Coding a 40x40 Pipelined Multiplier in VHDL

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

Coding a 40x40 Multiplier

SynthWorks

- 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

SynthWorks

- 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.

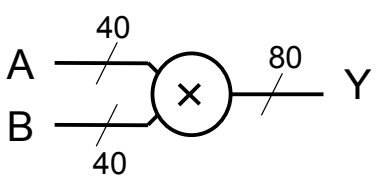
How fast is a 40x40 Multiplier?

SynthWorks

VHDL Code

```
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



How fast is a 40x40 Multiplier?

SynthWorks

	Actel	Xilinx
Frequency (MHz)	12.7	50.7
Area (cells / LUTs)	6903	1666

Parts & Tools

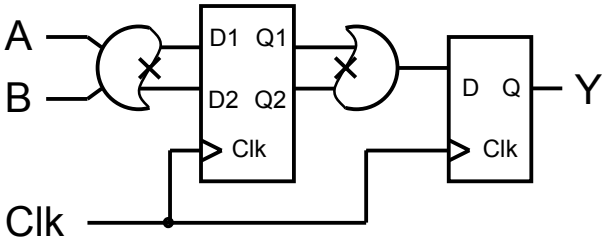
- Actel part = ProAsic Plus, APA450
- Xilinx part = Vertex-E XCV200E-8 FG256
- Synthesis Tool = Synplicity's SynplifyPro™

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

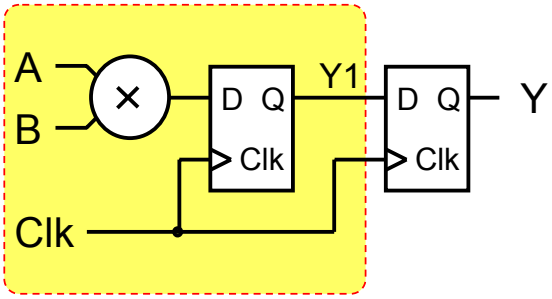
SynthWorks



- 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.

Synthesis Tool Representation

SynthWorks




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

- 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

---

Synthesis Tool Representation

SynthWorks

	Actel	Xilinx	Xilinx + Pipelining
Frequency (MHz)	12.6	49.7	79.8
Area (cells / LUTs)	7063	1665	2238

- 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

SynthWorks

- 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

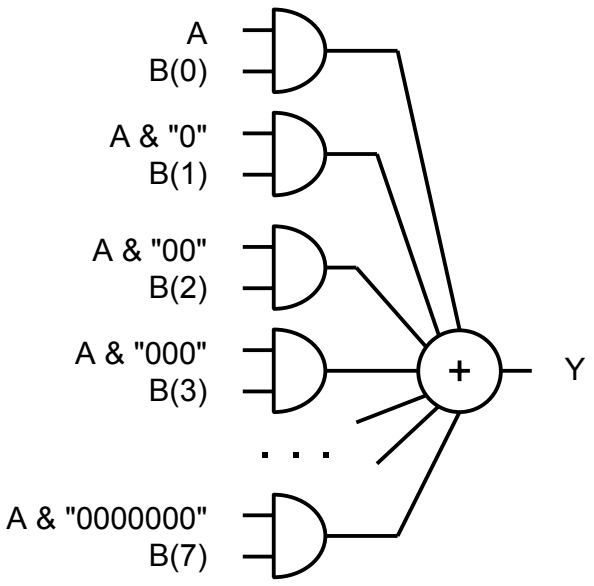
Multiplication by Shift and Add

SynthWorks

8x8 Multiply

