Design Tips for HDL Implementation of Arithmetic Functions

Author: Steven Elzinga, Jeffrey Lin, and Vinita Singhal

XILINX®

XAPP215 (v1.0) June 28, 2000

Summary

This application note provides design advice for implementing arithmetic logic functions in two High-Level Design Languages (HDLs), VHDL and Verilog.

Introduction

This application note discusses design considerations for HDL coding of simple arithmetic functions in Virtex™ devices. HDL code examples for implementing adders, subtracters, two's complementers, comparators and multipliers are provided in the reference design. Because it is without primitive instantiations, the HDL code is portable across the Virtex device families.

The strong connection between synthesis tools and HDL coding styles is discussed. The HDL code provided in the reference design files is available by downloading **XAPP215.zip** or **XAPP215.tar.gz**. Three different synthesis tools were used to gauge the effect of the code on expected results: Synopsys FPGA Express v3.3, Synplicity Synplify 5.2.2a, and Exemplar Leonardo Spectrum v1999.1i.

The focus on arithmetic functions is due to their common usage in Digital Signal Processing (DSP) based designs. DSP is increasingly being used in wireless applications and involve a large number of repetitive arithmetic operations. For the best, high-performance utilization of the FPGA, the fewest logic cells with a minimum delay must be used by these arithmetic operations. The Virtex series employs a powerful Configurable Logic Block (CLB) architecture with the requisite speed, utilization and re-programmability advantages. The Virtex architecture combined with HDL coding guidelines help to achieve the target performance.

Virtex CLB

For the best performance and to evaluate the performance of the synthesis tool, the Virtex architecture must be understood. An explanation of the Virtex congifurable logic block (CLB) is appropriate. The basic building block of the Virtex CLB is the logic cell (LC). In Figure 1, the CLB contains four LCs organized as two slices. Figure 2 shows a more detailed view of a single slice (i.e. half the CLB).

An LC includes a 4-input function generator, carry logic and a storage element. The function generator is implemented as a 4-input look-up table (LUT) and can implement ANY 4-input logic function. Furthermore, the two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM or 16 x 1-bit dual-port synchronous RAM. A Virtex LUT can also implement a 16-bit shift register to capture high-speed data or burst-mode data.

