Agenda for Today & Next Few Lectures

- The von Neumann model
- LC-3: An example of von Neumann machine

- LC-3 and MIPS Instruction Set Architectures
- LC-3 and MIPS assembly and programming

- Introduction to microarchitecture and single-cycle microarchitecture
- Multi-cycle microarchitecture
- Microprogramming
Readings

This week
- Von Neumann Model, LC-3, and MIPS
  - P&P, Chapter 4, 5
  - H&H, Chapter 6
  - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
  - H&H, Appendix B (MIPS instructions)
- Digital Building Blocks
  - H&H, Chapter 5

Next week
- Introduction to microarchitecture and single-cycle microarchitecture
  - P&P, Appendices A and C
  - H&H, Chapter 7.1-7.3
- Multi-cycle microarchitecture
  - P&P, Appendices A and C
  - H&H, Chapter 7.4
- Microprogramming
  - P&P, Appendices A and C
What Will We Learn Today?

- The von Neumann model
  - LC-3: An example of von Neumann machine

- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions

- Instruction formats

- Addressing modes
The Von Neumann Model
Basic Elements of a Computer

- In past lectures we learned
  - Combinational circuits
  - Sequential circuits
- With them, we can build
  - Decision elements
  - Storage elements
- Basic elements of a computer

- To get a task done by a computer we need
  - Computer
  - Data
  - Program: A set of instructions
    - Instruction: the smallest piece of work in a computer
The Von Neumann Model

- Let’s start building the computer

- In order to build a computer we need a model

- John von Neumann proposed a fundamental model in 1946
- It consists of 5 parts
  - Memory
  - Processing unit
  - Input
  - Output
  - Control unit

- Throughout this lecture, we consider two examples of the von Neumann model
  - LC-3
  - MIPS

Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.
The Von Neumann Model

CONTROL UNIT
- PC or IP
- Inst Register

PROCESSING UNIT
- ALU
- TEMP

MEMORY
- Mem Addr Reg
- Mem Data Reg

INPUT
- Keyboard, Mouse, Disk...

OUTPUT
- Monitor, Printer, Disk...

INPUT

OUTPUT
The Von Neumann Model

INPUT
Keyboard, Mouse, Disk...

OUTPUT
Monitor, Printer, Disk...

MEMORY
Mem Addr Reg
Mem Data Reg

PROCESSING UNIT
ALU
TEMP

CONTROL UNIT
PC or IP
Inst Register
Memory

- The memory stores
  - Data
  - Programs

- The memory contains bits
  - Bits are grouped into bytes (8 bits) and words (e.g., 8, 16, 32 bits)

- How the bits are accessed determines the addressability
  - E.g., word-addressable
  - E.g., 8-bit addressable (or byte-addressable)

- The total number of addresses is the address space
  - In LC-3, the address space is $2^{16}$
    - 16-bit addresses
  - In MIPS, the address space is $2^{32}$
    - 32-bit addresses
# Word-Addressable Memory

- Each **data word** has a **unique address**
  - In MIPS, a unique address for each **32-bit data word**
  - In LC-3, a unique address for each **16-bit data word**

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>MIPS memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td>8 9 A B C D E F</td>
<td>Word 0</td>
</tr>
<tr>
<td>0000000001</td>
<td>F 2 F 1 F 0 F 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>0000000002</td>
<td>1 3 C 8 1 7 5 5</td>
<td>Word 2</td>
</tr>
<tr>
<td>0000000003</td>
<td>D 1 6 1 7 A 1 C</td>
<td>Word 3</td>
</tr>
</tbody>
</table>
Byte-Addressable Memory

- Each byte has a **unique address**
  - Actually, MIPS is **byte-addressable**
  - LC-3b is **byte-addressable**, too

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>MIPS memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>00000004</td>
<td>F 2 F 1 F 0 F 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>00000008</td>
<td>1 3 C 8 1 7 5 5</td>
<td>Word 2</td>
</tr>
<tr>
<td>0000000C</td>
<td>D 1 6 1 7 A 1 C</td>
<td>Word 3</td>
</tr>
</tbody>
</table>

How are these four bytes addressed?
Big Endian vs Little Endian

- Jonathan Swift’s *Gulliver’s Travels*
  - Little Endians broke their eggs on the little end of the egg
  - Big Endians broke their eggs on the big end of the egg

 BIG ENDIAN - The way people always broke their eggs in the Lilliput land

 LITTLE ENDIAN - The way the king then ordered the people to break their eggs
## Big Endian vs Little Endian

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Endian</td>
<td>Little Endian</td>
</tr>
<tr>
<td>C D E F</td>
<td>C</td>
</tr>
<tr>
<td>8 9 A B</td>
<td>8</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
</tr>
<tr>
<td>LSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
</tr>
<tr>
<td>LSB</td>
</tr>
</tbody>
</table>

### Big Endian
- MSB = Most Significant Bit
- LSB = Least Significant Bit

### Little Endian
- MSB = Least Significant Bit
- LSB = Most Significant Bit
Big Endian vs Little Endian

Does this really matter?

