Previously: An Assembly Language Program

```
01 ;
02 ; Program to multiply a number by the constant 6
03 ;
04         .ORIG   x3050
05         LD      R1, SIX
06         LD      R2, NUMBER
07         AND     R3, R3, #0   ; Clear R3.  It will
08                              ; contain the product.
09 ; The inner loop
0A ;
0B AGAIN   ADD     R3, R3, R2
0C         ADD     R1, R1, #-1  ; R1 keeps track of
0D         BRp     AGAIN        ; the iteration.
0E ;
0F         HALT
10 ;
11 NUMBER  .BLKW   1
12 SIX     .FILL   x0006
13 ;
           .END
```

7.2 An LC-3 assembly language program contains the instruction:

```
ASCII   LD  R1, ASCII
```

The symbol table entry for ASCII is x4F08. If this instruction is executed during the running of the program, what will be contained in R1 immediately after the instruction is executed?

7.10 The following program fragment has an error in it. Identify the error and explain how to fix it.

```
        ADD     R3, R3, #30
        ST      R3, A
        HALT
A       .FILL   #0
```

Will this error be detected when this code is assembled or when this code is run on the LC-3?

7.13 The following program adds the values stored in memory locations A, B, and C and stores the result into memory. There are two errors in the code. For each, describe the error and indicate whether it will be detected at assembly time or at run time.

```
Line No.
1                .ORIG  x3000
2        ONE     LD  R0, A
3                ADD R1, R1, R0
4        TWO     LD  R0, B
5                ADD R1, R1, R0
6        THREE   LD  R0, C
7                ADD R1, R1, R0
8                ST  R1, SUM
9                TRAP  x25
10       A       .FILL x0001
11       B       .FILL x0002
12       C       .FILL x0003
13       D       .FILL x0004
14               .END
```

7.24 We want the following program fragment to shift R3 to the left by four bits, but it has an error in it. Identify the error and explain how to fix it.

```
        .ORIG  x3000
        AND    R2, R2, #0
        ADD    R2, R2, #4
LOOP    BRz    DONE
        ADD    R2, R2, #-1
        ADD    R3, R3, R3
        BR     LOOP
DONE    HALT
        .END
```

6.18 The LC-3 has no Divide instruction. A programmer needing to divide two numbers would have to write a routine to handle it. Show the systematic decomposition of the process of dividing two positive integers. Write an LC-3 machine language program starting at location x3000 that divides the number in memory location x4000 by the number in memory location x4001 and stores the quotient at x5000 and the remainder at x500l.

★5.50 Three instructions all construct an address by sign-extending the low nine bits of the instruction and adding it to the incremented PC.

The Conditional Branch	0000	111	xxxxxxxxx
The Load Effective Address	1110	111	xxxxxxxxx
The LD Instruction	0010	111	xxxxxxxxx

The xxxxxxxxx represents the nine-bit offset that is sign-extended.

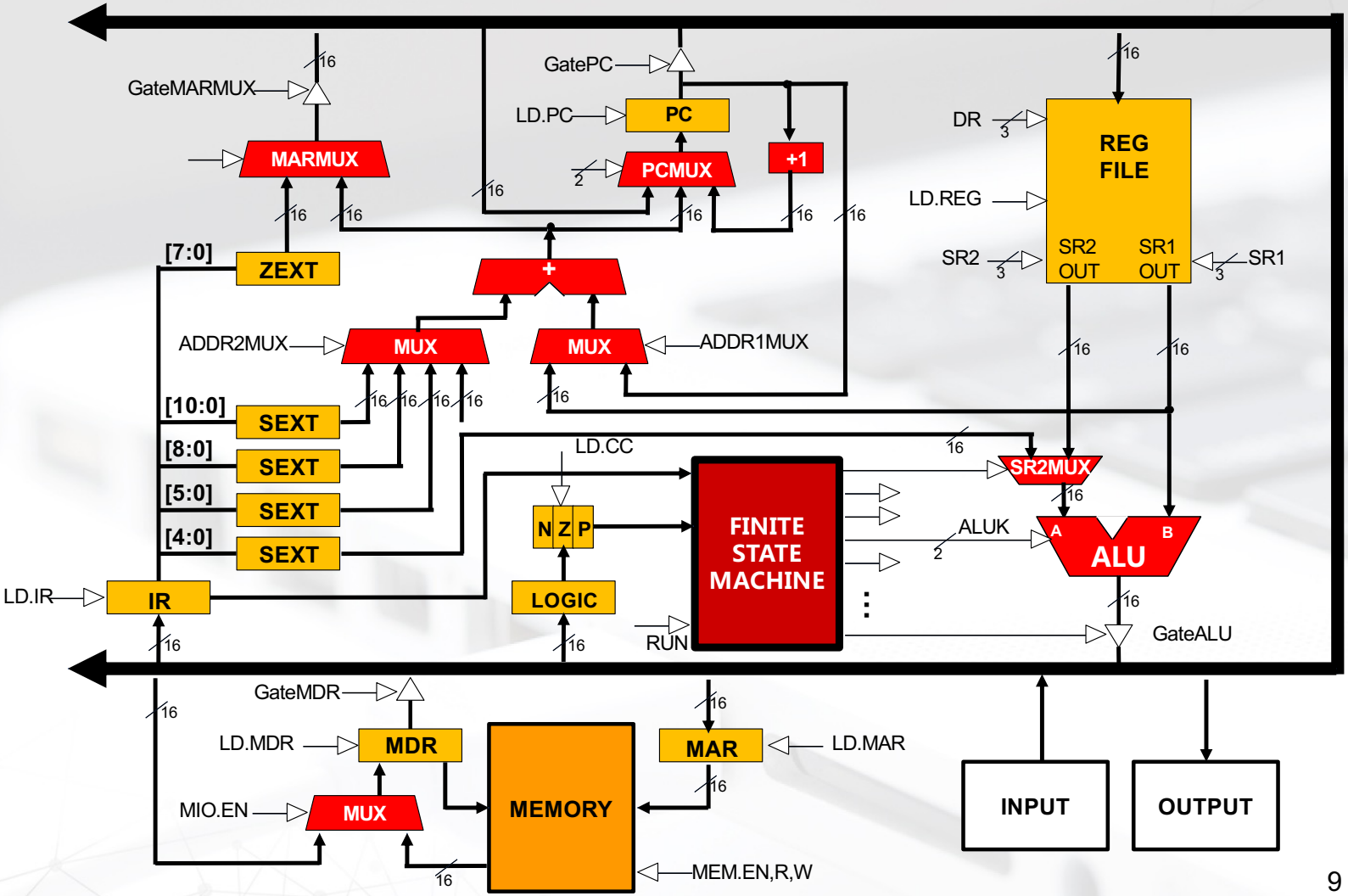
Where does the LC-3 microarchitecture put the result of adding the nine-bit sign-extended offset to the incremented PC?

★6.23 The PC is loaded with x3000, and the instruction at address x3000 is executed. In fact, execution continues and four more instructions are executed. The table below contains the contents of various registers at the end of execution for each of the five (total) instructions.
Your job: Complete the table.

	PC	MAR	MDR	IR	R0	R1
Before execution starts	x3000	——	——	——	x0000	x0000
After the first finishes			xB333	x2005		
After the 2nd finishes				x0601		
After the 3rd finishes			x1___			x0001
After the 4th finishes			x1___		x6666	
After the 5th finishes				x0BFC		

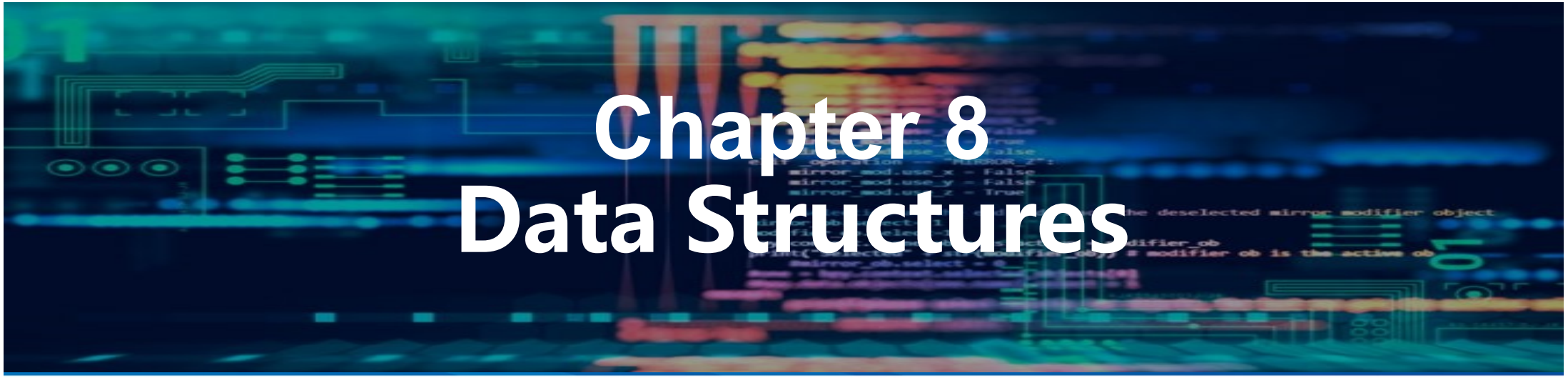
Let's start execution again, starting with PC = x3000. First, we re-initialize R0 and R1 to 0, and set a breakpoint at x3004. We press RUN eleven times, and each time the program executes until the breakpoint. What are the final values of R0 and R1?

