ECE 545—Digital System Design with VHDL
Lecture 4

Algorithmic State Machines and Dataflow VHDL Coding

9/16/08

Outline

- Algorithmic State Machines
  - Sorting Example
- Dataflow VHDL (for combinational logic)
- Arithmetic using std_logic_arith

Resources

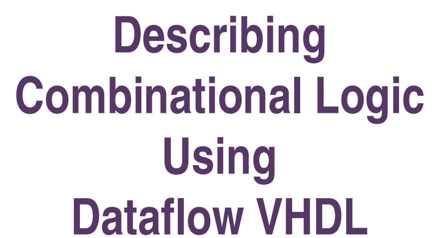
- Volnei A. Pedroni, *Circuit Design with VHDL*
  - ***Chapter 5, Concurrent Code***
  - ***Chapter 4.1, Operators***
- Stephen Brown and Zvonko Vranesic, *Fundamentals of Digital Logic with VHDL Design, 2nd or 3rd Edition*
  - ***Chapter 8.10 Algorithmic State Machine (ASM) Charts***
  - ***Chapter 10.2.1 A Bit-Counting Circuit***
  - ***Chapter 10.2.2 ASM Chart Implied Timing Information***
  - ***Chapter 10.2.6 Sort Operation***

    (handouts distributed in class)

Algorithmic State Machines

Algorithmic State Machines

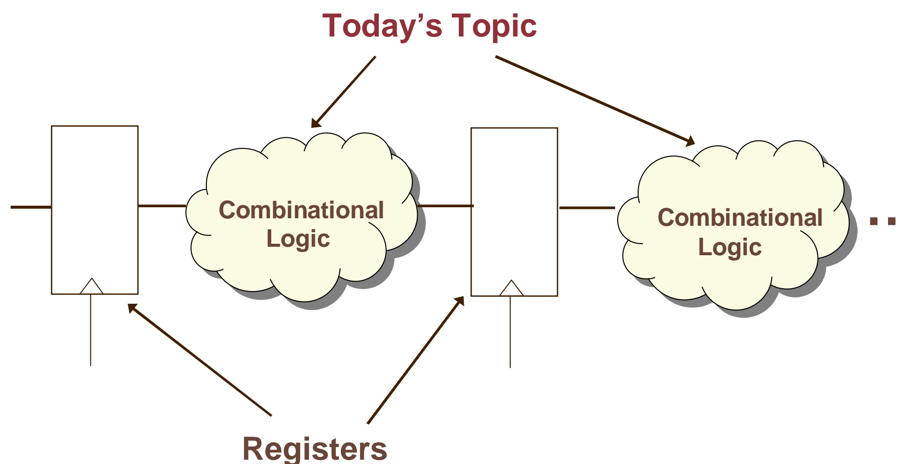
- We will continue to work through the Sorting Example from last week's lecture



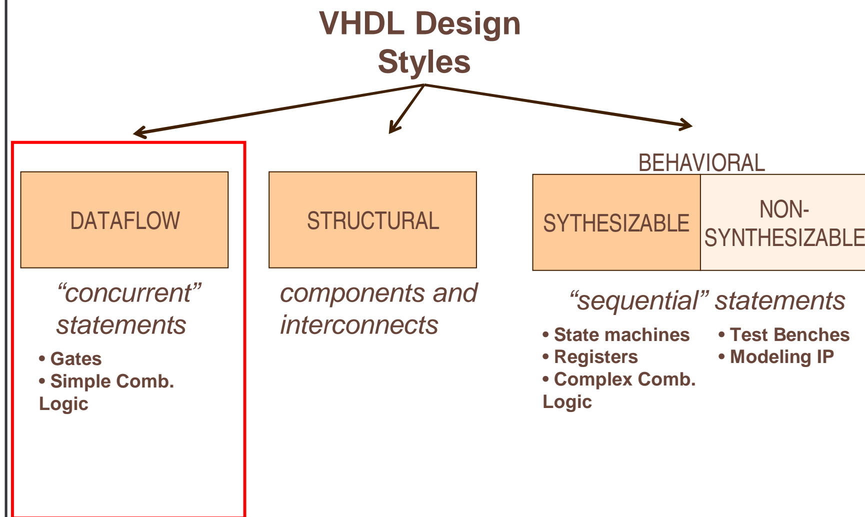
Brief Class Review

- In this class we have covered the following items **without using VHDL**:
  - Gates and combinational logic blocks (Lecture 2)
  - Sequential logic blocks (Lecture 3)
  - State machines (Lecture 3)
  - Algorithmic state machines (Lectures 3 & 4)
- Now we will learn how to code these blocks in VHDL

Register Transfer Level (RTL) Design Description



VHDL Design Styles (Architecture)