Answer: No, it is a convention

Qualified answer: No, except when one big-endian system and one little-endian system have to share data
Accessing Memory: MAR and MDR

- There are two ways of accessing memory
  - Reading or loading
  - Writing or storing

- Two registers are necessary to access memory
  - Memory Address Register (MAR)
  - Memory Data Register (MDR)

- To read
  - Step 1: Load the MAR with the address
  - Step 2: Data is placed in MDR

- To write
  - Step 1: Load the MAR with the address and the MDR with the data
  - Step 2: Activate Write Enable signal
The Von Neumann Model

- **INPUT**
  - Keyboard, Mouse, Disk…

- **OUTPUT**
  - Monitor, Printer, Disk…

- **PROCESSING UNIT**
  - ALU
  - TEMP

- **CONTROL UNIT**
  - PC or IP
  - Inst Register

- **MEMORY**
  - Mem Addr Reg
  - Mem Data Reg
The processing unit can consist of many functional units.

We start with a simple Arithmetic and Logic Unit (ALU):
- **LC-3**: ADD, AND, NOT (XOR in LC-3b)
- **MIPS**: add, sub, mult, and, nor, sll, slr,slt...

The ALU processes quantities that are referred to as words:
- **Word length** in LC-3 is 16 bits
- In MIPS it is 32 bits

Temporary storage: Registers
- E.g., to calculate \((A+B)\times C\), the intermediate result of \(A+B\) is stored in a register.
Registers

- **Memory** is big but slow

- **Registers**
  - Ensure fast access to operands
  - Typically one register contains *one word*

- **Register set or file**
  - LC-3 has 8 *general purpose registers* (GPR)
    - R0 to R7: 3-bit register number
    - Register size = Word length = 16 bits
  - MIPS has 32 registers
    - Register size = Word length = 32 bits
# MIPS Register File

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>2-3</td>
<td>function return value</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>4-7</td>
<td>function arguments</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>8-15</td>
<td>temporary variables</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>16-23</td>
<td>saved variables</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>24-25</td>
<td>temporary variables</td>
</tr>
<tr>
<td>$k0–$k1</td>
<td>26-27</td>
<td>OS temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>function return address</td>
</tr>
</tbody>
</table>
The Von Neumann Model

- MEMORY
  - Mem Addr Reg
  - Mem Data Reg

- PROCESSING UNIT
  - ALU
  - TEMP

- CONTROL UNIT
  - PC or IP
  - Inst Register

- INPUT
  - Keyboard, Mouse, Disk...

- OUTPUT
  - Monitor, Printer, Disk...
Input and Output

- Many devices can be used for input and output

- They are called peripherals
  - **Input**
    - Keyboard
    - Mouse
    - Scanner
    - Disks
    - Etc.
  - **Output**
    - Monitor
    - Printer
    - Disks
    - Etc.

- In LC-3, we consider keyboard and monitor
The Von Neumann Model

**MEMORY**
- Mem Addr Reg
- Mem Data Reg

**PROCESSING UNIT**
- ALU
- TEMP

**INPUT**
- Keyboard,
- Mouse,
- Disk...

**CONTROL UNIT**
- PC or IP
- Inst Register

**OUTPUT**
- Monitor,
- Printer,
- Disk...
Control Unit

- The control unit is the conductor of the orchestra

- It conducts the step-by-step process of executing a program

- It keeps track of the instruction being executed with an instruction register (IR), which contains the instruction

- Another register contains the address of the next instruction to execute. It is called program counter (PC) or instruction pointer (IP)
Programmer Visible (Architectural) State

Memory
array of storage locations
indexed by an address

Program Counter
memory address
of the current instruction

Registers
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Instructions (and programs) specify how to transform the values of programmer visible state
The Von Neumann Model

- **INPUT**
  - Keyboard, Mouse, Disk...

- **PROCESSING UNIT**
  - ALU
  - TEMP

- **MEMORY**
  - Mem Addr Reg
  - Mem Data Reg

- **CONTROL UNIT**
  - PC or IP
  - Inst Register

- **OUTPUT**
  - Monitor, Printer, Disk...
LC-3: A Von Neumann Machine
LC-3: A Von Neumann Machine

Finite State Machine (for Generating Control Signals)

Instruction Register

Program Counter

Control signals

Data

Memory Data Register

Memory Address Register

16-bit addressable

ALU: 2 inputs, 1 output

ALU operation

GateALU

8 General Purpose Registers (GPR)

Keyboard KBDR (data), KBSR (status)

Monitor DDR (data), DSR (status)

GateMDR

GateMAR

Figure 4.3 The LC-3 as an example of the von Neumann model
Stored Program & Sequential Execution

- Instructions and data are **stored in memory**
  - Typically the instruction length is the word length

- The processor fetches instructions from memory **sequentially**
  - Fetches one instruction
  - Decodes and executes the instruction
  - Continues with the next instruction

- The address of the current instruction is stored in the **program counter** (PC)
  - If **word-addressable** memory, the processor **increments the PC by 1** (in LC-3)
  - If **byte-addressable** memory, the processor **increments the PC by the word length** (4 in MIPS)
    - In MIPS the OS typically sets the PC to **0x00400000**
A Sample Program Stored in Memory

- A sample MIPS program
  - 4 instructions stored in consecutive words in memory
  - No need to understand the program now. We will get back to it

MIPS assembly

```
lw    $t2, 32($0)
add   $s0, $s1, $s2
addi  $t0, $s3, -12
sub   $t0, $t3, $t5
```

Machine code

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040000C</td>
<td>0 1 6 D 4 0 2 2</td>
</tr>
<tr>
<td>00400008</td>
<td>2 2 6 8 F F F 4</td>
</tr>
<tr>
<td>00400004</td>
<td>0 2 3 2 8 0 2 0</td>
</tr>
<tr>
<td>00400000</td>
<td>8 C 0 A 0 0 2 0</td>
</tr>
</tbody>
</table>

