EE5516 VLSI Architectures for Signal Processing and Machine Learning

Lab Report - Experiment No.3

Nakul C - 122101024

May 27, 2024

Abstract

The objective of this experiment is to determine the Greatest Common Divisor (gcd) of two positive integers, A and B, by employing Euclid's algorithm. According to Euclid's principle, $\mathbf{gcd}(A, B) = \mathbf{gcd}(B, A\%B)$. The methodology adopted involves hierarchical modeling, which entails breaking down the problem into manageable components. The experiment consists of two main modules: the control path and the data path. These modules are interconnected through control and status signals, allowing them to collaborate effectively. The data path is constructed using a structural approach, where the module is built from simpler components. Conversely, the control path utilizes Finite State Machine (FSM) abstraction to regulate the algorithm's overall flow. The primary aim is to develop an efficient and optimized implementation of the gcd algorithm while minimizing the utilization of hardware resources.

1 Introduction

The aim of this experiment is to find the greatest common divisor (GCD) of two positive integers, denoted as A and B. The GCD signifies the largest positive integer that can evenly divide both numbers. We'll utilize Euclid's algorithm, known for its simplicity and efficiency in determining the GCD. Before delving into the algorithm, it's essential to represent our digital system as a block that takes two positive integers as input and outputs their GCD. To ensure input and output reliability, qualifiers like "operands_valid" and "gcd_valid" will be used. Additionally, a clock and reset signal will synchronize and reset the system, respectively. To facilitate module communication effectively, we'll employ the acknowledgment (ack) protocol.

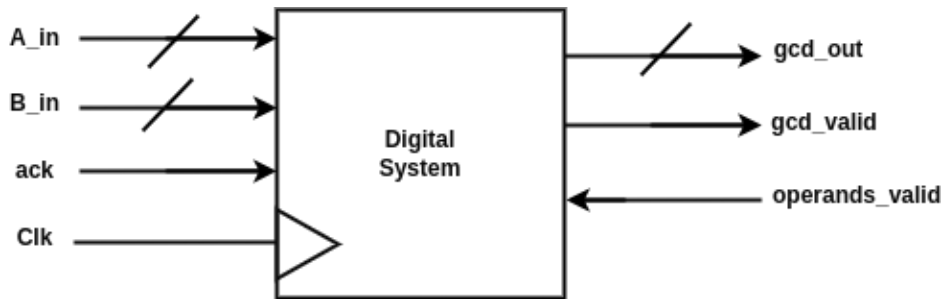

Figure 1: Block Diagram

2 Implementation

To implement the system, Euclid's algorithm needs to be translated into hardware architecture. This algorithm operates under the principle that the Greatest Common Divisor (GCD) of two numbers remains

unaltered if the smaller number is subtracted from the larger one. The algorithm progresses through a series of step

```
1. Accept inputs A_in and B_in.
2. If B_in = 0, then GCD(A_in, B_in) = A_in. return A_in.
3. Otherwise, if (B_in > A_in) then swap (A_in, B_in) else
   compute the remainder, r when A_in is divided by B_in.
4. Replace A_in with B_in and B_in with r.
5. Go back to step 2.
6. Repeat steps 2-5 until B_in is zero. The final value of
   A_in is the GCD of the original two numbers.
```

Although seemingly straightforward, hardware implementation of Euclid's algorithm demands a systematic approach to optimize performance and conserve hardware resources. A significant challenge lies in computing remainders when dividing two numbers, particularly in digital systems. To address this challenge, the modulus operator can be replaced with a subtractor, as repeated subtraction effectively mimics division and upholds the validity of the remainder property.

By iteratively subtracting the smaller number from the larger one, the remainder can be determined, and the algorithm can continue until the remainder equals zero. Implementing hardware can be further improved by adopting a pipeline architecture, enabling parallel execution of multiple operations to enhance system throughput. Leveraging optimized hardware components such as barrel shifters and adders facilitates efficient subtraction and comparison operations.