Dataflow VHDL

Major instructions

Concurrent statements

- concurrent signal assignment (⇐)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- generate scheme for equations (for-generate)

Dataflow VHDL

Major instructions

Concurrent statements

- **concurrent signal assignment (⇐)**
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- generate scheme for equations (for-generate)

Dataflow VHDL: Full Adder

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table

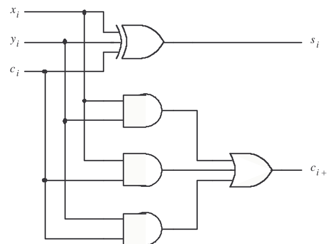
c_i \ $x_i y_i$	00	01	11	10
0		1		1
1	1		1	

$s_i = x_i \oplus y_i \oplus c_i$

c_i \ $x_i y_i$	00	01	11	10
0			1	
1		1	1	1

$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$

(b) Karnaugh maps



(c) Circuit

Full Adder Example

TIP: for architecture name for dataflow circuits use either "dataflow" or "entityname_dataflow"

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
       PORT ( x     : IN          STD_LOGIC ;
              y     : IN          STD_LOGIC ;
              cin   : IN          STD_LOGIC ;
              s     : OUT         STD_LOGIC ;
              cout  : OUT         STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE fulladd_dataflow OF fulladd IS
BEGIN
       s <= x XOR y XOR cin ;
       cout <= (x AND y) OR (cin AND x) OR (cin AND y) ;
END fulladd_dataflow ;
```


Logic Operators

- Logic operators

and	or	nand	nor	xor	not	xnor

only in VHDL-93 and later

- Logic operators precedence

Highest → Lowest

not					
and	or	nand	nor	xor	xnor

No Implied Precedence

Wanted: y **= ab + cd**

Incorrect

y <= a **and** b **or** c **and** d ;

equivalent to

y <= ((a **and** b) **or** c) **and** d ;

equivalent to

y = (ab + c)d

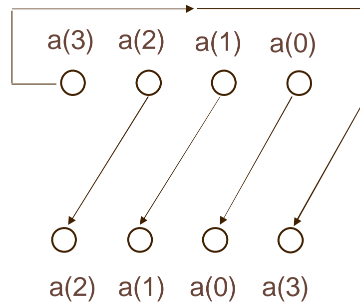
Correct

y <= (a **and** b) **or** (c **and** d) ;

Concatenation

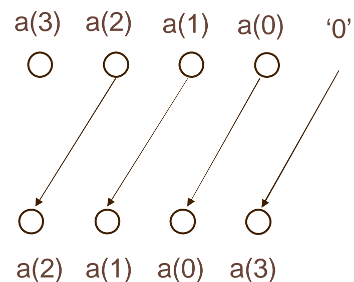
```
SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL c, d, e: STD_LOGIC_VECTOR(7 DOWNTO 0);

a <= "0000";
b <= "1111";
c <= a & b;              -- c = "00001111"

d <= '0' & "0001111";    -- d <= "00001111"

e <= '0' & '0' & '0' & '0' & '1' & '1' &
     '1' & '1';          -- e <= "00001111"
```

Rotations in VHDL



a_rotL <= a(2 downto 0) & a(3);

Shifts in VHDL (zero-stuff)



a_shiftL <= a(2 downto 0) & '0';

Be careful if doing sign-extension

Shifts in VHDL (using libraries)

- Using **std_logic_arith** package:
  - function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
  - function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
  - function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
  - function SHR(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
- Make sure your syntax is correct
- Recommend not using these functions, but "hard-wiring" the shifts as in previous examples

Dataflow VHDL

Major instructions

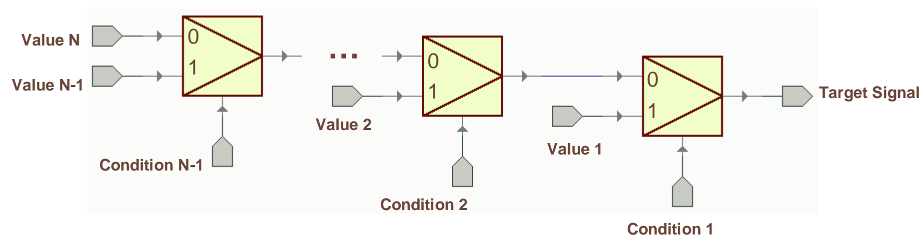
Concurrent statements

- concurrent signal assignment (⇐)
- **conditional concurrent signal assignment (when-else)**
- selected concurrent signal assignment (with-select-when)
- generate scheme for equations (for-generate)

Conditional concurrent signal assignment

When - Else