← PC
The Instruction

- An instruction the **most basic unit of computer processing**
  - Instructions are words in the language of a computer
  - Instruction Set Architecture (ISA) is the vocabulary

- The language of the computer can be written as

  - **Machine language**: Computer-readable representation (that is, 0’s and 1’s)
  - **Assembly language**: Human-readable representation

- We learn **LC-3 instructions** and **MIPS instructions**

- Let us start with some examples of instructions
Instruction Types

- There are **three main types of instructions**
  
- Operate instructions
  - Execute instructions in the ALU

- Data movement instructions
  - Read from or write to memory

- Control flow instructions
  - Change the sequence of execution
An Example of Operate Instruction

- **Addition**

<table>
<thead>
<tr>
<th>High-level code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b + c;</code></td>
<td><code>add a, b, c</code></td>
</tr>
</tbody>
</table>

- **add**: mnemonic to indicate the operation to perform

- **b, c**: source operands

- **a**: destination operand

- **a ← b + c**
Registers

- We map variables to registers

**Assembly**

```assembly
add a, b, c
```

**LC-3 registers**

```plaintext
b = R1
c = R2
a = R0
```

**MIPS registers**

```plaintext
b = $s1
c = $s2
a = $s0
```
From Assembly to Machine Code in LC-3

- Addition

**LC-3 assembly**

```
ADD R0, R1, R2
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR1</th>
<th>SR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR1</th>
<th>SR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>000</td>
<td>001</td>
<td>010</td>
</tr>
</tbody>
</table>

0x1042
Instruction Format or Encoding

- LC-3

### Format

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>DR</td>
<td>SR1</td>
<td>0</td>
<td>00</td>
<td>SR2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **OP** = opcode (what the instruction does)
  - E.g., ADD = 0001
    - DR ← SR1 + SR2
  - E.g., AND = 0101
    - DR ← SR1 AND SR2

- **SR1, SR2** = source registers

- **DR** = destination register
# From Assembly to Machine Code in MIPS

## Addition

**MIPS assembly**

```
add  $s0, $s1, $s2
```

**Field Values**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>16</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

**Machine Code**