```
            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
```

Hardware Visualization



Multiplication By Shift and Add

SynthWorks

VHDL Code for 8x8 Multiply

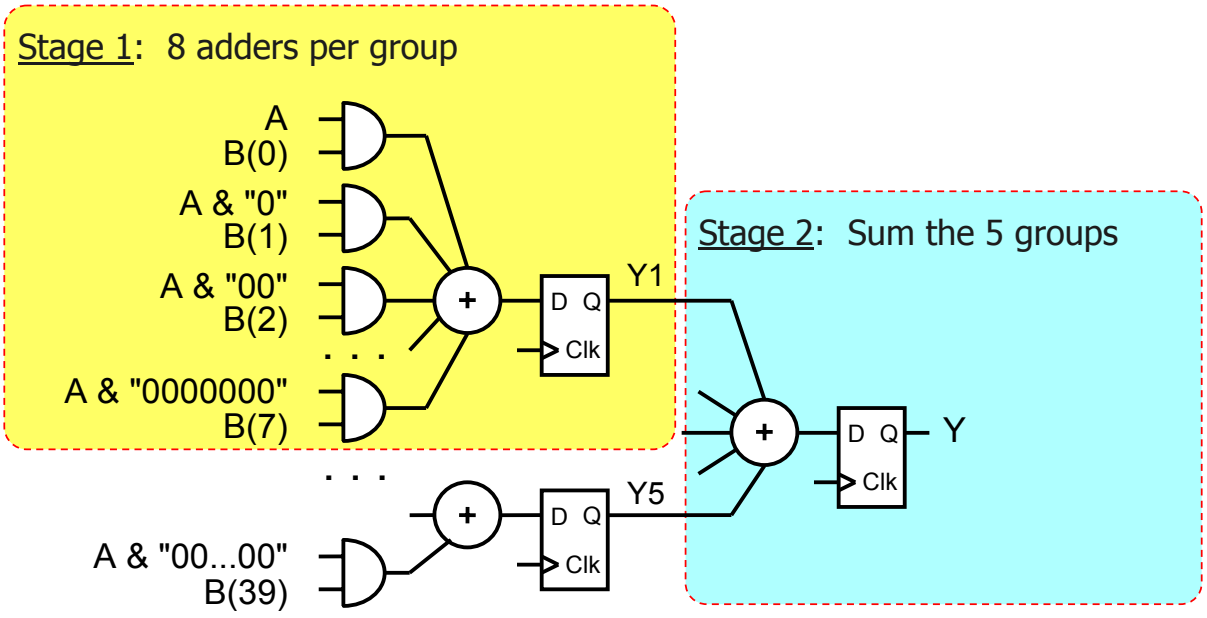
```
architecture Rtl_ShiftAdd2 of Mult is
  constant ZERO : unsigned (7 downto 0) := "00000000" ;
begin

 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

SynthWorks

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



PipelinedShiftAdd : Process

SynthWorks

```
 constant Z : unsigned (79 downto 0) := (others => '0') ;
begin
  wait until Clk = '1' ;
  Y1 <=                                                    Stage 1: 8 adders per group
    (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)
  Y <=                                                     Stage 2: Sum the 5 groups
    ( 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 ;
```

Pipelining Shift and Add

SynthWorks

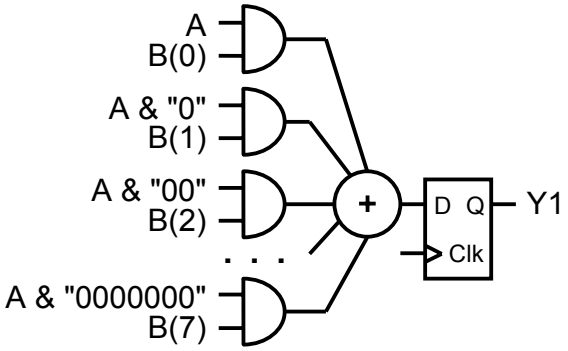
	Actel	Xilinx	Xilinx + Pipelining
Frequency (MHz)	20.3	69.1	85.9
Area (cells / LUTs)	10203	1646	1646