```
target_signal <= value1 when condition1 else
                 value2 when condition2 else
                   . . .
                 valueN-1 when conditionN-1 else
                 valueN;
```



Operators

- Relational operators

=	/=	<	<=	>	>=

- Logic and relational operators precedence

Highest → Lowest

		not			
=	/=	<	<=	>	>=
and	or	nand	nor	xor	xnor

Priority of Logic and Relational Operators

compare a = bc

Incorrect

… when a = b **and** c else …

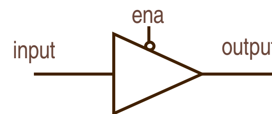
equivalent to

… when (a = b) **and** c else …

Correct

… when a = (b **and** c) else …

Tri-state Buffer – example




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tri_state IS
   PORT ( ena:     IN STD_LOGIC;
          input:   IN STD_LOGIC_VECTOR(7 downto 0);
          output: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
               );
END tri_state;
ARCHITECTURE tri_state_dataflow OF tri_state IS
BEGIN
     output <= input WHEN (ena = '0') ELSE
                     (OTHERS => 'Z');
END tri_state_dataflow;
```

OTHERS means all bits not directly specified, in this case all the bits.

Dataflow VHDL

Major instructions

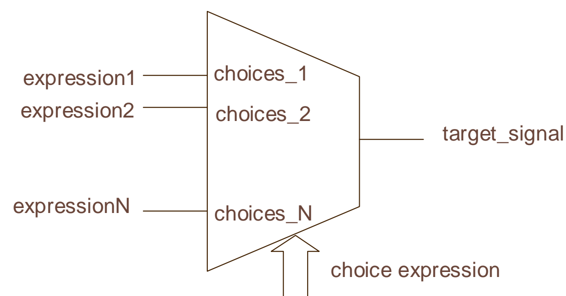
Concurrent statements

- concurrent signal assignment (⇐)
- conditional concurrent signal assignment (when-else)
- **selected concurrent signal assignment (with-select-when)**
- generate scheme for equations (for-generate)

Selected concurrent signal assignment

With –Select-When

```
with choice_expression select
   target_signal <= expression1 when choices_1,
                    expression2 when choices_2,
                         . . .
                    expressionN when choices_N;
```



Allowed formats of *choices_k*

```
WHEN value

WHEN value_1 to value_2

WHEN value_1 | value_2 | .... | value N
```

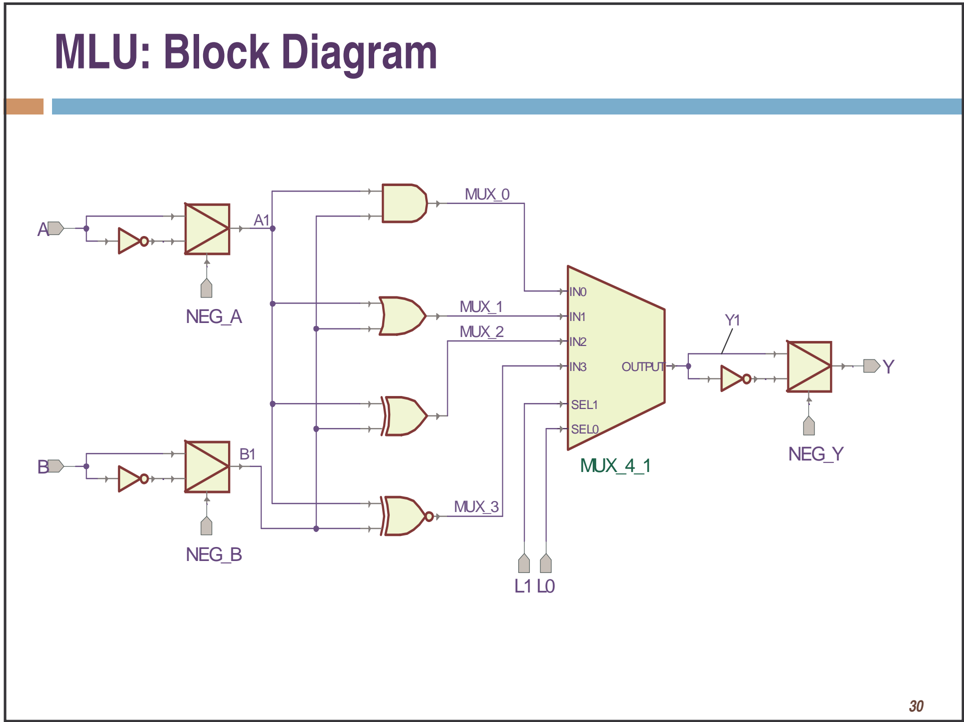
this means boolean “or”

Allowed formats of *choice_k - example*

```
WITH sel SELECT
    y <= a WHEN "000",
         b WHEN "011" to "110",
         c WHEN "001" | "111",
         d WHEN OTHERS;
```

