Instruction Set Architecture

Isaac Pérez Andrade

ITESM Guadalajara
Department of Engineering & Architecture
Department of Computer Science

August - December 2020

Instruction Set Architecture

Instruction Set Architecture (ISA)

- It is an **abstract** concept which defines the portion of a computer that is visible to both the programmer and the compiler.
- It is part of the link between an application (something a human does such as video recording, playing music, editing a spreadsheet, etc) and the physical layer of the computer, *i.e.* Hardware (HW).

- ISA **theoretically** describes how a computer executes its programs.
- It describes:
  - The fundamental operations, which are simply referred to as *instructions*, that the computer can execute.
  - How these instructions are executed.
  - The semantics and rules required for the interaction of the different building blocks of a computer.

- Overall, ISA provides valuable information to the programmer.
  - Is a computer stack-, accumulator- or register-based?
  - Does the computer have memory? Does it have registers?
  - How many steps, *i.e.* clock cycles, does it take to execute instructions?
  - Where are operands fetched from?
  - Where is the result stored?
  - How big are data types?

Microarchitecture (μA)

- μA is more closely related to the **physical** implementation of a design, *i.e.*, μA determines how the ISA is implemented in HW.
  - For example, it describes which building are necessary in order to model a Microprocessor (μP) and how these building blocks are connected with each other.

- The same ISA may be physically implemented in a variety of μAs.
  - For example, one ISA could be implemented by different HW approaches and vendors such as Intel, ARM or AMD, and all three could have different performances.
- A naive adder example:
  - ISA specifies data width as 64-bits.
  - μA defines the adder as ripple-carry, carry-lookahead, carry-save, carry-select, etc.

- The goal of a processor designer is to evaluate the different trade-offs between ISA and μA in order to find a Pareto optimal system.
    - Power consumption.
    - Latency - How long does it take to complete a task.
    - Throughput - How many tasks can be completed in a given time.
    - Chip area.

Instruction Set Architecture

Same ISA, different μA- 45 nm technology

- x86 ISA.
- Quad Core.
- 2.6 GHz.
- 125 W.

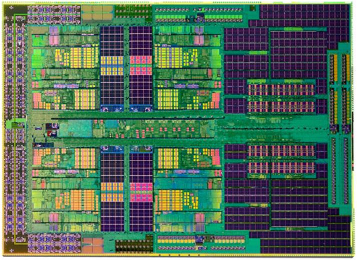


Figure 1: AMD Phenom X4

- x86 ISA.
- Dual Core.
- 1.6 GHz.
- 2 W.

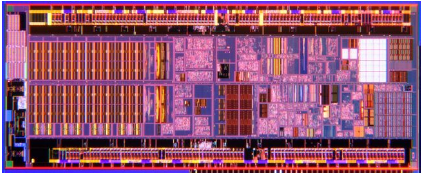


Figure 2: Intel Atom

Instruction Set Architecture

Different ISA, different μA- 45 nm technology

- x86 ISA.
- Quad Core.
- 2.6 GHz.
- 125 W.

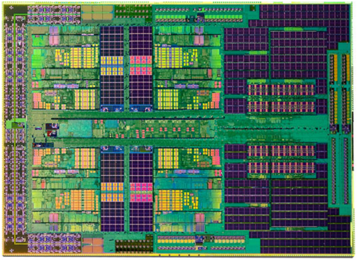


Figure 3: AMD Phenom X4

- Power ISA.
- Octa Core.
- 4.25 GHz.
- 200 W.

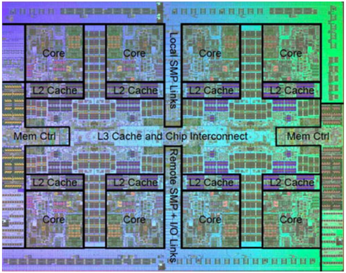


Figure 4: IBM Power7

ISA characteristics

ISA characteristics

- Type and size of instructions.
- Type and size of operands.
- Instruction encoding.
- Addressing modes.
- Registers

Registers

This is list of registers commonly found in μ**P**. Note that generally, μPs do not implement every single register in this list.

- **Program Counter (PC)**. Also called **IP**, points to the memory address of the next instruction to be executed.
- **Register File (RF)**. Set of registers used to store data.
- **Stack Pointer (SP)**. Points to the next location in the stack. Used in `PUSH` and `POP` operations.

