How do we know a device is ready
(has data or wants data)?

(1) Ask: poll the status register

(2) Have device tell us, don't ask device.

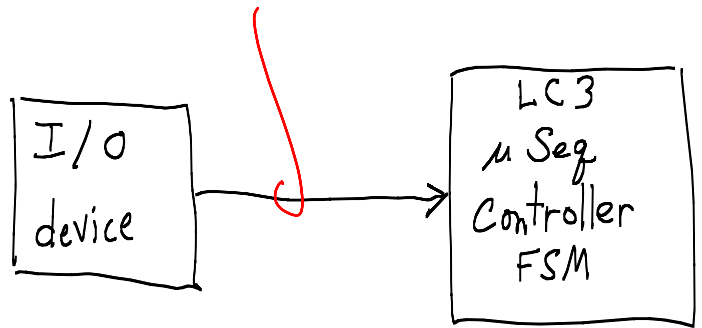
(a) who is "us"?

(b) how can a device talk to us?

(c) when can/will it speak?

(d) what should we do then?

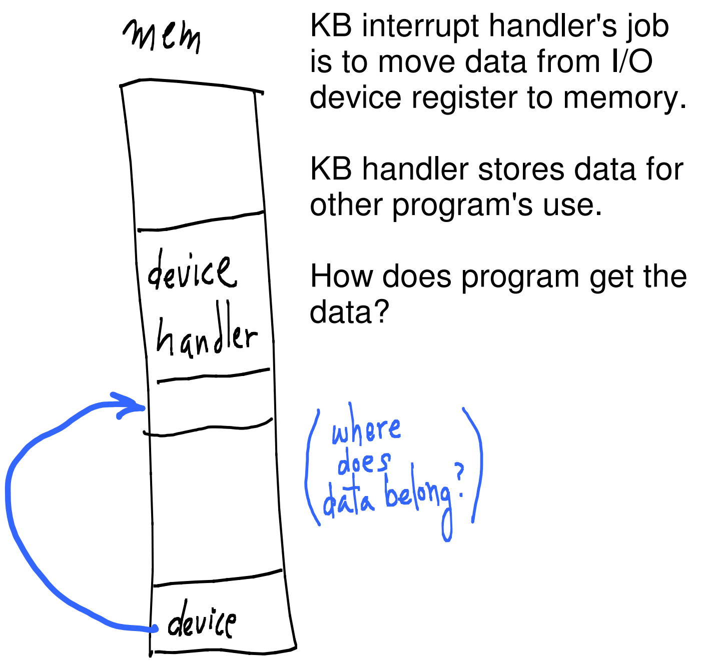
(e) what about the currently executing program?



data flow

Logically

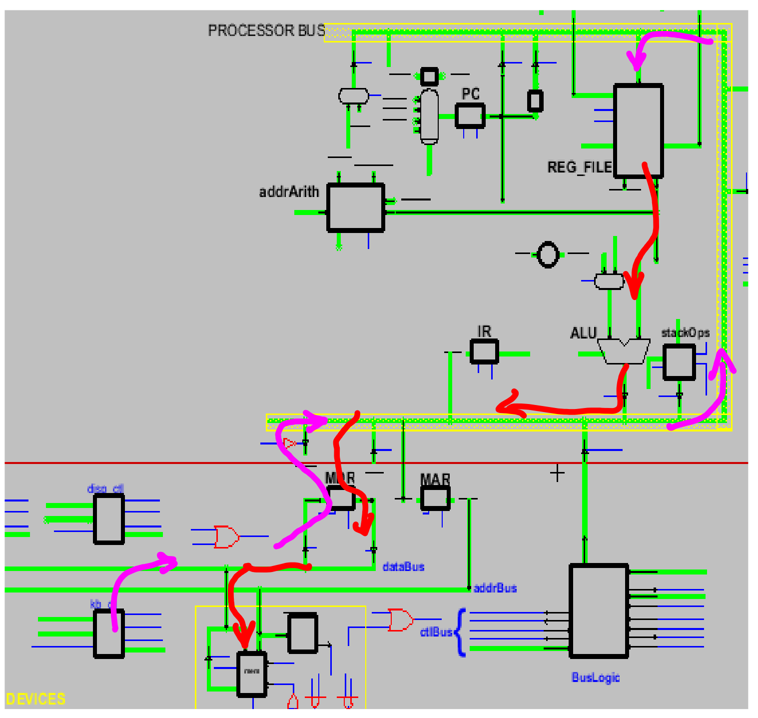
physically



KB interrupt handler's job is to move data from I/O device register to memory.

KB handler stores data for other program's use.

How does program get the data?



Mem ← dataBus ← MDR ← register ← MDR ← dataBus ← device register

device → INT → LC3 FSM Control

fetch

18
MAR←PC
PC++
[INT]

0 → the rest of fetch ... decode → RTI ... restore state, restore PC

1 → interrupt process ... save state, load PC

x0100 | TVT
x01FF | PTR 1234 | IVT, EVT
x1234 | Interrupt handler
RTI
do work
INT occurs

→ jmp via memory
→ RTI (reload state, restart)

Device does pseudo "Trap" on its own; gets OS's attention.

Interrupt handler code executes RTI (not RET) to restore interrupted program's state, restarting fetch cycle at original location.

Running program never knows anything happened (unless checks w/ OS).

If OS is designed in such a way that work can be done even while IO devices are busy, this saves a lot of cycles vs. polling.

Save state:
save PC, PSR (priv, prio, cc)
save regs

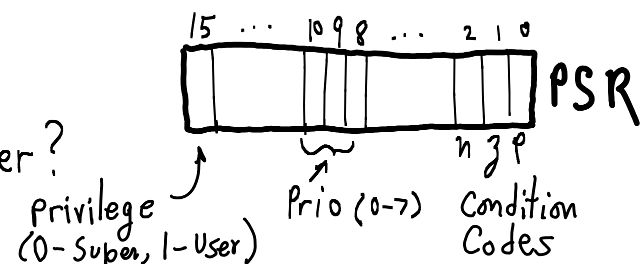
→ where should these be saved? when? By whom?

