Addressing Modes

PC-relative

MAR ← PC + IR[8..0]

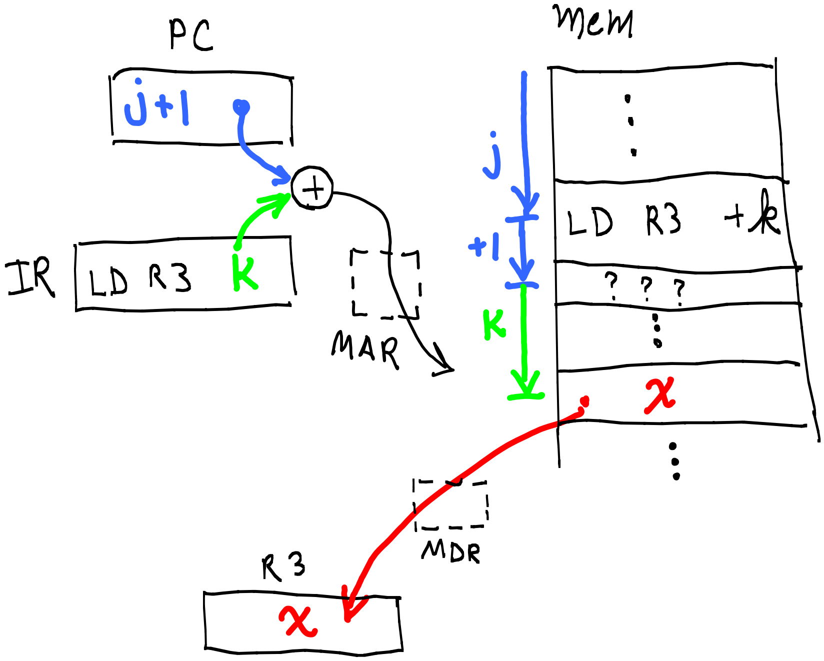
Regfile[IR[11..9]] ← MDR

0010 LD
0011 ST

op	DR	off9
11..9	8	... 0

IR

DR ← Mem[PC+off9]



PC-Indirect

MAR ← PC + IR[8..0]

MAR ← MDR

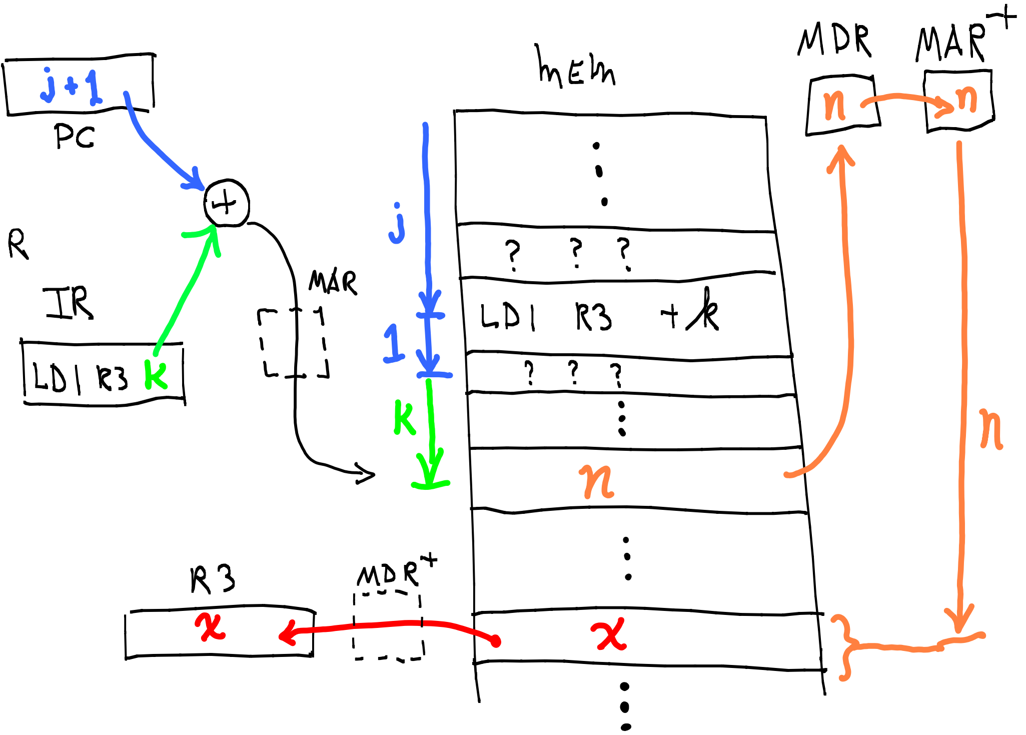
Regfile [IR[11..9]] ← MDR

1010 LDI
1011 STI

op	DR	off-9

IR

DR ← MEM[MEM[PC+off9]]

