HARVARD
School of Engineering
and Applied Sciences

Representation of Information

CS61, Lecture 2

Prof. Stephen Chong

September 6, 2011

Announcements

- Assignment 1 released
  - Posted on http://cs61.seas.harvard.edu/
  - Due one week from today, **Tuesday 13 Sept**
  - First question (survey) due **5:00pm Thursday 8 Sept**
  - Contains C self assessment
    - If you are not comfortable with all of the questions, may need to spend time getting up to speed with C
- Name tags
  - At back of room
  - Fill in, put in front of you, leave at end of class
- Sections will start next week
  - Section times and signup will be later this week
- Highscore binary

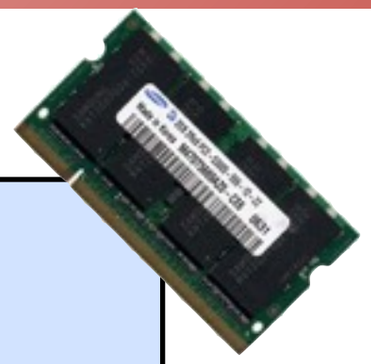
Topics for today

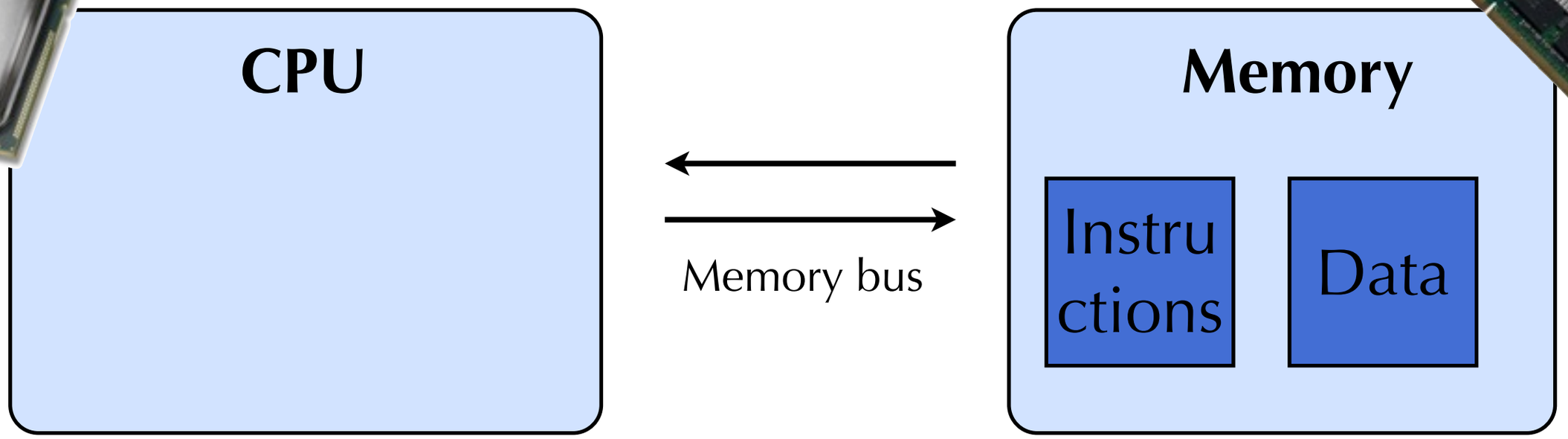
- Representing information
  - Hexadecimal notation
  - Representing integers
  - Storing information
    - Word size, data size
    - Byte ordering
  - Representing strings
  - Representing code
- Basic processor operation

Information

- Computers represent information using **bits**
  - 2-valued signals; binary digits
  - All kinds of information
    - Numbers, memory addresses, instructions, strings, …
- What do the bits `1100 0011` represent?
  - Could be (unsigned) integer 195
  - Could be (signed) integer -61
  - Could be instruction `ret`
  - Depends on context!
- Information is bits plus **context**
  - context = way of interpreting data

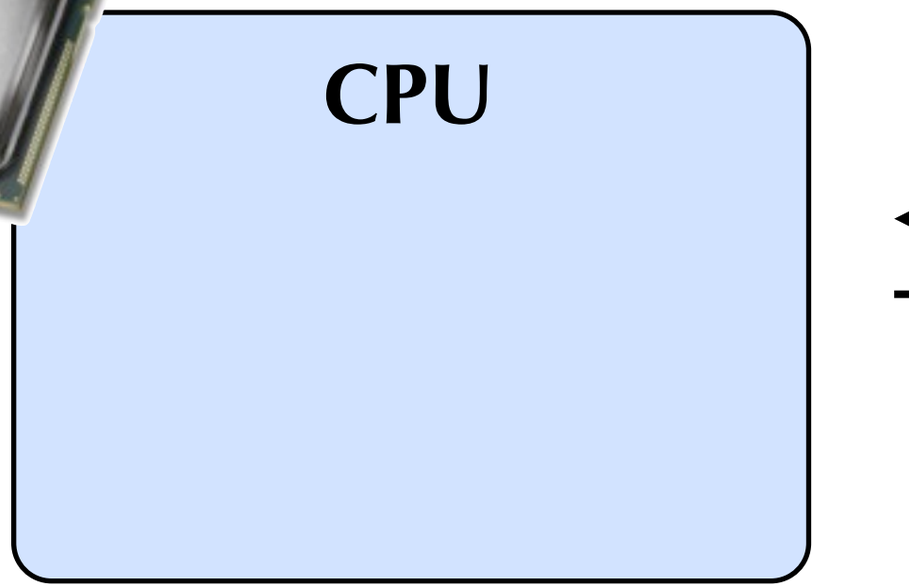
Computers

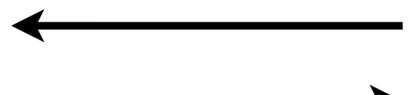


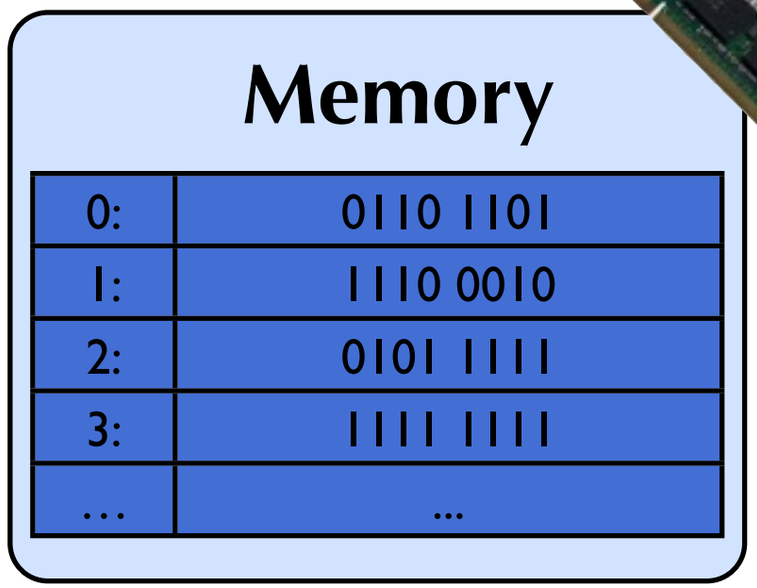


- Computers store bits in memory
- Information stored in memory is both instructions and data
  - But remember, instructions and data are just bits that get interpreted differently!

Computers



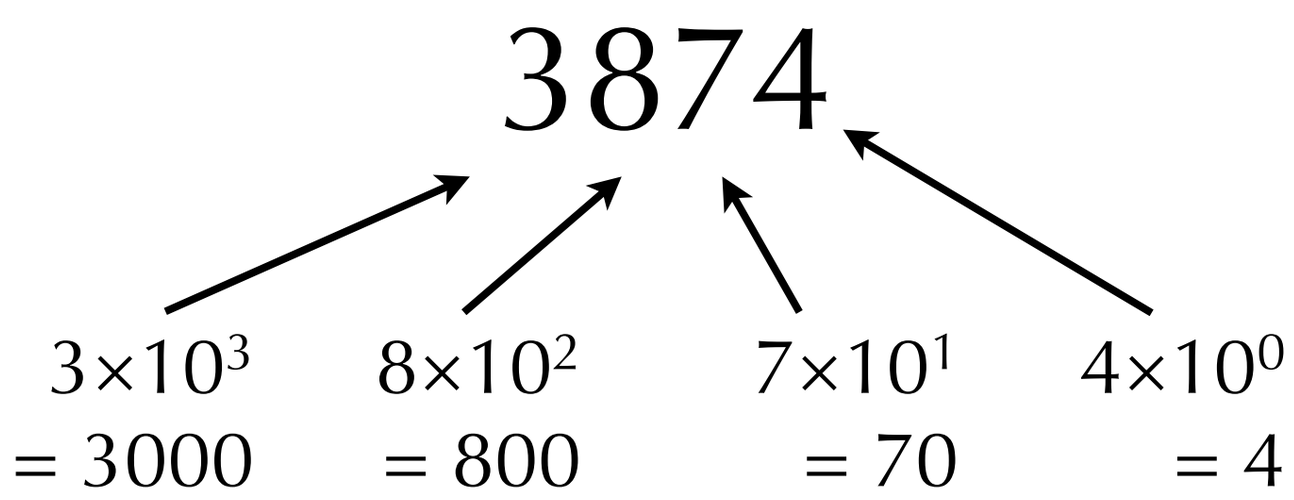




- Rather than accessing individual bits, most computers use blocks of 8 bits, called **bytes**
- View memory as a very large array of bytes
  - Memory addresses are another kind of data

