Coding a 40x40 Pipelined Multiplier

J. W. Lewis
SynthWorks Design Inc
Jim@SynthWorks.com 11898 SW 128th Ave, Tigard, OR 97223

Abstract

This paper covers effective VHDL coding styles to implement a pipelined 40 bit by 40 bit multiplier. We will analyze several designs in an attempt to find a pipelined design that will work on both synthesis tools that support pipelining and those that do not. The coding style arrived at first does a partial multiply followed by a register. Then partial results are shifted and summed to form the final result.

I. INTRODUCTION

The day is upon us where a synthesis tool can pipeline a multiplier for us. However if you read the fine print, pipelining is only available with a high-end synthesis tool license and it is not necessarily supported for all FPGA technologies. What happens when the synthesis tool or FPGA we are using does not support pipelining (register balancing)? Can we get good results by using a coding style?

This paper covers effective VHDL coding styles to implement a pipelined 40 bit by 40 bit multiplier. First we will look at a nominal test case, 40x40 Multiplier without any pipelining. This gives us a baseline to compare our pipelined designs against.

The first candidate coding style is a multiplier followed by two registers. Behaviorally this operates with the same characteristics as a pipelined multiplier. A synthesis tool that supports pipelining (register balancing) can move part of the multiplier through the first register and improve timing.

Next we look at how to do the pipelining in our code. The basis for this is to multiply by shift and add. A pipelined shift and add multiplier can get effective results, however, the VHDL code is tedious to write and difficult to read and maintain.

The final code solution looks for a balance between getting the desired results and coding at a high level. The first stage does a partial multiply followed by a register. The second stage shifts and adds the results of the partial multiply and then registers the final results. This solution leaves the tedious work for the synthesis tool and keeps our code readable.

II. BASELINE

The first step is to code a basic 40x40 multiplier and assess how fast it will operate. The code is shown in Figure 1. The synthesis results are shown in Table 1. Note the results in this paper come directly from a synthesis tool. This timing is an estimate and only for typical operating conditions. This is good enough to compare coding styles (as we are doing in this paper), but it is not good enough to predict actual device performance. The synthesis tool used was Synplicity's SynplifyPro™.

```
entity Mult is
  port (
    A, B   : In    unsigned (39 downto 0) ;
    Y      : Out unsigned (79 downto 0)
  ) ;
end Mult ;
architecture Rtl_Ref of Mult is
begin
    Y <= A * B ;
end Rtl_Ref ;
```

Figure 1. VHDL Code for 40x40 Multipler

Table 1. Baseline 40x40 Multiply Synthesis Results

Technology	Frequency (MHz)	Area (cells / LUTs)
Actel, ProAsic, APA450	12.7	6903
Xilinx Vertex-E XCV200E-8	50.7	1666

III. VISUALIZING PIPELINING

Pipelining trades increased latency for a higher frequency. It does this by adding register stages. When pipelining a multiplier, the goal is to put half of the multiplier before the added register and the other half of the multiplier after the register. This is shown below in Figure 2.

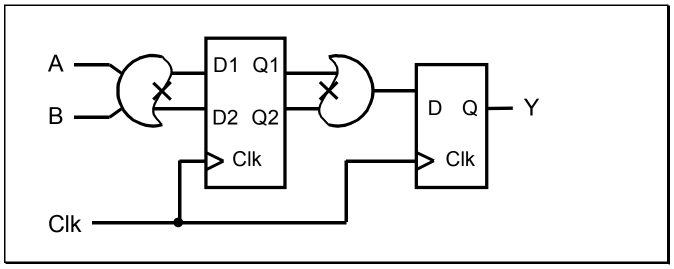

Figure 2. Visualizing Pipelining a Multiplier.

IV. Synthesis Tool Pipelining

Behaviorally a pipelined multiplier can be represented by a multiplier followed by two registers. A block diagram of this is shown in Figure 3. The VHDL code is shown in Figure 4. A synthesis tool that supports pipelining (register balancing) can move part of the multiplier through the first register and improve timing. Synthesis results are shown in Table 2. For Xilinx and Altera, this technique is simple and effective. For Actel, the synthesis tool did not support pipelining.