Restore state:
restore regs
restore PSR, PC

→ who does this?

} is this stack push/pop?

0) what part of state saving must be done by FSM, which in software?
1) how do devices send interrupt signal?
2) what if more than one device signals?
3) How to handle interrupting an interrupt handler?
4) Interrupt the same handler again?

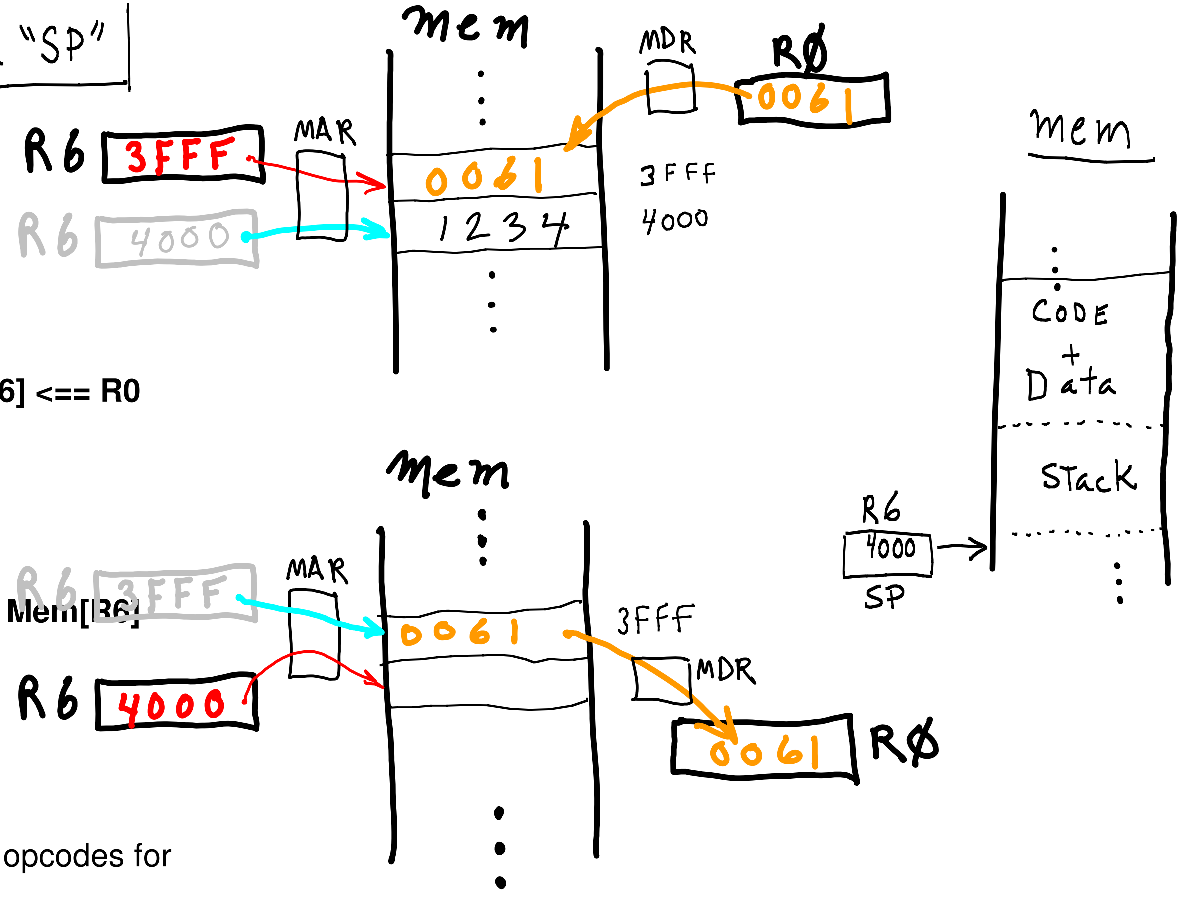


The Stack | R6 aka "SP"

```
jsr PUSH_R0
...
jsr POP_R0
```

```
PUSH_R0:
add r6, r6, #-1   ;= R6--
str r0, r6, #0    ;= Mem[R6] <== R0
ret
```

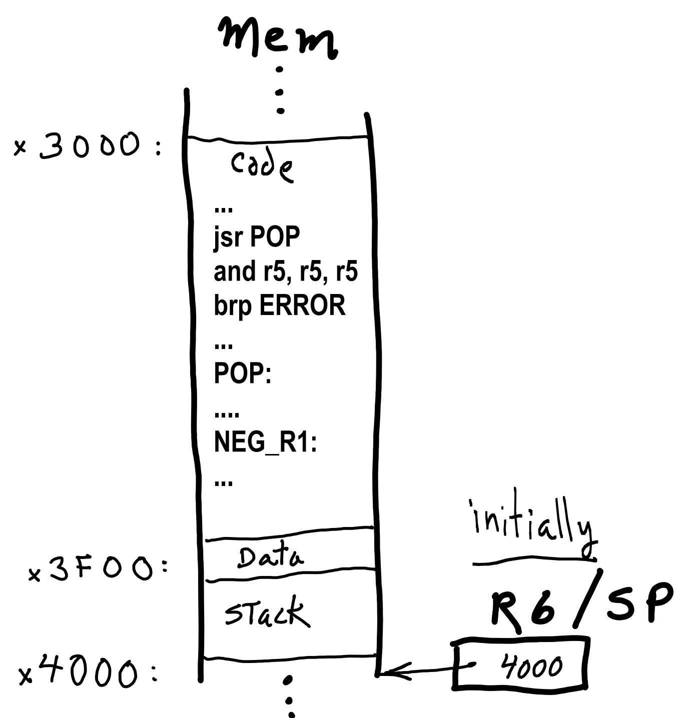
```
POP_R0:
ldr r0, r6, #0    ;= R0 <== Mem[R6]
add r6, r6, #1    ;= R6++
ret
```



Could we implement new opcodes for push/pop.

