8-BIT FAST CALCULATOR

Computer Aided Design VLSI

Rutgers University

Junior Brice

Abstract:

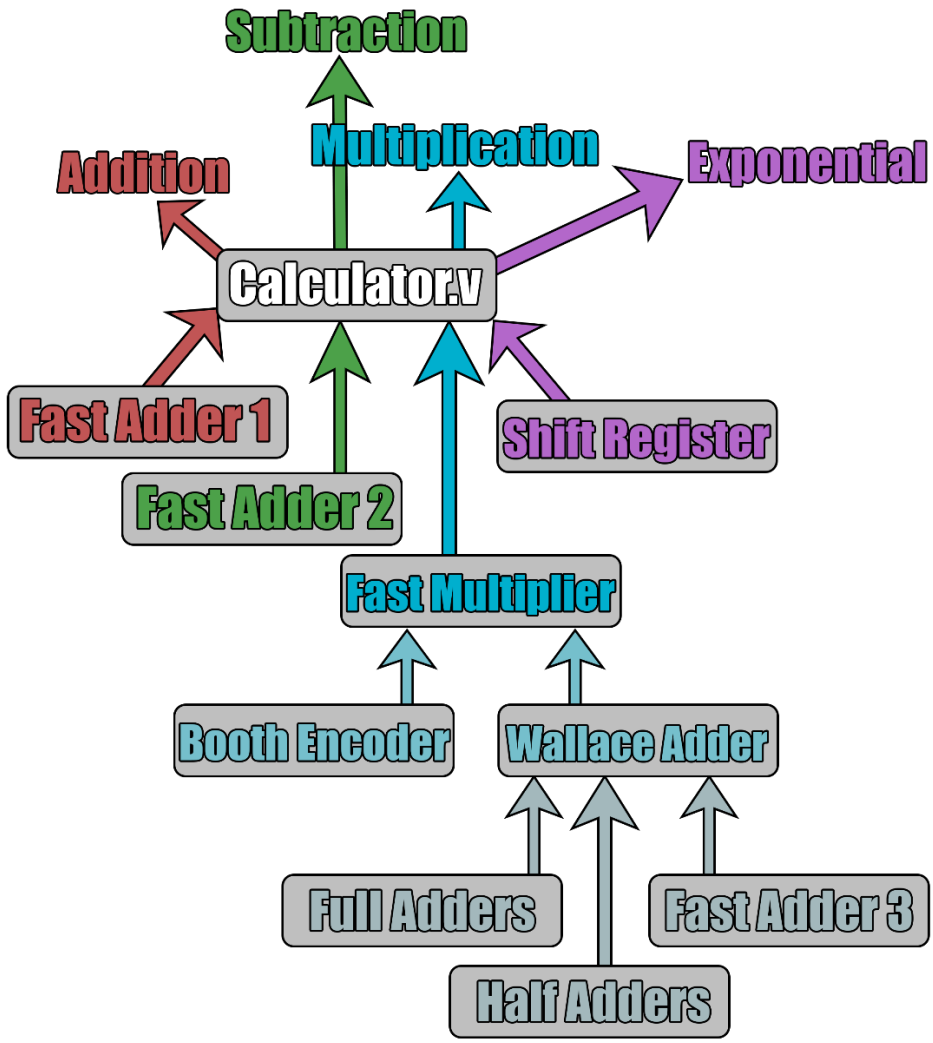
In this report I aim to showcase an 8-bit calculator created in Verilog HDL. I will explain the thought process, decisions, and abnormalities I may have encountered during my time working on this project. This includes references, model descriptions, verification, layered modeling techniques and potential improvements. This calculator will be a fast calculator, focusing on designs oriented around speed, and not by power optimization or surface area.

Introduction:

Our 8-bit calculator will perform 4 functions. **Addition, substruction, multiplication, and 2^y**, where y is one of our two 8-bit inputs. This calculator will be implemented across 7 Verilog modules. The top-level module, *calculator.v,* will instantiate the others in some form to perform the necessary operations.

