CSC 252/452: Computer Organization
Fall 2024: Lecture 12

Instructor: Yanan Guo

Department of Computer Science
University of Rochester

Class Review

- Data representation
  - Binary/Bits
  - Integer
  - Floating Point
- Assembly programing
  - ISA basics
  - Assembly basics
    - Memory, architectural states, pointers
  - Instructions (mov, arithmetic, jump)
  - Condition codes
- Functions
  - Stack & frame
- Data structures and buffer overflow
- Instruction Encoding
  - How does the assembler work?

Data Representation

- Binary (bits)
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion
  - Arithmetics
- Floating Point

Binary Notation

- Base 2 Number Representation (Binary)
- C.f., Base 10 number representation (Decimal)
- $21_{10} = 1*10^0 + 2*10^1 = 21$
- Weighted Positional Notation
  - Each bit has a weight depending on its position
- $1011_2 = 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 11_{10}$
- $b_3b_2b_1b_0 = b^0*2^0 + b^1*2^1 + b^2*2^2 + b^3*2^3$
- Binary Arithmetic

```
  0110
+ 0101
------
  1011
```

```
  6
+ 5
---
 11
```

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

Hexdecimal (Hex) Notation

- Base 16 Number Representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Four bits per Hex digit
  - $11111110_2 = FE_{16}$
- 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

Bit Manipulation

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

Shift Operations

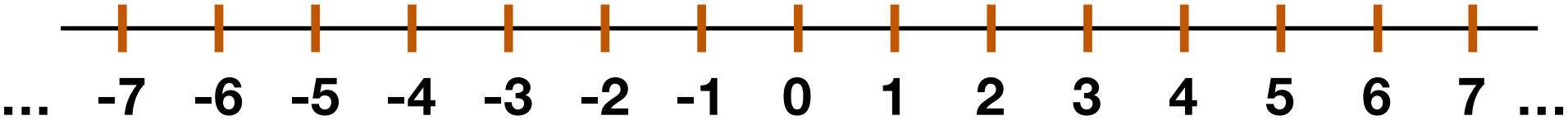
- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
  - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left

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

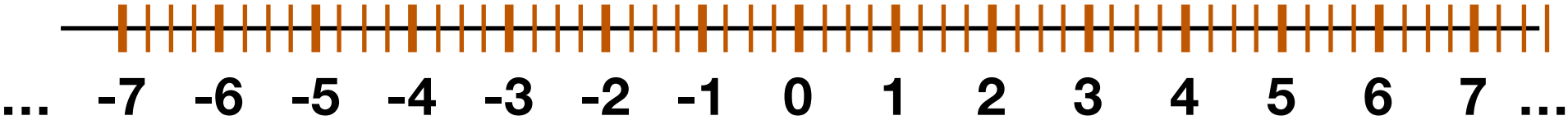
Representing Numbers in Binary

- Different types of number
  - Integer (Negative and Non-negative)
  - Fractions



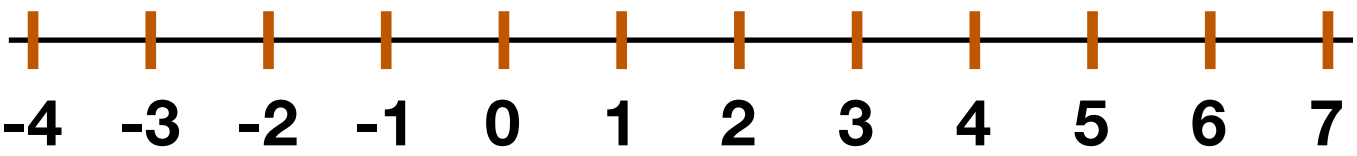
Representing Numbers in Binary

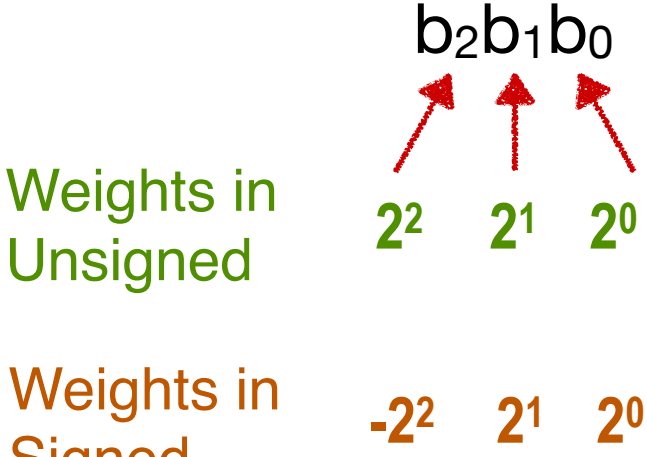
- Different types of number
  - Integer (Negative and Non-negative)
  - Fractions



Encoding Negative Numbers

- Two's Complement

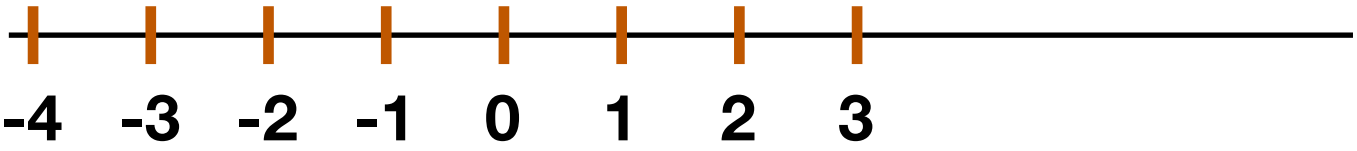


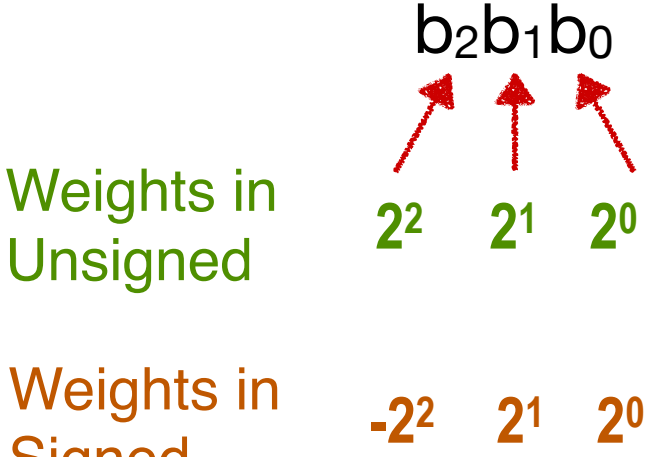


Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement

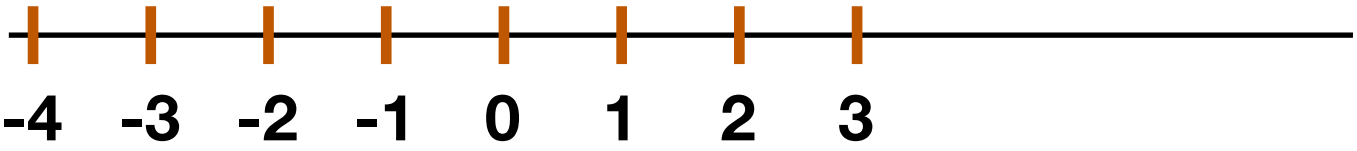




Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned: 2^2 2^1 2^0

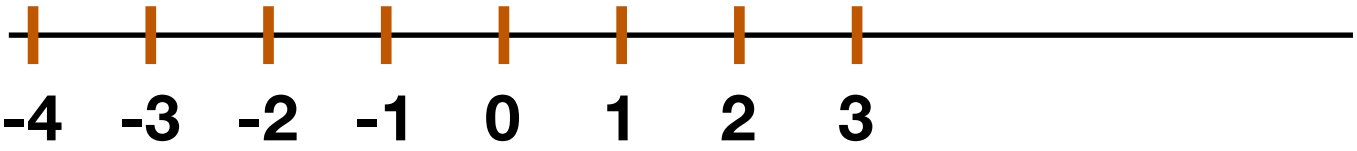
Weights in Signed: -2^2 2^1 2^0

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

$$101_2 = 1*2^0 + 0*2^1 + (-1*2^2) = -3_{10}$$

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned: 2^2 2^1 2^0

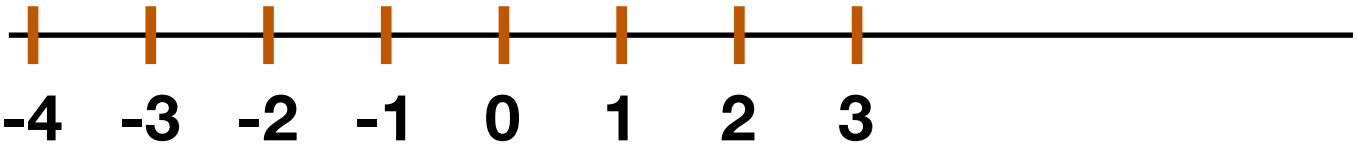
Weights in Signed: -2^2 2^1 2^0

$101_2 = 1*2^0 + 0*2^1 + (-1*2^2) = -3_{10}$

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned: 2^2 2^1 2^0

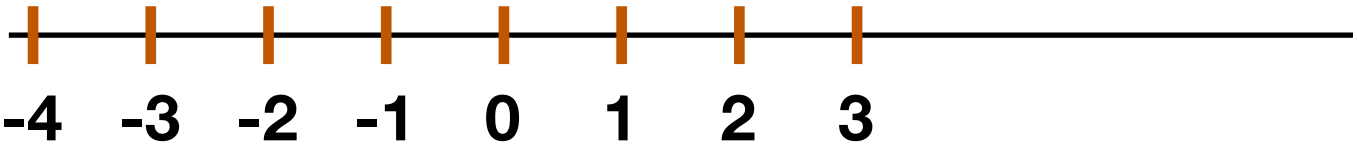
Weights in Signed: -2^2 2^1 2^0

$101_2 = 1*2^0 + 0*2^1 + (-1*2^2) = -3_{10}$

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned: 2^2 2^1 2^0

Weights in Signed: -2^2 2^1 2^0

$101_2 = 1*2^0 + 0*2^1 + (-1*2^2) = -3_{10}$

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Two-Complement Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

```
   010          2
+) 101       +) -3
-----        -----
   111          -1
```

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

- 3 + 1 becomes -4 (called overflow.)

