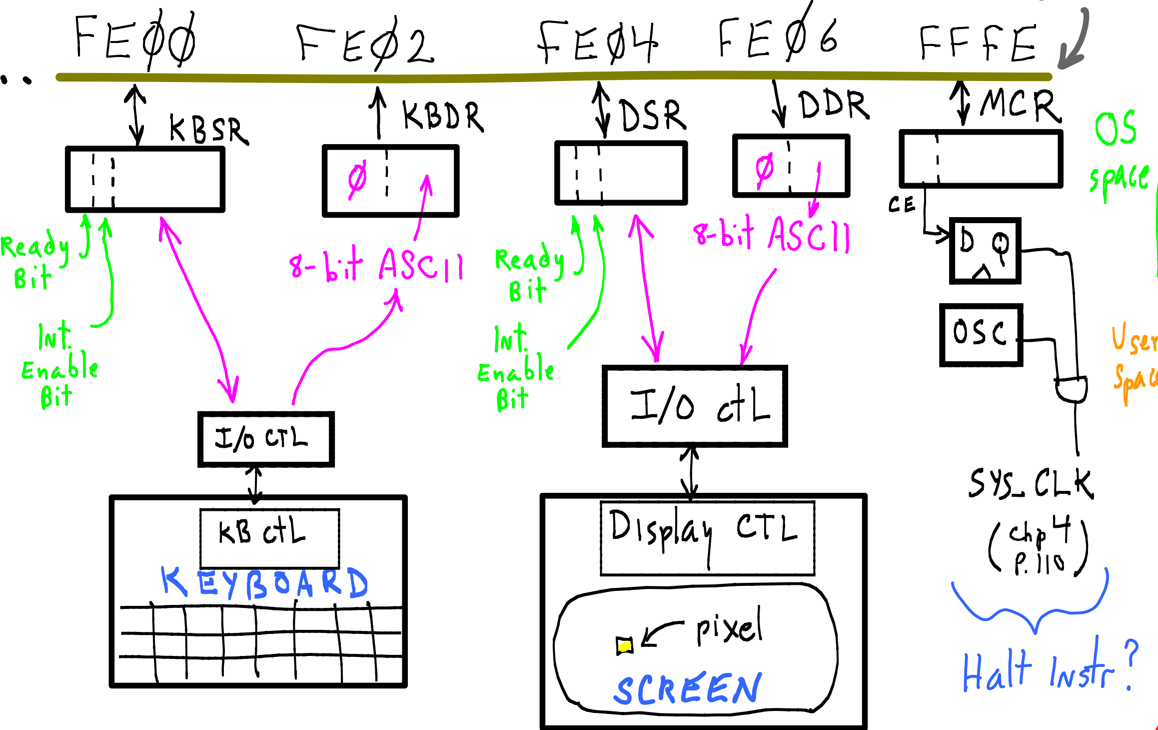
Lec-memMapIOtrapJSR

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
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

