PROJECT STEP 2

THE TURING MACHINE

List of Group Members

Brandon Markham

Paul Henson

Sam Arshad

CS.3339 Computer Architecture

Texas State University

March 25, 2024

1 Introduction

For the second section of the project, the group produced Verilog code to generate arithmetic logic unit (ALU) functions of variable bit widths. Several operations were implemented including bit-wise logic, integer arithmetic, and bit shifting.

2 Verilog Code

In this section, each operation's Verilog code will be shown and explained. Afterward, the code utilized to test the operations will be shown and explained.

NOTE: Each module includes a parameter **WIDTH** that determines the number of bits in the input and output vectors. By specifying different values for **WIDTH**, these modules may be instantiated with different numbers of input/output bits.

2.1 Bit-Wise Logic Modules

Each multi-bit gate module performs a bit-wise logic operation upon two input vectors 'a' and 'b' of length **WIDTH** and returns the result of that operation through its output wire 'out'.

2.1.1 Multi-Bit AND Module

```
module multibit_AND
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,  input [WIDTH-1 : 0] b,
    output [WIDTH-1 : 0] out);
  assign out = a & b;
endmodule
```

2.1.2 Multi-Bit NAND Module

```
module multibit_NAND
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,  input [WIDTH-1 : 0] b,
    output [WIDTH-1 : 0] out);
  assign out = ~(a & b);
endmodule
```

2.1.3 Multi-Bit OR Module

```
module multibit_OR
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,  input [WIDTH-1 : 0] b,
    output [WIDTH-1 : 0] out);
  assign out = a | b;
endmodule
```

2.1.4 Multi-Bit NOR Module

```
module multibit_NOR
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,  input [WIDTH-1 : 0] b,
    output [WIDTH-1 : 0] out);
  assign out = ~(a | b);
endmodule
```

2.1.5 Multi-Bit XOR Module

```
module multibit_XOR
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,  input [WIDTH-1 : 0] b,
    output [WIDTH-1 : 0] out);
  assign out = a ^ b;
endmodule
```

2.1.6 Multi-Bit XNOR Module

```
module multibit_XNOR
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,  input [WIDTH-1 : 0] b,
    output [WIDTH-1 : 0] out);
  assign out = a ~^ b;
endmodule
```

2.1.7 Multi-Bit NOT Module

```
module multibit_NOT
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,
    output [WIDTH-1 : 0] out);
  assign out = ~a;
endmodule
```

2.2 Integer Arithmetic Operations

NOTE: Operations inside these modules are performed in **"always"** blocks which valuate and update the outputs based on the inputs every time an input changes value. The **"assign"** statements are used to assign the final logic to each output wire.

2.2.1 Binary Multiplier

The **binaryMultiplier** module multiplies together two binary numbers provided by inputs **a** and **b**. The output of this multiplication takes the form of a binary number of twice the length of the original inputs, split between its most significant bits and least significant bits across two output wires, **MSB** and **LSB**.

```
module binaryMultiplier
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,
    input [WIDTH-1 : 0] b,
    output [WIDTH-1 : 0] MSB,
    output [WIDTH-1 : 0] LSB );

  reg [(2*WIDTH)-1 : 0] ans;

  always @(*) begin

    ans = a * b;

  end

  assign LSB = ans[WIDTH-1 : 0];
  assign MSB = ans[(2*WIDTH)-1 : WIDTH];

endmodule
```

2.2.2 Binary Adder

The **binaryAdder** module performs binary addition. It takes two multi-bit inputs **a** and **b** along with a **carryin** input and produces two outputs: a multi-bit output labelled **out** & **carryout**.

```
module binaryAdder
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,
```

```
    input [WIDTH-1 : 0] b,
    input carryin,
    output [WIDTH-1 : 0] out,
    output carryout );

  reg [WIDTH : 0] ans;

  always @(*) begin

    ans = a + b + carryin;

  end

  assign out = ans[WIDTH-1 : 0];
  assign carryout = ans[WIDTH];

endmodule
```

