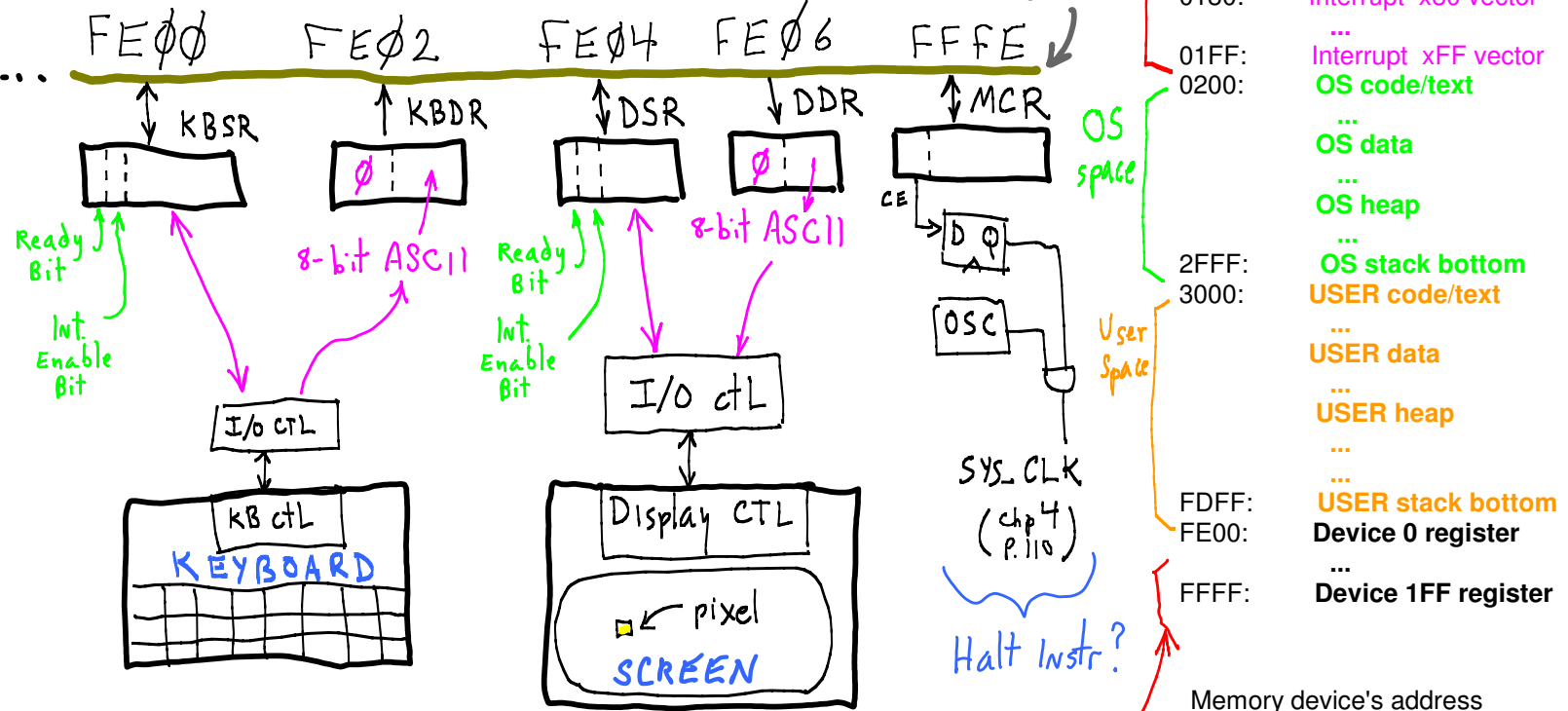# Lec-memMapIOtrapJSR

Memory Mapped I/O:
I/O Devices have read/write access via device registers.
To send a word of data to a device, write to its memory-mapped address:
  ST R1, <address?>
To receive a word of data, read (LD).

address   mem

Vector Table
0000: Trap x00 vector
       ...
00FF: Trap xFF vector
0100: Exception x00 vector
       ...
017F: Exception x7F vector
0180: Interrupt x80 vector
       ...
01FF: Interrupt xFF vector
0200: OS code/text
       ...
       OS data
       ...
       OS heap
       ...
2FFF: OS stack bottom
3000: USER code/text
       ...
       USER data
       ...
       USER heap
       ...
       ...
FDFF: USER stack bottom
FE00: Device 0 register
       ...
FFFF: Device 1FF register