Constructing the necessary architecture requires several essential hardware components. Primarily, registers are necessary to store input values A_{in} and B_{in}. D flip-flops equipped with an enable signal serve this purpose, allowing the flip-flop to either accept new inputs or retain previous values based on the enable signal. MUXes can be used to select appropriate inputs for the registers, requiring control signals for the MUXes. Comparators play a crucial role in comparing A and B, as well as B and zero, signifying the termination condition for Euclid's algorithm. Assuming A is always greater than B, a comparison between them is necessary. If this assumption is violated, A and B must be swapped to ensure a positive difference. A subtractor is essential to compute the remainder, while criss-cross connections at the hardware level facilitate swapping.

In addition to hardware resources, a 3-state finite state machine (FSM) orchestrates the sequencing of inputs and outputs. This ensures each process completes before the next one commences and guarantees that the loop persists until B reaches zero, marking the termination condition for the algorithm.

2.1 Structural Design

To effectively manage larger values of A and B, our system integrates 32-bit registers labeled A and B within its architecture. This configuration involves two Multiplexers (MUXes) - a 3:1 MUX dedicated to A and a 2:1 MUX allocated for B. These MUXes dictate the inputs to the registers at each stage, guided by control signals A_MUX_sel and B_MUX_sel. Both A and B registers share a common clock and possess enable signals A_en and B_en respectively.

The system initializes in the IDLE state and transitions to an operational state once the qualifier "operands_valid" asserts to 1. Upon this assertion, the system progresses to the BUSY state, facilitating the transmission of A_in and B_in to the input port of the registers. These inputs are processed by the registers at the subsequent positive clock edge, subject to the status of their respective enable signals.

The subtractor performs by computing the difference between A and B, with the resulting output fed back to register A through the 3:1 MUX. The third input to the MUX originates from B, enabling the seamless interchange of A and B when deemed necessary. The 2:1 MUX selects its inputs from B_in and the output of register A, influenced by the value of B_MUX_sel.

Output signals encompass the GCD, extracted directly from the output of register A, A_lt_B indicating the outcome of the comparison between A and B, and B_eq_0, discerned through comparison with zero.