Hexadecimal notation

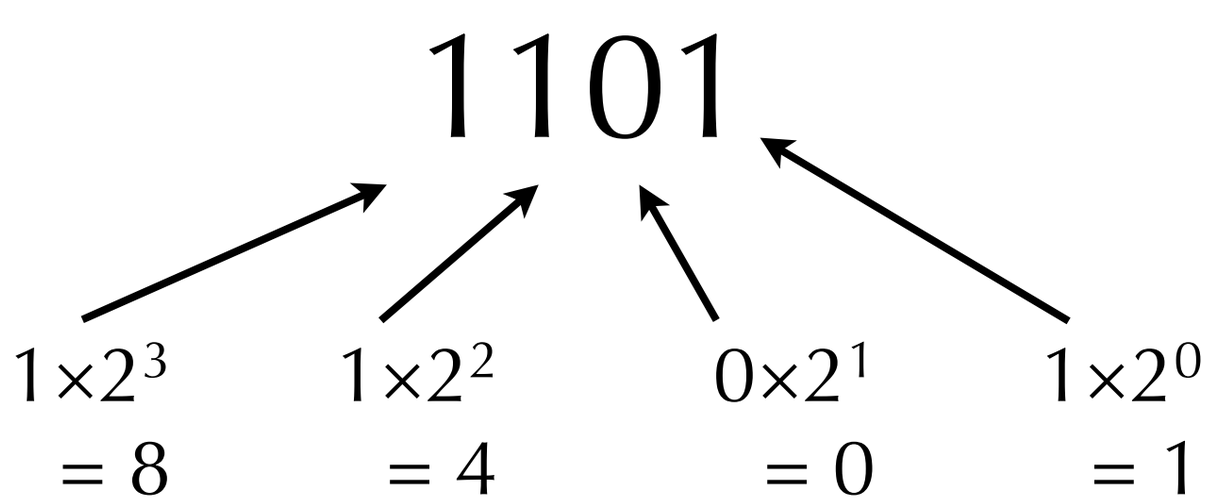
- To make it easier to read bits, we use **hexadecimal notation**
- Decimal notation is base 10
  - Uses digits 0,1,2,3,4,5,6,7,8,9
  - *xyz* represents number $x \times 10^2 + y \times 10^1 + z \times 10^0$
  - E.g.,



$$3000 + 800 + 70 + 4 = 3874$$

Binary notation

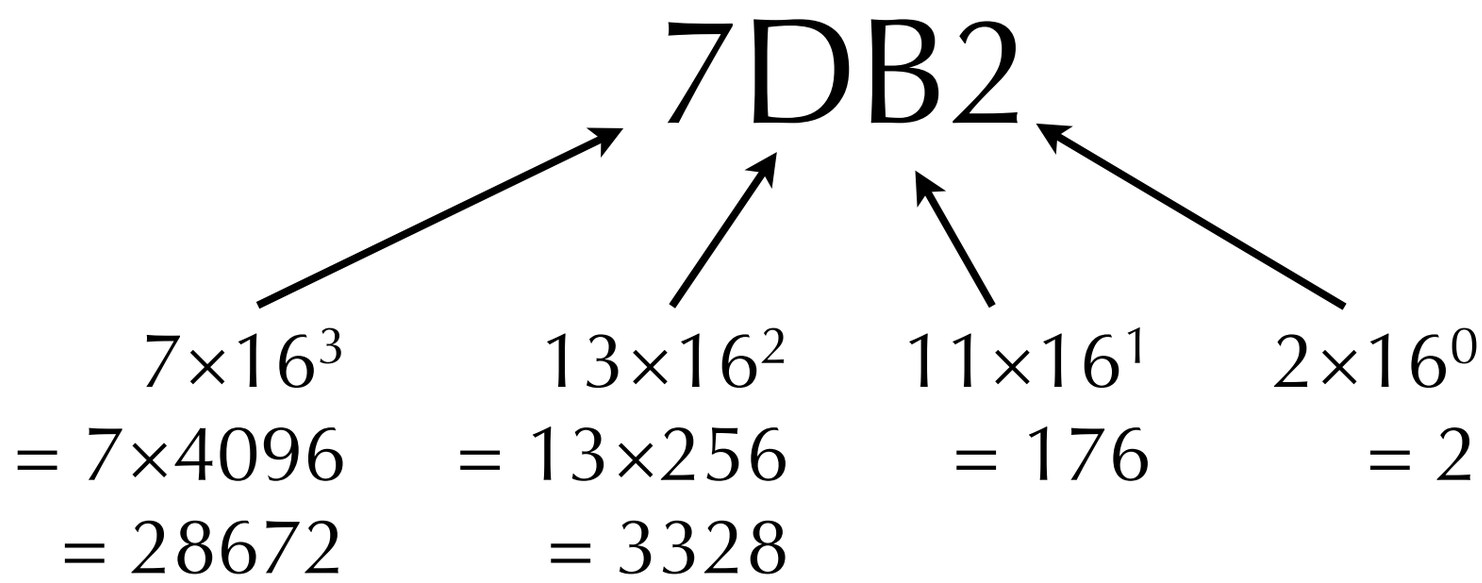
- Binary notation is base 2
  - Uses digits 0,1
  - *xyz* represents number $x\times2^2 + y\times2^1 + z\times2^0$
  - E.g.



$8 + 4 + 0 + 1 = 13$

Hexadecimal notation

- Hexadecimal notation is base 16
  - Use digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - *xyz* represents number $x\times16^2 + y\times16^1 + z\times16^0$
  - E.g.



$$28672 + 3328 + 176 + 2 = 32178$$

Hexadecimal notation

- Why is hexadecimal notation useful for computer programming?
- One hexadecimal digit represents 4 bits
- One byte is two hexadecimal digits
  - E.g., 0x8C = 10001100
- In C (and in this class) we prefix hexadecimal numbers with "0x"
  - E.g., 0x5A = 01011010 = 90 = 5×16 + 10
  - E.g., 0x42 = 01000010 = 66 = 4×16 + 2
- You will get comfortable and familiar with hexadecimal notation.

Binary value	Dec. value	Hex. digit
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Other bases

- Why hexadecimal?
  - Base 16, so corresponds to 4 bits

```
0101 1100
 5    C
```

  - More compact than binary, but 16 possible values small enough to be understandable
- Octal notation also common
  - Corresponds to 3 bits

```
 01 011 100
  1   3   4
```

- C notation
  - Leading 0 (zero) on integer constant means octal
  - Leading 0x on integer constant means hexadecimal
  - E.g., 31 = 037 = 0x1f

Base 64

- An ASCII representation of binary data

Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

```
Message-ID: <4E614710.8010700@seas.harvard.edu>
Date: Fri, 2 Sep 2011 17:13:52 -0400
From: John Harvard <jharvard@college.harvard.edu>
To: Stephen Chong <chong@seas.harvard.edu>
Subject: Homework
Content-Type: multipart/mixed;
       boundary="------------070603010202000801030000"
Return-Path: jharvard@college.harvard.edu
X-Originating-IP: [10.243.39.38]
MIME-Version: 1.0

--------------070603010202000801030000
Content-Type: text/plain; charset="ISO-8859-1"; format=flowed
Content-Transfer-Encoding: 7bit

Hi Prof. Chong,
  Attached is my homework. Sorry it's late.

--------------070603010202000801030000
Content-Type: application/octect-stream; name="hw1.pdf"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="hw1.pdf"

JVBERi0xLjMKJcTl8uXrp/Og0MTGCjQgMCBvYmoKPDwgL0xlbmd0aCA1IDAgUiAvRmlsdGVyIC9G
bGF0ZURlY29kZSA+PgpzdHJlYW0KeAGNlE1vm0AQhu/7K94jSM16vwA7xzqNlKiHRCDlUPVAyTqm
CmBjfGh/fQeWXdtN+iEOs9qdfd9nmIE9HrGHoEcKZZClS/QWT2ixWB8kqgMkDtWbhA0ET1GPScol
SfaHxMWD7Su7G47lK/qazEy6cnqZopisDKoGi7tG4qYjGuJhxAKtBDfGJFhmAipRZ1yT5cgsJraJ
ZE+g48aVHCtBkqnpNiPtjwXV5Q4pamQaBTneSk5Vo9jgC6J1jCvBFaI8ptIoIpXXMb6iuMenwkG9
pz+yk346lkLmFDVkQqq/G3jdH86IRYfZaLCN9/Zb2M1nvT/p/OLFL8JR2cTMoQedup3vh2yUfuvZ
13dR3F9f3kVxSiVviwvOZTXLb+c4o7DIItQQ0ANeoKt/lkMdEon51IBxTpWgcaAGygxylbLTpP5j
IsZ7KlvNzTLGNYsibfJUKw1j2NQy5Wci+myr4UgO8hp37dB3z8eK0Aip+O4mQvBE01yS9DsrRh/N
```

Representing integers

- Given *n* bits to store an integer, we can represent 2^n different values
- If we just care about non-negative (aka **unsigned**) integers, we can easily store the values

