Course overview

Computer Organization and Assembly Languages
Yung-Yu Chuang

with slides by Kip Irvine

Logistics

- **Meeting time**: 2:20pm-5:20pm, Wednesday
- **Classroom**: CSIE Room 111
- **Instructor**: 莊永裕 Yung-Yu Chuang
- **Teaching assistant**:黃子桓
- **Webpage**:
  http://www.csie.ntu.edu.tw/~cyy/asm
  id / password
- **Mailing list**: assembly@cmlab.csie.ntu.edu.tw
  Please subscribe via
  https://cmlmail.csie.ntu.edu.tw/mailman/listinfo/assembly/

Caveats

- It is a course from the old curriculum.
- It is not for you if you have taken or are taking computer architecture.
- It is not tested in your graduate school entrance exam, and not listed as a required course anymore.
- It is a fundamental course, not a geek-level one.
- It is more like advanced introduction to CS, better suited to freshman or sophomore.

Prerequisites

- Better to have programming experience with some high-level languages such C, C ++,Java …

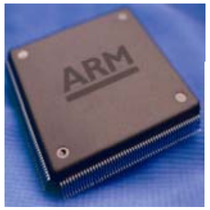
Textbook

- Readings and slides

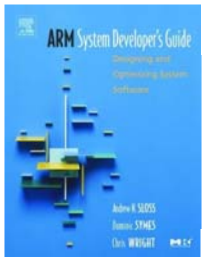
References (TOY)

Princeton's Introduction to CS,
http://www.cs.princeton.edu/introcs/50machine/
http://www.cs.princeton.edu/introcs/60circuits/

References (ARM)



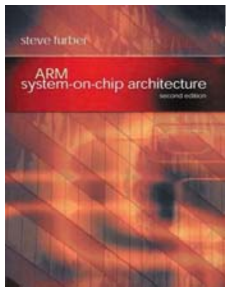
ARM Assembly Language Programming, Peter Knaggs and Stephen Welsh



ARM System Developer's Guide, Andrew Sloss, Dominic Symes and Chris Wright

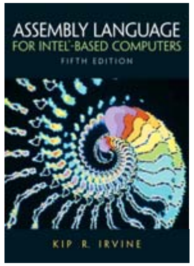
References (ARM)

Whirlwind Tour of ARM Assembly, TONC, Jasper Vijn.

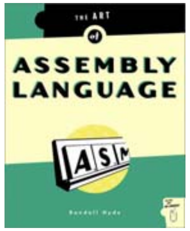


ARM System-on-chip Architecture, Steve Furber.

References (IA32)

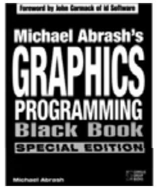


Assembly Language for Intel-Based Computers, 5th Edition, Kip Irvine

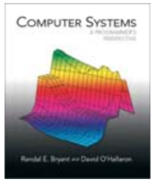


The Art of Assembly Language, Randy Hyde

References (IA32)



Michael Abrash' s Graphics Programming Black Book



Computer Systems: A Programmer's Perspective, Randal E. Bryant and David R. O'Hallaron

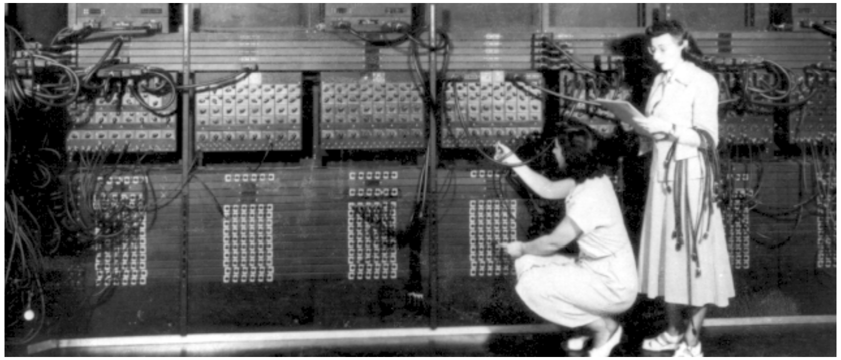
Grading (subject to change)

