A Structured VHDL Design Method

EEMCS, course ET 4351

April 24, 2014
A Structured VHDL Design Method

- Jiri Gaisler, CTH / Gaisler Research, Sweden
- ... also promoted by the ESA (European Space Agency)
- Hugo Hedberg, Lund Institute of Technology, Sweden
- Ian Lang, Design Abstractions Ltd, United Kingdom
- Mike Treseler
Outline of lecture

- Traditional 'ad-hoc' VHDL design style
- Proposed structured design method
- Various ways of increasing the abstraction level in synthesizable code
- A few design examples
Traditional VHDL design methodology

- Based on heritage from schematic entry (!):
  - many small processes or concurrent statements
  - usage of TTL-like macro blocks
  - usage of GUI tools for code generation

- Could be compared to schematics without wires (!)
- Hard to read due to many concurrent statements
- Auto-generated code even harder to read/maintain

Hard to read = difficult to maintain
Traditional ad-hoc design style

- Many concurrent statements
- Many signals
- Few and small process statements
- No unified signal naming convention
- Coding is done at low RTL level:
  - assignments with logical expressions
  - only simple array data structures
Unified signal naming convention

- **Ports**
  - **Inputs:** <inport_name>\_in
  - **Outputs:** <outport_name>\_out
- **Signals:** <signal_name>\_s
- **Variables:** <variable_name>\_v
- **Types:** <type_name>\_type
- **Packages:** <package_name>\_pkg
- **Process:** p\_<process_label>
- **Instance:** i\_<instance_label>
Simple VHDL example

CbandDatat_LatchPROC9F: process(MDLE, CB_In, Reset_Out_N)
begin
    if Reset_Out_N = '0' then
        CBLatch_F_1 <= "0000";
    elsif MDLE = '1' then
        CBLatch_F_1 <= CB_In(3 downto 0);
    end if;
end process;

CBandDatat_LatchPROC10F: process(MDLE, CB_In, DParIO_In, Reset_Out_N)
begin
    if Reset_Out_N = '0' then
        CBLatch_F_2 <= "0000";
    elsif MDLE = '1' then
        CBLatch_F_2(2 downto 0) <= CB_In(6 downto 4);
        CBLatch_F_2(3) <= DParIO_In;
    end if;
end process;

CBLatch_F <= CBLatch_F_2 & CBLatch_F_1;
Problems

► Dataflow coding difficult to understand
► Algorithm difficult to understand
► No distinction between sequential and combinational signals
► Difficult to identify related signals
► Large port declarations in entity headers
► Slow execution of simulations due to many signals and processes

► The ad-hoc style does not scale
Ideal model characteristics

- We want our models:
  - to be easy to understand and to maintain
  - to be synthesizable
  - to simulate as fast as possible
  - to show no simulation/synthesis discrepancies
  - to be usable for small and large designs

New design style/method needed!
1: abstraction of digital logic

- A synchronous design can be abstracted into two separate parts: a combinational and a sequential part.

```
d -> combinational
    q = f(d, q_r)

DFF
    q_r
    q

Clk
```
2: the abstracted view in VHDL: the two-process scheme

- A VHDL entity is made to contain only two processes: one sequential and one combinational
- Inputs are denoted d, outputs q
- Two local signals are declared: register-in \((rin)\) and register-out \((r)\)
- The full algorithm \((q = f(d, r))\) is performed in the combinational process
- The combinational process is sensitive to all input ports and the register outputs \(r\)
- The sequential process is only sensitive to the clock
Two-process VHDL entity

In-ports

combinational process

q = f_1(d, r)
ri = f_2(d, r)

Out-port

Q

sequential process

ri

r

Clk
Two-process scheme: data types

- The local signals $r$ and $rin$ are of composite type (record) and include all registered values.
- All outputs are grouped into one entity-specific record type, declared in a global interface package.
- Input ports can be of output record types from other entities.
- All registers declared in a record type.
- A local variable of the register record type is declared in the combinational processes to hold newly calculated values.
- Additional variables of any type can be declared in the combinational process to hold temporary values.
Example

use work.interface.all;

entity irqctrl is port (    clk : in std_logic;
  rst : in std_logic;
  sysif : in sysif_type;
  irqo : out irqctrl_type);
end;

architecture rtl of irqctrl is

  type reg_type is record
    irq : std_logic;
    pend : std_logic_vector(0 to 7);
    mask : std_logic_vector(0 to 7);
  end record;

