



Universidad de Ingeniería y Tecnología

Arquitectura de Computadoras

Diseño e implementación de una ALU para operaciones en punto flotante IEEE-754

Plataforma: FPGA Basys3 | Lenguaje: Verilog HDL

Integrantes

Campoverde San Martín, Yacira Nicol - 202120128

Pinedo Espinoza, Jose Eddison - 202120445

Gian Franco Aedo Farge - 202410291

Lima, 20 de octubre de 2025

Índice

1. Introducción	1
2. Fundamento teórico	1
2.1. Representación IEEE-754	1
2.2. Operaciones aritméticas en IEEE-754	2
3. Arquitectura del sistema ALU	3
3.1. Diagrama de bloques general	3
3.2. Interconexión de módulos	3
3.3. Paralelismo en las operaciones	4
4. Máquina de estados - Diseño e implementación	4
4.1. Diseño de la FSM principal	4
4.2. Estados y transiciones	5
4.2.1. Estado IDLE (000)	5
4.2.2. Estado DECODE (001)	5
4.2.3. Estado COMPUTE (010)	6
4.2.4. Estado NORMALIZE (011)	6
4.2.5. Estado ENCODE (100)	7
4.2.6. Estado DONE (101)	7
4.3. Señales de control	8
5. Módulos de decodificación y codificación IEEE-754	8
5.1. Decodificador (<code>ieee754_decoder</code>)	8
5.2. Detección de casos especiales	8
5.3. Codificador (<code>ieee754_encoder</code>) y formateo de salida	9
6. Módulos aritméticos: suma/resta y multiplicación	9
6.1. Módulo sumador/restador (<code>fp_adder_subber</code>)	9
6.2. Módulo multiplicador (<code>fp_multiplier</code>)	10
6.3. Normalización y manejo de overflow/underflow	11
7. Módulo divisor y casos especiales	11
7.1. Implementación del divisor (<code>fp_divider</code>)	11
7.2. Manejo de división por cero y casos límite	11
7.3. Generación de <i>flags</i>	12
8. Simulación	12
9. Resultados y análisis	15
9.1. Resultados de simulación	15
10. Conclusiones	15

1. Introducción

El procesamiento de números reales es fundamental en aplicaciones científicas, de ingeniería y multimedia, donde la representación de valores con alta precisión y amplio rango dinámico resulta esencial. A diferencia del cómputo en punto fijo o aritmética entera, que limita severamente el rango de valores representables y la precisión de los cálculos, el estándar IEEE-754 para aritmética de punto flotante permite representar números desde valores extremadamente pequeños (del orden de 10^{-38} en precisión simple) hasta valores muy grandes (del orden de 10^{38}), manteniendo una precisión relativa controlada en todo el rango.

La **Unidad Aritmético-Lógica (ALU)** constituye el corazón computacional de cualquier procesador, siendo responsable de ejecutar las operaciones aritméticas y lógicas fundamentales. Las ALUs especializadas en punto flotante no solo deben realizar correctamente las operaciones básicas (suma, resta, multiplicación y división), sino que también deben gestionar casos excepcionales definidos por el estándar, como el manejo de infinitos, valores NaN (*Not a Number*), overflow, underflow y números subnormales.

2. Fundamento teórico

2.1. Representación IEEE-754

La representación en punto flotante surge de la necesidad de aproximar números reales en sistemas digitales, ya que no es posible representar todos los valores reales de manera exacta con una cantidad finita de bits. Los sistemas de punto fijo ofrecen precisión limitada y errores de representación que pueden ser significativos, especialmente para valores pequeños. Por ello, el estándar IEEE-754 define un formato de punto flotante que permite una mayor gama dinámica y mejor manejo del error relativo.

Un número en punto flotante tiene cuatro componentes: signo, significando (mantisa), base (usualmente 2) y exponente. Se representa como:

$$x = \pm s \times b^e$$

donde s es la mantisa y e el exponente.

