Department of Electrical and Computer Engineering
The University of Texas at Austin

EE 306, Fall 2017
Yale Patt, Instructor
Stephen Pruett, Siavash Zangeneh, Aniket Deshmukh, Zachary Susskind, Meiling Tang, Jiahan Liu
Final Exam, December 16, 2017

Name: Solution

Part A:

Problem 1 (10 points):_____

Problem 2 (10 points):_____

Problem 3 (10 points):_____

Problem 4 (10 points):_____

Problem 5 (10 points):_____ **Part A (50 points):** [ ]

Part B:

Problem 6 (20 points):_____

Problem 7 (20 points):_____

Problem 8 (20 points):_____

Problem 9 (20 points):_____ **Part B (80 points):** [ ]

Total (130 points):_____

Note: Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

Note: Please be sure your name is recorded on each sheet of the exam.

I will not cheat on this exam.

_____
Signature

GOOD LUCK!
(HAVE A GREAT SEMESTER BREAK)

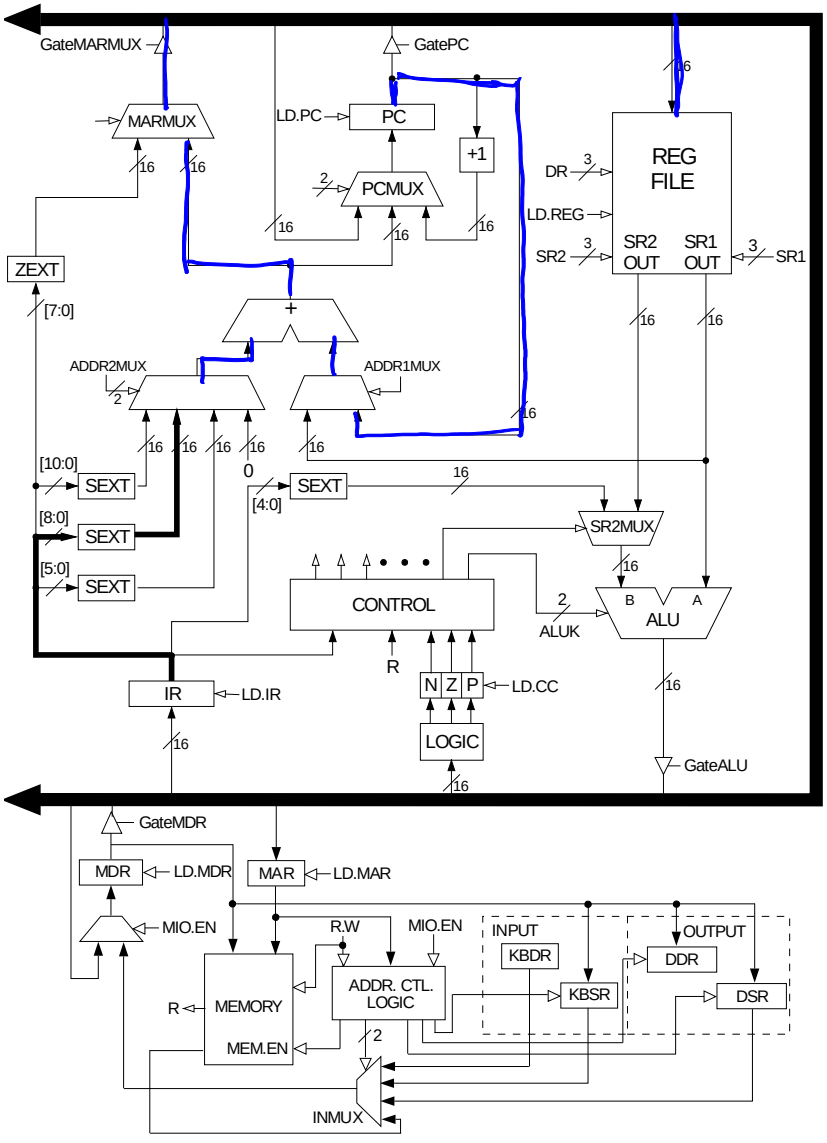
Name:_____

Part A, Problem 1. (10 points):

Part a. (5 points): We want to move a number from A to B. List all LC-3 opcodes that can be used to accomplish this in one instruction when A,B are as specified at the top of each column. We have provided four slots for each column. Use as many as you need.

A is memory location B is a register R0-R7	A is a register R0-R7 B is a register R0-R7	A is a register R0-R7 B is memory location	A is memory location B is memory location
LD	ADD	ST	
LDR	AND	STR	
LDI		STI	

Part b. (5 points): After an LC-3 instruction is decoded, it must be processed using the various paths in the data path. Show **all** paths in the data path that are used to process LEA after the instruction has been decoded by drawing a heavy line over each such path. For example, note the heavy line from the IR, through the sign-extended 9-bit value, and into the mux. Recall that LEA does not set the condition codes.



Name:_____

Part A, Problem 2. (10 points):
Consider the following semi-nonsense assembly language program:

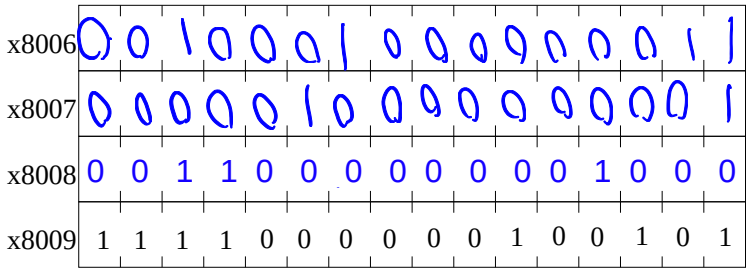
```
line 1:           .ORIG x8003
line 2:           AND R1,R1,#0
line 3:           ADD R0,R1,#5
line 4:           ST  R1,B
line 5:           LD  R1,A
line 6:           BRz SKIP
line 7:           ST  R0,B
line 8:    SKIP   TRAP x25
line 9:    A      .BLKW #7
line 10:   B      .FILL #5
line 11:   BANNER .STRINGZ "We are done!"
line 12:   C      .FILL x0
line 13:          .END
```

A separate module will store a value in A before the above program executes.

Part a. Construct the symbol table.

Part b. Show the result of assembly of lines 5 through 7 above. Note: the instruction at line 8 has already been assembled for you.

Symbol	Address
SKIP	x8009
A	x800A
B	x8011
BANNER	x8012
C	x801F



Part c. Note that two different things could cause location B to contain the value 5: the contents of line 7 or the contents of line 10. Explain the difference between line 7 causing the value 5 to be in location B and line 10 causing the value 5 to be in location B.

Line 7 stores 5 in B during execution of the program
Line 10 stores 5 in B at load time (assemble time is also acceptable)

Name:_____

Part A, Problem 3. (10 points): Memory locations x5000 to x5FFF contain 2's complement integers. What does the following program do?

