LC3 Architecture: LC3 Instruction Set Architecture (ISA) (Chapter 4,5)

What is the Hardware/Software Interface?...ISA
ISA: Types of Instruction

• 1. Operate Instructions
  o process data (addition, logical operations, etc.)

• 2. Data Movement Instructions …
  o move data between memory locations and registers.

• 3. Control Instructions …
  o change the sequence of execution of instructions in the stored program.
    • The default is sequential execution: the PC is incremented by 1 at the start of every Fetch, in preparation for the next one.
    • Control instructions set the PC to a new value during the Execute phase, so the next instruction comes from a different place in the program.
    • This allows us to build control structures such as loops and branches.

Instruction Cycle

• Six phases of the complete Instruction Cycle
  o Fetch: load IR with instruction from memory
  o Decode: determine action to take (set up inputs for ALU, RAM, etc.)
  o Evaluate address: compute memory address of operands, if any
  o Fetch operands: read operands from memory or registers
  o Execute: carry out instruction
  o Store results: write result to destination (register or memory)
Changing the Sequence of Instructions

• In the FETCH phase, we increment the Program Counter by 1.
• What if we don’t want to always execute the instruction that follows this one?
  o examples: loop, if-then, function call
• Need special instructions that change the contents of the PC.
  • These are called control instructions.
    o jumps are unconditional -- always change the PC
    o branches are conditional -- change the PC only if some condition is true (e.g., the result of an ADD is zero)

LC3 Instruction Set Architecture

• The Instruction set architecture (ISA) of the LC3
  o How is each instruction implemented by the control and data paths in the LC3
  o Programming in machine code
  o How are programs executed
    • Memory layout, programs in machine code
• Overview of LC3 microarchitecture
  o How are components connected to build the LC3 processor
    • read Appendix C in detail – required to complete processor design
• Assembly programming
  o Assembly and compiler process
  o Assembly programming with simple programs
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 generally 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
    - *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*
### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Function</th>
<th>Notes</th>
<th>Condition Codes Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD+</td>
<td>0001</td>
<td>DR SR1</td>
<td>0 00 SR2</td>
<td>+</td>
</tr>
<tr>
<td>ADD+</td>
<td>0001</td>
<td>DR SR1</td>
<td>1 imm5</td>
<td>+</td>
</tr>
<tr>
<td>AND+</td>
<td>0101</td>
<td>DR SR1</td>
<td>0 00 SR2</td>
<td>+</td>
</tr>
<tr>
<td>AND+</td>
<td>0101</td>
<td>DR SR1</td>
<td>1 imm5</td>
<td>+</td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n x p</td>
<td>PCoffset9</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>000 BaseR</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td>1</td>
<td>PCoffset11</td>
<td></td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>0 00 BaseR</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>LD+</td>
<td>0010</td>
<td>DR</td>
<td>PCoffset9</td>
<td></td>
</tr>
<tr>
<td>LDI+</td>
<td>1010</td>
<td>DR</td>
<td>PCoffset9</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Function</th>
<th>Notes</th>
<th>Condition Codes Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR+</td>
<td>0110</td>
<td>DR BaseR</td>
<td>offset6</td>
<td></td>
</tr>
<tr>
<td>LEA+</td>
<td>1110</td>
<td>DR</td>
<td>PCoffset9</td>
<td></td>
</tr>
<tr>
<td>NOT+</td>
<td>1001</td>
<td>DR SR</td>
<td>111111</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>000 111</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td>0000000000000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>0011</td>
<td>SR</td>
<td>PCoffset9</td>
<td></td>
</tr>
<tr>
<td>STI</td>
<td>1011</td>
<td>SR</td>
<td>PCoffset9</td>
<td></td>
</tr>
<tr>
<td>STR</td>
<td>0111</td>
<td>SR BaseR</td>
<td>offset6</td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>0000 trapvect8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reserved</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

+ Indicates instructions that modify condition codes.
Operate Instructions

• Only three operations: ADD, AND, NOT

• Source and destination operands are registers
  o These instructions do not reference memory.
  o ADD and AND can use “immediate” mode, where one operand is hard-wired into the instruction.

• Dataflow diagram associated with each instruction.
  o Illustrates when and where data moves to accomplish the desired operation
  o Used to design the datapath

Operate Instructions

• NOT

  \[
  \text{NOT}^+ \begin{array}{c|c|c|c|c}
  \hline
  \text{1001} & \text{DR} & \text{SR} & \text{111111} \\
  \hline
  \end{array}
  \]

• Addressing mode?
  o Where are the operands
  o DR and SR specify the register number/address
    - 3 bits needed to specify address of 8 registers
Specifying instruction semantics

- Is there a formal way to specify the transfers/controls needed to implement each instruction?
  - The register transfers needed.
- Is there a formal language
  - Can you specify these transfers by writing a program
- Hardware Description Languages (HDL)
  - Verilog, VHDL, ...
  - Processor design and specification today is done using a HDL
    - Use software to design hardware!
- Data flow and timing
  - Data transfers labelled at each step/cycle
    - What transfers and between which devices at each cycle
Operate Instructions

