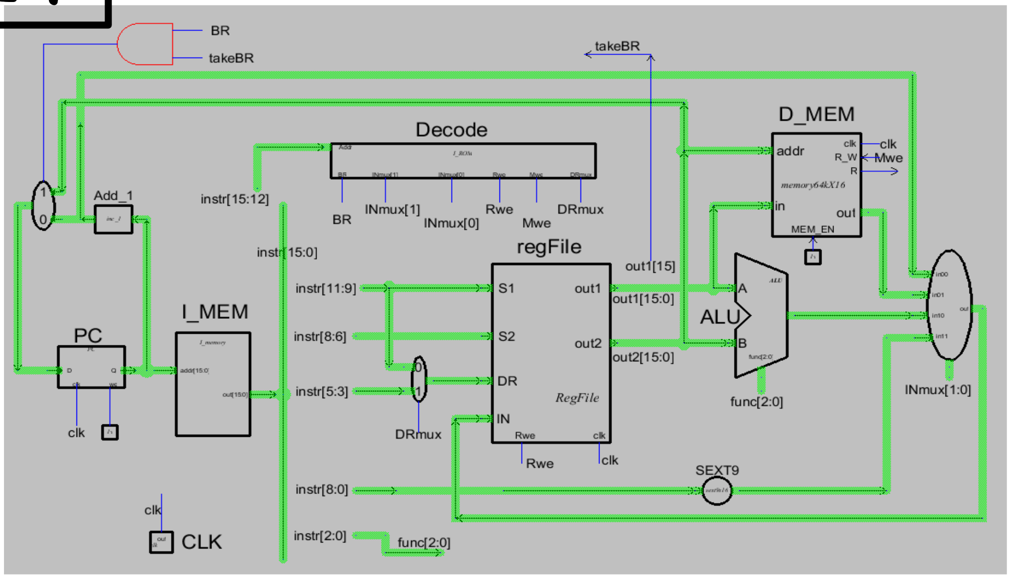
LC3 & LC4 ISA, micro-Architecture, and Programming

LC4



Controller:
6 states
5 control signals
PC

Instruction execution:
one cycle

clock:
2.5 (memory delay)

Word Size	**Registers**	**Memory (split)**	**Instructions**
16 bits	8 @ 16 bits	16-bit addresses	ALU, LIM, LDR, STR, LEA, BRR
16-bit ALU	R0 ... R7	word addressable	**ALU operations:**
	PC	**two address modes:**	ADD, SUB, AND, iOR, NOT, NOR, INC, DEC
		register-base, immediate	

LC3

Word Size: 16-bit

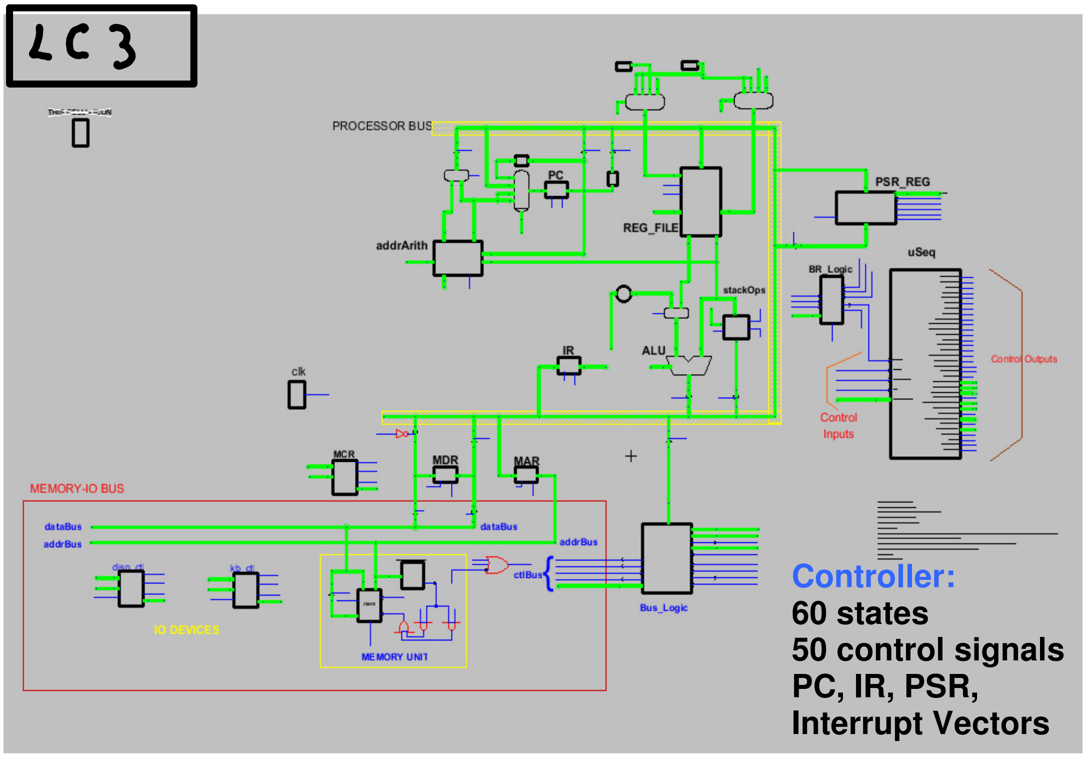
Registers: 8 @ 16-bits +
PC, IR, MAR, MDR, PSR,
interrupt VECT

Memory (unified): 16-bit word/address/addressable

13 instructions:
LD, ST, LDI, STI, LDR, STR,
ALU, LEA, BR, JMP,
JSR, JSSR, TRAP
ALU ops: ADD, NAND, NOT
BR ops: == , != , < , > ,
<= , >=

4 addressing modes:
PC-relative, PC-indirect,
register-relative, immediate

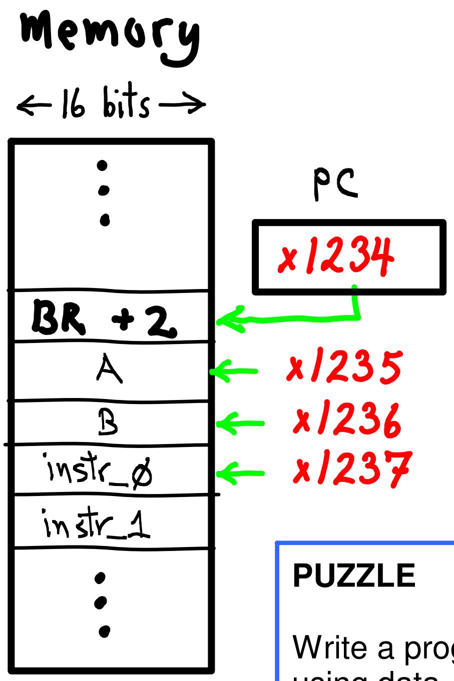
I/O: keyboard, display



Controller:
60 states
50 control signals
PC, IR, PSR,
Interrupt Vectors

Instruction execution: 4 to 10 states + (1 or 2 memory delays)
clock: 0.1 (memory delay)

MEMORY



ADDRESS SPACE:
16'h0000 to 16'hFFFF (16-bit MAR)

ACCESS OPERATIONS:
read or write one word (instruction or data)

WORD SIZE:
16-bit (16-bit MDR)

ADDRESSABILITY:
1-word (2-byte)
access is at 16-bit boundaries
(most machines have byte addressability)

PUZZLE

Write a program in LC3 or LC4 machine code which alters its first two instructions using data A and B:

(1) **instr_0.**opcode <== **instr_0.**opcode + **A** (only opcode is altered)
(2) **instr_1.**opcode <== **instr_1.**opcode + **B** (likewise)

Program is **independent of program's location** in memory.
PC points to the BR instruction.
Data **A and B are in consecutive words** of memory.
Below B are more instructions.

state-by-state operations (notation and display)

RTL (Register Transfer Language)
NON-ZERO control signals

MAR <== **PC** (**PC** content copied to **MAR** on next clock tick)
LD_MAR (**LD_MAR** == 1 , all other controls == 0)

Wire paths

IR[15:12]--->**FSM**.opcode (**IR**'s upper 4 bits connect to controller, **FSM**.opcode)

LC3 verilog testbench display (see projects/LC3trunk/lib/test.jelib)

```
-------------------------------------( time )---------------------------------------------------------------------
------((( FSM state )))------[ non-zero control signals ]-----[ non-zero MUX controls ]---------

-------------------------------------( 3 )-------------------------
-------((( 18 )))-------[ LD_MAR ]---------[ ]---------------
```

LC3

LC3 fetch instruction:

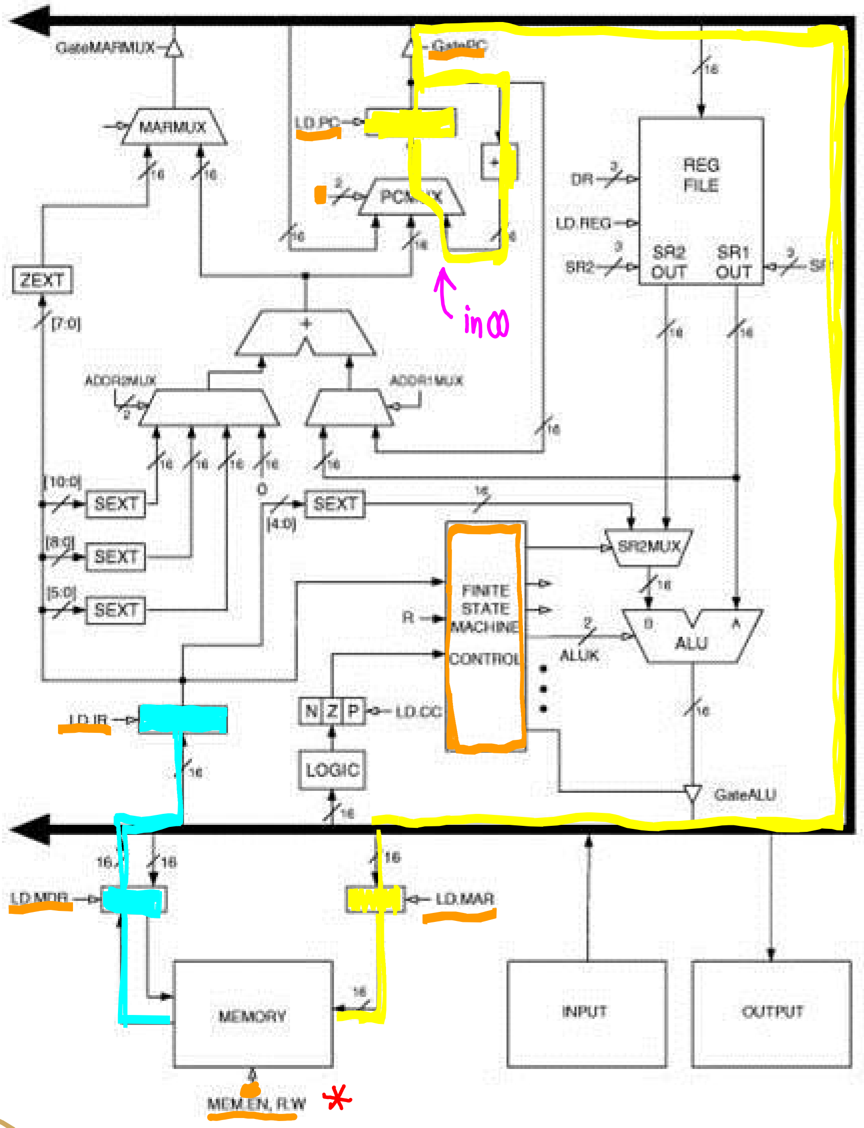
State 18:
MAR <== **PC**
GatePC
LD_MAR

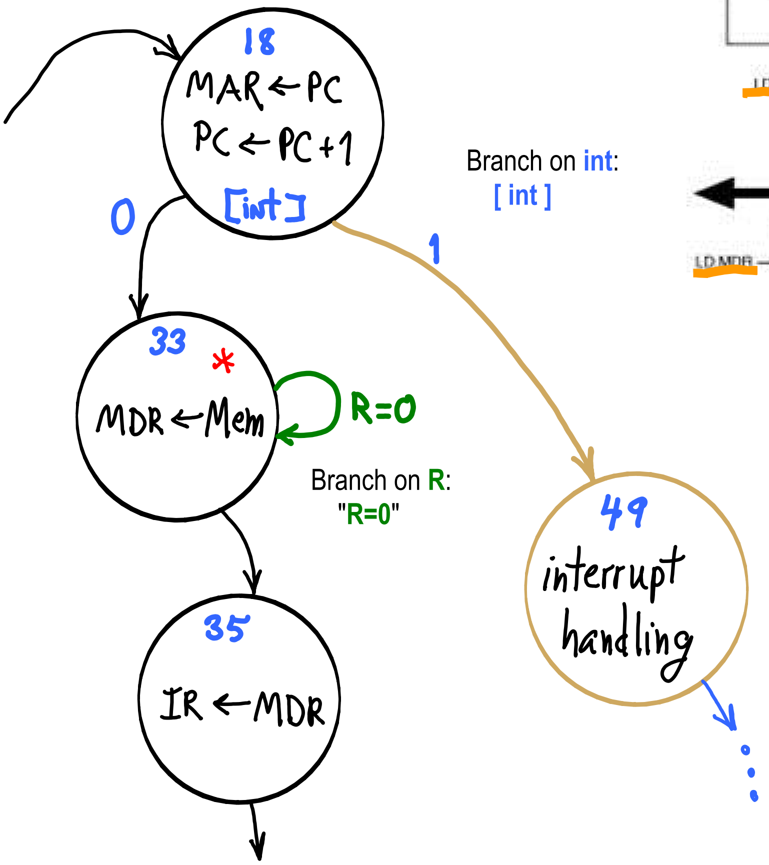
PC <== **PC** + 1
PCMUX = 00 (select)
LD_PC