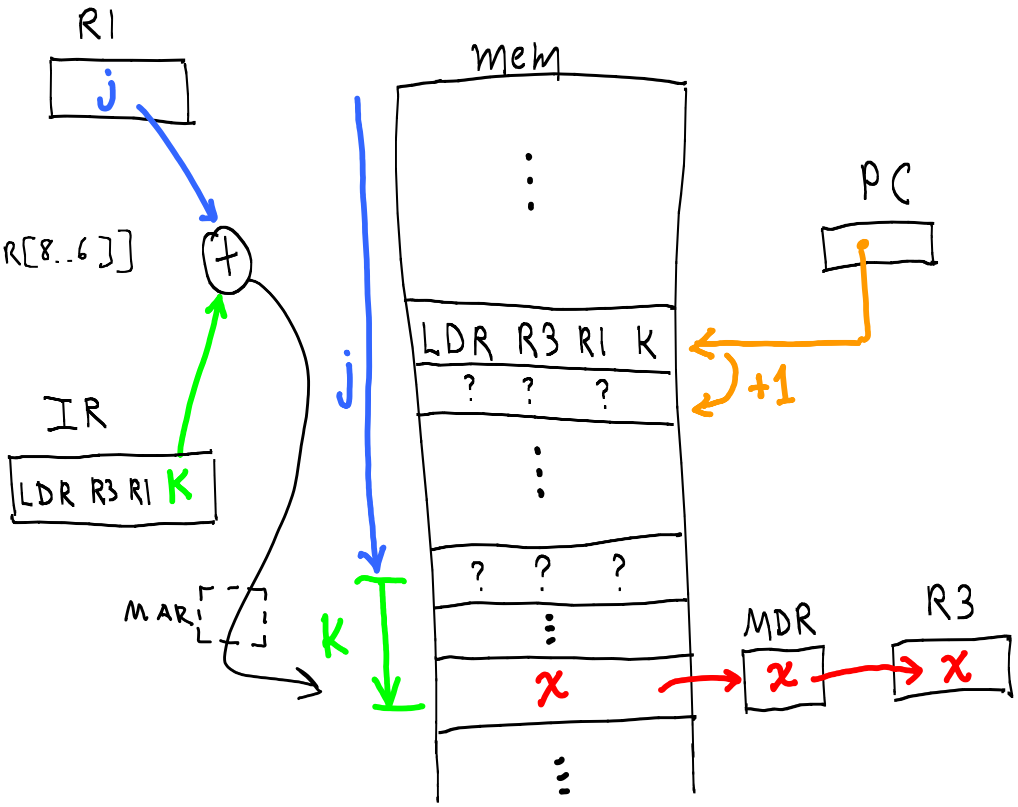


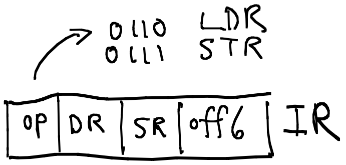
Base-offset

(Reg-relative)
(Reg-indirect)

MAR ← IR[5..0] + Regfile[IR[8..6]]

Regfile[IR[11..9]] ← MDR





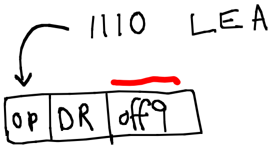
DR ← MEM[SR+off6]

IMMEDIATE

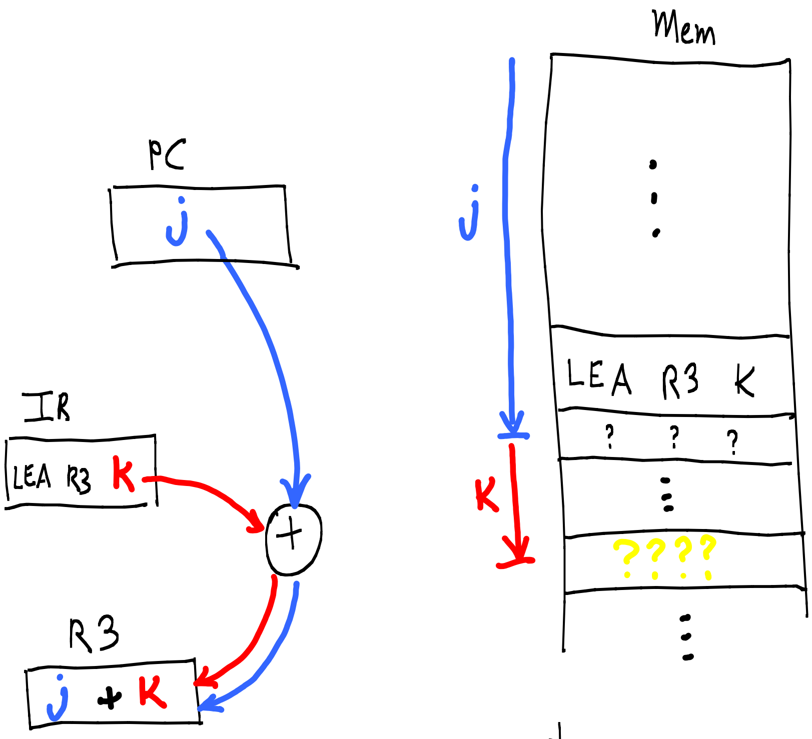
(Reg-immediate)
(PC-Immediate)

immediate data

Regfile[IR[11..9]] ← PC + IR[8..0]



DR ← PC + off9

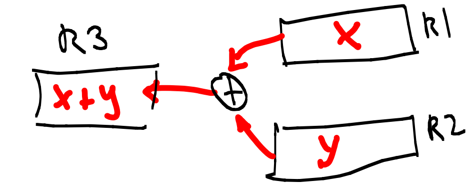


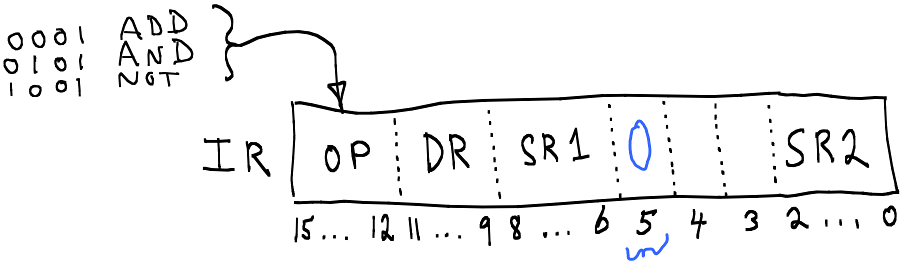
use later as pointer to data?

LDR R1, R3, 0

Reg-Reg

Regfile[IR[11..9]] ← Regfile[IR[8..6]] **OP** Regfile[IR[2..0]]



