Parte III

La Unidad Central de Proceso (CPU)

Competencias genéricas

- Que los estudiantes sepan aplicar sus conocimientos a su trabajo o vocación de una forma profesional y posean las competencias que suelen demostrarse por medio de la elaboración y defensa de argumentos y la resolución de problemas dentro de su área de estudio.
- Que los estudiantes tengan la capacidad de reunir e interpretar datos relevantes (normalmente dentro de su área de estudio) para emitir juicios que incluyan una reflexión sobre temas relevantes de índole social, científica o ética.

Competencias específicas

- Conocer la estructura, organización, funcionamiento e interconexión de los sistemas informáticos.
- Conocer la aplicación de los sistemas informáticos para la resolución de problemas propios de la ingeniería.
- Adquirir la capacidad de análisis y resolución de problemas.
- Saber interpretar los resultados.
- Desarrollar habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía.

Capítulo 4

Aritmética del computador

4.1. Introducción

La Unidad Aritmético-Lógica (ALU) es la parte del computador que realiza las operaciones aritméticas y lógicas con los datos. Como se ha comentado en capítulos anteriores, la ALU forma junto con la Unidad de Control (UC) la Unidad Central de Proceso (CPU). El resto de los elementos del computador (UC, registros, memoria y E/S) principalmente sirven para suministrar datos a la ALU, para que los procese y obtenga un resultado.

Los dos aspectos fundamentales de la aritmética del computador son la forma de representar los números (usando el formato binario) y los algoritmos utilizados para realizar las operaciones aritméticas básicas (suma, resta, multiplicación y división). Estas dos consideraciones se aplican tanto a la aritmética de enteros como a la de coma flotante.

Las palabras de los ordenadores se componen de bits y, por lo tanto, esas palabras se pueden interpretar como números binarios. Por ejemplo, el 0, el 1, el 2, etc, se representarán como 0000, 0001, 0010, etc. Teniendo en cuenta esto, ¿cómo representamos los negativos? ¿y los números reales?

4.2. Enteros sin signo

4.2.1. Representación

Si trabajamos con números naturales (sin signo), su representación será una conversión de su valor decimal a su valor binario. Por ejemplo, el valor 19 en base 10 se pasa a binario y se corresponde con el valor 10011 en base 2.

Para pasar un número de base 10 a base 2, nos fijamos en la mayor potencia de 2 que no supera el valor a codificar. En el caso del número 19, dicha potencia sería 2^4 que vale 16 ($2^5 = 32$ ya es mayor que 19). Después de restar a 19 el valor de 2^4 nos queda $19 - 2^4 = 3$. Buscamos la

	0101	(5)
+	0011	(3)
	1000	(8)

Tabla 4.1: *La suma de 5 (0101) con 3 (0011) usando 4 bits da 8 (1000).*

	1001	(9)
+	1000	(8)
1	0001	(17)

Tabla 4.2: *La suma de 9 (1001) con 8 (1000) usando 4 bits da 17. Sin embargo el resultado es 0001 y el acarreo de salida del bit de más peso es 1 porque 17 no es representable con 4 bits, se necesitan 5 (10001).*

mayor potencia de 2 que no supere el valor 3: $2^1 = 2$ ($2^2 = 4$ es mayor que 3). Después de restar a 3 el valor de 2^1 nos queda $3 - 2^1 = 1$. Buscamos la mayor potencia de 2 que no supere el valor 1: $2^0 = 1$. Al restar, ya obtengo el valor 0. Para escribir el valor binario correspondiente ponemos 0s en las potencias de 2 que no he utilizado y 1 en aquellas que sí: 10011.

4.2.2. Suma

Dados dos números sin signo, se suman como dos números binarios cualesquiera, mediante la suma binaria (como se muestra en la tabla 4.1). La suma produce desbordamiento si el acarreo de salida del bit de más peso es 1 (por ejemplo, en la tabla 4.2). Se produce desbordamiento cuando el resultado no se puede representar con el número de bits indicado.

4.2.3. Multiplicación binaria sin signo

Para multiplicar dos números enteros sin signo A y B representados en binario con n bits, se emplea el algoritmo [4, 3] que aparece en la figura 4.1. El resultado es un número de $2n$ bits.

Vemos que en caso de que a_0 sea 0, hay que hacer $P = P + 0$. Si este algoritmo se implementa en software, no haría nada. Si se implementa en hardware, hay que decir en el caso de la suma, qué hay que sumar, cuál es el segundo operando.

Este algoritmo se corresponde con la instrucción del MIPS `multu $rs, $rt` la cual deja el resultado (64 bits) en dos registros: los 32 bits de más peso (P) quedan en el registro `HI` y los 32 bits de menos peso (A) quedan en el registro `LO`.

Por ejemplo, si queremos multiplicar 3 (A=0011) por 7 (B=0111) usando n=4 bits aplicando el algoritmo, seguiríamos la secuencia de pasos mostrada en la tabla 4.3. El resultado vemos que es 21 y que tiene 8 bits.

En el caso en que multipliquemos números con 3 bits, el bucle se repite sólo 3 veces. Por ejemplo, si multiplicamos A=4 (100) y B=2 (010) con n=3 bits, seguiríamos la secuencia de pasos mostrada en la tabla 4.4. El resultado vemos que es 8 y que tiene 6 bits.