OS space

User Space

Data Bus

FE00 KBSR
FE02 KBDR
FE04 DSR
FE06 DDR
FFFE MCR

Ready Bit
Int. Enable Bit
8-bit ASCII

I/O CTL
KB ctl
KEYBOARD

Ready Bit
Int. Enable Bit
8-bit ASCII

I/O ctl
Display CTL
pixel
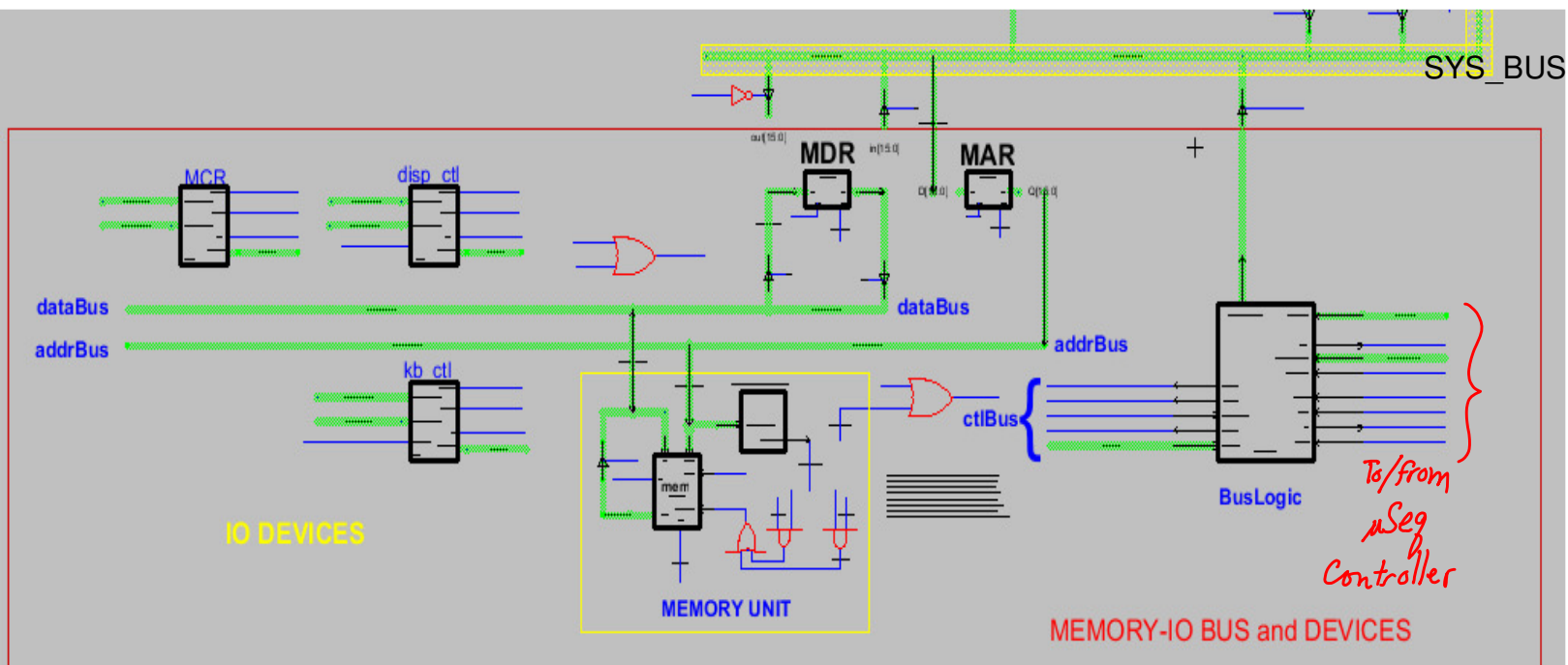SCREEN

CE
D Q
OSC
SYS_CLK (chp 4 p.110)
Halt Instr?

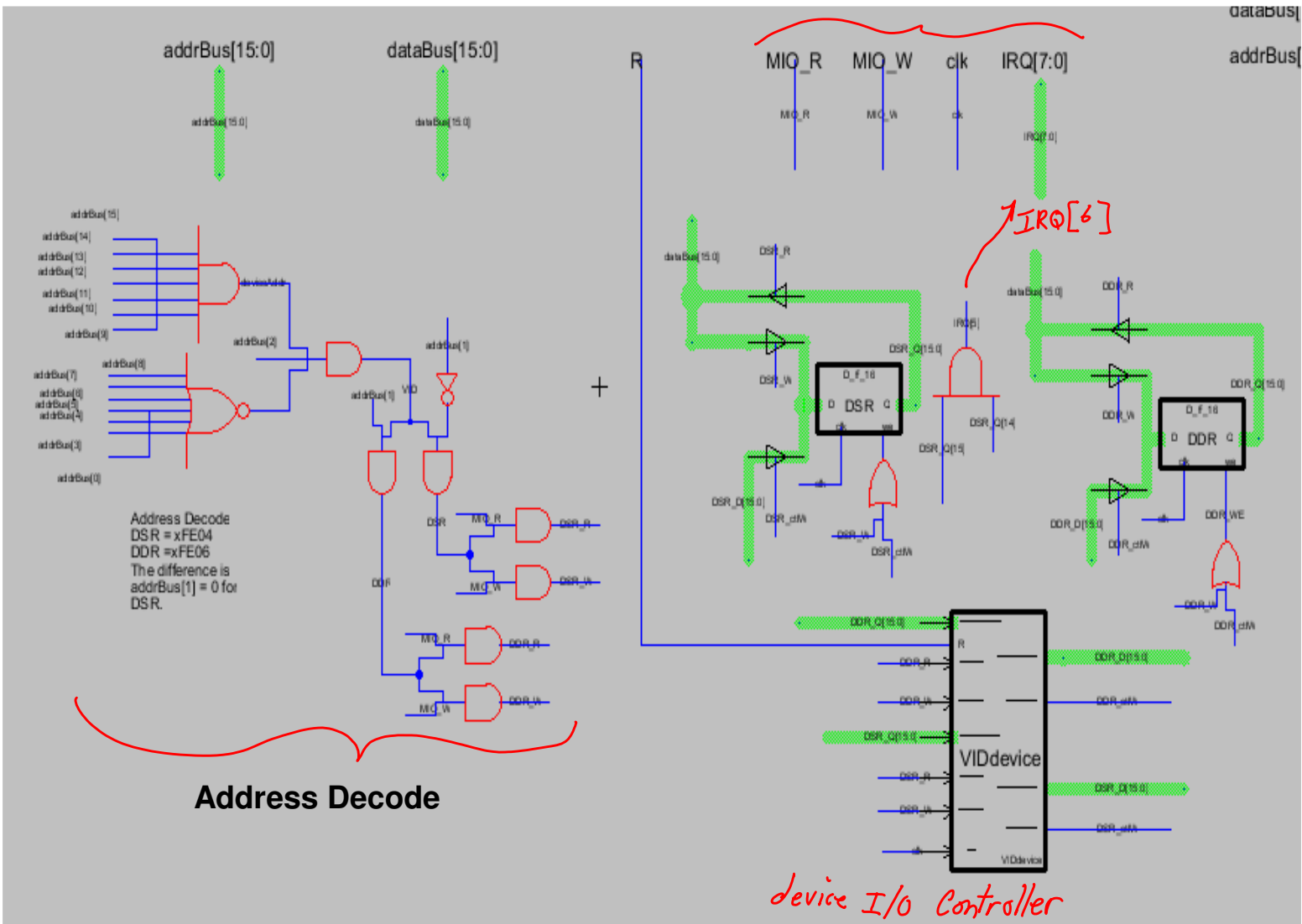Red parts of memory:
usage defined by Hardware.

