VLSI 1 - Notes Week 4

Ruben Schenk, ruben.schenk@inf.ethz.ch

January 11, 2022

Chapter 3: SystemVerilog Design – Sequential Circuits

3.1 Introduction to Circuits

Combinational circuits generate an output using only the present inputs. They are simple, but also a bit limited. There are only so many functions you can describe.

Sequential circuits have memory. The output of a sequential circuit is a function of present inputs and all *past inputs.* The present **state** of the sequential circuit is saved internally.

In VLSI 1 we will use *synchronous single edge triggered circuits.* The state will be saved in rising (falling) edge triggered flip-flops (registers). This is the simplest way to design digital circuits. The largest part of a sequential circuit is actually combinational. Sequential circuits divide the operation into time slots. Present state and the inputs are used to calculate the next state. The next state is saved in the flip-flops.

The **clock signal** is used to tell when to move from one state to the next state.

3.2 Sequential Circuits

Register Transfer Level (RTL) is a generic way of designing digital circuits:

- State is stored in registers
- During a time slot, inputs and present state is used to calculate the next state
- Next state is stored in a register
- The clock moves the circuit from the present state to next state

RTL defines the datapath circuits, which process data and do the main work, and *finite state machines,* which generate control signals for the datapath, defining what operations will be done.

Sequential circuits will need the `always` statement. We call this statement a *process.* `always`has a **sensitivity** list: every time a signal in the sensitivity list changes, the body of `always` is executed.

Remark: The process body is executed *sequentially,* unlike normal descriptions in SystemVerilog.

```
module example(
    //...
);

logic state_q, state_d, a_i, cnt;
always @ (state_q, a_i, cnt) begin
    state_d = 1' bo;
    if(cnt == 1' b1) begin
        state_d = state_q;
    end else begin
        state_d = a_i;
    end
end // always
```

As we see in the example, we use `begin ... end` for longer statements. If one has only a single statement in the process, SystemVerilog allows you to skip `begin ... end`, but it is not recommended doing so.

The example above results in a combinational circuit. It will do the same as `assign state_d = (cnt) ? state_q : a_i;`.

Remarks:

- You cannot assign the same signal in multiple processes! Basically, one process determines the value of a signal.
- A signal assigned in a process cannot be assigned to by a concurrent statement.
- Only *one* of the three statements in the example below can be used to determine the value of `state_d`.

```
module example (
    //...
);

logic a, state_d;

// either this statement
always @ (a) begin
    state_d = ~a;
end

// or this statement
always @ (a) begin
    state_d <=a;
end

// or this, but only one of the three
assign state_d = 1'b0;

endmodule
```

Remark:

- `=` is a *blocking statement.* In a `always` block, the line of code will be executed only after its previous line has executed. Hence, they happen one after another.
- `<=` is a *non-blocking statement.* This means that in a `always` block, every line will be executed in parallel, hence leading to the implementation of sequential elements.

3.3 States

The way you store a **state** in Verilog is by *forgetting* to define the output signal for all cases. In the example below, we don't know what happens when `cnt == 1' b0`. We therefore preserve the prior value of `state_q` and *memorize* it:

```
// Warning: Older Verilog Syntax
// COMBINATIONAL
always @ (state_d, a_i, cnt) begin
    state_q <= 1' b0;
    if (cnt == 1'b1) begin
        state_q <= state_d;
    end else begin
        state_q <= a_i;
    end
end

// SEQUENTIAL
always @ (state_d, a_i, cnt) begin
    // no initial value for state_q
```

```
    if (cnt == 1' b1) begin
        state_q <= state_d;
        // no else statement
    end
end

// D-type latch definition
always @ (state_d, clk_c) begin
    if (clk_c == 1' b1) begin
        state_Q <= state_d;
    end
end
```

The sensitivity list can have *attributes*, such as:

- `posedge` is a transition from 0 to 1
- `negedge` is a transition from 1 to 0

We usually also add a `reset` signal to determine initial values of flip-flops. Traditionally, the Reset is active low. New FPGAs use an active high reset (in that case we'd write `@posedge rst_i`):

```
// Warning: Older Verilog Syntax
always @ (posedge clk_ci, negedge rst_ni) begin
    if (rst_ni == 1'b0) begin
        ff_q <= 1' b0;
    end else begin
        ff_q <= ff_d;
    end
end
```

SystemVerilog introduces *intent* to `always` statements. There are three different `always` statements in SystemVerilog:

- Combinational circuit: `always_comb`
- Latches: `always_latch`
- Flip-flops: `always_ff`

