Bits and Bytes

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

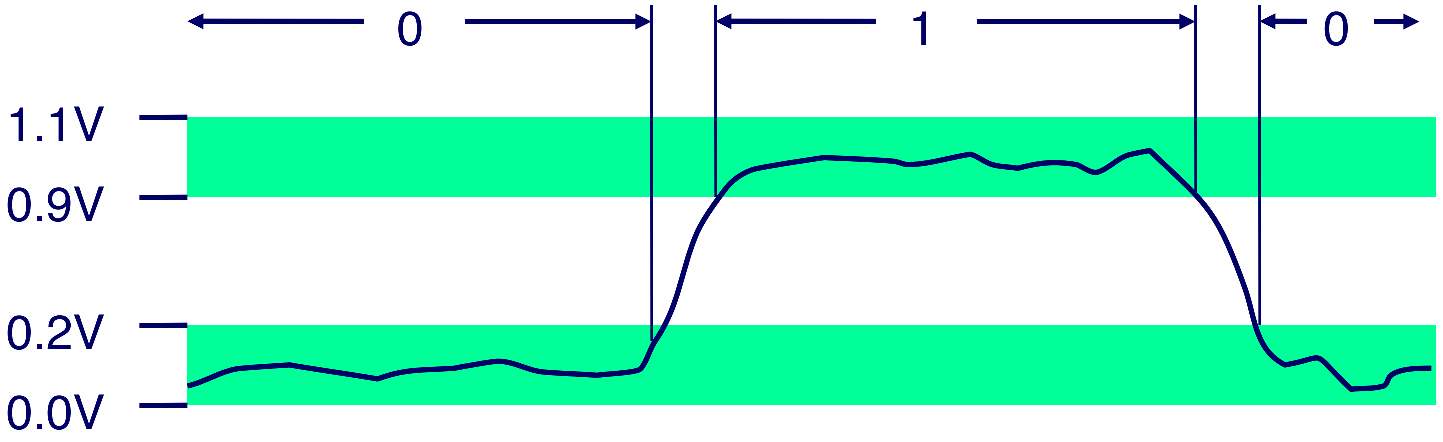
Today: Bits and Bytes

- Representing information as bits
- Bit-level manipulations
- Representations in memory

Everything is bits

- **Binary digits (*bits*) form the basis of the digital revolution**
  - Each bit is 0 or 1

- **Why bits? Electronic implementation**
  - Easy to store with bit elements
  - Reliably transmitted on noisy and inaccurate wires



Everything is bits

- **In isolation, a single bit is not very useful**

- **But, by encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…

- **Most computers use blocks of eight bits, or bytes, as the smallest addressable unit of memory**

Encoding byte values

- **Binary: $00000000_{(2)}$ to $11111111_{(2)}$**
- **Decimal: $0_{(10)}$ to $255_{(10)}$**
- **Hexadecimal: $00_{(16)}$ to $FF_{(16)}$**
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’

$$15213: \underbrace{0011}_{3}\ \underbrace{1011}_{B}\ \underbrace{0110}_{6}\ \underbrace{1101}_{D}$$

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

Today: Bits and Bytes

- **Representing information as bits**
- Bit-level manipulations
- Representations in memory

Using bits to represent numbers

- **Just like decimal except there are only two digits: 0 and 1**

- **Everything is based on powers of 2 (1, 2, 4, 8, 16, 32, ...)**
  - Instead of powers of 10 (1, 10, 100, 1000, ...)