LC-3 Data Path



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 8
Data Structures

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

Outline

Outline

Abstract Data Types： Data Structures

- Up to now, we have processed a single value
  - an integer
  - a floating point number, or
  - an ASCII character
- The information in the real world is far more complex than simple, single numbers. We call these complex items of information **abstract data types**, or more colloquially **data structures**，far more complex than simple, single numbers. E.g.
  - a company's organization chart
  - a list of items arranged in alphabetical order
- In this chapter, we will study three abstract data types:
  - stacks
  - queues
  - and character strings

Abstract Data Types： Data Structures

- We will write programs to solve problems that require expressing information according to its structure.
- Before we get to **stacks, queues, and character strings**, however, we introduce a new concept that will prove very useful in manipulating data structures: **subroutines**, or what is also called **functions**.

Outline

Subroutines

- A subroutine is a program fragment that. . .
  - Resides in user space (*i.e*, not in OS)
  - Performs a well-defined task
  - Is invoked (called) multiple times by a user program
  - Returns control to the calling program when finished
- Virtues
  - Reuse code without re-typing it (and debugging it!)
  - Divide task into parts (or among multiple programmers)
  - Use vendor-supplied library of useful routines that one software engineer writes a program that requires such fragments and another software engineer writes the fragments.
    - math library
    - square root, sine, and arctangent, etc.
- In C language, called function; In other languages, called procedures, subroutines, methods ...

A simple illustration of a part of a program

```
01; Service Routine for Keyboard Input
02          .ORIG    x04A0              ;System call starting address
03
04;START    ST       R7,SaveR7          ;Save the linkage back to the
05;                                     ;program?
06          ST       R1,SaveR1          ;Save the values in the registers
07          ST       R2,SaveR2          ;that are used so that they can
08          ST       R3,SaveR3          ;be restored before RET
09
10;Output Newline on CRT
11          LD       R2,Newline
12 L1       LDI      R3,DSR             ;Check DDR—is it free?
13          BRzp     L1                 ;Loop until monitor is ready
14          STI      R2,DDR             ;Move cursor to new clean line
15;
16;Output "Input a character"
17          LEA      R1,Prompt          ;Prompt is starting address
18                                     ;of prompt string
19 Loop     LDR      R0,R1,#0           ;Get next prompt character
20          BRzp     Input              ;Check for end of prompt string
21 L2       LDI      R3,DSR
22          BRzp     L2
23          STI      R0,DDR             ;Write next character of prompt
24                                     ;string
25          ADD      R1,R1,#1           ;Increment prompt point
26          BRnzp    Loop
```

```
Label       LDI      R3,DSR
            BRzp     Label
            STI      Reg,DDR
```

A simple illustration of a part of a program

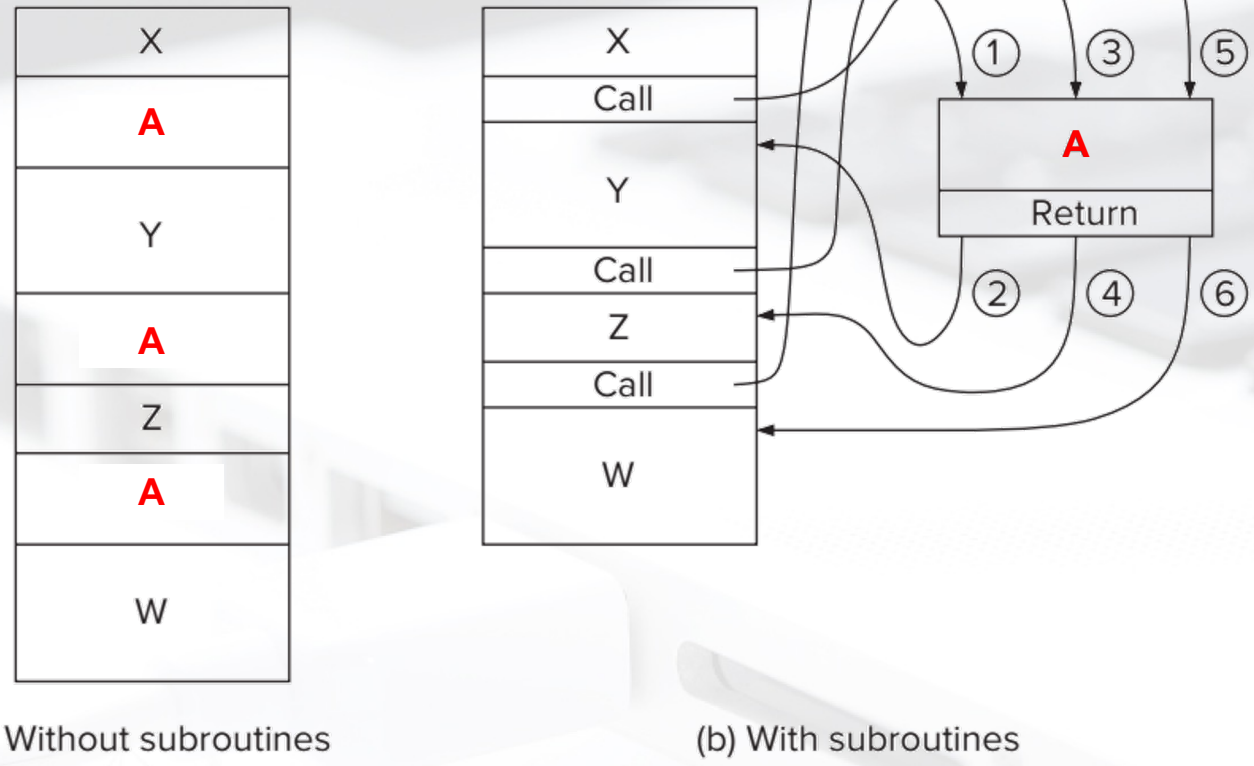
```
27;Input a character from KB
28 Input  LDI     R3,KBSR         ;Has a character been typed?
29        BRzp    Input
30        LDI     R0,KBDR         ;Load it into R0
31
32;Echo the character on CRT
33 L3     LDI     R3,DSR
34        BRzp    L3
35        STI     R0,DDR          ;Echo input character to the
36                                ;monitor
37;Output Newline on CRT
38 L4     LDI     R3,DSR          ;Check CRTDR—is it free?
39        BRzp    L4
40        STI     R2,DDR          ;Move cursor to new clean line
41
42;Restore
43        LD      R1,SaveR1       ;Service routine done, restore
44        LD      R2,SaveR2       ;original values in registers.
45        LD      R3,SaveR3       ;
46;       LD      R7,SaveR7       ;Restore linkage back
47;                               ;prior to RET?
48        RET                     ;Return to calling program
```

```
Label     LDI     R3,DSR
          BRzp    Label
          STI     Reg,DDR
```

A simple illustration of a part of a program

```
49;Memory for registers saved
50;       SaveR7   .FILL x0000
51        SaveR1   .FILL x0000
52        SaveR2   .FILL x0000
53        SaveR3   .FILL x0000
54
55        DSR      .FILL xF3FC
56        DDR      .FILL xF3FF
57        KBSR     .FILL xF400
58        KBDR     .FILL xF401
59 ;
59        Newline  .FILL x000A      ;ASCII code for newline
60        Prompt   .STRINGZ "Input a character>"
61                 .END
```