Basic registers of a computer

- **Instruction Register (IR)**. Holds the instruction currently being executed.
- **Memory Address Register (MAR)**. Also called **Address Register (AR)**, points to the memory address to/from which data is stored/fetched to/from.
- **Instruction Memory (IM)**. Memory that stores the instructions that the μP will execute.

Basic registers of a computer

- **Accumulator (ACC)**. Holds the result of arithmetic and logic operations.
- **Data Memory (DM)**. Also called **Data Register (DR)**, holds operand(s) to be used in arithmetic and logic operations.
- **General Purpose Register (GPR)**. Registers to temporary store data or addresses.
- **Status Register (SR)**. Also called **Flag Register (FR)**, holds the special conditions of the result of arithmetic and logic operations, as well as branch and jump status. For example, indicates if a comparison resulted in an equality, if the result of an operation is zero, overflow, etc.

Basic registers of a computer

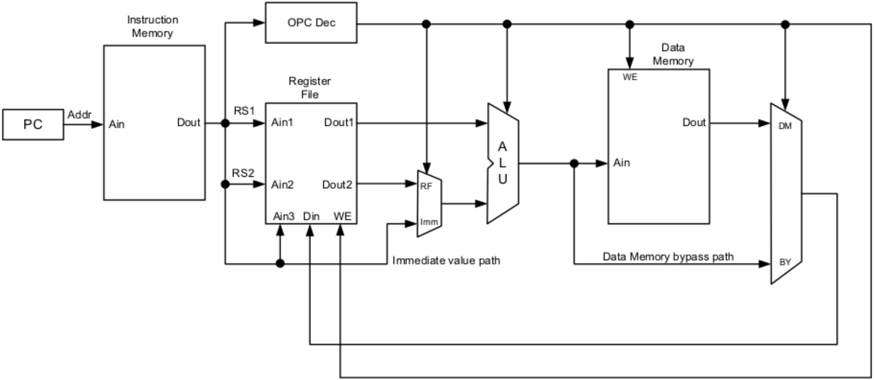


Figure 5: Basic structure of a μP.

Instructions

ISA characteristics

Instructions

- Instructions are the basic operations that a μP can understand and perform.
- The complete set of instructions that a μP may perform is called **instruction set**, **which is not the same as Instruction Set Architecture!**

Instruction type	Example[1]
Arithmetic & logical	`ADD`, `SUB`, `AND`, `OR`
Data transfer	`LOAD`, `SW`, `MOV`, `PUSH`, `POP`
Conditional branch	`BNE`, `BEQ`
Unconditional jumps	`JMP`, `JAL`, `CALL`, `RET`
System	`RD_INT`, `PRNT_CHR`
Floating Point	`FADD`, `FMULT`
String	`MOVSB`, `STR_MV`, `STR_CMP`
Signal processing[2]	`ADD_ARRAY`, `MULT_ARRAY`, `FFT`

[1] These examples are not specific to a particular ISA.
[2] Typically found in Single Instruction Multiple Data (SIMD) ISAs.

ISA characteristics

Instructions

Table 1: Intel's 80x86 top ten instructions based on five SPECint92 programs[3].

Rank	Type	Distribution
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%**

[3] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, 6th ed., p A-4, Morgan Kaufmann, 2019.

Operands and operations

- Where do operands come from?
- Where are results stored?
- What is the size of the operands?
- How many steps does an instruction take?

ISA characteristics

Operands and operations

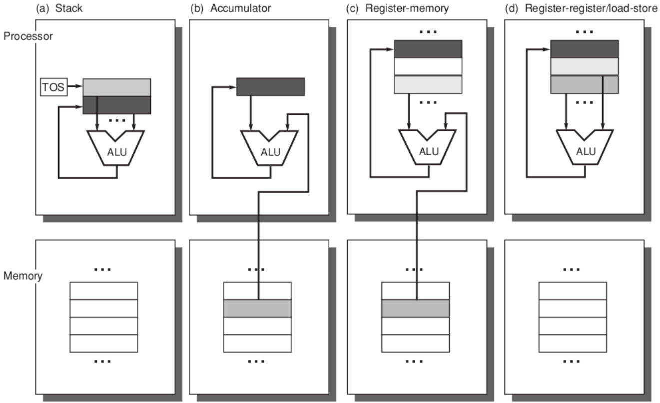
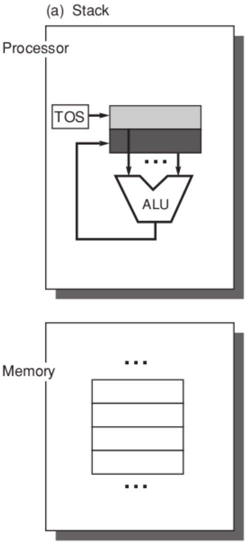


Figure 6: Operand locations for different ISAs [Figure A.1] [4].

