2023 ICS 期末习题课

hw4-T10, hw6

汤皓宇

2023 年 12 月 29 日

hw4-T10

題干

R0 and *R1* contain 16-bit bit vectors. The program below determines if rotating *R1* left by *n* bits produces the same bit vector that is in *R0*. If yes, the program stores the value *n* in M[x3020]. If not, the program stores -1 to M[x3020].

Rotating left a bit vector by one bit consists of left shifting the bit vector one bit, and then loading into bit[0] the bit that was shifted out of bit[15].

For example, rotating left **111**1000011110000 by 3 bits produces 1000011110000**111**.

Your job: Complete the program below by supplying the missing instructions so it stores *n* in location M[x3020] if rotating left *R1* by *n* bits produces the bit vector in *R0*, and store -1 if it is not possible to produce the bit vector of *R0* by rotating left *R1*. You are required to only use four registers: *R0*, *R1*, *R2*, and *R3*.

左移 R1 能否变 R0?

Address	Value	Comments
x3000	1001 000 000 111111	NOT R0, R0
x3001	0001 000 000 1 00001	ADD R0, R0, #1
x3002	0101 010 010 1 00000	AND R2, R2, #0
x3003	0001 011 000 0 00 001	ADD R3, R0, R1
x3004		
x3005	0001 010 010 1 00001	ADD R2, R2, #1
x3006		
x3007	0000 010 000000111	BRz x300F
x3008	0101 001 001 1 11111	AND R1, R1, xFFFF
x3009	0000 100 000000010	BRn x300C
x300A	0001 001 001 0 00 001	ADD R1, R1, R1
x300B	0000 111 111110111	BRnzp x3003
x300C		
x300D		
x300E	0000 111 111110100	BRnzp x3003
x300F	0101 010 010 1 00000	AND R2, R2, #0
x3010	0001 010 010 1 11111	ADD R2, R2, #-1
x3011		
x3012	1111 0000 0010 0101	TRAP x025

表 1: 要补全的程序

答案

Address	Value	Comments
x3000	1001 000 000 111111	NOT R0, R0
x3001	0001 000 000 1 00001	ADD R0, R0, #1
x3002	0101 010 010 1 00000	AND R2, R2, #0
x3003	0001 011 000 0 00 001	ADD R3, R0, R1
x3004	**0000 010 000001100**	**BRz x3011**
x3005	0001 010 010 1 00001	ADD R2, R2, #1
x3006	**0001 011 010 1 10000**	**ADD R3, R2, #-16**
x3007	0000 010 000000111	BRz x300F
x3008	0101 001 001 1 11111	AND R1, R1, xFFFF
x3009	0000 100 000000010	BRn x300C
x300A	0001 001 001 0 00 001	ADD R1, R1, R1
x300B	0000 111 111110111	BRnzp x3003
x300C	**0001 001 001 0 00 001**	**ADD R1, R1, R1**
x300D	**0001 001 001 1 00001**	**ADD R1, R1, #1**
x300E	0000 111 111110100	BRnzp x3003
x300F	0101 010 010 1 00000	AND R2, R2, #0
x3010	0001 010 010 1 11111	ADD R2, R2, #-1
x3011	**0011 010 000001110**	**ST R2, x3020**
x3012	1111 0000 0010 0101	TRAP x025

表 2: 补全的程序

hw6

T1

How many times is the loop executed? When the program halts, what is the value in R3?

```
1        .ORIG x3000
2        LD R2, TEXT
3        LD R3, TEST
4   AGAIN
5        ADD R3, R3, R2
6        ADD R2, R2, #-1
7        BRnzp TEST
8   TEXT
9        .STRINGZ "An LC-3 program"
10  TEST
11       BRnp AGAIN
12       TRAP x25
13       .END
```

T1

How many times is the loop executed? When the program halts, what is the value in R3?

```
1        .ORIG x3000
2        LD R2, TEXT
3        LD R3, TEST
4   AGAIN
5        ADD R3, R3, R2
6        ADD R2, R2, #-1
7        BRnzp TEST
8   TEXT
9        .STRINGZ "An LC-3 program"
10  TEST
11       BRnp AGAIN
12       TRAP x25
13       .END
```

答案

跳转看 *R2*，对应值为字符 'A' 的 ASCII 码 *x41*，在 *R2* 变为 0 后循环结束，因此循环执行 65 次。

至于 *R3* 中存的值，需要考虑最初 *LD* 进来的值，这个值又是 label *TEST* 处的值，也就是对应的机器码，再考虑字符串长度，得到机器码应该为 *x0BEC*，最终得到结果为十进制 5197.

T2

Two students wrote interrupt service routines for an assignment. Both service routines did exactly the same work, but the first student accidentally used *RET* at the end of his routine, while the second student correctly used *RTI*. There are three errors that arose in the first student's program due to his mistake. Describe any two of them.

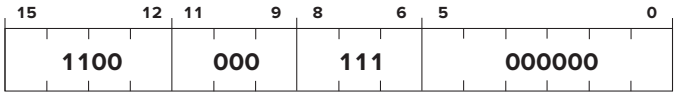
RET

Return from Subroutine

Assembler Format

RET†

Encoding



Operation

```
PC = R7;
```

Description

The PC is loaded with the value in R7. Its normal use is to cause a return from a previous JSR(R) instruction.

Example

RET ; PC ← R7

†The RET instruction is a specific encoding of the JMP instruction. See also JMP.

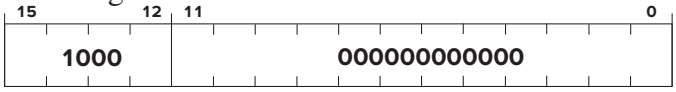
RTI

Return from Trap or Interrupt

Assembler Format

```
RTI
```

Encoding



Operation

```
if (PSR[15] == 1)
   Initiate a privilege mode exception;
else
   PC=mem[R6]; R6 is the SSP, PC is restored
   R6=R6+1;
   TEMP=mem[R6];
   R6=R6+1; system stack completes POP before saved PSR is restored
   PSR=TEMP; PSR is restored
   if (PSR[15] == 1)
      Saved_SSP=R6 and R6=Saved_USP;
```

Description

If the processor is running in User mode, a privilege mode exception occurs. If in Supervisor mode, the top two elements on the system stack are popped and loaded into PC, PSR. After PSR is restored, if the processor is running in User mode, the SSP is saved in Saved_SSP, and R6 is loaded with Saved_USP.

Example

```
RTI    ; PC, PSR ← top two values popped off stack.
```

Note

RTI is the last instruction in both interrupt and trap service routines and returns control to the program that was running. In both cases, the relevant service routine is initiated by first pushing the PSR and PC of the program that is running onto the system stack. Then the starting address of the appropriate service routine is loaded into the PC, and the service routine executes with supervisor privilege. The last instruction in the service routine is RTI, which returns control to the interrupted program by popping two values off the supervisor stack to restore the PC and PSR. In the case of an interrupt, the PC is restored to the address of the instruction that was about to be processed when the interrupt was initiated. In the case of an exception, the PC is restored to either the address of the instruction that caused the exception or the address of the following instruction, depending on whether the instruction that caused the exception is to be re-executed. In the case of a TRAP service routine, the PC is restored to the instruction following the TRAP instruction in the calling routine. In the case of an interrupt or TRAP, the PSR is restored to the value it had when the interrupt was initiated. In the case of an exception, the PSR is restored to the value it had when the exception occurred or to some modified value, depending on the exception. See also Section A.3.

T2

Two students wrote interrupt service routines for an assignment. Both service routines did exactly the same work, but the first student accidentally used *RET* at the end of his routine, while the second student correctly used *RTI*. There are three errors that arose in the first student's program due to his mistake. Describe any two of them.

