COM 6062

Network and Internetwork Architectures

Part 1: Computer System

- Data representation.
  - integer representations / arithmetic.
  - Boolean algebra.
- Computer architecture.
  - operating system (OS).
  - scheduling and memory management.
  - stored program architecture.
  - CPU instruction set and addressing modes.

Decimal System

Base 10 (or **radix 10**) system.

Decimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

$$(659)_{10} = (6 \times 10^2) + (5 \times 10^1) + (9 \times 10^0)$$

$$(26035)_{10} = (2 \times 10^4) + (6 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (5 \times 10^0)$$

Fractional numbers:

$$(0.09)_{10} = (0 \times 10^0) + (0 \times 10^{-1}) + (9 \times 10^{-2})$$

$$(3.142)_{10} = (3 \times 10^0) + (1 \times 10^{-1}) + (4 \times 10^{-2}) + (2 \times 10^{-3})$$

In general,

$$(\cdots d_2 d_1 d_0 \,.\, d_{-1}\, d_{-2} \cdots)_{10} = \sum_i d_i \times 10^i$$

Binary System

Base 2 (or **radix 2**) system.

Binary digits: 0, 1.

$$(1011)_2 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = (11)_{10}$$

$$(101101)_2 = (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = (45)_{10}$$

Fractional numbers:

$$(0.1001)_2 = (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4}) = (0.5625)_{10}$$

$$(1001.1001)_2 = (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4}) = (9.5625)_{10}$$

In general,

$$(\cdots b_2 b_1 b_0 \,.\, b_{-1}\, b_{-2} \cdots)_2 = \sum_i b_i \times 2^i$$

Binary ⇔ Decimal Conversion

Binary to decimal:

multiply each binary digit by the appropriate power of 2, then add.

Decimal to binary:

41	1
20	0
10	0
5	1
2	0
1	0
0	1

$(41)_{10} = (101001)_2$

$(0.375)_{10} = (0.011)_2$

0	.375
	x 2
0	.750
	x 2
1	.500
	x 2
1	.000

Powers of Two

binary	decimal		binary	decimal				
1 1 ••• 1	$2^d - 1$		1 0 0 ••• 0	2^d				
d one's			d zero's					
	$2^0 - 1$	0		2^0	1			
	$2^1 - 1$	1		2^1	2			
	$2^2 - 1$	3		2^2	4			
	$2^3 - 1$	7		2^3	8			
	$2^4 - 1$	15		2^4	16			
	$2^5 - 1$	31		2^5	32			
	$2^6 - 1$	63		2^6	64			
	$2^7 - 1$	127		2^7	128			
	$2^8 - 1$	255		2^8	256			
	$2^9 - 1$	511		2^9	512	kilo		
	$2^{10} - 1$	1023		2^{10}	1024	1k		
	$2^{11} - 1$	2047		2^{11}	2048	2k		
	$2^{12} - 1$	4095		2^{12}	4096	4k	mega	
	$2^{19} - 1$	524287		2^{19}	524288	512k	0.5M	
	$2^{20} - 1$	1048575		2^{20}	1048576	1024k	1M	
	$2^{21} - 1$	2097151		2^{21}	2097152	2048k	2M	giga
	$2^{29} - 1$	•		2^{29}	•	•	512M	0.5G
	$2^{30} - 1$	•		2^{30}	•	•	1024M	1G
	$2^{31} - 1$	•		2^{31}	•	•	2048M	2G

Hexadecimal System

Base 16 (or **radix 16**) system.

Hexadecimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

bin	hex	bin	hex	bin	hex	bin	hex
0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

Shorthand version of binary:

$$(11\,1101\,1110)_2 = (3DE)_{16}$$

$$(9B)_{16} = (1001\,1011)_2$$

$$(C8)_{16} = ((C)_{16} \times 16^1) + ((8)_{16} \times 16^0)$$

$$= ((12)_{10} \times 16^1) + ((8)_{10} \times 16^0) = (200)_{10}$$

$$(41)_{10} = (101001)_2 = (29)_{16}$$

$$(123)_{10} = (1111011)_2 = (7B)_{16}$$

Exercise

$(10\,0011)_2$

$(1001\,1000)_2$

$(101\,0010\,1001)_2$

$(0.01)_2$

$(0.111)_2$

$(11.01)_2$

$(7D)_{16}$

$(13A)_{16}$

$(CB13)_{16}$

$(2.8)_{16}$

$(12.C)_{16}$

$(2000)_{10}$

$(3.5)_{10}$

$(10.40625)_{10}$

$(19.3125)_{10}$

$(0.1)_{10}$

$(0.0012)_{10}$

Integers

Unsigned integer:

- positive integers, and zero.
  e.g., $(+170)_{10}$ is expressed as

$(1010\,1010)_2$: 8 bits

$(0000\,0000\,1010\,1010)_2$: 16 bits

$(0000\,0000\,0000\,0000\,0000\,0000\,1010\,1010)_2$: 32 bits

One's and **two's complements**:

- positive and negative integers, and zero.

One's Complement

For binary numbers having d digits,

- one's complement of N is $(2^d - 1) - N$:

$$(+42)_{10} = (0010\ 1010)_2 \quad : 8 \text{ bits}$$
$$(-42)_{10} = (1101\ 0101)_2 \quad : 8 \text{ bits}$$

  i.e., negative numbers are represented by bit-by-bit complementation of positive magnitude ($0 \to 1$, $1 \to 0$).

Drawback:

- there exist two representations of zeros.

Two's Complement

For binary numbers having d digits,

- the two's complement of N is $2^d - N$:

$$(+42)_{10} = (0010\ 1010)_2$$
$$(-42)_{10} = (1101\ 0110)_2$$

  i.e., obtained by adding one to the **one's complement**.

- extension of word length:

$$(0010\ 1010)_2 \qquad (1101\ 0110)_2 \quad : 8 \text{ bits}$$
$$(0000\ 0000\ 0010\ 1010)_2 \qquad (1111\ 1111\ 1101\ 0110)_2 \quad : 16 \text{ bits}$$

Benefit:

- simple arithmetic operation.
- only one representation of zero.

Binary Expressions for Integers

decimal	unsigned integer	one's complement	two's complement
256	—		
255	1111 1111		
254	1111 1110		
⋮	⋮		
128	1000 0000		
127	0111 1111	0111 1111	0111 1111
126	0111 1110	0111 1110	0111 1110
⋮	⋮	⋮	⋮
2	0000 0010	0000 0010	0000 0010
1	0000 0001	0000 0001	0000 0001
0	0000 0000	0000 0000	0000 0000
- 0		1111 1111	—
- 1		1111 1110	1111 1111
- 2		1111 1101	1111 1110
⋮		⋮	⋮
-126		1000 0001	1000 0010
-127		1000 0000	1000 0001
-128		—	1000 0000

Integer Addition and Subtraction

Additions and subtractions of signed integers:

- use the **two's complements**.
- subtraction:

$$a - b = a + (-b)$$

i.e., take the **two's complement** of the subtracter first, then achieve addition.

Integer Addition and Subtraction (2)

Example 1: 8 bits are sufficient for relatively small numbers.

+6	0 0 0 0 0 1 1 0	-6	1 1 1 1 1 0 1 0
+13	0 0 0 0 1 1 0 1	+13	0 0 0 0 1 1 0 1
+19	0 0 0 1 0 0 1 1	+7	0 0 0 0 0 1 1 1
+6	0 0 0 0 0 1 1 0	-6	1 1 1 1 1 0 1 0
-13	1 1 1 1 0 0 1 1	-13	1 1 1 1 0 0 1 1
-7	1 1 1 1 1 0 0 1	-19	1 1 1 0 1 1 0 1

Example 2:

+38	0 0 1 0 0 1 1 0	-38	1 1 0 1 1 0 1 0
+29	0 0 0 1 1 1 0 1	+29	0 0 0 1 1 1 0 1
+67	0 1 0 0 0 0 1 1	-9	1 1 1 1 0 1 1 1
+38	0 0 1 0 0 1 1 0	-38	1 1 0 1 1 0 1 0
-29	1 1 1 0 0 0 1 1	-29	1 1 1 0 0 0 1 1
+9	0 0 0 0 1 0 0 1	-67	1 0 1 1 1 1 0 1

Integer Addition and Subtraction (3)

Example 3: 8 bits are not sufficient if numbers are lager.