- **Binary numbers are shorthands for sums of powers of 2**
  - $11011_{(2)} = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
    $= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$
    $= 27_{(10)}$
  - $11101101101101_{(2)} = 15213_{(10)}$
  - $1.1101101101101_{(2)} \times 2^{13} = 1.5213 \times 10^4{}_{(10)}$

Data representations

- Computers use a **limited number of bits** to encode a number
- Hence, some **operations can *overflow/underflow*** when the results are too large to be represented

C data type	Intel IA32	x86-64
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8
float	4	4
double	8	8
long double	10/12	10/16
pointer	4	8

Other types of data

- **Digital representation means that everything is represented by numbers only**
  - Text, code, sound, pictures, …
- **For sound, pictures, other “real-world” values**
  - Make accurate measurements
  - Convert them to numeric values
- **The usual sequence:**
  - Data is converted into numbers by some mechanism
  - Numbers can be stored, retrieved, processed, transmitted
  - Numbers might be reconstituted into a version of the original

Using bits to represent characters

- **Each character encoded in ASCII format**
  - American Standard Code for Information Interchange
  - Standard 7-bit encoding of character set
- **Each value between 0 and 127 represents a specific character**
- **Most computers extend the ASCII character set to use the full range of 256 characters available in a byte**
  - The upper 128 characters handle special things like accented characters

Using bits to represent characters

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Using bits to represent code

- **A program, from the perspective of the machine, is simply a sequence of bytes**
  - It has no information about the original source program (except some auxiliary tables maintained to aid in debugging)
- **Consider the following C function:**

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

Using bits to represent code

- **When compiled on a set of sample machines, we generate machine code having the following byte representations**

Linux 32:	55 89 e5 8b 45 0c 03 45 08 c9 c3
Windows:	55 89 e5 8b 45 0c 03 45 08 5d 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

- **Different machine types use different and incompatible instructions and encodings**
  - Even identical processors running different OSes have differences in their coding conventions and hence are not binary compatible

Today: Bits and Bytes

- Representing information as bits
- **Bit-level manipulations**
- Representations in memory

Boolean algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Boolean algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

Extending Boolean algebra

- **Operate on bit vectors: operations applied bitwise**

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

- **Bitwise operations have many properties in common with integer arithmetic**
  - & → Intersection
  - | → Union
  - ^ → Symmetric difference
  - ~ → Complement

Bit-level operations in C

- **Operations &, |, ~, ^ available in C**
  - Apply to any "integral" data type (`signed` and `unsigned`)
    - `long, int, short, char, …`
  - View arguments as bit vectors
  - Arguments applied bit-wise

Examples (char data type)

```
~0x41(16)                       → 0xBE(16)
~01000001(2)                    → 10111110(2)
~0x00(16)                       → 0xFF(16)
~00000000(2)                    → 11111111(2)

0x69(16)      & 0x55(16)        → 0x41(16)
01101001(2)   & 01010101(2)     → 01000001(2)
0x69(16)      | 0x55(16)        → 0x7D(16)
01101001(2)   | 01010101(2)     → 01111101(2)
```

Contrast: Logic operations in C

- **Contrast to logical operators &&, ||, !**
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - **Early termination**

Watch out for && vs. & (and || vs. |)... a common mistake in C programming

Examples (char data type)

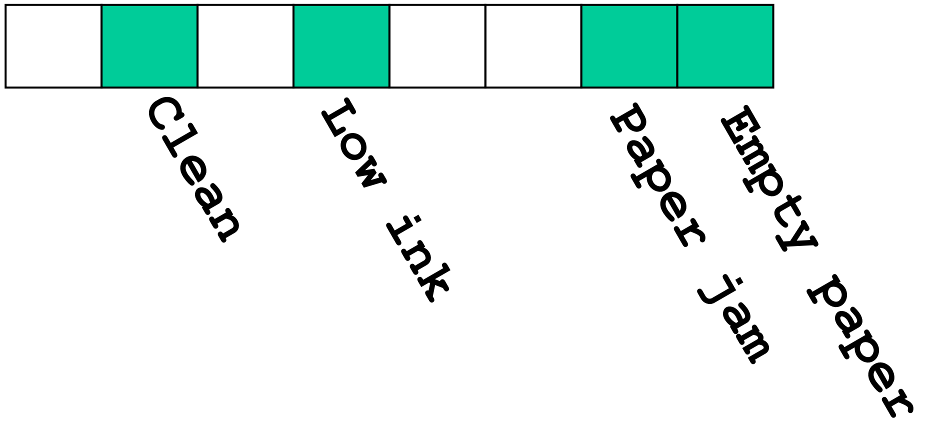
```
!0x41           →   0x00  (false)
!0x00           →   0x01  (true)
!!0x41          →   0x01  (true)

0x69 && 0x55    →   0x01  (true)
0x69 || 0x55    →   0x01  (true)
p && *p             (avoids null pointer access)
```