  signal r, rin : reg_type;

  begin
    comb : process (sysif, r)
      variable v : reg_type;
      begin
        v := r; v.irq := '0';
        for i in r.pend'range loop
          v.pend(i) := r.pend(i) or
            (sysif.irq(i) and r.mask(i));
          v.irq := v.irq or r.pend(i);
        end loop;
        rin <= v;
        irqo.irq <= r.irq;
      end process;

    reg : process (clk)
      begin
        if rising_edge(clk) then
          r <= rin;
        end if;
      end process;

  end architecture;
Hierarchical design

Grouping of signals makes code readable and shows the direction of the dataflow.

```
use work.interface.all;

entity cpu is port (  
  clk    : in  std_logic;  
  rst    : in  std_logic;  
  mem_in : in  mem_out_type;  
  mem_out: out mem_in_type);  
end;

architecture rtl of cpu is  
  signal cache_out : cache_type;  
  signal proc_out  : proc_type;  
  signal mctrl_out : mctrl_type;  
begin  
  u0 : proc port map  
    (clk, rst, cache_out, proc_out);  
  u1 : cache port map  
    (clk,rst,proc_out,mctrl_out,cache_out);  
  u2 : mctrl port map  
    (clk, rst, cache_out, mem_out,  
      mctrl_out, mem_in);  
end architecture;
```
Benefits

- Sequential coding is well known and understood
- Algorithm is easily extractable
- Uniform coding style simplifies maintenance
- Improved simulation and synthesis speed
- Development of models is less error prone
Adding a port

Traditional method:
- Add port in entity port declaration
- Add port in sensitivity list of appropriate processes (input ports only)
- Add port in component declaration
- Add signal declaration in parent module(s)
- Add port map in component instantiation in parent module(s)

Two-process method:
- Add element in the interface record
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity state_machine is
  generic (m : integer := 2) -- Used to process bus width
  port (clk : in STD_LOGIC;
  reset : in STD_LOGIC;
  input : in STD_LOGIC_VECTOR(m-1 downto 0);
  newinp : in STD_LOGIC_VECTOR(m-1 downto 0);
  output : out STD_LOGIC_VECTOR(m-1 downto 0);
end state_machine;

architecture implementation of state_machine is
begin
  combinatorial : process (input,newinp,state,next_state) -- Sensitivity list
  begin
    ...
  end process;

  synchronous : process (clk,reset)
  begin
    ...
  end process;
end architecture;

component state_machine
  -- Component declaration
  generic (m : integer := 2) -- Used to process bus width
  port (clk : in STD_LOGIC;
  reset : in STD_LOGIC;
  input : in STD_LOGIC_VECTOR(m-1 downto 0);
  newinp : in STD_LOGIC_VECTOR(m-1 downto 0);
  output : out STD_LOGIC_VECTOR(m-1 downto 0);
end component;

Stm : state_machine
  -- Component instantiation
  generic map ( m => 2 )
  port map ( clk => clk,
    reset => reset,
    input => input,
    newinp => newinp,
    output => output );

state_machine.vhd design.vhd
Structured VHDL programming

library IEEE; use IEEE.STD_LOGIC_1164.all;

package global_interface_pkg is
    input_type is record
        newsignal : std_logic;
    end record;
end package;

library IEEE; use IEEE.STD_LOGIC_1164.all;
Use work.global_interface_pkg.all;

entity state_machine is
    port (clk : in STD_LOGIC;
         reset : in STD_LOGIC;
         input : in input_type;
         output : out output_type;
    end state_machine;

state_machine

input_type

output_type
Adding a register

Traditional method:
- Add signal declaration (2 signals)
- Add registered signal in process sensitivity list (if not implicit)
- (Declare local variable)
- Add driving statement in clocked process

