2 Instruction set

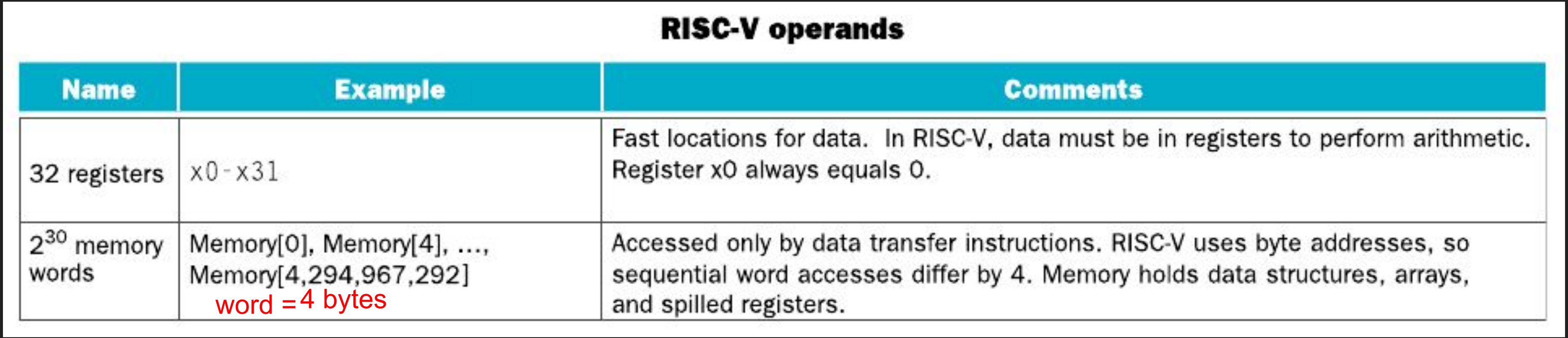
instrucciones disponibles para ejecutar en determinado procesador, y tmb
tipo de direccionamiento, cantidad de registros, tipos de registros (M2M, etc),etc

RISC-V

RISC-V is an open architecture that is controlled by RISC-V International, not a proprietary architecture that is owned by a company like ARM, MIPS, or x86. In 2020, more than 200 companies are members of RISC-V International, and its popularity is growing rapidly.

Condicion: todas las instrucciones tienen 32 bits (o 64 bits), A diferencia del CISC

Igual que RISC, tambien el x0
empieza en 0 y no se puede cambiar
ya que funciona como offset



RISC-V operands

Name	Example	Comments
32 registers	`x0-x31`	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

