Computer Organization I

LC-3 Instruction Sets
Assembly Programming

June 9, 2015

Outline

- Content: Chapter 5.2-5.6, 6.1-6.2
- LC-3 Instructions
- Operate
  - ADD, AND, OR
- Data Movement
  - LD/ST, LDR/STR, *LDI/STI,* LEA
- Control
  - BR, TRAP, *JMP*, *JSR, JSRR, RET, RTI*
- LC-3 Assembly Programming
- LC-3 Editor and Simulator

Recall UTM

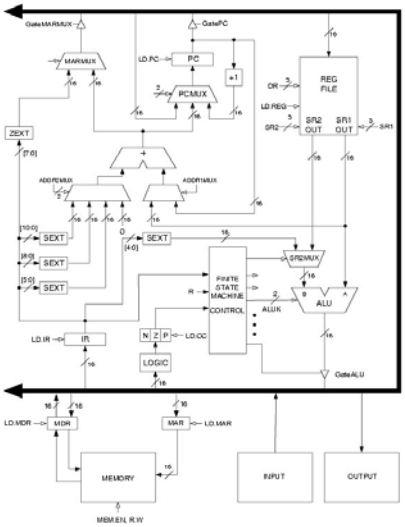
- An UTM is still a TM
- UTM Input – TM + TM's Input – Simulation
- Virtual TM inside UTM – How to describe?
- Simulation through Programming
- Programming through Instructions
  - Tape – Memory – LD / ST
  - Movement – Control – (PC), JMP, JZ
  - Content – Operate – ADD/AND/SHL

Instruction Set Architecture

- ISA = All of the programmer-visible components and operations of the computer
  - Memory organization
    - address space -- how may locations can be addressed?
    - addressibility -- how many bits per location?
  - Register set
    - how many? what size? how are they used?
  - Instruction set
    - Opcodes, data types, addressing modes
- ISA provides all information needed for someone that wants to write a program in machine language
- Intel vs. {ARM, PowerPC, SPARC}, Intel vs. AMD
- ISA so far – Intel, ARM, TinyCPU, LC-3

LC-3 Data Path Revisited

Filled arrow
= info to be processed.
Unfilled arrow
= control signal.



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
  - 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 (N,Z,P)

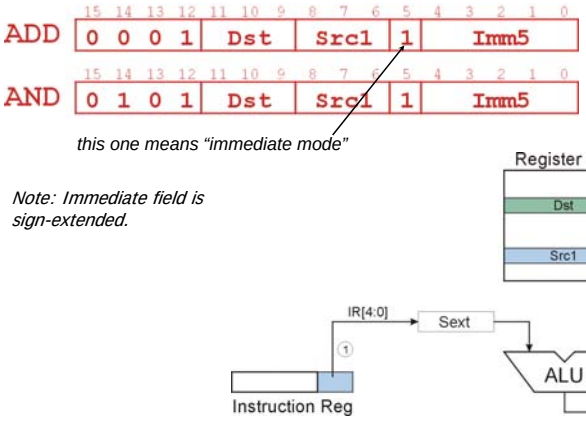
LC-3 Overview: Instruction Set

- Opcodes –15 opcodes
  - Operate instructions: ADD, AND, NOT
  - Data movement instructions: LD, ST, LDR, STR, LEA
    LDI, *STI*
  - Control instructions: BR, *JSR/JSRR*, *JMP*, *RET, 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 (5)
  - How is the location of an operand specified?
  - Non-memory addresses: *immediate*, *register*
  - Memory addresses: *PC-relative*, *indirect*, *base+offset*

Operate Instructions

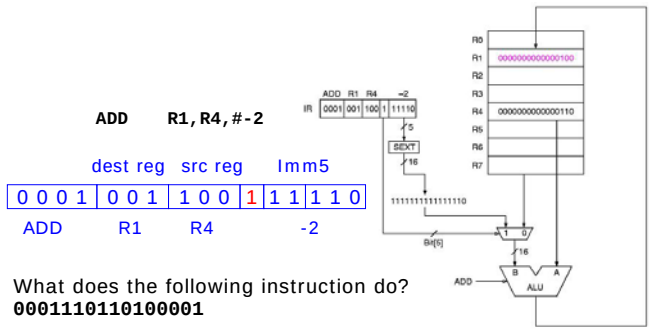
- Only three operations:
  - ADD (0001)
  - AND (0101)
  - NOT (1001)
- Source and destination operands are registers
  - These instructions *do not* reference memory.
  - ADD and AND can use "immediate" mode, where one operand is hard-wired into the instruction.
- Will show dataflow diagram with each instruction.
  - Illustrates *when* and *where* data moves to accomplish the desired operation

ADD/AND (Immediate)



Note: Immediate field is sign-extended.

