PROJECT STEP 3

TURING MACHINE

List of Group Members

Brandon Markham

Paul Henson

Sam Arshad

CS.3339 Computer Architecture

Texas State University

April 16, 2024

1 Introduction

For this final step of the project, the Turing Machine group implemented, tested and integrated our Verilog code into one package. We used an online web editor and compiler called "EDA Playground". We implemented behavioral-level logic to accomplish these tasks, and used EDA Playground's waveform generator feature in order to visualize the waveform(s).

2 Verilog Modules

In this section, we were required to assemble each previously-created bitwise logic and arithmetic functions into a top-level ALU module. Each module presented below (outside of the final ALU module) are written and function the same as they did in the previous reports with one exception in the bitshifting operation, which is described in its subsection.

2.1 MultiBit AND

```
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
```

2.2 MultiBit NAND

```
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
```

2.3 MultiBit OR

```
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
```

2.4 MultiBit NOR

```
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
```

2.5 MultiBit XOR

```
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
```

2.6 MultiBit XNOR

```
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
```

2.7 MultiBit NOT

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

2.8 Binary Adder

```
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
```

2.9 Binary Subtractor

```
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

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

endmodule
```

2.10 Binary Multiplier

```
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
```

2.11 Binary Dividor

```
module binaryDivider
  #(parameter WIDTH = 4) (
    input [WIDTH-1 : 0] a,
    input [WIDTH-1 : 0] b,
    output reg [WIDTH-1 : 0] ans,
    output reg [WIDTH-1 : 0] rem );

  always @(*) begin

    ans = a / b;
    rem = a % b;

  end

endmodule
```

2.12 Multi Bit Shifter

Notable Version Change: Lines 22 and 36. The continuation criteria of two for loops were expanded from the previous version in order to create an upper ceiling on the number of loops that were possible to be requested by the control input.

```
module multiShift
```

```
#(parameter WIDTH = 4) (
  input [WIDTH-1 : 0] in,
  input [WIDTH-1 : 0] control,
  output reg [WIDTH-1 : 0] outSubject,
  output reg [WIDTH-1 : 0] outOverflow );

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
  //$display("in = %b, control = %b, dir = %b, fill = %b, amt = %b", in,
  control, dir, fill, amt);
  if (dir) begin
    ans = in << amt;
    //$display("ans shifted = %b", ans);
    if(amt>0) begin
      for (i = 0; (i < amt) && (i < WIDTH*2); i = i+1) begin
        ans[i] = fill;
      end
    end
    //$display("ans filled  = %b", ans);
    outSubject  = ans[WIDTH-1 : 0];
    outOverflow = ans[(2*WIDTH)-1 : WIDTH];
  end
  else begin
    ans = in << WIDTH;
    //$display("ans before  = %b", ans);
    ans = ans >> amt;
    //$display("ans shifted = %b", ans);
    if(amt>0) begin
      for(i = 0; (i < amt) && (i < WIDTH*2); i = i+1) begin
        ans[(2*WIDTH-1) - i] = fill;
      end
    end
    //$display("ans filled  = %b", ans);
    outSubject  = ans[(2*WIDTH)-1 : WIDTH];
    outOverflow = ans[WIDTH-1 : 0];
  end
```

```
44
45    end
46
47
48  endmodule
```

