LC-3 Overview: Memory and Registers

Memory

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

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 (PSR)

5-3

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, RTI, TRAP
- some opcodes set/clear *condition codes*, based on result:
  - N = negative, Z = zero, P = positive (> 0) → PSR.CC

Data Types

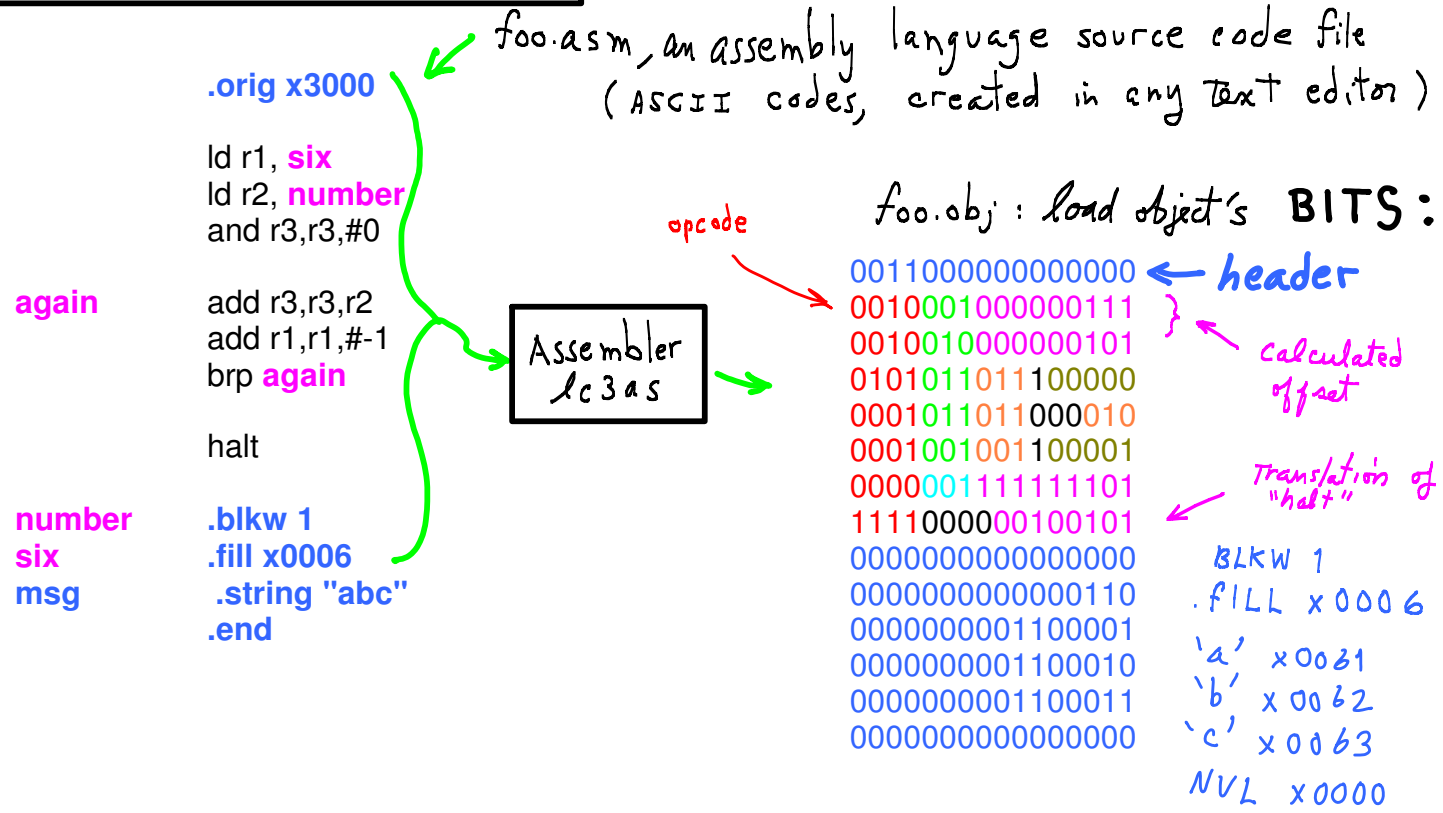
- 16-bit 2's complement integer

Addressing Modes

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

Assembly Language

P&P, Figure 7.1



Assembler (lc3as) **Directives** (to control the assembly process):
.orig: puts a load address into the .obj load-object file's header.
.end: tells assembler, this is the end of source code.
.blkw: tells assembler, create *n* blank words (all zeroes).
.fill: tells assembler, put these bits into a word.
.string: convert text to .FILL w/ one ascii code per word, NUL terminated.

The assembler produces machine code words:
--- ONE PER LINE expressing an LC3 instruction
--- ONE PER LINE where there is a .fill directive
--- n PER LINE where there is a .blkw directive
The assembler also calculates offsets for us using **symbols**. Symbols stand for memory addresses (starting for the .orig address). Offsets are calculated by subtraction. Symbols refer to the next instruction's location.

f.asm

```
        .orig x0200
main:
        ADD R1, R2, R3
        ADD R4, R5, var
foo:    .FILL x1234
var:    .FILL x012F
        .end
```

assemble LC3as

f.obj

```
0000001000000000
0001001001000011
0001100101100001
0001001000110100
0000000100101111
```

load

Mem

```
0200: 0001001001000011
0201: 0001100101100001
0202: 0001001000110100
0203: 0000000100101111
```

PC

203
-(201+1)
1

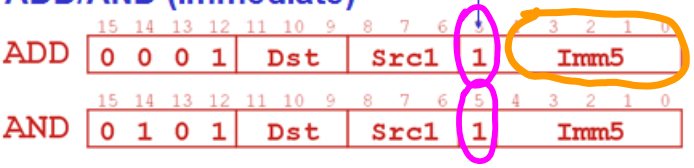
NOT (Register)

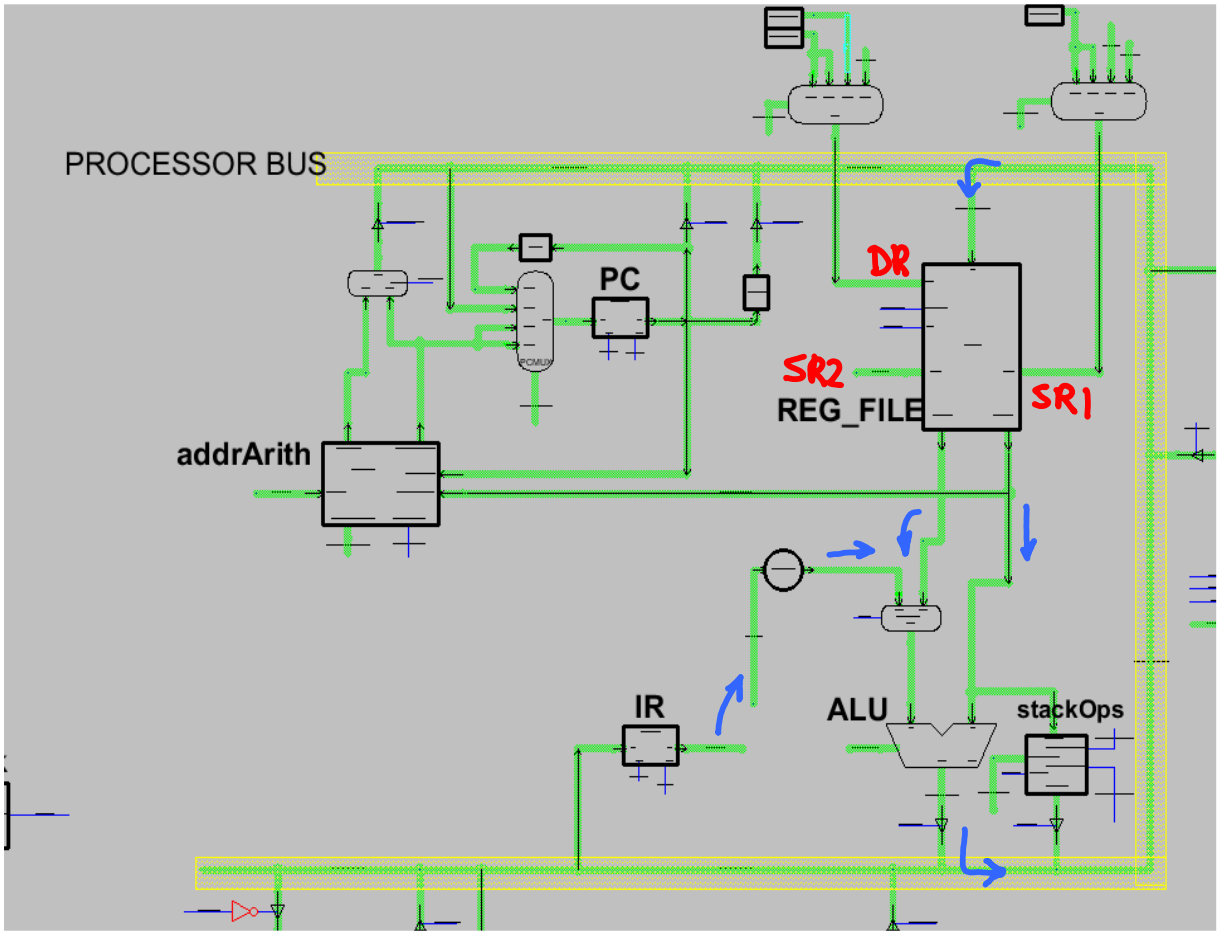
ADD/AND (Register)

