Computer Organization: A Programmer's Perspective

Basic Terms

Computer Science Department
Bar Ilan University, Israel

It’s all just bits to me…

A view from the CPU

1. Get next instruction from memory
2. Execute it
3. Go to step 1

Why bits? Why 0/1?

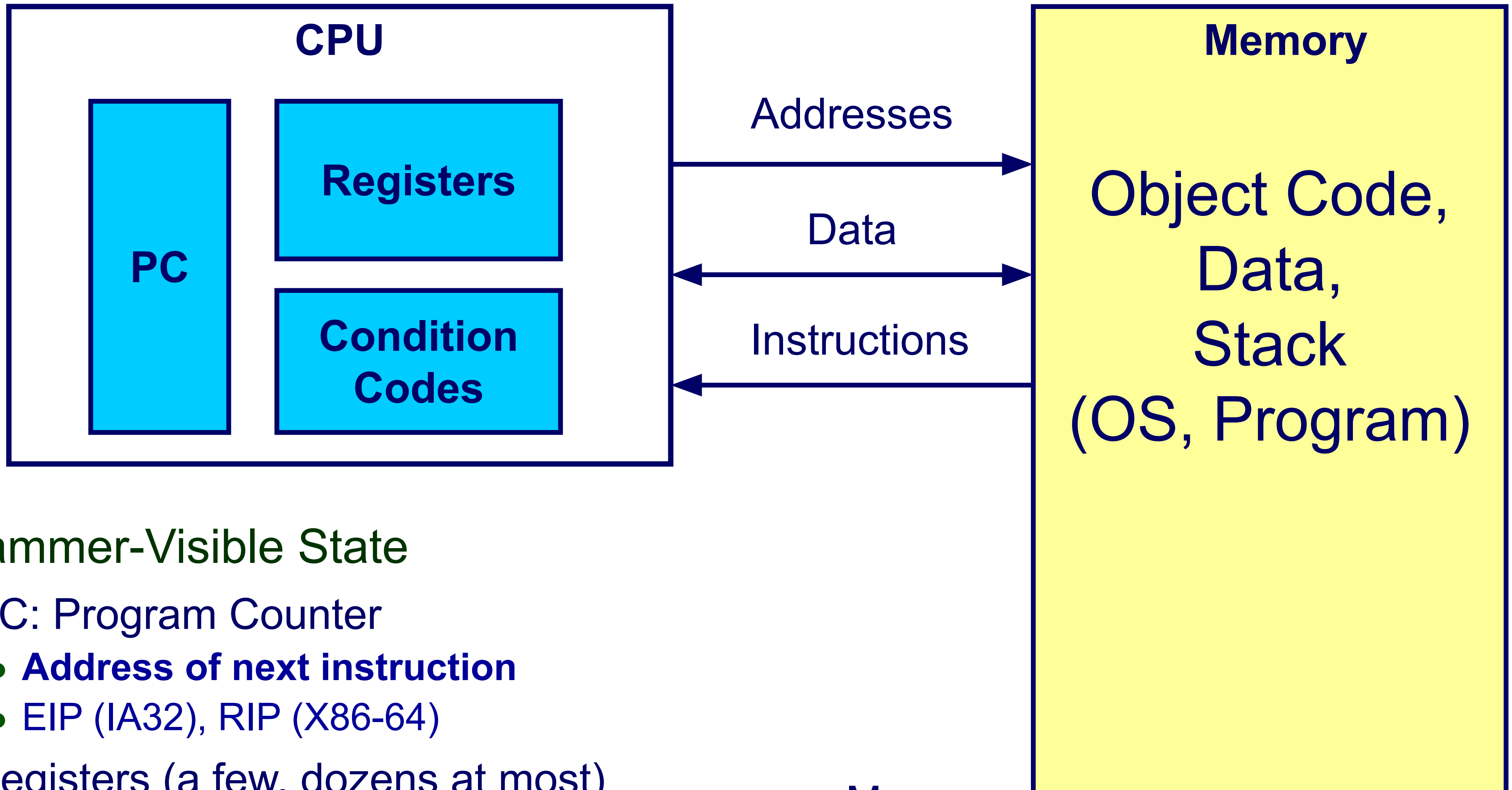
Basic terms: Bits, Bytes, Nibbles, Words

