



Ciclo 2025 - 2

Arquitectura de Computadores

Teoría 3

Proyecto Parcial

Integrantes del grupo:

Flavia Marife Honores Chiroque (202310550)

Sofía Valentina Ku Paredes (202310476)

Profesor: Carlos Eduardo Williams Valencia

Octubre, 2025

Diseño e Implementación de ALU en Punto Flotante IEEE-754 para 32 y 16 bits en FPGA Basys3

Resumen

En el presente informe se describe el diseño, implementación y verificación de una Unidad Aritmético-Lógica (ALU) para operaciones en punto flotante bajo el estándar IEEE-754 para *single precision* y *half precision*. Está implementado en lenguaje Verilog y desplegado sobre FPGA Basys3.

La ALU es capaz de ejecutar las operaciones suma, resta, multiplicación y división entre operandos en formato IEEE-754, seleccionadas mediante una señal de control. Se implementó una arquitectura modular, donde cada operación está definida como un submódulo independiente, interconectado en un módulo superior encargado de la selección de la operación. Esto se validó mediante simulaciones y pruebas en hardware, verificando la correcta ejecución de las operaciones, reglas de normalización y redondeo del estándar.

Objetivos

El objetivo de este proyecto es diseñar una ALU en punto flotante capaz de realizar operaciones básicas de forma precisa y eficiente, siguiendo las etapas de normalización, alineación, redondeo y control de excepciones. Se implementa en una FPGA Basys3.

1. Objetivo general

Diseñar una ALU en HDL (verilog o VHDL) que implemente operaciones aritméticas en punto flotante de 32 y 16 bits.

Verificar la funcionalidad mediante testbenches y vectores de prueba.

Implementar el diseño en la placa Basys3 y demostrar su funcionamiento

2. Objetivos específicos

Implementar las operaciones de suma, resta, multiplicación y división en punto flotante IEEE-754 para *single* y *half precision*.

Manejar modos de redondeo: *round to nearest even*.

Detectar y reportar excepciones o *flags*, las cuales son *overflow*, *underflow*, *divide-by-zero*, *invalid operation* e *inexact*.

Soporte correcto para NaN, $\pm\text{Inf}$, ceros con signo y números denormales.

Marco Teórico

Representación IEEE-754

El estándar IEEE-754 es la representación binaria de los números reales mediante lo siguiente:

	BITS			
TIPO	TOTAL	SIGNO	EXPONENTE	MANTISA
<i>Half precision</i>	16	1	5	10
<i>Single precision</i>	32	1	8	23

El valor representado se calcula de la siguiente forma:

$$N = (-1)^{\text{signo}} \times (1 + \text{mantisa}) \times 2^{\text{exponente} - \text{bias}}$$

Donde el bias es 15 para *half precision* y 127 para *single precision*.

Operaciones para punto flotante

Para cada operación aritmética se hacen los siguientes pasos:

1. **Desempaquetado:** Extrae el signo, exponente y mantisa
2. **Alineación:** Iguala los exponentes mediante desplazamiento de la mantisa menor
3. **Operación:** Se realiza la suma, resta, multiplicación o división de la mantisa
4. **Normalización:** Ajusta el resultado para mantener el formato.
5. **Reempaquetado:** Combinación del signo, exponente y mantisa para formar el resultado final.

ALU

La ALU es el componente encargado de ejecutar operaciones aritméticas y lógicas. En este caso, se implementa para trabajar exclusivamente con operandos IEEE-754, controlando las etapas ya mencionadas y manejando condiciones especiales como *overflow*, *underflow*, NaN e infinito.

Diseño y descripción

El diseño se basa en una arquitectura jerárquica modular, donde cada componente cumple una función específica dentro del cálculo IEEE-754 para operaciones de 16 y 32 bits.

Este diseño modular nos permite ingresar, probar y reutilizar bloques de código sin alterar el resto, lo que es favorable para su escalabilidad y depuración.

Estructura general

top.v: Módulo principal, tiene interfaz FPGA más el control de flujo FSM

falu.v: Núcleo de la ALU, se controlan las operaciones

fadd.v, fsub.v, fmul.v, fdiv.v: Implementa la división en IEEE-754

fp16_32.v, fp32_16.v: Conversores entre formatos de 16 y 32 bits

Módulo top

El archivo top.v es el módulo superior que integra la ALU con la placa Basys3.

Cumple con las siguientes funciones:

1. Gestión de datos mediante una máquina de estados finita que secuencia la entrada de operandos, selección de operación y visualización del resultado.
2. Lectura de entradas a través de los switches (sw[15:0]) y botones (btnU, btnC, btnR).
3. Salida de resultados por los LEDs, que muestran el resultado bajo distintas vistas (parte baja, parte alta o flags).

Durante s6 los LEDs alternan entre las tres vistas al presionar btnR:

- Vista 1: result[15:0]
- Vista 2: result[31:16]
- Vista 3: Flags + bit de validez

Módulo falu

El módulo falu.v actúa como controlador central. Recibe los operandos, el código de operación (op_code), el modo (*half o single precision*) y el modo de redondeo. Según el código de operación, selecciona uno de los módulos y enruta su salida.

El código de operación que corresponde a cada módulo:

CÓDIGO	MÓDULO
00	ADD
01	SUB
10	MUL

11	DIV
----	-----

Los resultados y flags se almacenan en registros sincronizados con el reloj, garantizando la validez de los datos solo cuando *start* está activo.

Las banderas IEEE-754 que genera cada operación:

BIT	NOMBRE	DESCRIPCIÓN
4 MSB	invalid	Operación inválida: NaN, Inf, etc.
3	overflow	Exponente excede el rango representable.
2	underflow	Exponente cae por debajo del rango.
1	div_zero	División entre cero.
0 LSB	inexact	Resultado redondeado, no es exacto.

Módulos de conversión

Los módulos fp16_32.v y fp32_16.v permiten que la ALU opere indistintamente con punto flotante *half precision* o *single precision*.

- fp16_32: convierte un valor de 16 bits, bias 15, a formato IEEE-754 32 bits, bias 127.
- fp32_16: realiza la conversión inversa, aplicando truncamiento y control de *overflow* o *underflow*.

Ambos módulos manejan los casos especiales o flags definidos: ± 0 , Números denormales, $\pm \text{Inf}$, NaN.

(gráfico de conversión entre formatos IEEE-754: diagrama de campos s-exp-mant)

Módulos funcionales

a. fadd.v - Suma

Implementa la adición haciendo lo siguiente:

- Se separan los campos de signo, exponente y mantisa de A y B.
- Se comparan los exponentes, donde si $\text{exp}A > \text{exp}B$, la mantisa de B se desplaza a la derecha por la diferencia.
- Operación:
 - Si los signos son iguales, se realiza una suma de mantisas.
 - Si los signos son distintos, se realiza una resta entre la mayor y la menor mantisa.

- Para la normalización, si el resultado tiene acarreo, se incrementa el exponente y se ajusta la mantisa.
- Redondeo configurable mediante **round_mode** con lo solicitado *round to nearest even* al bit de guarda.
- Reconocimiento de casos especiales: NaN, Inf, ± 0 .
- Se unen los tres campos para obtener el resultado.

b. fsub.v - Resta

Realiza la sustracción mediante la reutilización del módulo de suma invirtiendo el bit de signo del segundo operando.

c. fmul.v - Multiplicación

Implementa la multiplicación de mantisas y ajusta el exponente según el bias. Incluye control de casos especiales *overflow*, *underflow*, *invalid*, *inexact*; manejo de denormales mediante desplazamiento de mantisa, normalización, redondeo y control de excepciones. También, se aplica los tres bits G-R-S (*guard*, *round*, *sticky*) para el redondeo correcto IEEE-754. Etapas del proceso:

1. Conversión y desempaqueado de operandos

Si el modo (`mode_fp`) está en 0, los operandos FP16 se convierten a FP32 para mantener la precisión interna durante el cálculo. Luego, cada operando se descompone en sus tres campos:

- Signo: bit 31
- Exponente: bits [30:23]
- Mantisa: bits [22:0]

También se detectan valores especiales como cero, infinito y NaN.

2. Detección de casos especiales

Antes de realizar la multiplicación, se manejan condiciones definidas por el estándar IEEE-754:

- NaN propagado si alguno de los operandos es NaN.
- $\text{Inf} * 0$ genera NaN (flag de operación inválida).
- $\text{Inf} * X$ produce Inf (con flag de overflow).
- $0 * X$ produce 0.

En estos casos, el módulo devuelve el resultado especial correspondiente y activa las banderas de excepción pertinentes.

3. Cálculo del signo, exponente y mantisa

- El signo del resultado se obtiene como $sign = signA \text{ XOR } signB$.
- Los exponentes se desbiansan (restando 127 o 15 según el formato) y se suman.
- Las mantisas se multiplican incluyendo el bit implícito '1', generando un producto de 48 bits.

4. Normalización y ajuste del exponente

Dependiendo del rango del producto:

- Si el resultado se encuentra en el rango $[1,2)$, se conserva tal cual.
- Si se encuentra en $[2,4)$, se desplaza una posición a la derecha y se incrementa el exponente.
- Si el producto es menor a 1, se desplaza a la izquierda hasta que el bit más significativo sea 1, ajustando el exponente en consecuencia.

También se manejan casos de subnormales, cuando el exponente resultante cae por debajo de 1, mediante desplazamientos a la derecha y cálculo del bit sticky con una máscara lógica sintetizable.

5. Redondeo IEEE-754 (*round to nearest even*)

Se aplican los bits *Guard* (G), *Round* (R) y *Sticky* (S) para determinar si la mantisa debe incrementarse.

Si ocurre un *carry* tras el redondeo, se desplaza nuevamente la mantisa y se ajusta el exponente.

