Lecture 1: Tiny Machine

The processor as an instruction interpreter

Outline

Von-Neumann Machine

Instruction Cycle

Fetch and Execute Cycle

Two Tiny Machines

- **Accumulator Machine**
- **Load/Store Machine**

Assembly Language

Instruction Cycle

In the **von-Neumann machine**, the **Instruction Cycle** (also **Machine Cycle**) is a loop that repeats the following two steps over and over again:

- **Fetch Cycle**
- **Execute Cycle**

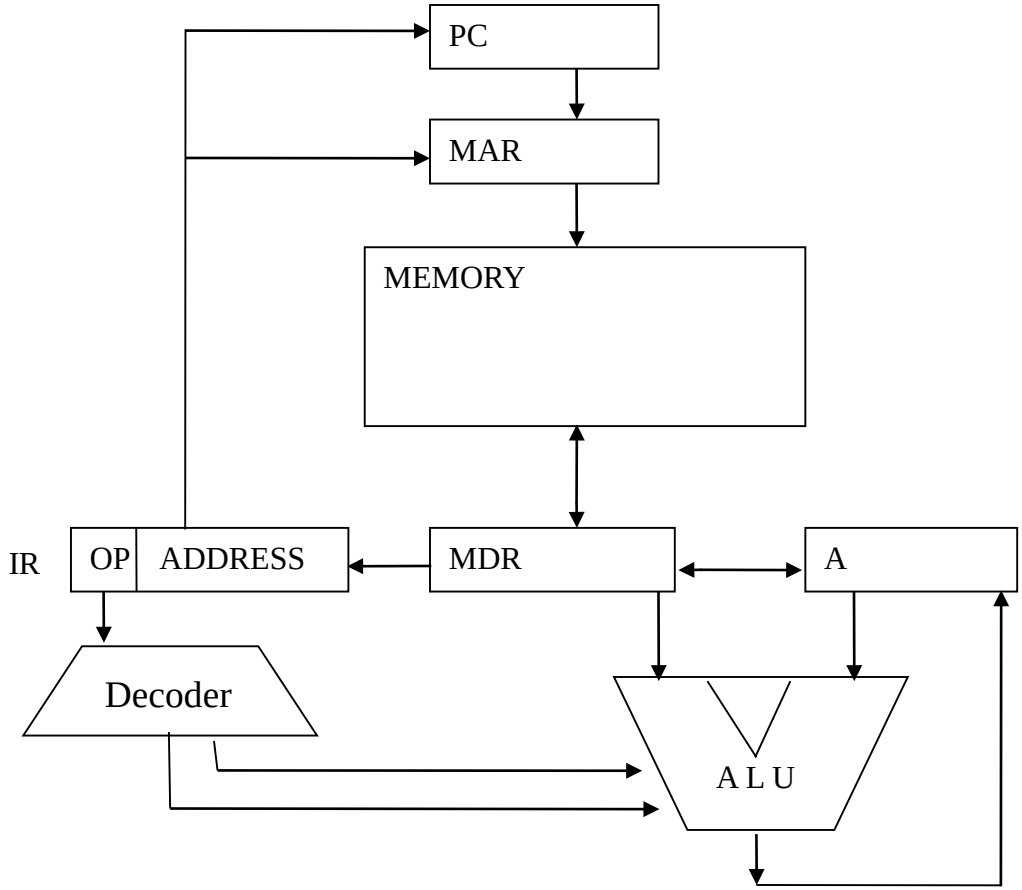
Fetch/Execution Cycle

- **Fetch Cycle:** Instruction is retrieved from memory.
- **Execute Cycle:** Instruction is executed.

Hardware Description Language

We will use a simple **Hardware Description Language** to understand how instructions are fetched and executed in the VN.

Accumulator Machine



Definitions

- **Program Counter (PC)** is a register that holds the address of the next instruction to be executed.
- **Memory Address Register (MAR)** is a register used to store the address to a specific memory location in Main Storage so that data can be written to or read from that location.
- **Main Storage (MEM)** is used to store programs and data. Random Access Memory (RAM) is a implementation of MEM.

Definitions

- **Memory Data Register (MDR)** is a register used to store data that is being sent to or received from the MEM. The data that it stores can either be in the form of instructions or simple data such as an integer.
- **Instruction Register (IR)** is a register that stores the instruction to be executed by the processor.