- Assignments (4 projects, 56%), most graded by performance
- Class participation (4%)
- Midterm exam (16%)
- Final project (24%)
  - Examples from previous years

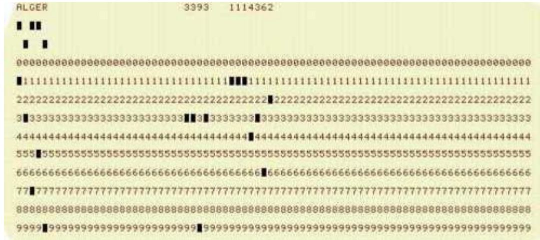
Computer Organization and Assembly language

- It is not only about assembly but also about “computer organization”.

Early computers



Early programming tools

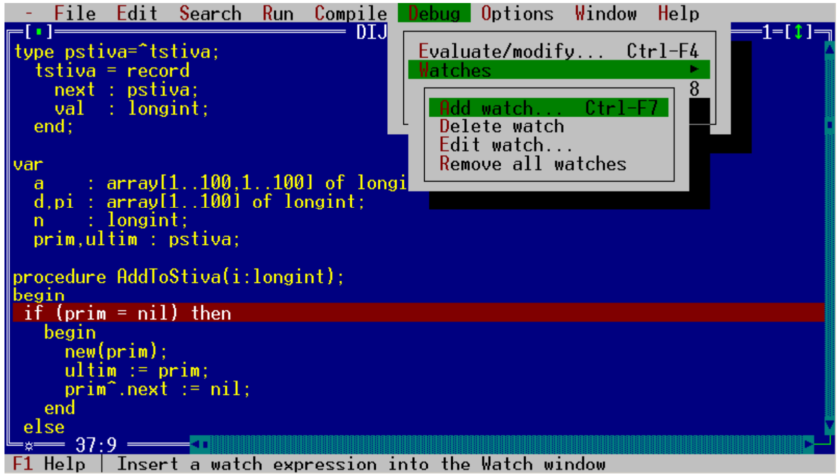


First popular PCs

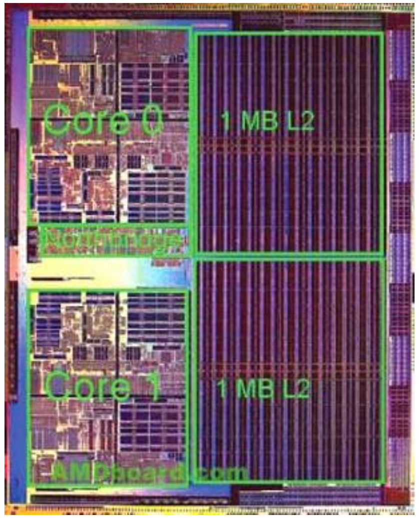
Early PCs

- Intel 8086 processor
- 768KB memory
- 20MB disk
- Dot-Matrix printer (9-pin)