```
000000 1001 10010 10000 00000 100000
```

```
0x02328020
```
Instruction Formats: R-Type in MIPS

- **R-type**
  - 3 register operands

- **MIPS**

<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- 0 = opcode
- rs, rt = source registers
- rd = destination register
- shamt = shift amount (only shift operations)
- funct = operation in R-type instructions
Reading Operands from Memory

- With the **operate instructions**, such as addition, we tell the computer to **execute arithmetic (or logic) computations** in the ALU.

- We also need instructions to **access the operands from memory**.

- Next, we see how to **read (or load) from memory**.

- **Writing (or storing)** is performed in a similar way, but we will talk about that later.
Reading Word-Addressable Memory

- Load word

High-level code | Assembly
---|---
a = A[i]; | load a, A, i

- **load**: mnemonic to indicate the load word operation

- **A**: base address

- **i**: offset
  - E.g., immediate or literal (a constant)

- **a**: destination operand

- `a ← Memory[A + i]`
Load Word in LC-3 and MIPS

- **LC-3 assembly**

  High-level code
  
  \[ a = A[2]; \]

  LC-3 assembly
  
  \[
  \begin{array}{l}
  \text{LDR R3, R0, } \#2 \\
  \end{array}
  \]

  \[ R3 \leftarrow \text{Memory}[R0 + 2] \]

- **MIPS assembly**

  High-level code
  
  \[ a = A[2]; \]

  MIPS assembly
  
  \[
  \begin{array}{l}
  \text{lw } \$s3, 2(\$s0) \\
  \text{\$s3 }\leftarrow \text{Memory}[\$s0 + 2] \\
  \end{array}
  \]

  These instructions use a particular addressing mode (i.e., the way the address is calculated), called **base+offset**
Load Word in Byte-Addressable MIPS

- **MIPS assembly**

  High-level code: \[ a = A[2]; \]

  MIPS assembly: \[ lw \quad \text{s3, 8($s0)} \]

  \[ s3 \leftarrow \text{Memory}[s0 + 8] \]

- **Byte address is calculated as:** word_address * bytes/word

  - 4 bytes/word in MIPS

  - If LC-3 were byte-addressable (i.e., LC-3b), 2 bytes/word
Instruction Format With Immediate

- LC-3
  - LC-3 assembly
    - `LDR R3, R0, #4`
    - Field Values:
      - OP: 6
      - DR: 3
      - BaseR: 0
      - offset6: 4

- MIPS
  - MIPS assembly
    - `lw $s3, 8($s0)`
    - Field Values:
      - op: 35
      - rs: 16
      - rt: 19
      - imm: 8
How are these Instructions Executed?

- By using instructions we can speak the language of the computer

- Thus, we now know how to tell the computer to
  - Execute computations in the ALU by using, for instance, an addition
  - Access operands from memory by using the load word instruction

- But, how are these instructions executed on the computer?
  - The process of executing an instruction is called is the instruction cycle
The Instruction Cycle

- The instruction cycle is a sequence of steps or **phases**, that an instruction goes through to be executed.
  - **FETCH**
  - **DECODE**
  - **EVALUATE ADDRESS**
  - **FETCH OPERANDS**
  - **EXECUTE**
  - **STORE RESULT**

- **Not all instructions have the six phases**
  - LDR does not require **EXECUTE**
  - ADD does not require **EVALUATE ADDRESS**

- Intel x86 instruction **ADD [eax], edx** is an example of instruction with six phases
After STORE RESULT, a New FETCH

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
The FETCH phase obtains the instruction from memory and loads it into the instruction register.

This phase is common to every instruction type.

Complete description:

- Step 1: Load the MAR with the contents of the PC, and simultaneously increment the PC.
- Step 2: Interrogate memory. This results the instruction to be placed in the MDR.
- Step 3: Load the IR with the contents of the MDR.
FETCH in LC-3

Step 1: Load MAR and increment PC

Step 2: Access memory

Step 3: Load IR with the content of MDR

Figure 4.3  The LC-3 as an example of the von Neumann model
The DECODE phase identifies the instruction.

Recall the decoder (Lecture 6, Slides 26-27)

- A 4-to-16 decoder identifies which of the 16 opcodes is going to be processed.

The input is the four bits IR[15:12]

The remaining 12 bits identify what else is needed to process the instruction.
DECODE identifies the instruction to be processed

Figure 4.3  The LC-3 as an example of the von Neumann model
The EVALUATE ADDRESS phase computes the address of the memory location that is needed to process the instruction.

This phase is necessary in LDR:
- It computes the address of the data word that is to be read from memory.
- By adding an offset to the content of a register.

But not necessary in ADD.
EVALUATE ADDRESS in LC-3

LDR calculates the address by adding a register and an immediate.

Figure 4.3  The LC-3 as an example of the von Neumann model
FETCH OPERANDS

The FETCH OPERANDS phase obtains the source operands needed to process the instruction.

- In LDR
  - Step 1: Load MAR with the address calculated in EVALUATE ADDRESS
  - Step 2: Read memory, placing source operand in MDR

- In ADD
  - Obtain the source operands from the register file
  - In most current microprocessors, this phase can be done at the same time the instruction is being decoded.
LDR loads MAR (step 1), and places the results in MDR (step 2)
EXECUTE

- The EXECUTE phase *executes the instruction*
  - In ADD, it performs addition in the ALU
EXECUTE in LC-3

ADD adds SR1 and SR2
The STORE RESULT phase writes to the designated destination.

Once STORE RESULT is completed, a new instruction cycle starts (with the FETCH phase).
STORE RESULT in LC-3

LDR loads MDR into DR
The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
Changing the Sequence of Execution

- A computer program executes in sequence (i.e., in program order)
  - First instruction, second instruction, third instruction and so on

- Unless we change the sequence of execution

- Control instructions allow a program to execute out of sequence
  - They can change the PC by loading it during the EXECUTE phase
  - That wipes out the incremented PC (loaded during the FETCH phase)
Jump in LC-3

- Unconditional branch or jump

**LC-3**

<table>
<thead>
<tr>
<th>BaseR</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>000</td>
</tr>
</tbody>
</table>

- BaseR = Base register
- PC ← R2 (Register identified by BaseR)

**Variations**
- RET: special case of JMP where BaseR = R7
- JSR, JSRR: jump to subroutine

This is register addressing mode
Jump in MIPS

- Unconditional branch or jump

**MIPS**

\[
j \ target
\]

<table>
<thead>
<tr>
<th>2</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- 2 = opcode
- target = target address

- PC ← PC⁺[31:28] | sign-extend(target) * 4

- Variations
  - jal: jump and link (function calls)
  - jr: jump register

**j uses pseudo-direct addressing mode**

**jr uses register addressing mode**

\(^\dagger\)This is the incremented PC
LC-3 Data Path

Figure 4.3  The LC-3 as an example of the von Neumann model
Control of the Instruction Cycle

- **State 1**
  - The FSM asserts GatePC and LD.MAR
  - It selects input (+1) in PCMUX and asserts LD.PC

- **State 2**
  - MDR is loaded with the instruction

- **State 3**
  - The FSM asserts GateMDR and LD.IR

- **State 4**
  - The FSM goes to next state depending on opcode

- **State 63**
  - JMP loads register into PC

Full state diagram in Patt&Pattel, Appendix C

Figure 4.4 An abbreviated state diagram of the LC-3
The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
LC-3 and MIPS
Instruction Set Architectures
The Instruction Set Architecture

The ISA is the interface between what the software commands and what the hardware carries out.

