VHDL for Logic Synthesis
Overview

- Design Flow for Hardware Design
- VHDL coding for synthesis
- General guidelines for hardware designers

This lecture includes the content from:

Nitin Yogi, Modelling for Synthesis with VHDL, Auburn University

Actel HDL Coding Style Guide
System Design Flow

Electronic System Level (ESL) flow – System C – TLM, Verification, Profiling - VISTA

Mixed-signal Wireless Comm
- MATLAB model - floating point
- MATLAB model - fixed point
- RTL coding (VHDL)

Palladium XP
- Simulink flow

Reconfigurable IP Cores – Internal & External (SystemC, MATLAB, VHDL)

Embedded Computing Architectures
- HwSw Partitioning – based on profiling

High Level Synthesis (CtoS, CatapultC, HandelC)

Software flow

Verification flow
- Based on system level verification
- Assertions and formal verification actively used
- Smart testbenches
- FPGA verification
- Palladium verification

ASIC Logic Synthesis (Synopsys), FPGA LS (Xilinx ISE)

ASIC Back-End (CADENCE SE), FPGA P&R (Xilinx ISE)

ASIC DRC & LVS (Cadence Assura, Polyteda)

DfT flow
- BIST for memory and logic
- Scan for logic

Digital design flow (ASIC; FPGA)
ASIC Design flow

1. Applications
2. System Specification
3. New RTL Designs
4. HDL Top Module Definition
5. Simulation
6. Result OK?
7. New synthesis
8. Logic Synthesis
9. Simulation
10. Result OK?
11. New layout run sufficient?
12. Layout Synthesis
13. Simulation
14. Result OK?
15. Final Chip Layout

Test Benches
IP Library
VLSI Levels of Abstraction

- Specification: what the chip does, inputs/outputs
- System Level Modeling: major resources, connections
- Register Transfer Logic: logic blocks, FSMs, memory, connections
- Circuit Level: transistors, parasitics, connections
- Layout: mask layers, polygons
- Logic: gates, flip-flops, latches, connections
Activity Flow in Digital Design

Requirements

- Functional Design
  - Behavioral Simulation

- Register Transfer Level Design
  - RTL Simulation Validation

- Logic Design
  - Logic Simulation Verification
  - Fault Simulation

- Circuit Design
  - Timing Simulation
  - Circuit Analysis

- Physical Design
  - Design Rule Checking

Description for Manufacture
From VHDL Code to Final Design

- Requirements
  - VHDL Model
    - Functional Design
  - VHDL Model
    - Register Transfer Level Design
  - Logic Simulation
    - Synthesis
    - Place and Route
    - Timing Extraction

Behavioral Simulation (VHDL)
Design Views and Abstraction Models

- Process of ASIC design starts with behavioral model, goes over structural until physical model
VHDL could be applied at multiple levels of abstraction

- VHDL can be used to model the circuit of very abstract behavioral level
- Also it can be used for describing the structural netlist
Synthesis Process and different coding Styles

- Synthesis converts RTL model to structural model
- As a result we get some sort of a netlist

---Behavioral (RTL) model

architecture behav of mux is
begin
  p1: process(A,B,S)
  begin
    if (S = '0') then
      Y <= A;
    else
      Y <= B;
    end if;
  end process p1;
end;

---Structural model

architecture behav of mux is
signal Sbar:std_logic
begin
  g1: not port map (Sbar,S);
  g2: and port map (ASbar,A,Sbar);
  g3: and port map (BS,B,S);
  g4: or port map (Y,ASbar,BS);
end;
Why is Understanding of Synthesis Process Important?

Behavioral model
architecture behav of ckt1 is
begin
  p1: process(A,B,S1,S2)
    begin
      if (S1 = '0') and (S2 = '1') then
        Y <= A;
      elsif (S2 = '0') then
        Y <= B;
      --else I do not care
      --what is assigned to Y
      end if;
    end process p1;
end;
Why is Understanding of Synthesis Process Important?

if (S1 = '0') and (S2 = '1') then
    Y <= A;
elsif (S2 = '0') then
    Y <= B;
    --else do not care
end if;
Issues with extra hardware

- Extra redundant components increase area utilization
- Timing closure is more difficult
- Hidden problems can create internal bugs
- More switching activity leading to power loss
- Issues with latches
  Setup and hold timing violations may prevent correct functioning
  Glitches can cause further problems in operation
Why is Understanding of synthesis Process Important?

