FACULTAD DE INGENIERIA
Universidad de Buenos Aires

[66.17/86.41] SISTEMAS DIGITALES

TRABAJO PRÁCTICO 1

2DO CUATRIMESTRE 2019

# Aritmética de punto flotante


AUTOR

Verstraeten, Federico. - #92892
<federico.verstraeten@gmail.com>

CÁTEDRA

Ing. Álvarez, Nicolás .

CURSO

Ing. Martos, Pedro.
Ing. Diograzia, Federico.
Ing. Alpago, Octavio.


FECHA DE ENTREGA 10 de noviembre de 2019

FECHA DE APROBACIÓN

CALIFICACIÓN

FIRMA DE APROBACIÓN

# Índice

# 1. Objetivos

El presente trabajo tiene los siguientes objetivos:

- Diseñar un procesador de aritmética de punto flotante que permita hacer las operaciones de suma, resta y multiplicación para diversos tamaños de datos.
- Repasar los conceptos del estándar IEEE 754, para operar con números de punto flotante.
- Utilizar el lenguaje *VHDL*, en cada una de las implementaciones necesarias para el desarollo de los programas.
- Utilizar el programa *Vivado* para la síntesis de los circuitos digitales descriptos en el código desarrollado.
- Verificar los casos de prueba solicitados por la cátedra.
- Ejercitar el concepto de *Pipeline*.

# 2. Diseño e implementación del programa

Se diseñaron tres programas en VHDL que permitan realizar cálculos con pares de números de representación de *Punto Flotante*, donde `fp_multiplier` permite realizar la multiplicación, `fp_adder` permite realizar la suma y `fp_subtractor` permite realizar la resta.

## 2.1. `fp_multiplier`

El Multiplicador está diseñado como un circuito combinacional puro siguiendo cada una de las etapas de cálculo del algoritmo de multiplicación de Punto Flotante y validando los casos de borde *underflow*, *overflow*, cálculo cero y cálculo infinito. La operación se hace en un solo ciclo de clock.
En la figura 4.1 se muestra la estructura general del funcionamiento de la implementación.

## 2.2. `fp_adder`

El Sumador está implementado por medio de circuitos combinacionales y secuenciales en una estructura tipo *pipeline*.
Como se observa en la figura 4.2, los bloques indicados como `block_xxx` son circuitos combinacionales que realizan una parte del procedimiento de la suma de números con representación de Punto Flotante, aplicando truncamiento en los pasos de redondeo.
Por otro lado, los bloques indicados como `pipe_xxx` son registros al estilo *Flip-Flop D*, que almacenan las salidas de las etapas anteriores y las propagan al nivel siguiente luego de un ciclo de *Clock* por cada una. Algunas de sus salidas de los registros pasan directamente a las etapas siguientes sin pasar por los bloques combinacionales, como por ejemplo determinados *flags*.

#### 2.2.1. `fp_adder_block_one`

Implementación del intercambio de operandos según el exponente más grande (Paso 1) y el reemplazo del significand B por su complemento a dos en caso que tengan distintos signos (Paso 2).

#### 2.2.2. `fp_adder_block_two`

Colocación del significand B en un registro de *p-bits* y desplazamiento a derecha $e_A - e_B$ posiciones, seteando los flags **g**,**r** y **s** (Paso 3). Cálculo de un significand preliminar $S = s_A + s_B$ sumando $s_A$ al registro de p-bits que contiene $s_B$ (Paso 4.1).

#### 2.2.3. `fp_adder_block_three`

Si los signos de los operandos A y B son diferentes, el MSB de S es 1, y no hubo carry-out entonces S es negativo. Reemplazar S por su complemento a dos (Paso 4.2).

#### 2.2.4. `fp_adder_block_four`

Desplazamiento de S. Si los signos de los operandos son iguales y hubo carry-out en el paso 4 desplazar S una posición a derecha, colocando en la posición más significativa un 1 (el carry-out). Si no se cumple cualquiera de las condiciones anteriores desplazar S a izquierda hasta que esté normalizado. En el primer desplazamiento introducir en el bit menos significativo el valor de g, luego introducir 0's. Ajuste del valor del exponente de acuerdo a los desplazamientos realizados (Paso 5).