this zero means "register mode"

ADD/AND (Immediate)

this one means "immediate mode"

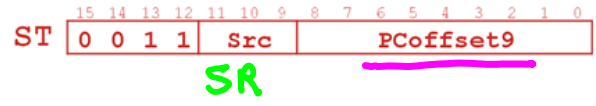




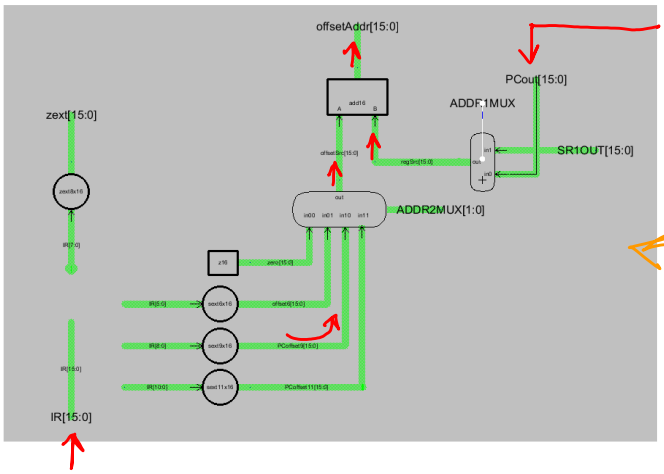
LD (PC-Relative)

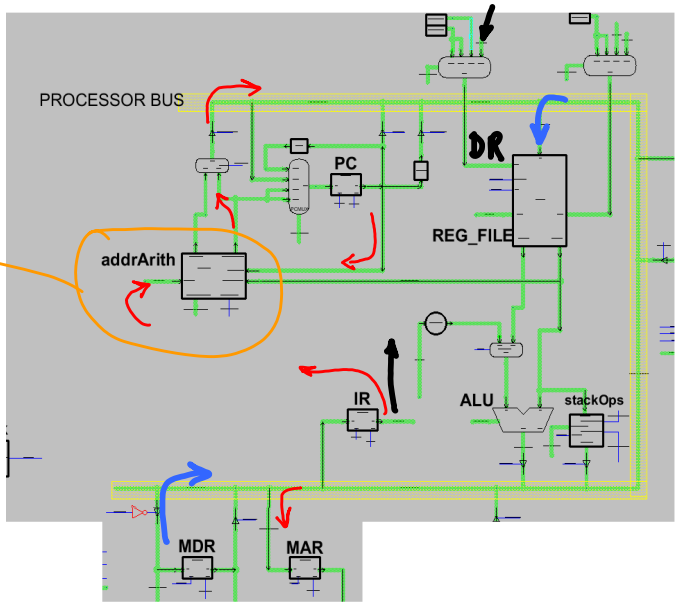
DR

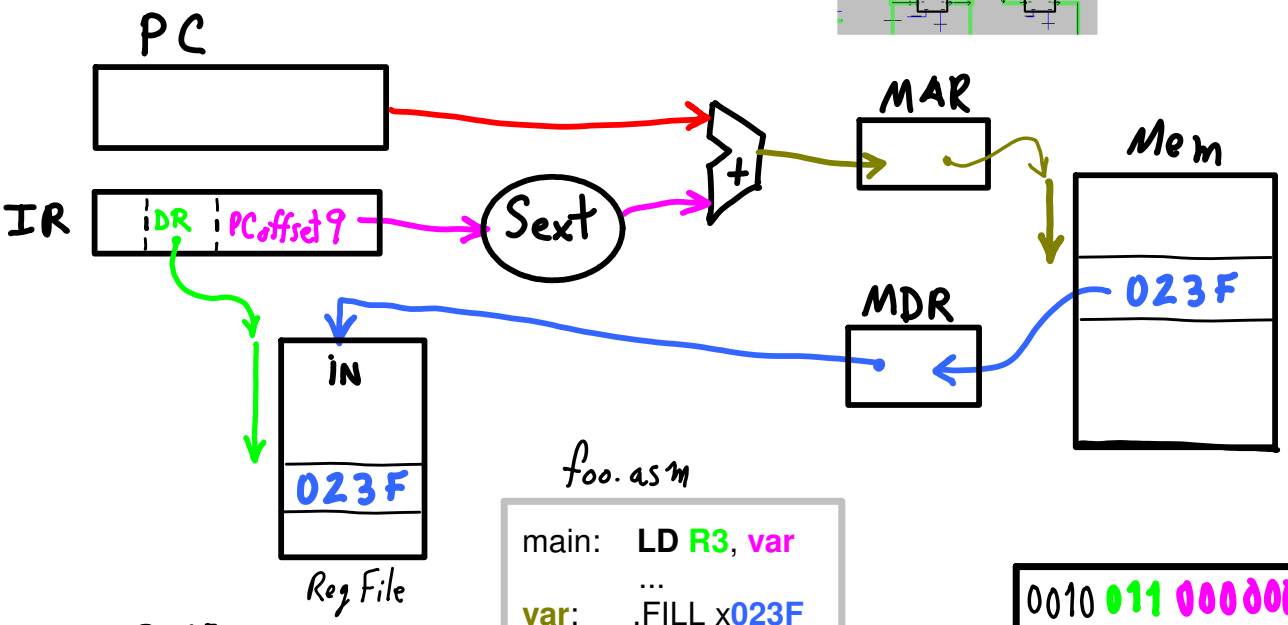
ST (PC-Relative)



SR







foo.asm

```
main:   LD R3, var
        ...
var:    .FILL x023F
```

ST

symbol	value
"main"	x0200
"var"	x0208

LC3as load

Mem

PC →	0200:	0010 011 000000111
	...	...
	0208:	0000 0010 0011 1111

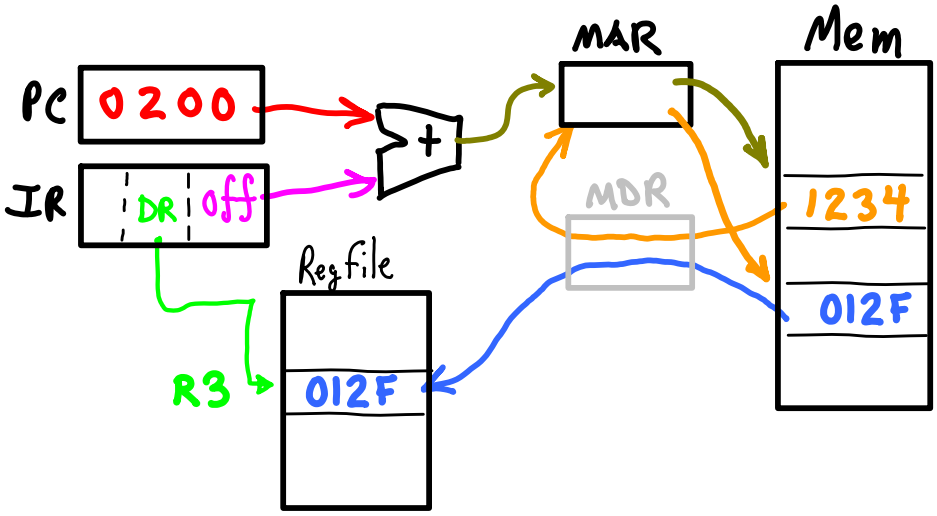
0010 011 000000111 IR

208
-201
7

LDI (Indirect)

f.asm

```
main:    LDI R3, dataPtr
         ...
dataPtr: .FILL var
         ...
var:     .FILL x0000
```



