The von Neumann Model & Instruction Set Architectures
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
What Will We Learn Today?

- Basic elements of a computer & the von Neumann model
  - LC-3: An example von Neumann machine

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

- Instruction formats

- Addressing modes
Readings

This week
- Von Neumann Model, ISA, LC-3, and MIPS
  - P&P, Chapters 4, 5 (we will follow these today & tomorrow)
  - H&H, Chapter 6 (until 6.5)
  - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
  - H&H, Appendix B (MIPS instructions)
- Programming
  - P&P, Chapter 6 (we will follow this tomorrow)
- Recommended: H&H Chapter 5, especially 5.1, 5.2, 5.4, 5.5

Next week
- Introduction to microarchitecture and single-cycle microarchitecture
  - H&H, Chapter 7.1-7.3
  - P&P, Appendices A and C
- Multi-cycle microarchitecture
  - H&H, Chapter 7.4
  - P&P, Appendices A and C
Quick Review of the von Neumann Model
Recall: The von Neumann Model

- **INPUT**: Keyboard, Mouse, Disk…
- **OUTPUT**: Monitor, Printer, Disk…
- **PROCESSING UNIT**: ALU, TEMP
- **MEMORY**: Mem Addr Reg, Mem Data Reg
- **CONTROL UNIT**: PC or IP, Inst Register
The Von Neumann Model

Stored program

Sequential instruction processing
Recall: von Neumann Model: Two Key Properties

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

  - **Stored program**
    - Instructions stored in a linear memory array
    - **Memory is unified** between instructions and data
      - The interpretation of a stored value depends on the control signals

  - **Sequential instruction processing**
    - One instruction processed (fetched, executed, completed) at a time
    - **Program counter** (instruction pointer) identifies the current instruction
    - **Program counter is advanced sequentially** except for control transfer instructions
Programmer Visible (Architectural) State

Memory
array of storage locations
indexed by an address

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

Program Counter
memory address
of the current (or next) instruction

Instructions (and programs) specify how to transform
the values of programmer visible state
Recall: LC-3: A von Neumann Machine

Figure 4.3 The LC-3 as an example of the von Neumann model
Recall: The Instruction

- An instruction is 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 will study **LC-3 instructions** and **MIPS instructions**
  - Principles are similar in all ISAs (x86, ARM, RISC-V, ...)

Recall: Instruction Types

- There are three main types of instructions
  - Operate instructions
    - Execute operations in the ALU
  - Data movement instructions
    - Read from or write to memory
  - Control flow instructions
    - Change the sequence of execution

- Let us start with some example instructions
Recall: Load Word in LC-3 and MIPS

- **LC-3 assembly**

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

  LC-3 assembly  
  ```
  LDR R3, R0, #2
  ```

  R3 ← Memory[R0 + 2]

- **MIPS assembly (assuming word-addressable)**

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

  MIPS assembly  
  ```
  lw $s3, 2($s0)
  ```

  $s3 ← Memory[$s0 + 2]

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

- **MIPS assembly**

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

  MIPS assembly:

  \[
  \text{lw} \quad \$s3, \quad 8(\$s0) \\
  \$s3 \leftarrow \text{Memory}[$s0 + 8]
  \]

- Byte address is calculated as: \text{word\_address} \times \text{bytes\_per\_word}
  - 4 bytes/word in MIPS
  - If LC-3 were byte-addressable (i.e., LC-3b), 2 bytes/word
Recall: Instruction Format With Immediate

**LC-3**

**LC-3 assembly**

LDR  R3, R0, #2

**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>3</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**MIPS**

**MIPS assembly**

lw  $s3, 8($s0)

**Field Values**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>16</td>
<td>19</td>
<td>8</td>
</tr>
</tbody>
</table>
Instruction (Processing) Cycle
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 (or, instruction processing 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 require 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 (IR).

This phase is common to every instruction type.

