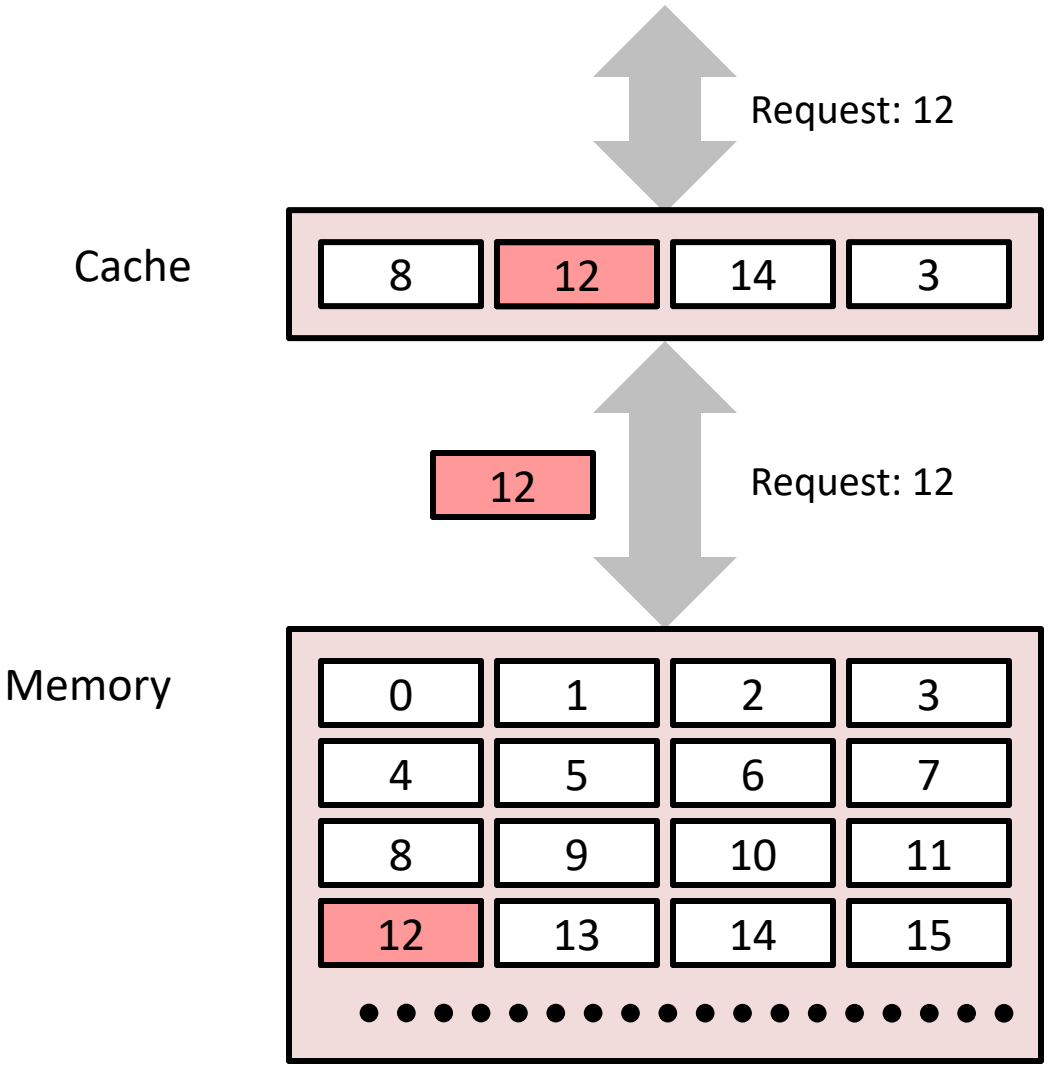
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
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

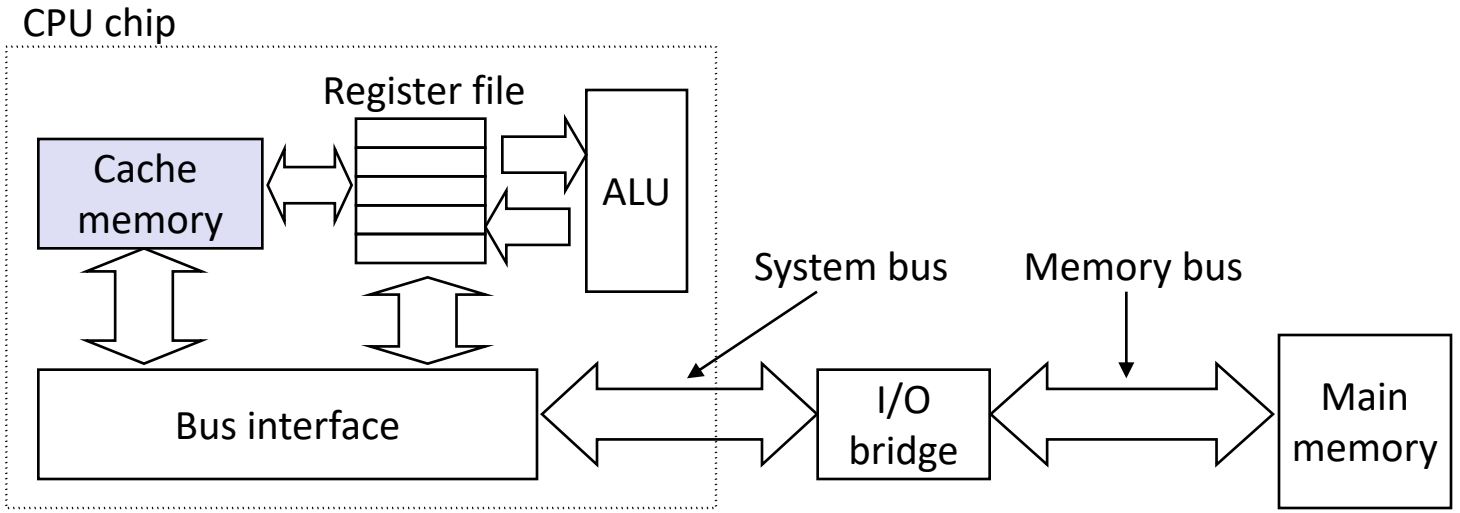
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