+38	0 0 1 0 0 1 1 0
+90	0 1 0 1 1 0 1 0
-128	1 0 0 0 0 0 0 0 ⬅ overflow occurred

		-38	1 1 0 1 1 0 1 0
		-90	1 0 1 0 0 1 1 0
no overflow ➡		-128	1 0 0 0 0 0 0 0

Example 4:

+38	0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0
+90	0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0
+128	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

-38	1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0
-90	1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 0
-128	1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0

Integer Addition and Subtraction (4)

Exercise —

- do two's complement binary additions:

$$\pm (2)_{10} \text{ and } \pm (3)_{10}$$
$$\pm (19)_{10} \text{ and } \pm (10)_{10}$$
$$\pm (23)_{10} \text{ and } \pm (83)_{10}$$

- how many bits are required for correct calculation?

Integer Multiplication

Suppose the multiplier is a positive number:

9	1 0 0 1		-7	1 0 0 1	← multiplicand
x 5	0 1 0 1		x 5	0 1 0 1	← multiplier
9×2^0	1 0 0 1		$(-7) \times 2^0$	1 1 1 1 1 0 0 1	
$+ 9 \times 2^2$	1 0 0 1		$+ (-7) \times 2^2$	1 1 1 0 0 1	
45	0 0 1 0 1 1 0 1		-35	1 1 0 1 1 1 0 1	← product
unsigned integers			two's complement integers		

(note) a product of two n-bit integers is at most 2n-bit.

Suppose the multiplier is negative:

- one possible approach:
  1. convert the multiplier to positive,
  2. do multiplication, and finally
  3. change the sign of the product.
- alternative:

  Booth's algorithm (which works on any positive/negative combination of two integers).

Integer Division

Division between unsigned integers:

```
                                             1 1 0 1  <- quotient
         1 3                     divisor -> 1 0 1 1 / 1 0 0 1 0 0 0 0  <- dividend
  1 1 / 1 4 4                                      1 0 1 1
        1 1                                        -------
        ---                                        0 0 1 1 1 0
          3 4                                          1 0 1 1
          3 3                                          -------
          ---                                          0 0 1 1 0 0
            1                                              1 0 1 1
                                                           -------
                                                             0 0 1  <- remainder
```


Boolean Operators

Dealing with binary variables:

true	on	1	active
false	off	0	not active

Basic logical operators:

NOT : $\overline{A}$, $\tilde{A}$, $\sim A$, $/A$.

AND : AB, A AND B, $A.B$, $A * B$, $A \bullet B$, $A \cap B$, $A \wedge B$.

OR : $A + B$, A OR B, $A \cup B$, $A \vee B$.

NAND : $\overline{AB}$.

NOR : $\overline{A + B}$.

XOR : $A \oplus B = A\overline{B} + \overline{A}B$.

XNOR : $\overline{A \oplus B}$.

Truth Table

NOT

A	$\overline{A}$
0	1
1	0

AND

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Boolean algebra

negative

A	-A
0	0
1	-1

multiplication

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

addition

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	2

usual algebra

A	B	$\overline{A}$	$\overline{B}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
0	0	1	1	0	1	0	1	0	1
0	1	1	0	0	1	1	0	1	0
1	0	0	1	0	1	1	0	1	0
1	1	0	0	1	0	1	0	0	1

summary

Boolean Operation

No carries:

NOT	1	1	1	0	0	0	1	1
	0	0	0	1	1	1	0	0

	0	0	1	0	0	1	1	0
AND	1	1	1	0	0	0	1	1
	0	0	1	0	0	0	1	0

	0	0	1	0	0	1	1	0
OR	1	1	1	0	0	0	1	1
	1	1	1	0	0	1	1	1

Exercise —

$A = (0101\,0010)_2, B = (0010\,1100)_2, C = (1001\,1000)_2$:

NOT A

A AND B

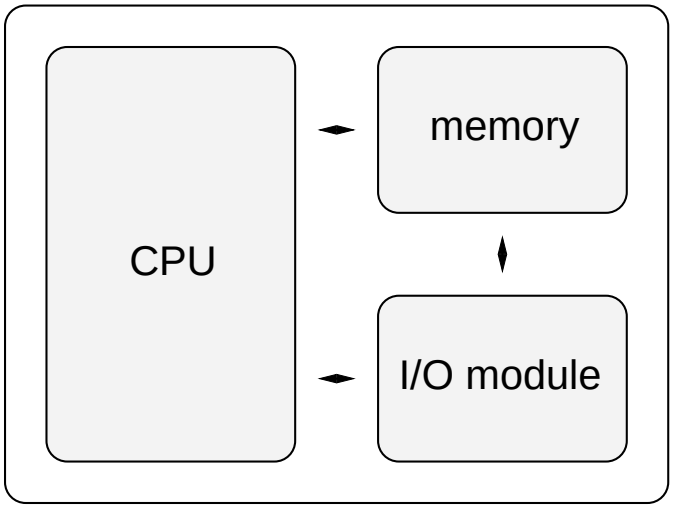
(A AND B) OR C

B OR C

A AND (NOT (B OR C))

B XOR (NOT C)

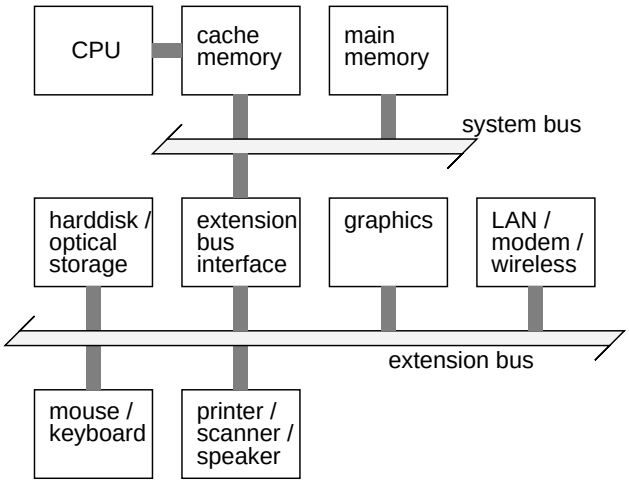
Computer Model



I/O Modules

Connection between peripherals $\Longleftrightarrow$ CPU and internal memories:

- adjust data format.
- interface through the system bus.



Memory System

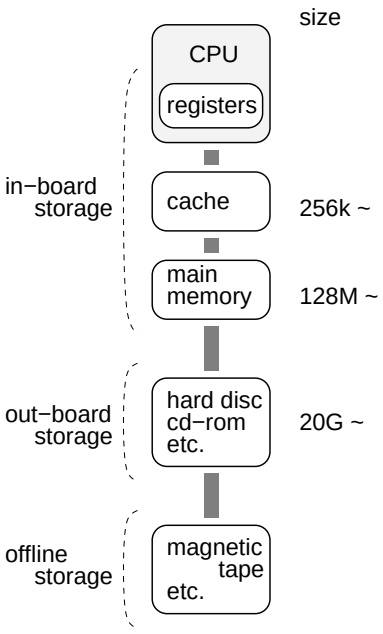
General consideration:

- faster access, greater cost per bit.
- greater capacity, smaller cost per bit, but slower access.

Principle of locality

> "during the program execution, memory reference by a processor tends to cluster."

We wish to organise memory hierarchy such that access to the lower level storage is substantially less than the upper level.



. memory hierarchy

Memory System (2)

Main memory:

- **RAM** (random access memory) —read / write memory.
- **ROM** (read only memory): system program, function table, libraries.

Magnetic disk:

- read and write capabilities. (*e.g.*) hard disk drive, floppy disk.
- large capacity, and relatively fast.

