VHDL Modeling for Synthesis

Combinational Logic Circuits

VHDL synthesis references

- "LeonardoSpectrum HDL Synthesis Manual"
  - Access from Leonardo "Help" menu or
  - mgcdocs _bk_leospec.pdf
- Smith Text: Chapter 12
  - VHDL and Verilog synthesis examples
  - Synopsys Design Compiler (DC) used for book examples
  - Synopsys DesignVision is the GUI interface to DC

Common types of synthesized circuits

- Combinational logic circuits (12.6.2 – 12.6.5)
  - random logic (12.6.2)
  - multiplexers (12.6.3)
  - decoders (12.6.4)
- Arithmetic functions (12.6.5, 12.6.9, 12.6.10)
- Sequential logic (registers) (12.6.6)
  - synchronous & asynchronous inputs
- Shift registers (12.6.8)
- Finite state machines (12.7, 12.7.2)
- Memory synthesis (12.8, 12.8.2)

Combinational logic

```
-- Specify behavior via concurrent signal assignments
entity Gates is
    port (a, b,c: in BIT; d: out BIT);
end Gates;
architecture behavior of Gates is
     signal e: BIT;
Begin
          -- concurrent signal assignment statements
          e <= (a and b) xor (not c);  -- synthesize gate-level ckt
          d <= a nor b and (not e);    -- in target technology
end;
```

VHDL “Process” Construct

- Allows conventional programming language methods to describe circuit behavior
- Supported language constructs (“sequential statements”) – only allowed within a process:
  - variable assignment
  - if-then-else (elsif)
  - case statement
  - while (condition) loop
  - for (range) loop

Process Format

[label:] process (*sensitivity list*)
declarations
begin
sequential statements
end process;

- Process statements executed once at start of simulation
- Process halts at "end" until an event occurs on a signal in the "sensitivity list"

Combinational logic via process

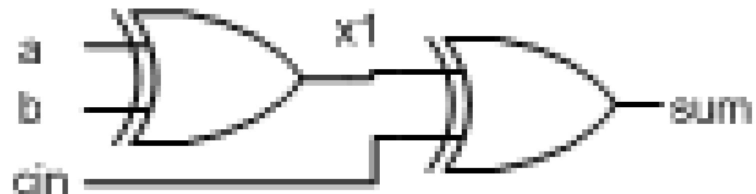
```
-- All signals referenced in process must be in the sensitivity list.
entity And_Good is
    port (a, b: in BIT; c: out BIT);
end And_Good;
architecture Synthesis_Good of And_Good is
   begin
     process (a,b)      -- gate sensitive to events on (a, b)
     begin
         c <= a and b;  -- c updated on a or b  "events"
    end process;
end;
```

Combinational logic via process

```
-- This example produces unexpected results.
entity And_Bad is
    port (a, b: in BIT; c: out BIT);
end And_Bad;
architecture Synthesis_Bad of And_Bad is
   begin
     process (a)          -- this should be process (a, b)
     begin
         c <= a and b;  -- will not react to changes in b
    end process;
end Synthesis_Bad;
-- synthesis tool may generate a flip flop, triggered by signal a
```

Structural architecture example
(no "behavior" specified)

```
architecture structure of full_add1 is
   component xor        -- declare component to be used
         port (x,y: in bit;
                   z: out bit);
   end component;
   for all: xor use entity work.xor(eqns); -- if multiple arch's
   signal x1: bit;      -- signal internal to this component
begin
   G1: xor port map (a, b, x1);          -- instantiate 1st xor gate
   G2: xor port map (x1, cin, sum);      -- instantiate 2nd xor gate
   …add circuit for carry output…
end;
```



Half adder “structural” architecture

```
architecture Netlist of Half_Adder is
    -- declare all components to be used
    component MyXor     -- component with formals
        port (A_Xor,B_Xor : in BIT; Z_Xor : out BIT);
    end component;
    component MyAnd    -- component with formals
        port (A_And,B_And : in BIT; Z_And : out BIT);
    end component;

begin
   -- instantiate components:
   Xor1: MyXor port map (X, Y, Sum);   -- instance with actuals
   And1: MyAnd port map (X, Y, Cout); -- instance with actuals
end;
```

Cells defined in an ASIC vendor library

```
-- AND2 Cell
entity AndGate is
    port (Ain_1, Ain_2 : in BIT; A_out : out BIT); -- formals
end;
architecture Simple of AndGate is
begin
     Aout <= Ain_1 and Ain_2;
end;
-- XOR2 Cell
entity XorGate is
    port (Xin_1, Xin_2 : in BIT;  Xout : out BIT); -- formals
end;
architecture Simple of XorGate is
begin
     Xout <= Xin_1 xor  Xin_2;
end;
```

File adk.vhd contains models of all ADK standard cells.

