4x4 Multiplier using CLA logic

ADDC Open-ended experiment

By: Seshasai Chillara 1RV23ET040

Introduction

In a traditional multiplier, partial products are generated by performing AND operations between bits of the multiplicand and multiplier. These partial products are then summed in multiple stages using adders, typically resulting in a large delay. The Wallace Tree approach tackles this by restructuring the summation process into parallel stages, reducing the overall addition levels and speeding up the operation.

1. **Partial Product Generation**:
    - For a 4x4 multiplier, 16 partial products are generated using AND gates. Each bit of the 4-bit multiplicand is ANDed with each bit of the 4-bit multiplier.
2. **Reduction Stages**:
    - The partial products are arranged in columns based on their bit significance.
    - Full adders and half adders are used to reduce the number of bits in each column while preserving the total value.
    - This reduction is done in stages, with each stage producing fewer rows of bits.
3. **Final Addition**:
    - Once the partial products are reduced to two rows, a fast carry-propagate adder (such as a ripple-carry or carry-lookahead adder) is used to compute the final 8-bit product.

Regular 4x4 multiplier

Design:

A regular 4x4 multiplier is a digital circuit designed to compute the product of two 4-bit binary numbers, producing an 8-bit result. It operates by generating 16 partial products through AND gates, where each bit of one input is multiplied with every bit of the other. These partial products are then aligned according to their binary significance and summed column-wise using half adders and full adders. The addition process starts from the least significant bit (LSB), propagating carries to higher bits, and continues until all partial products are reduced to a single row, yielding the final 8-bit output.

Verilog code:

```
module multiplier_ra(a,b,p);

input [3:0]a,b;

output [7:0]p;

wire [2:0]w1,w2;

wire c1,c2;

and a1(p[0],a[0],b[0]);

ripple_adder
r1(.a({1'b0,a[0]&b[3],a[0]&b[2],a[0]&b[1]}),.b({a[1]&b[3],a[1]&b[2],a[1]&b[1],a[1]&b[0]}),.cin(1'b0),.s({w1,p[1]}
),.c(c1));

ripple_adder r2(.a({c1,w1}),.b({a[2]&b[3],a[2]&b[2],a[2]&b[1],a[2]&b[0]}),.cin(1'b0),.s({w2,p[2]}),.c(c2));

ripple_adder
r3(.a({c2,w2}),.b({a[3]&b[3],a[3]&b[2],a[3]&b[1],a[3]&b[0]}),.cin(1'b0),.s({p[6],p[5],p[4],p[3]}),.c(p[7]));

endmodule
```

Here we use a ripple carry adder module:

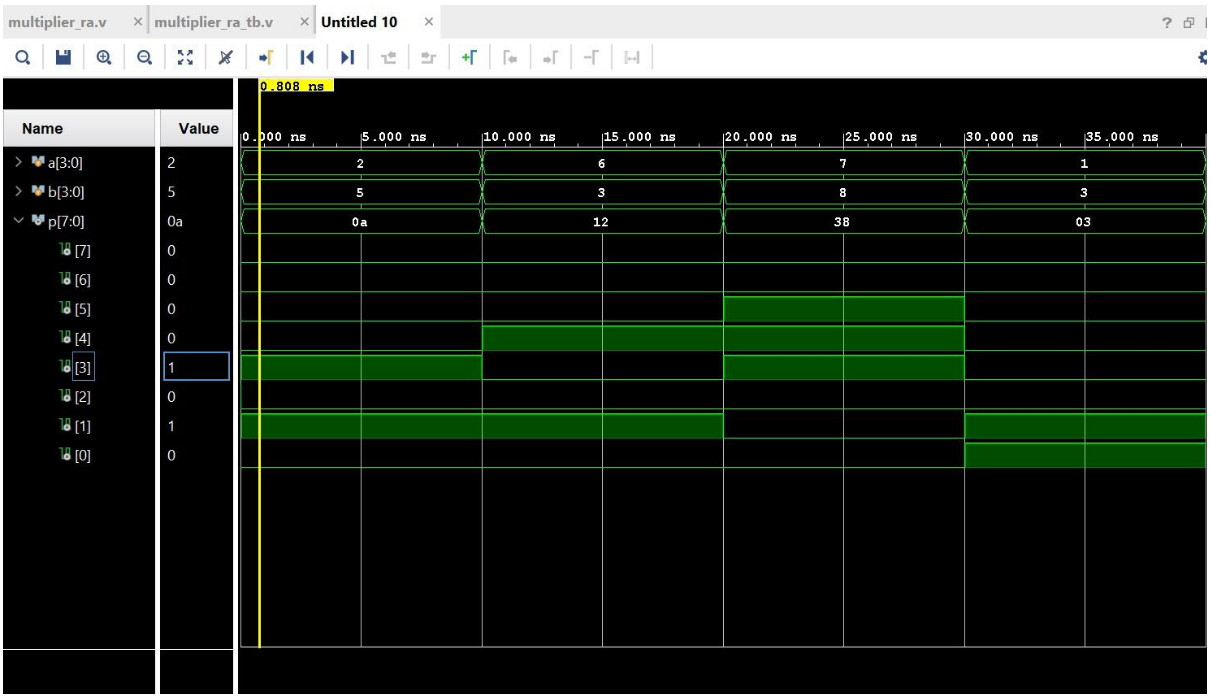
```
module ripple_adder(a,b,cin,s,c);

input [3:0]a,b;

input cin;
```

```
output [3:0]s;

output c;

wire c1,c2,c3;

full_adder fa1(a[0],b[0],cin,s[0],c1);

full_adder fa2(a[1],b[1],c1,s[1],c2);

full_adder fa3(a[2],b[2],c2,s[2],c3);

full_adder fa4(a[3],b[3],c3,s[3],c);

endmodule
```

