EE5516 VLSI Architectures for Signal Processing and Machine Learning

Lab Report - Assignment 1

Nakul C - 122101024

March 13, 2024

Abstract

This experiment seeks to compute the sum of squares of the first N natural numbers. It employs a hierarchical modeling approach by dividing the problem into two modules: the control path and the data path. The data path is constructed using a structural method, assembling the module from smaller, basic components. Meanwhile, the control path utilizes a finite state machine (FSM) abstraction to oversee the algorithm's overall flow. The objective is to attain an efficient and optimized implementation of the sum of squares algorithm while conserving hardware resources to the fullest extent possible.

1 Introduction

The aim of this experiment is to compute the sum of squares of the first N natural numbers. The sum of squares formula can be expressed as the sum of repeated additions of each natural number up to N. To achieve this, we'll design both a VLSI data path and control path using Verilog. Additionally, we'll create a testbench to validate the correctness of our design.

Our digital system can be depicted as a block that takes N as input and produces the sum of squares of the first N numbers. To ensure the integrity of both input and output, we'll incorporate qualifiers such as "N_valid" and "sum_valid." Furthermore, we'll integrate a clock and reset signal to synchronize and reset the system, respectively. To enable efficient communication between modules within the system, we'll employ an acknowledgment (ack) protocol.

2 Implementation

To implement this system, we will utilize hardware components such as MUXes, registers, comparators, and adders. The pseudo-code for the given problem is as follows:

```
input N;
sum = 0;
for i=1 to N
begin
   i_acc = 0;
   for j=1 to i
   begin
     i_acc = i_acc+ i;
   end
   sum = sum + i_acc;
end
output sum
```

The algorithm consists of a nested loop structure with an outer loop controlled by the loop variable i and an inner loop controlled by the variable j. The inner loop calculates the individual components like $(1), (2+2), (3+3+3)$, and so on, while the outer loop iteratively identifies the number i and calculates the sum of all the individual components computed by the inner loop.

