RISC-V

Mariano Street <mstreet@fceia.unr.edu.ar>

Arquitectura del Computador
Licenciatura en ciencias de la computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Versión 1.0
19 de noviembre de 2019

Índice general

1. Introducción **3**
- 1.1. ¿Qué es una arquitectura abierta? . . . 3
- 1.2. ¿Qué es una arquitectura RISC? . . . 3
- 1.3. ¿Por qué desarrollar una nueva arquitectura? . . . 4
- 1.4. Breve historia de RISC-V . . . 4
- 1.5. ¿Por qué enseñar RISC-V? . . . 5
- 1.6. ¿Quién usa RISC-V en la industria? . . . 5
- 1.7. ¿Cómo probar RISC-V? . . . 6
  - 1.7.1. Software . . . 6
  - 1.7.2. Hardware . . . 7

2. Arquitectura **8**
- 2.1. Objetivos de RISC-V . . . 8
- 2.2. Módulos . . . 8
  - 2.2.1. Distintas bases . . . 9
  - 2.2.2. RV32E: para embebidos . . . 10
  - 2.2.3. RV128I: planificación a futuro . . . 10
- 2.3. Registros . . . 11
- 2.4. Endianness . . . 12
- 2.5. Lenguaje de ensamblador . . . 13
  - 2.5.1. Sintaxis general . . . 13
  - 2.5.2. Directivas del ensamblador . . . 14
  - 2.5.3. Instrucciones . . . 16
  - 2.5.4. Etiquetas . . . 16
- 2.6. Instrucciones básicas . . . 17
  - 2.6.1. Operaciones aritméticas . . . 17
  - 2.6.2. Operaciones lógicas . . . 18
  - 2.6.3. Operaciones de corrimiento . . . 18
  - 2.6.4. Carga/Descarga . . . 19
  - 2.6.5. Saltos . . . 20
  - 2.6.6. Otras . . . 21

2.7. Pila 21
2.8. Convención de llamada 22
2.9. Codificación 23
2.9.1. Tipo R: registro/registro 23
2.9.2. Tipo I: inmediato 23
2.9.3. Tipo U: inmediato superior 23
2.9.4. Tipo S: almacenamiento 23
2.9.5. Tipo B: ramificación 23
2.9.6. Tipo J: salto 24

3. Extensión M: multiplicación y división 25
3.1. ¿Por qué en una extensión? 25
3.2. Instrucciones 26
3.3. División por cero y desborde 27

4. Extensiones F y D: punto flotante 28
4.1. Registros 28
4.2. Redondeo 29
4.3. Instrucciones de F 29
4.3.1. Operaciones 29
4.3.2. Conversión 31
4.3.3. Comparación y clasificación 31
4.3.4. Carga y descarga 32
4.3.5. Control y estado 32
4.4. Instrucciones de D 33
4.4.1. Operaciones 33
4.4.2. Conversión 34
4.4.3. Comparación y clasificación 35
4.4.4. Carga y descarga 35
4.5. Clasificación de números 36

5. Referencia de instrucciones 37
5.1. Instrucciones de I 37
5.2. Instrucciones de M 38
5.3. Instrucciones de F 39
5.4. Instrucciones de D 40
5.5. Pseudoinstrucciones 41

6. Enlaces y libros 43
6.1. Enlaces 43
6.2. Libros 44

Capítulo 1

Introducción

RISC-V es una arquitectura de computadoras abierta y moderna de tipo RISC. Surgió en la academia y hoy en día es desarrollada por la Fundación RISC-V, una asociación de cientos de organizaciones interesadas en esta arquitectura.

1.1. ¿Qué es una arquitectura abierta?

Una arquitectura abierta es un diseño que puede ser usado por cualquier persona para hacer su propio hardware o software. No está atada a una empresa que gobierne su desarrollo y decida a quién permitir usarla y bajo qué condiciones. No hay que pagar regalías, derechos de uso. No se está bajo la amenaza de que el fabricante pueda iniciar un juicio por patentes ante la difusión de un producto basado en la arquitectura.

Esto es algo poco común en la industria de las computadoras e incluso en la academia. Han surgido algunos proyectos o diseños que en un principio eran privativos y luego se liberaron o estandarizaron, pero ninguno había tomado tanto empuje hasta hace unos años. Si se contrasta con la popularidad y el desarrollo que alcanzó en contrapartida el software libre, todavía hay mucho por recorrer en el terreno del hardware libre. RISC-V puede considerarse un paso importante en este sentido.

1.2. ¿Qué es una arquitectura RISC?

RISC son las siglas de *Reduced Instruction Set Computer*. Se trata de un término acuñado en la década de 1980 en torno a desarrollos determinantes que se dieron en EEUU en cuanto al diseño de procesadores.

Las ideas de RISC giran en torno a dos objetivos que van de la mano y se complementan:

1. Sacar el máximo provecho a un "pipeline" de instrucciones.
2. Simplificar el diseño de procesadores.

1.3. ¿Por qué desarrollar una nueva arquitectura?

De acuerdo a lo relatado por los autores de RISC-V, de acuerdo a su experiencia en investigación y enseñanza usando arquitecturas comerciales, identificaron las siguientes desventajas:

- Las arquitecturas comerciales son privativas.
- Las arquitecturas comerciales solo son populares en ciertos dominios de mercado.
- Las arquitecturas comerciales vienen y van.
- Las arquitecturas comerciales populares son complejas.
- Las arquitecturas comerciales por sí solas no son suficientes para levantar aplicaciones.
- Las arquitecturas comerciales populares no se diseñaron para ser extensibles.
- Una arquitectura comercial modificada es una arquitectura nueva.

1.4. Breve historia de RISC-V

Empezó en 2010 como un proyecto académico en UCB, la Universidad de California en Berkeley (EEUU). David Patterson, famoso científico de la computación, quien estuvo en la década de 1980 a la cabeza del proyecto RISC original y quien publicó importantes libros sobre arquitectura de computadoras, fue quien lideró esta nueva iniciativa. Se trata de un sucesor espiritual de aquel RISC, y se lo identifica como la quinta generación, por eso el nombre RISC-V.

En 2011 se publica la versión 1.0 de la especificación de ISA de usuario de RISC-V, también conocida como volumen 1 de la especificación. El ISA privilegiado quedaría para un futuro volumen 2.

En 2014 se publica la versión 2.0 del volumen 1. Esta es la versión final congelada.

En 2015 se arma la Fundación RISC-V con más de 100 miembros. También se organiza la primera conferencia del proyecto, en EEUU.

En 2016, SiFive, empresa estadounidense, lanza el primer chip comercial de RISC-V: el microcontrolador Freedom E310, con un procesador a 320 MHz.

En 2017, SiFive lanza el primer chip RISC-V capaz de arrancar Linux y FreeBSD: U54-MC Coreplex.

En 2018 SiFive lanza la placa de desarrollo HiFive Unleashed, que contiene el chip anterior. También se revelan planes de Western Digital y NVIDIA para adoptar RISC-V. La empresa india Shakti lanza un procesador a 400 MHz capaz de arrancar Linux.

En 2019, la Fundación Linux lanza la Alianza CHIPS, un grupo industrial para fomentar el desarrollo de hardware abierto. Este año también se publica el volumen 2 de la especificación de RISC-V, el ISA privilegiado.

A lo largo de los últimos años, también se han ido adaptando los principales componentes de los sistemas GNU/Linux: el núcleo Linux, el compilador GCC, las herramientas Binutils, la biblioteca Glibc y el gestor de arranque GRUB ya son compatibles con la arquitectura.

1.5. ¿Por qué enseñar RISC-V?

Dos motivos: uno, que tiene una simplicidad tal que lo vuelve idóneo para aprender los conceptos de arquitecturas de computadoras; el otro, que es abierto y promueve el espíritu científico que se busca cultivar en esta carrera.

1.6. ¿Quién usa RISC-V en la industria?

En 2015 se fundó la Fundación RISC-V. De ella forman parte, de acuerdo a su sitio web, 275 organizaciones. Algunos nombres conocidos son: Google, NVIDIA, NXP, Qualcomm, Samsung, Western Digital, Allwinner Technology, Boston University, ETH Zurich, Hewlett Packard Enterprise, Hitachi, Huawei, IBM, Inria, MediaTek, Nokia, Princeton University, Raspberry Pi, Seagate, Siemens, Sony, STMicroelectronics y TSMC.

