# ISA

## CISC vs RISC

- Complex Instruction Set Computer
  - eg. x86-32 (IA-32), IA-64...
  - Single Instruction performs complex operation.
  - Smaller program size (memory is expensive <assumed>)
  - Complex implementation ⇒ no room for hardware optimisation.
- Reduced Instruction Set Computer
  - eg. MIPS, ARM
  - Keep instruction set small & simple, easier to optimise hardware.
  - Burden on software to combine simpler operations to implement high-level language statements.
  - assumes more/cheaper memory, compiler optimisation.

## ISA Design

Data Storage (where do we get the data)

Memory Addressing Modes (How we access data from memory)

Operations in the Instruction Set (types of operations supported by processor)

Instruction Formats

Encoding the Instruction Set (into binary bits).

# Data Storage

- Storage Architecture.

  - Eg. General Purpose Register Architecture (MIPS)

  - Definition:

    - $C = A \overset{\text{operator}}{+} B$

      operands : may be implicit / explicit (register)

    - von Neumann Architecture

      └ Data (operands) are stored in memory.

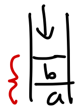
    - Concerns:

      1. Where do we store the operands so that computation can be performed.
      2. Where do we store the computation result afterwards
      3. How do we specify the operands.

  - Design.

    1. Stack Architecture

       - Operands are implicitly on top of the stack



push, pop.

LIFO

    2. Accumulator Architecture.

       - One operand is implicitly in the accumulator (a special register)
       - eg. IBM 701, DEC PDP-8.

    3. General Purpose Register Architecture  Modern / simple hardware design. \ balanced pipeline (optimisation)

       - Only explicit operands
       - CISC { Register-memory architecture (one operand in memory)
       - RISC { Register-register (load-store) architecture. MIPS., ARM

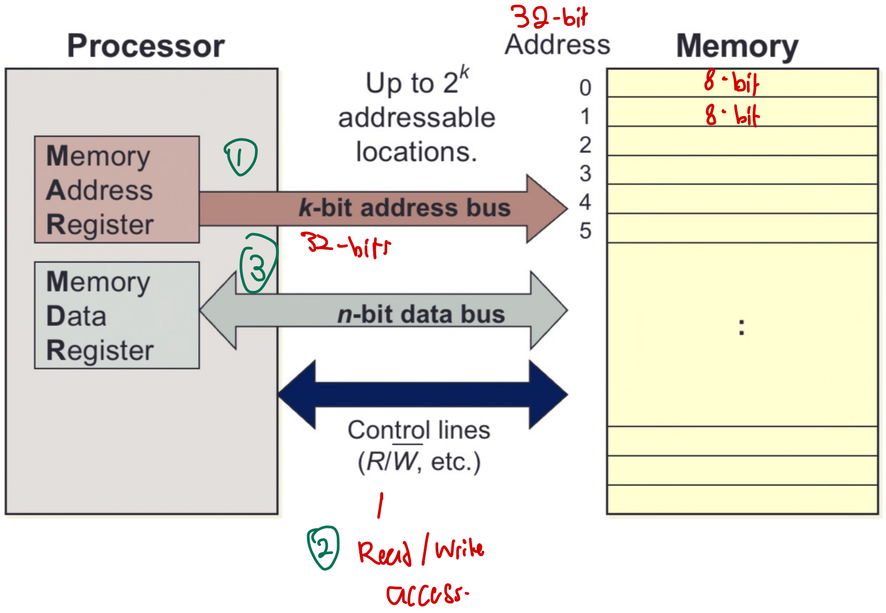
    4. Memory-Memory Architecture

       - All operands in memory.