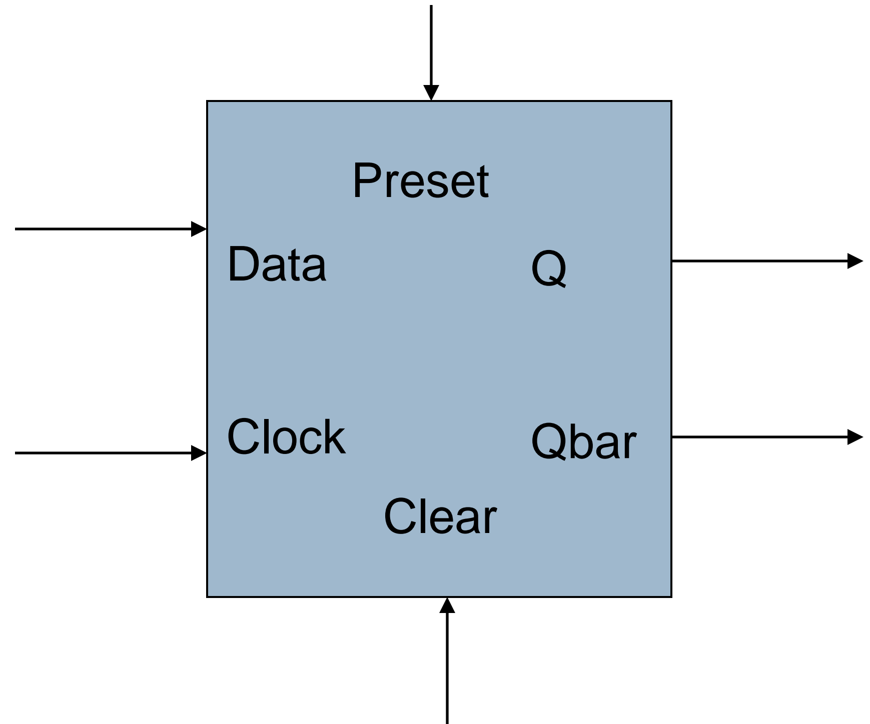
Associating signals with formal ports

```
component AndGate port
        (Ain_1, Ain_2 : in BIT;        -- formal parameters
                    Aout : out BIT);
 end component;


-- positional association:
A1:AndGate port map (X,Y, Z1);
-- named association:
A2:AndGate port map (Ain_2=>Y, Aout=>Z2, Ain_1=>X);
-- both (positional must begin from leftmost formal):
A3:AndGate port map (X, Aout => Z3, Ain_2 => Y);
```

Example: D flip-flop

```
entity DFF is
  port (Preset: in bit;
        Clear: in bit;
        Clock: in bit;
        Data: in bit;
        Q: out bit;
        Qbar: out bit);
end DFF;
```