The output from the 4-input LUT in each LC drives both the CLB output and the D-input of the flip-flop. Each additional 2-input dedicated AND gate per LUT implements an efficient 1-bit multiplier. The Virtex CLB is explained in detail in the **Virtex data sheet**.
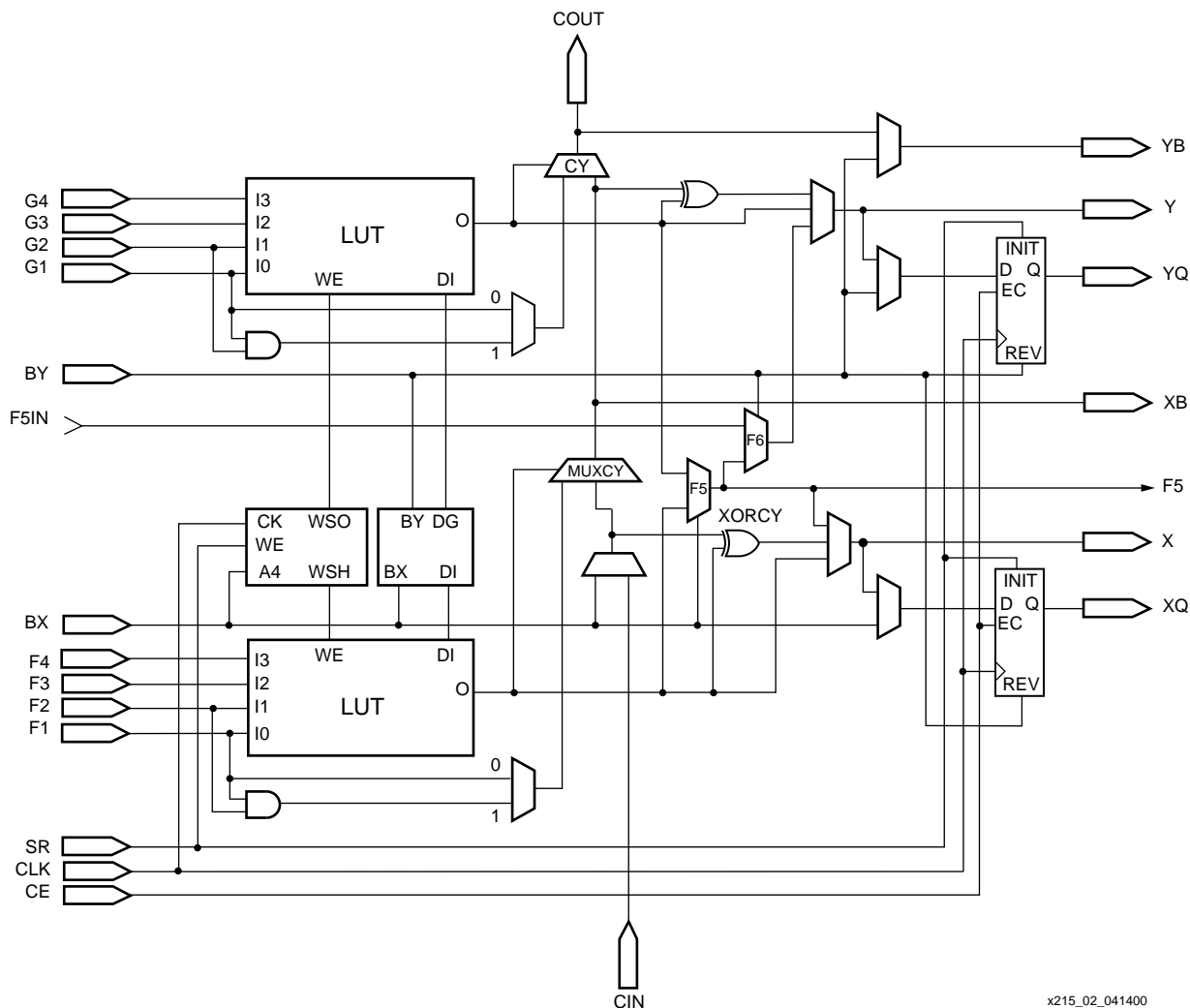
XILINX®
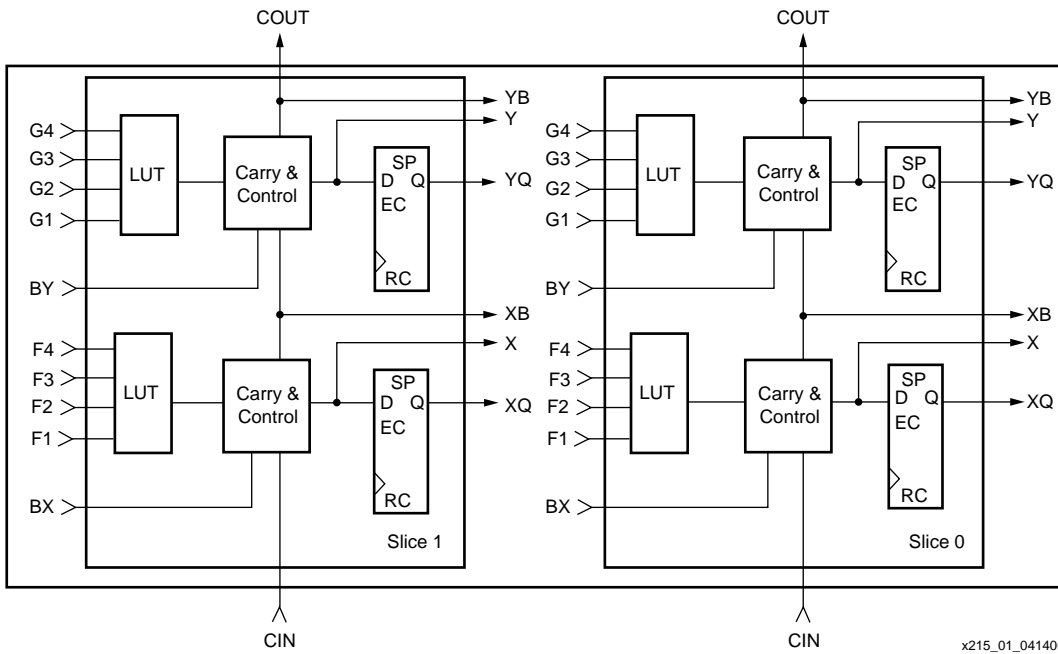
Figure 1: **2-Slice Virtex CLB**