El estándar IEEE-754 establece formatos binarios de 16, 32, 64 y 128 bits, conocidos como half, single, double y quadruple precision, respectivamente. Cada formato divide el número en tres campos:

- **Signo (1 bit):** Indica si el número es positivo o negativo.
- **Exponente (con sesgo):** Determina la escala del número. El sesgo depende del formato (por ejemplo, 127 para 32 bits).
- **Mantisa (fracción):** Representa la precisión del número. Incluye un bit implícito (el “1” oculto).

Formato	Bits exponente (sesgo)	Bits mantisa (+1 implícito)	Total de bits
Half (16)	5 (15)	10 + 1	16
Single (32)	8 (127)	23 + 1	32
Double (64)	11 (1023)	52 + 1	64
Quad (128)	15 (16383)	112 + 1	128

Cuadro 1: Formatos binarios IEEE 754-2008.

El estándar también define valores especiales: cero, infinito ($\pm\infty$), NaN (Not a Number) y números subnormales (o denormales), que permiten representar valores muy pequeños.

2.2. Operaciones aritméticas en IEEE-754

En la ALU propuesta se implementarán operaciones de suma, resta, multiplicación y división en punto flotante para los formatos half (16 bits) y single (32 bits).

La suma y resta en IEEE-754 requieren varios pasos: primero se alinean los exponentes de los operandos, luego se realiza la operación sobre las mantisas (significandos) ya alineadas, y finalmente se normaliza el resultado y se ajusta el exponente. Además, se debe gestionar el signo, el redondeo y los casos especiales como ceros, infinitos, NaN y subnormales.

Las operaciones de multiplicación y división: se suman o restan los exponentes, se multiplican o dividen las mantisas, y luego se normaliza el resultado. También es necesario manejar los casos especiales definidos por el estándar.

Todas estas operaciones requieren circuitos adicionales para el manejo correcto de los signos, la normalización y el tratamiento de valores especiales.

