# Instruction Set Architectures

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

November 21, 2023

# Outline

# Acknowledgement

The content of most slides come from the authors of the textbook:

Null, Linda, & Lobur, Julia (2018). The essentials of computer organization and architecture (5th ed.). Jones & Bartlett Learning.

# Table of Contents

# Lesson Objectives

Students are expected to be able to

1. Understand the factors involved in instruction set architecture design.
2. Gain familiarity with memory addressing modes.
3. Understand the concepts of instruction-level pipelining and its effect upon execution performance.

# Table of Contents

# Instruction Formats

Instruction sets are differentiated by the following:

- Number of bits per instruction.
- Stack-based or register-based.
- Number of explicit operands per instruction.
- Operand location.
- Types of operations.
- Type and size of operands.

# Architecture Consideration

Instruction set architectures are measured according to:

- Main memory space occupied by a program.
- Instruction complexity.
- Instruction length (in bits).
- Total number of instructions in the instruction set.

# Design Consideration

In designing an instruction set, consideration is given to:

- Instruction length. Whether short, long, or variable.
- Number of operands. Number of addressable registers.
- Memory organization. Whether byte- or word addressable.
- Addressing modes. Choose any or all: direct, indirect or indexed, and
- ...

# Byte Ordering

Endianness, or byte ordering, is another major architectural consideration.

- If we have a two-byte integer, the integer may be stored in memory so that
  - the least significant byte is followed by the most significant byte, or
  - vice versa.
- In little endian machines, the least significant byte is followed by the most significant byte. (the least significant first)
- Big endian machines store the most significant byte first (at the lower address). (the most significant first)

# Byte Ordering: Example 1

Suppose we have the hexadecimal number 0x12345678. The big endian and small endian arrangements of the bytes on a byte-addressable computer are shown below.

| Address (in Binary) | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| Little Endian (in Hex) | 78 | 56 | 34 | 12 |
| Big Endian (in Hex) | 12 | 34 | 56 | 78 |

# Byte Ordering: Example 2

A larger example: A byte-addressable computer uses 32-bit integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as here.

# Byte Ordering: Example 2

| Address | Big Endian | Little Endian |
|---|---|---|
| 0x200 | 0xAB | 0x34 |
| 0x201 | 0xCD | 0x12 |
| 0x202 | 0x12 | 0xCD |
| 0x203 | 0x34 | 0xAB |
| 0x204 | 0x00 | 0x21 |
| 0x205 | 0xFE | 0x43 |
| 0x206 | 0x43 | 0xFE |
| 0x207 | 0x21 | 0x00 |
| 0x208 | 0x00 | 0x10 |
| 0x209 | 0x00 | 0x00 |
| 0x20A | 0x00 | 0x00 |
| 0x20B | 0x10 | 0x00 |

# Big Endian vs Little Endian

Big endian:

- Is more natural.
- The sign of the number can be determined by looking at the byte at address offset 0.
- Strings and integers are stored in the same order.

Little endian:

- Makes it easier to place values on non-word boundaries.
- Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

# How the CPU will Store Data?

Design consideration: how the CPU will store data? We have three choices:

- A stack architecture
- An accumulator architecture
- A general purpose register architecture

In choosing one over the other, the tradeoffs are

- simplicity (and cost) of hardware design with
- execution speed and ease of use.

# Stack Architecture

In a stack architecture, instructions and operands are implicitly taken from the stack.

- A stack cannot be accessed randomly.

# Accumulator Architecture

In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.

- One operand is in memory, creating lots of bus traffic.

# GPR Architecture

In a general purpose register (GPR) architecture, registers can be used instead of memory.

- Faster than accumulator architecture.
- Efficient implementation for compilers.
- Results in longer instructions.

# GPR Architecture: Types

Most systems today are GPR systems. There are three types:

- Memory-memory where two or three operands may be in memory.
- Register-memory where at least one operand must be in a register.
- Load-store where no operands may be in memory.

The number of operands and the number of available registers has a direct effect on instruction length.

# Stack Architecture: Instruction Format

Stack machines use one- and zero-operand instructions.

- LOAD and STORE instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- PUSH and POP operations involve only the stack's top element.
- Binary instructions (e.g., ADD, MULT) use the top two items on the stack.

