66.17 - Sistemas Digitales
Trabajo Práctico 3 - Punto Flotante

Andrew Parlane

16 de junio de 2018

Índice

1. Introducción

El objetivo de este trabajo es a diseñar, simular y sintetizar dos circuitos digitales que calculan el producto y la suma/resta de dos números de punto flotante.

Los circuitos implementados deberían soportar cualquier tamaño de punto flotante, especialmente *single precision* y *double precision* definidos en el estándar *IEEE 754*, y los tamaños usados en los archivos de prueba dado para este trabajo.

Además deberían funcionar con los cuatro modos de redondeo y números desnormalizados definido en el estándar *IEEE 754*.

2. Herramientas

Las herramientas que usé para este trabajo son:

- Questa Sim v10.2c
- Quartus II v13.0sp1
- GNU Make v4.2.1
- GCC v6.4.0

3. Implementación

Los componentes principales de mi diseño son:

- Un paquete de apoyo.
- Un componente de redondeo.
- Un componente de multiplicación.
- Un componente de suma.
- Un componente de retardo.
- Unos bancos de pruebas para simular los diseños.

3.1. Paquete de apoyo

Ese paquete tiene algunos tipos comunes, y funciones para trabar con ellos.

3.1.1. Tipos

He definido tres tipos:

- fpNumType - un enum que especifica el tipo de número (zero, NAN, infinito, denormal o normal).
- RoundingMode - un enum para el modo de redondeo (nearest, zero, infinito negativo, infinito positivo).
- fpUnpacked - un record que guarda el valor del número punto flotante desempacado.

El record *fpUnpacked* tiene:

Nombre	Tipo	Bits	Descripción
sign	std_ulogic	1	El signo del número, 0 es positivo.
bExp	unsigned	11	El exponente (con el bias).
sig	unsigned	53	El significando (con el bit implícito).
numType	fpNumType	-	El tipo del número.
emin	natural	-	El valor del exponente mínimo (con el bias) para números normalizados.
emax	natural	-	El valor del exponente máximo (con el bias) para números normalizados.
bias	natural	-	El bias del exponente bias.

Al principio he implementado *fp_helper_pkg.vhd* con genéricos para el número de bits en el exponente y en el significando. Desafortunadamente quartus no soporta paquetes genéricos, así tuve que definir un tamaño máximo para las señales. Elije 11 bits para el exponente y 53 bits para el significando desde la definición de doble precisión en el estándar *IEEE 754*. El compilador debería optimizar el diseño a solo sintetizar el número de bits necesarios.

3.1.2. Funciones

He definidos varios funciones que pueden ser partido en cuatro grupos:

- Empacar / Desempacar - Conversión entre *std_ulogic_vector* y *fpUnpacked* y al revés.
- Acceso / Comprobación - Funciones que devuelven parte del número o comprueban si el número es especial (cómo cero, o infinito, ...).
- Asignación - Devuelven un *fpUnpacked* por un valor especial (cero, infinito, NAN, máximo).
- Debug - Reportan el valor en un *fpUnpacked*.

3.2. Redondeo

Una operación sobre dos números de punto flotante puede dar un resultado que no entra en el número de bits disponible, ese significa que necesitamos redondear el resultado. El bit 'r' es el bit más significativo que no entra en el significando nuevo. El bit 's' es el *OR* de todo los demás bits. Si 'r' y 's' están cero, el significando calculado es exacto.

El estándar *IEEE 754* defina cuatro modos de redondeo:

- A cero / Truncar - Nunca redondeamos.
- A infinito positivo - Añada uno si el signo es positivo y el significando no es exacto.
- A infinito negativo - Resta uno si el signo es negativo y el significando no es exacto.
- Al más cercano - Redondea hasta el número más cercano. En el caso de un empate redondea hasta el número par.

Otra parte del redondeo es si hay un overflow el resultado puede ser infinito o el número máximo que puede ser representado (saturación). Cuál depende en el signo y en el modo de redondeo. Si redondeamos alejando del infinito para este signo, el resultado será saturación. Si redondeamos hasta el infinito para este signo, el resultado será infinito. En el caso del redondeo al más cercano, siempre redondeamos hasta infinito.

3.2.1. Señales y Parámetros

Nombre	Tipo	Bits	Descripción
Genéricos			
EBITS	natural	-	El número de bits en el exponente.
SBITS	natural	-	El número de bits en el significando.
DENORMALS	boolean	-	Si soportamos números desnormalizados o no.
Entradas			
i_clk	std_ulogic	1	El reloj.
i_sig	unsigned	SBITS	El significando a redondear.
i_bExp	signed	EBITS + 2	El exponente antes de redondear.
i_sign	std_ulogic	1	El signo del número.
i_r	std_ulogic	1	El bit 'r'
i_s	std_ulogic	1	El bit 's'
i_rm	RoundingMode	-	El modo de redondeo
Salidas			
o_sig	unsigned	SBITS	El significando redondeado.
o_bExp	unsigned	EBITS	El exponente después de redondear.
o_type	fpNumType	-	El tipo del número.

