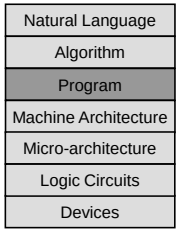
These notes are based on Dr Delmas' and many other people's notes

Chapter 7

Assembly Language



Assembly Language: Human-Readable Machine Language

Computers like ones and zeros…

```
0001110010000110
```

Humans like symbols…

```
ADD  R6,R2,R6
```

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 loop for carrying out multiplication
; through addition operation
;
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:

- an instruction
- an assember directive (or pseudo-op)
- 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 of textbook
  - 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
  - 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

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

LD (LoaD)

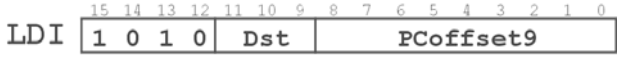
- The LD instruction loads a word stored in the memory into a register
- LD DR, Label
  - DR is the register holding the value retrieved from the memory
  - Label is a label assigned to a memory location
  - LD R1, L1

```
        .ORIG x3010
        LD      R1, L1
        HALT
L1      .FILL  x1234
        .END
```

- After the LD instruction is executed, the value in R1 is x1234

LDI (LoaD Indirect)

- The LDI instruction loads a word stored in the memory into a register
  - There is one more memory access compared to the LD instruction
- LDI DR, Label
  - DR is the register holding the value retrieved from the memory
  - Label is a label assigned to a memory location
    - The memory location stores the address of the value to be retrieved from the memory
  - LDI R1, L1

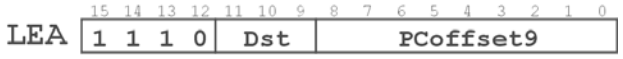



```
        .ORIG x3010

        LDI             R1, L1
        HALT

L1      .FILL x3013
        .FILL x5678   ; the address of this word is x3013
        .END
```

- After the execution of instruction LDI, the value stored in R1 is x5678

LEA (Load Effective Address)

- The LEA instruction loads the address of a memory location into a register
- LEA DR, Label
  - DR is the register holding the address of the memory
  - Label is a label assigned to a memory location
  - LEA R1, L1

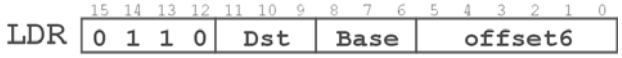


```
        .ORIG x3010

        LEA    R1, L1
        HALT

L1      .FILL x1234 ; the address of this word is x3012
        .END
```

- After the LEA instruction is executed, the value stored in R1 is x3012

LDR (LoaD Register-mode)

- The LDR instruction loads a word stored in the memory into a register
  - Similar to LD, but the way to calculate the memory address is different
- LDR DR, BR, Offset
  - DR is the register holding the value retrieved from the memory
  - BR is the register holding the address of the memory
  - Offset is the value to be added to the value of BR when calculating the address storing the value to be retrieved from the memory
  - LDR R1, R2, #0



; load the values in memory into R1 and R3

```
        .ORIG x3010

        LEA    R2, L1         ; R2 <-- x3015
        LDR    R1, R2, #0     ; R1 <-- x1234
        ADD    R2, R2, #1     ; R2 <-- x3016
        LDR    R3, R2, #0     ; R3 <-- x5678
        HALT

L1      .FILL  x1234 ; address is x3015
        .FILL  x5678 ; address is x3016
        .END
```

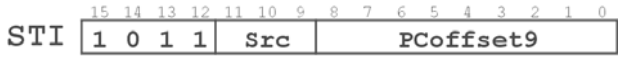

ST (STore)

- The ST instruction stores a value in a given register in a memory location
- ST SR, Label
  - SR is the register holding the value to be stored in the memory
  - Label is a label assigned to a memory location
  - ST R1, L1

```
; clear the memory location with label L1

        .ORIG x3010

        AND         R1, R1, #0    ; R1 ← 0
        ST          R1, L1        ; (L1) ← 0
        HALT

L1      .FILL   x1234
        .END
```


STI (STore Indirect)

- The STI instruction stores a value in a given register in a memory location
  - There is one more memory access compared to the ST instruction
- STI SR, Label
  - SR is the register holding the value to be stored in the memory
  - Label is a label assigned to a memory location
    - The memory location stores the address of the memory location that will store the value
  - STI R1, L1

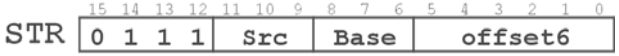



```
        .ORIG x3010

        AND   R1, R1, #0
        STI   R1, L1
        HALT

L1      .FILL x3014
        .FILL x5678   ; address is x3014
        .END
```

- After the STI instruction is executed, the value stored at location x3014 is 0

STR (STore Register-mode)

