CS356 Unit 4

x86 Instruction Set

Why Learn Assembly

- Understand hardware limitations
- Understand performance
- Use HW options that high-level languages don't allow (e.g., operating systems, utilizing special HW features, etc.)
- Understand security vulnerabilities
- Can help debugging

Compiling and Disassembling

- **From C to assembly code**

```
$ gcc -Og -c -S file1.c
```

- **Looking at binary files**

```
$ gcc -Og -c file1.c
$ hexdump -C file1.o
```

- **From binary to assembly**

```
$ gcc -Og -c file1.c
$ objdump -d file1.o
```

```
void abs_value (int x, int *res) {
  if (x < 0) {
    *res = -x;
  } else {
    *res = x;
  }
}
```

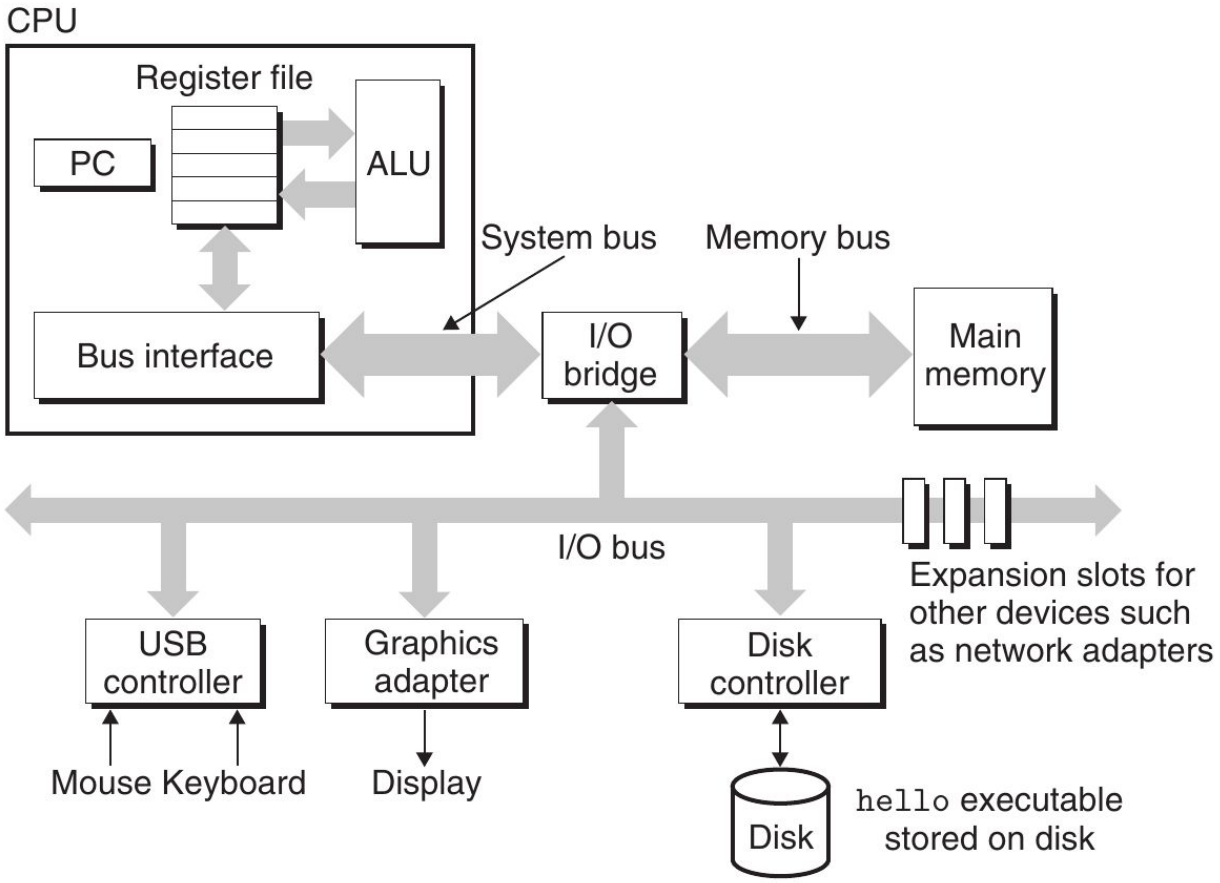
Original Code

```
Disassembly of section .text:

0000000000000000 <abs_value>:
   0:   85 ff           test %edi,%edi
   2:   78 03           js 7          "if(x<0) goto 7"
   4:   89 3e           mov %edi,(%rsi)
   6:   c3              retq
   7:   f7 df           neg %edi
   9:   89 3e           mov %edi,(%rsi)
   b:   c3              retq
```

Compiler Output
(Machine code & Assembly)
Notice how each instruction is turned into binary (shown in hex)

Basic Computer Organization



Check the recorded lecture

x86-64 Memory Organization

- Because each byte of memory has its own address we can picture memory as one column of bytes (Fig. 2)
- With 64-bit logical data bus we can access up to 8-bytes of data at a time
- We will usually show memory arranged in rows of 4 bytes (Fig. 3) or 8 bytes
  - Still with **separate addresses for each byte**

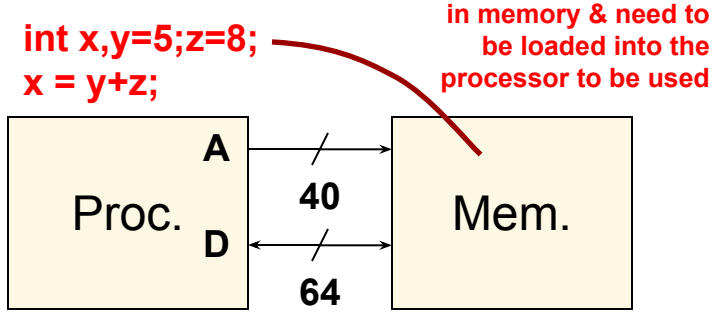


Fig. 2

...	
F8	0x000002
13	0x000001
5A	0x000000

Logical Byte-Oriented View of Mem.

Fig. 3

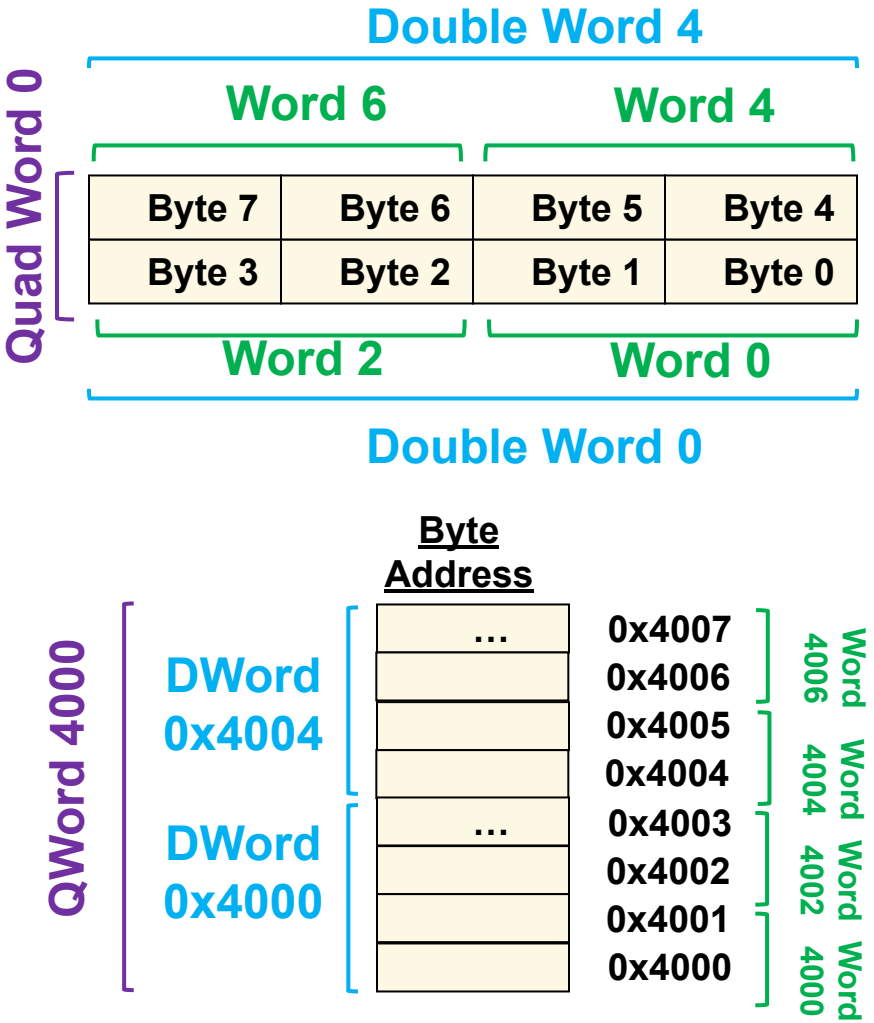
...				
b 8E	a AD	9 33	8 29	0x000008
7 8E	6 AD	5 33	4 29	0x000004
3 7C	2 F8	1 13	0 5A	0x000000

Logical DWord-Oriented View

Memory & Word Size

CS:APP 3.9.3

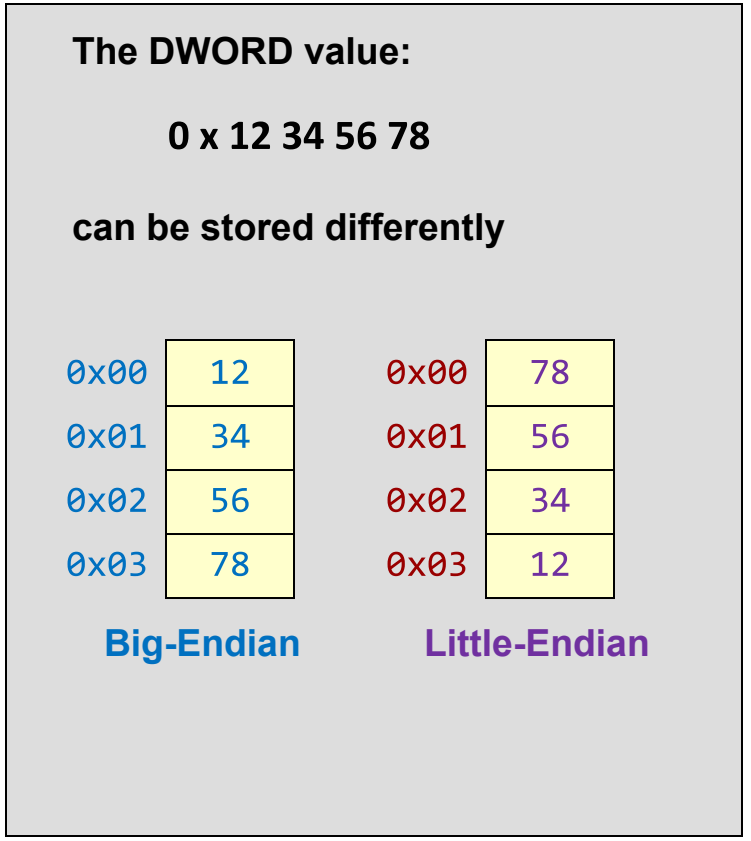
- To refer to a chunk of memory we must provide:
  - The starting address
  - The size: **B**, **W**, **L**, **Q**
- There are rules for valid starting addresses
  - **A valid starting address should be a multiple of the data size**
  - Words (2-byte chunks) must start on an even (divisible by 2) address
  - Double words (4-byte chunks) must start on an address that is a multiple of (divisible by) 4
  - Quad words (8-byte chunks) must start on an address that is a multiple of (divisible by) 8



Endian-ness

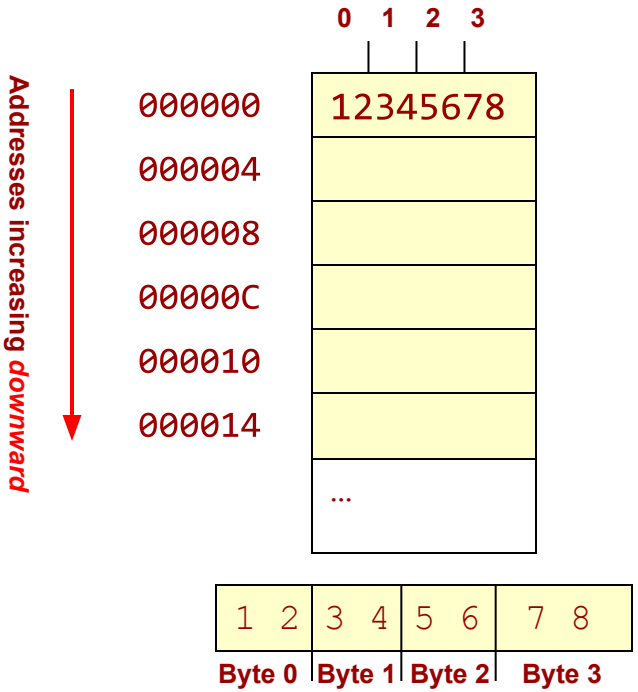
CS:APP 2.1.3

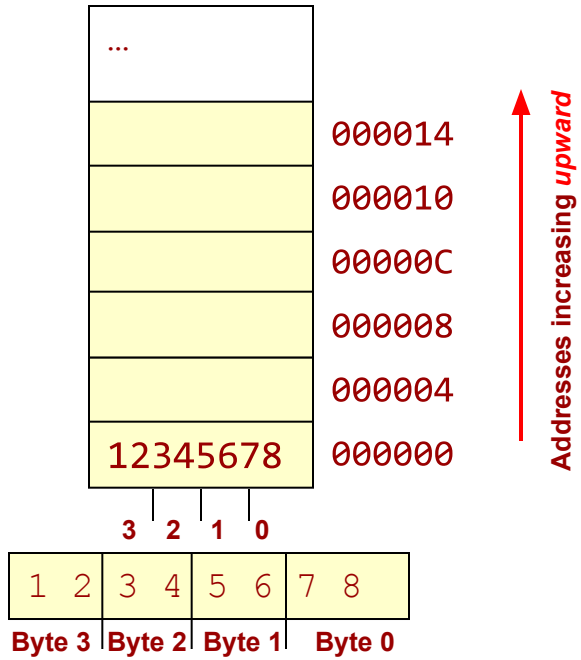
- **Endian-ness** refers to the two alternate methods of ordering the **bytes** in a larger unit (2, 4, 8 bytes)
  - **Big-Endian**
    - PPC, Sparc, TCP/IP
    - **MS byte is put at the starting address**
  - **Little-Endian**
    - used by Intel processors / original PCI bus
    - ***LS byte is put at the starting address***
- Some processors (like ARM) and busses can be configured for either big- or little-endian



Big-endian vs. Little-endian

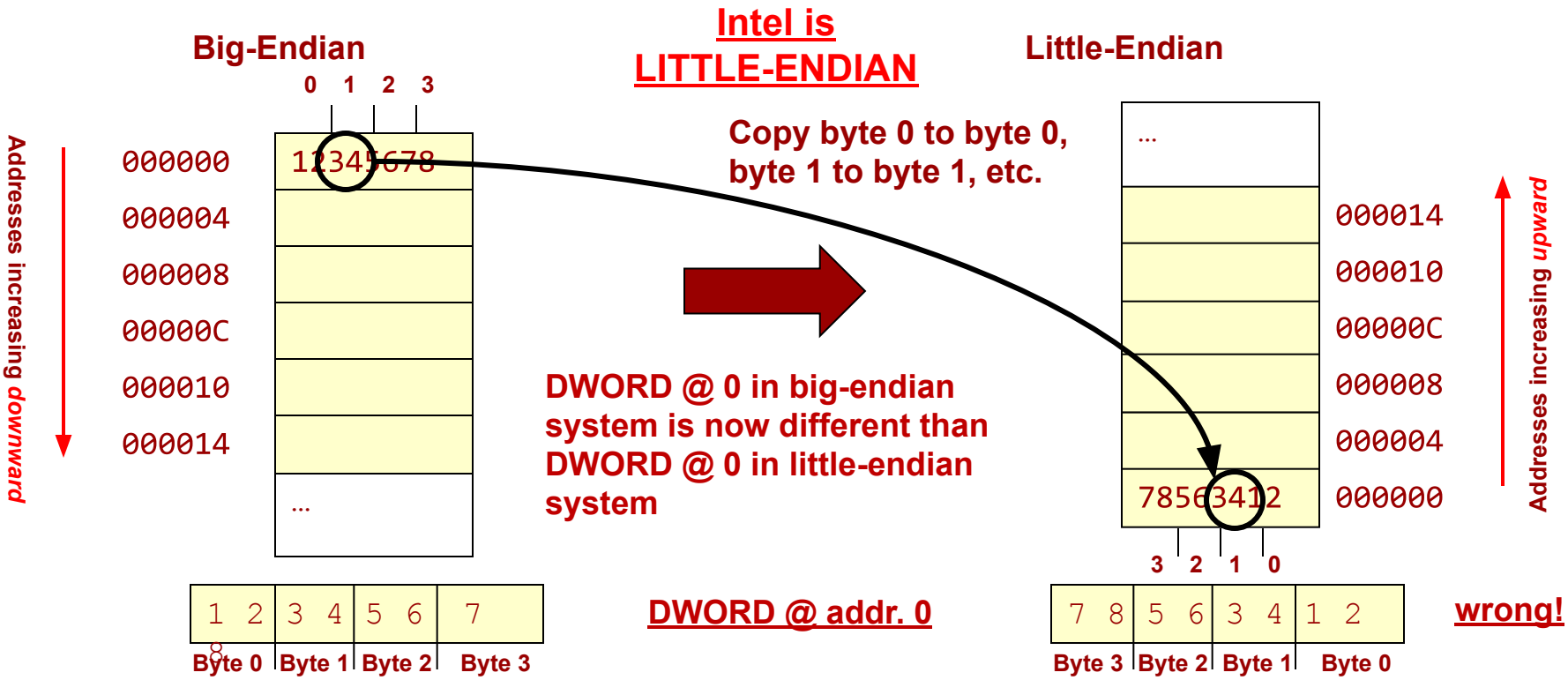
- Big-endian
  - makes sense if you view your memory as starting at the top-left and addresses increasing as you go down
- Little-endian
  - makes sense if you view your memory as starting at the bottom-right and addresses increasing as you go up





Big-endian vs. Little-endian Issues

- Issues arise when transferring data between different systems
  - Byte-wise copy of data from big-endian system to little-endian system
  - Major issue in networks (little-endian computer => big-endian computer) and even within a single computer (system memory => I/O device)



x86-64 ASSEMBLY

x86-64 Data Sizes

CS:APP 3.3

Integer

- 4 sizes
  - **Byte** (B)
    - 8-bits = 1 byte
  - **Word** (W)
    - 16-bits = 2 bytes
  - **Double Word** (L)
    - 32-bits = 4 bytes
  - **Quad Word** (Q)
    - 64-bits = 8 bytes

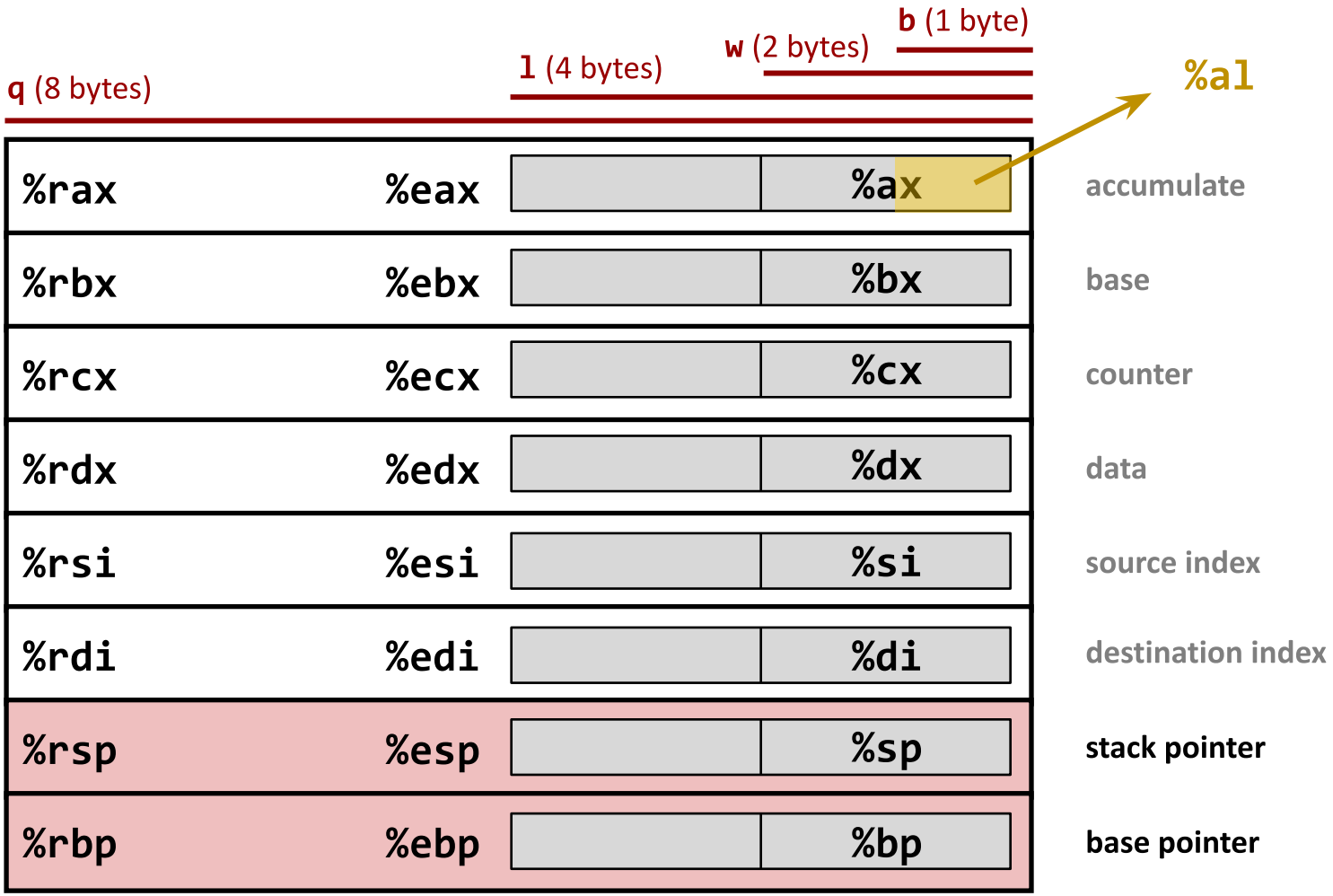
Floating Point

- 2 sizes
  - **Single** (S)
    - 32-bits = 4 bytes
  - **Double** (D)
    - 64-bits = 8 bytes
    - (For a 32-bit data bus, a double would be accessed from memory in 2 reads)