Other times we want HW push/pop?

```
;=========================
;=     POP
;= Returns popped value in R0 and 0 in R5.
;= If stack underflow, aborts and returns 1 in R5.
;=========================
POP:

and r5, r6, #0              ; retVal <== 0

ld r1, stack_bottom         ; r1 <== stackbottom
jsr NEG_R1                  ; r1 <== -stackbottom
add r1, r6, r1              ; sp - stackbottom

brnz else                   ; if (sp - stack_bottom) < 0
jsr POP_R0                  ;       do pop
ret                         ;       ret( 0 )
else:                       ; else
add r5, r5, #1              ;     retVal++
ret                         ;     ret( 1 )

stack_bottom: .FILL x4000

NEG_R1:
not r1, r1
add r1, r1, #1
ret
```

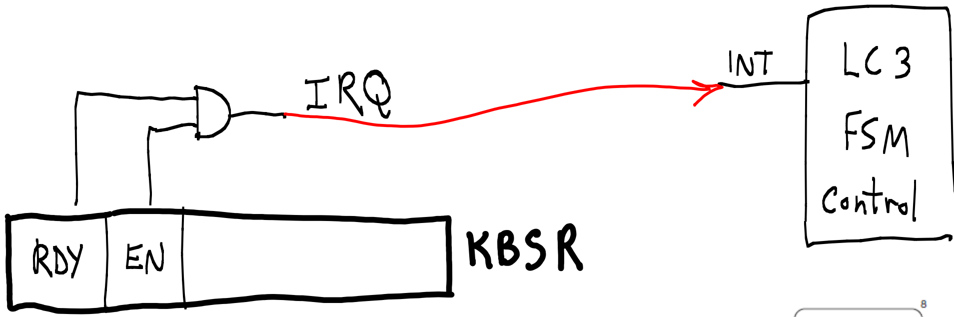


Stack overflow? Check that PUSH does not have SP = x3F00.

What about clobbering registers? Callee save?
(R0 is clobbered anyway on pop, and SP should not be saved.)

Back To Interrupts

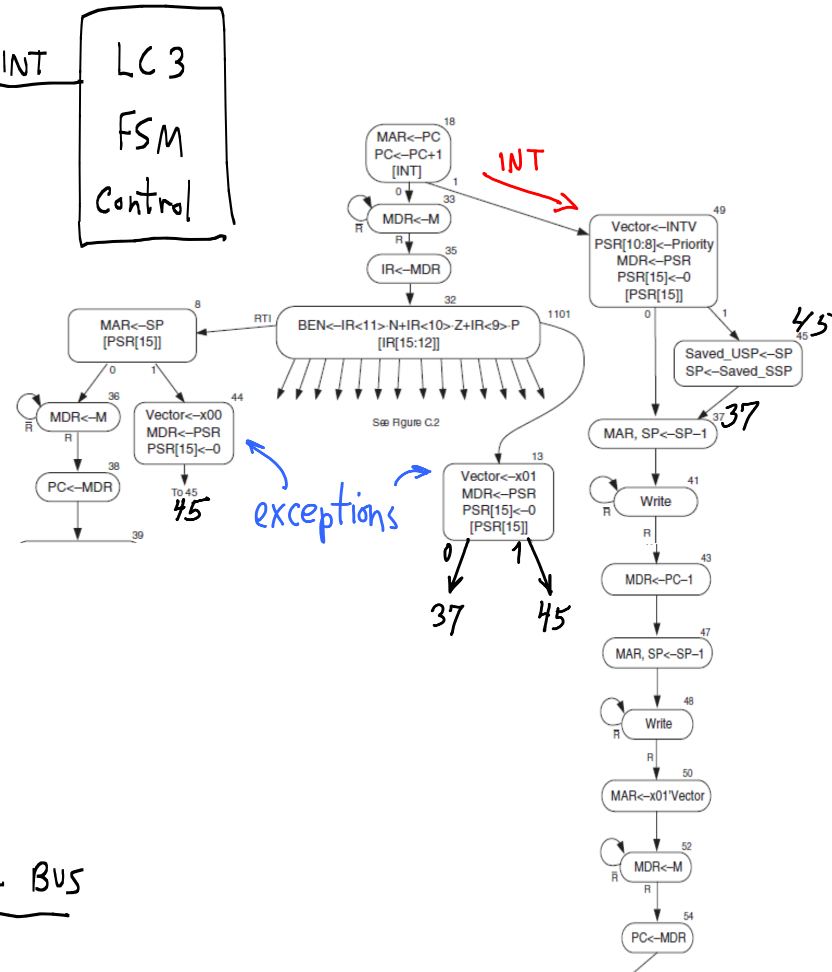
Enabling interrupts:
Set KBSR[14] = 1 allows
controller to be interrupted.



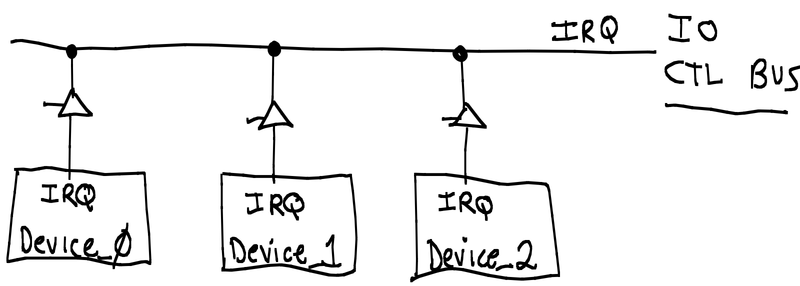
15 14

Programmer sets by writing into KBSR

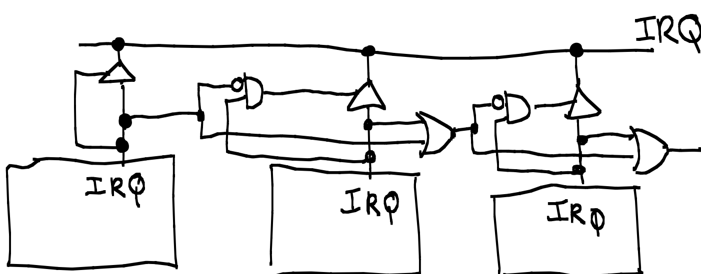
device sets when it puts new data into KBDR
(resets after a read of KBDR)



If only one device, then ok, but what
about multiple devices?



