EE5516 VLSI Architectures for Signal Processing and Machine Learning

Lab Report - Experiment No.2

Nakul C - 122101024

February 20, 2024

Abstract

The objective of this experiment is to gain familiarity with Verilog implementation of digital circuits and to demonstrate how to map an algorithm to an architecture. Specifically, the experiment aims to calculate the sum of natural numbers up to a given limit. This is accomplished by implementing a hierarchical design, which divides the entire block into two modules: the control path and the data path. These two modules are connected by control and status signals. The data path is designed using a structural approach, while the control path uses an FSM level of abstraction. The ultimate objective is to achieve the desired result using the minimum number of hardware resources, which may require modifying the native algorithm, to optimize it for hardware implementation

1 Introduction

For the given experiment we are required to find the sum of first 'N' natural numbers. While the problem may seem simpler in the software level, its hardware implementation requires systematic approach for optimal performance and minimum use of hardware resources. We have to consider various scenarios such as choice of algorithm, hierarchical design and hardware resource optimization.

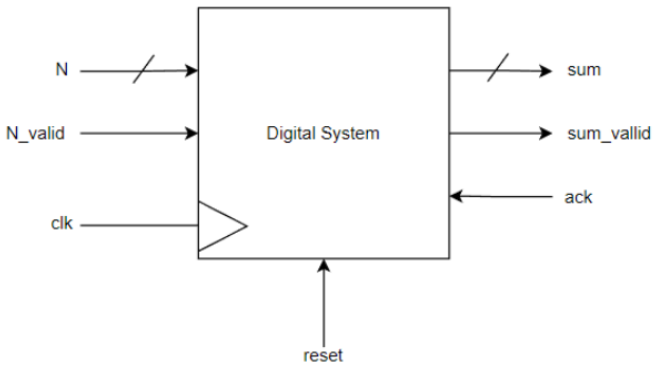

Figure 1: Block Diagram

The goal of our digital system is to accept an input 'N' and produce the sum of the first N natural numbers as the output. To ensure reliable operation, we incorporate qualifier signals 'N_valid' and 'sum_valid' that indicates when the input and output values are valid. Additionally, we utilize clock and reset signals to synchronize and reset the system, respectively. Since our module may be part of a larger

system, it is crucial to implement a proper acknowledgment protocol to ensure successful communication between modules. This protocol allows us to confirm receipt of a message before sending a response to the other module. A block diagram displaying all inputs and outputs of the digital system is presented above to provide a visual representation of the system's design.

2 Implementation with FSM

To implement our digital system, we require several components, including 32-bit registers for storing the input (N) and output (sum), a counter (i), an adder, and a comparator that determines the termination condition. Choosing the correct type of counter is crucial for optimizing our design. While up-counters and down-counters are two common types, using an up-counter would necessitate three registers and MUXes, which goes against our goal of minimizing hardware resources. Consequently, we opt for a down-counter instead, which eliminates the need to store the value of N and reduces our hardware requirements from three registers and MUXes to two of each. This exemplifies how modifying the native algorithm can lead to a more suitable hardware implementation. To ensure proper system operation, we use a 3-state FSM to manage the sequencing of inputs and outputs, allowing each process to complete before the next one begins. Critical path time delay (Tpd) is an important consideration in digital systems, with Tpd in our case being equal to the sum of the time delays incurred by the adder and comparator. To minimize the impact of these delays on the system's results, we always obtain the output directly from the registers.

