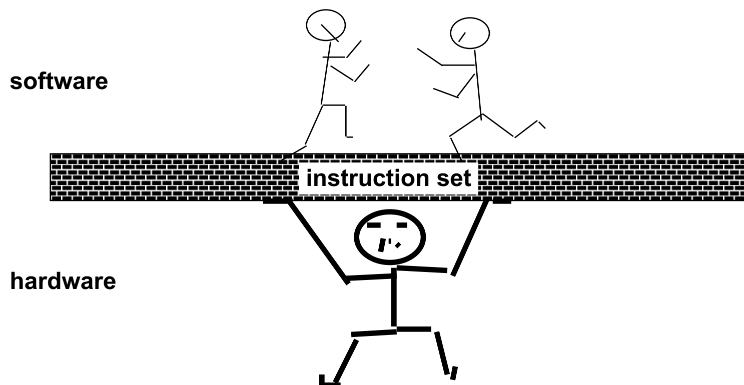
LC3 Architecture: LC3 Instruction Set Architecture (ISA) (Chapter 4,5)

Based on slides © McGraw-Hill
Additional material © 2013 Farmer
Additional material © 2020 Narahari

What is the Hardware/Software Interface ?...ISA



ISA: Types of Instruction

- *1. Operate* Instructions
  - process data (addition, logical operations, etc.)
- *2. Data Movement* Instructions …
  - move data between memory locations and registers.
- *3. Control* Instructions …
  - change the sequence of execution of instructions in the stored program.
    - The default is sequential execution: the PC is incremented by 1 at the start of every Fetch, in preparation for the next one.
    - Control instructions set the PC to a new value during the Execute phase, so the next instruction comes from a different place in the program.
    - This allows us to build control structures such as loops and branches.

3

3

Instruction Cycle

- Six phases of the complete Instruction Cycle
  - Fetch: load IR with instruction from memory
  - Decode: determine action to take (set up inputs for ALU, RAM, etc.)
  - Evaluate address: compute memory address of operands, if any
  - Fetch operands: read operands from memory or registers
  - Execute: carry out instruction
  - Store results: write result to destination (register or memory)

4

4

Changing the Sequence of Instructions

- In the FETCH phase, we increment the Program Counter by 1.
- What if we don't want to always execute the instruction that follows this one?
  - examples: loop, if-then, function call
- Need special instructions that change the contents of the PC.
- These are called *control instructions*.
  - jumps are unconditional -- always change the PC
  - branches are conditional -- change the PC only if some condition is true (e.g., the result of an ADD is zero)

5

5

LC3 Instruction Set Architecture

- The Instruction set architecture (ISA) of the LC3
  - How is each instruction implemented by the control and data paths in the LC3
  - Programming in machine code
  - How are programs executed
    - Memory layout, programs in machine code
- Overview of LC3 microarchitecture
  - How are components connected to build the LC3 processor
    - read Appendix C in detail – required to complete processor design
- Assembly programming
  - Assembly and compiler process
  - Assembly programming with simple programs

6

6

LC-3 Overview: Memory and Registers

- Memory
  - address space: 2^{16} locations (16-bit addresses)
  - addressability: 16 bits
- Registers
  - temporary storage, accessed in a single machine cycle
    - accessing memory generally takes longer than a single cycle
  - eight general-purpose registers: R0 - R7
    - each 16 bits wide
    - how many bits to uniquely identify a register?
  - other registers
    - not directly addressable, but used by (and affected by) instructions
    - PC (program counter), condition codes

LC-3 Overview: Instruction Set

- Opcodes
  - 15 opcodes
  - *Operate* instructions: ADD, AND, NOT
  - *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
  - *Control* instructions: BR, JSR/JSRR, JMP, RTI, TRAP
  - some opcodes set/clear *condition codes*, based on result:
    - N = negative, Z = zero, P = positive (> 0)
- Data Types
  - 16-bit 2's complement integer
- Addressing Modes
  - How is the location of an operand specified?
  - non-memory addresses: *immediate*, *register*
  - memory addresses: *PC-relative*, *indirect*, *base+offset*

Instruction	Opcode	Fields
ADD+	0001	DR, SR1, 0, 00, SR2
ADD+	0001	DR, SR1, 1, imm5
AND+	0101	DR, SR1, 0, 00, SR2
AND+	0101	DR, SR1, 1, imm5
BR	0000	n, z, p, PCoffset9
JMP	1100	000, BaseR, 000000
JSR	0100	1, PCoffset11
JSRR	0100	0, 00, BaseR, 000000
LD+	0010	DR, PCoffset9
LDI+	1010	DR, PCoffset9

