

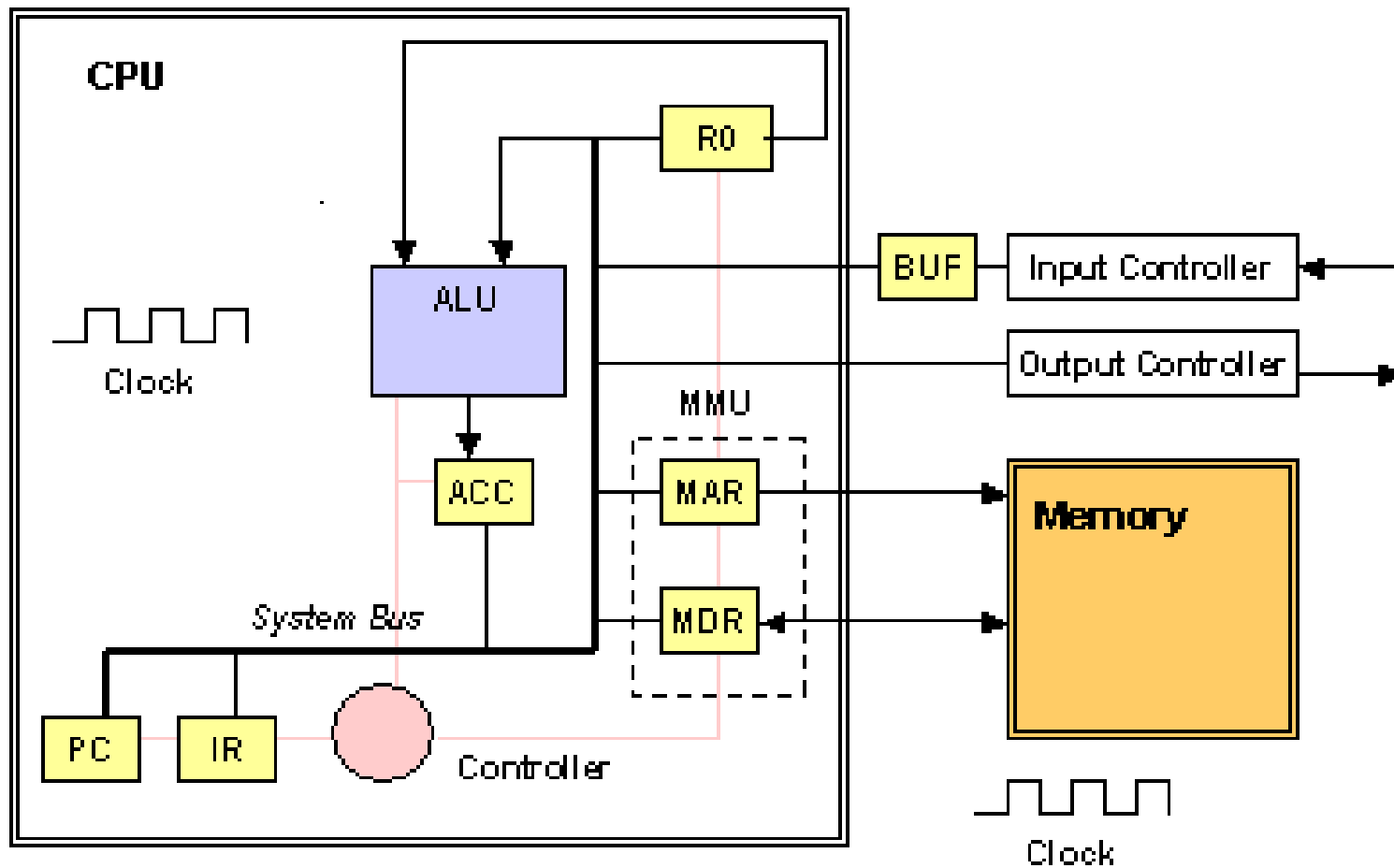
COMPS266F Chapter 5

Case Study: Little Man's Computer



Copyright © 2014 by Dr. Andrew Lui

Current Programmable Computer Design





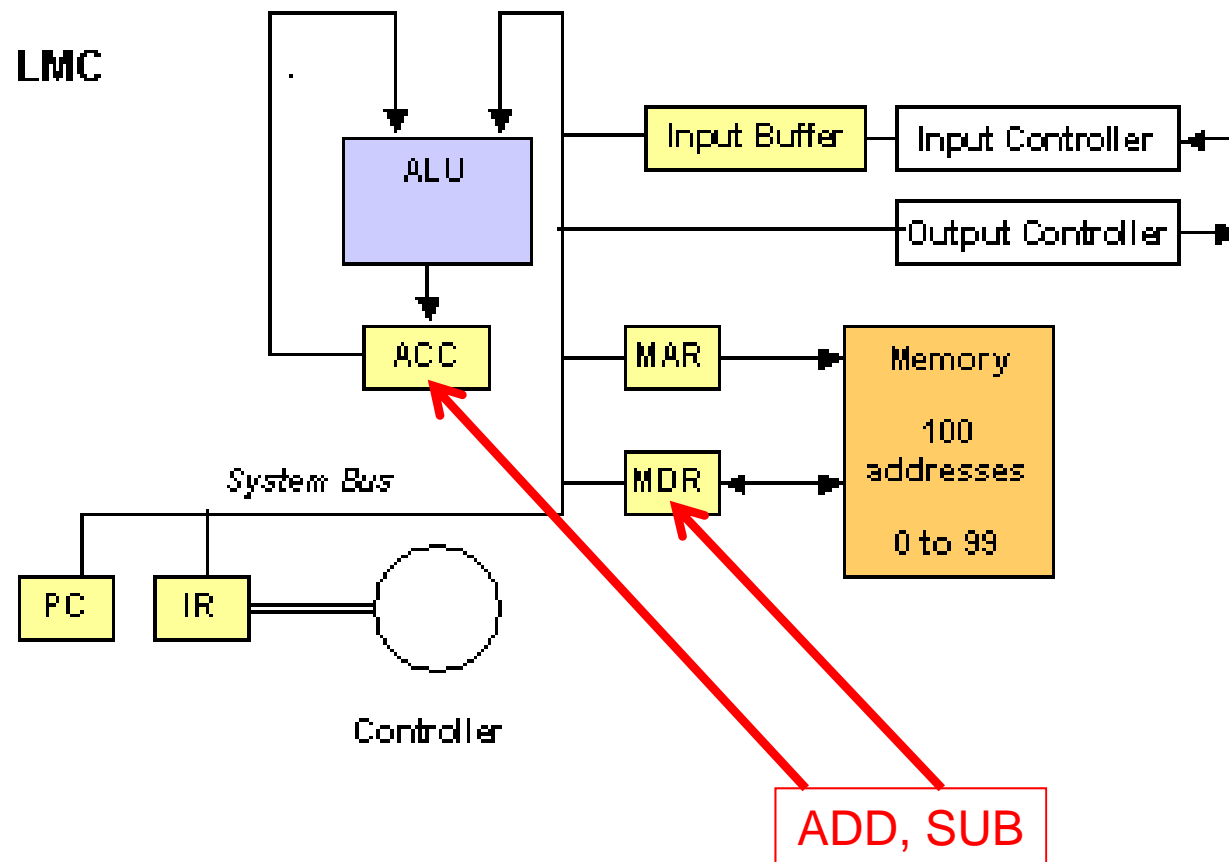
a case study of a real computer

Little Man's Computer (LMC)



- An *operational* computer
- Some deviations from the current design
 - R0 is removed
 - ACC is retained
 - ADD/SUB operand must involve the ACC and another from MDR
 - A decimal computer
 - Data and instructions use digital representation

Little Man's Computer (LMC)



Major Features of LMC



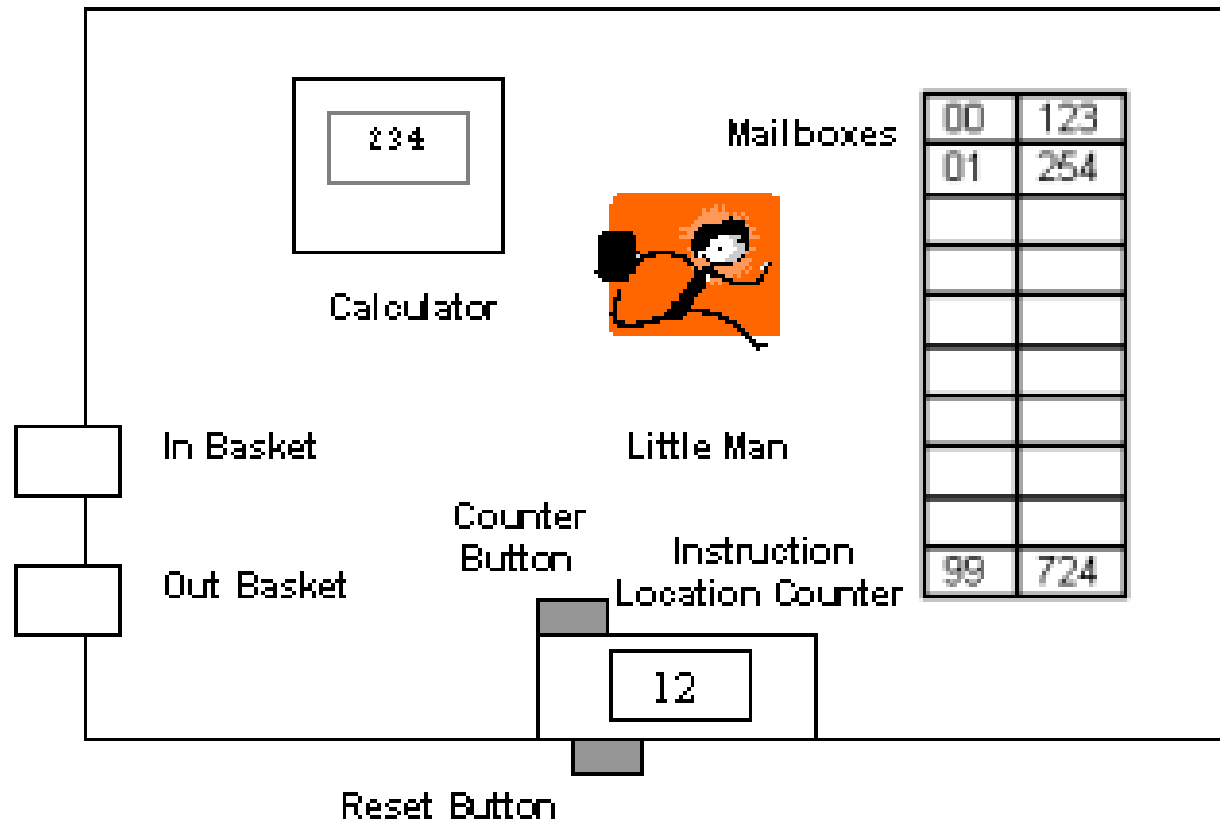
- Three-digit decimal representation
- The registers hold **3-digit** positive decimal
- The ALU supports **addition** and **subtraction**.
- Overflows are not reported and the carry is simply discarded.
- Two types of status (ACC): zero and positive.
- Memory system has 100 addresses, from 00 to 99.
 - Each address store a 3-digit positive decimal number.