答案

1. 因为是中断服务，所以 *R7* 中并不保存返回地址，而 *RET* 会跳转到 *R7* 保存的地址，因此在中断服务中直接使用 *RET* 会跳转到错误的位置
2. 直接调用 *RET* 并没有恢复在中断服务中被保存下来的 Processor Status Register，因而并没有从特权模式恢复到用户模式，之前的状态码也没有被恢复
3. *RET* 也不涉及对栈的操作，但从中断服务恢复，应该要恢复栈指针

T3–1

What does the student intend to print?

```
1         .ORIG x3000
2         JSR A
3         OUT
4         BRnzp DONE
5   A
6         AND R0, R0, #0
7         ADD R0, R0, #9
8         JSR B
9         RET
10  DONE
11        HALT
12  ASCII
13        .FILL x0040
14  B
15        LD R1, ASCII
16        ADD R0, R0, R1
17        RET
18        .END
```

T3–1

What does the student intend to print?

```
1         .ORIG x3000
2         JSR A
3         OUT
4         BRnzp DONE
5   A
6         AND R0, R0, #0
7         ADD R0, R0, #9
8         JSR B
9         RET
10  DONE
11        HALT
12  ASCII
13        .FILL x0040
14  B
15        LD R1, ASCII
16        ADD R0, R0, R1
17        RET
18        .END
```

答案

想打印的是字符 'I'，因为如果不考虑实际跳转，逻辑上在 *OUT* 前 *R0* 中保存的是 *x0049*，对应 ASCII 码表中的字符 'I'

T3–2

Will the program print the character? Can you explain why?

```
1         .ORIG x3000
2         JSR A
3         OUT
4         BRnzp DONE
5   A
6         AND R0, R0, #0
7         ADD R0, R0, #9
8         JSR B
9         RET
10  DONE
11        HALT
12  ASCII
13        .FILL x0040
14  B
15        LD R1, ASCII
16        ADD R0, R0, R1
17        RET
18        .END
```

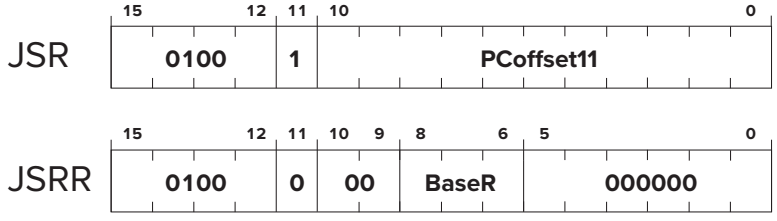
JSR
JSRR

Jump to Subroutine

Assembler Formats

```
JSR   LABEL
JSRR  BaseR
```

Encoding



Operation

```
TEMP = PC;†
if (bit[11] == 0)
   PC = BaseR;
else
   PC = PC† + SEXT(PCoffset11);
R7 = TEMP;
```

Description

First, the incremented PC is saved in a temporary location. Then the PC is loaded with the address of the first instruction of the subroutine, which will cause an unconditional jump to that address after the current instruction completes execution. The address of the subroutine is obtained from the base register (if bit [11] is 0), or the address is computed by sign-extending bits [10:0] and adding this value to the incremented PC (if bit [11] is 1). Finally, R7 is loaded with the value stored in the temporary location. This is the linkage back to the calling routine.

Examples

```
JSR   QUEUE  ; Put the address of the instruction following JSR into R7;
             ; Jump to QUEUE.
JSRR  R3     ; Put the address of the instruction following JSRR into R7;
             ; Jump to the address contained in R3.
```

†This is the incremented PC.

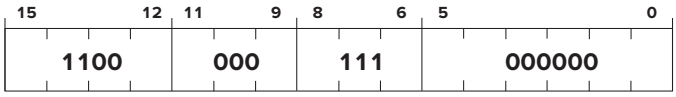
RET

Return from Subroutine

Assembler Format

RET†

Encoding



Operation

```
PC = R7;
```

Description

The PC is loaded with the value in R7. Its normal use is to cause a return from a previous JSR(R) instruction.

Example

RET ; PC ← R7

†The RET instruction is a specific encoding of the JMP instruction. See also JMP.

T3–2

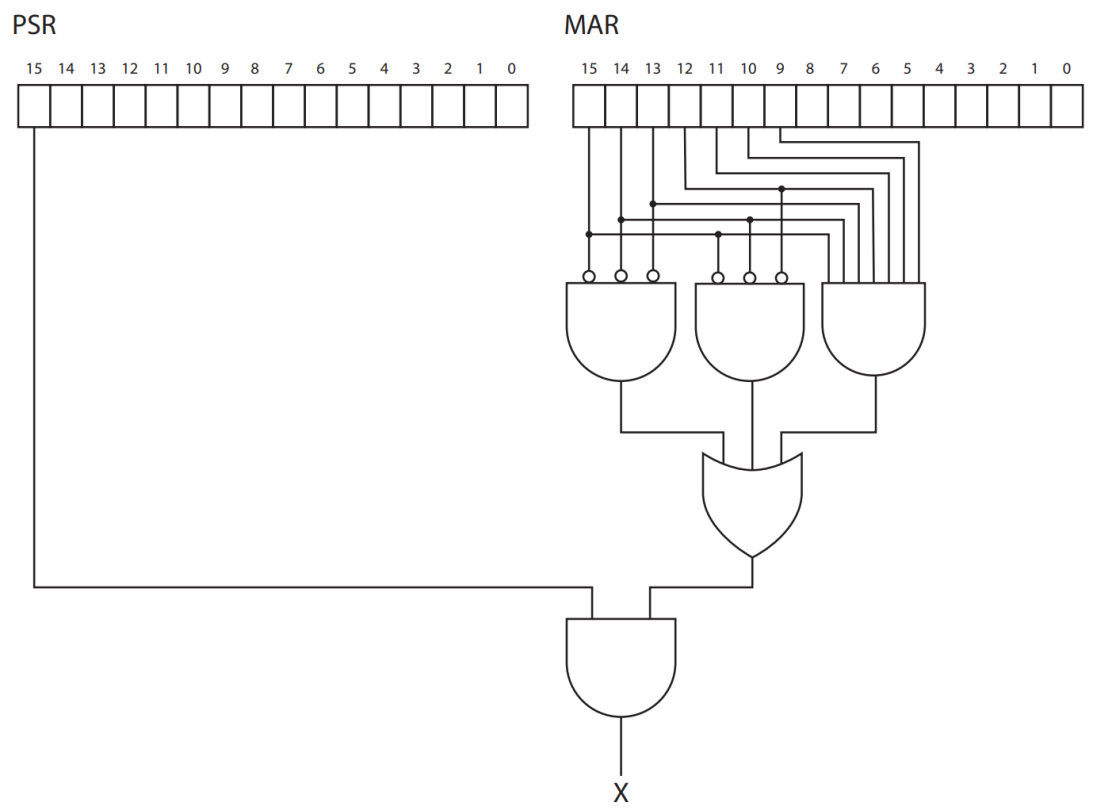
Will the program print the character? Can you explain why?

```
1         .ORIG x3000
2         JSR A
3         OUT
4         BRnzp DONE
5   A
6         AND R0, R0, #0
7         ADD R0, R0, #9
8         JSR B
9         RET
10  DONE
11        HALT
12  ASCII
13        .FILL x0040
14  B
15        LD R1, ASCII
16        ADD R0, R0, R1
17        RET
18        .END
```

答案

实际上什么也不会输出，因为调用了一次 *JSR* 后再次调用 *JSR* 时没有保存原本的返回地址 *R7*，因此返回地址已经被覆盖，因而从 *B* 中回到 *A* 中再次执行 *RET* 会在 *A* 中出现死循环，无法真正执行到 *OUT*

Can you tell what signal *X* is? When will *X* be set to 1?