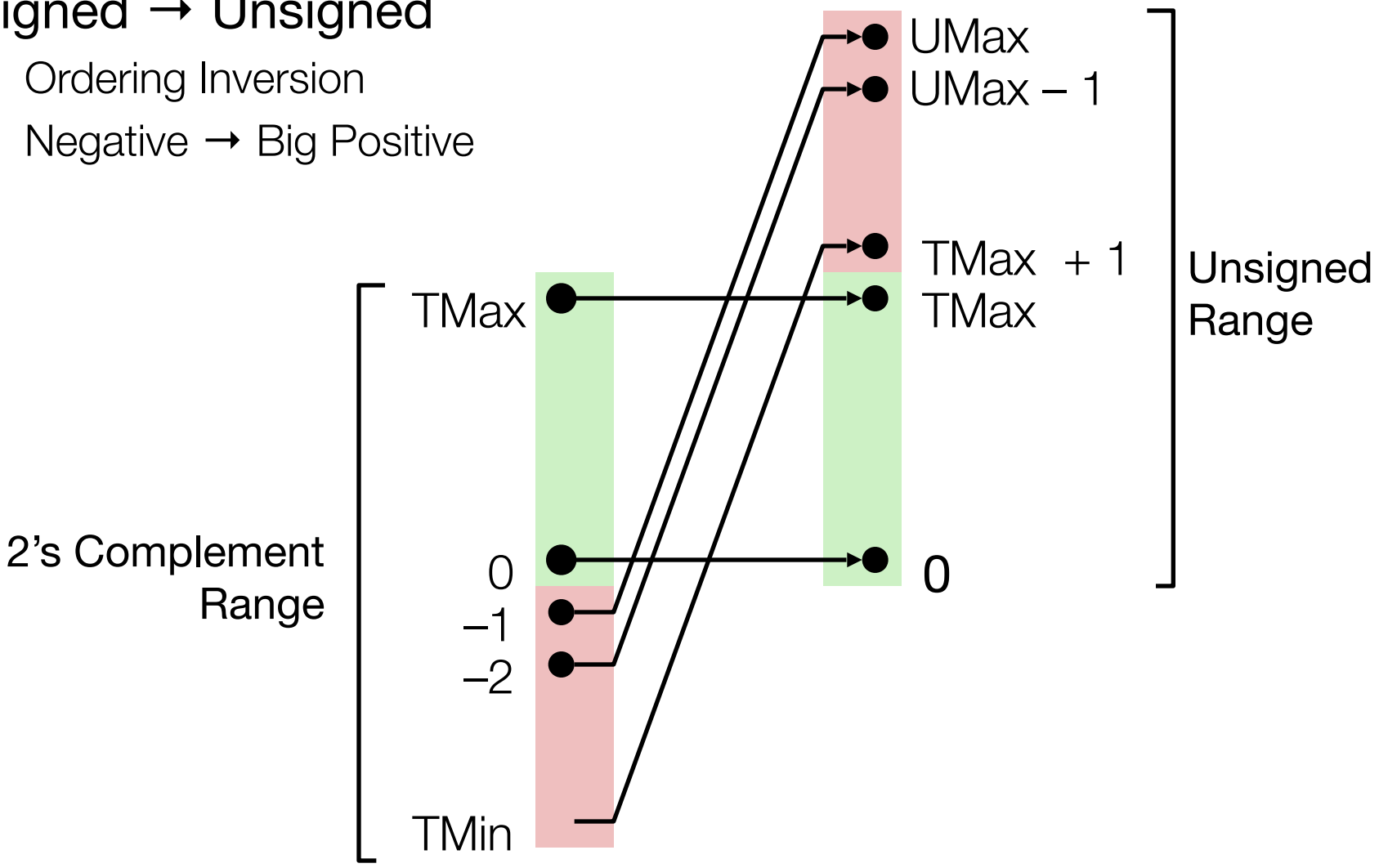
Data Representations in C (in Bytes)

- By default variables are signed
- Unless explicitly declared as unsigned (e.g., `unsigned int`)
- Signed variables use two-complement encoding

C Data Type	32-bit	64-bit
`(unsigned) char`	1	1
`(unsigned) short`	2	2
`(unsigned) int`	4	4
`(unsigned) long`	4	8

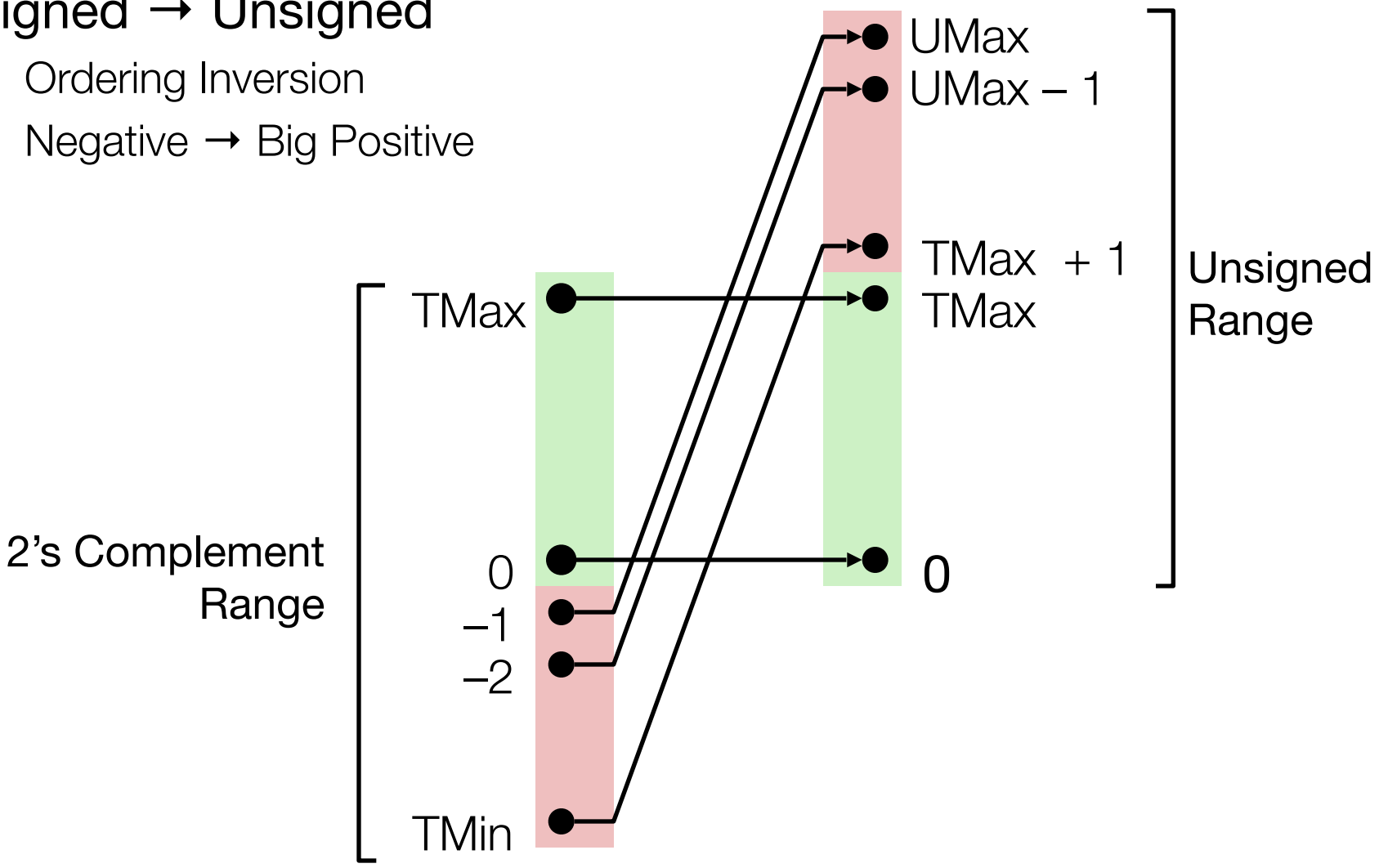
Conversion Visualized

- Signed → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



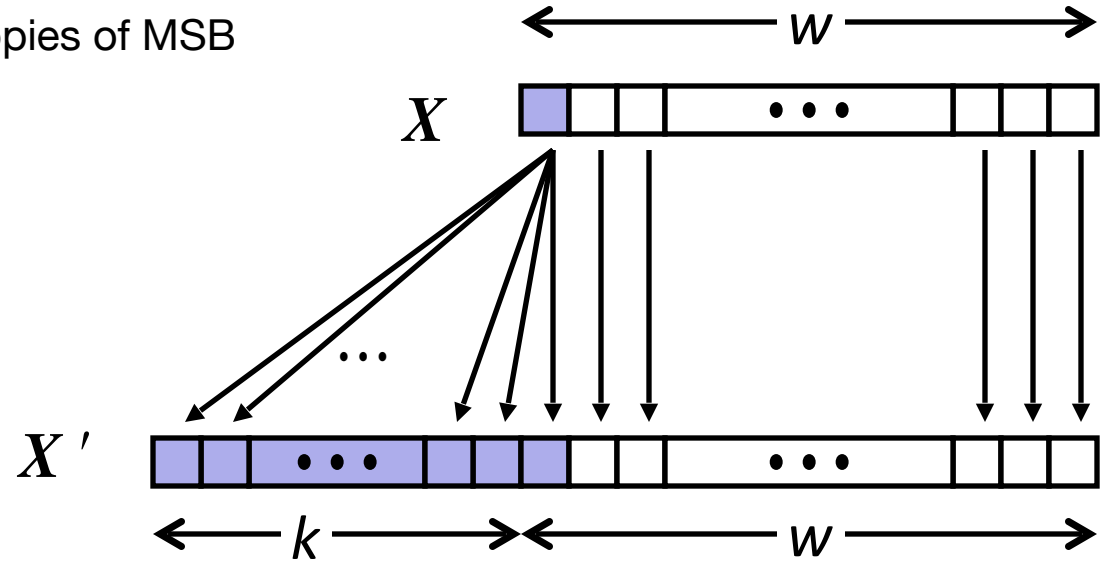
Conversion Visualized

- Signed → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



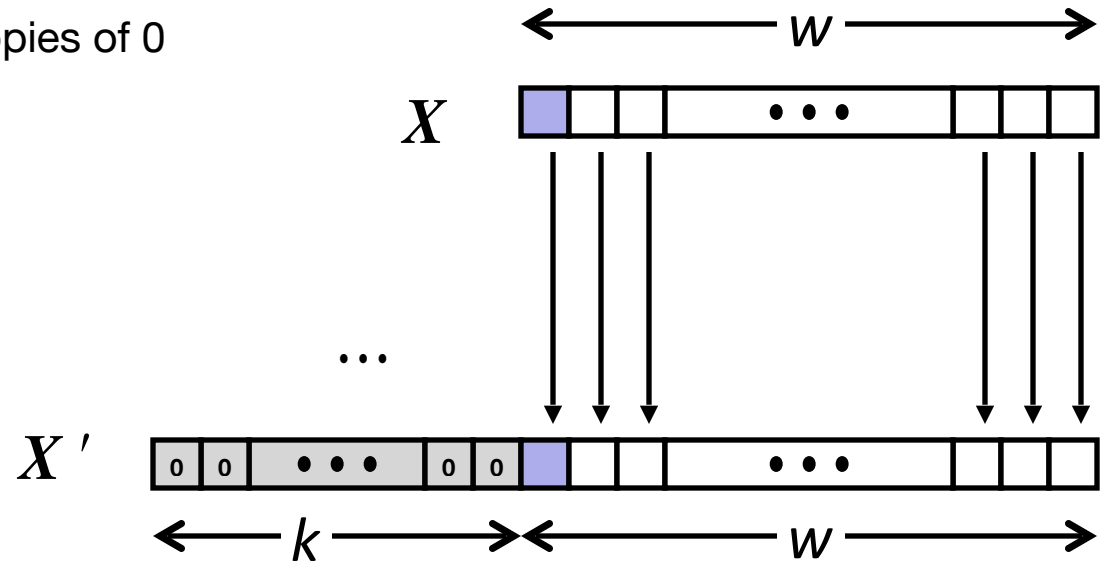
Signed Extension

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



Unsigned (Zero) Extension

- Task:
  - Given *w*-bit unsigned integer *x*
  - Convert it to (*w*+*k*)-bit integer with same value
- Rule:
  - Simply pad zeros:
  - $X' = \underbrace{0, \ldots, 0}_{k \text{ copies of } 0}, x_{w-1}, x_{w-2}, \ldots, x_0$