3 Structural Design

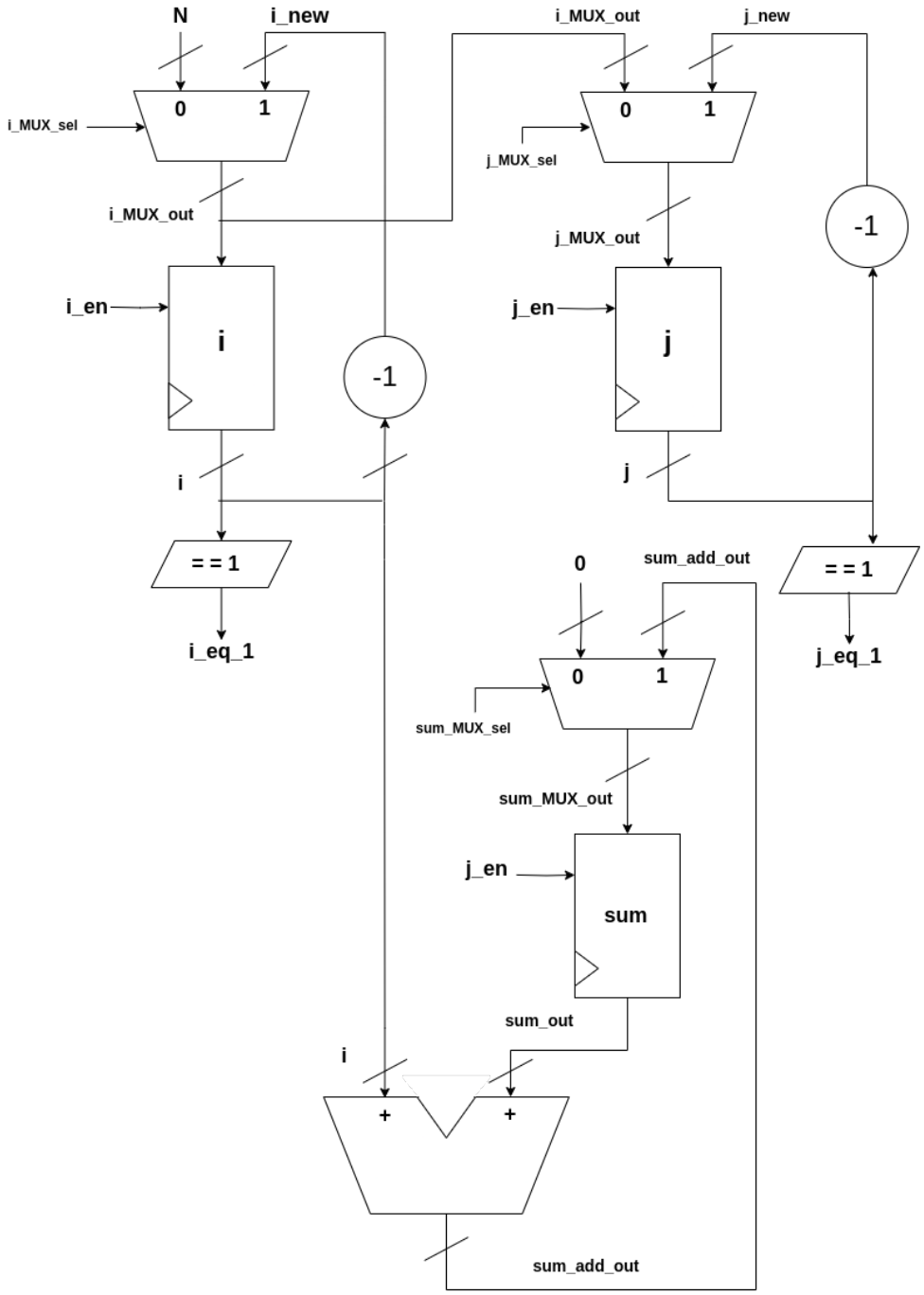

Figure 1: Data path Architecture

The architecture comprises three registers for storing the values of "i", "j" and "sum" respectively. It also contains three 2:1 MUXes, These MUXes select inputs to the registers at each stage based on control signals i_MUX_sel, j_MUX_sel and sum_MUX_sel

The main components necessary for implementing the data path are:

- **Down Counter**: A counter is needed to generate the values of i in the outer loop and j in the inner loop It should initialize to N at the program's start and decrement by 1 at the end of each outer loop iteration. The counter should stop when it reaches the value of 1. op iteration and stop at the inner loop's end.
- **Accumulator**: An accumulator is required to calculate the value of i_acc in each inner loop iteration. It should initialize to 0 at the inner loop's start and add the value of i to its current value in each iteration.
- **Input**: An input is needed to provide the value of N to the data path.
- **Output**: An output is necessary to display the final value of sum at the program's end.

4 FSM Level of Abstraction

The Finite State Machine (FSM) level abstraction provides an efficient means to represent the states of a system. In this particular system, there exist three distinct states: IDLE, BUSY and DONE. Initially, when the system has no inputs or when the reset signal is asserted, it resides in the IDLE state. Upon the Qualifier signal N_valid transitioning to one, the system becomes capable of accepting input and transitions to the BUSY state.

Within the BUSY state, the inner loop variable j iterates downwards from i to 1, with i initially set to N. Once j reaches one, i is decremented by one, starting from $(N-1)$ down to 1. If i is not equal to 1, the system returns to the BUSY1 state to continue with the inner loop. These transitions effectively manage the execution of both the inner and outer loops.

Upon completion of all loops and when i becomes 1, the process concludes, and the system transitions to the DONE state. Upon receiving an acknowledgement signal, the system reverts to the IDLE state.

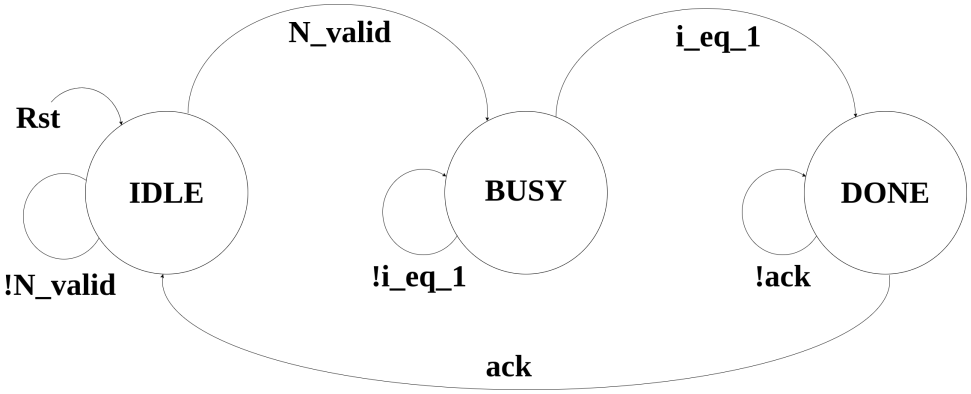

Figure 2: FSM level abstraction of the system states

5 Hierarchical Modeling

The hierarchical model is a popular modeling technique in Verilog, offering various advantages including flexibility and modularity in the design process. By decomposing the design into smaller modules or blocks, we can establish a more organized and manageable system. In this specific design, we have two modules: the data_path and control_path. The data_path module is responsible for computing the sum of squares, while the control_path module determines the current system state and controls the loop implementation. These modules are interconnected via status signals (e.g., j_eq_1, i_eq_1) and control signals (e.g., i_mux_sel, j_mux_sel, sum_mux_sel).

