Logic Synthesis with VHDL
System Synthesis

Bob Reese
Electrical Engineering Department
Mississippi State University
VHDL Packages

⇒ A VHDL package is a mechanism for collecting procedures, functions, constants, and components for future re-use.

⇒ A package contains a package declaration followed by a package body.

→ Package declaration

    package  package_name is

        {  external constant, procedure, function, component declarations }

    end  package_name;

→ Package body

    package body  package_name is

        { constant, procedure, function, component definitions }

    end  package_name;

⇒ Any items in the package declaration are available for external use. There can be items in the package body which are not in the package declaration; these items are only available for use within the package.
Example VHDL Package

Library IEEE; use IEEE.std_logic_1164.all;

package iscas is

    procedure ripple_adder (a,b: in std_logic_vector; cin: in std_logic;
                           sum: inout std_logic_vector; cout: out std_logic);

end iscas;

package body iscas is

    function xor3 (a,b,c: in std_logic) return std_logic is
        begin
            return (a xor b xor c);
        end xor3;

    procedure ripple_adder (a,b: in std_logic_vector; cin: in std_logic;
                            sum: inout std_logic_vector; cout: out std_logic) is

        variable c: std_logic_vector((a’high–a’low+1) downto 0);
        begin
            c(0) := cin;
            for i in 0 to (a’high–a’low) loop
                sum(i+sum’low) := xor3 (a(i+a’low), b(i+b’low), c(i) );
                c(i+1) := (a(i+a’low) and b(i+b’low)) or 
                          (c(i) and (a(i+a’low) or b(i+b’low)));
            end loop;
            cout := c(c’high);
            end ripple_adder;

end iscas;

VHDL Functions

⇒ General form:

function function_name (parameter list) return return_type is
{variable declarations}
begin
{sequential statements}
end function_name;

function xor3 (a,b,c: in std_logic) return std_logic is
begin
return (a xor b xor c);
end xor3;

⇒ A VHDL function computes a return value based upon its parameter list.

→ All parameters passed to a VHDL function must be of mode in; i.e, the function is not allowed to modify any of the function parameters.

→ The default class of the elements in a parameter list for either procedures or functions is variable.

→ Signals can be passed in the parameter list; in this case the parameter list would look like:
(signal a, b, c: std_logic)

→ More on the difference between variables and signals will be given later.
VHDL Procedures

⇒ General form:

procedure procedure_name ( parameter list ) is
{variable declarations}
begin
{sequential statements}
end procedure_name;

⇒ The ripple_adder procedure implements the ripple carry adder used in previous examples.

⇒ The ripple_adder procedure uses the local xor3 function defined within the package.

    sum(i+sum’low) := xor3 (a(i+a’low), b(i+b’low), c(i));

⇒ For generality, the input parameters ’a’ and ’b’ as well as the output ’sum’ are declared as unconstrained array types; i.e., no array bounds are given for the std_logic_vector type.

    → Allows any width vector to be passed as a parameter.

    → Array indices must be computed using the ’low attribute as an offset in order to achieve independence from the actual array indices which are passed in.
Signals vs Variables

⇒ Only signals are used as the connection ports for VHDL entities.

→ Variables are declared within process blocks, procedures, and functions.

→ Signals can only be declared within architecture bodies; they can be passed as parameters to functions and procedures.

⇒ Signals are assigned via ”<=”; Variables are assigned via ”:=”.

⇒ From a simulation point of view:

→ Signals have events occurring on them and this event history is tracked via an internal event list.

→ Signal assignment can be delayed such as:
  
a <= ’1’ after 10 ns

→ Variable assignment is always immediate.
  
a <= ’1’;

→ Signals require more overhead in terms of storage and simulation time than variables. A general rule of thumb is to use variables wherever possible.