Vector Table

OS space

User Space

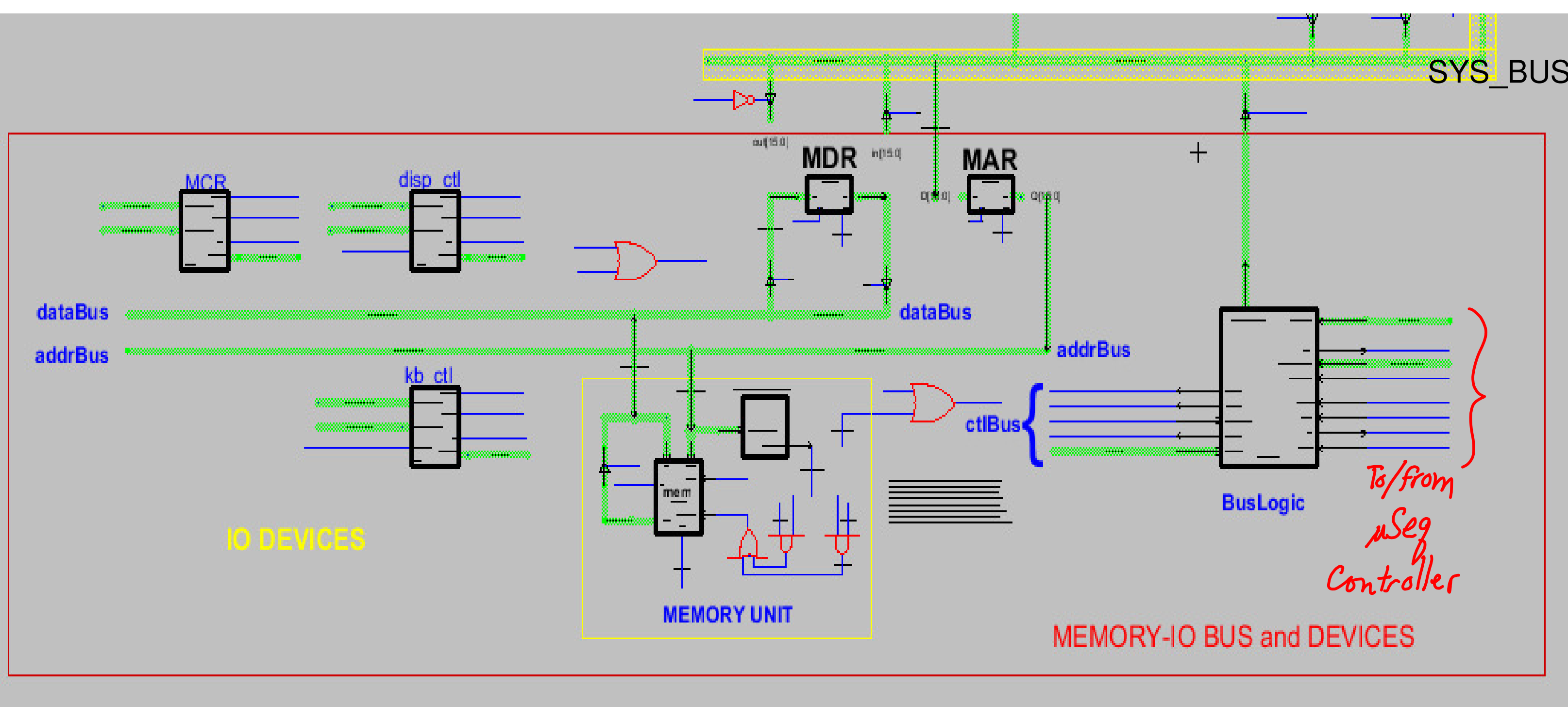
Halt Instr?

Red parts of memory: usage defined by Hardware.