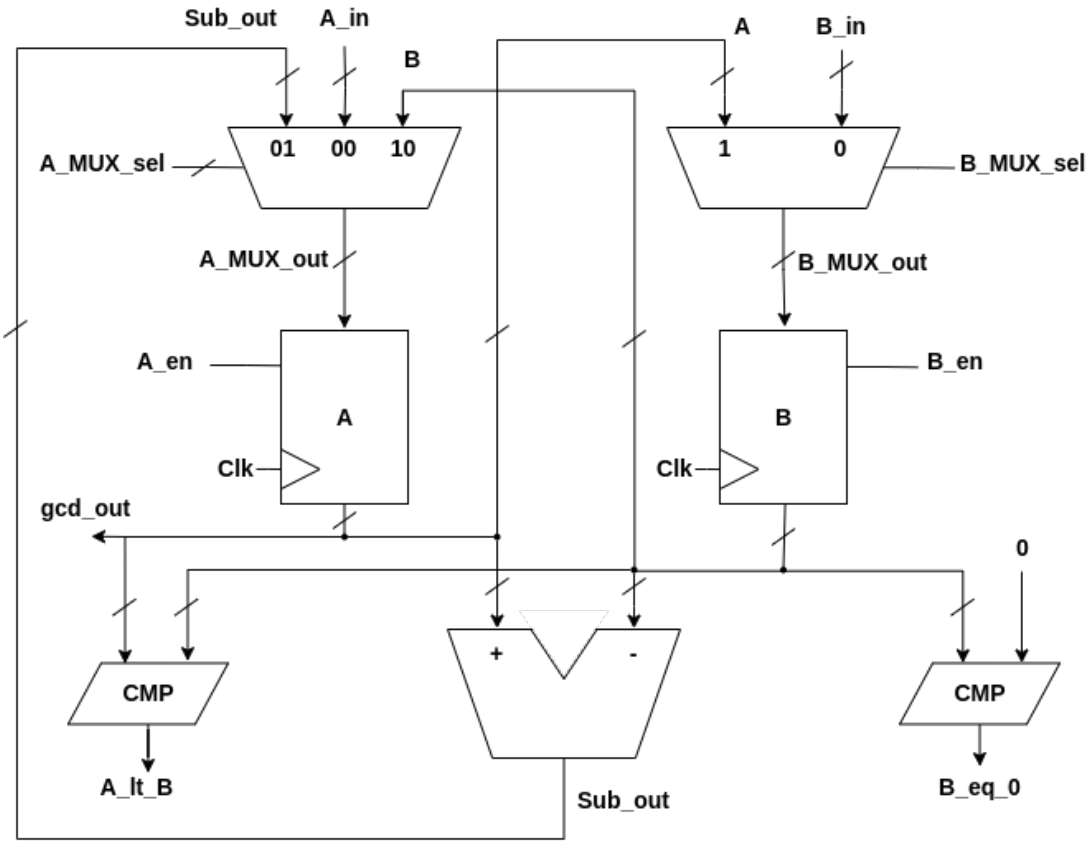

Figure 2: Architecture for the data path

3 FSM Level of Abstraction

To ensure the effective operation of the system, it's imperative to validate both inputs and outputs. This validation can be achieved through qualifiers like "operands_valid" and "gcd_valid". The MUXes integrated into the system necessitate control signals such as "A_ MUX_sel" and "B_ MUX_sel" to function properly. Managing the sequence of inputs and outputs is crucial to ensure orderly completion of each process. A practical strategy involves dividing the system into three states: IDLE, BUSY, and DONE. This segmentation can be facilitated by a three-state Finite State Machine (FSM), which assists in controlling the system's signals and ensures orderly progression through the states. By adopting this methodology, system reliability and accuracy can be upheld.

The system encompasses three states represented in binary digits: IDLE (2'b00), BUSY (2'b01), and DONE (2'b10). In the IDLE state, the system awaits input reception. Transition to the BUSY state occurs once the "operands_valid" signal attains a value of 1.

The BUSY state entails two phases: initially, swapping A and B if A is lesser than B, and subsequently, subtracting B from A. This iterative process persists until B equals 0. During the BUSY state, the system relies on two status signals: "A_lt_B" and "B_eq_0", furnished by the data path. Upon "B_eq_0" achieving a value of 1, the task concludes, and the system transitions to the DONE state.

Within the DONE state, the system awaits an "ack" signal. Upon detection of "ack" being 1, it reverts to the IDLE state. In the event of a reset, the system reverts to its initial state, namely the IDLE state.

The Finite State Machine (FSM) abstraction depicting the three states of the system is illustrated below.