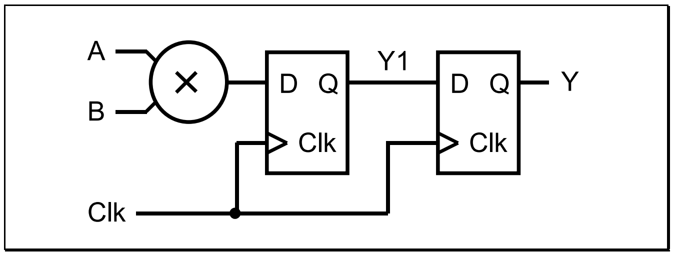

Figure 3. Block Diagram of Synthesis Tool Pipelining

```
PipeMultProc : process
begin
  wait until clk = '1' ;
  Y1 <= A * B ;
  Y <= Y1 ;
end process ;
```

Figure 4. VHDL Code for Synthesis Tool Pipelining

Table 2. Synthesis Results for Synthesis Tool Pipelining

Technology	Pipelining Enabled	Frequency (MHz)	Area (cells/LUTs)
Actel	Not Available	12.6	7063
Xilinx	No	49.7	1665
Xilinx	Yes	79.8	2238

V. Multiplication by Shift and Add

VHDL design methodology starts by drawing a block diagram. Next the block diagram is used as a flow chart to write the code. So from a methodology perspective, visualize and then code.

When a problem is encountered, the problem solving sequence starts by identifying the problem. In this case the problematic VHDL code is a 40x40 multiplier. Next we break the problem down into its sub-components. A multiplier can be broken down into shift and add operations. Now once the problem has a new visualization, re-code the design.

An example of shift and add for an 8x8 multiply is shown in Figure 5. A block diagram is shown in Figure 6. The VHDL code is shown in Figure 7. Note that the "AND" shown in the VHDL code is overloaded to "AND" type std_logic with type unsigned. The code for this is shown at the end of this paper.

```
           1010 1010
       x   1101 0101
--------------------
           1010 1010
         0 0000 000
        10 1010 10
       000 0000 0
      1010 1010
    0 0000 000
   10 1010 10
+ 101 0101 0
--------------------
 1000 1101 0111 0010
```

Figure 5. 8x8 Multiply of 170 x 213

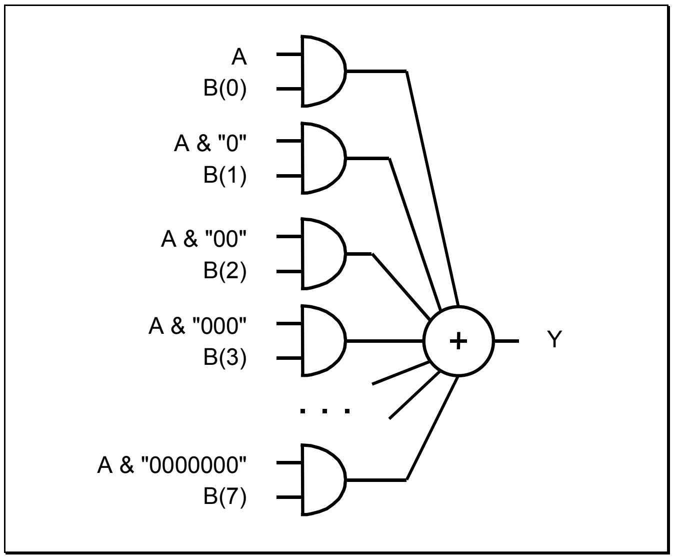

Figure 6. Block Diagram for 8x8 Multiply by Shift and Add

```
architecture Rtl_ShiftAdd2 of Mult is
  constant Z : unsigned (7 downto 0) := "00000000" ;
begin

 Y <=
   (B(0) and (Z(7 downto 0) & A)) +
   (B(1) and (Z(7 downto 1) & A & Z(0 downto 0))) +
   (B(2) and (Z(7 downto 2) & A & Z(1 downto 0))) +
   (B(3) and (Z(7 downto 3) & A & Z(2 downto 0))) +
   (B(4) and (Z(7 downto 4) & A & Z(3 downto 0))) +
   (B(5) and (Z(7 downto 5) & A & Z(4 downto 0))) +
   (B(6) and (Z(7 downto 6) & A & Z(5 downto 0))) +
   (B(7) and (Z(7 downto 7) & A & Z(6 downto 0))) ;

end Rtl_ShiftAdd2 ;
```

Figure 7. VHDL Code for 8x8 Multiplier by Shift and Add

VI. Pipelining Shift and Add

Pipelining a 40x40 multiplier consists of breaking up the 40 input adder into sets of adders and putting a register stage in the middle. Figure 8 shows the multiplier broken 5 sets of 8 input adders. The VHDL code for this is shown in Figure 9.