Se pueden destacar como casos concretos de adopción de RISC-V los siguientes:

Western Digital Lleva desde 2017 transicionando a RISC-V sus productos de procesamiento de datos de alto rendimiento. Desde 2018 tienen desarrollada su propia CPU RISC-V de 32 bits para dispositivos embebidos, SweRV Core, la cual fue publicada como código abierto.

NVIDIA Anunció en 2016 que desarrollaría un controlador RISC-V de 32 bits como sucesor del controlador Falcon que usan en sus GPU desde 2006. En 2019, se informó que también migrarían algunas de sus líneas de productos SoC a RISC-V.

Rambus Lanzó en 2018 núcleos de procesamiento seguro con RISC-V.

Otros Otros fabricantes vienen desarrollando desde aceleradores de aprendizaje profundo hasta controladores SSD, así como placas de desarrollo capaces de arrancar Linux.

Hay varias empresas que fabrican tanto CPU RISC-V como sistemas en chip (SoC) enteros que contienen varios módulos de los cuales uno es una CPU RISC-V. La más famosa es SiFive. También están lowRISC y otras...

https://riscv.org/risc-v-cores/

Hay muchas empresas que patrocinan RISC-V.

1.7. ¿Cómo probar RISC-V?

1.7.1. Software

Compilador cruzado

Para compilar, se puede usar GCC. Dado que se lo va a ejecutar en una máquina con otra arquitectura (como puede ser x86_64), es necesario lo que se llama un *compilador cruzado*: una versión de GCC compilada para generar código para una arquitectura distinta que aquella sobre la que corra.

Ubuntu ya trae tal versión en sus repositorios, basta con instalar el paquete `gcc-riscv64-linux-gnu`. Una vez instalado, se lo usa con el comando `riscv64-linux-gnu-gcc`. Los argumentos básicos son los mismos; en algunos casos, resulta conveniente agregar `-static` para generar un binario estático, sin enlaces a bibliotecas dinámicas que pueden no estar instaladas para la arquitectura destino. Por ejemplo:

```
$ riscv64-linux-gnu-gcc -static -o prog prog.c
```

Emulador/Simulador

Para ejecutar programas compilados para RISC-V, hay diversos emuladores y simuladores. QEMU es un emulador muy popular en GNU/Linux para multitud de arquitecturas y que desde hace unos años permite emular RISC-V. Una vez que se tiene un binario del programa a ejecutar, se puede correr `qemu-riscv64` de forma sencilla:

```
$ qemu-riscv64 prog
```

En Ubuntu, a partir de su versión 16.04 LTS Xenial Xerus, está disponible el paquete `qemu-system-misc`. Pero esto solo incluye el emulador de sistema.

Alternativamente, se pueden usar simuladores como rv8 y Spike.

1.7.2. Hardware

El ecosistema de dispositivos RISC-V está en sus primeras etapas. Se espera que surjan muchos productos en los próximos años. Por lo pronto, para desarrollos de propósito general, son de destacar las placas que ofrece SiFive:

HiFive1 Lanzada en 2016, primer producto comercial con un procesador RISC-V, de 32 bits. Compatible con Arduino.

HiFive1 Rev B Lanzada en 2019, versión mejorada de la anterior. Con más puertos, Wi-Fi y Bluetooth, entre otros cambios.

HiFive Unleashed Lanzada en 2018, primera placa de desarrollo que corre Linux. Incluye un SoC de 64 bits desarrollado por SiFive, 8 GiB de RAM, Gigabit Ethernet y ranura para tarjeta microSD. A 2019, su precio es muy prohibitivo.

Capítulo 2

Arquitectura

En esta sección se abordan los principales detalles técnicos de la arquitectura.

2.1. Objetivos de RISC-V

- Apertura.
- Adaptabilidad a muy diversos casos de uso.

2.2. Módulos

RISC-V tiene un diseño modular. Hay un conjunto de instrucciones y registros base, y extensiones que agregan conjuntos de instrucciones adicionales. Cada una de las posibles bases, así como las extensiones, se llaman módulos.

Para referirse a cierto conjunto de módulos, se usa una nomenclatura estandarizada. Esta tiene el formato $RV\langle bits\rangle\langle base\rangle[\langle exts\rangle]$, donde $\langle bits\rangle$ es el tamaño de palabra de la arquitectura (32, 64 o 128 bits), $\langle base\rangle$ es una letra que indica el perfil básico (I o E) y $\langle exts\rangle$, que es opcional, es una secuencia de letras que indica las extensiones. Cada extensión lleva una letra mayúscula, opcionalmente seguida de letras minúsculas. G es un caso especial, es una abreviatura de la secuencia de extensiones más común: MAFDZicsrZifencei. Ejemplos de cadenas completas: RV32I, RV64IGB, RV32ECZtso.

A continuación se listan los módulos oficiales.

Base	Descripción	Estado
RVWMO	–	ratificado
RV32I	base, enteros de 32 bits	ratificado
RV32E	base, enteros de 32 bits, para embebidos	borrador
RV64I	base, enteros de 64 bits	ratificado
RV128I	base, enteros de 128 bits	borrador
Extensión	**Descripción**	**Estado**
Zicsr	–	ratificado
Zifencei	–	ratificado
M	extensión para multiplicación y división entera	ratificado
A	extensión para instrucciones atómica	congelado
F	extensión para punto flotante de precisión simple	ratificado
D	extensión para punto flotante de precisión doble	ratificado
G	abreviatura para las seis extensiones anteriores	–
Q	extensión para punto flotante de precisión cuádruple	ratificado
L	extensión para punto flotante decimal	borrador
C	extensión para instrucciones comprimidas	ratificado
B	extensión para manipulación de bits	borrador
J	extensión para lenguajes traducidos dinámicamente	borrador
T	extensión para memoria transaccional	borrador
P	extensión para instrucciones de SIMD empaquetado	borrador
V	extensión para operaciones vectoriales	borrador
N	extensión para interrupciones de nivel de usuario	borrador
Ztso	–	congelado
Zam	–	borrador
Counters	–	borrador

¿Cuál estudiaremos nosotros? RV64IMFD.

2.2.1. Distintas bases

Como muestra la tabla anterior, hay 4 bases disponibles en RISC-V: RV32I, RV32E, RV64I y RV128I. ¿Qué diferencias hay entre ellas?

La primera diferencia es el tamaño de los registros enteros y las direcciones de memoria: 32 bits en el caso de RV32I y RV32E, 64 bits en RV64I y la poco común cifra de 128 bits en RV128I.

Las bases con el módulo I cuentan con 32 registros enteros, que se presentarán en la sección siguiente. Lo distintivo de E, por su parte, es que solo incluye los primeros 16 registros.

2.2.2. RV32E: para embebidos

La base RV32E difiere de RV32I en exactamente una cosa: la cantidad de registros se reduce de 32 a 16. Todas las instrucciones son las mismas.

¿Cuál es la importancia de esta base? Los registros resultan ser una buena parte del circuito de una CPU. Muchos sistemas embebidos son lo suficientemente pequeños como para que no se justifique el costo de tener 32 registros en el procesador. Al reducir la cantidad a la mitad, se logra una reducción importante en el tamaño del chip y todo lo que eso acarrea: consumo de energía, generación de calor y precio.

2.2.3. RV128I: planificación a futuro

El manual ofrece la siguiente cita:

> Hay solo un error que se puede cometer en diseño de computadoras del cual es difícil de recuperarse – no tener suficientes bits de direcciones para direccionar y gestionar memoria.
>
> – Bell and Strecker, ISCA-3, 1976.

Y luego presenta la motivación para la confección de esta base de la siguiente manera:

> La razón principal para extender el ancho de registros enteros es soportar espacios de direcciones más grandes. No está claro cuándo se requerirá un espacio de direcciones plano de más de 64 bits. Al momento de escribir esto, la supercomputadora más rápida del mundo de acuerdo a lo medido por el benchmark Top500 tenía más de 1 PB de DRAM, y requeriría más de 50 bits de espacio de direcciones si toda la DRAM residiera en un único espacio de direcciones. Algunas memorias a escala *warehouse* y tecnologías de interconexión rápida podrían impulsar una demanda de espacios de memoria incluso más grandes. La investigación de sistemas de exaescala está apuntando a sistemas de 100 PB de memoria, los cuales ocupan 57 bits de espacio de direcciones. A las tasas de crecimiento históricas, es posible que se requieran espacios de direcciones de más de 64 bits antes de 2030.
>
> La historia sugiere que en cuanto quede claro que se necesitarán más de 64 bits de espacio de direcciones, los arquitectos repetirán intensivos debates sobre alternativas a extender el espacio de direcciones, incluyendo segmentación, espacios de direcciones de 96 bits y paliativos por software, hasta que, finalmente, se adopten

espacios de direcciones de 128 bits como la solución más simple y mejor.