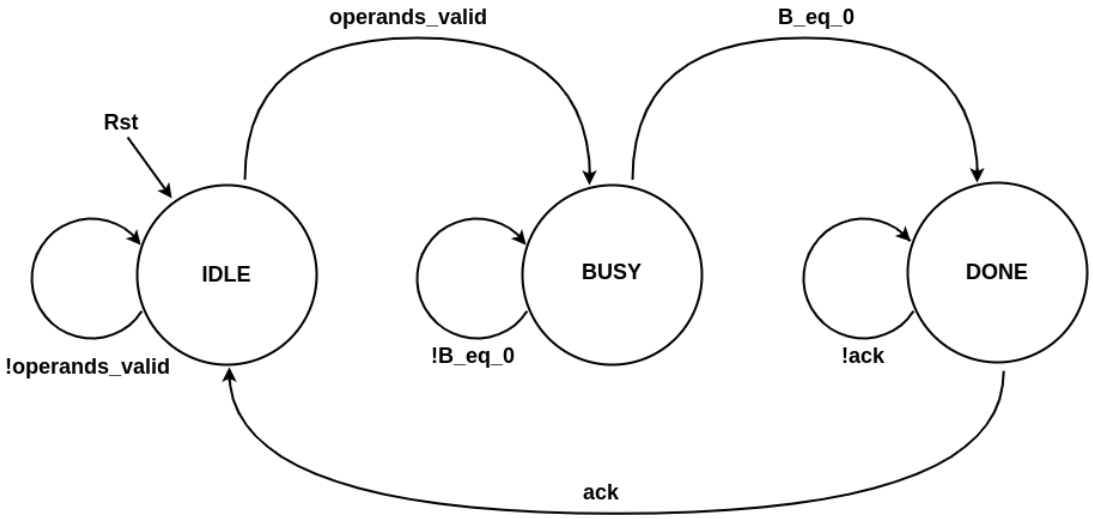

Figure 3: FSM for the states of the system

4 Hierarchical Modeling

The hierarchical model stands as a widely used modeling technique in Verilog, providing several benefits such as flexibility and modularity throughout the design process. By decomposing the design into smaller modules or blocks, we can establish a more structured and manageable system. In this specific design, we employ two modules: the data_path and control_path. The data_path module is tasked with determining the GCD of the two numbers, while the control_path governs the current state of the system. These modules are linked through status signals (B_eq_0, A_lt_B) and control signals (A_ MUX_sel, B_ MUX_sel, A_en, B_en).

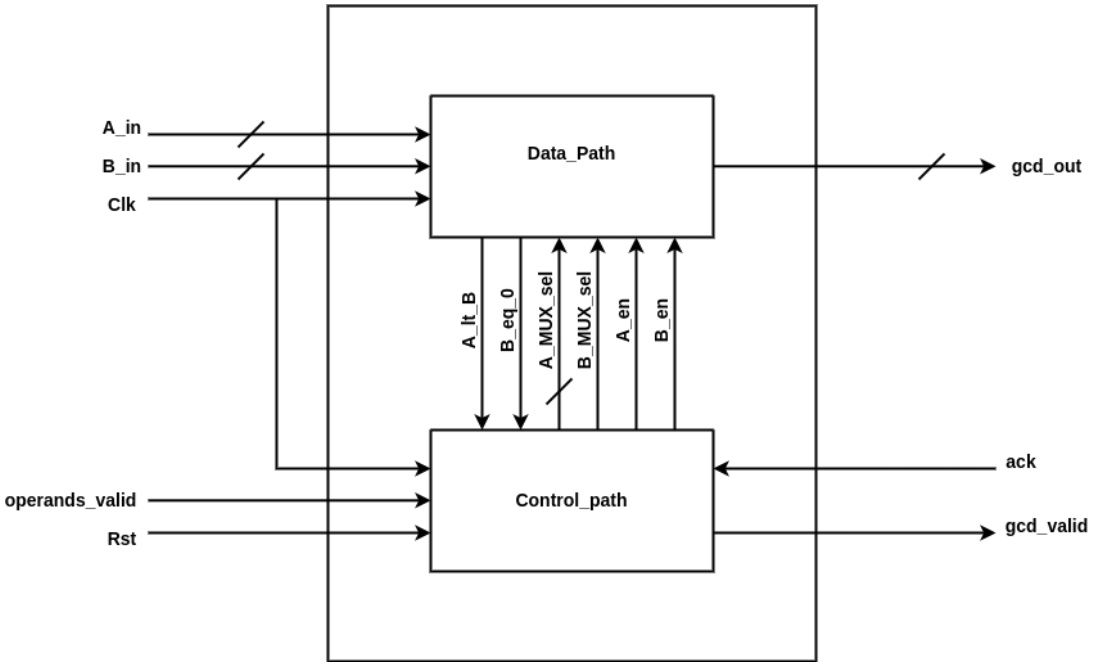

Figure 4: Hierarchical model of the system

Within the data_path module, inputs A_in and B_in are accepted, and the output GCD (A,B) is computed. Conversely, the control_path module receives external inputs operands_valid and ack, alongside status signals A_lt_B and B_eq_0 from the data_path. The control_path module produces gcd_valid and additional control signals directed to the data_path.