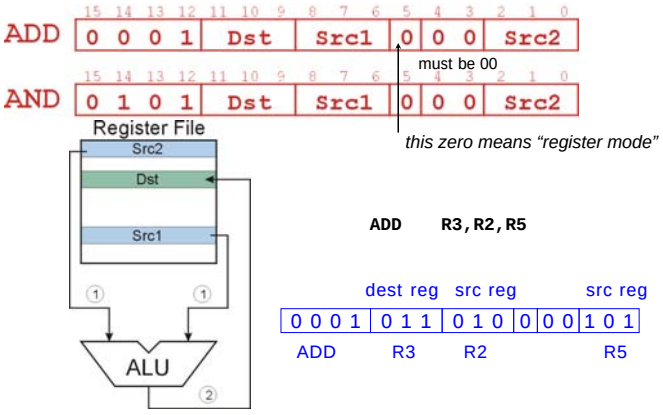
ADD (Immediate)



What does the following instruction do?
0001110110100001

How to perform "R2 ← R0-R1"?

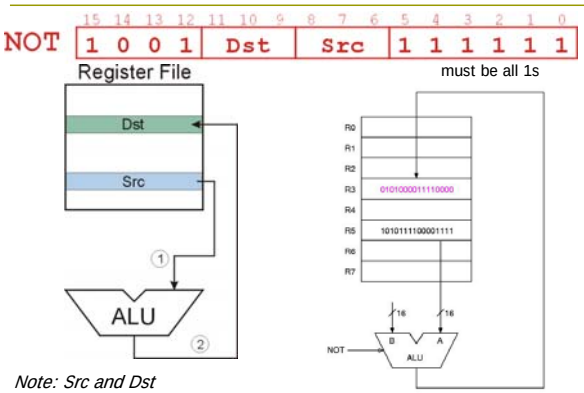
ADD/AND (Register)



ADD R3,R2,R5

dest reg	src reg		src reg
0 0 0 1	0 1 1	0 1 0	0 0 0 1 0 1
ADD	R3	R2	R5

NOT (Register)



Note: Src and Dst could be the same register.

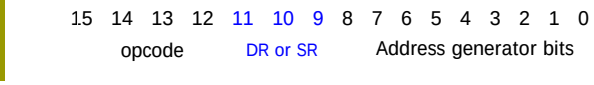
1001011101111111 NOT R3,R5

Using Operate Instructions

- With only ADD, AND, NOT…
  - How do we subtract?
  - How do we OR?
  - How do we copy from one register to another?
  - How do we initialize a register to zero?
  - How do we set a value to a register?

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*



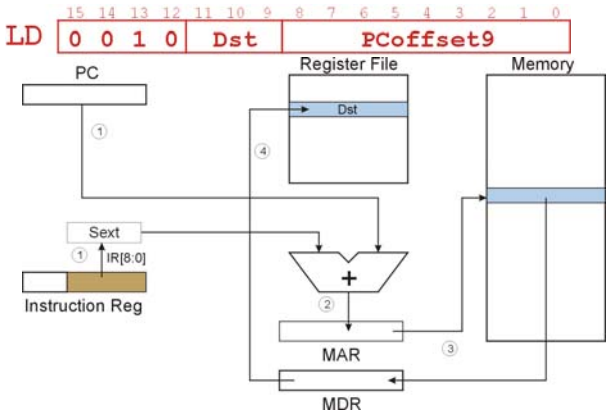
PC-Relative Addressing Mode

- 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:

$$PC - 256 \le X \le PC + 255$$

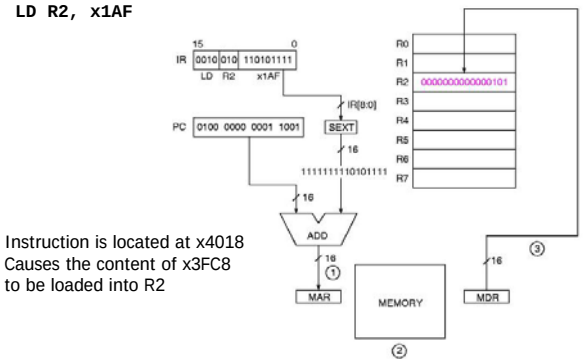
  - Data must be less than 255 bytes away from code

LD (PC-Relative)



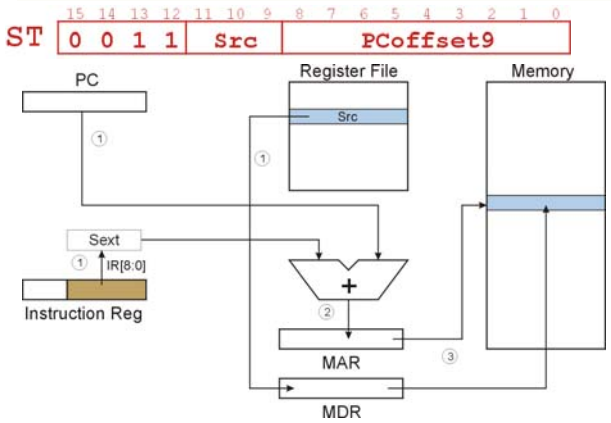
LD Data Path

LD R2, x1AF



Instruction is located at x4018
Causes the content of x3FC8
to be loaded into R2

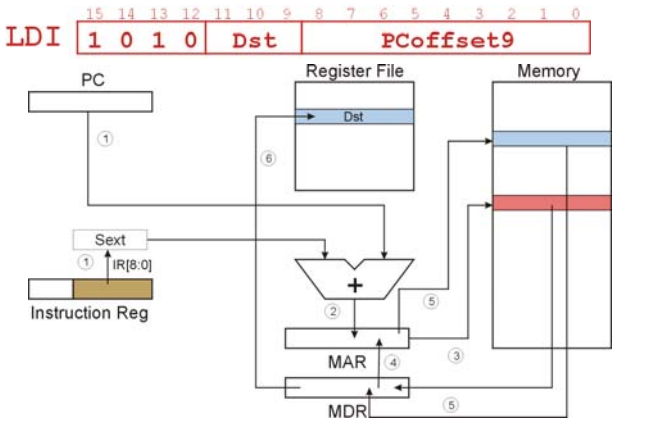
ST (PC-Relative)