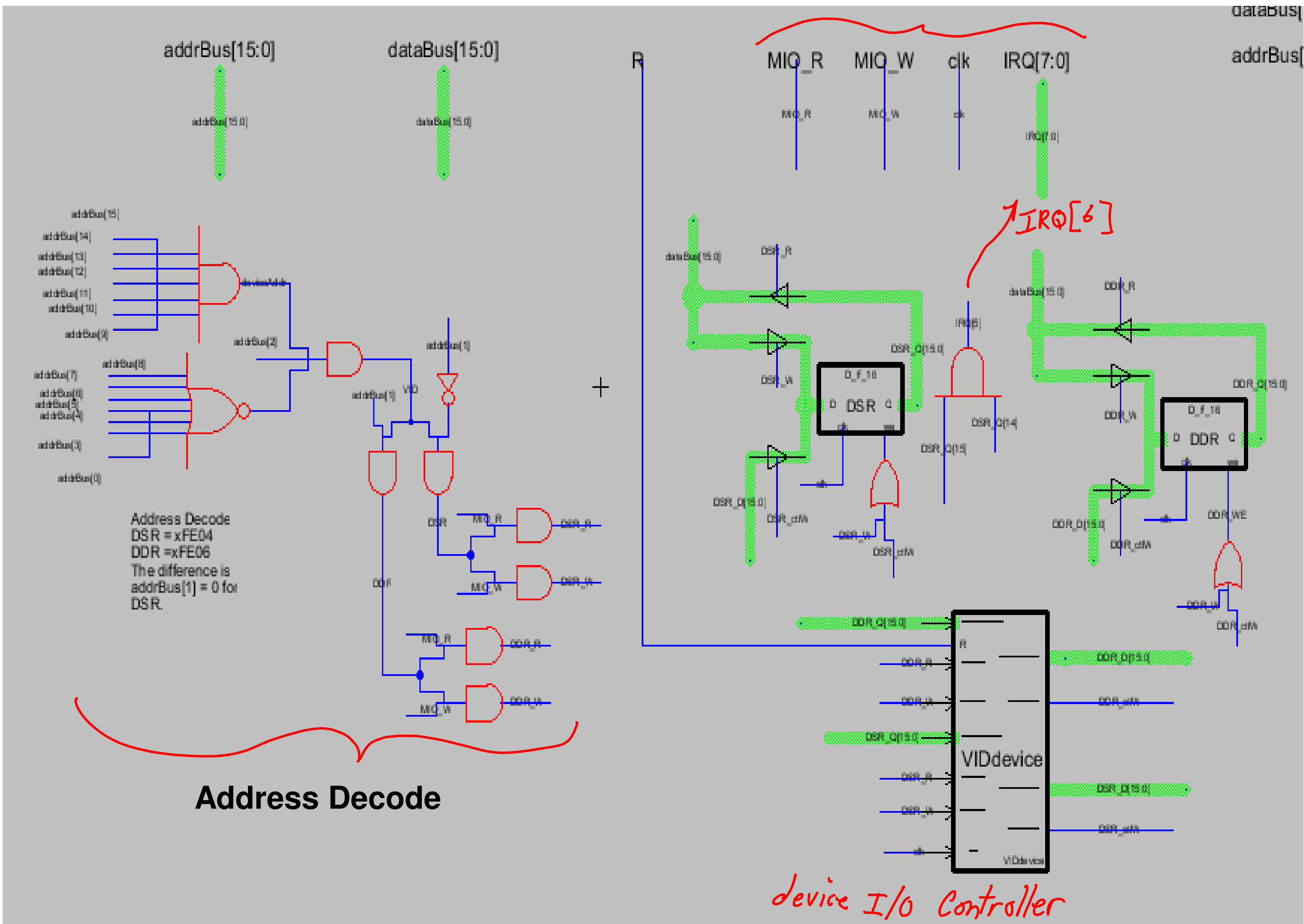
memory "shadowed" by devices

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

This portion of physical memory is not accessible.



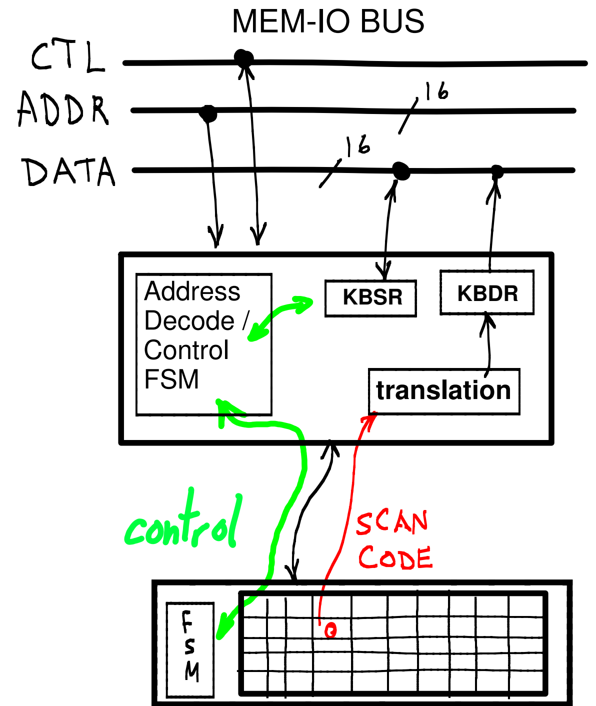
disp_ctl, The Display Device I/O Interface



