Memo 7

Drexel University

To: Dr Christopher Peters
From: Mahad Faisal
Date: 03/15/2024
Re: Lab 7- Adders and Subtractors

Purpose

The purpose of this lab is to develop a 4-bit carry ripple adder/subtractor using the Verilog hardware descriptive language with a piecemeal approach.

Method

The first step in our project is to simulate and test a full adder, which is a crucial component of a carry ripple adder/subtractor. The full adder comprises two AND gates, two XOR gates, and one OR gate. Its inputs include A, B (representing the binary numbers to be added/subtracted), and CIN (the carry-in value), while the outputs are S (the sum of A and B) and COUT (the carry-out value).

To implement the full adder, we need to create a Verilog module. The Verilog code for the full adder is straightforward, utilizing wires for continuous evaluation of inputs and outputs. Here's the Verilog implementation:

```
module full_adder(input A, input B, input CIN, output S, output COUT);
  wire A, B, CIN, X, Y, S, COUT;

  assign X = A ^ B;
  assign Y = A && B;
  assign S = CIN ^ X;
  assign COUT = Y || (CIN && X);

endmodule
```

In this implementation, A, B, and CIN are inputs, while S and COUT are outputs. The equations defining the output values are directly implemented using continuous assignment.

After implementing the full adder, the next step is to test it using a testbench. The testbench iterates over all possible combinations of inputs (A, B, CIN) to validate the functionality of the full adder. Here's the Verilog code for the testbench:

```
module full_adder_tb;

reg A, B, CIN;
wire S, COUT;

integer i;

full_adder dut(.A(A), .B(B), .CIN(CIN), .S(S), .COUT (COUT));

initial begin
  for (i = 0; i < 8; i = i +1) begin
    CIN <= (i >> 2) % 2;
    A <= (i >> 1) % 2;
    B <= i % 2;
    #1;
    $display("CIN = %b, A = %b, B = %b, S = %b, COUT = %b",CIN, A, B, S, COUT);
  end
  $finish();
end

endmodule
```

The testbench employs a for loop to generate all possible input combinations and then displays the resulting output values. The variables A, B, and CIN are assigned values based on the current iteration of the loop, and the results are printed using the $display system task.

In summary, the full adder is implemented as a Verilog module, and its functionality is tested using a testbench that verifies its output for all possible input combinations. This process ensures that the full adder operates correctly before integrating it into larger circuits.

Secondly, the two-bot ripple carry adder code is provided:

```verilog
module full_adder(A, B, CIN, S, COUT);
  input A;
  input B;
  input CIN;
  output S;
  output COUT;

  wire A, B, CIN, X, Y, S, COUT;

  assign X = A ^ B;
  assign Y = A && B;
  assign S = CIN ^ X;
  assign COUT = Y || (CIN && X);

endmodule

module ripple_carry_adder(A, B, CIN, S, COUT);
  input [1:0] A, B;
  input CIN;
  output [1:0] S;
  output COUT;
  wire w1;

  full_adder adder0(A[0], B[0], CIN, S[0], w1);
  full_adder adder1(A[1], B[1], w1, S[1], COUT);
endmodule
```

```verilog
module ripple_carry_adder_tb;
   reg [1:0] A;
   reg [1:0] B;
   reg CIN;
   wire [1:0]S;
   wire COUT;

   ripple_carry_adder dut(A, B, CIN, S, COUT);

   initial begin
     A <= 2'b11;
     B <= 2'b10;
     CIN <= 0;
     #1;
     $display("A = %2b, B = %2b, CIN = %b, S=%2b, COUT = %b",
              A, B, CIN, S, COUT);
     $finish();
   end

endmodule
```

The Verilog code presented implements a two-bit ripple carry adder, a core element in digital circuitry for binary addition. It comprises two modules: full_adder and ripple_carry_adder.

The full_adder module represents a single-bit full adder, taking three inputs (A, B, and CIN) and producing two outputs (S and COUT), indicating the sum and carry-out, respectively. Utilizing XOR and AND gates, it computes these bits based on the input values.

The ripple_carry_adder module employs two full_adder instances to form a two-bit adder. It connects the carry-out of the first adder to the carry-in of the second, enabling carry bit propagation. This module accepts two two-bit input vectors (A and B) and a single carry-in bit (CIN), yielding a two-bit sum (S) and a carry-out bit (COUT).

The accompanying testbench ripple_carry_adder_tb initializes inputs (A, B, and CIN) with specific values and simulates the adder's behavior. It then displays input and output values using $display statements and ends the simulation. This testbench aids in verifying the adder's functionality by testing it with predefined inputs and observing corresponding outputs.

In essence, the Verilog code implements a two-bit ripple carry adder using modular design principles, with a focus on full_adder and ripple_carry_adder modules. The testbench facilitates validation by assessing the adder's performance against predetermined inputs.