State 33:
MDR <== **MEM**.out
LD_MDR
MIO_EN *
R_W = 0

State 35:
IR <== **MDR**
LD_IR *

*See Tri-states in MEMORY-IO BUS





Branch on **int**:
[int]

Branch on **R**:
"**R=0**"

To 32, Decode

(See App. C)

mio BUS:
— databus
— address bus
— control bus

in Electric, LC3

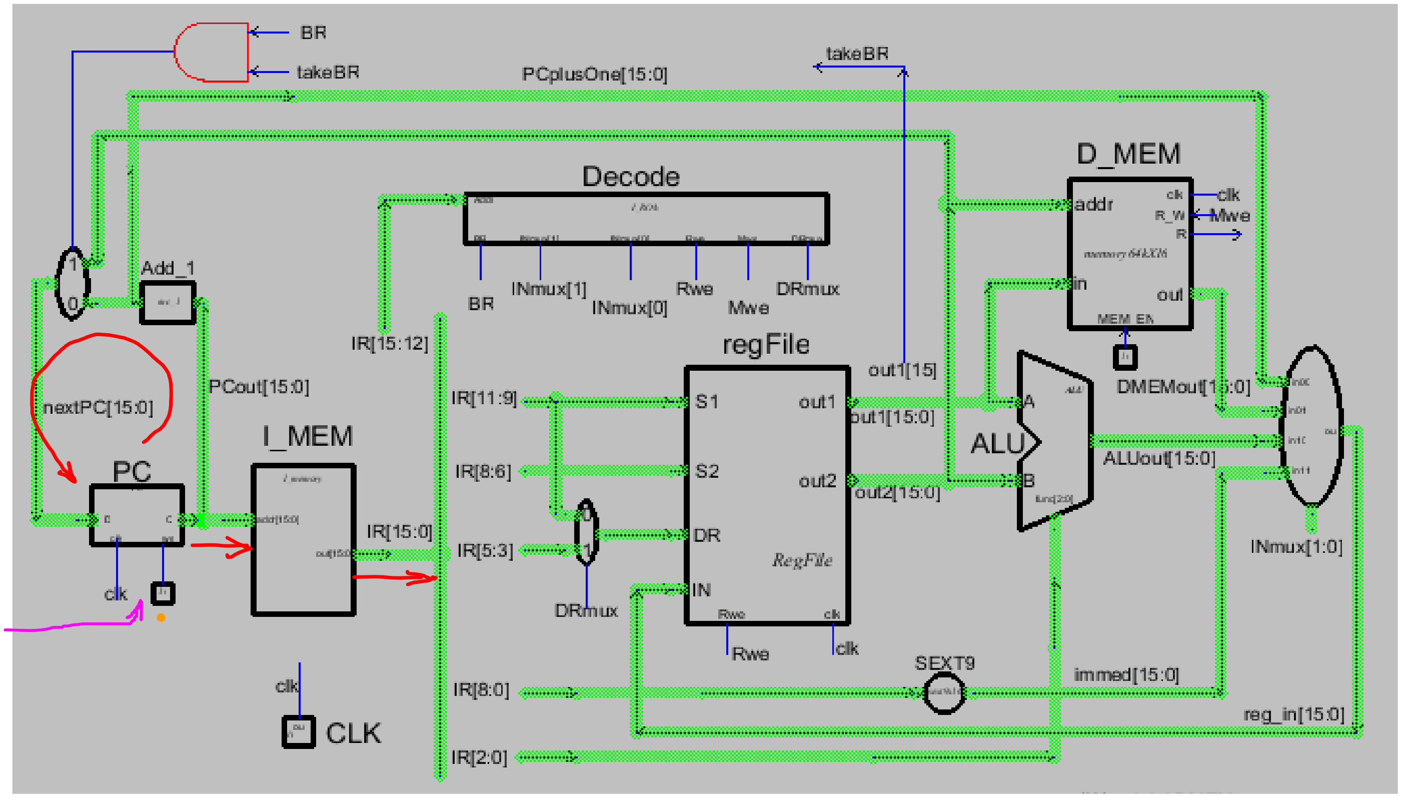
1. See **top.Mem-IO-Bus**

    --- address decode
    --- tri-states
    --- control bus

2. See **test.testInstr**

    --- initializing memory

LC4, fetch instruction



always enabled

PC <== PC+1 (not until next clock)
PC ---> **I_MEM**.addr
I_MEM.out ---> bus **IR**[15:0]

LC4 instruction format, ADD:

R2 + R3 → R6

0000	010	011	110	000
ALU	**R2**	**R3**	**R6**	**ADD**

address from **PC**.out
to **I_MEM**.addr;
word from **I_MEM**.out is
word "on" the **bus IR**[15:0]:

IR[15:0] == 16'b**0000010011110000**

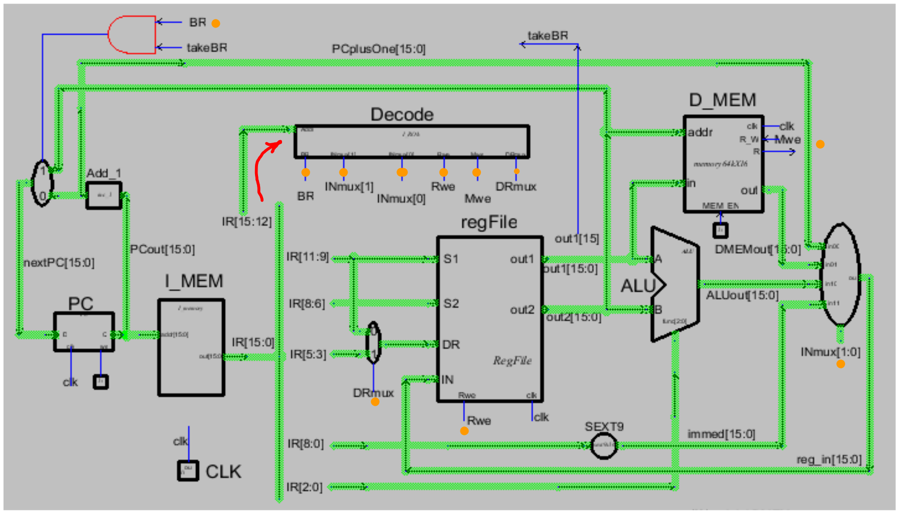
Opcode	S1	S2	DR	FUN
bits : 15...12	11...9	8...6	5...3	2...0

Decode:

IR[15:12] **---> Decode**.opcode[3:0]

Decode output (control signals)

BR	(instruction is BR)
INmux[1:0]	(steer data to regFile)
Rwe	(regFile write-enable)
Mwe	(D_MEM write-enable)
DRmux	(select IR field for DR)



Operand Fetch:

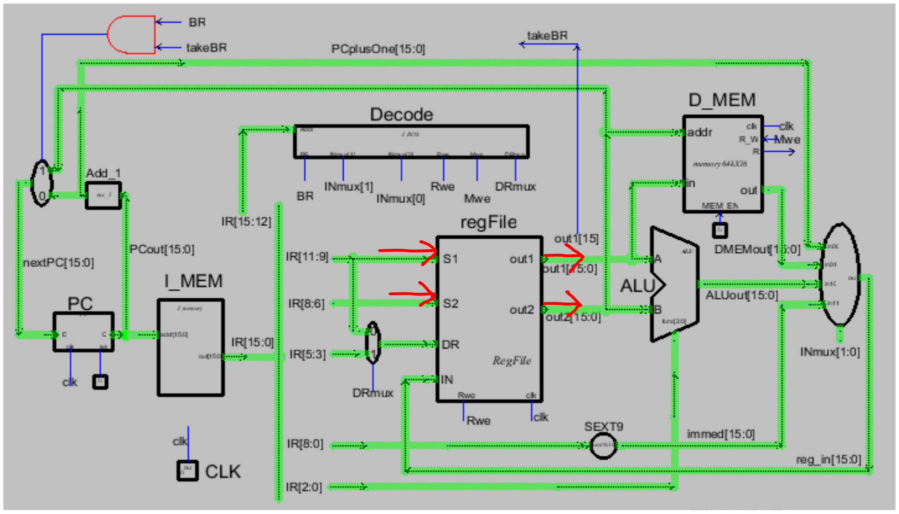
(Select which registers to get data from, controls regFile's output muxes:)

IR[11:9] ---> **regFile**.S1[2:0]
IR[8:7] ---> **regFile**.S2[2:0]

(Get data from registers:)

regFile.out1 ---> **out1**[15:0]
regFile.out1 ---> **out2**[15:0]

(**out1** and **out2** are buses)



Execute:

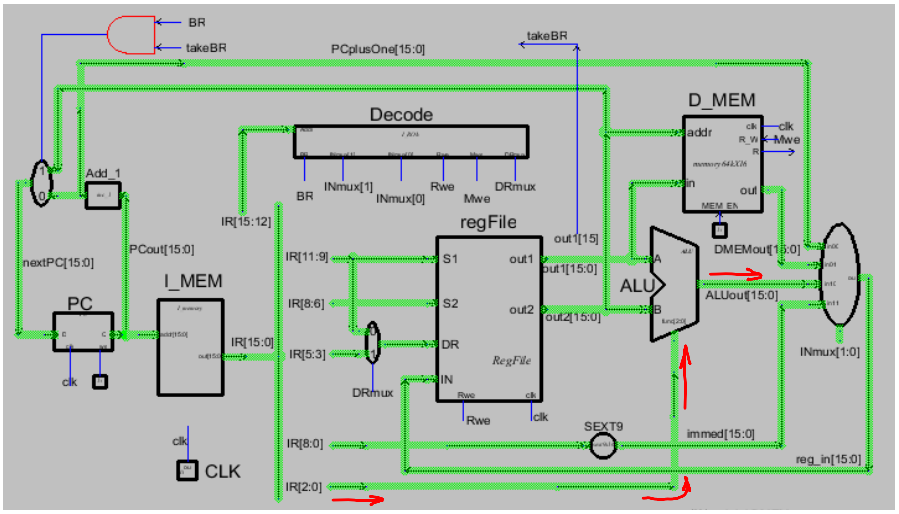
(Select which ALU operation, controls ALU's output mux:)

IR[2:0] **---> ALU**.func[2:0]

(Get result from ALU:)

ALU.out ---> **ALUout**[15:0]

(**ALUout** is a bus.)



