Pipelined 3 Level Unsigned Multiplier

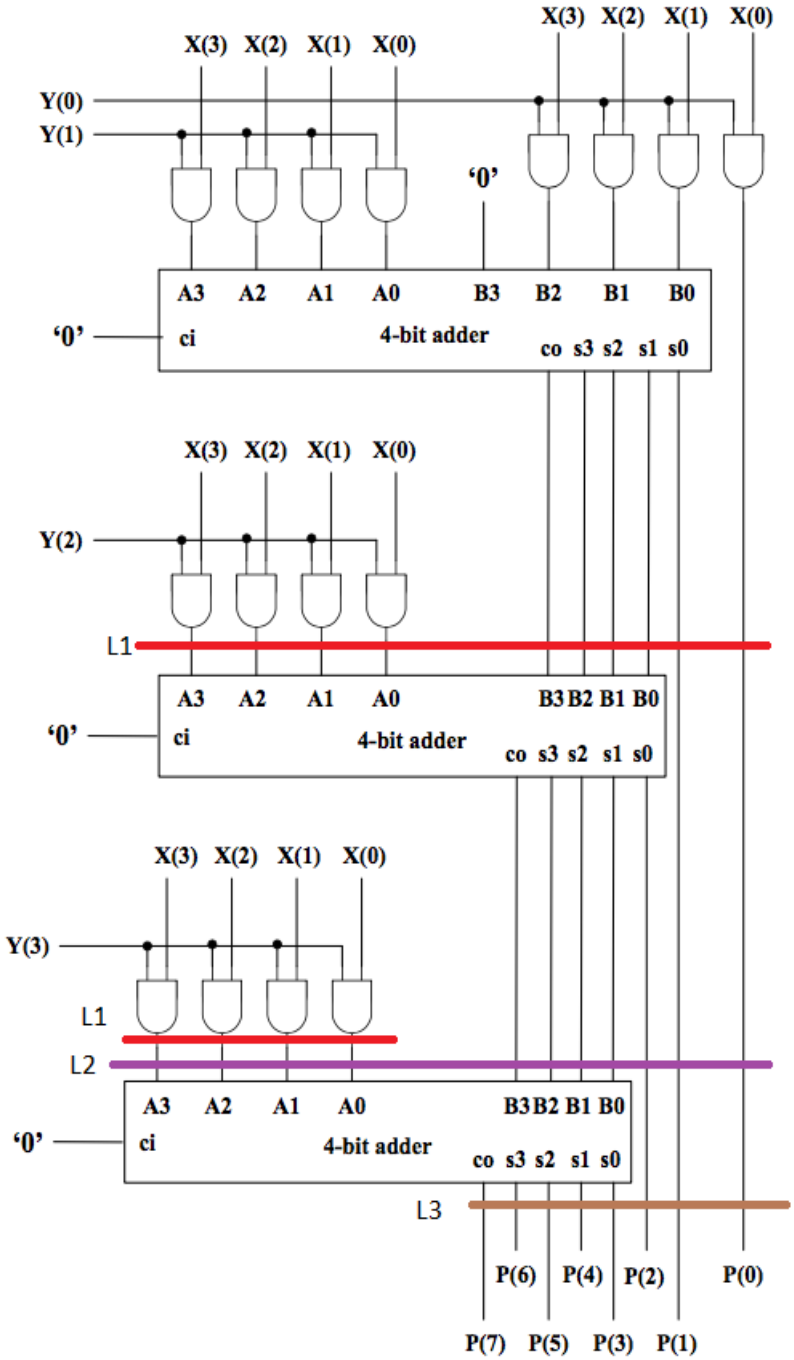
Description-

Pipelining a combinational logic involves putting levels of registers in the logic to introduce parallelism and, as a result, improve speed. Flip flops introduced by pipelining typically incur a minimum of additional area on FPGAs, by occupying the unused flip flops within logic cells that are already used for implementing combinational logic in the design.

Multiplication is one of the mostly used operations in all of the devices. This paper presents an efficient implementation of a pipelined multiplier designed with two stage pipelining. A range of multipliers architectures are available based on applications. The pipelined circuits are designed so that we can achieve high performance for a system as they can be operated at higher frequency. From the implementation results, it is verified that two stage pipelined circuit is faster by a factor of two times than the non-pipelined circuit. But we get an area and power overhead. Therefore, this circuit can be used where the performance of system is of major concern.

In the block diagram, L1, L2 and L3 represents the pipeline registers introduced. The three 4-bit adders are implemented using a 1-bit half adder (since all the 4 bit adders have no carry in) and three 1-bit full adders. From the Verilog code, simulation results are obtained. It can be inferred from the waveforms that multiplier output is obtained after two clock cycles (for eg.,1st result got at 3rd clock pulse) as a result of 3-level pipelining.

