LC-3
Instruction Set Architecture
and Beginning LC-3 Programming

Quiz today
7pm - 7 am
Binary Multiplication
Examples
Unsigned Binary Multiplication

\[
\begin{array}{c}
1011 \\
\times 1101 \\
\hline
\end{array}
\begin{array}{c}
1011 \\
000000 \\
1011000 \\
10110000 \\
\hline
10001111 \\
\end{array}
\]
### 2SC Binary Multiplication

\[
3 \times -5 = -3 \times 5
\]

\[\begin{array}{c}
5 \\
3
\end{array}\]

\[\begin{array}{c}
5 + 5 + 5
\end{array}\]

\[
\begin{array}{c|cccc}
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

\[
\begin{array}{c}
1001 \\
0101
\end{array}
\]

\[
\begin{array}{c}
1001 \\
0101 \\
\hline
110111101
\end{array}
\]
\[ \frac{8}{4} \cdot 8 = 64 = 2^6 \]

\[ 15 \div 5 = \]

\[ \frac{11111}{1111000} \]

\[ 11111000 \]

\[ 0 \]
\[
\frac{1}{10} + \frac{1}{10} + \frac{1}{10}
\]

\[
\frac{10}{100} + 1 = 1
\]

\[
100
\]
CISC vs. RISC

CISC: Complex Instruction Set Computer
Lots of instructions of variable size, very memory optimal, typically less registers.

RISC: Reduced Instruction Set Computer; Less instructions, all of a fixed size, more registers, optimized for speed. Usually called a “Load/Store” architecture.
What is “Modern”

For embedded applications and for workstations there exist a wide variety of CISC and RISC and CISCy RISC and RISCy CISC. Most current PCs use the best of both worlds to achieve optimal performance.
Instruction Set Architecture

ISA is all of the *programmer-visible* components and operations of the computer.

- **memory organization**
  - address space -- how many locations can be addressed?
  - addressability -- how many bits per location?
- **register set**
  - how many? what size? how are they used?
- **instruction set**
  - Opcodes (what commands can we give to the computer)
  - data types
  - addressing modes

The ISA provides all the information needed for someone to write a program in machine language (or translate from a high-level language to machine language).
LC-3 Architecture

- Very RISC, only 15 instructions
- 16-bit data and address ~64K
- 8 general purpose registers (GPR)
- Architecture
  - Program Counter (PC)
  - Instruction Register (IR)
  - Condition Code Register (CC)
  - Process Status Register (PSR)
Memory vs. 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 is 16 bits wide
  - how many bits to uniquely identify a register?
- other registers
  - not directly addressable, but used/effect by instructions
  - PC (program counter), condition codes
LC-3 Overview

Instruction Set

Opcodes
- 15 opcodes
- **Operate** (Logical or Arithmetic) 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 (< 0), 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
Hello World

- Traditional First program on a system
  - Can be difficult to get to
  .ORIG x3000
  LEA R0, HELLO
  PUTS
  HALT
  HELLO .STRINGZ "Hello CMPE12"
  .END
Syntax of LC-3

• One instruction, declaration per line
• Comments are anything on a line following ";"
• Comments may not span lines

ADD R0,R0,R0

; R0 = R0 + R0
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.
NOT

- Takes the bitwise not of the SRC and puts it in the DST.
- Note: SRC and DST could be the same register.

NOT DST, SRC
NOT R0, R1

NOT R0, R0
ADD/AND

- Takes the addition/and of SRC1 and SRC2 and puts it in the DST.
- Note: All three could be the same register.

\[ a = b + c \]

ADD DST, SRC1, SRC2
ADD R6, R1, R2
ADD R0, R0, R0
ADD/AND (with constants)

- Takes the addition/and of SRC1 and constant and puts it in the DST.
- Note: All three could be the same register.

ADD DST, SRC1,4
ADD R2,R3,6
ADD R3,R3,1
ADD R2,R2,1

MAXWELL JAMES DUNNE
Using Operate Instructions

With only ADD, AND, NOT...

