Hardware Design with VHDL

An Introduction (created by Ronald Hecht)

Henning Puttnies and Eike Schweißguth

University of Rostock
Institute of Applied Microelectronics and Computer Engineering

April 4, 2016

Outline

Outline

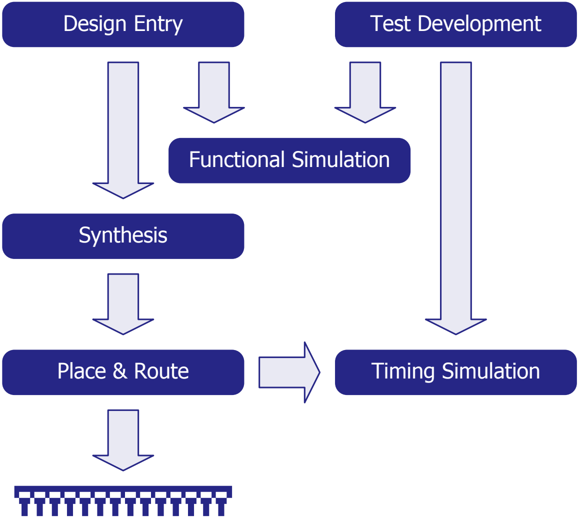
What are HDLs?

- Modeling language for electronic designs and systems
- VHDL, Verilog
- PALASM, ABEL
- Net list languages such as EDIF, XNF
- Test languages such as *e*, Vera
- SystemC for hardware/software co-design and verification

HDL for ...

- Formal description of hardware
- Specification on all abstraction layers
- Simulation of designs and whole systems
- Design entry for synthesis
- Standard interface between CAD tools
- Design reuse

Design methodology



Pros and Cons of HDLs

Pros

- Speeds up the whole design process
- Powerful tools
- Acts as an interface between tools
- Standardized
- Design reuse

Cons

- Learning curve
- Limited support for target architecture

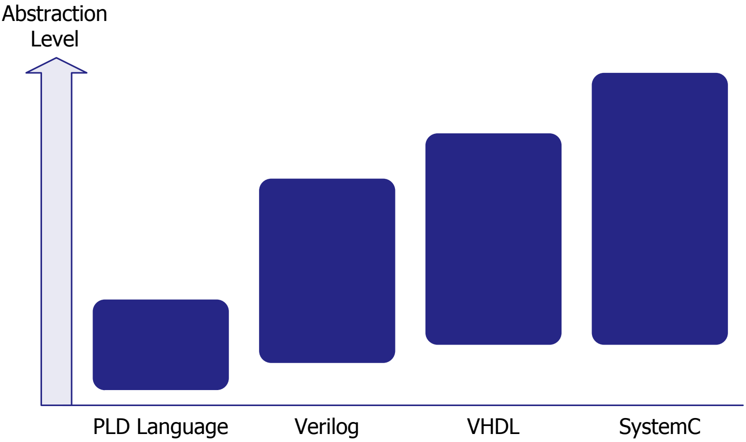
Design Abstraction Levels

Behavior Sequential statements, implicit registers

Data flow Register transfer level (RTL), Parallel statements, explicit registers

Structure Design hierarchy, Wiring components

HDL Abstraction Levels

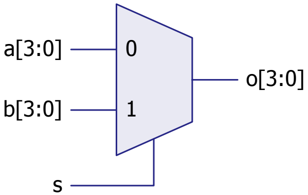


Outline

A First Example

Multiplexer

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

entity mux is
  port (
    a, b : in   std_logic_vector (3 downto 0);
    s    : in   std_logic;
    o    : out  std_logic_vector (3 downto 0));
end mux;

architecture behavior of mux is
begin -- behavior
  o <= a when s = '0' else b;
end behavior;
```



Library

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

- Makes library ieee visible
- Use objects within the library
- Use package std_logic_1164
- Makes std_logic and std_logic_vector visible

Entity

```
entity mux is
  port (
    a, b : in   std_logic_vector (3 downto 0);
    s    : in   std_logic ;
    o    : out  std_logic_vector (3 downto 0));
end mux;
```

- Defines the name of the module
- Describes the interface
  - Name of the ports
  - Direction
  - Type

Architecture

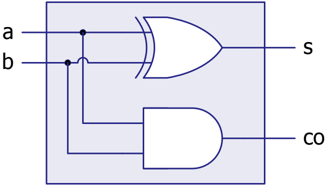
```
architecture behavior of mux is
begin  -- behavior
  o <= a when s = '0' else b;
end behavior;
```

- Implements the module
  - Behavior
  - Structure
  - RTL description
- Entity and architecture are linked by name (mux)
- Name of architecture (behavior)
- More than one architecture per entity allowed

Parallel Processing

Half Adder

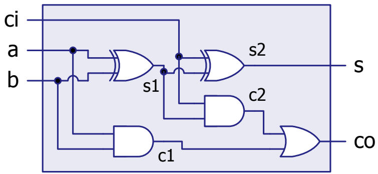
```
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
  port (
    a, b  : in  std_logic;
    s, co : out std_logic);
end half_adder;

architecture rtl of half_adder is
begin -- rtl
  s  <= a xor b;
  co <= a and b;
end rtl;
```



- Implicit gates
- Signals are used to wire gates
- Computes signals s and co from a and b in parallel

Full Adder

```
entity full_adder is
  port (
    a, b, ci : in  std_logic;
    s, co    : out std_logic );
end full_adder;

architecture beh_par of full_adder is
  signal s1, s2, c1, c2 : std_logic;
begin  -- behavior
  -- half adder 1
  s1 <= a xor b;
  c1 <= a and b;
  -- half adder 2
  s2 <= s1 xor ci;
  c2 <= s1 and ci;
  -- evaluate s and co
  s <= s2;
  co <= c1 or c2;
end beh_par;
```



- All statements are processed in parallel
- A signal must not be driven at two places
- Order of statements is irrelevant

Sequential Processing – Processes

