Performance Comparison Of Booth's Multiplier and Serial-Parallel Multiplier on Xilinx FPGA

EE2016 Microprocessors

Sanjeev Subrahmaniyan S B

EE23B102

Abstract

Booth's Multiplier is an integral part of a microprocessor due to its ability to perform 2s complement multiplication with greater speed and efficiency than other multiplication algorithms. In this lab experiment, a comparison of the performance of a Booth's Multiplier against the Serial-Parallel Multiplier is made using Verilog HDL. The HDL program is run on an EDGE 7 Artix Xilinx FPGA board and simulated on the Xilinx Vivado software. The performance is compared in terms of the number of clock cycles taken for the execution. The number of clock cycles taken as a function of the operand size is also plotted. It is observed that the Booth's Multiplier is faster in taking lesser number of clock cycles to perform the multiplication.

1 Introduction

Present day microprocessors have specialised hardware to execute multiplication arithmetic at the highest speeds. The most commonly used architectural implementation of multiplication of 2s complement numbers is called the Booth's algorithm. Booths algorithm works by intelligently handling the shift and add method that is commonly used in pen-paper multiplication to reduce the number of hardware operations performed(here, addition and shifting).
As an example, consider the multiplication of two 4-bit numbers 3 (0011) and 7 (0111). While the shift-and-add multiplication would proceed as:

$$3 \times (2^2 + 2^1 + 2^0)$$

Booth's algorithm recognizes this as:

$$3 \times (2^3 - 2^0)$$

In the first method, this would require 4 shifts and 3 additions for a total of 7 cycles. However, the second method using Booth's algorithm only requires 2 additions (1 addition and 1 subtraction) and 4 shifts, thus making it a more efficient implementation as the sizes of the operands increase. The booth's multiplier is typically implemented to perform addition and shift together in the same cycle, requiring a total of only 4 clock cycles(excluding the loading cycle). This difference in cycles taken is more pronounced as the size of the operands increase.

A FPGA is chosen to perform the experiment as they are tailor-made for prototyping and emulation of circuits. They are of immense practical value in circuit designing and analysis, and are hence ideal for this purpose.

2 Objectives

A method to compare performance of algorithms on the same hardware is to count the number of clock cycles taken to execute the same program with the same inputs. In this experiment, the algorithms are compared by being executed on a Xilinx FPGA programmed using Verilog HDL. The objectives of this lab experiment is broadly to:

1. Understand the hardware level working of Booth's and Serial-Parallel multiplier.
2. Implement the algorithms in Verilog HDL and simulate them by writing appropriate test benches.
3. Configure and program a Xilinx FPGA using constraints and run the programs and measure the performance by counting the number of cycles taken. Knowledge of writing constraints file for an FPGA is gained.
4. Compare the algorithms for different operand sizes and appreciate the difference in performance.

3 Verilog HDL Programs

3.1 Booth's Multiplier

3.1.1 HDL Program

```
module multiplier(prod, busy, mc, mp, clk, start);
output [7:0] prod;
output busy;
input [3:0] mc, mp;
input clk, start;
reg [3:0] A, Q, M;
reg Q_1;
reg [2:0] count;
reg [7:0] clock_count;
wire [3:0] sum, difference;
initial
begin
clock_count <= 0;
end

assign busy = (count < 4); //essentially finish
assign prod = {A, Q}; // make it fill up the arguments

wire a1, a2;
assign a1 = 1'b0;
assign a2 = 1'b1;
always @(posedge clk)
    begin
    if (start)
    begin
        clock_count <= 1;
        A <= 4'b0000;
        M <= mc;
        Q <= mp;
        Q_1 <= 0  ; // bit written to the left of lsb of number to
            be multiplied
        count <= 0;
    end
    else if(busy)
    begin
        clock_count <= clock_count + 1;
        case ({Q[0], Q_1})
            2'b01 : {A, Q, Q_1} <= {sum[3], sum, Q};
            2'b10 : {A, Q, Q_1} <= {difference[3], difference, Q};
            default: {A, Q, Q_1} <= {A[3], A, Q};
        endcase
        count <= count + 1'b1;
        end
    end
```

```
alu adder(.out(sum), .a(A) , .b(M) , .cin(a1)); // named port
    connection
alu subtracter(.out(difference) , .a(A) , .b(~M), .cin(1'b1));
endmodule

module alu(out, a, b, cin);
output [3:0] out;
input [3:0] a;
input [3:0] b;
input cin;
assign out = a + b + cin;
endmodule
```

