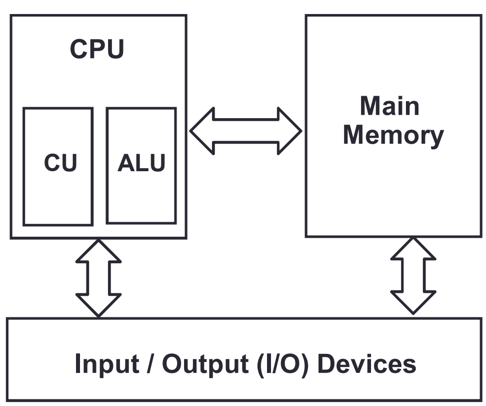
Computer Structure and Language

Hamid Sarbazi-Azad
Department of Computer Engineering
Sharif University of Technology (SUT)
Tehran, Iran

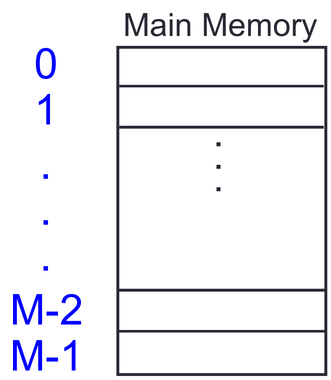
Computer System

In famous stored-program computer structure of John von Neumann (known as von Neumann model), CPU is connected to the memory system and I/O devices.



Main Memory

From the machine language programmer's view point, main memory is an array of locations addressed from 0 to M-1 (M being the memory size).



Main Memory

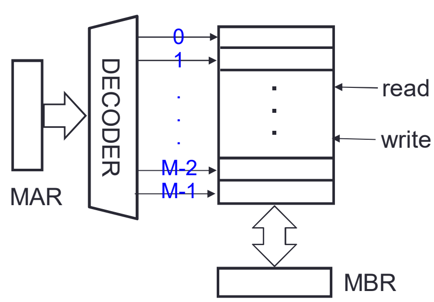
All accesses to memory are realized via two registers:

- MAR (Memory Address Register),
- MBR (Memory Buffer Register), and
- two control signals (Read and Write).

The address of the location to be read/written is placed in MAR.

The data communicated with memory is resided in MBR or MDR (Memory Buffer/Data Register).

What is MAR's size in bits?
What is MBR's size in bits?



Main Memory

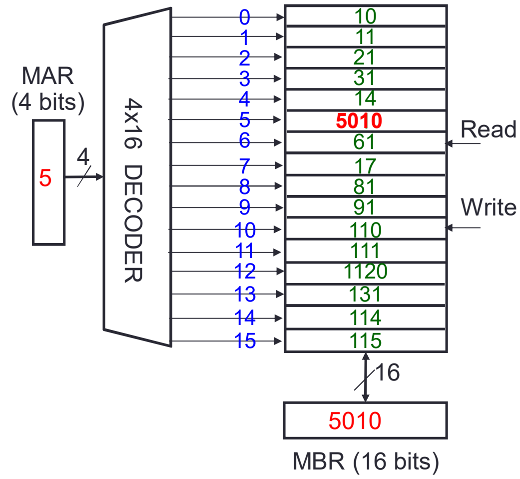
Example: In a 16x16 bit memory, show the steps of writing 5010 in address 5.

Write Operation:

Step 1. Store 5 in MAR.

Step 2. Store 5010 in MBR.

Step 2. Activate Write signal.
(data written in memory)



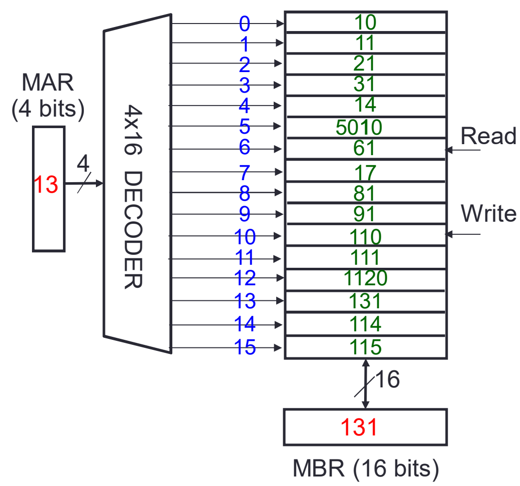
Main Memory

Example: In a 16x16 bit memory, show the steps of reading from address 13.

