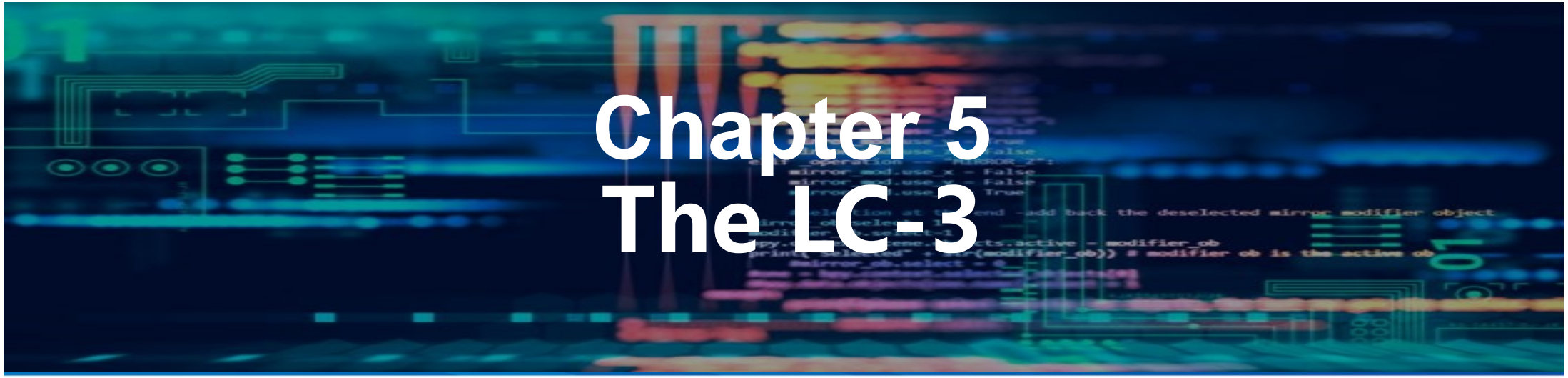
中国科学技术大学
University of Science and Technology of China

计算系统概论A
Introduction to Computing Systems
(CS1002A.03)

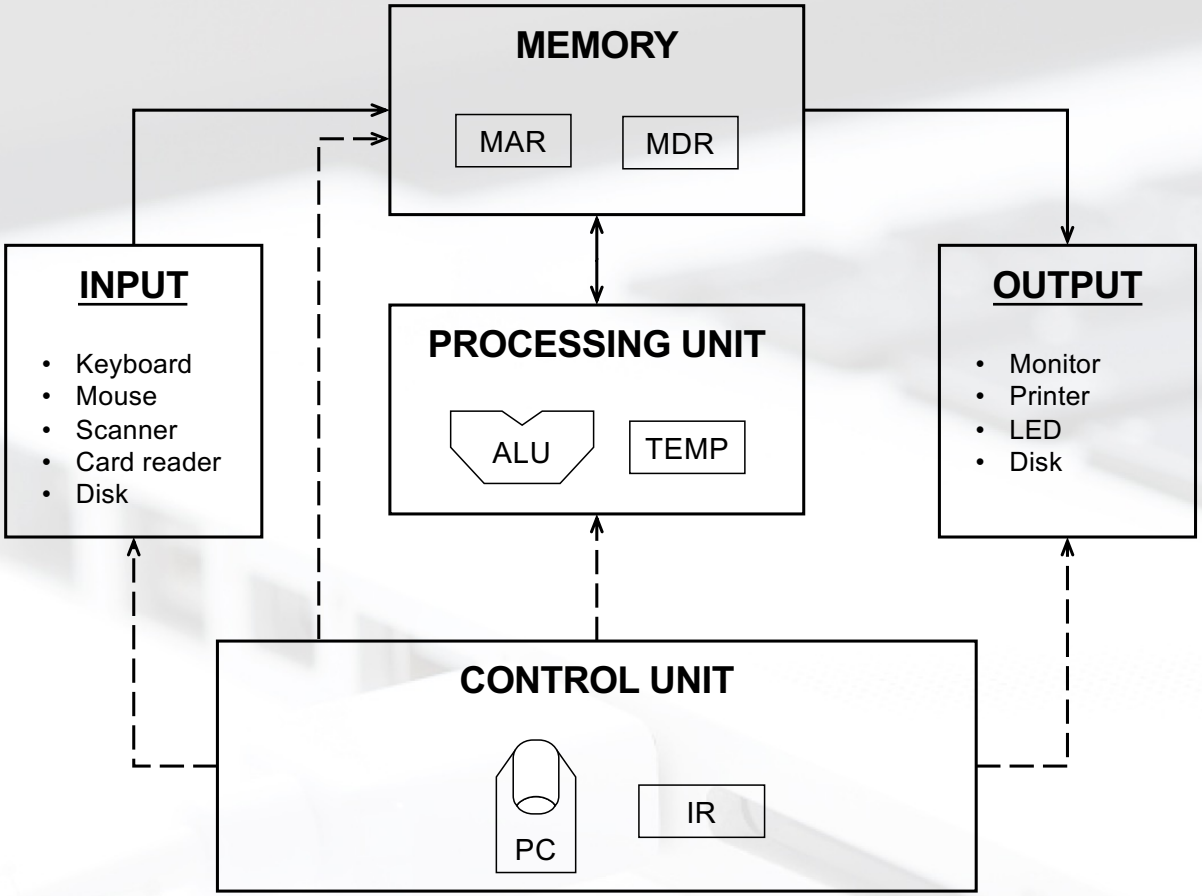


Chapter 5
The LC-3

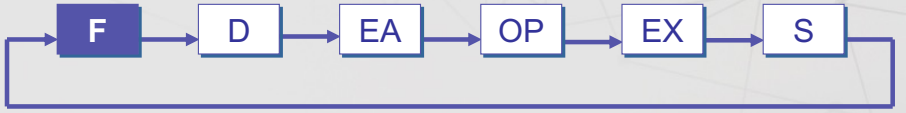
陈俊仕
cjuns@ustc.edu.cn
2023 Fall

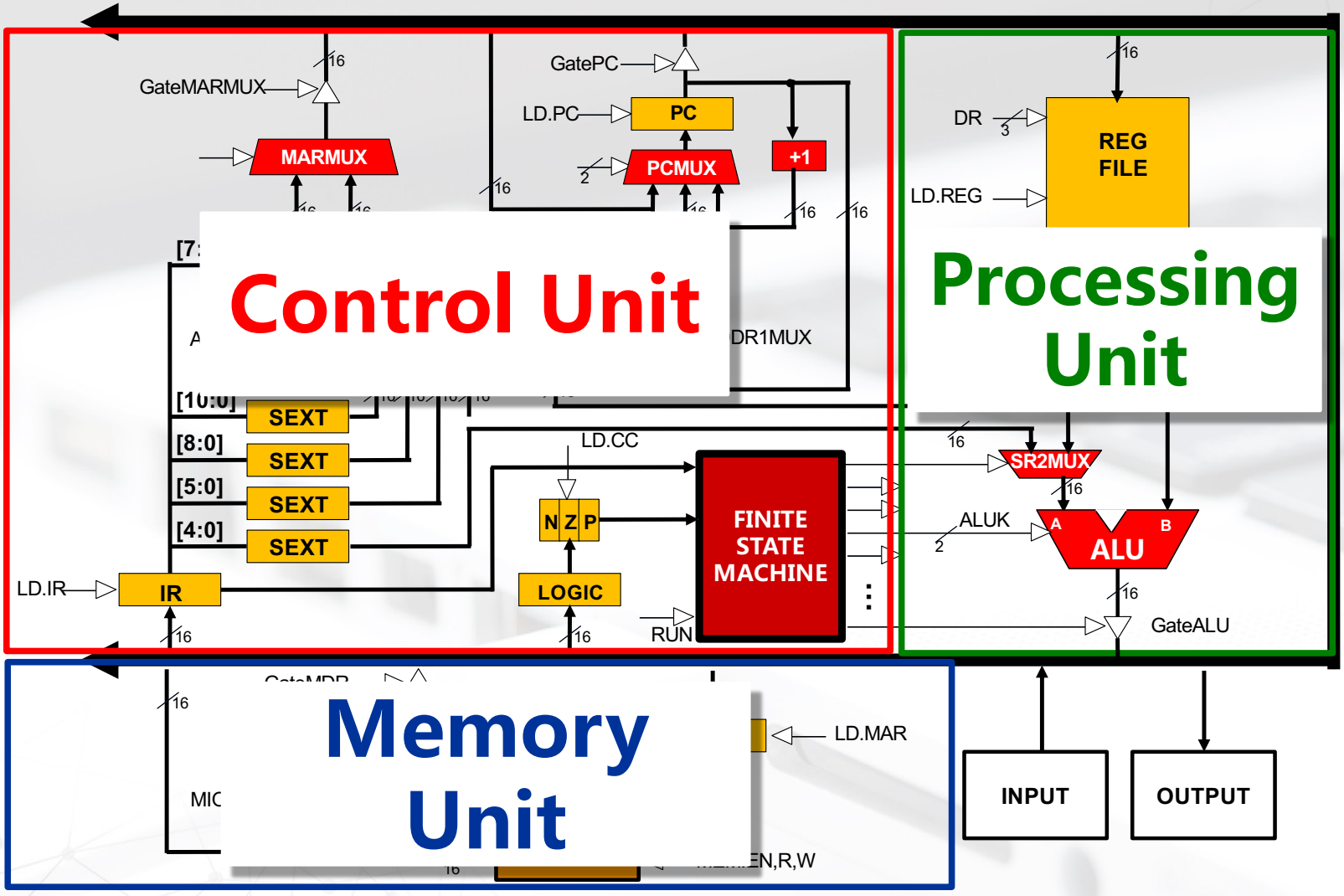
计算机科学与技术学院
School of Computer Science and Technology

Previously: von Neumann Model

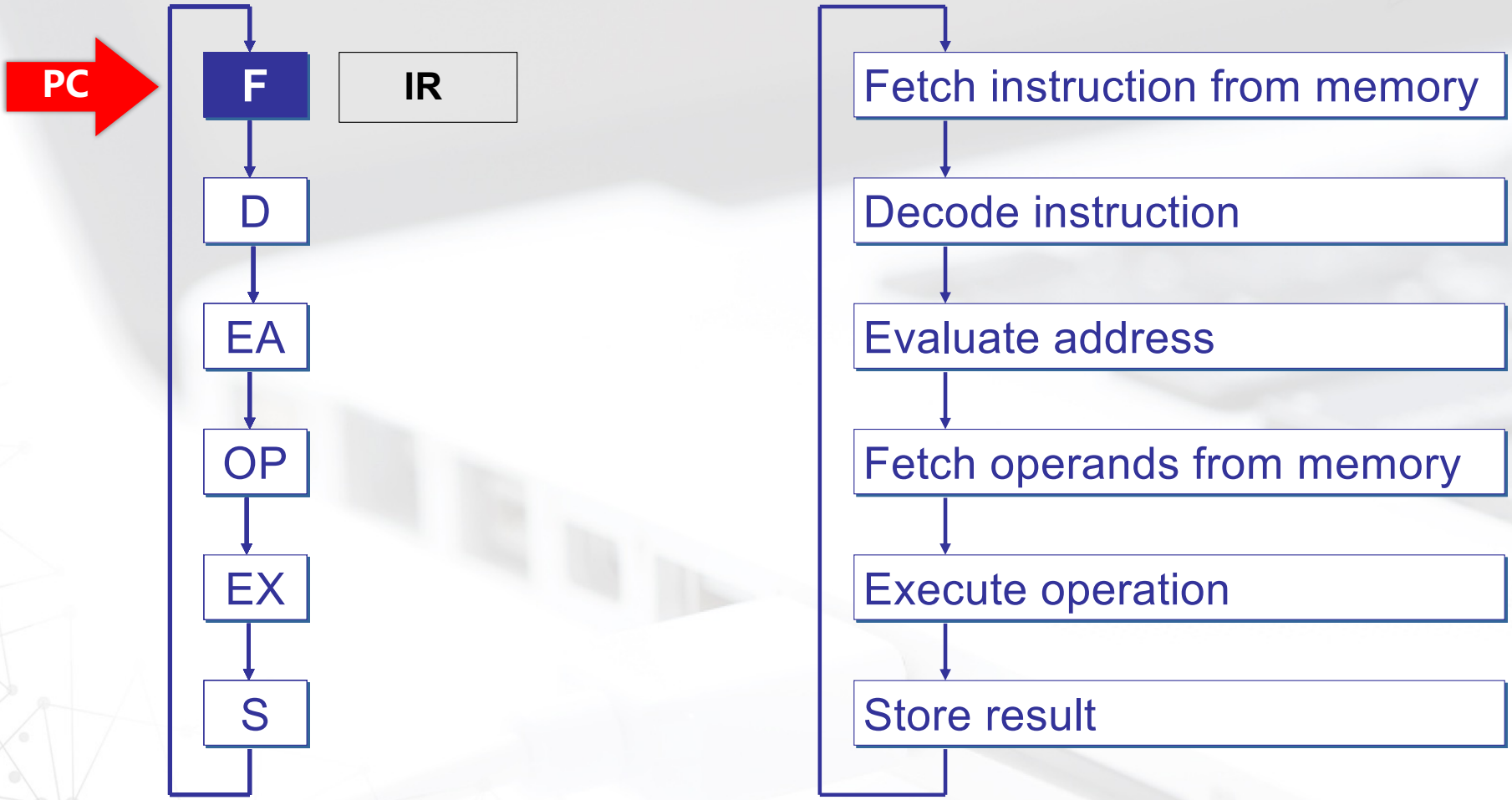


Previously: LC-3 Data Path





Review: Instruction Processing(State Transition)



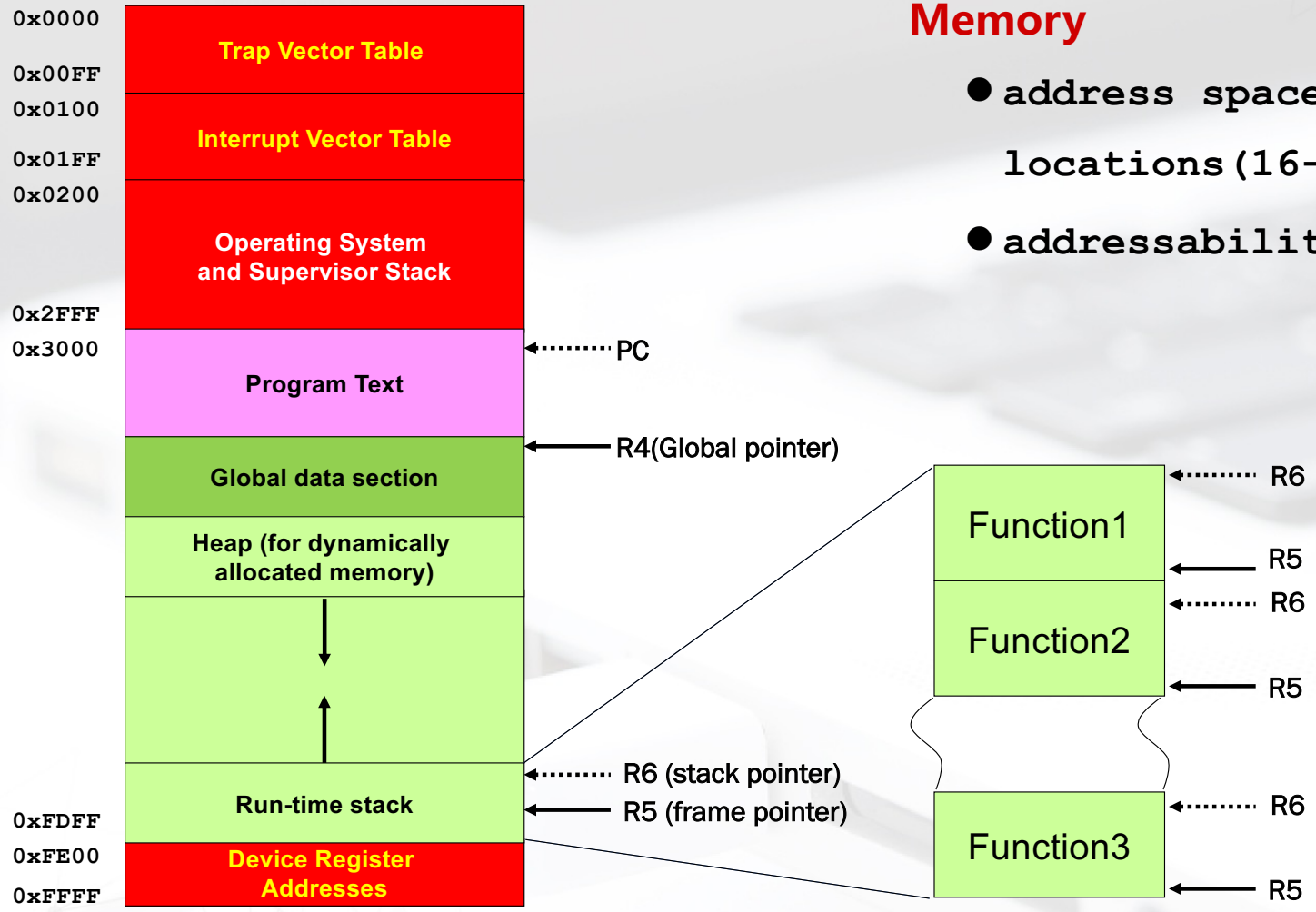
Instruction Set Architecture

ISA = All of the *programmer-visible* components and operations of the computer

- **`memory organization`**
  - address space -- how may locations can be addressed?
  - addressability -- how many bits per location?
- **`register set`**
  - how many? what size? how are they used?
- **`instruction set`**
  - opcodes
  - data types
  - addressing modes

ISA provides all information needed for someone that wants to write a program in machine language (or translate from a high-level language to machine language).

LC-3 Overview: Memory



Memory

- address space: 2^{16} locations(16-bit addresses)
- addressability: 16 bits

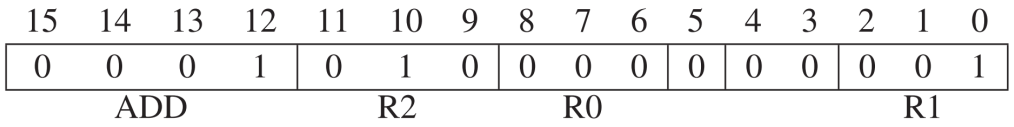
LC-3 Overview: Registers

Registers

- **temporary storage, accessed in a single machine cycle**
  - —accessing memory generally takes longer than a single cycle
- **eight general-purpose registers: R0 - R7**
  - —**each 16 bits wide**
  - —how many bits to uniquely identify a register?
- **other registers**
  - —not directly addressable, but used by (and affected by) instructions
  - —**PC (program counter), condition codes**

Register 0	(R0)	0000000000000001
Register 1	(R1)	0000000000000011
Register 2	(R2)	0000000000000101
Register 3	(R3)	0000000000000111
Register 4	(R4)	1111111111111110
Register 5	(R5)	1111111111111100
Register 6	(R6)	1111111111111010
Register 7	(R7)	1111111111111000

Register 0	(R0)	0000000000000001
Register 1	(R1)	0000000000000011
Register 2	(R2)	0000000000000100
Register 3	(R3)	0000000000000111
Register 4	(R4)	1111111111111110
Register 5	(R5)	1111111111111100
Register 6	(R6)	1111111111111010
Register 7	(R7)	1111111111111000



LC-3 Overview: Instruction Set

Opcodes

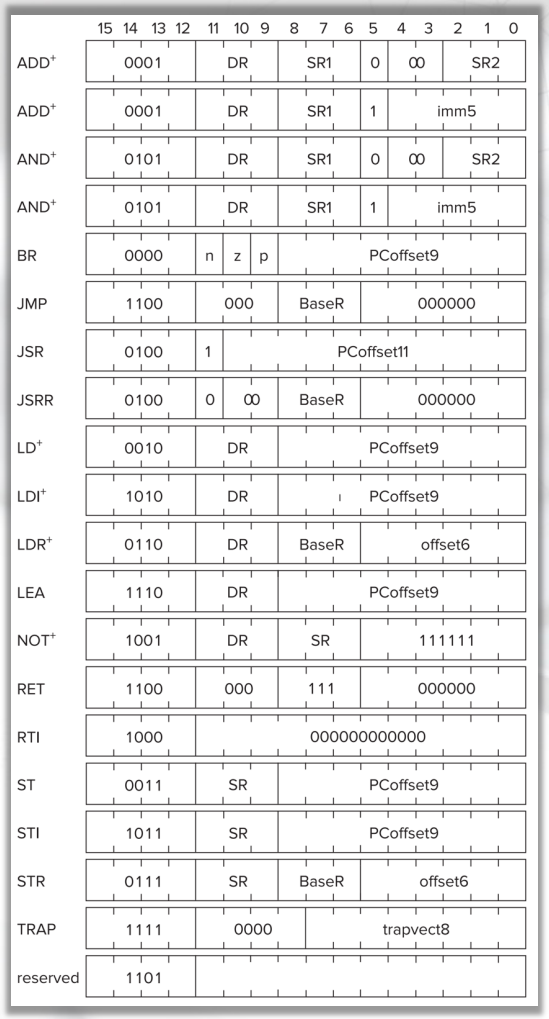
- 15 opcodes
- *Operate* instructions: ADD, AND, NOT
- *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
- *Control* instructions: BR, JSR/JSRR, JMP(RET), RTI, TRAP
- some opcodes(ADD, AND, NOT; LD, LDI, LDR, LEA) set/clear *condition codes*, based on result:
  - N = negative, Z = zero, P = positive (> 0)

Data Types

- 16-bit 2's complement integer

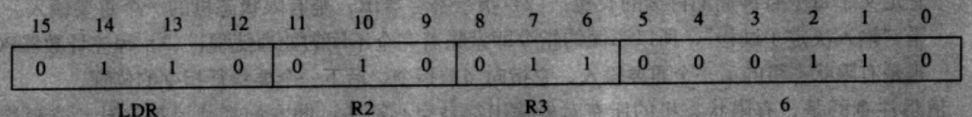
Addressing Modes

- How is the location of an operand specified?
- non-memory addresses: *immediate, register*
- memory addresses: *PC-relative, indirect, base+offset*



例4-2 LDR指令。

LDR指令需要两个操作数。LD代表“load”，按照计算机语言（computerese），其意思是“从内存某个地方读取其中的内容，并将其内容存入某寄存器中”。两个操作数分别对应内存地址和目的寄存器。LDR中的“R”代表计算内存读取地址的机制（或称为寻址模式），R代表的是“base+offset”模式。参考LC-3的LDR指令格式：



LDR指令的4-bit操作码是“0110”。

bit[11:9] = 010，代表内存读入数据所存放的目的寄存器R2。

bit[8:0] = 011，代表偏移寄存器R3。

bit[5:0] = 000110，代表基址值6。

bit[8:6]和bit[5:0]的内容与内存地址计算有关。LC-3只支持“基址加偏移”（BASE+offset）一种寻址模式，其计算方法是：将bit[8:6]指定的寄存器的内容和bit[5:0]中的补码数值相加，求得最终地址。因而，本例的解读是“将R3的内容和立即数6相加，求得内存地址，然后将该地址指向的内存单元内容读入，并存放在R2寄存器中”。

5.3.3 Base+offset Mode

LDR (opcode = 0110) and **STR** (opcode = 0111) specify the *Base+offset* addressing mode. The Base+offset mode is so named because the address of the operand is obtained by adding a sign-extended six-bit offset to a base register. The six-bit offset is obtained from the instruction, bits [5:0]. The base register is specified by bits [8:6] of the instruction.