Purpose of LMC



- LMC as an analogy of a programmable computer
 - LMC is a common teaching tool used by many universities

LMC Design



Features of LMC

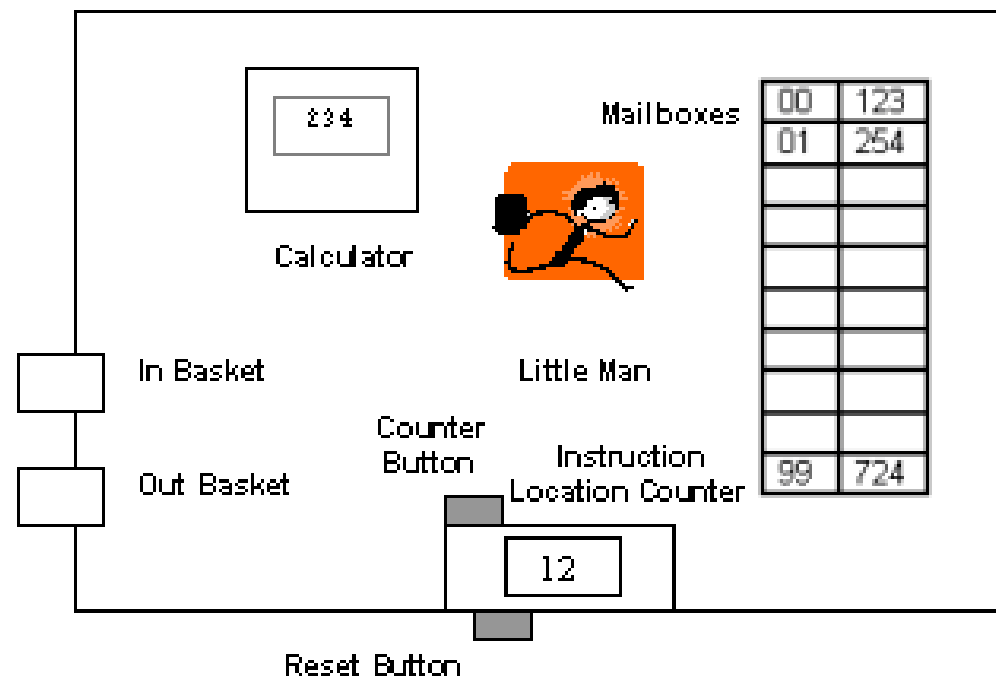


- Hidden inside a room
 - Few specific connections to the outside world only.
- 100 mailboxes
- Calculator is for doing simple arithmetic and storing data temporarily. Display is 3 digit wide.
- Location counter is a hand counter for keeping track of work.
 - Two digits number (from 00 to 99)
 - The counter has a reset button

Features of LMC



- Connections to the outside worlds are the in-basket and the out-basket.





LMC instruction set design

LMC Instruction Set Design



- Instructions for program composition
 - Types of instructions
 - Moving data from and to the memory
 - Input and output
 - Arithmetic and logic operations
 - Stopping the program
 - Format of instructions
 - Single 3-digit instruction
 - Two 3-digit instruction

LMC Instruction Set Design





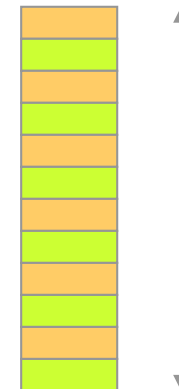
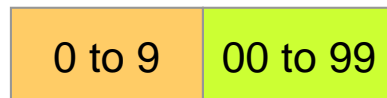
Format: Single 3-digit instruction



Format: Two 3-digit instruction



 Op-code
 Operand



LMC Instruction Set Design



■ Single 3-digit instruction format

<i>Instruction</i>	<i>Code</i>	<i>Remarks</i>
Load	5XX	Load from mailbox to calculator
Store	3XX	Store in mailbox from calculator
Add	1XX	Add from mailbox to calculator
Subtract	2XX	Subtract from calculator the mailbox value.
Input	901	Input
Output	902	Output
Halt	000	Coffee Break or Halt
Branch	6XX	Branch unconditionally
Branch if Zero	7XX	Branch if zero
Branch if Positive or Zero	8XX	Branch if positive or zero
Data		A location for data storage



demo of LMC instructions

Load/Store Instructions



- Memory operations (5XX, 3XX)
 - One operand as the address of memory operation
 - Load instruction: the value in the *calculator* is overwritten
 - Store instruction: the value in the *calculator* remains intact

Add/Subtract Instructions



- Memory and arithmetic operations (1XX, 2XX)
 - The operand in instruction as the address of memory operation
 - Calculator is the first operand and the memory address is the second operand
 - The value in the calculator is overwritten
 - Addition may end up overflow
 - No status for this error
 - Subtraction may end up with a negative value

Input and Output



- Input is buffered in the basket (901)
 - The IN instruction moves the data from basket and overwrite the value stored in *calculator*
- Output (902)
 - The OUT instruction moves the data stored in *calculator* to out basket

Branch Instructions



- LMC programs are executed sequentially
 - Program Counter (PC) [location counter] always increases by one after executing one instruction
- Branch instructions allow the PC to change to any value
- Two types
 - Unconditional branch
 - Conditional branch

Example LMC Program



```
00 901 ; IN
01 902 ; OUT
02 000 ; HALT
```

Input and Print

Address should start at 0

Example LMC Program



```
00 901 ; INPUT #1
01 399 ; STORE IN ADDR 99
02 901 ; INPUT #2
03 199 ; ADD ADDR 99
04 902 ; OUTPUT
05 000 ; STOP
99 ; DATA
```

Adding Two Numbers

***Memory reserved for data
storage***

Example LMC Program



```
00 901 ; INPUT #1
01 311 ; STORE IN ADDR 11
02 901 ; INPUT #2
03 312 ; STORE IN ADDR 12
04 211 ; #2 - #1
05 808 ; BRANCH IF POSITIVE TO 08
06 511 ; LOAD #1
07 609 ; BRANCH TO 09
08 512 ; LOAD #2
09 902 ; OUTPUT
10 000 ; STOP
11 000 ; DATA
12 000 ; DATA
```

Comparing Two Numbers

Prints the Larger Number



Example LMC Program

```
00 BR 50
50 IN
51 STO 00 ; storing the entered instruction from 00 onwards
52 BRZ 57 ; if the input is 000, branch to 57 to begin execution
53 LDA 51
54 ADD 99
55 STO 51
56 BR 50
57 LDA 51
58 STO 60
59 LDA 98
60 STO 00
61 LDA 97
62 STO 51
63 BR 00
97 DAT 300
98 DAT 650
99 DAT 01
```

Program Loader

Program the LMC



- Using the LMC emulator
 - Enter instructions starting at address 0
 - Press reset button
- Comments
 - Comments are placed after semicolon on a line
- Mnemonics
 - Abbreviations for the instructions, easier to read
 - Assembler to convert LMC programs in mnemonics to LMC code
 - A simple programming language
 - LMC code generated should start at address 0

Mnemonics



<i>Mnemonics</i>	<i>Code</i>	<i>Remarks</i>
LDA	5XX	Load from mailbox to calculator
STO	3XX	Store in mailbox from calculator
ADD	1XX	Add from mailbox to calculator
SUB	2XX	Subtract from calculator the mailbox value.
IN	901	Input
OUT	902	Output
COB or HLT	000	Coffee Break or Halt
BR	6XX	Branch unconditionally
BRZ	7XX	Branch if zero
BRP	8XX	Branch if positive or zero
DAT		A location for data storage



high level programming languages and LMC

High Level Programming



- Compilers and linker to convert a program written in high level program into CPU instructions
 - Code generation

Example: C Program



```
int x;  
scanf("%d", &x);  
if (x > 5)  
    printf("0");  
else  
    printf("1");
```



Example: C Program

```
00 IN ;      Input data
01 STO 99; Store x in 99
02 SUB 98;
03 BRP 06; x > 5?
04 LDA 97; Load '1'
05 BR 08
06 BRZ 04; x = 5?
07 LDA 96; Load '0'
08 OUT;      Print
09 HLT;
96 DAT 00; Constant 0
97 DAT 01; Constant 1
98 DAT 05; Constant 5
99 DAT ; Variable x
```

Output of a C Compiler for LMC

Example: C Program



```
int x = 0;  
while (x < 10) {  
    printf("%d", x);  
    x++;  
}
```

A repetitive structure



Example: C Program

```
00 LDA 99;  
01 SUB 98; Calculate X - 10  
02 BRZ 08; Jump to after the loop  
03 LDA 99; Load X  
04 OUT ; Print X  
05 ADD 97; X = X + 1  
06 STO 99;  
07 BR 00; Jump to the top of loop  
08 HLT;  
97 DAT 01; Constant 1  
98 DAT 10; Constant 10  
99 DAT 0; Variable X
```

Output of a C Compiler for LMC



Review of Von Neumann architecture

Benefits of Stored Program Architecture



- **Simpler computer design**
 - *Single* memory system for data and program instructions
- **Self-modifying instruction**
 - Allow programmer to write instructions that modify instructions during execution
 - Programmer can create other instructions during execution
 - Reduce program size
 - Improve programmability

Hazards of Stored Program Architecture



- Both data and program exists in the memory system.
 - No indicator to signify whether it is an instruction or data
 - Up to the programmer to take care
- A fetched instruction is assumed to be valid
 - If invalid (eg 903), cause a system exception
 - A data (eg 512) coincides with a valid instruction
 - The LMC cannot tell one from another
- Constants in LMC programs can also be instructions
 - It can be used as the stem of a self-modifying instruction



Hazards of Stored Program Architecture

```
00 BR 50
50 IN
51 STO 00 ← ; storing the entered instruction from 00 onwards
52 BRZ 57 ; if the input is 000, branch to 57 to begin execution
53 LDA 51
54 ADD 99
55 STO 51
56 BR 50
57 LDA 51
58 STO 60
59 LDA 98
60 STO 00
61 LDA 97
62 STO 51
63 BR 00
97 DAT 300 ; store
98 DAT 650 ; branch to 50 (last inst.)
99 DAT 01
```

300,301,302,...