Complete description:

1. Step 1: Load the MAR with the contents of the PC, and simultaneously increment the PC.

2. Step 2: Interrogate memory. This results in the instruction being placed in the MDR by memory.

3. 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

- Also generates the set of control signals to process the identified instruction in later phases of the instruction cycle

Recall the decoder (from our Combinational Logic lectures)

- 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

Also generates the set of control signals to process the instruction
Recall: Decoder

- “Input pattern detector”
- $n$ inputs and $2^n$ outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$Y_3$</th>
<th>$Y_2$</th>
<th>$Y_1$</th>
<th>$Y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Recall: Decoder (II)

- The decoder is useful in determining how to interpret a bit pattern
  - It could be the address of a location in memory, that the processor intends to read from
  - It could be an instruction in the program and the processor needs to decide what action to take (based on instruction opcode)
To Come: Full State Machine for LC-3b

Decode State

Figure C.2: A state machine for the LC-3b

EVALUATE ADDRESS

- 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

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

**Figure 4.3** The LC-3 as an example of the von Neumann model
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 some microprocessors, operand fetch from register file 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
  - In XOR, it performs bitwise XOR in the ALU
  - ...

EXECUTE in LC-3

ADD adds SR1 and SR2
STORE RESULT

- The STORE RESULT phase writes the result to the designated destination

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

ADD loads ALU
Result into DR
STORE RESULT in LC-3

LDR loads MDR into DR

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

```
JMP R2
```

<table>
<thead>
<tr>
<th>1100</th>
<th>000</th>
<th>BaseR</th>
<th>000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td></td>
<td></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

\[ jr \ $s0 \]

[^31:28]: This is the incremented PC

jr uses register addressing mode

j uses pseudo-direct addressing mode
PC UPDATE in LC-3

JMP loads SR1 into PC

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 Processor
The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
LC-3 and MIPS
Instruction Set Architectures
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
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>SR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0 00</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**
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: 8 bits)
      - Word- or Byte-addressable
  - The register set:
    - 8 registers (R0 to R7) in LC-3
    - 32 registers in MIPS
  - The instruction set:
    - Opcodes
    - Data types
    - Addressing modes
    - Length and format of instructions

<table>
<thead>
<tr>
<th>Problem</th>
<th>Algorithm</th>
<th>ISA</th>
<th>Microarchitecture</th>
<th>Circuits</th>
<th>Electrons</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

47
Instructions (Opcodes)
Opcodes

- A large or small **set of opcodes** could be defined
  - E.g, HP Precision Architecture: an instruction for \(A \times B + C\)
  - E.g, x86 ISA: multimedia extensions (MMX), later SSE and AVX
  - E.g, VAX ISA: opcode to save all information of one program prior to switching to another program

- **Tradeoffs** are involved. Examples:
  - Hardware complexity vs. software complexity
  - Latency of simple vs. complex instructions

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

Figure 5.3 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>Binary</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD+</td>
<td>0001</td>
<td>DR SR1 A</td>
<td>op.spec</td>
</tr>
<tr>
<td>AND+</td>
<td>0101</td>
<td>DR SR1 A</td>
<td>op.spec</td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n z p</td>
<td>PCoffset9</td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>000</td>
<td>BaseR 000000</td>
</tr>
<tr>
<td>JSR(R)</td>
<td>0100</td>
<td>A</td>
<td>operand specifier</td>
</tr>
<tr>
<td>LDB+</td>
<td>0010</td>
<td>DR BaseR</td>
<td>offset6</td>
</tr>
<tr>
<td>LDW+</td>
<td>0110</td>
<td>DR BaseR</td>
<td>offset6</td>
</tr>
<tr>
<td>LEA+</td>
<td>1110</td>
<td>DR</td>
<td>PCoffset9</td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td>00000000000000</td>
<td></td>
</tr>
<tr>
<td>SHF+</td>
<td>1101</td>
<td>DR SR A D</td>
<td>amount4</td>
</tr>
<tr>
<td>STB</td>
<td>0011</td>
<td>SR BaseR</td>
<td>offset6</td>
</tr>
<tr>
<td>STW</td>
<td>0111</td>
<td>SR BaseR</td>
<td>offset6</td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>0000</td>
<td>trapvec18</td>
</tr>
<tr>
<td>XOR+</td>
<td>1001</td>
<td>DR SR1 A</td>
<td>op.spec</td>
</tr>
</tbody>
</table>

not used 1010
not used 1011
# MIPS Instruction Types

## R-type

<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-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>6-bit</td>
</tr>
</tbody>
</table>

## I-type

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

## J-type

<table>
<thead>
<tr>
<th>opcode</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>26-bit</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] \ll \text{shamt})</td>
</tr>
<tr>
<td>000010 (2)</td>
<td>srl rd, rt, shamt</td>
<td>shift right logical</td>
<td>([rd] = [rt] \gg \text{shamt})</td>
</tr>
<tr>
<td>000011 (3)</td>
<td>sra rd, rt, shamt</td>
<td>shift right arithmetic</td>
<td>([rd] = [rt] \ggg \text{shamt})</td>
</tr>
</tbody>
</table>

