VHDL Math Tricks of the Trade

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

http://www.SynthWorks.com

VHDL Math Tricks of the Trade

VHDL is a strongly typed language. Success in VHDL depends on understanding the types and overloaded operators provided by the standard and numeric packages.

The paper gives a short tutorial on:

• VHDL Types & Packages
• Strong Typing Rules
• Converting between Std_logic_vector, unsigned & signed
• Ambiguous Expressions
• Arithmetic Coding Considerations
• Math Tricks
### Common VHDL Types

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Value</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>std_ulogic</td>
<td>'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'</td>
<td>std_logic_1164</td>
</tr>
<tr>
<td>std_ulogic_vector</td>
<td>array of std_ulogic</td>
<td>std_logic_1164</td>
</tr>
<tr>
<td>std_logic</td>
<td>resolved std_ulogic</td>
<td>std_logic_1164</td>
</tr>
<tr>
<td>std_logic_vector</td>
<td>array of std_logic</td>
<td>std_logic_1164</td>
</tr>
<tr>
<td>unsigned</td>
<td>array of std_logic</td>
<td>numeric_std, std_logic_arith</td>
</tr>
<tr>
<td>signed</td>
<td>array of std_logic</td>
<td>numeric_std, std_logic_arith</td>
</tr>
<tr>
<td>boolean</td>
<td>true, false</td>
<td>standard</td>
</tr>
<tr>
<td>character</td>
<td>191 / 256 characters</td>
<td>standard</td>
</tr>
<tr>
<td>string</td>
<td>array of character</td>
<td>standard</td>
</tr>
<tr>
<td>integer</td>
<td>-(2^31 - 1) to (2^31 - 1)</td>
<td>standard</td>
</tr>
<tr>
<td>real</td>
<td>-1.0E38 to 1.0E38</td>
<td>standard</td>
</tr>
<tr>
<td>time</td>
<td>1 fs to 1 hr</td>
<td>standard</td>
</tr>
</tbody>
</table>

### Packages for Numeric Operations

- **numeric_std** -- IEEE standard
  - Defines types signed, unsigned
  - Defines arithmetic, comparison, and logic operators for these types

- **std_logic_arith** -- Synopsys, a defacto industry standard
  - Defines types signed, unsigned
  - Defines arithmetic, and comparison operators for these types

- **std_logic_unsigned** -- Synopsys, a defacto industry standard
  - Defines arithmetic and comparison operators for std_logic_vector

**Recommendation:**
Use **numeric_std** for new designs
Ok to use **std_logic_unsigned with numeric_std**

* Currently, IEEE 1076.3 plans to have a numeric package that permits unsigned math with std_logic_vector
Packages for Numeric Operations

- Using IEEE Numeric Std

```vhdl
library ieee;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
```

Recommendation: Use numeric_std for new designs

- Using Synopsys Std_Logic_Arith

```vhdl
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

Recommendation, if you use Synopsys Packages:
Use std_logic_arith for numeric operations
Use std_logic_unsigned only for counters and testbenches
Don't use the package std_logic_signed.

Unsigned and Signed Types

- Used to represent numeric values:

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>0 to (2^N - 1)</td>
<td>2's Complement number</td>
</tr>
<tr>
<td>signed</td>
<td>(-2^{(N-1)}) to (2^{(N-1)} - 1)</td>
<td></td>
</tr>
</tbody>
</table>

- Usage similar to std_logic_vector:

```vhdl
signal A_unsigned    : unsigned(3 downto 0) ;
signal B_signed      : signed  (3 downto 0) ;
signal C_slv         : std_logic_vector (3 downto 0) ;

A_unsigned <= "1111" ;   => 15 decimal
B_signed   <= "1111" ;   => -1 decimal
C_slv      <= "1111" ;   => 15 decimal only if using std_logic_unsigned
```
Unsigned and Signed Types

- Type definitions identical to \texttt{std\_logic\_vector}

```
type UNSIGNED is array (natural range <>) of std\_logic;
type SIGNED is array (natural range <>) of std\_logic;
```

- How are the types distinguished from each other?
- How do these generate unsigned and signed arithmetic?
- For each operator, a unique function is called