3. Arquitectura del sistema ALU

3.1. Diagrama de bloques general

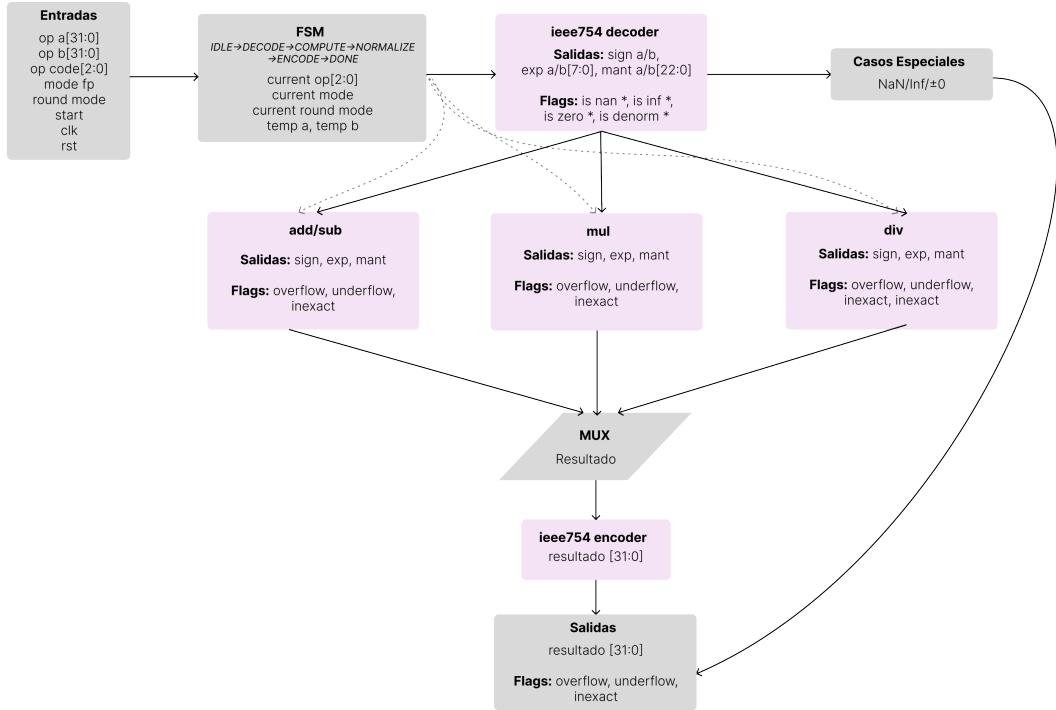


Figura 1: Diagrama de la ALU

3.2. Interconexión de módulos

La arquitectura del sistema se basa en un diseño jerárquico y modular, con una separación entre los módulos funcionales y los módulos de interfaz, donde cada componente cumple una función específica dentro del procesamiento de operaciones aritméticas en punto flotante. La **Unidad Aritmético-Lógica (ALU)** actúa como módulo central encargado de coordinar el flujo de datos y seleccionar la operación que debe ejecutarse, mientras que los módulos auxiliares realizan las operaciones aritméticas y la gestión del formato **IEEE-754**. La interconexión entre estos módulos permite ejecutar correctamente operaciones en punto flotante tanto en formato **single precision (32 bits)** como en **half precision (16 bits)**.

El módulo principal alu.v integra internamente cuatro submódulos funcionales:

- **fp_adder_subber:** Encargado de realizar operaciones de suma y resta en punto flotante.
- **fp_multiplier:** Realiza la multiplicación.
- **fp_divider:** Realiza la división.

- **ieee754_encoder** y **ieee754_decoder**: Utilizados para la conversión entre el formato binario IEEE-754 y sus componentes (signo, exponente y mantisa), permitiendo que las operaciones aritméticas se realicen sobre valores normalizados.

Todos los módulos se interconectan a través del archivo **alu**, donde comparten las mismas entradas **a**, **b** y se procesan en paralelo. Cada operación se ejecuta en su módulo correspondiente, y un multiplexor interno controlado por la señal **opcode** selecciona el resultado final que se propagará hacia la salida **result**. La señal adicional **mode_fp** permite alternar entre los modos de precisión **single** y **half**, controlando internamente cómo deben truncarse, extenderse o interpretarse los datos para adaptarse a cada formato.

3.3. Paralelismo en las operaciones

El diseño de la ALU implementa un enfoque de paralelismo estructural que permite ejecutar simultáneamente las operaciones de suma/resta, multiplicación y división. Cada módulo aritmético (**fp_adder_subber**, **fp_multiplier**, **fp_divider**) recibe en paralelo los operandos **a** y **b**, y calcula su resultado independientemente del tipo de operación requerida. Posteriormente, un multiplexor controlado por la señal **opcode** selecciona el resultado correspondiente, eliminando la necesidad de una lógica secuencial que active cada módulo por separado. Este paralelismo también se extiende al manejo de precisión, ya que tanto en modo **single** como **half**, la lógica de truncamiento o extensión se aplica de forma concurrente en todos los módulos. Como resultado, se obtiene una arquitectura más rápida y eficiente, ideal para su implementación en FPGA, donde los recursos lógicos permiten el uso simultáneo de múltiples unidades funcionales sin afectar la latencia global del sistema.

4. Máquina de estados - Diseño e implementación

La máquina de estados finitos (FSM) constituye el control de la ALU, coordinando la ejecución secuencial de las operaciones aritméticas en punto flotante. Esta FSM está implementada como un controlador sincrónico que gestiona el flujo de datos entre los módulos funcionales, asegurando que cada etapa del procesamiento se complete correctamente antes de avanzar a la siguiente.

4.1. Diseño de la FSM principal

La FSM implementada consta de **6 estados principales**.

Estado	Código	Señales activas/Función principal
IDLE	000	<code>valid_out=0</code> . Espera señal <code>start</code> . Captura operandos y configuración.
DECODE	001	Activa <code>ieee754_decoder</code> . Extrae signo, exponente y mantisa de ambos operandos.
COMPUTE	010	Ejecuta módulos aritméticos en paralelo. Detecta casos especiales (NaN, Inf, cero).
NORMALIZE	011	Verifica overflow/underflow. Espera latencia de MUL (2 ciclos) o DIV (<code>div_ready</code>).
ENCODE	100	Activa <code>ieee754_encoder</code> . Reconstruye formato IEEE-754 desde componentes.
DONE	101	<code>valid_out=1</code> . Resultado disponible en <code>result</code> . Flags actualizados.

