Estudiantes
Rosemberth Steeven Preciga Puentes
Luis Guillermo Vaca Rincón

Práctica 1: Implementaciones hardware/software

1. Descripción

La presente práctica tuvo como propósito principal introducirse en el uso de herramientas y lenguajes de descripción enfocados en el diseño a nivel de sistema (SDL) enfocado en el modelado de hardware y software mediante SystemC. Al finalizar esta misma se espera:

- Familiarizarse con entornos de simulación y descripción de sistemas digitales a alto nivel, específicamente SystemC.
- Comparar la implementación de una aplicación cuando se describe completamente en hardware versus una implementación en un entorno hardware/software.
- Desarrollar habilidades para automatizar pruebas mediante la generación de vectores de entrada y su análisis posterior.

Para lograr estos objetivos, se planteó como ejercicio la implementación del algoritmo del Máximo Común Divisor (GCD), tanto en su versión software (ejecutada sobre un procesador RISC modelado en SystemC), como en su versión hardware, incluyendo la unidad lógico-aritmética correspondiente y la máquina de control que permite realizar su secuencia de operaciones.

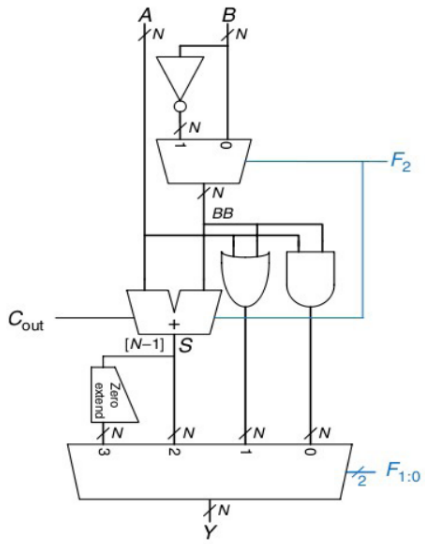

Figura 1: Camino de datos a ser implementado usando SystemC , imagen tomada de (2)

Además, se solicitó aplicar un conjunto de al menos 30 combinaciones de entrada para evaluar el comportamiento del sistema bajo distintas condiciones, y presentar los resultados mediante simulaciones utilizando `gtkwave`.

1.1. Algoritmo del Máximo Común Divisor (GCD)

El algoritmo de Euclides para calcular el máximo común divisor entre dos números enteros positivos X y Y, se basa en restas sucesivas hasta que ambos números sean iguales. El pseudocódigo puede expresarse como:

$$\textbf{while } X \neq Y \textbf{ do } \begin{cases} Y \leftarrow Y - X & \text{si } X < Y \\ X \leftarrow X - Y & \text{en otro caso} \end{cases} \tag{1}$$

Una vez que se cumple la condición $X = Y$, el valor resultante es el **máximo común divisor**:

$$\text{GCD}(X, Y) = X = Y \tag{2}$$

2. Marco teórico

2.1. SystemC

SystemC es un lenguaje basado en C++ que permite diseñar, modelar y verificar sistemas electrónicos complejos, integrando tanto hardware como software en una misma plataforma. Surgió para resolver la necesidad de unificar el desarrollo de estos dos mundos, facilitando la creación de sistemas embebidos y chips más eficientes, esto a un nivel de diseño de sistemas (SLD *System Level Design*).

Su ventaja principal es la flexibilidad: permite probar diferentes arquitecturas, repartir funciones entre hardware y software, y simular el comportamiento del sistema antes de fabricarlo. Grandes empresas de semiconductores y electrónica lo usan para explorar diseños, optimizar rendimiento y acelerar el desarrollo de plataformas virtuales.

Además, al estar estandarizado (IEEE 1666), garantiza compatibilidad y adopción en la industria. En proyectos como el tuyo (GCD en HW/SW), SystemC es útil para comparar ambas implementaciones y validar su funcionamiento de manera integrada (1).

2.2. Funcionamiento del procesador RISC en SystemC

El proyecto implementa un procesador simple con arquitectura RISC utilizando SystemC. Está compuesto por tres módulos principales: una caché de instrucciones, una unidad de decodificación y una unidad de ejecución. Cada uno de estos módulos está conectado y sincronizado a través de un reloj simulado, lo que permite modelar el comportamiento del procesador ciclo a ciclo.

