DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Parcial único

Organización del Computador I

14 de noviembre de 2021 — 2do cuatrimestre 2021

Alumno	LU	Correo electrónico
[redacted]	[redacted]	[redacted]

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Soluciones

Ejercicio 1

RISC-V maneja el principio de `simplicidad`, en relación a esto responda:

- **¿Acceder a un operando en registro es más rápido que buscar el operando en memoria?**
- **A partir del inciso anterior, ¿cómo cree que impacta al rendimiento del programa y a la arquitectura la cantidad de registros disponibles?**

Acceder a un operando en un registro es más rápido que a buscarlo en memoria. Leer un registro es más directo, la dirección del registro pasa por el multiplexor y su contenido entra al bus. Para leer una posición de memoria en cambio, se debe actualizar la posición en la memoria, mandar su contenido al bus, y restablecer la posición de memoria.

Como el acceso a los registros es más rápido que a la memoria, es preferible mantener las variables en registros. Cuántos más registros, menos probabilidades de que se tenga que guardar un dato en memoria. Sin embargo, agregar registros implica aumentar la cantidad de bits del formato de instrucción. Si se tienen 16 registros, se necesitan $log_2(16) = 4$ bits para representarlos. Si se tienen 32, $log_2(32) = 5$ bits son necesarios. Al aumentar el tamaño de las instrucciones, aumenta también la cantidad de ciclos de reloj que son necesarios para hacer el fetch, como consecuencia el rendimiento empeora.

Impacta en la arquitectura

Es relativo al tamaño del bus de datos en realidad.

Además, desde el punto de vista de la simplicidad, los operandos en las instrucciones de RISC-V provienen únicamente de los registros. Esto es para simplificar el set de instrucciones y para hacer más sencillas las estimaciones de desempeño, ya que gracias a eso, las instrucciones de RISC-V se ejecutan en un ciclo de reloj (exceptuando cache misses).

Ejercicio 2

Explique de qué modo se resuelven los saltos incondicionales.

Para hacer un salto incondicional, se pueden usar las instrucciones `jal` (jump and link) ó `jalr` (jump and link register).

imm[20|10:1|11|19:12]			rd	1101111	J jal
imm[11:0]	rs1	000	rd	1100111	I jalr

Figura 1: Formato de instrucción de jal y jalr

La instrucción `jal` cumple dos propósitos. Por un lado, se puede utilizar para hacer un llamado a una función. En este caso, se almacena la dirección de la siguiente instrucción (PC+4) en el registro destino. Pero si lo que buscamos es hacer un salto incondicional (sin retorno), simplemente se guarda el registro cero (que vale siempre cero). Entonces `jal` multiplica la dirección del inmediato de 20 bits por 2, extiende el signo (a 32) y suma el resultado al PC para obtener la dirección a saltar. Al igual que en el procesador de OrgaI, el salto es relativo al PC.

La instrucción `jalr` tiene un comportamiento muy parecido. La diferencia está en que este salto no es relativo al PC, sino al `rs1` que se pasa como parámetro. Entonces, en lugar de sumarle la dirección del inmediato (previamente multiplicada por 2 y extendida en signo) al PC, se lo suma al contenido de `rs1` y así obtiene la dirección a saltar.

Muy bien!

Ejercicio 3

Dados dos registros, mostrar la forma de intercambiarlos sin la intervención de un tercero.

Se pueden intercambiar los valores de dos registros sin usar un registro adicional, usando las propiedades de la operación *xor*.

$$a \oplus b = b \oplus a \tag{1}$$

$$(a \oplus b) \oplus c = a \oplus (b \oplus c) \tag{2}$$

$$a \oplus a = 0 \tag{3}$$

$$a \oplus 0 = a \tag{4}$$

El procedimiento es el siguiente:

```
xor x1,x1,x2
xor x2,x1,x2
xor x1,x1,x2
```

Primero, se reemplaza a x_1 por $x_1 \oplus x_2$. Luego, se reemplaza a x_2 por $x_1 \oplus x_2$. Hasta ahora tenemos que

$$x_1' = x_1 \oplus x2$$

$$x_2' = x_1' \oplus x_2 = x_1 \oplus x_2 \oplus x_2 \overset{(3)}{=} x_1 \oplus 0 \overset{(4)}{=} x_1$$

Finalmente, se reemplaza a x_1 por $x_1 \oplus x_2$.

$$x_1'' = x_1' \oplus x_2' = x_1 \oplus x_2 \oplus x_1 \overset{(1)}{=} x_2 \oplus x_1 \oplus x_1 \overset{(3)}{=} x_2 \oplus 0 \overset{(4)}{=} x_2$$

Entonces, llegamos al resultado que buscábamos, pues tenemos que

$$x_1'' = x_2$$

$$x_2' = x_1$$

Excelente!

Ejercicio 4

¿Cómo se resuelve la lógica de control (`branching`)? ¿Qué similitudes y/ó diferencias existen con la máquina Orga1?

En RISC-V se pueden hacer saltos condicionales con las instrucciones `beq`, `bne`, `bge`, `blt`, `bgeu`, `bltu`. En el mismo orden, se da un salto si el resultado de la comparación de los registros es igual, distinto, mayor o igual, menor, mayor y menor ó igual.