Definitions

- **Arithmetic Logic Unit (ALU)** is used to execute mathematical instructions such as ADD or SUB.
- **Decoder** is a circuit that decides which instruction the processor will execute. For example, it takes the instruction op-code from the IR as input and outputs a signal to the ALU to control the execution of the ADD instruction.
- **Accumulator (A)** is used to store data to be used as input to the ALU.

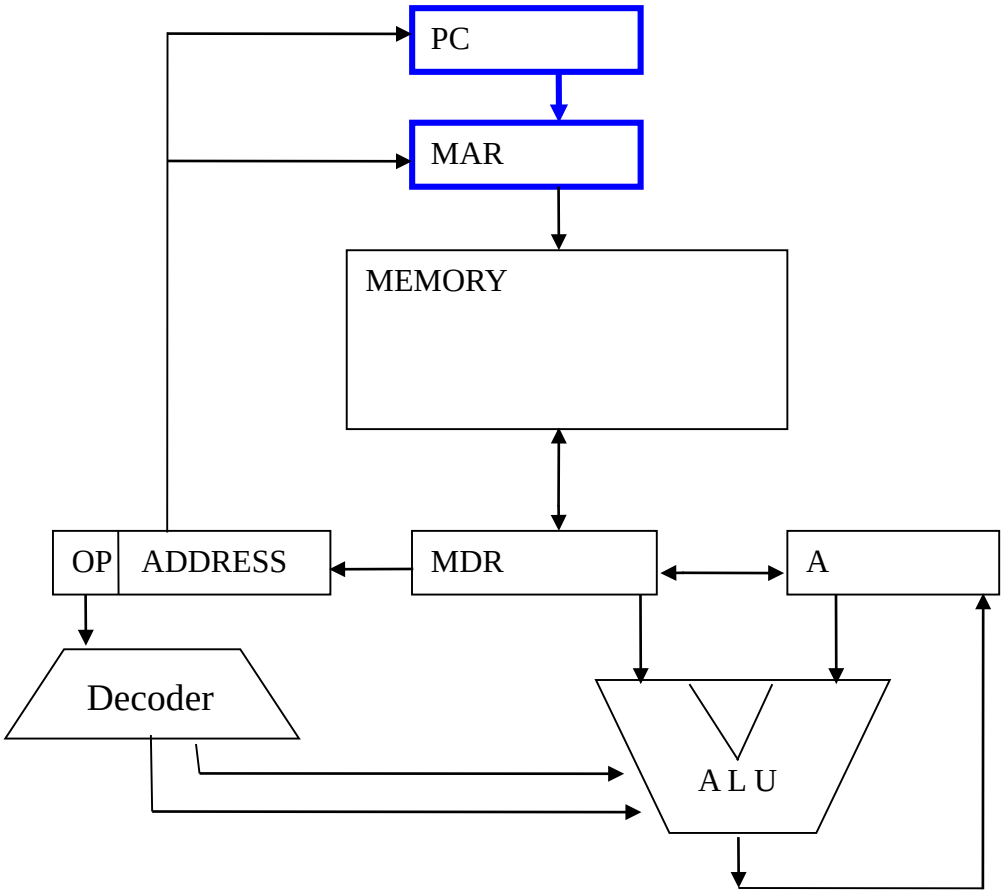
Fetch Cycle

The **data flow** for the Fetch Cycle consists of 4 steps.

Data Movement 1

- Given registers PC and MAR, the transfer of the contents of PC into MAR is indicated as:

  MAR←PC

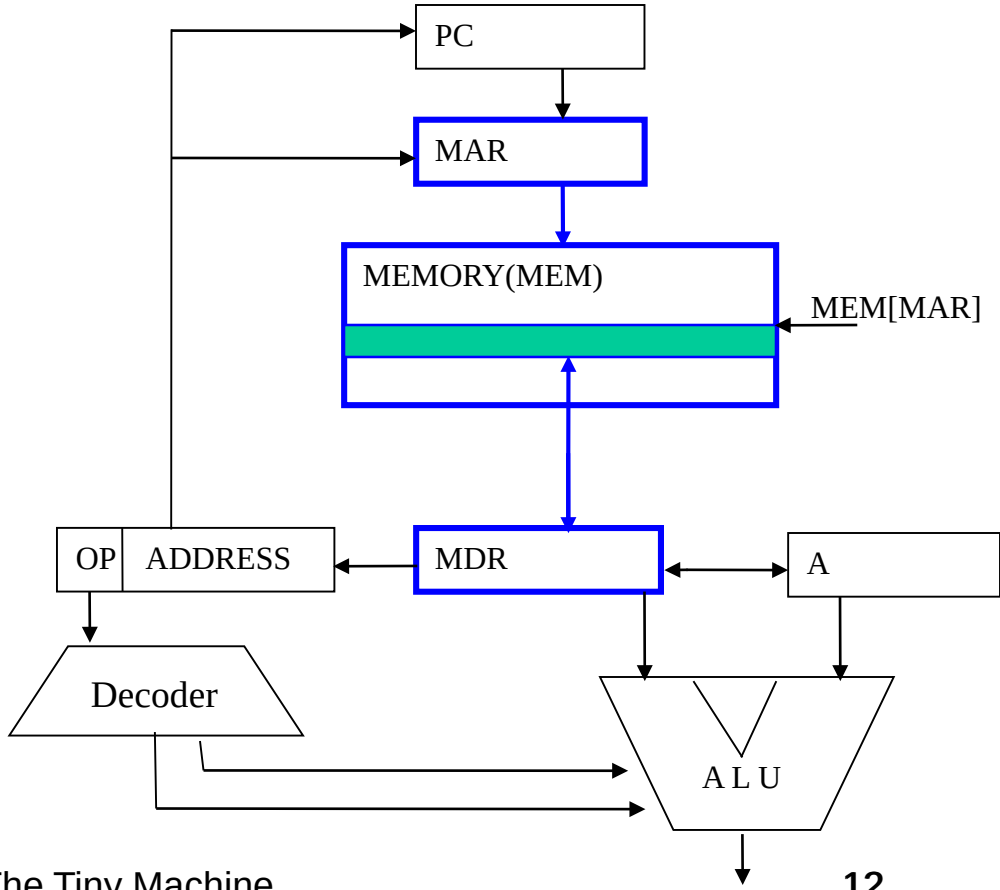


Data Movement 2

- To transfer information from a memory location to the register MDR, we use:

  MDR ← MEM[MAR]

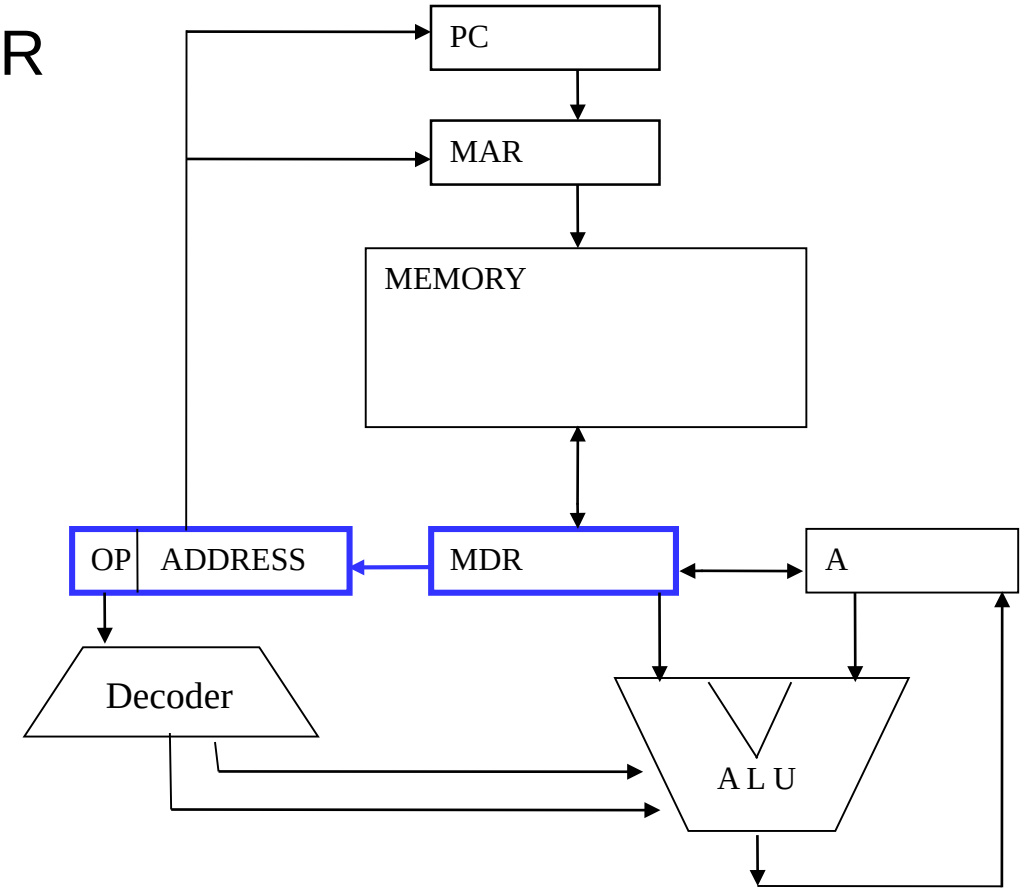
- The address of the memory location has been stored previously into the MAR register