[4] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, 6th ed., p A-4, Morgan Kaufmann, 2019.

ISA characteristics
Operands and operations

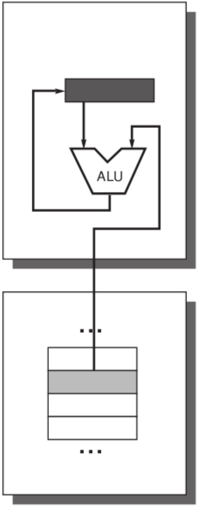


Stack-based ISA

`C = A + B`

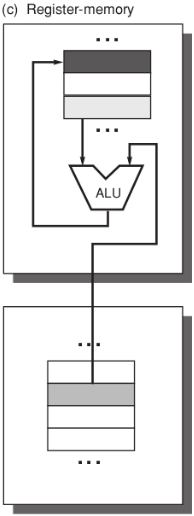
```
Push  A
Push  B
Add
Pop   C
```

(b) Accumulator



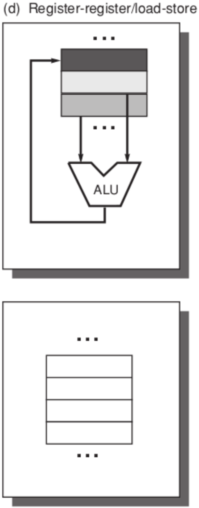
Accumulator-based ISA

```
C = A + B

Load    A
Add     B
Store   C
```



Memory-Register-based ISA

```
C = A + B

Load   R1  A
Add    R3  R1  B
Store  R3  C
```



Register-Register-based ISA

```
C = A + B
```

```
Load    R1   A
Load    R2   B
Add     R3   R1   R2
Store   R3   C
```

Addressing Modes

- How can we read/write data from/into memory?
- What types of memory exist?

Table 2: Examples of addressing modes

Mode	Example	Meaning
Immediate	`Add R4 , 3`	`R4` ← `R4` + `3`
Register	`Add R4 , R3`	`R4` ← `R4` + `R3`
Absolute (Direct)	`Add R2 , (100)`	`R2` ← `R2` + `Mem[100]`
Register indirect	`Add R4 , (R1)`	`R4` ← `R4` + `Mem[R1]`
Indexed	`Add R3 , (R1 + R2)`	`R3` ← `R3` + `Mem[R1 + R2]`
Displacement	`Add R4 , 100(R1)`	`R4` ← `R4` + `Mem[100+R1]`
Memory indirect	`Add R1 , @(R3)`	`R1` ← `R1` + `Mem[Mem[R3]]`

Table 3: Examples of addressing modes

Mode	Example	Meaning
Autoincrement	`Add R1 , (R2)+`	R1 $\leftarrow$ R1 + Mem[R2] R2 $\leftarrow$ R2 + d
Autodecrement	`Add R1 , -(R2)`	R2 $\leftarrow$ R2 - d R1 $\leftarrow$ R1 + Mem[R2]
Scaled	`Add R1 , 100(R2)[R3 ]`	R1 $\leftarrow$ R1 + Mem[100+R2 + R3 *d]

Note that Autoincrement and Autodecrement modes, the order of + and - signs influence the order of the operations. For example, `-(R2)` indicates decrementing `R2` before accessing to memory.

- Let's assume that registers `Ri`, $i \in [1, 4]$, and selected memory locations of a computer store the following values.

