211: Computer Architecture
Spring 2017

Instructor: Prof. Santosh Nagarakatte

Topics:

- C programming conclusion
- Data Representation
  - Reading: Chapter 2.1, 2.2, 2.3, 2.4, 2.5

Signed/Unsigned in C

- Unsigned values in C
  - Declare unsigned int i = 10;
  - Use typecasts i = (unsigned) j;

- -1 when interpreted as a unsigned value is a huge number is
  - $2^n - 1$

Sign extension

Signed integer	4-bit representation	8-bit representation	16-bit representation
+1	0001	00000001	0000000000000001
-1	1111	11111111	1111111111111111

Floating point

Integers typically written in ordinary decimal form

- E.g., 1, 10, 100, 1000, 10000, 12456897, etc.

But, can also be written in scientific notation

- E.g., $1x10^4$, $1.2456897x10^7$

What about binary numbers?

- Works the same way
- $0b100 = 0b1x2^2$

Scientific notation gives a natural way for thinking about floating point numbers

- $0.25 = 2.5x10^{-1} = 0b1x2^{-2}$

How to represent in computers? Why not use fixed point numbers?

IEEE floating point standard

Most computers follow IEEE 754 standard

Single precision (32 bits)

Double precision (64 bits)

Extended precision (80 bits)

S	Exponent	Fraction

Numerical Values

Three different cases:

- Normalized values
  - exponent field ≠ 0 and exponent field ≠ 2^k-1 (all 1's)
  - exponent = binary value – Bias
    - Bias = 2^{k-1}-1 (e.g., 127 for float)
  - Value of the number = 1.(mantissa field)
  - Ex: (sign: 0, exp: 1, mantissa: 1) would give 0b1.1x2^{-126}
- Denormalized values
  - exponent field = 0
  - exponent = 1 – Bias (e.g., -126 for float)
  - Value of the number = mantissa field (no leading 1)
  - Ex: (sign: 0, exp: 0, mantissa: 10) would give 0b10x2^{-126}
- Special values: represent +∞, -∞, and NaN

Decimal to IEEE Floating Point

5.625

In binary

101.101 $\rightarrow$ 1.01101 x 2^2

Exponent field has value 2

- add 127 to get 129

Exponent is 10000001

Mantissa is 01101

Sign bit is 0

0 10000001 01101000000000000000000

Floating point in C

32 bits single precision (type float)

- 1 bit for sign, 8 bits for exponent, 23 bits for mantissa
  - Sign bit: 1 = negative numbers, 0 = positive numbers
  - Exponent is power of 2
- Have 2 zero's
- Range is approximately -10^{38} to 10^{38}

64 bits double precision (type double)

- 1 bit for sign, 11 bits for exponent, 52 bits for mantissa
- Majority of new bits for mantissa ➔ higher precision
- Range is -10^{308} to $+10^{308}$

One more example

Convert 12.375 to floating point representation

Binary is 1100.011

1.100011×2^3

Exponent = 127 + 3 = 130 = 0b10000010

Mantissa = 100011

Sign = 0

Floating Point Operations

- No exact representation for a floating point
  - Mantissa is only 23 bits in 32 bit representation
  - Least significant bits may be dropped

- Floating point operations are not associative
  - (3.14 + 1e10) – 1e10 != 3.14 + (1e10 – 1e10). Why?

iClicker Pop Quiz 1

Let's say we have a 6-bit floating point representation with 1-bit for the sign, 3-bits for the exponent and 2-bits for the Mantissa.

What is the bias?

A: 7

B: 3

C: 1

D: 8

iClicker Pop Quiz 2

Let's say we have a 6-bit floating point representation with 1-bit for the sign, 3-bits for the exponent and 2-bits for the Mantissa.

What is the abstract representation for a normalized value?

A: $(-1)^{S} . (1.M). 2^{E-7}$

B: $(-1)^{S} . (1.M). 2^{E-3}$

C: $(1.M)$

D: 2^{E-3}

iClicker Pop Quiz 3

Let's say we have a 6-bit floating point representation with 1-bit for the sign, 3-bits for the exponent and 2-bits for the Mantissa.

Which of this is a normalized floating point value?

A: 1 001 11

B: 0 111 00

C: 1 110 11

D: 0 000 11

iClicker Pop Quiz 4

Let’s say we have a 6-bit floating point representation with 1-bit for the sign, 3-bits for the exponent and 2-bits for the Mantissa.

Which of this is a denormalized floating point value?

A: 0 001 11

B: 1 000 11

C: 0 100 00

D: 1 111 11

iClicker Pop Quiz 5

Let’s say we have a 6-bit floating point representation with 1-bit for the sign, 3-bits for the exponent and 2-bits for the Mantissa.

Which of this is a representation of infinity?

A: 1 101 11

B: 0 000 11

C: 0 111 11

D: 0 111 00

Hardware Software Interface

Programming Meets Hardware

High-Level Language Program

```c
#include <stdio.h>
int main() {
    int x, y, temp;
    x=1; y=2;
    temp =x; x=y;  y=temp;
    printf("%d %d %d\n",x,y,temp);
}
```

Compiler

Assembly Language Program

```
movl    $1, -8(%ebp)
movl    $2, -12(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -16(%ebp)
movl    -12(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -16(%ebp), %eax
movl    %eax, -12(%ebp)
movl    -16(%ebp), %eax
movl    %eax, 12(%esp)
movl    -12(%ebp), %eax
movl    %eax, 8(%esp)
movl    -8(%ebp), %eax
movl    %eax, 4(%esp)
```

ISA

Assembler

Machine Language Program

```
7f 45 4c 46 01 01 01
 00 00 00 00 00 00 00
 00 00 02 00 03 00 01
00 00 00 f0 82 04 08
34 00 00 00 c4 0c 00
00 00 00 00 00 34 00
```

How do you get performance?

Performance with Programs

(1) Program: Data structures + algorithms

(2) Compiler translates code

(3) Instruction set architecture

(4) Hardware Implementation

Instruction Set Architecture

(1) Set of instructions that the CPU can execute
   (1) What instructions are available?
   (2) How the instructions are encoded? Eventually everything is binary.

(2) State of the system (Registers + memory state + program counter)
   (1) What instruction is going to execute next
   (2) How many registers? Width of each register?
   (3) How do we specify memory addresses?
      - Addressing modes

(3) Effect of instruction on the state of the system

IA32 (X86 ISA)

There are many different assembly languages because they are processor-specific

- IA32 (x86)
  - x86-64 for new 64-bit processors
  - IA-64 radically different for Itanium processors
  - Backward compatibility: instructions added with time
- PowerPC
- MIPS

We will focus on IA32/x86-64 because you can generate and run on iLab machines (as well as your own PC/laptop)

- IA32 is also dominant in the market although smart phone, eBook readers, etc. are changing this

X86 Evolution

8086 – 1978 – 29K transistors – 5-10MHz

I386 – 1985 – 275K transistors – 16-33 MHz

Pentium4 – 2005 – 230M transistors – 2800-3800 MHz

Haswell – 2013 – > 2B transistors – 3200-3900 MHz

Added features

- Large caches
- Multiple cores
- Support for data parallelism (SIMD) eg AVX extensions

CISC vs RISC

CISC: complex instructions : eg X86

- Instructions such as strcpy/AES and others
- Reduces code size
- Hardware implementation complex?

RISC: simple instructions: eg Alpha

- Instructions are simple add/ld/st
- Increases code size
- Hardware implementation simple?