Cuadro 2: Tabla de estados y señales de control de la FSM.

4.2. Estados y transiciones

4.2.1. Estado IDLE (000)

La FSM permanece en este estado hasta que se activa la señal `start`. Durante este periodo:

- La señal `valid_out` se mantiene en bajo, indicando que no hay resultados válidos disponibles.
- Se realiza el reset de los registros internos si la señal `rst` está activa.
- Al detectar `start=1`, se capturan los operandos `op_a` y `op_b`, el código de operación `op_code`, el modo de precisión `mode_fp` y el modo de redondeo `round_mode` en registros temporales.

Transición: IDLE $\xrightarrow{\text{start}=1}$ DECODE

4.2.2. Estado DECODE (001)

En este estado se activa el módulo `ieee754_decoder`, que descompone ambos operandos en sus componentes IEEE-754:

- Bits de signo: `sign_a`, `sign_b`
- Exponentes con sesgo: `exp_a`, `exp_b`
- Mantisas normalizadas: `mant_a`, `mant_b`
- Flags de casos especiales: `nan_a/b`, `inf_a/b`, `zero_a/b`, `denorm_a/b`

Transición: DECODE → COMPUTE

4.2.3. Estado COMPUTE (010)

Estado donde se ejecuta la lógica aritmética y se gestionan los casos especiales del estándar IEEE-754. El procesamiento se divide en dos caminos principales:

Detección de casos especiales:

1. **NaN (Not a Number):** Si cualquier operando es NaN, el resultado es NaN:

```
result = 0x7FC00000 (SP) o 0x7E00 (HP)
flags[4] = 1
```

2. **Infinito:** Gestión según la operación:

- ADD/SUB: $\infty - \infty = \text{NaN}$; $\infty + \text{número} = \infty$
- MUL: $0 \times \infty = \text{NaN}$; $\text{número} \times \infty = \infty$
- DIV: $\frac{\infty}{\infty} = \text{NaN}$; $\frac{\infty}{\text{número}} = \infty$; $\frac{\text{número}}{\infty} = 0$

3. **División por cero:**

- $\frac{0}{0} = \text{NaN}$ (invalid operation)
- $\frac{\text{número}}{0} = \pm\infty$ (divide by zero)

4. **Multiplicación por cero:** $\text{número} \times 0 = \pm 0$ (con signo apropiado)

Ejecución de operaciones normales: Si no se detectan casos especiales, los tres módulos aritméticos (`fp_adder_subber`, `fp_multiplier`, `fp_divider`) procesan los operandos en paralelo. Selecciona el resultado correcto basándose en `current_op`.

Transiciones:

- COMPUTE $\xrightarrow{\text{casos especiales}} \text{DONE}$
- COMPUTE $\xrightarrow{\text{sin casos especiales}} \text{NORMALIZE}$

4.2.4. Estado NORMALIZE (011)

Este estado gestiona la normalización del resultado y sincroniza las operaciones con latencia variable:

1. **Verificación de overflow:** Si el exponente excede el valor máximo (255 en SP, 31 en HP), se genera infinito:

```
result = {sign, 8'hFF, 23'b0} (SP) o {sign, 5'h1F, 10'b0} (HP)
flags[2] = 1 (overflow)
```

2. **Verificación de underflow:** Si el exponente es menor que cero, se genera cero con signo:

```
result = {sign, 31'b0} (SP) o {sign, 15'b0} (HP)
flags[1] = 1 (underflow)
```

3. **Sincronización de operaciones:**

- ADD/SUB: Operaciones combinacionales, transición inmediata.
- MUL: Latencia de 2 ciclos. Se utiliza `mul_counter` que decrementa hasta 0.
- DIV: Latencia variable. Se espera la señal `div_ready=1`.

Transiciones:

- NORMALIZE $\xrightarrow{\text{overflow/underflow}}$ DONE
- NORMALIZE $\xrightarrow{\text{operación completa}}$ ENCODE

4.2.5. Estado ENCODE (100)

Activa el módulo `ieee754_encoder` para reconstruir el formato IEEE-754 completo a partir de los componentes calculados (`final_result_sign`, `final_result_exp`, `final_result_mant`). El resultado codificado se asigna al registro `result`.