Corrected behavioral code

architecture behav of ckt1 is
begin
  p1: process(A,B,S1,S2)
  begin
    if (S1 = '0') and (S2 = '1') then
      Y <= A;
    elsif (S2 = '0') then
      Y <= B;
    else
      -- a value is assigned to Y in the “else” clause
      Y <= 'X';
    end if;
  end process p1;
end;

Actual synthesized design

Thumb rule:
Ensure all cases are defined, using “else” clause for combinational scenarios
Types of Synthesized Circuits

- Combinational logic circuits
  - random logic
  - multiplexers
  - Decoders

- Arithmetic functions

- Sequential logic (registers)
  - synchronous & asynchronous inputs

- Shift registers

- Finite state machines

- Memory synthesis

- More advanced circuits (FIFOs, synchronizers, clock gates)
VHDL Simulation Mechanism

- VHDL model is simulated under an event driven simulation kernel (VHDL simulator).

- Simulation is a cyclic process;
  each simulation cycle consists of a signal update and a process execution phase.

- A global clock holds the current simulation time;
  as part of the simulation cycle this clock is incremented with discrete values.

- Current signal values are only updated by the simulator at certain moments during simulation!

- Signal driver contains the projected output waveform of a signal;
  a process that assigns values to a signal will automatically create a driver for that signal;
  projected output waveform is a set of transactions;

- Transaction: pair consisting of a value and a time

- A signal assignment only affects the projected output waveform.
VHDL Simulation Mechanism

- As the current time advances and becomes equal to the time component of the next transaction, the first transaction is deleted and the next becomes the current value of the driver.
- The driver gets a new value.
- Regardless if this value is different or not, the driver and the signal is said to be active during that simulation cycle.
- During each simulation cycle, the current value is updated for active signals.
- If the current value of the signal has changed, an event has occurred on that signal.
- Resolved signal - a signal for which several drivers exist.
- For each resolved signal the designer has to specify an associated resolution function.

*Petru Eles, Lectures
VHDL Coding Styles

• **Structural**

  architecture behav of mux is

  signal Sbar:std_logic

  Begin

  g1: not port map (Sbar,S);
  g2: and port map (ASbar,A,Sbar);
  g3: and port map (BS,B,S);
  g4: or port map (Y,ASbar,BS);

  end;

• **Behavioural**

  architecture behav of mux is

  begin

  p1: process(A,B,S)
  begin

  if (S = '0') then
    Y <= A;
  else
    Y <= B;
  end if;

  end process p1;

  end;

• **Dataflow**

  architecture behav of mux is

  begin

  Y<=A when S=‘0’ else B;

  end;
Delta Delay

- Default signal assignment propagation delay if no delay is explicitly prescribed
  VHDL signal assignments do not take place immediately
  Delta is an infinitesimal VHDL time unit so that all signal assignments can
  result in signals assuming their values at a future time

```
Output <= NOT Input;
-- Output assumes new value in one delta cycle
```

- Supports a model of concurrent VHDL process execution
  Order in which processes are executed by simulator does not affect simulation
  output

Please be careful:
A<=B;
In this case signals A and B are not identical, and there is a delta delay in
between them
Combinational Logic

-- Use concurrent assignment or a process.
-- All signals referenced in a process must be in the sensitivity list.