Aside About Implementation of x86

About 30 years ago, the instruction set actually reflected the processor hardware

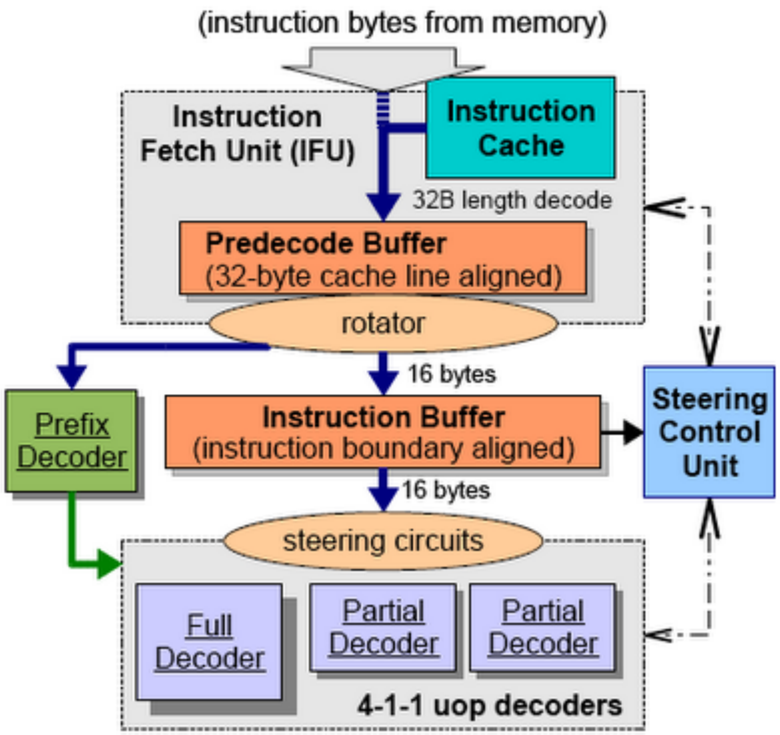
- E.g., the set of registers in the instruction set is actually what was present in the processor

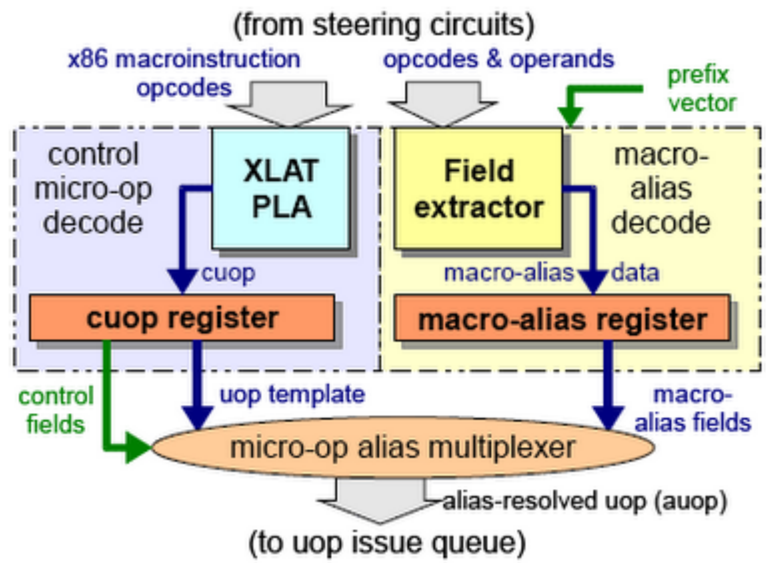
As hardware advanced, industry faced with choice

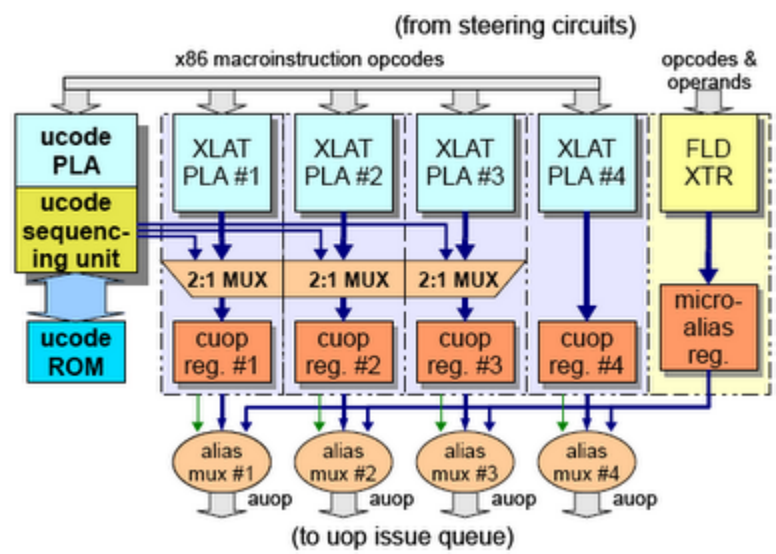
- Change the instruction set: bad for backward compatibility
- Keep the instruction set: harder to exploit hardware advances
  - Example: many more registers but only small set introduced circa 1980

Starting with the P6 (PentiumPro), IA32 actually got implemented by Intel using an “interpreter” that translates IA32 instructions into a simpler “micro” instruction set

P6 Decoder/Interpreter







Assembly Programming

Brief tour through assembly language programming

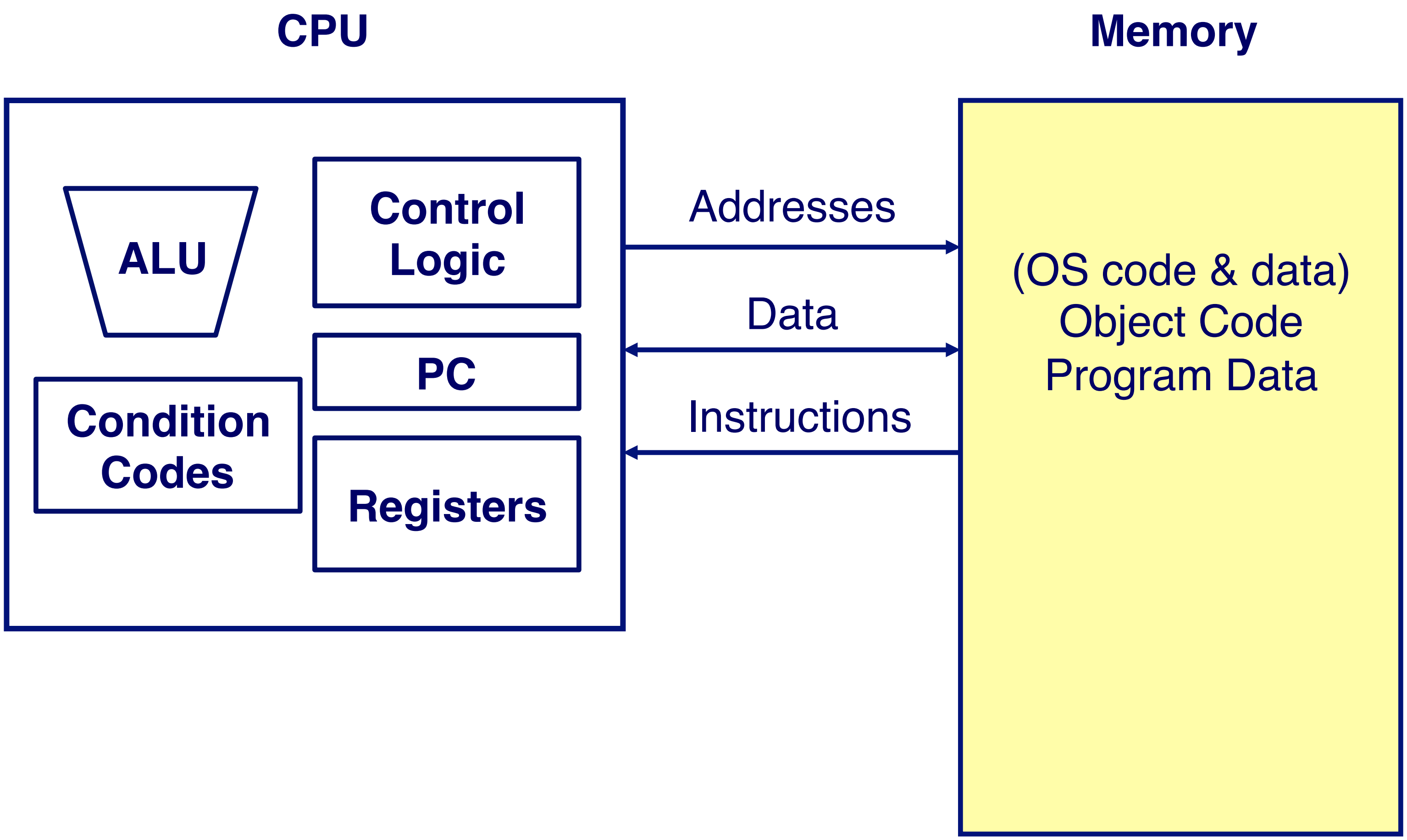
Why?

