HDL-based RTL design

Péter Horváth

Department of Electron Devices

March 23, 2016
Contents

- Challenges on RTL
  - Simulation-synthesis mismatch
  - Optimization fields: area, timing, power

- Synthesizable RTL
  - Synthesizable subset of HDLs
  - Synthesis of the most important VHDL language constructs
  - RTL design guidelines

- Register-Transfer Level HDL implementation schemes of data-processing systems
  - The behavioral RTL (FSMD) coding style
  - The structural RTL (FSM+D) coding style
Challenges on RTL
RTL – Register-Transfer Level

- RTL is the abstraction level between algorithm and logic gates.
- In RTL description, circuit is described in terms of **registers** (flip-flops or latches) and the **data is transferred** between them using logical operations (combinational logic, if needed). That is why the nomenclature: Register-Transfer Level (RTL).
Simulation-synthesis mismatch

- **HDLs** are the tools of RTL modeling.
- The HDLs originally were developed for **documentation purposes** (describe the behavior of existing circuits).
- Recently the only aim of RTL HDL models is to underlie the **automated logic synthesis**.
- The HDL LRM (Language Reference Manuals) define only **simulation semantics** of the language.
- Difficulties in HDL-based automated logic synthesis
  - The simulation semantics and the **synthesis semantics** of the different synthesis tools may differ.
  - The synthesis semantics of different synthesis tools may differ from each other as well.
  - There is a subset of the HDLs that is **NOT** synthesizable.
RTL optimization fields

- **Area**: The number of logic gates required for implementing the functionality of the RTL design highly depends on the RTL coding style.

- **Timing**
  - Although the critical path delay of a circuit is directly influenced by transistor-level properties, there are numerous design techniques on RTL to decrease the logic delay and increase the clock frequency hereby.
  - The RTL designer’s objective is to minimize the clock cycles required for a certain task and maximizing the throughput of the system.

- **Power**: The power consumption of a system mainly depends on device level and technology level issues but there are efficient coding techniques on RTL that are able to prevent high power consumption.
Synthesizable RTL
Synthesizable subset of HDLs

- There is a subset of the HDLs that cannot underlie an automated logic synthesis.
- The non-synthesizable subset is vendor-dependent.
- Three categories of the non-synthesizable language constructs can be distinguished
  - **Ignored** constructs: the parser of the synthesizer ignores these expressions, the post-synthesis behavior may differ from the pre-synthesis behavior
  - **Partially supported** constructs: these expressions are synthesizable with some restrictions
  - **Not supported** constructs: the parser of the synthesizer terminates with an error message
The timing-related statements are ignored by the synthesis tools.

```vhdl
process (a, b, c)
begin
  y <= (a or b) and c after 2 ns;
end process;
```

```vhdl
process (a, b, c)
begin
  y <= (a or b) and c;
end process;
```
Partly supported language constructs – examples

- In Altera Quartus II the range selection is only supported if at least one of the range boundaries is constant (computable in compilation time).

- If the target technology does not include hardware multipliers, then the multiplication operator is only supported if the multiplier is a power of 2.

- The loop statements are generally synthesizable, if the number of iterations is constant. The sub-circuit implementing the loop body is synthesized multiple times according to the maximum iteration number.

- Only one ’event attribute can be embedded into a process statement.
Not supported language constructs – examples

- In case of ASIC technology, the **initial block** of Verilog and the **initial values** of VHDL are not synthesizable. The register initialization can only be implemented with a reset signal.

- The **wait statement** of VHDL is not synthesizable. The real circuits "do not know anything about the concept of *time*", there is only cause-and-effect mechanism.

- A signal with **multiple drivers** cannot be synthesized directly. Two outputs cannot connected to each other, tri-state buffers are required.
library ieee;
use ieee.std_logic_1164.all;

entity assignment is
port (a: in std_logic_vector (3 downto 0);
b: in std_logic_vector (3 downto 0);
c: in std_logic_vector (3 downto 0);
y_1: out std_logic;
y_2: out std_logic_vector (3 downto 0));
end entity;

architecture behavior of assignment is
begin
  y_1 <= not (a(0) or b(1)) and c(2);
y_2 <= a xor c;
end architecture;