- ➔ Our **addition** operation will work with both positive and negative numbers.
- ➔ Our **subtraction** operation will work with both positive and negative numbers.
- ➔ Our **multiplication** functionality will be able to multiply any two signed 8-bit numbers, given that their product doesn't overflow. (There are only 8-bits of precision for the output as well; 1 signed bit and 7 numerical bits).
- ➔ Finally, our **exponential** operation will only work with *positive* inputs, and output up to 7-bits of precision (i.e., a maximum calculation of 2^7 due output bus size)

Architecture Visualization:



This architecture visualization is provided for clarity as you read the report. (Created by myself)

Module Descriptions & Methodology:

calculator.v (Top Level Module)

This module simply acts as the mediator for all the functions but does no calculation on its own besides the shift operation required for the exponential. It uses a set of parameters and an input select bit to determine which of the 4 operations the device will perform. Using an always block, it generates a non-priority mux that triggers every time either input operand *x*, input operant *y*, or the select bit is changed. This mux then assigns the output of the calculator to be the output of the selected operation, all of which (except

the exponential) do their calculations inside a submodule. An image of the top-level module is shown for you clarification.

calculator.v

```
1   `include "./carry_look_ahead_adder_8.v"
2   `include "./fast_multiplier.v"
3   module calculator (x, y, sel, out);
4
5   input signed [7:0] x, y;
6   input [1:0] sel;
7   output signed [7:0] out;
8
9   reg signed [7:0] temp_out;
10
11  /* parameters for case statment */
12  parameter ADD = 2'd0;
13  parameter SUB = 2'd1;
14  parameter MUL = 2'd2;
15  parameter EXP = 2'd3;
16
17  /* Fast Addition addition using carry look ahead adder */
18  wire [7:0] fast_add;
19  wire fast_add_carry;
20  carry_look_ahead_adder_8 CLA1 (.a(x), .b(y), .c_in(1'b0), .sum(fast_add),
21                                 .c_out(fast_add_carry));
22
23  /* Fast Subtraction using carry look ahead adder */
24  wire [7:0] fast_sub;
25  wire fast_sub_carry;
26  carry_look_ahead_adder_8 CLA2 (.a(x), .b(~y+1'b1), .c_in(1'b0), .sum(fast_sub),
27                                 .c_out(fast_sub_carry));
28
29  /* Fast Multuplication Using Booth Recoding and wallace addition */
30  wire [15:0] fast_mul;
31  fast_multiplier MUL1 (.x(x), .y(y), .out(fast_mul));
32
33  /* Exponent operation using shift operator */
34  wire [15:0] exp;
35  assign exp = 1'b1 << y;
36
37  /* case selection statement */
38  always @ (x or y or sel)begin
39
40      case (sel)
41          ADD:begin
42                  temp_out = fast_add;
43          end
44
45          SUB:begin
46                  temp_out = fast_sub;
47          end
48
49          MUL:begin
50                  temp_out = fast_mul;
51          end
52
53          EXP:begin
54                  temp_out = exp;
55          end
56
57          default:begin
58                  temp_out = 0;
59          end
60      endcase
61  end
62
63  /* final assignment */
64  assign out = temp_out;
65
66  endmodule
```

carry_look_ahead_adder_8.v (Two 8-bit Carry Look Ahead Fast Adders)

In *calculator.v,* we instantiate two identical 8-bit fast carry look ahead fast adders (*CLA1 and CLA2*), but utilize them separately. One for the addition operation of the calculator, and the other for the subtraction operation. An image of this 8-bit CLA adder is shown below:

carry_look_ahead_adder_8.v

```
1   module carry_look_ahead_adder_8 (a, b, c_in, sum, c_out);
2
3   parameter width = 8;
4
5   input c_in;
6   input [width-1:0] a, b;
7   output [width-1:0] sum;
8   output c_out;
9
10  wire [width-1:0] P, G;
11  wire [width:0] C;
12
13  assign C[0] = c_in;
14
15  genvar i;
16  generate
17      for (i=0; i < width; i=i+1)begin
18          assign P[i] = a[i] ^ b[i];
19          assign G[i] = a[i] & b[i];
20      end
21  endgenerate
22
23  generate
24      for (i = 1; i < width+1; i = i + 1)begin
25          assign C[i] = G[i-1] | (P[i-1] & C[i-1]);
26      end
27  endgenerate
28
29  generate for (i = 0; i < width; i=i+1)begin
30      assign sum[i] = P[i] ^ C[i];
31  end
32  endgenerate
33
34  assign c_out = C[width];
35
36  endmodule // carry_look_ahead_adder
```