Store Result:

(Select ALU's output to send to IN:)

INmux[2:0] **---> INmux.**select[2:0]

(Select which register to write:)

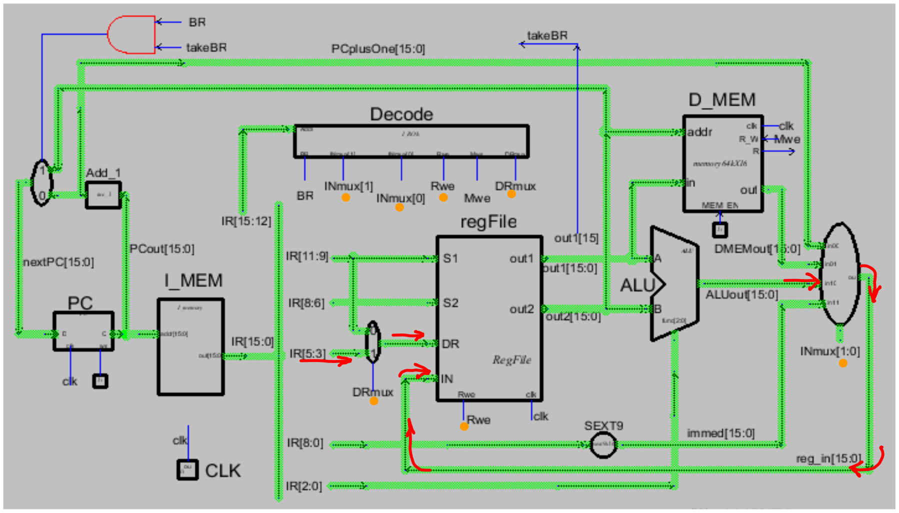
DRmux ---> **regFile**.DR[2:0]

(Send data to register:)

reg_in[15:0] ---> **regFile**.IN[15:0]

(Write data to register on next clock tick:)

Rwe ---> **regFile**.Rwe



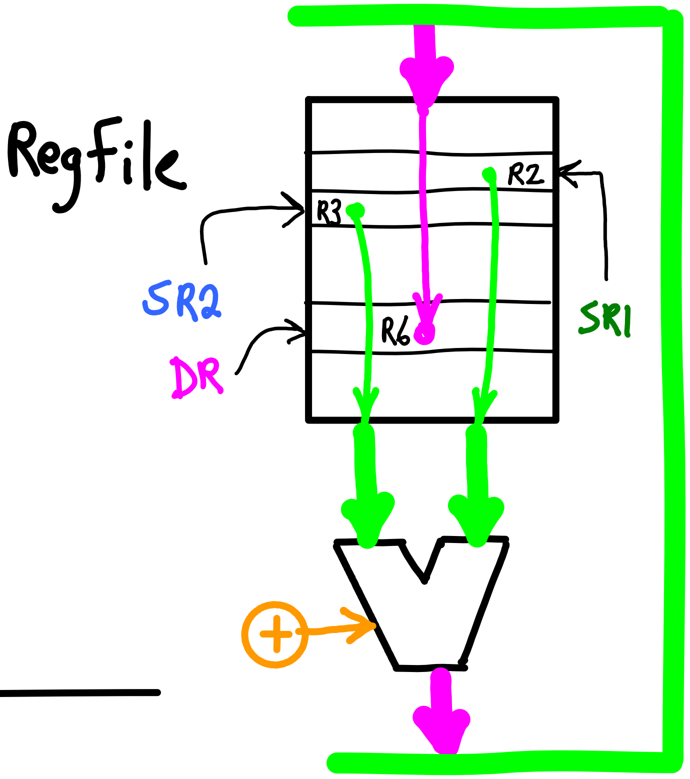
LC3 Instruction Formats

LC3, ADD: R6 ← R2 + R3

0001	110	010	000	011
ADD	**R6**	**R2**		**R3**

Opcode	**DR**	**SR1**	...	**SR2**
15...12	11...9	8...6	5...3	2...0

bits:



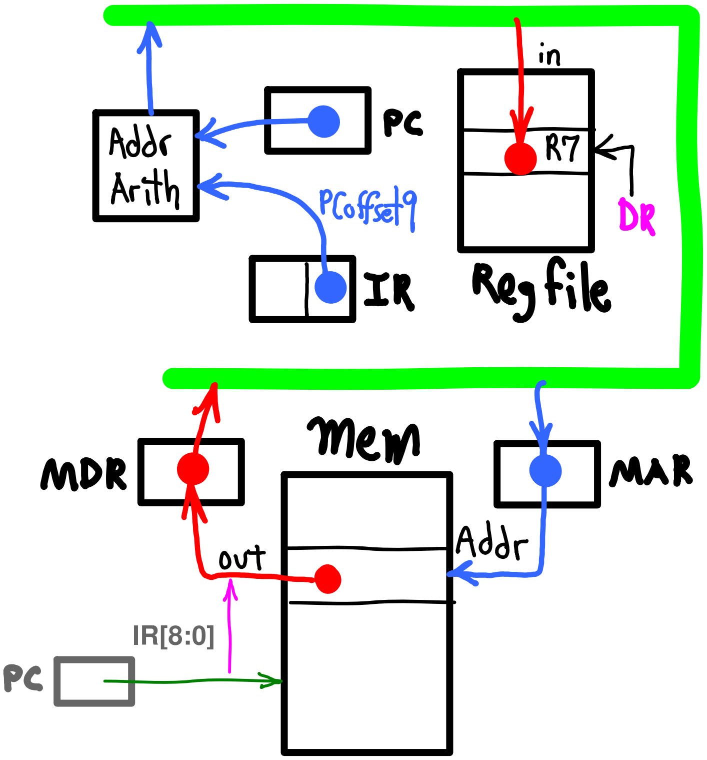
LC3, LD:
(PC-relative addressing mode)
MAR <== PC + IR[8:0] (address is within (PC +/- 511))

R7 ← Mem[PC + PCoffset9]

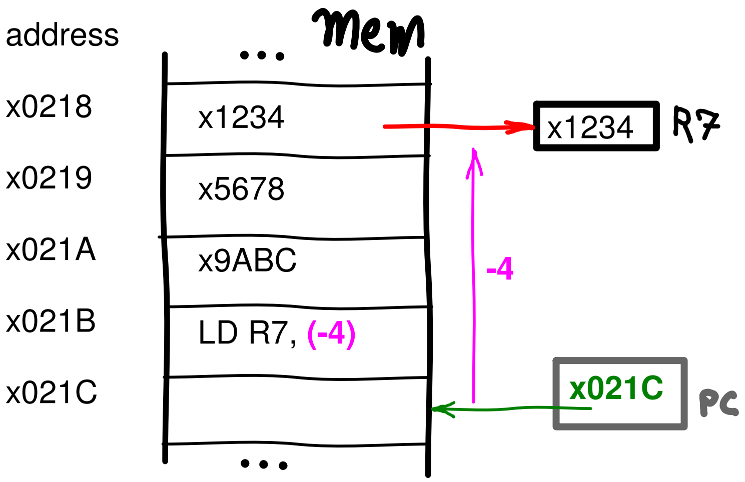
0010	111	1 1111 1000
LD	**R7**	**-x8**

Opcode	**DR**	**PCoffset9**
15...12	11...9	8......0

bits:



PC "points to" memory location.
Data's location is near where PC is pointing.
Offset to data is IR[8:0] (negative, in this case).



address	mem
	...
x0218	x1234
x0219	x5678
x021A	x9ABC
x021B	LD R7, (-4)
x021C	
	...

Q. Why isn't this x021B?

Same as LD, but uses Reg instead of PC

R7 ← Mem[R3 + offset6]

	0110	111	011	01 1000
	LDR	R7	R3	#22
	Opcode	DR	BaseR	offset6
bits:	15...12	11...9	8...6	5...0

Notation

6-bit value expressed

in **decimal**:

LC3 asm style	C style	Verilog style
#22	22	6'd22

in **hexadecimal**:

LC3 asm style	C style	Verilog style
x14	0x14	6'h14

in **binary**

LC3 asm style	C style	Verilog style
?	?	6'b011000

LC4, LDR

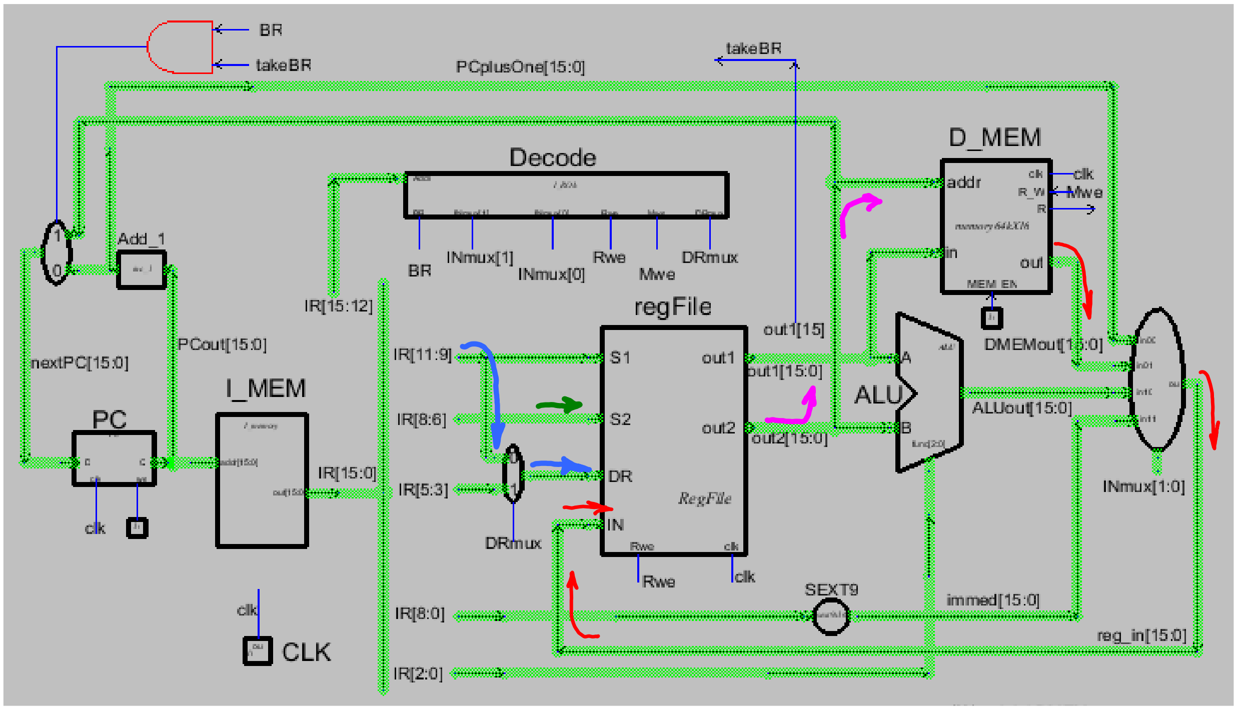
LDR DR7 SR3 xxxxxx

(no offset arithmetic)

DR7 <== **DMEM**[**SR3**]

[**OP DR S2**]
[**0010 111 011** xxxxxx
]

("x" means "It does not matter whether 0 or 1.")



IR[8:6] **---> regFile**.S2 (select R3)
regFile.out **---> D_MEM**.addr[15:0] (address to D_MEM)
IR[11:9] **---> regFile**.DR (select to write **DR7**)
D_ MEM.out[15:0] **---> regFile**.IN[15:0] (send content of memory location to register)

So, how would we do the equivalent of LC3's LD (PC-relative) on LC4?