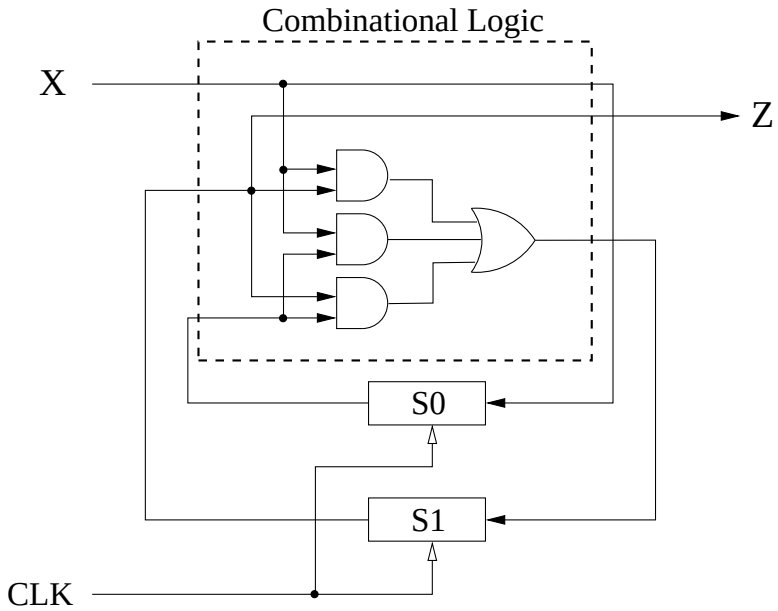
```
          .ORIG x3000
          LD  R1, ARRAY
          LD  R2, LENGTH
          AND R3, R3, #0
AGAIN     LDR R0, R1, #0
          AND R0, R0, #1
          BRz SKIP
          ADD R3, R3, #1
SKIP      ADD R1, R1, #1
          ADD R2, R2, #-1
          BRp AGAIN
          HALT
ARRAY    .FILL x5000
LENGTH   .FILL x1000
         .END
```

Please write your answer in the box below. Your answer must contain at most 15 words. Any words after the first 15 will NOT be considered in grading this problem.

Counts the number of odd numbers in the array.

Name:_____

Part A, Problem 4. (10 points): The logic circuit shown below has one input X, one output Z, and two state variables, s1 and s0. The circuit operates synchronously, controlled by a CLK signal, implementing a state machine.

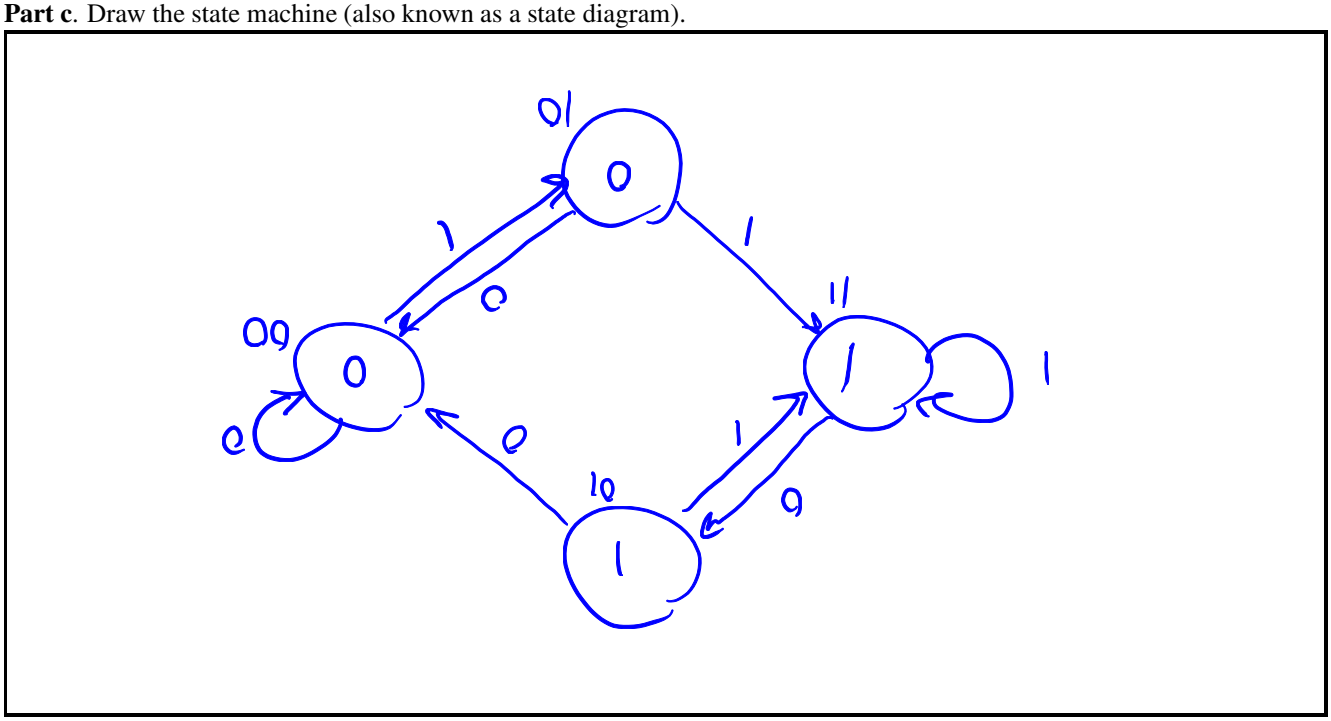


S1	S0	X	Z	S1'	S0'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	1	1	1

Part a. How many states in this state machine | 4 |

Part b. Fill in the truth table for each of the eight input combinations.

Part c. Draw the state machine (also known as a state diagram).



Name:_____

Part A, Problem 5. (10 points):

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	R2
Before execution starts	x3000	——	——	——	x0000	x0000	x0000
After the 1st instruction finishes	x3001	x3008	x0001	x2207	x0000	x0001	x0000
After the 2nd instruction finishes	x3002	x3009	xD635	x2007	xD635	x0001	x0000
After the 3rd instruction finishes	x3003	x3002	x0601	x0601	xD635	x0001	x0000
After the 4th instruction finishes	x3004	x3003	x1481	x1481	xD635	x0001	x0001
After the 5th instruction finishes	x3005	x3004	x1241	x1241	xD635	x0002	x0001