2.3. Registros

RISC-V, en sus módulos base I (RV32I, RV64I y RV128I), cuenta con 32 registros enteros de propósito general. Todos ellos tienen el mismo tamaño, que puede ser 32, 64 o 128 bits dependiendo de la variante de la arquitectura que se use.

De forma resumida, los registros son los siguientes:

`zero:` siempre cero.

`ra:` dirección de retorno.

`sp:` puntero de pila.

`gp:` puntero global.

`tp:` puntero de hilo.

`t0`–`t6:` temporales.

`s0`–`s11:` preservados (puntero de marco en el primero).

`a0`–`a7:` argumentos (valor de retorno en los dos primeros).

Estos nombres indican el uso que se da a cada registro, usos que están determinados en su mayor parte por la convención de llamada, con la excepción de `zero`, que está diseñado por hardware para que siempre contenga cero. Adicionalmente, los registros también van numerados de 0 a 31 y se los puede nombrar usando el número asociado más el prefijo `x`. A nivel técnico es lo mismo usar una nomenclatura o la otra; la primera tiene una evidente ventaja en claridad, mientras que la segunda es independiente de la convención de llamada; nosotros usaremos siempre la primera: los nombres funcionales.

La siguiente tabla indica todos los detalles sobre cada registro:

Nombre	Nombre funcional	Función	Tipo de uso
x0	zero	Siempre cero	–
x1	ra	Dirección de retorno	temporal
x2	sp	Puntero de pila	preservado
x3	gp	Puntero global	–
x4	tp	Puntero de hilo	–
x5	t0	Temporal / Dirección de retorno alternativa	temporal
x6–x7	t1–t2	Temporal	llamante
x8	s0/fp	Preservado / Puntero de marco	preservado
x9	s1	Preservado	preservado
x10–x11	a0–a1	Argumento de función / Valor de retorno	temporal
x12–x17	a2–a7	Argumento de función	temporal
x18–x27	s2–s11	Preservado	preservado
x28–x31	t3–t6	Temporal	temporal

En cuanto al tipo de uso, en la tabla se emplean los nombres temporal y preservado. Con los primeros se refiere a los registros preservados por la función llamante ("caller saved") y los segundos, por la llamada ("callee saved"). Véase la sección *Convención de llamada* para más información.

¿Por qué no son contiguas las asignaciones de registros temporales y preservados? Ciertos detalles de RISC-V llevan a cuestionarse por qué no se optó por un ordenamiento más simple. Más aun siendo que no tiene retrocompatibilidad que mantener, como pasa en otras arquitecturas. La respuesta está en la adaptabilidad de RISC-V: como se indicó en una sección anterior, hay una variante E de la arquitectura que, en contraste con la I, incluye solamente los primeros 16 registros. Al reducir la cantidad de registros, es deseable en lo posible mantener las distinciones de finalidad (temporales, preservados, etc.). Entonces hay que distribuir los 32 registros de forma que en la primera mitad haya de todos los tipos en alguna proporción, y que la segunda mitad amplíe esas proporciones.

2.4. Endianness

RISC-V es siempre little-endian. Es decir, los bytes menos significativos se codifican en memoria en las direcciones más bajas. Por ejemplo, el entero 0x12345678, de 4 bytes, se codifica así:

...	0x78	0x56	0x34	0x12	...

2.5. Lenguaje de ensamblador

Como toda arquitectura, RISC-V tiene su propio lenguaje de ensamblador. Un programa llamado ensamblador lee archivos en este lenguaje y genera código máquina. Los archivos llevan la extensión `.s` o `.S`. La segunda, con mayúscula, indica que se debe pasar el preprocesador de C antes de ensamblar: esto permite incluir directivas como `#include` y `#define`.

2.5.1. Sintaxis general

La sintaxis de ensamblador para RISC-V se caracteriza por ser limpia, en el sentido de que prescinde de adornos allá donde sea posible. Que el conjunto de instrucciones sea simple y ortogonal facilita llevar a cabo tal diseño.

Véanse algunos ejemplos. Esta es la implementación de una función que suma dos números enteros y devuelve el resultado. Se incluyen comentarios que describen cada instrucción:

```
            .text
            .global sumar
sumar:
            add a0, a0, a1    # a0 := a0 + a1
            ret               # Retorno de la función
            .global main
main:
            addi sp, sp, -16  # Se reserva espacio en la pila
            sd   ra, (sp)     # Se guarda dirección de retorno
            li   a0, 4        # Primer argumento para ‘sumar’
            li   a1, 5        # Segundo argumento para ‘sumar’
            call sumar        # Se llama a ‘sumar’
            ld   ra, (sp)     # Se restaura dirección de retorno
            addi sp, sp, 16   # Se libera espacio en la pila
            ret
```

En la variante que sigue, la función toma un arreglo de enteros de 4 bytes como argumento y retorna la suma de los primeros dos elementos:

```
            .rodata
valores:    .word 3
            .word 4
```

```
            .text
            .global sumar
sumar:
            lw  t0, (a0)
            lw  t1, 4(a0)
            add a0, t0, t1
            jr  ra
            .global main
main:
            addi sp, sp, -16
            sd   ra, (sp)
            la   a0, valores  # Argumento para ‘sumar‘
            call sumar
            ld   ra, (sp)
            addi sp, sp, 16
            jr ra
```

Características:

- Los operandos de destino van a la izquierda, los de fuente a la derecha.
- Las instrucciones de operación usan operandos separados para destino y fuente. Lo más común es el formato de 3 operandos: destino, fuente, fuente.
- Las instrucciones no llevan adornos, como pueden ser sufijos de tamaño. Toda la información requerida está indicada en el nombre de la instrucción misma.
- Los registros no llevan adornos, es decir que no se usa un prefijo para distinguirlos, basta con su nombre.
- Los literales de enteros no llevan adornos, es decir que no se usa un prefijo para distinguirlos, basta con indicar el número.
- Para direccionar memoria a partir de registros, hay una única sintaxis: `offset(reg)` donde `offset` es un entero opcional con signo y `reg` es un registro. Se suma `offset` al valor contenido en `reg`.

2.5.2. Directivas del ensamblador

Son las líneas que empiezan con punto. No se trata de instrucciones, sino indicaciones que se le dan al programa ensamblador para que genere el binario

ejecutable de cierta forma.

Directivas para emisión de datos y control de alineación:

`.byte N...`
: Emitir un byte.

`.half N...`
: Emitir media palabra (2 bytes), con alineación natural.

`.word N...`
: Emitir una palabra (4 bytes), con alineación natural.

`.dword N...`
: Emitir doble palabra (8 bytes), con alineación natural.

`.2byte N...`
: Emitir valor de 2 bytes sin alinear.

`.4byte N...`
: Emitir valor de 4 bytes sin alinear.

`.8byte N...`
: Emitir valor de 8 bytes sin alinear.

`.string C`
: Emitir cadena de caracteres, terminada en caracter nulo.

`.asciz C`
: Alias para `.string`.

`.zero N`
: Emitir una cantidad de ceros.

`.align N`
: Alinear a potencia de 2.

Directivas para control de símbolos y secciones:

`.globl S`
: Emitir símbolo global.

`.equ S, V`
: Definir constante. Ejemplo:

```
.equ MAGIC_NUMBER, 0x40302040
```

`.text`
: Emitir sección de código y hacerla la actual.

`.data`
: Emitir sección de datos inicializados y hacerla la actual.

`.rodata`
: Emitir sección de datos inicializados de solo lectura y hacerla la actual.

`.bss`
: Emitir sección de datos no inicializados y hacerla la actual.

`.section X`
: Emitir la sección indicada y hacerla la actual.

Directivas varias:

`.option O...`
: Opciones específicas de RISC-V.

`.macro A...`
: Empezar definición de macro.

`.endm`
: Terminar definición de macro.

2.5.3. Instrucciones

Las instrucciones son la primera palabra de una línea, descartando la posible etiqueta, y no empiezan con punto. El conjunto de instrucciones disponible variará dependiendo de la variante de RISC-V que se utilice. Como mínimo, sin embargo, las instrucciones básicas, definidas por el módulo I o E, estarán disponibles.