6. Control de excepciones y ensamblaje final

Se construye el resultado final en formato FP32 o FP16 según el modo. Se activan las banderas IEEE-754:

- flags[4] = Invalid Operation (NaN o Inf×0)
- flags[3] = Overflow
- flags[2] = Underflow
- flags[0] = Inexact (por redondeo)
- Finalmente, se empaqueta el número con $\{sign, exponent, mantissa\}$ para su salida.

d. fdiv.v - División

Divide las mantisas y ajusta el exponente. Detecta la división entre cero, Inf/NaN, y utiliza lógica de normalización y redondeo "*round to nearest*"

even”, y actualiza las banderas de excepción correspondientes. Etapas del proceso:

1. Conversión y desempaqueado de operandos

Cuando el modo *mode_fp* está desactivado, los operandos en formato FP16 se convierten internamente a FP32 para mantener la precisión del cálculo. Luego, cada operando se descompone en sus tres campos fundamentales:

- Signo (S): bit 31
- Exponente (E): bits [30:23]
- Mantisa (M): bits [22:0]

También se detectan valores especiales según el estándar:

- Cero: exponente = 0, mantisa = 0
- Infinito: exponente=255, mantisa=0
- NaN: exponente = 255, mantisa \neq 0

Estos valores se usan para decidir si la operación debe realizarse normalmente o manejarse como caso especial.

2. Detección y manejo de casos especiales

Antes de realizar la división, se evalúan las combinaciones especiales definidas por el estándar IEEE-754. El módulo genera directamente el resultado y las banderas.

OPERACIÓN	RESULTADO	FLAG
NaN, X o X, NaN	NaN	invalid
Inf / Inf	NaN	invalid
Inf / finito	+ - Inf	-
Finito / Inf	+ - 0	-
0 / 0	NaN	invalid
Finito / 0	+ - Inf	div_zero

Si *mode_fp* = 0, el resultado especial también se convierte automáticamente al formato FP16.

3. Cálculo del signo, exponente y mantisa

Si no se trata de un caso especial, se realiza la división normal:

- Signo del resultado: $sign = signA \text{ XOR } signB$
- Para el exponente no sesgado, se elimina el bias para ambos exponentes:

$$expA = e_a - 127$$

$$expB = e_b - 127$$

$$exp_{res} = expA - expB$$
- Mantisas normalizadas: Si el exponente es 0, el bit implícito es 0. Caso contrario, se agrega el bit implícito para formar una mantisa de 24 bits
- Se forma el numerador extendido $q_extended$ para conservar la precisión y se realiza la división entera

4. Normalización del resultado

Tras la división, la mantisa se ajusta para mantener el formato normalizado 1.x:

- Si el bit más significativo (MSB) es 0, la mantisa se desplaza hacia la izquierda hasta que aparezca un 1, ajustando el exponente: $exp_res_unb = exp_diff - shift_count$
- Los tres bits menos significativos (LSB) se guardan como *guard g*, *round rr* y *sticky ss*, para el redondeo posterior.

También se detectan condiciones extremas:

- Si $exp_res_unb > 127$, ocurre *overflow*, el resultado se satura a $\pm \text{inf}$.
- Si $exp_res_unb < -126$, ocurre *underflow*, el resultado se aproxima a 0 o número subnormal.

5. Redondeo IEEE-754 (Round to Nearest, Even)

Se aplican los tres bits de redondeo *guard g*, *round rr* y *sticky ss*:

- Si $g = 1$ y $rr = 1$ o $s = 1$, o si $g = 1$ y el LSB de la mantisa es 1, se incrementa la mantisa.
- Si este incremento genera un *carry*, se renormaliza desplazando la mantisa y aumentando el exponente.

Se activa la bandera de *inexact* si hubo redondeo o si hubo residuo no nulo.

6. Conversión final y establecimiento de banderas

El resultado se reconstruye combinando los campos $\{sign, exponent, mantissa[22:0]\}$.

Si el modo es FP16, el valor final se convierte mediante el módulo *fp32_16*. También se verifica si la conversión causa *overflow* o *underflow*.

Si el modo es FP32, se entrega el valor directamente.

Finalmente, se actualizan las banderas IEEE-754 según el tipo de excepción detectada:

- flags[4] = Invalid Operation: operaciones con NaN, 0/0 o Inf/Inf.
- flags[3] = Overflow: resultado saturado a infinito.
- flags[2] = Underflow: resultado subnormal o muy pequeño.
- flags[1] = Divide by Zero: divisor igual a cero, numerador distinto de 0.
- flags[0] = Inexact: resultado redondeado o residuo distinto de cero.

Módulo de control FSM y pruebas en FPGA

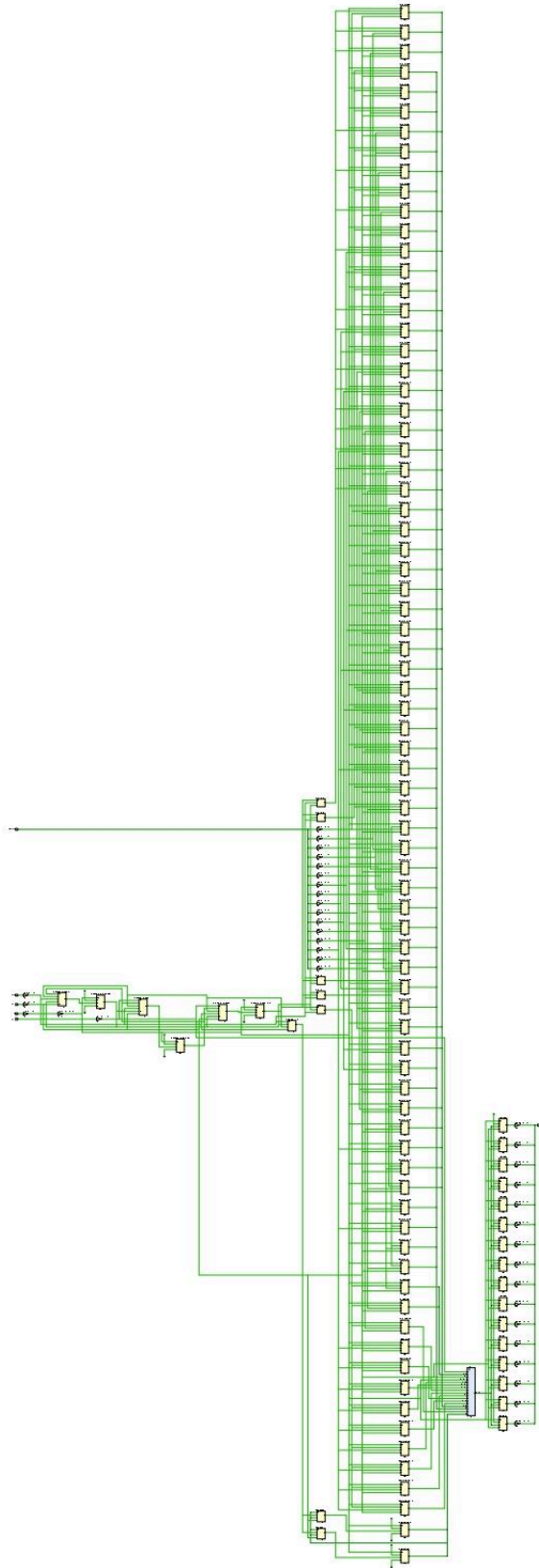
El sistema se implementó en Basys3, mediante Xilinx Vivado. Se asignaron los pines en el archivo tipo xdc de la siguiente manera:

COMPONENTE	DESCRIPCIÓN	PIN
sw[15:0]	Entradas A y B	Todos los switches
btn[1:0]	Selección de operación <i>op_code</i>	U18, T17
led[15:0]	Resultado binario de salida	Todas las leds
clk	Reloj	W5

El ingreso de datos se realiza mediante los switches, y los botones seleccionan la operación deseada. El resultado se observa en los LEDs, mostrando los bits del número IEEE-754 resultante.

La síntesis y descarga del bitstream se realizaron desde Vivado, validando el correcto funcionamiento con diversas combinaciones de entradas.

Diagrama de diseño



Resultados y validación

Nota: en algunos casos el resultado se evidencia en el siguiente flanco de clk debido a que la lógica combinacional toma más tiempo en procesar que el periodo que establecimos.

Mediante simulaciones se verificaron los siguientes casos:

a. ADD

i. 32 BITS

1. Números normales

```
//5555.55 + 2000.2 = 7555.75 (0x479391e0) -> 7555.7495 0x45ec1dff  
start = 1; op_code = 2'b00;  
op_a = 32'h45ad9c66; op_b = 32'h44fa0666; #10  
op_a = 32'h00000000; op_b = 32'h00000000; #10
```

Name	Value	0.000 ns	5.000 ns	10.000 ns	15.000 ns	20.000 ns	25.000 ns
clk	0						
rst	0						
start	1						
op_a[31:0]	00000000		00000000	45ad9c66		00000000	
op_b[31:0]	00000000		00000000	44fa0666		00000000	
op_c[1:0]	0			0			
mode_fp	1						
rou_ode	0						
result[31:0]	45ec1dff			00000000		45ec1dff	
valid_out	1						
flags[4:0]	01			00			01

2. A + (-A)

```
// +0 + -0 = +0 (0x00000000) check  
start = 1; op_code = 2'b00;  
op_a = 32'h00000000; op_b = 32'h80000000; #10  
op_a = 32'h00000000; op_b = 32'h00000000; #10
```

Name	Value	0.000 ns	5.000 ns	10.000 ns	15.000 ns	20.000 ns	25.000 ns
clk	0						
rst	0						
start	1						
op_a[31:0]	00000000			00000000			
op_b[31:0]	00000000		00000000	80000000		00000000	
op_c[1:0]	0			0			
mode_fp	1						
rou_ode	0						
result[31:0]	00000000			00000000			
valid_out	1						
flags[4:0]	00			00			