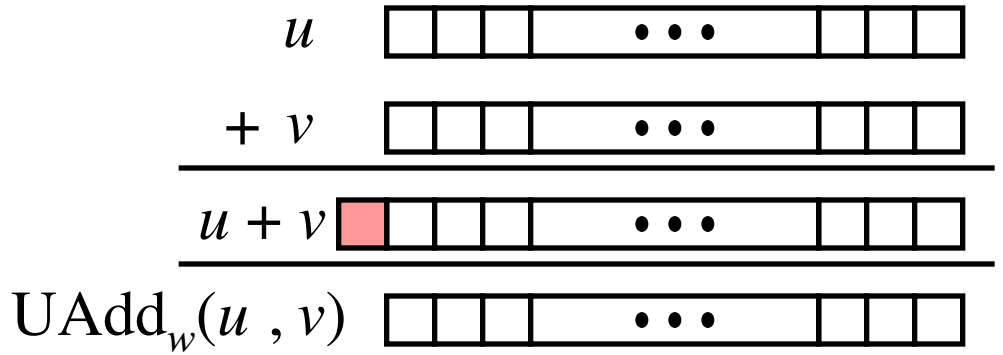


Unsigned Addition in C

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits

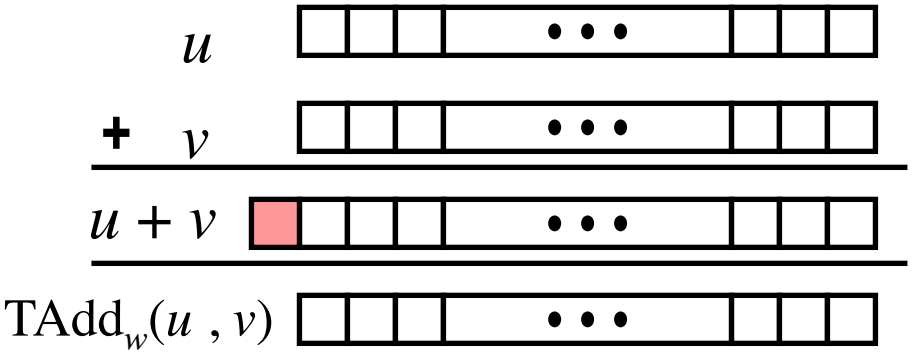


Two's Complement Addition in C

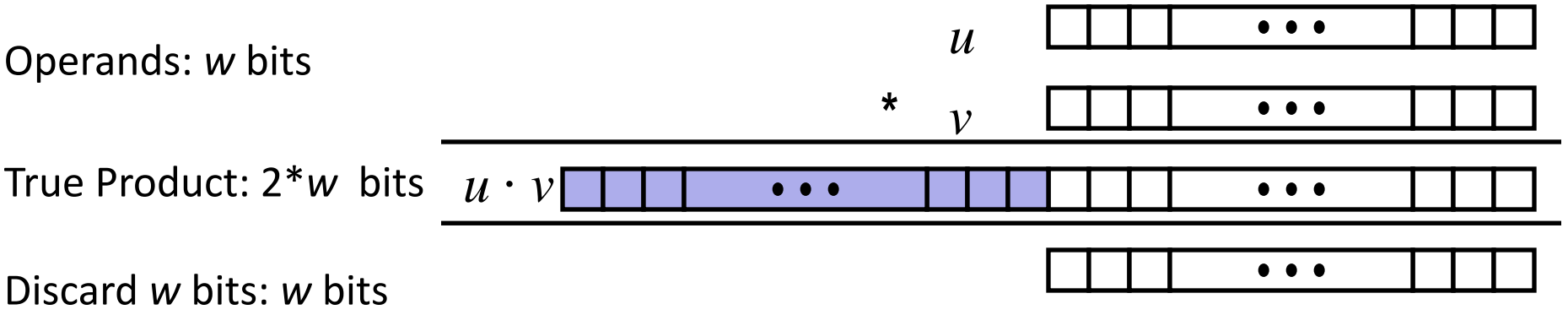
Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Unsigned Multiplication in C



- Standard Multiplication Function
  - Ignores high order w bits
- Effectively Implements the following:

$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

Represent Fractions in Binary

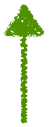
- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$

Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

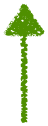
$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$



Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

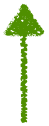
$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$



Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

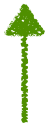
$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$



Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$



Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

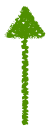
$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

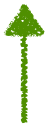


Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

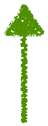


Represent Fractions in Binary

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$



Represent Fractions in Binary

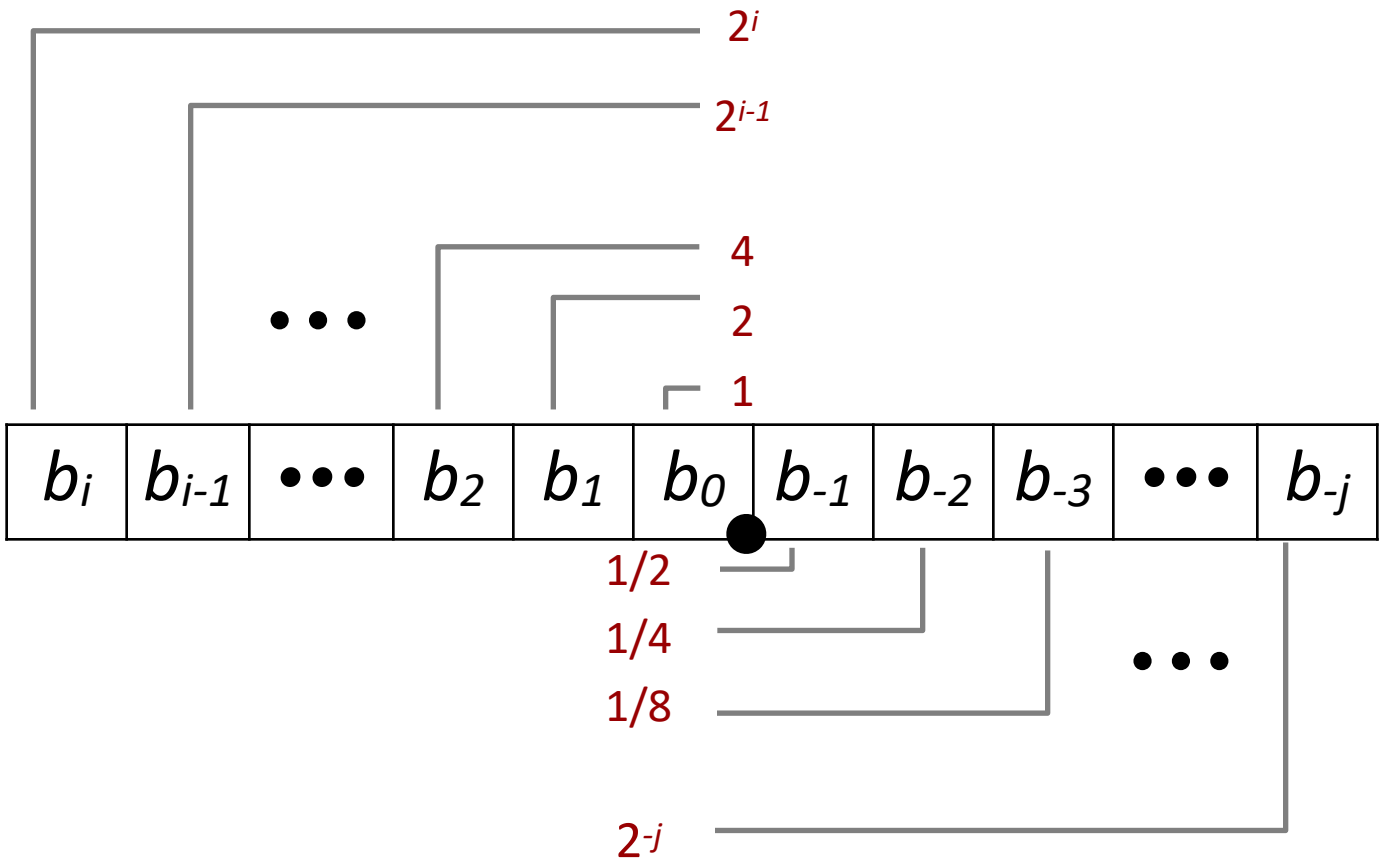
- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

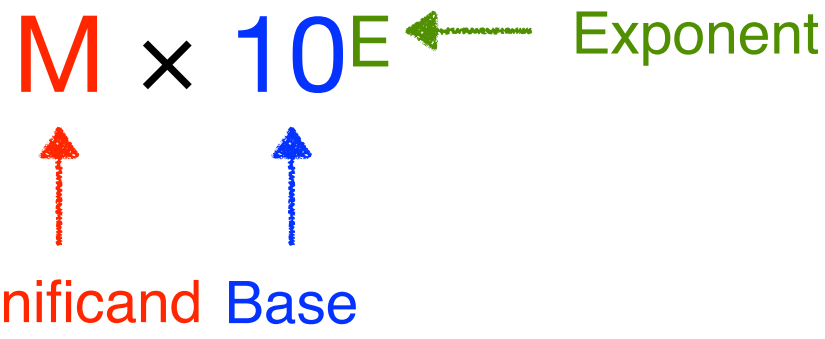
$$= 2.25_{10}$$

Fractional Binary Numbers



Primer: (Normalized) Scientific Notation

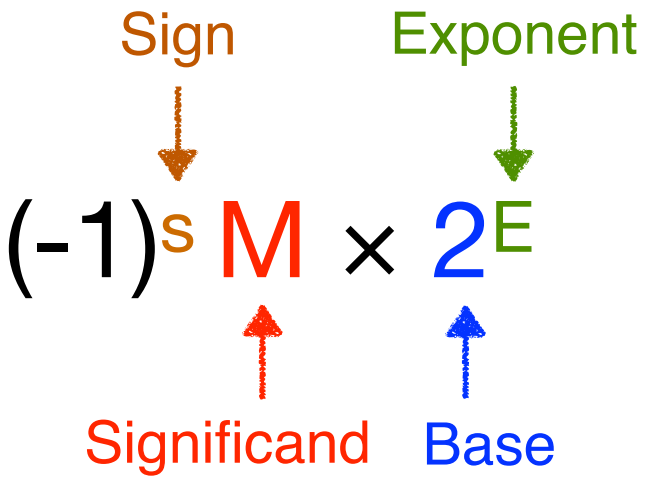
- In decimal: $M \times 10^E$
  - *E* is an integer
  - Normalized form: $1 <= |M| < 10$



Decimal Value	Scientific Notation
2	2×10^0
-4,321.768	-4.321768×10^3
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

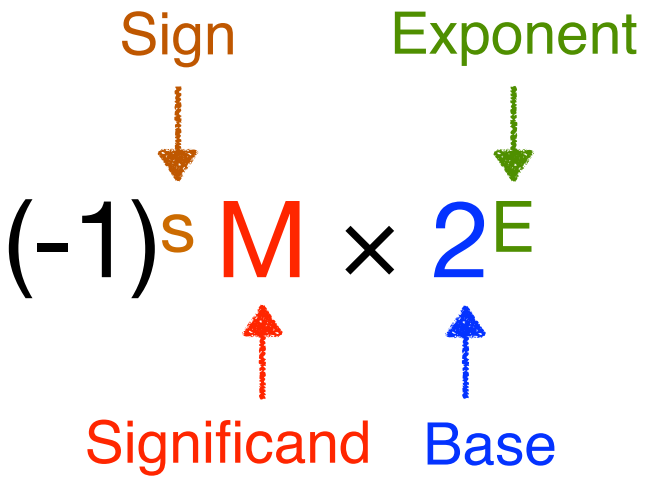
- In binary: $(-1)^s\ M\ 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.b_0b_1b_2b_3\ldots$
    Fraction



Binary Value	Scientific Notation
1110110110110	$(-1)^0\ 1.1101101101101 \times 2^{12}$
-101.11	$(-1)^1\ 1.0111 \times 2^2$
0.00101	$(-1)^0\ 1.01 \times 2^{-3}$

Primer: (Normalized) Scientific Notation

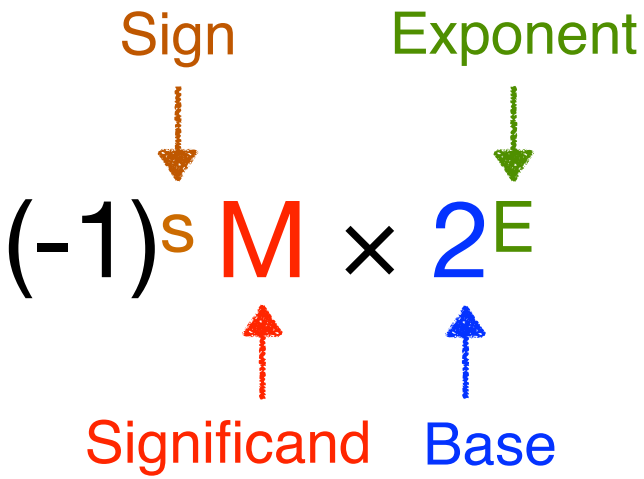
- In binary: $(-1)^s\ M\ 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.b_0b_1b_2b_3\ldots$
    Fraction



- If I tell you that there is a number where:
  - Fraction = 0101
  - s = 1
  - E = 10
  - You could reconstruct the number as $(-1)^1 1.0101 \times 2^{10}$