Bit-level manipulations

Boolean algebra

Expressing in C

Assembly Programmer's View of Computer



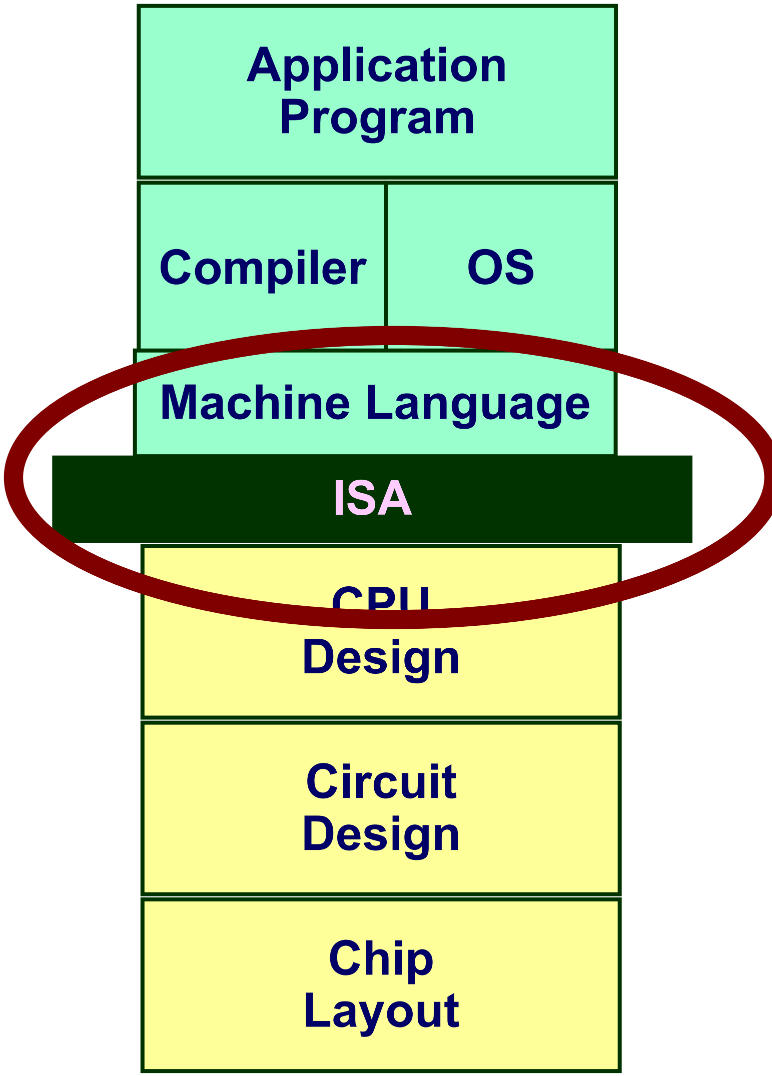
Programmer-Visible State

- PC: Program Counter
  - **Address of next instruction**
  - EIP (IA32), RIP (X86-64)
- Registers (a few, dozens at most)
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

- Memory
  - Byte addressable array
  - Code, user data, OS data
  - Includes stack used to support procedures
  - OS controls permissions