These are defined as stems of instructions

Von Neumann bottleneck



- Program or data only
 - Connection between CPU and memory carries both data and program
- Data not ready
 - Instructions cannot be executed (example: LDA 77)
- The limited throughput between CPU and memory



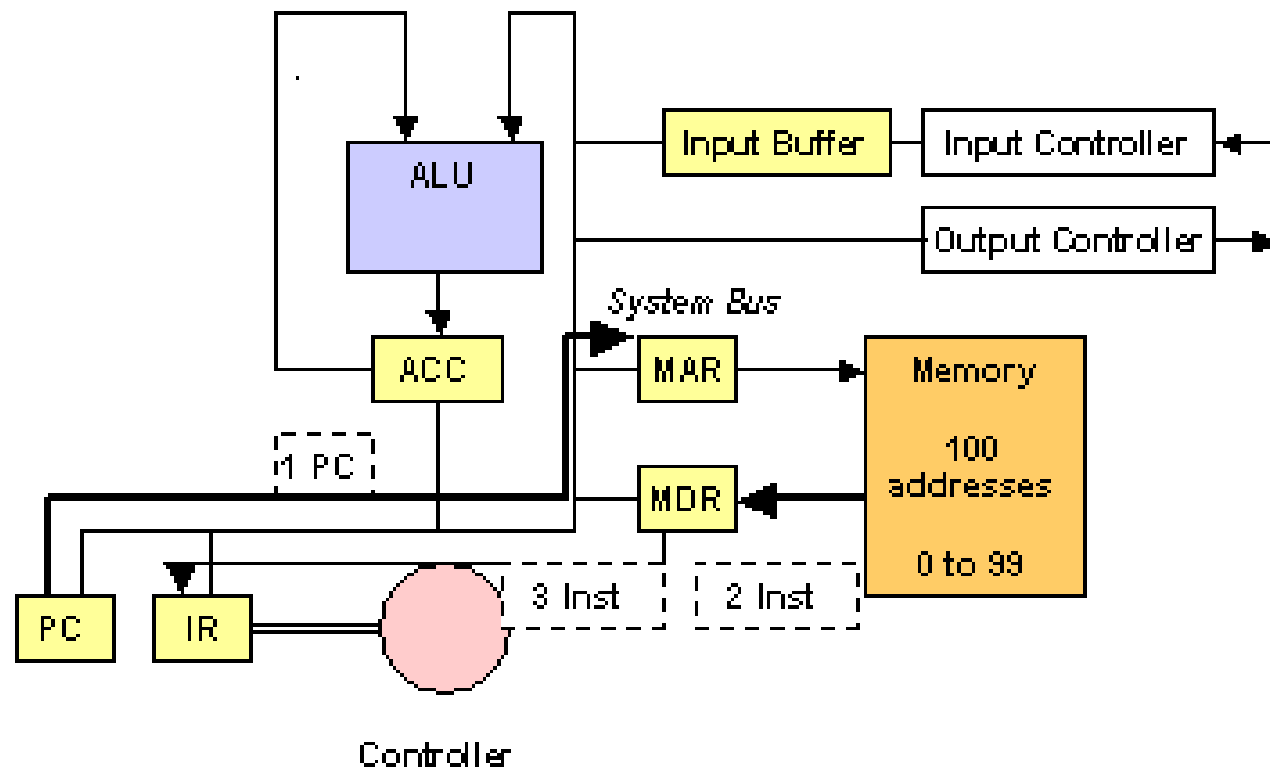
LMC in operation

Operation Cycles of LMC



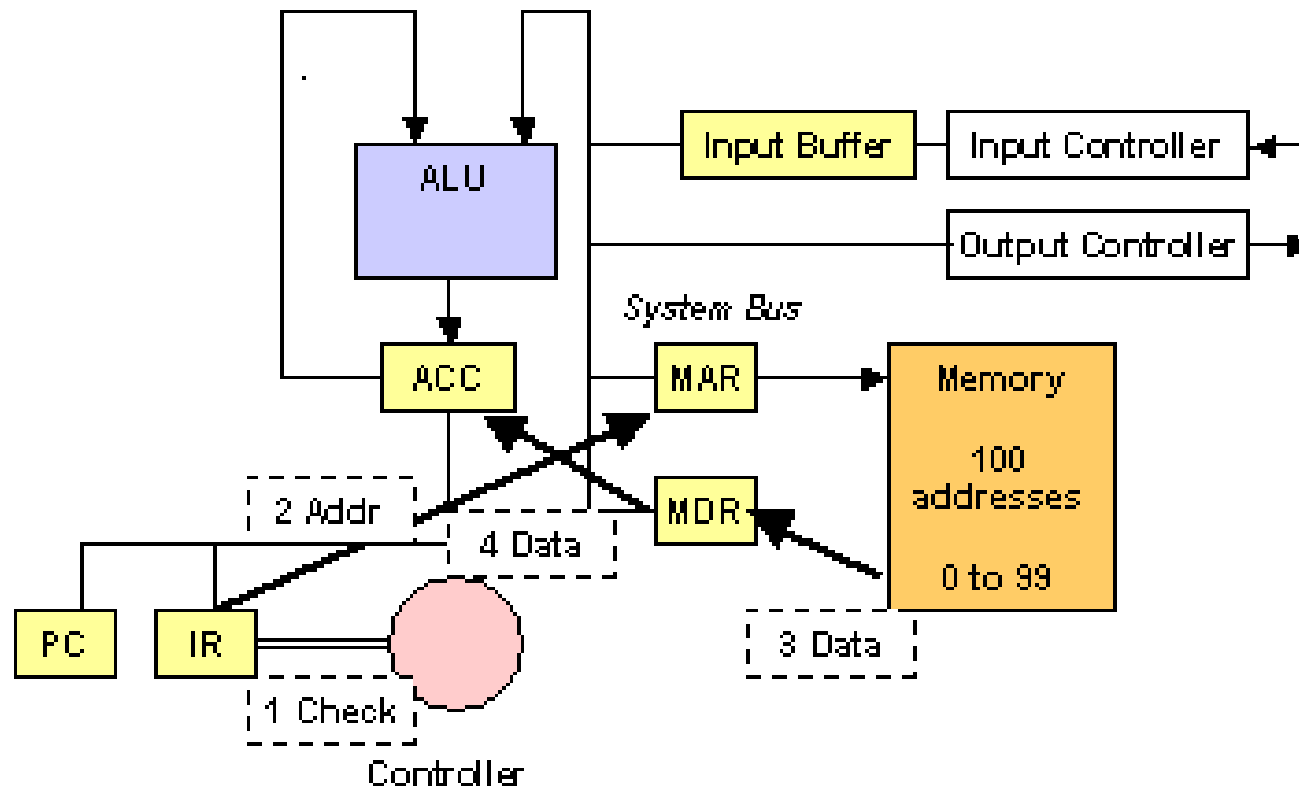
- LMC in operation is in a indefinite cycle
 - Each cycle executes one instruction
 - Each cycle has two parts
 - Part 1: fixed, regardless of the instruction
 - Part 2: dependent on the instruction

Operation Cycle of LMC: First Part



The aim is to move the next instruction to IR (LDR 70 / 570)

Operation Cycle of LMC: Second Part



The aim is to execute the instruction stored in IR (LDR 70 / 570)



data movement perspective of computer operation

A Data Movement Perspective



- Computer operation: moving data from one component to another
 - Execute instructions faster if data movement is faster
 - Execute instructions faster if data movement happen together