```
function "+" (L, R: signed) return signed;
function "+" (L, R: unsigned) return unsigned;
```

- \textbf{This feature is called Operator Overloading:}
  - An operator symbol or subprogram name can be used more than once as long as calls are differentiable.

Overloading Basics

- Simplified view of overloading provided by VHDL packages

<table>
<thead>
<tr>
<th>Operator</th>
<th>Left</th>
<th>Right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>TypeA</td>
<td>TypeA</td>
<td>TypeA</td>
</tr>
<tr>
<td>Numeric</td>
<td>Array</td>
<td>Array</td>
<td>Array$^1$</td>
</tr>
<tr>
<td></td>
<td>Array</td>
<td>Integer</td>
<td>Array$^1$</td>
</tr>
<tr>
<td></td>
<td>Integer</td>
<td>Array</td>
<td>Array$^1$</td>
</tr>
</tbody>
</table>

\textbf{Notes:}

- Array = unsigned, signed, \texttt{std\_logic\_vector}$^2$
- TypeA = boolean, \texttt{std\_logic}, \texttt{std\_ulogic}, \texttt{bit\_vector}
  \texttt{std\_logic\_vector}, \texttt{std\_ulogic\_vector},
  \texttt{signed}$^3$, \texttt{unsigned}$^3$

Array and TypeA types used in an expression must be the same.

1) for comparison operators the result is boolean
2) only for \texttt{std\_logic\_unsigned}.
3) only for \texttt{numeric\_std} and not \texttt{std\_logic\_arith}

Overloading Examples

Signal A_uv, B_uv, C_uv, D_uv, E_uv : unsigned(7 downto 0) ;
Signal R_sv, S_sv, T_sv, U_sv, V_sv : signed(7 downto 0) ;
Signal J_slv, K_slv, L_slv : std_logic_vector(7 downto 0) ;
signal Y_sv        : signed(8 downto 0) ;
...

-- Permitted
A_uv <= B_uv + C_uv ;   -- Unsigned + Unsigned = Unsigned
D_uv <= B_uv + 1 ;    -- Unsigned + Integer  = Unsigned
E_uv <= 1 + C_uv;    -- Integer  + Unsigned = Unsigned

R_sv <= S_sv + T_sv ; -- Signed   + Signed   = Signed
U_sv <= S_sv + 1 ;   -- Signed   + Integer  = Signed
V_sv <= 1 + T_sv;   -- Integer  + Signed   = Signed
J_slv <= K_slv + L_slv ;   -- if using std_logic_unsigned

-- Illegal Cannot mix different array types
-- Solution presented later in type conversions
-- Y_sv <= A_uv - B_uv ;   -- want signed result

Strong Typing Implications

- **Size and type of target** (left) = size and type of expression (right)
- Each operation returns a result that has a specific size based on rules of the operation. The table below summarizes these rules.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Size of Y = Size of Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y &lt;= &quot;10101010&quot; ;</td>
<td>number of digits in literal</td>
</tr>
<tr>
<td>Y &lt;= X&quot;AA&quot; ;</td>
<td>4 * (number of digits)</td>
</tr>
<tr>
<td>Y &lt;= A ;</td>
<td>A'Length = Length of array A</td>
</tr>
<tr>
<td>Y &lt;= A and B ;</td>
<td>A'Length = B'Length</td>
</tr>
<tr>
<td>W &lt;= A &gt; B ;</td>
<td>Boolean</td>
</tr>
<tr>
<td>Y &lt;= A + B ;</td>
<td>Maximum (A'Length, B'Length)</td>
</tr>
<tr>
<td>Y &lt;= A + 10 ;</td>
<td>A'Length</td>
</tr>
<tr>
<td>V &lt;= A * B ;</td>
<td>A'Length + B'Length</td>
</tr>
</tbody>
</table>

Some think VHDL is difficult because of strong typing
Master the above simple rules and it is easy
**Strong Typing Implications**

```vhdl
signal A8, B8, Result8 : unsigned(7 downto 0) ;
signal Result9         : unsigned(8 downto 0) ;
signal Result7         : unsigned(6 downto 0) ;

-- Simple Addition, no carry out
Result8 <= A8 + B8 ;

-- Carry Out in result
Result9 <= ('0' & A8) + ('0' & B8) ;

-- For smaller result, slice input arrays
Result7 <= A8(6 downto 0) + B8(6 downto 0) ;
```

**Strong Typing = Strong Error Checking Built into the Compiler**

This means less debugging.
Without VHDL, you better have a good testbench and lots of time to catch your errors.

---

**Type Conversions**

- VHDL is dependent on overloaded operators and conversions
- What conversion functions are needed?
  - Signed & Unsigned (elements) <=> Std_Logic
  - Signed & Unsigned <=> Std_Logic_Vector
  - Signed & Unsigned <=> Integer
  - Std_Logic_vector <=> Integer
- VHDL Built-In Conversions
  - Automatic Type Conversion
  - Conversion by Type Casting
- Conversion functions located in Numeric_Std
Automatic Type Conversion: Unsigned, Signed $\leftrightarrow$ Std_logic

- Two types convert automatically when both are subtypes of the same type.