# Stack Architecture: Arithmetic and Postfix Notation

We are accustomed to writing expressions using infix notation, such as: Z = X + Y.

- Stack architectures require us to think about arithmetic expressions differently.
- Stack arithmetic requires that we use postfix notation: Z = XY+.
- This is also called reverse Polish notation.

# Postfix Notation

The principal advantage of postfix notation is that parentheses are not used.

- For example, the infix expression,

$$Z = (X + Y) * (W - U)$$

becomes:

$$Z = XY + WU - *$$

in postfix notation.

# Infix to Postfix Notation: Example

Convert the infix expression $(2 + 3) - 6/3$ to postfix:

$$23 + 63/-$$

# Evaluating Postfix Expression: Example: 1

Use a stack to evaluate the postfix expression $23 + 63/-$:

1. Scan the expression from left to right, push operants onto the stack, until an operator is found

2 3 + 6 3 / -

↑

Stack:

3

2

# Evaluating Postfix Expression: Example: 2

Use a stack to evaluate the postfix expression $23 + 63/-$:

1. Pop the operants and carry out the operation indicated by the operator, and push the result back onto the stack

2 3 + 6 3 / -

↑

Stack:

5

# Evaluating Postfix Expression: Example: 3

Use a stack to evaluate the postfix expression $23 + 63/-$:

1. Push operants until another operator is found

2 3 + 6 3 / -
↑

Stack:
3
6
5

# Evaluating Postfix Expression: Example: 4

Use a stack to evaluate the postfix expression $23 + 63/-$:

1. Pop the operants and carry out the operation indicated by the operator, and push the result back onto the stack

2 3 + 6 3 / -
↑

Stack:
2
5

# Evaluating Postfix Expression: Example: 5

Use a stack to evaluate the postfix expression $23 + 63/-$:

1. Find another operator, pop the operants and carry out the operation indicated by the operator, and push the result back onto the stack.

2 3 + 6 3 / -
↑

Stack:
3

The answer is on the top of the stack

# Infix Expression Evaluation Example: Comparison

Let's see how to evaluate an infix expression such as

$$Z = (X + Y) * (W - U)$$

using different instruction formats such as

- three-address ISA
- two-address ISA
- one-address ISA
- stack ISA

# Infix Expression Evaluation Example: Three-Address ISA

Let's see how to evaluate an infix expression such as

$$Z = (X + Y) * (W - U)$$

using three-address ISA

```
ADD   R1, X,  Y
SUBR  R2, W,  U
MULT  Z,  R1, R2
```

# Infix Expression Evaluation Example: Two-Address ISA

Let's see how to evaluate an infix expression such as

$$Z = (X + Y) * (W - U)$$

using two-address ISA

```
LOAD   R1, X
ADD    R1, Y
LOAD   R2, W
SUBR   R2, U
MULT   R1, R2
STORE  Z,  R1
```

# Infix Expression Evaluation Example: One-Address ISA

Let's see how to evaluate an infix expression such as

$$Z = (X + Y) * (W - U)$$

using one-address ISA

```
LOAD   X
ADD    Y
STORE  TEMP
LOAD   W
SUBR   U
MULT   TEMP
STORE  Z
```

# Infix Expression Evaluation Example: Stack ISA

Let's see how to evaluate an infix expression such as

$$Z = (X + Y) * (W - U)$$

using stack ISA

```
PUSH  X
PUSH  Y
ADD
PUSH  W
PUSH  U
SUBR
MULT
POP   Z
```

# Table of Contents

# Instruction Length

It can be affected by the number of operands supported by the ISA.