A Data Movement Perspective



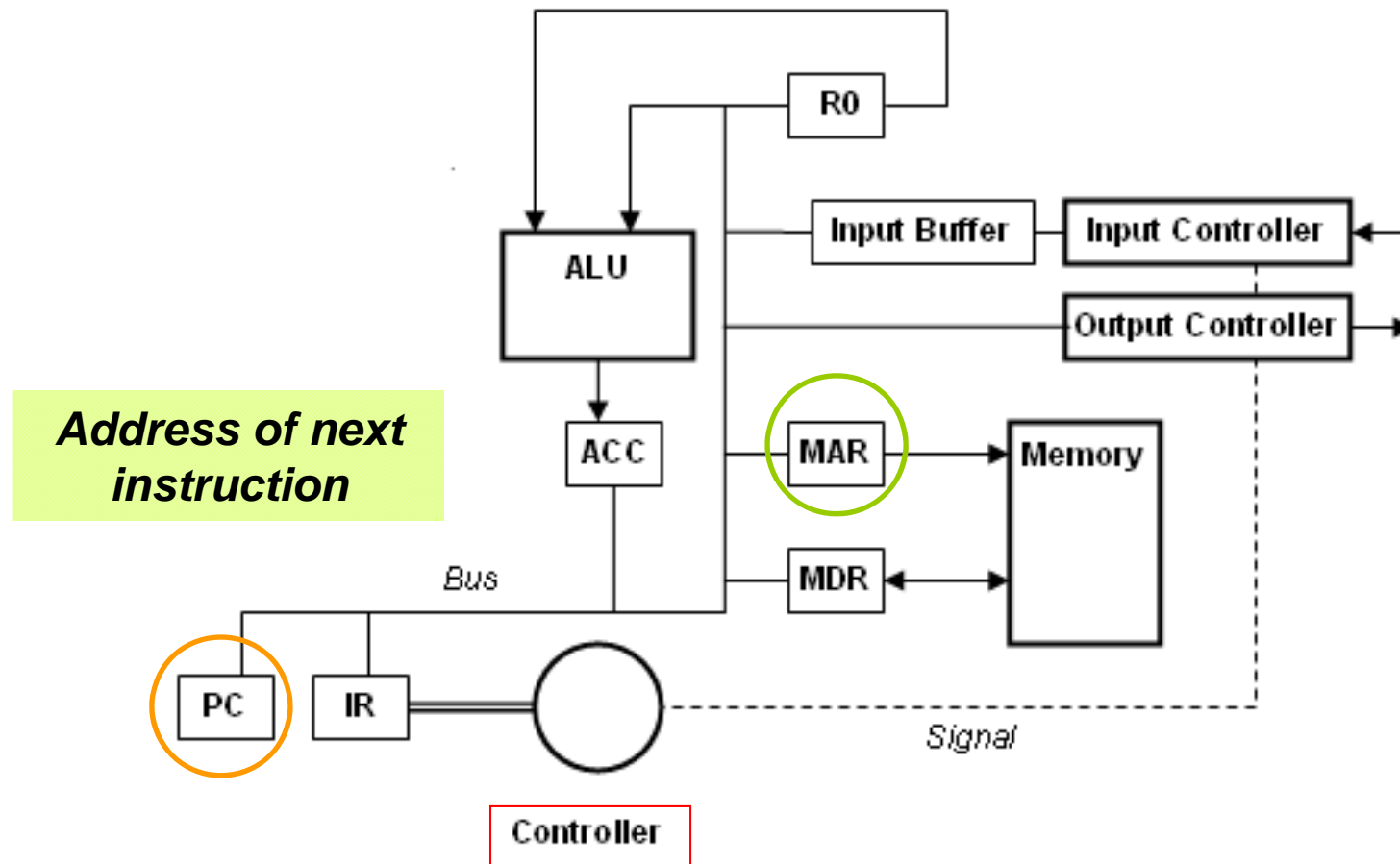
- Common data movement patterns involved in LMC instruction execution:
 - PC to MAR (the address of next instruction)
 - Memory System to MDR (memory read data)
 - MDR to Memory System (memory write data)
 - MDR to IR (the current instruction)
 - MDR to ACC (data)
 - ACC to MDR (data)
 - MDR to MAR (memory address)

A Data Movement Perspective

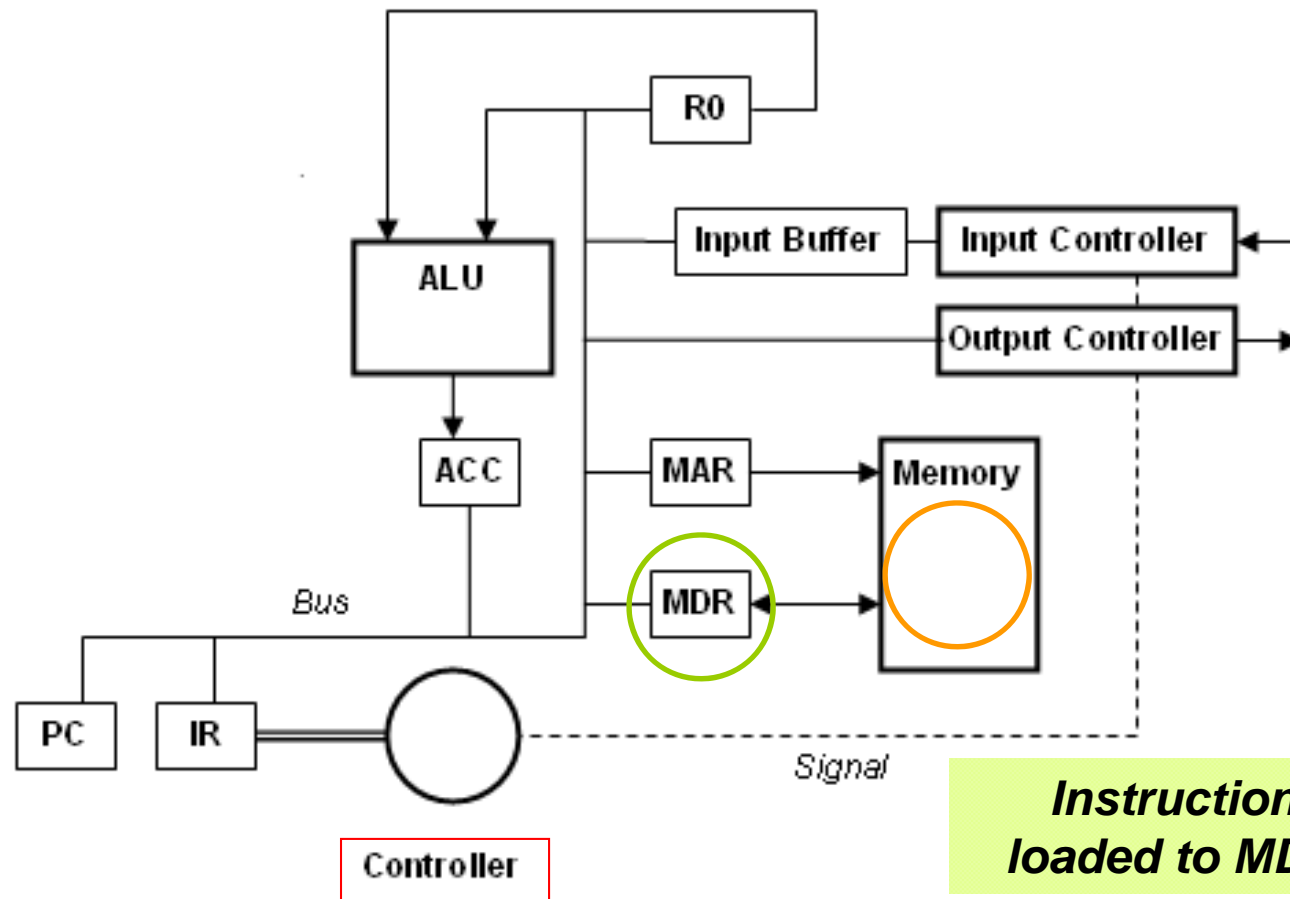


- Executing LOAD instruction
 - The address to load is part of the LOAD instruction
 - The destination is the Accumulator (ACC)
 - There are 8 steps
 - Step 1: PC to MAR
 - Step 2: Instruction to MDR
 - Step 3: MDR to IR
 - Step 4: PC + 1 to PC
 - Step 5: Check Instruction
 - Step 6: IR[Address Part] to MAR
 - Step 7: Data to MDR
 - Step 8: MDR to ACC