This coding style is effective and improved the timing, however, it is also tedious and difficult to read.

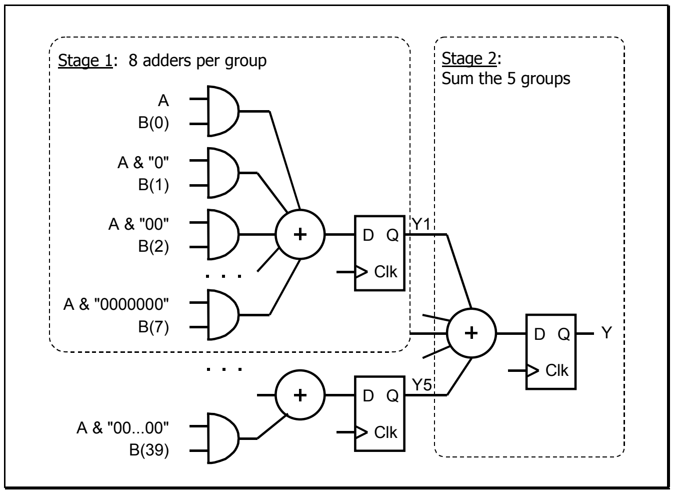

Figure 8. Block Diagram Pipelined Shift and Add

```
PipelinedShiftAdd  :  Process
 constant Z : unsigned(79 downto 0) := (others => '0') ;
begin
  wait until Clk = '1' ;
  -- Stage 1, 5 sets of 8 input adders (Y1 - Y5)
  Y1 <=
    (B(0)  and (Z(7 downto  0) & A)) +
    (B(1)  and (Z(7 downto  1) & A & Z(0 downto 0)) ) +
    (B(2)  and (Z(7 downto  2) & A & Z(1 downto 0)) ) +
    (B(3)  and (Z(7 downto  3) & A & Z(2 downto 0)) ) +
    (B(4)  and (Z(7 downto  4) & A & Z(3 downto 0)) ) +
    (B(5)  and (Z(7 downto  5) & A & Z(4 downto 0)) ) +
    (B(6)  and (Z(7 downto  6) & A & Z(5 downto 0)) ) +
    (B(7)  and (Z(7 downto  7) & A & Z(6 downto 0)) ) ;

  Y2 <=   . . . ;  -- B(15 downto 8)  -- details similar to Y1
  . . .
  Y5 <=   . . . ;  -- B(39 downto 32)

  -- Stage 2, Sum the 5 groups
  Y <=
    ( Z(79 downto 48) & Y1 )  +
    ( Z(79 downto 56) & Y2 & Z( 7 downto  0) ) +
    ( Z(79 downto 64) & Y3 & Z(15 downto  0) ) +
    ( Z(79 downto 72) & Y4 & Z(23 downto  0) ) +
    (                        Y5 & Z(31 downto  0) ) ;
end process ;
```

Figure 9. VHDL Code Pipelined Shift and Add

Table 3. Synthesis Results for Pipelined Shift and Add

Technology	Pipelining Enabled	Frequency (MHz)	Area (cells/LUTs)
Actel	Not Available	20.3	10203
Xilinx	No	69.1	1643
Xilinx	Yes	85.9	1646

VII. Partial Multiply, Shift and Add

The previous code can be simplified by replacing the 8-input adders with a 40x8 multiplier. Pictorially this is shown in Figure 10. The VHDL code is shown in Figure 11. The results are shown in Table 4. This solution is a balance between getting effective results and coding at a high level.

