Discuss VHDL Component: Wallace Tree Multiplier (Generic) at the VHDL within the OpenHDL - Verilog and VHDL Discussion Forums; Hi everyone! Today, I'm going to illustrate the construction and VHDL code of a generic ...



10-14-2011 #1

**VHDLCoder**

Administrator

**Join Date:** Oct 2011

**Posts:** 63

**Blog Entries:** 17

## VHDL Component: Wallace Tree Multiplier (Generic)

Hi everyone!

Today, I'm going to illustrate the construction and VHDL code of a generic coded Wallace tree multiplier. The Wallace tree multiplier is not necessarily *faster* than other multiplier designs, particularly in an FPGA but does result in a small number of layers needed to compute the sum of the partial products.

The Wallace tree has a 3-step algorithm:

1. Multiply (i.e. AND) each bit of one of the arguments, by each bit of the other, yielding n^2 results.
2. Reduce the number of partial products to two by layers of full and half adders.
3. Group the wires in two numbers, and add them with a conventional adder.

To reduce the number of partial products, we maximize the number of 3:2 and 2:2 (i.e. full and half adders respectively) that we can bit along a set of weights. Weights are define the relative alignment of a bit with respect to the other partial products. For every 3:2 and 2:2 compressor, the sum bit is fed into the next layer of the *same* weight and the carry bit is fed into the *next highest* weight of the next layer. If only 1 bit remains in the weight for that layer, pass it up to the same weight in the next layer. Here's a look at the structure of an 8x8 multiplier and the resulting Wallace tree reduction on the initial 8 partial products.

Notice the reduction in using compressors from layer to layer. The construction of a generic version of the Wallace tree multiplier can be tricky, so here's a look at the VHDL code. Take some time to look at how a set of recursive VHDL functions can cue the tree process on how many compressors to insert at each layer.

VHDL Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE my_funs IS
    FUNCTION clogb2 (a: NATURAL) RETURN NATURAL;
    FUNCTION prev_lvl_carry (width: NATURAL; this_weight: NATURAL; this_lvl: NATURAL) RETURN NATURAL;
    FUNCTION this_lvl_bits (width: NATURAL; this_weight: NATURAL; this_lvl: NATURAL) RETURN NATURAL;
    FUNCTION num_full_adders (width: NATURAL; this_weight: NATURAL; this_lvl: NATURAL) RETURN NATURAL;
    FUNCTION num_half_adders (width: NATURAL; this_weight: NATURAL; this_lvl: NATURAL) RETURN NATURAL;
END my_funs;

PACKAGE BODY my_funs IS
    FUNCTION clogb2 (a: NATURAL) RETURN NATURAL IS
        VARIABLE aggregate : NATURAL := a;
        VARIABLE return_val : NATURAL := 0;
    BEGIN
        compute_clogb2:
        FOR i IN a DOWNTO 0 LOOP

            IF aggregate > 0 THEN
                return_val := return_val + 1;
            END IF;

            aggregate := aggregate / 2;
        END LOOP;

        RETURN return_val;

    END clogb2;

    FUNCTION prev_lvl_carry (width: NATURAL; this_weight: NATURAL; this_lvl: NATURAL) RETURN NATURAL IS
        VARIABLE this_weight_base_bits: NATURAL := 0;
        VARIABLE this_num_bits: NATURAL := 0;
        VARIABLE num_carry: NATURAL := 0;
    BEGIN
        IF this_weight > (width-1) THEN
            IF this_weight = 2*width-1 THEN
                this_weight_base_bits := 1;
            ELSIF this_weight = width THEN
                this_weight_base_bits := 2*width-this_weight;
            ELSE
                this_weight_base_bits := 2*width-this_weight-1;
            END IF;
        ELSE
            this_weight_base_bits := this_weight+1;
        END IF;

        IF this_lvl > 0 THEN -- Recursive case
            IF this_weight > 0 THEN
                this_num_bits := this_lvl_bits(width,this_weight-1,this_lvl-1);
                num_carry := this_num_bits/3;
                num_carry := num_carry + (this_num_bits-num_carry*3)/2;
            ELSE
                num_carry := 0;
            END IF;
        ELSE
            num_carry := this_weight_base_bits/3;
            num_carry := num_carry + (this_weight_base_bits-num_carry*3)/2;
        END IF;

        RETURN num_carry;
    END prev_lvl_carry;