Nota que la señal *i_bExp* es *signed* y tiene *EBITS + 2*. Ese nos deja ver si el calculo ha producido un underflow o un overflow.

3.3. Multiplicación

Este componente toma dos vectores que representan dos números de punto flotante y devuelve un vector que representa el producto. Lo he implementado en una forma combinatorio, pero el resultado es guardado en un registro.

3.3.1. Algoritmo

El algoritmo de multiplicación es:

1. El exponente inicial es la suma de los exponentes de las entradas.
2. Multiplica los dos significandos usando multiplicación de enteros. He usado el operando de multiplicación (*).
3. Si el bit más significativo es un uno, desplaza el producto un bit a la derecha y añadir uno al exponente.
4. Deja caer el bit más significativo, que es un cero, y el nuevo significando es los siguiente *SBITS* bits.
5. El bit 'r' es el siguiente bit del producto.
6. El bit 's' es el *OR* de todo los demás bits.
7. Redondea el nuevo significando usando los bits 'r' y 's'.
8. El signo del resultado es el *XOR* de los signos de las entradas.
9. El resultado final es el signo calculado antes con:
   - Si uno de las entradas es NAN, el resultado es NAN.
   - Si una entrada es cero, y la otra es infinita, el resultado es NAN.
   - Si una entrada es infinita, el resultado es infinito.
   - Si una entrada es cero, el resultado es cero.
   - En los demás casos, el resultado final es el resultado del redondeo.

Para soportar números desnormilizados tenemos que cambiar el algoritmo un poco. Si la suma de los exponentes es menor que E_{MIN} desplaza el producto derecha y incrementa el exponente hasta es igual a E_{MIN}. Si multiplica un número desnormalizado por un número grande, el resultado puede ser desnormalizado con un exponente más grande que E_{MIN}. En este caso, desplaza el producto izquierda hasta que el exponente es E_{MIN} o el producto es normalizado. Este cuesta mucho en recursos.

Si quería mejorar el desempeño del diseño, implementaría un pipeline.

3.3.2. Señales y Parámetros

Nombre	Tipo	Bits	Descripción
Genéricos			
TBITS	natural	-	El número de bits en los vectores.
EBITS	natural	-	El número de bits en los exponente.
DENORMALS	boolean	-	Si soportamos números desnormalizados o no.
Entradas			
i_clk	std_ulogic	1	El reloj.
i_a	std_ulogic_vector	TBITS	La entrada A empacada.
i_b	std_ulogic_vector	TBITS	La entrada B empacada.
i_rm	RoundingMode	-	El modo de redondeo
Salidas			
o_res	std_ulogic_vector	TBITS	El resultado A*B empacada.

3.4. Suma / Resta

Este componente toma dos vectores que representan dos números de punto flotante y devuelve un vector que representa la suma. Para obtener la resta de las entradas, debería invertir el bit más significativo (el signo) de la entrada B. Lo he implementado con un pipeline de seis etapas.

3.4.1. Algoritmo

He implementado el algoritmo detallado en el apéndice J de *Computer Architecture A Quantitative Approach* de *Hennessy y Patterson*.

1. Si la entrada A es menor que la entrada B, intercambia las entradas. El exponente inicial es el exponente de la nueva A.
2. Si los signos están diferentes reemplazar el significando del B con el *twos complement*.
3. Desplaza el significando de B derecho por el exponente de la entrada A menos el exponente de la entrada B. Los bits que entran deberían estar unos si los signos de las entradas estuvieron diferentes, o ceros si no. El nuevo significando es los *SBITS* más altos del resultado. De los bits que estuvieron sacados el más significativo es el bit 'g', el siguiente es el bit 'r', y el *OR* de los demás es el bit 's'.
4. Calcula la suma de los dos significandos. Si los signos estuvieron diferentes, el bit más significativo del resultado es uno, y no había un *carry out*, reemplaza el resultado con el *twos complement* de su mismo.
5. Desplaza el resultado o derecha un bit (entrando el bit carry) o izquierda (entrando el bit 'g' y después ceros) hasta que el número es normalizado. Y actualiza el exponente como necesario.
6. Ajustar 'r' y 's'.
7. Calcula el signo nuevo usando la tabla J.12 en el apéndice de *Hennessy y Patterson*.

8. Redondea el resultado.
9. El resultado final es el signo calculado antes con:
    - Si uno de las entradas es NAN, el resultado es NAN.
    - Si una entrada es -infinito, y el otro es +infinito, el resultado es NAN
    - Si ambos entradas es cero, el resultado es cero. El signo es negativo si el modo de redondeo está a infinito negativa, y positivo si no.
    - Si una o ambos entradas es infinito, el resultado es infinito.
    - En los demás casos el resultado final es el resultado del redondeo.

