MODULE II
DATA REPRESENTATION

DR. ARIJIT ROY

COMPUTER SCIENCE AND ENGINEERING GROUP

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY SRI CITY

BITS, BYTES, AND INTEGERS

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary

NUMBER REPRESENTATIONS

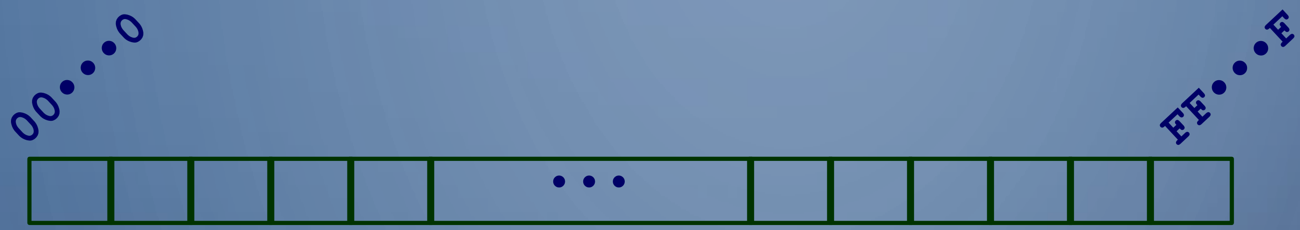
- Understand the ranges of values that can be represented and the properties of the different arithmetic operations.
- This understanding is critical to writing programs that work correctly over the full range of numeric values and that are portable across different combinations of machine, operating system, and compiler
- Whereas in an earlier era program bugs would only inconvenience people when they happened to be triggered, there are now legions of hackers who try to exploit any bug they can find to obtain unauthorized access to other people’s systems.
- This puts a higher level of obligation on programmers to understand how their programs work and how they can be made to behave in undesirable ways.

ENCODING BYTE VALUES

- Byte = 8 bits
  - 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’
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

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

BYTE-ORIENTED MEMORY ORGANIZATION



- Programs Refer to Virtual Addresses
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular "process"
    - Program being executed
    - Program can clobber its own data, but not that of others
- Compiler + Run-Time System Control Allocation
  - Where different program objects should be stored
  - All allocation within single virtual address space

MACHINE WORDS

the amount of data a CPU's internal data registers can hold and process at one time

- Machine Has "Word Size"
  - Nominal size of integer-valued data
    - Including addresses
  - Most current machines use 32 bits (4 bytes) words
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - High-end systems use 64 bits (8 bytes) words
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