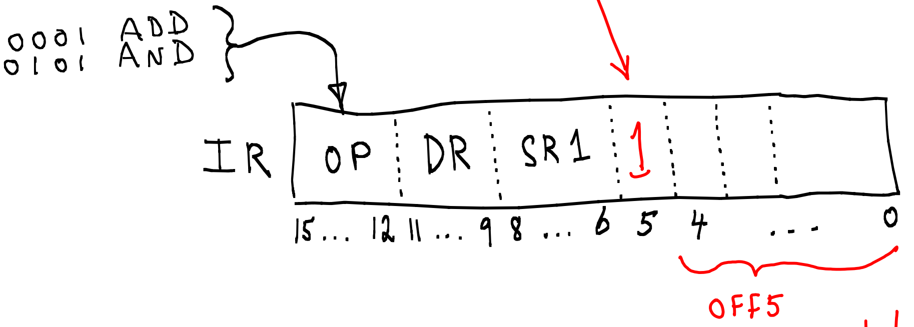


Select for input A to ALU

DR ← SR1 **OP** SR2

DR ← **NOT** SR1

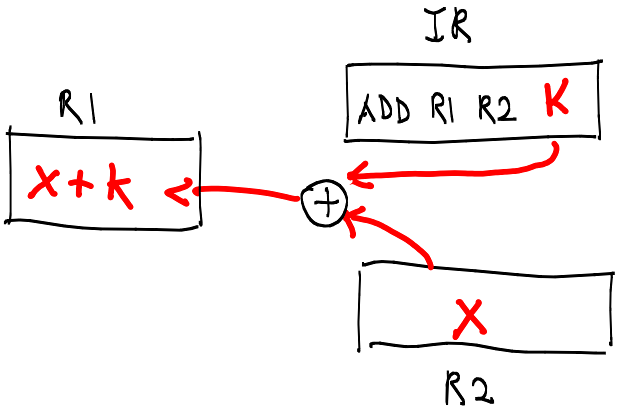
Reg-immediate

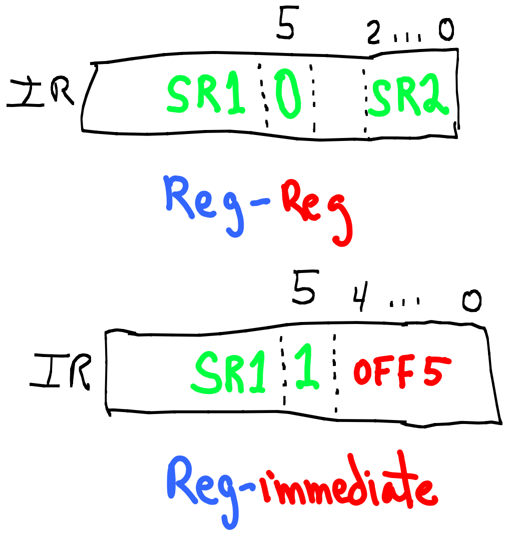


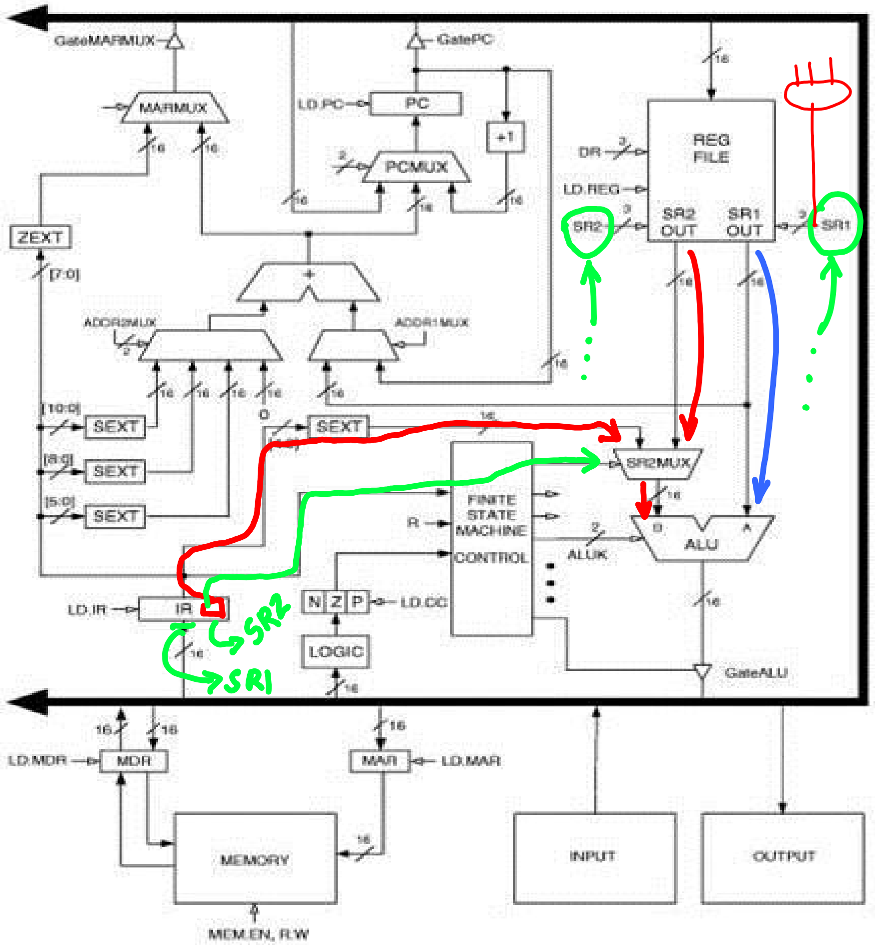
DR←SR **OP** off5

Reg file[IR[11..9]] ← Refile[IR[8..6]] **OP** IR[4..0]

immediate data sign-extended to 16 bits







Only for opcodes
ADD -- 0001
AND -- 0101

IR[5] ==> SR2MUX

Does it matter for other opcodes? (GateALU = 1?)

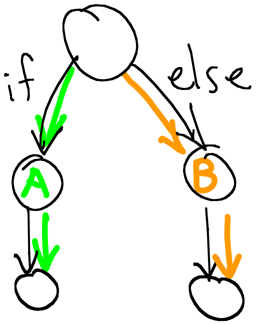
When does SR2 matter?

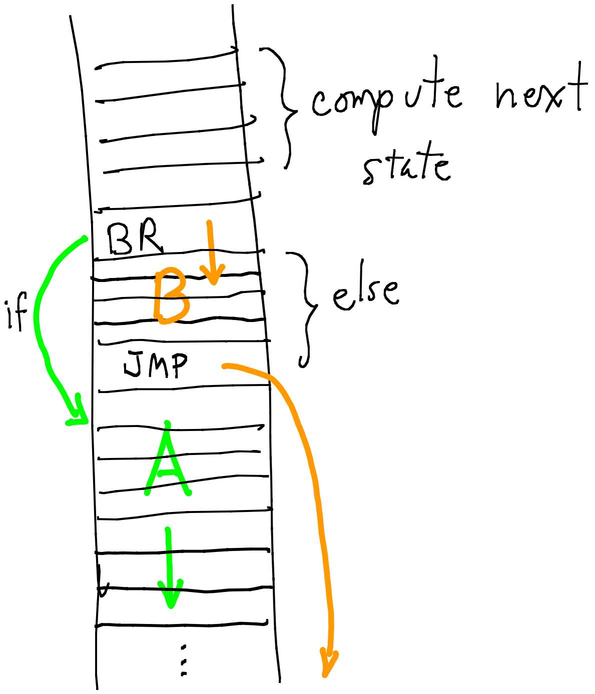
NEED language for TMs.
HAVE functions: NAND (AND, NOT) is universal (+ bonus, ADD)
HAVE tape: LD, ST (and variants)

BUT NO branching in FSM.