    FUNCTION this_lvl_bits (width: NATURAL; this_weight: NATURAL; this_lvl: NATURAL) RETURN NATURAL IS
        VARIABLE this_weight_base_bits: NATURAL := 0;
        VARIABLE prev_lvl_bits: NATURAL := 0;
        VARIABLE full_adder_sums: NATURAL := 0;
        VARIABLE half_adder_sums: NATURAL := 0;
        VARIABLE this_num_bits: NATURAL := 0;
    BEGIN
        IF this_weight > (width-1) THEN
            IF this_weight = 2*width-1 THEN
                this_weight_base_bits := 1;
            ELSIF this_weight = width THEN
                this_weight_base_bits := 2*width-this_weight;
            ELSE
                this_weight_base_bits := 2*width-this_weight-1;
            END IF;
```

```
        ELSE
            this_weight_base_bits := this_weight+1;
        END IF;

        IF this_lvl > 0 THEN -- Recursive case
            IF this_weight > 0 THEN
                prev_lvl_bits := this_lvl_bits(width,this_weight,this_lvl-1);
                full_adder_sums := prev_lvl_bits/3;
                half_adder_sums := (prev_lvl_bits-full_adder_sums*3)/2;
                this_num_bits := prev_lvl_bits - 2*full_adder_sums- half_adder_sums + prev_lvl_carry(width,this_weight,this_lvl);
            ELSE
                this_num_bits := this_lvl_bits(width,this_weight,this_lvl-1);
            END IF;
        ELSE
            this_num_bits := this_weight_base_bits;
        END IF;

        RETURN this_num_bits;
    END this_lvl_bits;

    FUNCTION num_full_adders (width: NATURAL; this_weight: NATURAL; this_lvl: NATURAL) RETURN NATURAL IS
        VARIABLE this_num_bits: INTEGER := this_lvl_bits(width,this_weight,this_lvl);
    BEGIN
        RETURN (this_num_bits/3);
    END num_full_adders;

    FUNCTION num_half_adders (width: NATURAL; this_weight: NATURAL; this_lvl: NATURAL) RETURN NATURAL IS
        VARIABLE this_num_bits: INTEGER := this_lvl_bits(width,this_weight,this_lvl);
        VARIABLE num_full_adds: INTEGER := 0;
    BEGIN
        num_full_adds := this_num_bits/3;
        RETURN ((this_num_bits-num_full_adds*3)/2);
    END num_half_adders;
END my_funs;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_funs.all;

ENTITY wallace_mult IS
    GENERIC (
        width : INTEGER := 4
    );
    PORT (
        a :    IN STD_LOGIC_VECTOR(width-1 DOWNTO 0);
        b :    IN STD_LOGIC_VECTOR(width-1 DOWNTO 0);
        prod : OUT STD_LOGIC_VECTOR(2*width-1 DOWNTO 0)
    );
END wallace_mult;

ARCHITECTURE behavioral OF wallace_mult IS
    TYPE layer_depth_type IS ARRAY(32 DOWNTO 3) OF INTEGER;
    CONSTANT layer_depth: layer_depth_type := (9,9,9,8,8,8,8,8,8,8,8,7,7,7,7,7,7,7,7,7,7,7,6  ,5,5,4,3,3);
    CONSTANT stages: INTEGER := layer_depth(width);
    TYPE W_type IS ARRAY(2*width-1 DOWNTO 0, width-1 DOWNTO 0, stages-1 DOWNTO 0) OF STD_LOGIC;
    TYPE P_type IS ARRAY(width-1 DOWNTO 0, width-1 DOWNTO 0) OF STD_LOGIC;
    SIGNAL P: P_type; -- Initial product tree
    SIGNAL W: W_type; -- Wallace tree
    SIGNAL add_a, add_b, add_sum: STD_LOGIC_VECTOR(2*width-1 DOWNTO 0);
    SIGNAL c_in: STD_LOGIC := '0';

    COMPONENT bk_adder
        GENERIC (
            width :     INTEGER := 4
        );
        PORT (
            a :    IN STD_LOGIC_VECTOR(width-1 DOWNTO 0);
            b :    IN STD_LOGIC_VECTOR(width-1 DOWNTO 0);
            c_in :    IN STD_LOGIC;
            sum :    OUT STD_LOGIC_VECTOR(width-1 DOWNTO 0);
            c_out :    OUT STD_LOGIC
        );
    END COMPONENT;