2.2.3 Binary Subtractor

The **binarySubtractor** module performs binary subtraction. It takes two multi-bit inputs **a** and **b** along with a **carryin** input to be used in the case of a borrow bit being necessary when running multiple subtractors in parallel. It produces two outputs: a multi-bit output labelled **out** & **carryout**.

```

module binarySubtractor
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,
    input [WIDTH-1 : 0] b,
    input carryin,
    output [WIDTH-1 : 0] out,
    output carryout );

  reg [WIDTH : 0] ans;

  always @(*) begin

    ans = {1'b1,a};
    ans = ans - b - carryin;


  end
```

```
19
20    assign carryout = ~ans[WIDTH];
21    assign out = ans[WIDTH-1 : 0];
22
23  endmodule
```

2.2.4 Binary Divider

The **binaryDivision** module performs binary division. It takes two multi-bit inputs **a** (dividend) & **b** (divisor). It produces two outputs, the whole number quotient and the remainder as seen with the output labels **ans** & **rem**.

```
1  module binaryDivider
2    #(parameter WIDTH = 4) (
3      input [WIDTH-1 : 0] a,
4      input [WIDTH-1 : 0] b,
5      output reg [WIDTH-1 : 0] ans,
6      output reg [WIDTH-1 : 0] rem );
7
8    always @(*) begin
9
10     ans = a / b;
11     rem = a % b;
12
13   end
14
15 endmodule
```

2.2.5 Bit Shifter

The **multiShift** module performs a multi-bit shifting operation. It takes two input **in** and **control**. It produces two multi-bit **outSubject** (the shifted subject) & **outOverflow** (the bits that overflow during shifting).
For convenient switching of outputs based on shift direction, the outputs of this module are of type ‘reg’ so as to be dynamically assigned inside the **"always"** block.

```
1  module multiShift
2    #(parameter WIDTH = 4) (
3      input [WIDTH-1 : 0] in,
4      input [WIDTH-1 : 0] control,
5      output reg [WIDTH-1 : 0] outSubject,
6      output reg [WIDTH-1 : 0] outOverflow );
7
```

```
  reg [(2*WIDTH)-1 : 0] ans;

  wire dir  = control[WIDTH-1]; // 1 <-- left ; right --> 0
  wire fill = control[0];
  wire [WIDTH-1 - 2 : 0] amt = control[WIDTH-1 - 1 : 1];

  integer i;

  always @(*) begin
    if (dir) begin
      ans = in << amt;
      if(amt>0) begin
        for (i = 0; i < amt; i = i+1) begin
          ans[i] = fill;
        end
      end
      outSubject  = ans[WIDTH-1 : 0];
      outOverflow = ans[(2*WIDTH)-1 : WIDTH];
    end
    else begin
      ans = in << WIDTH;
      ans = ans >> amt;
      if(amt>0) begin
        for(i = 0; i < amt; i = i+1) begin
          ans[(2*WIDTH-1) - i] = fill;
        end
      end
      outSubject  = ans[(2*WIDTH)-1 : WIDTH];
      outOverflow = ans[WIDTH-1 : 0];
    end

  end


endmodule
```