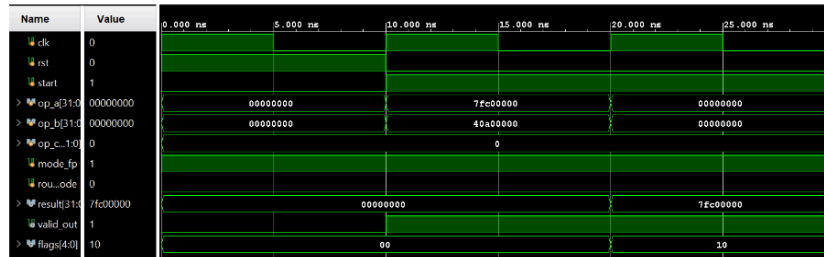
3. Caso especial: operaciones con Inf

```
// +Inf + (-Inf) = NaN (0x7FC00000) check  
start = 1; op_code = 3'b000;  
op_a = 32'h7F800000; op_b = 32'hFF800000; #10  
op_a = 32'h00000000; op_b = 32'h00000000; #10
```

Name	Value	0.000 ns	5.000 ns	10.000 ns	15.000 ns	20.000 ns	25.000 ns
clk	0						
rst	0						
start	1						
op_a[31:0]	00000000		00000000	7F800000		00000000	
op_b[31:0]	00000000		00000000	FF800000		00000000	
op_c[1:0]	0			0			
mode_fp	1						
rou_ode	0						
result[31:0]	7fc00000			00000000		7fc00000	
valid_out	1						
flags[4:0]	10			00			10

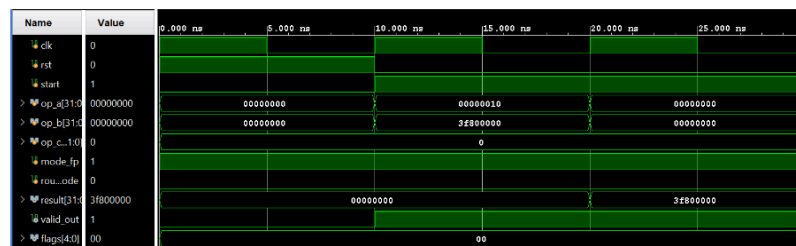
4. Caso especial: operaciones con NaN

```
// NaN + 5.0 = NaN (0x7FC00000) check
start = 1; op_code = 2'b00;
op_a = 32'h7FC00000; op_b = 32'h40A00000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



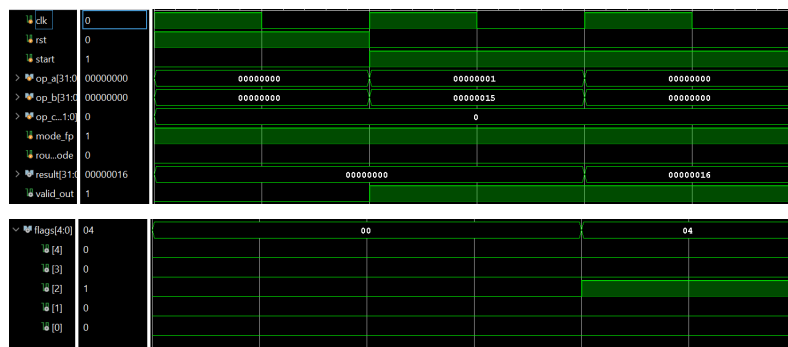
5. Caso específico: denormal + normal

```
// 2.2e-44 (denormal) + 1 (normal) = ~1 (0x3F800000) check
start = 1; op_code = 2'b00;
op_a = 32'h00000010; op_b = 32'h3F800000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



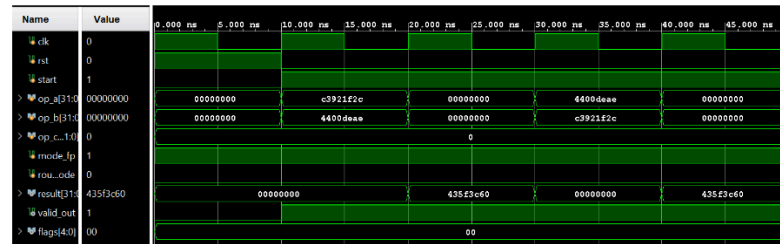
6. Caso específico: denormal + denormal

```
// 1e-45 + 3e-44 = 3.1e-44 (0x00000016)
start = 1; op_code = 2'b00;
op_a = 32'h00000001; op_b = 32'h00000015; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



7. Conmutatividad: A + B = B + A

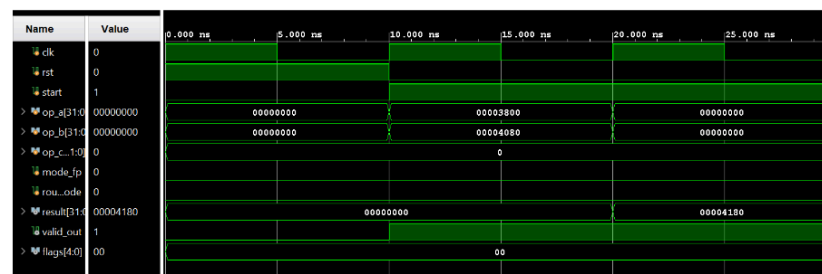
```
// conmutatividad (a+b == b+a) -> -292.24353 (0xc3921f2c)+ 515.4794 (0x4400deae) = 223.23587 (0x435f3c62)
start = 1; op_code = 2'b00;
op_a = 32'hc3921f2c; op_b = 32'h4400deae; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
op_a = 32'h4400deae; op_b = 32'hc3921f2c; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
/*
```



ii. 16 BITS

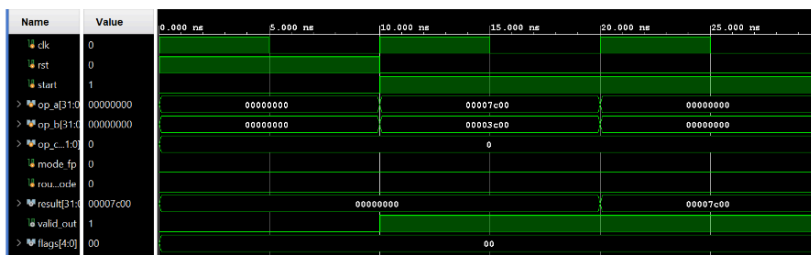
1. normal + normal

```
// 0.5 (0x3800) + 2.25 (0x4080) ≈ 2.75 (0x00004180) check
start = 1; op_code = 2'b00;
op_a = 32'h00003800; op_b = 32'h00004080; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```

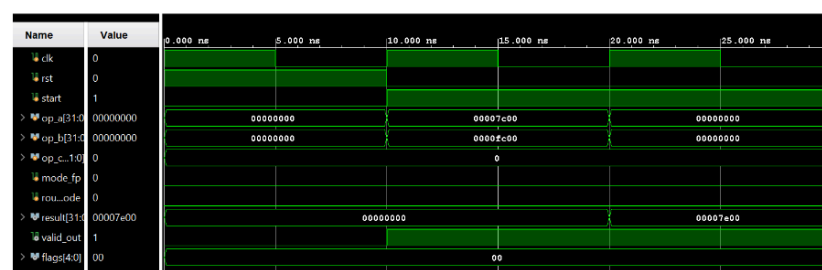


2. caso especial: operaciones con infinito

```
// +Inf (0x7C00) + 1.0 (0x3C00) = +Inf
start = 1; op_code = 2'b00;
op_a = 32'h00007C00; op_b = 32'h00003C00; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



```
//+inf (0x7C00) + -Inf (0xFC00) = NaN (0x7E00)
start = 1; op_code = 2'b00;
op_a = 32'h00007C00; op_b = 32'h0000FC00; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



3. caso especial: operaciones con NaN

```
//NaN (0x7E00) + -2.0 (0x4000) = NaN
start = 1; op_code = 2'b00;
op_a = 32'h00007E00; op_b = 32'h0000c000; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

Name	Value	0.000 ns	5.000 ns	10.000 ns	15.000 ns	20.000 ns	25.000 ns
clk	0						
rst	0						
start	1						
> op_a[31:0]	00000000		00000000	00007e00		00000000	
> op_b[31:0]	00000000		00000000	0000c000		00000000	
> op_c[1:0]	0			0			
mode_fp	0						
rou_ode	0						
> result[31:0]	00007e00		00000000			00007e00	
valid_out	1						
> flags[4:0]	00			00			

4. caso específico: denormal + normal

```
// 0.00006(0x0001) + 1.0 (0x3C00) = 1.00006 (0x3C00)
start = 1; op_code = 2'b00;
op_a = 32'h00000001; op_b = 32'h00003C00; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

Name	Value	0.000 ns	5.000 ns	10.000 ns	15.000 ns	20.000 ns	25.000 ns
clk	0						
rst	0						
start	1						
> op_a[31:0]	00000000		00000000	00000001		00000000	
> op_b[31:0]	00000000		00000000	00003c00		00000000	
> op_c[1:0]	0			0			
mode_fp	0						
rou_ode	0						
> result[31:0]	00003c00		00000000			00003c00	
valid_out	1						
> flags[4:0]	00			00			