```
subtype std_logic is resolved std_ulogic;
```

- Converting between std_ulogic and std_logic is automatic.

- Elements of Signed, Unsigned, and std_logic_vector = std_logic
  - Elements of these types convert automatically to std_ulogic or std_logic

Legal Assignments

```
A_sl <= J_uv(0);
B_sul <= K_sv(7);
L_uv(0) <= C_sl;
M_slv(2) <= N_sv(2);
```

Implication:

```
Y_sl <= A_sl and B_sul and
       J_uv(2) and K_sv(7) and M_slv(2);
```

Type Casting: Unsigned, Signed $\leftrightarrow$ Std_logic_Vector

- Use type casting to convert equal sized arrays when:
  - Elements have a common base type (i.e. std_logic)
  - Indices have a common base type (i.e. Integer)

- Unsigned, Signed $\leftrightarrow$ Std_logic_Vector

```
A_slv <= std_logic_vector( B_uv );
C_slv <= std_logic_vector( D_sv );
G_uv <= unsigned( H_slv );
J_sv <= signed( K_slv );
```

Motivation, Unsigned - Unsigned = Signed?

```
signal X_uv, Y_uv : unsigned (6 downto 0);
signal Z_sv     : signed  (7 downto 0);
...
Z_sv <= signed('0' & X_uv) - signed('0' & Y_uv);
```
Numeric_Std Conversions: Unsigned, Signed <-> Integer

- Converting to and from integer requires a conversion function.
  - Unsigned, Signed => Integer
    
    ```
    Unsigned_int <= TO_INTEGER ( A_uv ) ;
    Signed_int <= TO_INTEGER ( B_sv ) ;
    ```
  - Integer => Unsigned, Signed
    
    ```
    C_uv <= TO_UNSIGNED ( Unsigned_int, 8 ) ;
    D_sv <= TO_SIGNED ( Signed_int, 8 ) ;
    ```

- Motivation (indexing an array of an array):
  
  ```
  Data_slv <= ROM( TO_INTEGER( Addr_uv ) ) ;
  ```

---

Std_Logic_Arith Conversions: Unsigned, Signed <-> Integer

- Converting to and from integer requires a conversion function.
  - Unsigned, Signed => Integer
    
    ```
    Unsigned_int <= Conv_INTEGER ( A_uv ) ;
    Signed_int <= Conv_INTEGER ( B_sv ) ;
    ```
  - Integer => Unsigned, Signed
    
    ```
    C_uv <= Conv_UNSIGNED ( Unsigned_int, 8 ) ;
    D_sv <= Conv_SIGNED ( Signed_int, 8 ) ;
    ```

- Motivation (indexing an array of an array):
  
  ```
  Data_slv <= ROM( Conv_INTEGER( Addr_uv ) ) ;
  ```
**Std.Logic_Vector <=> Integer**

- Converting between std_logic_vector and integer is a two step process:

  - **Numeric Std:** Std.Logic_Vector => Integer
    
    | Unsigned_int <= to_integer(unsigned(A_slv)); |
    | Signed_int <= to_integer(signed(B_slv)); |

  - **Numeric Std:** Integer => Std.Logic_Vector
    
    | C_slv <= std_logic_vector(to_unsigned(Unsigned_int, 8)); |
    | D_slv <= std_logic_vector(to_signed(Signed_int, 8)); |

- **Ambiguous Expressions**

  - An expression / statement is ambiguous if more than one operator symbol or subprogram can match its arguments.

  - **Std.Logic_Arith** defines the following two functions:
    
    ```
    function "+" (L, R: SIGNED) return SIGNED;
    function "+" (L: SIGNED; R: UNSIGNED) return SIGNED;
    ```

  - The following expression is ambiguous and an error:
    
    ```
    Z_sv <= A_sv + "1010" ;
    ```
    
    - Issues typically only arise when using literals.

    - How do we solve this problem?
**Std_Logic_Arith:**

**Ambiguous Expressions**

- VHDL type qualifier (**type_name'**) is a mechanism that specifies the type of an operand or return value of a subprogram (or operator).

