ISA

You will be allowed to use the one page summary.
LC-3 Overview: Instruction Set

Opcodes

• 15 opcodes
• *Operate* 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, 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*
### ADD/AND (Immediate)

#### ADD

<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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Dst</td>
<td>Src1</td>
<td>1</td>
<td>Imm5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### AND

<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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Dst</td>
<td>Src1</td>
<td>1</td>
<td>Imm5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembly Ex:**

Add R3, R3, #1

---

**Note:** Immediate field is **sign-extended**.
Load and Store instructions

Example:  LD R1, Label1
R1 is loaded from memory location labelled Label1

Example:  LDI R1, Label1
R1 is loaded from address found at location Label1

Example:  LDR R1, R4, #1
R1 is loaded from address pointed by R4 with offset 1.

Store instructions use the same addressing modes, except the register contents are written to a memory location.
LEA (Immediate)

LEA (Immediate)

LEA

\[ \text{LEA} \quad 1 \ 1 \ 1 \ 0 \quad \text{Dst} \quad \text{PC} + \text{offset} \ 9 \]

- **Assembly Ex:**
  - LEA R1, Lab1

- **Used to initialize a pointer.**
Condition Codes

LC-3 has three condition code registers:  
- **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

Assembly Ex:  
BRz, Label
Count characters in a “file”: Flow Chart

Count = 0
(R2 = 0)

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

Input char from keybd
(TRAP x23)

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

Done?
(R1 ?= EOT)

Match?
(R1 ?= R0)

Incr Count
(R2 = R2 + 1)

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

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

Print count
(TRAP x21)

HALT
(TRAP x25)
Count Characters

Symbol Table:
fill yourself

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>x3004</td>
</tr>
<tr>
<td>GETCHAR</td>
<td>x300B</td>
</tr>
<tr>
<td>OUTPUT</td>
<td></td>
</tr>
<tr>
<td>ASCII</td>
<td></td>
</tr>
<tr>
<td>PTR</td>
<td>x3013</td>
</tr>
</tbody>
</table>

.ORIG  x3000
AND    R2, R2, #0 ; init counter
LD     R3, PTR ; R3 pointer to chars
GETC   ; R0 gets char input
LDR    R1, R3, #0 ; R1 gets first char
TEST   ADD R4, R1, #-4 ; Test for EOT
BRz    OUTPUT ; done?

; Test character for match, if so increment count.

NOT    R1, R1
ADD    R1, R1, R0 ; If match, R1 = xFFF
NOT    R1, R1 ; if match, R1 = x0000
BRnp   GETCHAR ; No match, no increment
ADD    R2, R2, #1

; Get next character from file.

GETCHAR ADD R3, R3, #1 ; Point to next char.
LDR    R1, R3, #0 ; R1 gets next char
BRnzp  TEST

; Output the count.

OUTPUT LD  R0, ASCII ; Load ASCII template
ADD    R0, R0, R2 ; Covert binary to ASCII
OUT    ; ASCII code is displayed
HALT   ; Halt machine

; Storage for pointer and ASCII template
ASCII  .FILL x0030
PTR    .FILL x4000
.END
Assembler Directives

Pseudo-operations

- do not refer to operations executed by program
- used by assembler
- look like instruction, but “opcode” starts with dot

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>address</td>
<td>starting address of program</td>
</tr>
<tr>
<td>.END</td>
<td></td>
<td>end of program</td>
</tr>
<tr>
<td>.BLKW</td>
<td>n</td>
<td>allocate n words of storage</td>
</tr>
<tr>
<td>.FILL</td>
<td>n</td>
<td>allocate one word, initialize with value n</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>n-character string</td>
<td>allocate n+1 locations, initialize w/characters and null terminator</td>
</tr>
</tbody>
</table>
Practice

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Machine Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R3, PTR</td>
<td>0010 011 000010000</td>
</tr>
<tr>
<td>ADD R4, R1, #−4</td>
<td></td>
</tr>
<tr>
<td>LDR R1, R3, #0</td>
<td></td>
</tr>
<tr>
<td>BRnp GETCHAR</td>
<td>0000 101 00000001</td>
</tr>
</tbody>
</table>
Memory

$2^k \times m$ array of stored bits

Address

- unique ($k$-bit) identifier of location

Contents

- $m$-bit value stored in location

Basic Operations:

LOAD

- read a value from a memory location

STORE

- write a value to a memory location
**TRAP Instruction**

<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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>trapvect8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Trap vector**

- identifies which system call to invoke
- 8-bit index into table of service routine addresses
  - in LC-3, this table is stored in memory at \(0x0000 - 0x00FF\)
  - 8-bit trap vector is zero-extended into 16-bit memory address

