CS 2110 - Lab 7

LC-3 Instructions & Tracing

Monday, June 13, 2022

Lab Assignment: LC3 Quiz!

1) Go to Quizzes on Canvas
2) Select **Lab 07**, password: **HALT**
3) Get a 100% to get attendance!
    a) Unlimited attempts
    b) Collaboration is **allowed**!
    c) Ask your TAs for help :)

Homework 3

- Released!
- **Due Today at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)

Homework 4

- Released on Friday, June 11th
- **Due Monday, June 20th at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)
- Will be demoed (logistics announced soon)

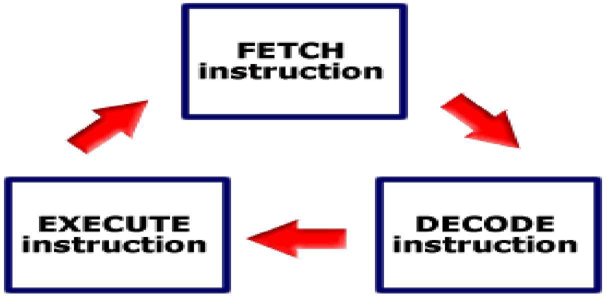
Quiz 2

- **This Wednesday, June 15th**
- Quiz opens at your assigned lab time (unless you have already established a different time with your professor)
- Full 75 minutes to take the quiz!
- After the quiz, you will join lab for the remaining period.
- A topic list will be posted on Canvas

Timed Lab 2

- **Scheduled on Wednesday June 22nd**
- 75 minutes to complete at the beginning of lab, then lab will be resumed.
- Will cover HW4 Material primarily
    - You may use any submitted homework files during the timed lab
    - Unlimited submissions throughout the TL period.

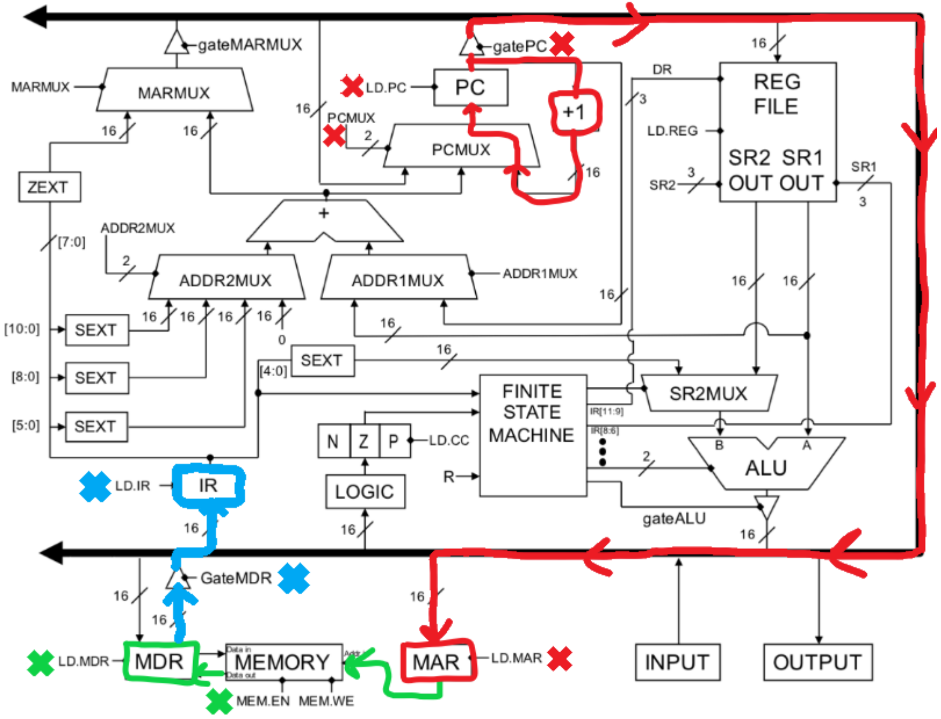
LC-3 Execution Cycle



- The LC-3 cycles between three *macrostates*:
  - **Fetch** (obtaining the instruction at the address stored in the PC)
  - **Decode** (determining which instruction to execute by reading the opcode)
  - **Execute** (doing whatever the instruction says to do)
- The current macrostate is tracked inside the FSM. Some take multiple clock cycles to execute.

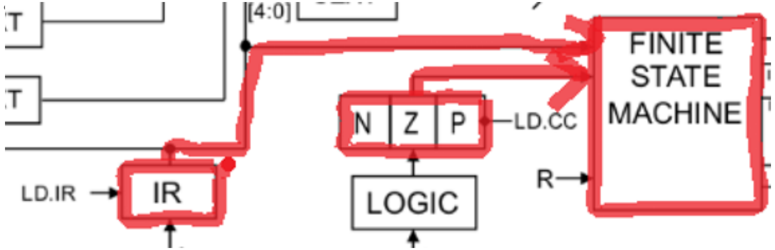
FETCH

- Use PC value (address of next instruction) to read the next instruction from memory
- Load this value into the IR so that it can be decoded and executed later
- Increment the value of PC – it must always point to the **next** instruction to be run



DECODE

- Now that the IR contains the instruction, the FSM can read the opcode.
- If the instruction is BR (branch), the FSM will also check the condition codes to ensure that the branching condition is met.
- The FSM will change to a specific state depending on the opcode. This takes a clock cycle in the LC-3, but some computers can combine DECODE with the last cycle of FETCH.



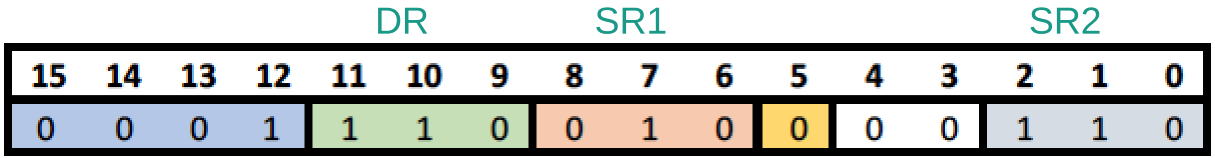
EXECUTE

- Once the instruction is decoded, the EXECUTE macrostate begins.
- Each instruction causes the EXECUTE phase to act differently.
- The EXECUTE macrostate takes a variable number of clock cycles; some instructions like ADD are quick, while others like LDI take longer.
- Once the EXECUTE macrostate completes, the FETCH macrostate restarts, using a new PC address—generally, this is the next instruction in memory
- You will be implementing the EXECUTE macrostate for many instructions (as well as the FETCH macrostate) in HW4

