# ELT3047 Computer Architecture

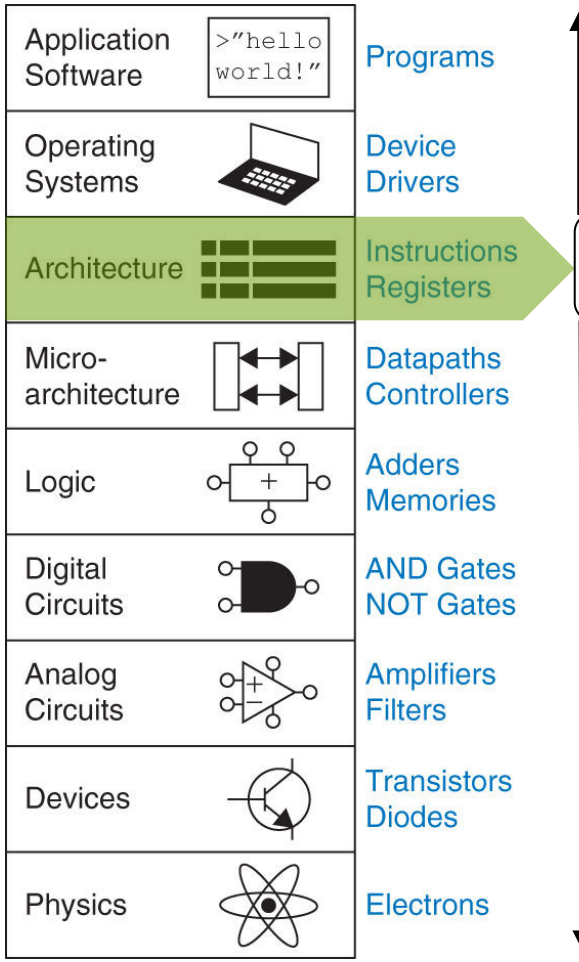
## Lesson 3: ISA design principles

Hoang Gia Hung
Faculty of Electronics and Telecommunications
University of Engineering and Technology, VNU Hanoi

# Last lecture review

- Various measures for computer performance
  - Execution time: the best performance measure for designers
  - MIPS/MFLOPS: easy to understand but contains many drawbacks
  - Benchmarks: use real applications – best performance measure for users
- Factors affecting execution time
  - Instruction counts
  - CPI
  - Clock cycle time (rate)
  - Power is a limiting factor (the power wall)
- Amdahl's law of diminishing returns
  - Improvement of one aspect is usually not proportional to improvement in overall performance.
- **Today's lecture**: ISA design principles

# Instruction Set Architecture




***Software***
(to be translated to the instruction set)

***Programmer's view of hardware: things to make machine code work correctly***

***abstraction*** layer that hides the complexity of CPU implementation from programmers.

***Hardware***
(implementation of the instruction set)

- Allows computer designers to think about functions ***independently*** from the hardware that performs them.
  - ✓ SW for Intel 80386 can still run on a Intel Rocket Lake.
- Different ISA's → different processors

# Processor Design Levels

- **Architecture (ISA)** **programmer/compiler view**
  - "functional appearance to its immediate user/system programmer"
  - **Data storage, addressing mode, instruction set, instruction formats & encodings.**
- **µ-architecture** **processor designer view**
  - "logical structure or organization that performs the architecture"
  - **Pipelining, functional units, caches, physical registers**
- **VLSI Realization (chip)** **chip designer view**
  - "physical structure that embodies the µ-architecture"
  - **Gates, cells, transistors, wires**

- Distinct Three Levels
  - Processors with identical ISA may be different in organization: Intel vs AMD
  - Processors with identical ISA and identical organization may still be different: Intel Core i9-11900K vs Intel Core i5-11600K

# The 5 Aspects in ISA Design

1. Data Storage
2. Memory Addressing Modes
3. Operations in the Instruction Set
4. Encoding the Instruction Set
5. The role of compilers

# ISA Design Principles

- Designing an ISA is hard:
  - What types of storage? How much?
  - How many instructions? What are they?
  - How to encode instructions? To minimize code size or to make hardware implementation simple?
  - How to future-proof?
- **Design principles:**
  1. Simplicity favors regularity
  2. Make the common case fast
  3. Smaller is faster
  4. Good design demands good compromises
- ***The quantitative methodology***
  - Take a set of benchmark programs expected to run on the system
  - Implement the benchmark programs with different ISA configurations
  - Pick the best one