Memory "Shadowed" by devices

Memory device's address decoder does not recognize these as memory addresses.

This portion of physical memory is not accessible.

SYS_BUS

MCR   disp_ctl   MDR   MAR
out[15:0] in[15:0]
dataBus
addrBus
kb_ctl
ctlBus
IO DEVICES
mem
MEMORY UNIT
BusLogic
To/from µSeq Controller
MEMORY-IO BUS and DEVICES

disp_ctl, The Display Device I/O Interface

ctl_Bus

addrBus[15:0]

dataBus[15:0]

R   MIO_R   MIO_W   clk   IRQ[7:0]

dataBus[

addrBus[

IRQ[6]

Address Decode
DSR = xFE04
DDR = xFE06
The difference is
addrBus[1] = 0 for
DSR.

Address Decode

VIDdevice

device I/O Controller

Memory-Mapped Keyboard I/O

To reference I/O device, we need 16-bit address.

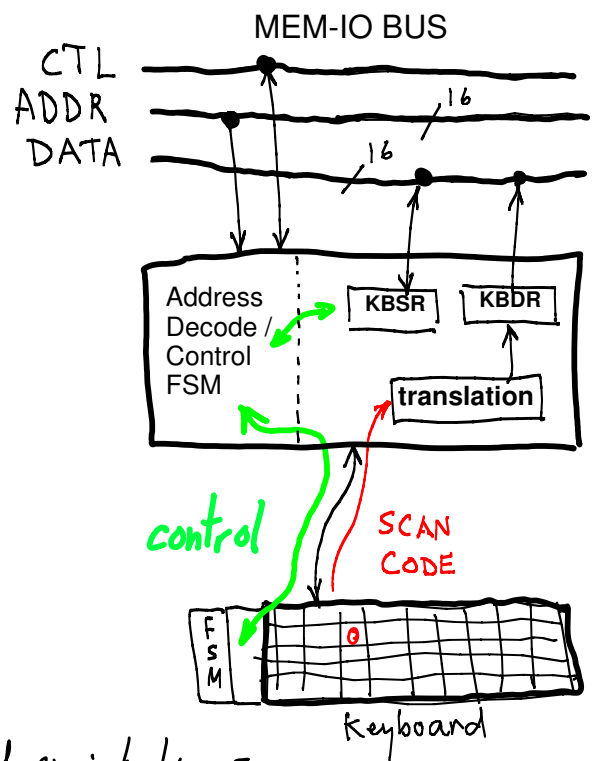==> Use pointer variable

```
================ ASM source code:
    .orig x3000
    LDI R0, KBDRptr    → PC-relative offset calculated
KBDRptr:                  by assembler
    .FILL xFE02

================ Machine code in memory:
3000: 1010 000 000000000
3001: 1111 1110 0000 0010

================ Execution:
R0 <== MEM[ MEM[ 3001 + 0 ] ]
     == MEM[ xFE02 ]
     == KBDR
```

MEM-IO BUS

CTL
ADDR
DATA

16

16

Address Decode / Control FSM

KBSR   KBDR

translation

control

SCAN CODE

FSM

Keyboard

NB- above code executes 3001 as if an instruction. Fix that code.