The Call/Return Mechanism

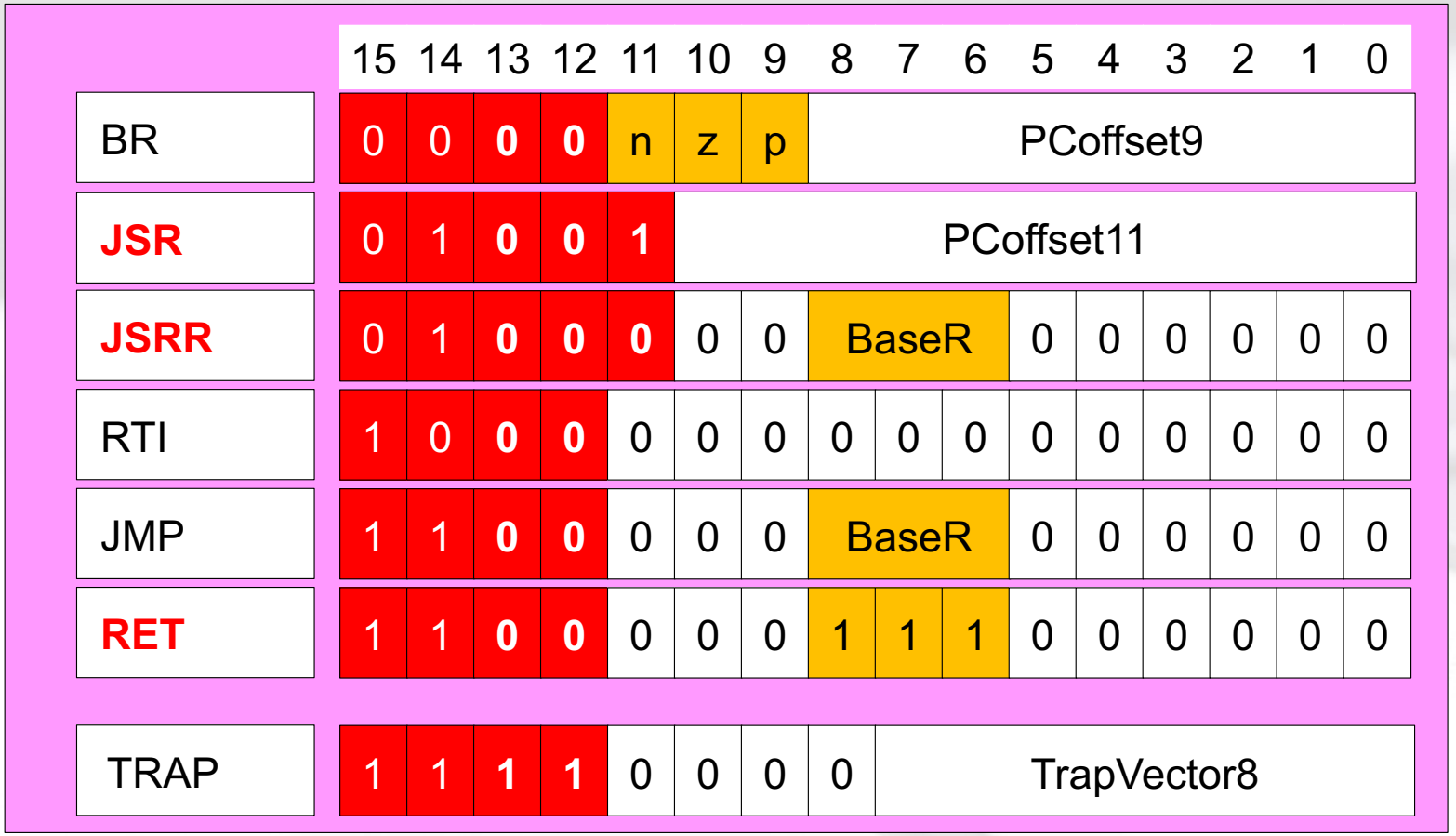


(a) Without subroutines

(b) With subroutines

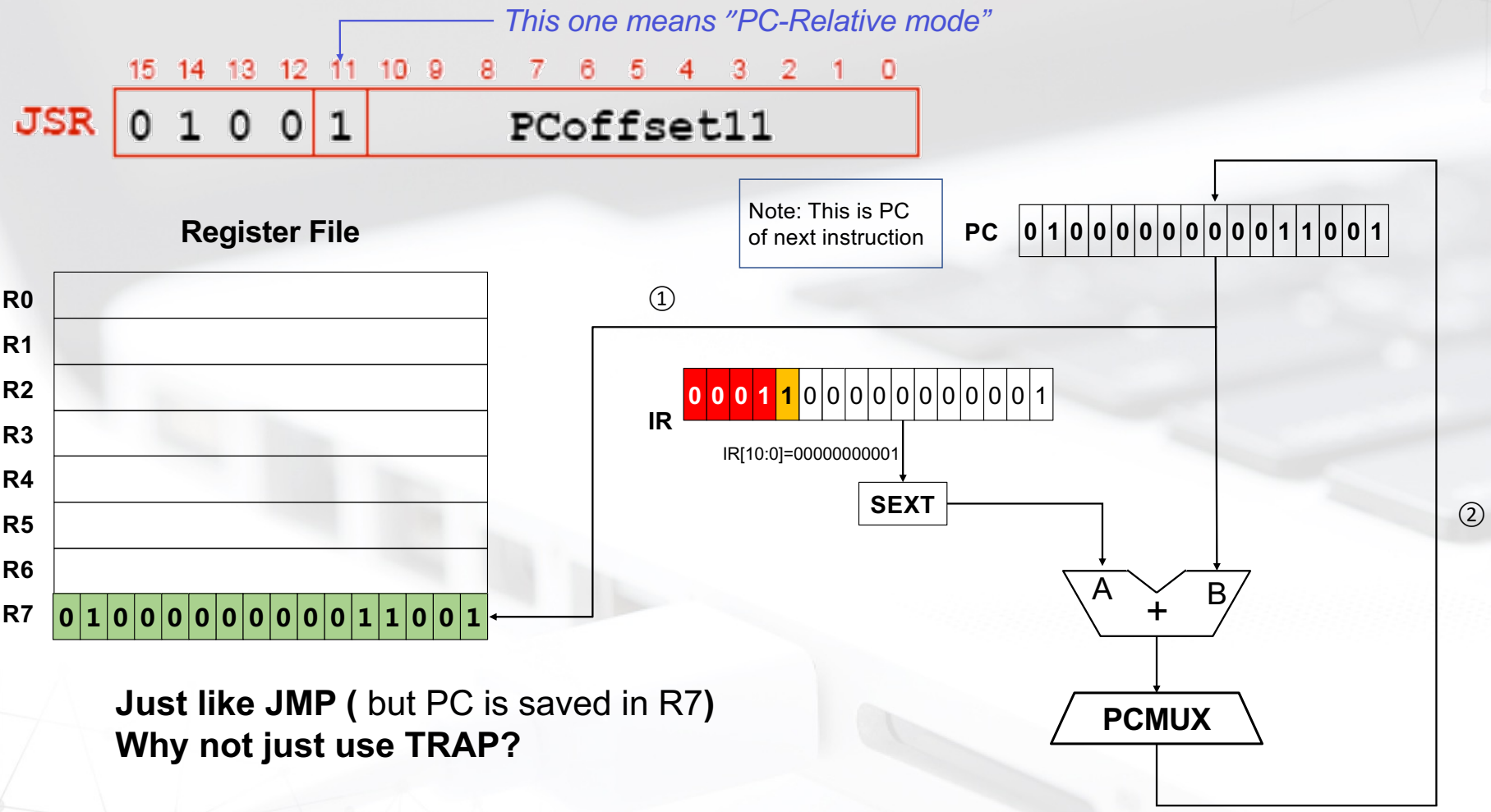
Outline

Control Instructions for Subroutines



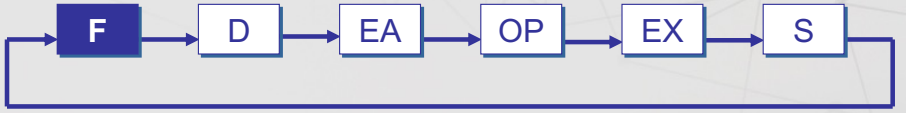
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	n	z	p	PCoffset9								
JSR	0	1	0	0	1	PCoffset11										
JSRR	0	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JMP	1	1	0	0	0	0	0	BaseR			0	0	0	0	0	0
RET	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
TRAP	1	1	1	1	0	0	0	0	TrapVector8							

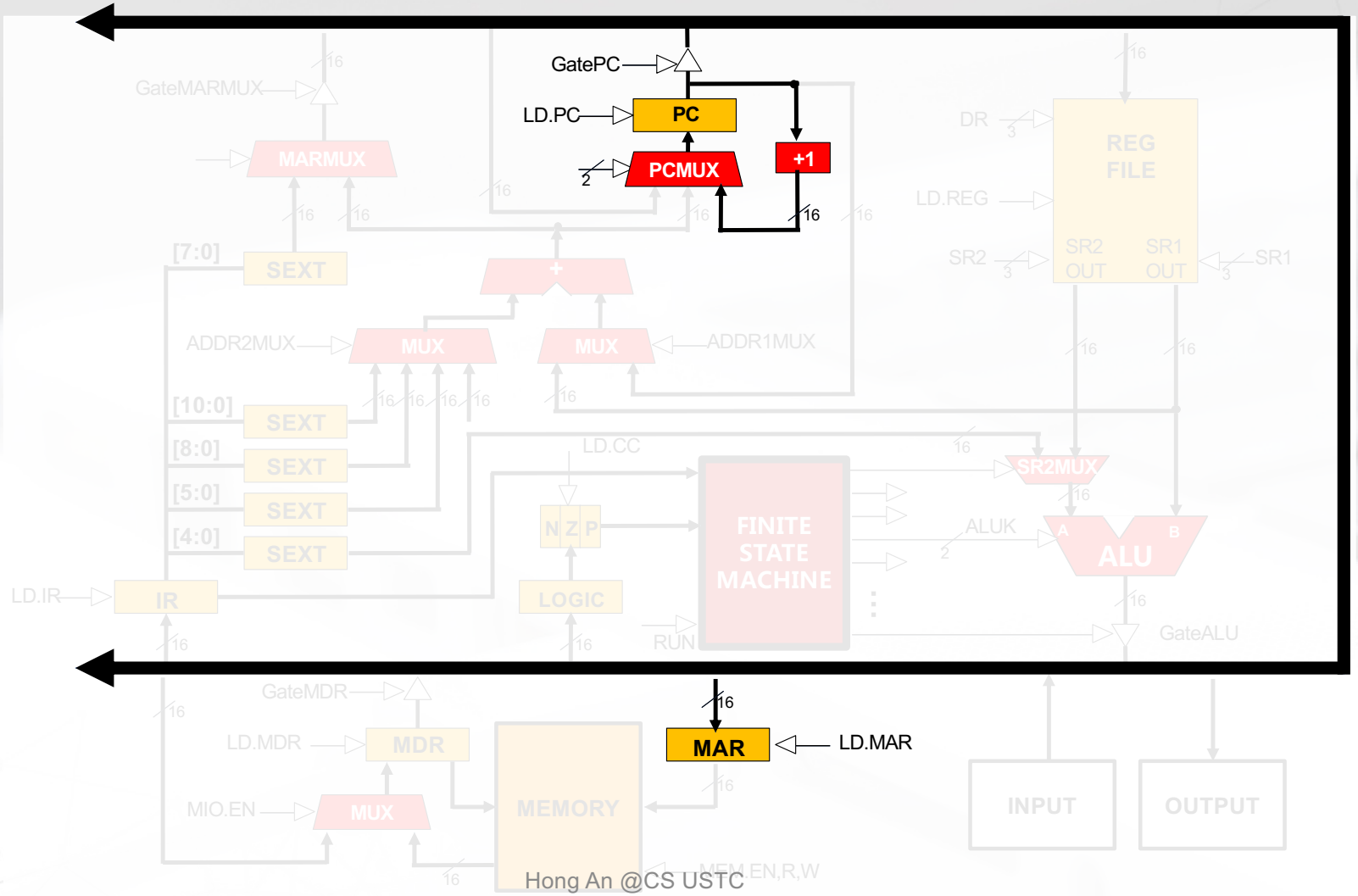
JSR (PC-Relative)