In x86-64, instructions generally specify what size data to access from memory and then operate upon.

x86-64 Register Names

CS:APP 3.4



- In addition: **%al**, **%bl**, **%cl**, **%dl**, **%sil**, **%dil**, **%spl**, **%bpl** for least significant byte
- In addition: **%r8** to **%r15** (**%r8d** / **%r8w** / **%r8b** for lower 4 / 2 / 1 bytes)

Intel x86 Register Set

- 8-bit processors in late 1970s
  - 4 registers for integer data: **A, B, C, D**
  - 4 registers for address/pointers: **SP** (stack pointer), **BP** (base pointer), **SI** (source index), **DI** (dest. index)
- 16-bit processors extended registers to 16-bits but continued to support 8-bit access!
  - Use prefix/suffix to indicate size:
    AL referenced the lower 8-bits of register A
    AH the higher 8-bits of register A
    AX referenced the full 16-bit value
- 32-/64-bit processors (see previous slide)

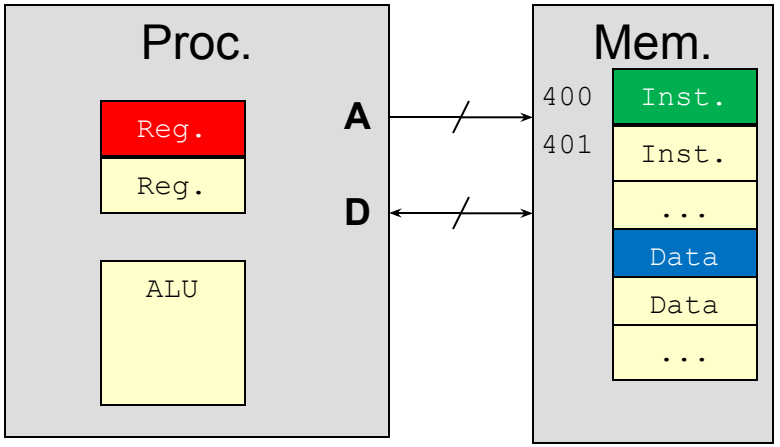
x86-64 Instruction Classes

- **Data Transfer**
  - `movq %rax,%rbx` means *"move rax into rbx"* akin to *"rbx = rax"*
  - Moves data **between registers**, or between **registers and memory** (One operand must be a processor register.)
  - Specifies **size via a suffix** on the instruction (`movb`, `movw`, `movl`, `movq`)
- **ALU Operations**
  - `addq %rax,%rbx` means *"add rax to rbx"* akin to *"rbx += rax"*
  - One operand must be a processor register or a constant
  - Size and operation specified by instruction (`addl`, `orq`, `andb`, `subw`)
- **Control / Program Flow**
  - `cmp %rax,%rbx` followed by `jl L1` means *"jump to label L1 if rbx < rax"*
  - Unconditional (`jmp L1`)
  - Conditional Branch (`cmpq/test x,y` followed by `je`, `jne`, `jl`, `jge L1`)
  - Subroutine Calls (`call` *procedure*, `ret`)

Privileged / System Instructions: can only be used by OS or other "supervisor" software (e.g. `int` to access certain OS capabilities, etc.)

Operand Locations

- **Source** operands must be in one of the following 3 locations:
  - A register value (e.g. **%rax**)
    addq %rax, ...
  - A value in a memory location (e.g. value at address **0x0200e8**)
    addq 0x0200e8, ...
  - A constant stored in the instruction itself (known as "immediate value")
    addq $1, ...
    addq $0x0200e8, ...
    ($ indicates the constant/immediate)
- **Destination** operands must be
  - A register: **addq ...,%rax**
  - A memory location: **addq ...,0x0200e8** (specified by its address 0x0200e8)



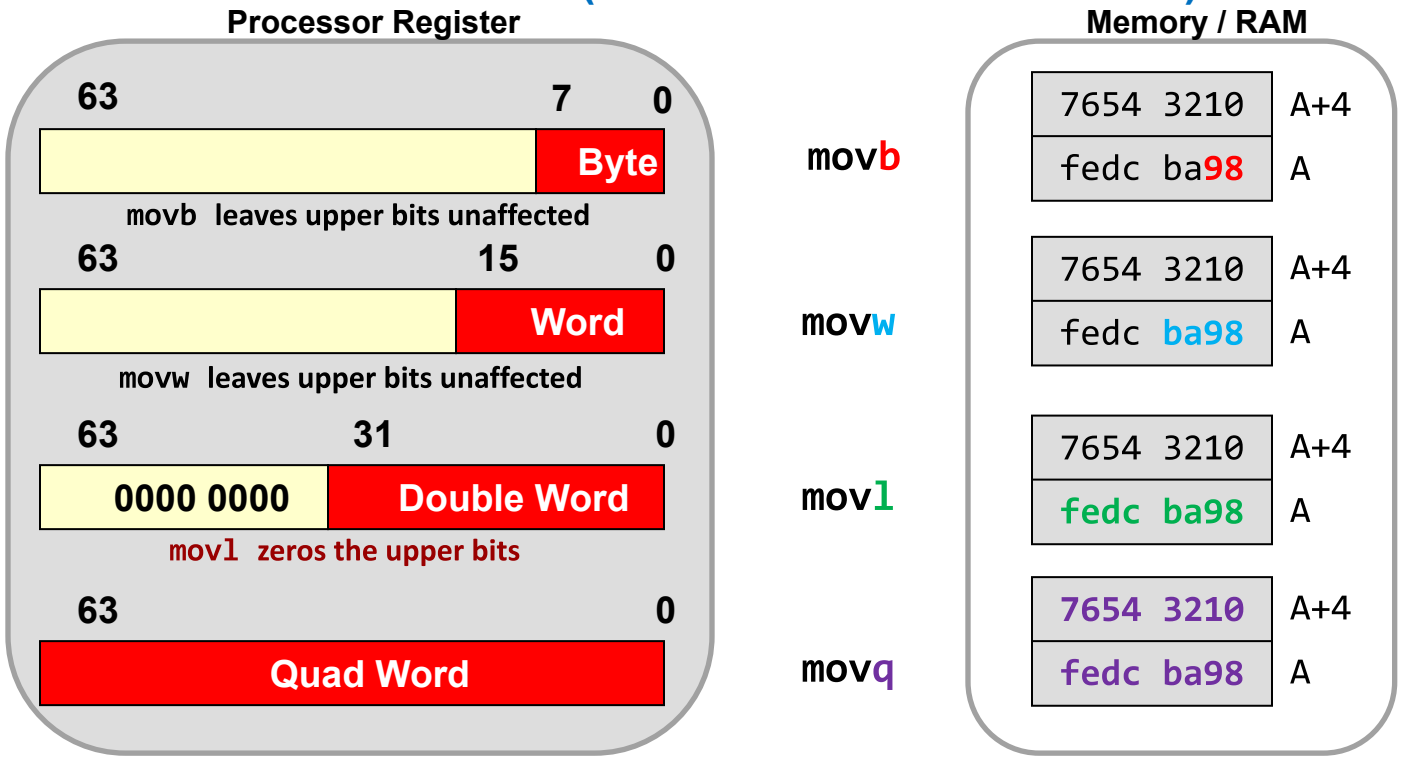
DATA TRANSFER INSTRUCTIONS

mov Instruction & Data Size

CS:APP 3.4.2

- **Moves data between memory and processor register**
- Always provide the **LS-Byte address (little-endian)** of the desired data
- Size is explicitly defined by the instruction suffix ('mov[bwlq]') used
- Recall: Start address **should** be divisible by size of access

(Assume start address = A)



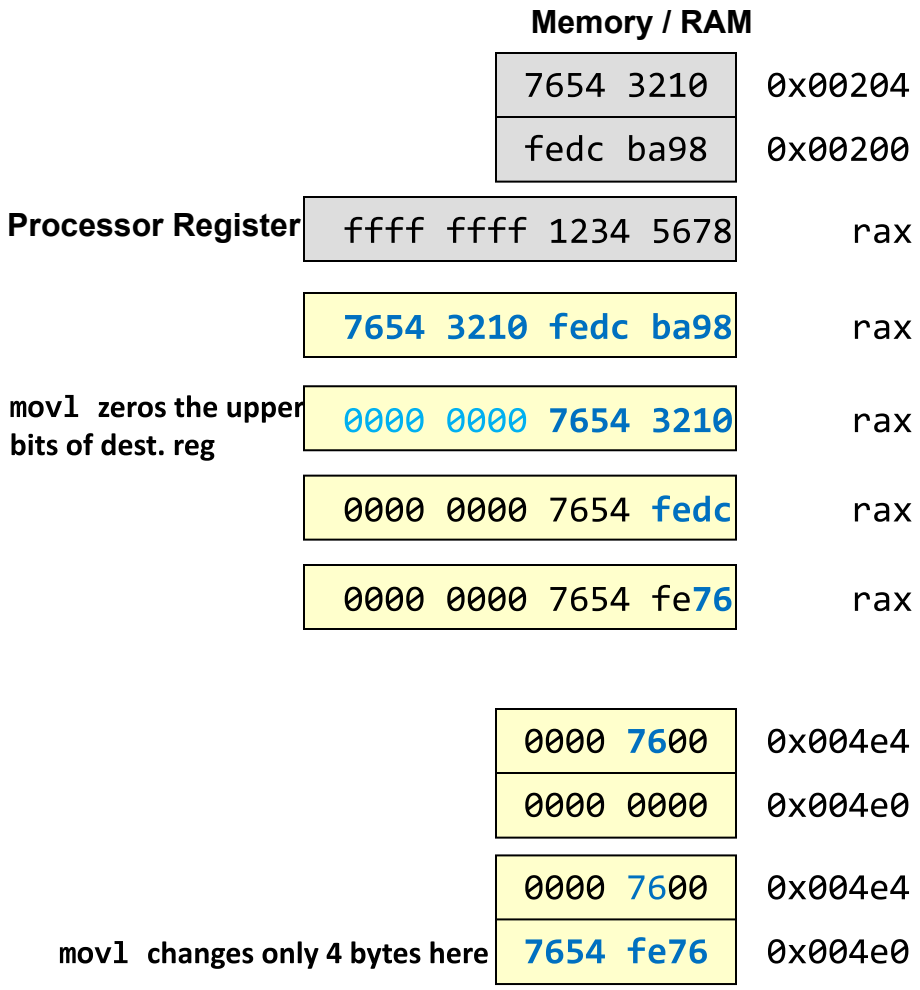
Mem/Register Transfer Examples

`mov[b,w,l,q] src, dst`

- Initial Conditions:
  - `movq 0x200, %rax`
  - `movl 0x204, %eax`
  - `movw 0x202, %ax`
  - `movb 0x207, %al`

  - `movb %al, 0x4e5`

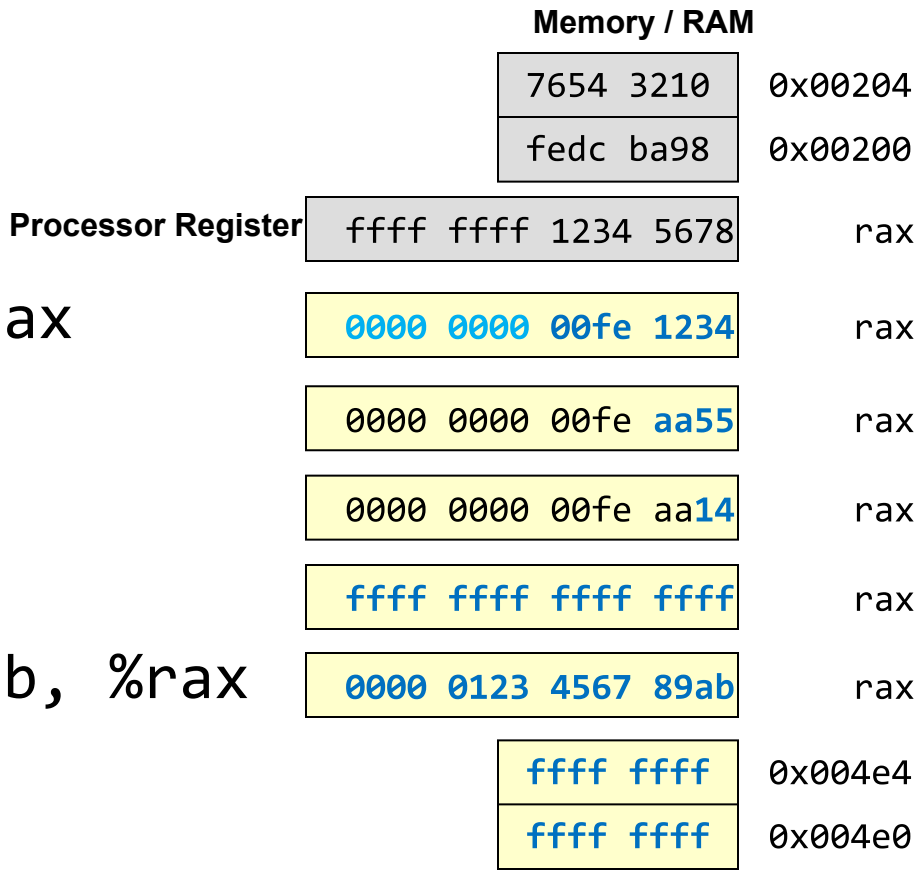
  - `movl %eax, 0x4e0`



Treat these instructions as a sequence where one affects the next.

Immediate Examples

- Immediate Examples
  - `movl $0xfe1234, %eax`
  - `movw $0xaa55, %ax`
  - `movb $20, %al`
  - `movq $-1, %rax`
  - `movabsq $0x123456789ab, %rax`
  - `movq $-1, 0x4e0`



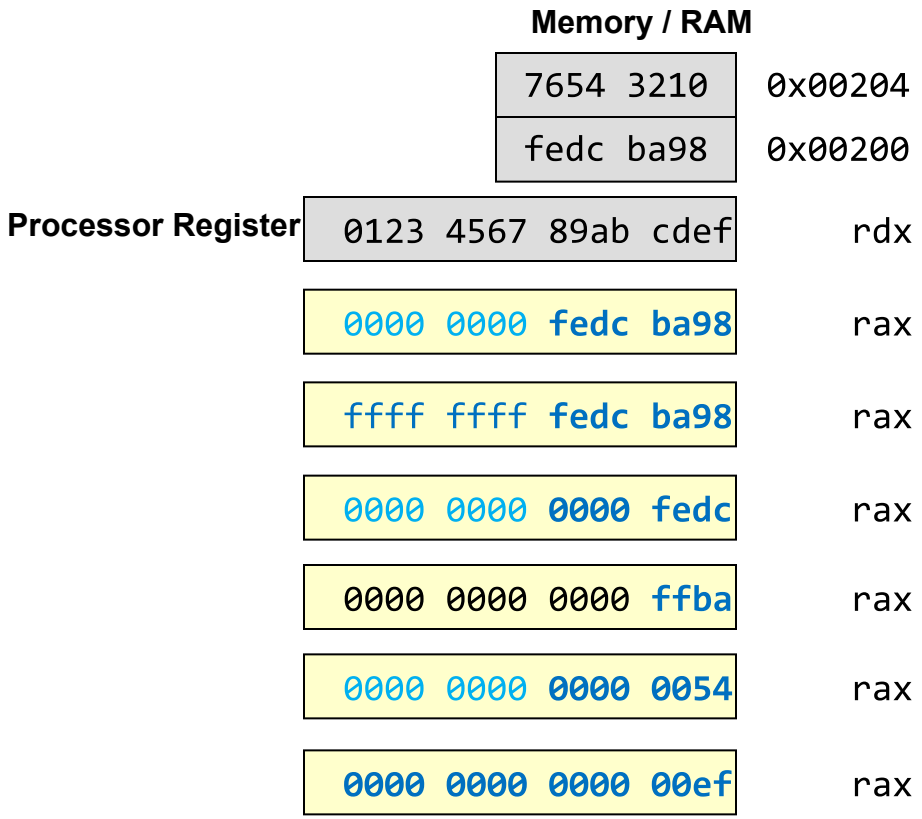
Rules:
- Immediates must be source operand
- Indicate with '$' and can be specified in decimal (default) or hex (start with 0x)
- movq can only support a 32-bit immediate (and will then sign-extend that value to fill the upper 32-bits)
- Use movabsq for a full 64-bit immediate value

Variations: Zero / Sign Extension

- There are several variations with register destination
  - Used to zero-extend or sign-extend the source
- Normal `mov` **does not affect upper portions** of registers (with exception of `movl`)
- `movz`*xy* will zero-extend the upper portion (up to size y)
  - `movzbw` (move a byte from the source but zero-extend it to a word in the destination register)
  - `movzbw, movzbl, movzbq, movzwl, movzwq` (but no `movzlq`!)
- `movs`*xy* will sign-extend the upper portion (up to size y)
  - `movsbw` (move a byte from the source but sign-extend it to a word in the destination register)
  - `movsbw, movsbl, movsbq, movswl, movswq, movslq`
  - `cltq` is equivalent to `movslq %eax,%rax` (but shorter encoding)

Zero/Signed Move Variations

- Initial Conditions:



- `movl 0x200, %eax`
- `movslq 0x200, %rax`
- `movzwl 0x202, %eax`
- `movsbw 0x201, %ax`
- `movsbl 0x206, %eax`
- `movzbq %dl, %rax`

Treat these instructions as a sequence where one affects the next.

Remember: Zero/Sign Extension

- Extension is the process of increasing the number of bits used to represent a number **without changing its value**

Unsigned = Zero Extension (**always add leading 0's**):

111011 = **00111011**

Increase a 6-bit number to 8-bit number by zero extending

2's complement = Sign Extension (**replicate sign bit**):

positive **011010** = **00011010**

negative **110011** = **11110011**

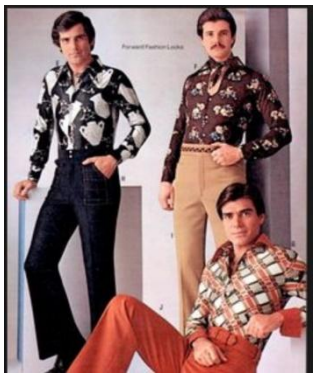
Sign bit is just repeated as many times as necessary

Why does it work?

111… = -128 + 64 + 32 = -32 and 1… = -32

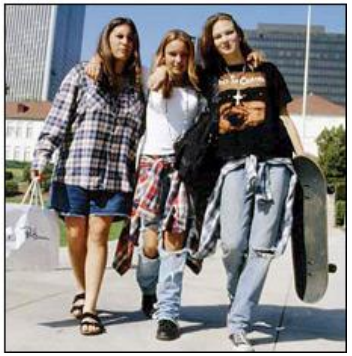
Why So Many Oddities & Variations

- The x86 instruction set has been around for nearly 40 years and each new processor has had to maintain backward compatibility (support the old instruction set) while adding new functionality
- If you wore one clothing article from each decade you'd look funny too and have a lot of oddities



70s

80s



90s

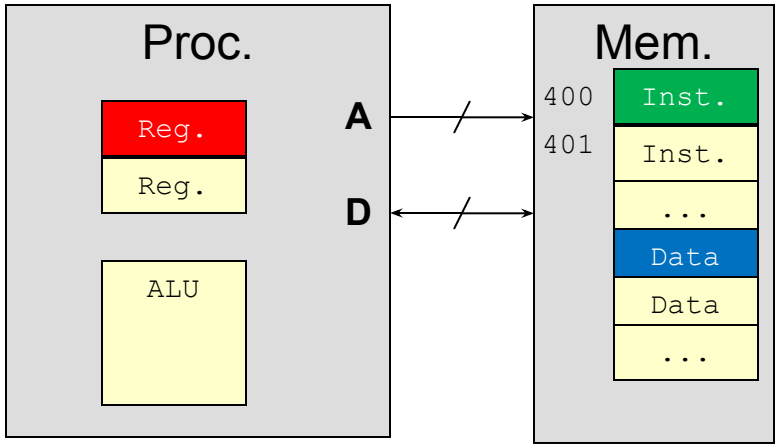
Summary

- Access to different size portions of a register requires different names in x86 (e.g. %al, %ax, %eax, %rax)
- Moving to a register may involve extension
  - 32-bit moves always set the upper 32 bits to 0
- Moving to memory never involves zero- or sign-extending since memory is broken into finer granularities

ADDRESSING MODES

What Are Addressing Modes

- Recall an operand must be:
  - A register value (e.g., **`%rax`**)
  - An immediate value (e.g., **`$0x42`**)
  - A value from memory (e.g., **`0x42`**)
- To access a **memory location** we must supply an address
  - However, there can be **many ways** to compute an address, each useful in particular contexts
    - Accessing an array element a[i] vs. object member obj.member
- Ways to specify operand locations are known as addressing modes



Addressing Modes

Different ways to specify source values and output location.