GUI/IDE

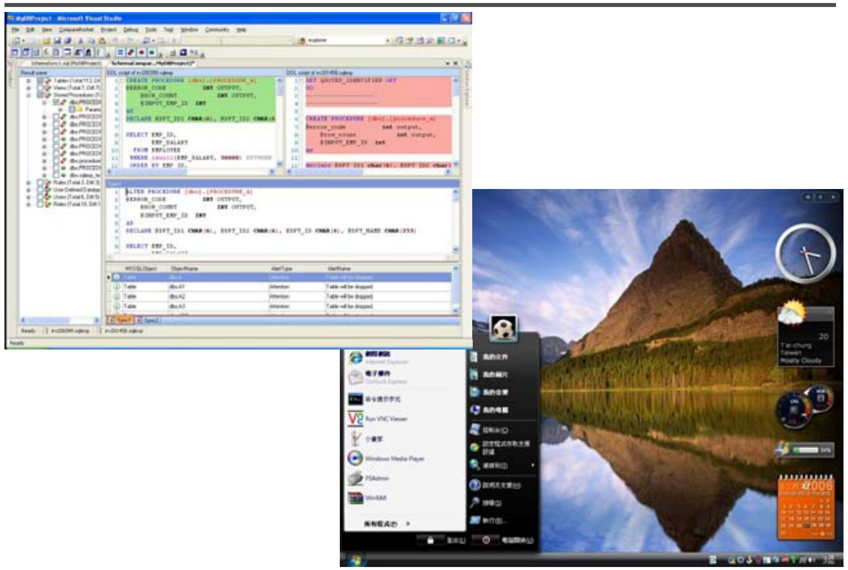


More advanced architectures



- Pipeline
- SIMD
- Multi-core
- Cache

More advanced software



More "computers" around us

My computers

Desktop
(Intel Pentium D
3GHz, Nvidia 7900)

VAIO Z46TD
(Intel Core 2 Duo P9700 2.8GHz)

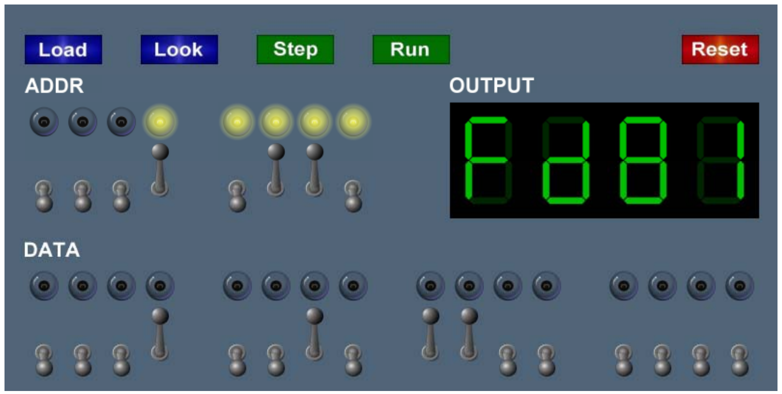
GBA SP
(ARM7 16.78MHz)

iPhone 3GS
(ARM Cortex-A8
833MHz)

Computer Organization and Assembly language

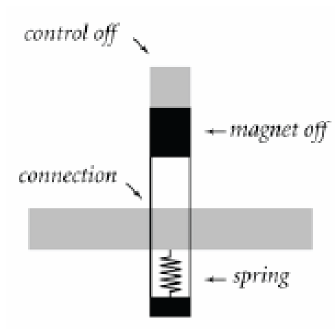
- It is not only about assembly but also about “computer organization”.
- It will cover
  - Basic concept of computer systems and architecture
  - ARM architecture and assembly language
  - x86 architecture and assembly language