The synthesis was performed in Altera Quartus II environment.
library ieee;
use ieee.std_logic_1164.all;

entity if_complete is
port (a: in std_logic_vector (3 downto 0);
b: in std_logic_vector (3 downto 0);
c: in std_logic_vector (3 downto 0);
y_1: out std_logic;
y_2: out std_logic_vector (3 downto 0));
end if_complete;

architecture behavior of if_complete is
begin
process (a,b,c)
begin
if ( a(0) = '1' ) then
  y_1 <= b(0);
y_2 <= c;
else
  y_1 <= a(2);
y_2 <= b;
end if;
end process;
end behavior;

The synthesis was performed in Altera Quartus II environment.
Synthesis of an incomplete if-then-else statement

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

entity if_incomplete is
  port (a: in std_logic_vector (3 downto 0);
        b: in std_logic_vector (3 downto 0);
        c: in std_logic_vector (3 downto 0);
        y_1: out std_logic;
        y_2: out std_logic_vector (3 downto 0));
  end if_incomplete;

architecture behavior of if_incomplete is
begin
  process (a,b,c)
  begin
    if ( a(0) = '1' ) then y_1 <= b(0);
       y_2 <= c;
  end if;
  end process;
end behavior;
```

The synthesis was performed in Altera Quartus II environment.
Synthesis of the process statement – combinational process

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

entity comb_process is
  port (a: in std_logic_vector (3 downto 0);
        b: in std_logic_vector (3 downto 0);
        c: in std_logic_vector (3 downto 0);
        y_1: out std_logic;
        y_2: out std_logic_vector (3 downto 0));
end comb_process;

architecture behavior of comb_process is
begin
  process (a,b,c)
  begin
    y_1 <= a(2) or c(1);
    y_2 <= a and b;
  end process;
end behavior;
```

The synthesis was performed in Altera Quartus II environment.
library ieee;
use ieee.std_logic_1164.all;

entity flip_flop is
port (clk: in std_logic;
      clear: in std_logic;
      ce: in std_logic;
      din: in std_logic;
      dout: out std_logic);
end flip_flop;

architecture behavior of flip_flop is
begin
  process (clk,clear)
  begin
    if ( clear = '1' ) then dout <= '0';
    elseif ( rising_edge(clk) ) then
      if ( ce = '1' ) then dout <= din;
      end if;
    end if;
  end process;
end behavior;

The synthesis was performed in Altera Quartus II environment.
Synthesis of the process statement – signal versus variable

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

entity signal_process is
  port (clk: in std_logic;
        a: in std_logic;
        b: in std_logic;
        y: out std_logic);
end signal_process;

architecture behavior of signal_process is
signal c: std_logic;
begin
  process (clk)
  begin
    if (rising_edge(clk)) then
      c <= a and b;
      y <= c;
    end if;
  end process;
end behavior;
```

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

entity variable_process is
  port (clk: in std_logic;
        a: in std_logic;
        b: in std_logic;
        y: out std_logic);
end variable_process;

architecture behavior of variable_process is
begin
  variable c: std_logic;
  process (clk)
  begin
    if (rising_edge(clk)) then
      c := a and b;
      y <= c;
    end if;
  end process;
end behavior;
```

The synthesis was performed in Altera Quartus II environment.
Synthesis of the arithmetic operators

- The synthesis of the arithmetic depends on the libraries and packages used.

- The most widely used arithmetic packages
  - integer packages in the ieee library
    - std_logic_unsigned: all standard logic vectors (std_logic_vector) are interpreted as unsigned integers in arithmetic and comparison operations
    - std_logic_signed: all standard logic vectors are interpreted as signed integers in arithmetic and comparison operations
    - std_logic_arith: the desired interpretation of the standard logic vectors should be declared at the operator calls
    - numeric_std: a standardized version of the std_logic_arith library
  - fixed-point and floating-point packages in the ieee\(^1\) library
    - fixed_pkg
    - float_pkg

\(^1\) another versions of these packages can be found in the floatfixlib library as well
The FSMs consist of three parts: state register, next-state logic, and output logic.

Based on the number of processes used to describe the FSM three approaches can be distinguished

- **one-process**: The parts of the FSM are described in the same clocked process. The outputs are buffered. That means, that the resource requirement is high but the timing is reliable (the control signals are glitch-free) and the clock-to-output delay is low.

- **two-process**: The next-state logic and the state register are described in the same process but the output logic is modeled with a separate combinatorial process. The outputs are not buffered, so the resource requirement is lower but the timing is less reliable, because the mealy-inputs may cause timing violations.