Reading KBDR device register

1. Keypress event, '4':
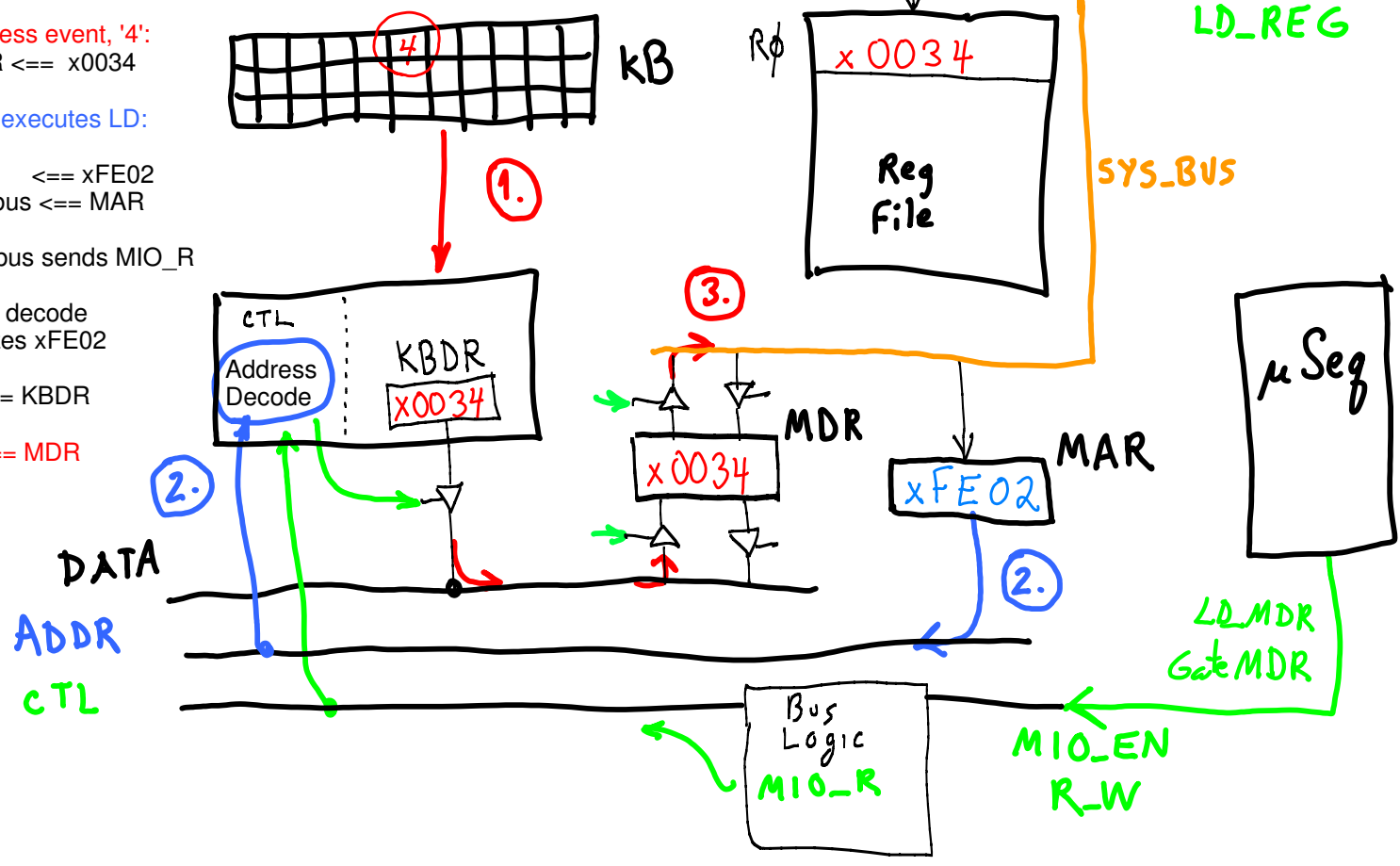   KBDR <== x0034

2. Code executes LD:

MAR   <== xFE02
ADDR_bus <== MAR

Control bus sends MIO_R

Address decode recognizes xFE02

MDR <== KBDR

3. R0 <== MDR

KB

4

R0
x0034

Reg File

LD_REG

SYS_BUS

3.

CTL
Address Decode
KBDR
x0034

MDR
x0034

MAR
xFE02

μSeq

2.

DATA

ADDR

CTL

Bus Logic
MIO_R

LD_MDR
Gate MDR

MIO_EN
R_W

What is Non-Memory Mapped I/O?

--- Separate opcodes and address space (Intel x86 IA-32 and IA-64 ISAs):

    in r4, x0D

--- Control-Bus signal "IO": 0, access memory; 1, access I/O device (uses same address wires)

DEVICE CONTROL Reg Polling

How do we know data register has valid data? Ask!
Ask whom? Device's status register.

(Later: have the status register alert us instead.)

KB device controller sets Ready bit when new data is available. Clears it when KBDR is read.
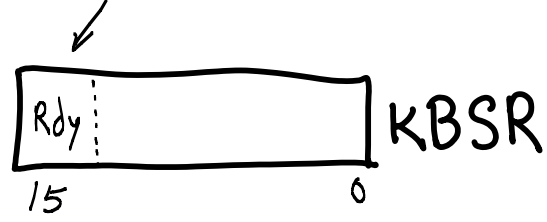
```
POLL:
        LDI R1, KBSRptr

        BRn Ready
        BRnzp POLL

Ready:
        LDI R0, KBDRptr

        BRnzp DONE

KBSRptr:    .FILL  xFE00
KBDRptr:    .FILL  xFE02

DONE:
        ;--- (do other stuff)
```

R1 ← KBSR
KBSR[15] == 1?   NO

yes

R0 ← KBDR

continue

Rdy   KBSR
15   0

LC3 Operating System I/O functions

Depends on OS version.

TRAP <n>	Assembly-psuedonymn	Description
TRAP x25	HALT	jump to OS w/ message, loops in OS forever.
TRAP x20	GETC	one char in, keyboard data ==> R0[7:0] (clears R0 first).
TRAP x21	OUT	one char out, R0[7:0] ==> display; ignores big-end byte, R0[15:8].
TRAP x22	PUTS	string out, Mem[R0++] ==> display until x0000. Ignore big-end byte, 1 char per word.
TRAP x23	IN	displays prompt, then one char in ala GETC.
TRAP x24	PUTSP	same as PUTS, but packed (2 chars per word, little-end byte then big-end byte).

See PP, Append. A.4, Table A.2

Who wrote this code? Which OS code is loaded into PennSim? Does lc3as translate "HALT", "halt", ...?
Who wrote lc3as? Does every assembler for LC3 do the same thing?