  ```vhdl
  Z_sv    <=  A_sv  + signed'("1010") ;
  ```

- Leaving out the ' is an error:

  ```vhdl
  -- Z_sv <=  A_sv  + signed("1010") ;
  ```

- Without ', it is type casting. Use type casting for:

  ```vhdl
  Z Sv    <=  A_sv  + signed(B_slv) ;
  ```

- Recommended solution, use integer:

  ```vhdl
  Z_sv    <=  A_sv  - 6 ;
  ```

---

**Addition Operators**

**Addition Operators:**  +  -

- Arrays with Arrays:

  ```vhdl
  Add_uv  <= A_uv + B_uv ;
  Sub_uv  <= C_uv - D_uv ;
  ```

  - Size of result =
    - Size of largest array operand
    - Size of Add = maximum(A, B)
    - Shorter array gets extended.

- Arrays with Integers:

  ```vhdl
  Inc_uv  <= Base_uv + 1 ;
  Y_uv    <= A_uv + 45 ;
  ```

  - **Caution:**  Integers must fit into an array the same size as the result.
    - Extra MSB digits are lost
    - A must be at least 6 bits

  By convention the left most bit is the MSB
Use Integers with Care

- Synthesis tools create a 32-bit wide resources for unconstrained integers

```vhdl
signal Y_int, A_int, B_int : integer;
Y_int <= A_int + B_int;
```

- Do not use unconstrained integers for synthesis

```vhdl
signal A_int, B_int: integer range -8 to 7;
signal Y_int : integer range -16 to 15;
Y_int <= A_int + B_int;
```

- **Recommendation**: Use integers only as constants or literals

```vhdl
Y_uv <= A_uv + 17;
```

Comparison Operators

- Comparison operators return type boolean

```vhdl
AGeB <= '1' when (A_uv >= B_uv) else '0';
AEq15 <= '1' when (A_uv = "1111") else '0';
```

- Std_Logic is our basic type for design.
  - How do we convert from boolean to std_logic?

- Arrays with Arrays:

```vhdl
AGeB <= '1' when (A_uv >= B_uv) else '0';
AEq15 <= '1' when (A_uv = "1111") else '0';
```

- Arrays with Integers (special part of arithmetic packages):

```vhdl
DEq15 <= '1' when (D_uv = 15) else '0';
```

**Result = Boolean**  **Input arrays are extended to be the same length**
Multiplication and Division

Multiplication Operators: * / mod rem

- Array Multiplication

signal A_uv, B_uv : unsigned(7 downto 0);
signal Z_uv     : unsigned(15 downto 0);

Z_uv <= A_uv * B_uv;

- Array with Integer (only numeric_std)

Z_uv <= A_uv * 2;

Note: */ mod rem* not well supported by synthesis tools.

Adder with Carry Out

Unsigned Algorithm:

'0',     A(3:0)
+ '0',     B(3:0)
-------------------
CarryOut, Result(3:0)

Unsigned Code:

Y5 <= ('0' & A) + ('0' & B);
Y <= Y5(3 downto 0);
Co <= Y5(4);

signal A, B, Y : unsigned(3 downto 0);
signal Y5      : unsigned(4 downto 0);
signal Co      : std_logic;
Adder with Carry In

Desired Result:

\[ A(3:0) + B(3:0) + \text{CarryIn} \]

Algorithm

\[ A(3:0), '1' + B(3:0), \text{CarryIn} \]

Result (4:1), Unused

Example: Carry = 0

\[
\begin{array}{c}
0010, 1 \\
0001, 0 \\
0011, 1 \\
\hline
\text{Result}
\end{array}
\]

Carry = 1

\[
\begin{array}{c}
0010, 1 \\
0001, 1 \\
0100, 0 \\
\hline
\text{Result}
\end{array}
\]

Code:

\[
y5 \leftarrow (A \& '1') + (B \& \text{CarryIn}); \\
y \leftarrow y5(4 \text{ downto } 1);
\]

ALU Functions

- ALU1:

<table>
<thead>
<tr>
<th>OpSel</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>A + B</td>
</tr>
<tr>
<td>01</td>
<td>C + D</td>
</tr>
<tr>
<td>10</td>
<td>E + F</td>
</tr>
<tr>
<td>11</td>
<td>G + H</td>
</tr>
</tbody>
</table>

- Three implementations
  - Tool Driven Resource Sharing
  - Code Driven Resource Sharing
  - Defeating Resource Sharing

- Since OpSel can select only one addition at a time, the operators are mutually exclusive.
Possible Solutions to ALU 1

As Specified:

Optimal results:

- This transformation of operators is called Resource Sharing

ALU 1: Tool Driven

ToolDrvnProc : process (OpSel,A,B,C,D,E,F,G,H)
begin
  case OpSel is
    when "00" =>  Z <= A + B ;
    when "01" => Z <= C + D ;
    when "10" =>  Z <= E + F ;
    when "11" =>  Z <= G + H ;
    when others => Z <= (others => 'X') ;
  end case ;
end process ;  -- ToolDrvnProc

- Important: to ensure resource sharing, operators must be coded in the same process, and same code (case or if) structure.

- Any potential issues with this?
ALU 1: Code Driven

\[
X \leftarrow \text{Mux4}(\text{OpSel}, A, C, E, G) ; \\
Y \leftarrow \text{Mux4}(\text{OpSel}, B, D, F, H) ; \\
Z \leftarrow X + Y ;
\]

- Best Synthesis, use for:
  - Sharing arithmetic operators
  - Sharing comparison operators
  - Sharing complex function calls
    - Resource sharing often is not possible when using third party arithmetic logic.

ALU 1: Defeating Resource Sharing *

- Bad Code will defeat Resource Sharing.