How can we have exactly one device driving the IRQ line? What if two devices want service at the same time? Priority daisy chain:



Highest Priority → Lower →

The Daisy Chain setup prevents a low-priority device from sending an IRQ when a high-priority device sends IRQ.

Note that a 1 propagates through all the OR gates to its right, and also disables all the tri-states to its right.

Details: how does higher-priority device interrupt lower priority?
How to tell which device caused interrupt?

Priority Encoding

ENCODER sends priority code of highest priority device whose IRQ line =1.

KB: IRQ[4] ==> A_4 ==> code 100.

PRIO Encoder

A_0 A_1 ... A_7 y 3 Code

IRQ Net | IRQ Disk | IRQ KB

IRQ[7:0] bus is part of memory-IO ctl bus

Bus

addrBus

ctlBus

MIO_W MIO_R MIO_READY bus_clk IRQ[7:0]

IRQ[7:0]

Vector_out[15:0] PSR[10:8] INT VectorMUX[1:0] LD_Vector MIO_W MIO_R MIO_READY bus_clk IRQ[7:0] MIO_EN R_W R sys_clk Bus_Logic

INT

BusLogic

Interrupted program has priority PSR[10:8].

KB is on IRQ[4].

Priority encoder has 8 inputs and 8 possible codes to output. What if none of the inputs is non-zero?

The priority code '000' means either (1) no IRQ is non-zero, or (2) IRQ[0] is 1.

Extra output "notZero" tells us which is the case.

P&P's comparator was ">=", not ">". Why?

EN IRQ[7:0] unused

log_0 log_0 log_0 log_0 log_0

KB IRQ[4] DISP IRQ[5] MCR IRQ[6]

Priority Encoder: in0 in1 in2 in3 in4 in5 in6 in7 notZero Priority PrioEncoder

Unused IRQ inputs are set to 0.

VectorMUX[1:0] VectorMUX[1:0]

IntPriority[2:0] notZero not Zero INT INT

Comparator: A[2:0] A>B? B[2:0] PrioComp IntPrioGreater

IntPriority[2:0]

IntPriority > PSR.Priority

PSR[10:8] PSR[10:8]

from PSR, current code's Priority

Keyboard: kb_ctl, IRQ[4], priority 4

Display: disp_ctl, IRQ[5]

MCR: IRQ[6]

Priority Coding

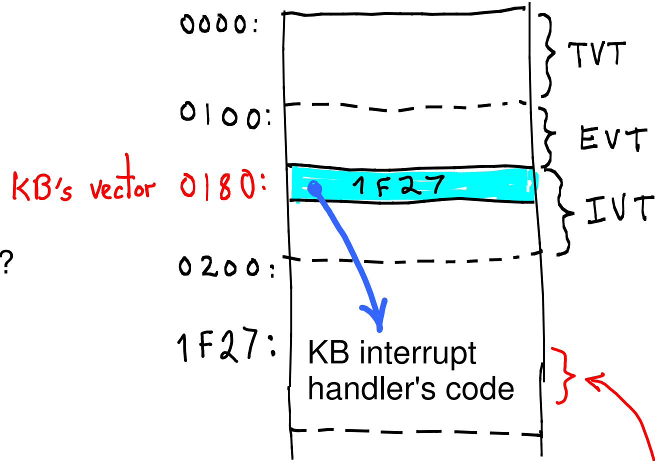
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Code
1	X	X	X	X	X	X	X	111
0	1	X	X	X	X	X	X	110
		.	.	.				...
0	0	0	0	0	0	0	X	000

Processor interrupted, now what?
How to jump, where to jump?
Maybe like a trap?

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

Interrupt:
  PC <== Mem[**?**]

How to address into Vector Table?
(We'll come back to that.)



Interrupt routines can be anywhere. OS fills in TVT, EVT, and IVT at boot-time.

— Jump similarly to a Trap instruction: via IVT.

— Each device has its own IVT slot

KB Driver 3 Parts:

– Int. handler

```
;;;------------------------------
;;;-- kbInt - IVT x180:
;;;--     Keyboard interrupt service
;;;------------------------------
kb_INT_BEGIN:
        ;;;---- Disable interrupts, KBSR[14] <== 0.
        ;;;---- Read KBDR, store data.
        LDI R0, KBDR
        STI R0, KB_Buff_head
        ;;;-- Move head pointer.
        ;;;-- Enable interrupts, KBSR[14] <== 1.
kb_INT_END: RTI
```

– init VT

```
kb_init_BEGIN:
  ;;;-- Set-up interrupt vector.
  LEA R1, kb_INT_BEGIN
  STI  R1, KB_INT_vector
  ;;;-- Set-up KB_Data_Buffer.
  ;;;-- Set-up Trap routine vector.
  ;;;-- Enable KB interrupts.
kb_init_END: RET
```

– data service

```
kb_Trap_BEGIN:
    ;;;-- KB data-request service.
kb_Trap_END: RET


kb_ConstantDataArea:
   KB_INT_vector:         .FILL x0180
   KB_TRAP_vector :       .FILL x0033
   KBSR:                  .FILL xFE00
   KBDR:                  .FILL xFE02
kb_VariableDataArea:
   KB_Data_Buffer:        .BLKW #80
   KB_Buff_head:          .BLKW #1
```

```
;;;==============================
;;;-- OS boot/initialization
;;;==============================
initOS_BEGIN:

 ;;---- Set up super's stack.
 ld  _sp, SUPER_STACK_ADDR

 ;;---- Init traps, exceptions, and interrupts
 _jsr( kb_init_BEGIN )

 ;;---- jump to main(), never returns.
 _intsOn
 lea r7, mainOS_BEGIN
 jmp r7

initOS_END:
```

USER PROG

```
TRAP x33    ;;;---- Get KB data
 ...        ;;;---- Use KB data
```

Addressing Into VT?