- How do we subtract?
  \[ R_3 = R_1 - R_2 \]
  \[ R_2 = \text{NOT}(R_2) \]
  \[ R_2 = R_2 + 1 \]
  \[ R_3 = R_1 + R_2 \]

- How do we OR?

- How do we copy from one register to another?
  \[ R_3 = R_2 \]
  ADD \( R_3, R_2, 0 \)

- How do we initialize a register to zero?
  \[ R_1 = R_1 \text{ AND } 0 \]
Data Movement Instructions

Load -- read data from memory to register
- LD
- LDR

Store -- write data from register to memory
- ST
- STR

Load effective address -- compute address, save in register
- LEA

LDI, and STI will be covered when we go over the architecture)

We will use labels instead for now.
Labels

• Symbolic names that are used to identify memory locations
• Location for target of a branch or jump
• [Location for a variable for loading and storing]
• Can be 1-20 characters in size
• We start at address 0x3000 by convention

LEA R0, HELLO

0x318A
LD (Load Data)

- Loads the contents of LABEL and stores it in DST

LD DST, LABEL
LD R3, FOO
SD (Store Data)

- Stores the contents of SRC in LABEL

SD SRC, LABEL
SD R3, FOO

↑
Load Effective Address

Computes a memory location from LABEL and stores it in DST.
We use it a lot for output

LEA DST, LABEL
LEA R0, HELLO
LDR (Load Data with Register)

- Use \texttt{SRC} as memory address and adds \texttt{OFFSET} to it. The contents of this new address is then stored in \texttt{DST}.
  - Offset can be 0

\begin{align*}
\text{LDR DST, SRC, OFFSET} \\
\text{LDR R3, R0, 2} \\
\text{LDR R3, R0, 0}
\end{align*}
SDR (Store Data with register)

- Use DST as memory address and adds OFFSET to it. This new memory address has SRC stored in it.

\[
\text{SDR SRC, DST, OFFSET} \\
\text{SD R1,R2,0}
\]
TRAP
(System Calls)

- Very tedious and dangerous for a programmer to deal with IO at the OS level.
- Need an instruction though to get the attention of the OS.

Use the “TRAP” instruction and a “trap vector”.
## Trap Service Routines

<table>
<thead>
<tr>
<th>Trap Vector</th>
<th>Assembler Name</th>
<th>Usage &amp; Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>GETC</td>
<td>Read a character from console into R0, not echoed.</td>
</tr>
<tr>
<td>0x21</td>
<td>OUT</td>
<td>Write character in R0 to console.</td>
</tr>
<tr>
<td>0x22</td>
<td>PUTS</td>
<td>Write string of characters to console. Start with character at address contained in R0. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>0x23</td>
<td>IN</td>
<td>Print a prompt to console and read in a single character into R0. Character is echoed.</td>
</tr>
<tr>
<td>0x24</td>
<td>PUTSP</td>
<td>Write a string of characters to console, 2 characters per address location. Start with characters at address in R0. First [7:0] and then [15:0]. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>0x25</td>
<td>HALT</td>
<td>Halt execution and print message to console.</td>
</tr>
</tbody>
</table>

**TRAP 0x20**
To print a character

; the char must be in R0.

TRAP \texttt{x21}

or

OUT

To read in a character

; will go into R0, no echo.