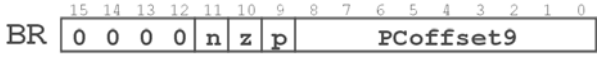
- The STR instruction stores a value in a given register in a memory location
  - Similar to ST, but the way to calculate the memory address is different
- STR SR, BR, Offset
  - SR is the register holding the value to be stored in the memory
  - BR is the register holding the address of the memory
  - Offset is the value to be added to the value of BR when calculating the memory address that will receive the value
  - STR R1, R2, #0



```
        .ORIG x3010

        AND   R1, R1, #0    ; R1 ← 0
        ADD   R1, R1, #1    ; R1 ← 1
        LEA   R2, L1        ; R2 ← x3016
        STR   R1, R2, #0    ; (x3016) ← 1
        STR   R1, R2, #1    ; (x3017) ← 1
        HALT

L1      .BLKW 2       ; L1 is at x3016
        .END
```


BR (BRanch)

- The BR instruction tests the specified condition codes and branch or not branch accordingly.
  - BR can be followed by n, z and p or a combination of them.
  - BRn, BRnz, etc.
- BRn Label
  - BRn Negative
  - Label is a label assigned to an instruction

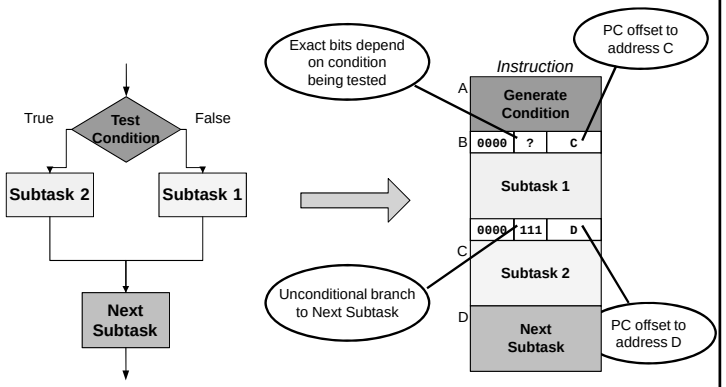


JMP (JuMP)

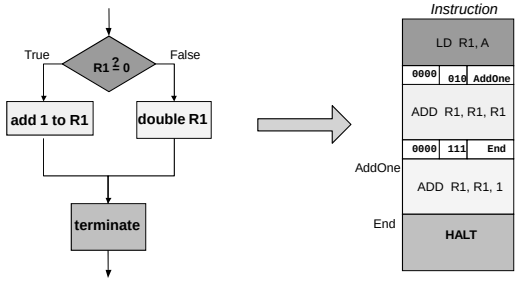
- The JMP instruction unconditionally branch to execute a specified instruction
- JMP BR
  - BR is the register holding the address of the instruction to be executed next
  - JMP R4
- JMP and BRnzp have the same effect
  - BRnzp can branch to an instruction that is within 256 words of BR instruction
  - JMP can branch to an instruction that is further away from the current instruction


```
                .ORIG x3000

                LD      R1, L1          ; R1 ← xFFFF
                BRn     Negative        ; if R1 is a negative number
                                        ; go to execute instruction
                                        ; with label Negative
                AND     R1, R1, #0
End             HALT

Negative        ADD     R1, R1, #1
                LEA     R4, End
                JMP     R4              ; go to execution instruction
                                        ; with label End

L1              .FILL xFFFF
                .END
```

Code for Conditional

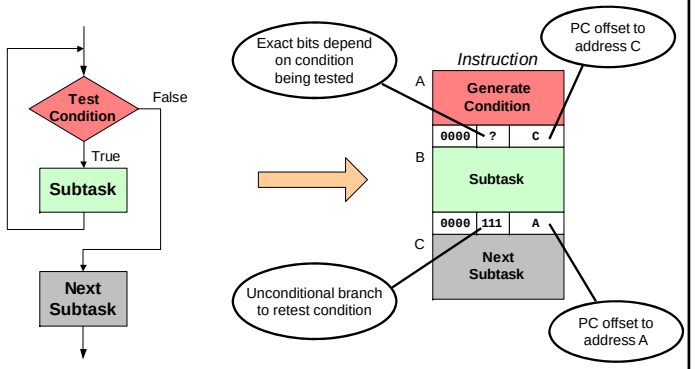


Assuming all addresses are close enough that PC-relative branch can be used.




```
            .ORIG X3000

            LD      R1, A
            BRz     AddOne      ; check whether R1=0
            ADD     R1, R1, R1
            BRnzp   End         ; branch to End
AddOne      ADD     R1, R1, 1
End         HALT

A           .FILL   X12
            .END
```