x2207 0010 0010 0000 0111 LD R1, PC+7

x2007 0010 0000 0000 0111 LD R0, PC+7

x0601 0000 0110 0000 0001 BRzp PC+1

x1481 0001 0100 1000 0001 ADD R2, R2, R1

x1241 0001 0010 0100 0001 ADD R1, R1, R1

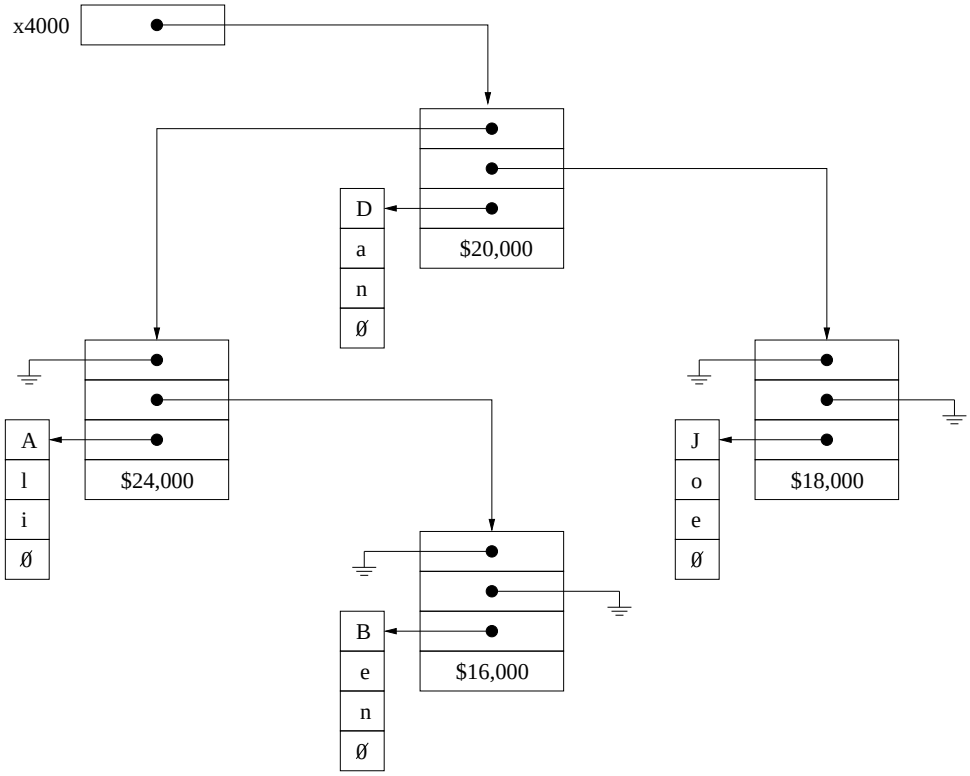
Name:_____

Part B, Problem 6. (20 points): Let's revisit Programming Lab 4, where you were asked to examine/search the nodes in a binary tree, looking for a professor's name. The search would have been done much faster if the tree had been a sorted binary tree.

First, what do we mean by a sorted binary tree: For every node A, ALL nodes in node A's left subtree must come before node A, and ALL nodes in node A's right subtree must come after node A.

If the binary tree is sorted, we can search for a match by starting at the root of the tree, and systematically examining nodes. Each such examination tells us either (a) we have a match, (b) we need to examine the node's left subtree (because the node we are looking for comes before the node we are examining), or (c) we need to examine the node's right subtree (because the node we are looking for comes after the node we are examining).

The figure below shows, like in Lab 4, our professors, only this time the binary tree is sorted in alphabetical order. Recall that each node contains four words. The 1st word points to its left subtree, the 2nd word points to its right subtree, the 3rd word points to the character string containing the professor's name, and the 4th word points to his/her salary.



On the next page, the subroutine SEARCH examines a sorted binary tree, looking for a match. R0 points to the root of the tree, R1 points to the name we are searching for, stored as a null-terminated character string. If the subroutine finds a match, it prints the professors's salary on the monitor. If the subroutine does not find a match, it prints "No Entry" on the monitor.

You will note that within the subroutine SEARCH, there are calls to two subroutines, COMPARE and PRINT_NUM. COMPARE compares two strings pointed to by R1 and R2. COMPARE puts a 0, 1, or -1 in R3 depending on whether the two strings are identical, the string pointed to by R2 comes after the string pointed to by R1, or the the string pointed to by R2 comes before the string pointed to by R1. PRINT_NUM prints the 2's complement integer in R0 to the console.

Your job: Fill in the missing instructions in SEARCH.

Name:_____

```
            .ORIG x3500
SEARCH      ST  R0, SAVER0
            ST  R2, SAVER2
            ST  R3, SAVER3
            ST  R7, SAVER7

            ADD R0, R0, #0
AGAIN       BRz NOT_FOUND
            LDR R2, R0, #2
            JSR COMPARE
            ADD R3, R3, #0
            BRz FOUND
            BRn RIGHT
            LDR R0, R0, #0
            BR  AGAIN
RIGHT       LDR R0, R0, #1
            BR  AGAIN

FOUND       LDR R0, R0, #3
            JSR PRINT_NUM
            BRnzp DONE

NOT_FOUND   LEA R0, NO_ENTRY
            TRAP x22
DONE        LD  R0, SAVER0
            LD  R2, SAVER2
            LD  R3, SAVER3
            LD  R7, SAVER7
            RET

NO_ENTRY    .STRINGZ "No Entry"
SAVER0      .BLKW #1
SAVER2      .BLKW #1
SAVER3      .BLKW #1
SAVER7      .BLKW #1
            .END
```

Name:_____

Part B, Problem 7. (20 points): In this problem we wish to examine the effects of keyboard interrupts while a main program is running. The program running is shown below, starting at location x3000. The keyboard interrupt service routine is also shown below, starting at location x1200.

x3000 1001 1111 1100

```
;          Main Program

           .ORIG x3000
           AND R1, R1, #0
LOOP       ADD R1, R1, #1
           LD R0, NEG_COUNT
           ADD R0, R0, R1
           BRn LOOP
           AND R1, R1, #0
           BR LOOP
NEG_COUNT  .FILL #-9
           .END
```

6 6 9 6 7 ; x5260 ; x1001 ; x09FC

```
;      Interrupt Service Routine

           .ORIG x1200
           ST R0, SAVER0
           ST R1, SAVER1
           LDI R0, KBDR  ;Set KBSR[15] = 0
           LD R0, ASCII
           ADD R0, R0, R1
POLL       LDI R1, DSR
           BRzp POLL
           STI R0, DDR
           LD R0, SAVER0
           LD R1, SAVER1
           RTI
ASCII      .FILL x30
KBDR       .FILL xFE02
DSR        .FILL xFE04
DDR        .FILL xFE06
SAVER0     .BLKW #1
SAVER1     .BLKW #1
           .END
```

Part a. What is the program starting at x3000 doing? ...in at most 15 words.

Continuously counts from 0 to 9 in R1 in an infinite loop

Part b. What does the interrupt service routine do? ...also in at most 15 words.

prints the current digit in R1

Problem continued on next page.

Part c. The partially completed table shows a snapshot of the LC-3 **DURING** several cycles of the computer's execution. You can assume the LC-3 has been properly initialized, that is, IE=1, SP is the user stack, and the interrupt vector table contains the necessary entries before the main program starts executing in cycle 1. Memory operations take 2 cycles each. The keyboard is pressed EXACTLY once during the execution of the main program.

Your job: fill in the missing entries in the table.

Cycle Number	State Number	Information
1	18	LD.PC: 1 LD.MDR: 0 GateMDR: 0 LD.MAR: 1 GateALU: 0 GatePC: 1
7	18	PCMUX: PC+1 MDR: x5260 IR: x5260 LD.IR: 0
27	1	LD.REG: 1 MDR: x1001 DRMUX: IR11.9 BUS: -8 (xFFF8) GateALU: 1 GateMDR: 0
33	0	PC: x3005 IR: x09FC
42	47	LD.MAR: 1 DRMUX: SP MDR: x3001 LD.MDR: 0 LD.REG: 1
49	18	PC: x1200 LD.PC: 1 MDR: x1200 IR: x09FC

Part d. The above table shows that the main program was interrupted by someone striking the keyboard sometime between cycle 29 and cycle 34

OR 28 35 for exclusive range

Name:_____

Part B, Problem 8. (20 points): It is easier to identify borders between cities on a map if adjacent cities are colored with different colors. For example, in a map of Texas, one would not color Austin and Pflugerville with the same color, since doing so would obscure the border between the two cities.

