CS356 Unit 4

Intro to
x86 Instruction Set

Why Learn Assembly

- To understand something of the limitation of the HW we are running on
- Helpful to understand performance
- To utilize certain HW options that high-level languages don't allow (e.g. operating systems, utilizing special HW features, etc.)
- To understand possible security vulnerabilities or exploits
- Can help debugging

Compilation Process

CS:APP 3.2.2

- Demo of assembler
  - $ g++ -Og -c -S file1.cpp
- Demo of hexdump
  - $ g++ -Og -c file1.cpp
  - $ hexdump -C file1.o | more
- Demo of objdump/disassembler
  - $ g++ -Og -c file1.cpp
  - $ objdump -d file1.o

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

Original Code

```
Disassembly of section .text:

0000000000000000 <_Z3absiPi>:
   0:    85 ff     test    %edi,%edi
   2:    79 05     jns     9 <_Z3absiPi+0x9>
   4:    f7 df     neg     %edi
   6:    89 3e     mov     %edi,(%rsi)
   8:    c3        retq
   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)

Where Does It Live

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

(1) Processor

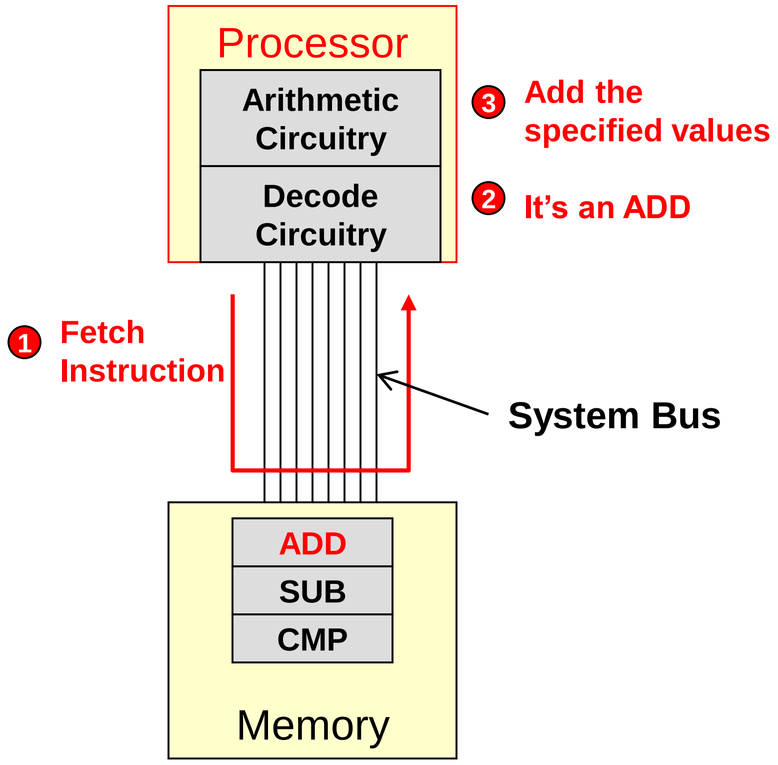
(2) Memory

(3) Disk Drive

BASIC COMPUTER ORGANIZATION

Processor

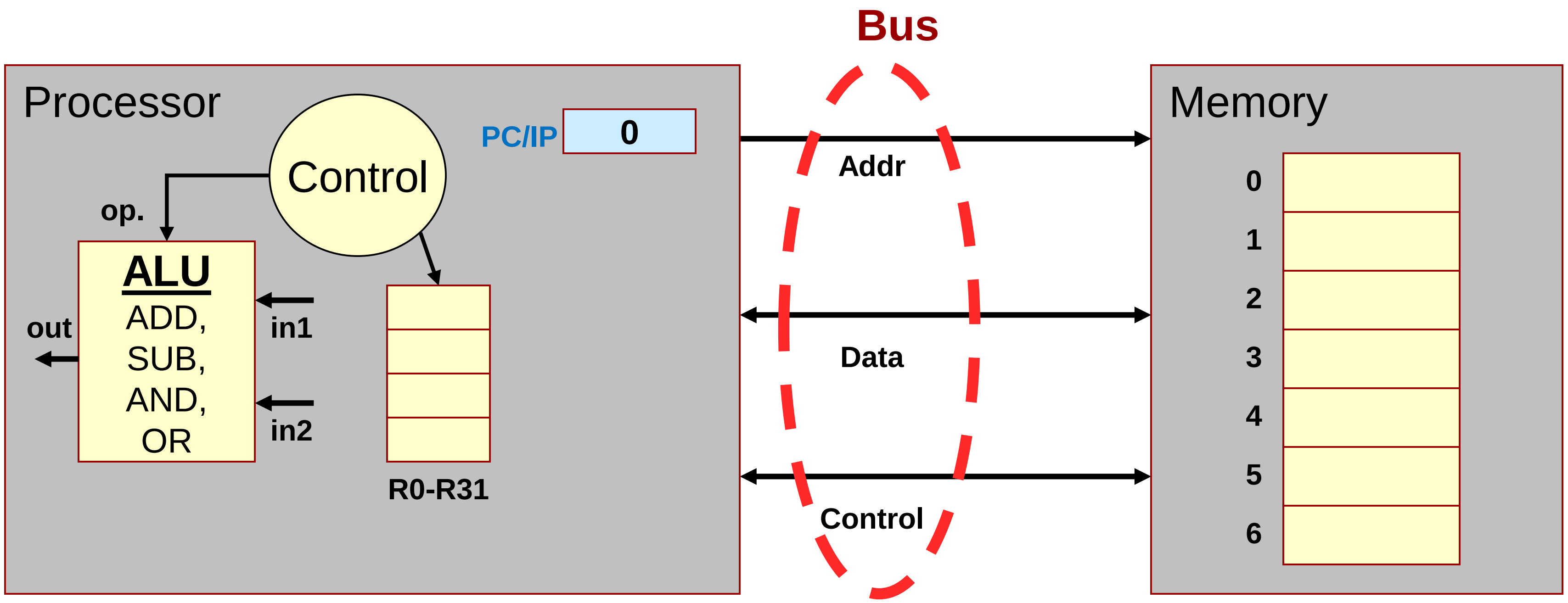
- Performs the same 3-step process over and over again
  - Fetch an instruction from memory
  - Decode the instruction
    - Is it an ADD, SUB, etc.?
  - Execute the instruction
    - Perform the specified operation
- This process is known as the **Instruction Cycle**



Processor

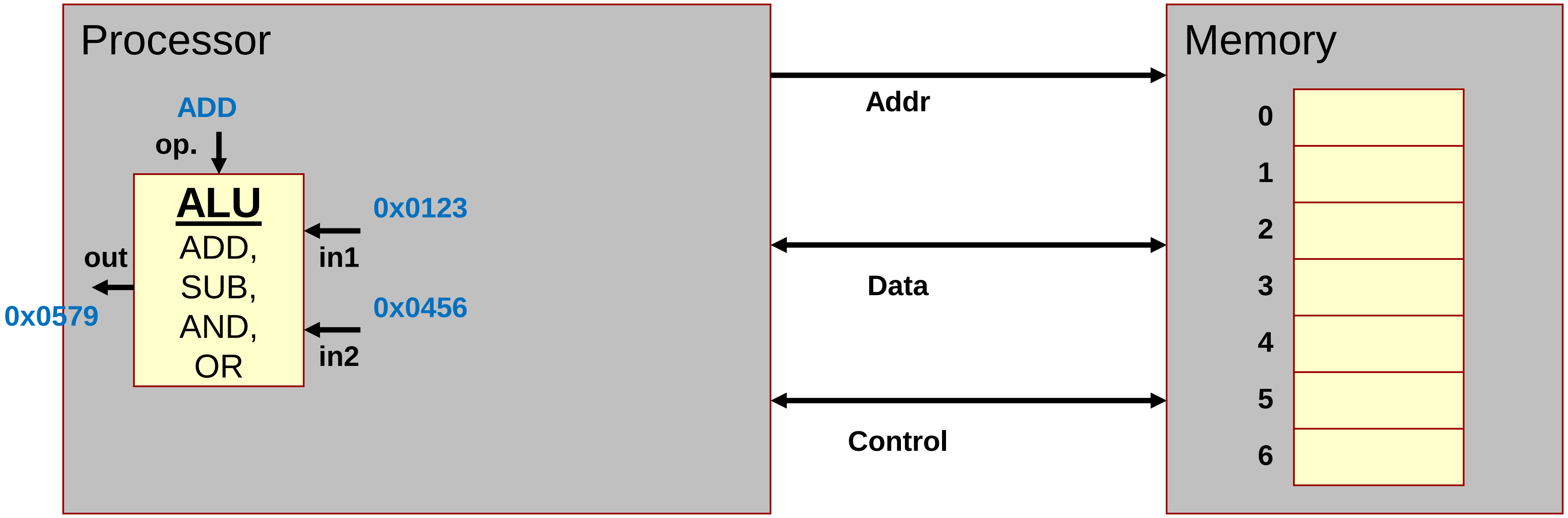
CS:APP 1.4

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



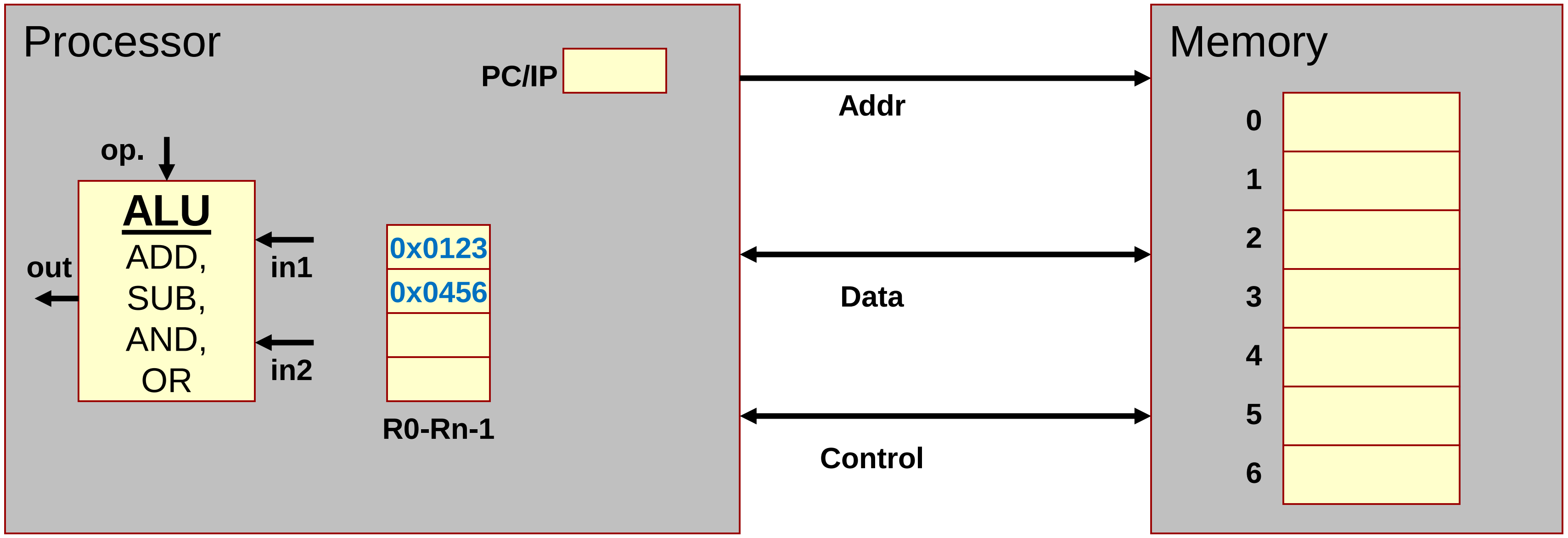
Arithmetic and Logic Unit (ALU)

- Digital circuit that performs arithmetic operations like addition and subtraction along with logical operations (AND, OR, etc.)



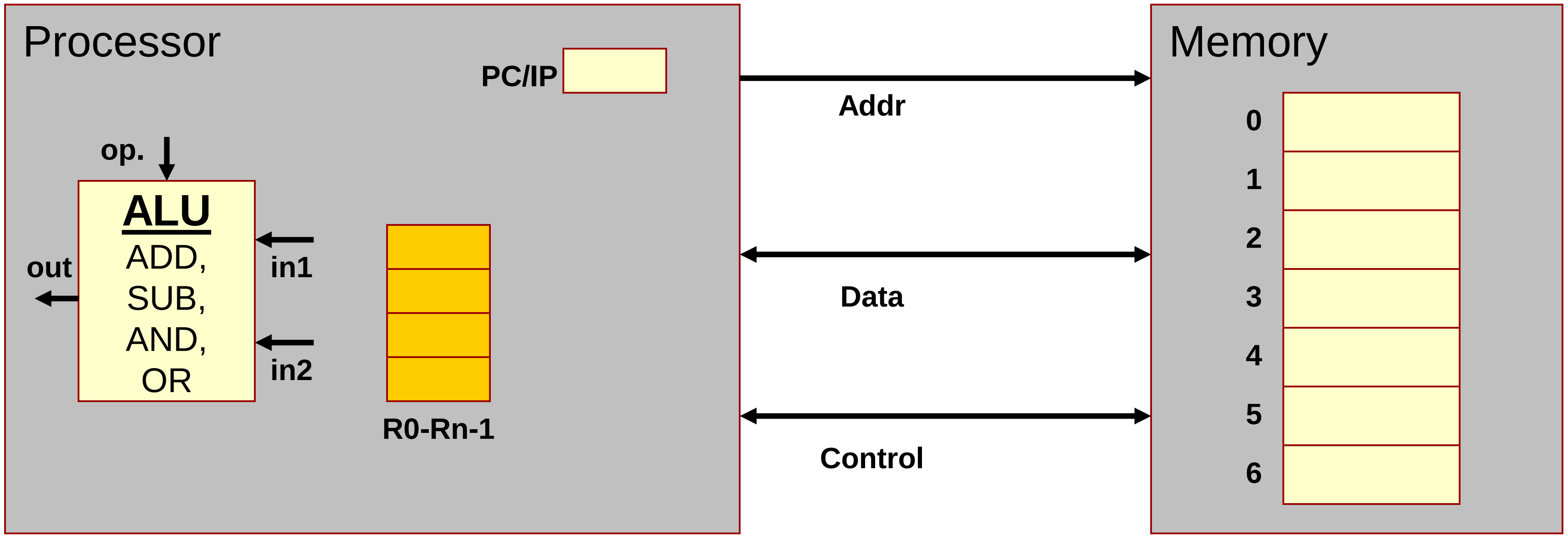
Registers