Reg	**Val**
`R1`	23
`R2`	11
`R3`	7
`R4`	19

Mem	**Val**
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

ISA characteristics

Addressing modes: Example

Mode	**Example**	**Meaning**
Immediate	`Add R4 , 3`	$\texttt{R4} \leftarrow \texttt{R4} + 3$

Reg	**Val**
R1	23
R2	11
R3	7
R4	19

Mem	**Val**
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

$\texttt{R4} = ?$

Mode	Example	Meaning
Register	`Add R4 , R3`	R4 $\leftarrow$ R4 + R3

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

R4 = ?

ISA characteristics

Addressing modes: Example

Mode	Example	Meaning
Absolute (Direct)	`Add R2 , (100)`	`R2` $\leftarrow$ `R2` + `Mem[100]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R2` = ?

Mode	Example	Meaning
Register indirect	`Add R4 , (R1)`	`R4` $\leftarrow$ `R4` + `Mem[R1]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R4` = ?

ISA characteristics

Addressing modes: Example

Mode	Example	Meaning
Indexed	`Add R3 , (R1 + R2)`	R3 $\leftarrow$ R3 + Mem[R1 + R2]

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

R3 = ?

ISA characteristics

Addressing modes: Example

Mode	Example	Meaning
Displacement	`Add R4 , 100(R1)`	`R4` $\leftarrow$ `R4` + `Mem[100+R1]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R4` = ?

ISA characteristics

Addressing modes: Example

Mode	Example	Meaning
Memory indirect	`Add R1 , @(R3)`	`R1` $\leftarrow$ `R1` + `Mem[Mem[R3]]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R1` = ?

Mode	Example	Meaning
Autoincrement	`Add R1 , (R2)+`	$\texttt{R1} \leftarrow \texttt{R1} + \texttt{Mem[R2]}$ $\texttt{R2} \leftarrow \texttt{R2} + d$

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

$d = 3$

$\texttt{R1} = ?$

Mode	**Example**	**Meaning**
Autodecrement	`Add R1 , -(R2)`	`R2` $\leftarrow$ `R2 -` d `R1` $\leftarrow$ `R1` $+$ `Mem[R2]`

Reg	**Val**
`R1`	23
`R2`	11
`R3`	7
`R4`	19

Mem	**Val**
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

$d = 4$

`R1` $=$?

ISA characteristics

Addressing modes: Example

Mode	Example	Meaning
Scaled	`Add R1 , 100(R2)[R3 ]`	R1 ← R1 + Mem[100+R2 + R3 *d]

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

$d = 3$

R1 = ?

Instruction encoding

- In stored-program computers, instructions and data are stored in memory.
- So, how does a processor know
  - how to differentiate between operations and operands?
  - which instruction to perform?
  - which `Reg` or `Mem` locations are the operands located?
  - which `Reg` or `Mem` locations should the result be stored to?
  - how to differentiate between

    $\texttt{R1} \leftarrow \texttt{R2 + R3}$

    $\texttt{R1} \leftarrow \texttt{R2 + Mem[R3]}$

    $\texttt{R1} \leftarrow \texttt{R2 + Mem[Mem[R3]]}$

    $\texttt{R1} \leftarrow \texttt{Mem[R2] + Mem[R3]}$
- Instruction encoding is a **convention used to differentiate the various operations in a μP, as well as operations from operands**.

- Instructions are encoded using binary representation.
- Suppose we want to design a processor that can implement the following instructions.

```
R1 ← R2 + R3
R1 ← R2 + Mem[R3]
R1 ← R2 + Mem[Mem[R3]]
R1 ← Mem[R2] + Mem[R3]
```

- We could assign a binary code to each of these 4 operations.

Instruction	**Binary code**
`R1 ← R2 + R3`	00
`R1 ← R2 + Mem[R3]`	01
`R1 ← R2 + Mem[Mem[R3]]`	10
`R1 ← Mem[R2] + Mem[R3]`	11

- Let's try to expand this implementation.

- Let `Rd` be the destination register and `Rsi` the source register, where `d` $\in [0, 3]$ and `i` $\in [0, 3]$.
- We could assign a binary code for each combination of `Rd` and `Rsi` in the instruction `Rd` $\leftarrow$ `Rs1` $+$ `Rs2`.

Instruction	Binary code	Hex code
`R0 ← R0 + R0`	00 0000	`00h`
`R0 ← R0 + R1`	00 0001	`01h`
⋮	⋮	⋮
`R1 ← R2 + R3`	01 1011	`1Bh`
⋮	⋮	⋮
`R2 ← R3 + R0`	10 1100	`2Ch`
⋮	⋮	⋮
`R3 ← R3 + R3`	11 1111	`3Fh`

- What about the μA of this encoding?