```
architecture beh_seq of full_adder is
begin -- beh_seq

  add: process (a, b, ci)
    variable s_tmp, c_tmp : std_logic;
  begin -- process add
    -- half adder 1
    s_tmp := a xor b;
    c_tmp := a and b;
    -- half adder 2
    c_tmp := c_tmp or (s_tmp and ci);
    s_tmp := s_tmp xor ci;
    -- drive signals
    s <= s_tmp;
    co <= c_tmp;
  end process add;

end beh_seq;
```

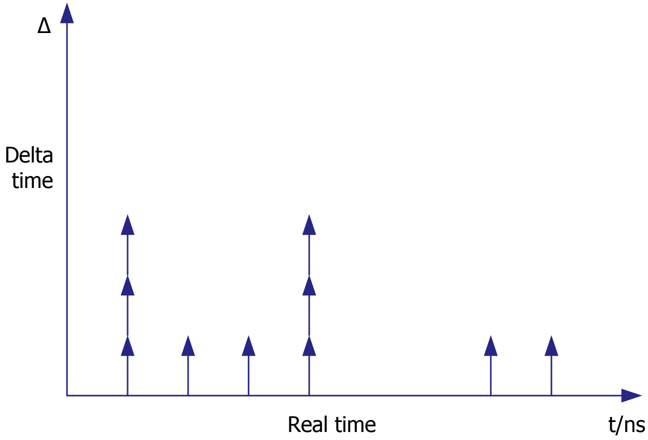
- All statements are processed sequential
- Sensitivity list (a, b, ci)
- Multiple variable assignments are allowed
- Order of statements is relevant
- Variables are updated immediately
- Signals are updated at the end of a process
- Try to avoid multiple signal assignments

Parallel versus Sequential Processing

```
p1: process (a, b)
begin  -- process p1
  -- sequential statements
  -- ...
  -- drive process outputs
  x <= ...
end process p1;

p2: process (c, x)
begin  -- process p2
  -- sequential statements
  -- ...
  -- drive process outputs
  y <= ...
end process p2;

-- drive module outputs
o <= x or y;
```

- process p1, process p2 and o are processed in parallel
- Statements within processes are sequential
- Inter-process communication with signals
- Signal values are evaluated recursively in zero time
- Simulator uses delta cycles

Real time and delta cycles



Signals and Variables

Signals

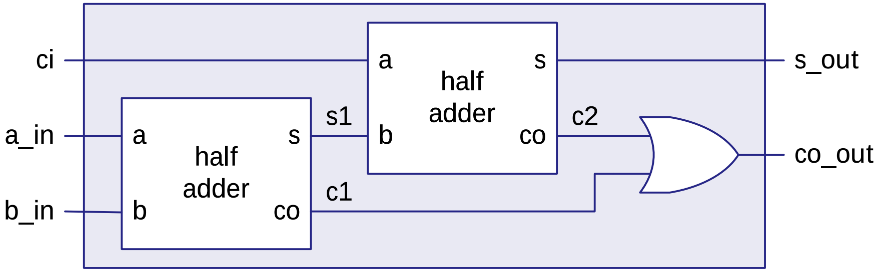
- Inside and outside processes
- Communication between parallel statements and processes
- Only the last assignment within a process is evaluated
- Signal is updated at the end of a process
- Signals are wires

Variables

- Inside processes only
- For intermediate results
- Multiple assignments allowed
- Immediate update

Structural Description

- Module composition
- Wiring of Modules
- Design Hierarchy
- "Divide and conquer"

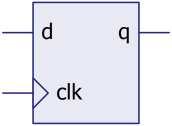


```
architecture structural of full_adder is
  component half_adder
    port (
      a, b  : in   std_logic ;
      s, co : out  std_logic );
  end component;
  signal s1, c1, c2 : std_logic ;
begin  -- structural
  half_adder_1 : half_adder
    port map (
      a  => a_in,  b  => b_in,
      s  => s1, co => c1);

  half_adder_2 : half_adder
    port map (
      a  => ci, b  => s1,
      s  => s_out,  co => c2);

  co_out <= c1 or c2;

end structural ;
```

- Make module half_adder known with component declaration
- Module instantiation
- Connect ports and signals

Register

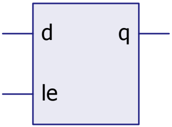
```
entity reg is
  port (
    d, clk : in  std_logic;
    q      : out std_logic );
end reg;

architecture rtl of reg is
begin -- rtl

  reg: process (clk)
  begin -- process reg
    if rising_edge(clk) then
      q <= d;
    end if;
  end process reg;

end rtl;
```



- Sensitivity list only contains the clock
- Assignment on the rising edge of the clock
- Not transparent

Latch

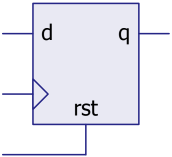
```
entity latch is
  port (
    d, le : in  std_logic;
    q     : out std_logic );
end latch;

architecture rtl of latch is
begin -- rtl

  latch: process (le, d)
  begin -- process latch
    if le = '1' then
      q <= d;
    end if;
  end process latch;

end rtl;
```



- Sensitivity list contains latch enable and data input
- Assignment during high phase of latch enable
- Transparent

Register with Asynchronous Reset

```
architecture rtl of reg_areset is
begin -- rtl

  reg : process (clk, rst)
  begin -- process reg
    if rising_edge(clk) then
      q <= d;
    end if;
    if rst = '0' then
      q <= '0';
    end if;
  end process reg;

end rtl;
```



- Sensitivity list contains clock and reset
- Reset statement is the last statement within the process
- Reset has highest priority

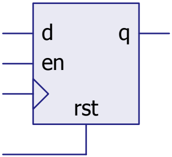
Register with Synchronous Reset

```
architecture rtl of reg_sreset is
begin -- rtl

  reg : process (clk)
  begin -- process reg
    if rising_edge(clk) then
      q <= d;
      if rst = '0' then
        q <= '0';
      end if;
    end if;
  end process reg;

end rtl;
```

- Sensitivity list only contains clock
- Reset statement is the last statement within the clock statement
- Should be used if target architecture supports synchronous resets

Register with Clock Enable