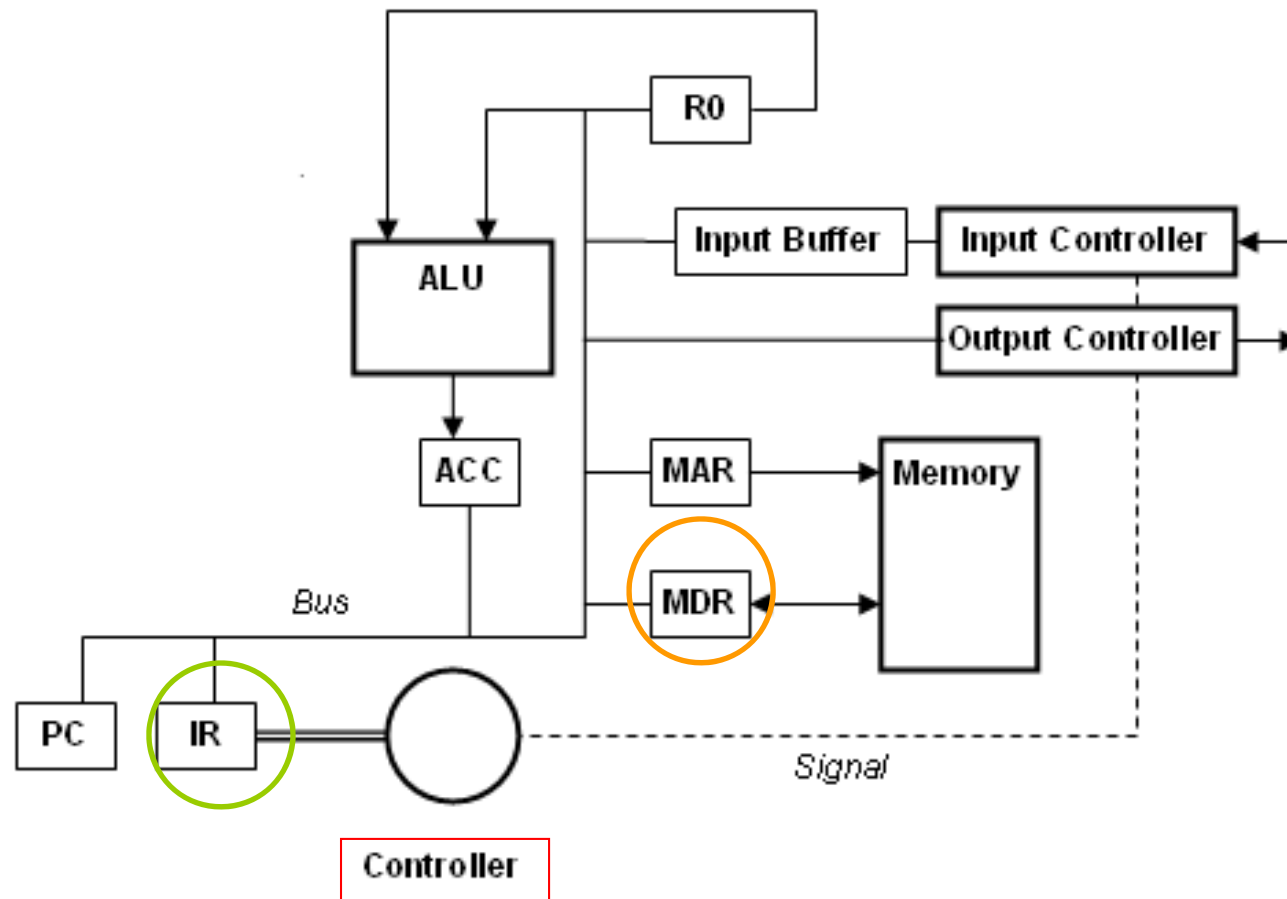
Step 1: PC to MAR



Step 2: Instruction to MDR

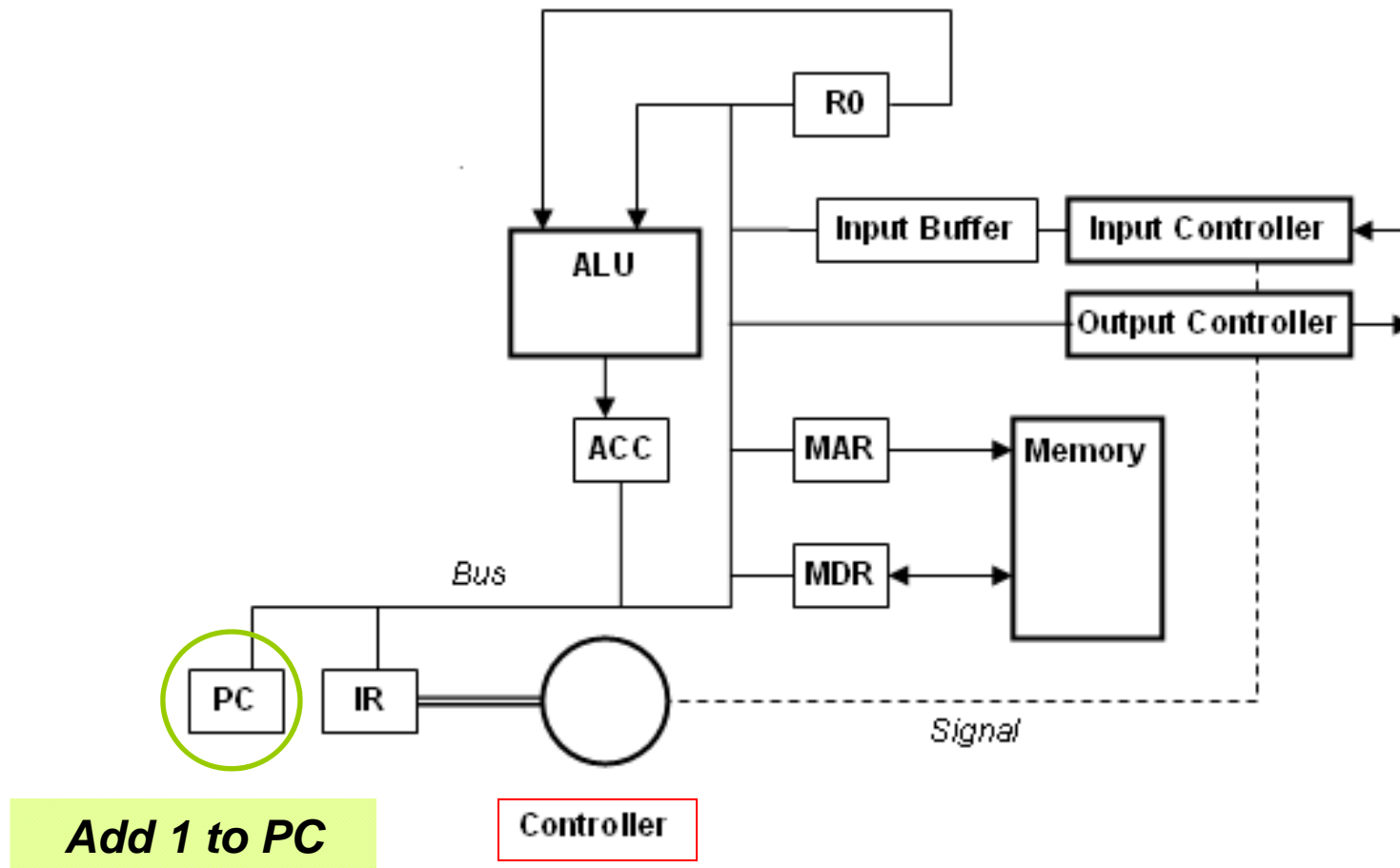


Step 3: MDR to IR

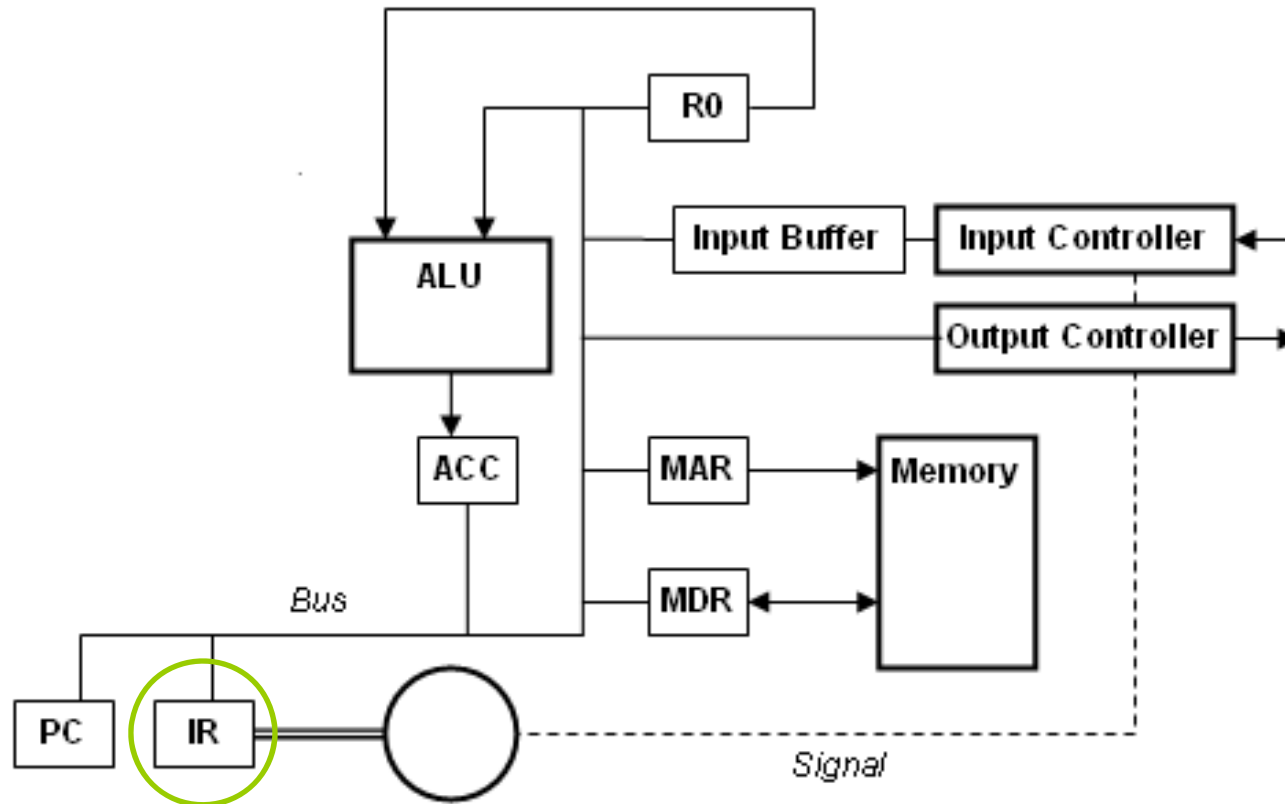


Instruction moved to IR

Step 4: PC Add 1



Step 5: Check Instruction

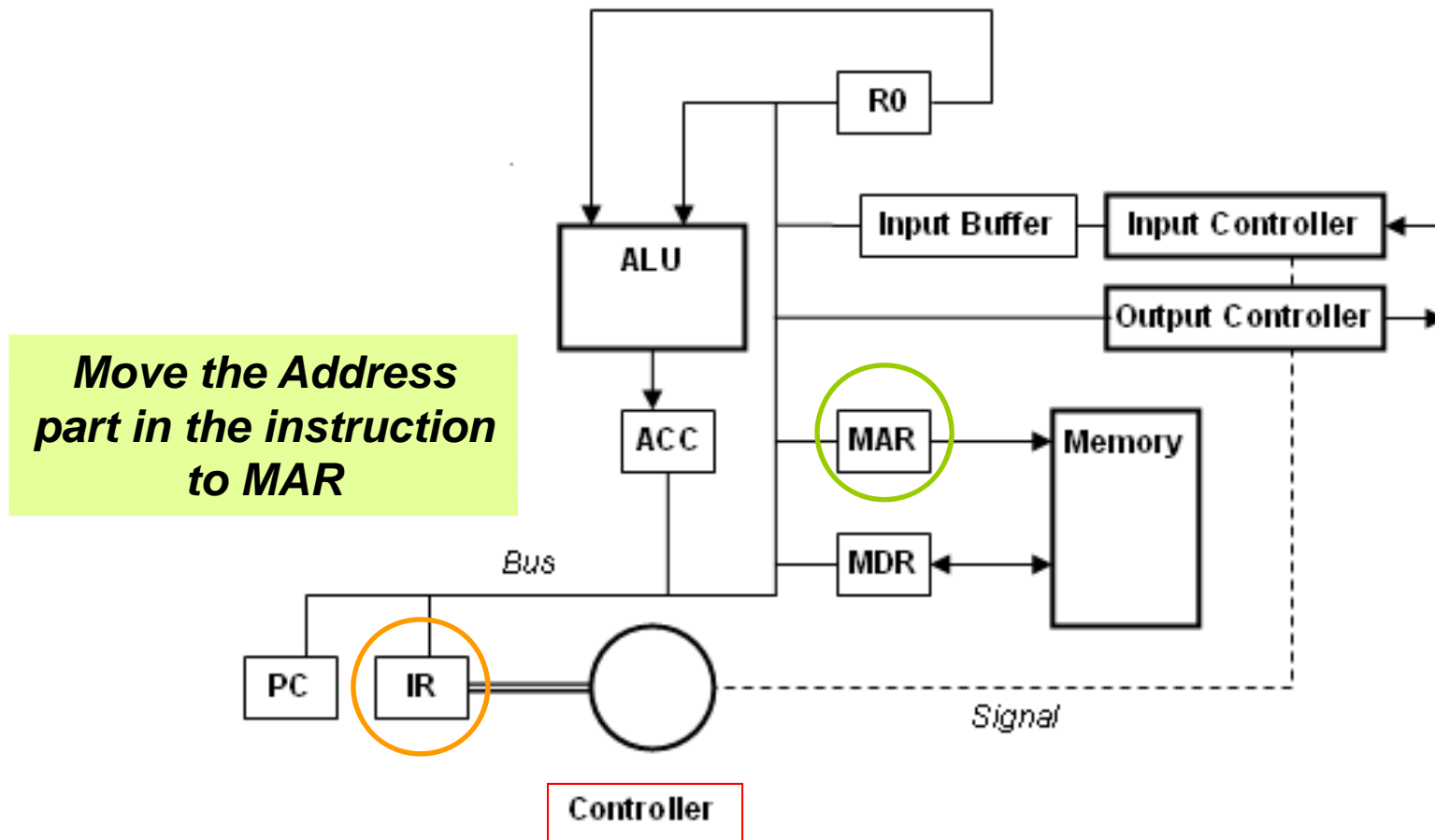


[LOAD Address]