If R2 contains the 16-bit quantity x2345, the following instruction loads R1 with the contents of x2362.

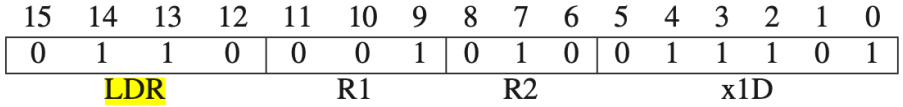


Figure 5.9 shows the relevant parts of the data path required to execute this instruction. First the contents of R2 (x2345) is added to the sign-extended value

LC-3 ISA Overview

运算指令

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	0	1	DR			SR1			0	0	0	SR2		
ADD	0	0	0	1	DR			SR1			1	Imm5				
AND	0	1	0	1	DR			SR1			0	0	0	SR2		
AND	0	1	0	1	DR			SR1			1	Imm5				
NOT	1	0	0	1	DR			SR1			1	1	1	1	1	1
Reserved	1	1	0	1												

控制指令

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	n	z	p	PCoffset9								
JSR	0	1	0	0	1	PCoffset11										
JSRR	0	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMP	1	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RET	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
TRAP	1	1	1	1	0	0	0	0	TrapVector8							

移动数据指令

取数指令

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LD	0	0	1	0	DR			PCoffset9								
LDR	0	1	1	0	DR			BaseR			PCoffset6					
LDI	1	0	1	0	DR			PCoffset9								
LEA	1	1	1	0	DR			PCoffset9								

存数指令

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ST	0	0	1	1	SR			PCoffset9								
STR	0	1	1	1	SR			BaseR			PCoffset6					
STI	1	0	1	1	SR			PCoffset9								

Outline

1 LC-3 ISA Overview

2 Operate Instructions and Data Path

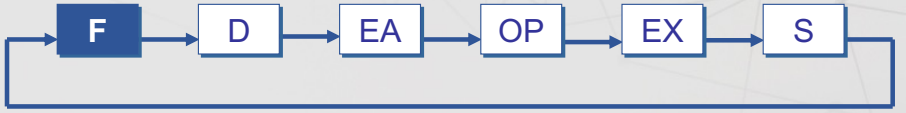
3 Data Movement Instructions and Data Path

4 Control Instructions and Data Path

5 Another Example: Counting Occurrences of a Computer

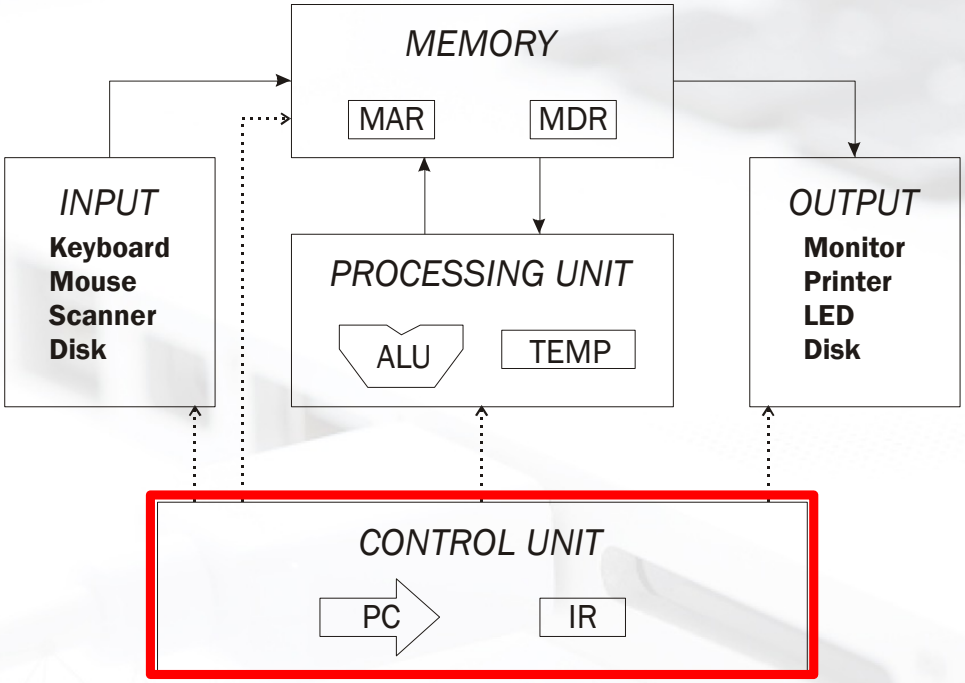
6 The Data Path Revisited

Today



We are going to learn how to:

- Used to alter the sequence of instructions (by changing the Program Counter)



Control Instructions

Conditional Branch

- **branch is *taken* if a specified condition is true**
  - — signed offset is added to PC to yield new PC
- **else, the branch is *not taken***
  - — PC is not changed, points to the next sequential instruction

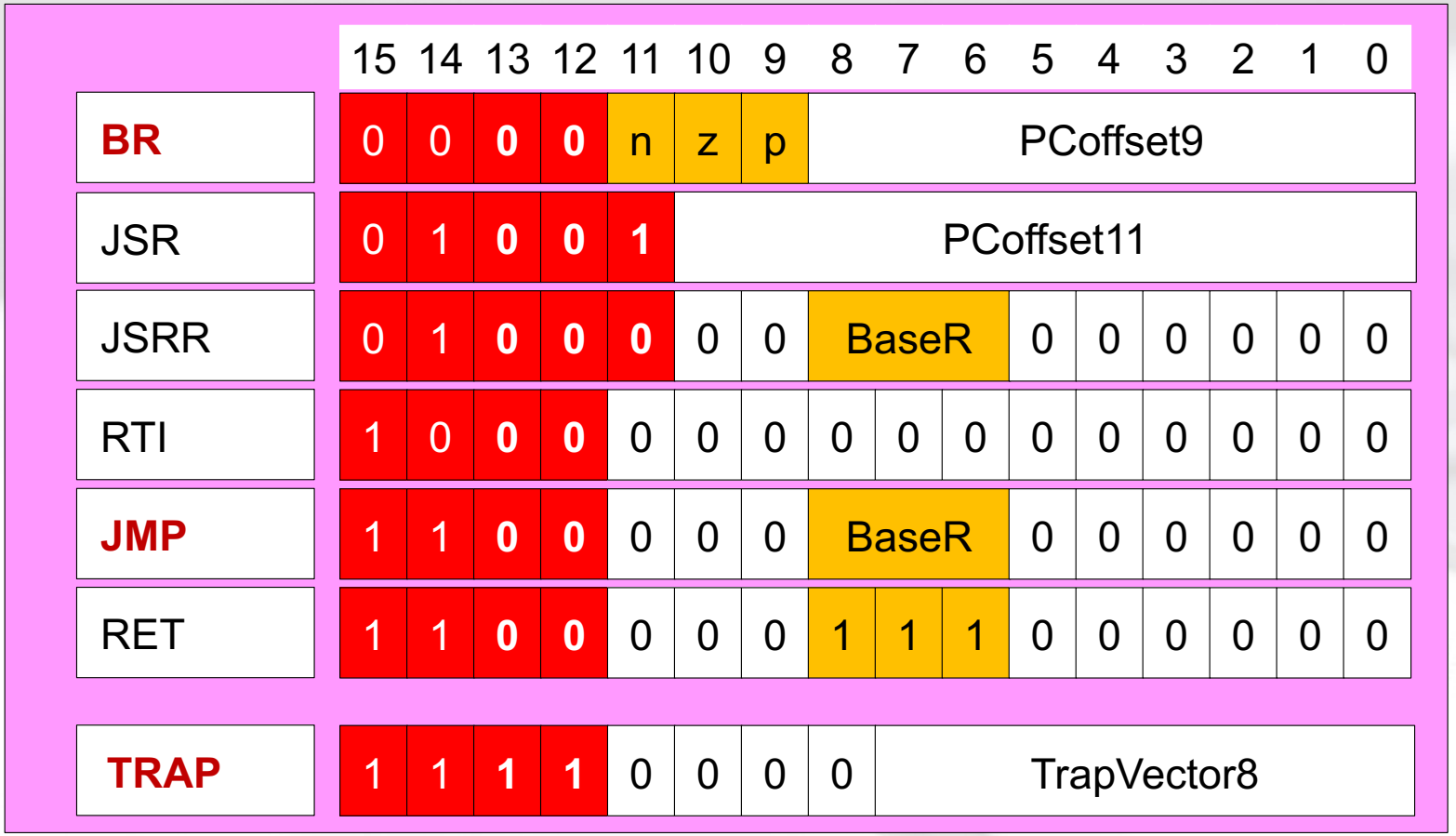
Unconditional Branch (or Jump)

- **always changes the PC**

TRAP

- **changes PC to the address of an OS "service routine"**
- **routine will return control to the next instruction (after TRAP)**

Control Instructions



Condition Codes

LC-3 has three condition code registers:

- N -- negative
- Z -- zero
- P -- positive (greater than zero)

Set by any instruction that writes a value to a register

(ADD, AND, NOT, LD, LDR, LDI, LEA)

Exactly one will be set at all times

- Based on the last instruction that altered a register

Conditional Branch Instruction

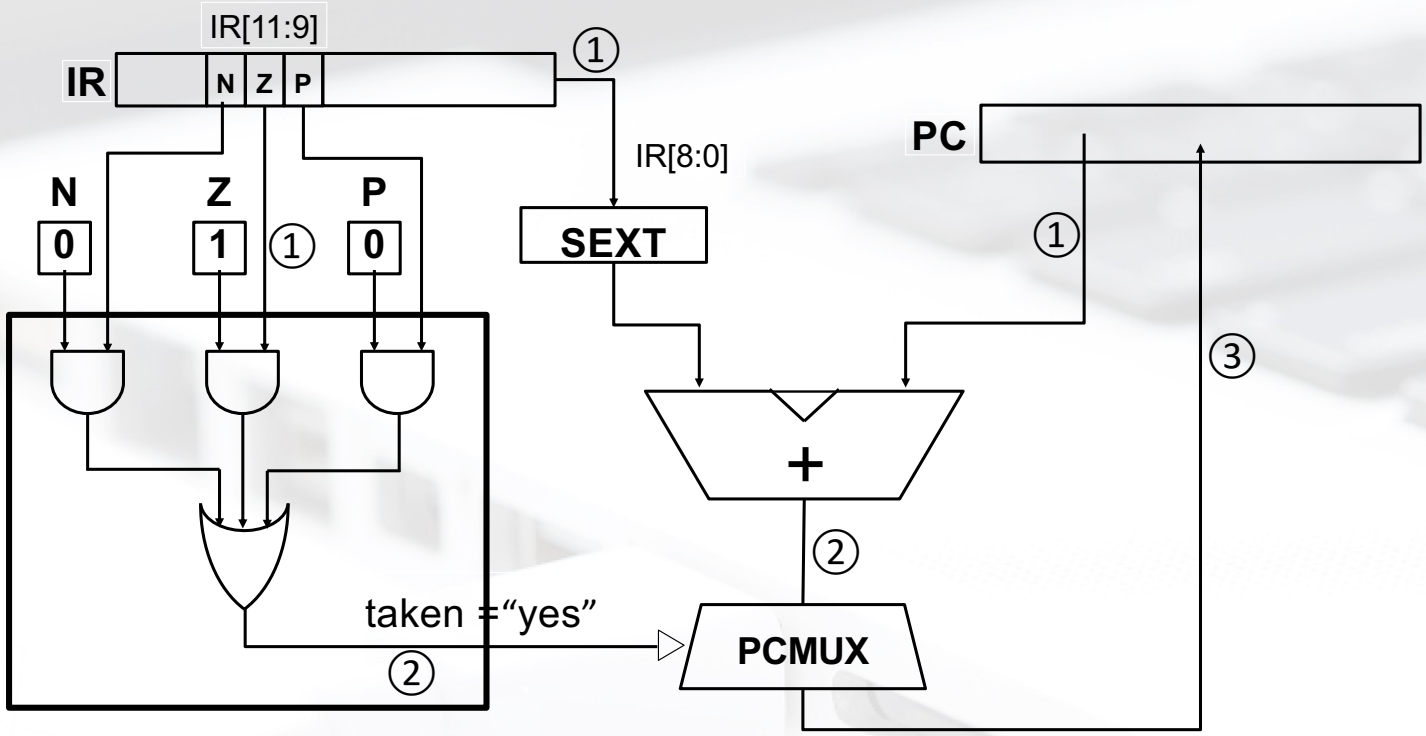
Branch specifies one or more condition codes.

- **If the specified bit is set, the branch is taken.**
  - PC-relative addressing:

    target address is made by adding signed offset (IR[8:0])

    to current PC.
  - Note: PC has already been incremented by FETCH stage.
  - Note: Target must be within 256 words of BR instruction.
- **If the branch is not taken, the next sequential instruction is executed.**

BR (PC-Relative)



What happens if bits [11:9] are all zero?
What happens if bits [11:9] are all one?

BR (PC-Relative)

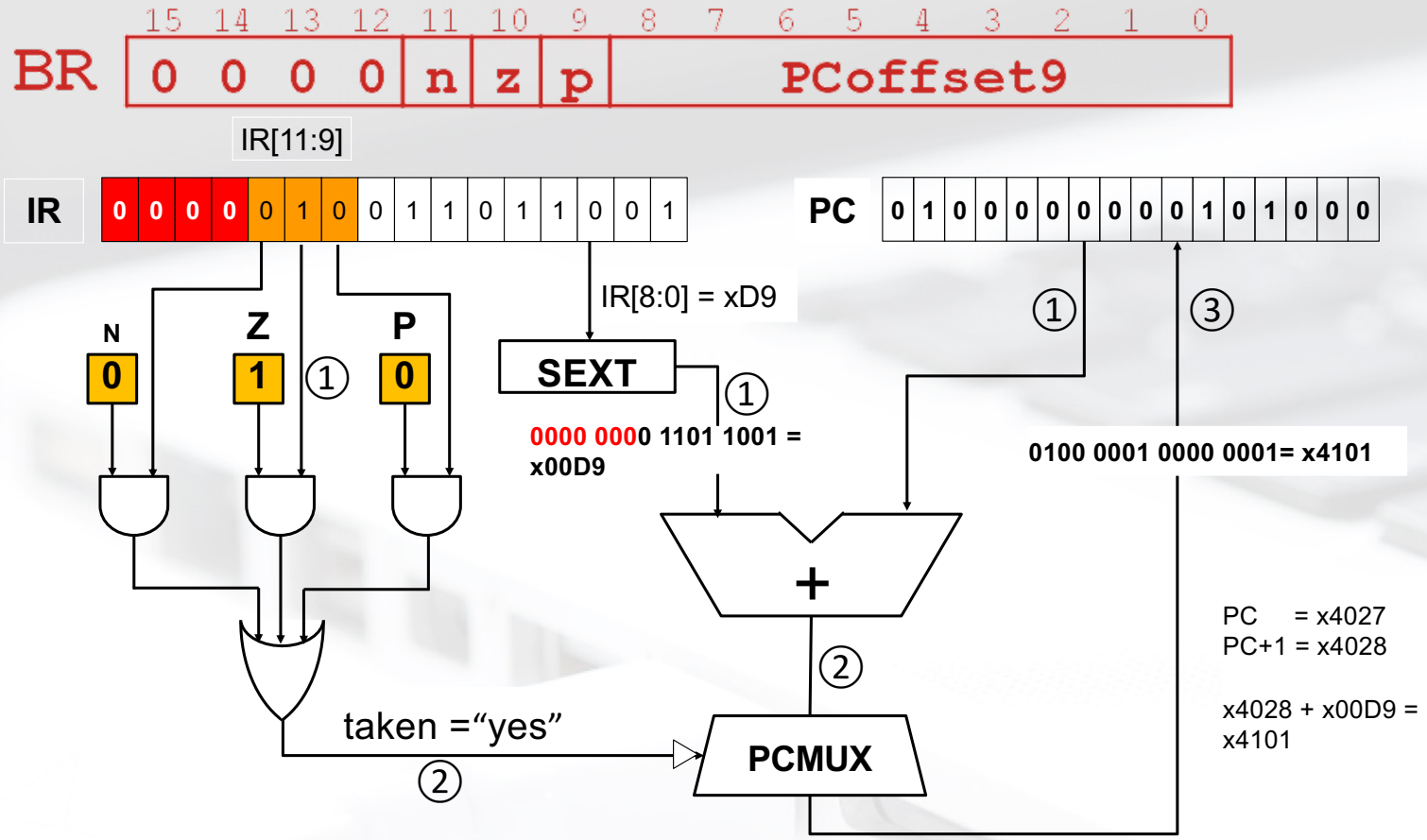
Check

- BR_{nzp} x4101 ; if (n=1 or z=1 or p=1)， JMP x4101
- BR_{n} x4101 ; if (n=1)
- BR_{z} x4101 ; if (z=1)
- BR_{p} x4101 ; if (p=1)
- BR_{nz} x4101 ; if (n=1 or z=1)
- BR_{np} x4101 ; if (n=1 or p=1)
- BR_{zp} x4101 ; if (z=1 or p=1)
- BR x4101 ; PC=PC+1

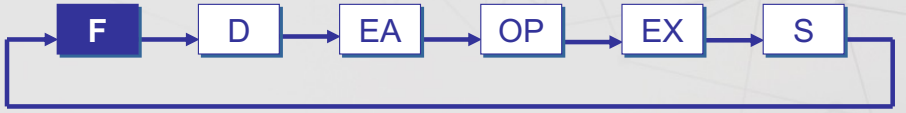
Set

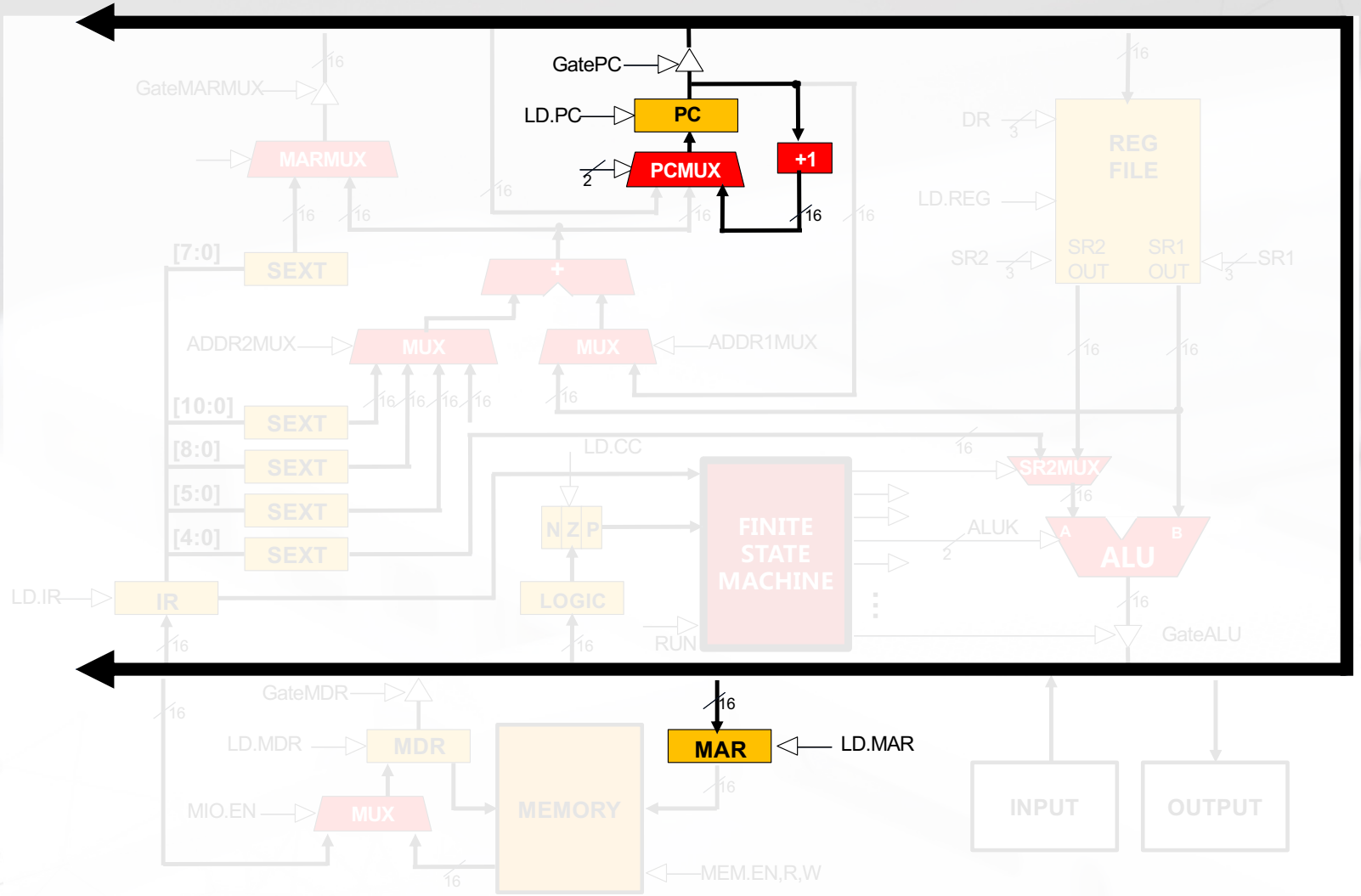
- If DR < 0, set N=1 and Z=0 and P=0
- If DR = 0, set N=0 and Z=1 and P=0
- If DR > 0, set N=0 and Z=0 and P=1