Figure 2: **Block Diagram of a Single-Slice Virtex CLB**

Virtex Carry Logic

The most relevant feature of the CLB is the dedicated carry logic to implement fast, efficient arithmetic functions. Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. Synthesis tools leveraging this carry logic produce fast and compact designs. There are two separate carry chains in the Virtex CLB, one per slice. The height of the carry chains is two bits per CLB. The logic consists of a 2-input MUX (MUXCY) and an XOR (XORCY) gate. The XOR gate allows a 1-bit full adder to be implemented within a logic cell (LC). In addition, a dedicated AND gate improves the efficiency of the multiplier implementation. The dedicated carry path is used to cascade LUT functions for implementing wide logic functions. This reduces logic delays due to the decreased number of logic levels even for very high fan-in functions.
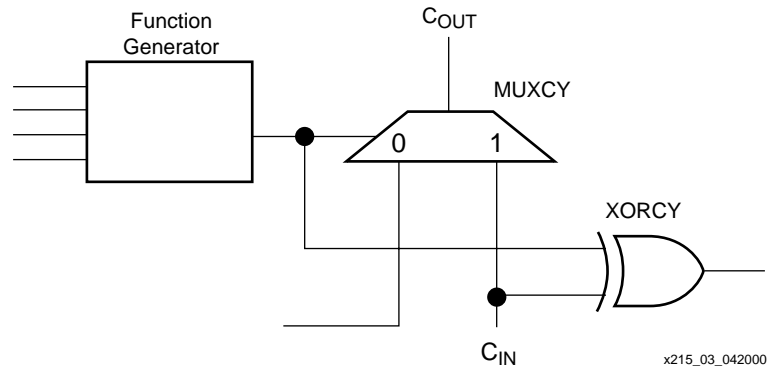
Figure 3: **Carry Logic Diagram**

Design Considerations for HDL Coding

When designing with HDL it is important to consider coding style and synthesis. By using synthesis vendor recommended coding style, the designer can control the synthesis results. This section discusses the optimum implementation for coding the arithmetic functions of a full-featured arithmetic module. The following VHDL code is for a synchronous, resetable, setable, loadable, clock-enabled, adder/subtractor arithmetic module.

```
if clk'event and clk = '1' then
   -- if the reset is asserted, set the counter value to zeros
   if reset = '1' then
     sum <= (others => '0');
   -- if the set is asserted, set the counter value to ones
   elsif set = '1' then
     sum <= (others => '1');
   -- if the load signal is asserted, load what is on the load input
   elsif load = '1' then
     sum <= load_value;
   -- finally, if the adder/subtractor is enabled, do the operation,
   -- which is dependant on the subtract input signal.
   elsif enable = '1' then
     if subtract = '1' then
       sum <= ('0'&a) - ('0'&b);
     else
       sum <= ('0'&a) + ('0'&b);
     end if;
   end if;
 end if;
```

In the Virtex architecture, if either a set or reset signal is used, the prior circuit is implemented at 2-bits per slice (half a CLB). In Table 1 and Figure 4, a creative use of the LOAD signal forces a "0" on the carry chain allowing the carry-out "C_{OUT}" signal to remain unaffected during a load operation. The carry out of the 2-bit adder/subtractor is computed with no additional logic overhead. A "0" on the input of the XOR gate in the carry-chain logic loads the desired value into the registers. The SUBTRACT and LOAD signals and related logic are absorbed into the 4-input LUT without additional logic overhead. Since in the Virtex architecture each LUT has a dedicated register, registering arithmetic operations yield synchronous pipelined designs at no extra register cost. A synchronous, resetable (or setable), loadable, clock-enabled, adder/subtractor compact design is implemented in one logic level. This translates into higher-performance since fewer logic levels allows designs to run at higher clock frequencies.