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



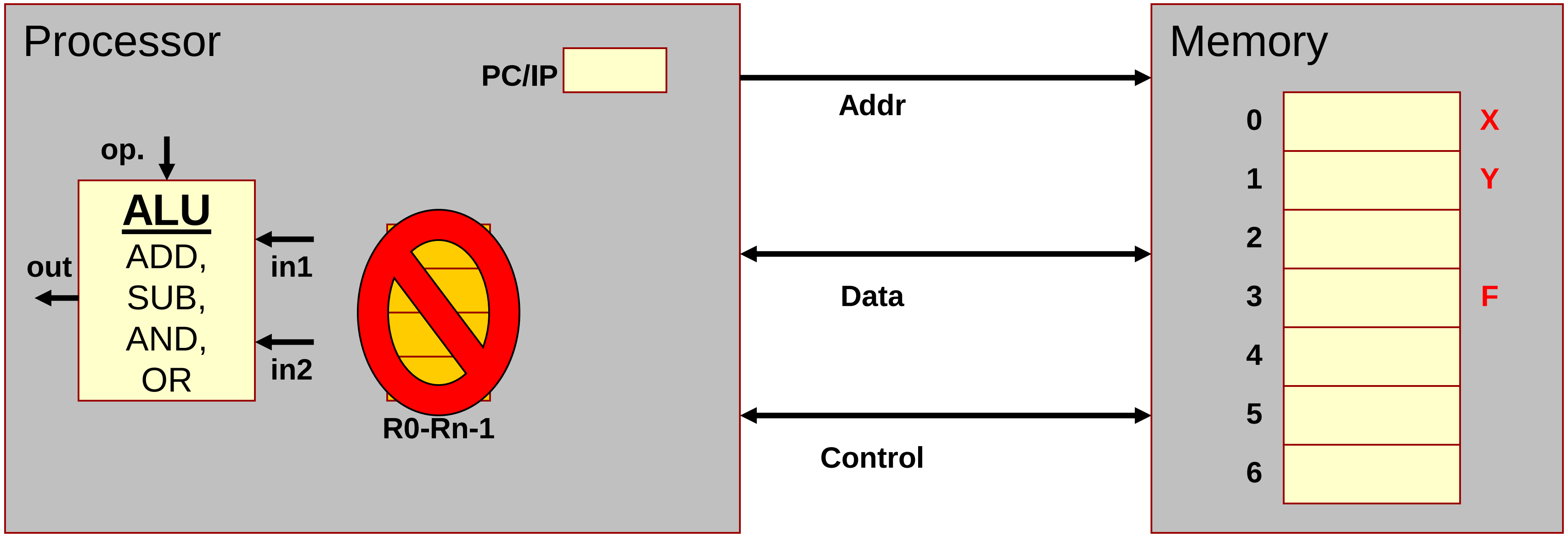
General Purpose Registers

- 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)



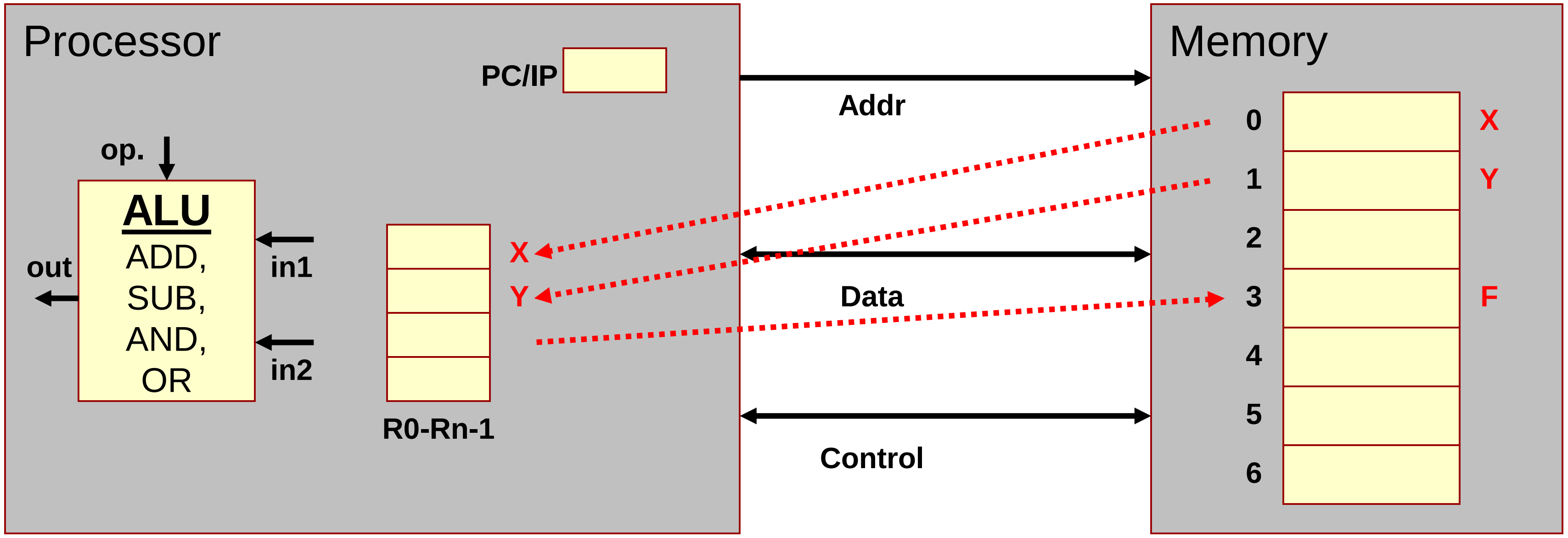
What if we didn’t have registers?

- 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



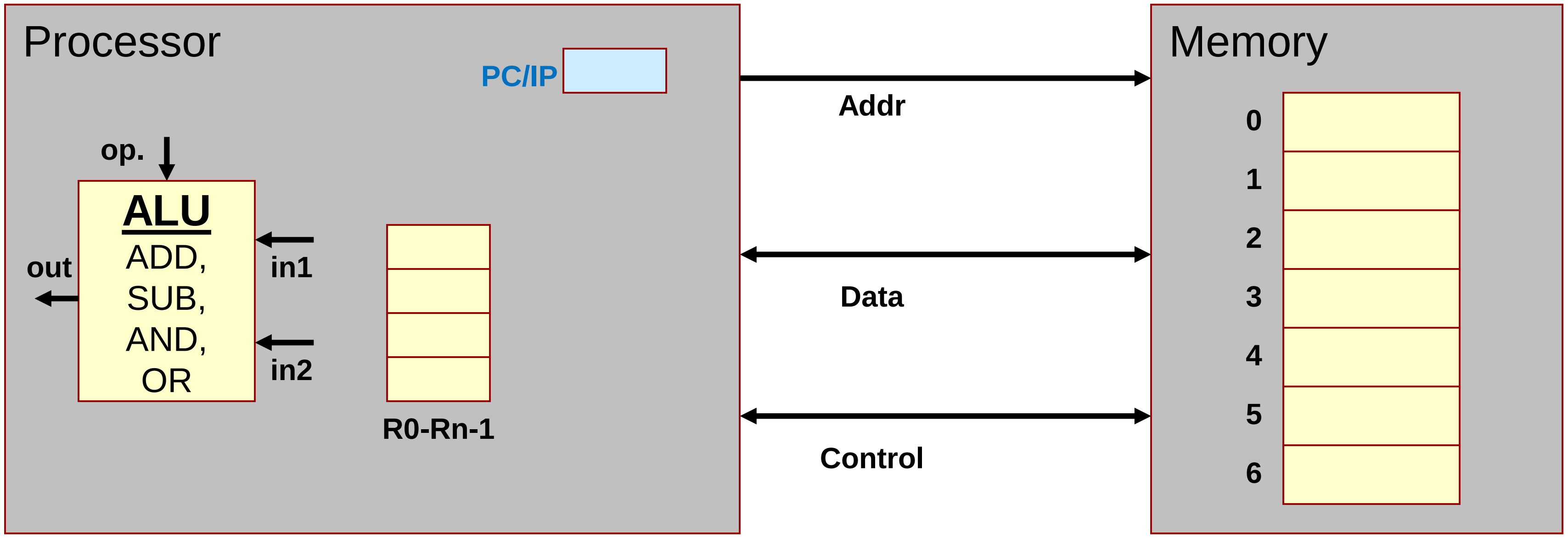
What if we have registers?

- Example w/ registers: F = (X+Y) – (X*Y)
  - Load X and Y into registers
  - 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



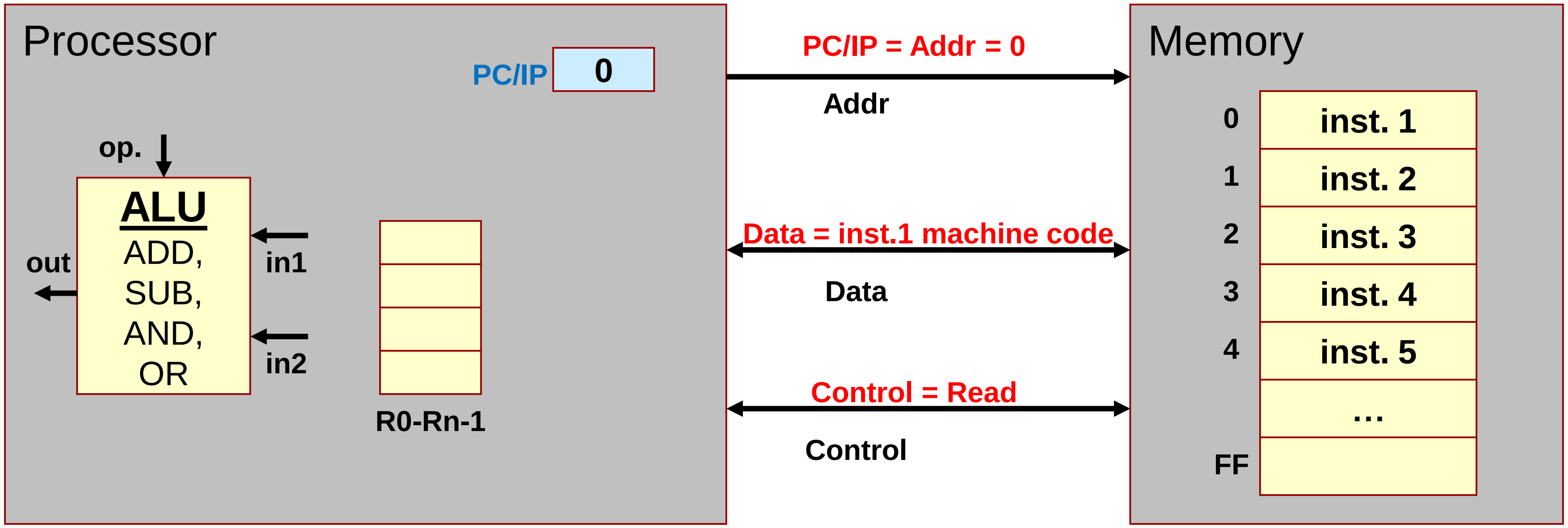
Other Registers

- 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



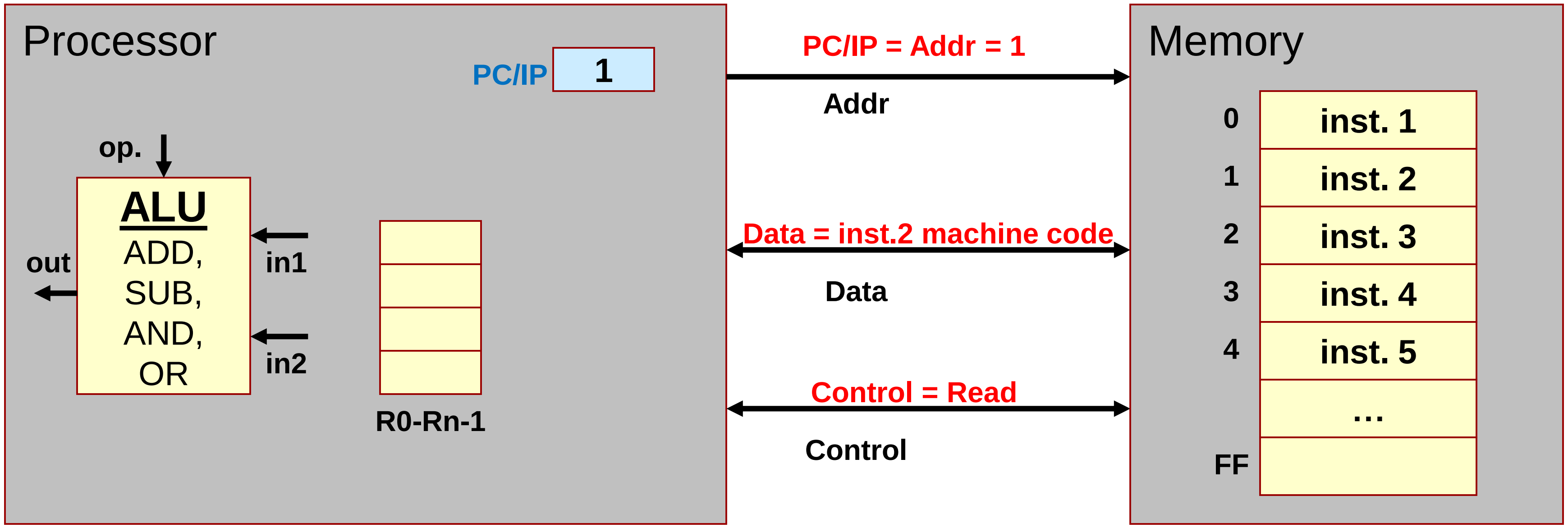
Fetching an Instruction

- 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



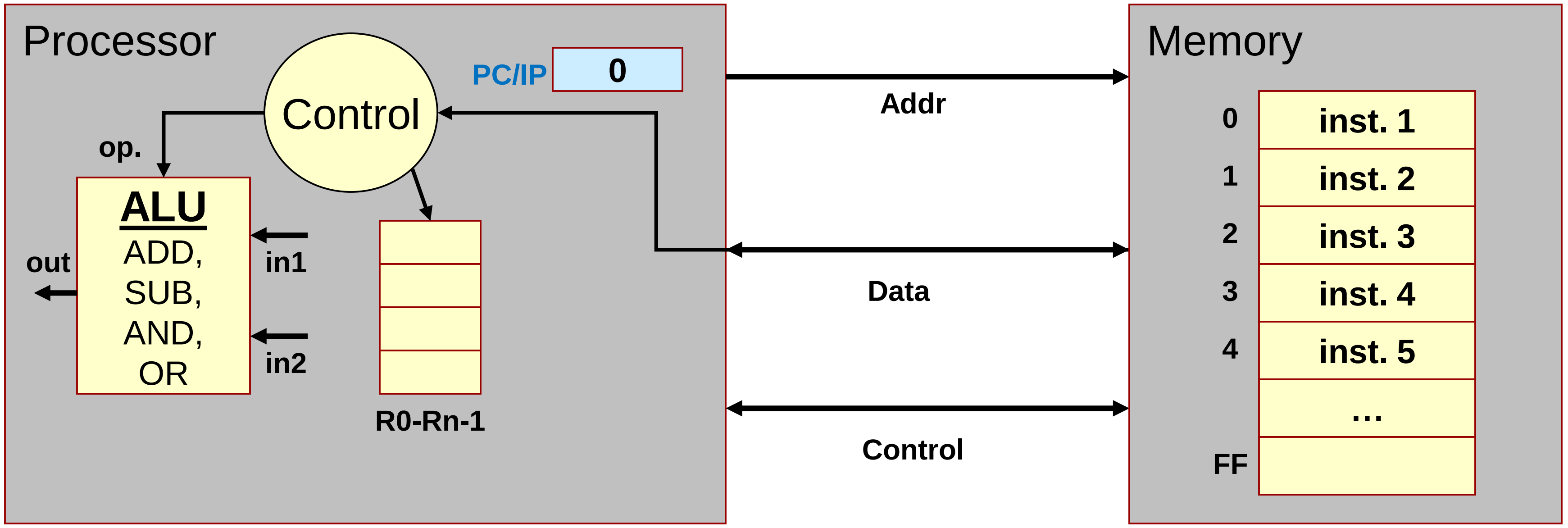
Fetching an Instruction

- 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



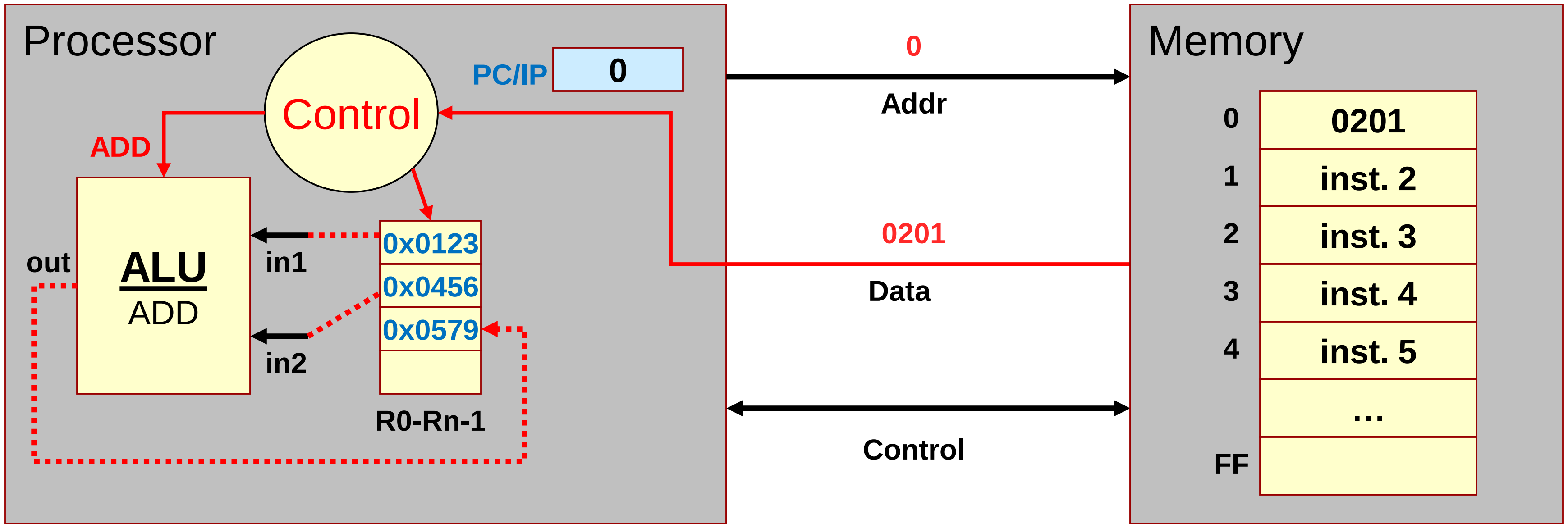
Control Circuitry

- 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 and 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)



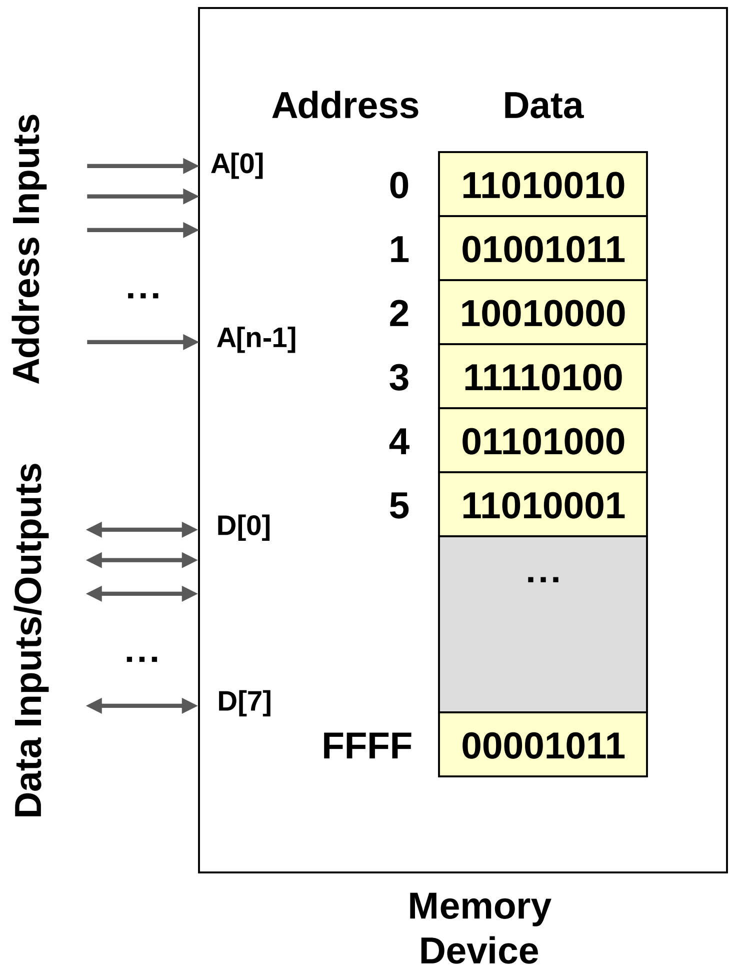
Summary

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

UNDERSTANDING MEMORY

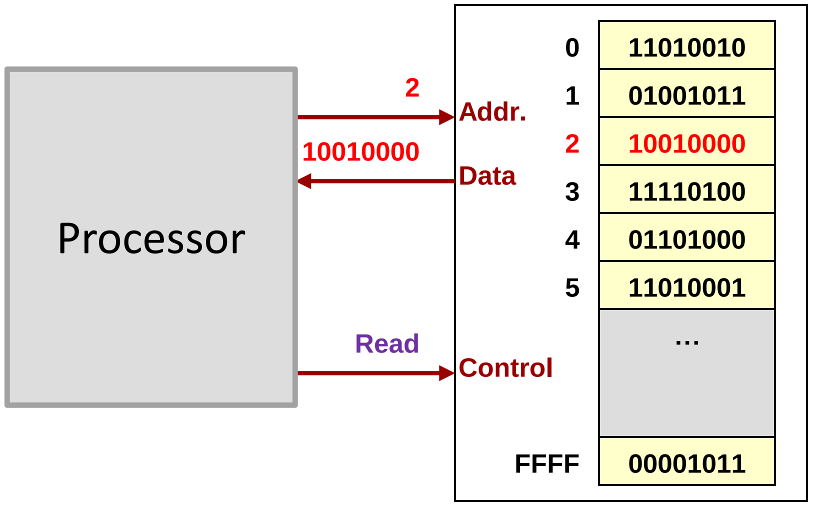
Memory and Addresses

- 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

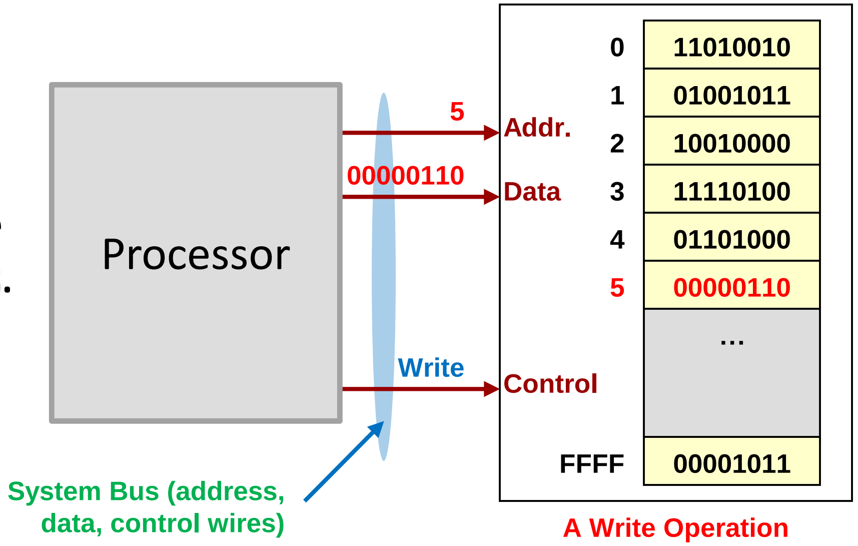


Reads & Writes

- 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.



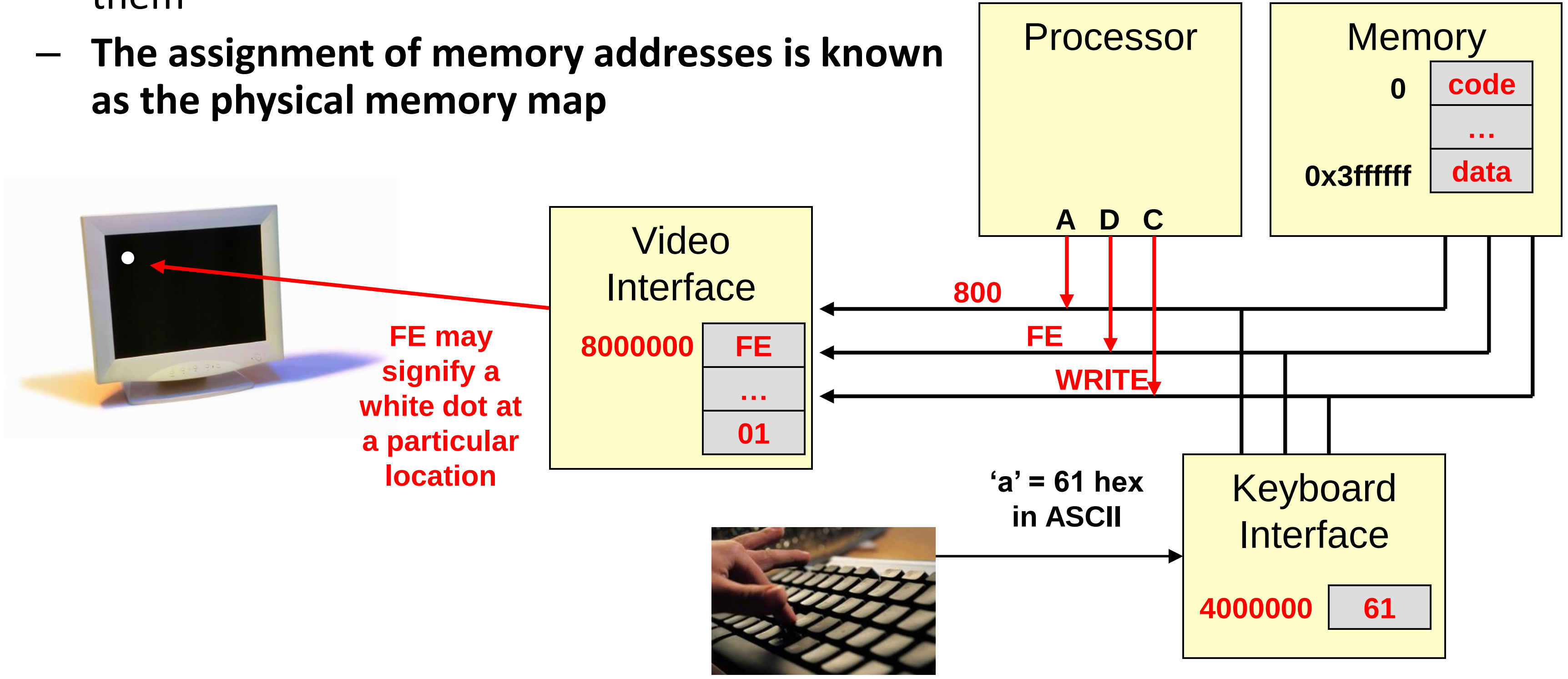
A Read Operation



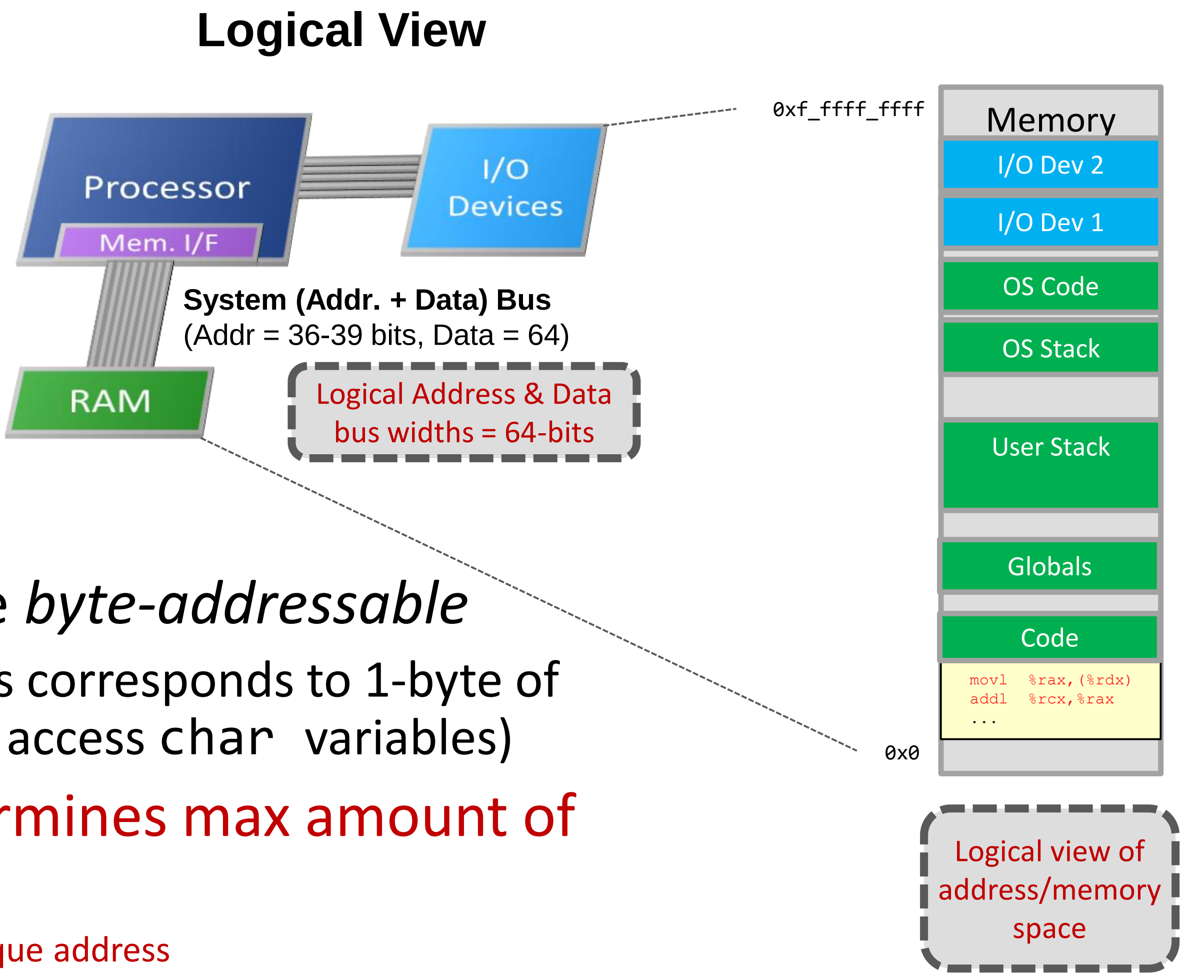
A Write Operation

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** which are used to identify them
  - **The assignment of memory addresses is known as the physical memory map**



Address Space Size and View


Logical View
Processor
Mem. I/F
I/O Devices
RAM
System (Addr. + Data) Bus
(Addr = 36-39 bits, Data = 64)
Logical Address & Data bus widths = 64-bits