The ISA specifies:

- The memory organization
  - Address space (LC-3: $2^{16}$, MIPS: $2^{32}$)
  - Addressability (LC-3: 16 bits, MIPS: 32 bits)
  - Word- or Byte-addressable

- The register set
  - R0 to R7 in LC-3
  - 32 registers in MIPS

- The instruction set
  - Opcodes
  - Data types
  - Addressing modes
The Instruction Set

- It defines **opcodes, data types, and addressing modes**
- ADD and LDR have been our first examples

### ADD

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR1</th>
<th>00</th>
<th>SR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Register mode**

### LDR

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

**Base+offset mode**
Opcodes

- Large or small sets of opcodes could be defined
  - E.g, HP Precision Architecture: an instruction for $A \times B + C$
  - E.g, x86: multimedia extensions
  - E.g, VAX: opcode to save all information of one program prior to switching to another program

- Tradeoffs are involved
  - Hardware complexity vs. software complexity

- In LC-3 and in MIPS there are three types of opcodes
  - Operate
  - Data movement
  - Control
## Opcodes in LC-3

![Figure 5.3](image-url) Formats of the entire LC-3 instruction set. Note: * indicates instructions that modify condition codes.
### Opcodes in LC-3b

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD*</td>
<td>0001 DR SR1 A</td>
<td>op.spec</td>
</tr>
<tr>
<td>AND*</td>
<td>0101 DR SR1 A</td>
<td>op.spec</td>
</tr>
<tr>
<td>BR</td>
<td>0000 n z p POffset9</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>1100 000 BaseR 000000</td>
<td></td>
</tr>
<tr>
<td>JSR(R)</td>
<td>0100 A</td>
<td>operand specifier</td>
</tr>
<tr>
<td>LDB*</td>
<td>0010 DR BaseR offset6</td>
<td></td>
</tr>
<tr>
<td>LDW*</td>
<td>0110 DR BaseR offset6</td>
<td></td>
</tr>
<tr>
<td>LEA*</td>
<td>1110 DR POffset9</td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td>1000 000000000000</td>
<td></td>
</tr>
<tr>
<td>SHF*</td>
<td>1101 DR SR A D amount4</td>
<td></td>
</tr>
<tr>
<td>STB</td>
<td>0011 SR BaseR offset6</td>
<td></td>
</tr>
<tr>
<td>STW</td>
<td>0111 SR BaseR offset6</td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td>1111 0000 trapvec18</td>
<td></td>
</tr>
<tr>
<td>XOR*</td>
<td>1001 DR SR1 A</td>
<td>op.spec</td>
</tr>
<tr>
<td>not used</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>not used</td>
<td>1011</td>
<td></td>
</tr>
</tbody>
</table>
### Opcode is 0 in MIPS R-Type instructions. **Funct** defines the operation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000 (0)</td>
<td>sll rd, rt, shamt</td>
<td>shift left logical</td>
<td>[rd] = [rt] &lt;&lt; shamt</td>
</tr>
<tr>
<td>000010 (2)</td>
<td>srl rd, rt, shamt</td>
<td>shift right logical</td>
<td>[rd] = [rt] &gt;&gt; shamt</td>
</tr>
<tr>
<td>000011 (3)</td>
<td>sra rd, rt, shamt</td>
<td>shift right arithmetic</td>
<td>[rd] = [rt] &gt;&gt;&gt; shamt</td>
</tr>
<tr>
<td>000100 (4)</td>
<td>sllv rd, rt, rs</td>
<td>shift left logical variable</td>
<td>[rd] = [rt] &lt;&lt; [rs]_4:0</td>
</tr>
<tr>
<td>000110 (6)</td>
<td>srlv rd, rt, rs</td>
<td>shift right logical variable</td>
<td>[rd] = [rt] &gt;&gt; [rs]_4:0</td>
</tr>
<tr>
<td>000111 (7)</td>
<td>srav rd, rt, rs</td>
<td>shift right arithmetic variable</td>
<td>[rd] = [rt] &gt;&gt;&gt; [rs]_4:0</td>
</tr>
<tr>
<td>010000 (8)</td>
<td>jr rs</td>
<td>jump register</td>
<td>PC = [rs]</td>
</tr>
<tr>
<td>010001 (9)</td>
<td>jalr rs</td>
<td>jump and link register</td>
<td>$ra = PC + 4, PC = [rs]</td>
</tr>
<tr>
<td>011000 (16)</td>
<td>mfhi rd</td>
<td>move from hi</td>
<td>[rd] = [hi]</td>
</tr>
<tr>
<td>011001 (17)</td>
<td>mthi rs</td>
<td>move to hi</td>
<td>[hi] = [rs]</td>
</tr>
<tr>
<td>011010 (18)</td>
<td>mflo rd</td>
<td>move from lo</td>
<td>[rd] = [lo]</td>
</tr>
<tr>
<td>011011 (19)</td>
<td>mtlo rs</td>
<td>move to lo</td>
<td>[lo] = [rs]</td>
</tr>
<tr>
<td>011100 (24)</td>
<td>mult rs, rt</td>
<td>multiply</td>
<td>[hi], [lo] = [rs] × [rt]</td>
</tr>
<tr>
<td>011101 (25)</td>
<td>multu rs, rt</td>
<td>multiply unsigned</td>
<td>[hi], [lo] = [rs] × [rt]</td>
</tr>
<tr>
<td>011101 (26)</td>
<td>div rs, rt</td>
<td>divide</td>
<td>[lo] = [rs]/[rt], [hi] = [rs] % [rt]</td>
</tr>
<tr>
<td>011111 (27)</td>
<td>divu rs, rt</td>
<td>divide unsigned</td>
<td>[lo] = [rs]/[rt], [hi] = [rs] % [rt]</td>
</tr>
</tbody>
</table>

(continued)
### Table B.2 R-type instructions, sorted by funct field—Cont’d

<table>
<thead>
<tr>
<th>Funct</th>
<th>Name</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000 (32)</td>
<td>add rd, rs, rt</td>
<td>add</td>
<td>([rd] = [rs] + [rt])</td>
</tr>
<tr>
<td>100001 (33)</td>
<td>addu rd, rs, rt</td>
<td>add unsigned</td>
<td>([rd] = [rs] + [rt])</td>
</tr>
<tr>
<td>100010 (34)</td>
<td>sub rd, rs, rt</td>
<td>subract</td>
<td>([rd] = [rs] - [rt])</td>
</tr>
<tr>
<td>100011 (35)</td>
<td>subu rd, rs, rt</td>
<td>subract unsigned</td>
<td>([rd] = [rs] - [rt])</td>
</tr>
<tr>
<td>100100 (36)</td>
<td>and rd, rs, rt</td>
<td>and</td>
<td>([rd] = [rs] &amp; [rt])</td>
</tr>
<tr>
<td>100101 (37)</td>
<td>or rd, rs, rt</td>
<td>or</td>
<td>([rd] = [rs] | [rt])</td>
</tr>
<tr>
<td>100110 (38)</td>
<td>xor rd, rs, rt</td>
<td>xor</td>
<td>([rd] = [rs] ^ [rt])</td>
</tr>
<tr>
<td>100111 (39)</td>
<td>nor rd, rs, rt</td>
<td>nor</td>
<td>([rd] = \sim([rs] | [rt]))</td>
</tr>
<tr>
<td>101010 (42)</td>
<td>slt rd, rs, rt</td>
<td>set less than</td>
<td>([rs] &lt; [rt] ? [rd] - 1 : [rd] = 0)</td>
</tr>
<tr>
<td>101011 (43)</td>
<td>sltu rd, rs, rt</td>
<td>set less than unsigned</td>
<td>([rs] &lt; [rt] ? [rd] - 1 : [rd] = 0)</td>
</tr>
</tbody>
</table>

Find the complete list of instructions in the appendix
Data Types

- An ISA supports one or several data types

- LC-3 only supports 2’s complement integers

- MIPS supports
  - 2’s complement integers
  - Unsigned integers
  - Floating point

- Again, tradeoffs are involved
Data Type Tradeoffs

- What is the benefit of having more or high-level data types in the ISA?
- What is the disadvantage?

- Think compiler/programmer vs. microarchitect

- Concept of semantic gap
  - Data types coupled tightly to the semantic level, or complexity of instructions

- Example: Early RISC architectures vs. Intel 432
  - Early RISC (e.g., MIPS): Only integer data type
  - Intel 432: Object data type, capability based machine
Addressing Modes

- An addressing mode is a mechanism for specifying where an operand is located.

- There are five addressing modes in LC-3:
  - Immediate or literal (constant)
    - The operand is in some bits of the instruction
  - Register
    - The operand is in one of R0 to R7 registers
  - Three of them are memory addressing modes
    - PC-relative
    - Indirect
    - Base+offset

- In addition, MIPS has pseudo-direct addressing (for j and jal), but does not have indirect addressing.
Operate Instructions
Operate Instructions

- In **LC-3**, there are three operate instructions
  - NOT is a **unary operation** (one source operand)
    - It executes bitwise NOT
  - ADD and AND are **binary operations** (two source operands)
    - ADD is 2’s complement addition
    - AND is bitwise SR1 & SR2

- In **MIPS**, there are many more
  - Most of R-type instructions (they are **binary operations**)
    - E.g., add, and, nor, xor...
  - I-type versions of the R-type operate instructions
  - F-type operations, i.e., floating-point operations
NOT in LC-3

- NOT assembly and machine code

**LC-3 assembly**

