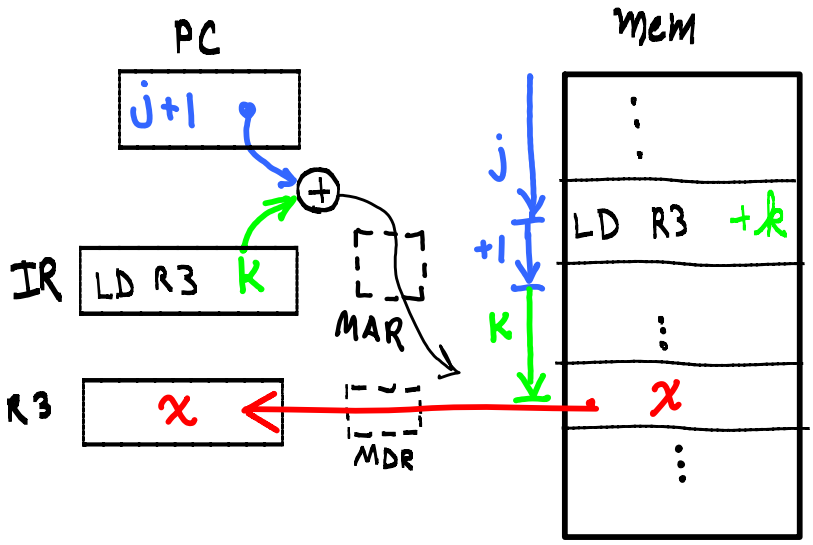
Addressing Modes

PC-relative

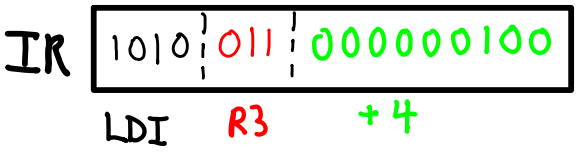
MAR <=== **PC** + **IR**[8 : 0]

Could also be thought of as PC used as a pointer



PC - indirect

MAR ← PC + IR[8:0]
MAR ← MDR



Really, a combination

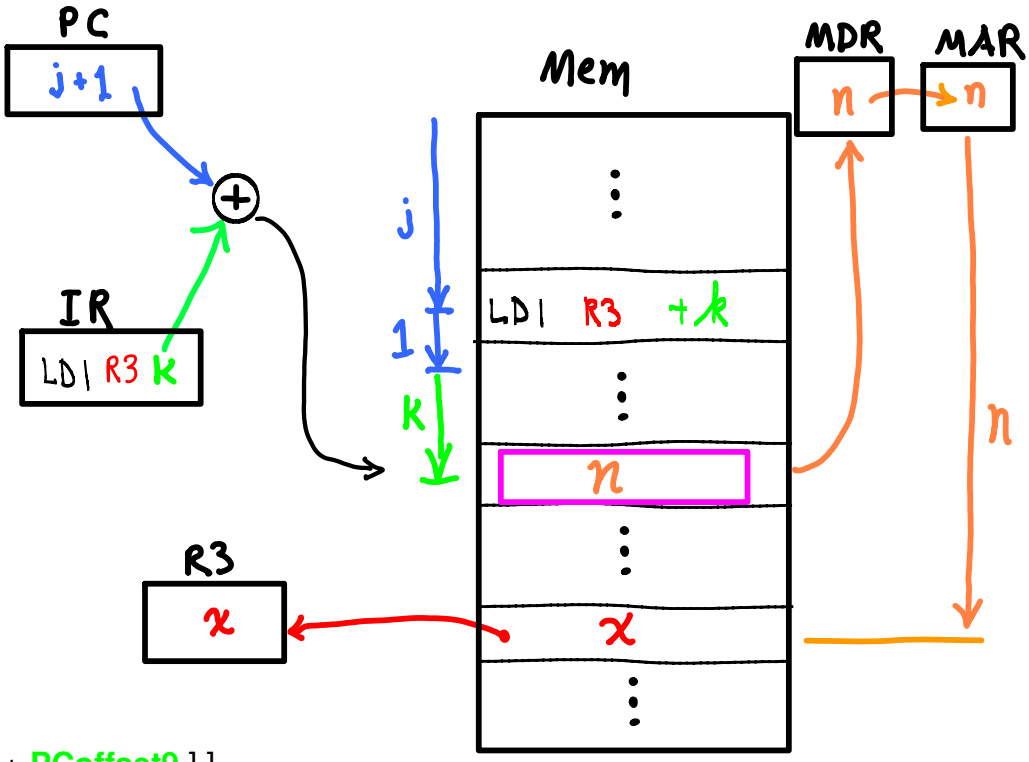
PC-relative
+ memory-indirect

memory cell at address

[j+1+k]

is/contains

A Pointer Variable



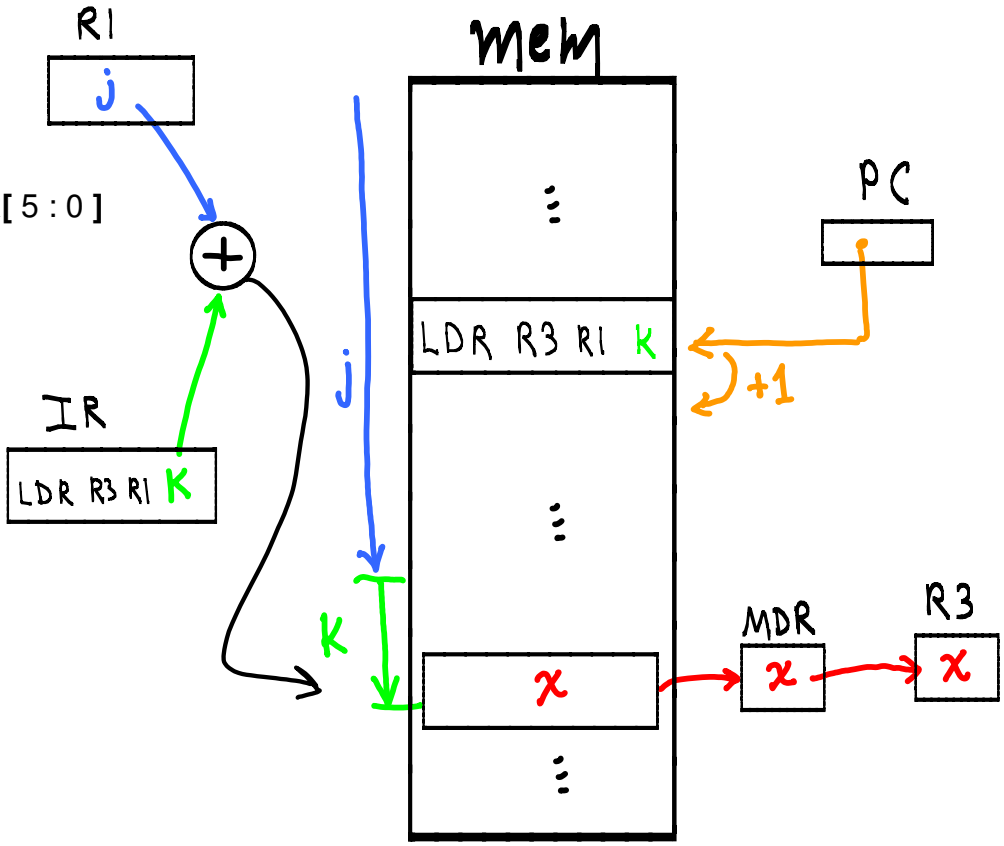
RegFile[DR] <== **Mem**[**Mem**[**PC** + **PCoffset9**]]