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; -- can miss changes in b
    end process;
end Synthesis_Bad;

Thumb rule:
All signals referenced in the **combinational** process must be in the sensitivity list
**Variables vs. Signals**

- A signal's value is updated only after a delta cycle interval, but in case of variables the data is updated instantly

```vhdl
count: process (x)
  variable cnt : integer := -1;
begin
  cnt:=x+1;
  b<=cnt;
end process;
```

```vhdl
count: process (x)
  signal cnt : integer := -1;
begin
  cnt<=x+1;
  b<=cnt;
end process;
```

- **Variable could end-up in generation of sequential logic**

```vhdl
count: process (x)
  variable cnt : integer := -1;
begin
  cnt:=cnt+1;
  b<=cnt;
end process;
```

**Thumb rule:**
Do not use initial values of signals and variables; they will be ignored by synthesis tool. Instead, use set and reset signals

**Thumb rule:**
Use variables only for intermediate values for arithmetical operations; avoid using variables to infer a sequential circuit; for this task use signals
entity Mux4 is
port (i: in BIT_VECTOR(3 downto 0);
    sel: in 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;

Thumb rule:
Case statement must be exhaustive
Multiplexer using concurrent signal assignment

architecture Synthesis_2 of Mux4 is
begin
    with sel select
    s <= i(0) when "00",
       i(1) when "01",
       i(2) when "10",
       i(3) when "11";
end Synthesis_2;

Preferable design style!

Thumb rule:
Cover all conditions
Priority encoders

an output signal should be assigned in every branch of the if and case statements to avoid unwanted latch

P1: process (C,I0,I1,I2,I3)
    if C= “00“ then
        Y=I0;
    elsif C= “01“ then
        Y=I1;
    elsif C= “10“ then
        Y=I2;
    else
        Y=I3;
    end if;
End process P1;

Thumb rule:
an output signal should be assigned in every branch of the if and case statements to avoid unwanted latch

Thumb rule:
include all branches of an if statement to avoid latch
Synthesizing arithmetic circuits

- Synthesis tools will generally recognize overloaded operators and generate corresponding circuits: +, -, *, and abs
- Special operations: “+1”, “-1”, unary “-”
- Use ranged integers instead of unbound to minimize generated logic. Ex. signal i: integer range 0 to 15;
- There are packages with embedded synthesizable functions that can support arithmetic functions
  - std_logic_arith
  - numeric_std
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;

Thumb rule:
Use bounded data-types whenever possible

Thumb rule:
Use constants in place of variables whenever possible
Multiple adder structures

\[ z \leq a + b + c + d; \quad \text{--3 adders stacked 3 deep} \]

Synthesis output

\[ z \leq (a + b) + (c + d); \quad \text{--3 adders stacked 2 deep} \]

Synthesis output
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.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;
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.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 Synthesis_1 ;
Sequential adder/subtractor and “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 (clk’event) and (clk='1') then
        result_t<= addout;
    end if;
end process;
Simplified “accumulator” model

--signal addout eliminated –circuit will be the same

process (clr, clk) begin
    if (clr= '0') then
        result_t<= (others => '0');
    elsif (clk'event) and (clk='1') 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;
process (a,b,c,sel) begin
    if (sel='0') then
        z <= a + b ; --either this evaluates
    else
        z <= a + c ; --or this evaluates
    end if ;
end process ;

Synthesis output
Equivalent model

process (a,b,c,sel) begin
  variable tmp: unsigned(0 downto 0) ;
  begin
    if (sel='0') then
      tmp:= b ; --mux will select b or c
    else
      tmp:= c ;
    end if ;
    z <= a + tmp; --mux output added to ‘a’
  end process ;
Latches and Flip-flops

- **Latches**

  ```
  process (EN, D) --Sensitivity list
  begin
      if (EN = '1') then
          Q <= D ;
      end if;
  end process;
  ```

- **Flip-flops**

  ```
  process (clk)
  begin
      if (clk'event and clk= '1') then
          Q <= D ;
      end if;
  end process;
  ```
process (clock, asynchronously_used_signals) begin
  if (boolean_expression) then
    asynchronous signal_assignments
  elsif (boolean_expression) then
    asynchronous signal assignments
  elsif (clock’event and clock = constant) then
    synchronous signal assignments
  end if;
end process;
Example: register with asynchronous reset and preset

process (clock, reset )
begin
  if (reset = ‘1’) then
    q <= ‘0’; --reset has precedence
  elsif (preset = ‘1’) then
    q <= ‘1’; --asyncpreset
  elsif (clock’event and clock =’1’) then
    q <= d ; --synchronous load
  end if ;
end process;
FFs generated from variables: 3-bit shift register example