This adder uses a width parameter for ease of editability, and generate statements for compact code. It utilizes a slightly modified version of the CLA adder I submitted for a prior homework assignment. For the addition operation, we uses the CLA adder as normal, but for subtraction, if you look back to *calculator.v*, we implement 2's compliment in its port wiring to facilitate easy substraction. Since this adder is already optimized to deal with signed inputs, the addition and subtraction operations work flawlessly with them.

fast_multiplier.v, booth_encoder.v, and wallace_addition.v

In *calculator.v*, we instantiate *fast_multipler.v* to facililitate multiplication. Inside the module appears as such:

fast_multiplier.v

```v
`include "./booth_encoder.v"
`include "./wallace_addition.v"
module fast_multiplier (x, y, out);

input signed [7:0] x, y;
output [7:0] out;

wire [15:0] temp_out;
wire [15:0] temp_out_neg;
reg [7:0] temp_x, temp_y;
wire product_sign_flag;

wire [15:0] partial_p [4:0];

/*The always block below will scan the signed bit of the inputs and peform two's compliment
if neccessary, before proceeding into multiplier algorithms*/
always @ (x or y)begin

if (x[7] == 1'b1 && y[7] == 1'b1)begin
    temp_x = {1'b1, ~x[6:0]} + 1'b1;
    temp_y = {1'b1, ~y[6:0]} + 1'b1;
    end

else if (y[7] == 1'b1)begin
    temp_x = x;
    temp_y = {1'b1, ~y[6:0]} + 1'b1;

    end

else if (x[7] == 1'b1)begin
    temp_x = {1'b1, ~x[6:0]} + 1'b1;
    temp_y = y;
    end

else begin
    temp_x = x;
    temp_y = y;
    end

end

/* booth encoding partial products with significant sign extention */
booth_encoder b1 (.x(temp_x), .operand({temp_y[1:0], 1'b0}), .partial_p(partial_p[0]));
booth_encoder b2 (.x(temp_x), .operand(temp_y[3:1]), .partial_p(partial_p[1]));
booth_encoder b3 (.x(temp_x), .operand(temp_y[5:3]), .partial_p(partial_p[2]));
booth_encoder b4 (.x(temp_x), .operand(temp_y[7:5]), .partial_p(partial_p[3]));
booth_encoder b5 (.x(temp_x), .operand({2'b0, temp_y[7]}), .partial_p(partial_p[4]));

/* performing wallace tree addition on the partial products */
wallace_addition w1 (._0PP(partial_p[0]), ._1PP(partial_p[1]), ._2PP(partial_p[2]),
                     ._3PP(partial_p[3]), ._4PP(partial_p[4]), .out(temp_out));

/* Determining the final sign of the multiplication by looking at the signs of the
original operands*/
assign product_sign_flag = x[7] ^ y[7];

/*adjusting the final 2's compliment output based on the sign flag*/

assign temp_out_neg = ~(temp_out-1'b1); //reversing 2's compliment if negative

assign out = product_sign_flag? temp_out_neg[7:0] : temp_out[7:0]; //asigning final out based on sign

endmodule
```

