McGill

COMP 208 – Computers in Engineering

COMP 208
Computers in Engineering

Lecture 02

Jun Wang
School of Computer Science
McGill University
Fall 2007

Computer Architecture

- We will briefly look at the structure of a modern computer
- That will help us understand some of the concepts that occur in Fortran and C

  - information representation
  - binary numbers
  - major hardware components
    - memory
    - CPU

Representing information as numbers

- functionally, a computer processes information according to instructions
  - for example, a web browser takes the information stored in an html file and displays a web-page

- Computers store *all* information as numbers:
  - numbers
  - text
  - graphics and images
  - audio
  - video
  - program instructions

Representing information as numbers: example

- American Standard Code for Information Interchange (ASCII)
  - a widely used text encoding scheme based on English alphabet
  - defines coding for 128 characters

+	43		A	65		a	97
.	46		B	66		b	98
?	63		C	67		c	99

Another example

ASCII Code

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01		SOH
^B	2	02		STX
^C	3	03		ETX
^D	4	04		EOT
^E	5	05		ENQ
^F	6	06		ACK
^G	7	07		BEL
^H	8	08		BS
^I	9	09		HT
^J	10	0A		LF
^K	11	0B		VT
^L	12	0C		FF
^M	13	0D		CR
^N	14	0E		SO
^O	15	0F		SI
^P	16	10		DLE
^Q	17	11		DC1
^R	18	12		DC2
^S	19	13		DC3
^T	20	14		DC4
^U	21	15		NAK
^V	22	16		SYN
^W	23	17		ETB
^X	24	18		CAN
^Y	25	19		EM
^Z	26	1A		SUB
^[	27	1B		ESC
^\	28	1C		FS
^]	29	1D		GS
^^	30	1E	▲	RS
^-	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂*

* ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key.

Computers use binary numbers to represent all information.

Decimal numbers (base 10)

- Examples:
  - "45 students registered for this course"
  - "Fortran was first introduced in 1957"

- Observations:
  - decimal number has 10 basic digits: 0 ~ 9
  - positions of digits matter: 45 != 54
  - 45 is shorthand of $4 \times 10^1 + 5 \times 10^0$
  - $1957 = 1 \times 10^3 + 9 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$

Binary numbers (base 2)

- 2 basic digits, 0 and 1
  - $45_{10} = 101101_2$
  - $101101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  - recall: $1957 = 1 \times 10^3 + 9 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$

	decimal	binary	base-b
digits	10	2	b
weight	10^p	2^p	b^p

A single binary digit (0 or 1) is called a *bit*.

Hexadecimal numbers (base 16)

- Has 16 basic digits, 0 ~ 9, A, B, C, D, E, F

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

- one-to-one mapping between 1 hex digit and 4 binary digits.

