LC3 Assembly Programming

1

Recap: LC3 ISA

- LC3 is a 16 bit processor
- 15 opcodes, 8 registers
- Unique encoding for each instruction
- Dataflow diagram for each instruction = how each instruction is implemented/executed
  - Languages to specify dataflow:
    - RTL (Register transfer language) ..used by gcc compilers
    - Hardware description languages (Verilog, VHDL)
- Given a segment of machine code = corresponds to instructions in a program
- After exam: design of LC3 datapath
  - Implementing Central Processing Unit (CPU) using the combinational and sequential devices at our disposal.

2

2

Assembly Language: Human-Readable Machine Language

- how painful was it to read (/write) machine instructions ?
- Computers like ones and zeros…

```
0001110010000110
```

- Humans like symbols…

```
Inclass Code:
LD R2, X  ; R2= X, X contained #5
ADD R1, R2,  # -8   ; R1= R2 -8
BRnz someplace
```

- Assembler is a program that turns symbols into machine instructions.

3

3

Programming in Assembly

- Assembly language level is one-step up from machine
  - All instructions used in Assembly are actual machine instructions….*somewhat!*
  - Use mnemonics and address labels to make it easier to understand the program
    - Labels converted to addresses and offsets by assembler
  - "macros" and utilities to make it easier
- Assembler directives
  - Tell assembler what to do without the programmer explicitly writing out the machine code to do the task
  - Allocating storage
  - Initializing data

Writing program in assembly requires knowing the Instruction set !

4

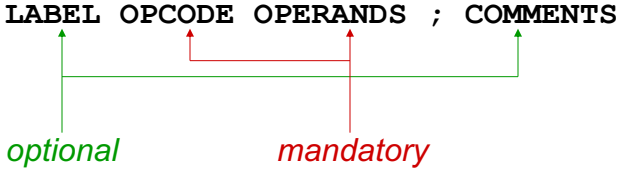
4

LC-3 Assembly Language Syntax

- Each line of a program is one of the following:
  - an instruction
  - an assembler directive (or pseudo-op)
  - a comment
- Whitespace (between symbols) and case are ignored.
- Comments (beginning with ";") on one line are also ignored.

- An instruction has the following format:

```
LABEL OPCODE OPERANDS ; COMMENTS
```



5

5

Opcodes and Operands

- Opcodes
  - reserved symbols that correspond to actual (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      ; R1 = R1 + R3
ADD   R1,R1,#3      ; R1 = R1 + #3
LD    R6,NUMBER     ; R6 = Mem(Address of Number)
BRz   LOOP          ; if previous zero go to address LOOP
```

6

6

Labels and Comments

- Label
  - placed at the beginning of the line
  - assigns a symbolic name to the address corresponding to line
    - ex: LOOP corresponds to some specific memory address

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

    - ex: temp1 corresponds to a memory address – int temp1=7; ?

```
temp1   .FILL #7   ; temp1 = 7
```

- 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

7

7

Assembler Directives

- Pseudo-operations.. To make programmer's life easier
  - do not refer to operations executed by program
  - used by assembler
  - **looks 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`	A	allocate one word, initialize with value A
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters and null terminator

8

8

Trap Codes

- LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them… *more on TRAP instructions later*…

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.

9

9

Example Assembly Program – Add 2 to non-negative number in memory (variable temp1) and store into another memory location (variable temp2)

```
; load number from location temp1,
        .ORIG x3000  ;program starts at address x3000
        LD     R1, temp1     ; load value from temp1 to
                                          register R1
           note: offset not specified by programmer
                     assembler calculates offset needed
        BRn Done  ;if number is Negative goto end
        ADD    R3, R1, #2    ; else Add 2 and store into R3
        ST     R3, temp2   ; store result in R3 into temp2
Done    HALT   ;halt program
;
temp2   .BLKW  1 ; reserve/set aside one word in memory
temp1   .FILL  x0005         ; initialize number to 5

        .END   ; end of program
```

C code:
temp1 = 5;
temp2 = temp1 +2 ;

10

10

Example Assembly Program – Add 2 to non-negative number in memory (variable temp1) and store into another memory location (variable temp2)

```
;
.ORIG x3000    ;program starts at address x3000
     LD R1, temp1  ;load value from temp1 to register R1
            note: offset not specified by programmer
                        assembler calculates offset needed
     BRn Done      ; if number negative then done/end
     ADD R3, R1, #2    ;else add 2 to number and store into R3
     ST R3, temp2 ;store result in R3 into memory loc. temp2
Done   HALT          ; halt program
temp2  .BLKW #1      ;reserve one word in memory
temp1  .FILL x0005   ;initialize location temp1 to 5
       .END          ;end of program
```

C code:
temp1 = 5;
temp2 = temp1 +2 ;

11

11

Label: refers to a memory location