1.) Get PC content into a register, R1.
2.) Get offset value into a register, R2.
3.) R1 + R2 ==> R3
4.) LDR DR7 AR3

1.) easy part

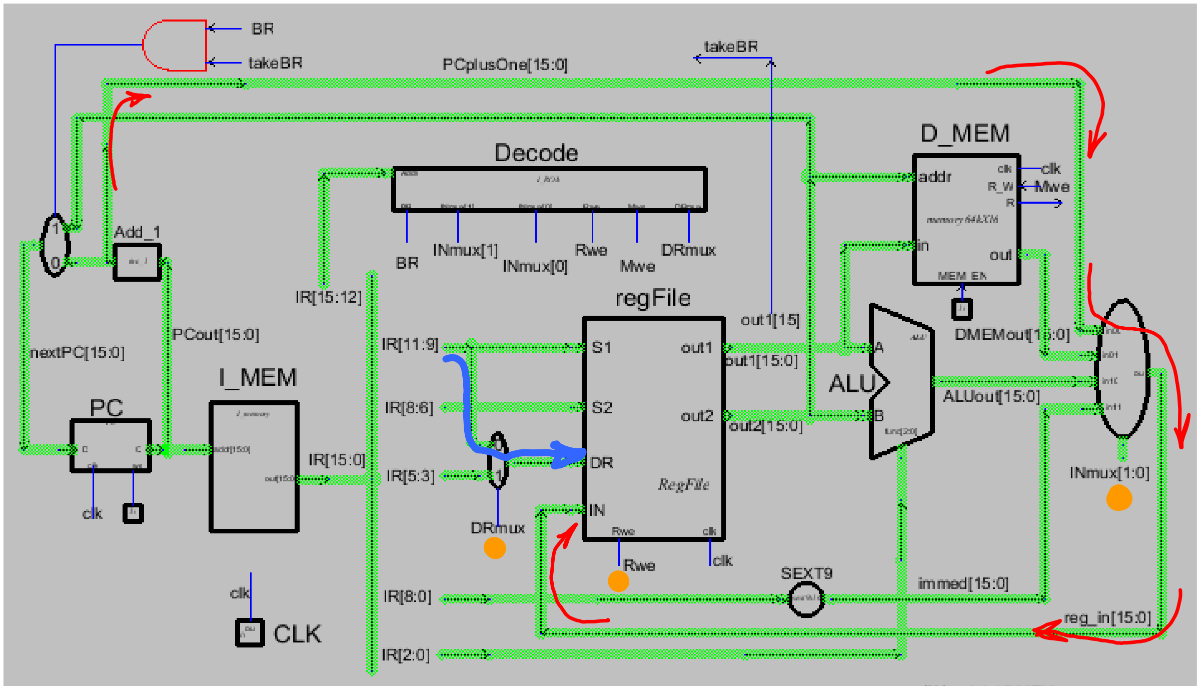
R1 <=== PC

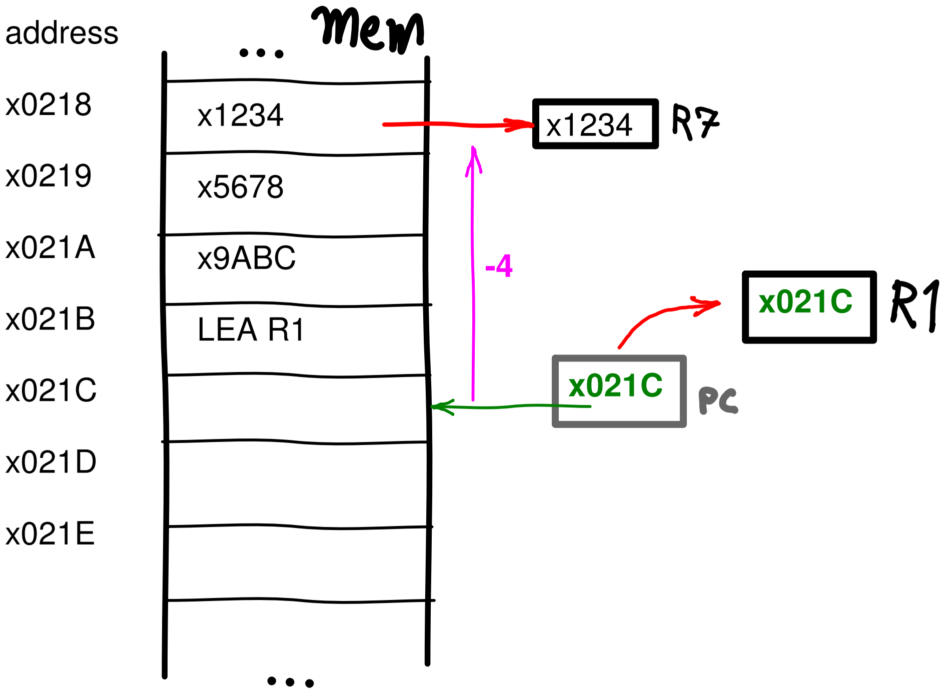
LEA DR1
[1000 001 xxxxxxxxx]

IR[11:9] ---> regFile.DR

PCplusOne ---> regFile.IN

(LC3's LD uses the incremented PC; so, LC4 uses same semantics.)





2.) a little tricky, immediate mode addressing

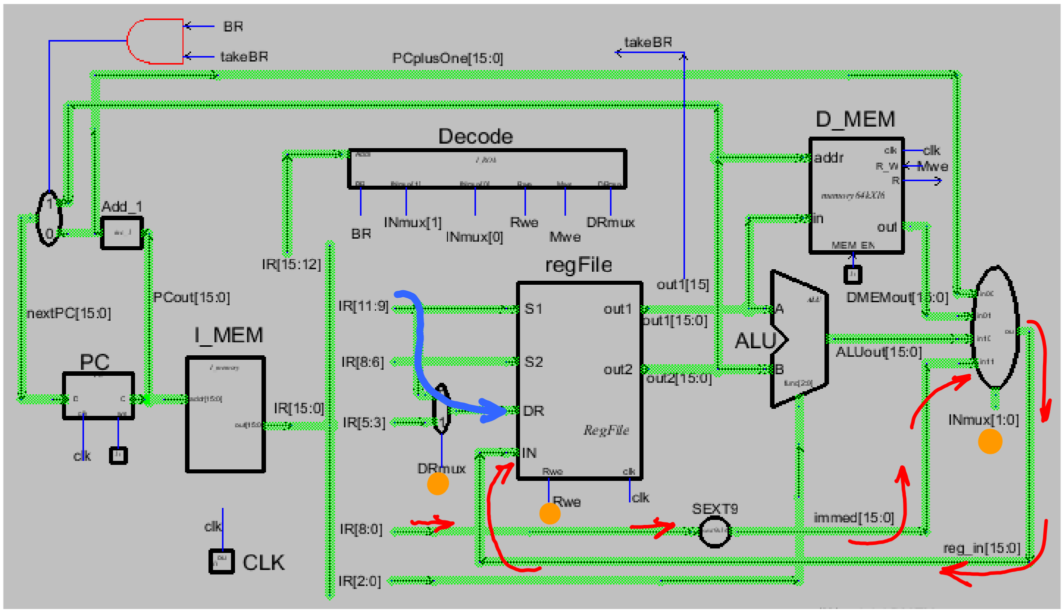
R2 <=== -x4

LIM DR2 -x4
[0001 001 1111111100]

IR[11:9] ---> regFile.DR

IR[8:0] ---> SEXT9.in[8:0]

SEXT9.out[15:0] ---> regFile.IN

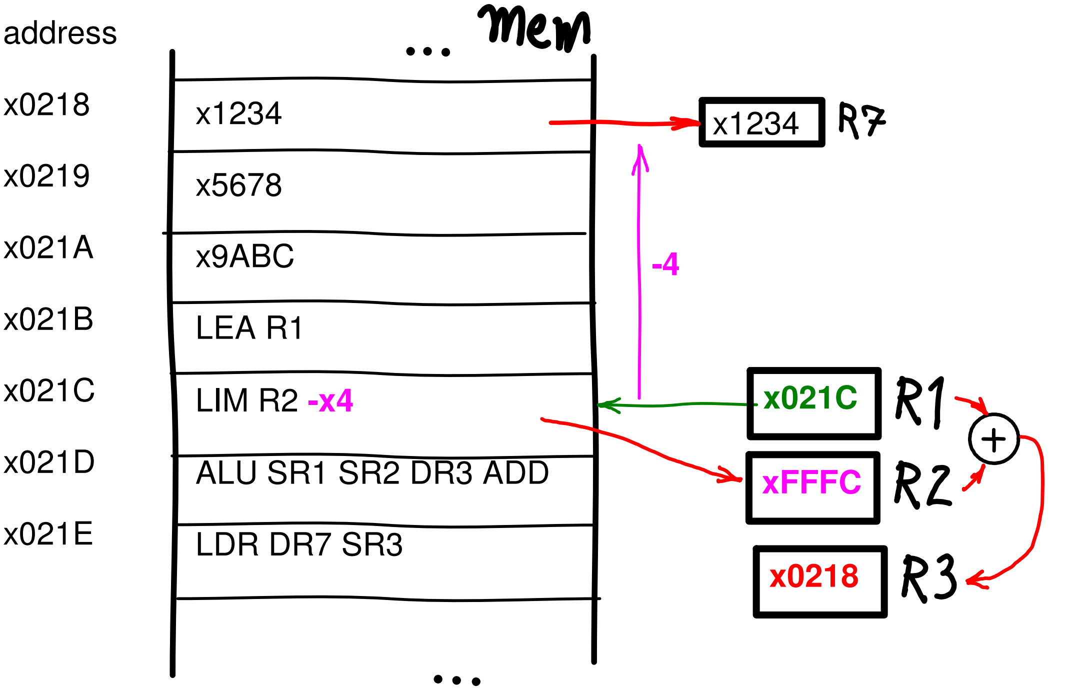


3.) easy

ALU SR1 SR2 DR3 ADD

4.) easy

LDR DR7 AR3



address	... mem
x0218	x1234
x0219	x5678
x021A	x9ABC
x021B	LEA R1
x021C	LIM R2 -x4
x021D	ALU SR1 SR2 DR3 ADD
x021E	LDR DR7 SR3
	...

So, what did we gain?

1. LC4 architecture and instruction set much simpler than LC3's.

2. But, requires 4 instructions instead of 1.

3. Also, LC4 code size might be 4X LC3's.

4. But, ALU operations have larger code size on LC3?

4. Well, maybe LC4 can run much faster? (but how?)

Q. How can we evaluate the tradeoffs?

LC3

R3 ← NOT(R5)

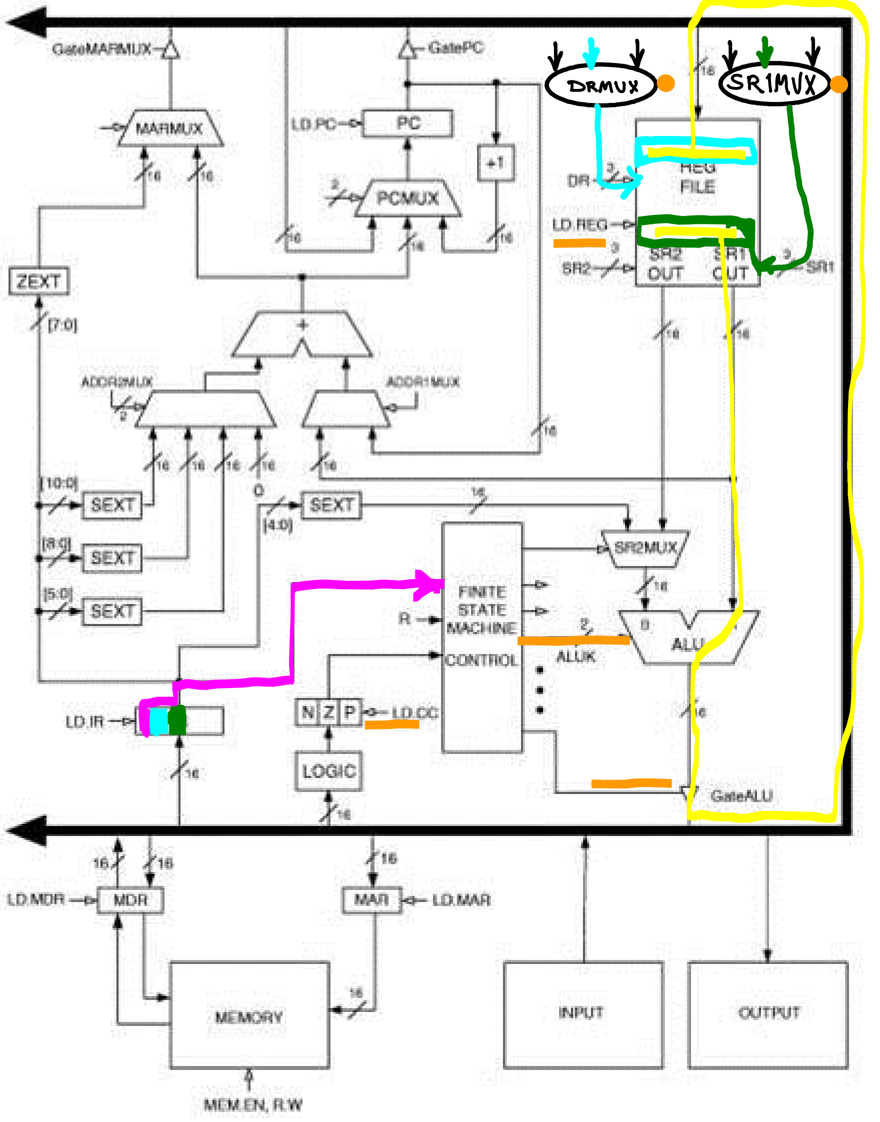
NOT R3 R5

1001	011	101	...

bits: 15...12 11...9 8...6 5......0

DR SR1

Register-Register Addressing



LC3 Operate Instructions (ADD, AND, NOT)

State-9 (**NOT**):

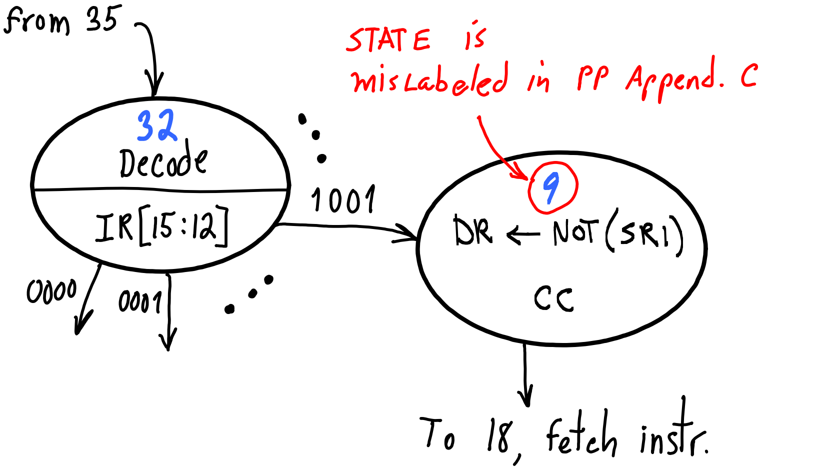
- **IR**[15..12] **---> FSM**.in
- **IR**[11..9] **---> DRMUX ---> RegFile**.DR
- **IR**[8..6] **---> DRMUX ---> RegFile**.SR1
- **ALU.out ---> RegFile**.in

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

GateALU **SR1MUX.select == ?**
LD_REG **DRMUX.select == ?**
LD_CC **ALUK = 10 (NOT)**

Before: Regfile[101] = 1100101011110000 (**R5**)

After: Regfile[011] = 0011010100001111 (**R3**)
Regfile[101] = 1100101011110000 (**R5**)



A complication:
Which IR bits control regFile's SR1 and DR selects?