Base-offset

aka, register-indirect
aka, register-relative

MAR <=== **RegFile[IR[** 8 : 6 **]]** + **IR[** 5 : 0 **]**

Register used as a pointer variable



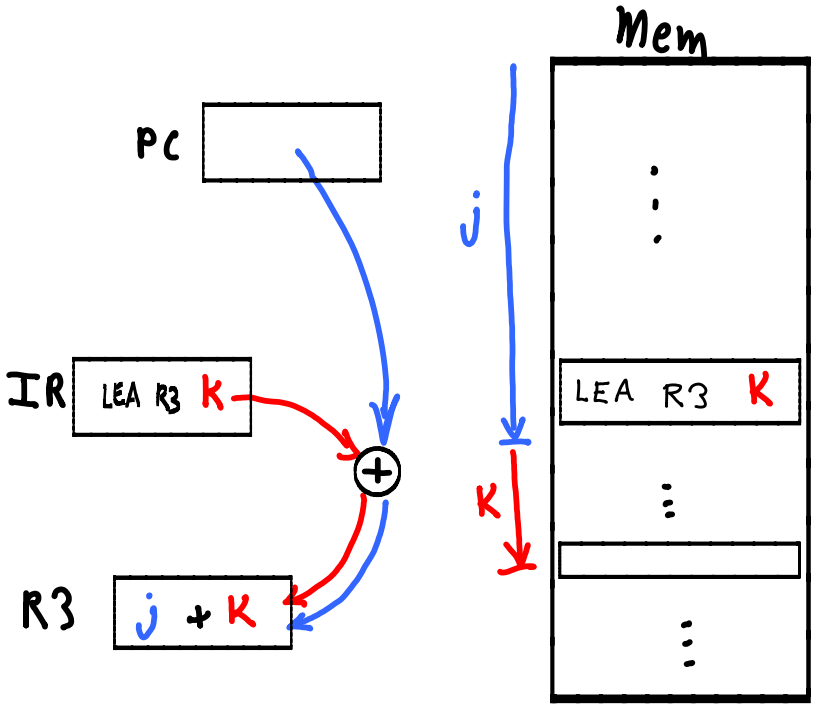
---

immediate

immediate data

RegFile[IR[11 : 9 **]]** <=== **PC** + **IR[** 8 : 0 **]**

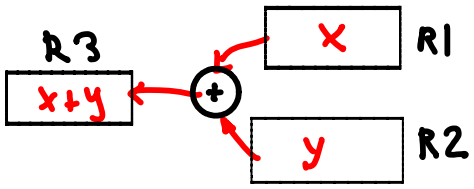
data is available immediately (in IR)



USE **R3** Later as pointer to data, e.g.

LDR R1, R3, 0

Register



ADD, AND

DR <=== **SR1** op **SR2**

NOT

DR <=== NOT(**SR1**)

↑ indicates selected register

Address are **formed** from,

--- **register content** (**PC** or **RegFile**)
--- **IR bits** (a portion of **instruction**)
--- **content of memory**

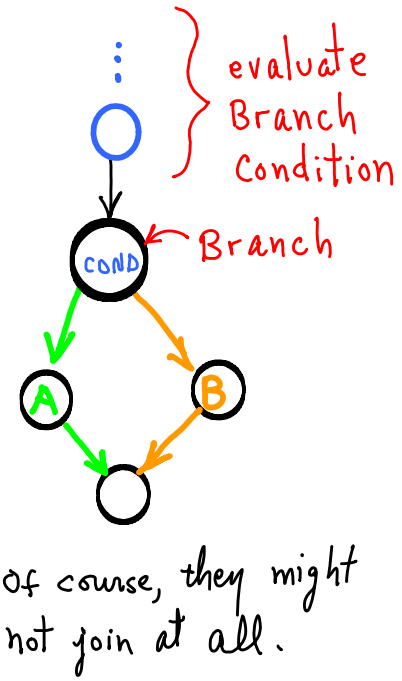
Addresses are **used**,

--- **access memory** (load **MAR**)
--- **change location of instruction fetch** (load **PC**)
--- **saved for later use** (load **RegFile** or **memory**)

WE NEED a **COMPLETE** language for describing machines (TMs).

WE HAVE **functions**: NAND (AND, NOT) is universal (+ bonus, ADD)
WE HAVE **tape**: LD, ST (and variants)

DON'T HAVE **branching** for simulated machine (described machine).

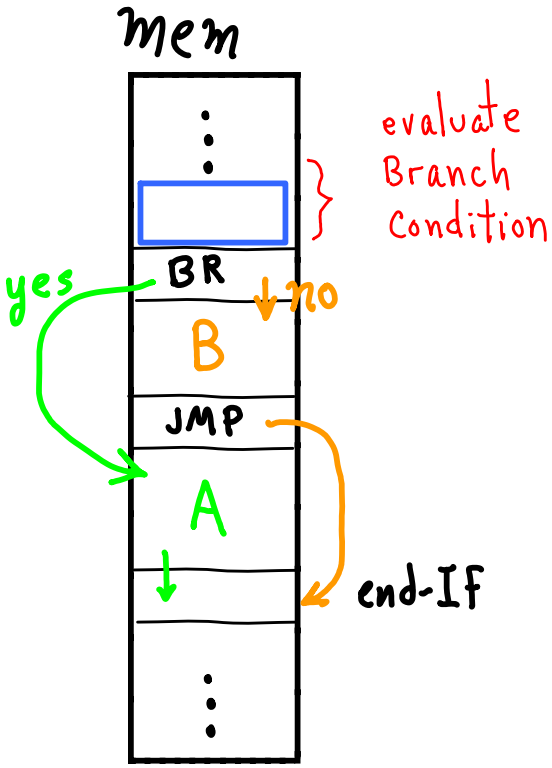


Of course, they might not join at all.

Machine Description

description of
next-state function:
cond <=== F( )

IF (cond) THEN
A
ELSE
B
end-IF



We need Two Language elements

BR, JMP

and a way of 'remembering' cond

Branching: load PC based on **condition evaluation**

--- LC3's **decode** state: **16-way branch**

--- **minimum branching**: **2-way** (if-then)

--- **k-way branches** can be built from **2-ways**

--- **same addressing** for branches

--- **condition** is **function of symbols** read (think, Turing Machines)

--- **compare symbols** (**is-equal** == **difference-is-zero**)

--- **LC3 stores cond** in **PSR Condition Codes** (**CC**) (on all register writes)

--- **is-zero Z**, **is-positive P**, **is-negative N (CC = {N, Z, P})**

(JMP + ?) == Function Calls

--- **abstraction** == **interface** + **hiding details**

--- **low-level abstraction** == **sub-cell** (Electric) == **function** (e.g., C)

--- **code == hardware**

--- **jump into** function (**signal in to sub-cell**'s export)

--- **jump back** from function (**signal out from sub-cell** == return value)

REMEMBER RESULT of FUNCTION EVALUTION

LD_CC

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

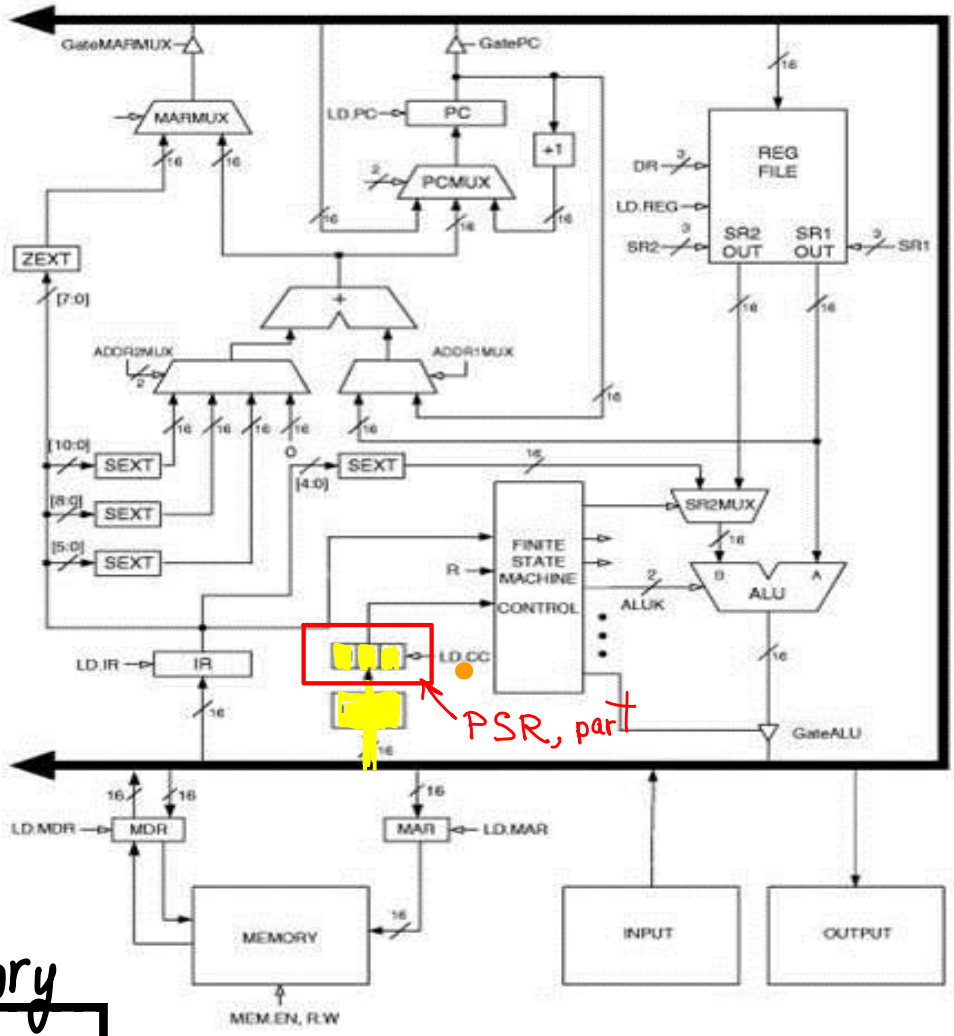
N = BUS[15] <was it negative?>
Z = NOR(BUS[15..0]) <was it zero?>
P = NOT(N)*NOT(Z) <was it positive?>