- **ADD, AND**

  - **Addressing Mode?**
    - Where are the operands?
    - Two modes: (1) register and (2) immediate value

  - **ADD**: \( \text{Dst} = \text{Src1} + \text{Src2} \)
    - \( R2 = R1 + R3 \)
  
  - **AND**: \( \text{Dst} = \text{Src1} \text{ AND } \text{Src2} \)
    - \( R2 = R1 \text{ AND } R3 \)

ADD/AND (Register)

This zero means "register mode"

- **ADD**: \( \begin{array}{cccccc}
0 & 0 & 0 & 1 & Dst & Src1 \\
0 & 0 & 0 & 0 & Src2
\end{array} \)

- **AND**: \( \begin{array}{cccccc}
0 & 1 & 0 & 1 & Dst & Src1 \\
0 & 0 & 0 & 0 & Src2
\end{array} \)

Register mode: Operands are in registers

If Dst=010, Src1=001, Src2=011

**ADD**:
- Dst= Src1 + Src2
- R2= R1 + R3

**AND**:
- Dst= Src1 AND Src2
- R2 = R1 AND R3
ADD/AND (Immediate)

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

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

Note: Immediate field is **sign-extended**.

If Dst=010, Src1=001, Imm5=00011
ADD R2,R1,#3
R2 = R1 + 3

**Immediate mode:**
One Operand in inst

Data Movement Instructions

- **LD+**
  0010 | DR | PCoffset9

- **LDI+**
  1010 | DR | PCoffset9

- **LDR+**
  0110 | DR | BaseR | offset6

- **LEA+**
  1110 | DR | PCoffset9

- **ST**
  0011 | SR | PCoffset9

- **STI**
  1011 | SR | PCoffset9

- **STR**
  0111 | SR | BaseR | offset6
Data Movement Instructions

- GPR ↔ Memory
- GPR ↔ I/O Devices
- GPR ← Memory ???
- Memory ← GPR ???

Addressing Modes

- Where can operands be found?
  
  If in memory, how to compute the address where the operand/variable is stored?

- Manner in which address is computed leads to three different types of Load/Store instructions
Basic Format

- Address generation bits
- DR or SR
- These encode information on how to form a 16 bit address

DR is register into which memory contents are read (Load)
SR contains data that has to be written into memory (Store)

Data Movement Instructions

- Load -- read data from memory to register
  - LD: PC-relative mode
  - LDR: base+offset mode
  - LDI: indirect mode
- Store -- write data from register to memory
  - ST: PC-relative mode
  - STR: base+offset mode
  - STI: 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: \(-256 \leq \text{offset} \leq +255\)
- Can form any address \(X\), such that: \(PC - 256 \leq X \leq PC + 255\)

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

LD (PC-Relative)
LD – Load PC-relative Addressing

- Suppose instruction is 0010011000000010
- This instruction is stored at address (decimal) 4999
- Ques: What is the result of this instruction?

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>500</td>
</tr>
<tr>
<td>5000</td>
<td>400</td>
</tr>
<tr>
<td>5001</td>
<td>2000</td>
</tr>
<tr>
<td>5002</td>
<td>5001</td>
</tr>
</tbody>
</table>

ST (PC-Relative)

ST | 0 0 1 1 | Src | PCoffset9

Diagram showing the flow of data from Instruction Reg to Memory, including steps for calculation and address calculation.
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.
  - analogy to pointers

LDI (Indirect)
Load Indirect

- Suppose instruction is 101001100000010
- This instruction is stored at address (decimal) 4999
- What is the result of this instruction?

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>500</td>
</tr>
<tr>
<td>5000</td>
<td>400</td>
</tr>
<tr>
<td>5001</td>
<td>2000</td>
</tr>
<tr>
<td>5002</td>
<td>5001</td>
</tr>
</tbody>
</table>

STI (Indirect)

<table>
<thead>
<tr>
<th>STI</th>
<th>Src</th>
<th>PCoffset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

Diagram:

- PC
- Instruction Reg
- Register File
- Memory
Base + Offset Addressing Mode

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

• Solution #2:
  o 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.
  o Offset is sign-extended before adding to base register.

LDR (Base+Offset)

![Diagram of LDR (Base+Offset)]
Suppose instruction is 0110011010000010

This instruction is at address 4999

Suppose Register R2 contains the decimal value 5000

What is the result?