2.1 Structural Design

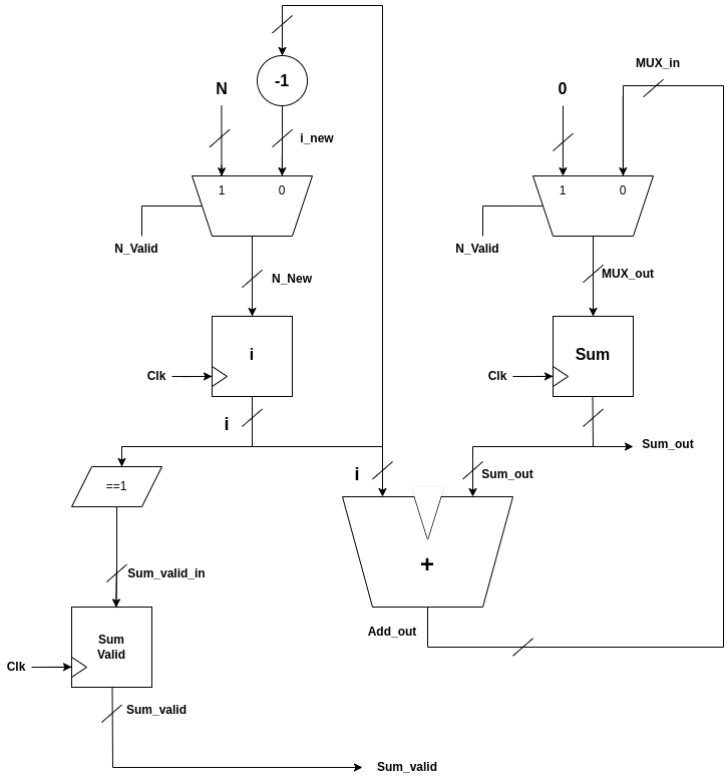

Figure 2: Architecture for data path

To implement our digital system, we have chosen a 32-bit down counter that counts down from N to 1. The counter is realized using a register and a 3:1 MUX, where the MUX switches its output based on the 2-bit control signal. Initially, the system is in the IDLE state where the MUX output is N, which is fed into the register i at the next positive clock edge, marking the start of the process. The state then changes to BUSY and the counter starts counting down from N. The value of i is decremented and fed back to the MUX through a unit decreamenter, while a comparator continuously compares the value of

i to 1. When the value of i is equal to 1, the process is completed and the status signal i_eq_1 is asserted. The system then transitions to the DONE state, where we have to freeze the output and wait for the ack signal. This is achieved by feeding back the value of I directly to the register. Another register, called sum, is used to store the sum in each stage, with its initial value being zero. The value of sum changes as i counts down from N to 1, and the changes are properly stored in sum. A 3:1 MUX is used to get the values of sum during different states of the system: during IDLE state, sum is equal to 0; during BUSY state, the value of sum is the sum of i and sum, i.e., sum = sum + i, and this is continuously calculated using an adder before being fed to the MUX; during DONE state, sum is equal to sum itself.

2.2 FSM Level of Abstraction

To ensure proper operation of the system, it is essential to validate both the inputs and outputs. Qualifiers such as N_valid and sum_valid can be used to achieve this. N_valid determines when the input can be accepted by the system, while sum_valid indicates that the final sum has been calculated. Additionally, it is important to manage the sequencing of inputs and outputs to allow each process to complete before the next one begins. This can be achieved by dividing the system into three states: IDLE, BUSY, and DONE.

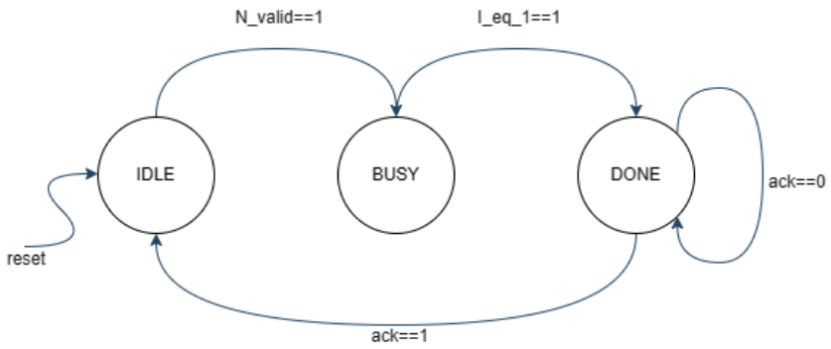

Figure 3: FSM for the states of the system

2.3 Hierarchical Modeling

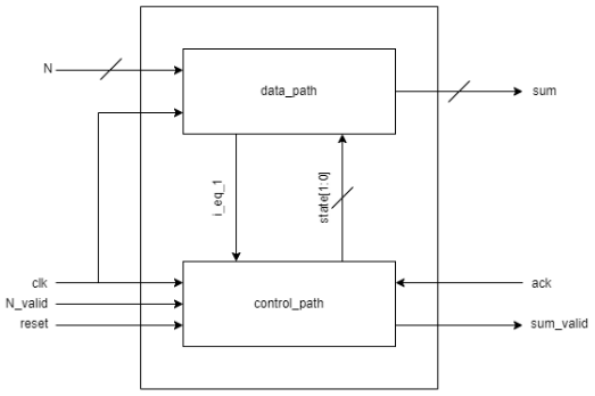

Figure 4: Hierarchical model of the system

The hierarchical model is a popular modeling technique in Verilog that offers numerous advantages, including flexibility and modularity in the design process. By breaking down the design into smaller

modules or blocks, the designer can create a more organized and manageable system. In this particular design, we have two modules: the data_path and control_path. The data_path is responsible for finding the sum, while the control_path decides the current state of the system. These two modules are connected by status signal (i_eq_i) and control signal (state).