La sección *Instrucciones básicas* de este capítulo describe cada instrucción del módulo I. Capítulos siguientes describen las de los módulos M, F y D.

2.5.4. Etiquetas

Una etiqueta es un nombre legible define una posición en alguna sección del programa; posteriormente será traducido a una dirección de memoria. Se define una etiqueta indicándola inmediatamente antes de la instrucción o directiva cuya posición va a indicar, con su un nombre y el caracter dos puntos como sufijo. Puede ir en la misma línea o en alguna anterior.

2.6. Instrucciones básicas

Solo veremos las instrucciones de nivel de usuario, no las privilegiadas. Hay un estándar separado para cada una: el volumen I y el volumen II del manual de referencia, respectivamente.

Antes que nada, es conveniente tener en cuenta los distintos tipos de operando que puede tomar una instrucción. Son los siguientes:

Registro El nombre de un registro. Indica que se va a operar con el valor contenido en este.

Inmediato Un entero indicado como constante en la instrucción misma; puede ser un literal, por ejemplo el número -12, o puede ser una etiqueta. Se opera directamente con ese valor literal en el primer caso, o con la dirección de memoria que se vaya a asociar a una etiqueta, en el segundo.

Dirección de memoria Un operando con la sintaxis de direccionamiento de memoria (`offset(reg)`). Se obtiene el valor del registro `reg`, se le suma `offset` y el resultado se usa como operando.

Algo fundamental de las arquitecturas RISC es que cada función toma ciertos tipos específicos de operando. No es posible pasar una dirección de memoria donde se espera un registro: para cada caso hay que usar una instrucción distinta.

Las mayoría de instrucciones aritméticas y lógicas llevan el formato de 3 registros: uno de salida, dos de entrada. Estos se notarán con los nombres *rd*, *rs1* y *rs2*. Aquellas que para la segunda fuente tomen en cambio un inmediato, lo notan con *imm*.

2.6.1. Operaciones aritméticas

`add rd, rs1, rs2` (*add*)
: Sumar registros.

`addi rd, rs1, imm` (*add immediate*)
: Sumar un registro y un inmediato (de 12 bits).

`sub rd, rs1, rs2` (*substract*)
: Restar registros.

`lui rd, imm` (*load upper immediate*)
: Cargar inmediato (de 20 bits) en la parte superior de un registro.

`auipc rd, imm` (*add upper immediate to PC*)
: Sumar inmediato (de 20 bits) a la superior del contador del programa (PC) y guardar resultado en registro.

En RV64I:

`addw rd, rs1, rs2` (*add as word*)
: Como `add` pero opera sobre palabras (4 bytes).

`addiw rd, rs1, imm` (*add immediate as word*)
: Como `addi` pero opera sobre palabras (4 bytes).

`subw rd, rs1, imm` (*substract as word*)
: Como `sub` pero opera sobre palabras (4 bytes).

2.6.2. Operaciones lógicas

`xor rd, rs1, rs2`
: Calcular *xor* (disyunción exclusiva) sobre registros.

`xori rd, rs1, imm`
: Calcular *xor* (disyunción exclusiva) sobre un registro y un inmediato.

`or rd, rs1, rs2`
: Calcular *or* (disyunción inclusiva) sobre registros.

`ori rd, rs1, imm`
: disyunción inclusiva de inmediato.

`and rd, rs1, rs2`
: Calcular *and* (conjunción) sobre registros.

`andi rd, rs1, imm`
: Calcular *and* (conjunción) sobre un registro y un inmediato (12 bits).

2.6.3. Operaciones de corrimiento

`sll rd, rs1, rs2` (*shift left logical*)
: Calcular corrimiento lógico a izquierda sobre registros.

`slli rd, rs1, imm` (*shift left logical immediate*)
: Calcular corrimiento lógico a izquierda sobre un registro y un inmediato.

`srl rd, rs1, rs2` (*shift right logical*)
Calcular corrimiento lógico a derecha.

`srli rd, rs1, imm` (*shift right logical immediate*)
Calcular corrimiento lógico a derecha sobre un registro y un inmediato.

`sra rd, rs1, rs2` (*shift right arithmetic*)
Calcular corrimiento aritmético a derecha sobre registros.

`srai rd, rs1, imm` (*shift right arithmetic immediate*)
Calcular corrimiento aritmético a derecha sobre un registro y un inmediato.

En RV64I:

`sllw rd, rs1, rs2` (*shift left logical as word*)
Como `sll` pero opera sobre palabras (4 bytes).

`slliw rd, rs1, imm` (*shift left logical immediate as word*)
Como `slli` pero opera sobre palabras (4 bytes).

`srlw rd, rs1, rs2` (*shift right logical as word*)
Como `srl` pero opera sobre palabras (4 bytes).

`srliw rd, rs1, imm` (*shift right logical immediate as word*)
Como `srli` pero opera sobre palabras (4 bytes).

`sraw rd, rs1, rs2` (*shift right arithmetic as word*)
Como `sra` pero opera sobre palabras (4 bytes).

`sraiw rd, rs1, imm` (*shift right arithmetic as word*)
Como `srai` pero opera sobre palabras (4 bytes).

2.6.4. Carga/Descarga

`lb rd, mem` (*load byte*)
Copiar byte de memoria a registro.

`lbu rd, mem` (*load byte unsigned*)
Copiar byte de memoria a registro, extender con ceros.

`lh rd, mem` (*load halfword*)
Copiar media palabra (2 bytes) de memoria a registro, extender con signo.

`lhu rd, mem` (*load halfword unsigned*)
: Copiar media palabra (2 bytes) de memoria a registro, extender con ceros.

`lw rd, mem` (*load word*)
: Copiar palabra (4 bytes) de memoria a registro, extender con signo si es necesario.

`sb rs1, mem` (*store byte*)
: Copiar byte de registro a memoria.

`sh rs1, mem` (*store halfword*)
: Copiar media palabra (2 bytes) de registro a memoria.

`sw rs1, mem` (*store word*)
: Copiar palabra (4 bytes) de registro a memoria.

En RV64I:

`lwu rd, mem` (*load word unsigned*)
: Copiar palabra (4 bytes) de memoria a registro, extender con ceros.

`ld rd, mem` (*load doubleword*)
: Copiar doble palabra (8 bytes) de memoria a registro.

`sd rs1, mem` (*load doubleword*)
: Copiar doble palabra (8 bytes) de memoria a registro.

2.6.5. Saltos

`jal rd, imm` (*jump and link*)
: Saltar a dirección inmediata y guardar dirección de retorno en registro.

`jalr rd, rs1, imm` (*jump and link register*)
: Saltar a dirección de memoria con registro y offset y guardar dirección de retorno en registro.

`beq rs1, rs2, imm` (*branch if equal*)
: Saltar si es igual.

`bne rs1, rs2, imm` (*branch if not equal*)
: Saltar si es distinto.

`blt rs1, rs2, imm` (*branch if less than*)
: Saltar si es menor, con signo.

`bltu rs1, rs2, imm` (*branch if less than, unsigned*)
: Saltar si es menor, sin signo.

`bge rs1, rs2, imm` (*branch if greater than or equal*)
: Saltar si es mayor o igual, con signo.

`bgeu rs1, rs2, imm` (*branch if greater than or equal, unsigned*)
: Saltar si es mayor o igual, con signo.

2.6.6. Otras

`slt rd, rs1, rs2` (*set if less than*)
: Comparar dos registros por menor, con signo, y guardar resultado (0 o 1) en registro.

`slti rd, rs1, imm` (*set if less than immediate*)
: Comparar un registro y un inmediato por menor, con signo, y guardar resultado (0 o 1) en registro.

`sltu rd, rs1, rs2` (*set if less than, unsigned*)
: Comparar dos registros por menor, sin signo, y guardar resultado (0 o 1) en registro.

`sltiu rd, rs1, imm` (*set if less than immediate, unsigned*)
: Comparar un registro y un inmediato por menor, sin signo, y guardar resultado (0 o 1) en registro.

`fence ...`
: Barrera para sincronizar hilos. Ordena accesos a memoria y dispositivos. *No la usaremos.*

`ecall` (*environment call*)
: Llamada al entorno de ejecución (llamada a sistema). *No la usaremos.*

`ebreak` (*environment break*)
: Interrupción al entorno de ejecución (punto de ruptura del depurador). *No la usaremos.*

2.7. Pila

Como en todas las arquitecturas de CPU modernas, en RISC-V cada proceso tiene una pila: un segmento dinámico de la memoria principal donde se van guardando los marcos de activación de cada llamada a función.