```
; assembly program for temp2= temp1 +2;
        .ORIG x3000   ;program starts at address x3000
        LD     R1, temp1    ; temp1 is location in memory
               ; note: offset not specified by programmer
        BRn Done  ;if number is Negative goto end
        ADD    R3, R1, #2    ; Add 2 store into R3
        ST     R3, temp2  ; store result into temp2
Done    HALT   ;halt program
;
temp2   .BLKW  1
temp1   .FILL  x0005

        .END   ; end of program
```

Must have Opcode and Operands

int temp2;

Immediate values/constants
Decimal #
Binary b
Hex x

int temp1 =5;

.BLKW is Assembler Directive (reserve one location with label 'temp2')

.FILL is Assembler Directive (reserve one location with label 'temp1') and Initialize the value there to be x0005

12

12

```
; Example Assembly Program – Add 2 to non-negative
number and store into another memory location
; load number from locations HERE,
        .ORIG x3000   ;program starts at address x3000
        LD      R1, HERE        ; location in memory
                ; note: replace temp1 by HERE
        BRn Done   ;if number is Negative goto end
        ADD     R3, R1, #2      ; Add 2 store into R3
        ST      R3, PLACE2      ; store result into PLACE2
Done    HALT    ;halt program
;
PLACE2  .BLKW   1               ; temp2 replaced by PLACE2
HERE    .FILL   x0005           ; temp1 replaced by HERE