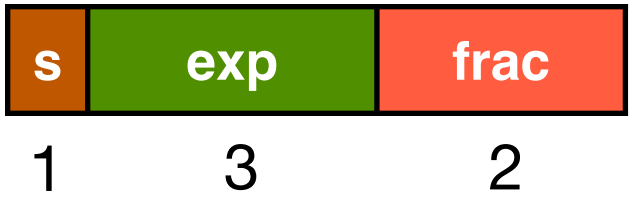
Primer: Floating Point Representation

- In binary: $(-1)^s\ M\ 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.b_0b_1b_2b_3\ldots$
    Fraction
- Encoding
  - MSB *s* is sign bit s
  - *exp* field encodes Exponent (but not exactly the same, more later)
  - *frac* field encodes Fraction (but not exactly the same, more later)



6-bit Floating Point Example

$v = (-1)^s M 2^E$



- *exp* has 3 bits, interpreted as an unsigned value
  - If *exp* were *E*, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: *E = exp - bias* (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits

E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~

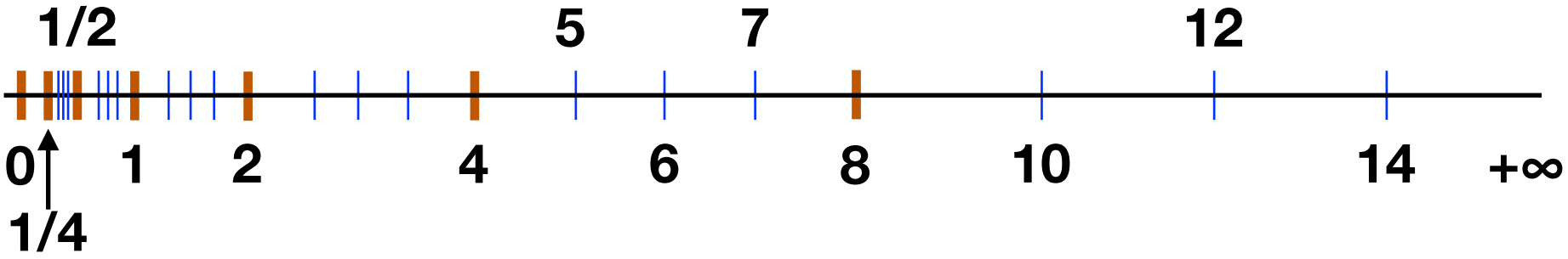
- Example when we use 3 bits for exp (i.e., *k* = 3):
  - bias = 3
  - If *E* = -2, *exp* is 1 (001_2)
  - Reserve 000 and 111 for other purposes (more on this later)
  - We can now represent exponents from **-2 (exp 001) to 3 (exp 110)**

Representable Numbers (Positive Only)

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

E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

- Uneven interval (c.f., fixed interval in fixed-point)
  - More dense toward 0, sparser toward infinite
  - Allow encoding small and large numbers at the same time



IEEE 754 Floating Point Standard

- Single precision: 32 bits

- Double precision: 64 bits

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent E: E1
    - Assume E1 > E2
- Fixing
  - If $M \geq 2$, shift *M* right, increment *E*
  - If $M < 1$, shift *M* left *k* positions, decrement *E* by *k*
  - Overflow if *E* out of range
  - Round *M* to fit *frac* precision

$1.000 \times 2^{-1} + 10.10 \times 2^{-4}$

align $1.000 \times 2^{-1} + 0.0101 \times 2^{-1}$

add 1.0101×2^{-1}

1.010×2^{-1}

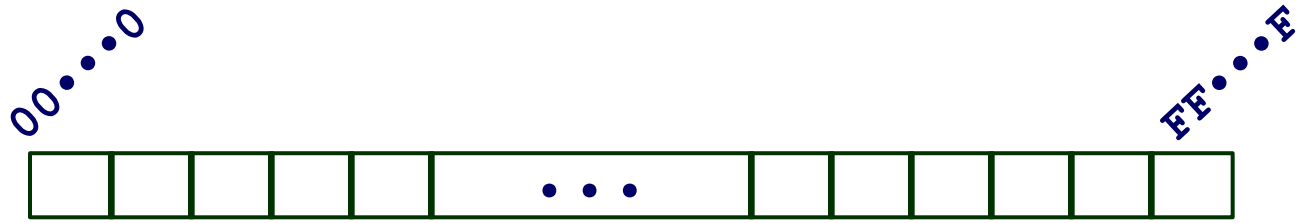
Mathematical Properties of FP Add

- Commutative? ***Yes***
- Associative? ***No***
  - Overflow and inexactness of rounding
  - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
- 0 is additive identity? ***Yes***
- Every element has additive inverse (negation)? ***Almost***
  - Except for infinities & NaNs
- Monotonicity: $a \geq b \Rightarrow a+c \geq b+c$? ***Almost***
  - Except for infinities & NaNs

Assembly Programming

- Basics
  - Memory
  - Pointers in C
  - Architectural states
- Instructions
  - Move instructions
  - Arithmetic instructions
  - Jump instructions

Byte-Oriented Memory Organization

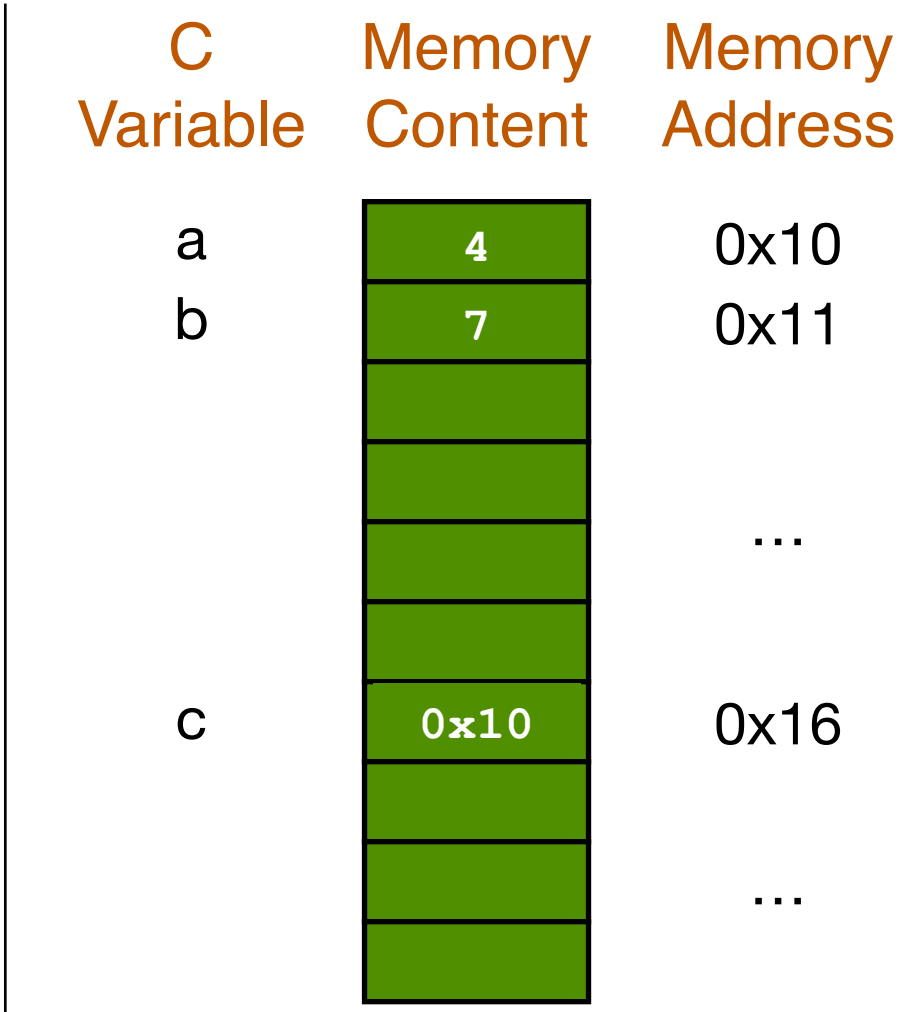


- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes: byte-addressable
  - An address is like an index into that array
    - and, a pointer variable stores an address

How Does Pointer Work in C???