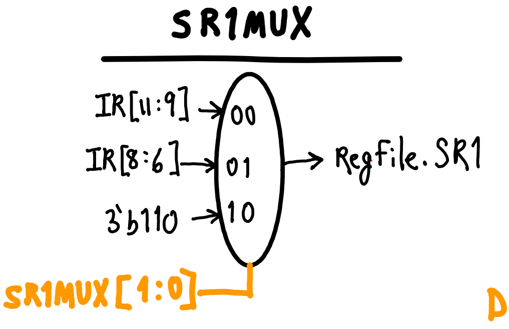
LDR, NOT, ADD, ... IR[8:6] ----> SR1
ST, STI, STR IR[11:9] ----> SR1
push/pop 3'b110 ----> SR1 (access R6)

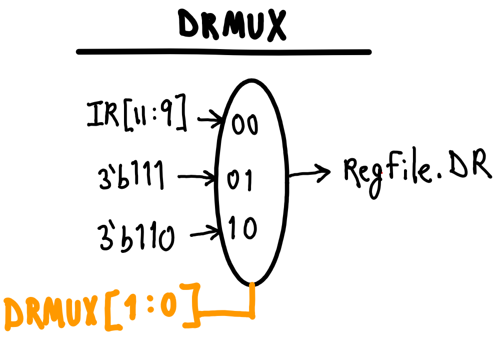
LD, ADD, ... IR[11:9] ----> DR
JSR, JSRR, TRAP 3'b111 ----> DR (access R7)
push/pop 3'b110 ----> DR (access R6)

We MUX the RegFile's inputs.

Control Signals: **DRMUX**
SR1MUX

(see, App. C, p. 574)





ADD (3-register addressing)

State-**1**:

IR[15..12] **---> FSM**.in

IR[11..9] **---> DRMUX ---> RegFile**.DR

IR[8..6] **---> SR2MUX ---> RegFile**.SR1

IR[2..0] **---> RegFile**.SR2

IR[5] **---> SR2MUX**

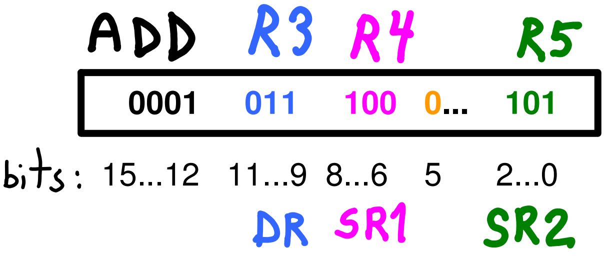
DR <=== RegFile[SR1] + RegFile[SR2]

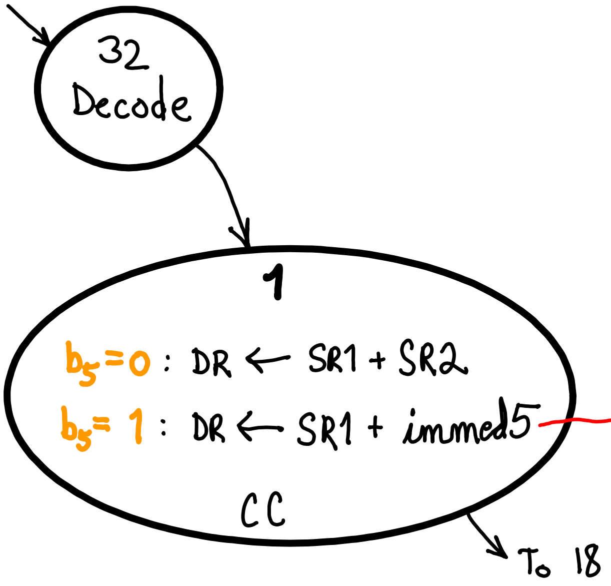
GateALU

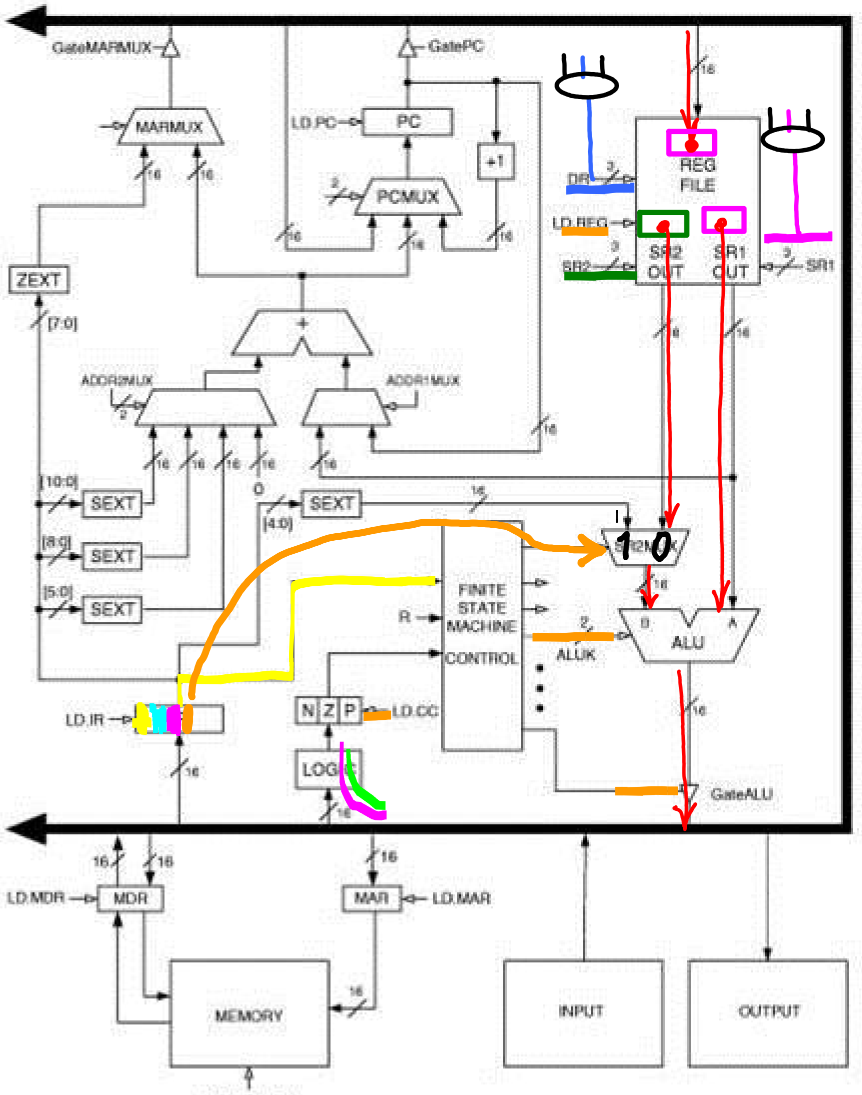
LD_REG

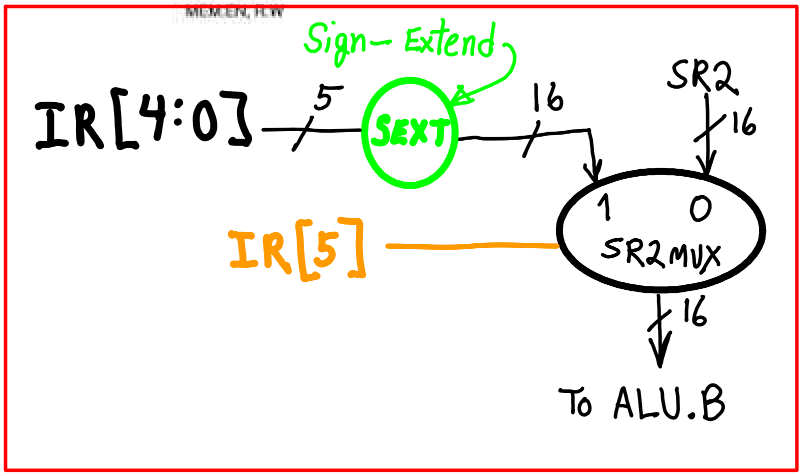
LD_CC

ALUK = 00







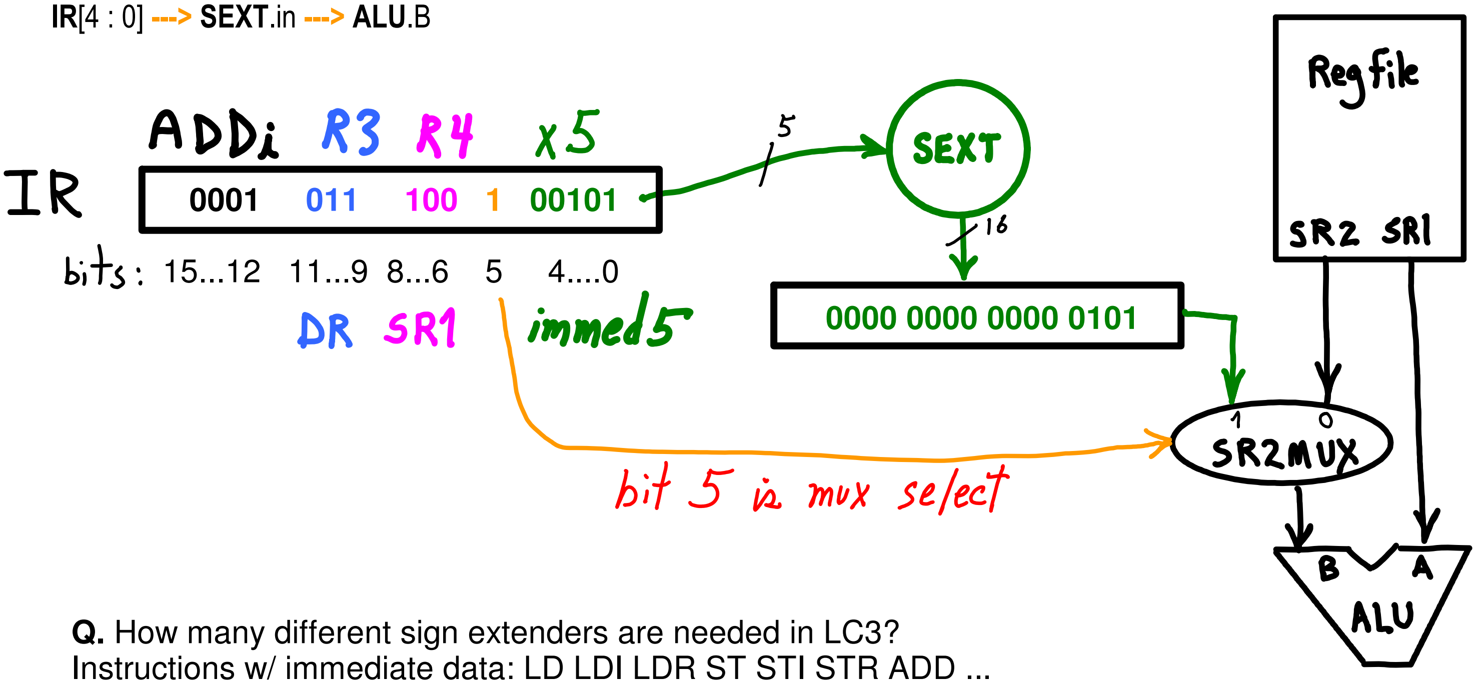


ADD can function two ways:

1. **ADD**: Get both operands from **RegFile,** SR1 and SR2
   ---- ***register-register-register addressing***

2. **ADDi**: Get one operand from **RegFile** (SR1) and the other from **IR**[4 : 0]
   ---- **IR**[4 : 0] (aka, ***immed5*** in this context) is sign-extended from a 5-bit number to a 16-bit number.
   ---- Sign-extending copies the low 5 bits, and then makes the upper 11 bits all 0 or all 1, depending on whether ***immed5*** is positive or negative
   ---- ***register-register-immediate addressing***

Sign-extending the **IR** immediate data bits:
IR[4 : 0] **--->** **SEXT**.in **--->** **ALU**.B



Q. How many different sign extenders are needed in LC3?
Instructions w/ immediate data: LD LDI LDR ST STI STR ADD ...

There are 5-bit, 6-bit offsets, 9-bit, and 11-bit offsets

A - B ? ===> Do **A + (-B)**
2s-complement with immediate constants.

