Coding a 40x40 Pipelined Multiplier in VHDL

by
Jim Lewis
Director of Training, SynthWorks Design Inc
Jim@SynthWorks.com

Coding a 40x40 Multiplier

- Goal
- How Fast is a 40x40 Multiplier?
- Visualizing Pipelining
- Synthesis Tool Representation
- Multiply by Shift and Add
- Pipelining Shift and Add
- Partial Multiply and Add
- Results Summary
- Lessons Learned
Goal

- Pipeline a 40x40 Multiplier to improve performance
- Find an effective VHDL coding style for pipelining.

Issues

- Some synthesis tools automate pipelining, but not for all FPGA vendors.
- Some synthesis tools only support pipelining for their high end tools.

How fast is a 40x40 Multiplier?

VHDL Code

```vhdl
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;
```

Hardware Equivalent

```
\[ \begin{array}{c}
\text{A} \\
\times \\
\text{B} \\
\text{Y}
\end{array} \]
```

```vhdl```

```vhdl```
How fast is a 40x40 Multiplier?

<table>
<thead>
<tr>
<th></th>
<th>Actel</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>12.7</td>
<td>50.7</td>
</tr>
<tr>
<td>Area (cells / LUTs)</td>
<td>6903</td>
<td>1666</td>
</tr>
</tbody>
</table>

Parts & Tools

- Actel part = ProAsic Plus, APA450
- Xilinx part = Vertex-E XCV200E-8 FG256
- Synthesis Tool = Synplicity’s SynplifyPro™

Caution

- The above results come directly from a synthesis tool.
- This timing is an estimate and is only for typical operating conditions.
- While this is good enough to compare coding styles, it is not good enough to predict actual design performance.

Visualizing Pipelining

- Pipelining adds latency to achieve a higher frequency
- Goal is to put approximately half of the multiplier before the first register and approximately half after it.
When a multiplier is coded followed by two registers, a synthesis tool which supports pipelining (register balancing) will move part of the multiplier to the right side of the first register.

This will result in a faster design.

Note: Hardware shown in box is generated by the code shown in the box.

For Xilinx and Altera, this technique is simple and effective.

For Actel, the synthesis tool did not have the capability to add pipelining.
**VHDL Coding Methodology**

- Draw a block diagram for the hardware
- Use the block diagram as a flow chart

**VHDL Problem Solving**

- Identify the problematic structure
  - 40x40 Multiplier
- Break it down into its sub-components
  - Multiply = Shift and Add
- Re-code the design using the new visualization

---

**Multiplication by Shift and Add**

**8x8 Multiply**

\[
\begin{array}{cccc}
1010 & 1010 \\
x & 1101 & 0101 \\
\hline
1010 & 1010 \\
0 & 0000 & 000 \\
10 & 1010 & 10 \\
000 & 0000 & 0 \\
1010 & 1010 \\
0 & 0000 & 000 \\
10 & 1010 & 10 \\
+ & 101 & 0101 & 0 \\
\hline
1000 & 1101 & 0111 & 0010
\end{array}
\]

**Hardware Visualization**
**Multiplication By Shift and Add**

**VHDL Code for 8x8 Multiply**

```vhdl
architecture Rtl_ShiftAdd2 of Mult is
  constant ZERO : unsigned (7 downto 0) := "00000000";
begin
  Y <=
    (B(0) and (ZERO(7 downto 0) & A)) +
    (B(1) and (ZERO(7 downto 1) & A & ZERO(0 downto 0))) +
    (B(2) and (ZERO(7 downto 2) & A & ZERO(1 downto 0))) +
    (B(3) and (ZERO(7 downto 3) & A & ZERO(2 downto 0))) +
    (B(4) and (ZERO(7 downto 4) & A & ZERO(3 downto 0))) +
    (B(5) and (ZERO(7 downto 5) & A & ZERO(4 downto 0))) +
    (B(6) and (ZERO(7 downto 6) & A & ZERO(5 downto 0))) +
    (B(7) and (ZERO(7 downto 7) & A & ZERO(6 downto 0)))
end Rtl_ShiftAdd2;
```

Note: "AND" shown above is overloaded to allow std_logic with unsigned. Implementation of this "AND" is shown at the end.