Data Movement 3

- We denote the transfer of MDR to IR by:

  IR⟵MDR



Instruction Register

- The instruction register IR has two fields:

  IR.OP operation

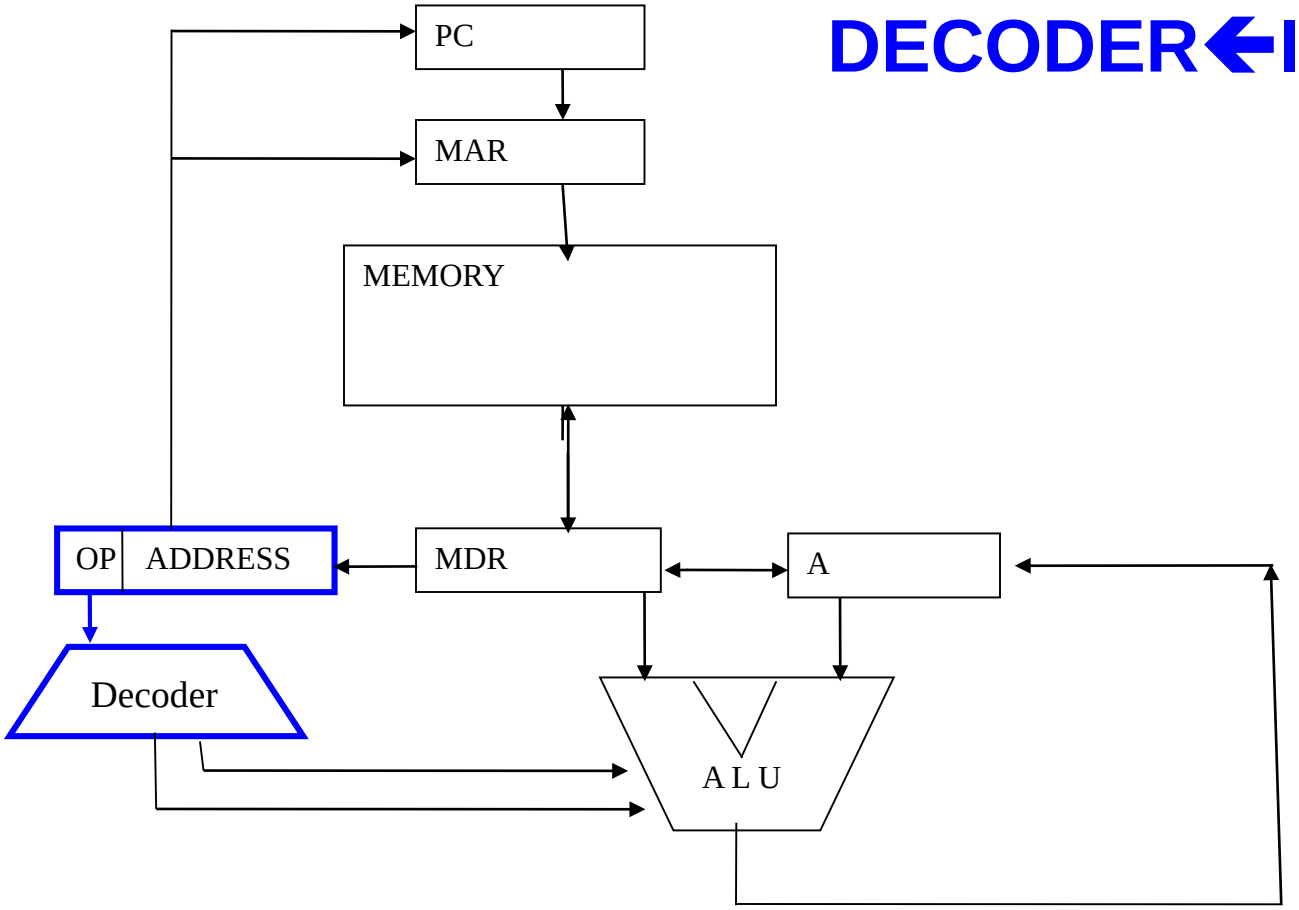
  IR.ADDR address

Instruction Format

- The **instruction format** of the accumulator machine is:

OP	ADDR
01	0000 0000 0010

Data Movement 4



Decoder

- If DECODER == 00, then the decoder will perform the fetch cycle.
- If DECODER != 00, then the decoder will perform the execution cycle.
- The operation is determined by DECODER.