Para soportar números desnormalizados el único cambio necesario es a limitar el número de bits a desplazar en el paso 5 así que el exponente no es menor que E_{MIN}.

He implementado el pipeline con una etapa para cada paso 1-4, una etapa para pasos 5 y 6, y una etapa para pasos 7, 8 y 9. Si quise mejor el diseño, comenzaría con cambiando las etapas del pipeline. Algunos de mis etapas están much más básicas que otras.

3.4.2. Señales y Parámetros

Nombre	**Tipo**	**Bits**	**Descripción**
Genéricos			
TBITS	natural	-	El número de bits en los vectores.
EBITS	natural	-	El número de bits en los exponente.
DENORMALS	boolean	-	Si soportamos números desnormalizados o no.
Entradas			
i_clk	std_ulogic	1	El reloj.
i_a	std_ulogic_vector	TBITS	La entrada A empacada.
i_b	std_ulogic_vector	TBITS	La entrada B empacada.
i_rm	RoundingMode	-	El modo de redondeo
Salidas			
o_res	std_ulogic_vector	TBITS	El resultado A+B empacada.

4. Simulación y Verificación

4.1. Bancos de Pruebas

Para verificar el funcionamiento de mis diseños uso un archivo de pruebas. Cada línea tiene dos argumentos y el resultado esperado. Paso los argumentos al componente a probar y verifico que el resultado es el valor esperado. Tuve que implementar una función que lea un número desde una línea de un archivo y vuelva un unsigned, porque el *read()* función devuelve un entero, y enteros en modelsim son limitados a 32 bits signed, o 31 bits unsigned. Así *read()* solo funciona por valores menor o igual a 2147483647.

En el caso del componente *fp_add* tuve que implementar un componente de retardo porque la unidad de suma usa un pipeline.

Para mejor manejar los diferentes tamaños de cada archivo, mis bancos de pruebas tienen genéricos que definen los tamaños *TBITS* y *EBITS*. Puedo pasar eses valores desde mi Makefile. También paso la ruta del archivo de prueba, el modo de redondeo, si soportamos números desnormalizados o no, y si deberíamos fallar si el signo del resultado no es esperado cuándo el número es cero. Ese último es porque en los archivos de prueba dados con el trabajo un cero negativo, es dada como cero positivo. También en el caso del banco de prueba de la unidad de suma, paso un genérico que dice si deberíamos invertir el bit más significativo de la entrada B para hacer pruebas de la operación *resta*.

4.2. Generación de pruebas

Después escribí un programa en C, para generar más archivos de prueba con argumentos aleatorias para las tres operaciones soportados. Tiene varios flags que me permite personalizar el archivo generado:

Flag	Descripción
num_tests	El número de pruebas a generar
no_denormal	Generar pruebas con argumentos / resultados desnormalizados
double_precision	Generar pruebas usando la precisión doble definido en el estándar *IEEE 754*. Sin este flag, genera pruebas de precisión simple.
op_multiply	Generar pruebas para la operación de multiplicación (default).
op_add	Generar pruebas para la operación de suma.
op_subtract	Generar pruebas para la operación de resta.
rounding_mode	Que modo de redondeo a usar.

Usando esto combinado con mis bancos de pruebas y un Makefile, puedo ejecutar millones de pruebas.

4.3. Coverage

He creado un programa de *systemverilog* con *covergroups* y *coverpoints* para verificar que estoy comprobando todos las combinaciones posibles. Obtengo un coverage de más de 95 %. Esto combinado de más de cien millones de pruebas ejecutados con entradas aleatorias me da mucho confianza que mis implementaciones son correctos.

5. Síntesis

He usado Quartus II para obtener un resumen de sintesis para cada uno de los siguiente unidades:

- Multiplicación de 32 bits sin números desnormalizados.
- Multiplicación de 32 bits con números desnormalizados.
- Suma de 32 bits sin números desnormalizados.
- Suma de 32 bits con números desnormalizados.

5.1. Resumen de síntesis

		Multiplicación		Suma	
		Sin denormals	Con denormals	Sin denormals	Con denormals
Ítem	**Disponible**	**Utilizado**			
Logic elements	33,216	313 (<1 %)	1,141 (3 %)	766 (2 %)	812 (2 %)
Registers	33,216	100 (<1 %)	101 (<1 %)	274 (<1 %)	274 (<1 %)
Bits de memoria	483,840	0 (0 %)	0 (0 %)	72 (<1 %)	72 (<1 %)
Global clocks	16	1 (6 %)	1 (6 %)	1 (6 %)	1 (6 %)
Frecuencia Máxima	-	49.55MHz	29.43MHz	83.96MHz	68.9MHz

6. Código

Todo el código es disponible en mi github: https://github.com/andrewparlane/fiuba6617/tree/master/TP3.