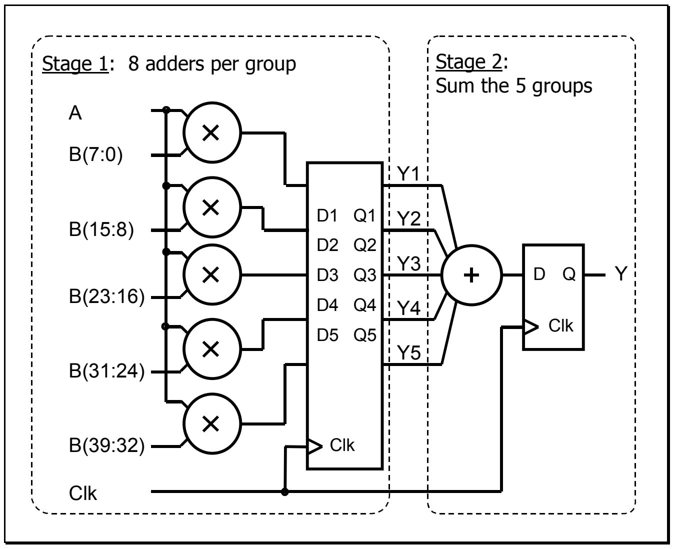

Figure 10. Block Diagram for Partial Multiply, Shift and Add

```
entity PipeMult is
  port (
    Clk          : In  std_logic ;
    A, B          : In  unsigned (39 downto 0) ;
    Y            : Out unsigned (79 downto 0)
  ) ;
end PipeMult ;
architecture Rtl3 of PipeMult is
    signal Y1, Y2, Y3, Y4, Y5 : unsigned (47 downto 0) ;
    constant Z : unsigned (79 downto 0) := (others => '0') ;
begin
  PipeMultProc : process
  begin
      wait until clk = '1' ;
      Y1  <=  A  *  B( 7 downto  0) ;
      Y2  <=  A  *  B(15 downto  8) ;
      Y3  <=  A  *  B(23 downto 16) ;
      Y4  <=  A  *  B(31 downto 24) ;
      Y5  <=  A  *  B(39 downto 32) ;

      Y <=
        (Z(79 downto 48)  &  Y1 )  +
        (Z(79 downto 56)  &  Y2  &  Z( 7 downto 0))  +
        (Z(79 downto 64)  &  Y3  &  Z(15 downto 0))  +
        (Z(79 downto 72)  &  Y4  &  Z( 23 downto  0))  +
        (                        Y5  &  Z( 31 downto  0)) ;
  end process ;

end Rtl3 ;
```

Figure 11. VHDL Code for Partial Multiply, Shift and Add

Table 4. Synthesis Results for Partial Multiply, Shift and Add

Technology	Pipelining Enabled	Critical Timing	Area
Actel	Not Available	17.1	9279
Xilinx	No	69.1	1641
Xilinx	Yes	85.9	1711

VIII. Conclusions

Good coding styles can help a design achieve timing and area goals. There are other tweaks to the code that could be explored to further restructure and optimize the logic. Typically once timing, area, and power constraints are met, the job is done (even if it could be optimized better). Coding for synthesis is intended to get an engineering job done, it is not a science project.

The approach used in this problem generalizes well. Write code based on an initial block diagram of the hardware. If the resulting hardware is not what is needed, refine the block diagram to more exactly represent the desired hardware and re-write the code.

IX. Lessons Learned

In the solution, the final addition (stage 2) presented the most problems and not the first stage multiplier. We will look at two of the issues.

A. Make Addition Operands Identical in Size

Write code with left side of operands padded as shown in Figure 12. Do not write code with different sized operands as shown in Figure 13. Different sized operands can result in a string of adders as shown in Figure 14. Note this is legal VHDL code since the result size is based on the largest array operand. Synthesis results are shown in Table 5.

```
iY <=
       ( Z(79 downto 48) & Y1 )  +
       ( Z(79 downto 56) & Y2 & Z( 7 downto  0) ) +
       ( Z(79 downto 64) & Y3 & Z(15 downto  0) ) +
       ( Z(79 downto 72) & Y4 & Z(23 downto  0) ) +
       (                       Y5 & Z(31 downto  0) ) ;
```