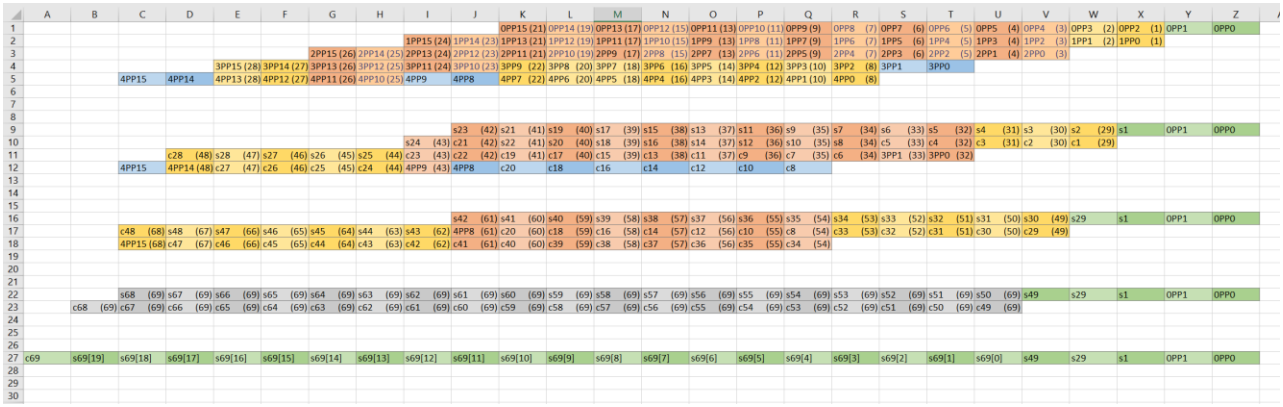
Notice, we instantiate our *booth_encoder.v* module 5 times to generate 5 partial products. These 5 partial products are based on groupings of 3 of the bits of the multiplicand, as dictated by the booth recoding algorithm. The partial products generated are propagated through a wire (*partial_*p) and are each 16 bits in length to provide sufficient sign extension. *booth_encoder.v* looks like such:

booth_encoder.v

```
1  module booth_encoder (x, operand, partial_p);
2
3      input signed [7:0] x;
4      input [2:0] operand;
5      output reg signed [15:0] partial_p;
6
7          always @ (x or operand)begin
8
9              case(operand)
10                 3'b000: partial_p = 16'b0;
11                 3'b001: partial_p = {8'b0, x};
12                 3'b010: partial_p = {8'b0, x};
13                 3'b011: partial_p = {7'b0, x, 1'b0};
14                 3'b100: partial_p = {7'd127, ~x, 1'b1} + 1'b1;
15                 3'b101: partial_p = {8'd255, ~x} + 1'b1;
16                 3'b110: partial_p = {8'd255, ~x} + 1'b1;
17                 3'b111: partial_p = 16'b0;
18             endcase
19
20         end
21
22
23 endmodule
```

Notice this follows the booth algorithm appropriately, and even implements the appropriate shifts, adds compensory bits if required by negative encoded values, and applies bit flips when needed.

After the partial products are encoded, they get passed to the *wallace_addition.v* module, which is very complex looking. For the sake of my planning I created a visualization that is also suitable for this report. Remember, the encoded partial products provided by the booth encoder are 16-bits in length to account for negative encoded values that require sign extension, and the inherent heavy shifts wallace addition implmenments. The wallace addition algorithm in *wallace_addition.v* is implemented exactly as follows:



Feel free to zoom in, or refer to the included excel sheet in the project submission. Each section highlighted in green is a completed column of wallace addition, in yellow are half-adders implemented by *half_adder.v*, in orange are full adders implemented by *full_adder.v*, in blue are bits that will persist to the next stage of addition, and in gray is addition implmented by *carry_look_ahead_adder_20.v* (distinct from the other carry look ahead adders mentioned prviously in the report, as it has 20 bits).

The total wallace addition for my project is broken into 5 stages as shown in the diagram, the last stage being the result. In parentheses, for each half or full-adder, is a number unique to that adder alone, implemented for organizational and troubleshooting purposes. For instance, adder 54 produces c54 (carry-bit 54) and s54 (sum-bit 54).

This diagram maps 1 to 1 to the verilog module of *wallace_addition.v*, which you can see the full module below in the following pages.