BR (PC-Relative): BR_z x4101

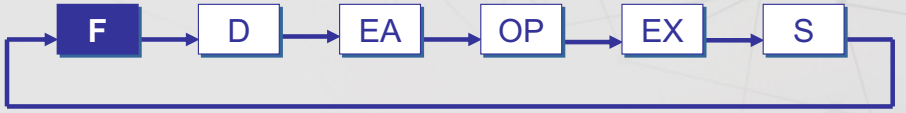


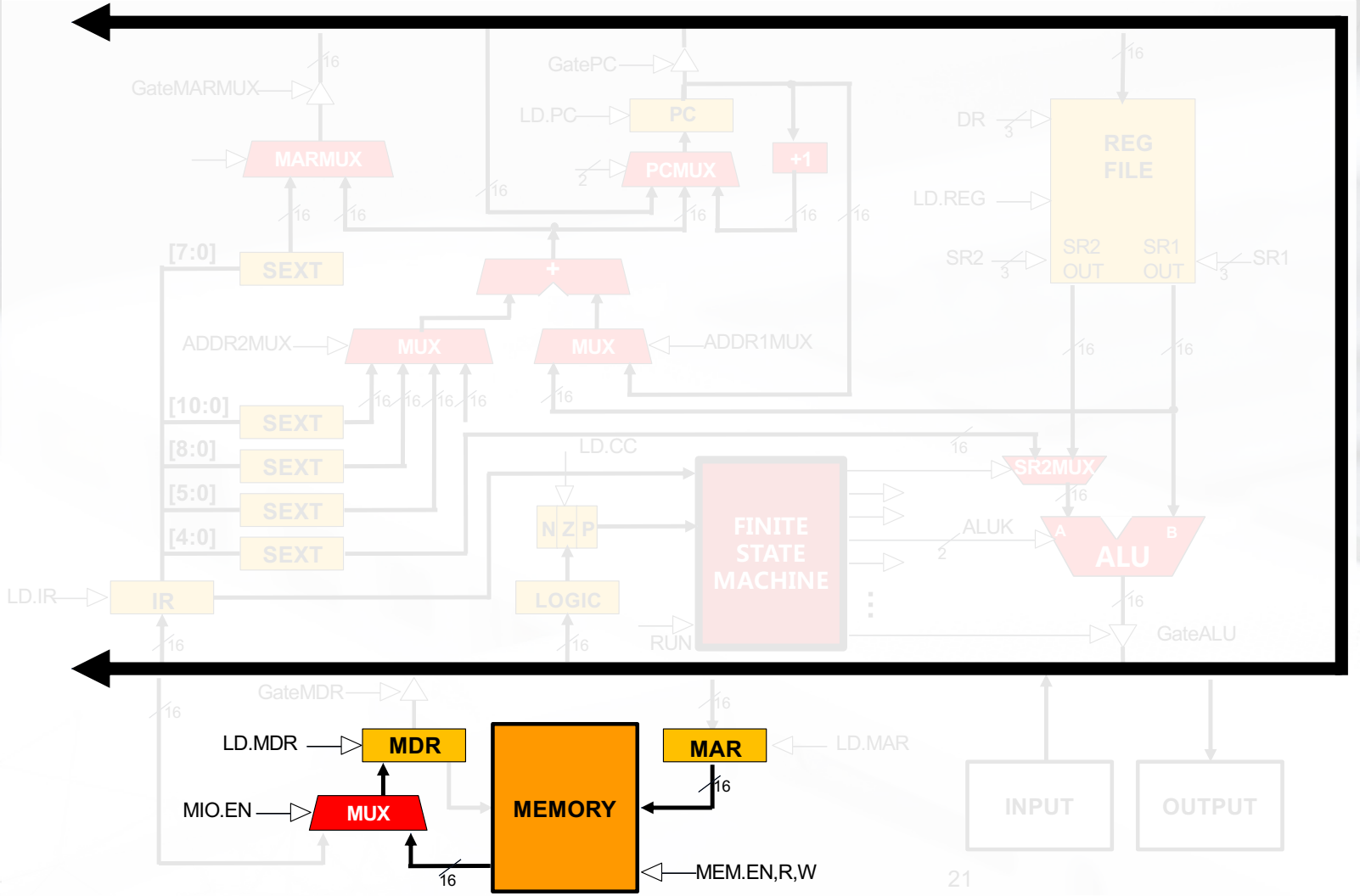
BR (PC-Relative)



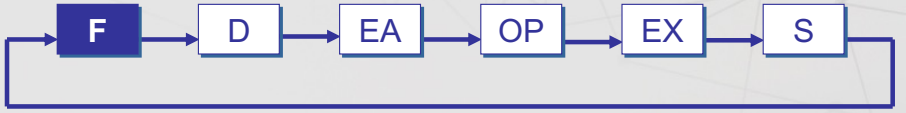


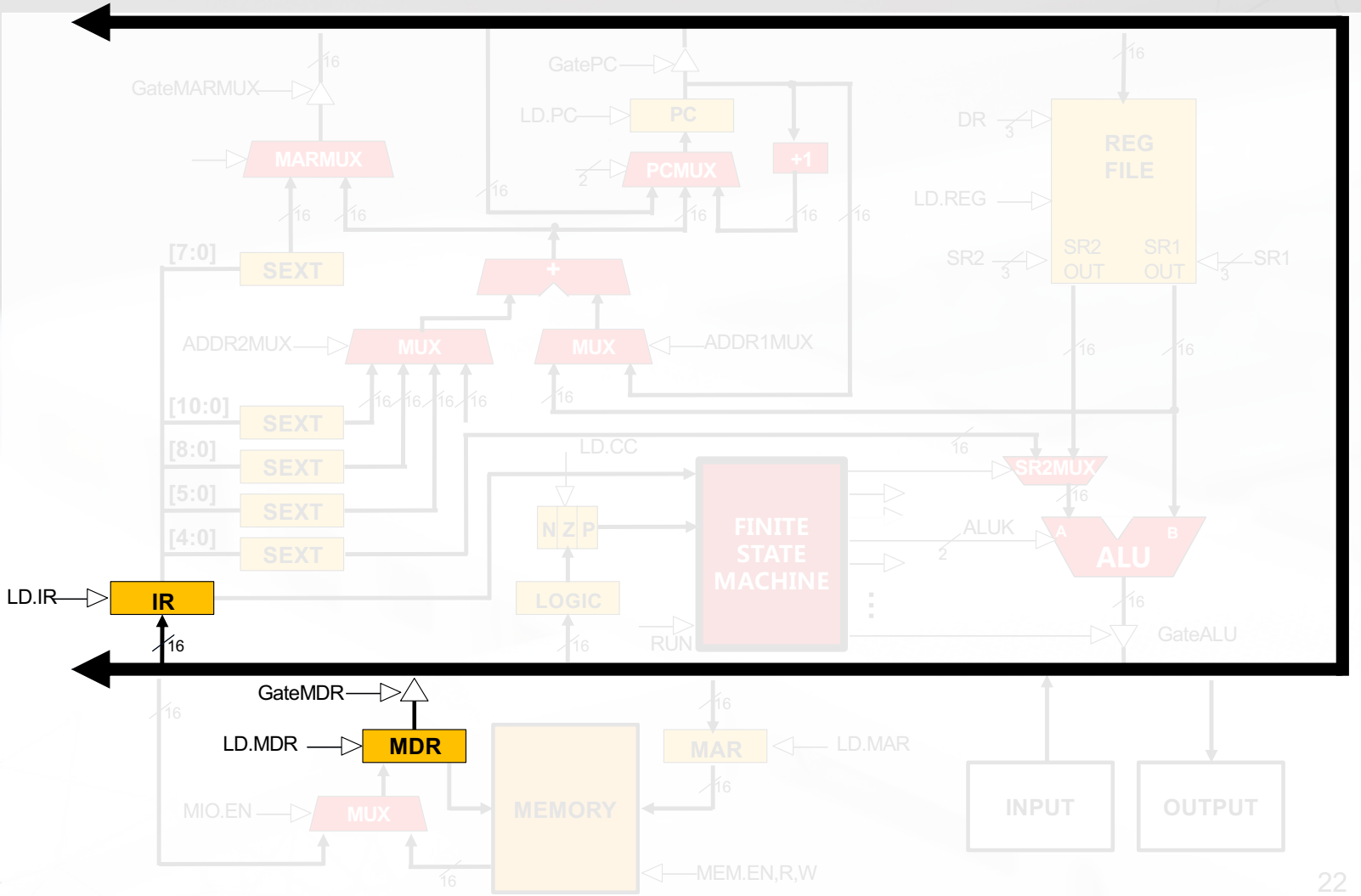
BR (PC-Relative)



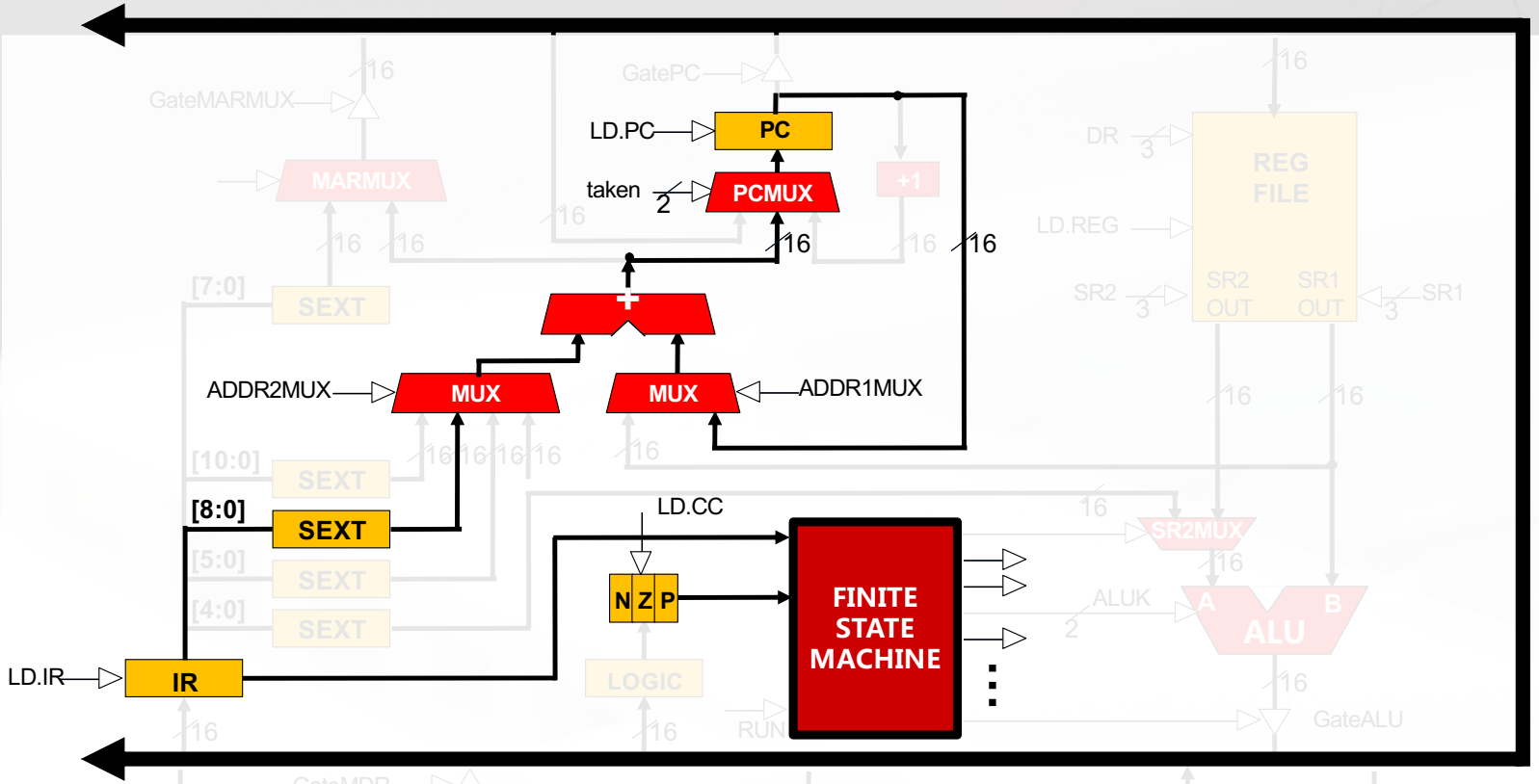


BR (PC-Relative)





BR (PC-Relative)



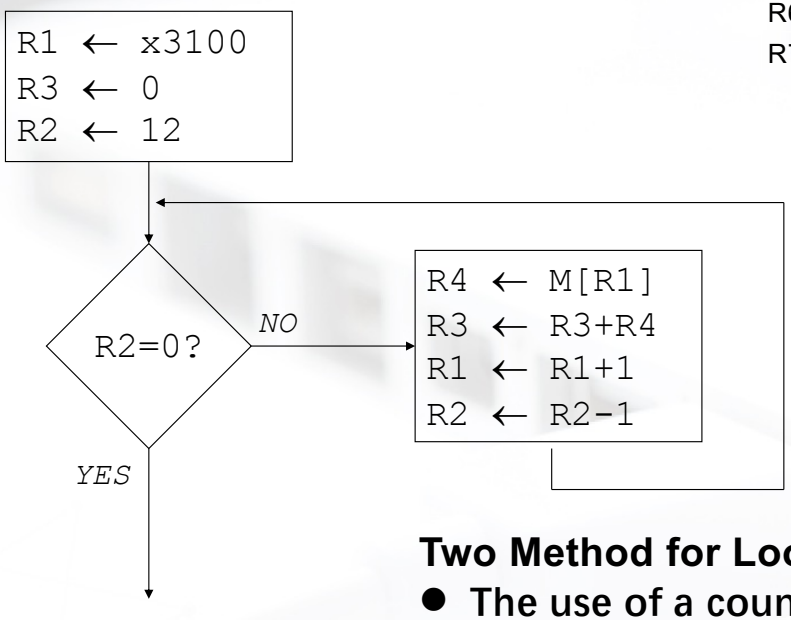
Using Branch Instructions

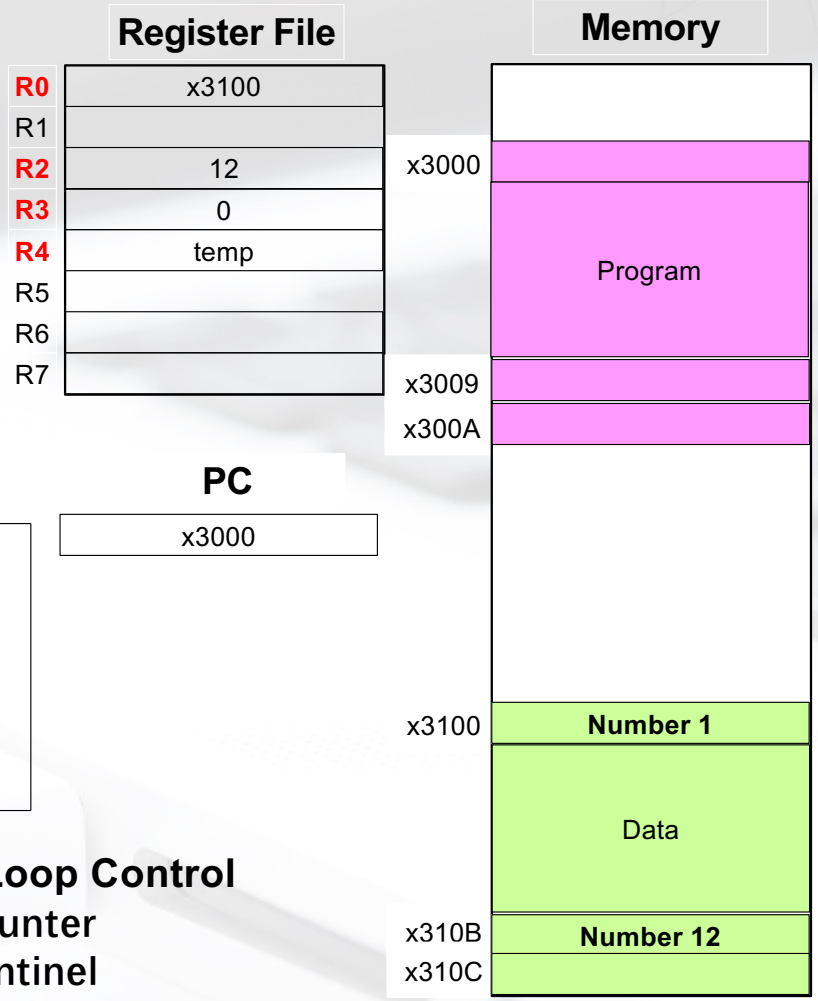
Compute sum of 12 integers.

Numbers start at location x3100.

Program starts at location x3000.

The use of a counter





Two Method for Loop Control

- The use of a counter
- The use of a sentinel

Sample Program(The use of a counter)

Address	*Instruction*	*Comments*
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	*R1 ← x3100 (PC+0xFF)*
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	*R3 ← 0*
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	*R2 ← 0*
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0	*R2 ← 12*
x3004	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1	*If Z, goto (PC+5) = x300A*
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	*Load next value to R4*
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	*R3 ← R3 + R4*
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	*Increment R1 (pointer)*
X3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	*Decrement R2 (counter)*
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0	*Goto (PC-6)=x3004*

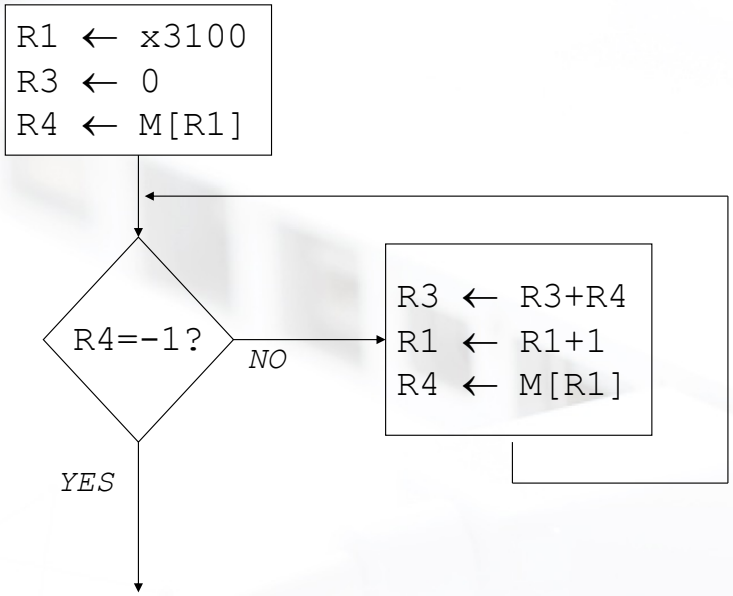
Using Branch Instructions

Compute sum of 12 integers.

Numbers start at location x3100.

Program starts at location x3000.

The use of a sentinel



A special character used to indicate the end of a sequence is often called a sentinel.

- Useful when you don't know ahead of time how many times to execute a loop.

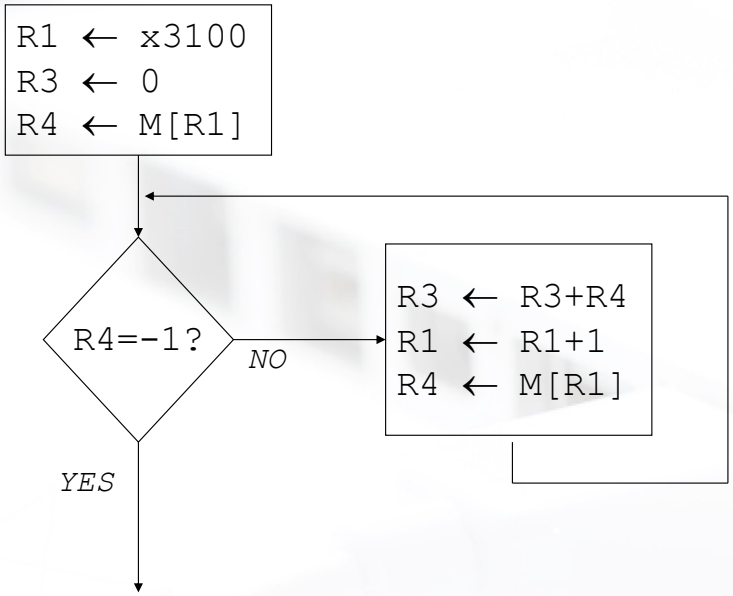
Using Branch Instructions

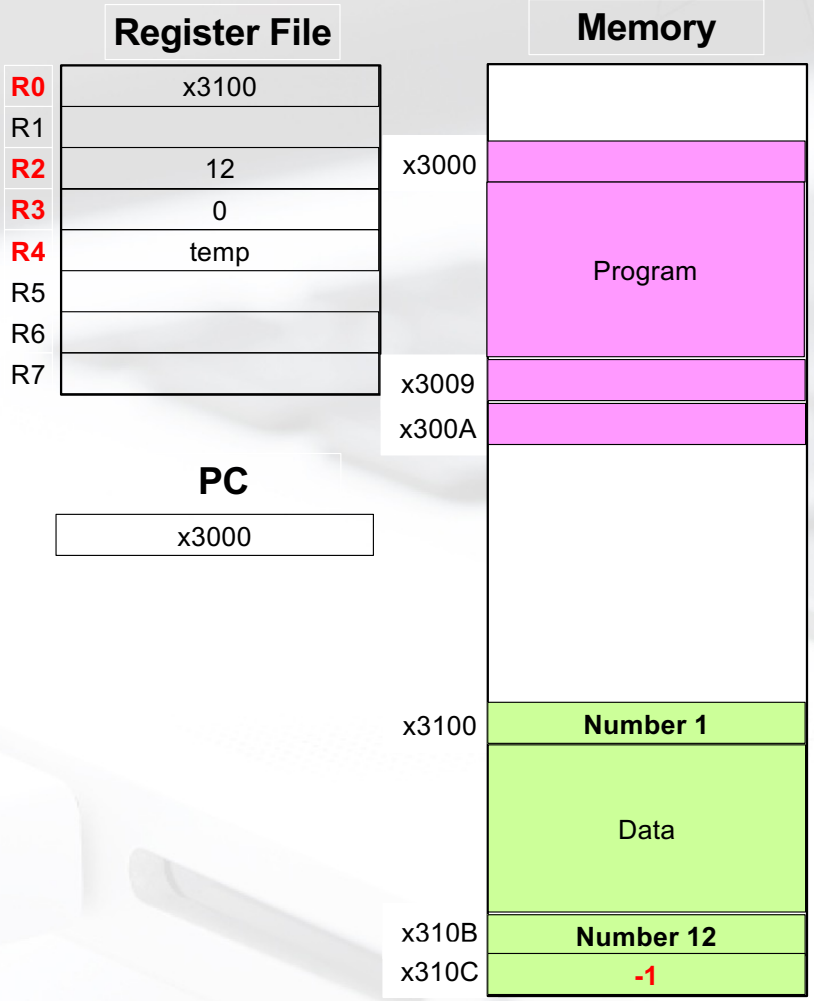
Compute sum of 12 integers.