See video for example L10P4

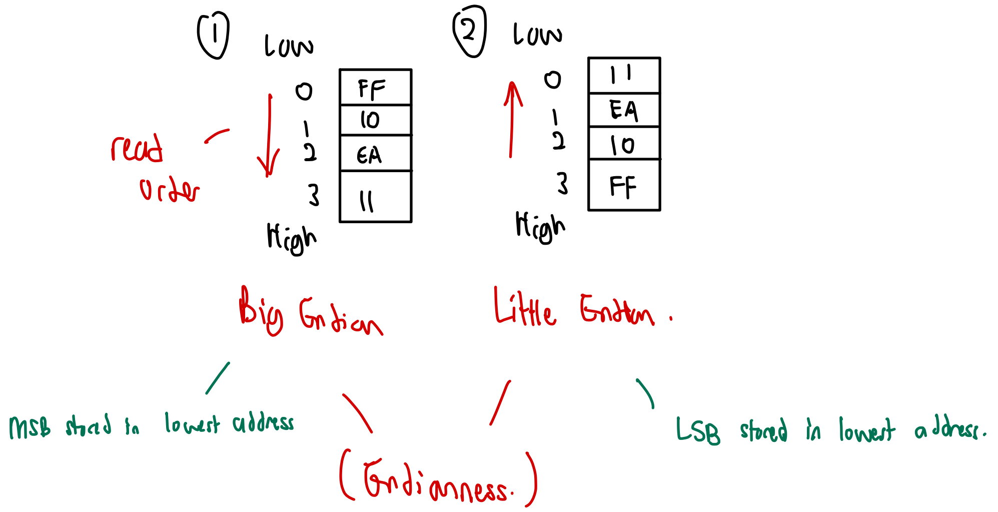
# Memory Addressing Mode.

- Given *k*-bit address, the address space is of size $2^k$ (32; $2^{32}$)
- Each memory transfer consists of one word of ***n*** bits



## Endianness.

- The relative ordering of bytes in a multiple-byte word stored in memory.
- Eg. 0x FF10EA11



# Addressing Modes

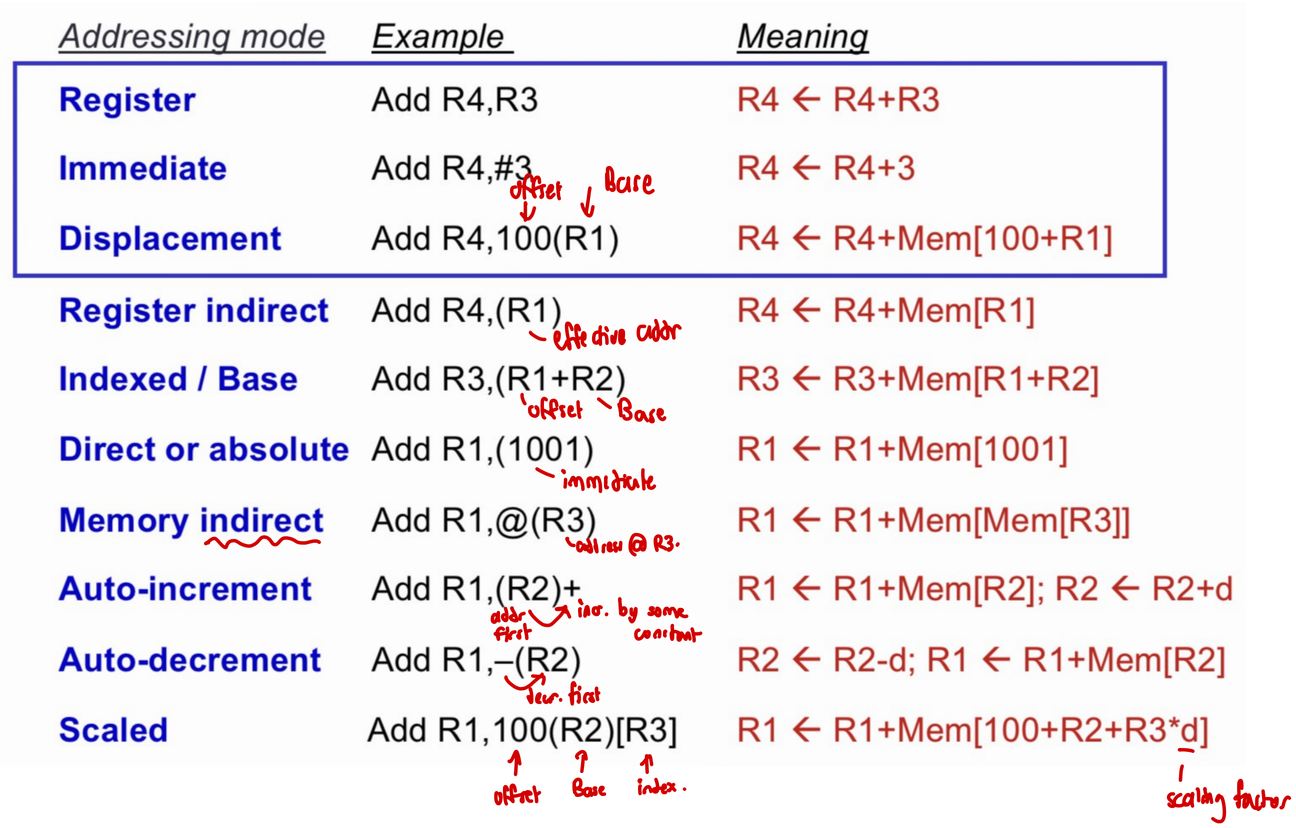
- Ways to specify an operand in an assembly language.
- In MIPS:

  Register - operand is in a register. eg. add

  Immediate - operand is specified in the instruction directly. eg. addi

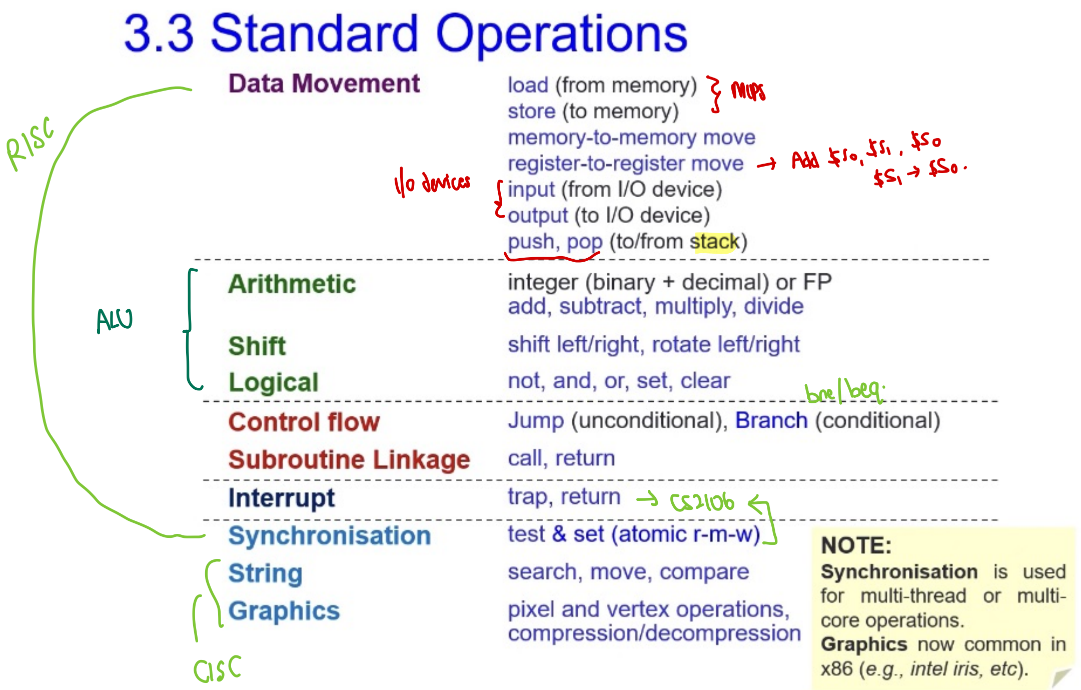
  Displacement - operand is in memory with address calculated as Base + offset eg. lw $t1, 20 ($t2)

- Others:



| Addressing mode | Example | Meaning |
|---|---|---|
| Register | Add R4,R3 | R4 ← R4+R3 |
| Immediate | Add R4,#3 | R4 ← R4+3 |
| Displacement | Add R4,100(R1)<br>offset ↓ 100, Base ↓ R1 | R4 ← R4+Mem[100+R1] |
| Register indirect | Add R4,(R1)<br>effective addr | R4 ← R4+Mem[R1] |
| Indexed / Base | Add R3,(R1+R2)<br>offset Base | R3 ← R3+Mem[R1+R2] |
| Direct or absolute | Add R1,(1001)<br>immediate | R1 ← R1+Mem[1001] |
| Memory indirect | Add R1,@(R3)<br>address @ R3. | R1 ← R1+Mem[Mem[R3]] |
| Auto-increment | Add R1,(R2)+<br>addr first, inc. by some constant | R1 ← R1+Mem[R2]; R2 ← R2+d |
| Auto-decrement | Add R1,–(R2)<br>decr. first | R2 ← R2-d; R1 ← R1+Mem[R2] |
| Scaled | Add R1,100(R2)[R3]<br>offset Base index. | R1 ← R1+Mem[100+R2+R3*d]<br>scaling factor |