Von Neumann Machines

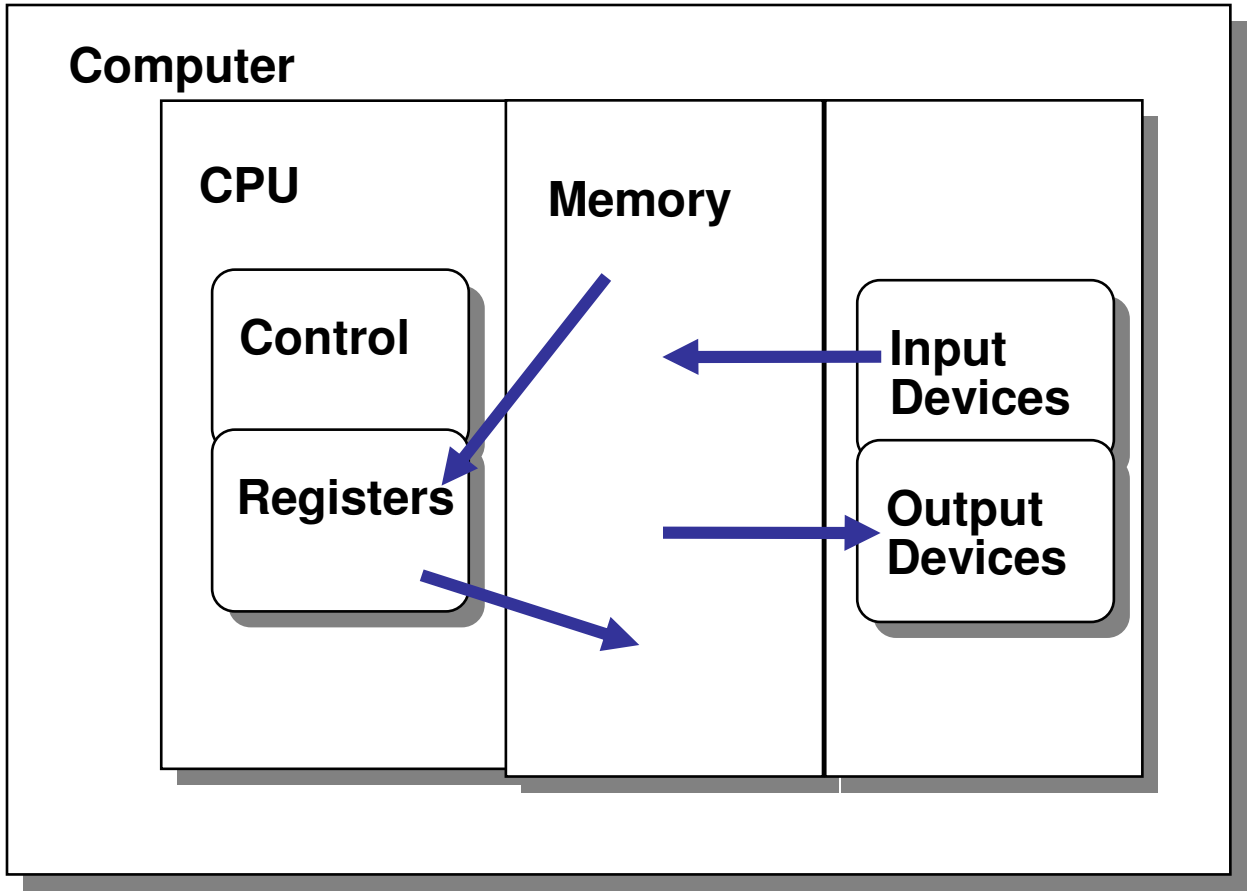
- Modern computers are called Von Neumann Machines

- John Von Neumann is credited with the idea that programs can be encoded and stored in the memory just like data

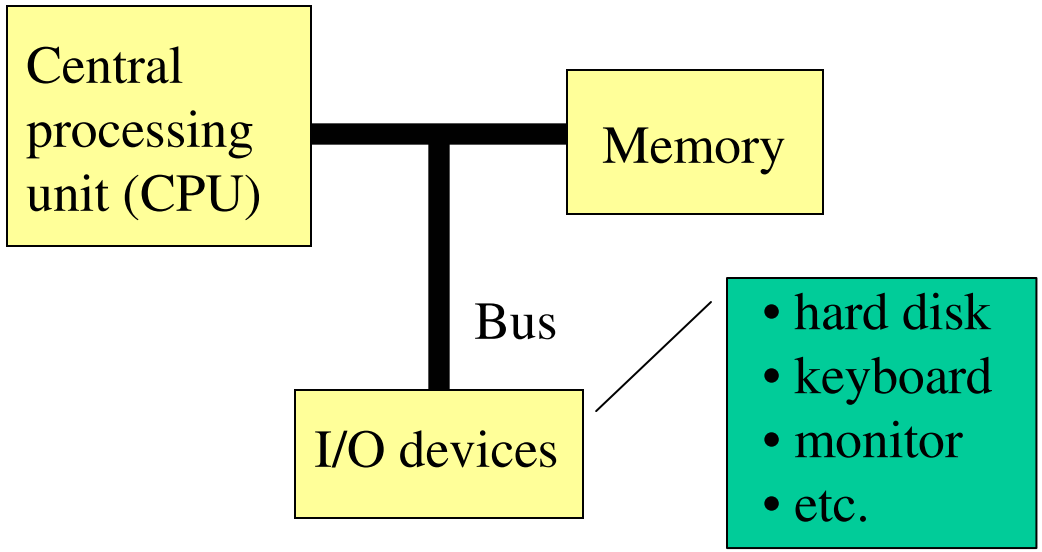
- There is one CPU (Central Processing Unit)