(next state function
Compute a,b)

```
if (a > b) {
    /* if part */
} else {
    /* else part */
}
```





need BR, JMP

Address are formed from,

1. some register content (PC or register in RegFile)
2. part of IR register's bits (some portion of the instruction)
3. content of memory location

Addresses are used to,

1. access memory (load address into MAR)
2. change location of next instruction to be fetched (load address into PC)
3. save for later use (load address to register in RegFile)
4. save for later use (load address to memory location)

Branching (reloading PC based on some condition)

1. LC3's controller branches from its decode state 2^4 ways (16 ways)
2. minimal branching is two-way (if-then)
3. nested if-then can form arbitrary k-way branches
4. same address forming mechanisms used for branches
5. condition is based on symbol seen (Turing Machines)
6. compare a symbol with another (is-equal == difference is zero)
7. LC3 stores result of comparison in PSR Condition Codes on every register write
8. simple logic for is-zero (Z), is-positive (P), is-negative (N)

Function Calls

1. abstraction == interface and hidden details, multiple levels
2. jumping to a "lower-level" abstraction == access a sub-cell in Electric hierarchy
3. code == hardware
4. jumping to function code, jumping back to next instruction after function call

(1) REMEMBER RESULT of FUNCTION EVALUTION

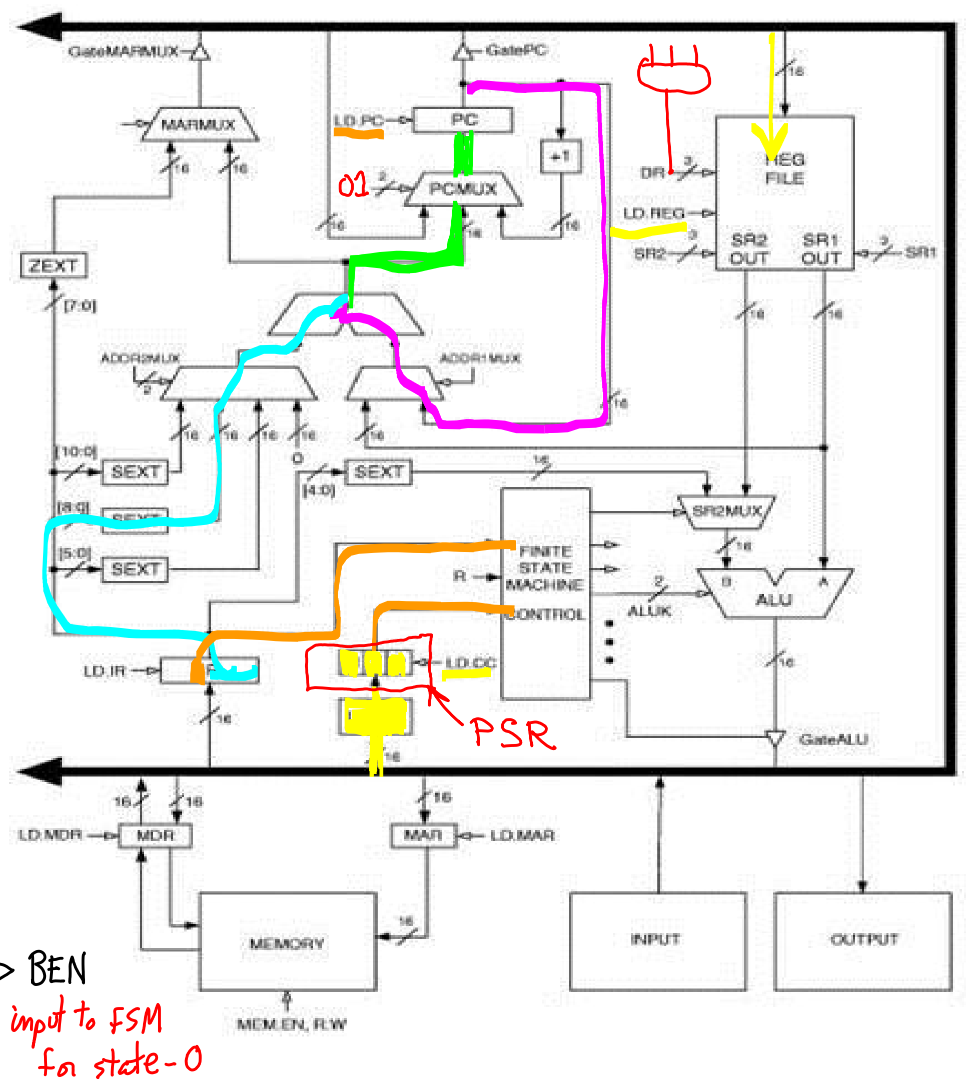
LD_CC == LD_REG

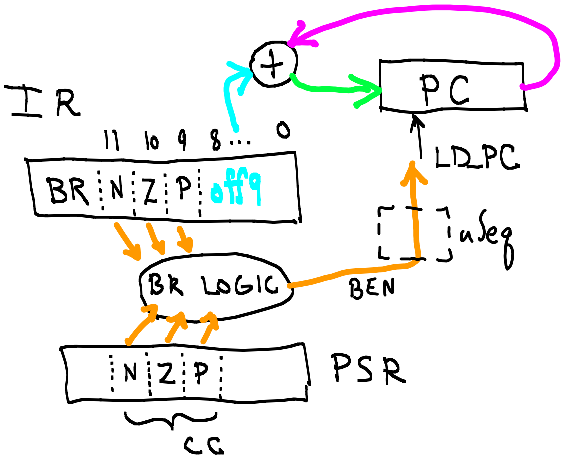
on ANY register load (AND, ADD, NOT, LD, ...)

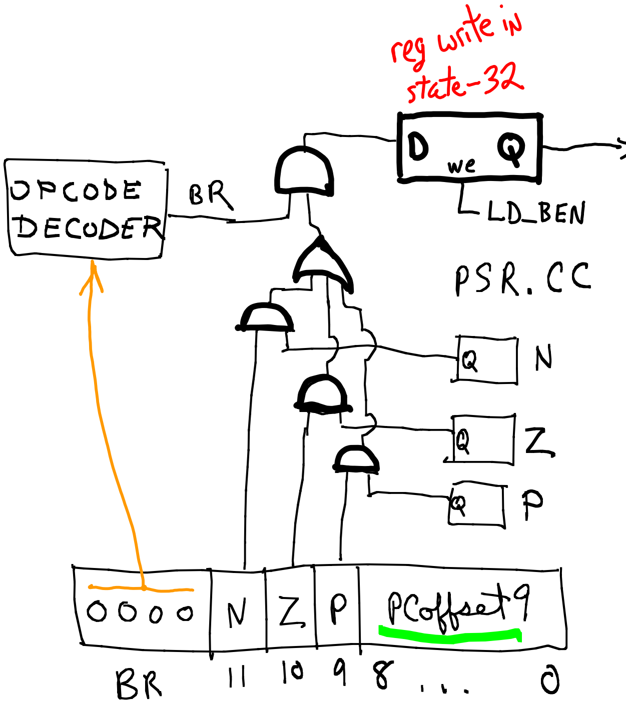
N = BUS[15] <was it negative?>
Z = NOR(BUS[15..0]) <was it zero?>
P = NOT(N)*NOT(Z) <was it positive?>