SAVE BRANCH CONDITION (State-**32**):

BEN <== **(CC** & **IR**[11:9] **)** && **(IR**[15:12] == **0000)**

BEN == 0 : Don't Branch
BEN == 1 : Do Branch

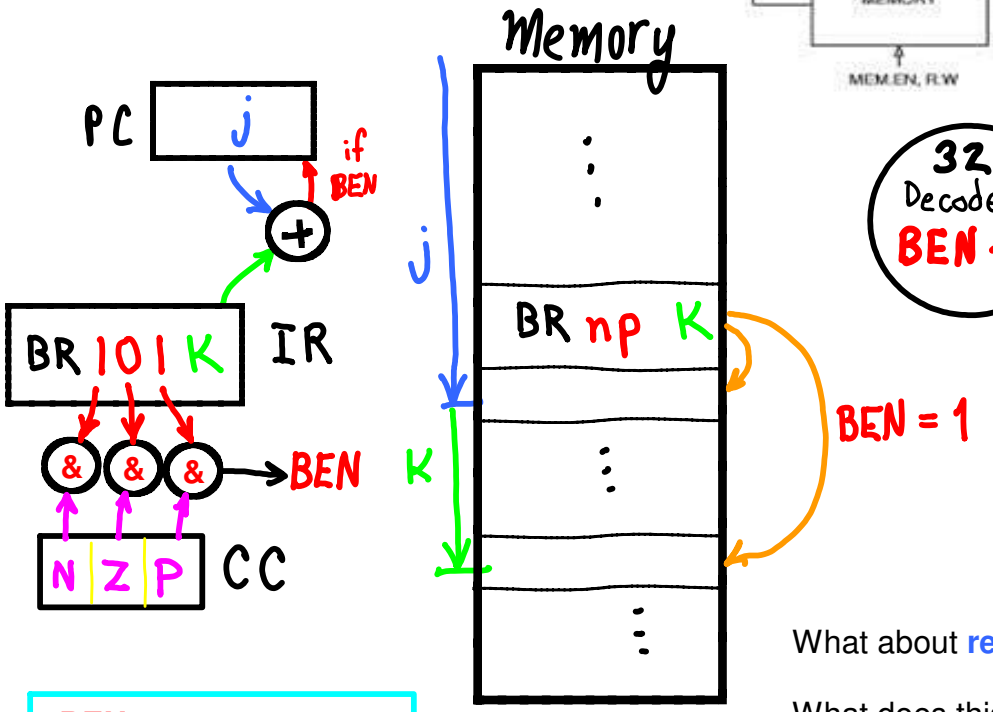
affects **LD_PC** in State-**22** (Branch taken)

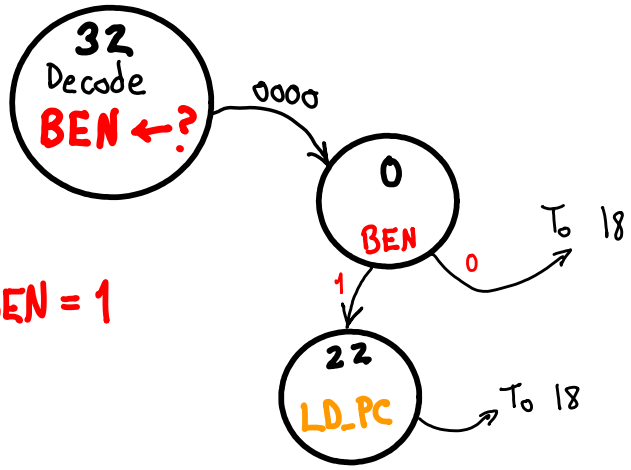


0000 101 0 00001 010 IR

BR NZP PCoffset9

np (not Z)



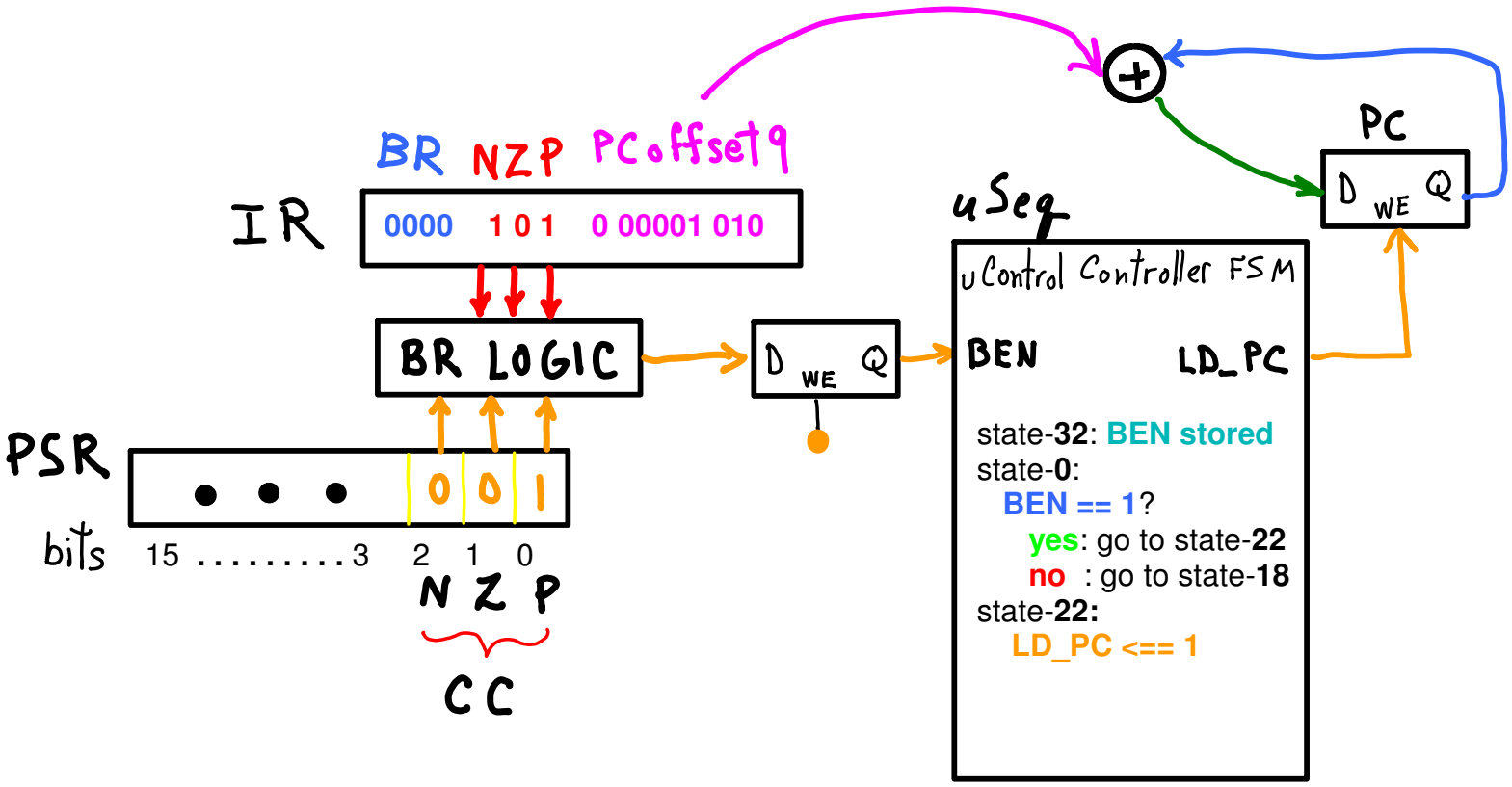


BEN <=== **1**
if
N AND **n** (BRn)
or
Z AND **z** (BRz)
or
P AND **p** (BRp)