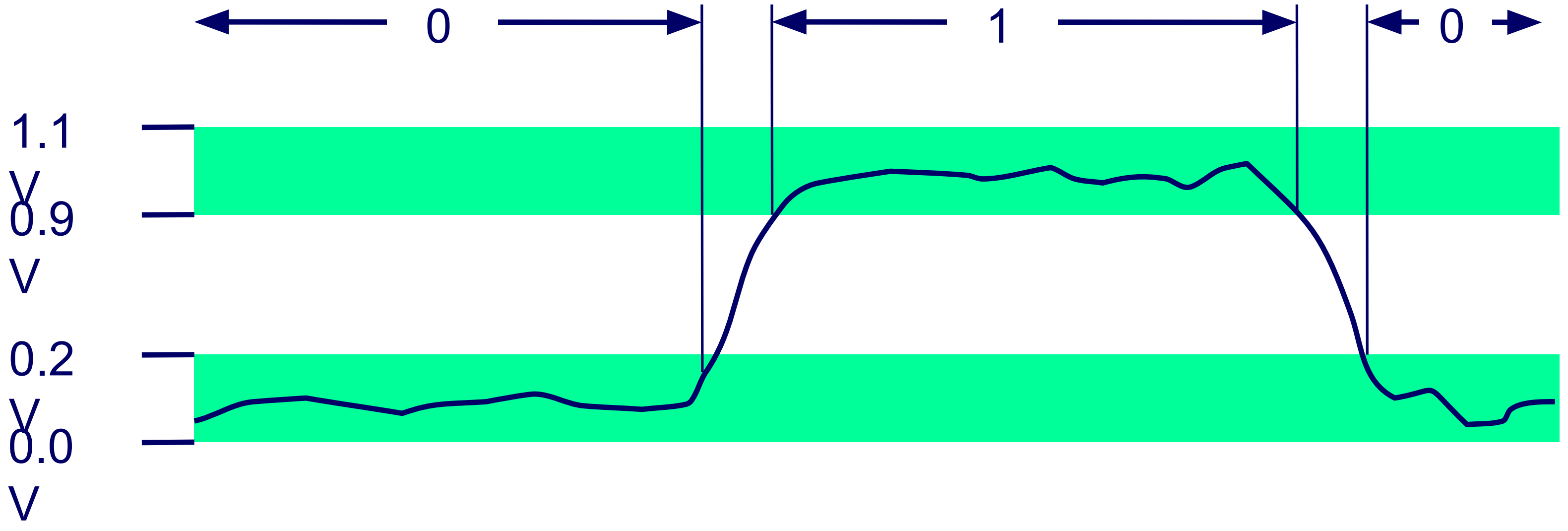
Instruction Set Architecture

- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addl`, `movl`, `leal`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers represent numbers, sets, strings, etc…
  - Computers manipulate representations (instructions)
- **Why bits? Electronic implementation is easy**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Binary Representations

- Numbers representation (base 2)
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$
  - Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
- Characters representation
  - e.g., ASCII table assigns symbol to number
- Instruction representation
  - Example in AMD64 machine code
    - RETQ command: C3 (hex) = 11000011_2
    - MOV $0, %EAX: b8 (hex) = 10111000_2 [00000000 …]

Terms

- Bit: Single binary digit, 0 or 1
- Byte: 8 bits.
  - Smallest unit of memory used in modern computers
- Nibble (English: small bite): 4 bits
  - 2 nibbles = 1 byte
- Word: 8-64 bits (1 to 8 bytes)
  - Depends on machine!

Encoding Byte Values

Byte = 8 bits

- Binary 00000000_2 to 11111111_2
  - in C: “`x=0b10010000;`”

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

Encoding Byte Values

Byte = 8 bits

- Binary 00000000_2 to 11111111_2
  - in C: “`x=0b10010000;`”
- Decimal: 0_{10} to 255_{10}

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

Encoding Byte Values

Byte = 8 bits

- Binary 00000000_2 to 11111111_2
  - in C: “`x=0b10010000;`”
- Decimal: 0_{10} to 255_{10}
- Hexadecimal (base 16) 00_{16} to FF_{16}
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Two nibbles
  - in C: “`x=0xFA1D37B`”, or “`x=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

Encoding Byte Values

Byte = 8 bits

- Binary 00000000_2 to 11111111_2
  - in C: "`x=0b10010000;`"
- Decimal: 0_{10} to 255_{10}
- Hexadecimal (base 16) 00_{16} to FF_{16}
  - Use characters '0' to '9' and 'A' to 'F'
  - Two nibbles
  - in C: "`x=0xFA1D37B`", or "`x=0xfa1d37b`"
- Octal (base 8): 0_8 to 377_8
  - in C as '`x=0256`' (0 zero)
  - 3 bits