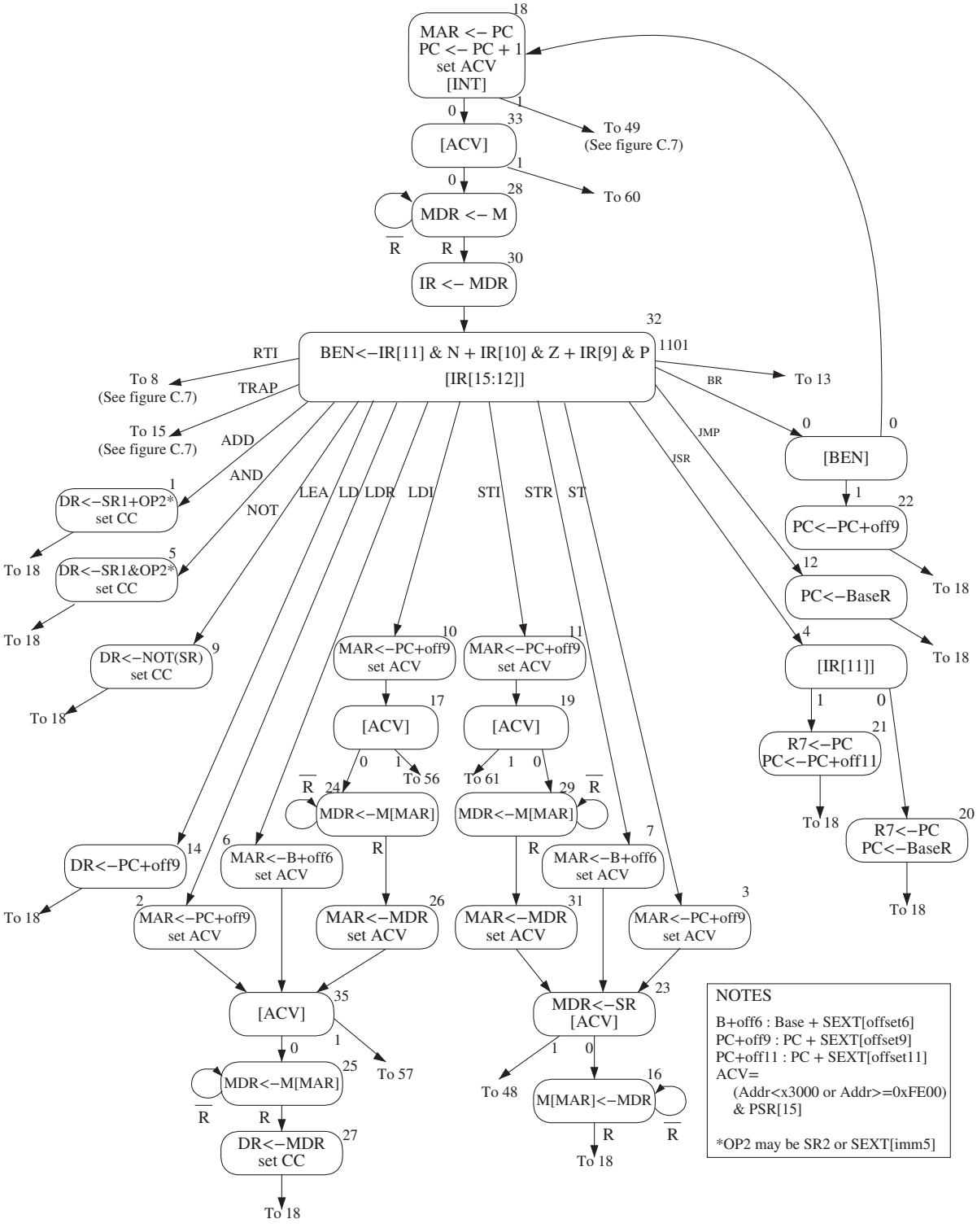
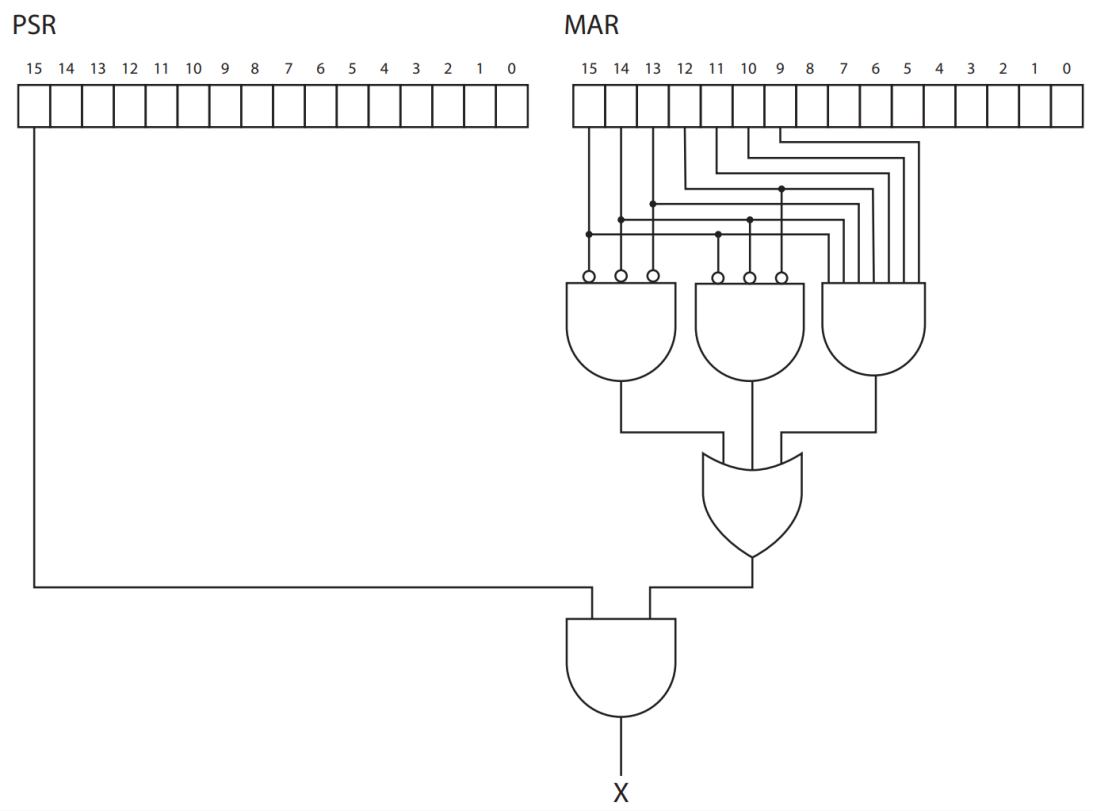


Figure C.2 **A state machine for the LC-3.**

T4

Can you tell what signal *X* is? When will *X* be set to 1?



答案

当 *PSR* 的最高位为 1 时 (此时位于用户模式)，且当 *MAR* 的值处于 $[x0000, x3000) \bigcup [xFE00, xFFFF]$ 时 (此时 *MAR* 位于特权内存区)，*X* 为 1，否则为 0. *ACV*.

T5

Imagine that you are writing a simple LC-3 program that is designed to receive a character from the keyboard and then display that character on the screen.

1. How do you check in LC-3 if there is a new character input from the keyboard?
2. Once a new character input is detected, how do you read this character from the keyboard?
3. How do you display the read character on the screen?

T5

Imagine that you are writing a simple LC-3 program that is designed to receive a character from the keyboard and then display that character on the screen.

1. How do you check in LC-3 if there is a new character input from the keyboard?
2. Once a new character input is detected, how do you read this character from the keyboard?
3. How do you display the read character on the screen?

答案

1. 检测 *KBSR* 最高位是否为 1
2. 读取 *KBDR*，并清除 *KBSR* 最高位
3. 当 *DSR* 最高位为 1 的时候，写入 *DDR*

T5–4

Provide a simple LC-3 assembly code snippet that demonstrates this process.

```
1        .ORIG x3000
2  GET LDI R1, KBSR
3        BRzp GET
4        LDI R0, KBDR
5  WAIT LDI R1, DSR
6        BRzp WAIT
7        STI R0, DDR
8        TRAP x25
9  KBSR .FILL xFE00
10 KBDR .FILL xFE02
11 DSR .FILL xFE04
12 DDR .FILL xFE06
13       .END
```

T6

Here's a subroutine that takes 4 chars in hex from keyboard and store the value they represent in *R0* using polling technique. Note that it assumes all possible input characters are in *0123456789ABCDEF*.

1. Fill in the blanks (denoted by underlines _) to complete the program.
2. Briefly explain what the four consecutive *ADD R0, R0, R0* do.
3. We have no idea what *R0* stores before the subroutine is called, so we placed the instruction *AND R0, R0, #0* before the label *GETCHAR* in order to clear *R0*. Is this instruction necessary? Why or why not?

T6-读入四位十六进制数

```
1  HEX_INPUT
2        ST R1, SAVE_R1
3        ST R2, SAVE_R2
4        ST R3, SAVE_R3
5        ST R4, SAVE_R4
6        LD R1, C1
7        LD R2, C2
8        AND R3, R3, #0
9        ADD R3, R3, #4
10       AND R0, R0, #0
11 GETCHAR
12       ADD R0, R0, R0
13       ADD R0, R0, R0
14       ADD R0, R0, R0
15       ADD R0, R0, R0
16 WAIT
17       LDI R4, KBSR
18       BRzp ____
19       LDI R4, KBDR
20       ADD R4, R4, R1
21       BRzp ____
```

```
22       ADD R4, R4, R2
23       BR ____
24 LETTER
25       ADD R4, R4, #10
26 CONTINUE
27       ADD R0, R0, R4
28       ADD R3, R3, #-1
29       BRp ____
30       LD R1, SAVE_R1
31       LD R2, SAVE_R2
32       LD R3, SAVE_R3
33       LD R4, SAVE_R4
34       RET
35 C1 .FILL #___
36 C2 .FILL #___
37 KBSR .FILL xFE00
38 KBDR .FILL xFE02
39 SAVE_R1 .BLKW 1
40 SAVE_R2 .BLKW 1
41 SAVE_R3 .BLKW 1
42 SAVE_R4 .BLKW 1
```

T6 补全

```
1  HEX_INPUT
2        ST R1, SAVE_R1
3        ST R2, SAVE_R2
4        ST R3, SAVE_R3
5        ST R4, SAVE_R4
6        LD R1, C1
7        LD R2, C2
8        AND R3, R3, #0
9        ADD R3, R3, #4
10       AND R0, R0, #0
11 GETCHAR
12       ADD R0, R0, R0
13       ADD R0, R0, R0
14       ADD R0, R0, R0
15       ADD R0, R0, R0
16 WAIT
17       LDI R4, KBSR
18       BRzp WAIT
19       LDI R4, KBDR
20       ADD R4, R4, R1
21       BRzp LETTER
```

```
22       ADD R4, R4, R2
23       BR CONTINUE
24 LETTER
25       ADD R4, R4, #10
26 CONTINUE
27       ADD R0, R0, R4
28       ADD R3, R3, #-1
29       BRp GETCHAR
30       LD R1, SAVE_R1
31       LD R2, SAVE_R2
32       LD R3, SAVE_R3
33       LD R4, SAVE_R4
34       RET
35 C1 .FILL #-65
36 C2 .FILL #17
37 KBSR .FILL xFE00
38 KBDR .FILL xFE02
39 SAVE_R1 .BLKW 1
40 SAVE_R2 .BLKW 1
41 SAVE_R3 .BLKW 1
42 SAVE_R4 .BLKW 1
```

T7

1. What is the output of the program?
2. How many bytes of memory does the program occupy?

```
1        .ORIG x3000
2        LEA R0, STRING
3        PUTS
4        LD R0, SYMBOL
5        OUT
6        HALT
7   STRING .STRINGZ "H3ll0_W0r1d"
8   SYMBOL .FILL #33
9        .END
```

T7

1. What is the output of the program?
2. How many bytes of memory does the program occupy?

```
1        .ORIG x3000
2        LEA R0, STRING
3        PUTS
4        LD R0, SYMBOL
5        OUT
6        HALT
7   STRING .STRINGZ "H3ll0_W0r1d"
8   SYMBOL .FILL #33
9        .END
```

答案

1. *'H3ll0_W0r1d!'*
2. $18 \times 2 = 36$ 字节. 每条命令占两个字节，注意字符串结尾有一个 '\0'.