3 ALU

The ALU module instantiates each of the modules shown above, and handles routing outputs from the operation selected via an input operation code (opcode).

```
1  module ALU
2    #(parameter WIDTH = 4) (
3      input [3:0] opcode,
4      input [WIDTH-1 : 0] a,
5      input [WIDTH-1 : 0] b,
6      input carryin,
7      output reg [WIDTH-1 : 0] out,
8      output reg [WIDTH-1 : 0] extra );
9
10   wire [WIDTH-1 : 0] and_out;
11   wire [WIDTH-1 : 0] nand_out;
12   wire [WIDTH-1 : 0] or_out;
13   wire [WIDTH-1 : 0] nor_out;
14   wire [WIDTH-1 : 0] xnor_out;
15   wire [WIDTH-1 : 0] xor_out;
16   wire [WIDTH-1 : 0] not_out;
17
18   multibit_AND #(.WIDTH( WIDTH )) alu_AND (
19     .a( a ),
20     .b( b ),
21     .out( and_out ));
22   multibit_NAND #(.WIDTH( WIDTH )) alu_NAND (
23     .a( a ),
24     .b( b ),
25     .out( nand_out ));
26   multibit_OR #(.WIDTH( WIDTH )) alu_OR (
27     .a( a ),
28     .b( b ),
29     .out( or_out ));
30   multibit_NOR #(.WIDTH( WIDTH )) alu_NOR (
31     .a( a ),
32     .b( b ),
```

```
33    .out( nor_out ));
34  multibit_XOR #(.WIDTH( WIDTH )) alu_XOR (
35    .a( a ),
36    .b( b ),
37    .out( xor_out ));
38  multibit_XNOR #(.WIDTH( WIDTH )) alu_XNOR (
39    .a( a ),
40    .b( b ),
41    .out( xnor_out ));
42  multibit_NOT #(.WIDTH( WIDTH )) alu_NOT (
43    .a( a ),
44    .out( not_out ));
45
46
47  wire [WIDTH-1 : 0] add_out;
48  wire [WIDTH-1 : 0] add_carryout;
49  wire [WIDTH-1 : 0] sub_out;
50  wire [WIDTH-1 : 0] sub_carryout;
51  wire [WIDTH-1 : 0] mul_msb;
52  wire [WIDTH-1 : 0] mul_lsb;
53  wire [WIDTH-1 : 0] div_ans;
54  wire [WIDTH-1 : 0] div_rem;
55
56  binaryAdder #(.WIDTH( WIDTH )) alu_add (
57    .a( a ),
58    .b( b ),
59    .carryin( carryin ),
60    .out( add_out ),
61    .carryout( add_carryout ));
62  binarySubtractor #(.WIDTH( WIDTH )) alu_sub (
63    .a( a ),
64    .b( b ),
65    .carryin( carryin ),
66    .out( sub_out ),
67    .carryout( sub_carryout ));
68  binaryMultiplier #(.WIDTH( WIDTH )) alu_mul (
69    .a( a ),
70    .b( b ),
71    .MSB( mul_msb ),
72    .LSB( mul_lsb ));
73  binaryDivider #(.WIDTH( WIDTH )) alu_div (
74    .a( a ),
75    .b( b ),
```

```
76    .ans( div_ans ),
77    .rem( div_rem ));
78
79  wire [WIDTH-1 : 0] shf_sub;
80  wire [WIDTH-1 : 0] shf_ovr;
81  multiShift #(.WIDTH( WIDTH )) alu_shf (
82    .in( a ),
83    .control( b ),
84    .outSubject( shf_sub ),
85    .outOverflow( shf_ovr ));
86
87
88  always_comb begin
89
90    case(opcode)
91      4'b0000 : out = and_out;
92      4'b0001 : out = or_out;
93      4'b0010 : out = nand_out;
94      4'b0011 : out = nor_out;
95      4'b0100 : out = xor_out;
96      4'b0101 : out = xnor_out;
97      4'b0110 : out = not_out;
98      4'b0111 : begin
99        out = add_out;
100       extra = add_carryout;
101     end
102     4'b1000 : begin
103       out = sub_out;
104       extra = sub_carryout;
105     end
106     4'b1001 : begin
107       out = mul_msb;
108       extra = mul_lsb;
109     end
110     4'b1010 : begin
111       out = div_ans;
112       extra = div_rem;
113     end
114     4'b1011 : begin
115       out = shf_sub;
116       extra = shf_ovr;
117     end
118   endcase
```

```
119
120   end
121
122
123
124
125 endmodule
```

4 Verilog Testbench

The testbench was designed in such a way as to be able to test ALU modules with widths of multiples of 4, defined by a WIDTH parameter on line 2 of the testbench file. The tested variables used Verilog's replication operator in order to generate useful input commands of varying length.
In testing, the WIDTH parameter was scaled to values 4, 8, 16, and 32 in order to test higher ALU register sizes.

```
1  module testbench
2    #(parameter WIDTH = 32);
3
4    reg [3:0] opcode;
5    reg [WIDTH-1 : 0] a;
6    reg [WIDTH-1 : 0] b;
7    reg carryin;
8
9    reg [WIDTH-1 : 0] out;
10   reg [WIDTH-1 : 0] extra;
11
12   ALU #(.WIDTH( WIDTH )) alu (
13     .opcode( opcode ),
14     .a( a ),
15     .b( b ),
16     .carryin( carryin ),
17     .out( out ),
18     .extra( extra ));
19
20   integer i;
21
22   initial begin
23
24     // Dump waves to file to be read by wave viewer
25     $dumpfile("dump.vcd");
```

```
26      $dumpvars(1);
27
28
29      for(i = 0; i < 12; i += 1) begin
30        opcode = i;
31        a = {WIDTH/4{4'b1010}};
32        b = {WIDTH/4{4'b0101}};
33        carryin = 0;
34        #1;
35        a = {WIDTH/4{4'b0110}};
36        b = {WIDTH/4{4'b0110}};
37        carryin = 1;
38        #1;
39
40      end
41
42
43    end
44
45  endmodule
```

5 Waveform Test

Shown below are the output waveform tables generated by testing.
Opcode interpretations and "extra" output signal labels have been added to each waveform table image for ease of interpretation.

