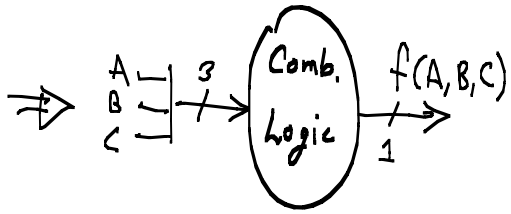
Implement FSM controller

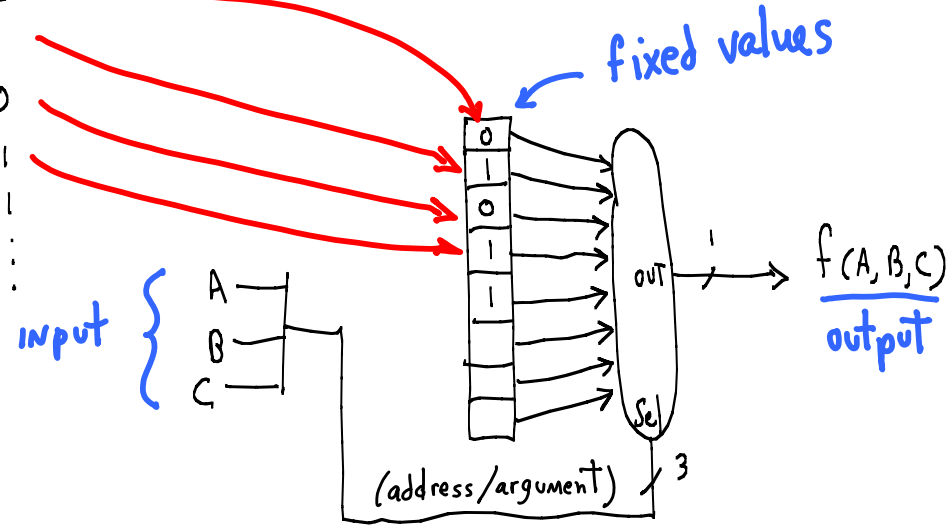
Implement uSeq's FSM's
--- Next-state function
--- Output function

We know:
Any function can be implemented in AND-OR

I) implement logic w/ ROM



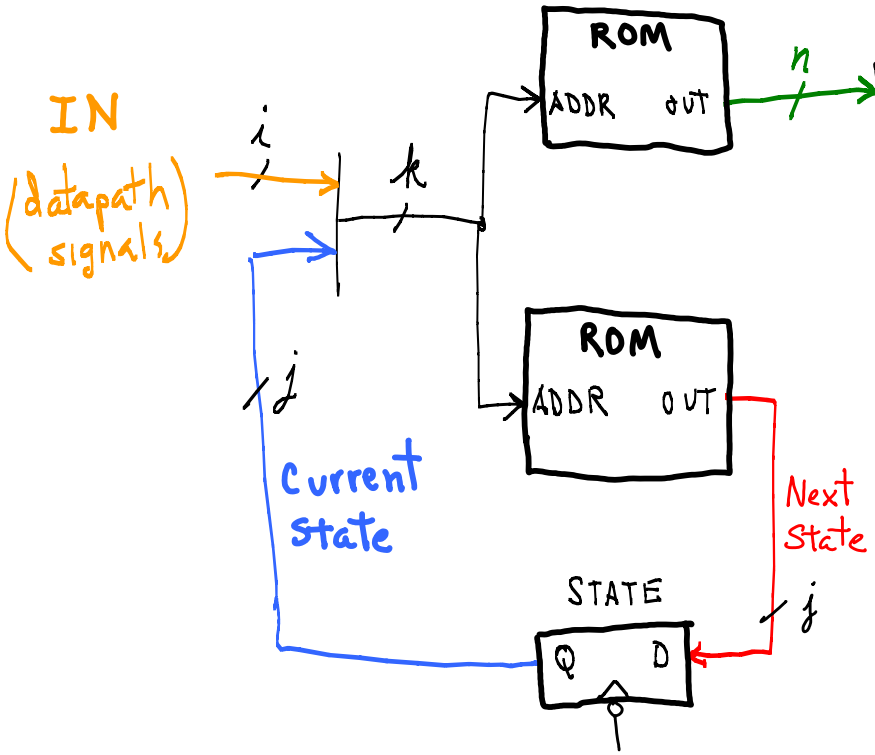
A	B	C	f(A,B,C)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
⋮	⋮	⋮	⋮



in general

k-bit inputs and n output functions (or n-bit word, or i+j bit words, or...)

