Tutorial

contents

1. Installation of verilog essentials
2. An example using the simple counter
    a. Subtle Nuances in the code
    b. GTKwave and vcd dump
3. Go to combinational karatsuba and show how the data path has been written
4. Go to sequential karatsuba and show how the data path has been written

Installation guide

Read the readme file provided.

Already provided in piazza

Just install iverilog and gtkwave.

```
module simple_counter(clk, rst, enable, out);
    input clk;
    input rst;
    input enable;

    output reg out;

    reg [1:0] state, nxt_state;
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;

    always @(posedge clk) begin
        if (rst) begin
            state <= S0;
        end
        else if (enable) begin
            state <= nxt_state;
        end
    end
```

Simple Counter - Sequential Example

What are regs, wires, parameters ?

What is the difference between reg and wire ?

What is the difference between sequential and combinational circuits ?

Why is a simple counter a sequential circuit ?

```
    always@(*) begin
        case(state)
            S0:
                begin
                    nxt_state = S1;
                    out = 0;
                end
            S1:
                begin
                    nxt_state = S2;
                    out = 0;
                end
            S2:
                begin
                    nxt_state = S3;
                    out = 0;
                end
            S3:
                begin
                    nxt_state = S0;
                    out = 1;
                end
            default:
                begin
                    nxt_state = S0;
                    out = 0;
                end
        endcase
    end
endmodule
```

What is an always loop ?

Types of assign statements

- Blocking assignment
- Non Blocking assignment

Which type assignment is being used here ?

```
`timescale 1ns/1ps
module tb_simple_counter;

reg clk;
reg rst;
reg enable;
wire out;

initial begin
    $display("time\t, clk\t, rst\t, enable\t, out\t ");
    $monitor ("%g\t %b\t %b\t %b\t %b\t", $time, clk, rst, enable, out);

    clk = 1;
    rst = 0;

    #10 rst = 1;
    #10 rst = 0;
    #10 enable  = 1;


    #150
    #5 $finish;
end

always begin
    #5 clk = ~clk;
end

simple_counter dut(.clk(clk), .rst(rst), .enable(enable), .out(out));

initial begin
    $dumpfile("simple_counter.vcd");
    $dumpvars(0,tb_simple_counter);
end

endmodule
```

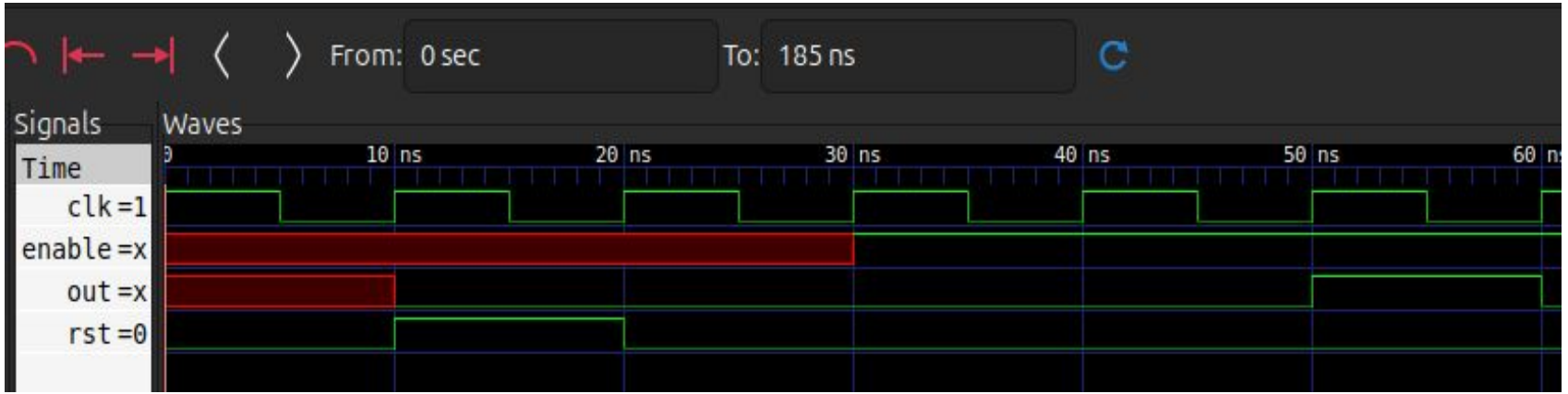
Testbench

What is timescale ?

