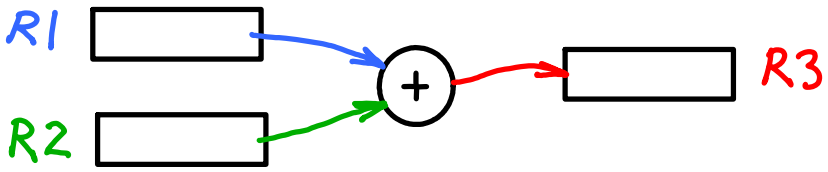
LC3 Addressing Modes

register

data is in a register




```
    ADD   R3,  R2,     R1
IR[ 0001  011  010  0 00  001 ]
```

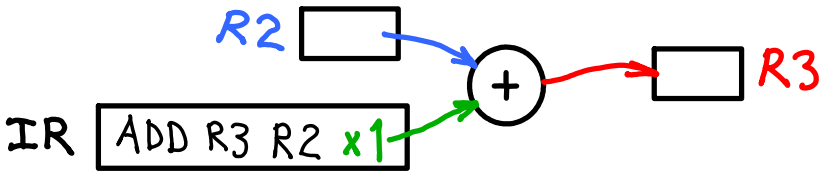
(register-register-register mode)

RegFile[IR[11:9]] <=== RegFile[IR[8:6]] + RegFile[IR[2:0]]

---

immediate

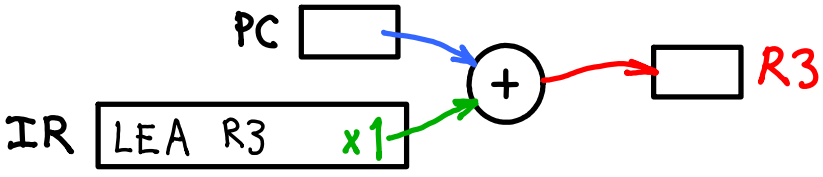
Data is available w/o accessing register or memory




```
    ADD   R3,  R2,     x1
IR[ 0001  011  010  1  00001 ]
```

(register-register-**immediate** mode)

RegFile[IR[11:9]] <=== RegFile[IR[8:6]] + IR[2:0]



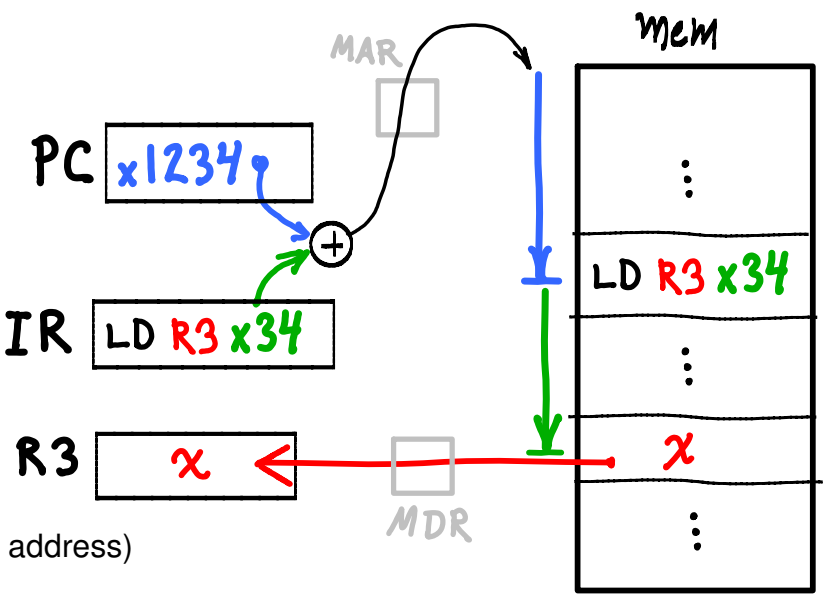

```
    LEA   R3,     x1
IR[ 1110  011  000000001 ]
```

(register-register-**immediate** mode)

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

PC-relative

PC used as pointer variable



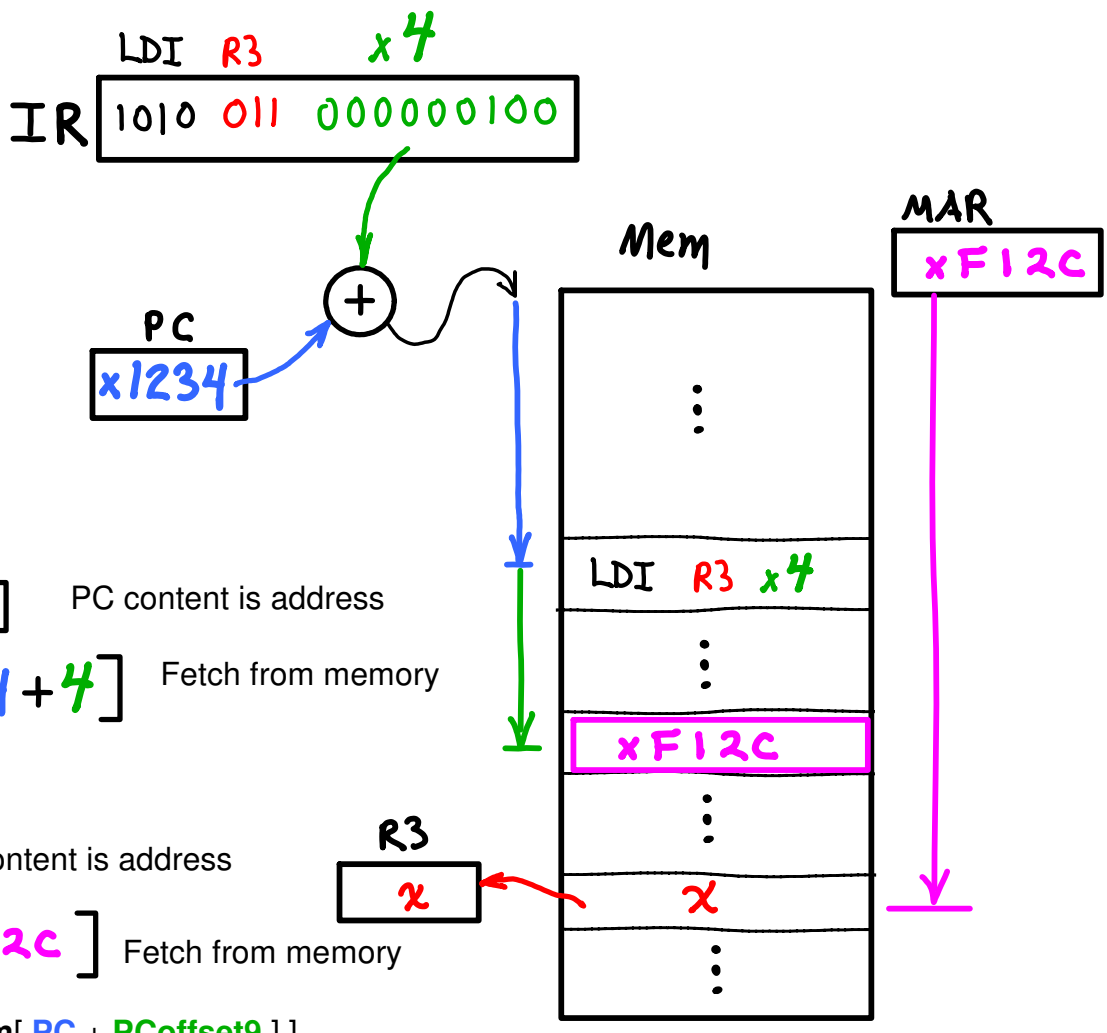
LD R3 x34

MAR <=== PC + IR[8 : 0] (PC content is address)

MDR <=== MEM[x1234 + x34] (Fetch from memory)

