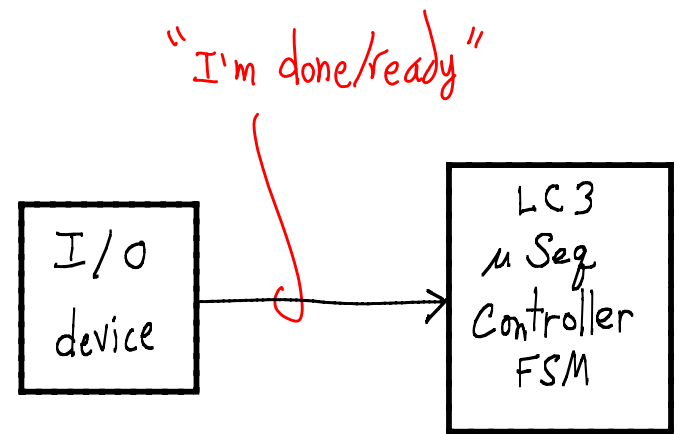


How do we know a device is ready (has data or wants data)?

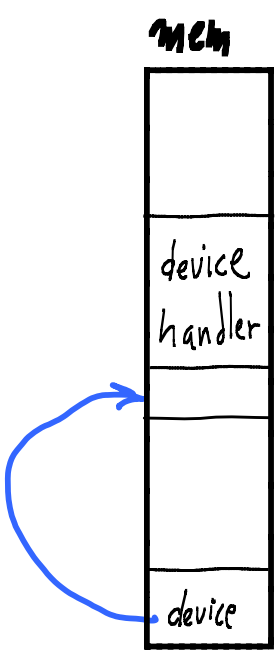
- (1) Ask: poll the status register
- (2) Have device tell us, don't ask device.
 - (a) who is "us"?
 - (b) how can a device talk to us?
 - (c) when can/will it speak?
 - (d) what should we do then?
 - (e) what about the currently executing program?



