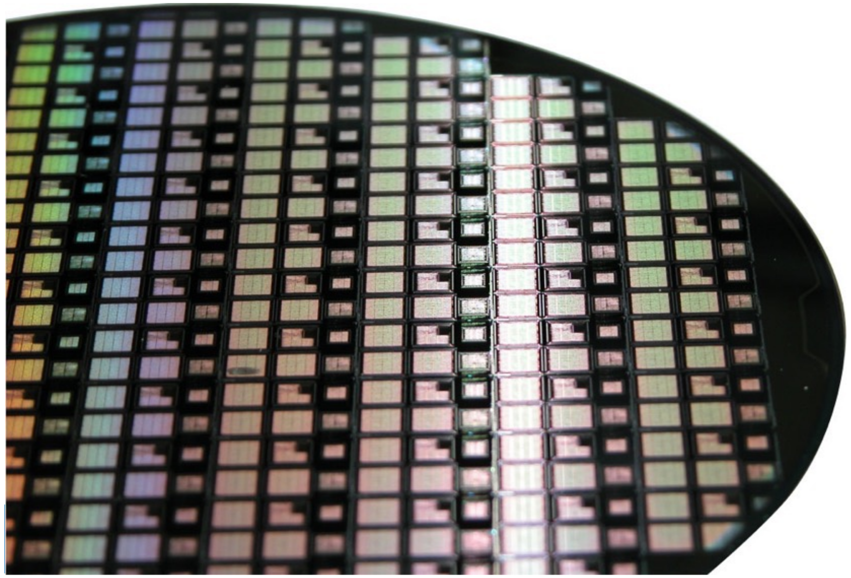
RTL: Refresher on SystemVerilog



19 February 2026

Luca Benini

Frank K. Gürkaynak

Goals for this lecture

- **Refresh our knowledge on SystemVerilog**
  - Module declarations
  - Hiereracy and instantiations
  - Combinational and Sequential statements
- **Some practical aspects to help our life**
  - Naming conventions we use at ETH Zürich to simplify our life
  - Additional good coding practices
- **Connect to next lecture on netlists**
  - Synthesis stage converts an HDL description (in SystemVerilog) to a netlist.

HDL was developed to simplify circuit schematics

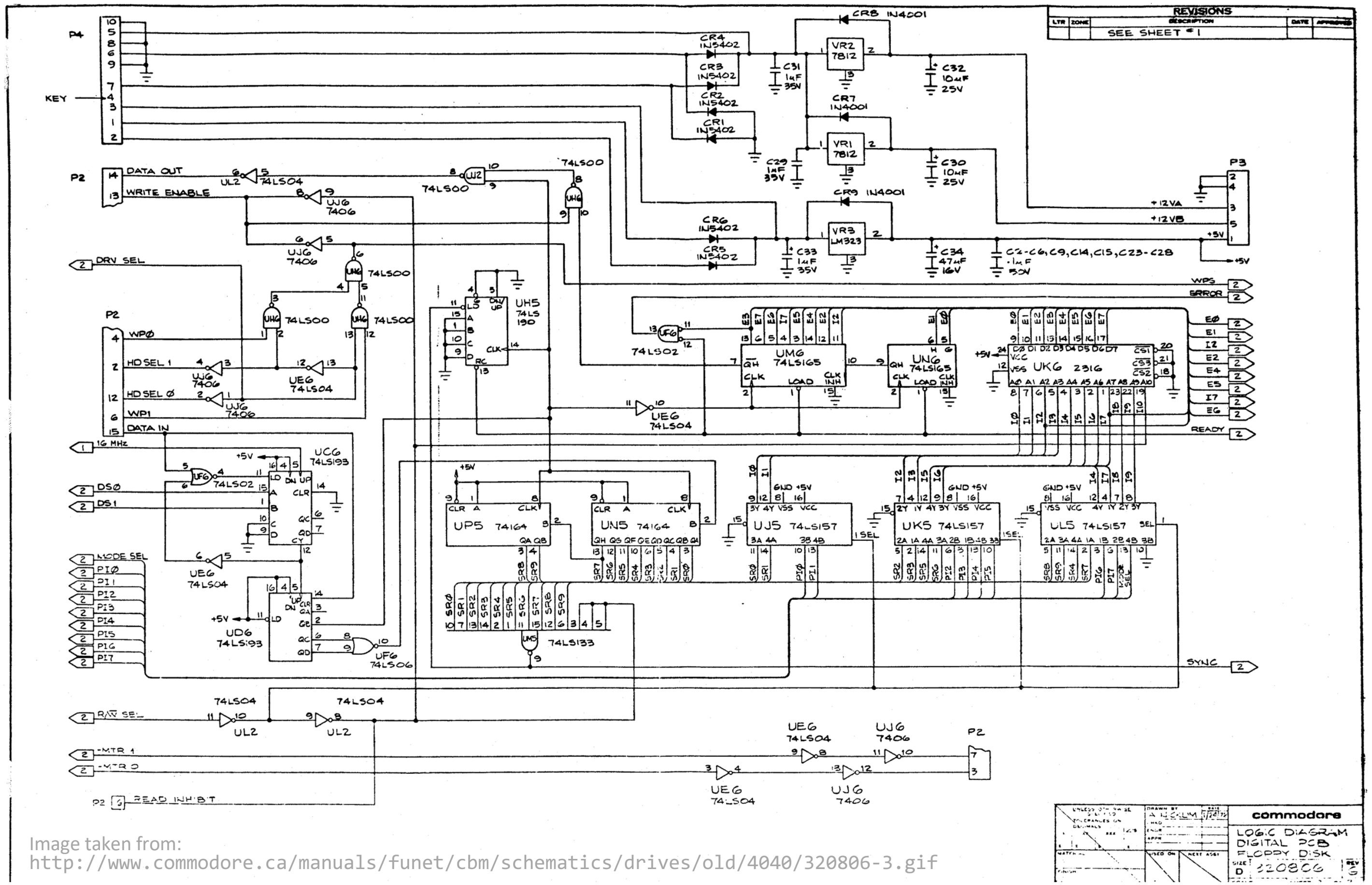


Image taken from:
http://www.commodore.ca/manuals/funet/cbm/schematics/drives/old/4040/320806-3.gif