(continued)
### 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>subtract</td>
<td>[rd] = [rs] - [rt]</td>
</tr>
<tr>
<td>100011 (35)</td>
<td>subu rd, rs, rt</td>
<td>subtract 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]</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] = ~([rs]</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>

More complete list of instructions are in H&H Appendix B
Data Types
Data Types

- An ISA supports one or several data types

- LC-3 only supports 2’s complement integers
  - Negative of a 2’s complement binary value $X = \text{NOT}(X) + 1$

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

- Tradeoffs are involved. Examples:
  - Hardware complexity vs. software complexity
  - Latency of operations on supported vs. unsupported data types

H&H Chapter 1.4.6
Why Have Different Data Types in ISA?

- An example of programmer vs. microarchitect tradeoff

- Advantage of more data types:
  - Enables better mapping of high-level programming constructs to hardware
  - Hardware can directly operate on data types present in programming languages → small number of instructions and code size
    - Matrix operations vs. individual multiply/add/load/store instructions
    - Graph operations vs. individual load/store/add/... instructions

- Disadvantage:
  - More work for the microarchitect
    - who needs to implement the data types and instructions that operate on data types
Data Types and Instruction Complexity

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

- Concept of semantic gap
  - how close instructions & data types are to high-level language

- Complex instructions + data types $\rightarrow$ small semantic gap
  - E.g., insert into a doubly linked list, multiply two matrices
  - VAX ISA: doubly-linked list, multi-dimensional arrays

- Simple instructions + data types $\rightarrow$ large semantic gap
  - E.g., primitive operations: load, store, multiply, add, nor
  - Early RISC machines: Only integer data type, simple operations
Semantic Gap

- How close instructions & data types are to high-level language (HLL)

```
ISA with Complex Inst & Data Types
  HLL

Small Semantic Gap

ISA with Simple Inst & Data Types
  HLL

Large Semantic Gap
```

HW Control Signals

HW Control Signals
Complex vs. Simple Instructions + Data Types

- **Complex instruction**: An instruction *does a lot of work*, e.g. many operations
  - Insert in a doubly linked list
  - Compute FFT
  - String copy
  - Matrix multiply
  - ...

- **Simple instruction**: An instruction *does little work* -- it is a primitive using which complex operations can be built
  - Add
  - XOR
  - Multiply
  - ...
Complex vs. Simple Instructions + Data Types

- **Advantages of Complex Instructions + Data Types**
  - **Denser encoding** → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  - **Simpler compiler**: no need to optimize small instructions as much

- **Disadvantages of Complex Instructions + Data Types**
  - **Larger chunks of work** → compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
  - **More complex hardware** → translation from a high level to control signals and optimization needs to be done by hardware
Aside: An Example: **Binary Coded Decimal**

- Each decimal digit is encoded with a fixed number of bits

![Binary Coded Decimal Diagram](Binary_Clock_Diagram.png)


Aside: An Example: **Binary Coded Decimal**

- Each decimal digit is encoded with a fixed number of bits
Addressing Modes
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 memory addressing modes**
  - PC-relative
  - Indirect
  - Base+offset

MIPS has **pseudo-direct addressing** (for j and jal), additionally, but does **not** have indirect addressing.
Why Have Different Addressing Modes?

- Another example of programmer vs. microarchitect tradeoff

- Advantage of more addressing modes:
  - Enables better mapping of high-level programming constructs to hardware
    - some accesses are better expressed with a different mode → reduced number of instructions and code size
      - Array indexing (one or multi-dimensional)
      - Pointer-based accesses (indirection)
      - Matrix element indexing; sparse matrix element indexing

- Disadvantages:
  - More work for the microarchitect
  - More options for the compiler to decide what to use
Semantic Gap Applies to Addressing Modes

- How close instructions & data types & addressing modes are to high-level language (HLL)

![Diagram showing small and large semantic gaps between HLL, ISA, and HW control signals with complex and simple instructions and data types and addressing modes.](image-url)
Many Tradeoffs in ISA Design...