Two-process method:
- Add definition in register record
library IEEE; use IEEE STD_LOGIC_1164.all;
entity state_machine is
  generic (m: integer := 2) -- Used to process bus width
  port (clk: in STD_LOGIC;
    reset: in STD_LOGIC;
    input: in STD_LOGIC_VECTOR(m-1 downto 0);
    output: out STD_LOGIC_VECTOR(m-1 downto 0);
  )
end state_machine;
architecture implementation of state_machine is
  type state_type is (st0, st1, st2, st3); -- defines the states;
  signal state, next_state, state_type;
  signal output, next_output
begin
  STD_LOGIC_VECTOR (m-1 downto 0);
begin
  synchronous : process (clk,reset)
  begin
    if clk'event and clk = '1' then
      if reset = '1' then
        state <= st0;
        output <= "00";
      else
        state <= next_state;
        output <= next_output; -- registered outputs
      end if;
    end if;
  end process;
end architecture;
combinatorial : process (input, state, next_state) -- Combinatorial part
begin
  next_state <= state;
  next_output <= output;
  case (state) is -- Current state and input dependent
  when st0 => if (input = '1') then
    next_state <= st1;
    next_output <= "01"
  end if;
  when st3 => if (input = '1') then
    next_state <= st0;
    next_output <= "00"
  end if;
  when others => next_state <= next_state; -- Default
    next_output <= "00";
end case;
end process;

---

**Old school**

adding stored signal

---

TU Delft
Gaisler’s method
The state machine using Gaisler’s method

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work \textit{global} \texttt{pkg};

entity state_machine is
  generic (m : integer := 2) -- Used to process bus width
  port (clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        input : in \texttt{INPUT\_TYPE};
        output : out \texttt{OUTPUT\_TYPE};
  end state_machine;

architecture implementation of state_machine is
  type state_type is (st0, st1, st2, st3); -- defines the states
  type reg_type is record
    output : STD_LOGIC_VECTOR (m-1 downto 0);
    state : state_type;
  end record;
  Signal r, rin : reg_type;

begin
  synchronous process (\texttt{clk}) -- This part is always the same
  begin
    if \texttt{clk} event and \texttt{clk} = '1' then
      r <= rin;
    end if;
  end process;

  \texttt{combinatorial} : process (input, reset, r) -- Combinatorial part
  variable \textit{v} : reg_type;
  \texttt{begin}
    \textit{v} := r; -- Setting the variable
    \texttt{case} \textit{r\_state} is -- Current state and input dependent
    when st0 => if (input = '1') then
                  \textit{v\_state} := st1;
                  \textit{r\_output} := "01"
                end if;
    when st1 => if (input = '0') then
                  \textit{v\_state} := st2;
                  \textit{r\_output} := "11"
                end if;
    when st2 => if (input = '1') then
                  \textit{v\_state} := st3;
                  \textit{r\_output} := "10"
                end if;
    when st3 => if (input = '1') then
                  \textit{v\_state} := st0;
                  \textit{r\_output} := "00"
                end if;
    when others => \textit{v\_state} <= \textit{st0}; -- Default
                  \textit{v\_output} <= "00";
    end case;
    if (reset = '1') then -- Synchronous reset
      \textit{v\_state} := \textit{st0}; -- Start in idle state
    end if;
    \texttt{rin} <= \textit{v}; -- Updating the registers
    \texttt{output} <= \texttt{\textit{v\_output}}; -- Combinational output
    -- output <= \texttt{r\_output}; Registered output
  end process;
library IEEE; use IEEE.STD_LOGIC_1164.all;
Use work.global_interface.pkg;

entity state_machine is
    port (clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          input : in input_type;
          output : out output_type;
    end state_machine;

architecture implementation of state_machine is
    type state_type is (st0, st1, st2, st3); -- defines the states

    type reg_type is record
        output : output_type;
        state : state_type;
        newsignal : std_logic;
    end record;

    Signal r, rin : reg_type;
    begin
        synchronous : process (clk)-- This part is always the same
        begin
            if clk'event and clk = '1' then
                r <= rin;
            end if;
        end process;
    end architecture;

    combinatorial : process (input, reset, r) -- Combinatorial part
    variable v : reg_type;
    begin
        v := r; -- Setting the variable
        case (r.state) is -- Current state and input dependent
            when st0 => if (input = '1') then
                v.state := st1;
                v.output := "01"
            end if;
            when st3 => if (input = '1') then
                v.state := st0;
                v.output := "00"
            end if;
            when others => v.state <= st0; -- Default
                v.output <= "00";
        end case;

        if (reset = '1') then -- Synchronous reset
            v.state := st0; -- Start in idle state
        end if;

        rin <= v; -- Updating the registers
        -- output <= v.output; -- Combinational output
        output <= r.output; Registered output
    end process;

    Input d  -> Behavioural Logic
              ^ Output q
    State r  |   \\
    ^ Clock
    |   \
    V next state
    (D) Sequential part