# CISC vs RISC: the famous ISA battle

- Two major design philosophies for ISA:
  - Complex instruction set computer (CISC)
  - Reduced Instruction Set Computer (RISC)

| CISC | RISC |
|---|---|
| Many instructions and addressing modes | Few instructions and addressing modes |
| Single instruction performs complex operation | Simple instructions, combined by SW to perform complex operations |
| Smaller program size | Larger program size |
| Complex implementation | Easier to build/optimize hardware |
| Intel, AMD, Cyrix | MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC |

- **This course's** case study: **MIPS** (RISC)

# Aspect #1 – Data Storage

- Storage Architecture
- General Purpose Register Architecture

**Aspect #1: Data Storage**

Aspect #2: Memory Addressing Modes

Aspect #3: Operations in the Instruction Set

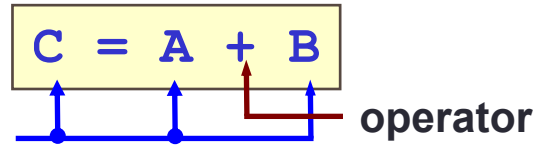
Aspect #4: Encoding the Instruction Set

Aspect #5: The role of compilers

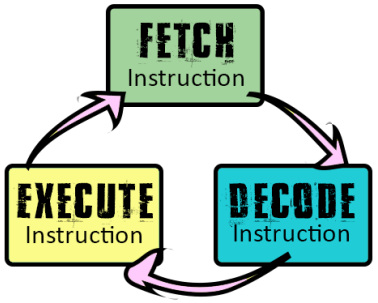
# Recap: Instruction & Instruction Set

- Instructions are fundamental (cơ bản) operations that CPU may execute.
  - Analogy to human sentence: operations (verbs) applied to operands (objects)
  - **Instruction set**: the repertoire of instructions like the vocabulary of the computer language.

*Operands may be implicit or explicit.*




- Stored program
  - A program is written as a sequence of instructions, which are stored in a memory, in conjunction with data, as binary bits.
  - Instructions are **automatically** fetched (lấy), decoded (giải mã), and executed (thực thi) one by one.




- **Registers**: small amount of fast memory built directly inside the processor by dedicated HW
  - Registers hold the fastest data available to the processor
  - Why is having registers a good idea? ← programs exhibit **data locality.**