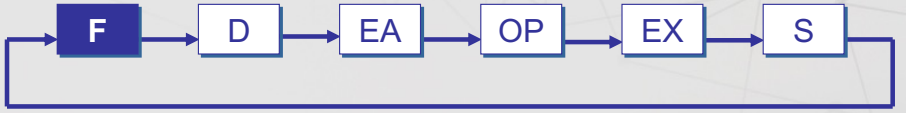
Just like JMP (but PC is saved in R7)
Why not just use TRAP?

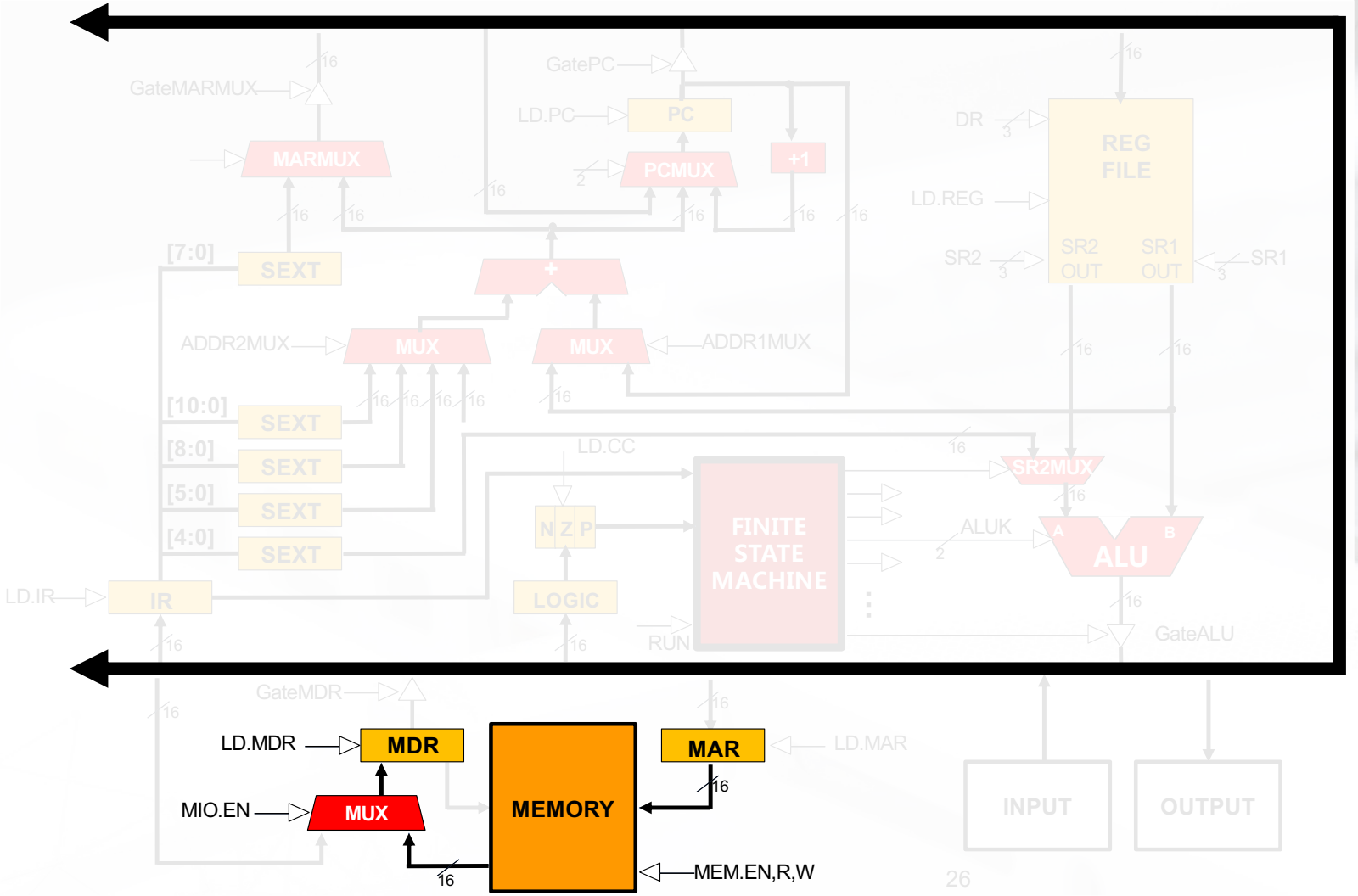
JSR (PC-Relative)



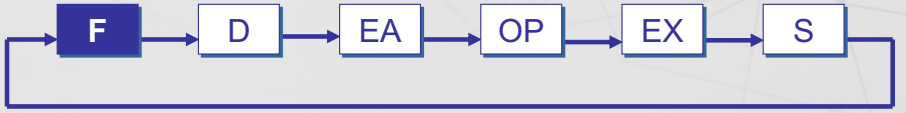


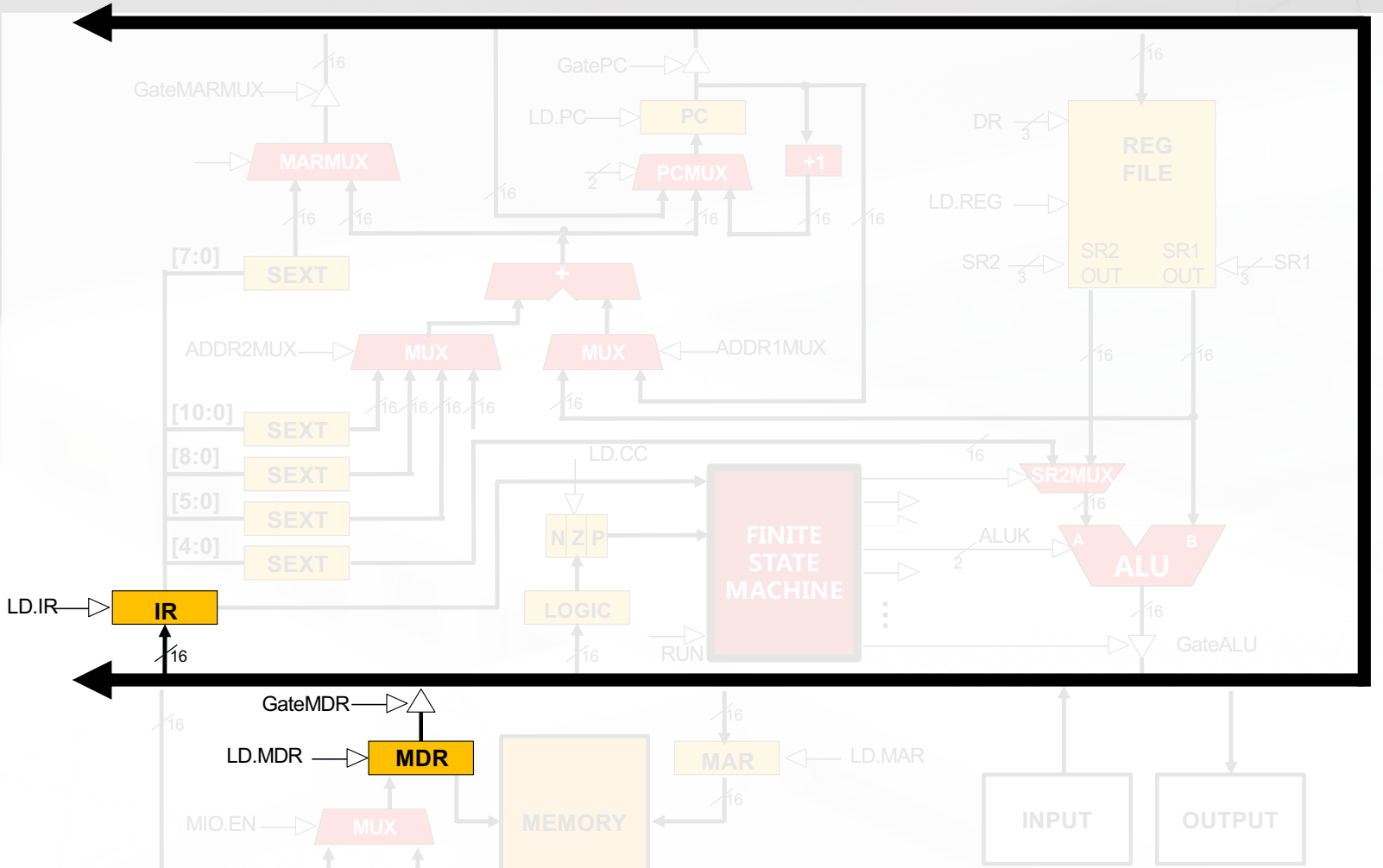
JSR (PC-Relative)