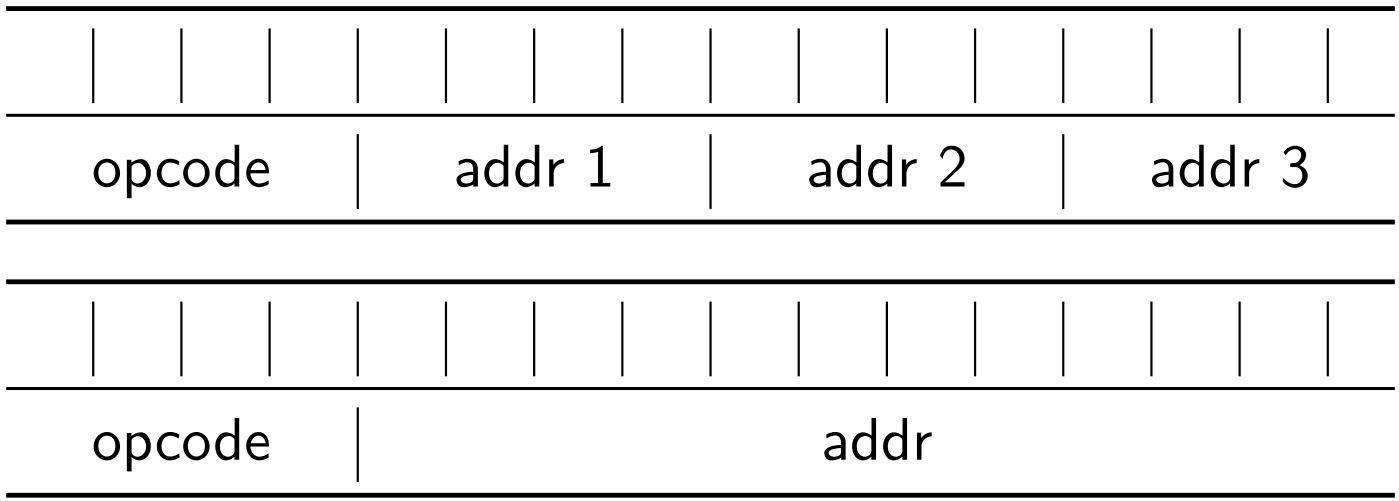
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as HALT, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use expanding opcodes.

# Opcode Consideration: Example

Assume that a system has 16 registers and 4K of memory.

- Need 4 bits to access one of the registers.
- Also need 12 bits for a memory address.

If the system is to have 16-bit instructions, we have two choices for the instructions:



# Varying Opcode Length

If we allow the length of the opcode to vary, we could create a very rich instruction set.

# Varying Opcode Length: Example

Given 8-bit instructions, is it possible to allow the following to be encoded?

- 3 instructions with 2 3-bit operands
- 2 instructions with 1 4-bit operand
- 4 instructions with 1 3-bit operand

We need:

- $3 \times 2^3 \times 2^3 = 192$
- $2 \times 2^4 = 32$
- $4 \times 2^3 = 32$

Total: $196 + 32 + 32 = 256$ bit patterns. With a total of 256 bit patterns required, we can exactly encode our instruction set in 8 bits! $(2^8 = 256)$

# Varying Opcode Length: Example Instructions

| | |
|---|---|
| 00 xxx xxx | |
| 01 xxx xxx | instructions with 2 3-bit operands |
| 10 xxx xxx | |
| 11 | espcape code |
| 1100 xxxx | instructions with 1 4-bit operand |
| 1101 xxxx | |
| 1110 | escape code |
| 1111 | escape code |
| 11100 xxx | |
| 11101 xxx | instructions with 1 3-bit operands |
| 11110 xxx | |
| 11111 xxx | |

# Table of Contents

# Instruction Types

Instructions fall into several broad categories:

- Data movement.
- Arithmetic.
- Boolean.
- Bit manipulation.
- I/O.
- Control transfer.
- Special purpose.

# Table of Contents

# Addressing

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- The actual location of an operand is its effective address.
- Certain addressing modes allow us to determine the address of an operand dynamically.

# Addressing Modes

- Immediate addressing is where the data is part of the instruction.
- Direct addressing is where the address of the data is given in the instruction.
- Register addressing is where the data is located in a register.
- Indirect addressing gives the address of the address of the data in the instruction.
- Register indirect addressing uses a register to store the address of the address of the data.
- Indexed addressing uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- Based addressing is similar except that a base register is used instead of an index register.
- In stack addressing the operand is assumed to be on top of the stack.

# Address Modes: Variations

There are many variations to these addressing modes including:

- Indirect indexed.
- Base/offset.
- Self-relative.
- Auto increment—decrement.

# Addressing Modes: Example

For the instruction shown, what value is loaded into the accumulator for each addressing mode?