```
NOT R3, R5
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>5</td>
<td>1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>011</td>
<td>001</td>
<td>1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

There is no NOT in MIPS. How is it implemented?
Operate Instructions

- We are already familiar with LC-3’s ADD and AND with register mode (R-type in MIPS)

- Now let us see the versions with one literal (i.e., immediate) operand

- Subtraction is another necessary operation
  - How is it implemented in LC-3 and MIPS?
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Operate Instr. with one Literal in LC-3

- ADD and AND

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR1</th>
<th>1</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td></td>
</tr>
</tbody>
</table>

- **OP** = operation
  - E.g., **ADD** = 0001 (same OP as the register-mode ADD)
    - **DR** ← **SR1** + sign-extend(imm5)
  - E.g., **AND** = 0101 (same OP as the register-mode AND)
    - **DR** ← **SR1** AND sign-extend(imm5)

- **SR1** = source register

- **DR** = destination register

- **imm5** = Literal or immediate (sign-extend to 16 bits)
ADD with one Literal in LC-3

- ADD assembly and machine code

**LC-3 assembly**

```
ADD R1, R4, #−2
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>001</td>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>

For example, if R4 contains the value 6 and R5 contains the value −18, then after the following instruction is executed,

```
ADD R1, R4, #−2
```

R1 will contain the value −12.

If bit [5] is 1, the second source operand is contained within the instruction. In fact, the second source operand is obtained by sign-extend of bits [4:0] to 16 bits before performing the ADD or AND. Figure 5.5 shows the key parts of the data path that are used to perform the instruction ADD R1, R4, #−2.

Since the immediate operand in an ADD or AND instruction must fit in bits [4:0] of the instruction, not all 2’s complement integers can be used as immediate operands. Which integers are OK (i.e., which integers can be used as immediate operands)?
Instructions with one Literal in MIPS

- I-type
  - 2 register operands and immediate
- Some operate and data movement instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- opcode = operation
- rs = source register
- rt =
  - destination register in some instructions (e.g., addi, lw)
  - source register in others (e.g., sw)
- imm = Literal or immediate
Add with one Literal in MIPS

- **Add immediate**

  **MIPS assembly**