Numbers start at location x3100.

Program starts at location x3000.

The use of a sentinel





Sample Program(The use of a sentinel)

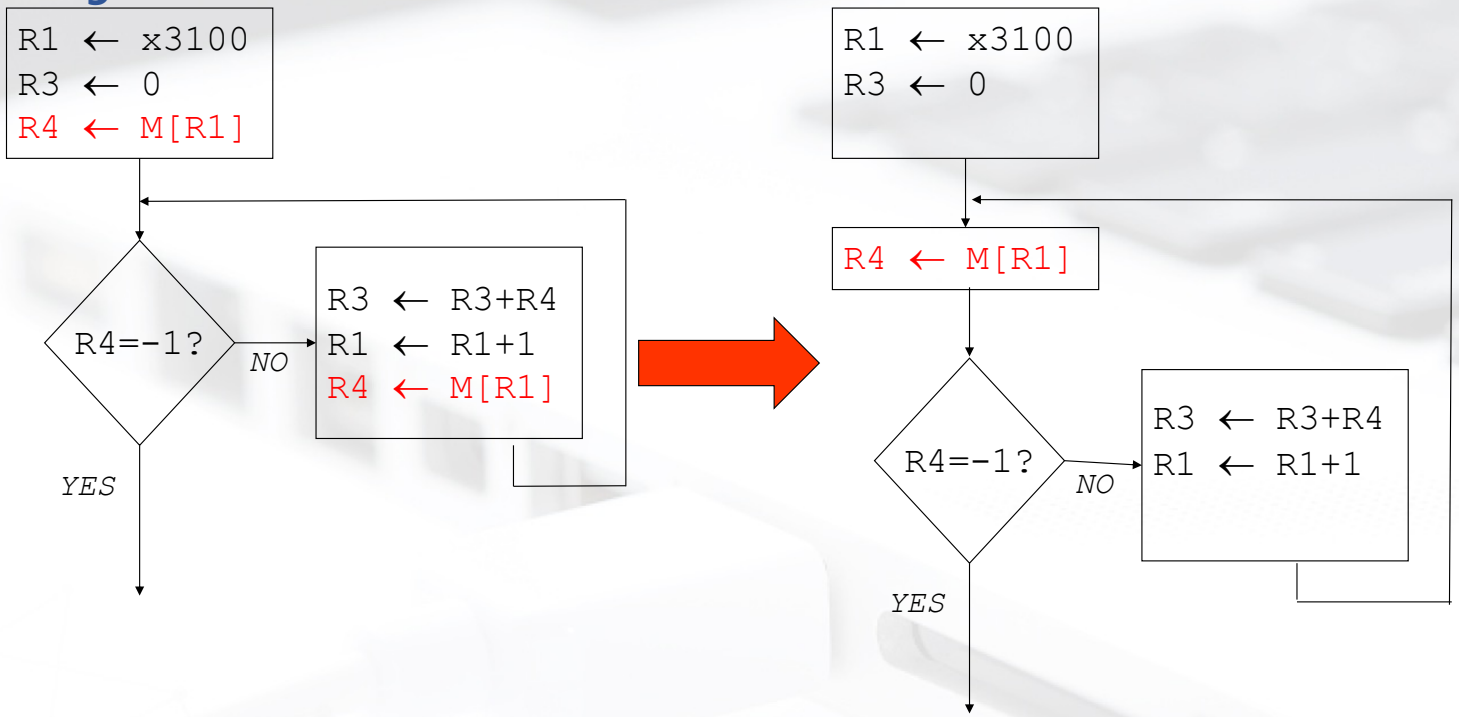
Address	*Instruction*	*Comments*
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	*R1 ← (PC+0xFF)=x3100*
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	*R3 ← 0*
x3002	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	*R4 ← M[R1]*
X3003	0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0	*If N, goto (PC+ 4)=X3008*
X3004	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	*R3 ← R3 + R4*
X3005	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	*R1 ← R1 + 1*
X3006	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	*R4 ← M[R1]*
x3007	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1	*Goto (PC-5)= x3003*

Using Branch Instructions(Code Optimization)

Compute sum of 12 integers.

Numbers start at location x3100.

Program starts at location x3000.



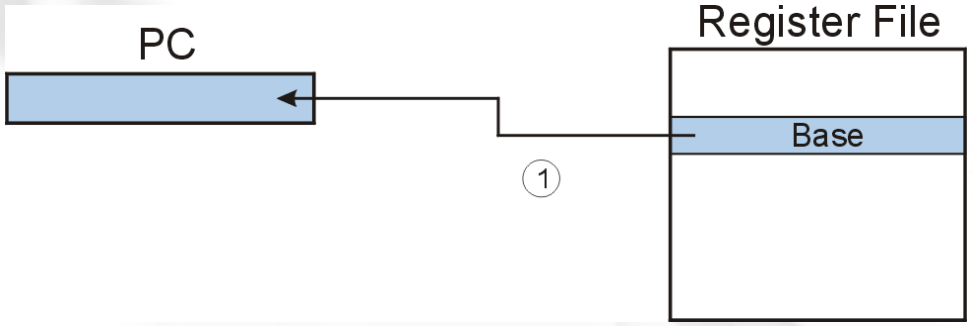
Sample Program

Address	Instruction	Comments
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	*R1 ← (PC+0xFF)= x3100*
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	*R3 ← 0*
x3002	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	*R4 ← M[R1]*
X3003	0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1	*If N, goto (PC+ 3)=X3007*
X3004	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	*R3 ← R3 + R4*
X3005	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	*R1 ← R1 + 1*
x3006	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1	*Goto (PC-5)= x3002*

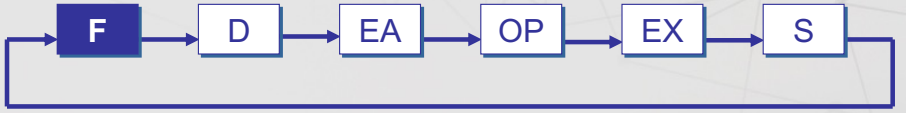
JMP (Register)

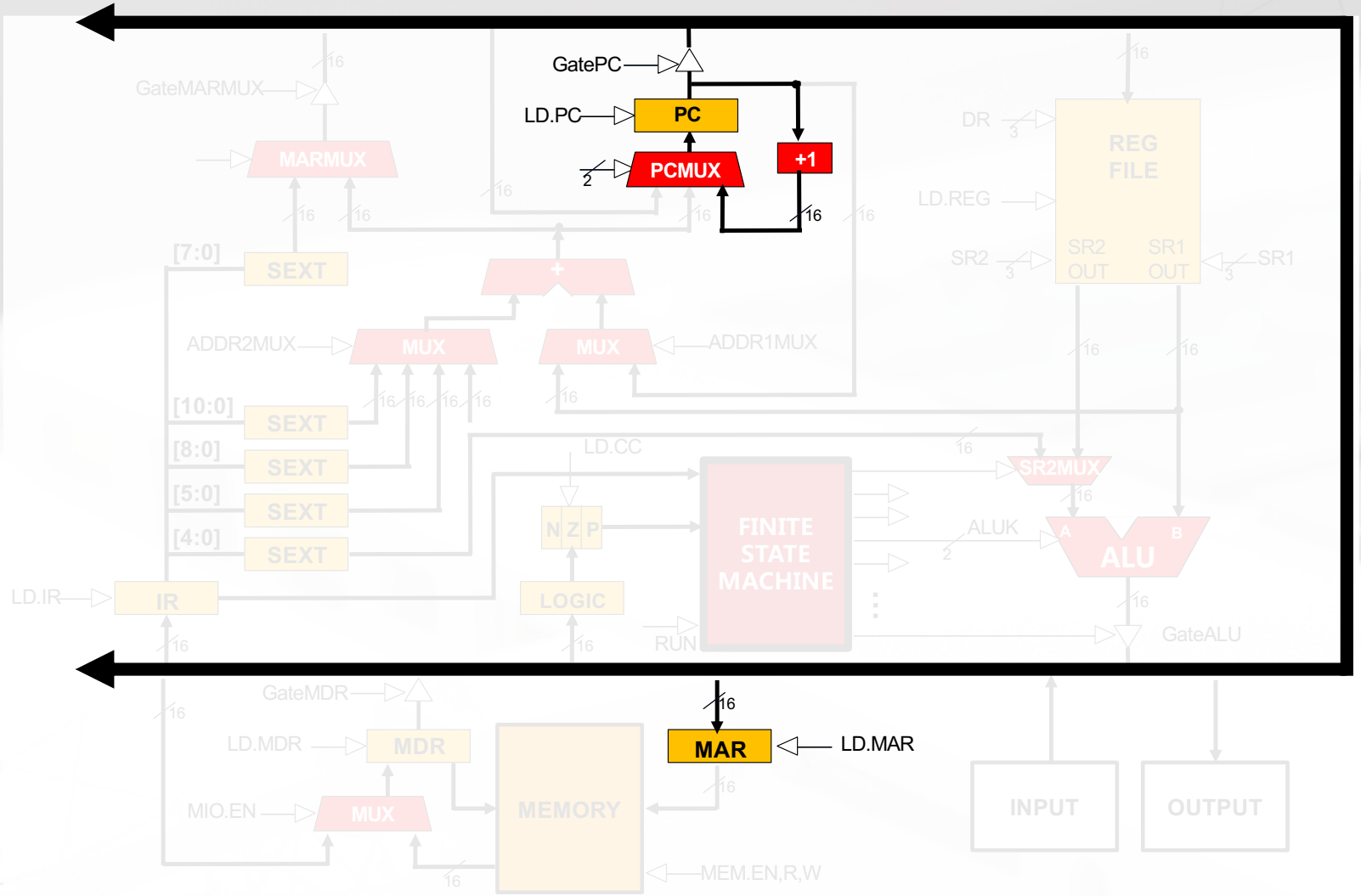
Jump is an unconditional branch -- *always* taken.

- Target address is the contents of a register.
- Allows any target address.

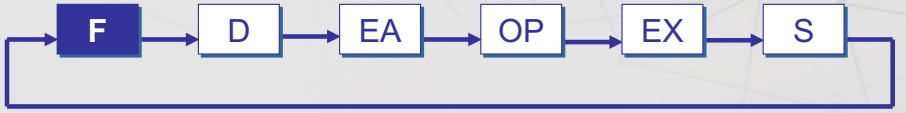


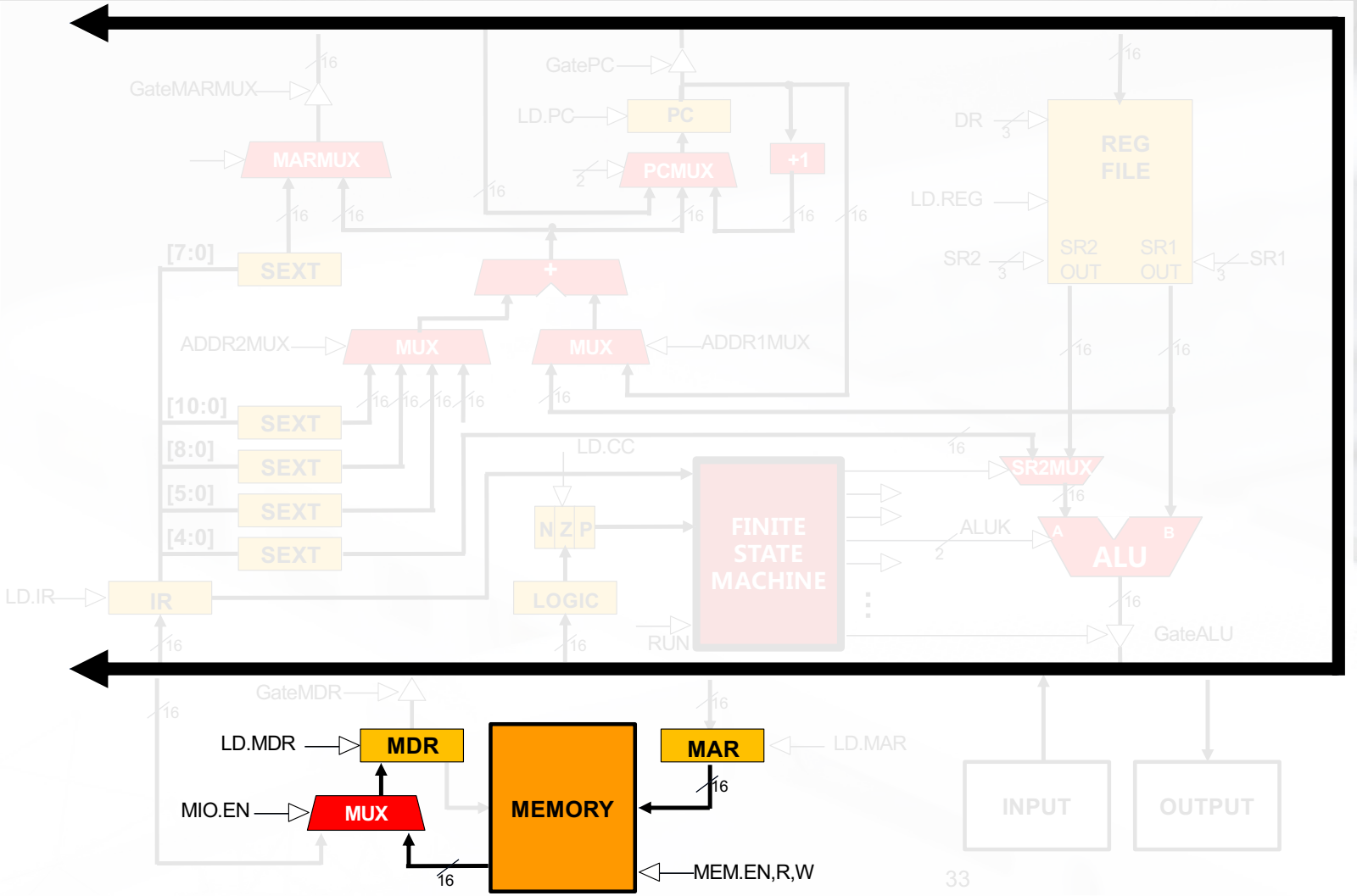
JMP (Register)



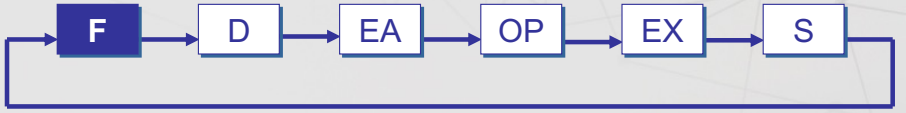


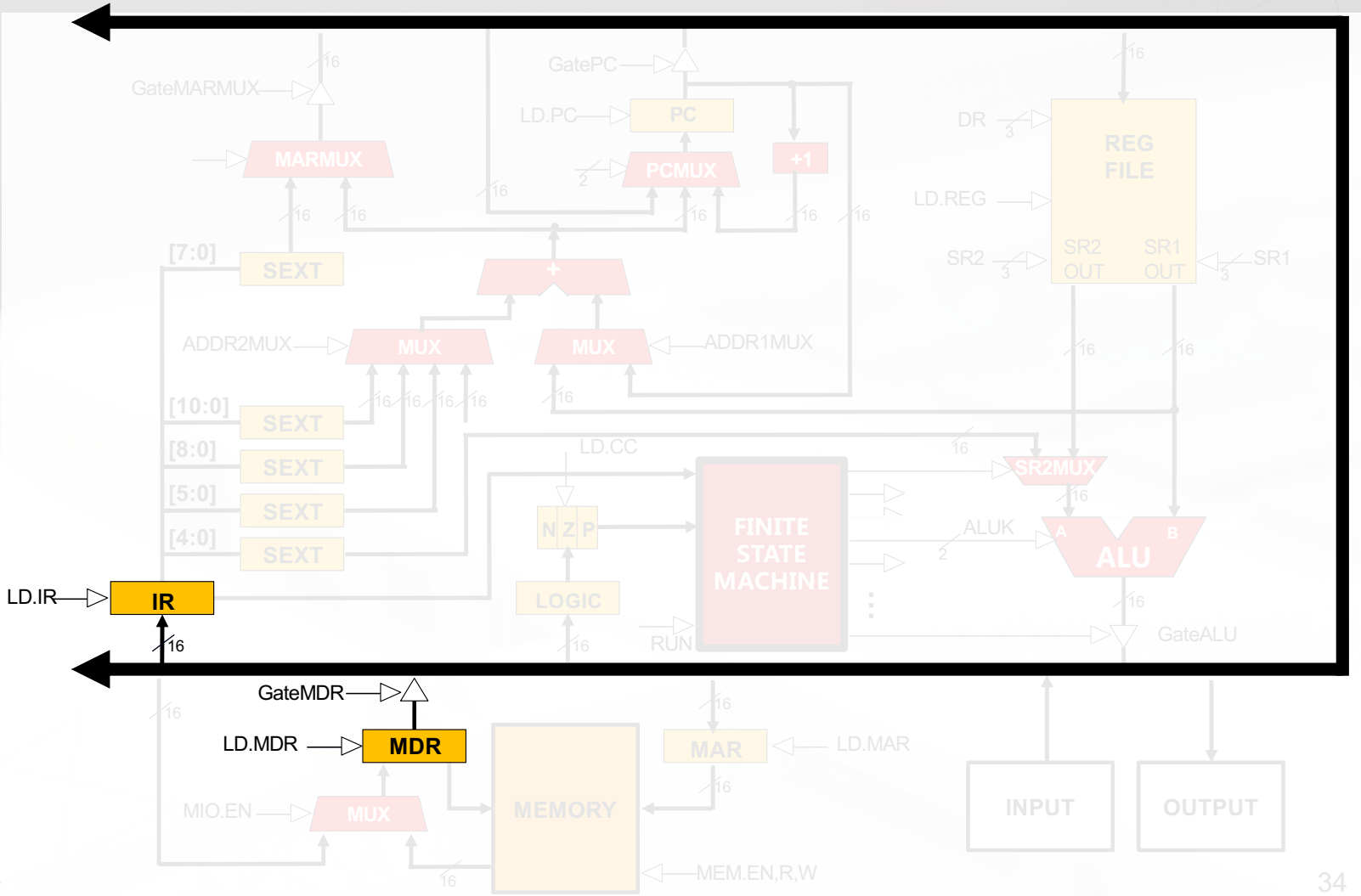
JMP (Register)