BEGIN

    partial_proc: PROCESS(a,b)
    BEGIN
        FOR i IN width-1 DOWNTO 0 LOOP
            FOR j IN width-1 DOWNTO 0 LOOP
                P(i,j) <= a(i) AND b(j);
            END LOOP;
        END LOOP;
    END PROCESS;

```

```
164.     wallace_proc: PROCESS(W,P)
165.         VARIABLE this_carry_bits: NATURAL := 0;
166.         VARIABLE num_full_adds: NATURAL := 0;
167.         VARIABLE num_half_adds: NATURAL := 0;
168.         VARIABLE num_wires: NATURAL := 0;
169.     BEGIN
170.         W(2*width-1,0,0) <= '1'; -- Extended sign bit
171.         W(width,width-1,0) <= '1'; -- Sign bit
172.         FOR i IN 2*width-2 DOWNTO 0 LOOP
173.             IF i <= (width-1) THEN
174.                 FOR j IN i DOWNTO 0 LOOP
175.                     IF (j = width-1) XOR (i-j = width-1) THEN
176.                         W(i,j,0) <= NOT(P(j,i-j));
177.                     ELSE
178.                         W(i,j,0) <= P(j,i-j);
179.                     END IF;
180.                 END LOOP;
181.             ELSE
182.                 FOR j IN width-1 DOWNTO i-width+1 LOOP
183.                     IF (j = width-1) XOR (i-j = width-1) THEN
184.                         W(i,j-i+width-1,0) <= NOT(P(j,i-j));
185.                     ELSE
186.                         W(i,j-i+width-1,0) <= P(j,i-j);
187.                     END IF;
188.                 END LOOP;
189.             END IF;
190.         END LOOP;
191.
192.         FOR k IN 0 TO stages-2 LOOP
193.             FOR i IN 2*width-1 DOWNTO 0 LOOP
194.                 this_carry_bits := prev_lvl_carry(width, i, k+1);

195.
196.                 -- Full adders (3:2 Compressors)
197.                 num_full_adds := num_full_adders(width,i,k);
198.                 FOR j IN 0 TO num_full_adds-1 LOOP
199.                     W(i,this_carry_bits+j,k+1) <= W(i,j*3,k) XOR W(i,j*3+1,k) XO
    R W(i,j*3+2,k);
200.                     IF i < 2*width-1 THEN
201.                         W(i+1,j,k+1) <= (W(i,j*3,k) AND W(i,j*3+1,k)) XOR (W(i,j
    *3+2,k) AND (W(i,j*3,k) XOR W(i,j*3+1,k)));
202.                     END IF;
203.                 END LOOP;
204.
205.                 -- Half adders (2:2 Compressors)
206.                 num_half_adds := num_half_adders(width,i,k);
207.                 FOR j IN 0 TO num_half_adds-1 LOOP
208.                     W(i,this_carry_bits+num_full_adds+j,k+1) <= W(i,num_full_add
    s*3+j*2,k) XOR W(i,num_full_adds*3+j*2+1,k);
209.                     IF i < 2*width-1 THEN
210.                         W(i+1,num_full_adds+j,k+1) <= W(i,num_full_adds*3+j*2,k)
     AND W(i,num_full_adds*3+j*2+1,k);
211.                     END IF;
212.                 END LOOP;
213.
214.                 -- Wires
215.                 num_wires := this_lvl_bits(width,i,k)-num_full_adds*3-num_half_a
    dds*2;
216.                 FOR j IN 0 TO num_wires-1 LOOP
217.                     W(i,this_carry_bits+num_full_adds+num_half_adds+j,  k+1) <=
    W(i,num_full_adds*3+num_half_adds*2+j,k);
218.                 END LOOP;
219.             END LOOP;
220.         END LOOP;
221.     END PROCESS;
222.
223.     -- Final Adder (Using a Brent Kung Adder)
224.     signal_vect_proc: PROCESS(W)
225.     BEGIN
226.         FOR i IN 2*width-1 DOWNTO 0 LOOP
227.             add_a(i) <= W(i,0,stages-1);
228.             add_b(i) <= W(i,1,stages-1);
229.         END LOOP;
230.     END PROCESS;
231.
232.     U_bk_add: bk_adder
233.     GENERIC MAP (
234.         width => 2*width
235.     )
236.     PORT MAP (
237.         a => add_a,
238.         b => add_b,
239.         c_in => c_in,
240.         sum => add_sum
241.     );
242.
243.     prod <= add_sum;
244.
245. END behavioral;
246.
247. LIBRARY ieee;
248. USE ieee.std_logic_1164.all;
```