5. caso específico: denormal + denormal

```
// 0.000000834465 + -5.9604645e-8 = -7.74860355e-7 0x800d
start = 1; op_code = 2'b00;
op_a = 32'h0000000e; op_b = 32'h00008001; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

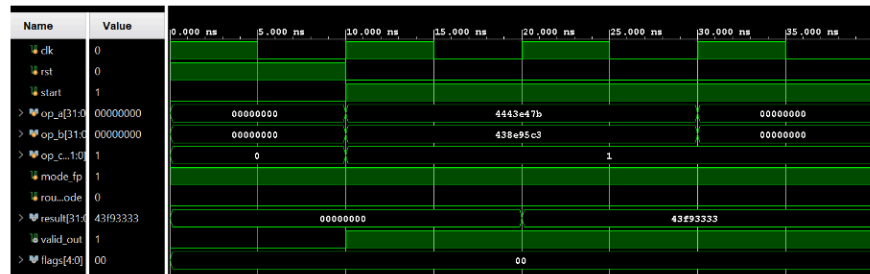
Name	Value	0.000 ns	5.000 ns	10.000 ns	15.000 ns	20.000 ns	25.000 ns
clk	0						
rst	0						
start	1						
> op_a[31:0]	00000000		00000000	0000000e		00000000	
> op_b[31:0]	00000000		00000000	00008001		00000000	
> op_c[1:0]	0			0			
mode_fp	0						
rou_ode	0						
> result[31:0]	00000000		00000000				
valid_out	1						
> flags[4:0]	00			00			

b. SUB

i. 32 BITS

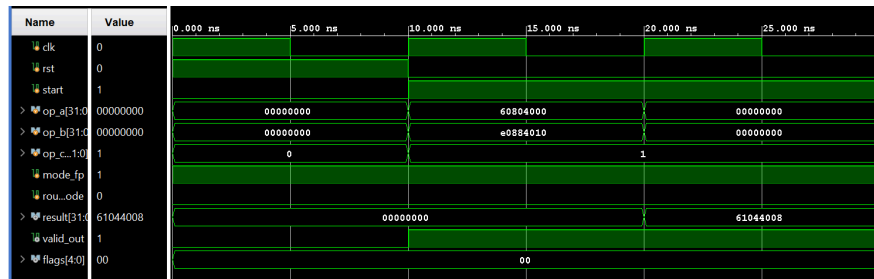
1. Normal - normal

```
//783.57 - 285.17 = 498.4 (0x43f93333) check
start = 1; op_code = 2'b01;
op_a = 32'h4443e47b; op_b = 32'h438e95c3; #20
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



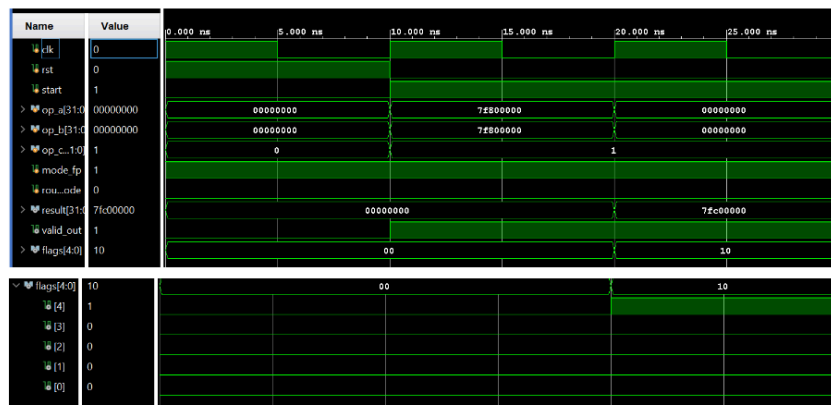
2. A - (-B)

```
// 7.393109e19 - -7.854292e19 = 1.5247401e+20 (0x61044008)
start = 1; op_code = 3'b001;
op_a = 32'h60804000; op_b = 32'h0884010; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



3. Caso especial: operaciones con Inf

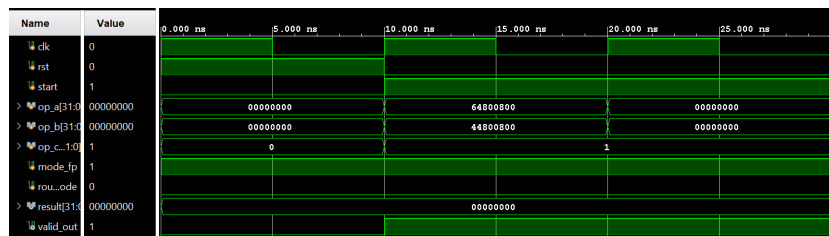
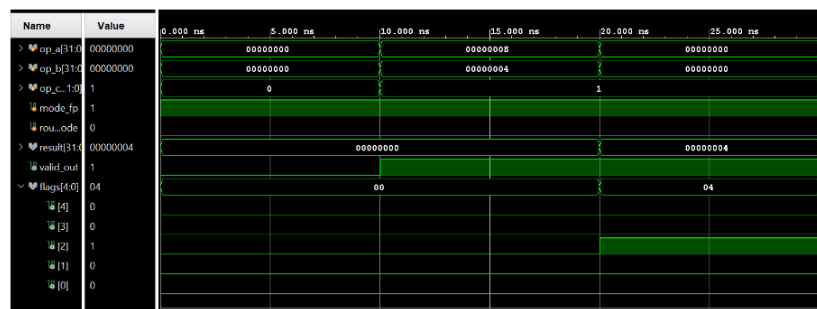
```
// Inf - Inf = NaN (0x7FC00000)
start = 1; op_code = 3'b001;
op_a = 32'h7F800000; op_b = 32'h7F800000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



4. Caso especial: operaciones con NaN

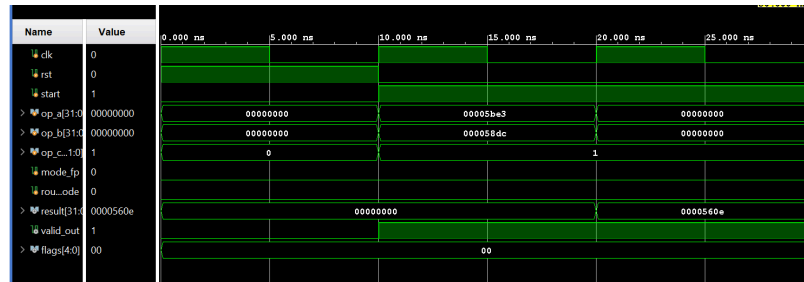
Name	Value	0.000 ns	5.000 ns	10.000 ns	15.000 ns	20.000 ns	25.000 ns
clk	0						
rst	0						
start	1						
> op_a[31:0]	00000000	00000000		7fc00000		00000000	
> op_b[31:0]	00000000	00000000		44800800		00000000	
> op_c[1:0]	1	0				1	
mode_fp	1						
rou_ode	0						
> result[31:0]	7fc00000	00000000				7fc00000	
valid_out	1						
> flags[4:0]	10	00				10	

```
// 1.8894078e22 - 1024.25 = 0
start = 1; op_code = 3'b001;
op_a = 32'h64800800; op_b = 32'h44800800; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```

[illegible]

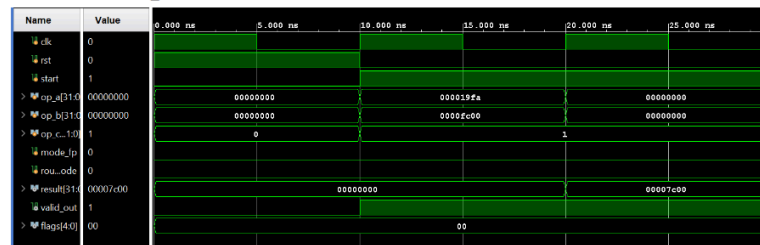
1. normal - normal

```
// 252.375 (0x5be3) - 155.55 (0x58dc) = 96.825 (0x560d)
start = 1; op_code = 2'b01;
op_a = 32'h0005be3; op_b = 32'h00058dc; #10
op_a = 32'h0000000; op_b = 32'h0000000; #10
```



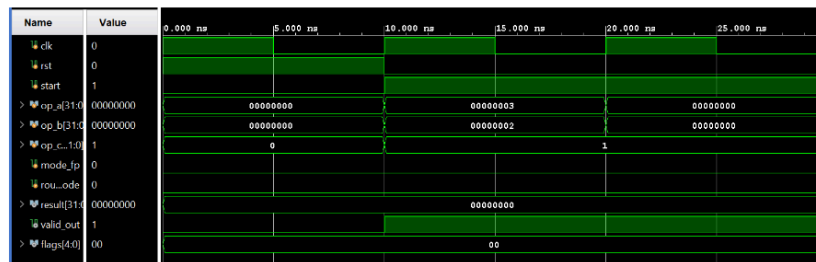
2. denormal - denormal

```
// 0.00000017881393 (0x0003) - 0.00000011920929 (0x0002) = 0 -> 0x0001
start = 1; op_code = 2'b01;
op_a = 32'h00000003; op_b = 32'h00000002; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



3. caso especial: operaciones con Inf

```
// 0.0029182434 (0x19fa) - -Inf (0xFC00) = +inf (0x7c00)
start = 1; op_code = 2'b01;
op_a = 32'h000019fa; op_b = 32'h0000FC00; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

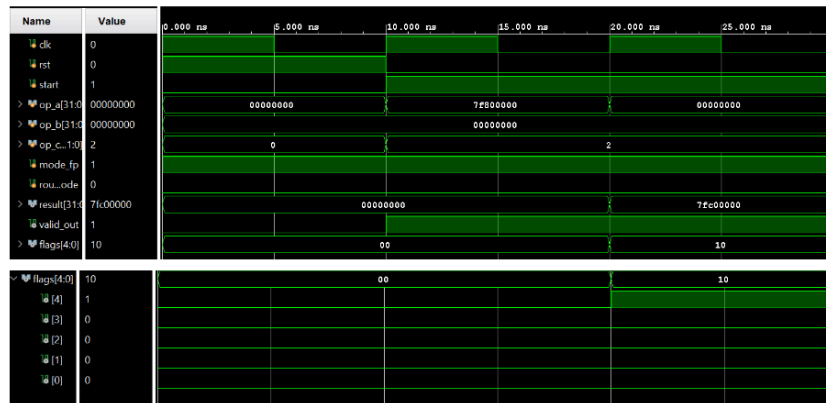


4. caso especial: operaciones con NaN

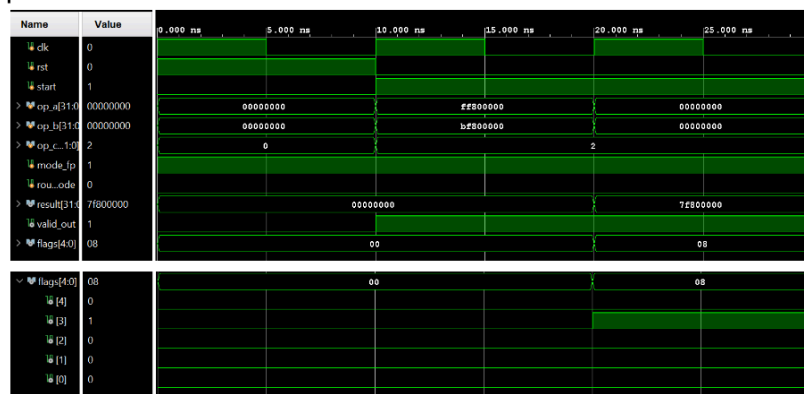
```
// NaN (0x7e00) - -0.060943604 (0xabcd) = NaN
start = 1; op_code = 2'b01;
op_a = 32'h00007e00; op_b = 32'h0000abcd; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



```
// +Inf * 0 = 0
start = 1; op_code = 3'b010;
op_a = 32'h7F800000; op_b = 32'h00000000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