Bit-level operations

- **One common use of bit-level operations is to implement *masking* operations**
  - A *mask* is a bit pattern that indicates a selected set of bits within a word
- **Very useful in a number of practical applications**
  - IP addressing and subnetting
  - Hash tables
  - Controlling devices
  - Image processing
  - …

Example: Printer status register

```
#define EMPTY   01
#define JAM     02
#define LOW_INK 16
#define CLEAN   64
```



```
char status;

if (status == (EMPTY | JAM)) ...;
if (status == EMPTY || status == JAM) ...;
while (!(status & LOW_INK)) ...;

status |= CLEAN; /* turns on CLEAN bit */
status &= ~JAM;  /* turns off JAM bit */
```

Shift operations

- **C also provides a set of *shift* operations for shifting bit patterns to the left and to the right**
- **Arithmetic operators have precedence over shifts**
  - Getting the precedence wrong in C expressions is a common source of program errors, and often these are difficult to spot by inspection
- **For a n-word size value, the shift amount should be a value between 0 and n – 1**
  - Undefined behavior when shift amount < 0 or ≥ word size

Shift operations

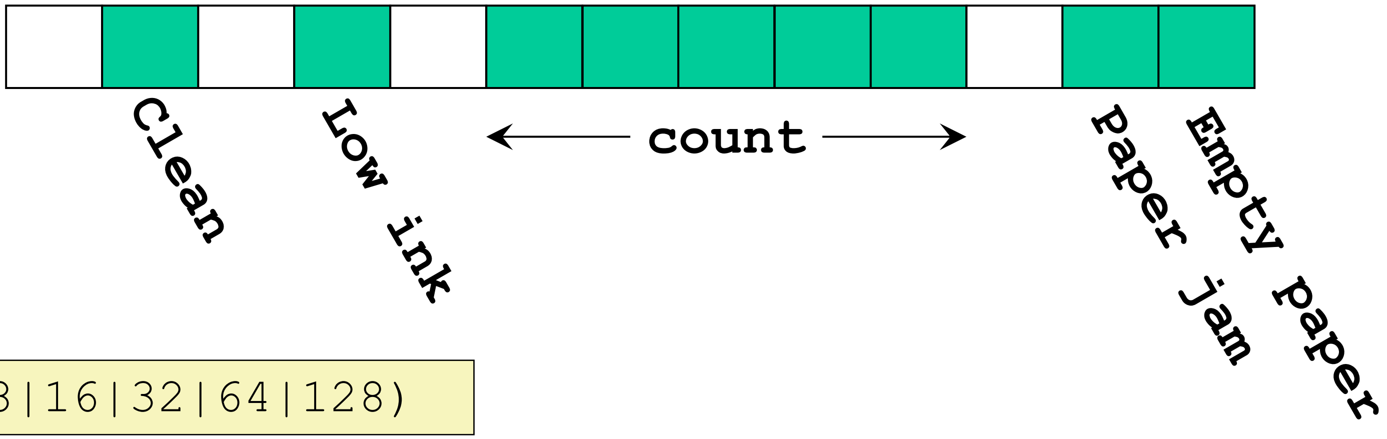
- **Left shift: `x << y`**
  - Shift bit-vector **`x`** left **`y`** positions
    - Throw away extra bits on left
    - **Fill with 0's on right**

Argument **x**	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

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

Argument **x**	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Example: Printer status register