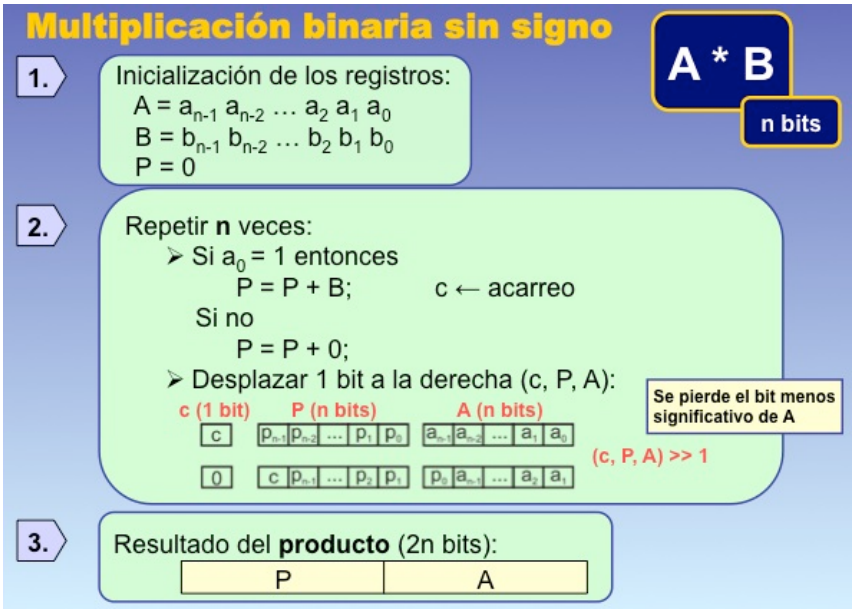

Figura 4.1: *Algoritmo para la multiplicación de dos números enteros sin signo A y B usando n bits para representarlos.*

iter.	c		P	A	Acción
0)	0		0000	0011	P=P+B
		+	0111	0011	
	0		0111	0011	>> 1
			0011	1001	
1)			0011	1001	P=P+B
		+	0111	1001	
	0		1010	1001	>> 1
			0101	0100	
2)			0101	0100	P=P+0
		+	0000	0100	
	0		0101	0100	>> 1
			0010	1010	
3)			0010	1010	P=P+0
		+	0000	1010	
	0		0010	1010	>> 1
			0001	0101	

Tabla 4.3: *El producto de 3 (0011) por 7 (0111) usando 4 bits da 21 (00010101).*

iter.	c		P	A	Acción
0)	0		000	100	P=P+0
		+	000	100	
	0		000	100	>> 1
			000	010	
1)			000	010	P=P+0
		+	000	010	
	0		000	010	>> 1
			000	001	
2)			000	001	P=P+B
		+	010	001	
	0		010	001	>> 1
			001	000	

Tabla 4.4: *El producto de 4 (100) por 2 (010) usando 3 bits da 8 (001000).*

4.2.4. División binaria sin signo

Para dividir enteros sin signo hay dos métodos [4, 3]: con reestablecimiento (ver figura 4.2) y sin reestablecimiento (ver figura 4.3). Se llama con reestablecimiento porque se va a restar un valor y, en ocasiones, dicho valor se va a volver a sumar. En hardware se implementa de forma muy parecida.

Se corresponde con la intrucción del MIPS `divu $rs, $rt` que deja el cociente en el registro de 32 bits `LO` y el resto en el registro `HI`, también de 32 bits.

Por ejemplo, si queremos dividir 6 (A=0110) entre 4 (B=0100) usando n=4 bits aplicando el algoritmo con reestablecimiento, seguiríamos la secuencia de pasos mostrada en la tabla 4.5. El cociente vemos que es 1 y el resto es 2.

También podemos usar el algoritmo sin reestablecimiento. Por ejemplo, si queremos dividir 3 (A=011) entre 2 (B=010) usando n=3 bits aplicando el algoritmo sin reestablecimiento, seguiríamos la secuencia de pasos mostrada en la tabla 4.6. El cociente vemos que es 1 y el resto es 1.

4.3. Enteros con signo

4.3.1. Representación

Si queremos representar enteros positivos y negativos necesitamos especificar el signo. Denominaremos bit de más peso o más significativo al bit de más a la izquierda y bit de menos peso o menos significativo al bit de más a la derecha. Hay distintas formas de representación de enteros con signo [4, 3].

Figura 4.2: *Algoritmo para la división de dos números enteros sin signo A y B usando n bits para representarlos (con reestablecimiento).*

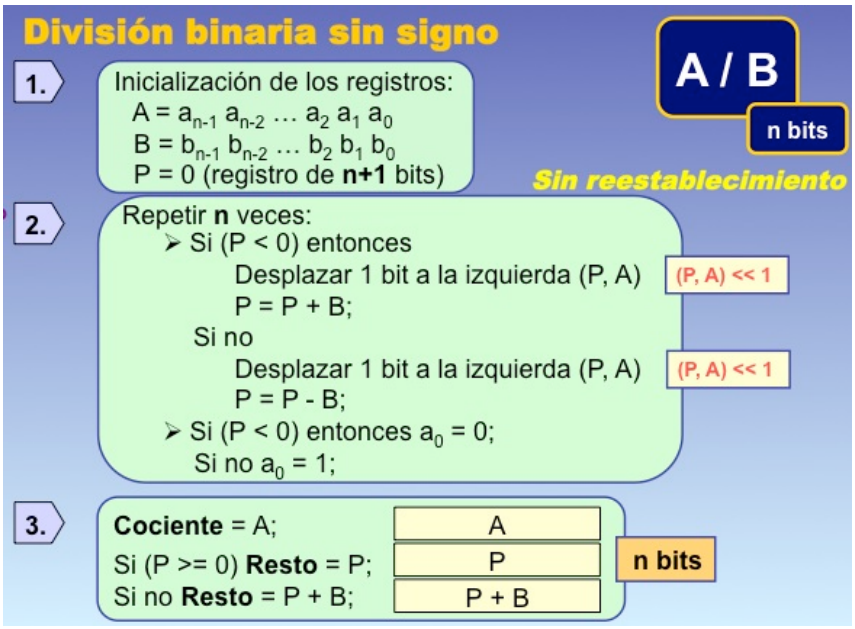

Figura 4.3: *Algoritmo para la división de dos números enteros sin signo A y B usando n bits para representarlos (sin reestablecimiento).*