```
// (-Inf) * (-1) = +Inf
start = 1; op_code = 3'b010;
op_a = 32'hFF800000; op_b = 32'hBF800000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



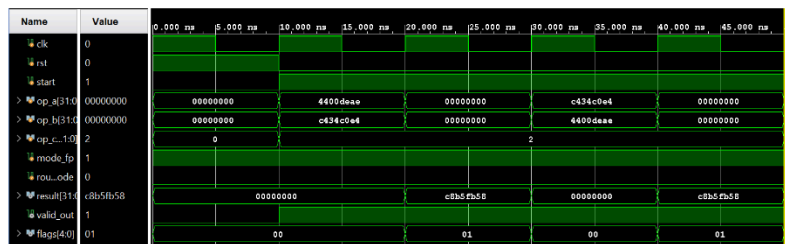
3. Caso especial: operaciones con NaN

```
// 5 * NaN = NaN
start = 1; op_code = 3'b010;
op_a = 32'h40a00000; op_b = 32'h7F800000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



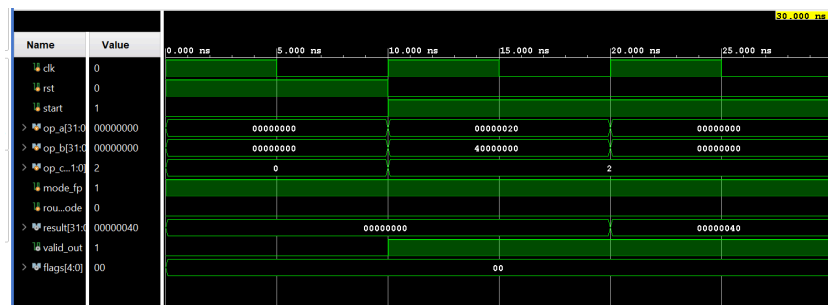
4. Conmutatividad : $a*b = b*a$

```
// conmutatividad (a*b == b*a) -> 515.4794 (0x4400deae) * -723.0139 (0xc43c0e4) = 372698.771364 (0xc8b5fb58)
start = 1; op_code = 3'b010;
op_a = 32'h4400deae; op_b = 32'hc43c0e4; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
op_a = 32'hc43c0e4; op_b = 32'h4400deae; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



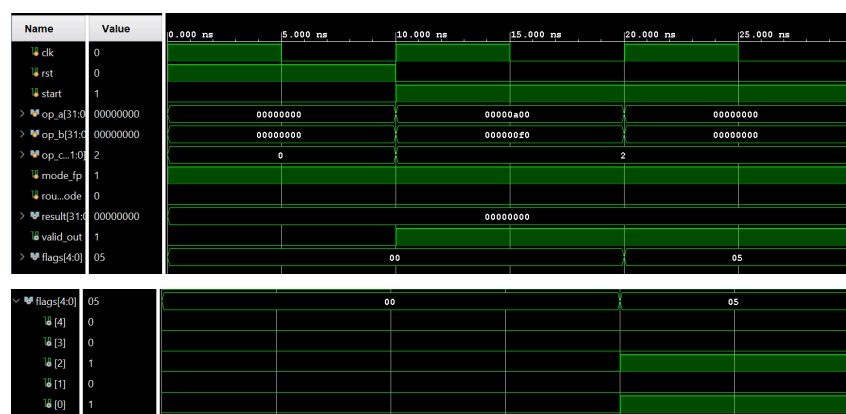
5. Caso específico: denormal * normal

```
// 4.5e-44 (denormal) * 2 = 9e-44 (0x00000040)
start = 1; op_code = 3'b010;
op_a = 32'h00000020; op_b = 32'h40000000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



6. Caso específico: denormal * denormal

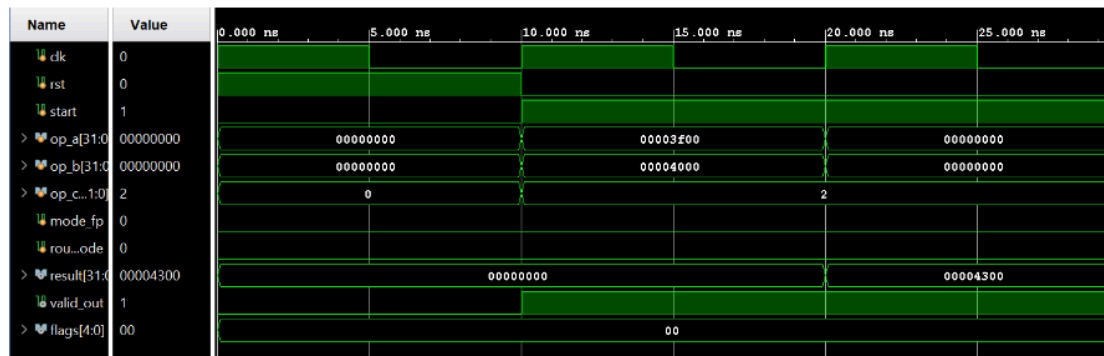
```
// 3.587e-42 * 3.36e-43 = 0
start = 1; op_code = 3'b010;
op_a = 32'h00000a00; op_b = 32'h000000f0; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



ii. 16 BITS

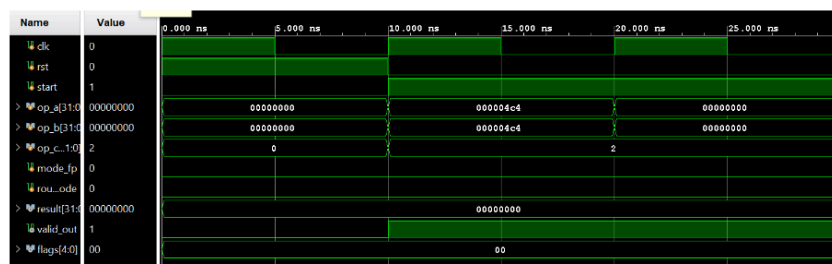
1. normal * normal

```
// 1.75 * 2 = 3.5 (0x00004300) CHECK
start = 1; op_code = 2'b10;
op_a = 32'h0003f00; op_b = 32'h00004000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```



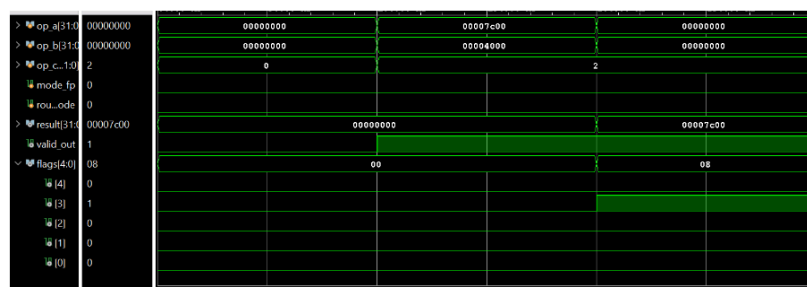
2. denormal * denormal

```
//1e-5 (0x04C4) * 1e-5 (0x04C4) ≈ 1e-10 (underflow → 0x0000)
start = 1; op_code = 2'b10;
op_a = 32'h000004C4; op_b = 32'h000004C4; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



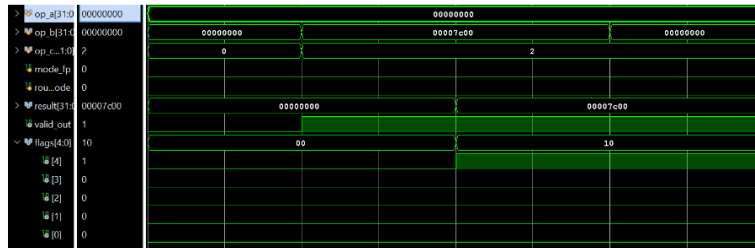
3. caso especial: operaciones con Inf

```
//inf * 2.0 = inf
start = 1; op_code = 2'b10;
op_a = 32'h00007C00; op_b = 32'h00004000; #10;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



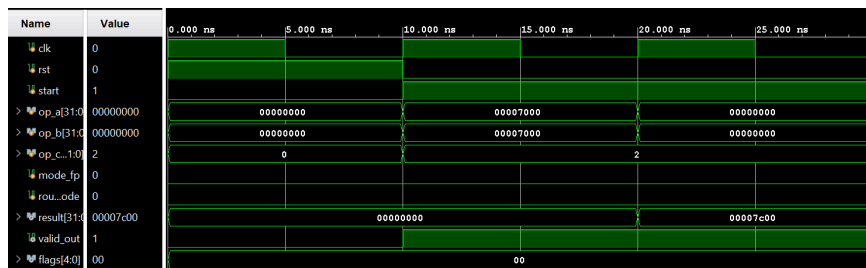
4. caso especial: operaciones con NaN

```
//0.0 (0x0000) * inf (0x7c00) = NaN (0x7E00)
start = 1; op_code = 2'b10;
op_a = 32'h00000000; op_b = 32'h00007c00; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
/*
```



5. overflow