0xf_ffff_ffff
Memory
I/O Dev 2
I/O Dev 1
OS Code
OS Stack
User Stack
Globals
Code
0x0
Logical view of address/memory space

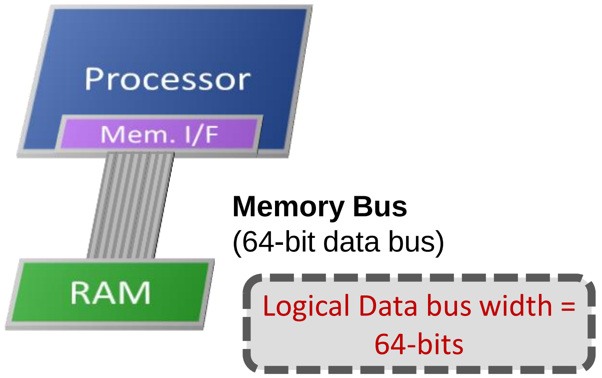
```
movl  %rax,(%rdx)
addl  %rcx,%rax
...
```

- 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 & Data Sizes

- 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 (i.e. 64-bits)
  - 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 Defined
  - Byte (B)
    - 8-bits
  - Word (W)
    - 16-bits = 2 bytes
  - Double word (L)
    - 32-bits = 4 bytes
  - Quad word (Q)
    - 64-bits = 8 bytes

Floating Point

- 3 Sizes Defined
  - 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)
- But, 64-bit logical data bus allows us to 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

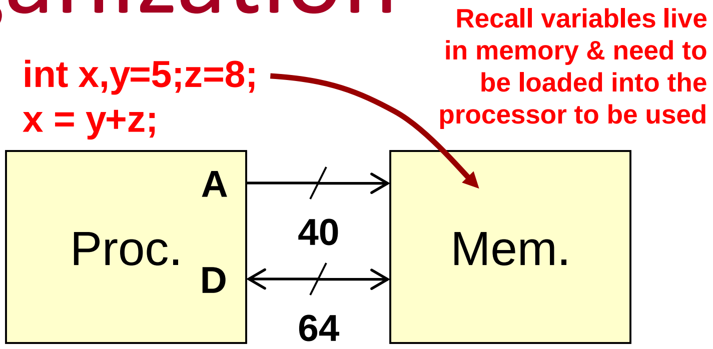

int x,y=5;z=8;
x = y+z;
Recall variables live in memory & need to be loaded into the processor to be used
Proc.
A
D
40
64
Mem.

Fig. 2

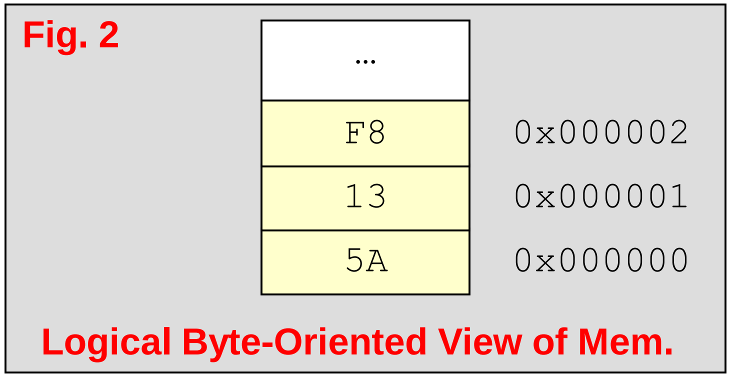
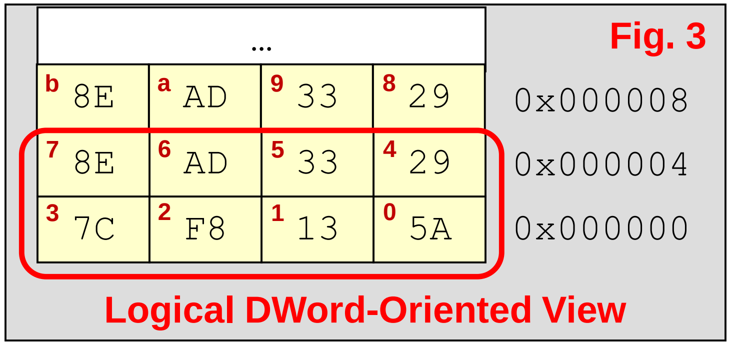


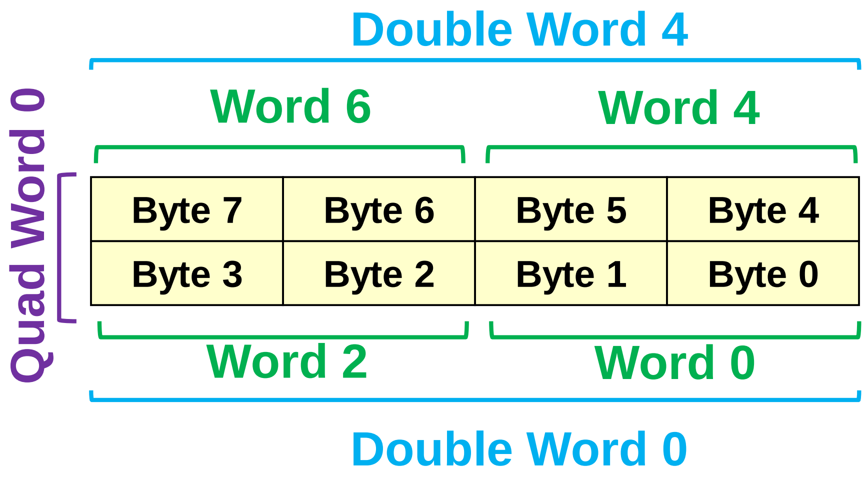
Fig. 3

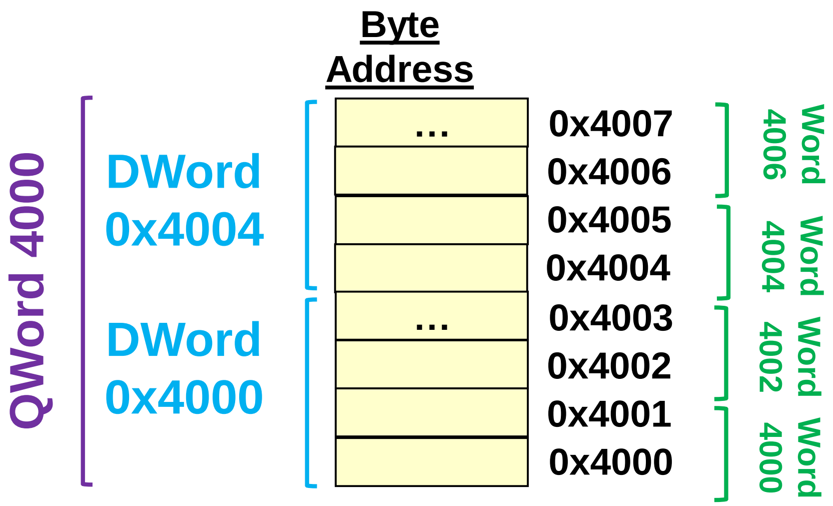

...
b 8E a AD 9 33 8 29 0x000008
7 8E 6 AD 5 33 4 29 0x000004
3 7C 2 F8 1 13 0 5A 0x000000
Logical DWord-Oriented View

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, D, L**
- There are rules for valid starting addresses
  - A valid starting address must 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

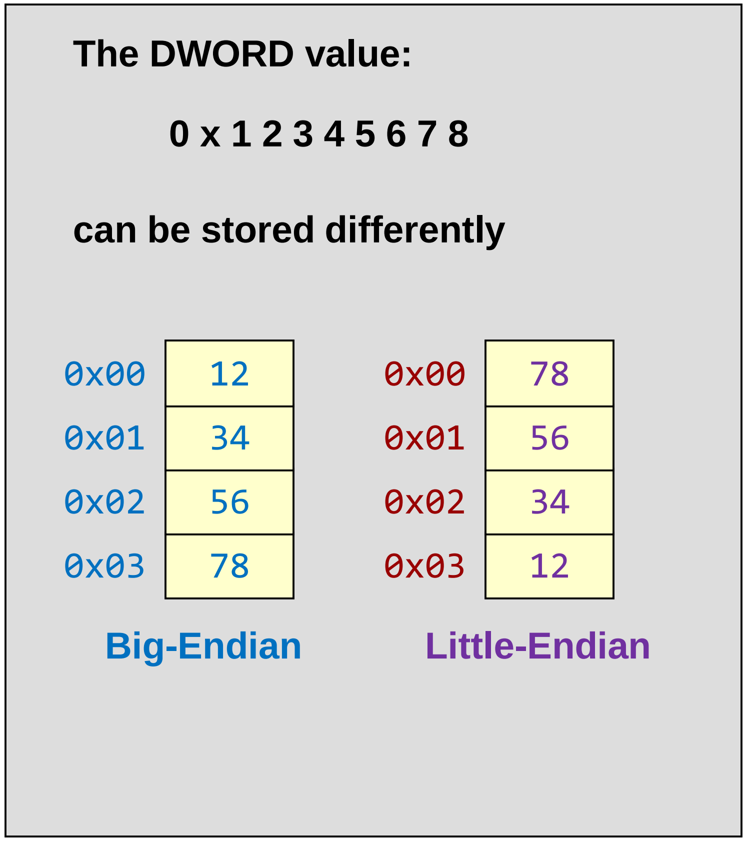




Endian-ness

CS:APP 2.1.3

- **Endian-ness** refers to the two alternate methods of ordering the **bytes** in a larger unit (word, DWORD, etc.)
  - Big-Endian
    - PPC, Sparc
    - *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 1 2 3 4 5 6 7 8

can be stored differently

0x00	12
0x01	34
0x02	56
0x03	78

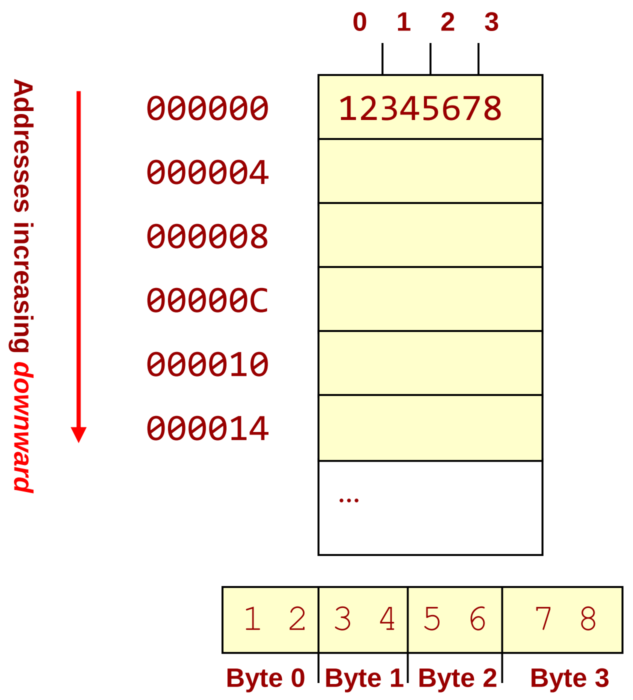
Big-Endian

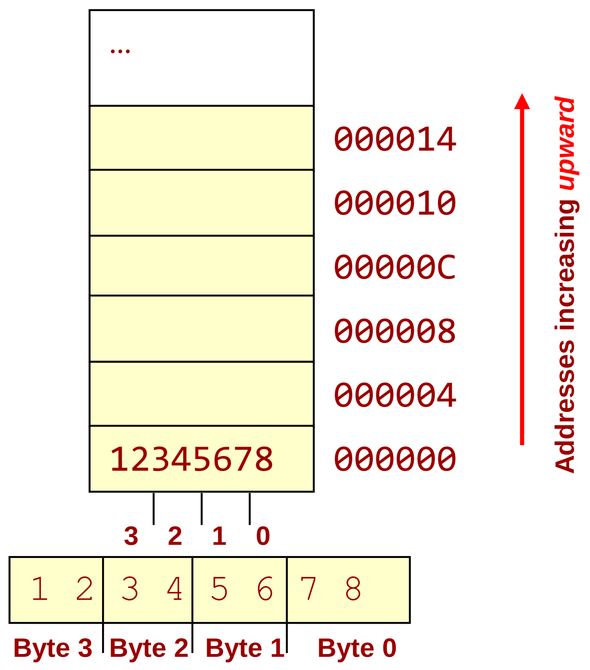
