Students
Rosemberth Steeven Preciga Puentes
Luis Guillermo Vaca Rincón

Practice 1: Implementations hardware/software

1. Description

The main purpose of this practice was to get started with tools and description languages focused on system-level design (SLD), specifically the modeling of hardware and software using SystemC. By the end, the goals were:

- Become familiar with high-level simulation and description environments for digital systems, specifically SystemC.
- Compare the implementation of an application when it is described entirely in hardware versus an implementation in a hardware/software co-design environment.
- Develop skills to automate tests through the generation of input vectors and their subsequent analysis.

To achieve these objectives, we implemented the Greatest Common Divisor (GCD) algorithm both in its software version (executed on a RISC processor modeled in SystemC) and in its hardware version, including the corresponding arithmetic-logic unit and the control machine that sequences its operations.

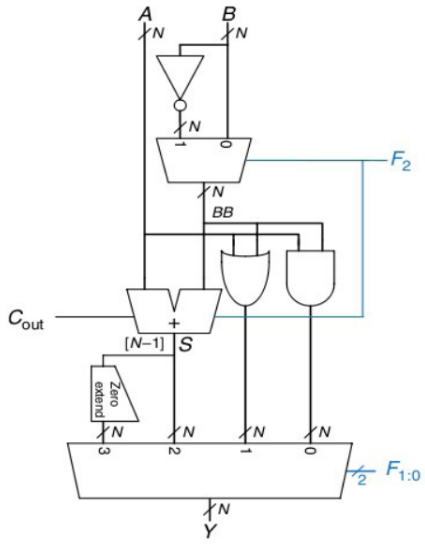

Figura 1: Datapath to be implemented using SystemC, image taken from (2)

In addition, at least 30 input combinations were applied to evaluate the system behavior under different conditions, and the results were presented through simulations using `gtkwave`.

1.1. Greatest Common Divisor (GCD) Algorithm

Euclid's algorithm to compute the greatest common divisor between two positive integers X and Y is based on successive subtractions until both numbers are equal. The pseudocode can be expressed as:

$$\textbf{while } X \neq Y \textbf{ do} \begin{cases} Y \leftarrow Y - X & \text{if } X < Y \\ X \leftarrow X - Y & \text{otherwise} \end{cases} \tag{1}$$

Once the condition $X = Y$ holds, the resulting value is the **greatest common divisor**:

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

2. Theoretical Framework

2.1. SystemC

SystemC is a C++-based language that enables the design, modeling, and verification of complex electronic systems, integrating both hardware and software on the same platform. It emerged to address the need to unify the development of these two worlds, facilitating the creation of embedded systems and more efficient chips at the system-level design (SLD) stage.

Its main advantage is flexibility: it allows testing different architectures, partitioning functions between hardware and software, and simulating system behavior prior to fabrication. Major semiconductor and electronics companies use it to explore designs, optimize performance, and accelerate the development of virtual platforms.

Moreover, being standardized (IEEE 1666) ensures compatibility and adoption in the industry. In projects like ours (HW/SW GCD), SystemC is useful for comparing both implementations and validating their integrated operation (1).

2.2. Operation of the RISC Processor in SystemC

The project implements a simple RISC-architecture processor using SystemC. It consists of three main modules: an instruction cache, a decoding unit, and an execution unit. Each of these modules is connected and synchronized through a simulated clock, enabling cycle-by-cycle modeling of processor behavior.

2.2.1. Instruction Flow

During simulation, in each clock cycle one part of the instruction cycle is executed. First, an instruction is fetched from memory (`icache`), then it is decoded (`decode`) to identify the operation and operands, and finally it is executed (`exec`). If the instruction produces a result, such as a sum, that result is written to the corresponding register. The instructions are stored in a file `icache.img`, previously generated from assembly code by the script `assembler.pl`.

2.2.2. Decoding and Registers

The decoding unit interprets the instruction code and accesses an array that simulates the processor registers. It also determines whether the instruction is a branch, sum, comparison, etc., and issues the corresponding control signals to the execution unit. If there is a branch, the decoder modifies the program counter (PC) to continue execution from the new address.