00 Fetch Cycle

MAR	←	PC
MDR	←	MEM[MAR]
IR	←	MDR
PC	←	PC+1
DECODER	←	IR.OP

01 LOAD

MAR	←	IR.ADDR
MDR	←	MEM[MAR]
A	←	MDR
DECODER	←	00

02 ADD

MAR	←	IR.ADDR
MDR	←	MEM[MAR]
A	←	A + MDR
DECODER	←	00

Data Movement

We denote the transfer of MDR to MEM at location by

MEM[MAR] ← MDR

03 STORE

MAR	←	IR.ADDR
MDR	←	A
MEM[MAR]	←	MDR
DECODER	←	00

07 HALT

Instruction Set Architecture (ISA)

- **00 - FETCH**
  Fetches instruction from memory (hidden instruction)

Instruction Set Architecture

- **01 - LOAD <X>**

  Loads the contents of memory location “X” into A (A stands for Accumulator).

- **02 - ADD <X>**

  The data value stored at address “X” is added to A and the result is stored back in A.

- **03 - STORE <X>**

  Store the contents of A into memory location “X”.

- **04 - SUB <X>**

  The data value stored at address “X” is subtracted from A and the result is stored back in A.

Instruction Set Architecture

- **05 - IN <Device #>**

  A value from the input device is transferred into A.

- **06 - OUT <Device #>**

  Print A to the output device.

Device #	Device
5	Keyboard
7	Printer
9	Screen

  For instance you can write: **003 IN <5> "23"** where "23" is the value you are typing in.

