CPT 101 Ultimate Notes

Lecture 1

Types of Computers
- 60s Mainframe computers
- 70s Super computers
- 80s large diversity o new computers
  - Workstations — for graphic designer, for chip designer, much smaller than mainframe and super computer fit on desktop, much cheaper price too
  - Microcomputer — for engineer
  - Personal computer — IBM
  - Microcontrollers — embedded or dedicated computers, calculator & automobiles
  - Servers — networking, web, cloud, allowed resources and storages sharing, applications stored on server and shared by network users
- Trend == increasing performance and down sizing of computers
- Future? — Brain computer interfaces

Computer Generations
- First: 1944 — 1958
  - Built from vacuum tubes
  - Mark I, ENIAC, UNIVAC(first commercial computer)
- Second 1959 — 1963
  - Built from transistors (much smaller)
  - IBM 1401, IBM 1410 with 1402 card read/punch
- Third generation (1964 — 1970)
  - Build from integrated circuits (IC)
  - DEC PDP-1(used to be second largest vendor, acquired by HP), IBM 360(one of the most successful mainframe, OS is also developed on this mainframe, this is an milestone of OS study.)
- Fourth generation (1971 to now (VLS())
  - Dominated by LSI(large scale integrated circuits) and VLSI(very-large-scale integrated circuits)
  - Applications includes word processing, spreadsheets, databases, and graphic programs became readily available
  - Cray-1 supercomputers(built with VLSI)
  - Apple II, IBM PC, Notebook PC, Palmtom , iPhone etc etc

Hardware & Software
- When hardware details is ignored will cause:
  - Unsuitable equipment are ordered for programmers
  - Unsuitable software fails to exploit the performance advantages offered by revolutionary new circuits.

Computer hardware
- 5 categories
  - Input
  - Processing (CPUs, ALU)
  - Output (printer)
  - storage (harddisk)
  - Communication (networking chips)

Computer software
- System software
  - Communication with hardware
  - Resource management
  - Facilitates communication among application programs
- Application software
  - Benefits/assists the user

Backward (downward) compatibility for new hardware

Most software written for computers with old hardware can be run on computers with newer hardware

Why? Preserve expenditure, so no double spending on new hardware and software, wasting time and money.

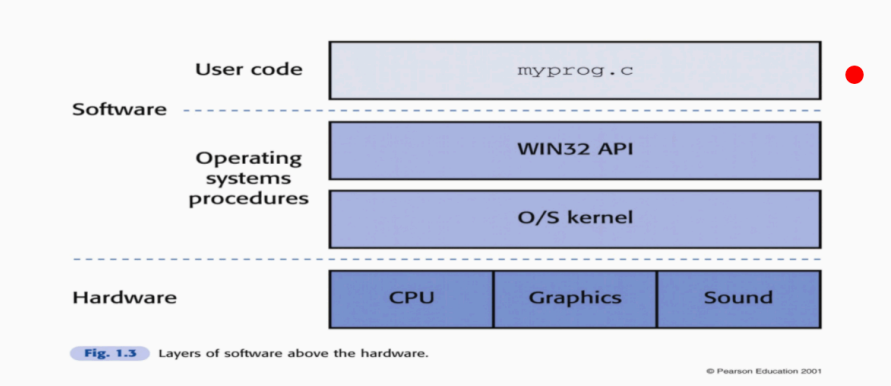
VHDL

very high speed integrated circuits hardware description language

Programming language used to specify structure and function of hardware circuits

Supports computer simulation as well as providing input to automatically layout packages which arranges the final circuits.

Computer systems hierarchy



Program must be supported by the operating system

Program calls API (application programming interface) to access system resources

Kernel — scheduling task for cpu to execute (for fair and efficient processing), security — so program don't damage each another(memory etc),

What are the benefit for a system to be designed in hierarchy?:

Easier for us to understand, multiple layers of application

Easier to design

Easier to change, with layer structure its easier to change

Advantages of OS

Easier to use

Easier programming

Protect system resources, so no user process touch on other users, protect application from each another, fair and efficient distribution of system resources

Moore's law

The amount of circuitry(# of transistors) which can be placed on a given chip area approximately doubles every two years.

Interaction between hardware and software

WIMP can help — windowing interfacing (also called GUI)

A by product of microprocessor revolution that allowed fast bitmapped graphics.

Improves user productivity.

The internet — connecting all networks

Web browsers boost the use of internet greatly

Trends of computing

Scientific computing — [computation]

Business computing — [data], commercial, bank
Personal computing — [interaction] (point and click, ease)
Pervasive computing — [ubiquity] Access computing resources everywhere
Mobile computing — [mobility] move along and still stay connected etc.
What is next?

Q&A Lecture 1

1. Mention four architectural level of computer systems:
   1:= assembly programming
   2:= instruction set CPU
   3:= micro architecture level
   4:= digital logic layer (gate, logic devices)
2. What does CPU stand for? ALU?
   Central processing unit
   Arithmetic logic unit
3. What are the components of a computer system?
   CPU, memory, IO, bus
4. Mention 4 different types of computers:
   Workstations
   Server
   Personal computer
   Super computer
5. 'Servers' are facilitated via the availability of?
   Inter-networking interconnection
6. Define the ares of study for 'computer systems'.
   To study interaction between various component
   To study interaction between hardware and software so that performance can be delivered and determined.
7. Define 'Downward Compatibility'
   Software written for old hardware can continue to be run on new hardwares
8. What is the description language that can be used to design high speed IC hardware
   VHDL
9. What are the advantages of having a hierarchical approach to computer system?
   Easy to understand
   Easy to design
   East to change
10. What is Moore's Law?
    The number of circuitry(transistors) that can be placed on a fixed chips area is typically doubled every two years
11. Functionalities of hardware systems can be brought out by what and thus offered to the user?
    Operating system therefore offered to the user
12. What are the advantages of having 'operating systems' wrapping around the computer hardware?
    Easier for programmer to use the system
    Provide protection among system resources and its user
    Provide efficient and fair access to computer resources
13. WIMP stand for? WIMP is available due to the development of?
    WIMP := Windows, Icons, Menus, Pointing device
    Is available due to the development graphic devices/display and interaction devices (such as mouse/joystick)
14. What is the focus of scientific computing?
    Computation
15. What is the focus of business computing?
    Data
16. What is the major characteristic of personal computing?
    Interaction

Lecture 2

Input-Process-Output-Model

Input data → Process → Output data

A highly conceptualized computer model
The input-process-output model is the fundamental structure of the current generation of digital computer
The process is to be controlled by a special custom-made program
This was an essential scheme of the von Neumann model(more detailed)

Components of the Computer System

There are three components required for the implementation of Input-Process-Output and von Neumann model(s):

Hardware — everything that can be seen inside the computer system (cpu, keyboard, system memory, hard disk, etc)
Software — collection of program, designed to achieve some task
Data — data entered for input and to be manipulated

Hardware

The most visible part of the computer system is the hardware
— CPU, memory, hard disk, keyboard, display screen, …
It is physical — you can touch it.
CPU == Central processing unit
is an active part which performs calculations and other operations
The main memory (primary storage or working storage),
or RAM (for random access memory) holds data and programs for access by CPU
Memory(Ram) is volatile, when system is shut down data is gone; Rom is not volatile (to store drivers BIOS software etc.)
The secondary storage.
Long term storage
Holds programs and data
Hard disk, CDs, DVDs, etc
Input devices
Keyboard, mouse, scanner, etc
Output devices
Monitor, speaker, printer, etc.

Software

The hardware of a computer (e.g. CPU) can carry out only very simple operations like adding numbers (very quickly)
To make it perform useful tasks, these simple steps are combined in the form of programs, which are collectively known as software.

Machine instructions

The CPU performs the execution of machine instructions.
Every CPU has its own instruction set (100-200 instructions, typically)
— for a particular machine, this set is fixed.
Although the instruction sets of different CPUs are similar, there is no standard instruction set.

Machine instructions categories

Input-output:
IN, OUT (Intel x86 and Pentium, but does not exist in some CPUs)
Data transfer and manipulations:
MOV, ADD, MUL, AND, OR
Transfer of program control:
JMP, JC
Machine control:
can halt processing, reset the hardware, INT, HLT

Compare and contrast Machine instructions and HLL

High Level Programming Languages (HLLs)

More suitable for programming than the languages of machine instructions.

The programs in HLL still have to be translated to machine codes.

Von Neumann

John von Neumann

Was an Austria-Hungary-born American mathematician

Made contributions to: quantum physics, functional analysis, set theory, topology, economics, computer science, numerical analysis, hydrodynamics.

The von Neumann Model

The idea was formulated by von Neumann (late 1940s)

The computer is general-purpose machine controlled by an executable program:

In this context:

A program is a list of instruction used to direct task.

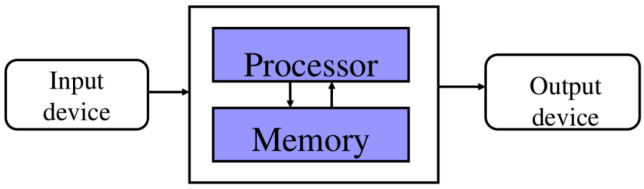
Both program and data ra held in computer's memory (store) and both represented by binary codes.

The fact that memory is re-writable makes a von Neumann machine especially powerful

A process is an active part of the machine that executes the program instructions

How does the specification make possible that machine being 'general-purpose'?

Rewritable memory



Input device is for transmitting information from a user into the computer's memory

Output device enables a user to see results of the program being performed/

Von Neumann bottleneck (between CPU and memory)

— CPU is continuously forced to wait for vital data (and instructions) to be transferred to or from memory (typically due to the difference in speed of cpu processing and memory transfer and retrieval

Potential problems:

How could computers distinguish data from instructions since they are both represented by binary codes?

A 16 bit instruction code could, in different circumstances, represent a number or two characters.

Solution

Data and instructions are stored in memory (the computer know where data and instruction is stored separately in the memory)

CPU knows were to fetch program instructions

CPU executes the program instruction

Instructions and data have to be in a special coded form in order to be understood by CPU

J, von Neumann

The term "von Neumann architecture" appeared from his paper *First Draft of a Report on the EDVAC* dated June, 30, 1945.

Konrad Zuse

Mentioned the concept in a patent application 1936

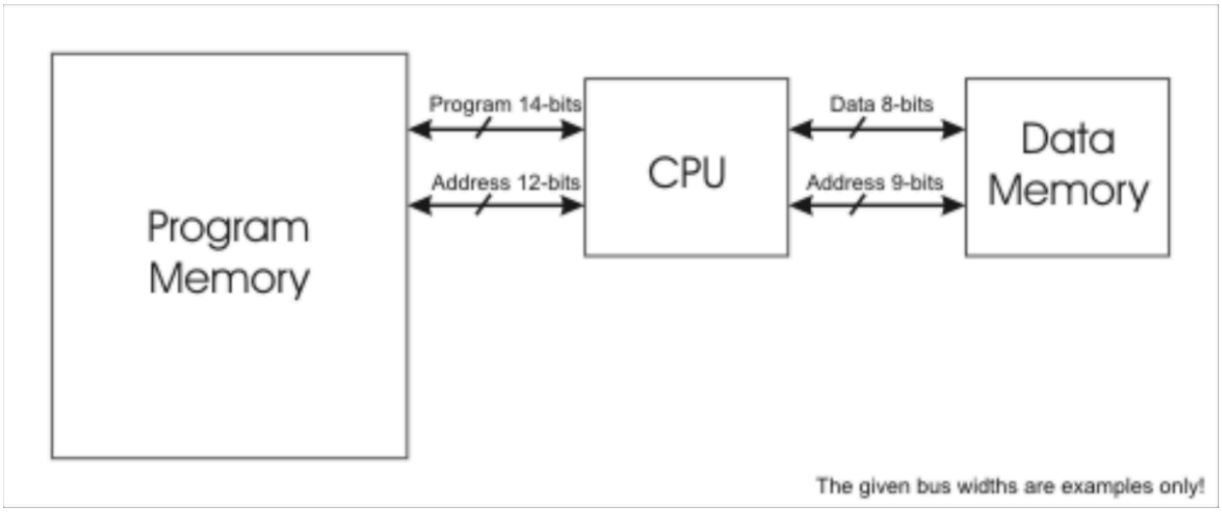
J.W. Mauchly and J.P. Eckert. (UPenn)

Wrote about the stored-program concept in December 1943 during their work on ENIAC — a general purpose stored-program computing machine.

Von Neumann's specifications has remained sound for more than 60 years and is implemented in almost all computers today.

A variation, “Harvard architecture”, has gained ground recently.
Separates data storage from programs storage
Requires different memories and access buses for program data.
The intention is to increase transfer rates, to over come the bottle neck.

Harvard architecture



CPU in the center, has two connections to program and data memory respectively.
Can have concurrent transfer (having program memory transfer while having data memory transfer)

Who Is the winner?
VonNeumann Architecture has “won” — most memory (hard drives and RAM) can hold data as well as instructions simultaneously
From another point of view the Harvard Architecture is still going strong — most desktop CPU have an internal “instruction cache” feeding the control unit and a completely separate data cache

Q&A Lecture 2
1. Using a diagram to illustrate the concept of Input-Process-Output Model?
   Input data → Process → Output data
2. Which model gives the fundamental structure of the current generation of digital computers?
   Input-Process-Output Model
3. Process is controlled by what?
   By user’s program/custom made program/
4. Highlight the three components required for the implementation of Input-Process-Output and Von Neumann model.
   Hardware, software, data
5. Name 5 examples of computer hardware
   CPU, memory — RAM, bus, input device — keyboard, output device — display
6. Identify the active part within a computer which performs calculations and other operations
   CPU
7. Which part of a computer holds data and programs for access by CPU
   System memory, primary memory, main memory
8. Give 3 examples of secondary storage
   Hard disk, CD, DVD
9. Give 3 examples of input devices
   Keyboard, mouse, microphone
10. Give 3 examples of output devices
    Printer, display, speaker,
11. Define ‘software’
    Software is a collection of program, and program consists of instruction to solve a specific problem

12. Difference between these two models — Input-Process-Output model & von Neumann model?
    Input-Process-Output model represents the external view of computer system, it prescribes a function performed by a computer system. But the von Neumann model is more inner detailed, it gave specification describes how a computer consist of CPU working together with the memory to hold program and data, von Neumann model describes the computer system from a more engineer-oriented point of view.
13. For a particular machine, the machine instruction set is usually fixed. True or false?
    True.
14. There exists a standard instruction set for industry purpose. True or false?
    False.
15. High Level Programming Languages (HLLs) are more suitable for programming than the languages of machine instructions. Why?
    True because its more user friendly and provides high level programming constructs closer to human understanding and application need, thus more suitable for programming.
16. Mention 4 major categories of machine instructions.
    1 := input-output
    2 := data transfer and manipulation
    3 := instruction for transfer of program control
    4 := hardware/machine control
17. Why HLL programs need to be translated before execution?
    Because the CPU can only understand machine code.
18. What is the von Neumann model?
    Von Neumann machine is a general purpose machine where it has a CPU being controlled by a program, with a rewritable memory storing programs and data inside.
19. Both [Program] and [Data] are held in computer's memory (store) and both represented by?
    Programs and Data are represented Binary in computer's memory.
20. Identify the von Neumann bottleneck?
    The connection between CPU and main memory could have speed mismatch, CPU need to wait for memory transfer to complete before the CPU can execute next instruction
21. How could computers distinguish data from instructions since they are both represented by binary codes?
    Data and instructions are stored on different parts of system memory.
22. What is the main difference between von Neumann machine and Harvard architecture?
    Harvard id different from von Neumann by having data memory separated from instruction memory, so that the CPU can concurrently retrieve and transfer data and instructions to both memories, while von Neumann relies on an unified system memory that stores both data and instructions simultaneously.
23. Motivate the use of Harvard architecture.
    Harvard architecture circumvents the von Neumann bottleneck, increased data transfer rate.
24. What is the additional cost form the usage of Harvard architecture?
    Cost from dual connection and cost form two memory banks for data and instructions respectively.
25. Most desktop CPUs have an internal "instruction cache" feeding the control unit and a completely separate "data cache" this mimics which computer architecture?
    Harvard architecture

Lecture 3

Machine instructions and HLL

In the previous lecture we have seen:

HLLs are more suitable for programming than the languages of machine instructions

Examples:

FORTAN — for scientific computation
COBOL — for commercial/business computation
C — for systems programming
C++, Java, Perl, etc.

Programs in HLL still have to be translated to the machine codes.

Semantic Gap

The term expressed the enormous difference between the way human languages expressing ideas and actions and the way computer instructions representing data processing activities.

Translation

Translation is done by special programs such as:

Compilers,

Translating HLL instructions into machine code (sequences of instructions) before the code can be run on the machine.

Assemblers,

Translating mnemonic form of machine instructions (like MOV, ADD, etc) into their binary codes.

Interpreters,

Translating HLL instructions into machine code on-the-fly (while the program is running).

Translation — compilers and assemblers:

Edit → *HLL source file* → **Compile** → *Binary object files* & *Library files* → **Link** → *Executable file* → **Load** → Run

What to do with Compile-time errors?

Check for syntax error, edit and recompile

Linking resolving external references

If partitioned program into separate modules or using system library the binary files will need to be linked by a Linker

If there's errors in linking such as making use of undefined variables in other modules then linker will not be able to resolve such external references.

What to do with Link-time errors?

Check external references, edit and recompile

What to do with Load-time errors?

Check for logical errors, edit and recompile

Or code size is too huge for memory to load.

Linking

Big programs usually are divided into several separate parts or modules.

Each module has to be designed, coded, and compiled.

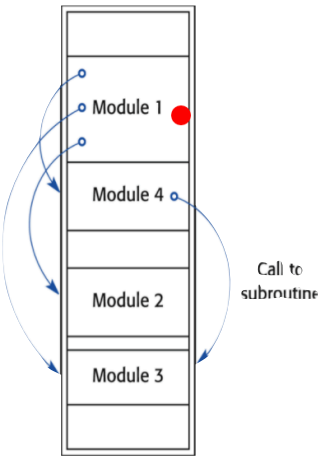
There are frequent occasions when code in one module needs to reference data or subroutines in another module.

A compiler can translate a module into binary codes, but it cannot resolve those references to other modules/

Those *external references* remain symbolic after the compilation, until the linker gets to work.

The linker is to join together all the binary parts.

The linker will report errors if it cannot find the module or code referred to by those external references



Library files

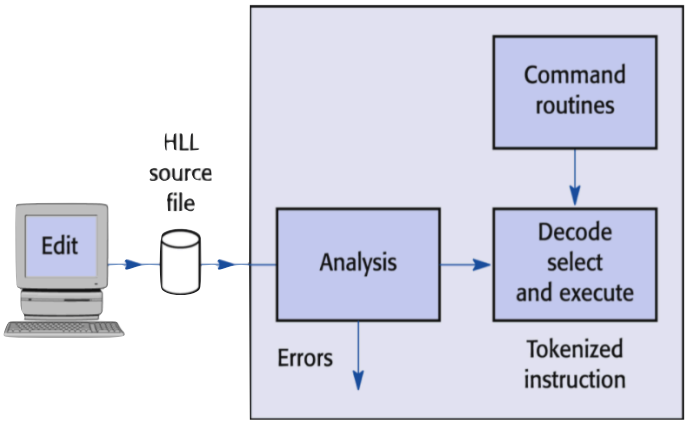
Existing defined code to simplify and reduce programmer's implementation effort.
Translated object code
Provide many functions for programmers, but are only usable if linked into your code.
In Unix:

Directories /lib and /usr/lib/.

In Windows:

DLL files.

Interpreters — alternative way of running HLL programs



Such as used with BASIC and Java
Instructions are converted into an intermediate form consisting of tokens.

In Java, tokens such as: static, boolean, file, String, void, return

Tokens are then passed to the decoder which selects appropriate routines for execution.

Compilers Vs Interpreters

Compilers:

Take a program and translate it as a while into machine code.
The process of translation and execution are separate.