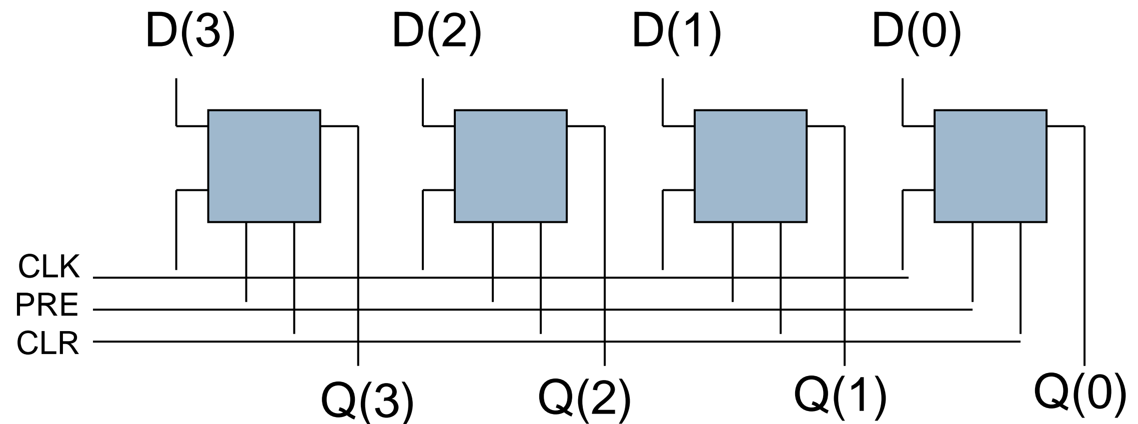


7474 D flip-flop equations

```
architecture eqns of DFF is
        signal A,B,C,D: bit;
        signal QInt, QBarInt: bit;
begin
        A <= not (Preset and D and B) after 1 ns;
        B <= not (A and Clear and Clock) after 1 ns;
        C <= not (B and Clock and D) after 1 ns;
        D <= not (C and Clear and Data) after 1 ns;
        Qint <= not (Preset and B and QbarInt) after 1 ns;
        QBarInt <= not (QInt and Clear and C) after 1 ns;
        Q <= QInt;          -- Can drive but not read "outs"
        QBar <= QBarInt;    -- Can read & drive "internals"
end;
```

4-bit Register (Structural Model)

```
entity Register4 is
  port ( D: in bit_vector(0 to 3);
         Q: out bit_vector(0 to 3);
         Clk: in bit;
         Clr: in bit;
         Pre: in bit);
end Register4;
```



Register Structure

```
architecture structure of Register4 is
    component DFF           -- declare library component to be used
       port (Preset: in bit;
             Clear: in bit;
             Clock: in bit;
             Data: in bit;
             Q: out bit;
             Qbar: out bit);
      end component;
      signal Qbar: bit_vector(0 to 3); -- dummy for unused FF outputs
begin
    -- Signals connect to ports in order listed above
    F3: DFF port map (Pre, Clr, Clk, D(3), Q(3), Qbar(3));
    F2: DFF port map (Pre, Clr, Clk, D(2), Q(2), Qbar(2));
    F1: DFF port map (Pre, Clr, Clk, D(1), Q(1), Qbar(1));
    F0: DFF port map (Pre, Clr, Clk, D(0), Q(0), Qbar(0));
end;
```

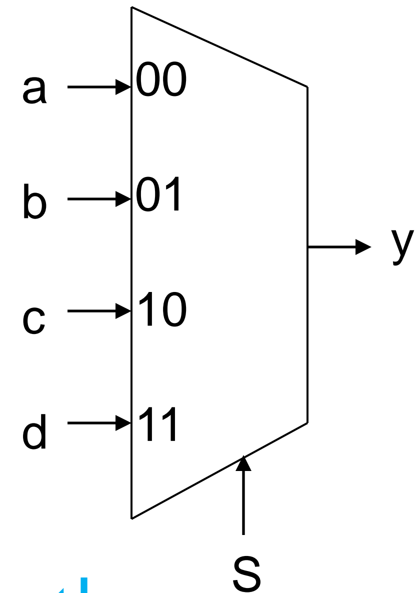
Short cut – “generate” statement

```
for k in 0 to 3 generate
    F: DFF port map (Pre, Clr, Clk, D(k), Q(k), Qbar(k));
end generate;
```

- Expanded similar to programming language “macro”
- Generates multiple copies of any given statement(s)
- Value of k inserted where specified
- Iteration number k is appended to each label F
- Result is identical to previous example
- Use for repeated constructs