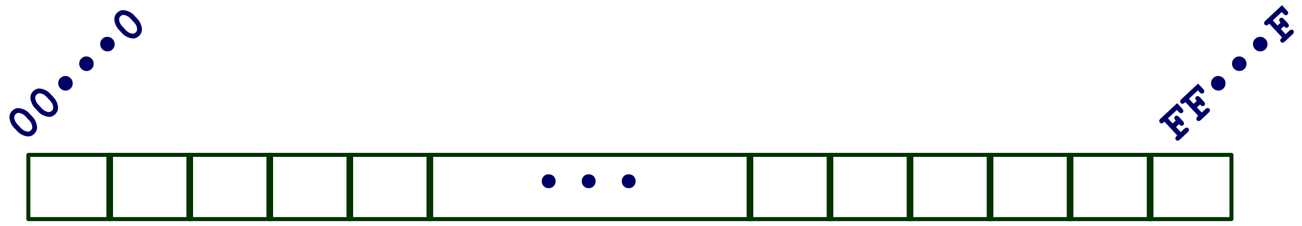
```
#define COUNT (8|16|32|64|128)
```

```
/* extract to c */
unsigned int c = (status & COUNT) >> 3;

/* insert v */
status = ((v << 3) | ~COUNT) | (status & ~COUNT);
```

Today: Bits and Bytes

- Representing information as bits
- Bit-level manipulations
- Representations in memory

Byte-oriented memory organization



- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it’s not, but can think of it that way
  - An address is like an index into that array

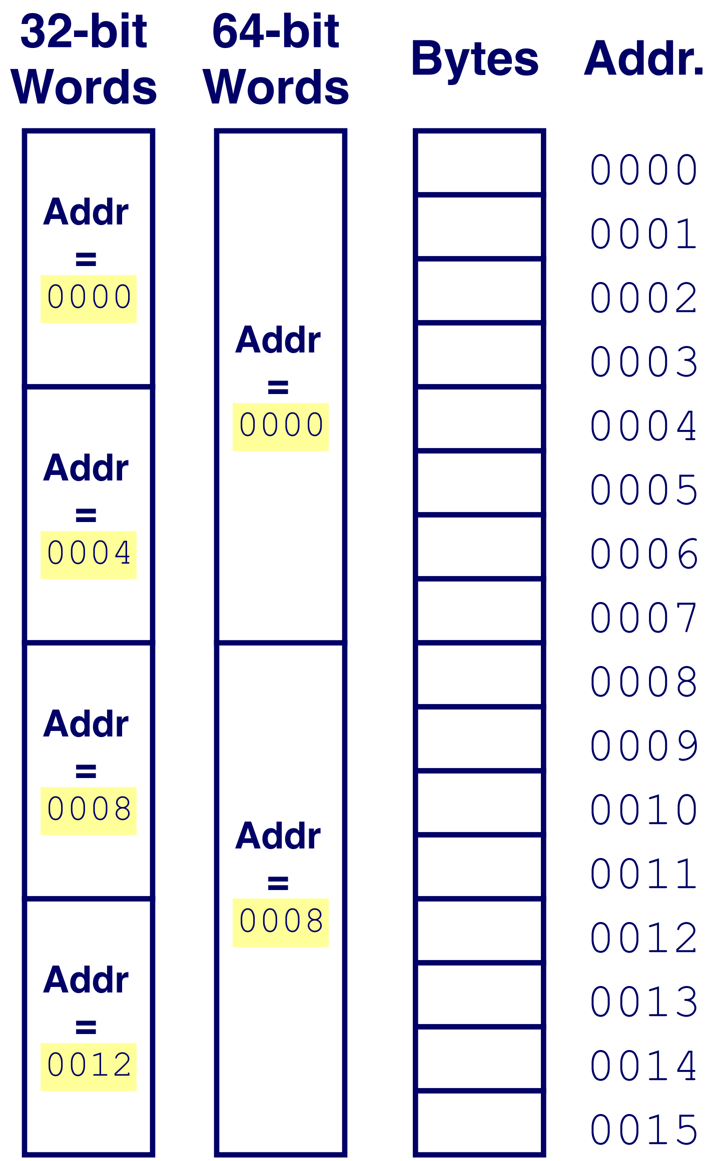
- **The OS provides private address spaces to each process**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine words