TOY machine



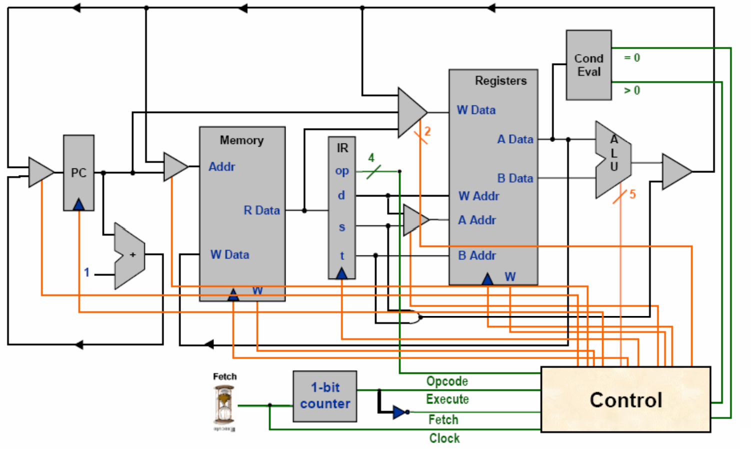
TOY machine

- Starting from a simple construct



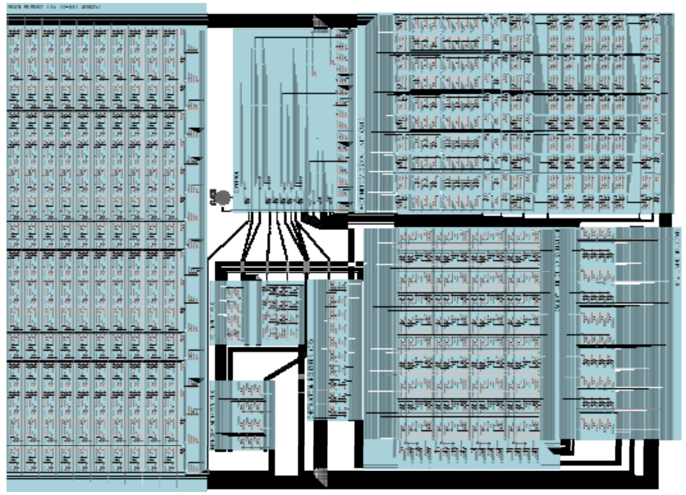
TOY machine

- Build several components and connect them together



TOY machine

- Almost as good as any computers



TOY machine

int A[32];	A	DUP	32	10: C020
		lda	R1, 1	20: 7101
		lda	RA, A	21: 7A00
i=0;		lda	RC, 0	22: 7C00
Do {				
RD=stdin;	read	ld	RD, 0xFF	23: 8DFF
if (RD==0) break;		bz	RD, exit	24: CD29
		add	R2, RA, RC	25: 12AC
A[i]=RD;		sti	RD, R2	26: BD02
i=i+1;		add	RC, RC, R1	27: 1CC1
} while (1);		bz	R0, read	28: C023
printr();	exit	jl	RF, printr	29: FF2B
		hlt		2A: 0000

ARM

- ARM architecture
- ARM assembly programming

IA32

- IA-32 Processor Architecture
- Data Transfers, Addressing, and Arithmetic
- Procedures
- Conditional Processing
- Integer Arithmetic
- Advanced Procedures
- Strings and Arrays
- High-Level Language Interface
- Real Arithmetic (FPU)
- SIMD
- Code Optimization

What you will learn

- Basic principle of computer architecture
- How your computer works
- How your C programs work
- Assembly basics
- ARM assembly programming
- IA-32 assembly programming
- Specific components, FPU/MMX
- Code optimization
- Interface between assembly to high-level language

Why taking this course?

- Does anyone really program in assembly nowadays?
- Yes, at times, you do need to write assembly code.
- It is foundation for computer architecture and compilers. It is related to electronics, logic design and operating system.

CSIE courses

- Hardware: electronics, digital system, architecture
- Software: operating system, compiler

wikipedia

- Today, assembly language is used primarily for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues. Typical uses are device drivers, low-level embedded systems, and real-time systems.

Reasons for not using assembly

- Development time: it takes much longer to develop in assembly. Harder to debug, no type checking, side effects…
- Maintainability: unstructured, dirty tricks
- Portability: platform-dependent

Reasons for using assembly

- Educational reasons: to understand how CPUs and compilers work. Better understanding to efficiency issues of various constructs.
- Developing compilers, debuggers and other development tools.
- Hardware drivers and system code
- Embedded systems
- Developing libraries.
- Accessing instructions that are not available through high-level languages.
- Optimizing for speed or space

To sum up

- It is all about lack of smart compilers

- Faster code, compiler is not good enough
- Smaller code , compiler is not good enough, e.g. mobile devices, embedded devices, also Smaller code → better cache performance → faster code
- Unusual architecture , there isn't even a compiler or compiler quality is bad, eg GPU, DSP chips, even MMX.