Memory

| Addr | Content |
|---|---|
| 0x800 | 0x900 |
| ... | |
| 0x900 | 0x1000 |
| ... | |
| 0x1000 | 0x500 |
| ... | |
| 0x1100 | 0x600 |
| 0x1600 | 0x700 |

Register
R1 0x800

Instruction: LOAD 800

| Mode | Value → AC |
|---|---|
| Immediate | |
| Direct | |
| Indirect | |
| Indexed | |

# Addressing Modes: Example

For the instruction shown, what value is loaded into the accumulator for each addressing mode?

Memory

| Addr | Content |
|---|---|
| 0x800 | 0x900 |
| ... | |
| 0x900 | 0x1000 |
| ... | |
| 0x1000 | 0x500 |
| ... | |
| 0x1100 | 0x600 |
| 0x1600 | 0x700 |

Register
R1 0x800

Instruction: LOAD 800

| Mode | Value → AC |
|---|---|
| Immediate | 0x800 |
| Direct | 0x900 |
| Indirect | 0x1000 |
| Indexed | 0x700 |

# Table of Contents

# Instruction Pipelining and Instruction Level Parallelism

- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called instruction pipelining.
- Instruction pipelining provides for instruction level parallelism (ILP)

# Example Setup

Suppose a fetch-decode-execute cycle were broken into the following smaller steps:

1. Fetch instruction
2. Decode opcode
3. Calculate effective address of operands
4. Fetch operands
5. Execute instruction
6. Store result

Suppose we have a six-stage pipeline.

1. S1 fetches the instruction,
2. S2 decodes it,
3. S3 determines the address of the operands,
4. S4 fetches them,
5. S5 executes the instruction, and
6. S6 stores the result.

# Pipelining

For every clock cycle, one small step is carried out, and the stages are overlapped

| Clock Cycle | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|
| Instruction 1 Step | S1 | S2 | S3 | S4 | S5 | S6 | |
| Instruction 2 Step | | S1 | S2 | S3 | S4 | S5 | S6 |
| ... | ... | | | | | | |

- S1. Fetch instruction.
- S2. Decode opcode.
- S3. Calculate effective address of operands.
- S4. Fetch operands.
- S5. Execute.
- S6. Store result.

# Completion Time with Pipeline

To determine the theoretical speedup offered by a pipeline, we estimate the time to complete $n$ instructions (tasks).

- Assume a $k$-stage pipeline
- An instruction represents a task, $T_i$, $i = 1, 2, \ldots n, \ldots$, in the pipeline.
- Let $t_p$ be the time per stage.
- $T_1$ requires $kt_p$ time to complete in the k-stage pipeline.
- $T_2$ – $T_n$ emerge from the pipeline one per cycle. So the total time to complete the remaining tasks is $(n-1)t_p$.
- Thus, to complete $n$ tasks using the $k$-stage pipeline requires:

$$(kt_p) + (n-1)t_p = (k+n-1)t_p$$

# Completion Time without Pipeline

| Clock Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Step | S1 | S2 | S3 | S4 | S5 | S6 | S1 | S2 | S3 | S4 | S5 | S6 |
| | Instruction 1 | | | | | | Instruction 2 | | | | | |

To complete $n$ tasks without a pipeline requires: $nt_n = nkt_p$.

# Speedup

The speedup using the pipeline is then,

$$\text{Speedup } S = \frac{nt_n}{(kt_p) + (n-1)t_p} = \frac{nkt_p}{(k+n-1)t_p}$$

Considering that CPU executes many instructions, we have the theatrical speedup (maximum speedup),

$$\begin{aligned}
\lim_{n\to\infty} S &= \lim_{n\to\infty} \frac{nt_n}{(k+n-1)t_p} \\
&= \lim_{n\to\infty} \frac{nkt_p}{(k+n-1)t_p} \\
&= \lim_{n\to\infty} \frac{n}{(k+n-1)t_p} \frac{kt_p}{t_p} \\
&= \frac{kt_p}{t_p} = k
\end{aligned}$$

# Max Speedup

Max Speedup cannot be obtained when:

- the architecture supports fetching instructions and data in parallel.
- the pipeline can be kept filled at all times.

which are not always the case.

- Pipeline hazards arise that cause pipeline conflicts and stalls.

# Pipeline Hazards

An instruction pipeline may stall, or be flushed for any of the following reasons:

- Resource conflicts.
- Data dependencies.
- Conditional branching.

Measures can be taken at the software level as well as at the hardware level to reduce the effects of these hazards, but they cannot be totally eliminated.