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
KBDRptr:
        .FILL xFE02
```

PC-relative offset calculated by assembler

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

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

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 <== x**FE02**
ADDR_bus <== **MAR**

Control bus sends **MIO_R**

Address decode
recognizes xFE02

MDR <== **KBDR**

3. R0 <== **MDR**

What is Non-Memory Mapped I/O?

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

in r4, x0D

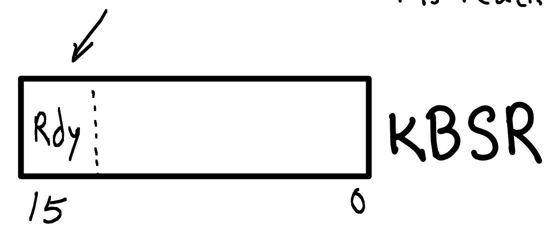
--- Control-Bus signal "IO": 0, access memory; 1, access I/O device (uses same address wires)

Device Control – 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

R∅ ← KBDR

continue

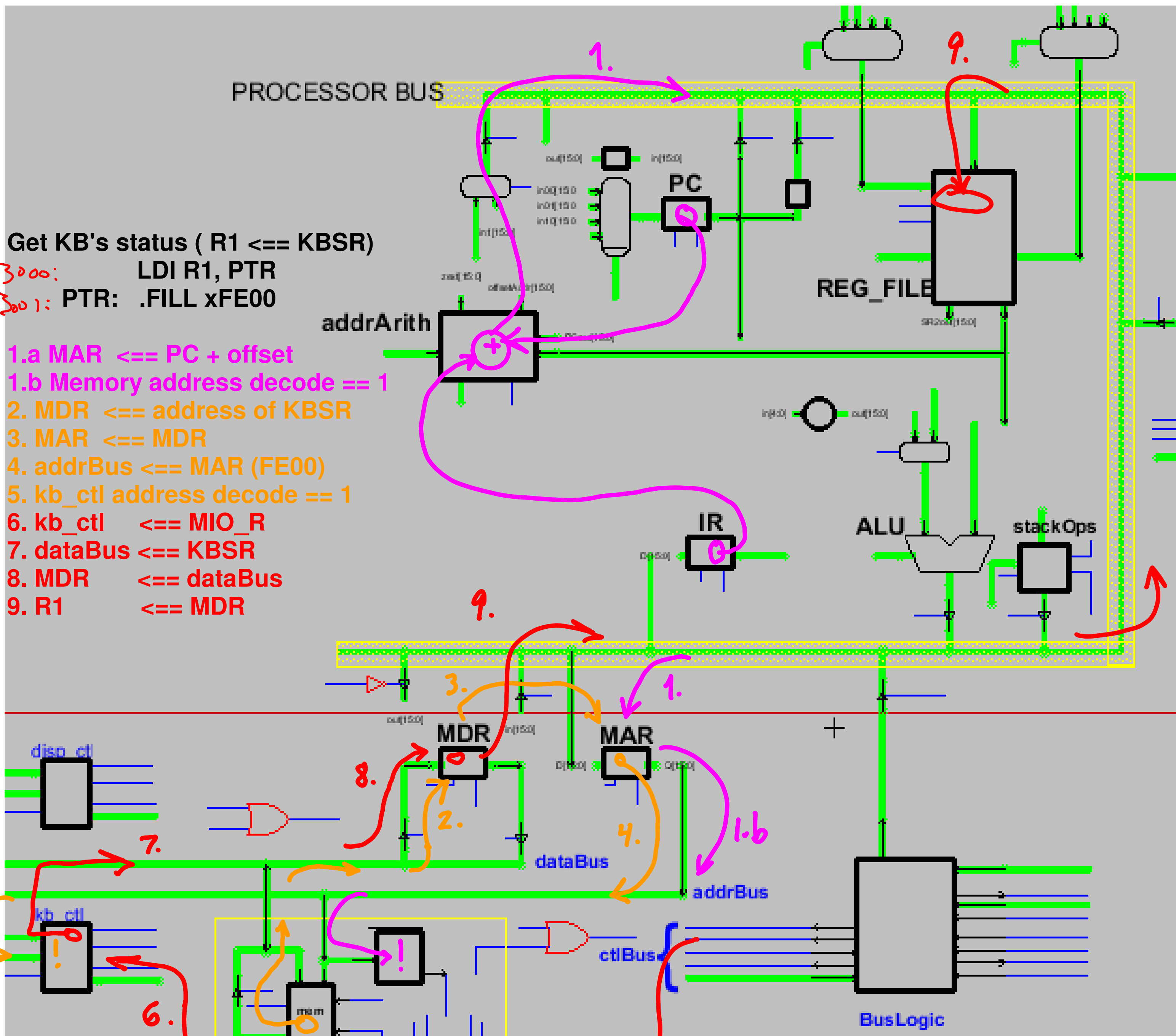
LC3 Operating System I/O functions

Depends on OS version.

TRAP <n>	Assembly-pseudonymn	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?



Suppose:
PC == 1000

Memory

...
1000: LDI R1, (+1)
1001: BRnzp (+1)
1002: FE00
...

instruction fetch:
IR <== 1010001000000001
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**

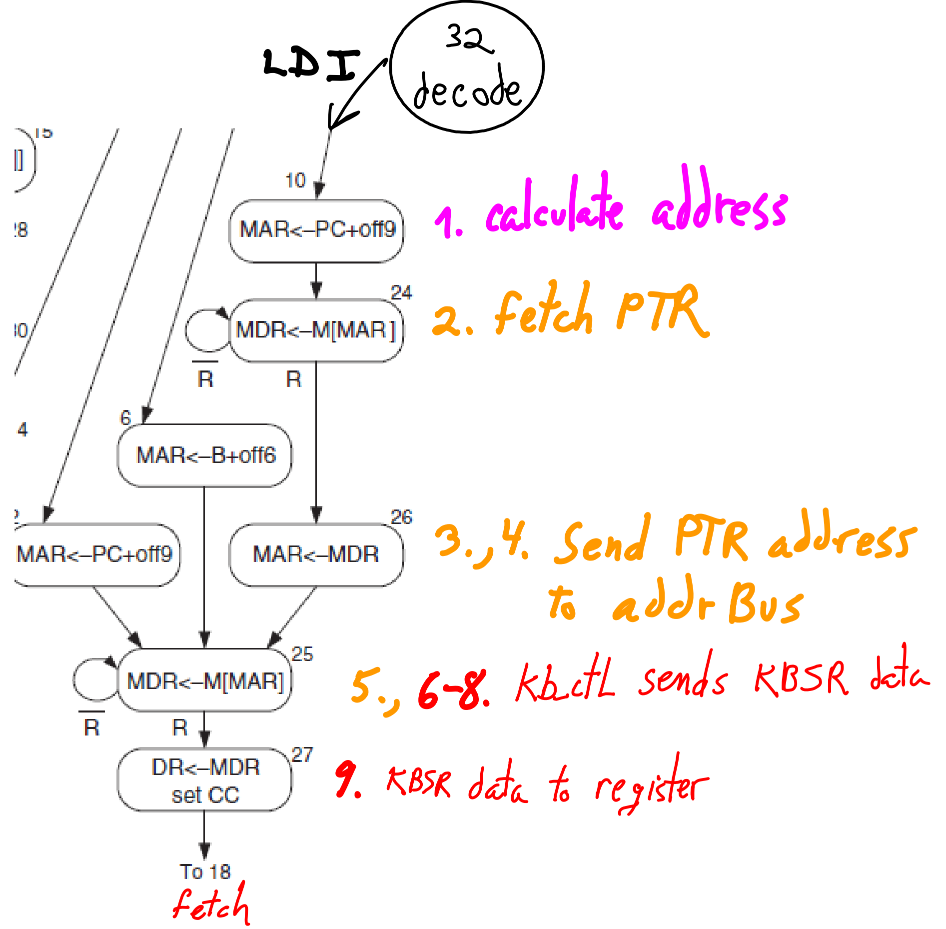
5. kb_ctl recognizes address:
addrBus[15:0] == 7'hFE00

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**



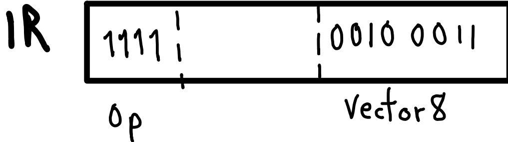