```
// 8192 (0x7000) * 8192 (0x7000) = 67108864 -> overflow inf 0x7c00
start = 1; op_code = 2'b10;
op_a = 32'h00007000; op_b = 32'h00007000; #10
op_a = 32'h00000000; op_b = 32'h00000000; #10
```

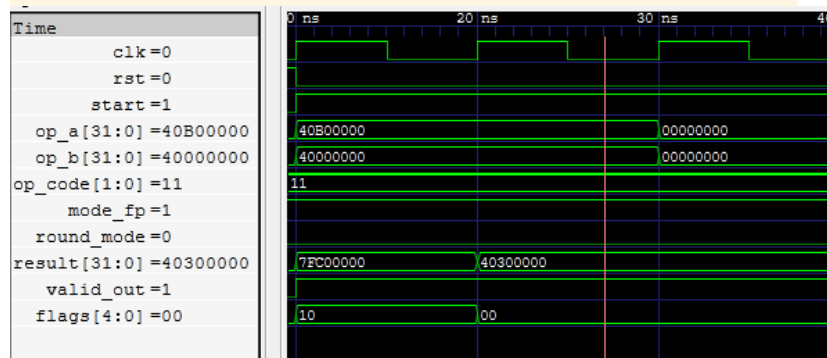


d. DIV

i. 32 BITS

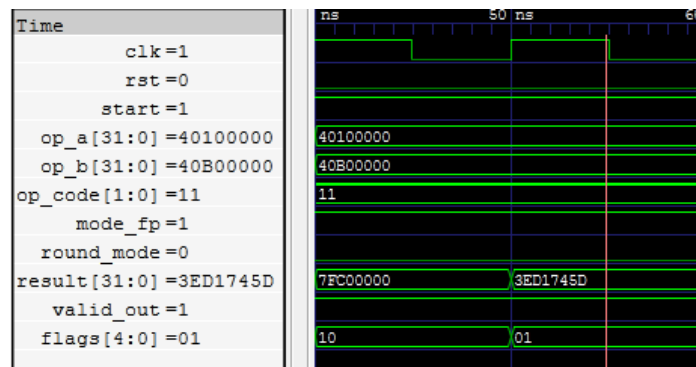
1. decimal / natural

```
// 1) 5.5 (0x40B00000) / 2 (0x40000000) = 2.75 (0x40300000)
// NO FLAG
start=1; op_code = 2'b11;
op_a = 32'h40B00000; op_b = 32'h40000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



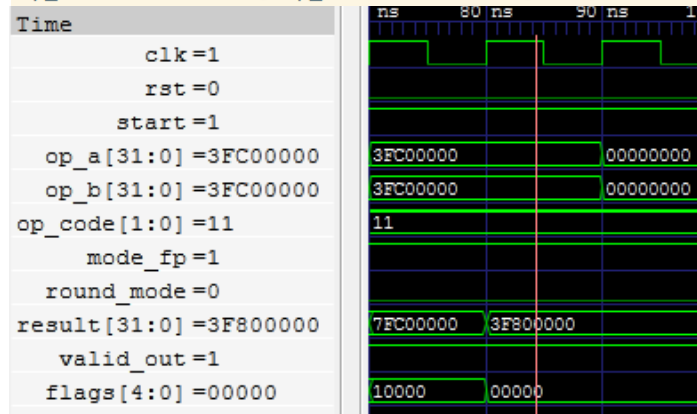
2. decimal / decimal

```
// 2) 2.25 (0x40100000) / 5.5 (0x40B00000) = 0.4090909 (0x3ED1745D)
// FLAG: inexact
start=1; op_code = 2'b11;
op_a = 32'h40100000; op_b = 32'h40B00000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



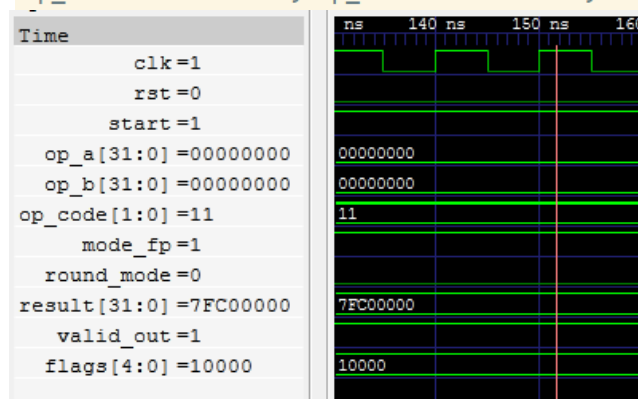
3. decimal / sí mismo

```
// 3) 1.5 (0x3FC00000) / 1.5 (0x3FC00000) = 1.0 (0x3F800000)
// NO FLAG
start= 1; op_code = 2'b11;
op_a = 32'h3FC00000; op_b = 32'h3FC00000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



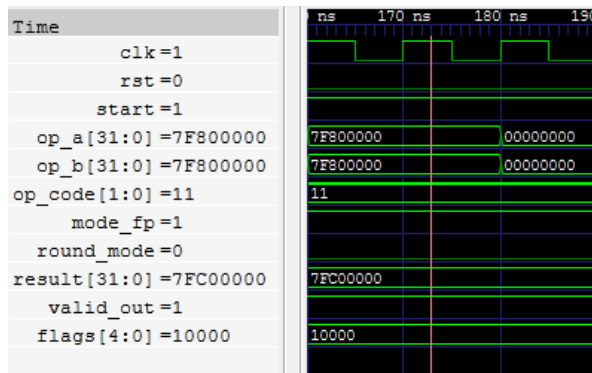
4. caso específico: NaN -> cero / cero

```
// 4) 0 (0x00000000) / 0 (0x00000000) -> NaN (0x7FC00000)
// FLAG: invalid
op_a = 32'h00000000; op_b = 32'h00000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



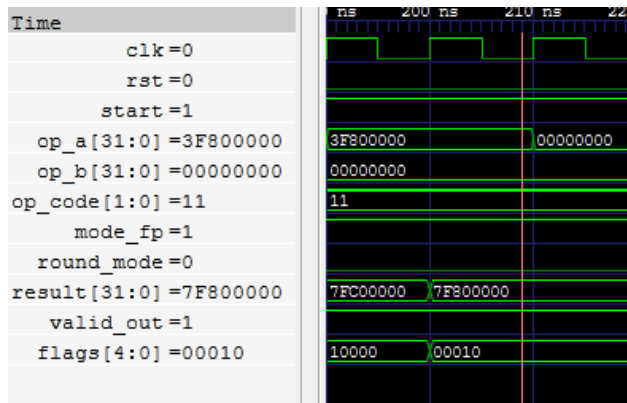
5. caso específico: inf / inf


```
// 5) Inf (0x7F800000) / Inf (0x7F800000) → NaN (0x7fc00000)
// FLAG: invalid
op_a = 32'h7F800000; op_b = 32'h7F800000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



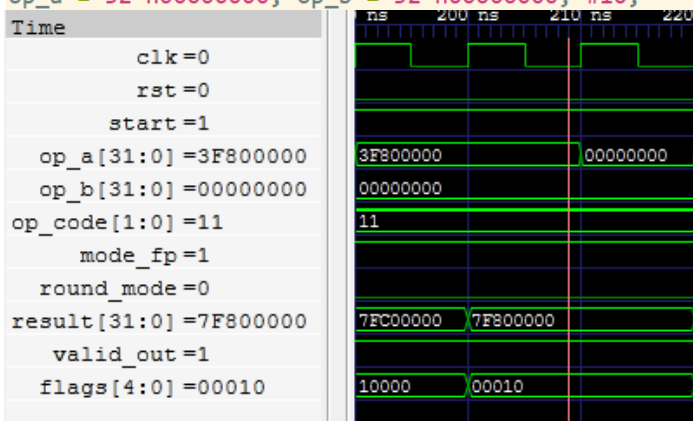
6. caso específico: número / 0

```
// 6) 1.0 (0x3F800000) / 0.0 (0x00000000) = +Inf (0x7F800000)
// FLAG: div_zero
op_a = 32'h3F800000; op_b = 32'h00000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



7. caso específico: -número / 0

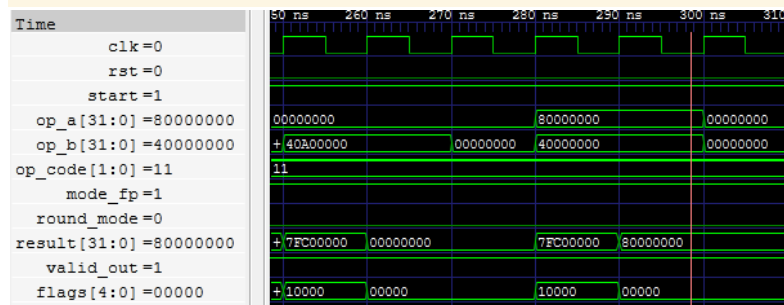
```
// 7) -1.0 (0xBF800000) / 0.0 (0x00000000) = -Inf (0xFF800000)
// FLAG: div_zero
op_a = 32'hBF800000; op_b = 32'h00000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



8. ± 0 / número

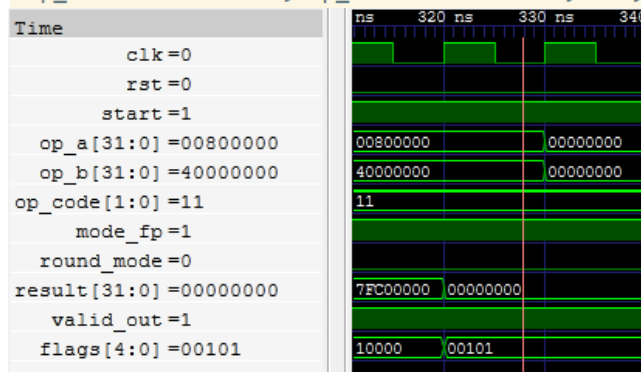
```
// 8) 0 (0x00000000) / 5.0 (0x40A00000) = +0 (0x00000000)
// NO FLAG
op_a = 32'h00000000; op_b = 32'h40A00000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;

// 9) -0 (0x80000000) / 2.0 (0x40000000) = -0 (0x80000000)
// NO FLAG
op_a = 32'h80000000; op_b = 32'h40000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



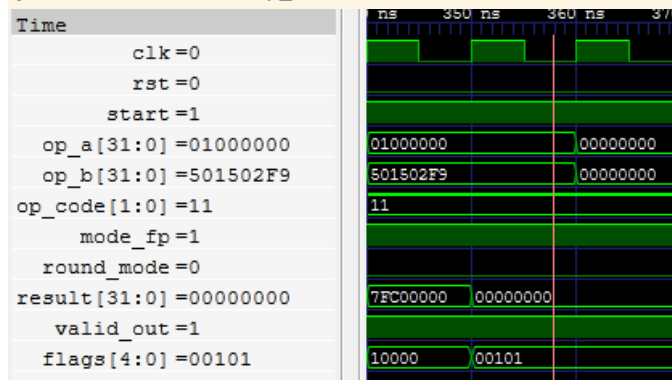
9. caso denormal pequeño