- Machine interface: where software meets hardware
- To understand how the hardware works, we have to understand the interface that it exports

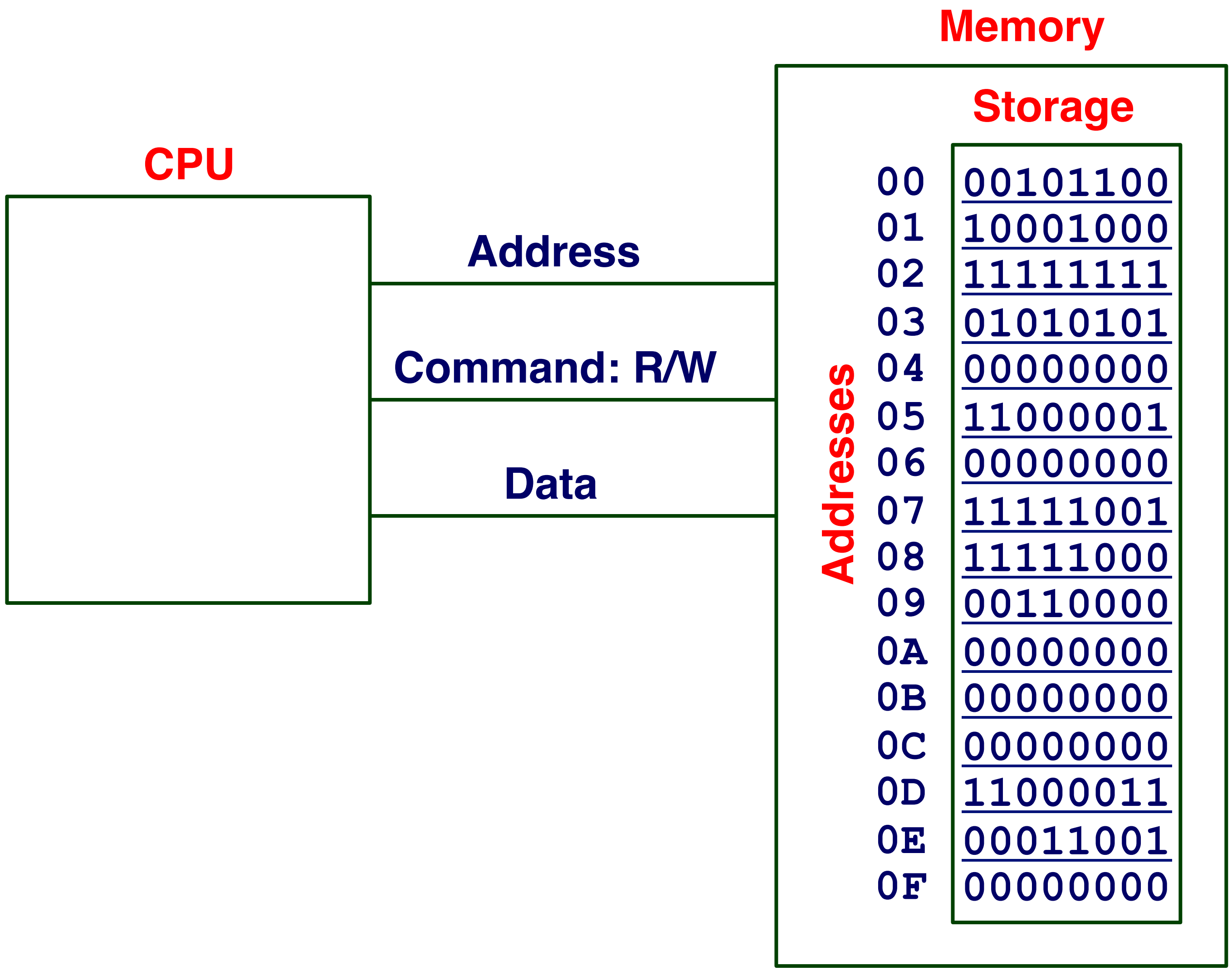
Why not binary language?

- Much easier for humans to read and reason about
- Major differences:
  - Human readable language instead of binary sequences
  - Relative instead of absolute addresses