Overview

- Virtual Machine Concept
- Data Representation
- Boolean Operations

Translating languages

English: Display the sum of A times B plus C.

C++:

```
cout << (A * B + C);
```

Assembly Language:

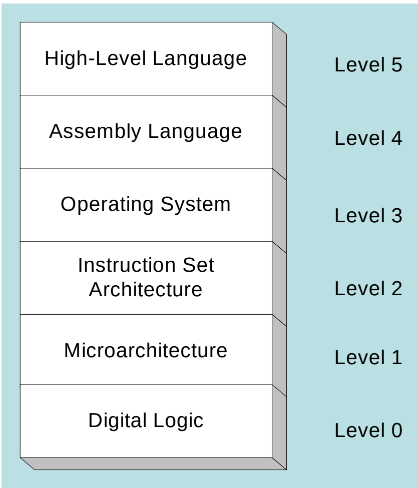
```
mov eax,A
mul B
add eax,C
call WriteInt
```

Intel Machine Language:

```
A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

Virtual machines

Abstractions for computers



High-level language

- Level 5
- Application-oriented languages
- Programs compile into assembly language (Level 4)

```
cout << (A * B + C);
```

Assembly language

- Level 4
- Instruction mnemonics that have a one-to-one correspondence to machine language
- Calls functions written at the operating system level (Level 3)
- Programs are translated into machine language (Level 2)

```
mov  eax, A
mul  B
add  eax, C
call WriteInt
```

Operating system

- Level 3
- Provides services
- Programs translated and run at the instruction set architecture level (Level 2)

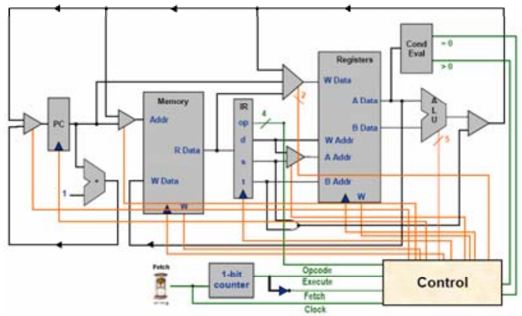
Instruction set architecture

- Level 2
- Also known as conventional machine language
- Executed by Level 1 program (microarchitecture, Level 1)

```
A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

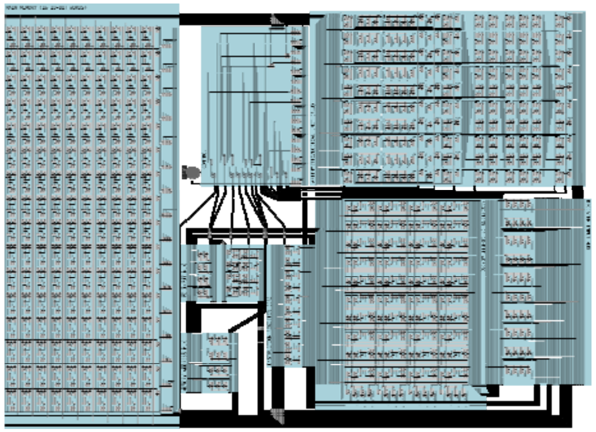
Microarchitecture

- Level 1
- Interprets conventional machine instructions (Level 2)
- Executed by digital hardware (Level 0)



Digital logic

- Level 0
- CPU, constructed from digital logic gates
- System bus
- Memory

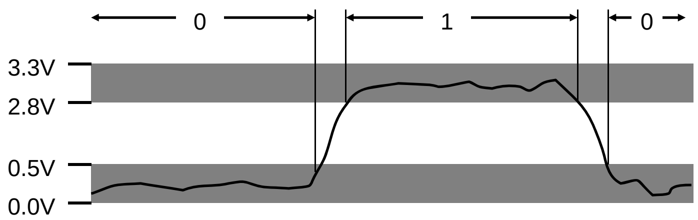


Data representation

- Computer is a construction of digital circuits with two states: *on* and *off*
- You need to have the ability to translate between different representations to examine the content of the machine
- Common number systems: binary, octal, decimal and hexadecimal