2.2.1. Flujo de instrucción

Durante la simulación, en cada ciclo de reloj se ejecuta una parte del ciclo de instrucción. Primero se busca (fetch) una instrucción desde la memoria (`icache`), luego se decodifica (`decode`) para identificar la operación y operandos, y finalmente se ejecuta (`exec`). Si la instrucción implica un resultado, como una suma, ese resultado se escribe en el registro correspondiente. Las instrucciones se almacenan en un archivo `icache.img`, generado previamente a partir de un código ensamblador por el script `assembler.pl`.

2.2.2. Decodificación y registros

La unidad de decodificación interpreta el código de la instrucción y accede a un arreglo que simula los registros del procesador. También determina si la instrucción es un salto, suma, comparación, etc., y emite las señales de control correspondientes hacia la unidad de ejecución. Si hay un salto, el decodificador modifica el contador de programa (PC) para continuar la ejecución desde la nueva dirección.

2.2.3. Unidad de ejecución

La unidad de ejecución realiza la operación indicada por la instrucción (por ejemplo, una resta o comparación entre registros). Luego, envía el resultado al decodificador para que este actualice el registro destino. En instrucciones de salto, también se evalúa la condición y se confirma si el salto debe tomarse.

2.2.4. Simulación en SystemC

Todo el sistema está sincronizado con un reloj definido con precisión de 1 ns. En cada ciclo, las señales entre módulos se actualizan, simulando cómo lo haría un procesador físico. Gracias a esto, se puede ver paso a paso cómo se ejecuta cada instrucción.

3. Software

3.1. Descripción de la implementación

En primer lugar, se analizaron las instrucciones disponibles en el lenguaje ensamblador para implementar el algoritmo de Euclides. Se determinó que, mediante las siguientes instrucciones, era posible llevar a cabo su implementación de forma correcta.

- `movi R, val`: Carga un valor inmediato en el registro `R`.
- `sub R1, R2, R3`: Resta el contenido de `R3` a `R2` y guarda el resultado en `R1`.
- `beq R1, R2, dir`: Realiza un salto a la dirección `dir` si `R1` es igual a `R2`.
- `blt R1, R2, dir`: Realiza un salto a la dirección `dir` si `R1` es menor que `R2`.
- `j dir`: Salta incondicionalmente a la dirección `dir`.
- `halt`: Finaliza la ejecución del programa.

Durante el proceso de implementación de la solución propuesta, nos encontramos con un problema importante, el programa no parecía ejecutar correctamente las instrucciones de salto. Después de revisar el archivo `decode.cpp`, notamos que, en el caso de instrucciones como `beq`, aunque se calculaba correctamente la dirección objetivo del salto mediante `pc_reg + label_tmp`, al final del bloque se volvía a escribir una nueva dirección secuencial utilizando `pc_reg + 1`.

Esto sobrescribía la dirección previamente calculada, haciendo que el procesador ignorara el salto y continuara con la ejecución secuencial. A continuación se muestra el fragmento relevante del código:

```
if (srcC_tmp == srcA_tmp) {
    branch_target_address.write(pc_reg + label_tmp);
    ...
}
//branch_target_address.write(pc_reg + 1);
```

Como se puede observar, la línea comentada al final es la que sobrescribía el salto. Por esta razón, fue necesario comentarla no solo aquí, sino también en los demás bloques que manejaban instrucciones de salto condicional, como `blt`, para que el programa pudiera ejecutar correctamente la lógica del algoritmo.

Dicho esto, durante el proceso de simulación se detectó otro comportamiento inesperado en el archivo `icache.img`, generado por el ensamblador `assembler.pl`. Específicamente:

- Antes de que se transcribiera la primera instrucción, el archivo contenía varias direcciones vacías (`0x00000000`).
- Después de cada instrucción ensamblada, se insertaba un ciclo adicional vacío (`0x00000000`) en la memoria.

Se analizó entonces el código presente en el archivo `assembler.pl`, y se confirmó que, de forma predeterminada, transcribía cinco direcciones vacías antes de las instrucciones del archivo `.asm`, e insertaba una dirección vacía adicional después de cada instrucción ensamblada:

```
printf ("0x00000000\n");
printf ("0x00000000\n");
printf ("0x00000000\n");
printf ("0x00000000\n");
printf ("0x00000000\n");

...

if ($CODEGEN) {
    printf ("0x00000000\n");
}
```

Esto fue importante al momento de escribir el código en ensamblador, ya que los valores de las direcciones de salto no debían corresponder a la posición de la instrucción destino dentro del archivo `.asm`, sino a su dirección real en memoria dentro del archivo `icache.img`.