Suppose: **A** in **R0**, **B** in **R1**

1001 001 001 111111
NOT R1 R1

0001 001 001 1 00001
ADD R1 R1 #1

$$\left.\begin{array}{l} R1 \leftarrow NOT(R1) \\ R1 \leftarrow R1 + 1 \end{array}\right\} R1 \leftarrow (-R1)$$

0001 011 000 0 010
ADD R2 R0 R1

$$R2 \leftarrow R0 + R1$$

$$R2 \leftarrow \underset{A}{R0} - \underset{B}{R1}$$

(But, R1 now has -B)

LD DR x0AF

0010	011	0 1010 1111

bits: 15...12 11...9 8......0

PCoffset9 9'h0AF

Load/Store (LD / ST ; pc-relative addressing **)**
load a register from memory / store register in memory
DOES Address ARITHMETIC

LD
State-**2**:

IR[8..0] ---> **SEXT-9x16** ---> **ADDR2MUX**
PC ---> **ADDR1MUX**
(**ADDR2MUX** + **ADDR1MUX**) ---> **MARMUX**

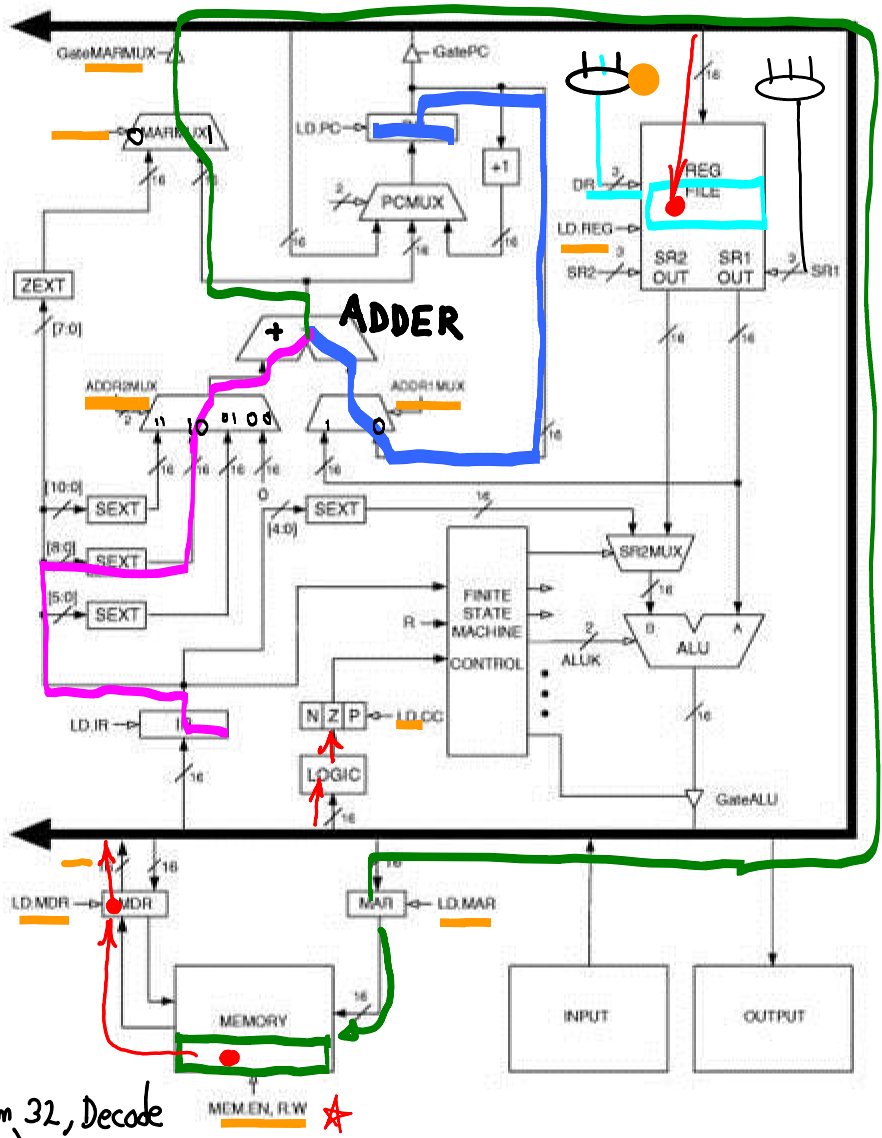
MAR <=== **PC** + **IR**[8..0]
GateMARMUX ADDR1MUX == 1'b0
LD_MAR ADDR2MUX == 2'b10

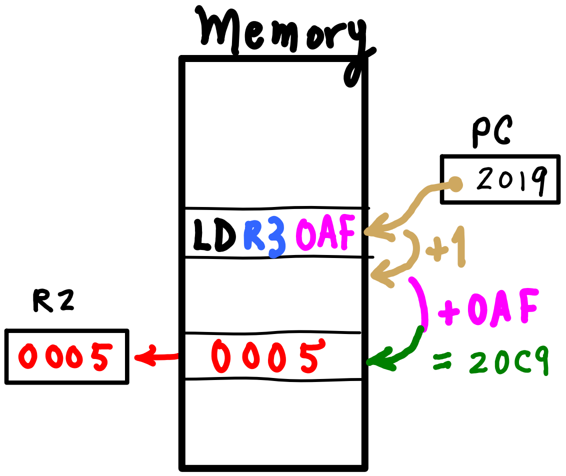
State-**25**:

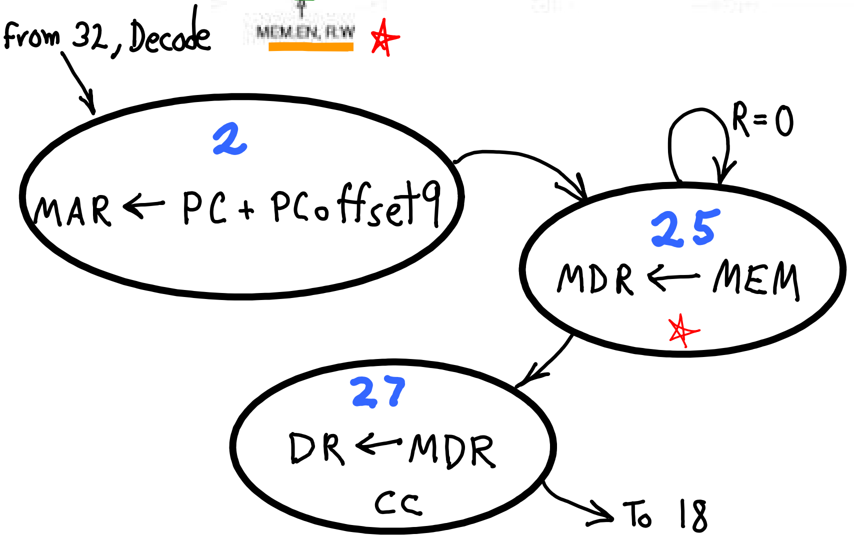
MDR <=== **MEM**.out
LD_MDR
MIO_EN
R_W == 0 } ☆ + Tri-states on MIO Bus

State-**27**:

DR <=== **MDR**
GateMDR LD_REG
LD_CC DRMUX == ?







PC <==	**PC**+1		0010 0000 0001 1010	(**x201A**)
	+ **SEXT**(**IR**[8 : 0])	+	0000 0000 1010 1111	(**x00AF**)
MAR <==		=	0010 0000 1100 1001	(**x20C9**)

R2 <== **MDR** <== **MEM**[**x20C9**] (**x0005**)

LDI R3 x1CC (-x34)

1010	011	1 1100 1100
15...12	11...9	8......0

bits:

LDI / STI
(memory indirect addressing)
Pointer Variables

LDI:

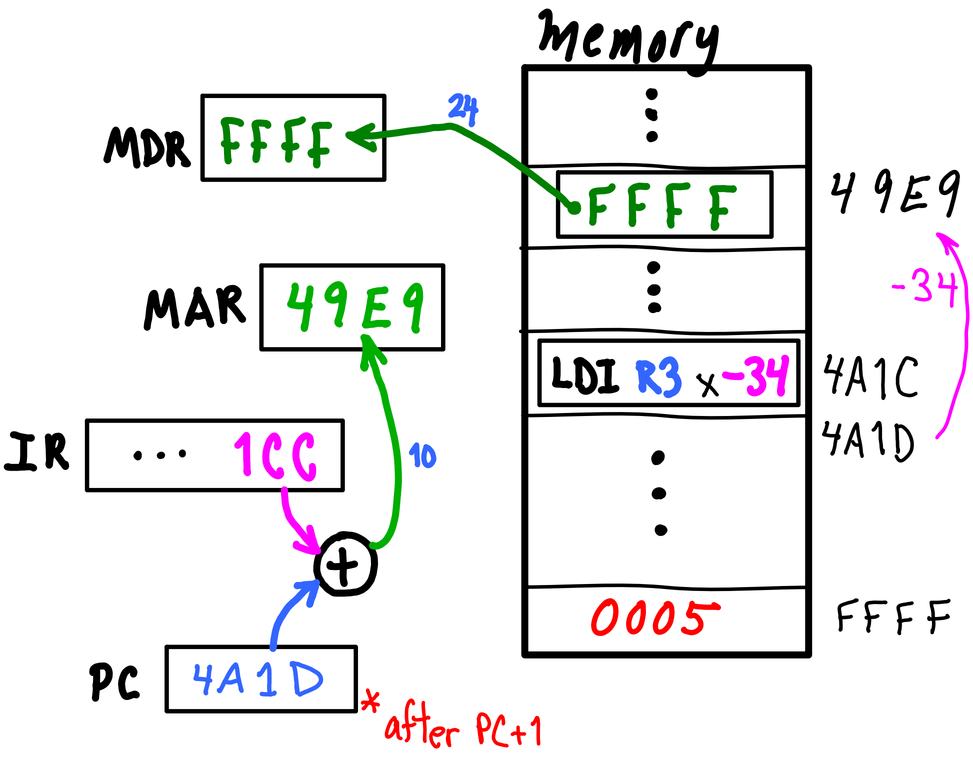
State-10:
MAR <=== PC + IR[8..0]

State-24:
MDR <=== MEM[PC + PCoffset9]

State-26:
MAR <=== MDR

State-25:
MDR <=== MEM[MAR]