En paralelo, se actualizan los flags de estado:

- `flags[4]`: Invalid operation
- `flags[3]`: Divide by zero
- `flags[2]`: Overflow
- `flags[1]`: Underflow
- `flags[0]`: Inexact

Transición: ENCODE → DONE

4.2.6. Estado DONE (101)

Estado final donde el resultado está disponible. Se activa `valid_out=1`, indicando al sistema externo que puede leer el resultado. La FSM permanece en este estado hasta que la señal `start` regresa a 0.

Transición: DONE $\xrightarrow{\text{start}=0}$ IDLE

4.3. Señales de control

- **start:** Señal de inicio que dispara la ejecución de una nueva operación desde IDLE.
- **valid_out:** Indica que el resultado en **result** es válido y estable.
- **op_code[2:0]:** Selecciona la operación: 000=ADD, 001=SUB, 010=MUL, 011=DIV.
- **mode_fp:** 0=half precision (16 bits), 1=single precision (32 bits).
- **round_mode:** Controla el modo de redondeo.

5. Módulos de decodificación y codificación IEEE-754

5.1. Decodificador (`ieee754_decoder`)

El módulo `ieee754_decoder` tiene como propósito descomponer un número en formato IEEE-754 en sus tres componentes fundamentales: **signo**, **exponente** y **mantisa**. Este proceso es esencial para permitir que las operaciones aritméticas se realicen de manera directa sobre los valores numéricos reales que representa el formato binario.

La entrada del decodificador es un número flotante codificado en 32 bits (para modo **single**) o 16 bits (para modo **half**), y su salida consiste en:

- El bit de signo (**sign**), que indica si el número es positivo (0) o negativo (1).
- El exponente (**exponent**), desplazado según el bias del estándar (127 para **single**, 15 para **half**).
- La mantisa normalizada (**mantissa**), que incluye el bit implícito 1 en la posición más significativa si el número no es denormalizado.

El módulo `ieee754_decoder` es utilizado como etapa inicial en los módulos aritméticos del sistema, proporcionando los componentes separados del número flotante para su posterior procesamiento.

5.2. Detección de casos especiales

El sistema implementa lógica específica para la detección de casos especiales definidos por el estándar IEEE-754, los cuales requieren un tratamiento diferenciado respecto a los números normales. Estos casos son detectados principalmente en los módulos de decodificación y en los módulos aritméticos, antes de realizar las operaciones.

Para identificar un valor **NaN** (Not a Number), el sistema verifica si el campo de exponente es todo unos (`exp = 255` para **single**, o `exp = 31` para **half**) y si la mantisa es distinta de cero. Por otro lado, los valores **±Inf** se detectan cuando el exponente es máximo y la mantisa es cero. En ambos casos, se propagan señales o flags que evitan que la operación continúe y asignan un resultado especial codificado.

Los **ceros con signo** (positivo o negativo) se detectan cuando tanto el exponente como la mantisa son ceros, y el bit de signo se conserva en el resultado. Los **números**

denormales, aquellos con exponente cero pero mantisa distinta de cero, también son identificados y tratados por los módulos para evitar errores durante la normalización y el alineamiento de mantisas.

Además, el sistema contempla la generación de **flags de excepción** que indican eventos como **overflow**, **underflow**, **divide-by-zero**, **invalid operation** e **inexact**, aunque algunos de estos aún pueden estar en desarrollo según la versión actual del proyecto. Estos flags permiten, en futuras versiones, activar mecanismos de manejo de errores o redondeos alternativos.

A modo de ejemplo, si se intenta dividir un número positivo entre cero, el sistema devuelve **+Inf** y activa el flag de **divide-by-zero**. Si se multiplica un número por **NaN**, el resultado también será **NaN**, conforme al comportamiento definido por IEEE-754.