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

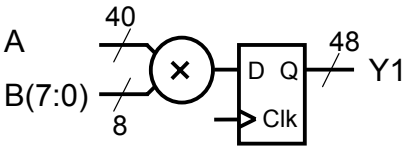
Partial Multiply and Add

SynthWorks

Shift and Add



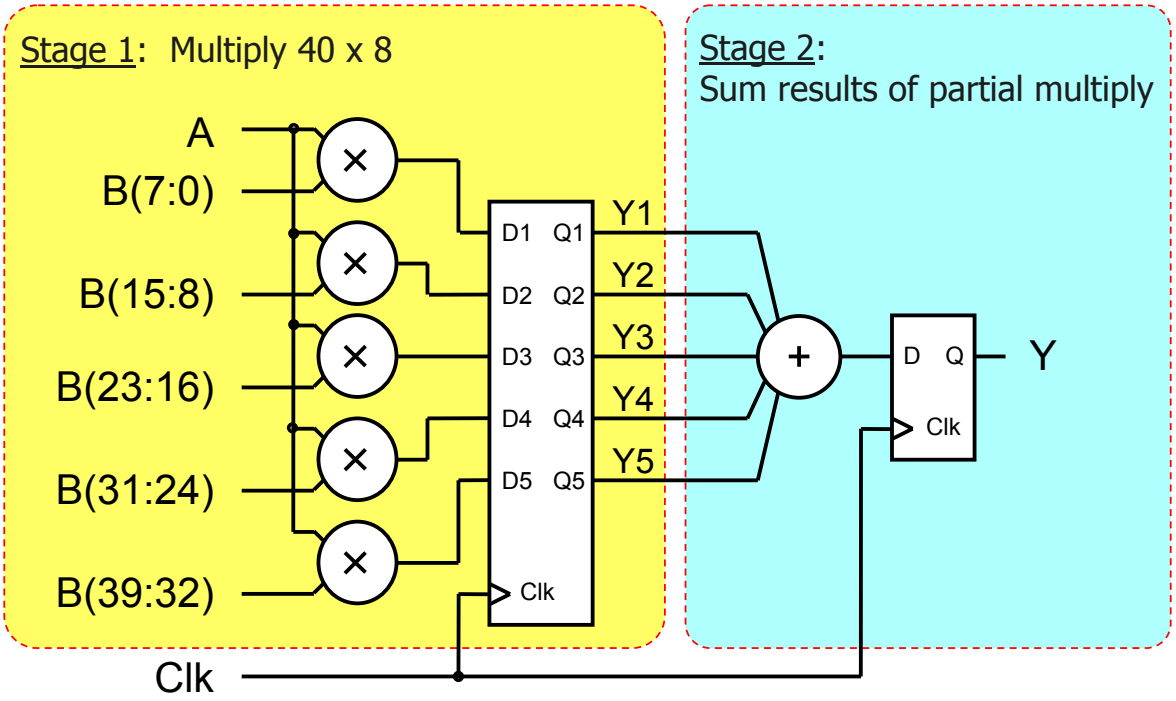
Partial Multiply



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

Partial Multiply and Add

SynthWorks



```
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' ;
```

Stage 1: Multiply 40 x 8

```
    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) ;
```