#### 2.2.5. `fp_adder_block_five`

Cálculo del signo del resultado final. Si los operandos tienen el mismo signo, éste es el signo del resultado. En caso contrario el signo del resultado depende de cuál de los operandos es negativo, si hubo un intercambio de operandos en el paso 1, y si S fue reemplazado por su complemento a dos en el paso 4 (Paso 8).

### 2.3. `fp_subtractor`

El Restador aprovecha el funcionamiento del algoritmo de suma de números con representación de Punto Flotante, como se observa en la figura 4.3, cambiando el bit de signo del segundo operando por medio de una compuerta XOR e ingresando luego ambos operandos al Sumador.

# 3. Compilación del programa

Debido a que el presente trabajo fue realizado en un Sistema Operativo en base Linux, se utilizaron los programas `ghdl` y `gtkwave` para simular los códigos desarollados en VHDL. A su vez, se creó un archivo `Makefile`, que por medio de la herramienta `make` permite compilar y ejecutar el código desarrollado de una manera más sencilla.

Para compilar el proyecto posicionarse en el directorio donde se encuentre el `Makefile`, con los directorios `src/` (código VHDL dispositivos), `testbench/` (código VHDL testbench) y `simulation/` (directorio auxiliar), y ejecutar el siguiente comando:

```
1    $ make compile TESTBENCH=name_device_tb
```

al hacerlo, se generará el ejecutable en el directorio `simulation/`.

Para correr el código compilado, ejecutar el comando:

```
1    $ make run TESTBENCH=name_device_tb
```

En caso de ejecutarse correctamente se creará un archivo con extensión `.vcdgz` en el directorio `simulation/` del que podremos visualizar el diagrama de ondas con GTKWave por medio del comando:

```
1    $ make view TESTBENCH=name_device_tb
```

Por último, para borrar los archivos auxiliares que se generaron y los ejecutables, ejecutar el comando:

```
1    $ make clean
```

