Chapter 5

The LC-3 Instruction Set Architecture

- ISA Overview
- Operate instructions
- Data Movement instructions
- Control Instructions
- LC-3 data path
A specific ISA: The LC-3

- We have:
  - Reviewed data encoding and simple digital concepts
  - Introduced a general model for computer organization
  - Discussed a general model for computer execution (the instruction cycle)

- Now its time to focus on a specific example: The LC-3 ISA
  - The LC-3 uses a 16-bit word and is word-addressable
    - How large can the LC-3 memory be?
  - All instructions are represented as 16-bit words
  - All data is represented as 16-bit words
    - Native Data Type: only 2’s complement integer
  - The LC-3 uses eight 16-bit GPRs: R0-R7
  - The LC-3 maintains three 1-bit status codes: N, Z, P
LC-3 Instructions

- Three types of instructions
  - **Operate**: Manipulate data directly
    - ADD, AND, NOT
  - **Data Movement**: Move data between memory and registers
    - LD, LDI, LDR, LEA, ST, STI, STR
  - **Control**: Change the sequence of instruction execution
    - BR, JMP/RET, JSR/JSSR, TRAP, RTI

- Addressing Modes:
  - Immediate (non-memory addressing mode)
  - Register (non-memory addressing modes)
  - Direct (memory addressing mode)
  - Indirect (memory addressing mode)
  - Base+Offset (memory addressing mode)
LC-3 Instruction word

- LC-3 Instructions word: 16 bits
  - 4-bit opcode => 16 instructions (RISC)
  - remaining 12 bits specify operand(s), according to the addressing mode proper to each instruction.
  - Opcode: Specifies what the instruction does
    - IR[15:12]: 4 bits allow 16 instructions
    - specifies the instruction to be executed
  - Operands: Specifies what the instruction acts on
    - IR[11:0]: contains specifications for:
      - Registers: 8 GPRs (i.e. require 3 bits for addressing)
      - Address Generation bits: Offset (11 or 9 or 6 bits) (more later)
      - Immediate value: 5 bits
- Examples
  - ADD        DR,      SR1,      SR2       ;   DR ← (SR1) + (SR2)
    [15:12]    [11:9]    [8:6]      [2:0]
  - LDR        DR,      BaseR,   Offset     ;   DR ← Mem[BaseR + Offset]
    [15:12]    [11:9]    [8:6]    [5:0]
Addressing Modes

- The LC-3 supports five addressing modes:
  - the operand is located:
    - in the instruction itself (#1: immediate a.k.a literal)
    - in a register (#2)
    - in memory:
      - Note: the effective address (ea) is the memory location of the operand
      - the ea is encoded in the instruction (#3: direct, or PC-relative)
      - a pointer to the ea is encoded in the instruction (#4: indirect)
      - a pointer to the ea is stored in a register (#5: base+offset, a.k.a. relative)
  
- LC-3 Operate instructions use only immediate and register modes
- LC-3 Data movement instructions use all five modes
Operate Instructions - 1

- Arithmetic and Logic
  - Arithmetic: add, subtract, multiply, divide (the LC-3 only has add)
  - Logic: and, or, not, xor (the LC-3 only has and, not)