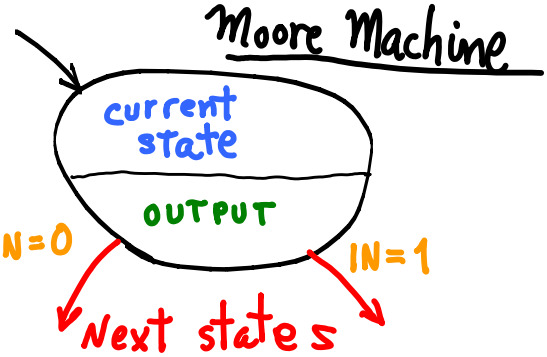
FSM functions in ROM (Moore Machine)



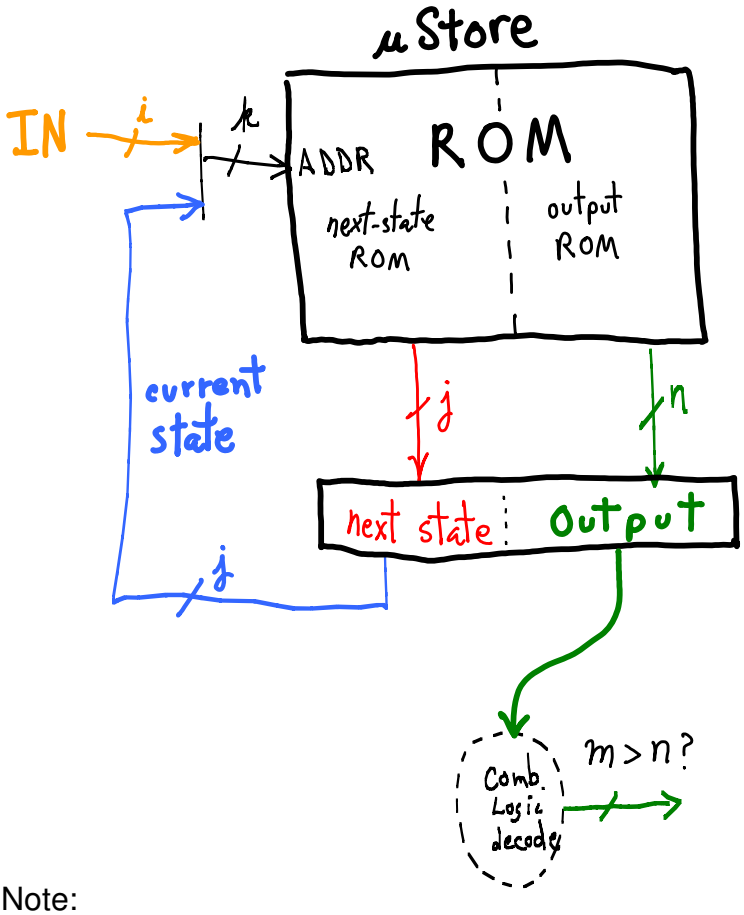
i --- # bits input to FSM (IR, PSR, ...)
j --- # bits of current state

k = i + j --- # function input/address bits

n + j --- # output bits, in total



Combine as 1 ROM



Note:
Could used control "codewords".
Decode to actual control signals.

OK, if not every control bit combo
is used. Saves columns of ROM.

Wasted Space?

IN + current-state == (i+j)-bit address
--- Many addresses ==> One LC3 state

LC3:
j = 6 (2^6 = 64 possible states, 59 used)
i = 9 (bits input: IR, PSR, ...)

ONE LC3 state ==> (2^9 = 512) ROM rows.
E.g., state-0, **BR**:

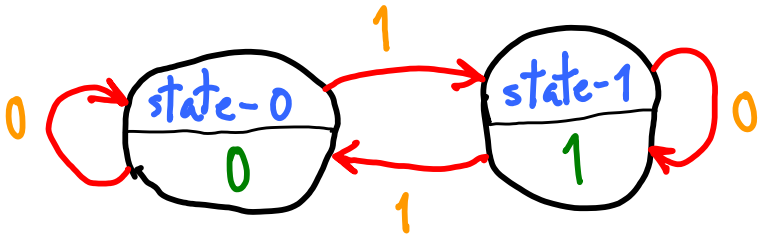
state	IN	ROM address
000000	000000000	x0000
000000	000000001	x0001
000000	000000010	x0002
...	...	...
000000	111111111	x01FF

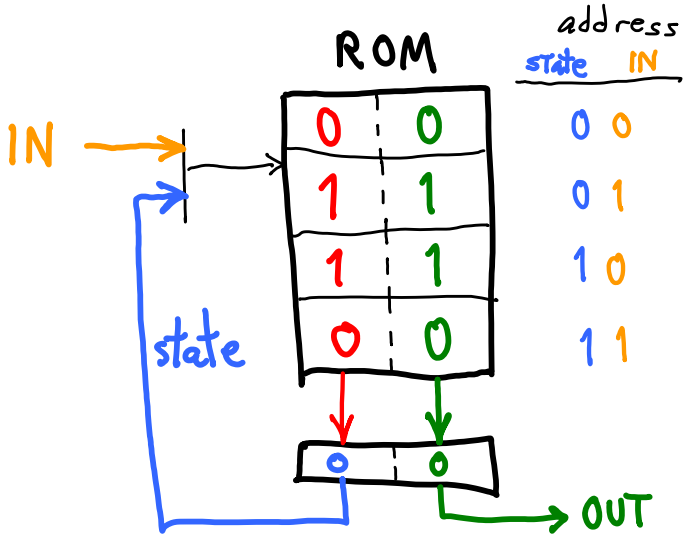
--- Uses 1-bit for next state (BEN).
--- All other inputs ignored.

Next-state is either 18 or 22.
No non-zero outputs.