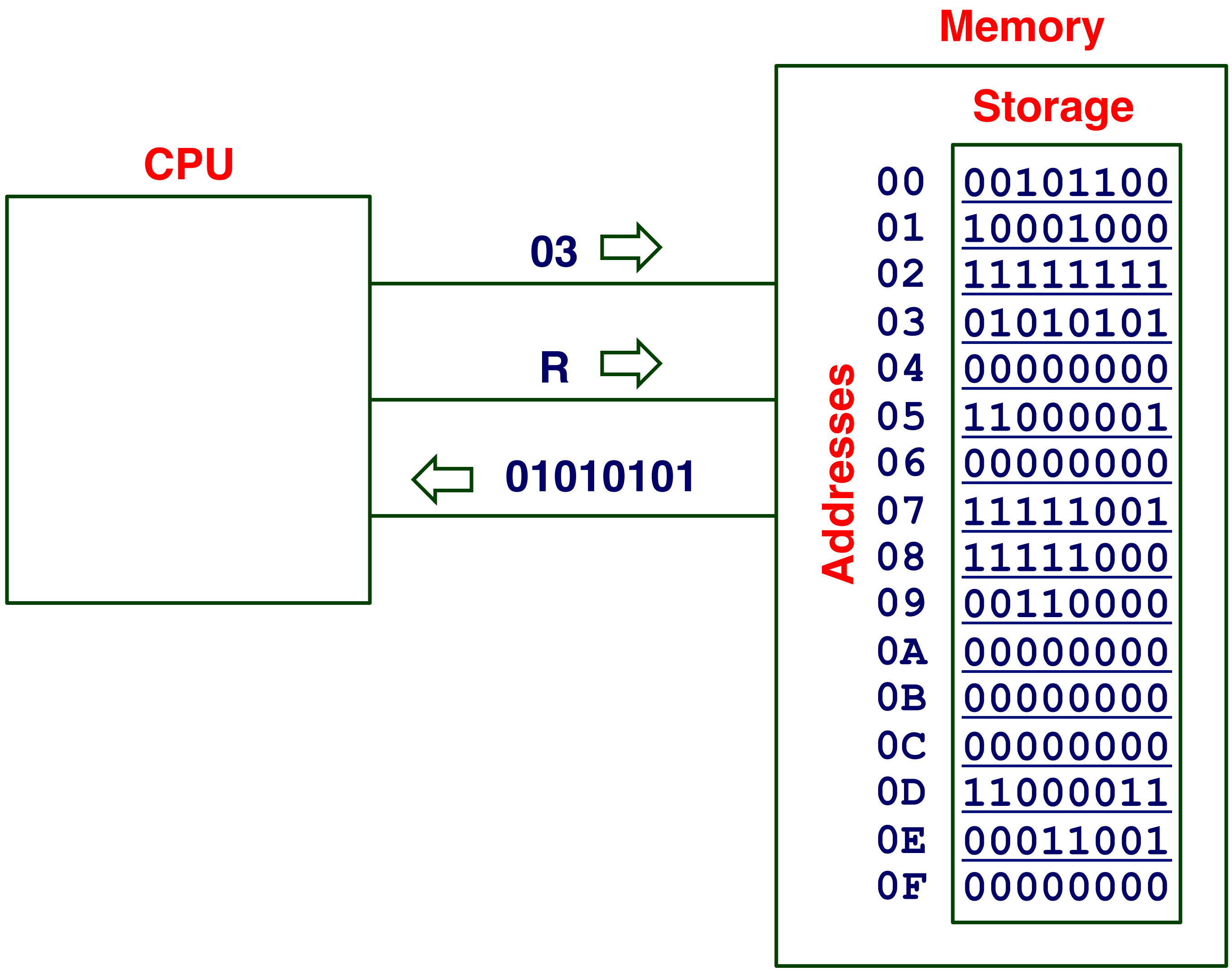
Assembly Programmer's View



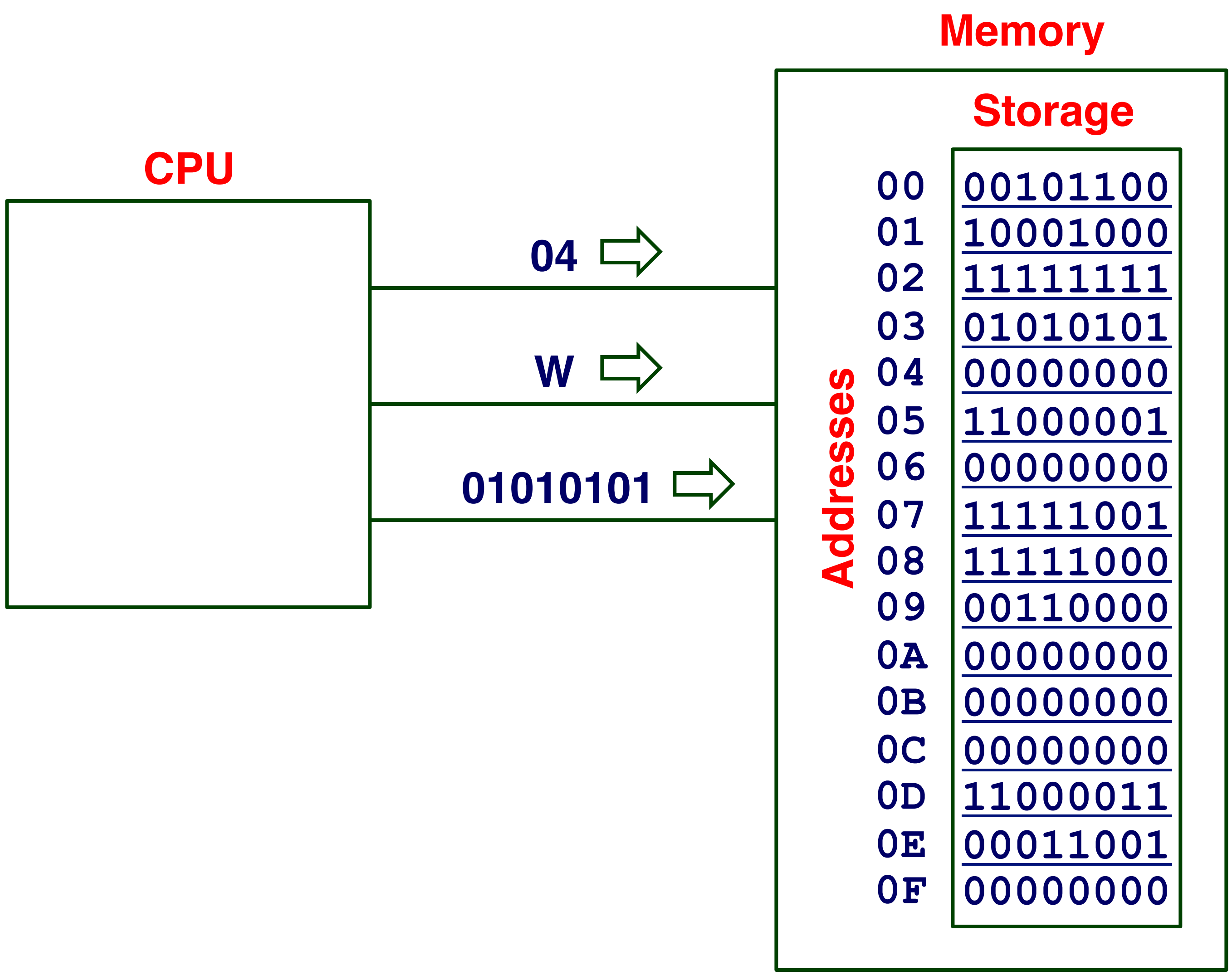
Memory


Memory
Storage
CPU
Address
Command: R/W
Data
Addresses
00 00101100
01 10001000
02 11111111
03 01010101
04 00000000
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000