`always_comb` has no sensitivity list. The process will be triggered when signals change. This essentially replace the old syntax of `always @ (*)`. We revisit the previous examples again:

- Combinational: `always_comb`
  - No sensitivity list needed
  - We will use blocking (`=`) statements

```
always_comb begin
    state_d = 1' b0;
    if (cnt == 1' b1) begin
        state_d = state_q;
    end else begin
        state_d = a_1;
    end
end
```

- Latches: `always_latch`
  - We will rarely use latches
  - This example is without a reset
  - We will use non-blocking (`<=`) statements

```
always_latch @ (clk_ci) begin
    if (clk_ci == 1'b1) begin
        state_q <= state_d;
    end
end
```

- Flip-flops: `always_ff`
  - Our main state holding element
  - This example is without a reset
  - We will use non-blocking (`<=`) statements

```
always_ff @ (posedge clk_ci) begin
    state_q <= state_d;
end
```

3.4 More on Naming Conventions

We expand our naming conventions a bit:

- We will add `_d` to signals that will be the *next state* signals of an FF
  - The `_d` signal will be assigned in a `always_comb` process
- We will add `_q` to signals that will be the *present state* if an FF
  - The `_q` signals will be assigned in a `always_ff` process

3.5 FSM Design

The basic SystemVerilog template for an FSM looks as follows:

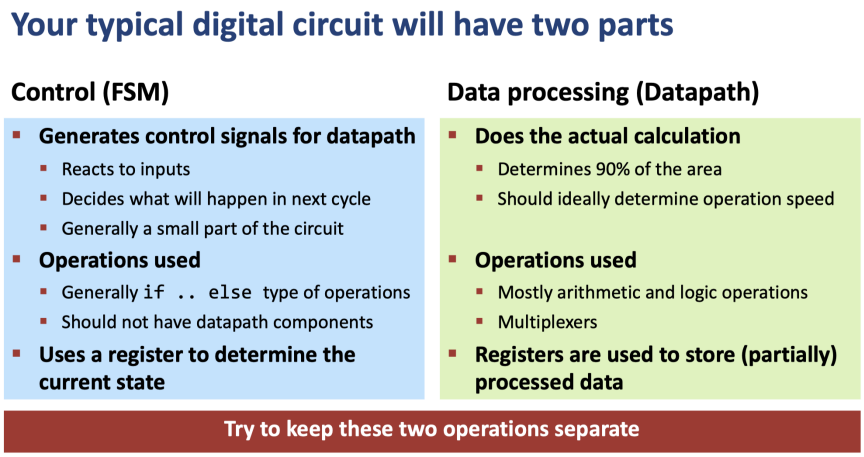
- Next state logic: Calculates the next state `state_d` depending on inputs and present `state_q`
- State holding element: One register with a reset
- Output logic:
  - Moore: Only dependent in state
  - Mealy: Dependent on state and inputs

```
// Next State Logic
assign state_d = (cont) ? state_q : ~state_q;

// State Holding Element
always_ff @ (posedge clk_ci, negedge rst_ni) begin
    if (rst_ni == 1'b0) begin
        state_q <= 1' b0;
    end else begin
        state_q <= state_d;
    end
end

// Output Calculation
assign out_o = ~state_q;
```

Your typical digital circuit will have two parts:



Enumerated types can be very handy for states:

- In an enumerated type you provide a list of names: these will internally map to binary code, it is up to the tool to decide.
- One can test and assign these types: makes code readable.
- Very useful for next-state code: notice the initial statement to prevent accidental sequential code.

```
typedef enum logic [1:0] {
    Init,
    Run,
    Stop,
    Wait} state_t;          // name of the type

state_t state_q, state_d;   // instances of the type

always_comb begin
    state_d = state_q;      // init
    if (state_q == Run) begin
        state_d = Wait;
    end else if ((state_q == Stop) || state_q == Wait) begin
        if (cont == 1'b1) begin
            state_d = Run;
        end
    end else if (state_q == Init) begin
        state_d = Run;
    end
end
```

One can also use `case` statements in a process:

- `case` can help a lot: more readable code in next state calculations.
- `default` is important: it maps all parasitic states
- Initial assignment is still useful

```
typedef enum log [1:0] {
    Init,
    Run,
    Stop,
    Wait} state_t;          // name of the type

state_t state_q, state_d;   // instances of the type

always_comb begin
    state_d = state_q;      // init

    case(state_q)
        Run:            state_d = Wait;
        Stop, Wait:     begin
            if (cont == 1'b1) begin
                state_d = Run;
            end
        end
        Init:           state_d = Run;
        default:        state_d = state_q;
    endcase
end
```