data flow

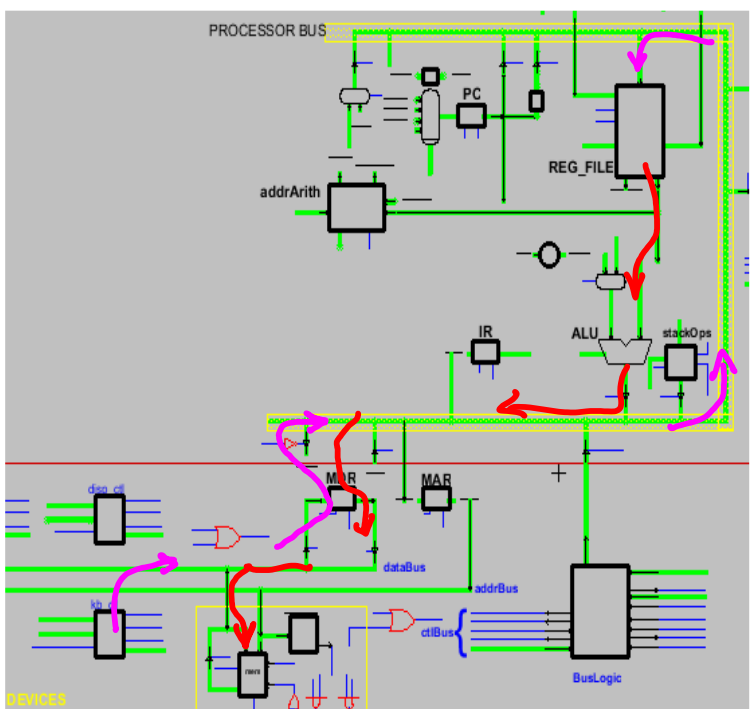
Logically

physically

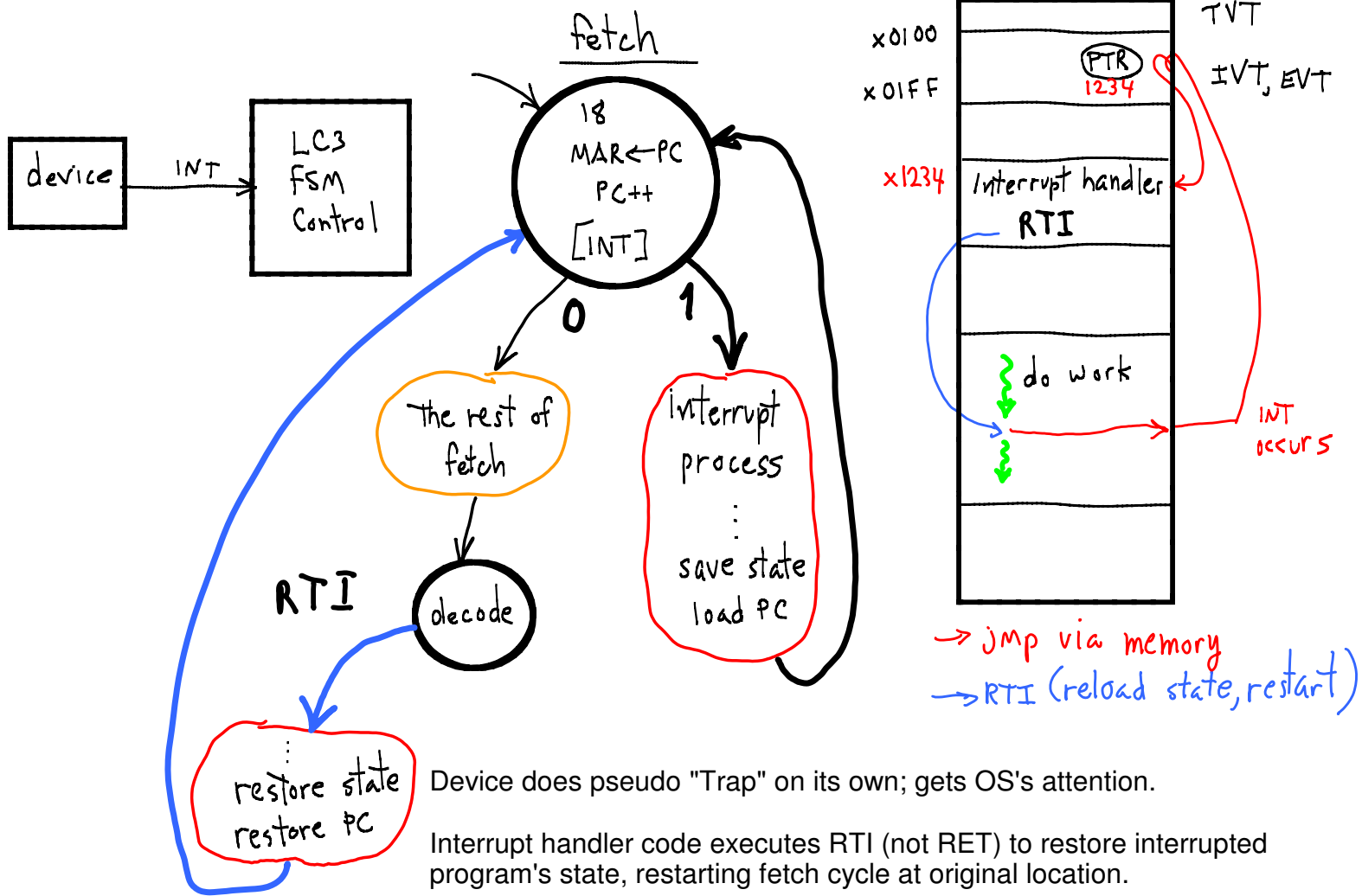


KB interrupt handler's job is to move data from I/O device register to memory.
 KB handler stores data for other program's use.
 How does program get the data?

(where does data belong?)



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



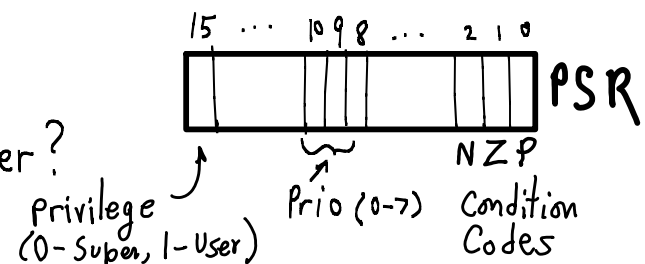
Device does pseudo "Trap" on its own; gets OS's attention.
 Interrupt handler code executes RTI (not RET) to restore interrupted program's state, restarting fetch cycle at original location.
 Running program never knows anything happened (unless checks w/ OS).
 If OS is designed in such a way that work 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

Restore state:
 restore regs
 restore PSR, PC

→ where should these be saved?
 when? By whom?
 } is this stack
 push/pop?
 → who does this?

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

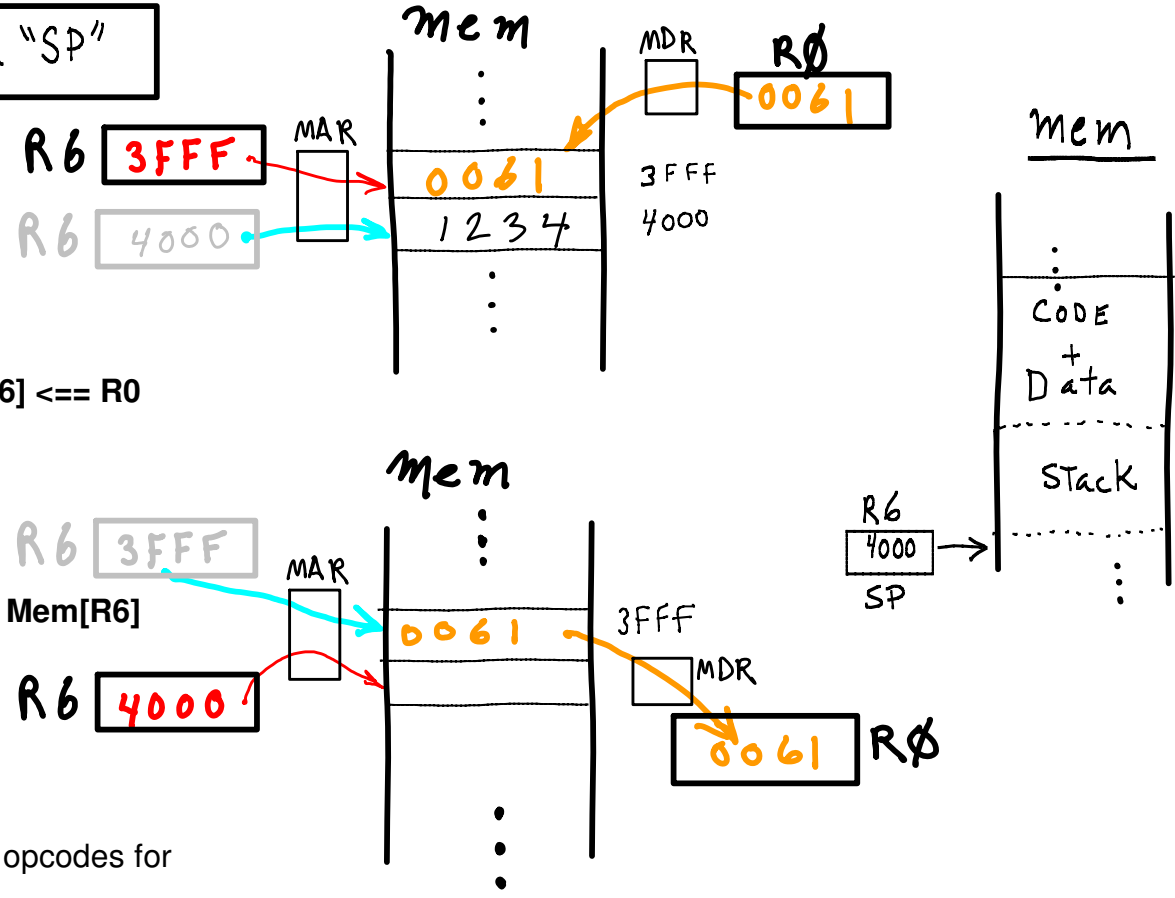


The Stack R6 aka "SP"

```
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

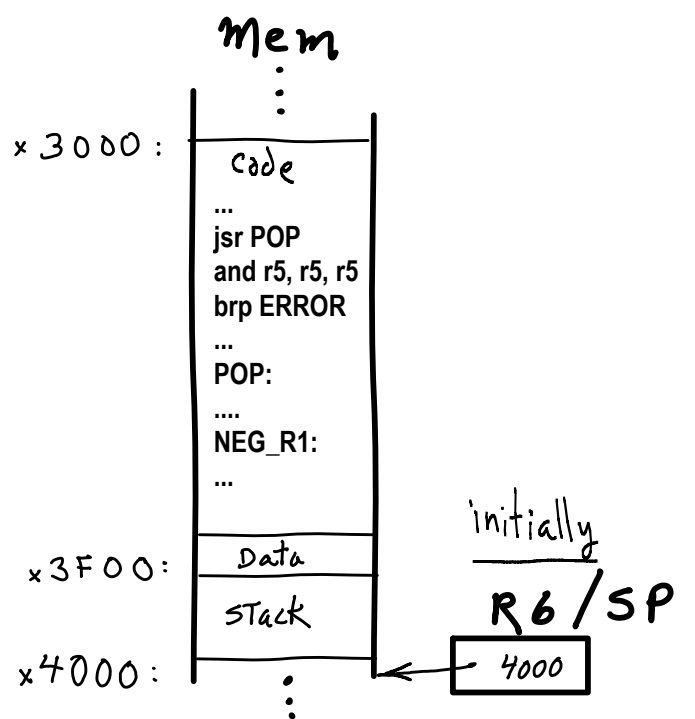
ld r1, stack_bottom ; r1 <== stackbottom
jsr NEG_R1 ; r1 <== -stackbottom
add r1, r6, r1 ; sp - stackbottom

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
```

Check stack bounds?