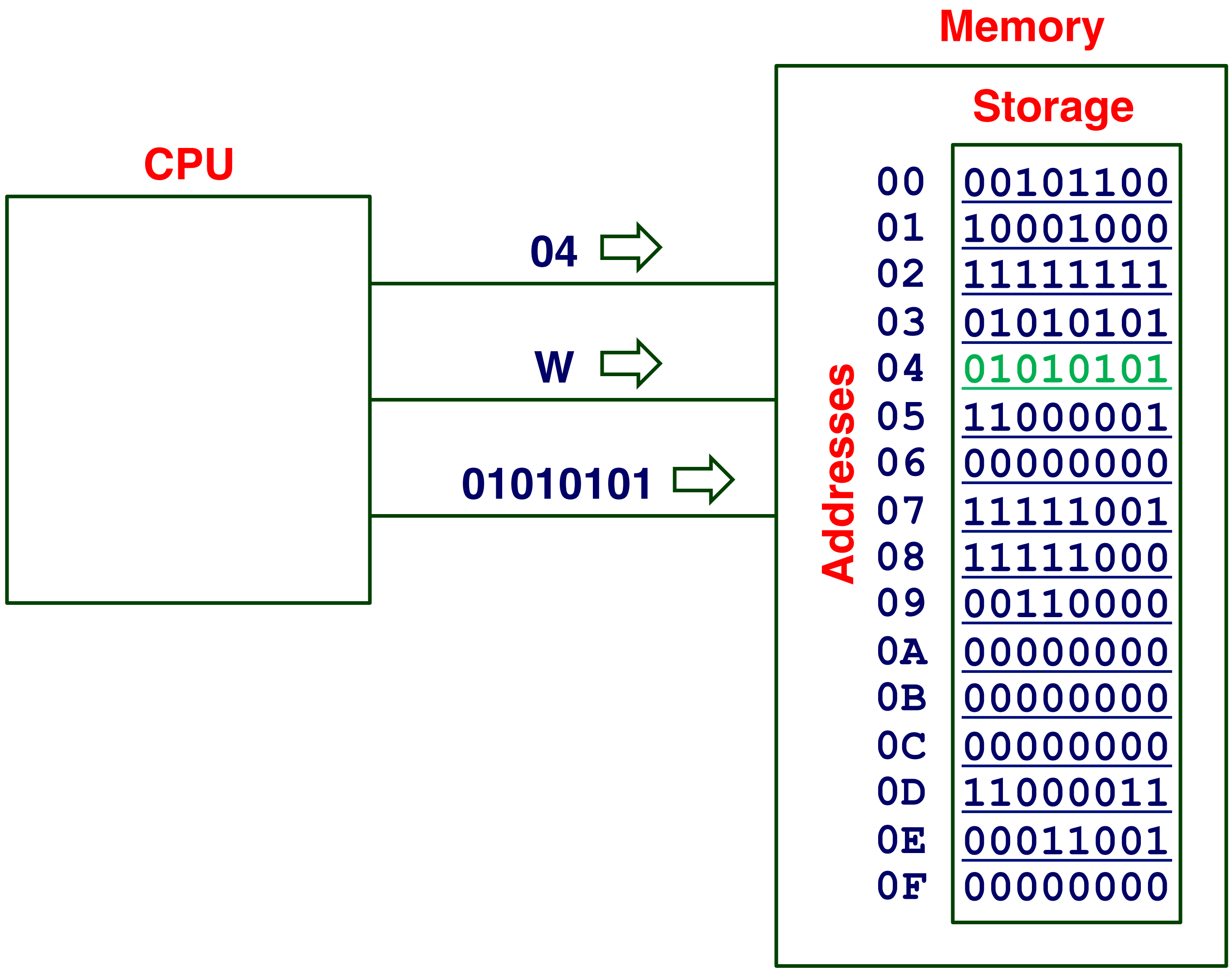
Memory Access: Read



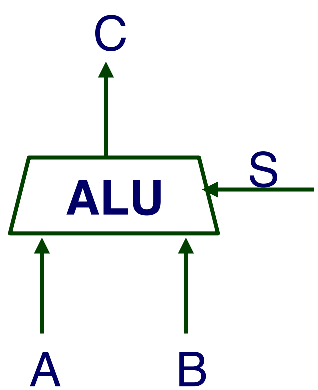
Memory Access: Write



Memory Access: Write

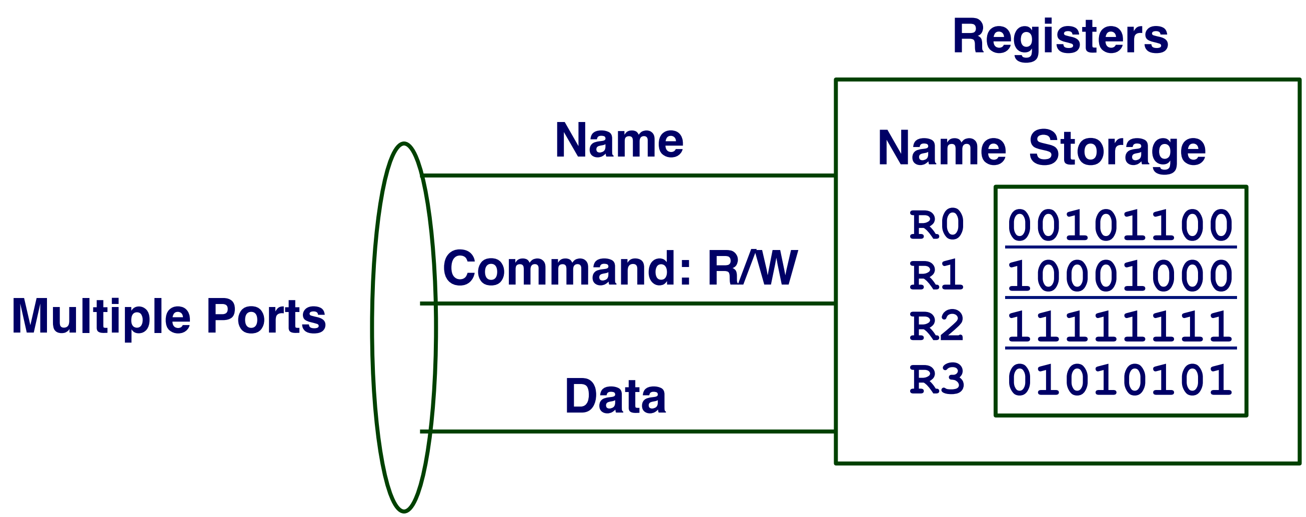


Processor: ALU & Registers

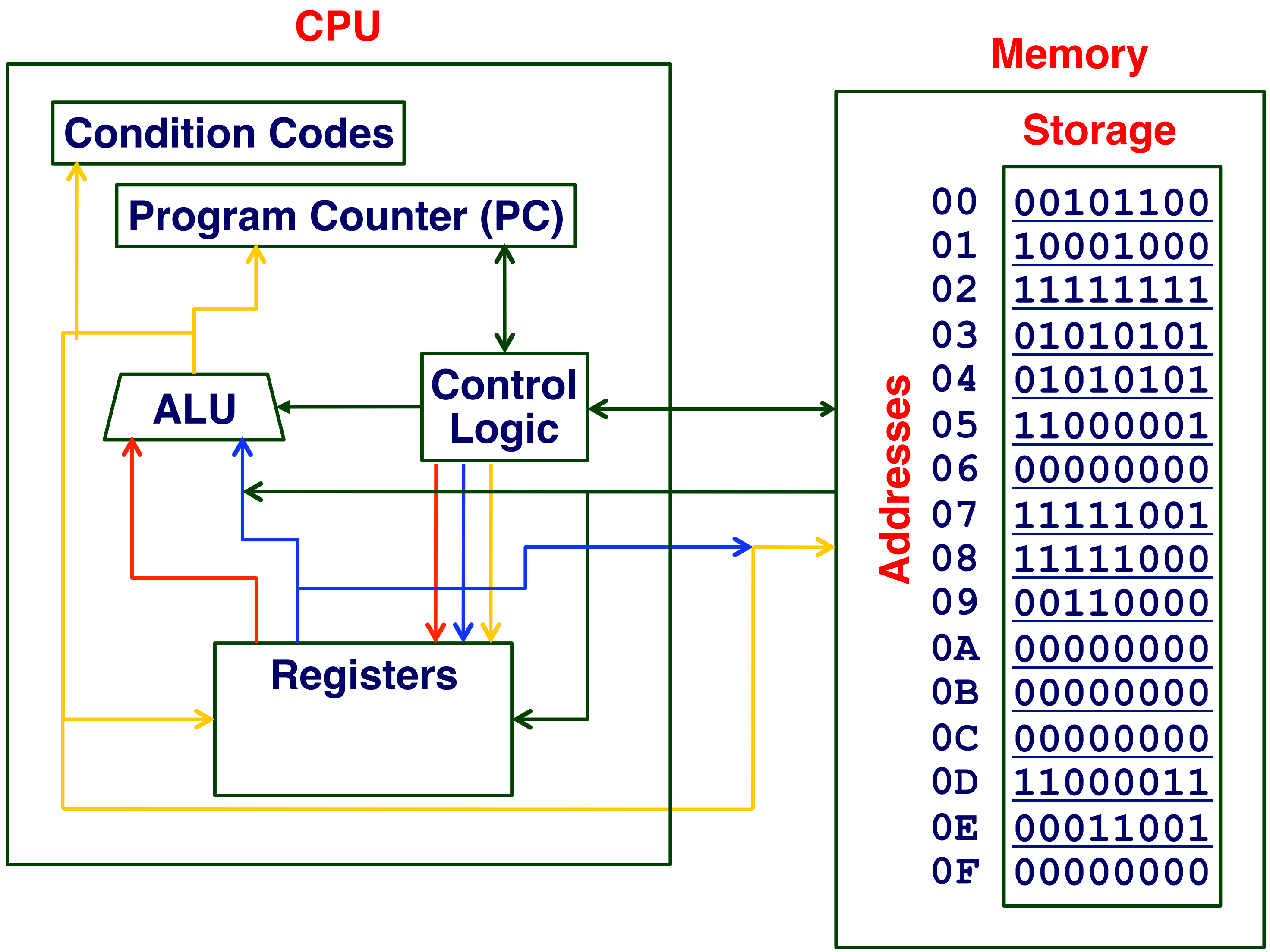


$C = F_S(A, B)$

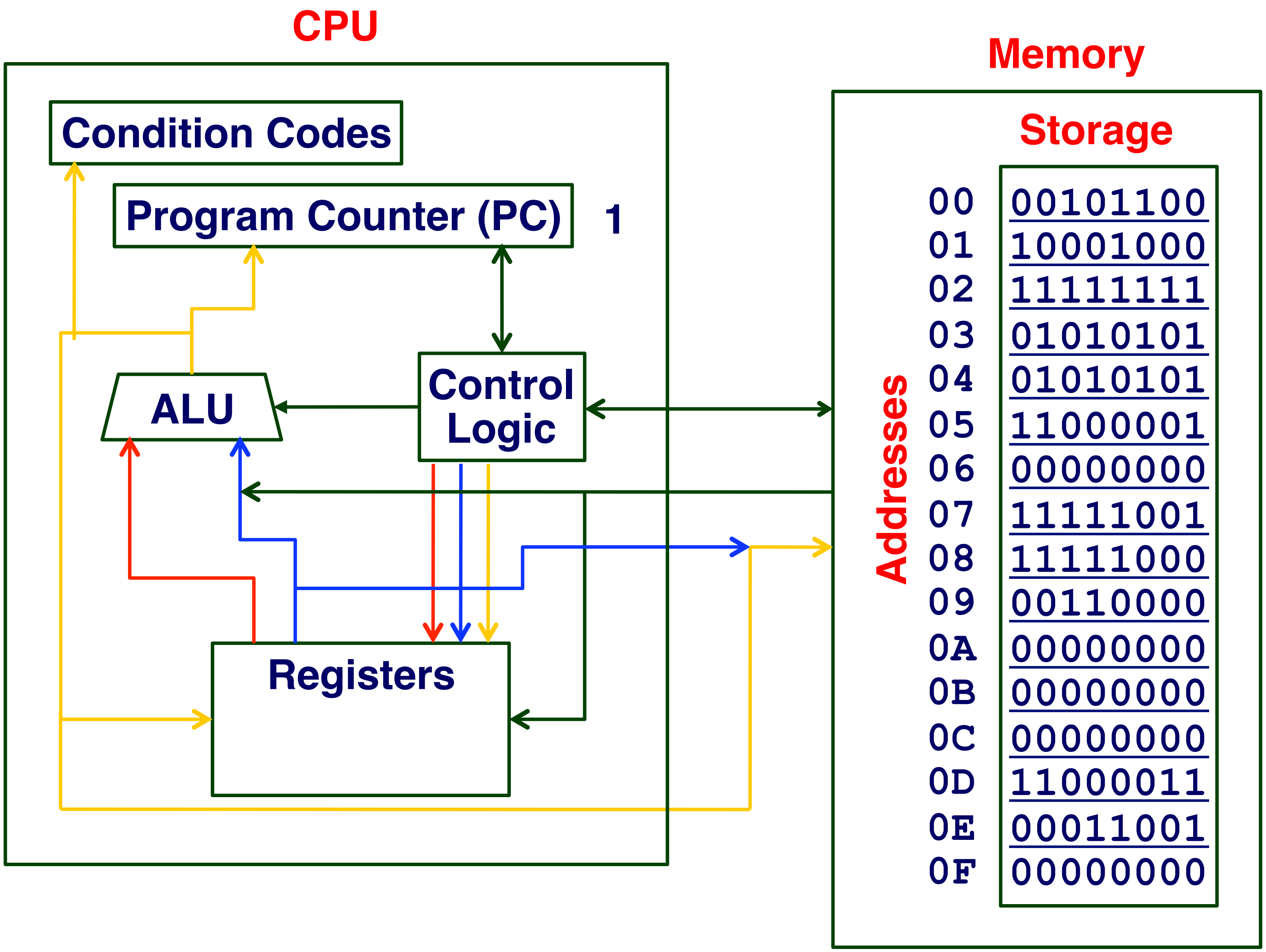
F includes
Arithmetic: +, -, *, /, ~, etc.
Logical: <, >, =, etc.