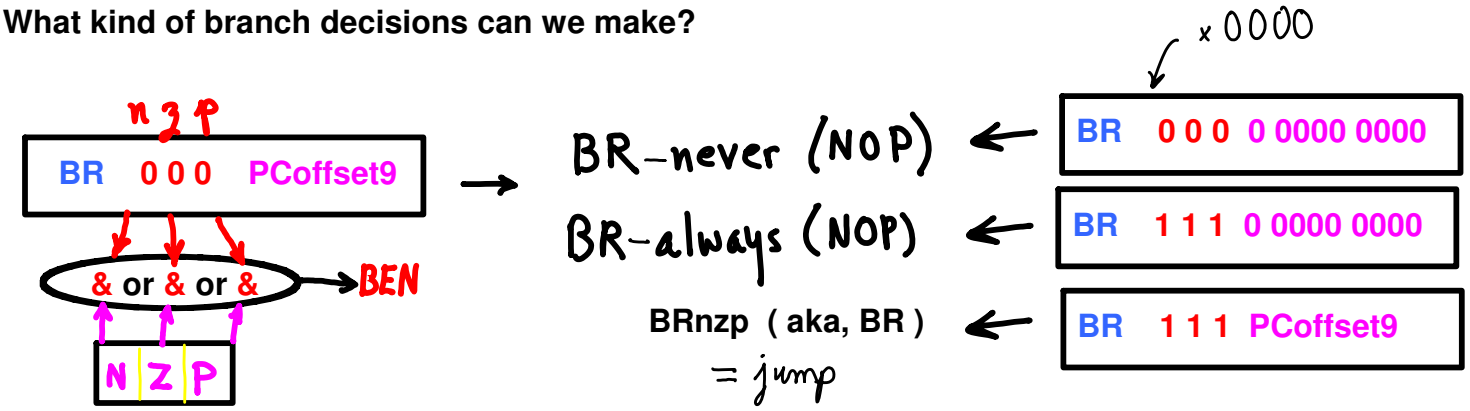
What about **remembering BEN**?

What does this instruction do?
0000 000 1 1111 1111
And this one?
0000 111 1 1111 1111

Range of BR?
The range of BR is limited (9 bit offset). We need to be able to jump anywhere. We could reach anywhere w/ chained BRs. But we'd like another instruction that **jumps anywhere**.



What kind of branch decisions can we make?



BR 1 0 0 PCoffset9 < BRn (BRp)

BR 1 1 0 PCoffset9 ≤ BRnz (BRzp)

BR 0 1 0 PCoffset9 = BRz

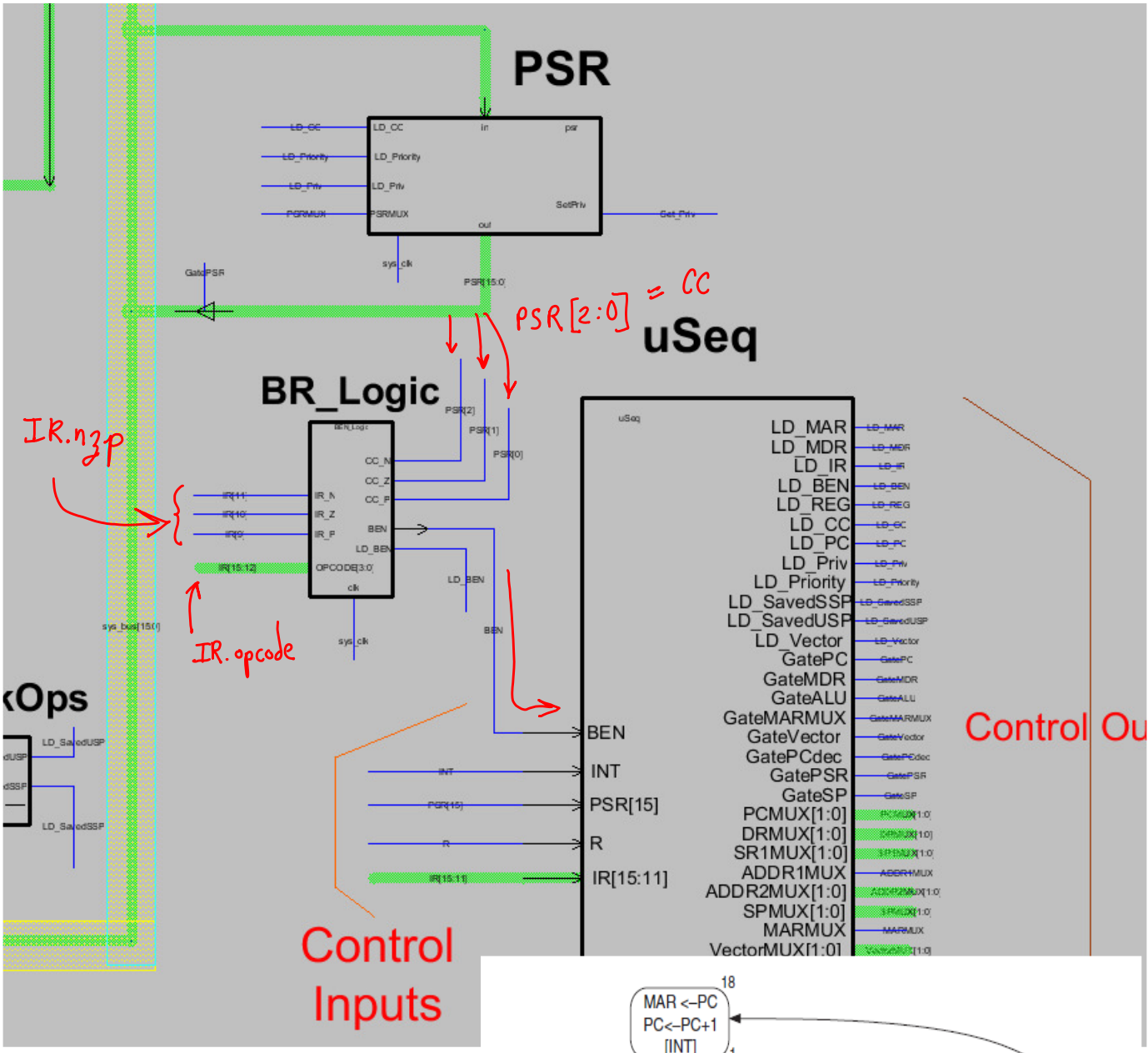
e.g.

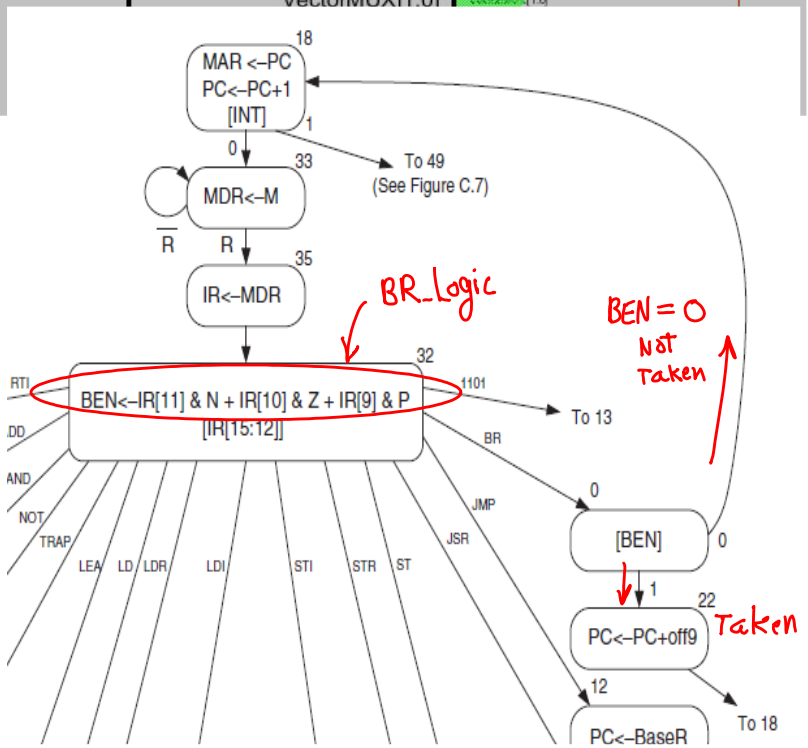
```
;-------------------------
;--- A in R1,  B in R2
;-------------------------
NOT  R3, R2          ;---   R3 <== -B
ADD  R3, R3, #1

ADD R3, R1, R3       ;---  R3 <== A-B

BRz  (+1)            ;---   if ( A == B )
BRnzp (+100)
                     ;--- then
```

else

LC3 Branch Logic

Condition **C**odes are **PSR**[2:0] == { **N**, **Z**, **P** }





control signal

STATE

state-**32:** LD_BEN[**32**] = 1'b**1**;