```
// 10) 1.1754943e-38 (0x00800000) / 2.0 (0x40000000) = 0x?
// FLAG: underflow + inexact
op_a = 32'h00800000; op_b = 32'h40000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



10. Caso denormal grande

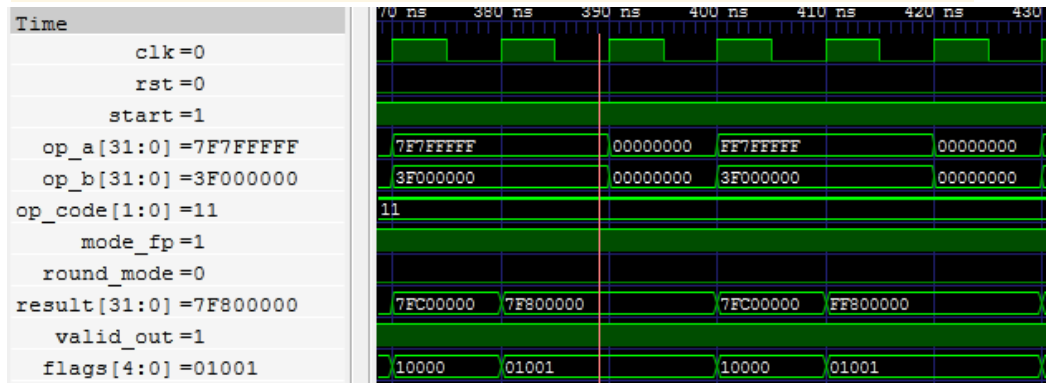
```
11) 2.3509887e-38 (0x01000000) / 1e+10 (0x501502F9) = 0x?
FLAG: underflow + inexact
.a = 32'h01000000; op_b = 32'h501502F9; #20;
.a = 32'h00000000; op_b = 32'h00000000; #10;
```



11. caso específico: overflow

```
// 12) 3.4028234e+38 (0x7F7FFFFFFF) / 0.5 (0x3F000000) = +Inf (0x7F800000)
// FLAG: overflow + inexact
op_a = 32'h7F7FFFFFFF; op_b = 32'h3F000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

```
// 13) -3.4028234e+38 (0x7F7FFFFFFF) / 0.5 (0x3F000000) = -Inf (0xFF800000)
// FLAG: overflow + inexact
op_a = 32'hFF7FFFFFFF; op_b = 32'h3F000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



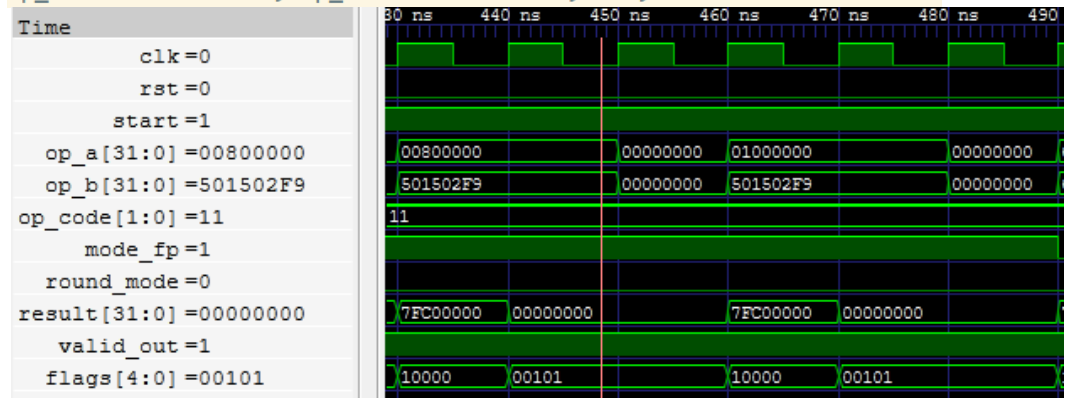
12. caso específico: underflow

```
// 14) 1.1754943e-38 (0x00800000) / 1e+10 (0x501502F9) = 0x0....?
// FLAG: underflow + inexact
```

```
op_a = 32'h00800000; op_b = 32'h501502F9; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

```
// 15) 2.3509887e-38 (0x01000000) / 1e+10 (0x501502F9) = underflow
// FLAG: underflow + inexact
```

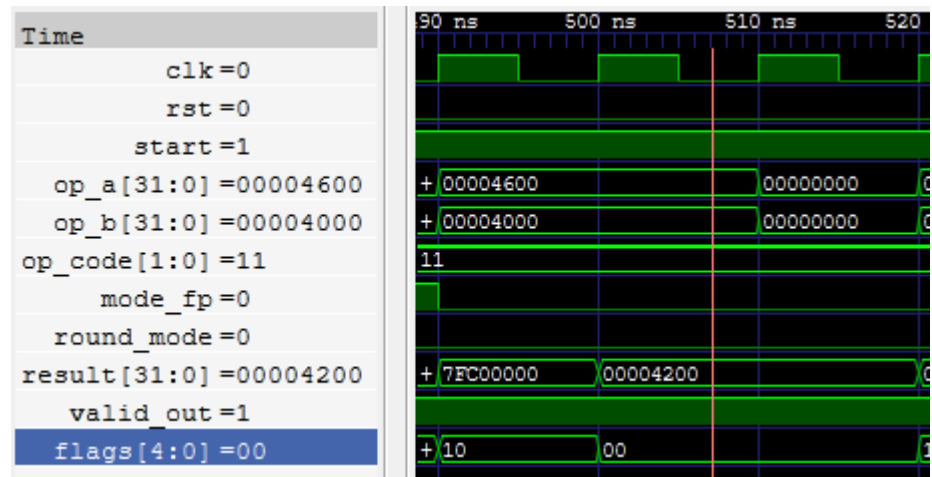
```
op_a = 32'h01000000; op_b = 32'h501502F9; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



ii. 16 BITS

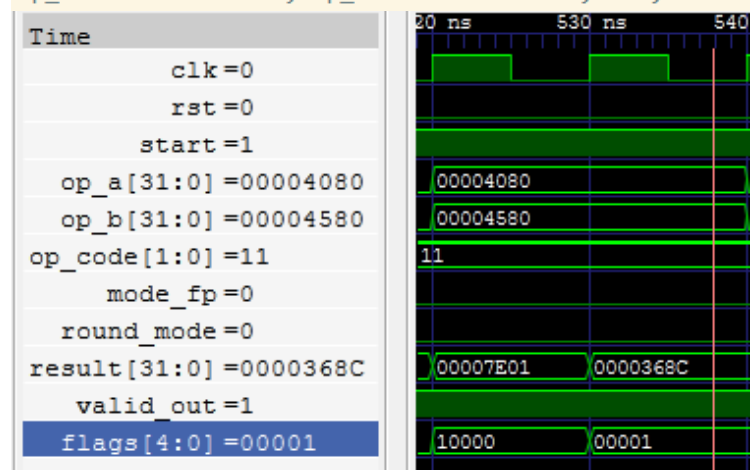
1. natural / natural

```
// 1) 6.0 (0x4600) / 2.0 (0x4000) = 3.0 (0x4200)
// NO FLAG
op_a = 32'h00004600; op_b = 32'h00004000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



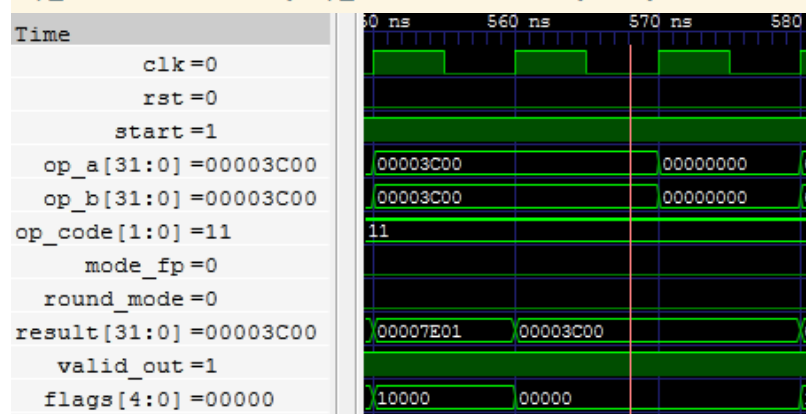
2. decimal / decimal

```
// 2) 2.25 (0x4080) / 5.5 (0x4580) ≈ 0.4091 (0x368B - 0x368C)
// FLAG: inexact
op_a = 32'h00004080; op_b = 32'h00004580; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