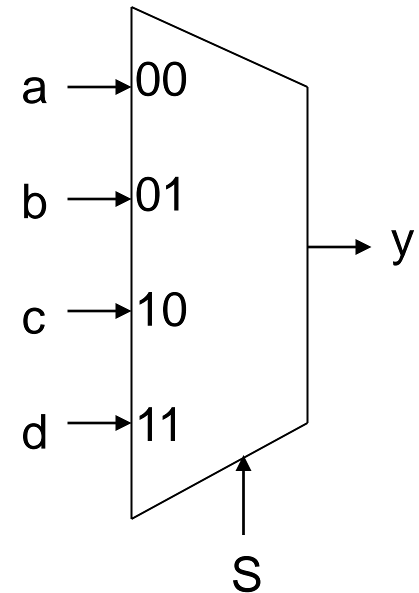
Conditional Signal Assignment

```
    signal a,b,c,d,y: std_logic;
    signal S: std_logic_vector(0 to 1);
begin
    with S select
        y <= a after 1 ns when "00",
             b after 1 ns when "01",
             c after 1 ns when "10",
             d after 1 ns when "11";
--Alternative "default":   d after 1 ns when others;
```



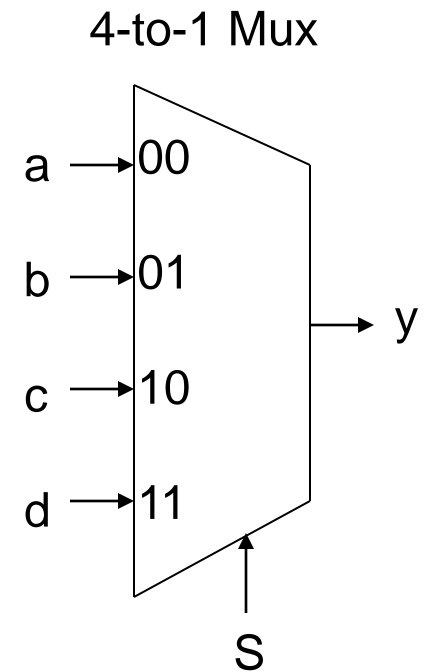
32-bit-wide 4-to-1 multiplexer

```
    signal a,b,c,d,y: bit_vector(0 to 31);
    signal S: bit_vector(0 to 1);
begin
    with S select
        y <=   a after 1 ns when "00",
               b after 1 ns when "01",
               c after 1 ns when "10",
               d after 1 ns when "11";
--a,b,c,d,y can be any type, as long as they are the same
```



Conditional Signal Assignment – Alternate Format

```
y <=  a after 1 ns when (S="00") else
      b after 1 ns when (S="01") else
      c after 1 ns when (S="10") else
      d after 1 ns;
```



Use any boolean expression for each condition:

y <= a after 1 ns when (F='1') and (G='0') …

Unconstrained Vectors

- Allows one generic model with different sizes:

```
entity mux is
        port ( a,b: in   bit_vector;  -- unconstrained
               c:  out bit_vector;  -- unconstrained
               s:  in    bit );
end mux;
architecture x of mux is
begin
        c <= a when (s='0') else b;
end;
```

Vector constrained when instantiated

```
signal s1,s2: bit;
signal a5,b5,c5: bit_vector (0 to 4);
signal a32,b32,c32: bit_vector (0 to 31);
component mux
 port (a,b: in bit_vector;  -- unconstrained
          c: out bit_vector;
          s: in bit );
end component;
begin
  M5:   mux port map (a5,b5,c5,s1);        -- 5-bit mux
  M32: mux port map (a32,b32,c32,s2); -- 32-bit mux
```

Multiplexer using an array

```
entity Mux8 is
   port (InBus :  in    STD_LOGIC_VECTOR(7 downto 0);
         Sel :        in    INTEGER range 0 to 7;
         OutBit : out STD_LOGIC);
end Mux8;
architecture Synthesis_1 of Mux8 is
begin
        OutBit <= InBus(Sel);
end;

-- if sel is std_logic_vector, must convert to integer
-- OutBit <= InBus(TO_INTEGER ( UNSIGNED (Sel) ) );
```

Multiplexer: Using “case” statement
(12.6.3)

```
entity Mux4 is
    port (i: BIT_VECTOR(3 downto 0);
            sel: BIT_VECTOR(1 downto 0); s: out BIT);
end Mux4;
architecture Synthesis_1 of Mux4 is
begin
 process(sel, i) begin
     case sel is
        when "00" => s <= i(0); when "01" => s <= i(1);
        when "10" => s <= i(2); when "11" => s <= i(3);
     end case;
end process; end Synthesis_1;
-- case statement must be exhaustive to avoid priority questions
```