- A control unit transfers instructions from the memory into registers so that a processing unit can execute them

The 5 Classic Components

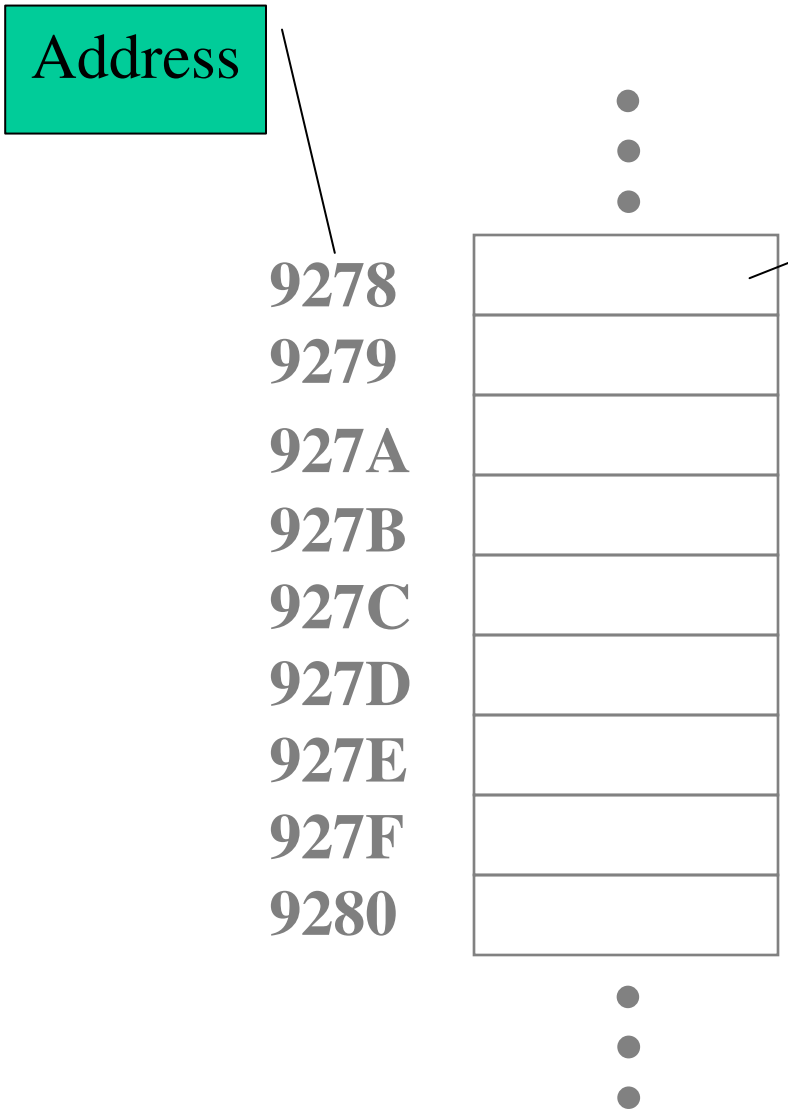


The Von Neumann Model



- programs loaded from disk to memory, and executed in CPU

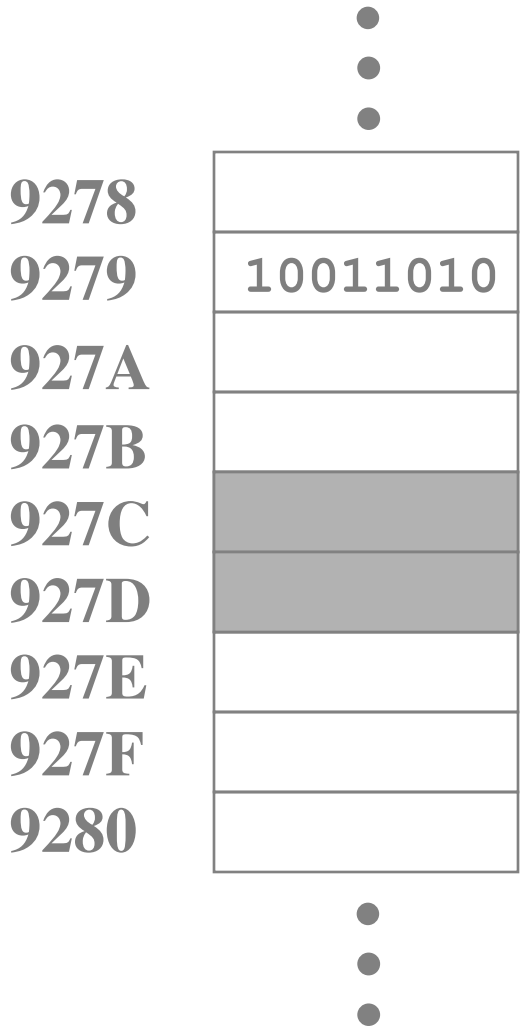
Memory



- Main memory is divided into many memory locations (or *cells*)