Shown below is the recursive subroutine EXAMINE. EXAMINE examines the data structure representing a map to see if any pair of adjacent cities have the same color. Each node in the data structure contains the city's color and the addresses of the cities it borders. If no pair of adjacent cities have the same color, EXAMINE returns the value 0 in R1. If at least one pair of adjacent cities have the same color, EXAMINE returns the value 1 in R1. The main program supplies the address of a node representing one of the cities in R0 before executing JSR EXAMINE.

```
        .ORIG x4000
EXAMINE ADD R6, R6, #-1
        STR R0, R6, #0
        ADD R6, R6, #-1
        STR R2, R6, #0
        ADD R6, R6, #-1
        STR R3, R6, #0
        ADD R6, R6, #-1
        STR R7, R6, #0

        AND R1, R1, #0        ; Initialize output R1 to 0
        LDR R7, R0, #0
        BRn RESTORE           ; Skip this node if it has already been visited

        LD  R7, BREADCRUMB
        STR R7, R0, #0        ; Mark this node as visited
        LDR R2, R0, #1        ; R2 = color of current node
        ADD R3, R0, #2

AGAIN   LDR R0, R3, #0        ; R0 = neighbor node address
        BRz RESTORE
        LDR R7, R0, #1
        NOT R7, R7            ; <-- Breakpoint here
        ADD R7, R7, #1
        ADD R7, R2, R7        ; Compare current color to neighbor's color
        BRz BAD
        JSR EXAMINE           ; Recursively examine the coloring of next neighbor
        ADD R1, R1, #0
        BRp RESTORE           ; If neighbor returns R1=1, this node should return R1=1
        ADD R3, R3, #1
        BR  AGAIN             ; Try next neighbor

BAD     ADD R1, R1, #1
RESTORE LDR R7, R6, #0
        ADD R6, R6, #1
        LDR R3, R6, #0
        ADD R6, R6, #1
        LDR R2, R6, #0
        ADD R6, R6, #1
        LDR R0, R6, #0
        ADD R6, R6, #1
        RET

BREADCRUMB .FILL x8000
        .END
```

Name:_____

Your job is to construct the data structure representing a particular map. Before executing JSR EXAMINE, R0 is set to x6100 (the address of one of the nodes), and a breakpoint is set at x4012. The table below shows relevant information collected each time the breakpoint was encountered during the running of EXAMINE.

PC	R0	R2	R7
x4012	x6200	x0042	x0052
x4012	x6100	x0052	x0042
x4012	x6300	x0052	x0047
x4012	x6200	x0047	x0052
x4012	x6400	x0047	x0052
x4012	x6100	x0052	x0042
x4012	x6300	x0052	x0047
x4012	x6500	x0052	x0047
x4012	x6100	x0047	x0042
x4012	x6200	x0047	x0052
x4012	x6400	x0047	x0052
x4012	x6500	x0052	x0047
x4012	x6400	x0042	x0052
x4012	x6500	x0042	x0047

Construct the data structure for the particular map that corresponds to the relevant information obtained from the breakpoints. Note: We are asking you to construct the data structure as it exists AFTER the recursive subroutine has executed.

x6100	x8000
x6101	x0042
x6102	x6200
x6103	x6400
x6104	x6500
x6105	x0000
x6106	

x6200	x8000
x6201	x0052
x6202	x6100
x6203	x6300
x6204	x6500
x6205	x0000
x6206	

x6300	x8000
x6301	x0047
x6302	x6200
x6303	x6400
x6304	x0000
x6305	
x6306	

x6400	x8000
x6401	x0052
x6402	x6100
x6403	x6300
x6404	x6500
x6405	x0000
x6406	

x6500	x8000
x6501	x0047
x6502	x6100
x6503	x6200
x6504	x6400
x6505	x0000
x6506	

Name:_____

Problem 9. (20 points):

Up to now, we have only had one output device, the monitor, with xFE04 and xFE06 used to address its two device registers. We now introduce a second output device, a light that requires a single device register, to which we assign the address xFE08. Storing a 1 in xFE08 turns the light on, storing a 0 in xFE08 turns the light off.

An Aggie decided to write a program which would control this light by a keyboard interrupt as follows: Pressing the key 0 would turn the light off. Pressing the key 1 would cause the light to flash on and off repeatedly. Shown below is the Aggie's code, and his Keyboard interrupt service routine.

```
The User Program:
             .ORIG x3000
0             LEA  R7, LOOP
1     LOOP    LDI  R0, ENABLE
2             LD   R1, NEG_OFF
3             ADD  R0, R0, R1   ; check if switch is on
4             BRnp BLINK
      ;
5             AND R0, R0, #0
6             STI R0, LIGHT     ; turn light off
7             RET
      ;
8     BLINK   ST  R7, SAVE_R7   ; save linkage
9             LDI R0, LIGHT
A             ADD R0, R0, #1
B             AND R0, R0, #1    ; toggle LIGHT between 0 and 1
C             STI R0, LIGHT
D             JSR DELAY         ; 1 second delay
E             LD R7, SAVE_R7
F             RET               ; <-- Breakpoint here
      ;
      LIGHT   .FILL xFE08
      ENABLE  .FILL x4000
      NEG_OFF .FILL x-30
      SAVE_R7 .BLKW #1
              .END

The Keyboard Interrupt Routine:
              .ORIG   x1500
0             ADD R6, R6, #-1   ; <-- Breakpoint here
1             STR R0, R6, #0    ; save R0 on stack
2             ADD R6, R6, #-1
3             STR R7, R6, #0    ; save R7 on stack
      ;
4             TRAP x20
5             STI  R0, ENABLE2
      ;
6             RTI               ; <-- Breakpoint here
7     ENABLE2 .FILL x4000
              .END
```

The DELAY subroutine was inserted in his program in order to separate the turning on and off of the light by one second in order to make the on-off behavior visible to the naked eye. The DELAY subroutine does not modify any registers.

Unfortunately, per usual, the Aggie made a mistake in his program, and things do not work as he intended. So, he decided to debug his program (see the next page).

Name:_____

He set three breakpoints, at x1500, at x1506, and at x300F. He initialized the PC to x3000, the keyboard IE bit to 1, and memory location x0180 to x1500.

Then he hit the Run button, which stopped executing when the PC reached x1500. He hit the Run button three more times, each time the computer stopping when the PC reached a breakpoint. While the program was running, he pressed a key on the keyboard EXACTLY ONCE.

The table below shows the data in various registers and memory locations each time a breakpoint was encountered. Note: Assume, when an interrupt is initiated, the PSR is pushed onto the system stack before the PC.

Your Job: complete the table.