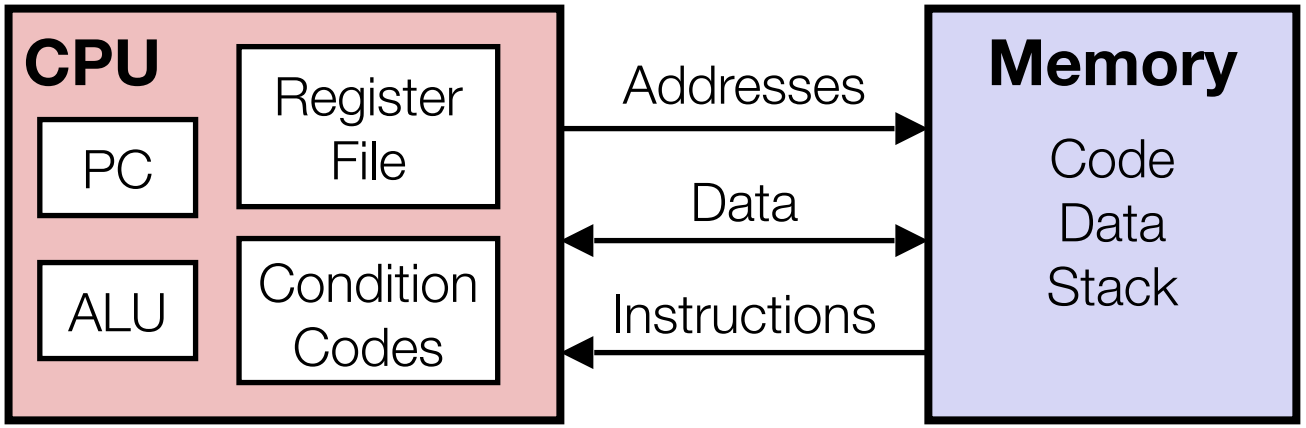
```
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.
- The '**&**' operator (address-of operator) returns the memory address of a variable.
- The '*****' operator returns the content stored at the memory location pointed by the pointer variable (dereferencing)



Assembly Code's View of Computer: ISA

Assembly Programmer's Perspective of a Computer



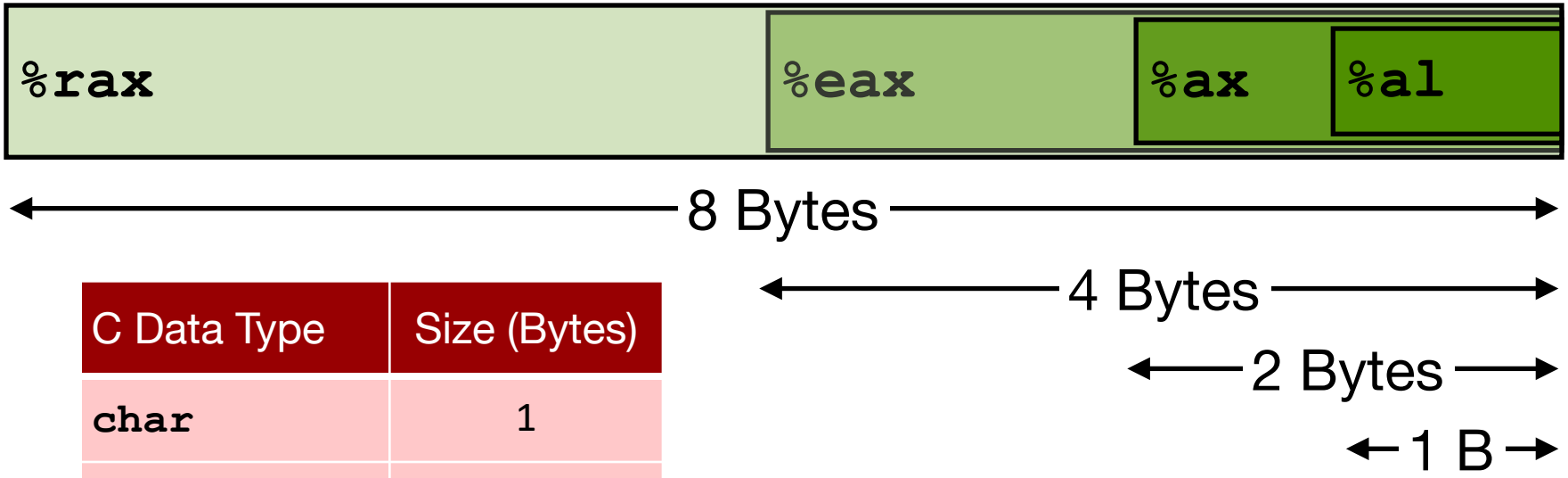
- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call
- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data
- PC: Program counter
  - A special register containing address of next instruction
  - Called "RIP" in x86-64
- Arithmetic logic unit (ALU)
  - Where computation happens
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branch

x86-64 Integer Register File

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

x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 1 bytes)

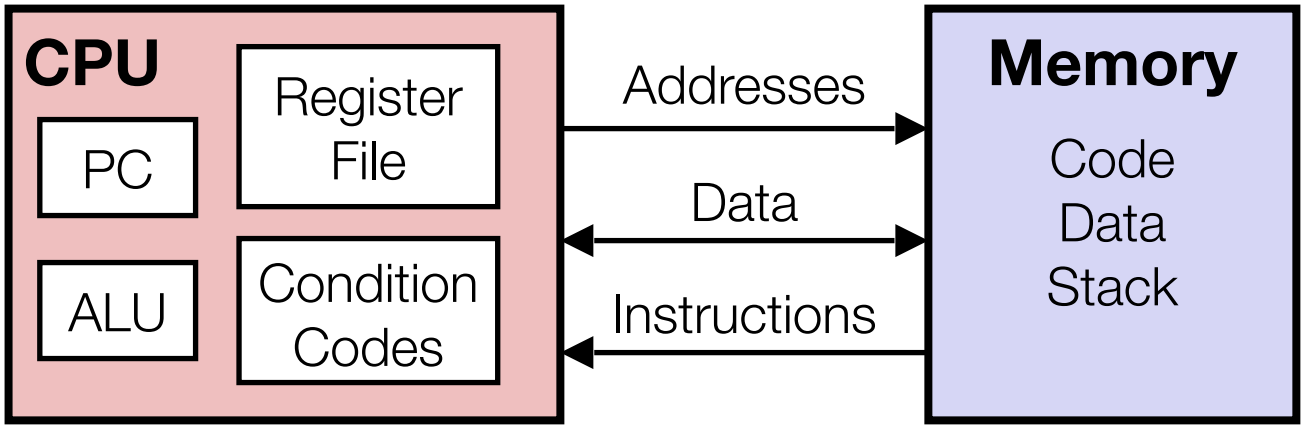


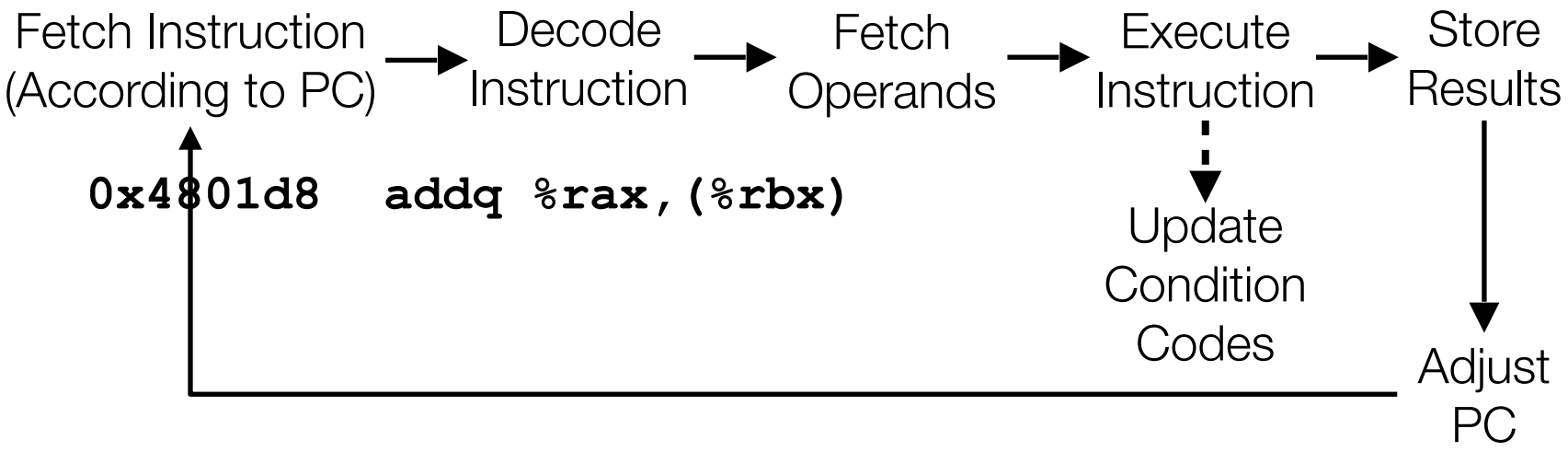
C Data Type	Size (Bytes)
`char`	1
`short`	2
`int`	4
`long`	8
Pointer	8

Floating point data is stored in a separate set of register file

Instruction Processing Sequence

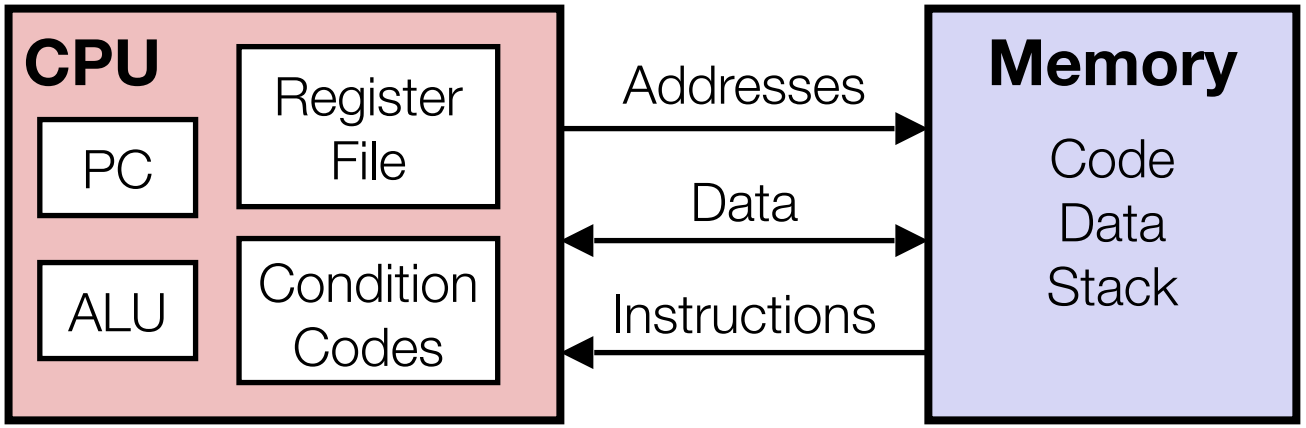
Assembly Programmer's Perspective of a Computer

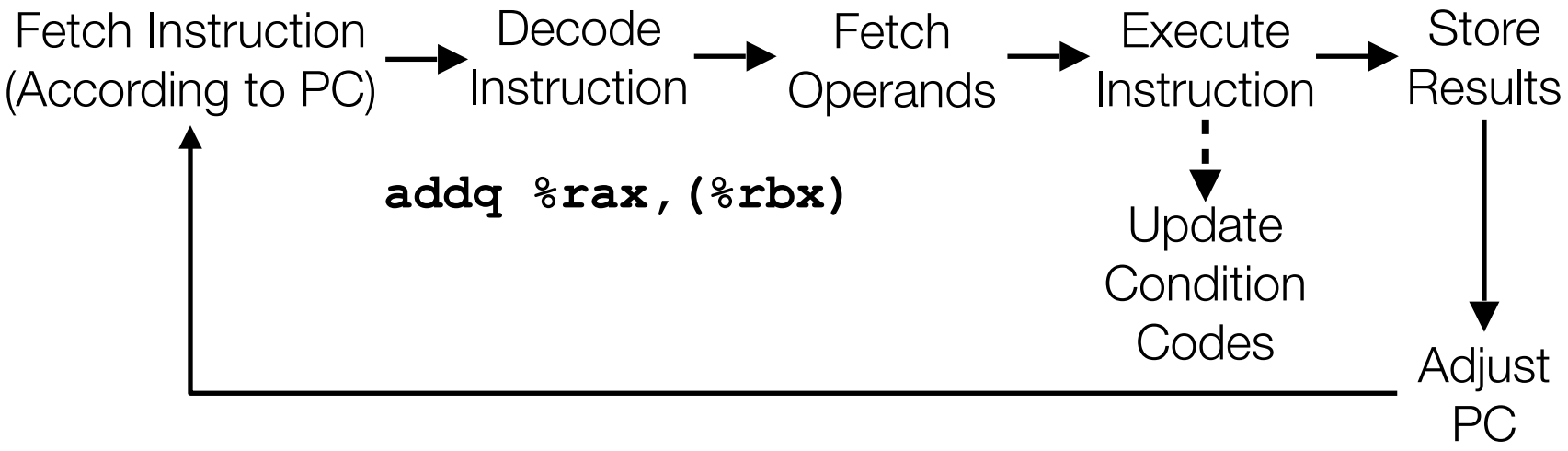




Instruction Processing Sequence

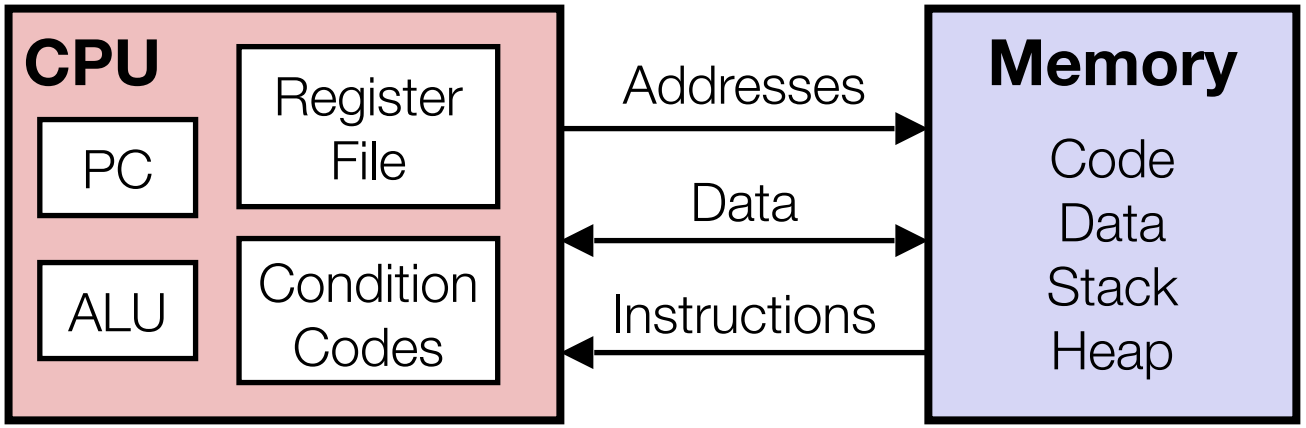
Assembly Programmer's Perspective of a Computer





Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



- *Data Movement Instruction*: Transfer data between memory and register
  - **`movq %eax, (%ebx)`**
- *Compute Instruction*: Perform arithmetics on register or memory data
  - **`addq %eax, %ebx`**
  - C constructs: +, -, >>, etc.
- *Control Instruction*: Alter the sequence of instructions (by changing PC)
  - **`jmp`**, **`call`**
  - C constructs: **`if-else`**, **`do-while`**, function call, etc.

`movq` Operand Combinations

	Source	Dest	Example	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 in x86.

Memory Addressing Modes

- An addressing mode specifies:
  - how to calculate the effective memory address of an operand
  - by using information held in registers and/or constants
- Normal: (R)
  - Memory address: content of Register R (**Reg[R]**)
  - Pointer dereferencing in C

```
movq (%rcx),%rax; // address = %rcx
```

- Displacement: D(R)
  - Memory address: **Reg[R]+D**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx; // address = %rbp + 8
```

Complete Memory Addressing Modes

- The General Form: D(Rb,Ri,S)
  - Memory address: Reg[Rb] + S * Reg[Ri] + D
  - E.g., **`8(%eax, %ebx, 4)`**; // address = **`%eax`** + 4 * **`%ebx`** + 8
  - D: Constant "displacement"
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp
  - S: Scale: 1, 2, 4, or 8
- What is **`8(%eax, %ebx, 4)`** used for?
- Special Cases