Interpreters:

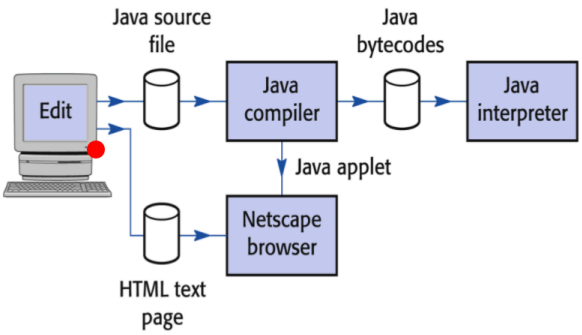
Take an instruction, one at a time, translate and execute it.
The process of translation and execution are interlaced.

C program compilation, linking & execution

C source code→compiler (program)→assembly language→assembler→machine code
Once we have machine code:
machine code→linking and loading (program)→program code execution (program)

Java: compilers and interpreters

Java source code→compiler (program)→Java "byte codes"→Java interpreter (program)



Compilers Vs Interpreters 2

Execution of compiled code is much faster than execution of interpreted code.

However this disadvantage is not a big problem for most applications

Interpreters are more suitable for rapid prototyping and for other situations when a program is frequently modified.

If writing program under tight deadline, then interpreter will be better, because interpreter analyze and execute one instruction at a time so it gives you sufficient program and process context for debugging purpose, to produce more accurate in terms of error reporting because you have access to the current process context.

Interpretation can provide uniform execution environment across several diverse computers (Portable)

Interpreters as Virtual Machines

Interpreters are somewhat similar to the computer hardware (CPU)

Both takes/fetches one instruction at a time and executes it

Because of that sometimes they are referred to as a virtual machine

Example: JVM — the Java Virtual Machine

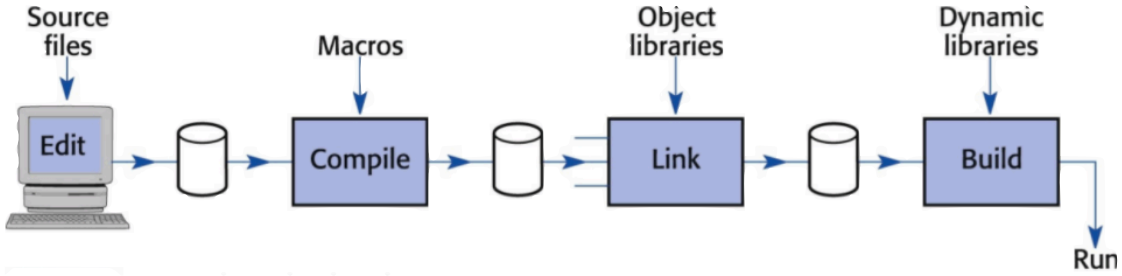
Coding sharing and reuse

The problem

How to reuse existing proven software when developing new systems?

Three solutions:

1. Source-level subroutines and macro libraries
   Sharing software in the source code level (#include, import, etc.)
2. Pre-translated re-locatable binary libraries
   Using existing binaries
3. Dynamic libraries and dynamic linking
   DLLs, Microsoft ActiveX



Source-level subroutines and macro libraries

Intention

Take copies of the library routines // using macros written by others

Edit them into your new code

Translate the whole together // save effort

Disadvantages

Who owned the code? // if you made some small changes who owns it?

Who should maintain it? // its unsure who's code is have responsibility

Pre-translated relocatable binary libraries

Intention

Libraries(commonly used code) are pre-translated into relocatable binary code.

Can be linked into your new code, but not altered(since its pre-translated).

Acceptance

Successful, and still essential for all software developments undertaken today

Disadvantage

Each program is to have a private copy of the subroutines, wasting valuable memory space, and swapping time, in a multitasking system

'Relocatable'

Even if the library code is shared across different programs in different addresses it must be producing the same results.

Dynamic libraries and dynamic linking

Intention

Load a program which uses “public” routines already loaded into memory. The memory-resident libraries are mapped through the memory management system to control access and avoid multiple code copies.

Allowing one single copy of the code to exist in the memory and use a memory management system to allow different programs to use its routines.

Mapped into process space, even if different programs is using the same DLL codes they will be running in different memory address.

Acceptance

Successful through Microsoft’s ActiveX standard.

Q&A Lecture 3

1. Name 4 examples of HLLs
   FORTRAN, COBOL, C, Java
2. Translation fills in the semantic gap in computer systems. (True or false?)
   Java
3. Name 3 different ways of translation. Identify the crucial role of translation under each.
   Assemble — translating assembly code into machine instructions
   Compile — translating HLLs into assembly
   Interpret — translating Instructions into an intermediate form consisting of tokens to be decoded
4. When compile-time errors occur, what do we do?
   Check syntax errors, edit and change the program, then recompile
5. What are the purposes of linking?
   To resolve external references, and to allow the program to be written in multiple modules in separate files
6. ‘Loading’ is carried out before ‘linking’ after a program is compiled. (True or false?)
   False, loading is carried out after compilation and link.
7. Program modules can be compiled separately (True or false?)
   True.
8. A compiler can translate a module into binary codes, but it cannot resolve those references to other modules. This occurs when …?
   When programs has been developed in multiple separate modules files.
9. Library files are usable if linked into your program code. (True or false?)
   True.
10. Interpreters typically convert program code into what?
    Intermediate code.
11. What is the output of program compilation?
    Binary code, or machine code.
12. Name 2 scenarios where interpreters are more useful than compilers.
    Rapid prototyping, or making frequent change to the program code.
13. Interpreters are sometimes called as virtual machine because … ?
    Because each instruction is translated and executed one by one resembling CPU.
14. Mention 3 approaches to code sharing.
    Macro libraries
    Precompiled relocatable libraries
    Dynamic libraries, dynamic library linking
15. Name 2 disadvantages (or issues) of macro libraries.
    Ambiguous code ownership and responsibility for maintenance.
16. Libraries can be linked into your program code, but not altered. (True or false?)
    True.
17. What are the disadvantages of program execution with pre-translated program library?
    Repetition of the library code in multiple program space, wasting memory.
18. There exists a de-facto standard of dynamic libraries and dynamic linking. (True or false?)
    True.

Lecture 4

Data, Information and Knowledge

Data — raw facts, figures, measurements, …

1.00001, 1.0000010, 2.0000101, 3.0000102, 5.000…

Information

Data organized into useful representation

1.000, 1.000, 2.000, 3.000, 5.000 (rounded off in consistent format)

Knowledge

Application of reasoned analysis of information

'Data are in increasing order', 'data can be derived based of Fibonacci sequencing', etc

Measurement of Information

In 1948, motivated by the problem of efficiently transmitting information over a noisy communication channel, Claude Shannon introduced a revolutionary new probabilistic way of thinking about communication and simultaneously created the first truly mathematical theory of **entropy**.

His ideas created two main lines of development:

Information theory, and coding theory.

Entropy of an event X is related to $\mathrm{p(X)}$

Information entropy

Which one carries more information?

'There is a terror attack scheduled today at 11 pm in LA.' ✓

'Today's weather is fine.' ✗

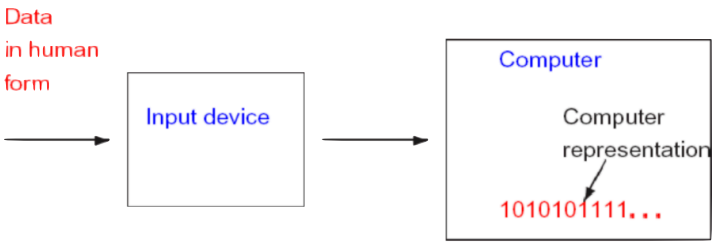
Data codes — numeric and character

Within a computer the binary number system is a system of choice, both for data storage and internal processing of operations.

Based on the two-state, ON/OFF technology.

We as human beings don't choose normally to work with binary form. We use language, images, sounds … numbers usually represented in decimal number system.

Thus, one need to convert words, numbers, images, sounds into a binary form and back.



Alpha numeric data

The majority of the data originally comes in the form of letters in alphabet, numbers and punctuation (**alphanumeric data**).

They are represented in computers by binary numbers.

Bit

A bit is the most basic unit of information possible; it contains the information necessary to distinguish two alternatives (1 or 0, yes or no, etc.)

We think of these alternative as being numbers 0 and 1.

Two alternatives are represented by two-state elements (memory cells is charged, or not).

Binary number system

A sequence of bits (binary digits) represents a number in the binary notation

Example: the sequence 00010110 represents the number 22:

$$10110_2 = 22_{10}.$$

As in decimal system a binary digit (1 or 0) represents some value depending on where it is written in the number:

$$\begin{array}{cccccccccccc} 1024 & 512 & 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}$$

$$1024 + 256 + 128 + 16 + 2 + 1 = 1427.$$

Therefore, 10110010011_2 is a binary representation of 1427_{10} (decimal)

Base n notation, $n = 10,2,16,\ldots$

Base 10 notation, or decimal. 10 digits are in use.

$$(a_1 a_2 \ldots a_k)_{10} = a_k \times 10^0 + a_{k-1} \times 10^1 + \ldots + a_1 \times 10^{k-1}$$

Base 2 notation , or binary. 2 digits are in use.

$$(a_1 a_2 \ldots a_k)_2 = a_k \times 2^0 + a_{k-1} \times 2^1 + \ldots + a_1 \times 2^{k-1}$$

Conversion of decimal numbers to binary

$$\begin{array}{rcccccccc} 2397_{10} & = & 2_{10} \cdot 1000_{10} & + & 3_{10} \cdot 100_{10} & + & 9_{10} \cdot 10_{10} & + & 7_{10} \cdot 1_{10} \\ & = & 10_2 \cdot 1111101000_2 & + & 11_2 \cdot 1100100_2 & + & 91001_2 \cdot 1010_2 & + & 111_2 \cdot 1_2 \\ & = & 100101011101_2 & & & & & & \end{array}$$

Human friendly method for converting decimal into binary

Repeatedly divide the decimal number by 2

Write down the remainder from each stage (1 or 0) from right to left

Binary vs. Decimal notation

Decimal notation is more compact than binary, generally uses less digit to represent the same number.

Decimal is more convenient for humans.

Binary is more “convenient” for computers due to two-state, ON/OF technology employed.

Hexadecimal notation

Hexadecimal notation uses 16 as a base

Why use hexadecimal?

Convenient shorthand for binary numbers.

Every hexadecimal digit exactly represents 4 binary bits.

Base 16 notation, or decimal. 16 digits are in use:

$$0,1,2,3,4,5,6,7,8,9,\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D},\mathrm{E},\mathrm{F}.$$

$$(a_1 a_2 \ldots a_k)_{16} = a_k \times 16^0 + a_{k-1} \times 16^1 + \ldots + a_1 \times 16^{k-1}$$

Hexadecimal and binary:

Any hexadecimal digit represents some 4-digit pattern (binary number) and vice versa:

$$\begin{array}{ll} 0 = 0000_2 & 8 = 1000_2 \\ 1 = 0001_2 & 9 = 1001_2 \\ 2 = 0010_2 & \mathrm{A} = 1010_2 \\ 3 = 0011_2 & \mathrm{B} = 1011_2 \\ 4 = 0100_2 & \mathrm{C} = 1100_2 \\ 5 = 0101_2 & \mathrm{D} = 1101_2 \\ 6 = 0110_2 & \mathrm{E} = 1110_2 \\ 7 = 0111_2 & \mathrm{F} = 1111_2 \end{array}$$

Translation from binary to hexadecimal form:

Break binary into 4-digit groups, then translate each group into one hexadecimal digit

$$10011110_2 = 1001\ 1110 = 9\mathrm{E}_{16}$$

Q&A Lecture 4

1. Define ‘bit’
   Basic information holder, basic in the sense such that each bit can only be mapped into one of two states (0 or 1), (true or false)
2. Decimal notations is more compact than binary (True or False?).
   True, consumes less digit in code
3. Binary is more “convenient” for computers due to?
   The underlying two state technology being used.
4. Every number can be used as a base (True or False?)
   False, 0 or 1 cannot be used as a base, all other can positive integer can.
5. Hexadecimal uses ? As a base.
   16
6. Octal notation uses ? As a base.
   8
7. Why do we use hexadecimal, or base 16, number notation?
   Closer than human in comparison with binary, more compact than binary and easily converted from binary.
8. How many bits are required to encode ASCII?
   7 bits.

Lecture 5

Alphanumeric characters

Data that computers deal with usually comprise:

Letters, such as ‘A’, ‘b’, etc

Numbers, such a s ‘1’, ‘9’

Other characters such as punctuation

Binary codes of alphanumeric characters

The need for coding

Since all data are represented by binary sequences in computers, the question comes up as how they should be represented.

The choice of code may be arbitrary (up to user to decide)

But, what matters is consistency, i.e. **standards** are needed.

For information to be interchanged standards are needed, without international standard, then no body can understand another user’s encoding and decoding standards unless its published.

Alphanumeric codes

ASCII code (American Standard Code for Information Interchange)

(7bit code) and its extensions (8-bit code). (Well-established)

EBCDIC code (Extended Binary Coded Decimal Interchange Code)

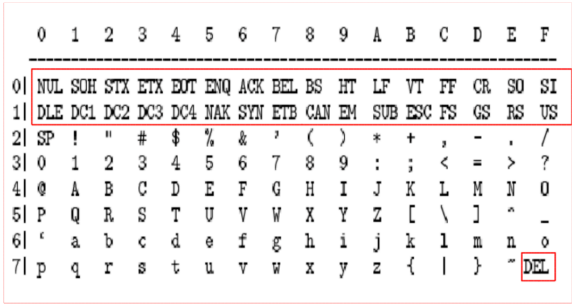
8-bit code. (IBM mainframe computer)

Unicode

Recent 160bit standard (Up to 2^{16} characters can be encoded)

ASCII code table

It is 7-bit code, so the code of any character can be represented as a two-digit hexadecimal number (row’s and column’s numbers are the first and second digits, respectively).



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

3 bits are represented by the first hexadecimal digit, and the next 4-bits are represented by the second hexadecimal digit.

Only half of possible byte (8-bits) patterns is used.

The second half is used in the 8-bit extension(s) to represent the codes of additional symbols, line shapes, foreign letters, etc.

The order of the letter codes is compatible with the alphabetical order of the letters,

The table is divided into two classes of codes:

Printing characters produce output on the screen or on printer

Control characters are used to control position of output on screen or paper, cause some action to occur, or to communicate status with I/O.

NUL	Null	**VT**	Vertical tab	**SYN**	Synchronous idle
SOH	Start of heading	**FF**	Form feed	**ETB**	End of transmission block
STX	Start of text	**CR**	Carriage return	**CAN**	Cancel
ETX	End of text	**SO**	Shift out	**EM**	End of medium
EOT	End of transmission	**SI**	Shift in	**SUB**	Substitute
ENQ	Enquiry	**DLE**	Data link escape	**ESC**	escape
ACK	Acknowledge	**DC1**	Device control 1	**FS**	File separator
BEL	Bell	**DC2**	Device control 2	**GS**	Group separator
BS	Backspace	**DC3**	Device control 3	**RS**	Record separator
TAB	Horizontal tab	**DC4**	Device control 4	**US**	Unit separator
LF	New line	**NAK**	Negative acknowledge	**ASCII CONTROL CHARACTERS**	

Limitation of ASCII code

The limitations of the well-established 8 bit ASCII codes.

Too limited for the display requirements of modern Windows-based word-processors.

The requirement of global software market for handling international character sets.

Unicode

Even 8-bit extensions of ACII code table is capable to code only up to 256 characters.

Unicode Standard (1991) is an 16-bit international encoding system for information interchange.

Code values are available for more than 65000 characters.

Standard includes the European alphabetic scripts, Middle Eastern right-to-left scripts, and scripts of Asia, Africa and America, ideographic characters of China, Japan, etc.

Standard includes punctuation marks, mathematical symbols, geometric shapes, etc.

It is ok to use Unicode in Java programming

To show Chinese characters in a java program unicode must be used.

Representation of numbers

Any representation of numbers capable of supporting arithmetic manipulation has to deal with both integers and real values, positive and negative.

Integers are whole numbers, with no fraction part.

Real numbers extend beyond the decimal point.

Representation of integers

Generally 4 bytes, i.e. 32 bits.

Two's complement as a method of representing and manipulating negative integers.

$$\begin{array}{cccccccccc} -128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 & & \\ (0 & 0 & 0 & 0 & 0 & 1 & 0 & 1)_2 & = & 5_{10} \\ (1 & 0 & 0 & 0 & 0 & 1 & 0 & 1)_2 & = & -5_{10} \end{array}$$

Representation of real numbers

IEEE 754 standard.

The most widely-used standard for floating-point computation.

Defines format for representing floating-point numbers, special values, and a set of floating-point operations that operates on these values.

Declaration of variables in programs

What happens when you declare variables in a program?

You are informing the compiler to reserve the correct amount of **memory space** to hold the variable.

You are also telling the compiler what **encoding**/**decoding**/representation scheme to be used.

```
char  letter;              // 1 byte
short count;               // 2 bytes
long  world_population;    // 4 bytes
long  double world_weight; // 8 bytes
```

Q&A Lecture 5

1. The most widely used binary code with non-IBM mainframes and virtually all microcomputers, is **ASCII**.
   a. EBCDIC
   b. DOS
   c. ASCII
   d. Lan
2. Which of the following coding schemes ins not yet commonly used?
   a. Unicode
   b. EBCDIC
   c. ASCII
   d. All of the above are common coding schemes
3. A draw back to ASCII is that it
   a. Cannot handle all the characters of some languages other than English and European languages.
   b. Uses only 4 bits to form each character
   c. Is slower than EBCDIC
   d. None of the above is correct
4. ASCII is divided into two classes of codes?
   True, printing and control characters.
5. What is the most widely-used standard for floating-point computation
   IEEE 754 standard.
6. Two things happen when you declare variables in a program. What are they?
   Informing compiler to reserve memory to store that variable
   Informing compiler what kind of encoding/decoding scheme use for the type of that variable

Lecture 6

Operating systems: examples

1960s — OS/360 for IBM System/360

Proprietary OS for IBM mainframe

1970s — Unix

K&R designed unix in C, open sourced the source code.

Berkley's distribution is made available (BSD Unix)

1980s — MS-DOS for IBM PC and Mac OS for Apple Macintosh

IBM made public the PC BIOS interface, so MS-DOS can run on IBM compatibles.

1990s — Windows 95, 98, NT

NT served as the basis for Microsoft's desktop operating system line starting in 2001.

2000s — Mac OSX

Linux, BSD Unix…

Operating systems have undergone more than 40 years of evolution, but surprisingly little has changed at the technical core, only the introduction of windowing interfaces really distinguishes.

Functions

Functions of OS — to interpret and carry out commands issued by:

Users of the computer or,

Application programs running on the computer.

Purposes

Two fundamental purposes:

Management:

To control and operate hardware in an efficient way.

Provide functionalities:

To allow the user efficient access to the facilities of the machine.

So system resources won't be wasted.

To allow the user fair access to the facilities of the machine.

So your instructions won't be ignored by the system, anything entered will be processed.

To allow the user protected access to the facilities of the machine.

So your data is safe, and u won't effect other user's data, ensure program integrity.

Interaction with the user.

Onion ring model

To realize the purposes of an operating system, the software system is often arranged in multiple layers

Why multiple layers?

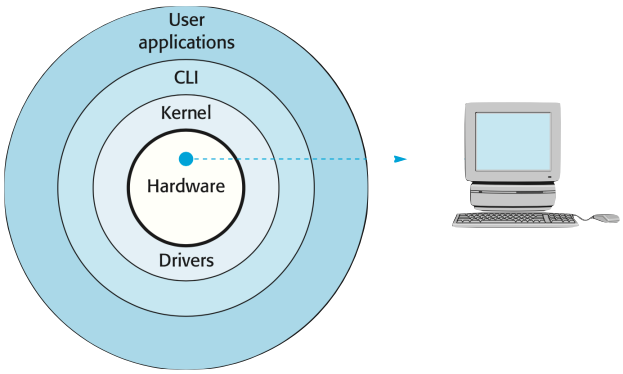
Allow system designer to design easier.

Allow system designer to make changes to the design easier.