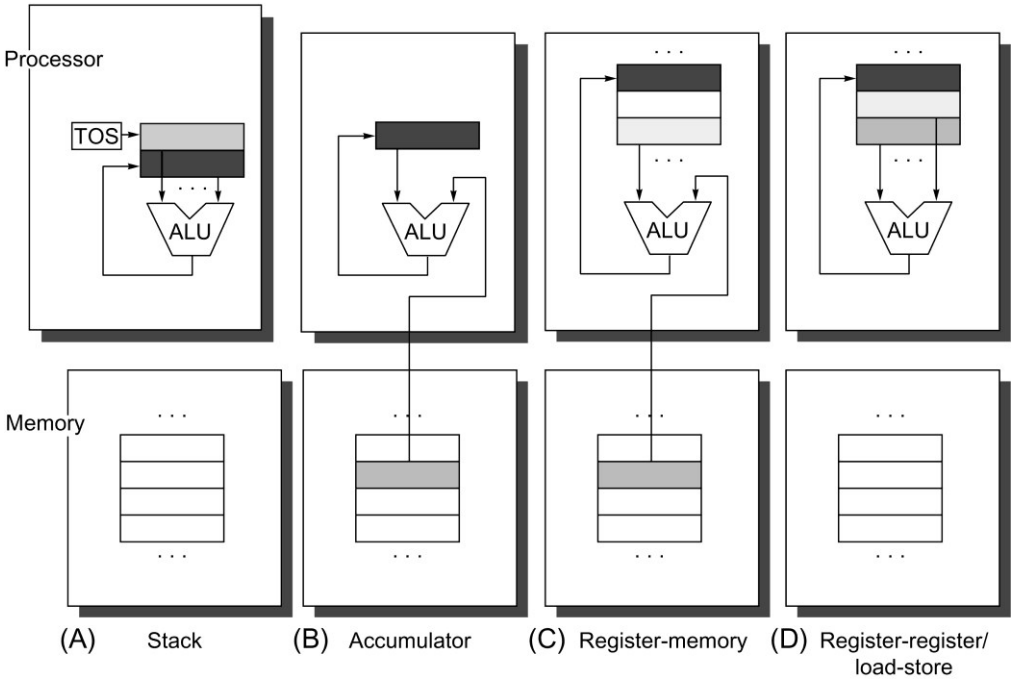
# Storage Architecture: Definition

- For a processor, **storage architecture** concerns with:
  - Where do we store the operands so that the computation can be performed?
  - Where do we store the computation result afterwards?
  - How do we specify the operands?
- Common storage architectures các loại kiến trúc bộ nhớ
  - **Stack:** usually implemented as a **register file** to store all operands & results; all operands are **implicitly** on top of the stack. kiến trúc ngăn xếp
  - **Accumulator** (1-operand machine): a special register (the accumulator) to store the result of a calulation, while also acting as an **implicit operand**. kiến trúc thanh ghi tích lũy
  - **General-purpose register architecture**: used only **explicit** operands, all registers good for all purposes kiến trúc GPRA
  - **Memory**: all operands & results are placed in the memory.

# Storage Architectures: Mechanisms

**C = A + B**

| Stack | Accumulator | Register (register-memory) | Register (load-store) |
|---|---|---|---|
| Push A | Load A | Load R1,A | Load R1,A |
| Push B | Add B | Add R3,R1,B | Load R2,B |
| Add | Store C | Store R3,C | Add R3,R1,R2 |
| Pop C | | | Store R3,C |



# Real-life implementation

- **Stack architecture:** a legacy from the “adding” machine days
  - Top portion of the stack inside CPU; the rest in memory.
    Examples: some technical handheld calculator, Z4 (by Conrad Zuse).
- **Accumulator architecture:**
  - One operand is implicitly in the accumulator.
    Examples: IBM 701, DEC PDP-8.
- **General-purpose register architecture:**
  - **Register-memory architecture:** one operand in memory. Examples: Motorola 68000, Intel 80386.
  - **Register-register (or load-store) architecture:** both operands in registers. Examples: MIPS, DEC Alpha.
- **Memory-memory architecture:**
  - All operands in memory. Example: DEC VAX.

# Storage Architecture: GPR Architecture

- For modern processors (after 1980):
  - General-Purpose Register (GPR) is the most common choice for storage design.
  - **RISC** computers typically uses Register-Register (Load/Store) design

    E.g. **MIPS**, ARM
  - **CISC** computers use a mixture of Register-Register and Register-Memory

    E.g. IA32.
- Reasons
  - Registers are much faster than memory
  - Registers are more efficient for a compiler to use

# Aspect #2 – Memory Addressing Mode

- Memory Locations and Addresses
- Addressing Modes

Aspect #1: Data Storage

**Aspect #2: Memory Addressing Modes**

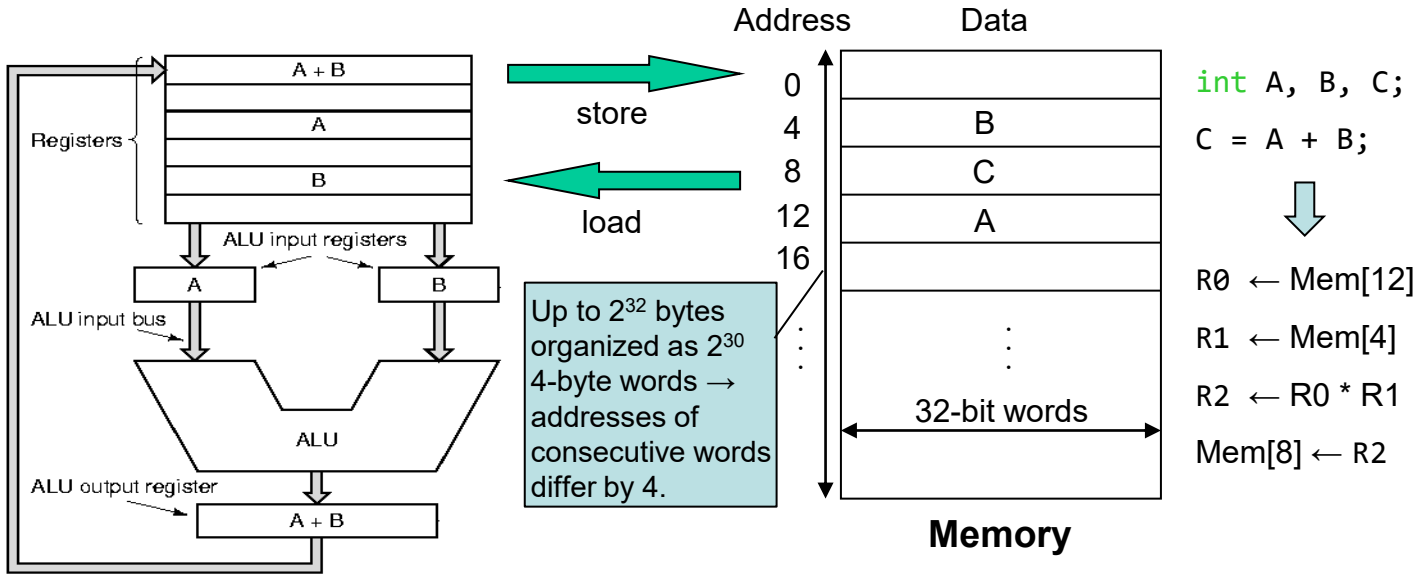
Aspect #3: Operations in the Instruction Set