T8

1. What problem might arise if a program does not check *KBSR* before *reading KBDR*?
2. What problem might arise if the keyboard does not check *KBSR* before *writing* to *KBDR*?
3. Which one of the two problems mentioned above is more likely to occur? Justify your answer.

T8

1. What problem might arise if a program does not check *KBSR* before *reading* *KBDR*?
2. What problem might arise if the keyboard does not check *KBSR* before *writing* to *KBDR*?
3. Which one of the two problems mentioned above is more likely to occur? Justify your answer.

答案

1. 键盘一次输入的一个字符可能被读取多次
2. 可能覆盖还没有被系统处理的字符
3. 前者更可能发生，CPU 比人的输入快得多

T9

What is the output of this program? Assume all registers are initialized to 0 before the program executes.

```
1         .ORIG x3000
2         LD R0, A
3         LD R1, B
4         AND R1, R0, R1
5         ST R0, #7
6         ST R1, #5
7         ST R2, #6
8         LEA R0, LABEL
9         TRAP x22
10        TRAP x25
11  LABEL .STRINGZ "FUNKY"
12  LABEL2 .STRINGZ "HELLO WORLD"
13  A .FILL #33
14  B .FILL #32
15        .END
```

T9

What is the output of this program? Assume all registers are initialized to 0 before the program executes.

```
1        .ORIG x3000
2        LD R0, A
3        LD R1, B
4        AND R1, R0, R1
5        ST R0, #7
6        ST R1, #5
7        ST R2, #6
8        LEA R0, LABEL
9        TRAP x22
10       TRAP x25
11   LABEL .STRINGZ "FUNKY"
12   LABEL2 .STRINGZ "HELLO WORLD"
13   A .FILL #33
14   B .FILL #32
15       .END
```

答案

F !

T10

The program uses only *R0* and *R1*.
Note also that one of the instructions **in the program** must be labeled *AGAIN*, but now the label is missing.
After execution of the program, the contents of *A* is *x1800*. In total, 9 instructions are executed.

```
1         .ORIG x3000
2         LD R0, A
3         LD R1, B
4         BRz DONE
5         ____________ (a)
6         ____________ (b)
7         BRnzp AGAIN
8   DONE
9         ST R0, A
10        HALT
11  A .FILL x0___ (c)
12  B .FILL x0001
13        .END
```

T10

Note that we doesn't say anything about how many clock cycles a memory access takes.

Cycle Number	State Number	Control Signals
1	18	LD.MAR: _____ LD.REG: _____ GateMDR: _____ LD.PC: _____ PCMUX: _____ GatePC: _____
–	0	LD.MAR: _____ LD.REG: _____ BEN: _____ LD.PC: _____ LD.CC: _____
–	–	LD.REG: LOAD DR: 000 GateMDR: _____ GateALU: _____ GateMARMUX: _____
57	1	LD.MAR: _____ ALUK: _____ GateALU: _____ LD.REG: _____ DR: _____ GatePC: _____
77	22	ADDR1MUX: _____ ADDR2MUX: _____ LD.PC: _____ LD.MAR: _____ PCMUX: _____
100	15	

```
1         .ORIG x3000
2         LD R0, A
3         LD R1, B
4         BRz DONE
5         _____ (a)
6         _____ (b)
7         BRnzp AGAIN
8    DONE
9         ST R0, A
10        HALT
11   A .FILL x0___ (c)
12   B .FILL x0001
13        .END
```