Si se cumple la condición, el modo de direccionamiento de branches multiplica el valor del inmediato por 2, le extiende el signo y lo suma al PC. También se pueden hacer saltos incondicionales como se explicó en el Ejercicio 2.

imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu

Figura 2: Lógica de control

El salto, al igual que el procesador de OrgaI, es relativo al PC. Como se explicó en el Ejercicio 2, para realizar un salto se multiplica la dirección del inmediato por 2, para correrla un bit a la izquierda. Esto es porque las instrucciones de RISC-V son de 32 bits ó de 16 si se considera la Extensión Opcional de Instrucciones Comprimidas. Por eso, nunca va a ocurrir un salto de, por ejemplo, 1 byte (unidad mínima de direccionamiento), porque todas las instrucciones son de 2 ó 4 bytes. Entonces los saltos van a ser múltiplos de 2 bytes, es decir, pares. Entonces, como sabemos que el último bit de la dirección del salto va a ser siempre cero, no lo escribimos en la instrucción, lo agregamos después de la decodificación. De esta forma, nuestro rango de saltos pasa de $(-524288, 524287)$ a $(-1048576, 1048575)$ bytes para saltos incondicionales y de $(-2048, 2047)$ a $(-4096, 4095)$ bytes para saltos condicionales.

Otra diferencia con el procesador de OrgaI es que con una sola instrucción se puede hacer tanto un salto incondicional como una llamada a función.

Ejercicio 5

¿Qué sucede si no hay suficientes registros como para pasar los parámetros de una función?

RISC-V clasifica a los registros en distintos tipos. Los registros a0-7 (x10-17) guardan los argumentos de una función, son temporales. a0-1 guardan el resultado. Luego, s0-11(x8-x9, x18-x27) guardan valores que no queremos perder durante la llamada a la función y el frame pointer. Finalmente, sp (x2) y ra (x1) guardan el stack pointer y la dirección de retorno respectivamente.

Registro	Nombre ABI	Descripción	¿Preservado en llamadas?
x0	zero	Alambrado a cero	—
x1	ra	Dirección de retorno	No
x2	sp	Stack pointer	Sí
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Link register temporal/alterno	No
x6–7	t1–2	Temporales	No
x8	s0/fp	Saved register/frame pointer	Sí
x9	s1	Saved register	Sí
x10–11	a0–1	Argumentos de función/valores de retorno	No
x12–17	a2–7	Argumentos de función	No
x18–27	s2–11	Saved registers	Sí
x28–31	t3–6	Temporales	No
f0–7	ft0–7	Temporales, FP	No
f8–9	fs0–1	Saved registers, FP	Sí
f10–11	fa0–1	Argumentos/valores de retorno, FP	No
f12–17	fa2–7	Argumentos, FP	No
f18–27	fs2–11	Saved registers, FP	Sí
f28–31	ft8–11	Temporales, FP	No

Figura 3: Clasificación de registros

Al hacer un llamado a una función debemos encargarnos de varias cuestiones. Primero, preservar los valores de todos los registros que se utilicen en la rutina principal (s) en el stack y restaurarlos luego de salir de la función. Además, debemos poder restaurar el PC y el SP. Finalmente, guardar los argumentos de la función en los registros a. Si no alcanzan, se guardan en el stack.

Excelente!

Ejercicio 6

¿En qué posición dentro de la instrucción se encuentran los bits de los registros destino y origen? ¿Depende del tipo de instrucción o de la instrucción en sí? ¿Por qué fue diseñado así el formato de instrucción?

Dependiendo de el tipo de instrucción, se puede requerir un registro destino `rd` (R, I, U, J) y un registro origen `rs1` (R, I, S, B). Además, existen operaciones que precisan de un registro adicional `rs2` (R, S, B).

En todas las instrucciones que requieran un registro destino, `rd` se ubica entre los bits 7 y 11. En las que requieran un registro origen `rs1`, lo ubican entre los bits 15 y 19. Y en las que requieran un registro origen `rs2`, entre los bits 20 y 24.

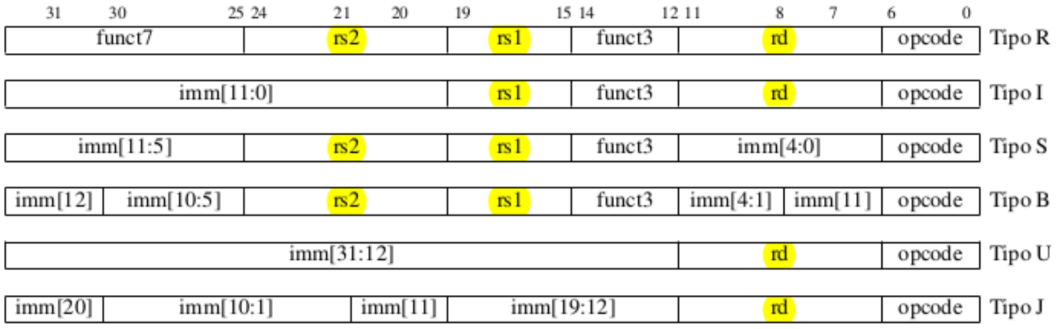

Figura 4: Formato de instrucción

El hecho de que los bits de los registros de origen y destino estén siempre en la misma posición en el formato de instrucción, nos permite acceder a dichos registros antes de la decodificación, sin importar cuál sea la instrucción particular. Esto también facilita la decodificación, ya que no hace falta agregar hardware extra para seleccionar el campo correcto.

¡Excelente trabajo!
Felicitaciones.