PC-indirect

PC is pointer to a pointer variable



Two memory accesses

1. PC-relative

MAR ← PC + IR[8:0] PC content is address

MDR ← MEM[x1234 + 4] Fetch from memory

2. Memory-relative

MAR ← MDR Memory content is address

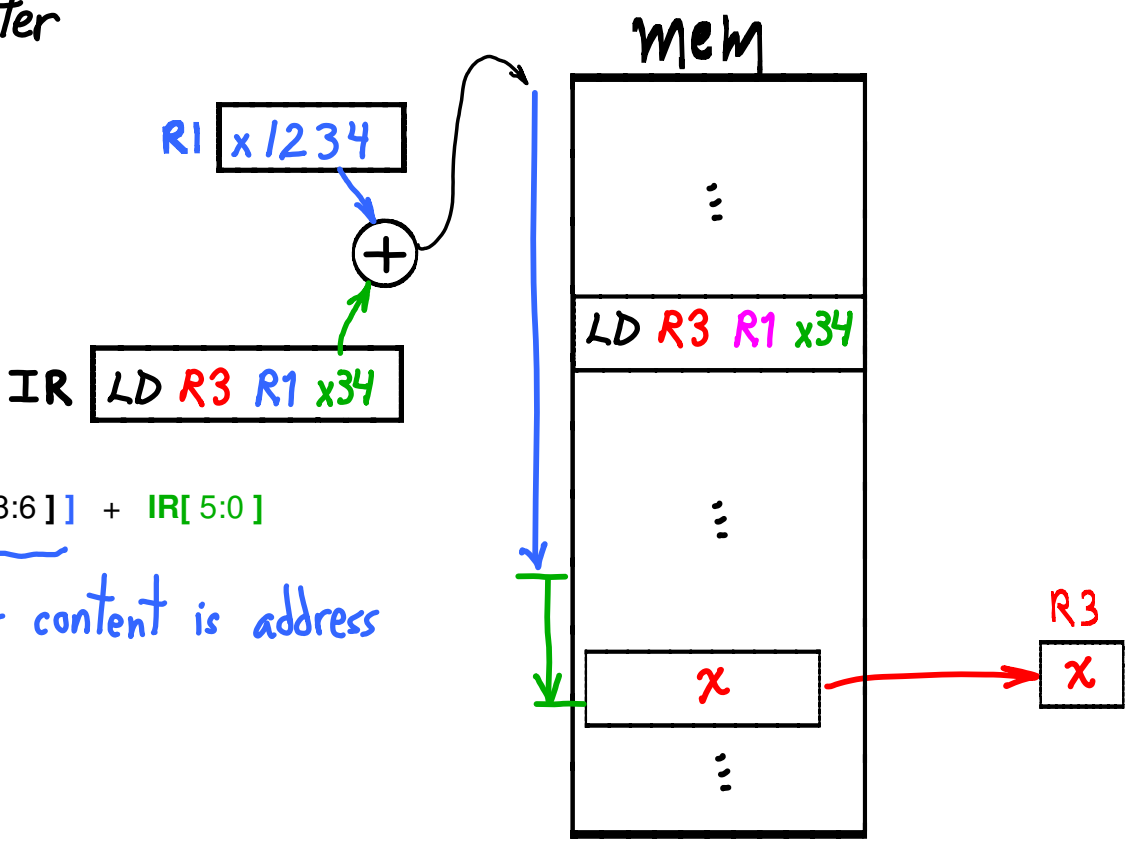
MDR ← MEM[xF12C] Fetch from memory

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

Base-offset

aka, register-indirect
aka, register-relative

register used as pointer

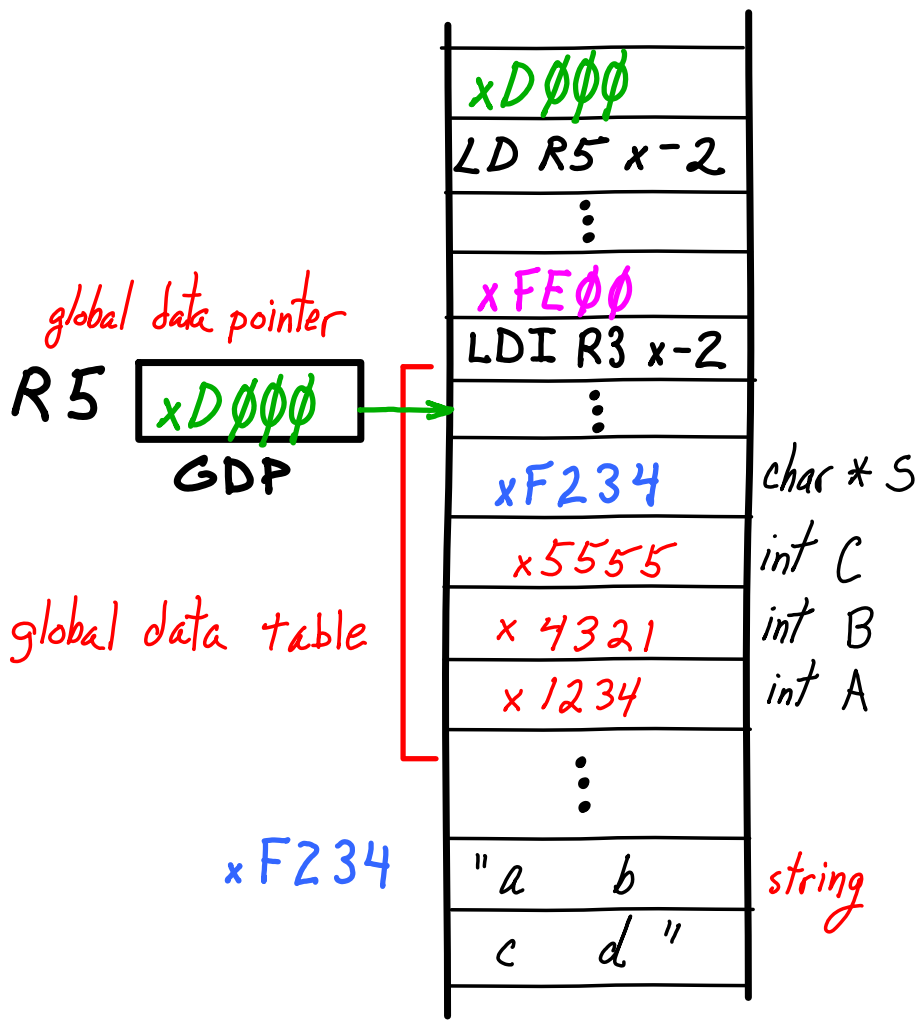


MAR <=== RegFile[IR[8:6]] + IR[5:0]

register content is address

Typical Usage

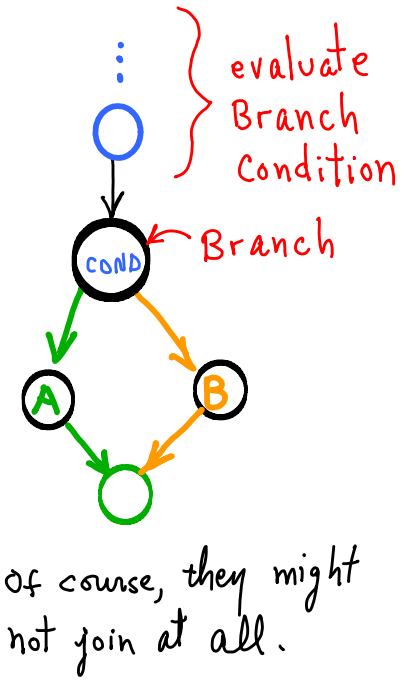
```
1100 0000 0000 0000   ;-- GDP
LD R5 x-2             ;-- init GDP
LDR R3 R5 x3          ;-- read A
LDR R4 R5 x2          ;-- read B
ADD R2 R3 R4          ;-- C <== A+B
STR R2 R5 x1          ;-- write C


LDR R1 R5 x0    ;-- load S pointer
LDR R2 R1 x0    ;-- read "ab"
...
STR R2 R1 x0    ;-- write "wx"


LDI R4 x-2        ;-- read KB i/o reg.
```



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).

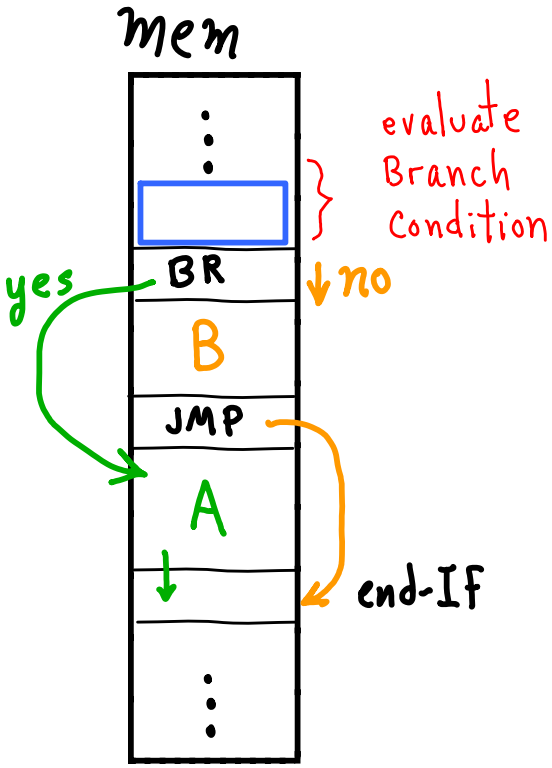


Of course, they might not join at all.

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

Can we simulate any machine?

Some machines have **many-way branching:jj**

--- 32-bit symbols ===> 4G-way branching

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

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

Branching: load the PC based on condition evaluation

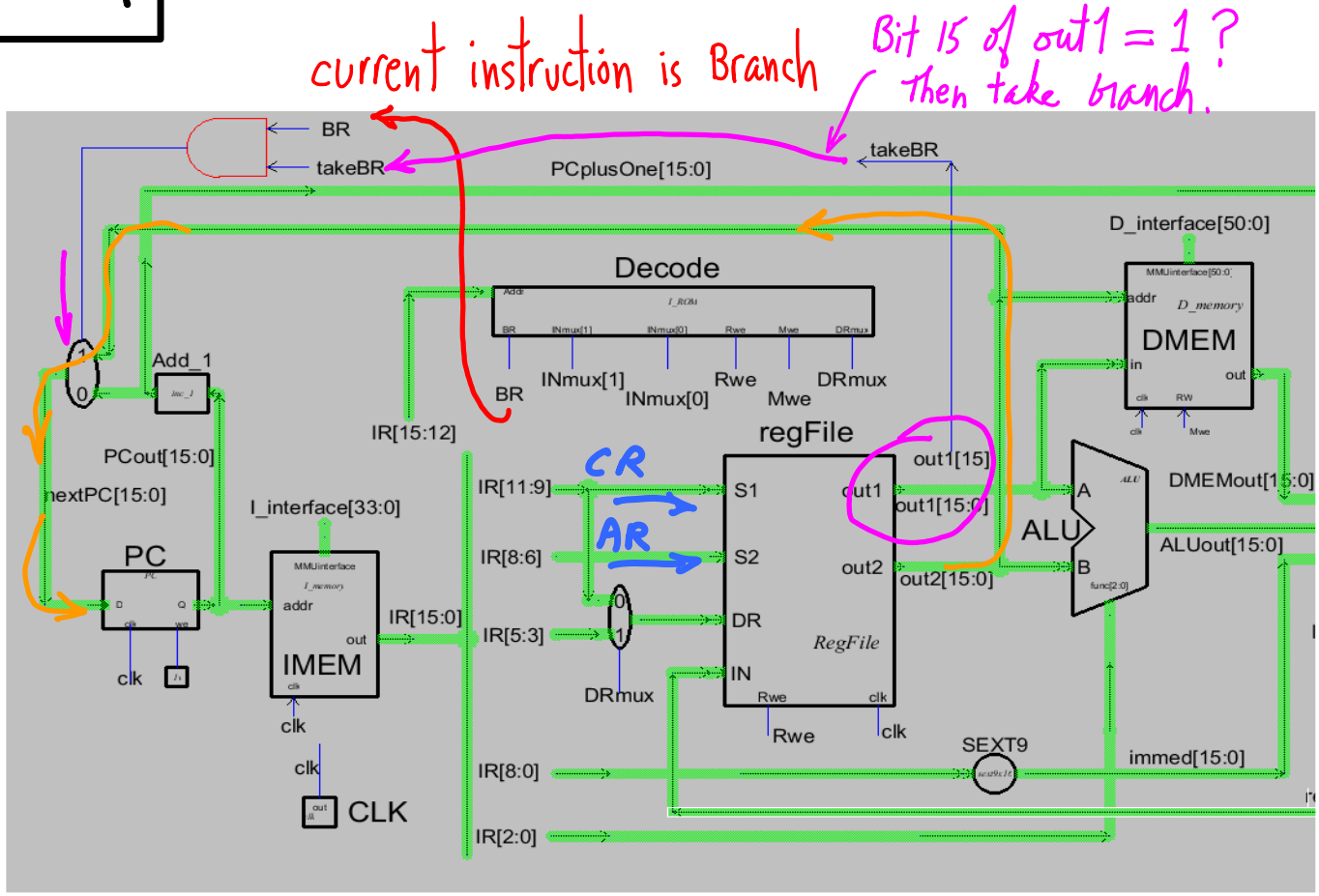
--- **same addressing modes** for branches

--- branch **condition**
== **function of symbols** that were read (think, Turing Machines)
==> **compare symbols** (**symbol1** == **symbol2 ?**)

--- What was the result of the comparison?
LC3 Branch Condition Codes (**CC**):
last value written to a register was,
Negative or **Zero** or **Positive**
PSR.N == 1 **PSR.Z == 1** **PSR.P == 1**

LC4 Branch

BRR CR1 AR2



Branch condition function is evaluated,

result goes into CR.