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

Encoding Byte Values

Byte = 8 bits, nibble = 4 bits

- Binary 00000000_2 to 11111111_2
  - in C: "`x=0b10010000;`"
- Decimal: 0_{10} to 255_{10}
- Hexadecimal (base 16) 00_{16} to FF_{16}
  - Use characters '0' to '9' and 'A' to 'F'
  - Two nibbles
  - in C: "`x=0xFA1D37B`", or "`x=0xfa1d37b`"
- Octal (base 8): 0_8 to 377_8
  - in C as '`x=0256`' (0 zero)
  - 3 bits

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

Bit level operations

Boolean Algebra
(George Boole, 19th century)

- **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

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

Operators applied bitwise

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

Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- **Operations**
  - & Intersection 01000001 { 0, 6 }
  - | Union 01111101 { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference 00111100 { 2, 3, 4, 5 }
  - ~ Complement 10101010 { 1, 3, 5, 7 }

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
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - ~0x00 → 0xFF
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - 0x69 & 0x55 → 0x41
    - $01101001_2 \,\&\, 01010101_2 \rightarrow 01000001_2$
  - 0x69 | 0x55 → 0x7D
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

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

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

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

Cool Stuff with Xor: SWAP

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse

  A ^ A = 0

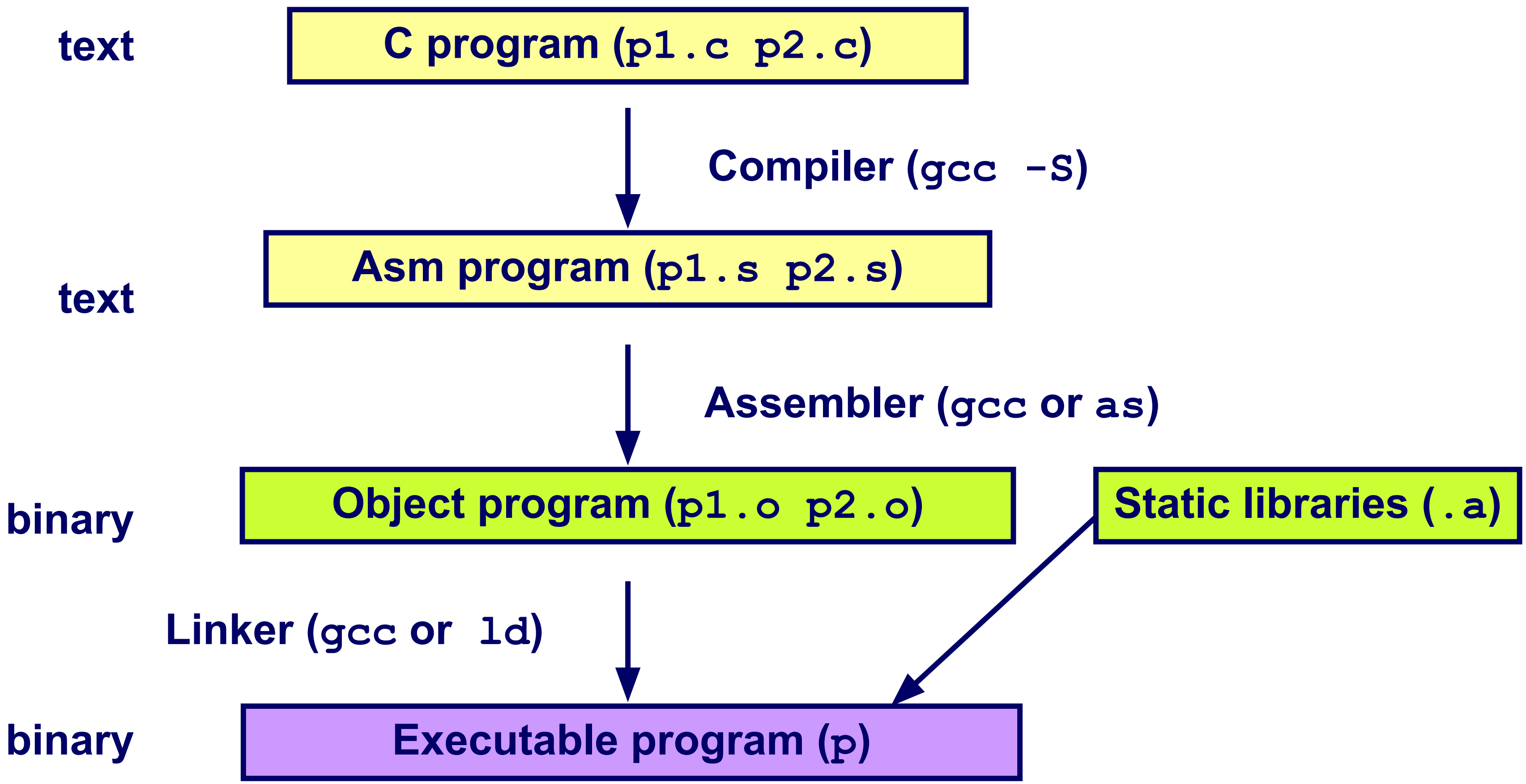
```
void funny(int *x, int *y)
{
   *x = *x ^ *y;    /* #1 */
   *y = *x ^ *y;    /* #2 */
   *x = *x ^ *y;    /* #3 */
}
```