- Execution model – sequencing model and processing style
- Instruction length
- Instruction format
- Instruction types
- Instruction complexity vs. simplicity
- Data types
- Number of registers
- Addressing mode types
- Memory organization (address space, addressability, endianness, ...)
- Memory access restrictions and permissions
- Support for multiple instructions to execute in parallel?
- ...

...
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 (i.e., with one immediate operand) 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>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1111111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>011</td>
<td>001</td>
</tr>
<tr>
<td>1111111</td>
<td></td>
<td></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

- We will use Subtraction as an example
  - How is it implemented in LC-3 and MIPS?
Recall: LC-3 Operate Instruction Format

- LC-3 Operate Instruction Format (Register OP Register)

  - **OP = opcode** (what the instruction does)
    - E.g., ADD = 0001
      - Semantics: $DR \leftarrow SR1 + SR2$
    - E.g., AND = 0101
      - Semantics: $DR \leftarrow SR1 \text{ AND } SR2$

  - **SR1, SR2 = source registers**

  - **DR = destination register**
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>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2</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>
<tr>
<td>11110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>11</td>
<td>9</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

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0001 001 100 1 11110
ADD R1 R4 R5

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 ing 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 immediate operands. Which integers are OK (i.e., which integers can be used as immediate operands)?
ADD with one Literal in LC-3 Data Path

Figure 5.18

The data path of the LC-3

ADD with one Literal in LC-3 Data Path

Sign extension (Operand)

Select Immediate or Register (as the 2\textsuperscript{nd} input to instruction)
Instructions with one Literal in MIPS

- **I-type MIPS Instructions**
  - 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

```
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>8</td>
<td>17</td>
<td>16</td>
<td>5</td>
</tr>
</tbody>
</table>

```
rt ← rs + sign-extend(imm)
```

### Machine Code

```
001000 10001 10010 0000 0000 0000 0101
```

0x22300005
Subtraction in MIPS vs. LC-3

- **MIPS assembly**

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

  MIPS assembly
  ```
  add $t0, $s0, $s1
  sub $s3, $t0, $s2
  ```

- **LC-3 assembly**

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

  LC-3 assembly
  ```
  ADD R2, R0, R1
  NOT R4, R3
  ADD R5, R4, #1
  ADD R6, R2, R5
  ```
  2’s complement of R3

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

- **MIPS assembly**
  
  **High-level code**
  \[ a = b - 3; \]

  **MIPS assembly**
  
  \[ \text{subi } $s1, $s0, 3 \]

  - Is **subi** necessary in MIPS?

  **MIPS assembly**
  
  \[ \text{addi } $s1, $s0, -3 \]

- **LC-3**
  
  **High-level code**
  \[ a = b - 3; \]

  **LC-3 assembly**
  
  \[ \text{ADD } R1, R0, #-3 \]
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

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

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

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

---

The memory address is only +255 to -256 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)

<table>
<thead>
<tr>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>DR/SR</td>
<td>PCoffset9</td>
</tr>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>9 bits</td>
</tr>
</tbody>
</table>

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

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

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

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

\( ^\dagger \text{This is the incremented PC} \)
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>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><strong>OP</strong></td>
<td><strong>DR/SR</strong></td>
<td><strong>BaseR</strong></td>
<td><strong>offset6</strong></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>3 bits</td>
<td>6 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>
</tr>
</tbody>
</table>

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

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

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

- **STR**: \( \text{Memory}[\text{BaseR} + \text{sign-extend}()\text{offset6}] \leftarrow SR \)
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>0 1 1 0</td>
<td>0 0 1</td>
<td>0 1 0</td>
<td>0 1 1 1 0 1</td>
</tr>
</tbody>
</table>

Again, the address of the operand can be anywhere in the memory.
Address Calculation in LC-3 Data Path

Figure 5.18 The data path of the LC-3
Base+Offset Addressing Mode in MIPS

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

High-level code

```
```

MIPS assembly

```
sw $s3, 8($s0)
```

Memory[$s0 + 8] ← $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

### High-level code

\[
a = A[0]; \\
c = a + b - 5; \\
B[0] = c;
\]

### MIPS registers

- \( A = \$s0 \)
- \( b = \$s2 \)
- \( B = \$s1 \)

### LC-3 registers

- \( A = R0 \)
- \( b = R2 \)
- \( B = R1 \)

### MIPS assembly

```
lw $t0, 0($s0)  
add $t1, $t0, $s2  
addi $t2, $t1, -5  
sw $t2, 0($s1)
```

### LC-3 assembly