+ Indicates instructions that modify condition codes 9

9

Instruction	Opcode	Fields
LDR+	0110	DR, BaseR, offset6
LEA+	1110	DR, PCoffset9
NOT+	1001	DR, SR, 111111
RET	1100	000, 111, 000000
RTI	1000	000000000000
ST	0011	SR, PCoffset9
STI	1011	SR, PCoffset9
STR	0111	SR, BaseR, offset6
TRAP	1111	0000, trapvect8
reserved	1101	

+ Indicates instructions that modify condition codes 10

10

Operate Instructions

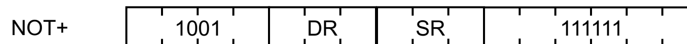
- Only three operations: ADD, AND, NOT
- Source and destination operands are registers
  - These instructions *do not* reference memory.
  - ADD and AND can use "immediate" mode, where one operand is hard-wired into the instruction.
- dataflow diagram associated with each instruction.
  - illustrates *when* and *where* data moves to accomplish the desired operation
  - *used to design the datapath*

11

11

Operate Instructions

- NOT

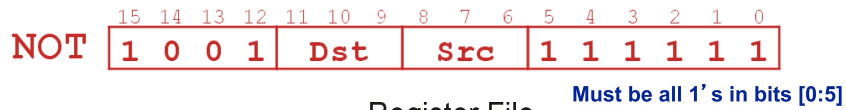


- Addressing mode?
  - Where are the operands
  - DR and SR specify the register number/address
    - 3 bits needed to specify address of 8 registers

12

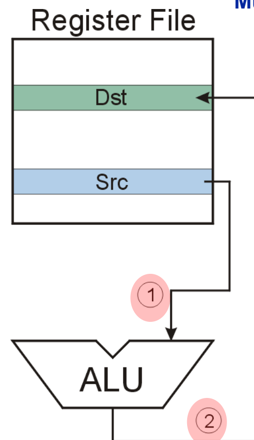
12

NOT (Register)



Must be all 1's in bits [0:5]

If Dst=010, Src=101
R2 = NOT(R3)



Note: Src and Dst could be the same register.

13

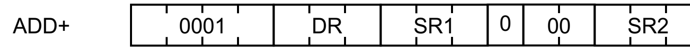
Specifying instruction semantics

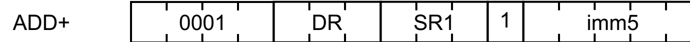
- Is there a formal way to specify the transfers/controls needed to implement each instruction ?
  - The register transfers needed.
- Is there a formal language
  - Can you specify these transfers by writing a program
- Hardware Description Languages (HDL)
  - Verilog, VHDL, …
  - Processor design and specification today is done using a HDL
    - Use software to design hardware!
- Data flow and timing
  - Data transfers labelled at each step/cycle
    - What transfers and between which devices at each cycle

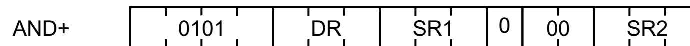
14

Operate Instructions

- ADD, AND







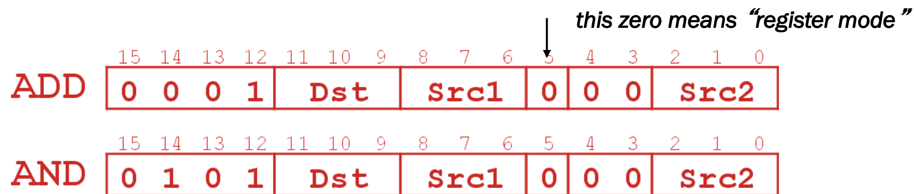
AND+ | 0101 | DR | SR1 | 1 | imm5

- Addressing Mode?
  - Where are the operands
  - Two modes: (1) register and (2) immediate value

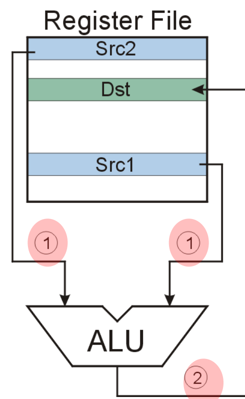
15

15

ADD/AND (Register)