Read Operation:

Step 1. Store 13 in MAR.

Step 2. Activate Read signal.
(data is read into MBR)



Main Memory

Notation. We use M_{addr} to show a memory location with address *addr*.

For example, M_{1000} indicates location# 1000 in the main memory.

The content of a memory location (or register, e.g. MAR or MBR) is shown by a pair of parentheses around the location address.

For example, (M_{1000}) indicates the data stored in location# 1000 in the main memory.

Main Memory

Example: In the following 16-location main memory, we have:

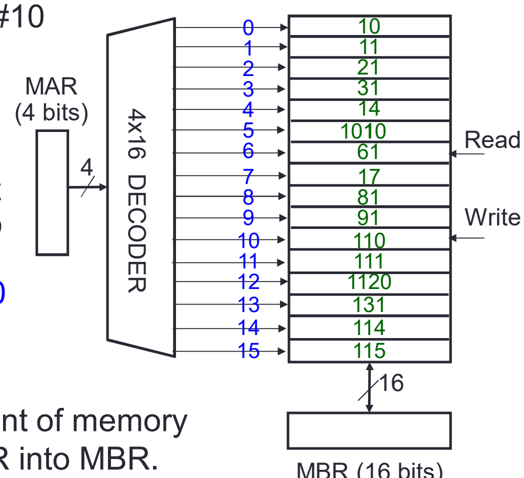
- $(M_9) = 91$
- $M_{(M\_0)}$ indicates location#10
- $(M_{(M\_1)}) = 111$

Command

$M_{10} \leftarrow (M_{12})$;

means copying the content of memory location#12 into memory location#10.

→ so, we have $(M_{10})= 1120$



By command

$MBR \leftarrow (M_{(MAR)})$;

we mean moving the content of memory location addressed by MAR into MBR.

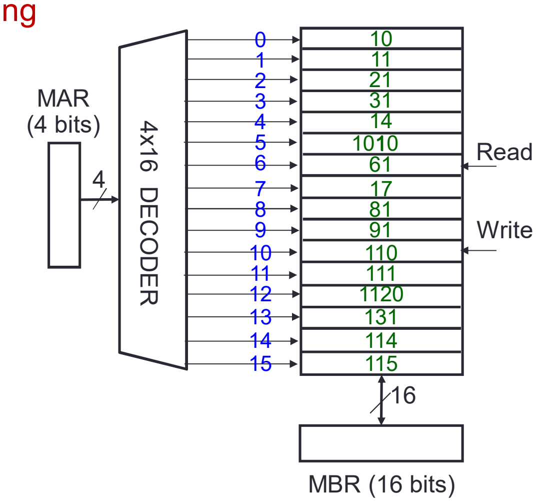
→ i.e. memory read operation!

Main Memory

What happens after realizing the below command?

$M_{(M\_0)} \leftarrow (M_0) + (M_{(M\_4)})$;

→ We have $(M_{10}) = 124$



Main Memory

Memory Characteristics:

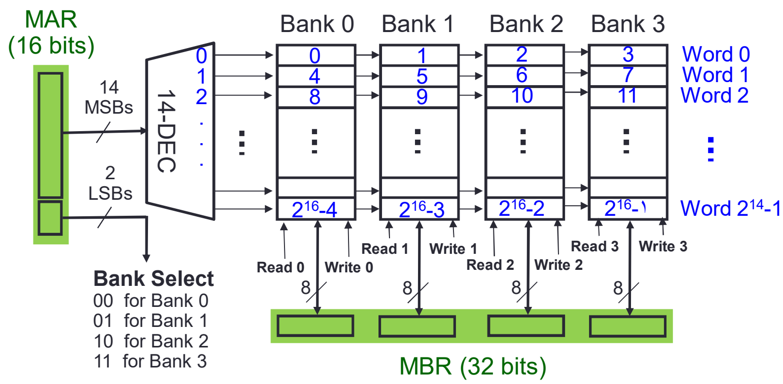
- Addressable unit: The smallest group of bits that has a unique address in memory. It is 8 bits in most machines.
- Access width/length: The number of bits that can be read or written in one memory access. It is different from machine to machine.
- Word: The number of bits of the operands that CPU process when a machine instruction is executed. It is n-bit in an n-bit machine. In first generation CPUs, we had n=8 or 16. For most current machines we have n=32. But machine with n=64 and larger word sizes are also available.

Typically, Addressable unit size < Word size < Access width.

If not explicitly mentioned, then assume all are of equal size.

Sample Main Memory Architecture

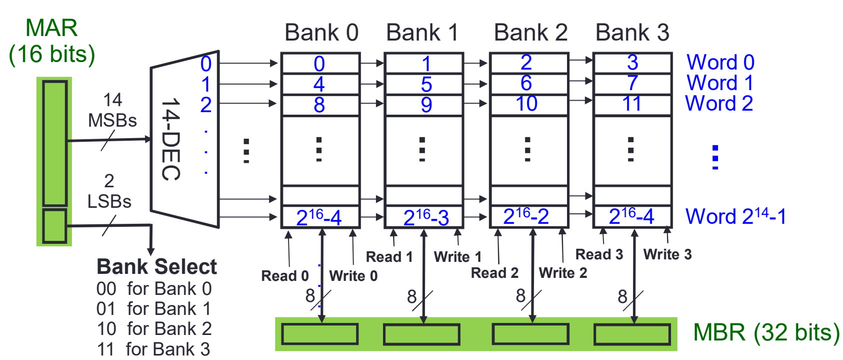
Addressable unit = 8 bits
Word size = 32 bits
Access length = 32 bits
Memory capacity = 2^{16} addressable units = 2^{14} words



Word Alignment

Aligned Word Memory Architecture: Word address is a multiple of its length. In below memory @ 0, 4, 8, 12, …
→ Word is read/written in one memory access ☺

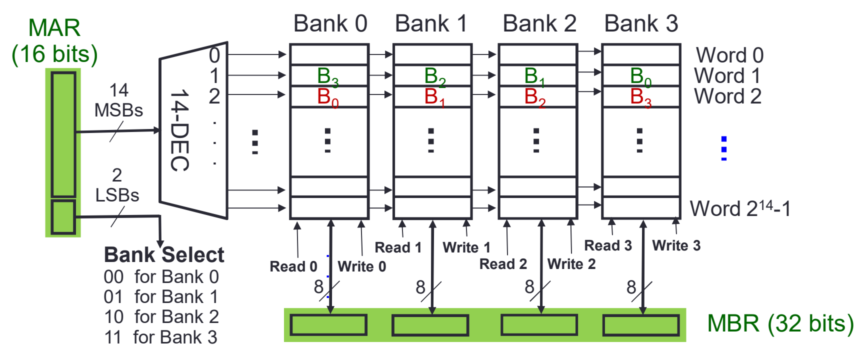
Unaligned Word Memory Architecture: Word address can be anything. In below memory @ 0, 1, 2, 3, …
→ Word may be read/written in two memory accesses ☹



Endianness

Big Endian: When word is stored from most significant addressable unit toward least significant addressable units in consecutive memory addresses. Word $B_3B_2B_1B_0$ @ 0004h.

Little Endian: When word is stored from least significant addressable unit toward most significant addressable units in consecutive memory addresses. Word $B_3B_2B_1B_0$ @ 0008h.



Instruction Format

An Imperative Sentence in human languages is formatted as:

Subject + Verb + Object(s).