Optical disk:

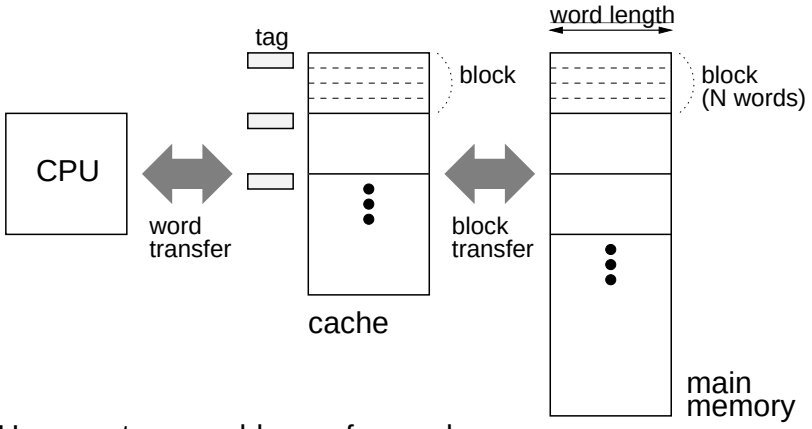
- often read only, but some with erasable and writable capabilities.
- **CDrom**: digital audio compact disk (CD) plus error correction scheme.
- very large capacity (about 800M bytes per CDrom) but slow access.

Magnetic tape:

- offline storage with huge capacity. Typically for backup purpose.

Cache: Principles

- faster access to memory.
- typically of the size between 1k and 512k words.



Once the CPU generates an address of a word,

- if the word is in the cache (hit), it is sent to the CPU.
- if not (miss), then the block containing that word is loaded to the cache, and the word is delivered to the CPU.

Operating System: Objective

Resource management:

- allowing the computer system resources to be used efficiently.

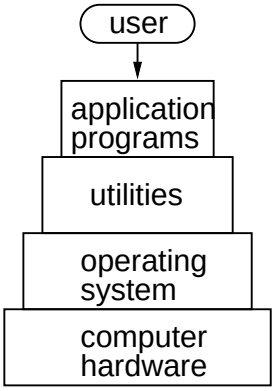
User and computer hardware interface:

- **program execution**: loading instruction and data to main memory, initialising I/O devices and files, *etc.*
- hiding a particular nature of each I/O device (*e.g.*, disk, graphics) from a user and simplifying its operation.
- **system access**: controlling access to the system by shared users (or programs), protecting resources from unauthorised usage.

Operating System is a Program

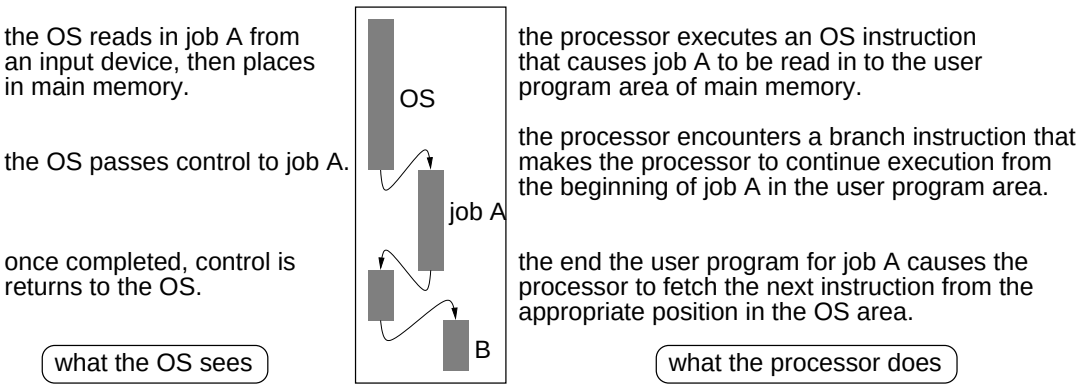
OS is nothing more than a computer program —

- **same** as other programs:
  - residing in the main memory.
  - executed by the processor.
  - frequently giving up control for the other program to be executed.
  - relying on the processor to regain control.
- but the purpose is **different**:
  - directing the processor in the use of the system resources and in the execution timing of other programs.



computer system

Batch Operating Systems



Some desirable hardware features for OS support:

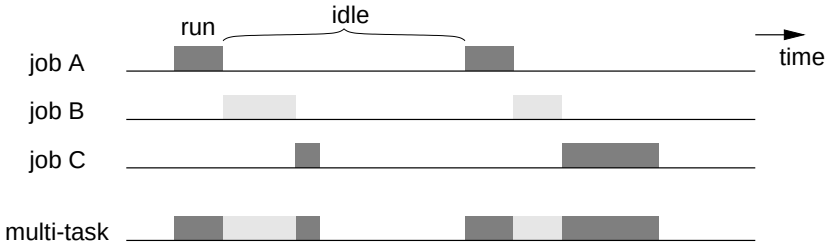
- protection of memory area where the OS resides.
- timer to prevent a single job from monopolising the system.
- privileged instructions for the OS (*e.g.*, I/O instructions).
- interrupts for giving up/regain control over the processor.

Multi-tasking Operating Systems

Batch OS: the processor is often idle for many instructions (*e.g.*, I/O).

Multi-tasking OS: while waiting for one job to complete an I/O instruction, the processor can switch to the other job.

- in order to have several jobs ready to run, jobs must be kept in main memory $\Longrightarrow$ **memory management**.
- the OS must decide which job to run $\Longrightarrow$ **scheduling**.
- called '**time sharing**', if it handles multiple interactive jobs (or users).



Scheduling

Long-term scheduler controls the number of processes in memory.

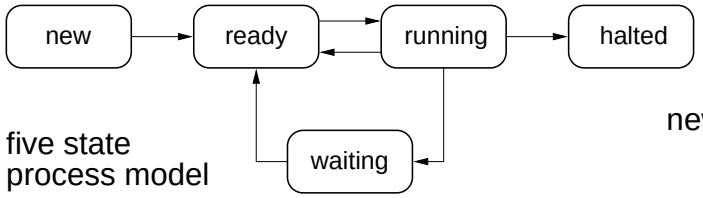
- determines which programs are admitted to the system for processing.
- once admitted, it becomes a process and added to a queue for the **short-term scheduler**.
- for **time sharing** system, the OS will accept all authorised interactive jobs (or users) up to pre-defined saturation level.

Medium-term scheduler is part of swapping in **memory management**.

Short-term scheduler makes fine-grained decision of which available process to execute next.

I/O scheduler decides which process's pending I/O request shall be handled by an available I/O device.

Short-term Scheduler



five state process model

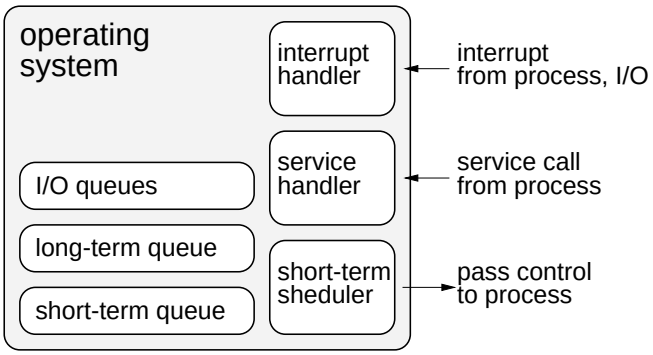
new: a program is admitted by the long-term scheduler, but is not yet ready to execute.

ready: the process is ready to execute, and is awaiting access to the processor.

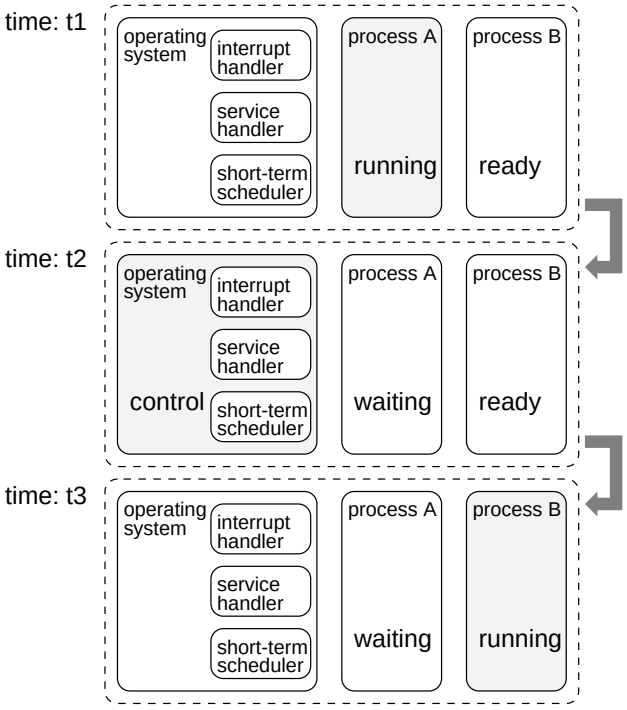
running: the process is being executed by the processor.

waiting: the process is suspended from execution, waiting for some system resources, such as I/O.

halted: the process has terminated and will be destroyed by the operating system.