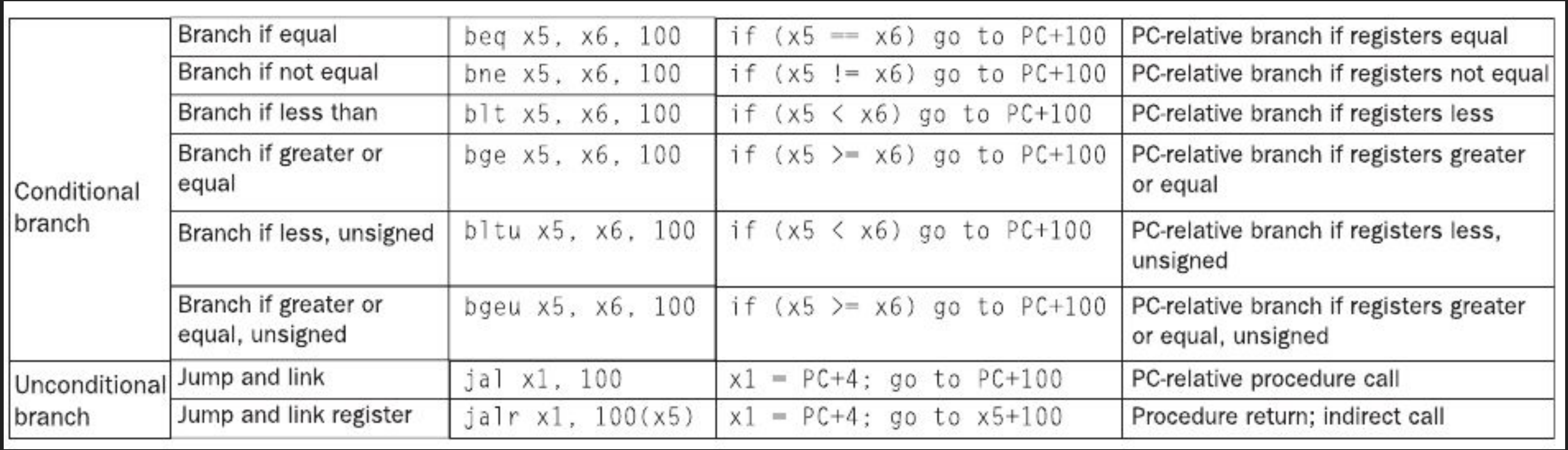
word = 4 bytes

registros: unidades de memoria auxiliares

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

un for() tambien sería conditional



Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

un return de una función es unconditional

example

C:

```
    a = b + c;

    d = a - e;
```

Risc-V assembler:

```
add a, b, c

sub d, a, e
```

example:

C:

```
f = (g + h) - (i + j)
```

Risc-V assembler:

```
add t0, g, h // temporary variable t0 contains g + h

add t1, i, j // temporary variable t1 contains i + j

sub f, t0, t1 // f gets t0 − t1, which is (g + h) − (i + j)
```

Memory operands

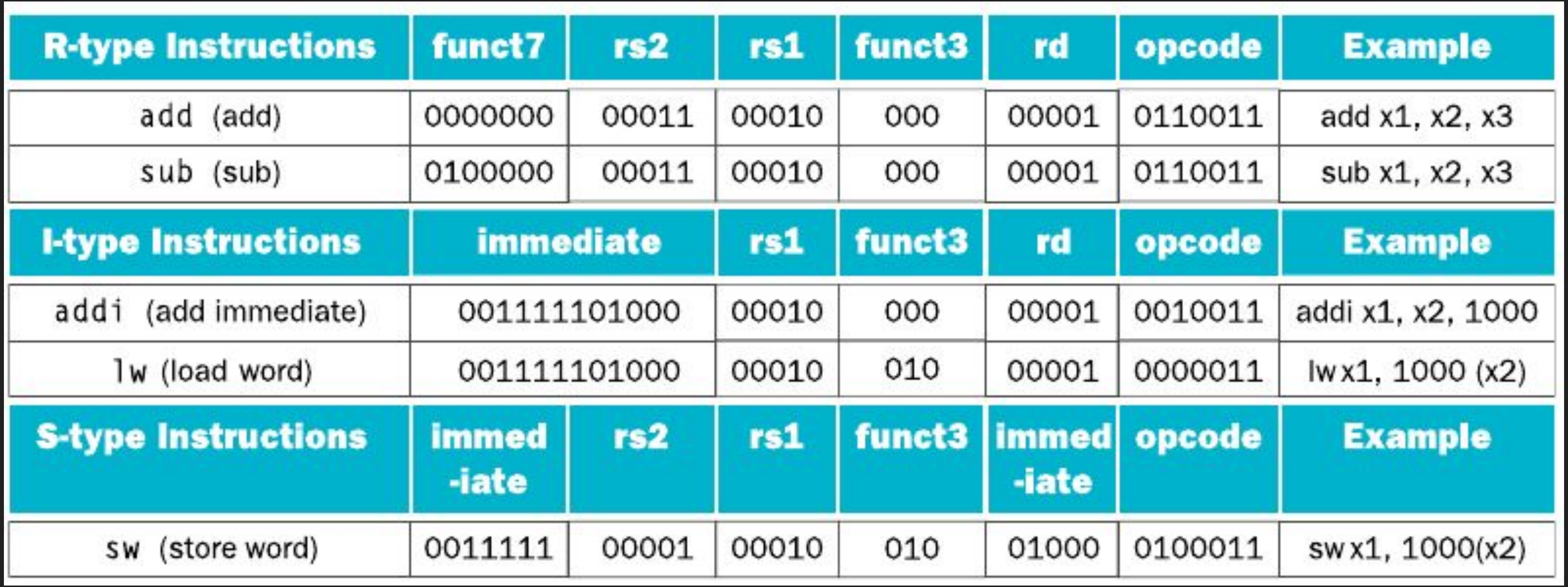
- arithmetic operations occur only on registers in RISC-V
- data transfer instructions.
- The format of the load instruction is the name of the operation followed by the register to be loaded, then register and a constant used to access memory.
- The sum of the constant portion of the instruction and the contents of the second register forms the memory address.

```
g = h + A[8];


lw      x9, 8(x22) // Temporary reg x9 gets A[8]
```

x22 tiene la direccion base del arreglo A (A[0]) y el 8 suma, por lo que va al valor x30 o A[8]

