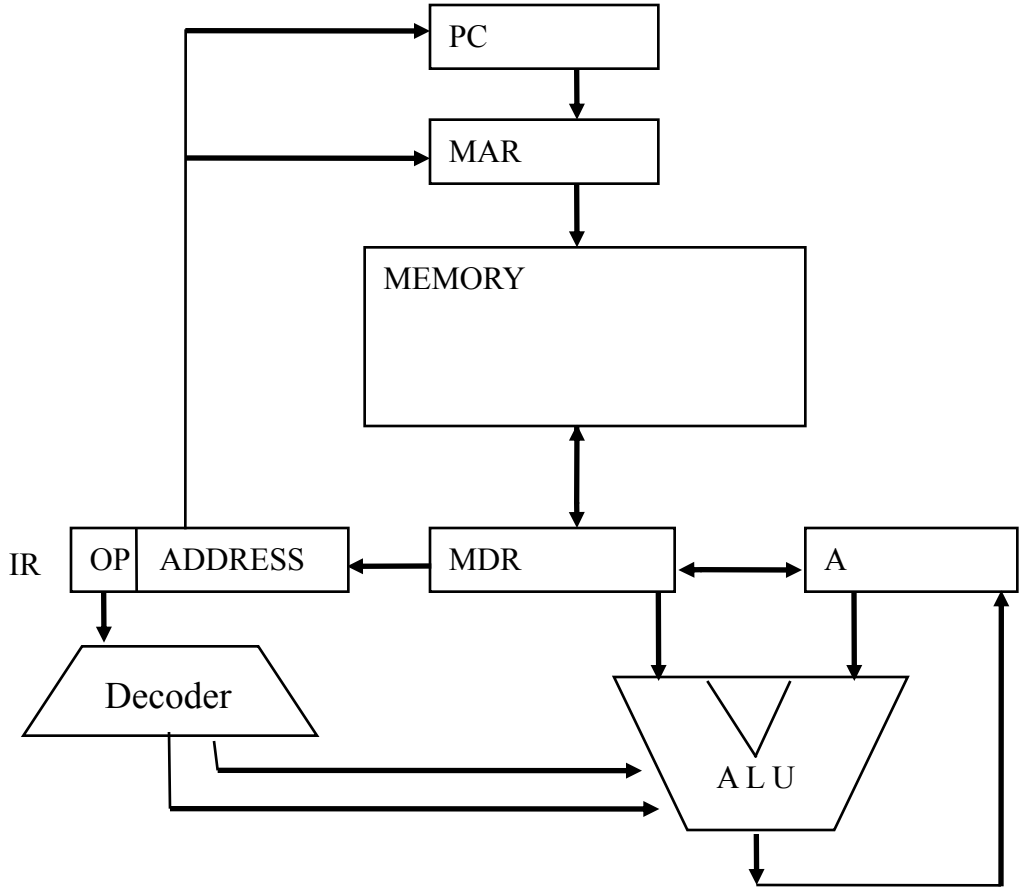
COP 3402 Systems Software

The processor as an instruction interpreter.

Outline

1. The structure of a tiny computer.
2. A program as an isolated system.
3. The instruction format.
4. Assembly language.

Von-Neumann Machine (VN)



Instruction Cycle

- The Instruction Cycle, or Machine Cycle, in the Von-Neumann Machine (VN) is composed of 2 steps:
  - 1. **Fetch Cycle:** Instruction is retrieved from memory.
  - 2. **Execution Cycle:** Instruction is executed.

- **A simple Hardware Description Language will be used in order to understand how instructions are executed in VN.**

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.