Binary representations

- Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Binary numbers

- Digits are 1 and 0
  (a binary digit is called a bit)
  1 = true
  0 = false
- MSB –most significant bit
- LSB –least significant bit
- Bit numbering:

MSB LSB
1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 0
15 0

- A bit string could have different interpretations

Unsigned binary integers

- Each digit (bit) is either 1 or 0
- Each bit represents a power of 2:

1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Every binary number is a sum of powers of 2

Table 1-3 Binary Bit Position Values.

2^n	Decimal Value	2^n	Decimal Value
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

Translating binary to decimal

Weighted positional notation shows how to calculate the decimal value of each binary bit:

$$dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + ... + (D_1 \times 2^1) + (D_0 \times 2^0)$$

D = binary digit

binary 00001001 = decimal 9:

$(1 \times 2^3) + (1 \times 2^0) = 9$

Translating unsigned decimal to binary

- Repeatedly divide the decimal integer by 2. Each remainder is a binary digit in the translated value:

Division	Quotient	Remainder
37 / 2	18	1
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1

37 = 100101

Binary addition

- Starting with the LSB, add each pair of digits, include the carry if present.

carry: 1

	0	0	0	0	0	1	0	0	(4)
+	0	0	0	0	0	1	1	1	(7)
	0	0	0	0	1	0	1	1	(11)
bit position:	7	6	5	4	3	2	1	0	

Integer storage sizes

Standard sizes:

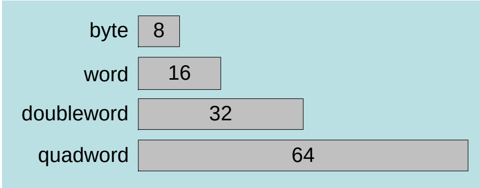


Table 1-4 Ranges of Unsigned Integers.

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

Practice: What is the largest unsigned integer that may be stored in 20 bits?

Large measurements

- Kilobyte (KB), 2^{10} bytes
- Megabyte (MB), 2^{20} bytes
- Gigabyte (GB), 2^{30} bytes
- Terabyte (TB), 2^{40} bytes
- Petabyte
- Exabyte
- Zettabyte
- Yottabyte

Hexadecimal integers

All values in memory are stored in binary. Because long binary numbers are hard to read, we use hexadecimal representation.

Table 1-5 Binary, Decimal, and Hexadecimal Equivalents.

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

Translating binary to hexadecimal

- Each hexadecimal digit corresponds to 4 binary bits.
- Example: Translate the binary integer 000101101010011110010100 to hexadecimal:

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

Converting hexadecimal to decimal

- Multiply each digit by its corresponding power of 16:

  dec = $(D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$

- Hex 1234 equals $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$, or decimal 4,660.

- Hex 3BA4 equals $(3 \times 16^3) + (11 * 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal 15,268.

Powers of 16

Used when calculating hexadecimal values up to 8 digits long:

16^n	Decimal Value	16^n	Decimal Value
16^0	1	16^4	65,536
16^1	16	16^5	1,048,576
16^2	256	16^6	16,777,216
16^3	4096	16^7	268,435,456

Converting decimal to hexadecimal

Division	Quotient	Remainder
422 / 16	26	6
26 / 16	1	A
1 / 16	0	1

decimal 422 = 1A6 hexadecimal

Hexadecimal addition

Divide the sum of two digits by the number base (16). The quotient becomes the carry value, and the remainder is the sum digit.

		1	**1**
36	28	28	6A
42	45	58	4B
78	6D	80	B5

Important skill: Programmers frequently add and subtract the addresses of variables and instructions.