⇒ From a synthesis point of view, both variables and signals can turn into internal circuit nodes.
Using the *ripple_adder* Procedure

Library IEEE;
use IEEE.std_logic_1164.all;
use work.iscas.all;

entity adder_test is
port (
signal a,b: in std_logic_vector (15 downto 0);
signal cin: in std_logic;
signal sum: out std_logic_vector (15 downto 0);
signal cout: out std_logic
);
end adder_test;

architecture behavior of adder_test is
begin
process (a,b,cin)
variable temp_sum: std_logic_vector (sum’range);
variable temp_cout: std_logic;
begin
    *ripple_adder*(a, b, cin, temp_sum, temp_cout);
    sum <= temp_sum;
cout <= temp_cout;
end process;
end behavior;

'work' is the default library name for packages. The 'all' keyword says to use all externally available package items in the 'iscas' package.

Call the 'ripple_adder' procedure. Variables are used as parameters within 'ripple_adder' so variables must be passed in as arguments. These variables are then assigned to the target signals.
A Carry Select Adder

Each stage computes part of the sum. Typically, the stage sizes increase; so a 16 bit adder stage sizes might be 4, 5, 7 = total of 16 bits. Ripple adders are used for stage adders.

........ etc.
procedure **carry_select_adder**
    (groups: iarray; a,b: in std_logic_vector; cin: in std_logic;
        sum: inout std_logic_vector; cout: out std_logic) is

    variable low_index, high_index :integer;
    variable temp_sum_a, temp_sum_b : std_logic_vector(sum’range);
    variable carry_selects :std_logic_vector(groups’range);
    variable carry_zero :std_logic_vector(groups’low to (groups’high–1));
    variable carry_one :std_logic_vector(groups’low to (groups’high–1));

    begin
        low_index := 0;
        for i in groups’low to groups’high loop
            high_index := (groups(i)–1) + low_index;
            if (i = 0) then  — first group, just do one ripple–carry
                ripple_adder (a(high_index downto low_index), b(high_index downto low_index),
                cin, sum(high_index downto low_index), carry_selects(0)   );
            else
                — need to do two ripple carry adders then use mux to select
                ripple_adder (a(high_index downto low_index), b(high_index downto low_index),
                ‘0’, temp_sum_a(high_index downto low_index), carry_zero(i–1));
                ripple_adder (a(high_index downto low_index), b(high_index downto low_index),
                ‘1’, temp_sum_b(high_index downto low_index), carry_one(i–1));
            if (carry_selects(i–1) = ’0’) then
                sum(high_index downto low_index) := temp_sum_a(high_index downto low_index);
            else
                sum(high_index downto low_index) := temp_sum_b(high_index downto low_index);
            end if;
            carry_selects(i) := (carry_selects(i–1) and carry_one(i–1) ) or carry_zero(i–1);
        end loop;
        cout := carry_selects(groups’high);
        end ripple_adder;
iscas Package Declaration

Library IEEE;
use IEEE.std_logic_1164.all;

package iscas is

type IARRAY is array (natural range <>) of integer;

procedure ripple_adder (a,b: in std_logic_vector; cin: in std_logic;
sum: inout std_logic_vector; cout: out std_logic);

procedure carry_select_adder
  (groups: iarray; a,b: in std_logic_vector; cin: in std_logic;
sum: inout std_logic_vector; cout: out std_logic);

end iscas;

⇒ We need to declare an array type for integers; call this IARRAY. This type will be used to pass in an integer array to the carry_select_adder procedure; the integer array will be define the stage sizes for the adder.

⇒ Since xor3 is to be local to the iscas package; it is not in the package declaration. However, if it was to be made externally available, its declaration would be:

  function xor3 (a,b,c: in std_logic) return std_logic;
Using the `carry_select_adder` Procedure

Library IEEE;
use IEEE.std_logic_1164.all;
use work.iscas.all;

entity adder_cs is
  port(
    signal a,b: in std_logic_vector (15 downto 0);
    signal cin: in std_logic;
    signal sum: out std_logic_vector(15 downto 0);
    signal cout: out std_logic
  );
end adder_cs;

architecture behavior of adder_cs is
begin
  process (a,b,cin)
    variable temp_sum: std_logic_vector (sum’range);
    variable temp_cout: std_logic;
  constant groups: iarray(0 to 2) := (4,5,7);
  begin
    carry_select_adder(groups,a,b,cin,temp_sum, temp_cout);
    sum <= temp_sum;
    cout <= temp_cout;
  end process;
end behavior;