```
USE work.my_funs.all;

ENTITY bk_adder IS
    GENERIC (
        width :     INTEGER := 7
    );
    PORT (
        a :    IN STD_LOGIC_VECTOR(width-1 DOWNTO 0);
        b :    IN STD_LOGIC_VECTOR(width-1 DOWNTO 0);
        c_in :    IN STD_LOGIC;
        sum :    OUT STD_LOGIC_VECTOR(width-1 DOWNTO 0);
        c_out :    OUT STD_LOGIC
    );
END bk_adder;

ARCHITECTURE behavioral OF bk_adder IS
    CONSTANT nn: INTEGER := clogb2(width);
    CONSTANT inv_nn: INTEGER := clogb2(width+2**(nn-2))-2;
    TYPE T_type IS ARRAY(nn+inv_nn-1 DOWNTO 0, width-1 DOWNTO 0) OF STD_LOGIC_VECTOR(1 DOWNTO 0);
    SIGNAL T: T_type;
BEGIN

    -- Carry tree with maximum number of stages
    tree_proc: PROCESS(T,a,b,c_in)
    BEGIN
        -- First bit is a full adder
        T(0,0)(0) <= (a(0) AND b(0)) OR (c_in AND (a(0) XOR b(0)));
        T(0,0)(1) <= a(0) XOR b(0) XOR c_in;

        -- Leaves of tree
        FOR j IN width-1 DOWNTO 1 LOOP
            T(0,j)(0) <= a(j) AND b(j); -- Generate bit base
            T(0,j)(1) <= a(j) XOR b(j); -- Propagate bit base
        END LOOP;

        -- Carry tree
        FOR i IN 1 TO nn-1 LOOP
            FOR j IN width-1 DOWNTO 0 LOOP
                IF(j mod 2**i = (2**i)-1) THEN
                    IF((j-2**(i-1)) >= 0) THEN
                        T(i,j)(0) <= (T(i-1,j)(1) AND T(i-1,j-2**(i-1))(0)) OR T(i-1,j)(0); -- G = (P_i and G_i_prev) or G_i
                        T(i,j)(1) <= T(i-1,j)(1) AND T(i-1,j-2**(i-1))(1); -- P = P_i and P_i_prev
                    ELSE
                        T(i,j)(0) <= T(i-1,j)(0); -- G = G_i (since we are at tree's edge, there is no G_i_prev)
                        T(i,j)(1) <= T(i-1,j)(1); -- P = P_i (since we are at tree's edge, there is no P_i_prev)
                    END IF;
                ELSE
                    T(i,j)(0) <= T(i-1,j)(0);
                    T(i,j)(1) <= T(i-1,j)(1);
                END IF;
            END LOOP;
        END LOOP;

        -- Inverse carry tree
        FOR i IN nn+inv_nn DOWNTO nn+1 LOOP
            FOR j IN width-1 DOWNTO 0 LOOP
                IF((j-2**(nn+inv_nn-(i))) mod 2**((nn+inv_nn-(i))+1) = 2**((nn+inv_nn-(i))+1)-1) THEN
                    IF(j >= 2**(nn+inv_nn-i)) THEN
                        T(i-1,j)(0) <= (T(i-2,j)(1) AND T(i-2,j-2**((nn+inv_nn-(i))))(0)) OR T(i-2,j)(0); -- G = (P_i and G_i_prev) or G_i
                        T(i-1,j)(1) <= T(i-2,j)(1) AND T(i-2,j-2**((nn+inv_nn-(i))))(1); -- P = P_i and P_i_prev
                    ELSE
                        T(i-1,j)(0) <= T(i-2,j)(0);
                        T(i-1,j)(1) <= T(i-2,j)(1);
                    END IF;
                ELSE
                    T(i-1,j)(0) <= T(i-2,j)(0);
                    T(i-1,j)(1) <= T(i-2,j)(1);
                END IF;
            END LOOP;
        END LOOP;
    END PROCESS;

    -- Basic summation for carry tree
    sum_proc: PROCESS(T)
    BEGIN
        sum(0) <= T(0,0)(1);
        FOR i IN width-1 DOWNTO 1 LOOP
            sum(i) <= T(0,i)(1) XOR T(nn+inv_nn-1,i-1)(0);
        END LOOP;
    END PROCESS;

    c_out <= T(nn+inv_nn-1,width-1)(0) OR (T(nn+inv_nn-1,width-1)(1) AND T(nn+inv_nn-1,width-2)(0));

```