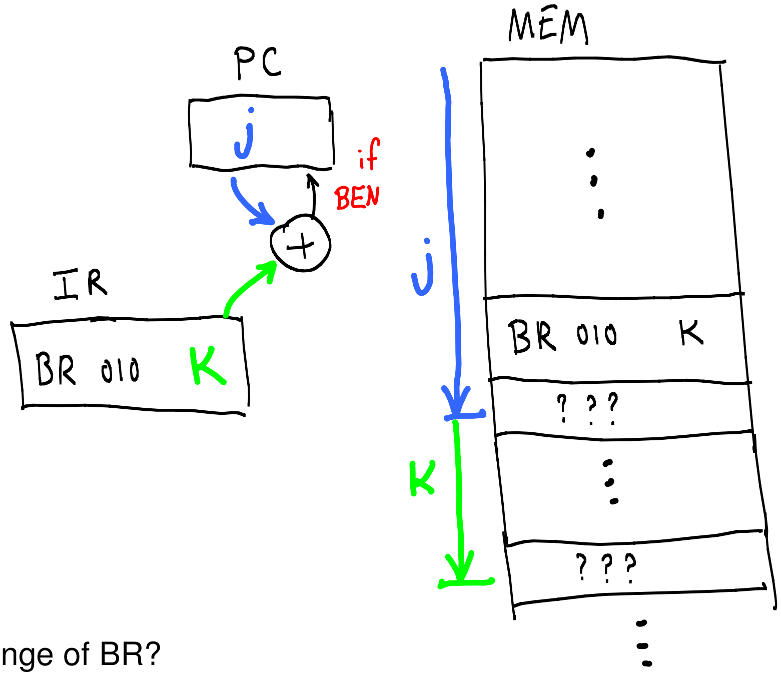
(2) BRANCH on result (calculated in State-32):

BEN = (CC & IR[11..9]) && (IR[15:12] == 0000)
affects LD_PC in State-0 (BR)









(BRn, BRz, BRp, BRNz, BRzp
BRNp, Bnzp, BR)

0000 | 0 | 0 | 0 | XX...X IR

BR ≡ NOP

0000 | 1 | 1 | 1 | XX ... X IR

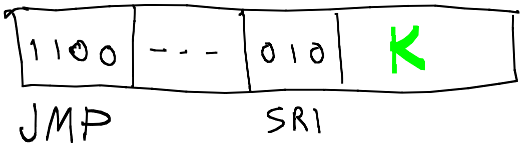
BRnzp ≡ uncond. Br.

Range of BR?
The range of BR is limited. We need to be able to jump anywhere. We could reach anywhere w/ chained BRs. But we'd like another instruction that jumps anywhere.

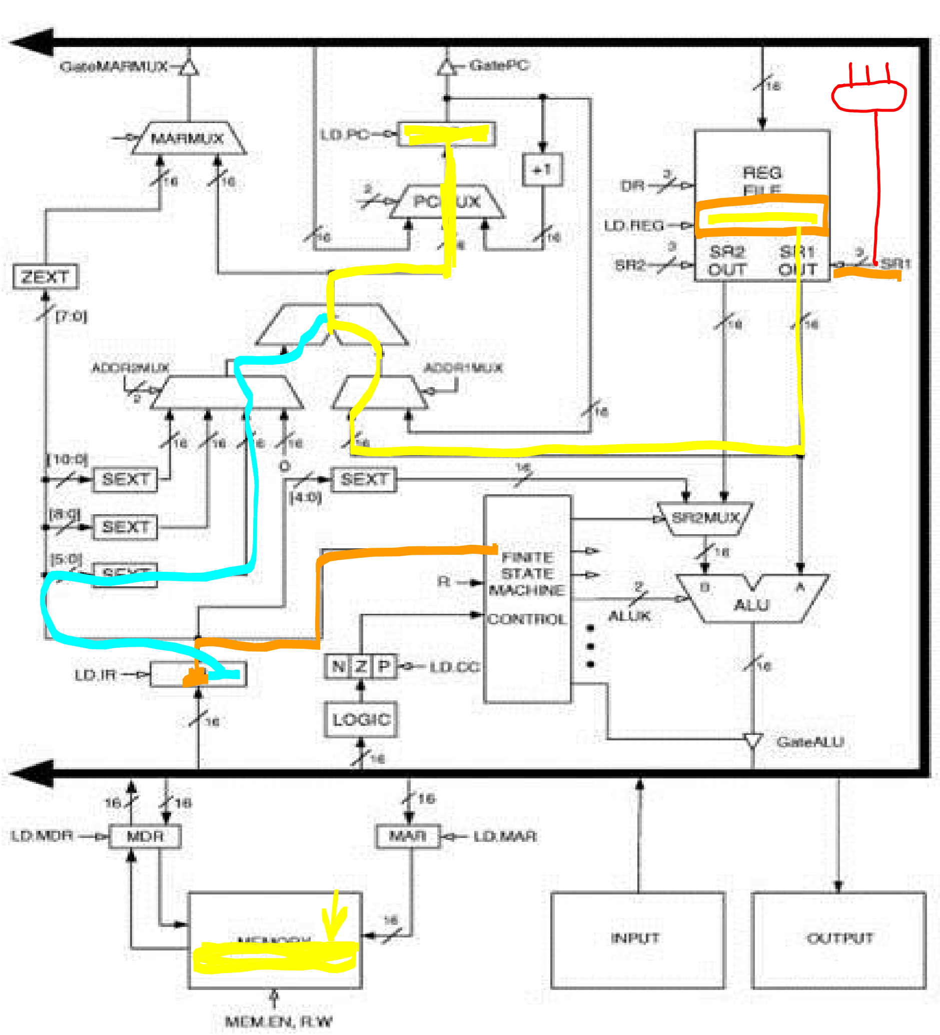
Jump via register

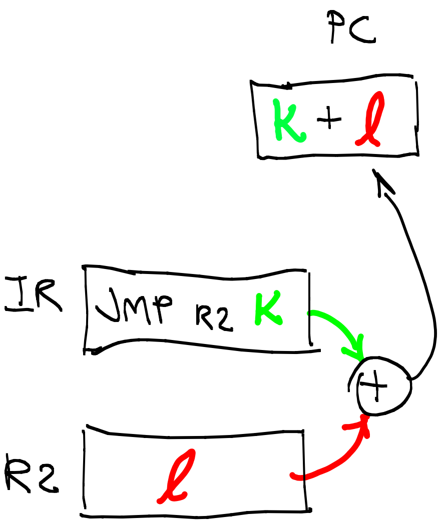
PC <= REGfile[SR1] + IR[5:0]