TRAP \texttt{x20}

or

GETC
To end your program:

TRAP or HALT

x25
Directives give information to the assembler. All directives start with ‘.’ (period)

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>Always 0x3000 for now (the start of our program)</td>
</tr>
<tr>
<td>.FILL</td>
<td>Declare a memory location</td>
</tr>
<tr>
<td>.BLKW</td>
<td>Reserve a group of memory locations</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>Declare a group of characters in memory</td>
</tr>
<tr>
<td>.END</td>
<td>Tells assembly where your program source ends</td>
</tr>
</tbody>
</table>
Hello World (again)

- Traditional First program on a system
  - Can be difficult to get to

```
.ORIG x3000
LEA R0, HELLO
PUTS
HALT
HELLO .STRINGZ "Hello CMPE12"
.END
```
Program Flow Charting

How to tackle the beginning stage of a program design
A Program

Set of instructions written in a programming language that tells the computer what to do
Programmers

- Prepare instructions that make up the program
- Run the instructions to see if they produce the correct results
- Make corrections (Debug)
- Document the program
- Interact with
  - Users
  - Managers
  - Systems analysts
- Coordinate with other programmers to build a complete system
The Programming Process

- Defining the problem
- Planning the solution
- Coding the program
- Testing the program
- Documenting the program

3 weeks
The Programming Process: Defining the Problem

- What is the input
- What output do you expect
- How do you get from the input to the output
The Programming Process: Planning the Solution

- Algorithms
  - Detailed solutions to a given problem
    - Sorting records, adding sums of numbers, etc..
- Design tools
  - Flowchart
  - Pseudocode
    - Has logic structure, but no command syntax

\[ a = b + c \]
The Programming Process: Planning the Solution

- Desk-checking
  - Personal code design walk through
- Peer Reviews
  - “Code walk through”/structured walk through
Flow Control Elements

The Programming Process: Planning the Solution
Accept series of numbers and display the average
The Programming Process: 
_Coding the Program_

- Translate algorithm into a formal programming language
- Within syntax of the language
- How to key in the statements?
  - Text editor
  - Programming environment
    - Interactive Development Environment (IDE)
The Programming Process: Testing the Program

• Translation – compiler
  – Translates from source module into object module
  – Detects syntax errors

• Link – linkage editor (linker)
  – Combines object module with libraries to create load module
  – Finds undefined external references

• Debugging
  – Run using data that tests all statements
  – Logic errors
The Programming Process: Documenting the Program

- Performed throughout the development
- Material generated during each step
  - Problem definitions
  - Program plan
  - Comments within source code
  - Testing procedures
  - Narrative
  - Layouts of input and output
  - Program listing
Procedural Level Languages

- 1\textsuperscript{st} Generation: Machine Level
- 2\textsuperscript{nd} Generation: Assembly Level
- 3\textsuperscript{rd} Generation: High Level
FORTRAN

```fortran
C FORTRAN PROGRAM
C AVERAGING INTEGERS ENTERED THROUGH THE KEYBOARD
WRITE (6,10)
SUM = 0
COUNTER = 0
WRITE (6,60)
READ (5,40) NUMBER
1 IF (NUMBER .EQ. 999) GOTO 2
SUM = SUM + NUMBER
COUNTER = COUNTER + 1
WRITE (6,70)
READ (5,40) NUMBER
GO TO 1
2 AVERAGE = SUM / COUNTER
WRITE (6,80) AVERAGE
10 FORMAT (1X, 'THIS PROGRAM WILL FIND THE AVERAGE OF',
          'INTEGERS YOU ENTER',/1X, 'THROUGH THE',
          'KEYBOARD. TYPE 999 TO INDICATE END OF DATA.',/)
40 FORMAT (13)
60 FORMAT (1X, 'PLEASE ENTER A NUMBER ')
70 FORMAT (1X, 'PLEASE ENTER THE NEXT NUMBER ')
80 FORMAT (1X, 'THE AVERAGE OF THE NUMBERS IS ', F6.2)
STOP
END
```

This program will find the average of integers you enter through the keyboard. Type 999 to indicate end of data.