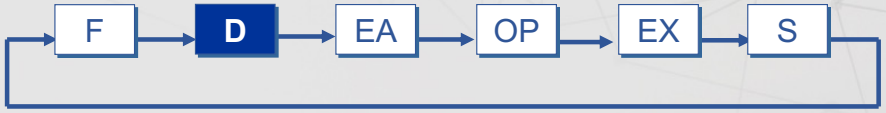


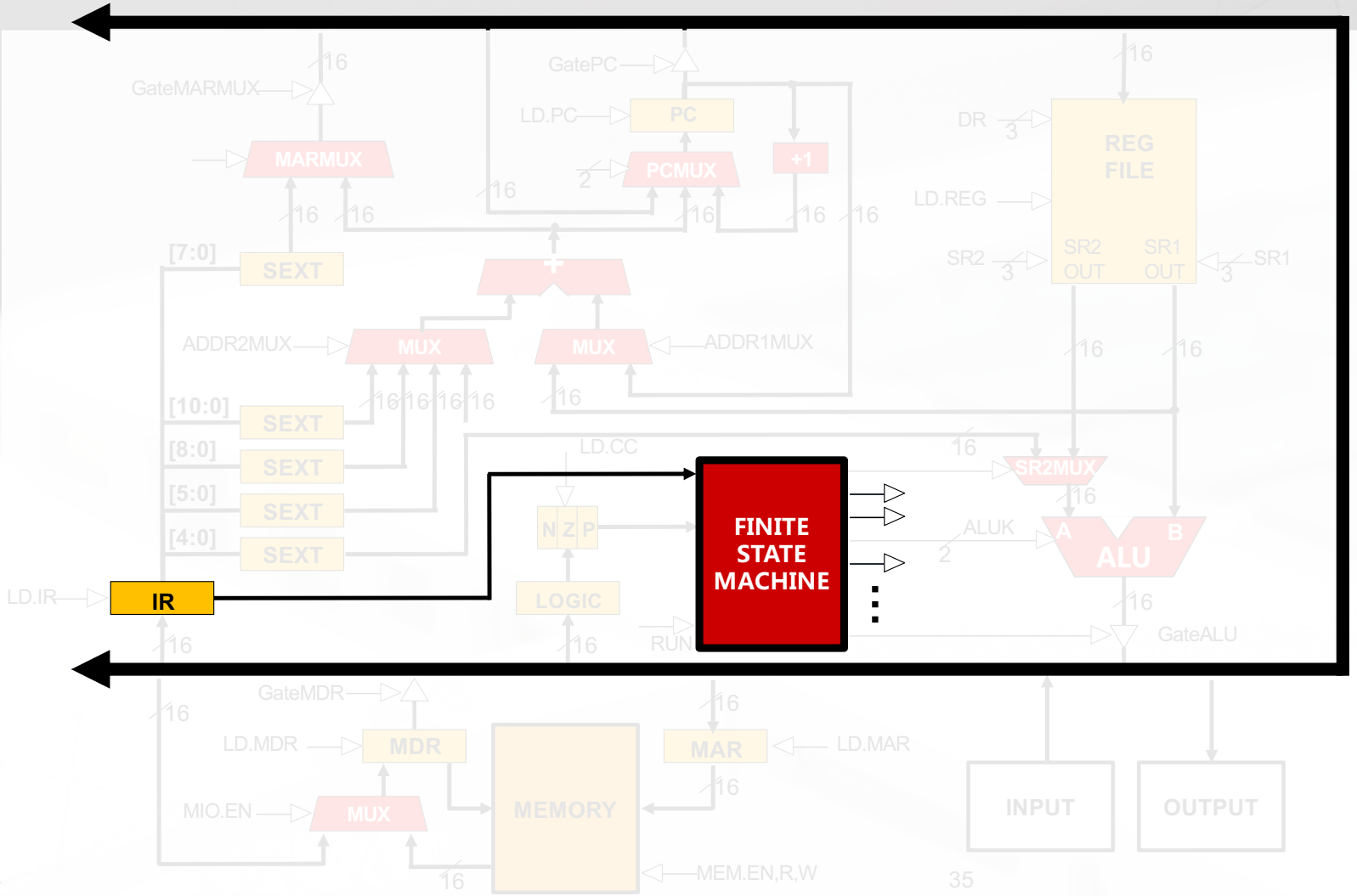
JMP (Register)



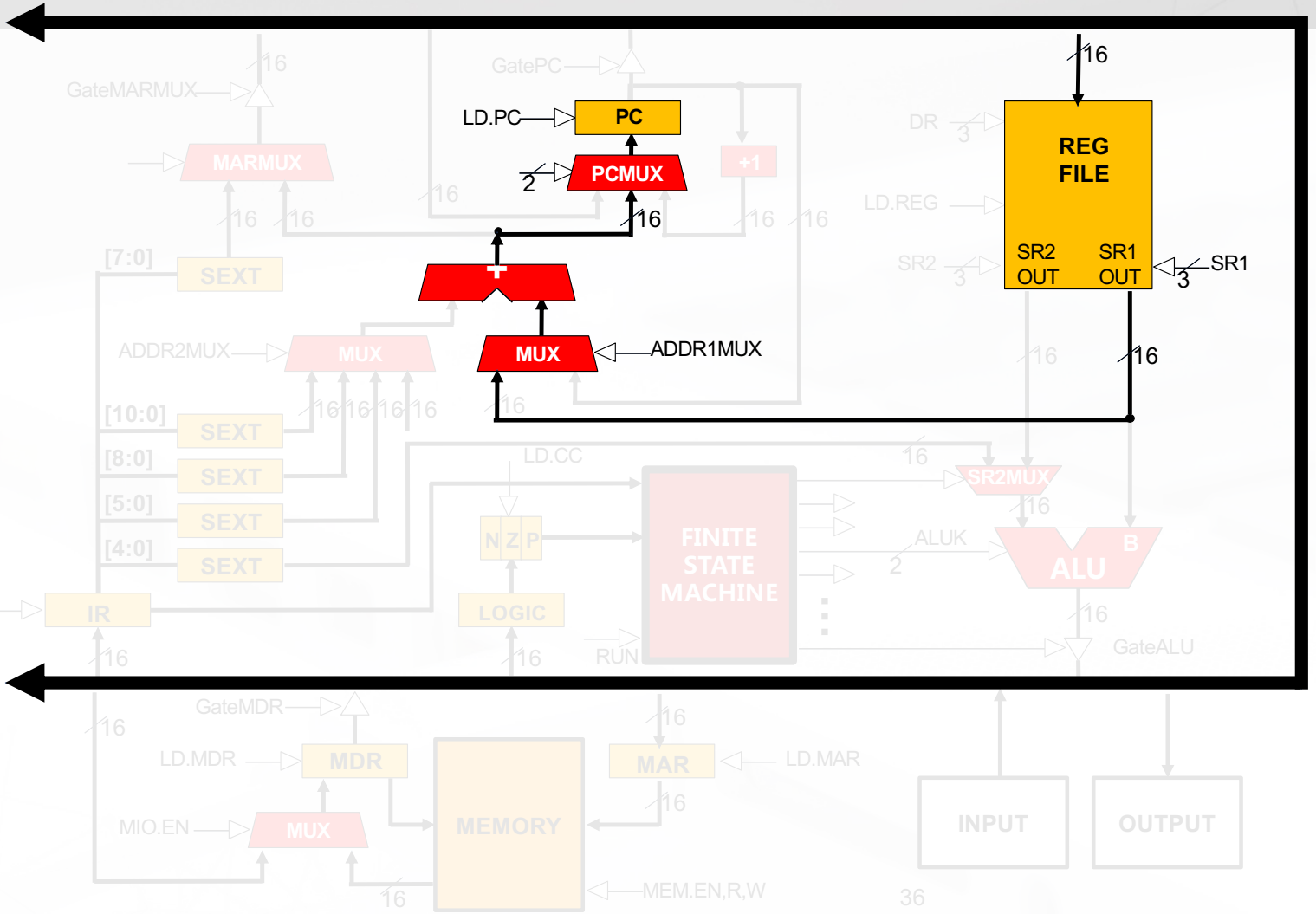


JMP (Register)





JMP (Register)



TRAP

Calls a service routine, identified by 8-bit "trap vector."

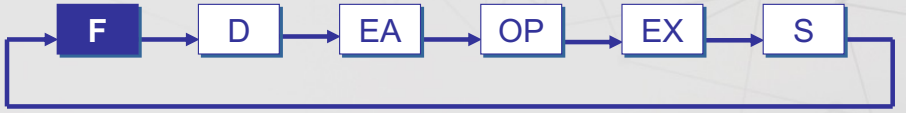
vector	*routine*
x23	input a character from the keyboard
x21	output a character to the monitor
x25	halt the program

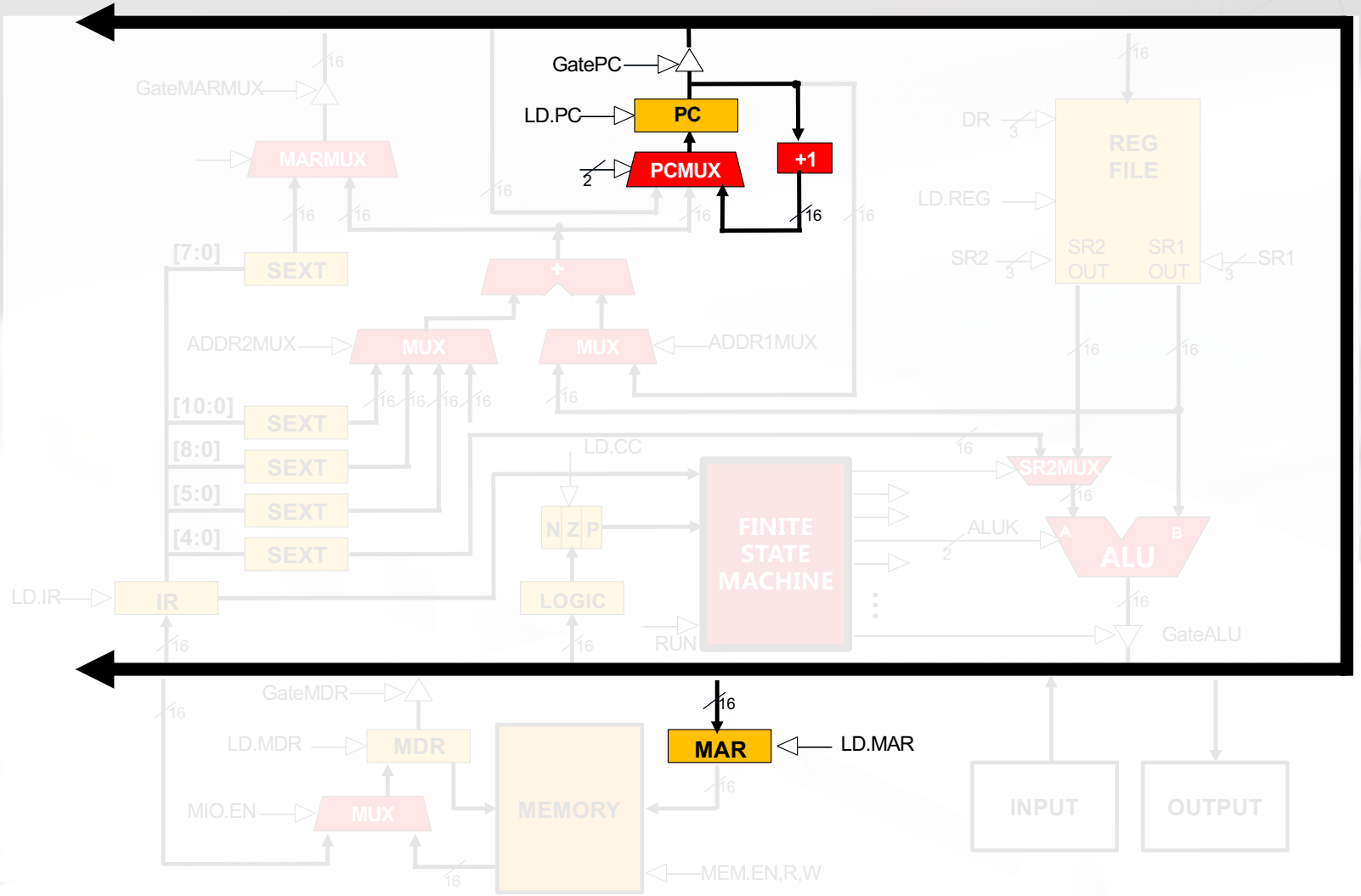
When routine is done,

PC is set to the instruction following TRAP.

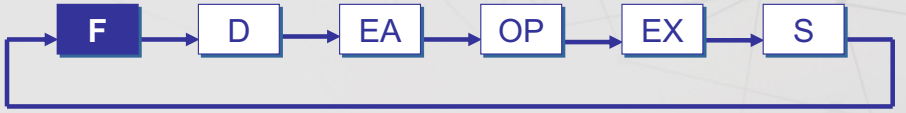
(We'll talk about how this works later.)

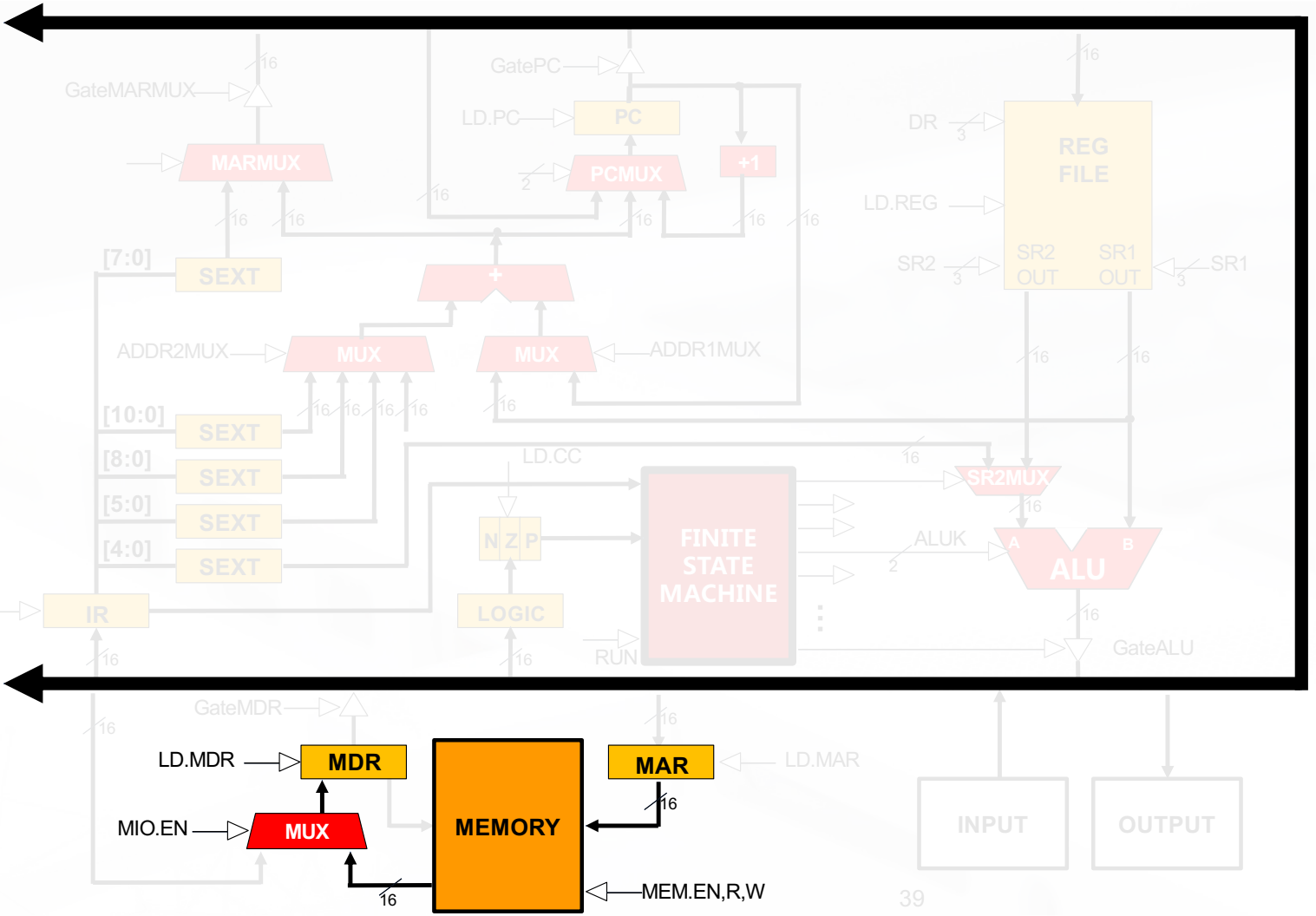
TRAP


F
D
EA
OP
EX
S

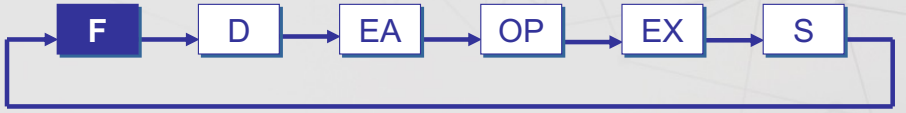

GatePC
LD.PC
PC
PCMUX
+1
2
16
GateMARMUX
MARMUX
[7:0]
SEXT
+
ADDR2MUX
MUX
MUX
ADDR1MUX
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
LD.CC
N Z P
FINITE STATE MACHINE
LD.IR
IR
LOGIC
RUN
DR
REG FILE
LD.REG
SR2
SR2 OUT
SR1 OUT
SR1
SR2MUX
ALUK
A
B
ALU
GateALU
GateMDR
MDR
LD.MDR
MIO.EN
MUX
MEMORY
MEM.EN,R,W
MAR
LD.MAR
INPUT
OUTPUT

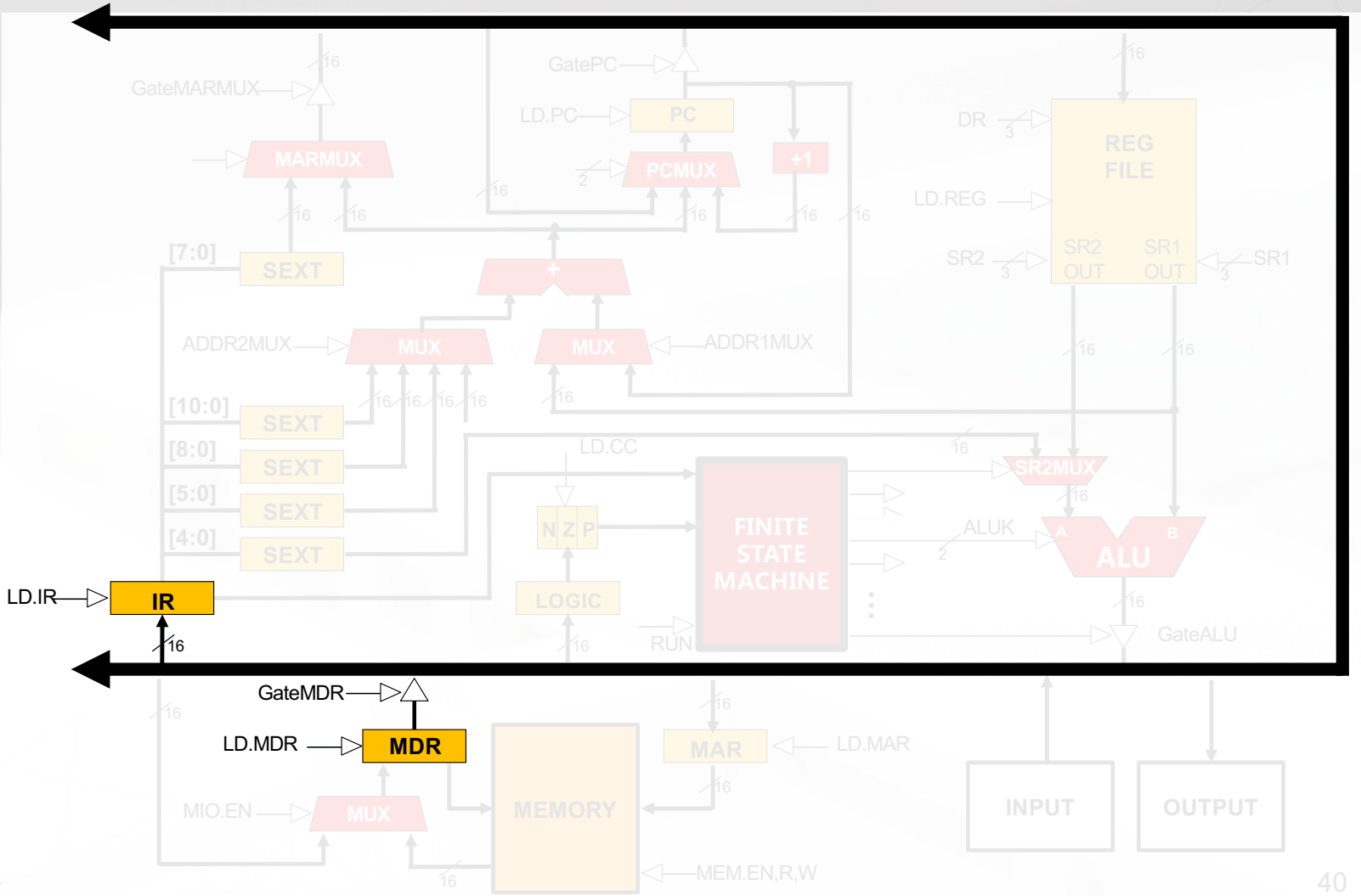
TRAP


F
D
EA
OP
EX
S

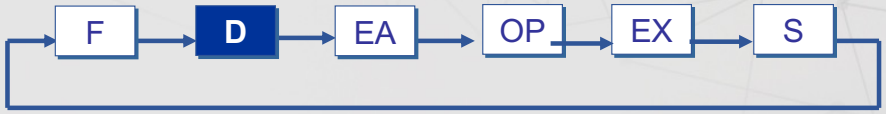

GateMARMUX
GatePC
LD.PC
PC
MARMUX
PCMUX
+1
DR
REG FILE
LD.REG
SR2
SR2 OUT
SR1 OUT
SR1
[7:0]
SEXT
+
ADDR2MUX
MUX
MUX
ADDR1MUX
[10:0]
[8:0]
[5:0]
[4:0]
LD.CC
SR2MUX
N Z P
FINITE STATE MACHINE
ALUK
A
B
ALU
LD.IR
IR
LOGIC
RUN
GateALU
GateMDR
LD.MDR
MDR
MAR
LD.MAR
MIO.EN
MUX
MEMORY
MEM.EN,R,W
INPUT
OUTPUT