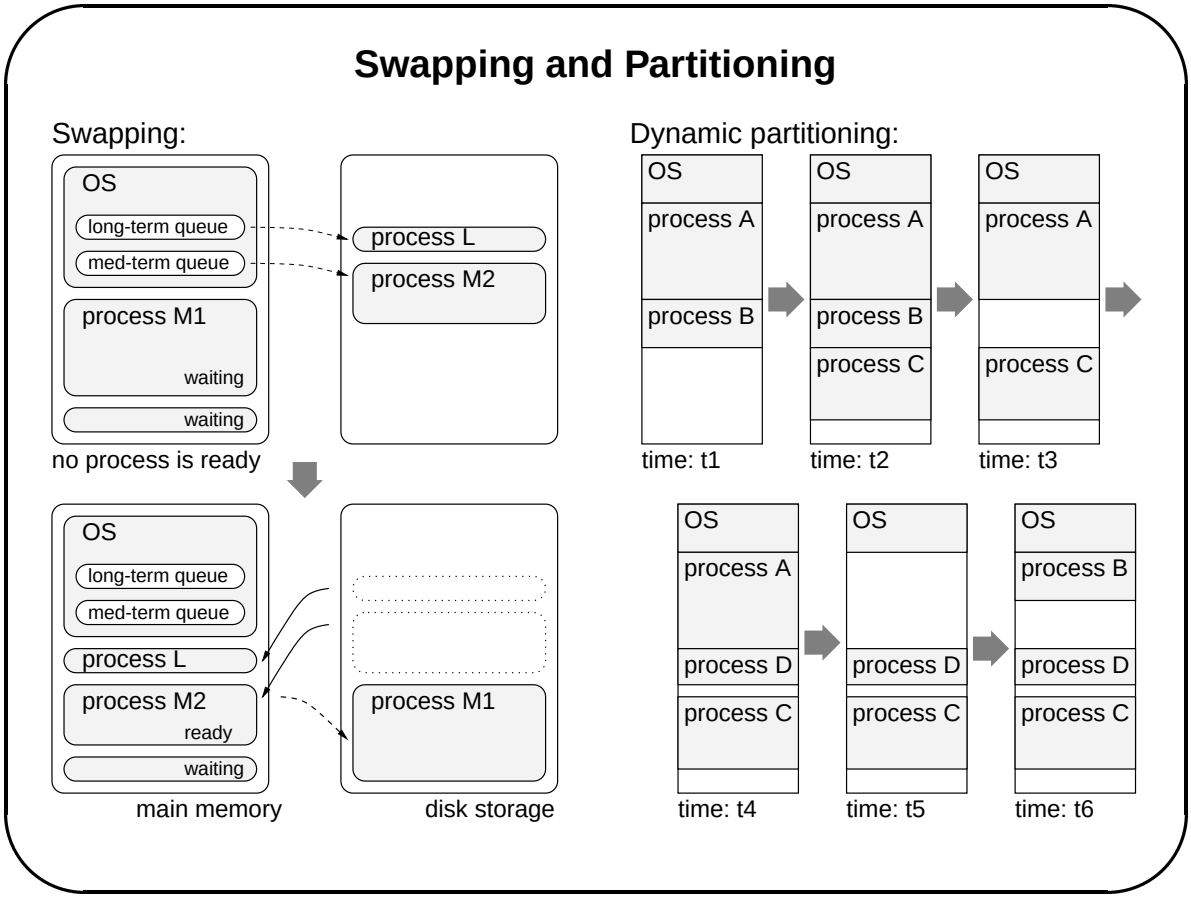
Scheduling Example

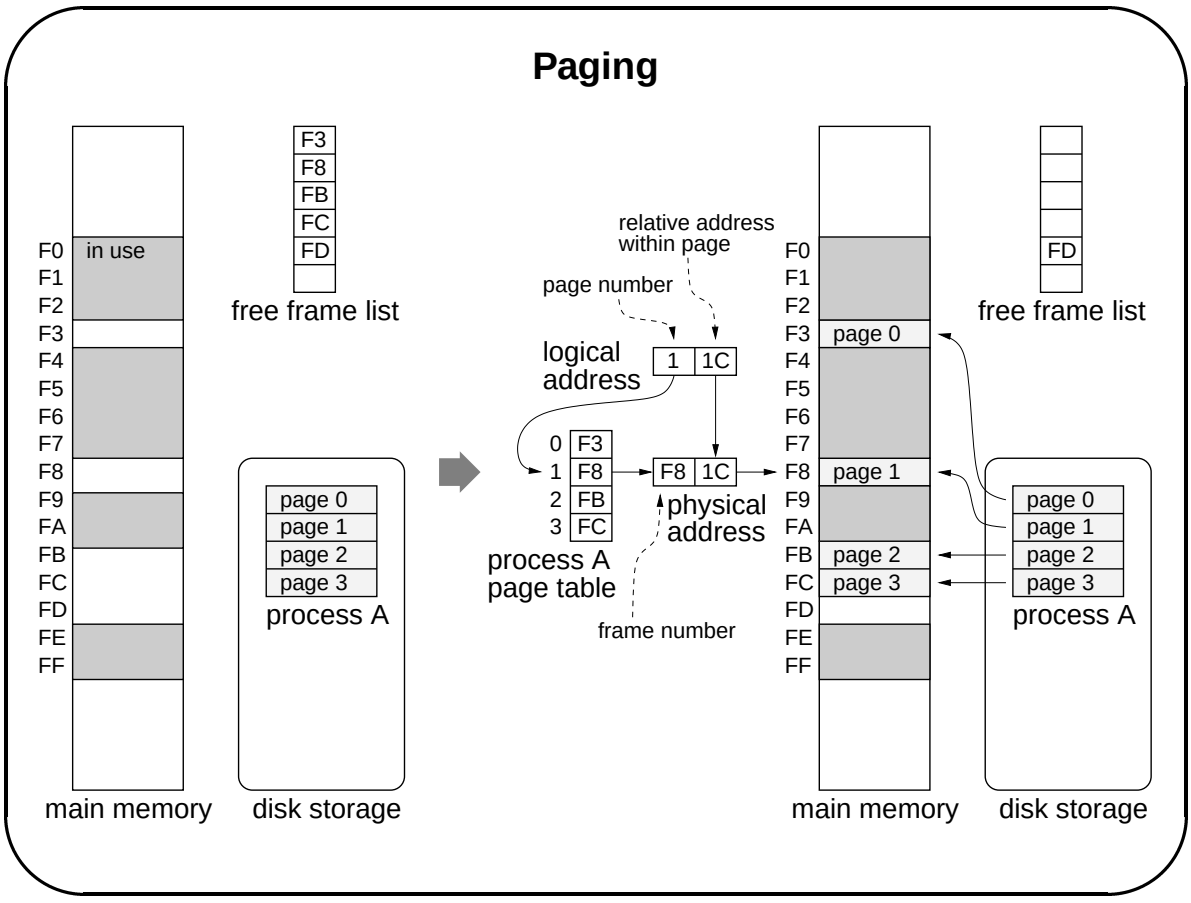


Possible reasons for transition of control: process A -> OS

1. Process A issues a service call (eg, an I/O request) to the OS. Execution of A is suspended until this call is handled.
2. Process A causes an interrupt. It is a hardware generated signal, resulting from, eg, an error or timeout. When this signal is detected, execution is transferred to the interrupt handler.
3. Some event, not related to process A (eg, completion of I/O operation), that requires attention causes an interrupt.

Process B is chosen by the short-term scheduler.