  0, 1, 2, …, 2^n-1

  - E.g., for 4 bits
    - `0x2` = 2
    - `0xB` = 11
    - `0xF` = 15 = 2^4-1

Integer overflow

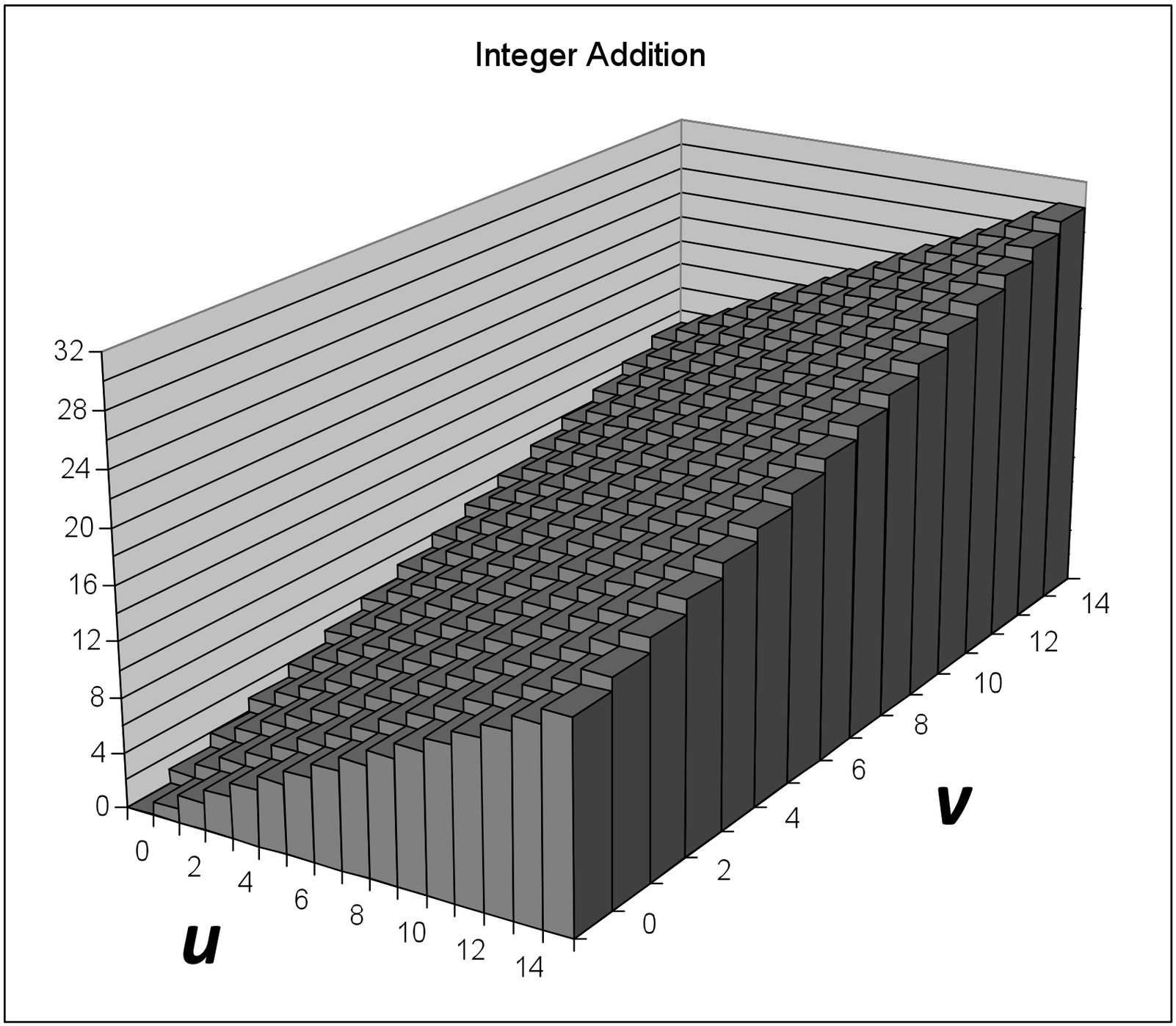
- With *n* bits, we can represent values 0, 1, 2, …, 2^n-1
- **Overflow** occurs when we have a result that doesn't fit in the *n* bits
  - E.g., using 4 bits: `0xF + 0x1`

```
0xF =    1111
0x1 =    0001
        ------
        10000
```

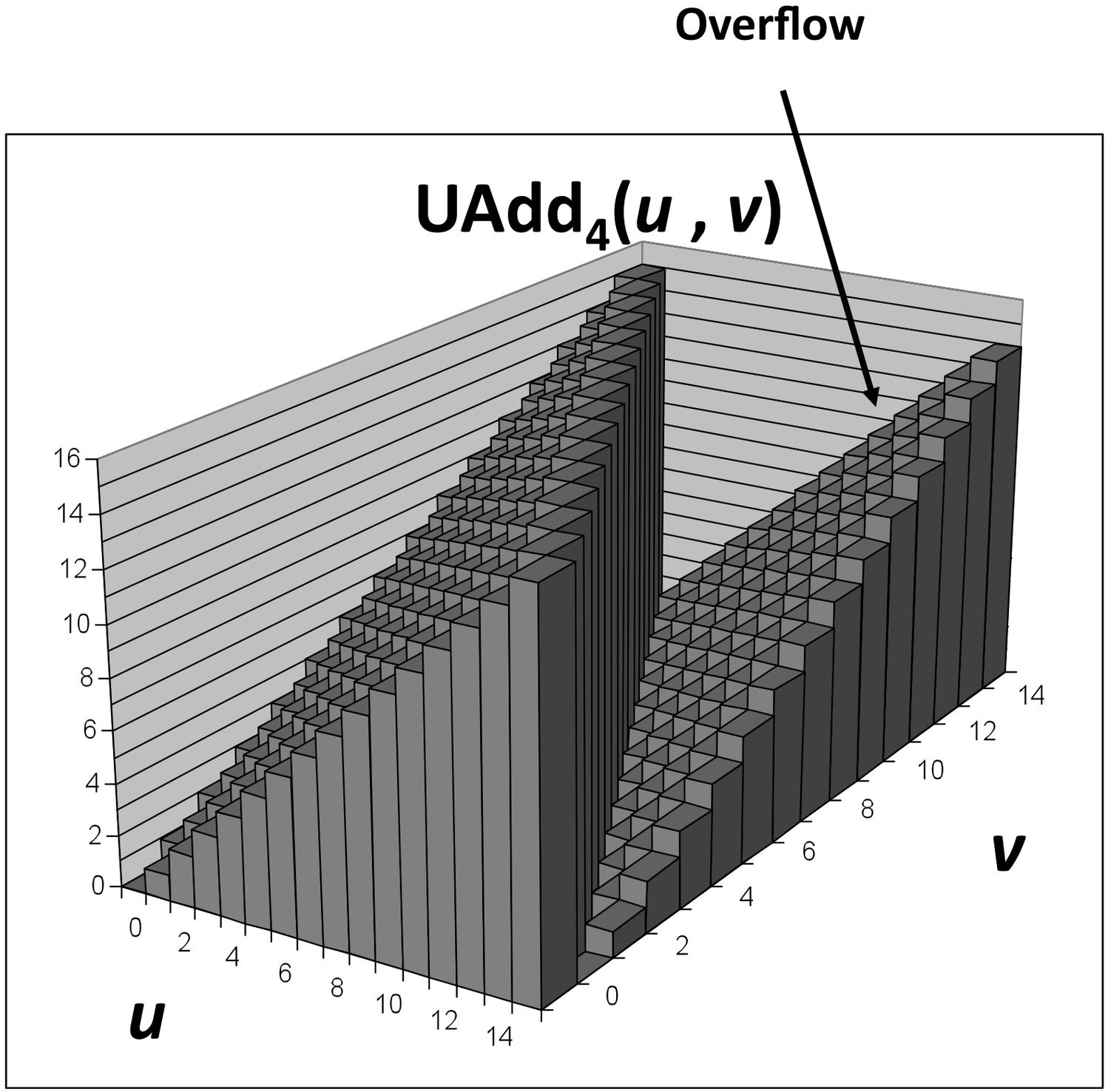
`0xF + 0x1 = 0x0` Overflow!!

Integer overflow

$\mathbf{Add_4}(u, v)$



Integer overflow



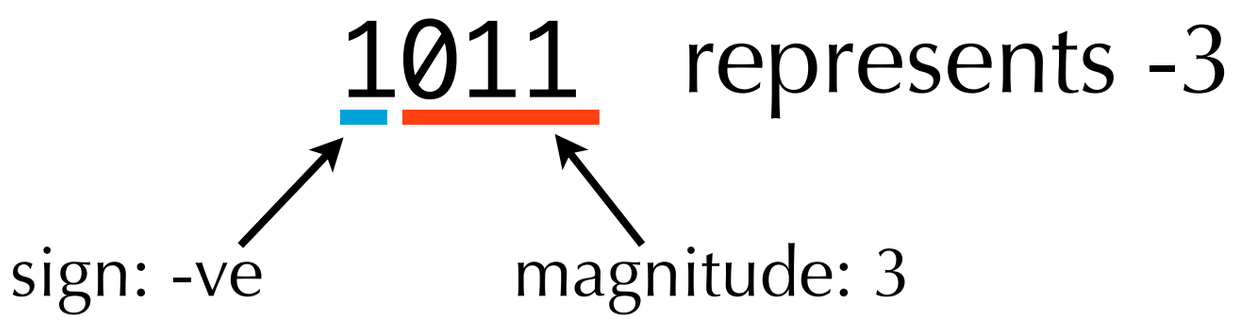
Representing negative integers

- Have seen how to represent **unsigned integers** (i.e., non-negative integers)
- How do we represent negative integers?
- Three common encodings:
  - Sign and magnitude
  - Ones' complement
  - Two's complement