For any state **k** that writes a register: LD_CC[**k**] <== 1'b**1**

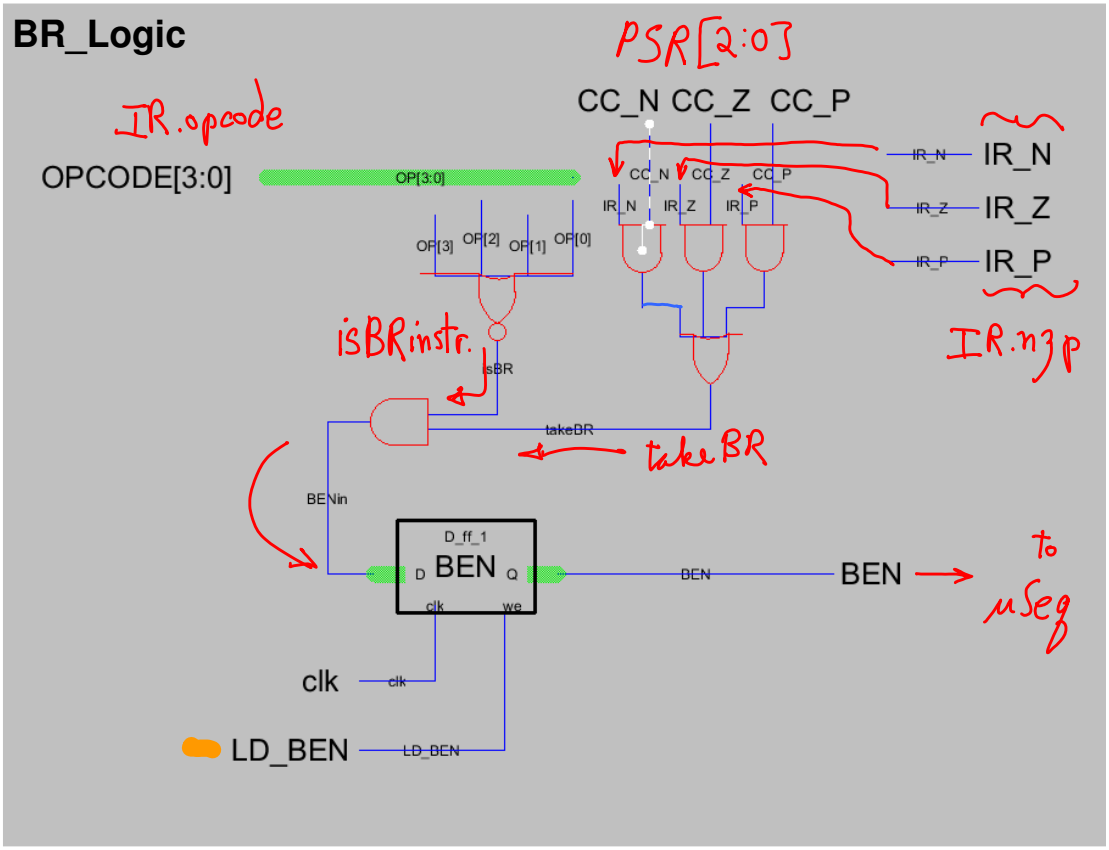
PSR[2 : 0] ==
{ CC_**N**, CC_**Z**, CC_**P** }

IR[11 : 9] ==
{ IR_**N**, IR_**Z**, IR_**P** }

If (CC_**N** == IR_**N** **OR**
CC_**Z** == IR_**Z** **OR**
CC_**P** == IR_**P**)
and
(current instruction is BR)

then

Let controller know to jump
BEN <== **1**



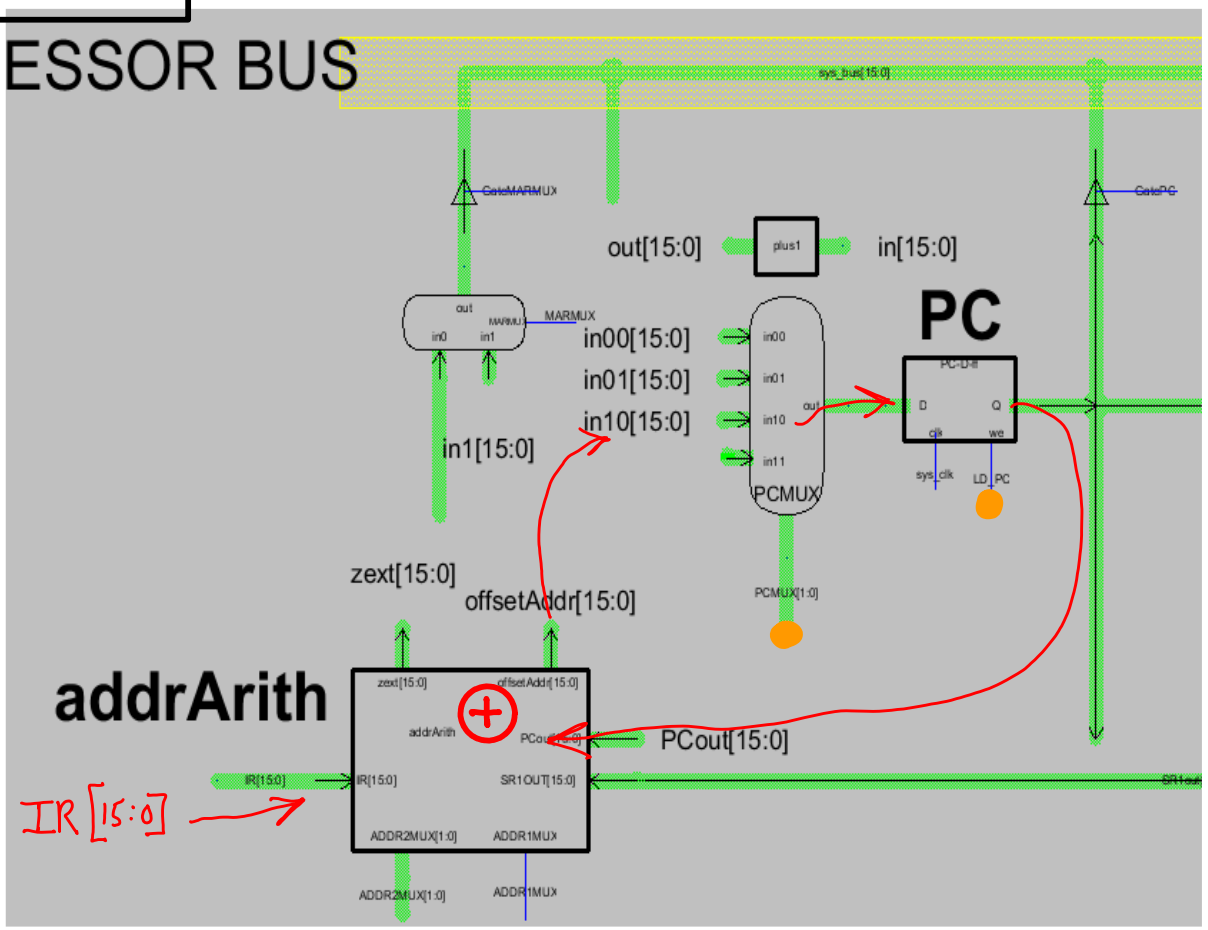
Taking the Branch

PC incremented in fetch-instruction phase.

Offset **IR[** 8 : 0 **]** comes into **addrArith** and through **SEXT**9x16.

Branch target address evaluated in **addrArith**.

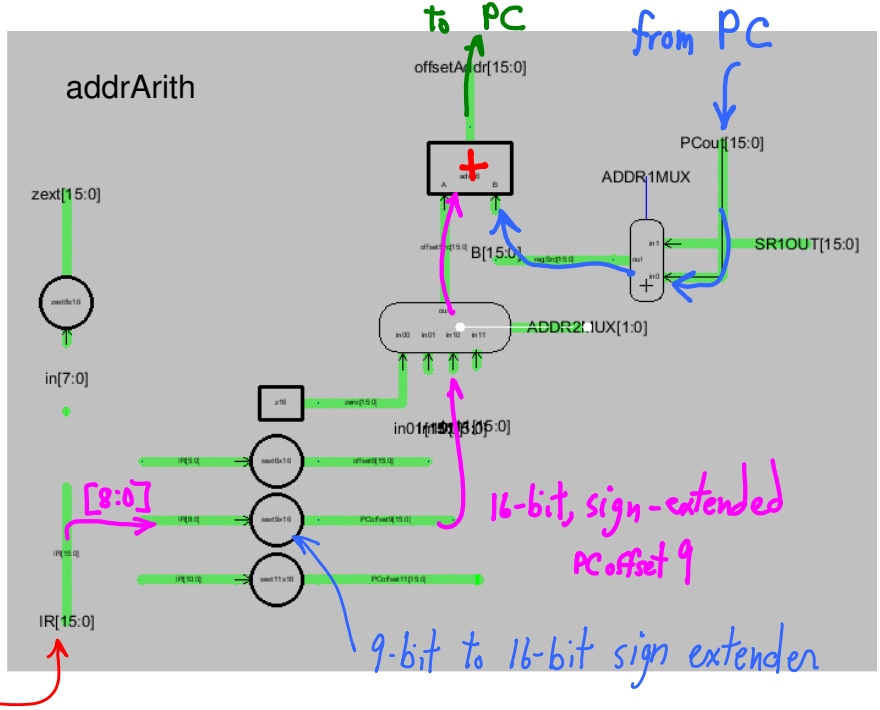
offsetAddr loaded to **PC**.