# 4. Diagramas en bloque

## 4.1. Multiplicador

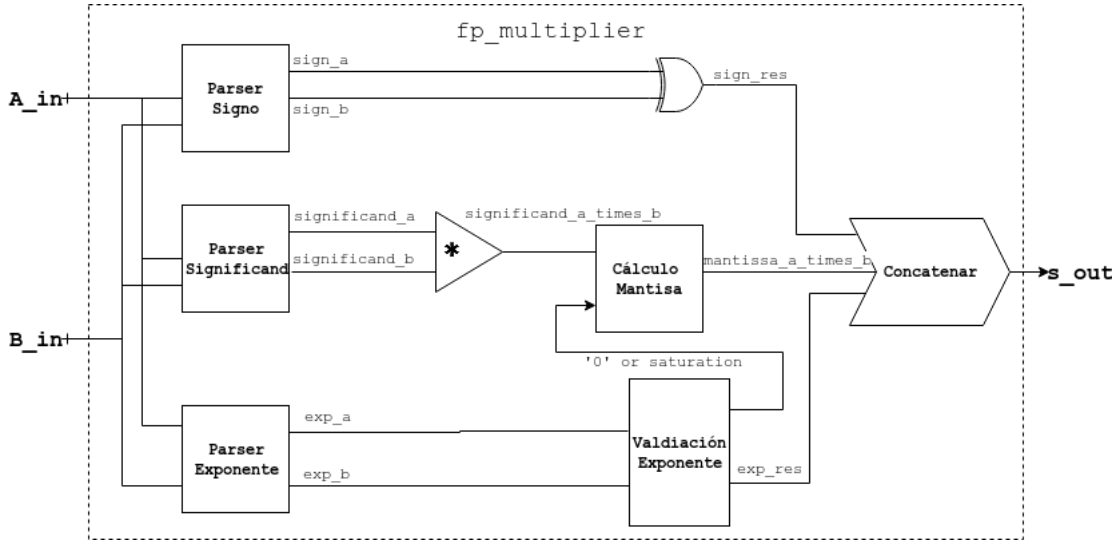


Figura 4.1: Diagrama en bloques del Multiplicador

## 4.2. Sumador

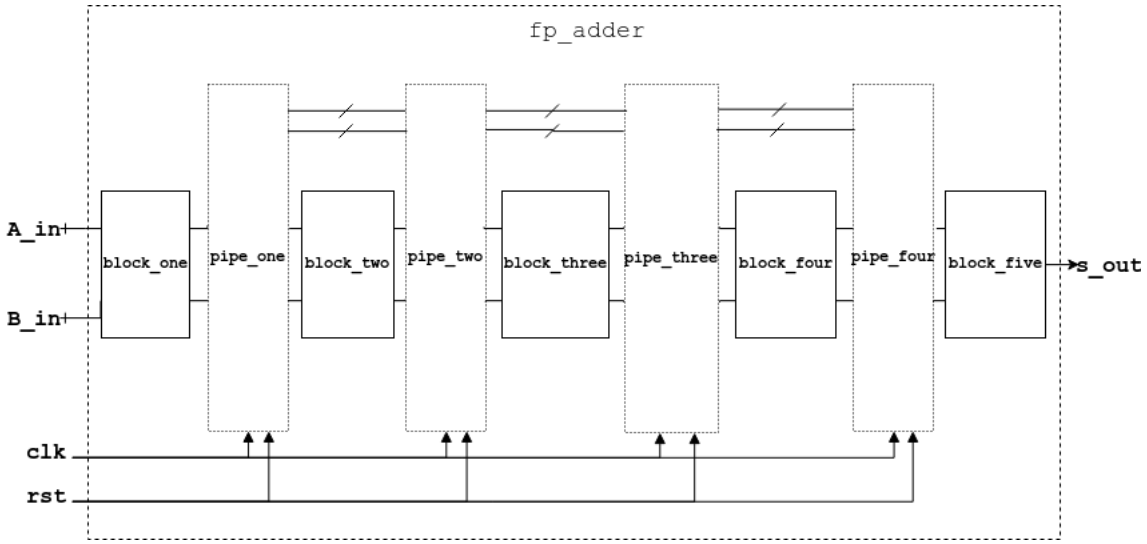


Figura 4.2: Diagrama en bloques del Sumador

## 4.3. Restador

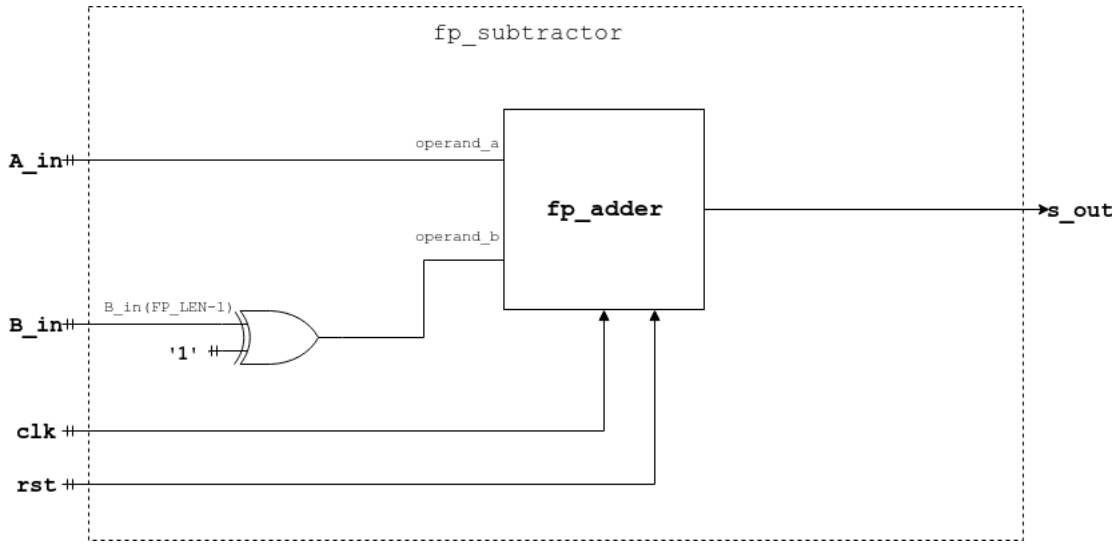


Figura 4.3: Diagrama en bloques del Restador

# 5. Recursos de hardware utilizados

El cálculo de recursos utilizados se realiza mediante la síntesis de componentes mediante con la suite de *Vivado*. Se seleccionó una longitud de 32 bits por elemento de Punto Flotante, con 8 bits de exponente.