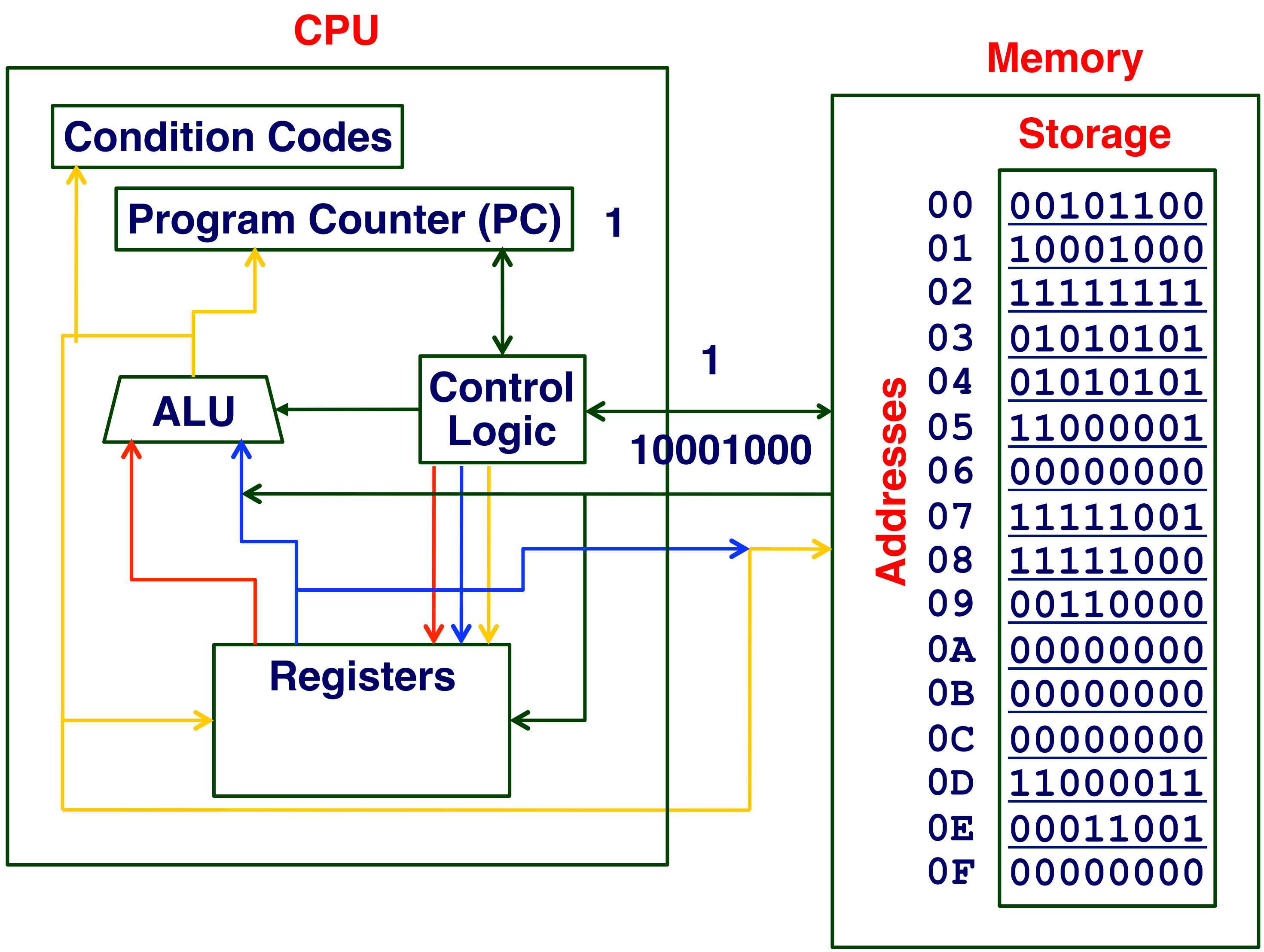
Putting It All Together



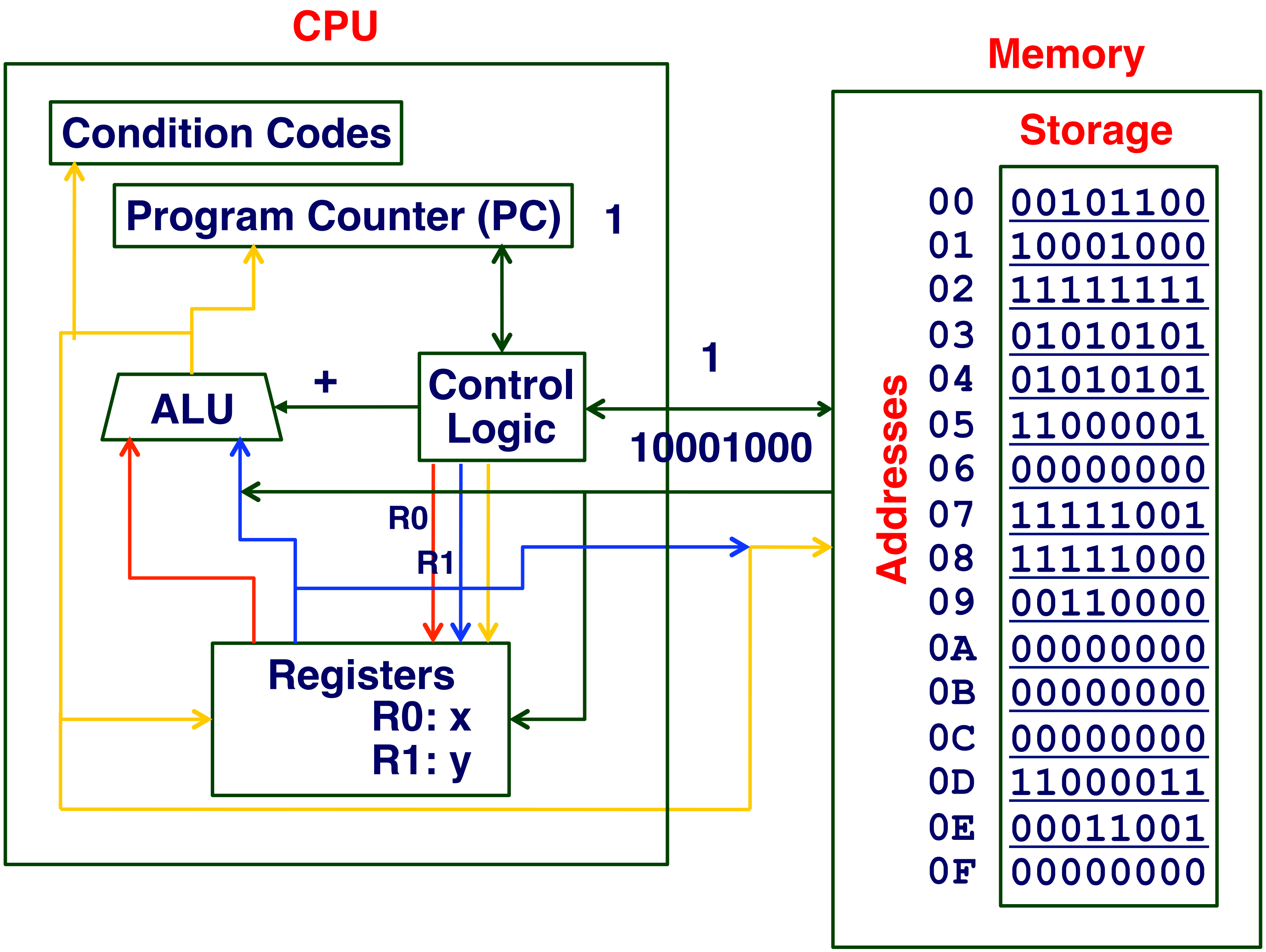
Putting It All Together



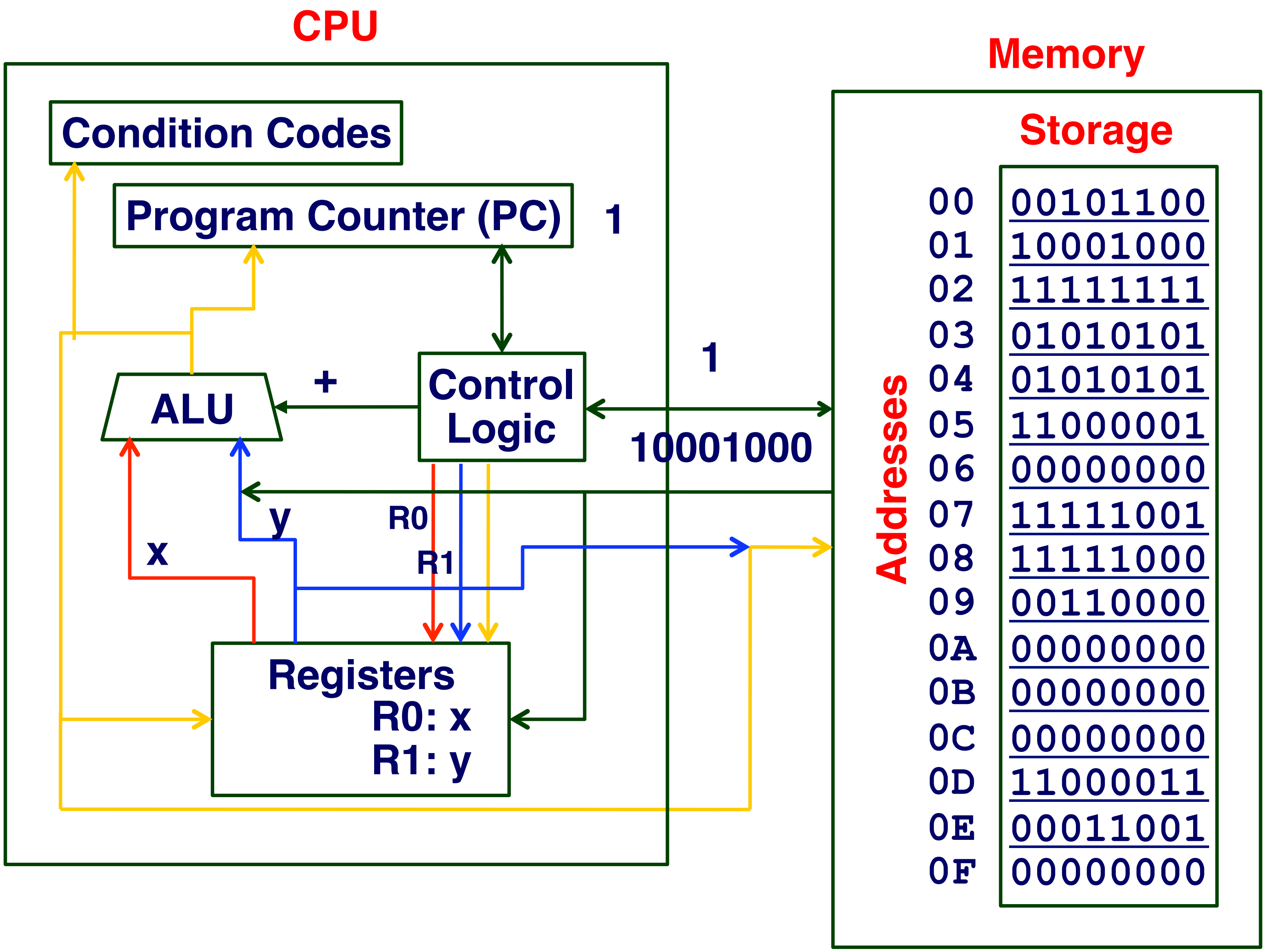
Putting It All Together



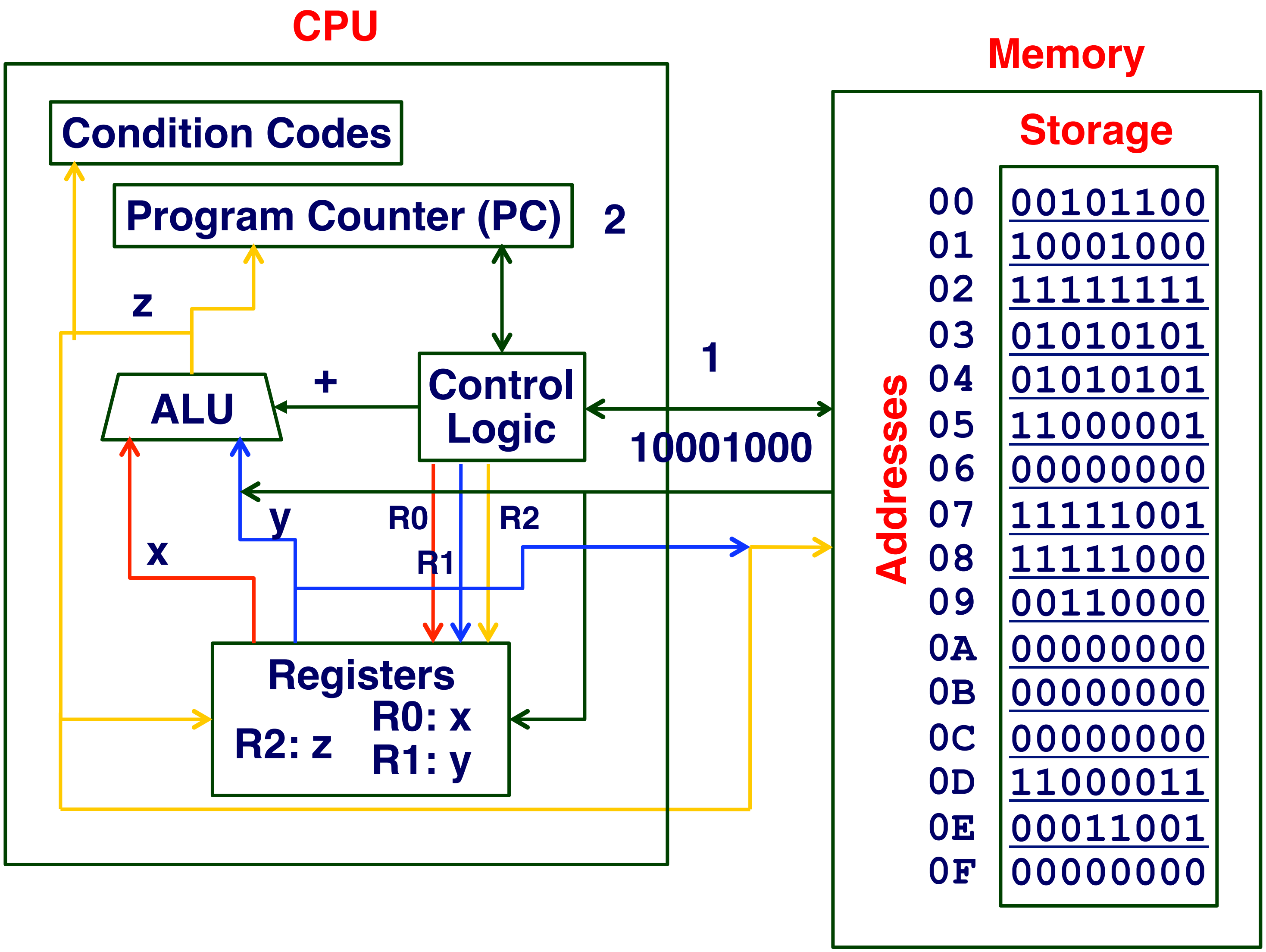
Putting It All Together



Putting It All Together



Putting It All Together



C & Assembly Code

Sample C Code

```c
int accum;
int sum(int x, int y)
{
  int t = x + y;
  accum += t;
  return t;
}
```

gcc -O1 -m32 –S code.c

Generated Assembly

```
sum:
        push %ebp
        movl %esp, %ebp
        movl 12(%ebp), %eax
        addl 8(%ebp), %eax
        addl %eax, accum
        popl %ebp
        ret
```

C & Machine Code

Sample C Code

```
int accum;
int sum(int x, int y){
  int t = x + y;
  accum += t;
  return t;
}
```

gcc -O1 -m32 –c code.c

objdump –d code.o

```
0000000 <sum>:
 0:   55                  push  %ebp
 1:   89 e5               mov   %esp,%ebp
 3:   8b 45 0c            mov   0xc(%ebp),%eax
 6:   03 45 08            add   0x8(%ebp),%eax
 9:   01 05 00 00 00 00   add   %eax, accum
 f:   5d                  pop   %ebp
10:   c3                  ret
```

gdb code.o

```
(gdb) x/100xb sum
```

```
<sum>: 0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45
0x8 <sum+8>: 0x08 0x01 0x05 0x00 0x00  0x00  0x00  0x5d
0x10 <sum+16>: 0xc3 Cannot access memory at address 0x11
```