ST

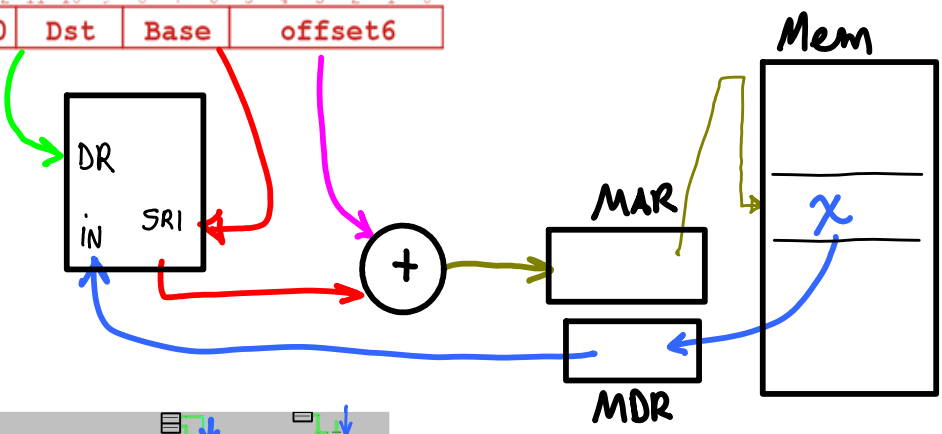
"main"	x0200
"dataPtr"	x0210
"var"	x1234

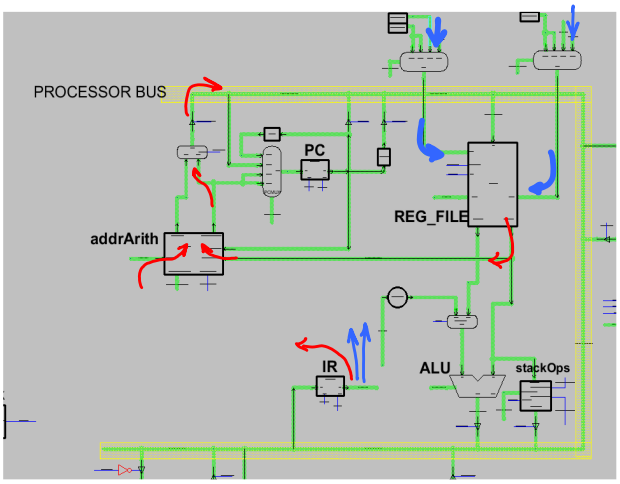
210
-(200+1)
F

Mem

```
0200:  1010 011 000001111
       ...
0210:  0001 0010 0011 0100
       ...
1234:  0000 0001 0010 1111
```

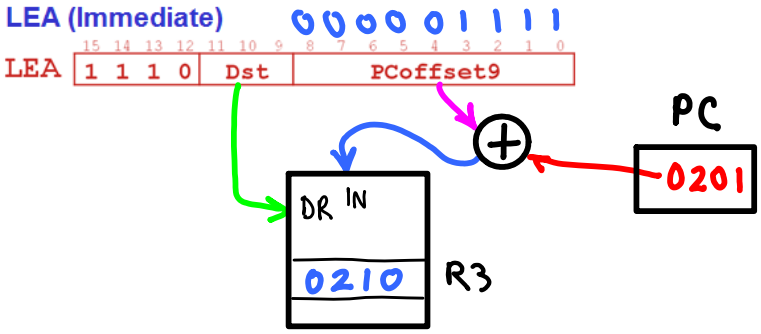
LDR (Base+Offset)






```
main:    LD R2, tablePTR
         LDR R1, R2, #2
                ...
tablePTR:
         .FILL table
                ...
table:   .FILL x0000
         .FILL x0001
         .FILL x0002
```

"main"	x0200
"tablePTR"	x02F0
"table"	xFF00

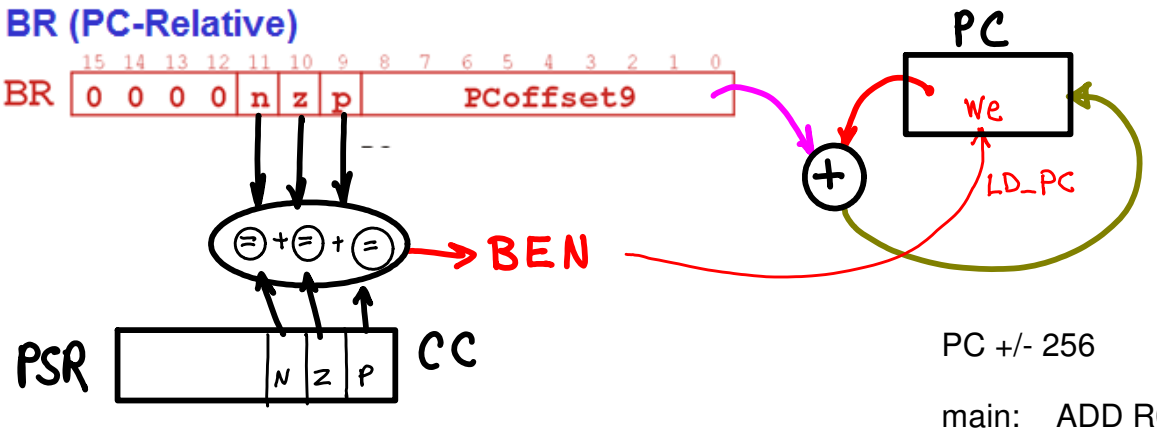


210
- 201
F

main: LEA R3, array
...
array: .BLKW 100

Symbol Table
"main" x0200
"array" x0210

Memory
0200: 1110 011 000001111
... ...
0210: ????
0211: ????
... ...

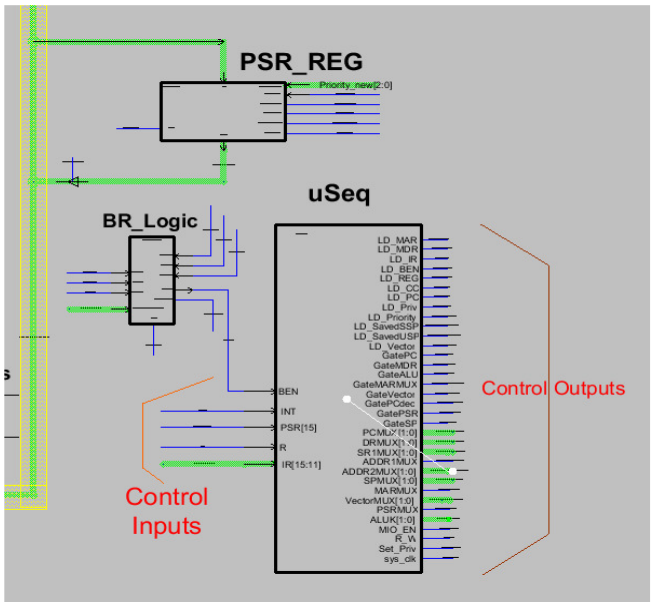


PC +/- 256

main: ADD R0, R0, 1
BRp main

Symbol Table
"main" x0200

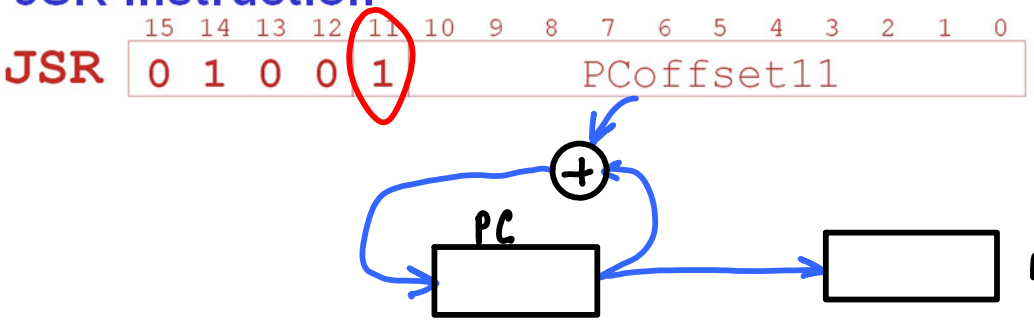
Memory
0200: 0001 000 000 0 00001
0201: 0000 001 111111110