LC-3 Assembly Instructions

ADD

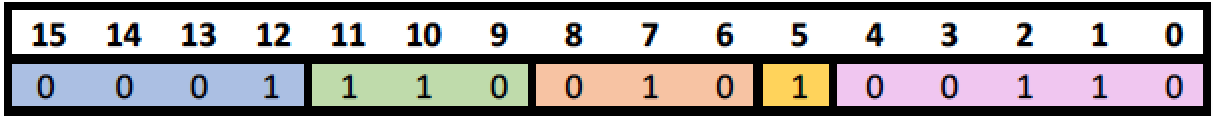
- The ADD instruction is opcode 0001.
- It has two formats (“regular” and immediate) which each work slightly differently.
- Bit 5 signals the format in use: immediate ADD if it is set, and “regular” two-register ADD if it is cleared
- ADD adds the two arguments and stores the result in the destination register (DR).
    - `DR = SR1 + SR2`
- ADD sets the condition codes.
- ADD instruction example: `ADD R0, R1, R2`



					DR			SR1						SR2	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0

ADD (immediate)

- The ADD instruction has a second version, which is enabled by setting bit 5 in the instruction: this bit is wired directly to the SR2MUX selector input
- This interprets the lower 5 bits as a 5-bit 2's complement number. It is sign-extended to 16 bits before the addition takes place.
    - `DR = SR + imm5`
- As a result, the range of numbers that can be supplied to this instruction is [-16,15]. To add larger numbers, either use a register to hold the second value, or use multiple immediate-add instructions.
- Immediate ADD example: `ADD R0, R1, #5`



					DR			SR1					immediate5		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	1	0	0	1	1	0

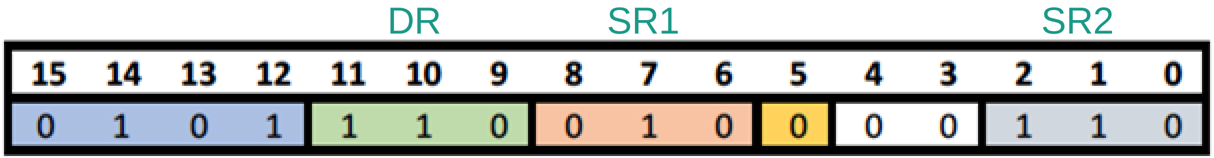
What does this instruction do?

- ADD R0, R2, R0

Which instruction copies the value from R3 to R4?

AND

- The AND instruction is opcode 0101.
- It has two formats as well and works in the exact same way as ADD except performing bitwise AND instead of ADD.
  - `DR = SR1 & SR2`
- As with ADD, the result is stored in the destination register
- AND sets the condition codes.
- AND instruction example: `AND R0, R1, R2`



					DR			SR1						SR2	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	0

AND (immediate)

- Like ADD, AND has a version that supports ANDing a register with a 5-bit 2's complement immediate value.
  - `DR = SR & imm5`
- This is extremely useful when you need to store the value 0 in a register; any register ANDed with zero results in zero.
- Immediate AND instruction example: `AND R3, R3, #0`

				DR			SR1				immediate5				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	1	0	1	0	0	1	1	0

NOT

- The NOT instruction is opcode 1001.
- It only has one format as shown below.
- NOT performs bit-wise negation of the source register (SR) and stores the result in destination register (DR).
    - `DR = ~SR`
- NOT sets the condition codes.
- NOT instruction example: `NOT R1, R0`
- Unlike ADD and AND, there is only one source register for NOT.

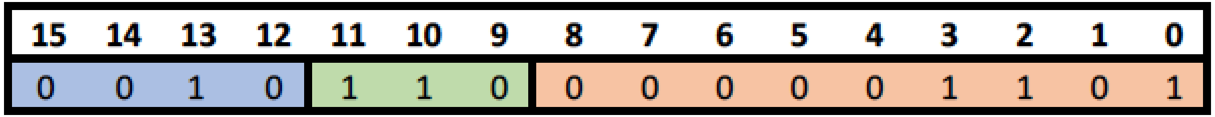
					DR			SR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	0	1	0	1	1	1	1	1	1

LD

Remember: PC* is the incremented PC.

- The LD instruction is opcode 0010.
- LD is the load instruction. It loads the value at the memory address PC* + PCOffset9 and puts it in the destination register.
    - `DR = mem[PC* + PCOffset9]`
- PCOffset9 can be a 9-bit 2's complement literal value or calculated using a label
- LD sets the condition codes.
- `LD R0, LABEL` reads the value at the memory address corresponding to `LABEL`
- `LD R0, #0` reads the value at the location in memory addressed by PC*

DR PCoffset9

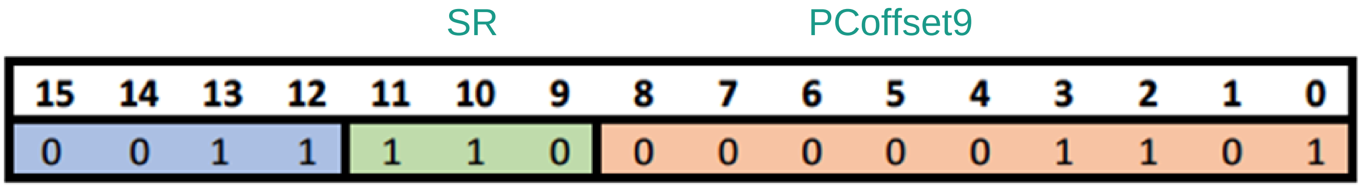


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	0	0	0	0	1	1	0	1

ST

Remember: PC* is the incremented PC.

- The ST instruction is opcode 0011.
- ST is the store instruction and is very similar to LD. It stores the value in the source register at the memory address PC* + PCOffset9.
  - `mem[PC* + PCOffset9] = SR`
- `ST R0, LABEL` stores the value in R0 at the memory address marked by LABEL
- Like LD, ST can also be written with literal numbers (`ST R0, #1`)



SR PCoffset9

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0	0	0	0	0	1	1	0	1

Labels

- Some LC- 3 instructions use the PC-relative addressing mode for operands
- These accept labels as a second parameter, allowing programmers to load from labeled memory addresses.
- Labels are not stored anywhere in memory. Instead, the assembler calculates the right numerical offset to get to the desired label.
- To calculate the offset yourself, find the difference between the memory address of the label and the **incremented** address of the instruction (which will be the value stored in the PC when it executes).

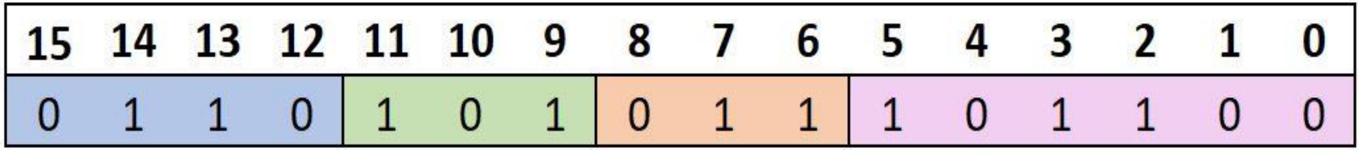
Instruction	Address	PC* (value of incremented PC)
LD R1, LABEL	x3000	x3001
...	x3001	x3002
...	x3002	x3003
...	x3003	x3004
LABEL .fill #1234	x3004	x3005

Address of LABEL – value of incremented PC
= x3004 – x3001
= 3

Therefore, `LD R5, LABEL` becomes `LD R5, 3`
In machine code: `[0010][101][000000011]`

LDR

- The LDR instruction is opcode 0110.
- LDR is the Load Base+Offset instruction. It calculates its address by adding a register's value to a 6-bit 2's complement value that comes from the instruction.
    - `DR = mem[BaseR + offset6]`
- LDR sets the condition codes.
- LDR instruction example: `LDR R0, R5, #0`
- LDR is useful when working with arrays. For example, if R5 were the address of the start of an array, the instruction would put the first element of the array into R0.