Purposes

We use a computer through the operating system **(gatekeeper)**

It is often the operating system that governs the overall efficiency of a computer **(butler)**



Core of operating system： dealing directly with the hardware

Kernel — device driver, memory allocator…

CLI (Command line interpreter): provides user accessibilities to the system.

Modern Operating Systems

Many modern operating systems also:

Allow for the simultaneous processing of multiple programs.

DOS (Disk Operating System)

To run a second program you have to wait the current one finishes. (Efficiency?)

Background spooling provides minimal degree of concurrency, this is to print in background.

Windows

Programs can run simultaneously, if they are not very resource consuming. (Same with Unix and Linux) (Multi-tasking)

Support for multiple users. (Multi-user)

Interaction with operating system

CLI (command line interpreter)

DOS: type a command in a command line

Can only enter commands from keyboard.

Unix/Linux: shell scripts (sequence of instructions)

User can define shell scripts and run multiple instructions sequentially

Windows/MacOS: click with mouse on icons

Function calls from within use program (API)

For example the following C program:

```
#include <stdio.h>
void main() {
    printf("Hello, World!");
}
```

Example: compiling Java programs

To compile a Java program the `javac` compiler need to be run

So: `javac myProc.java` [`Return` key] is pressed

Observation:

`javac` is a program to be executed.

Any program should be in the main memory (RAM) to be executed

Thus, `javac` should be loaded into the main memory from the disk and then be executed.

Question: who, or what does it? — Answer: The operating system.

Example: Linux commands

In Linux you type `ls` (similar to `dir` in DOS) command and press `Return` key to list the files in current directory

When `ls` is running it needs information read the files in the current directory.

Question: Who, or what provides such information? — Answer: OS. `ls` program makes request to the appropriate system program to this end.

Complexity of OS operation

Example: To accept a command from a user OS must:

Accept the input keystrokes from the keyboard.

Interpret this keystrokes as a command.

Determine the location of the file with a program for this command.

Read the appropriate blocks of file from (secondary storage) device into main memory.

Set up context for the program to be executed.

Transfer control to the program being executed.

Resume control when the program is finished.

If multiple programs are executing simultaneously, OS must include also:

Program to allocate memory and other resources to each program. (**Memory manager**)

Program to allocate CPU time to each program. (**Scheduler**)

Program to maintain integrity of each program. (**Security kernel**)

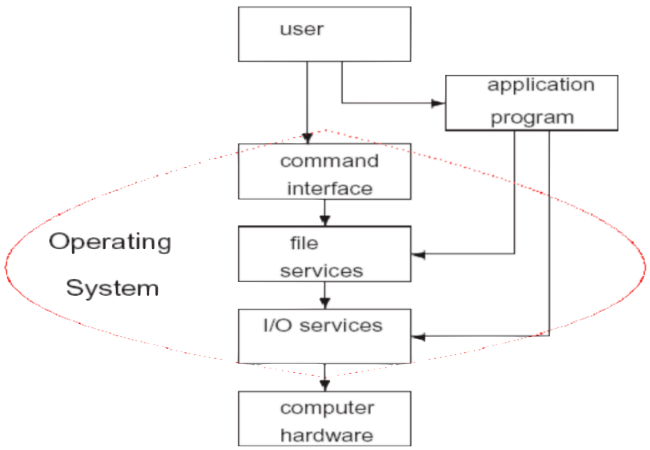
Simplified picture of OS services

User can either be running application programs or interacting with the system via the command line interpreter.

If a program is external then its located somewhere in the system's storage, so OS must be able to support file services.

Any file services must to be translated into I/O services.

Then eventually all the services will be translated into instructions for the computer hardware.



Computer Networks

Networking is perhaps the most far-reaching changes ever produced to von Neumann's original blueprint.

Operating systems usually provides access to network facilities. (via networking API such as socket interfaces)

Computer network is an interconnected collection of autonomous computers to facilitate fast information exchange.

Why computer networks?

To increase computing power.

Distributed computing projects: SETI@home, Folding@home

To share valuable resources (printers, large disks facilities, programs, databases, etc).

Central printing and filing system at XJTLU

To organize convenient interaction s of users working at their computers, often from different locations.

Such as online games, video chatting

Information at our fingertips…

Connectivity

Connecting devices for communication

Voice, data, multimedia

— foundation for information age

E-mail

Send and receive messages over a local area network or a large network, including the internet.

Telecommuting

Working at home or on the road

Communicating with the office through phone, fax, and/or computer

Online shopping

E-Commerce

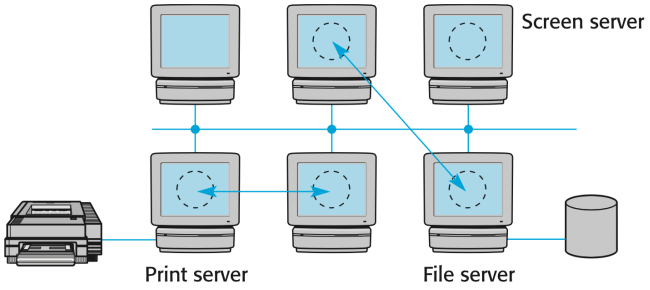
Browser and the world Wide Web

Google chrome, Microsoft Edge, Apple Safari, Mozilla Firefox

Client-server computing

Client: The originator of a request.

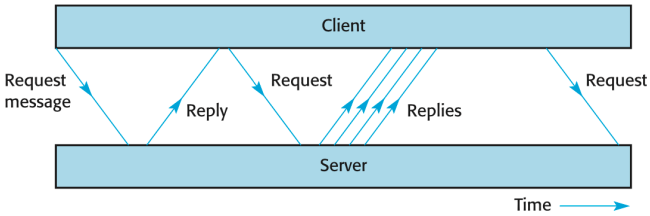
Server: The supplier of the service



Client server interaction

Client starts the interaction by sending a request message to the server.

Server responds by sending replies back…



Q&A Lecture 6

1. What is the most important development during the past 40 years evolution of OS?
   The invention of windowing interfaces.
2. OS needs to provide fair & protected access to system resources, why?
   Fair so no users will be stopped from using system resources, need to be protected so no user's private data will not be read or damaged by other users.
3. Explain 'multi-tasking'.
   Multiple programs can run concurrently.
4. Mention 3 functions that need to be provided when an OS is to support 'multi-tasking'
   OS must provide memory management, scheduling, and security kernel.
5. Explain 'multi-user'
   Multi user means computer machine supports multiple users to execute programs.
6. How does operating system provides access to network facilities?
   By supporting with socket interfaces and networking APIs.
7. Telecommuting and online shopping are enabled through the introduction of
   Networking.

Lecture 7

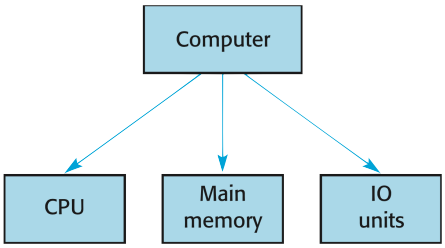
Reminder: The von Neumann model

The idea formulated by von Neumann (late 1940s):

The computer is a general-purpose machine controlled by an executable program.
A program is a list of instructions used to direct a task.
Both program and data are held in computer's memory (storage) and both represented by binary codes.
A processor is an active part of the machine that executes the program instructions.

The principal components of a computer

These are the minimum set of components for a working Digital computer.
A PC motherboard often appears much more complicated.



Mother board

There is the three major subsystems:

CPU, Main memory, and input-output units.

Each of these is often mad up of many components.
They need a medium to exchange data.

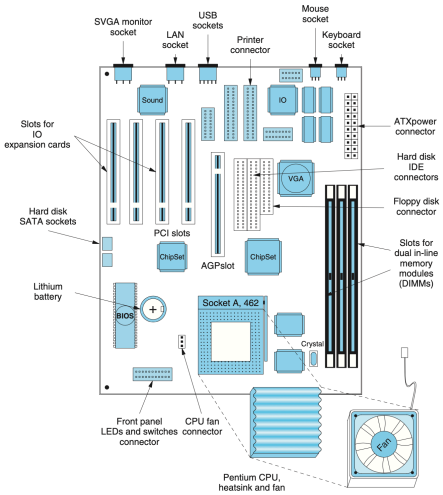
Processor

Arithmetic/logic unit (ALU)

Capable of executing arithmetic and logic related instructions.

Control unit:

Part of the CPU responsible for performing the machine cycle — fetch, decode, execute, store.



Registers

Program counter (PC):

Contains the address of the next instruction to execute

Instruction register (IR):

Part of a CPU control unit that stores an instruction

Coprocessors: Assistants to the CPU

Coprocessors: microprocessors performing specialized functions that CPU cannot perform or cannot perform as well and as quickly

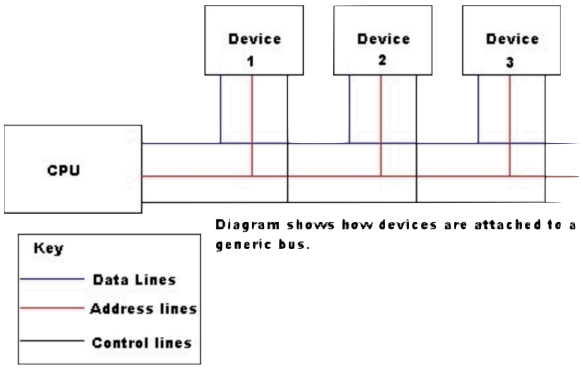
Math, Graphics

Board and Chips

Circuit boards:

Mother board provide sockets for extension boards

Use aluminum or copper to conduct electronic messages
Chips of silicon
Semiconductor



How do they exchanged data — via Buses

On the motherboard, all the components are interconnected by buses ("signal highways").
A bus is a bundle of conductors, wires, or tracks.
Typically there are address, data and control buses, each including several signal lines.

Intel 8086: 20 shared address/data lines, and a further 17 lines for control
Intel Pentium : data buses 64 lines, and the address bus 32 lines.

Bus bottle neck

Only two parties can make use of the bus, everything else need to wait, if device clock mismatches the wait time is longer.

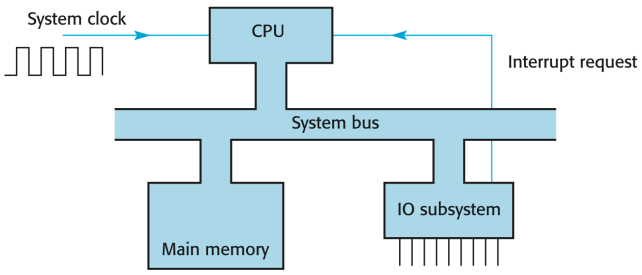
Each hardware unit is connected to all these buses.

A simple way of building up complex systems in which each unit can communicate with each other.

Advantage: Little disruption when plugging in new units and swapping out failed units.

System interconnections

The connected devices can have access to any signal lines they require
Bus interconnection is often represented in diagrams as a wide pathway, rather than showing the individual wires
CPU



Bus vs. point-to-point connections

An alternative scheme is point-to point connection

The number of pathways needed to link even possible pair of n units $n(n-1)/2$
Each pathway will still require a full-width data highway.

Could be 32 datelines, and say 6 control lines/.

The result will be a huge number of wires.

The number of wires required for a bus is much smaller than that for point-to-point connections.
However, a bus can only transfer one item at a time, like a railway line.

Which leads to limit of performance — Bus Bottleneck
It cannot be solved by simply increasing the speed of a processor.

Registers

CPU registers: small block of fast memory.

Temporarily store for data and address variables/

Some CPU registers:

Instruction pointer (IP) or Program counter (PC)

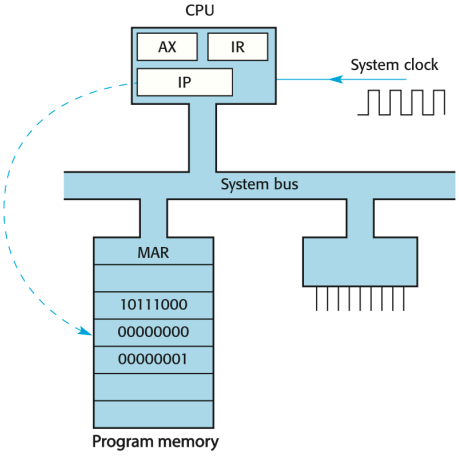
Stores the address of the next instruction.

Accumulator (AX, EAX in Pentium)

General purpose data register.

Instruction register (IR)

Stores the instruction that is being executed.



Memory address register (MAR)

Temporarily holds address of the memory location during a bus transfer.

Memory buffer register (MBR)

Exists in some architecture, to hold retrieval of any operand from memory into the CPU or to decode any data to be placed into the memory

Computer Instruction Set

The collection of machine language instructions that a particular processor understands

Machine language instructions

Instructions for a specific CPU

Designed to be executed by a computer without being translated
Also called machine code
We can represent these machine code as more user friendly symbolic mnemonics: ADD, SUB, INC, DEC, etc.

How instructions are executed?

The basic operation, known as the fetch-execute cycle or machine cycle.

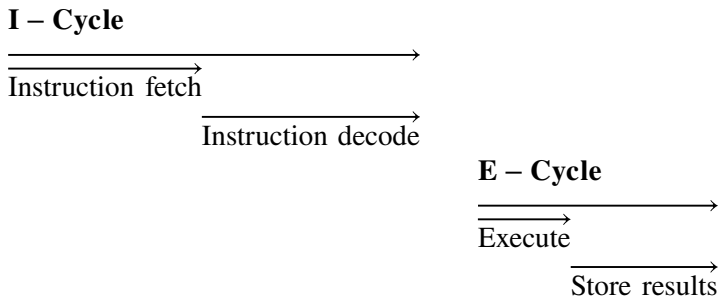
The sequence whereby each instruction of the program is exec used:

Read from the memory — read instruction with IP and stores it in IP.

Decoded — CPU reads into the binary and decide what action to take.

Executed — when semantic id decoded CPU can perform the instruction.

Machine Cycle



The fetch phase of the cycle

The address in IP register is copied onto the address bus and further to MAR register.

Ip is incremented ready for the next cycle. IP now points to the next location in the program memory.

Memory selects location and copies the content onto the data bus.

CPU copies the instruction code from the data bus into IR.

Decoding of the instruction starts.

The execution phase of the cycle

Execute phase depends on the type of instruction

Example: the execution of MOV AX, 256

Instruction includes:

IP is copied to address bus and latched into memory.

IP is incremented.

The value selected in memory is copied onto the data bus.

CPU copies the value from the data bus into AX

CISC & RISC: not all processors are designed equal

CISC (“sisk”)

Complex instruction set, most mainframe and PCs.

RICS(“risk”)

Reduced instruction set, cheaper and faster, shift some work to software

CISC vs. RISC

In RISC an instruction usually consist of a single word, but in CISC an instruction may be several word long requiring several fetches.

RISC is also faster because:

The vacated area of the chip can be used to accelerate the performance of more commonly used instructions rather than compensating for those rarely used instructions.

Easier to optimize the design

Simplifies translation from high-level languages into the smaller instruction set that the hardware understands, resulting in more efficient programs

Output Hardware

Hardcopy output

Graphics, Letters

Softcopy output

Video, audio

Screen resolution

Pixel is the basic unit for display

Communication Hardware

Facilitate networks

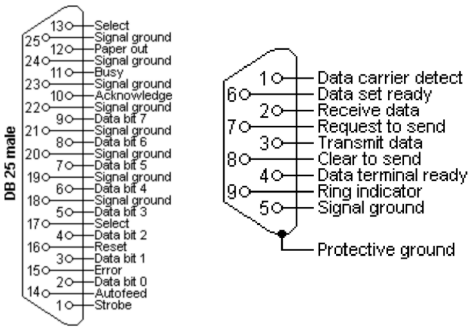
Modems — (modulation and demodulation)

In the past computer is connected via telephone wires, the signals sent is analog rather than digital. So computer's digital binary data must be modulated to analog using a modem before being transferred by the telephone wire.

Hubs and other components of a network

Assemble input to be concentrated from multiple devices.

Ports:



Parallel port (IEEE 1284)

Bits sent in parallel at once.

Printers, some scanners

Serial port (RS-232)

Bit sent in sequence one by one.

Modems, scanners, mice

These are now being replaced by:

USB (Universal Serial Bus)

Industry standard developed in the mid 90s that defines the cables, connectors, and protocols used for connection, communication and power supply between computers and electronic devices.

Standardized the connection of computer peripherals such as keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters to PCs.

Replaced earlier interfaces, such as serial and parallel ports, as well as separate power chargers for portable devices.

What else is inside a computer? Power Supply

Power supply

Protected by power surge protector or uninterrupted power supply unit (UPS)

Q&A Lecture 7

1. What is the difference between a ‘general purpose machine’ and a ‘special purpose machine’ (‘CPU’ vs ‘coprocessor’)
   General purpose machine is able to run any program provided by user, special purpose machine can only run program for special applications.
2. Assume there are 6 devices to be interconnected via 8 data lines (wires) plus 2 control lines (wires), how many wires will be needed if point to point connection scheme is used?
   $n = 6 \implies 6(6-1)/2 = 15$
   $15 \times (8+2) = 150 \text{ wires}$
3. Name 2 registers that are always used during each instruction execution.
   IP (Instruction pointer) and IR (Instruction register)
4. How many pixels are rendered with 1024×768 screen resolution?
   $1024 \times 768 = 786{,}432 \text{ px}$
5. Name two typical applications run by coprocessors.
   Graphics and mathematics processing.
6. Name 3 different types of bus in a computer system.
   Data, addressing and control.
7. List two reasons behind bus bottle neck.
   Bus can only be used by two parties each time.
   There exists speed mismatch between devices and the CPU.
8. Highlight 2 major components of a CPU.
   CU (control unit) and ALU (arithmetic & logic unit)
9. What tasks are performed during a machine cycle.
   The I-Cycle and E-Cycle, in which consists of instruction fetch, instruction decode, execution and result storing.
10. Computers & printers are usually connected via what types of ports?
    In the old days Parallel ports, now USB ports.
11. RISC refers to
    a. RAM that supports fewer instructions than do CISC chips
    b. Instructions that support fewer codes than do CISC chips
    c. <u>Processors that supports fewer instructions than do CISC chips</u>
    d. Coding schemes that are used as a back-up to CISC
12. The computer’s main processor follows its instruction to manipulate data into information.
    a. Hardware
    b. CPU
    c. <u>Software</u>
    d. Unicode
13. This type of hardware consists of devices that translate data into a form the computer can process.
    a. Application
    b. <u>Input</u>
    c. System
    d. None of the above is correct
14. Name 4 registers that are always used during each instruction execution
    IP, IR, MAR, MBR

Lecture 8

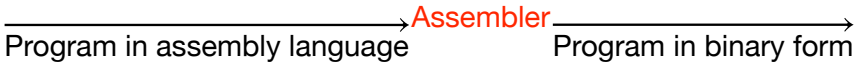
CPU and instructions

Each different kind of CPU will be different in the set of instruction that it can perform, and in the way these are represented in binary machine code.
General principles are similar.

Mnemonic form and assembly language

The mnemonic form of machine code is called assembly language

Assembly language and assembler



The assembler program takes as input a program written in assembly language, and:

Translate instructions into their binary **machine-code** form
Associates labels with memory **addresses**
Produces a binary **machine object code program.**

Associates **labels** with memory addresses.

```
L1:   JZ    L2
  ...
  CALL doD
  JMP L1
L2:   ...
```

Labels, for example, signify the beginning of a loop, the beginning of a block of memory cells, etc
A program has to be loaded into somewhere in main memory in order to be executed.

Programs must be **relocatable**. This means the program can be executed and the out come must be reproducible regardless of which memory area it is allocated in.

Binary machine object code program.

Also called object file.
Contains binary machine code.
May still need a linker to generate an executable or library by linking object files together.

Main memory, RAM

A CPU may executed only instructions loaded in the main memory

RAM: **Random Access memory**
The name is to set it apart form **sequential access memory** (e.g. serial tape storage) which you cannot access data stored in the last blocks without going through the first blocks of data.