Example: Hasan call Mike.

Zahra eat your food.

In machine language, we follow a similar format. Note that in a computer the only Subject is CPU. So, we need not mention it.

So, a machine instruction looks like:

Verb + Object(s) or Operation + Operand(s)

Instruction Format

So, a machine instruction looks like:

Operation + Operand(s).

Operation can be coded by a bit pattern named Operation Code (or OpCode) and Objects are location addresses indicating Operand(s) in binary patterns.

So, the instruction format in an n-address machine looks like:

Opcode	Operand 1	......	Operand n
? bits	? bits	......	? bits

Instruction Format

Example: In a 2-address machine:

- memory size is 128 units of 16 bits.
- Operations include Add (opcode=00), Subtract (opcode =01), Move (opcode=10), and No-operation (opcode=11).

Opcode	Operand 1	Operand 2
2 bits	7 bits	7 bits

For example, machine instruction 01 0001111 0001010 means the operation is Subtract and the two operands are M_{15} and M_{10}.

So, the notational instruction can be shown as:

$$M_{15} \leftarrow (M_{15}) - (M_{10});$$

Opcode

What information can Opcode give?

1. Operation type: add, sub, ….
2. Number of operands (if multiple has multiple instruction formats)
3. Instruction length (if machine has multiple instruction formats)
4. Type of operands: integer, real, signed, …
5. Length of operands: 8 bits, 16 bits, n bytes
6. Addressing mode of operands: direct, indirect,…

The internal paths in a machine

Path #1: Fetching instructions

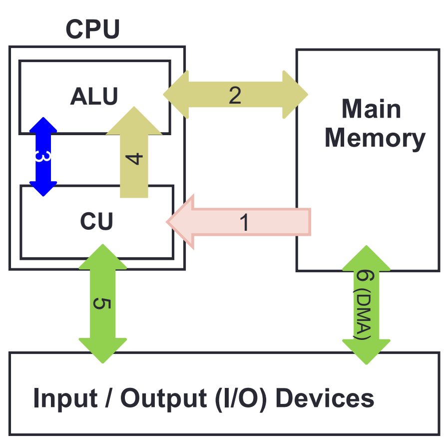
Path #2: Data access

Path #3: Command/status communication

Path #4: Special data access (immediate data)

Path #5: I/O data (via CPU)

Path #6: I/O data (by DMA)



The internal paths in a machine

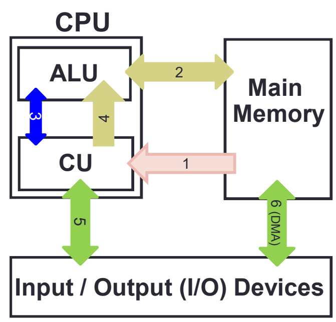
Example: The 2-address machine with 4 instructions

Add (opcode=00),
Subtract (opcode =01),
Move (opcode=10), and
No-operation (opcode=11), and 128-location memory.

Opcode	Operand 1	Operand 2
2 bits	7 bits	7 bits

To fetch and execute machine instruction 01 0001111 0001010 the internal paths are used as follows:

1. Path#1 to fetch instruction 01 0001111 0001010 to CU.
2. Path#2 to get data (M_{15}) to ALU.
3. Path#2 to get data (M_{10}) to ALU.
4. Path#3 to send subtract command (opcode =01) to ALU.
5. Path#2 to write result to M_{15}.



Assembly Language

Example Machine 1: A 2-address machine with 4 instructions

Add (opcode=00),
Subtract (opcode =01),
Move (opcode=10), and
No-operation (opcode=11), and a128-location memory.