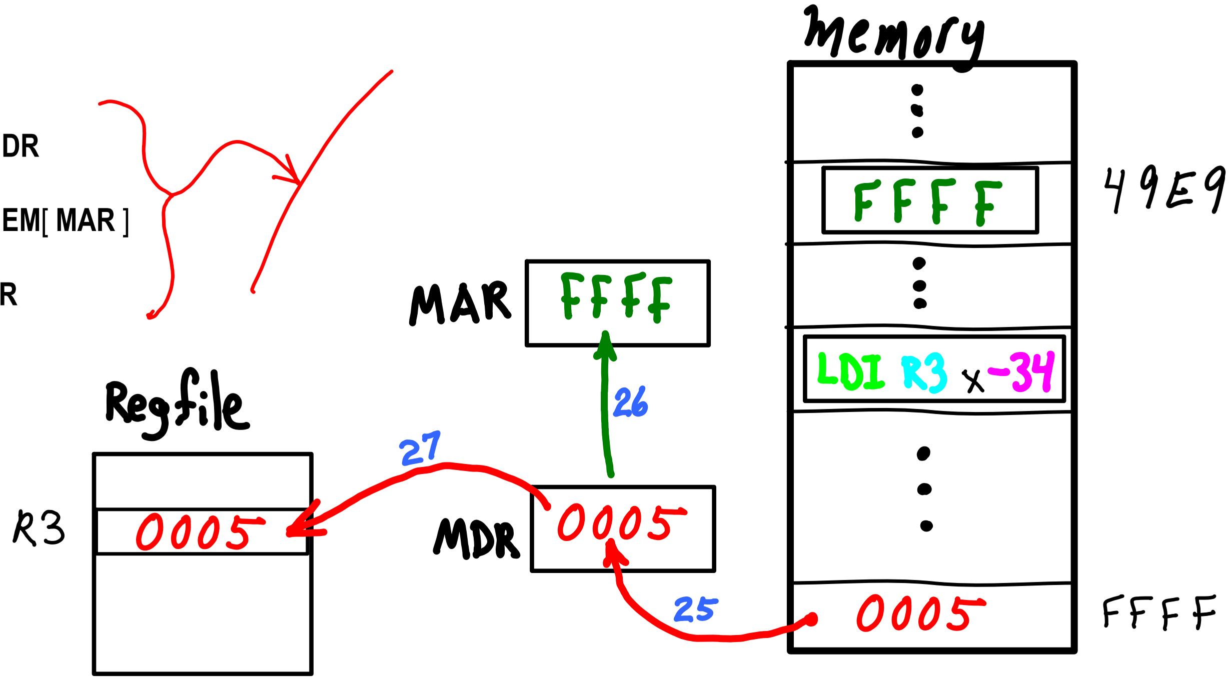
State-27:
DR <=== MDR



State-26:
MAR <=== MDR

State-25:
MDR <=== MEM[MAR]

State-27:
DR <=== MDR

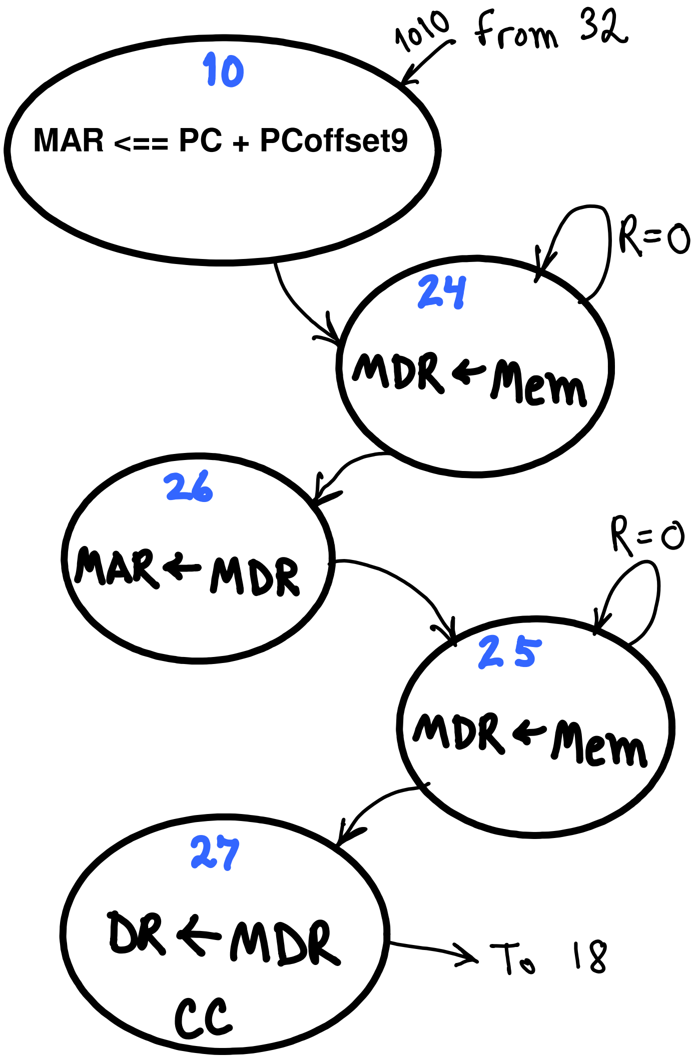


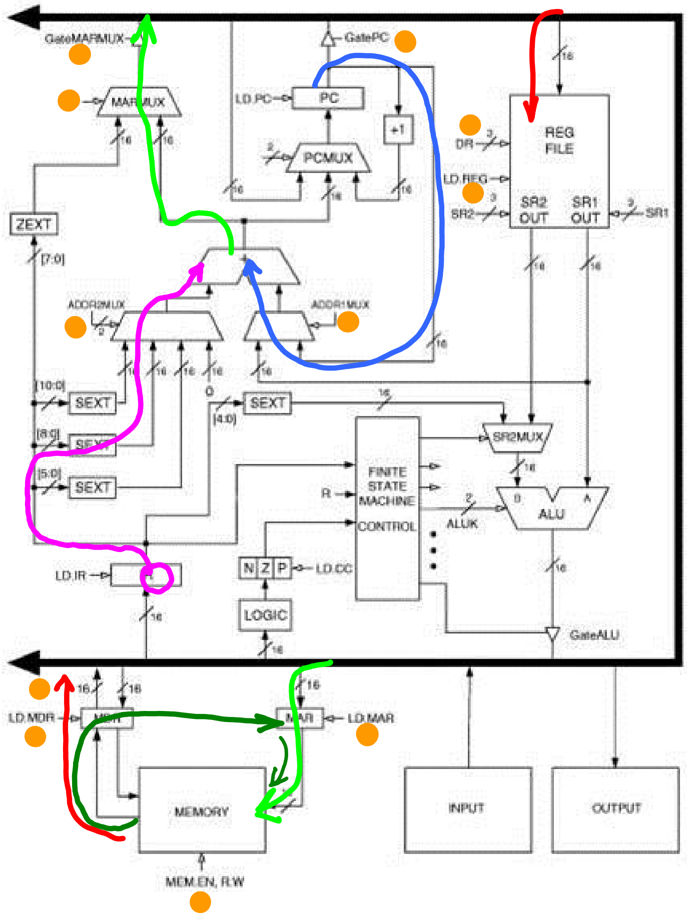
Sanity check

$$\begin{array}{r} {}^{1}4\;A\;{}^{1}1\;D \\ +\;F\;F\;C\;C \\ \hline {}^{1}\leftarrow 4\;9\;E\;9 \end{array}
\qquad
\begin{array}{r} 4 = 0\overset{1}{1}00 \\ +F = 1111 \\ \hline 1 \leftarrow 0011 \end{array}
\qquad
\begin{array}{r} A = \overset{1}{1}\overset{1}{0}10 \\ +F = 1111 \\ \hline 1 \leftarrow 1001 \end{array}
\qquad
\begin{array}{r} D = \overset{1}{1}101 \\ +C = 1100 \\ \hline 1 \leftarrow 1001 \end{array}$$

A	1010
B	1011
C	1100
D	1101
E	1110

$$\begin{array}{r} 4\overset{-1}{A}1D \\ -34 \\ \hline 49E9 \end{array}$$





LC3, What we've got so far:

NOT R1, R1

ADD R1, R2, R3
ADDi R1, R2, x10

AND R1, R2, R3 5-bit data ⇒ ± 16 $(2^5 = 32)$ $[-16, +15] \subseteq \mathbb{Z}$
ANDi R1, R2, #13
decimal

LD R1, R2, x-34
ST R1, R2, x-34

LDI R1, R2, x-34
STI R1, R2, x-34 9-bit PC offset ⇒ ± 258 $(2^9 = 512)$
hex

More addressing range — 16-bit

LDR / STR (register-indirect addressing)
Pointer in a register

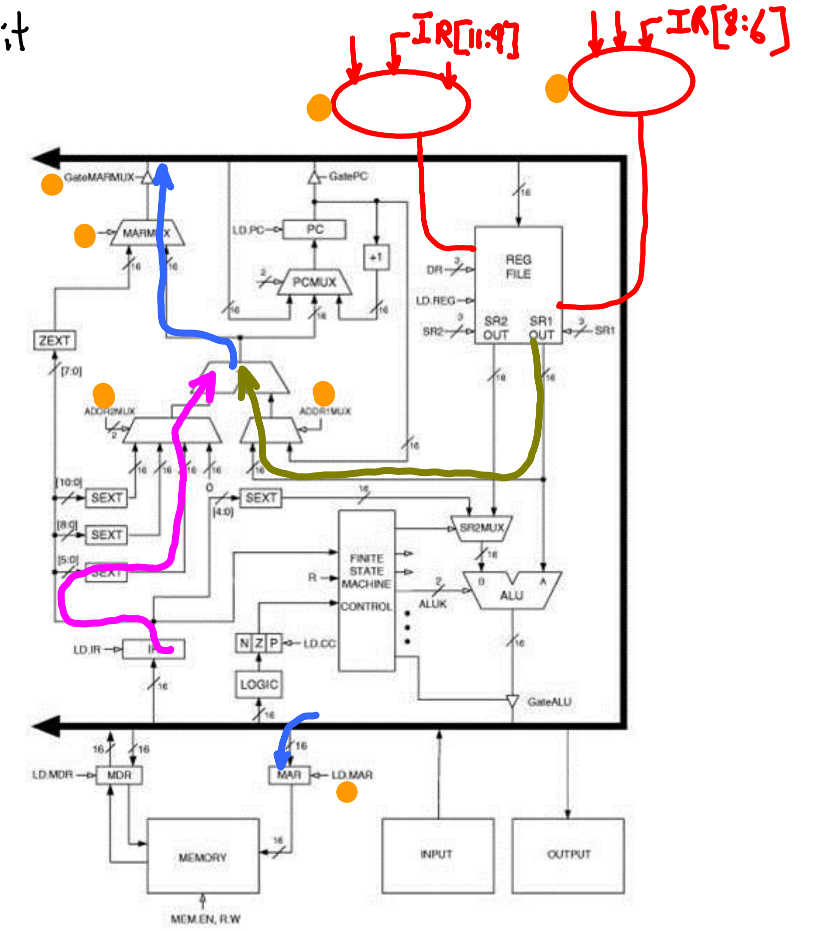
LDR

State-6:

```
        IR[11..9]        ---> RegFile.DR
        IR[8..6]         ---> RegFile.SR1
        RegFile.SR1out   ---> ADDR1MUX
        IR[5..0]         ---> ADDR2MUX
   MAR <== BaseR + offset6
```

State-25:
MDR <== MEM[MDR]

State-27:
DR <== MDR



LDR R3 R6 x0C

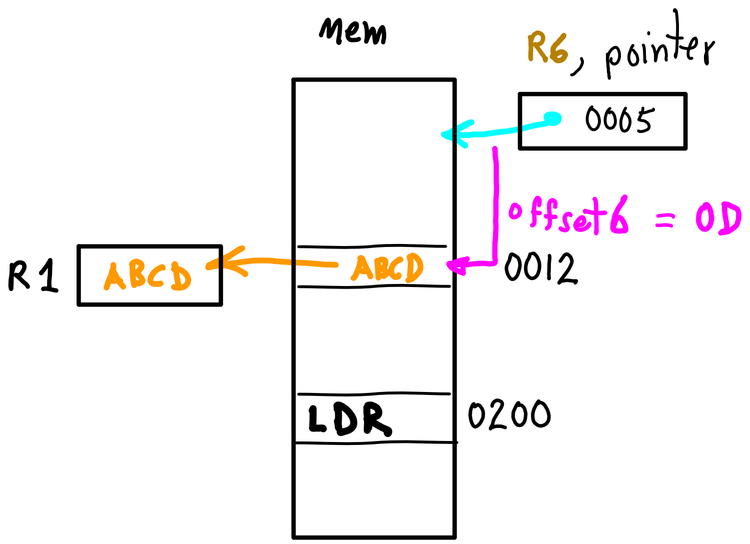
0110	011	110	001101

bits: 15...12 11...9 8...6 5......0

offset6

Full 16-bit addressing

address	Memory content
...	
x0012:	ABCD
...	
x0200:	1010 011 110 001101
...	