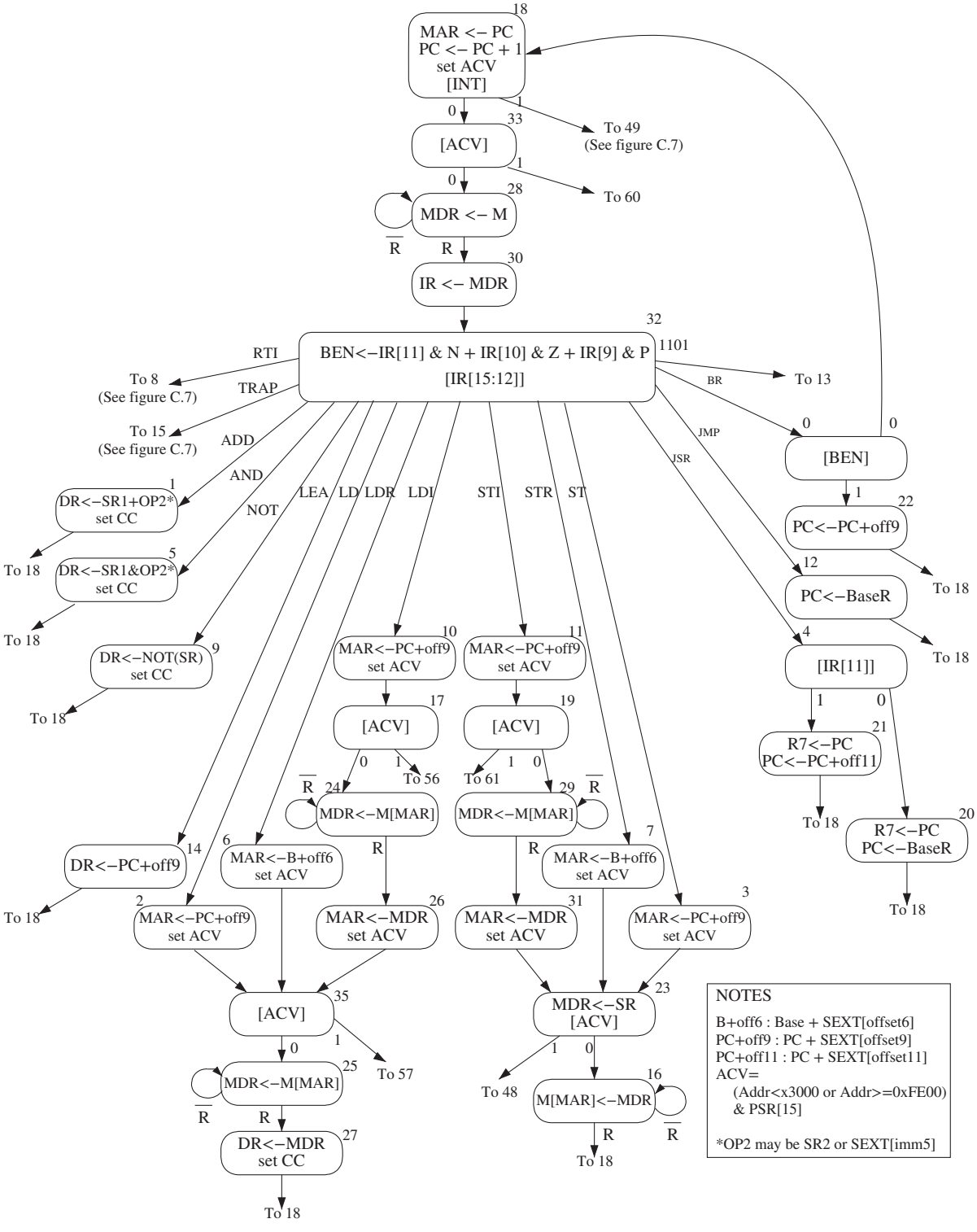


Figure C.2 **A state machine for the LC-3.**

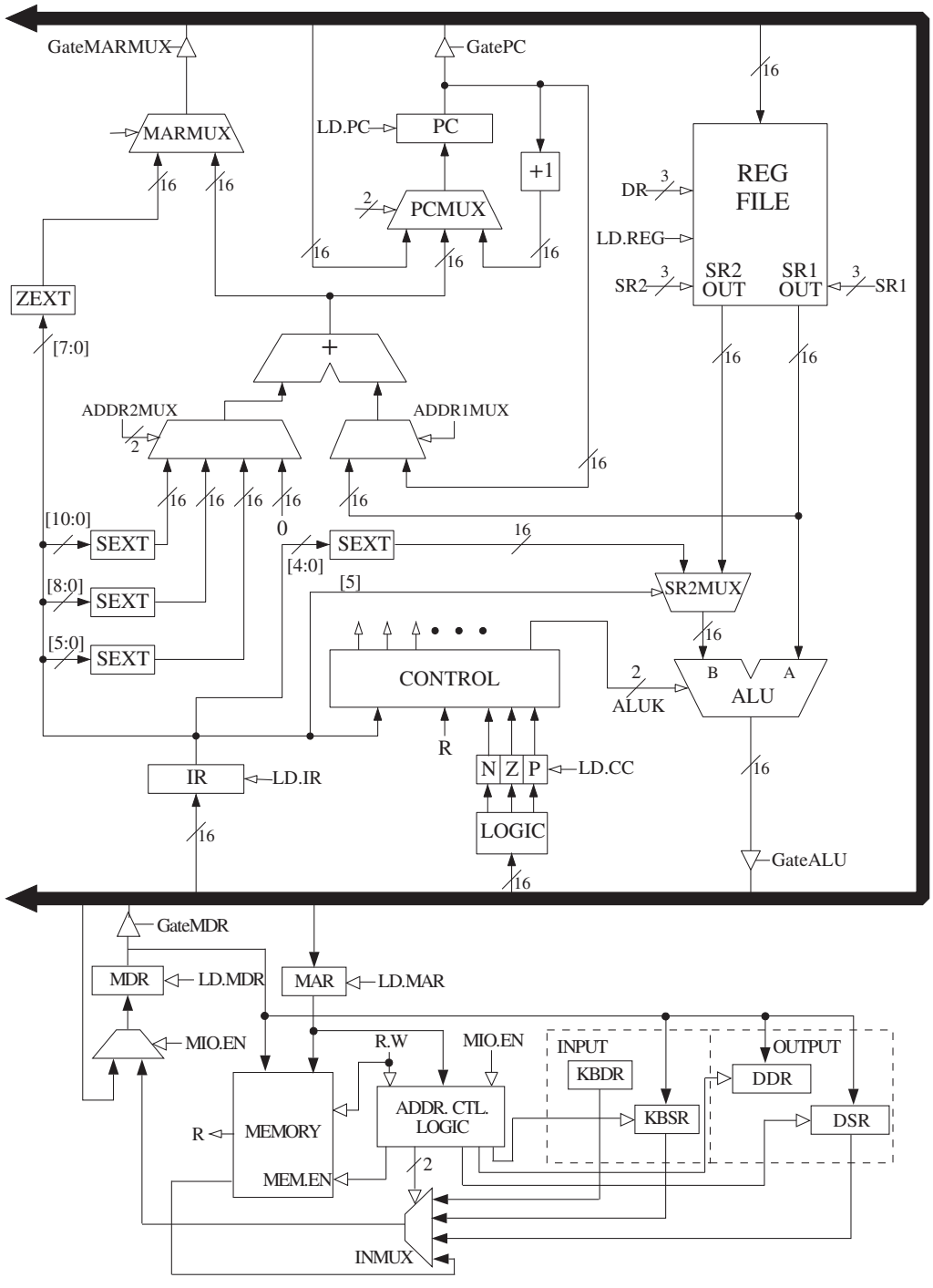


Figure C.3 The LC-3 data path.

Table C.1 Data Path Control Signals

Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
LD.Priv/1:	NO, LOAD	
LD.Priority/1:	NO, LOAD	
LD.SavedSSP/1:	NO, LOAD	
LD.SavedUSP/1:	NO, LOAD	
LD.ACV/1:	NO, LOAD	
LD.Vector/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateVector/1:	NO, YES	
GatePC-1/1:	NO, YES	
GatePSR/1:	NO, YES	
GateSP/1:	NO, YES	
PCMUX/2:	PC+1	;select pc+1
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/2:	11.9	;destination IR[11:9]
	R7	;destination R7
	SP	;destination R6
SR1MUX/2:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
	SP	;source R6
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
SPMUX/2:	SP+1	;select stack pointer+1
	SP−1	;select stack pointer−1
	Saved SSP	;select saved Supervisor Stack Pointer
	Saved USP	;select saved User Stack Pointer
MARMUX/1:	7.0	;select ZEXT[IR[7:0]]
	ADDER	;select output of address adder
TableMUX/1:	x00, x01	
VectorMUX/2:	INTV	
	Priv.exception	
	Opc.exception	
	ACV.exception	
PSRMUX/1:	individual settings, BUS	
ALUK/2:	ADD, AND, NOT, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
Set.Priv/1:	0	;Supervisor mode
	1	;User mode

T10–答案

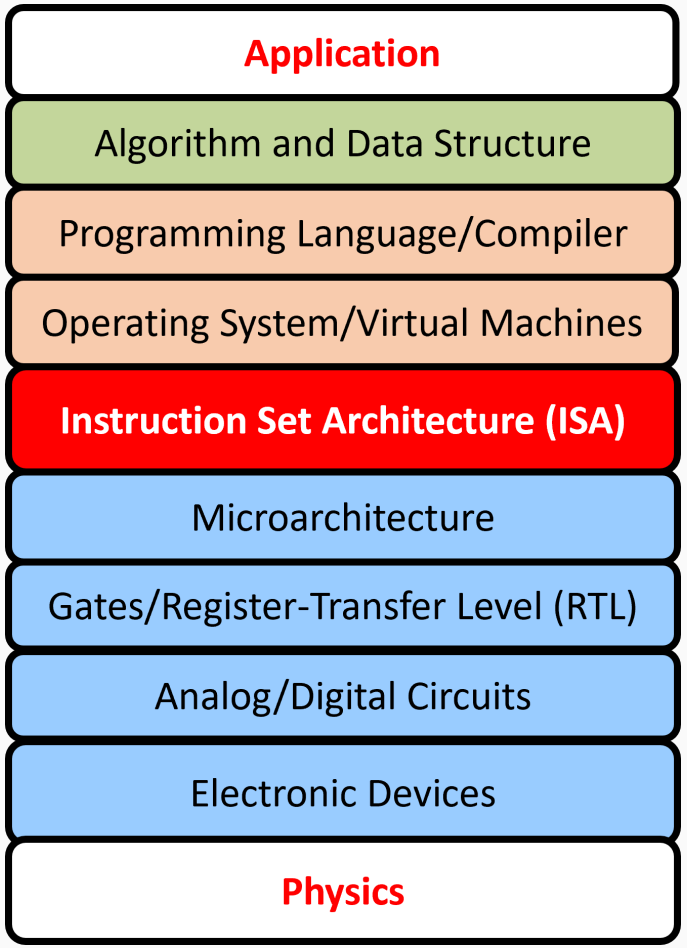
Cycle Number	State Number	Control Signals
1	18	LD.MAR: LOAD LD.REG: NO GateMDR: NO LD.PC: LOAD PCMUX: PC+1 GatePC: YES
39	0	LD.MAR: NO LD.REG: NO BEN: 0/FALSE LD.PC: NO LD.CC: NO
48	1	LD.REG: LOAD DR: 000 GateMDR: NO GateALU: YES GateMARMUX: NO
57	1	LD.MAR: NO ALUK: ADD GateALU: YES LD.REG: LOAD DR: 001 GatePC: NO
77	22	ADDR1MUX: PC ADDR2MUX: PCoffset9 LD.PC: LOAD LD.MAR: NO PCMUX: ADDER
100	15	

```
1         .ORIG x3000
2         LD R0, A
3         LD R1, B
4   AGAIN BRz DONE
5         ADD R0, R0, R0
6         ADD R1, R1, #-1
7         BRnzp AGAIN
8   DONE
9         ST R0, A
10        HALT
11  A .FILL x0c00
12  B .FILL x0001
13        .END
```