By segmenting the system into two modules, we gain the ability to concentrate on specific functionalities and implement changes without impacting the entire design. Furthermore, each module can undergo independent verification and testing, mitigating the risk of errors and enhancing the overall design integrity.

5 Experimental Procedure

Now that the system architecture has been finalized, the subsequent phase involves drafting the requisite Verilog code to execute it. In Verilog, each statement delineates a digital circuit, demanding meticulous attention to ensure accurate modeling of the digital circuit.

For instance, an always block can be utilized to simulate a register, while assign or case statements serve for modeling MUXes. Combinational circuits like subtractors and comparators can also be replicated using assign statements. It is imperative to guarantee that each statement is crafted to execute the intended operation, thereby ensuring logical coherence and operational efficiency of the overall code.

By leveraging the appropriate statements and adhering to best practices, we can formulate Verilog code that faithfully emulates the digital circuit of the system, thereby validating its correct and efficient functionality.

```
1  module top_module(
2        input [7:0] A_in,
3        input [7:0] B_in,
4        input operands_val,
5        input Clk, Rst, ack,
6        output [7:0] gcd_out,
7        output gcd_valid
8  );
9        wire [1:0] A_MUX_sel;
10       wire B_MUX_sel, A_en, B_en, A_lt_B, B_eq_0;
11
12       datapath d1 (
13       .A_in(A_in),
14       .B_in(B_in),
15       .Clk(Clk),
16       .A_MUX_sel(A_MUX_sel),
17       .B_MUX_sel(B_MUX_sel),
18       .A_en(A_en),
19       .B_en(B_en),
20       .gcd_out(gcd_out),
21       .A_lt_B(A_lt_B),
22       .B_eq_0(B_eq_0));
23
24       controlpath c1 (
25       .ack(ack),
26       .Rst(Rst),
27       .Clk(Clk),
28       .operands_val(operands_val),
29       .A_lt_B(A_lt_B),
30       .B_eq_0(B_eq_0),
31       .A_MUX_sel(A_MUX_sel),
32       .B_MUX_sel(B_MUX_sel),
33       .gcd_valid(gcd_valid),
34       .A_en(A_en),
35       .B_en(B_en));
36
37 endmodule
```

Figure 5: Top Module

The top_module encapsulates the entire system, with its inputs and outputs declared within the port.

The inputs comprise A_in, B_in, operands_valid, ack, clk, and reset, while the outputs consist of gcd and gcd_valid. Status and control signals are defined as wire datatype within the module.

Subsequently, the data_path and control_path submodules are instantiated within the top_module as d1 and c1, respectively. Ports are then linked to the nets by name, facilitating communication and data exchange between the modules.

```
41 module datapath(
42       input [7:0] A_in, B_in,
43       input Clk,
44       input [1:0] A_MUX_sel,
45       input B_MUX_sel,A_en, B_en,
46       output [7:0] gcd_out,
47       output A_lt_B, B_eq_0
48 );
49       wire [7:0] A_MUX_out, B_MUX_out;
50       reg [7:0] A,B;
51       wire [7:0] Sub_out;
52
53       always @(posedge Clk)
54             begin
55                   if (A_en)
56                         A <= A_MUX_out;
57                   if (B_en)
58                         B <= B_MUX_out;
59             end
60
61       assign A_MUX_out = A_MUX_sel[1] ? B : (A_MUX_sel[0] ? Sub_out : A_in);
62       assign B_MUX_out = B_MUX_sel ? A : B_in;
63       assign Sub_out = A - B ;
64       assign A_lt_B = ( A<B );
65       assign B_eq_0 = (B==0);
66       assign gcd_out = A;
67
68 endmodule
```

Figure 6: DataPath Module

The data_path module encompasses the structural-level code of the system. Following the declaration of inputs and outputs, the designer proceeds to define additional components necessary for computing the sum.

