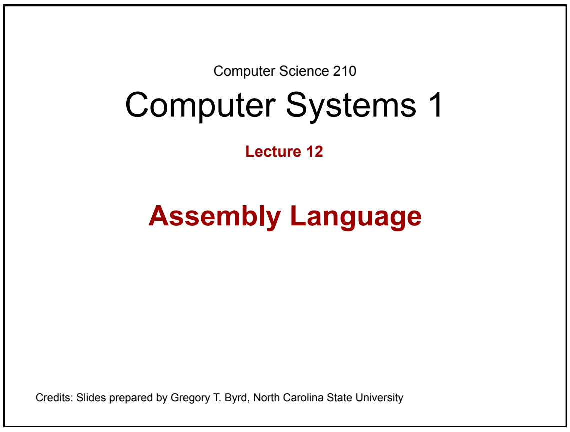


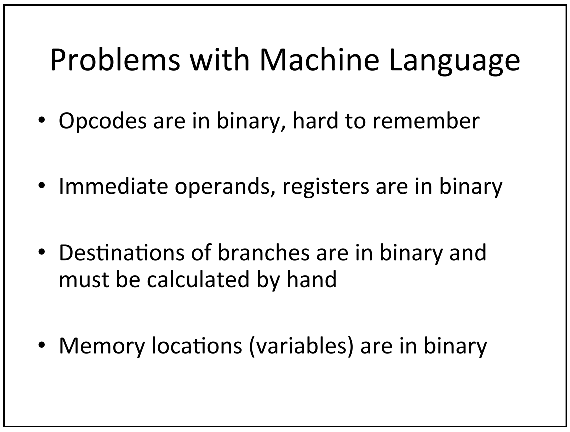
Computer Science 210

Computer Systems 1

Lecture 12

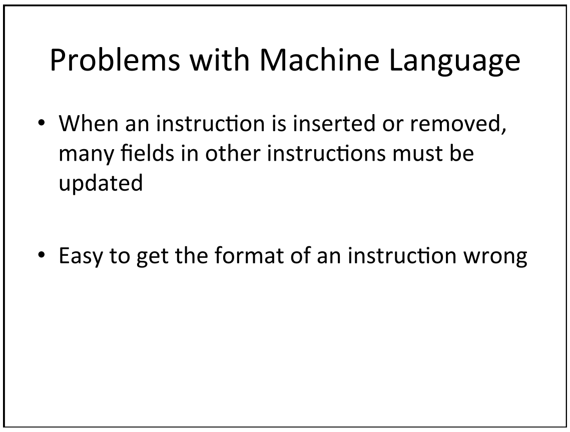
Assembly Language

Credits: Slides prepared by Gregory T. Byrd, North Carolina State University



Problems with Machine Language

- Opcodes are in binary, hard to remember
- Immediate operands, registers are in binary
- Destinations of branches are in binary and must be calculated by hand
- Memory locations (variables) are in binary



Problems with Machine Language

- When an instruction is inserted or removed, many fields in other instructions must be updated
- Easy to get the format of an instruction wrong

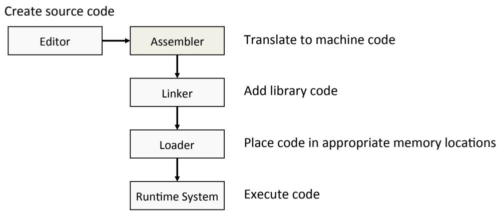
Needed Improvements

- Mnemonic symbols (ADD, BRp) for opcodes
- Mnemonic symbols (count, endwhile) for data variables in memory and destinations of branches
- Automatic update of addresses after modifications to code

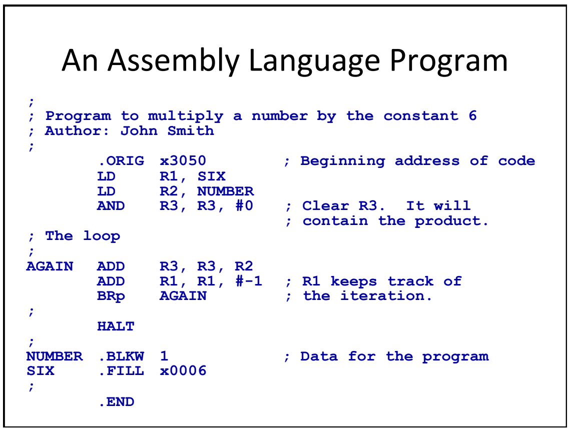
Needed Improvements

- Use of decimal or hex numeric literals for immediate operands
- #101, x3A0B
- Simple syntax checking (format of instructions, undeclared labels, etc.)
- Reserve memory and initialize it