--External input/output din/dout
process (clk)
  variable a, b: bit;
begin
  if (clk'event and clk = '1') then
    dout <= b;
    b := a;
    a := din;
  end if;
end process;
--note: a, b used before being assigned new values
3-bit shift register example Unexpected resulting structure

process (clk)
  variable a,b: bit;
begin
  if (clk’eventand clk= ‘1’) then
    a := din;
    b := a;
    dout<= b;
  end if;
end process;
--a, b changed before used so values are not stored, so they become “wires”. (Only one flip flop from din -> dout)
Try your examples

- Try with simple circuits:
  - 8-bit counter with Load and Asynchronous Reset
  - Shift register (Shift left, right, rotate)
  - Tri-state buffer

- Bi-directional buffer

**Thumb rule:**
If not explicitly required use a multiplexer instead of an internal tri-state bus

*Why is this so?*
Finite state machines (FSM)

• Model states as enumerated type

• Decouple combinational and sequential logic

• Model “present_state” and “next_state” signals/variables

• Construct two processes to model FSM
  One process updates state with next_state
  One process updates next_state

• Allow synthesis tool to encode states
  (binary, one hot, random, gray code, etc.)

• Consider how initial state will be forced
State machine synthesis issues

- **Two-types of FSM models**
  - Mealy model: outputs = f (inputs, state)
  - Moore model: outputs = f (state)

- “case” more efficient than “if-then-elsif…” to test present_state (later builds a priority encoder)

- **Signal assignment consideration**
  - Assign outputs & next_state for every case/condition.
  - If an output or next_state is not assigned something under a certain condition in the case statement, the synthesis tools will have to preserve the value with extra latches/flip-flops.

- Left-most value of enumeration type is default simulation starting value (use reset to initialize real circuit)
Moore/Mealy FSMs

Moore Format

Mealy Format

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity SM1 is
port ( aIn, clk : in std_logic ; yOut : out std_logic );
end SM1;

architecture Moore of SM1 is

    type state is (s1, s2, s3, s4);
    signal pS, nS : state; pS,

    begin
        process ( aIn, pS ) begin -- next state and output functions aIn, pS)
            case pS is
                when s1 => yOut <= '0'; if ( aIn = '1') nS <= s4; else nS <= s2; end if;
                when s2 => yOut <= '1'; if ( aIn = '1') nS <= s3; else nS <= s1; end if;
                when s3 => yOut <= '1'; nS <= s1;
                when s4 => yOut <= '1'; nS <= s2;
            end case;
        end process;

        process begin
            wait until clk = '1';
            pS <= nS ; -- update state variable on next clock nS;
        end process;
end Moore ;
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity SM1 is
port ( aIn, clk : in std_logic ; yOut : out std_logic );
end SM1;

architecture Mealy of SM1 is
    type state is (s1, s2, s3, s4);
    signal pS, nS : state;
begin
    process (aIn, pS) begin -- output & nS are functions of aIn and pS
        case pS is
            when s1 => if (aIn = '1') then yOut <= '0'; nS <= s4;
                else yOut <= '1'; nS <= s3; end if;
            when s2 => yOut <= '1'; nS <= s3;
            when s3 => yOut <= '1'; nS <= s1;
            when s4 => if (aIn = '1') then yOut <= '1'; nS <= s2;
                else yOut <= '0'; nS <= s1; end if;
        end case;
    end process;
    process begin
        wait until clk = '1';
        pS <= nS ; -- update state variable on next clock nS;
    end process;
end Mealy ;
Memory Synthesis

- **Approaches:**

  Random logic using flip-flops or latches – easy to be used, ineffective in respect to area and power

  Register files in datapaths

  memory standard components – no configurability, hard macros

  memory compilers – one can choose the configuration and architecture, much more optimal then FF based, limited number of access ports
Memory Generator
Involving memories in Code