200
- 202
-2

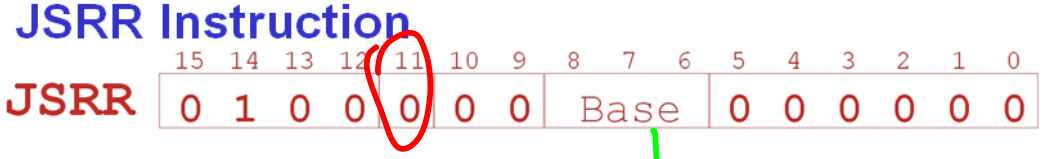
➢ only certain instructions set the codes
(ADD, AND, NOT, LD, LDI, LDR, LEA)

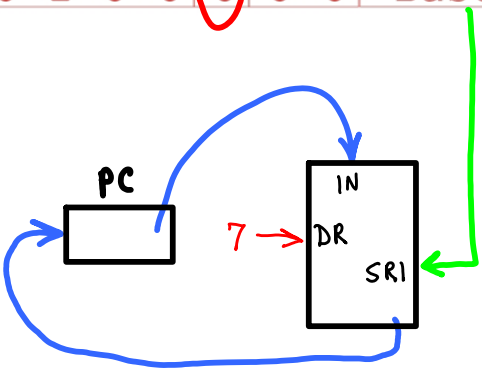
JSR Instruction



"RET" is a synonym for "JMP R7"

JSRR Instruction

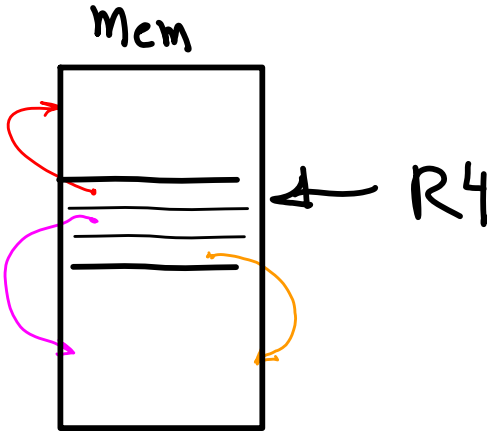




linking

```
        ...
        .EXTERNAL SQRT
        ...
        LD    R2, SQAddr
        JSRR R2
        ...
SQAddr  .FILL SQRT
```

Not supported by LC3 assembler, lc3as, but see lcc, C compiler for LC3.

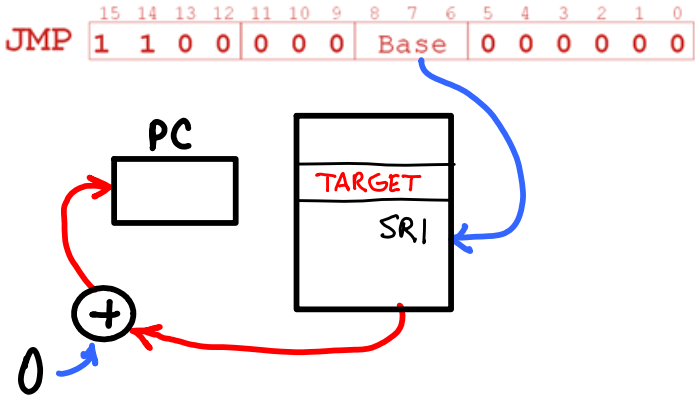


Problem

Jumping or accessing data **far away** requires having the distant address available. LD can get the address into a register, then **JMP REG or LDR** can reference the distant location. But then LD must use w/ a local pointer variable.

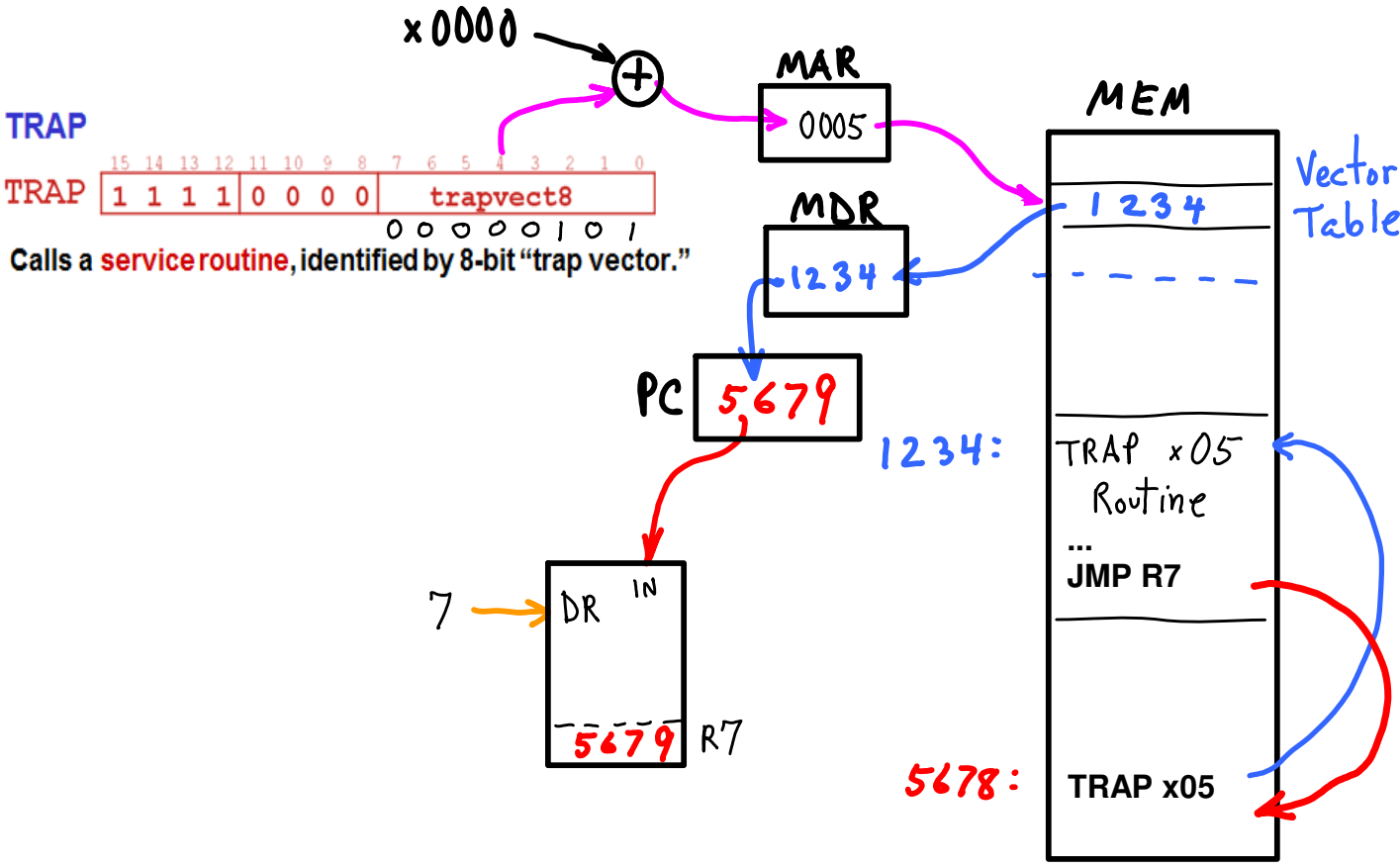
Solution

Have a **data table in memory** containing memory address, and set a **Global Data Pointer**, GDP/R4, to point to it. Now all remote address are available via "**LDR REG, R4, offset**".




```
main:   LEA R7, next
        BRnzp foo
next:   ADD R0, R1, #11

...
foo:    ADD R0, R1, #10
        JMP R7
```



Implement TRAP using other instructions.