2.4 Experimental Procedure

Now that the system architecture has been finalized, the next step is to write the necessary Verilog code to implement it. In Verilog, each statement describes a digital circuit, and it is essential to ensure that our statements accurately model the digital circuit. For instance, we can use an always block to model a register, while assign or case statements can be used to model MUXes. Combinational circuits such as adders and comparators can also be modeled using assign statements. It is crucial to ensure that each statement is designed to perform the desired operation, and the overall code is logically sound and efficient. By using the appropriate statements and following best practices, we can create Verilog code that accurately models the system's digital circuit, ensuring that it functions correctly and efficiently.

```
// This is the Implementation to find the sum of N Natural Numbers with FSM

module Datapath(N,N_valid,Sum,Sum_valid_out,Clk,Rst); // Defining DataPath Module
  input [7:0] N; // Defining Inputs and Outputs
  input N_valid , Clk, Rst;
  reg [17:0] Sum_out;
  output [17:0] Sum;
  output Sum_valid_out;
  reg Sum_valid ;
  wire[17:0] MUX_out, Add_out;
  reg[7:0] i;
  wire[7:0] i_new, N_new;
  wire Sum_valid_in, i_eq_1, Rst;
  wire [1:0] State_out;
  wire [1:0] MUX_sel;

  // Calling the Control Path inside the Datapath Module
  FSM F0(State_out, N_valid, i_eq_1, Sum_valid_out, Clk, Rst);

  assign i_eq_1 = (i == 1);
  assign Sum = Sum_out;
  assign MUX_out = MUX_sel ? 18'b0 : Add_out ;
  assign MUX_sel = (State_out == 2'b00);
  always @ (posedge Clk)
    begin
    if(Rst)
    begin
    Sum_out <=  0 ;
    i <= 0 ;
    end
    else
    begin
        Sum_out <= MUX_out ;
    i <= N_new ;
        end
    end
  assign Add_out = i + Sum_out;
  assign i_new = i - 1;
  assign N_new = MUX_sel ? N : i_new ;
  assign Sum_valid_in = (i == 8'b1);
  assign Sum_valid_out = (State_out==2'b10);
endmodule

module FSM(State_out,N_valid, i_eq_1, Sum_valid, Clk, Rst); //Defiinition of ControlPath Module
input N_valid, i_eq_1, Clk, Rst; // Defining Inputs and Outputs
output Sum_valid;
output [1:0] State_out;
reg [1:0] State;
reg [1:0] State_next;

parameter Idle = 2'b00; //Defining States
parameter Busy = 2'b01;
parameter Done = 2'b10;


always@(posedge Clk)   //Idle = 0 , Busy = 1 , Done = 2
    if(Rst) State <= Idle;
    else State <= State_next;

always@(*)
case(State)
 Idle: if (N_valid) State_next = Busy; else State_next = Idle;
 Busy: if (i_eq_1) State_next = Done; else State_next = Busy;
 Done: State_next = Idle;
endcase

assign State_out=State;

endmodule
```

Figure 5: Verilog Code for finding the sum of N Natural Numbers using FSM

2.5 Results

In our test bench, we provided $N = 3$, and accordingly, we expected the output to be 6. As anticipated, we obtained the output from `Sum_out` when `sum_valid` was high, yielding a result of 6.

```
// Testbench for Implementation of Sum of Natural Numbers with FSM.

module Testbench;
  reg [7:0] N;
  reg N_valid, Clk, Rst;
  wire [17:0] Sum;
  wire Sum_valid_out;

  Datapath DUT(N,N_valid,Sum,Sum_valid_out,Clk, Rst);

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, DUT);

    Clk = 0;
    forever #10 Clk = ~Clk;
  end

  initial begin
    Rst = 0;
    #10
    N_valid = 1;
    N = 3; // Given N = 3 Hence the expected output is 6.
    #15
    Rst = 1;
    #10
    Rst = 0;
    #10
    N_valid = 1;
    #20
    N_valid = 0;
  end

  initial begin
    $monitor($time);
    #500
    $finish;
  end
endmodule


```

Figure 6: Testbench for finding the sum of N Natural Numbers using FSM

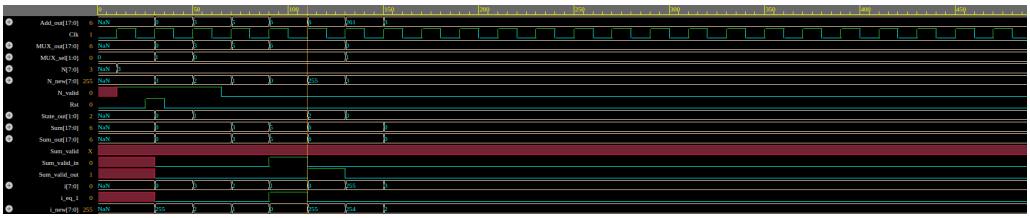

Figure 7: Timing Diagram for finding the sum of N Natural Numbers using FSM

3 Implementation without FSM

In this subsection, we'll discuss an implementation approach that does not involve using a Finite State Machine (FSM). The design focuses on a more direct approach to achieving the desired functionality without the overhead of FSM-based control.