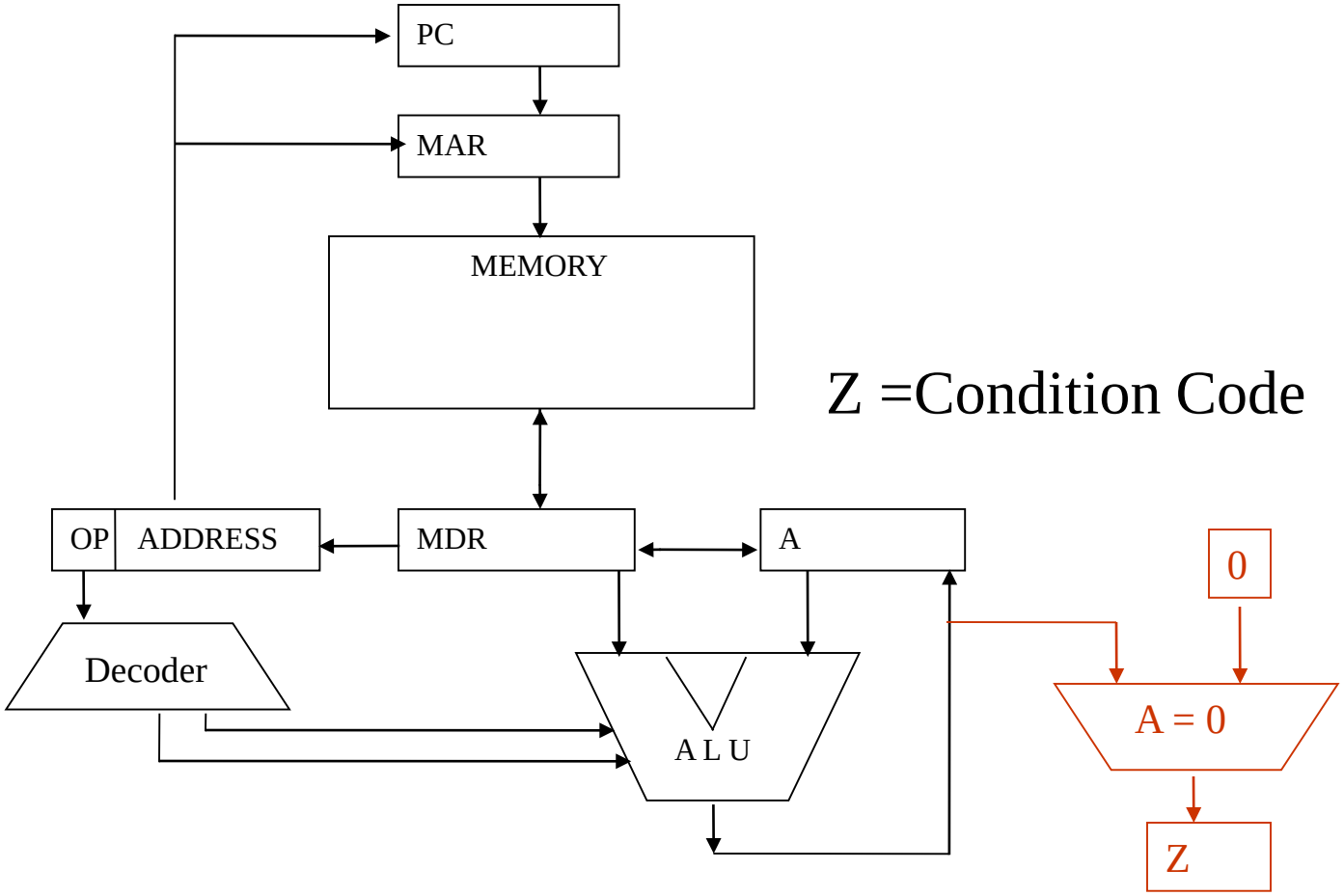
Instruction Set Architecture

- **07 - Halt**
  The machine stops execution of the program.
  (Return to the OS)

- **08 - JMP <X>**
  Causes an unconditional branch to address “X”.
  PC ← X

- **09 - SKIPZ**
  If A equals 0 then PC ← PC + 1 (the next instruction is skipped).
  (If the output of the ALU equals zero, the Z flag is set to 1. In this machine we test the flag and if Z = 1 the next instruction is skipped (PC ← PC + 1)

If the output of the ALU equals zero, the Z flag is set to 1



Instruction Set Architecture

- For this tiny assembly language, we are using only one condition code (CC) Z = 0 .
- Condition codes indicate the result of the most recent arithmetic operation
- Two more flags (CC) can be incorporated to test negative and positives values:

  G = 1 Positive value

  Z = 1 Zero

  L = 1 Negative value

Program State Word (Condition Codes - CC)

The PSW is a register in the CPU that provides the OS with information on the status of the running program

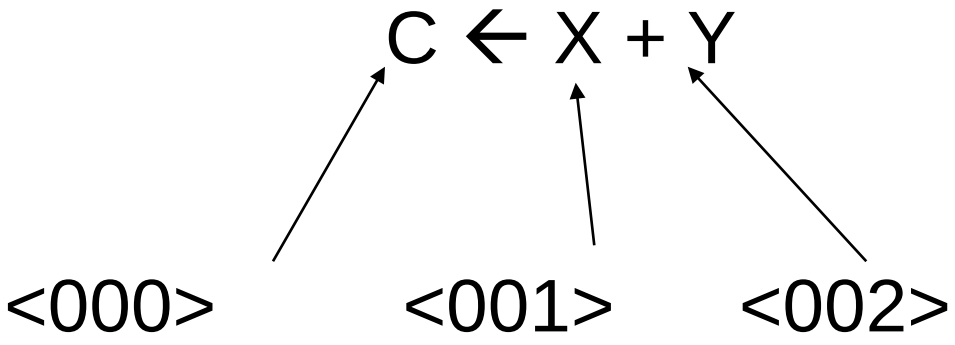
PC	Interrupt Flags						MASK	CC			Mode
	OV	MP	PI	TI	I/O	SVC	To be defined later	G	Z	L	

Instruction Descriptions

opcode	mnemonic		meaning		
0001	LOAD	<x>	A	←	MEM[x]
0010	ADD	<x>	A	←	A + MEM[x]
0011	STORE	<x>	MEM[x]	←	A
0100	SUB	<x>	A	←	A – MEM[x]
0101	IN	<Device_#>	A	←	Device
0110	OUT	<Device_#>	A	→	Device
0111	HALT		Stop		
1000	JMP	<x>	PC	←	x
1001	SKIPZ		Z = 1		skip
1010	SKIPG		G = 1		skip
1011	SKIPL		L= 1		skip

Assembly Language Programming Examples

Assign a memory location to each variable:



Assembly Language Programming Examples

Memory

000	1245
001	1755
002	0000
003	Load <000>
004	Add <001>
005	Store <002>
006	Halt

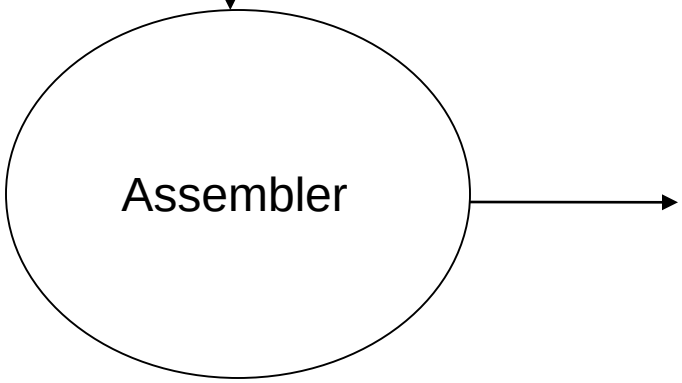
After execution →

Memory

000	1245
001	1755
002	3000
003	Load <000>
004	Add <001>
005	Store <002>
006	Halt

Assembler: translate symbolic code to executable code (binary)

Assembly Language
003 **Load <000>**
004 Add <001>
005 Store <002>
006 Halt



01 → LOAD
02 → ADD
03 → STORE
04 → SUB
05 → IN
06 → OUT
07 → HALT
08 → JMP
09 → SKIPZ

In binary

003	**0001 0000 0000 0000**
004	0010 0000 0000 0001
005	0011 000000000010
006	0111 000000000000

Assembler Directives

The next step to improve our assembly language is the incorporation of **pseudo-ops** (**assembler directives**) to invoke *a* special service from the assembler (pseudo-operations do not generate code)

.begin → tells the assembler where the program starts

.data → tells the assembler to reserve a memory location.

.end → tells the assembler where the program ends.

Labels are symbolic names used to identify memory locations.

Assembler Directives

This is an example of the usage of **assembler directives**

.begin

Assembly language instructions

halt return to OS

.data tells to reserve a memory location

.end tells the assembler where the program ends

Assembly Language Programming Examples

Label		opcode	address
start		.begin	
	in	x005	
	store	a	
	in	x005	
	store	b	
	load	a	
	sub	TWO	
	add	b	
	out	x009	
	halt		
a	.data	0	
b	.data	0	
TWO	.data	2	
	.end	start	

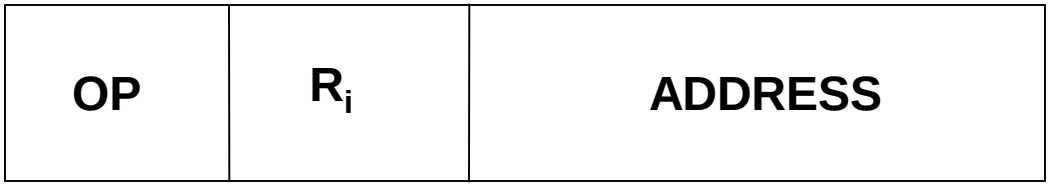
Text section (code)

Data section

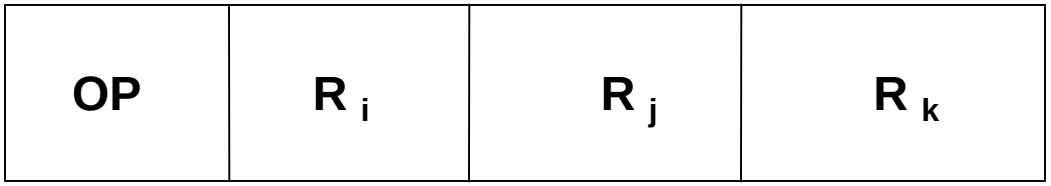
Load/Store Architecture

A load/store architecture has a “register file” in the CPU and it uses three instruction formats. Therefore, its assembly language is different from the one of the accumulator machine**.**