LDI
32
decode
10
MAR<–PC+off9
1. calculate address
24
MDR<–M[MAR]
2. fetch PTR
R
R
6
MAR<–B+off6
MAR<–PC+off9
26
MAR<–MDR
3., 4. Send PTR address to addr Bus
25
MDR<–M[MAR]
5., 6-8. kbctL sends KBSR data
R
R
27
DR<–MDR
set CC
9. KBSR data to register
To 18
fetch

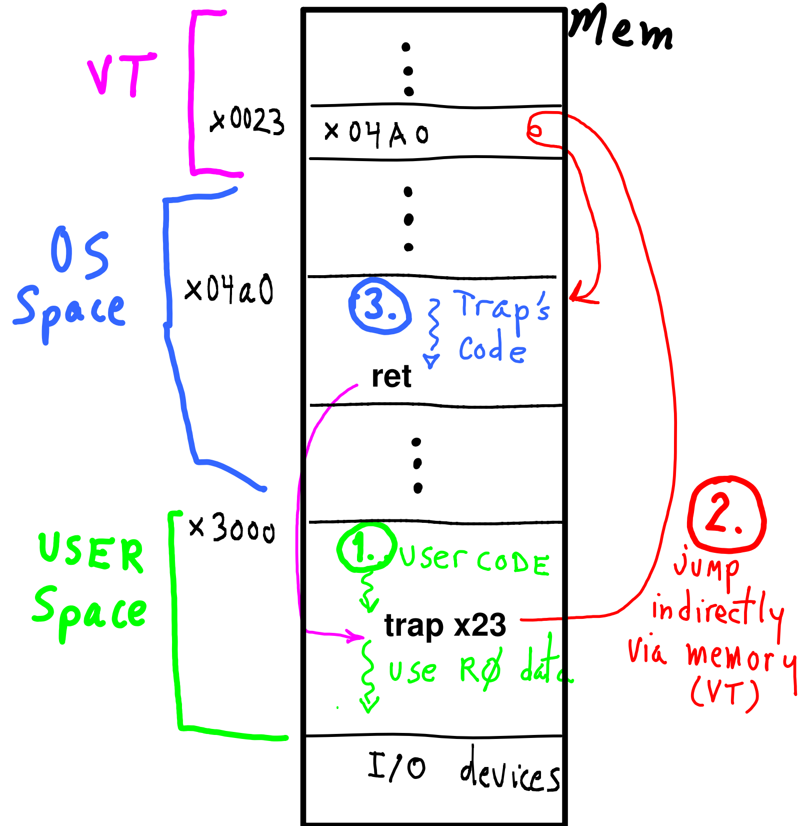
Trap instruction execution

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] **]**





1. USER CODE executes Trap
2. jump: PC loaded from VT
3. Trap service code executes
4. ret executed

"ret" = "jmp R7"
(PC ← R7)

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.

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