```
LDR R5, R0, #0  
ADD R6, R5, R2  
ADD R7, R6, #–5  
STR R7, R1, #0
```
Immediate Addressing Mode (in LC-3)

- LEA (Load Effective Address)

- **OP = 1110**

- **DR = destination register**

- **LEA:** \( DR \leftarrow PC^\dagger + \text{sign-extend}(PC_{\text{offset9}}) \)

What is the difference from PC-Relative addressing mode?

Answer: Instructions with PC-Relative mode load from memory, but LEA does not \( \rightarrow \) Hence the name *Load Effective Address*

\(^\dagger\) 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

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>101</td>
<td>1111111101</td>
</tr>
</tbody>
</table>

-  Field Values
  - OP: E
  - DR: 5
  - PCoffset9: 0x1FD

- Machine Code
  - OP: 1110
  - DR: 101
  - PCoffset9: 1111111101

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 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?

P&P, Chapter 5.3.5
## LC- ISA Instruction Encodings

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Function</th>
<th>Flags</th>
<th>Condition Codes</th>
<th>Condition Codes</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD*</td>
<td>0001</td>
<td>DR</td>
<td>SR1</td>
<td>00</td>
<td>SR2</td>
<td>imm5</td>
</tr>
<tr>
<td>ADD*</td>
<td>0001</td>
<td>DR</td>
<td>SR1</td>
<td>1</td>
<td>imm5</td>
<td></td>
</tr>
<tr>
<td>AND*</td>
<td>0101</td>
<td>DR</td>
<td>SR1</td>
<td>00</td>
<td>SR2</td>
<td></td>
</tr>
<tr>
<td>AND*</td>
<td>0101</td>
<td>DR</td>
<td>SR1</td>
<td>1</td>
<td>imm5</td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n z p</td>
<td></td>
<td>PCoffset9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>000</td>
<td>BaseR</td>
<td>000000000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td>1</td>
<td></td>
<td>PCoffset11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>0 00</td>
<td>BaseR</td>
<td>000000000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD*</td>
<td>0010</td>
<td>DR</td>
<td></td>
<td>PCoffset9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDI*</td>
<td>1010</td>
<td>DR</td>
<td></td>
<td>PCoffset9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR*</td>
<td>0110</td>
<td>DR</td>
<td>BaseR</td>
<td>offset6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEA*</td>
<td>1110</td>
<td>DR</td>
<td></td>
<td>PCoffset9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT*</td>
<td>1001</td>
<td>DR</td>
<td>SR</td>
<td>1111111111111111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>000</td>
<td>111</td>
<td>0000000000000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td>000</td>
<td></td>
<td>0000000000000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>0011</td>
<td>SR</td>
<td></td>
<td>PCoffset9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STI</td>
<td>1011</td>
<td>SR</td>
<td></td>
<td>PCoffset9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR</td>
<td>0111</td>
<td>SR</td>
<td>BaseR</td>
<td>offset6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>0000</td>
<td>trapve</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Patt & Patel, Back cover inside

Figure 5.3 Formats of the entire LC-3 instruction set. NOTE: * indicates instructions that modify condition codes.
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 unconditional 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
  - We have already seen these
Conditional Control Flow
(Conditional Branching)
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 \( Z \) are cleared

- x86 and SPARC 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</td>
</tr>
</tbody>
</table>

- **n, z, p = which condition code is tested (N, Z, and/or P)**
  - n, z, p: instruction bits to identify the condition codes to be tested
  - N, Z, P: values of the corresponding condition codes

- **PCoffset9 = immediate or constant value**

- **if ((n AND N) OR (p AND P) OR (z AND Z))**
  - then PC ← PC\(^\dagger\) + sign-extend(PCoffset9)

- **Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp**

\(^\dagger\) This is the incremented PC
Conditional Branches in LC-3

**BRz**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Program Counter</th>
<th>Instruction Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRz 0x0D9</td>
<td>PC 0100 0000 0010 1000</td>
<td>IR 0000 01 0 011011001</td>
</tr>
</tbody>
</table>

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

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

---

*\( n, z, p \) are the instruction bits to identify the condition codes to be tested
Conditional Branches in MIPS

- **beq (Branch if Equal)**

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

  \[
  \begin{array}{cccc}
  \text{4} & \text{rs} & \text{rt} & \text{offset} \\
  6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 16 \text{ bits}
  \end{array}
  \]

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

\(^{†}\) This is the incremented PC
Branch If Equal in MIPS and LC-3

- 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**
What We Learned

- **Basic elements of a computer** & the von Neumann model
  - LC-3: An example von Neumann machine