	Initial	**Breakpoint 1**	**Breakpoint 2**	**Breakpoint 3**	**Breakpoint 4**
PC	x3000	x1500	x1506	x1506	x300F
R0	x1234	x0001	x0030	x0000	x0000
R6	x3000	x2FFE	x2FFC	x2FFE	x3000
R7	x1234	x3001	x1505	x1505	x3001
M[x2FFC]	x0000	x0000	x3001	x3001	x3001
M[x2FFD]	x0000	x0000	x0001	x0001	x0001
M[x2FFE]	x0000	x300D	x300D	x300D	x300D
M[x2FFF]	x0000	x8001	x8001	x8001	x8001
M[x4000]	x0031	x0031	x0030	x0000	x0000
M[xFE00]	x4000	xC000	x4000	x4000	x4000

Data Path Control Signals

Signal Name	Signal Values
LD.MAR/1:	NO(0), LOAD(1)
LD.MDR/1:	NO(0), LOAD(1)
LD.IR/1:	NO(0), LOAD(1)
LD.REG/1:	NO(0), LOAD(1)
LD.PC/1:	NO(0), LOAD(1)
Gate.PC/1:	NO(0), YES(1)
Gate.MDR/1:	NO(0), YES(1)
Gate.ALU/1:	NO(0), YES(1)
PCMUX/2:	PC+1(00), BUS(01), ADDER(10)
DRMUX/2:	IR11.9(00), R7(01), SP(10)

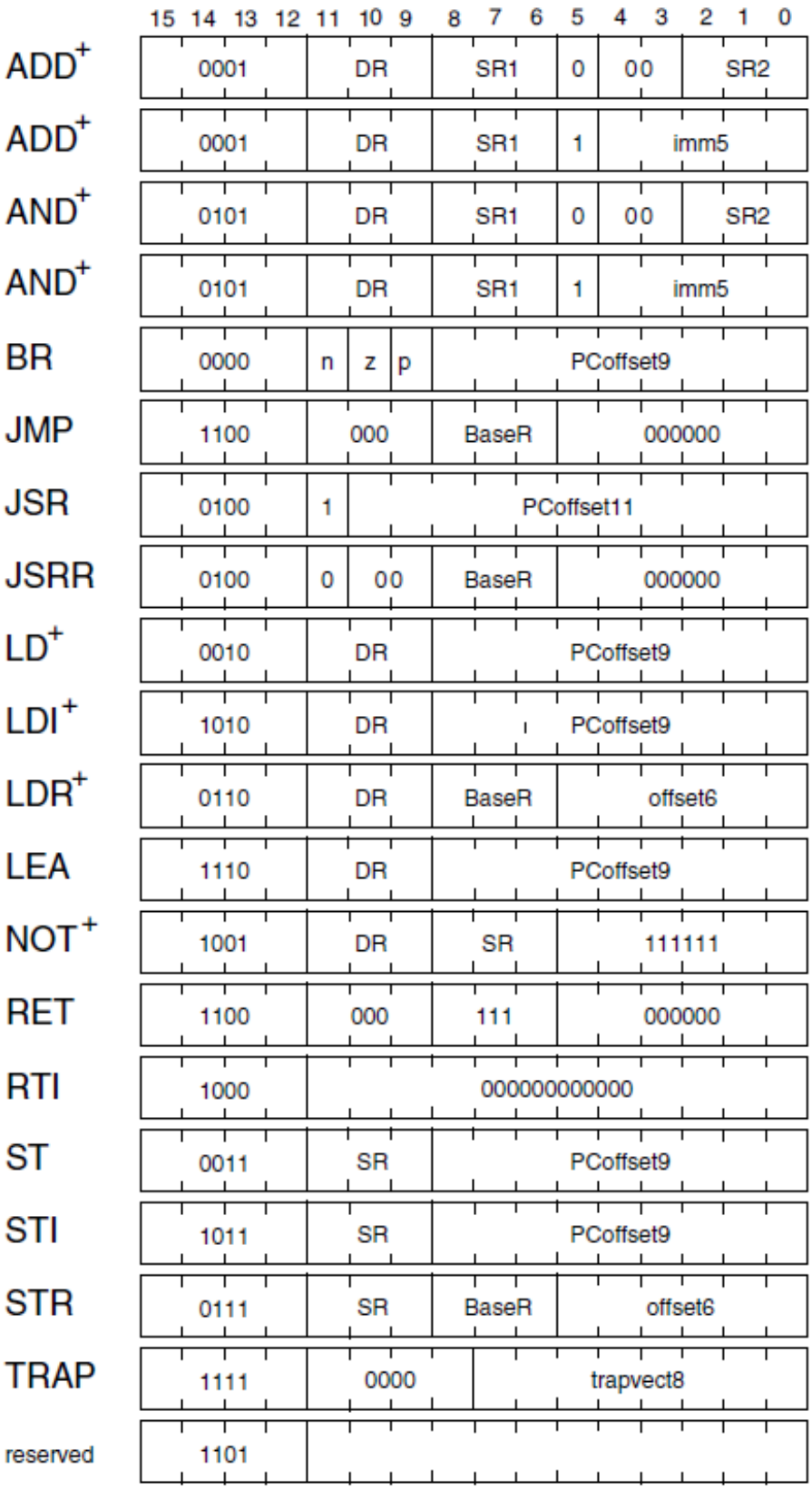
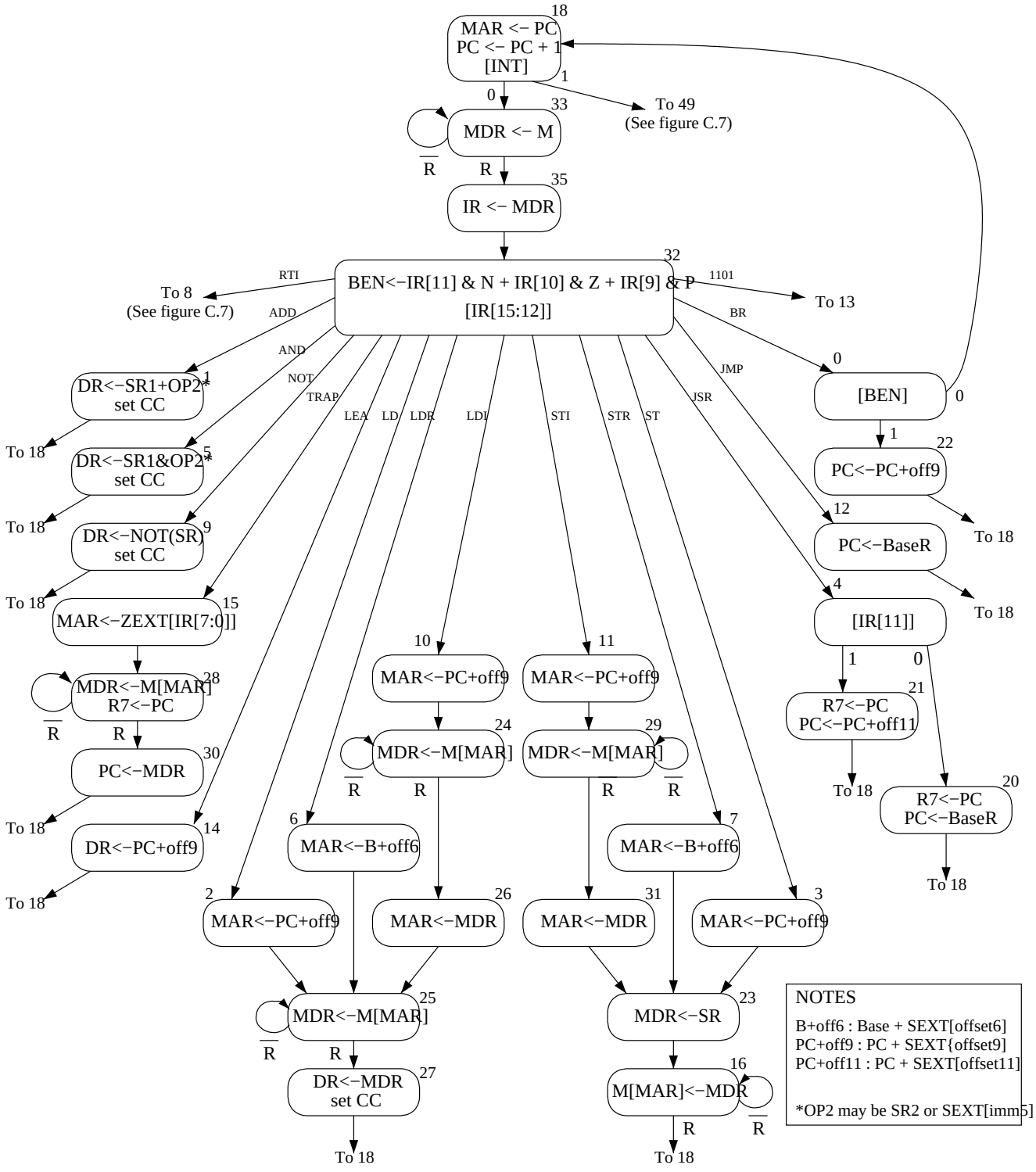