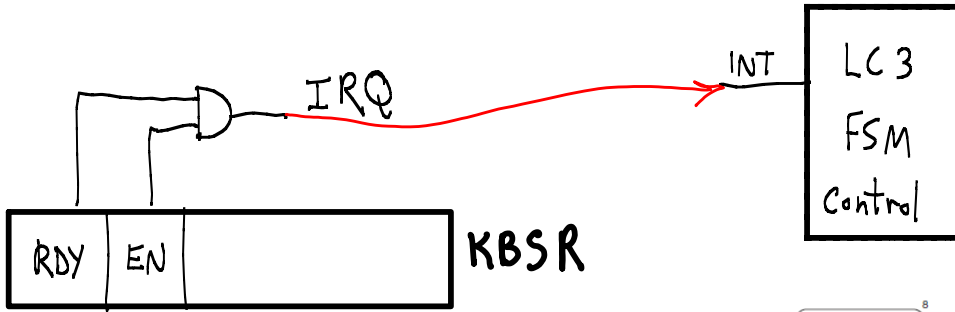


Stack overflow? Check that PUSH does not have SP = x3F00.

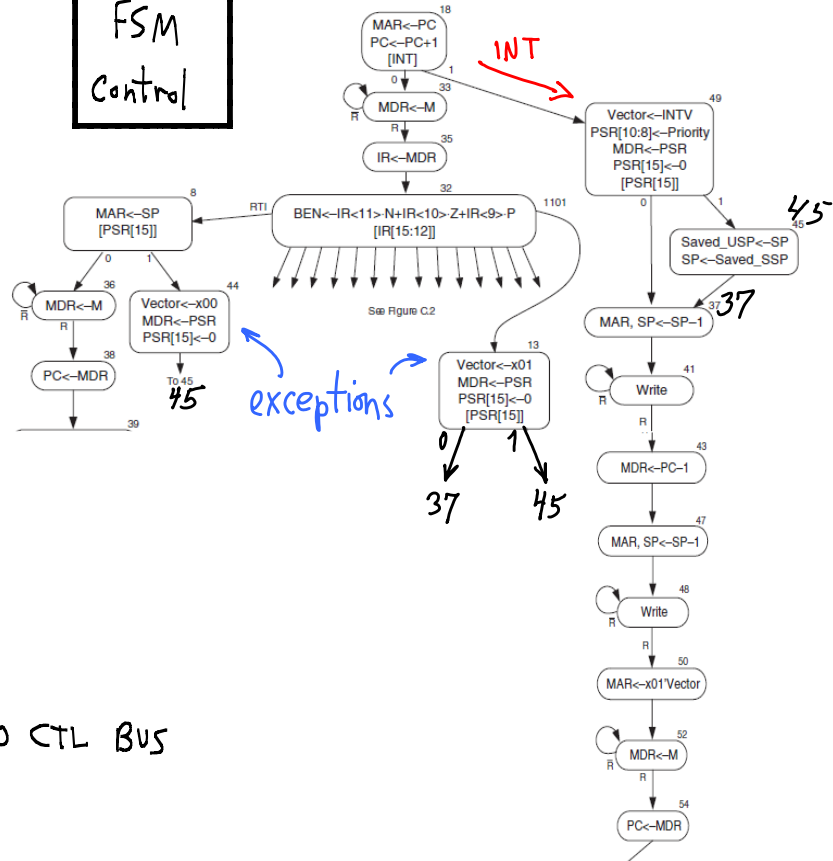
What about clobbering registers? Callee save? (R0 is clobbered anyway on pop, and SP should not be saved.)

Back To Interrupts

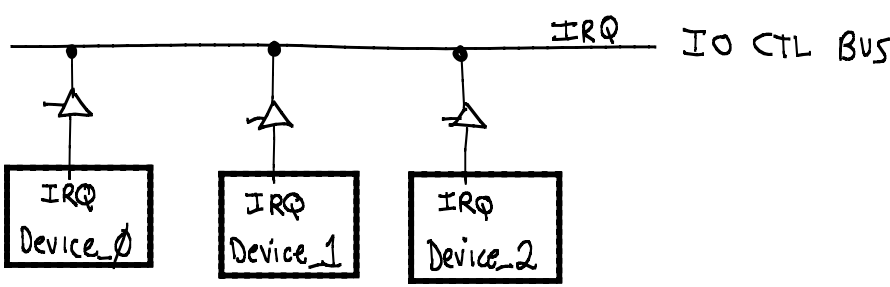
Enabling interrupts:
Set $KBSR[14] = 1$ allows controller to be interrupted.



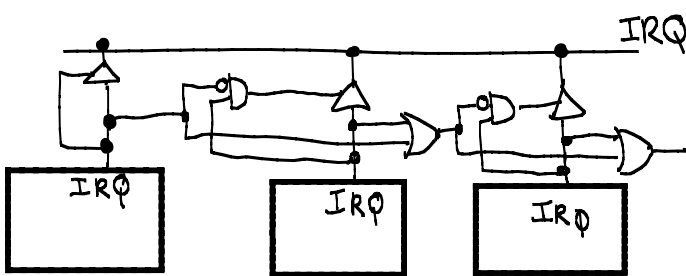
Programmer sets by writing into KBSR device sets when it puts new data into KBDR (resets after a read of KBDR)



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 →

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

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.