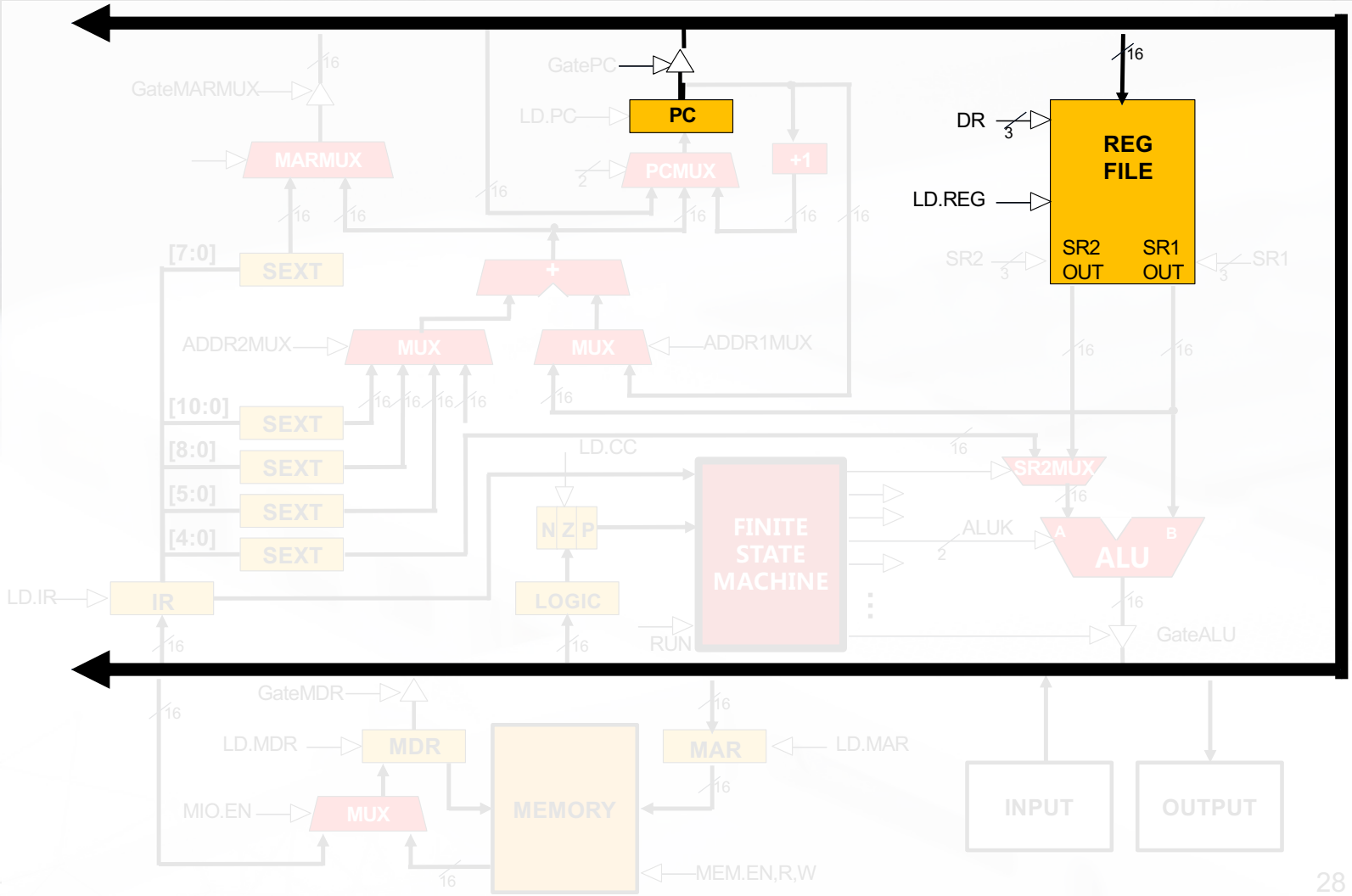


JSR (PC-Relative)

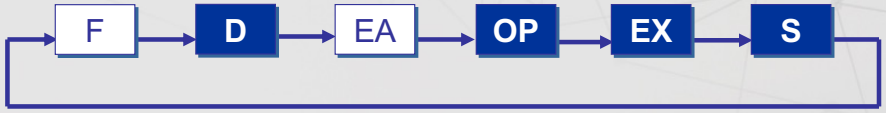


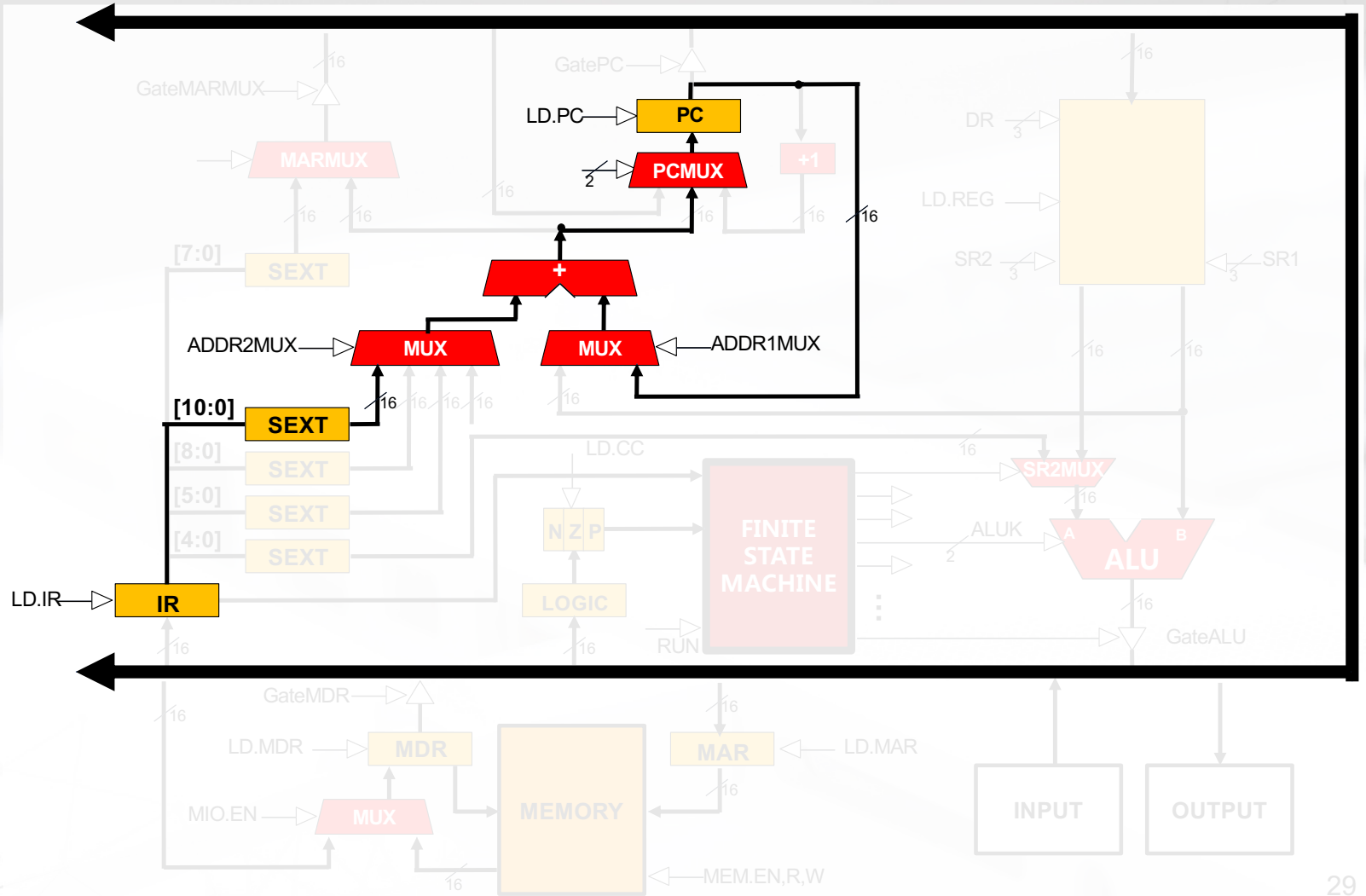


JSR (PC-Relative)