Overall Effect

DR <=== MEM[BaseR + Offset6]

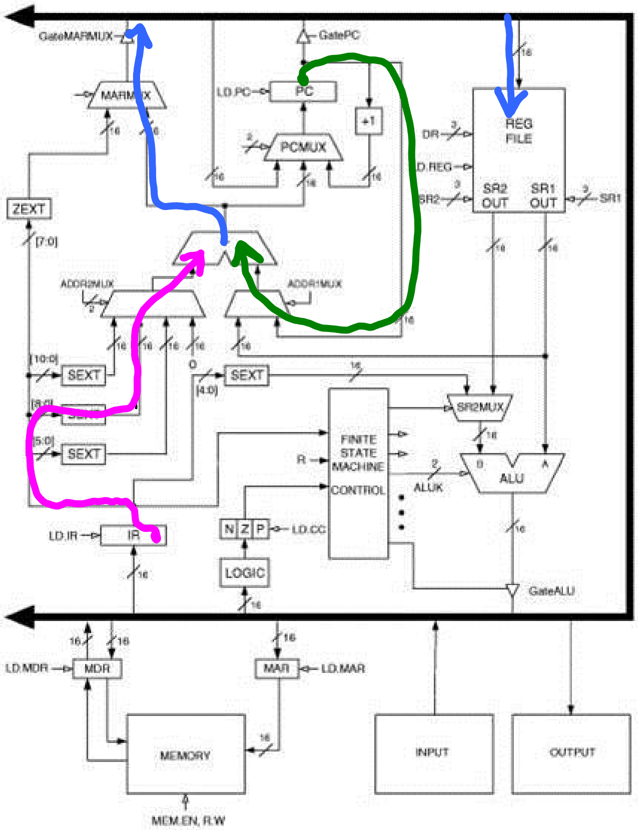
But, how did we get an address (16-bit) into R6?

LEA
(immediate addressing)

State-14:

PC ---> ADDR1MUX
IR[8..0] ---> ADDR2MUX
MARMUX ---> RegFile.in

DR <= PC + PCoffset9



LEA R3 1FD (-3)

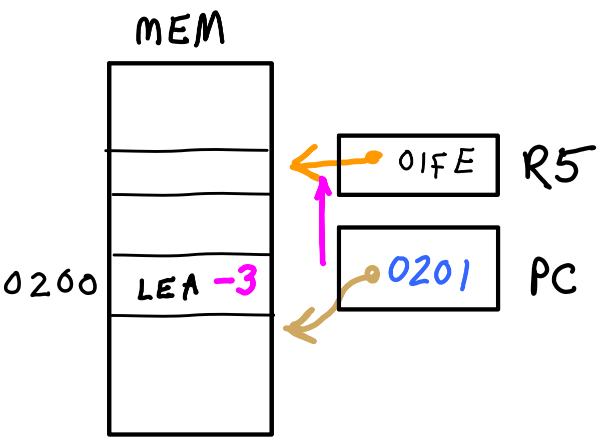
1110	011	1 1111 1101

bits: 15...12 11...9 8......0

Offset9

Memory Content

x01FE:
x01FF:
x0200: 1110 101 111111101 (PC <== x0201)
x0201:



R5 <=== PC + SEXT(1FD)

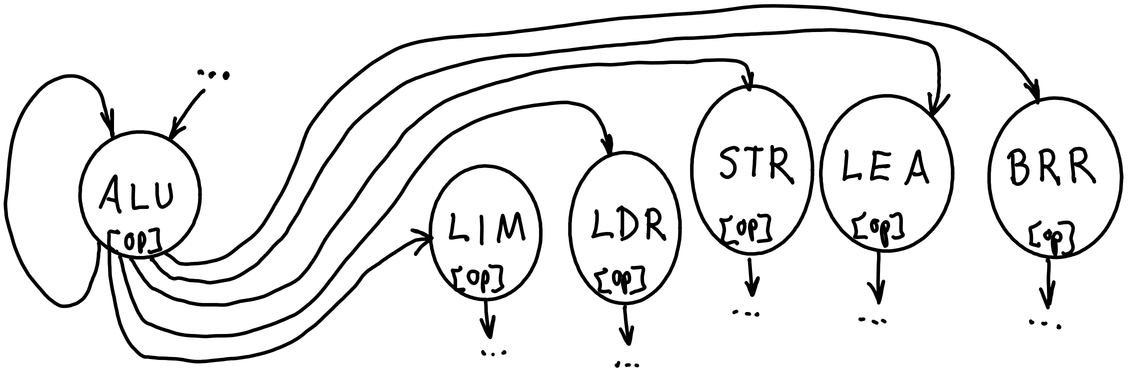
1111 1111 1111 1101
0000 0010 0000 0001

0201 + FFFD = 0201 - 3 = 01FE

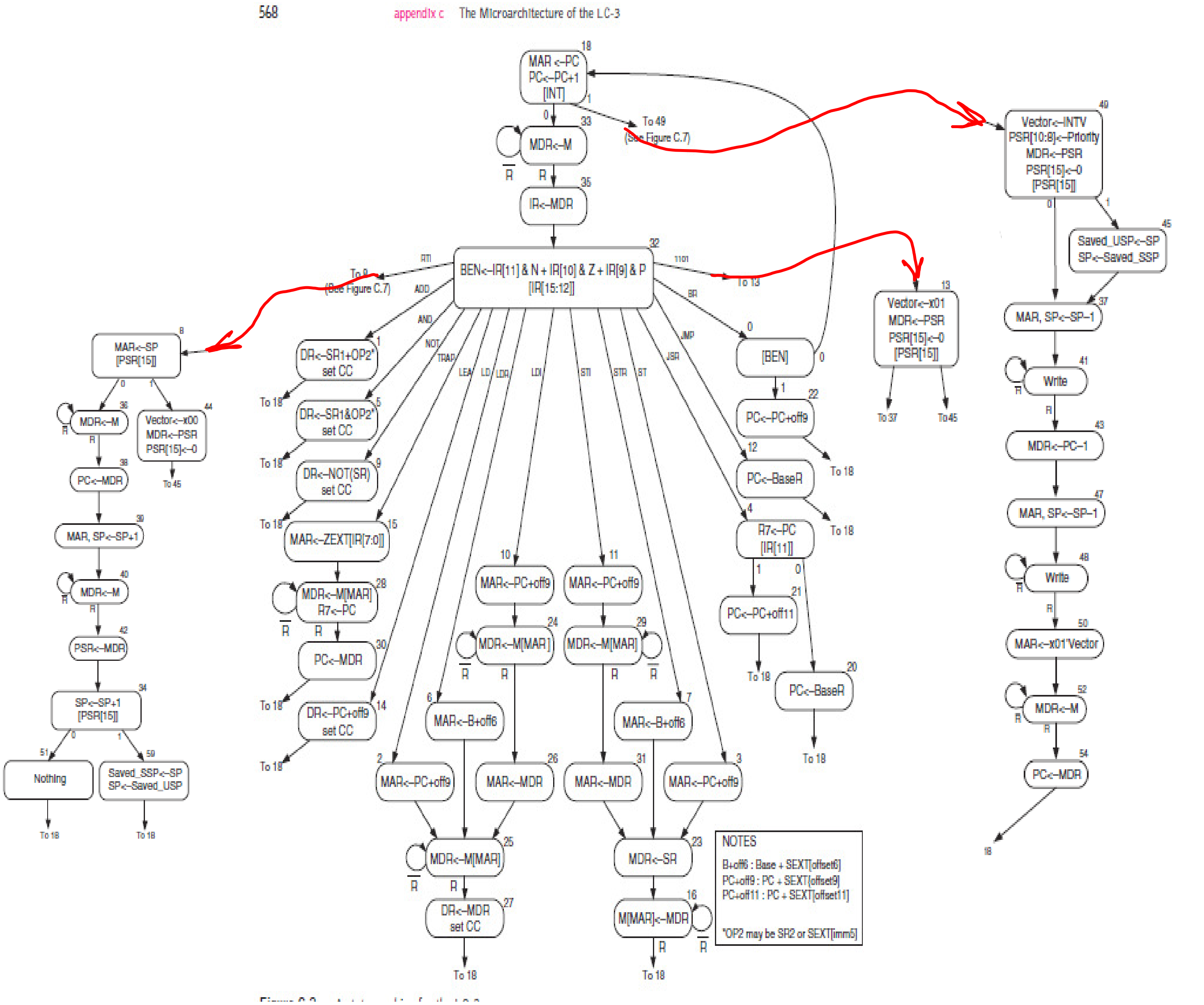
LC4 states

(one state per instruction)

Each state branches to all other states. Which branch is taken depends on next opcode fetched from IMEM.



LC3 states



LC3, What we've got so far:

NOT R1, R1

ADD R1, R2, R3
ADDi R1, R2, x10

AND R1, R2, R3
ANDi R1, R2, #13

LD R1, x-34
ST R1, x-34

LDI R1, x-34
STI R1, x-34

LDR R1, R2, x23
STR R1, R2, x23

LEA R1, x0100

What's left:

BR x123

JSR x123
JSRR R4

JMP R7

TRAP x23

(interrupts and exceptions)

LC4, what we've got so far

ALU SR1 SR2 DR3 ADD
(also SUB, AND, iOR, NOT, NOR, INC, DEC)

LDR DR1 AR2
STR SR1 AR2

LIM DR3 x0A2

LEA DR1

What's left:

BRR CR2 AR7

Simulate LD
LEA DR2
LIM DR3 x-34
ALU SR2 SR3 DR2 ADD
LDR DR1 AR2

Simulate LDI
LEA DR2
LIM DR3 x-34
ALU SR2 SR3 DR2 ADD
LDR DR1 AR2

Simulate LDR
LIM DR3 x23
ALU SR2 SR3 DR2 ADD
LDR DR4 AR2

Simulate LEA
LEA DR2
LIM DR3 x23
ALU SR2 SR3 DR2 ADD

```
-----------------------------------------------
-- Instruction set summaries
-- LC4 details are in header comments in lcasm.c,
-- LC3 details are in Patt&Patel, Appendix A.
-- Notation:
--    SEXTi  = i-to-16-bit-sign-extension
--    d = decimal, h = hex, b = binary, x = hex, # = decimal
--    LC3 branch Condition Codes: N = negative, Z = zero, P = positive
-----------------------------------------------

//-----------------------------------------------
//-- LC3 instructions
//--     syntax                    semantics
//-- ------------------         --------------------
//-- ADD R1 R2 R2               R1 <=== R2 + R3
//-- ADD R1 R2 #3               R1 <=== R2 + SEXT5( d3 )
//-- AND R1 R2 R2               R1 <=== R2 AND R3
//-- ADD R1 R2 #3               R1 <=== R2 AND SEXT5( d3 )
//-- NOT R1 R2                  R1 <=== NOT( R2 )
//-- LD  R1 x89                 R1 <=== MEM[ PC + SEXT9( h89 ) ]
//-- ST  R1 x89                 R1 ===> MEM[ PC + SEXT9( h89 ) ]
//-- LDR R1 R2 x89              R1 <=== MEM[ R2 + SEXT6( h89 ) ]
//-- STR R1 R2 x89              R1 ===> MEM[ R2 + SEXT6( h89 ) ]
//-- LTI R1 x89                 R1 <=== MEM[ MEM[ R2 + SEXT9( h89 ) ] ]
//-- STI R1 x89                 R1 ===> MEM[ R2 + SEXT9( h89 ) ]
//-- LEA R1 x89                 R1 <=== PC + SEXT9( h89 )
//-- BRnzp  x89                 PC <=== PC + SEXT9( h89 ), if (N==n | Z==z | P==p)
//-- JMP R1                     R1 ===> PC
//-- JSR  x89                   R7 <=== PC; PC + SEXT11( x89 ) ===> PC
//-- JSRR R1                    R7 <=== PC; R1 ===> PC
//-- RTI                        PC <=== MEM[ R6 ]; PSR <=== MEM[ R6 + 1 ]
//-- TRAP x8                    R7 <=== PC; MEM[ h8 ] ===> PC

//-----------------------------------------------
//-- LC4
//-- Register Notation:
//--     S = source, D = destination, A = address, C = condition
//--     syntax                    semantics
//-- ------------------         --------------------
//-- ALU SR4 SR3 DR0 ADD         R4 + R3  ===> R0   (also, SUB, AND, iOR, NOR)
//-- ALU SR4 SR4 DR0 NOT         NOT( R4 )===> R0   (also, INC, DEC)
//-- LIM DR1 h36                 R1 <=== SEXT9(h136)   (bits 15-8 are 1; if h36, they are 0)
//-- LDR DR4 AR5                 R4 <=== DMEM[ R5 ]
//-- STR SR3 AR6                 R3 ===> DMEM[ R6 ]
//-- LEA DR3                     R3 <=== PC
//-- BRR CR1 AR7                 R7 ===> PC, if R1<0; else PC+1 ===> PC
```