A Structured VHDL Design Method

Jiri Gaisler

CTH / Gaisler Research
Outline of lecture

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

- Based on heritage from schematic entry (!):
  - Many small processes or concurrent statements
  - Use of TTL-like macro blocks
  - Use of GUI tools for code-generation
- Could be compared to schematic 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 signal
- 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 are used
Simple VHDL example

CbndDatat_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(6 downto 4) <= CB_In(6 downto 4);
    CBLatch_F_2(7)          <= 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 comb. signals
- Difficult to identify related signals
- Large port declarations in entity headers
- Slow execution 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 maintain
  - Synthesisable
  - Simulate as fast as possible
  - No simulation/synthesis discrepancies
  - Usable for small and large designs

- **New design style/method needed!**
A synchronous design can be abstracted into two separate parts; a combinational and a sequential.

\[ q = f(d, q_r) \]
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 \( (ri) \) 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 clock
Two-process VHDL entity

Comb. Pro.

\[ q = f_1(d, r) \]
\[ ri = f_2(d, r) \]

Seq. Process

In-ports
- d
- Clk

Out-port
- Q

ri

r
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 package.
- Input ports can be of output record types from other entities.
- 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 for 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) 
    begin 
      variable v : reg_type; 
      begin 
        v := r; v.irq := '0';
        for i in r.pend'range loop  
          v.pend := 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

```vhdl
use work.interface.all;

entity cpu is port (
    clk    : in std_logic;
    rst    : in std_logic;
    mem_in : in mem_in_type;
    mem_out : out mem_out_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, mem_out cache_out);
    u2 : mctrl port map
         (clk, rst, cache_out, mem_in, mctrl_out, mem_out);
end architecture;
```
Benefits

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

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

- **Two-process method:**
  - Add definition in register record
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 interface records, \( r \) & \( rin \)
  - Signals are grouped according to function and easy to understand
  - Addition/deletion of record elements automatically propagated to trace window
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 $v$
when others => raddr := address(26 downto 12);
end case;

-- sdram access FSM

case r.sdstate is
when idle =>
  if startsd = '1' then
    v.address(16 downto 0) := raddr;
    v.sdbs := v.address(16 downto 15);
    v.sdcss := (not ahbsi.haddr(28)) & ahbsi.haddr(26);
    v.rsan := '0';
  end if;
  v.sdstate := act1;
  v.bs := '0';
end if;
when act1 =>
  v.rsan := '1';
  if r.ncfg2.casdel = '1' then v.sdstate := act2;
  else
    v.sdstate := act3;
    v.hready := r.hwrite and ahbsi.htrans(0) and ahbsi.htrans(1);
  end if;
when act2 =>
  v.sdstate := act3;
  v.hready := r.hwrite and ahbsi.htrans(0) and ahbsi.htrans(1);
when act3 =>
  v.csan := '0',
  v.address(14 downto 0) := "0000" & r.haddr(10 downto 0);
  v.dqm := dqm; v.burst := r.hready;
  if r.hwrite = '1' then
    v.sdstate := wr1; v.sdwen := '0'; v.bdrive := "1111";
  else
    v.sdstate := rd1; end if;
when wr1 =>
  v.address(14 downto 0) := "0000" & r.haddr(10 downto 0);
  if ((r.burst and r.hready) = '1') and (r.htrans = "11") then
    v.hready := ahbsi.htrans(0) and ahbsi.htrans(1) and r.hready;
  else
    v.sdstate := wr2; v.bdrive := "0000"; v.csan := '1'; v.sdwen := '1';
  end if;
  v.dqm := "1111";
end if;
when wr2 =>
  v.rsan := '0'; v.sdwen := '0';
  v.sdstate := wr3,
  when wr3 =>
  v.sdcss := "11"; v.rsan := '1'; v.sdwen := '1';
Complete algorithm can be a sub-program

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

```plaintext
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 synthesisable 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

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

- LEON SPARC processor
- Two-process method (mostly)
- 15,000 lines of code
- 37 entities, 75 processes
- 300 signals
- 800 signal assignments
- 100 Kgates, 2 man-years, no bugs in first silicon
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 algorithm for less skilled engineers
Using records

- Useful to group related signals
- Nested records further improves readability
- Directly synthesisable
- Element name might be difficult to find in synthesised 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 : regtype;

v.reg1.f3 := "0011001100";
```
Using ieee.numeric_std.all;

- Declares to additional types: signed and unsigned
- Declares arithmetic and various conversion operators: +, -, *, /, <, >, =, <=, >=, /=, conv_integer
- Built-in, optimised versions available in all simulators and synthesis tools
- Should be preferred to std_logic_arith

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

variable u1, u2, u3 : unsigned;
variable v1 : std_logic_vector;

u1 := u1 + (u2 * u3);
if (v1 >= v2) then ...
v1(0) := u1(conv_integer(u2));
```
Use of 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;
```
Multiplexing using integer conversion

- **N to 1 multiplexing**

  ```
  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;  -- needed to get correct index
      i := conv_integer(unsigned(s));
      return(res(i));
    end;
  ```

- **N to 2**\(^2\)N decoding

  ```
  function decode(v : std_logic_vector) return std_logic_vector is
    variable res :
      std_logic_vector((2**v'length)-1 downto 0);
    variable i : natural;
    begin
      res := (others => '0');
      i := conv_integer(unsigned(v));
      res(i) := '1';
      return(res);
    end;
  ```
State machines

- Simple case-statement implementation
- Maintains current state
- Both combinational and registered output possible

```vhdl
architecture rtl of mymodule is
type state_type is (first, second, last);
type reg_type is record
    state : state_type;
    drive : std_logic;
end record;
signal r, rin : reg_type;
begin
    comb : process(...., r)
    begin
        case r.state is
            when first =>
                if cond0 then v.state := second; end if;
            when second =>
                if cond1 then v.state := first;
                elsif cond2 then v.state := last; end if;
            when others =>
                v.drive := '1'; v.state := first;
        end case;
        if reset = '1' then v.state := first; end if;
        modout.cdrive <= v.drive;  -- combinational
        modout.rdrive <= r.drive;  -- registered
    end process;
```

.
Conclusions

- The two-process design method provides a uniform structure, and a natural division between algorithm and state
- It improves
  - Development time (coding, debug)
  - Simulation and synthesis speed
  - Readability
  - Maintenance and re-use