1. Los registros

En total hay 32 registros, denominados como `x0`-`x31`. La Tabla 1 los define todos. Es importante ver que los podemos nombrar tanto por su identificador de registro (con la "x"), como por su nombre. Aunque técnicamente podemos usar los registros con una funcionalidad diferente a la descrita, es ***esencial*** seguir la función *estándar* a fin de escribir código *compatible* con otro ya escrito.

Código 5-bit	Registro	Nombre	Descripción	Tipo
00000	`x0`	`zero`	Siempre cero	Cero
00001	`x1`	`ra`	dirección de retorno	Reservado
00010	`x2`	`sp`	puntero de pila	Reservado
00011	`x3`	`gp`	puntero global	Reservado
00100	`x4`	`tp`	puntero de hilo	Reservado
00101	`x5`	`t0`	registro temporal 0	Temporal
00110	`x6`	`t1`	registro temporal 1	Temporal
00111	`x7`	`t2`	registro temporal 2	Temporal
01000	`x8`	`s0`	registro salvado 0*	Salvado
01001	`x9`	`s1`	registro salvado 1	Salvado
01010	`x10`	`a0`	argumento 0 / retorno 0	Argumento
01011	`x11`	`a1`	argumento 1 / retorno 1	Argumento
01100	`x12`	`a2`	argumento 2	Argumento
.....	...	...	argumentos 3-6	Argumento
10001	`x17`	`a7`	argumento 7	Argumento
10010	`x18`	`s2`	registro salvado 2	Salvado
.....	...	...	registros salvados 3-10	Salvado
10010	`x27`	`s11`	registro salvado 11	Salvado
11100	`x28`	`t3`	registro temporal 3	Temporal
11101	`x29`	`t4`	registro temporal 4	Temporal
11110	`x30`	`t5`	registro temporal 5	Temporal
11111	`x31`	`t6`	registro temporal 6	Temporal

Tabla 1: Registros de RISC-V. * El registro `s0` también se usa como puntero de marco `fp`.

2. Las instrucciones

Dentro del repertorio base de instrucciones **RV32I**, encontramos instrucciones variadas. Las aritmético-lógicas nos permiten operar con valores en los registros. Las de acceso a memoria permiten leer (cargar) y escribir (guardar) datos en memoria. Las instrucciones de salto servirán para programar estructuras y condicionales (*if, while, for...*). Por último, las de control nos ayudarán a implementar funciones y código reutilizable.

Instrucción	Uso	Descripción	Tipo
`add rd, rs1, rs2`	Suma	$r_d = r_{s1} + r_{s2}$	R - Aritmética
`sub rd, rs1, rs2`	Resta	$r_d = r_{s1} - r_{s2}$	R - Aritmética
`xor rd, rs1, rs2`	Or exclusiva	$r_d = r_{s1} \wedge r_{s2}$	R - Lógica
`or rd, rs1, rs2`	Or	$r_d = r_{s1} \mid r_{s2}$	R - Lógica
`and rd, rs1, rs2`	And	$r_d = r_{s1} \,\&\, r_{s2}$	R - Lógica
`sll rd, rs1, rs2`	Shift lógico izquierda	$r_d = r_{s1} \ll r_{s2}$	R - Lógica
`srl rd, rs1, rs2`	Shift lógico derecha	$r_d = r_{s1} \gg r_{s2}$	R - Lógica
`sra rd, rs1, rs2`	Shift aritmético derecha	$r_d = r_{s1} \gg r_{s2}$	R - Lógica
`slt* rd, rs1, rs2`	Activa si $<$	$r_d = (r_{s1} < r_{s2})\,?\,1 : 0$	R - Lógica
`beq rs1, rs2, i12`	Salta si $=$	if $r_{s1} = r_{s2}$ $\{pc = pc + i_{12}\}$	B - Salto
`bne rs1, rs2, i12`	Salta si $\neq$	if $r_{s1} \neq r_{s2}$ $\{pc = pc + i_{12}\}$	B - Salto
`blt* rs1, rs2, i12`	Salta si $<$	if $r_{s1} < r_{s2}$ $\{pc = pc + i_{12}\}$	B - Salto
`bge* rs1, rs2, i12`	Salta si $\geq$	if $r_{s1} \geq r_{s2}$ $\{pc = pc + i_{12}\}$	B - Salto
`sb rd, i12(rs1)`	Guarda byte	$M[r_{s1} + i_{12}] = r_d$	S - Guardado
`sh rd, i12(rs1)`	Guarda short	$M[r_{s1} + i_{12}] = r_d$	S - Guardado
`sw rd, i12(rs1)`	Guarda palabra	$M[r_{s1} + i_{12}] = r_d$	S - Guardado
`lb* rd, i12(rs1)`	Carga byte	$r_d = M[r_{s1} + i_{12}]$	I - Carga
`lh* rd, i12(rs1)`	Carga short	$r_d = M[r_{s1} + i_{12}]$	I - Carga
`lw rd, i12(rs1)`	Carga palabra	$r_d = M[r_{s1} + i_{12}]$	I - Carga
`addi rd, rs1, i12`	`add` inmediato	$r_d = r_{s1} + i_{12}$	I - Aritmética
`xori rd, rs1, i12`	`xor` inmediato	$r_d = r_{s1} \wedge i_{12}$	I - Aritmética
`ori rd, rs1, i12`	`or` inmediato	$r_d = r_{s1} \mid i_{12}$	I - Lógica
`andi rd, rs1, i12`	`and` inmediato	$r_d = r_{s1} \,\&\, i_{12}$	I - Lógica
`slli rd, rs1, i12`	`sll` inmediato	$r_d = r_{s1} \ll i_{12}$	I - Lógica
`srli rd, rs1, i12`	`srl` inmediato	$r_d = r_{s1} \gg i_{12}$	I - Lógica
`srai rd, rs1, i12`	`sra` inmediato	$r_d = r_{s1} \gg i_{12}$	I - Lógica
`slti* rd, rs1, i12`	`slt` inmediato	$r_d = (r_{s1} < i_{12})\,?\,1 : 0$	I - Lógica
`jalr rd, i12(rs1)`	Salta y enlaza registro	$r_d = pc + 4$ $pc = rs1 + i_{20}$	I - Control
`jal rd, i20`	Salta y enlaza	$r_d = pc + 4$ $pc = pc + i_{20}$	J - Control
`auipc rd, i20`	Suma inmediato a pc	$r_d = pc + i_{20} \ll 12$	U - Control
`lui rd, i20`	Carga inmediato	$r_d = i_{20} \ll 12$	U - Datos