Program Development

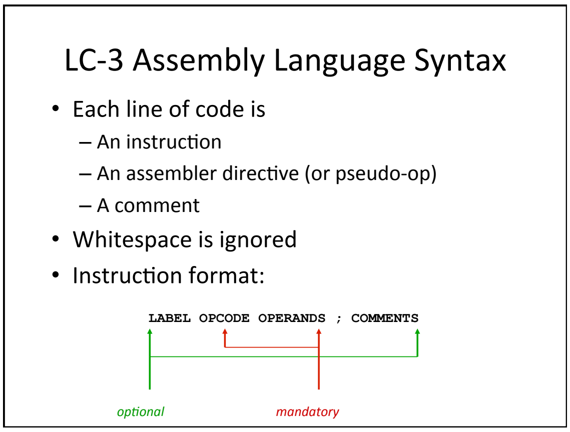


/opt/lc3tools/lc3as
usage: ./lc3as <ASM filename>



An Assembly Language Program

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

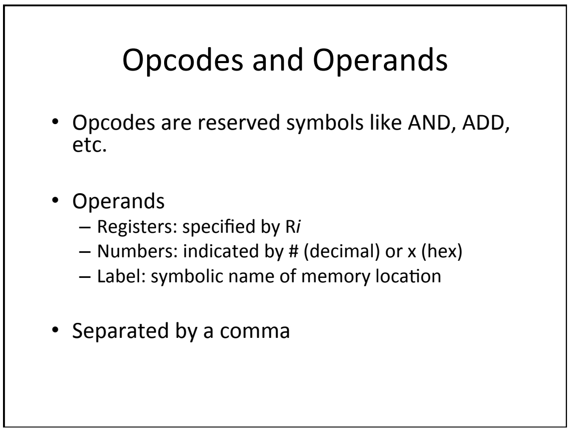


LC-3 Assembly Language Syntax

- Each line of code is
  - An instruction
  - An assembler directive (or pseudo-op)
  - A comment
- Whitespace is ignored
- Instruction format:

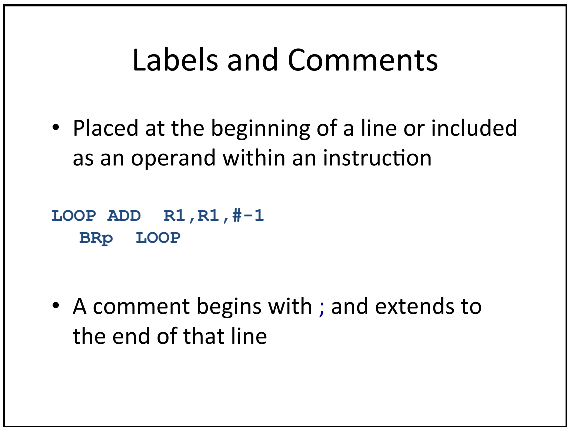
`LABEL OPCODE OPERANDS ; COMMENTS`

optional *mandatory*



Opcodes and Operands

- Opcodes are reserved symbols like AND, ADD, etc.
- Operands
  - Registers: specified by R*i*
  - Numbers: indicated by # (decimal) or x (hex)
  - Label: symbolic name of memory location
- Separated by a comma

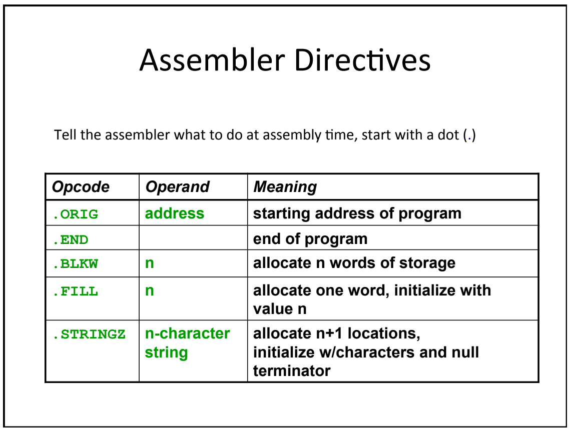


Labels and Comments

- Placed at the beginning of a line or included as an operand within an instruction