The data_path module accepts input N and calculates the sum of squares of the first N numbers. Conversely, the control_path module receives inputs N_valid and ack from outside the system, along with status signals i_eq_1 and j_eq_1 from the data_path. The control_path outputs sum_valid and other control signals to the data_path. By partitioning the system into two modules, we can easily focus on specific functionality and make changes without affecting the entire design. Furthermore, each module can be independently verified and tested, reducing the risk of errors and enhancing the overall design

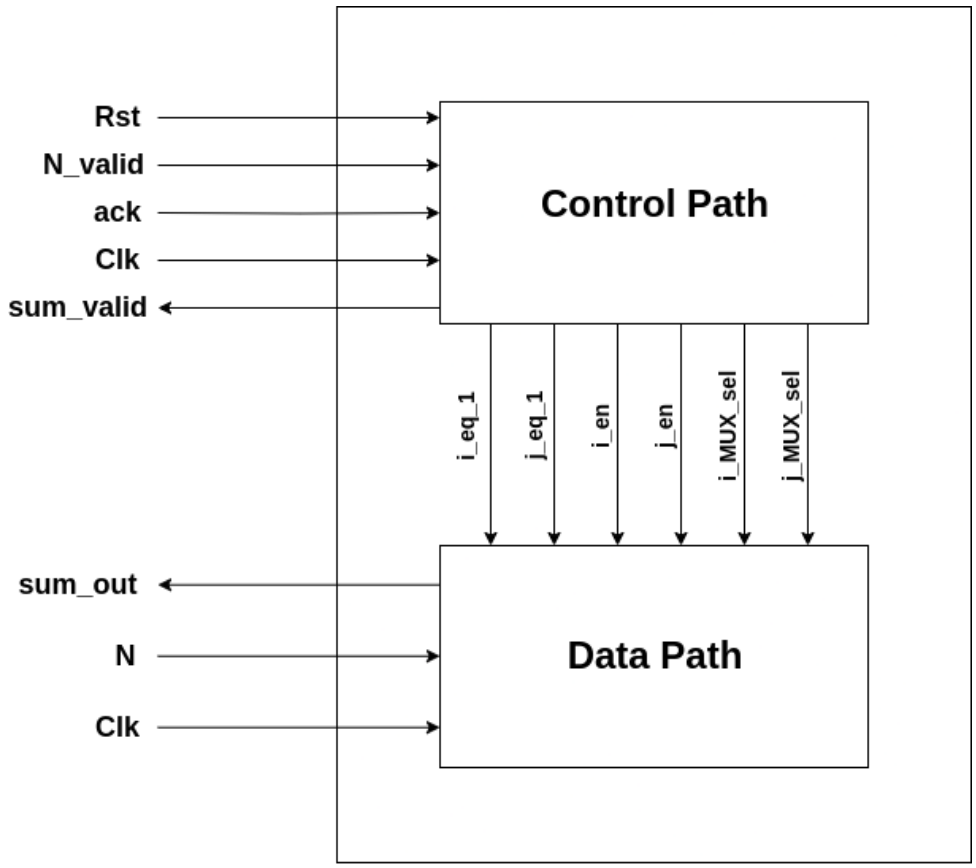

Figure 3: Hierarchical Model

quality.

The top_module represents the complete system, with its inputs and outputs declared inside the port. Inputs include N, N_valid, ack, clk, and reset, while outputs consist of sum and sum_valid. The status and control signals are declared as wire data types within the module. The Datapath and control_path submodules are instantiated inside the top_module as datapath_inst and fsm_inst , respectively. Ports are then connected to nets by name.

The data_path module contains the structural-level code of the system. After declaring the inputs and outputs, the designer needs to define additional components required for computing the sum. The output from the three MUXes is declared as reg data type. Depending on the value of the 2-bit MUX select signals, the values of i_MUX_out and sum_MUX_out change. These changes are implemented using a case statement inside an always block. These regs are then connected to the registers i, sum, j At each positive edge of the clock, the registers take the value of these MUX outputs, which can be modeled using a non-blocking assignment inside an always block with posedge clk in its sensitivity list.

There is one accumulators for computing the sum of squares. These are designed using assign statements. Comparators which continuously compare the values of i and j with 1 can also be designed using assign statements.