  ```mips
  addi $s0, $s1, 5
  ```

  **Field Values**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>16</td>
<td>5</td>
</tr>
</tbody>
</table>

  \[ rt \leftarrow rs + \text{sign-extend}(imm) \]

  **Machine Code**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>10001</td>
<td>10010</td>
<td>0000 0000 0000 0101</td>
</tr>
</tbody>
</table>

  0x22300005
Subtract in LC-3

- **MIPS assembly**

  High-level code
  \[ a = b + c - d; \]

  MIPS assembly
  \[
  \begin{align*}
  &\text{add } $t0, $s0, $s1 \\
  &\text{sub } $s3, $t0, $s2
  \end{align*}
  \]

- **LC-3 assembly**

  High-level code
  \[ a = b + c - d; \]

  LC-3 assembly
  \[
  \begin{align*}
  &\text{ADD } R2, R0, R1 \\
  &\text{NOT } R4, R3 \\
  &\text{ADD } R5, R4, #1 \\
  &\text{ADD } R6, R2, R5
  \end{align*}
  \]

- **Tradeoff in LC-3**
  - More instructions
  - But, simpler control logic
# Subtract Immediate

## MIPS assembly

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b - 3;</code></td>
<td><code>subi $s1, $s0, 3</code></td>
</tr>
</tbody>
</table>

**Is subi necessary in MIPS?**

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b - 3;</code></td>
<td><code>addi $s1, $s0, -3</code></td>
</tr>
</tbody>
</table>

## LC-3

<table>
<thead>
<tr>
<th>High-level code</th>
<th>LC-3 assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b - 3;</code></td>
<td><code>ADD R1, R0, #-3</code></td>
</tr>
</tbody>
</table>
Data Movement Instructions and Addressing Modes
Data Movement Instructions

- In LC-3, there are seven data movement instructions
  - LD, LDR, LDI, LEA, ST, STR, STI

- Format of load and store instructions
  - Opcode (bits [15:12])
  - DR or SR (bits [11:9])
  - Address generation bits (bits [8:0])
  - Four ways to interpret bits, called addressing modes
    - PC-Relative Mode
    - Indirect Mode
    - Base+offset Mode
    - Immediate Mode

- In MIPS, there are only Base+offset and immediate modes for load and store instructions
PC-Relative Addressing Mode

- **LD (Load) and ST (Store)**

  
  - **OP = opcode**
  - E.g., LD = 0010
  - E.g., ST = 0011

  - **DR = destination register in LD**
  - **SR = source register in ST**

  - **LD: DR ← Memory[PC† + sign-extend(PCoffset9)]**

  - **ST: Memory[PC† + sign-extend(PCoffset9)] ← SR**

  
  †This is the incremented PC
**LD in LC-3**

- **LD assembly and machine code**
  
  **LC-3 assembly**

  \[
  \text{LD R2, 0x1AF}
  \]

  **Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0x1AF</td>
</tr>
</tbody>
</table>

  **Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0</td>
<td>0 1 0</td>
<td>1 1 0 1 0 1 1 1 1</td>
</tr>
</tbody>
</table>

  0 15 12 11 9 8 7 6 5 4 3 2 1 0 0 1 0 1 0 1 1 1 1

  The memory address is **only +256 to -255 locations away of the LD or ST instruction**

  **Limitation:** The PC-relative addressing mode cannot address far away from the instruction
Indirect Addressing Mode

- LDI (Load Indirect) and STI (Store Indirect)

- OP = opcode
  - E.g., LDI = 1010
  - E.g., STI = 1011

- DR = destination register in LDI
- SR = source register in STI

- LDI: \( DR \leftarrow \text{Memory}[\text{Memory}[PC^\dagger + \text{sign-extend}(\text{PCoffset9})]] \)

- STI: \( \text{Memory}[\text{Memory}[PC^\dagger + \text{sign-extend}(\text{PCoffset9})]] \leftarrow SR \)

\( ^\dagger \text{This is the incremented PC} \)
LDI in LC-3

- LDI assembly and machine code

**LC-3 assembly**

```
LDI R3, 0x1CC
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0x1CC</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>011</td>
<td>111001100</td>
</tr>
</tbody>
</table>

Now the address of the operand can be anywhere in the memory
Base+Offset Addressing Mode

- LDR (Load Register) and STR (Store Register)

<table>
<thead>
<tr>
<th>OP</th>
<th>DR/SR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- OP = opcode
  - E.g., LDR = 0110
  - E.g., STR = 0111

- DR = destination register in LDR
- SR = source register in STR

- LDR: \( DR \leftarrow \text{Memory[BaseR + sign-extend(offset6)]} \)

- STR: \( \text{Memory[BaseR + sign-extend(offset6)]} \leftarrow SR \)
LDR in LC-3

- LDR assembly and machine code

**LC-3 assembly**

```
LDR R1, R2, 0x1D
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>0x1D</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>001</td>
<td>010</td>
<td>011101</td>
</tr>
</tbody>
</table>

The address of the operand can also be anywhere in the memory.
Base+Offset Addressing Mode in MIPS

- In MIPS, `lw` and `sw` use base+offset mode (or base addressing mode)

High-level code

\[ \text{A}[2] = a; \]

MIPS assembly

\[ \text{sw } \$s3, 8(\$s0) \]

\[ \text{Memory}[\$s0 + 8] \leftarrow \$s3 \]

Field Values

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>16</td>
<td>19</td>
<td>8</td>
</tr>
</tbody>
</table>

- `imm` is the 16-bit offset, which is sign-extended to 32 bits
### An Example Program in MIPS and LC-3

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS registers</th>
<th>LC-3 registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = A[0]; )</td>
<td>( A = $s0 )</td>
<td>( A = R0 )</td>
</tr>
<tr>
<td>( c = a + b - 5; )</td>
<td>( b = $s2 )</td>
<td>( b = R2 )</td>
</tr>
<tr>
<td>( B[0] = c; )</td>
<td>( B = $s1 )</td>
<td>( B = R1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIPS assembly</th>
<th>LC-3 assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw  $t0, 0($s0)</code></td>
<td><code>LDR  R5, R0, #0</code></td>
</tr>
<tr>
<td><code>add  $t1, $t0, $s2</code></td>
<td><code>ADD  R6, R5, R2</code></td>
</tr>
<tr>
<td><code>addi $t2, $t1, -5</code></td>
<td><code>ADD  R7, R6, #-5</code></td>
</tr>
<tr>
<td><code>sw  $t2, 0($s1)</code></td>
<td><code>STR  R7, R1, #0</code></td>
</tr>
</tbody>
</table>
Immediate Addressing Mode

- **LEA (Load Effective Address)**

<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>OP</td>
<td>DR</td>
<td>PCoffset9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>9 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **OP = 1110**
- **DR = destination register**
- **LEA:** \( DR \leftarrow PC^† + \text{sign-extend}(\text{PCoffset9}) \)

What is the difference from PC-Relative addressing mode?

**Answer:** Instructions with PC-Relative mode access memory, but LEA does not.

† This is the incremented PC
LEA in LC-3

- LEA assembly and machine code

**LC-3 assembly**

