Synthesizable VHDL
Outline

• Introduction to synthesis
• Synthesis levels
• VHDL templates and implementation
• Synthesis issues
• Conclusion
VHDL

- VHDL - Very high speed integrated circuit Hardware Description Language.

- Designed for simulation
  - discrete time simulation
  - based on processes

- Now is also used for synthesis
Design flow

- Design Entry
- Design Synthesis
- Design Implementation
- Back Annotation
  - Download to a Xilinx Device
- Design Verification
  - Functional Simulation
  - Static Timing Analysis
  - Timing Simulation
  - In-Circuit Verification
Synthesis

• Synthesis is a general term that describes the process of transformation of the model of a design, usually described in a hardware description language (HDL), from one level of behavioral abstraction to a lower, more detailed behavioral level.

• These transformations try to improve upon a set of objective metrics (e.g., area, speed, power dissipation) of a design, while satisfying a set of constraints (e.g., I/O rates, MIPS, sample period) imposed on it.
Levels of Synthesis

• Behavioral Synthesis - synthesis of abstract behavior or control-flow behavior from a high level algorithm description
• RTL Synthesis - synthesis of register-transfer structure from abstract, control-flow, or register-transfer behavior
• Logic Synthesis - synthesis of gate-level logic (an implementation) from register-transfer structure or Boolean equations
Levels of Synthesis

VHDL Descriptions at:

- Algorithmic Level
- Combinational logic, clocked registers, state machines, memories
- Boolean equations
- Structural gate net lists, technology libraries

Behavioral Synthesis

RTL Synthesis

Logic Synthesis

input

output
Behavioral level

\[ f \leq (a + b) \times e; \]
\[ g \leq (a + b) \times (c + d); \]
RTL level

Data Path Behavior

A(7:0) B(7:0) C(7:0) D(7:0)

Register R1

Register R2

E(7:0)

+ 

Register R3

Register R4


Control Flow
(not all shown)

Select A; Load R1;

Select B; Load R2;

Add; Load R4;

RTL Structure
(not all shown)

A(7:0) B(7:0) C(7:0) D(7:0)

R1

R2

ALU

RTL Control Flow
(not all shown)

Select A; Load R1;

Select B; Load R2;

Add; Load R4;
Logic level
Behavioral level

• The input to a behavioral synthesis system describes the dataflow, input/output constraints, memory accesses, and user defined constraints (sample period, latency).

• The output of the behavioral synthesis system provides the data path, registers, memories, I/O structures, and a state machine-based controller that satisfies the same test bench as the input.
Behavioral level example

Library IEEE
Use IEEE.Numeric_STD.all
Entity complexb_nty is
  port ( datain_p: in unsigned (4 downto 0);
       output_re, output_im :
       out unsigned (9 downto 0));
end complexb_nty;

Architecture complexb_a of complexb_nty is
begin
  behave: process
  variable a, b, c, d : unsigned (4 downto 0);
  begin
    calc: loop
    wait until clk’event and clk =‘1’;
    a := datain_p;
    wait until clk’event and clk =‘1’;
    b := datain_p;
    wait until clk’event and clk =‘1’;
    c := datain_p;
    wait until clk’event and clk =‘1’;
    d := datain_p;
  -- Computation begins
    Output_re <= a*c - b*d ;
    Output_im <= a*d - b*c ;
    end loop;
  end process;
end complexb_a;

I/O Constraints

IEEE Synthesis Package

Dataflow

User specifies arithmetic ops independent of clock (clk) event.

Single process architecture.
RTL level synthesis

• Input to the RTL synthesis environment includes the number of data path components (adders, multipliers,), the mapping of operations to data path components, and a controller (finite state machine) that contains the detailed schedule (related to clock edge) of computational, I/O, and memory operations.