The control_path module can be implemented using FSM level abstraction code. Inside an FSM level abstraction, sequential logic determines the current state of the system, a combination of combinatorial and sequential logic finds the next state, and combinatorial logic generates the system output. Two registers, state and state_next, are used to store the current and next states of the system. Local parameters such as idle, busy and done are declared to represent the different states of the system. An always block triggers at the positive edge of the clock to determine the

```
module top_module(
  input [2:0] N,
  input Clk, Rst, ack, N_valid,
  output [7:0] sum_out,
  output sum_valid
);
  wire j_MUX_sel, i_MUX_sel, sum_MUX_sel, j_en, i_en, sum_en, i_eq_1, j_eq_1;

  Datapath datapath_inst(
    .N(N),
    .Clk(Clk),
    .i_en(i_en),
    .j_en(j_en),
    .i_MUX_sel(i_MUX_sel),
    .j_MUX_sel(j_MUX_sel),
    .sum_en(sum_en),
    .sum_MUX_sel(sum_MUX_sel),
    .sum_out(sum_out),
    .i_eq_1(i_eq_1),
    .j_eq_1(j_eq_1)
  );

  FSM fsm_inst(
    .ack(ack),
    .Rst(Rst),
    .Clk(Clk),
    .N_valid(N_valid),
    .i_eq_1(i_eq_1),
    .j_eq_1(j_eq_1),
    .i_en(i_en),
    .j_en(j_en),
    .i_MUX_sel(i_MUX_sel),
    .j_MUX_sel(j_MUX_sel),
    .sum_MUX_sel(sum_MUX_sel),
    .sum_en(sum_en),
    .sum_valid(sum_valid)
  );

endmodule


// Datapath
module Datapath(
  input [2:0] N,
  input Clk,
  input i_en, j_en, i_MUX_sel, j_MUX_sel, sum_en, sum_MUX_sel,
  output reg [7:0] sum_out,
  output i_eq_1, j_eq_1
);

  reg [2:0] i, j;
  wire [2:0] i_MUX_out, j_MUX_out, i_new, j_new;
  wire [7:0] sum_add_out, sum_MUX_out;

```

Figure 4: Verilog Implementation Part 1

```
56          always @(posedge Clk)
57                  begin
58                          if (i_en)
59                                  i <= i_MUX_out;
60                          if (j_en)
61                                  j <= j_MUX_out;
62                          if (sum_en)
63                                  sum_out <= sum_MUX_out;
64                  end
65
66          assign i_MUX_out = i_MUX_sel ? i_new : N;
67          assign j_MUX_out = j_MUX_sel ? j_new : i_MUX_out;
68          assign sum_MUX_out = sum_MUX_sel ? sum_add_out : 8'b0;
69          assign i_new = i - 1;
70          assign j_new = j - 1;
71          assign i_eq_1 = ( i==1 );
72          assign j_eq_1 = ( j==1 );
73          assign sum_add_out = sum_out + i;
74
75  endmodule
76
77
78
79  module FSM(
80    input ack, Rst, Clk, N_valid, i_eq_1, j_eq_1,
81    output reg i_en, j_en, i_MUX_sel, j_MUX_sel, sum_en, sum_MUX_sel,
82    output sum_valid
83  );
84
85
86          reg[1:0] state;
87          reg[1:0] state_next;
88
89          parameter idle = 2'b00;
90          parameter busy = 2'b01;
91          parameter done = 2'b10;
92
93  //finding current state
94
95          always@(posedge Clk)   // Idle = 0 , Busy = 1 , Done = 2
96          begin
97                  if (Rst)
98                          state <= idle;
99
100                 else
101                         state <= state_next;
102         end
103
```

Figure 5: Verilog Implementation Part 2

```
104 //combinational logic to find next state
105
106 always @(*)
107         case(state)
108                 idle: begin
109                         sum_en = 1'b1;
110                         sum_MUX_sel = 1'b0;
111                         i_MUX_sel = 1'b0;
112                         j_MUX_sel=1'b0;
113                         i_en=1'b1;
114                         j_en=1'b1;
115
116                         if (N_valid ==1) state_next = busy;
117                         else state_next = idle;
118                 end
119
120                 busy: begin
121                         sum_en = 1'b1;
122                         sum_MUX_sel = 1'b1;
123                         if (j_eq_1) begin
124                                 i_MUX_sel=1'b1;
125                                 j_MUX_sel=1'b0;
126                                 i_en = 1'b1;
127                                 j_en=1'b1;
128                         end
129
130                         else begin
131                                 i_MUX_sel = 1'b1;
132                                 j_MUX_sel =1'b1;
133                                 i_en=1'b0;
134                                 j_en=1'b1;
135                         end
136
137                         if (i_eq_1 == 1) state_next =done;
138                         else state_next = busy;
139                 end
140
141                 done: begin
142                         i_en= 1'b0;
143                         j_en= 1'b0;
144                         sum_en = 1'b0;
145                         if (ack==1) state_next = idle;
146                         else state_next = done;
147
148                         end
149         endcase
150
151 //assigning the output
152
153 assign sum_valid = (state == done);
154
155 endmodule
```