MLU Example

MLU: Block Diagram



MLU: Entity Declaration

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mlu IS
    PORT(
         NEG_A : IN STD_LOGIC;
         NEG_B : IN STD_LOGIC;
         NEG_Y : IN STD_LOGIC;
         A :          IN STD_LOGIC;
         B :          IN STD_LOGIC;
         L1 :        IN STD_LOGIC;
         L0 :        IN STD_LOGIC;
         Y :         OUT STD_LOGIC
        );
END mlu;
```


MLU: Architecture Declarative Section

```
ARCHITECTURE mlu_dataflow OF mlu IS

SIGNAL  A1 :  STD_LOGIC;
SIGNAL  B1 :  STD_LOGIC;
SIGNAL  Y1 : STD_LOGIC;
SIGNAL  MUX_0 : STD_LOGIC;
SIGNAL  MUX_1 : STD_LOGIC;
SIGNAL  MUX_2 : STD_LOGIC;
SIGNAL  MUX_3 : STD_LOGIC;
SIGNAL  L: STD_LOGIC_VECTOR(1 DOWNTO 0);
```

MLU - Architecture Body

```
BEGIN
   A1<= NOT A  WHEN (NEG_A='1') ELSE
        A;
   B1<= NOT B  WHEN (NEG_B='1') ELSE
        B;
   Y <= NOT Y1 WHEN (NEG_Y='1') ELSE
        Y1;

   MUX_0 <= A1  AND  B1;
   MUX_1 <= A1   OR   B1;
   MUX_2 <= A1  XOR  B1;
   MUX_3 <= A1  XNOR B1;

      L <= L1 & L0;

   with (L) select
       Y1 <=  MUX_0  WHEN "00",
              MUX_1  WHEN "01",
              MUX_2  WHEN  "10",
              MUX_3  WHEN OTHERS;

END mlu_dataflow;
```


Data-flow VHDL

Major instructions

Concurrent statements

- concurrent signal assignment (⇐)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- **generate scheme for equations (for-generate)**

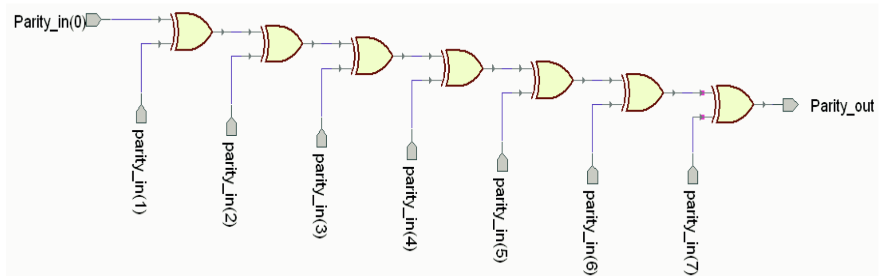
For Generate Statement

For - Generate

```
label: FOR identifier IN range GENERATE
          BEGIN
           {Concurrent Statements}
          END GENERATE [label];
```


PARITY Example

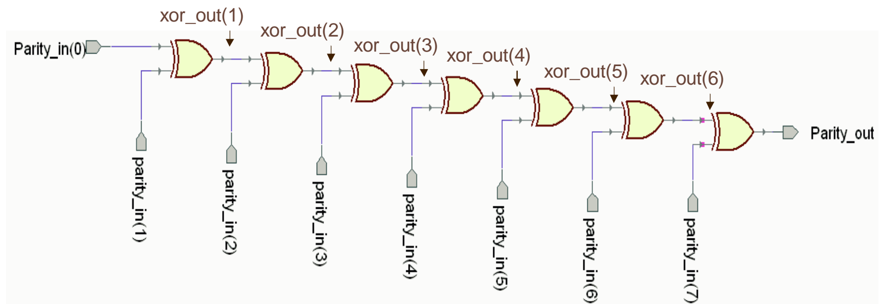
PARITY: Block Diagram



PARITY: Entity Declaration

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY parity IS
    PORT(
        parity_in    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        parity_out   : OUT STD_LOGIC
        );
END parity;
```