• Output of the RTL synthesis provides logic level implementations that can be evaluated through the optimization of the data path, memory, and controller components, individually or in a unified manner, through mapping to gate-level component libraries.
**Datapath**: process
variable reg_1, reg2 : unsigned (4 downto 0);
variable reg_3, reg_4, reg_5: unsigned (9 downto 0);
begin
wait until clk'event and clk'event='1';
if le_reg1 = '1' then reg1 := a;
end if;
if le_reg2 = '1' then reg2 := b;
end if;
if le_reg3 = '1' then reg3 := c;
end if;
if le_mult = '1' then reg4 := reg1 * reg2;
end if;
if le_add = '1' then reg5 := reg4 + reg3;
end if;
if le_z = '1' then output <= reg5;
end if;
end process;

**Fsm**: process(clk, reset);
begin
if clk'event and clk = '1' then
ir reset = '1' then cur_state <= s0;
else cur_state <= next_state;
end if;
end process;

**state_machine**: process (cur_state, reset)
begin
case cur_state is
when s0 => next_state <= s1;
  le_reg1 <= '1';
when s1 => next_state <= s2;
  le_reg2 <= '1';
when s2 => next_state <= s3;
  le_reg3 <= '1';
when s3 => next_state <= s4;
  le_mult <= '1';
when s4 => next_state <= s5;
  le_add <= '1';
when s5 => next_state <= s0;
  le_z <= '1'; end case;
end process;
Development diagram

- VHDL Specification
- RTL Simulation
- Behavior Synthesis
- RTL Synthesis
- Logic Synthesis
- Test Insertion/Gate-level Timing/Optimization
- Floor plan, place & route

Verification

To Vendor
Practical templates

• combinatorial circuits and logic
  • multiplexer and de-multiplexer
  • counter, comparators, etc.

• register and sequential circuits
  • synchronous and asynchronous registers
  • shift registers

• State machines and controllers
Decoder or Multiplexer

using case statement

process (sel, a, b, c, d)
begin
  case sel is
    when "00" => mux_out <= a;
    when "01" => mux_out <= b;
    when "10" => mux_out <= c;
    when "11" => mux_out <= d;
    when others => null;
  end case;
end process;

when statement

Mx_out <= a when "00" else
  b when "01" else
  c when "10" else
  d;
Register

with asynchronous reset

```
process (clk, reset)
begin
  if reset='1' then -- asynchronous reset
    dout <= '0';
  elsif (clk'event and clk='1') then
    dout <= din;
  end if;
end process;
```

with synchronous reset

```
process (clk)
begin
  if clk'event and clk='1' then
    if reset='1' then
      dout <= '0';
    else
      dout <= din;
    end if;
  end if;
end process;
```
Finite state machine

• A state machine has three basic parts
  • Next State Logic
  • Output Logic
  • Memory

Mealy machine model

Moore machine model
Finite state machine (cont.)

• Use enumerated type to encode state variables

Signal preset_state, next_state: state_type;
Type state_type is (idle, init, test, add, shift);

• User can choose state encoding type in synthesis tool
  • Sequential
  • One Hot
  • Gray
  • Others

• enum_encoding attribute can be used to specify encoding directly

<table>
<thead>
<tr>
<th>Binary</th>
<th>Gray code</th>
<th>One-hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>00000001</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>00000100</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>00001000</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>00001000</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
<td>00010000</td>
</tr>
<tr>
<td>101</td>
<td>111</td>
<td>00100000</td>
</tr>
<tr>
<td>110</td>
<td>101</td>
<td>01000000</td>
</tr>
<tr>
<td>111</td>
<td>100</td>
<td>10000000</td>
</tr>
</tbody>
</table>
Finite state machine (example)

process (clk, reset)
begin
  if reset='1' then
    cur_state <= fst_state;
  elsif (clk'event and clk='1') then
    cur_state <= next_state;
  end if;
end process;

ns_logic : process (cur_state, inputs)
begin
  next_state <= fst_state;
  case cur_state is
    when fst_state =>
      next_state <= s_something;
    when s_something =>
      next_state <= fst_state;
    when others =>
      null;
  end case;
end process ns_logic;

o_logic : process (cur_state, inputs)
begin
  out_sig <= '1';
  case cur_state is
    when fst_state =>
      out_sig <= '0';
    when s_something =>
      out_sig <= '1';
    when others =>
      null;
  end case;
end process o_logic;
Single Port Memory