				DR			BaseR			PCoffset6					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	1	1	1	0	1	1	0	0

STR

- The STR instruction is opcode 0111.
- STR is the Store Base+Offset instruction.
- It stores the value from the source register at the memory location given by the sum of the base register’s value and the immediate offset.
    - `mem[BaseR + offset6] = SR`
- Like LDR, STR is useful when working with arrays.
- STR instruction example: `STR R0, R5, #6`

				SR			BaseR			PCoffset6					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	1	1	0	0	0	1	0	0	1	1

LDI

Remember: PC* is the incremented PC.

- The LDI instruction is opcode 1010.
- LDI is the Load Indirect instruction.
- It reads the value at the memory address PC* + PCOffset9, then uses this value as the memory address to retrieve the final data.
  - `DR = mem[mem[PC* + PCOffset9]]`
- PCOffset9 may be a literal 9-bit 2’s complement value or use a label.
- LDI sets the condition codes.

DR PCoffset9

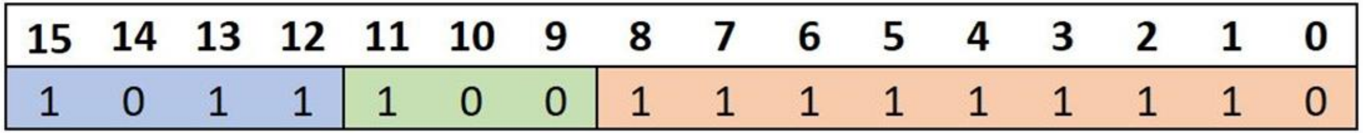
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0	0	1	0	1	0	1	1	1

STI

Remember: PC* is the incremented PC.

- The STI instruction is opcode 1011.
- STI is the Store Indirect instruction.
- It reads the value at the memory address PC* + PCOffset9, then uses this value as the memory address at which it will store data retrieved from a general-purpose register.
    - `mem[mem[PC* + PCOffset9]] = SR`
- PCOffset9 may be a literal 9-bit 2’s complement value or use a label.

SR PCoffset9



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	1	1	1	1	1	1	1	1	0

LEA

Remember: PC* is the incremented PC.

- The LEA instruction is opcode 1110.
- LEA is load effective address.
- It stores the value of PC* + PCOffset9 in the destination register.
    - `DR = PC* + PCOffset9`
- LEA does **NOT** set the condition codes (as of version 3 of the textbook).
- LEA is great when you want to easily obtain the address of a label in assembly code without having to count lines.
- LEA instruction example: `LEA R0, LABEL`

				DR			PCoffset9								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	0	0	0	1	1	0	1

BR

Remember: PC* is the incremented PC.

- The BR instruction is opcode 0000.
- The BR instruction conditionally sets the PC to the value PC* + PCOffset9, allowing you to branch to a different instruction rather than running the next one
- The "conditional part" is handled by only taking the branch if the value in CC is equal to any of the control bits set in the instruction. For example, if the NZP bits are 110 and the value in the CC registers is 001, then the branch will not be taken and the PC will be incremented by one like normal.
- The instructions BR and BRnzp both function as unconditional branches; they will always be taken.

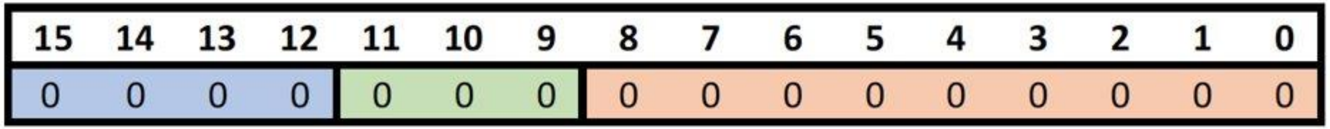
				N	Z	P	PCoffset9								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	1

Assuming we have a loop WHILE:
How do we unconditionally branch to WHILE?

- (Two ways)

NOP

- When a BR instruction has a 0 in each of the NZP locations, it will never activate its branch condition.
- For this reason, a Branch instruction composed of all 0s is considered a NOP (should be said as No Operation or No-Op).
- Running this instruction will have no effect at all on the datapath, beside the ordinary incrementation of the PC during the FETCH stage.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

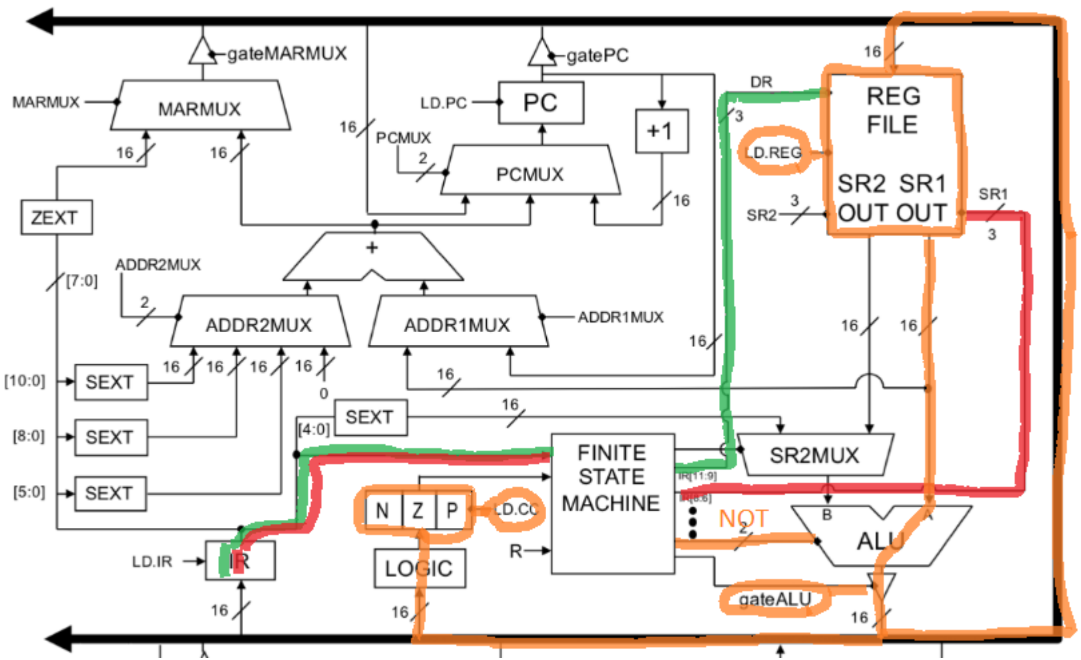
Which instructions set the condition codes?