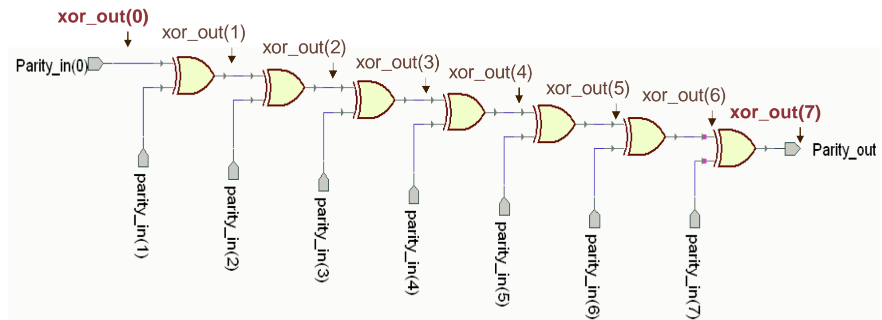
PARITY: Block Diagram



PARITY: Architecture

```
ARCHITECTURE parity_dataflow OF parity IS

SIGNAL xor_out: std_logic_vector (6 downto 1);

BEGIN

   xor_out(1) <= parity_in(0) XOR parity_in(1);
   xor_out(2) <= xor_out(1) XOR parity_in(2);
   xor_out(3) <= xor_out(2) XOR parity_in(3);
   xor_out(4) <= xor_out(3) XOR parity_in(4);
   xor_out(5) <= xor_out(4) XOR parity_in(5);
   xor_out(6) <= xor_out(5) XOR parity_in(6);
   parity_out <= xor_out(6) XOR parity_in(7);

END parity_dataflow;
```

PARITY: Block Diagram (2)



PARITY: Architecture

```
ARCHITECTURE parity_dataflow OF parity IS

SIGNAL xor_out: STD_LOGIC_VECTOR (7 downto 0);

BEGIN

    xor_out(0) <= parity_in(0);
    xor_out(1) <= xor_out(0) XOR parity_in(1);
    xor_out(2) <= xor_out(1) XOR parity_in(2);
    xor_out(3) <= xor_out(2) XOR parity_in(3);
    xor_out(4) <= xor_out(3) XOR parity_in(4);
    xor_out(5) <= xor_out(4) XOR parity_in(5);
    xor_out(6) <= xor_out(5) XOR parity_in(6);
    xor_out(7) <= xor_out(6) XOR parity_in(7);
    parity_out <= xor_out(7);

END parity_dataflow;
```

PARITY: Architecture (2)

```
ARCHITECTURE parity_dataflow OF parity IS

SIGNAL xor_out: STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN

  xor_out(0) <= parity_in(0);

  G2: FOR i IN 1 TO 7 GENERATE
      xor_out(i) <= xor_out(i-1) XOR parity_in(i);
  END GENERATE;

  parity_out <= xor_out(7);

END parity_dataflow;
```


Combinational Logic Synthesis for Beginners

Simple Rules

For combinational logic,
use only concurrent statements

- concurrent signal assignment (⇐)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- generate scheme for equations (for-generate)

Simple Rules

- For circuits composed of:
  - simple logic operations (logic gates)
  - simple arithmetic operations (addition, subtraction, multiplication)
  - shifts/rotations by a constant
- Use
  - concurrent signal assignment (⇐)

Simple Rules

- For circuits composed of
  - multiplexers
  - decoders, encoders
  - tri-state buffers
- Use:
  - conditional concurrent signal assignment (when-else)
  - selected concurrent signal assignment (with-select-when)

Left versus Right Side

Left side <= **Right side**

<= when-else

with-select <=

- Internal signals (defined in a given architecture)
- Ports of the mode
  - **- out**
  - - inout
  - - buffer (don't recommend using buffer in this class)

Expressions including:

- Internal signals (defined in a given architecture)
- Ports of the mode
  - **- in**
  - - inout
  - - buffer

Signed and Unsigned Arithmetic

Arithmetic operations

Synthesizable arithmetic operations:

- Addition: +
- Subtraction: -
- Comparisons: >, >=, <, <=
- Multiplication: *
- Division by a power of 2: /2**6
  (equivalent to right shift)
- Shifts by a constant: SHL, SHR

Arithmetic operations

The result of synthesis of an arithmetic operation is a
- combinational circuit
- without pipelining

The exact internal **architecture** used (and thus delay and area of the circuit) **may depend** on the **timing constraints** specified during synthesis (e.g., the requested maximum clock frequency).

IEEE Packages for Arithmetic

- std_logic_1164
  - Official IEEE standard
  - Defines **std_logic** and **std_logic_vector**
    - example: std_logic_vector(7 downto 0)
  - These are for logic operations (AND, OR) not for arithmetic operations (+, *)
- std_logic_arith
  - Not official IEEE standard (created by Synopsys)
  - Defines **unsigned** and **signed** data types, example: unsigned(7 downto 0)
  - These are for arithmetic (+,*) not for logic (AND, OR)
- std_logic_unsigned
  - Not official IEEE standard (created by Synopsys)
  - Including this library tells compiler to treat **std_logic_vector** like **unsigned** type in certain cases
    - For example, can perform addition on std_logic_vector
  - Used as a helper to avoid explicit conversion functions
- std_logic_signed
  - Not official IEEE standard (created by Synopsys)
  - Same functionality as std_logic_unsigned except tells compiler to treat **std_logic_vector** like **signed** type in certain cases
  - **Do not use both std_logic_unsigned and std_logic_signed at the same time!**
    - If need to do both signed and unsigned arithmetic in same entity, do not use std_logic_unsigned or std_logic_signed packages → do all conversions explicitly