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

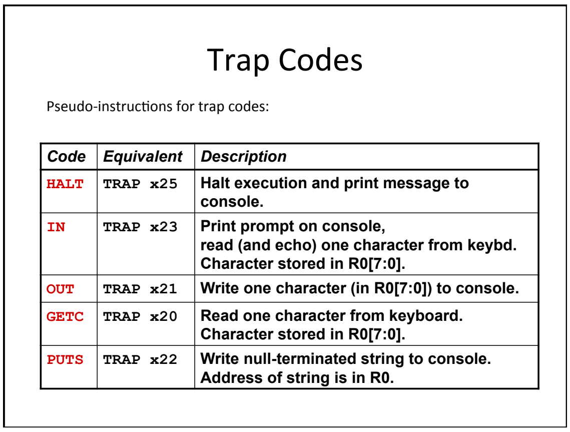
- A comment begins with ; and extends to the end of that line



Assembler Directives

Tell the assembler what to do at assembly time, start with a 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

Pseudo-instructions for trap codes:

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:

- Provide a program header, with author's name, date, etc., and purpose of program.
- Start labels, opcode, operands, and comments in same column for each line (unless entire line is a comment).
- Use comments to explain what each register does.
- Give explanatory comment for most instructions.

Style Guidelines

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

- Use meaningful symbolic names.
- Mixed upper and lower case for readability
  ASCIItoBinary, InputRoutine, SaveR1
- Provide comments between program sections.
- Each line must fit on the page -- no wraparounds
- Long statements split in aesthetically pleasing manner

Human-Readable Machine Language

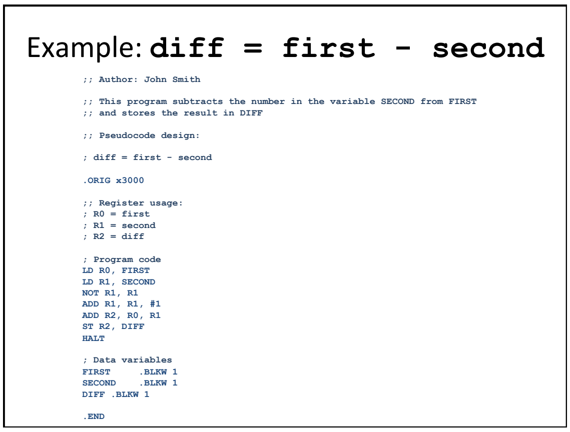
Computers like 1s and 0s:

```
0001001001100001
```

People like symbols:

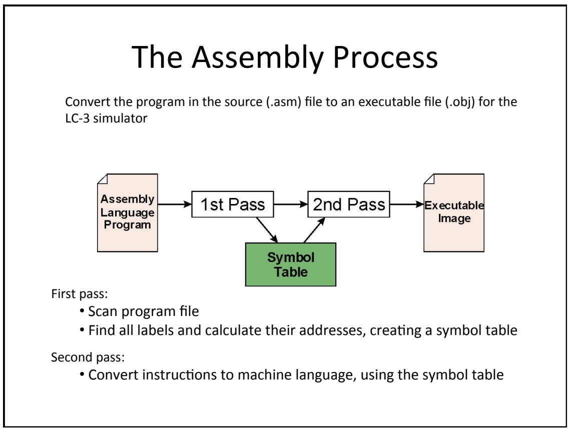
```
ADD R1, R1, #1   ; Increment R1
```

The assembler makes this happen!



Example: **`diff = first - second`**

```
;; Author: John Smith

;; This program subtracts the number in the variable SECOND from FIRST
;; and stores the result in DIFF

;; Pseudocode design:

; diff = first - second

.ORIG x3000

;; Register usage:
; R0 = first
; R1 = second
; R2 = diff

; Program code
LD R0, FIRST
LD R1, SECOND
NOT R1, R1
ADD R1, R1, #1
ADD R2, R0, R1
ST R2, DIFF
HALT

; Data variables
FIRST     .BLKW 1
SECOND    .BLKW 1
DIFF .BLKW 1

.END
```