iter.		P	A	Acción
0)		00000	0110	<< 1
		00000	1100	
		00000	1100	P=P-B=P+(-B)=P+C2(B)
	+	11100	1100	
		11100	1100	Como P<0 (empieza en 1) $a_0 = 0$
	+	00100	1100	Como P<0 (empieza en 1) P=P+B
		00000	1100	
1)		00000	1100	<< 1
		00001	1000	
		00001	1000	P=P-B=P+(-B)=P+C2(B)
	+	11100	1000	
		11101	1000	Como P<0 (empieza en 1) $a_0 = 0$
	+	00100	1000	Como P<0 (empieza en 1) P=P+B
		00001	1000	
2)		00001	1000	<< 1
		00011	0000	
		00011	0000	P=P-B=P+(-B)=P+C2(B)
	+	11100	0000	
		11111	0000	Como P<0 (empieza en 1) $a_0 = 0$
	+	00100	0000	Como P<0 (empieza en 1) P=P+B
		00011	0000	
3)		00011	0000	<< 1
		00110	0000	
		00110	0000	P=P-B=P+(-B)=P+C2(B)
	+	11100	0000	
		00010	0001	Como P>0 (empieza en 0) $a_0 = 1$

Tabla 4.5: *El cociente de 6 (0110) entre 4 (0100) usando 4 bits da 1 (0001) y se encuentra en el registro A. El resto da 2 (0010) y queda en el registro P.*

iter.		P	A	Acción
0)		0000	011	Como P no es <0:
		0000	011	<< 1
		0000	110	P=P-B=P+(-B)=P+C2(B)
	+	1110	110	
		1110	110	Como P<0 (empieza en 1) $a_0 = 0$
1)		1110	110	Como P<0 entonces << 1
		1101	100	
		1101	100	P=P+B
	+	0010	100	
		1111	100	Como P<0 (empieza en 1) $a_0 = 0$
2)		1111	100	Como P<0 (empieza en 1) << 1
		1111	000	
		1111	000	P=P+B
	+	0010	000	
		0001	001	Como P>0 (empieza en 0) $a_0 = 1$

Tabla 4.6: *El cociente de 3 (011) entre 2 (010) usando 3 bits da 1 (001) y se encuentra en el registro A. El resto da 1 (001) y queda en el registro P.*

a) Signo-Magnitud.

Se utiliza un bit de más peso para representar el signo: 0 si el número es positivo y 1 si el número es negativo. La magnitud se codifica calculando su valor en binario sin considerar el signo usando los bits restantes.

Por ejemplo, si queremos representar el número 6 y - 6 con 5 bits, la codificación sería la siguiente:

- Número 6: Signo = positivo (0). Magnitud = 6 representado en binario con 4 bits (0110). Por tanto, su representación sería 00110.
- Número -6: Signo = negativo (1). Magnitud = 6 representado en binario con 4 bits (0110). Por tanto, su representación sería 10110.

b) Complemento a 1.

Los números positivos se representan en base dos. Los números negativos se representan sustituyendo cada bit del número en positivo por su complemento.

Por ejemplo, si queremos representar el número 6 y - 6 con 5 bits, la codificación sería la siguiente:

- Número 6: como es positivo, se representa en binario con 5 bits (00110). Por tanto, su representación sería 00110.
- Número -6: como es negativo, se calcula el complemento a 1 C1(00110) cambiando 0s por 1s y 1s por 0s. Por tanto, su representación sería 11001.

c) Complemento a 2.

Los números positivos se representan en base dos. Los números negativos se representan calculando el complemento a 1 y sumándole 1. Una manera más rápida de calcular el complemento a 2 es localizar el 1 de más a la derecha del número y desde ese 1 hasta el final, se deja como está; los 0s y 1s que quedan en la parte de la izquierda, se sustituyen 0s por 1s y 1s por 0s.

Por ejemplo, si queremos representar el número 6 y - 6 con 5 bits, la codificación sería la siguiente:

- Número 6: como es positivo, se representa en binario con 5 bits (00110). Por tanto, su representación sería 00110.
- Número -6: como es negativo, se calcula el complemento a 2 C2(00110). Localizamos el último 1 y desde dicho 1 hasta la derecha lo dejamos como está (10). El resto, cambiamos 0s por 1s y 1s por 0s (110). Por tanto, su representación sería 11010.

El valor decimal de un número representado en complemento a 2, es el siguiente: $00110_{(2} = 0*(-1)*2^4+0*2^3+1*2^2+1*2^1+0*2^0 = 0+0+4+2+0 = 6$. Para el caso de un número negativo

(aquellos cuyo bit de más peso es 1) sería $11010_{(2} = 1*(-1)*2^4+1*2^3+0*2^2+1*2^1+0*2^0 = -16+8+0+2+0 = -6$.

En general, dado un número binario $a_{n-1}a_{n_2}...a_2a_1a_0$ siendo $a_i \in 0,1$ representado en complemento a 2, se calcula el valor decimal al que representa según la siguiente fórmula:

$$ValorDecimal = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \tag{4.1}$$

Según la representación en complemento a 2, el valor máximo y mínimo representable utilizando complemento a 2 y n bits, sería el siguiente:

- Positivos: El mayor valor será 01111...1111. Anulamos el término negativo ($-a_{n-1}2^{n-1} = 0$) poniendo el bit de más peso a 0 ($a_{n-1} = 0$). El resto de bits valen 1 para poder sumar las potencias de 2 ($a_{n-2}2^{n-2}+a_{n-3}2^{n-3}+...+a_2 2^2+a_1 2^1+a_0 2^0 = 2^{n-2}+2^{n-3}+...+2^2+2^1+2^0$).
- Negativos: El menor valor será 10000...0000. Anulamos los términos positivos ($\sum_{i=0}^{n-2} a_i 2^i$=0) poniendo todos los bits menos el de más peso a 0 ($a_i = 0$ para $i = 0, 1, ...n-2$). El bit de más peso vale 1 para representar el valor negativo ($-a_{n-1}2^{n-1} = -1*2^{n-1} = 2^{n-1}$).

d) Exceso e.