wallace_addition.v

```
`include "./half_adder.v"
`include "./full_adder.v"
`include "./carry_look_ahead_adder_20.v"
module wallace_addition (_0PP, _1PP, _2PP, _3PP, _4PP, out);

input signed [15:0] _0PP, _1PP, _2PP, _3PP, _4PP;
output signed [7:0] out;

/* internal sum wires needed */
wire s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13,
    s14, s15, s16, s17, s18, s19, s20, s21, s22, s23, s24,
    s25, s26, s27, s28, s29, s30, s31, s32, s33, s34, s35,
    s36, s37, s38, s39, s40, s41, s42, s43, s44, s45, s46,
    s47, s48, s49, s50, s51, s52, s53, s54, s55, s56, s57,
    s58, s59, s60, s61, s62, s63, s64, s65, s66, s67, s68;

/* interal carry wires needed */
wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13,
    c14, c15, c16, c17, c18, c19, c20, c21, c22, c23, c24,
    c25, c26, c27, c28, c29, c30, c31, c32, c33, c34, c35,
    c36, c37, c38, c39, c40, c41, c42, c43, c44, c45, c46,
    c47, c48, c49, c50, c51, c52, c53, c54, c55, c56, c57,
    c58, c59, c60, c61, c62, c63, c64, c65, c66, c67, c68;

/* internal sum and carry wires for carry look ahead addition stage */
wire [19:0] a69, b69, s69;
wire c69;

/* first stage, first addition */
half_adder ha1 (.a(_0PP[2]), .b(_1PP[0]), .sum(s1), .c_out(c1));
half_adder ha2 (.a(_0PP[3]), .b(_1PP[1]), .sum(s2), .c_out(c2));
full_adder fa3 (.a(_0PP[4]), .b(_1PP[2]), .c_in(_2PP[0]), .sum(s3), .c_out(c3));
full_adder fa4 (.a(_0PP[5]), .b(_1PP[3]), .c_in(_2PP[1]), .sum(s4), .c_out(c4));
full_adder fa5 (.a(_0PP[6]), .b(_1PP[4]), .c_in(_2PP[2]), .sum(s5), .c_out(c5));
full_adder fa6 (.a(_0PP[7]), .b(_1PP[5]), .c_in(_2PP[3]), .sum(s6), .c_out(c6));
full_adder fa7 (.a(_0PP[8]), .b(_1PP[6]), .c_in(_2PP[4]), .sum(s7), .c_out(c7));
half_adder ha8 (.a(_3PP[2]), .b(_4PP[0]), .sum(s8), .c_out(c8));
full_adder fa9 (.a(_0PP[9]), .b(_1PP[7]), .c_in(_2PP[5]), .sum(s9), .c_out(c9));
half_adder ha10 (.a(_3PP[3]), .b(_4PP[1]), .sum(s10), .c_out(c10));
full_adder fa11 (.a(_0PP[10]), .b(_1PP[8]), .c_in(_2PP[6]), .sum(s11), .c_out(c11));
half_adder ha12 (.a(_3PP[4]), .b(_4PP[2]), .sum(s12), .c_out(c12));
full_adder fa13 (.a(_0PP[11]), .b(_1PP[9]), .c_in(_2PP[7]), .sum(s13), .c_out(c13));
half_adder ha14 (.a(_3PP[5]), .b(_4PP[3]), .sum(s14), .c_out(c14));
full_adder fa15 (.a(_0PP[12]), .b(_1PP[10]), .c_in(_2PP[8]), .sum(s15), .c_out(c15));
half_adder ha16 (.a(_3PP[6]), .b(_4PP[4]), .sum(s16), .c_out(c16));
full_adder fa17 (.a(_0PP[13]), .b(_1PP[11]), .c_in(_2PP[9]), .sum(s17), .c_out(c17));
half_adder ha18 (.a(_3PP[7]), .b(_4PP[5]), .sum(s18), .c_out(c18));
full_adder fa19 (.a(_0PP[14]), .b(_1PP[12]), .c_in(_2PP[10]), .sum(s19), .c_out(c19));
half_adder ha20 (.a(_3PP[8]), .b(_4PP[6]), .sum(s20), .c_out(c20));
full_adder fa21 (.a(_0PP[15]), .b(_1PP[13]), .c_in(_2PP[11]), .sum(s21), .c_out(c21));
half_adder ha22 (.a(_3PP[9]), .b(_4PP[7]), .sum(s22), .c_out(c22));
full_adder fa23 (.a(_1PP[14]), .b(_2PP[12]), .c_in(_3PP[10]), .sum(s23), .c_out(c23));
full_adder fa24 (.a(_1PP[15]), .b(_2PP[13]), .c_in(_3PP[11]), .sum(s24), .c_out(c24));
full_adder fa25 (.a(_2PP[14]), .b(_3PP[12]), .c_in(_4PP[10]), .sum(s25), .c_out(c25));
full_adder fa26 (.a(_2PP[15]), .b(_3PP[13]), .c_in(_4PP[11]), .sum(s26), .c_out(c26));
half_adder ha27 (.a(_3PP[14]), .b(_4PP[12]), .sum(s27), .c_out(c27));
half_adder ha28 (.a(_3PP[15]), .b(_4PP[13]), .sum(s28), .c_out(c28));