| Lista de dispositivos utilizados | | | |
|---|---|---|---|
| Utilización Lógica | Usados | Disponible | % Utilización |
| LUT Flip Flop Pairs | 63 | 17600 | 0.36 |
| LUTs | 73 | 17600 | 0.41 |
| Slices | 21 | 4400 | 0.48 |
| DSPs | 2 | 80 | 2.50 |
| Bonded IOB | 96 | 100 | 96.0 |

Tabla 1: Tabla resumen de síntesis del Multiplicador.

| Lista de dispositivos utilizados | | | |
|---|---|---|---|
| Utilización Lógica | Usados | Disponible | % Utilización |
| Flip Flop Register | 182 | 35200 | 0.52 |
| LUTs | 73 | 17600 | 0.41 |
| Slices Register | 196 | 35200 | 0.56 |
| Slices LUTs | 463 | 17600 | 2.63 |
| F7 Muxes | 6 | 8800 | 0.07 |
| Bonded IOB | 98 | 100 | 98.0 |
| BUFGCTRL | 1 | 32 | 3.13 |

Tabla 2: Tabla resumen de síntesis del Sumador-Restador.

# 6. Simulación casos de prueba

Para ejecutar cada uno de los casos de prueba provistos por la catedra, deberá modificarse en los códigos fuentes los valores constantes `WORD_SIZE_T` que define el tamaño en bits del elemento de *Punto Flotante* y `EXP_SIZE_T` que define el tamaño del exponente. Además, si fuese necesario ajustar la ruta o PATH a los archivos de prueba, deberá modificarse las constantes `TEST_PATH` y `TEST_FILE`.

A continuación se muestra la salida por consola de cada testbench para las operaciones de multiplicación, suma y resta, donde en caso de que alguno de los test falle, se emitirá un mensaje indicando los valores que fallan, la cantidad de ciclos ejecutados y errores acumulados. A su vez se muestra una captura del analizador de ondas de GTKWave para cada caso.

## 6.1. Testbench multiplicación: `fp_multiplier_tb`

**Ejecución:**

```
$ make run TESTBENCH=fp_multiplier_tb
```

**Salida:**

```
simulation/fp_multiplier_tb --stop-time=450us --vcdgz=↩
   simulation/fp_multiplier_tb.vcdgz
simulation/fp_multiplier_tb:info: simulation stopped by --stop↩
   -time
```

Todos los test pasan satisfactoriamente, ejecutando la prueba de `test_mul_float_25_7.txt` (25 bits de tamaño de datos y 7 bits para el exponente).