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

- **Instruction formats**

- **Addressing modes**
There Is A Lot More to Cover on ISAs

https://www.youtube.com/onurmutlulectures
Many Different ISAs Over Decades

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM, RISC-V, ...

- What are the fundamental differences?
  - E.g., how instructions are specified and what they do
  - E.g., how complex are instructions, data types, addr. modes
Complex vs. Simple Instructions + Data Types

- **Complex instruction**: An instruction does a lot of work, e.g., many operations
  - Insert in a doubly linked list
  - Compute FFT
  - String copy
  - Matrix multiply
  - ...

- **Simple instruction**: An instruction does little work -- it is a primitive using which complex operations can be built
  - Add
  - XOR
  - Multiply
  - ...
Complex vs. Simple Instructions + Data Types

- **Advantages of Complex Instructions + Data Types**
  + Denser encoding $\rightarrow$ smaller code size $\rightarrow$ better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  + Simpler compiler: no need to optimize small instructions as much

- **Disadvantages of Complex Instructions + Data Types**
  - Larger chunks of work $\rightarrow$ compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
  - More complex hardware $\rightarrow$ translation from a high level to control signals and optimization needs to be done by hardware
Semantic Gap

- How close instructions & data types & addressing modes are to high-level language (HLL)

Easier mapping of HLL to ISA
Less work for software designer
More work for hardware designer
Optimization burden on HW

Harder mapping of HLL to ISA
More work for software designer
Less work for hardware designer
Optimization burden on SW
How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different “implementation” ISA
In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.[2][8]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.[9]

Rosetta 2's performance has been praised greatly.[10][11] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.[12]

Although Rosetta 2 works for most software, some software doesn't work at all[13] or is reported to be "sluggish".[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.[15]
An Example: Rosetta 2 Binary Translator

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Another Example: Intel and AMD Processors

HLL

Small Semantic Gap

Hardware Translator

X86-64
ISA with
Complex Inst
& Data Types
& Addressing Modes

Implementation ISA with
Simple Inst
& Data Types
& Addressing Modes

Secret
Micro-operations
HW
Control
Signals
Another Example: Intel and AMD Processors

Source: https://twitter.com/Locuza_/status/1454152714930331652

Intel Alder Lake, 2021
Another Example: Intel and AMD Processors

AMD Ryzen 5000, 2020

Core Count:
8 cores/16 threads

L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA's decision to forgo a traditional out-of-order design for Denver means that much of Denver's potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

Accomplishing this task is NVIDIA's dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver's native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO's task to find instruction level parallelism within a thread to fill Denver's many execution units, and to reorder instructions around potential stalls, something that is no simple task.
Transmeta: x86 to VLIW Translation

Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.


https://www.wikiwand.com/en/Transmeta_Efficeon
ISA-level Tradeoffs: Number of Registers

- **Affects:**
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- **Large number of registers:**
  - + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  - -- Larger instruction size
  - -- Larger register file size
There Is A Lot More to Cover on ISAs

https://www.youtube.com/onurmutlulectures
There Is A Lot More to Cover on ISAs

ISA-level Tradeoffs: Number of Registers

Affects:
- Number of bits used for encoding register address
- Number of values kept in fast storage (register file)
- (uarch) Size, access time, power consumption of register file

Large number of registers:
- Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  -- Larger instruction size
  -- Larger register file size

https://www.youtube.com/onurmutlulectures
Detailed Lectures on ISAs & ISA Tradeoffs

- Computer Architecture, Spring 2015, Lecture 3
  - ISA Tradeoffs (CMU, Spring 2015)
    - https://www.youtube.com/watch?v=QKdiZSfwg-g&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybT DtKq&index=3

- Computer Architecture, Spring 2015, Lecture 4
  - ISA Tradeoffs & MIPS ISA (CMU, Spring 2015)
    - https://www.youtube.com/watch?v=RBgeCCW5Hjs&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=4

- Computer Architecture, Spring 2015, Lecture 2
  - Fundamental Concepts and ISA (CMU, Spring 2015)
    - https://www.youtube.com/watch?v=NpC39uS4K4o&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWyb T DtKq&index=2

https://www.youtube.com/onurmutlulectures
ISA Design and Tradeoffs: More Critical Thinking
The Von Neumann Model/Architecture