La pila crece hacia abajo: los nuevos elementos se van agregando en direcciones de memoria más bajas que las anteriores. El registro `sp` se llama puntero de pila ("stack pointer") y es el que determina el tope de la pila: contiene la dirección del último elemento agregado.

Al ser de tipo RISC, esta arquitectura no ofrece instrucciones específicas para manipular la pila. En cambio, se usan la suma y resta para cambiar el tope, alterando el registro `sp`, y las instrucciones generales de carga y descarga para leer y escribir en la memoria.

Opcionalmente, uno puede usar, además de `sp`, un puntero de marco ("frame pointer"): un registro que apunte al principio del marco de activación actual. La convención es usar para esto el registro `s0`, al cual también se lo llama `fp`.

2.8. Convención de llamada

- Los primeros 8 argumentos enteros van en registros: `a0`..`a7`.
- Los primeros 8 argumentos flotantes van en registros: `fa0`..`fa7`.
- Los argumentos adicionales van en la pila.
- Las estructuras se desmembran y sus partes se colocan en registros o pila de acuerdo a sus tipos y cantidades, a grandes rasgos como si fueran varios argumentos.
- El valor de retorno entero va en el registro `a0` y, de ser necesario más tamaño, en el par `a0`, `a1`.
- El valor de retorno flotante va en el registro `fa0` y, de ser necesario más tamaño, en el par `fa0`, `fa1`.
- Registros temporales enteros: `ra`, `a0`..`a7`, `t0`..`t6`.
- Registros temporales flotantes: `fa0`..`fa7`, `ft0`..`ft11`.
- Registros preservados enteros: `sp`, `s0`..`s11`.
- Registros preservados flotantes: `fs0`..`fs11`.
- El puntero de pila siempre va alineado a 16 bytes.

2.9. Codificación

Las instrucciones que uno escribe textualmente en lenguaje de ensamblador deben luego traducirse a código máquina, ejecutable por una CPU. ¿Cómo es la codificación que adquieren en código máquina?

Por defecto, todas las instrucciones en RISC-V se codifican en 32 bits. Dentro de estos 32 bits, se distribuyen los distintos datos que debe indicar cada instrucción. De acuerdo a cuáles sean estos datos, las instrucciones adoptan unos formatos u otros; hay 6 formatos distintos para las instrucciones base, cada una emplea uno de estos.

La extensión C agrega codificaciones alternativas de 16 bits, lo cual reduce el tamaño de los programas y es importante en sistemas embebidos donde el espacio escasea. Por otro lado, la arquitectura incorpora previsiones para el posible agregado de instrucciones de más de 32 bits en un futuro.

A continuación se presenta cada formato base de 32 bits.

2.9.1. Tipo R: registro/registro

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
funct7	rs2	rs1	funct3	rd	opcode

2.9.2. Tipo I: inmediato

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode

2.9.3. Tipo U: inmediato superior

31 – 12	11 – 7	6 – 0
imm[31:12]	rd	opcode

2.9.4. Tipo S: almacenamiento

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

2.9.5. Tipo B: ramificación

31	30 – 25	24 – 20	19 – 15	14 – 12	11 – 8	7	6 – 0
[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	[11]	opcode

2.9.6. Tipo J: salto

31	30 – 21	20	19 – 12	11 – 7	6 – 0
[20]	imm[10:1]	[11]	imm[19:12]	rd	opcode

Capítulo 3

Extensión M: multiplicación y división

La extensión M agrega instrucciones de multiplicación y división de enteros. Emplea los mismos registros enteros de I.

3.1. ¿Por qué en una extensión?

¿Por qué las operaciones de multiplicación y división vienen en una extensión? ¿No son lo suficientemente básicas como para merecer ser incluidas en el conjunto básico, I, de la misma forma que la suma y la resta?

Resulta que a nivel de hardware, implementar estas operaciones no es tan simple como implementar suma y resta. En procesadores embebidos, ciertos diseños limitados podrían directamente prescindir de las mismas: se trata de una decisión de diseño del procesador que reduce directamente la cantidad de transistores, el consumo de energía y la generación de calor. En cuanto al software, hay casos en que se sabe de antemano que no necesitarán multiplicar y dividir; pero si no es así, siempre se puede relegar estas operaciones a implementaciones por software (es decir, escribir funciones con los algoritmos de multiplicación y división en base a la suma y la resta), que serán más lentas que instrucciones nativas pero funcionarán.

En resumen, son caras y no siempre hacen falta.

Así lo justifica el manual de la arquitectura:

> Separamos la multiplicación y la división enteras de lo que es la base para simplificar implementaciones de baja gama, o aplicaciones donde las operaciones de multiplicación y división enteras o bien sean infrecuentes o bien se manejen mejor en aceleradores añadidos.

3.2. Instrucciones

`mul rd, rs1, rs2` (*multiply*)
Multiplicar.

`mulh rd, rs1, rs2` (*multiply upper half*)
Multiplicar mitad superior.

`mulhsu rd, rs1, rs2` (*multiply upper half, signed × unsigned*)
Multiplicar mitad superior, con signo × sin signo.

`mulhu rd, rs1, rs2` (*multiply upper half, unsigned*)
Multiplicar mitad superior, sin signo.

`div rd, rs1, rs2` (*divide*)
Dividir y obtener cociente.

`divu rd, rs1, rs2` (*divide unsigned*)
Dividir sin signo y obtener cociente.

`rem rd, rs1, rs2` (*remainder*)
Dividir y obtener resto.

`remu rd, rs1, rs2` (*remainder unsigned*)
Dividir sin signo y obtener resto.

En RV64M:

`mulw rd, rs1, rs2` (*multiply as word*)
Como `mul` pero opera sobre palabras (4 bytes).

`divw rd, rs1, rs2` (*divide as word*)
Como `mul` pero opera sobre palabras (4 bytes).

`divuw rd, rs1, rs2` (*divide unsigned as word*)
Como `mul` pero opera sobre palabras (4 bytes).

`remw rd, rs1, rs2` (*remainder as word*)
Como `mul` pero opera sobre palabras (4 bytes).

`remuw rd, rs1, rs2` (*remainder unsigned as word*)
Como `mul` pero opera sobre palabras (4 bytes).

3.3. División por cero y desborde

En algunas arquitecturas, la división por cero genera una excepción, se interrumpe el flujo de ejecución para pasar a ejecutar código dedicado al tratamiento de esta situación anormal. Lo mismo puede ocurrir con el desborde en la división con signo, cuando el mínimo negativo se multiplica por -1.

RISC-V, en cambio, no tiene instrucciones aritméticas que generen excepciones. La división no es la excepción. En lugar de esto, hay resultados predefinidos para cuando ocurre alguna de las circunstancias mencionadas.

En el caso de la división por cero, el cociente resulta en la secuencia de todos bits en 1 ($2^L - 1$ sin signo, -1 con signo), y el resto por su parte resulta en el valor del divisor.

En el caso del desborde en la divsiión con signo, el cociente resulta en -2^{L-1} y el resto en 0.

Capítulo 4

Extensiones F y D: punto flotante

Las extensiones F, D y Q de RISC-V incorporan manejo nativo de números en punto flotante. Agregan un conjunto propio de registros y diversas instrucciones. Son compatibles con el estándar IEEE 754.

Las tres extensiones se diferencian en el tamaño de flotante que permiten manejar: F ("float") habilita flotantes de precisión simple (32 bits), D ("double") los habilita precisión doble (64 bits) y Q ("quad"), de precisión cuádruple (128 bits).

En este material se presenta solo el contenido de las extensiones F y D. No tendremos en cuenta Q debido a que no está tan extendida, el GCC que usamos no la emplea por defecto y no aporta algo nuevo a nivel pedagógico: sus agregados son análogos a los de F y D.

4.1. Registros

Se agregan 33 registros: 32 de punto flotante (`f0` .. `f31`) y uno de estado y control (`fcsr`).

Los 32 registros flotantes solo pueden almacenar números en punto flotante, nunca enteros. Hay instrucciones para trasladar valores de un conjunto de registros al otro, ya sea convirtiendo el número en el proceso o dejando la secuencia de bits tal cual.

