LC-3
Instruction Set Architecture
(Ch5)
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
  - 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).
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
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*
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.

Will show dataflow diagram with each instruction.
- illustrates *when* and *where* data moves to accomplish the desired operation
NOT

\[ \text{NOT} \begin{array}{c|c} 1001 \end{array} \begin{array}{c|c} \text{Dst} & \text{Src} \\ 1111111 \end{array} \]

\[ \text{NOT} \begin{array}{c|c} R1, R1 \end{array} \]

Note: \( \text{Src} \) and \( \text{Dst} \) could be the \textit{same} register.

Note: works only with registers.
ALU

Diagram:

- NUM1
- NUM2
- NUM
- SUM
- AND

Connections:
- NUM1 to NUM
- NUM2 to SUM
- SUM to AND
- AND receives another input from outside the diagram

Legend:
- Arrowheads indicate connections
- Circles represent operations (e.g., addition, AND)
Instructions

ADD/AND

ADD
0 0 0 1 | Dst | Src1 | 0 0 0 | Src2

AND
0 1 0 1 | Dst | Src1 | 0 0 0 | Src2

ADD RF, R1, R2

This zero means "register mode"
Instructions

ADD 0 0 0 1 | Dst | Src1 1  | Imm5

AND 0 1 0 1 | Dst | Src1 1  | Imm5

AND R1, R1, 3

AND R1, R1, 0

Note: Immediate field is sign-extended.

-2^4 -16 -15

ALU

Register File

Sext

IR[4:0]

Instruction Reg
Using Operate Instructions

With only ADD, AND, NOT...

- How do we subtract? $R_3 = R_1 - R_2$
  $R_2 = \overline{R_2} + (R_2) \rightarrow R_3 = R_1 + R_2$
  $R_2 = R_2 + 1$
  $R_3 = R_1$ or $R_2$

- How do we OR? $R_3 = R_1$ and $R_2$
  $R_3 = \overline{R_3}$

- How do we copy from one register to another? $R_3 = R_2$

- How do we initialize a register to zero? $R_1 = R_1$ and $0$
Data Movement Instructions

Load -- read data from memory to register
- LD: PC-relative mode
- LDR: base+offset mode
- LD1: indirect mode

Store -- write data from register to memory
- ST: PC-relative mode
- STR: base+offset mode
- ST1: indirect mode

Load effective address -- compute address, save in register
- LEA: immediate mode
  - does not access memory
Addressing Modes

- How memory is addressed.
- Different instructions use different addressing modes.
- Some instructions support more than one addressing mode.
LC-3 Addressing Modes

- **PC-Relative**
  - Address is a displacement from PC

- **Indirect**
  - Use PC-Relative to get address from memory

- **Base plus Offset**
  - Use contents of a register as base address and add offset to find address (most common for load/store architectures)

\[ 4 + 3 = 7 \]
\[ 334 - 72 \]
Addressing Modes

PC-Relative

The Problem:

We want to specify address directly in the instruction

– But an address is 16 bits, and so is an instruction!
– After subtracting 4 bits for opcode and 3 bits for register, we have only 9 bits available for address.
The Solution:
Use the 9 bits as a signed offset from the current PC.

9 bits allows the offset range to be:

\[-256 \leq \text{offset} \leq +255\]

We can now form any address \( X \), such that:

\[(PC - 256) \leq X \leq (PC + 255)\]

Remember that the PC is incremented as part of the FETCH phase; This is done before the EVALUATE ADDRESS stage.
PC-Relative Addressing Mode

**LD (Load Data)**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Dst</td>
<td></td>
<td>PCoffset</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Instruction Reg → Sext → IR[8:0]
2. Sext → +
3. + → MAR
4. MAR → Memory
5. Memory → Dst
6. Dst → Register File
7. Register File → LD

PC-Relative Addressing Mode

ST (Store Data)

ST 0 0 1 1 Src PC offset 9

Diagram:
- PC
- Register File
- Memory
- Instruction Reg
- Sext
- IR[8:0]
- MAR
- MDR
Indirect

The Problem:
With PC-relative mode, we can only address data within 256 words of the instruction.

– What about the rest of memory? How do we access it?
Indirect Addressing Mode

Solution #1:
- Read address from memory location, then load/store to that address.