--- 256 ROM rows contain
010010 00000000...000 (6+40 bits)
--- 256 ROM rows contain
010100 00000000...000 (6+40 bits)

FSM in ROM, e.g.

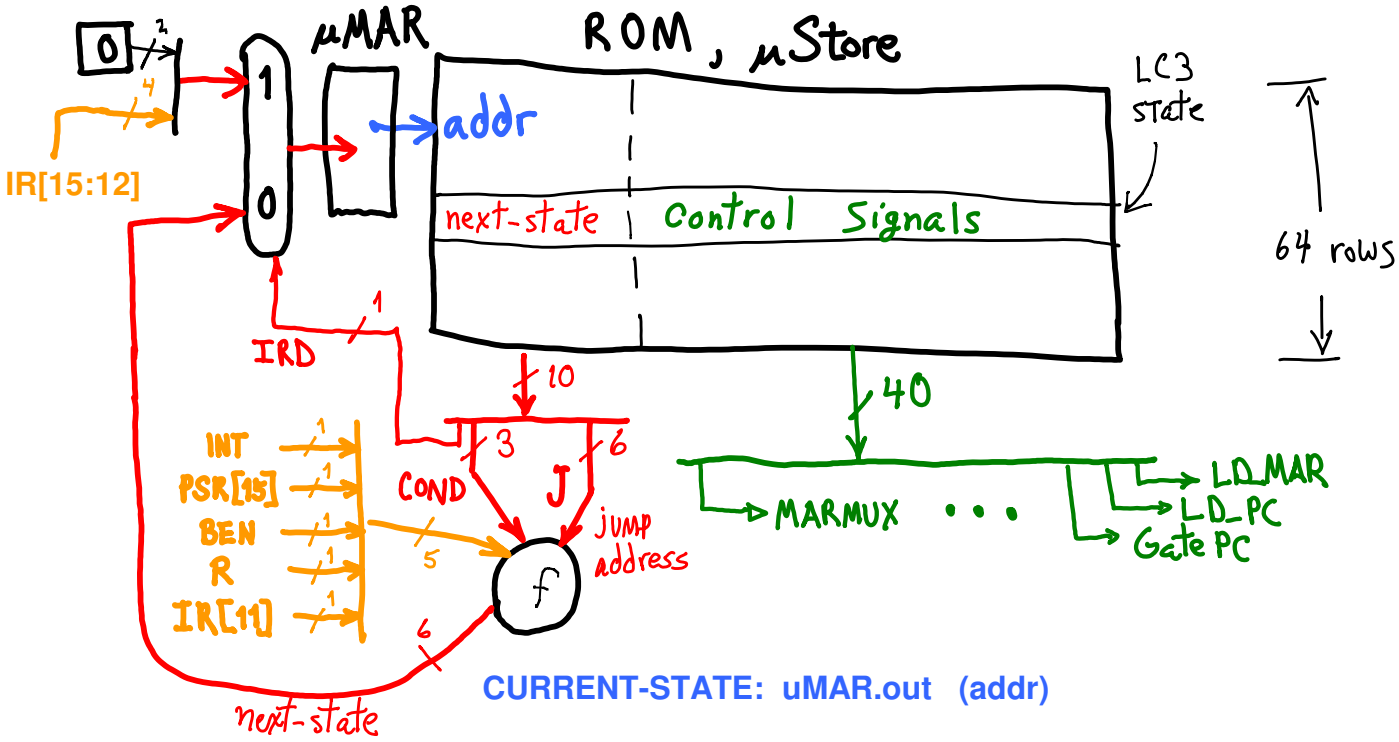




Aside: If the next state is 0, then next output is 0 (rows 00 and 11). What if that wasn't so? Would this be a Mealy Machine?

LC3 micro-Sequencer, μSeq

59 states. We need 6 bits to designate state. 2^6 = 64.



CURRENT-STATE: uMAR.out (addr)

OUTPUT: All control signals, selected by **addr.**

NEXT-STATE: Not so simple! Determined by:

--- **IR.opcode**, if **IRD** == 1

uMAR <== { **00**, **IR[15:12]** } (e.g., **00**1001 for **NOT** instruction)
(**IRD** is only set for state-32)

--- Otherwise
COND bits are mixed with **INT, PSR[15], BEN, R, IR[11]**
JUMP address is modified accordingly.

Note:

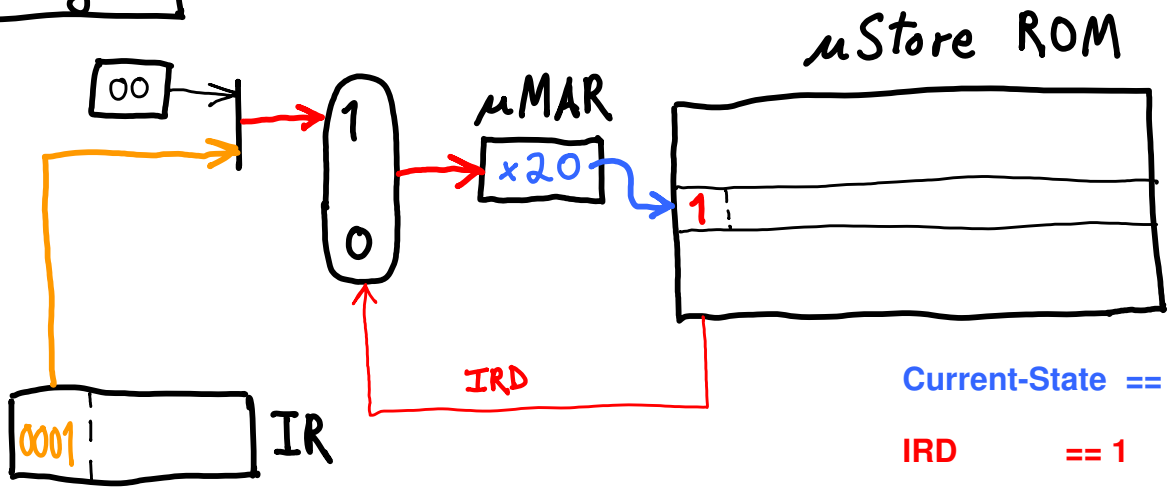
An instruction's 1st state in its execution control path is,

00xxxx,

where xxxx = IR[15:12] is its opcode.

No other states have 00 as a prefix.

e.g.

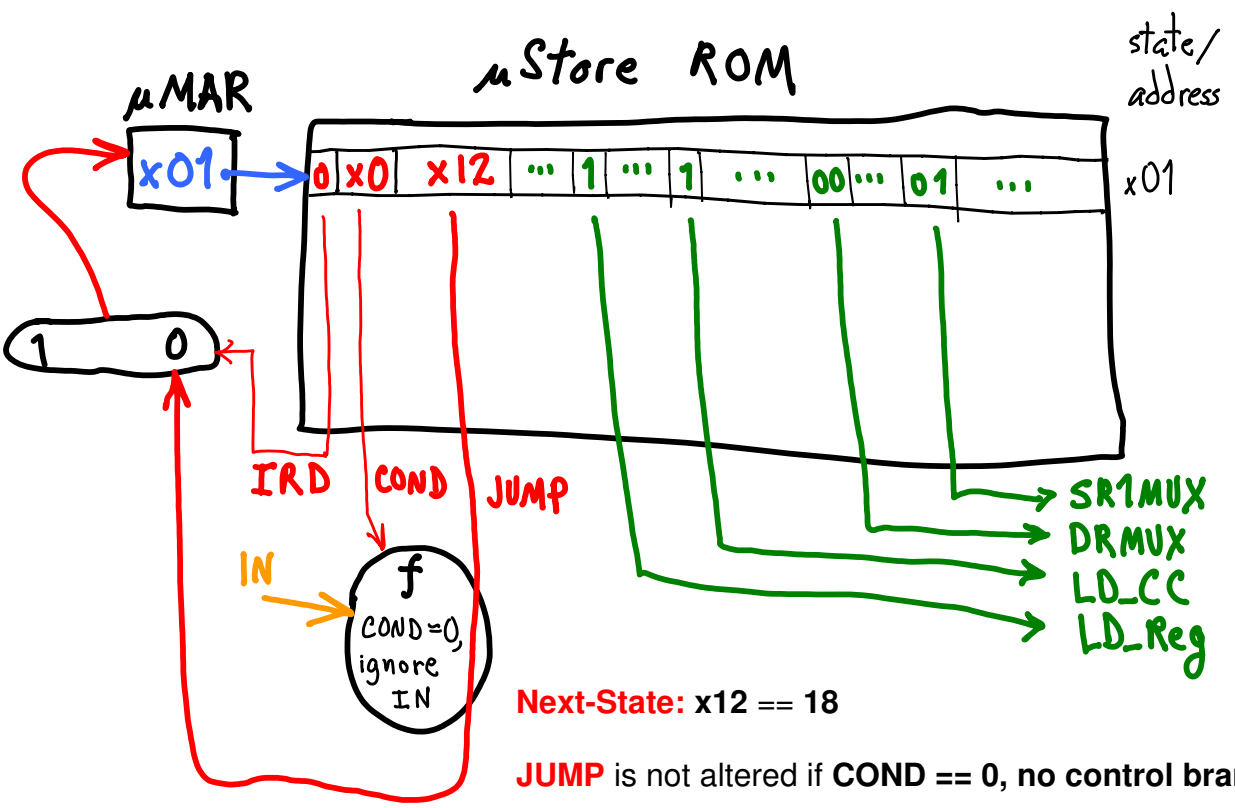


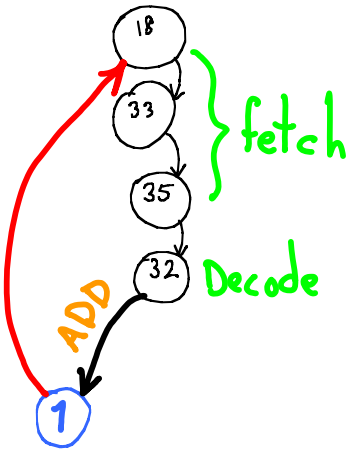
Current-State == x20 == 32 (DECODE)

IRD == 1

IR[15:12] == 0001 (ADD)