5.0.1 GATES

Figure 1: 4-bit wide bitwise logic functions.

5.0.2 OPERATIONS

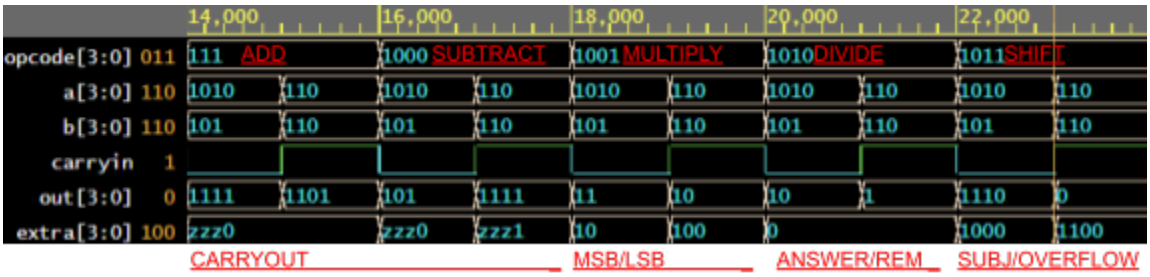

Figure 2: 4-bit wide arithmetic operations.

5.1 Extra Credit: Higher-Width Register Waveforms

5.1.1 8, 16, 32 bit Gates and Operations

Higher-width register testing waveforms shown below have signal values listed in hexadecimal to accommodate reasonable viewing.

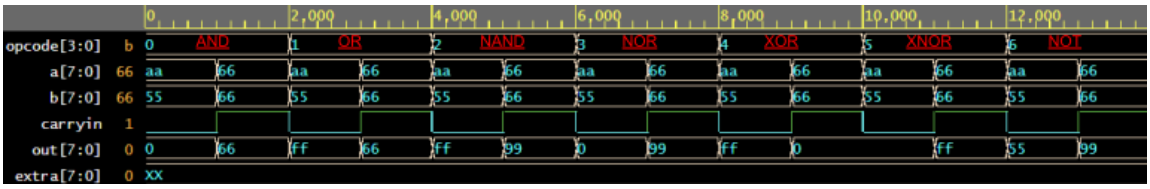

Figure 3: 8-bit wide bitwise logic functions.

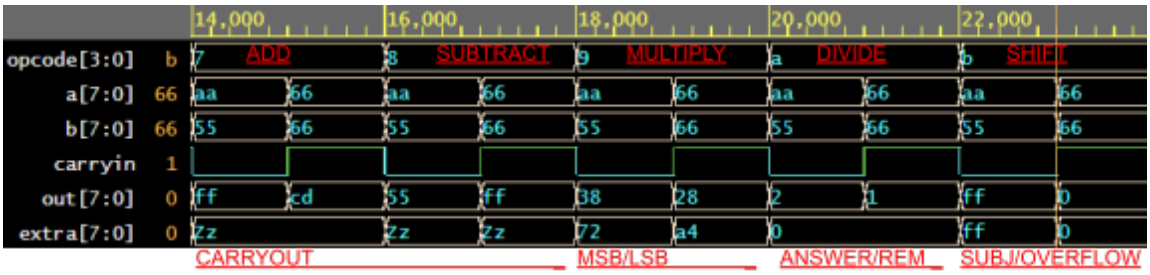

Figure 4: 8-bit wide arithmetic operations.

Figure 5: 16-bit wide bitwise logic functions.

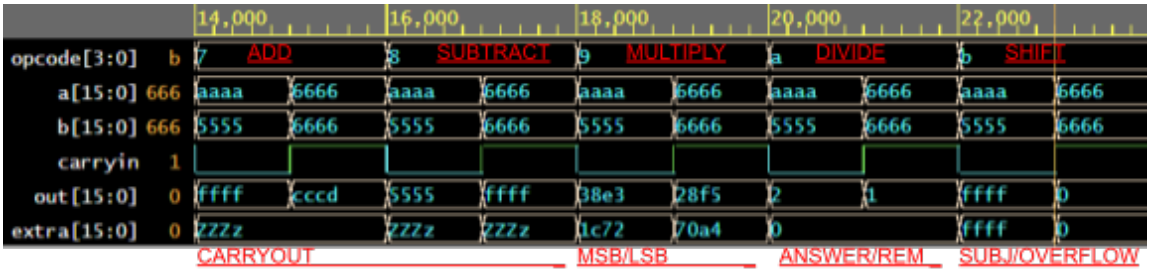

Figure 6: 16-bit wide arithmetic operations.