Also part of BusLogic is generating the address of the interrupting device's vector.

E.g. KB interrupt

Priority Encoder sends IntPriority (100)

IntPriority addresses into INTV_ROM

INTV_ROM[100] == x80

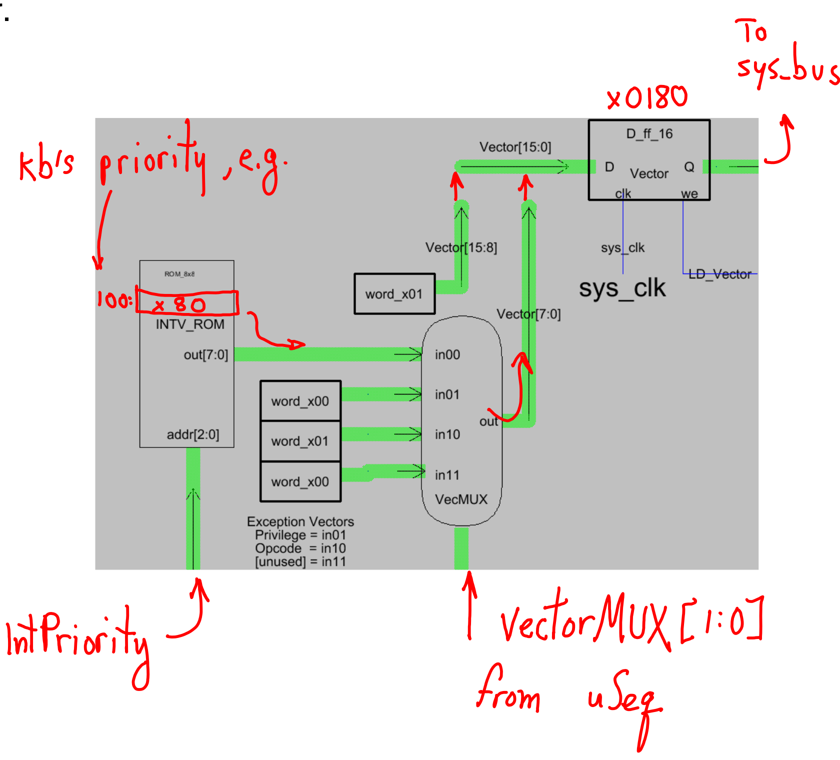
Vector's prefix comes from word_x01

Vector <== { x01, x80 } == x0180

NOTE: The VectorMUX[1:0] control signal selects according to whether this is a,

- 2'b00: I/O hardware interrupt
- 2'b01: Privilege exception
- 2'b10: Opcode exception

This is all just a lookup table, but addressing is split into two parts.



making the jump using Vector register

if INT = 1
and state-18:

JUMP to Handler:

MAR <== Vector

MDR <== VT entry

PC <== MDR

This is the jump to the interrupt handler code. But what about the executing code's state?

How to save state of interrupted program?

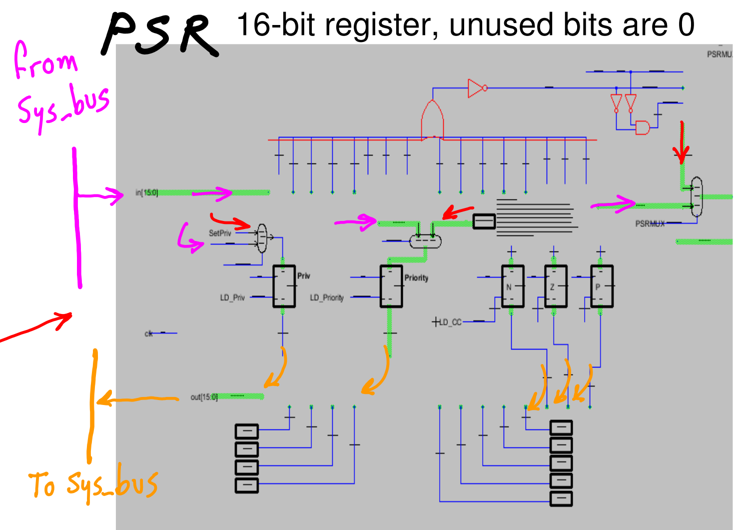
Push essentials to STACK (PSR, PC).

But what about stack pointer, R6?
User's SP or Super's SP?

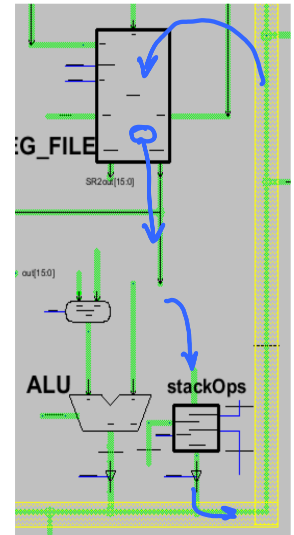
How many users?
How many are being interrupted? Just one.