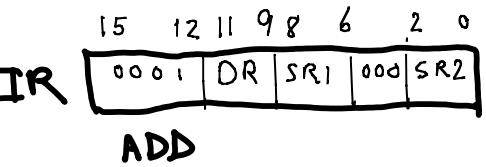
Next-State == 000001 (state 1)

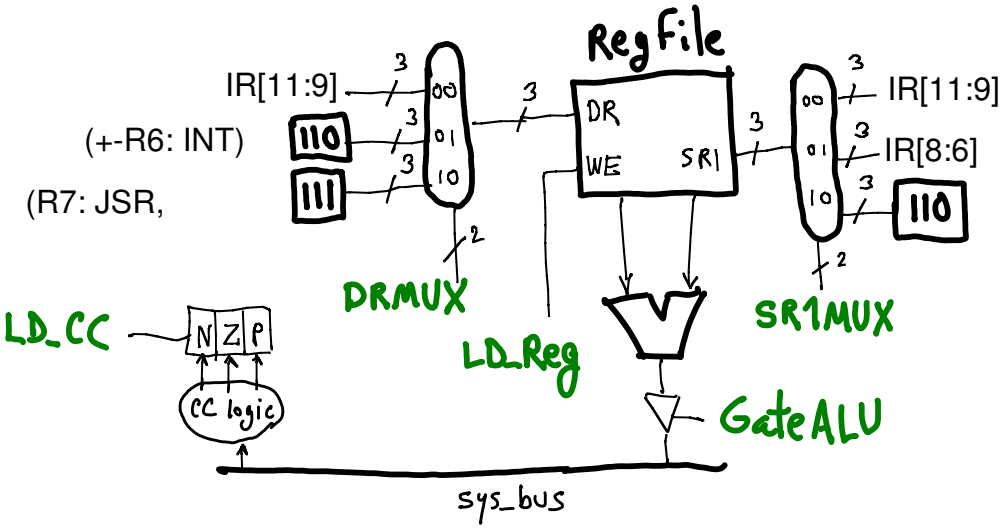




Next-State: x12 == 18

JUMP is not altered if **COND == 0, no control branching.**





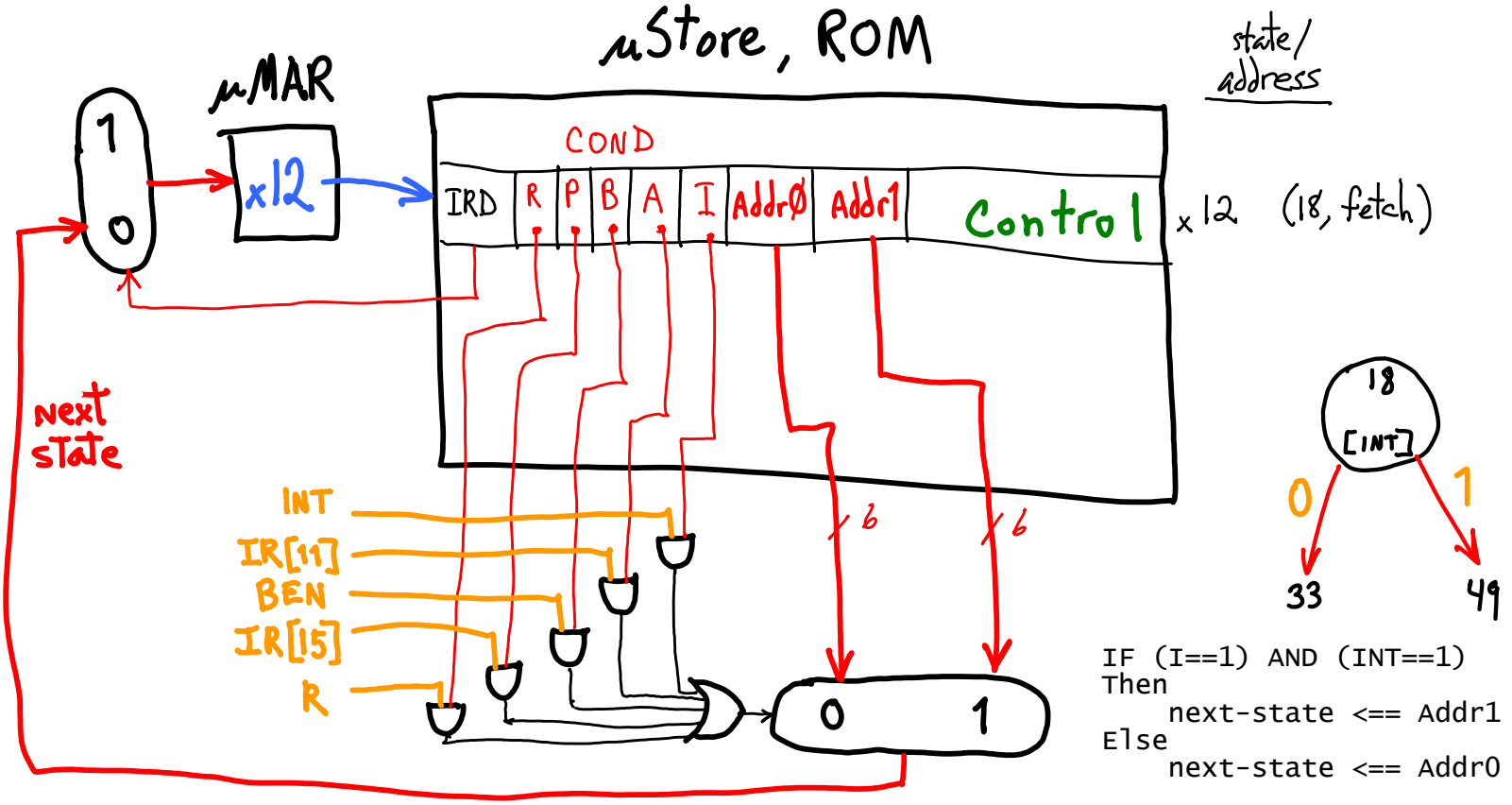
Control Branching in LC3