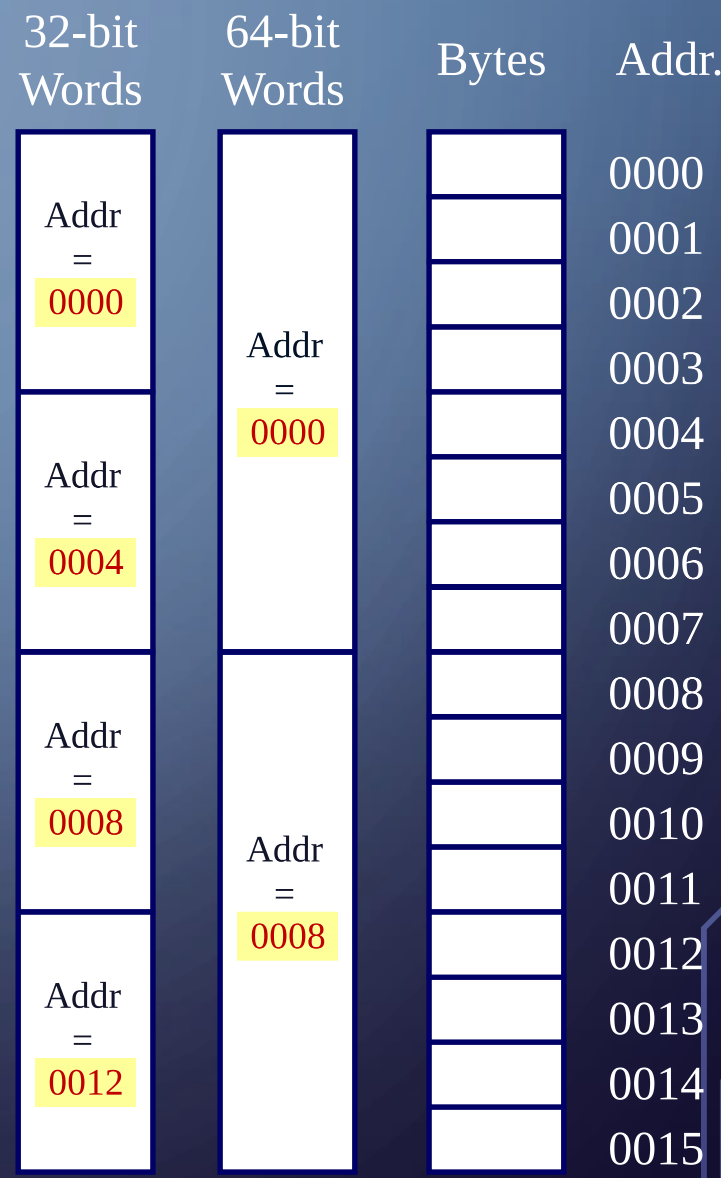
ADDRESS ACCESSIBLE AND WORD SIZE

- 32 bit word size
- Address range: $0\text{-}2^{32\text{-}1}$
- 4 GB RAM

- Check your PCs specifications

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)



In recent years, there has been a widespread shift from machines with 32-bit word sizes to those with word sizes of 64 bits.

Most 64-bit machines can also run programs compiled for use on 32-bit machines, a form of backward compatibility. So, for example, when a program prog.c is compiled with the directive

```
linux> gcc -m32 prog.c
```

then this program will run correctly on either a 32-bit or a 64-bit machine. On the other hand, a program compiled with the directive

```
linux> gcc -m64 prog.c
```

will only run on a 64-bit machine. We will therefore refer to programs as being either "32-bit programs" or "64-bit programs," since the distinction lies in how a program is compiled, rather than the type of machine on which it runs.

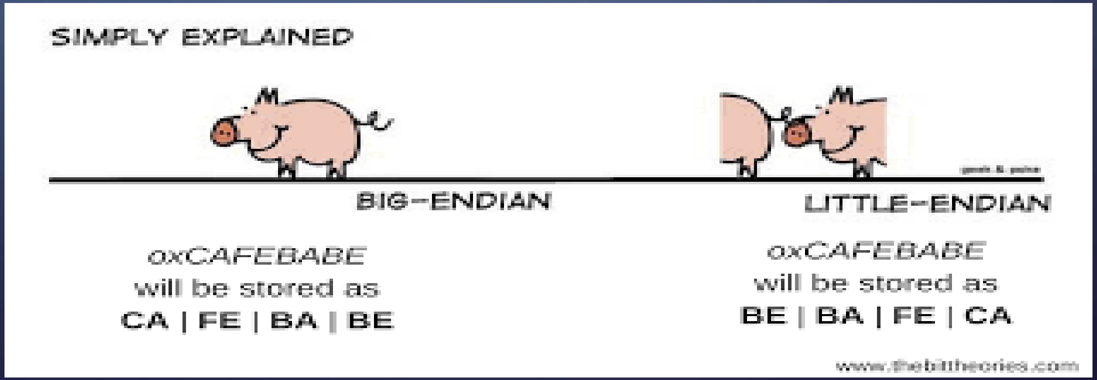
DATA REPRESENTATIONS

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

BYTE ORDERING

- How should bytes within a multi-byte word be ordered in memory?
- Example: suppose a variable x of type int has address 0x100, that is, the value of the address expression &x is 0x100. Then the 4 bytes of x would be stored in memory locations 0x100, 0x101, 0x102, and 0x103.
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - most significant byte comes first
  - Little Endian: x86
    - least significant byte comes first

the terms “little endian” and “big endian” come from the book *Gulliver’s Travels* by Jonathan Swift



BYTE ORDERING EXAMPLE

- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

A FEW POINTS…

At times, byte ordering becomes an issue. The first is when binary data are communicated over a network between different machines.