Sign and magnitude

- Use one bit to represent sign, remaining bits represent magnitude
- With *n* bits, have *n*-1 bits for magnitude
  - E.g., with 4 bits, can represent integers

    -7, -6, …, -1, 0, 1, …, 6, 7



Properties of sign and magnitude

- Straight-forward and intuitive
- Two different representations of zero!
  - E.g., using 4 bits, `1000` and `0000` both represent zero!
- Arithmetic operations need different implementation than for unsigned
  - E.g., addition, using 4 bits
    - unsigned: `0001` + `1001` = 1 + 9 = 10 = `1010`
    - sign and magnitude: `0001` + `1001` = 1 + -1 = 0 = `0000`

Ones' complement

- If integer k is represented by bits $b_1 \ldots b_n$, then $-k$ is represented by 11...11 - $b_1 \ldots b_n$ (where |11...11|=n)
  - Equivalent to flipping every bit of b
  - E.g., using n=4 bits:
    - 6 = 0110
    - -6 = 1111 - 0110 = 1001
- Using n bits, can represent numbers 2^n-1 values
  - E.g., using 4 bits, can represent integers

    -7, -6, ..., -1, 0, 1, ..., 6, 7
  - Like sign and magnitude, first bit indicates whether number is negative

Properties of ones' complement

- Same implementation of arithmetic operations as for unsigned
  - E.g., addition, using 4 bits
    - unsigned: `0001` + `1001` = 1 + 9 = 10 = `1010`
    - ones' complement: `0001` + `1001` = 1 + -6 = -5 = `1010`

- Two different representations of zero!
  - E.g., using 4 bits, `1111` and `0000` both represent zero!

Two's complement

- If integer k is represented by bits $b_1 \ldots b_n$, then $-k$ is represented by `100...00` - $b_1 \ldots b_n$ (where |`100...00`|=n+1)
  - Equivalent to taking ones' complement and adding 1
  - E.g., using 4 bits:
    - 6 = `0110`
    - -6 = `10000-0110 = 1010 = (1111-0110)+1`
- Using n bits, can represent numbers 2^n values
  - E.g., using 4 bits, can represent integers

    -8, -7, …, -1, 0, 1, …, 6, 7
  - Like sign and magnitude and ones' complement, first bit indicates whether number is negative

Properties of two's complement

- Same implementation of arithmetic operations as for unsigned
  - E.g., addition, using 4 bits
    - unsigned: `0001` + `1001` = 1 + 9 = 10 = `1010`
    - two's complement: `0001` + `1001` = 1 + -7 = -6 = `1010`
- Only one representation of zero!
  - Simpler to implement operations
- Not symmetric around zero
  - Can represent more negative numbers than positive numbers
- Most common representation of negative integers

Converting to and from two's complement

- To encode a negative number in two's complement in *n* bits:
  - Compute out the binary notation for the absolute value using *n* bits
  - Invert the bits
  - Add 1
  - E.g., to encode -5 using 8 bits
    - 5 = `00000101` using 8 bits
    - Invert the bits: `11111010`
    - Add one: `11111010` + 1 = `11111011`
    - -5 encoded in two's complement using 8 bits is `11111011`

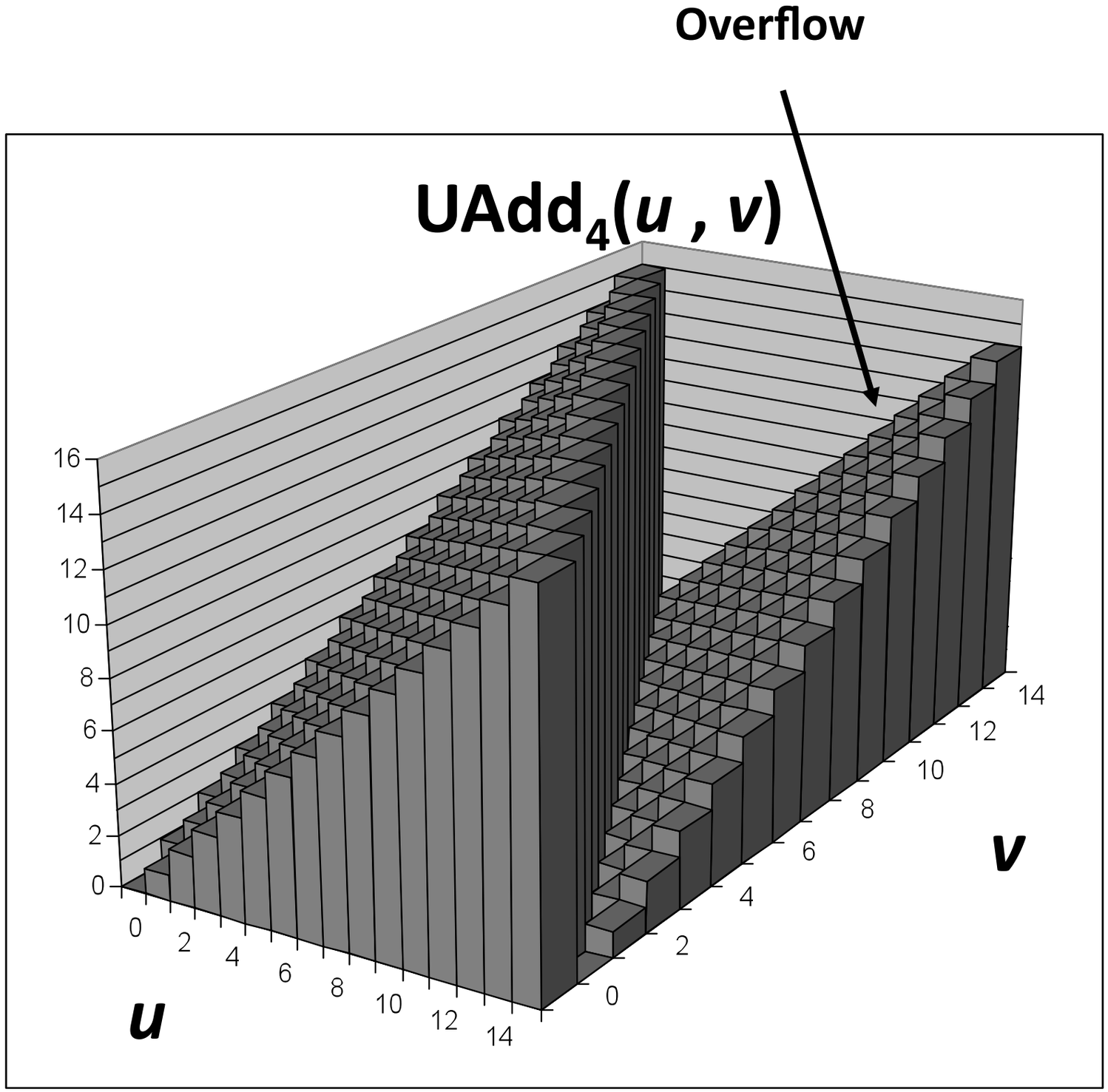
Converting to and from two's complement

- To decode two's complement:
  - If the first bit is `0` then number is positive
  - If the first bit is `1`, then number is negative:
    - subtract `1`
    - invert bits
- E.g., `110010`
  - Subtract one: `110010` - `1` = `110001`
  - Invert the bits: `001110` = 14
  - `110010` encodes -14