- **three-process**: An own process is assigned to the structural parts of the FSM respectively. The HDL code is difficult to read and the mealy-inputs may cause timing violations.
Synthesis of FSMs – one process

The synthesis was performed in Altera Quartus II environment.
The synthesis was performed in Altera Quartus II environment.
The synthesis was performed in Altera Quartus II environment.

L_STATE: process (clk)
begin
    if ( rising_edge(clk) ) then
        if ( reset = '1' ) then
            pres_state <= s0;
        else
            pres_state <= next_state;
        end if;
    end if;
end process;

L_NSL: process (pres_state, start, stop)
begin
    case pres_state is
        when s0 => if ( start = '1' ) then
            next_state <= s1;
        else
            next_state <= s0;
        end if;
        when s1 => if ( stop = '1' ) then
            next_state <= s0;
        else
            next_state <= s1;
        end if;
        when others => report "?" severity failure;
    end case;
end process;

L_OL: process (pres_state)
begin
    case pres_state is
        when s0 => output <= '0';
        when s1 => output <= '1';
        when others => report "?" severity failure;
    end case;
end process;
There are numerous RTL design techniques making it possible to obtain appropriate reliability, low area, high speed, or low power consumption.

The aim of guidelines

1. They help to avoid undesirable hardware
2. They help to maintain the identical behavior between the RTL model and the gate level model
### RTL design guidelines

- **Avoid undesirable hardware**
  1. **To avoid latches**, set all outputs of combinatorial blocks to default values at the beginning of the sequential blocks.
  2. **To avoid internal buses**, do not assign regs/signals from two separate always blocks/processes.
  3. **To avoid tristate buffers**, do not assign the value 'Z' (VHDL) or 1’bz (Verilog).

- **RTL versus gate level behavior**
  1. All inputs must be listed in the sensitivity list of a combinatorial block.
  2. The clock and asynchronous reset must be in the sensitivity list of a sequential block.
  3. Use a non-blocking assignment when assigning to a reg intended to be inferred as a flip-flop (Verilog). (in VHDL: use a signal assignment instead of a variable assignment)
RTL HDL Implementation Schemes for Data-Processors
Concepts – Data processing systems

- It performs **transformations** on input data and transfers the processed data to outputs.
- It may include internal **data-storage** subsystems.
- The data manipulation is based on a **controlling mechanism** (application-specific algorithm or stored program).
- Two categories of the data-processor resources can be distinguished:
  - **controlling** resources\(^2\) are related to the controlling mechanism
  - **datapath** resources\(^3\) are related to the data-manipulation and internal data-storage
- **Signal and I/O types**
  - **data/control I/O**: direct interface between the outside world and the datapath/controlling resources
  - **control signal**: from control resources to datapath resources
  - **status signal**: from datapath resources to controlling resources

---
\(^2\) e.g. FSM, microprogrammed controller
\(^3\) e.g. ALU, register-file
Since HDLs are **rich in language constructs**, a functional description can be transformed into a RTL model diversely.

**Examples**
- an adder can be implemented as a complete design entity or as a subroutine or function.
- a register can be implemented as a complete design entity or a single signal inside another design entities architecture body.

**An implementation scheme is a proposed HDL coding style that determines**
- the HDL **model structure**
- applied **language constructs** for implementation of the functional model elements
- clocking scheme
The behavioral RTL (FSMD) coding style

- FSMD: Finite State Machine *with* Datapath

- The HDL model includes the control resources and the datapath resources as well in a **single design entity** (coarse-grained HDL model).

- The HDL model includes the controller: The design entity describing the system is a **FSM implemented in a single process**.

- The HDL model includes the datapath resources
  - data-manipulation: operator, subroutine, function calls
  - data-storage: signals and arrays of signals
The behavioral RTL (FSMD) coding style