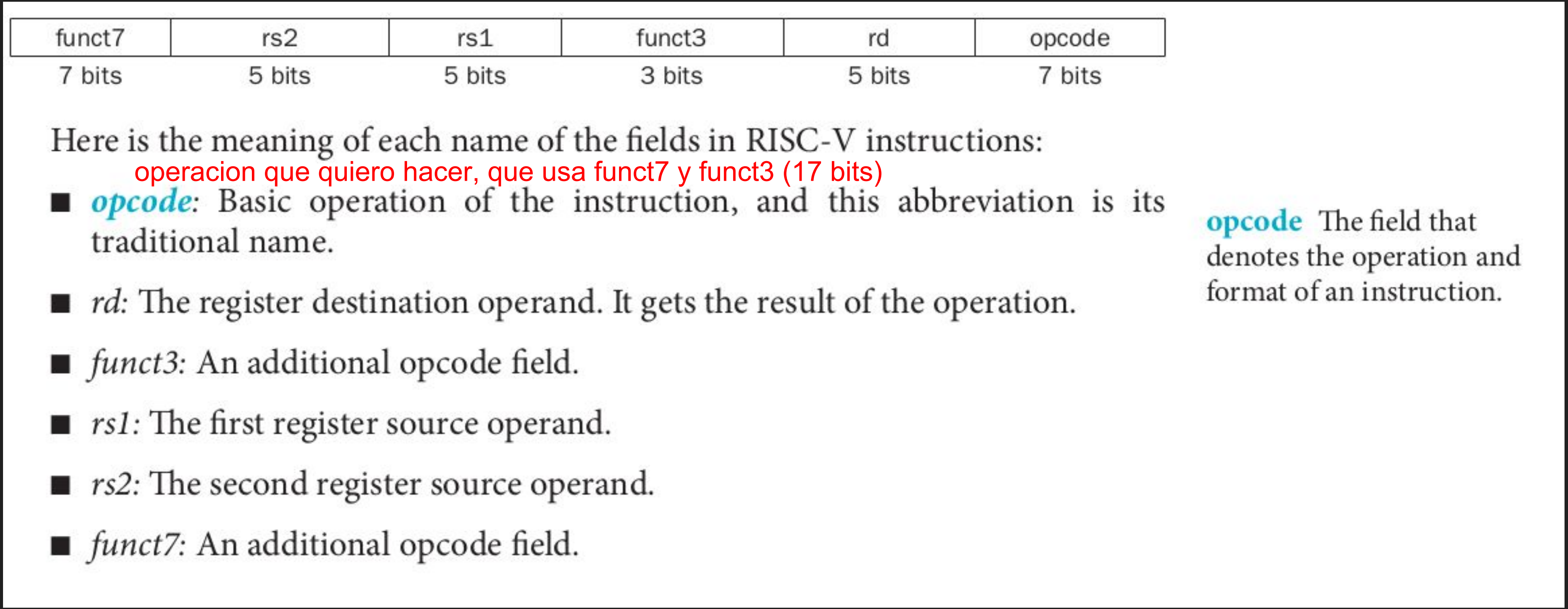
Representing instruction in the computer



R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
lw (load word)	001111101000		00010	010	00001	0000011	lw x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sw (store word)	0011111	00001	00010	010	01000	0100011	sw x1, 1000(x2)