First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.
Indirect Addressing Mode

LDI

1010 Dst PCoffset9

PC

1

Sext

1 R[8:0]

Instruction Reg

Register File

Dst

Memory

MAR

MDR

CMPE-012/L

Indirect Addressing Mode

**STI**

1011 0 1 1 1 5 4 3 2 1 0

---

1. IR[8:0]
2. Sext
3. Memory
4. MAR
5. Register File
6. MDR

Instruction Reg

---

Base + Offset

Remember The Problem:
With PC-relative mode, can only address data within 256 words of the instruction.

- What about the rest of memory? How do we access it?
Solution #2:

– Use a register to generate a full 16-bit address.

4 bits for opcode, 3 bits for src/dest register, 3 bits for base register – the remaining 6 bits are used as a signed offset.

– Offset is sign-extended before adding to base register.
Base + Offset Addressing Mode

LDR

\[ \text{LDR } 0 \ 1 \ 1 \ 0 \ \text{Dst} \ \text{Base} \ \text{offset6} \]

PC-relative

Instruction Reg

Register File

Memory

IR[5:0]

Sext

MAR

MDR
Base + Offset Addressing Mode

STR

STR | 0 1 1 1 | Src | Base | offset 6

Diagram showing the process of the STR instruction, including the Instruction Register (IR), Sext, Register File, Base, MAR, MDR, and Memory.
Load Effective Address

Computes address like PC-relative (PC plus signed offset) and stores the result into a register.

Note: The *address* is stored in the register, not the contents of the memory location.
LEA (Immediate)

LEA 1110 Dst PCoffset9

Labels

LEA, Foo

LEA, R1, 44
### Example Code

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x30F6</td>
<td>1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 1</td>
<td>LEA</td>
<td>R1 ← PC - 3 = x30F4</td>
</tr>
<tr>
<td>x30F7</td>
<td>0 0 0 1 0 1 0 0 0 1 1 1 0 1 1 0</td>
<td>ADD</td>
<td>R2 ← R1 + 14 = x3102</td>
</tr>
</tbody>
</table>
| x30F8  | 0 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 | ST | M[PC - 5] ← R2  
M[x30F4] ← x3102 |
| x30F9  | 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 | AND | R2 ← 0 |
| x30FA  | 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1 | ADD | R2 ← R2 + 5 = 5 |
| x30FB  | 0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0 0 | STR | M[R1+14] ← R2  
M[x3102] ← 5 |
| x30FC  | 1 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 | LDI | R3 ← M[M[x30F4]]  
R3 ← M[x3102]  
R3 ← 5 |
Control Instructions

Used to alter the sequence of instructions. This is done by changing the PC.

\[ \text{if} \,(x=0) \]

Conditional Branch

– branch is *taken* if a specified condition is true
  - signed offset is added to PC to yield new PC
– else, the branch is *not taken*
  - PC is not changed, points to the next sequential instruction
Unconditional Branch (or Jump)

- always changes the PC

TRAP

- changes PC to the address of an OS “service routine” Keyboard/terminal
- 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

\[ 2p \quad 2np \]

- 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.

Label
If the branch is not taken, the next sequential instruction is executed.

**Labels**

- Note: PC has already been incremented by FETCH stage.

- Note: Target must be within 256 words of BR instruction.
BR (PC-Relative)

BR: 0 0 0 0 n z p PCoffset9

BRn
BRz

ADD R1, R1, 0
R1 = R1 + 0
Example: Using a Branch

Compute sum of 12 integers
Numbers start at location x3100. Program starts at location x3000.

```
R1 ← x3100
R3 ← 0
R2 ← 12
```

R2=0?

NO

```
R4 ← M[R1]
R3 ← R3+R4
R1 ← R1+1
R2 ← R2-1
```

YES
### Example: Using a Branch

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\times3000)</td>
<td>1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1</td>
<td>LEA</td>
<td>(R1 \leftarrow \times3100) (PC+0xFF)</td>
</tr>
<tr>
<td>(\times3001)</td>
<td>0 1 0 1 0 1 1 0 1 1 1 1 0 0 0 0</td>
<td>AND</td>
<td>(R3 \leftarrow 0)</td>
</tr>
<tr>
<td>(\times3002)</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 0</td>
<td>AND</td>
<td>(R2 \leftarrow 0)</td>
</tr>
<tr>
<td>(\times3003)</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 1 0 1 0</td>
<td>ADD</td>
<td>(R2 \leftarrow 12)</td>
</tr>
<tr>
<td>(\times3004)</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1</td>
<td>BRz</td>
<td>If Z, goto (\times300A) (PC+5)</td>
</tr>
<tr>
<td>(\times3005)</td>
<td>0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0</td>
<td>LDR</td>
<td>Load next value to R4</td>
</tr>
<tr>
<td>(\times3006)</td>
<td>0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 0 0</td>
<td>ADD</td>
<td>(R3 \leftarrow R4 + R3)</td>
</tr>
<tr>
<td>(\times3007)</td>
<td>0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>Increment R1 (pointer)</td>
</tr>
<tr>
<td>(\times3008)</td>
<td>0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>(\times3009)</td>
<td>0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 1 0</td>
<td>BRnzp</td>
<td>Goto (\times3004) (PC-6)</td>
</tr>
</tbody>
</table>