Define local constant array of integers to define the stage sizes for the adder. 4 + 5 + 7 = 16 bits. Must be a constant array so that stage sizes are known at compile time.
Library IEEE;
use IEEE.std_logic_1164.all;
use work.iscas.all;

entity adder_test is
generic ( N : integer := 16);
port ( signal a,b: in std_logic_vector (N–1 downto 0);
signal cin: in std_logic;
signal sum: out std_logic_vector(N–1 downto 0);
signal cout: out std_logic
);
end adder_test;

architecture behavior of adder_test is
begin
 process (a,b,cin)
variable temp_sum: std_logic_vector (sum’range);
variable temp_cout: std_logic;
begin
 ripple_adder(a, b, cin, temp_sum, temp_cout);
 sum <= temp_sum;
cout <= temp_cout;
end process;
end behavior;
VHDL Generic lists (cont.)

⇒ VHDL generic lists are used in entity declarations for passing static information.

→ Typical uses of generics are for controlling bus widths, feature inclusion, message generation, timing values.

⇒ A generic will usually have a specified default value; this value can be overridden via VHDL configurations or by vendor-specific back-annotation methods.

→ Generics offer a method for parameterizing entity declarations and architectures. Because the method of specifying generic values (other than defaults) can be vendor specific, generics will not be covered further in this tutorial.
Operator Overloading

Library IEEE; use IEEE.std_logic_1164.all;

package genmux is

— 2/1 version, 1 bit inputs
function mux (a,b: std_logic; sel: std_logic) return std_logic;

— 2/1 version, N bit inputs
function mux (a,b: std_logic_vector; sel: std_logic) return std_logic_vector;

— 3/1 version, 1 bit inputs
function mux (a,b,c: std_logic; sel: std_logic_vector) return std_logic;

— 3/1 version, N bit inputs
function mux (a,b,c: std_logic_vector; sel: std_logic_vector) return std_logic_vector;

— 4/1 version, 1 bit inputs
function mux (a,b,c,d: std_logic; sel: std_logic_vector) return std_logic;

— 4/1 version, N bit inputs
function mux (a,b,c,d: std_logic_vector; sel: std_logic_vector) return std_logic_vector;

end genmux;

package body genmux is

function mux (a,b: std_logic; sel: std_logic) return std_logic is
variable y: std_logic;
begin
  y := a;
  if (sel = '1') then  y := b;  end if;
  return(y);
end mux; — 2/1 version, 1 bit inputs

function mux (a,b: std_logic_vector; sel: std_logic) return std_logic_vector is
variable y: std_logic_vector(a'range);
begin
  y := a;
  if (sel = '1') then  y := b;  end if;
  return(y);
end mux; — 2/1 version, N bit inputs
function \texttt{mux} (a,b,c: std\_logic; sel: std\_logic\_vector) return std\_logic is
variable y: std\_logic;
begin
  y := '–'; — Don’t care for default state
  if (sel = "00") then y := a; end if;
  if (sel = "01") then y := b; end if;
  if (sel = "10") then y := c; end if;
return(y);
end mux; — 3/1 version, 1 bit inputs

function \texttt{mux} (a,b,c: std\_logic\_vector; sel: std\_logic\_vector) return std\_logic\_vector is
variable y: std\_logic\_vector(a’range);
begin
  y := (others => '–'); — Don’t care for default state
  if (sel = "00") then y := a; end if;
  if (sel = "01") then y := b; end if;
  if (sel = "10") then y := c; end if;
return(y);
end mux; — 3/1 version, N bit inputs

function \texttt{mux} (a,b,c,d: std\_logic; sel: std\_logic\_vector) return std\_logic is
variable y: std\_logic;
begin
  y := d;
  if (sel = "00") then y := a; end if;
  if (sel = "01") then y := b; end if;
  if (sel = "10") then y := c; end if;
return(y);
end mux; — 4/1 version, 1 bit inputs

function \texttt{mux} (a,b,c,d: std\_logic\_vector; sel: std\_logic\_vector) return std\_logic\_vector is
variable y: std\_logic\_vector(a’range);
begin
  y := d;
  if (sel = "00") then y := a; end if;
  if (sel = "01") then y := b; end if;
  if (sel = "10") then y := c; end if;
return(y);
end mux; — 4/1 version, N bit inputs