Block Diagram



Code:-

```
module pipelined_multiplier(x,y,out,clk);
input [3:0] x,y;                                    //4-bit input multiplier
input clk;
output [7:0] out;                                   // 8 bit output
wire [3:0]PP1,PP2,PP3,PP4;                          //Partial products
wire [3:0]s1,s2,s3;
wire c1,c2,c3;

//For pipelining
reg L1_13,L1_12,L1_11,L1_10,L1_9,L1_8,L1_7,L1_6,L1_5,L1_4,L1_3,L1_2,L1_1, L1_0;
//latches for storing temperory values

reg L2_10,L2_9,L2_8,L2_7,L2_6,L2_5,L2_4,L2_3,L2_2,L2_1,L2_0;
//latches for storing temperory values

reg L3_7,L3_6,L3_5,L3_4,L3_3,L3_2,L3_1,L3_0;
//latches for storing temperory values

//Partial products
assign PP1 = x & {4{y[0]}};              //replication for taking bitwise AND operation
assign PP2 = x & {4{y[1]}};             //replication for taking bitwise AND operation
assign PP3 = x & {4{y[2]}};            //replication for taking bitwise AND operation
assign PP4 = x & {4{y[3]}};           //replication for taking bitwise AND operation

RCA A1({1'b0,PP1[3:1]},PP2,s1,c1);     //(a,b,sum,cout)                  //concatenation
RCA A2({L1_5,L1_4,L1_3,L1_2},{L1_9,L1_8,L1_7,L1_6},s2,c2);             //concatenation
RCA A3({L2_10,L2_9,L2_8,L2_7},{L2_6,L2_5,L2_4,L2_3},s3,c3);           //concatenation

assign z = {L3_7,L3_6,L3_5,L3_4,L3_3,L3_2,L3_1,L3_0};

always @(posedge(clk))                              //stage 1
{L1_13,L1_12,L1_11,L1_10,L1_9,L1_8,L1_7,L1_6,L1_5,L1_4,L1_3,L1_2,L1_1,L1_0} <=
{PP4,PP3,c1,s1,PP1[0]};

always @(posedge(clk))                             //stage 2
{L2_10,L2_9,L2_8,L2_7,L2_6,L2_5,L2_4,L2_3,L2_2,L2_1,L2_0} <=
{L1_13,L1_12,L1_11,L1_10,c2,s2,L1_1,L1_0};

always @(posedge(clk))                            //stage 3
{L3_7,L3_6,L3_5,L3_4,L3_3,L3_2,L3_1,L3_0} <= {c3,s3,L2_2,L2_1,L2_0};
endmodule
```

```
//4 bit Ripple Carry Adder module using 1HA and 3FA
module RCA(a,b,sum,cout);
input [3:0]a,b;
output [3:0]sum;
output cout;
wire t1,t2,t3;
Halfadder HA(a[0],b[0],sum[0],t1);                    //(a,b,sum,cout)
Fulladder FA1(a[1],b[1],t1,sum[1],t2);                //(a,b,cin,sum,cout)
Fulladder FA2(a[2],b[2],t2,sum[2],t3);                //(a,b,cin,sum,cout)
Fulladder FA3(a[3],b[3],t3,sum[3],cout);              //(a,b,cin,sum,cout)
endmodule

module Fulladder(a,b,cin,sum,cout);
 input  a,b,cin;
output sum,cout;
wire   w1,w2,w3;
Halfadder HA1(a,b,w1,w2);                             //(a,b,sum,cout)
Halfadder HA2(w1,cin,sum,w3);
assign cout=w2|w3;
endmodule


module Halfadder(a,b,sum,cout);
 input  a,b;
output sum,cout;
 assign sum=a^b;
assign cout=a&b;
endmodule
```

Test Bench:-

```
`timescale 1ns/1ns
`include "pipelined_multiplier.v"

module pipelined_tb;

// Inputs
reg [3:0] x;
reg [3:0] y;
reg clk;
// Outputs
wire [7:0] out;
// Instantiate the Unit Under Test (UUT)
pipelined_multiplier uut (.x(x),.y(y),.out(out),.clk(clk));

initial begin
   $dumpfile("pipelined_tb.vcd");
   $dumpvars(0,pipelined_tb);
// Initialize Inputs
clk = 0;
x=3;y=2;
#10 x=8;y=5;
#10 x=2;y=8;
#10 x=9;y=3;
#10 x=6;y=8;
#10 x=7;y=5;
#10 x=2;y=9;
#10 x=6;y=4;
#30 $finish;
end
always #5 clk=~clk;
endmodule
```

Simulation Results:-