Binary decoder function (12.6.4)

```
library IEEE;  use IEEE.NUMERIC_STD.all;
use IEEE.STD_LOGIC_1164.all;
entity Concurrent_Decoder is
port ( enable : in BIT;
        Din : in STD_LOGIC_VECTOR (2 downto 0);
        Dout : out STD_LOGIC_VECTOR (7 downto 0));
end Concurrent_Decoder;
```

Decoder using shifter

-- inputs enable, Din (3-bit vector), Dout (8-bit vector)

```
with enable select
   Dout <= STD_LOGIC_VECTOR(UNSIGNED'(
              shift_left("00000001",TO_INTEGER (UNSIGNED(Din)))) )
                 when '1',
              "00000000" when '0',
              "11111111" when others; -- default for synthesis tool
```

Decoder – alternate model

```
process (Din, enable)
    variable T : STD_LOGIC_VECTOR(7 downto 0);
begin
   if (enable = '1') then
         T := "00000000";
         T( TO_INTEGER (UNSIGNED(Din))) := '1';
          Dout <= T ;
   else
          Dout <= (others => ‘0');
   end if;
end process;
```

Synthesizing arithmetic circuits
(12.6.5, 12.6.9, 12.6.10)

- Leonardo recognizes overloaded operators and generates corresponding circuits:

  “+”,“-”,“*”, and “abs”
- Special operations:

  “+1”,“-1”, unary “-”
- Relational Operators:

  “=“,“/=“,“<“,“>“,“<=“,“>=“
- Use “ranged integers” instead of unbound to minimize generated logic.

  signal i : integer range 0 to 15;

Leonardo restrictions

- Non-integer data types (std_logic_vector) require operator overloading to produce arithmetic circuits (IEEE libarary packages)
- Multiply operator "*" will produce a multiplier, but more efficient technology-specific modules may be better.
- Divide operator "/" only works if dividing by a power of 2, unless using a technology-specific module

Behavioral model of an adder

```
entity adder is
   port ( a:       in integer;  -- abstract data type
          b:       in integer;
          sum:     out integer);
end adder ;
architecture behave of adder is
   begin
      sum <= a + b;  --synthesis should produce adder ckt
   end;
```

Adders/subtracters automatically generated for integer data

```
variable a,b,c: integer;
c := a + b;    -- produces 32-bit adder (signed)

variable a,b,c: integer range 0 to 255;
c := a + b;    -- produces 8-bit adder (unsigned)
```

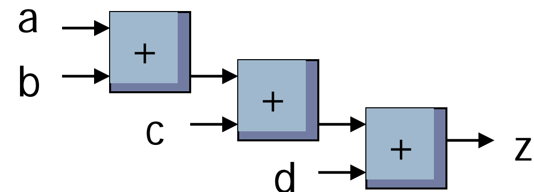
Constant operands result in reduced logic by removing logic due to hard-wired values.

Ex: `c := a + 5;`

Multiple adder structures

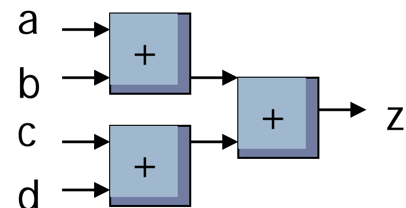
z <= a + b + c + d;

-- 3 adders stacked 3 deep



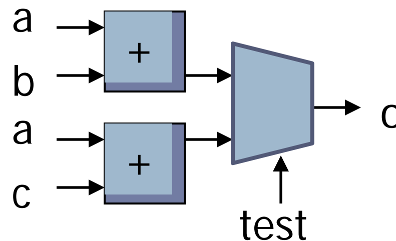
z <= (a + b) + (c + d);