Durante las pruebas observamos que, si se intentaba saltar directamente a la dirección donde estaba la instrucción objetivo, esta no se ejecutaba. En su lugar, era necesario saltar a la línea vacía previa que el ensamblador insertaba automáticamente antes de cada instrucción.

Teniendo en cuenta estas consideraciones, a continuación se presenta el código ensamblador que implementa la función GCD:

```
movi R0, 0
movi R1, 546
movi R2, 2310
beq R1, R0, 16
beq R2, R0, 14
beq R1, R2, 12
blt R1, R2, 5
sub R1, R1, R2
j 14
sub R2, R2, R1
j 14
halt
```

Estas son las instrucciones ensambladas y guardadas en el archivo `icache.img`:

```
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0xf1000000
0x00000000
0xf1100222
0x00000000
0xf1200906
0x00000000
0x10100010
0x00000000
0x1020000e
0x00000000
0x1012000c
0x00000000
0x14120005
0x00000000
0x04112000
0x00000000
0x1600000e
0x00000000
0x04221000
0x00000000
0x1600000e
0x00000000

0x00000000
0xffffffff
```

Como se puede observar, las instrucciones se encuentran codificadas en formato hexadecimal. Para evitar que el programa entrara en un bucle infinito en caso de que alguna de las entradas fuera cero, se añadieron dos instrucciones `beq` al inicio que detienen la ejecución si `R1` o `R2` son iguales a cero.

3.2. Automatización del análisis con Python

Para cumplir con el requerimiento de generar y codificar un mecanismo que estableciera al menos 30 valores de entrada distintos, se pensó en diseñar un script en Python. La idea fue automatizar el proceso de prueba del algoritmo de GCD mediante la modificación dinámica de los registros `R1` y `R2` con valores aleatorios, compilando y ejecutando el programa en cada iteración.

Después de modificar los valores de entrada, el script debía ejecutar secuencialmente, en orden, los siguientes comandos del sistema:

- `perl assembler.pl gcd.asm -code >icache.img`: ensambla el archivo de código fuente `gcd.asm` y genera el archivo `icache.img`, que contiene las instrucciones en formato hexadecimal que el simulador ejecutará.
- `make clean`: elimina los archivos temporales generados durante la compilación anterior. Esto garantiza que la siguiente ejecución comience desde un estado limpio.
- `make`: compila los archivos fuente del simulador (escritos en SystemC) y genera el ejecutable principal, generalmente llamado `risc_cpu.x`.

- `make run`: ejecuta el simulador con el archivo `icache.img` previamente generado. Durante esta ejecución, se imprimen en la terminal los estados de los registros y la información temporal simulada.

Además, el script debía leer la salida generada en la terminal después de cada ejecución para extraer información relevante. Cada ejecución generaba bloques de texto como el siguiente:

```
FU ALERT: **BRANCH**
------------------------
IFU : mem=0x0
IFU : pc= 15 at CSIM 1272 ns
------------------------
                        ------------------------------
                        ID: clear branch at CSIM 1273 ns
                        ------------------------------

                        ******************************
                        ID: REGISTERS DUMP at CSIM 1274 ns
                        ******************************
REG :=========================================================================
  R 0(00000000)    R 1(00000005)    R 2(00000005)    R 3(f...ffff)
  ...
=========================================================================
```

A partir de este bloque, era posible obtener tanto el tiempo de simulación total como los valores finales de los registros. En particular, se podía identificar cuántas iteraciones había necesitado el algoritmo para llegar al resultado observando cuántas veces se había ejecutado la instrucción `blt`.

De esta forma, se automatizó todo el proceso de evaluación, permitiendo ejecutar múltiples pruebas de forma rápida, registrar los tiempos de ejecución y generar estadísticas comparativas entre los diferentes casos de prueba.

En particular, se desarrollaron con el apoyo de herramientas de inteligencia artificial dos scripts en Python: el primero, `gcd_auto.py`, genera automáticamente 30 casos de prueba con entradas aleatorias; el segundo, `gcd_prueba.py`, contiene 30 vectores de prueba específicos diseñados para comparar los resultados de la implementación en software con la solución en hardware.