**opcode**
Jump is an **unconditional branch** -- *always* taken.
- Target address is the contents of a register.
- Allows any target address.
TRAP

1 1 1 1 0 0 0 0 trapvect8

Calls a service routine, identified by 8-bit “trap vector.”

<table>
<thead>
<tr>
<th>Vector</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>x23</td>
<td>input a character from the keyboard</td>
</tr>
<tr>
<td>x21</td>
<td>output a character to the monitor</td>
</tr>
<tr>
<td>x25</td>
<td>halt the program</td>
</tr>
</tbody>
</table>

When routine is done, PC is set to the instruction following TRAP.
Another Example

"Hello World"

Count the occurrences of a character in an array

- Program begins at location x3000
- Read character from keyboard
- Load each character from an array
  - An array is a sequence of memory locations
  - Starting address of array is stored in the memory location immediately after the program
- If array character equals input character, increment counter
- End of array is indicated by a special ASCII value: EOT (x04)
- At the end, print the number of characters and halt (lets assume there will be less than 10 occurrences of the character)
Flow Chart

Count = 0
(R2 = 0)

Ptr = 1st character of array
(R3 = M[x3012])

Input char from keybd
(TRAP x23)

Load char from array
(R1 = M[R3])

Done?
(R1 != EOT)

YES

Convert count to ASCII character
(R0 = x30, R0 = R2 + R0)

NO

Match?
(R1 != R0)

YES

Incr Count
(R2 = R2 + 1)

Halt
(TRAP x25)

NO

Load next char from array
(R3 = R3 + 1, R1 = M[R3])

Print count
(TRAP x21)
\[ n \equiv 32k \equiv R2 = R1 \equiv 21 \]

\[ R2 - R1 = 0 \]

\[ 'H' - 4 \]
\[ 72 - 4 \]
\[ 68 \]

\[ '1' - 'H' \]
\[ 108 + 72 \]
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>×3000</td>
<td>010101010010101000000000</td>
<td>AND</td>
<td>R2 ← 0 (counter)</td>
</tr>
<tr>
<td>×3001</td>
<td>00100110000010000000</td>
<td>LD</td>
<td>R3 ← M[×3012] (ptr)</td>
</tr>
<tr>
<td>×3002</td>
<td>11111000000001000111</td>
<td>TRAP</td>
<td>Input to RO (TRAP ×23)</td>
</tr>
<tr>
<td>×3003</td>
<td>011000101100000000000000</td>
<td>LDR</td>
<td>R1 ← M[R3]</td>
</tr>
<tr>
<td>×3004</td>
<td>000110000011111000000000</td>
<td>ADD</td>
<td>R4 ← R1 - 4 (EOT)</td>
</tr>
<tr>
<td>×3005</td>
<td>000000100000101000000000</td>
<td>BRz</td>
<td>If Z, goto ×300E</td>
</tr>
<tr>
<td>×3006</td>
<td>10010010011111111111</td>
<td>NOT</td>
<td>R1 ← NOT R1</td>
</tr>
<tr>
<td>×3007</td>
<td>000100100111000000001</td>
<td>ADD</td>
<td>R1 ← R1 + 1</td>
</tr>
<tr>
<td>×3008</td>
<td>000100100110000000000000</td>
<td>ADD</td>
<td>R1 ← R1 + RO</td>
</tr>
<tr>
<td>×3009</td>
<td>0000101000000000001</td>
<td>BRnp</td>
<td>If N or P, goto ×300B</td>
</tr>
</tbody>
</table>