A common problem is for data produced by a little-endian machine to be sent to a big-endian machine, or vice versa, leading to the bytes within the words being in reverse order for the receiving program.

To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation

READING BYTE-REVERSED LISTINGS

Objdump –d test.o

- Disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- Example Fragment

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add $0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl $0x0,0x28(%ebx)

- Deciphering Numbers
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00

EXAMINING DATA REPRESENTATIONS

- ```
  typedef unsigned char *pointer;

  void show_bytes(pointer 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
  ```

SHOW_BYTES EXECUTION EXAMPLE

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

```c
void test_show_bytes(int val) {
    int ival = val;
    float fval = (float) ival;
    int *pval = &ival;
    show_int(ival);
    show_float(fval);
    show_pointer(pval);
}
```

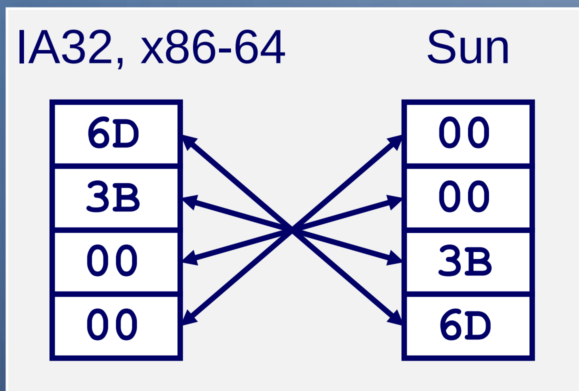
What do you notice?

Machine	Value	Type	Bytes (hex)
Linux 32	12,345	`int`	39 30 00 00
Windows	12,345	`int`	39 30 00 00
Sun	12,345	`int`	00 00 30 39
Linux 64	12,345	`int`	39 30 00 00
Linux 32	12,345.0	`float`	00 e4 40 46
Windows	12,345.0	`float`	00 e4 40 46
Sun	12,345.0	`float`	46 40 e4 00
Linux 64	12,345.0	`float`	00 e4 40 46
Linux 32	`&ival`	`int *`	e4 f9 ff bf
Windows	`&ival`	`int *`	b4 cc 22 00
Sun	`&ival`	`int *`	ef ff fa 0c
Linux 64	`&ival`	`int *`	b8 11 e5 ff ff 7f 00 00