0x00	78
0x01	56
0x02	34
0x03	12

Little-Endian

Big-endian vs. Little-endian

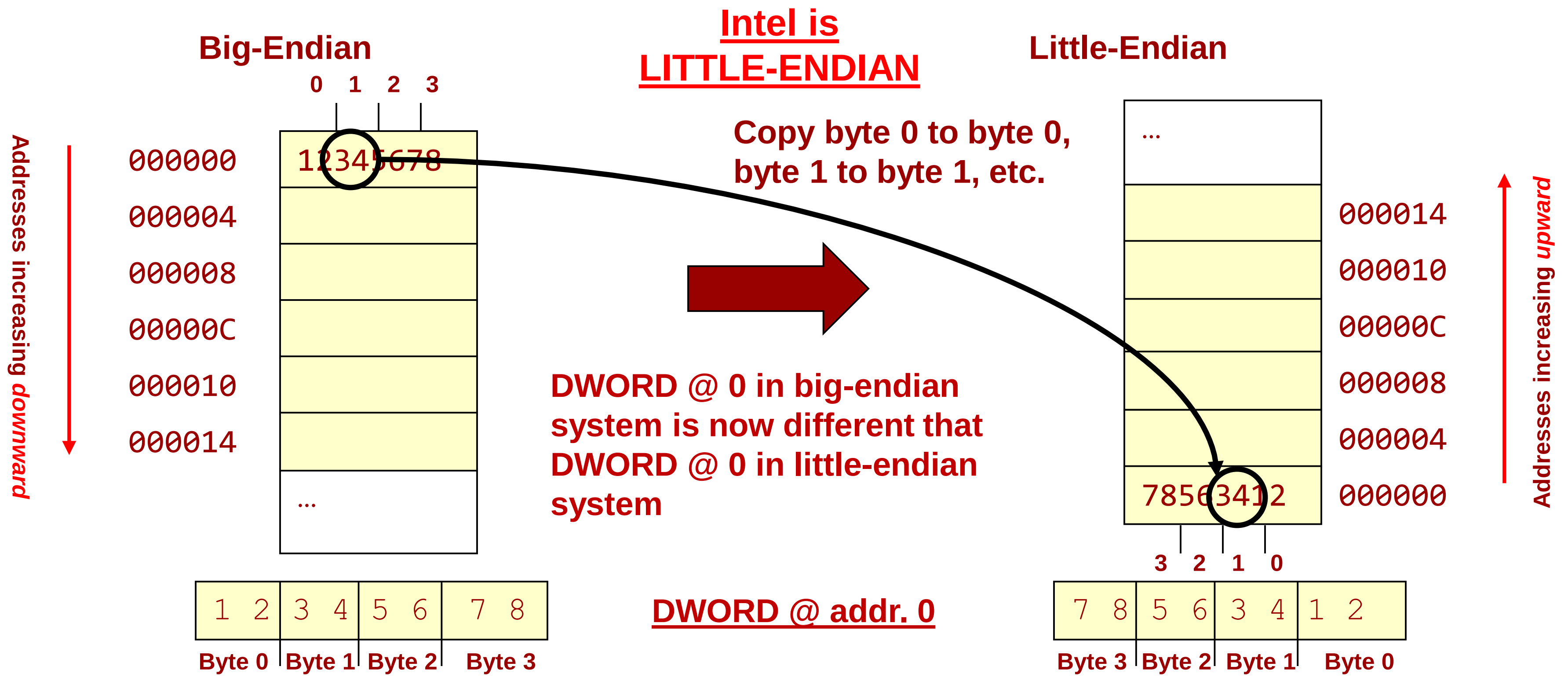
- Big-endian
  - makes sense if you view your memory as starting at the top-left and addresses increasing as you go down
- Little-endian
  - makes sense if you view your memory as starting at the bottom-right and addresses increasing as you go up





Big-endian vs. Little-endian

- Issues 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)



Big-Endian
Intel is LITTLE-ENDIAN
Little-Endian
0 1 2 3
Addresses increasing downward
000000
000004
000008
00000C
000010
000014
12345678
...
Copy byte 0 to byte 0, byte 1 to byte 1, etc.
DWORD @ 0 in big-endian system is now different that DWORD @ 0 in little-endian system
000014
000010
00000C
000008
000004
000000
78563412
3 2 1 0
Addresses increasing upward
1 2 | 3 4 | 5 6 | 7 8
Byte 0 Byte 1 Byte 2 Byte 3
DWORD @ addr. 0
7 8 | 5 6 | 3 4 | 1 2
Byte 3 Byte 2 Byte 1 Byte 0

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 4
- x86 uses little-endian
  - The start address of a word (or dword or qword) refers to the LS-byte

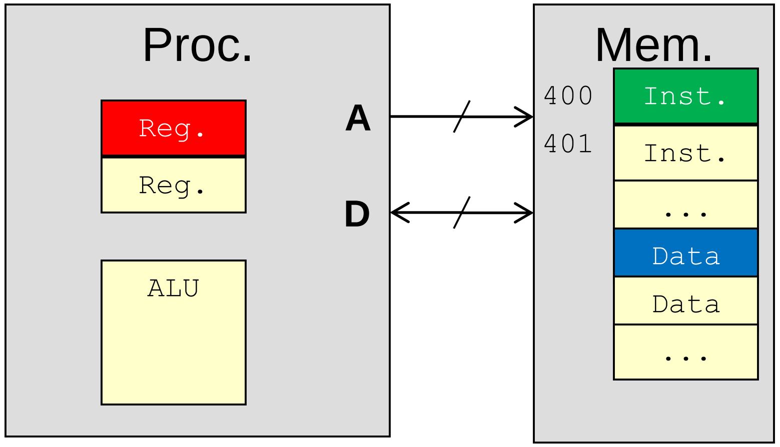
X86-64 ASSEMBLY

x86-64 Instruction Classes

- Data Transfer (mov instruction)
  - Moves data between processor & memory (loads and saves variables between processor and memory)
  - One operand must be a processor register (can't move data from one memory location to another)
  - Specifies size via a suffix on the instruction (`movb`, `movw`, `movl`, `movq`)
- ALU Operations
  - One operand must be a processor register
  - Size and operation specified by instruction (`addl`, `orq`, `andb`, `subw`)
- Control / Program Flow
  - Unconditional/Conditional Branch (`cmpq`, `jmp`, `je`, `jne`, `jl`, `jge`)
  - Subroutine Calls (`call`, `ret`)
- Privileged / System Instructions
  - Instructions that 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)