**Analizador de ondas:**

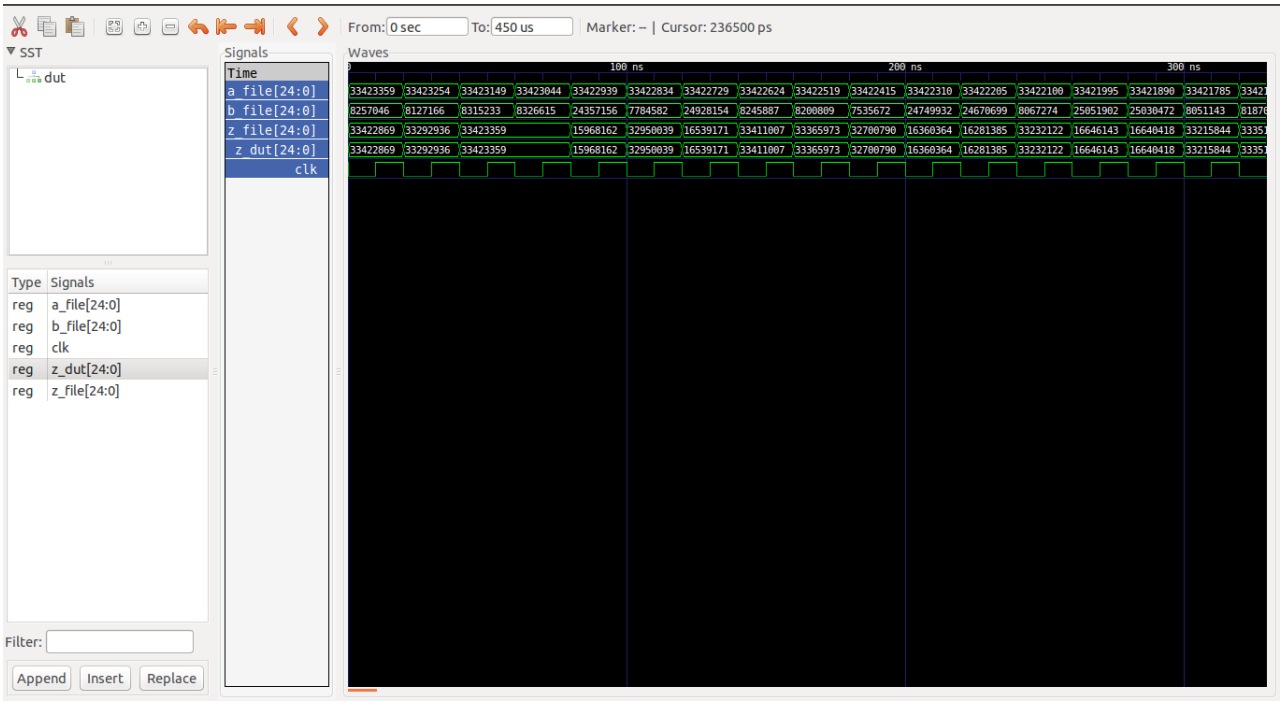


Figura 6.1: Diagrama de onda del Multiplicador

## 6.2. Testbench suma: `fp_adder_tb`

**Ejecución:**

```
$ make run TESTBENCH=fp_adder_tb
```

**Salida:**

```
testbench/fp_adder_tb.vhd:189:9:@90ns:(report note): Ciclos = 4, Errores = 1
testbench/fp_adder_tb.vhd:178:7:@100ns:(assertion warning): Calculation performed 8002001 + 8121616 = 0 and the expected result was 8192880
testbench/fp_adder_tb.vhd:189:9:@10130ns:(report note): Ciclos = 506, Errores = 2
testbench/fp_adder_tb.vhd:178:7:@10140ns:(assertion warning): Calculation performed 16528179 + 16515072 = 0 and the expected result was 16646143
simulation/fp_adder_tb:info: simulation stopped by --stop-time
```

Puede observarse que hay dos casos en los que el programa falla, donde se ejecutó la prueba de `test_sum_float_25_7.txt` (25 bits de tamaño de datos y 7 bits para el exponente).