2.3 Testbench File

This testbench code instantiates each type of operation module and applies a number of different input cases to each. This testbench was used to produce the waveforms show later on in this report.
While the examples shown in this report use a 4-bit register width, each module was written with scalability in mind. The 'BENCHWIDTH' parameter may be scaled to larger sizes in order to produce a ALU operations of higher register sizes.

```
// Code your testbench here
// or browse Examples
//`timescale 1ns / 1ps

module testbench;

  // Parameters
  parameter BENCHWIDTH = 4 ;
  //*********************^ *****
//REGISTER SIZE PARAMETER!
//Scale to 4, 8, 16, 32 for extra credit criteria.

  // Inputs
  reg [BENCHWIDTH-1 : 0] a;
  reg [BENCHWIDTH-1 : 0] b;
  reg carryin;

  // Outputs
  wire [BENCHWIDTH-1 : 0] MSB_Product;
  wire  [BENCHWIDTH-1 : 0] LSB_Product;
  wire  [BENCHWIDTH-1 : 0] Quotient;
  wire  [BENCHWIDTH-1 : 0] Remainder;
  wire  [BENCHWIDTH-1 : 0] Difference;
  wire  CarryOut_Difference;
  wire  [BENCHWIDTH-1 : 0] Sum;
  wire  CarryOut_Sum;
  wire  [BENCHWIDTH-1 : 0] Shift_Subject;
  wire  [BENCHWIDTH-1 : 0] Shift_Overflow;

  wire  [BENCHWIDTH-1 : 0]  AND_out;
  wire  [BENCHWIDTH-1 : 0]  NAND_out;
  wire  [BENCHWIDTH-1 : 0]  OR_out;
  wire  [BENCHWIDTH-1 : 0]  NOR_out;
  wire  [BENCHWIDTH-1 : 0]  XOR_out;
```

```
35  wire  [BENCHWIDTH-1 : 0]  XNOR_out;
36  wire  [BENCHWIDTH-1 : 0]  NOT_out;
37
38  multibit_AND  #(.WIDTH(BENCHWIDTH)) and0  ( .a(a), .b(b), .out(AND_out) )
      ;
39  multibit_NAND #(.WIDTH(BENCHWIDTH)) nand0 ( .a(a), .b(b), .out(NAND_out)
      );
40  multibit_OR #(.WIDTH(BENCHWIDTH)) or0 ( .a(a), .b(b), .out(OR_out)  );
41  multibit_NOR  #(.WIDTH(BENCHWIDTH)) nor0  ( .a(a), .b(b), .out(NOR_out) )
      ;
42  multibit_XOR  #(.WIDTH(BENCHWIDTH)) xor0  ( .a(a), .b(b), .out(XOR_out) )
      ;
43  multibit_XNOR #(.WIDTH(BENCHWIDTH)) xnor0 ( .a(a), .b(b), .out(XNOR_out)
      );
44  multibit_NOT  #(.WIDTH(BENCHWIDTH)) not0  ( .a(a),        .out(NOT_out) )
      ;
45
46  // Instantiate the modules
47  binaryMultiplier  #(.WIDTH(BENCHWIDTH)) mult_inst (.a(a), .b(b), .MSB(
      MSB_Product), .LSB(LSB_Product));
48  binaryDivider   #(.WIDTH(BENCHWIDTH)) div_inst  (.a(a), .b(b), .ans(
      Quotient), .rem(Remainder));
49  binarySubtractor  #(.WIDTH(BENCHWIDTH)) sub_inst  (.a(a), .b(b), .carryin
      (carryin), .out(Difference), .carryout(CarryOut_Difference));
50  binaryAdder   #(.WIDTH(BENCHWIDTH)) add_inst  (.a(a), .b(b), .carryin(
      carryin), .out(Sum), .carryout(CarryOut_Sum));
51  multiShift    #(.WIDTH(BENCHWIDTH)) shift_inst  (.in(a), .control(b), .
      outSubject(Shift_Subject), .outOverflow(Shift_Overflow));
52
53
54
55  // Stimulus
56  initial begin
57
58    // Dump waves to file to be read by wave viewer
59    $dumpfile("dump.vcd");
60    $dumpvars(1);
61
62    // Test vectors
63    a = 4'b0000;
64    b = 4'b1111;
65    carryin = 1'b0;
66    #1;
```

```
67      a = 4'b0101;
68      b = 4'b0101;
69      carryin = 1'b1;
70      #1;
71      a = 4'b0101;
72      b = 4'b1010;
73      carryin = 1'b0;
74      #1;
75      a = 4'b1010;
76      b = 4'b1010;
77      carryin = 1'b1;
78      #1;
79      a = 4'b0011;
80      b = 4'b0110;
81      carryin = 1'b0;
82      #1;
83      a = 4'b0011;
84      b = 4'b1100;
85      carryin = 1'b1;
86      #40;
87
88    end
89
90  endmodule
```

3 Waveform Tests

In this section we will showcase the wave forms created using our test benches for each circuit we coded in Verilog. We used **EDAPlayground** and the **EPWave** generator to create these wave forms. Additionally, the Verilog circuit visualizer at digitaljs.tilk.eu was utilized for efficient testing of modules outside of the context of generating waveforms for this report.