Address to branch to goes to AR.

Branches if CR < 0; else PC <=== PC+1

LC3, REMEMBER RESULT of FUNCTION EVALUTION

LD_CC

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

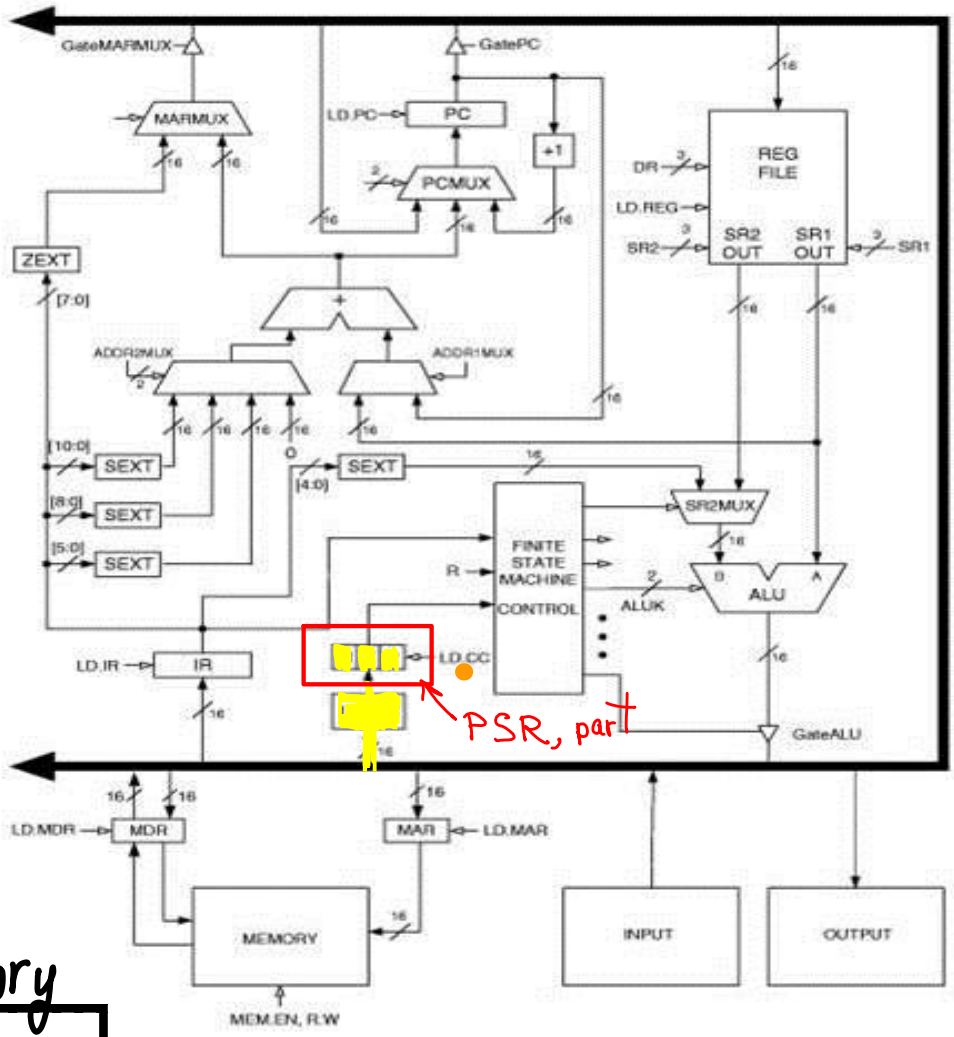
N = BUS[15] <was it negative?>
Z = NOR(BUS[15..0]) <was it zero?>
P = NOT(N)*NOT(Z) <was it positive?>

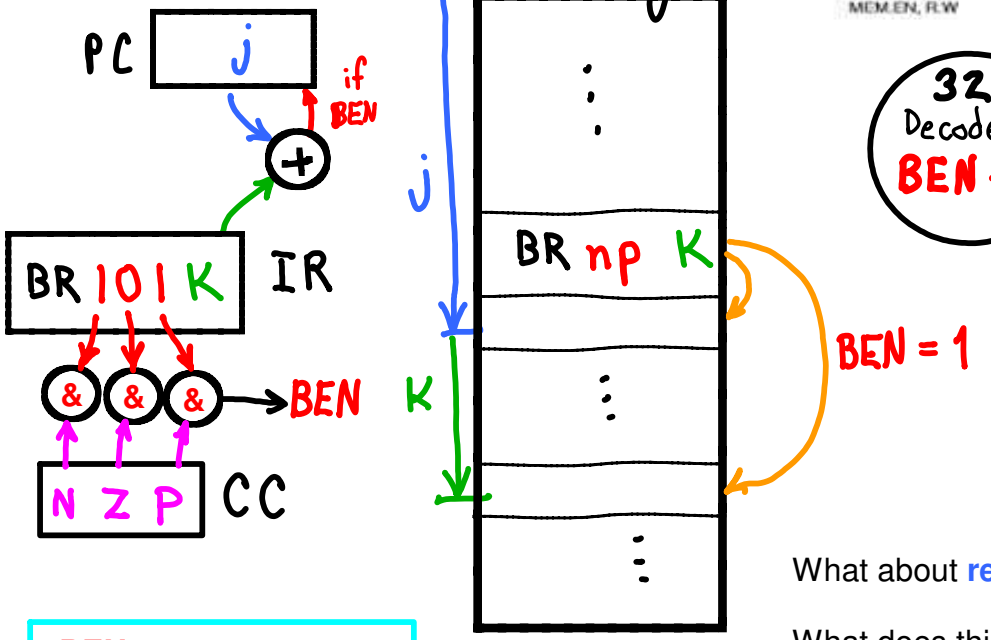
SAVE BRANCH CONDITION (State-**32**):

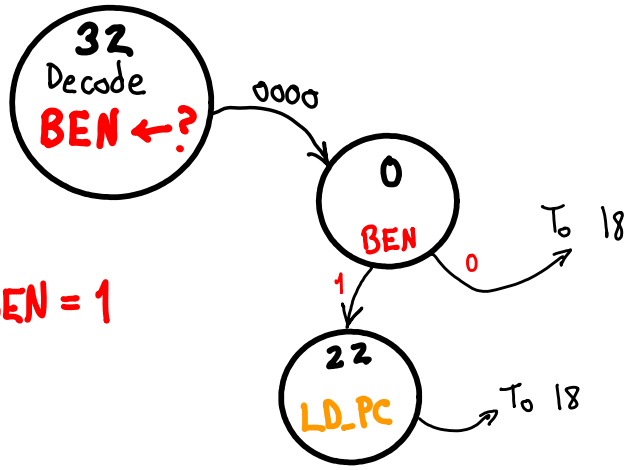
BEN <== **(CC** & **IR**[11:9] **)** && **(IR**[15:12] == **0000)**

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

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





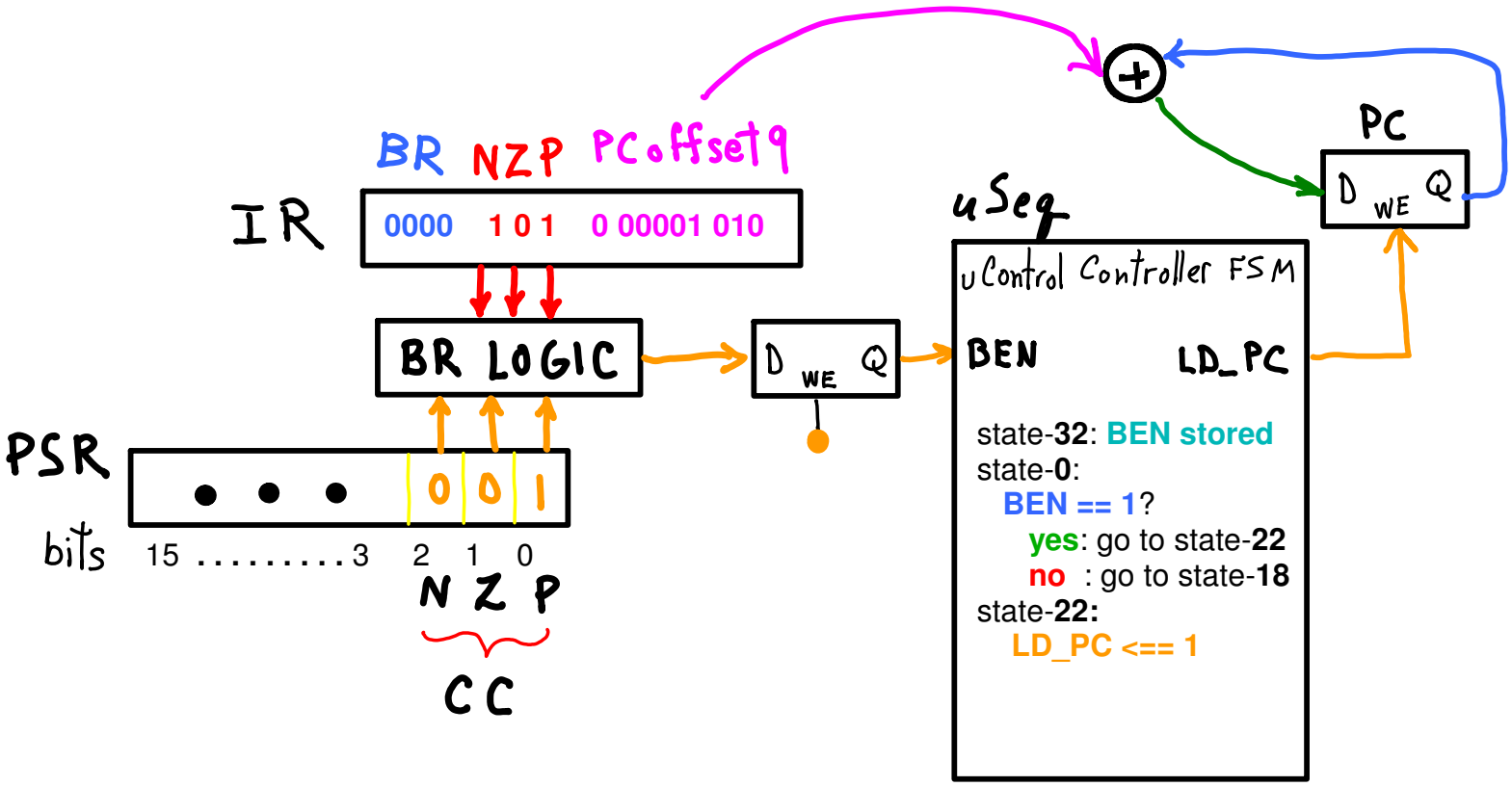


BEN <=== **1**
if
(**N** AND **n**) (BRn)
OR
(**Z** AND **z**) (BRz)
OR
(**P** AND **p**) (BRp)