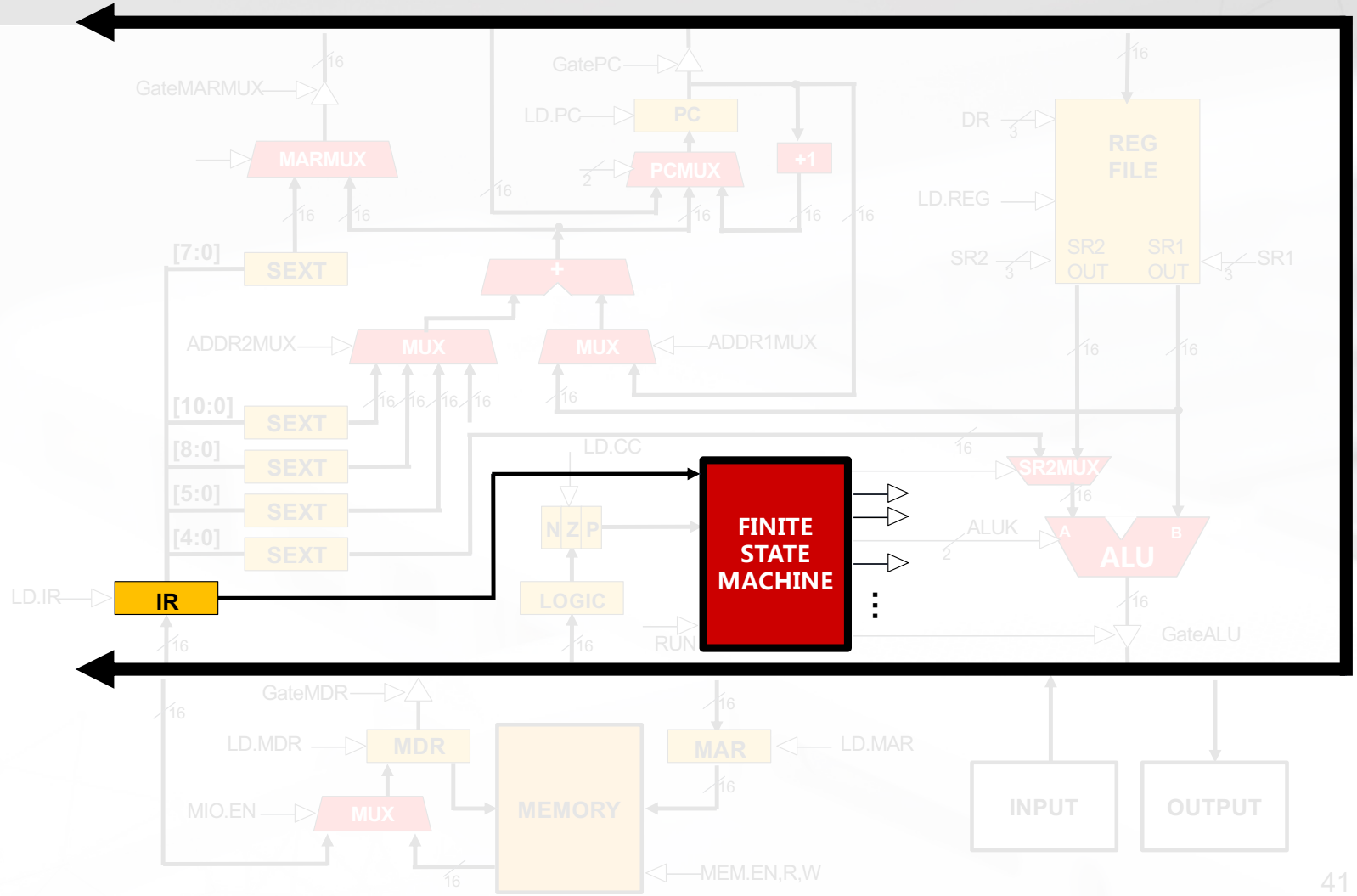
TRAP


F
D
EA
OP
EX
S


GateMARMUX
GatePC
LD.PC
PC
PCMUX
+1
MARMUX
DR
REG FILE
LD.REG
SR2
SR1
SR2 OUT
SR1 OUT
[7:0]
SEXT
+
ADDR2MUX
MUX
MUX
ADDR1MUX
[10:0]
[8:0]
[5:0]
[4:0]
LD.CC
SR2MUX
N Z P
FINITE STATE MACHINE
ALUK
A
B
ALU
LD.IR
IR
LOGIC
RUN
GateALU
GateMDR
LD.MDR
MDR
MAR
LD.MAR
MIO.EN
MUX
MEMORY
INPUT
OUTPUT
MEM.EN,R,W

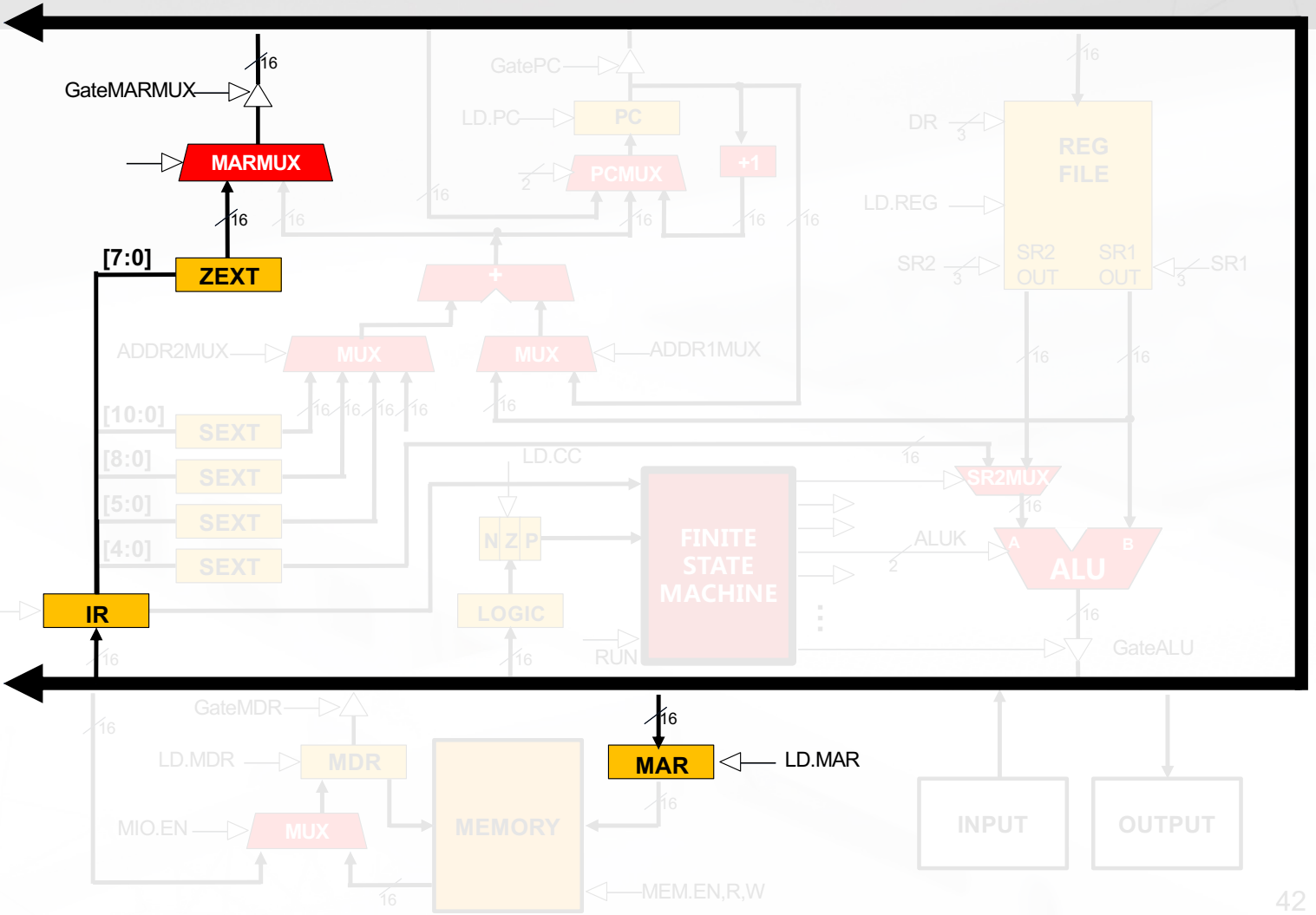
TRAP


F
D
EA
OP
EX
S


GateMARMUX
GatePC
LD.PC
PC
MARMUX
PCMUX
+1
DR
REG FILE
LD.REG
SR2
SR2 OUT
SR1 OUT
SR1
[7:0]
SEXT
+
ADDR2MUX
MUX
MUX
ADDR1MUX
[10:0]
[8:0]
[5:0]
[4:0]
LD.CC
SR2MUX
N Z P
FINITE STATE MACHINE
ALUK
A
B
ALU
LD.IR
IR
LOGIC
RUN
GateALU
GateMDR
LD.MDR
MDR
MAR
LD.MAR
MIO.EN
MUX
MEMORY
INPUT
OUTPUT
MEM.EN,R,W

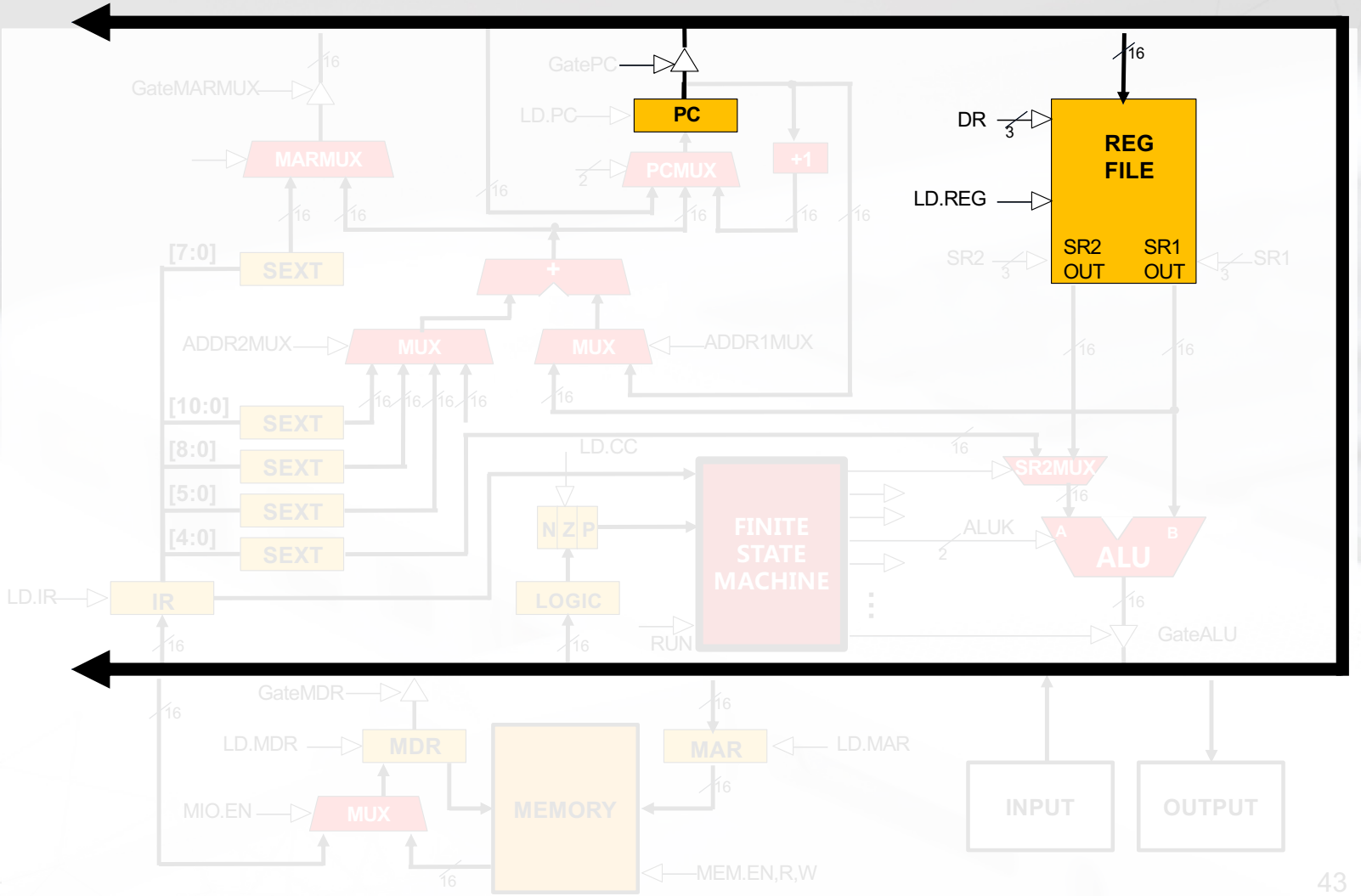
TRAP


F
D
EA
OP
EX
S

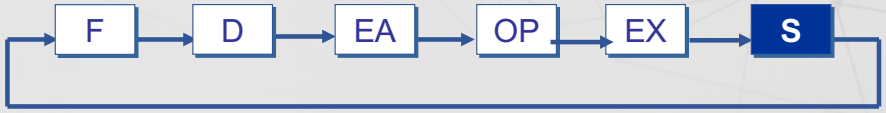

GateMARMUX
16
MARMUX
16
[7:0]
ZEXT
GatePC
LD.PC
PC
PCMUX
+1
+
ADDR2MUX
MUX
MUX
ADDR1MUX
[10:0]
SEXT
[8:0]
SEXT
[5:0]
SEXT
[4:0]
SEXT
LD.CC
N Z P
LOGIC
FINITE STATE MACHINE
RUN
LD.IR
IR
DR
REG FILE
LD.REG
SR2
SR2 OUT
SR1 OUT
SR1
SR2MUX
ALUK
A
B
ALU
GateALU
GateMDR
LD.MDR
MDR
MIO.EN
MUX
MEMORY
MEM.EN,R,W
MAR
LD.MAR
INPUT
OUTPUT

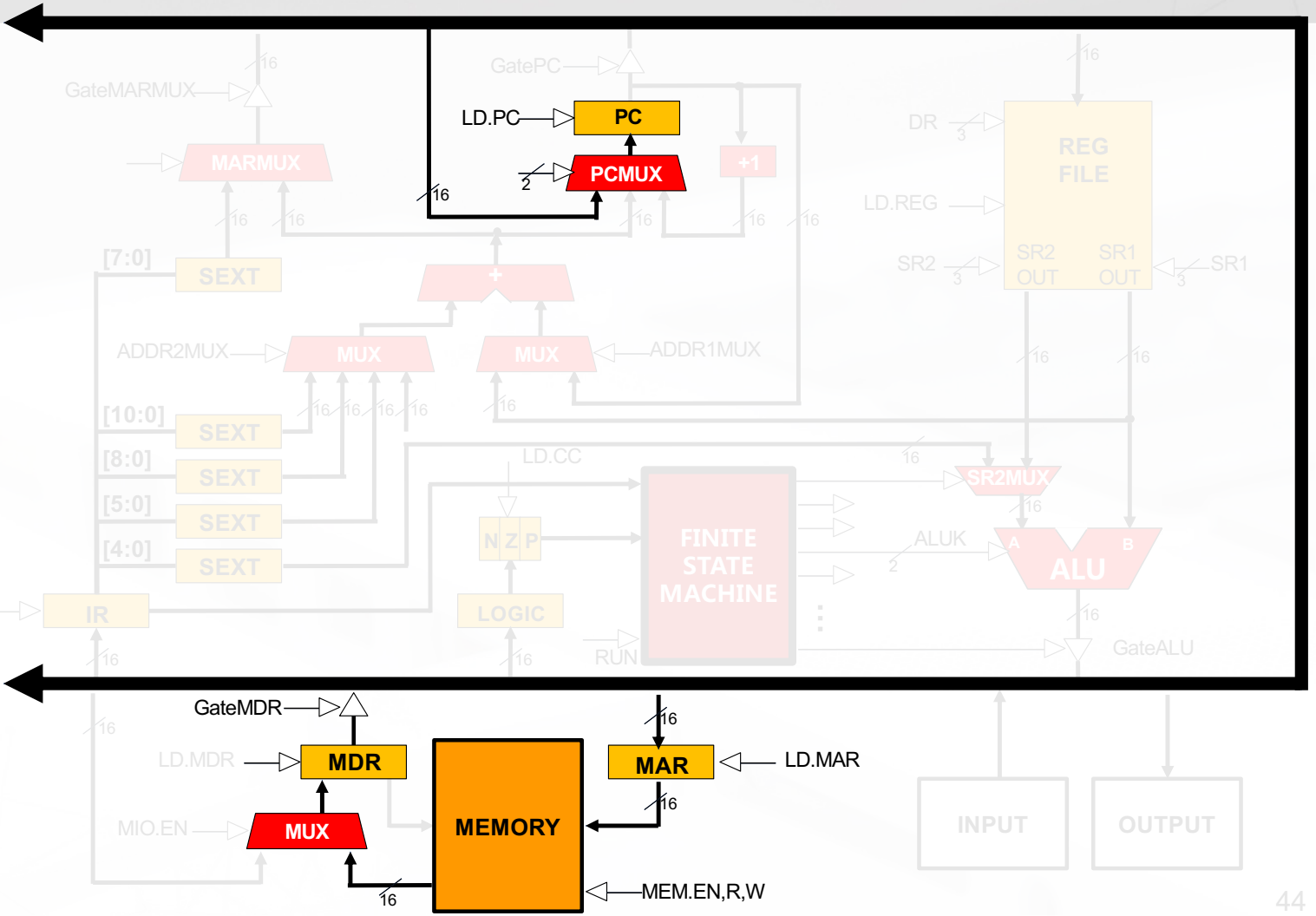
TRAP


F
D
EA
OP
EX
S

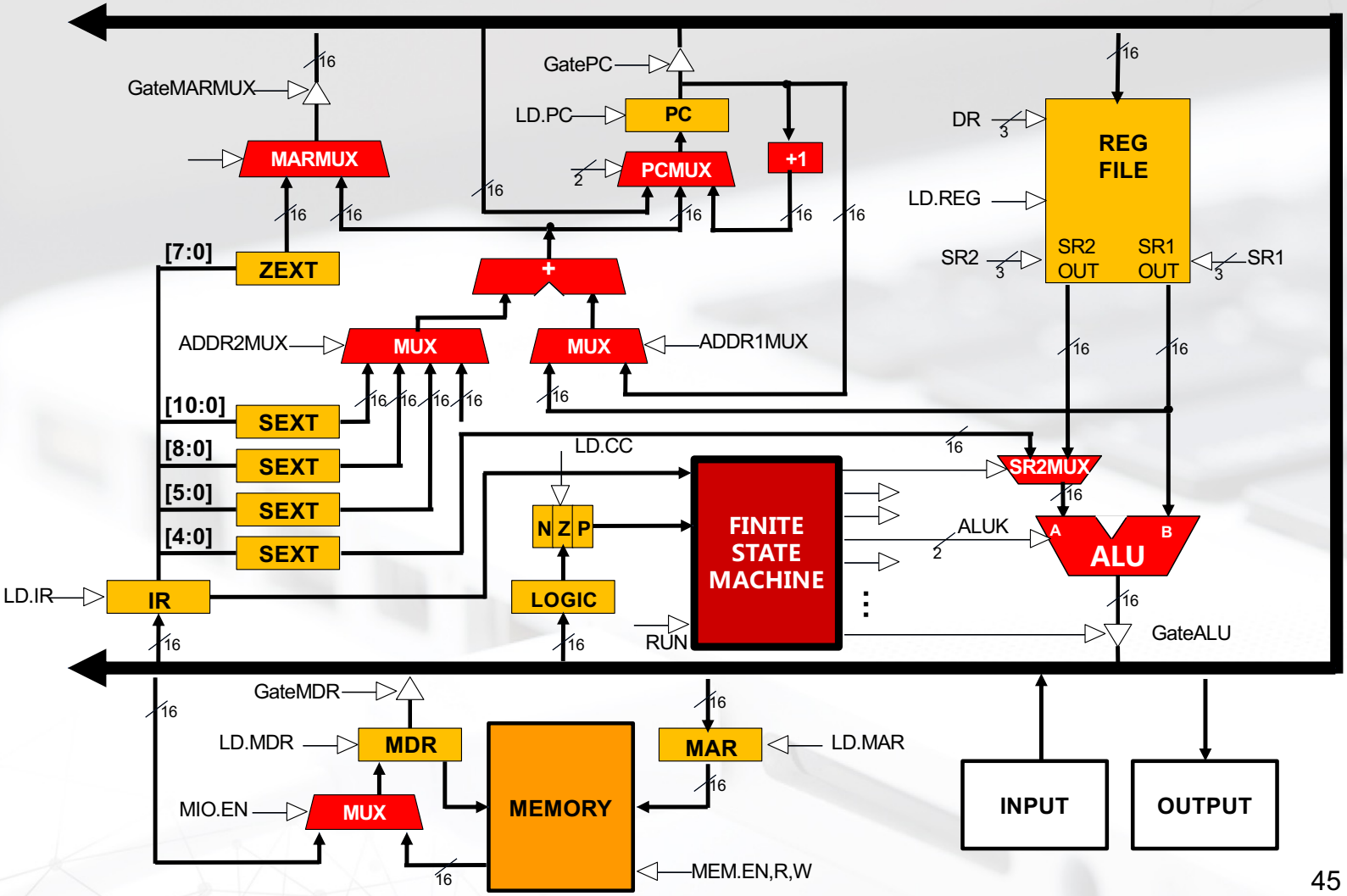

GatePC
LD.PC
PC
PCMUX
+1
GateMARMUX
MARMUX
16
DR
3
REG FILE
LD.REG
SR2 OUT
SR1 OUT
SR2
SR1
[7:0]
SEXT
+
ADDR2MUX
MUX
MUX
ADDR1MUX
[10:0]
[8:0]
[5:0]
[4:0]
LD.CC
SR2MUX
N Z P
FINITE STATE MACHINE
ALUK
A
B
ALU
LD.IR
IR
LOGIC
RUN
GateALU
GateMDR
LD.MDR
MDR
MAR
LD.MAR
MIO.EN
MUX
MEMORY
INPUT
OUTPUT
MEM.EN,R,W

TRAP


F
D
EA
OP
EX
S


GatePC
LD.PC
PC
PCMUX
+1
16
2
GateMARMUX
MARMUX
DR
REG FILE
LD.REG
SR2
SR1
SR2 OUT
SR1 OUT
[7:0]
SEXT
+
ADDR2MUX
MUX
MUX
ADDR1MUX
[10:0]
[8:0]
[5:0]
[4:0]
LD.CC
SR2MUX
N Z P
FINITE STATE MACHINE
ALUK
A
B
ALU
LD.IR
IR
LOGIC
RUN
GateALU
GateMDR
LD.MDR
MDR
MAR
LD.MAR
MIO.EN
MUX
MEMORY
MEM.EN,R,W
INPUT
OUTPUT