```
;=======================
;= 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
```

get input:

```
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.

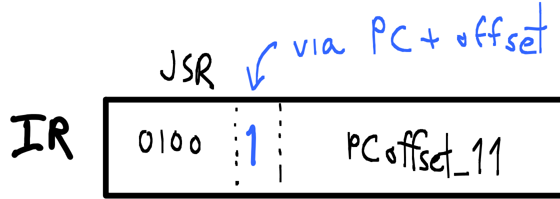
Who (caller/callee) is responsible for saving/restoring registers?

If code is generated by compiler? ==> by compiler convention.

(What about CC?)

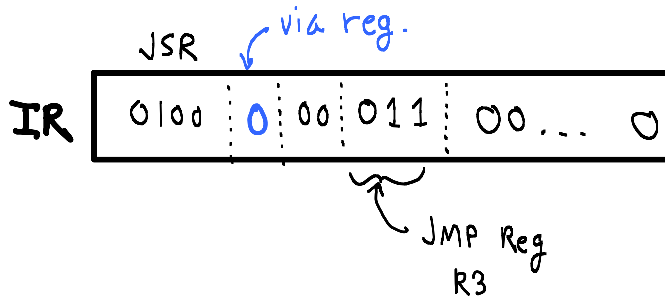
```
.orig x3000
   ...              ;= do stuff
jsr foo             ;= jump to foo()
   ...              ;= use foo()'s result
jsr foo             ;= jump to foo()  again
   ...

foo:                ;= code for foo()
   ...
ret                 ;= jmp r7
```



"JSR foo"