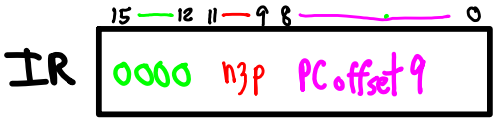


Calculating Target address

offsetAddr is

(incremented **PC**)
+
(**sign-extended PCoffset-9** from **IR**)





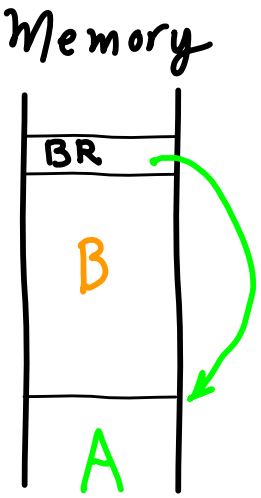
signed, 9-bit value

IR[15:0]

9-bit PCoffset9 ===> + or - (1/2) 2^9
about 2^8 range (256)

Not very far, out of 2^16 (64k) memory locations.

How can we jump farther?



What if B is very big? Need a long jump to A.

But, we still need JMP

Jump via register

PC <= REGfile[SR1]

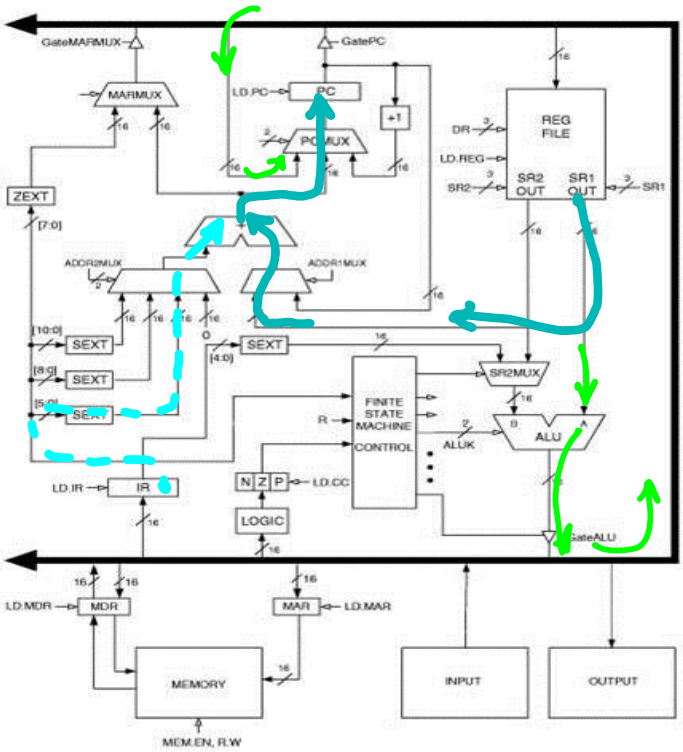
Use any 16-bit address, jump anywhere.

JMP SR1

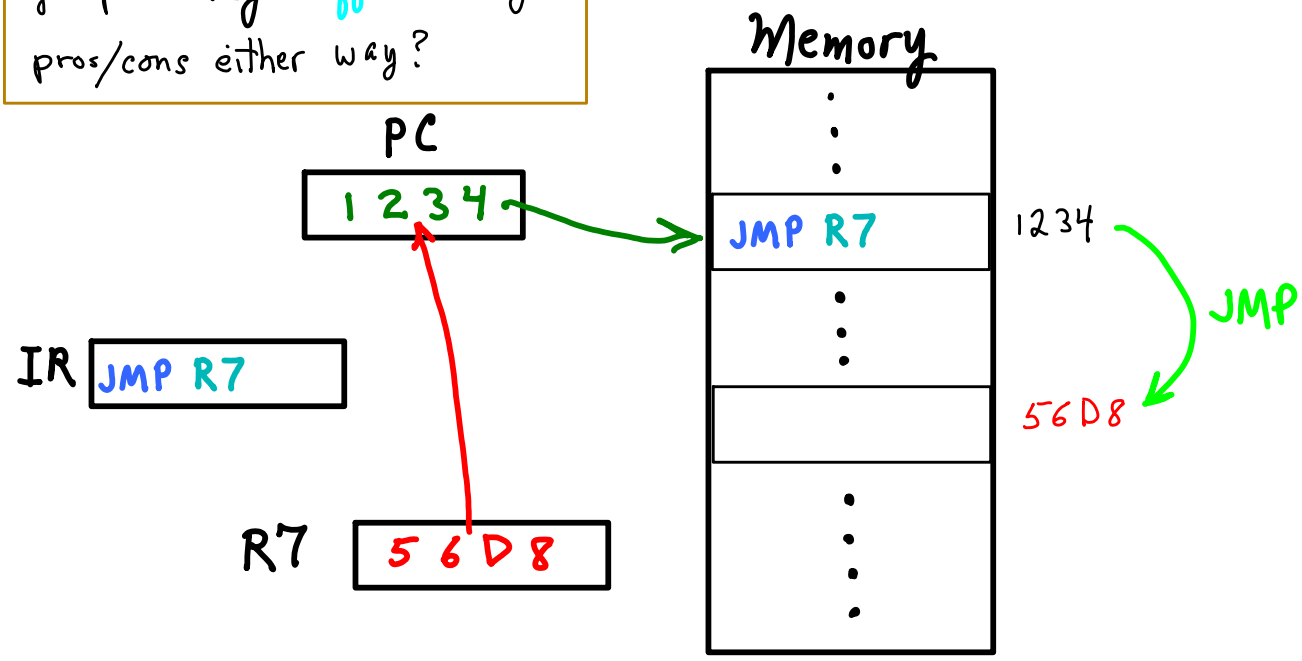
1100	---	111	------

Jump via reg

Could just as easily implement jump via reg + offset. Any pros/cons either way?



alternate path?



Dereferencing a pointer to a function

=

jump to function

A pointer in R7, e.g..

A pointer variable in memory?

How?

function calls

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

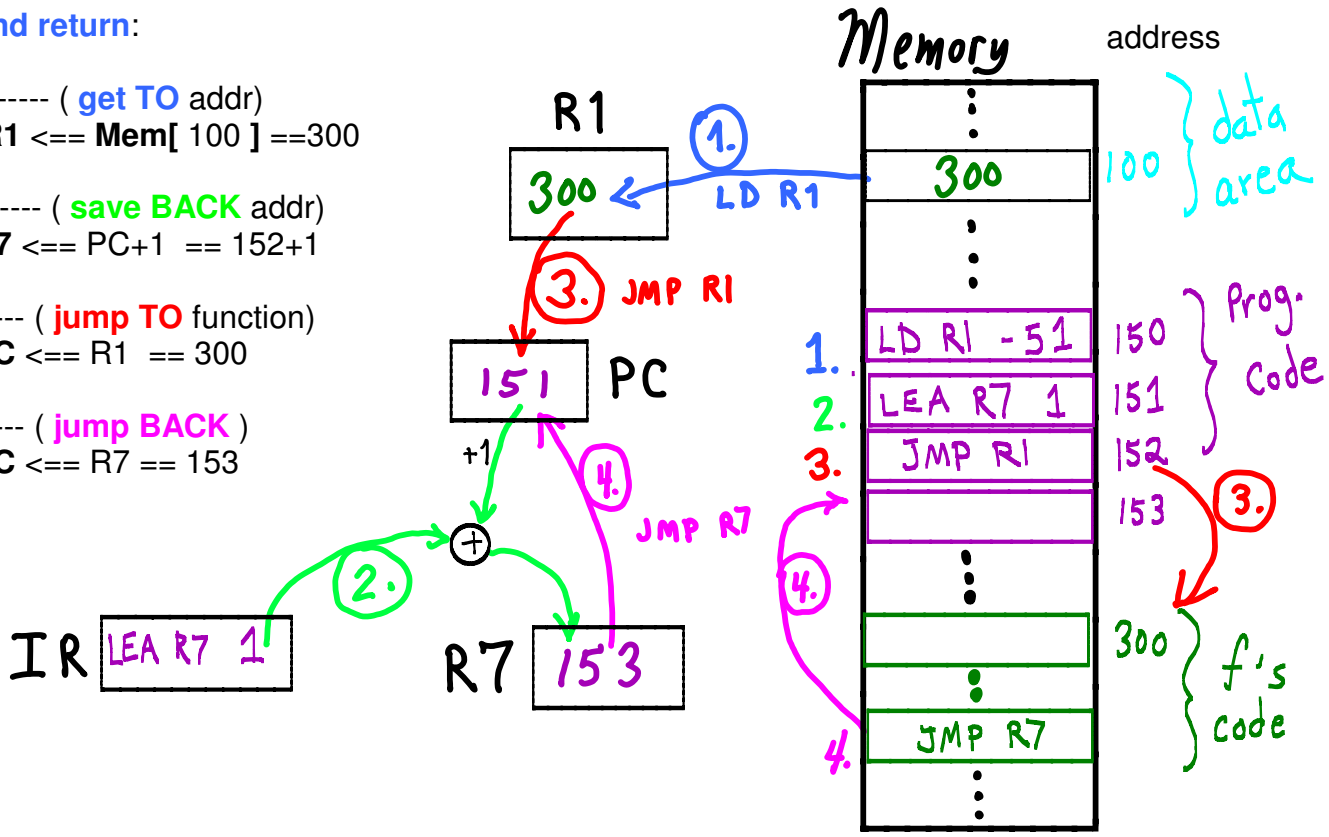
What we have so far: **AND, NOT, ADD, LD, ST, LEA, BR, JMP**