--- Could we **eliminate TRAP** from LC3's instruction set?
--- Can we implement the **same effects**?
--- Need **boot process**: set up TVT vector.
--- **Subroutine** called via **TRAP x12** (**x12** is our **TRAP** number)

```
        .ORIG x0200

;*************************************************************
;******** OS SPACE ***********************************************
;*************************************************************

        ;=================================================
        ;=== OS text area
                            ;-----------------------------------
os_boot:                    ;------- OS BOOT Process: ----------

initGDP:                     ;--  init OS's data pointer:
        LEA R4, os_gdtPTR   ;--     get pointer's address
        LDR R4, R4, #0      ;--     GDP (R4) <== pointer
        BRnzp initTVT       ;--     jump over data
os_gdtPTR: .FILL os_gdTbl    ;--- Data: address of OS's data table

initTVT:                     ;--  init TVT:
        LDR R3, R4, #0      ;--     R3 <== address of trap12()
        LDR R2, R4, #1      ;--     R2 <== address of TVT[x12]
        STR R3, R2, #0      ;--     TVT[x12] <==  address of trap12()

goUSER:                      ;--  Jump to user's text:
        LDR R7, R4, #2      ;--     R7 <== pointer to user's text
        JMP R7               ;--     jump

                            ;-----------------------------------
trap12:                     ;------ TRAP x12 (aka trap12)
        JMP R7              ;------    does nothing but return.

        ;=================================================
        ;=== OS data area ================================
os_gdTbl:
os_trap12PTR:     .FILL trap12       ;-- address of trap12()   (offset = 0)
os_const_0012:    .FILL x0012        ;-- address of TVT[x12]   (offset = 1)
os_usrPTR:        .FILL usr_preamble ;-- address of user text  (offset = 2)


;*************************************************************
;******* USER SPACE (faking it) ***********************************
;*************************************************************

        ;=================================================
        ;=== user text area ==============================

usr_preamble:                ;--- init user's data pointer:
        LEA R4, usr_gdtPTR   ;--     get pointer's address
        LDR R4, R4, #0       ;--     GDP (R4) <== pointer
        BRnzp user_main      ;--     jump over data
usr_gdtPTR: .FILL usr_gdTbl   ;--- Data: address of user's data table

user_main:
                             ;-------- emulate TRAP x12:
        LDR R2, R4, #0       ;--    R2 <== address of TVT[x12]
        LDR R1, R2, #0       ;--    R1 <== content of TVT[x12]
        LEA R7, continue     ;--    set return linkage via R7
        JMP R1               ;--    Make the jump to trap12().

continue:                   ;-- TRAP call returns here. What next?
       ADD R0, R0, #2        ;--   do something?

        ;=================================================
        ;=== user data area ==============================
usr_gdTbl:
const_0012: .FILL x0012      ;-- address of TVT[x12]   (offset = 0)

       .END
```

lcc f.c

a.asm

```
//****************
//**  f.c
//***************
int main(void)
{
    return(0);
}
```

```
.Orig x3000
INIT_CODE
LD R6, STACK_POINTER
LD R5, STACK_POINTER
LD R4, GLOBAL_DATA_POINTER
LD R7, GLOBAL_MAIN_POINTER
jsrr R7
HALT

GLOBAL_DATA_POINTER .FILL GLOBAL_DATA_START
GLOBAL_MAIN_POINTER .FILL main
STACK_POINTER .FILL xF000
;;;;;;;;;;;;;;;;;;;;;;;;;;;;main;;;;;;;;;;;;;;;;;;;;;;;;
main
    ;;------- BEGIN ENTER --------
ADD R6, R6, #-1  ;;-- SP-- allocate ret val space
ADD R6, R6, #-1  ;;-- SP--
STR R7, R6, #0   ;;-- push ret addr
ADD R6, R6, #-1  ;;-- SP--
STR R5, R6, #0   ;;-- push BP
ADD R5, R6, #-1  ;;-- set new BP
                 ;;----- allocate locals
ADD R6, R6, #-1
    ;;------- END ENTER ----------
ADD R7, R4, #1
ldr R7, R7, #0
lc3_L1_f
    ;;------- BEGIN-LEAVE --------
ADD R6, R5, 3  ;;-- sp to last arg
STR R7, R6, #0  ;;-- ret val to last arg
LDR R7, R5, #2  ;;-- get saved ret addr
LDR R5, R5, #1  ;;-- restore BP
    ;;------- END-LEAVE --------
RET

GLOBAL_DATA_START
L1_f .FILL lc3_L1_f
L2_f .FILL #0

.END
```

init

go to main as a sub-routine call

TRAP x25

body of main()

this get used? what is it?

what's this? How is it referred to above?

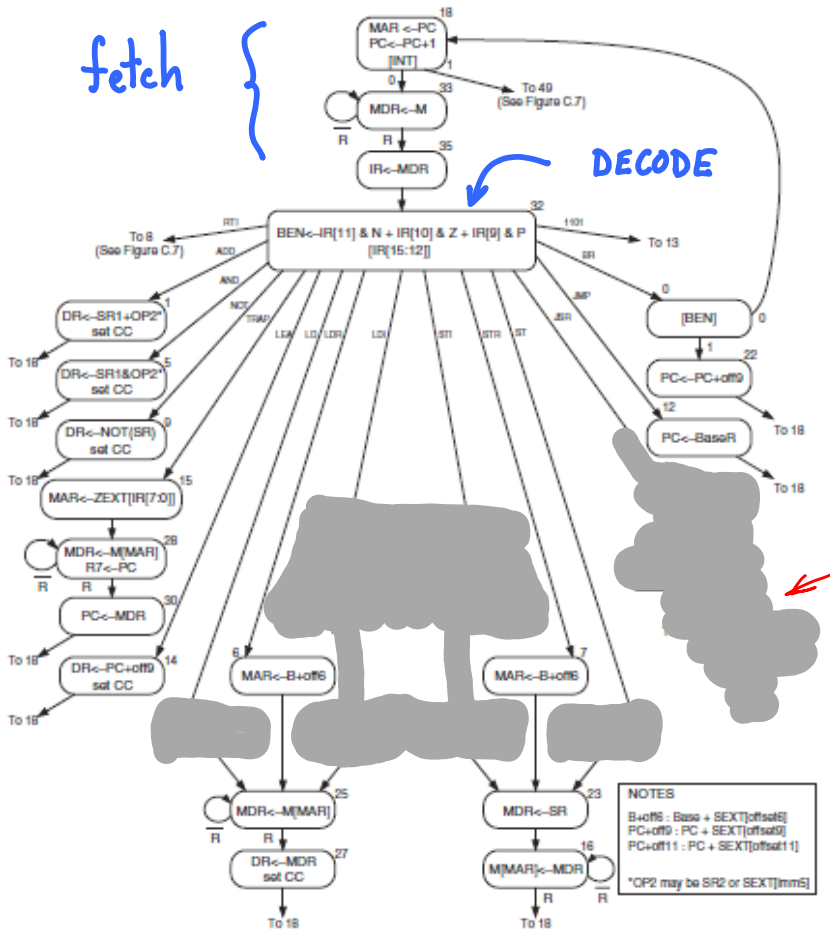
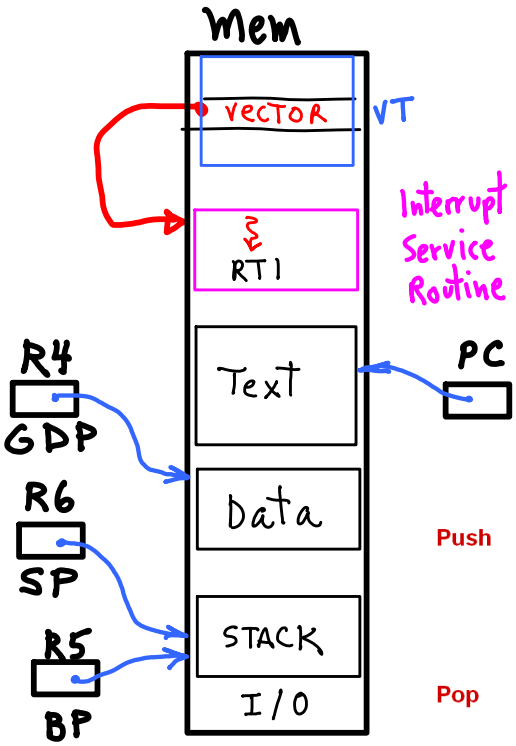