But first, an easier method.

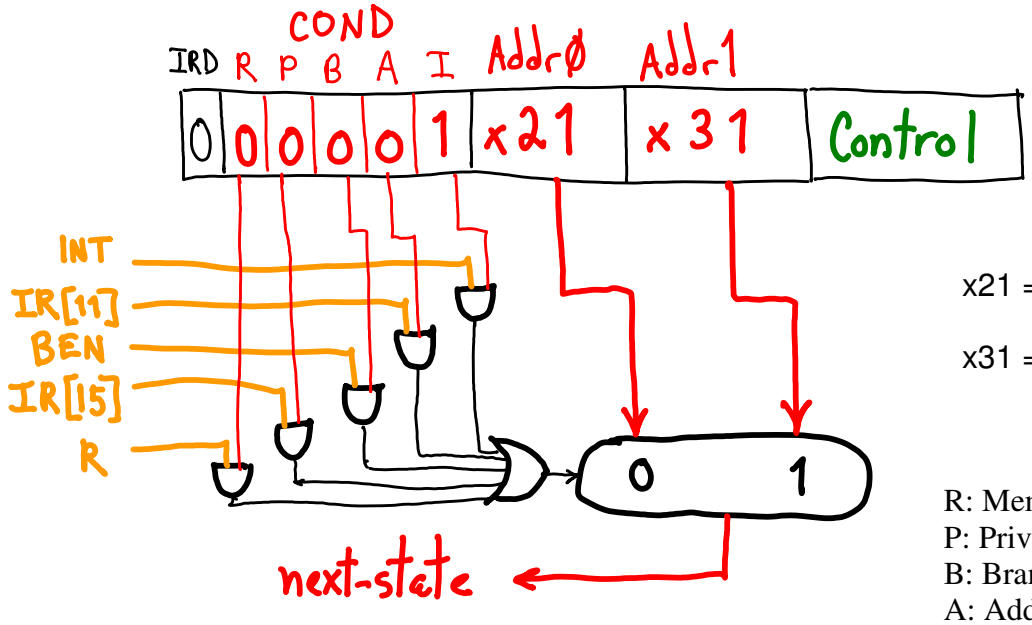
Let's have 2-way branching (except state-32).

Fields for both targets.

1-bit of input determines next state.




```
IF (I==1) AND (INT==1)
Then
    next-state <== Addr1
Else
    next-state <== Addr0
```