	`*x`	`*y`
Begin	`A`	`B`
1	`A^B`	`B`
2	`A^B`	`(A^B)^B = A`
3	`(A^B)^A = B`	`A`
End	`B`	`A`

Instructions as Bits

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code (sum.c)

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

AMD64 assembly (optimized)

```
_sum:
    leal   (%rdi,%rsi), %eax
    ret
```

gcc -O1 -S sum.c

IA32 assembly (non-optimized)

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

gcc -m32 -O0 -S sum.c

IA32 assembly (optimized)

```
_sum:
    movl 4(%esp),%eax
    addl 8(%esp),%eax
    ret
```

gcc -m32 -O1 -S sum.c

Machine-Level Code Representation

- Encode Program as Sequence of Instructions
  - Arithmetic, logical, math operations
  - Read or write memory
  - Conditional branches, jumps

- Different machines, different instructions
  - Code not binary compatible (in general)
  - Different CPU “families” do not agree
    - PowerPC, ARM (tablets, phones) use fix-length instructions
    - PC’s (AMD, Intel) use variable length instructions
    - Follow different design approaches (RISC vs CISC)

Representing Instructions

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

- For this example, Alpha & Sun use two 4-byte instructions
  - Use differing numbers of instructions in other cases
- PC uses 7 instructions with lengths 1, 2, and 3 bytes

Alpha	Sun	Intel 32
00	81	55
00	C3	89
30	E0	E5
42	08	8B
01	90	45
80	02	0C
FA	00	03
6B	09	45
		08
		89
		EC
		5D
		C3

Different CPUs use totally different instructions and encodings

Storing bits (data)

Machines have Words

- Imprecise definitions
  - Nominal size of integer-valued register
  - Sometimes size of address, memory bus width
- Current desktop machines are 64 bits (8 bytes)
  - Potentially address ≈ 1.8×10^{19} bytes
  - 32-bit machines phasing out (but in phones, tablets)
- Low-end use 8- or 16-bit words
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

