Course 2
Logic Circuit Design - Register Transfer Level (RTL)

Prof. Marius Marcu

Universitatea Politehnica Timisoara

marius.marcu@cs.upt.ro

October 5, 2017

Overview

Register Transfer Level Methodology

- Design methodology using Hardware Description Languages (HDLs) - Verilog or VHDL
  - Source code is Verilog/VHDL
- Abstraction layer for digital circuits
  - Provides a highly detailed description of hardware architecture
    - Pros - efficient description of the targeted architecture
    - Cons - difficult to implement/long design time with respect to software development
    - However, at software level, architectural trade-offs cannot be described

Register Transfer Level Methodology

- Technology independent
  - Allows portability of code for different technologies - significant amount in legacy code
  - Cannot describe circuit level blocks, such as clocking network, IO buffers power supply or ground lines
    - Front-end design cycle - technology independent
    - Back-end design cycle - technological implementation using a specific ASIC (e.g. 28 nm, 22 nm, 14 nm) or FPGA implementation
  - Timing representation - in clock cycles
    - Cannot determine the maximum clock frequency for a digital circuit based on the RTL representation
    - Accurate performance evaluation - based on the clock frequency estimates - can be performed after logic synthesis for a specific technology

RTL Design Principles

- RTL is based on the following:
  - Description of registers within the described hardware module
  - Description of the interaction between the registers for a clock cycle
- The registers are composed of synchronous master-slave D-Flip-Flops
  - Clocked circuits - clock signal input
  - Flip-flop update - on the edge (usually positive edge) of the clock signal - ensures stable (constant) output between two positive edges
  - Implementation using D-FF due to the suitable implementation in CMOS technology for it (more cost efficient with respect to JK or RS)

RTL coding styles

- One segment
  - More compact description
  - The entire circuit/block is viewed as a single clocked entity
  - Bug prone
- Two segment
  - One segment for sequential elements
  - One segment for combinational elements
  - More lines of code with respect to one segment
  - Clear description - significant less bugs with respect to one segment
  - Preferred coding style

2-segment RTL

- Segment for sequential component
  - Each sequential component is viewed as an N-bit parallel-in, parallel-out register using N D-FFs
  - Inputs: clock and reset signals
  - Data inputs: the N-bit data input to the register
  - Data outputs: the N-bit data output of the register
  - Data output is stable (constant) during a clock cycle (due to usage of FFs)
  - The data output updates with the value of the data input each positive edge of the clock signal

2-segment RTL

- Segment for combinational component
  - Inputs:
    - Inputs of the circuit, other than clock and reset (clock and reset are associated only to the sequential part)
    - The outputs of the sequential component
  - Outputs:
    - Outputs of the circuit
    - Inputs for the sequential components

Combinational circuits

- Logic circuits for which any change in the inputs lead to an "instant" evaluation of the output, and a possible change in it
- Combinational circuits do not have memory

Multiplexers

Combinational circuits with n-inputs, $\log_2 n$ selection inputs and 1 output, which connects the output to the input specified by the binary combination of the selection inputs.

```
module mux4_1
    (input [3:0] d_in,
    input [1:0] sel,
    output reg d_out);

    always
        @*
    begin
        d_out = 0;
        case(sel)
            2'b00:  d_out = d_in[0];
            2'b01:  d_out = d_in[1];
            2'b10:  d_out = d_in[2];
            2'b11:  d_out = d_in[3];
        endcase
    end
endmodule
```

Demultiplexers

Combinational circuits with 1 input, $\log_2 n$ selection inputs and n outputs, which connects the input to the output specified by the binary combination of the selection inputs.

```
module demux4_1
    (input d_in,
    input [1:0] sel,
    output reg [3:0] d_out);

    always
    @*
    begin
        d_out = 4'b0000;
        case(sel)
            2'b00:  d_out = {3'b000, d_in};
            2'b01:  d_out = {2'b0, d_in, 1'b0};
            2'b10:  d_out = {1'b0, d_in, 2'b0};
            2'b11:  d_out = {d_in, 3'b000};
        endcase
    end
endmodule
```