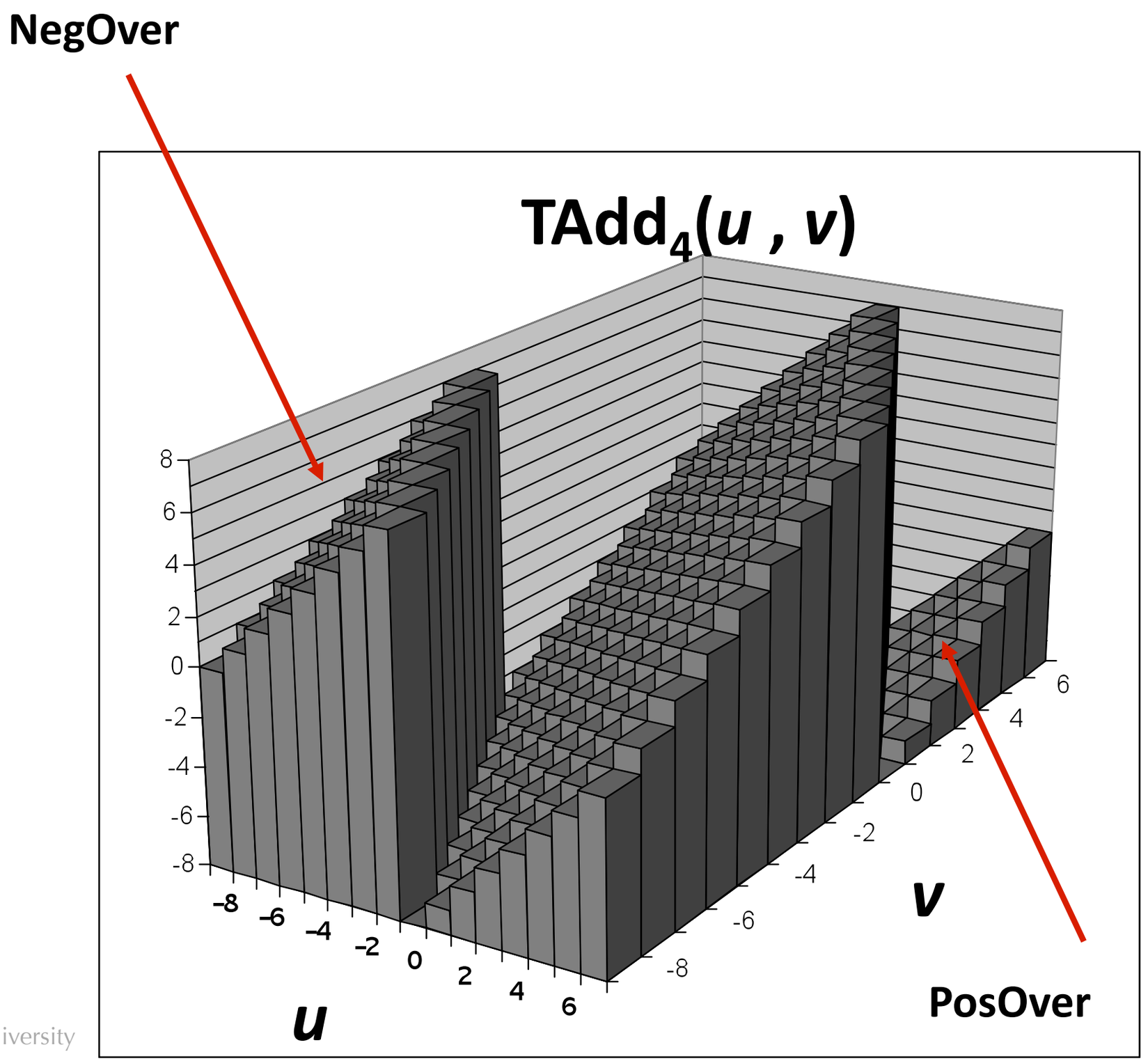
Integer overflow

- Overflow can also occur with negative integers
- With 32 bits, maximum integer expressible in 2's complement is 2^{31}-1 = `0x7fffffff`
- `0x7fffffff` + `0x1` = `0x80000000` = -2^{31}
  - Minimum integer expressible in 32-bit 2's complement
- `0x80000000` + `0x80000000` = `0x0`

Integer overflow



Integer overflow



Topics for today

- Representing information
  - Hexadecimal notation
  - Representing integers
  - Storing information
    - Word size, data size
    - Byte ordering
  - Representing strings
  - Representing code
- Basic processor operation

Word size

- Every computer has a **word size**
  - Indicates number of bits that can be used to store integers, memory addresses
- We are in transition between 32-bit machines and 64-bit machines
  - 32-bit machines can name 2^{32} different memory locations
    - 1 byte per memory location = 4 gigabytes (= 4×2^{30} bytes)
  - 64-bit machines can name 2^{64} different memory locations
    - 1 byte per memory location = 16 exabytes (= 16×2^{60} bytes)

Data sizes

- C language has multiple data formats for integer and floating-point data

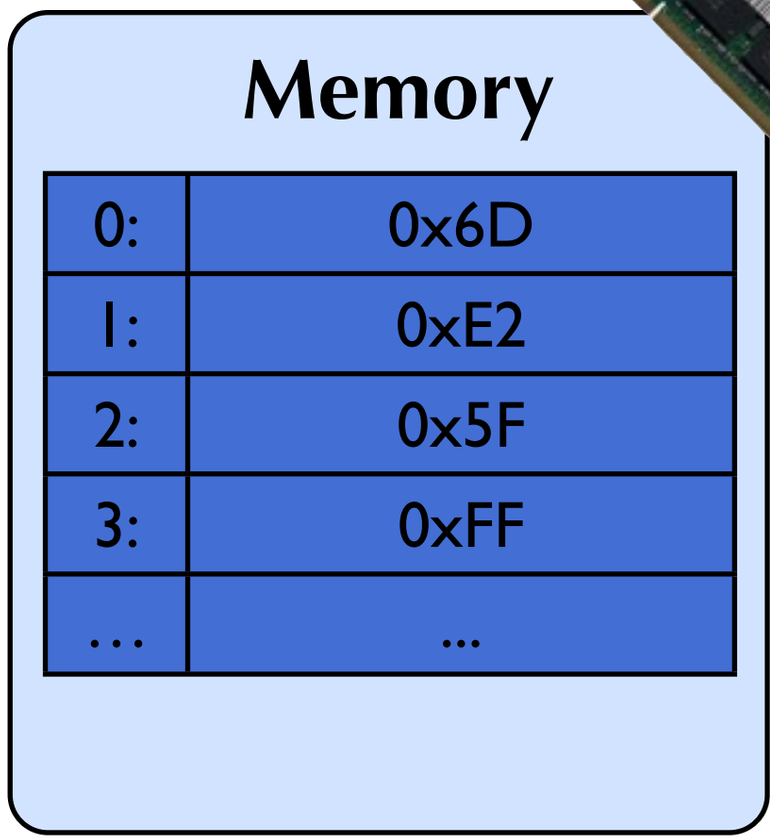
C declaration	32-bit	64-bit
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8
char *	4	8
float	4	4
double	8	8

Typical ranges

C declaration	32-bit machine		64-bit machine	
	min.	max.	min.	max.
`char`	-128	127	-128	127
`unsigned char`	0	255	0	255
`short`	-32,768	32,767	-32,768	32,767
`unsigned short`	0	65,535	0	65,535
`int`	-2,147,483,648	2,147,483,647	-2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295	0	4,294,967,295
`long`	-2,147,483,648	$2{,}147{,}483{,}647 = 2^{31}-1$	-2^{63}	$2^{63}-1$
`unsigned long`	0	$4{,}294{,}967{,}295 = 2^{32}-1$	0	$2^{64}-1$
`long long`	-2^{63}	$2^{63}-1$	-2^{63}	$2^{63}-1$
`unsigned long long`	0	$2^{64}-1$	0	$2^{64}-1$

Byte ordering

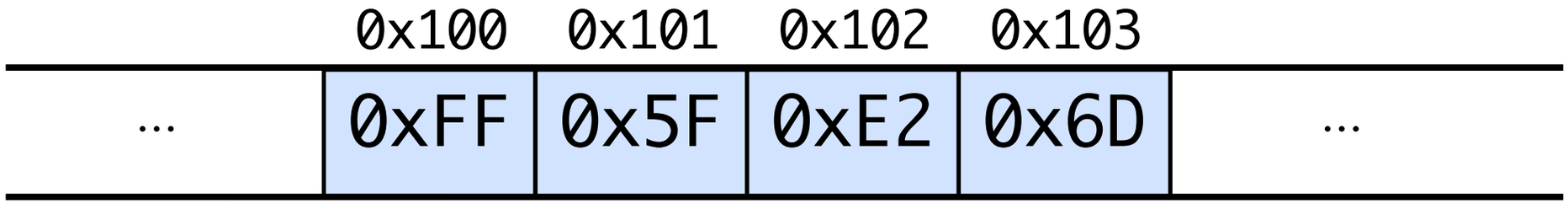
- Memory is big array of bytes
  - Address of location is integer index into array
- When we have data that is more than one byte long, which order do we store the bytes?



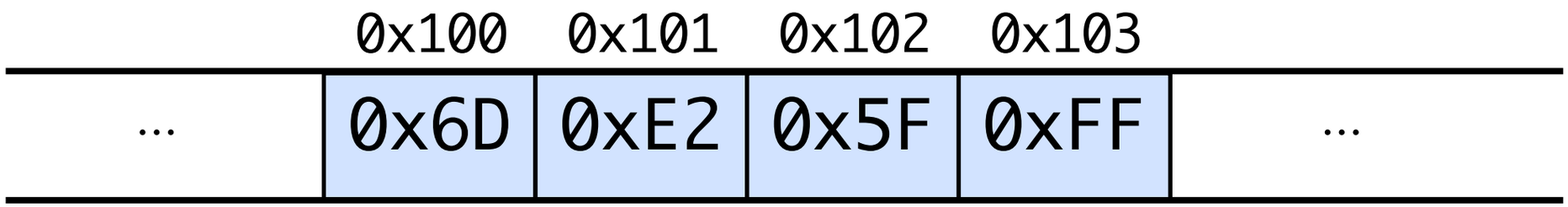
- **Big-endian**: most significant bytes first in memory
- **Little-endian**: least significant bytes first in memory
  - Most Intel-compatible machines are little-endian

Byte ordering example

- Consider 32-bit (4 byte) integer `0xFF5FE26D`
- Suppose stored at memory address `0x100`
  - i.e., occupies locations `0x100`, `0x101`, `0x102`, `0x103`
- Big endian: most significant bits first