The outputs from the two MUXes are declared as wire type, with names A_ MUX_out and B_ MUX_out. Depending on the value of the 2-bit MUX select signals, A_ MUX_out and B_ MUX_out alter. These alterations are implemented using a conditional operator. Subsequently, these wires are linked to the registers A and B, respectively.

During each positive edge of the clock, the registers adopt the value of these wires. This process can be modeled through a non-blocking assignment within an always block with posedge clk enlisted in its sensitivity list.

The subtractor within the architecture can be represented using a simple assign statement. The output from the subtractor, declared as a wire datatype (wire [31:0] diff_out), subtracts B from A. This output is then directed to the MUX associated with register A. Similarly, the comparator, which contrasts the values of A and B, yields the status signal A_lt_B. Additionally, the comparator that juxtaposes B with zero and produces B_eq_0 can be modeled using an assign statement.

```
74 module controlpath(
75       input ack, Rst, Clk, operands_val, A_lt_B, B_eq_0,
76       output reg [1:0] A_MUX_sel,
77       output reg B_MUX_sel,
78       output gcd_valid,
79       output reg A_en, B_en
80 );
81
82       reg [1:0] state ;
83       reg [1:0] state_next;
84
85 //parameters to define the states
86
87       parameter idle = 2'b00;
88       parameter busy = 2'b01;
89       parameter done = 2'b10;
90
91 //finding current state
92
93 always @(posedge Clk or posedge ack)
94       begin
95             if (Rst)
96                   state <= idle;
97             else
98                   state <= state_next;
99       end
100
101 //combinational logic to find next state
102
103 always @(*)
104       case(state)
105             idle: begin
106                   A_MUX_sel = 2'b00;
107                   B_MUX_sel=1'b0;
108                   A_en=1'b1;
109                   B_en=1'b1;
110                   if (operands_val ==1) state_next = busy;
111                   else state_next = idle;
112             end
113
114             busy: begin
115                   if (A_lt_B) begin
116                         A_MUX_sel=2'b10;
117                         B_MUX_sel=1'b1;
118                         A_en = 1'b1;
119                         B_en=1'b1;
120             end
121
122                   else begin
123                         A_MUX_sel = 2'b01;
124                         B_MUX_sel=1'b0;
125                         A_en=1'b1;
126                         B_en=1'b0;
127       end
128       if (B_eq_0 == 1) state_next =done;
129       else state_next = busy;
130       end
131
132             done: begin
133                   A_en=0;
134                   B_en=0;
135
136                   if (ack==1) state_next=idle;SS
137                   else state_next = done;
138
139                   end
140       endcase
141
142 //assigning the output
143
144 assign gcd_valid = (state == done);
145
146 endmodule
```

Control Path Module

The control_path module can be constructed using FSM level abstraction code. Within an FSM level abstraction, sequential logic is utilized to determine the current state of the system, a blend of combinational and sequential logic is employed to ascertain the subsequent state of the system, and finally, combinational logic is used to derive the system's output.

Two registers, namely state and state_next, are utilized to retain the current and next states of the system. Additionally, local parameters idle (2'b00), busy (2'b01), and done (2'b10) are declared to denote the three distinct states of the system.

An always block triggers at the positive edge of the clock to ascertain the current state of the system. If the reset signal is activated, the current state transitions to idle; otherwise, the state should adopt the value of state_next. A case statement, incorporating the variable state within an always @ (*) block, is employed to determine the subsequent state of the system. Concurrently, appropriate values for the control signals must be assigned.