(Rb,Ri)	address = Reg[Rb]+Reg[Ri]
D(Rb,Ri)	address = Reg[Rb]+Reg[Ri]+D
(Rb,Ri,S)	address = Reg[Rb]+S*Reg[Ri]

Address Computation Instruction

```
leaq 4(%rsi,%rdi,2), %rax
```

%rax = %rsi + %rdi * 2 + 4

- **`leaq`** Src, Dst
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
  - No actual memory reference is made
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of p = &x[i];

Some Arithmetic Operations (2 Operands)

Format	Computation	Notes
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 shift
shrq src, dest	Dest = Dest >> Src	Logical shift
xorq src, dest	Dest = Dest ^ Src	
andq src, dest	Dest = Dest & Src	
orq src, dest	Dest = Dest | Src	

Some Arithmetic Operations (2 Operands)

- No distinction between signed and unsigned (why?)
  - Bit level behaviors for signed and unsigned arithmetic are exactly the same — assuming truncation

Bit-level	Signed	Unsigned
010	2	2
+) 101	+) -3	+) 5
111	-1	7

```
long signed_add
(long x, long y)
{
  long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

```
long unsigned_add
(unsigned long x, unsigned long y)
{
  unsigned long res = x + y;
  return res;
}
```

```
#x in %rdx, y in %rax
addq    %rdx, %rax
```

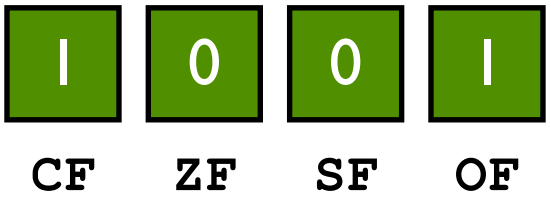
Condition Codes

`addq %rax, %rbx`

Arithmetic instructions implicitly set condition codes (think of it as side effect)

- **CF** set if %rax + %rbx generates a carry (i.e., unsigned overflow)
- **ZF** set if %rax + %rbx == 0
- **SF** set if %rax + %rbx < 0
- **OF** set if %rax + %rbx overflows when %rax and %rbx are treated as signed numbers
  - %rax > 0, %rbx > 0, and (%rax + %rbx) < 0), or
  - %rax < 0, %rbx < 0, and (%rax + %rbx) >= 0)

```
   100
+) 111
-------
  c011
```



Compare Instruction

`cmpq a, b`

- Computes $b - a$ (just like **sub**)
- Sets condition codes based on result, but…
- **Does not change b**
- All it does is setting condition codes!

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`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

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

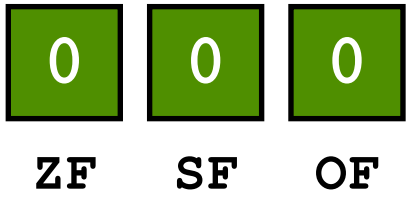
```
long absdiff
   (long x, long y)
{
   long result;
   if (x > y)
     result = x-y;
   else
     result = y-x;
   return result;
}
```

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

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

`cmpq` sets ZF, SF, OF

`jle` checks ZF | (SF ^ OF)



“Do-While” Loop Example

- Popcount: Count number of 1’s in argument `x`

`do-while` version

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

`goto` Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

“Do-While” Loop Assembly

```
long pcount_goto
   (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
   movl    $0, %rax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %rdx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    $1, %rdi    #  x >>= 1
   jne     .L2         #  if (x) goto loop
   ret
```

General “Do-While” Translation

`do-while` version

```
<before>;
do {
   body;
} while (A < B);
<after>;
```

`goto` Version

```
     <before>
.L1: <body>
     if (A < B)
       goto .L1
     <after>
```

Assembly Version

```
     <before>
.L1: <body>
     cmpq B, A
     jl .L1
     <after>
```

General “While” Translation

`while` version

```
<before>;
while (A < B) {
   body;
}
<after>;
```

`goto` Version

```
      <before>
      goto .L2
.L1:  <body>
.L2:  if (A < B)
        goto .L1
      <after>
```

Assembly Version

```
      <before>
      jmp .L2
.L1:  <body>
.L2:  cmpq A, B
      jg .L1
      <after>
```

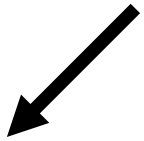
Convert “For” Loop to “While” Loop

For Version

```
before;
for (init; test; update) {
  body;
}
after
```

While Version

```
before;
init;
while (test) {
    body;
    update;
}
after;
```



Assembly Version

```
      before
      init
      jmp .L2
.L1:  body
      update
.L2:  cmpq A, B
      jg .L1
      after
```

Switch Statement Example

```
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;
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;}
    return w;
}
```

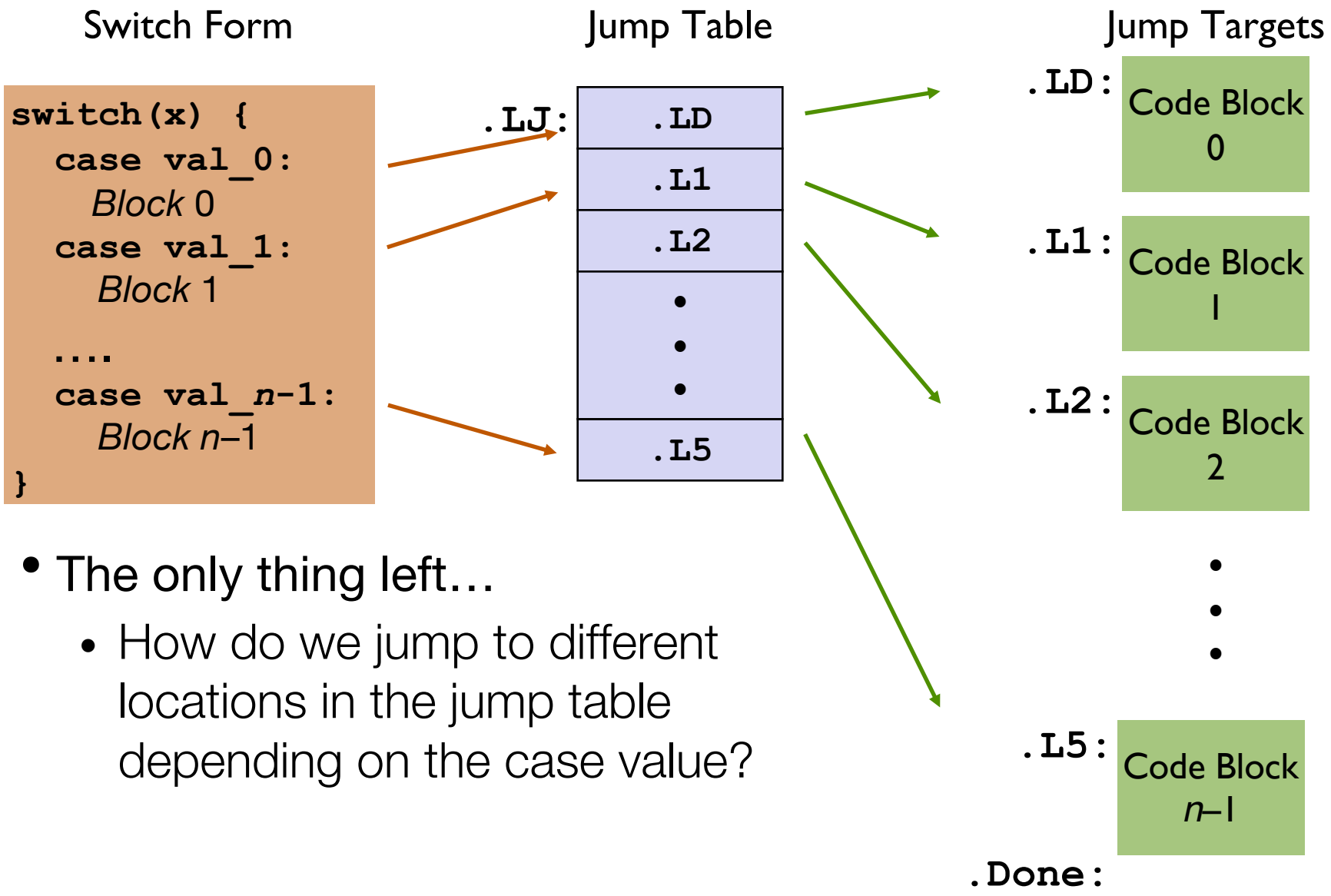
Fall-through case

Multiple case labels

For missing cases, fall back to default

Converting to a cascade of if-else statements is simple, but cumbersome with too many cases.

Implementing Switch Using Jump Table



- The only thing left…
  - How do we jump to different locations in the jump table depending on the case value?

Indirect Jump Instruction

The address we want to jump to is stored at `.LJ + 8 * x`

```
.LJ:
  .quad .LD # x = 0
  .quad .L1 # x = 1
  .quad .L2 # x = 2
  .quad .L3 # x = 3
  .quad .LD # x = 4
  .quad .L5 # x = 5
  .quad .L5 # x = 6
```

```
# assume x in %rdi
movq   .LJ(,%rdi,8), %rax
jmp    *%rax
```

- Indirect Jump: **`jmp *%rax`**
  - `%rax` specifies the address to jump to (PC = `%rax`)
- Direct Jump (**`jmp .LJ`**), directly specifies the jump address
- Indirect Jump specifies where the jump address is located

An equivalent syntax in x86:

```
jmp      *.LJ(,%rdi,8)
```

Function Calls

- `call` & `ret`
- Frame & Stack
- Stack Pointer
- Stack Content

Code Examples

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

…

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

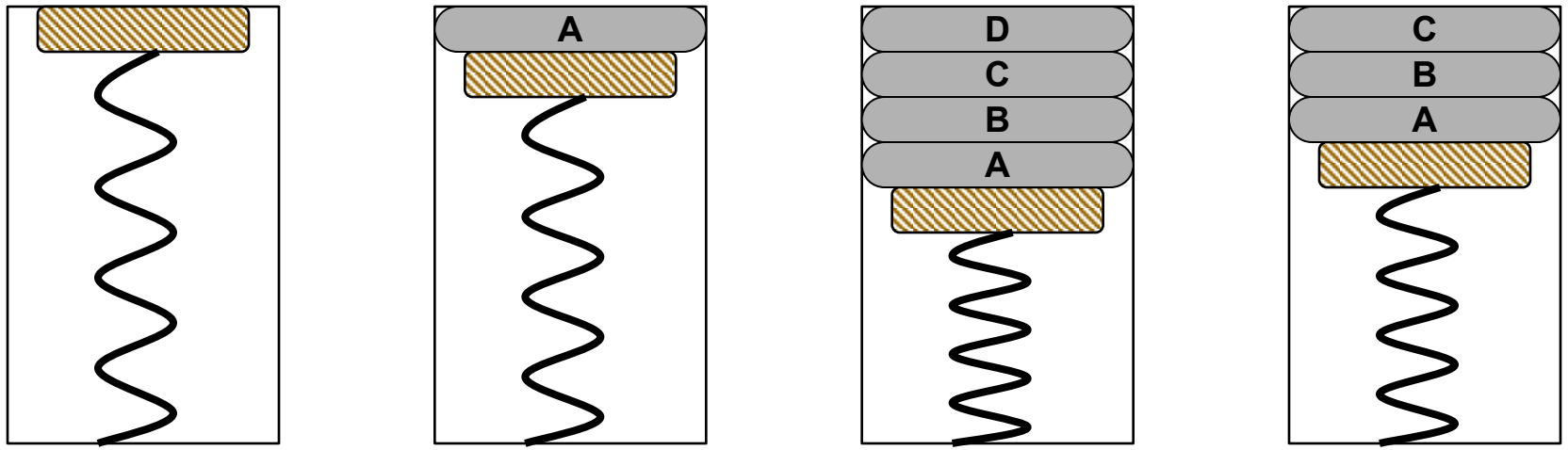
```
400540 <multstore>:
  400540: push   %rbx
  400541: mov    %rdx,%rbx
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  40054c: pop    %rbx
  40054d: retq