```
END behavioral;
```

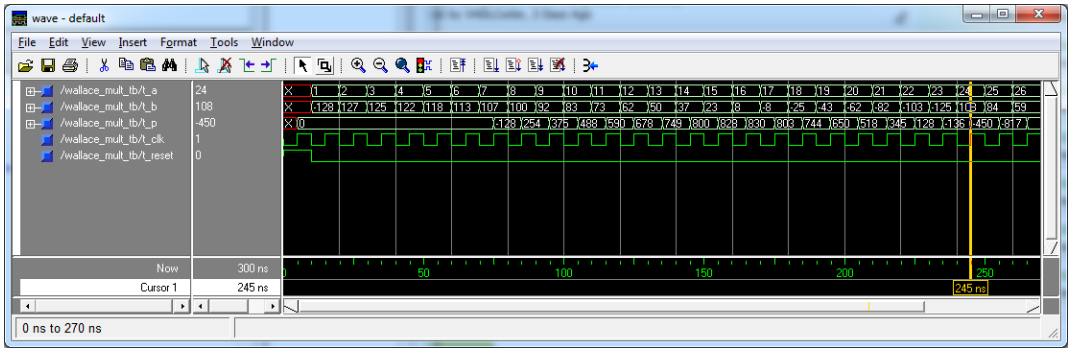
Note that this Wallace tree multiplier is set up as a ***combinational*** and ***signed operand*** multiplier. An **unsigned synchronous** and **signed synchronous** version of each is attached. Also, note that I have used a Brent-Kung Adder for the final partial sums that I previously discussed. Now, here's a test bench to verify proper function of the Wallace tree multiplier.

VHDL Code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE std.textio.all;

ENTITY wallace_mult_tb IS
    GENERIC (
        width: INTEGER := 11
    );
END wallace_mult_tb;

ARCHITECTURE tb OF wallace_mult_tb IS
    SIGNAL t_a: STD_LOGIC_VECTOR(width-1 DOWNTO 0);
    SIGNAL t_b: STD_LOGIC_VECTOR(width-1 DOWNTO 0);
    SIGNAL t_p: STD_LOGIC_VECTOR(2*width-1 DOWNTO 0);
    SIGNAL t_clk: STD_LOGIC;
    SIGNAL t_reset: STD_LOGIC;

    COMPONENT wallace_mult
        GENERIC (
            width : INTEGER := 4
        );
        PORT (
            a :    IN STD_LOGIC_VECTOR(width-1 DOWNTO 0);
            b :    IN STD_LOGIC_VECTOR(width-1 DOWNTO 0);
            clk :    IN STD_LOGIC;
            reset :    IN STD_LOGIC;
            prod : OUT STD_LOGIC_VECTOR(2*width-1 DOWNTO 0)
        );
    END COMPONENT;

    FUNCTION to_string(sv: Std_Logic_Vector) return string is
            USE Std.TextIO.all;
            USE ieee.std_logic_textio.all;
        VARIABLE lp: line;
      BEGIN
            write(lp, to_integer(unsigned(sv)));
            RETURN lp.all;
      END;
BEGIN
    U_wallace_mult: wallace_mult
    GENERIC MAP (
        width => width
    )
    PORT MAP (
        a => t_a,
        b => t_b,
        clk => t_clk,
        reset => t_reset,
        prod => t_p
    );

    -- Clock Process
    clk_prc: PROCESS
    BEGIN
        t_clk <= '0';
        WAIT FOR 5 ns;
        t_clk <= '1';
        WAIT FOR 5 ns;
    END PROCESS;

    -- Input Processes
    inp_prc: PROCESS
        VARIABLE v_a: INTEGER := 0;
        VARIABLE v_b: INTEGER := 2**(width-1);
    BEGIN
        FOR i IN 0 TO 2**width LOOP
            WAIT FOR 10 ns;
            v_a := v_a + 1;
            v_b := v_b - i;
            t_a <= std_logic_vector(to_signed(v_a,width));
            t_b <= std_logic_vector(to_signed(v_b,width));

        END LOOP;
    END PROCESS;

    -- Reset Process
```

```
78.      rst_prc: PROCESS
79.      BEGIN
80.          t_reset <= '1';
81.          WAIT FOR 10 ns;
82.          t_reset <= '0';
83.          WAIT;
84.      END PROCESS;
85.
86.  END tb;
```