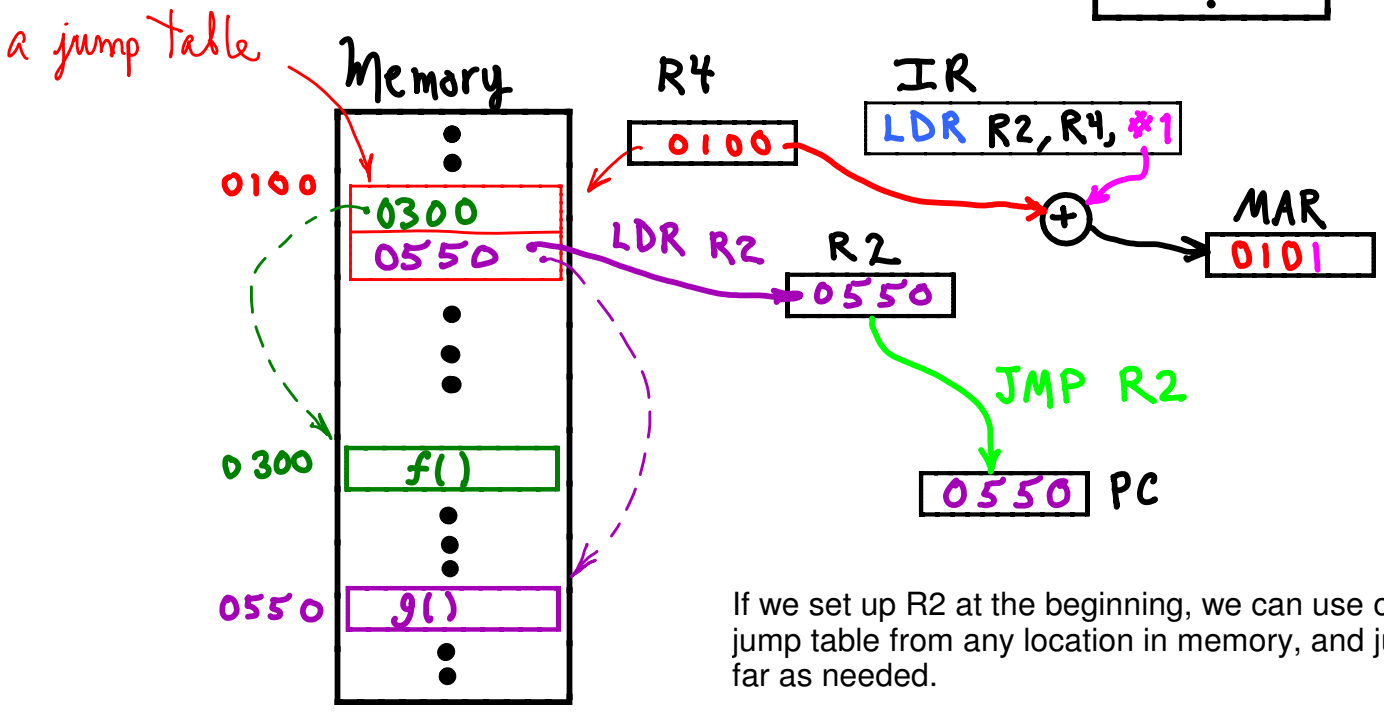
Can we **jump**:
-- **TO** function code FROM anywhere?
-- **BACK** to where we came from?

IDEA: use
-- **MEMORY POINTER** to jump **TO**
-- **REGISTER** to jump **BACK**

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** //------ (**jump TO** function)
   // **PC** <== R1 == 300
4. **JMP R7** //------ (**jump BACK**)
   // **PC** <== R7 == 153





If we set up R2 at the beginning, we can use our jump table from any location in memory, and jump as far as needed.

function calls

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

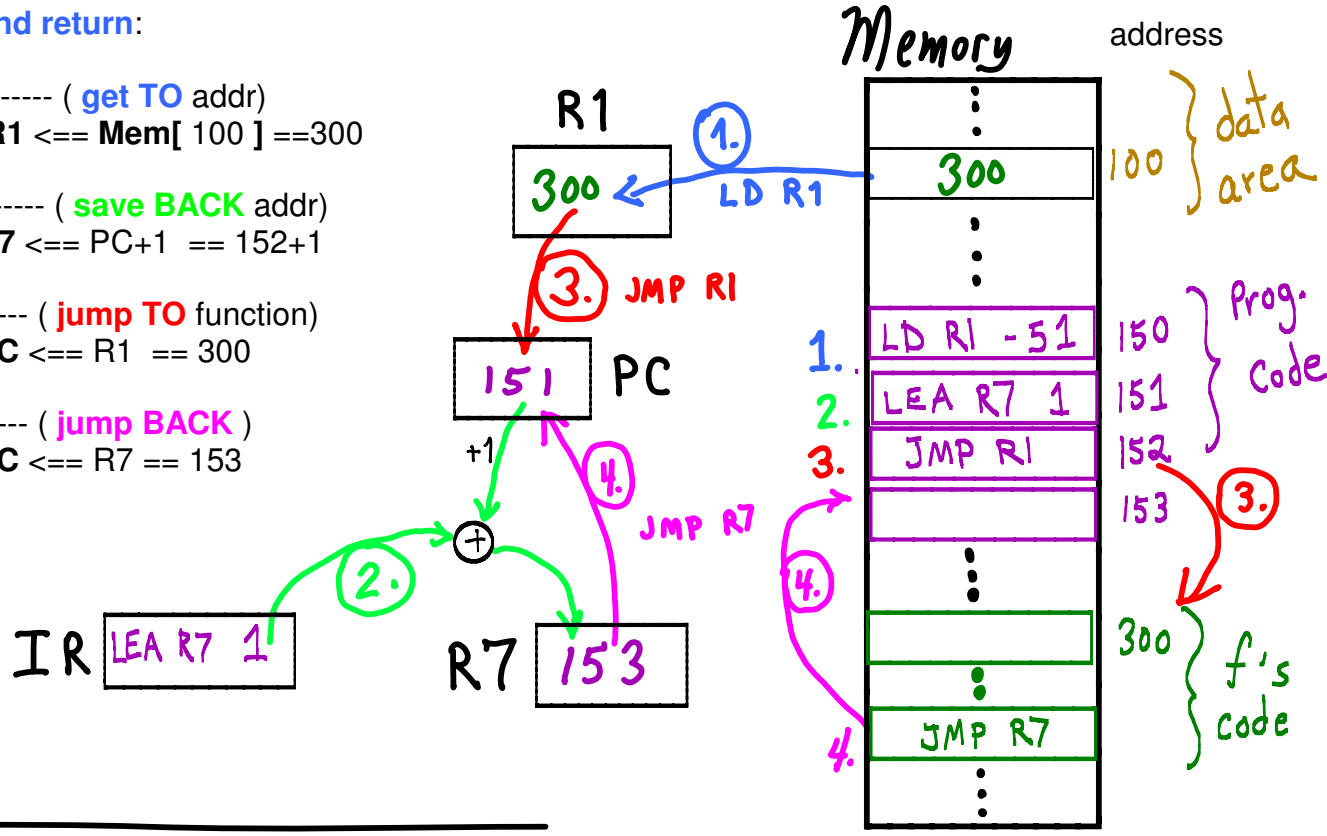
What we have so far: **AND, NOT, ADD, LD, ST, LEA, BR, JMP**