Risc-V fields: r-type

aritmético-logicas



funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

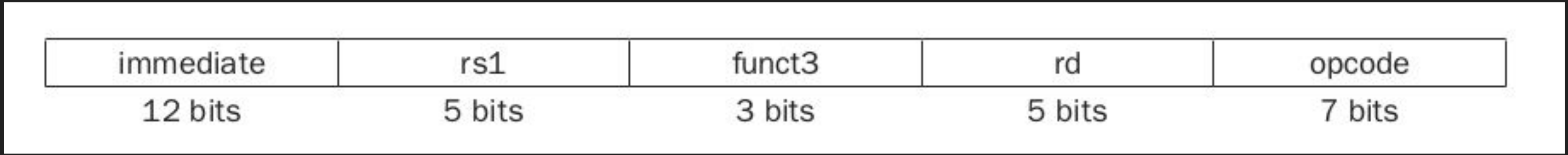
Here is the meaning of each name of the fields in RISC-V instructions:

operacion que quiero hacer, que usa funct7 y funct3 (17 bits)

- *opcode*: Basic operation of the instruction, and this abbreviation is its traditional name.
- *rd*: The register destination operand. It gets the result of the operation.
- *funct3*: An additional opcode field.
- *rs1*: The first register source operand.
- *rs2*: The second register source operand.
- *funct7*: An additional opcode field.

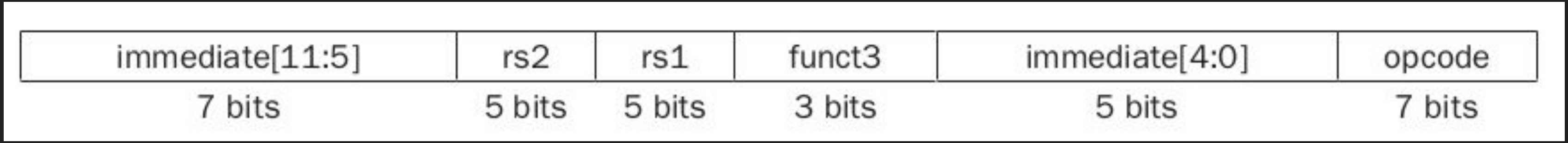
opcode The field that denotes the operation and format of an instruction.

I-type son las que tienen inmediato



immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

s-type store



immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Translating RISC-V Assembly into Machine Language

```
A[30] = h + A[30] + 1;
```

x9 = A[30]

x10 tiene la base de A (A[0])

```
lw x9, 120(x10)      // Temporary reg x9 gets A[30]
```

120(x10) porque tenemos que llegar al valor 31 del arreglo
y cada palabra tiene 4bytes (30*4=120)

```
add x9, x21, x9      // Temporary reg x9 gets h+A[30]
```

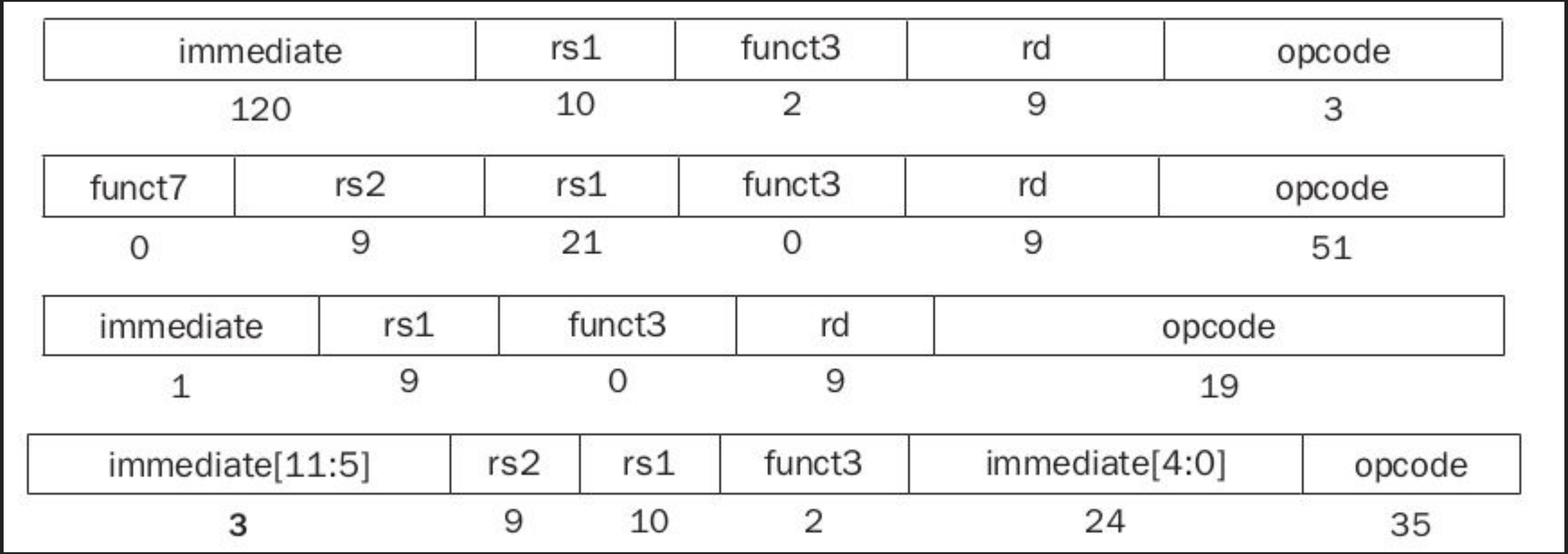
x9 = h + A[30]

```
addi x9, x9, 1       // Temporary reg x9 gets h+A[30]+1
```