- ALU instructions
  - `ADD, AND, NOT`
- Load Instructions
  - `LD, LDR, LDI`
  - **Not** LEA
- Basically, any instruction that updates the register file (except `LEA`)

Instruction Tracing Examples

Tracing NOT

- NOT is a simple instruction: it only uses two registers.
- The ALU is set to NOT.
- The destination register is overwritten in the same way as in ADD and AND.
- The condition codes are set based on the value of the bitwise NOT.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	0	1	0	1	1	1	1	1	1

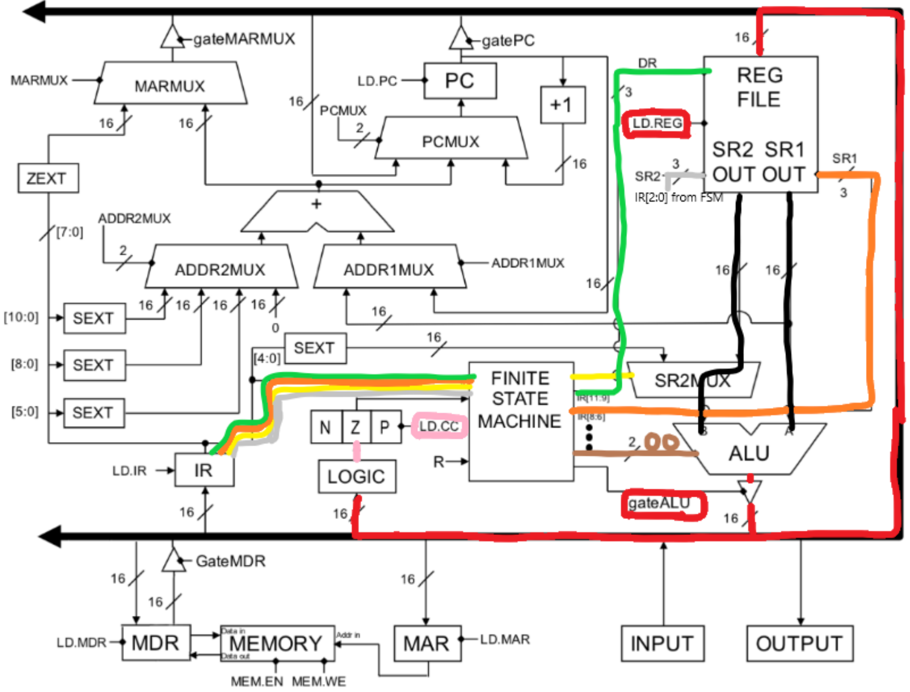
Destination Register

Source Register 1

Tracing ADD (2 registers version)

- Several values from the IR are used to control the SR1, SR2, and DR 3-bit inputs to the register file.
- Bit 5 controls the SR2MUX, making it select SR2 and not the imm5 value.
- The ALU is configured to ADD (00), and its value is pushed to the bus so that it can reach the register file.
- The condition codes are set using the result of the addition.
- What would look different for an immediate ADD?

DR=SR1+SR2



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0

Destination Register

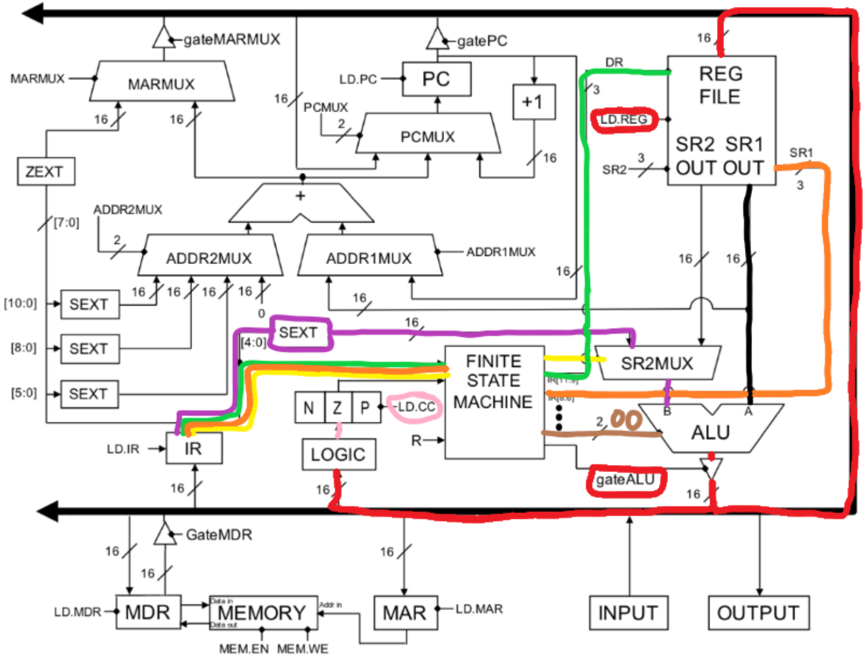
Source Register 1

Source Register 2

Tracing ADD (immediate version)

- Like the 2-register version, ADD uses bit 5 to control SR2MUX. Except now, it is selecting the sign-extended lower 5 bits of the IR.
- Nearly everything else is handled identically as in the 2-register ADD instruction.

DR=SR1+imm5



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	1	0	0	1	1	0

Destination Register

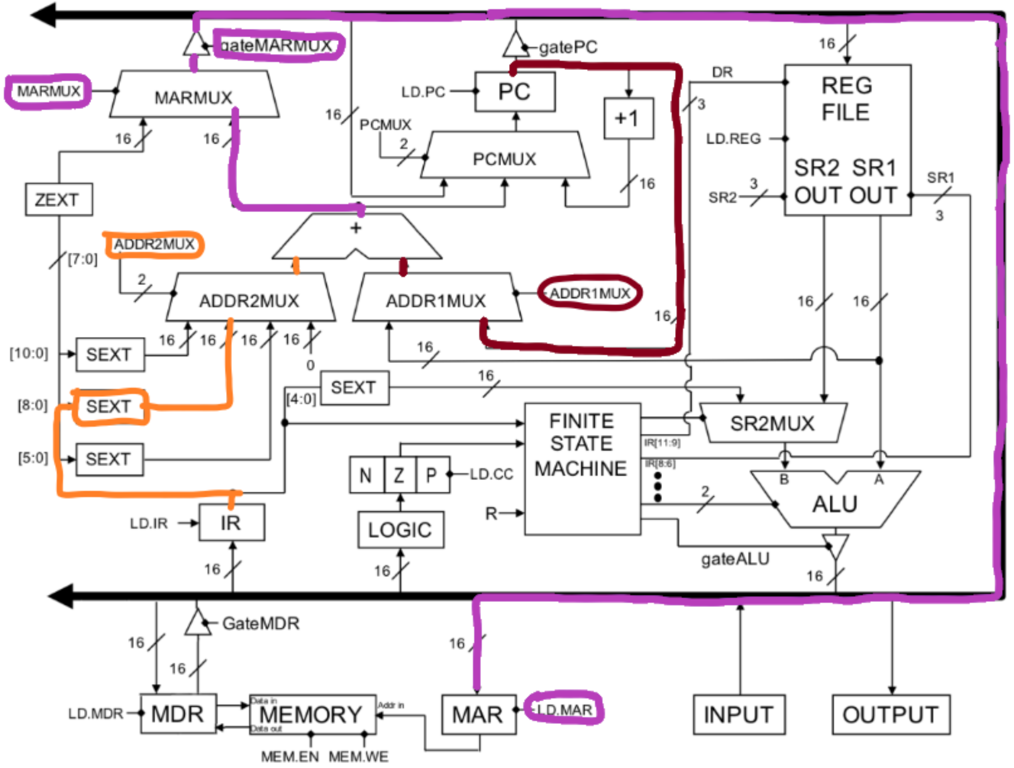
Source Register 1

5-bit 2's Complement Immediate Value

Tracing ST Cycle 1

- The first step of nearly every load and store instruction is to calculate the address.
- In this case, ST calculates the address by adding the current value in the PC (remember that it was incremented!) to the 9-bit 2's complement number included in the instruction.
- The two ADDRMUXes control which numbers get added.
- This calculated address is passed through the MARMUX so it can reach the MAR.