Un valor entero decimal se codifica sumándole el valor e y representando el resultado en binario sin signo. Con n bits es habitual $e = 2^{n-1}$. Si el número resultante de sumarle el exceso e es negativo, no es representable en dicho exceso.

Por ejemplo, si queremos representar el número 6 y - 6 con 5 bits en exceso 16, la codificación sería la siguiente:

- Número 6: se le suma el exceso 16 y nos da $6+16 = 22$. Se representa 22 en binario con 5 bits (10110). Por tanto, su representación sería 10110.
- Número -6: se le suma el exceso 16 y nos da $-6+16 = 10$. Se representa 10 en binario con 5 bits (01010). Por tanto, su representación sería 01010.

Por otro lado, si queremos representar el número - 19 con 5 bits en exceso 16, la codificación sería la siguiente:

- Número -19: se le suma el exceso 16 y nos da $-19+16 = -3$. Como el resultado es un número negativo, podemos decir que -19 no es representable en exceso 16 con 5 bits.

4.3.2. Suma

Dados dos números con signo, se suman como dos números binarios cualesquiera, mediante la suma binaria (como se muestra en las tablas 4.7 y 4.8). La suma produce desbordamiento si el acarreo de entrada y de salida del bit de más peso son distintos (por ejemplo, en la tabla 4.9).

	0101	(5)
+	0011	(3)
	1000	(8)

Tabla 4.7: *La suma de 5 (0101) con 3 (0011) usando 4 bits da 8 (1000).*

	1011	(-5)
+	0011	(3)
	1110	(-2)

Tabla 4.8: *La suma de -5 (1011) con 3 (0011) usando 4 bits da -2 (1110).*

Se produce desbordamiento cuando el resultado no se puede representar con el número de bits indicado.

Si sumamos dos números con signo con distinto número de bits, lo primero que hay que hacer es extender el signo del operando con menor número de bits para igualar dicho número. Por ejemplo, si sumamos 1001 y 01, tendríamos que extender el signo del segundo operando para que ocupe 4 bits, quedando la suma 1001 + 0001. Lo mismo ocurre si el número es negativo, en este caso, el bit que se repite (bit de signo) vale 1. Por ejemplo, si sumamos 1001 y 101, extendemos el signo del segundo operando (1) y nos quedaría 1101.

4.3.3. Resta

Dados dos números con signo, se restan calculando el complemento a 2 del segundo operando y realizando la suma como dos números binarios con signo cualesquiera. Por ejemplo, si queremos restar 4 y 5, se calcula la suma de 4 y -5, siendo -5 el complemento a 2 de 5 (ver tabla 4.10).

4.3.4. Desplazamiento lógico y aritmético

Mediante desplazamientos podemos multiplicar (desplazamientos a la izquierda) y dividir (desplazamientos a la derecha) por potencias de 2. Por ejemplo, si tenemos el número 6 en complemento a 2 representado con 6 bits, 000110, podemos:

- Multiplicarlo por 4, desplazándolo 2 posiciones hacia la izquierda: $000110 << 2 = 011000$. El número resultante es 24 ($-0*2^5+1*2^4+1*2^3+0*2^2+0*2^1+0*2^0 = 0+16+8+0+0+0 = 24$).

	0110	(6)
+	0101	(5)
	1011	(-5)

Tabla 4.9: *La suma de 6 (0110) con 5 (0101) usando 4 bits da 11. Sin embargo el resultado es -5 (1011). El acarreo de entrada del bit de más peso es 1. El acarreo de salida del bit de más peso es 0. Como son distintos, se ha producido desbordamiento. Esto ocurre porque 11 no es representable con 4 bits, se necesitan 5 (01011).*

$$\begin{array}{rcc} & 0100 & (4) \\ + & 1011 & (\text{-}5) \\ \hline & 1111 & (\text{-}1) \end{array}$$

Tabla 4.10: *La resta de 4 (0100) con 5 (0101) usando 4 bits da -1 (1111) y equivale a sumar 4 con -5.*

- Dividirlo entre 2, desplazándolo 1 posición hacia la derecha: 000110 >> 2 = 000011. El número resultante es 3 ($-0*2^5+0*2^4+0*2^3+0*2^2+1*2^1+1*2^0 = 0+0+0+0+2+1 = 3$).

Veamos qué ocurre con los números negativos. Si tengo el número -6 en complemento a 2 representado con 6 bits, 111010, de igual forma, puedo:

- Multiplicarlo por 4, desplazándolo 2 posiciones hacia la izquierda: 111010 << 2 = 101000. El número resultante es -24 ($-1*2^5+0*2^4+1*2^3+0*2^2+0*2^1+0*2^0 = -32+0+8+0+0+0 = -24$).

- Dividirlo entre 2, desplazándolo 1 posición hacia la derecha: 111010 >> 1 = 011101. El número resultante es ¿29? ($-0*2^5+1*2^4+1*2^3+1*2^2+0*2^1+1*2^0 = 0+16+8+4+0+1 = 29$). Nos debería dar -3. Esto pasa porque hemos perdido el signo al desplazar a la derecha.

Existen dos tipos de desplazamiento:

- *Desplazamiento lógico*: al desplazar, se insertan 0s (por la izquierda o por la derecha). Su símbolo es << para desplazar hacia la izquierda y >> para desplazar hacia la derecha. Las instrucciones en MIPS son `sll` (*shift left logical*) y `srl` (*shift right logical*).