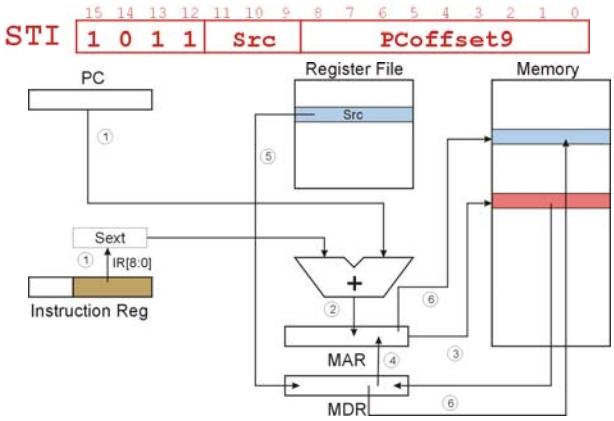
Indirect Addressing Mode

- 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.
  - Since memory content is 16 bits, it can address all 16-bit memory addresses.

LDI (Indirect)

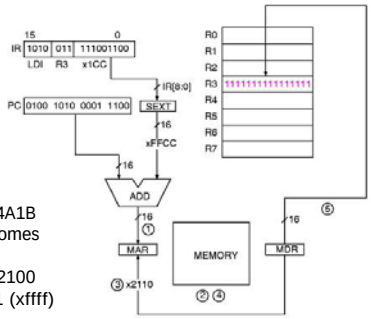


STI (Indirect)



LDI Data Path

```
LDI R3, x1CC
```

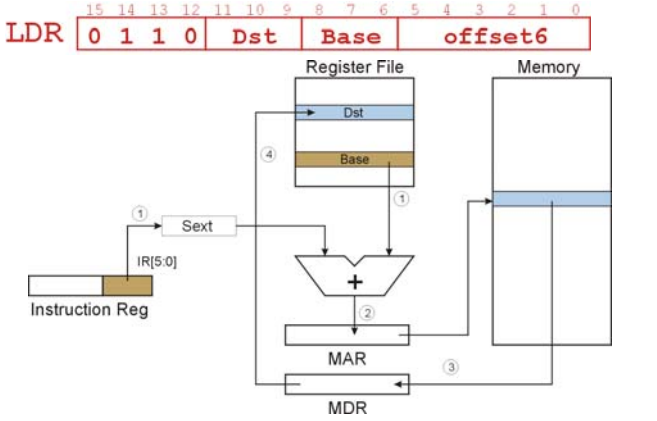


1) Instruction is located at x4A1B
2) The content of x49E8 becomes the memory address
3) The content of x49E8 is x2100
4) The content of x2100 is -1 (xffff)
5) –1 is loaded into R3

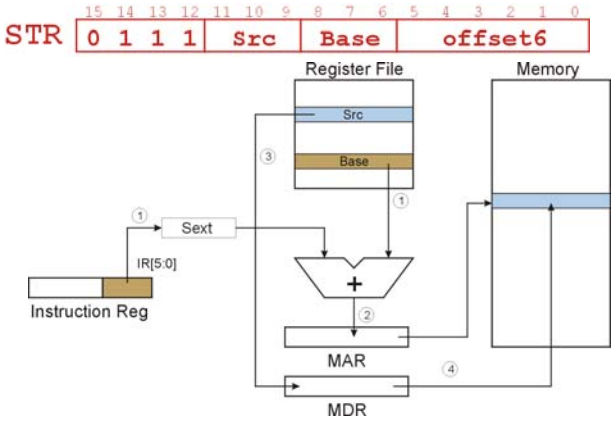
Base + Offset Addressing Mode

- With PC-relative mode, can only address data within 256 words of the instruction.
  - What about the rest of memory?
  - Solution #1 – Indirect Mode
- 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.
  - The range of offset is between -32 and +31

LDR (Base+Offset)

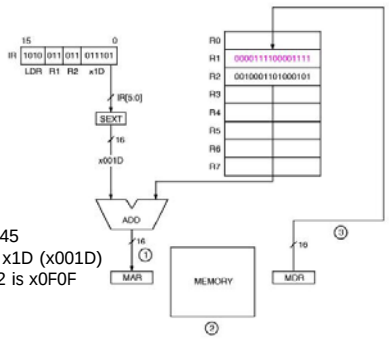


STR (Base+Offset)