The Assembly Process

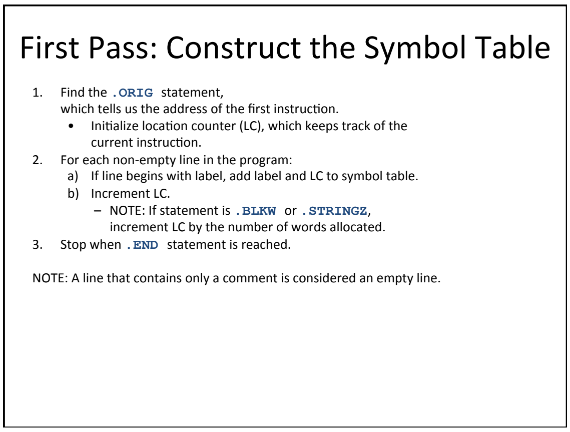
Convert the program in the source (.asm) file to an executable file (.obj) for the LC-3 simulator

First pass:

- Scan program file
- Find all labels and calculate their addresses, creating a symbol table

Second pass:

- Convert instructions to machine language, using the symbol table



First Pass: Construct 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 begins with 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.

Example Symbol Table

Code in subtract.asm	Table in subtract.sym

`LD R0, FIRST`
`LD R1, SECOND`
`NOT R1, R1`
`ADD R1, R1, #1`
`ADD R2, R0, R1`
`ST R2, DIFF`
`HALT`
`FIRST  .BLKW 1`
`SECOND .BLKW 1`
`DIFF   .BLKW 1`
	// Symbol table
	// Scope level 0:
	// Symbol Name Page Address
	// ---------------- ------------
	// FIRST 3007
	// SECOND 3008
	// DIFF 3009

Second Pass: Generate Machine Code

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 errors to detect and flag:

Improper number or type of arguments

```
ex:   NOT  R1,#7
      ADD  R1,R2
      ADD  R3,R3,NUMBER
```

Immediate argument too large

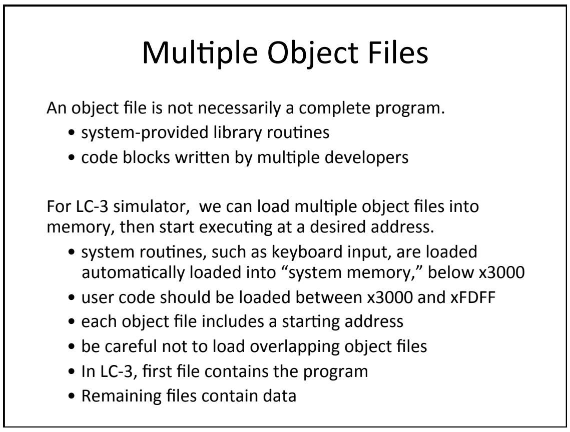
```
ex:   ADD  R1,R2,#1023
```

Address (associated with label) more than 256 from instruction; can't use PC-relative addressing mode

Object File Format

An LC-3 object file contains

- Starting address (location where program must be loaded)
- followed by…
- Machine language instructions



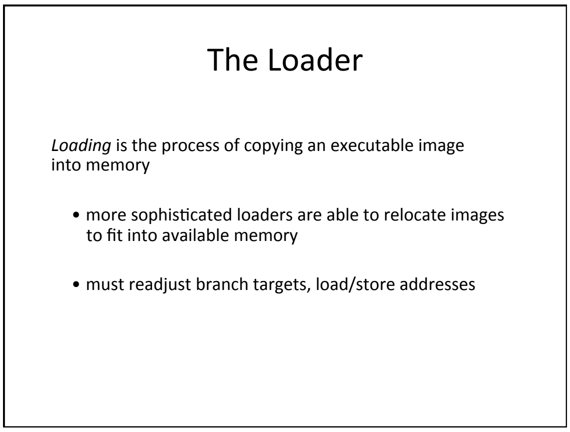
Multiple Object Files

An object file is not necessarily a complete program.