In the Verilog program given above, we implement the Booth's algorithm for four bit numbers. It will be seen from the test bench that the performance of this algorithm is better than the following serial parallel multiplier.

3.1.2 Testbench/stimulus for simulation

```
module stimulus;

reg [3:0] Multiplier, Multiplicand;
wire [7:0] Output;
wire busy;
reg clk;
reg start;
initial
begin
    clk <= 1'b1;
end

initial
begin
    $dumpfile("serial_multiplier.vcd"); //Used to create a dump which can be
    $dumpvars(0, stimulus);
end

always@(*)
begin
    #1 clk <= ~clk;
end

multiplier mult(.prod(Output), .busy(busy), .mc(Multiplicand), .mp(Multiplier
initial
begin
    start = 1;
    Multiplier = 4'b0010; Multiplicand = 4'b0011;
    #2 start = 0;
    wait(!busy);
    #4 $finish;
end
endmodule
```

3.2 Serial Parallel Multiplier

3.2.1 HDL Program

```
module mult_4x4(
input reset,clk,
input [3:0] A,B,
output reg [7:0] O, // 8-bits output
output Finish, output reg [7:0] count);
reg [3:0] State; // state machine
reg [8:0] ACC; // Accumulator
initial
begin
  count = 0;
end

// logic to create 2 phase clocking when starting
assign Finish = (State === 0)? 1 : 0; // Finish Flag
always@(posedge clk)
  begin
    count <= count + 1;
    if(reset)
      begin
        count <= 0;
        State <= 0;
        ACC <= 0;
        O <= 0;
      end
    else if (State==0)
      begin
        ACC[8:4] <= 8'b0; // begin cycle
        ACC[3:0] <= A; // Load A (one of our inputs)
        State <= 1;
      end
    else if (State==1 || State == 3 || State == 5 || State == 7)
      // add/shift State
      begin
        if(ACC[0] == 1'b1)
          begin // add multiplicand
            ACC[8:4] <= {1'b0,ACC[7:4]} + B;
            State <= State + 1;
          end
        else
          begin
            ACC <= {1'b0, ACC[8:1]}; // shift right
            State <= State + 2;
          end
        end

        else if(State == 2 || State == 4 || State == 6 || State ==
            8)
          // shift State
          begin
            ACC <= {1'b0,ACC[8:1]}; // shift right
            State <= State + 1;
          end
        else if(State == 9 ) // What state for end?
          begin
          //State <= 0;
          O = ACC[7:0]; // loading data of accumulator in output
          State = 0;
          end
      end
endmodule
```

Here, the approach followed is to divide the operation into two fundamental steps : addition and shifting or shifting alone. This is executed by using even and odd valued states as implemented in the program above. The following testbench is used in the simulation of the design, with functionality for using a dump file to visualize the plots in a software like gtkwave.