```
architecture rtl of reg_enable is
begin -- rtl

  reg : process (clk, rst)
  begin -- process reg
    if rising_edge(clk) then
      if en = '1' then
        q <= d;
      end if;
    end if;
    if rst = '0' then
      q <= '0';
    end if;
  end process reg;

end rtl;
```



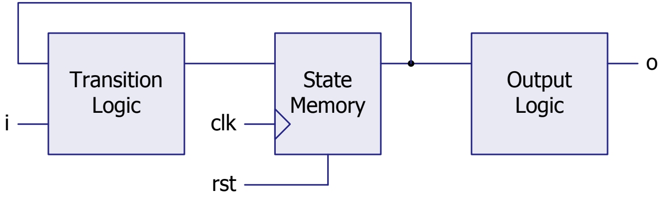
- Enable statement around signal assignment
- Use this semantic!

Storage Elements – Summary

- Before you start
  - Register or latch?
  - What kind of reset?
  - Clock enable?
- When you code, be precise
  - Sensitivity list
  - Clock statement
  - Enable semantic
  - Reset order/priority
- Prefer registers with synchronous resets
- Check synthesis results

Register Transfer Level (RTL)

- A Module may consist of
  - Pure combinational elements
  - Storage elements
  - Mixture of combinational and storage elements
- Use RTL for
  - shift registers, counters
  - Finite state machines (FSMs)
  - Complex Modules



Shift register

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

entity shifter is

  port (
    clk : in  std_logic;
    rst : in  std_logic;
    o   : out std_logic_vector (3 downto 0));

end shifter;

architecture beh of shifter is

  signal value : unsigned(3 downto 0);
```

- Use ieee.numeric_std for VHDL arithmetics
- Always std_logic for ports
- Internal signal for register

```
begin  -- beh

  shift : process (clk, rst)
  begin  -- process shift

    if rising_edge(clk) then
      value <= value rol 1;
    end if;
    if rst = '0' then
      value <= (others => '0');
    end if;

  end process shift;

  o <= std_logic_vector(value);

end beh;
```

- Clocked signal assignments are synthesized to registers
- Do not forget to drive the outputs

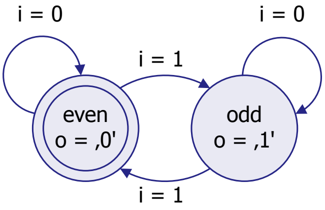
Counter

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

entity counter is

  port (
    clk, rst : in  std_logic;
    o        : out std_logic_vector (3 downto 0));

end counter;

architecture beh of counter is

  signal value : integer range 0 to 15;
```

```
begin -- beh

  count: process (clk, rst)
  begin -- process count

    if rising_edge(clk) then
      value <= value + 1;
    end if;
    if rst = '0' then
      value <= 0;
    end if;

  end process count;

  o <= std_logic_vector(to_unsigned(value, 4));

end beh;
```

Finite State Machine: OPC

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

entity opc is
  port (
    i        : in  std_logic;
    o        : out std_logic;
    clk, rst : in  std_logic);
end opc;

architecture rtl of opc is

  type state_type is (even, odd);
  signal state : state_type;
```



- Declare state types
- Signal for state memory

```
-- State memory and transition logic
trans : process (clk, rst)
begin -- process trans
  if rising_edge (clk) then
    case state is
      when even =>
        if i = '1' then state <= odd;
        end if;
      when odd =>
        if i = '1' then state <= even;
        end if;
    end case;
  end if;
  if rst = '0' then
    state <= even;
  end if;
end process trans;
```

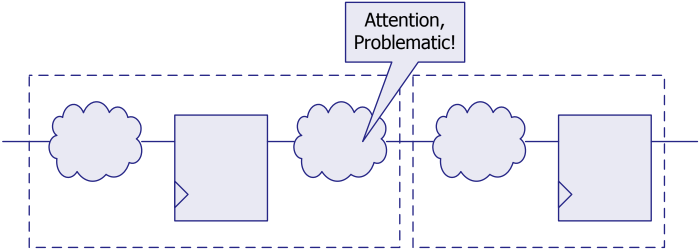
- Transition logic and memory in one process
- Always reset state machines!

```
-- Output logic
output : process (state)
begin  -- process output
  case state is
    when even => o <= '0';
    when odd => o <= '1';
  end case;
end process output;

end rtl;
```

- Output logic is placed in a second process
- Unregistered outputs are often problematic

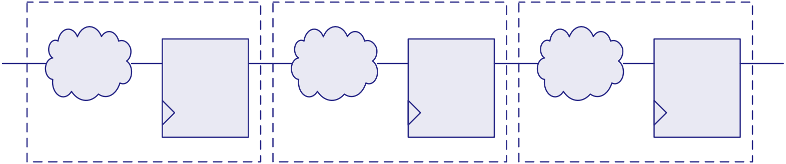
Unregistered Outputs

- Difficult timing analysis
- Combinational loops possible
- Undefined input delay for attached modules
- Glitches



Registered Outputs

- Prefer registered module outputs
- Simplifies the design process
- Prevents glitches and combinational loops



- Exception: Single-cycle handshake
  - Request registered
  - Acknowledge unregistered

Problems of Traditional RTL Design

Error-prone

- Many processes and parallel statements
- Many signals
- Accidental latches, multiple signal drivers

Inflexible Design Patterns

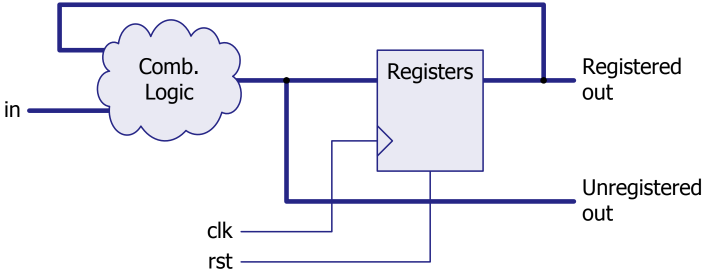
- Combinational logic
- Registers, FSMs

Schematic-VHDL

- Difficult to understand, to debug, and to maintain
- Focused on the schematic and not on the algorithm