Decoders

Combinational circuits with n inputs and 2^n outputs, which activates the output corresponding to the binary value of the inputs.

```
module dec2_4
    (input [1:0] in_vec,
    output reg [3:0] y);

    always
        @*
    begin
        y=0;
        case(in_vec)
            2'b00:  y=4'b0001;
            2'b01:  y=4'b0010;
            2'b10:  y=4'b0100;
            2'b11:  y=4'b1000;
        endcase
    end
endmodule
```

Priority encoders

Combinational circuit with n inputs and $\log_2 n$ outputs, for which the output represent the binary value of the active inputs. In case 2 or more inputs are active at the same time, a priority rule is applied.

```
module enc4_2
//priority rule: 1,2,0,3 (1 - highest priority)
//output active denoting whether at least 1 input is active
(input [3:0] in_vec,
output reg [2:0] y,
output reg active);

    always
        @*
    begin
        y=2'b0;
        active = 1'b0

        if(in_vec[1] == 1'b1)
            begin
                y=2'b01;
                active = 1'b1
            end
        else
            if(in_vec[2] == 1'b1)
                begin
                    y=2'b10;
                    active = 1'b1;
                end
            else
                if(in_vec[0]==1'b1)
                    begin
                        y=2'b00;
                        active = 1'b1;
                    end
                else
                    if(in_vec[3]==1'b1)
                        begin
                            y=2'b11;
                            active = 1'b1;
                        end
    end
endmodule
```

Arithmetic circuits

- Addition, subtraction: use +/- operators, as most libraries contain optimized adders for their corresponding technologies
- Multiplication: OK to use * operator for FPGA; for ASIC, a multiplier has to be designed
- Division, square root: do not use operator; dividers and square rooters have to be designed

Arithmetic circuits

- Adders

```
module adder_4
    (input [3:0] a,
    input [3:0] b,
    output reg [3:0] sum);

    always
        @*
    begin
        sum = a+b;
    end
endmodule
```

- FPGA multipliers

```
module mult_4
    (input [15:0] a,
    input [15:0] b,
    output reg [31:0] prod);

    always
        @*
    begin
        prod = a*b;
    end
endmodule
```

Combinational circuits

- The sensitivity list in the *always* should include only input signals
- However, in most common cases (not toy cases), the combinational block include more then 10 input signals - forgetting to specify one of them is highly probable
- @$*$ - denotes all the signals in the module are in the sensitivity list
- It is the preferred usage in the sensitivity list

Sequential circuits

- Sequential circuits - used as memory elements in digital logic
- Composed of master-slave D-Flip-Flops
- The basic sequential components is considered the parallel-in/parallel-out register composed of n D-Flip-Flops
- For each register, two signals are defined:
  - *_reg* signal - corresponds to the output of the register and represents the value in the current clock cycle
  - *_nxt* signal - corresponds to the data input of the register and represents the value which will be update in the following positive edge of the clock cycle
- Sensitivity list for the sequential part will always include the active edge of the clock signal (e.g. posedge) and (optional) the active edge of the reset

D Flip-flop

- Basic sequential element used in logic circuits
- Composed of 2 D-latches in master-slave architecture
- Update on the clock signal edge (usually positive edge)
- Output stable (constant) between two consecutive positive edges

```
module d_ff //FF with no reset input
    (input clk,
    input d_nxt,
    output reg d_reg);

    always
        @(posedge clk)
    begin
        d_reg <= d_nxt;
    end
endmodule
```

Synchronous vs asynchronous reset

- Synchronous reset - the reset signal is active only at active edge of the clock signal
- In this case, it does not appear in the sensitivity list associated to the sequential component
- Due to noise from the mechanical push-buttons used for reset purposes, synchronous reset is preferred

```
module d_ff
    (input clk, rst
    input d_nxt,
    output reg d_reg);

    always
        @(posedge clk)
    begin
        if(rst == 1'b0)
            begin
                d_reg <= 0;
            end
        else
            begin
                d_reg <= d_nxt;
            end
    end
endmodule
```