3.2.2 Testbench/stimulus

```
module stimulus;

  reg [3:0] Multiplier, Multiplicand;
  wire [7:0] Output;
  reg clk;
  reg reset;
  wire Finish;
  wire [7:0] count;

  mult_4x4 multiplier(.reset(reset), .clk(clk), .A(Multiplier), .B
     (Multiplicand), .O(Output), .Finish(Finish), .count(count));

  initial
  begin
    clk = 1'b1;
  end

  always@(*)
    #1 clk <= ~clk;

  initial
  begin
    $dumpfile("mult_4x4.vcd");
    $dumpvars(0, stimulus);
  end

  initial
  begin
    reset = 1;
    wait(Finish);
    reset = 0; Multiplicand = 4'b0000; Multiplier = 4'b0000;
    wait(Finish);

    #2 reset = 1;
    #2 reset = 0; Multiplicand = 4'b0010; Multiplier = 4'b0110;
    #2;
    wait(Finish) #2 $display("The number of clock cycles (
       including loading cycle) is: %d \n", count); $finish; //
       Minus one because the final answer is available at the
       start of the cycle itself
  end
endmodule
```

4 Procedure

The experiment was performed in the following steps:

1. Study the hardware level implementation of the two algorithms.
2. Write Verilog code to emulate the operation of Serial-Parallel multiplier and its corresponding test bench.
3. Simulate and verify the Verilog code using iverilog and gtkwave.
4. Repeat the same for Booth's multiplier.
5. Configure the Xilinx Vivado devlopment lab for the edge Artix 7 board and add the verilog files and testbenches.
6. Add design constraints to specify the mapping between top level module ports in the program to lines on the FPGA.
7. Simulate the programs on Vivado lab.
8. Interface the Artix 7 board to the computer and run the programs on the board, and collect data corresponding to different input values and sizes.
9. Record the values and plot them.
10. Understand the data, make inferences and present them in the report.

5 Interpretation of results

The Verilog programs and test benches presented above implement the multiplication of the numbers 2(0010) and 3(0011) on the two different architectures of the multipliers. The corresponding timing plots from the simulator on Vivado are shown below:

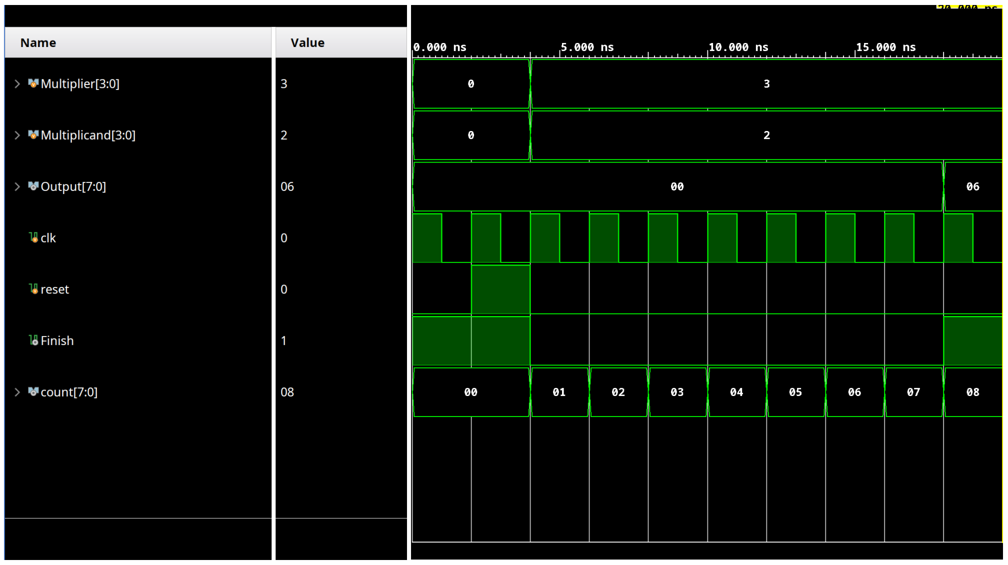

Figure 1: Timing diagram for the multiplication on a serial parallel multiplier

For the serial parallel multiplier, the registers are cleared and initialised in the first two clock cycles. Then the values are loaded into the Multiplier and Multiplicand registers, along with the release of the reset pin. This starts the multiplication as seen by the change in the value of Finish. After 7 cycles, the value of the output changes to 6 and the Finish variable indicates the completion of the operation.