  - A value in a memory location (e.g. value at address 0x0200e8)
  - A constant stored in the instruction itself (known as an 'immediate' value)
    [e.g. ADDI $1,D0]
  - The $ indicates the constant/immediate
- Destination operands must be
  - A register
  - A memory location (specified by its address)

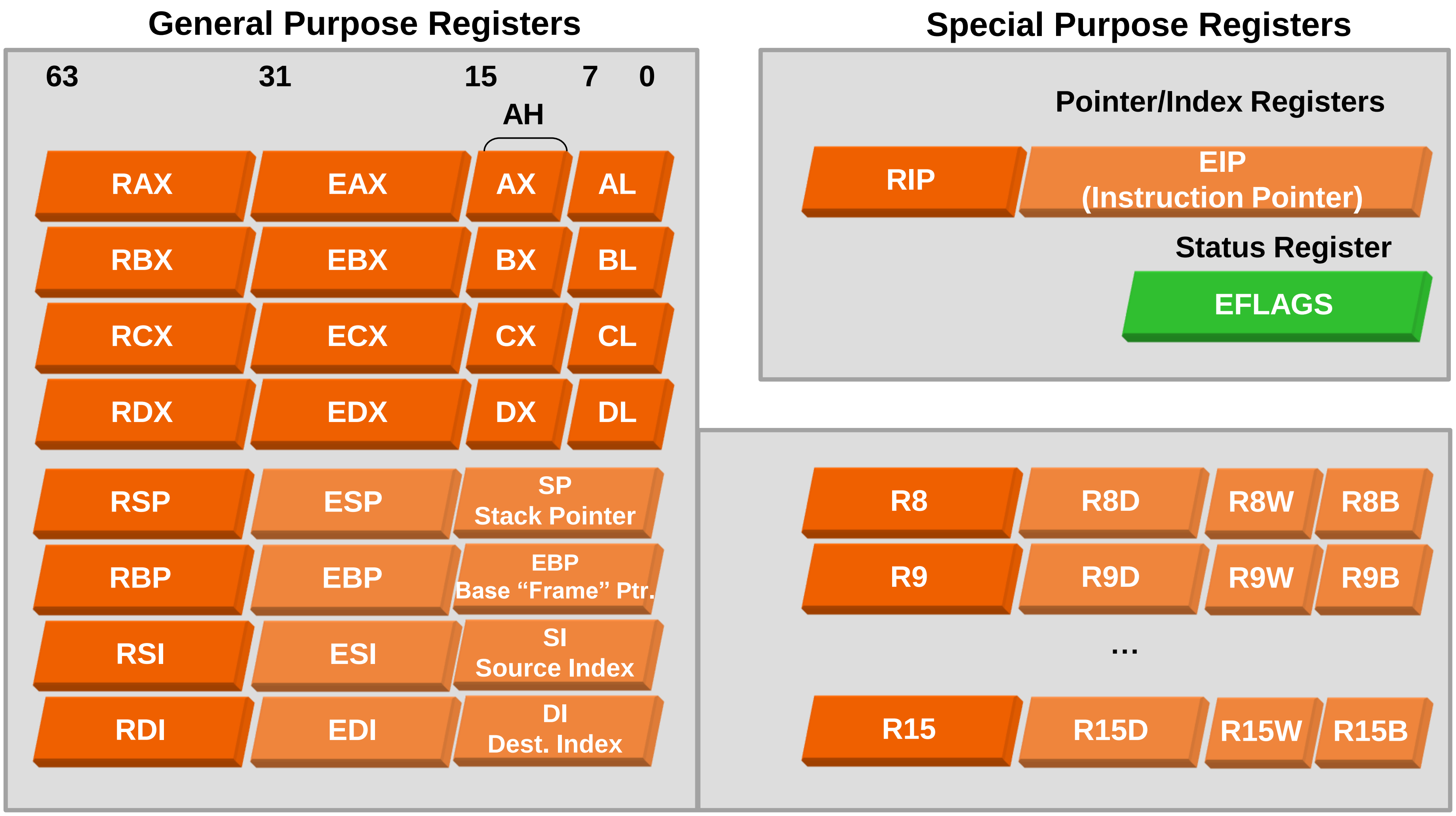


Intel x86 Register Set

- 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 referenced the high 8-bits, AX referenced the full 16-bit value
- 32-/64-bit processors (see next slide)

Intel (IA-32/64) Architectures

CS:APP 3.4



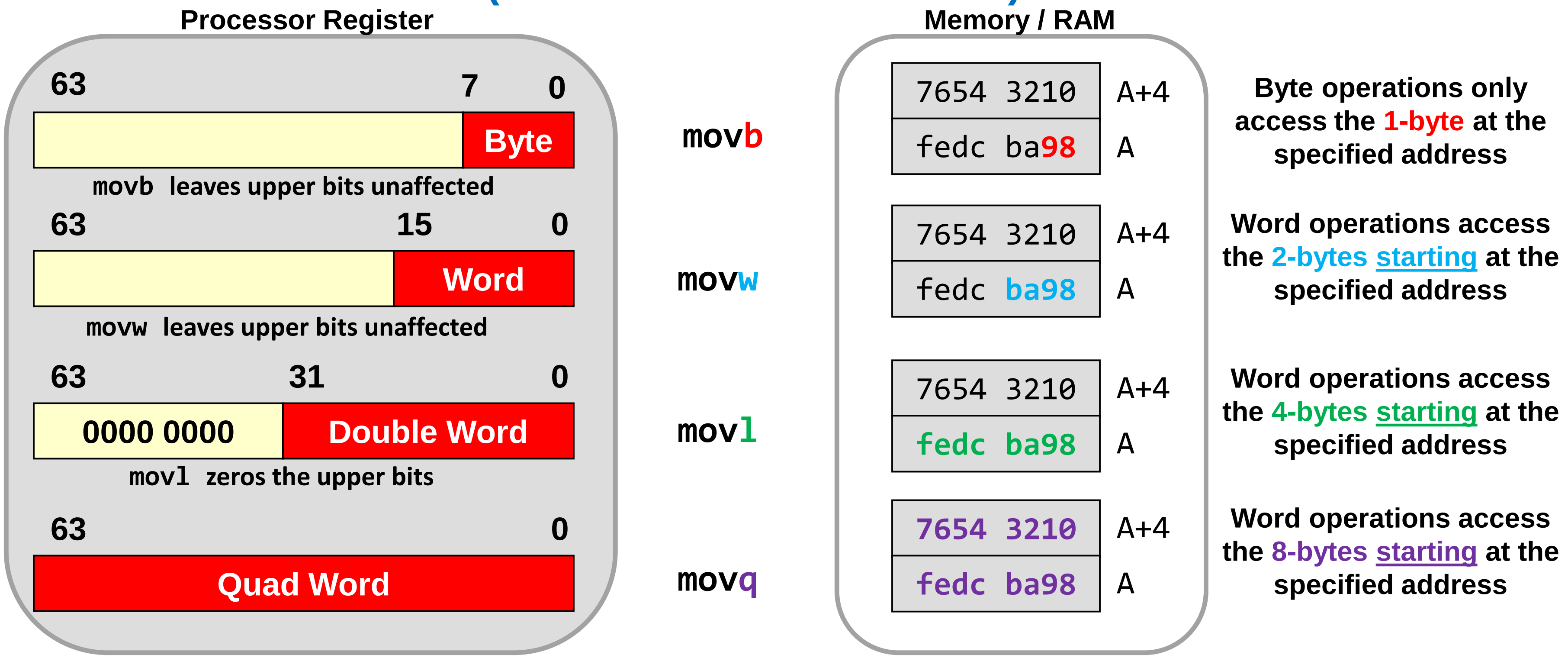
DATA TRANSFER INSTRUCTIONS

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)



Byte operations only access the 1-byte at the specified address

Word operations access the 2-bytes starting at the specified address

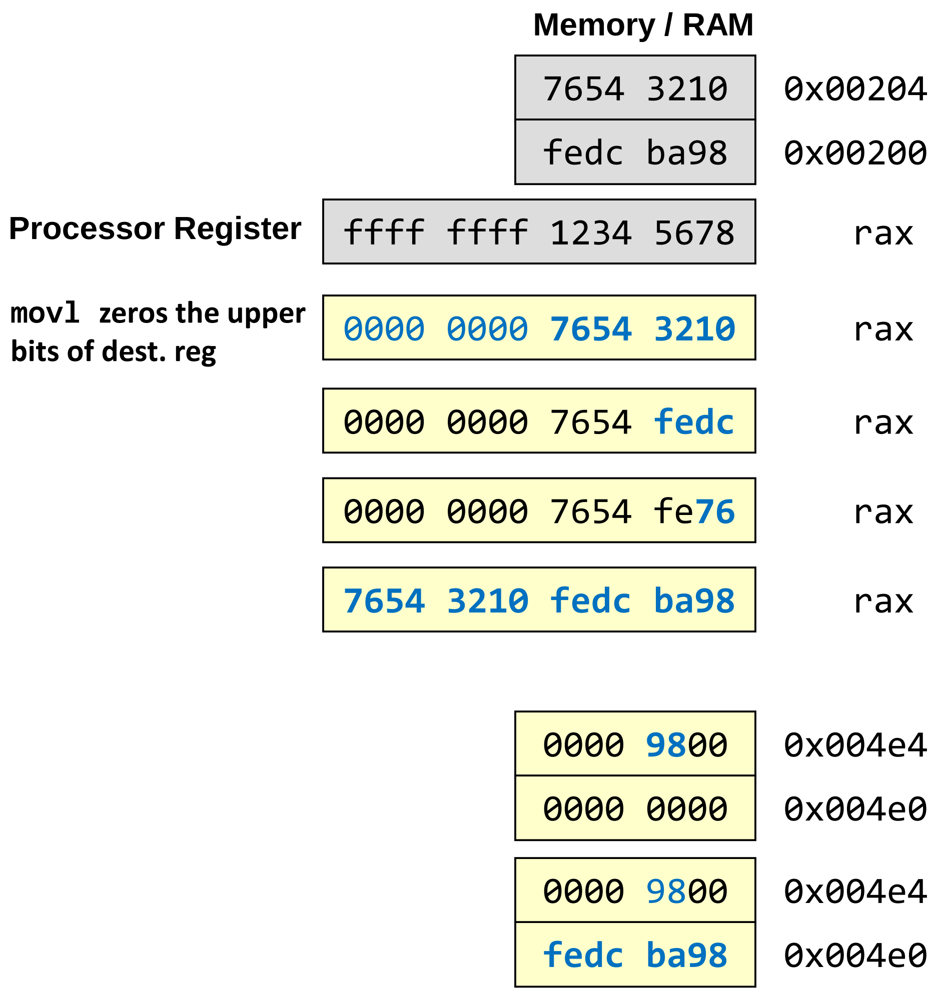
Word operations access the 4-bytes starting at the specified address

Word operations access the 8-bytes starting at the specified address

Mem/Register Transfer Examples

- `mov[b,w,l,q] src, dst`
- Initial Conditions:
  - `movl 0x204, %eax`
  - `movw 0x202, %ax`
  - `movb 0x207, %al`
  - `movq 0x200, %rax`

  - `movb %al, 0x4e5`
  - `movl %eax, 0x4e0`



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

Immediate Examples

- Immediate Examples

Memory / RAM

7654 3210	0x00204
fedc ba98	0x00200

Processor Register	ffff ffff 1234 5678	rax
– `movl     $0xfe1234, %eax`	0000 0000 00fe 1234	rax
– `movw     $0xaa55, %ax`	0000 0000 00fe aa55	rax
– `movb     $20, %al`	0000 0000 00fe aa14	rax
– `movq     $-1, %rax`	ffff ffff ffff ffff	rax
– `movabsq  $0x123456789ab, %rax`	0000 0123 4567 89ab	rax
– `movq     $-1, 0x4e0`	ffff ffff	0x004e4
	ffff ffff	0x004e0

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

Move Variations

- There are several variations when the destination of a `mov` instruction is a register