```
LEA R5, #-3
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>5</td>
<td>0x1FD</td>
</tr>
</tbody>
</table>

**Machine Code**

```
1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1
```

---

Note that the Base+offset addressing mode also allows the address of the operand to be anywhere in the computer's memory.

5.3.4 Immediate Mode

The fourth and last addressing mode used by the data movement in instructions is the immediate (or, literal) addressing mode. It is used only with the load effective address (LEA) instruction.

**LEA** (opcode = 1110) loads the register specified by bits [11:9] of the instruction with the value formed by adding the incremented program counter to the sign-extended bits [8:0] of the instruction. The immediate addressing mode is so named because the operand to be loaded into the destination register is obtained immediately, that is, without requiring any access of memory.

The LEA instruction is useful to initialize a register with an address that is very close to the address of the instruction doing the initializing. If memory location x4018 contains the instruction LEA R5, #−3, and the PC contains x4018, R5 will contain x4016 after the instruction at x4018 is executed.

Figure 5.9 shows the relevant parts of the data path required to execute the LEA instruction. Note that no access to memory is required to obtain the value to be loaded.
Immediate Addressing Mode in MIPS

- In MIPS, `lui` (load upper immediate) loads a 16-bit immediate into the upper half of a register and sets the lower half to 0.

- It is used to assign 32-bit constants to a register.

**High-level code**

\[
a = 0x6d5e4f3c;
\]

**MIPS assembly**

```
# $s0 = a
lui $s0, 0x6d5e
ori $s0, 0x4f3c
```
Addressing Example in LC-3

What is the final value of R3?

<table>
<thead>
<tr>
<th>Address</th>
<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>x30F6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x30F7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x30F8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x30F9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x30FA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x30FB</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x30FC</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

R1← PC–3
R2← R1+14
M[x30F4]← R2
R2← 0
R2← R2+5
M[R1+14]← R2
R3← M[M[x30F4]]
What is the final value of R3?

The final value of R3 is 5.
Control Flow Instructions
Control Flow Instructions

- Allow a program to execute out of sequence

- Conditional branches and jumps
  - Conditional branches are used to make decisions
    - E.g., if-else statement
  - In LC-3, three condition codes are used
  - Jumps are used to implement
    - Loops
    - Function calls
  - JMP in LC-3 and j in MIPS
Condition Codes in LC-3

- Each time one GPR (R0-R7) is written, three single-bit registers are updated

- Each of these condition codes are either set (set to 1) or cleared (set to 0)
  - If the written value is negative
    - N is set, Z and P are cleared
  - If the written value is zero
    - Z is set, N and P are cleared
  - If the written value is positive
    - P is set, N and P are cleared

- SPARC and x86 are examples of ISAs that use condition codes
Conditional Branches in LC-3

- **BRz (Branch if Zero)**

<table>
<thead>
<tr>
<th>BRz</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>n z p</td>
</tr>
</tbody>
</table>

  "n, z, p = which N, Z, and/or P is tested"

  "PCoffset9 = immediate or constant value"

  "if ((n AND N) OR (p AND P) OR (z AND Z))
  then PC ← PC + sign-extend(PCoffset9)"

  "Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp"

\[ ^\dagger \text{This is the incremented PC} \]
Conditional Branches in LC-3

- **BRz**

**BRz 0x0D9**

**What if** \( n = z = p = 1 \)?
(i.e., BRnzp)

**And what if** \( n = z = p = 0 \)?

5.4.2 An Example
We are ready to show by means of a simple example the value of having control instructions in the instruction set.

Suppose we know that the 12 locations x3100 to x310B contain integers, and we wish to compute the sum of these 12 integers.
Conditional Branches in MIPS

- **beq** (Branch if Equal)

  \[
  \text{beq } \$s0, \$s1, \text{offset}
  \]

<table>
<thead>
<tr>
<th>4</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- 4 = opcode
- rs, rt = source registers
- offset = immediate or constant value
- if rs == rt
  - then PC ← PC\(^{+}\) + sign-extend(offset) \* 4
- Variations: beq, bne, blez, bgtz

\(^{+}\)This is the incremented PC
This is an example of tradeoff in the instruction set

- The same functionality requires more instructions in LC-3
- But, the control logic requires more complexity in MIPS
Lecture Summary

- The von Neumann model
  - LC-3: An example of von Neumann machine

- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions

- Instruction formats

- Addressing modes