Synchronous vs asynchronous reset

- Asynchronous reset - the reset signal is active all the time
- The reset event (negative or positive edge) appears in the sensitivity list associated to the sequential component

```
module d_ff
    (input clk, rst
    input d_nxt,
    output reg d_reg);

    always
        @(posedge clk, negedge rst)
    begin
        if(rst == 1'b0)
            begin
                d_reg <= 0;
            end
        else
            begin
                d_reg <= d_nxt;
            end
    end
endmodule
```

Parallel-in parallel-out register

- It is composed of n D-Flip-Flops, with no interconnection between them
- Can be seen a generalization of the D-Flip-Flop

```
module reg8_bit
//with synchorunous reset
    (input clk, rst
    input [7:0] d_nxt,
    output reg [7:0] d_reg);

    always
        @(posedge clk)
    begin
        if(rst == 1'b0)
            begin
                d_reg <= 0;
            end
        else
            begin
                d_reg <= d_nxt;
            end
    end
endmodule
```

Serial-in serial-out register

Register with 1 data input, 1 data output, whose behaviour is described by a shift register.

```
module reg8_siso
//with synchorunous reset and right shifting behaviour
//data input updates the most significant bit
(input clk, rst
input data_in,
output data_out);
//signal corresponding to the data register
    reg [7:0] d_reg, d_nxt;

//sequential logic segment
    always
        @(posedge clk)
    begin
        if(rst == 1'b0)
            begin
                d_reg <= 0;
            end
        else
            begin
                d_reg <= d_nxt;
            end
        end
```

```
//combinational logic segment
    always
        @*
    begin
        //circuit output evaluation
        data_out = d_reg[0];
        //next value of the data register evalu
        d_nxt = {data_in, d_reg[7:1]};
    end
endmodule
```

Counters

- Sequential circuits used to count through a predefined range of states
- A wide range of forms or applications:
  - Timers - increase/decrease counters which update their value each clock cycle; the counter counts a number of clock cycles determined by the desired measured time ($T_{clock\_cycle} * Number\_clock\_cycles\_per\_time\_unit$)
  - Event counters - increase/decrease their value each time an input signal is activated (the input signal represents the event)
  - Encoding counters - their counting values are given for a specific encoding - e.g. Gray counters for Gray encoding

Timers

- One second measurement for an input clock cycle of 1 MHz
- After each second, an output signal is activated for 1 clock cycle

```
module timer
(input clk, rst
output sec);
//signal corresponding to the data register
reg [23:0] cnt_reg, cnt_nxt;

//sequential logic segment
    always
        @(posedge clk)
    begin
        if(rst == 1'b0)
            begin
                cnt_reg <= 0;
            end
        else
            begin
                d_reg <= d_nxt;
            end
    end
```

```
//combinational logic segment
    always
        @*
    begin
        sec = 0;
        cnt_nxt = cnt_reg + 1;

        if(cnt_reg == 1000000)
            begin
                cnt_nxt = 0;
                sec = 1'b1;
            end
    end
endmodule
```

Event counters

- Increase the value of the counter each time every input becomes active

```
module event_counter
    (input clk, rst,
    input active_event
    output [15:0] cnt_events);
//signal corresponding to the data register
    reg [15:0] cnt_reg, cnt_nxt;

//sequential logic segment
    always
        @(posedge clk)
    begin
        if(rst == 1'b0)
            begin
                cnt_reg <= 0;
            end
        else
            begin
                d_reg <= d_nxt;
            end
        end
```

```
//combinational logic segment
    always
    @*
    begin
        cnt_nxt = cnt_reg;

        if(active_event == 1'b1)
            begin
                cnt_nxt = cnt_reg + 1;;
            end
        cnt_out = cnt_reg;
    end
endmodule
```

Next week

- **RTL at full throttle:**
- Implement algorithms in hardware