- LC-3: NOT, ADD, AND

```
<table>
<thead>
<tr>
<th>dest reg</th>
<th>src reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>NOT</td>
<td>R3</td>
</tr>
<tr>
<td>0 1 0 1 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dest reg</th>
<th>src reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>ADD</td>
<td>R3</td>
</tr>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dest reg</th>
<th>src reg</th>
<th>src reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
<td>0 1 1</td>
<td>0 1 0 0 0 1 0 1</td>
</tr>
<tr>
<td>ADD</td>
<td>R3</td>
<td>R2</td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- AND (opcode = 0101) has the same structure as ADD
"Hiding" in the control unit, we find the Processor Status Register (PSR)

<table>
<thead>
<tr>
<th>Priv</th>
<th>Priority</th>
<th>N</th>
<th>Z</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10 – 8</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Many instructions set the condition codes according to their results
- Z = result was zero
- N = result was negative
- P = result was positive

Note that *one and only one* of these conditions can be true
NOT: Bitwise Logical NOT

- **UNARY OPERATION**

- Assembler Inst.
  \[ \text{NOT DR, SR} \quad ; \quad DR \leftarrow \text{NOT (SR)} \]

- Encoding
  \[ \begin{array}{ccc}
    1001 & DR & SR \quad 111111 \\
  \end{array} \]

- Example
  \[ \begin{array}{c}
    x977F \\
    1001 \ 0111 \ 0111 \ 1111 \\
    1001 \ 011 \ 101 \ 11 \ 1111 \\
    \text{NOT R3, R5} \\
  \end{array} \]

  - Note: Condition codes are set.
Control signals for NOT

**NOT R3, R5**

- SR1 = 101; ALUK = NOT; GateALU = 1
- Wait for signal propagation/sub-cycle tick
- DR = 011; LD.REG = 1
- Wait for signal propagation/sub-cycle tick
- RELEASE ALL

**x977F**
1001 0111 0111 1111
1001 011 101 11 1111
NOT R3, R5
ADD: Two's complement 16-bit Addition

- **BINARY OPERATION**

- **Assembler Instruction**

  *(register addressing)*
  
  ADD DR, SR1, SR2 ; DR = SR1 + SR2

  *(immediate addressing)*
  
  ADD DR, SR1, imm5 ; DR = SR1 + Sext(imm5)

- **Encoding**

  0001 DR SR1 0 00 SR2
  0001 DR SR1 1 imm5

- **Examples**

  ADD R1, R4, R5 0001 001 100 00 101
  ADD R1, R4, # -2 0001 001 100 1 11110

  - Note: Condition codes are set
Control signals for ADD (immed5)

**ADD R1,R4,-2**

- SR1 = 100; SR2MUX = IR; ALUK = ADD; GateALU = 1
- *Wait for signal propagation/sub-cycle tick*
- DR = 001; LD.REG = 1
- *Wait for signal propagation/sub-cycle tick*
- RELEASE ALL

\[
\begin{align*}
\times133E \\
0001\ 0011\ 0011\ 1110 \\
0001\ 001\ 100\ 1\ 11110
\end{align*}
\]

ADD R1,R4,-2
Control signals for ADD (register)

**ADD R1, R4, R5**

- SR1 = 100; SR2 = 101; SR2MUX = REGfile; ALUK = ADD; GateALU = 1
- *Wait for signal propagation/sub-cycle tick*
- DR = 001; LD.REG = 1
- *Wait for signal propagation/sub-cycle tick*
- RELEASE ALL

```
x1305
0001 0011 0000 0101
0001 001 100 0 00 101
ADD R1, R4, R5
```
AND: Bitwise Logical AND

- **Assembler Instruction**
  
  \[
  \text{AND } DR, SR1, SR2 \quad ; DR = SR1 \text{ AND } SR2 \\
  \text{AND } DR, SR1, \text{imm5} \quad ; DR = SR1 \text{ AND } \text{Sext(imm5)}
  \]

- **Encoding**
  
  \[
  0101 \quad \text{DR} \quad \text{SR1} \quad 0 \quad 00 \quad \text{SR2} \\
  0101 \quad \text{DR} \quad \text{SR1} \quad 1 \quad \text{imm5}
  \]

- **Examples**
  
  \[
  \text{AND} \quad R2, R3, R6 \\
  \text{AND} \quad R2, R2, \#0 \quad ; \text{Clear R2 to 0}
  \]

- Note: Condition codes are set.
The complete instruction cycle

ADD R1,R4,R5 0001 0011 0000 0101

- GatePC = 1; LD.MAR = 1; MEM.EN, R.W = Read, LD.MDR = 1
- Wait, then release all
- GateMDR = 1; LD.IR = 1; PCMUX = +1; LD.PC
- Now the control unit can see the IR, so it "knows" what to do next
- Wait, then release all
- SR1 = 100; SR2 = 101; SR2MUX = REGfile; ALUK = ADD; GateALU = 1
- DR = 001; LD. REG = 1
- Wait, then release all

Now, where is the PC pointing?
Data Movement Instructions - 1

- **Move Data**
  - from register to memory => store
    - nominated register is Source
  - from memory to register => load
    - nominated register is Destination
  - The LC-3 cannot move data from memory to memory
  - also to/from I/O devices (later)

- **LC-3 Load/Store Instructions**
  - LD, LDI, LDR, LEA, ST, STI, STR
  - Format:

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
    opcode      DR or SR           Address generator bits
```
Data Movement Instructions - 2

- What do you do with the 9 Addr Gen bits IR[8:0]?
- LC-3 Load/Store Addressing modes:
  - direct or PC-Relative: LD & ST
    - The EA is the Sign Extended Addr. Generator added to the current value of the Program Counter - i.e.
      - \( EA = (PC) + \text{SEXT}( \text{IR}[8:0]) \)
      - \( DR \Leftarrow \text{Mem}[EA] \)
  - indirect: LDI & SDI
    - \( EA = \text{Mem}[ (PC) + \text{SEXT}( \text{IR}[8:0]) ] \)
    - \( DR \Leftarrow \text{Mem}[EA] \)
  - base + offset (relative): LDR & STR (BaseReg is specified by IR[8:6])
    - \( EA = \text{BaseReg} + \text{SEXT}( \text{IR}[5:0]) \)
    - \( DR \Leftarrow \text{Mem}[EA] \)
  - immediate: LEA
    - Does NOT access Memory! It “LOADS” an address for later use!; - i.e.
      - \( DR \Leftarrow (PC) + \text{SEXT}( \text{IR}[8:0]) \)
LD: Load Direct