Figure C.2 A state machine for the LC-3

Interrupts, Exceptions



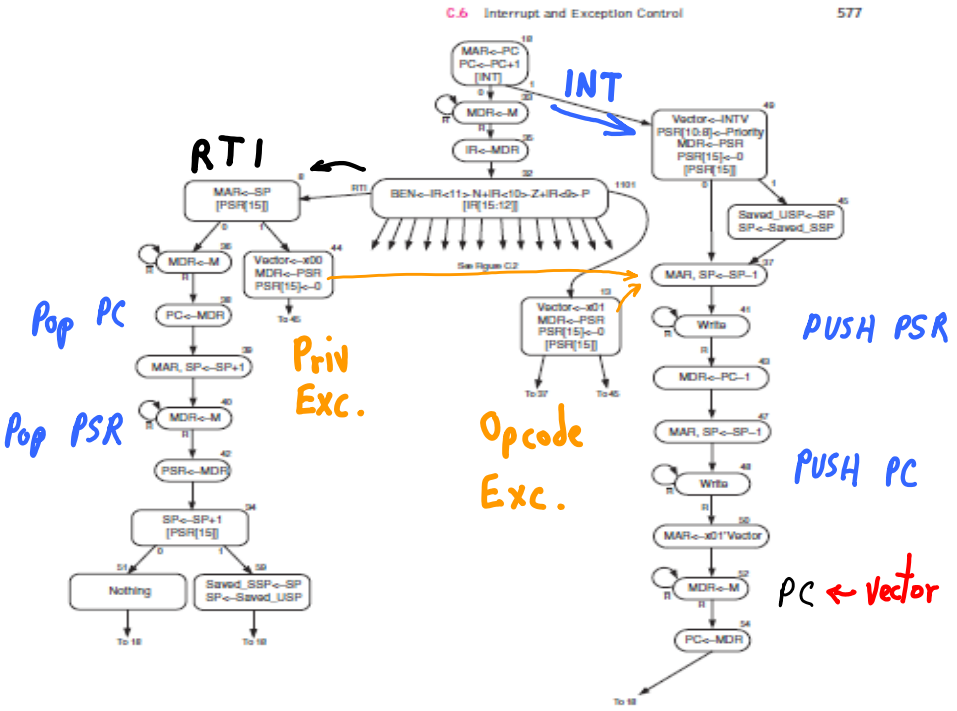


Figure C.7 LC-3 state machine showing interrupt control

Push

```
ADD   R6, R6, #-1  ; decrement stack ptr
STR   R0, R6, #0   ; store data (R0)
```

Pop

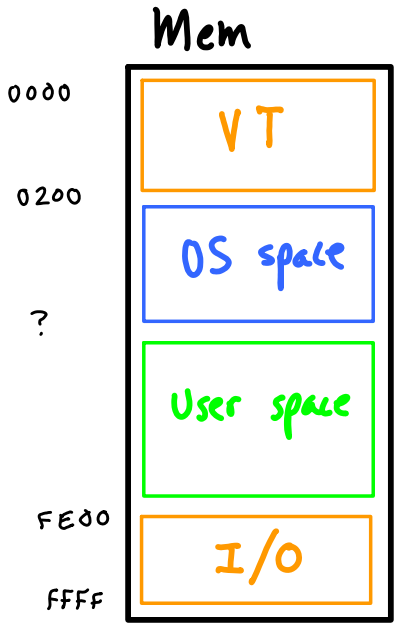
```
LDR   R0, R6, #0   ; load data from TOS
ADD   R6, R6, #1   ; decrement stack ptr
```

R7 - linkage } Hardware defined
R6 - SP

+1

LC-3
Memory-mapped I/O (Table A.3)

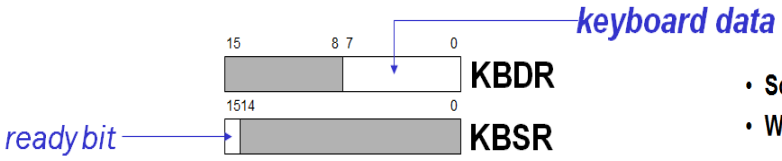
Location	I/O Register	Function
xFE00	Keyboard Status Reg (KBSR)	Bit [15] is one when keyboard has received a new character.
xFE02	Keyboard Data Reg (KBDR)	Bits [7:0] contain the last character typed on keyboard.
xFE04	Display Status Register (DSR)	Bit [15] is one when device ready to display another char on screen.
xFE06	Display Data Register (DDR)	Character written to bits [7:0] will be displayed on screen.



Input from Keyboard

When a character is typed:

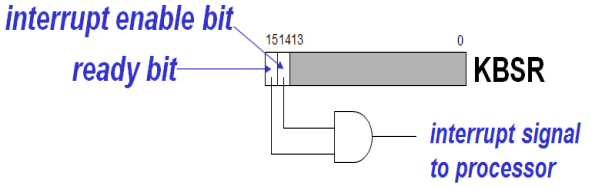
- its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
- the "ready bit" (KBSR[15]) is set to one
- keyboard is disabled -- any typed characters will be ignored