---

**Pipelining Shift and Add**

- 40x40 Multiply = 40 input Adder
- Partition the adder into 5 groups of 8 input adders

**Stage 1:** 8 adders per group

- A
- B(0)
- A & "0"
- B(1)
- A & "00"
- B(2)
- A & "0000000"
- B(7)

**Stage 2:** Sum the 5 groups

- Y1
- D Q
- Clk
- Y
- D Q
- Clk
- Y5
- D Q
- Clk
PipelinedShiftAdd : Process

constant Z : unsigned (79 downto 0) := (others => '0') ;
begin
  wait until Clk = '1' ;
  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)

end process ;

Pipelining Shift and Add

<table>
<thead>
<tr>
<th></th>
<th>Actel</th>
<th>Xilinx</th>
<th>Xilinx + Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>20.3</td>
<td>69.1</td>
<td>85.9</td>
</tr>
<tr>
<td>Area (cells / LUTs)</td>
<td>10203</td>
<td>1646</td>
<td>1646</td>
</tr>
</tbody>
</table>

- Coding style is effective and improved timing (and area) for all scenarios
- Coding style is quite tedious and difficult to read.
Partial Multiply and Add

Shift and Add

- The 8 adders on the left are equivalent to $A \times B(7:0)$

Partial Multiply

Stage 1: Multiply $40 \times 8$

Stage 2: Sum results of partial multiply
architecture Rtl3 of PipeMult is

  signal Y1, Y2, Y3, Y4, Y5 : unsigned (47 downto 0);
  constant ZERO : unsigned(31 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 <=
    (ZERO(31 downto 0) & Y1) +
    (ZERO(31 downto 8) & Y2 & ZERO( 7 downto 0)) +
    (ZERO(31 downto 16) & Y3 & ZERO(15 downto 0)) +
    (ZERO(31 downto 24) & Y4 & ZERO(23 downto 0)) +
    (Y5 & ZERO(31 downto 0));
  end process;
end Rtl3;

Partial Multiply and Add

<table>
<thead>
<tr>
<th></th>
<th>Actel</th>
<th>Xilinx</th>
<th>Xilinx + Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>17.1</td>
<td>69.1</td>
<td>85.9</td>
</tr>
<tr>
<td>Area (cells / LUTs)</td>
<td>9279</td>
<td>1641</td>
<td>1711</td>
</tr>
</tbody>
</table>

- Coding style is effective.
- Results similar to Pipelined Shift and Add
- Coding style is easier to read (than Shift and Add)
Results Summary

<table>
<thead>
<tr>
<th></th>
<th>Actel</th>
<th></th>
<th>Xilinx</th>
<th></th>
<th>Xilinx + pipeline</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Freq</td>
<td>Area</td>
<td>Freq</td>
<td>Area</td>
<td>Freq</td>
<td>Area</td>
</tr>
<tr>
<td>Reference Multiplier</td>
<td>12.7</td>
<td>6903</td>
<td>50.7</td>
<td>1666</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool Preferred</td>
<td>12.6</td>
<td>7063</td>
<td>49.7</td>
<td>1665</td>
<td>79.8</td>
<td>2238</td>
</tr>
<tr>
<td>Shift Add</td>
<td>20.3</td>
<td>10203</td>
<td>69.1</td>
<td>1646</td>
<td>85.9</td>
<td>1646</td>
</tr>
<tr>
<td>Partial Multiply</td>
<td>17.1</td>
<td>9279</td>
<td>69.1</td>
<td>1641</td>
<td>85.9</td>
<td>1711</td>
</tr>
</tbody>
</table>

Conclusion

- Good coding styles can help a design achieve timing and area goals

Lessons Learned

- Make Operands Identical in Size
- Unstructured vs. Structured Equations
- Nonpreferred Synthesis Tool Coding Styles
- Overloaded "AND" for std_logic with unsigned
Make Operands Identical in Size

- Write code with left side of operands padded:

  ```
iY <=
  (ZERO(31 downto 0) & Y1) +
  (ZERO(31 downto 8) & Y2 & ZERO(7 downto 0)) +
  (ZERO(31 downto 16) & Y3 & ZERO(15 downto 0)) +
  (ZERO(31 downto 24) & Y4 & ZERO(23 downto 0)) +
  (Y5 & ZERO(31 downto 0));
  ```

- Do not write code with different sized operands*

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

*Legal since result size is based on largest array operand

---

Make Operands Identical in Size

- Implementation due to using different sized operands:

  ![Diagram](image)