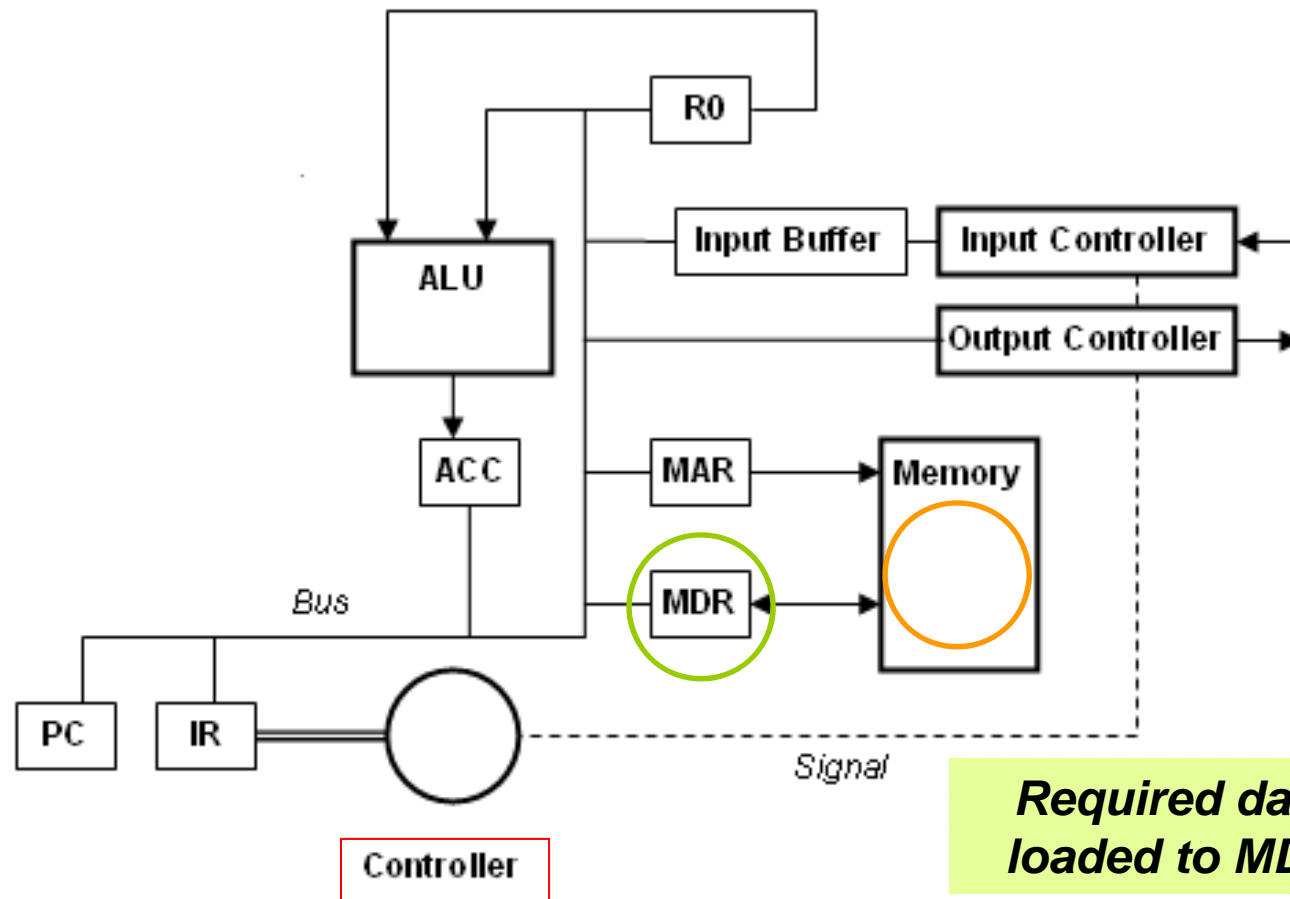
Controller

***IR is checked by Controller
and understood to be LOAD***

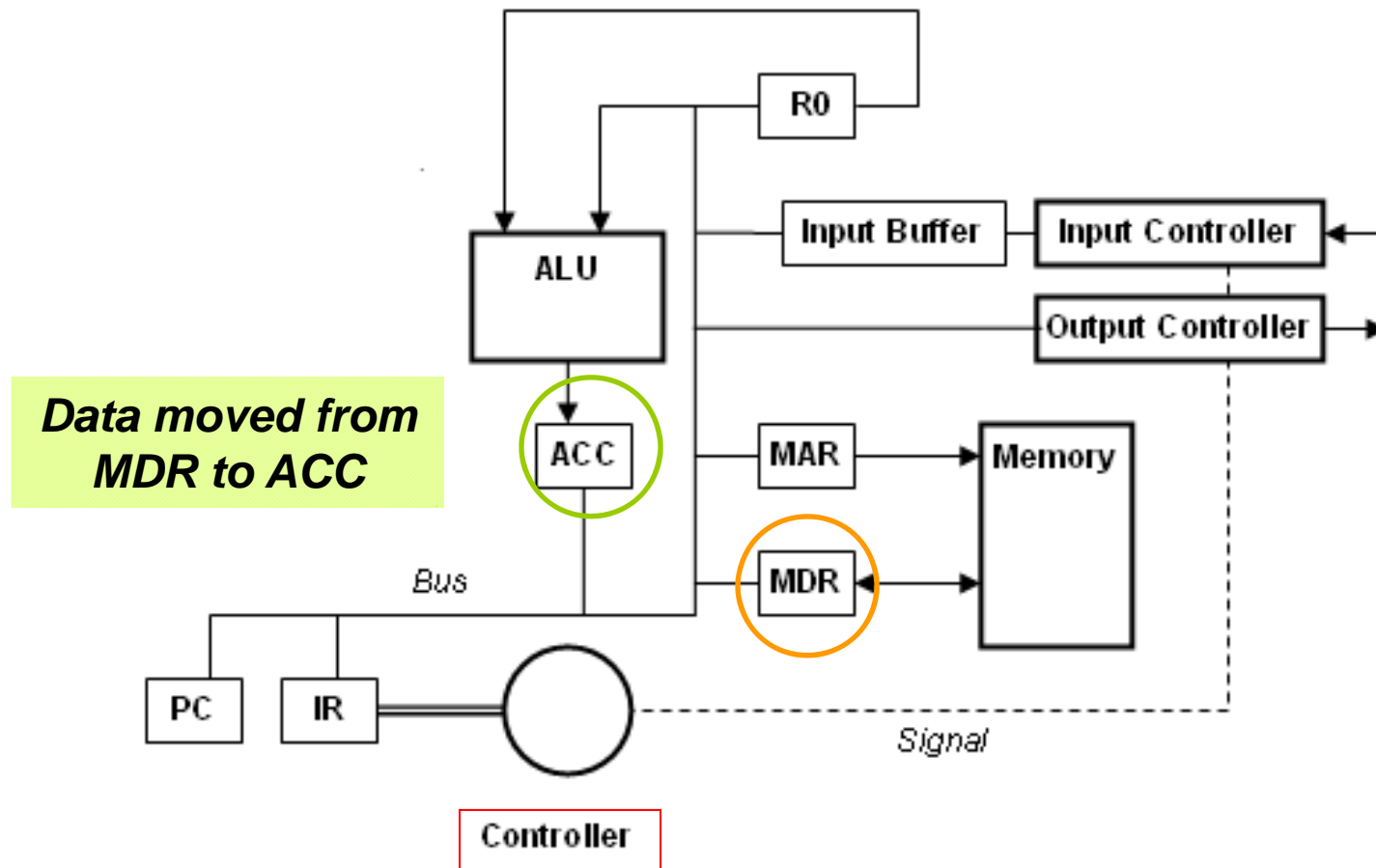
Step 6: IR[Address Part] to MAR



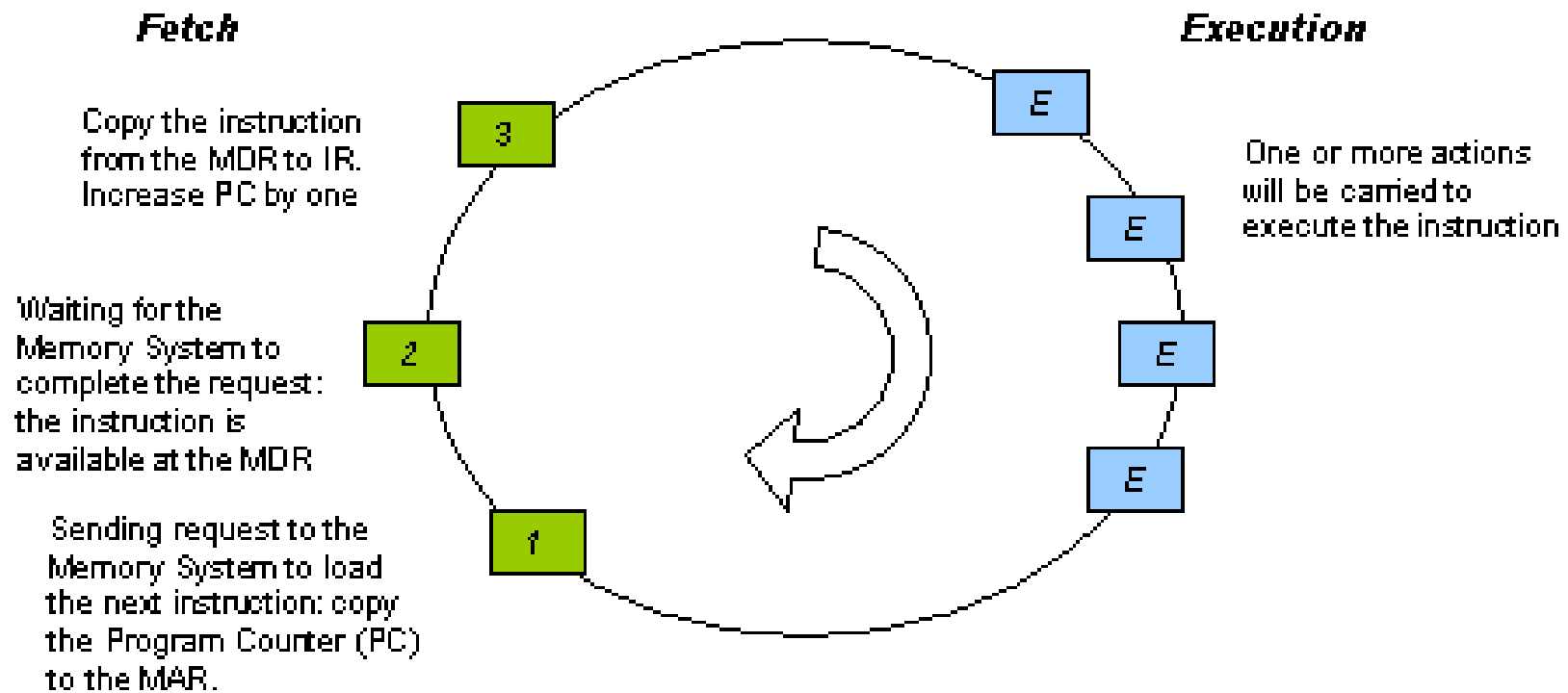
Step 7: Data to MDR



Step 8: MDR to ACC



Fetch and Execution Cycle



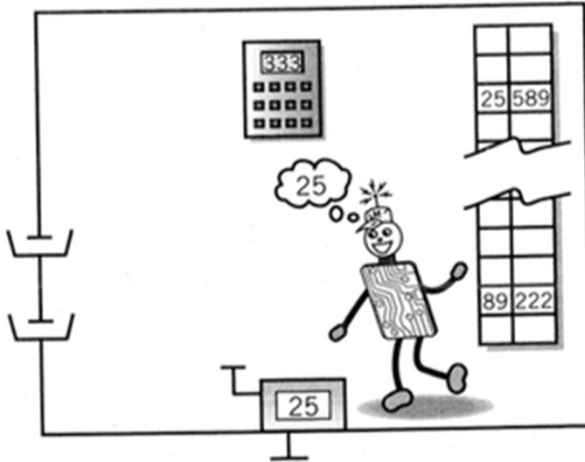
Fetch and Execution Cycle



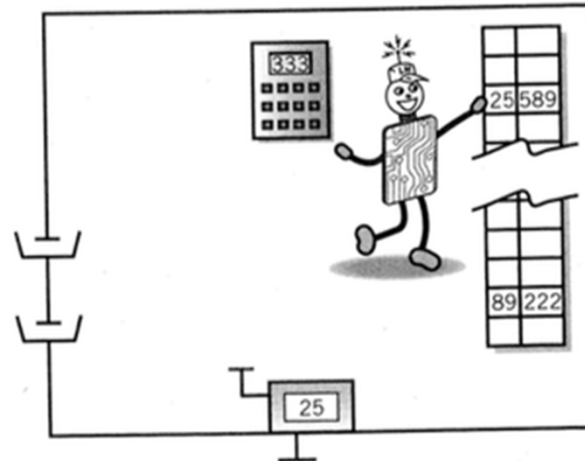
- Each step in the cycle
 - Moving data between memory and the CPU
 - MAR and MDR as the interface
 - Moving data between the CPU components



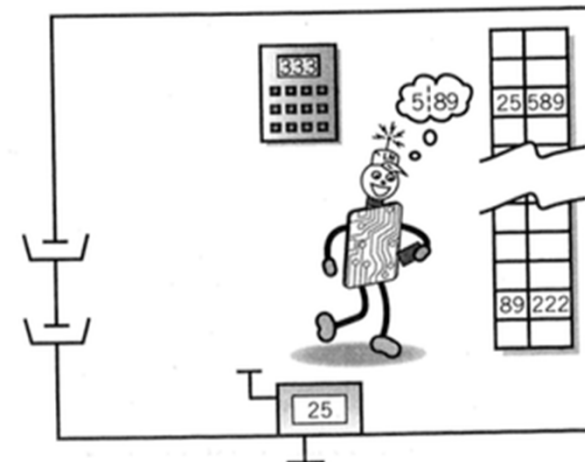
LMC Fetch



(1) The Little Man reads the address from the location counter



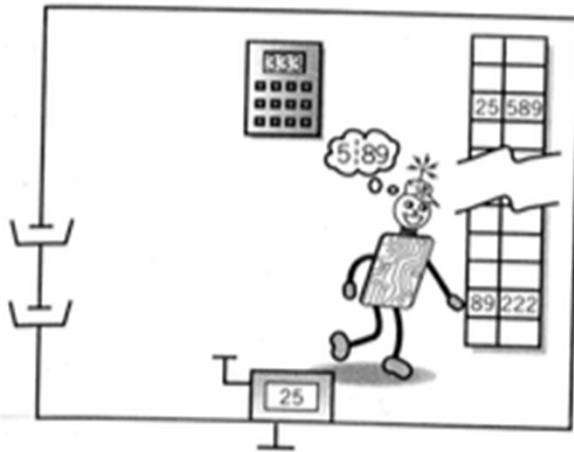
(2) . . . walks over to the mailbox that corresponds to the location counter



(3) . . . and reads the number on the slip of paper. (He then puts the slip of paper back, in case he should need to read it again later.)

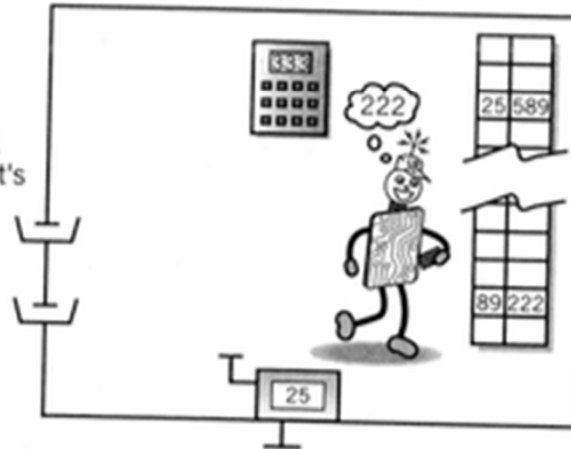
LDA 89

LMC Execute

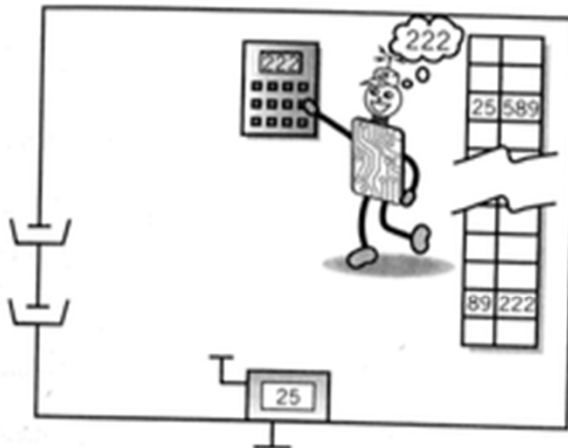


(1) The Little Man goes to the mailbox address specified in the instruction he previously fetched

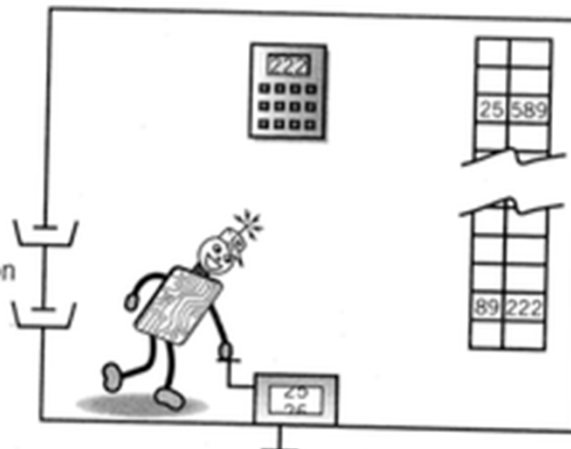
(2) . . . he reads the number in that mailbox (he remembers to replace it in the case it's needed again)



(3) . . . he walks over to the calculator and punches the number in



(4) . . . finally, he walks over to the location counter and clicks it, which gets him ready to fetch the next instruction



Load content of
mailbox #89
Into ACC



register transfer language

Register Transfer Language (RTL)



- A concise language for us to describe data movement in our programmable computer
 - Describe the micro-operations
 - Describe the fetch and execution cycle
 - Detailing the data movement routes and pattern