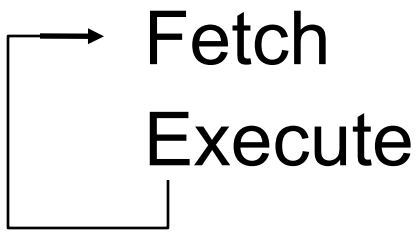
Definition

- **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-Execute Cycle

- In the VN, the Instruction Cycle is defined by the following loop:

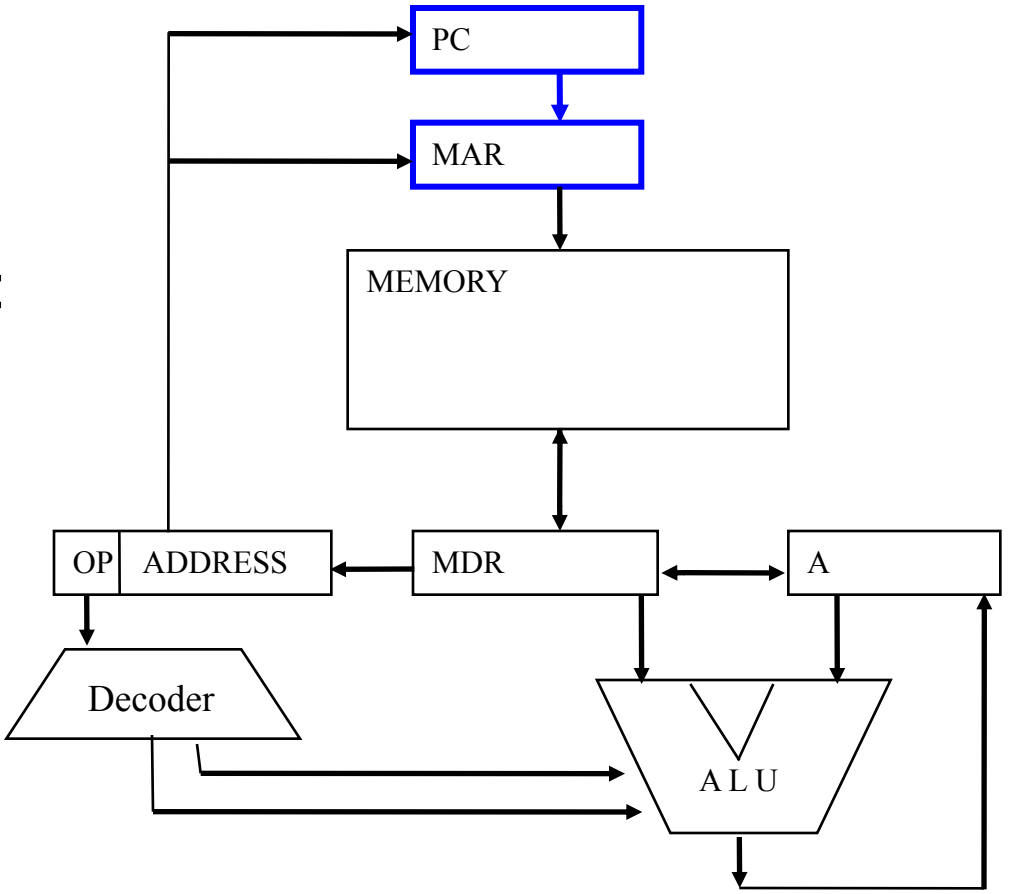


- In order to fully explain the Fetch Cycle we need to study the details of the VN data flow. The data flow consists of 4 steps.