- *Desplazamiento aritmético*: al desplazar hacia la derecha se repite el bit de más peso. Si es 0 se rellena con 0s y si es 1 se rellena con 1s. Es decir, se rellena con el signo, así que no se pierde al hacer el desplazamiento. Su símbolo es >>>. Sólo tiene sentido hacia la derecha. La instrucción en MIPS es `sra` (*shift right arithmetic*). En el ejemplo anterior, si desplazamos aritméticamente -6 una posición hacia la derecha tenemos 111010 >>> 1 = 111101. El número resultante es -3 ($-1*2^5+1*2^4+1*2^3+1*2^2+0*2^1+1*2^0 = -32+16+8+4+0+1 = -32+29 = -3$). También funciona para números positivos, en su caso el bit de signo es 0 y coincide con el desplazamiento lógico (000110 >>> 1 = 000011).

4.3.5. Multiplicación binaria con signo

Para multiplicar dos números enteros con signo A y B representados en binario con n bits, se emplea el algoritmo [4, 3] que aparece en la figura 4.4.

Por ejemplo, si queremos multiplicar 3 (A=011) por -2 (B=110) usando n=3 bits aplicando el algoritmo de recodificación de Booth, seguiríamos la secuencia de pasos mostrada en la tabla 4.11. El producto tiene 6 bits y es -6.

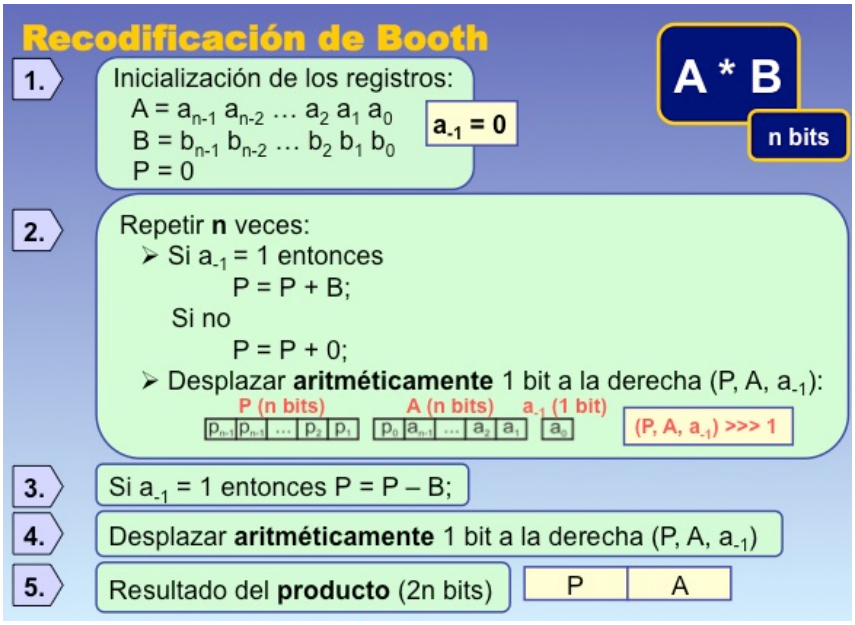

Figura 4.4: *Algoritmo para la multiplicación de dos números enteros con signo A y B usando n bits para representarlos.*

iter.		P	A	a_{-1}	Acción
0)		000	011	0	Como a_{-1} es 0: P=P+0
	+	000	011	0	
		000	011	0	
		000	011	0	>>> 1
		000	001	1	
1)		000	001	1	Como a_{-1} es 1: P=P+B
	+	110	001	1	
		110	001	1	
		110	001	1	>>> 1
		111	000	1	
2)		111	000	1	Como a_{-1} es 1: P=P+B
	+	110	000	1	
		101	000	1	
		101	000	1	>>> 1
		110	100	0	
		110	100	0	Fin de las iteraciones
		110	100	0	Como a_{-1} es 0, no sumo
		110	100	0	>>> 1
		111	010	0	

Tabla 4.11: *El producto de 3 (011) por -2 (110) usando 3 bits da -6 (111010).*

iter.		P	A	a_{-1}	Acción
0)		0000	1000	0	Como a_{-1} es 0: P=P+0
	+	0000	1000	0	
		0000	1000	0	
		0000	1000	0	>>> 1
		0000	0100	0	
1)		0000	0100	0	Como a_{-1} es 0: P=P+0
	+	0000	0100	0	
		0000	0100	0	
		0000	0100	0	>>> 1
		0000	0010	0	
2)		0000	0010	0	Como a_{-1} es 0: P=P+0
	+	0000	0010	0	
		0000	0010	0	
		0000	0010	0	>>> 1
		0000	0001	0	
3)		0000	0001	0	Como a_{-1} es 0: P=P+0
	+	0000	0001	0	
		0000	0001	0	
		0000	0001	0	>>> 1
		0000	0000	1	
		0000	0000	1	Fin de las iteraciones
		0000	0000	1	Como a_{-1} es 1, P=P-B=P+(-B)=P+C2(B)
					Como C2(B) no es representable con 4 bits C2(B)=01000 uso 5
		00000	0000	1	Extiendo el signo a P para usar 5 bits
	+	01000	0000	1	Codifico C2(B) con 5 bits (01000)
		01000	0000	1	
		0100	0000	0	>>> 1 P vuelve a tener 4 bits (el bit de signo es 0)
		0100	0000	0	

Tabla 4.12: *El producto de -8 (1000) por -8 (1000) usando 4 bits da 64 (01000000).*

En ocasiones, cuando hacemos una operación, necesitamos usar un bit más para no perder información del signo, como se muestra en la tabla 4.12. Aquí, se multiplica -8 por -8 usando 4 bits. Para hacer la resta, como el complemento a 2 de -8 no se puede representar con 4 bits, momentáneamente usamos 5 para no perder la información del signo.