5.3. Codificador (ieee754_encoder) y formateo de salida

En la entrada, el codificador recibe el bit de signo (**sign**), el exponente ajustado con el sesgo correspondiente (127 para single y 15 para half), y la mantisa normalizada. Si la operación dio lugar a un resultado denormalizado, se elimina el bit implícito que normalmente se agrega a la mantisa durante la decodificación. A continuación, se ensamblan los campos en un

6. Módulos aritméticos: suma/resta y multiplicación

6.1. Módulo sumador/restador (fp_adder_subber)

Interfaz. El módulo recibe: **clk**, **rst**, **mode_fp** (0=half, 1=single), **operation** (0=suma, 1=resta), campos decodificados de los operandos (**sign_a**, **sign_b**, **exp_a**, **exp_b**, **mant_a**, **mant_b**) y **round_mode**. Entrega **result_sign**, **result_exp**, **result_mant** y **flags** (**overflow**).

Algoritmo. Se sigue el flujo IEEE-754 clásico:

1. *Alineación de exponentes.* Se identifica el mayor exponente y se desplaza la mantisa del operando con exponente menor. Internamente se antepone el 1 implícito: $\widehat{M} = 1.\text{mantisa}$ para números normales. Para denormales, el bit implícito es 0.
2. *Suma/resta de significandos.* Según **operation** y signos, se realiza $M_{\text{mayor}} \pm M_{\text{menor_alineado}}$.
3. *Normalización.* Si hay acarreo en la suma (≥ 2), se desplaza a la derecha y se incrementa el exponente. Si hay cancelación (cabezas de cero), se desplaza a la izquierda contando ceros líderes hasta restaurar el 1 implícito, decrementando el exponente.
4. *Redondeo.* Modo por defecto: *round to nearest, ties to even*. Se utilizan bits Guard/Round/Sticky (GRS). Si GRS indica que debe redondearse, se suma 1 a la mantisa. Un nuevo acarreo vuelve a normalizar e incrementa el exponente.

5. *Re-empaquetado.* Con el sesgo apropiado (15 para half, 127 para single), se generan los campos finales.

Casos especiales. Antes del cómputo se prioriza la *propagación de excepciones*:

- $\text{NaN} \oplus X \rightarrow \text{NaN}$ (se propaga).
- $+\infty + (-\infty) \rightarrow \text{invalid}(\text{NaN})$.
- $X \pm 0 = X$ preservando el signo para ceros con signo.
- Denormales: se tratan con bit implícito 0 y exponente mínimo.

Saturación flags. Si tras normalizar el exponente excede el máximo, se reporta `overflow` y el resultado se satura a $\pm\infty$. Si el exponente cae por debajo del mínimo, el resultado se denormaliza; si la magnitud es demasiado pequeña, se reporta `underflow` y se entrega cero con signo (este flag lo reporta la ALU al empacar).

Complejidad y latencia. Implementación secuencial sincronizada a `clk`; la alineación mediante desplazadores y el conteo de ceros líderes dominan el tiempo crítico. El diseño está preparado para half y single con el mismo camino de datos (`mode_fp`).

6.2. Módulo multiplicador (fp_multiplier)

Interfaz. Señales análogas al módulo anterior; salida `result_sign`, `result_exp`, `result_mant` y flags `overflow`, `underflow`.

Diseño. Se utilizan dos etapas (pipeline ligero):

1. *Etapa 1: producto y suma de exponentes.* Se forman las mantisas extendidas $\widehat{M}_a = \{1, M_a\}$ y $\widehat{M}_b = \{1, M_b\}$, se calcula el producto entero $P = \widehat{M}_a \times \widehat{M}_b$ (48 bits para single), y se computa $E = (E_a - \text{bias}) + E_b$ con `SP_EXP_BIAS`= 127 (o equivalente para half).
2. *Etapa 2: normalización y redondeo.* Si P tiene la forma $10.xxx$, se desplaza una posición a la derecha y se incrementa E . Luego se aplica redondeo al modo *nearest-even* con GRS y se empaqueta.

Casos especiales y signos.