课程总结

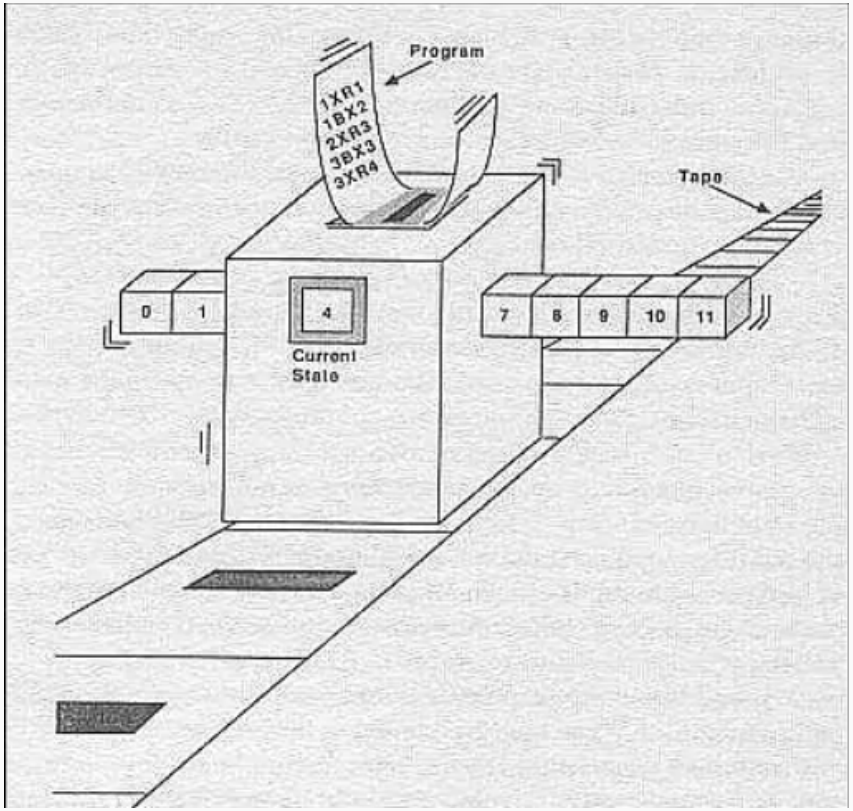
这门课讲了什么?

- 如何表示数 (数制、补码、浮点数)
- 如何进行运算 (晶体管-> 逻辑门-> 组合电路)
- 如何记忆值 (锁存器-> 寄存器-> 存储器-> 时序电路)
- 状态机 (其实所有计算机系统本质也都是一个状态机)
- 冯诺依曼架构 (实际存在其他架构，如哈佛架构)
- 以 LC-3 指令集为例，了解数据结构、函数、IO、中断



什么是计算机系统?

- interrupt/trap handler
- 图灵机



- 操作数：必须包含标识数字的表示进制的符号。用 # 表示十进制，x 表示十六进制，b 表示二进制
- 标签：由 1 到 20 个字母或数字字符组成 (每个字符是大写或小写字母，或十进制数字)，以字母开头。注意不能是保留字 (ADD, NOT, x1000, R4 等)
- 伪操作：.BLKW 分配 n 个空间 (n 为操作数)，.STRINGZ 分配 n + 1 个空间 (n 为字符串长度，最后一个是 x0000)
- 汇编时有两遍扫描：第一遍找到所有标签并计算出对应的地址，创建符号表，第二遍翻译生成对应的机器码
- 注意检查地址是否越界，如 LD 指令 PCoffset9 为 [-256, 255]，因此 LD 后面的标签距离当前 LD 指令的地址范围应该为 [-255, 256] (注意 PC 自增)
- 注意指令符号扩展补码

- JSR、JSRR 指令，用来调用子程序，注意与 JMP 的区别 (R7 记录自增的 PC，即下一条指令的地址)
- 调用者保存与被调用者保存
- 栈：R6 栈顶指针 (指向栈顶)，注意 Underflow (栈空 Pop) 和 Overflow (栈满 Push) 以及对应的检测
- (循环) 队列：R3 队首 (FRONT) 指针 (下一个是队首)，R4 队尾 (REAR) 指针 (指向队尾)。队空 FRONT = REAR，队满 FRONT - 1 = REAR，同样注意 Underflow 和 Overflow 及检测。

递归程序

函数栈帧

```
1  FACT
2        ADD R6, R6, #-1
3        STR R1, R6, #0
4        ADD R1, R0, #-1
5        BRz NO_RECURSE
6        ADD R6, R6, #-1
7        STR R7, R6, #0
8        ADD R6, R6, #-1
9        STR R0, R6, #0
10       ADD R0, R0, #-1
```

```
11 B
12       JSR FACT
13       LDR R1, R6, #0
14       ADD R6, R6, #1
15       MUL R0, R0, R1
16       LDR R7, R6, #0
17       ADD R6, R6, #1
18 NO_RECURSE
19       LDR R1, R6, #0
20       ADD R6, R6, #1
21       RET
```

LC-3 内存

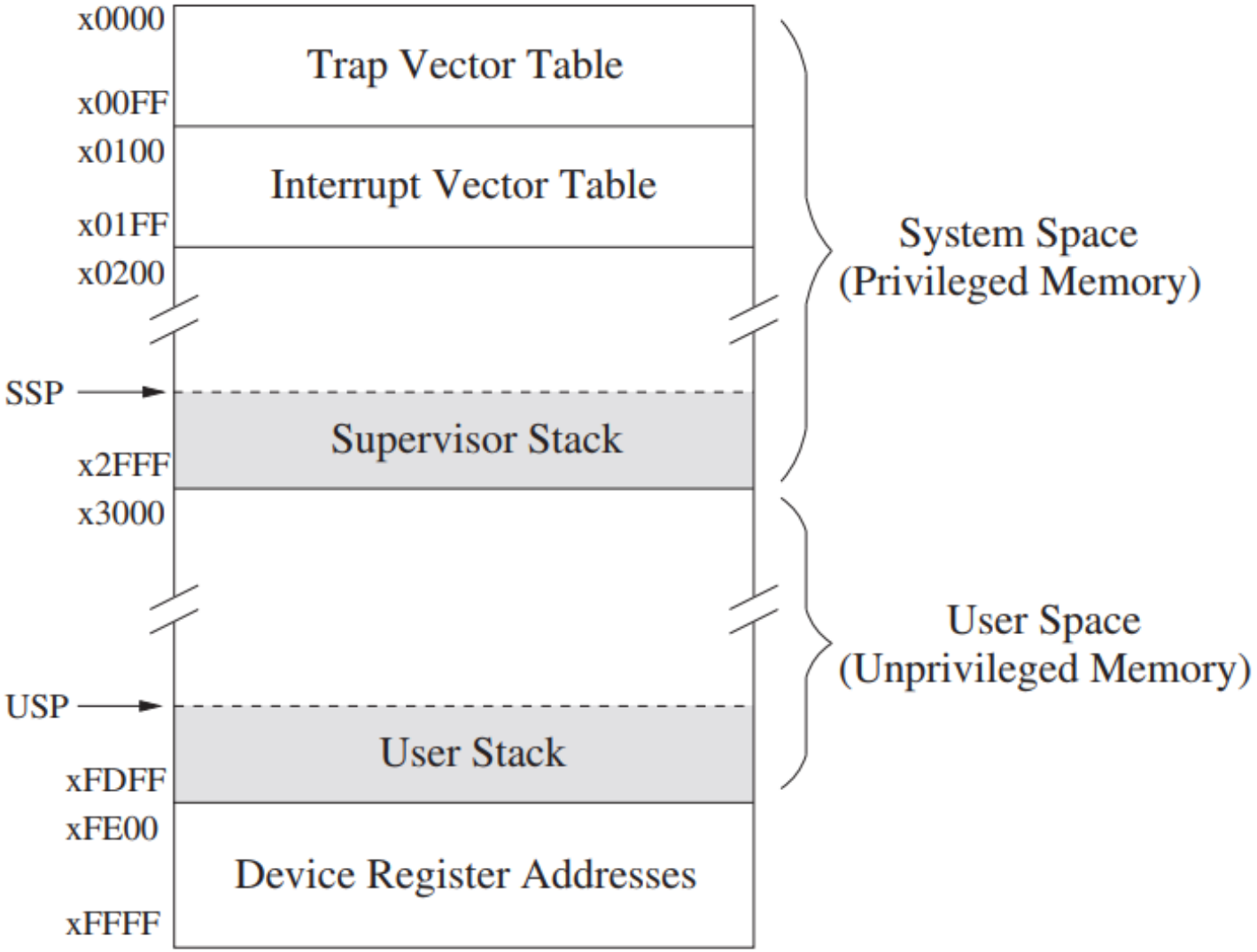


Figure A.1 Memory map of the LC-3

Input

输入涉及寄存器

- keyboard status register (KBSR) *xFE00*
- keyboard data register (KBDR) *xFE02*

KBSR[15] 控制键盘和处理器的同步

- 当敲击键盘上的一个键时，该键的 ASCII 码被载入 *KBDR*[7:0] 中，与键盘相关的电路自动将 *KBSR*[15] 设置为 1
- 当 LC-3 读取 *KBDR* 时，与键盘相关的电路自动清除 *KBSR*[15]，允许敲击另一个键
- 如果 *KBSR*[15]=1，则最后一个击键对应的 ASCII 码还没有被读取，因此键盘被禁用

Output

输出涉及寄存器

- display status register (DSR) *xFE04*
- display data register (DDR) *xFE06*

DSR[15] 控制处理器和显示器的同步

- 当 LC-3 向 *DDR*[7:0] 存入一个 ASCII 码进行输出后，显示器会在开始处理 *DDR*[7:0] 时，自动清除 *DSR*[15]
- 当显示器在屏幕上显示完字符时，它会自动设置 *DSR*[15]。这是给处理器的一个信号，表明处理器可以将另一个 ASCII 码传输到 *DDR* 进行输出
- 只要 *DSR*[15] 是清零的，说明显示器仍在处理前一个字符，因此就处理器的额外输出而言，此时显示器是禁用的