El registro `fcsr` no se accede de forma directa sino con instrucciones dedicadas. Consta de dos campos: uno indica el modo de redondeo y el otro, banderas de excepción acumuladas. Se puede leer y escribir tanto el registro entero como cada campo por separado.

El campo de redondeo contiene 3 bits y permite establecer el redondeo

dinámico. Véase la sección *Redondeo* para más información.

El campo de banderas contiene 5 bits, que indican las siguientes banderas:

Bit	Bandera	Significado
0	NX	Inexacto
1	UF	*Underflow*
2	OF	Desborde (*Overflow*)
3	DZ	División por cero
4	NV	Operación inválida

4.2. Redondeo

Se permiten distintos modos de redondeo. Hay dos maneras de seleccionarlos. Una es el redondeo estático, determinado por un campo contenido en cada instrucción que indica el modo que esta debe usar. Otra es redondeo dinámico, que lo determina el valor del campo correspondiente de `fcsr`.

En ambos casos, el modo se indica con 3 bits, que combinados permiten las siguientes opciones:

Secuencia	Nombre	Significado
000	RNE	Redondear al más cercano, desempata el par
001	RTZ	Redondear a cero
010	RDN	Redondear para abajo (hacia $-\infty$)
011	RUP	Redondear para arriba (hacia $+\infty$)
100	RMM	Redondear al más cercano, desempata el de mayor magnitud
101	–	*Reservado para uso futuro*
110	–	*Reservado para uso futuro*
111	DYN	En instrucción: selección dinámica; en *fcsr*: *inválido*

4.3. Instrucciones de F

4.3.1. Operaciones

`fadd.s rd, rs1, rs2`
: Sumar dos flotantes simples.

`fsub.s rd, rs1, rs2`
: Restar dos flotantes simples.

`fmul.s rd, rs1, rs2`
: Multiplicar dos flotantes simples.

`fdiv.s rd, rs1, rs2`
: Dividir dos flotantes simples.

`fsqrt.s rd, rs1`
: Calcular la raíz cuadrada de un flotante simple.

`fmin.s rd, rs1, rs2`
: Calcular el mínimo entre dos flotantes simples.

`fmax.s rd, rs1, rs2`
: Calcular el máximo entre dos flotantes simples.

`fmadd.s rd, rs1, rs2, rs3`
: Multiplicar dos flotantes simples y sumarles otro.
$rd := rs1 \times rs2 + rs3$

`fmsub.s rd, rs1, rs2, rs3`
: Multiplicar dos flotantes simples y restarles otro.
$rd := rs1 \times rs2 - rs3$

`fnmsub.s rd, rs1, rs2, rs3`
: Multiplicar dos flotantes simples, negar producto y sumarles otro.
$rd := -(rs1 \times rs2) + rs3$

`fnmadd.s rd, rs1, rs2, rs3`
: Multiplicar dos flotantes simples, negar producto y restarles otro.
$rd := -(rs1 \times rs2) - rs3$

`fsgnj.s rd, rs1, rs2`
: Inyectar signo en precisión simple. Tomar de $rs1$ todos los bits salvo el de signo, y de $rs2$ tomar el de signo.

`fsgnjn.s rd, rs1, rs2`
: Inyectar signo negado en precisión simple. Tomar de $rs1$ todos los bits salvo el de signo, y de $rs2$ tomar el de signo negado.

`fsgnjx.s rd, rs1, rs2`
: Inyectar signo con *xor* en precisión simple. Tomar de $rs1$ todos los bits salvo el de signo, y para el signo calcular el *xor* de los signos de $rs1$ y $rs2$.

4.3.2. Conversión

`fcvt.w.s rd, rs1`
: Convertir flotante simple a palabra entera (4 bytes) con signo. rd debe ser un registro entero y $rs1$ un registro flotante.

`fcvt.wu.s rd, rs1`
: Convertir flotante simple a palabra entera (4 bytes) sin signo. rd debe ser un registro entero y $rs1$ un registro flotante.

`fcvt.s.w rd, rs1`
: Convertir palabra entera (4 bytes) con signo a flotante simple. rd debe ser un registro flotante y $rs1$ un registro entero.

`fcvt.s.wu rd, rs1`
: Convertir palabra entera (4 bytes) sin signo a flotante simple. rd debe ser un registro flotante y $rs1$ un registro entero.

En RV64F:

`fcvt.l.s rd, rs1`
: Convertir flotante simple a doble palabra entera (8 bytes) con signo. rd debe ser un registro entero y $rs1$ un registro flotante.

`fcvt.lu.s rd, rs1`
: Convertir flotante simple a doble palabra entera (8 bytes) sin signo. rd debe ser un registro entero y $rs1$ un registro flotante.

`fcvt.s.l rd, rs1`
: Convertir doble palabra entera (8 bytes) con signo a flotante simple. rd debe ser un registro flotante y $rs1$ un registro entero.

`fcvt.s.lu rd, rs1`
: Convertir doble palabra entera (8 bytes) sin signo a flotante simple. rd debe ser un registro flotante y $rs1$ un registro entero.

4.3.3. Comparación y clasificación

`feq.s rd, rs1, rs2`
: Comparar flotantes simples por igualdad y guardar el resultado en un registro. Si cualquier operando fuente es NaN, el resultado es 0.

`flt.s rd, rs1, rs2`
: Comparar flotantes simples por menor y guardar el resultado en un registro. Si cualquier operando fuente es NaN, el resultado es 0. Comparar por menor.

`fle.s rd, rs1, rs2`
: Comparar flotantes simples por menor o igual y guardar el resultado en un registro. Si cualquier operando fuente es NaN, el resultado es 0.

`fclass.s rd, rs1`
: Clasificar flotante simple según la clase de valor. Asigna un entero que indica si es normalizado, desnormalizado, cero, infinito o NaN (véase la sección *Clasificación de números* para más información). rd debe ser un registro entero y $rs1$ un registro flotante.

4.3.4. Carga y descarga

`flw rd, mem`
: Copiar flotante simple de memoria a registro. rd debe ser un registro flotante.

`fsw rs1, mem`
: Copiar flotante simple de registro a memoria. $rs1$ debe ser un registro flotante.

`fmv.x.w rd, rs1`
: Copiar secuencia de bits representando un flotante simple en registro flotante a registro entero. Se copia tal cual, sin conversión. rd debe ser un registro entero y $rs1$ un registro flotante.

`fmv.w.x rd, rs1`
: Copiar secuencia de bits representando un flotante simple en registro entero a registro flotante. Se copia tal cual, sin conversión. rd debe ser un registro flotante y $rs1$ un registro entero.

4.3.5. Control y estado

`frcsr rd`
: Copiar `fcsr` en registro entero.

`fscsr rs1`
: Copiar registro entero en `fcsr`.

`frrm rd`
: Copiar campo RM (modo de redondeo) de `fcsr` en registro entero.

`fsrm rs1`
: Copiar registro entero en campo RM (modo de redondeo) de `fcsr`.

`frflags rd`
: Copiar campo de banderas acumuladas de `fcsr` en registro entero.

`fsflags rs1`
: Copiar registro entero en campo de banderas acumuladas de `fcsr`.

4.4. Instrucciones de D

4.4.1. Operaciones

`fadd.d rd, rs1, rs2`
: Sumar dos flotantes dobles.

`fsub.d rd, rs1, rs2`
: Restar dos flotantes dobles.

`fmul.d rd, rs1, rs2`
: Multiplicar dos flotantes dobles.

`fdiv.d rd, rs1, rs2`
: Dividir dos flotantes dobles y obtener cociente.

`fsqrt.d rd, rs1`
: Calcular la raíz cuadrada de un flotante doble.

`fmin.d rd, rs1, rs2`
: Obtener el mínimo entre dos flotantes dobles.

`fmax.d rd, rs1, rs2`
: Obtener el máximo entre dos flotantes dobles.

`fmadd.d rd, rs1, rs2, rs3`
: Multiplicar dos flotantes dobles y sumarles otro.
$rd := rs1 \times rs2 + rs3$

`fmsub.d rd, rs1, rs2, rs3`
: Multiplicar dos flotantes dobles y restarles otro.
$rd := rs1 \times rs2 - rs3$

`fnmsub.d rd, rs1, rs2, rs3`
: Multiplicar dos flotantes dobles, negar producto y sumarles otro.
$rd := -(rs1 \times rs2) + rs3$

`fnmadd.d rd, rs1, rs2, rs3`
: Multiplicar dos flotantes dobles, negar producto y restarles otro.
$rd := -(rs1 \times rs2) - rs3$

`fsgnj.d rd, rs1, rs2`
: Inyectar signo en precisión doble. Tomar de $rs1$ todos los bits salvo el de signo, y de $rs2$ tomar el de signo.