Swapping and Partitioning
Swapping:
OS
long-term queue
med-term queue
process M1
waiting
waiting
process L
process M2
no process is ready
OS
long-term queue
med-term queue
process L
process M2
ready
waiting
process M1
main memory
disk storage
Dynamic partitioning:
OS
process A
process B
process C
time: t1
OS
process A
process B
process C
time: t2
OS
process A
process C
time: t3
OS
process A
process D
process C
time: t4
OS
process D
process C
time: t5
OS
process B
process D
process C
time: t6


Paging
F0
F1
F2
F3
F4
F5
F6
F7
F8
F9
FA
FB
FC
FD
FE
FF
in use
main memory
F3
F8
FB
FC
FD
free frame list
page 0
page 1
page 2
page 3
process A
disk storage
relative address
within page
page number
logical
address
1
1C
0 F3
1 F8
2 FB
3 FC
process A
page table
F8
1C
physical
address
frame number
page 0
page 1
page 2
page 3
FD
free frame list
page 0
page 1
page 2
page 3
process A
main memory
disk storage

Virtual Memory

real $\Longleftrightarrow$ **virtual**:

- the **principle of locality** indicates that only a few pages may be needed for execution at a time.
- instead of loading whole process, each page of a process is brought in to main memory when it is demanded (**on demand paging**).
- it has advantages of saving memory space and reducing loading time, but memory management scheme by the OS needs to be refined.
- this, in turn, implies that a process may possibly be larger than user area of main memory (or 'real memory') that physically exists.
- on the other hand, a user may execute a process, without recognising the **on demand paging** is at work and only a few pages actually reside within main memory (*i.e.*, 'virtual memory').

Virtual Memory (2)

Page table:

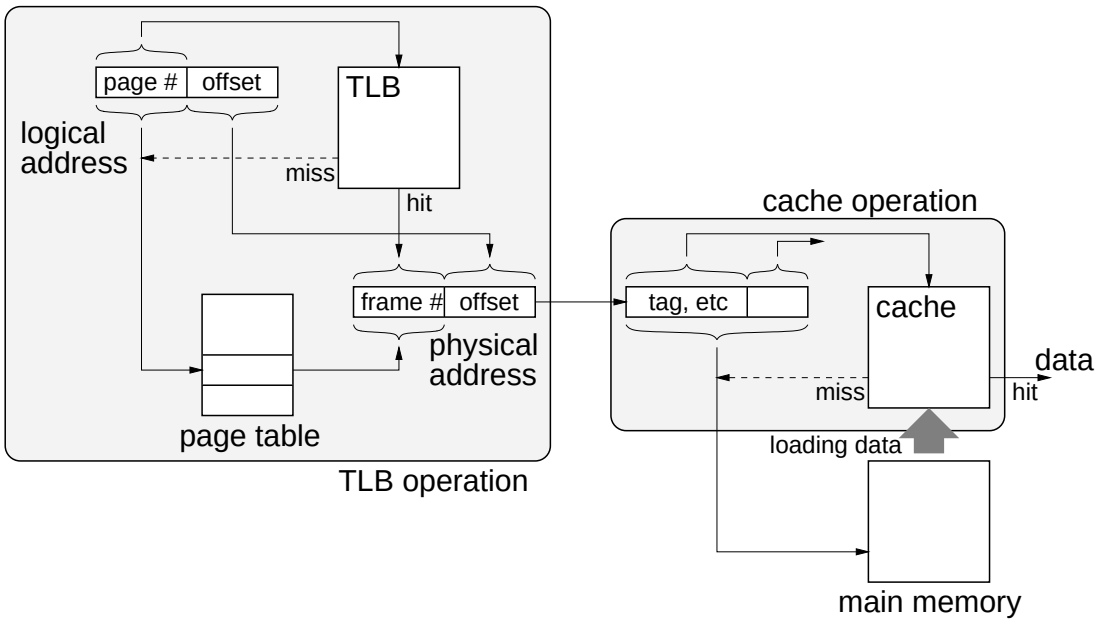
- page table translates a logical address (*i.e.*, page number and offset) into a physical address (*i.e.*, frame number and offset).
- (because each process can occupy huge amounts of virtual memory) a table size could be very large

  $\Longrightarrow$ store a page table in virtual memory.
- every virtual memory reference can cause two physical memory accesses (to fetch the page table entry and to get desired data).

  $\Longrightarrow$ use of a special cache for page table entries

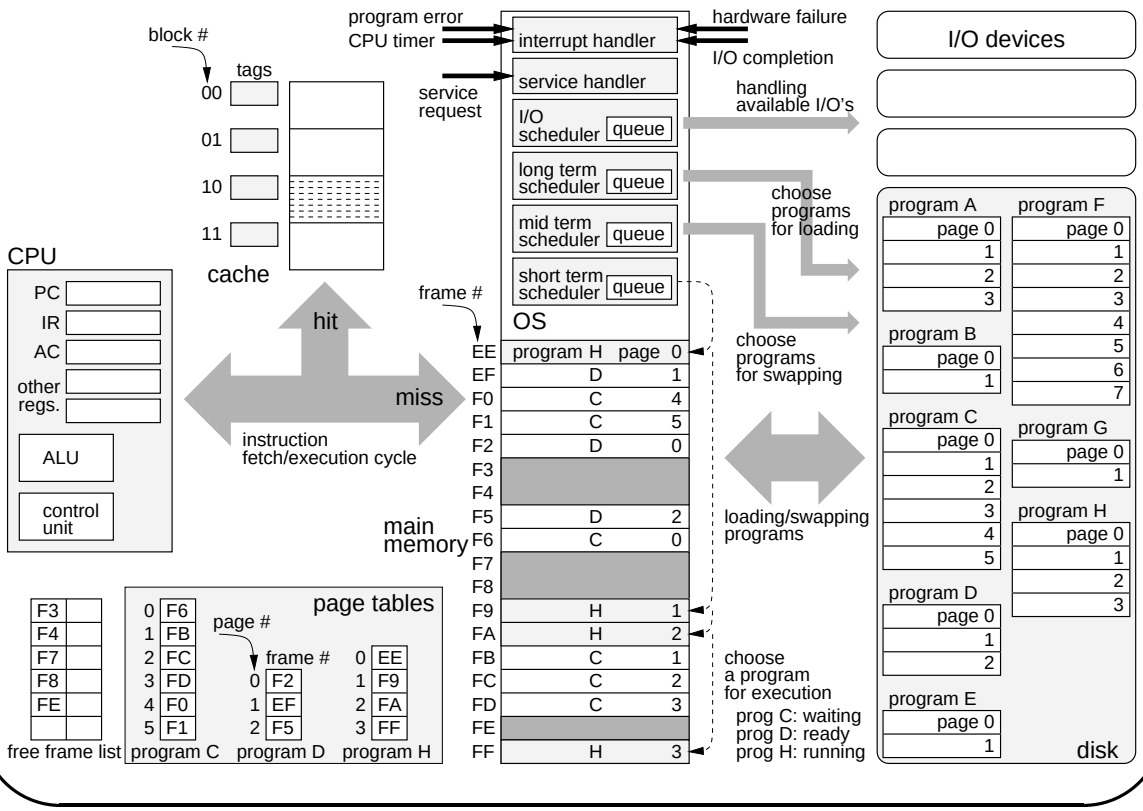
  (referred to as a **translation lookaside buffer**, or **TLB**).

Virtual Memory (3)



interaction of TLB and cache operations

Computer System



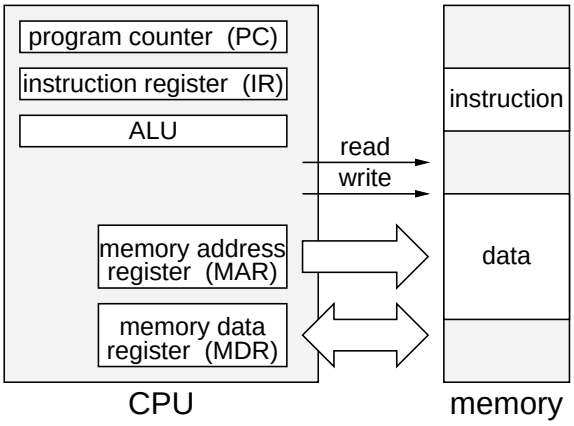
Stored Program Architecture

- Data and instructions are stored in a memory.
- Contents of this memory are addressable by location, regardless to the type of data contained there.
- execution occurs sequentially from one instruction to the next.

Alternatively known as **'von Neumann architecture'.**

To read from memory:
- set address to MAR.
- activate 'read' signal.
- instruction or data arrives to MDR.

To write to memory:
- set address to MAR.
- set data to MDR.
- activate 'write' signal.



Control Registers

Program counter (**PC**):

contains the address of an instruction to be fetched. PC is updated after each instruction fetch, unless otherwise required.

Instruction register (**IR**):

contains the instruction most recently fetched. The **opcode** and data reference(s) are analysed in IR.