Immediate: `$imm` to use a constant input value, e.g., `$0xFF` or `$255`

Register: `%reg` to use the value contained in a register, e.g., `%rax`

Memory reference

- **Absolute**: `addr`, e.g., `0x1122334455667788` [use a fixed address]
- **Indirect**: `(%reg)`, e.g., `(%rax)` [use address contained in a **q register**]
- **Base+displacement**: `imm(%reg)`, e.g., `16(%rax)` [add a displacement]
- **Indexed:** `(%reg1,%reg2)`, e.g., `(%rax,%rbx)` [add another register]
- **Indexed+displacement:** `imm(%reg1,%reg2)` [add both]
- **Scaled indexed:** `imm(%reg1,%reg2,c)` [use address: `imm+reg1+reg2*c`]

  Restriction: `c` must be one of 1, 2, 4, 8

  Variants: omit **`imm`** or **`reg1`** or **both**. E.g., `(,%rax,4)`

(A memory reference selects the first byte.)

Common x86-64 Addressing Modes

CS:APP 3.4.1

Name	Form	Example	Description
Immediate	$imm	movq $-500,%rax	R[rax] = imm.
Register	r_a	movq %rdx,%rax	R[rax] = R[rdx]
Direct Addressing	imm	movq 2000,%rax	R[rax] = M[2000]
Indirect Addressing	(r_a)	movq (%rdx),%rax	R[rax] = M[R[r_a]]
Base w/ Displacement	$imm(r_b)$	movq 40(%rdx),%rax	R[rax] = M[R[r_b]+40]
Scaled Index	$(r_b,r_i,s†)$	movq (%rdx,%rcx,4),%rax	R[rax] = M[R[r_b]+R[r_i]*s]
Scaled Index w/ Displacement	$imm(r_b,r_i,s†)$	movq 80(%rdx,%rcx,2),%rax	R[rax] = M[80 + R[r_b]+R[r_i]*s]

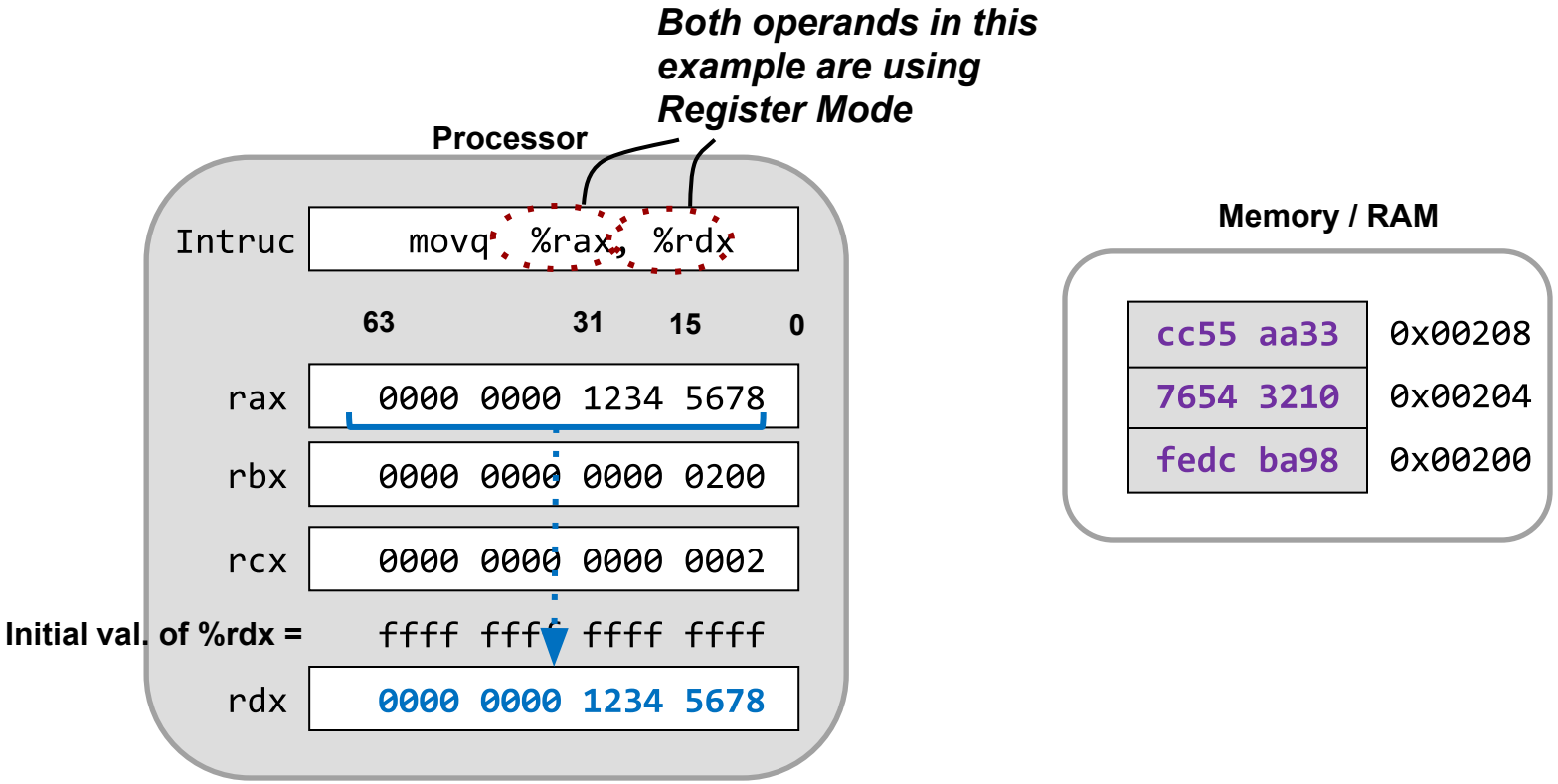
†Known as the scale factor and can be **{1,2,4, or 8}**

Imm = Constant, R[*x*] = Content of register *x*, M[addr] = Content of memory @ addr.

Purple values = ***effective address (EA)*** *= Actual address used to get the operand*

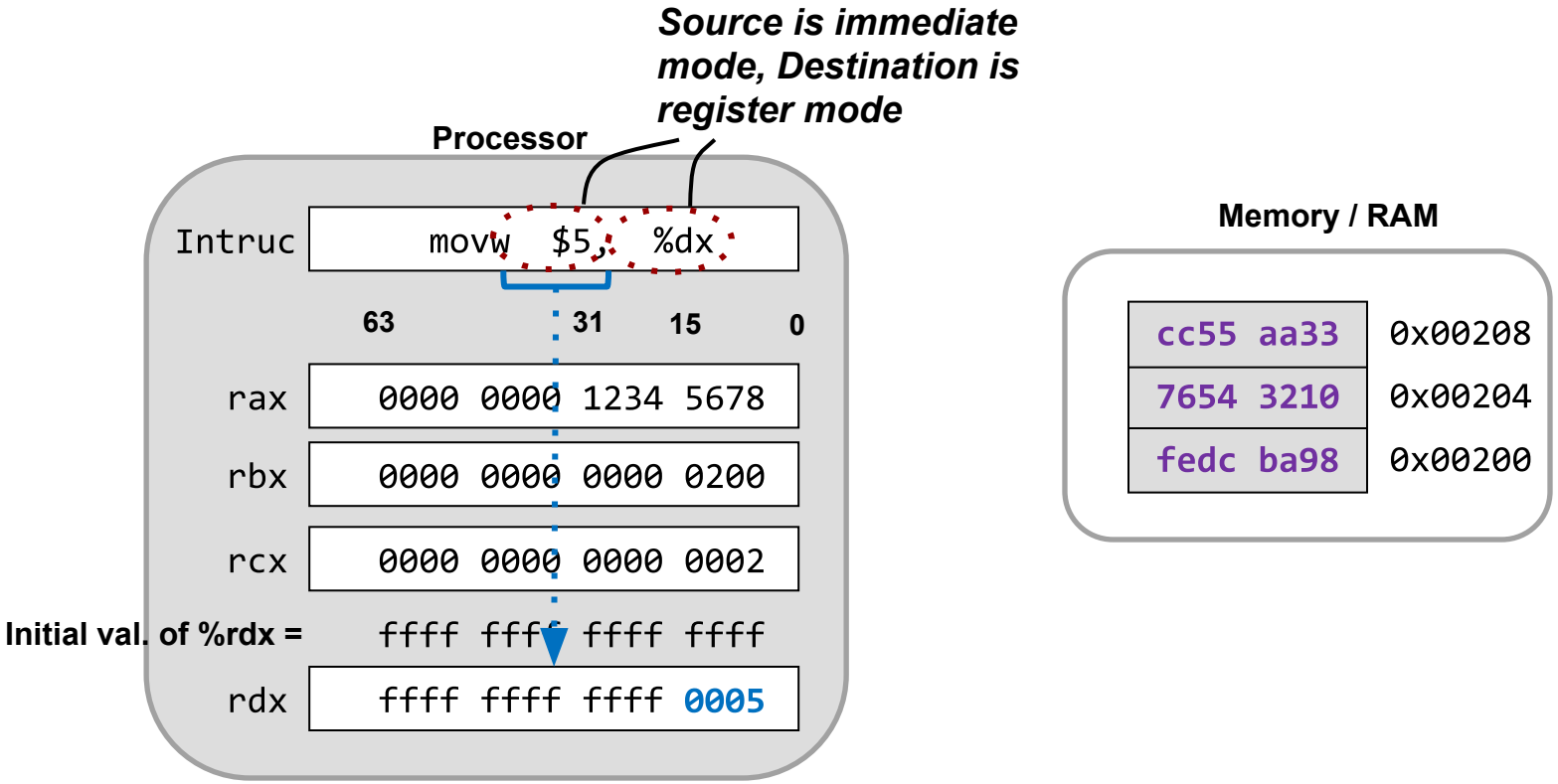
Register Mode

- Specifies the contents of a register as the operand



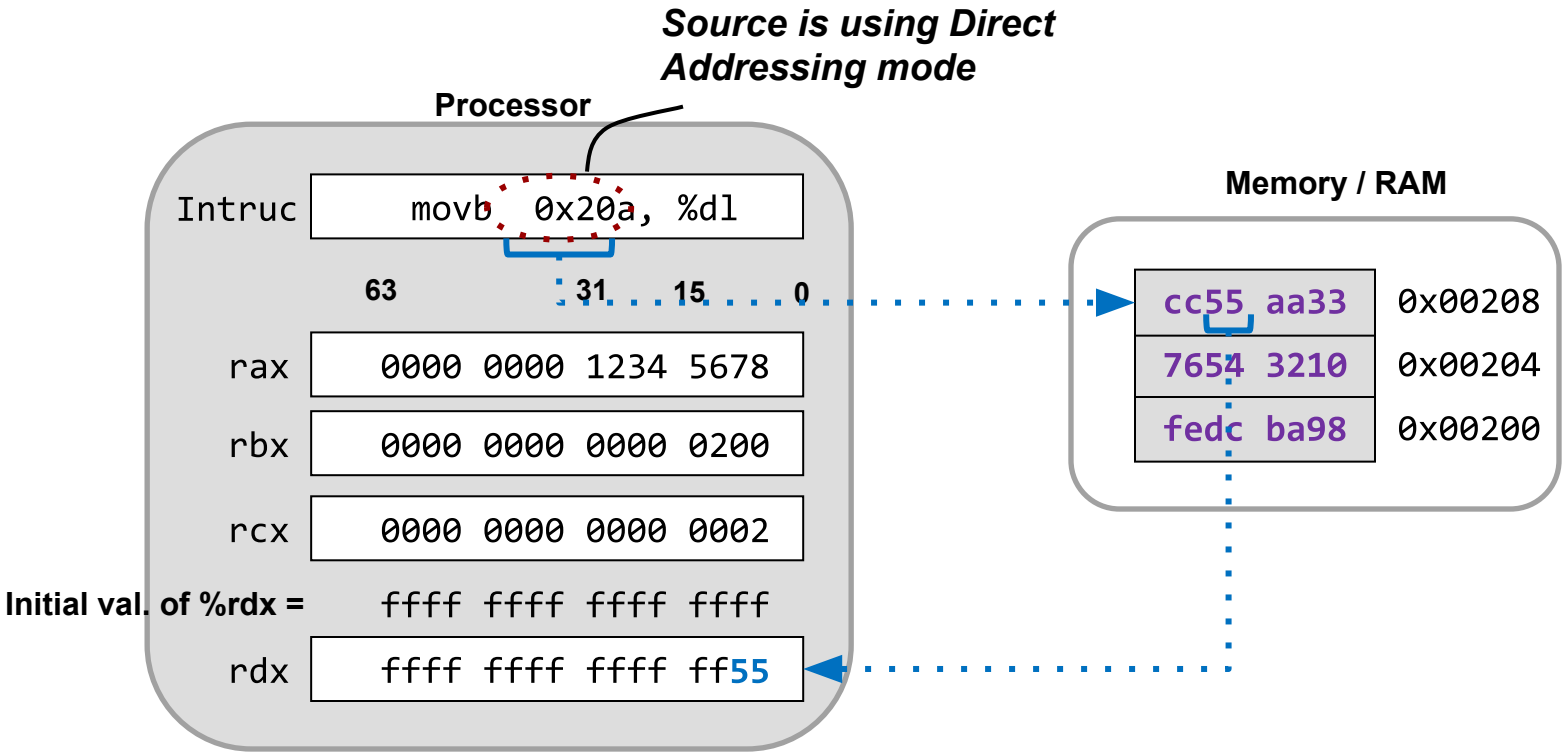
Immediate Mode

- Specifies a constant *stored in the instruction* as the operand
- Immediate is denoted with '$' and can be in hex or decimal



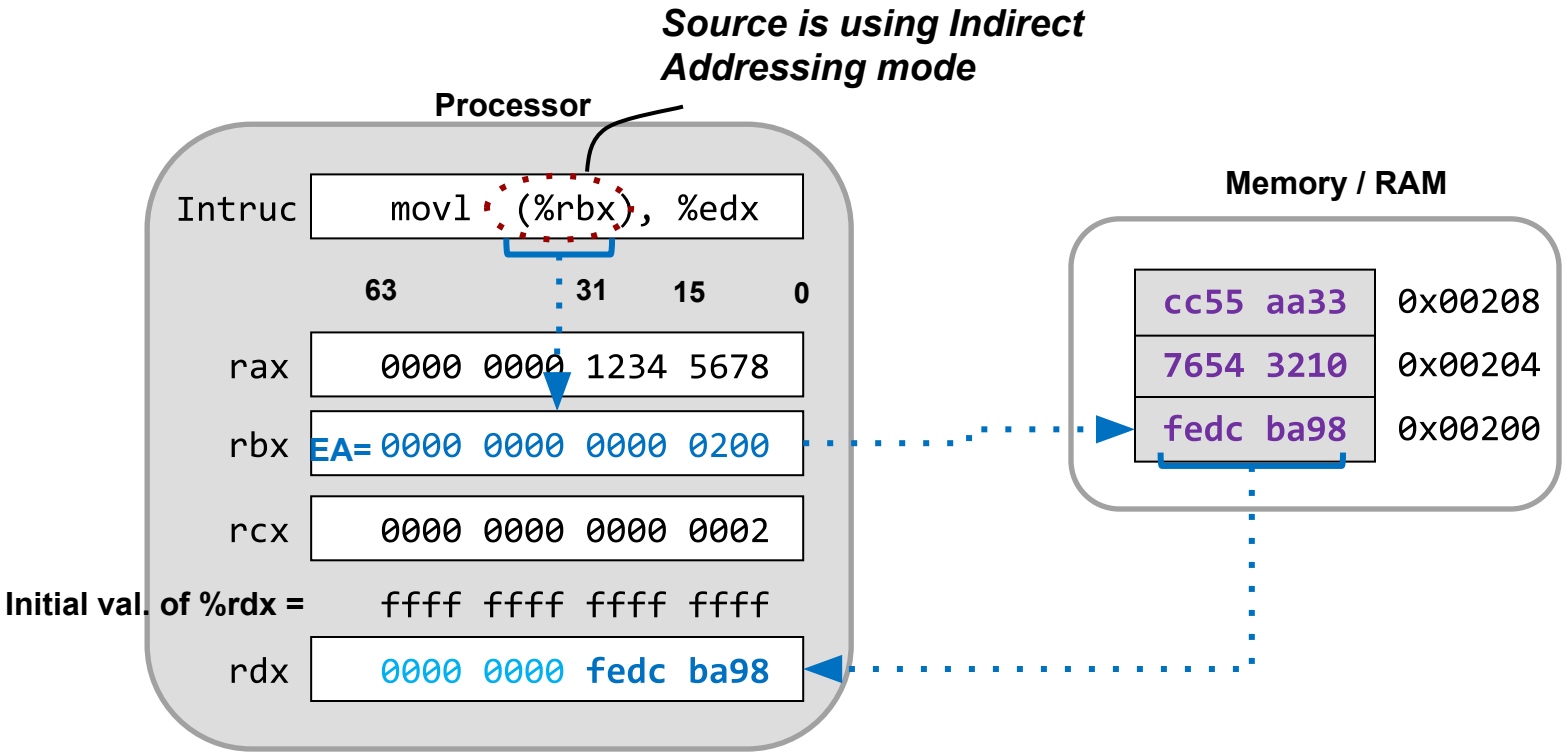
Direct Addressing Mode

- Use the operand located at a constant memory address *stored in the instruction*
- Address can be specified in decimal or hex



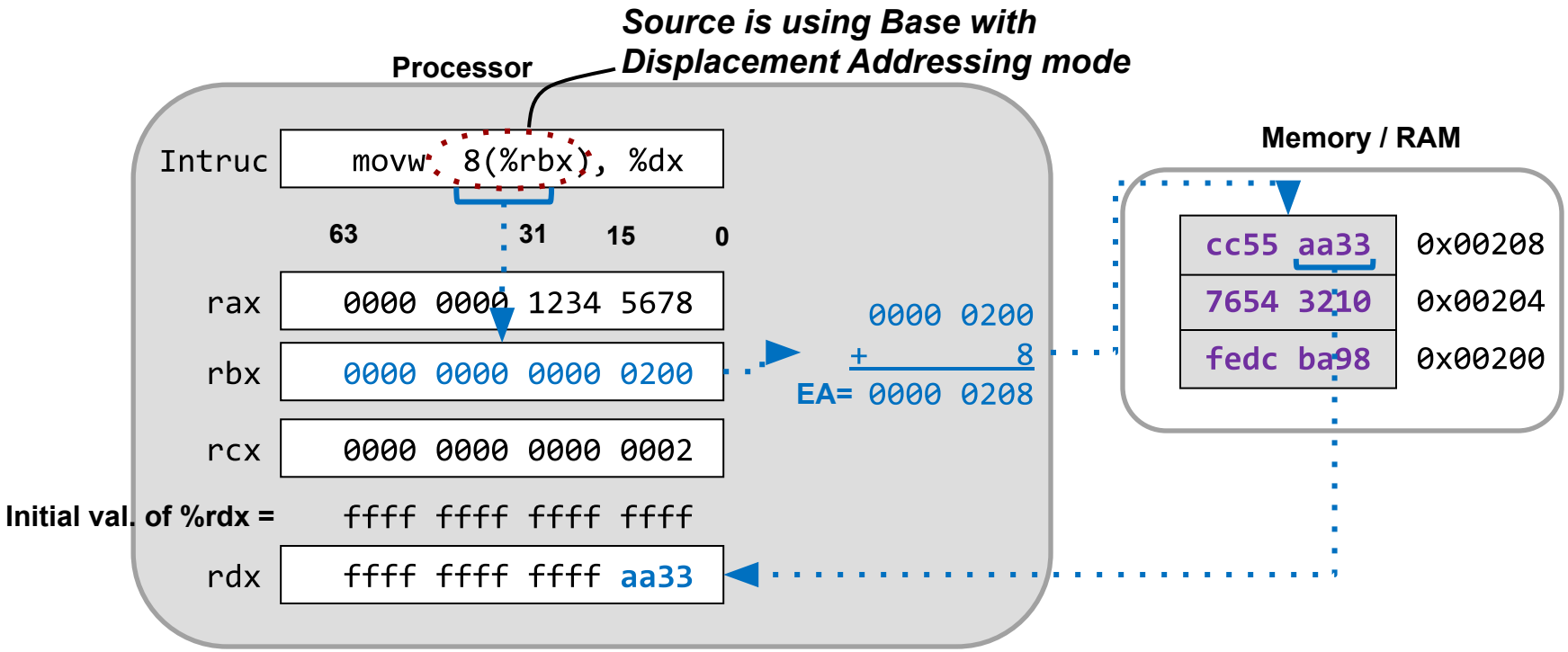
Indirect Addressing Mode

- Use the operand located at a memory address contained in a register (similar to dereferencing a pointer in C)
- Parentheses indicate indirect addressing mode



Indirect with Displacement

- Use the operand located at address (register value + constant)
- Form: d(%reg)



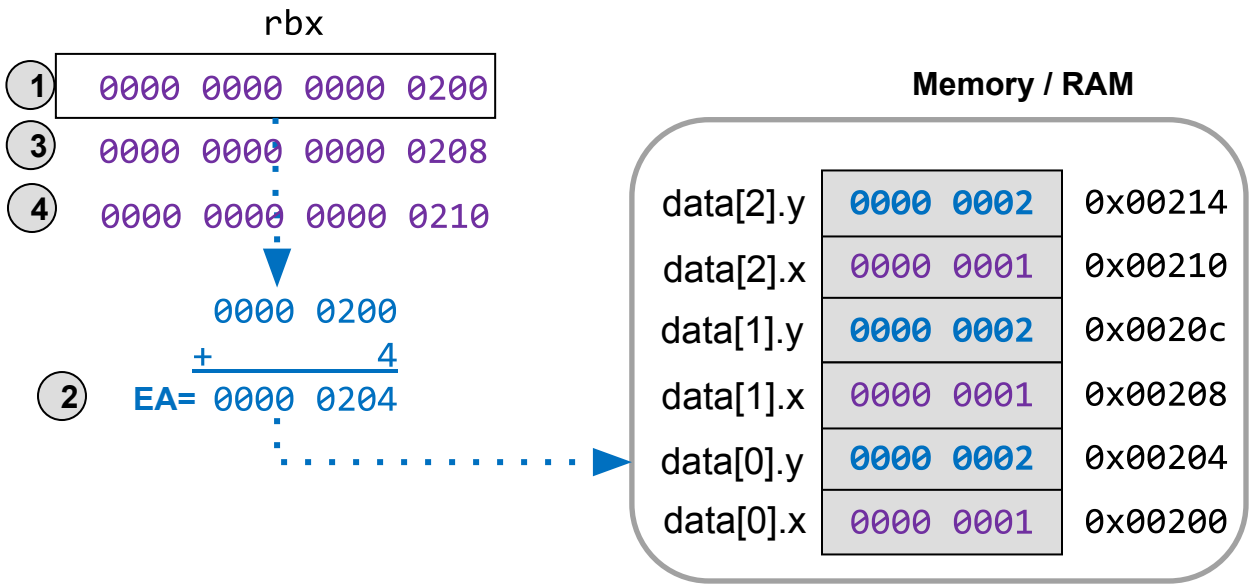
Indirect with Displ. Example

- Useful for access members of a struct or object

```
struct mystruct {
  int x;
  int y;
};
struct mystruct data[3];

int main()
{
  for(int i=0; i<3; i++){
    data[i].x = 1;
    data[i].y = 2;
  }
}
```