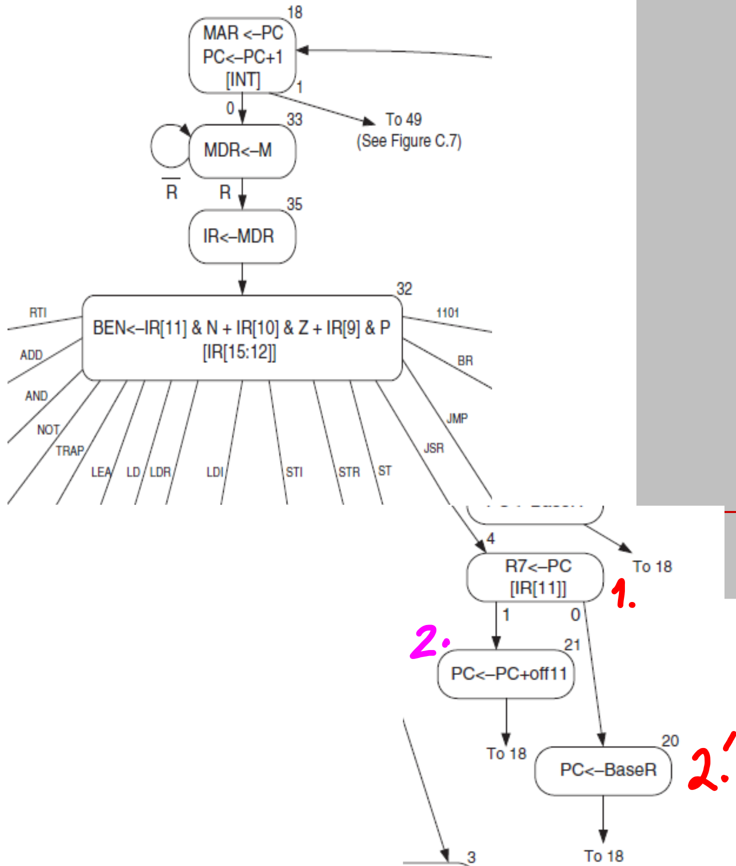
R7 ← PC
PC ← PC + offset

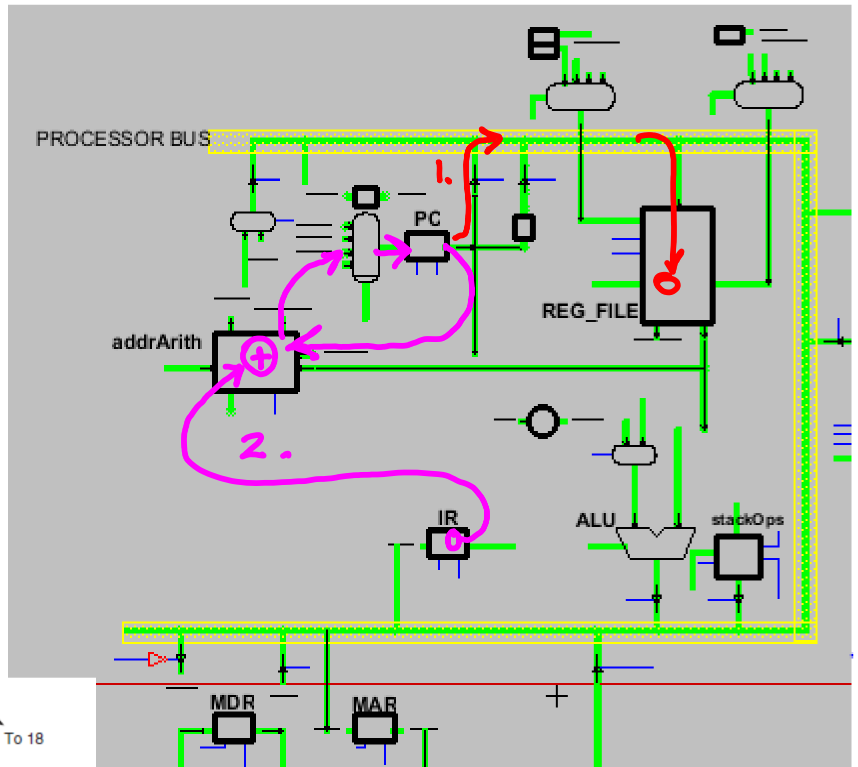


"JSRR r3"

R7 ← PC
PC ← reg

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

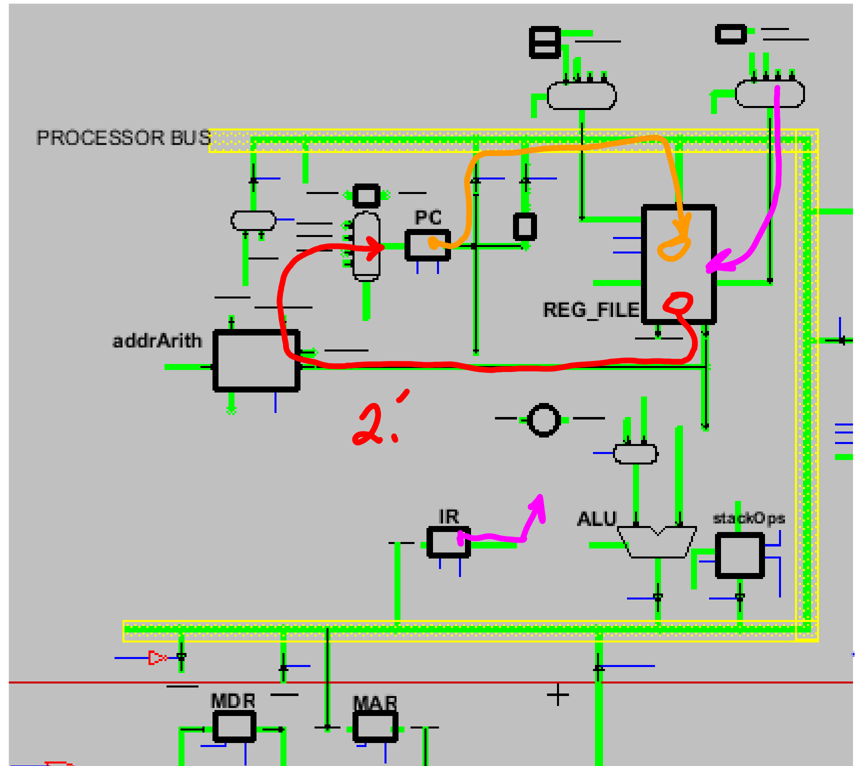




See complete state:

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**]** = **Condition Codes** (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).

Puzzler, PP Fig. 9.8 (see Figs C.2 + C.7, FSM states for Trap)

```
;===================
;=    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"?

<u>Where</u> did this Trap x23 "IN" code come from?
If its' in <u>OUR</u> LC3 memory, <u>how</u> 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