Data Movement 1

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

  MAR←PC

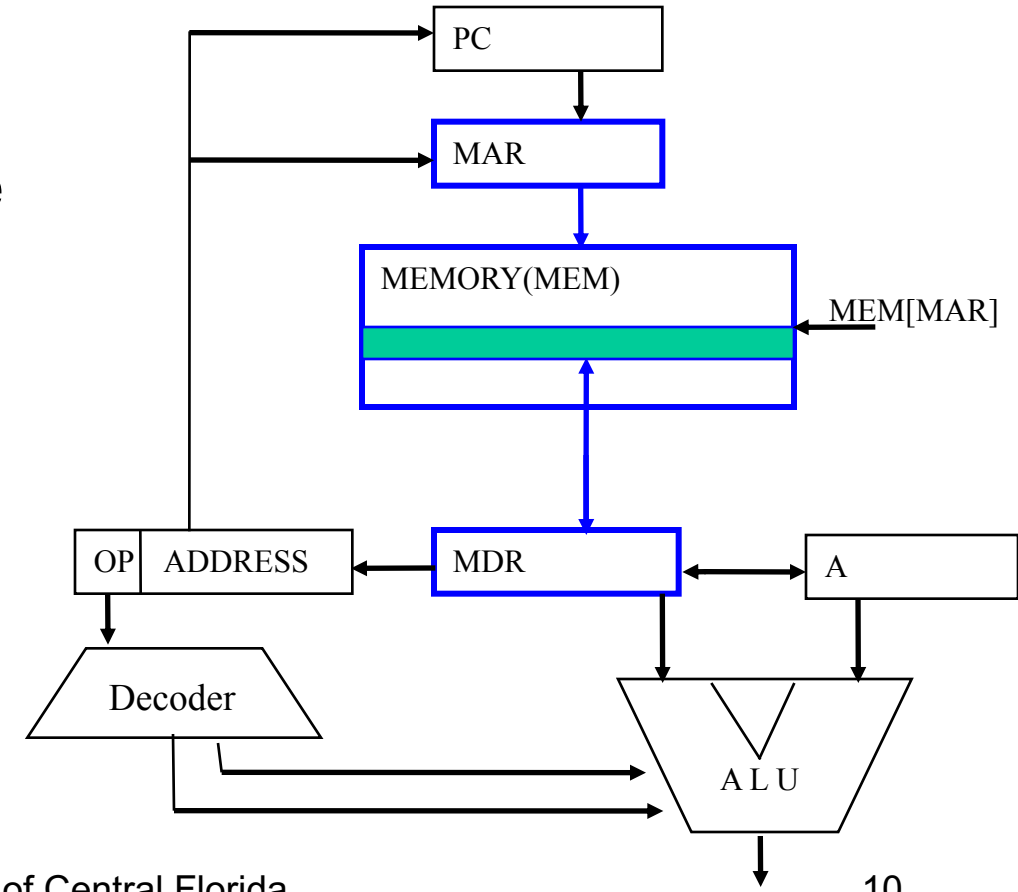


Data Movement 2

- 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 2 (Cont.)