PROCESSOR BUS
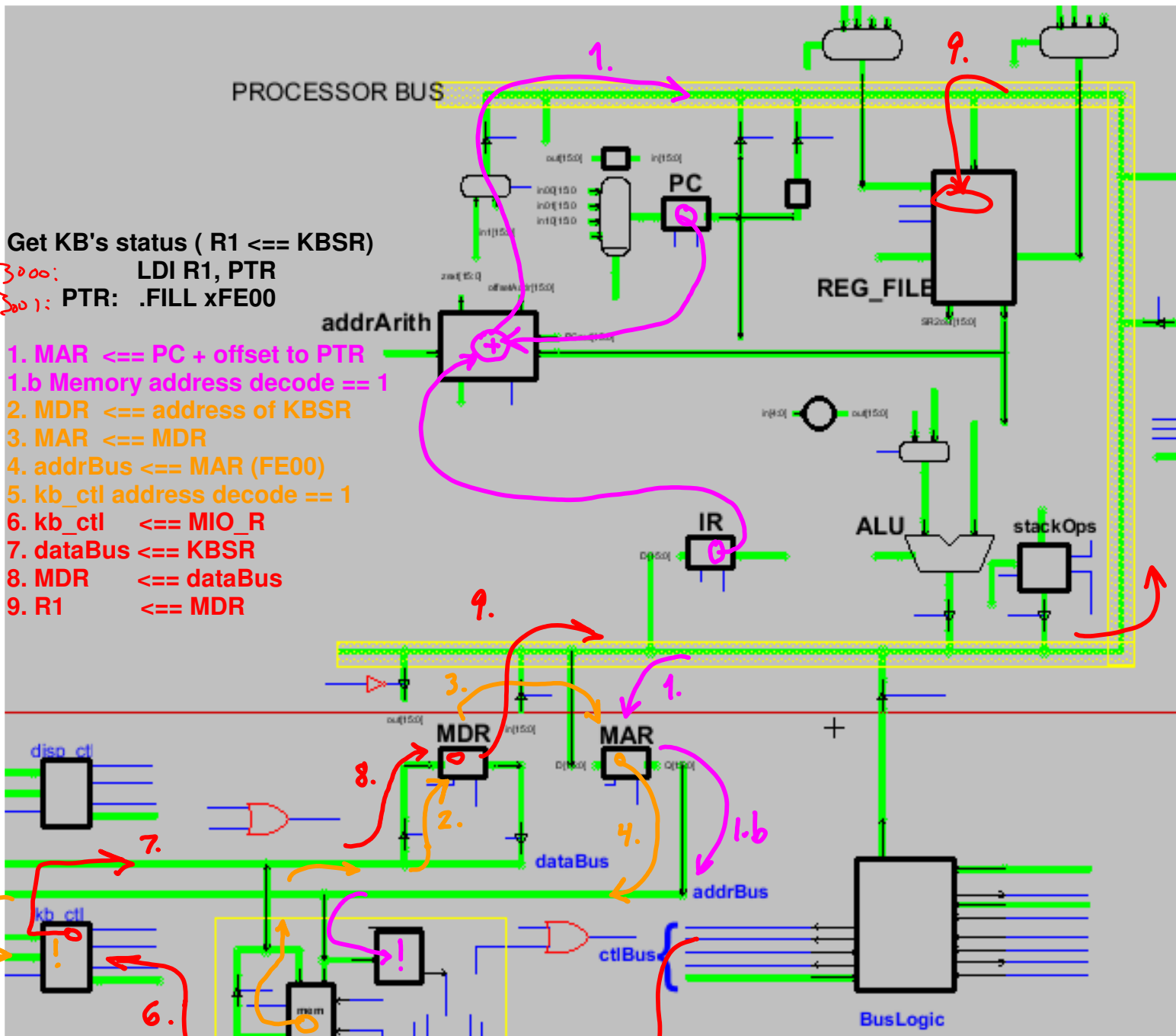
Get KB's status (R1 <== KBSR)
3000:   LDI R1, PTR
3001: PTR: .FILL xFE00

1. MAR <== PC + offset to PTR
1.b Memory address decode == 1
2. MDR <== address of KBSR
3. MAR <== MDR
4. addrBus <== MAR (FE00)
5. kb_ctl address decode == 1
6. kb_ctl   <== MIO_R
7. dataBus <== KBSR
8. MDR   <== dataBus
9. R1   <== MDR

PC

REG_FILE

addrArith

IR

ALU   stackOps

disp_ctl

MDR   MAR

dataBus

kb_ctl

mem

addrBus

ctlBus

BusLogic

MIO_R

Suppose:
PC == 1000
    Memory
    ...
1000: LDI R1, (+1)
1001: BRnzp (+1)
1002: FE00
    ...

instruction fetch:
IR <== 1010 001 000000001
PC <== 1001

1. calculate address:
    MAR <== **PC** + **IR**[8:0]   (1001 + 1)

2. fetch PTR's data from memory:
    MDR <== **Mem**[1002] == FE00

3, 4. send address to addrBus:
    MAR   <== **MDR** (FE00)
    addrBus <== **MAR**

5. kb_ctl recognizes address:
   **addrBus**[15:0] == 7'hFE00

6. uSeq Controller sends Read
    MIO_R == 1

7. kb_ctl sends KBSR to dataBus
    dataBus <== **KBSR**

8. MDR gets KBSR data:
    MDR <== **dataBus**

9. KBSR data to destination register:
R1 <== sys_bus <== **MDR**

LDI

15

]

8

10

MAR<–PC+off9

1. calculate address

24

MDR<–M[MAR]

2. fetch PTR

$\overline{R}$ R

0

4

6

MAR<–B+off6

2

MAR<–PC+off9

26

MAR<–MDR

3., 4. Send PTR address to addr Bus

25

MDR<–M[MAR]