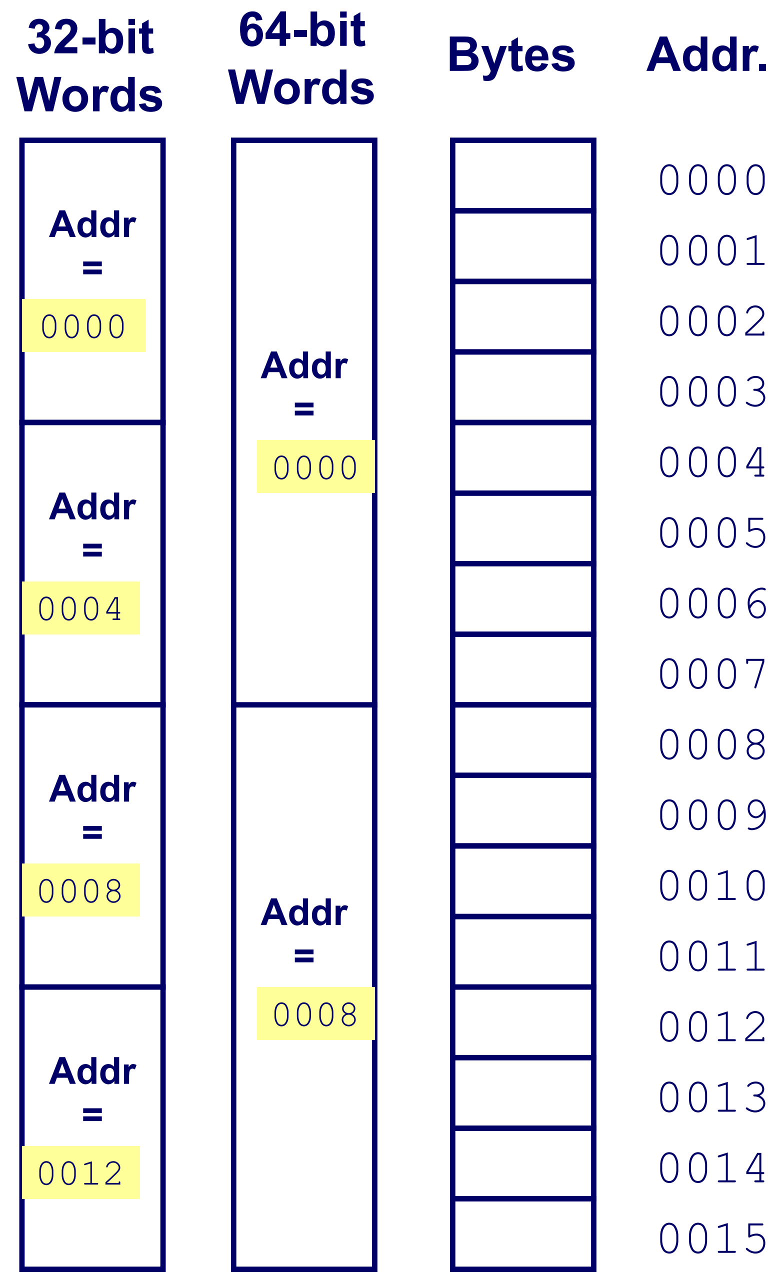
Byte-Oriented Memory Organization

- Programs Refer to Virtual Addresses
  - Conceptually very large array of bytes
  - Implemented with hierarchy of different memory types
    - SRAM, DRAM, disk
  - In Unix and Windows NT, address space private to “process”
    - Program can clobber its own data, but not that of others
  - You will see this again, in much more detail
- Compiler + Run-Time System Control Allocation
  - Where different program objects should be stored
  - Multiple mechanisms: static, stack, and heap
  - In any case, all allocation within single virtual address space
  - You will see this again, in much more detail

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)



Byte Ordering

How should bytes within multi-byte word be ordered in memory?

Conventions

- Sun’s, Mac’s are “Big Endian” machines
  - Least significant byte has highest address
- ARM, Intel are “Little Endian” machines
  - Least significant byte has lowest address
  - Some ARM CPUs have “big endian” mode

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		

Reading Byte-Reversed Listings

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 4 bytes: `0x000012ab`
- Split into bytes: `00 00 12 ab`
- Reverse: `ab 12 00 00`