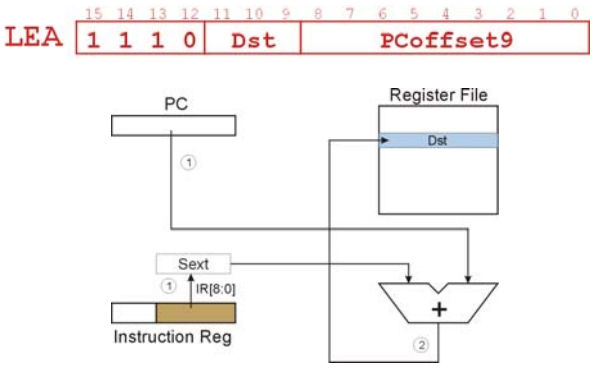
LDR Data Path

```
LDR R1, R2, x1D
```



1) The content of R2 is x2345
2) Add signed ext. value of x1D (x001D)
3) Assume content of x2362 is x0F0F
4) x0F0F is loaded into R1

LEA (Immediate)

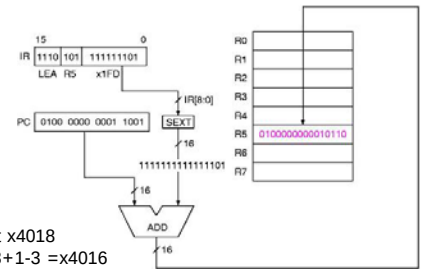


Load Effective Address

- The last addressing mode is the immediate address mode.
- It is used only with the load effective address (LEA)
- 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.

LEA Data Path

```
LEA R5, # -3
```



1) Instruction is located at x4018
2) R5 has the value x4018+1-3 =x4016

Control Instructions

- Used to alter the sequence of instructions (by changing the Program Counter – PC)
- 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)

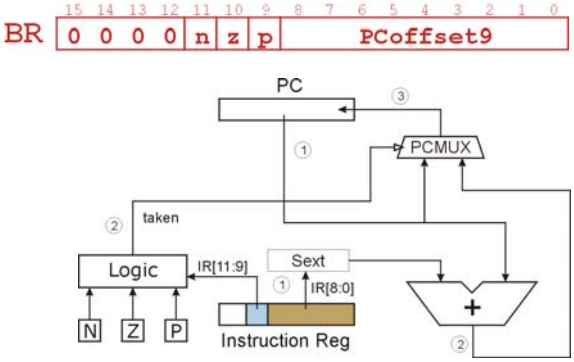
Condition Codes

- LC-3 has three condition code registers:
  - N – negative
  - Z – zero
  - P – positive (greater than zero)
- The conditional codes are used by the conditional branch instructions to determine whether to change the instruction flow
- 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

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 made 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.
- How about branch in higher level languages?

BR (PC-Relative)

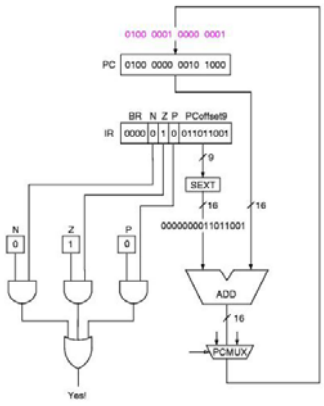


What happens if bits [11:9] are all zero? All one?

BR Data Path

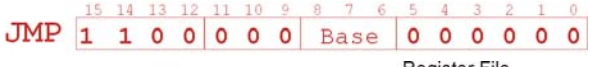
BRz x0D9

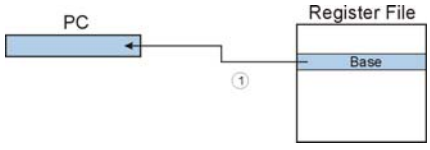
1) The instruction is located at x4027
2) Next instruction executed would be x4101 not x4028



JMP (Register)

- Jump is an unconditional branch – always taken.
  - Target address is the contents of a register.
  - Allows any target address.
- Jump in higher level languages?





1 1 0 0 | 0 0 0 | 0 1 0 | 0 | 0 0 | 0 0 0 **JMP R2**

TRAP

- 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.)

Human-Readable Machine Language

- Computers like ones and zeros…

```
0001110010000110
```

- Humans like symbols…

```
ADD    R6,R2,R6      ; increment index reg.
```

- Assembler is a program that turns symbols into machine instructions.
  - ISA-specific: close correspondence between symbols and instruction set
    - mnemonics for opcodes
    - labels for memory locations
  - additional operations for allocating storage and initializing data

An Assembly Language Program

```
;
; Program to multiply a number by the constant 6
;
        .ORIG  x3050
        LD     R1, SIX
        LD     R2, NUMBER
        AND    R3, R3, #0     ; Clear R3.  It will
                              ; contain the product.
; The inner loop
;
AGAIN   ADD    R3, R3, R2
        ADD    R1, R1, #-1    ; R1 keeps track of
        BRp    AGAIN          ; the iteration.
;
        HALT
;
NUMBER  .BLKW  1
SIX     .FILL  x0006
;
        .END
```

LC-3 Assembly Language Syntax

- Each line of a program is one of the following:
  1. an instruction
  2. an assember directive (or pseudo-op)
  3. a comment
- Whitespace (between symbols) and case are ignored.
- Comments (beginning with ";") are also ignored.
- An instruction has the following format:

```
LABEL          OPCODE OPERANDS ; COMMENTS
```

optional *mandatory*

Opcodes and Operands

- Opcodes
  - Reserved symbols that correspond to LC-3 instructions, listed in Appendix A
  - ex: **ADD, AND, LD, LDR, …**
- Operands
  - Registers -- specified by Rn, where n is the register number
  - Numbers -- indicated by # (decimal) or x (hex)
  - Label -- symbolic name of memory location separated by comma
  - Number, order, and type correspond to instruction format, ex:

```
ADD    R1,R1,R3
ADD    R1,R1,#3
LD     R6,NUMBER
BRz    LOOP
```