        .END    ; end of program
```

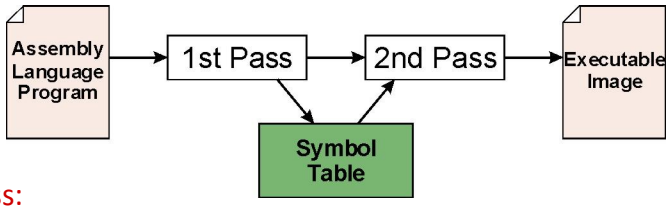
This code would generate identical machine code as previous with labels temp1, temp2

13

13

Assembly Process

- Assembler: Converts assembly language file (.asm) into an executable file (.obj) …*for the LC-3 simulator in our case*.



- First Pass:
  - scan program file
  - find all labels (variables?) and calculate the corresponding addresses; this is called the *symbol table*
- Second Pass:
  - convert instructions to machine language, using information from symbol table

14

14

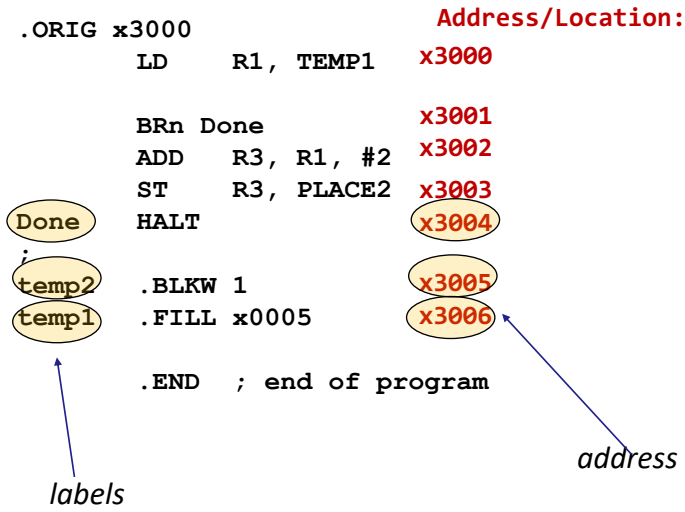
First Pass: Constructing the Symbol Table

1. Find the `.ORIG` statement,
   which tells us the address of the first instruction.
   - Initialize location counter (LC), which keeps track of the current instruction.
2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      - **NOTE: If statement is `.BLKW` or `.STRINGZ`, increment LC by the number of words allocated.**
3. Stop when `.END` statement is reached.

- NOTE: A line that contains only a comment is considered an empty line.

15

15



16

16

Pass 1

- Construct the symbol table for the program

Symbol	Address
Done	x3004
temp2	x3005
temp1	x3006

LD R1, TEMP1 is at address x3000
PC is x3001 when this is executed…therefore offset = ??

17

17

Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.
  - If operand is a label, look up the address from the symbol table.
  - Calculate offset from current instruction PC
    - Ex: LD R1, TEMP1 is LD R1, offset #5
    - Encoding: **0010 001** 000000101
- Potential problems (i.e., syntax errors):
  - Improper number or type of arguments
    - ex:
      ```
      NOT   R1,#7
      ADD   R1,R2
      ADD   R3,R3,NUMBER
      ```
  - Immediate argument too large
    - ex: ADD R1,R2,#1023
  - Address (associated with label) more than 256 from instruction
    - can't use PC-relative addressing mode

18

18

Pass 2

▪Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

Statement	Machine Language
`LD R1, temp1`	0010 001 000000101
`BRn Done`	0000 100 000000010
`ADD R3, R1, #2`	0001 011 001 1 00010

19

Multiple Object Files

▪An object file is not necessarily a complete program.

- system-provided library routines
- code blocks written by multiple developers

▪For LC-3 simulator, can **manually** load multiple object files into memory, then start executing at a desired address.

- system routines, such as keyboard input, are loaded automatically
  - loaded into "system memory," below x3000
  - user code should be loaded between x3000 and xFDFF
- each object file includes a starting address
- be careful not to load overlapping object files

20

Linking and Loading

More on Linkers and loaders in Systems Programming next semester

- *Loading* is the process of copying an executable image into memory.
  - more sophisticated loaders are able to *relocate* images to fit into available memory
    - must readjust branch targets, load/store addresses

- *Linking* is the process of resolving symbols between independent object files.
  - suppose we define a symbol in one module, and want to use it in another (analogy: multiple files in C)
  - some notation, such as .EXTERNAL, is used to tell assembler that a symbol is defined in another module
  - linker will search symbol tables of other modules to resolve symbols and complete code generation before loading

21

21

Programming in assembly..

- Style guidelines
- Problem decomposition and mapping to assembly

22

22

Style Guidelines

Every program starts with .ORIG command, has HALT when computations are done, and a .END at the end of your assembly code.

1. Provide a program header…standard stuff
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.
   1. Mixed upper and lower case for readability.
   2. ASCIItoBinary, InputRoutine, SaveR1
6. Provide comments between program sections.

23

23

Recap: Problem Solving and Problem Decomposition

- With an eye towards writing assembly programming/low-level software
- Flowcharts anyone ?
- Decomposition:
  - Break problem/solution into sub-problems/modules
    - Structured programming
  - Connect the modules…
    - With conditionals, iterations, sequence,….

24

24

Example – similar to program tracing in Labs

- Array of N numbers
- Read length N of the array
- Replace negative numbers by 0 - *check condition*
- Add all the (new) numbers - *iterate over array*
- Print the sum

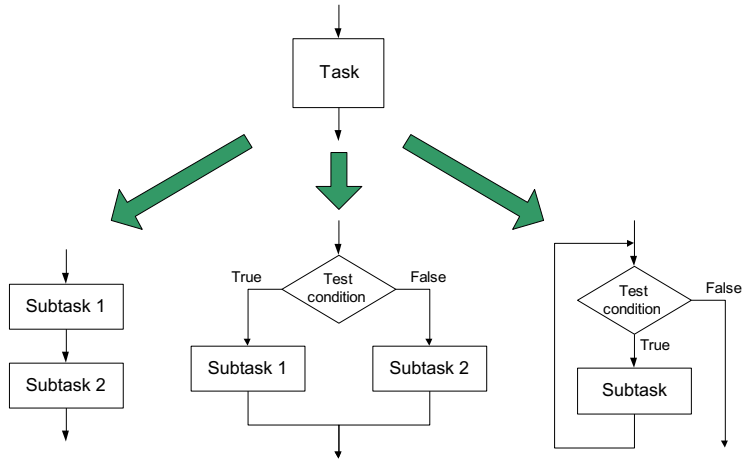
```
i=N;
sum=0;
while (i>0) {
       if A[i}<0  { A[i] = 0;}
       sum = sum + A[i];
       i = i-1;
       }
Printf("sum = %d", sum);
```

25

25

Three Basic Constructs

- There are three basic ways to decompose a task:



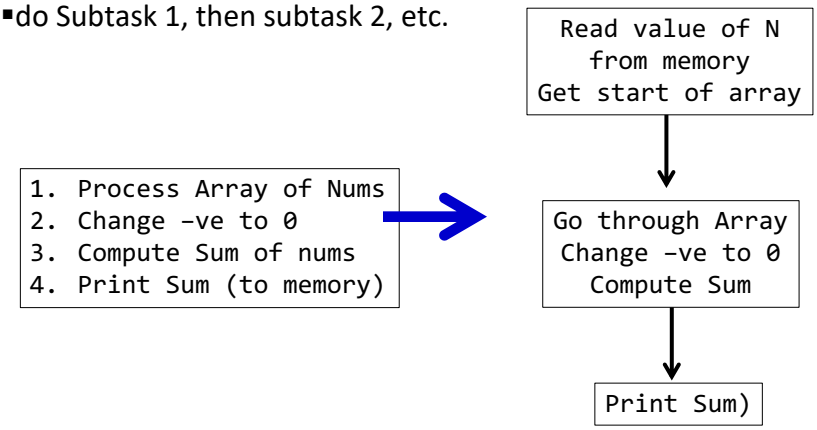
26

26

Sequential

- do Subtask 1, then subtask 2, etc.