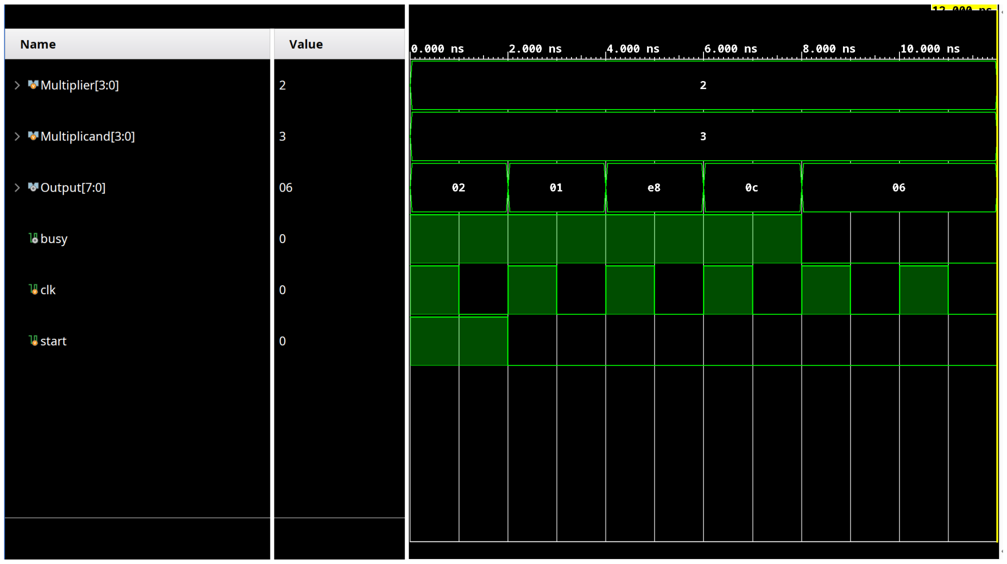

Figure 2: Timing diagram for the multiplication on a Booth's multiplier

Similarly, for the Booth's multiplier, the input values are loaded and the start of the process is triggered by the positive edge of the start variable. Then, after 4 clock cycles, the value of the output has changed to the true output of 6. As mentioned earlier, the Booth's multiplier handles the operands in an intelligent way and is able to make the computations faster. As seen in the next plot, the difference in the number of clock cycles taken also increase as the size of the operands increase. This means an implementation of Booth's algorithm can be many times faster for a practical application.

The decrease in the number of clock periods taken by the Booth's multiplier makes it a very fast implementation for multiplication on a computer. The figure 3 on the next page shows the number of clock cycles each of the method takes for different operand sizes. **This plot was generated by using the parameter feature in Verilog and generalizing the program to work for an arbitrary operand width**. For example,

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

module booth_multiplier #(
  parameter WIDTH = 4)(output wire [2*WIDTH - 1:0] product, input
    [WIDTH - 1:0] Multiplier, Multiplicand, input clk, start,
    output finish, output reg[2*WIDTH - 1:0] clock_count);

reg [WIDTH - 1:0] A;
reg [WIDTH - 1:0] M;
reg [WIDTH - 1:0] Q;
reg Q1;
integer count;
wire [WIDTH - 1:0] sum, diff;
always@(posedge clk)
begin
  if(start)
  begin
          A <= 0;
          Q1 <= 0;
          M <= Multiplicand;
          Q <= Multiplier;
    clock_count <= 1;
    count = WIDTH;
  end
  else if(!finish)
  //else
  begin
    clock_count <= clock_count + 1;
    case({Q[0], Q1})
      2'b01: {A, Q, Q1} <= {sum[WIDTH - 1], sum[WIDTH - 1:0], Q[
         WIDTH - 1:0]};
      2'b10: {A, Q, Q1} <= {diff[WIDTH - 1], diff[WIDTH - 1:0], Q[
         WIDTH - 1:0]};
      default: {A, Q, Q1} <= {A[WIDTH - 1], A, Q};
    endcase
    count = count - 1;
  end
end