Solution: Abstracting Digital Logic

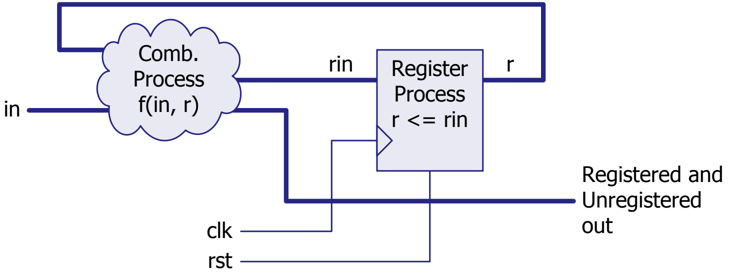
- Synchronous designs consist of
  - Combinational logic
  - Registers



- Forget about Moore and Mealy when designing large modules
- They are inflexible and they have unregistered outputs

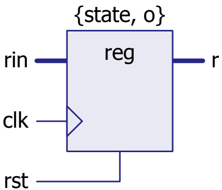
VHDL Realization

- A module only consists of two processes
  - Combinational process $rin = f(in, r)$
  - Clocked process $r = rin$
- Combinational process is sensitive to inputs and registers
- Sequential process is sensitive to clock and reset



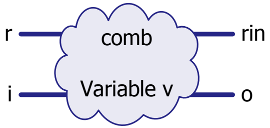
Two-Process Methodology

```
architecture rtl of edge_detect is
  type state_type is (low, high);
  type reg_type is
    record
      state : state_type;
      o     : std_logic;
    end record;
  signal r, rin : reg_type;
begin -- rtl
  reg : process (clk, rst)
  begin -- process reg
    if rising_edge(clk) then
      r <= rin;
    end if;
    if rst = '0' then
      r.state <= low;
    end if;
  end process reg;
```



- Record type for all registers
- Register input rin and output r
- Single process for all registers
- Simplifies adding new registers to the module

```
comb : process (r, i)
  variable v : reg_type;
begin  -- process comb
  -- Default assignment
  v := r;
  -- Output is mostly '0'
  v.o := '0';
  -- Finite state machine
  case r.state is
    when low => if i = '1' then
                  v.state := high;
                  v.o := '1';
                end if;
    when high => if i = '0' then
                   v.state := low;
                 end if;
  end case;
  -- Drive outputs
  rin <= v;
  o <= r.o;
end process comb;
end rtl;
```



- Single process for combinational logic
- Variable v to evaluate the next register state rin
- Default assignment avoids accidental latches
- Modify v with respect to r and inputs
- Assign v to rin and drive outputs

Advantages

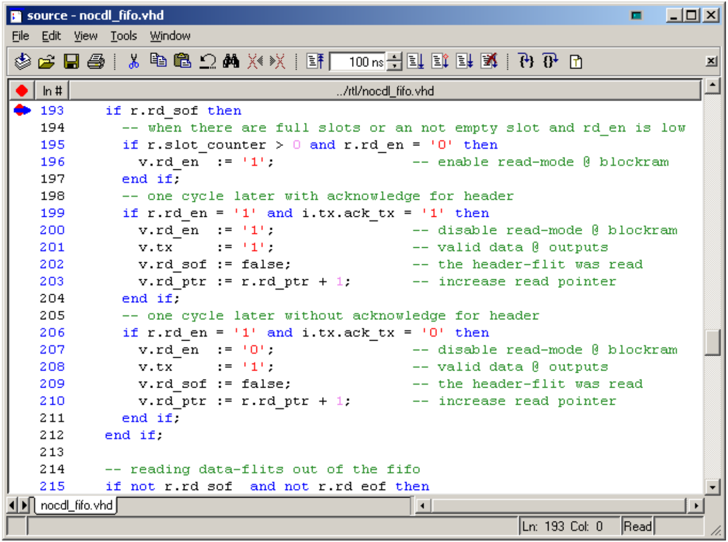
Consistent Coding Style

- Easy to write, to read, and to maintain
- Less error-prone
- High abstraction level
- Concentrates on the algorithm and not on the schematic
- Increases productivity

Excellent Tool Support

- Fast simulation
- Easy to debug
- Perfect synthesis results

Debugging


source - nocdl_fifo.vhd
File Edit View Tools Window
100 ns
ln #
../rtl/nocdl_fifo.vhd
if r.rd_sof then
-- when there are full slots or an not empty slot and rd_en is low
if r.slot_counter > 0 and r.rd_en = '0' then
v.rd_en := '1'; -- enable read-mode @ blockram
end if;
-- one cycle later with acknowledge for header
if r.rd_en = '1' and i.tx.ack_tx = '1' then
v.rd_en := '1'; -- disable read-mode @ blockram
v.tx := '1'; -- valid data @ outputs
v.rd_sof := false; -- the header-flit was read
v.rd_ptr := r.rd_ptr + 1; -- increase read pointer
end if;
-- one cycle later without acknowledge for header
if r.rd_en = '1' and i.tx.ack_tx = '0' then
v.rd_en := '0'; -- disable read-mode @ blockram
v.tx := '1'; -- valid data @ outputs
v.rd_sof := false; -- the header-flit was read
v.rd_ptr := r.rd_ptr + 1; -- increase read pointer
end if;
end if;
-- reading data-flits out of the fifo
if not r.rd_sof and not r.rd_eof then
nocdl_fifo.vhd
Ln: 193 Col: 0
Read

Tips

Keep the Naming Style

```
signal r, rin : reg_type;
variable v : reg_type;
```

Use Default Assignments

```
v.ack := '0';
if condition then v.ack := '1';
end if;
```

Use Variables for Intermediates

```
variable vinc;
vinc := r.value + 1;
```

Use Functions and Procedures

```
v.crc := crc_next(r.crc, data, CRC32);
seven_segment <= int2seg(value);
full_add(a, b, ci, s, co);
```

Variables for Unregistered Outs

```
variable v : reg_type
variable vaddr;
ack <= v.ack;
addr <= v.addr;
```

Design Strategies

Raise the Abstraction Level!