**Analizador de ondas:**

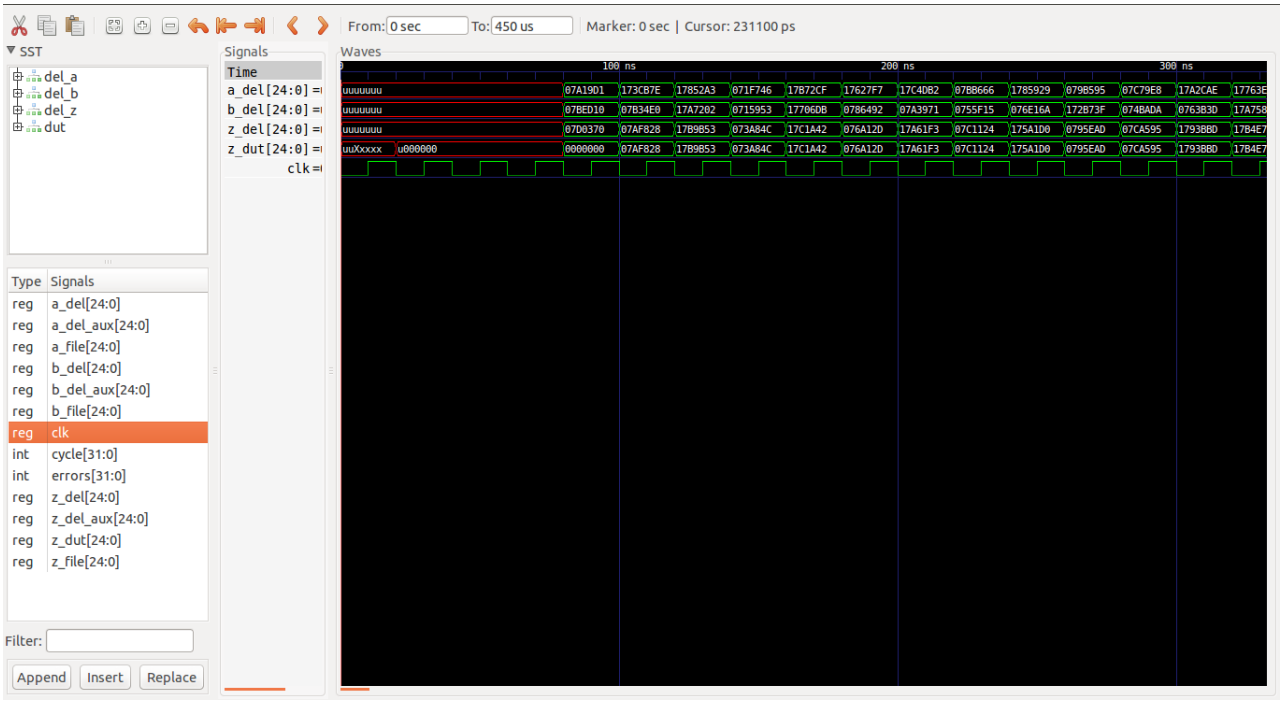


Figura 6.2: Diagrama de onda del Sumador

## 6.3. Testbench resta: `fp_subtractor_tb`

**Ejecución:**

```
$ make run TESTBENCH=fp_subtractor_tb
```

**Salida:**

```
testbench/fp_subtractor_tb.vhd:189:9:@90ns:(report note): Ciclos = 4, Errores = 1
testbench/fp_subtractor_tb.vhd:178:7:@100ns:(assertion warning): Calculation performed 8002001 - 24898832 = 0 and the expected result was 8192880
testbench/fp_subtractor_tb.vhd:189:9:@10130ns:(report note): Ciclos = 506, Errores = 2
testbench/fp_subtractor_tb.vhd:178:7:@10140ns:(assertion warning): Calculation performed 16528179 - 33292288 = 0 and the expected result was 16646143
simulation/fp_subtractor_tb:info: simulation stopped by --stop-time
```

Puede observarse que hay dos casos en los que el programa falla, donde se ejecutó la prueba de `test_dif_float_25_7.txt` (25 bits de tamaño de datos y 7 bits para el exponente).