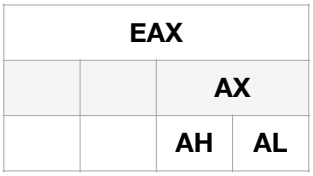
The memory can be seen as a set of numbered storage elements, called **words**, each of which can contain some information.
Each word is numbered with its **address**.
Any word of memory can be accessed “without touching the preceding words (Random Access).
Access time is the same for all the stored items.

Words, Bytes and Bits

A **word** may be visualized as a set of more elementary storage elements (memory cells), called **bits**, arranged in a row.
The number of bits in the word may vary between different computers; 32 bits in word is common.
A standard unit of 8 bits is called a **byte**.

CPU Registers in Pentium

EAX — accumulator
EBX — base register
ECX — counter register
EDX — data register
ESI — source index register
EDI — destination index register
EBP — pointer to base of program stack
EIP — instruction pointer
ESP — pointer to top of program stack
Flags
CS — code segment
SS — stack segment
DS — data segment
ES — extra segment



EAX register — the accumulator

Used as a general-purpose data register during arithmetic and logical operations.
Some instructions are optimized to work faster with the accumulator
In some instructions (MUL and DIV) the accumulator is assumed, no need to write it explicitly.
It can be accessed in various ways as 8, 16, or 32 bit chunks, referred to as AL, AH, AX EAX.

`MOV EAX, 1234H`	Load constant value 4660 into 32 bit accumulator.
`INC EAX`	Add 1 to accumulator value.
`MOV maxval, EAX`	Store accumulator value to memory variable ‘maxval’.
`DIV CX`	Divide accumulator by value in 16 bit register CX.

EBX register — the base register

The base register can hold addresses to point to the base of data structures, such as array in memory

`LEA EBX, marks`	Initialize EBX with address of the variable ‘marks’.
`MOV AL,[EBX]`	Get 1-byte value into AL using EBX as the memory pointer.

ECX — the Count Register

It is used as a counter in loops

`MOV ECX, 100`	Initialize ECX as the FOR loop index.
`...`	
`for1:`	Declare symbolic address label for1.
`...`	
`LOOP for1`	Decrement ECX, test for zero, JMP back to for1if ECX is non-zero.

EIP — the instruction pointer

The instruction pointer (program counter) holds the address of the next instruction.

`JMP mylabel`	Forces a new address into EIP

Further registers: ESI, EDI

ESI — the source index register, is used as appointed for string or array operations.

`MOV AX, [EBI+ESI]`	Get one 16-bit word using Base address and Index register.

EDI — the destination index register is used as a pointer for string or array operations.

`MOV EAX,[ESI]` `MOV [EDI],EAX`	Moves one 32-bit word from source to destination.

Further registers: ESP, EBP

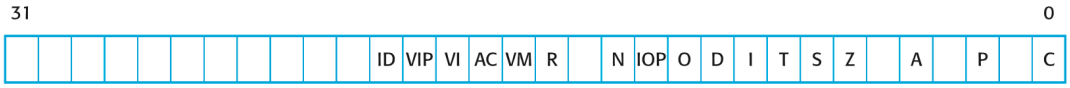
EBP — the stack base pointer
ESP — the stack pointer.

Q&A Lecture 8

1. The ‘execute’ phase is the same for different types of instructions (T or F?)
   False.
2. The assembler program takes as input a program in assembly language, and translate instruction into ? form.
   Binary machine object code.
3. Assembler associate labels with ?
   Memory addresses.
4. A CPU may execute only instructions loaded in the main memory (T or F?)
   True.
5. Access time is the same for all the RAM cells (T or F?)
   True.
6. What is the meaning of `MOV EBX,[EBX]`? What is the side effect of this instruction?
   Retrieve the content in the address of EBX and copy the content into EBX register, the side effect is that the original EBX content is overwritten.
7. `JNZ L2` means? When will there be compile time error for the above instruction?
   Forces the address of L2 into EIP if the most recent calculation sets the zero flag as off. A compile time error will occur when label L2 does not exist.
8. Is this a valid assembly code? `MOV maximal, loc1`
   Memory to memory direct transfer is now allowed.

CPU status flags

EFLAG: The Flag registers holds the CPU **status flags**.
The status flags are separate bits in EFLAG
Where information on important arisings conditions such as **overflow** or **carry bits**, is recorded.
A way of communication between one instruction and the subsequent instructions.



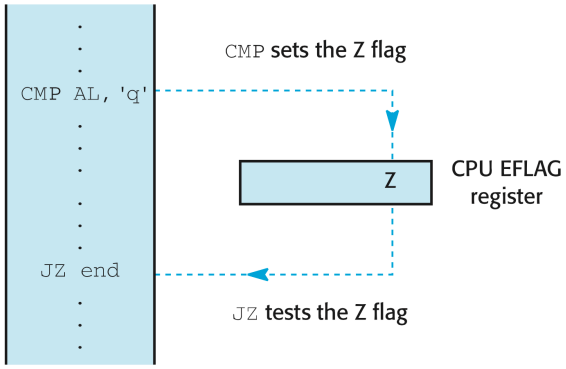
ID	Identification flag or CPUID availability
VIP	Virtual interrupt pending
VI	Virtual interrupt active
AC	Alignment check
VM	Virtual 8086 mode active
RFR	Resume task after breakpoint interrupt
NT	Nested task
IOPL	IO privilege level
O	Arithmetic overflow error
D	Direction of accessing string arrays
IE	External interrupt enable
T	Trap, single-step debugging, generates an INT #1 after each instruction
S	Sign, MS bit value
Z	Zero, result being zero
A	Auxiliary carry, used by BCD arithmetic on 4 LS bits
P	Parity, operand status
C	Carry, indicates an arithmetic carry or borrow result

Most often used flags are:
- S: sign, most significant bit value
- Z: zero, result being zero.
- C: carry, indicates an arithmetic carry occurs.
- O: arithmetic overflow error.
  - The addition of two large positive numbers which may give a negative result in a Two's complement system.

How CPU flags operates:

```
CMP AL,'q'
JZ end
```

CMP: Compare two values, subtract with no results only setting flags.
JZ: Conditional jump according to the Z flag



Inline Assembler

Inline assembly code can refer to C or C++ variables by name:

```
_asm mov eax, var ; stores the value of var in EAX
```

This example illustrates two more aspects:
- One can use `_asm` without brackets, in that case it works only for one line.
- Semicolon `;` is used for the comments in the assembly program. Alternatively, one can use `//` for the comments.

An `_asm {… …}` block can call C function, including C library routines.
We use C library routines `scanf` and `printf` for input and output in our programs.
To pass arguments to `printf` we use a **stack**.

Stack

Stack is a memory management (data structure) for storing and retrieval information (values).
The order of storing and retrieving the values for the stack can be described as **LIFO** (last in, first out)
The **ESP** register stores the address of the item that is on top of the stack.
Example of an alternative memory arrangement is a **queue**: **FIFO** (first in, first out)
Every stack is equipped with **push** and **pop** operation.

`PUSH reg/mem/imm` ; push element into stack and SUB ESP,[4/2 for 32/16 bit]
`POP reg/mem/imm` ; pop element from stack then ADD (E)SP,[4/2 for 32/16 bit]

Stack is a simple but useful data structure.
Almost any assembly language has special instructions for implementing a stack.
In inline assembly language there are `PUSH` and `POP` instructions.
`PUSH` and `POP` operations use the stack pointer register ESP to hold the address of the item on the top of the stack.

Assuming an upside-down stack PUSH and POP

Assume an upside-down stack in memory

PUSH:=
Decrement address in ESP
Write item into ESP

POP:=
Read item in ESP
Increment ESP by a correct amount to remove item from the stack.

Manual stack pointer adjustment

Sometimes the stack pointer has to be adjusted explicitly

`ADD ESP, 4`	Take 4 bytes off the stack.
`SUB ESP, 256`	Create 256 bytes of stack space.

Stack as the temporary storage

Examine the following C program

```
/* Prints "Hello, World!" 10 times */
#include <stdio.h>
#include <stdlib.h>
int main (void) {
    char format[] = "Hello, World!\n";
    _asm {
        mov ecx, 10     ; initialize loop counter
    Lj: push ecx        ; loop count index saved on stack
        lea eax, format ; load address of format array
        push eax        ; push address onto stack
        call printf     ; call library subroutine code
        add esp, 4      ; clear off 4 bytes from the stack
        pop ecx         ; restore loop counter
        loop Lj         ; dec ECX then jump to Lj if ECX != 0
    }
    // Equivalent to: for(int i = 10; i > 0; --i) printf(format);
    return 0;
}
```

If we wish to call a library function with parameters, we are assumed to use stack.
In the above example stack was used as a scratch-pad temporary storage to keep the value of loop counter and free ECX register for other use.

Passing parameters

`push eax`	Place the address of the string to be printed on the stack
`call printf`	Reads the address from the top of the stack and prints the string but it does not remove this address from the stack
`add esp, 4`	cleans the top of the stack manually

Printf routine needs extra information on what to print and how to print, it should be passed as parameters.
It is done via stack:

Q&A Lecture 9

1. Name 3 use cases of the Flag register in Pentium
   Carry flag, Over flow flag, Parity flag.
2. The Flag register can be used to pass information between one instruction and the subsequent instruction. (T or F?)
   True.
3. Which register is used to store the result of subtraction from this instruction? `CMP AL, BL`
   No register is used to store the result, but zero flag will be determined from `CMP`.
4. Under inline assembly what mechanism is used to pass arguments to printf routine?
   Stack is used for parameter passing.
5. Inline assembler can be used to call a C library function within an assembly segment in a C program (T or F?)
   True.
6. Status flags are set (or cleared) before an instruction is being executed (T or F?)
   False, it is set or cleared after execution.
7. D flag is used to set the direction of looping (T or F?)
   False, D flag is used to set the direction of string array cross section.

Structure of instructions

The binary codes of almost all instructions contain three pieces of information:

The **action** or operation of the instruction.

The **operands** involved (where to find the information to operate with).

Where the **result** is to go.

Machine instructions are encoded with distinct bit fields in the prefix to contain information about:

The operation required.

The location of operands and results.

The data type of the operand.

Pentium intersections can be from 1 to 15 bytes long, depending on:

The operation required.

The addressing modes employed.

Examples of the coding structure of some instructions

```
03 C3 ADD   AX, BX
  0000 00      11       11          11            000        011
  ADD Op       Dest     Word   Reg – Reg mode      AX         BX
                                                Destination  Source

66 B8 00 00 00 00 00 00 12 00     Mov     EAX,  12H
          01100110                1001     1    000   0000 0000 0000 0000 0000 0001 0010
        Prefix = 66H              MOV Op   Word AX              Immediate data

3C 71 CMP   AL, 'q'
  0011110      0        0111 0001
CMP A Op     Byte    Immediate data
```

Addressing modes

Addressing mode

The way of forming operand addresses.

More technically, the way one can compute an **effective address**.

Effective address is the address of operand used by instruction.

Offering various addressing modes support better the needs of HLLs when they need to manipulate large data structures.

Immediate (operand) mode.

Example: `mov eax, 104 ; eax = 104;`

Part of the binary code here is the **value(104)** of the operand

Data Register Direct:

Example: `mov eax, ebx ; eax = ebx;`

The operand is contained in the registers and the instruction contains code of the registers, this is the fastest to execute.

Memory Direct:

Example: `mov eax, a ; eax = a;`

Here `a` is a variable, stored in memory and the instruction contains the address of this variable.

Decode the address into a temporary register within the CPU, and then go to the memory location to read the data into the CPU.

Address Register Direct:

Example: `lea eax, message1 ; *eax = &message1;`

The instruction, contains the address of `message1` variable, which is loaded into `eax` register after the execution of the instruction.

Commonly used to initialize pointers which then reference data arrays.

This is different from `mov eax, message1`, which is to move the content of `message1` to `eax`.

Register Indirect:

Example: `mov eax, [ebx] ; eax = *ebx;`

The instruction copies to the `eax` register the content of a memory location with the address stored in `ebx`.

Register Indirect mode in Arrays

```
/* Handling arrays with Register Indirect addressing mode */
...
    int myarray[5] = {1,3,5,7,9};
    _asm {
        lea ebx, myarray ;    *ebx = &message1;
        mov ecx, 5       ;     ecx = 0; // size of array as ecx
        mov eax, 0       ;     eax = 0; // eax to store sum
    L1: add eax, [ebx]   ; do{ eax += *ebx;
        add ebx, 4       ;     ebx++; // int occupies 4bytes(32bits)
        loop L1          ; }while(--ecx != 0);
    }
```

Indexed Register Indirect with displacement:

Examples:

```
mov eax, [table+esi]
mov eax, table[esi]
```

In both cases the effective address is obtained by adding the address '`table`' contained in the operand field of the instruction to the content of a register (registers).

```
/* Arrays with Register Indirect Displacement addressing mode */
...
    int myarray[5] = {1,3,5,7,9};
    _asm {
        mov ecx, 5            ;     ecx = 0; // size of array as ecx
        mov eax, 0            ;     eax = 0; // eax to store sum
        mov ebx, 0            ;     ebx = 0
    L1: add eax, myarray[ebx] ; do{ eax += myarray[ebx];
        add ebx, 4            ;     ebx++;
        loop L1               ; }while(--ecx != 0);
    }
```

Q&A Lecture 10

1. The Pentium instruction set has a fixed width for all instructions to speed up instruction decoding. (T or F?)
   False, pentium instruction size can be 1 up to 15 bits long.
2. Which sample program is more efficient (Register Indirect or Indexed Register Indirect)?
   The initialization is not much different, but then indexed register indirect code runs longer because `myarray[ebx]` is an extra summation for each iteration of the loop. Thus, Register indirect mode is faster.
3. What are those 3 major components covered by every instruction?
   Type of operation, address of operand, and type of operand. Address to store result is optional.
4. Machine instructions are encoded with what to contain information about the operation required?
   Encoded with distinct bitfield in the prefix.
5. Which of the following is the fastest to execute?
   a. `mov eax, b   ; mov reg, mem`
   b. `mov a, ebx   ; mov mem, reg`
   c. `mov eax, ebx ; mov reg, reg` is the fastest
6. The following instruction will involve the calculation of 'effective address' (T or F):
   `mov eax, table[esi]`
   True, the effective address of `table` is summed with the value of `esi`.
7. After the execution of the following instruction, `eax` will contain the content of `message1` inside. (T or F)
   `mov eax, message1`
   False, `eax` will not contain the content of `message1`, since it is a char[], so the assignment will assign the address pointer to the array instead of value.

Output in inline assembly

```
…
char format[] =  "Hello World\n";
…
…
lea eax, format ; load address of format into eax
push eax        ; push address of format onto stack
call printf     ; call library code subroutine
add esp, 4      ; manually clean off 4 bytes off stack
```

We have seen the above code fragment implementing the call to `printf` function.
Equivalent C code is: `printf("Hello World\n");`

Printing numbers

To print the value of the integer variable myInt one can use the following call to the standard C library routine:

```
printf("%d", myInt);
```

Qualifier "`%d`" means the content of '`myInt`' will be printed as a decimal integer.
The following implements such a call in assembly code:

Push the 2nd parameter (integer variable) to the stack
Push the 1st parameter (address of the format string) to the stack
Call `printf` routine.
Clean up top two position in the stack.

```
/* Printing numbers in inline assembly */
...
int main (void) {
    char format[] = "%d";
    int  myInt    = 42;
    _asm {
        push myInt      ; push 2nd parameter onto stack
        lea eax, format ; load address of format array
        push eax        ; push 1st parameter onto stack
        call printf     ; call library subroutine code
        add esp, 8      ; clear 8 bytes(2 positions) off stack
    }// Equivalent to: printf("%d", myInt);
    return 0;
}
```

Input in inline assembly

To input the value into the integer variable '`input`' one can use the following call to the standard C library routine

```
scanf("%d", &input);
```

Qualifier "`%d`" means the input will be read as a decimal integer, `&input` presents the address of the variable '`input`'.

```
/* Reading numbers in inline assembly */
...
int main (void) {
    char format[] = "%d";
    int  input;
    _asm {
        lea eax, input  ; load address of integer
        push eax        ; push 2nd parameter onto stack
        lea eax, format ; load address of format array
        push eax        ; push 1st parameter onto stack
        call scanf      ; call library subroutine code
        add esp, 8      ; clear 8 bytes(2 positions) off stack
    } // Equivalent to: scanf("%d", &input);
    return 0;
}
```

Example: Read a number and print it out:

```
int main (void) {
    char message[] = "Your number is %d!\n";
    char format[]  = "%d";
    int  input;
    _asm {
        ; Equivalent to: scanf(format, &input);
        lea eax, input
        push eax
        lea eax, format
        push eax
        call scanf
        add esp, 8
        ; Equivalent to: printf(message, input);
        push, input
        lea eax, message
        push eax
        call printf
        add esp, 8
    }
    return 0;
}
```

More about `printf` and `scanf`

There are more qualifiers (types) which can be used in `printf` and `scanf`.

%c // character.
%d, %i // signed decimal number.
%s // string of characters util a white space terminator

Controlling program flow

Very few programs execute all instructions sequentially, from the first till the last one.
Usually, one needs to control the **flow of the program**:

Jump from one point to another, often depending on some conditions.
Repeat some actions while some condition is maintained, or until some condition is reached.
Passing control to and from procedures.

Jumps

Jump is the most straight forward way to change program control from one location to another.
Jump instructions falls into two categories: Unconditional and Conditional.

Unconditional jumps:

`JMP` instruction transfers control unconditionally to another instruction:
It has the syntax: `JMP <address of the target instruction>`
The address of the target instruction is given by its label.
Typical use of `JMP` instruction is to skip over code that should not be executed.

Conditional jumps:

Jumps that first test a condition, then jump if condition is true else continues.
There are more than 30 jump instructions.

`JCXZ` and `JECXZ`, test whether the counter register `CX` or `ECX` is zero.
Remaining jump instructions test the status flags.

Example:

```
            JECXZ finish
            mov eax, inp
    finish: add esp, 4
```

At `JECXZ`, if is `ECX` zero then next instruction is `add esp, 4` else its `mov eax, inp`.

Jumping based on status flags

Instruction	Jump if
JC/JB	Carry flag is set (=1)
JNC/JNB	Carry flag is clear (=0)
JE/JZ	Zero flag is set (=1)
JNE/JNZ	Zero flag is clear (=0)
JS	Sign flag is set (=1)
JNS	Sign flag is clear (=0)
JO	Overflow flag is set (=1)
JNO	Overflow flag is clear (=0)

Jumps based on comparison of two values

The `CMP` instruction is the most common way to test for conditional jumps.
It compares two values without changing them, while it changes the status flags according to the results of the comparison.
For eaxmple if the value of `eax` and `ebx` are the same then the execution:

```
cmp eax, ebx
```

Will set zero flag Z=1

Jumping based on Comparison

Instruction	Jump if
JE	Two operands are equal
JNE	Two operands are not equal
JG	First operand is greater
JGE	First operand is greater, or equal
JL	First operand is less
JLE	First operand is less, or equal

Example 1:

```
cmp ax, bx ; compare AX and BX.
jg  label1 ; if (AX > BX) goto label1.
jl  label2 ; if (AX < BX) goto label2.
...        ; else continue to execute.
```

Example 2:

```
add ax, input ; AX += input.
cmp ax, 0     ; compare AX with 0.
jge label1    ; if (AX >= 0) goto label1.
jl  label2    ; if (AX < 0)  goto label2.
```

Q&A Lecture 11

1. To pass two parameters to `printf` in inline assembly code, the first parameter should be pushed onto the stack first. (T or F?)
   False, the parameters should be passed in reverse order.
2. When calling `printf` in inline assembly, the parameter passed to it will be popped off stack by `printf`. (T or F?)
   False, we have to manually adjust `ESP` after the `printf` call.
3. When calling `scanf` in inline assembly, the address of the variable to receive input needs to be pushed onto the stack. (T or F?)
   True.
4. What is the conversion specifier to be used when printing a string under `printf`?
   %s
5. If three integer parameters were pushed onto stack when calling '`scanf`' in inline assembly, how would you adjust the value of register '`esp`' when returining from '`scanf`'
   Add 4*3 onto esp, that is: `add esp, 12`
6. The execution `cmp eax, ebx` of will check upon the setting of zero flag. (T or F?)
   False, execution of `cmp` will set the setting of zero flag, not check.