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

  MEM [MAR] ←MDR

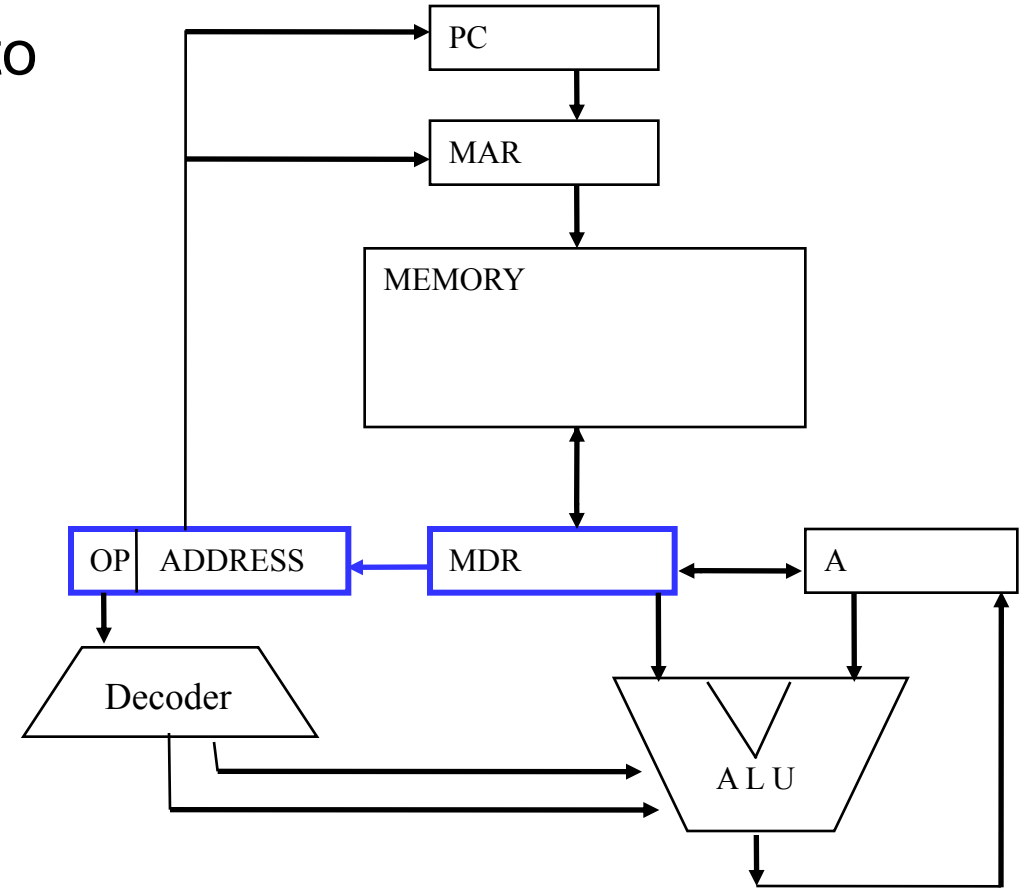
  *see previous slide for diagram

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

Data Movement 3

- Transferring the contents of MDR into IR is indicated as:

  IR←MDR



Instruction Register Properties

- The Instruction Register (IR) has two fields:

  Operation (OP) and the ADDRESS.

- These fields can be accessed using the selector operator "**.**"