Library inclusion

- When dealing with unsigned arithmetic use:

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;  ← needed for using unsigned data type
USE ieee.std_logic_unsigned.all;
```

- When dealing with signed arithmetic use:

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;  ← need for using signed data type
USE ieee.std_logic_signed.all;
```

- When dealing with both unsigned and signed arithmetic use:

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

→ Then do all type conversions explicitly

History: std_logic_arith versus numeric_std

- History
  - Package std_logic_arith created by Synopsys as a stop-gap measure
  - Eventually, the IEEE created their own official "version" of std_logic_arith called numeric_std
- numeric_std
  - Official IEEE standard
  - Defines **unsigned** and **signed**
  - These are for arithmetic (+,*) not for logic (AND, OR)
    - example: unsigned(7 downto 0)
  - **Use either numeric_std or std_logic_arith, not both!**
- When dealing with unsigned and/or signed types using numeric_std, use:

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

- Since different CAD vendors may implement std_logic_arith differently, you can use numeric_std to be safe
  - However … many legacy designs and companies use std_logic_arith in combination with std_logic_unsigned or std_logic_signed
  - **Examples in class and in exams will use std_logic_arith with std_logic_unsigned/signed, and not use numeric_std**
  - If you use numeric_std, make sure to declare it properly so all code compiles

Example: std_logic_arith versus numeric_std

UNSIGNED ADDER WITH NO CARRYOUT

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all; -- not needed but keep for style
use IEEE.std_logic_unsigned.all;
entity adder is
  port(
    a : in STD_LOGIC_VECTOR(2 downto 0);
    b : in STD_LOGIC_VECTOR(2 downto 0);
   c : out STD_LOGIC_VECTOR(2 downto 0)   );
end adder;

architecture adder_arch of adder is
begin
  c <= a + b;
end adder_arch;
```

Tells compiler to treat std_logic_vector like unsigned type

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;

entity adder is
  port(
    a : in STD_LOGIC_VECTOR(2 downto 0);
    b : in STD_LOGIC_VECTOR(2 downto 0);
   c : out STD_LOGIC_VECTOR(2 downto 0)   );
end adder;

architecture adder_arch of adder is
begin
  c <= std_logic_vector(unsigned(a) + unsigned(b));
end adder_arch;
```


Conversion functions

		numeric_std	std_logic_arith
Type Conversion			
std_logic_vector	-> unsigned	unsigned(*arg*)	unsigned(*arg*)
std_logic_vector	-> signed	signed(*arg*)	signed(*arg*)
unsigned	-> std_logic_vector	std_logic_vector(*arg*)	std_logic_vector(*arg*)
signed	-> std_logic_vector	std_logic_vector(*arg*)	std_logic_vector(*arg*)
integer	-> unsigned	to_unsigned(*arg*, *size*)	conv_unsigned(*arg*, *size*)
integer	-> signed	to_signed(*arg*, *size*)	conv_signed(*arg*, *size*)
unsigned	-> integer	to_integer(*arg*)	conv_integer(*arg*)
signed	-> integer	to_integer(*arg*)	conv_integer(*arg*)
integer	-> std_logic_vector	integer -> unsigned/signed ->std_logic_vector	
std_logic_vector	-> integer	std_logic_vector -> unsigned/signed ->integer	
unsigned + unsigned	-> std_logic_vector	std_logic_vector(*arg1* + *arg2*)	*arg1* + *arg2*
signed + signed	-> std_logic_vector	std_logic_vector(*arg1* + *arg2*)	*arg1* + *arg2*
Resizing			
unsigned		resize(*arg*, *size*)	conv_unsigned(*arg*, *size*)
signed		resize(*arg*, *size*)	conv_signed(*arg*, *size*)

From: http://dz.ee.ethz.ch/support/ic/vhdl/vhdlsources.en.html

More information?

- Go to:
  - http://dz.ee.ethz.ch/support/ic/vhdl/vhdlsources.en.html
- Download
  - std_logic_arith.vhd
  - std_logic_unsigned.vhd
  - std_logic_signed.vhd
  - numeric_std
  - …and compare them

Unsigned and Signed Numbers

- N-bit unsigned number: unsigned(N-1 downto 0)
  - Has a decimal range of 0 to 2^{N}-1
    - Example: unsigned(7 downto 0) has a decimal range 0 to 255
- N-bit signed number (two's complement) number: signed(N-1 downto 0)
  - Has a decimal range of -2^{N-1} to 2^{N-1}-1
    - Asymmetric range due to non-redundant representation of 0
    - Example: signed(7 downto 0) has a decimal range -128 to 127
  - MSB indicates sign
    - '0' in MSB means non-negative (0 or positive)
    - '1' in MSB means negative
  - To negate a two's complement number: invert all bits, add '1' to LSB
    - 010 = 2
    - to get -2, invert then add '1': 101 +1 = 110
  - Sign extension does not affect value
    - Example: 010 and 00010 both represent decimal 2
    - Example: 110 and 11110 both represent decimal -2