LC-3 Data Path After Control Instruction



Outline

1 LC-3 ISA Overview

2 Operate Instructions and Data Path

3 Data Movement Instructions

4 Control Instructions

5 **Another Example: Counting Occurrences of a Computer**

6 The Data Path Revisited

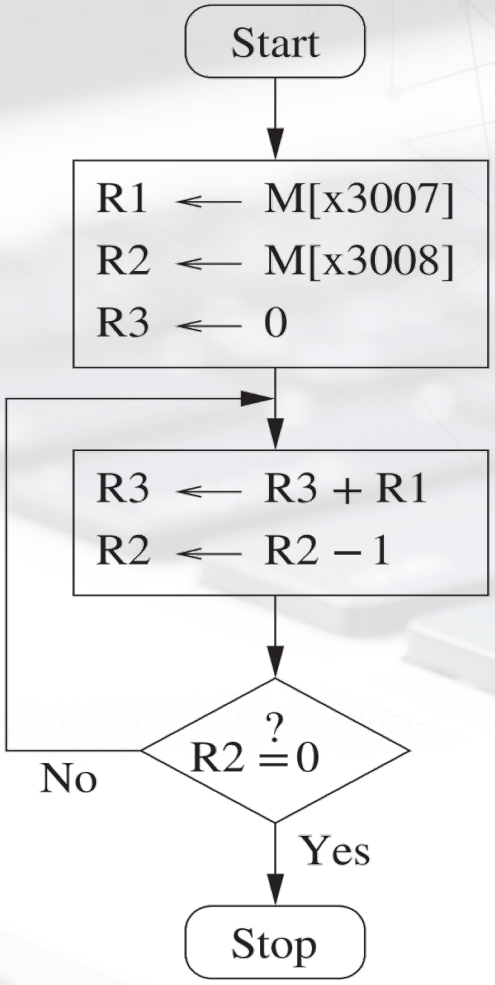
Previously: An algorithm for 5 x 4

- We had ADD, AND, LD, BR, HALT(TRAP)
- We had ADD instructions, but did not have multiply instructions. So, we do

5 x 4 =5+5+5+5

M[x3007]=5

M[x3008]=4



A program that multiplies without a multiply instruction

Address	*Instruction*	*Comments*
x3000	0 0 1 0 0 0 1 0 0 0 0 0 0 1 1 0	*R1 ← M[x3007]*
x3001	0 1 0 1 0 1 0 0 0 0 0 0 0 1 1 0	*R2 ← M[x3008]*
x3002	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	*R3 ← 0*
x3003	0 0 0 1 0 1 1 0 1 1 0 0 0 0 0 0	*R3 ← R3+R1*
x3004	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	*R2 ← R2-1*
x3005	0 0 0 0 1 0 1 1 1 1 1 1 1 1 0 1	*BR not zero M[x3003]*
x3006	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	*HALT*
x3007	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	*The value 5*
X3008	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0	*The value 4*

The Three Constructs: Sequential, Conditional, Iterative

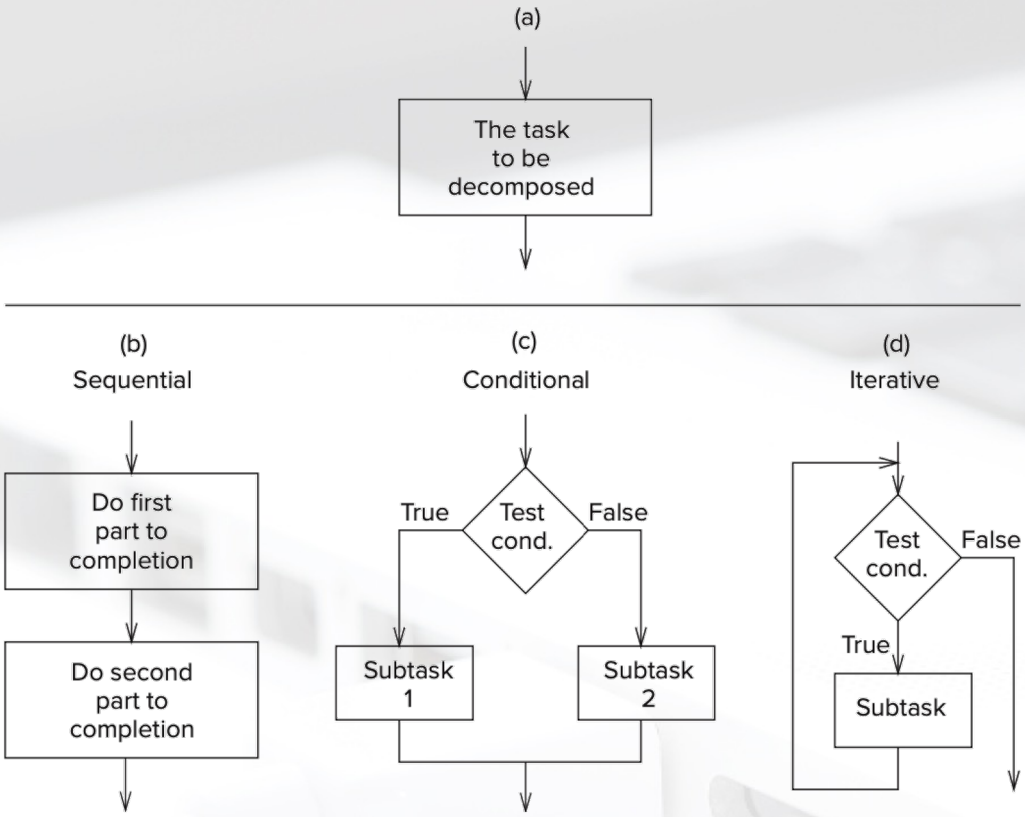


Figure 6.1 The basic constructs of structured programming.

LC-3 Control Instructions to Implement the Three Constructs

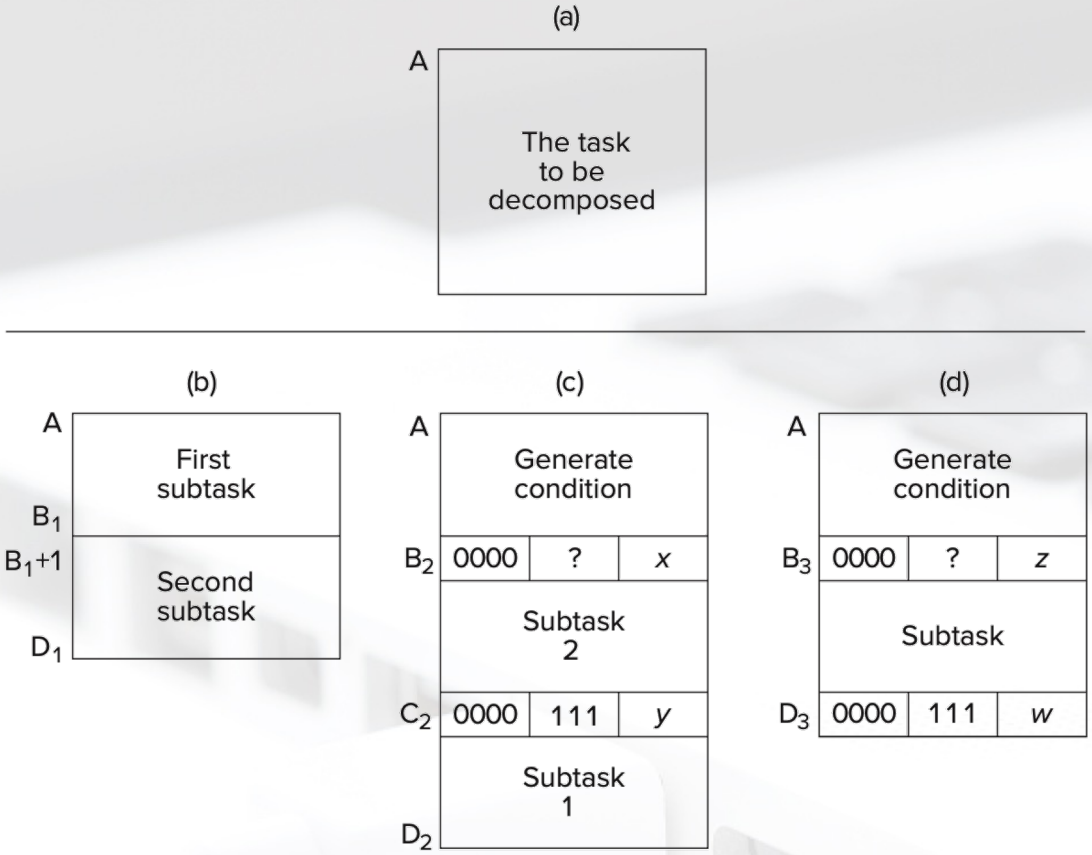


Figure 6.2 Use of LC-3 control instructions to implement structured programming.

Counting the occurrences of a character in a file

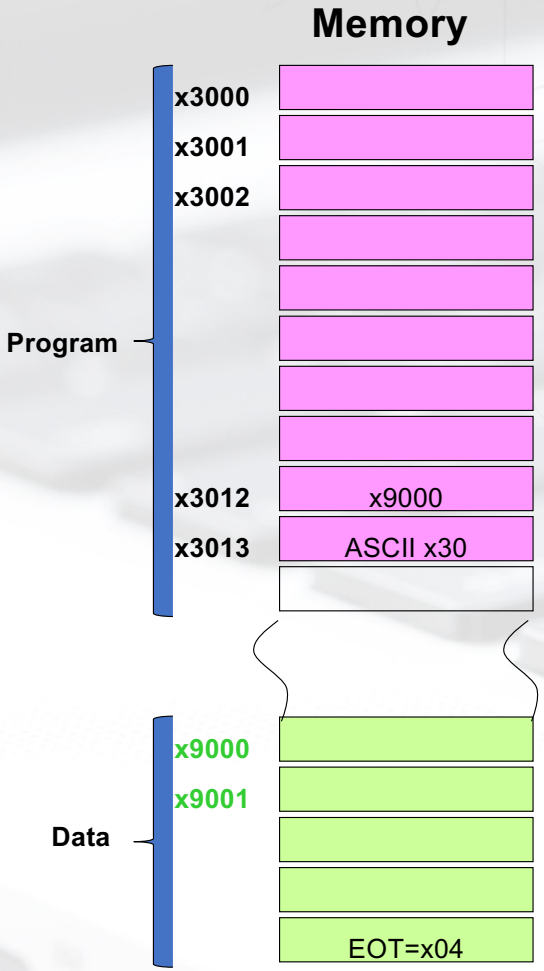
- Program begins at location x3000
- Read character from keyboard
- Load each character from a "file"
  - `File is a sequence of memory locations`
  - `Starting address of file is stored in the memory location immediately after the program`
- If file character equals input character, increment counter
- End of file is indicated by a special ASCII value: EOT (x04)
- At the end, print the number of characters and halt

(assume there will be less than 10 occurrences of the character)

Counting the occurrences of a character in a file

A special character used to indicate the end of a sequence is often called a sentinel.

- Useful when you don't know ahead of time how many times to execute a loop.



Register and Memory

Register

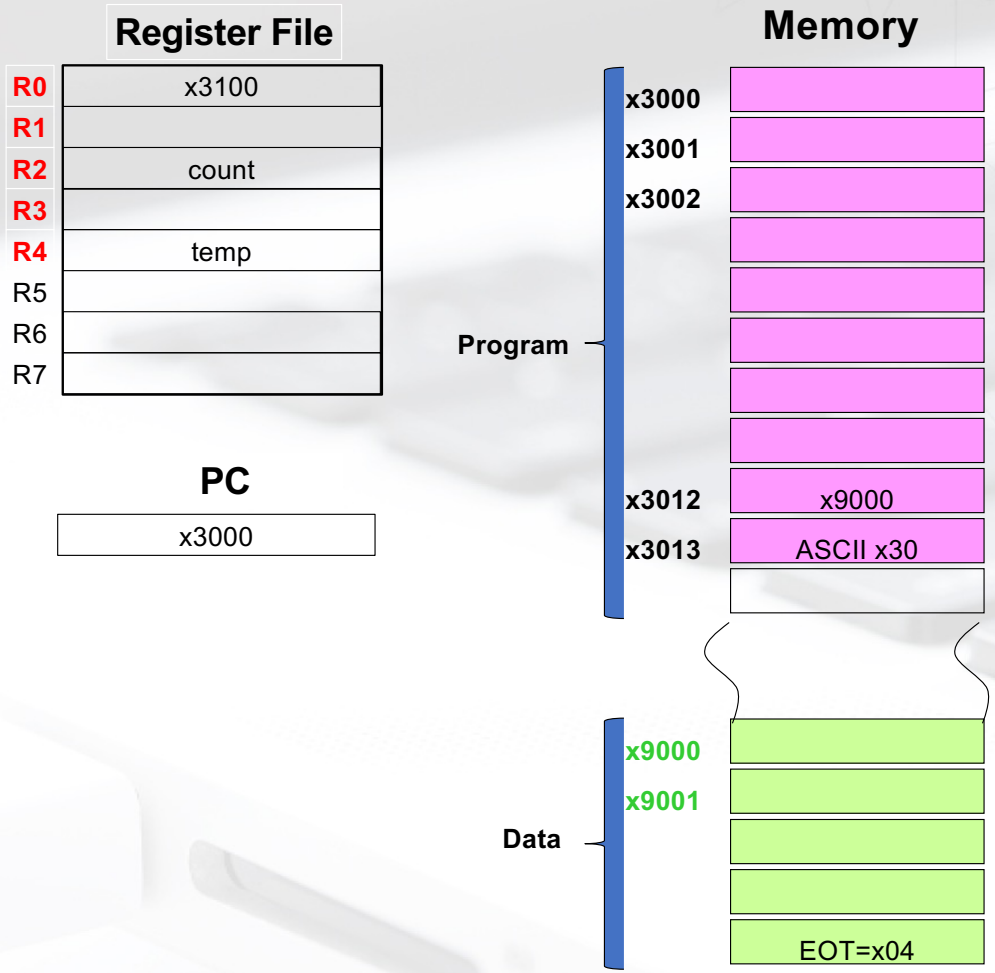
R0: hold the character that is being counted （typed from keyboard）

R1: hold, in turn, each character that we get from the file being examined

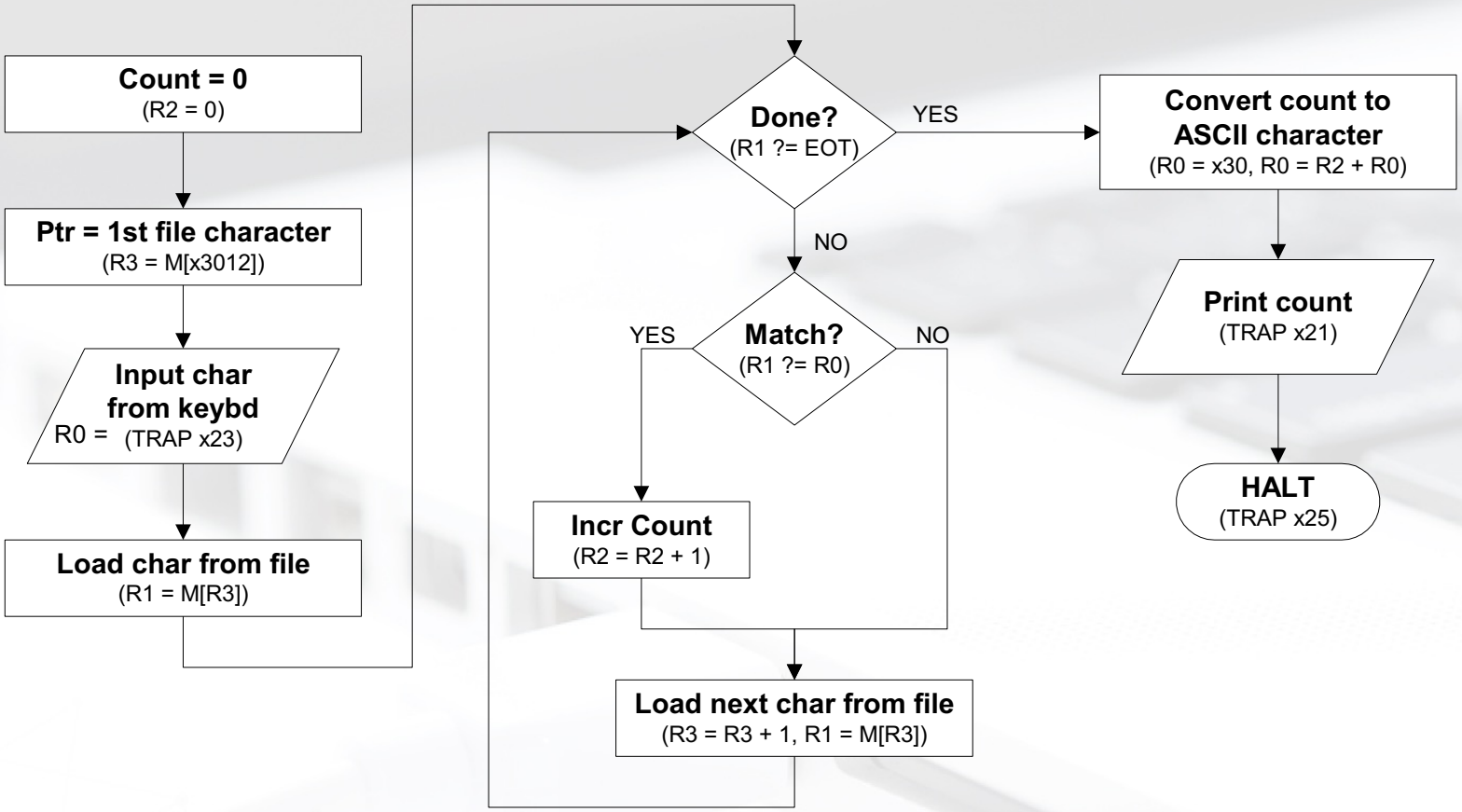
R2: keep track of the number of occurrences

R3: pointer(Ptr), at first, M[x3012]=x9000

R4: temp, checking R4= R1-ASCII(EOT)



Flow Chart


Count = 0
(R2 = 0)
Ptr = 1st file character
(R3 = M[x3012])
Input char
from keybd
R0 = (TRAP x23)
Load char from file
(R1 = M[R3])
Done?
(R1 ?= EOT)
YES
NO
Match?
(R1 ?= R0)
YES
NO
Incr Count
(R2 = R2 + 1)
Load next char from file
(R3 = R3 + 1, R1 = M[R3])
Convert count to
ASCII character
(R0 = x30, R0 = R2 + R0)
Print count
(TRAP x21)
HALT
(TRAP x25)

Program (1 of 2)

Address	Instruction	Comments
x3000	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0 (counter) AND R2,R2, #0
x3001	0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0	R3 ← M[x3012] (ptr) LD R3, x3012 (LD R3, PTR)
x3002	1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1	Input to R0 (TRAP x23) TRAP x23 (GETC)
x3003	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	R1 ← M[R3] LDR R1, R3, #0
x3004	0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0	R4 ← R1 – 4 (EOT) ADD R4,R1, #-4
x3005	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0	If Z, goto x300E BRz x300E (BRz OUTPUT)
x3006	1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1	R1 ← NOT R1 NOT R1,R1
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	R1 ← R1 + 1 ADD R1,R1,#1
X3008	0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0	R1 ← R1 + R0 ADD R1,R1,R0
x3009	0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1	If N or P, goto x300B BRnp x300B (BRnp GETCHAR)

Program (2 of 2)

Address	Instruction	Comments
x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	*R2 ← R2 + 1* *ADD R2,R2,#1*
x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	*R3 ← R3 + 1* *ADD R3,R3,#1*
x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	*R1 ← M[R3]* *LDR R1,R3,#0*
x300D	0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0	*Goto x3004* *BRnzp x3004 (BRnzp TEST)*
x300E	0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0	*R0 ← M[x3013]* *LD R0,x3013 (LD R0, ASCII)*
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0	*R0 ← R0 + R2* *ADD R0,R0,R2*
x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1	*Print R0* *TRAP x21 (OUT)*
x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	*HALT* *TRAP x25 (HALT)*
X3012	1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0	Starting Address of File (X9000)
x3013	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	*ASCII x30 ('0')*

Outline

1 LC-3 ISA Overview

2 Operate Instructions and Data Path

3 Data Movement Instructions

4 Control Instructions

5 Another Example: Counting Occurrences of a Computer

6 The Data Path Revisited

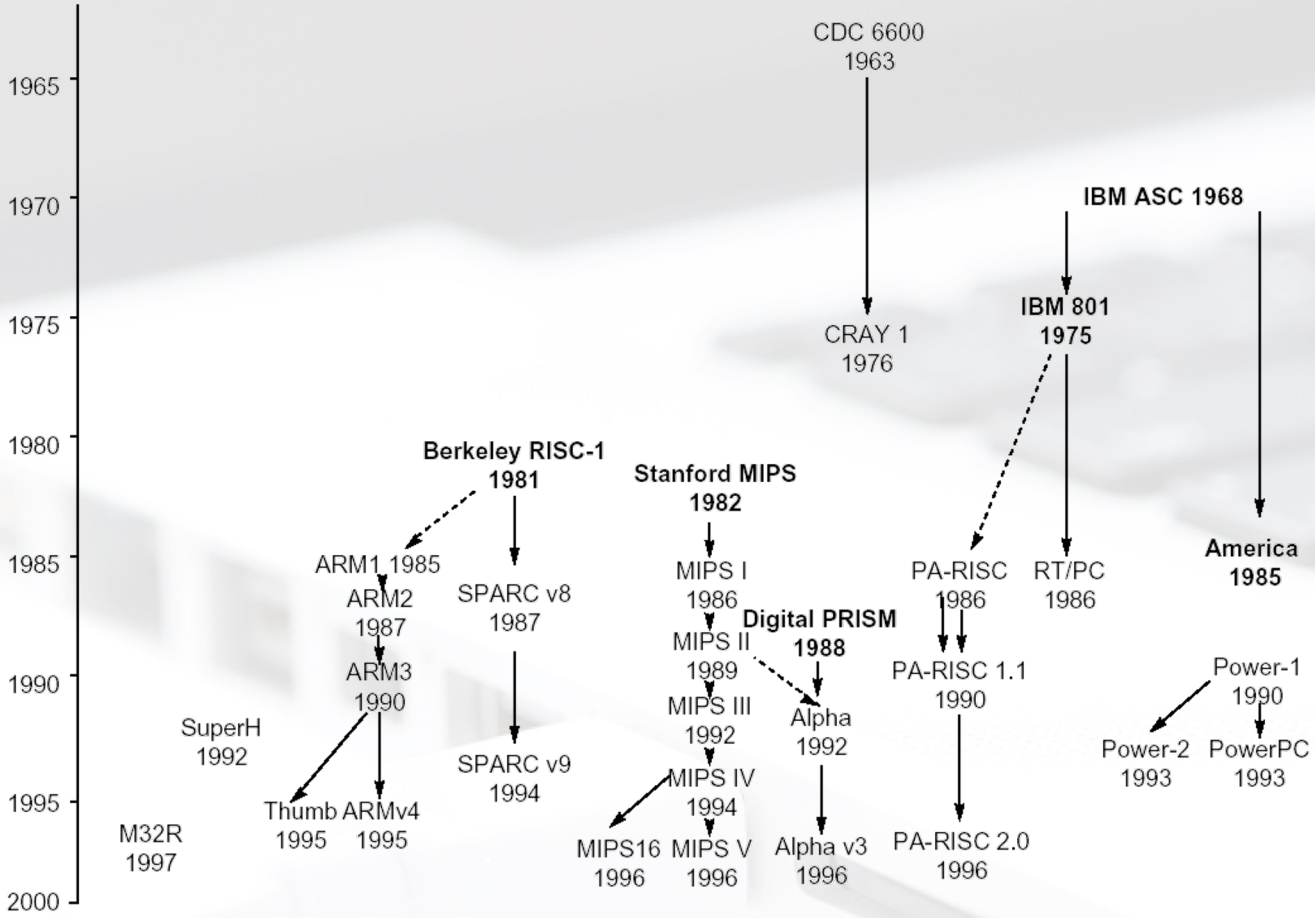
Instruction Set Architecture (ISA)

- Computer’s native operations called **instructions.**
- Job of a CPU (Central Processing Unit, aka Core): execute instructions
  - Instructions: CPU’s primitives operations
  - Instructions performed one after another in sequence
  - Each instruction does a small amount of work (a tiny part of a larger program).
  - Each instruction has an operation applied to operands, and might be used change the sequence of instruction.
- Instruction set architecture (ISA) specifies the set of commands (instructions) a computer can execute
- Hardware registers provide a few very fast variables for instructions to operate on

Instruction Set Architecture (ISA)

- The instruction set defines all the valid instructions.
- CPUs belong to "families," each implementing its own set of instructions
- CPU's particular set of instructions implements an Instruction Set Architecture (ISA)
- Examples:
  - ARM,
  - Intel x86
  - MIPS
  - RISC-V
  - IBM/Motorola PowerPC (old Mac)
  - Intel IA64,
  - ...