end genmux;
Test of ’mux’ Function

Library IEEE;
use IEEE.std_logic_1164.all;
use work.genmux.all;

entity muxtest is
port (  
signal a,b,c: in std_logic;
signal s_a: in std_logic_vector(1 downto 0);
signal y: out std_logic;
signal j,k,l: in std_logic_vector(3 downto 0);
signal s_b: in std_logic_vector(1 downto 0);
signal z: out std_logic_vector(3 downto 0)
);
end muxtest;

architecture behavior of muxtest is

begin

  y <= mux (a,b,c,s_a);
  z <= mux (j,k,l,s_b);

end behavior;

*The mux operator is overloaded; the correct mux function is chosen by doing template matching on the parameter lists.*
BlackJack Dealer

⇒ This example will be a BlackJack Dealer circuit (example taken from *The Art of Digital Design*, Prosser & Winkel, Prentice–Hall).

⇒ One VHDL model will be written for the control and one for the datapath. A schematic will be used to tie these two blocks together.

→ Later, a VHDL structural model will be used to connect the blocks.

⇒ Control:

→ Four States:
   - Get — get a card
   - Add — add current card to score
   - Use — use an ACE card as 11
   - Test — see if we should stand or if we are broke

⇒ Datapath:

→ 5–bit register for loading score; needs a synchronous clear.

→ Mux for choosing between card value, plus 10 and minus 10.

→ Adder for adding card with current score.

→ ACE card detect (an ACE card has value ’0001’)

→ Comparator logic for checking is score is greater than 16 or greater than 21.
BlackJack Dealer Control

GET

BlackJack Dealer

card.rdy.sync

wait for card

MUX: Select Card
REG: Load score
T \rightarrow hit

F \rightarrow stand.out
F \rightarrow broke.out

F \rightarrow card.rdy.delay

wait until button is lifted

stand + broke

ADD

MUX: Select Card
REG: Load score

Add card value

acecard

F \rightarrow ace11flag

T \rightarrow broke.out

F \rightarrow ace11flag.out

F \rightarrow ace11flag.out

REG: Clear score

Start game over

F \rightarrow ace11flag.out

To TEST state

Use ACE as 11

F \rightarrow ace11flag.out

System Design with VHDL
BlackJack Dealer Control (cont)

- If the test is true, go to the next state. If false, go to the state where "MUX: Select SUB10 REG: Load score F → ace11flag.out".
- If the score is greater than 21 and the ace flag is true, output "broke.out". Otherwise, if the score is greater than 16 and less than 21, go to the state "stand.out".
- Otherwise, if the score is greater than 21 and we can't adjust an ACE value, go to the state "broke.out".
- To get the next state, go to the state "To GET state".
BlackJack Datapath

Card Switches

MUX

ACE Finder

ACE11 Flag

Miscellaneous Flip Flops to be included in Control

Score

ADDER

REG

Comparator

Score16gt

Score21gt

Stand

Stand.out

Broke

Broke.out

Card.Rdy Button

Card.rdy.sync

Card.rdy.delay
entity bjdpath is port (  
  signal clk, reset_b, load, clear_b: in std_logic;  
signal sel: in std_logic_vector(1 downto 0);  
signal card: in std_logic_vector(3 downto 0);  
signal acecard, score16gt, score21gt: out std_logic;  
signal score: out std_logic_vector(4 downto 0)  
); end bjdpath;
architecture behavior of bjdpath is  
signal adder_out, score_in: std_logic_vector(4 downto 0)  
mux_out, score_out : std_logic_vector(4 downto 0);  
— temporary signal for carries  
signal c: std_logic_vector (5 downto 0);
begin
  score_state: process(clk, reset_b)  
  begin
    if (reset_b = '0') then score_out <= "00000";
    elsif (clk'event and clk = '1') THEN
      score_out <= score_in;
    end IF;
  end process score_state;
  — combinational logic for score register
  score_in <= "00000" when (clear_b = '0') else
    adder_out when (load = '1') else
    score_out;