**Where to go**

- lookup starting address from table; place in PC

**How to get back**

- save address of next instruction (current PC) in R7
**Trap Codes**

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>TRAP x25</td>
<td>Halt execution and print message to console.</td>
</tr>
<tr>
<td>IN</td>
<td>TRAP x23</td>
<td>Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>OUT</td>
<td>TRAP x21</td>
<td>Write one character (in R0[7:0]) to console.</td>
</tr>
<tr>
<td>GETC</td>
<td>TRAP x20</td>
<td>Read one character from keyboard. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>PUTS</td>
<td>TRAP x22</td>
<td>Write null-terminated string to console. Address of string is in R0.</td>
</tr>
</tbody>
</table>
Example: Using the TRAP Instruction

```assembly
.ORIG x3000
LD R2, TERM ; Load negative ASCII ‘7'
LD R3, ASCII ; Load ASCII difference
AGAIN
TRAP x23 ; input character
ADD R1, R2, R0 ; Test for terminate
BRz EXIT ; Exit if done
ADD R0, R0, R3 ; Change to lowercase
TRAP x21 ; Output to monitor...
BRnzp AGAIN ; ... again and again...
TERM .FILL xFFC9 ; -‘7'
ASCII .FILL x0020 ; lowercase bit
EXIT TRAP x25 ; halt
.END
```
Example: Output Service Routine

```
.ORIG x0430 ; syscall address
ST R7, SaveR7 ; save R7 & R1
ST R1, SaveR1

; ----- Write character
TryWrite LDI R1, DSR ; get status
BRzp TryWrite ; look for bit 15 on
WriteIt STI R0, DDR ; write char

; ----- Return from TRAP
Return LD R1, SaveR1 ; restore R1 & R7
LD R7, SaveR7
RET ; back to user

DSR .FILL xF3FC
DDR .FILL xF3FF
SaveR1 .FILL 0
SaveR7 .FILL 0
.END
```

stored in table, location x21
# JSR Instruction

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>[PC]</td>
</tr>
<tr>
<td>14</td>
<td>[PC]</td>
</tr>
<tr>
<td>13</td>
<td>[PC]</td>
</tr>
<tr>
<td>12</td>
<td>[PC]</td>
</tr>
<tr>
<td>11</td>
<td>[PC]</td>
</tr>
<tr>
<td>10</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>9</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>8</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>7</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>6</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>5</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>4</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>3</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>2</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>1</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
<tr>
<td>0</td>
<td>Source Address (PC + Sext(IR[10:0]))</td>
</tr>
</tbody>
</table>

Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.

- saving the return address is called “linking”
- target address is PC-relative (PC + Sext(IR[10:0]))
- bit 11 specifies addressing mode
  - if =1, PC-relative: target address = PC + Sext(IR[10:0])
  - if =0, register: target address = contents of register IR[8:6]
Example: Negate the value in R0

2sComp
NOT R0, R0 ; flip bits
ADD R0, R0, #1 ; add one
RET ; return to caller

To call from a program (within 1024 instructions):

; need to compute R4 = R1 - R3
ADD R0, R3, #0 ; copy R3 to R0
JSR 2sComp ; negate
ADD R4, R1, R0 ; add to R1
...

Note: Caller should save R0 if we’ll need it later!
RET (JMP R7)

How do we transfer control back to instruction following the TRAP?

We saved old PC in R7.

• **JMP R7** gets us back to the user program at the right spot.
• LC-3 assembly language lets us use **RET** (return) in place of “**JMP R7**”.

Must make sure that service routine does not change R7, or we won’t know where to return.
Stack
Memory Usage

- Instructions are stored in code segment
- Global data is stored in data segment
- Local variables, including arrays, uses stack
- Dynamically allocated memory uses heap

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>▼</td>
</tr>
<tr>
<td>Heap</td>
<td>▼</td>
</tr>
<tr>
<td></td>
<td>▼</td>
</tr>
<tr>
<td>Stack</td>
<td>▼</td>
</tr>
</tbody>
</table>

- Code segment is write protected
- Initialized and uninitialized globals
- Stack size is usually limited
- Stack generally grows from higher to lower addresses.
Basic Push and Pop Code

For our implementation, stack grows downward (when item added, TOS moves closer to 0)

**Push R0**

ADD R6, R6, #-1 ; decrement stack ptr
STR R0, R6, #0 ; store data (R0)