2.2.3. Execution Unit

The execution unit performs the operation indicated by the instruction (for example, a subtraction or comparison between registers). It then sends the result to the decoder so that it can update the destination register. For branch instructions, the condition is also evaluated and it is confirmed whether the branch should be taken.

2.2.4. Simulation in SystemC

The entire system is synchronized with a clock defined with 1 ns precision. In each cycle, signals between modules are updated, simulating how a physical processor would behave. Thanks to this, we can observe step by step how each instruction is executed.

3. Software

3.1. Implementation Description

First, we analyzed the instructions available in the assembly language to implement Euclid's algorithm. We determined that the following instructions were sufficient to implement it correctly.

- `movi R, val`: Loads an immediate value into register `R`.
- `sub R1, R2, R3`: Subtracts the contents of `R3` from `R2` and stores the result in `R1`.
- `beq R1, R2, dir`: Branches to address `dir` if `R1` equals `R2`.
- `blt R1, R2, dir`: Branches to address `dir` if `R1` is less than `R2`.
- `j dir`: Unconditional jump to address `dir`.
- `halt`: Ends program execution.

During implementation we encountered an important issue: the program did not seem to execute branch instructions correctly. After reviewing `decode.cpp`, we noticed that for instructions like `beq`, although the branch target address was correctly computed using `pc_reg + label_tmp`, at the end of the block a new sequential address was written using `pc_reg + 1`.

This overwrote the previously computed address, causing the processor to ignore the branch and continue with sequential execution. The relevant code fragment is shown below:

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

As can be seen, the commented line at the end is the one that overwrote the branch. For this reason, it was necessary to comment it out not only here but also in the other blocks that handled conditional branch instructions, such as `blt`, so that the program could correctly execute the algorithm's logic.

Having said that, during simulation we detected another unexpected behavior in the file `icache.img`, generated by the assembler `assembler.pl`. Specifically:

- Before the first instruction was transcribed, the file contained several empty addresses (`0x00000000`).

- After each assembled instruction, an additional empty cycle (`0x00000000`) was inserted into memory.

We then analyzed the code in `assembler.pl` and confirmed that by default it transcribed five empty addresses before the `.asm` instructions and inserted an additional empty address after each assembled instruction:

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

...

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

This was important when writing the assembly code, since the branch target values should not correspond to the position of the target instruction within the `.asm` file, but rather to its actual address in memory within `icache.img`.

During testing we observed that if we tried to branch directly to the address where the target instruction resided, it would not execute. Instead, it was necessary to branch to the preceding empty line that the assembler automatically inserted before each instruction.

Taking these considerations into account, the following assembly code implements the GCD function:

```
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
```

These are the instructions assembled and stored in `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
```

As can be seen, the instructions are encoded in hexadecimal format. To prevent the program from entering an infinite loop in case either input was zero, two `beq` instructions were added at the beginning to stop execution if `R1` or `R2` is zero.

3.2. Automating the Analysis with Python

To meet the requirement of generating at least 30 different input values, we designed a Python script. The idea was to automate testing of the GCD algorithm by dynamically modifying registers `R1` and `R2` with random values, compiling, and executing the program in each iteration.

After modifying the inputs, the script sequentially executed the following system commands:

- `perl assembler.pl gcd.asm -code >icache.img`: assembles the source file `gcd.asm` and generates `icache.img` containing the hexadecimal instructions for the simulator.
- `make clean`: removes temporary files generated during the previous build to ensure a clean start.
- `make`: compiles the simulator source files (SystemC) and generates the main executable, typically `risc_cpu.x`.
- `make run`: runs the simulator with the previously generated `icache.img`. During execution, register states and simulated time information are printed to the terminal.

The script also read the terminal output after each run to extract relevant information. Each execution generated text blocks such as:

```
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)
  ...
=========================================================================
```

From this block, it was possible to obtain both the total simulation time and the final register values. In particular, we could identify how many iterations the algorithm required by observing how many times the `blt` instruction was executed.

In this way, the entire evaluation process was automated, allowing multiple tests to run quickly, logging execution times, and generating comparative statistics across test cases.

In particular, with the support of AI tools we developed two Python scripts: `gcd_auto.py`, which automatically generates 30 random test cases; and `gcd_prueba.py`, which contains 30 specific test vectors designed to compare the software implementation with the hardware solution.

The results are stored automatically in a CSV file (`resultados.csv`), containing the columns: test number, initial values for `R1` and `R2`, expected value, obtained value, number of simulation cycles, and iterations required by the algorithm. Both scripts, along with the rest of the project, are available in the GitHub repository (5).

It is worth mentioning that, for cycle counting, the RISC simulator clock is configured with a 1-nanosecond period. This can be confirmed in `main.cpp` in the following line:

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

3.3. Simulation Results

The following results correspond to the 30 test vectors defined to validate the implementation:

Test	R1	R2	Expected	Obtained	Cycles	Iterations
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

It can be observed that in all 30 cases the obtained result matches the expected value, confirming the correct operation of the software implementation.

Additionally, it is important to highlight that the inputs requiring the greatest number of cycles were `R1 = 9999` and `R2 = 9996`, with a total of 286671 cycles, due to 3332 iterations of Euclid's algorithm. In contrast, when either value was zero, the number of cycles was only 106, since no iterations were required.

4. Hardware

4.1. ALU Description

The first step for implementing the algorithm was to describe the arithmetic-logic unit shown in Figure 1. As can be seen, it includes several components such as:

- **Inverter**: Computes the negation of input B.
- **2:1 MUX**: Selects between B and $\overline{B}$.
- **Logic gates**: AND and OR for basic operations.
- **Adder**: With carry input (F_2) for two's complement.
- **Zero extended**: Extends the most significant bit of the result to N bits.

- **4:1 MUX**: Selects the final operation using 2 control bits ($F_{1:0}$).

Based on this identification, we began coding each part. For simplicity, three modules were defined:

1. **Main Module:** This module comprises the logic gates (Inverter, AND, OR), the adder with its carry out, and the zero-extended logic. It also includes the connections identified as ***sc_ signal*** required for data transmission with the following submodules (Filename: *ALU.h*).
2. **Submodule 1**: Describes the 4-input, 1-output MUX. Selection is based on the selector value via a switch case (Filename: *MUX_ 1_ 4.h*).
3. **Submodule 2**: Describes the 2-input MUX. As in Submodule 1, selection is via a switch case (*MUX_ 1_ 2.h*).

To verify correct ALU operation, a testbench was created to check subtraction, the AND operation, addition, and the most significant bit extension. Signals *A_ in* and *B_ in* represent ALU inputs, F is the selector, and Y is the output.

For better understanding, the following table shows the functions performed by the ALU as a function of the selector (*Note: not all possible cases are considered, only those necessary for the implementation.)

Tabla 1: Selected ALU functions

$F_{2:0}$	Function
000	$A \wedge B$ (bitwise AND)
010	$A + B$ (Addition)
110	$A - B$ (Subtraction)
111	MSB (Most Significant Bit)

The following figures show these results:

- **Subtraction**:

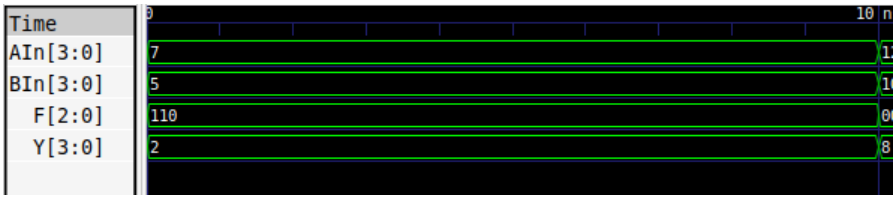

Figura 2: GTKWave of the ALU performing subtraction. $A = 5 \quad B = 7$

- **Addition**:

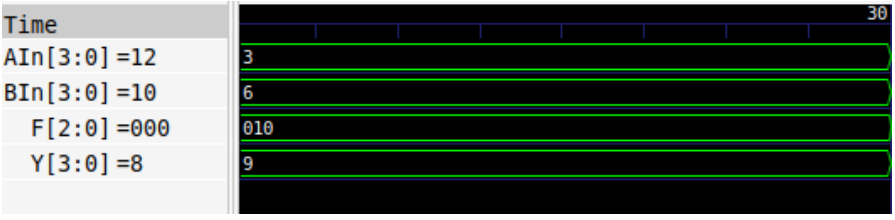

Figura 3: GTKWave of the ALU performing addition. $A = 3 \quad B = 6$

- **AND**:

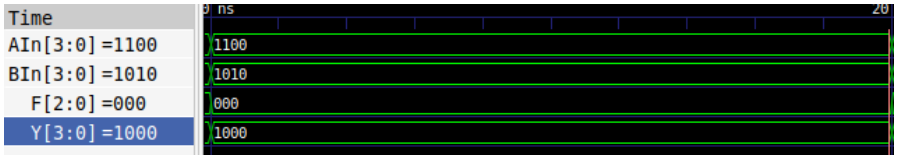

Figura 4: GTKWave of the ALU performing AND. $A = 1100 \quad B = 1010$

- **MSB**:

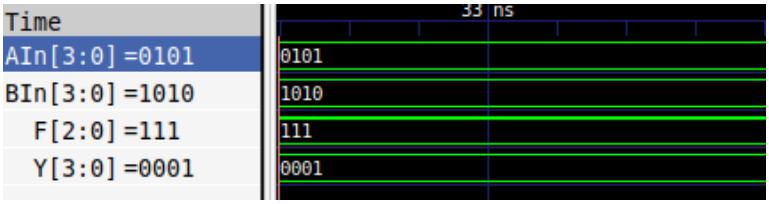

Figura 5: GTKWave of the ALU obtaining the MSB with zero extension. $A = 3 \quad B = 6$

4.2. Control Unit Description

Once the ALU was correctly implemented, we designed a control unit capable of executing Euclid's algorithm to obtain the Greatest Common Divisor (GCD). The following state machine was proposed to control the ALU:

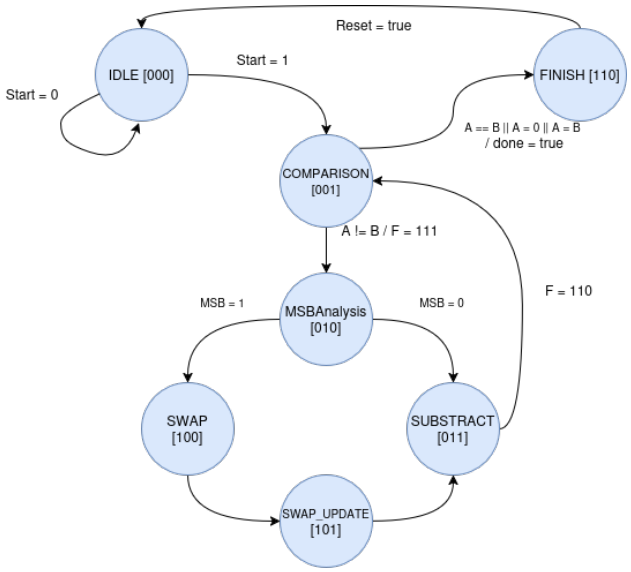

Figura 6: FSM designed for ALU control

From the schematic in Figure 6, we defined the required sequential logic to implement the state machine. It is also important to clarify that four *sc_ methods* with specific functions define the FSM behavior:

- ***state_register***: Method in charge of updating the current state of the state machine according to the defined operation flow. It is sensitive to the rising edge of the clock and to reset.

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

- ***next_state_logic***: This method defines the state transition path according to the conditions in Figure 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***: This method defines the output logic in each state, setting the value of signal F (selector) which determines the ALU output.