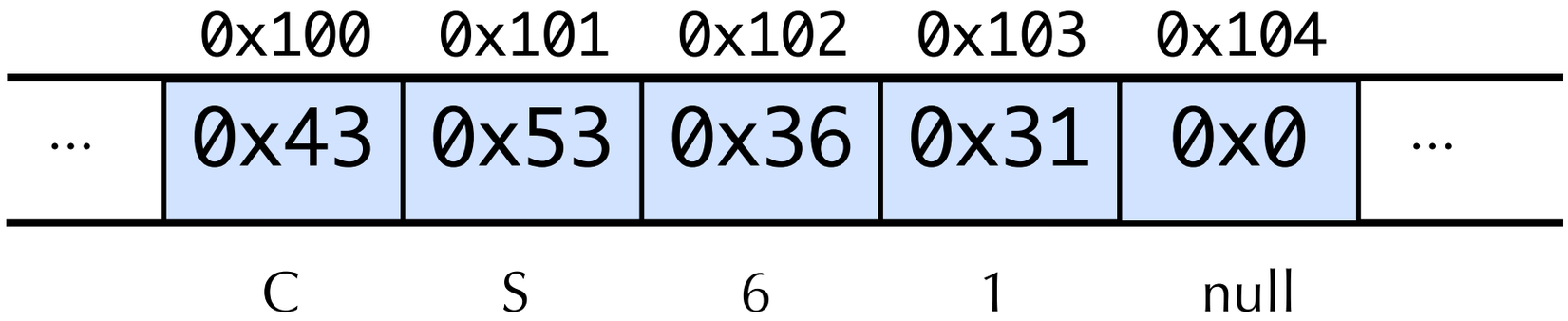


- Little endian: least significant bits first



Representing strings

- A string in C is an array of characters terminated by the null character (the character having encoding 0)
- Each character is encoded
  - Most common encoding is ASCII
    - Each character encoded in a single byte
    - Good for English-language documents, but not so good for special characters, e.g., é, ç, Φ, ش, א, …
    - Run `man ascii` to see the encoding
- E.g., Encoding the string "CS61"
  - Encoding of 'C' is `0x43`, 'S' is `0x53`, '6' is `0x36`, '1' is `0x31`



Representing code

- A computer program can also be encoded in bytes
  - Bytes represent instructions for the computer to perform
- Different types of machines use different (and incompatible) instructions and encodings
- E.g., given C function

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

the following machine code is produced

- Linux 32: `55 89 E5 8B 45 0C 03 45 08 C9 C3`
- Windows: `55 89 E5 8B 45 0C 03 45 08 C9 C3`
- Sun: `81 C3 E0 08 90 02 00 09`
- Linux 64: `55 48 89 E5 89 7D FC 89 75 F8 03 45 FC C9 C3`

Topics for today

- Representing information
  - Hexadecimal notation
  - Representing integers
  - Storing information
    - Word size, data size
    - Byte ordering
  - Representing strings
  - Representing code
- Basic processor operation

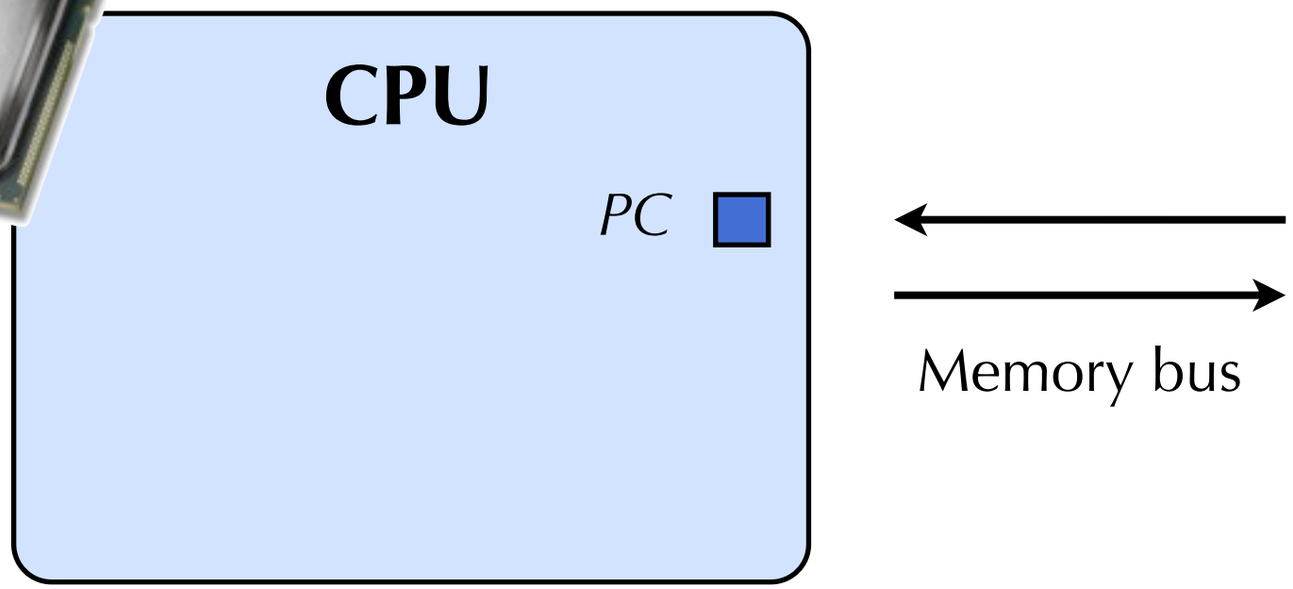
Machine code and assembly

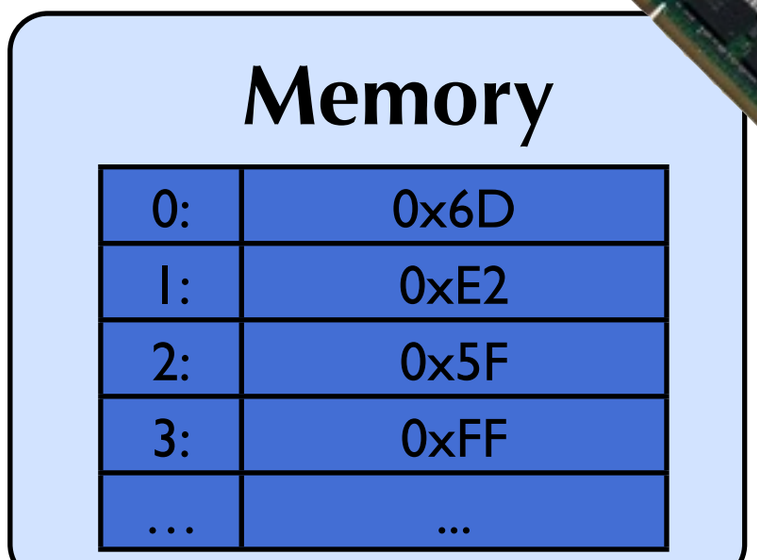
- An **instruction** is a single operation that the CPU can perform
  - add, subtract, copy, call, …
- **Machine code** is bit-level representation of instructions
  - E.g., in x86: `0x83 0xEC 0x10` represents "subtract 0x10 from value in the `%esp` register"
  - Different instructions may take different number of bits to represent
  - Hard for humans to read
- **Assembly** is human-readable form of machine code
  - E.g,. `sub 0x10, %esp`

Instruction Set Architecture (ISA)

- Definition of machine instructions and format used internally by CPU
  - What instructions the processor can perform, how they are represented, what data types they operate on, etc.
- Specific to the kind of chip and manufacturer
  - Many ISAs, e.g., Alpha, ARM, MIPS, PowerPC, SPARC
- In this course we study the **Intel IA-32 ISA** (aka **x86**)
  - Originated by Intel
  - For 32 bit architectures
  - Evolved (and backward compatible) from earlier ISAs
- Will see some of **x86-64**
  - 64 bit extension of x86