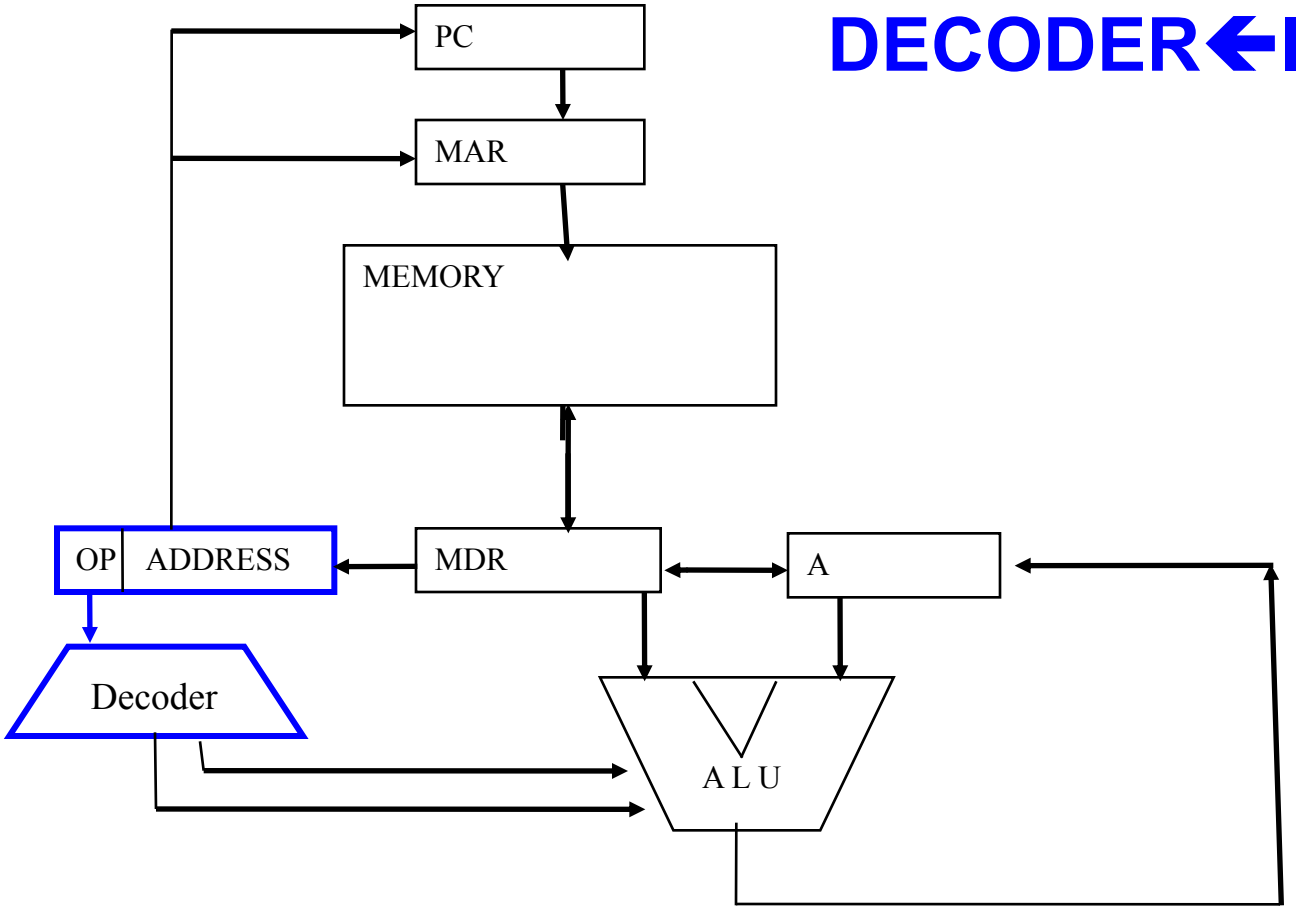
Data Movement 4

- The Operation portion of the field is accessed as IR.OP
- The operation field of the IR register is sent out to the DECODER using:

  DECODER←IR.OP

- DECODER: If the value of IR.OP==00, then the decoder can be set to execute the fetch cycle again.

Data Movement 4 Cont.



DECODER←IR.OP

Instruction Cycle

- The Instruction Cycle has 2 components.
- Fetch Cycle which retrieves the instruction from memory.
- Execution Cycle which carries out the execution of the instruction retrieved.

00 Fetch Cycle

1. MAR ← PC
2. MDR ← MEM[MAR]
3. IR ← MDR
4. PC ← PC+1
5. DECODER ← IR.OP

1. Copy contents of PC into MAR
2. Load content of memory location into MDR
3. Copy value stored in MDR to IR
4. Increment PC Register

Execution: 01 LOAD

1. MAR ←IR.ADDR
2. MDR ←MEM[MAR]
3. A ←MDR
4. DECODER ←00

1. Copy the IR address value field into MAR
2. Load the content of a memory location into MDR
3. Copy content of MDR into A register
4. Set Decoder to execute Fetch Cycle

Execution: 02 ADD

1. MAR ←IR.ADDR
2. MDR ←MEM[MAR]
3. A ←A + MDR
4. DECODER ←00

1. Copy the IR address value field into MAR
2. Load content of memory location to MDR
3. Add contents of MDR and A register and store result into A
4. Set Decoder to execute Fetch cycle

Execution: 03 STORE

1. MAR ←IR.ADDR
2. MDR ←A
3. MEM[MAR] ←MDR
4. DECODER ←00

1. Copy the IR address value field into MAR
2. Copy A register contents into MDR
3. Copy content of MDR into a memory location
4. Set Decoder to execute fetch cycle

Execution: 07 HALT

1. STOP	1. Program ends normally

Instruction Set Architecture (ISA)

00 Fetch (hidden instruction)

MAR ←PC

MDR ←MEM[MAR]

IR ←MDR

PC ←PC+1

DECODER ←IR.OP

02 Add

MAR←IR.Address

MDR ←MEM[MAR]

A ← A + MDR

DECODER ←00

01 Load

MAR←IR.Address

MDR ←MEM[MAR]

A ← MDR

DECODER←00

03 Store

MAR←IR.Address

MDR ←A

MEM[MAR] ←MDR

DECODER ←00

07 Halt

One Address Architecture (instruction format)

- The instruction format of this one-address architecture is:

OP	ADDRESS
LOAD	0000 0000 0010

Instruction Set Architecture

- **01 - LOAD <X>**
  Loads the contents of memory location “X” into the A (A stands for Accumulator).