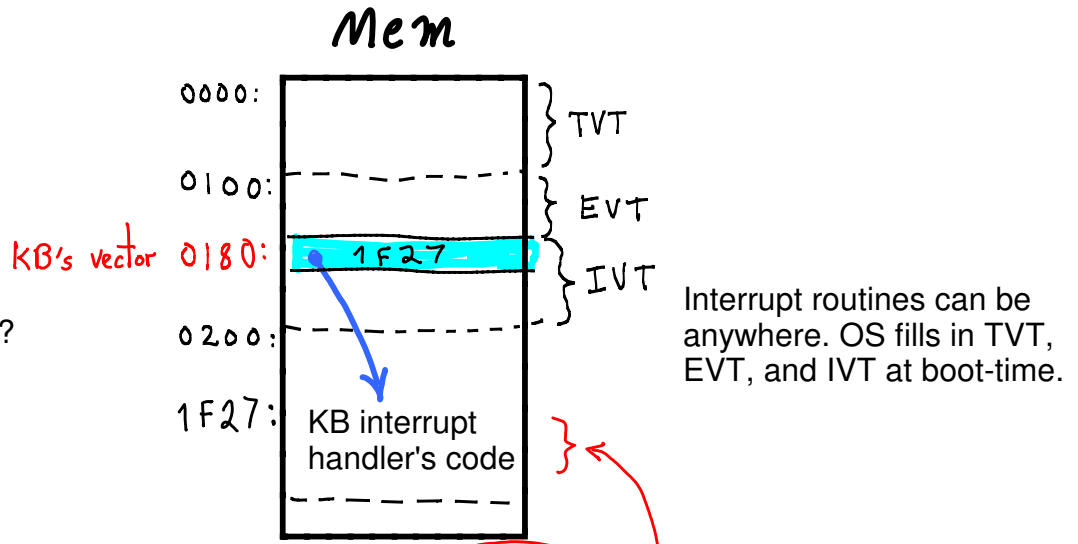
Processor interrupted, now what?
 How to jump, where to jump?
 Maybe like a trap?

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

Interrupt:
 PC <== Mem[?]

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

- Jump similarly to a Trap instruction:
 via IVT.
- Each device has its own IVT slot



KB Driver
 3 Parts:

— int. handler

— init VT

— data service

```

.....
;;;
;;;-- 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

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

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?

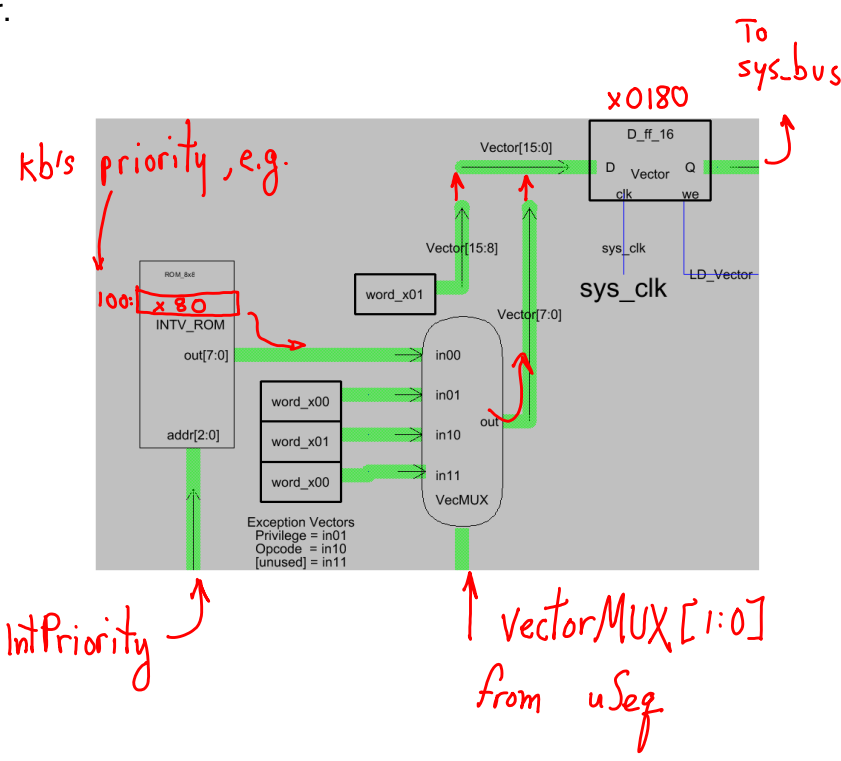
Also part of BusLogic is generating the address of the interrupting device's vector.

- 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

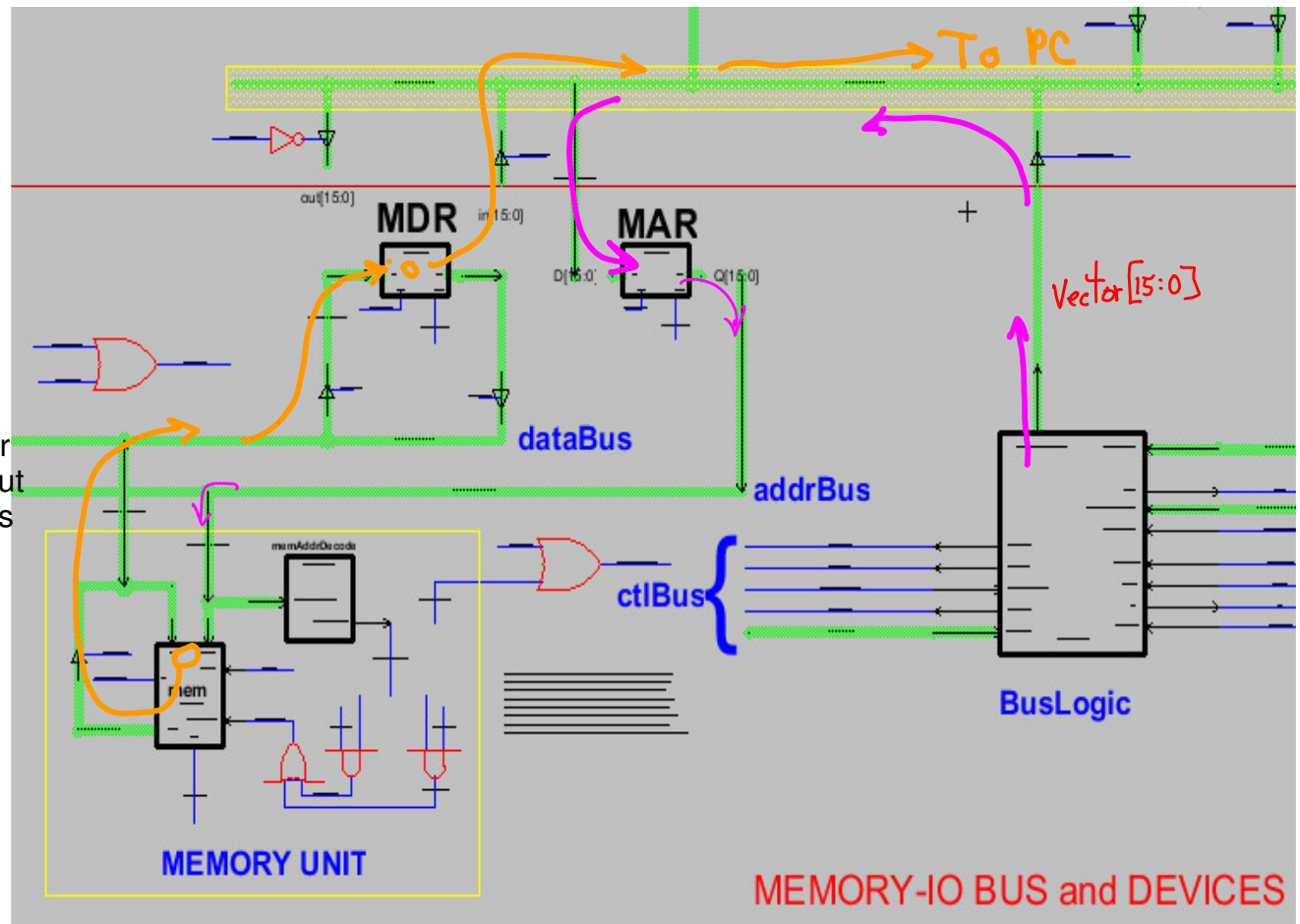
This is all just a lookup table, but addressing is split into two parts.