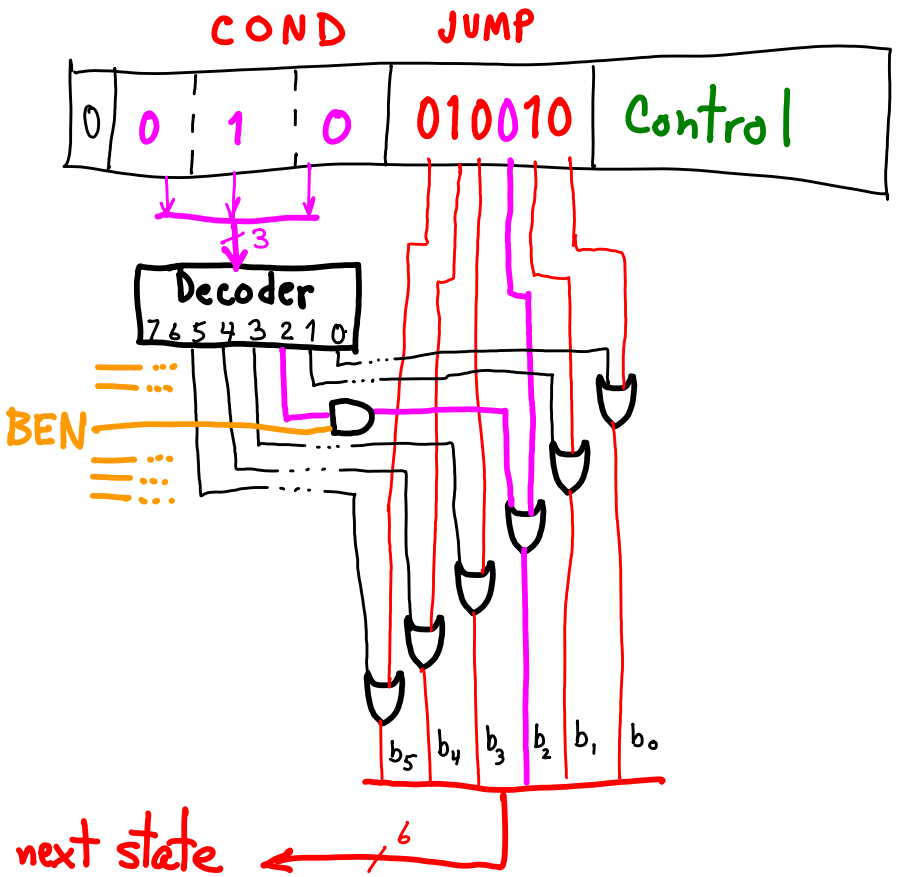


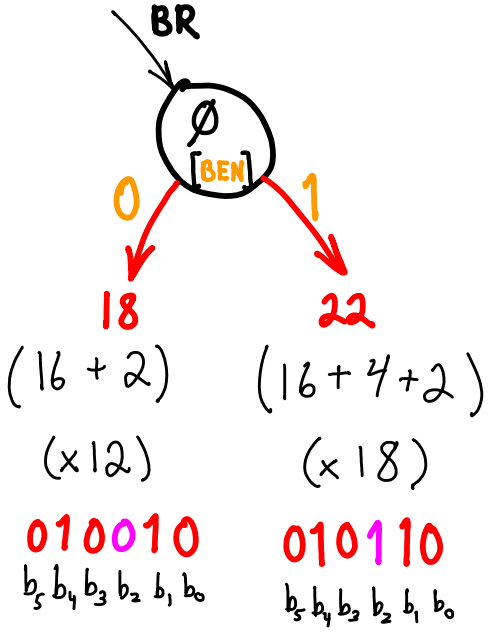
x21 == 100001 == 33

x31 == 110001 == 49

R: Mem-IO Ready
P: Privilege bit
B: Branch
A: Addressing mode (JSR, JSSR)
I: Interrupt

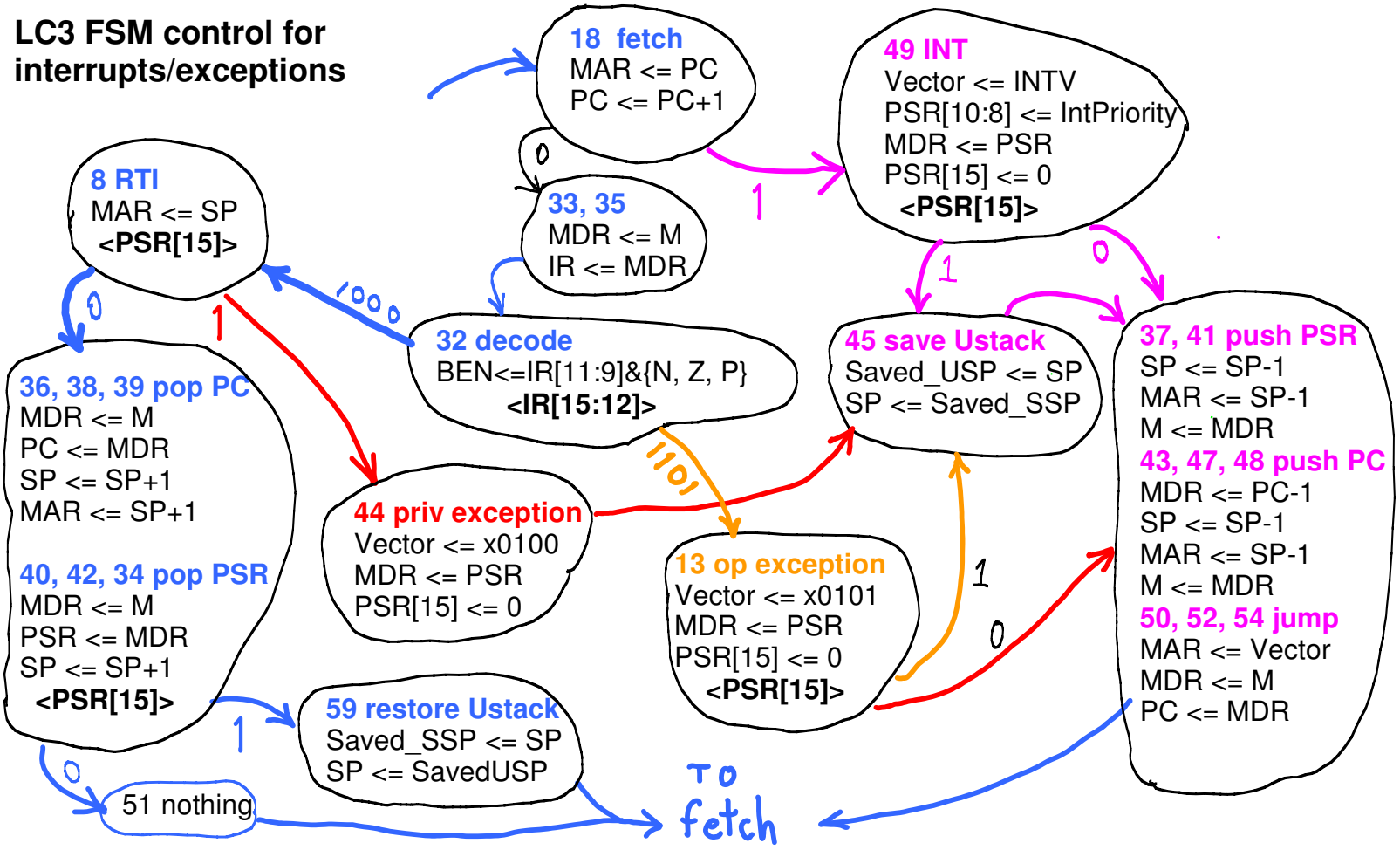
Actually, in P&P





COND code i
alters JUMP[i]