C Code

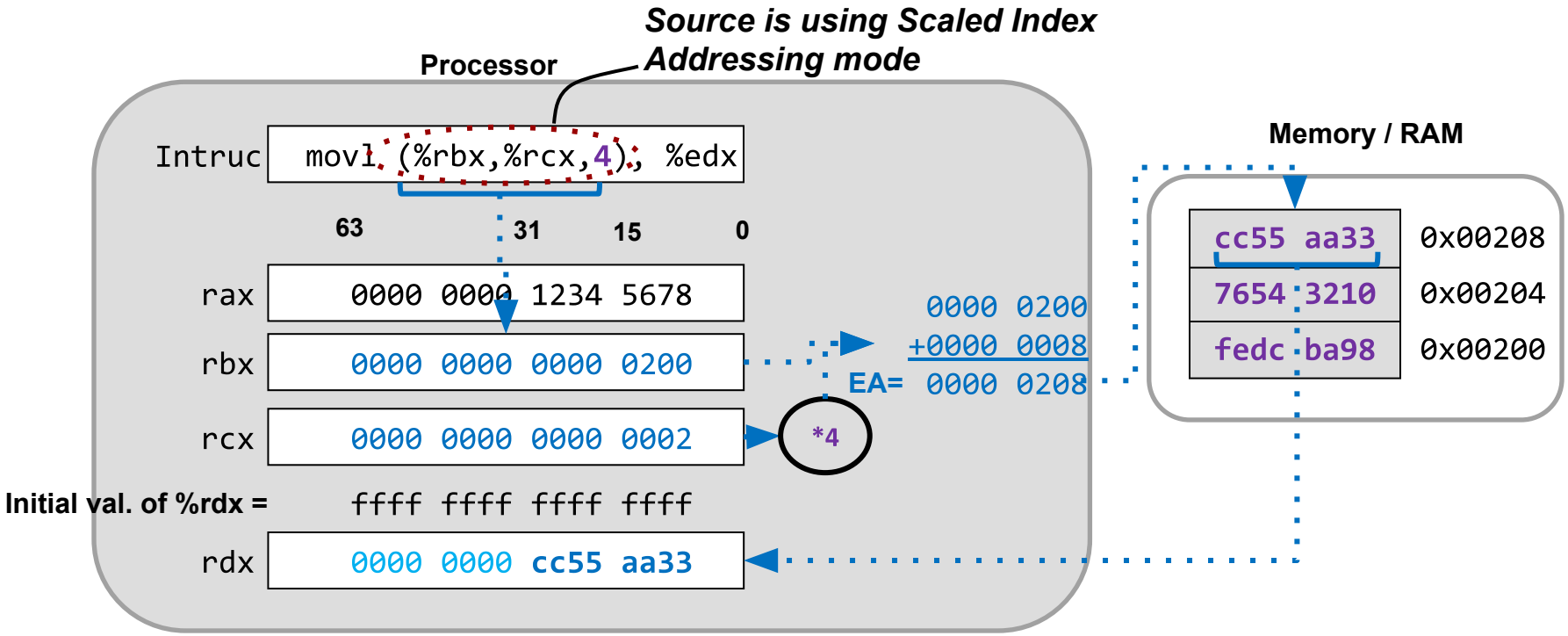



```
movq    $0x0200,%rbx
Loop 3 times {
1 3 4 movl  $1, (%rbx)
2     movl  $2, 4(%rbx)
      addq  $8, %rbx
}
```

Assembly

Base with Scaled Index Addressing Mode

- Form: **(%reg1,%reg2,s)** [s = 1, 2, 4, or 8]
- Uses the result of "**%reg1 + %reg2*s**" as the effective address of the actual operand in memory



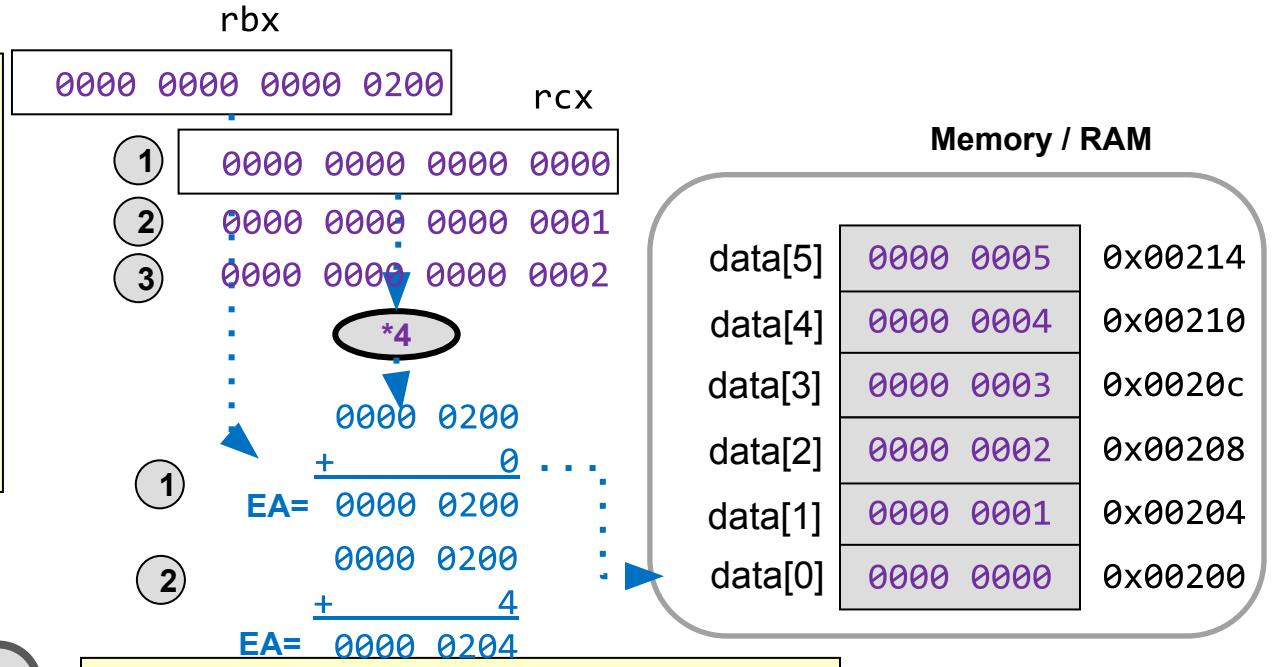
Base with Scaled Index Example

- Useful for accessing array elements

```
int data[6];

int main()
{
  for(int i=0; i<6; i++){
    data[i] = i;
 // *(startCharPtr+4*i) = i;
  }
}
```

C Code



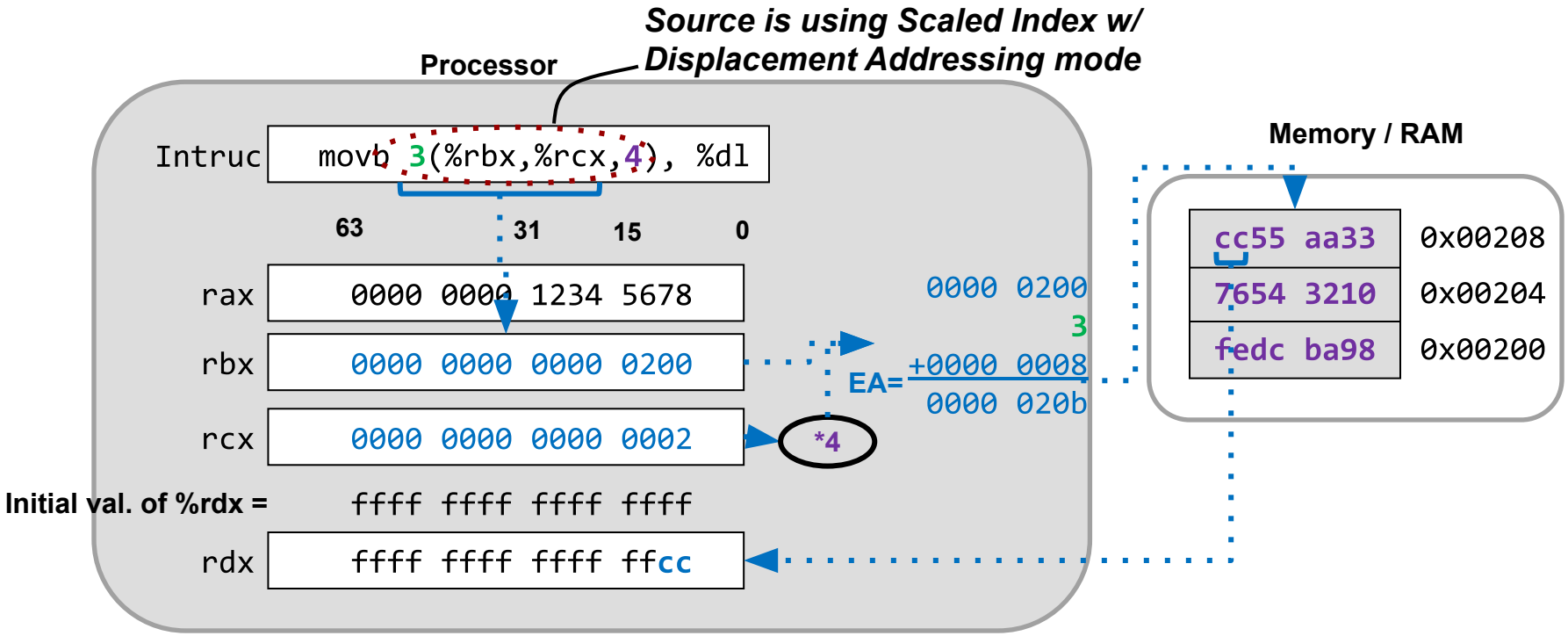
Array of:
- chars/bytes => Use s=1
- shorts/words => Use s=2
- ints/floats/dwords => Use s=4
- long longs/doubles/qwords => Use s=8

```
movq    $0x0200,%rbx
movq    $0, %rcx
Loop 6 times {
  movl  %ecx, (%rbx,%rcx,4)
  addq  $1, %rcx
}
```

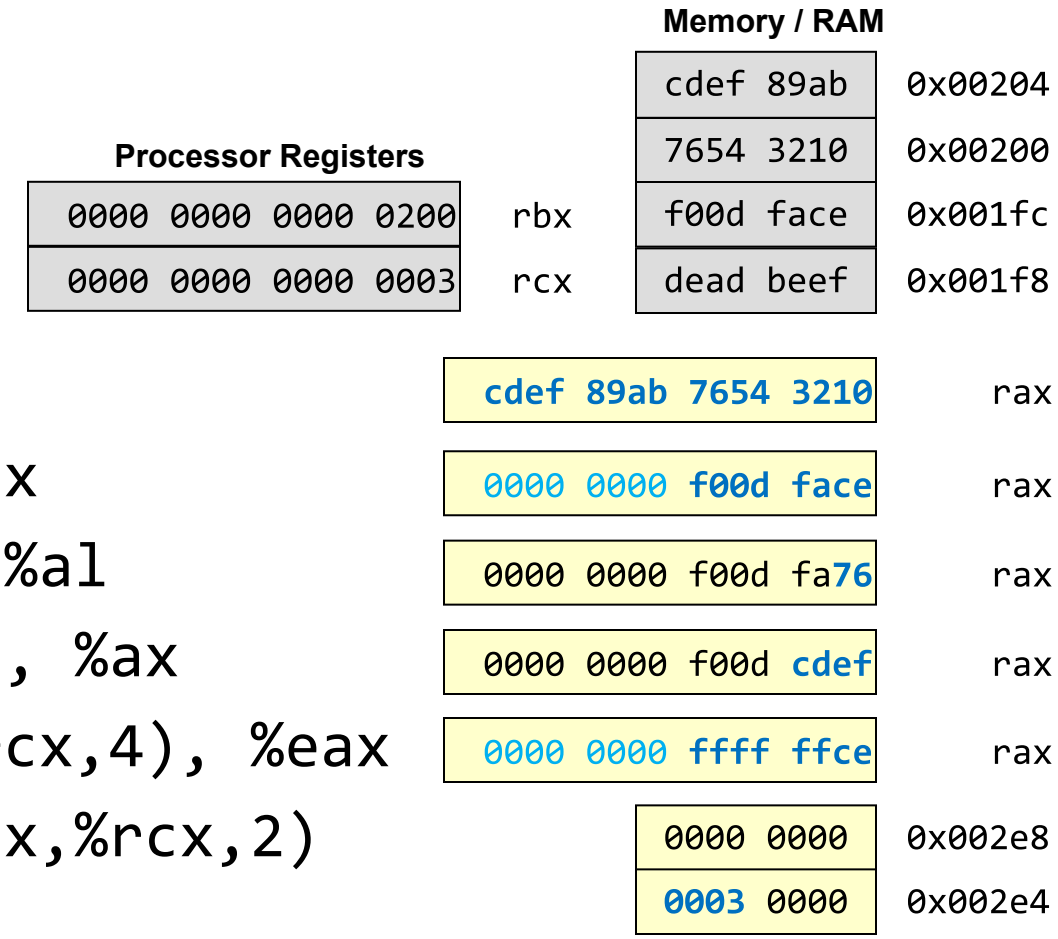
Assembly

Base and Scaled Index *with Displacement*

- Form: **`d(%reg1,%reg2,s)`** [s = 1, 2, 4, or 8]
- Uses the operand located at EA: **`d + %reg1 + %reg2*s`**



Addressing Mode Exercises



Treat these instructions as a sequence where one affects the next.

Quiz

Values at each memory address:

- `0x100: 0xFF`
- `0x104: 0xAB`
- `0x108: 0x13`
- `0x10C: 0x11`

Values in registers:

- `%rax: 0x100`
- `%rcx: 0x1`
- `%rdx: 0x3`

Consider these in `movl ... , %ebx`

- `%eax`
- `0x104`
- `$0x108`
- `(%rax)`
- `4(%rax)`
- `9(%rax,%rdx)`
- `260(%rcx,%rdx)`
- `0xFC(,%rcx,4)`
- `(%rax,%rdx,4)`

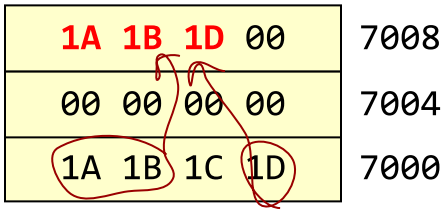
What goes in `%ebx` is:

- `0x100`
- `0xAB`
- `0x108`
- `0xFF`
- `0XAB`
- `0x11`
- `0x13`
- `0xFF`
- `0x11`

Addressing Mode Examples

		%eax	%ecx	%edx
1	movl $0x7000,%eax	0x0000 7000		
2	movl $2,%ecx		0x0000 0002	
3	movb (%rax),%dl			0x???? ??1d
4	movb %dl,9(%rax)			
5	movw (%rax,%rcx),%dx			0x???? 1a1b
6	movw %dx,6(%rax,%rcx,2)			

Main Memory



Instruction Limits on Addressing Modes

- To make the HW faster and simpler, there are restrictions on the combination of addressing modes
  - Aids overlapping the execution of multiple instructions
- **Not allowed: memory locations for both operands**
  - `movl 2000, (%rax)` is not allowed
  - `movl (%rax), (%rax)` is not allowed
  - `addl (%rax), (%rax)` is not allowed
  - To move mem->mem use two move instructions with a register as the intermediate storage location
- Legal move combinations:
  - `Imm -> Reg`
  - `Imm -> Mem`
  - `Reg -> Reg`
  - `Reg -> Mem`
  - `Mem -> Reg`

Quiz: Spot the Mistake

These are all wrong! Why?

```
movb $0xF, (%ebx)
movl %rax, (%rsp)
movw (%rax), 4(%rsp)
movb %al, %sl
movq %rax, $0x123
movl %eax, %rdx
movb %si, 8(%rbp)
```

ARITHMETIC INSTRUCTIONS

At a glance

Unary (with q / l / w / b variants)

- `incq x` is equivalent to `x++`
- `decq x` is equivalent to `x--`
- `negq x` is equivalent to `x = -x`
- `notq x` is equivalent to `x = ~x`

Binary (with q / l / w / b variants)

- `addq x,y` is equivalent to `y += x`
- `subq x,y` is equivalent to `y -= x`
- `imulq x,y` is equivalent to `y *= x`
- `andq x,y` is equivalent to `y &= x`
- `orq x,y` is equivalent to `y |= x`
- `xorq x,y` is equivalent to `y ^= x`
- `salq n,y` is equivalent to `y = y << n`
- `sarq n,y` is equivalent to `y = y >> n`
- `shrq n,y` is equivalent to `y = y >> n`

imulq works for signed/unsigned but keeps only the least significant half of the result … more on this later

Any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

Except for right shift, all instructions are the same for signed/unsigned values (thanks to 2's-complement)

n is $imm or %cl (mod 32 or 64)

arithmetic: fill in sign bit from left

logical: fill in zeros from left

ALU Instructions

CS:APP 3.5

- Perform arithmetic/logic operation on the given size of data
- Restriction: Both operands cannot be memory
- Format
  - `add[b,w,l,q] src2, src1/dst`

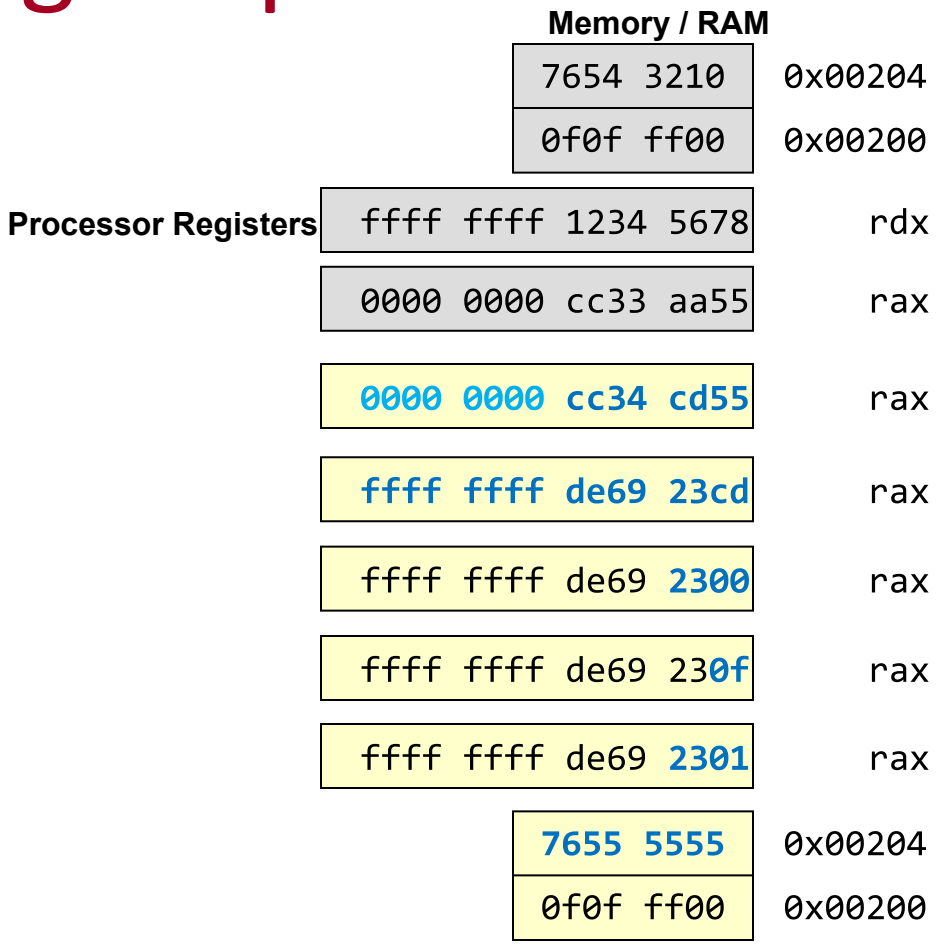
Work from right->left->right

- Examples
  - `addq %rbx, %rax  (%rax += %rbx)`
  - `subq %rbx, %rax  (%rax -= %rbx)`

Arithmetic/Logic Operations

Initial Conditions:

- addl $0x12300, %eax
- addq %rdx, %rax
- andw 0x200, %ax
- orb 0x203, %al
- subw $14, %ax
- addl $0x12345, 0x204



Consider the instructions as a sequence. Rules:

- addl, subl, etc. zero out the upper 32-bits
- addq, subq, etc. can only support a 32-bit immediate (they sign-extend that value to fill the upper 32-bits)