/* second stage, second addition */
```

```
/* second stage, second addition */
half_adder ha29 (.a(s2), .b(c1), .sum(s29), .c_out(c29));
half_adder ha30 (.a(s3), .b(c2), .sum(s30), .c_out(c30));
half_adder ha31 (.a(s4), .b(c3), .sum(s31), .c_out(c31));
full_adder fa32 (.a(s5), .b(c4), .c_in(_3PP[0]), .sum(s32), .c_out(c32));
full_adder fa33 (.a(s6), .b(c5), .c_in(_3PP[1]), .sum(s33), .c_out(c33));
full_adder fa34 (.a(s7), .b(s8), .c_in(c6), .sum(s34), .c_out(c34));
full_adder fa35 (.a(s9), .b(s10), .c_in(c7), .sum(s35), .c_out(c35));
full_adder fa36 (.a(s11), .b(s12), .c_in(c9), .sum(s36), .c_out(c36));
full_adder fa37 (.a(s13), .b(s14), .c_in(c11), .sum(s37), .c_out(c37));
full_adder fa38 (.a(s15), .b(s16), .c_in(c13), .sum(s38), .c_out(c38));
full_adder fa39 (.a(s17), .b(s18), .c_in(c15), .sum(s39), .c_out(c39));
full_adder fa40 (.a(s19), .b(s20), .c_in(c17), .sum(s40), .c_out(c40));
full_adder fa41 (.a(s21), .b(s22), .c_in(c19), .sum(s41), .c_out(c41));
full_adder fa42 (.a(s23), .b(c21), .c_in(c22), .sum(s42), .c_out(c42));
full_adder fa43 (.a(s24), .b(c23), .c_in(_4PP[9]), .sum(s43), .c_out(c43));
half_adder ha44 (.a(s25), .b(c24), .sum(s44), .c_out(c44));
half_adder ha45 (.a(s26), .b(c25), .sum(s45), .c_out(c45));
half_adder ha46 (.a(s27), .b(c26), .sum(s46), .c_out(c46));
half_adder ha47 (.a(s28), .b(c27), .sum(s47), .c_out(c47));
half_adder ha48 (.a(c28), .b(_4PP[14]), .sum(s48), .c_out(c48));

/* third stage, third addition */
half_adder ha49 (.a(s30), .b(c29), .sum(s49), .c_out(c49));
half_adder ha50 (.a(s31), .b(c30), .sum(s50), .c_out(c50));
half_adder ha51 (.a(s32), .b(c31), .sum(s51), .c_out(c51));
half_adder ha52 (.a(s33), .b(c32), .sum(s52), .c_out(c52));
half_adder ha53 (.a(s34), .b(c33), .sum(s53), .c_out(c53));
full_adder fa54 (.a(s35), .b(c8), .c_in(c34), .sum(s54), .c_out(c54));
full_adder fa55 (.a(s36), .b(c10), .c_in(c35), .sum(s55), .c_out(c55));
full_adder fa56 (.a(s37), .b(c12), .c_in(c36), .sum(s56), .c_out(c56));
full_adder fa57 (.a(s38), .b(c14), .c_in(c37), .sum(s57), .c_out(c57));
full_adder fa58 (.a(s39), .b(c16), .c_in(c38), .sum(s58), .c_out(c58));
full_adder fa59 (.a(s40), .b(c18), .c_in(c39), .sum(s59), .c_out(c59));
full_adder fa60 (.a(s41), .b(c20), .c_in(c40), .sum(s60), .c_out(c60));
full_adder fa61 (.a(s42), .b(_4PP[8]), .c_in(c41), .sum(s61), .c_out(c61));
half_adder ha62 (.a(s43), .b(c42), .sum(s62), .c_out(c62));
half_adder ha63 (.a(s44), .b(c43), .sum(s63), .c_out(c63));
half_adder ha64 (.a(s45), .b(c44), .sum(s64), .c_out(c64));
half_adder ha65 (.a(s46), .b(c45), .sum(s65), .c_out(c65));
half_adder ha66 (.a(s47), .b(c46), .sum(s66), .c_out(c66));
half_adder ha67 (.a(s48), .b(c47), .sum(s67), .c_out(c67));
half_adder ha68 (.a(c48), .b(_4PP[15]), .sum(s68), .c_out(c68));

/* fourth stage, carry loook ahead adder addition */
assign a69 = {1'b0, s68, s67, s66, s65, s64, s63, s62, s61, s60, s59, s58,
              s57, s56, s55, s54, s53, s52, s51, s50};

assign b69 = {c68, c67, c66, c65, c64, c63, c62, c61, c60, c59, c58, c57,
              c56, c55, c54, c53, c52, c51, c50, c49};


carry_look_ahead_adder_20 CLA69 (.a(a69), .b(b69), .c_in(1'b0), .sum(s69), .c_out(c69));

/* fifth stage, final assignments and output concatenation*/

wire [25:0] pre_out;

assign pre_out = {c69, s69, s49, s29, s1, _0PP[1], _0PP[0]};

assign out = pre_out[7:0];

endmodule
```