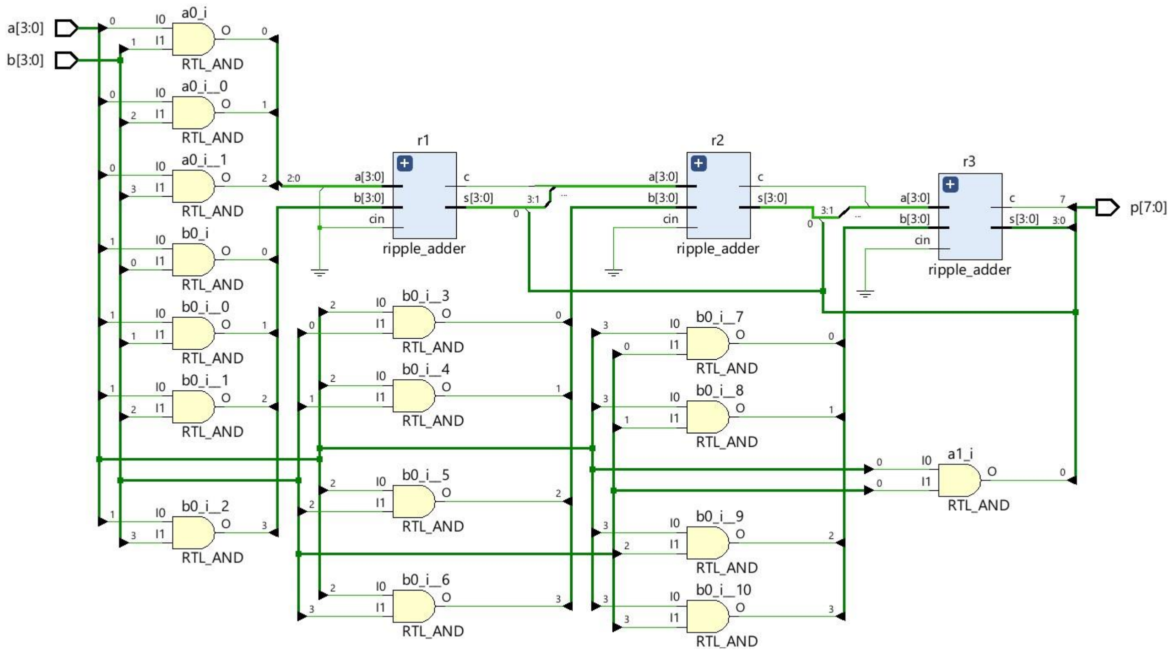
Testbench:

```
module multiplier_ra_tb();

reg [3:0]a,b;

wire [7:0]p;

multiplier_ra dut(a,b,p);

initial

begin

a = 4'b0010;b = 4'b0101;

#10 a = 4'b0110;b = 4'b0011;

#10 a = 4'b0111;b = 4'b1000;

#10 a = 4'b0001;b = 4'b0011;

#10 $finish;

end

endmodule
```

Waveform:



Schematic:



Optimized Wallace Tree 4x4 multiplier

Design:

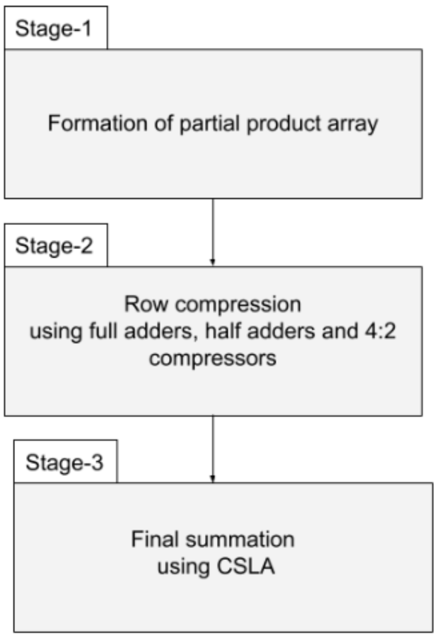

Stage-1
Formation of partial product array
Stage-2
Row compression
using full adders, half adders and 4:2 compressors
Stage-3
Final summation
using CSLA

Fig. 1. Block diagram of proposed Dadda-multiplier

Verilog Code:

```
module multiplier_cla(a,b,p);

input [3:0]a,b;

output [7:0]p;

wire s0,c0,s1,c1,s2,c2,s3,c3,s4,c4,s5,c5;

and(p[0],a[0],b[0]);


half_adder ha1(a[2]&b[0],a[1]&b[1],s0,c0);

full_adder fa1(a[3]&b[0],a[2]&b[1],a[1]&b[2],s1,c1);

half_adder ha2(a[3]&b[1],a[2]&b[2],s2,c2);


half_adder ha3(s1,a[0]&b[3],s3,c3);

full_adder fa2(a[1]&b[3],s2,c1,s4,c4);
full_adder fa3(a[3]&b[2],a[2]&b[3],c2,s5,c5);


cla_adder cla1({a[3]&b[3],s5,s4,s3,s0,a[1]&b[0]},{c5,c4,c3,c0,a[0]&b[2],a[0]&b[1]},1'b0,p[6:1],p[7]);

endmodule
```

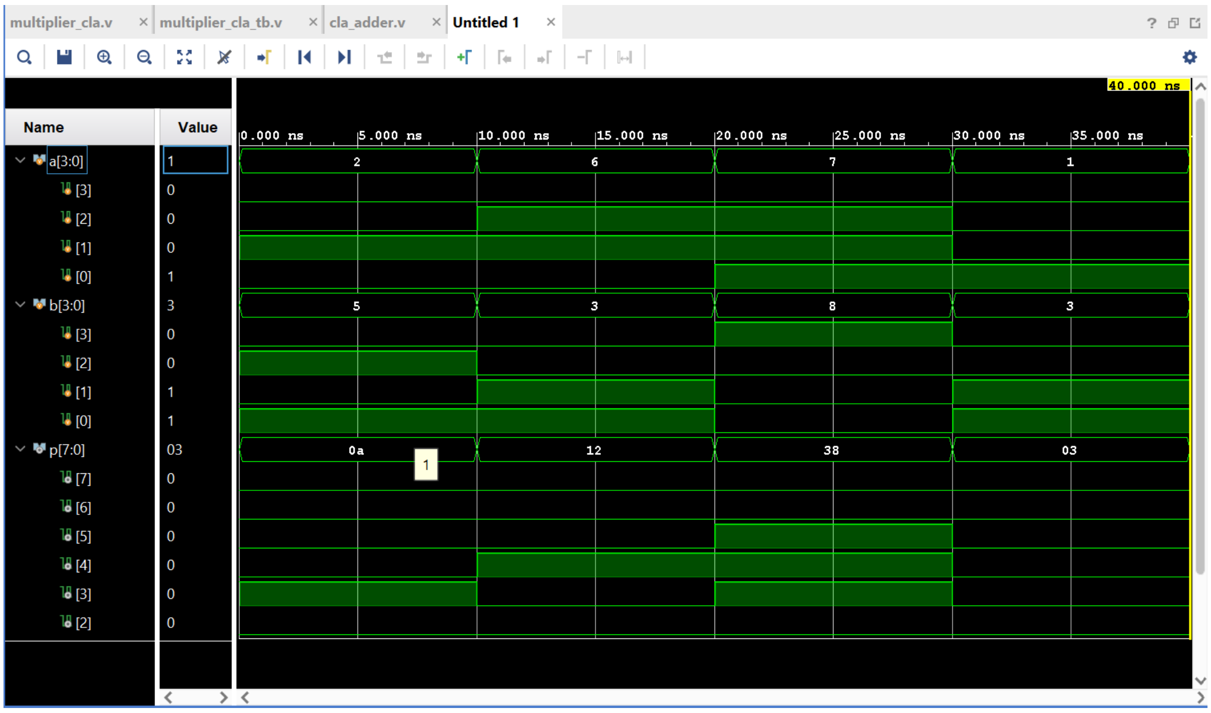
This uses a Carry Look Ahead adder module:

```
module cla_adder(a,b,cin,s,cout);

input [5:0]a,b;

input cin;

output [5:0]s;

output cout;

wire c1,c2,c3,c4,c5;


assign c1 = (a[0]&b[0])|((a[0]^b[0])&cin);

assign c2 = (a[1]&b[1])|((a[1]^b[1])&c1);

assign c3 = (a[2]&b[2])|((a[2]^b[2])&c2);

assign c4 = (a[3]&b[3])|((a[3]^b[3])&c3);

assign c5 = (a[4]&b[4])|((a[4]^b[4])&c4);

assign cout = (a[5]&b[5])|((a[5]^b[5])&c5);

assign s[0] = a[0]^b[0]^cin;

assign s[1] = a[1]^b[1]^c1;

assign s[2] = a[2]^b[2]^c2;

assign s[3] = a[3]^b[3]^c3;

assign s[4] = a[4]^b[4]^c4;

assign s[5] = a[5]^b[5]^c5;

endmodule
```