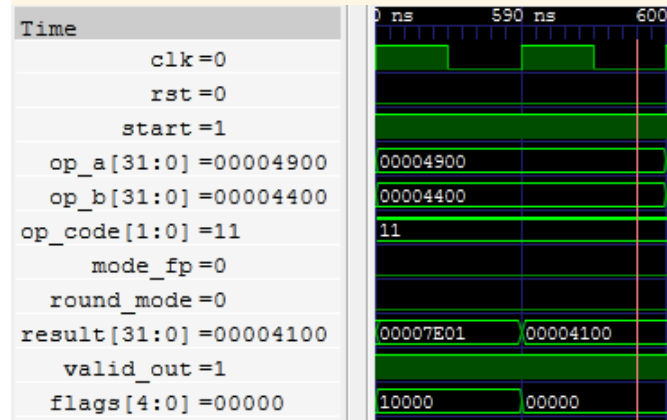
3. natural / sí mismo

```
// 3) 1.0 (0x3C00) / 1.0 (0x3C00) = 1.0 (0x3C00)
// NO FLAG
op_a = 32'h00003C00; op_b = 32'h00003C00; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



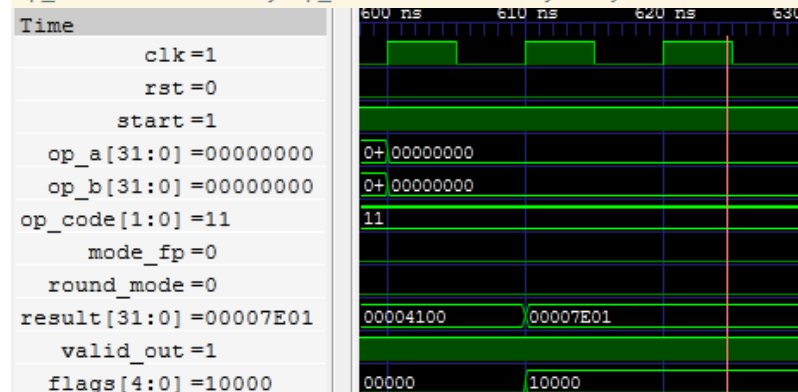
4. natural / natural = decimal

```
// 4) 10.0 (0x4900) / 4.0 (0x4400) = 2.5 (0x4100)
// NO FLAG
op_a = 32'h00004900; op_b = 32'h00004400; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



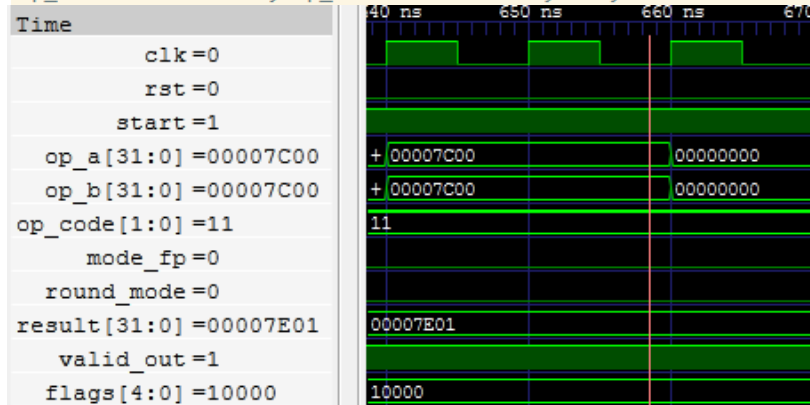
5. Caso específico: 0 / 0

```
// 5) 0 (0x0000) / 0 (0x0000) → NaN (0x7E00)
// FLAG: invalid
op_a = 32'h00000000; op_b = 32'h00000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



6. Caso específico: + inf / +inf

```
// 6) +Inf (0x7C00) / +Inf (0x7C00) → NaN (0x7E00)
// FLAG: invalid
op_a = 32'h00007C00; op_b = 32'h00007C00; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```



7. Caso específico: división entre 0

```
// 7) 1.0 (0x3C00) / 0.0 (0x0000) = +Inf (0x7C00)
// FLAG: div_zero
op_a = 32'h00003C00; op_b = 32'h00000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
// 8) -1.0 (0xBC00) / 0.0 (0x0000) = -Inf (0xFC00)
// FLAG: div_zero
op_a = 32'h0000BC00; op_b = 32'h00000000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

[illegible]

8. ± 0 / número

```
// 9) +0 (0x0000) / 5.0 (0x4500) = +0 (0x0000)
// NO FLAG
op_a = 32'h00000000; op_b = 32'h00004500; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
// 10) -0 (0x8000) / 2.0 (0x4000) = -0 (0x8000)
// NO FLAG
op_a = 32'h00008000; op_b = 32'h00004000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

Time	0 ns	730 ns	740 ns	750 ns	760 ns	770 ns	780 ns	790 ns
clk=0								
rst=0								
start=1								
op_a[31:0]=00000000	00000000 00008000 00000000							
op_b[31:0]=00004500	00000000 00004500 00000000 00004000 00000000							
op_code[1:0]=11	11							
mode_fp=0								
round_mode=0								
result[31:0]=00000000	0000FC00 00007E01 00000000 00007E01 00008000							
valid_out=1								
flags[4:0]=00000	00010 10000 00000 10000 00000							

9. Denormales pequeño y grande

```
// 11) Denormal pequeño: 5.97e-8 (0x0001) / 2.0 (0x4000) ≈ 0.0000002985 (0x0000)
// FLAG: underflow + inexact
// PROBAR
op_a = 32'h00000001; op_b = 32'h00004000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
// 12) Denormal grande: 1.2e-7 (0x0002) / 1024 (0x6400) ≈ 0.000000001 (0x0000)
// FLAG: underflow + inexact
op_a = 32'h00000002; op_b = 32'h00006400; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

Time	800 ns	810 ns	820 ns	830 ns	840 ns	850 ns
clk=0						
rst=0						
start=1						
op_a[31:0]=00000001	00000001	00000000	00000002		00000000	
op_b[31:0]=00004000	00004000	00000000	00006400		00000000	
op_code[1:0]=11	11					
mode_fp=0						
round_mode=0						
result[31:0]=00000000	00007E01	00000000	00007E01	00000000		
valid_out=1						
flags[4:0]=00101	10000	00101	10000	00101		

10. Caso especial: overflow

```
// 13) Overflow: 65504 (0x7BFF) / 0.5 (0x3800) = +Inf (0x7C00)
// FLAG: overflow + inexact
op_a = 32'h00007BFF; op_b = 32'h00003800; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
// 14) Overflow: -65504 (0xFBFF) / 0.5 (0x3800) = -Inf (0xFC00)
// FLAG: overflow + inexact
op_a = 32'h0000FBFF; op_b = 32'h00003800; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

Timing diagram showing the execution of the floating-point adder. The diagram displays signals over time (0 ns to 910 ns). A red vertical line indicates the 870 ns mark.

Signals and values shown:

- clk=0
- rst=0
- start=1
- op_a[31:0]=00007BFF
- op_b[31:0]=00003800
- op_code[1:0]=11
- mode_fp=0
- round_mode=0
- result[31:0]=00007C00
- valid_out=1
- flags[4:0]=01001

11. Caso especial: underflow

```
// 15) Underflow extremo:  $6.1e-5(0x03FF)/8192(0x7000) \approx 0.000000074 (0x0000)$ 
// FLAG: underflow + inexact
op_a = 32'h000003FF; op_b = 32'h00007000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;

// 16) Underflow suave:  $1.2e-7(0x0002)/512(0x6000) \approx 0.0000000023(0x0000)$ 
// FLAG: underflow + inexact
op_a = 32'h00000002; op_b = 32'h00006000; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

Timing diagram showing signals over time (910 ns to 970 ns). The signals are:

- clk=0
- rst=0
- start=1
- op_a[31:0]=000003FF
- op_b[31:0]=00007000
- op_code[1:0]=11
- mode_fp=0
- round_mode=0
- result[31:0]=00000000
- valid_out=1
- flags[4:0]=00101

12. negativo / negativo

```
// 18) -2.75 (0xC180) / -5.5 (0xC580) ≈ 0.5 (0x3800)
// FLAG: inexact
op_a = 32'h0000C180; op_b = 32'h0000C580; #20;
op_a = 32'h00000000; op_b = 32'h00000000; #10;
```

Time	0 ns	980 ns	990 ns	1 us	101 ns
clk=0					
rst=0					
start=1					
op_a[31:0] = 0000C180	0000C180 00000000				
op_b[31:0] = 0000C580	0000C580 00000000				
op_code[1:0] = 11	11				
mode_fp=0					
round_mode=0					
result[31:0] = 00003800	00007E01 00003800 00007E01				
valid_out=1					
flags[4:0] = 00000	10000 00000 10000				

En todos los casos, los resultados obtenidos fueron consistentes con los valores esperados. Además, se observó la correcta gestión de *overflow*, *underflow* y valores especiales.

Conclusiones

El proyecto permitió demostrar el funcionamiento interno de las operaciones en punto flotante y su implementación en hardware.

La ALU cumple con el estándar *floating point* IEEE-754 para los formatos de 16 y 32 bits, ejecutando correctamente las operaciones solicitadas de suma, resta, multiplicación y división.

El enfoque modular facilitó la verificación individual de cada componente y su posterior integración.

Otra métrica proxy que complementa la evaluación del desarrollo es la Tasa de Fecundidad Total (TFR). Este indicador refleja la percepción ciudadana sobre la estabilidad socioeconómica, la conciliación entre trabajo y familia, y la confianza en el futuro. En 2024, Corea del Sur registró una TFR de 0.75 hijos por mujer, apenas por encima del 0.72 del año anterior, pero muy por debajo del nivel de reemplazo poblacional de 2.1 (SwissInfo, 2024).

Esta cifra sugiere que el modelo económico surcoreano, a pesar de su sofisticación tecnológica, no ha garantizado una sostenibilidad demográfica. La baja fecundidad implica un futuro descenso en la población activa, poniendo en riesgo el principal motor del crecimiento: el capital humano calificado. Además, evidencia un desequilibrio entre los logros económicos y las condiciones sociales necesarias para la reproducción generacional.

Referencias bibliográficas

Harris, D. & Harris S. (2016). *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann. <https://dl.acm.org/doi/10.5555/2815529>

IEEE Computer Society (2008). *IEEE Standard for Floating-Point Arithmetic*.
https://www.dsc.ufcg.edu.br/~cnum/modulos/Modulo2/IEEE754_2008.pdf

Anexos

Anexo A

Repositorio principal de github + esquemático + video
[<https://github.com/flaqq0/proyecto-1---arquitectura-de-computadores.git>]