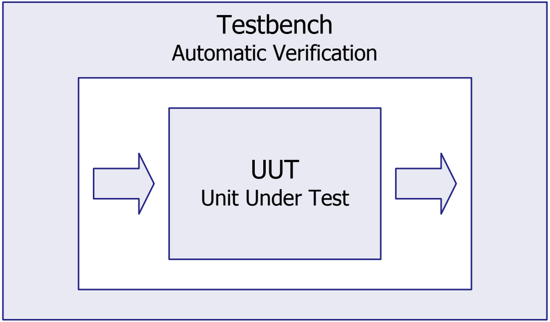
- Use the two-process methodology
- Use variables
- Use integers, booleans, signed, and unsigned
- Use functions and procedures
- Use synthesizable operators

Structural Design

- "Devide and Conquer"
- Do not overuse structural design
- But keep the modules testable

Outline

Testbench



- Testbed for unit under test (UUT), Not synthesizable
- Testbench instantiates UUT
- Generates inputs
- Checks outputs

A simple VHDL Testbench

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

entity half_adder_tb is
end half_adder_tb;

architecture behavior of half_adder_tb is

  component half_adder
    port (
      a, b  : in  std_logic;
      s, co : out std_logic);
  end component;

  signal a_sig, b_sig  : std_logic;
  signal s_sig, co_sig : std_logic;
```

- Entity is empty
- Declare UUT half_adder
- Declare signals to interface UUT

```
begin  -- behavior

  -- Unit Under Test
  UUT: half_adder
    port map (
      a  => a_sig,
      b  => b_sig,
      s  => s_sig,
      co => co_sig);
```

- Instantiate UUT
- Connect signals

```
  stimuli : process
  begin  -- process stimuli
    -- generate signals
    a_sig <= '0'; b_sig <= '0';
    wait for 10 ns;
    a_sig <= '1';
    wait for 10 ns;
    a_sig <= '0'; b_sig <= '1';
    wait for 10 ns;
    a_sig <= '1';
    wait for 10 ns;
    -- stop simulation
    wait;
  end process stimuli;

end behavior;
```

- Process to generate test pattern
- No sensitivity list
- Execution until wait statement
- Without wait: Cyclic execution

Automatic Verification and Bus-functional Procedures

```
library  ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.syslog.all;

entity adder_tb is
  generic (
    -- Adder delay
    period : time := 10 ns);
end adder_tb;

architecture behavior of adder_tb is

  component adder
    port (
      a, b : in   std_logic_vector (3 downto 0);
      sum  : out  std_logic_vector (3 downto 0);
      co   : out  std_logic );
  end component;
```

- Syslog package to generate messages
- Parametrized testbench
- UUT declaration

```
  signal  a_sig , b_sig : std_logic_vector (3 downto 0)
    := (others => '0');
  signal  sum_sig : std_logic_vector (3 downto 0);
  signal  co_sig   : std_logic ;

begin  -- behavior
  -- Unit Under Test
  UUT : adder
    port map (
      a   => a_sig,
      b   => b_sig,
      sum => sum_sig,
      co  => co_sig);
```

- Declare UUT signals
- Initialize inputs
- Instantiate UUT
- Connect signals

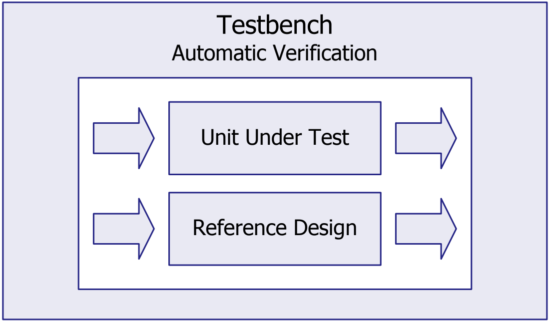
```
-- Stimuli and Verification
tester : process

  -- Bus-functional procedure for UUT
  procedure add (m, n : in integer range 0 to 15; s : out integer) is
  begin -- do_operation
    -- set UUT operand inputs
    a_sig <= std_logic_vector(to_unsigned(m, a_sig'length));
    b_sig <= std_logic_vector(to_unsigned(n, b_sig'length));
    -- wait some time
    wait for period;
    -- get UUT result
    s := to_integer(unsigned(co_sig & sum_sig));
  end add;
```

- Bus-functional procedure abstracts UUT interface

```
begin

    syslog_testcase ("Test␣all␣input␣combinations");
    for i in 0 to 15 loop
        syslog(debug, "Operand␣a␣=␣" & image(i) & "␣...");
        for j in 0 to 15 loop
            add(i, j, s);
            if s /= i + j then
                syslog(error, "Bad␣result␣for␣" &
                       image(i) & "␣+␣" & image(j) & "␣=␣" & image(i + j) &
                       ",␣obtained:␣" & image(s));
            end if;
        end loop;
    end loop;

    syslog_terminate;
end process tester;