Register mode:
Operands are in registers



If Dst=010, Src1=001, Src2=011

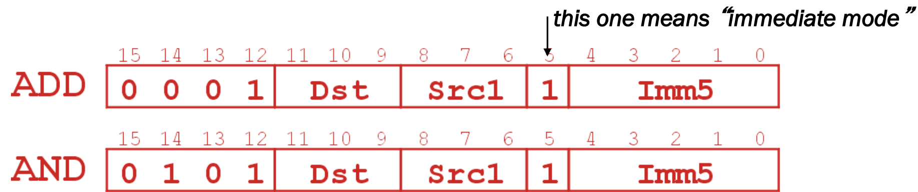
ADD:
Dst= Src1 + Src2
R2= R1 + R3

AND:
Dst= Src1 AND Src2
R2 = R1 AND R3

16

16

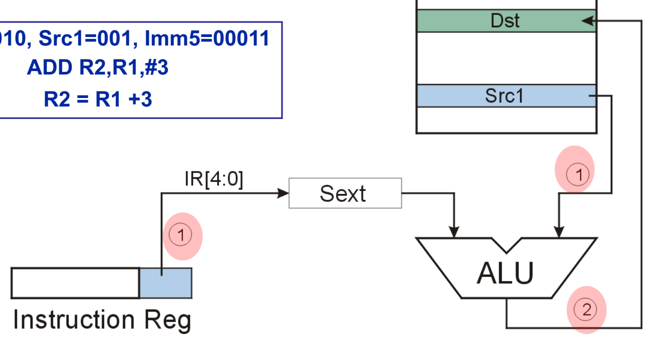
ADD/AND (Immediate)



Note: Immediate field is ***sign-extended****.*

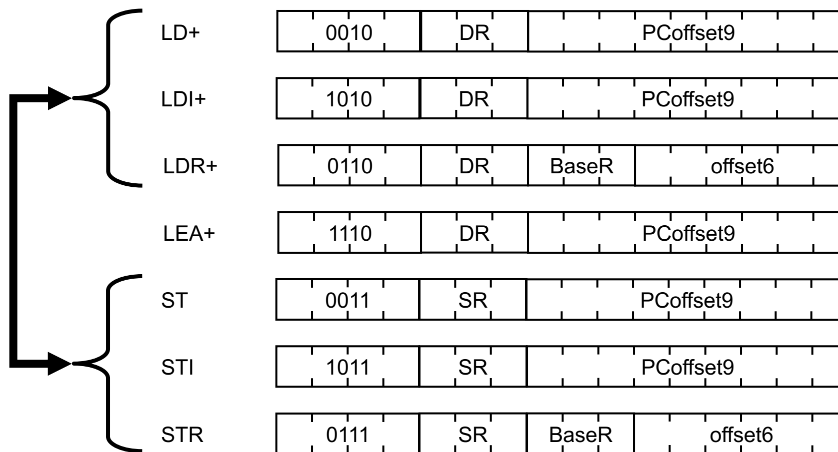
Immediate mode:
One Operand in inst

If Dst=010, Src1=001, Imm5=00011
ADD R2,R1,#3
R2 = R1 +3



17

Data Movement Instructions



18

18

Data Movement Instructions

- GPR ↔ Memory
- GPR ↔ I/O Devices
- GPR ← Memory ???
- Memory ← GPR ???

19

19

Addressing Modes

- Where can operands be found?

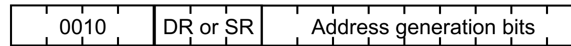
  If in memory, how to compute the address where the operand/variable is stored ?

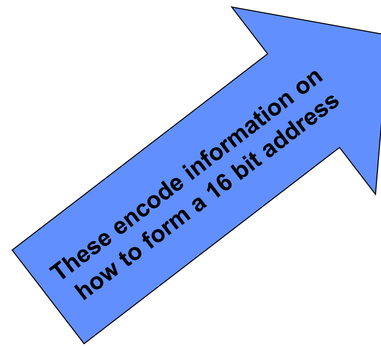
- Manner in which address is computed leads to three different types of Load/Store instructions

20

20

Basic Format





DR is register into which memory contents are read (Load)
SR contains data that has to be written into memory (Store)

21

21

Data Movement Instructions

- Load -- read data from memory to register
  - LD: PC-relative mode
  - LDR: base+offset mode
  - LDI: indirect mode
- Store -- write data from register to memory
  - ST: PC-relative mode
  - STR: base+offset mode
  - STI: indirect mode
- Load effective address -- compute address, save in register
  - LEA: immediate mode
  - *does not access memory*

22

22

PC-Relative Addressing Mode

- Want to specify address directly in the instruction
  - But an address is 16 bits, and so is an instruction!
  - After subtracting 4 bits for opcode and 3 bits for register, we have 9 bits available for address.
- Solution:
  - Use the 9 bits as a *signed offset* from the current PC.