Unsigned versus Signed

Binary	Unsigned value (decimal)	Signed value (decimal)
0000	0 **(zero)**	0 **(zero)**
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7 **(largest positive value)**
1000	8	-8 **(largest negative value)**
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15 **(largest positive value)**	-1 **(smallest negative value)**

Addition of Unsigned Numbers

A (2 bits)	B (2 bits)	SUM = A + B DECIMAL	SUM BINARY	SUM and CARRYOUT BINARY
00	00	0 + 0 = 0	00	000
01	00	1 + 0 = 1	01	001
10	00	2 + 0 = 2	10	010
11	00	3 + 0 = 3	11	011
00	01	0 + 1 = 1	01	001
01	01	1 + 1 = 2	10	010
10	01	2 + 1 = 3	11	011
11	01	3 + 1 = 4	00	100
00	10	0 + 2 = 2	10	010
01	10	1 + 2 = 3	11	011
10	10	2 + 2 = 4	00	100
11	10	3 + 2 = 5	01	101
00	11	0 + 3 = 3	11	011
01	11	1 + 3 = 4	00	100
10	11	2 + 3 = 5	01	101
11	11	3 + 3 = 6	10	110

RED indicates loss of information (i.e. incorrect answer) if carryout not kept and just tried to zero-pad SUM

Unsigned Addition: No Carryout

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity adder_unsigned is
  port(
    a : in STD_LOGIC_VECTOR(1 downto 0);
    b : in STD_LOGIC_VECTOR(1 downto 0);
    sum : out STD_LOGIC_VECTOR(1 downto 0));
end adder_unsigned;

architecture dataflow of adder_unsigned is

begin
  sum <= a + b;
end dataflow;
```


Unsigned Addition: With Carryout

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity adder_unsigned_carryout is
  port(
    a : in STD_LOGIC_VECTOR(1 downto 0);
    b : in STD_LOGIC_VECTOR(1 downto 0);
    sum : out STD_LOGIC_VECTOR(1 downto 0);
    cout : out STD_LOGIC );
end adder_unsigned_carryout;

architecture dataflow of adder_unsigned_carryout is
  signal tempsum : std_logic_vector(2 downto 0);
begin
  tempsum <= ('0' & a) + ('0' & b); -- pad with '0' before addition
  sum <= tempsum(1 downto 0);
  cout <= tempsum(2);
end dataflow;
```

Addition of Signed Numbers

A (2 bits)	B (2 bits)	SUM = A + B DECIMAL	SUM BINARY	SUM and CARRYOUT BINARY
00	00	0 + 0 = 0	00	000
01	00	1 + 0 = 1	01	001
10	00	-2 + 0 = -2	10	110
11	00	-1 + 0 = -1	11	111
00	01	0 + 1 = 1	01	001
01	01	1 + 1 = 2	10	010
10	01	-2 + 1 = -1	11	111
11	01	-1 + 1 = 0	00	000
00	10	0 + -2 = -2	10	110
01	10	1 + -2 = -1	11	111
10	10	-2 + -2 = -4	00	100
11	10	-1 + -2 = -3	01	101
00	11	0 + -1 = -1	11	111
01	11	1 + -1 = 0	00	000
10	11	-2 + -1 = -3	01	101
11	11	-1 + -1 = -2	10	110

RED indicates loss of information (i.e. incorrect answer) if carryout not kept and just tried to sign-extend SUM

Signed Addition: No Carryout

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_SIGNED.all;

entity adder_signed is
  port(
    a : in STD_LOGIC_VECTOR(1 downto 0);
    b : in STD_LOGIC_VECTOR(1 downto 0);
    sum : out STD_LOGIC_VECTOR(1 downto 0));
end adder_signed;

architecture dataflow of adder_signed is

begin
  sum <= a + b;
end dataflow;
```

Signed Addition: With Carryout

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_SIGNED.all;

entity adder_signed_carryout is
  port(
    a : in STD_LOGIC_VECTOR(1 downto 0);
    b : in STD_LOGIC_VECTOR(1 downto 0);
    sum : out STD_LOGIC_VECTOR(1 downto 0);
    cout : out STD_LOGIC );
end adder_signed_carryout;

architecture dataflow of adder_signed_carryout is
  signal tempsum : std_logic_vector(2 downto 0);
begin
  tempsum <= (a(1) & a) + (b(1) & b); -- sign extend BEFORE addition, very important
  sum <= tempsum(1 downto 0);
  cout <= tempsum(2);
end dataflow;
```