During the IDLE state, A_ MUX_sel and B_ MUX_sel assume values of 2'b00 and 1'b0, respectively, facilitating the selection of initial inputs A_in and B_in. Furthermore, A_en and B_en should be assigned 1'b1 to enable the registers to accept new inputs.

Within the BUSY state, swapping and subtraction operations occur. If A_lt_B equals 1, A and B are swapped to ensure A always represents the greater number. Criss-cross connections are established to feed the swapped numbers into the registers. Accordingly, A_ MUX_sel is set to 2'b10 and B_ MUX_sel to 1'b1. Again, A_en and B_en should be set to 1 to accept these new inputs. If A is greater than B, no swapping is required, and subtraction occurs directly. Thus, A_ MUX_sel is set to 2'b01, selecting A-B as the input to register A while retaining the previous value of B. Consequently, A_en is set to 1'b1 and B_en to 1'b0.

In the DONE state, the system completes its calculations and cannot accept new inputs until the existing output is forwarded to the next module. Hence, A_en and B_en should be set to zero. These assignments can be encapsulated within the aforementioned case statement used to determine the next state.

Once the current state and next state are established, the output from the control path, gcd_valid, can be determined using a simple assignment statement. gcd_valid is assigned 1'b1 if the state is equivalent to DONE.

6 Results

In this section, we delve into a thorough evaluation of our design's performance by constructing a comprehensive test bench in Verilog. The primary aim of this test bench is to provide a varied set of inputs, known as stimuli, to our design. Subsequently, we analyze the resulting outputs meticulously to assess the effectiveness and reliability of our system.

Employing a test bench serves as a crucial step in validating our design's functionality and ensuring its alignment with the specified requirements. Through careful planning and thoughtful selection of stimuli, we conduct a comprehensive testing process to validate the integrity and robustness of our system's operation.

```
module gcd_test;
  reg [7:0] A_in, B_in, A_gen, B_gen;
reg clk,reset, operands_val, ack_rcvd;
  wire [7:0] GCD, out_behav;
wire gcd_valid;

integer delay,i;

top_module gcd1(
.A_in(A_in),
.B_in(B_in),
.operands_val(operands_val),
.Clk(clk),
.Rst(reset),
.ack(ack_rcvd),
 .gcd_out(GCD),
.gcd_valid(gcd_valid)
);

gcd_behav gcd_behav1(.inA(A_in), .inB(B_in), .Y(out_behav));

always
#10 clk = ~clk;

initial begin
$dumpfile("dump2.vcd");
$dumpvars(0);
end

initial begin
drive_reset;

  for (i=0;i<5;i=i+1)
begin
A_gen = 14;
B_gen = 42;
drive_input(A_gen, B_gen);
check_output;
end

repeat(20)@(negedge clk)
$finish;
end

task drive_reset;
  begin
  $display ("Driving the reset");
clk = 1'b0;
@ (negedge clk)
reset = 0;
@ (negedge clk)
reset = 1;
@ (negedge clk)
reset = 0;
  end
endtask

  task drive_input(input [7:0] A_gen, B_gen);
  begin
  $display ("Driving the Input");
@ (negedge clk)
operands_val = 1;
A_in = A_gen;

B_in= B_gen;
@ (negedge clk)
operands_val = 0;
  end
endtask

task check_output;
  begin
@ (posedge gcd_valid)
    $display ("Recieved GCD Valid");S
  if(GCD == out_behav)
    $display ("Test Succeeded");
else
  $display("Test failed");
delay = 10;
repeat(delay)@(negedge clk)
```

```
module gcd_test;
  reg [7:0] A_in, B_in, A_gen, B_gen;
reg clk,reset, operands_val, ack_rcvd;
  wire [7:0] GCD, out_behav;
wire gcd_valid;

integer delay,i;

top_module gcd1(
.A_in(A_in),
.B_in(B_in),
.operands_val(operands_val),
.Clk(clk),
.Rst(reset),
.ack(ack_rcvd),
 .gcd_out(GCD),
.gcd_valid(gcd_valid)
);

gcd_behav gcd_behav1(.inA(A_in), .inB(B_in), .Y(out_behav));

always
#10 clk = ~clk;

initial begin
$dumpfile("dump2.vcd");
$dumpvars(0);
end

initial begin
drive_reset;

  for (i=0;i<5;i=i+1)
begin
A_gen = 14;
B_gen = 42;
drive_input(A_gen, B_gen);
check_output;
end