- Signo: $S = S_a \oplus S_b$.
- $\text{NaN} \times X \rightarrow \text{NaN}; 0 \times \infty \rightarrow \text{invalid}(\text{NaN})$.
- Cualquier factor ∞ produce ∞ (con signo) si el otro no es cero/ NaN .

Flags. `overflow` si E excede el máximo representable tras normalizar; `underflow` si cae por debajo del mínimo (se denormaliza y, si se pierde significancia, se marca `inexact`).

6.3. Normalización y manejo de overflow/underflow

Normalización. Se usa un *Leading Zero Counter* (LZC) para desplazar a izquierda en sumas que sufren cancelación y una detección de acarreo en multiplicación para desplazar a derecha. El exponente se ajusta en consecuencia. **GRS** proviene de los bits descartados por los desplazamientos/alineación.

Overflow. Si $E_{\text{final}} > E_{\text{max}}$ se satura a $\pm\infty$ y se eleva **overflow**.

Underflow. Si $E_{\text{final}} < E_{\text{min}}$ se intenta resultado subnormal; si los desplazamientos agotan la mantisa, el resultado es ± 0 con **underflow** e **inexact**.

7. Módulo divisor y casos especiales

7.1. Implementación del divisor (fp_divider)

Interfaz. Además de `clk/rst` y campos decodificados, el divisor expone `start` y un FSM interno con estados `DIV_IDLE`, `DIV_RUN`, `DIV_DONE`.

Algoritmo (restaurativo enteros).

1. *Preparación.* $\widehat{M}_a = \{1, M_a\}$, $\widehat{M}_b = \{1, M_b\}$ (tratando denormales). Exponente intermedio $E = (E_a - E_b) + \text{bias}$; signo $S = S_a \oplus S_b$.
2. *Iteración.* Se realiza división entera restaurativa sobre 24/24 bits (single) acumulando cociente y residuo. En cada ciclo: desplazar–restar–corregir. El resultado crudo es Q (46–47 bits útiles incluyendo GRS).
3. *Normalización.* Si Q comienza en 0,1... se ajusta con un desplazamiento a la izquierda y $E \leftarrow E - 1$. Se obtiene la fracción final (23/10 bits) y GRS para redondeo.
4. *Redondeo y empaquetado.* *Nearest-even* con propagación de acarreo si corresponde. FSM pasa a `DIV_DONE`.

Latencia. Aproximadamente 24 ciclos (single) / 11 ciclos (half) para la fase iterativa, más ciclos de normalización/paquete. Es determinista y estable para implementación en FPGA de baja frecuencia.

7.2. Manejo de división por cero y casos límite

Se implementa una matriz de resolución previa (*short-circuit*) antes de arrancar la FSM:

Caso	Resultado	Flag(s)
$X/0$ con $X \neq 0$	$\text{sign}(S) \cdot \infty$	<code>divide_by_zero</code>
$0/0$	NaN	<code>invalid</code>
∞/∞	NaN	<code>invalid</code>
∞/X con $X \neq \infty$	$\text{sign}(S) \cdot \infty$	–
X/∞ con X finito	± 0	<code>inexact</code> (si GRS $\neq 0$)
NaN op	NaN	– (se propaga)

Denormales. Si el divisor es subnormal, se normaliza internamente desplazando su mantisa y ajustando E_b . Si el dividendo es subnormal, el cociente tenderá a cero tras suficientes desplazamientos; se marca `underflow` si corresponde.

7.3. Generación de *flags*

- **invalid:** operaciones prohibidas (NaN propagado, $0/0$, $\infty - \infty$, $0 \times \infty$).
- **divide_by_zero:** $X/0$ con $X \neq 0$.
- **overflow:** exponente final $>$ máximo.
- **underflow:** exponente final $<$ mínimo con pérdida de precisión en normalización.
- **inexact:** GRS $\neq 0$ en el resultado final (redondeo aplicado).