x9 = h + A[30] +1

```
sw x9, 120(x10)      // Stores h+A[30]+1 back into A[30]
```

x9 es un registro temporal

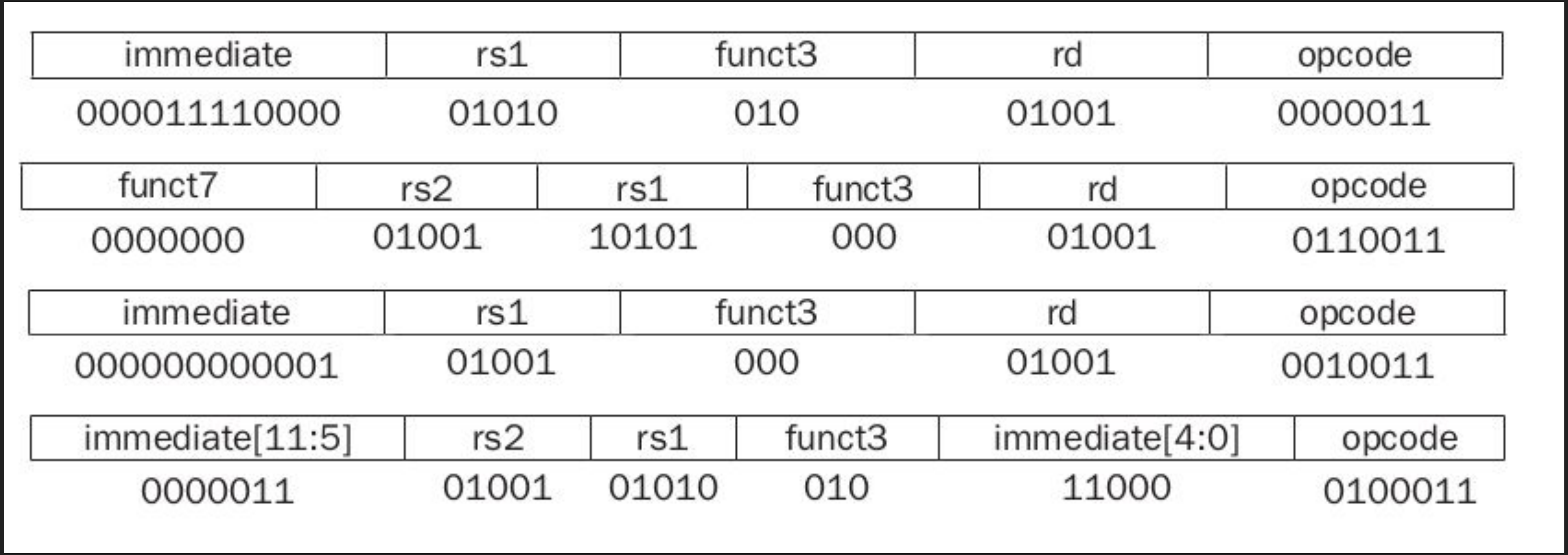


immediate	rs1	funct3	rd	opcode
120	10	2	9	3

funct7	rs2	rs1	funct3	rd	opcode
0	9	21	0	9	51

immediate	rs1	funct3	rd	opcode
1	9	0	9	19

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
3	9	10	2	24	35



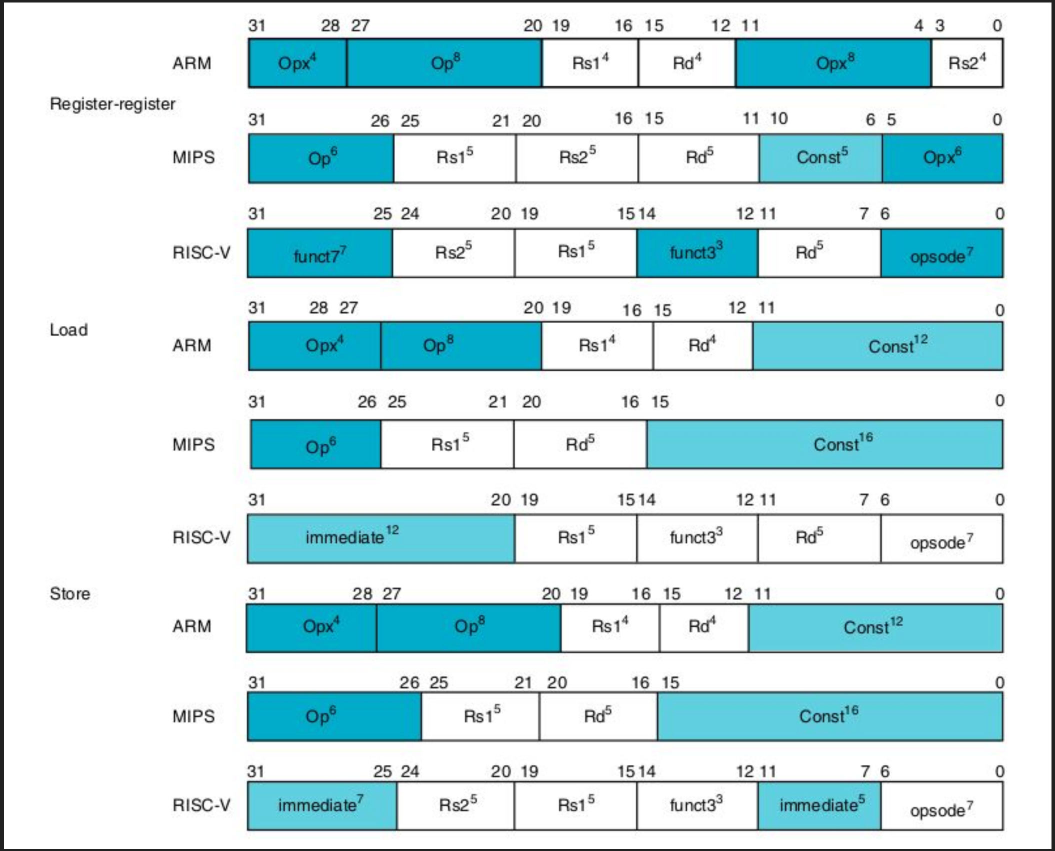
immediate	rs1	funct3	rd	opcode
000011110000	01010	010	01001	0000011

funct7	rs2	rs1	funct3	rd	opcode
0000000	01001	10101	000	01001	0110011

immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	0010011

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
0000011	01001	01010	010	11000	0100011

ARM vs MIPS vs Risc-V



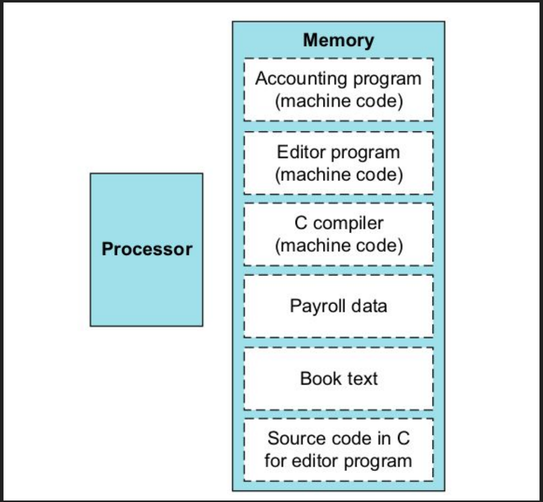
Stored-program concept

Today’s computers are built on two key principles:

1. Instructions are represented as numbers.

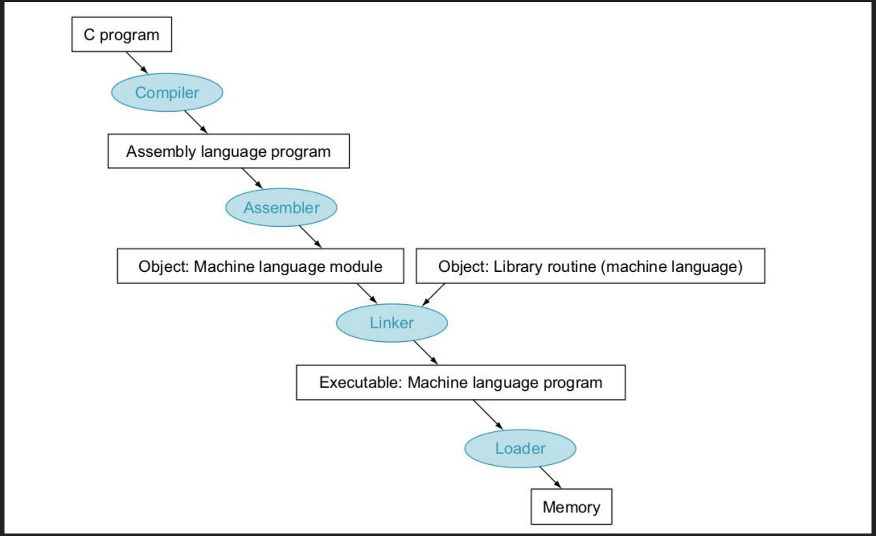
2. Programs are stored in memory to be read or written, just like data.

Memoria de programa


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

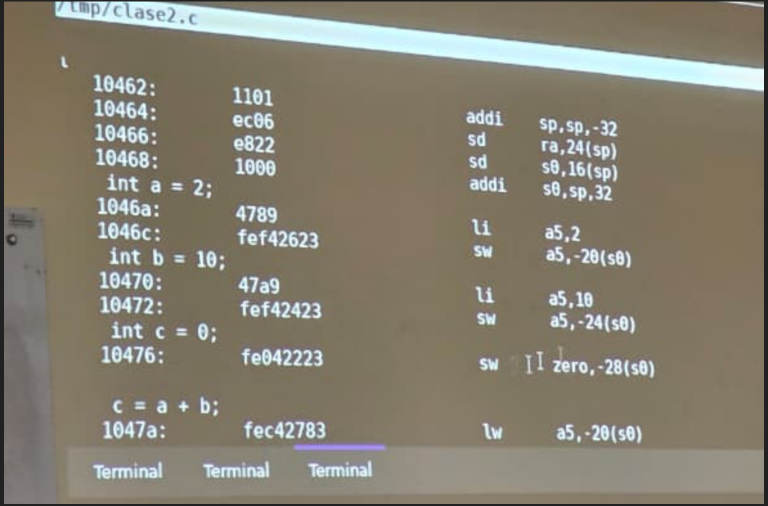
La memoria la podemos pensar como una tabla de verdad, la cual utilizamos para identificar el tipo de instruccion y ver para donde tiene que encarar el procesador

Translating and Starting a Program



C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Ejemplo:



```
/tmp/clase2.c

10462:      1101            addi   sp,sp,-32
10464:      ec06            sd     ra,24(sp)
10466:      e822            sd     s0,16(sp)
10468:      1000            addi   s0,sp,32
 int a = 2;
1046a:      4789            li     a5,2
1046c:      fef42623        sw     a5,-20(s0)
 int b = 10;
10470:      47a9            li     a5,10
10472:      fef42423        sw     a5,-24(s0)
 int c = 0;
10476:      fe042223        sw     zero,-28(s0)

 c = a + b;
1047a:      fec42783        lw     a5,-20(s0)
```