end behavior;
```

- Automatic verification, Debug messages, Termination

Testing with Reference Models

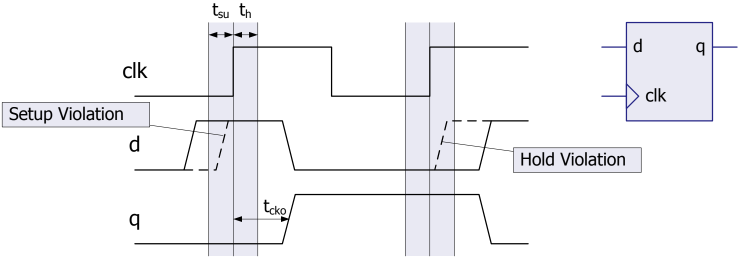


- Instantiate UUT and reference model
- Same test pattern
- Compare results

Backannotation and Timing Simulation

- Synthesize and place & route UUT
- Backannotate: Extract netlist and timing
- Simulation with testbench
- Use the same testbench as for functional simulation
- Testbench has to consider real delays
  - Setup and hold time of registers
  - Pads and wires
- Never set inputs on clock edge

Setup and Hold Time



- The setup time t_{su} defines the time a signal must be stable before the active clock edge
- The hold time t_h defines the time a signal must be stable after the active clock edge
- The clock-to-out time defines the output delay of the register after the active clock edge
- When setup or hold violation occurs the output is undefined

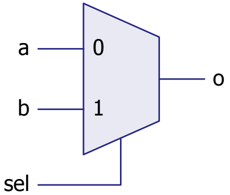
Hardware Testbench

- A synthesizable testbench allows to download it into an FPGA
- Examples:
  - Testbench with reference model
  - Build-in self test (BIST)
  - Microprocessor for test pattern generation
- Testbench has ports such as clock, reset and test results
- Very fast
- Considers real wire and logic delays
- Most accurate

Outline

If statement

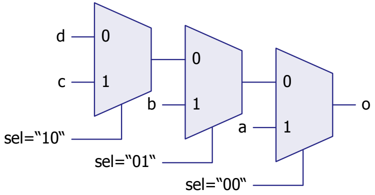
```
good_if: process (a, b, sel)
begin -- process good_if
  if sel = '1' then
    o <= a;
  else
    o <= b;
  end if;
end process good_if;
```



- Implements a multiplexer
- This description is optimal

Cascaded If

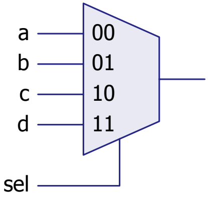
```
bad_if : process (a, b, c, d, sel)
begin  -- process bad_if
  if sel = "00" then
    o <= a;
  elsif sel = "01" then
    o <= b;
  elsif sel = "10" then
    o <= c;
  else
    o <= d;
  end if;
end process bad_if;
```



- Cascaded if statements
- Results in a cascaded multiplexer
- Long delay

Case statement

```
good_case: process (a, b, c, d, sel)
begin  -- process bad_if
  case sel is
    when "00" => o <= a;
    when "01" => o <= b;
    when "10" => o <= c;
    when others => o <= d;
  end case;
end process good_case;
```

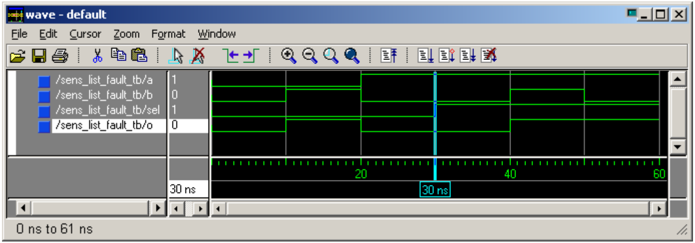


- Use case statement for multiplexers
- Best synthesis results

Missing sensitivities

```
mux: process (a, b)
begin  -- process mux
  if sel = '1' then
    o <= a;
  else
    o <= b;
  end if;
end process mux;
```

- Signal sel is missing in sensitivity list
- Process is only activated on a or b
- Simulation error
- Synthesis often correct
- Watch out for synthesis warnings



Accidental Latches with If Statements

```
 if_latch : process (a, b, sel)
begin  -- process if_latch
  if sel = "01" then
    o <= a;
  elsif sel = "10" then
    o <= b;
  end if;
end process if_latch ;
```

- Only cases “01” and “10” are covered, other cases missing
- During missing cases o must be stored
- Accidental latch is inferred by synthesis tool
- sel = “01” or sel = “10” acts as latch enable

Accidental Latches with Case Statements

```
case_latch : process (a, b, c, d, sel)
begin  -- process case_latch
  case sel is
    when "00" => o1 <= a;
                 o2 <= b;
    when "01" => o1 <= b;
    when "10" => o1 <= c;
                 o2 <= a;
    when others => o1 <= d;
                   o2 <= c;
  end case;
end process case_latch;
```

- Assignment for o2 is missing in case sel = "01"
- Accidental latch to store o2

Circumvent Accidental Latches

```
case_default : process (a, b, c, d, sel)
begin -- process case_default
  o1 <= a; o2 <= c;
  case sel is
    when "00" => o2 <= b;
    when "01" => o1 <= b;
    when "10" => o1 <= c;
                 o2 <= a;
    when others => o1 <= d;
  end case;
end process case_default;
```

- Use default assignments
- Missing cases are intentional

Incorrect use of Signals and Variables

```
architecture beh_seq of full_adder is
begin -- beh_seq

  add: process (a, b, ci)
    variable s_tmp, c_tmp : std_logic;
  begin -- process add
    -- half adder 1
    s_tmp := a xor b;
    c_tmp := a and b;
    -- half adder 2
    c_tmp := c_tmp or (s_tmp and ci);
    s_tmp := s_tmp xor ci;
    -- drive signals
    s <= s_tmp;
    co <= c_tmp;
  end process add;