…

400550 <mult2>:
  400550:  mov    %rdi,%rax
  400553:  imul   %rsi,%rax
  400557:  retq
```

Function Calls — General Idea

- Frame (Active Record)
  - A frame refers to a piece of memory that contains (almost) all the information needed to execute a function, e.g., arguments and local variables
- When a function is called, create a frame, and push it to the memory
- When a function is returned, pop the frame out of the memory
- Frames are stored in memory in a *stack* fashion

A Physical Stack: A Coin Holder

First quarter out is the last quarter in.



Initial State

After
One Push

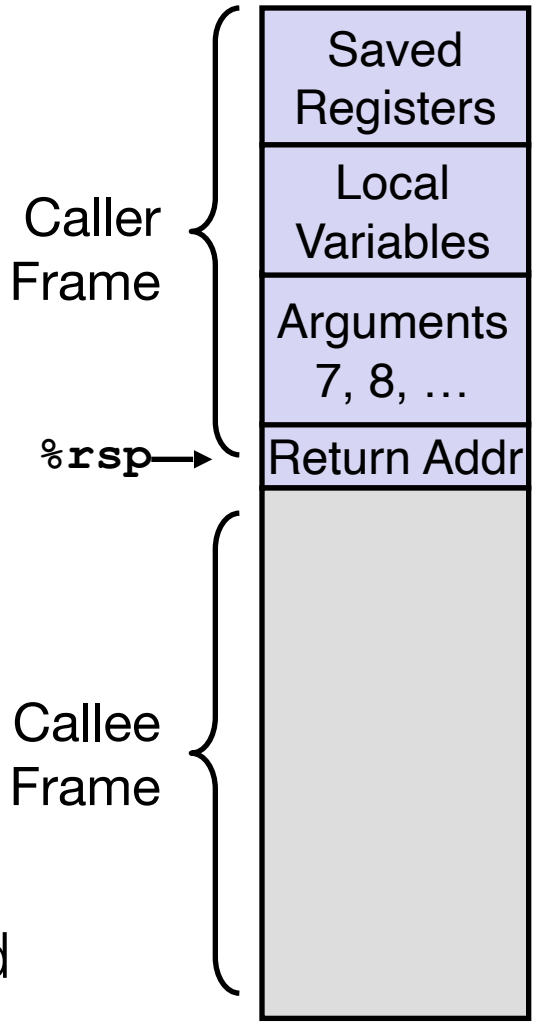
After Three
More Pushes

After
One Pop

- Stack is the right data structure for function call / return
  - If A calls B, then B returns before A

Stack

- Stack pointer: %rsp
- Stack content
  - Return address
  - Arguments
    - Convention?
  - Local variables
    - Compiler decides which local variables are stored on the stack
  - Saved registers
    - Convention?
    - Caller-saved VS callee-saved registers



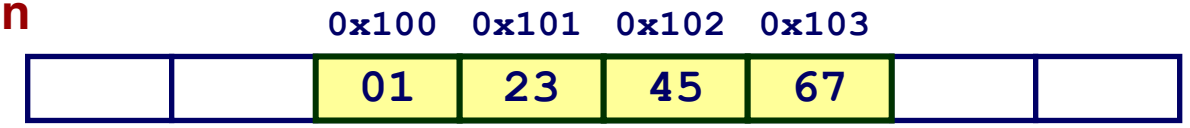
Data Structures and Buffer Overflow

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
- Structures
  - Allocation
  - Access
  - Alignment
- Buffer Overflow

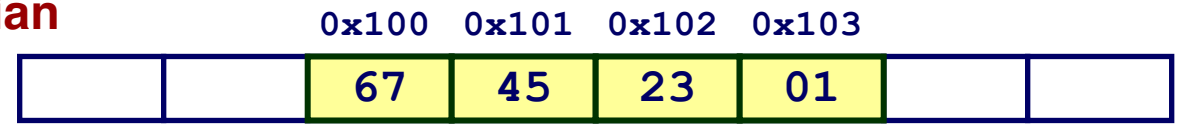
Byte Ordering

- How are the bytes of a multi-byte variable ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)

Big Endian



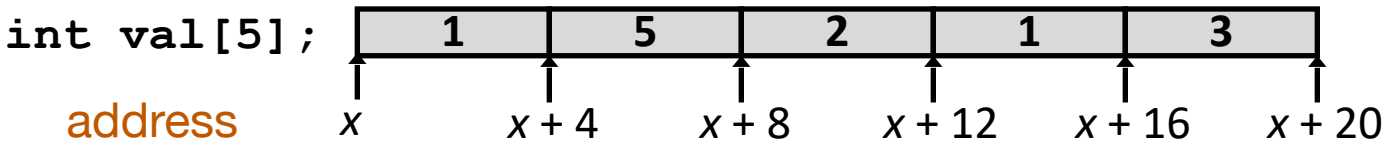
Little Endian



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

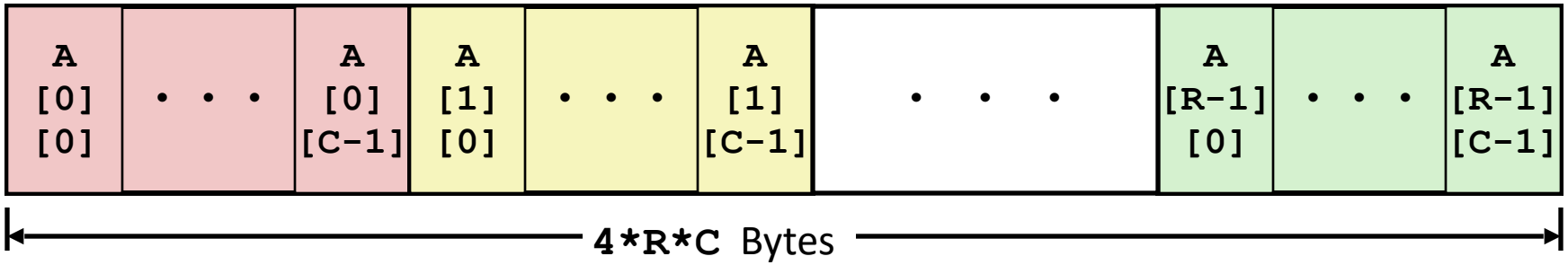
Multidimensional (Nested) Arrays

- Declaration
  - *T* **A**[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes

```
[ A[0][0]   • • •   A[0][C-1]
      •                 •
      •                 •
      •                 •
  A[R-1][0] • • • A[R-1][C-1] ]
```

- Array Size
  - *R* * *C* * *K* bytes
- Arrangement
  - Row-Major Ordering in most languages, including C

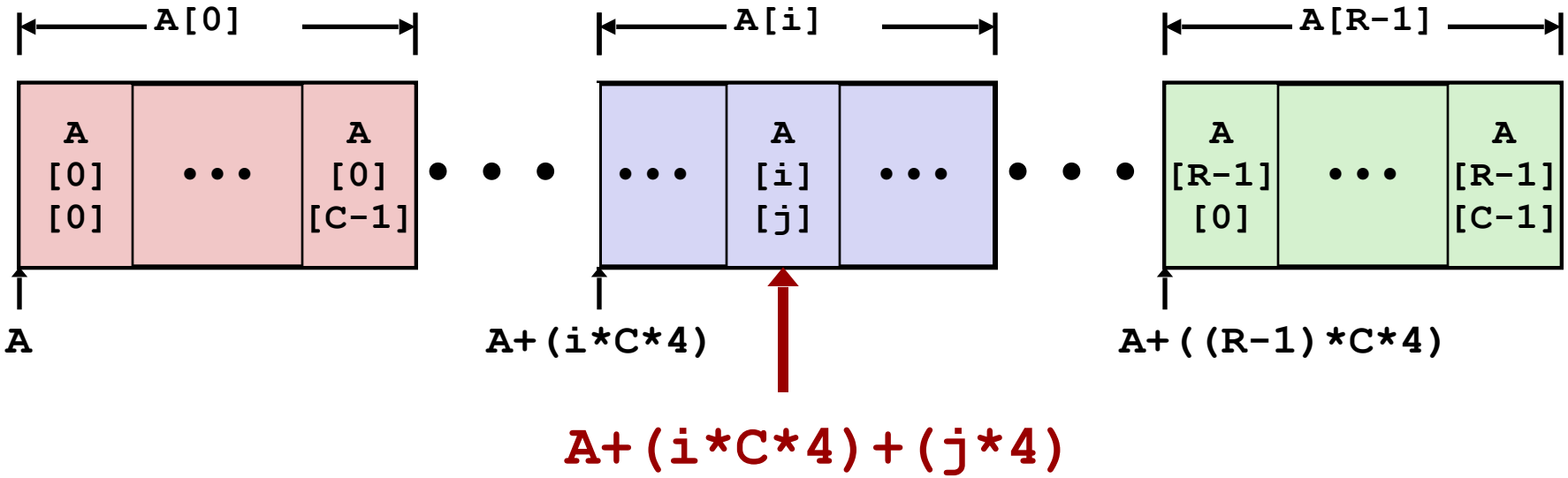
int A[R][C];



Nested Array Element Access

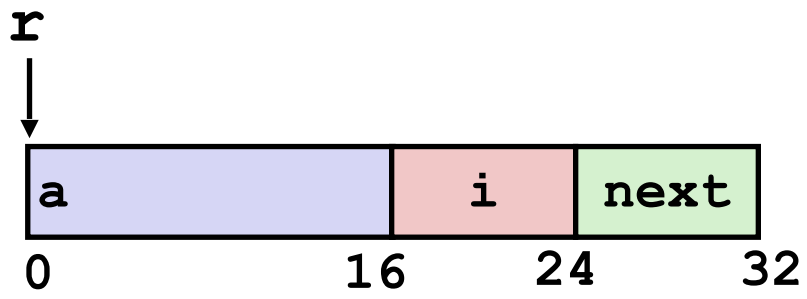
- Array Elements
  - **`A[i][j]`** is element of type *T,* which requires *K* bytes
  - Address **`A +`** *i* * (*C* * *K*) + *j* * *K* = *A* + (*i* * *C* + *j*)* *K*

```
int A[R][C];
```



Structures

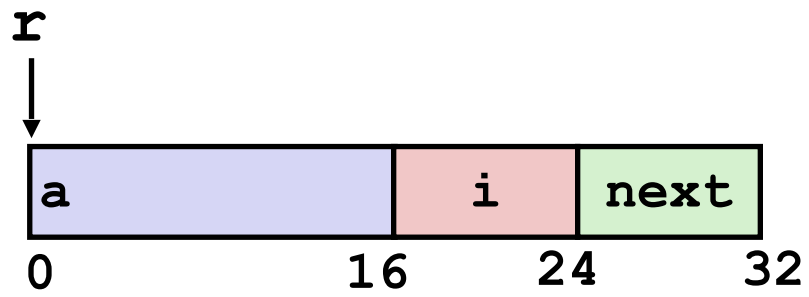
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



- Characteristics
  - Contiguously-allocated region of memory
  - Refer to members within struct by names
  - Members may be of different types

Access Struct Members

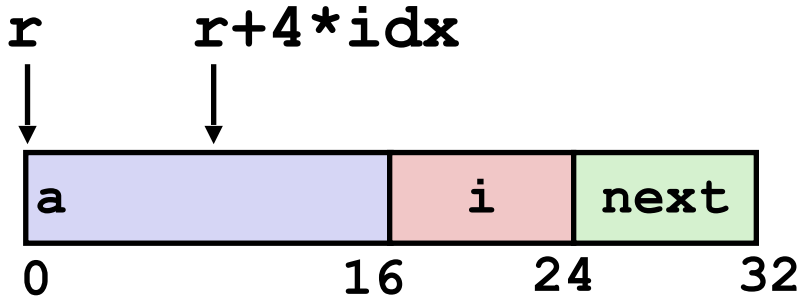
```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```



- Given a struct, we can use the . operator:
  - `struct rec r1; r1.i = val;`
- Suppose we have a pointer `r` pointing to `struct res`. How to access `res`'s member using `r`?
  - Using * and . operators: `(*r).i = val;`
  - Or simply, the -> operator for short: `r->i = val;`