```plaintext
BadAluProc:  process (OpSel, A, B, C, D, E, F, G, H)
begin
if (OpSel = "00") then   Z <= A + B;   end if;
if (OpSel = "01") then   Z <= C + D;   end if;
if (OpSel = "10") then   Z <= E + F;   end if;
if (OpSel = "11") then   Z <= G + H;   end if;
end process ;
```

- *Not Recommended*,
synthesis tool may create a separate resource for each adder.
Defeating Resource Sharing

- When does this happen?

```vhdl
case StateReg is
  when S1 =>
    if (in1 = '1') then
      Z <= A + B;
      . . .
    end if;
  when S2 =>
    if (in2 = '1') then
      Z <= C + D;
      . . .
    end if;
  . . .
  when Sn =>
  . . .
  when others =>
```

- Separate statemachines and resources

```vhdl
Statemach : process(...)
begin
  -- generate function
  -- select logic (OpSel)
end process;

Resources : process(...)
begin
  -- code:
  -- arithmetic operators
  -- comparison operators
end process;
```

More Information

There is work in progress to extend VHDL's math capability. For more information see the following IEEE working groups websites:

<table>
<thead>
<tr>
<th>Group</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE 1164</td>
<td><a href="http://www.eda.org/vhdl-std-logic">http://www.eda.org/vhdl-std-logic</a></td>
</tr>
<tr>
<td>IEEE 1076.3/numeric std</td>
<td><a href="http://www.eda.org/vhdlsynth">http://www.eda.org/vhdlsynth</a></td>
</tr>
<tr>
<td>IEEE 1076.3/floating point</td>
<td><a href="http://www.eda.org/fphdl">http://www.eda.org/fphdl</a></td>
</tr>
</tbody>
</table>

Also see the DVCon 2003 paper, "Enhancements to VHDL's Packages" which is available at:

http://www.synthworks.com/papers
Author Biography

Jim Lewis, Director of Training, SynthWorks Design Inc.

Jim Lewis, the founder of SynthWorks, has seventeen years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr. Lewis does ASIC and FPGA design, custom model development, and consulting. Mr. Lewis is an active member of IEEE Standards groups including, VHDL (IEEE 1076), RTL Synthesis (IEEE 1076.6), Std_Logic (IEEE 1164), and Numeric_Std (IEEE 1076.3). Mr. Lewis can be reached at jim@SynthWorks.com, (503) 590-4787, or http://www.SynthWorks.com

SynthWorks VHDL Training

Comprehensive VHDL Introduction  4 Days
http://www.synthworks.com/comprehensive_vhdl_introduction.htm
A design and verification engineers introduction to VHDL syntax, RTL coding, and 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