Tabla 2: La lista de instrucciones base. Las instrucciones con * tienen variantes acabadas en "u" (e.g: `bltu`) que operan con los valores como si fueran sin signo.

3. Las pseudoinstrucciones

Para facilitar nuestro trabajo como programadores, RISC-V incluye pseudoinstrucciones que nos ayudarán a escribir códigos más sencillos. Las pseudoinstrucciones se traducen por instrucciones base, cambiando operandos o utilizando varias instrucciones base para conseguir un funcionamiento más amplio del repertorio.

Pseudoinstrucción	Uso	Traducción
`li rd, i12`	Carga inmediato corto	`addi rd, zero, i12`
`li rd, i32`	Carga inmediato (32-bit)	`lui rd, i32[31:12]` `addi rd, rd, i32[11:0]`
`la rd, sym`	Carga dirección (relativa a pc)	`auipc rd, sym[31:12]` `addi rd, rd, sym[11:0]`
`mv rd, rs`	Copia de valor	`addi rd, rs, 0`
`not rd, rs`	Complemento a 1	`xori rd, rs, -1`
`neg rd, rs`	Complemento a 2	`sub rd, zero, rs`
`bgt* rs1, rs2, i12`	Salta si $>$	`blt rs2, rs1, i12`
`ble* rs1, rs2, i12`	Salta si $\leq$	`bge rs2, rs1, i12`
`beqz rs1, i12`	Salta si $= 0$	`beq rs1, zero, i12`
`bnez rs1, i12`	Salta si $\neq 0$	`bne rs1, zero, i12`
`bgez rs1, i12`	Salta si ≥ 0	`bge rs1, zero, i12`
`blez rs1, i12`	Salta si ≤ 0	`bge zero, rs1, i12`
`bgtz rs1, i12`	Salta si > 0	`blt zero, rs1, i12`
`j i20`	Salto incondicional	`jal zero, i20`
`call i12`	Salto a función (cerca)	`jalr ra, i12(ra)`
`call i32`	Salto a función (lejos)	`auipc ra, i32[31:12]` `jalr ra, i32[11:0](ra)`
`ret`	Vuelta de función	`jalr zero, 0(ra)`
`nop`	No operación	`addi zero, zero, 0`

Tabla 3: La lista de pseudoinstrucciones base. Las instrucciones con * tienen variantes acabadas en "u" (e.g: `bgtu`) que operan con los valores como si fueran sin signo.

4. Las etiquetas

Las instrucciones de salto, en su forma primitiva, aceptan un valor numérico como cantidad de bytes a saltar. Al escribir un código, calcular este valor a mano es muy incómodo y poco práctico. Por ello, en el código se pueden poner etiquetas de la forma:

```
destino: add a0, a1, a2
```

Para saltar a esa instrucción, por ejemplo cuando cierta igualdad se cumpla, nos bastará con hacer tan solo:

```
beq s0, s1, destino
```

Y el compilador traducirá "destino" por la cantidad de bytes adecuada de manera automática.