Please enter a number 6
Please enter the next number 4
Please enter the next number 11
Please enter the next number 999
The average of the numbers is 7.00
IDENTIFICATION DIVISION.
PROGRAM-ID. AVERAGE.
* COBOL PROGRAM
* AVERAGING INTEGERS ENTERED THROUGH THE KEYBOARD.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-CONTROL. m-f 9:00.
OBJECT-CONTROL. m-f 9:00.
*============================================
DATA DIVISION.
*============================================
FILE SECTION.
*============================================
WORKING-STORAGE SECTION.
01 AVERAGE FIC +9.99.
01 COUNTER FIC 9(02) VALUE ZERO.
01 NUMBER-ITDN FIC 9(03).
01 SUM-ITDN FIC S9(03) VALUE ZERO.
01 BLANK-LINE FIC X(80) VALUE SPACES.
*============================================
PROCEDURE DIVISION.
*============================================
100-CONTROL-Routine.
PERFORM 200-DISPLAY-PROCEDURE.
PERFORM 300-INITIALIZE-Routine.
END-ALL.
*============================================
200-DISPLAY-PROCEDURE.
DISPLAY 'THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER'.
DISPLAY 'THROUGH THE KEYBOARD. TYPE 999 TO INDICATE END OF DATA.'.
DISPLAY BLANK-LINE.
300-INITIALIZE-Routine.
DISPLAY 'PLEASE ENTER A NUMBER'.
ACCEPT NUMBER-ITDN.
400-ENTER-AND-ADD.
ADD NUMBER-ITDN TO SUM-ITDN.
ADD 1 TO COUNTER.
DISPLAY 'PLEASE ENTER THE NEXT NUMBER'.
ACCEPT NUMBER-ITDN.
500-CALCULATE-AVERAGE.
DIVIDE SUM-ITDN BY COUNTER GIVING AVERAGE.
600-DISPLAY-RESULTS.
DISPLAY 'THE AVERAGE OF THE NUMBERS IS ', AVERAGE.

(a)

THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER THROUGH THE KEYBOARD.
TYPE 999 TO INDICATE END OF DATA.

PLEASE ENTER A NUMBER
6
PLEASE ENTER THE NEXT NUMBER
4
PLEASE ENTER THE NEXT NUMBER
11
PLEASE ENTER THE NEXT NUMBER
999
THE AVERAGE OF THE NUMBERS IS 7.00

(b)
'BASIC PROGRAM
'AVERAGING INTEGERS ENTERED THROUGH THE KEYBOARD
CLS
PRINT "THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER"
PRINT "THROUGH THE KEYBOARD. TYPE 999 TO INDICATE END OF DATA."
PRINT
SUM=0
COUNTER=0
PRINT "PLEASE ENTER A NUMBER"
INPUT NUMBER
DO WHILE NUMBER <> 999
    SUM=SUM+NUMBER
    COUNTER=COUNTER+1
    PRINT "PLEASE ENTER THE NEXT NUMBER"
    INPUT NUMBER
LOOP
AVERAGE=SUM/COUNTER
PRINT "THE AVERAGE OF THE NUMBERS IS"; AVERAGE
END

(a)

(b)

THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER THROUGH THE KEYBOARD. TYPE 999 TO INDICATE END OF DATA.

PLEASE ENTER A NUMBER
?6
PLEASE ENTER THE NEXT NUMBER
?4
PLEASE ENTER THE NEXT NUMBER
?11
PLEASE ENTER THE NEXT NUMBER
?999
THE AVERAGE OF THE NUMBERS IS 7
// C++ PROGRAM
// AVERAGING INTEGERS ENTERED THROUGH THE KEYBOARD

#include <iostream.h>

main()
{
    float average;
    int number, counter = 0; int sum = 0;
    cout << "THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER\n";
    cout << "THROUGH THE KEYBOARD. TYPE 999 TO INDICATE END OF DATA. \n";
    cout << "PLEASE ENTER A NUMBER:";
    cin >> number;
    while (number != 999)
    {
        sum = sum + number;
        counter ++;
        cout << "\nPLEASE ENTER THE NEXT NUMBER:";
        cin >> number;
    }
    average = sum / counter;
    cout << "\nTHE AVERAGE OF THE NUMBERS IS " << average
}

C++