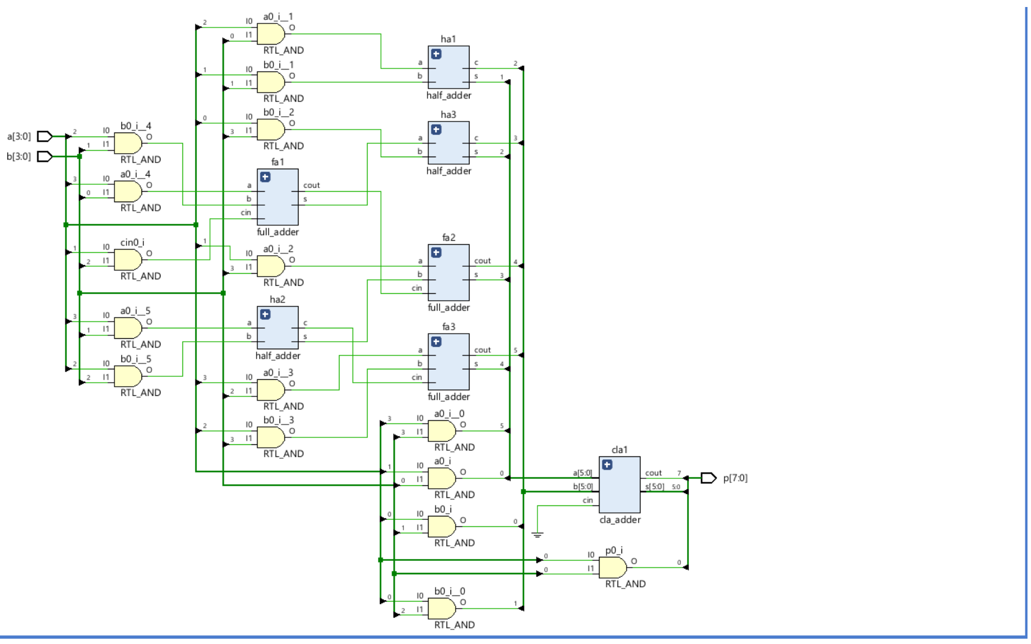
Test Bench:

```
module multiplier_cla_tb();

reg [3:0]a,b;

wire [7:0]p;

multiplier_cla dut(a,b,p);

initial

begin

a = 4'b0010;b = 4'b0101;

#10 a = 4'b0110;b = 4'b0011;

#10 a = 4'b0111;b = 4'b1000;

#10 a = 4'b0001;b = 4'b0011;

#10 $finish;

end

endmodule
```

Waveforms:



Schematic:



Conclusion:

The Optimized Wallace Tree Multiplier with CLA logic is ideal for applications requiring high performance and low latency, where speed is a priority. On the other hand, the Regular 4x4 Multiplier is better suited for low-power or resource-constrained environments, where simplicity and power efficiency are more critical than speed. The choice between the two designs should be driven by the specific requirements of the application, balancing power, speed, and hardware constraints.

References:

1. **A Design Technique for Delay and Power Efficient Dadda-Multiplier**
2. **Design and Study of Dadda Multiplier by using 4:2 Compressors and Parallel Prefix Adders for VLSI Circuit Designs**
3. **https://stackoverflow.com/**
4. **https://www.wikipedia.org/**