Controlling program flow: Loops

Simple loop:

`loop <label>`

Decrements `(E)CX`, if `(E)CX!=0` jumps to `<label>`, else continues.

Loop upon two conditions

`loopne <label>`

Decrements `(E)CX`, if `(E)CX!=0&&Z=0` jumps to `<label>`, else continues.

Implementing higher-order constructs: conditional statements

```
/* Java Conditional Jump*/
if (c > 0) pos += c;
else       neg += c;
…
```

```
; Assembly1:
         mov eax, c
         cmp eax, 0
         jg positive
negative:add neg, eax
         jmp endif
positive:add pos, eax
   endif:...
```

```
; Assembly2:
         mov eax, c
         cmp eax, 0
         jle negative
positive:add pos, eax
         jmp endif
negative:add neg, eax
   endif:...
```

Implementing higher-order constructs: the for statement

```
/* Java for statement*/
for(int i = 0; i < 10; i++)
   y += x;
```

```
; Assembly1:
         mov eax, 0
for_loop:add y, eax
         inc eax
         cmp eax, 10
         jl  for_loop

; this implementation is problematic
because it is hard to debug
```

```
/* Java for statement*/
for(int i = 3; i < 20; i += 2)
   y += x;
```

```
; Assembly2:
         mov eax, 3
for_loop:add y, eax
         add eax, 2
         cmp eax, 20
         jl  for_loop
```

Implementing higher-order constructs: the while statement

```
/* Java while statement*/
while(fib2 < 1000) {
    fib0 = fib1;
    fib1 = fib2;
    fib2 = fib1 + fib0;
}
```

```
; Assembly:
    while:mov eax, fib2
          cmp eax, 1000
          jge end_while
          mov eax, fib1
          mov fib0, eax
          mov eax, fib2
          mov fib1, eax
          add eax, fib0
          mov fib2, eax
          jmp while
end_while: ...
```

Do-while statement: Exercise

```
/* Java do-while statement*/
do {
    fib0 = fib1;
    fib1 = fib2;
    fib2 = fib1 + fib0;
}(fib2 < 1000);
```

```
; Assembly:
    while:mov eax, fib1
          mov fib0, eax
          mov eax, fib2
          mov fib1, eax
          add eax, fib0
          mov fib2, eax
          cmp eax, 1000
          mov eax, fib2
          jl  while
end_while: …
```

Switch-case statement:

```
/* Java switch-case statement*/
switch(num) {
    case 1: …;
        break;
    case 2: …;
        break;
    default:…;
}
```

```
; Assembly:
         mov eax, num
         cmp eax,1
         je, case_1
         cmp eax,2
         je, case_2
         ;default
         jmp end_case
   Case1:…
         jmp end_case
   Case2:…
         jmp end_case
end_case:…
```

Implementing `loop` with `dec,cmp,jne`

`loop <label>`
is equal to
`dec ecx`
`cmp ecx, 0`
`jne <label>`

Q&A Lecture 12

1. Conditional jumps in assembly can be used to implement HLL constructs like while, for and switch (T or F?)
   True.
2. '`loop`' instruction in assembly as a branching effect based upon the value of decremented `ECX` register. (T or F?)
   True.
3. Explain what '`loopne <label>`' does
   Decrement `ECX`, then jump to label if `ECX!=0 & Z=0`
4. Explain what the following instructions do:
   `CMP EAX, EBX`
   `LOOPNE label`
   Jump to label if `ECX!=0 & EAX!=EBX`

Subroutines

A **subroutine** is a general term. In different programming languages it may be called differently:

Procedure in Pascal, **Function** in C, **Method** in Java.

A subroutine is a part of the code which can be used repeatedly within the program that is being executed.

`CALL` is the instruction to call a subroutine (i.e. `call printf`)

Subroutines: what are they good for?

Save the effort in programming.

Reduces repeated parts of the program, reducing time cost to repeatedly implement them.

Reduce the size of programs.

By extracting repeated parts of program as subroutine it will reduce the size of program.

Share the code.

Facilitates code sharing and reuse across multiple places of the program, it also facilitates code sharing across multiple projects.

Encapsulate, or package, a particular activity

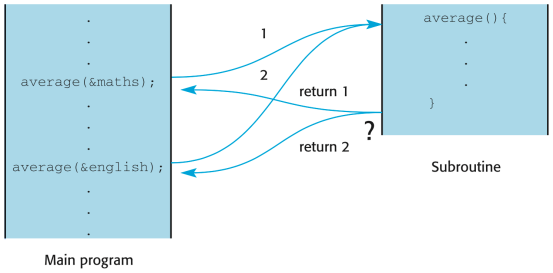
Hide implication from the user

Provide easy access to tried and tested code.

Facilitate code reuse.

Subroutines: Fundamental issue — where did I come from?

The subroutine is invoked at different moments from two different places in the main program.



Return addresses

How does the computer know where to return from a subroutine?

Different calls of the same subroutine return to different places.

One needs to store a **return address** for every call of the subroutine.

Where is the return address stored?

How much room do we need?

Subroutines in assembly language

Declaration of a simple procedure looks as follow

```
<label> PROC
    … ; subroutine implementation
    RET
<label> ENDP
```

The procedure can be called by the instruction: `CALL <label>`

Return form the subroutine

The instruction `RET` changes the control, causing execution to continue from the point following the `CALL` instruction.

How does `CALL` work

`CALL` records the current value of `EIP` (Instruction pointer) as the **return address**.

Places the required subroutine address into `EIP`, so the next instruction to execute is the first instruction of the subroutine.

Nested calls

If the return address was stored in a dedicated memory location, it would work for the first call, but for second call there will be no place to store the return address.

The solution to this is to use a **stack** to store all return addresses.

CALL and RET working together

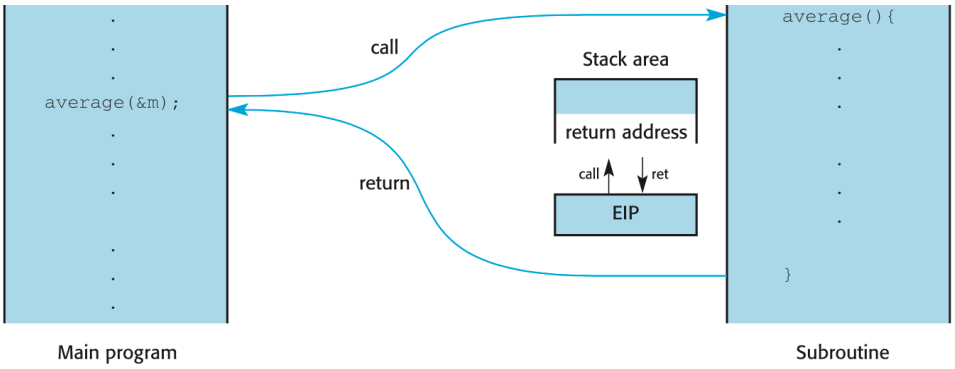
CALL does the following:

Records the current value of EIP as the **return address**. Copy the address from EIP to the stack (PUSH).

Puts the required subroutine address into EIP, so the next instruction to execute is the first instruction of the subroutine.

RET does the following:

Pop the last address stored in the stack and put it into EIP.



Q&A Lecture 13

1. A subroutine can be called at various moments from different places in the main program. (T or F?)
   True.
2. The subroutine can be called by itself.
   True.
3. How many return addresses can you store using a stack?
   As much as memory allows.
4. How many nested calls can you make within a program?
   As many as possible as long as program stack can hold.
5. How does the computer know when to return from a subroutine?
   When the RET instruction is encountered.
6. What happens when a 'CALL …' instruction is executed?
   When the CPU detects the 'CALL …' instruction it will push the EIP onto the program stack, then use the entry point address of the subroutine to replace the EIP value.
7. What happens when a 'RET' instruction is executed
   It will pop from the program stack and write that address into the current EIP.

Parameters

The simplest kind of subroutine performs an identical function each time it is called. Such subroutine requires no further information to run.

But most subroutines require additional information, given in **parameters**.

For example, the '`printf`' subroutine requires information of what to print and how to print

Stack can be used to pass parameters to subroutines (lecture 11), and to store return address of subroutine calls (lecture 13).

In genera, parameters can take a number of forms, depending on the nature of the information required by the subroutine.

Two main forms of parameters:

Value parameters.

Reference parameters.

Value parameters

In many cases, the additional information required by a subroutine will be a simple value (or values) of gym type(s):

For example, a numeric value or ASCII code value, etc.

Imagine a subroutine, when given two numbers, has to decide and return which number is bigger. All this subroutine needs is the values of two variables.

Such parameters are called **value parameters**.

Example: value parameters

```
        mov eax, first
        mov ebb, second
        call bigger         ; call bigger
        mov max, eax
…
bigger proc
        mov save1, eax      ; procedure which uses value parameters
        mov save2, ebx      ; passed through registers EAX and EBX
        sub eax, ebx
        jg first_big        ; if(eax > ebx) goto first_big
        mov eax, save2      ; else eax = ebx
        ret
first_big: mov eax, save1 ; first_big: eax = eax
        ret
bigger endp
```

Exercise1: return difference of EAX and EBX via EAX

```
diff proc
        mov save, ebx
        cmp eax, ebx
        jg first_big        ; if(eax > ebx) return
        sub eax, ebx        ; else eax = ebx
        ret
first_big: sub ebx, eax
        mov eax, ebx
        mov ebx, save
        ret
diff endp
```

Exercise2: Swap values in EAX and EBX

```
swap proc
        mov temp, eax
        mov eax, ebx
        mov ebx, temp
        ret
swap endp
```

Reference parameter

Consider the following subroutine: “Given two variables, swap their values”.
Having only the values of variables is not enough. We need another type of parameter.
A variable is a location in the memory, the name of the variable determines the address of this location.
Thus, it will be enough for the above procedure to have the address of the variables as an additional information.
Such kind of parameters are called **reference parameters**.

Example: Swap value of addresses in EAX and EBX

```
lea eax, first          ; let eax and ebx be the addresses
lea ebx, second         ; of two variables first and second
call swap               ; call subroutine swap
…
swap proc
       mov temp, [eax] ; dereference the addresses
       mov [eax],[ebx] ; then swap the contents
       mov [ebx],temp  ; via registers.
       ret
swap endp
```

Passing parameters via registers

The simplest method of passing parameters into a subroutine is to use a register:
City a value into an available CPU register and then jump to the subroutine.
Both value and reference parameters can be passed using registers.
But, this method would be too constraining, because there only exists limiting number of registers inside CPU to allow passing parameters by reference.

Stack instead of registers

For subroutines with a few parameters one may pass these parameters (in address, or value form) using general purpose registers.
However, in general, the number of registers available is very limited, and is not enough to implement subroutines with an arbitrarily large number of parameters.
Thus, we need to use the memory in order to implement parameter handling for subroutines.
The solution in most computers is to use the **stack** for passing parameters and keeping the local variables.

Local variables

In the example with value parameters the instruction make use of two variables, `save1` and `save2` as a temporary storage for parameters
They may be through of as local variables of the subroutine, in contrast to the global variables used throughout the program

Stack frame

The area of the stack which holds all data related to one subroutine call is called a **stack frame**.
This data includes:
Parameters of the subroutine.
Return addresses.
Local variables.

EBP and ESP for stack frames

Because of the nested calls several stack frames may be present at the same time.
Two registers are used to work with stack frames:
`EBP`: The stack frame base pointer.
To indicate the base of the current stack frame.
`ESP`: The stack pointer
To hold the address of the top of the stack.

How does stack frame works?

ESP is always pointing to the top of the stack

EBP initially contains an address of the base of the stack,

Just before and during the call of a subroutine the following happens:

The parameters are pushed onto the stack.

The return address is pushed onto the stack

The address stored in EBP is pushed on the stack.

A new stack frame is created.

The current address of the top of the new stack frame is saved in EBP.

The local variable are installed on the new stack.

Ones the subroutine done executing:

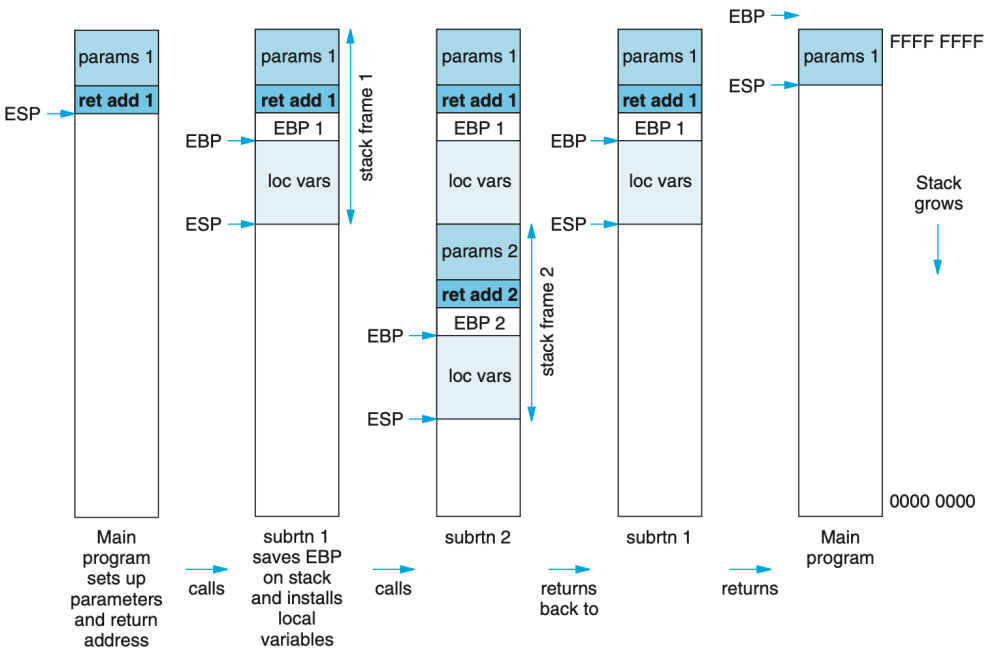
Pop all local variables out of the stack.

Pop the previous EBP address from the top of the stack and restore it into EBP (stack frame base pointer)

Clean up parameters in the stack.

Pop the return address and save it in EIP.

// Note: the popping order is crucial



Q&A Lecture 14

1. When is value parameter good for?
   When only the value of parameter is needed for subroutine processing.
2. When is reference parameter good for?
   When you need to use the address for subroutine processing
3. Why multiple stack frames can coexist the same time?
   Because subroutine call exists.
4. What type of data are stored under a stack frame
   Parameters, retuning data, previous value of EBP, local variables of the subroutine is stored.
5. What happens when a 'CALL …' instruction is executed?
   Parameters are being pushed, returning address is stored, the base of EBP is recorded, then new stack frame is created, let EBP and EBP to point to the new stack frame.
6. What happens when a 'RET' instruction is executed?
   Pop all the local variable, restore EBP value, pop all parameters, restore ESI for CPU's next execution

Stack frame

The area of the stack which holds all the data related to one call of a a subroutine.
The data includes:
- Parameters of the subroutines.
- Return addresses.
- Old stack pointer contents (`EBP`).
- Local variables.

Recursive subroutines

A recursive subroutine, or procedure, is one that may in some circumstances calls itself to perform some subsidiary task.
For example a subroutine `SUBR` may `CALL SUBR`.
Recursion may appear in a more subtle form of **mutual recursion**, when, e.g.,
`SUBR` calls `SUBR2` and
`SUBR2` in turn calls `SUBR`.

Examples of recursive definitions (procedures)

Factorial function.
- factorial(1) = 1
- factorial(n) = n * factorial(n-1).

Merge sort.
- Given a list, split it into two parts.
- Apply merge sort to each part.
- Merge the results.

Recursive method for factorial function in Java

```
static long factorial(int n) {
    if (n < 2) return 1;
    else return n * factorial(n-1);
}
```

Implementation in the assembly language

One auxiliary procedure to be used in the recursive factorial procedure:

```
multiply PROC
      pop eax
      mov aux, eax
      pop eax
      mul eax, aux
      ret
multiply ENDP
```

It takes two top values on the stack, multiplies them and return the result in `eax` register.
Side effects:
- Stack modified (2 pops)
- `eax`'s original content replaced with the product.

Stacks for recursion: main procedure for the factorial

```
factorial PROC        ; input n in eax
       push eax       ; push current n onto the stack
       dec eax        ; decrement n
       jz finish      ; jump to finish if n = 0
       call factorial ; call factorial recursively
       push eax       ; push the result of the last factorial call
                      ; onto the stack
       call multiply  ; call multiply subroutine
       ret            ; return
Finish:pop eax        ; pop the parameter from the stack into eax
       ret            ; return with the results in eax
factorial ENDP
```

Side effects: Stack content will be changed

Q&A Lecture 15

1. A recursive procedure will typically provide a n exiting condition. (T or F?)
   True, there should be a base case.
2. A recursive procedure will typically have a divide-and-conquer step. (T of F?)
   True, it must have a divide-and-conquer step.
3. What is the side-effect of the procedure '`multiply`'?
   The value for multiplication sits inside the program stack, and the stack need to be popped twice, changes program stack content, also changes value of `EAX`.
4. A recursive procedure can always be re-implemented using iteration without recursion. (T or F?)
   True, any recursive procedure can be implemented with an iterative construct.

Numbers vs. text

Numbers (coasting of numeric characters) are often processed differently from text.
But still they must be entered into computer just like any other alphanumeric characters, one digit at a time

Number 314 consists of 3 alphanumeric characters
Number 3.14 consists of 4 alphanumeric characters
Number -3.14 consists of 5 alphanumeric characters

Conversion between alphanumeric representation of numbers and special representation of numbers is done usually in a program (software).

The method `java.lang.String.valueOf(double)` converts a double float number to its string representation.
The method `java.lang.Integer.parseInt(String)` converts a string into the integer value that it represents.

Number representations

We have seen that a computer stores all data in binary form.
To store numbers we need an **encoding scheme**, which would allow us to encode:

The **algebraic sign** of numbers (+/-).
Decimal point that might be associated with a fractional number.

One solution:

Store the number as a string of characters
However, it is not practical for calculations

Unsigned integer can be represented directly in binary form.

Two categories of integer data

Unsigned integer & Signed integer

Unsigned integers

Unsigned binary representation - just store any whole number in its binary representation.
Thus, a byte can store any **unsigned integer** of value between 0 and 255.
A 32-bit word can store **unsigned integer** in a range $\left[0,\ 2^{32}-1\right] = \left[0,\ 4{,}294{,}967{,}295\right]$
Multiple storage locations (words) can be used to represent bigger unsigned integers.

Unsigned integers: BCD

Binary-coded decimals (BCD):

Digit-by-digit binary representation of original decimal integer.

Each decimal digit is **individually** converted to binary

This requires 4 bits per digit (not all 4-bits patterns are used).

Example

Decimal value 68 is represented in BCD as 01001000.

One byte can store **unsigned integer** in a range 0-99 under BCD encoding, because one byte is 8-bits.

Binary vs, BCD representation

BCD is less economical than binary representation, because using 4-bit to encode just 10 digit is wasting 6/16 possible 4-bit binary patterns to encode.
Calculations in BCD are more difficult.
But, translation between BCD and character form is easier.

Last 4 bits of ASCII numeric character codes correspond precisely to BCD representation.

Example

ASCII code of ‘5’ is 0110101, its BCD representation is 0101.

Binary representation is more common, BCD is used for some business applications.

Signed integers

Sign-and-magnitude representation.
Complementary representations.

Sign-and-magnitude representation

It is representation of **signed integer** by a **plus** or **minus** sign and a **value**.

Agreement.

Left-most bit represent a sign, e.g., 0 stands for + and 1 stands for -.

Example

00100111 represents 39
10100111 represents -39
00000000 represents 0
10000000 represents -0

Calculations are more difficult than in the case of binary (unsigned) representation.
Two different binary codes exists for number 0: 00000000 and 10000000 (for 8-bit codes)
Positive range of signed integer is one-half as large as the unsigned integer of the same number of bits.

The same holds for negative range.