**opcodes**
### Program (page 2 of 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x300A</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R2 ← R2 + 1</td>
</tr>
<tr>
<td>x300B</td>
<td>0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R3 ← R3 + 1</td>
</tr>
<tr>
<td>x300C</td>
<td>0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0</td>
<td>LDR</td>
<td>R1 ← M[R3]</td>
</tr>
<tr>
<td>x300D</td>
<td>0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0</td>
<td>BRnzp</td>
<td>Goto x3004</td>
</tr>
<tr>
<td>x300E</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td>LD</td>
<td>RO ← M[x3013]</td>
</tr>
<tr>
<td>x300F</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td>ADD</td>
<td>RO ← RO + R2</td>
</tr>
<tr>
<td>x3010</td>
<td>1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1</td>
<td>TRAP</td>
<td>Print RO (TRAP x21)</td>
</tr>
<tr>
<td>x3011</td>
<td>1 1 1 1 0 0 0 0 0 0 0 1 0 0 1 0 1</td>
<td>TRAP</td>
<td>HALT (TRAP x25)</td>
</tr>
<tr>
<td>x3012</td>
<td>Starting Address of File</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3013</td>
<td>0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0</td>
<td>Data</td>
<td>ASCII x30 ('0')</td>
</tr>
</tbody>
</table>

**opcode**
LC-3
Data Path

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
- There are 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
- There are 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.
Summary of ISA

- Instruction Set Architecture
- 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).
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

Edge cases
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

- 1st Generation: Machine Level
- 2nd Generation: Assembly Level
- 3rd Generation: High Level
FORTRAN

FORTRAN PROGRAM

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',
     & 'INTEGER YOU ENTER',/1X, 'THROUGH THE',
     & 'KEYBOARD. TYPE 999 TO INDICATE END OF DATA.',/)
40 FORMAT (1X)
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
Third Generation Languages

IDENTIFICATION DIVISION.
PROGRAM-ID. AVERAGE.
* COBOL PROGRAM
* AVERAGING INTEGERS ENTERED THROUGH THE KEYBOARD.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-IDENTIFIER. MFP 9000.
OBJECT-IDENTIFIER. MFP 9000.
*
DATA DIVISION.
*
FILE SECTION.
*
WORKING-STORAGE SECTION.
01 AVERAGE FIO 9.99.
01 COUNTER FIO 9(02).
01 NUMBER-ITD FIO 9(03).
01 SUM-ITD FIO 9(03).
01 BLANK-LINE FIO X(90).
*
PROCEDURE DIVISION.
*
100-CONTROL-Routine.
PERFORM 200-DISPLAY-INSTRUCTIONS.
PERFORM 300-INITIALIZATION-Routine.
PERFORM 400-ENTER-AND-ADD UNTIL NUMBER-ITD = 999.
PERFORM 500-Calculate-AVERAGE.
PERFORM 600-Display-RESULTS.
STOP RUN.
*
200-Display-Instructions.
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-INITIALIZATION-Routine.
DISPLAY PLEASE ENTER A NUMBER.
ACCEPT NUMBER-ITD.
*
400-ENTER-AND-ADD.
ADD NUMBER-ITD TO SUM-ITD.
ADD 1 TO COUNTER.
DISPLAY PLEASE ENTER THE NEXT NUMBER.
ACCEPT NUMBER-ITD.
*
500-Calculate-AVERAGE.
DIVIDE SUM-ITD 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)
Third Generation Languages

'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
Third Generation Languages

```cpp
// 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++
LC-3
Assembly Language
(Ch7)
LC-3 is a load/store RISC architecture

- Has 8 general registers
- Has a flat 16-bit addressing range
- Has a 16-bit word size
- Load variables from memory to register
Syntax of LC-3

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

LC-3 has 2 basic data types
- Integer
- Character

Both take 16-bits of space (a word) though a character is only 8-bits in size.
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

0x3006 foo :
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>Where to start in placing things in memory</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>
.ORIG

• Tells simulator where to put your code in memory
• Only one allowed per program
• PC gets set to this address at start up
• Similar to the "main" in "C"
**“C”**

```
    type   varname;
```

type is

- int (integer)
- char (character)
- float (floating point)

**“LC-3”**

```
    varname .FILL value
```

value is required – the initial value
LC-3 Syntax

```
add R1, R2, R3
```

flag .FILL x0001
counter .FILL x0002
letter .FILL x0041 ; A
letters .FILL x4241 ; BA

- One declaration per line
- Always declaring 16-bits, the word size of LC-3
- Don’t mix in with your code, will be treated like an instruction
.BLKW

- Tells assembler to set aside some number of sequential memory locations
- Useful for arrays
- Can be initialized
Examples of .BLKW:

; set aside 3 locations
.BLKW 3

; set aside 1 location and label it.
Bob .BLKW 1
Bob .FILL 1
; set aside 1 location, label and initialize to 4.
Num .BLKW 1 #4
LC-3 Syntax

**.STRINGZ**

- Used to declare a string of characters
- Is terminated by `\x0000`
- One character per memory location

Example:

```
hello .STRINGZ "Hello World!"
```
LC-3 Syntax

.END

- Tells the assembler where your program ends
- Only one allowed in your program
<table>
<thead>
<tr>
<th>&quot;LC-3&quot;</th>
<th>&quot;C&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, X</td>
<td></td>
</tr>
<tr>
<td>LD R2, Y</td>
<td></td>
</tr>
<tr>
<td>ADD R3, R2, #0</td>
<td>Z = Y</td>
</tr>
<tr>
<td>ADD R3, R1, R2</td>
<td>Z = X + Y</td>
</tr>
<tr>
<td>???</td>
<td>Z = X - Y</td>
</tr>
<tr>
<td>???</td>
<td>Z = X * Y</td>
</tr>
<tr>
<td>???</td>
<td>Z = X / Y</td>
</tr>
<tr>
<td>ST R3, Z</td>
<td></td>
</tr>
</tbody>
</table>

An immediate is a value specified in an instruction, not by a .FILL declaration.
Simple LC-3 program

\[
\begin{align*}
R2 &= 0 \\
R0 &= 4 \\
R1 &= 2 \\
\text{Loop} \\
R2 &= 0 + 4 = 4 \\
R1 &= R1-1 = 1 \\
\text{Done} \\
R2 &= 8 \\
R1 &= 0 \\
\text{Result} \\
\text{Zero} \\
M0 \\
M1
\end{align*}
\]

\begin{align*}
&.ORIG x3000 \\
&LD R2, Zero \\
&LD R0, M0 \\
&LD R1, M1 \\
&Done \\
&BR R2, R2, R0 \\
&R1, R1, -1 \\
&\text{Loop} \\
&ST R2, Result \\
&\text{HALT} \\
&.FILL x0000 \\
&.FILL x0000 \\
&.FILL x0004 \\
&.FILL x0002 \\
&.END
\end{align*}

- What does this program do?
- What is in “Result” at the end?
Program Execution

- Assembler translates to executable – machine language
- Linker combines multiple LC-3 files – if any
- Loader puts executable into memory and makes the CPU jump to first instruction, .ORIG.
- Executes
- When executing is done returns control to OS
  - Or simulator or monitor
- Load again to run again with different data
  - In this case, assemble again, too, since data is in program.
HLL – if/else statements...

if (condition)
    statement;
else
    statement;
“C” if (count < 0)
    count = count + 1;

“LC-3”
    LD R0, count
    BRpz
    ADD R0, R0, #1
    greatzero ; next instruction goes here
Loops can be built out of IF’s – WHILE:

“C”

while (count > 0) {
    a = a + count;
    count--;
}
"LC-3"

LD R1, a
LD R0, count
while BRnz
ADD R1, R1, R0
ADD R0, R0, #-1
BR while
endwhile
ST R1, a
ST R0, count
endwhile
Procedure Calls

Simple procedure calls require 2 instructions:

“JSR” or “JSRR” Jump Service Routine
• Saves the return address into R7

“RET” Jump Return
• Be careful with registers!!
• Cannot nest unless R7 is saved elsewhere
• Cannot be recursive without a stack
Example

JSR Sub ; calls procedure
...

; calculate R2 = R0-R1
Sub NOT R2, R1
ADD R2, R2, #1
ADD R2, R2, R0
RET ; returns to line after
; JSR Sub
Repeat loops

"C"
/* do statement while expression is TRUE */
/* when expression is FALSE, exit loop */
do {
    if (a < b)
        a++;  
    if (a > b)
        a--;  
} while (a != b)
LC-3 Programming

"LC-3"

```
LD R0, a
LD R1, b
JSR Sub ; R2 = R0-R1
repeat BRpz secondif
ADD R0, R0, #1
JSR Sub
secondif BRnz until
ADD R0, R0, #-1
until JSR Sub
repeat BRnp
```
For loops

"C"

for ( l = 3; l <= 8; l++)
{ a = a+l;}
"LC-3"

; R0=a, R1=l, R2=temp

LD R0, a
AND R1, R1, #0 ; init l to zero
ADD R1, R1, #3 ; now make 3

for
ADD R2, R1, #-8
BRp endfor
ADD R0, R0, R1 ; a=a+l
ADD R1, R1, #1 ; l++
BR for ; same as BRnzp

endfor
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>PUTC</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 \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 x25

or

HALT