Labels and Comments

- Label
  - Placed at the beginning of the line
  - Assigns a symbolic name to the address corresponding to line, ex:

```
LOOP    ADD    R1,R1,#-1
        BRp    LOOP
```

- Comment
  - Anything after a semicolon is a comment
  - Ignored by assembler
  - Used by humans to document/understand programs
  - Tips for useful comments:
    - Avoid restating the obvious, as "decrement R1"
    - Provide additional insight, as in "accumulate product in R6"
    - Use comments to separate pieces of program

Assembler Directives

- Pseudo-operations
  - Do not refer to operations executed by program
  - Used by assembler
  - Look like instruction, but "opcode" starts with dot

Opcode	*Operand*	*Meaning*
.ORIG	address	starting address of program
.END		end of program
.BLKW	n	allocate n words of storage
.FILL	n	allocate one word, initialize with value n
.STRINGZ	n-character string	allocate n+1 locations, initialize w/characters and null terminator

Trap Codes

- LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them.

Code	*Equivalent*	*Description*
HALT	TRAP x25	Halt execution and print message to console.
IN	TRAP x23	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
OUT	TRAP x21	Write one character (in R0[7:0]) to console.
GETC	TRAP x20	Read one character from keyboard. Character stored in R0[7:0].
PUTS	TRAP x22	Write null-terminated string to console. Address of string is in R0.

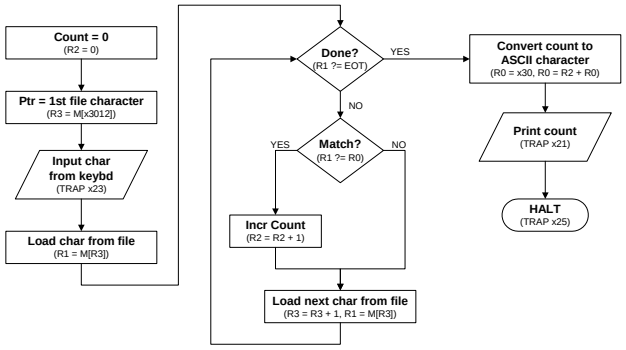
Style Guidelines

- Use the following style guidelines to improve the readability and understandability of your programs:

1. Provide a program header, with author's name, date, etc., and purpose of program.
2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
3. Use comments to explain what each register does.
4. Give explanatory comment for most instructions.
5. Use meaningful symbolic names.
   - Mixed upper and lower case for readability.
   - ASCIItoBinary, InputRoutine, SaveR1
6. Provide comments between program sections.
7. Each line must fit on the page -- no wraparound or truncations.
   - Long statements split in aesthetically pleasing manner.

*Sample Program

- Count the occurrences of a character in a file.
  Remember this?



*Char Count in Assembly Language (1)

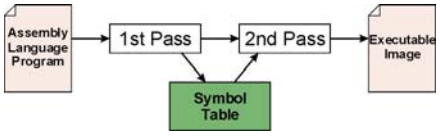
```
;
; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.
;
;
; Initialization
;
        .ORIG   x3000
        AND     R2, R2, #0      ; R2 is counter, initially 0
        LD      R3, PTR         ; R3 is pointer to characters
        GETC                    ; R0 gets character input
        LDR     R1, R3, #0      ; R1 gets first character
;
; Test character for end of file
;
TEST    ADD     R4, R1, #-4     ; Test for EOT (ASCII x04)
        BRz     OUTPUT          ; If done, prepare the output
;
; Test character for match.  If a match, increment count.
;
```

*Char Count in Assembly Language (2)

```
        NOT     R1, R1
        ADD     R1, R1, R0  ; If match, R1 = xFFFF
        NOT     R1, R1      ; If match, R1 = x0000
        BRnp    GETCHAR     ; If no match, do not increment
        ADD     R2, R2, #1
;
; Get next character from file.
;
GETCHAR ADD     R3, R3, #1  ; Point to next character.
        LDR     R1, R3, #0  ; R1 gets next char to test
        BRnzp   TEST
;
; Output the count.
;
OUTPUT  LD      R0, ASCII   ; Load the ASCII template
        ADD     R0, R0, R2  ; Covert binary count to ASCII
        OUT                 ; ASCII code in R0 is displayed.
        HALT                ; Halt machine
;
; Storage for pointer and ASCII template
;
ASCII   .FILL   x0030
PTR     .FILL   x4000
        .END
```

*Assembly Process

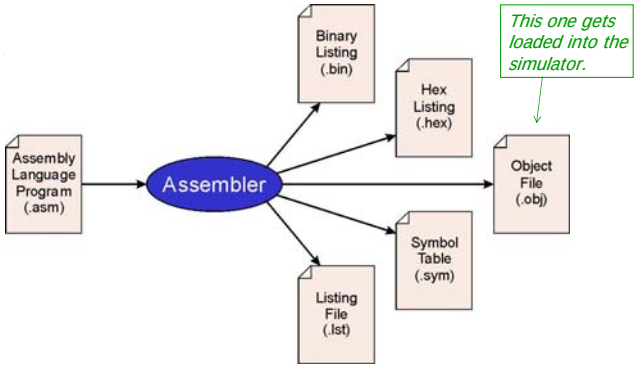
- Convert assembly language file (.asm) into an executable file (.obj) for the LC-3 simulator.