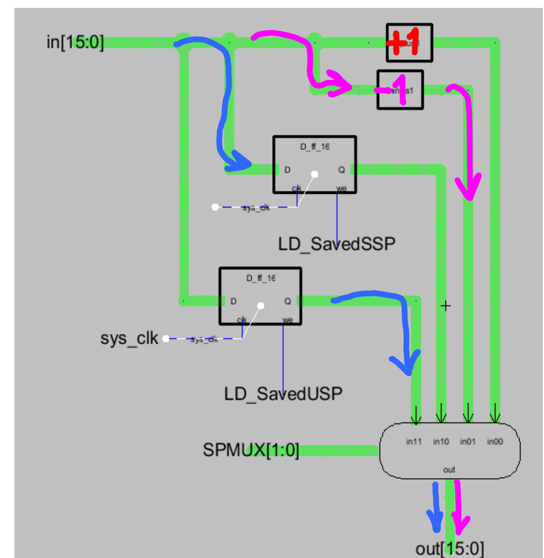
PSR input is muxed: if select (PSRMUX) is
- 1'b0: input from sys_bus
- 1'b1: input from control's SetPriv, IntPriority, and CC logic (separate load signals)

LC3 uSeq Control outputs, plus CC logic

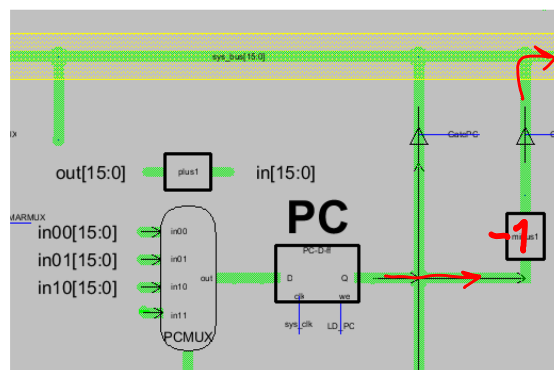


State	Operation	Annotation
49	Vector<–INTV	Load Vector register w/ VT slot address
	PSR[10:8]<–Priority	New PSR will have priority of interrupting device.*
	MDR<–PSR	OLD (current) PSR is being saved
	PSR[15]<–0	New PSR will have SUPER Privilege (SetPriv)
	[PSR[15]] → 0 / 1	1: Interrupted code had user privilege.
45	Saved_USP<–SP	Save user's SP
	SP<–Saved_SSP	Switch to SUPER's stack
37	MAR, SP<–SP–1	SP, MAR ← SP−1. Pushing OLD PSR onto SUPER's stack
41	Write (R)	MDR = OLD PSR, MAR = decremented SP
43	MDR<–PC–1	Saved PC points to interrupted instruction
47	MAR, SP<–SP–1	SP, MAR ← SP−1
48	Write (R)	PC pushed
50	MAR<–x01'Vector *	PC ← Mem[Vector]





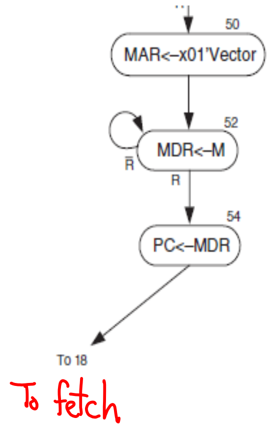
TO MAR   To SP



To MDR

* Our LC3 is slightly different: PSR.Priority gets loaded with 111, Vector register has all 16 bits of VT address.

This is the jump to the handler:

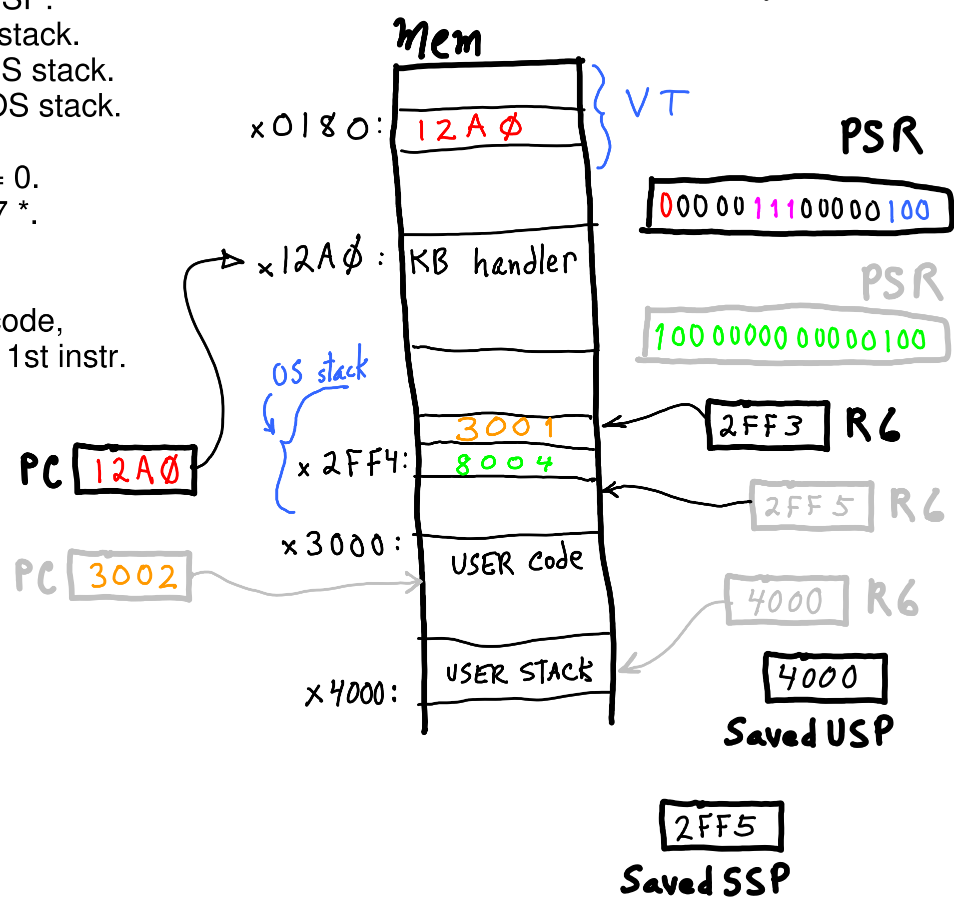


--- User SP saved.
--- R6 <== savedSSP:
SP set to OS stack.
--- User PSR on OS stack.
--- User PC-1 on OS stack.