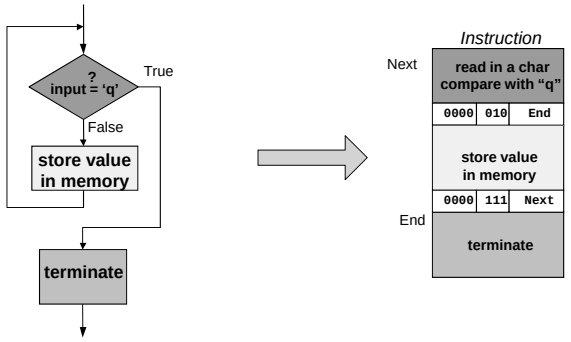

Code for Iteration



Assuming all addresses are on the same page.

example

- read in a sequence of single digits
- store the integer value of the digits in memory
  - the value being read in by the program is the ASCII value of the digit
  - digit “0” ASCII value is 48 while the integer value of the digit is 0
  - digit “1” ASCII value is 49 while the integer value of the digit is 1
  - the integer value of digit n can be obtained by :
    ASCII value of n – ASCII value of 0
- the input is terminated by character “q”
- assume there is at least one digit in the input



Trap Codes

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

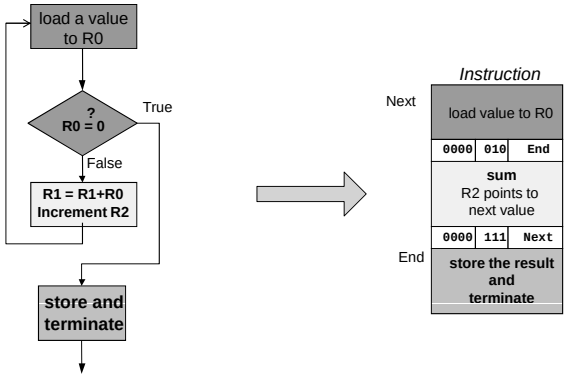
Code	*Equivalent*	*Description*
HALT	TRAP x25	Halt execution and print message to console.
IN	TRAP x23	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
OUT	TRAP x21	Write one character (in R0[7:0]) to console.
GETC	TRAP x20	Read one character from keyboard.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


```
        .ORIG X3000
        LEA     R0,     Msg     ; prompt the user for input
        PUTS
        LEA     R1, Num         ; R1 points to the first location for
                                ; storing the input value
Next    LD      R2, Mq
        IN                      ; read in one character
        ADD     R3, R2, R0      ; check whether the input is “q”
        BRz     End             ; if input is “q”, terminate
        LD      R2, M30
        ADD     R0, R0, R2      ; obtain the value of the input digit
        STR     R0, R1, 0       ; store the value in memory
        ADD     R1, R1, 1       ; R1 points to the next location for
                                ; storing the value
        BRnzp   Next
End     HALT
```

```
Msg     .STRINGZ    "Example:"
Num     .BLKW  10
Mq      .FILL    -113    ; -1 * the ASCII value of “q”
M30     .FILL    -48     ; -1 * the ASCII value of “0”
        .END
```


example

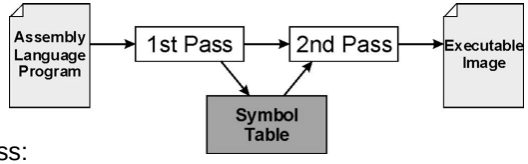
- sum a list of non-zero integers stored in memory
  - the end of the list is marked by a location stored with 0
- the result will be stored in memory
- the algorithm
  - R0 holds the integer that is being processed
  - R1 stores the accumulated value
  - R2 holds the address of the location that stores the value to be loaded into R0
  - use a loop to load each value in the list into register R0 and carry out the accumulation operation




```
        .ORIG X3000

        LEA     R2, List        ; R2 points to the start of the integer list
        AND     R1, R1, 0       ; initialize the value in R1 to 0
Next    LDR     R0, R2, 0       ; load a value into R0
        BRz     End             ; check whether it is the end of the list
        ADD     R1, R1, R0      ; calculate the sum
        ADD     R2, R2, 1       ; R2 points to the next integer in the list
        BRnzp   Next            ; process the next value
End     ST      R1, Result      ; store the result
        HALT
List    .FILL   1               ; list of integers to be processed
        .FILL   2
        .FILL   3
        .FILL   4
        .FILL   0               ; mark the end of the integer list
Result  .FILL   0               ; location for holding the result
        .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

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.

Practice

Construct the symbol table for the program on slides 32 to 33

Symbol	Address
Next	
End	
Msg	
Num	
Mq	
M30	

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.

Potential problems:

- 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

Practice

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

Statement	Machine Language
`LEA R0, Msg`	
`ADD R3,R2,R0`	
`BRz End`	
`STR R0,R1,0`	

Reviews

- Use LC3 instructions to write assembly language programs
- Understand the assembly process
  - Able to build the symbol table
  - Able to convert an assembly instruction into a machine instruction and vice versa
- Read chapter 6 and 7 of the textbook