- **02 - ADD <X>**
  The data value stored at address “X” is added to the A and the result is stored back in the A.

- **03 - STORE <X>**
  Store the contents of the A into memory location “X”.

- **04 - SUB <X>**
  Subtracts the value located at address “X” from the A and stored the result back in the A.

Instruction Set Architecture

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

  A value from the input device is transferred into the AC.

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

  Print out the contents of the AC in 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.

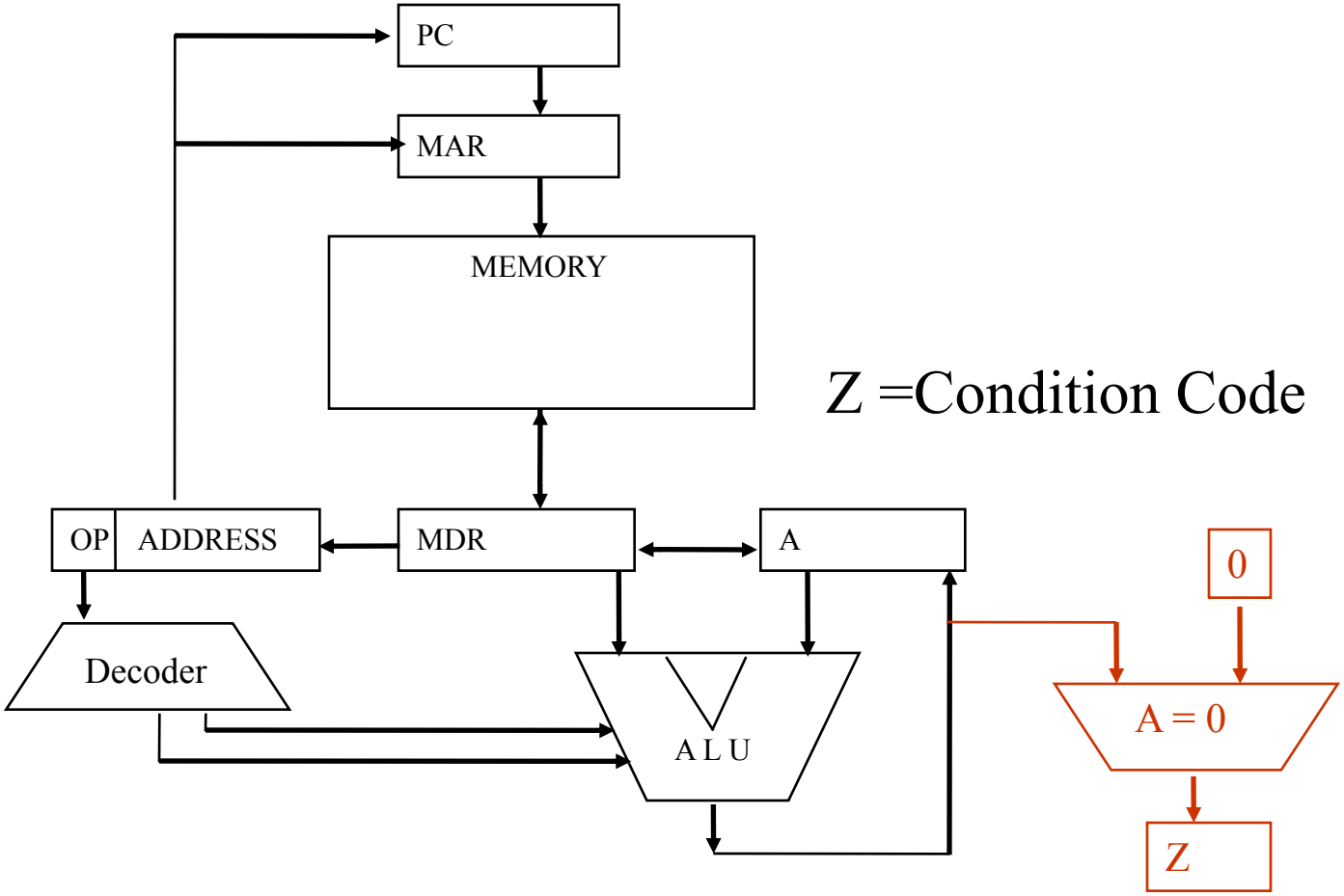
Instruction Set Architecture

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

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

- **09 - SKIPZ**
  If the contents of Accumulator = 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 skiped (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	