mem[PC*+PCOffset9] = SR

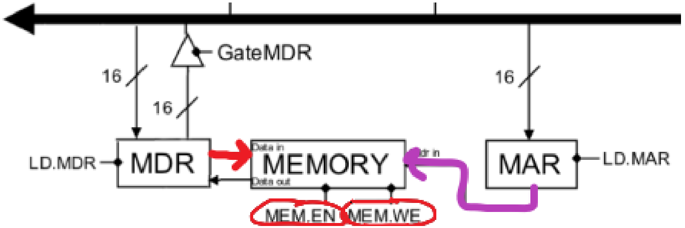


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0	0	0	0	0	1	1	0	1

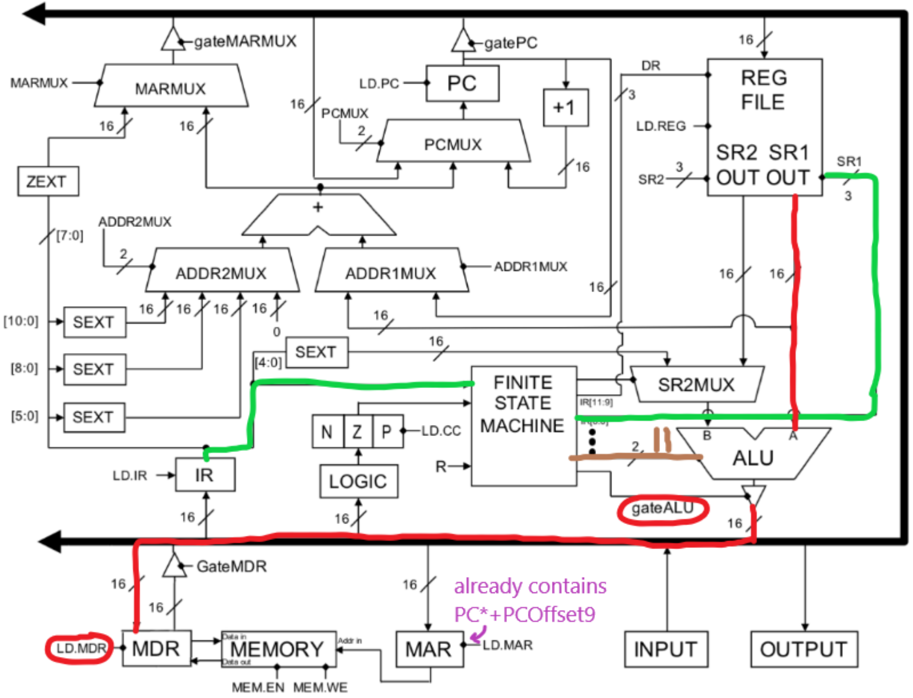
SR PCOffset9

Tracing ST Cycles 2, 3

- Cycle 2 (right) moves the value from the source register into the MDR. The PASS A operation is used here.
- Cycle 3 (below) activates both the MEM.EN and MEM.WE signals. This causes the memory to overwrite the location defined by MAR using the data stored in MDR.