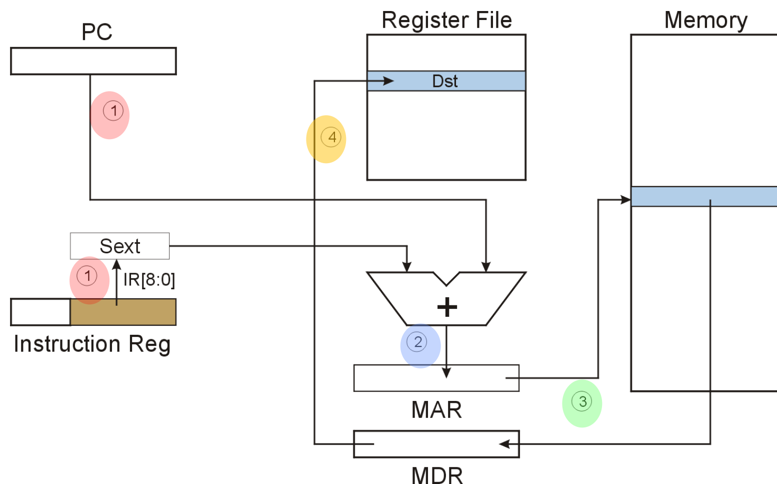
- 9 bits: $-256 \le \text{offset} \le +255$
- Can form any address X, such that: $PC - 256 \le X \le PC + 255$

- Remember that PC is incremented as part of the FETCH phase;
- This is done before the EVALUATE ADDRESS stage.

23

23

LD (PC-Relative)



24

LD – Load PC-relative Addressing

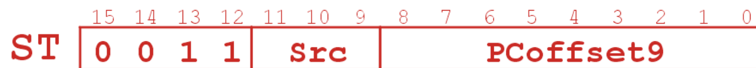
- Suppose instruction is 0010011000000010
- This instruction is stored at address (decimal) 4999
- Ques: What is the result of this instruction ?

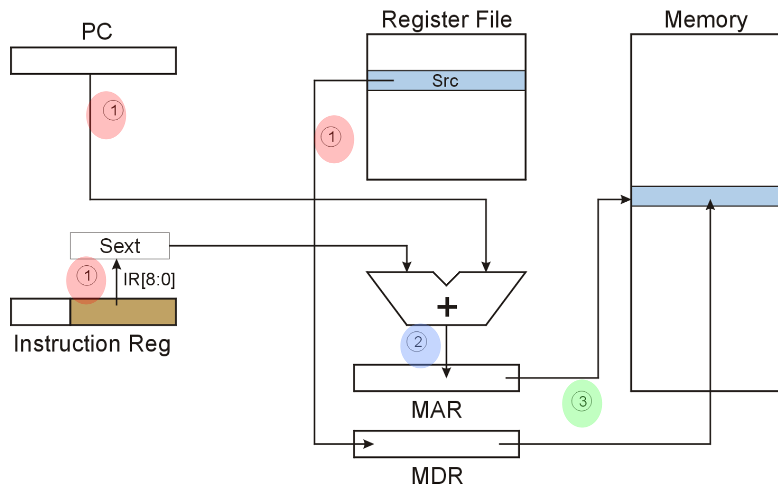
Address	Memory contents
1200	500
5000	400
5001	2000
5002	5001

25

25

ST (PC-Relative)