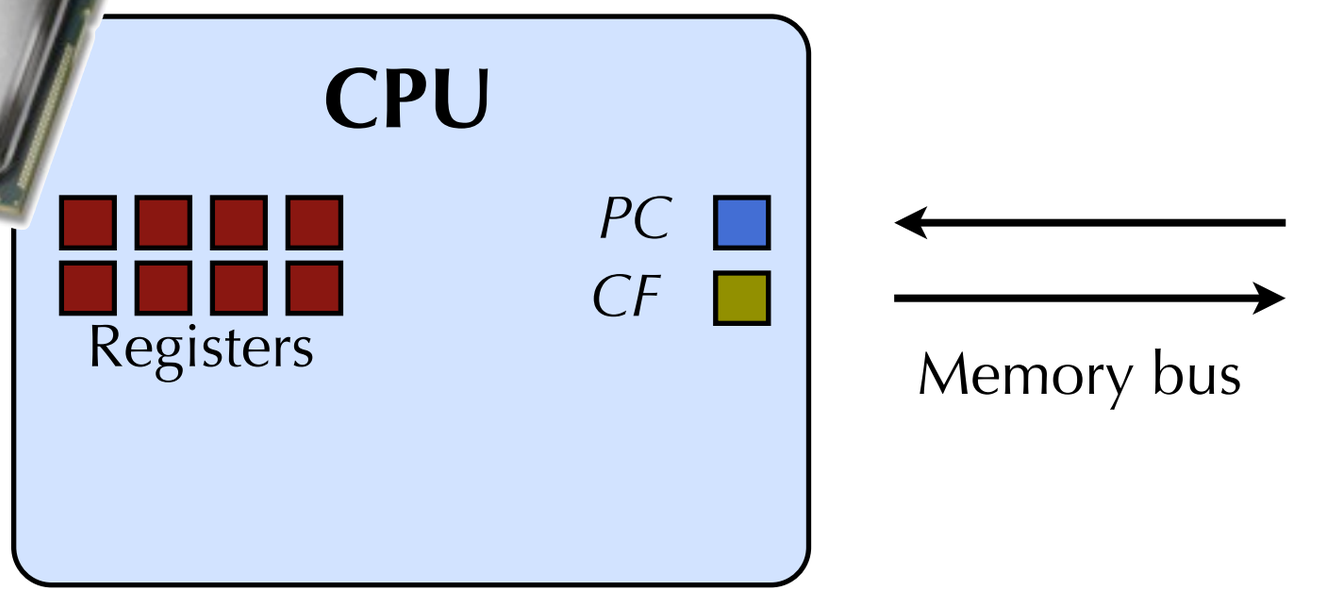
Processor architecture





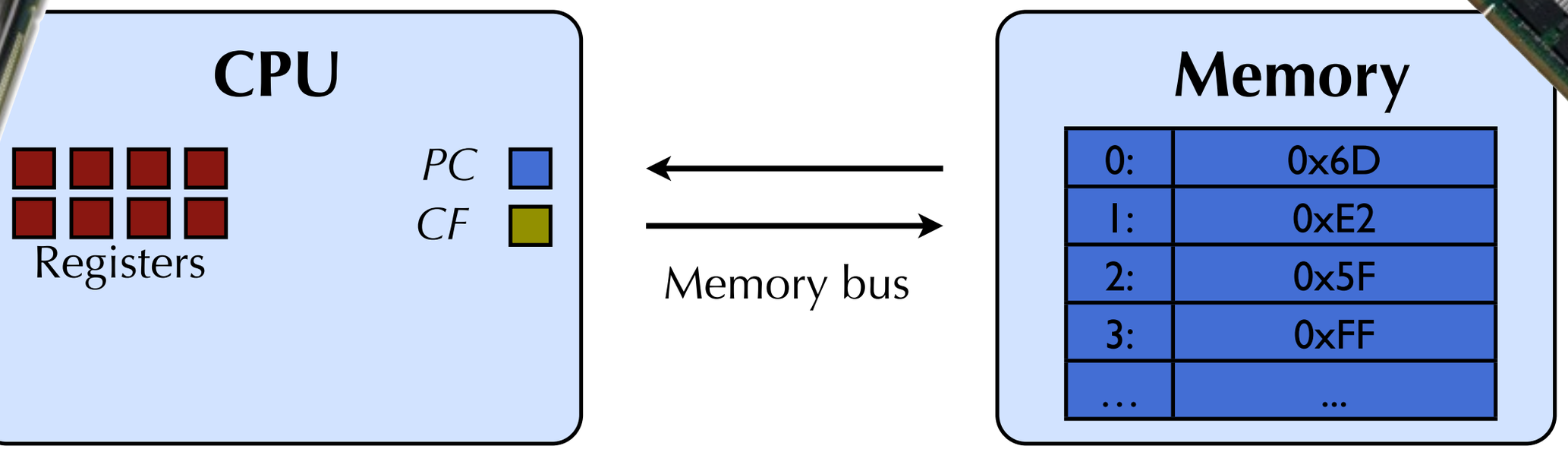
- CPU executes a series of instructions
  - Each instruction is a simple operation: add, load, store, jump, etc.
  - Instructions stored in memory
  - Program Counter (PC) holds memory address of next instruction
- CPU can read or write memory over the memory bus
  - Can generally read or write a single byte or word at a time

Processor operation



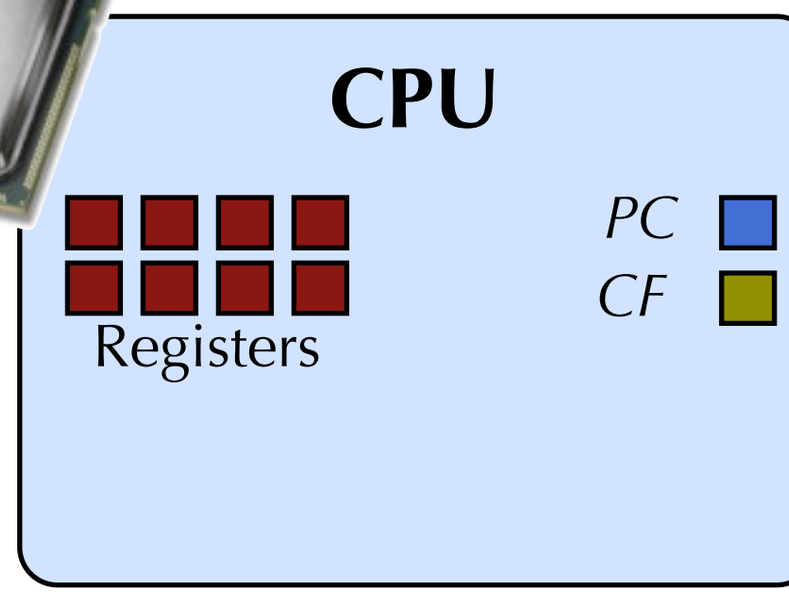
- Basic processor operation:
  - 1) Fetch instruction from memory address pointed to by program counter *PC*
  - 2) Execute instruction
  - 3) Set *PC* to address of next instruction
- Where is the next instruction?
  - Not just "*PC* + 1" – each instruction can be a different size!
  - "Jump" instruction also sets *PC* to new value

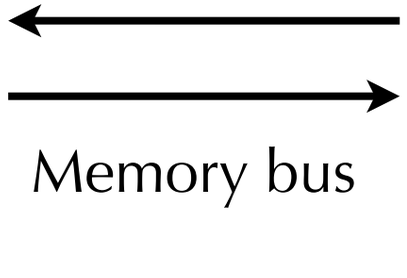
Registers

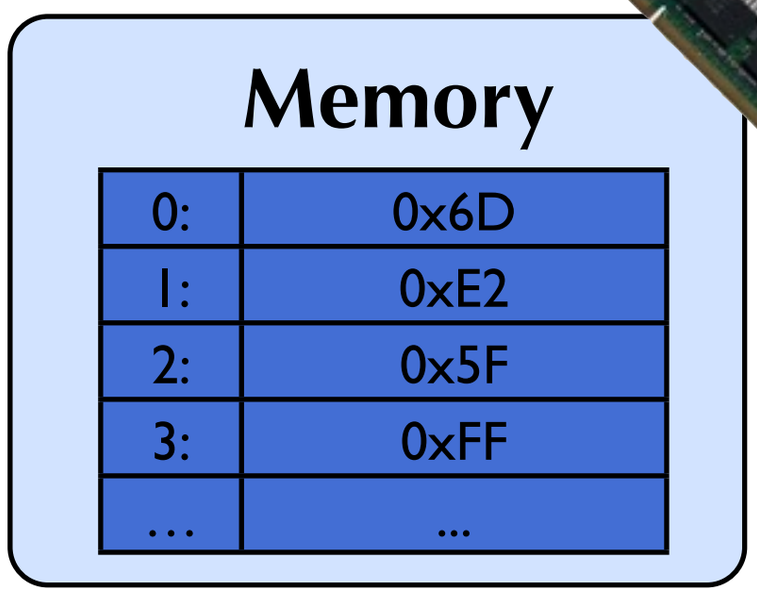


- Registers are used to store "temporary" data on the CPU itself
  - Extremely fast to access a register: 1 clock cycle (0.4 ns on a 2.4 GHz processor)
  - But reading or writing memory can ~40 ns (depends on a lot of factors)
    - Nearly 100x "slowdown" to go to memory!
- The Intel x86 has eight 32-bit registers.
  - Named `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, `%ebp`
  - There are conventions on how certain registers are used

Condition flags



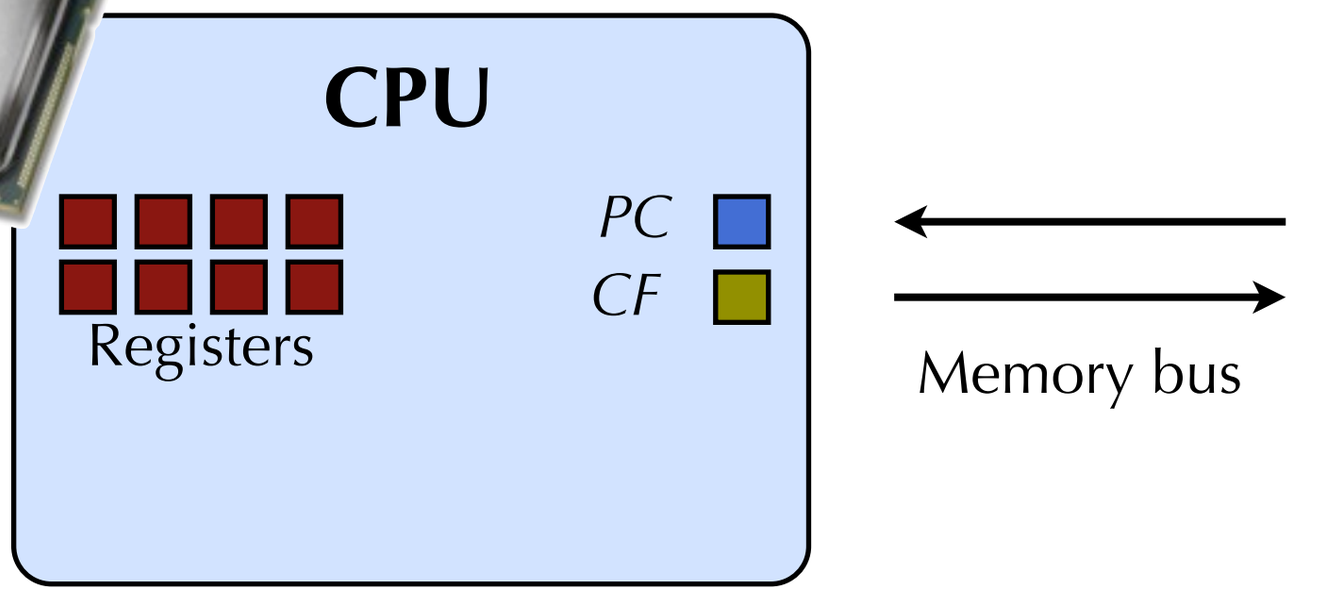




- Condition Flags (CF) hold information on state of last instruction
  - Each flag is one bit.
  - Often used by other instructions to decide what to do.
  - e.g., **Overflow flag** is set to 1 if you add two registers, and the value overflows a word.
  - **Zero flag** set to 1 if result of an operation is zero

```
subl $0x42, %eax  # Subtract 0x42 from value in %eax
jz   $0x80495BC   # If zero flag set, jump to instruction
                  #   at 0x80495BC
```