**behavioral RTL model example**

```vhdl
architecture behavioral_RTL of sum is

    type   state_type is (s0, s1, s2, s3, s4, s5);
    signal state: state_type := s0;
    signal s_acc: std_logic_vector (7 downto 0) := X"00";

begin

    process (clk)
    begin
        if ( rising_edge(clk) ) then
            if ( reset = '1' ) then
                s_acc <= X"00";
                state <= s0;
            else
                case state is
                    when s0 =>
                        if ( start = '1' ) then
                            ready <= '0';
                            s_acc <= X"00";
                            state <= s1;
                        else
                            state <= s0;
                        end if;
                    when s1 =>
                        s_acc <= s_acc + in1;
                        state <= s2;
                    when s2 =>
                        s_acc <= s_acc + in2;
                        state <= s3;
                    when s3 =>
                        s_acc <= s_acc + in3;
                        state <= s4;
                    when s4 =>
                        s_acc <= s_acc + in4;
                        state <= s5;
                    when s5 =>
                        if ( s_acc = X"00" ) then
                            zero <= '1';
                            else
                                zero <= '0';
                            end if;
                            ready <= '1';
                            state <= s0;
                        when others => report "?" severity failure;
                    end case;
                end if;
            end if;
        end process;

    acc <= s_acc;

end behavioral_RTL;
```

...
The structural RTL (FSM+D) coding style

- FSM+D: Finite State Machine + Datapath
- The controller, the datapath, the data-manipulating, and storage resources are described in separate design units (fine-grained HDL model).
- The controller is a finite state machine. The design unit including the FSM does not include any data-storage resources (except the state register). The FSM description only includes port-assignments.
- The datapath only includes the instantiations of the data-manipulating (e.g. ALUs) and storage (registers, register files) resources.
The structural RTL (FSM+D) coding style

structural RTL model example – components

architecture behavior of standard_register is
begin
  process (clk)
  begin
    if ( rising_edge(clk) ) then
      if ( reset = '1' ) then dout <= X"00";
      elsif ( ce = '1' ) then dout <= din;
      end if;
    end if;
  end process;
end behavior;

architecture behavior of nor_8 is
begin
  zero <= '1' when in1 = X"00" else '0';
end behavior;

architecture behavior of mux5 is
begin
  process (sel,input0,input1,input2,input3,input4)
  begin
    case (sel) is
    when "000" => output <= input0;
    when "001" => output <= input1;
    when "010" => output <= input2;
    when "011" => output <= input3;
    when "100" => output <= input4;
    when others => output <= (others => '-');
  end case;
end process;
end behavior;

architecture behavior of adder is
begin
  result <= std_logic_vector( unsigned(in1) + unsigned(in2) );
end behavior;
The structural RTL (FSM+D) coding style

architecture behavior of controller is
  type state_type is (s0,s1,s2,s3,s4,s5,s6);
  signal state: state_type := s0;
begin
  process (clk) begin
    if ( rising_edge(clk) ) then
      if ( reset = '1' ) then
        state <= s0; sel_acc <= "000"; ce_acc <= '0';
        ready <= '0'; zero <= '0';
      else
        case state is
          when s0 => if ( start = '1' ) then
            sel_acc <= "100"; ce_acc <= '1';
            state <= s1;
          else state <= s0; end if;
          when s1 => sel_acc <= "000"; state <= s2;
          when s2 => sel_acc <= "001"; state <= s3;
          when s3 => sel_acc <= "010"; state <= s4;
          when s4 => sel_acc <= "011"; state <= s5;
          when s5 => ce_acc <= '0'; state <= s6;
          when s6 => zero <= ss_zero; ready <= '1';
            state <= s0;
          when others => report "?" severity failure;
        end case;
      end if;
    end if;
  end process;
end behavior;

architecture structure of datapath is
  signal from_mux: std_logic_vector (7 downto 0);
  signal from_add: std_logic_vector (7 downto 0);
  signal from_acc: std_logic_vector (7 downto 0);
  signal from_nor: std_logic;
begin
  L_MUX: entity work.mux5(behavior)
    port map (in1,in2,in3,in4,X"00",
      sel_acc,
      from_mux);
  L_ADD: entity work.adder(behavior)
    port map (from_acc,from_mux,
      from_add);
  L_ACC: entity work.standard_register(behavior)
    port map (clk,reset,
      ce_acc,
      from_acc);
  L_NOR: entity work.nor_8(behavior)
    port map (from_acc,from_nor);
  ss_zero <= from_nor;
  acc <= from_acc;
end structure;
Additional readings

- Sanjay Churivala, Sapan Garg – Principles of VLSI RTL Design
- Enoch O. Hwang – Digital Logic and Microprocessor Design with VHDL
- J. Bhasker – Verilog HDL Synthesis
- Weng Fook Lee – VHDL Coding and Logic Synthesis with SYNOPSIS
- Janick Bergeron – Functional Verification of HDL models