```
1. Process Array of Nums
2. Change –ve to 0
3. Compute Sum of nums
4. Print Sum (to memory)
```

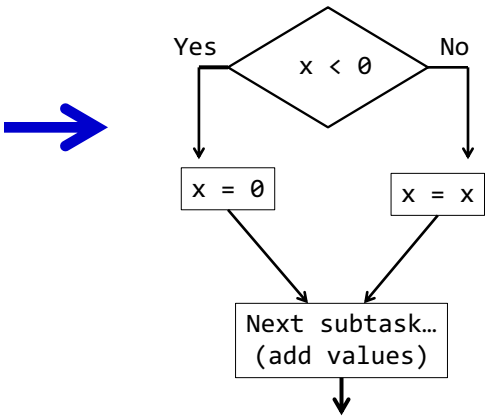


27

Conditional

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

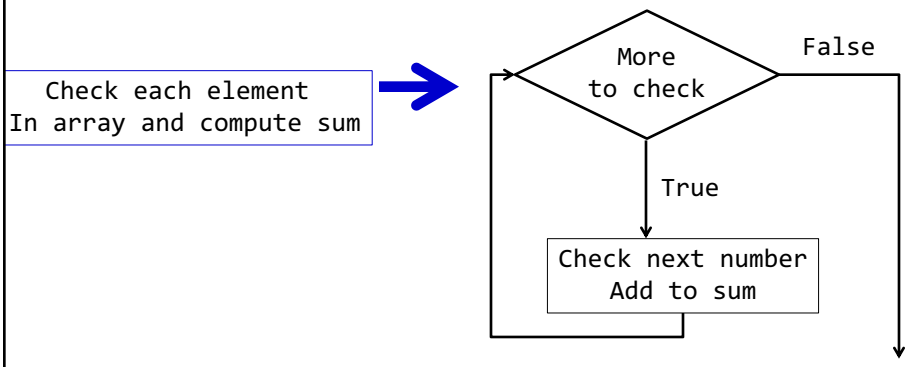
```
Check if number >=0
  Change –ve to 0
```



28

Iterative

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



29

29

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.
  - use BR instruction to transfer control to subtask.

Two instructions to negate R1, i.e. to compute 2's complement of R1

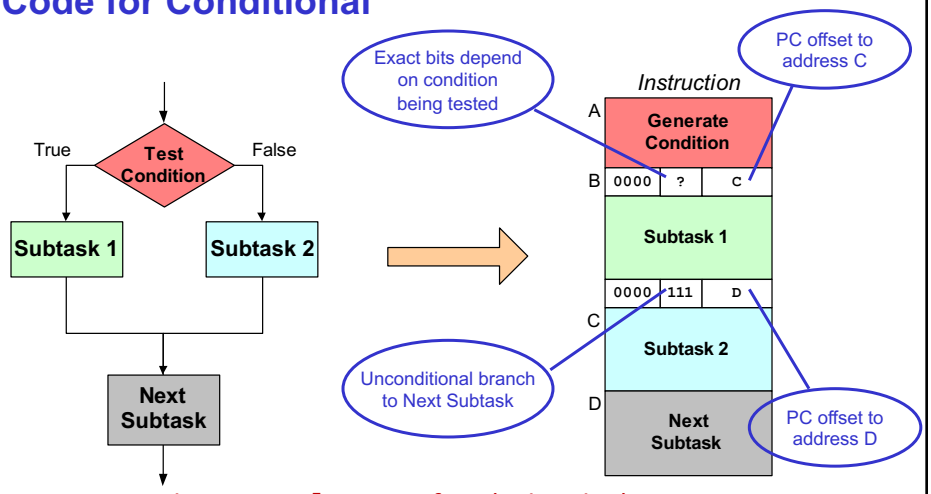
```
NOT R1, R1
ADD R1, R1, #1
ADD R2, R0, R1
BRz equal
```

Question: if condition is "R0 > R1" then what do we change Branch condition to ?

30

30

Code for Conditional



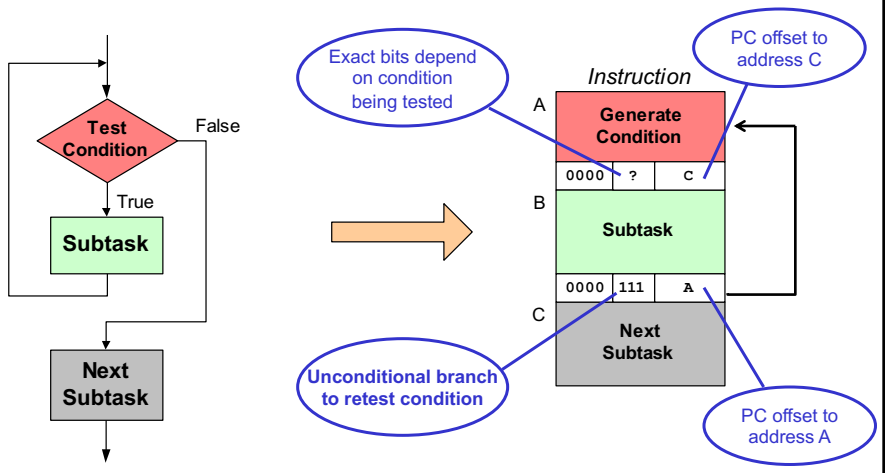
We can also write `else` part first (subtask 1) and the `then` part second (subtask 2)

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

31

31

Code for Iteration



Assuming all addresses are on the same page.

32

Converting Code to Assembly

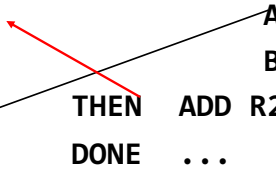
- Can use a standard template approach
- To declare a variable: `int sum;`
  - `Sum .BLKW #1`
- To declare and initialize variable: `int Count 8;`
  - `Count .FILL #8`
- Clear register (R2 = 0;): `AND R2, R2, #0 ; R2 = R2 & 0`
- Copy from R1 to R2: `AND R2, R1, #0 ; R2=R1+0`
- Increment counter/register R0: `ADD R0, R0, #1`
- Decrement counter/register R0: `ADD R0, R0, # -1`

- Typical Constructs
  - `if/else`
  - `while`
  - `do/while`
  - `for`

33

if/else