Aspect #4: Encoding the Instruction Set

Aspect #5: The role of compilers

# Memory Address and Content

- Memory is viewed as a large, single-dimension array.
  - Each element of the array must be indexed → has a specific address. **MIPS**: ***byte*** addressed.
  - Given $k$-bit address → address space is of size $2^k$.
  - Each memory transfer consists of one ***word*** of $n$ bits → requires ***alignment***.
- Registers hold temporary values (operands)
  - Each register is referred to by a number/name (MIPS example next slide)



# MIPS register conventions

| Name | Register Number | Usage | Preserved on call |
|---|---|---|---|
| $zero | 0 | the constant value 0 | n.a. |
| $at | 1 | reserved for the assembler | n.a. |
| $v0-$v1 | 2-3 | value for results and expressions | no |
| $a0-$a3 | 4-7 | arguments (procedures/functions) | yes |
| $t0-$t7 | 8-15 | temporaries | no |
| $s0-$s7 | 16-23 | saved | yes |
| $t8-$t9 | 24-25 | more temporaries | no |
| $k0-$k1 | 26-27 | reserved for the operating system | n.a. |
| $gp | 28 | global pointer | yes |
| $sp | 29 | stack pointer | yes |
| $fp | 30 | frame pointer | yes |
| $ra | 31 | return address | yes |

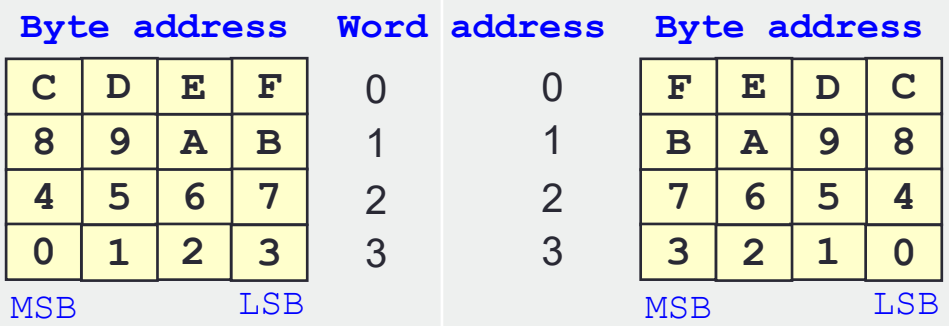
# More on MIPS registers

- There are other registers!
  - Not accessible to user (no $name/number).
- **PC:** Program counter
  - holds the address of the next instruction to be fetched from memory
- **LO** and **HI**
  - used specifically for multiply and divide (later in this course).

# Memory Content: Endianness

- **Endianness:**
  - The relative ordering of the bytes in a multiple-byte word stored in memory.