How to initialize the fields ?

What does #10 mean?

GTK-Wave



Compare the output with testbench initialization ?

Adders - Combinational Example

```
module half_adder(a, b, S, cout);
input a;
input b;

output S;
output cout;

xor(S, a, b);
and(cout, a, b);

endmodule
```

```
module full_adder(a, b, cin, S, cout);
input a;
input b;
input cin;
output S;
output cout;

assign S = a ^ b ^ cin;
assign cout = (a&b) ^ (b&cin) ^ (a&cin);

endmodule
```

Ripple Carry Adder

```
module full_adder_with_ha (a, b, cin, S, cout);
input a;
input b;
input cin;
output S;
output cout;

wire w1;
wire w2;
wire w3;

half_adder ha1(.a(a), .b(b), .S(w1), .cout(w2) );
half_adder ha2(.a(w1), .b(cin), .S(S), .cout(w3) );
or(cout, w3, w2);

endmodule
```

```
/* 4-bit RCA adder */

module adder_4bit(a, b, cin, S, cout);
input [3:0] a;
input [3:0] b;
input cin;
output [3:0] S;
output cout;

wire c1;
wire c2;
wire c3;

full_adder add1 (.a(a[0]), .b(b[0]), .cin(cin), .S(S[0]), .cout(c1));
full_adder add2 (.a(a[1]), .b(b[1]), .cin(c1), .S(S[1]), .cout(c2));
full_adder add3 (.a(a[2]), .b(b[2]), .cin(c2), .S(S[2]), .cout(c3));
full_adder add4 (.a(a[3]), .b(b[3]), .cin(c3), .S(S[3]), .cout(cout));
```

Parameterization

```
/* N-bit RCA adder (Unsigned) */
module adder_Nbit #(parameter N = 32) (a, b, cin, S, cout);
input [N-1:0] a;
input [N-1:0] b;
input cin;
output [N-1:0] S;
output cout;

wire [N:0] cr;

assign cr[0] = cin;


generate
    genvar i;
    for (i = 0; i < N; i = i + 1) begin
        full_adder addi (.a(a[i]), .b(b[i]), .cin(cr[i]), .S(S[i]), .cout(cr[i+1]));
    end
endgenerate


assign cout = cr[N];

endmodule
```

parameters in verilog is nothing but a way to instantiate constants in the digital design.

`parameter` is local to a module. but it can change value when the module is instantiated. It is used to define a property of the module.

Combinational karatsuba

Naive N bit multiplication needs **O(N^2)** partial products

Large multipliers dominate **area, power, and delay**

Hence, We try to **reduce** the number of partial products by using the **divide-and-conquer** method.

Combinational karatsuba

$$
\begin{aligned}
(A\_H << N/2 + A\_L) * (B\_H << N/2 + B\_L) = \\
A\_H * B\_H << N \quad + \\
A\_L \ * B\_H << N/2 + \\
A\_H * B\_L \ << N/2 + \\
A\_L \ * B\_L
\end{aligned}
$$

Takes 4 multiplications, but We can optimize this as,

$$
\begin{aligned}
(A\_L * B\_H + A\_H * B\_L) = (A\_H + A\_L)*(B\_H + B\_L) \\
- (A\_H * B\_H) \\
- (A\_L \ * B\_L)
\end{aligned}
$$

We see that this is just 3 multiplications now, but one of the multiplications is a N/2+1 bit multiplication

Combinational karatsuba

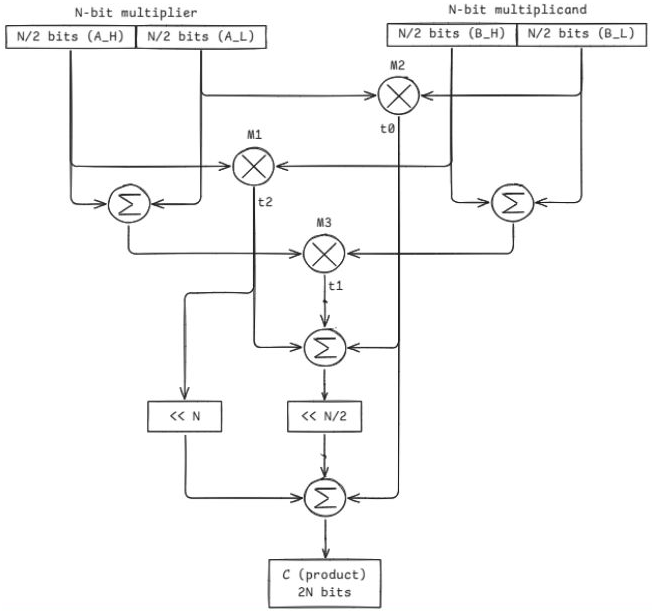
This N/2 + 1 bit multiplier can be split into a N/2 multiplier and 2 adders

$$aA * bB = ab << N + A*B + (b\&A + a\&B) << N/2$$ (& ands all bits)

Now, we can implement these also as karatsuba multipliers ,with the base case as N=2.