Table 1:

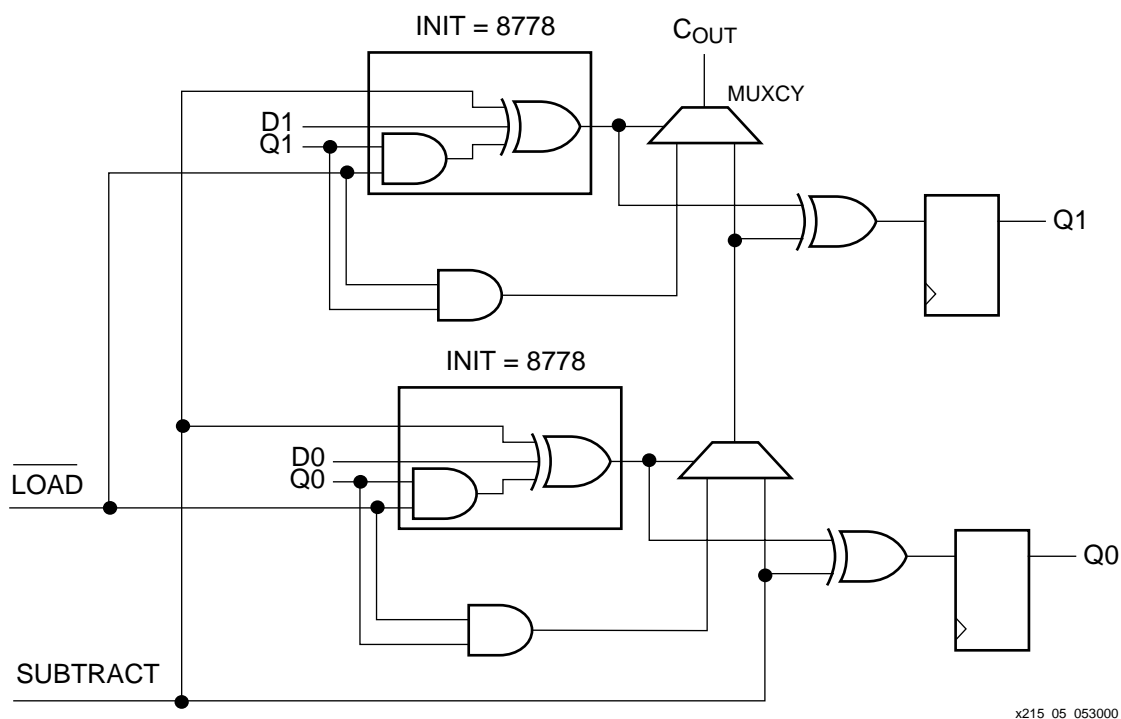| LOAD	SUBTRACT	Function
0	0	Q <= D
0	1	Q <= –D
1	0	Q <= Q + D
1	1	Q <= Q – D

Figure 4:

Use of Set and Reset

Typically, designs use either a set or a reset for initializing registers which are easily absorbed into the dedicated set/reset pin on the flip-flops in the CLB. An application that uses both set and reset could yield a larger circuit, depending upon the number of inputs each bit slice requires to produce an output. When set and reset (synchronous) are both specified, the set/reset logic is synthesized to use the LUT logic feeding the data input of the flip-flop. This leaves two inputs for the logic function feeding the data. If the logic function already has three or more input signals, it will be implemented in two LUTs thus increasing the number of logic levels by one.

Signed and Unsigned Values

Any arithmetic function created can take signed and/or unsigned numbers as operands. When dealing with unsigned numbers, both Verilog and VHDL handle them with equal ease. The two languages differ in signed arithmetic support.

In the case of synthesizable Verilog, every number is treated as an unsigned value. All inferred arithmetic functions will be created for unsigned values and operate correctly without any special number handling. Signed numbers are handled differently. In Verilog, only values of

type "integer" or "real" can be processed as negative numbers. Both signed and unsigned types are at least 32-bits wide.

Verilog

All exceptions in Verilog for variable sized signed numbers must be handled by the designer. Verilog does not automatically handle signed numbers. Despite the fact that placing a negative sign ("-") in front of a variable automatically makes it a two's complement of the original number, these negative numbers will not be processed correctly by inferred operations. For example, if the following is used to infer a magnitude comparator:

    c = (a < b) : 1'b1 ? 1'b0;

"a" "b" are to be treated as signed numbers, the output, "c", will be incorrect.

The designer must manually handle the exceptions of dealing with signed numbers in verilog, since there is no generic way to create minimum sized circuits. Also, special methods of handling one type of function, may not work equally well for a different function.