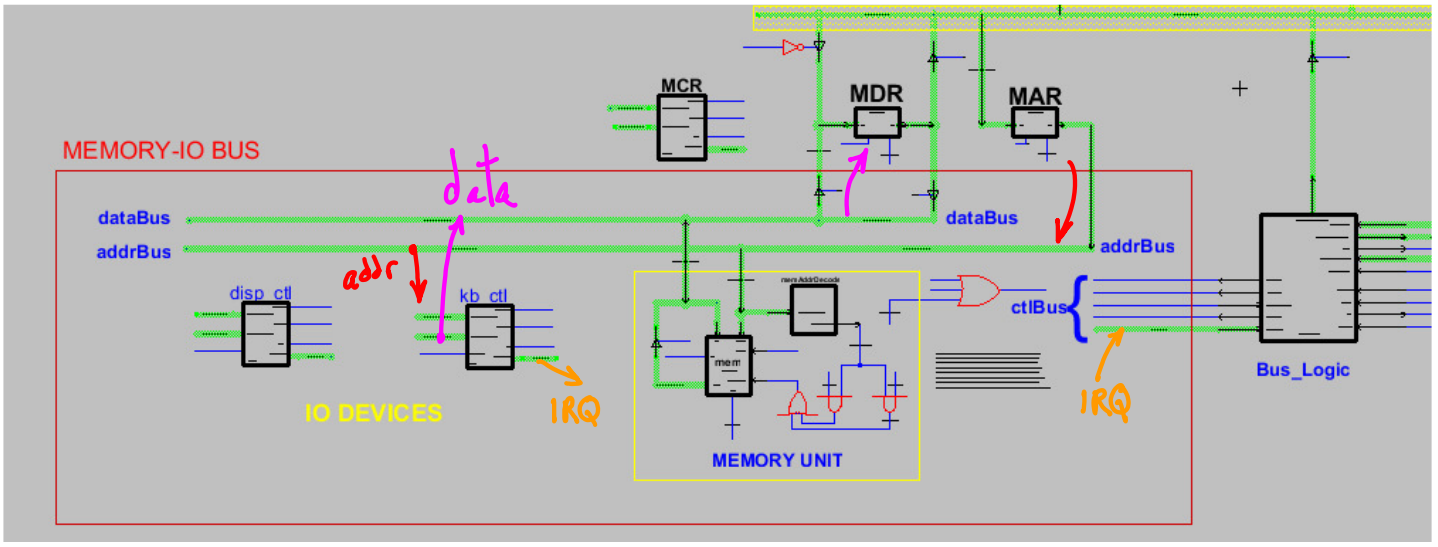


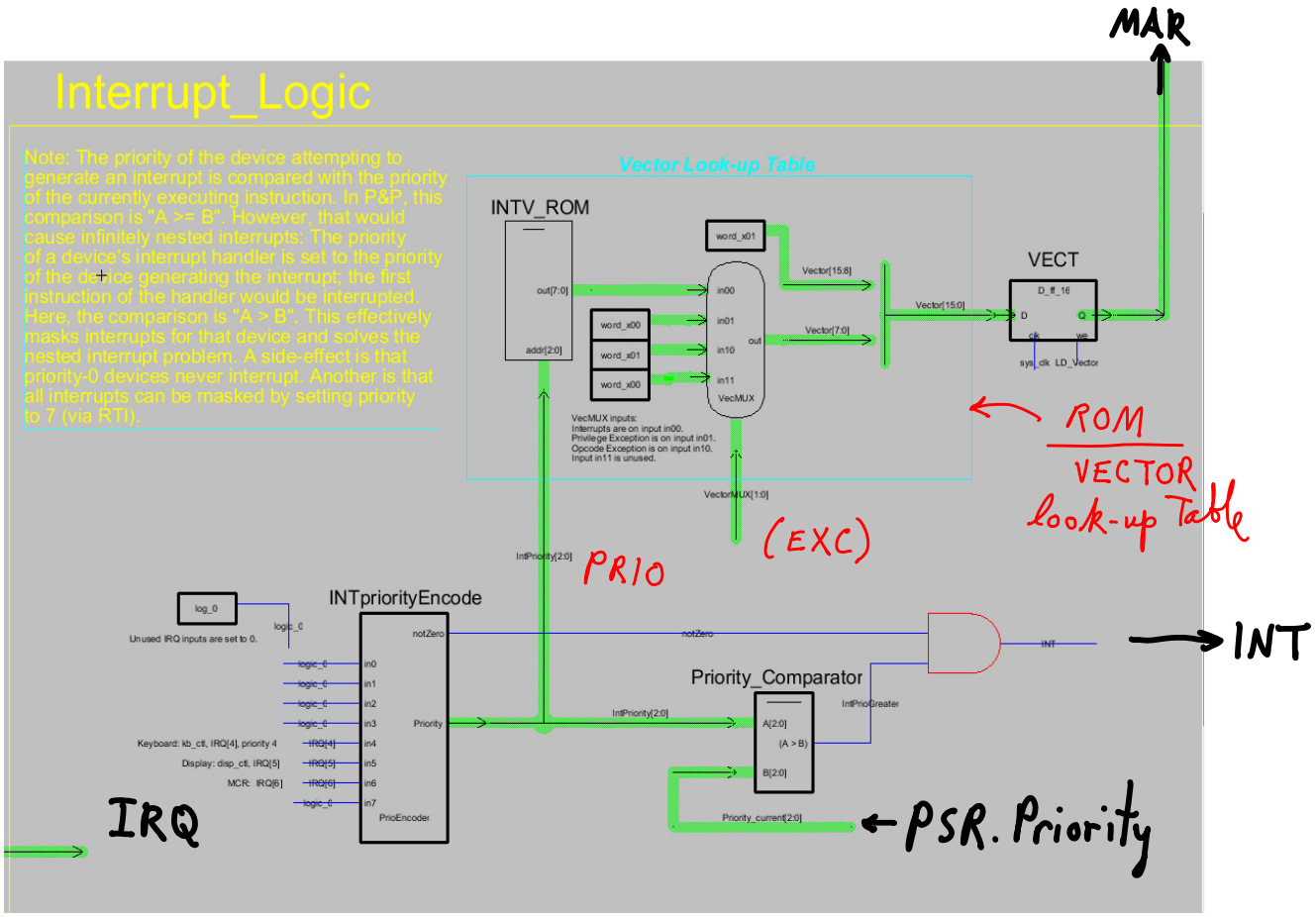
When KBDR is read:

- KBSR[15] is set to zero
- keyboard is enabled

- Software sets "interrupt enable" bit in device register.
- When ready bit is set and IE bit is set, interrupt is signaled.

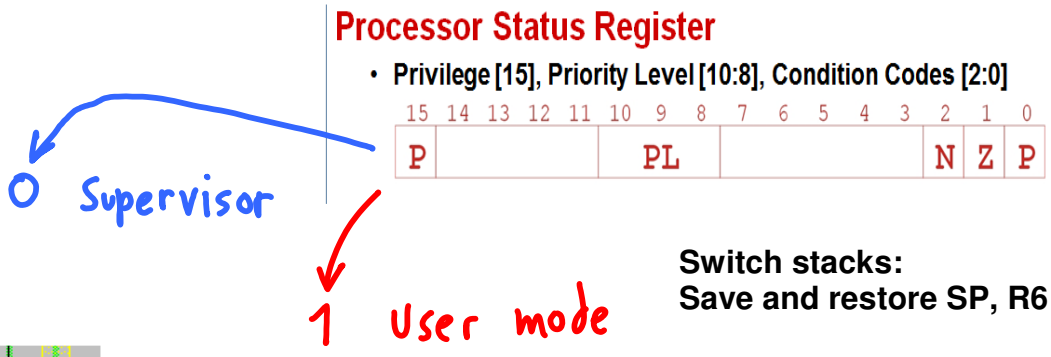




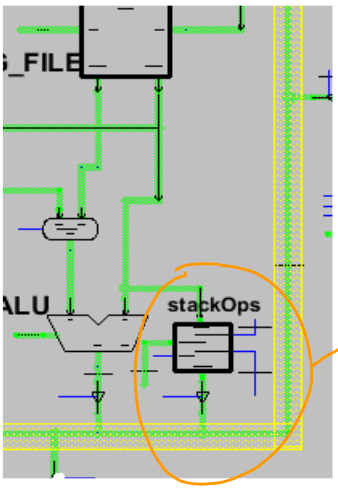

MAR
Interrupt_Logic
Note: The priority of the device attempting to generate an interrupt is compared with the priority of the currently executing instruction. In P&P, this comparison is "A >= B". However, that would cause infinitely nested interrupts. The priority of a device's interrupt handler is set to the priority of the device generating the interrupt: the first instruction of the handler would be interrupted. Here, the comparison is "A > B". This effectively masks interrupts for that device and solves the nested interrupt problem. A side-effect is that priority-0 devices never interrupt. Another is that all interrupts can be masked by setting priority to 7 (via RTI).
Vector Look-up Table
INTV_ROM
out[7:0]
addr[2:0]
word_x01
word_x00
word_x01
word_x00
in00
in01
in10
in11
out
VecMUX
Vector[15:8]
Vector[7:0]
Vector[15:0]
VECT
D_ff_16
D
Q
clk
we
sys_clk LD_Vector
VecMUX inputs:
Interrupts are on input in00.
Privilege Exception is on input in01.
Opcode Exception is on input in10.
Input in11 is unused.
VectorMUX[1:0]
ROM
VECTOR
look-up Table
(EXC)
IntPriority[2:0]
PRIO
log_0
Unused IRQ inputs are set to 0.
logic_0
INTpriorityEncode
notZero
Priority
in0
in1
in2
in3
in4
in5
in6
in7
PrioEncoder
Keyboard: kb_ctl, IRQ[4], priority 4
Display: disp_ctl, IRQ[5]
MCR: IRQ[6]
IRQ[4]
IRQ[5]
IRQ[6]
IRQ
Priority_Comparator
A[2:0]
(A > B)
B[2:0]
IntPriorityGreater
Priority_current[2:0]
PSR.Priority
INT
INT