  State process for score register flip–flops.
  Combinational logic for Score Register
adder process

adder_out <= score_out + mux_out

adder:process (score_out, mux_out)
begin
    c(0) <= '0';
    for i in score_out’range loop
        adder_out(i) <= score_out(i) xor mux_out(i) xor c(i);
        c(i+1) <= (score_out(i) and mux_out(i)) or
            (c(i) and (score_out(i) or mux_out(i)));
    end loop;
end process adder;

mux_out <= "01010" when (sel = B”00”) else
    "10110” when (sel = B”10”) else
    ’0’ & card;

acecard <= '1’ when (card = B”0001”) else ’0’；

score <= score_out;

score16gt <= ’1’ when (score_out > B”10000”) else ’0’;
score21gt <= ’1’ when (score_out > B”10101”) else ’0’;

end behavior;

MUX for card, plus 10, minus 10.

ACE Finder

Comparators
VHDL File for BlackJack Control

entity bjcontrol is port (  
  signal clk, reset_b, card_rdy, acecard: in std_logic;  
  signal score16gt, score21gt: in std_logic;  
  signal hit, broke, stand: out std_logic;  
  signal sel: out std_logic_vector(1 downto 0);  
  signal score_clear_b, score_load: out std_logic  
); end bjcontrol;

architecture behavior of bjcontrol is

— declare internal signals here
signal n_state, p_state : std_logic_vector(1 downto 0);  
signal ace11flag_pstate, ace11flag_nstate: std_logic;  
signal broke_pstate, broke_nstate: std_logic;  
signal stand_pstate, stand_nstate: std_logic;  
signal card_rdy_dly, card_rdy_sync: std_logic;

— state assignments are as follows
constant get_state: std_logic_vector(1 downto 0) := B”00”;  
constant add_state: std_logic_vector(1 downto 0) := B”01”;  
constant test_state: std_logic_vector(1 downto 0) := B”10”;  
constant use_state: std_logic_vector(1 downto 0) := B”11”;  
constant add_10_plus: std_logic_vector(1 downto 0) := B”00”;  
constant add_card: std_logic_vector(1 downto 0) := B”01”;  
constant add_10_minus: std_logic_vector(1 downto 0) := B”10”;
VHDL File for BlackJack Control (cont.)

begin

— state process to implement flag flip–flops and FSM state
state: process(clk, reset_b)
begin
if (reset_b = '0') then  p_state <= "00";
elsif (clk’event and clk = '1') THEN
    p_state <= n_state;
    ace11flag_pstate <= ace11flag_nstate;
    broke_pstate <= broke_nstate;
    stand_pstate <= stand_nstate;
    card_rdy_dly <= card_rdy_sync;
    card_rdy_sync <= card_rdy;
END IF;
end process state;
broke <= broke_pstate;
stand <= stand_pstate;

State process to define flip–flops for various flags and finite state machine.
VHDL File for BlackJack Control (cont.)

comb: process (p_state, ace11flag_pstate, broke_pstate, stand_pstate, acecard, card_rdy_dly, card_rdy_sync, score16gt, score21gt)
begin
sel <= B"00";
score_load <= '0'; score_clear_b <= '1';
hit <= '0'; n_state <= p_state;
ace11flag_nstate <= ace11flag_pstate;
stand_nstate <= stand_pstate; broke_nstate <= broke_pstate;

case p_state is
when get_state =>
if (card_rdy_sync = '0') then hit <= '1';
elsif (card_rdy_dly = '0') then
stand_nstate <= '0'; broke_nstate <= '0';
if (stand_pstate = '1' or broke_pstate = '1') then
score_clear_b <= '0';
ace11flag_nstate <= '0';
end if;
end if;

when add_state =>
sel <= add_card; score_load <= '1';
if (acecard = '1' and ace11flag_pstate = '0') then
n_state <= use_state;
else n_state <= test_state;
end if;
end case;
when use_state =>
    sel <= add_10_plus;
    score_load <= ’1’;
    ace11flag_nstate <= ’1’;
    n_state <= test_state;

when test_state =>

    if (score16gt = ’0’) then
        n_state <= get_state;
    elsif (score21gt = ’0’) then
        stand_nstate <= ’1’;
        n_state <= get_state;
    elsif (ace11flag_pstate = ’0’) then
        broke_nstate <= ’1’;
        n_state <= get_state;
    else
        sel <= add_10_minus;
        score_load <= ’1’;
        ace11flag_nstate <= ’0’;
    end if;

    when OTHERS => n_state <= p_state;

end case;
end process comb;
end behavior;
Top Level Schematic for Dealer
Blackjack Dealer Simulation