3.1 Bit-Wise Logic Waveforms

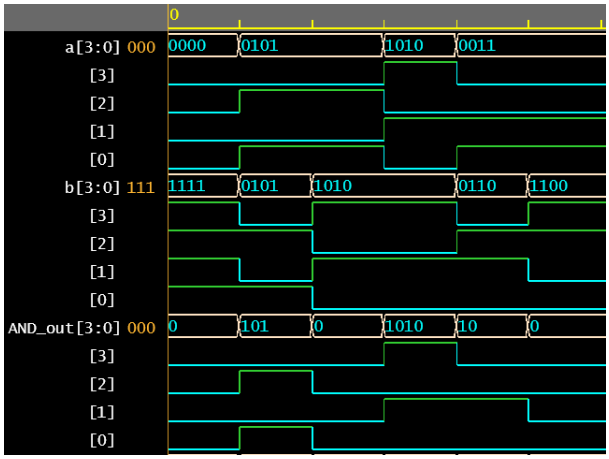

Figure 1: AND Logic Module Waveforms

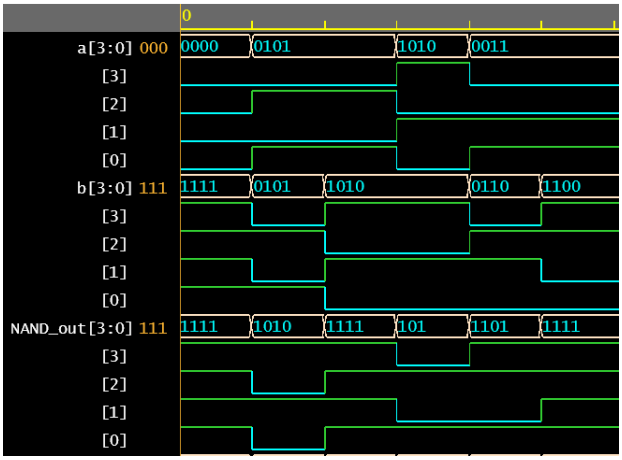

Figure 2: NAND Logic Module Waveforms

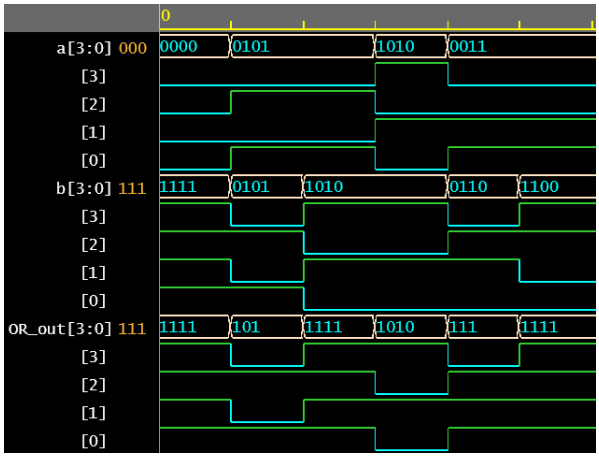

Figure 3: OR Logic Module Waveforms

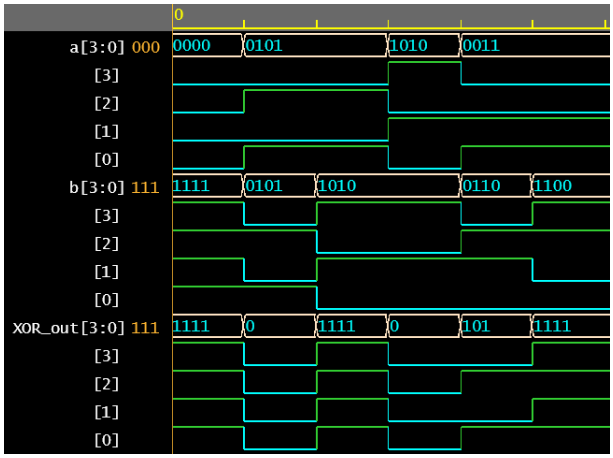

Figure 4: XOR Logic Module Waveforms

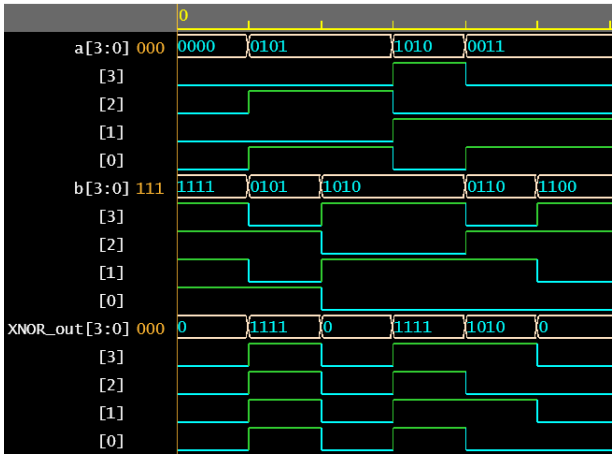

Figure 5: XNOR Logic Module Waveforms

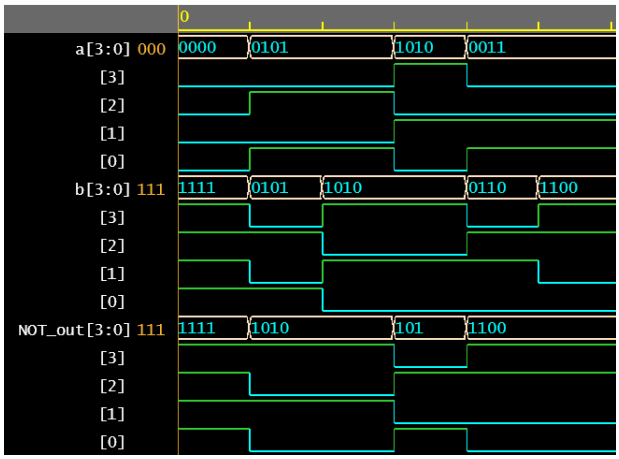