  - This only applies when the destination is a register
- Normal `mov` **<u>does not affect upper portions</u>** of registers (with exception of `movl`)
- `movz`*`xy`* will zero-extend the upper portion
  - `movzbw` (move a byte from the source but zero-extend it to a word in the dest. register)
  - `movzbw, movzbl, movzbq, movzwl, movzwq`
- `movs`*`xy`* will sign-extend the upper portion
  - `movsbw` (move a byte from the source but sign-extend it to a word in the dest. register)
  - `movsbl, movsbl, movsbq, movswl, movswq, movslq`

Zero/Signed Move Variations

- Initial Conditions:

Memory / RAM

	7654 3210	0x00204
	fedc ba98	0x00200
Processor Register	0123 4567 89ab cdef	rdx
– movslq 0x200, %rax	ffff ffff **fedc ba98**	rax
– movzwl 0x202, %eax	ffff ffff **0000 fedc**	rax
– movsbw 0x201, %ax	ffff ffff 0000 **ffba**	rax
– movsbl 0x206, %eax	ffff ffff **0000 0054**	rax
– movzbq %dl, %rax	**0000 0000 0000 00ef**	rax

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

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

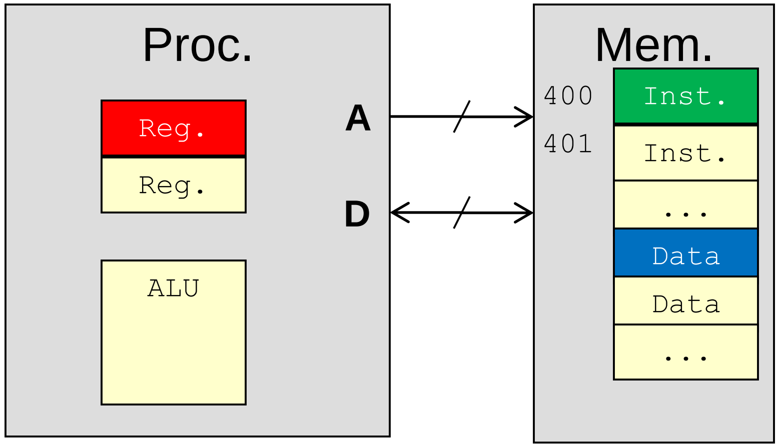
Summary

- To access different size portions of a register requires different names in x86 (e.g. AL, AX, EAX, RAX)
- Moving to a register may involve zero- or sign-extending since registers are 64-bits
  - Long (dword) operations always 0-extend the upper 32-bits
- Moving to memory never involves zero- or sign-extending since it memory is broken into finer granularities

ADDRESSING MODES

What Are Addressing Modes

- Recall an operand must be:
  - A register value (e.g. %rax)
  - A value in a memory location
  - An immediate
- 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 [e.g. accessing an array element, a[i] vs. object member, obj.member]
- The ways to specify the operand location are known as addressing modes



Common x86-64 Addressing Modes

CS:APP 3.4.1