26

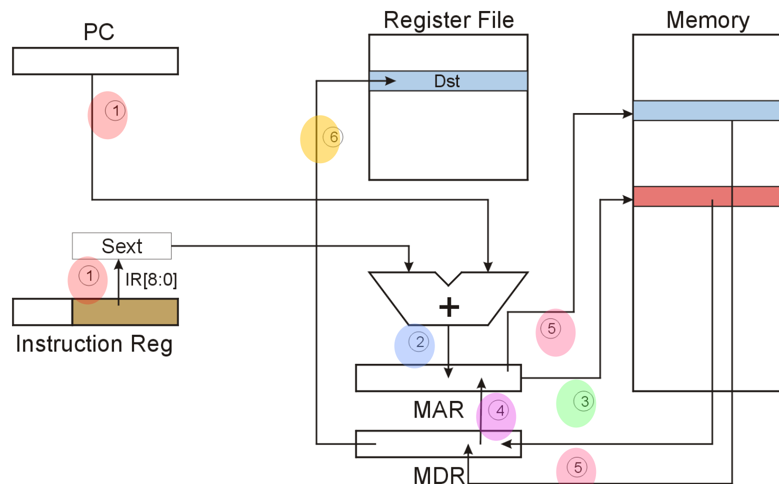
Indirect Addressing Mode

- With PC-relative mode, can only address data within 256 words of the instruction.
  - What about the rest of memory?
- Solution #1:
  - Read address from memory location, then load/store to that address.
- First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.
  - analogy to pointers

27

27

LDI (Indirect)



28

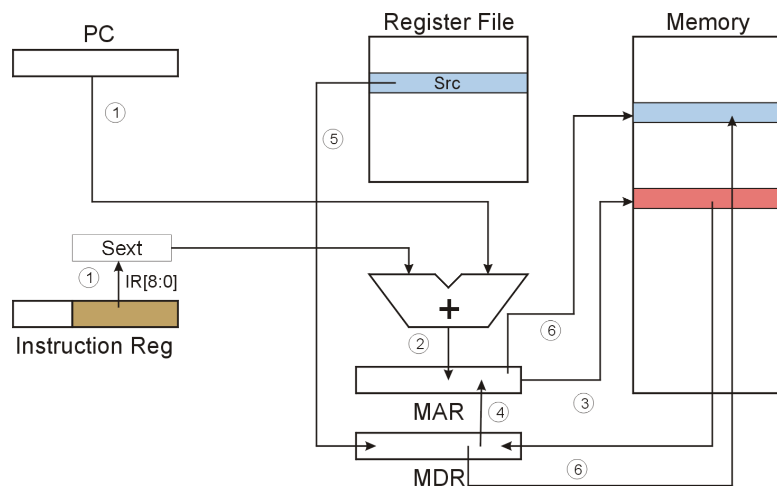
Load Indirect

- Suppose instruction is 1010011000000010
- This instruction is stored at address (decimal) 4999
- What is the result of this instruction ?

Address	Memory contents
1200	500
5000	400
5001	2000
5002	5001

29

STI (Indirect)



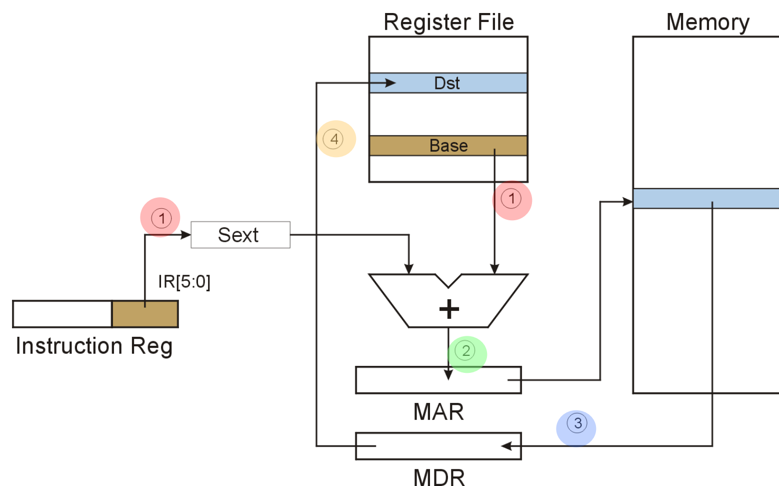
Base + Offset Addressing Mode

- With PC-relative mode, can only address data within 256 words of the instruction.
  - What about the rest of memory?
- Solution #2:
  - Use a register to generate a full 16-bit address.
- 4 bits for opcode, 3 for src/dest register, 3 bits for *base* register -- remaining 6 bits are used as a *signed offset*.
  - Offset is *sign-extended* before adding to base register.