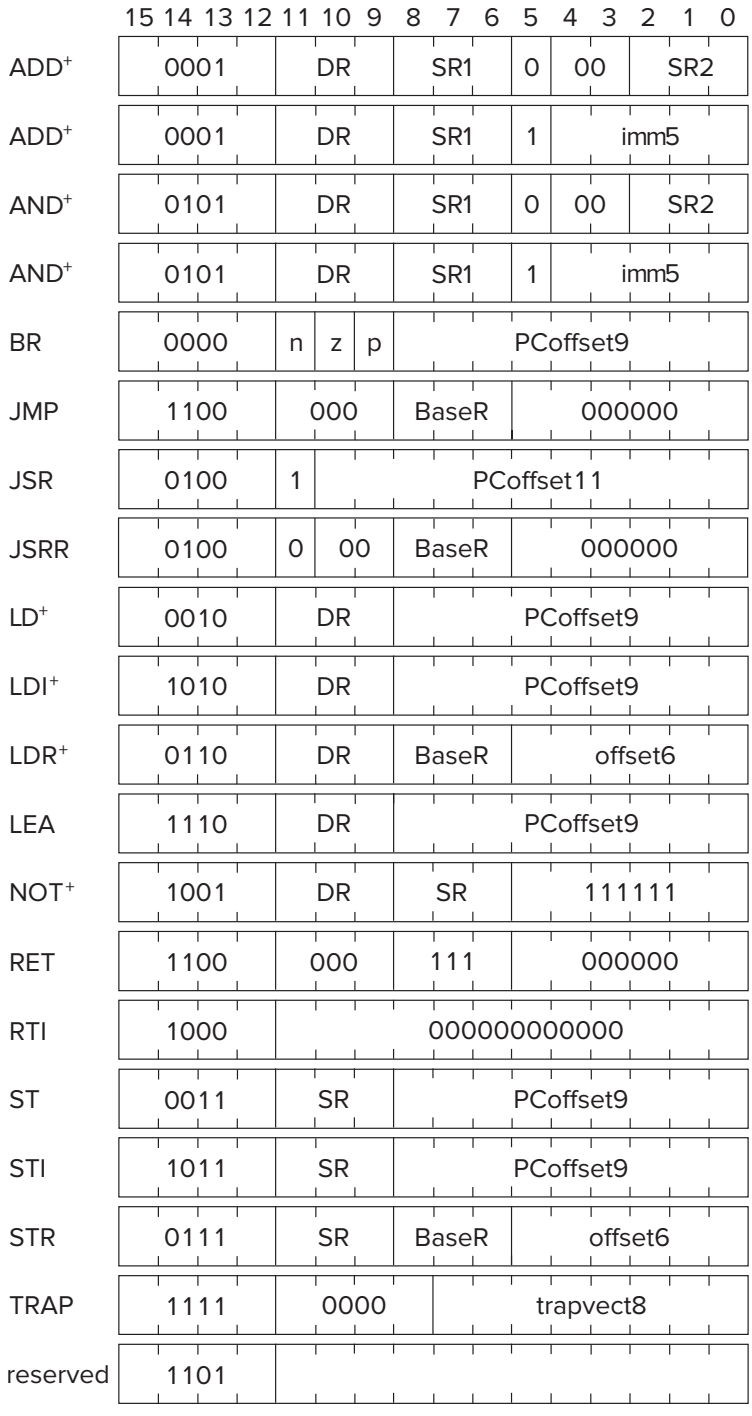


Figure A.2 Format of the entire LC-3 instruction set. *Note:* + indicates instructions that modify condition codes

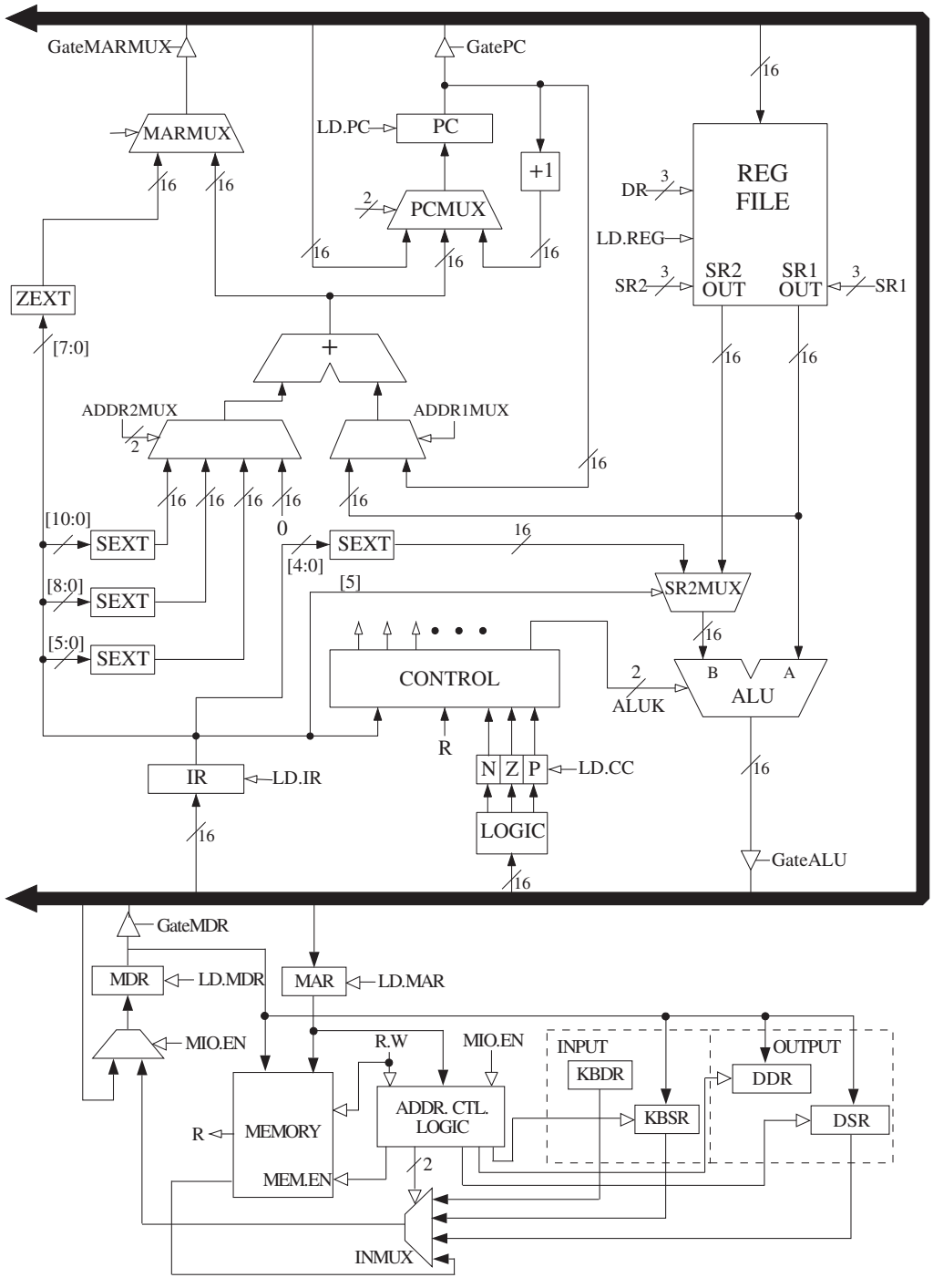


Figure C.3 The LC-3 data path.

