Chapter 5
The LC-3

Instruction Set Architecture

ISA = 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

ISA provides all information needed for someone that
wants to write a program in *machine language*
  - or translate from a high-level language to machine language.

LC-3 Overview: Memory and Registers

- **Memory**
  - address space: $2^{16}$ locations (16-bit addresses)
  - addressability: 16 bits

- **Registers**
  - temporary storage, accessed in a single machine cycle
  - accessing memory takes longer than a single cycle
  - eight general-purpose registers: R0 - R7
    - each 16 bits wide
    - how many bits to uniquely identify a register?
  - other registers
    - not directly addressable, but used by (and affected by) instructions
    - PC (program counter), condition codes
**LC-3 Overview: Instruction Set**

- **Opcodes**
  - 15 opcodes, 3 types of instructions
  - **Operate**: ADD, AND, NOT
  - **Data movement**: LD, LDI, LDR, LEA, ST, STR, STI
  - **Control**: BR, JSR/JSRR, JMP, RTI, TRAP

- **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.

**NOT (Register)**

Note: Src and Dst could be the same register.

**ADD/AND (Register)**

ADD [0 0 0 1 Dest Src1 0 0 0 Src2]
AND [0 1 0 1 Dest Src1 0 0 0 Src2]
ADD/AND (Immediate)

Note: Immediate field is sign-extended.

Using Operate Instructions

- With only ADD, AND, NOT...
  - How do we subtract?
  - How do we OR?
  - How do we copy from one register to another?
  - How do we initialize a register to zero?

Data Movement Instructions

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

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

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

PC-Relative Addressing Mode

- 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 9 bits available for address.

- Solution:
  - Use the 9 bits as a signed offset from the current PC.
  - 9 bits: -256 ≤ offset ≤ +255
  - Can form address such that: PC - 256 ≤ X ≤ PC + 255

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

- With PC-relative mode, can only address data within 256 words of the instruction.
  - What about the rest of memory?

**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.
With PC-relative mode, can only address data within 256 words of the instruction.

- What about the rest of memory?

**Solution #2:**

- Use a register to generate a full 16-bit address.
- 4 bits for opcode, 3 for src/dest register, 3 bits for base register — remaining 6 bits are used as a **signed offset**.
- Offset is sign-extended before adding to base register.

---

**LDR (Base + Offset)**

**STR (Base + Offset)**
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.

Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x30F6</td>
<td>1 1 1 0 1 1 1 1 1 1 1 0 1</td>
<td>$R1 \leftarrow PC - 3 = x30F4$</td>
</tr>
<tr>
<td>x30F7</td>
<td>0 0 0 1 1 0 0 0 1 1 0 1 1 0</td>
<td>$R2 \leftarrow R1 + 14 = x3102$</td>
</tr>
<tr>
<td>x30F8</td>
<td>0 0 1 1 1 0 1 1 1 1 1 0 1 1</td>
<td>( M[PC - 5] \leftarrow R2 )</td>
</tr>
<tr>
<td>x30F9</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 0 0 0</td>
<td>$R2 \leftarrow 0$</td>
</tr>
<tr>
<td>x30FA</td>
<td>0 0 0 1 0 0 0 1 0 1 0 0 0 1 0</td>
<td>$R2 \leftarrow R2 + 5 = 5$</td>
</tr>
<tr>
<td>x30FB</td>
<td>0 1 1 1 0 1 0 0 1 1 0 1 0 1 1 0</td>
<td>( M[R1+14] \leftarrow R2 )</td>
</tr>
<tr>
<td>x30FC</td>
<td>1 0 1 0 1 1 1 1 1 1 1 0 1 1 1</td>
<td>( R3 \leftarrow M[x3102] )</td>
</tr>
</tbody>
</table>

Control Instructions

- Used to alter the sequence of instructions (by changing the Program Counter)
  - **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 instruction
  - **Unconditional Branch (or Jump)**
    - always changes the PC
  - **TRAP**
    - changes PC to the address of an OS “service routine”
    - routine will return control to the next instruction (after the TRAP)
## 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

## Branch Instruction

- Branch specifies one or more condition codes.
- If the set bit is specified, the branch is taken.
  - PC-relative addressing: target address is made by adding signed offset (IR[8:0]) to current PC.
  - Note: PC has already been incremented by FETCH stage.
  - Note: Target must be within 256 words of BR instruction.
- If the branch is not taken, the next sequential instruction is executed.