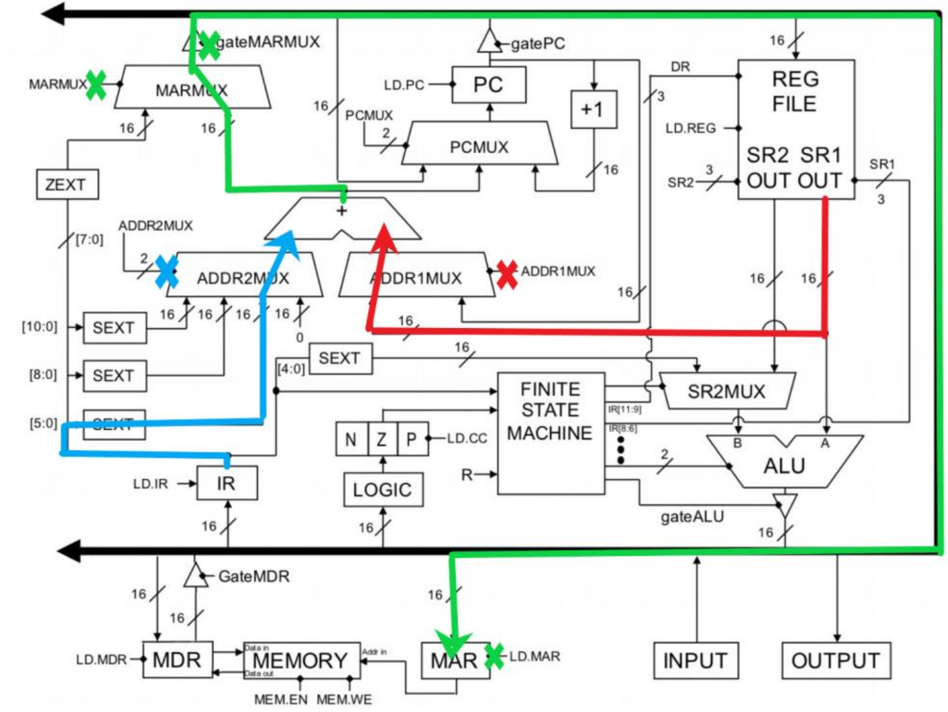


mem[PC*+PCOffset9] = SR

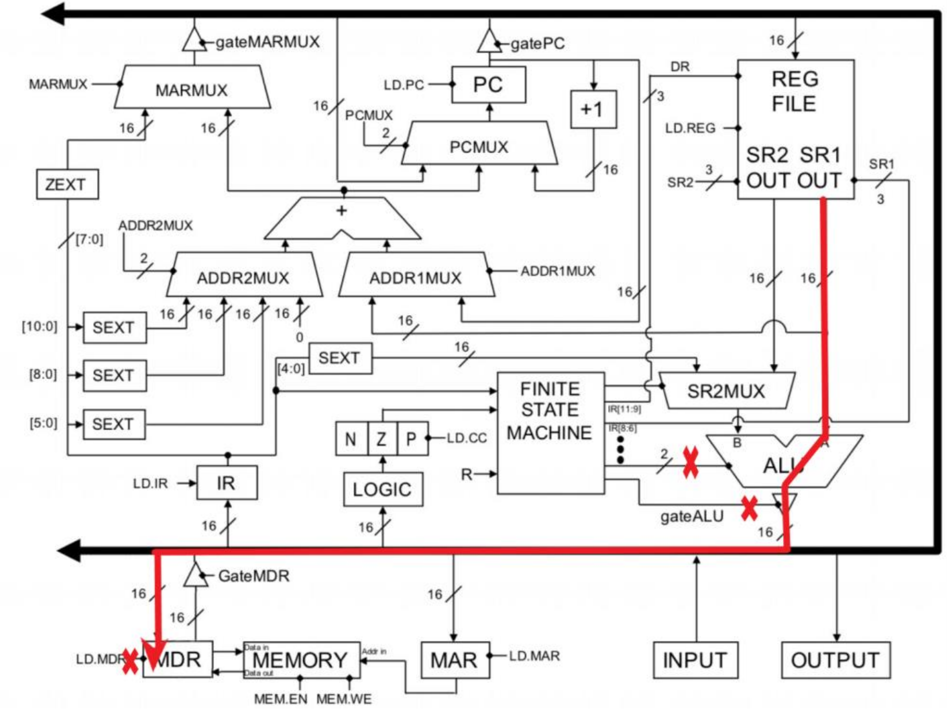


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0	0	0	0	0	1	1	0	1
					SR						PCOffset9				

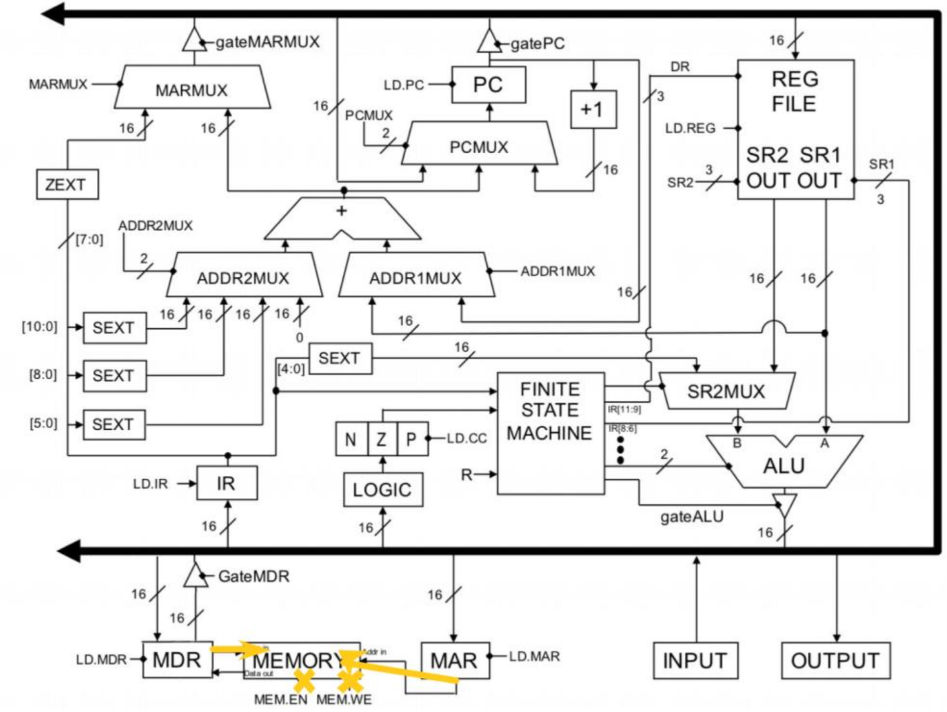
Execute: STR (Part 1)



Execute: STR (Part 2)



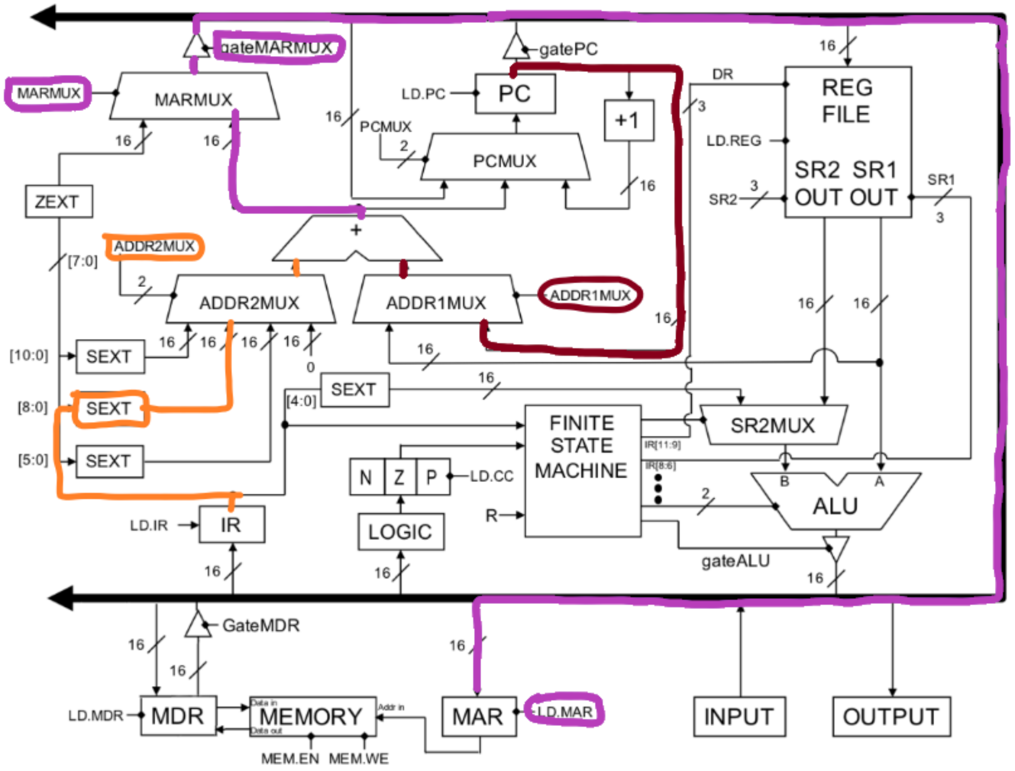
Execute: STR (Part 3)



Tracing LDI Cycle 1

- Just like in ST, we first want to calculate the address to read from. However, as we are using indirect addressing, this will take **three clock cycles**, as determining the address involves a memory address.
- We must calculate the first address that we are reading from. In fact, this is identical to the first cycle of ST, as we are calculating PC* + PCOffset9 and putting it in the MAR.