Figure A.2 Format of the entire LC-3 instruction set. **Note:** + indicates instructions that modify condition codes


18
MAR <− PC
PC <− PC + 1
[INT]
1
To 49
(See figure C.7)
0
33
MDR <− M
R
R
35
IR <− MDR
32
BEN<−IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]
RTI
To 8
(See figure C.7)
ADD
AND
NOT
TRAP
LEA
LD
LDR
LDI
STI
STR
ST
JSR
JMP
BR
1101
To 13
0
[BEN]
0
1
22
PC<−PC+off9
To 18
12
PC<−BaseR
To 18
4
[IR[11]]
1
0
21
R7<−PC
PC<−PC+off11
To 18
20
R7<−PC
PC<−BaseR
To 18
1
DR<−SR1+OP2*
set CC
To 18
5
DR<−SR1&OP2*
set CC
To 18
9
DR<−NOT(SR)
set CC
To 18
15
MAR<−ZEXT[IR[7:0]]
28
MDR<−M[MAR]
R7<−PC
R
R
30
PC<−MDR
To 18
14
DR<−PC+off9
To 18
10
MAR<−PC+off9
24
MDR<−M[MAR]
R
R
6
MAR<−B+off6
2
MAR<−PC+off9
26
MAR<−MDR
25
MDR<−M[MAR]
R
R
27
DR<−MDR
set CC
To 18
11
MAR<−PC+off9
29
MDR<−M[MAR]
R
R
7
MAR<−B+off6
31
MAR<−MDR
3
MAR<−PC+off9
23
MDR<−SR
16
M[MAR]<−MDR
R
R
To 18
NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT{offset9]
PC+off11 : PC + SEXT[offset11]
*OP2 may be SR2 or SEXT[imm5]