```

Test Bench Code

We begin by initializing the test bench, specifying inputs as reg types and outputs as wire types. Subsequently, we instantiate the top_module and establish connections between its ports and the net by name. To generate a clock cycle lasting 10 seconds, we utilize an always block. Within the initial-begin block, we assign initial values and furnish stimuli necessary for testing our system. Initially, the clock and reset signals are set to zero and one, respectively. Subsequently, reset transitions to zero. Additionally, we set operand_valid to one to enable the system to accept inputs A_in = 14 and B_in = 42. Upon receiving input, the system initializes the GCD calculation. After a specified duration, we assert the ack signal. Finally, we utilize $finish to conclude the simulation.

Expanding on the above, the test bench serves as a critical component in our design verification process. It ensures that our system operates as intended and meets the prescribed specifications. By meticulously setting up the test bench and providing appropriate stimuli, we can thoroughly assess the functionality and performance of our design. This rigorous testing process allows us to identify any potential issues or discrepancies and address them before deployment, ensuring the reliability and accuracy of our system.

The digital system initiates in the idle state with the clock at zero and reset set to one. Initially, the operand_valid signal remains inactive, holding the system in the idle state. After 10 seconds, operand_valid becomes active, allowing the system to accept inputs A_ = 14 and B_in = 42. In the idle state, the MUX select signals are set to zero, directing values from the MUX outputs to the registers' inputs. Upon the second clock edge, the system transitions from idle to busy state, with A and B registers assuming values 0x0E and 0x2A, respectively. As B exceeds A, A_lt_B = 1 and B_eq_0 = 0. Thus, B is fed directly to MUX A and A is fed into MUX B (They both are Swapped). During the busy state, A_ MUX_sel = 2'b10 and B_MUX_sel = 1'b1 , resulting in A_MUX_out = 42 and B_MUX_out = 14. Subsequently, these values are transferred to A and B registers at the third clock edge.

Figure 7: Output Waveforms

Consequently, the system adjusts Sub_out to 28 , with A_MUX_sel = 2'b01 and B_MUX_sel = 1'b1, yielding A_MUX_out = 0x1C and B_MUX_out = 0x0E. This cycle continues until B values equalize to zero. Upon reaching this state, the system transitions to the done state, with A_en and B_en signals set to zero, indicating cessation of input acceptance by the registers. The gcd_valid signal is set to one, with gcd assigned the value of A, which is 0x0E (14). Finally, ack becomes high, leading gcd_valid to become zero at the subsequent clock edge, while gcd remains unchanged for one additional clock cycle.

7 Conclusion

In conclusion, our experiment successfully implemented a Verilog-based digital system that effectively finds the greatest common divisor (GCD) of two positive integers. We thoroughly considered various scenarios, including the selection of Euclid's algorithm for finding GCD, hierarchical modelling to divide the system into separate control and data paths, and hardware minimization using a subtractor instead of a divider or modulus operator. By optimizing our design, we were able to create an efficient and easy-to-implement system. The hierarchical modelling technique we utilized was particularly effective in improving the design quality and ease of use. Breaking down the system into smaller, more manageable modules allowed for comprehensive testing and the correction of errors before the modules were combined within the top module.