DR = mem[mem[PC*+PCOffset9]]



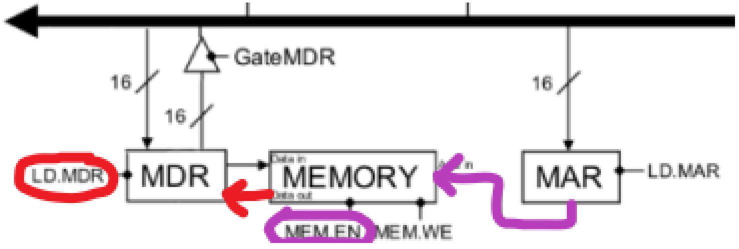
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0	0	1	0	1	0	1	1	1

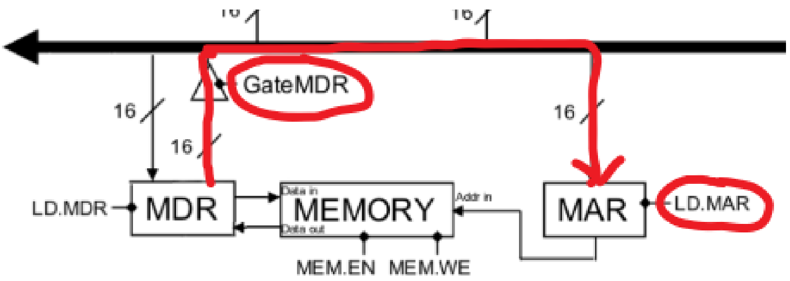
DR PCOffset9

Tracing LDI Cycles 2–4

`DR = mem[mem[PC*+PCOffset9]]`

- Cycle 2 (top): now that we have loaded the first address we want to read from, we can perform that read by enabling MEM.EN but not MEM.WE
- Cycle 3 (bottom): This first value read from memory contains *another* memory address that has the actual data we want to read. Therefore, we want to move that value back into the MAR, so we can read from the new address.
- Cycle 4: Identical to cycle 2! We just want to read from memory again, this time at our new memory address.



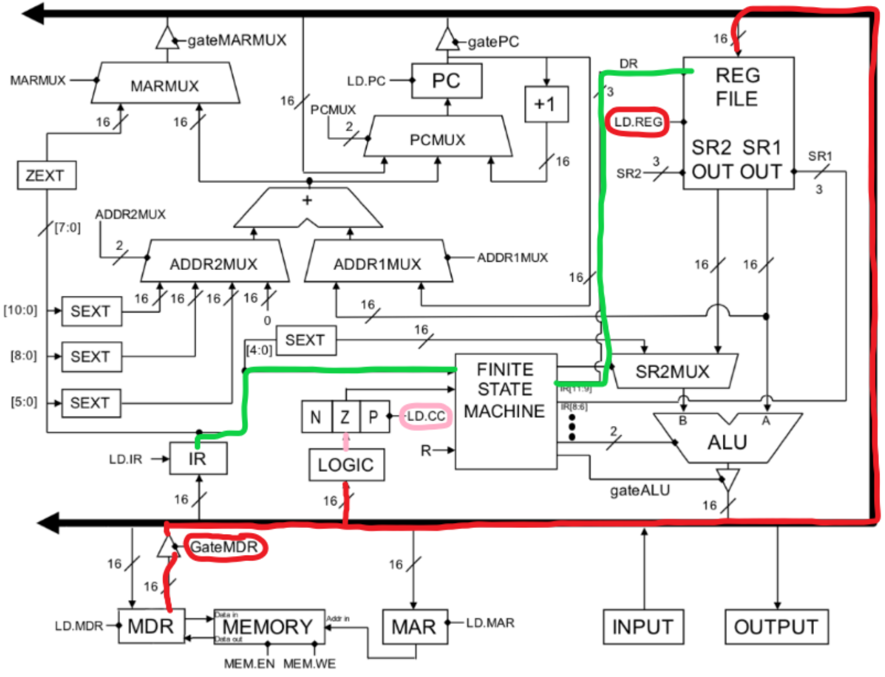


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0	0	1	0	1	0	1	1	1
				DR			PCOffset9								

Tracing LDI Cycle 5

- We now have the data that we wanted to read stored in the MDR.
- First, we must transfer this value to the destination register specified in the instruction.
- Last, but not least, since this is a load instruction, we need to set the condition codes on the value read from memory.

DR = mem[mem[PC*+PCOffset9]]

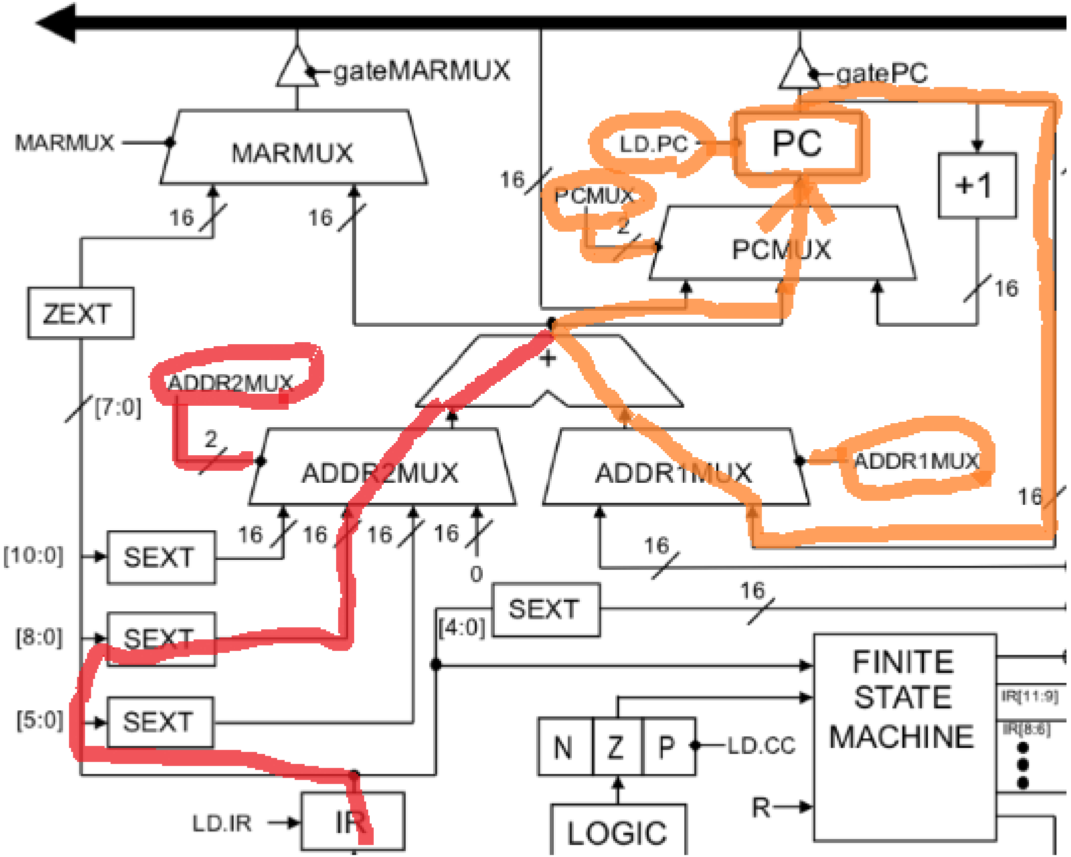


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0	0	1	0	1	0	1	1	1

DR PCOffset9

BR Tracing

- The diagram on the right shows what happens if the FSM determines that the program should branch and modify the PC. If it doesn't branch, nothing happens and the FSM switches to FETCH.
- This is one of several instructions that can overwrite the value in the PC.