In addition to the Z flag, we can incorporate two more flags:

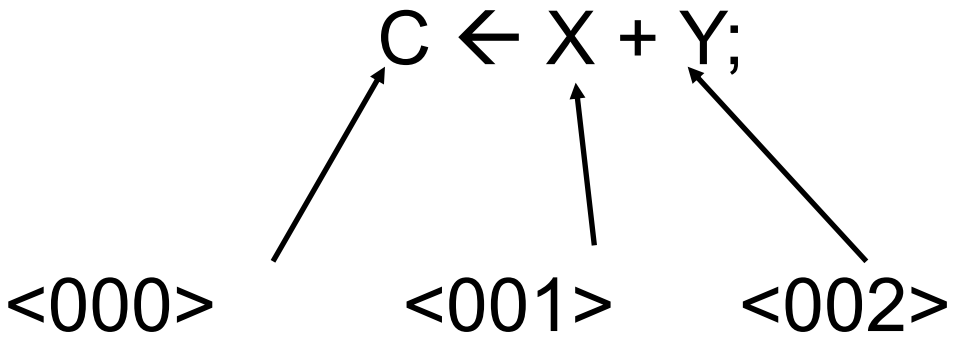
1) G meaning "greater than zero"
2) L meaning "less than zero"

ISA
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 ← read from Device
0110	OUT <Device_#>	A → output to Device
0111	HALT	Stop
1000	JMP <x>	PC ← x
1001	SKIPZ	If Z = 1 Skip next instruction
1010	SKIPG	If G = 1 Skip next instruction
1011	SKIPL	If L = 1 Skip next instruction

Assembly language Programming examples

Assign a memory location to each variable:



If it is necessary to use temporary memory locations, assign labels (names) to them.

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

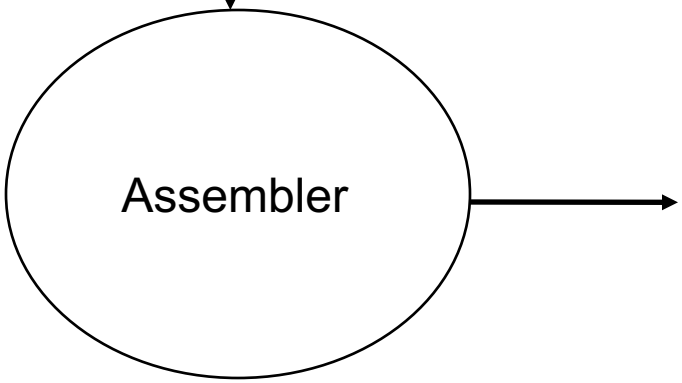
One Address Architecture

- The instruction format of this one-address architecture consists of 16 bits: 4 bits to represent instructions and 12 bits for addresses :

OP	ADDRESS
0001	0000 0001 0001

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 → tell the assembler where the program starts

  .data → 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 *(to reserve a memory location)*

.end **(** *tells the assembler where the program ends)*

note:
the directive .end can be used to indicate where the program starts (for example: ".end <insert label here>"

Assembly language Programming examples

Label	opcode	address	
start		.begin	
	in	x005	
	store	a	
	in	x005	
	store	b	
	load	a	Text section (code)
	sub	TWO	
	add	b	
	out	x009	
	halt		
a	.data	0	
b	.data	0	Data section
TWO		.data 2	
	.end	start	