- Each memory cell has a numeric *address*, which uniquely identifies it
- Memory contents are lost when power off

Memory goes from address 0 to N-1 where N is the amount you purchased.

Storing Information



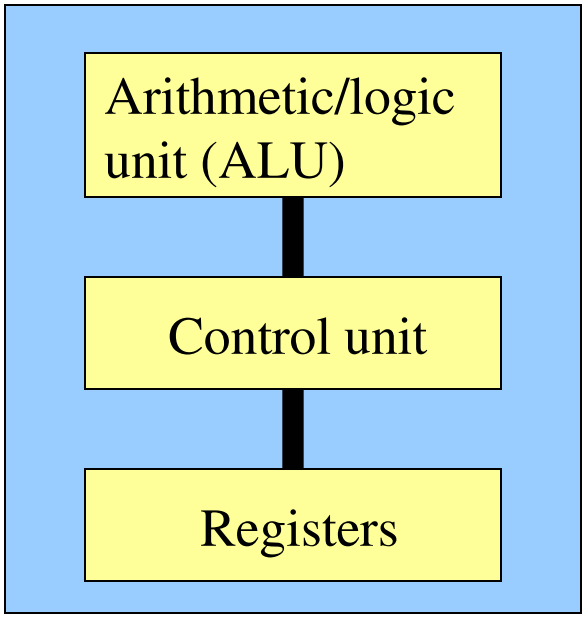
Each memory cell stores a set number of bits (usually 8 bits, or one ***byte***)