Los resultados obtenidos se almacenan automáticamente en un archivo CSV (`resultados.csv`), el cual contiene las columnas: número de prueba, valores iniciales de `R1` y `R2`, valor esperado, valor obtenido, cantidad de ciclos de simulación e iteraciones requeridas por el algoritmo. Ambos scripts, junto con el resto del proyecto, están disponibles en el repositorio de GitHub (5).

Es importante mencionar que, para el cálculo del número de ciclos, el reloj del simulador RISC está configurado con un periodo de 1 nanosegundo. Esto se puede confirmar en el archivo `main.cpp`, donde aparece la siguiente línea de código:

```
sc_clock clk("Clock", 1, SC_NS, 0.5, 0.0, SC_NS);
```

3.3. Resultados de simulación

Los siguientes resultados corresponden a los 30 vectores de prueba definidos para validar la implementación:

Prueba	R1	R2	Esperado	Obtenido	Ciclos	Iteraciones
1	18	30	6	6	377	3
2	20	5	5	5	347	3
3	9	16	1	1	620	6
4	48	16	16	16	276	2
5	100	101	1	1	7249	100
6	256	256	256	256	134	0
7	840	1260	420	420	291	2
8	17	34	17	17	220	1
9	13	19	1	1	792	8
10	1024	768	256	256	377	3
11	0	27	0	0	106	0
12	0	0	0	0	106	0
13	4	6	2	2	291	2
14	1	100	1	1	8648	99
15	9999	3	3	3	236706	3332
16	25	49	1	1	2269	25
17	81	27	27	27	276	2
18	360	840	120	120	448	4
19	128	32	32	32	347	3
20	9999	9996	3	3	286671	3332
21	123	41	41	41	276	2
22	36	81	9	9	519	5
23	8	27	1	1	691	7
24	100	75	25	25	377	3
25	14	25	1	1	706	7
26	12	35	1	1	1237	13
27	54	72	18	18	362	3
28	24	36	12	12	291	2
29	50	100	50	50	220	1
30	546	2310	42	42	934	10

Se puede observar que en los 30 casos evaluados el resultado obtenido coincide con el valor esperado, lo que permite confirmar el correcto funcionamiento de la solución implementada en software.

Adicionalmente, es importante destacar que los valores de entrada que mayor cantidad de ciclos requirieron fueron `R1 = 9999` y `R2 = 9996`, con un total de 286671 ciclos, debido a que se necesitaron 3332 iteraciones del algoritmo de Euclides para hallar el máximo común divisor. En contraste, cuando alguno de los dos valores era cero, el número de ciclos se redujo a solo 106, ya que no fue necesaria ninguna iteración del algoritmo.

4. Hardware

4.1. Descripción ALU

El primer paso realizado para la implementación del algoritmo fue describir la unidad logico-aritmetica mostrada en la figura 1. Como se puede apreciar está cuenta con diversos componentes tales como:

- **Negador**: Para calcular la negación de la entrada B.
- **MUX 2:1**: Selecciona entre B y $\overline{B}$
- **Compuertas lógicas**: AND y OR para operaciones básicas.
- **Sumador**: Con entrada de carry (F_2) para complemento a dos.

- **Zero extended**: Extiende el bit más significativo del resultado a N bits.
- **MUX 4:1**: Selecciona la operación final mediante 2 bits de control ($F_{1:0}$).

A partir de esta identificación, se comienza con la implementación en código de cada una de las respectivas partes. Para una mayor facilidad se definen 3 módulos :

1. **Modulo Principal:** Este modulo comprende las compuertas lógicas (Negador, AND y OR), el sumador con su carry out y el Zero extended (con su respectiva lógica). Así mismo abarca las conexiones identificadas como ***sc_ signal*** necesarias para la transmisión de datos con los submódulos siguientes (Nombre de archivo : *ALU.h*).
2. **Sub-Modulo 1**: Este modulo comprende la descripción del Mux de 4 entradas y una única salida. La selección que valor se ve reflejado en la salida, se da mediante la lectura del selector y un switch case (Nombre de archivo : *MUX_ 1_ 4.h*).
3. **Sub-Modulo 2**: Este modulo comprende la descripción del Mux de 2 entradas. De igual manera que el sub-modulo 1, la selección se da mediante un switch case *MUX_ 1_ 2.h*).