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?



MEMORY-IO BUS and DEVICES

How to save state of interrupted program?

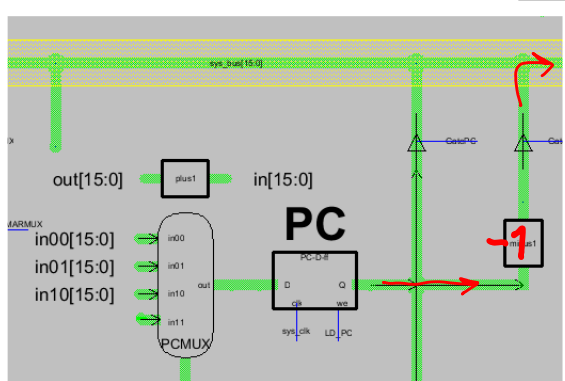
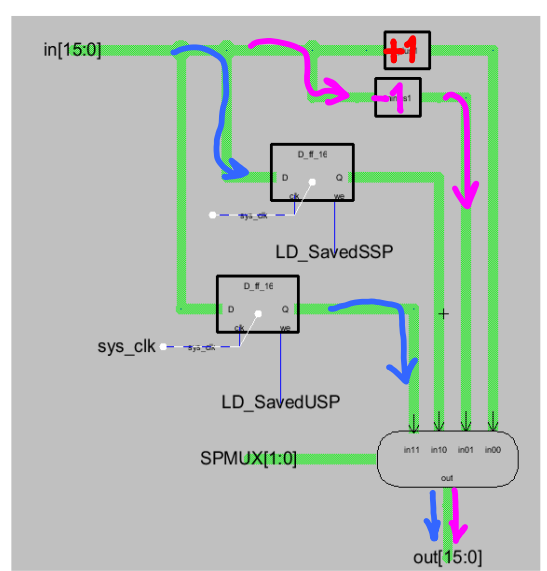
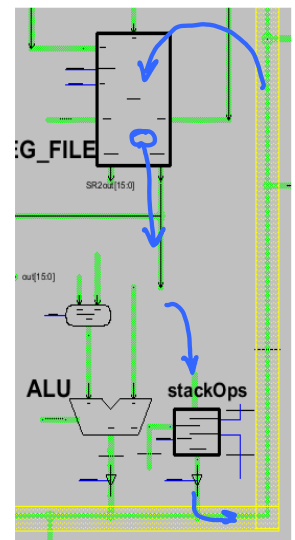
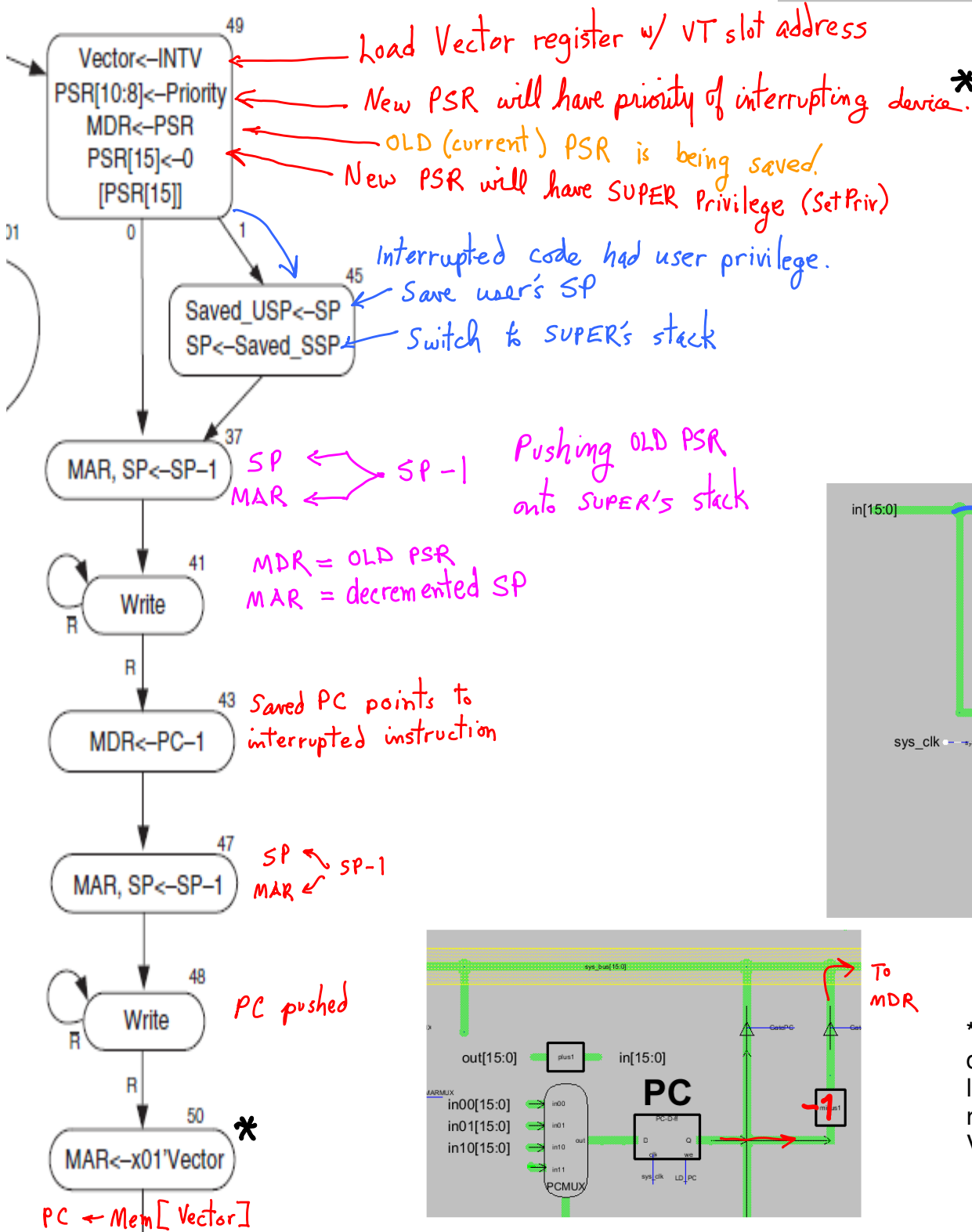
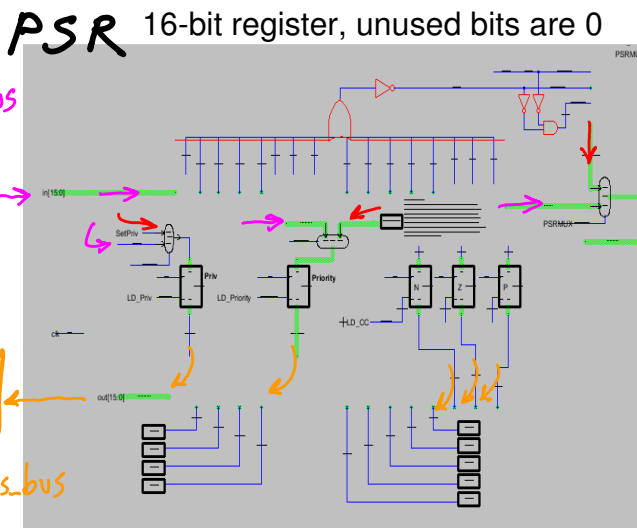
Push essentials to STACK (PSR, PC).

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)



TO MAR

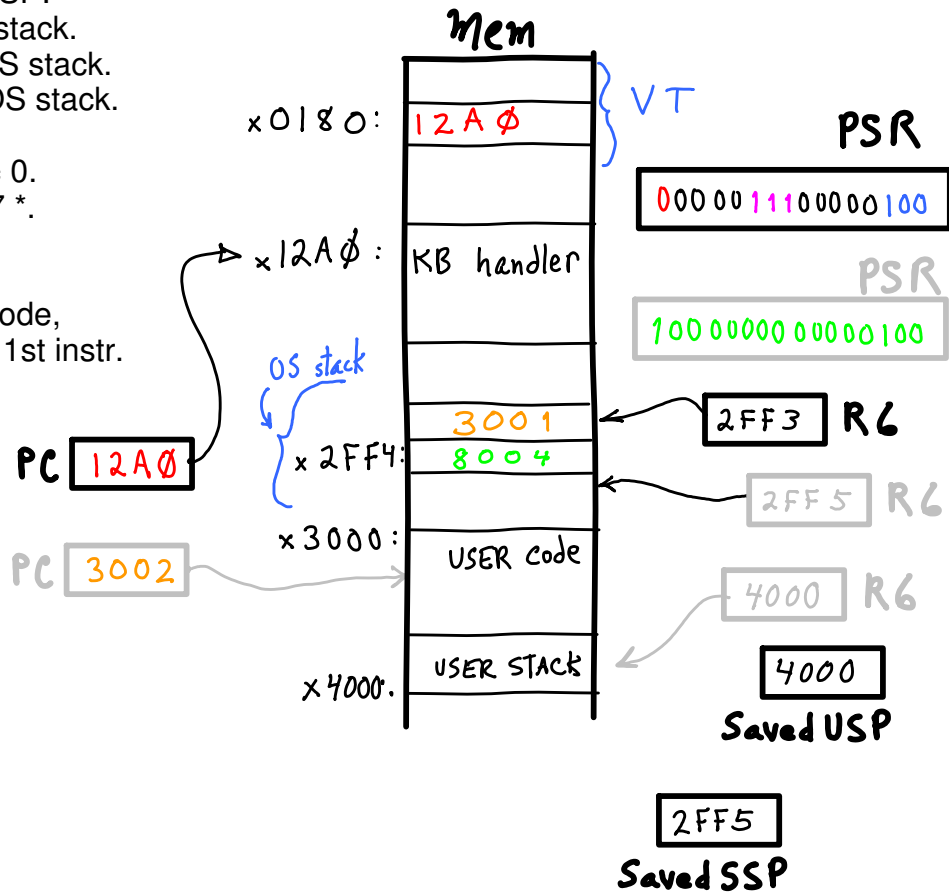