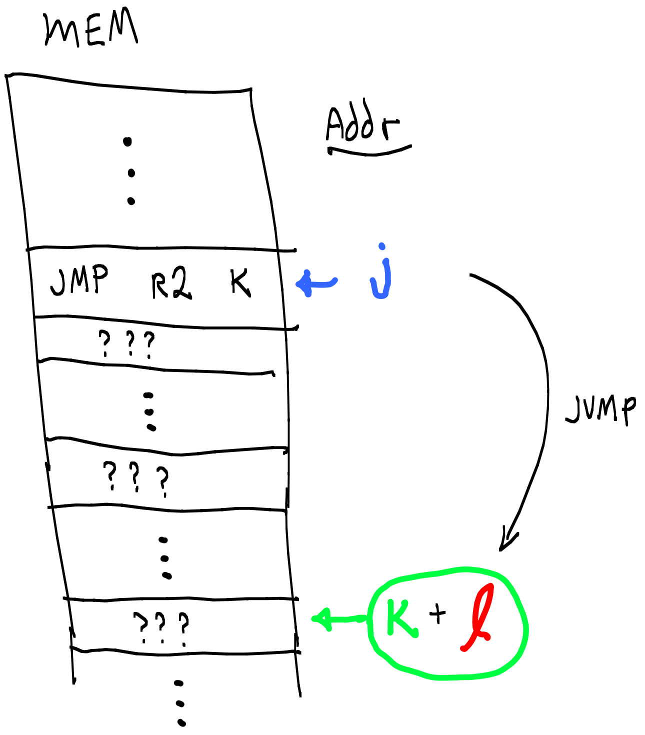
The ability to use any 16-bit address:
jump anywhere.



Jump via reg + offset







Function calls

ABSTRACTION == FUNCTIONS: Write code ONCE -- use ANYWHERE

What we have so far: AND, NOT, ADD, LD, ST, LEA, BR, JMP

Can we jump:
-- TO function code FROM anywhere?
-- BACK to where we came from?

IDEA: use
-- MEMORY for TO-part
-- REGISTERS for BACK-part

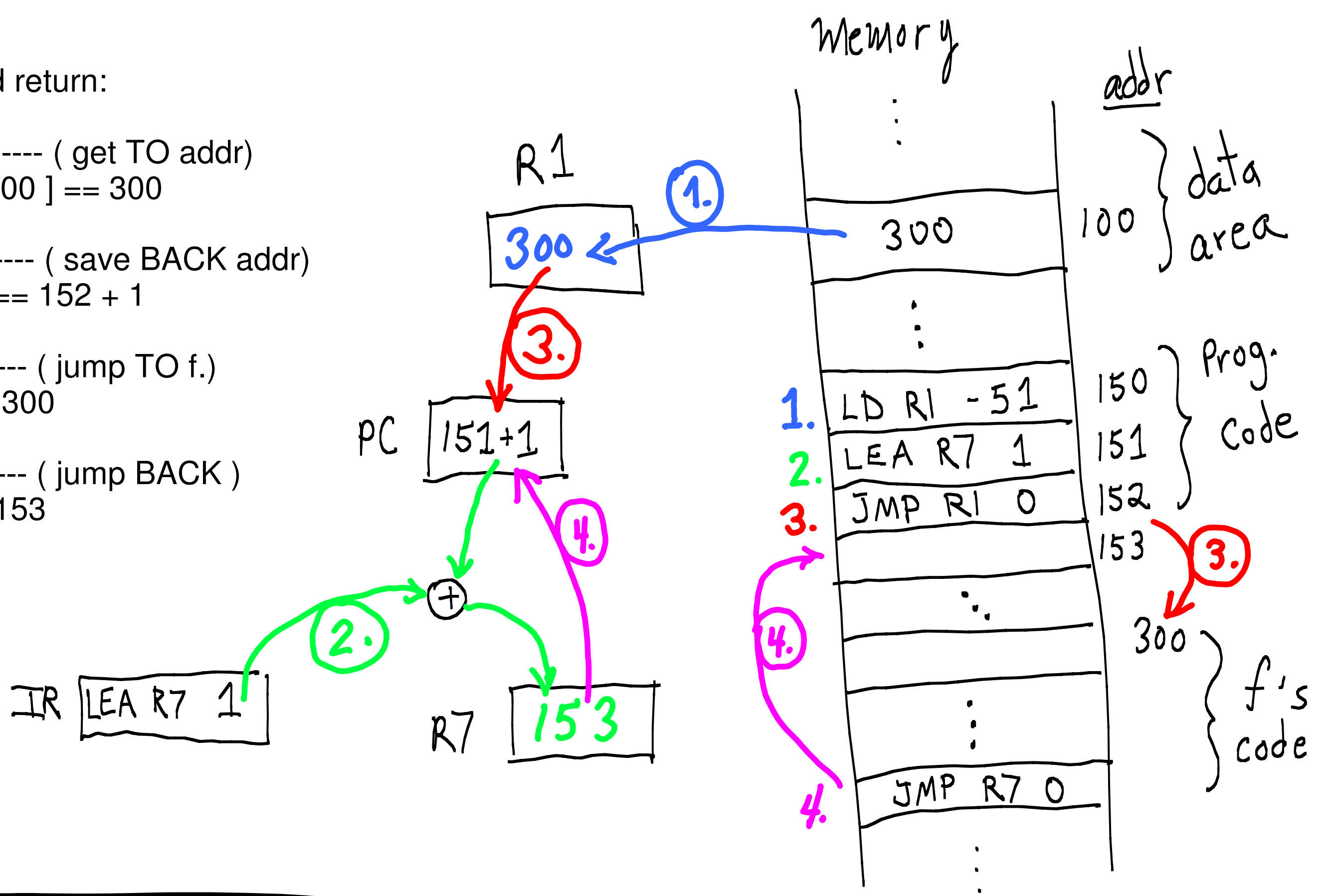
Funtion call and return:

1. LD R1, -51 //------- (get TO addr)
R1 <== Mem[100] == 300

2. LEA R7 1 //------- (save BACK addr)
R7 <== PC+1 == 152 + 1

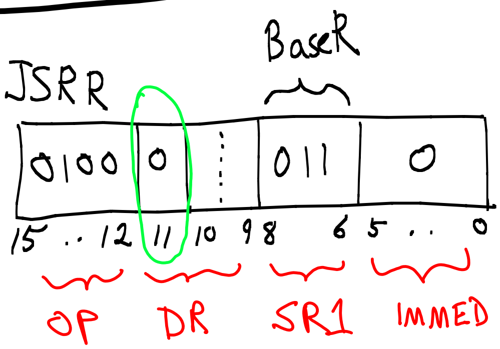
3. JMP R1 0 //------ (jump TO f.)
PC <== R1 == 300

4. JMP R7 0 //------ (jump BACK)
PC <== R7 == 153

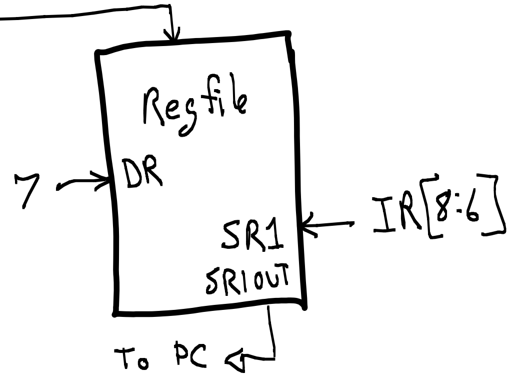


JSR, JSRR, RET

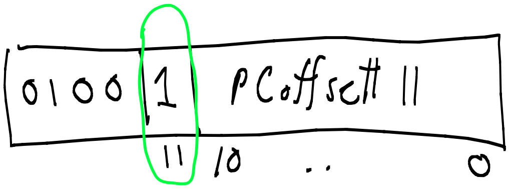
JSRR R3



PC ← Regfile[SR1]
R7 ← PC



JSR #100



PC ← PC + IR[10:0]
R7 ← PC

			R7	
JMP	1100	xxx	BR	xx..x
RET	1100	xxx	111	xxx..x

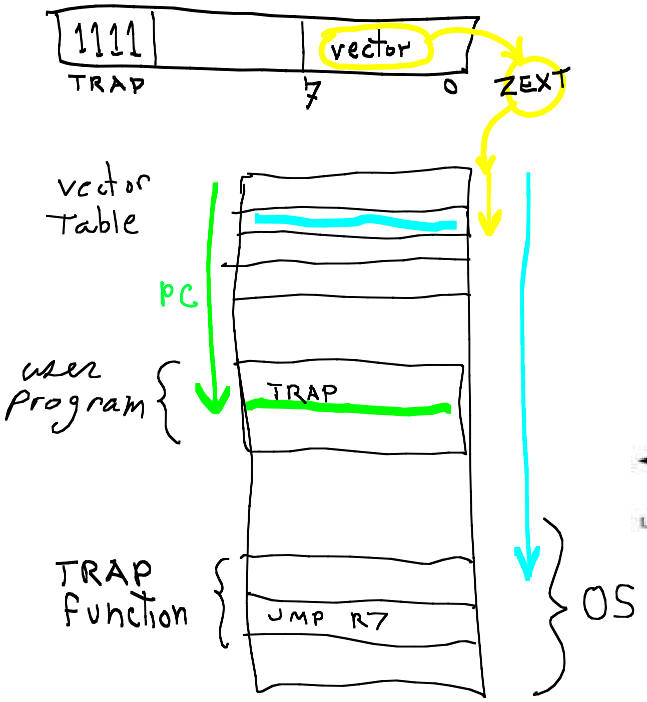
RET = JMP R7 : PC ← R7

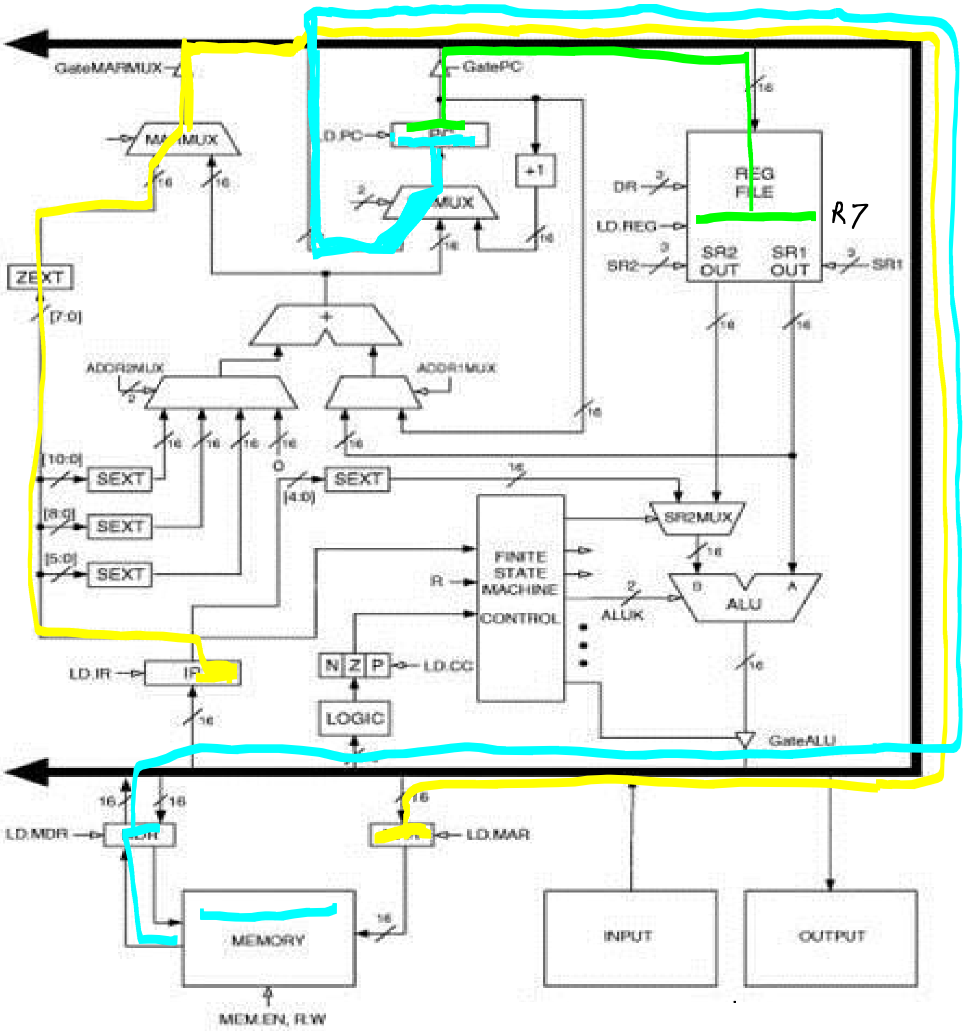
TRAP (indirect function call)