<table>
<thead>
<tr>
<th>Results</th>
<th>Actel</th>
<th>Xilinx</th>
<th>Xilinx + pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Freq</td>
<td>Area</td>
<td>Freq</td>
</tr>
<tr>
<td>Equal Sized Operands</td>
<td>17.1</td>
<td>9279</td>
<td>69.1</td>
</tr>
<tr>
<td>Different Sized Operands</td>
<td>12.9</td>
<td>9792</td>
<td>61.9</td>
</tr>
</tbody>
</table>
Structured vs Unstructured Code

- Unstructured arithmetic is a good starting point:
  
  ```vhdl
  iY <=
  (ZERO(31 downto 0)  & Y1  ) +
  (ZERO(31 downto 8)  & Y2  & ZERO( 7 downto 0)) +
  (ZERO(31 downto 16) & Y3  & ZERO(15 downto 0)) +
  (ZERO(31 downto 24) & Y4  & ZERO(23 downto 0)) +
  (                     Y5 & ZERO(31 downto 0)) ;
  ```

  ![Good Implementation Diagram]

  ![Bad Implementation Diagram]

  *Results will vary with tools. Analyze using RTL View*

Structured vs Unstructured Code

- Coding structure based on timing can improve results:

  ```vhdl
  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)) ;
  ```

  ![Resulting Implementation Structure]

  *Parentheses sometimes are good for structuring, but they are not as reliable as intermediate signals (shown above)*
Structured vs Unstructured Code

<table>
<thead>
<tr>
<th>Results, Partial Multiply</th>
<th>Actel Freq</th>
<th>Area</th>
<th>Xilinx Freq</th>
<th>Area</th>
<th>Xilinx + pipeline Freq</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured, Signals</td>
<td>16.3</td>
<td>9841</td>
<td>79.8</td>
<td>1656</td>
<td>93.3</td>
<td>1726</td>
</tr>
<tr>
<td>Structured, Parentheses</td>
<td>17.1</td>
<td>9680</td>
<td>70.0</td>
<td>1640</td>
<td>88.0</td>
<td>1710</td>
</tr>
<tr>
<td>Unstructured, Good</td>
<td>17.1</td>
<td>9279</td>
<td>69.1</td>
<td>1641</td>
<td>85.9</td>
<td>1711</td>
</tr>
<tr>
<td>Unstructured, Bad?</td>
<td>12.9</td>
<td>9792</td>
<td>61.9</td>
<td>1632</td>
<td>93.3</td>
<td>1702</td>
</tr>
</tbody>
</table>

- Start with unstructured code
- If timing dictates, use timing reports to structure code.
  - Intermediate signals can be better than parentheses

Non-Preferred Tool Coding Styles

<table>
<thead>
<tr>
<th></th>
<th>Xilinx with Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Above Structure</td>
<td>79.8</td>
</tr>
<tr>
<td>Multiplier followed by two registers</td>
<td>79.8</td>
</tr>
</tbody>
</table>

- *Results may vary with tools.
- Recommend use the coding style shown on slide 7
Overloaded "AND"

- The following functions make the "AND" of std_logic with unsigned done in the Shift-Add solutions possible.

```vhdl
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 ;
```

```vhdl
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 ;
```

Note: Revisions to std_logic_1164 and Numeric_Std have been proposed.

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).

- I would like to thank Jeff Garrison of Synplicity for allowing me to publish the results from the SynplicityPro™ synthesis tool.

- All trademarks are property of their respective owners
**About SynthWorks VHDL Training**

Comprehensive VHDL Introduction 4 Days
[http://www.synthworks.com/comprehensive_vhdl_introduction.htm](http://www.synthworks.com/comprehensive_vhdl_introduction.htm)
Engineers learn VHDL Syntax plus basic RTL coding styles and simple procedure-based, transaction testbenches. Our designer focus ensures that your engineers will be productive in a VHDL design environment.

VHDL Coding Styles for Synthesis 4 Days
[http://www.synthworks.com/vhdl_rtl_synthesis.htm](http://www.synthworks.com/vhdl_rtl_synthesis.htm)
Engineers learn RTL (hardware) coding styles that produce better, faster, and smaller logic.

VHDL Testbenches and Verification 3 days
[http://www.synthworks.com/vhdl_testbench_verification.htm](http://www.synthworks.com/vhdl_testbench_verification.htm)
Engineers learn how create a transaction-based verification environment based on bus functional models.

For additional courses see: [http://www.synthworks.com](http://www.synthworks.com)