Memory address register (**MAR**):

contains the address of memory.

Memory data register (**MDR**):

contains data to be written to memory, or data most recently read.

User Registers

General purpose registers:

can be assigned to a variety of operations. Some restriction may apply for some cases (*e.g.*, floating point registers).

Data registers:

may be used only to hold data, but cannot be employed in the calculation of a data address.

Address registers:

may be used for particular addressing modes (*e.g.*, **stack pointer**).

Flags (**condition codes**):

hold results of ALU operations (*e.g.*, sign, zero, carry, overflow). Flags are bits set by the processor. A user can only read.

Condition Code

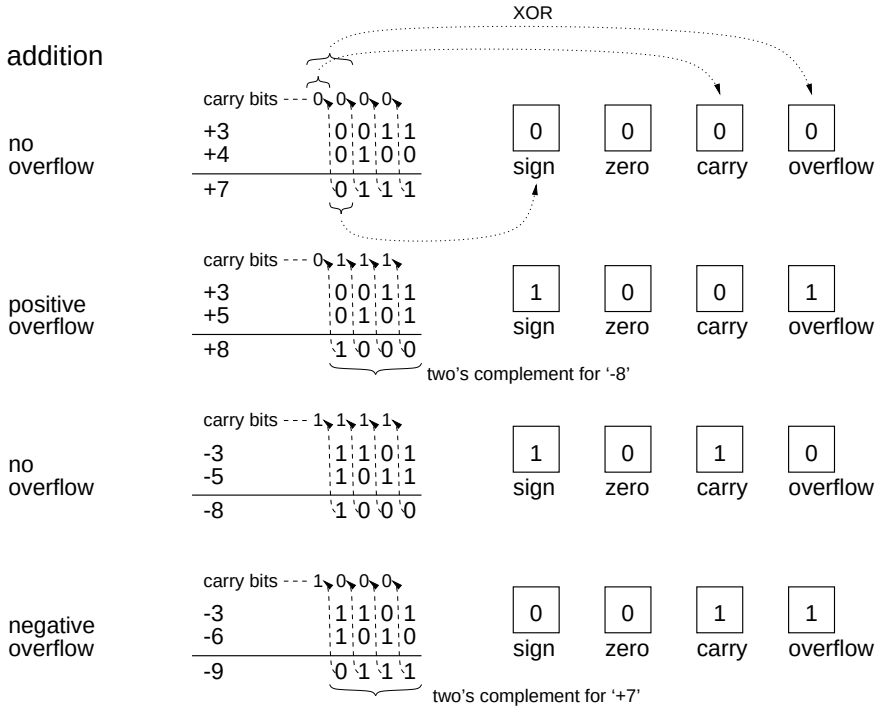
As a result of an operation performed in the ALU,

- **Sign** (**S**):
  is set to 1 if the highest order bit is 1,
- **Zero** (**Z**):
  is set to 1 if the output of the ALU contains all 0's,
- **Carry** (**C**):
  is set to 1 if the end carry is 1,
- **Overflow** (**O**):
  is set to 1 if an overflow occurs (an overflow is identified if the XOR of highest two carries is 1 — *i.e.*, one of them is 0 and the other is 1),

and cleared to 0, otherwise.

Carry and Overflow

4-bit machine
two's complement addition



Instruction Fetch / Execution Cycle

Fetch cycle:

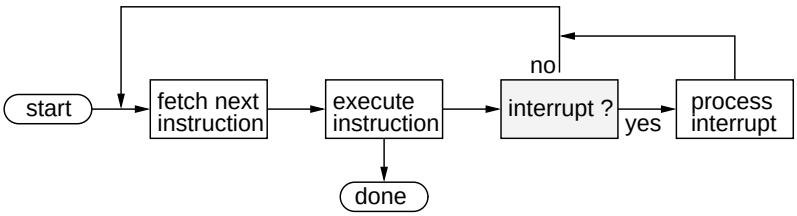
- an instruction address is set at the program counter (PC).
- an instruction is loaded to the instruction register (IR).

Execution cycle:

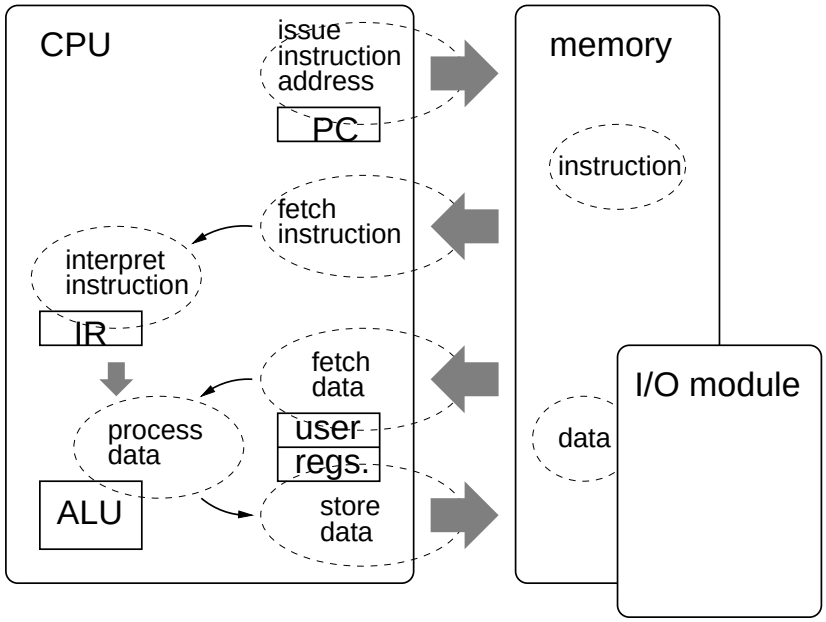
- the instruction is interpreted and processed at, *e.g.*, the ALU.

Interrupt cycle: when an interrupt signal is active,

- program execution is suspended, and the current CPU status is saved.
- the PC is set to a starting address of an interrupt handler routine.



Instruction Fetch / Execution Cycle (2)



Interrupting the Normal Process

Program error:

- arithmetic overflow, division by zero, illegal machine instruction, *etc.*
- reference to outside the user's allowed memory space, *etc.*

CPU timer:

- allows the operating system to do certain tasks in a regular basis.

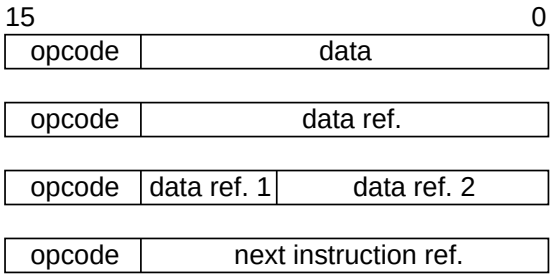
I/O modules:

- signal (1) normal completion of an operation, or (2) a variety of error conditions.

Hardware failure:

- memory parity error, power failure, *etc.*

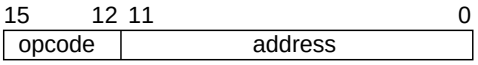
Instruction Format



(hypothetical)

- operation code (or **opcode**) specifies operation to be performed.
- data (**source** or **destination**) reference specifies the location of data (*i.e.*, '**operand**') to fetch or to store. The location can be either CPU registers, main memory (or virtual memory), or I/O modules.
- next instruction reference shows where to fetch the next instruction (implicit when the next instruction immediately follows the current one).