Generating Pointer to Structure Member

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



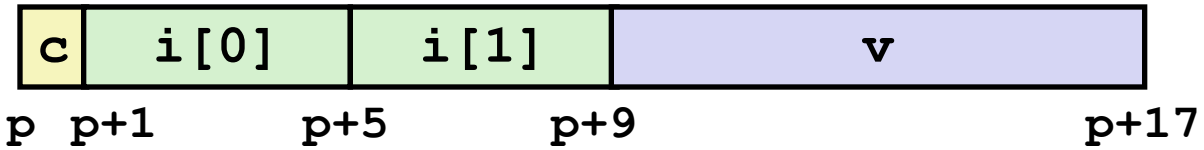

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &(r->a[idx]);
}
```

```
&((*r).a[idx])
```

```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

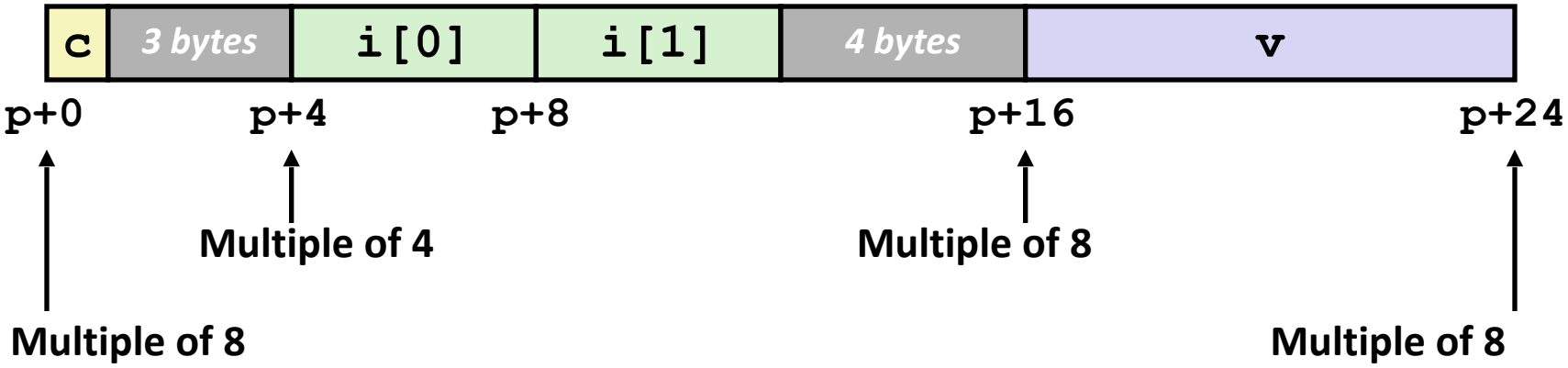
Alignment

- Unaligned Data



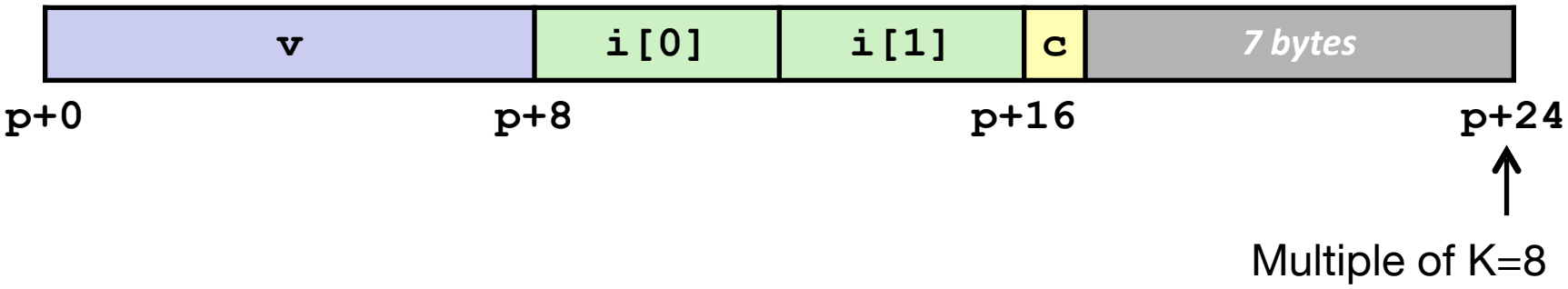

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - If the data type requires ***K*** bytes, address must be multiple of ***K***



Satisfying Alignment with Structures

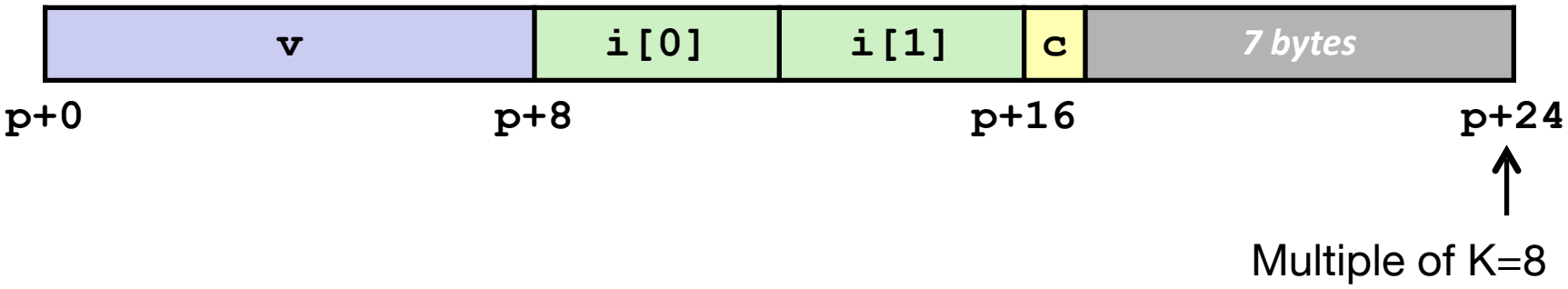
- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Structure length must be multiples of **K**, where:
    - **K** = Largest alignment of any element



Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Structure length must be multiples of **K**, where:
    - **K** = Largest alignment of any element

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```

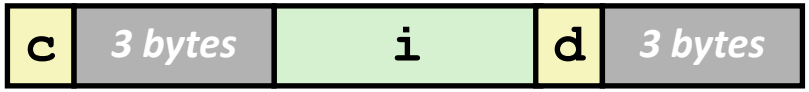


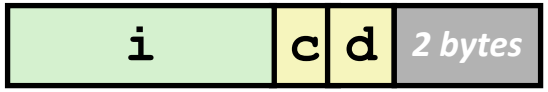
Saving Space

- Put large data types first in a Struct
- This is not something that a C compiler would always do
  - But knowing low-level details empower a C programmer to write more efficient code

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

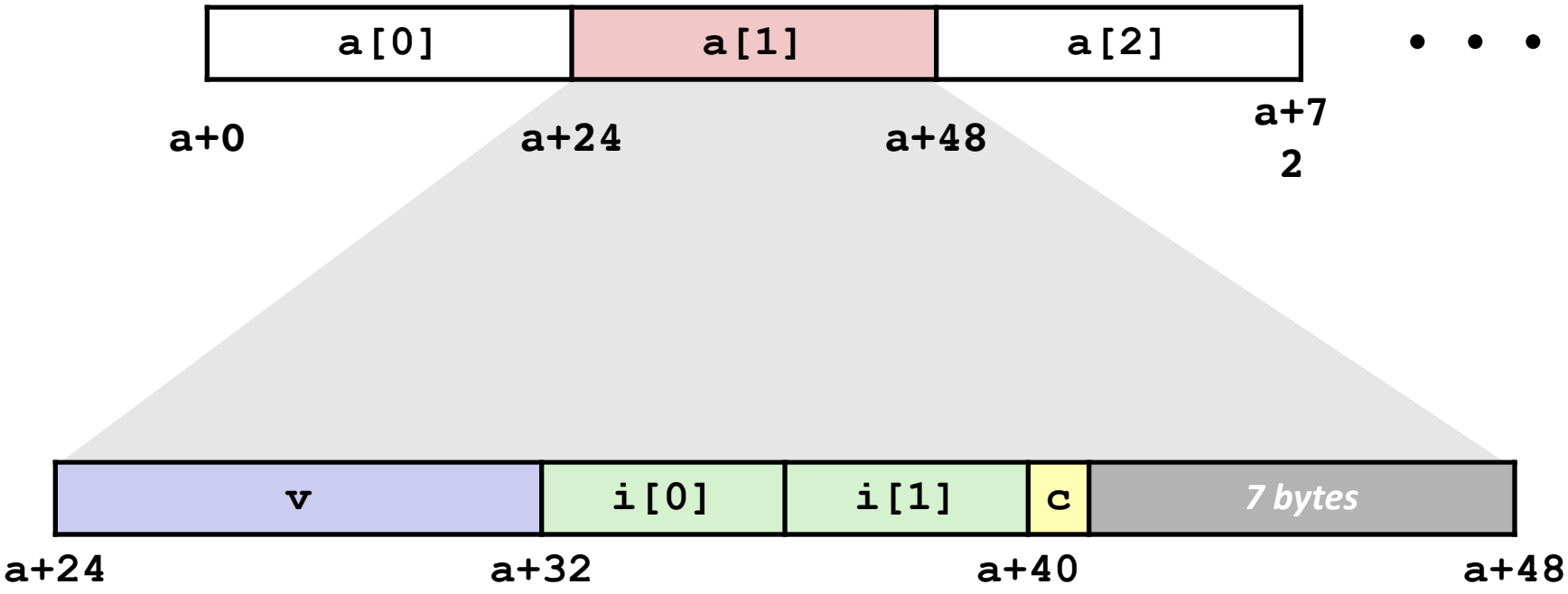




Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:0123
0123
```

```
unix>./bufdemo-nsp
Type a string:01234
Segmentation Fault
```

Instruction Encoding

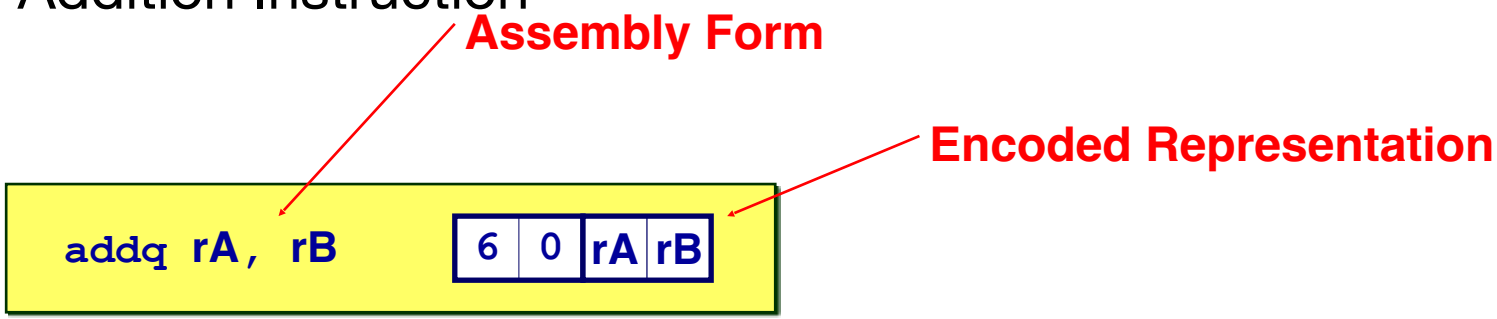
- How to translate assembly instructions to binary
  - Essentially how an assembler works
- Using the Y86-64 ISA: Simplified version of x86-64

How are Instructions Encoded in Binary?

- Remember that instructions are stored in memory **as bits** (just like data)
- Each instruction is fetched (according to the address specified in the PC), decoded, and executed by the CPU
- The ISA defines the format of an instruction (syntax) and its meaning (semantics)
- Idea: encode the two major fields, opcode and operand, separately in bits.
  - The OPCODE field says what the instruction does (e.g. ADD)
  - The OPERAND field(s) say where to find inputs and outputs

Y86 Instruction Example

Addition Instruction



- Add value in register rA to that in register rB
  - Store result in register rB
- 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

How Does An Assembler Work?

- The assembler is a program that translates assembly code to binary code
- The OS tells the assembler the start address of the code (sort of…)
- Translate the assembly program line by line
- Need to build a "label map" that maps each label to its address