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

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

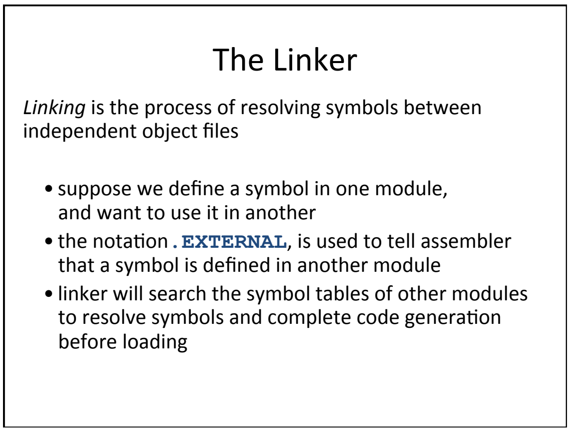
- system routines, such as keyboard input, are loaded automatically loaded into "system memory," below x3000
- user code should be loaded between x3000 and xFDFF
- each object file includes a starting address
- be careful not to load overlapping object files
- In LC-3, first file contains the program
- Remaining files contain data



The Loader

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



The Linker

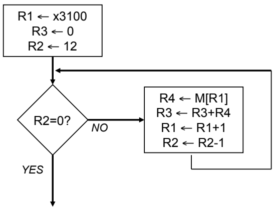
Linking is the process of resolving symbols between independent object files

- suppose we define a symbol in one module, and want to use it in another
- the notation `.EXTERNAL`, is used to tell assembler that a symbol is defined in another module
- linker will search the symbol tables of other modules to resolve symbols and complete code generation before loading

Using Branch Instructions

Compute sum of 12 integers.

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



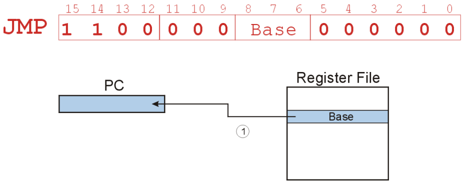
Sample Program

Address	Instruction	Comments
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	R1 ← x3100 (PC+0x0FF)
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	R3 ← 0
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0	R2 ← 12
x3004	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1	If Z, goto x300A (PC+5)
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	Load next value to R4
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	Add to R3
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	Increment R1 (pointer)
x3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	Decrement R2 (counter)
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0	Goto x3004 (PC-6)

JMP (Register)

Jump is an unconditional branch -- *always* taken.

- Target address is the contents of a register.
- Allows any target address.



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.)

Another Example

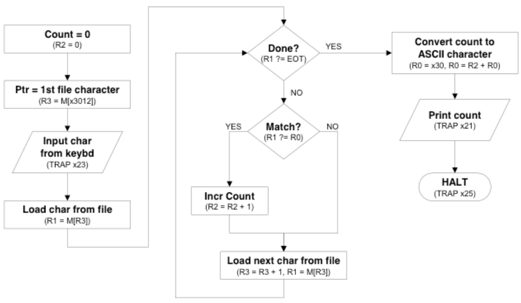
Count the occurrences of a character in a file

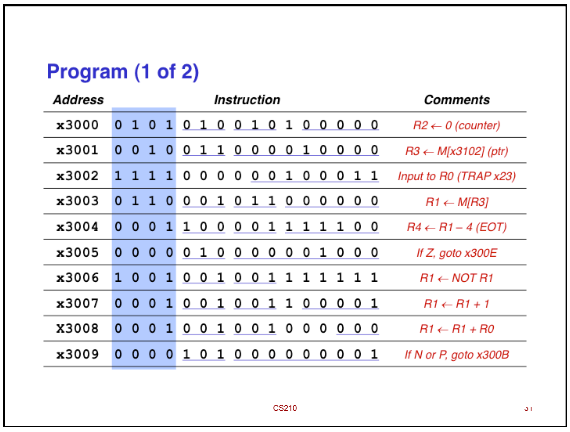
- Program begins at location x3000
- Read character from keyboard
- Load each character from a "file"
  - File is a sequence of memory locations
  - Starting address of file is stored in the memory location immediately after the program
- If file character equals input character, increment counter
- 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.