JMP <address>

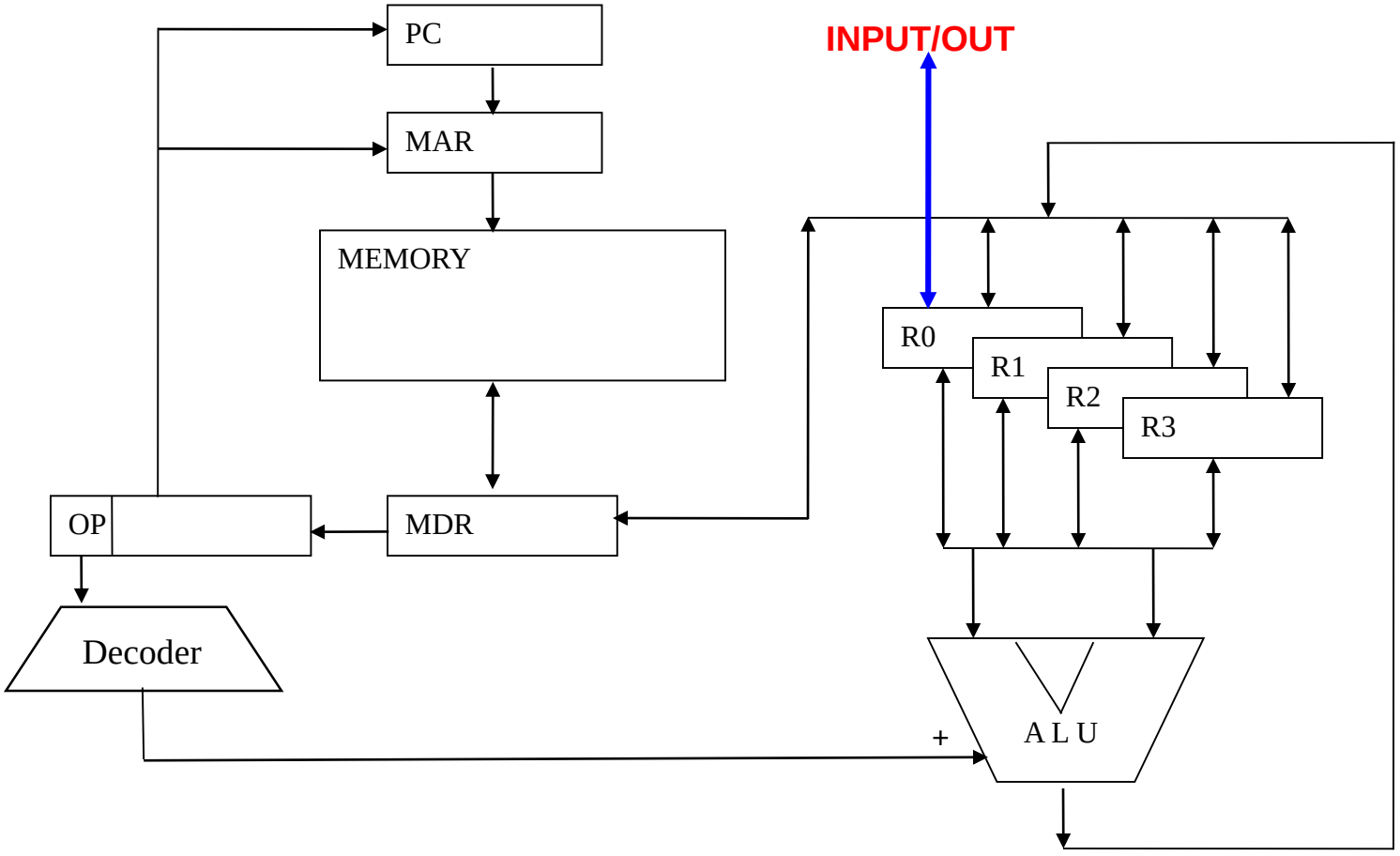


Load R3, <address>



Add R3, R2, R1

Load/Store Architecture



Multiplying two numbers

```
Label                    opcode   address
start     .begin
          in             x005
          store          a
          in             x005
          store          b
here      load           result
          add            a
          store          result
          load           b
          sub            ONE
          store          b
          skipz
          jmp            here
          load           result
          out            x009
          halt
a         .data          0
b         .data          0
ONE       .data          1
result                   .data          0
          .end           start
```

One address Architecture
(six memory access inside the loop)

```
Label                    opcode   address
start     .begin
          in             x005
          store          R0, a
          in             x005
          store          R0, b
          load           R2, result
          load           R3, a
          load           R0, b
          load           R1, ONE
here      add            R2, R2, R3
          sub            R0, R0, R1
          skipz
          jmp            here
          store          R2, result
          load           R0, result
          out            x009
          halt
a         .data          0
b         .data          0
ONE       .data          1
result                   .data          0
          .end           start
```

Load/Store architecture
(no memory access inside the loop)

Next time will talk about the PM/0 virtual machine