Accessing memory

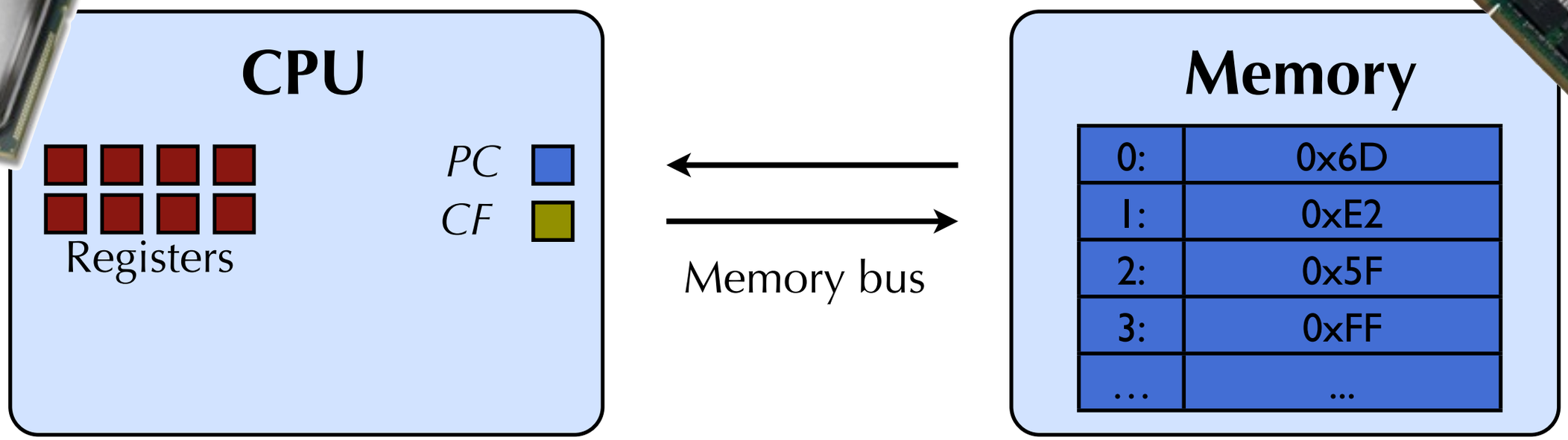


- "Move" instructions are used to read/write registers and memory locations

```
movl $0x00001000, %eax  # Set %eax register to value 0x1000
```

$ prefix indicates constant.
e.g., `$0x5395` is value `0x5395`

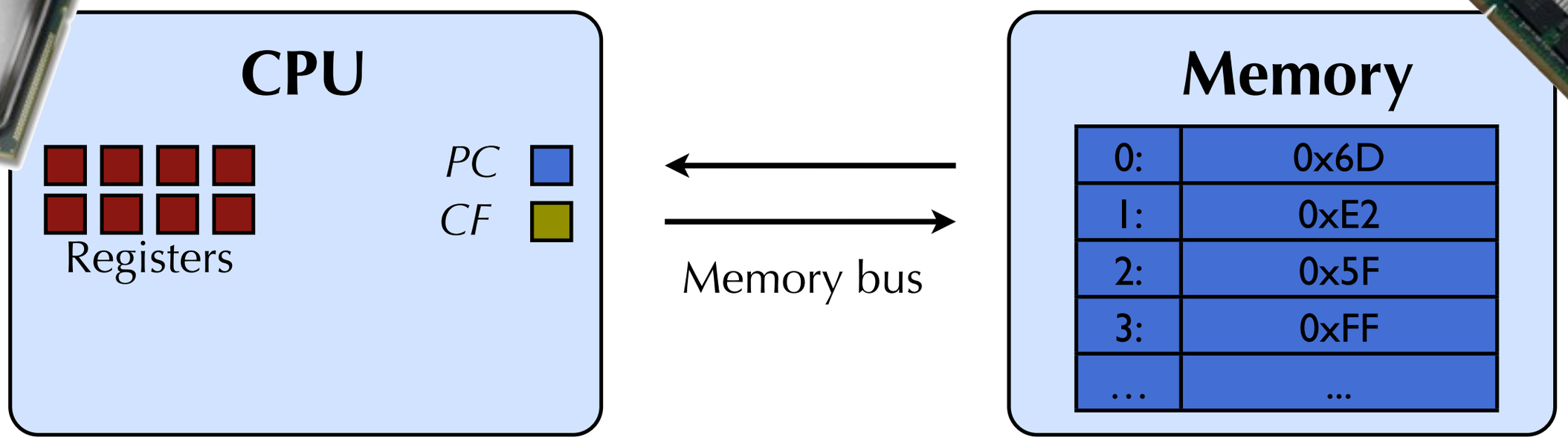
Accessing memory



- "Move" instructions are used to read/write registers and memory locations

```
movl $0x00001000, %eax  # Set %eax register to value 0x00001000
movl 0x0000ea5f, %eax   # Set %eax register to contents of
                        #   memory address 0x0000ea5f
```

Accessing memory



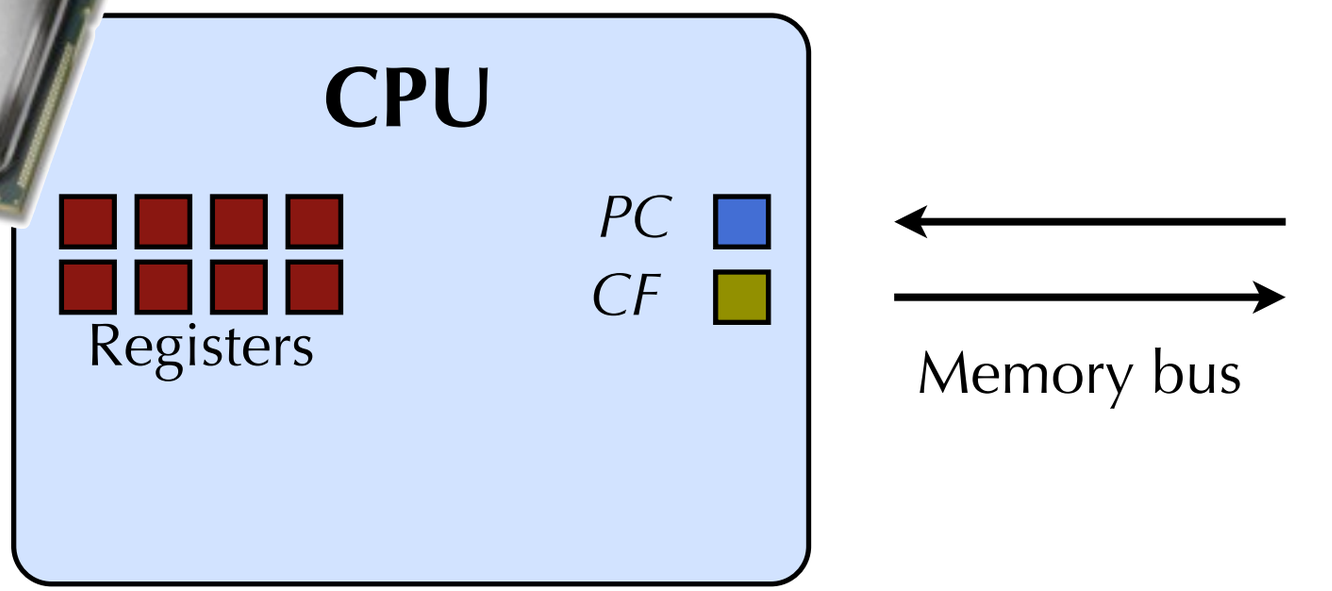
- "Move" instructions are used to read/write registers and memory

(%eax) means access memory at the address contained in %eax.
Just like dereferencing a pointer!

```
movl $                                  gister to value 0x00001000
movl 0                                  gister to contents of
                          #   memory address 0x0000ea5f

movl (%eax), %ebx         # Set %ebx register to contents of
                          #   memory address stored in %eax
```

Accessing memory



- "Move" instructions are used to read/write registers and memory locations

```
movl $0x00001000, %eax  # Set %eax register to value 0x00001000
movl 0x0000ea5f, %eax   # Set %eax register to contents of
                        #   memory address 0x0000ea5f

movl (%eax), %ebx       # Set %ebx register to contents of
                        #   memory address stored in %eax
```

More on memory

- View memory as large array of bytes
- Some conventions on how array is used
- Stack
  - Used to implement function calls, local storage
  - Every time function called, stack grows
  - Every time function returns, stack shrinks
- Heap
  - Dynamically allocated storage for program
  - Expands and contracts as result of calls to malloc and free