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

entity sp_ram is
  generic
    (
      DATA_WIDTH : integer := 8;
      ADDR_WIDTH : integer := 8
    );
  port
    (
      clk : in std_logic;
      addr : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
      data_w : in std_logic_vector(DATA_WIDTH - 1 downto 0);
      data_r : out std_logic_vector(DATA_WIDTH - 1 downto 0);
      nwr : in std_logic;
      ncs : in std_logic
    );
end sp_ram;

architecture beh of sp_ram is
  subtype ram_entry is std_logic_vector(DATA_WIDTH - 1 downto 0);
  type ram_type is array(0 to (2 ** ADDR_WIDTH) - 1) of ram_entry;
  signal ram : ram_type;
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if ncs = '0' then
        if nwr = '0' then
          ram(to_integer(unsigned(addr))) <= data_w;
        else
          data_r <= ram(to_integer(unsigned(addr)));
        end if;
      end if;
    end if;
  end process;
end beh;
The standard for VHDL Register Transfer Level Synthesis is IEEE Std 1076.6-1999.

- basic synthesis library
- arithmetic operations
- works with unsigned words

The purpose of the standard is to define a syntax and semantics for VHDL RTL synthesis. It defines a subset of IEEE 1076 (VHDL) that is intended to be used in common by all RTL synthesis tools.

Use ieee.std_logic_1164.all;
Use ieee.std_logic_arith.all;
Use ieee.std_logic_unsigned.all;
Design issues

- Inferring latches
- Incomplete sensitivity list
- Loop and Generate statements
- Sequential assignment
- Technology consideration
Inferring latches

Code and circuit example

Process (sel, a, b, c)
Begin
  case sel is
    when "00" => mux_out <= a;
    when "01" => mux_out <= b;
    when "10" => mux_out <= c;
  end case;
End process;

Process (sel, a, b, c)
Begin
  mux_out <= a;
  case sel is
    when "00" => mux_out <= a;
    when "01" => mux_out <= b;
    when "10" => mux_out <= c;
  end case;
End process;

Uncovered values
Latch generation
Incomplete sensitivity list

- Missing signal at sensitivity list
- different behavior in function and time simulation.
- it is difficult to find this error

```
Process (sel)
Begin
  Case sel is
    when "00" => mux_out <= a;
    when "01" => mux_out <= b;
    when "10" => mux_out <= c;
    when "11" => mux_out <= d;
    when others => null;
  end case;
end process;
```
Loop and Generate statement

- Loops and Generate - this syntax construction can be used only for constant integer generation or loops.

```
dis_mem : for i in 0 to DATA_WIDTH-1 generate
    INST_RAM16X1D : RAM16X1D
    port map (
        D => data_in(i),
        WE => write_en,
        WCLK => clk,
        A0 => addr_a(0),
        A1 => addr_a(1),
        A2 => addr_a(2),
        A3 => addr_a(3),
        DPRA0 => addr_b(0),
        DPRA1 => addr_b(1),
        DPRA2 => addr_b(2),
        DPRA3 => addr_b(3),
        SPO => data_out_a(i),
        DPO => data_out_b(i)
    );
end generate;
```
**Sequential assignment**

- Order of signal assignments does not affect synthesis and simulation as long as the sensitivity list is complete. However, if using signals and variables the order may affect simulation and synthesis.

```vhdl
Sig_s <= not test1_v;
Test1_v := test2_v and B and C;
Test2_v := D and E;
```

Swapping these statements

- Only effects simulation (not synthesis)

```vhdl
Var1_v := B and C;
Var1_v := A and B;
Output_s <= var1_v;
```

Swapping these statements

- Affects synthesis and simulation
Technology consideration

• The developer of a synthesizable behavioral VHDL description MUST (not?!) consider the implementation technology when writing the code
  
  • Most (all?) FPGA families do not have internal tri-state devices - behavioral descriptions that are based on internal tri-state busses will fail in the technology mapping phase
  
•
Design improvements

• Basic issues
  – Timing and placement constraints
  – Using parallel techniques
    » Parallel blocks
    » Pipeline
Parallel blocks

- Basic parallel technique
- Easy for implementation
- Increasing throughput
Pipeline

• Basic parallel technique
• Parallel computing - similar computation time
• Can eliminate wire delay
• Higher frequency can be reach
Conclusion

- Model translation from higher to lower level
- Synthesis levels - use only behavior and RTL levels
- Use only predefined templates
- Issues - inferring latches, incomplete sensitivity list, etc
- Design speed-up - pipeline and other parallel techniques