Examining Data Representations

Code to Print Byte Representation of Data

- Casting pointer to `unsigned char *` creates byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
  int i;
  for (i = 0; i < len; i++)
    printf("0x%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));
```

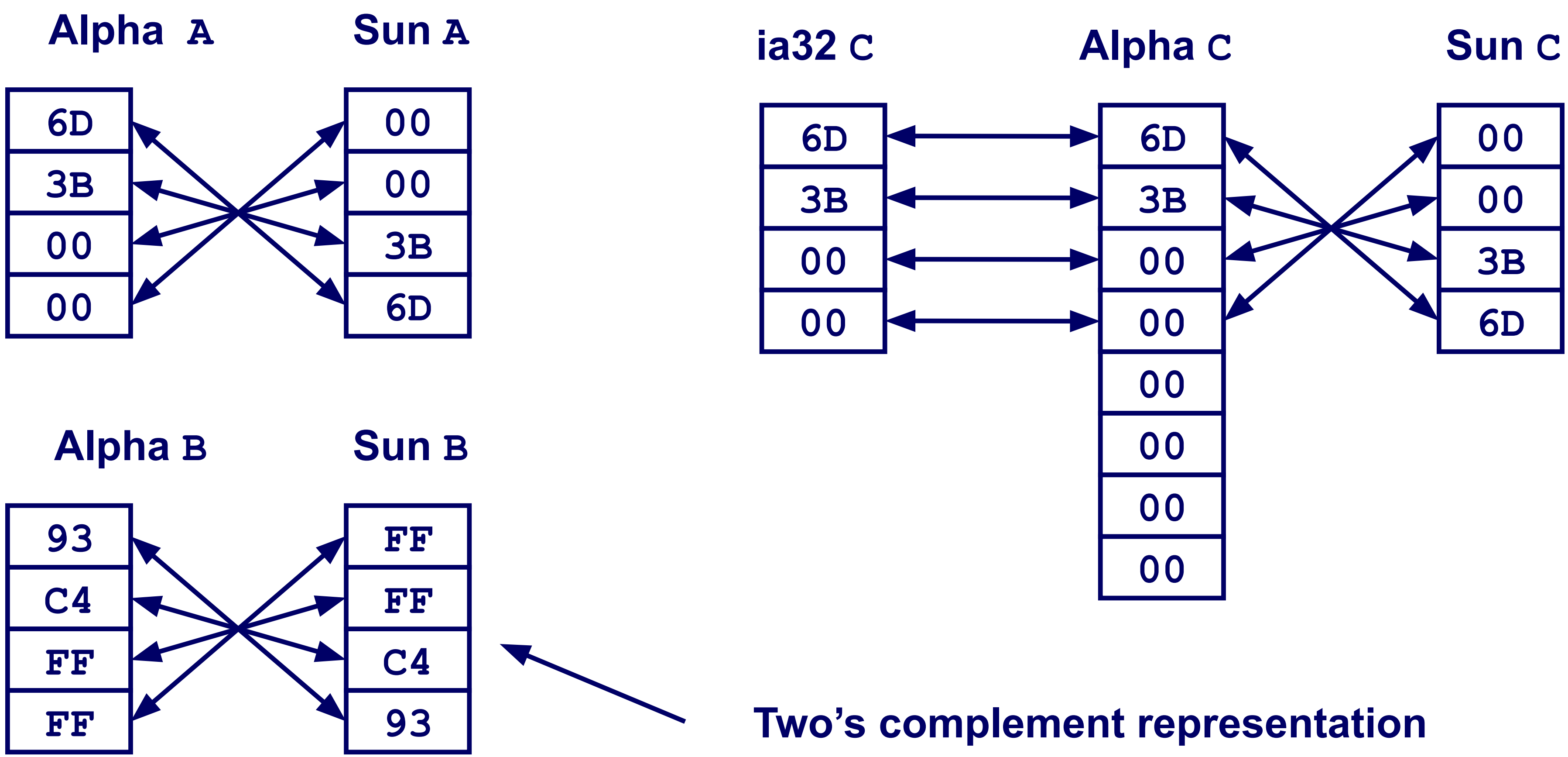
Result (on 32bit, little endian machine):

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

Representing Integers

```
int A = 15213;
int B = -15213;
long int C = 15213;
```

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



Representing Pointers

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

Alpha Address

Hex:	... 0	1	F	F	F	F	F	C	A	0
Binary:	0000	0001	1111	1111	1111	1111	1111	1100	1010	0000

Alpha P

A0
FC
FF
FF
01
00
00
00

Sun Address

Hex:	E	F	F	F	F	B	2	C
Binary:	1110	1111	1111	1111	1111	1011	0010	1100

Sun P

EF
FF
FB
2C

Linux Address

Hex:	B	F	F	F	F	8	D	4
Binary:	1011	1111	1111	1111	1111	1000	1101	0100

Linux P

D4
F8
FF
BF

Different compilers & machines assign different locations to objects

Main Points

- Boolean Algebra is Mathematical Basis
  - Basic form encodes “false” as 0, “true” as 1
  - General form like bit-level operations in C
  - Good for representing & manipulating sets

- It’s All About Bits & Bytes
  - Numbers, text, programs
- Different Machines Follow Different Conventions
  - Representations
  - Word size
  - Byte ordering