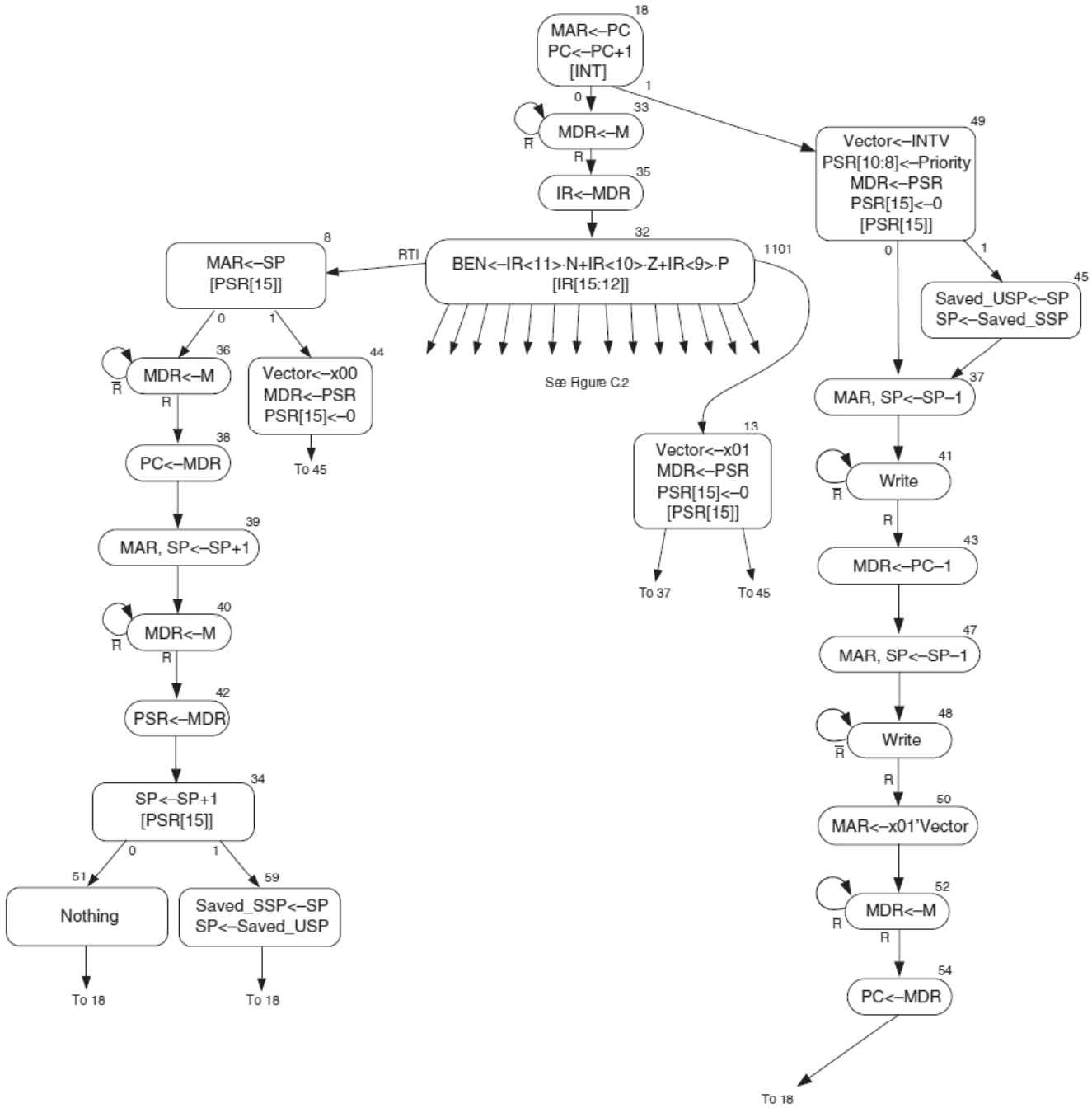
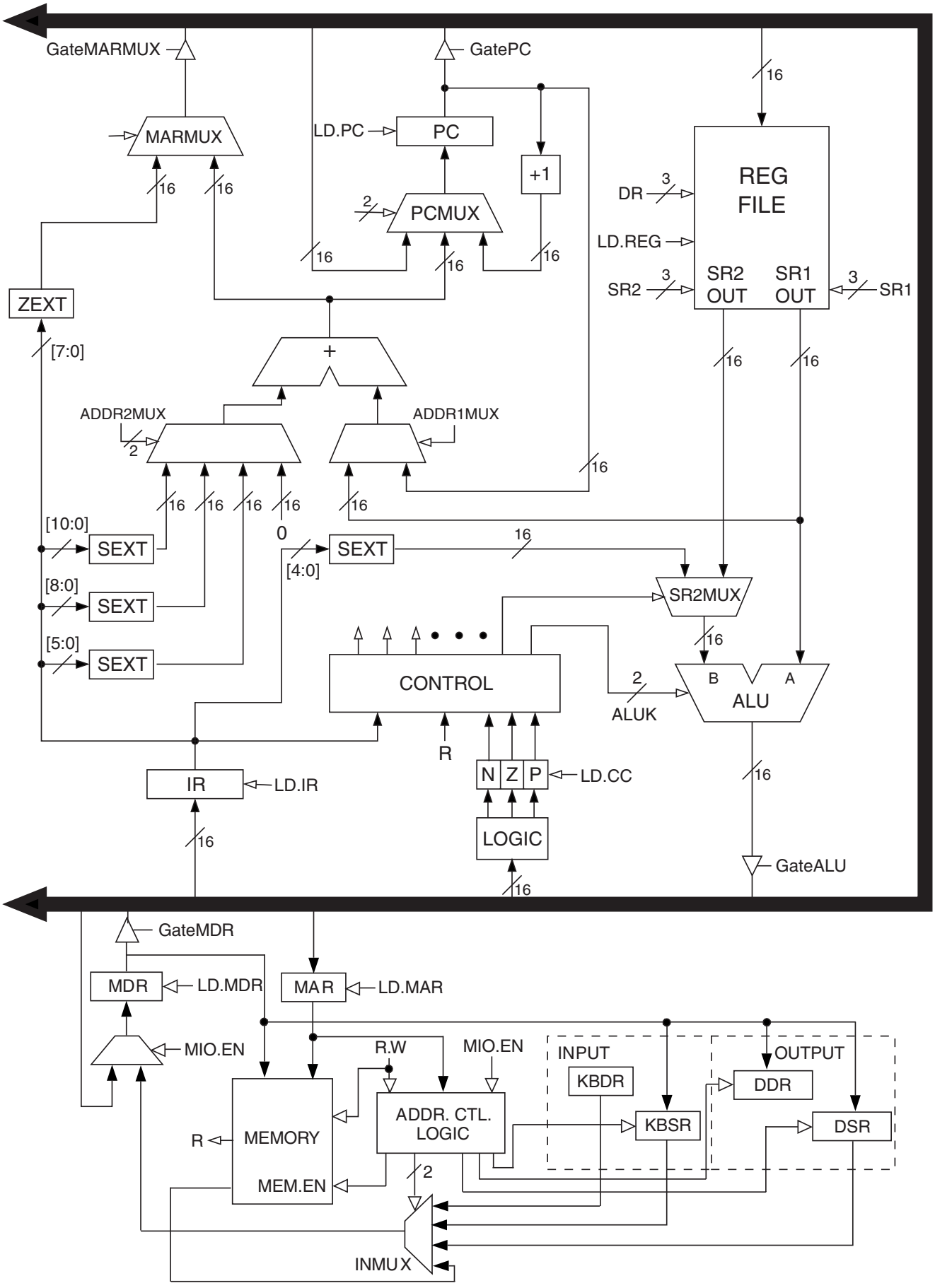
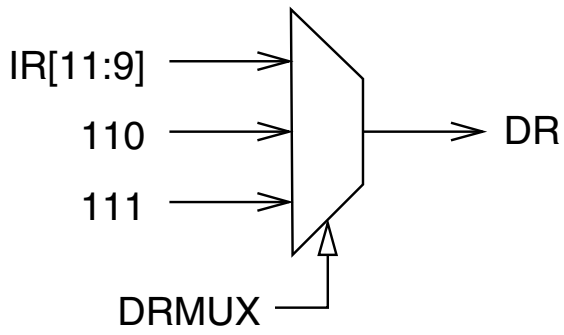
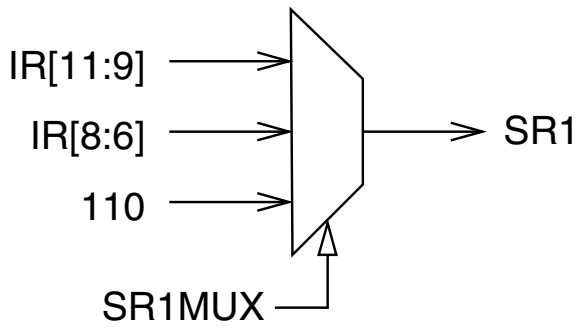


Figure C.7 LC-3 state machine showing interrupt control

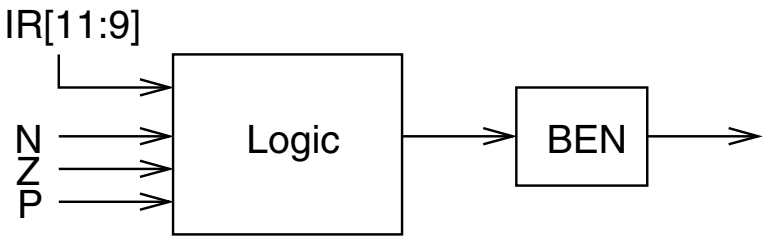

GateMARMUX
GatePC
16
MARMUX
LD.PC
PC
+1
REG
FILE
DR
3
16
16
2
PCMUX
LD.REG
16
16
16
SR2
OUT
SR1
OUT
3
SR2
3
SR1
ZEXT
[7:0]
+
16
16
ADDR2MUX
ADDR1MUX
2
16
16
16
16
16
16
[10:0]
SEXT
0
SEXT
16
[4:0]
[8:0]
SEXT
SR2MUX
16
[5:0]
SEXT
CONTROL
2
B
A
ALU
ALUK
R
N
Z
P
LD.CC
IR
LD.IR
16
LOGIC
16
16
GateALU
GateMDR
MDR
LD.MDR
MAR
LD.MAR
MIO.EN
R.W
MIO.EN
INPUT
KBDR
OUTPUT
DDR
R
MEMORY
MEM.EN
ADDR. CTL.
LOGIC
KBSR
DSR
2
INMUX


IR[11:9]
110
111
DR
DRMUX

(a)


IR[11:9]
IR[8:6]
110
SR1
SR1MUX

(b)


IR[11:9]
N
Z
P
Logic
BEN

(c)

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.SavedSSP/1:	NO, LOAD	
LD.SavedUSP/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
VectorMUX/2:	INTV	
	Priv.exception	
	Opc.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