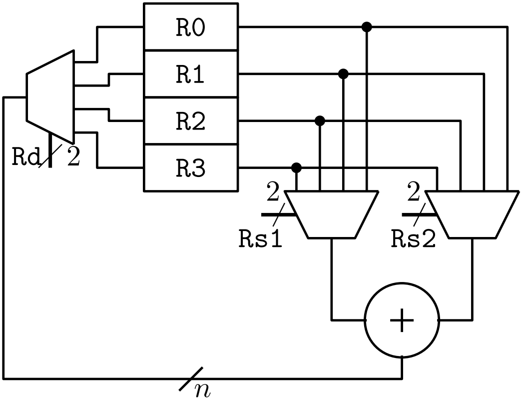


Figure 7: A naive μA for adding two registers.

ISA characteristics

Instructions encoding

- Is our previous scheme feasible?
- What's wrong with it?
- Could it be generalised?
- What about other addressing modes?
- How could we include other operations such as subtractions or jumps?
- Are all instructions represented using the same number of bits?

- We can continue to expand this scheme in order to add other operations, *e.g.*, subtraction and logical operations.
- Moreover, we can continue to include bits that represent different addressing modes.
- The ultimate goal of this, is to design an encoding feasible for all operations and addressing modes in our ISA.

ISA characteristics

Instructions encoding

- We could have a bit for selecting addition/subtraction in our previous design.

Table 4: Naive encoding for adding and subtracting two numbers.

Instruction	Binary code	Hex code
R0 ← R0 + R0	000 0000	00h
⋮	⋮	⋮
R1 ← R2 + R3	001 1011	1Bh
⋮	⋮	⋮
R3 ← R3 + R3	011 1111	3Fh
R0 ← R0 - R0	100 0000	10h
⋮	⋮	⋮
R1 ← R2 - R3	101 1011	5Bh
⋮	⋮	⋮
R3 ← R3 - R3	111 1111	7Fh

- As seen in the previous example, the source, destination and operation type may be represented with a **single** binary code.
- This concept may be further expanded for other operations and addressing modes.
- For this purpose, we may use longer binary codes, which may be divided into several fields.
- For example, we may use a field called **opcode** in order to represent the type of operation to be performed.
- Another field may represent the **source**, *i.e.*, the memory location where data to operate with is.
- Another field may represent the **destination**, *i.e.*, the memory location where the result of the operation should be stored to.

ISA characteristics

Instructions encoding

Variable encoding

Operation & #Operands	Addressing Mode 1	Address 1	• • •	Addressing Mode n	Address n

Fixed encoding

Operation	Address 1	Address 2	Address 3

Hybrid encoding

Operation	Addressing Mode	Address

Operation	Addressing Mode 1	Addressing Mode 2	Address

Operation	Addressing Mode	Address 1	Address 2

Figure 8: Generalised instruction encoding.

- **Variable encoding.**
  - Supports any number of operands, with each operand having a specific addressing mode.
  - The number of encoded bits varies between instructions.
  - **Compact machine programs.**
  - **Harder to decode.**
- **Fixed encoding.**
  - Every instruction has the same number of operands, with addressing modes specified in the opcode.
  - The number of encoded bits is always the same regardless of the instruction or addressing modes of the operands.
  - **Easier to decode.**
  - **Wasted bits in some instructions.**
- **Hybrid encoding.**
  - Instructions two different encoding lengths (16- and 32-bits, for example).

- ISAs may be classified into two main categories according to the complexity of their instruction encoding.
- **Reduced Instruction Set Computer (RISC).**
- **Complex Instruction Set Computer (CISC).**

CISC

- ISAs that perform complex operations and the instruction formats are not uniform.
- Large number of instructions available.
- Microcode approach.
  - A single instruction may be divided into several smaller instructions.
  - For example, a single instruction may perform a load from memory, an arithmetic operation and a store to memory.
- Reduced size of the compiled code due to **variable-length** encoding.
  - Shortest encodings represent the most commonly used instructions.

RISC

- ISAs that have a small number of simple, **fixed-length** instructions.
- Single-cycle instructions.
- Load-store approach.
  - Only load and store instructions are used for transferring data between registers and memory.