- **Any given computer has a "word size"**
  - Nominal size of integer-valued data and of addresses
- **Until recently, most machines used 32 bits as word size**
  - Limits addresses to 4GB (2^{32} bytes)
- **Increasingly, machines have 64-bit word size**
  - Potentially, could have 18 PB (petabytes) of addressable memory
  - That's 18.4 X 10^{15}
- **Machines still support multiple data formats**
  - Fractions or multiples of word size
  - Always integral number of bytes

Word-oriented memory organization

- **Addresses specify byte locations**
  - Address of first byte in word
- **Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)**



Pointers in C

- **A *pointer* is a reference to another variable (memory location) in a program**

```
int b = -15213;
int *p1 = &b;
char *p2 = (char*)&b;
```

- **The value of a pointer in C is the address of the first byte of some block of storage to which it points to**
  - Whether it points to an integer, a structure, or some other program object

Pointers in C

- **Pointer type impacts pointer arithmetic and the number of bytes that are written/read to/from memory**
  - The computed value is scaled according to the size of the data type referenced by the pointer

```
int b = -15213;
int *p1 = &b;              // p1+1 increments 4 bytes
char *p2 = (char*)&b;      // p2+1 increments 1 byte
```

- **Pointer size is determined by the the full word size of the machine**
  - 4 bytes in IA32, 8 bytes in x86-64

Byte ordering

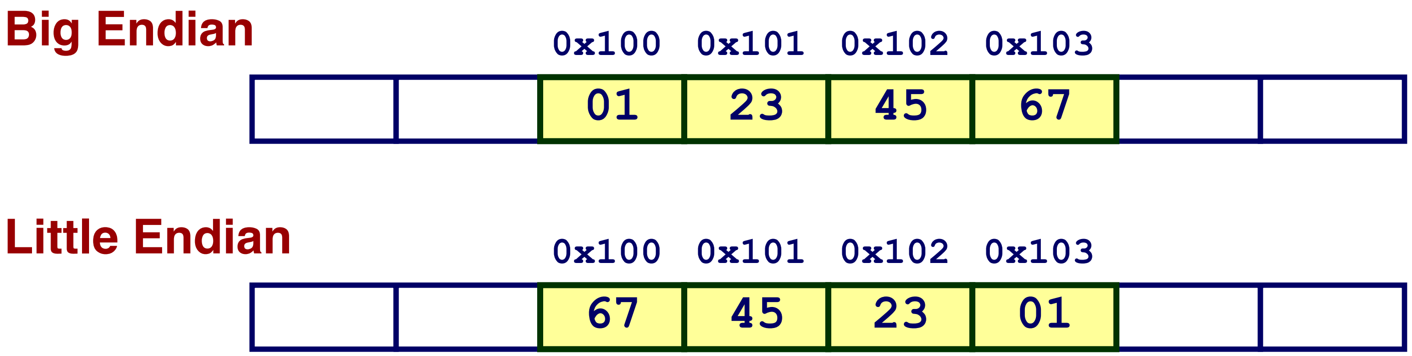
- **So, how are the bytes within a multi-byte word ordered in memory?**

- **Big Endian convention**
  - Least significant byte has highest address
  - Adopted by Sun, PPC Mac, Internet

- **Little Endian convention**
  - Least significant byte has lowest address
  - Adopted by x86, ARM processors running Android, iOS, and Windows