Example: Signed Numbers

```
// This example is of a signed magnitude comparator
module mag_comp_sign (a, b, a_gtet_b);
input [7:0]a;
input [7:0]b;
output a_gtet_b;
reg a_gtet_b;
// make an intermediate variable used to do a subtraction
wire [8:0]intermediate;
assign intermediate = a - b;
always @(a or b or intermediate)
begin
  // if the subtraction is zero they are equal and we can assert
  // the greater than equal to signal.
  if(intermediate == 9'd0)
    a_gtet_b = 1'b1;
  // if the subtraction is not zero, we must find out which number is larger
  else
    begin
      if(a[7] == 1'b1) // check the msb of 'A' to see if it is negative
        begin
        if(b [7]== 1'b0)  // if B is positive, B is larger
          a_gtet_b = 1'b0;
        else
          begin
            // if the sum of the number is negative, B is larger
            // if the sum is positive, A is larger
            // determine sign by checking sign bit, the MSB
            if(intermediate [8] == 1'b1)
              a_gtet_b = 1'b0;
            else
              a_gtet_b = 1'b1;
          end
        end
      else  // since the msb is not a '1', A is positive
        begin
          if(b [7] == 1'b1)  // if B is negative, A is larger
            a_gtet_b = 1'b1;
          else
            begin
              // if the sum is negative, B is larger, else A is larger
              if (intermediate [8] == 1'b1)
                a_gtet_b = 1'b0;
              else
```

www.xilinx.com
1-800-255-7778

```
              a_gtet_b = 1'b1;
           end
        end
     end
  end

  endmodule
```

Because of the nested structure of this example, the circuit produced is not optimal. Optimal circuitry may be created with more task specific coding, however this task specific coding may not work with other function types. This example simply illustrates that when the data types in the design are not integers or real types, the designer should treat any signed numbers as signed numbers. No assumptions can be made about the ability of the inferred logic to work with signed numbers.

VHDL

For VHDL, arithmetic operations with unsigned and signed values are inferred by including the appropriate arithmetic libraries. When designing with signed numbers, the VHDL libraries will create circuits which will handle two's complement numbers.

When dealing with unsigned values, include the following libraries:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
```

When dealing with signed values, include the following:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
```

The IEEE.std_logic_1164.all simply allows for the usage of IEEE definitions such as std_logic signals and std_logic_vector busses. The IEEE.std_logic_arith.all allows the synthesis tools to infer arithmetic functions from operators. The IEEE.std_logic_signed and IEEE.std_logic_unsigned libraries instruct the synthesis tools to create circuits to handle signed or unsigned numbers, respectively. When using VHDL, a line change to have inferred functions for either signed or unsigned values. To ensure that signed circuits are created, the operands to a operator should be declared as a "signed" value.

VHDL Example: Signed Numbers

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity magcomp is

  port (
    a: in signed (7 downto 0);
    b: in signed (7 downto 0);
    a_gtet_b: out std_logic;
  );
end magcomp;

architecture magcomp_arch of magcomp is

begin
  a_gtet_b <= '1' when a >= b else '0';

end magcomp_arch;
```

In this example, the inputs, a and b are defined as "signed" values. The comparator created does a proper comparison for "-1" and "1". If a mixture of both signed and unsigned functions are created, both the IEEE.std_logic_signed and IEEE.std_logic_unsigned libraries should be included. Each signal in the design is declared as a "signed" value or "unsigned" value.

VHDL Example: Signed and Unsigned Numbers

```
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_signed.all;
  use IEEE.std_logic_unsigned.all;

  entity magcomp is

    port (
      a:  in signed (7 downto 0);
      b:  in signed (7 downto 0);
      a2: in unsigned (7 downto 0);
      b2: in unsigned (7 downto 0);
      a_gtet_b:  out std_logic;
      a_gtet_b2: out std_logic
    );
  end magcomp;

  architecture magcomp_arch of magcomp is

  begin

    a_gtet_b  <= '1' when a >= b else '0';
    a_gtet_b2 <= '1' when a2 >= b2 else '0';