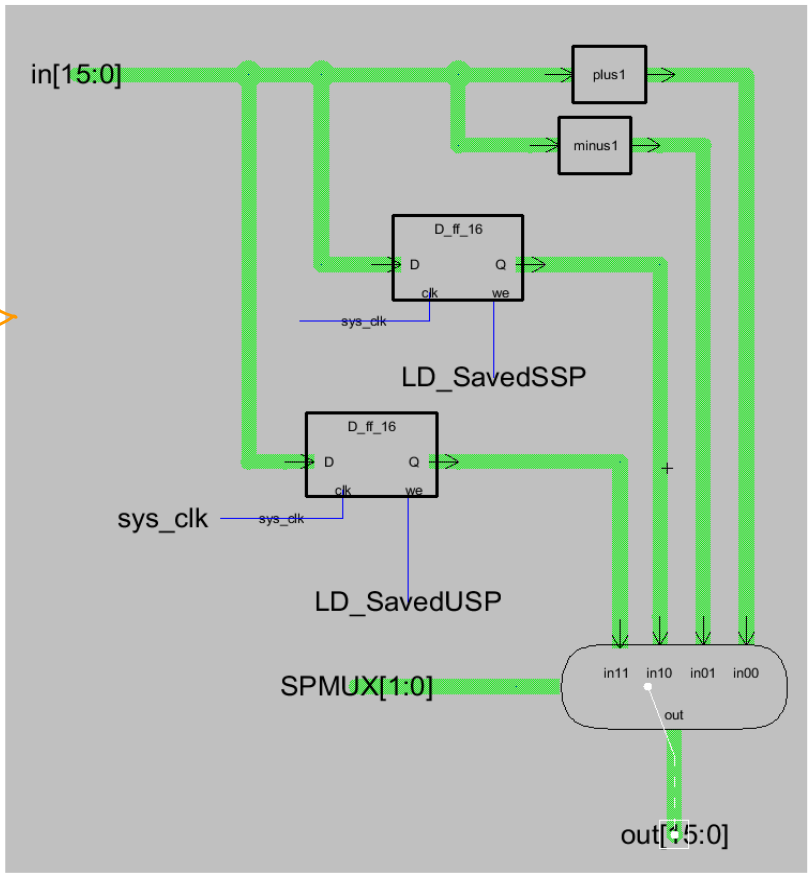
-

1. **Pop PC from supervisor stack.** (PC = M[R6]; R6 = R6 + 1)
2. **Pop PSR from supervisor stack.** (PSR = M[R6]; R6 = R6 + 1)
3. **If PSR[15] = 1, R6 = Saved.USP.**
   (If going back to user mode, need to restore User Stack Pointer.)



Switch stacks:
Save and restore SP, R6





Hardware stack operations

push
pop
switch stacks:
UsersSP <==> SupersSP

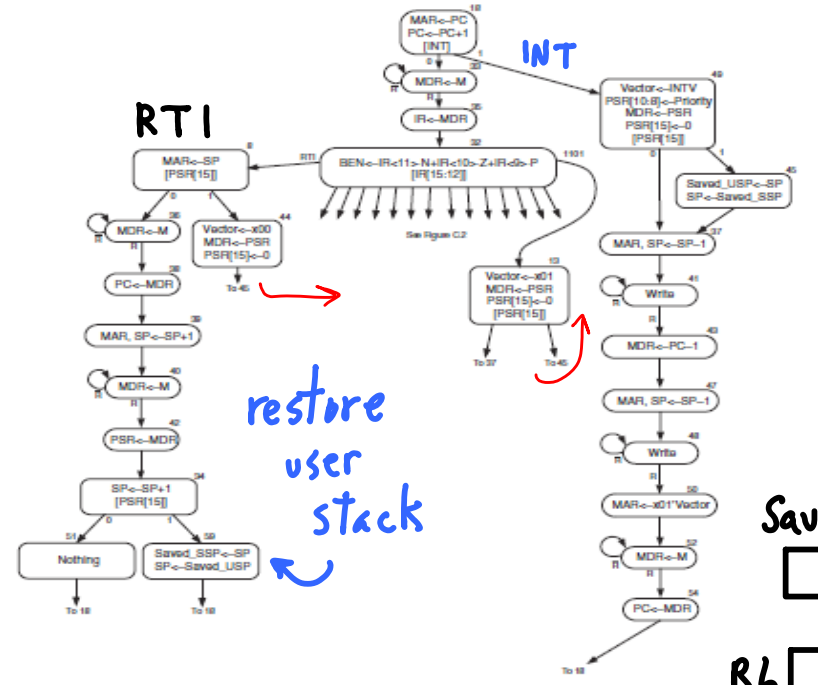
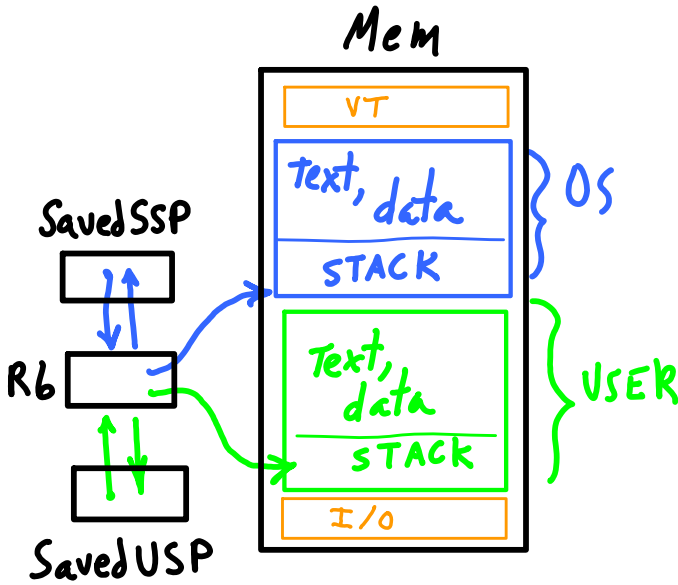


Figure C.7 LC-3 state machine showing interrupt control



Supplemental Readings
(PP) Patt & Patel, *Introduction to Computing Systems, 2e* (revised printing). McGraw-Hill. (Also see online material in LCA3-trunk/docs/ or on the Web.)

PP, Chp. 7: 7.1-7.4 (LC-3 Assembly language: instruction syntax, labels, comments, assembler directives, 2-pass assembly, object files and linking, executable images).

PP, Chp 8: 8.1.1-8.3.3 (device registers, memory-mapped I/O, keyboard and display I/O), 8.5 (interrupts).

PP, Chp 9: 9.1.1-9.2.2 (TRAP/JSR subroutine calls, register saving).

PP, Chp 10: 10.1-10.2 (stacks, push/pop, stack under/overflow, interrupt I/O, saving/restoring program state).

PH, Chp 8: 8.1-8.5 (I/O, disks, disk structure, RAID, buses, polling vs. interrupts, interrupt priority, DMA). [Also see on the CD, 8.3 (networks).]

PP Appendices A and C (also see LC3-turnk/docs/):
LC-3 Instruction notation definitions: App. A.2
LC-3 Instruction descriptions: App. A.3
LC-3 TRAP routines: App. A.3, Table A.2
LC-3 I/O device registers: App. A.3, Table A.3
LC-3 Interrupt and exception execution: App. A.4 and C.6
LC-3 FSM state diagram: App. C, Fig. C.2 and C.7
LC-3 Complete datapath: App. C, Fig. C.8
LC-3 memory map: App. A.1

Supplemental Exercises from PP

PP, Chp 6:
6.13 (shift right [NB-use assembly language])

PP, Chp 7:
7.1 (instr. assembly w/ labels [NB-show instr. bits])
7.14 (replace an opcode in assembled prog. to debug)
7.24 (debug loop control)

PP, Chp 8:
8.5 (what is KBSR[15]?)
8.11 (polling vs. intr. efficiency)
8.14 (I/O addr. decode)
8.15 (KBSR[14] and intr. handling)

PP, Chp 9:
9.2 (TRAP execution)
9.13 (debugging JSR and RET)
9.19 (complete the intr. priority service call)

PP, Chp 10:
10.10 (CCs pushed on INT)
10.11 (device registers and IVT)
10.24 (prog. and INT service interaction)

Supplemental Exercises from PP

PP, Chp 6:
6.13 (shift right [NB-use assembly language])

PP, Chp 7:
7.1 (instr. assembly w/ labels [NB-show instr. bits])
7.14 (replace an opcode in assembled prog. to debug)
7.24 (debug loop control)

PP, Chp 8:
8.5 (what is KBSR[15]?)
8.11 (polling vs. intr. efficiency)
8.14 (I/O addr. decode)
8.15 (KBSR[14] and intr. handling)

PP, Chp 9:
9.2 (TRAP execution)
9.13 (debugging JSR and RET)
9.19 (complete the intr. priority service call)

PP, Chp 10:
10.10 (CCs pushed on INT)
10.11 (device registers and IVT)
10.24 (prog. and INT service interaction)