Computer Electronics

Lecture 8: Digital Circuits and Verilog III

TÉCNICO LISBOA

Verilog/iob-lib: what we have learned

- Create modules with module/endmodule
  - Create interface signals with `INPUT, `OUTPUT, `INOUT
- Instantiate components in modules with component name – instance name – interface signals
- Create registers with `REG
- Create combinational circuits with `COMB

Registers

Register with enable signal

- Last class: REG(CLK, OUT, IN)
- REG_E(CLK, EN, OUT, IN)
  - CLK: clock signal
  - EN: enable signal
  - OUT: output signal
  - IN: input signal
  - How it works: OUT registers IN at clock rising edge only **if** EN is asserted, otherwise keeps previous value
- Verilog description of register:

```
always @(posedge clk)
  if(en)
    my_reg <= my_reg_next;
```

- Always: process repeats endlessly; if: like in software but used in hardware descriptions, <= : non-blocking assignment
- Or just use iob-lib:

```
`REG(clk, en, my_reg, my_reg_next)
```

Register with synchronous reset

- REG_R(CLK, RST, RST_VAL, OUT, IN)
  - CLK: clock signal
  - RST: synchronous reset signal
  - RST_VAL: value after reset
  - OUT: output signal
  - IN: input signal
  - How it works: **if** RST is asserted then OUT takes value RST_val on the **next clock rising edge**; **else** OUT registers IN at clock rising edge, otherwise keeps previous value

Verilog for register with <u>synchronous</u> reset

- Verilog description

```
always @(posedge clk)
    if(rst)
        my_reg <= some_init_value;
    else
        my_reg <= my_reg_next;
```

- Or simply use iob-lib:
  - `REG_R(clk, rst, some_init_val, my_reg, my_reg_next)

Register with enable and synchronous reset

- REG_RE(CLK, RST, RST_VAL, EN, OUT, IN)
  - CLK: clock signal
  - RST: synchronous reset signal
  - RST_VAL: value after reset
  - OUT: output signal
  - IN: input signal
  - How it works: **if** RST is asserted then OUT takes value RST_val on the **next clock rising edge**; **else** OUT registers IN at clock rising edge only if enable is asserted, otherwise keeps previous value

Verilog for register with enable and synchronous reset

- Verilog description

```
always @(posedge clk)
    if(rst)
        my_reg <= some_init_value;
    else if (en)
        my_reg <= my_reg_next;
```

- Or simply use iob-lib:
  - `` `REG_RE(clk, rst, some_init_val, my_reg, my_reg_next) ``

Register with asynchronous reset

- REG_AR(CLK, RST, RST_VAL, OUT, IN)
  - CLK: clock signal
  - RST: asynchronous reset signal
  - RST_VAL: value after reset
  - OUT: output signal
  - IN: input signal
  - How it works: **if** RST is asserted then OUT takes value RST_VAL **immediately**; **else** OUT registers IN at clock rising edge, otherwise keeps previous value

Verilog for register with <u>asynchronous</u> reset

- Verilog description

```
always @(posedge clk, posedge rst)
   if(rst)
      my_reg <= some_init_value;
   else
      my_reg <= my_reg_next;
```

- Or simply use iob-lib:
  - `REG_AR(clk, rst, some_init_value, my_reg, my_reg_next)

Register with enable and asynchronous reset

- REG_ARE(CLK, RST, RST_VAL, EN, OUT, IN)
  - CLK: clock signal
  - RST: asynchronous reset signal
  - RST_VAL: value after reset
  - EN: enable signal
  - OUT: output signal
  - IN: input signal
  - How it works: **if** RST is asserted then OUT takes value RST_VAL **immediately**; **else if** enable is asserted, OUT registers IN at clock rising edge; otherwise keeps previous value

Verilog for register with enable and asynchronous reset

- Verilog description

```
always @(posedge clk, posedge rst)
   if(rst)
      my_reg <= some_init_value;
   else if (en)
      my_reg <= my_reg_next;
```

- Or simply use iob-lib:

```
`REG_ARE(clk, rst, some_init_value, en, my_reg, my_reg_next)
```

Counters

Simple counter

- COUNTER(CLK, CNT): free running, no reset => IMPOSSIBLE TO SIMULATE!!!
- COUNTER_R(CLK, RST, CNT)
  - CLK: clock signal
  - RST: synchronous reset signal
  - CNT: counter signal
  - How it works: if RST is asserted CNT will be 0 on the next clock rising edge; otherwise CNT will be incremented by one on the next clock rising edge.
- Verilog description of counter:

```
always @(posedge clk)
   if(rst)
      my_cnt <= 0;
   else
      my_cnt <= my_cnt+1;
```