31

31

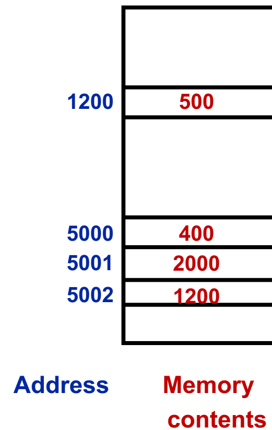
LDR (Base+Offset)



32

Load Register

- Suppose instruction is 0110011010000010
- This instruction is at address 4999
- Suppose Register R2 contains the decimal value 5000
- What is the result ?



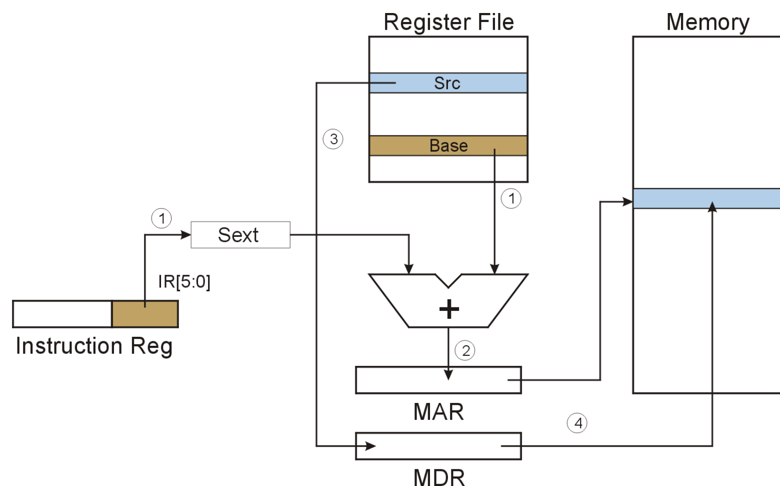
Address Memory contents

33

33

STR (Base+Offset)

STR



34

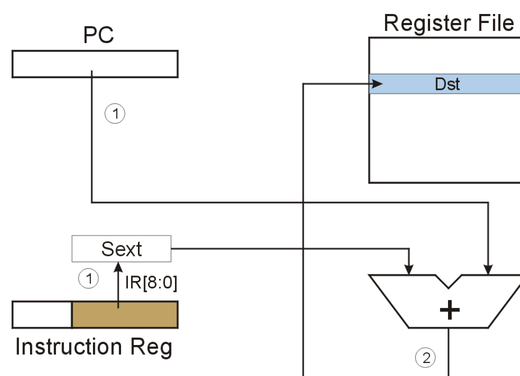
Load Effective Address

- Computes address like PC-relative (PC plus signed offset) and stores the result into a register.

- Note: The *address* is stored in the register, not the contents of the memory location.

35

LEA (Immediate)



Example

Address	Instruction		Comments
	opcode		
x30F6	1 1 1 0	0 0 1 1 1 1 1 1 1 1 0 1	LEA
x30F7	0 0 0 1	0 1 0 0 0 1 1 0 1 1 1 0	ADD imm5.
x30F8	0 0 1 1	0 1 0 1 1 1 1 1 1 0 1 1	ST
x30F9	0 1 0 1	0 1 0 0 1 0 1 0 0 0 0 0	AND imm5
x30FA	0 0 0 1	0 1 0 0 1 0 1 0 0 1 0 1	ADD imm5
x30FB	0 1 1 1	0 1 0 0 0 1 0 0 1 1 1 0	STR
x30FC	1 0 1 0	0 1 1 1 1 1 1 1 0 1 1 1	LDI

37

37

Example

Address	Instruction		Comments
	opcode		
x30F6	1 1 1 0	0 0 1 1 1 1 1 1 1 1 0 1	R1 ← PC – 3 = x30F4
x30F7	0 0 0 1	0 1 0 0 0 1 1 0 1 1 1 0	R2 ← R1 + 14 = x3102
x30F8	0 0 1 1	0 1 0 1 1 1 1 1 1 0 1 1	M[PC - 5] ← R2 M[x30F4] ← x3102
x30F9	0 1 0 1	0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0
x30FA	0 0 0 1	0 1 0 0 1 0 1 0 0 1 0 1	R2 ← R2 + 5 = 5
x30FB	0 1 1 1	0 1 0 0 0 1 0 0 1 1 1 0	M[R1+14] ← R2 M[x3102] ← 5
x30FC	1 0 1 0	0 1 1 1 1 1 1 1 0 1 1 1	R3 ← M[M[x30F4]] R3 ← M[x3102] R3 ← 5