- **Assembler Inst.**
  ```
  LD  DR, LABEL  ; DR <= Mem[LABEL]
  ```

- **Encoding**
  ```
  0010  DR  PCoffset9
  ```

- **Examples**
  ```
  LD  R2, param  ; R2 <= Mem[param]
  ```

**Notes:**
- The LABEL must be within +256/-255 of the instruction.
- Condition codes are set.
ST: Store Direct

- **Assembler Inst.**
  
  \[
  \text{ST SR, LABEL} \quad ; \quad \text{Mem[LABEL]} <= \text{SR}
  \]

- **Encoding**
  
  0011 SR offset9

- **Examples**
  
  \[
  \text{ST R2, VALUE} \quad ; \quad \text{Mem[VALUE]} <= \text{R2}
  \]

**Notes:**
- The LABEL must within +/- 256 lines of the instruction.
- Condition codes are NOT set.
LDI: Load Indirect

- **Assembler Inst.**
  LDI DR, LABEL  ; DR <= Mem[Mem[LABEL]]

- **Encoding**
  1010 DR PCoffset9

- **Examples**
  LDI R2, POINTER  ; R2 <= Mem[Mem[POINTER]]

Notes:
- The LABEL must be within +256/-255 lines of the instruction.
- Condition codes are set.
STI: Store Indirect

• **Assembler Inst.**

  \[
  \text{STI SR, LABEL} \quad ; \quad \text{Mem}[\text{Mem}[\text{LABEL}]] \leq \text{SR}
  \]

• **Encoding**

  \[0011 \quad \text{SR} \quad \text{offset9}\]

• **Examples**

  \[
  \text{STI R2, POINTER} \quad ; \quad \text{Mem}[\text{Mem}[\text{POINTER}]] \leq \text{R2}
  \]

**Notes:**
- The LABEL must be within +/- 256 lines of the instruction.
- Condition codes are NOT set.
LDR: Load Base+Index

- **Assembler Inst.**
  
  LDR DR, BaseR, offset ; DR <= Mem[ BaseR+SEXT( IR[5:0] ) ]

- **Encoding**
  
  0110 DR BaseR offset6

- **Examples**
  
  LD R2, R3, #15 ; R2 <= Mem[(R3)+15]

Notes:

- The 6 bit offset is a 2's complement number, so range is -32 to +31.
- Condition codes are set.
STR: Store Base+Index

- **Assembler Inst.**
  
  \[ \text{STR SR, BaseR, offset6 ; Mem[BaseR+SEXT(offset6)] } \leq (SR) \]

- **Encoding**
  
  \[ 0111 \text{ SR BaseR offset6} \]

- **Examples**
  
  \[ \text{STR R2, R4, #15 ; Mem[R4+15] } \leq (R2) \]

  Notes: The offset is sign-extended to 16 bits. Condition codes are not set.
LEA: Load Effective Address

- **Assembler Inst.**
  
  ```
  LEA DR, LABEL ; DR <= LABEL
  ```

- **Encoding**
  
  1110 DR offset9
  
  (i.e. address of LABEL = (PC) + SEXT(offset9))

- **Examples**
  
  ```
  LEA R2, DATA ; R2 gets the address of DATA
  ```

**Notes:**
- The LABEL must be within +/- 256 lines of the instruction.
- Condition codes are set.

**LEA R5, # -3**
What is the EA for the following instructions?

Given: PC = x2081, R6 = x2035, LOC = x2044, Mem[x2044] = x3456

0110 001 110 00 1100

**LDR R1, R6, #12**
Register addressing:
EA = (R6)+12 = x2035 + x000C
= x2041

1010 010 1 1100 0011

**LDI R2, LOC**
Indirect addressing:
EA = Mem[x2044] = x3456

**ADD R1, R3, R2**

**ADD R5, R1, #15**

**LD R1, LOC**

Immediate addressing:
DR <= ?