$\overline{R}$ R

5., 6-8. KbctL sends KBSR data

DR<–MDR
set CC   27

9. KBSR data to register
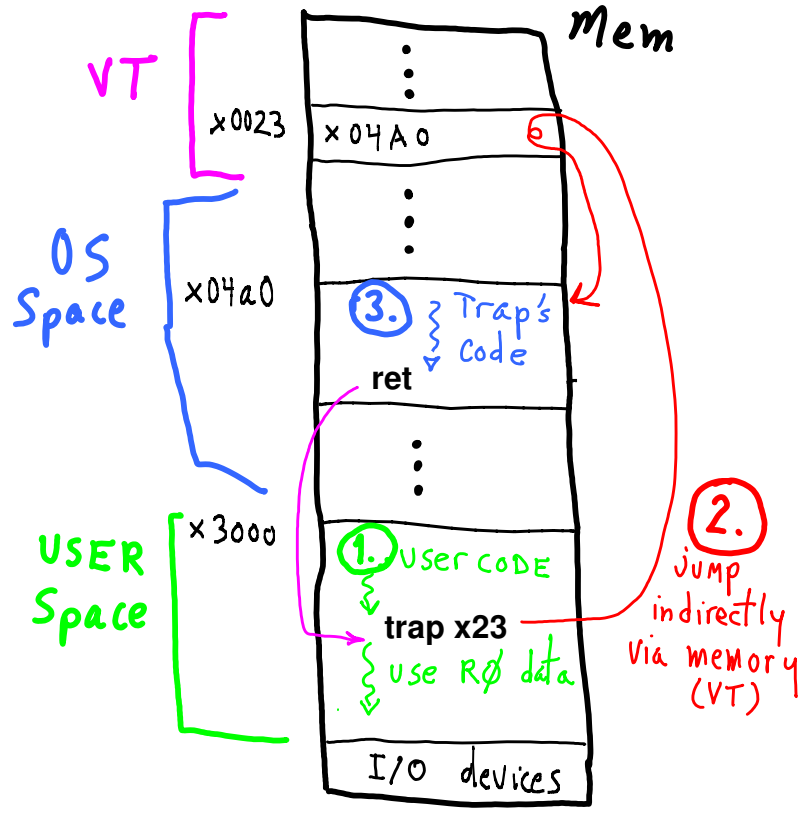
To 18

fetch
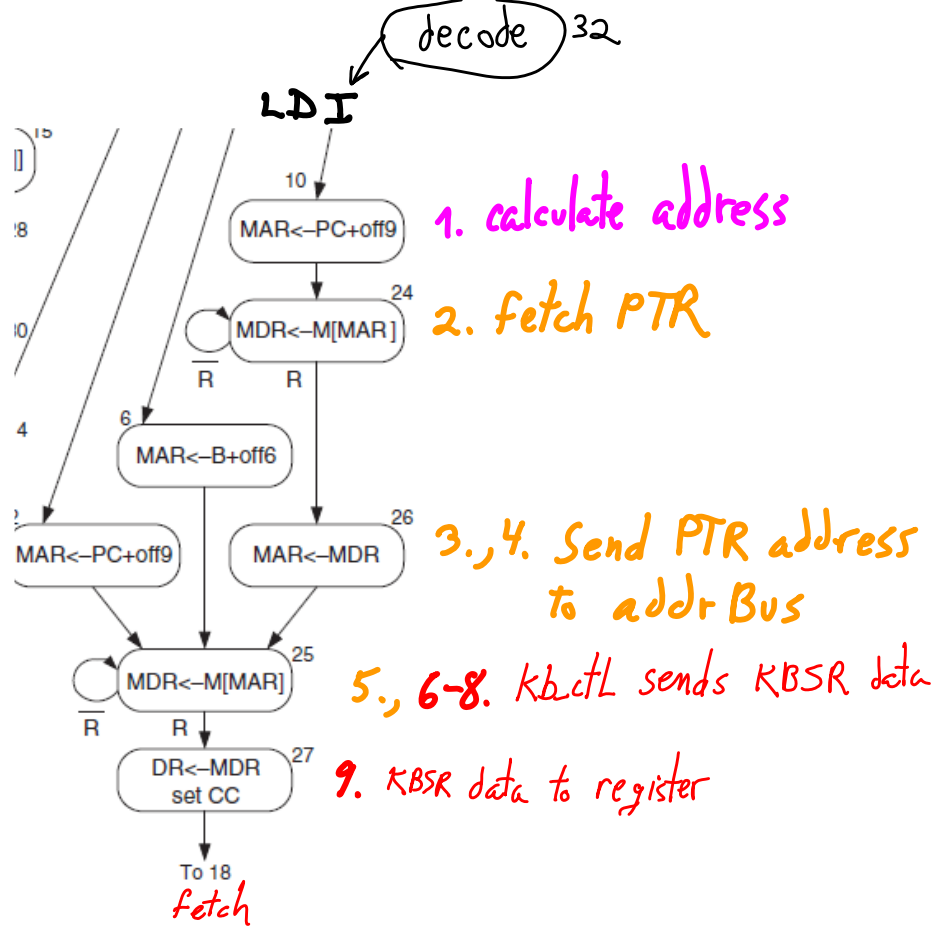
Trap instruction execution

TRAP routines provide services.
User (and OS) code jumps to Trap's code to use service.

TRAP x 23

R7 <== PC
PC <== Mem[IR[7:0]]

IR

| 1111 | | 0010 0011 |

Op   Vector 8
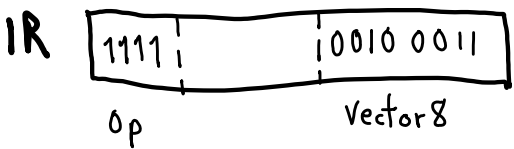
(1.) USER CODE executes Trap

(2.) jump: PC loaded from VT

(3.) Trap service code executes

(4.) ret executed

"ret" = "jmp R7"
(PC ← R7)

Mem

VT
x0023   x04A0

OS Space
x04a0

(3.) Trap's Code
ret

USER Space
x3000

(1.) USER CODE
trap x23
use R0 data

I/O devices

(2.) jump indirectly via memory (VT)

TRAP ROUTINE USAGE
Saving/Restoring Regs

Whose job is it to save/restore registers and state?
Depends:
-- Register?   ==> caller or callee? by convention.
-- PC, CC/PSR?   ==> hardware
-- stack SP?   ==> hardware/OS/user code.

```
;======================
;= Trap x23 Routine (aka "IN")
;======================
.ORIG x04a0         ;== code's runtime location.

st r1, saved_r1   ;== save user's registers.
st r2, saved_r2
   ...            ;== get input
   ...
ld r1, saved_r1   ;== restore user's registers.
ld r2, saved_r2
ret               ;== return to user's code

saved_r1:   .FILL x0000   ;== reg value storage
saved_r2:   .FILL x0000   ;== reg value storage

.END
```

Here, convention is callee saves.
Stated in OS's documentation.