Notice at the end of this wallace addition module, we assign our final multiplication value, which is passed to *fast_multipler.v*, then adjusted if the product is a negative 2's compliment value, and finally passed all the way back to *calculator.*v to be used as a result.

full_adder.v, half_adder.v, carry_look_ahead_adder_20.v

In *wallace_addition.v* we implemented over 60 distinct adders to properly represent the algorithm. The module mainly comprises a full adder sub-module and a half_adder submodule. Both of which are straightforward and can be seen down below.

full_adder.v

```
1   module full_adder(a, b, c_in, sum, c_out);
2
3   input a, b, c_in;
4   output sum, c_out;
5
6   assign {c_out, sum} = a + b + c_in;
7
8   endmodule
```

half_adder.v

```
1   module half_adder(a, b, sum, c_out);
2
3   input a, b;
4   output sum, c_out;
5
6   assign {c_out, sum} = a + b;
7
8   endmodule
```

These were instanstiated a multitude of times to gain the desired effect.

We also implemented a 20-bit carry look ahead adder for the fourth stage of our addition. This carry look ahead adder is no different from out 8-bit CLA adders from a hardware description language standpoint, as I simply just changed the width parameter of the adder from 8 to 20.

Shift Resgister (Implements 2^y)

As mentioned earlier, the expontential function is the only one of the four operations that did not need a distinct submodule. This is because it fits rather compactly in our *calculator.v* top-level module. The section that implements the operation is as follows:

```
/* Exponent operation using shift operator */
wire [7:0] exp;
assign exp = 1'b1 << y;
```

Here we see that the wire *exp* is 8-bits wide, to accomplish 8-bits of accuracy at the output. In pratice however, this means only the first 7 shifts will be accurate 100% of the time, as an operation of 2^8 (8 shifts) would overload the register and force it to display zeroes due to the 1-bit bit being shifted off the left end of the MSB. Due to this, the exponential operation can only obtain a respectable maximum value of 128.

Testing:

While all submodules were tested separatetly, I will only be showing the testing of the top-level module. I also encourage you to modify my testbench and test it yourself, as I will only be showing a single testbench in this report. The testbench is as follows:

calculator_tb.v

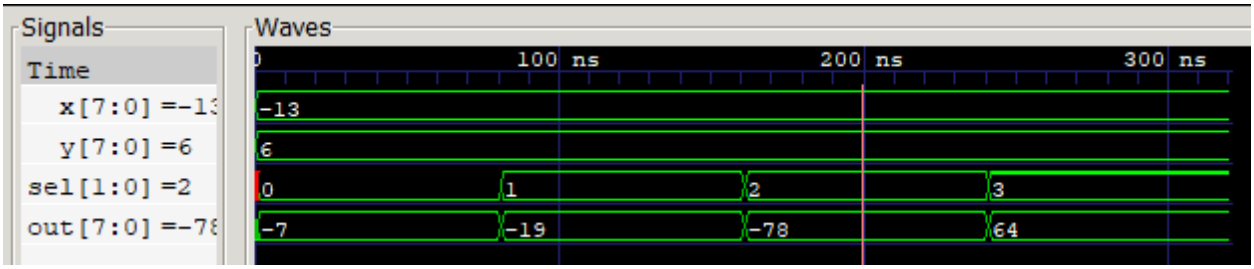
```
1   `timescale 1ns/1ps
2   `include "./calculator.v"
3
4   module calculator_tb();
5
6   reg signed [7:0] x, y;
7   reg [1:0] sel;
8
9   wire signed [7:0] out;
10
11  calculator dut (.x(x), .y(y), .sel(sel), .out(out));
12
13
14  initial begin
15
16  x = -13;
17  y = 6;
18  #1
19  sel = 0;
20  #80
21  sel = 1;
22  #80
23  sel = 2;
24  #80
25  sel = 3;
26  #80
27  $finish;
28
29  end
30
31  initial begin
32      $dumpfile("calculator.vcd");
33      $dumpvars;
34  end
35
36  endmodule
```

In this testbench, we first set x and y to -13 and 6 respectively. Then after a period of 1ns then intervals of 80ns, we initiate and increment our select bit until we reach select = 3, and end the simulation 80ns later.

In theory, this means that our generated waveform should show us our x and y bits be unchanged for the entire duration of the test, but we cycle through which operation we perform with said bits.

Particularly, we should see our output start with addition ((-13)+6), then subtraction ((-13)-6), then multiplication ((-13)*6), and finally exponential (2^6).

And in practice this is exactly what we see:



So our calculuator seems to be working as intended (albeit with the limitations mentioned throughout this report). (PLEASE BE CAREFUL TO MAKE NOTE OF OVERFLOW

WHEN TESTING YOURSELF. OUTPUT WILL ONLY BE CORRECT IS FINAL ANSWER CAN FIT INTO FINAL BUS WIDTH OF 8-BITS).

Potential Improvements:

- While the calculator is working, there are still some things can be improved on. For instance, in *wallace_addition.v*, we do a lot of uneccessary addition and calculate many zero bits that will never make it into the final answer due to the necessary truncation. Initially, I included so many leading zeroes in in this module to make certain I would have enough sign-extension bits if I needed them, but instead it ended up causing the calulator to be much more complex and consuming more power than necessary. Cutting down on this wallace addition module would be a huge improvement. We couldve also made use of generate statements to implement the module, which might have saved a lot of time had I been able to figure out how to properly implement the alternating half and full adders required for the algorithm.
- We could have also implemented a feature that allows the user to select which of the 8-bit inputs to use in the exponential calculation.
- We could have improved precision of the calculator, to allow for bigger operations. Though this is in fact an 8-bit caculator with an 8-bit output, to improve answer accuracy I couldve made the output bus width larger.
- Potentially add an overflow indication bit

Conclusion:

We were able to develop a calculator that is able to perform basic operations. Since it's built on carry look ahead adders, booth recoding, and wallace addition, it should be one of the faster calculators you can build using only basic methods. It was very engaging to work on this project and interesting to work through the initial issues!

Acknowledgements:

I would like to acknowledge the user *Vipin* on the Verilogcodes blog, who provided me with inspiration on how to tackle the multiplication aspect of the calculator. You can find his blog post in the references section.

References:

➔ Bo Yuan, Lecture 8 and 9 CAD VLSI.
➔ User Vipin, Verilogcodes blog. https://verilogcodes.blogspot.com/2015/11/verilog-code-for-4-bit-wallace-tree.html

- GATEBOOK Video Lectures, YouTube. https://www.youtube.com/watch?v=9c6JLHP3rwQ
- Adi Teman, YouTube. https://www.youtube.com/watch?v=4FwESTOVT-o
- Collaborative, Wikipedia. https://en.wikipedia.org/wiki/Wallace_tree