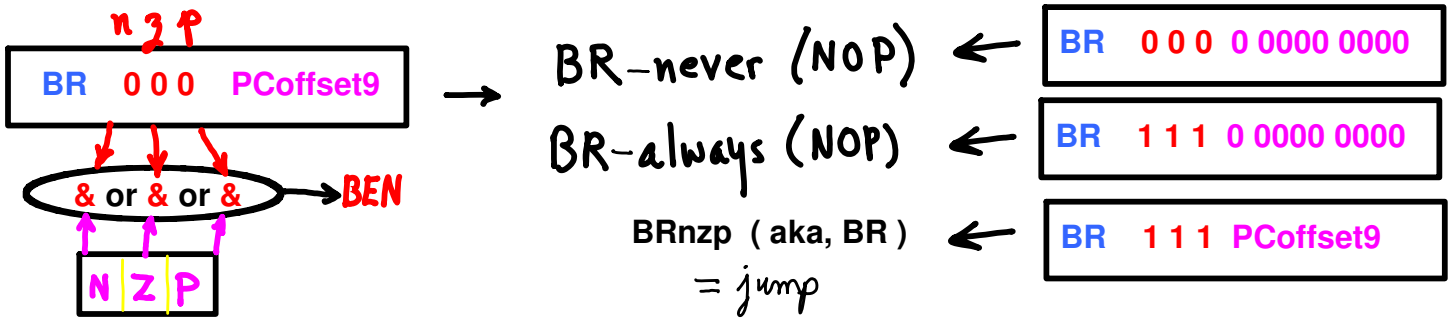
What about **remembering BEN**?

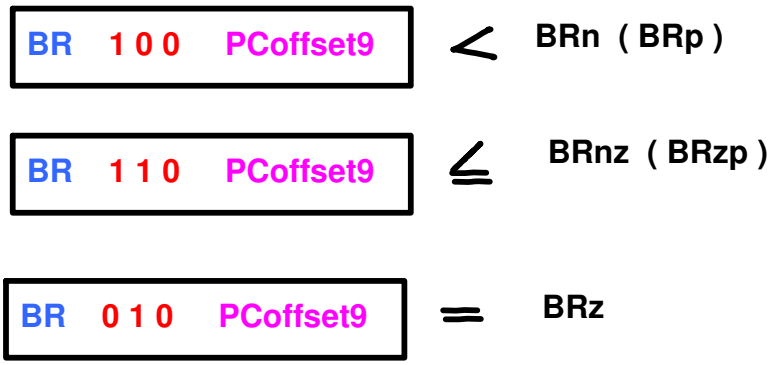
What does this instruction do?
0000 000 1 1111 1111
And this one?
0000 111 1 1111 1111

Range of BR?
The range of BR is limited (9 bit offset ~ 256 +-). We need to be able to jump anywhere (64k). We could reach anywhere w/ chained BRs. But we'd like another instruction that **jumps anywhere**.



What kind of branch decisions can we make?





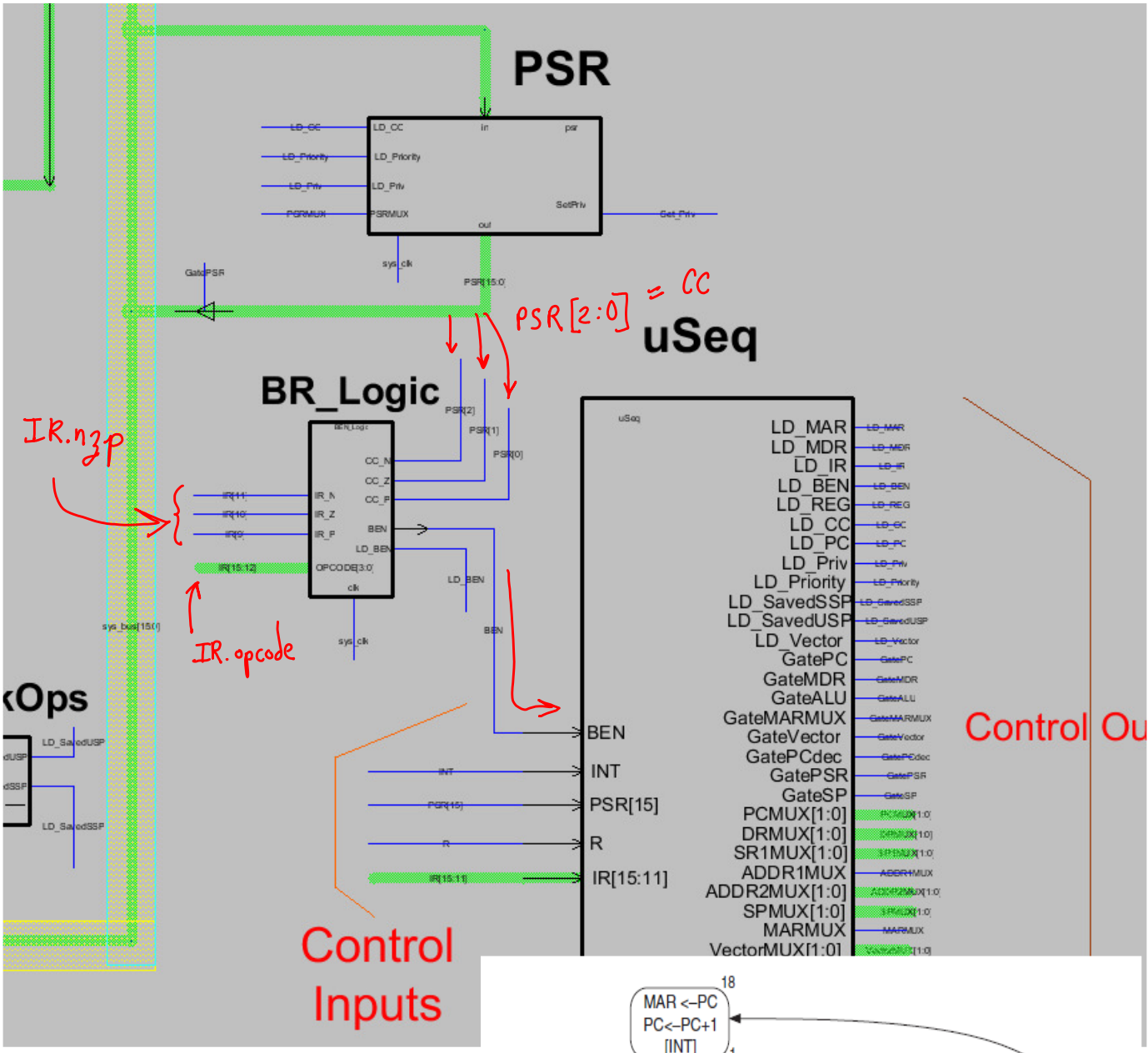
e.g.