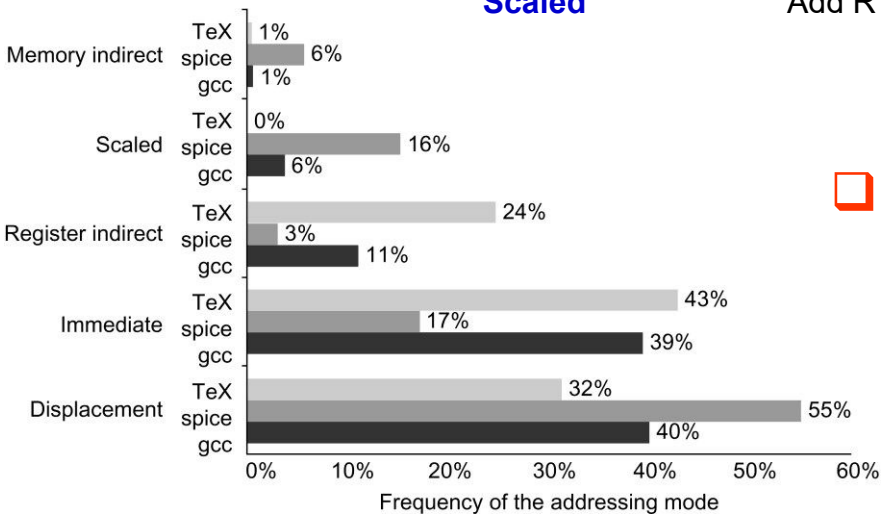
| Big-endian: | Little-endian: |
|---|---|
| Most significant byte stored in lowest address.<br>Example:<br>IBM 360/370, Motorola 68000, MIPS, SPARC. | Least significant byte stored in lowest address.<br>Example:<br>Intel 80x86, DEC VAX, DEC Alpha. |
| **Example:** 16 consecutive bytes **(0x) 0,1, …, E, F** are stored as | |