Arithmetic and Logic Instructions

C operator	Assembly	Notes
+	`add[b,w,l,q]  src1,src2/dst`	`src2/dst += src1`
-	`sub[b,w,l,q]  src1,src2/dst`	`src2/dst -= src1`
&	`and[b,w,l,q]  src1,src2/dst`	`src2/dst &= src1`
|	`or[b,w,l,q]  src1,src2/dst`	`src2/dst |= src1`
^	`xor[b,w,l,q]  src1,src2/dst`	`src2/dst ^= src1`
~	`not[b,w,l,q]  src/dst`	`src/dst = ~src/dst`
-	`neg[b,w,l,q]  src/dst`	`src/dst = (~src/dst) + 1`
++	`inc[b,w,l,q]  src/dst`	`src/dst += 1`
--	`dec[b,w,l,q]  src/dst`	`src/dst -= 1`
* (signed)	`imul[b,w,l,q]  src1,src2/dst`	`src2/dst *= src1`
<< (signed)	`sal  cnt, src/dst`	`src/dst = src/dst << cnt`
<< (unsigned)	`shl  cnt, src/dst`	`src/dst = src/dst << cnt`
>> (signed)	`sar  cnt, src/dst`	`src/dst = src/dst >> cnt`
>> (unsigned)	`shr  cnt, src/dst`	`src/dst = src/dst >> cnt`
==, <, >, <=, >=, != (src2 ? src1)	`cmp[b,w,l,q]  src1, src2` `test[b,w,l,q]  src1, src2`	`cmp` performs: src2 – src1 `test` performs: src1 & src2

`lea` Instruction

CS:APP 3.5.1

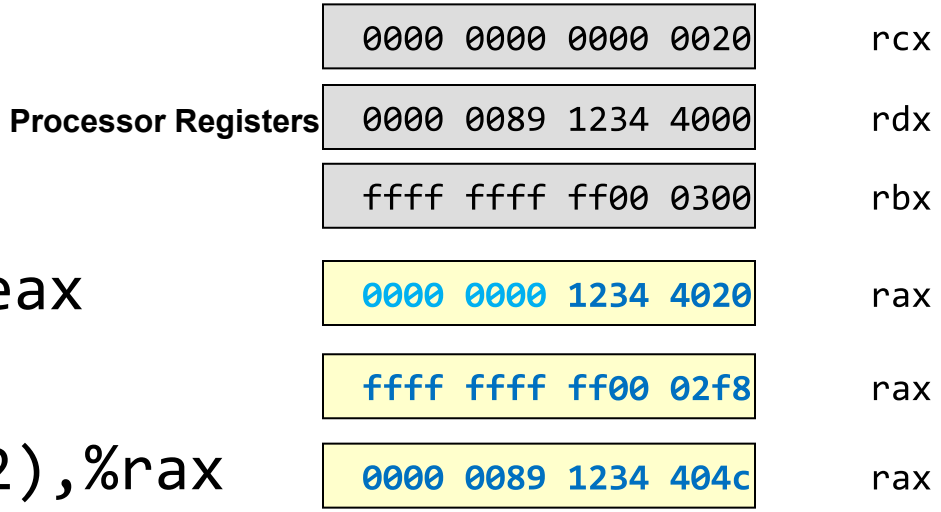
- Recall the exotic addressing modes supported by x86

Scaled Index w/ Displacement	`imm(`r_b`,`r_i`,s)`	`movq 80(%rdx,%rcx,2),%rax`	R[rax] = M[80 + R[r_b]+R[r_i]*s]

- The hardware has to support the calculation of the effective address (i.e., 3 adds + 1 mul [by 2, 4, or 8])
- Meanwhile normal `add` and `mul` instructions can only do 1 operation at a time
  - **Idea:** Create an instruction that can use the address calculation hardware but for normal arithmetic ops
- **`lea`** = Load Effective Address
  - `leaq 80(%rdx,%rcx,2),%rax  // %rax = 80+%rdx+2*%rcx`
  - Computes the "address" and just puts it in the destination for later (**doesn't load anything from memory**)

lea Examples

- Initial Conditions
  - `leal (%rdx,%rcx),%eax`
  - `leaq -8(%rbx),%rax`
  - `leaq 12(%rdx,%rcx,2),%rax`



Caveats:
- `leal` zeroes out the upper 32-bits

About "offset(x, y, multiplier)"

4. **Addressing Modes and Data Movements** (12 points)

```
--------- REGISTERS --------
%rax = 0x1122 3344 4455 77FF
%rbx = 0xFFFF FFFF FFFF FFFF
%rcx = 0x0000 0000 0000 2016
%rdx = 0xFFFF FFAA 0044 22AA
```

```
----- MEMORY ------
DADA F00D @ M[2018]
B310 E818 @ M[2014]
D323 F415 @ M[2010]
1122 FFEE @ M[200C]
2A2A 1717 @ M[2008]
AA20 18BB @ M[2004]
0001 0000 @ M[2000]
```

Given the values above for registers/memory, what is the effect of the following instructions on %rdx? (Consider each instruction individually from the same initial state.)

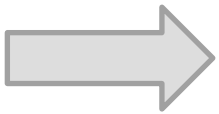
(d) [2 points] `movw 6(%rcx,%rbx,2),%dx` results in %rdx = FFFF FFAA 0044 DADA

rbx is all F's so it's -1… we need to **add -1*2+6 to rcx** in this example to obtain the EA (applies to mov, lea, add...)

Optimization with `lea`

```
// x is stored inside %edi
// return value in %eax
int f1(int x)
{
  return 9*x+1;
}
```

Original Code

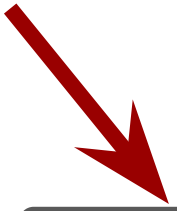


```
f1:
    movl  %edi,%eax    # tmp = x
    sall  $3,  %eax    # tmp *= 8
    addl  %edi,%eax    # tmp += x
    addl  $1,  %eax    # tmp += 1
    retq
```

Unoptimized

```
f1:
    leal  1(%rdi,%rdi,8),%eax
    retq
```

Optimized With lea Instruction

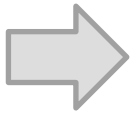


x86 Convention: %edi/%rdi used for **first argument**, %eax/%rax used for **return value**

Reusing addresses with `lea`

```
// ptr is stored inside %rdi
void incr(int *ptr) {
  *ptr = *ptr + 2;
}

// data is stored inside %rdi
// i    is stored inside %esi
void f(int *data, int i) {
  // uses address of data[i]
  data[i] = 1;
  // uses the same address
  incr(&data[i]);
}
```



```
incr:
  // load int from address rdi,
  // add 2 and store it back
  movl    (%rdi), %eax
  addl    $2, %eax
  movl    %eax, (%rdi)
  ret

f:
  // save addr of data[i] in rdi
  movslq  %esi, %rsi
  leaq    (%rdi,%rsi,4), %rdi
  // write one to that address
  movl    $1, (%rdi)
  // pass same address to incr
  call    incr
  ret
```

Compiler Example 1

```
// data = %rdi
// val  = %rsi
// i    = %edx
int f1(int data[], int *val, int i)
{
  int sum = *val;
  sum += data[i];
  return sum;
}
```

Original Code

```
f1:
    movslq %edx,%rdx
    movl (%rdi,%rdx,4),%eax
    addl (%rsi),%eax
    retq
```

Compiler Output

x86 Convention: Return value in %rax, inputs in %rdi, %rsi, %rdx

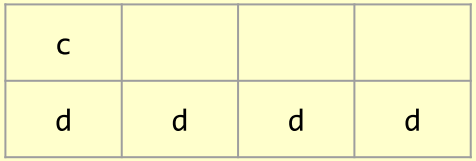
Compiler Output 2

```
struct Data {
  char c;  // 1 byte
  int d;   // 4 bytes
};

// ptr  = %rdi
// x    = %esi
int f1(struct Data *ptr, int x)
{
  ptr->c++;
  ptr->d -= x;
}
```

Original Code

```
f1:
    movzbl (%rdi),%eax
    addl $0x1,%eax
    movb %al,(%rdi)
    subl %esi,0x4(%rdi)
    retq
```



Compiler Output

x86 Convention: Return value in %rax, inputs in %rdi, %rsi, %rdx

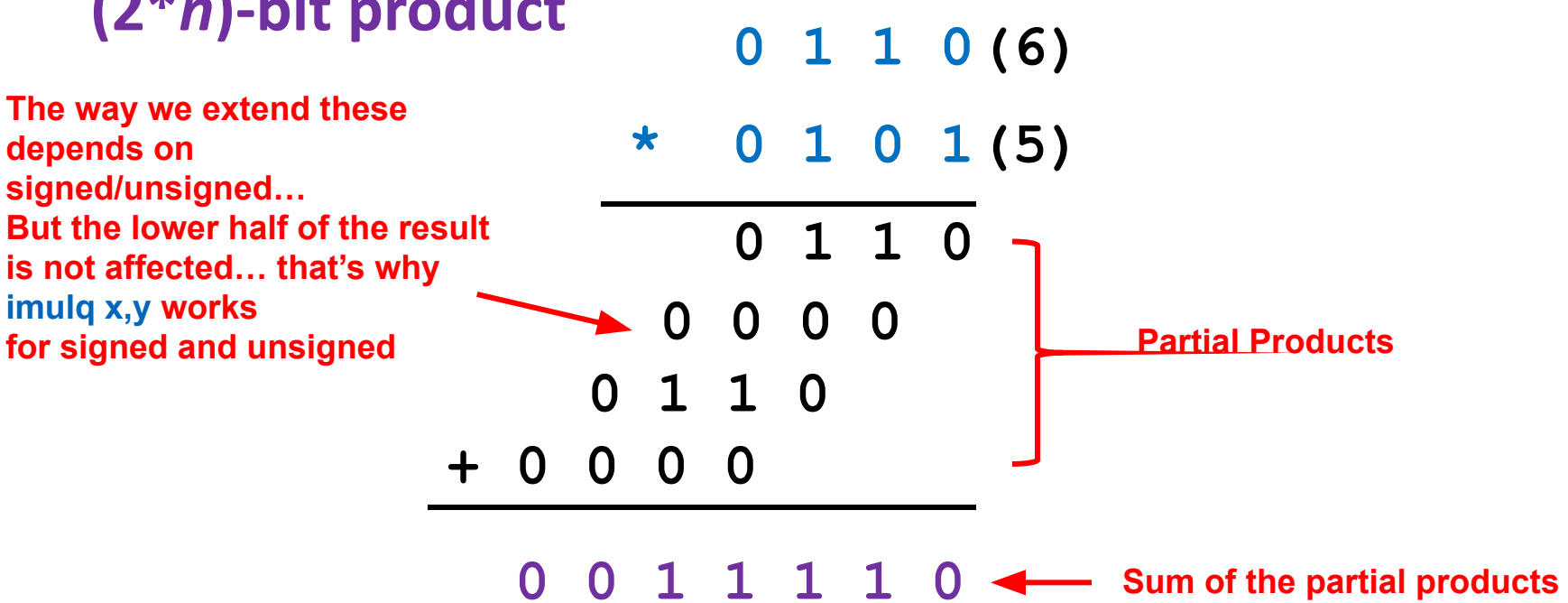
mov and add/sub examples

Instruction	M[0x7000]	M[0x7004]	%rax
	5A13 F87C	2933 ABC0	0000 0000 0000 0000
movl $0x26CE071B, 0x7000	26CE 071B	2933 ABC0	0000 0000 0000 0000
movsbw 0x7002,%ax	26CE 071B	2933 ABC0	0000 0000 0000 FFCE
movzwq 0x7004,%rax	26CE 071B	2933 ABC0	0000 0000 0000 ABC0
movw $0xFE44,0x7006	26CE 071B	FE44 ABC0	0000 0000 0000 ABC0
addl 0x7000,%eax	26CE 071B	FF4E ABC0	0000 0000 26CE B2DB
subb %al,0x7007	26CE 071B	244E ABC0	0000 0000 26CE B2DB

x86 MULTIPLICATION AND DIVISION

Binary Multiplication

- Like decimal multiplication, compute partial products, shift them, then tally up
  - 3-5x more expensive on modern CPUs
- **Multiplying two *n*-bit numbers yields at most a (2**n*)-bit product**

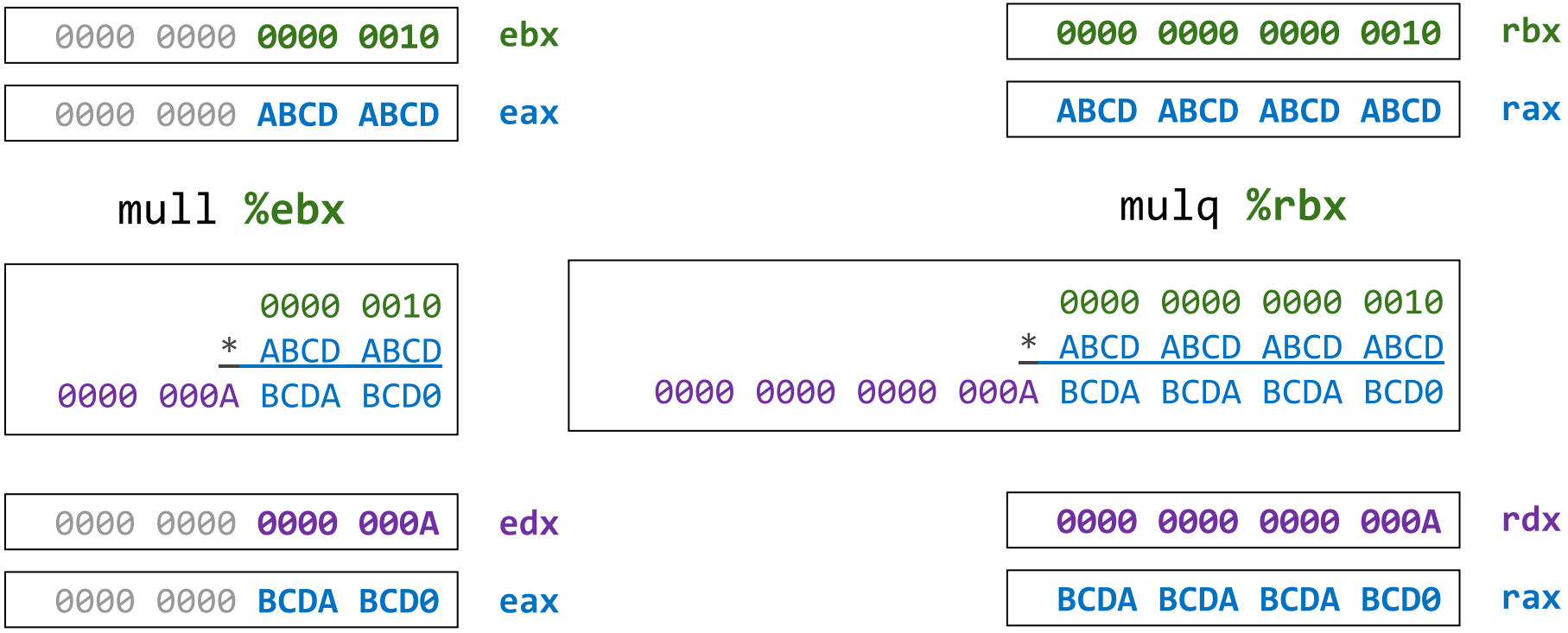


x86 multiply

- Since the product of a multiplication may result in twice as many bits as the input operands, the x86 architecture splits the output across two registers (%rdx and %rax)
- **Format**: `mul[l,q] src` (Unsigned multiply)
  `imul[l,q] src` (Signed multiply)
- **Operation**: Long `%edx:%eax = %eax * src (mull)`
  Quad `%rdx:%rax = %rax * src (mulq)`
  - Implicit 2nd operand is %eax or %rax
  - Results is split across %edx:%eax (or %rdx:%rax)
    - MSBs (Upper half) are saved to %edx (or %rdx)
    - LSBs (Lower half) are saved in %eax (or %rax)

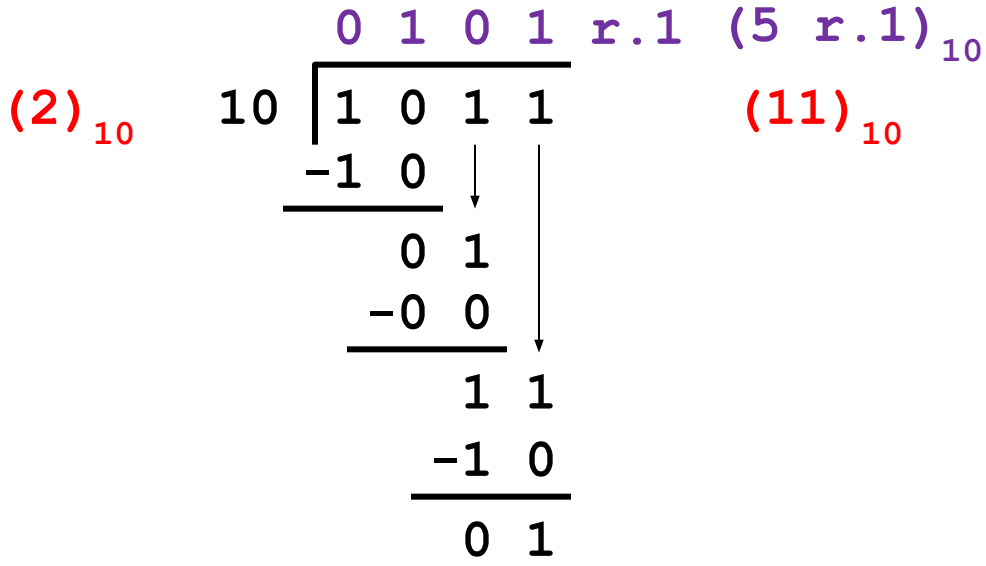
x86 multiply Examples

- Since the product of a multiplication may result in twice as many bits as the input operands, the x86 architecture splits the output across two registers (%rdx and %rax)



Binary Division

- **Dividing two *n*-bit numbers may yield an *n*-bit quotient and *n*-bit remainder**
- Division operations on a modern processor can take **17-41 times** longer than addition operations



x86 division

- Since an n-bit division produces **two *n*-bit results** (quotient and remainder), the x86 architecture splits the output across two registers (**%rdx** and **%rax**)
- **Format**: `div[l,q] src` (Unsigned divide)
  `idiv[l,q] src` (Signed divide)
- **Operation**:

```
%eax = %edx:%eax / src   (divl)
%edx = %edx:%eax % src
%rax = %rdx:%rax / src   (divq)
%rdx = %rdx:%rax % src
```

  - **Implicit dividend** is in %edx:%eax (or %rdx:%rax)
  - **Divisor** is specified as src 32-bit (or 64 bits)
  - **Quotient** goes in %eax (%rax), **remainder** in %edx (%rdx)

x86 division Examples

- Since the product of a multiplication is twice as many bits as the input operands, the x86 architecture splits the output across two registers (%rdx and %rax)

0000 0000 0000 000A	edx
0000 0000 BCDA BCD1	eax
0000 0000 0000 0010	ebx

`divl %ebx`

```
          A BCDA BCD1
/         0000 0010
Quotient = ABCD ABCD
Remainder =       1
```

0000 0000 0000 0001	edx
0000 0000 ABCD ABCD	eax

0000 0000 0000 000A	rdx
BCDA BCDA BCDA BCD1	rax
0000 0000 0000 0010	rbx

`divq %rbx`

```
                    A BCDA BCDA BCDA BCD1
/                   0000 0000 0000 0010
Quotient =          ABCD ABCD ABCD ABCD
Remainder =                          1
```

0000 0000 0000 0001	rdx
ABCD ABCD ABCD ABCD	rax

Compiler output

ASSEMBLY TRANSLATION EXAMPLE

Translation to Assembly

- We will now see some C code and its assembly translation
- A few things to remember:
  - Data variables live in memory (stack for local variables)
  - Data must be brought into registers before being processed
  - You often need an address/pointer in a register to load/store data to/from memory
- Generally, you will need 4 steps to translate C to assembly:
  - Setup a pointer in a register
  - Load data from memory to a register (mov)
  - Process data (add, sub, and, or, shift, etc.)
  - Store data back to memory (mov)

Translating HLL to Assembly

- Variables are simply locations in memory
  - A variable name really translates to an address in assembly

C operator	Assembly	Notes

`int x,y,z;`
`…`
`z = x + y;`
	`movl $0x10000004,%ecx`
	`movl (%rcx), %eax`
	`addl 4(%rcx), %eax`
	`movl %eax, 8(%rcx)`
		Assume:
		- x @ 0x10000004
		- y @ 0x10000008
		- z @ 0x1000000C
		• Purple = Pointer init
		• Blue = Read data from mem.
		• Red = ALU op
		• Green = Write data to mem.

`char a[100];`
`…`
`a[1]--;`
	`movl $0x1000000c,%ecx`
	`decb 1(%rcx)`
		Assume array ‘a’ starts at 0x1000000C

Translating HLL to Assembly

C operator	Assembly	Notes

`int dat[4],x;`
`…`
`x = dat[0];`
`x += dat[1];`
	`movl $0x10000010,%ecx`
	`movl (%rcx), %eax`
	`movl %eax, 16(%rcx)`
	`movl 16(%rcx), %eax`
	`addl 4(%rcx), %eax`
	`movl %eax, 16(%rcx)`
		Assume
		- dat @ 0x10000010
		- x @ 0x10000020
		• Purple = Pointer init
		• Blue = Read data from mem.
		• Red = ALU op
		• Green = Write data to mem.