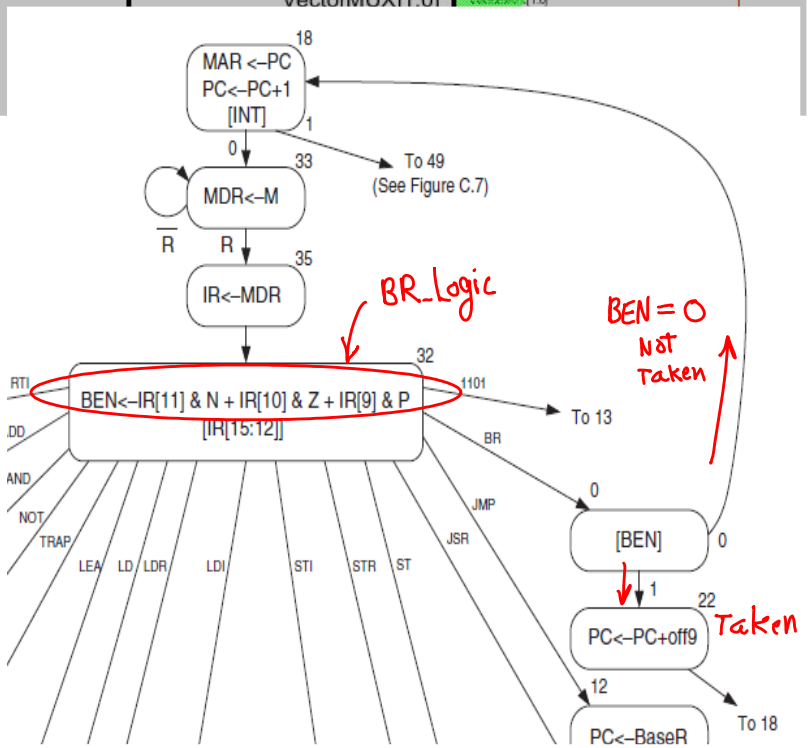
```
;-------------------------
;--- A in R1,  B in R2
;-------------------------
NOT  R3, R2          ;---   R3 <== -B
ADD  R3, R3, #1

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**

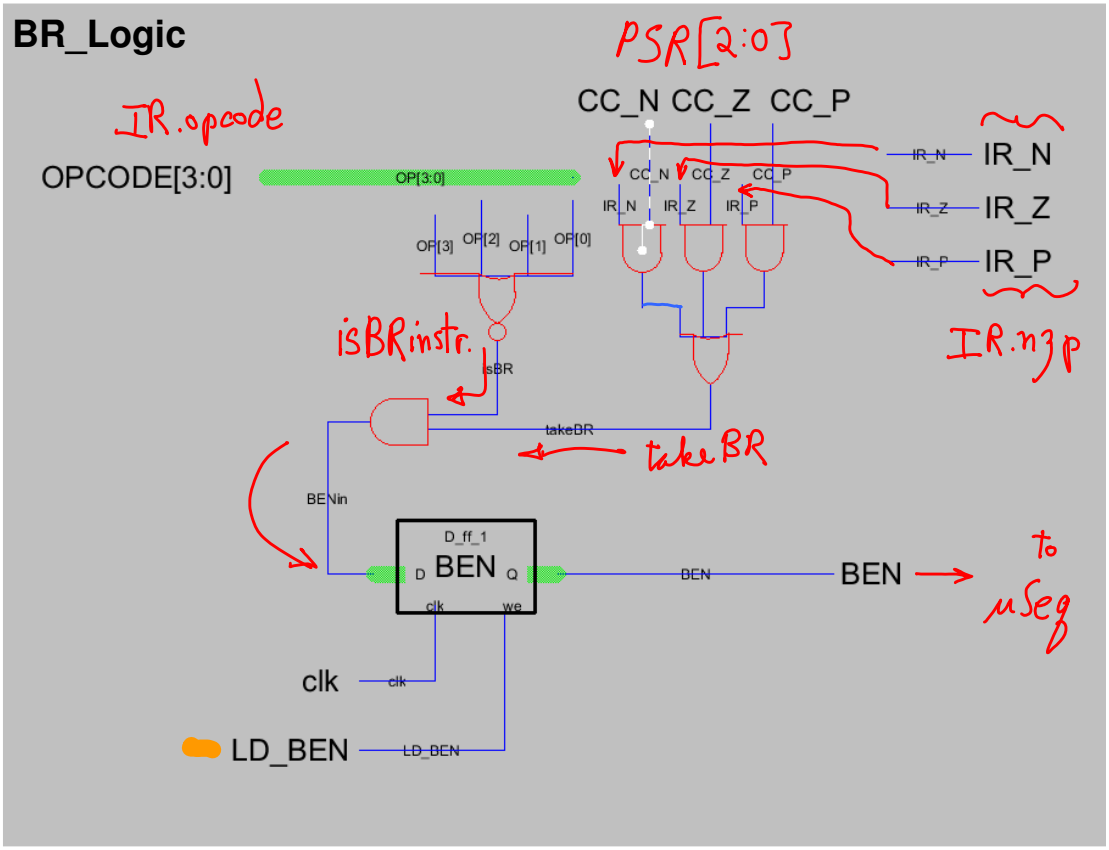
PSR[2 : 0] ==
{ CC_**N**, CC_**Z**, CC_**P** }

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**



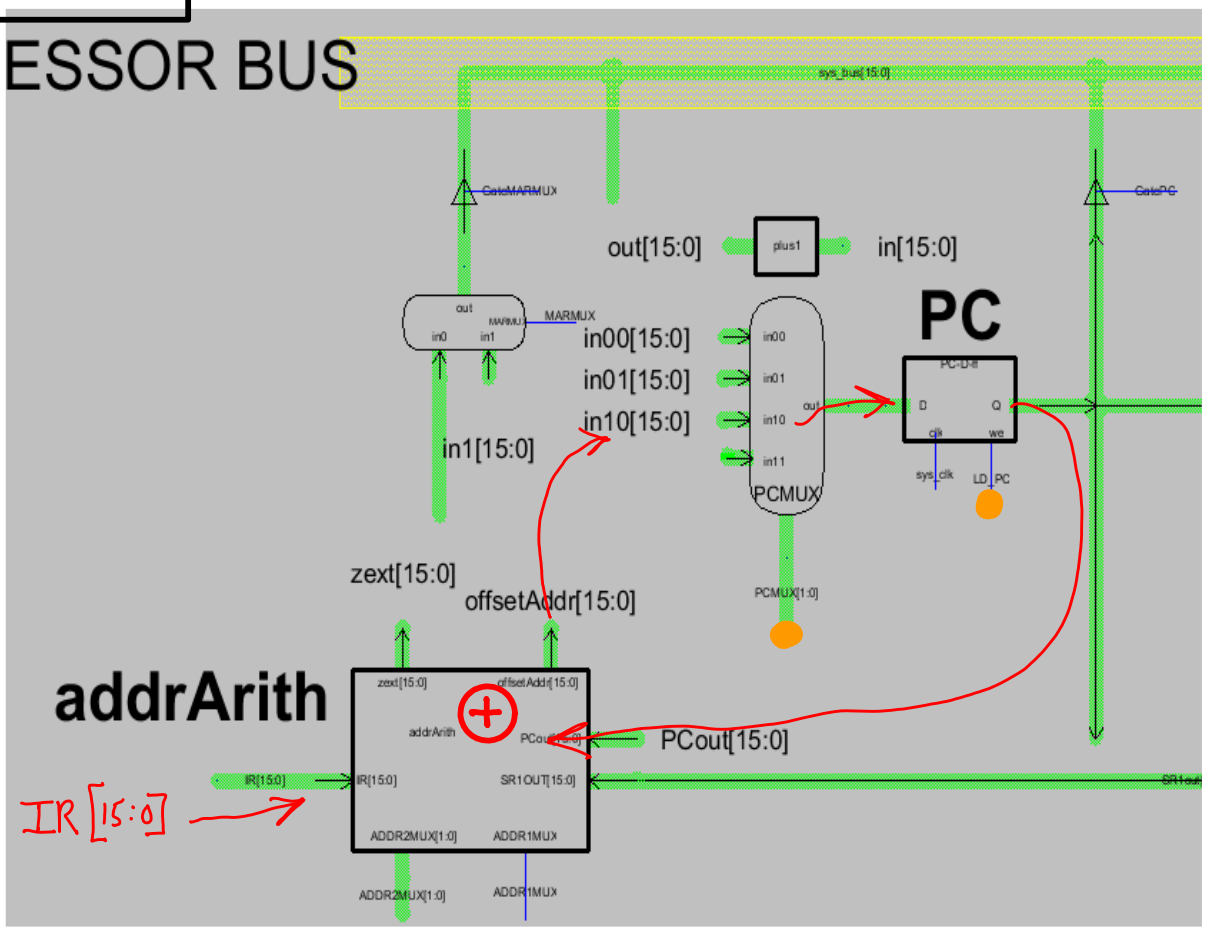
Taking the Branch

PC incremented in fetch-instruction phase.

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

Branch target address evaluated in **addrArith**.

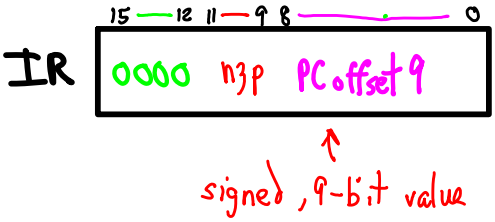
offsetAddr loaded to **PC**.

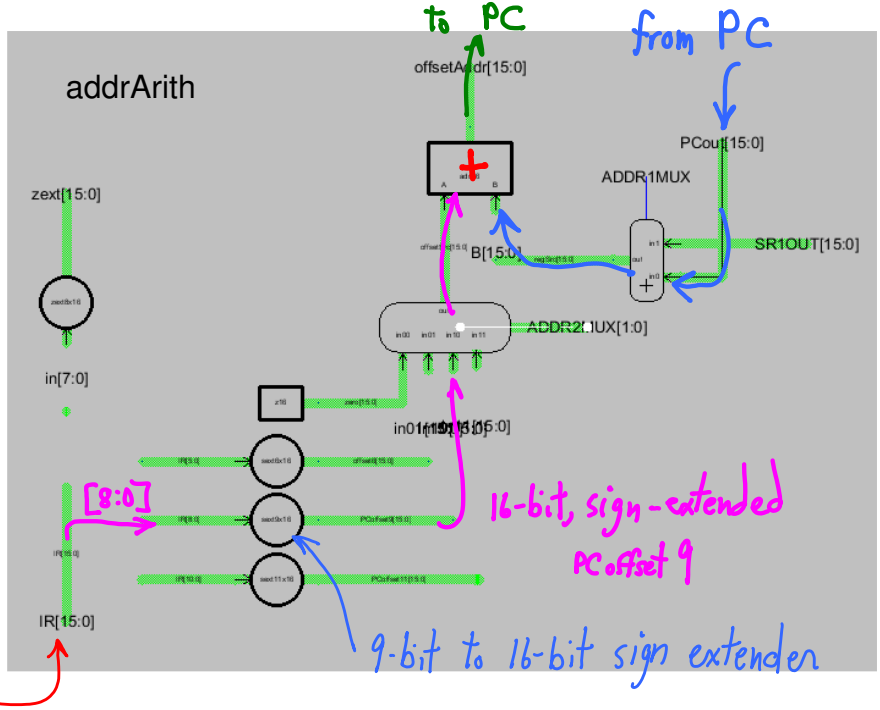


Calculating Target address

offsetAddr is

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

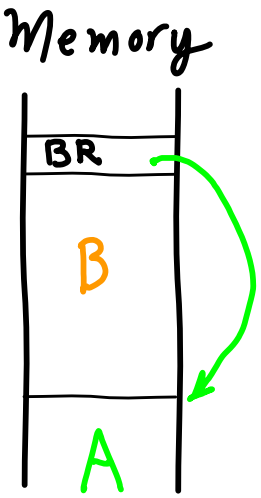




9-bit PCoffset9 ===> + or - (1/2) 2^9
about 2^8 range (256)

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.

LC4: How to
if (R1 == R2) (A in R0, B in R1)

R7 <=== (branch target address)