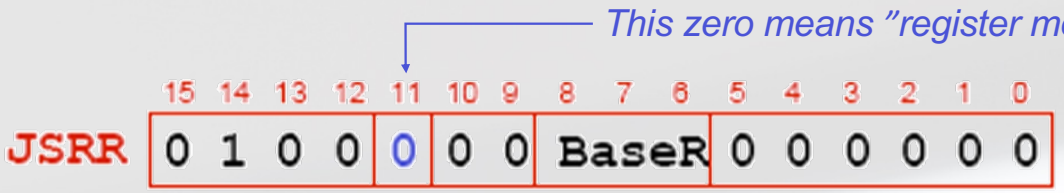


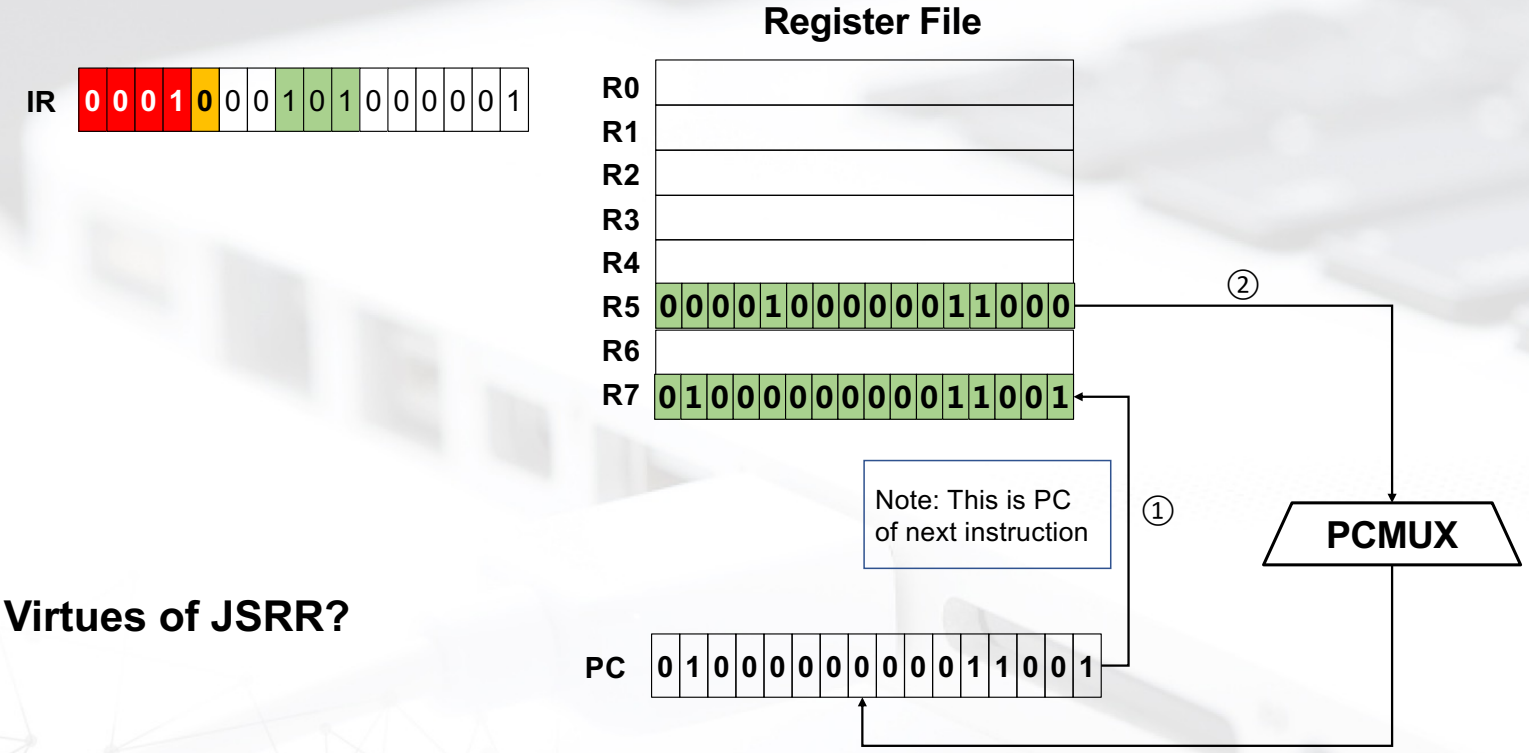
JSR (PC-Relative)





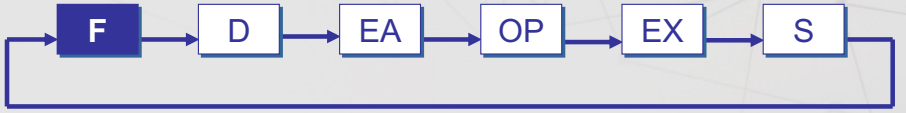
JSRR (Register)

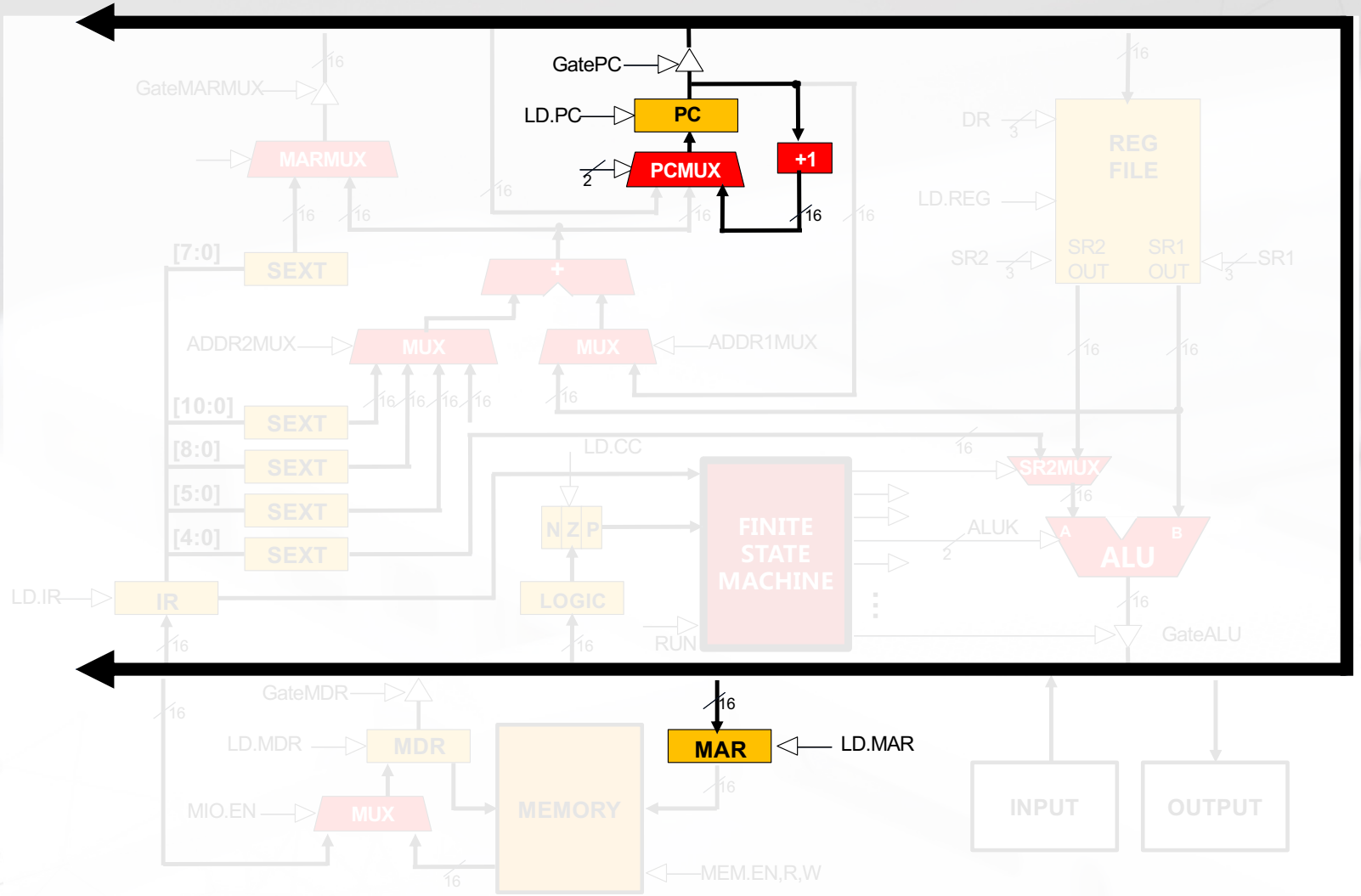




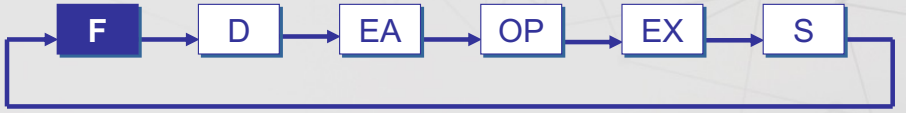
Virtues of JSRR?