**Pop R0**

LDR R0, R6, #0 ; load data from TOS
ADD R6, R6, #1 ; decrement stack ptr

Sometimes a Pop only adjusts the SP.
Run-Time Stack

Memory

Before call  During call  After call

main  Watt  main

R6  R5  R6  R5  R6  R5
Activation Record

int NoName(int a, int b)
{
    int w, x, y;
    .
    .
    return y;
}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>4</td>
<td>NoName</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
<td>5</td>
<td>NoName</td>
</tr>
<tr>
<td>w</td>
<td>int</td>
<td>0</td>
<td>NoName</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>-1</td>
<td>NoName</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>-2</td>
<td>NoName</td>
</tr>
</tbody>
</table>

Compiler generated Symbol table. Offset relative to FP R5
Example Function Call

int Volta(int q, int r)
{
    int k;
    int m;
    ...
    return k;
}

int Watt(int a)
{
    int w;
    ...
    w = Volta(w,10);
    ...
    return w;
}
Summary of LC-3 Function Call Implementation

1. Caller pushes arguments (last to first).
2. Caller invokes subroutine (JSR).
3. Callee allocates return value, pushes R7 and R5.
4. Callee allocates space for local variables.
5. Callee executes function code.
6. Callee stores result into return value slot.
7. Callee pops local vars, pops R5, pops R7.
8. Callee returns (JMP R7).
9. Caller loads return value and pops arguments.
10. Caller resumes computation…
Example: LC-3 Code

; i is 1st local (offset 0), ptr is 2nd (offset -1)

; i = 4;

AND R0, R0, #0 ; clear R0
ADD R0, R0, #4 ; put 4 in R0
STR R0, R5, #0 ; store in i

; ptr = &i;
ADD R0, R5, #0 ; R0 = R5 + 0 (addr of i)
STR R0, R5, #-1 ; store in ptr

; *ptr = *ptr + 1;
LDR R0, R5, #-1 ; R0 = ptr
LDR R1, R0, #0 ; load contents (*ptr)
ADD R1, R1, #1 ; add one
STR R1, R0, #0 ; store result where R0 points
Input/Output
Input from Keyboard

When a character is typed:
- its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
- the “ready bit” (KBSR[15]) is set to one
- keyboard is disabled -- any typed characters will be ignored

When KBDR is read:
- KBSR[15] is set to zero
- keyboard is enabled
Basic Input Routine

Polling

new char?

YES

read character

NO

```
POLL   LDI  R0, KBSRPtr
       BRzp POLL
       LDI  R0, KBDRPtr

...  

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```
Output to Monitor

When Monitor is ready to display another character:

- the “ready bit” (DSR[15]) is set to one

When data is written to Display Data Register:

- DSR[15] is set to zero
- character in DDR[7:0] is displayed
- any other character data written to DDR is ignored (while DSR[15] is zero)
Keyboard Echo Routine

Usually, input character is also printed to screen.
  • User gets feedback on character typed and knows its ok to type the next character.

```
POLL1    LDI  R0, KBSRPtr
         BRzp POLL1
         LDI  R0, KBDRPtr
POLL2    LDI  R1, DSRPtr
         BRzp POLL2
         STI  R0, DDRPtr
         ...
KBSRPtr  .FILL xFE00
KBDRPtr  .FILL xFE02
DSRPtr   .FILL xFE04
DDRPtr   .FILL xFE06
```
Interrupt-Driven I/O

External device can:
(1) Force currently executing program to stop;
(2) Have the processor satisfy the device’s needs; and
(3) Resume the stopped program as if nothing happened.

*Interrupt is an unscripted subroutine call, triggered by an external event.*
Interrupt-Driven I/O

To implement an interrupt mechanism, we need:

- A way for the I/O device to signal the CPU that an interesting event has occurred.
- A way for the CPU to test whether the interrupt signal is set and whether its priority is higher than the current program.

Generating Signal

- Software sets "interrupt enable" bit in device register.
- When ready bit is set and IE bit is set, interrupt is signaled.
Testing for Interrupt Signal

CPU looks at signal between STORE and FETCH phases. If not set, continues with next instruction. If set, transfers control to interrupt service routine.

More details in Chapter 10.
Processor State

- Must be saved before servicing an interrupt.
- What state is needed to completely capture the state of a running process?

Processor Status Register

<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>P</td>
<td>PL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td>Z</td>
<td>P</td>
</tr>
</tbody>
</table>

LC-3: 8 priority levels (PL0-PL7)

Program Counter

- Pointer to next instruction to be executed.

Registers

- All temporary state of the process that’s not stored in memory.
Direct Memory Access Structure

high-speed I/O devices
Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
Only one interrupt is generated per block