```
ALU SR0 SR1 DR2 SUB   //-- set R2 = A - B
BRR CR2 AR7           //-- branch if A < B)
ALU SR1 SR0 DR3 SUB   //-- set R2 = B - A
BRR CR2 AR7           //-- branch if B < A
 ...   (no branches, A == B)
```

But, we still need JMP

Jump via register

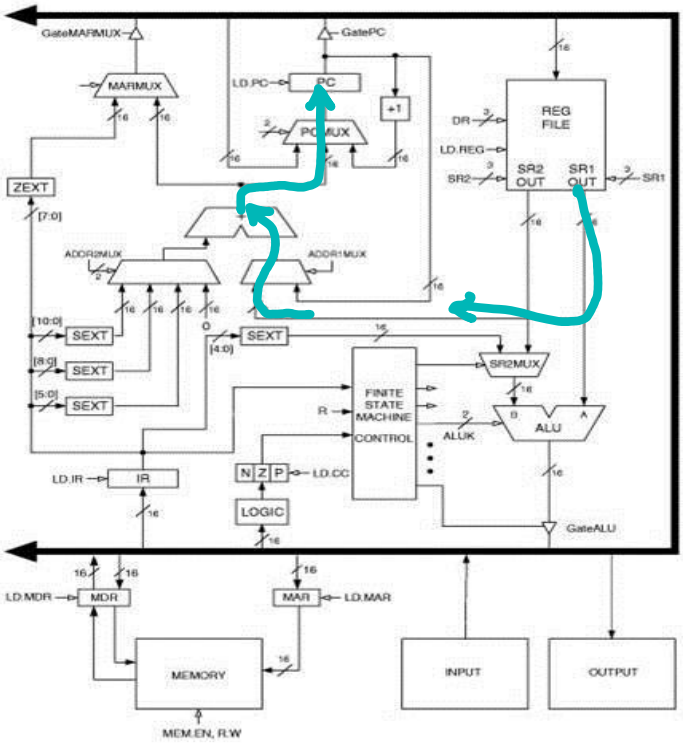
PC <= REGfile[SR]

Use any 16-bit address,
jump anywhere.

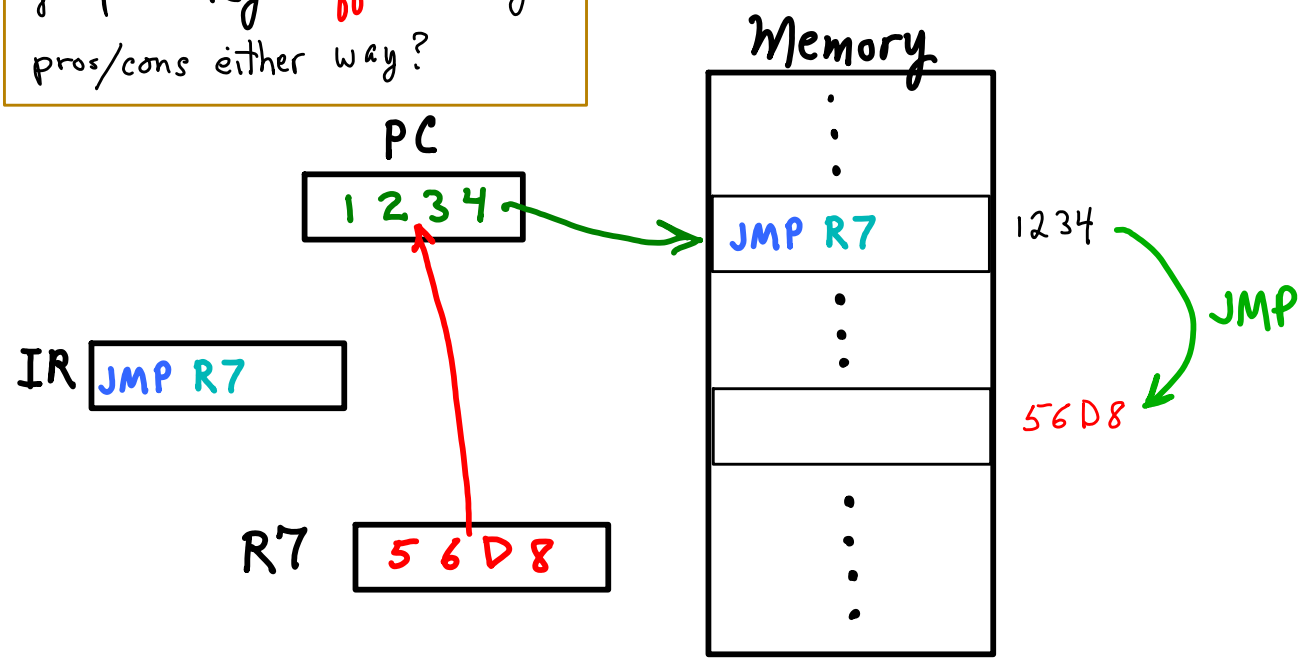
JMP SR

1100	---	111	------

Jump via reg



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



Jump in LC4:

R7 <=== target address
R2 <=== -1
BRR CR2 AR7

```
LIM DR7 h56                  #-- R7 <== 8 msb of address
(ALU SR7 SR7 DR7 ADD ) x 8   #-- 8-bit left-shift R7
LIM DR6 hD8                  #-- R6 <== 8 lsb of address
ALU SR6 SR7 DR7 ADD          #-- target (h56D8) ==> R7
ALU SR2 SR2 DR2 SUB          #--  0 ==> R2
ALU SR2 SR2 DR2 DEC          #-- -1 ==> R2
BRR CR2 AR7                  #-- jump R7
```

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

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**

--- **low-level abstraction**
== **sub-cell** (Electric)
== **function** (e.g., C)
--- **jump to** function
data to input ports
== arguments
--- **jump back** from function
data from output ports
== return values

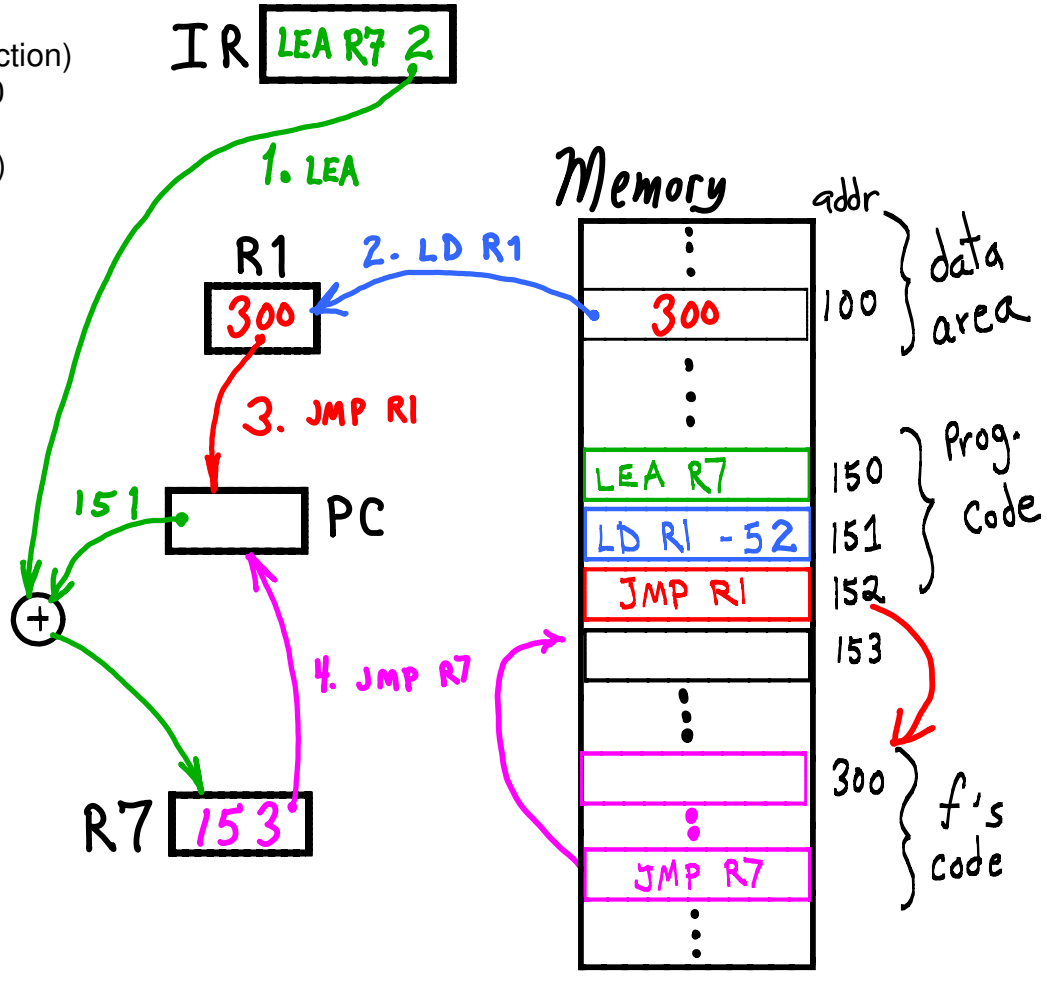
Funtion call and return:

1. **LEA R7, 2** //------- (**save BACK** addr)
// **R7** <== PC+2 == 151+2

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

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 **SR**

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

R7 <== PC as in (2) above
PC <== RegFile[SR **]** as in (3) above

JSR **PCoffset11**

0100	1	000 1000 0010

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

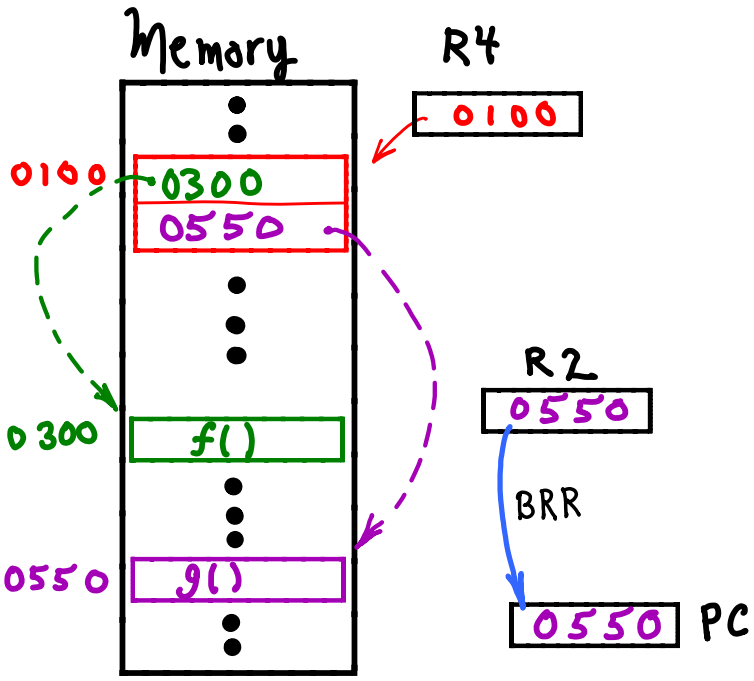
JMP R7
(aka, **RET**)

PC <== R7

general method for calling

a jump table

Set R4 to point to a table permanently.
Use jump table from any location in memory.
Full 16-bit addresses: jump anywhere.



LC4, jump to function using the table

First, we get the location of g's pointer:

```
LIM DR0 h1            #-- R0 <== g's offset
ALU SR0 SR4 DR0 ADD   #-- R0 + R4 ==> R0
```

Next we get g's address from the table:

```
LDR DR2 AR0           #-- R2 <== g's address
```

Finally, jump to g:

```
ALU SR0 SR0 DR0 SUB   #--  0 ==> R0
ALU SR0 SR0 DR0 DEC   #-- -1 ==> R0
BRR CR0 AR2
```

Note, we didn't set R7; we cannot jump back. See below.

LC3, function call using the table

First, we evaluate the address of g's pointer, and load from that address into R2:

```
LDR R2, R4, #1 ;-- R2 <== MEM[ R4 + g's offset ]
```

Then we jump, setting R7 to the return address:

```
JSRR R2          ;-- R7 <== PC; PC <== R2
```

JSR in LC4

Let's use a Global Data Table as part of our program. It holds addresses and constants our program needs. We put the address of the function in the table.

Define the Global Data Table (GDT):

```
.ORIG h0100
#-- This is our Global Data Table
.FILL d0       # 1st thing in table
.FILL d0       # 2nd thing in table
.FILL h3000    # function's address
```

Initialize Global Data Pointer (GDP):
R4 is our GDP. Program starts at 0200.