TU Delft
Tracing signals during debugging

- Traditional method:
  - Figure out which signals are registered, which are their inputs, and how they are functionally related
  - Add signals to trace file
  - Repeat every time a port or register is added/deleted

- Two-process method:
  - Add definition in register record
  - Add interface records, r and rin
  - Signals are grouped according to function and easy to understand
  - Addition/deletion of record elements automatically propagates to trace window
variable vpdiff1_v : std_logic_vector(15 downto 0);
variable vpdiff2_v : std_logic_vector(15 downto 0);
variable prevVal17_v : std_logic_vector(16 downto 0);
variable vpdiffExt_v : std_logic_vector(16 downto 0);
variable predValTmp_v : std_logic_vector(16 downto 0);
variable newIndexTmp_v : std_logic_vector(7 downto 0);

begin
  regs_v := regs0_s; -- default assignment
  -- 'divide' and clamp
  step0_v := regs_v.stepSize;
  step1_v := "0" & step0_v(15 downto 1);
  step2_v := "00" & step0_v(15 downto 2);
  vpdiff0_v := "000" & step0_v(15 downto 3);
  if (adpcm4_i(2) = '1') then
    vpdiff1_v := (vpdiff0_v + step0_v);
  else
    vpdiff1_v := vpdiff0_v;
  end if;
  if (adpcm4_i(1) = '1') then
    vpdiff2_v := (vpdiff1_v + step1_v);
  else
    vpdiff2_v := vpdiff1_v;
  end if;
  if (adpcm4_i(0) = '1') then
    vpdiff_v := (vpdiff2_v + step2_v);
  else
    vpdiff_v := vpdiff2_v;
  end if;
Stepping through code during debugging

Traditional method:
- Connected processes do not execute sequentially due to delta signal delay
- A breakpoint in every connected process needed
- New signal value in concurrent processes not visible

Two-process method:
- Add a breakpoint in the beginning of the combinational process
- Single-step through code to execute complete algorithm
- Next signal value \( r_i \) directly visible in variable \( v \)
Complete algorithm can be a sub-program

- Allows re-use if placed in a global package
- Can be verified quickly with local test-bench
- Meiko FPU (20k gates):
  - 1 entity, 2 processes
  - 44 sub-programs
  - 13 signal assignments
  - Reverse-engineered from verilog: 87 entities, ~800 processes, ~2500 signals

source: Jiri Gaisler, CTH / Gaisler Research

```vhdl
comb : process (sysif, r, rst)
  variable v : reg_type;
begin
  proc_irqctl(sysif, r, v);
  rin <= v;
  irqo.irq <= r.irq;
end process;
```
Sequential code and synthesis

- Most sequential statements directly synthesizable by modern tools
- All variables have to be assigned to avoid latches
- Order of code matters!
- Avoid recursion, division, access types, text/file IO.

```vhdl
comb : process (sysif, r, rst)
    variable v : reg_type;
begin
    proc_irqctl(sysif, r, v);
    if rst = '1' then
        v.irq := '0';
        v.pend := (others => '0');
    end if;
    rin <= v;
    irqo.irq <= r.irq;
end process;
```
Comparison MEC/LEON

source: Jiri Gaisler, CTH / Gaisler Research

- ERC32 memory controller MEC
- Ad-hoc method (15 designers)
  - 25,000 lines of code
  - 45 entities, 800 processes
  - 2000 signals
  - 3000 signal assignments
  - 30k gates, 10 man-years, numerous bugs, 3 iterations

- LEON SPARC-V8 processor
- Two-process method (mostly)
  - 15,000 lines of code
  - 37 entities, 75 processes
  - 300 signals
  - 800 signal assignments
  - 100k gates, 2 man-years, no bugs in first silicon
Single process template program

entity sync_template is
  generic (vec_len : positive := 8);
  port (clock, reset, a : in std_ulogic;
        q : out std_logic_vector(vec_len-1 downto 0) );
end entity sync_template;