end beh_seq;
```

- This description is correct
- Do not declare s_tmp and c_tmp as signals
- If signals
  - Only last assignment is significant
  - Half adder 1 is removed
  - Combinational loops
- Use signals for wires
- Use varibles for combinational intermediate results

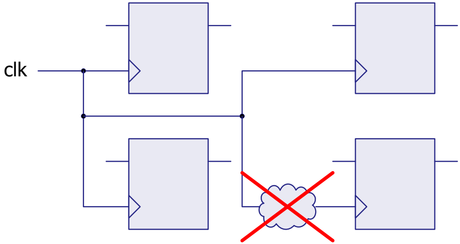
The Package ieee.numeric_std

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

- Use the package numeric_std instead of obsolete std_logic_signed and std_logic_unsigned
- Avoids ambiguous expressions
- Strict destinction between signed and unsigned vectors
- Sometimes a bit cumbersome but exact

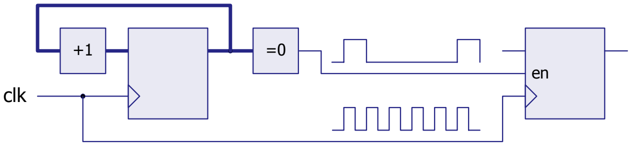
Clocking

- One phase, One clock!
- No clock gating
- Use rising_edge(clk)
- Avoid latches, Check synthesis results

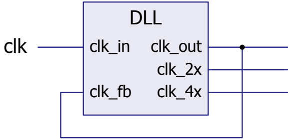


Clock scaling

- Scale clock with synchronous counters and enables

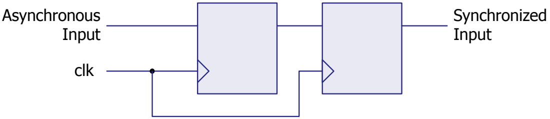


- Use DLLs and PLLs to create other clocks

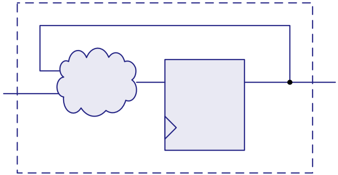


Asynchronous Signals

- Synchronize asynchronous signals with at least two Registers

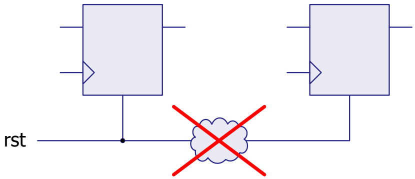


- Prefer registered module outputs – on-chip and off-chip

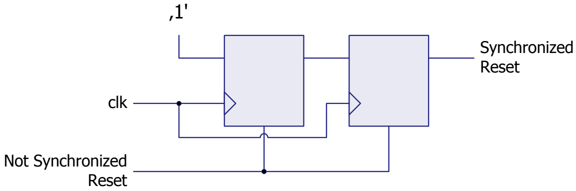


Resetting

- Do not touch reset without knowledge
- This may cause problems



- Synchronize reset to clock



Other Mistakes

- Use downto for all logic vectors

```
std_logic_vector (3 downto 0);
```

- Constrain integers

```
integer range 0 to 7;
```

- Be careful in testbenches with setup and hold time
- Implement signal assignments close to reality

Not Synthesizable VHDL

- Initializing signals

```
signal q : std logic := '0';
```

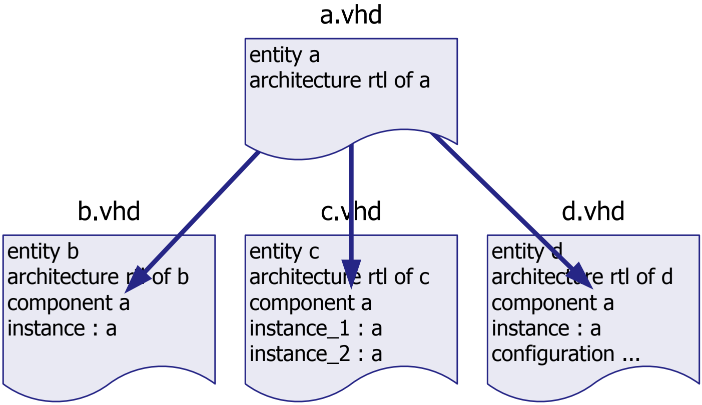
- Run-time loops
- Timing specification

```
wait for 10 ns; r <= '1' after 1 ns;
```

- Text I/O
- Floating point

Outline

Design Units



- Entities, architectures, components, instances
- Only one module (entity, architecture) in a single file
- Multiple component declarations are redundant

Packages

```
package package_name is
  -- Declaration of
  --  Types and Subtypes
  --  Constants, Aliases
  --  Signals, Files, Attributes
  --  Functions, Procedures
  --  Components
  -- Definition of
  --  Constants, Attributes
end package_name;

package body package_name is
  -- Definition of earlier
  -- declared Objects
  --  Funktions, Procedures
  --  Constants
  -- Decaration/Definition
  -- of additional Objects
end package_name;
```

- Place global objects and parameters in packages
- Solves redundancies
- Package holds declarations
- Package body holds definitions
- Include the package with

```
use work.package_name.all;
```

Libraries

- A library contains design units
  - Entities, architectures
  - Packages and package bodies
  - Configurations
- Mapped to a physical path
  - work ⇒ ./work
  - ieee ⇒ $MODEL_TECH/../ieee
- Tools place all design units in library work by default
- Libraries work and std are visible by default
- Include other libraries with

```
library library_name;
```

Flexible Interfaces

- Problem of large designs
  - Entities have many ports, confusing, difficult naming
  - Redundant interface description (entity, component, instance)
  - Modifying an interface is cumbersome and error-prone
- Solution: Define complex signal records
- Record aggregates wires of an interface

```
type clk_type is
  record
    clk     : std_logic ;
    clkn    : std_logic ;
    clk_2x  : std_logic ;
    clk_90  : std_logic ;
    clk_270 : std_logic ;
  end record;
```

```
-- Create and buffer clocks
entity clk_gen is
  port (
    clk_pad : in  std_logic ;
    clk     : out clk_type );
end clk_gen;
```

Flexible Interfaces – In and Out

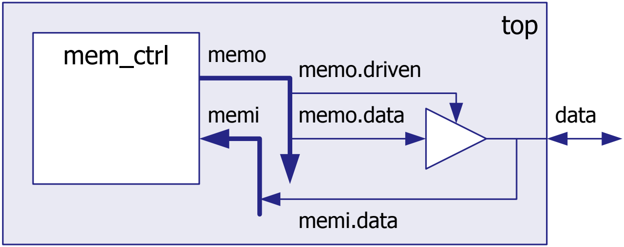
- Split interface in input and output records
- Declare name_in_type and name_out_type

```
type mem_in_type is
  record
    data  :  mem_data_type;
  end record;

type mem_out_type is
  record
    address      :  mem_address_type;
    data         :  mem_data_type;
    drive_datan  :  std_logic ;
    csn          :  std_logic ;
    oen          :  std_logic ;
    writen       :  std_logic ;
  end record;
```