`unsigned int y;`
`short z;`
`y = y / 4;`
`z = z << 3;`
	`movl $0x10000010,%ecx`
	`movl (%rcx), %eax`
	`shrl 2, %eax`
	`movl %eax, (%rcx)`
	`movw 4(%rcx), %ax`
	`salw 3, %ax`
	`movw %ax, 4(%rcx)`
		Assume
		- y @ 0x10000010
		- z @ 0x10000014

How instruction sets differ

INSTRUCTION SET ARCHITECTURE

Instruction Set Architecture (ISA)

- Defines the software interface of the processor and memory system
- Instruction set is the vocabulary the HW can understand and the SW is composed with
- 2 approaches
  - **CISC** = Complex instruction set computer
    - Large, rich vocabulary
    - More work per instruction but slower HW
  - **RISC** = Reduced instruction set computer
    - Small, basic, but sufficient vocabulary
    - Less work per instruction but faster HW

Components of an ISA

- Data and Address Size
  - 8-, 16-, 32-, 64-bit
- Which instructions does the processor support
  - SUBtract instruc. vs. NEGate + ADD instrucs.
- Registers accessible to the instructions
  - How many and expected usage
- Addressing Modes
  - How instructions can specify location of data operands
- Length and format of instructions
  - How is the operation and operands represented with 1's and 0's

General Instruction Format Issues

- Different instruction sets specify these differently
  - 3 operand instruction set (ARM, PPC)
    - Similar to example on previous page
    - Format: ADD DST, SRC1, SRC2 (DST = SRC1 + SRC2)
  - 2 operand instructions (Intel)
    - Second operand doubles as source and destination
    - Format: ADD SRC1, S2/D (S2/D += SRC1)
  - 1 operand instructions (Old Intel FP, Low-End Embedded)
    - Implicit operand to every instruction usually known as the Accumulator (or ACC) register
    - Format: ADD SRC1 (ACC += SRC1)

General Instruction Format Issues

- Consider the pros and cons of each format when performing the set of operations
  - F = X + Y – Z
  - G = A + B
- Simple embedded computers often use single operand format
  - Smaller data size (8-bit or 16-bit machines) means limited instruc. size
- Modern, high performance processors use 2- and 3-operand formats

Single-Operand	Two-Operand	Three-Operand

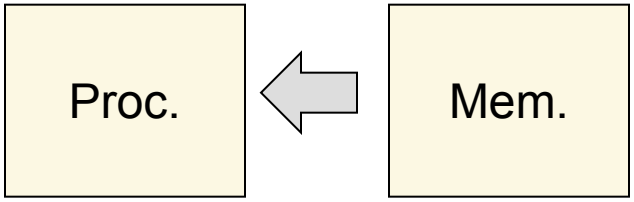
LOAD X
ADD Y
SUB Z
STORE F
LOAD A
ADD B
STORE G
	MOVE F,X
	ADD F,Y
	SUB F,Z
	MOVE G,A
	ADD G,B
		ADD F,X,Y
		SUB F,F,Z
		ADD G,A,B

(+) Smaller size to encode each instruction (-) Higher instruction count to load and store ACC value	Compromise of two extremes	(+) More natural program style (+) Smaller instruction count (-) Larger size to encode each instruction

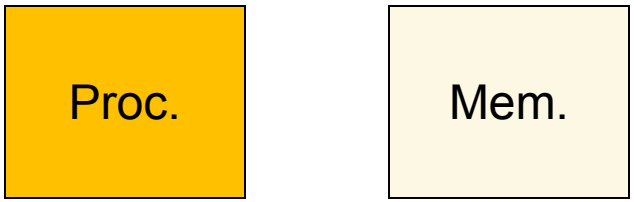
Instruction Format

- Load/Store architecture
  - Load (read) data values from memory into a register
  - Perform operations on registers
  - Store (write) data values back to memory
  - Different load/store instructions for different operand sizes (i.e. byte, half, word)

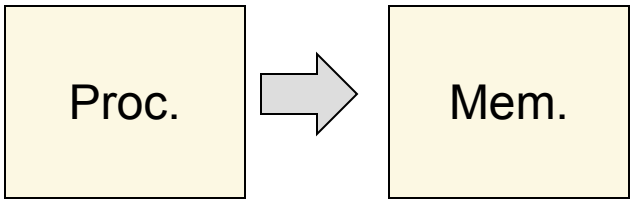
Load/Store Architecture



1.) Load operands to proc. registers



2.) Proc. Performs operation using register values



3.) Store results back to memory

Recorded Lecture

Basic Computer Organization

Processor, instructions, registers

BASIC COMPUTER ORGANIZATION

Where Does It Live

- Match (1-Processor / 2-Memory / 3-Disk Drive) where each item resides:
  - Source Code (.c / .java)
  - Compiled Executable (Before It Executes)
  - Running Program Code
  - Global Variables
  - Local Variables
  - Current Instruction Being Executed

(1) Processor

(2) Memory

(3) Disk Drive

Where Does It Live

- Match (1-Processor / 2-Memory / 3-Disk Drive) where each item resides:
  - Source Code (.c / .java) = 3
  - Compiled Executable (Before It Executes) = 3
  - Running Program Code = 2
  - Global Variables = 2
  - Local Variables = 2
  - Current Instruction Being Executed = 1

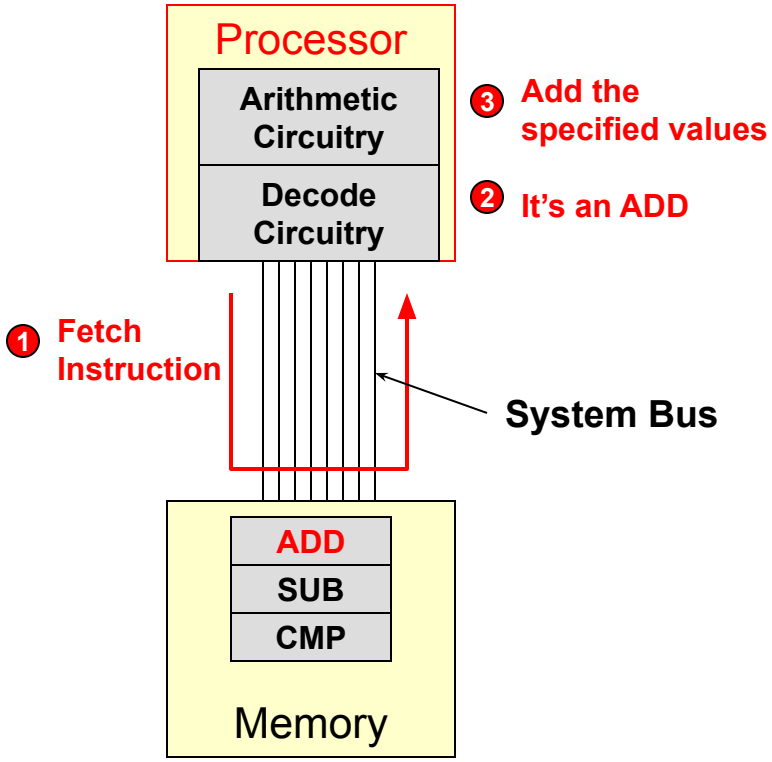
(1) Processor

(2) Memory

(3) Disk Drive

Processor

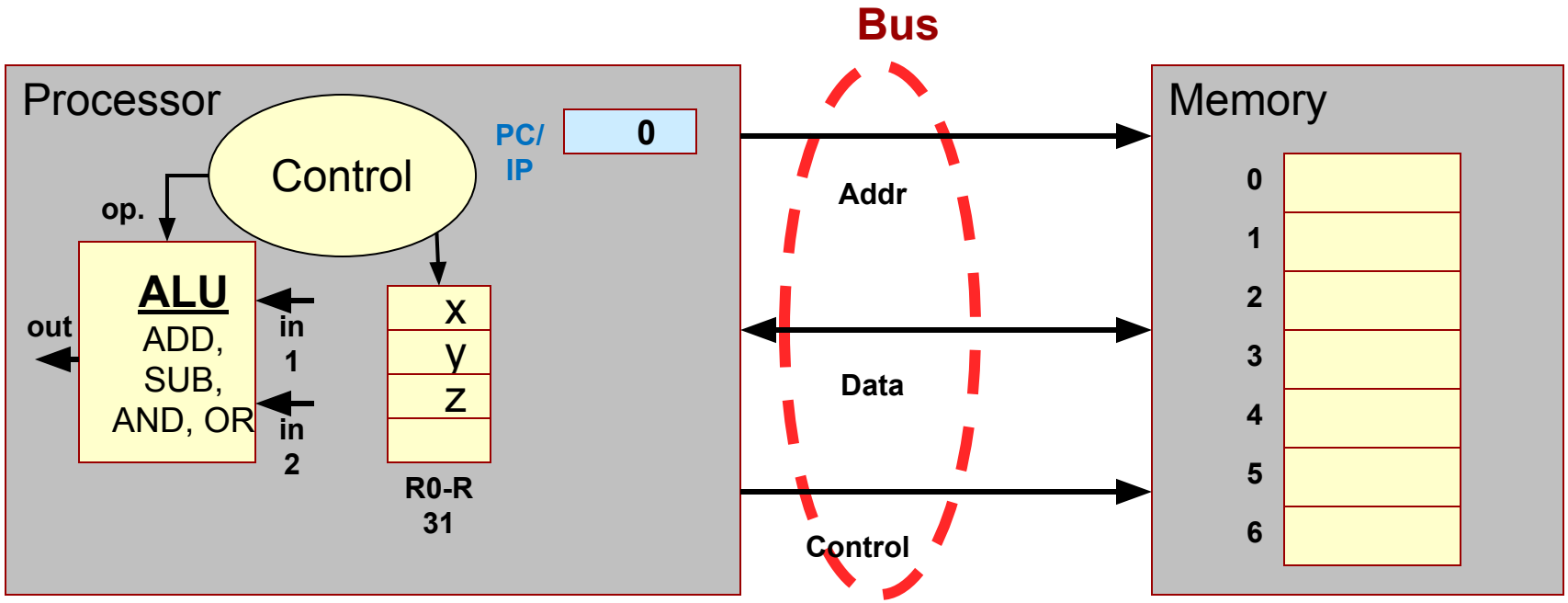
- Performs the same 3-step process over and over again
  - Fetch an instruction from memory
  - Decode the instruction
    - Is it an ADD, SUB, etc.?
  - Execute the instruction
    - Perform the specified operation
- This process is known as the **Instruction Cycle**



Processor

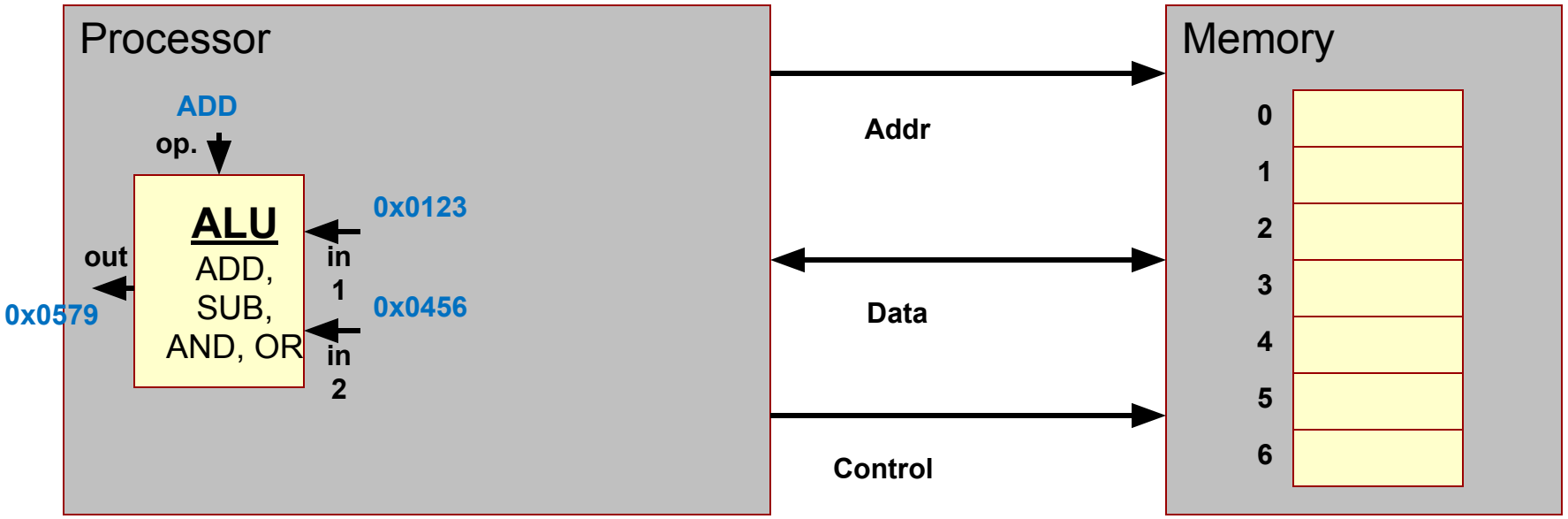
CS:APP 1.4

- 3 Primary Components inside a processor
  - **ALU**
  - **Registers**
  - **Control Circuitry**
- Connected to memory and I/O via **address**, **data**, and **control** buses (**bus** = group of wires)



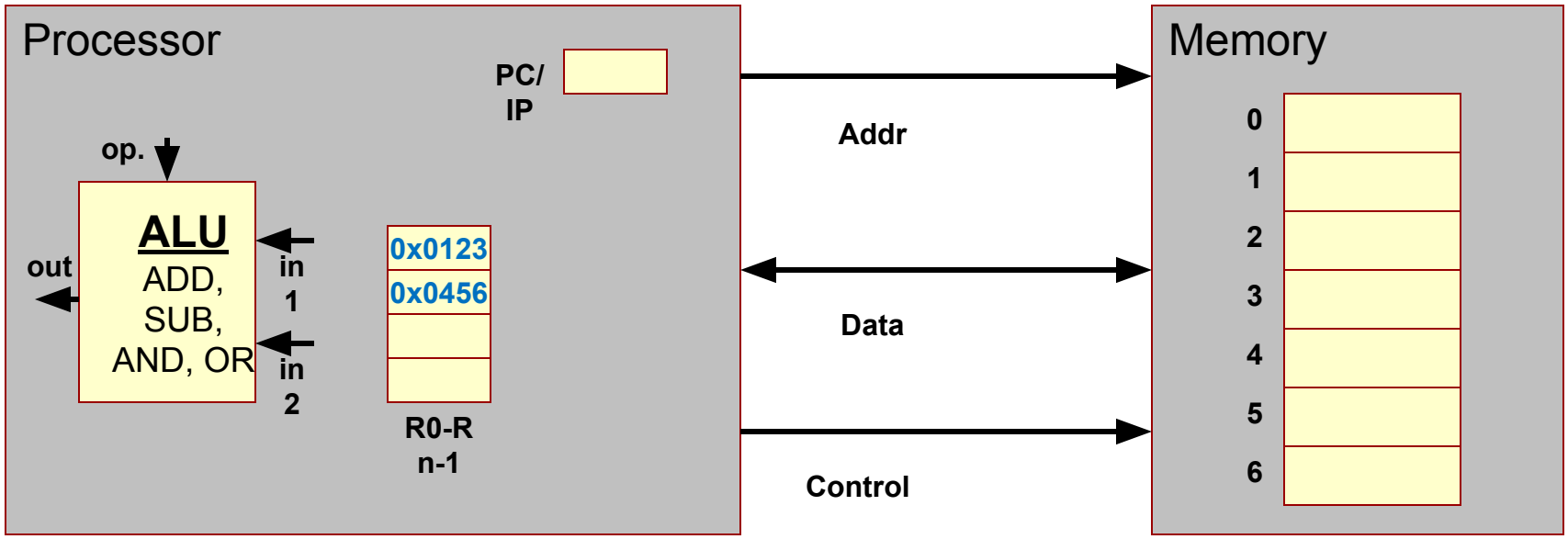
Arithmetic and Logic Unit (ALU)

- Digital circuit that performs arithmetic / logic operations (ADD, SUB, AND, OR, etc.)



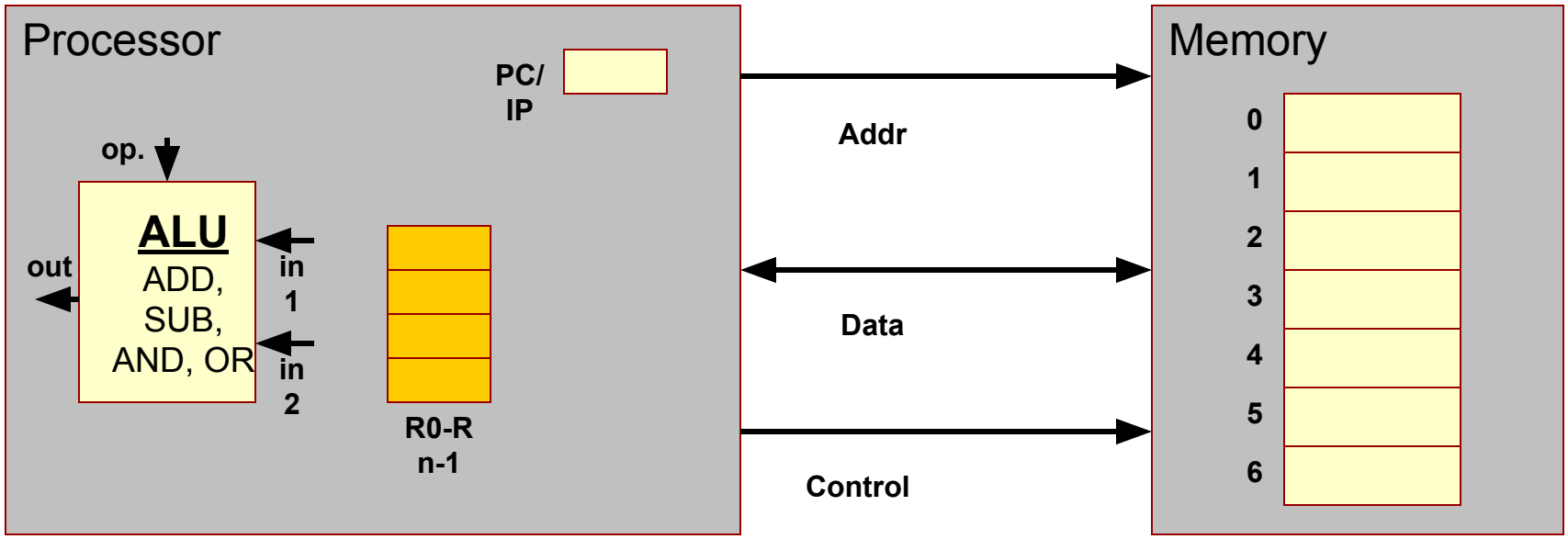
Registers

- Recall memory is SLOW compared to a processor
- Registers provide fast, temporary storage locations within the processor



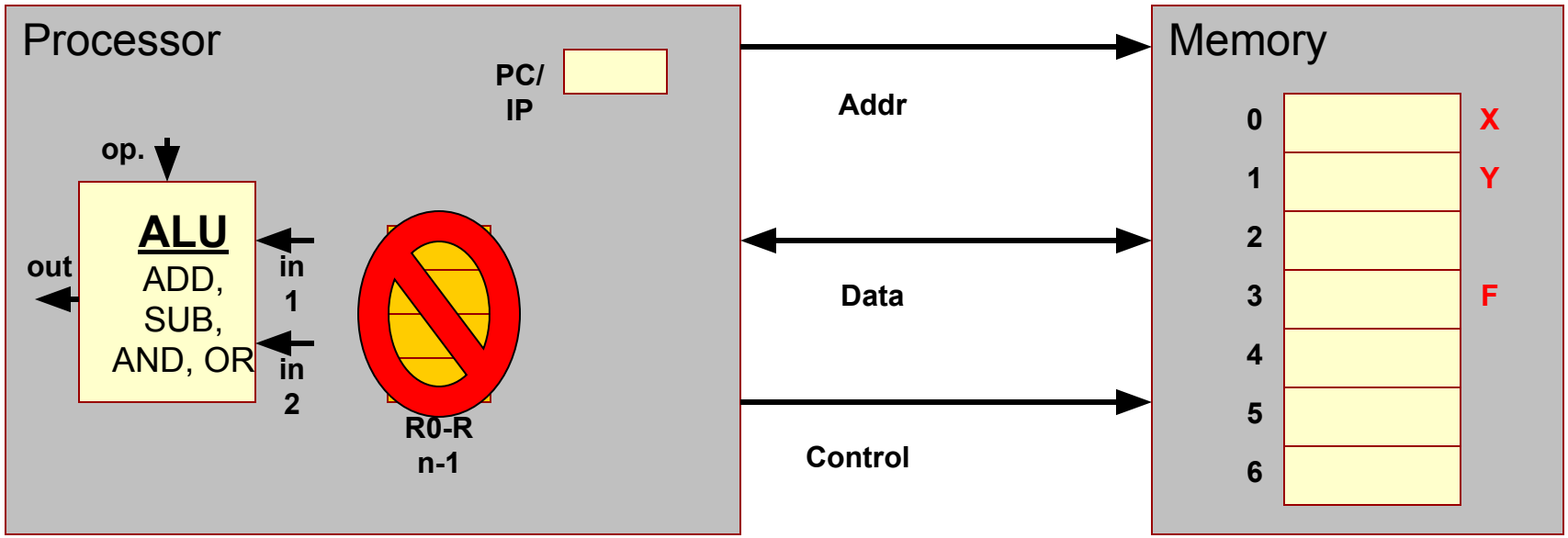
General Purpose Registers

- Registers available to software instructions for use by the programmer/compiler
- Programmer/**compiler** is in charge of using these registers as inputs (source locations) and outputs (destination locations)



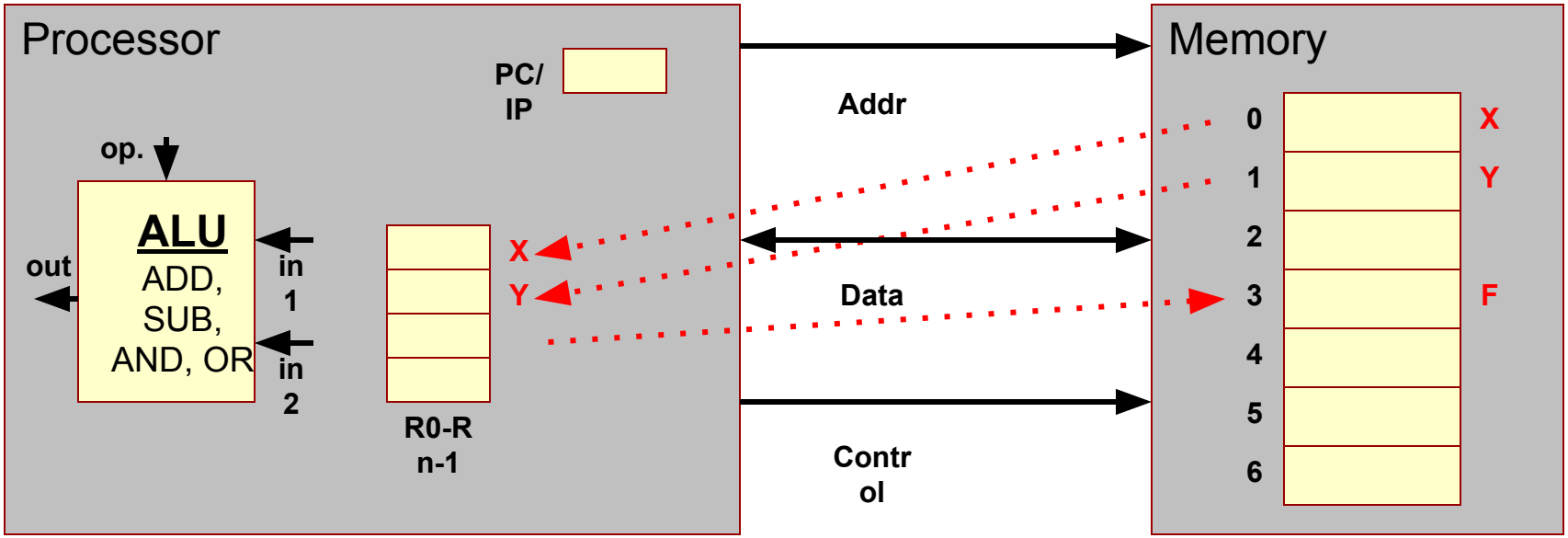
What if we didn't have registers?