 - Improve the efficiency
 - Number of steps and time taken to execute an instruction becomes more visible

Register Transfer Language (RTL)



- Capitalized names: components
 - ACC, R0, ALU
- Square brackets []
 - A part of a register or the address of memory
 - IR[Address]
- The equals sign = indicates the value of a component
 - Memory[3] = 20
- The -> sign indicates data movement
 - MDR -> Memory[3]

Example 1: Fetch



PC -> MAR	;Send instruction address to MAR
M[MAR] -> MDR	;Read the current instruction
MDR -> IR	;Copy the instruction to the IR
PC + 1 -> PC	;Point to the next instruction

Example 2: ADD A, ACC



```
; adds the data of a memory address A and the data in ACC
PC -> MAR                ;Send instruction address to MAR
M[MAR] -> MDR             ;Read the current instruction
MDR -> IR                ;Copy the instruction to the IR
PC + 1 -> PC             ;Point to the next instruction
IR[address_field] -> MAR ;Send the operand address A to MAR
M[MAR] -> MDR             ;Read the operand from memory
MDR + ACC -> ACC         ;Perform the addition and put it to ACC
```




Example 2: ADD A, R0

Consider A is 4

```
; adds the data of a memory address A and the data in register R0
PC -> MAR                ;Send instruction address to MAR
M[MAR] -> MDR             ;Read the current instruction
MDR -> IR                 ;Copy the instruction to the IR
PC + 1 -> PC              ;Point to the next instruction
IR[address_field] -> MAR  ;Send the operand address A to MAR
M[4] -> MDR               ;Read the operand from memory addr 4
MDR + R0 -> R0            ;Perform the addition and put it to R0
```

Example



Example: RTL

Question: Write down the RTL for an instruction ADD R0, 4. The instruction adds the content of Memory Address 4 to R0.