  end magcomp_arch;
```

Carry-in

Implementation of the carry-in bit for addition is simpler than a "borrow" for a subtract operation. The HDL coding is shown in the following example.

Example:

VHDL: sum <= (a + b) + c_in;

Verilog: sum = (a + b) + c_in;

In both cases the carry-in (c_in) is only one-bit wide. Most synthesis tools correctly infer the adder and use the carry logic hardware for the carry-in bit. Using parentheses to separate the adder and the carry-in generates clearer code for the desired results.

Overflows and Carry-Outs

Overflows in addition operations must be handled by design. Typically, synthesis tools will not automatically infer circuitry to handle overflows irrespective of the HDL used.

If the result of an accumulator or counter overflows, causing the value to "roll-over" from all "1"s to all "0"s, the design must detect overflow, handle it by designing the overflow logic, and provide for the overflow bit in the HDL code. Overflow detection is also necessary when using signed numbers. Using two's complement numbers, the maximum negative value will always be one more than the maximum positive value. As an example, for an 8-bit signed number, the smallest negative number is -128, and the largest positive number is 127. When calculating the complement -128 the answer (+128) will overflow and the 8-bit result will be zero since the signed representation of +128 takes up 9 bits not 8 bits.

To design for overflow, extend the function to handle one more bit than is necessary. The following examples use 8-bit signed numbers, sign extended as 9-bit signed numbers, so the 9-bit result covers all possibilities.

Example:

VHDL: sum <= (a(7)&a) + (b(7)&b); -- sum is a 9-bit value

Verilog: sum = {a[7],a} + {b[7],b}; // sum is a 9-bit value

The sign extension copies the MSB of the input numbers and places it in front to create a number which has the same value, but more bits.

A carry-out of a function is accessed in the same manner. Extend the inputs and use the MSB of the function output (in the above example, "sum") as the carry out.

Mismatched Sized Inputs

Sign extension for processing different bit-wide operands may be handled automatically by some synthesis tools. Since synthesis tool results vary, Xilinx recommends designing and explicitly coding the sign extension. Much like the case of the overflow, sign extension is carried out by concatenation of the appropriate number of most significant bits (MSBs). For unsigned numbers, an appropriate number of zeroes should be concatenated. For signed numbers, the concatenation should be the MSB of the number, which in turn is the sign bit.

Variable Description:

sum - a 9-bit variable to hold the result of the operation. It is 9-bits to ensure that overflow conditions can be detected.

a - a 6-bit input operand

b - an 8-bit input operand

Example: Unsigned Numbers

For unsigned numbers, the extension should be zeros. This is also called zero-extended.

VHDL concatenation: sum <= ("000"&a) + ('0'&b);

Verilog concatenation: sum = {3'b000,a} + {1'b0,b);

Example: Signed Numbers

For signed numbers, the extension is the value of the sign bit minus the MSB. This is called sign-extension.

VHDL: sum <= (a(7)&a(7)&a(7)&a) + (b(7)&b);

Verilog: sum = {a[7],a[7],a[7],a} + {b[7],b};

Other Considerations

To achieve the highest levels of utilization and performance, carefully consider the architecture of the actual, targeted Virtex device. The number of rows and columns of CLBs in the target device will affect the partitioning of the size of the implemented arithmetic operation. The number of I/O pins on the target device and other available device resources must also be considered.