Table C.1 Data Path Control Signals

Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
LD.Priv/1:	NO, LOAD	
LD.Priority/1:	NO, LOAD	
LD.SavedSSP/1:	NO, LOAD	
LD.SavedUSP/1:	NO, LOAD	
LD.ACV/1:	NO, LOAD	
LD.Vector/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateVector/1:	NO, YES	
GatePC-1/1:	NO, YES	
GatePSR/1:	NO, YES	
GateSP/1:	NO, YES	
PCMUX/2:	PC+1	;select pc+1
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/2:	11.9	;destination IR[11:9]
	R7	;destination R7
	SP	;destination R6
SR1MUX/2:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
	SP	;source R6
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
SPMUX/2:	SP+1	;select stack pointer+1
	SP−1	;select stack pointer−1
	Saved SSP	;select saved Supervisor Stack Pointer
	Saved USP	;select saved User Stack Pointer
MARMUX/1:	7.0	;select ZEXT[IR[7:0]]
	ADDER	;select output of address adder
TableMUX/1:	x00, x01	
VectorMUX/2:	INTV	
	Priv.exception	
	Opc.exception	
	ACV.exception	
PSRMUX/1:	individual settings, BUS	
ALUK/2:	ADD, AND, NOT, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
Set.Priv/1:	0	;Supervisor mode
	1	;User mode

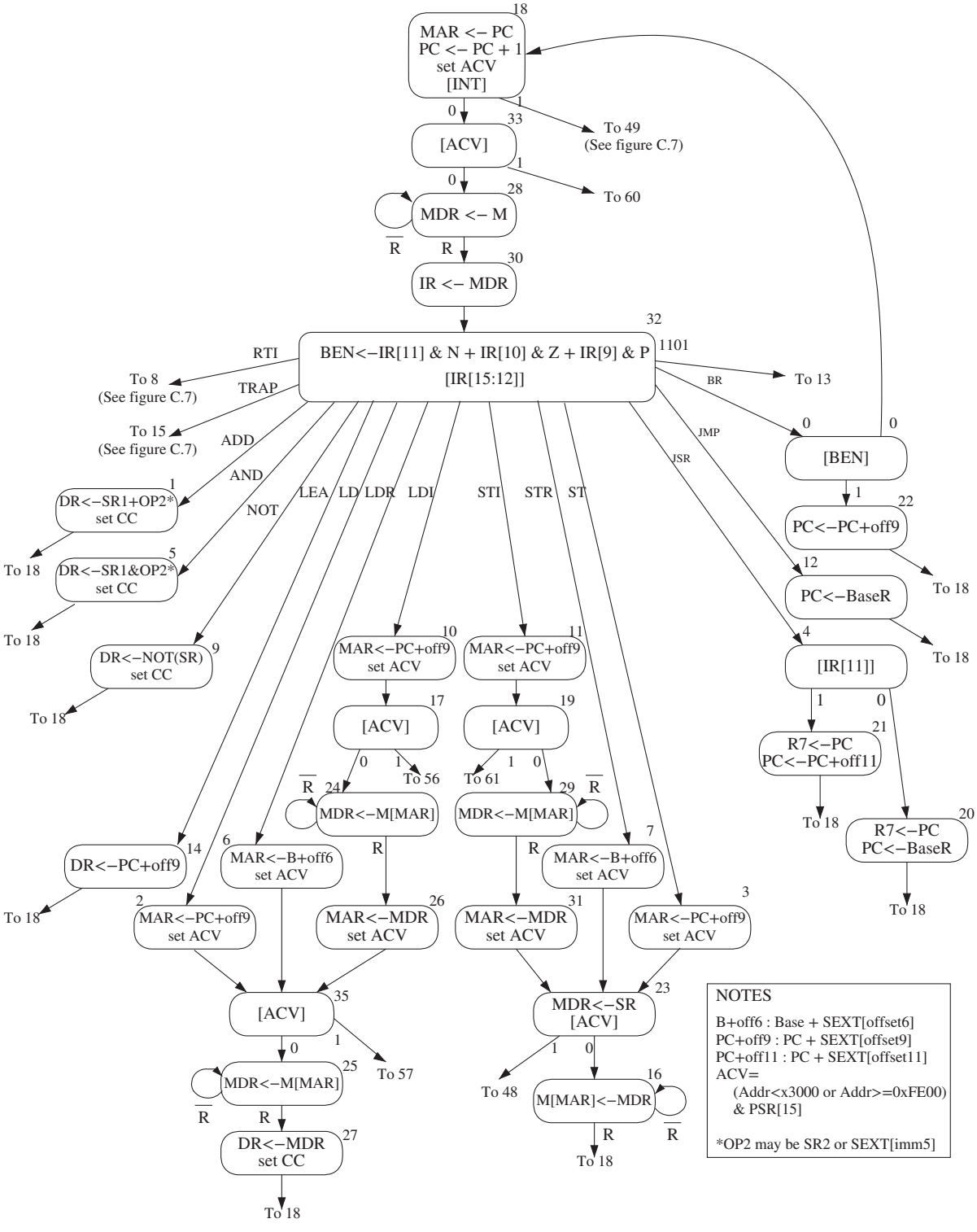


Figure C.2 **A state machine for the LC-3.**

the event that causes the program that is executing to stop. Interrupts are events that usually have nothing to do with the program that is executing. Exceptions are events that are the direct result of something going awry in the program that is executing. The LC-3 specifies three exceptions: a privilege mode violation, an illegal opcode, and an ACV exception. Figure C.7 shows the state machine that carries these out. Figure C.8 shows the data path, after adding the additional structures to Figure C.3 that are needed to make interrupt and exception processing work.

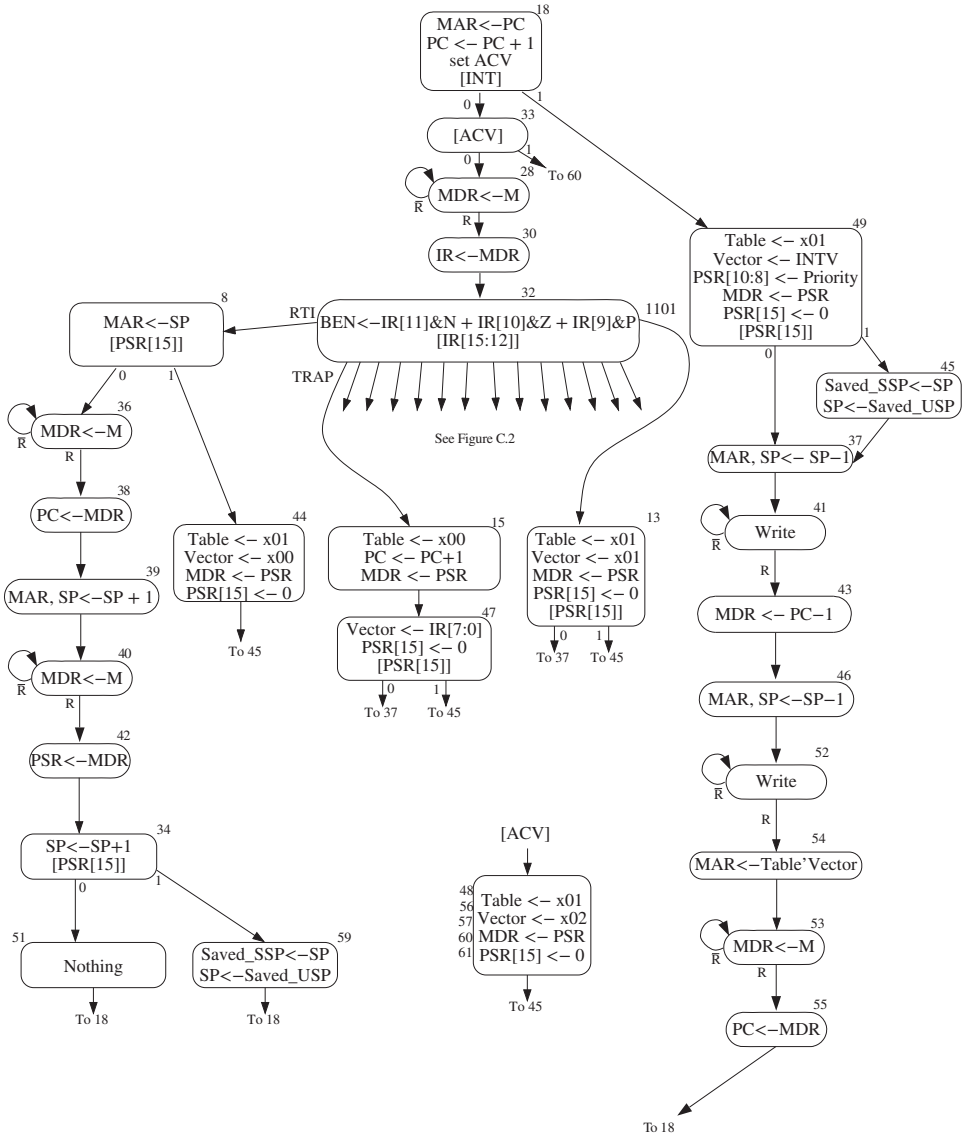


Figure C.7 LC-3 state machine showing interrupt control.

附录 A

- A.1 LC-3 概述：一些基本概念的介绍
- A.2 指令集：用于速查各条指令的行为
- A.3 中断和异常处理：包括一些处理细节，建议看一遍，其中的异常处理部分可以不看

附录 C

- C.1 LC-3 微结构概述：描述主要组件以及信号，可以不看
- C.2 状态机，重点必看
- C.3 数据通路，重点必看
- C.4 控制结构：看一下 DRMUX 和 SR1MUX
- C.5 TRAP：TRAP 细节，建议看一遍
- C.6 内存映射 I/O：简单看一下即可
- C.7 中断和异常处理以及 RTI，异常可以不看，其它建议看一遍
- C.8 如果能自行填完那张表，就能从底层理解 LC-3 的运行了，不过要注意其中有不少属于考试不会涉及的内容

Questions?

祝大家取得好成绩！