`fsgnjn.d rd, rs1, rs2`
: Inyectar signo negado en precisión doble. Tomar de $rs1$ todos los bits salvo el de signo, y de $rs2$ tomar el de signo negado.

`fsgnjx.d rd, rs1, rs2`
: Inyectar signo con *xor* en precisión doble. Tomar de $rs1$ todos los bits salvo el de signo, y para el signo calcular el *xor* de los signos de $rs1$ y $rs2$.

4.4.2. Conversión

`fcvt.w.d rd, rs1`
: Convertir flotante doble a palabra entera (4 bytes) con signo. rd debe ser un registro entero y $rs1$ un registro flotante.

`fcvt.wu.d rd, rs1`
: Convertir flotante doble a palabra entera (4 bytes) sin signo. rd debe ser un registro entero y $rs1$ un registro flotante.

`fcvt.d.w rd, rs1`
: Convertir palabra entera (4 bytes) con signo a flotante doble. rd debe ser un registro flotante y $rs1$ un registro entero.

`fcvt.d.wu rd, rs1`
: Convertir palabra entera (4 bytes) sin signo a flotante doble. rd debe ser un registro flotante y $rs1$ un registro entero.

`fcvt.s.d rd, rs1`
: Convertir flotante doble a flotante simple.

`fcvt.d.s rd, rs1`
: Convertir flotante simple a flotante doble.

En RV64D:

`fcvt.l.d rd, rs1`
: Convertir flotante doble a doble palabra entera (8 bytes) con signo. rd debe ser un registro entero y $rs1$ un registro flotante.

`fcvt.lu.d rd, rs1`
: Convertir flotante doble a doble palabra entera (8 bytes) sin signo. rd debe ser un registro entero y $rs1$ un registro flotante.

`fcvt.d.l rd, rs1`
: Convertir doble palabra entera (8 bytes) con signo a flotante doble. rd debe ser un registro flotante y $rs1$ un registro entero.

`fcvt.d.lu rd, rs1`
: Convertir doble palabra entera (8 bytes) sin signo a flotante doble. rd debe ser un registro flotante y $rs1$ un registro entero.

4.4.3. Comparación y clasificación

`feq.d rd, rs1, rs2`
: Comparar flotantes dobles por igualdad y guardar el resultado en un registro. Si cualquier operando fuente es NaN, el resultado es 0.

`flt.d rd, rs1, rs2`
: Comparar flotantes dobles por menor y guardar el resultado en un registro. Si cualquier operando fuente es NaN, el resultado es 0.

`fle.d rd, rs1, rs2`
: Comparar flotantes dobles por menor o igual y guardar el resultado en un registro. Si cualquier operando fuente es NaN, el resultado es 0.

`fclass.d rd, rs1`
: Clasificar flotante doble según la clase de valor. Asigna un entero que indica si es normalizado, desnormalizado, cero, infinito o NaN (véase la sección *Clasificación de números* para más información). rd debe ser un registro entero y $rs1$ un registro flotante.

4.4.4. Carga y descarga

`fld rd, mem`
: Copiar flotante doble de memoria a registro. rd debe ser un registro flotante.

`fsd rs1, mem`
: Copiar flotante doble de registro a memoria. $rs1$ debe ser un registro flotante.

En RV64D:

`fmv.x.d rd, rs1`
: Copiar secuencia de bits representando un flotante doble en registro flotante a registro entero. Se copia tal cual, sin conversión. rd debe ser un registro entero y $rs1$ un registro flotante.

`fmv.d.x rd, rs1`
: Copiar secuencia de bits representando un flotante doble en registro entero a registro flotante. Se copia tal cual, sin conversión. rd debe ser un registro flotante y $rs1$ un registro entero.

4.5. Clasificación de números

Las instrucciones `fclass.s` y `fclass.d` clasifican valores flotantes de acuerdo a la clase de valor flotante que representan. Producen como resultado un valor entero del cual cada bit indica que se trata de una clase de valor flotante distinta. La siguiente tabla muestra las asociaciones:

Bit	Significado
0	$-\infty$
1	Número negativo normalizado
2	Número negativo desnormalizado
3	-0
4	$+0$
5	Número positivo desnormalizado
6	Número positivo normalizado
7	$+\infty$
8	NaN señalizador
9	NaN silencioso

Capítulo 5

Referencia de instrucciones

5.1. Instrucciones de I

La siguiente tabla lista todas las instrucciones del módulo I. Son 40 para RV32 y 12 más para RV64.

Instrucción	Descripción
`add R, R, R`	Sumar
`addi R, R, I12`	Sumar inmediato
`sub R, R, R`	Restar
`lui R, I20`	Cargar inmediato superior
`auipc R, I20`	Sumar inmediato a PC superior
`xor R, R, R`	Calcular disyunción exclusiva (*xor*) de bits
`xori R, R, I12`	Calcular disyunción exclusiva (*xor*) de bits con inmediato
`or R, R, R`	Calcular disyunción inclusiva (*or*) de bits
`ori R, R, I12`	Calcular disyunción inclusiva (*or*) de bits con inmediato
`and R, R, R`	Calcular conjunción (*and*) de bits
`andi R, R, I12`	Calcular conjunción (*and*) de bits con inmediato
`sll R, R, R`	Correr bits a izquierda
`slli R, R, I12`	Correr bits a izquierda con inmediato
`srl R, R, R`	Correr bits a derecha de forma lógica
`srli R, R, I12`	Correr bits a derecha de forma lógica con inmediato
`sra R, R, R`	Correr bits a derecha de forma aritmética
`srai R, R, I12`	Correr bits a derecha de forma aritmética con inmediato
`lb R, R, M`	Cargar byte, con signo
`lbu R, R, M`	Cargar byte, sin signo
`lh R, R, M`	Cargar media palabra, con signo
`lhu R, R, M`	Cargar media palabra, sin signo
`lw R, R, M`	Cargar palabra

`sb R, R, M`	Almacenar byte
`sh R, R, M`	Almacenar media palabra
`sw R, R, M`	Almacenar palabra
`jal R, M20`	Saltar y enlazar
`jalr R, R, M`	Saltar a registro y enlazar
`beq R, R, M`	Saltar si es igual
`bne R, R, M`	Saltar si es distinto
`blt R, R, M`	Saltar si es menor, con signo
`bltu R, R, M`	Saltar si es menor, sin signo
`bge R, R, M`	Saltar si es mayor o igual, con signo
`bgeu R, R, M`	Saltar si es mayor o igual, sin signo
`slt R, R, R`	Comparar por menor, con signo
`slti R, R, I12`	Comparar por menor que inmediato, con signo
`sltu R, R, R`	Comparar por menor, sin signo
`sltiu R, R, I12`	Comparar por menor que inmediato, sin signo
`fence ...`	Ordenar accesos a memoria y dispositivos
`ecall`	Pedir servicio al entorno de ejecución (llamada a sistema)
`ebreak`	Retornar control a un entorno de depuración
`addw R, R, R`	Sumar
`addiw R, R, I`	Sumar inmediato
`subw R, R, I`	Restar
`sllw R, R, R`	Correr bits a izquierda
`slliw R, R, I`	Correr bits a izquierda con inmediato
`srlw R, R, R`	Correr bits a derecha de forma lógica
`srliw R, R, I`	Correr bits a derecha de forma lógica con inmediato
`sraw R, R, R`	Correr bits a derecha de forma aritmética
`sraiw R, R, I`	Correr bits a derecha de forma aritmética con con inmediato
`lwu R, M`	Cargar palabra, sin signo
`ld R, M`	Cargar doble palabra
`sd R, M`	Almacenar doble palabra

5.2. Instrucciones de M

La extensión M incorpora 8 instrucciones en RV32 y 5 más en RV64.