# Addressing Modes

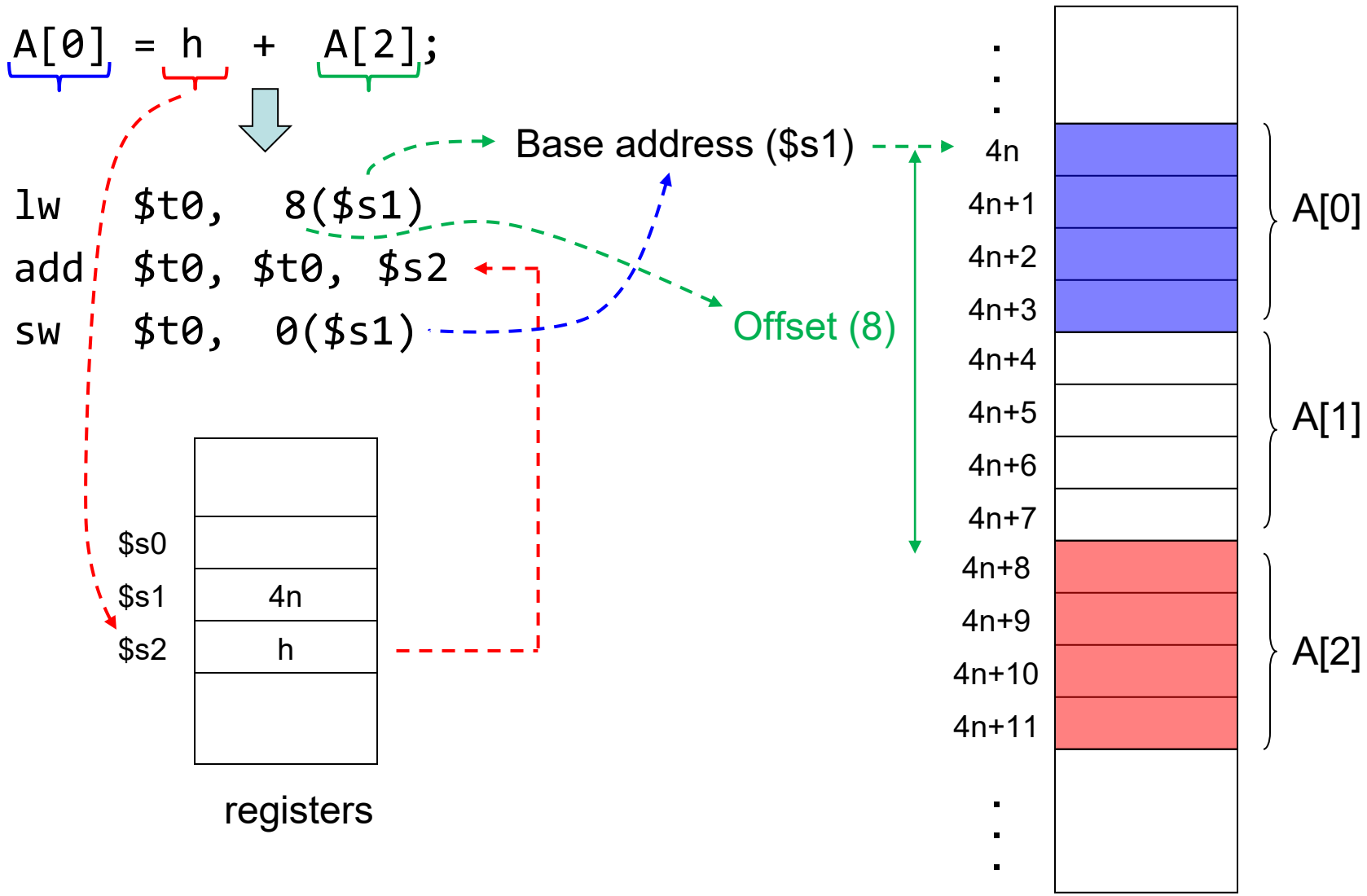
MIPS uses only the first 3 modes

| *Addressing mode* | *Example* | *Meaning* |
|---|---|---|
| **Register** | Add R4,R3 | R4 ← R4+R3 |
| **Immediate** | Add R4,#3 | R4 ← R4+3 |
| **Displacement** | Add R4,100(R1) | R4 ← R4+Mem[100+R1] |
| **Register indirect** | Add R4,(R1) | R4 ← R4+Mem[R1] |
| **Indexed / Base** | Add R3,(R1+R2) | R3 ← R3+Mem[R1+R2] |
| **Direct or absolute** | Add R1,(1001) | R1 ← R1+Mem[1001] |
| **Memory indirect** | Add R1,@(R3) | R1 ← R1+Mem[Mem[R3]] |
| **Auto-increment** | Add R1,(R2)+ | R1 ← R1+Mem[R2]; R2 ← R2+d |
| **Auto-decrement** | Add R1,–(R2) | R2 ← R2-d; R1 ← R1+Mem[R2] |
| **Scaled** | Add R1,100(R2)[R3] | R1 ← R1+Mem[100+R2+R3*d] |




- **Addressing Modes:**
  - Ways to obtain an operand of an instruction.

# Addressing modes example



# Aspect #3 – Operations in Instructions Set

- Standard Operations in an Instruction Set
- Frequently Used Instructions

Aspect #1: Data Storage

Aspect #2: Memory Addressing Modes

**Aspect #3: Operations in the Instruction Set**

Aspect #4: Encoding the Instruction Set

Aspect #5: The role of compilers

# Standard Operations

| Operator type | Examples |
|---|---|
| Arithmetic and logical | Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide |
| Data transfer | Loads-stores (move instructions on computers with memory addressing) |
| Control | Branch, jump, procedure call and return, traps |
| System | Operating system call, virtual memory management instructions |
| Floating point | Floating-point operations: add, multiply, divide, compare |
| Decimal | Decimal add, decimal multiply, decimal-to-character conversions |
| String | String move, string compare, string search |
| Graphics | Pixel and vertex operations, compression/decompression operations |

- Which one should ***our*** ISA support?
  - **MIPS** (RISC): complex operations $\xrightarrow{compiler}$ sequences of ***basic operations***

# Frequently Used Instructions

- Design principle #2: "*Make the common case fast*"

| Rank | 80x86 instruction | Integer average % total executed) |
|---|---|---|
| 1 | Load | 22% |
| 2 | Conditional branch | 20% |
| 3 | Compare | 16% |
| 4 | Store | 12% |
| 5 | Add | 8% |
| 6 | And | 6% |
| 7 | Sub | 5% |
| 8 | Move register-register | 4% |
| 9 | Call | 1% |
| 10 | Return | 1% |
| **Total** | | **96%** |

- *Average of five SPECint92 programs*

# Aspect #4 – Encoding the Instruction Set

- Instruction format
  - Instruction fields
  - Instruction length
- Instruction encoding alternatives