Assembly Characteristics

Sequence of simple instructions

Minimal Data Types

- “Integer” data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

No type checking

- Interpretation of data format depends on instruction
- No protection against misinterpretation of data

Assembly Characteristics

3 types of Primitive Operations

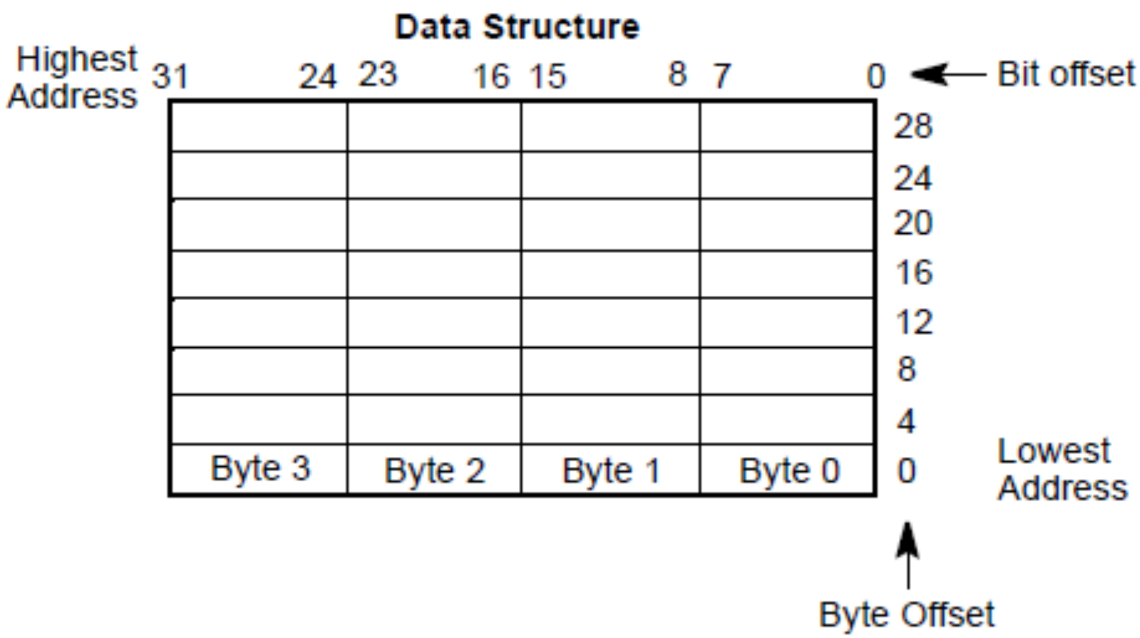
- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

x86 Characteristics

Variable length instructions: 1-15 bytes

Can address memory directly in most instructions

Uses Little-Endian format (Least significant byte in the lowest address)



Instruction Format

General format:

```
opcode operands
```

Opcode:

- Short mnemonic for instruction's purpose
  - `movb,addl, etc.`

Operands:

- Immediate, register, or memory
- Number of operands command-dependent

Example:

- movl %ebx, (%ecx)

Machine Representation

Remember, each assembly instruction translated to a sequence of 1-15 bytes

Opcode	addressing mode	other bytes

First, the binary representation of the opcode

Second, instruction specifies the addressing mode

- The type of operands (registers or register and memory)
- How to interpret the operands

Some instructions can be single-byte because operands and addressing mode are implicitly specified by the instruction

- E.g., pushl

x86 Registers

General purpose registers are 32 bit

- Although operations can access 8-bits or 16-bits portions

Originally categorized into two groups with different functionality

- Data registers (EAX, EBX, ECX, EDX)
  - Holds operands
- Pointer and Index registers (EBP, ESP, EIP,ESI,EDI)
  - Holds references to addresses as well as indexes

Now, the registers are mostly interchangeable

Segment registers

- Holds starting address of program segments
  - CS, DS, SS, ES

x86 Registers

16 BITS

8 BITS

EAX	AX	AH	AL
ECX	CX	CH	CL
EDX	DX	DH	DL
EBX	BX	BH	BL
ESP --Stack Pointer			
EBP --- Base register of current stack frame			
ESI --- Source index for string operations			
EDI --- Destination index for string operations			

32 BITS

x86 Programming

- Mov instructions to move data from/to memory
  - Operands and registers
- Addressing modes
- Understanding swap
- Arithmetic operations
- Condition codes
- Conditional and unconditional branches
- Loops and switch statements

Data Format

Byte: 8 bits

- E.g., char

Word: 16 bits (2 bytes)

- E.g., short int

Double Word: 32 bits (4 bytes)

- E.g., int, float

Quad Word: 64 bits (8 bytes)

- E.g., double

Instructions can operate on any data size

- `movl, movw, movb`
  - `Move double word, word, byte, respectively`
- End character specifies what data size to be used

MOV instruction

Most common instruction is data transfer instruction

- Mov SRC, DEST: Move source into destination
- SRC and DEST are operands
- DEST is a register or a location
- SRC can be the contents of register, memory location, constant, or a label.
- If you use gcc, you will see movl <src>, <dest>
- All the instructions in x86 are 32-bit

Used to copy data:

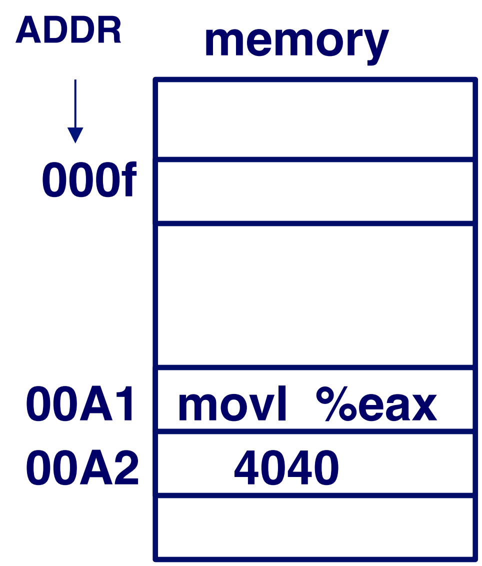
- Constant to register (immediate)
- Memory to register
- Register to memory
- Register to register

Cannot copy memory to memory in a single instruction

Immediate Addressing

Operand is immediate

- Operand value is found immediately following the instruction
- Encoded in 1, 2, or 4 bytes
- $ in front of immediate operand
- E.g., movl $0x4040, %eax



Register Mode Addressing

Use % to denote register

- E.g., %eax

Source operand: use value in specified register

Destination operand: use register as destination for value

Examples:

- movl %eax, %ebx
  - Copy content of %eax to %ebx
- movl $0x4040, %eax → immediate addressing
  - Copy 0x4040 to %eax
- movl %eax, 0x0000f → Absolute addressing
  - Copy content of %eax to memory location 0x0000f

Indirect Mode Addressing

Content of operand is an address

- Designated as parenthesis around operand

Offset can be specified as immediate mode

Examples:

- movl (%ebp), %eax
  - Copy value from memory location whose address is in ebp into eax
- movl -4(%ebp), %eax
  - Copy value from memory location whose address is -4 away from content of ebp into eax

Indexed Mode Addressing

Add content of two registers to get address of operand

- movl (%ebp, %esi), %eax
  - Copy value at (address = ebp + esi) into eax
- movl 8(%ebp, %esi),%eax
  - Copy value at (address = 8 + ebp + esi) into eax

Useful for dealing with arrays

- If you need to walk through the elements of an array
- Use one register to hold base address, one to hold index
  - E.g., implement C array access in a for loop
- Index cannot be ESP

Scaled Indexed Mode Addressing

Multiply the second operand by the scale (1, 2, 4 or 8)

- movl 0x80 (%ebx, %esi, 4), %eax
  - Copy value at (address = ebx + esi*4 + 0x80) into eax

Where is it useful?

Address Computation Examples

%edx	0xf000
%ecx	0x100

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

`movl` Operand Combinations

	Source	Destination		C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

- Cannot do memory-memory transfers with single instruction

Stack Operations

By convention, %esp is used to maintain a stack in memory

- Used to support C function calls

%esp contains the address of top of stack

Instructions to push (pop) content onto (off of) the stack

- pushl %eax
  - esp = esp – 4
  - Memory[esp] = eax
- popl %ebx
  - ebx = Memory[esp]
  - esp = esp + 4

Where does the stack start? We'll discuss later