4.3.6. División binaria con signo

Para dividir dos números enteros con signo A y B representados en complemento a 2 con n bits, no se utiliza ningún algoritmo [4, 3]. El procedimiento es el siguiente:

- Se calcula el signo del resultado como signo(resultado)=signo(A)*signo(B)
- Se calcula la magnitud del resultado considerando los enteros sin signo abs(A) y abs(B) y dividiéndolos mediante cualquiera de los algoritmos de división sin signo estudiados anteriormente.

- Al cociente obtenido se le asigna el signo correspondiente. Si el signo del resultado es positivo se deja como está, si es negativo, se calcula el complemento a 2 del cociente.

4.3.7. Instrucciones de carga en el MIPS

Las instrucciones de carga escriben en un registro el contenido de una posición de memoria. El registro tiene 32 bits, pero la información que se lee de memoria puede ser de 32 bits (`lw`: *load word*), de 16 bits (`lh`: *load half*) o de 8 bits (`lb`: *load byte*). En el caso de `lw`, los 32 bits de memoria se almacenan en el registro; además, los datos de tipo `word` representan enteros con signo. Sin embargo, los datos de tipo `half` y `byte`, puede representar enteros con y sin signo.

La diferencia entre las instrucciones de carga con y sin signo (`lb` y `lbu`, `lh` y `lhu`) estriba en que en caso de usar las instrucciones con signo (`lb` y `lh`), se rellena los bits del registro con el bit de signo. Es decir, si cargo un dato de 8 bits, se rellenan los 24 bits de más peso con el bit de signo y si cargo un dato de 16 bits, se rellenan los 16 bits de más peso con el bit de signo. En cambio, si uso las instrucciones sin signo (`lbu` y `lhu`) el resto de bits del registro quedan a 0.

4.4. Ejercicios

Ejercicio 1

La comparación entre dos números depende de si éstos son con signo o sin signo. Por tanto el MIPS dispone de versiones de las instrucciones de comparación para ambos tipos de datos (`slt` y `sltu`; `slti` y `sltiu`). Si `$t0` tiene el valor `0xFFFFFFFF` y `$t1` vale `2`, ¿cuánto valen `$t2` y `$t3` después de ejecutar las instrucciones `slt $t2, $t0, $t1` y `sltu $t3, $t0, $t1`?

Primero pasamos los números a binario:

- `0xFFFFFFFF` corresponde con `1111 1111 1111 1111 1111 1111 1111 1111`
- `0x00000002` corresponde con `0000 0000 0000 0000 0000 0000 0000 0010`

Calculamos su valor decimal, teniendo en cuenta que cuando trabajamos con signo (`slt`) están en complemento a 2 y si trabajamos sin signo (`sltu`) todas las potencias de 2 van con signo positivo:

- Con signo:
  - `0xFFFFFFFF` corresponde con $-1*2^{31}+1*2^{30}+1*2^{29}+1*2^{28}+1*2^{27}+1*2^{26}+1*2^{25}+1*2^{24}+1*2^{23}+1*2^{22}+1*2^{21}+1*2^{20}+1*2^{19}+1*2^{18}+1*2^{17}+1*2^{16}+1*2^{15}+1*2^{14}+1*2^{13}+1*2^{12}+1*2^{11}+1*2^{10}+1*2^{9}+1*2^{8}+1*2^{7}+1*2^{6}+1*2^{5}+1*2^{4}+1*2^{3}+1*2^{2}+1*2^{1}+1*2^{0} = -1$
  - `0x00000002` corresponde con $-0*2^{31}+0*2^{30}+0*2^{29}+0*2^{28}+0*2^{27}+0*2^{26}+0*2^{25}+0*2^{24}+0*2^{23}+0*2^{22}+0*2^{21}+0*2^{20}+0*2^{19}+0*2^{18}+0*2^{17}+0*2^{16}+0*2^{15}+0*2^{14}+0*2^{13}+$

$0*2^{12}+0*2^{11}+0*2^{10}+0*2^9+0*2^8+0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+1*2^1+0*2^0 = 2$

- Sin signo:
  - `0xFFFFFFFF` corresponde con $1*2^{31}+1*2^{30}+1*2^{29}+1*2^{28}+1*2^{27}+1*2^{26}+1*2^{25}+1*2^{24}+1*2^{23}+1*2^{22}+1*2^{21}+1*2^{20}+1*2^{19}+1*2^{18}+1*2^{17}+1*2^{16}+1*2^{15}+1*2^{14}+1*2^{13}+1*2^{12}+1*2^{11}+1*2^{10}+1*2^9+1*2^8+1*2^7+1*2^6+1*2^5+1*2^4+1*2^3+1*2^2+1*2^1+1*2^0 = 4294967294$
  - `0x00000002` corresponde con $0*2^{31}+0*2^{30}+0*2^{29}+0*2^{28}+0*2^{27}+0*2^{26}+0*2^{25}+0*2^{24}+0*2^{23}+0*2^{22}+0*2^{21}+0*2^{20}+0*2^{19}+0*2^{18}+0*2^{17}+0*2^{16}+0*2^{15}+0*2^{14}+0*2^{13}+0*2^{12}+0*2^{11}+0*2^{10}+0*2^9+0*2^8+0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+1*2^1+0*2^0 = 2$

Teniendo en cuenta lo anterior, para la instrucción `slt $t2, $t0, $t1` lo que hacemos es comparar el valor con signo de `$t0`, es decir, -1 con el valor con signo de `$t1`, es decir 2. Como -1 es menor que 2, `$t2` vale 1.

En cambio, para la instrucción `sltu $t3, $t0, $t1` comparamos el valor sin signo de `$t0`, es decir, `4294967294` con el valor sin signo de `$t1`, es decir 2. Como `4294967294` es mayor que 2, `$t3` vale 0.

4.5. Números reales

4.5.1. Representación

La notación más utilizada para números reales es la de coma flotante. Un número viene representado por 3 campos y su valor real corresponde con $ValorReal = (s) * (m) * base^{exp}$, siendo:

- *(s)*: Signo.
- *(m)*: Mantisa.
- *exp*: Exponente
- *base*: Un valor implícito de la representación utilizada.