ISA characteristics

CISC vs RISC

```
mul16:
        pushl   %ebp            ; 01010101
        movl    %esp, %ebp      ; 1000100111100101
        movl    8(%ebp), %ecx   ; 100010000100110100001000
        pushl   %ebx            ; 01010011
        movl    12(%ebp), %edx  ; 100010110101010100001100
        xorl    %ebx, %ebx      ; 0011000111011011
        movl    $15, %eax       ; 1011100000001111
        .p2align 2,,3           ; 000000000000000000000000
                                ; 100011010111011000000000
.L6:
        testb   $1, %dl         ; 111101101100001000000001
        je      .L5             ; 0111010000000010
        addl    %ecx, %ebx      ; 0000000111001011
.L5:
        sall    %ecx            ; 1101000111100001
        shrl    %edx            ; 1101000111101010
        decl    %eax            ; 01001000
        jns     .L6             ; 0111100111110010
        movl    %ebx, %eax      ; 1000100111011000
        popl    %ebx            ; 01011011
        leave                   ; 11001001
        ret                     ; 11000011
```

Figure 9: CISC code example.

```
mul16:
        move    $6, $0          # 00000000000000000011000000100001
        li      $3, 15          # 00100100000000110000000000001111
$L6:
        andi    $2, $5, 0x1     # 00110000101000100000000000000001
        addiu   $3, $3, -1      # 00100100011000111111111111111111
        beq     $2, $0, $L5     # 00010000010000000000000000000010
        srl     $5, $5, 1       # 00000000000001010010100001000010
        addu    $6, $6, $4      # 00000000110001000011000000100001
$L5:
        bgez    $3, $L6         # 00000100011000011111111111111010
        sll     $4, $4, 1       # 00000000000001000010000001000000
        j       $31             # 00000011111000000000000000001000
        move    $2, $6          # 00000000110000000001000000100001
```

Figure 10: RISC code example.

RISC-V

RISC-V main features.

- 32-bit encoding.
- 4 types of instructions.
  - R-type. Register.
  - I-type. Immediate.
  - S-type. Store, compare and branch.
  - U-type. Jump.

ISA characteristics
RISC-V example

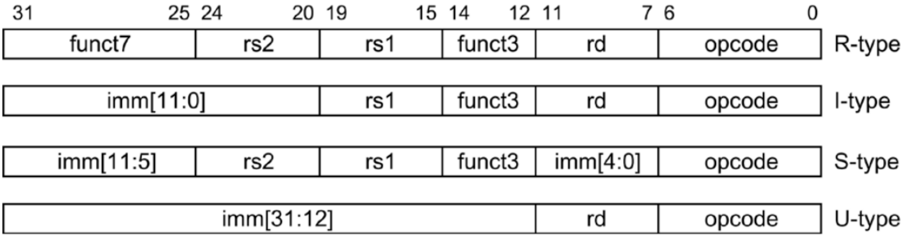


Figure 11: RISC instruction format.

- `funct7` and `funct3` complement the opcode.
- `rd`, `rs1` and `rs2` are destination, source 1 and source 2 registers, respectively.
- `imm` is an immediate value.

ISA characteristics
RISC encoding example

Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

Figure 12: RISC instruction description.

QUIZ

- Which of the following are affected by the instruction encoding?
  - A) The execution time of each instruction.
  - B) The μA of the processor.
  - C) Global warming.
  - D) The size of the compiled program.
  - E) All of the above.
  - F) None of the above.

QUIZ

- Which of the following are affected by the instruction encoding?
  - A) **The execution time of each instruction.**[5]
  - B) **The μA of the processor.**
  - C) Global warming.
  - D) **The size of the compiled program.**
  - E) All of the above.
  - F) None of the above.

[5] Think about CISC and RISC differences.

Summary

Summary

- ISA is the link between applications and HW.
- μA refers to the physical implementation of the ISA.
- The same ISA can be implemented in different μAs.
- ISA encloses
  - Type and size of instructions and operands.
  - Addressing modes.
  - Instruction encoding.
- There are RISC and CISC ISAs.
- There are several trade-offs associated between ISAs and μAs, and our goal is to find a Pareto-optimal design.

Further Reading

- Read about the difference between Von-Neumann and Harvard architectures.