And here's the output waveform:

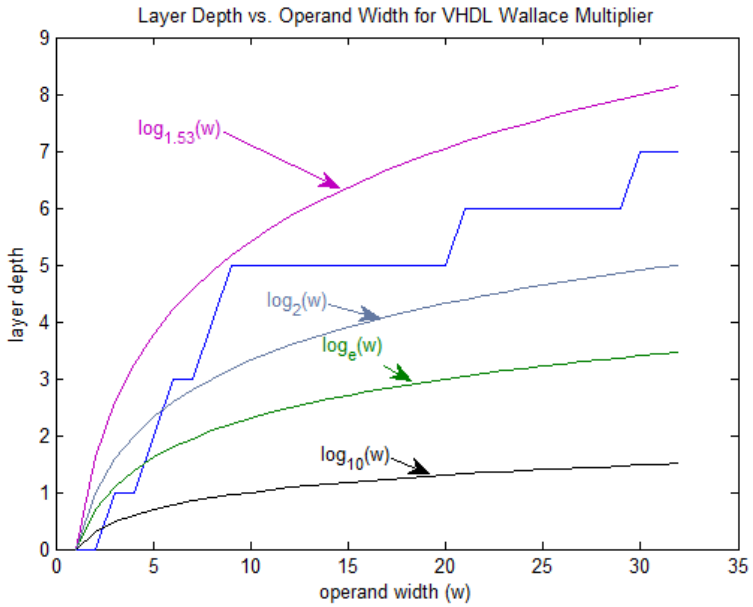



Next, let's take a look at the performance of a 16x16 bit (combinatorial!) version of this Wallace tree multiplier. Using Synplify Pro and choosing Xilinx Virtex2 XC2V40 with CS144 Package and -6 Speed yields the following performance data:

Code:

```
Performance Summary
*******************


Worst slack in design: -1.450

                   Requested     Estimated     Requested     Estimated               Clock
        Clock
Starting Clock     Frequency     Frequency     Period        Period        Slack     Type
        Group
------------------------------------------------------------------------------------------------
------------------------------
top|clk            108.6 MHz     93.8 MHz      9.207         10.657        -1.450    infer
red     Autoconstr_clkgroup_0
================================================================================================
==============================


----------------------------------------
Resource Usage Report for top
```

This 16x16 bit multiplier may be unsuitable for use in a high-speed FPGA application requiring many multipliers (that's when device specific multipliers should be used). However, a synchronous version of the multiplier (attached) may be useful for lower power and simpler devices that do not have multipliers built in. Here's a look at some layer and delay specifics for this particular multiplier implementation:

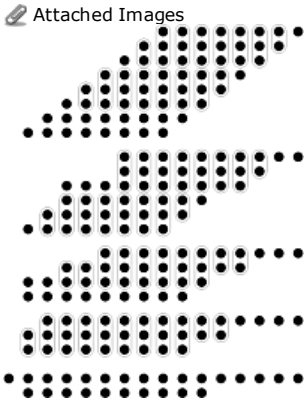


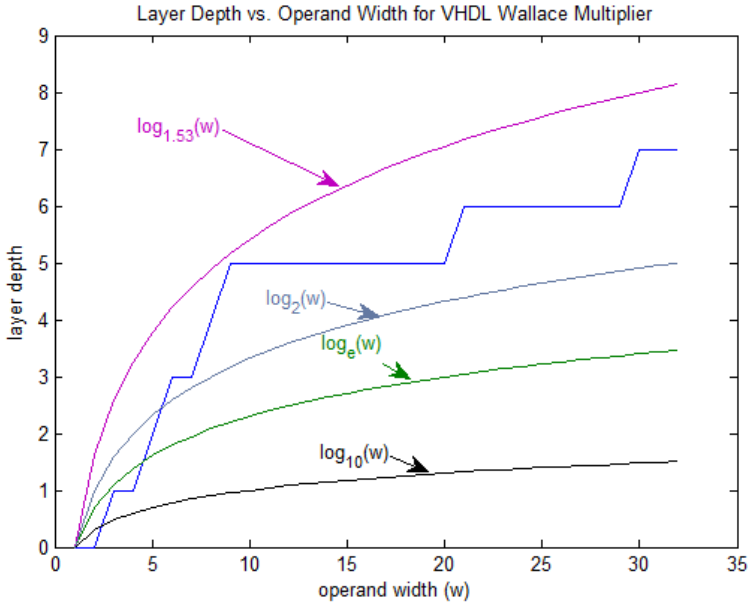

The layer depth is charted out to 32 bit operands (you'll have to experiment yourself if you are interested in larger operands). The layer depth is a good indicator of how much delay the multiplier would have if it were placed in a pipelined situation. This implementation is "delay fat" so there are actual 2 additional delays on top of the layer depth. You are welcome to play with it to remove the input and output registers.

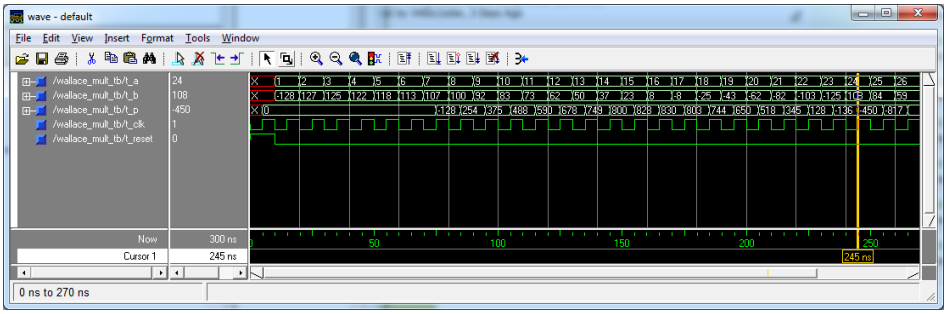
Until next time!

VHDL Coder

Attached Images






Attached Files
wallace_mult_unsigned.vhd (10.4 KB, 251 views)
wallace_mult_signed.vhd (10.8 KB, 265 views)
wallace_mult_signed_comb.vhd (10.1 KB, 246 views)
wallace_mult_tb.vhd (1.7 KB, 256 views)

*Last edited by VHDLCoder; 10-14-2011 at 07:14 PM.*

**Reply With Quote**

11-20-2011 #2

jeevitha

Guest

sir/mam, I need to do my project. im new to this dept vhdl n verilog n to wallace. i cudnt get the output for the above prog. In output the prod,add_a, sum is not produced...Can u pls explain me.. Thank u in advance..

**Reply With Quote**

11-20-2011 #3

**VHDLCoder**

Administrator

| | |
|---|---|
| **Join Date:** | Oct 2011 |
| **Posts:** | 63 |
| **Blog Entries:** | 17 |

Hi jeevitha,

Thanks for trying out my wallace tree multiplier! Perhaps you have identified a bug. Can you explain what version of the code you used (i.e. signed, unsigned, or combinatorial signed)? Were you using the test bench from the original post? What software are you using to synthesize and simulate the multiplier? Are any other signals not working properly?

Thanks!

VHDLCoder

Reply With Quote

11-21-2011 #4

jeevitha

Guest

hi
Im using modelsim and xilinx in both i got the same result. In xilinx its mentioning as "Matrix not supported yet" for the W type array. in modelsim it dint show any error after simulation i dint get the output for add_b, add_sum, and the pro. The last three four bits are undefines inthem. i ve tried to trace the program but i couldnt succeed.
Thank you .

Reply With Quote

11-21-2011 #5

**VHDLCoder**

Administrator

**Join Date:** Oct 2011
**Posts:** 63
**Blog Entries:** 17

Hi jeevitha,

I may understand what is happening with the modelsim simulation. The multiplier has a separate entity called 'bk_adder' which is a type of carry tree adder to add the final two vectors of the Wallace carry save adder tree. Try compiling the wallace_mult and wallace_mult_tb first, then you may see a new entity 'bk_adder' appear in your workspace in ModelSim. Then, recompile all 3 entities and try the simulation again. I noticed that in the way the code is written, ModelSim may give the error:

"** Warning: (vsim-3473) Component 'u_bk_add' is not bound."

Recompiling again after compiling bk_adder will remedy this.

As for the Xilinx ISE, you may need to adjust some of your compiler settings to get it to accept multi-dimensional arrays in VHDL.

Good luck!