38

38

Control Instructions

- Used to alter the sequence of instructions (by changing the Program Counter)
- Conditional Branch
  - branch is *taken* if a specified condition is true
    - signed offset is added to PC to yield new PC
  - else, the branch is *not taken*
    - PC is not changed, points to the next sequential instruction
- Unconditional Branch (or Jump)
  - always changes the PC
- TRAP
  - changes PC to the address of an OS "service routine"
  - routine will return control to the next instruction (after TRAP)

39

Control Instructions

Instruction	Bits 15–12	Bits 11–0		
BR	0000	n z p	PCoffset9	
JMP	1100	000	BaseR	000000
JSR	0100	1	PCoffset11	
JSRR	0100	0 00	BaseR	000000
RET	1100	000	111	000000
RTI	1000	000000000000		
TRAP	1111	0000	trapvect8	

40

Condition Codes

- LC-3 has three condition code registers:
  - N -- negative
  - Z -- zero
  - P -- positive (greater than zero)
- Set by any instruction that writes a value to a register (ADD, AND, NOT, LD, LDR, LDI, LEA)
- Exactly one will be set at all times
  - ***Based on the last instruction that altered a register***

41

41

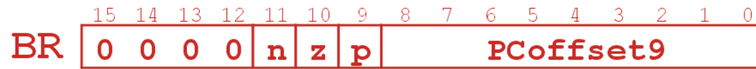
Branch Instruction

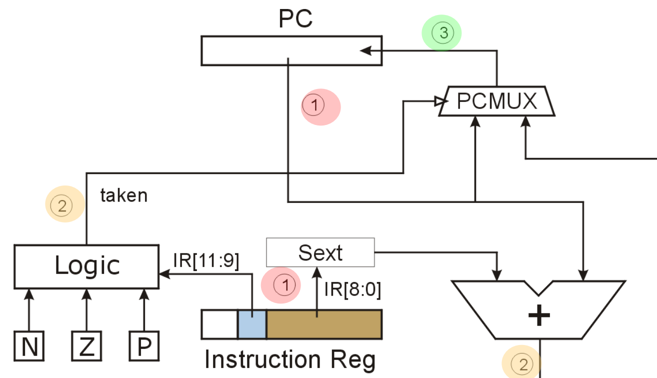
- Branch specifies one or more condition codes.
- If the set bit is specified, the branch is taken.
  - PC-relative addressing:
    target address is computed by adding signed offset (IR[8:0]) to current PC.
  - Note: PC has already been incremented by FETCH stage.
  - Note: Target must be within 256 words of BR instruction.
- If the branch is not taken,
  the next sequential instruction is executed.

42

42

BR (PC-Relative)





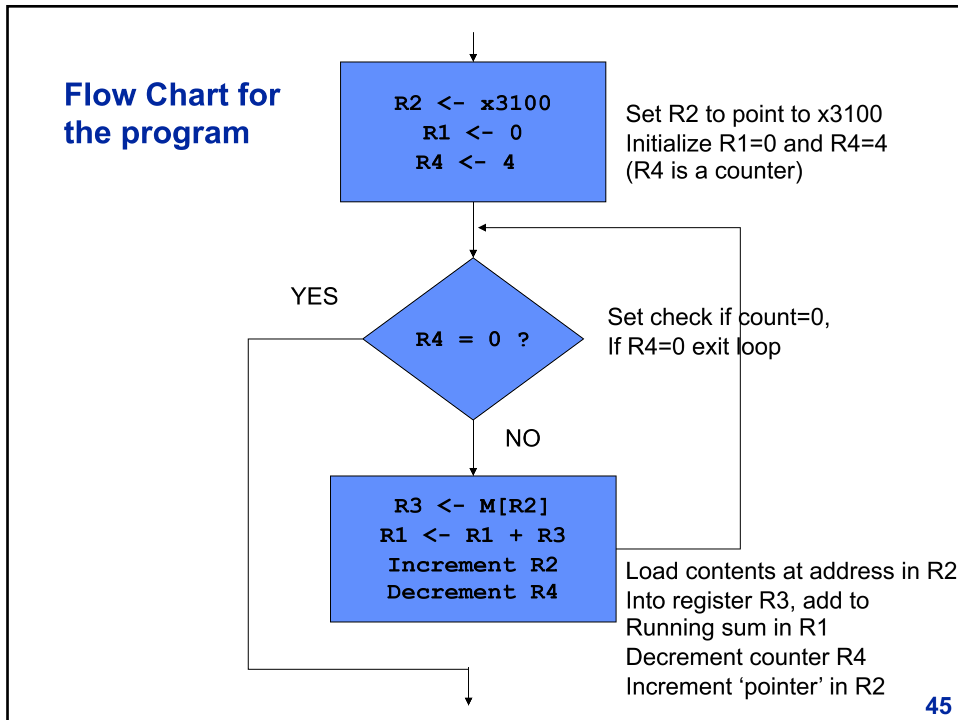
What happens if bits [11:9] are all zero? All one?