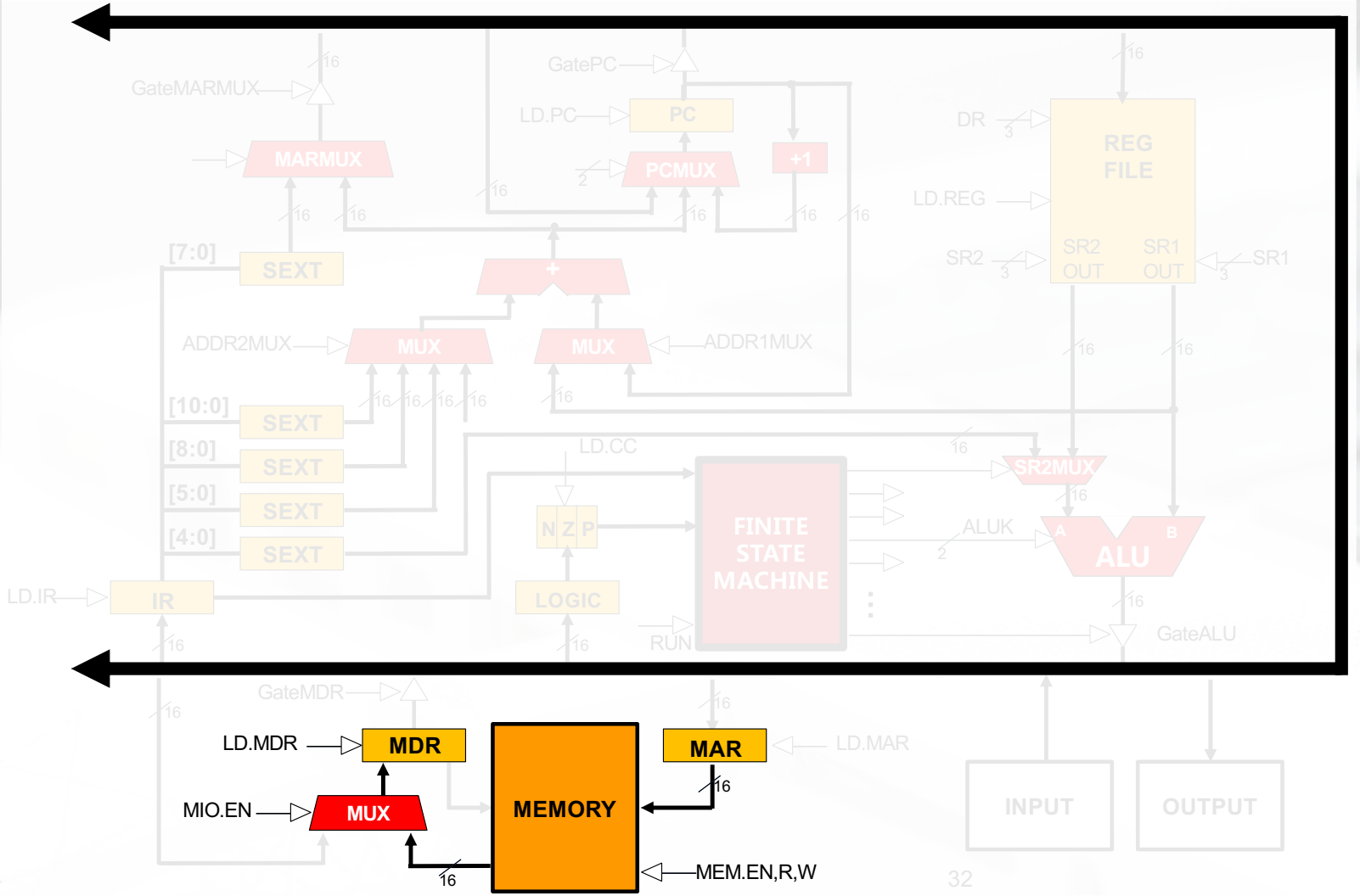
JSRR (Register)



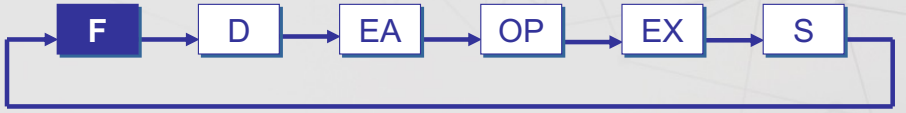


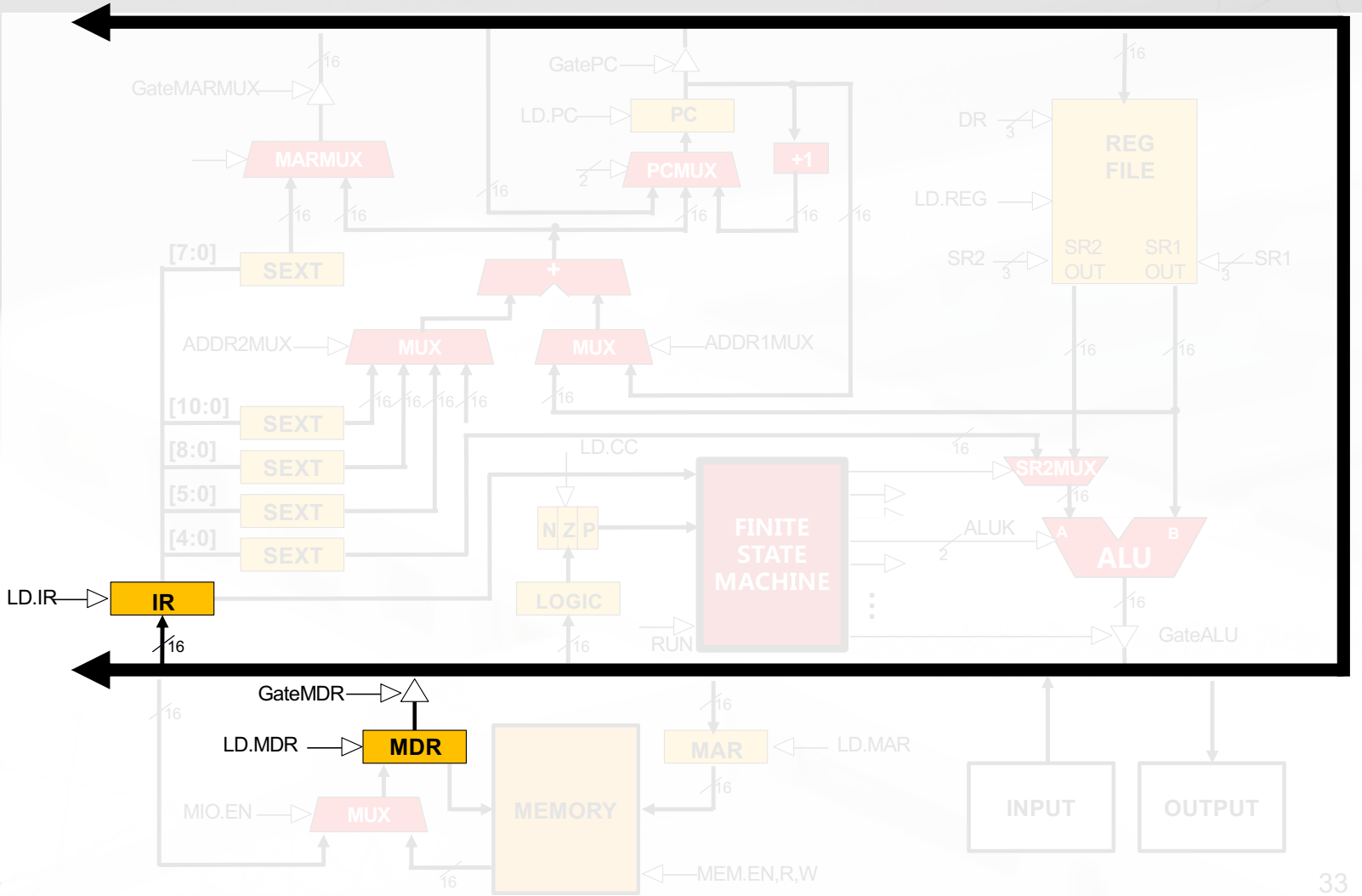
JSRR (Register)