--- PSR.Privilege = 0.
--- PSR.Priority = 7 *.
--- SP <== SP - 2.

--- PC at handler code,
ready to fetch 1st instr.

The overall effect



2FF5
Saved SSP

Suppose we had interrupted an interrupt routine. What happens next after this current interrupt routine finishes?

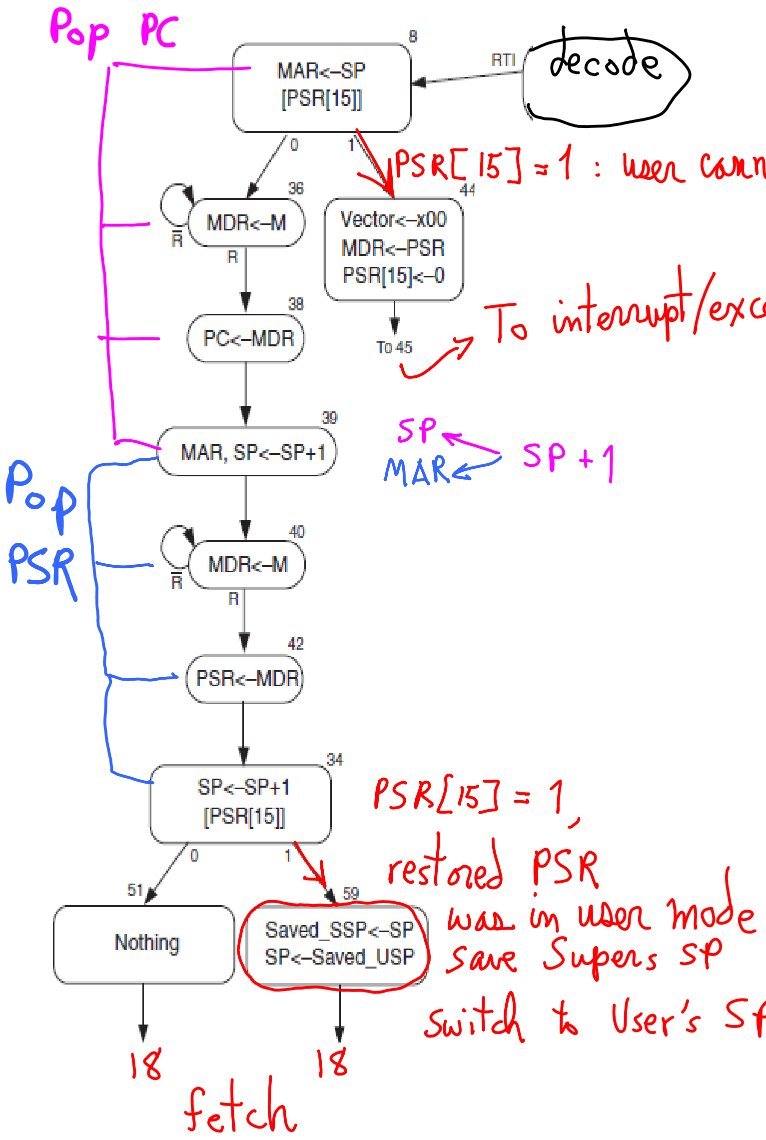
RegFile source and destination select

SR1MUX	SR1out data is	
1'b00:	RegFile[IR[11:9]]	(SR field)
1'b01:	RegFile[IR[8:6]]	(SR1 field)
1'b10:	RegFile[110]	(R6/SP)

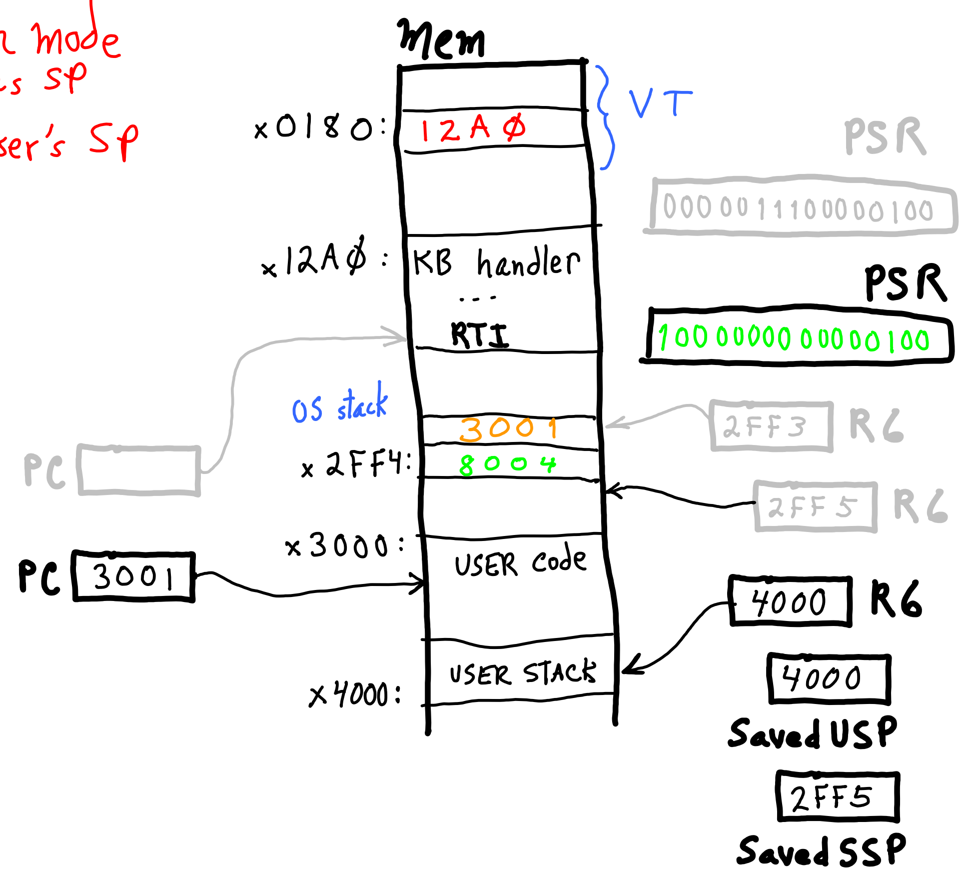
DRMUX	data in goes to	
1'b00:	RegFile[IR[11:9]]	(DR field)
1'b01:	RegFile[111]	(R7)
1'b10:	RegFile[110]	(R6/SP)