Using Branch Instructions

- Compute sum of 4 integers.

Numbers start at location x3100. Program starts at location x3000.

- Add numbers from location x3100 to x311B
- Store first address in R2
- R4 has “counter” – counts down from 4 to 0
- R1 will store the running Sum



45

Program

```
x3000 R2 <- x3100
x3001 R4 <- 0
x3002 R1 <- 0
x3003 R4 <- 4
x3004 BRz x300A   /* if R4=0 exit loop */
x3005 R3 <- M[R2]
x3006 R1 <- R1 + R3
x3007 R2 <- R2 + 1
x3008 R4 <- R4 - 1
x3009 BRnzp x3004. /* repeat loop */
x300A Halt       /* end of loop */
```

46

Program

		opcode	
x3000 R2 <- x3100	LEA	1110	010011111111
x3001 R1 <- 0	AND	0101	001001111100000
x3002 R4 <- 0	AND	0101	100010100000
x3003 R4 <- 12	ADD	0001	100010100100
x3004 BRz x300A	BRz	0000	010000000101
x3005 R3 <- M[R2]	LDR	0110	011010000000
x3006 R1 <- R1 + R3	ADD	0001	001001000011
x3007 R2 <- R2 + 1	ADD	0001	010010100001
x3008 R4 <- R4 - 1	ADD	0001	100100111111
x3009 BRnzp x3004	BRnzp	0000	111111111010

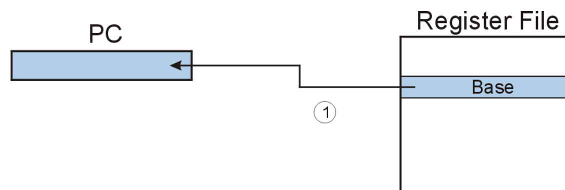
dest Immediate
source

47

JMP (Register)

- Jump is an unconditional branch -- *always* taken.
  - Target address is the contents of a register.
  - Allows any target address.

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



48

48

TRAP Instruction

- Modern computers contain hardware and software protection schemes to prevent user programs from accidentally (or maliciously) interfering with proper system function.
- Suffice it to say, we need a way to communicate with the operating system

49

TRAP

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							

•Calls a service routine, identified by 8-bit "trap vector."

vector	*routine*
x23	**input a character from the keyboard**
x21	**output a character to the monitor**
x25	**halt the program**

•When routine is done,
PC is set to the instruction following TRAP.

•(We'll talk about how this works later.)

50

The LC-3 ISA: summary

- 16 bit instructions and data
- 2's complement data type
- Operate/ALU instructions: ADD, NOT, AND
- Data movement Inst: Load and Store
  - Addressing mode: PC-relative, Indirect, Register/Base+Offset,Immediate
- Transfer of control instructions
  - Branch – using condition code registers
  - Jump – unconditional branch
  - Traps, Subroutine calls – discuss later
- Now we look at the LC3 datapath and controller design

51

51

Taking stock: what we have now

- Datatypes of machines: Number Representation
  - 2's complement integers, Floating point
  - Arithmetic on 2's complement
  - Logic operations
- Digital logic: devices to build the circuits
  - CMOS transistor is the starting point
  - Basic logic gates: AND, OR, NOT, NAND, etc.
  - Combinational logic 'blocks': MUX, Decoder, PLA
  - Sequential Logic: storage element, finite state machines
  - Putting it all together to build a simple processor- LC3
- Von Neumann Model of computing
- Instruction set architecture (ISA) of LC3
  - Instructions of a processor – how program execution takes place
  - Addressing modes to data movement, branches, operations
  - Encoding an LC3 instruction
- Next question: how to program the processor
  - Using instructions in the ISA

52

52