Thus, 8-bits can represent the numbers from -127 to 127 (0 being represented twice)

Complementary representation

For most purposes, computers use different methods of representing signed integer known as **complementary representation (conventions)**.
Two's binary complementary representation is the most common.

10's complementary coding

10's complementary coding is decimal analogue of 2's complementary
Consider the case of 3 decimal digits
The problem is:

How to use 3 decimal digits to represent positive and negative numbers, so that the usual operations (addition, etc) are computable.

Solution:

Representation $500\ldots999\,|\,0\ldots499$

Number $-500\ldots-001\,|\,0\ldots499$ being represented

Thus, the number (3-digit sequence):

From 000 to 499 are representing themselves.

Any number x from 500 to 999 represent negative number $-(1000-x)$, that is, negated complement to 1000. The number 0 is not represented twice

This convention is easily extended to the case of n decimal digits

By dividing the range of numbers into two half, the first half represents themselves, the second half represents negated complements. $-(10^n-x)$.

Methods of complements

A technique in mathematics and computing used to subtract one number from another using only addition of positive numbers. Commonly used in mechanical calculators and modern computers.

Example of (x-y)

Suppose x > y, instead of subtracting y from x, the complement of y is added to x and the leading '1' result is discarded.

$$\begin{array}{lcll} 373 \;|(x) & & 373 & |\quad (x) \\ -218 \;|(y) & \Longrightarrow & +748 & |\quad (10's\ \text{complement of} -y) \\ \quad ? & & 1155 & \end{array}$$

The first '1' digit is then dropped, giving 155, the correct answer.
In the case x < y, there will not be a '1' digit to cross out after addition.

$$\begin{array}{lcll} 185 \;|(x) & & 185 & |\quad (x) \\ -329 \;|(y) & \Longrightarrow & +671 & |\quad (10's\ \text{complement of} -y) \\ \quad ? & & 856 & |\quad (10's\ \text{complement of} -144) \end{array}$$

Giving the negative numbers as 10's complements.

The first '1' digit is then dropped, giving 155, the correct answer.

Q&A Lecture 16

1. What are the pros & cons of BCD?
   Pros, easy conversion from BCD to ASCII, by taking lowest four bit.
   Cons, wasting storage, only 10/16 capacity of code is used, and hard to compute.
2. Is it possible to have a 3's complement scheme?
   Yes.
3. Can overflow still occur under 2's complement addition?
   Yes, when two large positive number sum results a negative number then over flow occurs.
4. What are the pros & cons of 2's complement?
   Pros: very efficient to compute, takes care of computation efficiency issue.
   Cons: it's not intuitive for human.

10's complement representation

For the 10's complementary representation we have to specify the number of digits (some *n*).

The number being represented depends on that *n*-digit used for encoding. For example:

- 3-digit complement: '567' represents -433.
- 4-digit complement: '0567' represents 567.

Addition

Operation of addition for *n*-digit numbers represented by 10's complementary convention:

- Addition modulo: 10^n
- Use existing addition procedure, except that any carry beyond the specified number of digit (*n*) is "thrown away".

Examples of addition in 3 digits

10's complementary codes	Actual values
699	-301
400	400
——	———
1099	99
099	

10's complementary codes	Actual values
814	-186
923	- 77
——	———
1737	-263
737	

Subtraction

To subtract B from A one may first compute negated B and add it to A.

$$A - B = A + (-B).$$

Given B to compute $-B$ just subtract B from 10^n.

For example, negated value of 250 = 1000-250 = 750 = -250

- Addition modulo: 10^n
- Use existing addition procedure, except that any carry beyond the specified number of digit (*n*) is "thrown away".

Overflow testing

Example:

Code: '347' + '230' = '577'.

Values: 347 + 240 = -423.

this is an overflow, the result 577 is too big to fit into 3-digit complementary representation (-500 - 499)

Rule to test for overflow:

If both inputs to an addition have the same sign, and the output sign is different, overflow has occurred.

Positive and Negative

In 10's complement system the range is **unevenly** divided between positive and negative integers, for example 4-digit complement system represents all integer value between -5000 to 4999.

Two's complement in 8 bits.

Two's complement representation for binary (n=8) is similar to 10's complement representation for decimal.

binary codes: 10000000 … 11111111	00000000 … 01111111
two's complement: -128 … -1	0 … 127

Number that begin with 0 are considered to be positive representing themselves
Number that begin with 1 are representing negative numbers.
A complement of the number in 2's complementary *n*-digit encoding can be found in one of two ways:

Either subtract the value from the modulus 10_2^n
Or, invert all bits and add 1 to the result.
Example: 11011101 represents -00100011.

Consider 2's complementary code 11011101

The code lead with 1, hence is a negative number

1. invert all bits 11011101 -> 00100010
2. Add 1 00100010 -> 00100011

The binary number represented is -00100011.

Addition

Addition modulo 10_2^n — excludes leftmost carry bit.

Example:

011011	27
+101100	-20
1000111	7

Subtraction and overflow

Same as 10's complementary

$A - B = A + (-B)$.

IF both inputs to addition have the same sign and the output sign is different overflow occurs.

Numerical types in Java

`byte` 8-bit : $-2^{8-1} \sim 2^{8-1} - 1$

`short` 16-bit : $-2^{16-1} \sim 2^{16-1} - 1$

`int` 32-bit : $-2^{32-1} \sim 2^{32-1} - 1$

`long` 64-bit : $-2^{64-1} \sim 2^{64-1} - 1$

all signed integers using 2's complement representation.

Q&A Lecture 17

1. Addition for *n*-digit numbers represented by 10's complementary convention is based upon addition modulo n. (T or F)?
   False, it is modulo 10^n.
2. How is overflow detected during the addition
   When operand have same sign and result have a different sign.
3. Can overflow result from subtractions?
   Yes. Subtraction can be reduced into addition, then use the same rule as addition overflow can be detected.
4. Under 2's complement system, numbers that begin with 1 are representing negative numbers. (T or F)?
   True.
5. How many bits are required for a Java short integer?
   16 bits that hold values from $-2^{15} \sim 2^{15} - 1$
6. How many bits are required for a Java long integer?
   64 bits that hold values from $-2^{63} \sim 2^{63} - 1$
7. An 'int' type under Java encodes integer under this range: $-2^{32} \sim 2^{32} - 1$?
   Wrong. In its a 32 bit value that hold values from $-2^{31} \sim 2^{31} - 1$

Lecture 18

Floating Point Numbers

It is not always possible to express numbers in integer form.

Real, or **floating point** numbers are used in a computer when:

The number to be expressed is outside of the integer range of the computer, like 5.375×10^{25}.

Consists of Sign (+), Mantissa (5.375) and Exponent (10^{25}, or can be base 2 in computer).

Or, when the number contains a fraction, like 345.0256.

Exponential notation (base 10)

In general, this notation represents numbers in a form $a \times 10^b$

Components of exponential notation

The **sign** if the number.

The **magnitude** of the number, known as the **mantissa**.

The sign of the **exponent**.

The magnitude of the exponent.

Two additional pieces of information:

The **base** of the exponent (e.g., 10 or 2)

The location of the **decimal point**. (or binary point in binary)

Floating point formats

Any format for floating point numbers should specify how the components of an exponential notation are stored (in a word, or several words).

The **base of the exponent** and the **location of the binary point** are standardized as part of the format and, therefore, **do not have to be stored** at all.

Example: Suppose, that the standard code consists of space for eleven digits and a sign:

SEEMMMMM

So, there are two digits for the exponent and 5 digits for the mantissa

Trade-off: precision (mantissa) vs. range (exponent).

Most commonly, the mantissa is stored using **sign-magnitude** format.

What about the sign of the exponent -> Excess-*n* notation for the exponent?

Excess-*n* notation for the exponent:

Excess-50 notation for the 2-digit decimal representation of the exponent

Representation: 0…49 | 50…99

Exponent: -50…-1 | 0…49

Offset the value of the exponent by a chosen amount (here it is 50), deducting 50 from the actual representation.

It is simpler to use for exponent compared to the complementary form.

Thus, 5-digit excess-50(2-digit for exponent) notation allows us a magnitude range of:

$0.00001 \times 10^{-50} < \ldots < 0.99999 \times 10^{+49}$

We assume that the decimal point is located at the beginning of five-digit mantissa.

Normalization of floating point numbers

To maximize the precision for a given number of digits, numbers are, usually stored **with no leading zeros**.

Process of transformation the number into such a form is called **normalization**.

Example: 0.00123×10^7 has normalized form 1.23×10^4

Floating point in the computer: binary representation

Typically, 4, 8 or 16 bytes are used to represent a floating point number.

Typical floating point format: 32 bits are used to provide a range of approximately $10^{-38} \sim 10^{+38}$.

1 bit is used for sign of mantissa.
8 bits are used to store exponent in excess 128 notation.
23 bits are used for mantissa.

Floating point in binary

Assuming **normalized** representation one can omit the storage of most significant bit (it is always 1)

so 23 bits provide 24 bits of precision.

Binary point should be specified (but not stored)

Most common choice is **after the most significant bit, i.e 1.xxx**

Thus this is not stored.

Excess-128 example

Consider the code:

$$0\ 10000001\ 11001100000000000000000$$

Sign of mantissa is '+' left most bit = 0.

Mantissa is $1.11001100000000000000000$, where the 1. is assumed.

Exponent is $10000001 - 10^7 = 00000001$

Therefore the number represented is:

$$+(1.11001100000000000000000 \times 10^{1})$$

$$= +\ 11.10011000000000000000000$$

Consider the code:

$$1\ 10000100\ 10001111000000000000000$$

Sign of mantissa is '-' left most bit = 1.

Mantissa is $1.10001111000000000000000$, where the 1. is assumed.

Exponent is $10000100 - 10^7 = 00000100$

Therefore the number represented is:

$$-(1.10001111000000000000000 \times 10^{100})$$

$$= -\ 11000.11110000000000000000$$

Consider the code:

$$1\ 01111110\ 10101010101010101010101$$

Sign of mantissa is '-' left most bit = 1.

Mantissa is $1.10101010101010101010101$, where the 1. is assumed.

Exponent is $01111110 - 10^7 = -\ 00000010$

Therefore the number represented is:

$$-(1.10101010101010101010101 \times 10^{-10})$$

$$-0.01101010101010101010101$$

IEEE standard 754 - Single-precision floating point format

Single-precision floating point format:

Almost the same format as we just describe, with some exceptions

32bits: 1bit for sign, 8 bits of exponent, 23 of mantissa

The exponent is formatted using excess-127 notation.

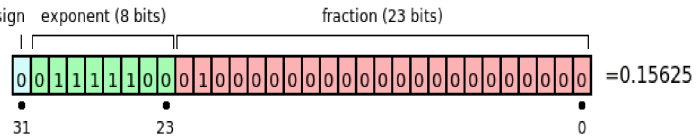
Overall, the standard allows approximately 7 decimal digit precision and approximate value range $10^{-45} \sim 10^{38}$.

Exponent biasing:

The exponent is based by $2^7 - 1$, that is based by 127.

exponents in the range $-127 \sim 127$ are representable

$e = 128$ reserved for NaN, Infinity



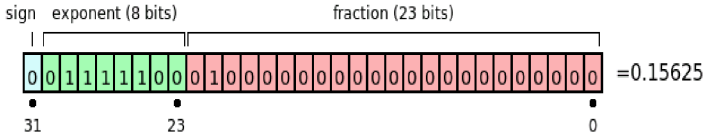
The represented number has value v:

$$v = s \times 2^{e} \times m, \text{ where}$$

$$s = \begin{cases} +1 & \text{signBit} = 0 \\ -1 & \text{signBit} = 1 \end{cases}$$

$$e = \text{exponent} - 01111111_2\ (127)$$

$$m = 2.\text{fraction}.$$



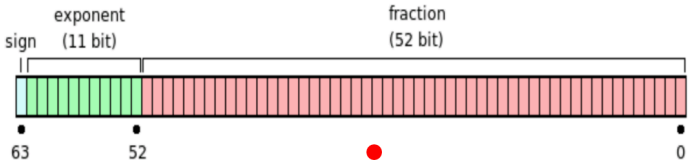
Referring back to the example, where $s = +1, e = -3, m = 1.01_2 = 1.25$.
The represented binary number is therefore $+1.01_2 \times 2^{-3} = +0.15625$

Special cases ($e = 128$)

$\text{exponent} = 0$ & $\text{fraction} \neq 0 \rightarrow \quad v = \pm 0$ (depending on the sign bit)
$\text{exponent} = 128$ & $\text{fraction} = 0 \rightarrow v = \pm\text{infinity}$(depending on the sign bit)
$\text{exponent} = 128$ & $\text{fraction} \neq 0 \rightarrow v = \text{NaN}$

IEEE standard 754 - Double-precision floating point format

64-bits: 1 bit for sign, 11bits of exponent, 52 of mantissa.
The exponent is formatted using excess-1023 notation.
The standard allow approximately 15 decimal digit precision and approximate value range $10^{-325} \sim 10^{308}$.



Q&A Lecture 18

1. Under the IEEE 754 standard how many bits are required to specify the sign of the magnitude?
   1 bit.
2. Under the IEEE 754 standard how many bits are required to specify the sign of the exponent?
   No bit, uses excess value to represent negative exponent.
3. Under the IEEE 754 standard, how many bits are required to specify the decimal point position
   Not bit used to encode, 1. is assumed because the floating point is normalized.
4. Does IEEE 754 provide a specification for Nan?
   Yes, reserved & defined as encodings where exponent is excess num+1 and fraction != 0.
5. Under IEEE 754 standard for single-precision floating point format, what type of excess notation is used for exponent specification?
   Excess-127.
6. Under IEEE 754 standard for double-precision floating point format, what type of excess notation is used for exponent specification?
   Excess-1023.
7. An 'int' type under Java encodes integer under this range: $-2^{32} \sim 2^{32} - 1$?
   Wrong. In its a 32 bit value that hold values from $-2^{31} \sim 2^{31} - 1$.

The von Neumann Model (remainder)

Input Device → [Processor ⟺ Memory] → Output Device

Input device is for transmitting information from a user into the computer's memory.
Output device enables a user to see the results of a program.

Data storage - overview

Main memory
Mass storage
Memory hierarchy
Coding of information

Data storage

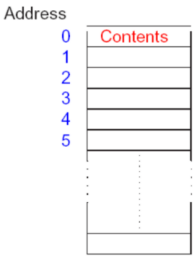
Storage is the capacity of device to hold and retain data.
Two main types of storage in a computer:
Main memory - system memory, volatile
Mass storage - permanent, long-term

Main memory

Refers to *physical memory* that is internal to the computer.
The computer(CPU) can manipulate only data that is inside the main memory.
The amount of main memory in a computer is crucial because:
It determines how many programs can be executed one time. (concurrency)
How much space can be allocated to a program. (size of program)
If memory is insufficient → suspension of program.

Main memory, RAM

A CPU may execute only instruction loaded in the main memory.
RAM: Random Access Memory
SAM: Sequential Access Memory
The name is to set it apart from serial tape storage with which you cannot access data stored in the second or following blocks without going through the first block of data.



RAM

The memory can be seen as a set of numbered storage elements, called **words**, each of which contains some information.
Originally intel set standard of 16bit words, then 32bit, now 64bits, varies by architecture.
Each word is numbered with its **address**.
Any word of memory can be accessed "without touching" the preceding words (Random Access).
Access time is the same for all the stored items.

Words, Bytes and Bits

A word may be visualized as a set of more elementary storage elements (memory cells), called **bits**, arranged in a row.
Length of bits varies by architecture, intel starts with 16bit words, then 32bit, now 64bits.
A standard unit of 8 bits is called a **byte**.

Bytes, Kilobytes, Megabytes, etc

1 Byte (B)	=	8 bits.		
1 Kilobyte (KB)	=	2^{10} bytes	=	1024 bytes.
1 Megabyte (MB)	=	2^{10} KB	=	1024 KB.
1 Gigabyte (GB)	=	2^{10} MB	=	1024 MB.
1 Terabyte (TB)	=	2^{10} GB	=	1024 GB.

Typical sizes

Typical RAM module has 512MB / 1GB of memory.

Common sizes:

floppy disk:	1.44 MB.
CD:	650 MB.
Memory disc:	1+ GB.
DVD:	4.7 GB.
HDD:	120, 200, 300 GB.

Compare:

Everything written by William Shakespeare can be stored in 8MB.

Human genome contains about 0.8GB of data.

Types of Ram

Dynamic Ram (DRAM):

Cheaper, but slower

Implemented via capacitors

DRAM needs to be **refreshed periodically**.

to retain the data the system need to keep recharging the DRAM device so that the data aren't lost.

Static RAM (SRAM):

Faster, but more expensive

Implemented via flip-flops.

no need for refreshing

Both types of RAM are **volatile**:

They lose their contents when the power is turned off.

Refreshing DRAM

Real capacitors leak charge.

Stored data eventually fades.

To retain data, capacitor charge has to be refreshed periodically.

ROM Chips

Read-only memory (ROM)

Typically used to store **firmware**.

Helps boot up system BIOS — Basic Input Output System (Drivers)

Bootstrap software & BIOS is stored in ROM so it cannot be altered.

Bootstrap rom: consist instruction to fetch OS image.

Cache memory

Quick access memory, Internal or external to the processor

Bridge between the processor and RAM

including **simultaneous read/write**

Video memory

VRAM, for graphics memory and video, this is important for gaming.

CPU $\Longleftrightarrow_{\text{word transfer}}$ Cache $\Longleftrightarrow_{\text{block transfer}}$ Main memory

Mass storage

Various techniques & devices for storing a large amount of data.

Can retain data even when computer is turned off.

Types of Mass storage

- Hard disks
- Optical disks: CD-ROM, CD-RW, DVD, etc.
- USB disks, Floppy disks.
- Etc.

Hard Disk Drives (HDD)

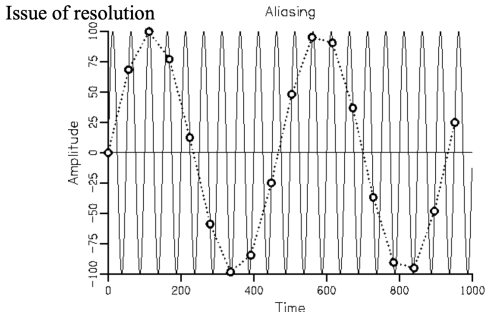
- Most important type of permanent storage used in computers
- Differ from other mass storage in 3 ways:
  - Size (usually larger)
  - Speed (usually faster)
  - Permanence (usually fixed in computer and non removable)

RAM vs. Mass Storage (HDD)

- Ram is volatile, HDD are not.
- In general RAM is faster than mass storage:
  - 10ns(10e-8s) vs 10ms(10e-2s) seek time.
- RAM is more expensive:
  - 50 pounds for: 8GB RAM vs 1TB HDD.
  - 1MB RAM is 125x more expensive than 1MB HDD.

Storing real-world data in digital

- Analog data → Digital data
- Converting Analog Signal to Digital Signal: **Sampling** Rate
  - Sampling = reduce data to be stored.
  - Solid curve = analog signal at comparatively high frequency.
  - Circles = Samples taken at relatively lower sample rate.
  - Dotted line = illustrate apparent frequency of sampled waveform, completing 2 cycles in the period that the original signal completed in 20 cycles.



  - Computer can only store discrete value, discrete resolution.
  - High resolution = costs more memory.