```
-- Memory controller
entity mem_ctrl is
  port (
    clk    : in   clk_type ;
    reset  : in   reset_type ;
    memo   : out  mem_out_type;
    memi   : in   mem_in_type
    -- Other signals
    -- Control and data wires
    );
end mem_ctrl;
```

```
entity top is
  port (
    data    : inout mem_data_type;
    address : out   mem_address_type;
    csn     : out   std_logic ;
    oen     : out   std_logic ;
    writen  : out   std_logic
    -- other signals
    );
end top;
```

- Pads of a chip are often bidirectional
- Infer tristate buffers in top entity
- Do not use inout records
- Strictly avoid bidirectional on-chip wires/buses



Flexible Interfaces – Inout Implementation

Memory Controller

```
-- instantiate memory controller
mem_ctrl_i: mem_ctrl
  port map (
    clk   => clk,
    reset => reset,
    memo  => memo,
    memi  => memi);

-- drive data out
data <= memo.data when memo.driven = '0' else (others => 'Z');
-- read data in
memi.data <= data;
```

- Use the 'Z' value of std_logic to describe tri-state buffers

Design Strategies

- Aggregate signals belonging to a logical interface
  - System bus
  - Control and status
  - Receive
  - Transmit
- Use hierarchical records to aggregate multiple interfaces
- Do not place clock and reset into records
- Do not use records for top entity

Advantages

- Reduces the number of signals
- Simplifies adding and removing of interface signals
- Simplifies route-through of interfaces
- Raises the abstraction level, Improves maintainability

Customizable VHDL Designs

What's this?

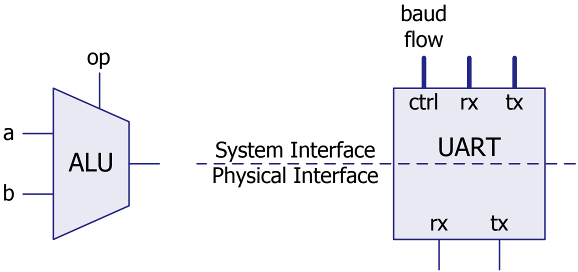
- VHDL Design is made configurable by parameters
- Modifies its structure, functionality, or behavior
- Facilitates design reuse
- Flexibility

VHDL gives you

- Entity Signals
- Entity Generics
- Package Constants

Control Signals

- Entity contains additional control signals
- Locally to module
- Customization at runtime
- Consumes additional hardware



Generics

- Entity contains parameters, Locally to module
- Customization at design (compile) time
- Consumes no additional hardware
- Unused logic will be removed

```
entity adder is
  generic (
    n : integer := 4);
  port (
    a, b : in   std_logic_vector (n-1 downto 0);
    ci   : in   std_logic ;
    s    : out  std_logic_vector (n-1 downto 0);
    co   : out  std_logic );
end adder;
```

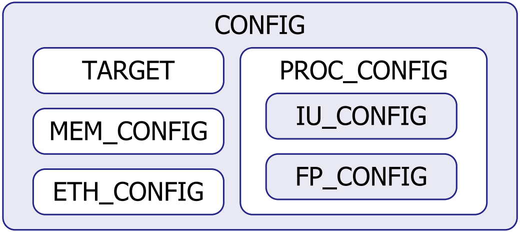
Constants

- Define constants in a package
- Visible for all design units using the package
- Global parameters
- Be careful with naming, Use upper case

```
constant MEM_ADDRESS_WIDTH : integer := 16;
constant MEM_DATA_WIDTH : integer := 8;

subtype  mem_address_type is
   std_logic_vector (MEM_ADDRESS_WIDTH-1 downto 0);
subtype  mem_data_type is
   std_logic_vector (MEM_DATA_WIDTH-1 downto 0);
```

Hierarchical Customization

- Reducing the number of global constants
- Structuring the parameters in configuration records
- Aggregate configurations hierarchically
- Simple and flexible selection of configurations



```
type mem_config_type is
  record
    address_width, data_width, banks     : integer;
     read_wait_states ,  write_wait_states  : integer;
  end record;

type config_type  is
  record
    mem   : mem_config_type; target  :  target_config_type ;
  end record;

constant MEM_SMALL_FAST : mem_config_type := (
  address_width  => 16, data_width => 8, banks => 1,
   read_wait_states  => 0,  write_wait_states  => 1);

constant MEM_LARGE_SLOW : mem_config_type := (
  address_width  => 20, data_width => 32, banks => 4,
   read_wait_states  => 4,  write_wait_states  => 4);

constant MY_SYSTEM : config_type := (
  mem => MEM_LARGE_SLOW, target => VIRTEX);
```

Generate Statement

Generic Adder

```
begin  -- struct

  c(0) <= ci; co <= c(n);

  fa_for : for i in 0 to n-1 generate
    fa_1 : full_adder
      port map (
        a  => a(i),
        b  => b(i),
        ci => c(i),
        s  => s(i),
        co => c(i+1));
  end generate fa_for;

end struct;
```

- Use generate for structural composition
- Applicable for components, processes and parallel statements
- "For"-generate
- "If"-generate but no "else"

Loops Instead of Generate

Generic Adder

```
add : process (a, b, ci)
   variable sv : std_logic_vector (n-1 downto 0);
   variable cv : std_logic_vector (n downto 0);
begin  -- process add

  cv(0) := ci;

  for i in 0 to n-1 loop
    full_add (a(i), b(i), cv(i), sv(i), cv(i+1));
  end loop;  -- i

  s <= sv;
  co <= cv(n);

end process add;
```

- Use loops for generic designs
- Inside processes
- Much simpler than generate statement
- Loop range must be known at compile time

If-Then-Else Instead of Generate

```
if CALIBRATE then
  -- remove this logic when
  -- CALIBRATE = false
  v.period_counter := r.period_counter + 1;
  if rising then
    v.period := v.period_counter;
    v.period_counter := 0;
  end if;
else
  -- remove this logic when
  -- CALIBRATE = true
  v.period := CLK_RATE / FREQUENCY;
end if;
```

- If-Then-Else is much simpler than generate
- Inside processes
- Unused logic is removed
- CALIBRATE must be known at compile time