Figure 6: Verilog Implementation Part 3

6 Experimental Procedure

```
module Testbench;
  reg [2:0] N;
  reg N_valid, Clk, Rst,ack;
  wire [7:0] sum_out;
  wire sum_valid;

  top_module DUT(N, Clk, Rst, ack, N_valid, sum_out, sum_valid);

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

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

  initial begin
    Rst = 0;
    #10
    N_valid = 1;
    N = 4;
    #15
    Rst = 1;
    #10
    Rst = 0;
    #10
    N_valid = 1;
    #20
    N_valid = 0;
    #300
    ack = 1;
    #10
    ack = 0;
    #10
    Rst = 0;
    #10
    N_valid = 1;
    N = 3;
    #15
    Rst = 1;
    #10
    Rst = 0;
    #10
    N_valid = 1;
    #20
    N_valid = 0;
   end

  initial begin
    $monitor($time);
    #600
    $finish;
  end
endmodule
```

Figure 7: Test bench code

In this section, we assess the functionality of our design by constructing an appropriate test bench in Verilog. The objective of the test bench is to provide random inputs (stimuli) to the design and validate the corresponding outputs. Employing a test bench serves as a robust means for verifying the functionality of our design and ensuring its adherence to the specified requirements. We employed a linear Testbench to to do the verification.

Utilizing the test bench, we supply inputs to the top_module and capture the results in the form of output waveforms, as shown in Figure

We generated a value of 4 followed by 3 in the current example.

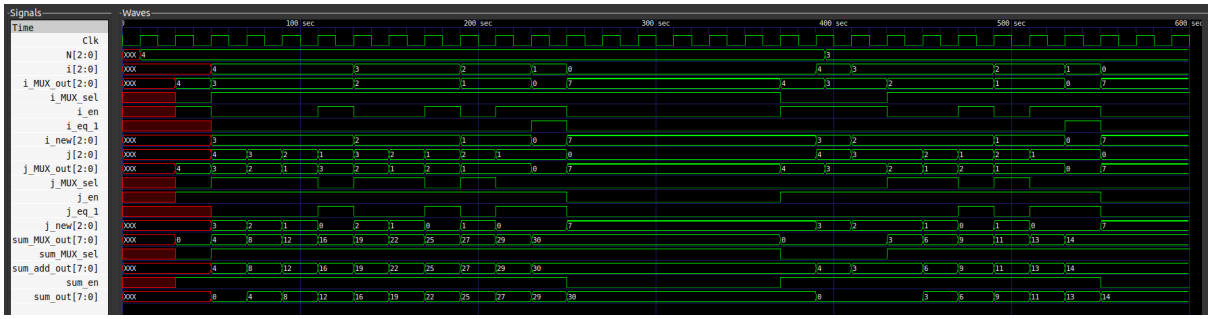

Figure 8: Output waveforms

In the provided example, the value of N is 4. Initially, i is set to 4, and the system transitions from the IDLE state to the BUSY state. Here, j assumes the value of i_mux_out, which equals 4. As j decrements from 4 to 1, the value of sum_out becomes $4 + 4 + 4 + 4 = 16$. Upon j reaching one, the system transitions to the next state. At the onset of this state, j is zero until it is updated with a new value. Subsequently, the value of i_mux_out becomes 3, and upon the next rising edge of the clock, the state transitions back to 1 and the value of I becomes 3. The sum is updated from 0 to 25, and i_acc resets to zero. The process repeats as j decrements until it reaches 1, continuing until the value of i becomes 1. At this point, the system reaches the final state, where sum_valid becomes one and sum equals 30.

Similarly we got the value of sum_out as 13 for the value of N equal to 3

7 Conclusion

In our experiment, we developed a Verilog-based digital system to compute the sum of squares of the first N natural numbers. Utilizing hierarchical modeling proved to be highly beneficial in enhancing the overall design quality and enhancing usability. By decomposing the system into smaller, more manageable modules, we could conduct thorough testing and rectify any issues before integrating the modules into the top module.