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 mod 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.

Cache Memories

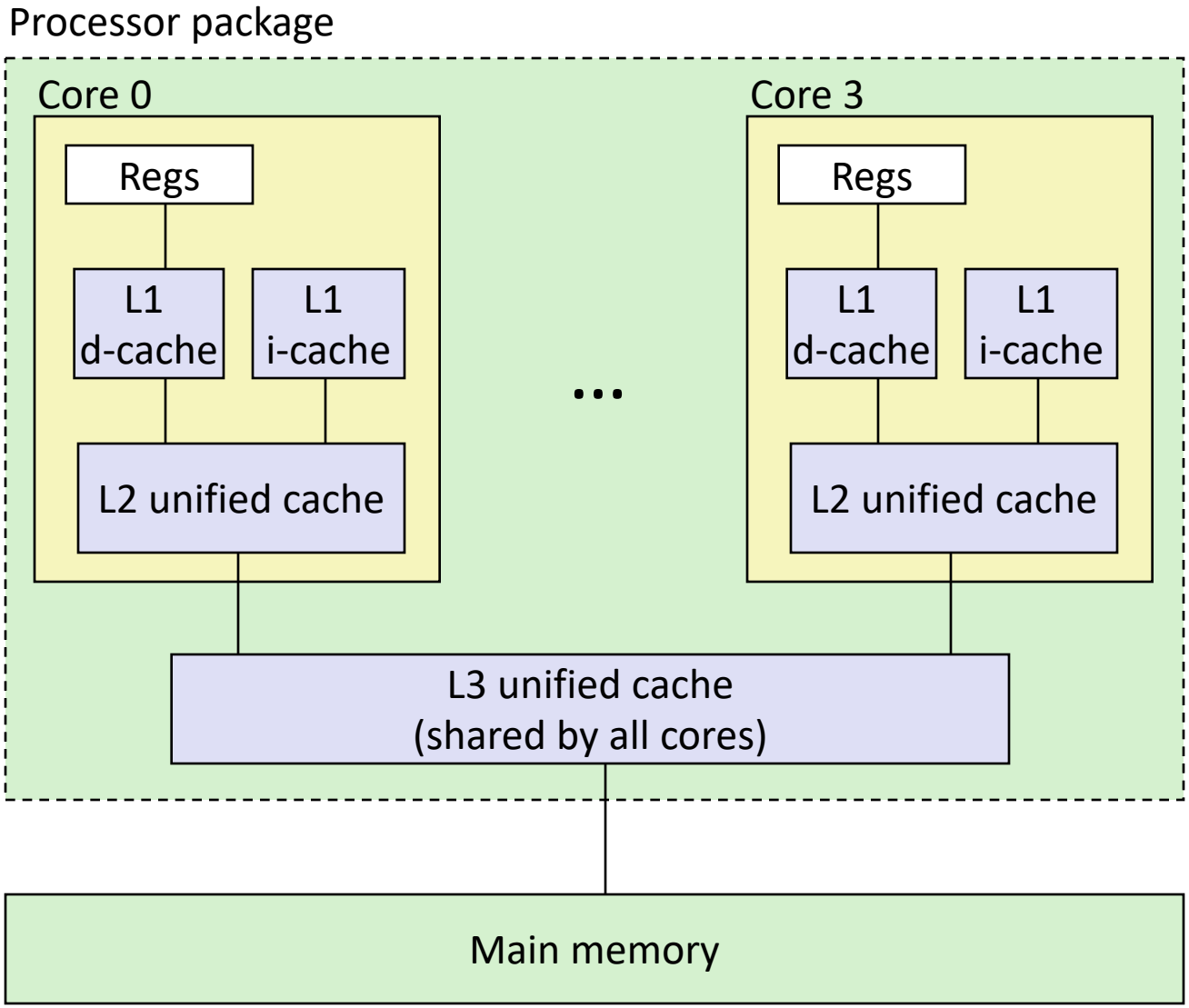
- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



What about writes?

- Multiple copies of data exist:
  - L1, L2, L3, 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 straight to memory, does not load into cache)
- 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.

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

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size