Figure 12. Write code with left side of operands padded.

```
iY <=
       Y1  +
       (Y2  &  Z( 7 downto 0))  +
       (Y3  &  Z(15 downto 0))  +
       (Y4  &  Z( 23 downto  0))  +
       (Y5  &  Z( 31 downto  0)) ;
```

Figure 13. Do not write code with different sized operands

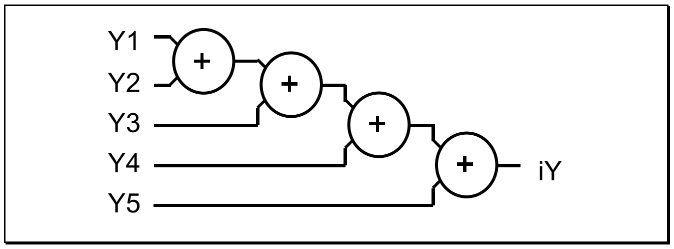

Figure 14. Implementation due to different sized operands

Table 5. Synthesis Results for Multipler between Two Registers

Technology	Actel Freq/Area	Xilinx	Xilinx + Pipe
Equal Sized Ops	17.1 / 9279	69.1 / 1641	85.9 / 1711
Different Sized Ops	12.9 / 9792	61.9 / 1632	93.3 / 1702

B. Unstructured vs. Structured Arithmetic

As a starting point, write unstructured arithmetic operations as shown previously in Figure 12. Structuring the arithmetic operations based on timing can improve results. For most consistent results, use intermediate signals to create structure as shown in Figure 15. Resulting hardware implementation is shown in Figure 16. Synthesis results are shown in Table 6.

```
iY12 <= Y1 +   (Y2 & ZERO(7 downto 0)) ;
iY45 <= Y4 +   (Y5 & ZERO(7 downto 0)) ;
iY35 <= Y3 +   (iY45 & ZERO(7 downto 0)) ;
iY     <= iY12 + (iY35 & ZERO(15 downto 0)) ;
```

Figure 15. Use Intermediate Signals to Structure Code

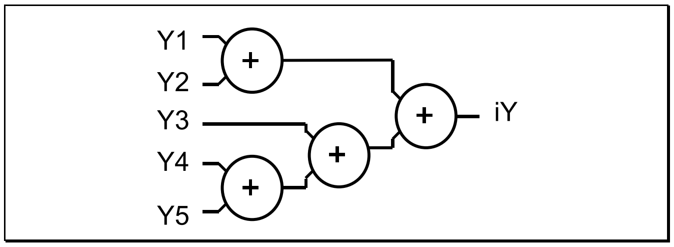

Figure 16. Implementation Using Intermediate Signals

Table 6. Impact of Structuring on Synthesis Results

Technology	Actel Freq/Area	Xilinx	Xilinx + Pipe
Structure, Intermediate Signals	16.3 / 9841	79.8 / 1656	93.3 / 1726
Structure, Parentheses	17.1 / 9680	70.0 / 1640	88.0 / 1710
Unstructured, Good	17.1 / 9279	69.1 / 1641	85.9 / 1711
Unstructured, Bad	12.9 / 9792	61.9 / 1632	93.3 / 1702

C. Overloaded AND Function

The two overloaded "AND" functions shown in Figure 17 permit "AND"ing of std_logic with unsigned that was shown in the Shift-Add solutions.

```
function "and" (L: std_ulogic ; R: unsigned
) return unsigned is
begin
  case L is
    when '0' | 'L' =>       return (R'range => '0');
    when '1' | 'H' =>       return R ;
    when others  =>         return (R'range => 'X');
  end case
end ;

function "and" (L: unsigned ; R: std_ulogic
) return unsigned is
begin
  case R is
    when '0' | 'L' =>       return (L'range => '0');
    when '1' | 'H' =>       return L ;
    when others  =>         return (L'range => 'X');
  end case
end ;
```

Figure 17. Overloaded And Functions

IV. ACKNOWLEDGEMENTS

I would like to thank the students of my intermediate and advanced VHDL training classes for bringing me interesting problems to help them with. In particular, I would like to thank Sundara Murthy for bringing me a block diagram of a Hilbert transform (which provided the basis for this paper).

All trademarks are property of their respective owners.