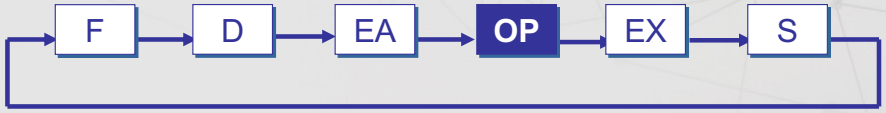


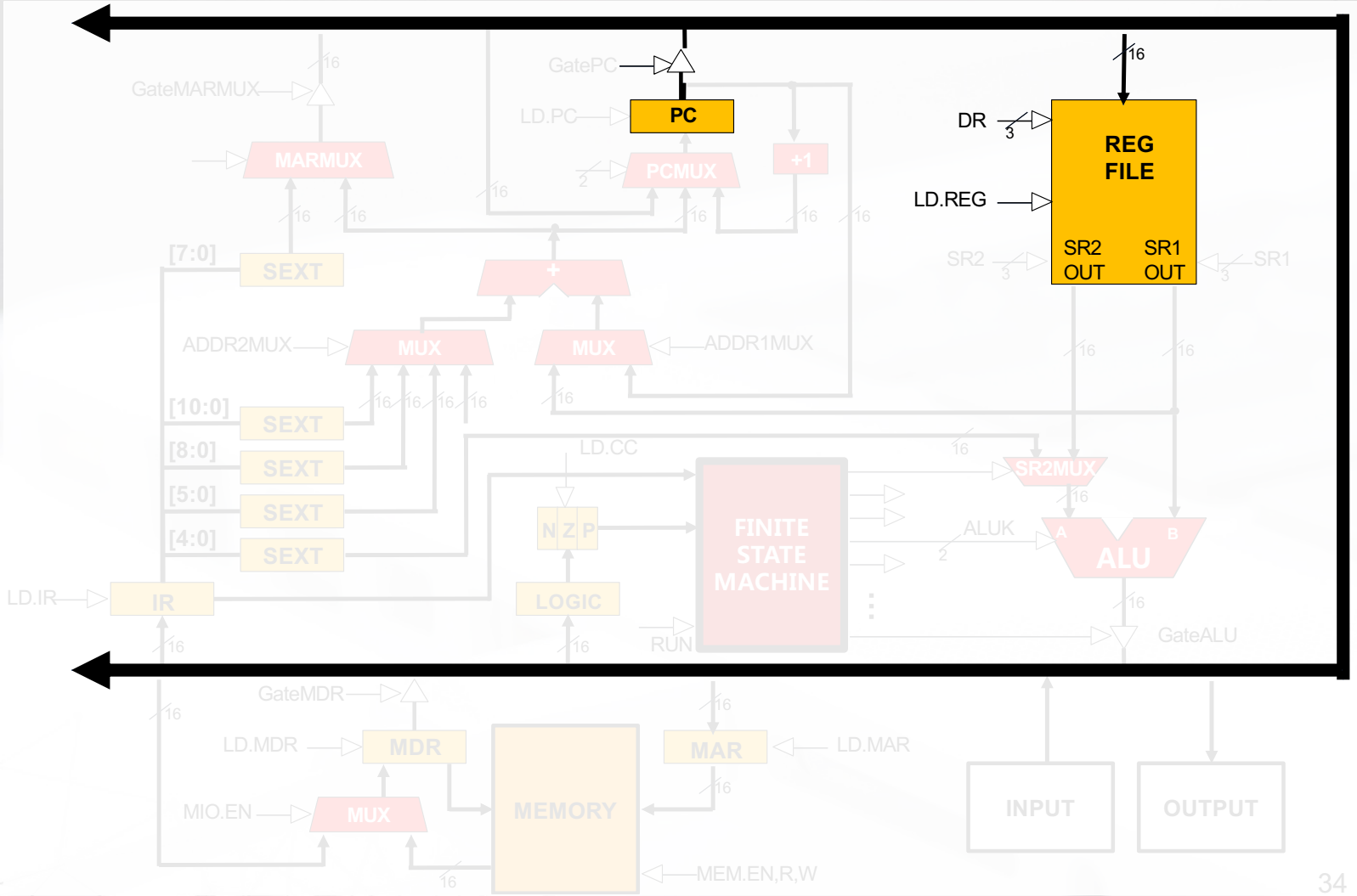
JSRR (Register)



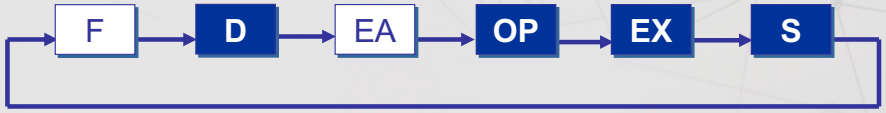


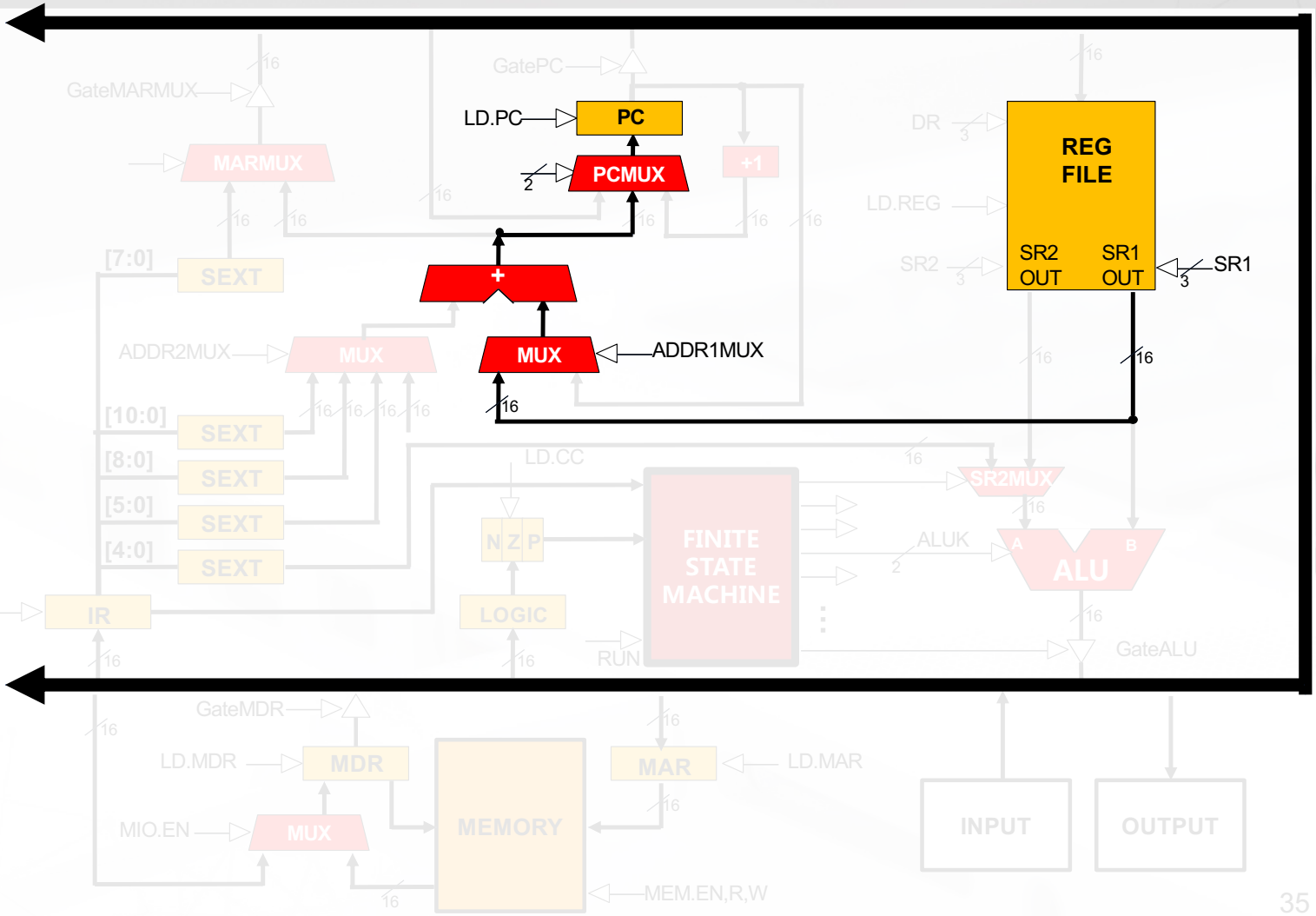
JSRR (Register)





JSRR (Register)





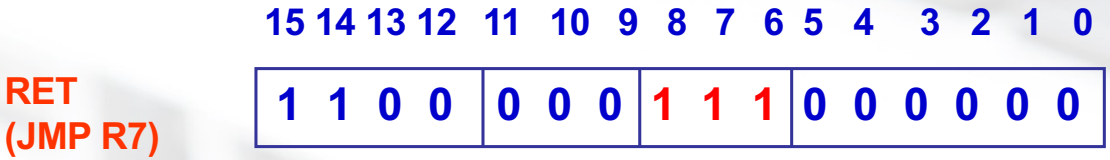
RET instruction

RET – return instruction

- **How to return**

  — Place address in R7 in PC, Return the execution to the last calling point.

- **PC ← (R7)**



Example: Negate the value in R0

```
TwosComp    NOT    R0,R0        ;flip bits
            ADD    R0,R0,#1     ;add one
            RET                 ;return to caller
```

To call from a program

```
;need to compute R4 = R1-R3
            ADD    R0,R3,#0     ;copy R3 to R0
            JSR    TwosComp     ;negate
            ADD    R4,R1,R0     ;add to R1
            ...
```

6.17 Shown below are the contents of registers before and after the LC-3 instruction at location x3210 is executed. Your job: Identify the instruction stored in x3210. *Note:* There is enough information below to uniquely specify the instruction at x3210.

	Before	After
R0:	xFF1D	xFF1D
R1:	x301C	x301C
R2:	x2F11	x2F11
R3:	x5321	x5321
R4:	x331F	x331F
R5:	x1F22	x1F22
R6:	x01FF	x01FF
R7:	x341F	x3211
PC:	x3210	x3220
N:	0	0
Z:	1	1
P:	0	0