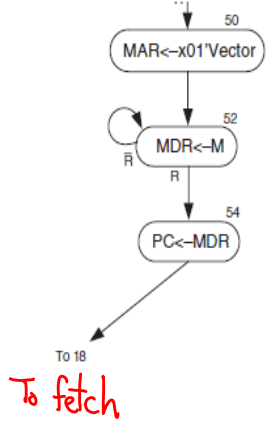
To SP

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

The overall effect

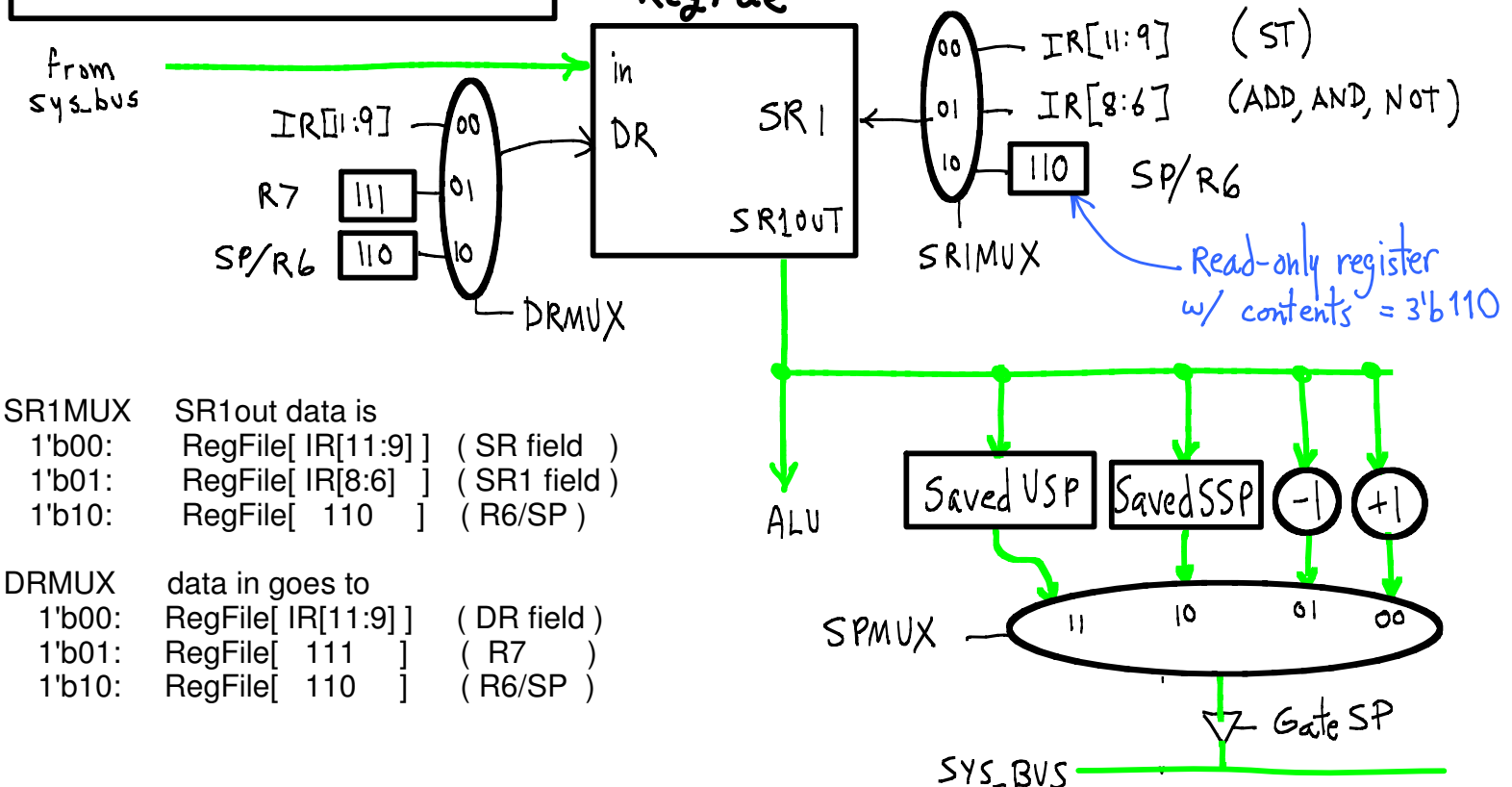
- User SP saved.
- R6 <== savedSSP:
SP set to OS stack.
- User PSR on OS stack.
- User PC-1 on OS stack.
- PSR.Privilege = 0.
- PSR.Priority = 7 *.
- SP <== SP - 2.
- PC at handler code,
ready to fetch 1st instr.

This is the jump to the handler:



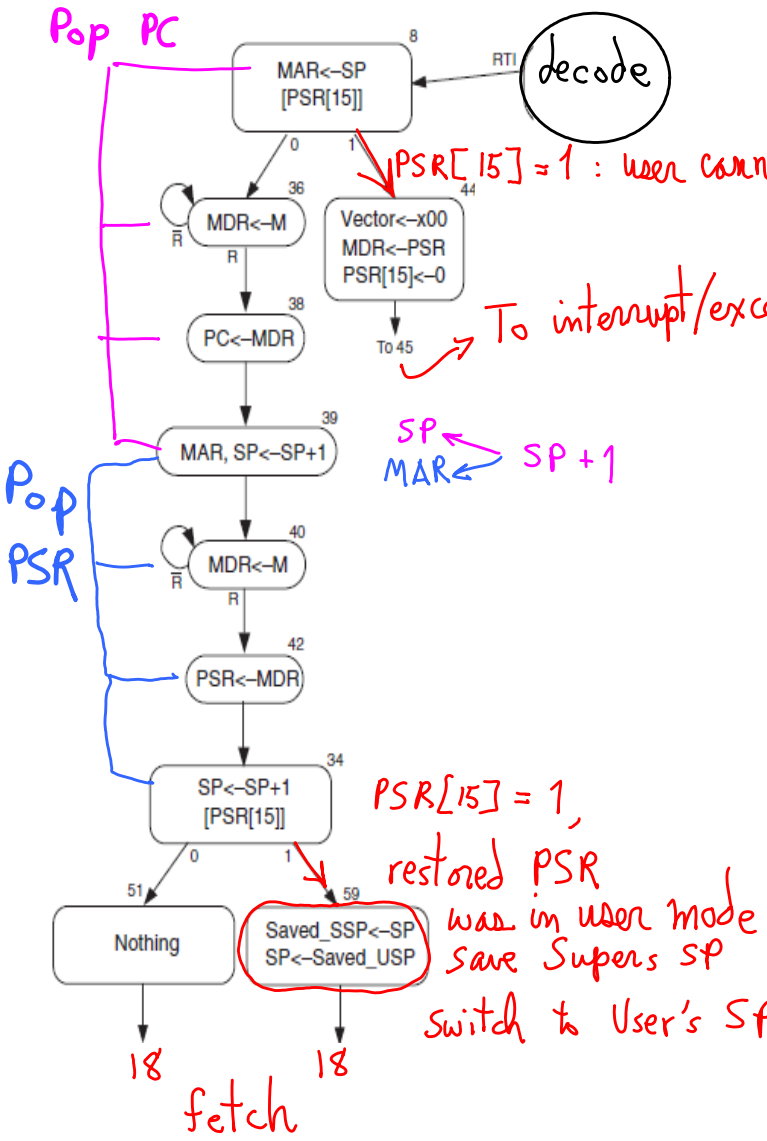
Suppose we had interrupted an interrupt routine. What happens next after this current interrupt routine finishes?