# Pipeline Hazards: Example

Consider the two sequential statements `X = Y + 3` and `Z = 2 * X` that each can be realized via an instruction.

| Time Period | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| X = Y + 3 | fetch instruction | decode | fetch | execute & store X | |
| Z = 2 * X | | fetch instruction | decode | fetch X | |

The problem arises at time period 4.

- The second instruction needs to fetch X,
- but the first instruction does not store the result until the execution is finished,
- so X is not available at the beginning of the time period.

# Table of Contents

# Brief Introduction of Several Real-World Architectures

- Intel
- MISC
- ARM
- JVM

# Intel

Pipelining

- Intel introduced pipelining to their processor line with its Pentium chip.
- The first Pentium had two 5-stage pipelines. Each subsequent Pentium processor had a longer pipeline than its predecessor with the Pentium IV having a 24-stage pipeline.
- The Itanium (IA-64) has only a 10-stage pipeline

# Intel

Addressing Modes

- Intel processors support a wide array of addressing modes.
- The original 8086 provided 17 ways to address memory, most of them variants on the methods presented in this module.
- Owing to their need for backward compatibility, the Pentium chips also support these 17 addressing modes.
- The Itanium, having a RISC core, supports only one: register indirect addressing with optional post increment.

What is RISC?

# MISC

MIPS was an acronym for Microprocessor Without Interlocked Pipeline Stages.

- The architecture is little endian and word-addressable with three-address, fixed-length instructions.
- Like Intel, the pipeline size of the MIPS processors has grown:
  - The R2000 and R3000 have five-stage pipelines;
  - the R4000 and R4400 have 8-stage pipelines.
  - The R10000 has three pipelines: A five-stage pipeline for integer instructions, a seven-stage pipeline for floating-point instructions, and a six-state pipeline for LOAD/STORE instructions.

# MISC

Addressing Modes

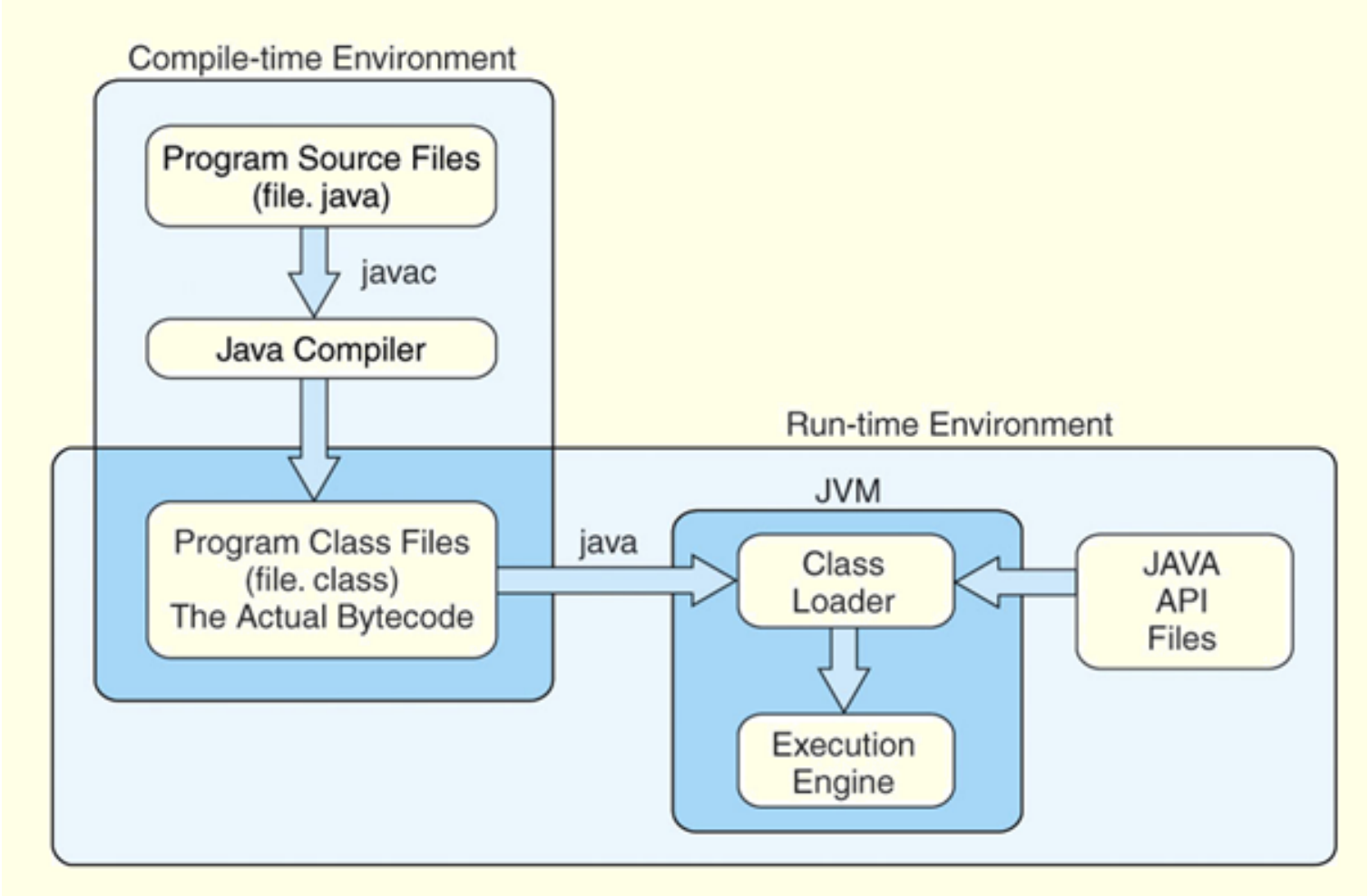
- In all MIPS ISAs, only the LOAD and STORE instructions can access memory.
- The ISA uses only base addressing mode.
- The assembler accommodates programmers who need to use immediate, register, direct, indirect register, base, or indexed addressing modes.