```
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); // subtraction
                break;

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

            default:
                break;
        }
    }
```

- ***register_logic***: In this method, once *start* is asserted, registers *A_reg* and *B_reg* are updated as copies of the inputs so that the iterative subtractions are applied to them. At the end of the state machine their final values are loaded into a signal printed on the console containing the final result.

```
    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. Top Implementation

After describing and implementing in SystemC both the ALU and the control unit, we created a top file that encapsulates both functional blocks. It also makes the connections between them, as shown in the following figure representing the connection and hierarchy:

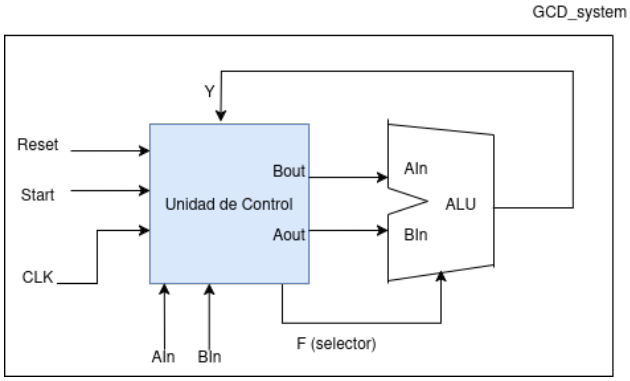

Figura 7: Top module schematic encompassing ALU and control unit

It should be noted that in this case the *AIn* and *BIn* inputs to the control unit are provided by the testbench to perform the functional tests.

Note The module is defined as ***GCD_System.h***

4.4. Automation and Performed Tests

Finally, after defining the main module, a testbench file was created containing the 30 required tests. The loaded values are predefined in order to compare an expected value vs. the obtained result as an indicator of correct operation.

For a better understanding of the interaction between the different modules, the following schematic is presented:

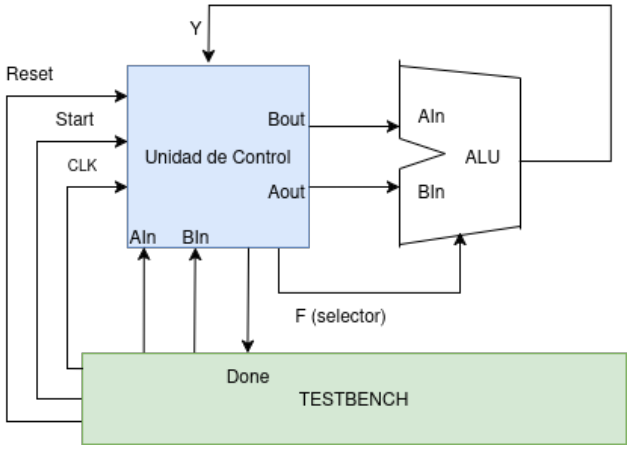

Figura 8: Testbench connections to verify operation

It is important to clarify that the testbench code was generated using artificial intelligence (DeepSeek). Additionally, at the end of each test a flag (***done***) indicates that the iteration has finished. After each test a reset is performed to reinitialize the internal registers of the control unit, and the new values established in the testbench are loaded (This file is named ***GCD_ tb.cpp*** in GitHub).

Thus, each result is presented on the console as follows:

```
Test #1: A = 18, B = 30 | Result: 6 (Expected: 6) | Cycles: 17
Test #2: A = 20, B = 5 | Result: 5 (Expected: 5) | Cycles: 11
Test #3: A = 9, B = 16 | Result: 1 (Expected: 1) | Cycles: 28
Test #4: A = 48, B = 16 | Result: 16 (Expected: 16) | Cycles: 8
```

This allows us to identify the test number, the values of A and B, the result obtained after execution, the expected result, and the number of clock cycles required to reach the answer.

4.5. Simulation Results

Finally, the following table shows all the proposed cases and their respective results:

Test	A	B	Expected	Obtained	Cycles
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

Likewise, the GTKWave file associated with the tests is ***gcd_wave.vcd***.

5. Comparison of Results Between Software and Hardware

To make a direct comparison between the efficiency of the software and hardware solutions, the following table shows the algorithm iterations and the ratio between cycles for each solution.

Test	SW Cycles	HW Cycles	Iterations	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

It is clear that the hardware solution is much more efficient in terms of the number of cycles required to solve each test. In particular, for cases where the two numbers were equal or one was 0, i.e., when no iterations of Euclid's algorithm were needed, the greatest difference between the solutions can be observed, with the hardware solution being up to 67 times more efficient than the software solution.

This demonstrates the performance benefits achievable with a specialized hardware-based solution for a specific problem. However, it is important to mention that the software solution offered greater versatility and much easier modification when adding or removing features.

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