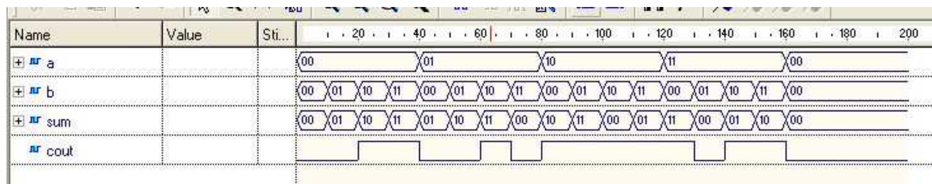

Testbench: Signed Addition with Carry Out

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

entity adder_signed_carryout_tb is
end adder_signed_carryout_tb;

architecture TB_ARCHITECTURE of adder_signed_carryout_tb is
          component adder_signed_carryout
          port(
                    a : in std_logic_vector(1 downto 0);
                    b : in std_logic_vector(1 downto 0);
                    sum : out std_logic_vector(1 downto 0);
                    cout : out std_logic );
          end component;
          signal a : std_logic_vector(1 downto 0) := "00";
          signal b : std_logic_vector(1 downto 0) := "00";
          signal sum : std_logic_vector(1 downto 0);
          signal cout : std_logic;
```

Tesbench cont'd

```
begin
          UUT : adder_signed_carryout
                    port map (
                              a => a,
                              b => b,
                              sum => sum,
                              cout => cout
                    );
          process
          begin
                    for i in 0 to 3 loop
                              for j in 0 to 3 loop
                                        wait for 10 ns;
                                        b <= b + 1;
                              end loop;
                              a <= a + 1;
                    end loop;
                    wait;
          end process;
end TB_ARCHITECTURE;
```


Waveform



Multiplication

- N-bit unsigned number x M-bit unsigned number results in N+M bit unsigned number
  - Example: 3 bits x 4 bits = 7 bits
  - 111 (7) x 1111 (15) = 1101001 (105) → most positive x most positive needs 7 bits
- N-bit signed number x M-bit signed number results in N+M bit unsigned number
  - Example: 3 bits x 4 bits
  - 100 (-4) x 1000 (-8) = 0100000 (+32) → most negative x most negative needs 7 bits **(this is the only scenario which requires the full output range)**
  - 100 (-4) x 0111 (7) = [1]100100 (-28) → most negative x most positive needs 6 bits, [ ] indicates not necessary
  - 011 (3) x 0111 (7) = [0]010101 (21) → most positive x most positive needs 6 bits
  - 011 (3) x 1000 (-8) = [1]101000 (-24) → most positive x most negative needs 6 bits

Multiplication of signed and unsigned numbers (1)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity multiply is
    port(
        a : in STD_LOGIC_VECTOR(15 downto 0);
        b : in STD_LOGIC_VECTOR(7 downto 0);
        cu : out STD_LOGIC_VECTOR(23 downto 0);
        cs : out STD_LOGIC_VECTOR(23 downto 0)
       );
end multiply;

architecture dataflow of multiply is

SIGNAL sa: SIGNED(15 downto 0);
SIGNAL sb: SIGNED(7 downto 0);
SIGNAL sres: SIGNED(23 downto 0);

SIGNAL ua: UNSIGNED(15 downto 0);
SIGNAL ub: UNSIGNED(7 downto 0);
SIGNAL ures: UNSIGNED(23 downto 0);
```

Since using both signed and unsigned data types, don't use std_logic_unsigned/signed. Do all conversions by hand.

Multiplication of signed and unsigned numbers (2)

```
begin

-- signed multiplication
   sa <= SIGNED(a);
   sb <= SIGNED(b);
   sres <= sa * sb;
   cs <= STD_LOGIC_VECTOR(sres);

-- unsigned multiplication
   ua <= UNSIGNED(a);
     ub <= UNSIGNED(b);
   ures <= ua * ub;
   cu <= STD_LOGIC_VECTOR(ures);

end dataflow;
```


Integer Types

Operations on signals (variables)
of the integer types:

INTEGER, NATURAL,

and their sybtypes, such as

TYPE day_of_month IS **RANGE 0 TO 31;**

are synthesizable in the range

$-(2^{31}-1)$ .. 2^{31} -1 for INTEGERs and their subtypes

0 .. 2^{31} -1 for NATURALs and their subtypes

Integer Types cont'd

Operations on signals (variables)

of the integer types:

INTEGER, NATURAL,

are less flexible and more difficult to control

than operations on signals (variables) of the type

STD_LOGIC_VECTOR

UNSIGNED

SIGNED, and thus

are recommended to be avoided by beginners unless necessary.