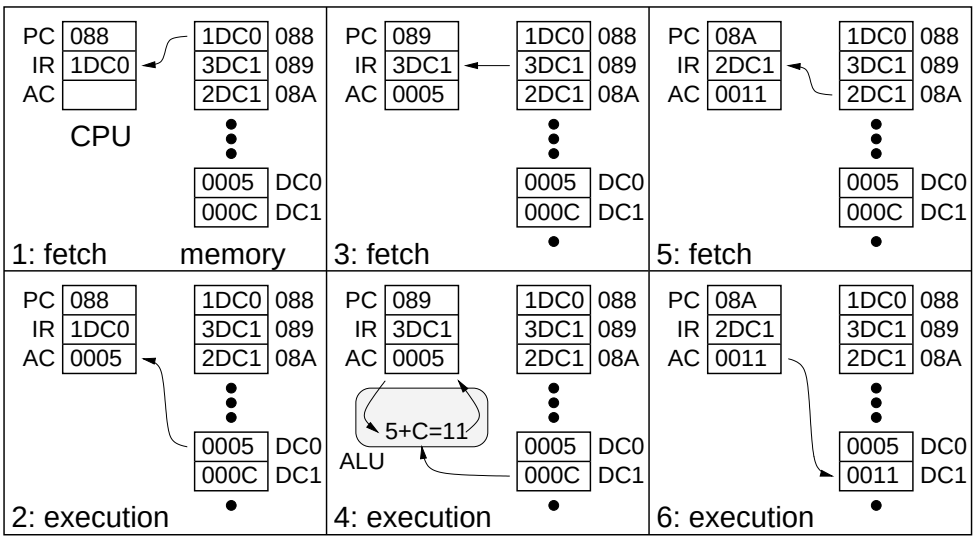
Program Execution: Example



instruction format

0001 : load AC from memory
0010 : store AC to memory
0011 : add to AC from memory

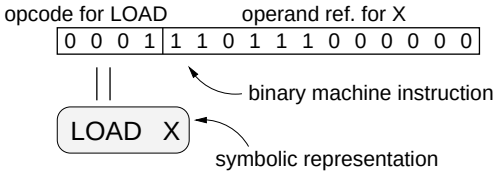
opcodes



Symbolic Representation of Instructions

Program code by 'high-level language':

```
x = 5;
y = 12;
y = x + y;
```



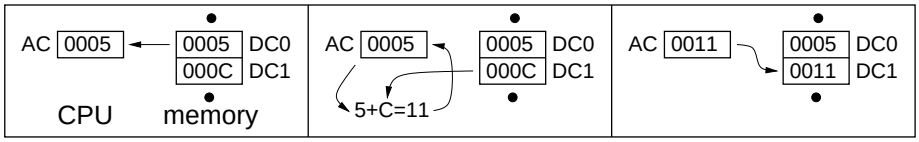
The first two lines ('x = 5' and 'y = 12') result in

'a value 5 is stored at some memory address, say, DC0' and 'a value 12 is stored at DC1'.

'y = x + y' indicates that we

'add contents of memory addresses DC0 and DC1, then store the result at DC1'.

Using the hypothetical machine with one general purpose register AC (accumulator),



This operation may be written using symbolic representations:

(note)
AC is implicit by this symbolic representation.

LOAD	X	(AC ← X)
ADD	Y	(AC ← AC + Y)
STORE	Y	(Y ← AC)
opcode	operand	meaning

Symbolic Representation of Instructions (2)

- symbolic representation symplifies binary machine instruction.
- there exists one-to-one match between symbolic form and binary instruction.

#operand	symbolic representation		interpretation
1	opcode	A	(AC ← AC opcode A)
2	opcode	A,B	(A ← A opcode B)
3	opcode	A,B,C	(A ← B opcode C)

basic rule

(Example)

different types of instruction sets executing the arithmetic calculation:

X = (A - B) / (C + D x E)

LOAD	D	(AC ← D)
MPY	E	(AC ← AC x E)
ADD	C	(AC ← AC + C)
STORE	X	(X ← AC)
LOAD	A	(AC ← A)
SUB	B	(AC ← AC - B)
DIV	X	(AC ← AC / X)
STORE	X	(X ← AC)

one-operand instructions

MOVE	X,A	(X ← A)
SUB	X,B	(X ← X - B)
MOVE	Y,D	(Y ← D)
MPY	Y,E	(Y ← Y x E)
ADD	Y,C	(Y ← Y + C)
DIV	X,Y	(X ← X / Y)

two-operand instructions

SUB	X,A,B	(X ← A - B)
MPY	Y,D,E	(Y ← D x E)
ADD	Y,Y,C	(Y ← Y + C)
DIV	X,X,Y	(X ← X / Y)

three-operand instructions

Operation Types

Data transfer:

- moving data between registers/memories.
- if one or more data are in memory, then the CPU does:
  1. calculate the memory address, based on the addressing mode.
  2. translate from virtual to real memory address, if necessary.
  3. if the addressed item is in cache, then get it from there.
  4. if not, transfer from memory and update cache.

Conversion:

- converting data from one form to the other (*e.g.*, decimal to binary).

Arithmetic operations:

- signed integer and floating point operations, *e.g.*,

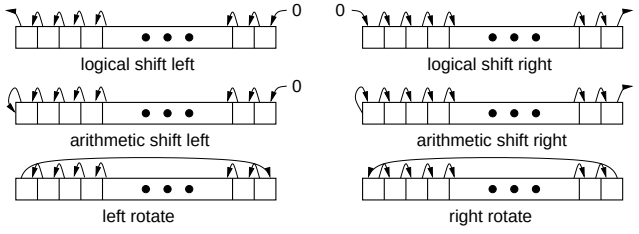
  add, **sub**, **mul**, **div**, **abs**, **neg**, **inc**, **dec**.

Operation Types (2)

Logical operations:

- bit manipulation, (*e.g.*, Boolean operations: AND, OR, NOT, XOR).
- **shift** and **rotate** operations:

arithmetic shifts treat the data as a signed integer. (it does not shift the sign bit.)



I/O operations:

- issuing commands to programmed I/O devices.

System control:

- can be executed only when a processor is in a certain privileged state — usually reserved for the use by the OS.

Operation Types (3)

Transfer of control — **branch instructions**:

- branch is made based on a condition code set by a processor as the result of certain operations. For example, addition 'ADD P' may affect condition codes such as sign, zero, carry, overflow. Based on one of these codes, a branch operation can be done:

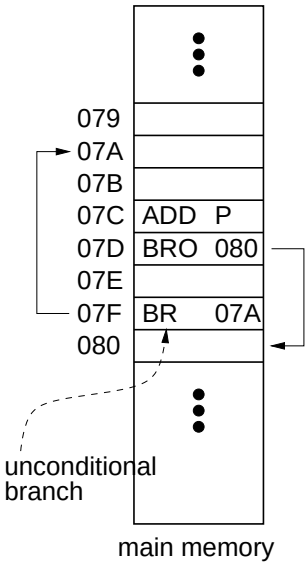
  BRP X (branch to location X if positive.)
  BRN X (branch to location X if negative.)
  BRZ X (branch to location X if zero.)
  BRO X (branch to location X if overflow occurred.)

  If the condition is not satisfied, then the next instruction in the memory is executed.

- alternative is a three-operand instruction that does a comparison and specifies a branch in the same instruction. For example,

  BRE X,Y,Z

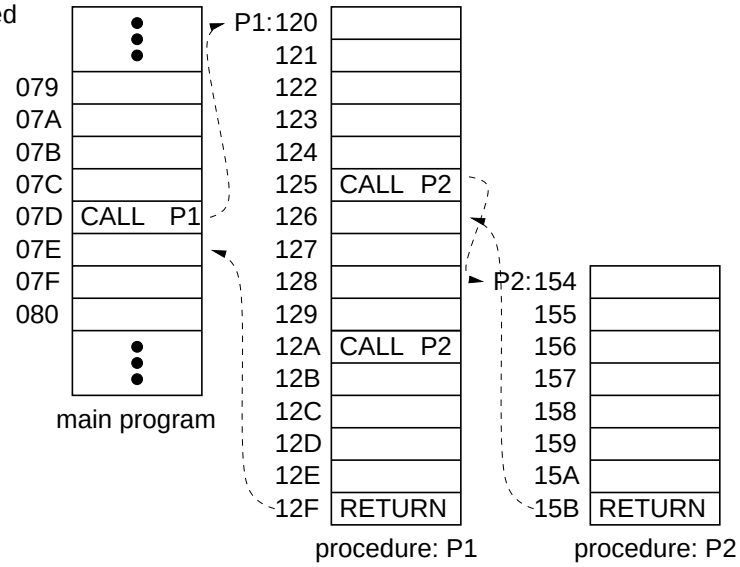
  compares contents in locations X and Y, then, if they are equivalent, make a branch to location Z.