Display prompt
    r1   <== char
    DSDR <== r1
Read 1 char from kb
    r2 <== KBSR
    poll
    r0 <== KBDR
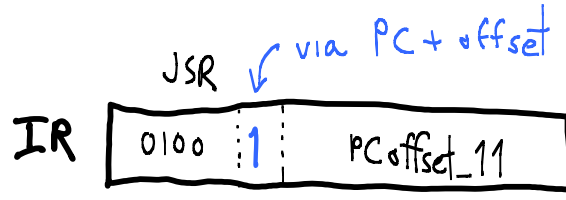
USER-LEVEL "TRAPS"
i.e., Sub-routines

Reuse code.

Who (caller/callee) is responsible for saving/restoring registers?

If code is generated by compiler? ==> by compiler convention.
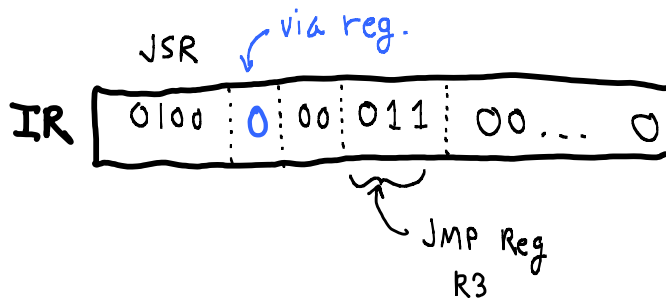
(What about CC?)

```
.orig x3000
    ...              ;= do stuff
jsr foo              ;= jump to foo()
    ...              ;= use foo()'s result
jsr foo              ;= jump to foo()  again
    ...

foo:                 ;= code for foo()
    ...
ret                  ;= jmp r7
```
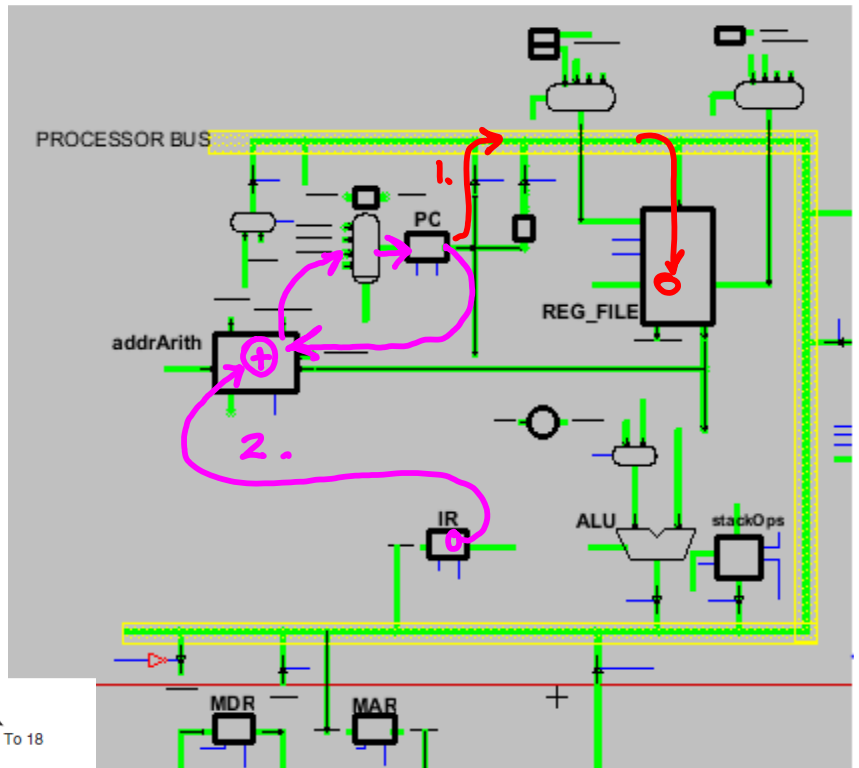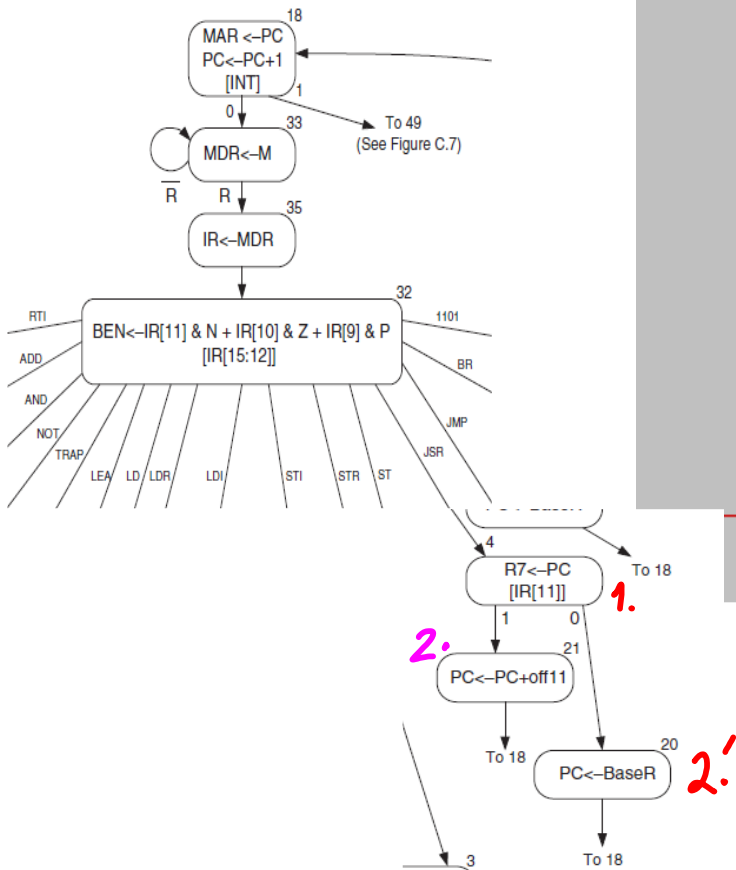
JSR   via PC + offset