A manera de comprobar el correcto funcionamiento de la ALU, se realiza un testbench que verifica la resta, la operación AND, la suma y la extensión del bit mas significativo. Las señal *A_ in*, *B_ in* representan los valores de entrada a la ALU, la señal F representa el selector y la señal Y representa la salida.

Para una mejor comprensión se muestra una tabla con las funciones que realiza la ALU en función del selector (*Aclaración: No se contemplan todos los casos posibles, solo los necesarios para la implementación)

Tabla 1: Funciones seleccionadas de la ALU

$F_{2:0}$	Función
000	$A \wedge B$ (AND bit a bit)
010	$A + B$ (Suma)
110	$A - B$ (Resta)
111	MSB (Most Significant Bit)

Estos resultados se presentan en las siguientes figuras:

- **Resta**:

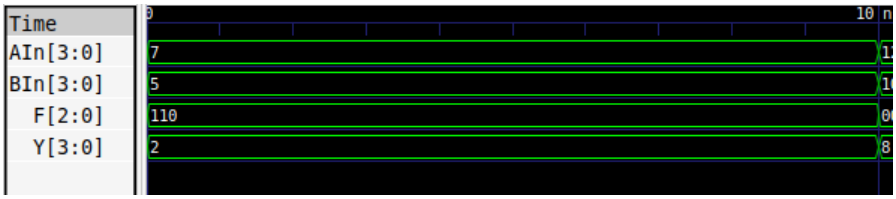

Figura 2: Gtkwave de la ALU para operar resta. $A = 5 \quad B = 7$

- **Suma**:

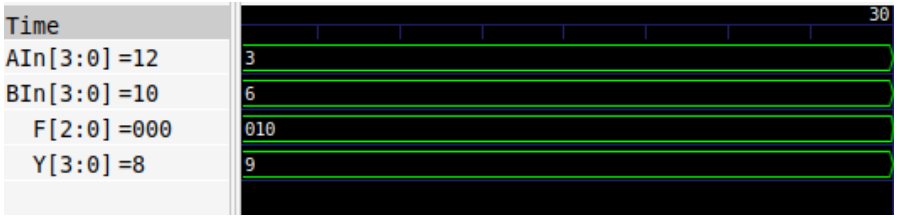

Figura 3: Gtkwave de la ALU para operar suma. $A = 3 \quad B = 6$

- **AND**:

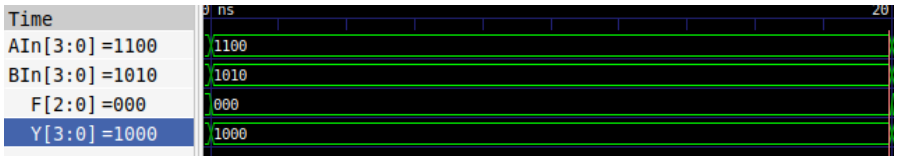

Figura 4: Gtkwave de la ALU para operar AND. $A = 1100 \quad B = 1010$

- **MSB**:

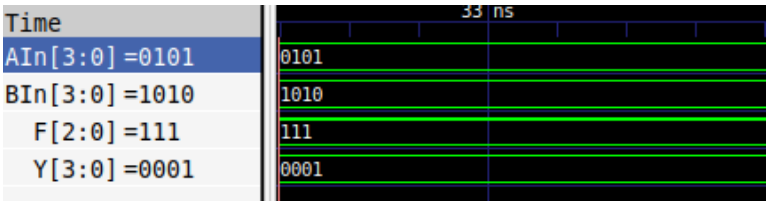

Figura 5: Gtkwave de la ALU para obtener el MSB con Zero Extended. $A = 3 \quad B = 6$

4.2. Descripción Unidad de Control

Una vez implementada correctamente la ALU, se plantea una unidad de control capaz de ejecutar el algoritmo de Euclides para obtener el Máximo Común Divisor (*Greatest Common Divisor (GCD)*). De esta manera, se plantea la siguiente maquina de estados para realizar el control de la ALU:

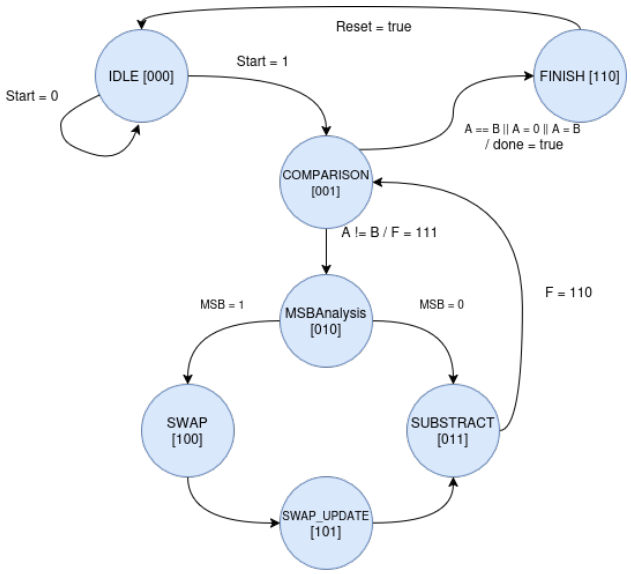

Figura 6: FSM diseñada para control de ALU

A partir del esquemático mostrado en la figura 6 se plantea la lógica secuencial requerida para realizar la respectiva maquina de estados. Es importante aclarar también que se contemplan 4 *sc_methods* con funciones especificas que definen el comportamiento de la FSM:

- ***state_register***: Método encargado de cambiar el estado actual de la maquina de estados de acuerdo al definido en la trayectoria de funcionamiento. Es sensible al flanco de subida del reloj y al reset.

```
void state_register() {
    if (reset.read()) {
        current_state.write(IDLE);
    } else {
        current_state.write(next_state.read());
    }
}
```

- ***next_state_logic***: En este método se define la ruta de transición de estados en función de las condiciones establecidas en la figura 6.

```
void next_state_logic() {
    auto state = current_state.read();
    sc_uint<3> next = state;

    switch (state) {
        case IDLE:
            if (start.read()) next = COMPARISON;
            break;
        case COMPARISON:
        next = (A_reg.read() == 0 || B_reg.read() == 0 || A_reg.read() == B_reg.read()) ?
        break;
        case MSBAnalysis:
            next = (Y.read() == 0) ? SUBSTRACT : SWAP;
            break;
```

```
            case SWAP:
                next = UPDATE_SWAP;
                break;
            case UPDATE_SWAP:
                next = SUBSTRACT;
                break;
            case SUBSTRACT:
                next = COMPARISON;
                break;
            case FINISH:
                next = IDLE;
                break;
        }

        next_state.write(next);
    }
```

- ***output_ logic***: En este método se define la lógica de salida en cada estado, define el valor de la señal F (selector) la cual determina que salida se ve reflejada en la ALU.

```
    void output_logic() {
        Aout.write(0);
        Bout.write(0);
        F.write(0);
        done.write(false);

        switch (current_state.read()) {
            case COMPARISON:
                Aout.write(A_reg.read());
                Bout.write(B_reg.read());
                break;

            case MSBAnalysis:
                Aout.write(A_reg.read());
                Bout.write(B_reg.read());
                F.write(111); // MSB
                break;

            case SWAP:
                break;

            case SUBSTRACT:
                Aout.write(A_reg.read());
                Bout.write(B_reg.read());
                F.write(110); // resta
                break;

            case FINISH:
                result.write(A_reg.read());
                done.write(true);
                break;

            default:
                break;
```

```
        }
    }
```

- ***register_logic***: En este método una vez se da start, se actualizan los registros *A_reg* y *B_reg* los cuales son una copia de las entradas con el fin de ejecutar las restas iterativas sobre estas y al finalizar la maquina de estados sus valores finales se cargan en una señal que se muestra en consola, la cual contiene el resultado final.

```
    void register_logic() {
if (reset.read()) {
    A_reg.write(0);
    B_reg.write(0);
    reg_temp.write(0);
} else {
    switch (current_state.read()) {
        case IDLE:
            A_reg.write(Ain.read());
            B_reg.write(Bin.read());
            break;
        case COMPARISON:
            break;
        case SWAP:
            reg_temp.write(A_reg.read());
            break;
        case UPDATE_SWAP: //Permite un ciclo de reloj adicional para que se refleje el cambio
            A_reg.write(B_reg.read());
            B_reg.write(reg_temp.read());
            break;
        case SUBSTRACT:
            A_reg.write(Y.read());
            break;

        default:
            // No se modifican registros en los demás estados
            break;
    }
}
}
```