**Analizador de ondas:**

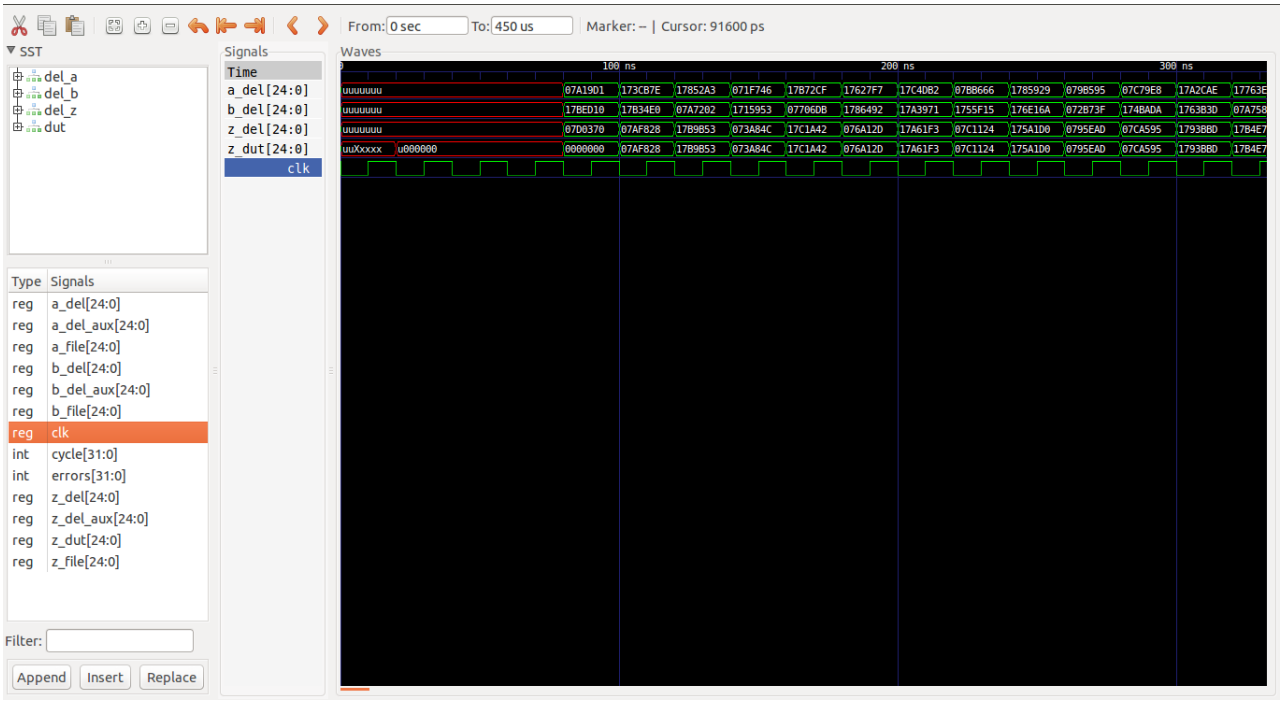


Figura 6.3: Diagrama de onda del Restador

# 7. Herramientas de hardware y software utilizadas

La computadora utilizada para realizar el desarrollo y las pruebas tiene las siguientes especificaciones:

- Procesador: Intel(R) Core(TM) i3-4005U CPU 64 bits @ 1.70GH.
- Memoria: 8GB RAM DDR4.
- Almacenamiento: Disco magnético de 500GB de 7200RPM.

Los programas se desarrollaron en el sistema operativo Linux Ubuntu, cuyos datos de distribución son

```
Distributor ID: Ubuntu
Description: Ubuntu 14.04.2 LTS
Release: 14.04
Codename: Trusty Tahr
```

Además, se utilizaron las siguientes herramientas:

- Compilador VHDL: GHDL 0.33 (20150921) [Dunoon edition]. Compiled with GNAT Version: 4.8, llvm code generator, Written by Tristan Gingold [4].
- Analizador de ondas: GTKWave Analyzer v3.3.58 (w)1999-2014 BSI [5].
- Sínstesis de hardware: Vivado v2015.1 (64-bit) - SW Build 1215546 - IP Build 1209967 - Copyright 1986-2015 Xilinx, Inc [6].
- Control del proceso de compilación: GNU Make 4.1 [7].
- Compilador del presente informe: Latex pdfTeX 3.14159265-2.6-1.40.16 (TeX Live 2015/Debian) [8].
- Edición de código fuente: VIM - Vi IMproved 7.4 (2013 Aug 10, compiled Nov 24 2016 16:44:48) y Sublime 3.2 [9] [10].

# Referencias

[1] Goldberg, David - *Computer Arithmetic - Appendix H* - Xerox Palo Alto Research Center - Elseiver Science USA - 2003.

[2] Hennesy, J.L. & Patterson D.A. - *Computer Architecture a Quantitative Approach* - Morgan Kaufmann (ISBN 0123704901) - 2006.

[3] Zaianalabedin, Navibi - *Analysis and Modeling of Digital Systems* - McGraw-Hill Professional - 1997.

[4] *GNU GHDL* - http://ghdl.free.fr/

[5] *GTKWave Analyzer* - http://gtkwave.sourceforge.net/

[6] *Vivado* - https://www.xilinx.com/products/design-tools/vivado.html

[7] *GNU Make* - https://www.gnu.org/software/make/

[8] LaTeX- https://www.latex-project.org/

[9] VIM - https://vim.sourceforge.io/

[10] Sublime Text - https://www.sublimetext.com/