3.1 Structural Design

We'll discuss a structural design approach where the control of the system and the computation of the sum are closely integrated without the explicit use of an FSM. This approach aims to streamline the design while ensuring efficient resource utilization.

```
// This is the Implementation to find Sum of N Natural Numbers without FSM

module Datapath N,N_valid,Sum_out,Sum_valid,Clk ; // Defining Inputs and Outputs
  input [7:0] N;
  input N_valid , Clk;
  output reg [17:0] Sum_out;
  output reg Sum_valid ;
  wire [17:0] MUX_out, Add_out;
  reg [7:0] i;
  wire [7:0] i_new, N_new;
  wire Sum_valid_in;

  assign MUX_out = N_valid ? 18'b0 : Add_out ; // MUX_A

  always@ posedge Clk // Registar_A
    begin
        Sum_out <= MUX_out ;
    end

  assign Add_out = i + Sum_out; // Adder_A

  assign i_new = i - 1; // Decrementer_A

  assign N_new = N_valid ? N : i_new ; // MUX_B

  always@ posedge Clk // Register_B
    begin
        i <= N_new ;
    end

  assign Sum_valid_in = i == 8'b1 ; // Comparator_A

  always@ posedge Clk // Register_C
    begin
        Sum_valid <= Sum_valid_in ;
    end

endmodule
```

Figure 8: Code for finding the sum of N Natural Numbers without using FSM

3.2 Experimental Results

In this section, we test the functionality of our design by creating a suitable test bench in Verilog. The purpose of the test bench is to provide random inputs (stimuli) to the design and check the corresponding outputs. We use the test bench to input values to the top_module and obtain the results in the form of output waveforms.

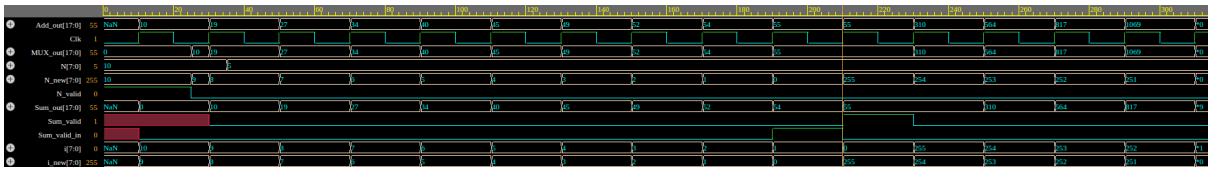

Figure 9: Timing Diagram for finding the sum of N Natural Numbers without using FSM

In our test bench, we provided N = 10, and accordingly, we expected the output to be 55. As anticipated, we obtained the output from Sum out when sum valid was high, yielding a result of 55. But after the expected output we observed junk values in Sum out because of the absence of a Control Path.

```
// Test Bench for the Sum of Natural Numbers without FSM

module tb;
    reg [7:0] N; // input max size 1 byte
    reg N_valid;  //Input Qualifier
    reg Clk;
    wire [17:0] Sum_out; // output size 2 byte since input is 1 byte
    wire Sum_valid; //Output Qualifier

    Datapath DUT N,N_valid,Sum_out,Sum_valid,Clk ;
    initial
    begin
        $dumpfile "dump1.vcd" ;
        $dumpvars;
        N = 10; // Given N = 10 Hence Expected output is 55
        N_valid = 1;
        Clk = 1'b0;
        #25 N_valid = 0;
        #10 N = 5;
        #10000 $finish;
    end
    always #10 Clk = ~Clk;

endmodule

```

Figure 10: Testbench for finding the sum of N Natural Numbers without using FSM

4 Conclusion

In this experiment, we successfully implemented a Verilog-based digital system that calculates the sum of the first N natural numbers and verified its functionality using a test bench. This provided us with valuable insights into the implementation of digital circuits using Verilog. By exploring various scenarios, such as the choice of algorithm, hierarchical modeling, and hardware minimization, we were able to optimize the design and reduce hardware resources. The hierarchical modeling of the Verilog code proved to be a powerful technique in the implementation of our system, allowing us to break down the design into smaller, more manageable modules, which improved the overall design quality and ease of use. Moreover, we were able to demonstrate how the hierarchical model offers designers unparalleled flexibility and modularity, enabling faster development times and lower costs. In conclusion, this experiment provided us with a comprehensive understanding of Verilog and its application in digital circuit design. We learned that the hierarchical model is an essential tool that enables the design of complex systems while improving their quality, ease of use, and development speed.