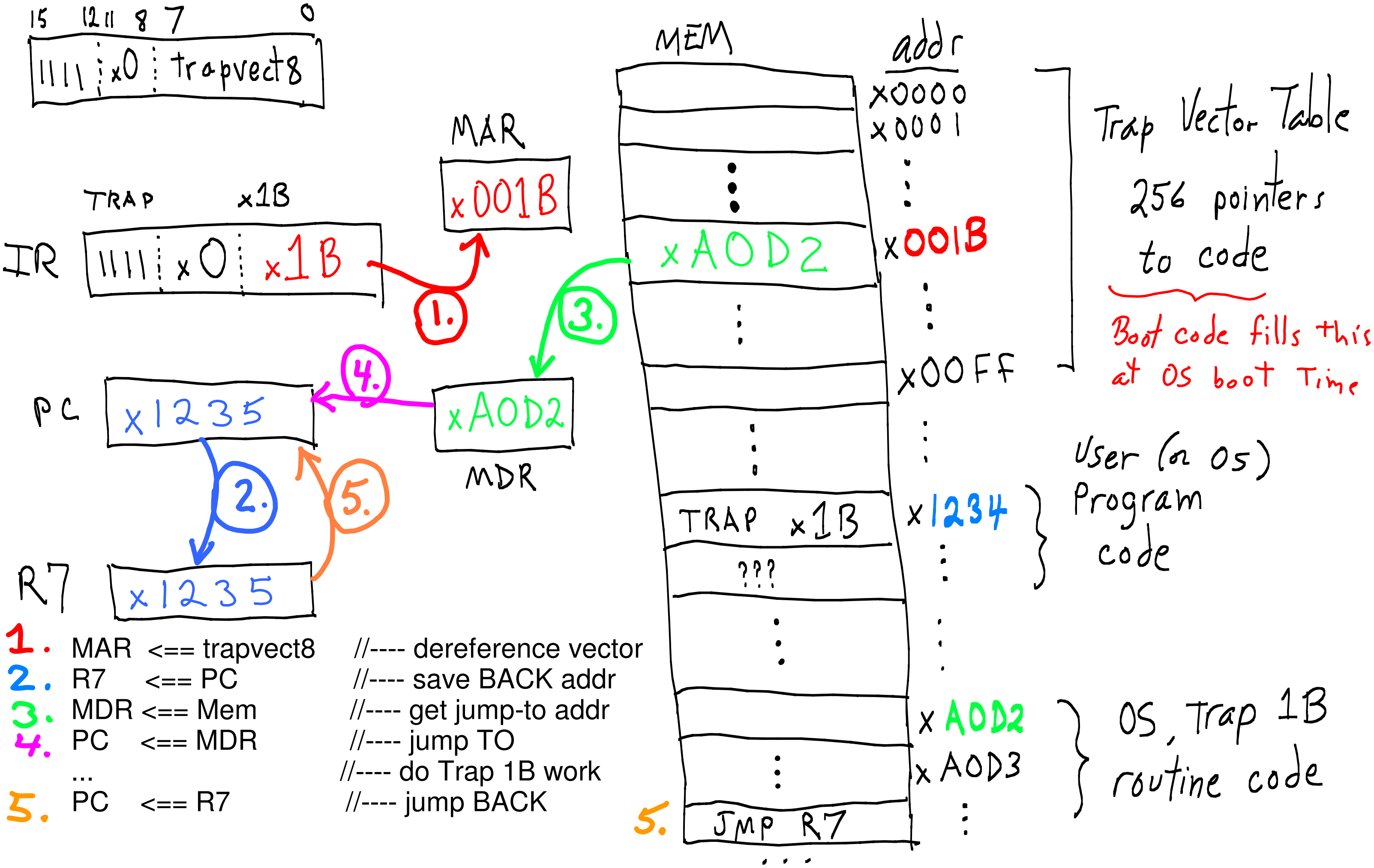
State 15:
----(get address of function's TRAP VECTOR TABLE slot)---
MAR <= ZeroExtend(IR[7..0])

State-28:
--------(fetch function's address, save "return" address)----
MDR <= MEM
R7 <= PC

State-30:
---------(jump to function)----
PC <= MDR







```
1. MAR  <== trapvect8     //---- dereference vector
2. R7   <== PC            //---- save BACK addr
3. MDR  <== Mem           //---- get jump-to addr
4. PC   <== MDR           //---- jump TO
   ...                    //---- do Trap 1B work
5. PC   <== R7            //---- jump BACK
```

--- User's or OS's code can jump to OS conveniently: jump via vectors, not directly, no need to know function's address when writing program, just use its vector number. Future versions of OS code can be moved without causing programming errors; OS can also move itself during runtime. (OS initializes/rewrites vector table.)

--- Uses same return mechanism as our usual function calls (JMP R7, aka "RET").

--- TRAP is a type of function call, but where does a program put the function's arguments? For that matter, how do JSR or JSRR function calls get there arguments? Possibilities: registers, memory stack (more later).

--- Trap Vector Table TVT: 8-bit vector numbers, 256 vectors in TVT.
--- The vector_number-to-address translation is easy:

1. prepend 12 zeroes on to vector number to get vector's 16-bit address.
2. use that address, which points to a slot in TVT, to fetch vector ("vector" = address of function).

--- One vector can be used by multiple functions: code at vector address looks at content of specified register and then jumps to particular corresponding function. Linux uses TVT vector 80 for all entries to OS: 32-bit register allows for 4-G different functions.

--- Example,TRAP routines doing I/O:
OS knows how to talk to I/O devices via device registers. Access to device registers can be

1. through (LD/STR) using memory addresses ("memory mapped" as in LC3)
2. via special instructions (IN/OUT) and separate address space (as in x86)

OS knows how to do polling of I/O devices and handle interrupts generated by devices.
--- All device controller code is in OS, user's code never needs to know details, much simpler for developers.

--- Other machine mechanisms that are similar to TRAPS:

1. Interrupts: mechanism for devices to make service requests, a function call to OS.
2. Exceptions: jumps to OS for error handling: divide-by-zero, illegal opcode, ...