- Or just use iob-lib:

```
`COUNTER_R(clk, rst, my_cnt)
```

Other counter types

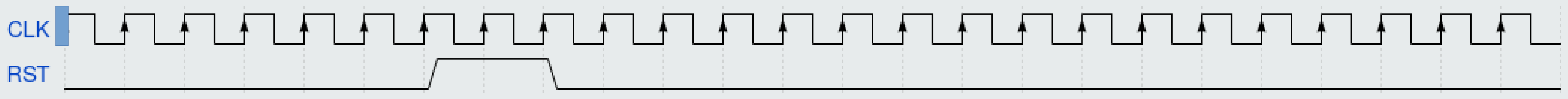
- COUNTER_RE(CLK, RST, EN, CNT)
- COUNTER_AR(CLK, RST, CNT)
- COUNTER_ARE(CLK, RST, EN, CNT)
- WRAPCNT_R(CLK, RST, CNT, WRAP_VAL)

```
always @(posedge clk)
    if(rst)
      my_cnt <= 0;
    else if (cnt == my_wrap_value)
      my_cnt <= 0;
    else
      my_cnt <= my_cnt + 1;
```

- WRAPCNT_RE(CLK, RST, EN, CNT, WRAP_VAL)
- WRAPCNT_AR(CLK, RST, CNT, WRAP_VAL)
- WRAPCNT_ARE(CLK, RST, EN, CNT, WRAP_VAL)

lob-lib macros for the testbench

- CLOCK(CLK_NAME, PER)
  - CLK: Clock name
  - PER: Clock period in units given by the `timescale directive
  - Initial value is 1, first high-low (1→0) transition happens at time PER/2
- RESET(RST_NAME, RISE_TIME, DURATION)
  - RST_NAME: reset signal name
  - RISE_TIME: time instant of the first low-high transition
  - DURATION: active reset duration in units given by the `timescale directive



The timescale directive

`timescale 1ns/ps

- Used to specify <u>simulation</u> timing
- #<x.y> is a delay of x nanoseconds + y picoseconds (eg. 3.045 ns)
- #<x.y> directives are meaningless and ignored during synthesis; only meaningful in simulation
- In synthesis you cannot force <u>when</u> something will happen
- The most you can do is to impose a time limit, e.g, a clock period
- For example

  `COMB #10 c = a & b;

  simulates an AND gate that responds in 10ns but synthesizes an AND gate that responds whenever the respective physical gate responds

Explaining the CLOCK iob-lib macro

- `define CLOCK(CLK, PER) reg CLK always #(PER/2) CLK = ~CLK
- If you type `CLOCK(clk, 10) you create a clock signal <u>in the testbench</u> having a period of 10ns
- The `CLOCK macro does not work in design files that are synthesized, only in simulation
- Synthesis tools cannot implement clocks
- A clock is an oscillator, mostly analog, and has been studied in the Electronics II course
- However, the synthesis tool needs to know that a signal is a clock in order to impose <u>timing constraints</u>

Explaining the RESET iob-lib macro

```
`define RESET(RST_NAME, RISE_TIME, DURATION) reg RST=0; \
initial begin #RISE_TIME RST=1; #DURATION RST=0; end
```

- If you type `RESET(rst, 10, 20) you create a reset pulse signal <u>in the testbench</u> having a duration of 10ns rising at time 10
- The `RESET macro does not work in design files that are synthesized, only in simulation
- Synthesis tools cannot implement reset pulses of an <u>arbitary duration</u>
- A reset signal with an arbitray duration is implemented by a pulse generator also studied in the basic Electronics courses
- However it is possible to synthesize pulse generators if the duration of the pulse is a multiple of the clock period
- The synthesis tool does need to be told that a signal is a reset: it can infer this info from the Verilog code itself

The Wavedrom program

- In the iob-soc repo you can see the hardware-softwate-documentation triumvirate
- Documentation is key to a successful product
- Timing diagrams are a succint and effective way to document
- However, creating timing diagrams by hand is timing consuming
- One possibility is to simulate and then screen shoot the waveforms into a timing diagram for the documentation
  - Avoids errors: design and documentation become consistent
  - However to update the document, you need to rerun the simulation and repeat the process
- The Wavedrom program, available online and offline is a very productive tool to create complex timing diagrams – visit wavedrom.com
- The input file is a .json file, a user-friendly type of dictionary used in Java and also in Python
- An automation flow consisting of simulation→.vcd file→vcd2wavedrom→wavedrom→png file is possible and is being developed at IObundle