Instrucción	Descripción
`mul R, R, R`	Multiplicar y obtener parte baja
`mulh R, R, R`	Multiplicar y obtener parte alta, con signo × con signo
`mulhsu R, R, R`	Multiplicar y obtener parte alta, sin signo × sin signo
`mulhu R, R, R`	Multiplicar y obtener parte alta, con signo × sin signo

`div R, R, R`	Dividir y obtener cociente, con signo
`divu R, R, R`	Dividir y obtener cociente, sin signo
`rem R, R, R`	Dividir y obtener resto, con signo
`remu R, R, R`	Dividir y obtener resto, sin signo
`mulw R, R, R`	Como `mul` pero sobre palabras
`divw R, R, R`	Como `div` pero sobre palabras
`divuw R, R, R`	Como `divu` pero sobre palabras
`remw R, R, R`	Como `rem` pero sobre palabras
`remuw R, R, R`	Como `remu` pero sobre palabras

5.3. Instrucciones de F

El módulo F incorpora 36 instrucciones.

Instrucción	Descripción
`fadd.s F, F, F`	Sumar
`fsub.s F, F, F`	Restar
`fmul.s F, F, F`	Multiplicar
`fdiv.s F, F, F`	Dividir
`fsqrt.s F, F`	Raíz cuadrada
`fmin.s F, F, F`	Mínimo
`fmax.s F, F, F`	Máximo
`fmadd.s F, F, F`	Multiplicar y sumar
`fmsub.s F, F, F`	Multiplicar y restar
`fnmsub.s F, F, F`	Multiplicar, negar y sumar
`fnmadd.s F, F, F`	Multiplicar, negar y restar
`fsgnj.s F, F, F`	Inyectar signo
`fsgnjn.s F, F, F`	Inyectar signo negado
`fsgnjx.s F, F, F`	Inyectar signo con *xor*
`fcvt.w.s R, F`	Convertir flotante a entero con signo
`fcvt.l.s R, F`	Convertir flotante a entero con signo
`fcvt.s.w F, R`	Convertir entero con signo a flotante
`fcvt.s.l F, R`	Convertir entero con signo a flotante
`fcvt.wu.s R, F`	Convertir flotante a entero sin signo
`fcvt.lu.s R, F`	Convertir flotante a entero sin signo
`fcvt.s.wu F, R`	Convertir entero sin signo a flotante
`fcvt.s.lu F, R`	Convertir entero sin signo a flotante
`feq.s R, F, F`	Comparar por igualdad
`flt.s R, F, F`	Comparar por menor
`fle.s R, F, F`	Comparar por menor o igual

`fclass.s R, F`	Clasificar valor flotante
`flw F, M`	Cargar flotante
`fsw F, M`	Almacenar flotante
`fmv.x.w R, F`	Copiar secuencia de bits
`fmv.w.x F, R`	Copiar secuencia de bits
`frcsr R`	Leer `fcsr`
`fscsr R`	Escribir `fcsr`
`frrm R`	Leer campo de modo de redondeo de `fcsr`
`fsrm R`	Escribir campo de modo de redondeo de `fcsr`
`frflags R`	Leer campo de banderas acumuladas de `fcsr`
`fsflags R`	Escribir campo de banderas acumuladas de `fcsr`

5.4. Instrucciones de D

El módulo D incorpora 32 instrucciones.

Instrucción	Descripción
`fadd.d F, F, F`	Sumar
`fsub.d F, F, F`	Restar
`fmul.d F, F, F`	Multiplicar
`fdiv.d F, F, F`	Dividir
`fsqrt.d F, F`	Raíz cuadrada
`fmin.d F, F, F`	Mínimo
`fmax.d F, F, F`	Máximo
`fmadd.d F, F, F`	Multiplicar y sumar
`fmsub.d F, F, F`	Multiplicar y restar
`fnmsub.d F, F, F`	Multiplicar, negar y sumar
`fnmadd.d F, F, F`	Multiplicar, negar y restar
`fsgnj.d F, F, F`	Inyectar signo
`fsgnjn.d F, F, F`	Inyectar signo negado
`fsgnjx.d F, F, F`	Inyectar signo con *xor*
`fcvt.w.d R, F`	Convertir flotante a entero con signo
`fcvt.l.d R, F`	Convertir flotante a entero con signo
`fcvt.d.w F, R`	Convertir entero con signo a flotante
`fcvt.d.l F, R`	Convertir entero con signo a flotante
`fcvt.wu.d R, F`	Convertir flotante a entero sin signo
`fcvt.lu.d R, F`	Convertir flotante a entero sin signo
`fcvt.d.wu F, R`	Convertir entero sin signo a flotante
`fcvt.d.lu F, R`	Convertir entero sin signo a flotante
`fcvt.s.d F, F`	Convertir flotante doble a simple

`fcvt.d.s F, F`	Convertir flotante simple a doble
`feq.d R, F, F`	Comparar por igualdad
`flt.d R, F, F`	Comparar por menor
`fle.d R, F, F`	Comparar por menor o igual
`fclass.d R, F`	Clasificar valor flotante
`fld F, M`	Cargar flotante
`fsd F, M`	Almacenar flotante
`fmv.x.d R, F`	Copiar secuencia de bits
`fmv.d.x F, R`	Copiar secuencia de bits

5.5. Pseudoinstrucciones

La siguiente tabla muestra algunas seudoinstrucciones comunes.

Pseudoinstrucción	Descripción
`la R, M32`	Cargar dirección
`lla R, M32`	Cargar dirección local
`lb R, M32`	Cargar byte global
`lh R, M32`	Cargar media palabra global
`lw R, M32`	Cargar palabra global
`sb R, M32, R`	Guardar byte global
`sh R, M32, R`	Guardar media palabra global
`sw R, M32, R`	Guardar palabra global
`nop`	No hacer nada
`li R, I32`	Cargar inmediato
`mv R, R`	Copiar registro
`not R, R`	Complemento a uno
`neg R, R`	Complemento a dos
`seqz R, R`	Comparar por igualdad a cero
`snez R, R`	Comparar por distinto de cero
`sltz R, R`	Comparar por menor que cero
`sgtz R, R`	Comparar por mayor que cero
`beqz R, M`	Saltar si es igual a cero
`bnez R, M`	Saltar si es distinto de cero
`blez R, M`	Saltar si es menor o igual que cero
`bgez R, M`	Saltar si es mayor o igual que cero
`bltz R, M`	Saltar si es menor que cero
`bgtz R, M`	Saltar si es mayor que cero
`bgt R, R, M`	Saltar si es mayor, con signo
`bgtu R, R, M`	Saltar si es mayor, sin signo

`ble R, R, M`	Saltar si es menor o igual, con signo
`bleu R, R, M`	Saltar si es menor o igual, sin signo
`j M`	Saltar
`jal M`	Saltar y enlazar
`jr R`	Saltar a registro
`jalr R`	Saltar a registro y enlazar
`ret`	Retornar de subrutina
`call M`	Llamar a subrutina lejana
`tail M`	Llamar de cola a subrutina lejana

En RV64I:

`ld R, M32`	Cargar
`sd R, M32, R`	Almacenar
`sext.w R, R`	Extender signo en palabra

En RV32F:

`fmv.s R, R`	Copiar flotante a flotante
`fabs.s R, R`	Valor absoluto
`fneg.s R, R`	Negación

En RV32D:

`fmv.d R, R`	Copiar flotante a flotante
`fabs.d R, R`	Valor absoluto
`fneg.d R, R`	Negación

Capítulo 6

Enlaces y libros

6.1. Enlaces

- Sitio web oficial de RISC-V:
  `https://riscv.org/`
- Manual de RISC-V, volumen 1 (conjunto de instrucciones no privilegiado):
  `https://content.riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf`
- Panorama del ecosistema de software de RISC-V:
  `https://riscv.org/software-status/`
- Arquitectura RISC-V en Devopedia:
  `https://devopedia.org/risc-v-architecture`
- Convención de llamada:
  `https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf`
- rv8 – RISC-V instruction set reference:
  `https://rv8.io/isa`
- Libros de RISC-V:
  `https://riscv.org/risc-v-books/`
- Materiales educativos de RISC-V:
  `https://riscv.org/educational-materials/`

- *Instruction sets should be free: the case for RISC-V*: `https://people.eecs.berkeley.edu/~krste/papers/EECS-2014-146.pdf`

6.2. Libros

- David Patterson, John Hennessy. *Computer organization and design: the hardware/software interface.* RISC-V 5th edition. ISBN: 978-0128122754.
- David Patterson, Andrew Waterman. *The RISC-V reader: an open architecture atlas.* ISBN: 978-0999249116. `http://www.riscvbook.com/`
- John Hennessy, David Patterson. *Computer architecture: a quantitative approach.* 6th edition. ISBN: 978-0128119051.