IR | 0100 | 1 | PCoffset_11 |

"JSR foo"
R7 ← PC
PC ← PC + offset

JSR   via reg.

IR | 0100 | 0 | 00 | 011 | 00 ... 0 |
                      JMP Reg
                      R3

"JSRR r3"
R7 ← PC
PC ← reg

JSR
R7 <== PC
PC <== PC + IR[10:0]

18
MAR <–PC
PC<–PC+1
[INT]
1
0   33   To 49
                  (See Figure C.7)
MDR<–M
R̄   R   35
IR<–MDR
                                              32
RTI   1101
ADD   BEN<–IR[11] & N + IR[10] & Z + IR[9] & P   BR
AND   [IR[15:12]]
NOT   JMP
TRAP   JSR
LEA LD/LDR   LDI   STI   STR ST

4
R7<–PC   To 18
[IR[11]]   1.
1   0
        21
PC<–PC+off11
2.

To 18   20
        PC<–BaseR   2.'
To 18   To 18

See complete state:

docs/LC3-uArch-ControlStates.html
docs/LC3-uArch-PPappendixC.pdf

```
JSSR
R7 <== PC
PC <== RegFile[ IR[8:6] ]
```
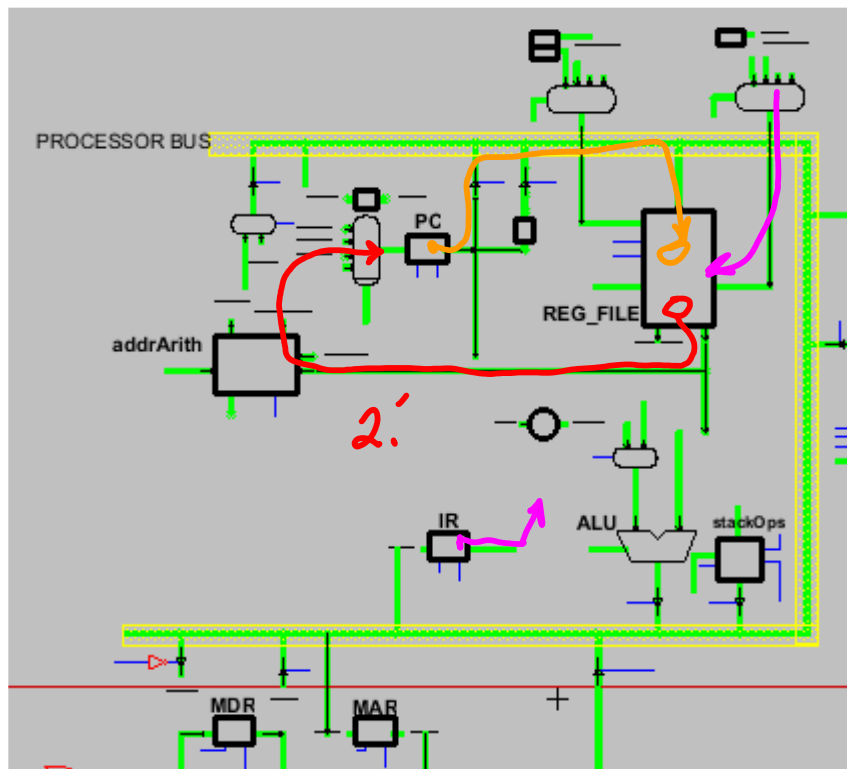

OK, So far, so good, BUT

What about passing arguments and return values?

What about nested calls, recursion?

Generally, saving state involves more than RegFile:

```
PSR[15]   =  Privilege level           ( 1=user, 0=super)
PSR[10:8] =  Interrupt Priority level    (0=low, 7=high)
PSR[2:0]  =  CC (N, Z, P flags)
```

Also, there are usually other important bits in PSR (but not for LC3).
Also, there are other status and control registers (but not for LC3).

```
;====================
;=    Trap x23 (aka "IN")
;====================
.orig x04a0

st r7, saved_r7
jsr save_regs
   ...
save_regs:
st r1, saved_r1
st r2, saved_r2
   ...
st r6, saved_r6
ret

saved_r1: .FILL x0000
saved_r2: .FILL x0000
   ...
saved_r6: .FILL x0000
saved_r7: .FILL x0000
```

In PP's Fig. 9.8,

Why isn't "**st r7, saved_r7**" inside the routine "**save_regs**"?

Does this mechanism work for nested calls?

Could we use the stack instead?

Why not "jsr save_regs" before "st r7, saved_r7"?

Where did this Trap x23 "IN" code come from?
If its' in OUR LC3 memory, how did it get there?

Where can I find the source code for the trap x23 routine?

(1) run PennSim.jar or lc3sim or Simulate.exe,
    ---- look at VT, at address x0023
    ---- see what address is stored there
    ---- look at code at that address

(2) see OS source code in src/lc3os.asm