The RTI instruction: restore interrupted program

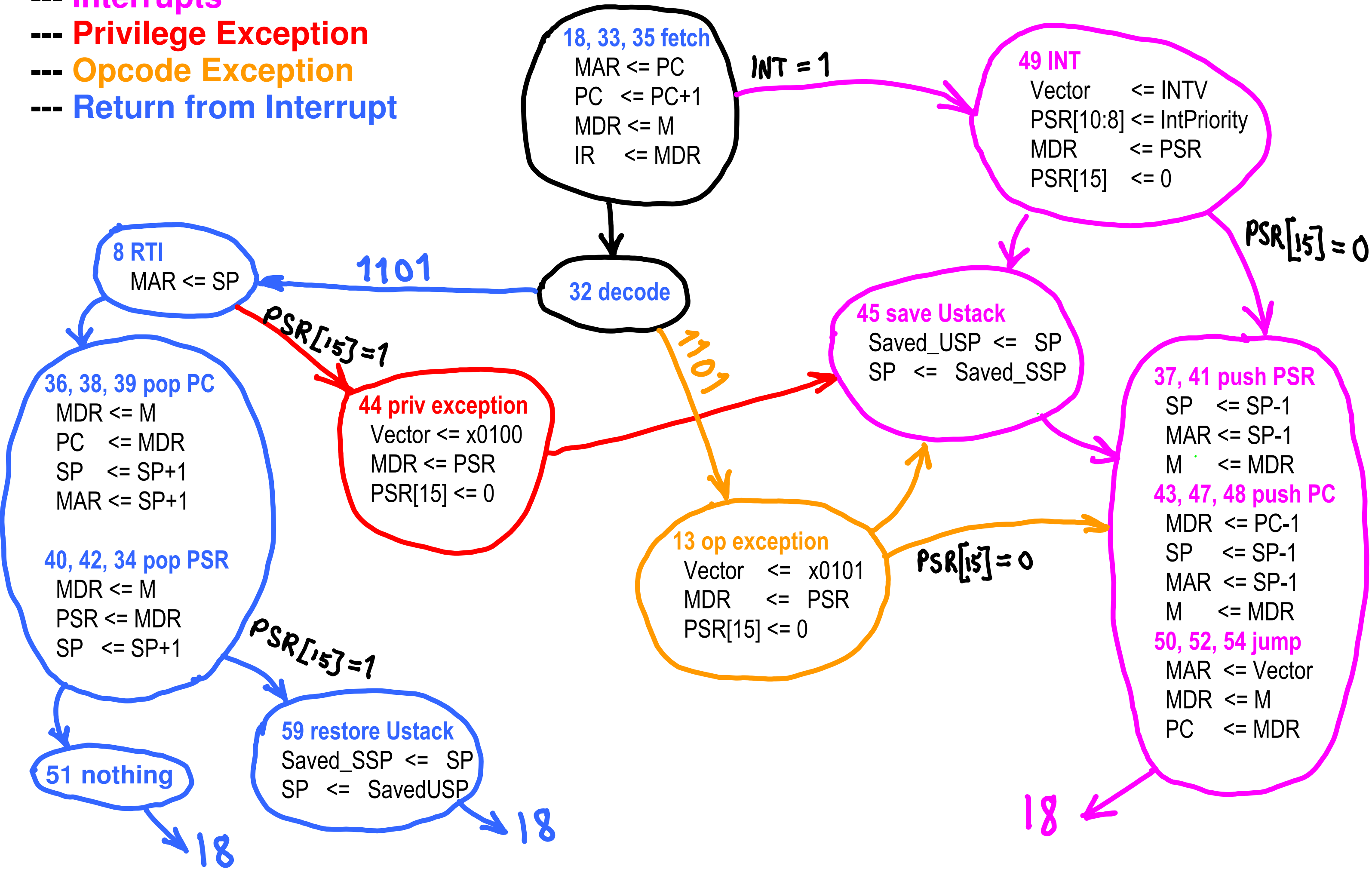
1000 ¦ 00 ... 000 IR



The overall effect



LC3 FSM control for
--- Interrupts
--- Privilege Exception
--- Opcode Exception
--- Return from Interrupt



Simplified Vect_Reg input.

Some parts of P&P's hardware could be simplified for the sake of easier understanding.

The Vector register is loaded from what actually is a ROM, but doesn't look like one:
---- address inputs
Priority bits (3)
VectorMUX bits (2)
---- output
16-bit Vector Table address

We could implement this as a 32-word ROM. Addresses that start with 00 would be for hardware interrupts. For instance, address 00100 (Priority = 100 = 4) would be for the KB interrupt. That word would contain the 16-bit address x0180.

All addresses that start with 01 (01000 to 01111) would be for the Privilege exception, and contain the 16-bit address x0100. The low 3 bits are in effect ignored.

Addresses that start with 10 (10000 to 10111) would be for the Illegal Opcode exception, and contain the 16-bit address x0101.

Of the 32 words, 22 are redundant. Space is wasted, but life is simpler?