Instruction set architecture evolution



LC-3 ISA Overview

运算指令

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	0	1	DR			SR1			0	0	0	SR2		
ADD	0	0	0	1	DR			SR1			1	Imm5				
AND	0	1	0	1	DR			SR1			0	0	0	SR2		
AND	0	1	0	1	DR			SR1			1	Imm5				
NOT	1	0	0	1	DR			SR1			1	1	1	1	1	1
Reserved	1	1	0	1												

控制指令

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	n	z	p	PCoffset9								
JSR	0	1	0	0	1	PCoffset11										
JSRR	0	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMP	1	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RET	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
TRAP	1	1	1	1	0	0	0	0	TrapVector8							

移动数据指令

取数指令

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LD	0	0	1	0	DR			PCoffset9								
LDR	0	1	1	0	DR			BaseR			PCoffset6					
LDI	1	0	1	0	DR			PCoffset9								
LEA	1	1	1	0	DR			PCoffset9								

存数指令

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ST	0	0	1	1	SR			PCoffset9								
STR	0	1	1	1	SR			BaseR			PCoffset6					
STI	1	0	1	1	SR			PCoffset9								

Instruction Set Architecture (ISA) vs. Finite State Automata

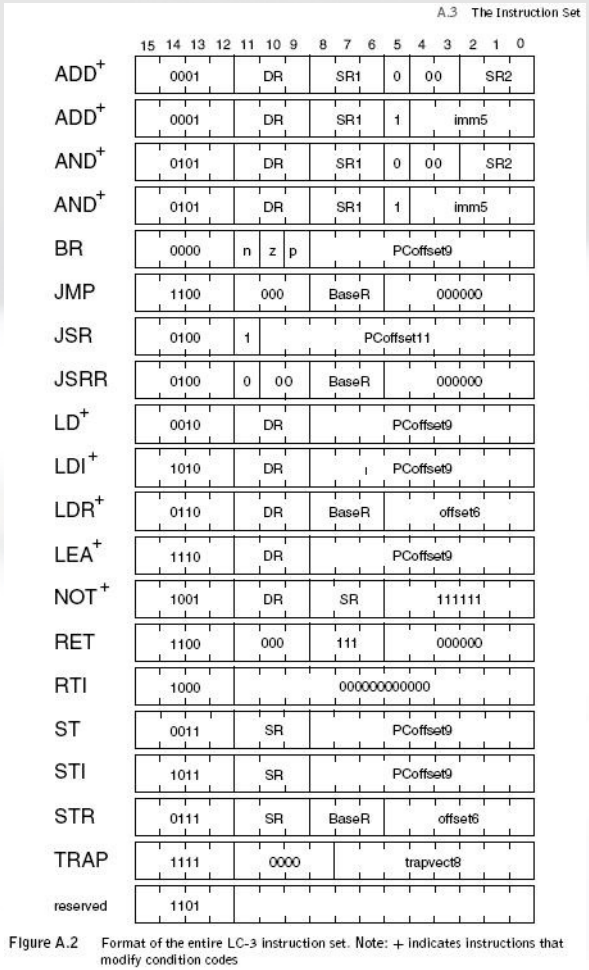


Figure A.2 Format of the entire LC-3 instruction set. Note: + indicates instructions that modify condition codes

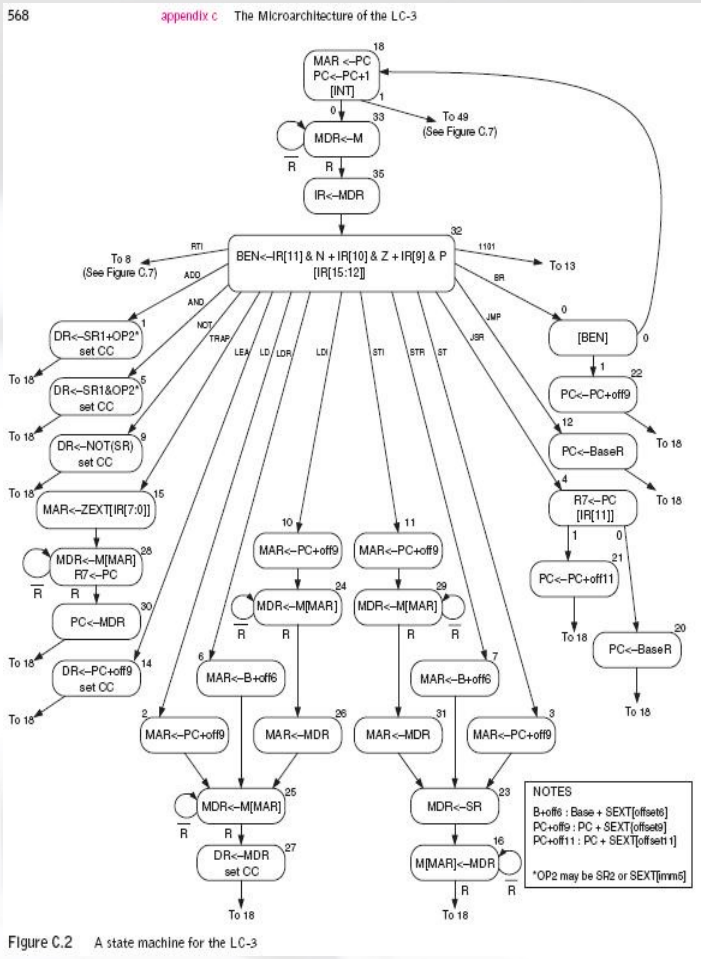
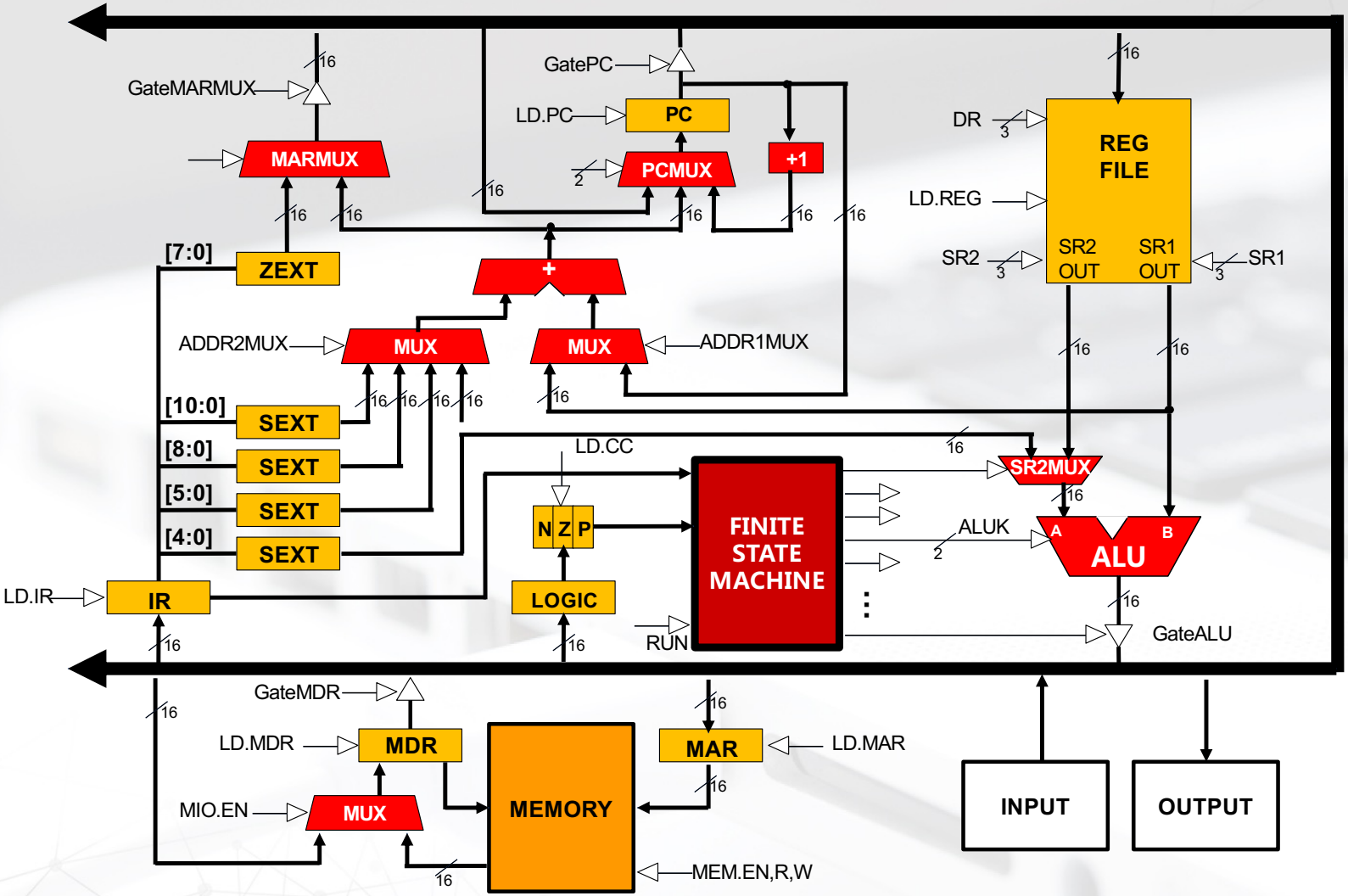


Figure C.2 A state machine for the LC-3

LC-3 Data Path



Basic Components of the Data Path

The Global bus

- special set of wires that carry a 16-bit signal to many components
- inputs to the bus are "tri-state devices," that only place a signal on the bus when they are enabled
- only one (16-bit) signal should be enabled at any time
  - control unit decides which signal "drives" the bus
- any number of components can read the bus
  - register only captures bus data if it is write-enabled by the control unit

Memory

- Control and data registers for memory and I/O devices
- memory: MAR, MDR (also control signal for read/write)

Basic Components of the Data Path

The ALU

- **Accepts inputs from register file and from sign-extended bits from IR (immediate field).**
- **Output goes to bus.**
  - used by condition code logic, register file, memory

The Register File

- **Two read addresses (SR1, SR2), one write address (DR)**
- **Input from bus**
  - result of ALU operation or memory read
- **Two 16-bit outputs**
  - used by ALU, PC, memory address
  - data for store instructions passes through ALU

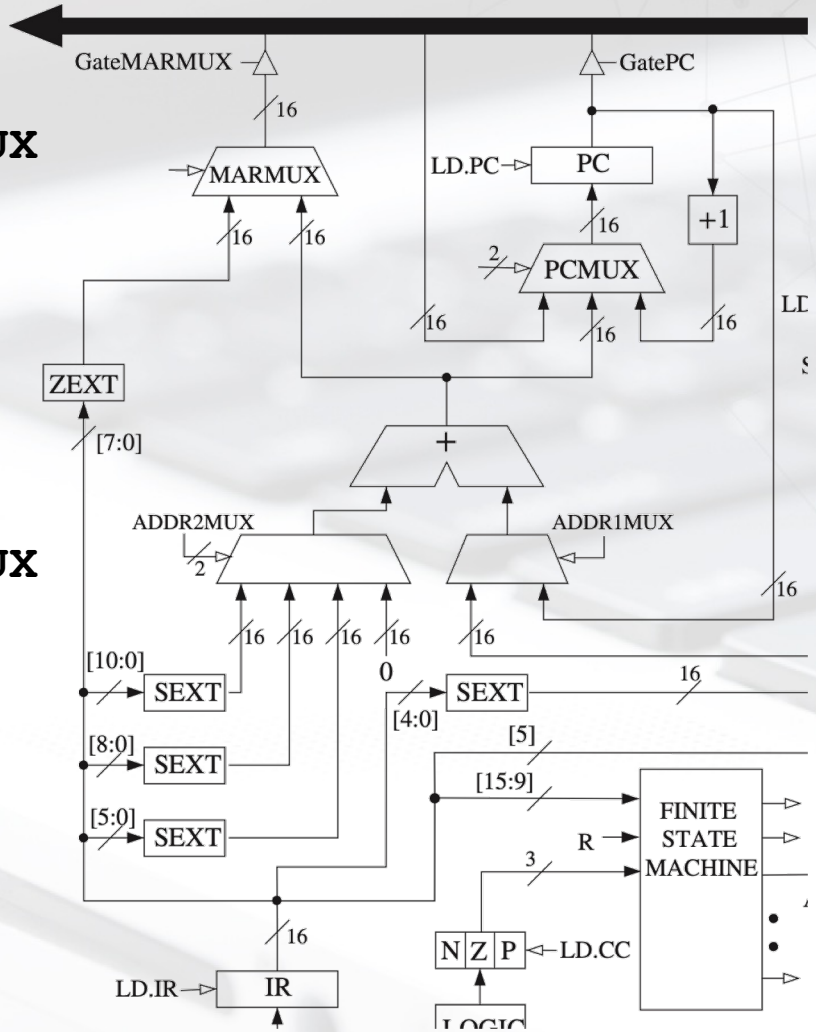
Basic Components of the Data Path

The PC and PCMUX

- **Three inputs to PC, controlled by PCMUX**
  1. PC+1 – FETCH stage
  2. Address adder – BR, JMP
  3. bus – TRAP (discussed later)

The MAR and MARMUX

- **Two inputs to MAR, controlled by MARMUX**
  1. Address adder – LD/ST, LDR/STR
  2. Zero-extended IR[7:0] -- TRAP (discussed later)



Basic Components of the Data Path

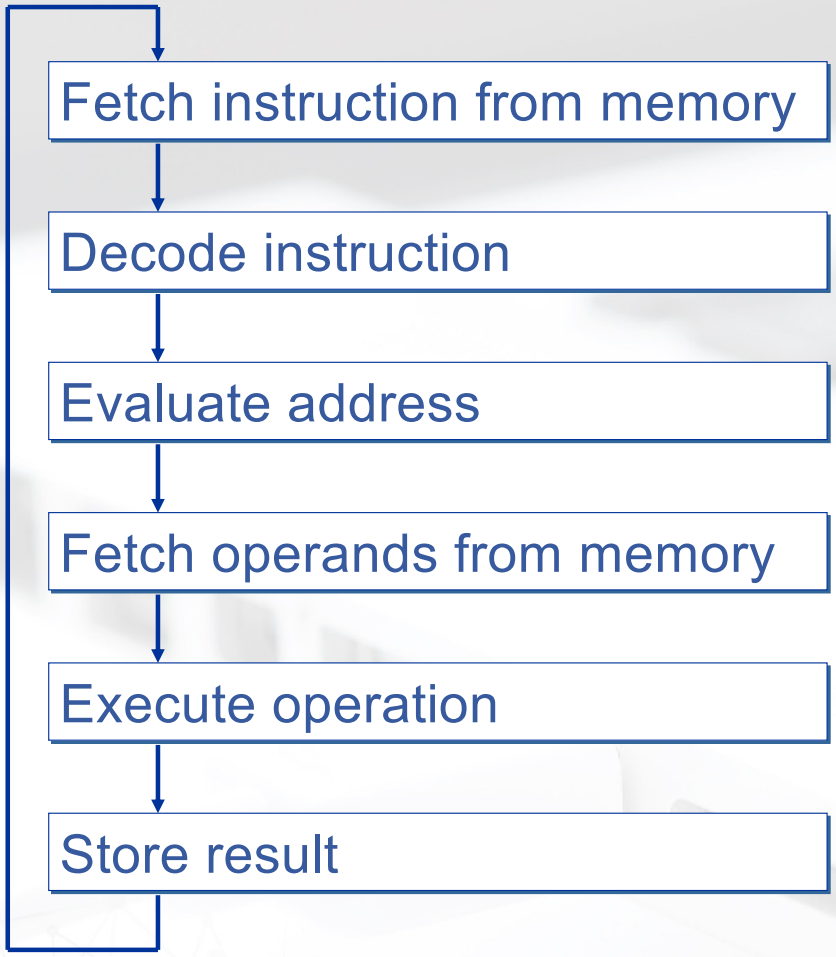
Condition Code Logic

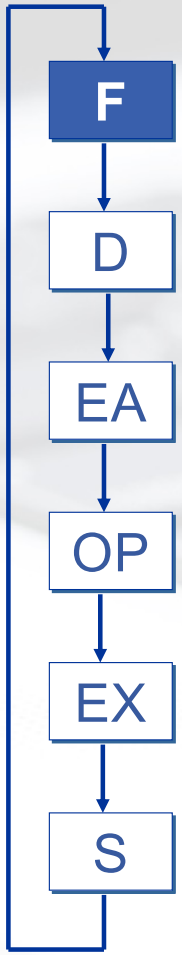
- **Looks at value on bus and generates N, Z, P signals**
- **Registers set only when control unit enables them (LD.CC)**
  - only certain instructions set the codes (ADD, AND, NOT, LD, LDI, LDR, LEA)

Control Unit – Finite State Machine

- **On each machine cycle, changes control signals for next phase of instruction processing**
  - who drives the bus? (GatePC, GateALU, …)
  - which registers are write enabled? (LD.IR, LD.REG, …)
  - which operation should ALU perform? (ALUK)
  - …
- **Logic includes decoder for opcode, etc.**

The Instruction Cycle Specific to the LC-3





5.14 The LC-3 does not have an opcode for the logical function OR. That is, there is no instruction in the LC-3 ISA that performs the OR operation. However, we can write a sequence of instructions to implement the OR operation. The following four-instruction sequence performs the OR of the contents of register 1 and register 2 and puts the result in register 3. Fill in the two missing instructions so that the four-instruction sequence will do the job.

```
(1): 1001 100 001 111111
(2):
(3): 0101 110 100 000 101
(4):
```

5.22 The PC contains x3010. The following memory locations contain values as shown:

x3050:	x70A4
x70A2:	x70A3
x70A3:	xFFFF
x70A4:	x123B

The following three LC-3 instructions are then executed, causing a value to be loaded into R6. What is that value?

x3010	1110 0110 0011 1111
x3011	0110 1000 1100 0000
x3012	0110 1101 0000 0000

We could replace the three-instruction sequence with a single instruction. What is it?

5.23 Suppose the following LC-3 program is loaded into memory starting at location x30FF:

```
x30FF    1110 0010 0000 0001
x3100    0110 0100 0100 0010
x3101    1111 0000 0010 0101
x3102    0001 0100 0100 0001
x3103    0001 0100 1000 0010
```

If the program is executed, what is the value in R2 at the end of execution?

5.28 It is the case that we REALLY don't need to have load indirect (1010) and store indirect (1011) instructions. We can accomplish the same results using other instruction sequences instead of using these instructions. Replace the store indirect (1011) instruction in the code below with whatever instructions are necessary to perform the same function.

```
x3000    0010 0000 0000 0010
x3001    1011 0000 0000 0010
x3002    1111 0000 0010 0101
x3003    0000 0000 0100 1000
x3004    1111 0011 1111 1111
```

5.31 The figure below shows a snapshot of the eight registers of the LC-3 before and after the instruction at location x1000 is executed. Fill in the bits of the instruction at location x1000.

	BEFORE		AFTER
R0	x0000	R0	x0000
R1	x1111	R1	x1111
R2	x2222	R2	x2222
R3	x3333	R3	x3333
R4	x4444	R4	x4444
R5	x5555	R5	xFFF8
R6	x6666	R6	x6666
R7	x7777	R7	x7777

0x1000 :	0	0	0	1												