```
.ORIG h01ff
.FILL h0100           # pointer to GDT
LEA DR0               # R0 <=== PC
ALU SR0 SR0 DR0 DEC   # R0--
ALU SR0 SR0 DR0 DEC   # R0--
LDR DR4 AR0           # R4 <=== pointer
```

Our GDP is now ready to use.

To jump to a function, we need to: a) get its address; b) put the return address into R7; c) make the jump. We did (a) and (c) above.

LC4, function call using GDT

```
#-- Get address of function into R3:
LIM DR0 d2            # R0 <=== offset 2
ALU SR0 SR4 DR0 ADD   # R0 + GDP ==> R0
LDR DR3 AR0           # R3 <=== address

#-- Set R7 w/ return address
LEA DR7               # R7 <=== PC
LIM DR0 d4            # R0 <=== 4
ALU SR0 SR7 DR7 ADD   # R0 + R7 ===> R7

#-- Jump to function
ALU SR0 SR0 DR0 SUB   #--  0 ===> R0
ALU SR0 SR0 DR0 DEC   #-- -1 ===> R0
BRR CR0 AR3           #-- jump to function
```

The constant, 4, is not obvious until we have all the code so we can see where to return to.

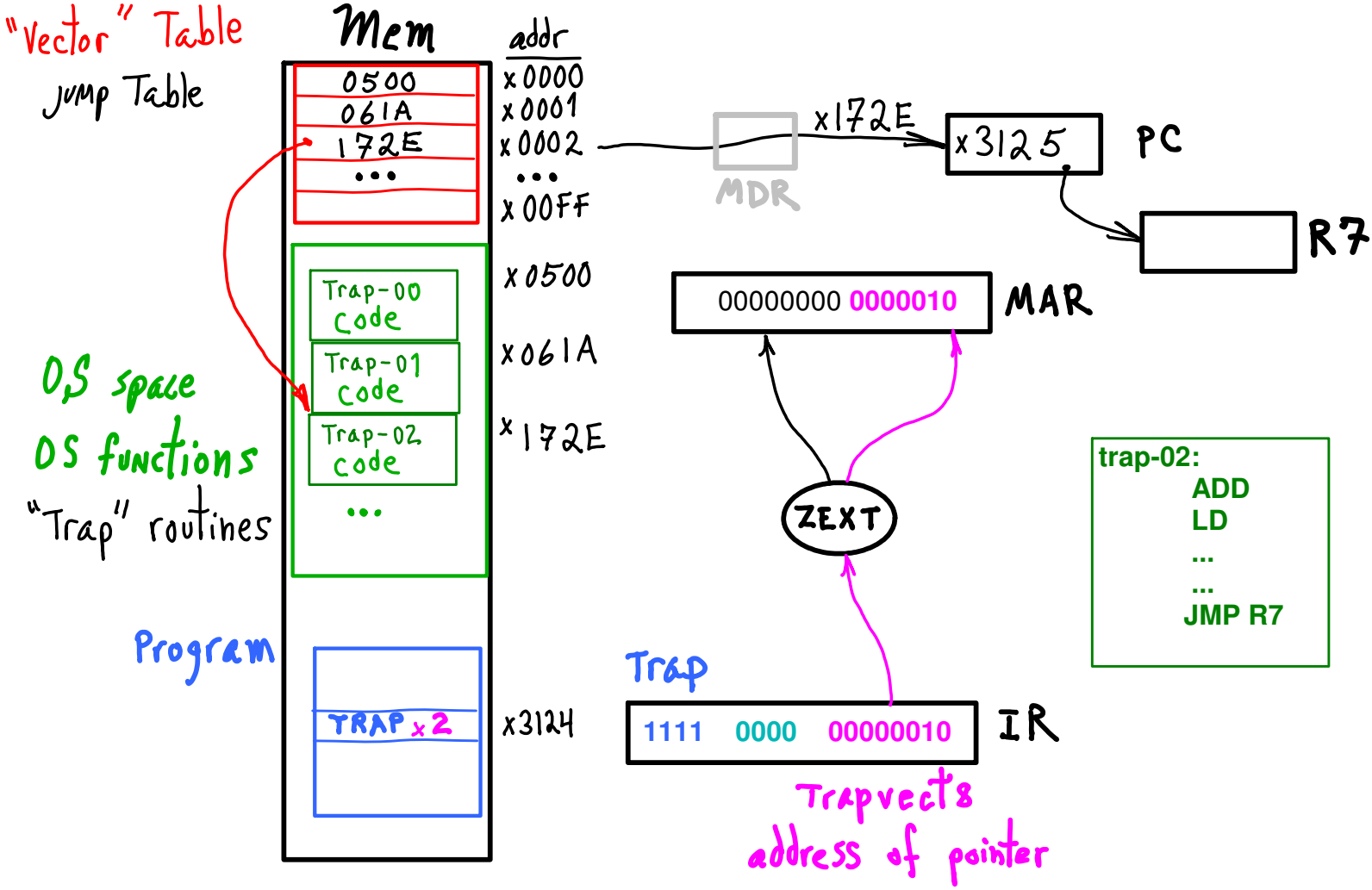
We load our function at address h3000. All this function does is return using the address in R7.

```
.ORIG h3000
#-- Jump back via R7
ALU SR0 SR0 DR0 SUB   #  0 ===> R0
ALU SR0 SR0 DR0 DEC   # -1 ===> R0
BRR CR0 AR7           # jump R7
```

Jump Table w/o GDP

OS provides many functions programs can call.

LC3:
1. put all the pointers in a table.
2. provide access via **special instruction**.




```
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?

--- PROGRAM INDEPENDENCE

jump via STANDARD VECTORS

--- OS convention (see OS manual)

Write **VT** **at boot time, functions are relocatable**.

--- OS FUNCTION CALL

--- OS provides services

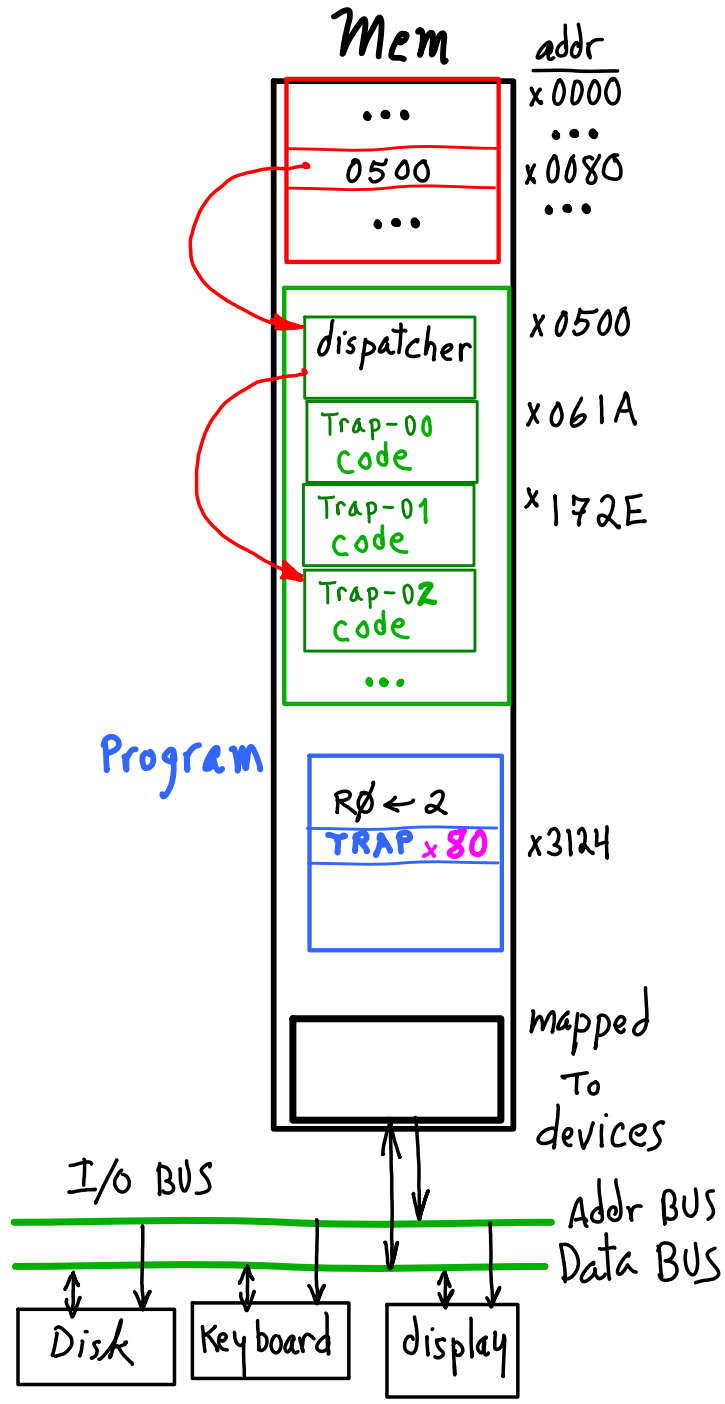
Programs **never need to know details**.

===> get arguments, return results?

registers, memory, stack (more later).

--- **Trap Vector Table** (**VT**)
8-bit index ==> 256 entries [x0000 -- x00FF]
256 OS functions (OS entries)

--- **Linux** uses **VT vector 80 for all entries to OS**
Use R0 for function code
32-bit register ==> 4G functions possible.