-- 3 adders stacked 2 deep



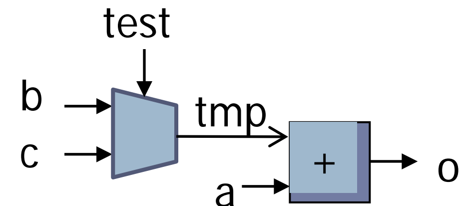
Resource sharing for mutually-exclusive operations

```
process (a,b,c,test) begin
    if (test=TRUE) then
        o <= a + b ;    -- either this evaluates
    else
        o <= a + c ;    -- or this evaluates
    end if ;
end process ;
-- Leonardo generates two adders & one mux
```



Equivalent model

```
process (a,b,c,test) begin
    variable tmp : integer range 0 to 255 ;
begin
    if (test=TRUE) then -- mux will select b or c
        tmp := b ;
    else
        tmp := c ;
    end if ;
    o <= a + tmp ;  -- mux output added to a
end process ;
-- one adder and one mux generated
```



IEEE Std. 1076.3 Synthesis Libraries

Text: Chap 10.12, 12.6.5, 12.6.9

- Support for arithmetic models
- numeric_std
  - defines UNSIGNED and SIGNED as arrays of std_logic

```
type SIGNED is array(NATURAL range <>) of STD_LOGIC;
type UNSIGNED is array(NATURAL range <>) of STD_LOGIC;
```

  - defines standard arithmetic/relational operators on these types
- numeric_bit
  - same as above except SIGNED/UNSIGNED are arrays of bit
- std_logic_arith *(from Synopsis)*
  - Non-standard predecessor of numeric_std/numeric_bit

NUMERIC_STD package contents

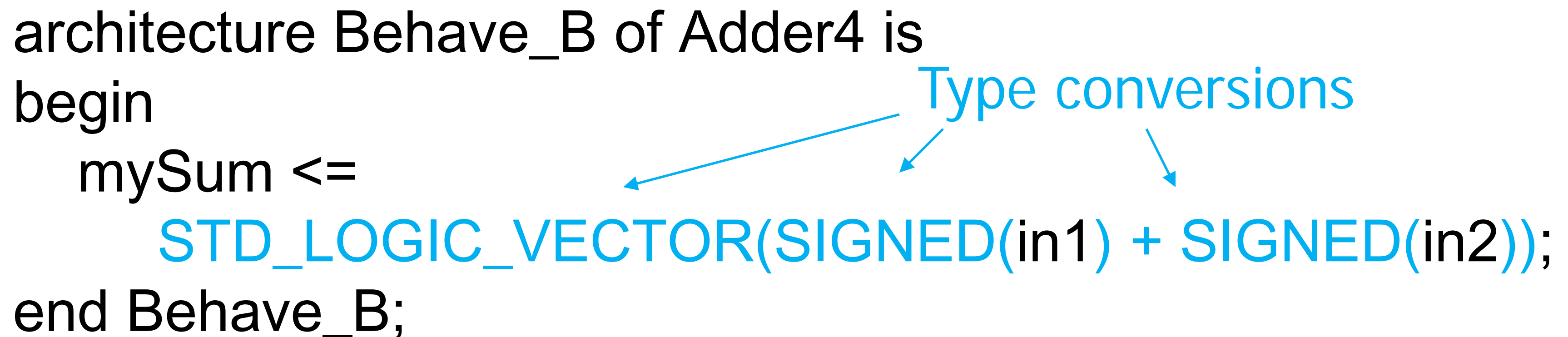
- Arithmetic functions: + - * / rem mod
  - Combinations of operands allowed:
    - SIGNED + SIGNED return SIGNED
    - SIGNED + INTEGER return SIGNED
    - INTEGER + SIGNED return SIGNED
- 6 relational operators for above operands
- TO_INTEGER(signed), TO_SIGNED(integer,#bits)
- RESIZE(signed,#bits) – changes # bits
- SHIFT_LEFT/RIGHT, ROTATE_LEFT/RIGHT
- PLUS: all of above with UNSIGNED/NATURAL

Conversion of “closely-related” types

arrays of same dimension + elements of same type

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
entity Adder4 is
   port ( in1, in2  : in   STD_LOGIC_VECTOR(3 downto 0) ;
            mySum : out STD_LOGIC_VECTOR(3 downto 0) ) ;
end Adder4;

architecture Behave_B of Adder4 is
begin
    mySum <=
        STD_LOGIC_VECTOR(SIGNED(in1) + SIGNED(in2));
end Behave_B;
```



Arithmetic with NUMERIC_STD package

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
entity Adder4 is
   port ( in1, in2  : in   UNSIGNED(3 downto 0) ;
             mySum : out UNSIGNED(3 downto 0) ) ;
end Adder4;

architecture Behave_B of Adder4 is
begin
    mySum <= in1 + in2; -- overloaded '+‘ operator
end Behave_B;
```

Example

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
ENTITY counter IS
   port( Q:  out std_logic_vector(3 downto 0);
....