Hexadecimal subtraction

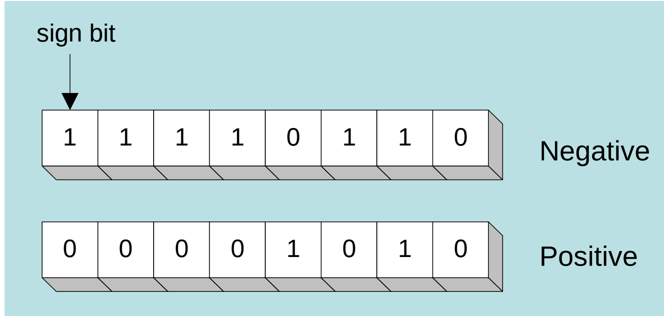
When a borrow is required from the digit to the left, add 10h to the current digit's value:

```
     -1
C6   75
A2   47
-------
24   2E
```

Practice: The address of **var1** is 00400020. The address of the next variable after var1 is 0040006A. How many bytes are used by var1?

Signed integers

The highest bit indicates the sign. 1 = negative, 0 = positive



If the highest digit of a hexadecmal integer is > 7, the value is negative. Examples: 8A, C5, A2, 9D

Two's complement notation

Steps:

- Complement (reverse) each bit
- Add 1

Starting value	00000001
Step 1: reverse the bits	11111110
Step 2: add 1 to the value from Step 1	11111110 +00000001
Sum: two's complement representation	11111111

Note that 00000001 + 11111111 = 00000000

Binary subtraction

- When subtracting A – B, convert B to its two's complement
- Add A to (–B)

```
  0 1 0 1 0  ------>   0 1 0 1 0
– 0 1 0 1 1            1 0 1 0 0
                       ---------
                       1 1 1 1 1
```

Advantages for 2's complement:

- No two 0's
- Sign bit
- Remove the need for separate circuits for add and sub

Ranges of signed integers

The highest bit is reserved for the sign. This limits the range:

Storage Type	Range (low–high)	Powers of 2
Signed byte	−128 to +127	-2^7 to $(2^7 - 1)$
Signed word	−32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	−2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Character

- Character sets
  - Standard ASCII (0 – 127)
  - Extended ASCII (0 – 255)
  - ANSI (0 – 255)
  - Unicode (0 – 65,535)
- Null-terminated String
  - Array of characters followed by a *null byte*
- Using the ASCII table
  - back inside cover of book

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
  - Same for NT and for Linux
  - NT / Linux not fully binary compatible

Alpha sum	Sun sum	PC sum
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 machines use totally different instructions and encodings

Boolean algebra

- Boolean expressions created from:
  - NOT, AND, OR

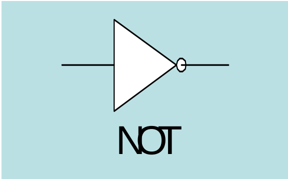
Expression	Description
$\neg X$	NOT X
$X \wedge Y$	X AND Y
$X \vee Y$	X OR Y
$\neg X \vee Y$	(NOT X) OR Y
$\neg(X \wedge Y)$	NOT (X AND Y)
$X \wedge \neg Y$	X AND (NOT Y)

NOT

- Inverts (reverses) a boolean value
- Truth table for Boolean NOT operator:

X	¬X
F	T
T	F

Digital gate diagram for NOT:

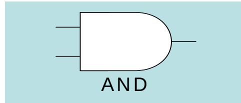


AND

- Truth if both are true
- Truth table for Boolean AND operator:

X	Y	X ∧ Y
F	F	F
F	T	F
T	F	F
T	T	T

Digital gate diagram for AND:

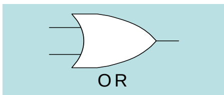


OR

- True if either is true
- Truth table for Boolean OR operator:

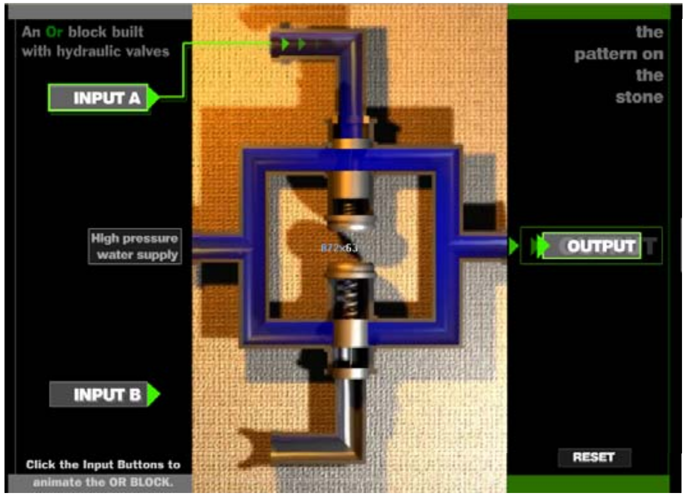
X	Y	X ∨ Y
F	F	F
F	T	T
T	F	T
T	T	T

Digital gate diagram for OR:

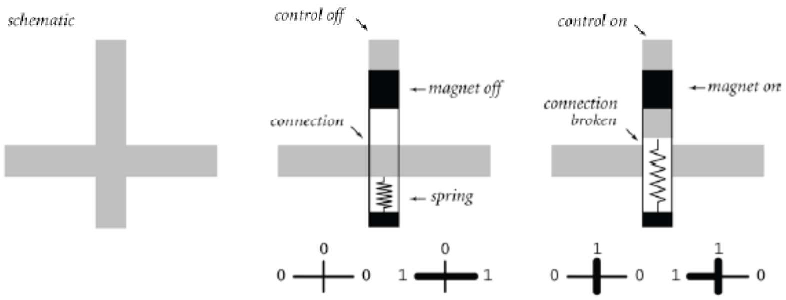


Implementation of gates

- Fluid switch (http://www.cs.princeton.edu/introcs/lectures/fluid-computer.swf)

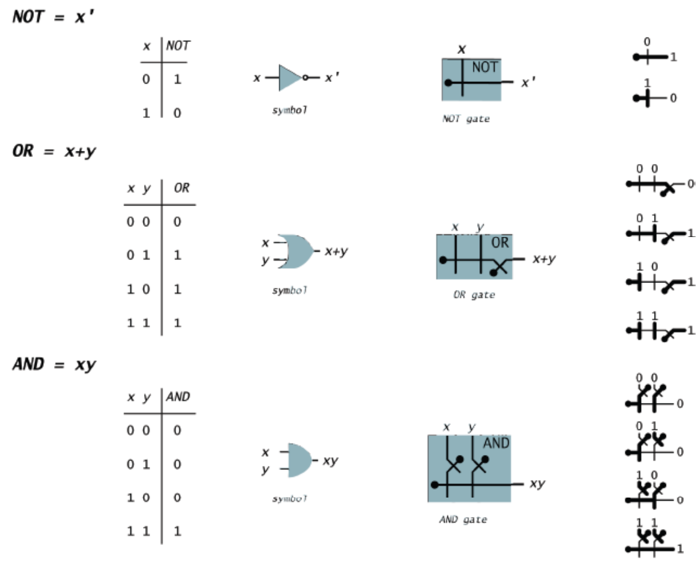


Implementation of gates



Anatomy of a relay (controlled switch)

Implementation of gates



Truth Tables (1 of 2)

- A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
- A truth table shows all the inputs and outputs of a Boolean function

Example: ¬X ∨ Y

X	¬X	Y	¬X ∨ Y
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

Truth Tables (2 of 2)

- Example: X ∧ ¬Y

X	Y	¬Y	X ∧ ¬Y
F	F	T	F
F	T	F	F
T	F	T	T
T	T	F	F