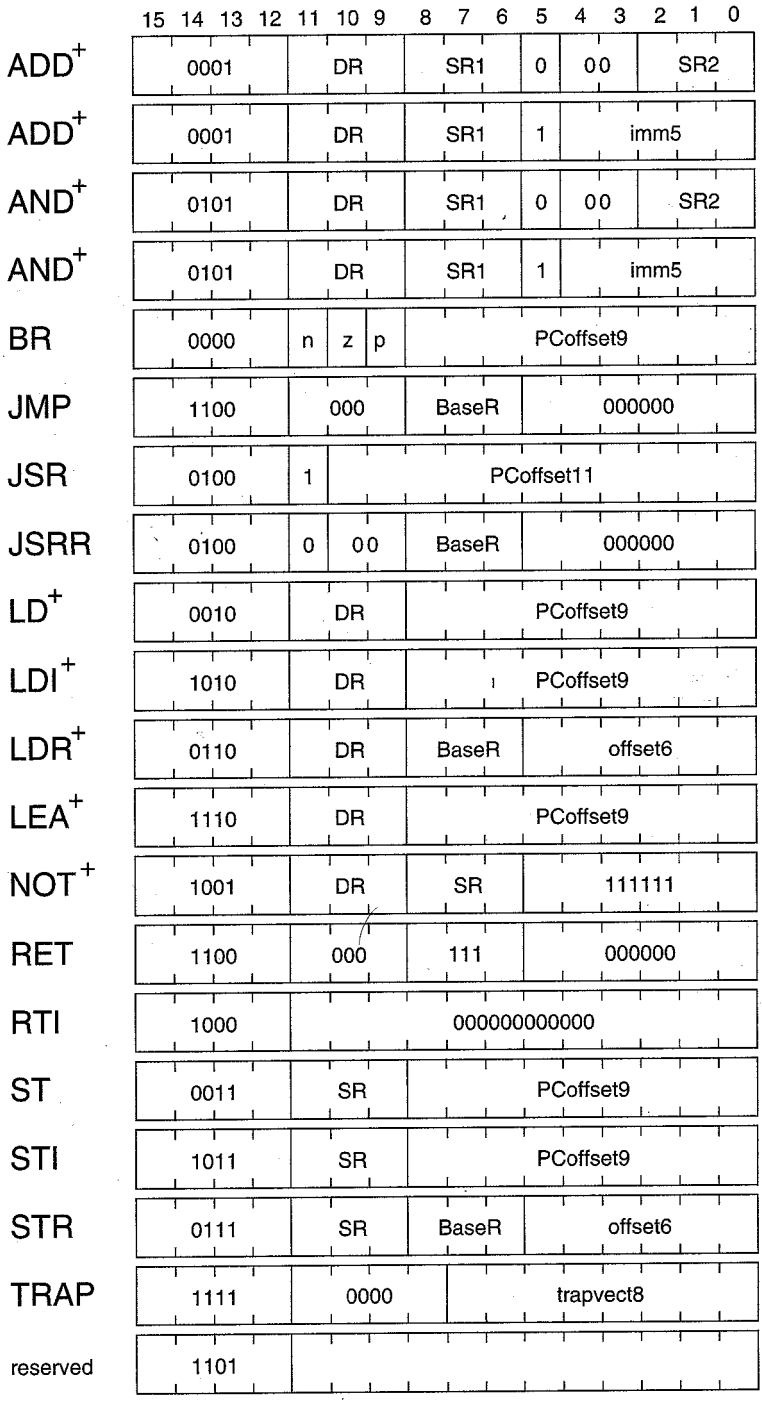


Figure A.2 Format of the entire LC-3 instruction set. **Note:** + indicates instructions that modify condition codes

The Standard ASCII Table

ASCII			ASCII			ASCII			ASCII		
Character	Dec	Hex	Character	Dec	Hex	Character	Dec	Hex	Character	Dec	Hex
nul	0	00	sp	32	20	@	64	40	`	96	60
soh	1	01	!	33	21	A	65	41	a	97	61
stx	2	02	"	34	22	B	66	42	b	98	62
etx	3	03	#	35	23	C	67	43	c	99	63
eot	4	04	$	36	24	D	68	44	d	100	64
enq	5	05	%	37	25	E	69	45	e	101	65
ack	6	06	&	38	26	F	70	46	f	102	66
bel	7	07	'	39	27	G	71	47	g	103	67
bs	8	08	(	40	28	H	72	48	h	104	68
ht	9	09	)	41	29	I	73	49	i	105	69
lf	10	0A	*	42	2A	J	74	4A	j	106	6A
vt	11	0B	+	43	2B	K	75	4B	k	107	6B
ff	12	0C	,	44	2C	L	76	4C	l	108	6C
cr	13	0D	-	45	2D	M	77	4D	m	109	6D
so	14	0E	.	46	2E	N	78	4E	n	110	6E
si	15	0F	/	47	2F	O	79	4F	o	111	6F
dle	16	10	0	48	30	P	80	50	p	112	70
dc1	17	11	1	49	31	Q	81	51	q	113	71
dc2	18	12	2	50	32	R	82	52	r	114	72
dc3	19	13	3	51	33	S	83	53	s	115	73
dc4	20	14	4	52	34	T	84	54	t	116	74
nak	21	15	5	53	35	U	85	55	u	117	75
syn	22	16	6	54	36	V	86	56	v	118	76
etb	23	17	7	55	37	W	87	57	w	119	77
can	24	18	8	56	38	X	88	58	x	120	78
em	25	19	9	57	39	Y	89	59	y	121	79
sub	26	1A	:	58	3A	Z	90	5A	z	122	7A
esc	27	1B	;	59	3B	[	91	5B	{	123	7B
fs	28	1C	<	60	3C	\	92	5C	|	124	7C
gs	29	1D	=	61	3D	]	93	5D	}	125	7D
rs	30	1E	>	62	3E	^	94	5E	~	126	7E
us	31	1F	?	63	3F	_	95	5F	del	127	7F

Table A.2 Trap Service Routines

Trap Vector	Assembler Name	Description
x20	GETC	Read a single character from the keyboard. The character is not echoed onto the console. Its ASCII code is copied into R0. The high eight bits of R0 are cleared.
x21	OUT	Write a character in R0[7:0] to the console display.
x22	PUTS	Write a string of ASCII characters to the console display. The characters are contained in consecutive memory locations, one character per memory location, starting with the address specified in R0. Writing terminates with the occurrence of x0000 in a memory location.
x23	IN	Print a prompt on the screen and read a single character from the keyboard. The character is echoed onto the console monitor, and its ASCII code is copied into R0. The high eight bits of R0 are cleared.
x24	PUTSP	Write a string of ASCII characters to the console. The characters are contained in consecutive memory locations, two characters per memory location, starting with the address specified in R0. The ASCII code contained in bits [7:0] of a memory location is written to the console first. Then the ASCII code contained in bits [15:8] of that memory location is written to the console. (A character string consisting of an odd number of characters to be written will have x00 in bits [15:8] of the memory location containing the last character to be written.) Writing terminates with the occurrence of x0000 in a memory location.
x25	HALT	Halt execution and print a message on the console.

Table A.3 Device Register Assignments

Address	I/O Register Name	I/O Register Function
xFE00	Keyboard status register	Also known as KBSR. The ready bit (bit [15]) indicates if the keyboard has received a new character.
xFE02	Keyboard data register	Also known as KBDR. Bits [7:0] contain the last character typed on the keyboard.
xFE04	Display status register	Also known as DSR. The ready bit (bit [15]) indicates if the display device is ready to receive another character to print on the screen.
xFE06	Display data register	Also known as DDR. A character written in the low byte of this register will be displayed on the screen.
xFFFE	Machine control register	Also known as MCR. Bit [15] is the clock enable bit. When cleared, instruction processing stops.