Figure 6: NOT Logic Module Waveforms

3.2 Arithmetic Waveforms

Figure 7: Addition Waveforms

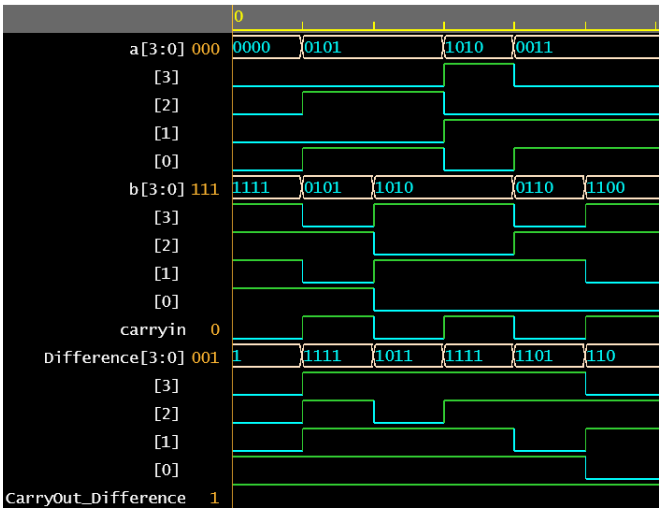

Figure 8: Subtraction Waveforms

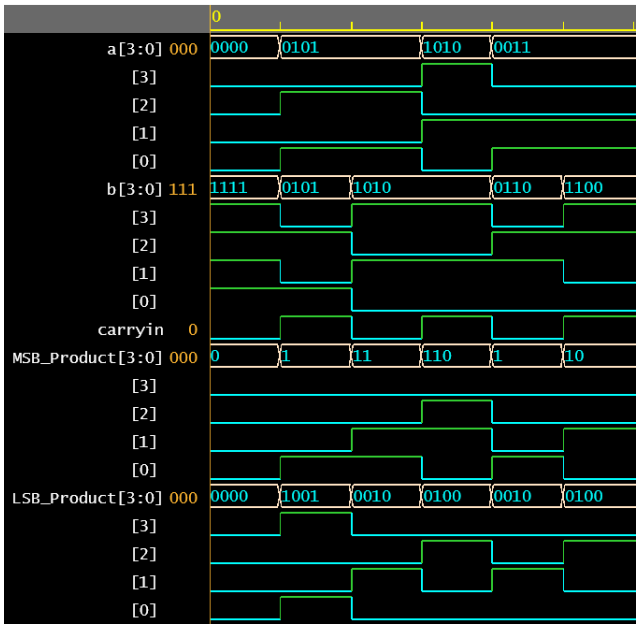

Figure 9: Multiplication Waveforms

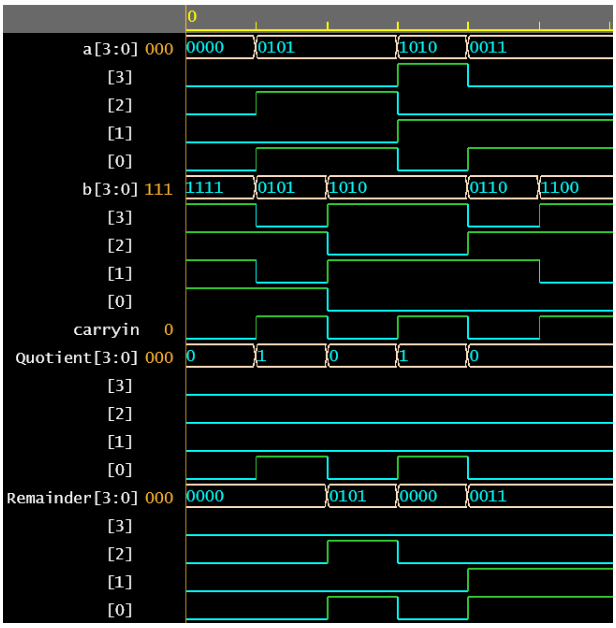

Figure 10: Division Waveforms

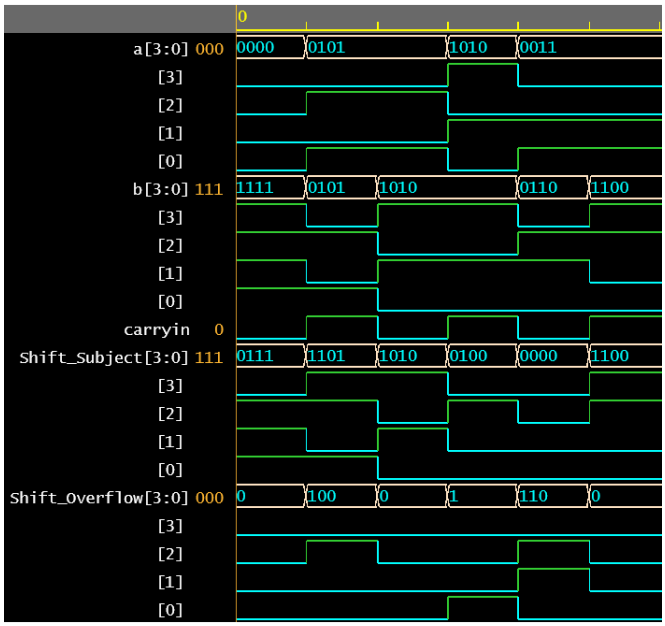

Figure 11: Bitshifting Waveforms

4 Conclusion

In conclusion, we successfully created and tested the required multi-bit binary logic circuits and arithmetic integer operation modules using using **EDAPlayground** and its waveform generation feature **EPWave Waveform Viewer**. Due to misunderstanding instructions, writing the code was challenging until the misunderstanding was clarified. The scale of this project has been greatly simplified in our minds, and we are now nailing down our solutions with the goal of going above and beyond as well as time permits for the final part of the project.