ARCHITECTURE behavior OF counter IS
   signal Qinternal:  unsigned(3 downto 0);
begin
      Qinternal <= Qinternal + 1;              -- + from numeric_std
     Q <= std_logic_vector(Qinternal);     -- unsigned->std_logic
    ....
```

Handling Overflow (Carry)

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

entity Adder_1 is
   port ( A, B  : in   UNSIGNED(3 downto 0) ;
             C :     out UNSIGNED(4 downto 0) ) ; -- C(4) = carry
end Adder_1;

architecture Synthesis_1 of Adder_1  is
begin
    C <= ('0' & A) + ('0' & B); -- leading '0' to balance # bits
end Behave_B;
-- can also use:  C <= resize(A,5) + resize(B,5)
```

Add/Sub "Accumulator"

```
-- result_t , xin, addout are UNSIGNED
with addsub select – combinational add/sub
      addout <=  (xin + result_t) when '1',
                 (xin - result_t)  when '0',
                 (others => '-')   when others;

process (clr, clk) begin  -- register part
        if (clr = '0') then
                result_t <= (others => '0');
        elsif rising_edge(clk) then
                result_t <= addout;
        end if;
end process;
-- "when others" creates exhaustive list of choices
```

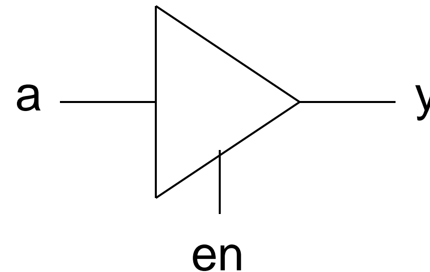
Simplified "accumulator" model

```
-- signal addsub eliminated – circuit will be the same
process (clr, clk) begin
        if (clr = '0') then
                result_t <= (others => '0');
        elsif rising_edge(clk) then
              case addsub is
                    when '1'      => result_t <=  (xin + result_t);
                    when '0'      => result_t <=  (xin - result_t);
                    when others => result_t <=  (others => '-');
              end case;
        end if;
 end process;
```

Tristate buffer example (incorrect)

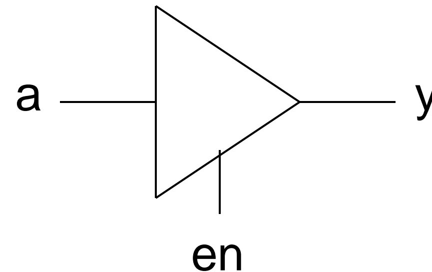
```
library ieee;
use ieee.std_logic_1164.all;
entity tristate is
  port ( a: in bit;
             y: out std_logic;
             en: in bit);
end tristate;
architecture a1 of tristate is
begin
   y <= a after 1 ns when (en='1') else
         'Z' after 1 ns;
end;

--Type mismatch between y and a
```



Tristate buffer example (correct)

```
library ieee;
use ieee.std_logic_1164.all;
entity tristate is
  port ( a: in bit;
              y: out std_logic;
              en: in bit);
end tristate;
architecture a1 of tristate is
begin
   y <= '0' after 1 ns when (en='1') and (a='0') else
          '1' after 1 ns when (en='1') and (a='1') else
          'Z' after 1 ns;
end;
```



Tristate bus buffer example

```
library ieee;
use ieee.std_logic_1164.all;
entity tristate is
  port ( a: in bit_vector(0 to 7);
          y: out std_logicvector(0 to 7);
          en: in bit);
end tristate;
architecture a1 of tristate is
begin
   y <= to_stdlogicvector(a) after 1 ns when (en='1') else
          "ZZZZZZZZ" after 1 ns;
end;
```