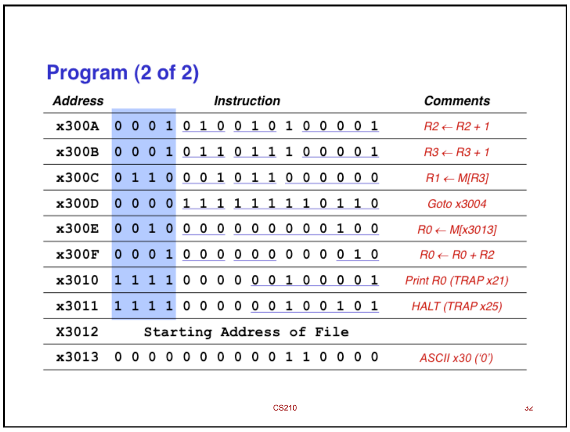
Flow Chart





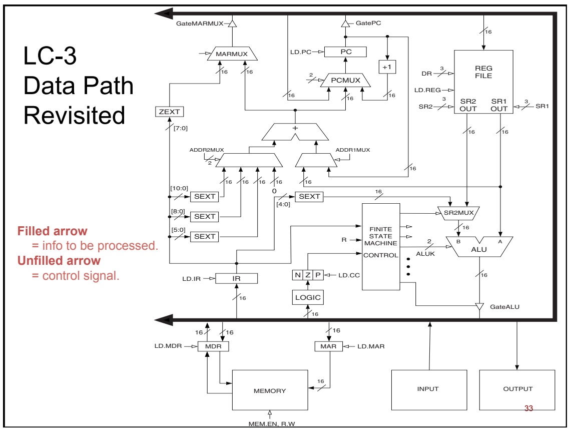
Program (1 of 2)

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 0 1 0 0	R0 ← M[x3013]
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 0 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 Data Path Revisited

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

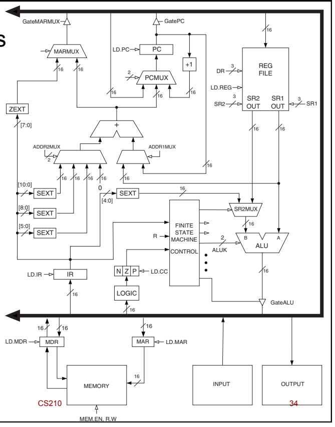
Data Path Components

Global bus

- special set of wires that carry a 16-bit signal to many components
- inputs to the bus are “tri-state devices,” that only place a signal on the bus when they are enabled
- only one (16-bit) signal should be enabled at any time
  - control unit decides which signal “drives” the bus
- any number of components can read the bus
  - register only captures bus data if it is write-enabled by the control unit

Memory

- Control and data registers for memory and I/O devices
- memory: MAR, MDR (also control signal for read/write)

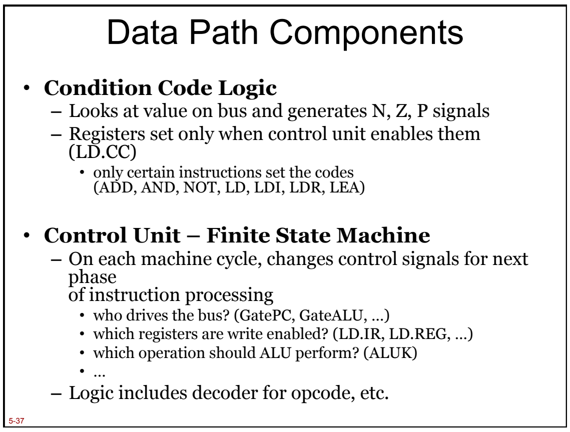


Data Path Components

- **ALU**
  - Accepts inputs from register file and from sign-extended bits from IR (immediate field).
  - Output goes to bus.
    - used by condition code logic, register file, memory
- **Register File**
  - Two read addresses (SR1, SR2), one write address (DR)
  - Input from bus
    - result of ALU operation or memory read
  - Two 16-bit outputs
    - used by ALU, PC, memory address
    - data for store instructions passes through ALU

Data Path Components

- **PC and PCMUX**
  - Three inputs to PC, controlled by PCMUX
    1. PC+1 – FETCH stage
    2. Address adder – BR, JMP
    3. bus – TRAP (discussed later)
- **MAR and MARMUX**
  - Two inputs to MAR, controlled by MARMUX
    1. Address adder – LD/ST, LDR/STR
    2. Zero-extended IR[7:0] -- TRAP (discussed later)



Data Path Components

- **Condition Code Logic**
  - Looks at value on bus and generates N, Z, P signals
  - Registers set only when control unit enables them (LD.CC)
    - only certain instructions set the codes (ADD, AND, NOT, LD, LDI, LDR, LEA)
- **Control Unit – Finite State Machine**
  - On each machine cycle, changes control signals for next phase
    of instruction processing
    - who drives the bus? (GatePC, GateALU, …)
    - which registers are write enabled? (LD.IR, LD.REG, …)
    - which operation should ALU perform? (ALUK)
    - …
  - Logic includes decoder for opcode, etc.

5-37