Aspect #1: Data Storage

Aspect #2: Memory Addressing Modes

Aspect #3: Operations in the Instruction Set

**Aspect #4: Encoding the Instruction Set**

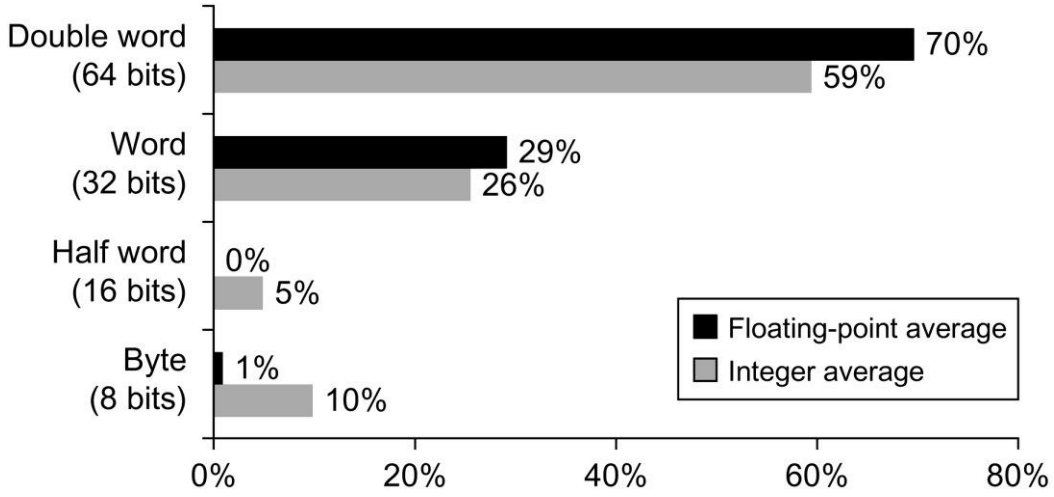
Aspect #5: The role of compilers

# Instruction Encoding: Overview

- Encoding = how instructions are represented in binary format for execution by the processor.
  - **Affects**: Code size, performance, hardware complexity
- Things to be decided:
  - Number of registers
  - Number of addressing modes
  - Instruction length
  - Number of operands in an instruction
- Different competing forces:
  - Have many registers and addressing modes
  - Reduce code size
  - Have instruction length that makes hardware implementation simpler.

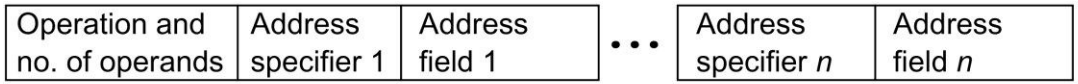
# Instruction fields

- An instruction consists of at least one of the following fields
  - **opcode**: unique code to specify the desired operation
  - **operands**: additional information needed for the operation
- The operation designates the type and size of the operands
  - Most accessed data type and size by SPEC: integer (byte, half-word, word, double word), floating point (byte, word, double word).

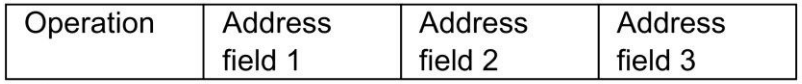


# Instruction length

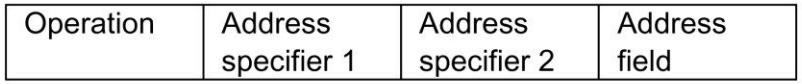
- Design choices for instruction length:

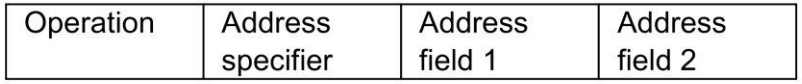



(A) Variable (e.g., Intel 80x86, VAX)




(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)


(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

# Instruction encoding alternatives

- **Variable-length** instructions.
  - Support ***any*** number of operands → enables smallest code size, because unused fields need not be included
  - Require multi-step fetch and decode → worst performance.
- **Fixed-length** instructions.
  - Fixed number of operands, with addressing modes (if options exist) specified as part of the opcode → largest code size.
  - Allow for easy fetch and decode + simplify pipelining and parallelism → best performance.
- **Hybrid** instructions.
  - Has multiple formats: fixed-length plus one or two variable-length instructions.
  - Improving the variability in size and work of the variable-length architecture while reducing the code size of fixed-length counterpart.