architecture synth of sync_template is       -- no signals required
begin
  sync_template : process(reset, clock) is
    -- Process declarations for the Template Procedures
    -- Template Procedures: Always same three names. Contents varies.
    procedure init_regs is ...  -- init of register variables only
    procedure update_regs is ... -- distilled functional description
    procedure update_ports is ... -- wire register variables out to port
    -- Top Process Template -- Always exactly the same:
    begin     -- process template
      if reset = '1' then  -- Assumes synched trailing edge reset
        init_regs;
      end if;
      elsif rising_edge(clock) then
        update_regs;
      end if;
      update_ports;
    end process sync_template;  -- will infer port wires ok for reset or clock
  end process
end architecture synth;
Cont’d Single process template program

source: Mike Treseler

subtype vec_t is unsigned(vec_len-1 downto 0);
constant vec_init : vec_t := "10110011";
variable reg_v : vec_t ;

procedure init_regs is -- init of register variables only
begin
    reg_v := vec_init;
end procedure init_regs;

procedure update_regs is -- distilled functional description
begin
    if a='1' then
        reg_v := rotate_left(reg_v,1);
    end if;
end procedure update_regs;

procedure update_ports is -- wire register variables out to port
begin
    q <= std_logic_vector(reg_v);
end procedure update_ports;
Synthesis result from template
http://mysite.verizon.net/miketreseler/
Increasing the abstraction level

Benefits
- Easier to understand the underlying algorithm
- Easier to modify/maintain
- Faster simulation
- Use built-in module generators (synthesis)

Problems
- Keep the code synthesizable
- Synthesis tool might choose wrong gate-level structure
- Problems to understand the algorithm for less skilled engineers
Using records

- Useful to group related signals
- Nested records further improves readability
- Directly synthesizable
- Element name might be difficult to find in synthesized netlist

```vhdl
type reg1_type is record
  f1 : std_logic_vector(0 to 7);
  f2 : std_logic_vector(0 to 7);
  f3 : std_logic_vector(0 to 7);
end record;

type reg2_type is record
  x1 : std_logic_vector(0 to 3);
  x2 : std_logic_vector(0 to 3);
  x3 : std_logic_vector(0 to 3);
end record;

type reg_type is record
  reg1 : reg1_type;
  reg2 : reg2_type;
end record;

variable v : reg_type;

v.reg1.f3 := "0011001100";
```
Using loops

- Used for iterative calculations
- Index variable implicitly declared
- Typical use:
  - iterative algorithms
  - priority encoding,
  - sub-bus extraction,
  - bus turning

```vhdl
variable v1 : std_logic_vector(0 to 7);
variable first_bit : natural;

-- find first bit set
for i in v1’range loop
  if v1(i) = ’1’ then
    first_bit := i; exit;
  end if;
end loop;

-- reverse bus
for i in 0 to 7 loop
  v1(i) := v2(7-i);
end loop;
```
Using ieee.numeric_std.all;

- Declares two additional types:
  - signed and unsigned
- Declares arithmetic operators:
  - +, -, *, /, <, >, =, <=, >=, /=
  - and conversion operators:

```vhdl
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

```vhdl
variable u1, u2, u3 : unsigned;
variable v1 : std_logic_vector;
u1 := u1 + (u2 * u3);
if (v1 >= v2) then ...
v1(0) := u1(to_integer(u2));
```

From | To | Function
--- | --- | ---
un, lv | sg | SIGNED(from)
sg, lv | un | UNSIGNED(from)
un, lv | std_logic_vector | STD_LOGIC_VECTOR(from)
sg | un | TO_INTEGER(from)
sg | un | TO_UNSIGNED(from, size)
Na | un | TO_UNSIGNED(from, size)
in | sg | TO_SIGNED(from, size)
Multiplexing using integer conversion

- **N to 1 multiplexing**
  ```vhdl
  function genmux (s, v : std_logic_vector)
    return std_logic is
    variable res : std_logic_vector (v'length –1 downto 0);
    variable i : integer;
    begin
      res := v;
      i := to_integer(unsigned(s));
      return(res(i));
    end;
  end;
  ```

- **N to 2**^**N** decoding
  ```vhdl
  function decode (v : std_logic_vector)
    return std_logic_vector is
    variable res : std_logic_vector (( 2**v’length) –1 downto 0);
    variable i : integer;
    begin
      res := (others => '0');
      i := to_integer(unsigned(v));
      res(i) := '1';
      return(res);
    end;
  end;
  ```
Conclusions

- The two-process design method provides a uniform structure, and a natural subdivision between algorithm and state.
- It will improve:
  - development time (coding, debug)
  - simulation and synthesis speed
  - readability
  - maintenance and re-use possibilities