- First Pass:
  - Scan program file
  - Find all labels and calculate the corresponding addresses;this is called the symbol table
- Second Pass:
  - Convert instructions to machine language, using information from symbol table

LC-3 Assembler

- Using "assemble" (Unix) or LC3Edit



Object File Format

- LC-3 object file contains
  - Starting address (location where program must be loaded), followed by…
  - Machine instructions
- Example
  - Beginning of "count character" object file looks like this:

```
0011000000000000   .ORIG x3000
0101010010100000   AND R2, R2, #0
0010011000010001   LD R3, PTR
1111000000100011   TRAP x23
.
.
.
```

Solving Problems using a Computer

- Methodologies for creating computer programs that perform a desired function.
- Problem Solving
  - How do we figure out what to tell the computer to do?
  - Convert problem statement into algorithm, using *stepwise refinement*.
  - Convert algorithm into LC-3 machine instructions.
- Debugging
  - How do we figure out why it didn't work?
  - Examining registers and memory, setting breakpoints, etc.

Time spent on the first can reduce time spent on the second!

Stepwise Refinement

- Also known as systematic decomposition.
- Start with problem statement:

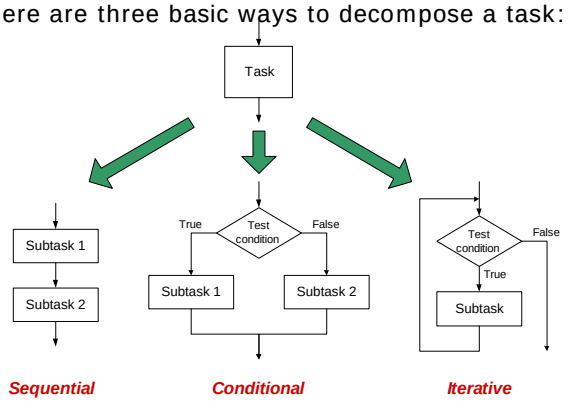
  "We wish to count the number of occurrences of a character in a file. The character in question is to be input from the keyboard; the result is to be displayed on the monitor."
- Decompose task into a few simpler subtasks.
- Decompose each subtask into smaller subtasks,and these into even smaller subtasks, etc.... until you get to the machine instruction level.

Problem Statement

- Because problem statements are written in English, they are sometimes ambiguous and/or incomplete.
  - Where is "file" located? How big is it, or how do I know when I've reached the end?
  - How should final count be printed? A decimal number?
  - If the character is a letter, should I count both upper-case and lower-case occurrences?
- How do you resolve these issues?
  - Ask the person who wants the problem solved, or
  - Make a decision and document it.

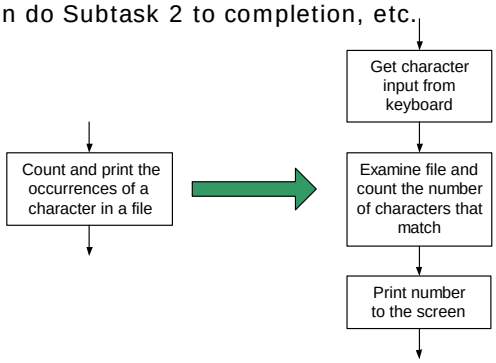
Three Basic Constructs

- There are three basic ways to decompose a task:



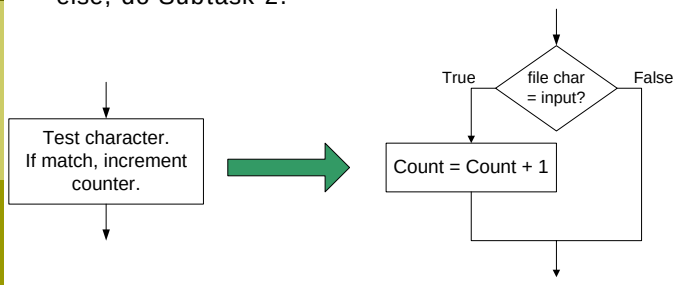
Sequential

- Do Subtask 1 to completion, then do Subtask 2 to completion, etc.



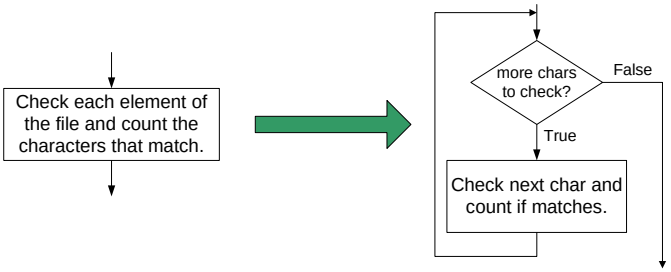
Conditional

- If condition is true, do Subtask 1; else, do Subtask 2.