Stored program

Sequential instruction processing
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

- **Stored program**
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - The interpretation of a stored value depends on the control signals
      
        When is a value interpreted as an instruction?

- **Sequential instruction processing**
Whether a value fetched from memory is interpreted as an instruction depends on 
when that value is fetched in the instruction processing cycle.
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

  - **Stored program**
    - Instructions stored in a linear memory array
    - **Memory is unified** between instructions and data
      - The interpretation of a stored value depends on the control signals

  - **Sequential instruction processing**
    - One instruction processed (fetched, executed, completed) at a time
    - **Program counter (instruction pointer)** identifies the current instruction
    - **Program counter is advanced sequentially** except for control transfer instructions

When is a value interpreted as an instruction?
The von Neumann Model/Architecture

- **Recommended reading**
  - Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.

- **Important reading**
  - Patt and Patel book, Chapter 4, “The von Neumann Model”

- **Stored program**

- **Sequential instruction processing**
The Von Neumann Model (of a Computer)
Q: Is this the only way that a computer can process computer programs?

A: No.

Qualified Answer: No. But, it has been the dominant way

- i.e., the dominant paradigm for computing
- for N decades

Let’s examine a completely different model for processing computer programs.
The Dataflow Execution Model of a Computer
The Dataflow Model (of a Computer)

- **Von Neumann model:** An instruction is fetched and executed in *control flow order*
  - As specified by the *program counter (instruction pointer)*
  - Sequential unless explicit control flow instruction

- **Dataflow model:** An instruction is fetched and executed in *data flow order*
  - i.e., when its operands are ready
  - i.e., there is *no program counter (instruction pointer)*
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel
Von Neumann vs. Dataflow

Consider a Von Neumann program

- What is the significance of the program order?
- What is the significance of the storage locations?

\[
\begin{align*}
v &= a + b; \\
w &= b \times 2; \\
x &= v - w \\
y &= v + w \\
z &= x \times y
\end{align*}
\]

Sequential

Dataflow

a, b are the only inputs
z is the only output

Which model is more natural to you as a programmer?
More on Dataflow

- In a dataflow machine, a program consists of dataflow nodes
  - A dataflow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens

- Dataflow node and its ISA representation
Example Dataflow Nodes

*Conditional

*Relational

*Barrier Synch
What is the value of OUT?

N is a non-negative integer
ISA-level Tradeoff: Program Counter

- Do we want a Program Counter (PC or IP) in the ISA?
  - **Yes:** Control-driven, sequential execution
    - An instruction is executed when the PC points to it
    - PC automatically changes sequentially (except for control flow instructions) → **sequential**
  - **No:** Data-driven, parallel execution
    - An instruction is executed when all its operand values are available → **dataflow**

- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?
ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level.

  **ISA:** Specifies how the programmer sees the instructions to be executed.
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a dataflow execution order.

  **Microarchitecture:** How the underlying implementation actually executes instructions.
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software.
  - Programmer should see the order specified by the ISA.
Let’s Get Back to the von Neumann Model

- But, if you want to learn more about dataflow...


- A later lecture

- If you are really impatient:
  - http://www.youtube.com/watch?v=D2uue7izU2c
Lecture Video on Dataflow Architectures

http://www.youtube.com/watch?v=D2uue7izU2c
All major *instruction set architectures* today use this model
- x86, ARM, MIPS, SPARC, Alpha, POWER, RISC-V, ...

Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
- Pipelined instruction execution: *Intel 80486 uarch*
- Multiple instructions at a time: *Intel Pentium uarch*
- Out-of-order execution: *Intel Pentium Pro uarch*
- Separate instruction and data caches

But, what happens underneath that is *not consistent* with the von Neumann model is *not exposed* to software
- Difference between ISA and microarchitecture
What is Computer Architecture?

- **ISA+implementation definition:** The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.

- **Traditional (ISA-only) definition:** “The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior as distinct from the organization of the dataflow and controls, the logic design, and the physical implementation.”

  *Gene Amdahl*, IBM Journal of R&D, April 1964
ISA vs. Microarchitecture

- **ISA**