Stage 2: Sum the results of the partial multiply

```
    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

	Actel	Xilinx	Xilinx + Pipelining
Frequency (MHz)	17.1	69.1	85.9
Area (cells / LUTs)	9279	1641	1711

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

Results Summary

SynthWorks

	Actel		Xilinx		Xilinx + pipeline	
	Freq	Area	Freq	Area	Freq	Area
Reference Multiplier	12.7	6903	50.7	1666		
Tool Preferred	12.6	7063	49.7	1665	79.8	2238
Shift Add	20.3	10203	69.1	1646	85.9	1646
Partial Multiply	17.1	9279	69.1	1641	85.9	1711

Conclusion

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

---

Lessons Learned

SynthWorks

- 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

SynthWorks

- 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*

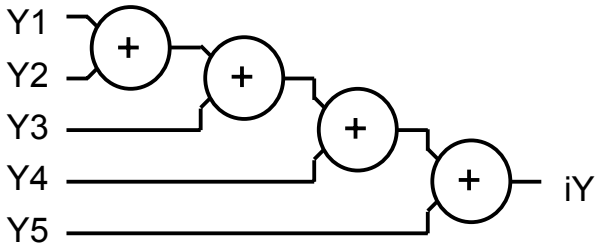
```
iY <=
   Y1  +
  (Y2 & ZERO( 7 downto  0)) +
  (Y3 & ZERO(15 downto  0)) +
  (Y4 & ZERO(23 downto  0)) +
  (Y5 & ZERO(31 downto  0)) ;
```

- *Legal since result size is based on largest array operand

Make Operands Identical in Size

SynthWorks

- Implementation due to using different sized operands:



Results	Actel		Xilinx		Xilinx + pipeline	
	Freq	Area	Freq	Area	Freq	Area
Equal Sized Operands	17.1	9279	69.1	1641	85.9	1711
Different Sized Operands	12.9	9792	61.9	1632	93.3	1702

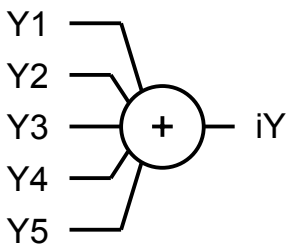
Structured vs Unstructured Code

SynthWorks

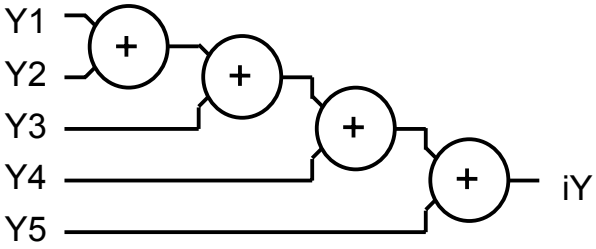
- Unstructured arithmetic is a good starting point:

```
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



Bad Implementation



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

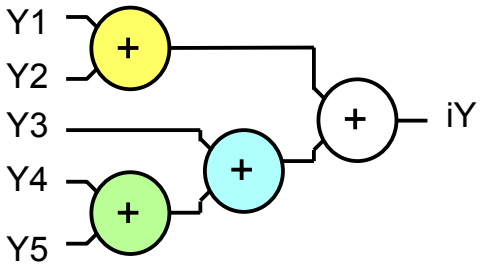
Structured vs Unstructured Code

SynthWorks

- Coding structure based on timing can improve results:

```
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

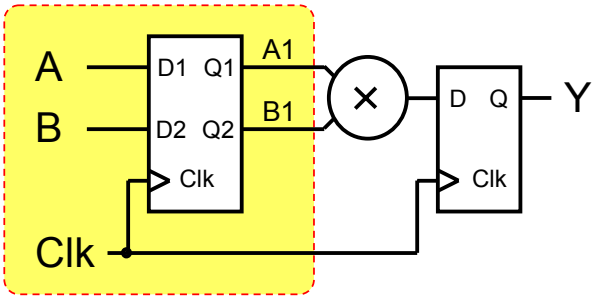
SynthWorks

Results, Partial Multiply	Actel		Xilinx		Xilinx + pipeline	
	Freq	Area	Freq	Area	Freq	Area
Structured, Signals	16.3	9841	79.8	1656	93.3	1726
Structured, 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

- 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

SynthWorks




```
PipeMultProc : process
begin
  wait until Clk = '1' ;
  A1 <= A ;
  B1 <= B ;
  Y <= A1 * B1 ;
end process ;
```

	Xilinx with Pipelining	
	Frequency	Area
Above Structure	79.8	2238
Multiplier followed by two registers	79.8	2238

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

Overloaded "AND"

SynthWorks

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

```
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 ;
```

Note: Revisions to std_logic_1164 and Numeric_Std have been proposed

---

Acknowledgements

SynthWorks

- 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.

About SynthWorks VHDL Training

Comprehensive VHDL Introduction 4 Days
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
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
Engineers learn how create a transaction-based verification environment based on bus functional models.

For additional courses see: http://www.synthworks.com