```
if(x > 0)                    LD   R1, X
{                            BRp  THEN
     r2 = r3 + r4;           ADD R5,R6,R7
}                            BRnzp DONE
else                   THEN  ADD R2,R3,R4
{                      DONE  ...
     r5 = r6 + r7;
}
```



34

if/else

```
if(x > 0)                    LDR1,X
{                            BRnz ELSE
      r2 = r3 + r4;          ADD R2,R3,R4
}                            BRnzp DONE
else                   ELSE  ADD R5,R6,R7
{                      DONE  ...
      r5 = r6 + r7;
}
```

35

while

```
x = 0;                       AND R1,R1,#0
i = 10;                      AND R2,R2,#0
while(i > 0)                 ADD R1,R1,#10
{                      WHL   BRnz DONE
      x = x + i;             ADD R2,R2,R1
      i--;                   ADD R1,R1,#-1
}                            BRnzp WHL
                       DONE  HALT
```

Clearing a register/variable

Initialize register/variable

CC registers set by previous instruction…
i.e., value in R1

Decrement count by 1

Always branch

36

Exercise…. Write assembly program

- Download Exercise1.asm for template
- Fill in code between .ORIG x3000 and the .END
- Variable Num is a place in memory… initialized to 5
- label `array` is starting address of array (x3030)
- Sum is memory location set aside for result

1. First assign registers to each variable (i, sum, A[i])
2. (Write out flowchart)
3. Identify conditionals
4. get starting address of array

```
i=Num; /* array size
sum=0;
while (i>0) {
       if A[i}<0  { A[i] = 0;}
       sum = sum + A[i];
       i = i-1;
       }
Printf("sum = %d", sum);
```

37

37