LC-3
Instruction Set Architecture and Beginning LC-3 Programming
CISC vs. RISC

\[ x \times 64 \]

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 may 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
- 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? 3
- other registers
  - not directly addressable, but used/effectied by instructions
  - PC (program counter), condition codes

RAM 7x slower registers
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

LED

2 weeks
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 ; ADD R0 to itself
```
Operate Instructions

Only three operations: ADD, AND, NOT, Pass

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
ADD/AND

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

```
ADD DST, SRC1, SRC2
ADD R6, R1, R2
R6 = R1 + R2
```
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
Using Operate Instructions

With only ADD, AND, NOT...

- How do we subtract?
  - $R_3 = R_1 - R_2$
  - $R_2 = R_2$
  - $R_3 = R_2 + 1$
  - $R_3 = R_2 + R_1$

- How do we OR?
  - Demorgans

- How do we copy from one register to another?
  - $R_3 = R_2 + 0$ or $R_3 = R_2 \text{ and } R_2$

- 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
1) ADD R1, R2, R0
2) NOT R0, R2
3) FOO

\( n+2 \)
LD (Load Data)

- Loads the contents of LABEL and stores it in DST

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

- Stores the contents of SRC in LABEL

ST SRC, LABEL
ST 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 SRC as memory address and adds OFFSET to it. The contents of this new address is then stored in DST.
  - Offset can be 0

LDR DST, SRC, OFFSET
LDR R3, R0, 2
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>
To print a character
    ; the char must be in R0.
TRAP x21

or

OUT

To read in a character
    ; will go into R0, no echo.
TRAP x20

or

GETC
To end your program:

- TRAP
- HALT

or

x25
Mac

5-6 restarts
5-10 minutes
**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
    PUTF
    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
- 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
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
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
LC-3
More LC-3 Programming
Control Instructions

Used to alter the sequence of instructions. This allows us to move to a particular instruction

Conditional Branch

- branch is *taken* if a specified condition is true
- else, the branch is *not taken*
  - next sequential instruction is executed
Unconditional Branch (or Jump)

- always changes instruction

TRAP

- changes to an OS “service routine”
- routine will return control to the next instruction (after TRAP) when finished
Condition Codes

LC-3 has three condition code bits:

N -- negative
Z -- zero
P -- positive (greater than zero)

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 is used
  - target address is made by adding signed offset (IR[8:0]) to current PC.
If the branch is not taken, the next sequential instruction is executed.

There are hardware limits on how far you can branch.
BR (Branch)

- It does no computation, only looks at condition codes
- If condition code is set, go to LABEL, can combine codes
  - BRz
  - BRn
  - BRp
  - BRzp

ADD R0,R1,R2
BRz FOO ; if zero we go to label FOO
BR (unconditionally)

- Degenerate case, always goes to LABEL
  - BRnzp

BRnzp FOO
BRnzp FOO ; always go to label FOO
Example: Using a Branch

Compute sum of 12 integers

Numbers start at label NUMS. Program starts at location x3000.
0 \Rightarrow 12
R2 < 12
12 \Rightarrow 0
N \geq P
32k \leq 1 \quad 32k
$R_2 = 12$

$R_1 = R_2 - 12$

$B \geq 100$
### Example: Using a Branch

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LEA R1,NUMS</strong></td>
<td>R1 ← NUMS</td>
</tr>
<tr>
<td><strong>AND R3,R3,0</strong></td>
<td>R3 ← 0</td>
</tr>
<tr>
<td><strong>AND R2,R2,0</strong></td>
<td>R2 ← 0</td>
</tr>
<tr>
<td><strong>ADD R2,R2,12</strong></td>
<td>R2 ← 12</td>
</tr>
<tr>
<td><strong>START BRz END</strong></td>
<td>If Z, goto END</td>
</tr>
<tr>
<td><strong>LDR R4,R1,0</strong></td>
<td>Load next value to R4</td>
</tr>
<tr>
<td><strong>ADD R3,R4,R3</strong></td>
<td>R3 ← R4 + R3</td>
</tr>
<tr>
<td><strong>ADD R1,R1,1</strong></td>
<td>Increment R1 (pointer)</td>
</tr>
<tr>
<td><strong>ADD R2,R2,-1</strong></td>
<td>Decrement R2 (counter)</td>
</tr>
<tr>
<td><strong>BRnzp START</strong></td>
<td>Goto START</td>
</tr>
<tr>
<td><strong>END</strong></td>
<td># done adding</td>
</tr>
</tbody>
</table>
JMP

Jump is an unconditional branch -- *always* taken.
- Target is contents of a register, not a label.

JMP R1
LC-3 Subtraction

\[ R2 = R1 - R0 \]

\[
\begin{align*}
\text{NOT} & \quad R0, R0 \\
\text{ADD} & \quad R0, R0, 1 \\
\text{ADD} & \quad R2, R1, R0
\end{align*}
\]
Print Single Digit Number

8 → '8'
8 + 48 → 56

ADD R1, R1, 48
LD R0, ASCII OFF
ADD R0, R1, RO
OUT
ASCII OFF . Fill 48
Print Single Digit Number

8 → '8'
8 + 48 → 56

ADD R1, R1, 48
ADD R1, R1, 16
ADD R1, R1, 16
LC-3 Multiply

2 \times 3 = 2 + 2 + 2

\begin{align*}
R2 &\leftarrow 3 \\
R1 &\leftarrow 2 \\
R0 &\leftarrow 0
\end{align*}

\begin{align*}
R2 &\leftarrow 0 \\
\text{YES} \\
R0 &\leftarrow R0 + R1 \\
R2 &\leftarrow R2 - 1
\end{align*}
LC-3 Integer Division

\[
24 \div 6 = 24 - 6 \\
18 - 6 \\
12 - 6 \\
6 - 6 \\
0 - 6 \\
-6
\]
4 R 4

\[
\frac{24}{5} = 1 \frac{19}{5} = 1 \frac{4}{5}
\]