LOAD/STORE ARCHITECTURE

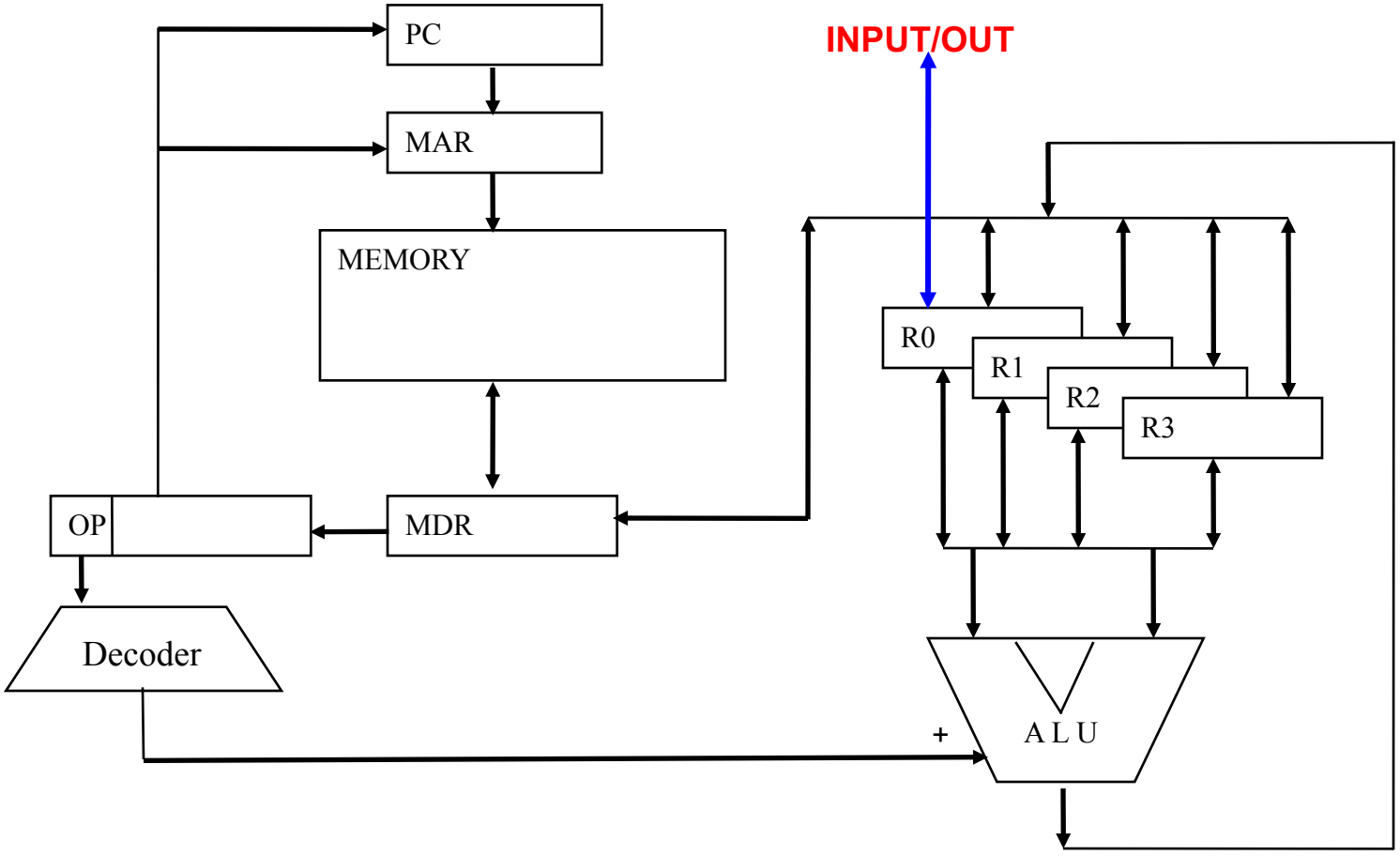
A load/store architecture has a "register file" in the CPU and it uses three instruction formats. Therefore, its assembly language is different to the one of the accumulator machine.

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)