El formato IEEE754 es el más utilizado para representar números reales. Hay representaciones de datos en precisión simple (32 bits) y en precisión doble (64 bits).

- *Precisión Simple*. Ocupa 32 bits y presenta 3 campos (Fig. 4.5):
  - *Campo Signo*: ocupa 1 bit y se corresponde con el bit de más peso. Se almacena un 0 si el número es positivo y un 1 si es negativo.

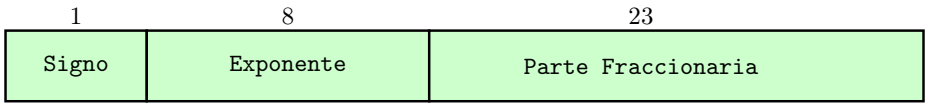

Figura 4.5: *Formato IEEE754. Precisión simple.*

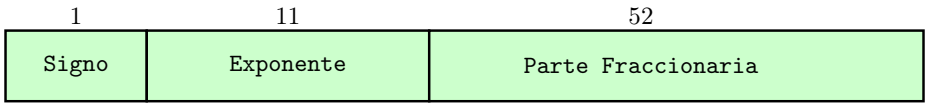

Figura 4.6: *Formato IEEE754. Precisión doble.*

- *Campo Exponente*: ocupa los siguientes 8 bits. Se representa en exceso 127.
- *Campo Parte Fraccionaria*: ocupa los 23 bits restantes. Se representa como un número binario sin signo.

- *Precisión Doble*. Ocupa 64 bits y presenta 3 campos (Fig. 4.6):
  - *Campo Signo*: ocupa 1 bit y se corresponde con el bit de más peso. Se almacena un 0 si el número es positivo y un 1 si es negativo.
  - *Campo Exponente*: ocupa los siguientes 11 bits. Se representa en exceso 1023.
  - *Campo Parte Fraccionaria*: ocupa los 52 bits restantes. Se representa como un número binario sin signo.

La manera de interpretar un número codificado en IEEE754 es la siguiente: $ValorReal = (s)1,(frac) * 2^{exp-e}$. Como vemos, IEEE754 sólo representa la parte fraccionaria, por ello las mantisas han de estar normalizadas de la forma $1,(frac)$ siendo *(frac)* la parte fraccionaria. Como el exponente se representa en exceso 127 ó 1023 según la precisión, hay que restárselo para obtener el valor real representado.

a) Representación binaria de una parte fraccionaria.

Para codificar una parte fraccionaria (lo que está a la derecha de la coma) en binario, se sigue el siguiente proceso. Se multiplica por 2 la parte fraccionaria y nos quedamos con la parte entera, repitiendo este proceso hsata que salga exacto o tantas veces como se necesite para poder redondear.

Por ejemplo, si queremos pasar el número 0,125 a base 2:
0.125 * 2 = **0**.250
0.250 * 2 = **0**.500
0.500 * 2 = **1**.000
Por tanto, 0.125 en binario es 0.**001**.

b) Interpretación de un número real binario.

Para saber a qué número corresponde un número real binario, hay que multiplicar por las potencias de 2, teniendo en cuenta que los dígitos de la parte fraccionaria se van multiplicando por potencias negativas: 2^{-1}, 2^{-2}, 2^{-3}, etc.

Por ejemplo, el número 1011,11101 se corresponde con el número $1*2^3+0*2^2+1*2^1+1*2^0+1*2^{-1}+1*2^{-2}+1*2^{-3}+0*2^{-4}+1*2^{-5} = 8+2+1+0{,}5+0{,}25+0{,}125+0{,}03125 = 11{,}90625$.

c) Redondeo.

No todos los números reales se pueden representar con un número discreto de bits. La técnica del redondeo es la siguiente: se redondea al más próximo y en caso de distancias iguales, de forma que el bit de menos peso sea par. La mitad del intervalo es un 1 seguido de 0s.

Por ejemplo:

- Si queremos redondear 10,00001 para que la parte fraccionaria tenga 2 bits, el resultado sería 10,00.
- Si queremos redondear 10,00101 para que la parte fraccionaria tenga 2 bits, el resultado sería 10,01.
- Si queremos redondear 10,00100 para que la parte fraccionaria tenga 2 bits, el resultado sería 10,00.
- Si queremos redondear 10,01100 para que la parte fraccionaria tenga 2 bits, el resultado sería 10,10.

d) Números denormales y valores especiales.

En la representación IEEE754 en precisión simple utiliza 8 bits para el campo exponente. Por tanto, con 8 bits tenemos $2^8 = 256$ enteros representables. El rango es $[00000000, 11111111] = [0, 255]$. Como hemos visto anteriormente, el exponente está en exceso 127, con lo que para saber a qué exponente corresponde hay que restar el exceso 127. Así que el rango de exponentes es $[0-127, 255-127] = [-127, 128]$.

En el estándar IEEE754 se utilizan los números -127 y 128 para casos especiales. Por tanto, el rango de exponentes utilizado en precisión simple es $[-126, 127] = [00000001, 11111110]$.

- Cuando el campo exponente vale `11111111`, se dan dos casos:
  - Que la parte fraccionaria sea todo 0s:
    - Si el signo es 0, representa a $+\infty$ (01111111100000000000000000000000).
    - Si el signo es 1, representa a $-\infty$ (11111111100000000000000000000000).
  - Que la parte fraccionaria sea distinta de 0 (puede tener desde un 1, hasta todos los bits a 1): es un Not a Number (NaN).