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

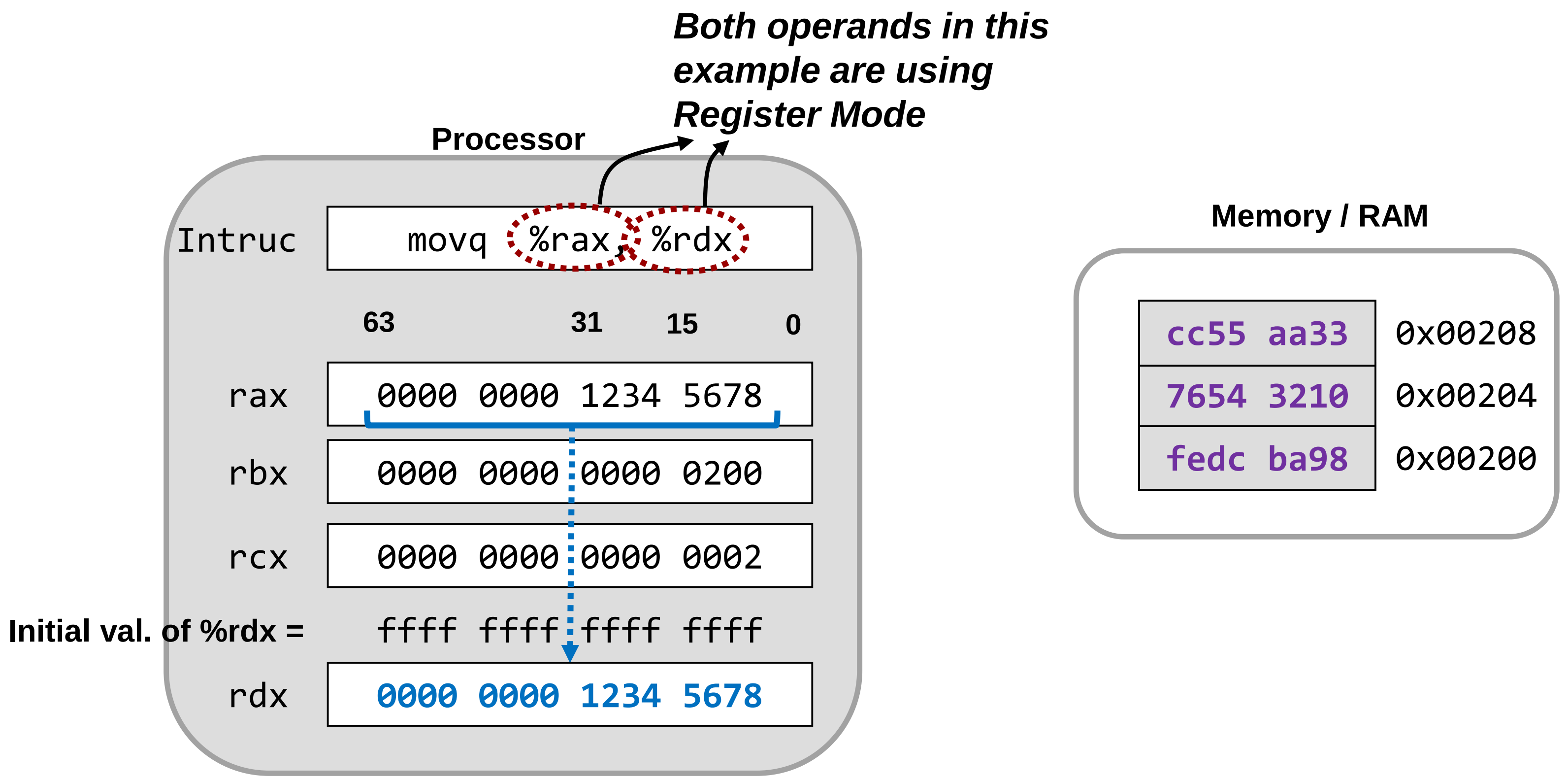
†Known as the scale factor and can be {1,2,4, or 8}

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

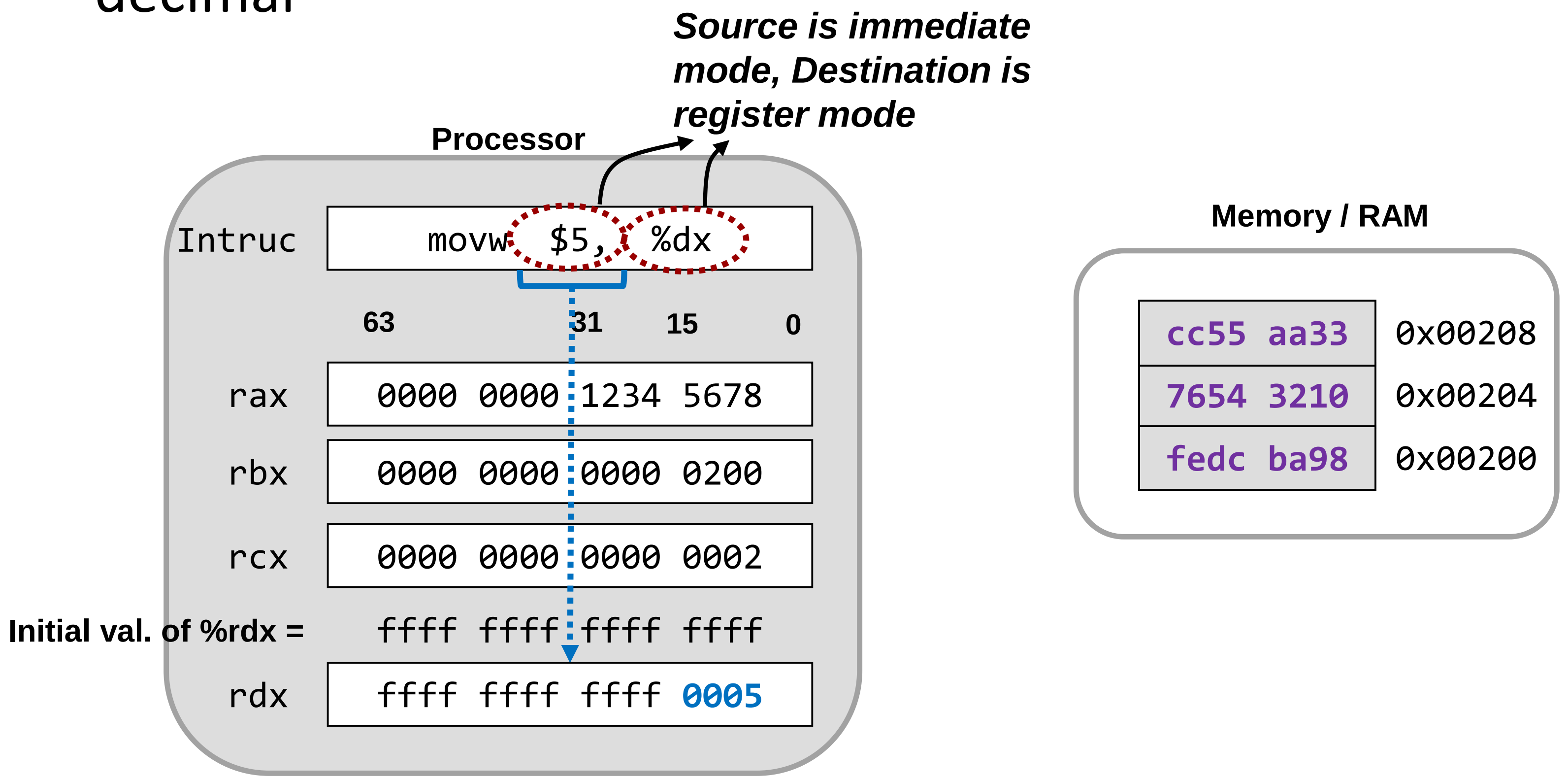
Register Mode

- Specifies the contents of a register as the operand



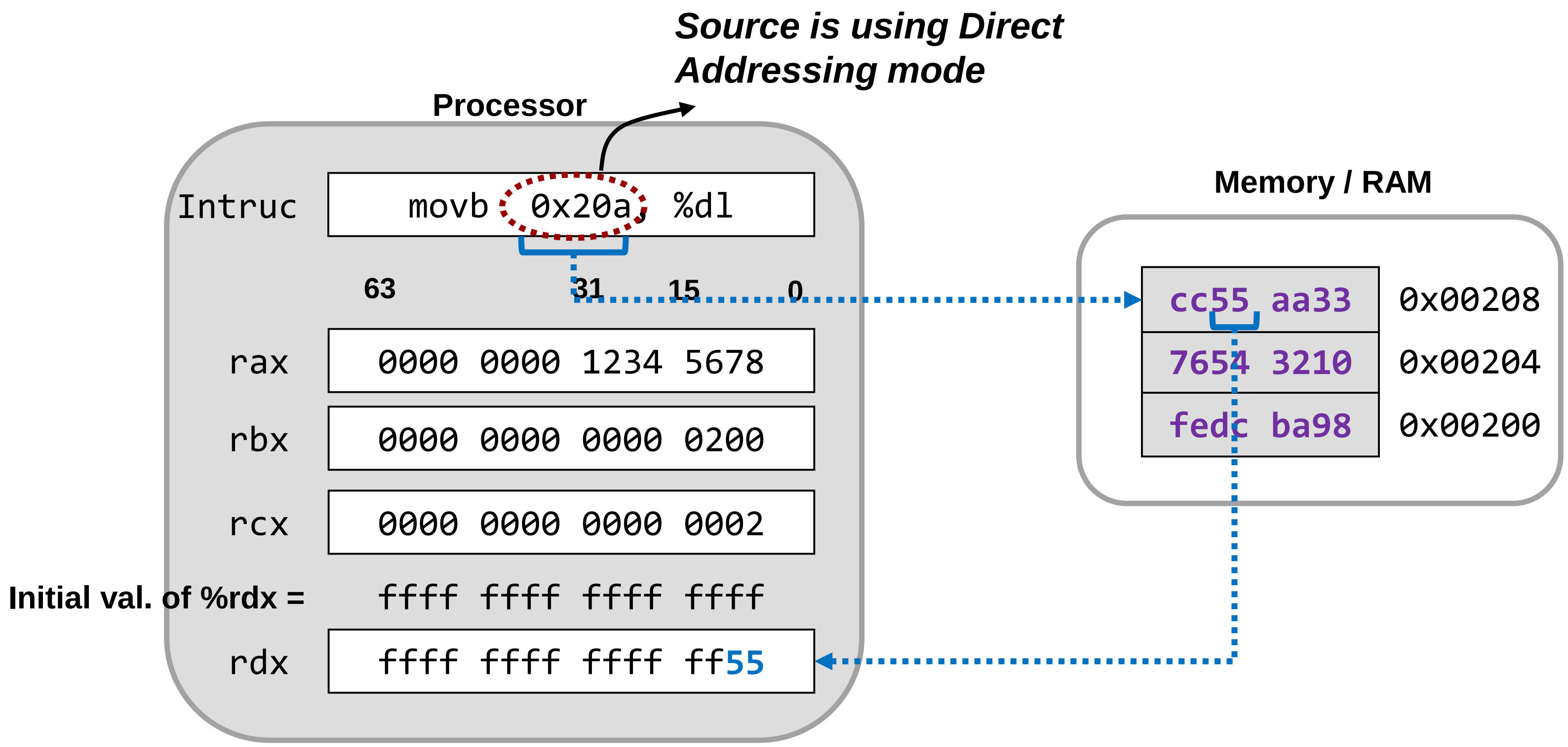
Immediate Mode

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



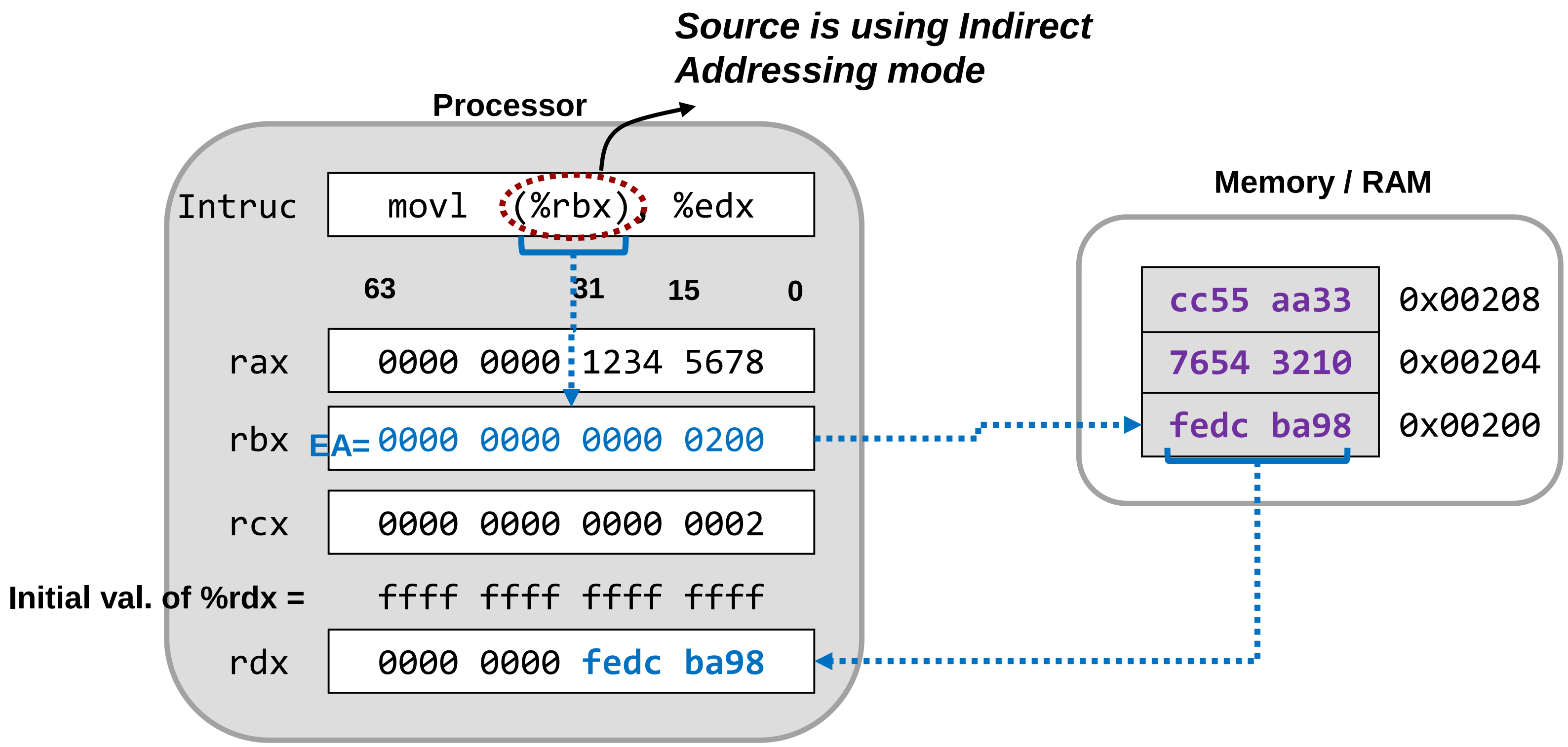
Direct Addressing Mode

- Specifies a constant memory address where the true operand is located
- Address can be specified in decimal or hex



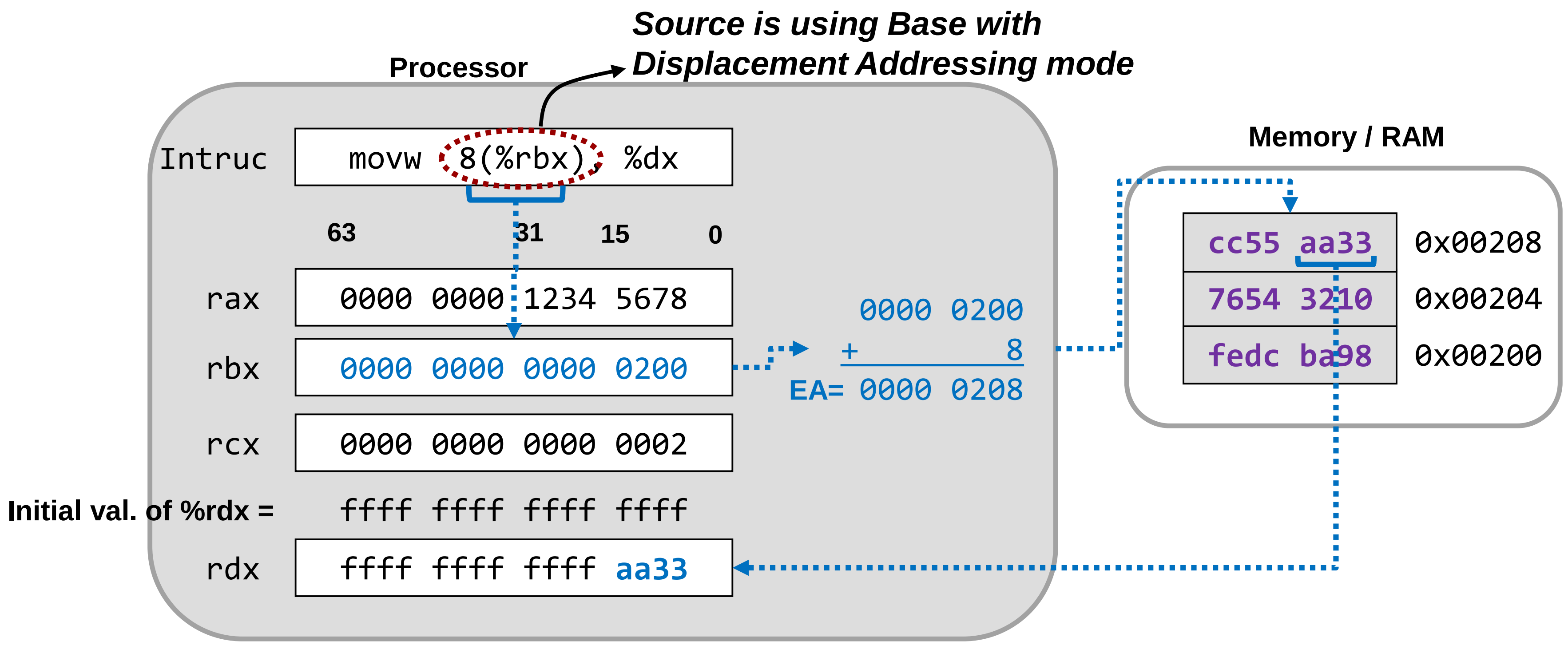
Indirect Addressing Mode

- Specifies a register whose value will be used as the effective address in memory where the true operand is located
  - Similar to dereferencing a pointer
- Parentheses indicate indirect addressing mode



Base/Indirect with Displacement Addressing Mode

- Form: d(%reg)
- Adds a constant displacement to the value in a register and uses the sum as the effective address of the actual operand in memory



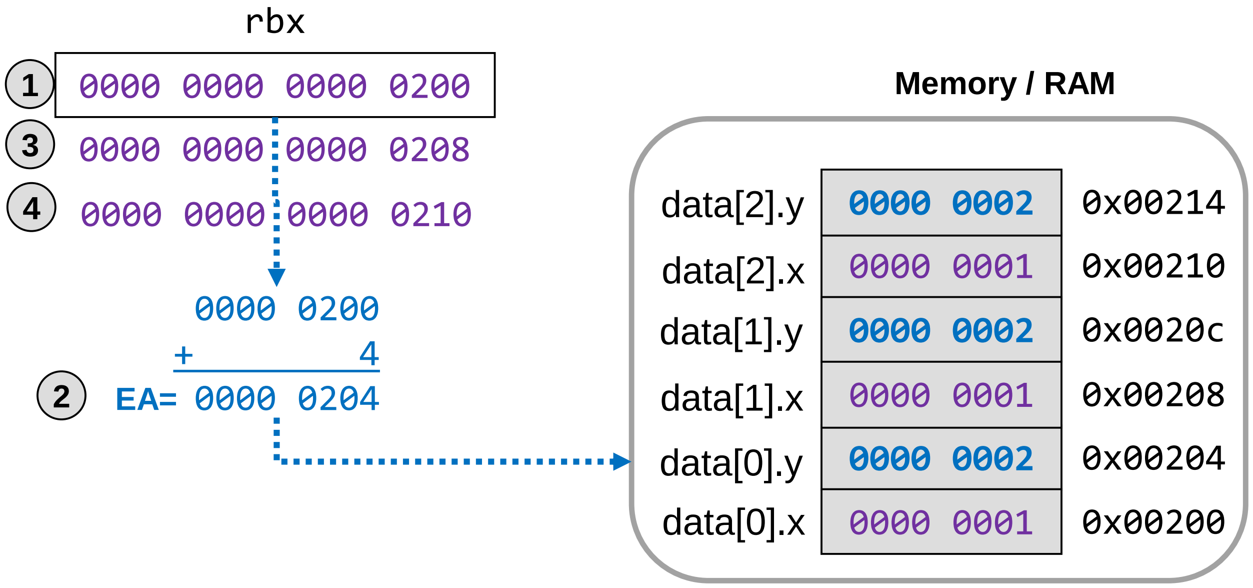
Base/Indirect with Displacement Example

- Useful for access members of a struct or object

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

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

C Code

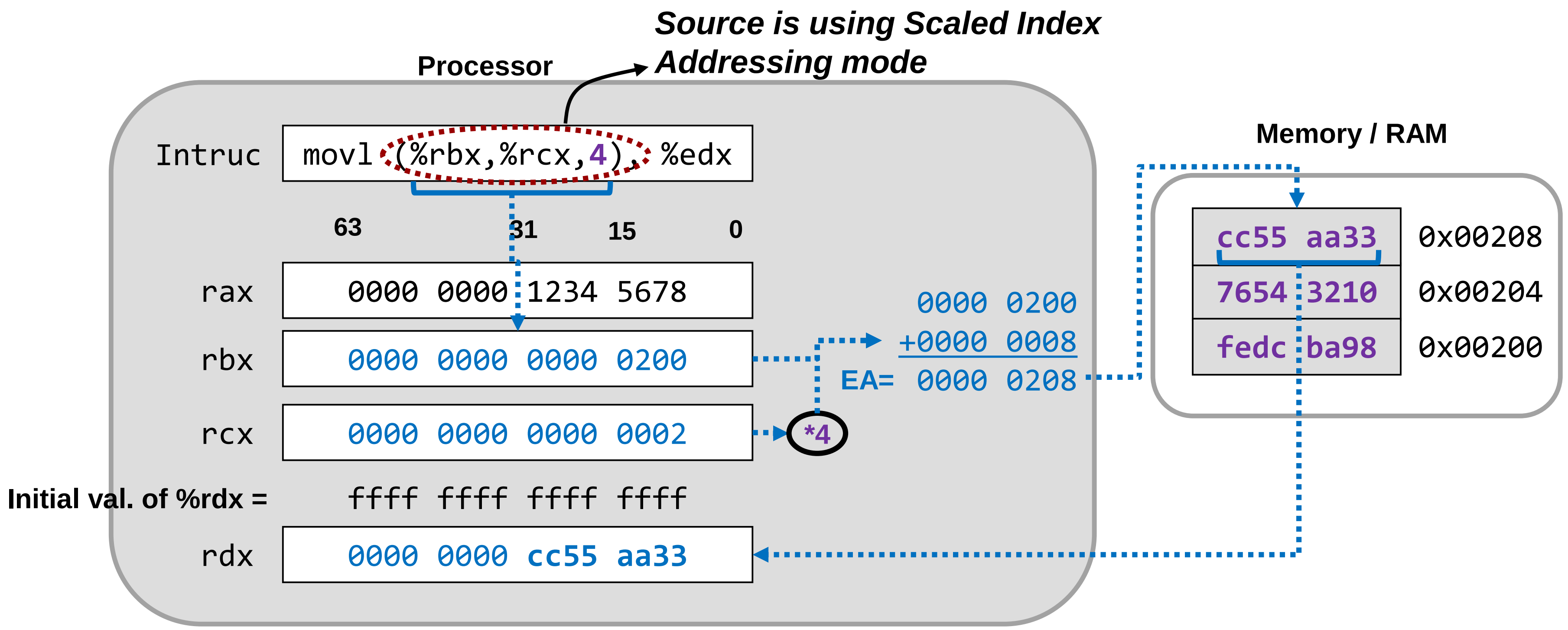

rbx
1 0000 0000 0000 0200
3 0000 0000 0000 0208
4 0000 0000 0000 0210
0000 0200
+ 4
2 EA= 0000 0204
Memory / RAM
data[2].y 0000 0002 0x00214
data[2].x 0000 0001 0x00210
data[1].y 0000 0002 0x0020c
data[1].x 0000 0001 0x00208
data[0].y 0000 0002 0x00204
data[0].x 0000 0001 0x00200

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

Assembly

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



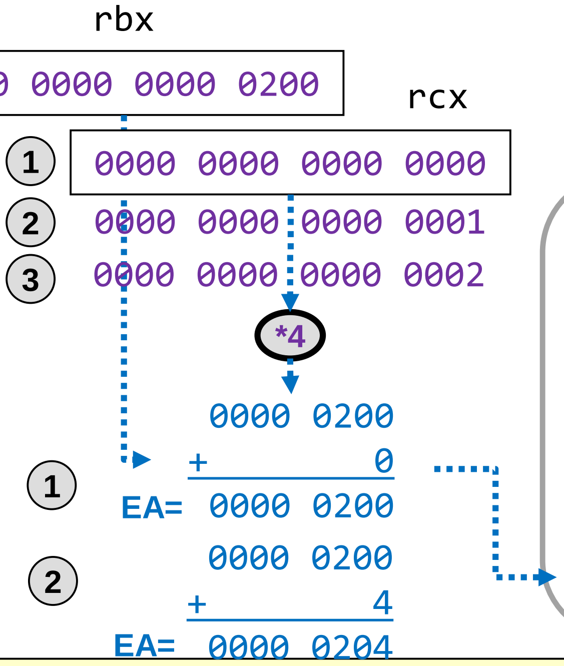
Scaled Index Addressing Mode Example

- Useful for accessing array elements