Direct addressing:
DR <= ?
Control Instructions

- Change the Program Counter
  - Conditionally or unconditionally
  - Store the original PC (subroutine calls) or not (goto)

- LC-3 Control Instructions
  - BRx, JMP/RET, JSR/JSRR, TRAP, RTI
    - JMP/RET & JSRR use base+offset addressing with zero offset
    - BRx uses PC-Relative addressing with 9-bit offset
    - JSR uses PC-Relative addressing with 11-bit offset
JMP: Jump or GoTo

- **Assembler Inst.**
  
  ```
  JMP BaseR
  ```
  
  Take the next instruction from the address stored in BaseR

- **Encoding**
  
  ```
  1100 000 BaseR 00 0000
  ```

- **Example**
  
  ```
  JMP R5 ; if (R5) = x3500, the address x3500 is written to the PC
  ```
BR: Conditional Branch

- **Assembler Inst.**
  
  BRx LABEL
  
  where x = n, z, p, nz, np, zp, or nzp

  Branch to LABEL iff the selected condition code are set

- **Encoding**
  
  0000 n z p PCoffset9

- **Examples**
  
  BRzp LOOP ; branch to LOOP if previous op returned zero or positive.
Building loops using BR

Counter control

Add 12 integers in array @ 0x3100
There are two Branches: what are they?

Sentinel control

Add integers in null terminated array @ 0x3100
TRAP Instruction (The basics)

- Traps will be covered in detail later
  - Used to invoke a system routine.
  - Trap vector table: a list of locations of the service call routines.
  - TRAP has one operand, the trap vector:
    - PC is set to the value stored at that location of the vector table.

- But we need some I/O services now:
  * x20: R0 <- input ASCII character from stdin
  * x23: R0 <- input ASCII character from stdin with prompt & echo
  * x21: output ANSI character to stdout <- R0
  * x25: halt the program
  - More details later

- Assembler Inst.
  - TRAP trapvec

- Encoding
  - 1111 0000 trapvect8

- Examples
  - TRAP x23

Notes:
- R7 <= (PC) (for eventual return)
- PC <= Mem[Zext(trapvect8)]
Immediate & Register Operands

- **Immediate**

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15:12]</td>
<td>[11:9]</td>
</tr>
<tr>
<td>[8:6]</td>
<td>[5]</td>
</tr>
<tr>
<td>[4:0]</td>
<td>imm</td>
</tr>
</tbody>
</table>

  - If bit 5 = 1, the value in IR[4:0] (“immediate”) is sign extended (SEXT) to 16 bits and added to the contents of the source register SR1 (IR[8:6]).

- **Register**

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15:12]</td>
<td>[11:9]</td>
</tr>
<tr>
<td>[8:6]</td>
<td>[5]</td>
</tr>
<tr>
<td>[2:0]</td>
<td>SR2</td>
</tr>
</tbody>
</table>

  - If bit 5 = 0, the contents of source register SR2 (IR[2:0]) are added to the contents of source register SR1 (IR[8:6]).
  - In both cases, the result goes to the destination register DR (IR[11:9]).
Memory Addressing Modes

- **Direct addressing (PC-Relative)**
  
  - Effective address = (PC) + SEXT( IR[8:0] )
  
  - Operand location must be within approx. 256 locations of the instruction
    - Actually between +256 and -255 locations of the instruction being executed (why?)
Memory Addressing Modes - 2

- Indirect addressing

<table>
<thead>
<tr>
<th>[15:12]</th>
<th>[11:9]</th>
<th>[8:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI</td>
<td>DR</td>
<td>Addr. Gen. bits</td>
</tr>
</tbody>
</table>

- Same initial mechanism as direct mode (i.e. PC-Relative), but the calculated memory location now contains the address of the operand, (i.e. the ea is indirect):
  - pointer address = (PC) + SEXT( IR[8:0] )
  - effective address = Mem[ (PC) + SEXT( IR[8:0] ) ]

- Note that the memory has to be accessed twice to get the actual operand.
Memory Addressing Modes - 3

- Relative (Base+Offset) addressing

- effective address = (BaseRegister) + offset
  - sign extend (SEXT) the 6 bit offset ([5:0]) to 16 bits
  - add it to the contents of the Base Register ([8:6])

- differences from Direct addressing (PC-Relative):
  - base+offset field is 6 bits, PC-Relative offset field is 9 bits
  - base+offset can address any location in memory, PC-Relative offset only within +/- 256 locations of the instruction.
Data Path - 1

- Global Bus
  - 16-bit, data & address
  - connects all components
  - is shared by all

- Memory
  - Memory Address Register: MAR
    - address of location to be accessed
  - Memory Data Register: MDR
    - data loaded or to be stored
Data Path - 2

- **ALU & Registers**
  - Two ALU sources
    - source 1: register
    - source 2: register or IR
  - Result: goes onto bus, then to DR

- **PC & PCMUX**
  - PC sends address to MAR for instruction fetch
  - PCMUX: a 3:1 mux that selects the new PC
    - Incremented PC
    - offset PC (9 or 11 bits)
    - offset BaseR (6 bits or 0)
    - TRAP vector contents
Data Path - 3

- MARMUX
  - A 2:1 mux that selects the source of MAR
    - PC-Relative addressing
    - BaseR + offset addressing
    - Trap vector
Data Path