This is now $\mathbf{O(N^{\log(3)})}$ partial products

Combinational karatsuba



Combinational karatsuba

```
module karatsuba_16(A,B,out);
input [15:0] A;
input [15:0] B;
output [31:0] out;

//split the numbers
wire [7:0]A_L;
assign A_L[7:0] = A[7:0];
wire [7:0]A_H ;
assign A_H[7:0] = A[15:8];
wire [7:0]B_L;
assign B_L[7:0] = B[7:0];
wire [7:0]B_H;
assign B_H[7:0] = B[15:8];
```

Combinational karatsuba

```
wire [15:0]mid;
karatsuba8 mult0 (A_L,B_L,mid[15:0]);
wire [15:0]mid1;
karatsuba8 mult1 (A_H,B_H,mid1[15:0]);
wire posval;
wire negval;
assign posval = 1;
assign negval = 0;
wire [8:0]AHALsum;
wire [8:0]BHBLsum;
rca_Nbit #(8) AHALsum (A_L,A_H,negval,AHALsum[7:0],AHALsum[8]);
rca_Nbit #(8) BHBLsum (B_H,B_L,negval,BHBLsum[7:0],BHBLsum[8]);
```

Combinational karatsuba

```
wire [15:0]mid;
karatsuba8 mult0 (A_L,B_L,mid[15:0]);
wire [15:0]mid1;
karatsuba8 mult1 (A_H,B_H,mid1[15:0]);
wire posval;
wire negval;
assign posval = 1;
assign negval = 0;
wire [8:0]AHALsum;
wire [8:0]BHBLsum;
rca_Nbit #(8) AHALsum (A_L,A_H,negval,AHALsum[7:0],AHALsum[8]);
rca_Nbit #(8) BHBLsum (B_H,B_L,negval,BHBLsum[7:0],BHBLsum[8]);
```

Notice that the sum registers are 9 bits long

Combinational karatsuba

```
wire [17:0]mid2;
wire [7:0]mid3;
wire [7:0]mid4;
generate
    genvar i;
    for(i=0;i<8;i=i+1)begin
        assign mid3[i] = AHALsum[8] & BHBLsum[i];
        assign mid4[i] = BHBLsum[8] & AHALsum[i];
    end
endgenerate
wire [8:0]mid5;
rca_Nbit #(8) midl1 (mid3,mid4,negval,mid5[7:0],mid5[8]);
wire [15:0]mid6;
karatsuba8 midl (AHALsum[7:0],BHBLsum[7:0],mid6[15:0]);
assign mid2[7:0] = mid6[7:0];
rca_Nbit #(8) midl2(mid5[7:0],mid6[15:8],negval,mid2[15:8],carryover);
wire AHALsumBHBLsum;
assign AHALsumBHBLsum = AHALsum[8] & BHBLsum[8];
full_adder midl3(carryover,mid5[8],AHALsumBHBLsum,mid2[16],mid2[17]); //mid2 has (x0+x1)(y0+y1)
```

Combinational karatsuba

```
wire [17:0]mid7;
rca_Nbit #(16) midl4 (mid,mid1,negval,mid7[15:0],mid7[16]);
assign mid7[17] = 1'b0;
wire [17:0]mid8;
assign mid8 = ~mid7;
wire [17:0]mid9;
rca_Nbit #(18) subr (mid8,mid2,posval,mid9,carryign); //mid9 has x0y1 + x1y0

assign out[7:0] = mid[7:0];
wire [23:0]ends;
assign ends[7:0] = mid[15:8];
assign ends[23:8] = mid1[15:0];
wire [23:0]mid10;
assign mid10[17:0] = mid9[17:0];
assign mid10[23:18] = 6'h00;
rca_Nbit #(24) finaladd (ends,mid10,negval,out[31:8],carryign2); //doneee!


endmodule
```