# Operations in Instructions Set



## 3.3 Standard Operations

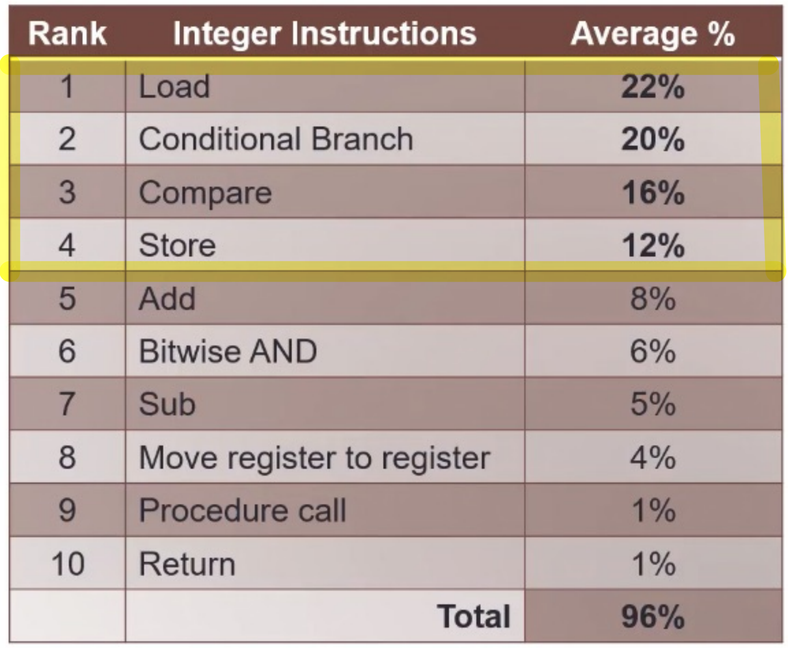
| | |
|---|---|
| **Data Movement** | load (from memory) } MIPS<br>store (to memory)<br>memory-to-memory move<br>register-to-register move → Add $s0, $s1, $s0<br>$s1 → $s0.<br>input (from I/O device) } I/O devices<br>output (to I/O device)<br>push, pop (to/from stack) |
| **Arithmetic** | integer (binary + decimal) or FP<br>add, subtract, multiply, divide |
| **Shift** | shift left/right, rotate left/right |
| **Logical** | not, and, or, set, clear |
| **Control flow** | Jump (unconditional), Branch (conditional) bne / beq |
| **Subroutine Linkage** | call, return |
| **Interrupt** | trap, return → CS2106 |
| **Synchronisation** | test & set (atomic r-m-w) |
| **String** | search, move, compare |
| **Graphics** | pixel and vertex operations,<br>compression/decompression |

RISC

ALU

CISC

**NOTE:**
**Synchronisation** is used for multi-thread or multi-core operations.
**Graphics** now common in x86 (*e.g., intel iris, etc*).



| Rank | Integer Instructions | Average % |
|---|---|---|
| 1 | Load | **22%** |
| 2 | Conditional Branch | **20%** |
| 3 | Compare | **16%** |
| 4 | Store | **12%** |
| 5 | Add | 8% |
| 6 | Bitwise AND | 6% |
| 7 | Sub | 5% |
| 8 | Move register to register | 4% |
| 9 | Procedure call | 1% |
| 10 | Return | 1% |
| | **Total** | **96%** |

70% of code
to optimise
processes

↓

Profiler Software

- memory / code
- Cache performance (mem in process)
- most freq. used instn.

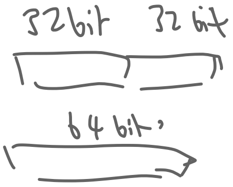
# Instruction Formats

- Instruction Length MIPS → 32 bits

CISC
- Variable-length Instructions
    - Req: multi-step fetch & decode — load one part of instruction, then load next part...
    - more flexible but complex and compact instruction set.

RISC
- Fixed-length Instructions — MIPS - 4 bytes long.
    - RISC
    - easy fetch & decode
    - Simple pipelining & parallelism — optimisation.
    - Instruction bits are scarce

↘ ARM 64 bit



VLIW arch.

(very-long-instruction-word)

- Hybrid Instruction.
    - mix of variable and fixed-length Instruction.

- Instruction Field Divide into smaller fields.

type & size of operands: - typical char (8 bits), word (32 bits)

- opcode: unique code to specify desired operation.
- operands: zero or more additional information needed for the operation.

- support for 8-, 16-, and 32-bit integer

$$\begin{tikzcd} 8 \arrow[d, dash] & 16 \arrow[d, dash] & 32 \arrow[d, dash] \\ lb & lhw & lw \end{tikzcd}$$

& 32/64-bit floating point operations.

