INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Unidad de Aprendizaje

Arquitectura de Computadoras

Marzo 2025

Registros y ABI (App. Binary Interface)

Registros enteros del Conjunto Base (set base)

Reg.	ABI	Descripción	Guarda
x0	zero	Siempre vale cero. Sólo lectura.	
x1	ra	**R**eturn **A**ddress. Dirección de retorno en una subrutina.	
x2	sp	**S**tack **P**ointer. Puntero de la pila.	Sí
x3	gp	**G**lobal **P**ointer. Puntero global usado para acceder al ".bss"	
x4	tp	**T**hread **P**ointer. Análogo a gp, pero multihilo.	
x5..x7	t0..t2	**T**emporales. Se puede sobrescribir su valor sin restaurarlos.	
x8	s0/fp	**S**aved Register/**F**rame **P**ointer. Suele usarse para depurar (GDB).	Sí
x9	s1	**S**aved Register. Como Temporal, pero hay que restaurar valor.	Sí
x10..x11	a0..a1	**F**unction **A**rgument/Return Value. Devolver valor sin stack.	
x12..x17	a2..a7	**F**unction **A**rgument. Recibir valores sin usar el stack.	
x18..x27	s2..s11	**S**aved Register. (No disponibles en modo comprimido).	Sí
x28..x31	t3..t6	**T**emporales. (No disponibles en modo comprimido).	Sí

Formatos de Instrucciones de 32 bits

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

Formato	Uso principal	Ejemplos de instrucciones
R	Operaciones entre registros	`ADD, SUB, AND, OR, XOR, MUL`
I	Operaciones con inmediatos y acceso a memoria	`ADDI, LW, JALR`
S	Almacenamiento en memoria	`SW, SB, SH`
B	Saltos condicionales	`BEQ, BNE, BLT, BGE`
U	Carga de inmediatos de 20 bits	`LUI, AUIPC`
J	Saltos incondicionales	`JAL`

Guía Práctica de RISC-V: El atlas de una arquitectura abierta-**Tarjeta de referencia para RISC-V (código abierto),** David Patterson, Andrew Waterman

Formato Tipo B (Branch - Saltos Condicionales)

Se emplean dos registros fuente y un desplazamiento.

RV32I permite comparar dos registros y saltar si el resultado es igual (beq), distinto (bne), mayor o igual (bge), o menor (blt), estos dos últimos casos son comparaciones con signo (aunque también hay versiones sin signo: bgeu y bltu).
Las relaciones de mayor que y menor o igual se obtienen intercambiando los argumentos, dado que $x < y$ implica $y > x$ y $x \geq y$ equivale a $y \leq x$.

Algunas de las instrucciones características son:

BEQ - Branch if Equal
BNE - Branch if Not Equal
BLT - Branch if Less Than (BGT-Branch if Greater Than es una pseudo-instrucción no existe de manera nativa).
BGE - Branch if Greater or Equal

Ejemplos:

```
beq  x1, x2,  label        # Salta a label si  x1 == x2
bgt rs1, rs2, etiqueta     # Se traduce a la instrucción de abajo
blt rs2, rs1, etiqueta     # Esta instrucción invierte el orden de los registros
```

Debe tomarse eque como RISC-V es una arquitectura RISC (*Reduced Instruction Set Computing*), lo que significa que sus instrucciones tienen tamaños bien definidos y alineados para facilitar la decodificación y ejecución eficiente en el hardware.

Por lo que en este tipo de arquitectura:

1. **La mayoría de las instrucciones (instrucciones estándar) tienen un tamaño de 4 bytes (32 bits)** y deben almacenarse en direcciones de memoria que sean **múltiplos de 4.** Esto significa que una instrucción no puede comenzar en cualquier dirección de memoria arbitraria, sino que debe estar en una dirección que sea **divisible por 4.**

   Ejemplo:

   Direcciones válidas para instrucciones de 4 bytes:

   0x1000
   0x1004
   0x1008
   0x100C

   Direcciones no válidas (No son múltiplos de 4:

   0x1001
   0x1002
   0x1003

2. Algunas instrucciones compactas (de la extensión RVC) pueden ocupar solo 2 bytes (16 bits).

RISC-V también tiene una extensión llamada RVC (*RISC-V Compressed*), donde algunas instrucciones pueden estar alineadas en múltiplos de 2 bytes, es decir, ocupan 2 bytes (16 bits) en lugar de los 4 bytes tradicionales, lo que permite ahorrar memoria en algunos casos.

Estas instrucciones sí pueden empezar en cualquier dirección de memoria que sea múltiplo de 2 (pero no en direcciones impares).