--- **OS services**, e.g.,
I/O via device registers:
LD/STR and **memory addresses**
OS contains all "driver" code

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

TRAP (**function call**)

State-**15**:

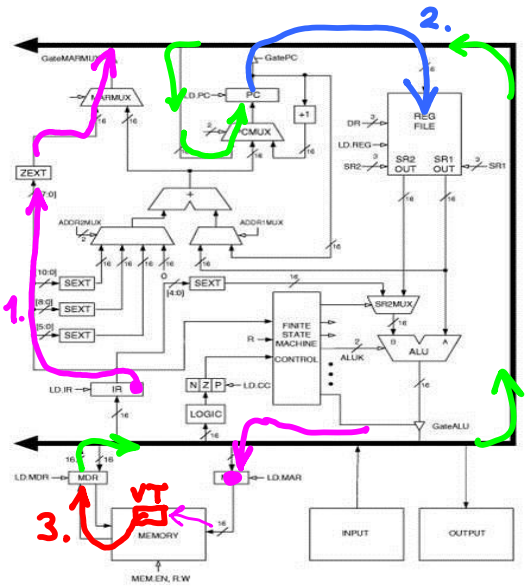
MAR <== ZEXT(**IR[** 7 : 0 **]**) //--- get f()'s VT entry's address (1)

State-**28**:

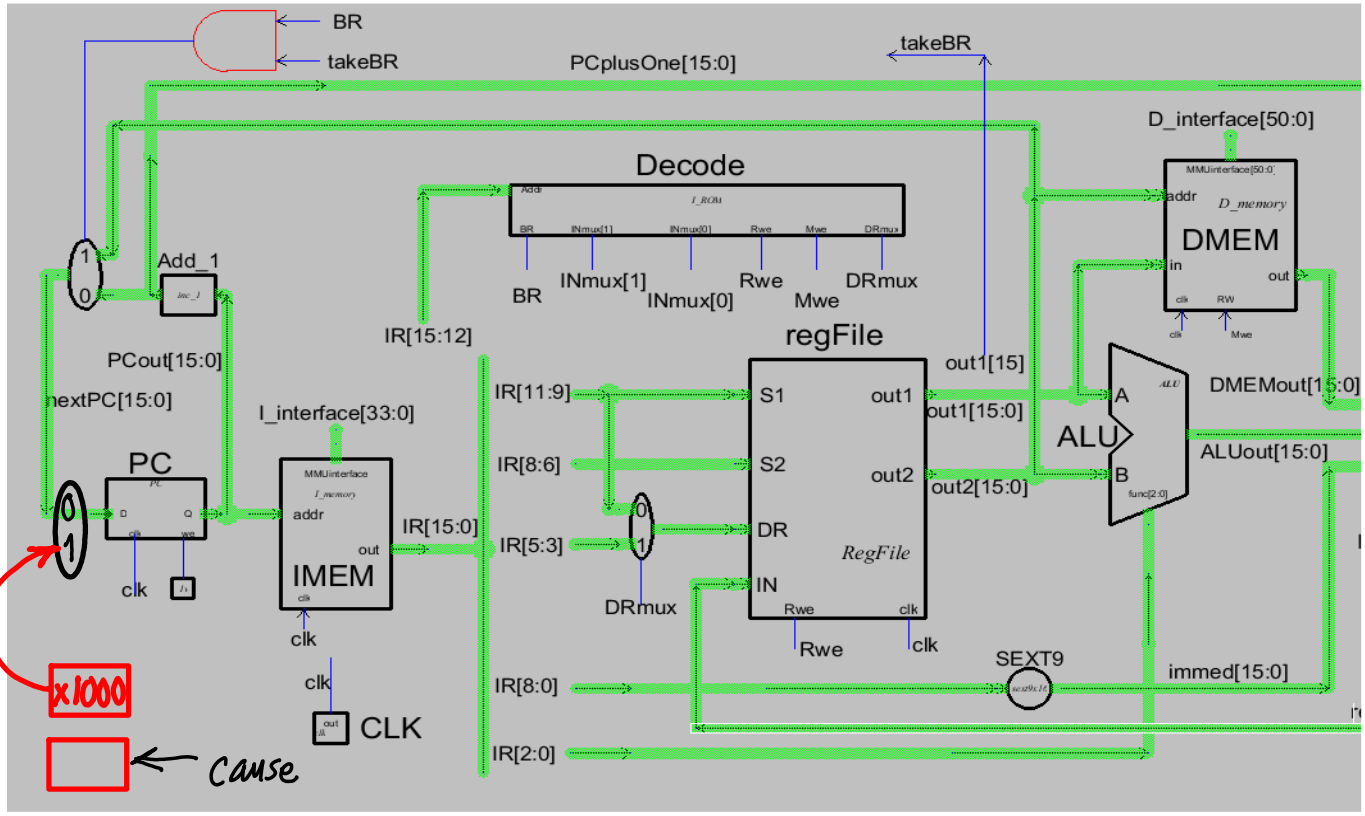
R7 <== **PC** //--- save "return" address (2)
MDR <== **MEM** //--- get f()'s address from VT (3)

State-**30**:

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



LC4 syscalls, interrupts ?



Add a MUX to PC

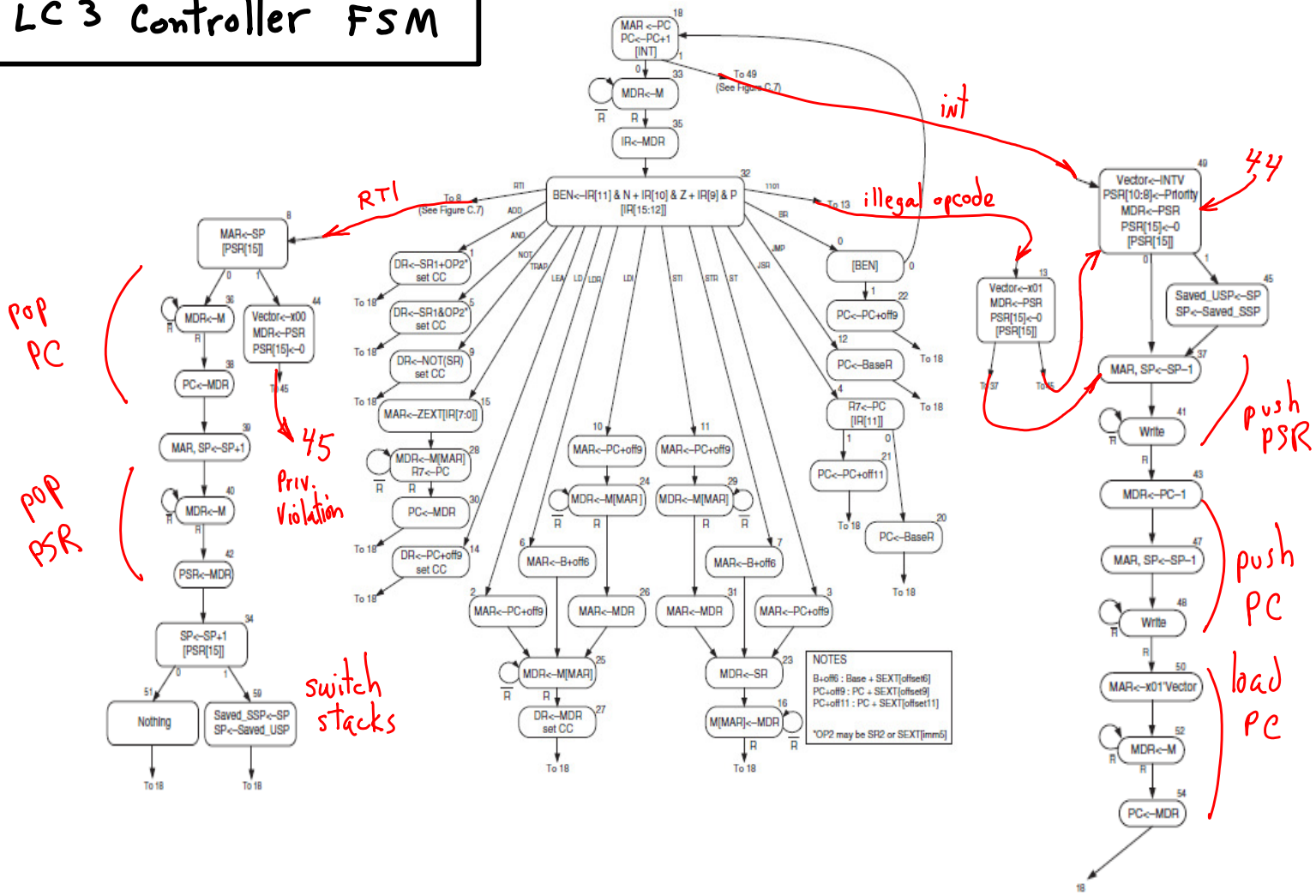
Set MUX → jump to x1000, dispatcher

Why? "cause" stored in cause-register, for interrupts

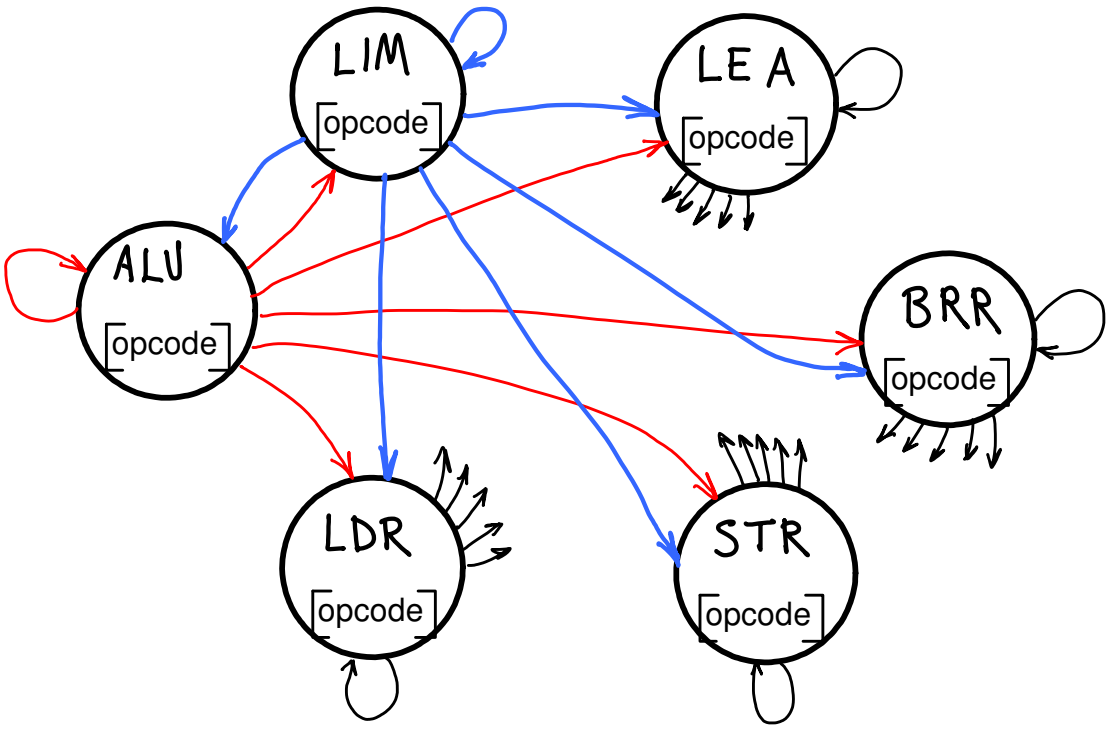
function number in R0, for syscalls.

Add new instruction to access cause reg.

LC 3 Controller FSM



LC4 Controller FSM



Each state executes one instruction.

Each state branches depending on new opcode.

Actually, there are 10 more states, all are Illegal-opcode. One for each unused 4-bit opcode.