Large values are stored in consecutive memory locations

For example, a integer may take 4 bytes

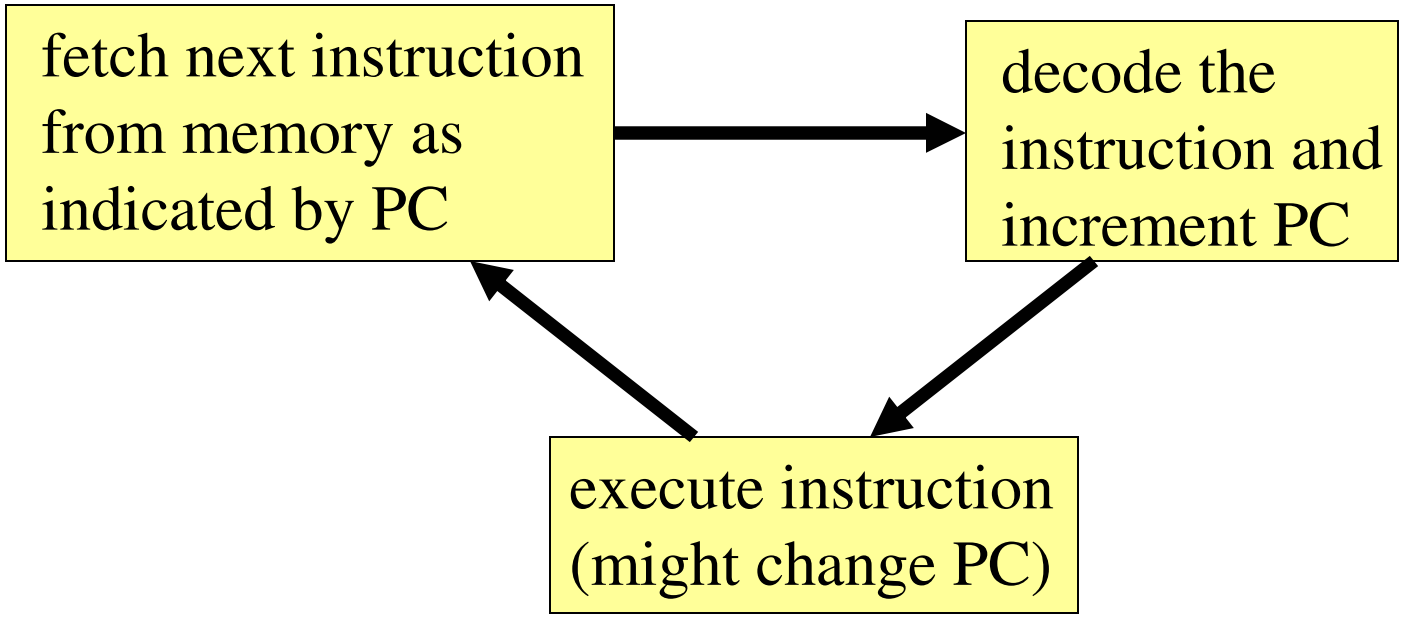
1 byte = 8 bits

CPU

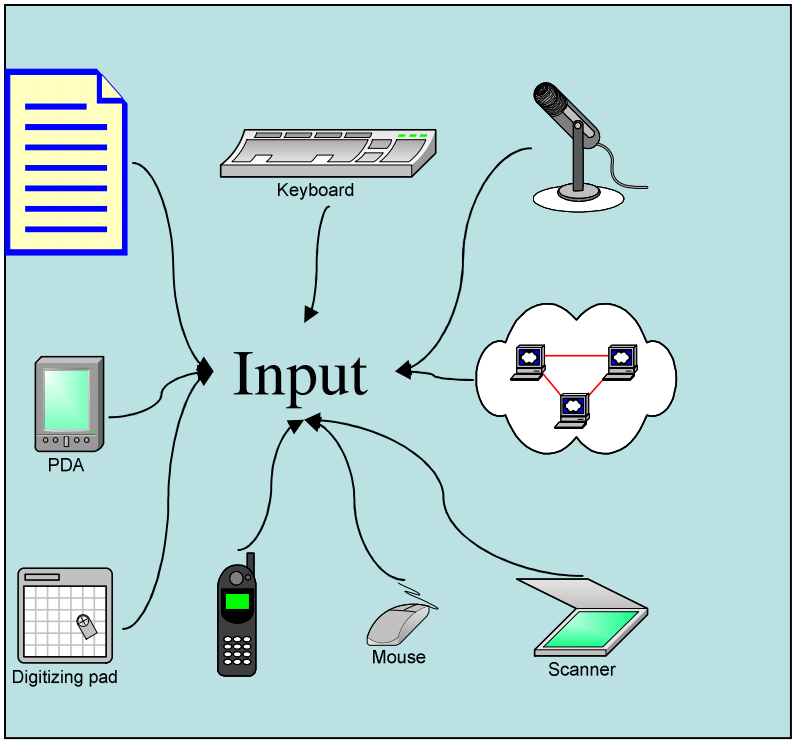


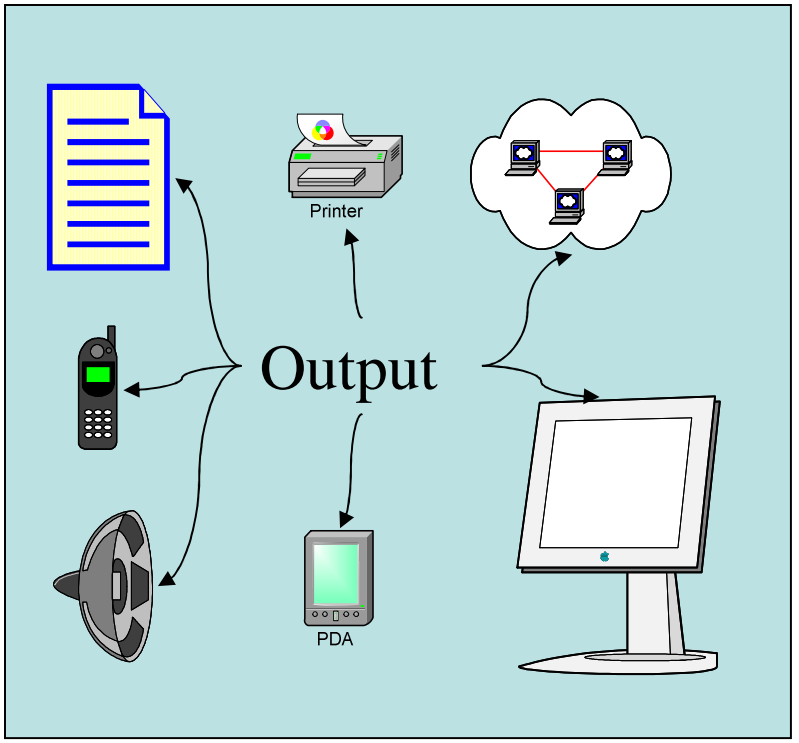
- Registers are storage units
  - general-purpose and special registers
  - names instead of addresses
  - Program Counter (PC), a.k.a. Instruction Pointer (IP), points to the memory address of the next instruction.

Fetch-decode-execute



Input / Ouput


Keyboard
Input
PDA
Digitizing pad
Mouse
Scanner


Printer
Output
PDA

The Von Neumann Model

- Input devices (keyboard, pda, cell phone, . . .) allow us to place data (and programs) into memory
- Output devices allow us to display values stored in memory (on screen, pda, cell phone, . . .)

How to program this machine?

- Understand where high-level languages come from

CPU instruction set

- CPU only understands a small set of pre-defined instructions – its instruction set
  - arithmetic/logic operations; data movement
- Each instruction tells the CPU to carry out a basic operation
- For example, Intel 80x86 CPU:
  - move the byte at address 1000 to the register AL:
    10100000 00000000 00010000 (opcode data)
    (a0 00 10 in hex)
  - add 1 to the register CX:
    10000000 10100001 00000001 00000000
    (80 a1 01 00 in hex)

3 generations of programming languages

- generation 1: machine language

```
a0 00 10       // move byte at 1000 to AL
02 05 01 10    // add byte at 1001 to AL
a2 02 10       // move AL to 1002
```

- generation 2: assembly language

```
mov al, x    // move byte x to AL
add al, y    // add byte y to AL
mov z, al    // move AL to z
```