4.3. Implementación top

Una vez realizada la descripción e implementación en systemC tanto de la ALU como de la Unidad de control, se plantea un archivo top, el cual realiza el englobe de ambos bloques funcionales. Asi mismo dentro de este se realizan las conexiones respectivas entre ambos, como se observa la siguiente figura que representa la conexión y la jerarquia:

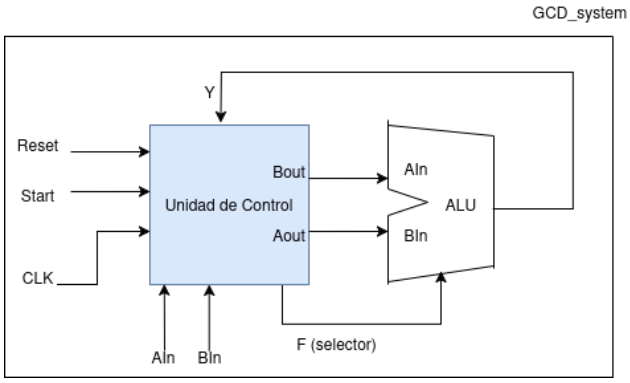

Figura 7: Esquematico modulo TOP que abarca ALU y U.C

Cabe resaltar que en este caso las entradas AIn y BIn que ingresan en la unidad de control son proporcionadas por el testbench para realizar las respectivas pruebas de funcionamiento.

Aclaración El modulo se encuentra definido como ***GCD_System.h***

4.4. Automatización y pruebas realizadas

Finalmente tras definir el modulo principal, se define un archivo de Testbench el cual contiene las 30 pruebas solicitadas, los valores cargados son valores predefinidos, el propósito de esto ultimo es poder comparar un valor esperado vs el obtenido, esto como un indicador de correcto funcionamiento.

Para una mayor comprensión de la interacción entre los diferentes módulos se presenta el siguiente esquemático:

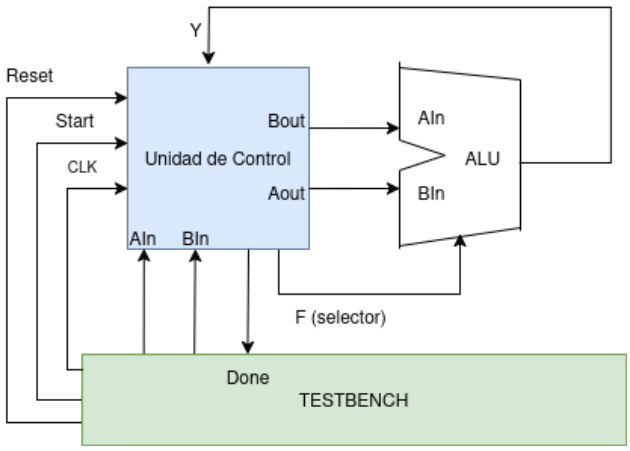

Figura 8: Conexion banco de pruebas para comprobar funcionamiento

Es importante aclarar que el código del banco de pruebas fue generado mediante inteligencia artificial (DeepSeek). Adicionalmente, al finalizar cada prueba se genera una bandera (***done***) que permite identificar que la iteración ha terminado, con esto tras cada prueba se realiza un reset para reiniciar los valores de los registros internos de la U.C. y se cargan los nuevos valores establecidos en el banco de pruebas (Este archivo se nombra ***GCD_tb.cpp*** dentro del GitHub).

Así entonces, cada resultado se presenta en la consola de la siguiente manera:

```
Prueba #1: A = 18, B = 30 | Resultado: 6 (Esperado: 6) | Ciclos: 17
Prueba #2: A = 20, B = 5 | Resultado: 5 (Esperado: 5) | Ciclos: 11
Prueba #3: A = 9, B = 16 | Resultado: 1 (Esperado: 1) | Ciclos: 28
Prueba #4: A = 48, B = 16 | Resultado: 16 (Esperado: 16) | Ciclos: 8
```

Esto permite identificar el numero de prueba realizado, los valores de A y B, el resultado obtenido tras la ejecución del proceso y el resultado esperado. Así mismo se muestran cuantos ciclos de reloj son necesarios para llegar la resultado.