LC-3 Assembly Programming

LC-3 Assembly “Pseudo-Ops”

- Not actually instructions, but directions given to the assembler
- **.orig** – tells assembler to put block of code at desired address
  - *.orig x3000*
- **.stringz** – assembler will put a string of characters in memory followed by ‘\0’
  - *.stringz “something”*
- **.blkw** – allocates memory for specified label
  - *.blkw 10*
- **.fill** – puts value in memory location
  - *.fill x4040*
- **.end** – denotes end of block, closing tag for .orig

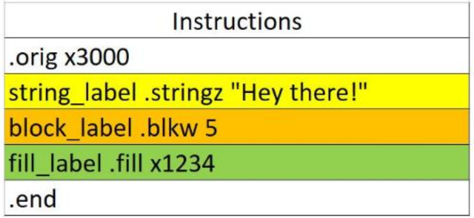
Valid LC-3 Pseudo-Ops

- .orig
- .blkw
- .empty
- .null
- .string
- .stringz
- .trap

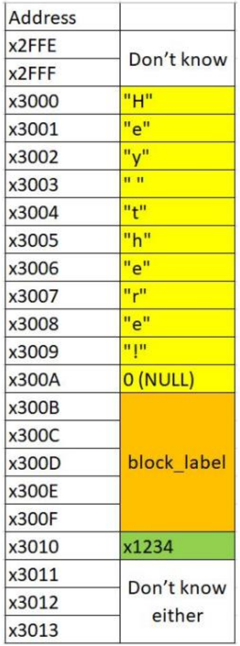
Define the following

- ; (semi-colon)
- Immediates
- Case/Spacing/Indentation
- HALT

Pseudo-Ops Example



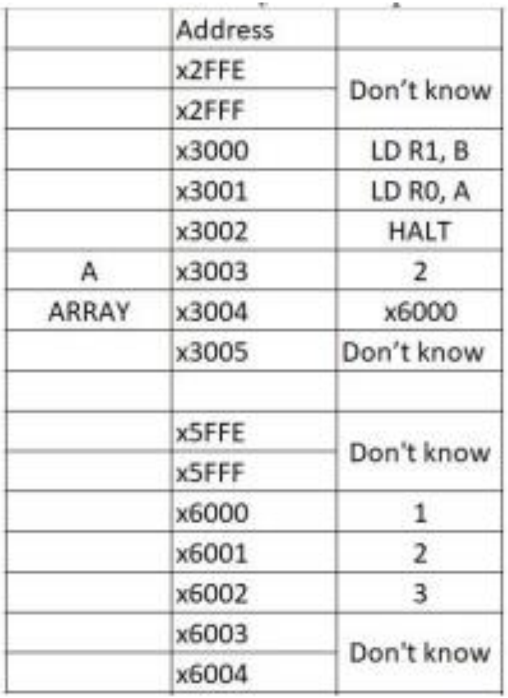
Instructions
.orig x3000
string_label .stringz "Hey there!"
block_label .blkw 5
fill_label .fill x1234
.end



Address	
x2FFE	Don't know
x2FFF	
x3000	"H"
x3001	"e"
x3002	"y"
x3003	" "
x3004	"t"
x3005	"h"
x3006	"e"
x3007	"r"
x3008	"e"
x3009	"!"
x300A	0 (NULL)
x300B	block_label
x300C	
x300D	
x300E	
x300F	
x3010	x1234
x3011	Don't know either
x3012	
x3013	

LC-3 Assembly File Layout

```
.orig x3000
            LD R1, B
            LD R0, A
            HALT
A           .fill 2
ARRAY .fill x6000
.end
.orig x6000
            .fill 1
            .fill 2
            .fill 3
.end
```



	Address	
	x2FFE	Don't know
	x2FFF	
	x3000	LD R1, B
	x3001	LD R0, A
	x3002	HALT
A	x3003	2
ARRAY	x3004	x6000
	x3005	Don't know
	x5FFE	Don't know
	x5FFF	
	x6000	1
	x6001	2
	x6002	3
	x6003	Don't know
	x6004	

Practice

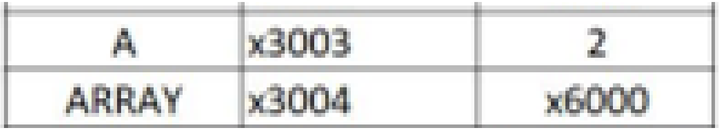
What's the memory address of the final `.fill` in this block of code?

Hint: try plugging it into Complx to visualize how the pseudo-ops affect the memory layout

```
.orig x4000
ARRAY
        .fill 5
        .fill 6
        .fill 7
LEN     .fill 3
SPACE   .blkw x10
HI      .stringz "Hello"

NUM     .fill 15
.end
```

LC-3 Assembly Tricks and Tips



A	x3003	2
ARRAY	x3004	x6000

- Labels
  - Assembler calculates appropriate PC offset and inserts it into machine code for you!
  - Labels “attach” to the operation/pseudo-op that follows them
  - *LD R0, A*
  - *ST R4, ARRAY*
- Branches
  - BRnp will branch on negative or positive
  - BR LOOP = BRnzp LOOP; they are the same instruction
- TRAPs
  - Useful “functions” that aren’t in the instruction set; we will cover them in detail later
  - HALT is an alias for TRAP x25; stops execution of program

LC-3 Assembly Tricks and Tips

- Clearing a register
  - `AND R0, R0, #0  ; don't be the one that writes "ADD" instead on a quiz!`
  - Don't need to clear a register every time you're going to write to it
  - Only the times you would write "x = 0" in pseudocode
- Subtraction
  - A – B = A + (-B)
  - Two's complement negation: flip the bits and add 1
  - `NOT R5, R5`
  - `ADD R5, R5, #1  ; R5 = -R5`
- Copy value from one register to another
  - `ADD R3, R2, #0  ; R3 = R2 + 0`
- Setting the condition codes based on value in a register
  - `ADD R1, R1, #0  ; R1 = R1 + 0`

Know your loads!

What are the values in registers R0-R3 after the code executes?

Step through in Complx to check your answer

Using the wrong load instruction is an easy way to lose points—think carefully which one you want!

```
.orig x3000
    LD   R0, A
    LDI  R1, A
    LEA  R2, A
    LDR  R3, R2, #1
    HALT
.end

.orig x3010
    A .fill x4000
      .fill -8
.end

.orig x4000
    .fill 5
    .fill 6
.end
```

Assembly / Complx Example

- Given two inputs (at labels A and B), calculate A OR B and store it at the label RESULT
  - LC-3 doesn’t have an OR instruction… how can we calculate this?
  - What instructions can we use to read from or write to memory at a label?
- Guide to using Complx Simulator:
  - Files ->LC-3 Resources ->“ComplxMadeSimple.pdf”