Ejemplo de alineación para instrucciones de 2 bytes:

Direcciones válidas para instrucciones de 2bytes:

0x1000
0x1002
0x1004
0x1006

Direcciones no válidas (No son múltiplos de 2:

0x1001
0x1003
0x1005

3. **Todas las instrucciones están alineadas en direcciones de memoria que son múltiplos de 2 bytes.**

El modo de direccionamiento de branches multiplica el valor inmediato de 12 bits por 2, le extiende el signo y lo suma al PC.

RISC-V hace dicha validación en software para validar si hay desbordamiento (overflow) aritmético.

Suma sin signo requiere solamente un branch adicional luego de la suma

```
addu t0, t1, t2
bltu t0, t1, overflow
```

Suma con signo, si se sabe el signo de un operando, validar el desbordamiento requiere un solo branch luego de la suma

```
addi t0, t1, +imm
blt t0, t1, overflow
```

En general, para validar el desbordamiento en suma con signo, se requieren tres instrucciones adicionales (xor, and y bnez) donde la suma debe ser menor que uno de los operandos, si y solo si, el otro operando es negativo.

El desbordamiento (overflow) ocurre cuando el resultado de una operación aritmética excede el rango que se puede representar con el número de bits disponibles.

Ejemplo:
En un sistema de 8 bits (para simplificar el ejemplo)

Si se suma 200 + 100 en una variable de 8 bits se tiene:

200 en binario: 11001000
100 en binario: 01100100
Resultado de la suma: **11001000** + **01100100** = **1 00101100**

Pero como solo tenemos 8 bits, el resultado real es **00101100** (44 en decimal) y el bit extra se pierde.

Este es un desbordamiento porque el resultado correcto debería ser 300, pero esta cantidad no se puede representar con 8 bits.

En arquitecturas como x86, hay una **bandera de desbordamiento (overflow flag)**, en RISC-V la CPU no detecta automáticamente el desbordamiento.

Por lo tanto, el programador (o el compilador) debe *hacer la validación en software.*

Desbordamiento en suma sin signo (addu + bltu)

En números sin signo (unsigned), el desbordamiento ocurre si: el resultado de la suma es menor que uno de los operandos.

Ejemplo:

```
addu t0, t1, t2          # t0 = t1 + t2
bltu t0, t1, overflow    # Si t0 < t1, ocurre desbordamiento porque debería ser
                           mayor o igual, puesto que t2  es positivo).
```

Desbordamiento en suma con signo (addi + blt)

En números con signo (signed), el desbordamiento ocurre si:

- Dos números positivos suman un número negativo (resultado incorrecto).
- Dos números negativos suman un número positivo (resultado incorrecto).

Nota: Si se sabe el signo de un operando, se puede usar una sola comparación para detectar el desbordamiento.

Ejemplo:

```
addi t0, t1, imm        # t0 = t1 + imm
blt  t0, t1, overflow   # Si t0 < t1, ocurre desbordamiento
```

Análisis:
Si imm es positivo, entonces t0 debería ser mayor que t1 después de la suma.
Si imm es negativo, entonces t0 debería ser menor que t1 después de la suma.
Si el resultado no cumple esta condición, ocurrió un desbordamiento.

Validación general de desbordamiento en suma con signo

Cuando no se saben los signos de los operandos a priori, entonces se requiere de tres instrucciones para verificar el desbordamiento:

Ejemplo:

```
add t0, t1, t2          # t0 = t1 + t2
xor t3, t1, t2          # t3 = t1 ⊕ t2 (si los signos son diferentes, t3 tendrá un 1
                          en el bit de signo. Si son iguales t3 tendrá un 0)
xor t4, t1, t0          # t4 = t1 ⊕ t0 (si el signo cambió ( respecto a t1) después
                          de la suma, por lo que t4 tendrá un 1 en el bit de signo)
and t5, t3, t4          # t5 = t3 & t4 (si ambos valoresson 1, hubo desbordamiento)
bnez t5, overflow       # Si t5 != 0 (hubo desbordamiento).
```

Las **instrucciones de corrimiento** (o desplazamiento) en RISC-V se utilizan para mover los bits de un registro hacia la izquierda o hacia la derecha.

Estas instrucciones son útiles para operaciones como multiplicar o dividir por potencias de 2 y para manipulación de bits.

La sintaxis de este tipo de instrucción es:
[instrucción] destino, origen, desplazamiento

destino es donde se almacenará el resultado.
origen es el registro que contiene el valor que se va a desplazar.
desplazamiento es el número de posiciones que se moveran los bits del registro.

slli: Desplazamiento lógico a la izquierda (Shift Left Logical Immediate)
srli: Desplazamiento lógico a la derecha (Shift Right Logical Immediate)
srai: Desplazamiento aritmético a la derecha (Shift Right Arithmetic Immediate)

slli- (Shift Left Logical Immediate): Desplazamiento lógico a la izquierda, y llena con ceros en los bits vacíos.