- For generating memory models one should use memory generators
- Memory instances should be included in the structural VHDL code
- For memory wrappers (glue logic) generate the separate instance
- Normally for behavioral simulation use VHDL memory models
- For back-annotation use verilog memory models
FIFO Implementation

- FIFOs should be implemented as circular buffers
Combinational and sequential logic – Coding Guidelines

- **Avoid the instances with only combinational logic**
  The output signals should be registered

- **For pure combinational path use non-process description style**
  (dataflow)

- **For sequential parts always use flip-flop template**
  For most of the applications is best to use consistently asynchronous reset
Naming conventions

- **Design name and entity name should be the same**
  
  example (vhdl): `design.vhd`
  
  example (verilog): `design.v`
  
  entity `design` is
  
  module `design`

- **Port, signal, process, and instance names should be meaningful**

  clk, data, data, reset, ack, cs, wr, rd, test_si, etc

- **Don’t mix lower and upper case (however VHDL is not case sensitive)**

- **For signals and variables that are active low, this shall be clearly indicated by their name, by suffixing _n**

- **Every process shall have a name; the name shall be formed by suffixing _proc**

- **Architecture name shall be formed by suffixing _arch**
Design Organization & Partitioning

- Don’t mix structural and behavioral code
  Avoid glue logic in structural designs; if necessary put the glue logic in a separate design entity.

- Use comments to describe important issues related to the code functionality.

- Make the header for each entity with corresponding comments

- For large designs try to organise your files to be distributed in separate folders; each folder should contain data related to a larger structural unit of the design.

- Avoid using generics at the top level; it is recommended to use packages with definitions of constants instead of generics

- Make clock dividers and reset synchronisers as separate entities and include them on the top
Architectural Decisions

- When some system is coded on RTL level, the designer has to have in mind the system architecture
- Memory insertion must be considered
- Area trade-off
- Performance trade-off
- Power trade-off
Reset Issue

- Ensure that all the registers in the design are resettable; Non-resettable registers are not testable and their behaviour is hard to debug.

- If the design requirements prefer non-resettable cells make sure to provide proper initialisation procedure in the simulation;

- Asynchronous resets are most commonly used
Don’t mix synchronous and asynchronous resets.

- Use reset synchronizers
Clocking strategy

• **Reasonable clock frequencies are (rule of thumb)**
  for 0.25 um up to 100 MHz,
  for 0.13 um up to 166 MHz.

• **If possible avoid different clock domains**

• **Clock control circuits (gates, divider, multiplexers) should be grouped in the single entity on the top level of design**

• **For high performance design or complex clocking (divided clock domains) use PLL (DLL) in design**

• **Take care about the delta cycles!**
Clock issues and clock-gating

• Don’t mix rising and falling edge flip-flops

• Use glitch-free clock gates for clock gating or special standard cells (if they are available)
Synchronization

- If you have to transfer the data between the different unrelated clock domains use synchronizers. Otherwise you will have problems with metastability.

- Two-flop or single-flop synchronizers for a single bit

- For bus synchronization do not synchronize each bit individually; introduce an “enable” signal and then synchronize this enable signal
Making design testable

• **Provide test_mode signal**
  In this mode all clocks must come directly from PADs without any gating
  In this mode all reset signals must come directly from PADs without any gating or registering

• **DFT strategies commonly used**
  Structural test (Scan test)
  Memory BIST
  Logic BIST

• **Advanced rules for memory insertion, combinational loops, reset definition, complex clocking with DFT**

![Standard and Scan Flip-Flop Diagram]
Test and Verification

- **Writing the good testbench is as much important as making the good design**

- **The input data could be read from the file (textio package)**
The output data should be compared to the golden model (coming from C, MATLAB etc.)

- **The tests should be as much exhaustive as possible**
  Code coverage shall be reported to ensure the quality and the thoroughness of the testbench

- **Use assertions and avoid relying on GUI**
Conclusions

• Writing a VHDL is not the same as writing C-code

• The designer must understand the consequences of particular coding style

• The designer should write the code such that this fully defines resulting hardware after synthesis

• Some guidelines should be followed to have the efficient code generation