RegFile source and destination select



The RTI instruction: restore interrupted program

1000:00...000 IR



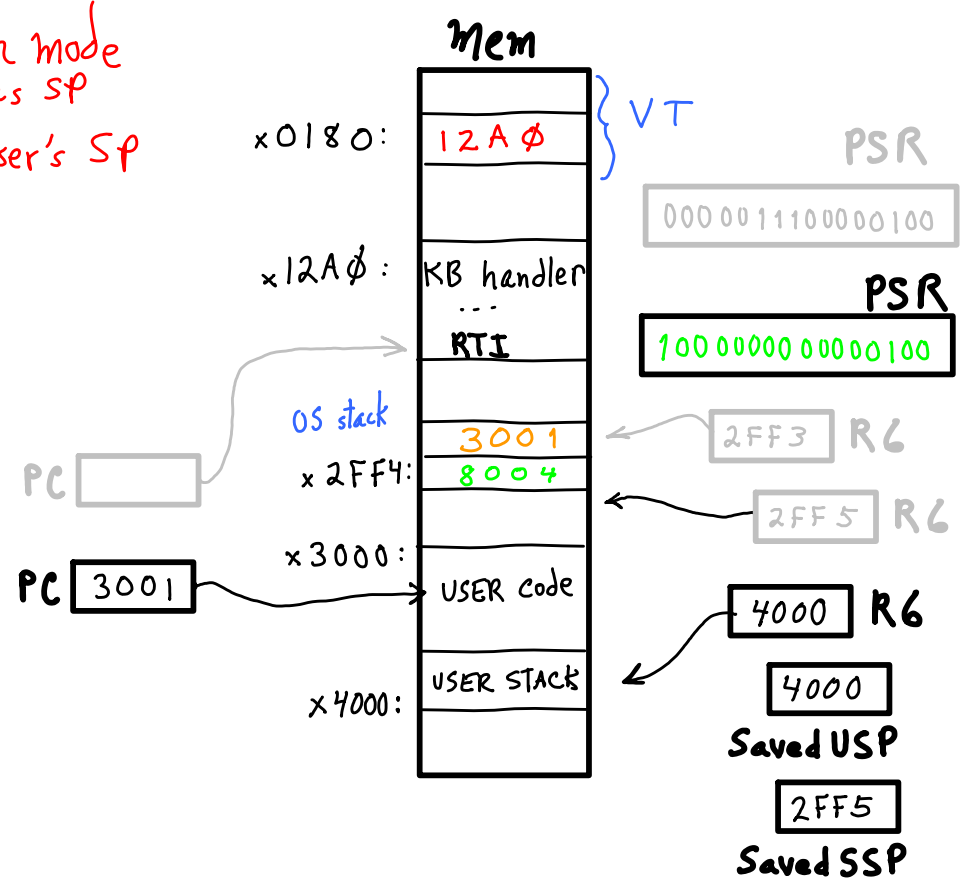
PSR[15] = 1 : user cannot alter Super's stack : Privilege Exception

To interrupt/exception states

SP ← SP + 1
MAR ← SP + 1

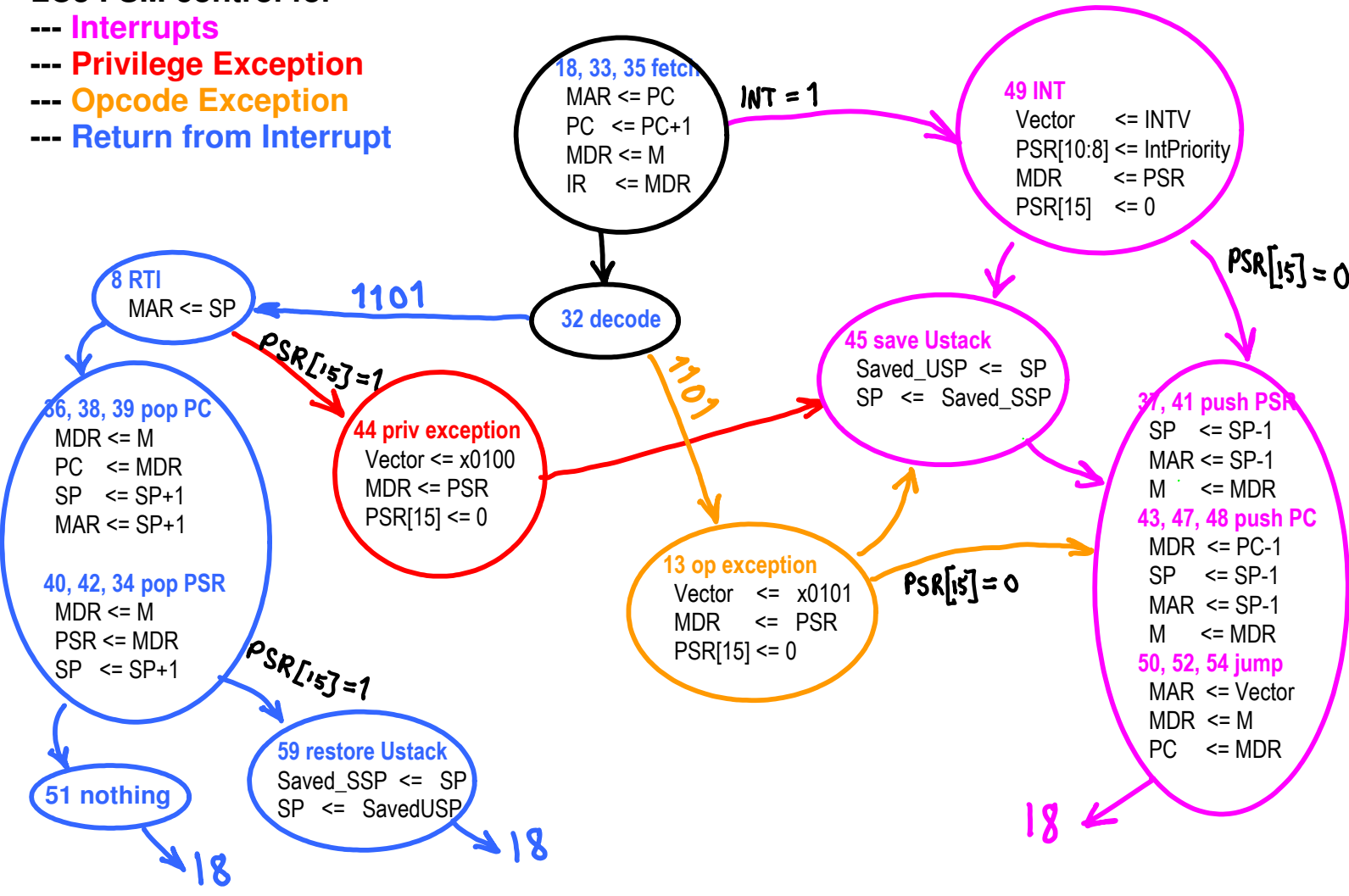
PSR[15] = 1, restored PSR was in user mode save Super's SP switch to User's SP

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?

