# 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

SynthWorks

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

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

# 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

```
library ieee ;
  use ieee.std_logic_1164.all ;
  use ieee.numeric_std.all ;
```

Recommendation:
Use numeric_std for new designs

Use **numeric_std** or **std_logic_arith**, but never both

- Using Synopsys Std_Logic_Arith

```
library ieee ;
  use ieee.std_logic_1164.all ;
  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:

| TYPE | Value | Notes |
|---|---|---|
| unsigned | 0 to $2^{N}$ - 1 | |
| signed | - $2^{(N-1)}$ to $2^{(N-1)}$ - 1 | 2's Complement number |

- Usage similar to std_logic_vector:

```
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" ;
B_signed   <= "1111" ;
C_slv      <= "1111" ;
```

**= 15 decimal**

**= -1 decimal**

**= 15 decimal only if using std_logic_unsigned**

# Unsigned and Signed Types

SynthWorks

- Type definitions identical to 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 ;
```

- 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

SynthWorks

- Simplified view of overloading provided by VHDL packages

| Operator | Left | Right | Result |
|---|---|---|---|
| Logic | TypeA | TypeA | TypeA |
| Numeric | Array | Array | Array[1] |
| | Array | Integer | Array[1] |
| | Integer | Array | Array[1] |

Notes:

Array = unsigned, signed, std_logic_vector[2]

TypeA = boolean, std_logic, std_ulogic, bit_vector<br>
std_logic_vector, std_ulogic_vector,<br>
signed[3], 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 std_logic_unsigned.
3) only for numeric_std and not std_logic_arith

- For a detailed view of VHDL's overloading, get the VHDL Types and Operators Quick Reference card at: http://www.SynthWorks.com/papers

# 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 persented 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.

| Operation | Size of Y = Size of Expression |
|---|---|
| `Y <= "10101010" ;` | number of digits in literal |
| `Y <= X"AA" ;` | 4 * (number of digits) |
| `Y <= A ;` | A'Length = Length of array A |
| `Y <= A and B ;` | A'Length = B'Length |
| `W <= A > B ;` | Boolean |
| `Y <= A + B ;` | Maximum (A'Length, B'Length) |
| `Y <= A + 10 ;` | A'Length |
| `V <= A * B ;` | A'Length + B'Length |

Some think VHDL is difficult because of strong typing

Master the above simple rules and it is easy

# Strong Typing Implications

```
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 <=> 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 <=> 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 <=> 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 ) ;
```

Array width = 8

  - Motivation (indexing an array of an array):

```
Data_slv <=  ROM(   TO_INTEGER( Addr_uv) ) ;
```

```
signal A_uv, C_uv    : unsigned (7 downto 0) ;
signal Unsigned_int : integer range 0 to 255 ;
signal B_sv, D_sv    : signed( 7 downto 0) ;
signal Signed_int    : integer range -128 to 127;
```

# 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 ) ;
```

Array width = 8

  - Motivation (indexing an array of an array):

```
Data_slv <=  ROM( Conv_INTEGER( Addr_uv) ) ;
```

```
signal A_uv, C_uv    : unsigned (7 downto 0) ;
signal Unsigned_int : integer range 0 to 255 ;
signal B_sv, D_sv    : signed( 7 downto 0) ;
signal Signed_int    : integer range -128 to 127;
```

# 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 ));
```

```
signal A_slv, C_slv   : std_logic_vector (7 downto 0)  ;
signal Unsigned_int   : integer range 0 to 255 ;
signal B_slv, D_slv   : std_logic_vector( 7 downto 0)  ;
signal Signed_int     : integer range -128 to 127;
```

# 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" ;
```

Is "1010" Signed or Unsigned

`"1010" =   -6 or 10`

  - Issues typically only arise when using literals.

- How do we solve this problem?

SynthWorks

# 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).

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

Effects all numeric operators in std_logic_arith

- Leaving out the ' is an error:

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

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

```
Z_sv    <=  A_sv  + signed(B_slv) ;
```

- Recommended solution, use integer:

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

SynthWorks

# Addition Operators

Addition Operators:    +    -

- Arrays with Arrays:

```
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:

```
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

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

  - Do not use unconstrained integers for synthesis

- Specify a range with integers:

```
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

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

# Comparison Operators

Comparison Operators: = /= > >= < <=

  - Comparison operators return type boolean

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

- Arrays with Arrays:

```
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):

```
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;
```

- Size of result =
  - Sum of the two input arrays

- Array with Integer (only numeric_std)

```
Z_uv <= A_uv * 2 ;
```

- Size of result =
  - 2 * size of array input

**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

```
signal A, B, Y : unsigned(3 downto 0);
signal Y5      : unsigned(4 downto 0) ;
signal CarryIn : std_logic ;
```

## Desired Result:

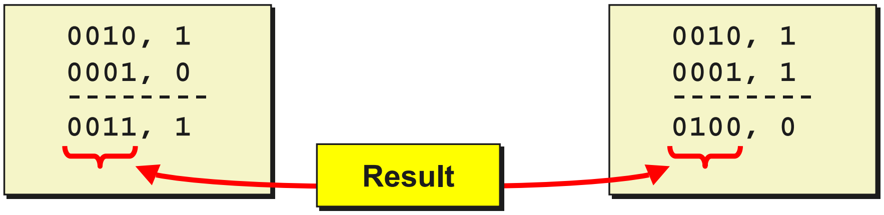
```
A(3:0) + B(3:0) + CarryIn
```

## Algorithm

```
  A(3:0), '1'
+ B(3:0), CarryIn
--------------------
  Result(4:1), Unused
```

## Example: Carry = 0

## Carry = 1



```
0010, 1
0001, 0
---------
0011, 1
```

Result

```
0010, 1
0001, 1
---------
0100, 0
```

## Code:

```
Y5  <= (A & '1') + (B & CarryIn);
Y   <= Y5(4 downto 1) ;
```

# ALU Functions

- ALU1:

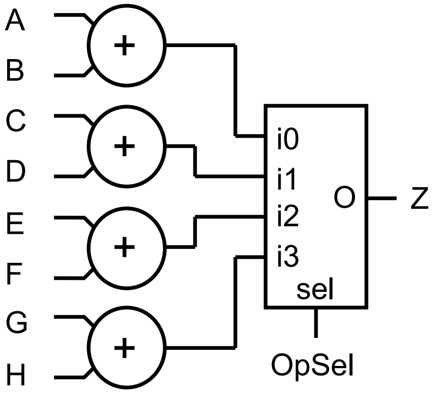
| OpSel | Function |
|---|---|
| 00 | A + B |
| 01 | C + D |
| 10 | E + F |
| 11 | G + H |

- Since OpSel can select only one addition at a time, the operators are mutually exclusive.

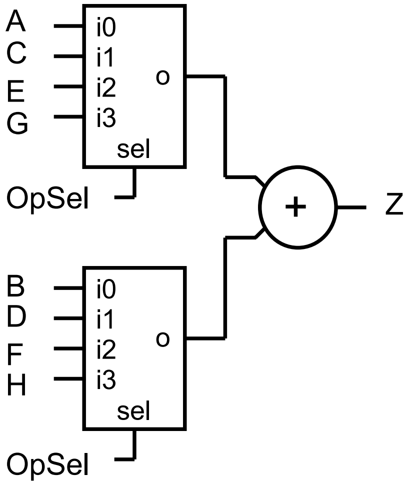
- Three implementations
  - Tool Driven Resource Sharing
  - Code Driven Resource Sharing
  - Defeating Resource Sharing

# 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 <= Mux4(OpSel, A, C, E, G) ;
Y <= Mux4(OpSel, B, D, F, H) ;

Z <= 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.

```
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 ;
```

Uses "end if", rather than "elsif"

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

# Defeating Resource Sharing

- When does this happen?

```
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

```
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:

| Group | Website |
|---|---|
| IEEE 1164 | http://www.eda.org/vhdl-std-logic |
| IEEE 1076.3/numeric std | http://www.eda.org/vhdlsynth |
| IEEE 1076.3/floating point | http://www.eda.org/fphdl |

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