Storage Requirements for Digital Audio

- CD quality Audio:
  - 44.1 KHz sampling rate, 16bits / sample
  - $\rightarrow 16 \text{ bits sample}^{-1} \times 44.1 \text{ KHz} = 705{,}600 \text{ bps}$
  - $(\text{bps} = \text{bits s}^{-1})$
- Stereo:
  - 44.1 KHz sampling rate, 32bits / sample
  - $\rightarrow 32 \text{ bits sample}^{-1} \times 44.1 \text{ KHz} = 1{,}411{,}200 \text{ bps}$

Q&A Lecture 19

1. A computer system encodes data by means of this coding scheme to represent letters, numbers, and special characters.
   a. magnetic
   b. analog
   c. binary
   d. optical
2. Which of the following is volatile
   a. ROM
   b. RAM
   c. DVD
   d. HDD
3. Which of the following has best access
   a. Register - within CPU so its fastest to access
   b. Cache
   c. DVD
   d. RAM

Review Session 2

1. Given that a CD-ROM can store up to 70mins of stereo auto file, calculate the Megabytes of data a CD-ROM stores. (Audio file has specs: Stereo, sampling rate at 44.1KHz, 32 bits/ sample)

   $\text{Stereo} = 2 \text{ channel}$

   Data size of each auto sample:

   $$2 \times 16 \text{ bits sample}^{-1} \times 44.1 \text{ KHz} = 1{,}411{,}200 \text{ bits sec}^{-1}$$

   Data size for 70 mins

   $$= 1{,}411{,}200 \text{ bits sec}^{-1} \times 70 \text{ min} \times 60 \text{ sec min}^{-1}$$
   $$= 5{,}927{,}040{,}000 \text{ bits}$$
   $$\approx 706.56 \text{ Mb}$$

2. Given that a CD-ROM can store 700 Mb of data, how many minutes of stereo audio data can it store? (Audio file has specs: Stereo, sampling rate 44.1KHz, 32 bits / sample)

   $$2 \times 16 \text{ bits sample}^{-1} \times 44.1 \text{ KHz} = 1{,}411{,}200 \text{ bits sec}^{-1}$$
   $$700 \text{ Mb} = 700 \times 2^{20} \times 8 = 5{,}872{,}025{,}600 \text{ bits}$$
   $$\text{Maximum Time} = \frac{5{,}872{,}025{,}600 \text{ bits}}{1{,}411{,}200 \text{ bits sec}^{-1}} \approx 4161.015873 \text{ sec} \approx 69.35 \text{ min.}$$

3. If we are going to store (320×240 pixel, 24 bits pixel^{-1}, 30 frames sec^{-1}) video only (no audio), how many minutes can we store in a CD-ROM with a capacity of 700 Mb.

   $$320 \times 240 \text{ pixel} \rightarrow 76{,}800 \text{ pixel frame}^{-1} \rightarrow 1{,}843{,}200 \text{ bits frame}^{-1}$$
   $$\rightarrow 55{,}296{,}000 \text{ bits sec}^{-1} \rightarrow 6{,}750 \text{ Kb sec}^{-1}$$
   $$700 \text{ Mb} = 700 \times 2^{20} \times 8 = 5{,}872{,}025{,}600 \text{ bits}$$
   $$\text{Maximum Time} = \frac{5{,}872{,}025{,}600 \text{ bits}}{55{,}296{,}000 \text{ bits sec}^{-1}} \approx 106.1925926 \text{ sec} \approx 1.77 \text{ min.}$$

4. To store (320×240 pixel, 24 bits pixel^{-1}, 30 frames sec^{-1}) video only (no audio) for 60 minutes using a CD-ROM with capacity of 700 Mb, find compression ratio

   Without compression such a CD-ROM can store up to 1.77mins of video.

   $$\text{Compression ratio} = \frac{60}{1.77} = 33.9 : 1 \approx 34 : 1$$

5. If we are going to store (320×240 pixel, 24 bits pixel^{-1}, 30 frames sec^{-1}) video with CD quality audio, how many minutes can we store in a CD-ROM with a capacity of 700 Mb.

   Sample rate for CD-quality audio =

   44.1 KHz sampling rate, 16bits / sample

   $$\rightarrow 16 \text{ bits sample}^{-1} \times 44.1 \text{ KHz} = 705{,}600 \text{ bps}$$

   Sample rate for video =

   $$320 \times 240 \text{ pixel} \rightarrow 76{,}800 \text{ pixel frame}^{-1}$$
   $$\rightarrow 1{,}843{,}200 \text{ bits frame}^{-1} \rightarrow 55{,}296{,}000 \text{ bits sec}^{-1}$$

   Sum sample rate = $56{,}001{,}600$ bps

   $$700 \text{ Mb} = 700 \times 2^{20} \times 8 = 5{,}872{,}025{,}600 \text{ bits}$$
   $$\text{Maximum Time} = \frac{5{,}872{,}025{,}600 \text{ bits}}{56{,}001{,}600 \text{ bits sec}^{-1}} \approx 104.8546042 \text{ sec} \approx 1.74 \text{ min}$$

6. To store (320×240 pixel, 24 bits pixel^{-1}, 30 frames sec^{-1}) video with CD quality audio for 60 minutes using a CD-ROM with capacity of 700 Mb, find compression ratio

   Without compression such a CD-ROM can store up to 1.74mins of video & audio.

   $$\text{Compression ratio} = \frac{60}{1.74} = 34.48 : 1 \approx 35 : 1.$$

Lecture 20

Memory Length and Address

Any memory location in main memory has its own **address**.
The larger the memory the larger address are needed.
Maximal memory length depends on **address width**.

Address Width

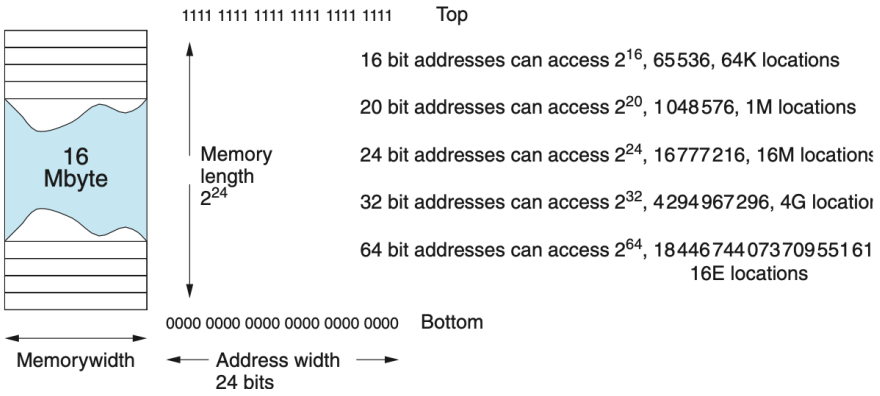
determined by:

The number of bits in the CPU address registers such as IP, MAR.
The number of lines in the address bus.

Address Width and Memory Length

Address width	Maximal memory length
16	64 Kb
20	1 Mb
24	16 Mb
32 Pentium	4 Gb
64 64-bit architecture	> 17 billion Gb

Memory parameters



Ideal configuration

With 32-bit address width the maximum memory addressing space is 4Gb.
Ideal configuration would be to have a single memory chip, and sending an address via 32 address lines.

32-bit CPU $\Leftarrow [A_0 \sim A_{31}] \Rightarrow$ 4 GB SRAM.

$A_0 \sim A_{31}$ are the address lines

Not ideal in practice:

4Gb is a lot of space and many systems don't use it all.
DRAM usually hav size 128Mbits, 256Mbits, 1Gbits, combined into memory modules of 8 to 16 chips.
Tp address memory, the memory chips and location within the chips both need to be selected.

Memory mapping

Given an address, system maps the address to the memory location on the chip.
When CPU send out an address:

A part of the address locates the correct chip.
Another part specifies an address within the correct chip.

How actually the addresses are mapped to the memory locations is defined by **memory maps**.

Memory Map for a small system:

Device	Size	Pins	32bit address bus	Address range
PROM1	1Mb	20	0000 0000 XXXX ++++ ++++ ++++ ++++ ++++	0000 0000 - 000F FFFF
RAM1	16Mb	24	0000 0001 ++++ ++++ ++++ ++++ ++++ ++++	0100 0000 - 01FF FFFF
RAM2	16Mb	24	0000 0010 ++++ ++++ ++++ ++++ ++++ ++++	0200 0000 - 02FF FFFF
RAM3	16Mb	24	0000 0011 ++++ ++++ ++++ ++++ ++++ ++++	0300 0000 - 03FF FFFF
RAM4	16Mb	24	0000 0100 ++++ ++++ ++++ ++++ ++++ ++++	0400 0000 - 04FF FFFF

\+ Address line used directly for internal section
X Line ignored, indicates partial (degenerate) addressing
0 Must be 0 for chip selection
1 Must be 1 for chip selection

Memory Address Decoding

Memory chips are not normally matched to the width of the address bus. For example:
CPU may send 32-bit address while RAM may receive directly 24-bit address.
Special **Memory Address Decoding** circuit implements necessary decoding

Memory Levels

- Registers
- Cache memory (Level 1 and Level 2)
- Main memory
- Mass storage

Why we need memory levels?
To keep up with demand from faster CPUs
To limit system cost.
To cope with ever-expanding software systems.

Registers

Memory cells which are core part of the processor itself
Fast access speed (few nanoseconds)
Small in memory size: tens of 32/64/80-bit registers (typically).

Cache memory

A memory (more expansive, but faster SRAM) placed between CPU and the main memory.
Contains a copy of the protein of main memory.
The aim is to maintain in fast cache the currently active sections of code and data.
Processor first checks cache when accessing information.
If information not found in cache, the block of memory containing the target information is copied into the cache.

Levels of cache

CPU $\Longleftrightarrow$ L1 cache $\Longleftrightarrow$ L2 cache $\Longleftrightarrow$ Main memory
Typical L1 cache has size 8-64 Kb.
Typical L2 cache has size 128-512 Kb.

Localization of access

The ideal of cache memory exploits **localization of Memory Access** principle:
Computer tend to spend periods of time accessing the same locality of memory.
Therefore:
A portion of code or data which require access needs to be loaded into the fastest memory nearest to CPU.
Other sections of the program and data can be held in readiness lower down the memory hierarchy.
Why this works:
programmer clustering related data items together in arrays or records.
repeating patterns in a program (i.e. loops)
compiler attempting to organize the code in an efficient manner

Cache memory and cache control unit

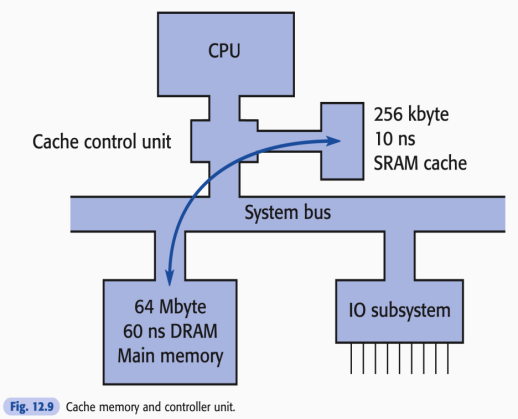


Fig. 12.9 Cache memory and controller unit.

When CPU request for memory it will always go through the cache control unit, it will determine if the memory is in the cache or not, if not a **cache miss** occurs then the cache control unit will initiate memory transfer into the cache.

Memory hierarchy

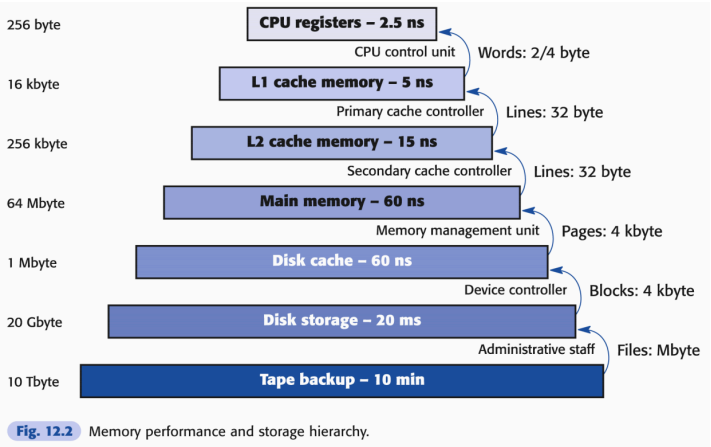


Fig. 12.2 Memory performance and storage hierarchy.

Going down the hierarchy:
- Increased capacity
- Increased access time
- Decreased frequency t of access of the memory by the processor
- Decreased cost per bit.

Q&A Lecture 20

1. Identify locality using the following program code:

```
...
_asm {
mov ecx, 5 // size of array
mov eax, 0 // store sum
mov ebx, 0
L1: add eax, myarray[ebx]   // element at myarray+ebx
add ebx, 4 // int occupies 4bytes, so increment to point to next.
loop L1
}
...
```

The elements in an array always have element in consecutive memory blocks, hence this shows locality.

2. Which of the following has fastest access?
   A. **register**
   B. cache
   C. disk
   D. tape

3. Identify locality using the following program code:

```
...
int myarray[5] = { 1, 3, 4, 7, 9};
_asm {
lea ebx, myarray
mov ecx, 5
mov eax, 0
L1: add eax, [ebx]
add ebx, 4
loop L1
}
...
```

   The elements in an array always have element in consecutive memory blocks, hence this shows locality.

4. Which of the following is most expensive in terms of cost?
   A. **register**
   B. cache
   C. disk
   D. tape

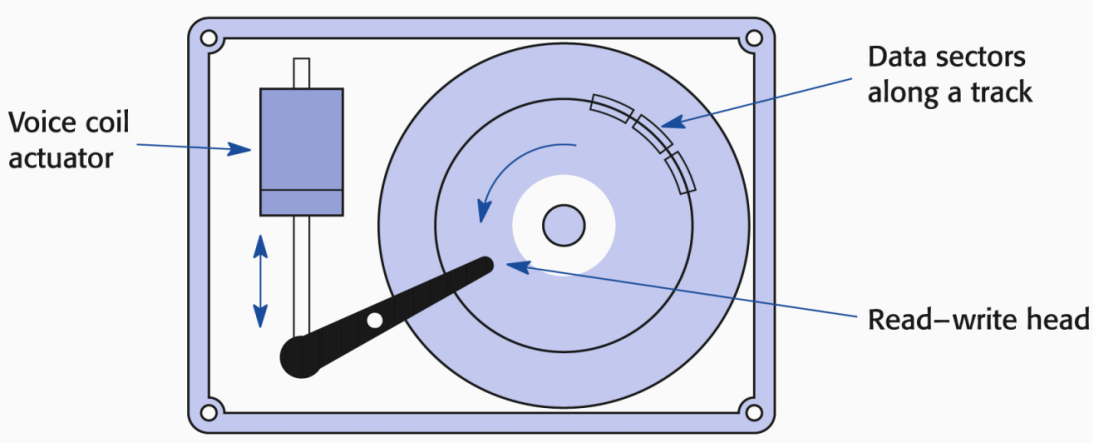
5. Given the following components, form a reasonable memory hierarchy to ensure cost-speed tradeoff: Cache, disk, RAM

   Cache
   RAM
   Disk

6. Locality can be temporal, which means data that are recently used tend to be accessed more likely than the others. (T or F)

   True.

Schematic Diagram of hard disk



Hard disk consists of platters stacked one on another.
For each platter there is two head one on top another on bottom associated with it.
Track is a region partitioned into multiple sectors.
Heads can move in and out to read different tracks.
Sector is typically 512 bytes.
Hard disk is also capable of rotating, so head only need to move on dimension (in out)

Tracks, sectors and cylinders

Each disk platter has its information recorded on both **surfaces**.
Each platter has two **heads**.
The information is recorded in concentric circles called **tracks**.
Each track is partitioned into **sectors**, each holding 512 bytes of information.
Cylinder is tracks positioned at same distance from center of the disk — concurrently accessible.
Double sided floppy is a disk consisting of one platter.

Addressing

CHS — Cylinder, Head, Sector system
Telling the disk controller which cylinder, head and sector to access.
Can be mapped onto LBA.
LBA — Large/Linear Block Addressing
by absolute number of a sector, prescribes each sector with a sequence of number.

How does it work — simplified example:

1. figure where on the disk to look for the needed information
2. location on disk → address expressed either in terms of CHS or LBA.
3. request is sent to the drive over the **disk drive interface** giving it this address and asking for the sector to be read.

Progress in the last 25 years

First PC hard disk: capacity 10 MB and cost 100 pounds per MB
Capacity > 300 GB and cost < half penny for 1 MB.

Hard disk vs. main memory

Larger, Slower, Cheaper per memory

Disk cache

A portion of main memory used as a **buffer** to temporarily hold data for the disk.
Disk write operations are ***clustered***.
Some data written out may be needed again
The data are retrieved rapidly from the disk cache instead of slowly from disk.
Java 's `java.io.PrintWriter.flush()`.

Storage Technology

Retrieving files into RAM is called *reading*

loading an application

opening a file

files can be programs or documents

Copying data from RAM onto a secondary storage device is called *writing*

Files, records, fields, keys

Files: e.g. `PERSONNEL FILE`

Records: e.g. Adam's personal data:

```
Adam Smith 35    Manager    Purchasing
```

Fields: e.g. `name, age, position, job function.`

Key: e.g. `Adam Smith`.

Virtual Memory

Virtual memory is a technique, in a sense, opposite to caching:

It is the use of low-level memory (i.e. hard disk) to 'expand' high-level (main) memory.

It provides a convenient expansion of main memory through 'overflowing' data and program code onto magnetic disk.

The area **on disk** reserved for this purpose is known as the **swap area**.

Memory management

when processor needs more RAM space, it swaps unused data onto designated hard disk space

improves flexibility but is slower than RAM to which the processor has direct access

Virtual Memory Management

Main memory is divided into **frames**, often 4KB.

The executable program is similarly divided into frame-sized chunks known as **pages**.

When a program is invoked not all the pages are loaded into main memory, only sufficient to get it started — entry point of the program.

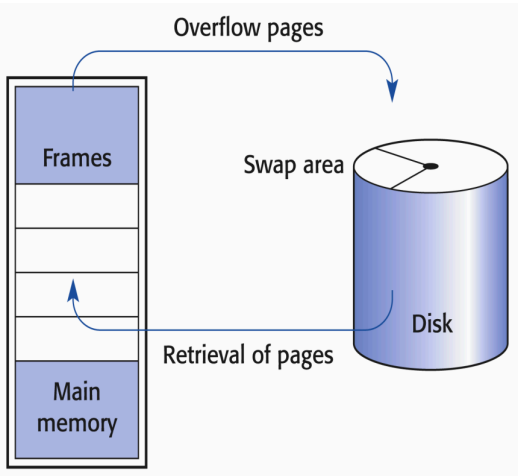
The rest are copied into the swap area.

When an instruction is needed from a page not yet in the main memory it is loaded from the disk.

If no empty frames exist at the moment *the least used frame* is freed to allow the new pages to be loaded — this is called **swapping**.

Main memory → Hard disk Swap area : **Page Out**

Main memory ← Hard disk Swap area : **Page In**

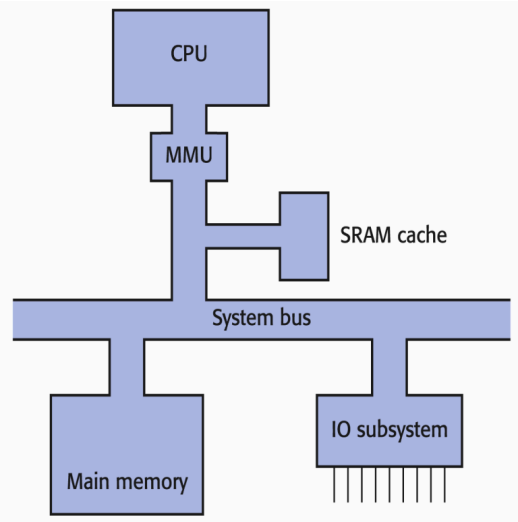


Virtual Memory Addressing

Within an user program addresses are in a form of 32bit **logical** address.
In the case of 4KB paging system:

Upper 20 bits serve as the 'page number'.
Lower 12 bits are 'address within a page / page offset'.

Memory Management Unit maps logical addresses into references to frame numbers and addresses within the frames.



Q&A Lecture 21

1. When you save to this, your data will remain intact even when the computer is turned off.
   a. RAM
   b. motherboard
   c. secondary storage device
   d. primary storage device
2. This data access method will slow down the process of data retrieval
   a. Direct access storage
   b. Sequential storage
   c. Random access storage
3. When you retrieve a file from secondary storage and display it on the screen,
   a. you are actually retrieving a copy of the desired file and putting it on the desktop.
   b. an old version of the file remains in secondary storage.
   c. that file is then sent to ROM.
   d. if no file content is modified after retrieval, the original file will not be replaced when you finish.

Lecture 22

Building computers from logic

Computer systems may be described at different levels of understanding.