- eg. R, I, J-format

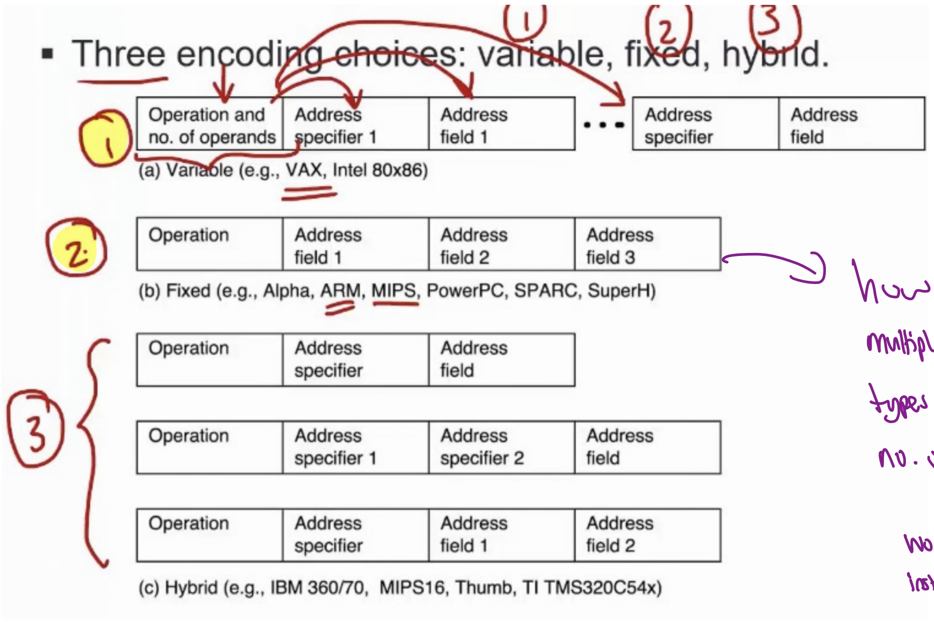
# Encoding the Instruction Set:

- Issues:
  - Code size RISC → Prog size↑
  - speed / performance
  - design complexity

- Decisions:
  - no. of registers → 32 reg → 5 bits in Instr. Encoding.
  - no. of addressing modes. → ↑ addr modes ∝ ↑ complexity
  - no. of operands in an Instruction. → 3 reg | 2reg + 1 Imm | reg + memory.

- Competing forces:
  - registers & addressing modes.
  - reduce code size. RISC (10 Instr) vs CISC (1 Instr)
  - instruction length that is easy to handle (i.e. fixed-length)

    reduced flexibity.



- Three encoding choices: variable, fixed, hybrid.

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

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

how to fit multiple set of instruction types into same (limited) no. of bits.

⇓

Work w/ most constrained instruction types.

⇓

expanding opcode scheme:

- opcode has variable length for different instruction.
- maximise the instruction bits.

i.e. when there's unused bits.

expand from operator's 5 bit.

$2^6-1$
$+\ 2^5$
instructions.

(type A) 6 bits → 11 bits (type B)

$2^6-\underline{1}$ — 1 assigned to type B

i.e. 111111 00000

$2^5$ (first 6 fixed to 111111)

i.e. 111111xxxxx. — $2^5$ instr.

# Maximise!

① maximise instr. that has more no. of bits, minimise the other

Type A → A $\boxed{000000.}$ – 1 instruction (minimised).

Type B → B 000001 $\left.\begin{matrix}00000\\ \vdots\\ 11111\end{matrix}\right\} 2^5$

$\underbrace{\phantom{000000}}_{6\text{-bit}}$ $\underbrace{\phantom{00000}}_{5\text{-bits.}}$

Cannot use anything

② Start w/ $\underline{000000}$

1 instr. Type A.

prefixes. ③

000010 → $2^5$

⋮

111111 → $2^5$

Total for type B = $(2^6-1)(2^5)$

$$\text{Total} = 1 + (2^6-1)(2^5)$$

$$= 2017$$

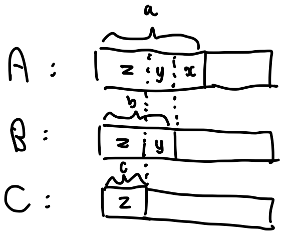
## Design an expanding opcode:

→ ① start w/ most restrictive case.

→ ② set the bits to satisfy most restrictive case

→ ③ Continue....

Technique:



Maximum = $2^a - 2^{a-x} - 2^{a-x-y}$ .... + 1 + 1 ....

Minimum = $(2^z-1) + (2^y-1) + (2^x)$ ...