PC -> MAR Send instruction address to MAR

M[MAR] -> MDR Read the current instruction

MDR -> IR Copy the instruction to the IR

PC + 1 -> PC Point to the next instruction

IR[address_field] -> MAR Send the operand address A to MAR

M[4] -> MDR Read the operand from memory

MDR + R0 -> R0 Perform the addition and put it to R0

Example



- Assuming that the address of the instruction **ADD R0, 4** is 2000, the data in address 4 is 10, the data in R0 is 30. The content of the major registers *after* the execution of the instruction are shown below

<i>Register</i>	<i>Data/Content</i>
PC	2001
IR	Holding the instruction code of ADD 4, R0
R0	40
MAR	4
MDR	10

Example



Exercise: LMC program

Question: Given the following LMC program.

```
00      IN           ; 901  Input the data
01      STO 10       ; 310  Store to location 10
02      IN           ; 901  Input the data
03      STO 11       ; 311  Store to location 11
04      ADD 10        ; 110  Add with location 10
05      OUT          ; 902  Output
06      BR 00         ; 600  Branch to 00
07      COB          ; 000  End of Program
10      DATA
11      DATA
```

Example



Exercise: LMC program

Question: With the same LMC program, write down the steps in executing the instruction BR 00 at address 06 with RTL.

Answer:

PC -> MAR

M[MAR] -> MDR

MDR -> IR

IR[ADDRESS] -> PC

Example



Exercise: LMC program

Question: The LMC is executing the instruction BR 00 at address 06. Write down the content of MAR, MDR, IR, and PC after the execution.

Answer:

We should look at the RTL and know what values have been loaded to these registers.

MAR = 06

MDR = 600

IR = 600

PC = 00

Benefits of RTL

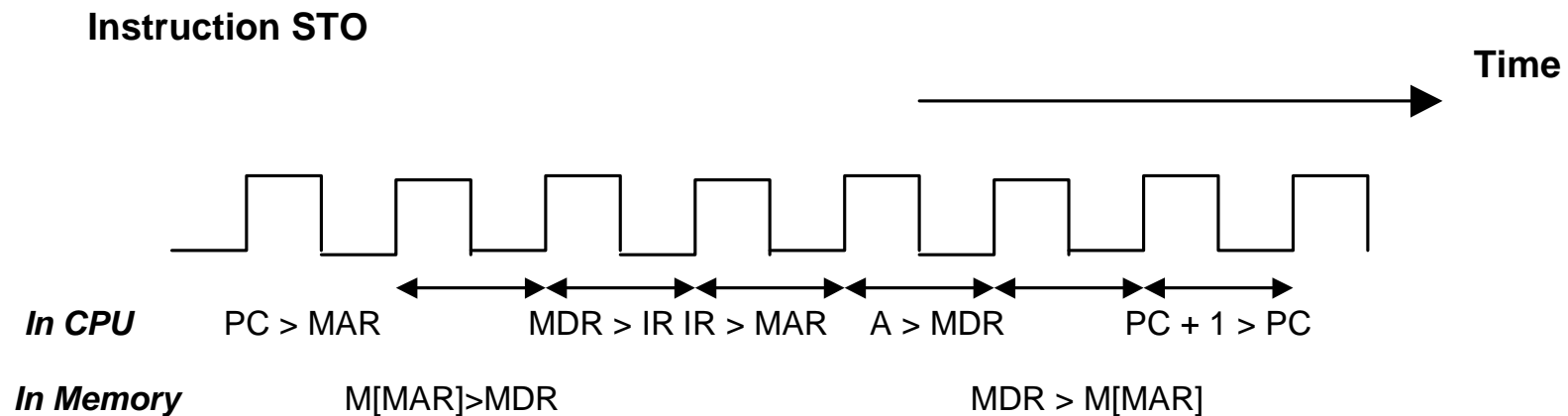


- A clock signal controls the execution of RTL steps
 - One clock cycle per step
 - Work out the number of clock cycle per instruction
 - Work out if RTL steps carried out in parallel
- RTL language can support micro-program
 - CPU Control Unit operates according to RTL steps

Number of Clock Cycles



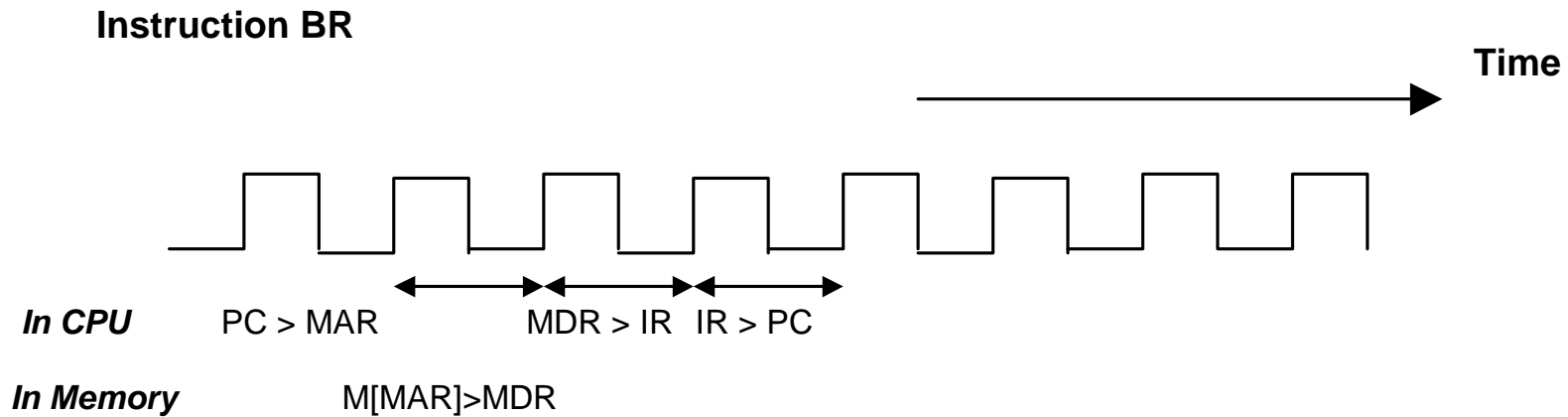
STO may require 7 clock cycles



Number of Clock Cycles



BR may require 4 clock cycles





Performance analysis of LMC



Number of execution cycles

- Fetch part (4 execution cycles and 1 memory operation)

PC \rightarrow MAR	;Send instruction address to MAR
M[MAR] \rightarrow MDR	; Read the current instruction
MDR \rightarrow IR	;Copy the instruction to the IR
PC + 1 \rightarrow PC	;Point to the next instruction

- The theoretical figures based on the assumptions below:
 - Each data movement between registers takes one cycle
 - Each memory system operation takes one cycle
 - Each input and output operation takes one cycle

Example



Exercise: Execution Speed of LMC Programs

Question: You have written a LMC program. In one execution of the program, you counted the number of instructions executed: there are 300 LDA or STO instructions, 120 ADD or SUB instructions, 20 BR, BRZ, or BRP instructions, and 5 IN or OUT instructions. If the CPU clock rate is 100 MHz, calculate the time taken to execute the program

Answer:

Total number of cycles is calculated from the summation of number of cycles for each instruction.

$$= 300 \times 7 \text{ cycles} + 120 \times 7 \text{ cycles} + 20 \times 4 \text{ cycles} + 5 \times 5 \text{ cycles}$$

$$= 3045 \text{ cycles}$$

The clock rate is 100 MHz, which means 100 M cycles per second

The time take to execute the program is $3045 / 100 \text{ M} = 0.00003045$ seconds

LMC Performance Analysis



- Number of execution cycles of each LMC instruction
 - Calculate speed of LMC program execution
- Running time of a program
 - total number of instructions executed
 - composition of the instructions.

Example



Exercise: Execution Speed of LMC Programs

Question: Both Anders and Betsy have written a LMC program to find out the square of a number.

Anders' program execution has involved 40 instructions, including 15 ADD/SUB, 20 LDA/STO, 3 BR/BRZ/BRP, and 2 IN/OUT.

Betsy program execution has involved 42 instructions, including 14 ADD/SUB, 18 LDA/STO, 8 BR/BRZ/BRP, and 2 IN/OUT.

Which program is better in terms of execution speed?

Answer:

Total number of cycles is calculated from the summation of number of cycles for each instruction.

Anders' program

$= (15 + 20) \times 7 \text{ cycles} + 3 \times 4 \text{ cycles} + 2 \times 5 \text{ cycles}$

$= 267 \text{ cycles}$

Betsy's program

$= (14 + 18) \times 7 \text{ cycles} + 8 \times 4 \text{ cycles} + 2 \times 5 \text{ cycles}$

$= 266 \text{ cycles}$

Betsy's program ran faster, even if the total number of instructions is more.