Iterative

- Do Subtask over and over, as long as the test condition is true.



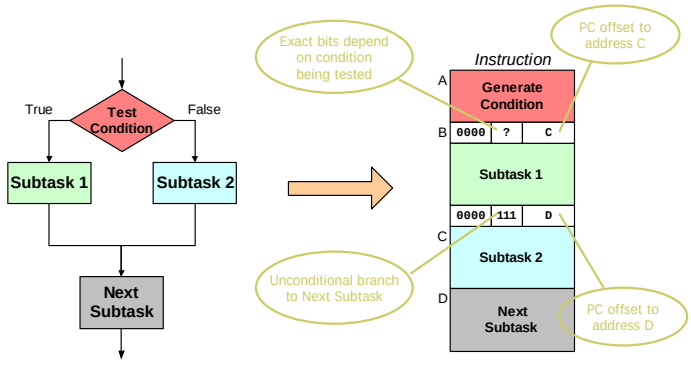
Problem Solving Skills

- Learn to convert problem statement into step-by-step description of subtasks.
  - Like a puzzle, or a "word problem" from grammar school math.
    - What is the starting state of the system?
    - What is the desired ending state?
    - How do we move from one state to another?
  - Recognize English words that correlate to three basic constructs:
    - "do A then do B" ⇒ sequential
    - "if G, then do H" ⇒ conditional
    - "for each X, do Y" ⇒ iterative
    - "do Z until W" ⇒ iterative

LC-3 Control Instructions

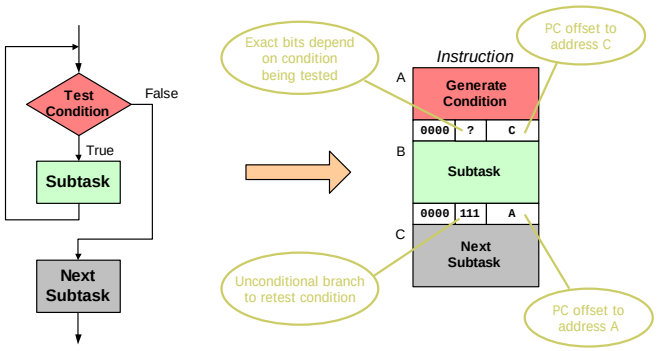
- How do we use LC-3 instructions to encode the three basic constructs?
- Sequential
  - Instructions naturally flow from one to the next, so no special instruction needed to go from one sequential subtask to the next.
- Conditional and Iterative
  - Create code that converts condition into N, Z, or P. Example:
    Condition: "Is R0 = R1?"
    Code: Subtract R1 from R0; if equal, Z bit will be set.
  - Then use BR instruction to transfer control to the proper subtask.

Code for Conditional



Assuming all addresses are close enough that PC-relative branch can be used.

Code for Iteration

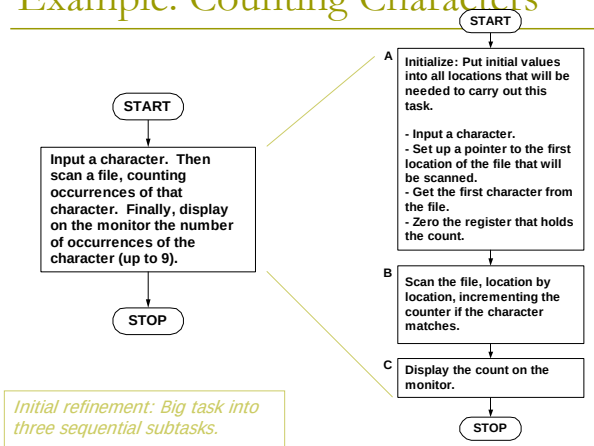


Assuming all addresses are on the same page.

Example 5.5 Revisited

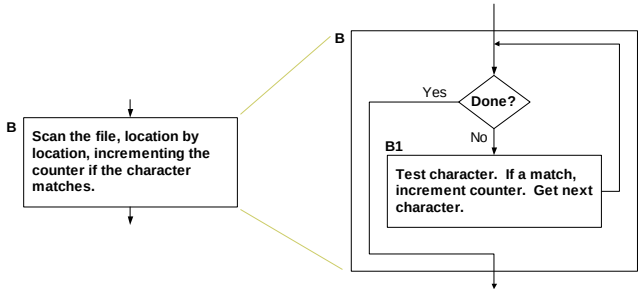
- Count the occurrences of a character in a file
  - Program begins at location x3000
  - Read character from keyboard
  - Load each character from a "file"
    - File is a sequence of memory locations
    - Starting address of file is stored in the memory location immediately after the program
  - If file character equals input character, increment counter
  - End of file is indicated by a special ASCII value: EOT (x04)
  - At the end, print the number of characters and halt (assume there will be less than 10 occurrences of the character)
- A special character used to indicate the end of a sequence is often called a sentinel.
  - Useful when you don't know ahead of time how many times to execute a loop.