8. Simulación

1. Half precision

- Suma: $2+2$

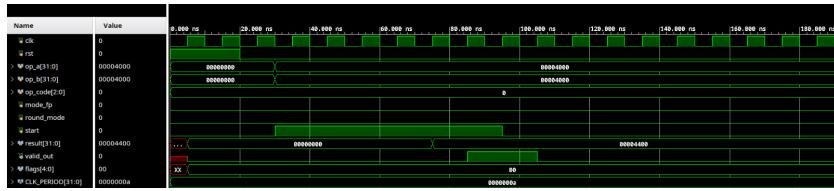


Figura 2: $2+2$ - Half precision



Figura 3: Resultado - Half precision

■ Resta: 3-2

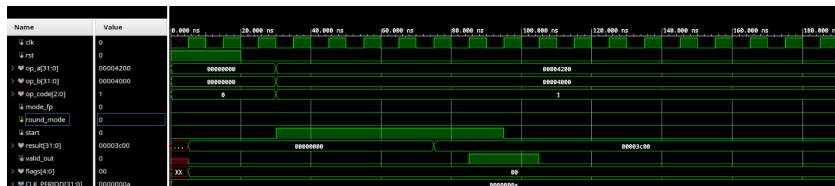


Figura 4: 3-2 - Half precision



Figura 5: Resultado - Half precision

■ División: 2/0



Figura 6: 2/0 - Half precision



Figura 7: Resultado - Half precision

2. Single precision

■ Suma: 2+2



Figura 8: 2+2



Figura 9: Resultado

■ Resta: INF - INF



Figura 10: INF - INF

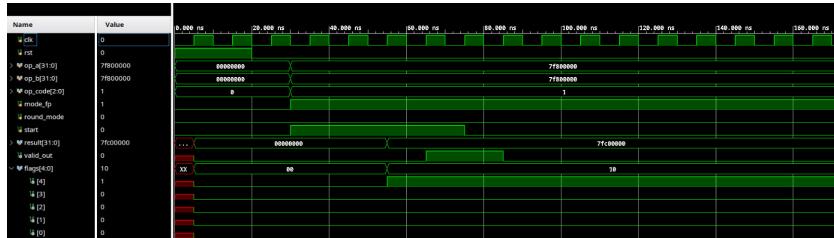


Figura 11: Resultado

- Multiplicación: 100*100

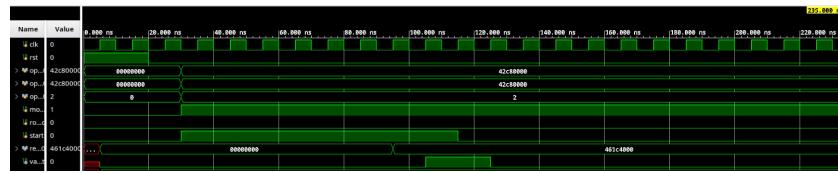


Figura 12: 100*100



Figura 13: Resultado

9. Resultados y análisis

9.1. Resultados de simulación

Las simulaciones realizadas confirmaron el funcionamiento correcto de (ALU) implementada en Verilog, verificando su capacidad para ejecutar operaciones aritméticas en punto flotante conforme al estándar IEEE-754, en formatos **half precision (16 bits)** y **single precision (32 bits)**. Se observaron correctamente las etapas de decodificación, normalización, ajuste de exponentes, detección de casos especiales (como **NaN**, **zero**, **overflow** y **underflow**), así como la posterior codificación del resultado. En todos los casos (suma, resta, multiplicación y división), los resultados fueron consistentes con los valores esperados, evidenciando la precisión del diseño y la coherencia del flujo de datos interno.

10. Conclusiones

El diseño e implementación de la ALU en punto flotante permitió alcanzar los objetivos del proyecto, logrando un sistema funcional y conforme al estándar IEEE-754. Las operaciones aritméticas básicas se resolvieron con precisión en ambos formatos (16 y 32 bits), incluyendo el tratamiento adecuado de casos especiales. La validación mediante simulación demostró la solidez del diseño modular y jerárquico, mientras que la síntesis en FPGA brindó una experiencia práctica en la integración de módulos, gestión de señales de control y despliegue de lógica digital.

Referencias

- [1] D. Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic.*
- [2] IEEE Standards Association, *IEEE Standard for Floating-Point Arithmetic (IEEE 754).*
- [3] Xilinx, *Vivado Design Suite User Guide.*