The Virtex series has dedicated arithmetic hardware (called carry-logic) designed to speed the propagation of critical signals such as carry outs. This dedicated hardware physically runs vertically on the device cascading from one slice to the slice directly above it. The carry-logic hardware terminates at the top of a column. For maximum performance, the carry-logic chain should not extend past the top slice. This will constrain the maximum bit-width or size of the arithmetic function to the number of Virtex device rows.

For example, an XCV50 device has 16 rows and 24 columns, i.e. 16 slices per column. The largest carry-logic chain is 16 slices long. Translating this to the input bit size depends on the arithmetic operation. For example a 2-input unsigned adder will require one Virtex slice for

processing 2-bits from each of the two inputs. The XCV50 with 16 slices per column can implement a fast and efficient 32-bit unsigned adder in a column. Consult the Virtex series data sheet for sizing of other Virtex devices.

Role of Synthesis in HDL Designs

Although specific coding styles yield the expected implementation, there is still some dependency on the synthesis technology. This section reports on some synthesis results obtained for commonly used arithmetic operations, using three major synthesis tools.

Adders and Subtractors

This section examines the add/subtract arithmetic operation in the context of a Virtex CLB. In a binary addition operation, the partial sum is realized by using an XOR gate. Since a 4-input Look-Up Table (LUT) in the Virtex CLB can implement any 4-input operation, each LUT can implement the partial sum of two 2-bit wide operands. The dedicated carry chain logic implements the carry. A subtract operation is similar to the adder implementation, with the second operand being represented as a 2's complement. Again, since the LUT implements any 4-input function, complementing the second operand is absorbed into the LUT and the carry logic calculates the "borrow". When synthesized correctly, an addition or subtraction operation performed on two 8-bit operands takes four slices (two CLBs). Table 2 shows that all three synthesis tools used four slices and inferred the carry logic.

Example:

Verilog: SUM = {1'b0, A} – {1'b0, B};

VHDL: SUM $\Leftarrow$ ('0' & A) – ('0' & B);

                 where SUM is 9 bits and A and B are 8 bits.

Since the addition of two 8-bit operands generates a 9-bit full SUM, the two operands A and B need to be declared as 9-bit wide (set most significant bit = 0) so that SUM $\Leftarrow$ A + B has the same bit width on both sides of the equation. In both instances, the carry out of the MSB will be used as the ninth bit of SUM.

Comparators

Comparators test the equality of two input arguments. The output for a comparator is asserted only if the two input numbers are exactly equal. The Verilog and VHDL code for a comparator is available at:
ftp://ftp.xilinx.com/pub/apps/xapp215.zip.

The logic is synthesized by operator inference, "= =" for verilog and "=" for VHDL.

Magnitude Comparators

Magnitude comparators compare the actual values of two arguments; A and B. The output is asserted if A is greater than or equal to B. To implement a magnitude comparator in schematic format, a subtract is performed with the final carry (borrow) output indicating a Logic "1" for A $\geq$ B is true and Logic "0" indicating if the condition is false. In the Verilog and VHDL examples in the reference design file, all three synthesis tools produced the most efficient logic using operator inference on the "$\geq$" operator. For the signed implementation, the VHDL signals are defined as "signed" signals rather than the "std_logic". Include the following use statement in the VHDL code `"use IEEE.std_logic_unsigned.all;"`

Two's Complement