- Example w/o registers: F = (X+Y) – (X*Y)
  - Requires an ADD instruction, MULtiply instruction, and SUBtract Instruction
  - w/o registers
    - ADD: Load X and Y from memory, store result to memory
    - MUL: Load X and Y again from mem., store result to memory
    - SUB: Load results from ADD and MUL and store result to memory
    - 9 memory accesses



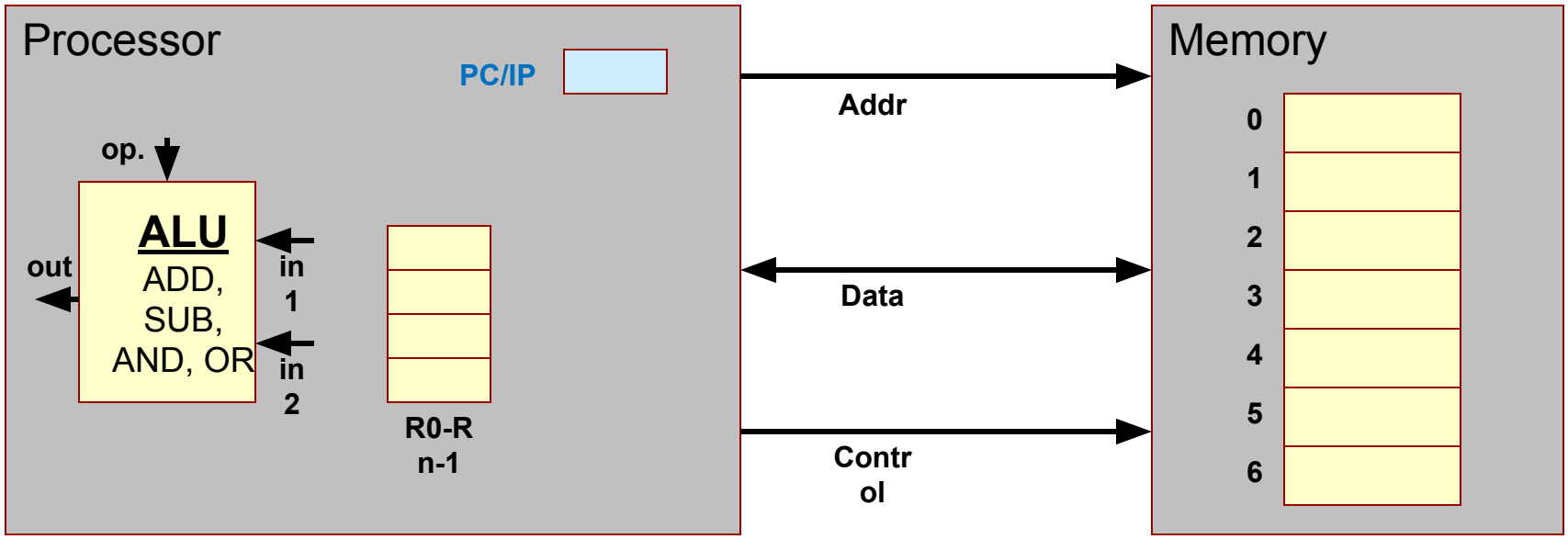
How to use registers?

- Example w/ registers: F = (X+Y) – (X*Y)
  - Load X and Y into registers R0 and R1
  - ADD: R0 + R1 and store result in R2
  - MUL: R0 * R1 and store result in R3
  - SUB: R2 – R3 and store result in R4
  - Store R4 back to memory
  - 3 total memory access



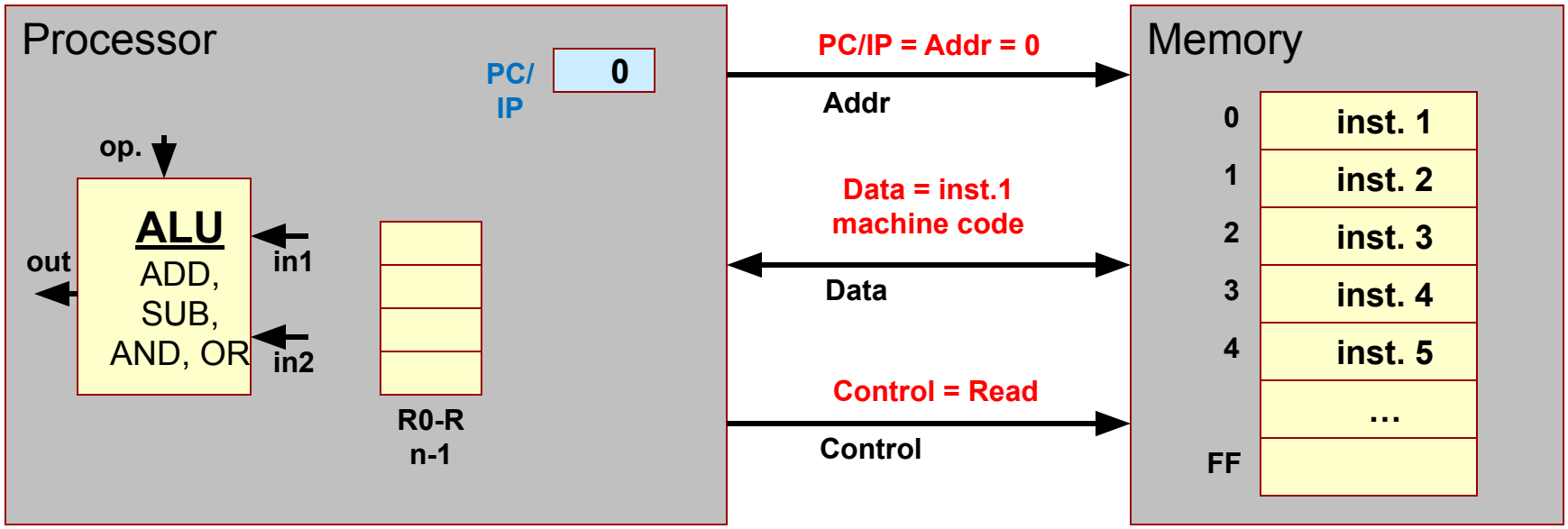
Other Registers

- Some bookkeeping information is needed to make the processor operate correctly
- Example: Program Counter/Instruction Pointer (PC/IP) Reg.
  - Recall that the processor must fetch instructions from memory before decoding and executing them
  - PC/IP register holds the address of the next instruction to fetch



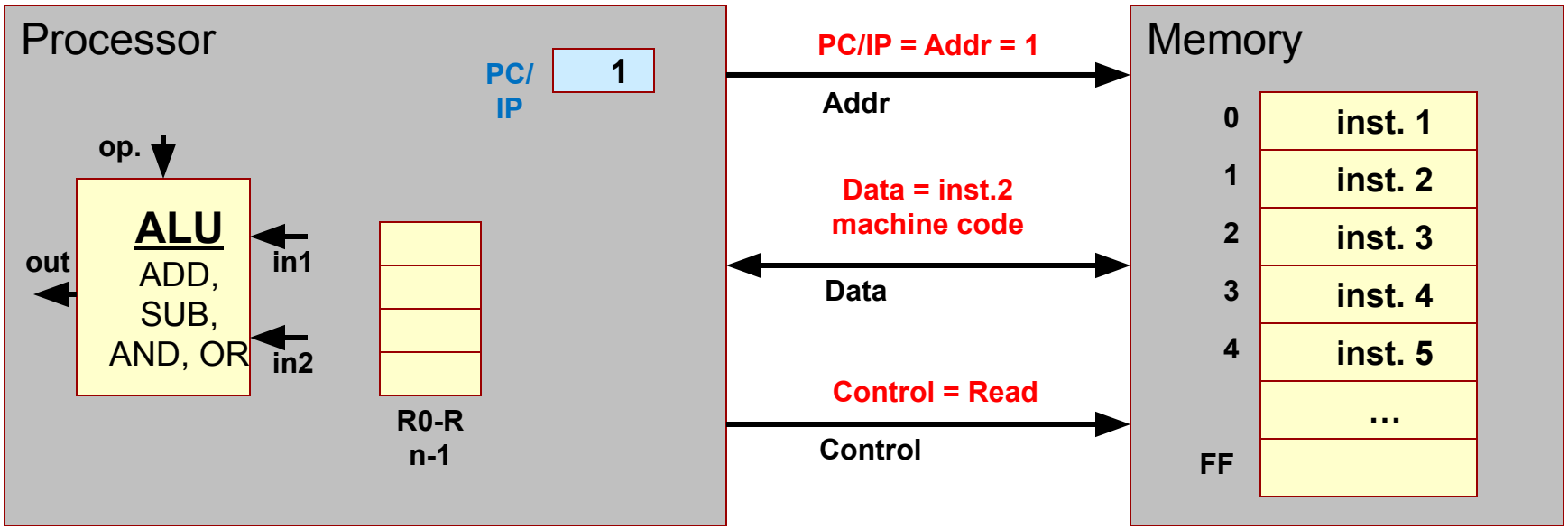
Fetching an Instruction

- To fetch an instruction
  - PC/IP contains the address of the instruction
  - The value in the PC/IP is placed on the address bus and the memory is told to read
  - The PC/IP is incremented, and the process is repeated for the next instruction



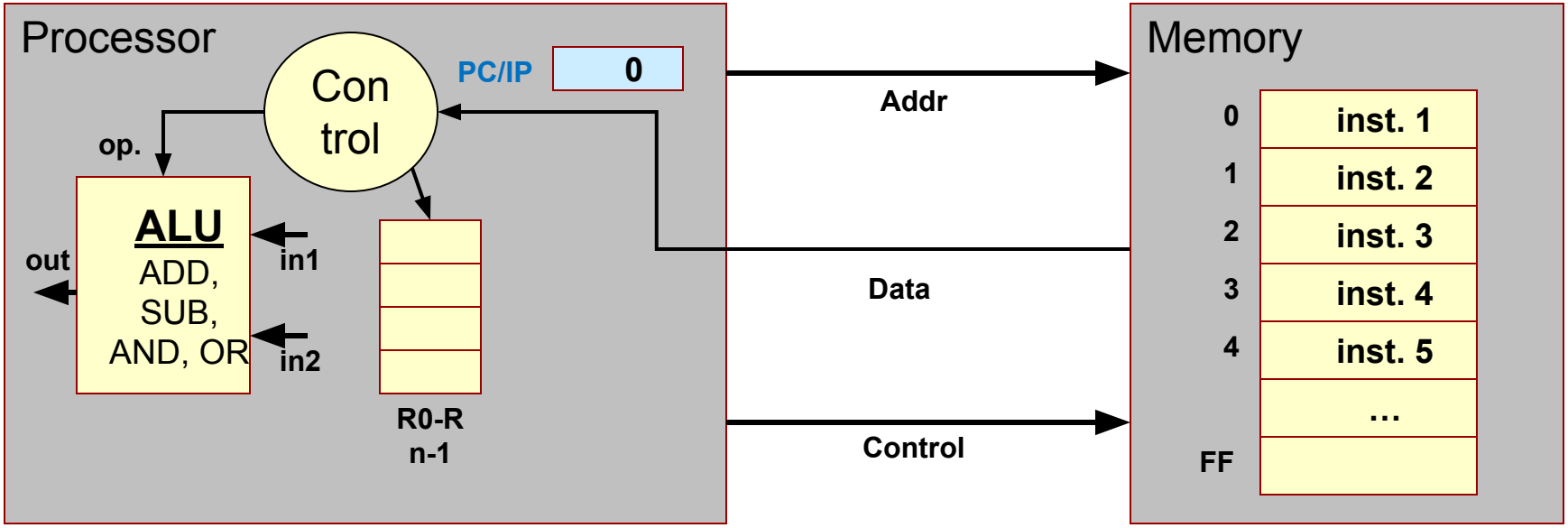
Fetching an Instruction

- To fetch an instruction
  - PC/IP contains the address of the instruction
  - The value in the PC/IP is placed on the address bus and the memory is told to read
  - The PC/IP is incremented, and the process is repeated for the next instruction



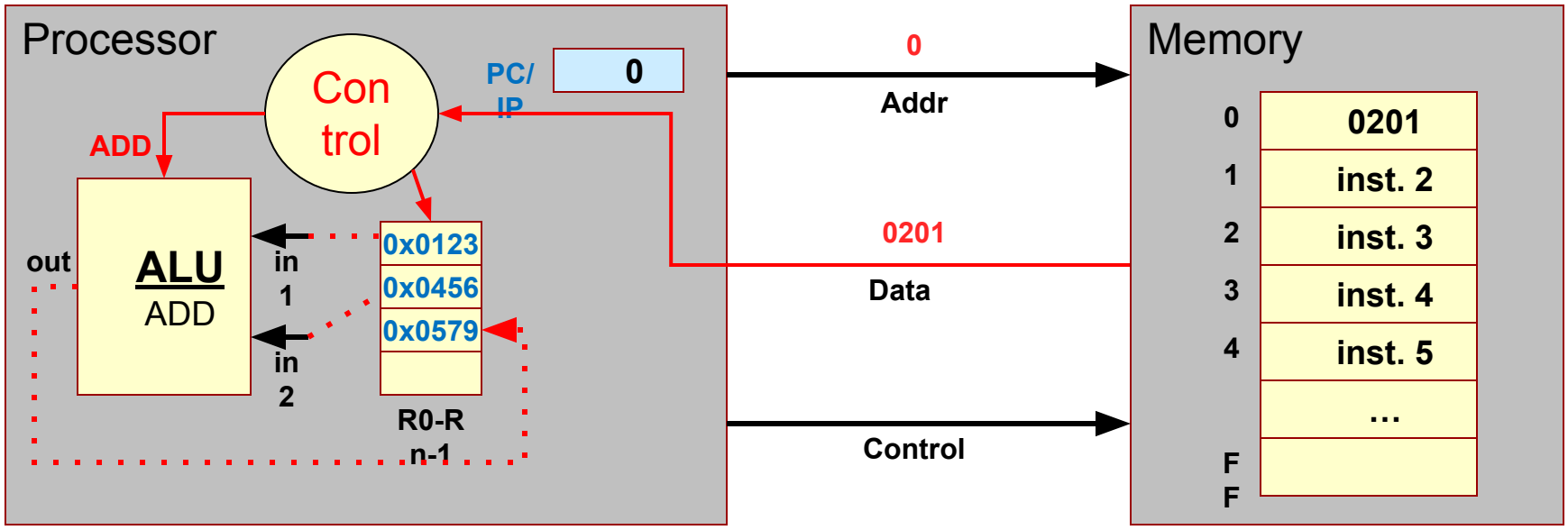
Control Circuitry

- Control circuitry is used to decode the instruction and then generate the necessary signals to complete its execution
- Controls the ALU
- Selects registers to be used as source / destination locations



Control Circuitry

- Assume 0x0201 is machine code for an ADD instruction of R2 = R0 + R1
- Control Logic will…
  - select the registers (R0 and R1)
  - tell the ALU to add
  - select the destination register (R2)



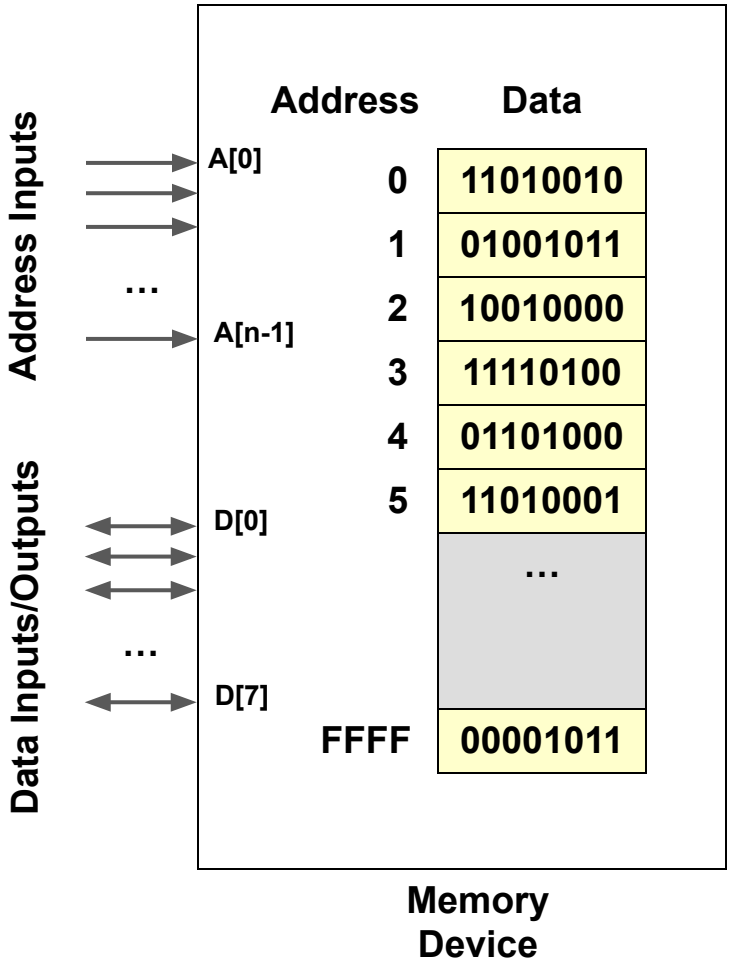
Summary

- **Registers** are used for fast, temporary storage in CPU
  - Data (usually) must be moved into registers
- The **PC register** stores the address of **next instruction** to be executed
  - Maintains the current execution location in the program

UNDERSTANDING MEMORY

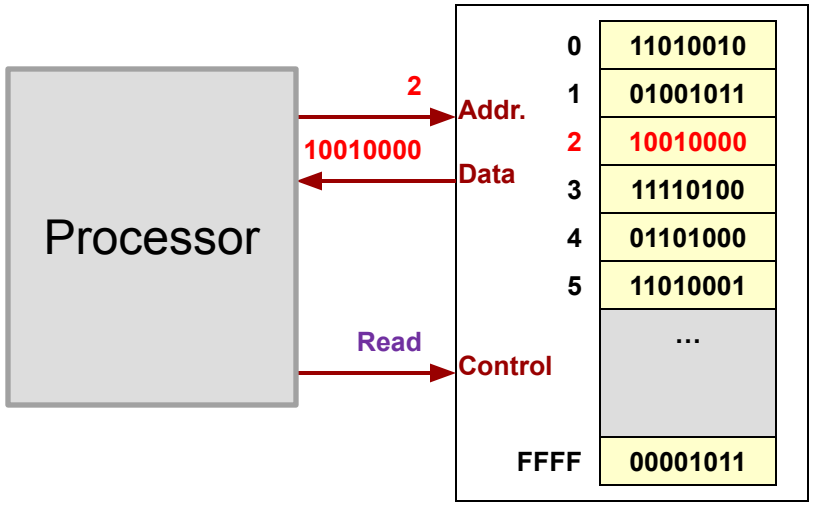
Memory and Addresses

- Set of cells that each store a group of bits
  - Usually, **1 byte (8 bits) per cell**
- Unique address (number) assigned to each cell
  - Used to reference the value in that location
- Data and instructions are both stored in memory and are always represented as a string of 1's and 0's

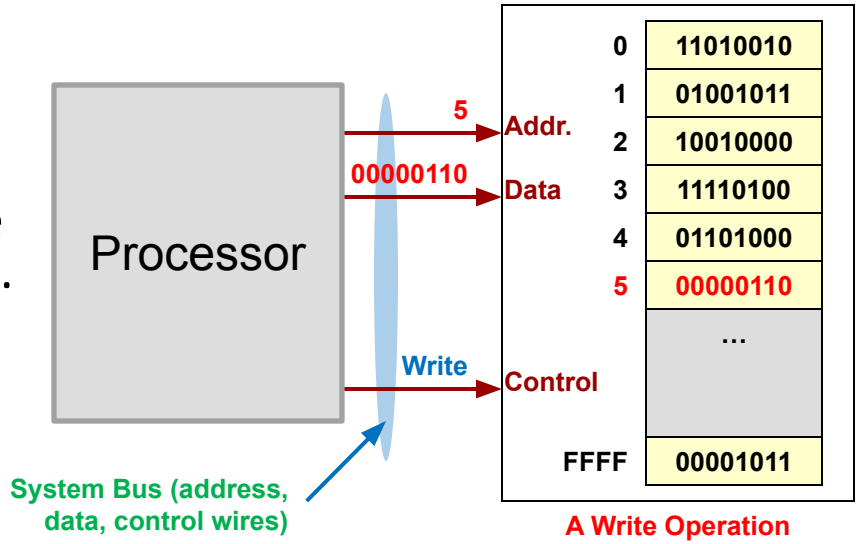


Reads & Writes

- Memories perform 2 operations
  - **Read**: retrieves data value in a particular location (specified using the address)
  - **Write**: changes data in a location to a new value
- To perform these operations a set of address, data, and control wires are used to talk to the memory
  - Note: A group of wires/signals is referred to as a '**bus**'
  - Thus, we say that memories have an address, data, and control bus.