- give each instruction a mnemonic name
- use symbols to represent memory locations

symbols representing memory locations are called *variables*

- generation 3: high-level language

```
z = x + y;
```

each line of high-level program usually translates into many lines of machine code

variables are symbols representing memory locations

- **task:**
  1. put the number 20 in a memory cell
  2. put the number 4 in another cell
  3. add the 2 numbers and put result in a third cell
  4. print the number in the 3rd cell
- **machine code**
  1. put the number 20 in memory cell **1000**
  2. put the number 4 in cell **1001**
  3. add the 2 numbers and put result in cell **1002**
  4. print the number in the cell **1002**
- **generation 2 and beyond**
  1. put the number 20 in memory cell **x**
  2. put the number 4 in cell **y**
  3. add the 2 numbers and put result in cell **z**
  4. print the number in cell **z**

Low Level Programming

- Programmers in the late 1940's had to use binary numbers to encode the instructions and the data

- This was very time consuming and error prone so written mnemonic codes were created. Programs were written using these codes and then translated into binary by hand

- Soon programs were written to convert these symbols to binary. These programs are called **assemblers** and the instruction names are called **assembly language**

Assembly Program Example

We may want to evaluate the expression

```
f = (g + h) – (i + j)
```

Assembly program (where all the names refer to registers)

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

Load and Store instructions are part of the assembly language and allow transferring data values between memory and registers

Assembly Language

- Low level language
- Simple instructions of the form

```
op result, arg1, arg2
```

- Machine dependent – each processor has its own assembler

Native Code

An assembler translates this code into a binary sequence of instructions

```
add t0, g, h   -> 0110100000010010
add t1, i, j   -> 0100100100110100
sub f, t0, t1  -> 0101010110001000
```

High Level Languages

Programming in assembly language is still difficult and tedious

Programs are very specific to specific machines

High level languages provide a more natural mathematically based formalism for expressing algorithms

High Level Languages

High level languages

- Hide details of memory allocation and hardware details
- Allow expressing complex operations together, not just one step at a time
- Provide a more natural way of programming
- Allow programs to be ported from one machine to another
- Liberate programmers from low-level hardware issues, so they can focus on problem solving, and program structures

High Level Languages

These languages make it easier to write programs but they are still very formal, precisely structure languages that follow very specific syntax rules

In addition to learning how to formulate algorithms for the computer, we will have to learn the rules for these languages

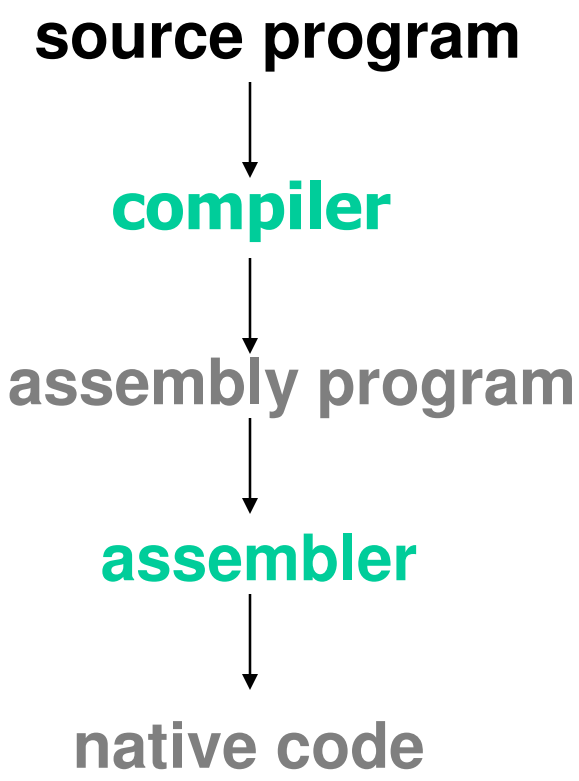
From high to low

- High-level programs must be translated into machine code for CPU to execute
- Compiler:

  translates high-level program into assembly program
- Assembler:

  translates assembly program into machine code
- A compiler usually combines the two

The Translation Process



Source Program

- A program written in a high level language (FORTRAN, C, Java, C++, Ada)
- Created with a text editor in human readable form
- File name extension often says what language is used (a1.f90, a4.c, test.java)