Ejem:
slli t0, t1, 2 # Desplaza el contenido del registro t1 2 posiciones a la izquierda y guarda el resultado en t0, lo que equivale a multiplicar por $2^2 = 4$.

srli-(Shift Right Logical Immediate): Desplazamiento lógico a la derecha y llena con ceros en los bits vacíos.

Ejem:
srli t0, t1, 3 #Desplaza el contenido de t1 en 3 posiciones a la derecha y guarda el resultado en t0, lo que equivale a dividir por $2^3 = 8$.

srai-(Shift Right Arithmetic Immediate): Desplazamiento aritmético a la derecha y mantiene el signo del número (por lo tanto, se utiliza para números negativos).

Ejem:
srai t0, t1, 3 #Desplaza el contenido de t1 en 3 posiciones a la derecha y guarda el resultado en t0, lo que equivale a dividir por $2^3 = 8$ pero conserva el signo del número

Formato Tipo J (Jump-Salto)

En el **formato J (Jump)** de RISC-V, la instrucción **Jal (Jump and Link)** se usa para realizar un **salto incondicional** a una dirección específica y almacenar la dirección de retorno en un registro. Esta dirección destino es de 26 bits, permitiendo saltos lejanos.

```
jal  x1, etiqueta     # Salto largo a "etiqueta" y guarda la dirección de
                      # retorno en el registro x1

jal x1, 10        #realiza un salto de 10 bytes (en términos de dirección de
                  #memoria)
```

x1 registro donde se almacena la dirección de retorno
10 Desplazamiento (offset) relativo al contador de programa (PC)

Formato Tipo J (Jump-Salto)

En el **formato J (Jump)** de RISC-V, la instrucción **Jal (Jump and Link)** se usa para realizar un **salto incondicional** a una dirección específica y almacenar la dirección de retorno en un registro. Esta dirección destino es de 26 bits, permitiendo saltos lejanos.

```
jal  x1, funcion     # Salto largo a “funcion" y guarda la dirección de
                     # retorno en el registro x1
```

1. La instrucción jal x1, funcion, guarda la dirección de retorno (PC+4) en el registro x1 (para regresar después).

2. Calcula la nueva dirección de ejecución sumando la dirección de funcion al valor actual del PC.

3. Realiza un salto incondicional a la nueva dirección calculada.

```
jal  x1, funcion     # Salto largo a “funcion" y guarda la dirección de
                     # retorno en el registro x1
```

Para regresar de la función se utiliza:

```
jal  x0, x1, 0       # Regresa a la dirección almacenada en x1
```

Este formato se utiliza para llamadas a funciones en subrutinas con lo que se pueden implementar estructuras de control como bucles o excepciones

Formato Tipo J (Jump-Salto)

```
main:
jal  x1,  subrutina_ejemplo    # Llama o salto a la subrutina ejemplo y guarda la
                                 dirección a la que debe retornar en el registro x1

# Aquí  va algún código después de la llamada


subrutina_ejemplo:

# Código de la subrutina ejemplo y la instrucción para regresar al código después
   de la llamada

jal  x0, x1, 0     # Regresa a la dirección almacenada en x1
```

Ejemplo

```
 .data
mensaje: .string "La suma es: "      # Mensaje a imprimir (se usa string en lugar de .asciz que es una cadena terminada en \0)
    .text
    .globl main

main:
    li a0, 5                        # Cargar primer número en a0
    li a1, 10                       # Cargar segundo número en a1

    jal  suma                       # Llamar a la subrutina de suma

    mv s0, a0                       # Guardar resultado en s0

   jal imprimir                     # Llamar al procedimiento de impresión

                                    # Finalizar el programa
    li a7, 10                       # Código de salida para la llamada al sist.(syscall)
    ecall                           # Llamada al sistema
```

```
 # SUBRUTINA: SUMA
suma:
   add a0, a0, a1          # Suma los valores contenidos en a0 y a1 y el
                           # resultado lo deja en registro a0  a0 = a0 + a1

   jr ra                   # Retornar a la llamada en main

# PROCEDIMIENTO: IMPRIMIR
imprimir:                  # Imprimir mensaje
   li a7, 4                # Código de syscall para imprimir string
   la a0, mensaje          # Cargar dirección del mensaje
   # ecall
                           # Imprimir número
   li a7, 1                # Código de syscall para imprimir entero
   mv a0, s0               # Cargar resultado en a0 ecall

                           # Imprimir salto de línea
   li a7, 11               # Código de syscall para imprimir un carácter
   li a0, 10               # de código ASCII de nueva línea
   ecall
   jr ra                   # Retornar a la llamada en main
```