Example: Counting Characters



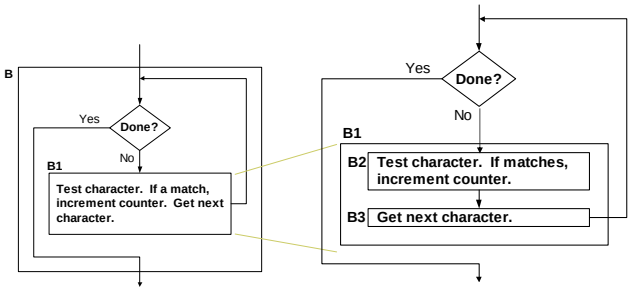
Initial refinement: Big task into three sequential subtasks.

Refining B



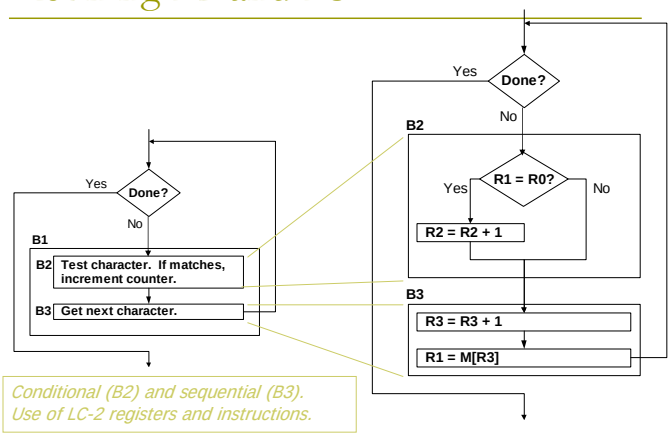
Refining B into iterative construct.

Refining B1



Refining B1 into sequential subtasks.

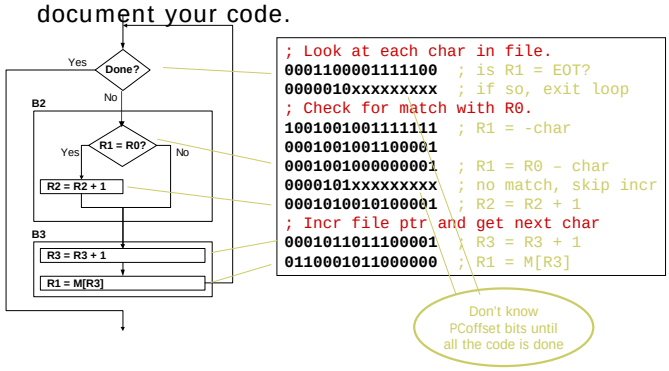
Refining B2 and B3



Conditional (B2) and sequential (B3). Use of LC-2 registers and instructions.

The Last Step: LC-3 Instructions

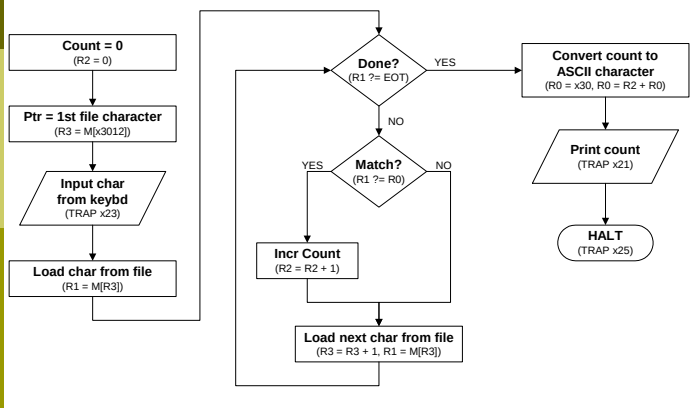
- Use comments to separate into modules and to document your code.




```
; Look at each char in file.
0001100001111100  ; is R1 = EOT?
0000010xxxxxxxxx  ; if so, exit loop
; Check for match with R0.
1001001001111111  ; R1 = -char
0001001001100001
0001001000000001  ; R1 = R0 - char
0000101xxxxxxxxx  ; no match, skip incr
0001010010100001  ; R2 = R2 + 1
; Incr file ptr and get next char
0001011011100001  ; R3 = R3 + 1
0110001011000000  ; R1 = M[R3]
```

Don't know PCoffset bits until all the code is done

Flow Chart



Program (1 of 2)

Address	Instruction	Comments
x3000	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	*R2 ← 0 (counter)*
x3001	0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0	*R3 ← M[x3102] (ptr)*
x3002	1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1	*Input to R0 (TRAP x23)*
x3003	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	*R1 ← M[R3]*
x3004	0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0	*R4 ← R1 – 4 (EOT)*
x3005	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0	*If Z, goto x300E*
x3006	1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1	*R1 ← NOT R1*
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	*R1 ← R1 + 1*
X3008	0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0	*R1 ← R1 + R0*
x3009	0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1	*If N or P, goto x300B*

Program (2 of 2)

Address	Instruction	Comments
x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	*R2 ← R2 + 1*
x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	*R3 ← R3 + 1*
x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	*R1 ← M[R3]*
x300D	0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0	*Goto x3004*
x300E	0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0	*R0 ← M[x3013]*
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0	*R0 ← R0 + R2*
x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1	*Print R0 (TRAP x21)*
x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	*HALT (TRAP x25)*
X3012	Starting Address of File	
x3013	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	*ASCII x30 ('0')*

LC-3 Simulator