Figure 7: 32-bit wide bitwise logic functions, table 1 of 2.

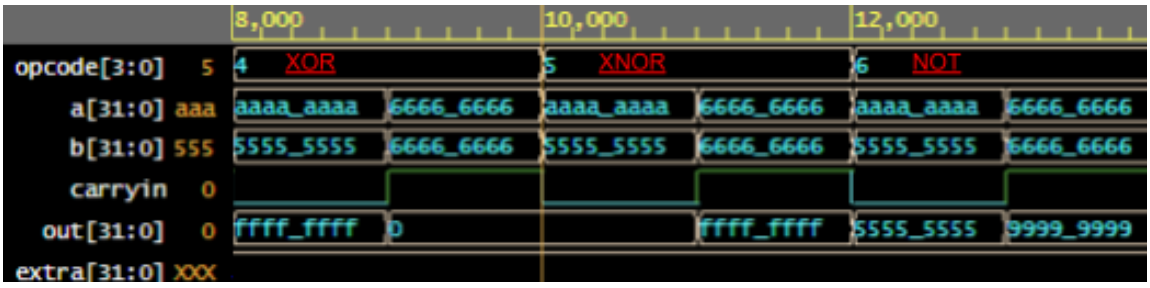

Figure 8: 32-bit wide bitwise logic functions, table 2 of 2.

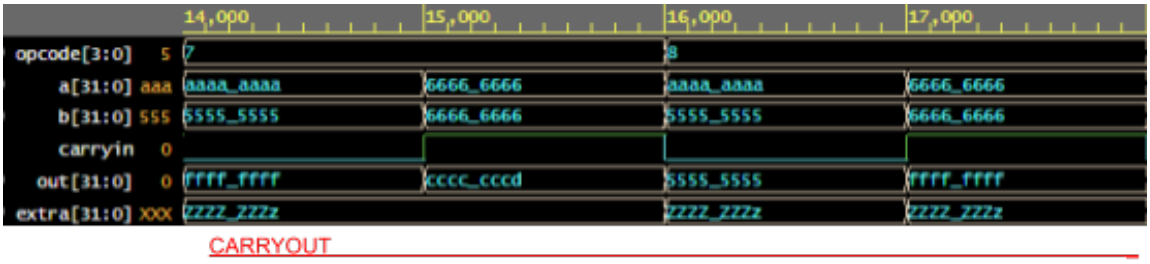

Figure 9: 32-bit wide arithmetic operations, table 1 of 2. Add and Subtract operations shown in order.

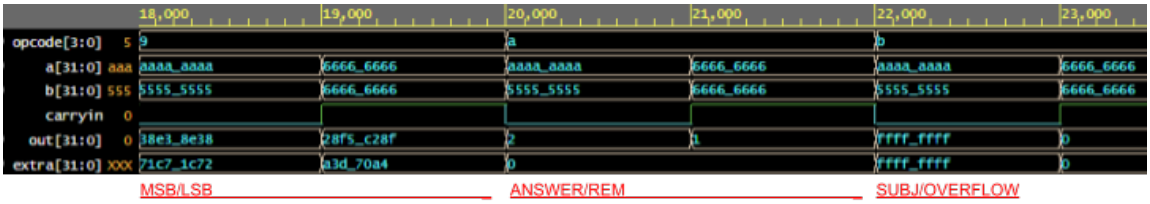

Figure 10: 32-bit wide arithmetic operations, table 2 of 2. Multiply, Divide, and Bitshift operations shown in order.

6 Conclusion

This final segment of the project required very little extra overhead in terms of learning and work as compared to the previous two parts. The testbench was, by now, a familiar format that required only one novel technique in the form of parametrizable variable generation via the replication operator.

Notably, Verilog's switch (case) statement was implemented for the first time in this project inside of the ALU module. Otherwise, only one small adjustment was required between the previously created modules.
The majority of the group's effort for this section was in compiling and verifying the outputs of each test case rather than in learning and writing code, which is nonetheless likely representative of many real-world tasks we may encounter in the future.

As closing notes for the entirety of this project, the members of our group have gained a valuable understanding of both a hardware descriptive language and an advanced markup document-creation language, have gained a sense for the amount of work that must be put in to learn new languages without actively guidance, and have learned a lesson about fully exploring the confines of an assignment and requesting clarification before misguidedly reinventing the wheel.