# JVM

The Java programming language is an interpreted language that runs in a software machine called the Java Virtual Machine (JVM).

- A JVM is written in a native language for a wide array of processors, including MIPS and Intel.
- Like a real machine, the JVM has an ISA all of its own, called bytecode. This ISA was designed to be compatible with the architecture of any machine on which the JVM is running.

# JVM


Compile-time Environment
Program Source Files
(file. java)
javac
Java Compiler
Run-time Environment
JVM
Program Class Files
(file. class)
The Actual Bytecode
java
Class
Loader
JAVA
API
Files
Execution
Engine

# JVM

- Java bytecode is a stack-based language.
- Most instructions are zero address instructions.
- The JVM has four registers that provide access to five regions of main memory.
- All references to memory are offsets from these registers. Java uses no pointers or absolute memory references.
- Java was designed for platform interoperability, not performance!

# ARM

- ARM is a load/store architecture: all data processing must be performed on values in registers, not in memory.
- It uses fixed-length, three-operand instructions and simple addressing modes.
- ARM processors have a minimum of a three-stage pipeline (consisting of fetch, decode, and execute);
- Newer ARM processors have deeper pipelines (more stages). Some ARM8 implementations have 13-stage integer pipelines.

# ARM

- ARM has 37 total registers but their visibility depends on the processor mode.
- ARM allows multiple register transfers.
- It can simultaneously load or store any subset of the16 general-purpose registers from/to sequential memory addresses.
- Control flow instructions include unconditional and conditional branching and procedure calls
- Most ARM instructions execute in a single cycle, provided there are no pipeline hazards or memory accesses.

# Table of Contents

# Summary and Q&A

- ISAs are distinguished according to
  - their bits per instruction,
  - number of operands per instruction, and
  - operand location and types and sizes of operands.
- Endianness as another major architectural consideration.
- CPU can store data based on:
  - A stack architecture
  - An accumulator architecture
  - A general purpose register architecture.

# Summary and Q&A

- Instructions can be fixed length or variable length.
- To enrich the instruction set for a fixed length instruction set, expanding opcodes can be used.
- The addressing mode of an ISA is also another important factor.
  - Immediate
  - Direct
  - Register
  - Register Indirect
  - Indirect
  - Indexed
  - Based
  - Stack

# Summary and Q&A

- A k-stage pipeline can theoretically produce execution speedup of k as compared to a non-pipelined machine.
- Pipeline hazards such as resource conflicts and conditional branching prevents this speedup from being achieved in practice.

Example architectures

- Intel, MIPS, JVM, and ARM architectures