Byte ordering example

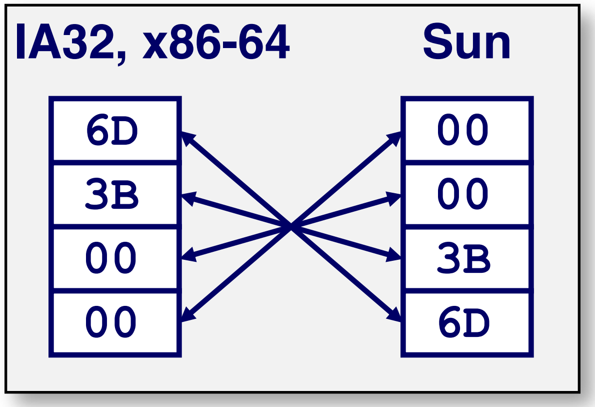
- Assume `int x` has value of 0x01234567…
- … and the address given by `&x` is 0x100



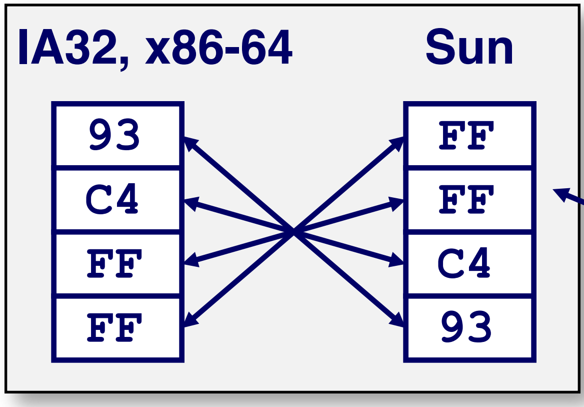
Representing integers

Decimal:	`15213`
Binary:	`0011 1011 0110 1101`
Hex:	`3 B 6 D`

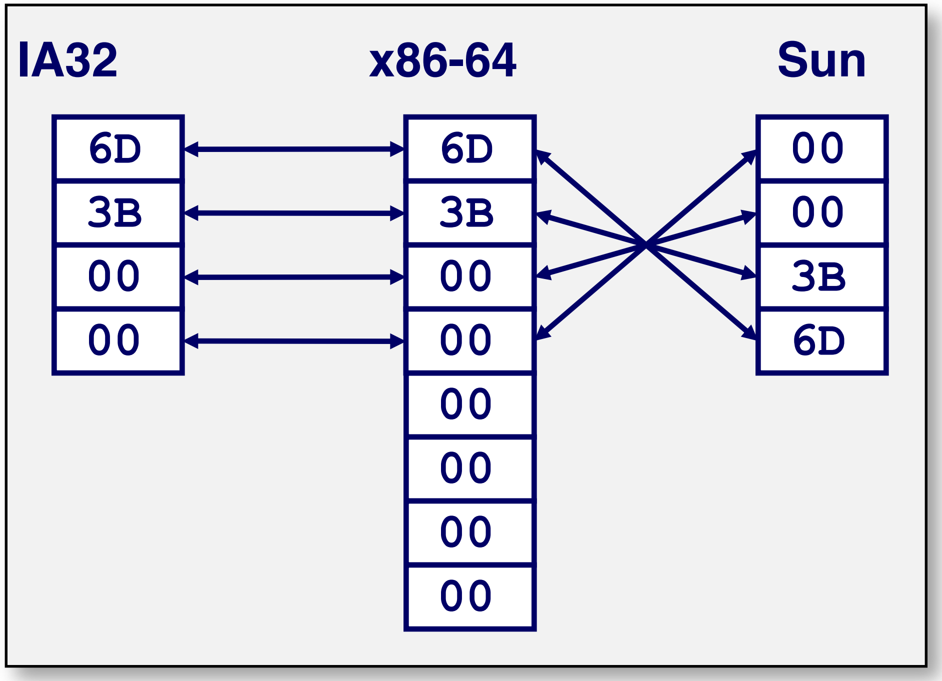
`int A = 15213;`



`int B = -15213;`



`long int C = 15213;`

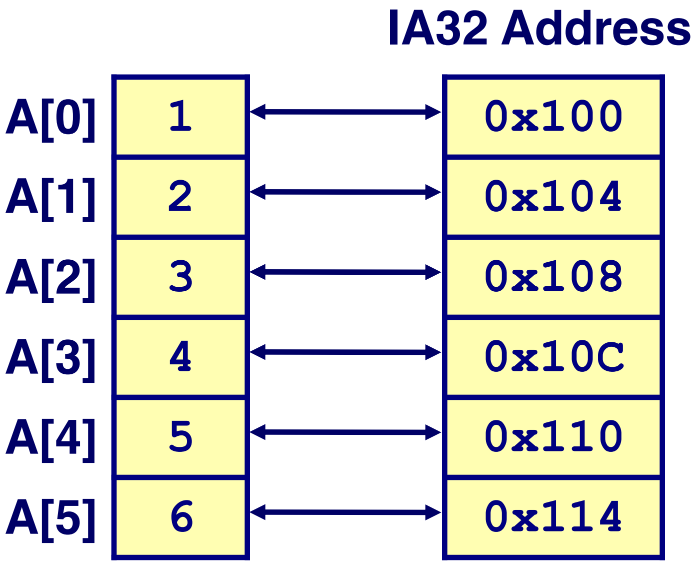


Two’s complement representation

Representing arrays

```
int A[6] = {1,2,3,4,5,6};
```

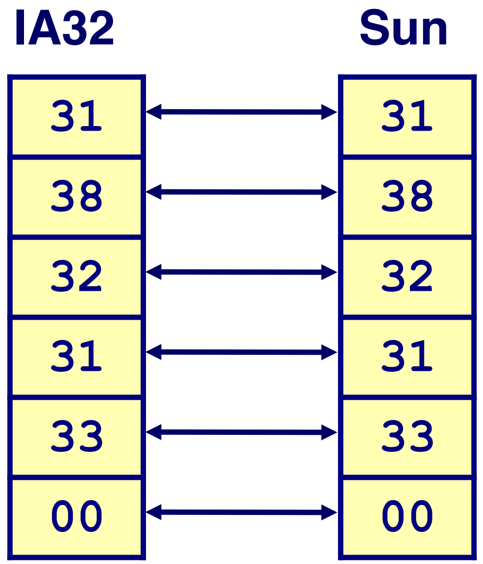
- **A fixed-size sequential collection of elements of the same type**
- **The size of the block depends on the number and type of elements**
- **Contiguous memory locations**
  - The lowest address corresponds to the first element
  - The highest address corresponds to the last element
  - Elements can be accessed by indexing the array name



Representing strings

```
char S[6] = "18213";
```

- **Represented by array of characters**
- **Each character encoded in ASCII format**
  - Character ‘0’ has code 0x30
    - Digit *i* has code 0x30 + *i*
- **String should be null-terminated**
  - Final character is 0
- **Compatibility**
  - Byte ordering is not an issue



Examining data representations

- **Code to print byte representation of data**
  - Casting pointer to *unsigned char** allows treatment as a byte array

```
void show_bytes(unsigned char* start, int len){
  int i;
  for (i = 0; i < len; i++)
    printf(”%p\t0x%.2x\n",start+i, *(start+i));
  printf("\n");
}
```

***printf* directives:**

%p:	Print pointer
%x:	Print hexadecimal

Example: `show_bytes` execution

```c
int a = 15213; /* 0x3B6D */
show_bytes((unsigned char*) &a, sizeof(int));
```

Output (Linux/IA32):

```
    0x7fffb7f71dbc    6d
    0x7fffb7f71dbd    3b
    0x7fffb7f71dbe    00
    0x7fffb7f71dbf    00
```

Summary

- **Computers encode information as bits, generally organized as sequences of bytes**

- **Different models of computers use different conventions for encoding numbers and for ordering the bytes within multi-byte data**

- **The C language is designed to accommodate a wide range of word sizes and numeric encodings**
  - Understanding these encodings at the bit level is important for writing programs that operate correctly over the full range of numeric values