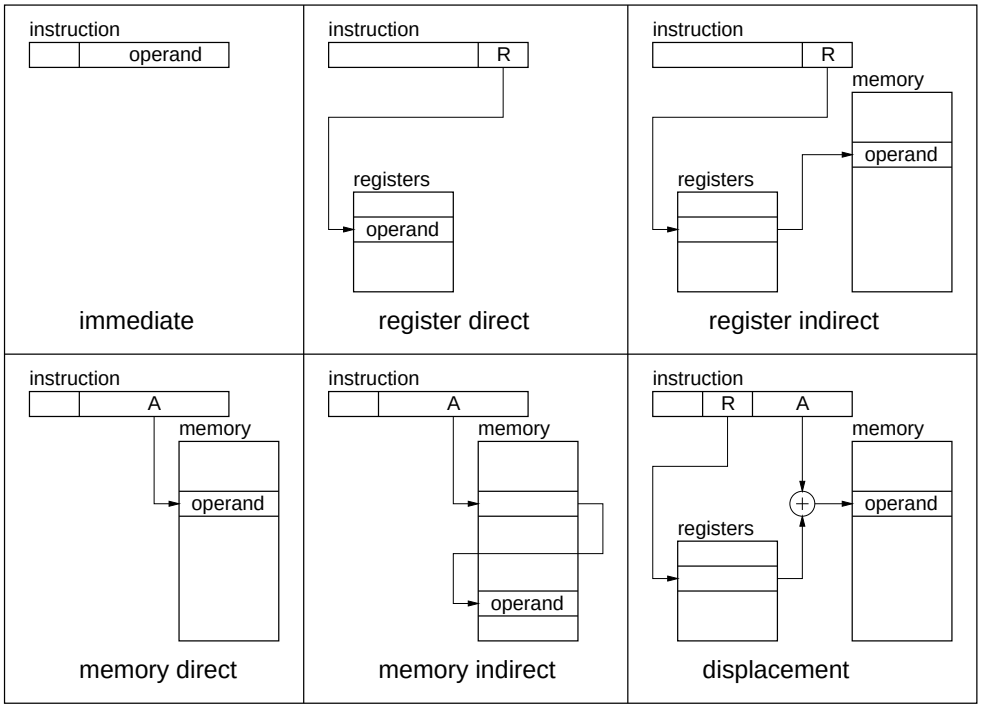
Operation Types (4)

Transfer of control — **procedure calls**:

- a 'procedure' is a self-contained computer program that is incorporated into a larger program.
- economical:
  - allows the same piece code to be used many times.
- modular:
  - allows large programming tasks to be subdivided into smaller unites.



Addressing Mode



Addressing Mode (2)

A	instruction reference field that points to a memory address
R	instruction reference field that points to a register
EA	effective address (actual address) of the referenced operand
(X)	contents of location X

Immediate addressing:

- the operand is present in the instruction, *i.e.*, the instruction has a operand field, rather than an address field.
- no memory reference other than instruction fetch (thus simple), but usually the field size is smaller than the word length.

Memory direct addressing: $EA = A$

- the address field contains the EA of an operand.
- simple (but one memory reference), addressable space is limited.

Addressing Mode (3)

Memory indirect addressing: $EA = (A)$

- the address field of the instruction refers to a memory address that, in turn, contains a full length of an operand.
- large memory space is available, but need two memory references.

Register direct addressing: $EA = R$

- the instruction field refers to a register instead of a memory address.
- no memory reference (*i.e.*, fast), but very limited addressable space.

Register indirect addressing: $EA = (R)$

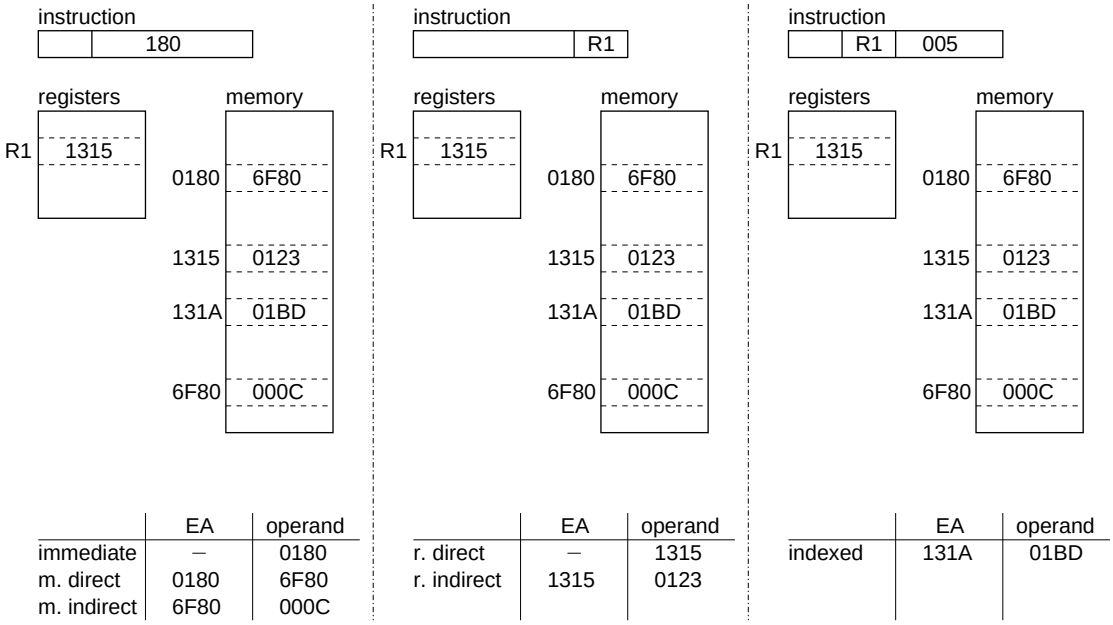
- the address field refers to a register that contains the EA.
- large memory space with one memory reference, but small # registers.

Addressing Mode (4)

Displacement addressing: $EA = A + (R)$

- **relative addressing**: the current instruction address (in the PC) is added to the address field to produce the EA.
- **base register addressing**: the referenced register contains a memory address, and the address field contains a displacement.
- **indexed addressing**: the address field contains a memory address, and the referenced register contains a displacement from that address (different interpretation from the **base register addressing**).
  (*e.g.*) wish to manipulate contents of memory locations $A, A+1, A+2, \ldots \implies$ set A to the address field, and increment the referenced register contents by 1 for each time.
- flexible, but complex.

Addressing Mode: Example



	EA	operand
immediate	–	0180
m. direct	0180	6F80
m. indirect	6F80	000C

	EA	operand
r. direct	–	1315
r. indirect	1315	0123

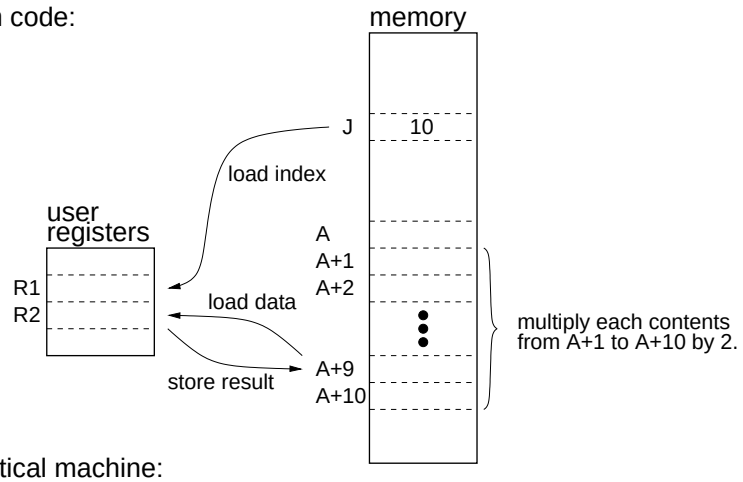
	EA	operand
indexed	131A	01BD

Sample Program

‘high−level language’ program code:

```
j = 10;
do {
   a(j) = a(j) * 2;
   j−−;
} while (j > 0);
```

translation



assembly program by a hypothetical machine:

```
        LOAD   R1,J      load contents of memory location J to R1 −− direct addressing mode
LOOP:   LDIX   R2,R1,A   load contents of memory location A + (R1) to R2 −− indexed addressing mode.
        MPYI   R2,2      multiply (R2) by an immediate value 2.
        STIX   R2,R1,A   store (R2) to memory location A + (R1) −− indexed addressing mode.
        DEC    R1        decrease contents of R1 by 1.
        BRP    LOOP      branch to LOOP if the condition codes, set by the previous instruction, indicate
                                                                    the positove result.
        HALT             terminate the processing.
```