LC3 FSM control for interrupts/exceptions



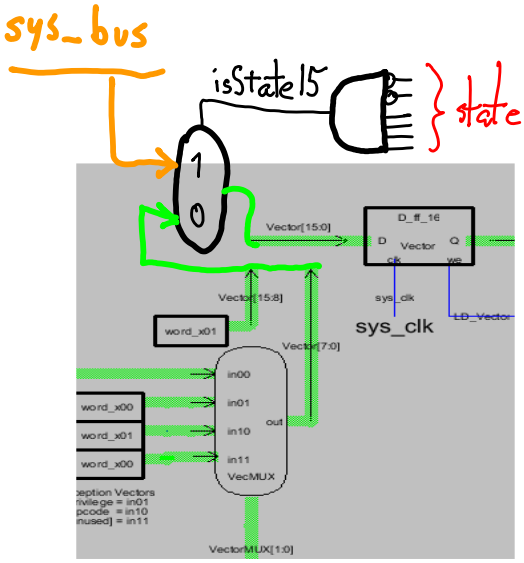
Hardware Additions (privilege + memory protection).
(low-cost additions)

1. TRAP switches to supervisor mode.
2. Return switches back to user mode.

Have TRAP act like exception? Use RTI for return?

1.A TRAP's state-15 acts like state-13 (opcode exception), except for **jump**:

Vector <== sys_bus <== MARMUX <== ZEXT <== { x00, IR[7:0] }
MDR <= PSR
PSR[15] <= 0



1.B Branch on PSR[15]: same as state-13's:
to 37 or 45 (into interrupt chain). [Unused states: 15, 28, 30.]

1.C mux Vector's input:

select = 0: the usual input
select = 1: **sys_bus**

Control logic?
Don't redesign uStore to have more columns?
a. Add uSeq output "state" (controller's current state).
b. **select == AND(x0F, state)** [6-bit AND: select == 1 if state-15]
c. caveat: a problem for pipelining?

2. RTI returns to calling code (restores privileges).

MORE ADDITIONS (low cost?)

Disable INTERRUPTS?

--- A. handler needs time to set up its state before allowing a new interrupt.
--- B. handler needs to disable multiple interrupts on same device.

We have:
--- Interrupt: PSR.Priority <== 2b'111
--- Priority Comparator: (IntPriority > PSR.Priority)
--- Effect: All interrupts are disabled initially on interrupt.

We can:
--- mask interrupts: device-status-register[14] <== 1b'0

```
    LDI   R1, KBSR
    LD    R2, BIT_14_MASK
    AND R1, R1, R2              ;--- There is also an "OR" macro, see src/lc3pre.
    STI   R1, KBSR
        ...
    KBSR:           .FILL xFE00
    BIT_14_MASK:    .FILL xBFFF    ;---  ( 1011 1111 1111 1111 )
```

Oh, ok, we've already handled this.

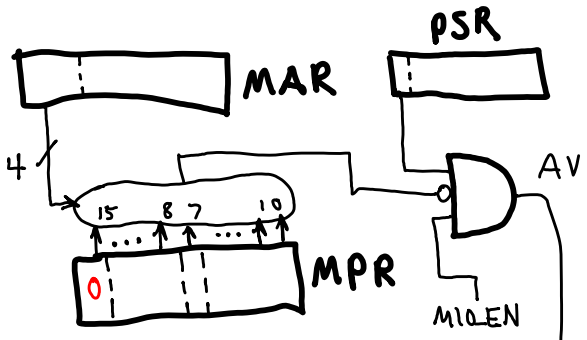
Memory Protection?

--- Privileged status is nice.
--- Useless unless memory is protected.

Need:
--- Hardware to detect address + protection bits
--- Exception for access violation

Add:
--- MPR, Memory Protection Register?
16-bit addresses, 16-bits in MPR,
--- divide into 16 "pages", one bit in MPR per page: (2^16)/(2^4) == (2^12) == 4k words per page.
--- 0: super only, 1: user or super.
--- Large-end 4 bits of address determines "page number"

```
0000: OS space      (x0000 - x0FFF)
0001: OS space      (x1000 - x1FFF)
  ...
0011: User space    (x3000 - x3FFF)
  ...
1110: User space    (xE000 - xEFFF)
1111: OS space      (xF000 - xFFFF)
```



R/W protection: More than one bit per page?
Multiple users: Owner bits?

Mapping Addresses?
--- virtual MAR
--- physical MAR

==> runtime independence