4.5. Resultados de simulación

Finalmente se presenta la siguiente tabla donde se enseñan todos los casos planteados y sus respectivos resultados:

Prueba	A	B	Esperado	Obtenido	Ciclos
1	18	30	6	6	17
2	20	5	5	5	11
3	9	16	1	1	28
4	48	16	16	16	8
5	100	101	1	1	306
6	256	256	256	256	2
7	840	1260	420	420	12
8	17	34	17	17	7
9	13	19	1	1	32
10	1024	768	256	256	14
11	0	27	0	0	2
12	0	0	0	0	2
13	4	6	2	2	12
14	1	100	1	1	301
15	9999	3	3	3	9998
16	25	49	1	1	83
17	81	27	27	27	8
18	360	840	120	120	18
19	128	32	32	32	11
20	9999	9996	3	3	10001
21	123	41	41	41	8
22	36	81	9	9	21
23	8	27	1	1	31
24	100	75	25	25	13
25	14	25	1	1	33
26	12	35	1	1	47
27	54	72	18	18	15
28	24	36	12	12	12
29	50	100	50	50	7
30	546	2310	42	42	39

Asi mismo el GTKWave asociado a las pruebas se encuentra en el archivo ***gcd_ wave.vcd***

5. Comparación de resultados entre Software y Hardware

Para poder realizar una comparación directa entre la eficiencia de las soluciones de software y hardware, se presenta la siguiente tabla, que evidencia las iteraciones del algoritmo y la relación entre los ciclos de cada solución.

Prueba	Ciclos SW	Ciclos HW	Iteraciones	SW/HW
1	377	17	3	22.18
2	347	11	3	31.55
3	620	28	6	22.14
4	276	8	2	34.50
5	7249	306	100	23.69
6	134	2	0	67.00
7	291	12	2	24.25
8	220	7	1	31.43
9	792	32	8	24.75
10	377	14	3	26.93
11	106	2	0	53.00
12	106	2	0	53.00
13	291	12	2	24.25
14	8648	301	99	28.73
15	236706	9998	3332	23.67
16	2269	83	25	27.34
17	276	8	2	34.50
18	448	18	4	24.89
19	347	11	3	31.55
20	286671	10001	3332	28.66
21	276	8	2	34.50
22	519	21	5	24.71
23	691	31	7	22.29
24	377	13	3	29.00
25	706	33	7	21.39
26	1237	47	13	26.34
27	362	15	3	24.13
28	291	12	2	24.25
29	220	7	1	31.43
30	934	39	10	23.95

Es posible observar cómo la solución de hardware resultó ser mucho más eficiente en cuanto al número de ciclos que le tomaba resolver cada prueba. En particular, para los casos en que los dos números eran iguales o alguno era 0, es decir, cuando no se necesitaba ninguna iteración del algoritmo de Euclides, se puede observar la mayor diferencia entre las soluciones, ya que la solución de hardware llegó a ser 67 veces más eficiente que la solución en software.

Esto demuestra entonces el beneficio en el rendimiento que se puede obtener con una solución especializada basada en hardware para un problema específico. Sin embargo, es importante mencionar que la solución en software ofrecía mayor versatilidad y mucha mayor facilidad para modificar la solución y añadir o quitar funcionalidades.

Referencias

[1] SystemC Website `https://systemc.org/overview/systemc/`

[2] Digital Design and Computer Architecture - David Harris - Sarah Harris. `https://www.r-5.org/files/books/computers/hw-layers/hardware/digital-desigh/David_Harris_Sarah_Harris-Digital_Design_and_Computer_Architecture-EN.pdf`.

[3] OpenAI, "ChatGPT". `https://chat.openai.com`.

[4] DeepSeek, "DeepSeek Chat". `https://chat.deepseek.com`.

[5] L. Vaca, "Repositorio GCD_SW-HW – Implementación en SystemC (software y hardware)," GitHub, 2025. [Online]. Available: https://github.com/LuisVaca1503/GCD_SW-HW.

[6] IEEE Standard for Standard SystemC Language Reference Manual, IEEE Std 1666-2011. [Online]. Available: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://staff.fnwi.uva.nl/e.h.steffens/Downloads/Arco/Innopolis/Lab7/SystemC_Reference_Manual.pdf&ved=2ahUKEwjXp4jH3v6NAxVVTTABHSfRCHYQFnoECCAQAQ&usg=AOvVaw30FpovAPfR7-iNM8BONdHJ