VHDLCoder

Reply With Quote

11-21-2011 #6

gorand2

Guest

**Hi**

Hi VHDLCoder
I have looked at wallace_mult_signed.vhd
It seems that it always assumes one of the numbers is negative and the other positive, and doesn't work if this is not the case. Is that correct?
Also i would like to ask if you know a book describing how to implement signed wallace tree multiplier.

Reply With Quote

11-21-2011 #7

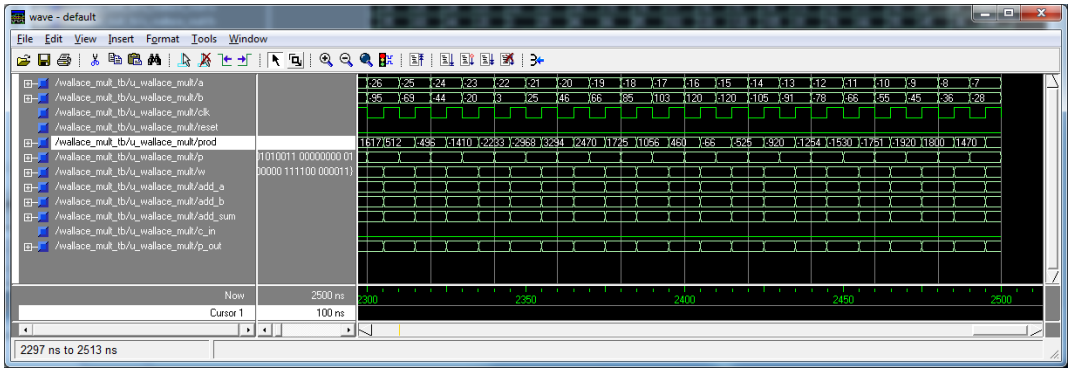
**VHDLCoder**

Administrator

**Join Date:** Oct 2011
**Posts:** 63
**Blog Entries:** 17

Hi gorand2,

The input process uses the "engineering approach" to signed binary multiplication that you can read more about at **Wikipedia**. Both multiplicands can be signed and here's a section of the simulation of the signed synchronous wallace tree multiplier implementation where the multiplicands are alternating negative and positive.



I'd have to look through my material to see if there are any specific examples of implementing a Wallace tree multiplier. I am not sure if I have seen an example of one in a book.

Hope that helps!

VHDLCoder

Reply With Quote

11-27-2012 #8

DEVESH

New Member

**Join Date:** Nov 2012

**Posts:** 4

sir i am trying to write vhdl code for logarithmic multiplier.. i am in need help in this

Reply With Quote

11-28-2012 #9

**VHDLCoder**

Administrator

**Join Date:** Oct 2011

**Posts:** 63

**Blog Entries:** 17

Hi devesh,

Did you see my article for logarithms? You can find it here:

**http://www.openhdl.com/blogs/vhdlcod...hesizable.html**

Does that help? It will compute logarithms in VHDL given any base and any input of your desired size.

Reply With Quote

01-23-2013 #10

srinivas

New Member

**Join Date:** Dec 2012

**Posts:** 4

hello sir, i tried your generic modified wallace tree combinational multplier in xlinx ISE and i got the synthesised result but lots of warnings are coming, and in test bench it is not getting the exact output and we want to extend the length of the multplier for that we placed 8 in width of the above code, is it enough or any other changes to made?

Reply With Quote



« VHDL Component: Kogge-Stone Adder (Generic) | booth multiplier »

LinkBacks (?)

VHDL code for 32 bit wallace tree multiplier please..... - All About Circuits Forum 11-15-2011, 12:42 PM
Refback This thread

VHDL code for 32 bit wallace tree multiplier please..... - All About Circuits Forum 10-28-2011, 02:02 AM
Refback This thread

VHDL code for 32 bit wallace tree multiplier please..... - All About Circuits Forum 10-26-2011, 01:49 AM
Refback This thread

VHDL code for 32 bit wallace tree multiplier please..... - All About Circuits Forum 10-18-2011, 02:25 AM
Refback This thread

VHDL code for multiplier 10-17-2011, 02:38 PM
Refback This thread

Posting Permissions

You may not post new threads
You may not post replies
You may not post attachments
You may not edit your posts

**BB code** is On
**Smilies** are On
**[IMG]** code is On
**[VIDEO]** code is On
HTML code is Off
**Trackbacks** are On

**Pingbacks** are On

**Refbacks** are On

**Forum Rules**

All times are GMT -4. The time now is 09:34 AM.