## BR (PC-Relative)

What happens if bits [11:9] are all zero? All one?

## Using Branch Instructions

- Compute sum of the first 12 integers.
  - Program starts at location x3000.
  - R3 ← 0
  - R2 ← 12
  - R2 ← R3 + R2
  - R2 ← R2 - 1
  - R3 ← R3 + R2
Sample Program

- The solution to the previous problem is posted on the website.

JMP (Register)

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

<table>
<thead>
<tr>
<th>JMP</th>
<th>1 1 0 0 0 0 0 0</th>
</tr>
</thead>
</table>

- Target address is the contents of a register.

```
JMP 1 1 0 0 0 0 0 0
```

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

TRAP

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

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

- 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 an 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

Count = 0 (R2 = 0)
Ptr = 1st file character (R3 = M[x3012])
Input char from keybd (TRAP x23)
Done? (R1 ?= EOT)
Load char from file (R1 = M[R3])
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)

Program (1 of 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0 1 0 1 0 1 0 0 0 1 0 0 0 0 0</td>
<td>R2 = 0 (counter)</td>
</tr>
<tr>
<td>x3001</td>
<td>0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 0</td>
<td>R3 ← M[x3012] (ptr)</td>
</tr>
<tr>
<td>x3002</td>
<td>1 1 1 0 1 0 0 0 1 0 0 0 0 1 1 1</td>
<td>Input to R0 (TRAP x23)</td>
</tr>
<tr>
<td>x3003</td>
<td>0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0</td>
<td>R1 ← M[R2]</td>
</tr>
<tr>
<td>x3004</td>
<td>0 0 0 0 1 1 0 0 0 1 1 1 1 1 1 0</td>
<td>R4 ← R1 = 1 (EOT)</td>
</tr>
<tr>
<td>x3005</td>
<td>0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0</td>
<td>If Z, goto x300E</td>
</tr>
<tr>
<td>x3006</td>
<td>1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1</td>
<td>R1 ← NOT R1</td>
</tr>
<tr>
<td>x3007</td>
<td>0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1</td>
<td>R1 ← R1 + 1</td>
</tr>
<tr>
<td>x3008</td>
<td>0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0</td>
<td>R1 ← R1 + R0</td>
</tr>
<tr>
<td>x3009</td>
<td>0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1</td>
<td>If N or P, goto x300B</td>
</tr>
</tbody>
</table>

Program (2 of 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x300A</td>
<td>0 0 0 1 0 1 0 1 0 0 0 0 0 1</td>
<td>R2 ← R2 + 1</td>
</tr>
<tr>
<td>x300B</td>
<td>0 0 0 1 0 1 1 0 1 1 0 0 0 0 1</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>R1 ← M[R3]</td>
</tr>
<tr>
<td>x300D</td>
<td>0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0</td>
<td>Goto x3004</td>
</tr>
<tr>
<td>x300E</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td>R6 ← 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>R0 ← R0 + R2</td>
</tr>
<tr>
<td>x3010</td>
<td>1 1 1 1 0 0 0 0 0 1 0 0 0 0 0 1</td>
<td>Print R0 (TRAP x23)</td>
</tr>
<tr>
<td>x3011</td>
<td>1 1 1 0 0 0 0 0 1 0 0 1 0 1 0 1</td>
<td>HALT (TRAP x25)</td>
</tr>
<tr>
<td>x3012</td>
<td>Starting Address of File</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>ASCII x30 (&quot;0&quot;)</td>
</tr>
</tbody>
</table>
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)

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

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)

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.