At the **architectural** level the computer is described as a machine for executing instructions:

This level is most appropriate for understanding how programs are executed.

The layers **below** the architectural model is the **engineering level**

Engineering level

Engineering model of the computer represents the machine as a complex electrical circuit.

Within the circuit there are a large number of physical connections, along each of which a current may flow during the operation of the machine.

Presence of a current is used to represent transmission of the binary digit **1**, while

Absence of a current represents a value **0**.

Digital Systems

Digital systems based on electronic circuitry

1s and 0s, or on and off

Each 1 or 0 is called a bit; or binary digit

Computers use digital data representation

Analog Systems

Continuous variable values, along a range, such as

Temperature, Pressure values

Traditional analog recording devices are

Humidity recorders, Mercury thermometers, Pressure Gauges

Standard telephone lines transmit analog signals

Error resilient.

Boolean Operations and Boolean Gates

All operations that computer perform may be defined in terms of **basic boolean functions**, **operating on bits**.

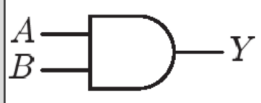
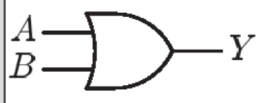
The digital electronic circuits and their components can be built from the devices implementing basic boolean operations — **boolean gates** (logic gates).

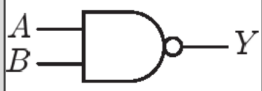
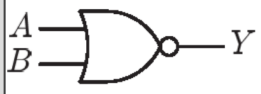
Boolean Gates

There is electronic **inputs** and **outputs** to the gates, along each of which a current may

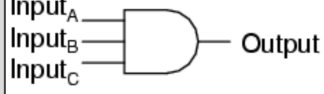
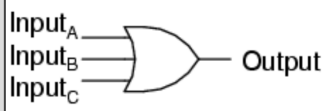
flow (boolean value 1) or may

not flow (boolean value 0).

Gate	Diagram	Equation	Alternative notations	Truth tables
AND Gate		$Y = A \text{ and } B$	$\wedge$, &	$0 \wedge 0 = 0$ $0 \wedge 1 = 0$ $1 \wedge 0 = 0$ $1 \wedge 1 = 1$
OR Gate		$Y = A \text{ or } B$	$\vee$	$0 \vee 0 = 0$ $0 \vee 1 = 1$ $1 \vee 0 = 1$ $1 \vee 1 = 1$
NOT Gate		$Y = \text{not } A$	$\neg$, ¯	$\neg 0 = 1$ $\neg 1 = 0$
XOR Gate		$Y = A \text{ xor } B$	| , $\oplus$	$0 \oplus 0 = 0$ $0 \oplus 1 = 1$ $1 \oplus 0 = 1$ $1 \oplus 1 = 0$

Gate	Diagram	Equation	Alternative notations	Truth tables
NAND gate		$Y = \text{not}(A \text{ and } B)$	$\bar{\wedge}$	$0 \bar{\wedge} 0 = 1$ $0 \bar{\wedge} 1 = 1$ $1 \bar{\wedge} 0 = 1$ $1 \bar{\wedge} 1 = 0$
NOR gate		$Y = \text{not}(A \text{ or } B)$	$\bar{\vee}$	$0 \bar{\vee} 0 = 1$ $0 \bar{\vee} 1 = 0$ $1 \bar{\vee} 0 = 0$ $1 \bar{\vee} 1 = 0$

Three-input gates:

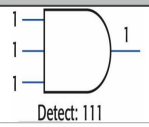
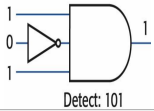
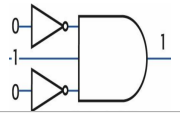
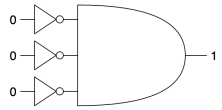
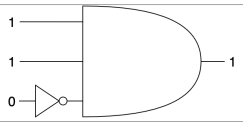
Gate	Diagram	Equation	Truth tables
3 input AND Gate		$Y = A \text{ and } B \text{ and } C$	$0 \wedge 0 \wedge 0 = 0$ $0 \wedge 0 \wedge 1 = 0$ $0 \wedge 1 \wedge 0 = 0$ $0 \wedge 1 \wedge 1 = 0$ $1 \wedge 0 \wedge 0 = 0$ $1 \wedge 0 \wedge 1 = 0$ $1 \wedge 1 \wedge 0 = 0$ $1 \wedge 1 \wedge 1 = 1$
3 input OR Gate		$Y = A \text{ or } B \text{ or } C$	$0 \vee 0 \vee 0 = 0$ $0 \vee 0 \vee 1 = 1$ $0 \vee 1 \vee 0 = 1$ $0 \vee 1 \vee 1 = 1$ $1 \vee 0 \vee 0 = 1$ $1 \vee 0 \vee 1 = 1$ $1 \vee 1 \vee 0 = 1$ $1 \vee 1 \vee 1 = 1$

Boolean Circuits

Elementary boolean gates can be combined into **boolean circuits**, implementing more complex **boolean functions** (operations)/

In fact, any boolean function can be implemented with a set of basic boolean gates.

Examples of circuits:

Function	Diagram	Equation
Detect 111		$Y = A \text{ and } B \text{ and } C$
Detect 101		$Y = A \text{ and } (\text{not } B) \text{ and } C$
Detect 010		$Y = (\text{not } A) \text{ and } B \text{ and } (\text{not } C)$
Detect 000		$Y = (\text{not } A) \text{ and } (\text{not } B) \text{ and } (\text{not } C)$
Detect 110		$Y = A \text{ and } B \text{ and } (\text{not } C)$

Data flow control circuit — Filter

D X	Out
0 0	0
0 1	0
1 0	0
1 1	1

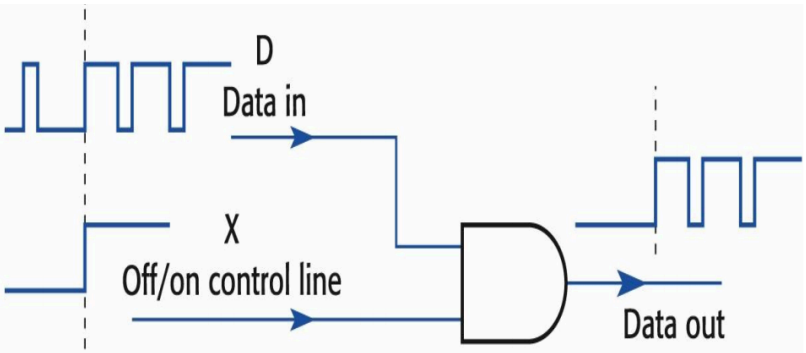
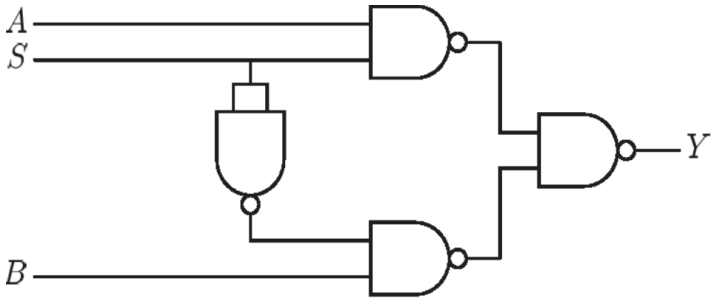


Fig. 4.4 Data flow control circuit.

Selector circuit



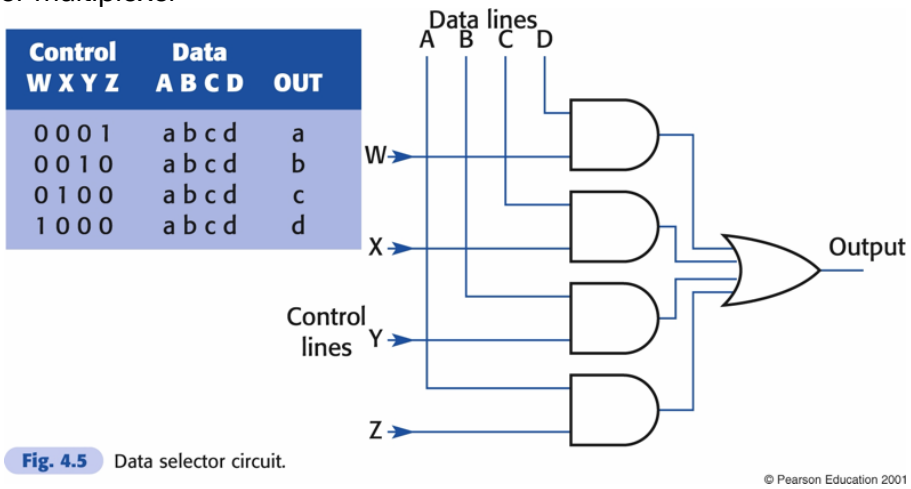
If S = 1 then Y = A

if S = 0 then Y = B

$$Y = (S \wedge A) \vee (\neg S \wedge B)$$
$$Y = \neg(\neg(A \wedge S) \wedge \neg(\neg S \wedge B))$$
$$Y = (A \barwedge S) \barwedge [(S \barwedge S) \barwedge B)]$$

S	A	B	$A \barwedge S$	$S \barwedge S$	$(S \barwedge S) \barwedge B$	Selector Circuit Output $Y = (A \barwedge S) \barwedge [(S \barwedge S) \barwedge B)]$
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	1	1	0
0	1	1	1	1	0	1
1	0	0	1	0	1	0
1	0	1	1	0	1	0
1	1	0	0	0	1	1
1	1	1	0	0	1	1

Data selector, or multiplexer



$$O = (A \wedge Z) \vee (B \wedge Y) \vee (C \wedge X) \vee (D \wedge W)$$

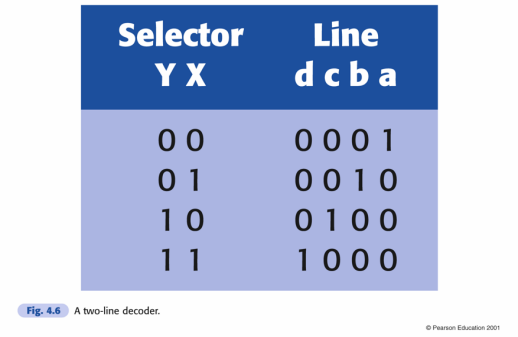
Problem with the circuit:

When control line has more then one signal 1the circuit does not behave as a selector

This problem can be solve using a **two-line decoder** to replace four control lines.

Two-line decoder

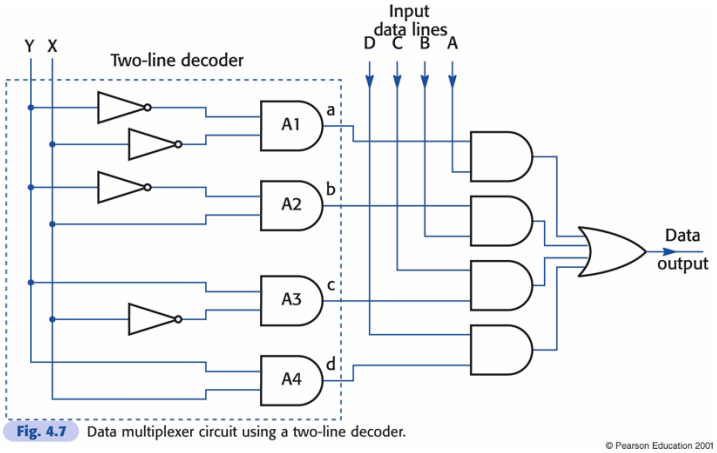
Two-line decoder



Implementation of the two-line decoder:

Detect pattern 00 (on YX lines), output result from line A
Detect pattern 01 (on YX lines), output result from line B
Detect pattern 10 (on YX lines), output result from line C
Detect pattern 11 (on YX lines), output result from line D

Data selector with two-line decoder



$$O = (A \wedge \neg X \wedge \neg Y) \vee (B \wedge X \wedge \neg Y) \vee (C \wedge \neg X \wedge Y) \vee (D \wedge X \wedge Y)$$

Cost comparison in gate count

Multiplexer: 4 AND gates + 1 OR gate.

2-line decoder: 8 AND gates + 1 OR gate + 4 NOT gates.

Implementing a function

Given a truth table for a logic function, to implement the function by a logic circuit one may proceed as follows:

Implement detectors (i.e. using AND/NOT gates) for all inout patterns on which the function gives the output 1/

Connect the outputs of all detectors to the input by an OR gate.

Truth Table				
i_1	i_2	i_3	i_4	O
1	**1**	**0**	**0**	1
1	**1**	**0**	**1**	1
0	**0**	**1**	**0**	1
1	**0**	**1**	**0**	1
0	**1**	**0**	**1**	1
All other configurations…				0

↓

Short form				
i_1	i_2	i_3	i_4	O
1	**1**	**0**	*	1
*	**0**	**1**	**0**	1
0	**1**	**0**	**1**	1

Logic expression:

$$O = (i_1 \wedge i_2 \wedge \neg i_3) \vee (\neg i_2 \wedge i_3 \wedge \neg i_4) \vee (\neg i_1 \wedge i_2 \wedge \neg i_3 \wedge i_4).$$

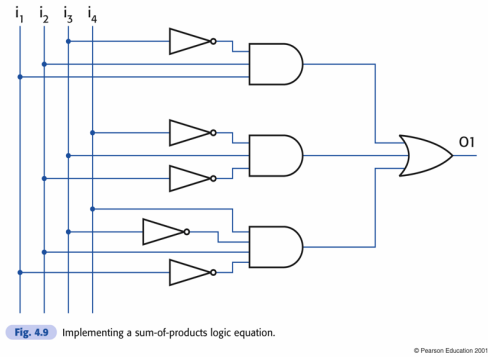


Fig. 4.9 Implementing a sum-of-products logic equation.

Example implementation

i_1	i_2	i_3	i_4	O
1	**1**	**0**	**0**	1
*	**0**	**1**	**0**	1
All other configurations…				0

$$O = (i_1 \wedge i_2 \wedge \neg i_3 \wedge \neg i_4) \vee (\neg i_2 \wedge i_3 \wedge \neg i_4)$$

Doing arithmetic via logic

Binary addition. For the addition of single-bit binary numbers: (Half-)adder

Input		Output	
X	**Y**	**C**	**S**
0	**0**	0	0
0	**1**	0	1
1	**0**	0	1
1	**1**	1	0

Adder v.1 $\begin{cases} C = X \wedge Y \\ S = \neg C \wedge (X \vee Y) \end{cases}$

Adder v.2 $\begin{cases} C = X \wedge Y \\ S = (\neg X \wedge Y) \vee (X \wedge \neg Y) \end{cases}$

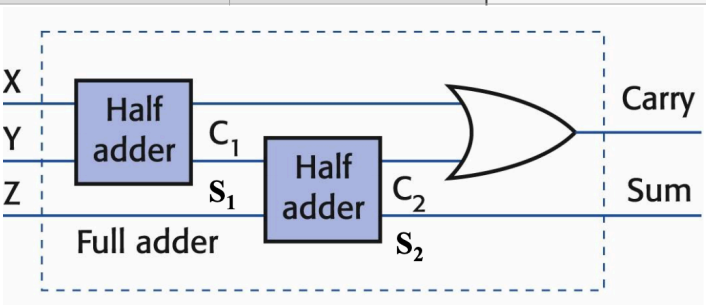
Adder v.3 $\begin{cases} C = \neg(X \bar{\wedge} Y) \\ S = [X \wedge (X \bar{\wedge} Y)] \vee [Y \wedge (X \bar{\wedge} Y)] \end{cases}$

Adder v.4 $\begin{cases} C = X \wedge Y \\ S = X \bar{\vee} Y \end{cases}$

Full adder

For the addition of multi-bit binary numbers one needs to deal with carry-in from previous stage, that means the adder should have 3 inputs.

Input			Output	
X	**Y**	**Z**	**C**	**S**
0	**0**	**0**	0	0
0	**0**	**1**	0	1
0	**1**	**0**	0	1
0	**1**	**1**	1	0
1	**0**	**0**	0	1
1	**0**	**1**	1	0
1	**1**	**0**	1	0
1	**1**	**1**	1	1



Sequential logic circuits

Combinational (combinatorial) logic circuits.

In all Boolean circuits that we have seen so far, the output at any moment depends only on the input at the same moment.

There are **no memory.**

Sequential logic circuits.

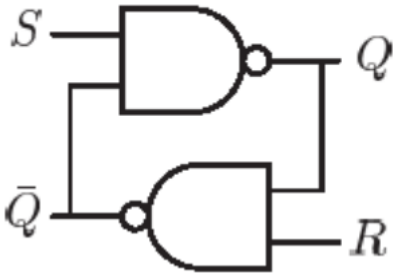
The output depends also on the state of the circuit. The state of the circuit is somehow stored within the circuit.

There are **memory.**

Flip-flops, or latches

The basic memory element of the sequential circuits is called a **flip-flop**, or **latch**.

The simplest flip-flop is made up of two NAND gates. It is called set-reset (SR) flip-flop.



Inputs are S and R while outputs are Q and $\bar{Q}$.

SR Flip-flop

Suppose that S and R are both initially set to 1, then neither Q nor $\bar{Q}$ can be determined.

Two variants are possible:

$Q = 1$ and $\bar{Q} = 0$

or

$Q = 0$ and $\bar{Q} = 1$

Everything is consistent with any of these variants.

If both input is fixed to be 1, then the circuit may have one of the two possible **states**.

The circuit is **stable** as long as R and S remain at 1.

Assume the flip-flop is in a stable state with $Q = 0$.

If R is turned to become 0, This forces changing the value of Q to 0.

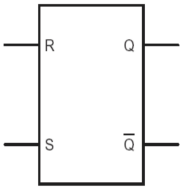
The circuit will stay in this state even after input R returns to 1.

The flip-flop **switched** the states.

If then input S momentarily becomes 0 the system will switch the state back.

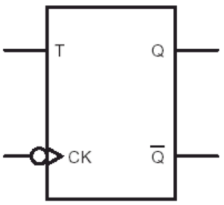
Thus, SR flip-flop **remembers** which input was set to 0 last.

One constraintL the logic surrounding this flip-flop must avoid situation where both R and S are 0 at the same time.

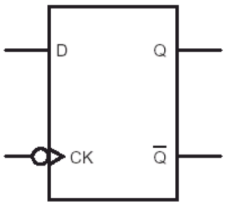
State table of SR flip-flop			SR Representation
R	S	Q	
0	**0**	?	
0	**1**	0	
1	**0**	1	
1	**1**	Q_{prev}	FlipFlop(SR)

Other types of flip-flops

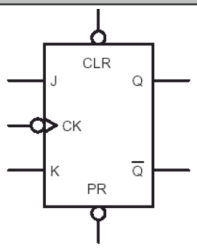
All these flip-flops has an input marked for **clock**. The new output occurs when the clock is pulsed, i.e. momentarily changed form 1 to 0.
CLR (clear) and PR (preset) inputs are used to initialize the flip-flop to known value (0 and 1, respectively).

Toggle		**SR Representation**
T	Q	
0	Q_{prev}	
1	$\bar{Q}_{prev}$	

Toggle

D flip-flop		**SR Representation**
D	Q	
0	0	
1	1	

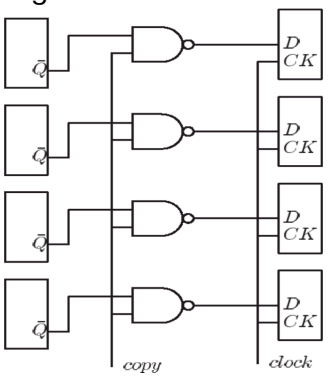
D flip-flop

J-K flip-flop			**SR Representation**
J	K	Q	
0	**0**	Q_{prev}	
0	**1**	0	
1	**0**	1	
1	**1**	$\bar{Q}_{prev}$	

J-K flip-flop

Use of D flip-flops — Copying data

Circuit to copy data from one register to another.



Q&A Lecture 24

1. What type of flip-flop has an illegal state?
   SR flip-flops when both S and R are 0.
2. What type of flip-flop allow us to copy data
   D flip-flops
3. What type of circuit has 'memory'?
   Sequential circuits.