Opcode	Operand 1	Operand 2
2 bits	7 bits	7 bits

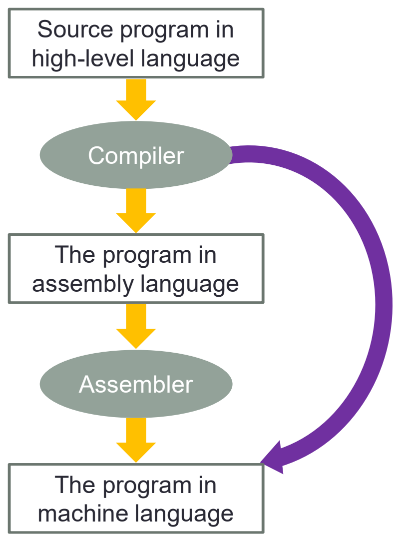
Since writing, reading and understanding programs of machine instructions in binary (machine) format is obscure and complex, we use symbolic representation of instructions, named assembly instruction.

For example, the machine instruction 01 0001111 0001010 may be written, in a symbolic form, as:

sub num1,num2

where num1 and num2 are symbolic equivalents of operands M_{15} and M_{10} and sub is symbolic equivalent (Mnemonic) of opcode=01.

Program execution flow:



Advantages of Assembly Language:

Understanding of assembly language provides knowledge of:

- Interface of programs with OS, processor and BIOS (basic input/output system);
- Representation of data in memory and other external devices;
- How processor X accesses and executes instructions;
- How instructions access and process data in processor X;
- How a program running on X accesses external devices.

Advantages of using assembly language are:

- It requires less memory and execution time;
- It allows hardware-specific complex tasks done easier;
- It is suitable for time-critical tasks;
- It is most suitable for writing interrupt service routines and other memory resident programs;
- It is used to crack locked software;
- It is used to write viruses/worms/….

Example Machine 2:

A 2-address machine has a 2^{16} addressable unit (each of 8 bits) 64 KB main memory. The word size is 16 bits (2 addressable units). Instruction format and list of instructions are shown below:

Opcode	addr1	addr2
8 bits	16 bits	16 bits

Opcode	Mnemonic		Operation
00000000	mov	addr1,addr2	$M_{addr1} \leftarrow (M_{addr2})$;
00000001	add	addr1,addr2	$M_{addr1} \leftarrow (M_{addr1}) + (M_{addr2})$;
00000010	sub	addr1,addr2	$M_{addr1} \leftarrow (M_{addr1}) - (M_{addr2})$;
00000011	and	addr1,addr2	$M_{addr1} \leftarrow (M_{addr1}) \wedge (M_{addr2})$;
00000100	or	addr1,addr2	$M_{addr1} \leftarrow (M_{addr1}) \vee (M_{addr2})$;
00000101	xor	addr1,addr2	$M_{addr1} \leftarrow (M_{addr1}) \oplus (M_{addr2})$;

L_{MAR} = ?
16 bits

L_{MBR} = ?
16 bits

What are the lengths of registers MAR, MBR?

Example Machine 2:

Opcode	addr1	addr2
8 bits	16 bits	16 bits

- **Write an assembly program to sum up 7 elements of Fibonacci sequence (i.e. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …).**
- **Now, translate the assembly program into machine code.**

Address	Machine Code	Assembly Instruction			
			org	100h	; start program from address 100h
0100	0102020204		add	t1,t2	; calculate element#3 in t1
0105	0102000202		add	sum,t1	; add element#3 to sum
010A	0102040202		add	t2,t1	; calculate element#4 in t2
010F	0102000204		add	sum,t2	; add element#4 to sum
0114	0102020204		add	t1,t2	; calculate element#5 in t1
0119	0102000202		add	sum,t1	; add element#5 to sum
011E	0102040202		add	t2,t1	; calculate element#6 in t2
0123	0102000204		add	sum,t2	; add element#6 to sum
0128	0102020204		add	t1,t2	; calculate element#7 in t1
012D	0102000202		add	sum,t1	; add element#7 to sum
			org	200h	; start variable from address 200h
0200	0001	sum:	dw	1	; define a word as variable sum
0202	0000	t1:	dw	0	; define a word as variable t1
0204	0001	t2:	dw	1	; define a word as variable t2
			end		; end of program

END OF SLIDES