# Aspect #5 – The role of compilers

- The role of compilers
- ISA factors that affect compiler performance

Aspect #1: Data Storage

Aspect #2: Memory Addressing Modes

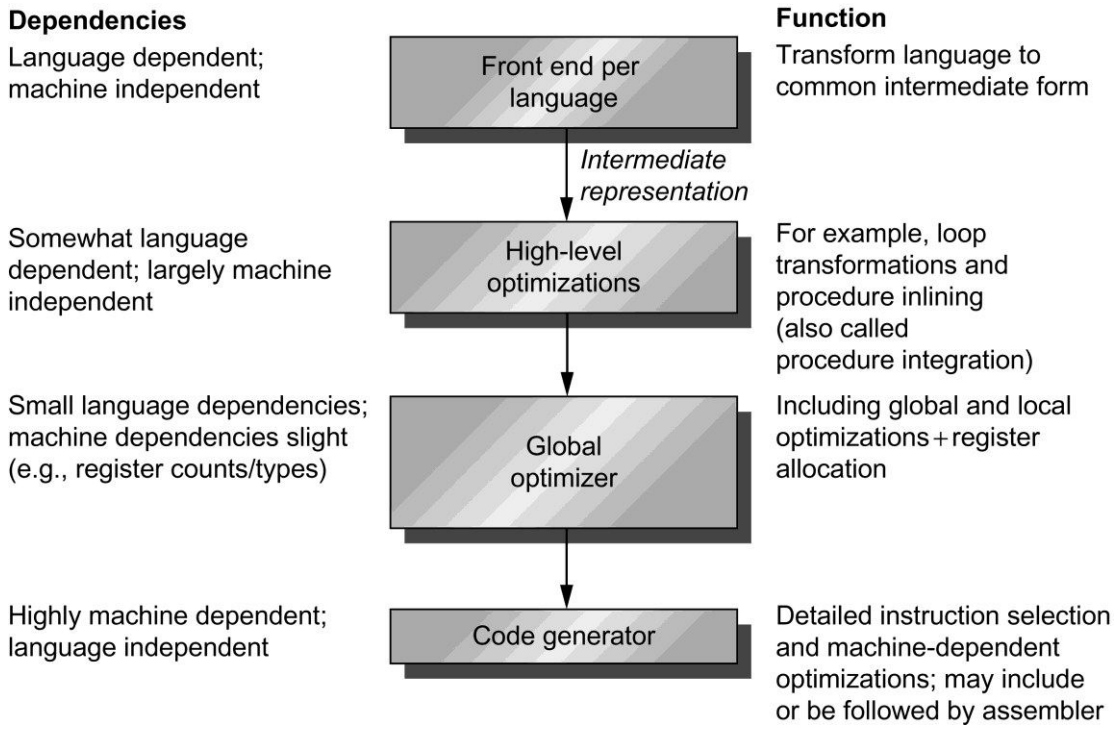
Aspect #3: Operations in the Instruction Set

Aspect #4: Encoding the Instruction Set

**Aspect #5: The role of compilers**

# The role of compilers

- To translate programs written in HLL to a target instruction set.
  - Significantly affects the performance of a computer.
  - Goals: ***correctness, speed of the compiled code***, then: fast compilation, debugging support, and interoperability among languages.



# ISA factors that affect compiler performance

- **Register allocation**
  - Optimizing ***passes*** must use registers to achieve best performance.
  - Register allocation algorithms require at least 16 (preferably 32) GPR's.
- **Regularity**, a.k.a. "law of least astonishment"
  - All addressing modes apply to all data transfer instructions (i.e. addressing modes & data transfer operations are ***orthogonal***).
  - Simplify code generation ← **design principle #1**: "*simplicity favors regularity*"
- **Instruction simplicity**
  - Special features that "match" a language construct (e.g. FOR and CASE statements) or a kernel function often make the compiler work more.
  - "*Provide primitives, not solutions*" – compiler works best with a minimalist instruction set.

# Summary

- ISA design is hard
  - Adhere to 4 qualitative principles
  - Applying quantitative method
- Five aspects of ISA design
  - ***Data Storage*** choices: GPR (load/store, register-memory), Stack, Register-memory, Accumulator.
  - Common ***addressing modes***: displacement, immediate, register indirect
  - Most important ***operations*** are simple instructions (96% of the instructions executed) → *make the common case fast*.
  - ***Instruction encoding***: performance vs code size trade-off (fixed- vs variable-length)
  - To support the ***compiler performance***: at least 16 (preferably 32) GPR's, aim for a minimalist instruction set, & ensure all addressing modes apply to all data transfer instructions.
- Next lecture: case study for MIPS ISA.