```
int data[6];

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

C Code



Memory / RAM

data[5]	0000 0005	0x00214
data[4]	0000 0004	0x00210
data[3]	0000 0003	0x0020c
data[2]	0000 0002	0x00208
data[1]	0000 0001	0x00204
data[0]	0000 0000	0x00200

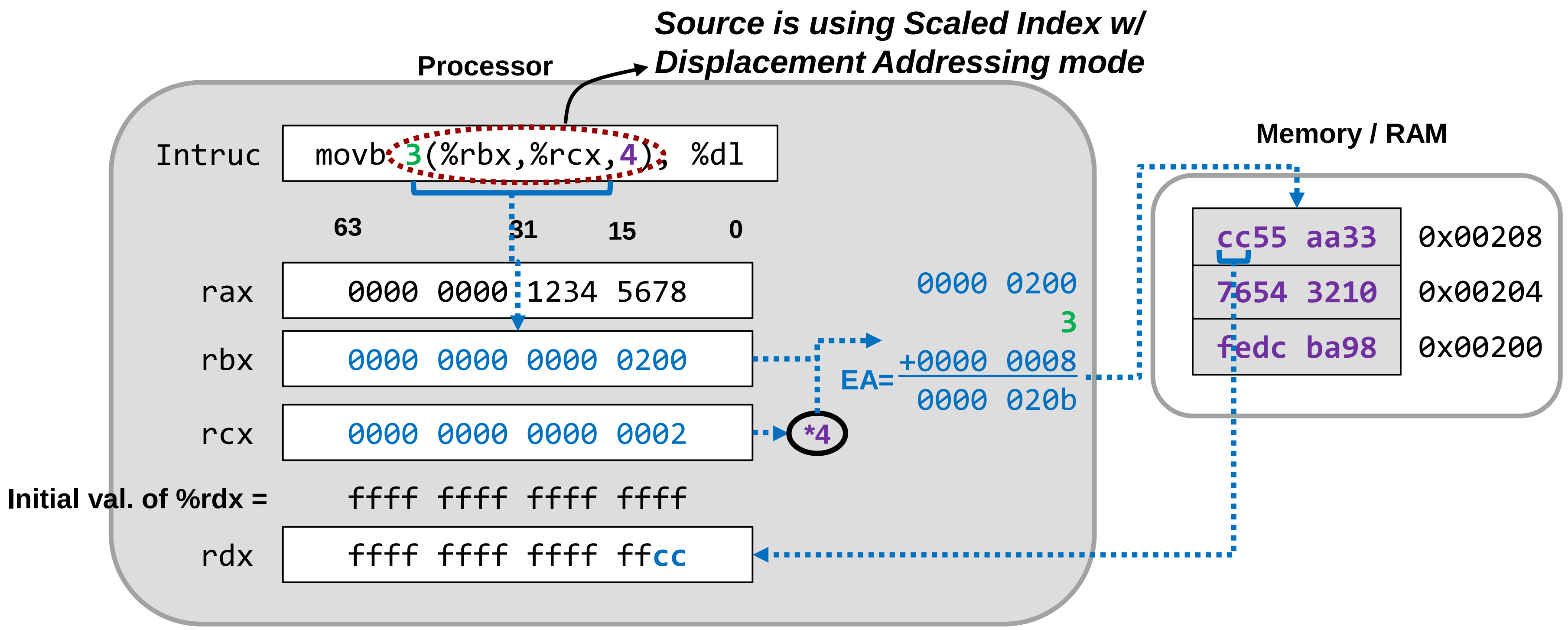
Array of:
- chars/bytes => Use s=1
- shorts/words => Use s=2
- ints/floats/dwords => Use s=4
- long longs/doubles/qwords => Use s=8

```
movq    $0x0200,%rbx
movl    $0, %rcx
loop 6 times {
   movl   %rcx, (%rbx,%rcx,4)
   addl   $1, %rcx
}
```

Assembly

Scaled Index w/ Displacement Addressing Mode

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



Addressing Mode Exercises

Processor Registers

0000 0000 0000 0200	rbx
0000 0000 0000 0003	rcx

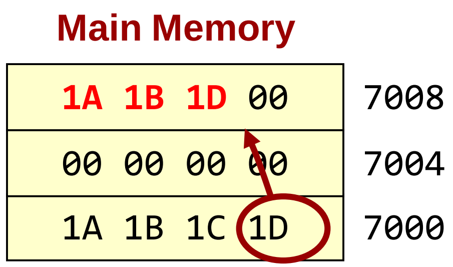
Memory / RAM

cdef 89ab	0x00204
7654 3210	0x00200
f00d face	0x001fc
dead beef	0x001f8

– movq (%rbx), %rax	cdef 89ab 7654 3210	rax
– movl -4(%rbx), %eax	0000 0000 f00d face	rax
– movb (%rbx,%rcx), %al	0000 0000 f00d fa76	rax
– movw (%rbx,%rcx,2), %ax	0000 0000 f00d cdef	rax
– movsbl -16(%rbx,%rcx,4), %eax	0000 0000 ffff ffce	rax
– movw %cx, 0xe0(%rbx,%rcx,2)	0000 0000	0x002e8
	0003 0000	0x002e4

Addressing Mode Examples

		%eax	%ecx	%edx
1	`movl $0x7000,%eax`	0x0000 7000		
2	`movl $2,%ecx`		0x0000 0002	
3	`movb (%eax),%dl`			0x0000 001d
4	`movb %dl,9(%eax)`			
5	`movw (%eax,%ecx),%dx`			0x0000 1a1b
6	`movw %dx,6(%eax,%ecx,2)`			



Main Memory

1A 1B 1D 00	7008
00 00 00 00	7004
1A 1B 1C 1D	7000

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
- Primary restriction is both operands cannot be memory locations
  - `movl 2000, (%eax)` is not allowed since both source and destination are in memory
  - 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`
  - `Mem -> Reg`
  - `Reg -> Mem`

Summary

- Addressing modes provide variations for how to specify the location of an operand
- EA = Effective Address
  - Computed address used to access memory

ARITHMETIC INSTRUCTIONS

ALU Instruction(s)

CS:APP 3.5

- Performs arithmetic/logic operation on the given size of data
- Restriction: Both operands cannot be memory
- Format
  - `add[b,w,l,q] src2, src1/dst`
  - Example 1: `addq %rbx, %rax  (%rax += %rbx)`
  - Example 2: `subq %rbx, %rax  (%rax -= %rbx)`

Work from right->left->right

Arithmetic/Logic Operations

Memory / RAM

7654 3210	0x00204
0f0f ff00	0x00200

Processor Registers

ffff ffff 1234 5678	rdx
0000 0000 cc33 aa55	rax

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

0000 0000 cc34 cd55	rax
ffff ffff de69 23cd	rax
ffff ffff de69 **2300**	rax
ffff ffff de69 23**0f**	rax
ffff ffff de69 **2301**	rax
7655 5555	0x00204
0f0f ff00	0x00200

Rules:
- addl, subl, etc. zero out the upper 32-bits
- addq, subq, etc. can only support a 32-bit immediate (and will then 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)	`movl 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. 2 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
  – `lea 80(%rdx,%rcx,2),$rax; // $rax=80+%rdx+2*%rcx`
  – Computes the "address" and just puts it in the destination (doesn't load anything from memory)

lea Examples

- Initial Conditions

Processor Registers

0000 0000 0000 0020	rcx
0000 0089 1234 4000	rdx
ffff ffff ff00 0300	rbx

  - leal (%edx,%ecx),%eax
  - leaq -8(%rbx),%rax
  - leaq 12(%rdx,%rcx,2),%rax

0000 0000 1234 4020	rax
ffff ffff ff00 02f8	rax
0000 0089 1234 404c	rax

Rules:
- leal zeroes out the upper 32-bits

Optimization with lea

```
// x = %edi
int f1(int x)
{
  return 9x+1;
}
```

Original Code

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

    ret
```

Unoptimized Output

```
f1:
    leal  1(%edi,%edi,8),%eax

    ret
```

Optimized With lea Instruction

x86 Convention: The return value of a function is expected in %eax / %rax

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	FF4E ABC0	0000 0000 0000 abc0
addl 0x7000,%eax	26CE 071B	FF4E ABC0	0000 0000 26CE B2DB
subb %eax,0x7007	26CE 071B	244E ABC0	0000 0000 26CE B2DB

Compiler Example 1

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

Original Code

```
f1:

    movl    (%esi), %eax
    addl    (%edi,%edx,4), %eax



    ret
```

Compiler Output

x86 Convention: The return value of a function is expected in %eax / %rax

Compiler Output 2

```
struct Data {
  char c;
  int d;
};

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

Original Code

```
f1:

    addb    $1, (%edi)
    subl    %esi, 4(%edi)


    ret
```

Compiler Output

x86 Convention: The return value of a function is expected in %eax / %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
  - 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 (%ecx), %eax`
	`addl 4(%ecx), %eax`
	`movl %eax, 8(%ecx)`
		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(%ecx)`
		Assume array 'a' starts @ 0x1000000C

Translating HLL to Assembly

C operator	Assembly	Notes

int dat[4],x;
…
x = dat[0];
x += dat[1];
	`movl $0x10000010,%ecx`
	`movl (%ecx), %eax`
	`movl %eax, 16(%ecx)`
	`movl 16(%ecx), %eax`
	`addl 4(%ecx), %eax`
	`movl %eax, 16(%ecx)`
		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 (%ecx), %eax`
	`shrl 2, %eax`
	`movl %eax, (%ecx)`
	`movw 4(%ecx), %ax`
	`salw 3, %ax`
	`movw %ax, 4(%ecx)`
		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 + S2/D)
  - 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 = ACC + SRC1)

General Instruction Format Issues

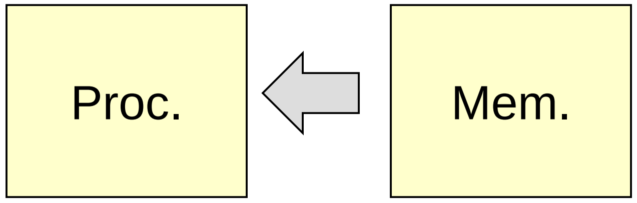
- Consider the pros and cons of each format when performing the set of operations
  - F = X + Y – Z
  - G = A + B
- Simple embedded computers often use single operand format
  - Smaller data size (8-bit or 16-bit machines) means limited instruc. size
- Modern, high performance processors use 2- and 3-operand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

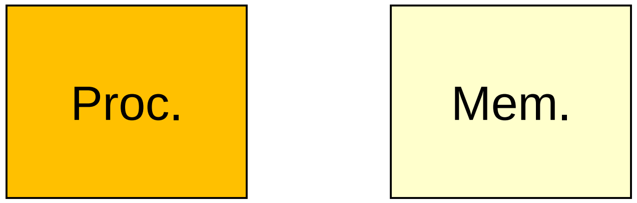
Instruction Format

- 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)

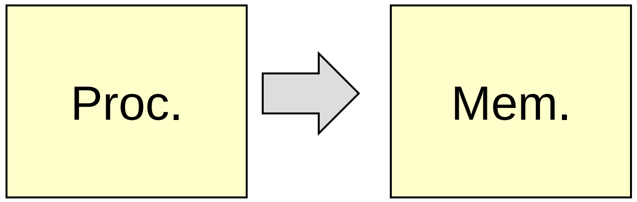
Load/Store Architecture



1.) Load operands to proc. registers



2.) Proc. Performs operation using register values



3.) Store results back to memory