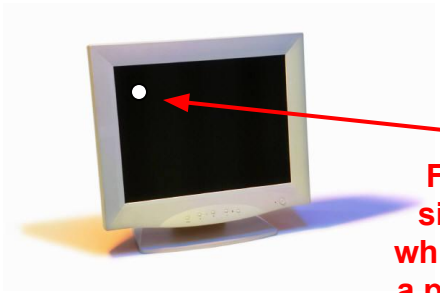
A Read Operation



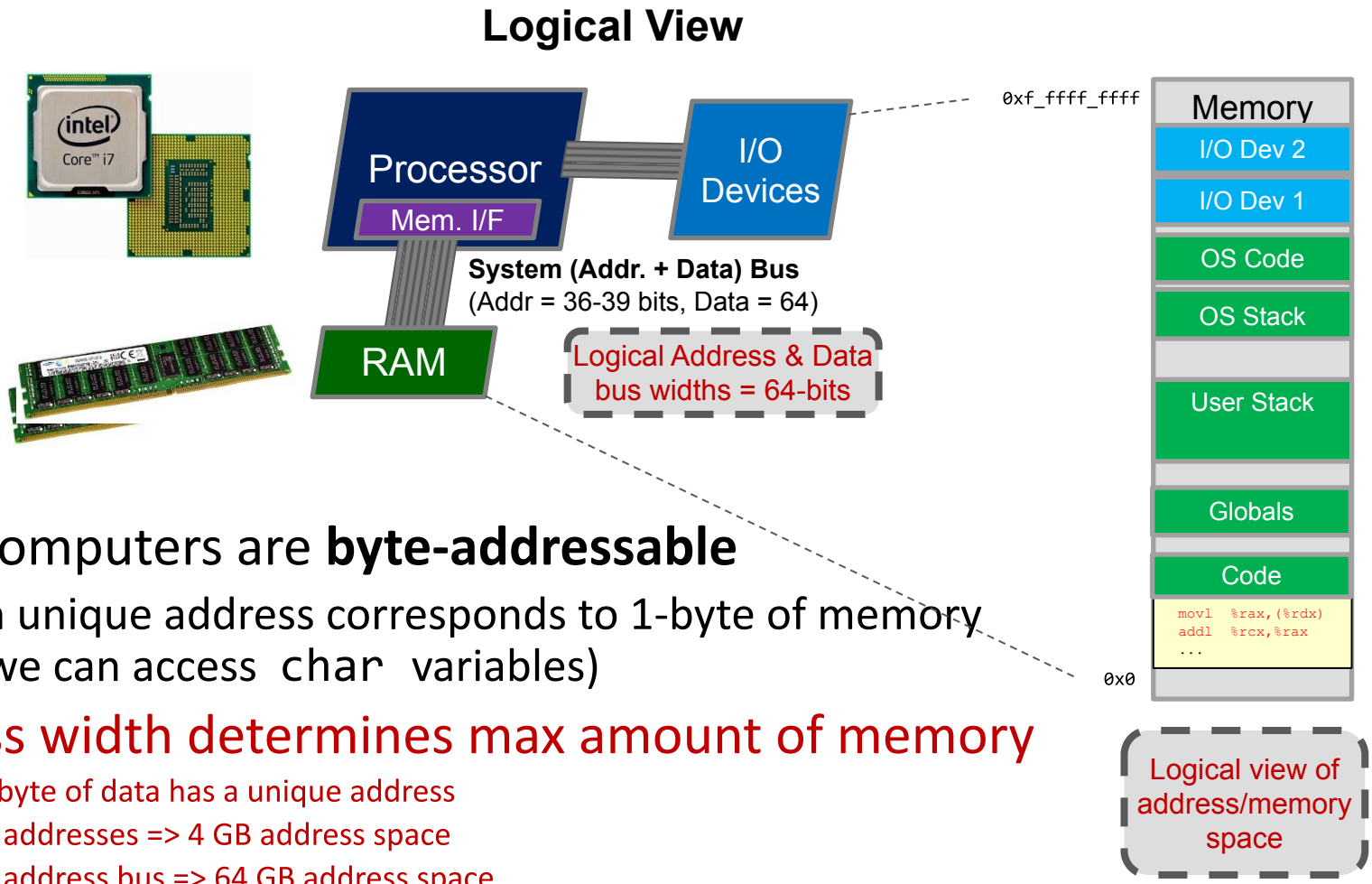
A Write Operation

Memory vs. I/O Access

- Processor performs reads and writes to communicate with memory and I/O devices
  - I/O devices have memory locations that contain data that the processor can access
  - All memory locations (be it RAM or I/O) have **unique addresses** used to identify them
  - **The assignment of memory addresses is known as the physical memory map**



Address Space (Size and View)

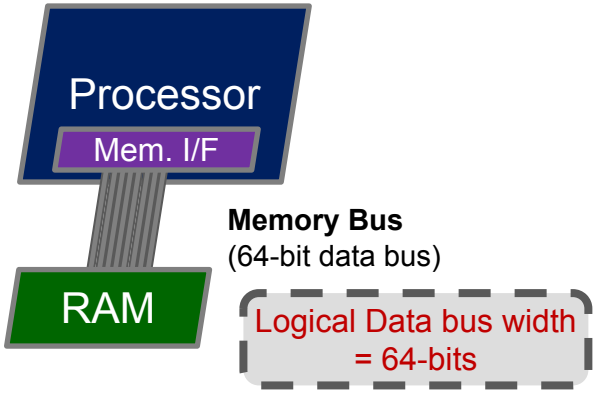


- Most computers are **byte-addressable**
  - Each unique address corresponds to 1-byte of memory (so we can access `char` variables)
- Address width determines max amount of memory
  - Every byte of data has a unique address
  - 32-bit addresses => 4 GB address space
  - 36-bit address bus => 64 GB address space

Data Bus (and Data Size)

- Moore's Law meant we could build systems with more transistors
- More transistors meant greater bit-widths
  - Just like more physical space allows for wider roads/freeways, more transistors allowed us to move to 16-, 32- and 64-bit circuitry inside the processor
- To support smaller variable sizes (`char` = 1-byte) we still need to access only 1-byte of memory per access, but to support `int` and `long ints` we want to access 4- or 8-byte chunks of memory per access
- Thus the data bus (highway connecting the processor and memory) has been getting wider
  - The processor can use 8-, 16-, 32- or all 64-bits of the bus (lanes of the highway) in a single access based on the size of data that is needed

Processor	Data Bus Width
Intel 8088	8-bit
Intel 8086	16-bit
Intel 80386	32-bit
Intel Pentium	64-bit



Intel Architectures

Processor	Year	Address Size	Data Size
8086	1978	20	16
80286	1982	24	16
80386/486	’85/’89	32	32
Pentium	1993	32	32
Pentium 4	2000	32	32
Core 2 Duo	2006	36	64
Core i7 (Haswell)	2013	39	64

x86-64 Data Sizes

CS:APP 3.3

Integer

- 4 sizes
  - **Byte** (B)
    - 8-bits = 1 byte
  - **Word** (W)
    - 16-bits = 2 bytes
  - **Double Word** (L)
    - 32-bits = 4 bytes
  - **Quad Word** (Q)
    - 64-bits = 8 bytes

Floating Point

- 2 sizes
  - **Single** (S)
    - 32-bits = 4 bytes
  - **Double** (D)
    - 64-bits = 8 bytes
    - (For a 32-bit data bus, a double would be accessed from memory in 2 reads)

In x86-64, instructions generally specify what size data to access from memory and then operate upon.

x86-64 Memory Organization

- Because each byte of memory has its own address we can picture memory as one column of bytes (Fig. 2)
- With 64-bit logical data bus we can access up to 8-bytes of data at a time
- We will usually show memory arranged in rows of 4 bytes (Fig. 3) or 8 bytes
  - Still with **separate addresses for each byte**

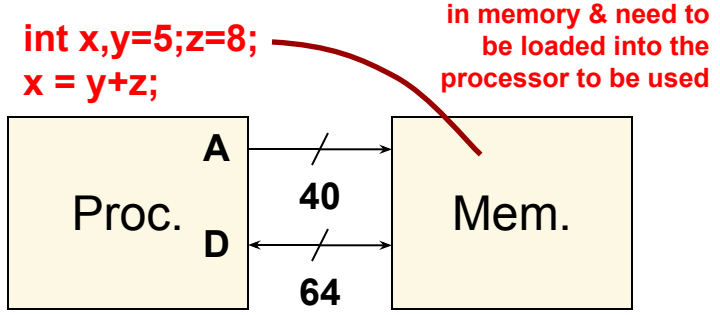
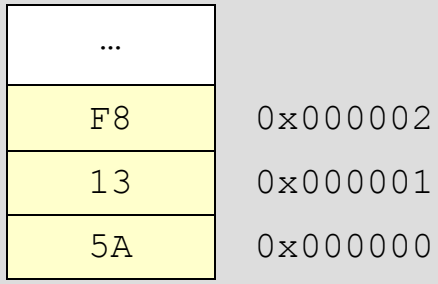
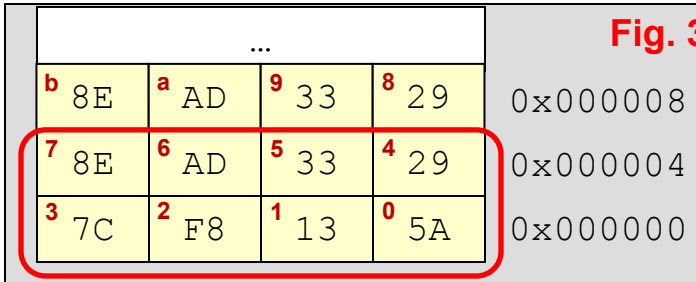


Fig. 2



Logical Byte-Oriented View of Mem.

Fig. 3

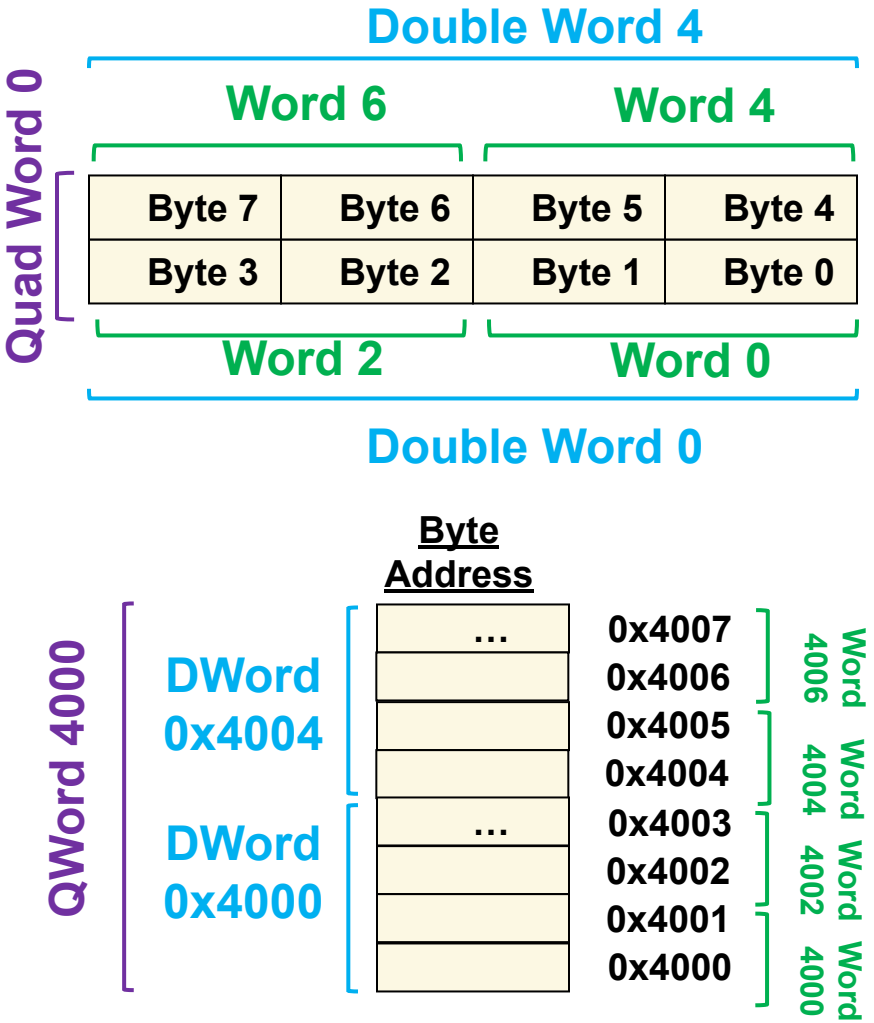


Logical DWord-Oriented View

Memory & Word Size

CS:APP 3.9.3

- To refer to a chunk of memory we must provide:
  - The starting address
  - The size: **B**, **W**, **D**, **L**
- There are rules for valid starting addresses
  - **A valid starting address should be a multiple of the data size**
  - Words (2-byte chunks) must start on an even (divisible by 2) address
  - Double words (4-byte chunks) must start on an address that is a multiple of (divisible by) 4
  - Quad words (8-byte chunks) must start on an address that is a multiple of (divisible by) 8



Endian-ness

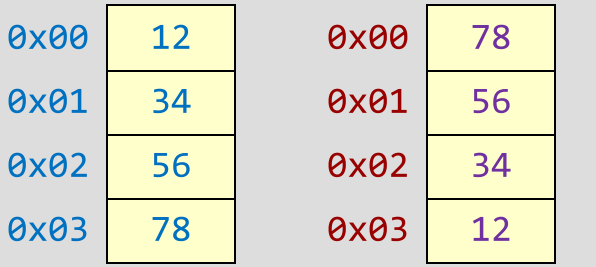
CS:APP 2.1.3

- **Endian-ness** refers to the two alternate methods of ordering the **bytes** in a larger unit (2, 4, 8 bytes)
  - **Big-Endian**
    - PPC, Sparc, TCP/IP
    - **MS byte is put at the starting address**
  - **Little-Endian**
    - used by Intel processors / original PCI bus
    - ***LS byte is put at the starting address***
- Some processors (like ARM) and busses can be configured for either big- or little-endian

The DWORD value:

0 x 12 34 56 78

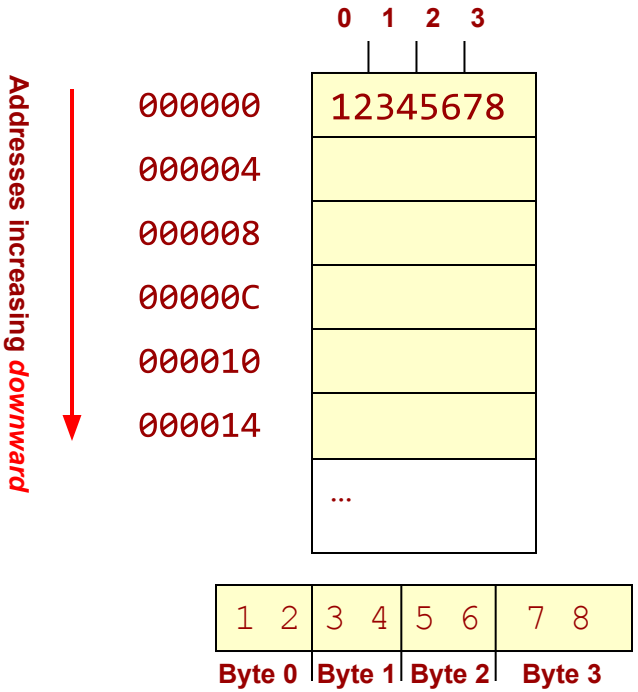
can be stored differently

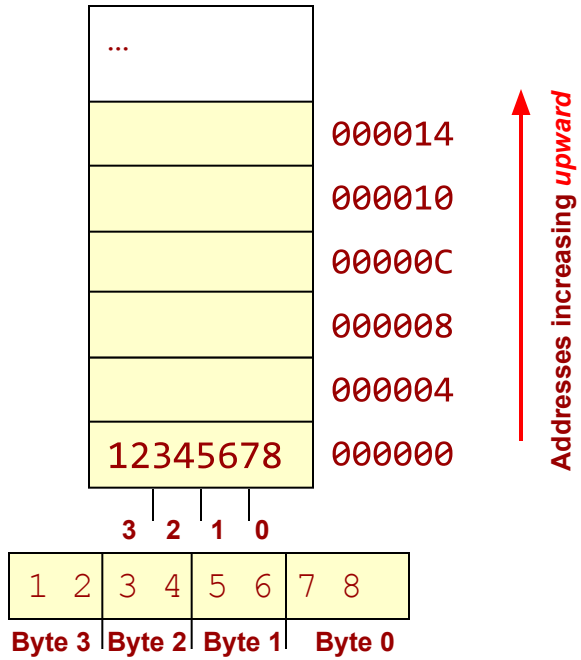


Big-Endian **Little-Endian**

Big-endian vs. Little-endian

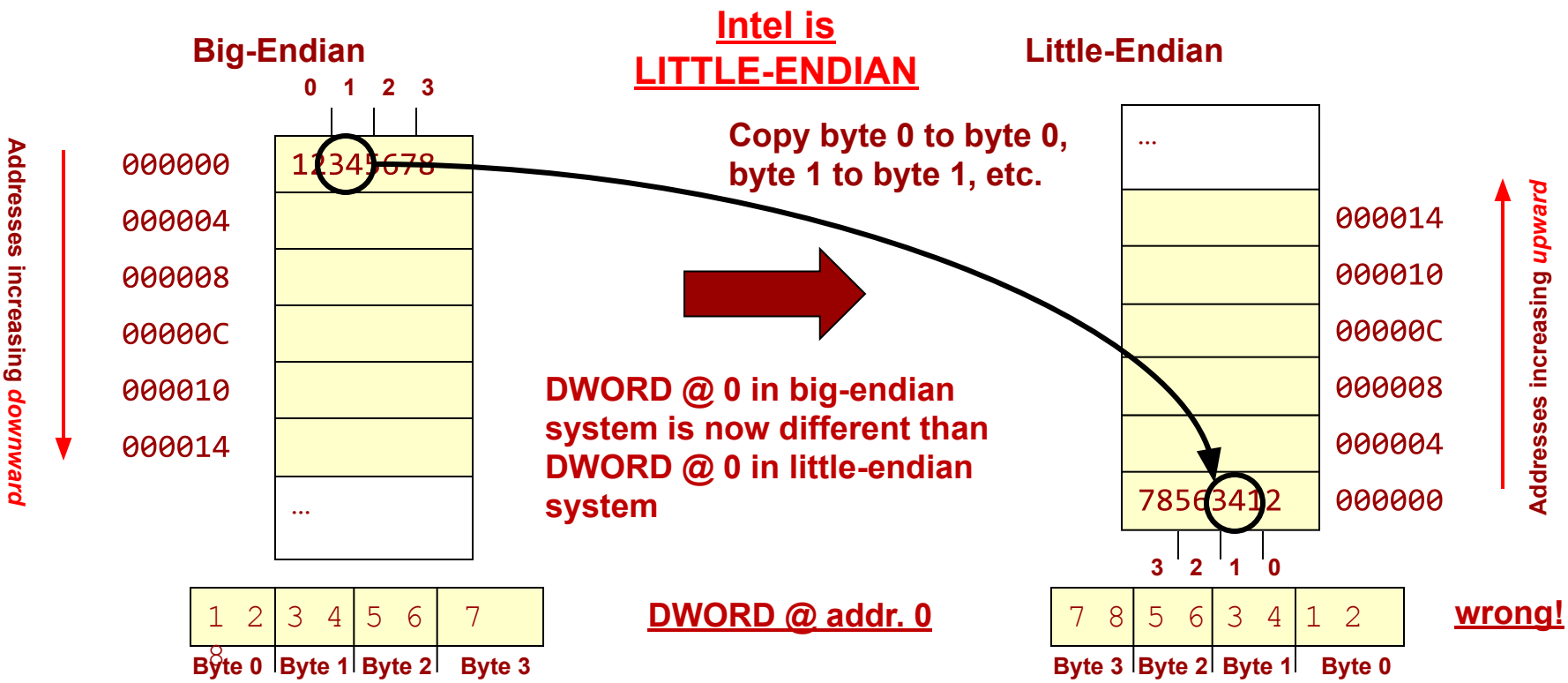
- Big-endian
  - makes sense if you view your memory as starting at the top-left and addresses increasing as you go down
- Little-endian
  - makes sense if you view your memory as starting at the bottom-right and addresses increasing as you go up





Big-endian vs. Little-endian Issues

- Issues arise when transferring data between different systems
  - Byte-wise copy of data from big-endian system to little-endian system
  - Major issue in networks (little-endian computer => big-endian computer) and even within a single computer (system memory => I/O device)



Summary

- The processor communicates with all other components in the processor via reads/writes using unique addresses for each component
- Memory can be accessed in different size chunks (byte, word, dword, quad word)
- Alignment rules: data of size *n* should start on an address that is a multiple of size *n*
  - dword should start on multiples of 4
  - size 8 should start on an address that is a multiple of 8
- x86 uses little-endian
  - The start address of a word (or dword or qword) refers to the LS-byte