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>500</td>
</tr>
<tr>
<td>5000</td>
<td>400</td>
</tr>
<tr>
<td>5001</td>
<td>2000</td>
</tr>
<tr>
<td>5002</td>
<td>1200</td>
</tr>
</tbody>
</table>

---

STR (Base+Offset)

STR 0111Src Base offset6

Instruction Reg

Register File

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.
**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 0 0 1 1 1 1 1 1 1 1 1 0 1</td>
<td>LEA</td>
</tr>
<tr>
<td>x30F7</td>
<td>0 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0</td>
<td>ADD imm5.</td>
</tr>
<tr>
<td>x30F8</td>
<td>0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1</td>
<td>ST</td>
</tr>
<tr>
<td>x30F9</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 0</td>
<td>AND imm5</td>
</tr>
<tr>
<td>x30FA</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 1</td>
<td>ADD imm5</td>
</tr>
<tr>
<td>x30FB</td>
<td>0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0</td>
<td>STR</td>
</tr>
<tr>
<td>x30FC</td>
<td>1 0 1 0 0 1 1 1 1 1 1 1 0 1 1</td>
<td>LDI</td>
</tr>
</tbody>
</table>

**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 0 0 1 1 1 1 1 1 1 1 1 0 1</td>
<td>R1 ← PC - 3 = x30F4</td>
</tr>
<tr>
<td>x30F7</td>
<td>0 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0 1 0</td>
<td>R2 ← R1 + 14 = x3102</td>
</tr>
<tr>
<td>x30F8</td>
<td>0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 1</td>
<td>M[PC - 5] ← R2</td>
</tr>
<tr>
<td>x30F9</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 0 0</td>
<td>R2 ← 0</td>
</tr>
<tr>
<td>x30FA</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 0 0 1 0 1</td>
<td>R2 ← R2 + 5 = 5</td>
</tr>
<tr>
<td>x30FB</td>
<td>0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0</td>
<td>M[R1+14] ← R2</td>
</tr>
<tr>
<td>x30FC</td>
<td>1 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1</td>
<td>R3 ← M[R1+14]</td>
</tr>
</tbody>
</table>

R3 ← M[x3102] 
R3 ← 5
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 sequential 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 TRAP)

### Table of Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>0000</th>
<th>nzp</th>
<th>PCoffsetn</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>000</td>
<td>BaseR</td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td>1</td>
<td>PCoffset1</td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>000</td>
<td>11</td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td></td>
<td>000000000000</td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>000</td>
<td>trapvect8</td>
</tr>
</tbody>
</table>
**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 computed 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 4 integers.
  Numbers start at location x3100. Program starts at location x3000.
  o Add numbers from location x3100 to x311B
  o Store first address in R2
  o R4 has “counter” – counts down from 4 to 0
  o R1 will store the running Sum
Program

x3000 R2 <- x3100
x3001 R4 <- 0
x3002 R1 <- 0
x3003 R4 <- 4
x3004 BRz x300A /* if R4=0 exit loop */
x3005 R3 <- M[R2]
x3006 R1 <- R1 + R3
x3007 R2 <- R2 + 1
x3008 R4 <- R4 - 1
x3009 BRnzp x3004. /* repeat loop */
x300A Halt /* end of loop */
Program

| x3000 R2 <- x3100 | LEA 1110010011111111 |
| x3001 R1 <- 0 | AND 0101001011100000 |
| x3002 R4 <- 0 | AND 0101100010100000 |
| x3003 R4 <- 12 | ADD 0001100010100100 |
| x3004 BRz x300A | BRz 0000010000000010 |
| x3005 R3 <- M[R2] | LDR 0110011010000000 |
| x3006 R1 <- R1 + R3 | ADD 0001100100100011 |
| x3007 R2 <- R2 + 1 | ADD 0001010010100001 |
| x3008 R4 <- R4 - 1 | ADD 0001100100111111 |
| x3009 BRnzp x3004 | BRnzp 0000101111111010 |

JMP (Register)

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

- Modern computers contain hardware and software protection schemes to prevent user programs from accidentally (or maliciously) interfering with proper system function.

- Suffice it to say, we need a way to communicate with the operating system.

---

**TRAP**

**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.
  - *(We’ll talk about how this works later.)*
The LC-3 ISA: summary

- 16 bit instructions and data
- 2’s complement data type
- Operate/ALU instructions: ADD, NOT, AND
- Data movement Inst: Load and Store
  - Addressing mode: PC-relative, Indirect, Register/Base+Offset,Immediate
- Transfer of control instructions
  - Branch – using condition code registers
  - Jump – unconditional branch
  - Traps, Subroutine calls – discuss later
- Now we look at the LC3 datapath and controller design

Taking stock: what we have now

- Datatypes of machines: Number Representation
  - 2’s complement integers, Floating point
  - Arithmetic on 2’s complement
  - Logic operations
- Digital logic: devices to build the circuits
  - CMOS transistor is the starting point
  - Basic logic gates: AND, OR, NOT, NAND, etc.
  - Combinational logic ‘blocks’: MUX, Decoder, PLA
  - Sequential Logic: storage element, finite state machines
  - Putting it all together to build a simple processor- LC3
- Von Neumann Model of computing
- Instruction set architecture (ISA) of LC3
  - Instructions of a processor – how program execution takes place
  - Addressing modes to data movement, branches, operations
  - Encoding an LC3 instruction
- Next question: how to program the processor
  - Using instructions in the ISA