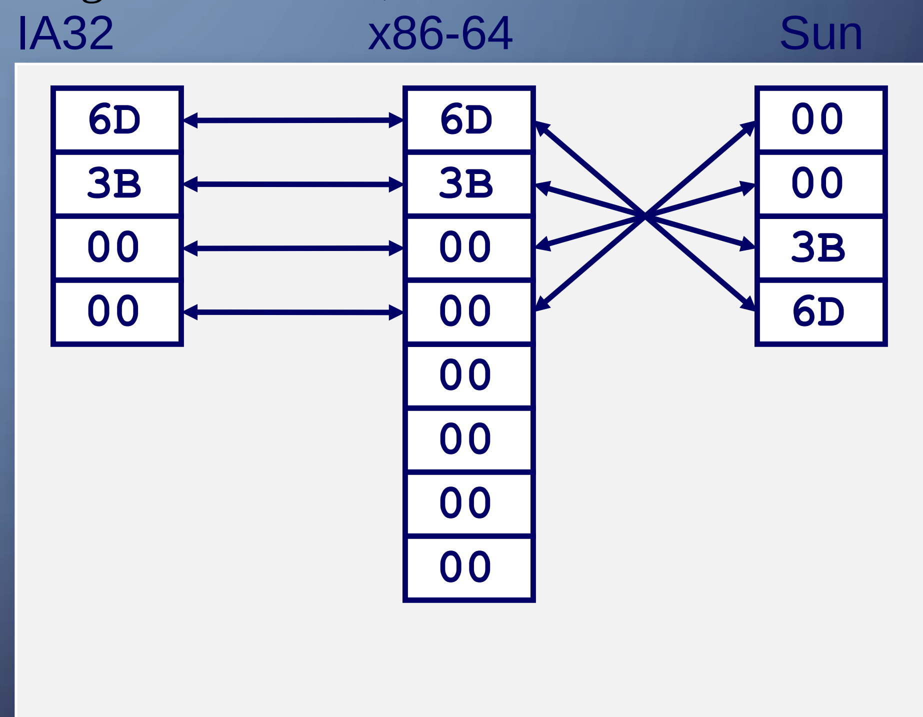
REPRESENTING INTEGERS

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

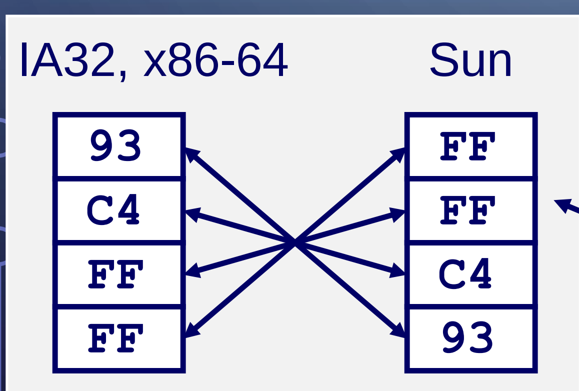
int A = 15213;



long int C = 15213;



int B = -15213;



Two’s complement representation
(Covered later)

REPRESENTING POINTERS

```
int B = -15213;
int *P = &B;
```

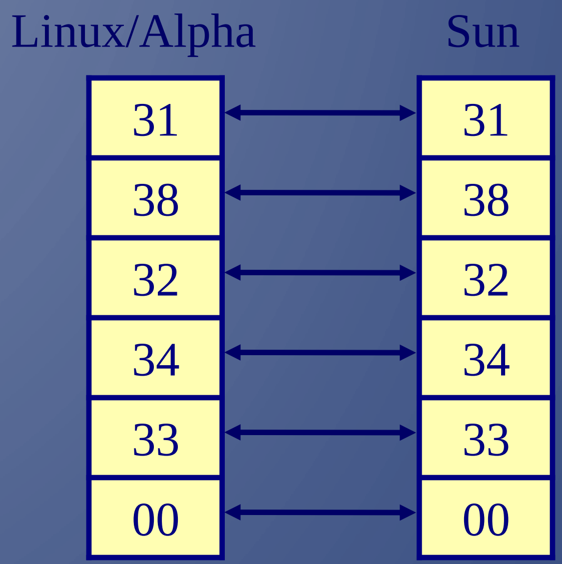
Sun	IA32	x86-64
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

Different compilers & machines assign different locations to objects

REPRESENTING STRINGS

```
char S[6] = "18243";
```

- Strings in C
  - 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 = 0
- Compatibility
  - Byte ordering not an issue



If you have a simple 8-bit character representation (e.g. extended ASCII), then no, endianness does not affect the layout, because each character is one byte.
If you have a multi-byte representation, such as UTF-16, then yes, endianness is still important

BITS, BYTES, AND INTEGERS

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary

BOOLEAN ALGEBRA

- Developed by George Boole in 19th Century, applied to logic reasoning
  - 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

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

GENERAL BOOLEAN ALGEBRAS

- Operate on Bit Vectors
  - Operations applied bitwise

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

- All of the Properties of Boolean Algebra Apply

BIT-LEVEL OPERATIONS IN C

- Operations &, |, ~, ^ Available in C
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - ~0x41 = 0xBE
    - $\sim 01000001_2 = 10111110_2$
  - ~0x00 = 0xFF
    - $\sim 00000000_2 = 11111111_2$
  - 0x69 & 0x55 = 0x41
    - $01101001_2 \ \& \ 01010101_2 = 01000001_2$
  - 0x69 | 0x55 = 0x7D
    - $01101001_2 \mid 01010101_2 = 01111101_2$

CONTRAST: LOGIC OPERATIONS IN C

- Contrast to Logical Operators
  - &&, ||, !
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
- Examples (char data type)
  - !0x41 = 0x00
  - !0x00 = 0x01
  - !!0x41 = 0x01

  - 0x69 && 0x55 = 0x01
  - 0x69 || 0x55 = 0x01
  - p && *p (avoids null pointer access) (short-circuit)

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
- 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
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

THANKS!