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs

- **Microarchitecture**
  - Specific implementation of an ISA
  - Not visible to the software

- **Microprocessor**
  - ISA, \texttt{uarch}, circuits
  - “Architecture” = ISA + microarchitecture
Microarchitecture

- A specific **implementation** of the ISA

- How do we implement the ISA?
  - We will discuss this for many lectures

- There can be many implementations of the same ISA
  - **MIPS** R2000, R3000, R4000, R6000, R8000, R10000, ...
  - **x86**: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cove, Sapphire Rapids, ..., AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, ...
  - **POWER** 4, 5, 6, 7, 8, 9, 10 (IBM), ..., **PowerPC** 604, 605, 620, ...
  - **ARM** Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, ...
  - **Alpha** 21064, 21164, 21264, 21364, ...
  - **RISC-V** ...
  - ...

146
ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implement “acceleration”

- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture (see H&H Chapter 5.2.1)
  - x86 ISA has many implementations:
    - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cover, Sapphire Rapids, ..., AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, ...

- Microarchitecture usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha, RISC-V) but many uarchs
  - Why?
ISA: What Does ItSpecify?

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes

- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management

- Call, Interrupt/Exception Handling

- Access Control, Priority/Privilege

- I/O: memory-mapped vs. instructions

- Task/thread Management

- Power & Thermal Management

- Multithreading & Multiprocessor support

...
## ISA Manuals: Some Good Bedtime Reading

### Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer’s Manuals

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4</td>
<td>This document contains the following:</td>
</tr>
<tr>
<td></td>
<td><strong>Volume 1</strong>: Describes the architecture and programming environment of processors supporting IA-32 and Intel® 64 architectures.</td>
</tr>
<tr>
<td></td>
<td><strong>Volume 2</strong>: Includes the full instruction set reference, A-Z. Describes the format of the instruction and provides reference pages for instructions.</td>
</tr>
<tr>
<td></td>
<td><strong>Volume 3</strong>: Includes the full system programming guide, parts 1, 2, 3, and 4. Describes the operating-system support environment of Intel® 64 and IA-32 architectures, including: memory management, protection, task management, interrupt and exception handling, multi-processor support, thermal and power management features, debugging, performance monitoring, system management mode, virtual machine extensions (VMX) instructions, Intel® Virtualization Technology (Intel® VT), and Intel® Software Guard Extensions (Intel® SGX). NOTE: Performance monitoring events can be found here: <a href="https://perfmon-events.intel.com/">https://perfmon-events.intel.com/</a></td>
</tr>
<tr>
<td></td>
<td><strong>Volume 4</strong>: Describes the model-specific registers of processors supporting IA-32 and Intel® 64 architectures.</td>
</tr>
<tr>
<td>Intel® 64 and IA-32 Architectures Software Developer’s Manual Documentation Changes</td>
<td>Describes bug fixes made to the Intel® 64 and IA-32 architectures software developer's manual between versions.</td>
</tr>
<tr>
<td></td>
<td>NOTE: This change document applies to all Intel® 64 and IA-32 architectures software developer’s manual sets (combined volume set, 4 volume set, and 10 volume set).</td>
</tr>
</tbody>
</table>

ISA Manuals: Some Good Bedtime Reading

Specifications

The RISC-V instruction set architecture (ISA) and related specifications are developed, ratified and maintained by RISC-V International contributing members within the RISC-V International Technical Working Groups. Work on the specification is performed on GitHub, and the GitHub issue mechanism can be used to provide input into the specification.

If you would like more information on becoming a member, please see the membership page.

ISA Specification
The specifications shown below represent the current, ratified releases. Work is being done on GitHub.

- Volume 1, Unprivileged Spec v. 20191213 [PDF]
- Volume 2, Privileged Spec v. 20211203 [PDF]
- Recently ratified, but not yet integrated, extension specifications

Debug Specification
This is the currently ratified specification:

- External Debug Support v. 0.13.2 [PDF] [GitHub]

This is the current stable draft:

- External Debug Support v. 1.0.0-STABLE [PDF]

Trace Specification
The processor trace specification was approved on March 20, 2020.

- Trace Specification v. 1.0 [PDF] [GitHub]

Compatibility Test Framework
The RISC-V Architectural Compatibility Test Framework Version 2 is now available. This framework compares arbitrary models against a reference signature, and currently covers RV[32|64]MC unprivileged specifications only. Tests for the not-yet-ratified Crypto Scalar extension and RV32EMC extensions are also available.

Work on Version 3.0 framework (RISCOF) is