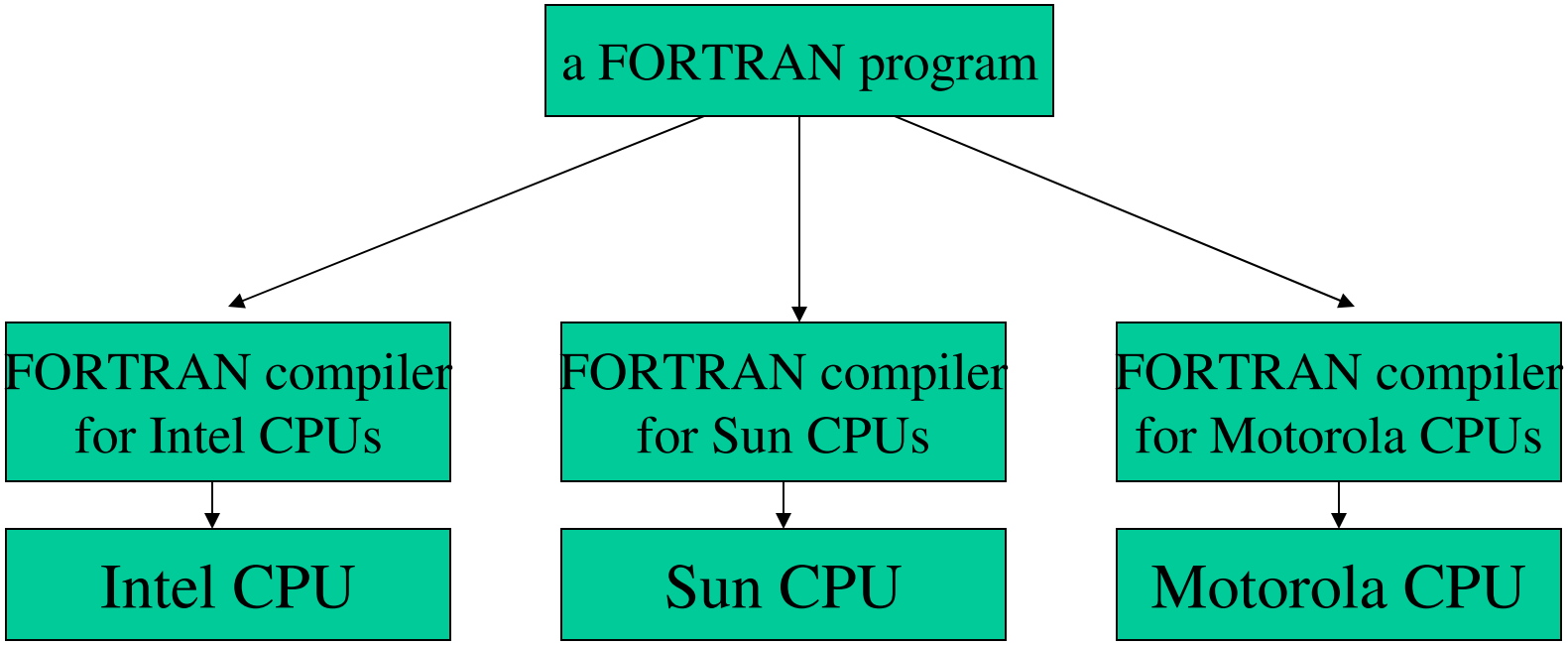
Compiler

- A program that analyses the source program and translates it into a form the computer can understand
- Result is not readable by humans
- Each high level language requires its own compiler

Linker/Loader

- The Linker combines the assembler code with other programs that were compiled another time or are standard programs available in libraries (sin, sqrt, etc)
- The Loader puts the complete program in memory and begins execution with the first instruction

Portability of high-level programs



A First FORTRAN Program

```
PROGRAM hello
  IMPLICIT NONE
  !This is my first program

  WRITE (*,*) "Hello World!"

END PROGRAM hello
```