- Cuando el campo exponente vale `00000000`, se dan dos casos:
  - Que la parte fraccionaria sea todo 0s:
    - Si el signo es 0, representa a +0 (00000000000000000000000000000000).
    - Si el signo es 1, representa a −0 (10000000000000000000000000000000).
  - Que la parte fraccionaria sea distinta de 0 (puede tener desde un 1, hasta todos los bits a 1): es un número denormal. Los números denormales están muy próximos a 0 y su valor se interpreta como $ValorReal = (s)0,(frac) * 2^{-126}$. En este caso no hay que restarle el exceso al exponente, sino que es directamente -126. Además, la parte entera de la mantisa vale 0 en vez de 1.

4.5.2. Ejercicios

Ejercicio. Representa $-13,6$ usando el estándar IEEE754 en precisión simple.

Primero pasamos el número a binario:
La parte entera es $13_{(10} = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1101_{(2}$.
La parte fraccionaria es $0,6 = 0,100110011001...$ puesto que:
0,6*2=**1**,2
0,2*2=**0**,4
0,4*2=**0**,8
0,8*2=**1**,6
A partir de aquí se repite (la parte fraccionaria es periódica siendo el periodo 1001).
Por tanto el número en binario es 1101,100110011001...
A continuación, normalizamos la mantisa para dejarla de la forma 1,(frac):
1101,100110011001...= 1,101100110011001... * 2^3
Como la parte fraccionaria sólo tiene 23 bits, hay que redondearla. Como es mayor (100110011001...) que la mitad del intervalo (1000000000...), se redondea:
1,10110011001100110011001 100110011001... * 2^3
El número redondeado es 1,10110011001100110011010 * 2^3.
Identificamos los campos:

- Signo: 1 porque el número es negativo.
- Exponente: 3 en exceso 127 y representado con 8 bits. Por tanto 3 + 127 = 130, en binario con 8 bits: 10000010 (128+2=$2^7 + 2^1$).
- Parte fraccionaria: 23 bits en binario. Es decir, 10110011001100110011010.

Por tanto el número es 1 10000010 10110011001100110011010. Agrupando de 4 en 4, 1100 0001 0101 1001 1001 1001 1001 1010, calculamos su valor en hexadecimal: 0xC159999A.

Ejercicio. ¿A qué número corresponde 0x40A80000 usando la representación IEEE754 en precisión simple?

Primero pasamos el número a binario: 0100 0000 1010 1000 0000 0000 0000 0000.
Identificamos los campos (0 10000001 01010000000000000000000):

- Signo: es el primer bit, como vale 0, es un número positivo.
- Exponente: son los 8 bits siguientes 10000001. Su valor es $1*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+0*2^1+1*2^0 = 128+1 = 129$. Hay que quitarle el exceso 127 $129-127 = 2$. Por tanto el exponente es 2.
- Parte fraccionaria: la parte fraccionaria son los 23 bits restantes 01010000000000000000000.

Por tanto el número es el $+1,01010000000000000000000 * 2^2 = +1,0101 * 2^2 = +101,01 = 1*2^2+0*2^1+1*2^0+0*2^{-1}+1*2^{-2} = 4+1+0,25 = 5{,}25$.

Ejercicio. Representa 2^{-128} usando el estándar IEEE754 en precisión simple.

Primero reducimos el exponente del número para que sea representable: $2^{-128} = 1*2^{-128} = 0,01*2^2*2^{-128} = 0,01*2^{-126}$. Es un número denormal. A continuación identificamos los campos.:

- Signo: es el primer bit, como es un número positivo vale 0.
- Exponente: son los 8 bits siguientes. Como es un número denormal vale 00000000.
- Parte fraccionaria: la parte fraccionaria son los 23 bits restantes que se encuentran a la derecha de la coma 01000000000000000000000.

Por tanto el número (0 00000000 01000000000000000000000) en hexadecimal es (0000 0000 0010 0000 0000 0000 0000 0000) $0x00200000$.

Bibliografía

[1] Dormido, S., Canto M.A. Mira J. Delgado A.E. 2002. *Estructura y Tecnología de Computadores*. Sanz y Torres. 1.4, 1.4.1, 1.4.3, 1.4.3, 2.4, 2.4.1

[2] Murdocca, M.J., Heuring V.P. 2002. *Principios de Arquitectura de Computadores*. Prentice Hall. 1.3.1, 1.3.2, 1.3.3, 2.2, 2.3.5

[3] Patterson, D.A., Hennessy J.L. 1997. *Computer Organization and Design*. Morgan Kaufmann. (document), 1.1.1, 1.1.2, 1.2.2, 1.2.3, 2.3.1, 2.3.2, 2.3.3, 2.3.3, 2.3.3, 2.3.4, 3.1, 3.2, 3.3, 3.4, 3.5, 3.5.1, 3.6.1, 3.6.2, 3.7, 3.8, 3.8.2, 3.8.3, 3.9.2, 4.2.3, 4.2.4, 4.3.1, 4.3.5, 4.3.6

[4] Patterson, D.A., Hennessy J.L. 2000. *Estructura y Tecnología de Computadores*. Reverté. (document), 1.1.1, 1.1.2, 1.2.2, 1.2.3, 2.3.1, 2.3.2, 2.3.3, 2.3.3, 2.3.3, 2.3.4, 3.1, 3.2, 3.3, 3.4, 3.5, 3.5.1, 3.6.1, 3.6.2, 3.7, 3.8, 3.8.2, 3.8.3, 3.9.2, 4.2.3, 4.2.4, 4.3.1, 4.3.5, 4.3.6

[5] Stallings, W. 2000. *Organización y Arquitectura de Computadores*. Prentice Hall. 1.1.1, 1.1.2, 1.1.2, 1.1.3

[6] Wikipedia. 2009a. *Triodo — Wikipedia, La enciclopedia libre*. [Internet; descargado 15-febrero-2009]. 1.2.2

[7] Wikipedia. 2009b. *Válvula termoiónica — Wikipedia, La enciclopedia libre*. [Internet; descargado 15-febrero-2009]. 1.2.2