The Verilog implementation for the 4-bit adder/subtractor can be seen below:

SV/Verilog Testbench

```
module ripple_carry_adder_tb;

reg [3:0] A;
reg [3:0] B;
reg SUB;
wire [3:0] S;
wire OV_RF;

ripple_carry_adder dut(
    .A(A),
    .B(B),
    .SUB(SUB),
    .S(S),
    .OV_RF(OV_RF)
);

initial begin
    // Test cases for addition
    $display("Addition Tests:");
    A = 4'b0000; B = 4'b0000; SUB = 0;
    #1;
    $display("Test 1: A = %b, B = %b, SUB = %b, S = %b, OV_RF =
%b", A, B, SUB, S, OV_RF);

    A = 4'b1011; B = 4'b0111; SUB = 0;
    #1;
    $display("Test 2: A = %b, B = %b, SUB = %b, S = %b, OV_RF =
%b", A, B, SUB, S, OV_RF);

    // Test cases for subtraction
    $display("\nSubtraction Tests:");
    A = 4'b1011; B = 4'b0111; SUB = 1;
    #1;
    $display("Test 3: A = %b, B = %b, SUB = %b, S = %b, OV_RF =
%b", A, B, SUB, S, OV_RF);

    A = 4'b1101; B = 4'b0011; SUB = 1;
    #1;
    $display("Test 4: A = %b, B = %b, SUB = %b, S = %b, OV_RF =
%b", A, B, SUB, S, OV_RF);


    $finish;
end

endmodule
```

```
module full_adder (
    input A,
    input B,
    input CIN,
    output S,
    output COUT
);

assign S = A ^ B ^ CIN; // Sum output
assign COUT = (A & B) | (B & CIN) | (A & CIN); // Carry
output

endmodule

module ripple_carry_adder (
    input [3:0] A,
    input [3:0] B,
    input SUB,
    output [3:0] S,
    output OV_RF
);

wire [3:0] B_subtracted;

assign B_subtracted = B ^ {SUB, SUB, SUB, SUB};

wire [3:0] carry;

full_adder fa0(A[0], B_subtracted[0], SUB, S[0],
carry[0]);
full_adder fa1(A[1], B_subtracted[1], carry[0], S[1],
carry[1]);
full_adder fa2(A[2], B_subtracted[2], carry[1], S[2],
carry[2]);
full_adder fa3(A[3], B_subtracted[3], carry[2], S[3],
carry[3]);

assign OV_RF = carry[3] ^ carry[2];

endmodule
```

The Verilog code presents a comprehensive implementation of a ripple carry adder, featuring two modules: **full_adder** and **ripple_carry_adder**. This design allows for the addition and subtraction of four-bit binary numbers with efficient utilization of hardware resources.

Firstly, the **full_adder** module encapsulates the functionality of a single-bit full adder. It takes three inputs representing the two operands (A and B) and the carry-in (CIN), and produces two outputs: the sum (S) and the carry-out (COUT). The sum output (S) is computed using the XOR operation, while the carry-out (COUT) is determined through the logical OR operation of various combinations of the input bits.

Secondly, the **ripple_carry_adder** module extends the functionality to perform addition and subtraction operations on four-bit binary numbers. It takes two four-bit input vectors (A and B), along with a subtraction control signal (SUB), and produces a four-bit sum output (S) and an overflow/underflow flag (OV_RF). This module orchestrates four instances of the **full_adder** module to execute the binary addition or subtraction operation, depending on the value of the subtraction control signal.

Finally, the testbench module **ripple_carry_adder_tb** provides a series of test cases to validate the functionality of the ripple carry adder. It sets input values (A, B, and SUB) for both addition and subtraction operations and verifies the output (S) and overflow (OV_RF) against the expected results.

In summary, this Verilog design offers a versatile and efficient solution for performing arithmetic operations on four-bit binary numbers, demonstrating the flexibility and utility of ripple carry adders in digital circuit design.

Results

```
Addition Tests:
Test 1: A = 0000, B = 0000, SUB = 0, S = 0000, OV_RF = 0
Test 2: A = 1011, B = 0111, SUB = 0, S = 0010, OV_RF = 0

Subtraction Tests:
Test 3: A = 1011, B = 0111, SUB = 1, S = 0100, OV_RF = 1
Test 4: A = 1101, B = 0011, SUB = 1, S = 1010, OV_RF = 0
Finding VCD file...
```

These tests were verified manually and were found to be correct so the Verilog implementation of the 4-bit adder/subtractor is successful.

Memo 6

Drexel University

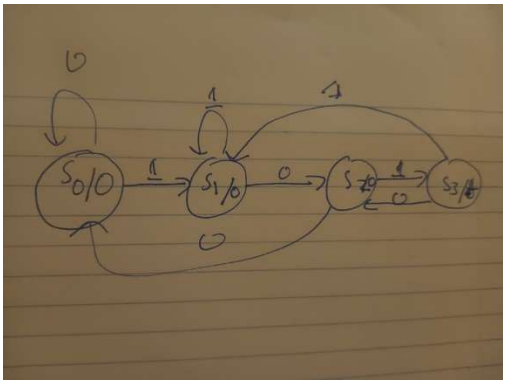
To: Dr Christopher Peters
From: Mahad Faisal
Date: 02/29/2024
Re: Lab 6- Finite State Machines

Purpose

The purpose of this lab is to demonstrate how a finite state machine can be used to detect a sequence ‘101’ with overlapping. For this project, the finite state machine will first be designed using a state transition drawing, table, circuit and a state table.

Methods

The first step in designing this finite state machine is to draw a state transition diagram. Since the sequence is 101 with an overlap using a Moore machine is suitable and the state transition diagram looks like this:



State 0 is when the input is 0 and out is zero as well. The next state is reached when the input is 1 and that state is defined as State 1. Since the next number in the sequence is 0, the input must be zero for the state to transition to State 2 or else it returns to state 2. At state 3 the input must be 0 to complete the sequence and return an output of 1. This helps us design the state transition table which is used in writing the Verilog code seen in Fig 1.1. Using the state transition table 3 equations were derived: F= AB, A* = x’B + xAB’ and B* = x. This was used to develop a set of inputs and outputs to determine hat the correct output of the code should be.

In the Verilog there is the clock signal (CLK), which serves as the timing reference for the entire design. It toggles at regular intervals determined by the specified clock period (CLK_PERIOD), ensuring synchronous operation and precise timing control within the FSM simulation environment.