The calculation of two's complement of a binary argument (n-bit vector) is used extensively in digital signal processing for signed arithmetic signals and is the most interesting in the context of HDL synthesis and logic inference. For an n-bit vector, the two's complement is obtained by subtraction of the n-bit vector from an n-bit vector of all zeroes (00… n times) and adding "1". The most efficient implementation for an 8-bit vector is expected to use 8 LUTs (4 slices) with carry-in set to "1". Only one synthesis tool produced the optimum implementation as shown in Table 2.

Multipliers Implemented in Virtex Devices

The multiplication of two (unsigned) binary multiplicands is essentially a series of shift and add operations. Figure 5 illustrates how an N x 2 Full Multiplier is implemented using the same logic resources as a simple adder. The key to implementing multipliers efficiently in a Virtex device is leveraging the extra AND gate (next to each LUT) and the carry-chain logic.

While an HDL based design flow is portable across device families, Xilinx offers device specific cores and reference designs for applications requiring high density and fast performance. An M x N pipelined multiplier reference design for Virtex devices is found at www.xilinx.com/ipcenter/cores_virtex.htm.
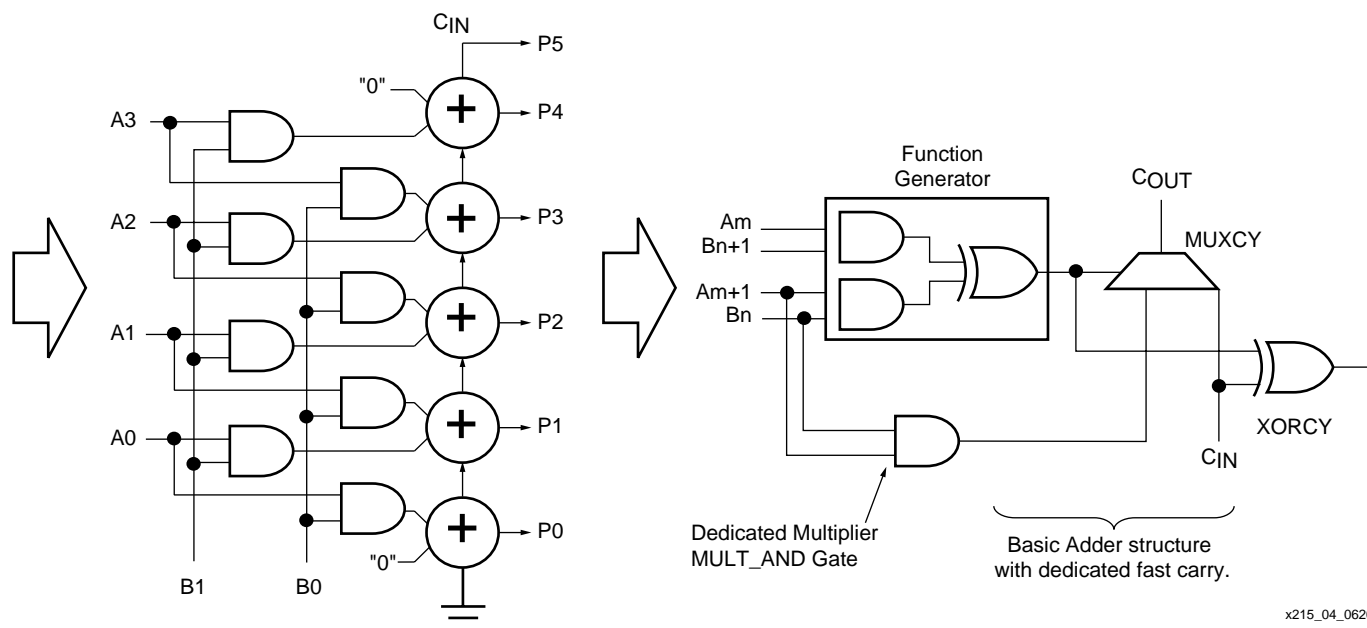
x215_04_062000

Figure 5: **N x 2 Full Multiplier Implementation**

Table 2: **Summary of Synthesis Results**

Arithmetic Function		FPGA Express # Slices (1 slice = 2 LUTs)	Synplify # Slices (1 slice = 2 LUTs)	Leonardo Spectrum # Slices (1 slice = 2 LUTs)	Comments on Synthesis Results
8-bit Add/Subtract		8 LUTs/ 4 slices	8 LUTs/ 4 slices	8 LUTs/ 4 slices	Synthesized design inferred carry logic. Optimum design obtained. Good logic packing is indicated by LUT to Slice ratio.
8-bit Comparator		5 LUTs/ 4 Slices No Carry Chain	4 LUTs/ 2 Slices with Carry Chain	4 LUTs/ 2 Slices with Carry Chain	Synplify and Leonardo utilized the carry chain logic. The carry out of the final stage is a success equality signal. FPGA Express implements the logic in LUTs.
8-bit Unsigned Magnitude Comparator		8 LUTs/ 4 slices	8 LUTs/ 4 slices	8 LUTs/ 4 slices	Optimum results. Slice to LUT ratio shows good packing.
Two's Complement	$\overline{A}$ = "0...0" − bit-vectorA + 1	9 LUTs/ 5 slices No Carry Chain	8 LUTs/ 12 slices with Carry Chain	8 LUTs/4 slices with Carry Chain	Leonardo spectrum yields the most efficient implementation.
	$\overline{A}$ = "1...1" XOR bit-vectorA + 1	8 LUTs/ 4 slices Optimal Implementation	8 LUTs/ 12 slices with Carry Chain. Inefficient Logic Packing	8 LUTs/11 slices with Carry Chain	FPGA Express yields the best implementation.
	N x M Multiplier N = 8 bit, M = 2 bit	9 LUTS/ 5 Slices	9 LUTS/ 5 Slices	9 LUTS/ 5 Slices	Optimum results, carry logic used in all three cases.

Conclusion

HDL coding style can greatly affect the way arithmetic logic is synthesized. Dealing with signed and unsigned numbers and operands with different bits are important considerations when coding arithmetic operations. The Virtex architecture is very efficient for implementing arithmetic operations using features of the CLB such as the carry logic and the dedicated AND gate. In DSP applications, fast and efficient implementation of elementary arithmetic function blocks becomes especially important. A good example is the Multiply and Accumulate (MAC) operation used repeatedly, in very large numbers, in DSP building blocks.

In addition to coding style, the role of the synthesis tool in leveraging the features of the Virtex architecture is significant. Synthesis tools are observed to infer fast and dense logic for the most part by operator inference. Designers using an HDL flow need to be aware of the capabilities of their synthesis tool and the target device architecture.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
6/28/00	1.0	Initial Xilinx release.