alu #(.WIDTH(WIDTH))adder(sum, A, M, 1'b0);
alu #(.WIDTH(WIDTH))subtractor(diff, A, ~M, 1'b1);

assign product = {A, Q};
assign finish = (count == 0) ? 1 : 0;

endmodule
```

The code above is the parameter program used to vary the operand width for the plotting. The operand width can be specified in the test bench to run for different sizes.

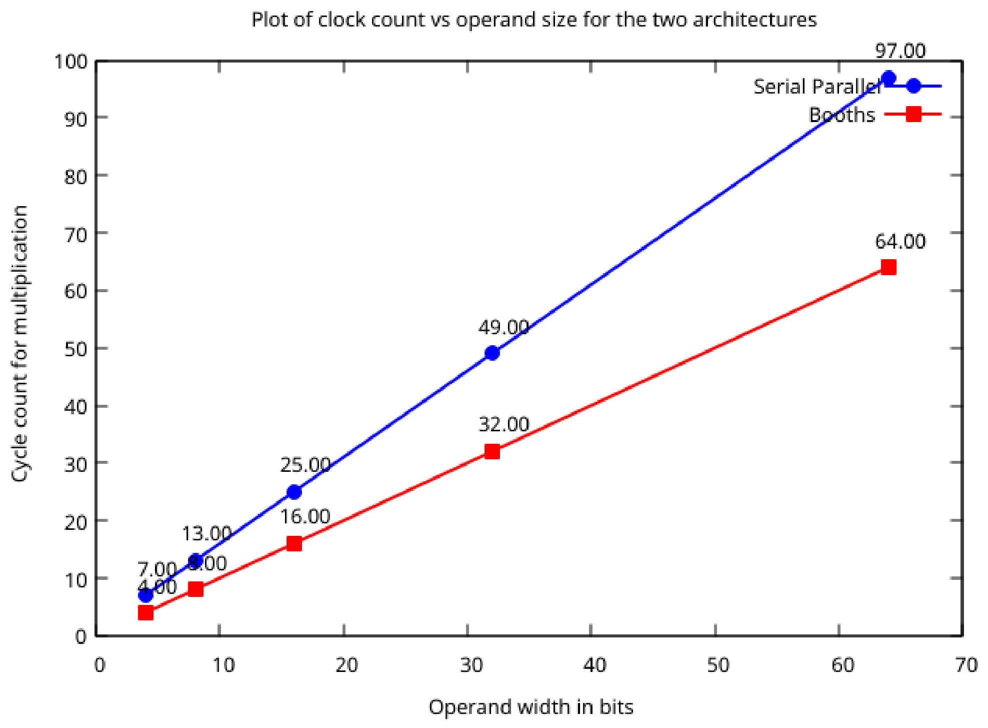

Figure 3: Comparison of cycle counts for the two architectures

6 My contribution to the experiment

1. Wrote the verilog program for the serial parallel multiplier and wrote the test benches for both the programs.
2. Helped configure the Vivado board by adapting the master xdc file.
3. Wrote the parameter based generalized Verilog code to generate the cycle vs size relationship.
4. I was also actively involved in debugging the code and troubleshooting the implementation on hardware.

7 Conclusion

This experiment lets us observe the differences in implementation and performance of the two algorithms. Such comparisons and studies are integral to designing efficient and fast computer hardware and software, which provide valuable insights to engineers to choose the best for an application. It is often the case that one implementation is not always the best.

While multiplication in a microprocessor can be implemented with a number of different architectural and organizational methods, the Booth's algorithm stands out due to its reduction in number of operations and support for 2s complement numbers. This makes it the ideal application where resource and performance optimization is crucial. By understanding these differences, one can appreciate the trade-offs in developing hardware/software for an application.