Initial Score of zero

\[ \text{Score} = 5 + 0 = 5 \]

\[ \text{Score} = 8 + 5 = 13 \]

\[ \text{Score} = 13 + 4 = 17; \text{ score is } > 16 \text{ so we STAND.} \]
Blackjack Dealer Simulation (cont.)

Assertion of STAND from previous game causes us to start new game.

First card is an 'Ace'.
Next card is a '2'.
Next card is a '9'.
Final card is a '10' (facecard).

Score = 11 + 2 = 13

We will use the first 'Ace' as a value of '11'.

13 + 9 > 21 so we break; however, we have an 'Ace' so we can treat it as a value of '1'; the new score is: 
1 + 2 + 9 = 12.

12 + 10 > 21 so we are 'BROKE'.

Bob Reese 5/95  System–31  System Design with VHDL
Structural VHDL

⇒ You do not have to use a schematic to connect VHDL blocks. You can write a **structural** VHDL model which ties the blocks together.

⇒ **Pros:**

  → When you synthesize the design all of the VHDL blocks are flattened (collapsed into one block) and it is possible that the resulting logic may be more efficient.

  → The structural VHDL code is more portable to other design systems than a schematic.

⇒ **Cons:**

  → Writing structural VHDL code can be more error prone than creating a schematic (very easy to misplace a net when you don’t have a ‘picture’ to go by).

  → The resulting flattened netlist can be more difficult to debug.
Structural VHDL for BlackJack Player

entity bj_struct is port (  
  signal reset_b, clk, card_rdy : in std_logic;  
  signal card: in std_logic_vector(3 downto 0);  
  signal stand, broke,hit: out std_logic;  
  signal score: out std_logic_vector(4 downto 0);  
end bj_struct;

architecture structure of bj_struct is  
component bjcontrol port (  
  signal clk resets_b: in std_logic;  
  signal card_rdy, acecard: in std_logic;  
  signal score16gt, score21gt: in std_logic;  
  signal hit, broke,stand: out std_logic;  
  signal sel: out std_logic_vector(1 downto 0);  
  signal score_clear_b: out std_logic;  
  signal score_load: out std_logic;  
end component;  

component bjdpath  
port (  
  signal clk, reset_b: in std_logic;  
  signal load, clear_b: in std_logic;  
  signal sel: in std_logic_vector(1 downto 0);  
  signal card: in std_logic_vector(3 downto 0);  
  signal acecard, score16gt: out std_logic;  
  signal score21gt: out std_logic;  
  signal score: out std_logic_vector(4 downto 0);  
end component;
Structural VHDL for BlackJack Player (cont)

```vhdl
signal load_net, clear_net, acecard_net : std_logic;
signal sel_net : std_logic_vector (1 downto 0);
signal s21gt_net, s16gt_net: std_logic;
begin
  c1: bjcontrol
  port map (
    clk => clk,
    reset_b => reset_b,
    card_rdy => card_rdy,
    acecard => acecard_net,
    score16gt => s16gt_net,
    score21gt => s21gt_net,
    hit => hit, broke => broke, stand => stand,
    sel => sel_net,
    score_clear_b => clear_net,
    score_load => load_net);
  
  c2: bjdpath
  port map (
    clk => clk,
    reset_b => reset_b,
    load => load_net,
    clear_b => clear_net,
    sel => sel_net,
    card => card,
    acecard => acecard_net,
    score16gt => s16gt_net,
    score21gt => s21gt_net,
    score => score );
end structure;
```

- Internal signal declaration for those nets not connected to external ports.
- Each component used in the design is given along with its port map.
- 'c1' is the component label, 'bjcontrol' gives the component type.
- Only two components in this design.
Results of \textit{bj} \textit{struct} Synthesis