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

IDEA: use
-- **MEMORY POINTER** to jump **TO**
-- **REGISTER** to jump **BACK**

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** //------ (**jump TO** function)
   // **PC** <== R1 == 300
4. **JMP R7** //------ (**jump BACK**)
   // **PC** <== R7 == 153



JSR, JSRR, RET

Function calls are common.
Let's make it easier for the programmer.

aka, BaseR

JSRR SR1

0100	0	--	0 0 1	------

R7 <== **PC** per (2) above
PC <== **RegFile[** SR1 **]** per (3) above

JSR PCoffset11

0100	1	000 1000 0010

R7 <== **PC**
PC <== **PC** + **PCoffset11**

JMP R7
(aka, **RET**)

PC <== **R7**

TRAP (indirect function call: use a pointer to jump)

State-**15**:

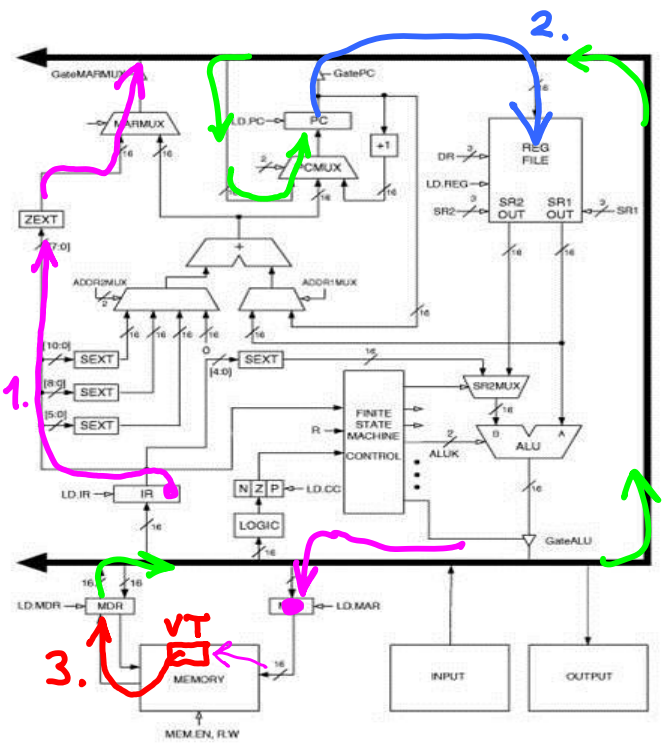
MAR <== ZEXT(**IR[** 7 : 0 **]**) //--- get f()'s VT entry's address (1)

State-**28**:

R7 <== **PC** //--- save "return" address (2)
MDR <== **MEM** //--- get f()'s address from VT (3)

State-**30**:

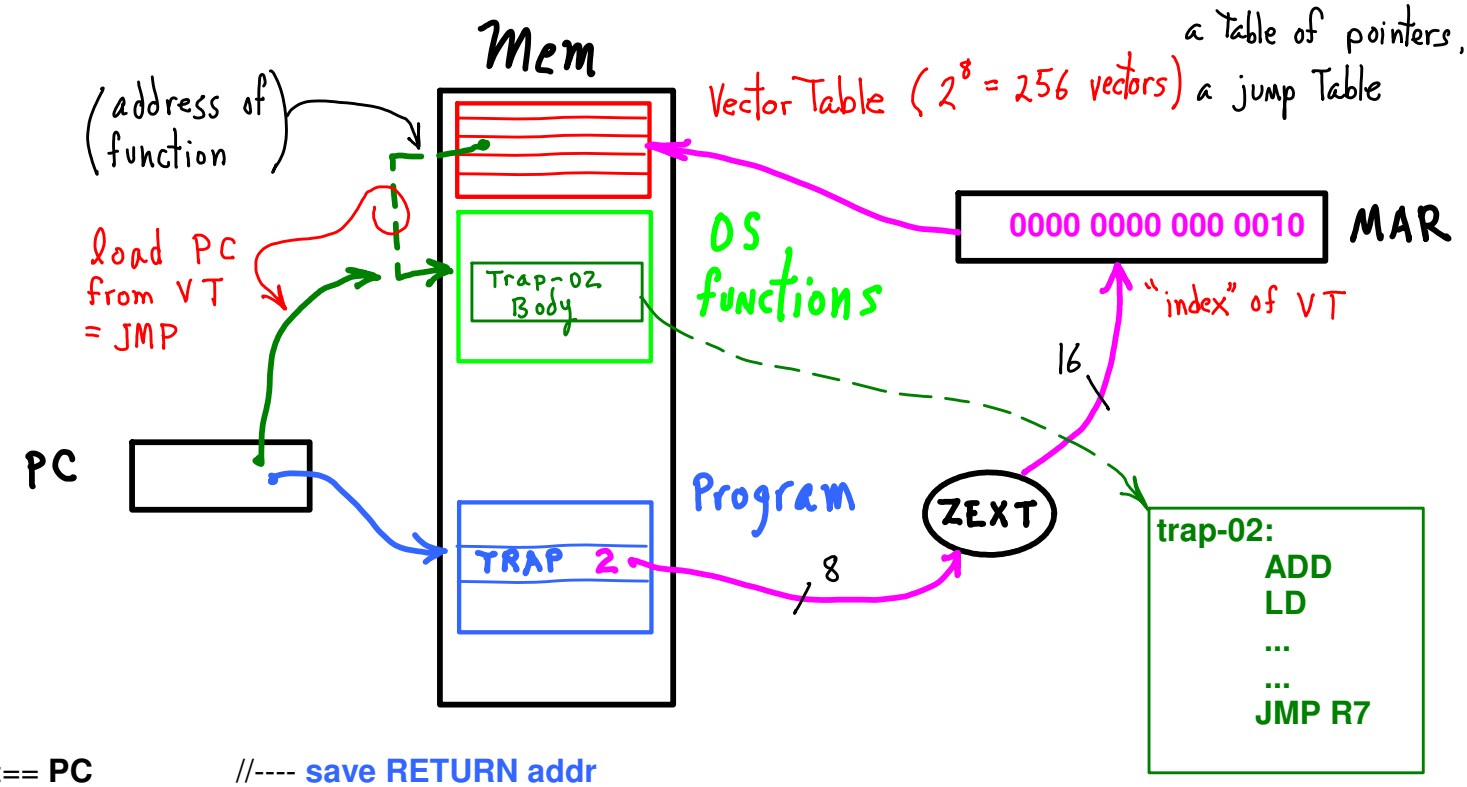
PC <== **MDR** //--- jump to f() (4)



Trap Trapvect8

1111	0000	0000 0010	IR

↖ address of pointer



R7 <== **PC** //---- **save RETURN addr**

MAR <== **trapvect8** //---- **dereference vector, i.e., get**
MDR <== **Mem** //---- **Trap-02's address from VT**

PC <== **MDR** //---- **JMP to Trap-02 body**
...
... //---- do Trap-02 work
...
PC <== **R7** //---- **JMP back to RETURN address**

Ok, great. But why all the bother? Why not just use JSR or JSRR?

--- **TRAP is a FUNTION CALL**, typically to an **OS function**
Programs **never need to know details** ==> much simpler.
Hmmm, but how do functions **get arguments**, **return results**?
Possibilities: **registers, memory, stack** (more later).

--- **PROGRAM INDEPENDENCE**
Programs don't have to know actual code location
jump via STANDARD VECTORS, **OS convention** known/published
OS can move function anywhere, then re-initialize **VT**.

--- **Trap Vector Table** (**VT**), 8-bit index ==> **256 entries [x0000 -- x00FF]**
Or, One-address, multiple-functions? Number in register chooses which function.
Linux uses **VT vector 80 for all entries to OS**: 32-bit register ==> 4G functions possible.

--- **TRAP routines doing Input/Out: OS provides services to programs**
OS talks to I/O devices via device registers:
1. **LD/STR** and **memory addresses** ("memory mapped I/O" as in LC3)
2. **IN/OUT** and **separate address space** (x86 architecture)

OS contains all "driver" code, access via TRAP

--- **Other mechanisms** similar to TRAPS:
1. **Interrupts**: I/O devices make service requests, jump to OS routine.
2. **Exceptions**: errors such as divide-by-zero, illegal opcode, etc., cause jump to OS.