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 synthesizable code
- A few design examples
Traditional design methods

- 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
Problems

- Slow execution due to many signals and processes
- 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
Modelling requirements

- We want our models to be:
  - Easy to understand and maintain
  - Simulate as fast as possible
  - Synthesisable
  - No simulation/synthesis discrepancies
Abstraction of digital logic

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

Mathematical expression:
\[ q = f(d, q_r) \]

Diagram:
- **Comb** block with input `d` and output `q = f(d, q_r)`.
- **DFF** (Data Flow Flip-Flop) block with input `q_r` and output `q`.
- `Clk` signal to both blocks.
- `q` output from the DFF block.
Implementing the abstracted view in VHDL:  
The two-process scheme

- A VHDL entity is made to contain only two processes: one sequential and one combinational.
- 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

Comb. Process
\[ q_{\text{cn}} = f(d,r) \]

Seq. Process

In-ports
- d
- Clk

Out-port
- \( q_c \)

Port In
- \( r_{\text{in}} \)
- \( r \)
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 interface package.
- Input ports are of output record types from other entities.
- A local variable of the registered 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)
    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 sig.)
  - 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
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, $r$ and $rin$
  - 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 \((rin)\) directly visible in variable \(v\)
when others => raddr := address(16 downto 10);
end case;

-- aclcm access FSM

    case r.adstate is
      when idle =>
        if started = '1' then
          v.adress(16 downto 0) := raddr;
          v.adrs := v.address(16 downto 12);
          v.adcm := (not aclcm_haddr(2)) & aclcm_haddr(2);
          v.ram := '0';
          v.adstate := acl1;
        end if;
      when acl1 =>
        v.ram := '1';
        if aclmg2.casdel = '1' then v.adstate := acl2;
        else
          v.adstate := acl3;
          v.rready := r lwrrite and aclcm_btrans(0) and aclcm_btrans(1);
        end if;
      when acl2 =>
        v.adstate := acl3;
        v.rready := r lwrrite and aclcm_btrans(0) and aclcm_btrans(1);
        when acl3 =>
          v.cen := '0';
          v.address(16 downto 0) := "0000" & r.haddr(16 downto 0);
          v.dgm := dgm; v.burst := r rready;
          if r lwrrite = '1' then
            v.adstate := wr1; v.adcm := '0'; v bdrive := "111111";
            if aclcm_btrans = '1' or (v.rready = '0') then v.rready := '1'; end if;
          else v.adstate := rd1; end if;
        end when;
      when wr1 =>
        v.adress(16 downto 0) := "0000" & r.haddr(16 downto 0);
        if (v.burst and r rready) = '1' then
          v.rready := aclcm_btrans(0) and aclcm_btrans(1) and r lwrrite;
        else
          v.adstate := wr2; v bdrive := "00000"; v.cen := '1'; v adcm := '1';
          v.dgm := "111111";
        end if;
      when wr2 =>
        v.ram := '0'; v.adcm := '0';
        v.adstate := wr3;
      when wr3 =>
      when vl3 =>
        v.adcm := '11'; v.ram := '1'; v adcm := '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

```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 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-V8 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 synthesisable
- 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

```plaintext

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.std_logic_arith.all;

- Written by Synopsys, now freely available
- 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
- IEEE alternative: numeric_std

```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

```
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**^**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;
begins
    
    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