Anatomy of a SystemVerilog file: module

- **Describe circuit**
  - Name
  - Connections
- **module defines**
  - Name of the block
  - (optional) parameters
  - Signal names, direction and types

We will discuss syntax details of these later in the lecture

```
module top #(
  parameter int Width = 16
) (
  input  logic              clk_i,
  input  logic              rst_ni,
  input  logic              mode_i,
  input  logic [Width-1:0] data_in_i,
  output logic [Width-1:0] result_o
);
// Declare signals to be used in the module
  logic  [Width-1:0] first, second;
  logic  [Width-1:0] combine_and, combine_or;

// instantiate two blocks with different names i_reg_1 and frank
  ffs #(.Width(Width)) i_reg_1 (
      .clk_i(clk_i), .rst_ni(rst_ni), .in_i(data_in_i), .out_o(first));
  ffs #(.Width(Width)) frank   (
      .clk_i(clk_i), .rst_ni(rst_ni), .in_i(first),     .out_o(second));

// combine outputs
  assign combine_and = first & second ;
  assign combine_or  = first | second ;

// assign result
  assign result_o = mode_i ? combine_and : combine_or ;
endmodule // top
```

HDL syntax will allow you to do unnecessary things !!!

- **In this lecture we will talk about good practices in using HDL**
  - The language syntax will allow you to do more / different things
  - Not all of these are useful / practical for digital design

EXAMPLES:

- **Always use one file per SystemVerilog module**
  - Technically you can add more than one module inside one file
- **Make sure that the file and the module are named the same**
  - This is good practice, the tools would let you define:
    module `a` inside `b.sv` and `a` inside `b.txt`
- **SystemVerilog is case sensitive**
  - The following are different `clk_ci, CLK_CI, Clk_Ci, cLK_cI...`

Anatomy of a SystemVerilog file: body

- **The body describes the function**
- **We declare local wires to use for internal connections**
  - Unlike a program, these do not cost resources directly
  - Declare as many as you need!

```
module top #(
  parameter int Width = 16
) (
  input  logic              clk_i,
  input  logic              rst_ni,
  input  logic              mode_i,
  input  logic [Width-1:0] data_in_i,
  output logic [Width-1:0] result_o
);
// Declare signals to be used in the module
  logic  [Width-1:0] first, second;
  logic  [Width-1:0] combine_and, combine_or;

// instantiate two blocks with different names i_reg_1 and frank
  ffs #(.Width(Width)) i_reg_1 (
  .clk_i(clk_i), .rst_ni(rst_ni), .in_i(data_in_i), .out_o(first));
  ffs #(.Width(Width)) frank   (
  .clk_i(clk_i), .rst_ni(rst_ni), .in_i(first),     .out_o(second));
// combine outputs
  assign combine_and = first & second ;
  assign combine_or  = first | second ;

// assign result
  assign result_o = mode_i ? combine_and : combine_or ;
endmodule // top
```

Anatomy of a SystemVerilog file: instantiation

You can include other modules in your module

- Allows hierarchy
- Two instances of the component ffs called i_reg_i and frank are instantiated
- These are described in another module
- You do not have to describe the interface of the components in this module.

```
module top #(
  parameter int Width = 16
) (
  input  logic              clk_i,
  input  logic              rst_ni,
  input  logic              mode_i,
  input  logic [Width-1:0] data_in_i,
  output logic [Width-1:0] result_o
);

// Declare signals to be used in the module
  logic  [Width-1:0] first, second;
  logic  [Width-1:0] combine_and, combine_or;

// instantiate two blocks with different names i_reg_1 and frank
  ffs #(.Width(Width)) i_reg_1 (
    .clk_i(clk_i), .rst_ni(rst_ni), .in_i(data_in_i), .out_o(first));
  ffs #(.Width(Width)) frank   (
    .clk_i(clk_i), .rst_ni(rst_ni), .in_i(first),     .out_o(second));

// combine outputs
  assign combine_and = first & second ;
  assign combine_or  = first | second ;

// assign result
  assign result_o = mode_i ? combine_and : combine_or ;

endmodule // top
```

Anatomy of a SystemVerilog file: instantiation

The module you instantiate (ffs) will be described separately

- You must make sure that the module description and the instantiation match.
- Same component can be instantiated multiple times. Here it was done twice: called i_reg_i and frank

```
module ffs #(
  parameter int Width=8
) (
  input  logic        clk_i,
  input  logic        rst_ni,
  input  logic [15:0] in_i,
  output logic [15:0] out_o
);
//  .. Rest of description for module 'ffs'
endmodule // ffs

module top (
  input  logic        clk_i,
  input  logic        rst_ni,
  input  logic        mode_i,
  input  logic [15:0] data_in_i,
  output logic [15:0] result_o
);

//  .. Rest of description

// instantiate two blocks with different names i_reg_1 and frank
  ffs #(.Width(Width)) i_reg_1 (
    .clk_i(clk_i), .rst_ni(rst_ni), .in_i(data_in_i), .out_o(first));
  ffs #(.Width(Width)) frank   (
    .clk_i(clk_i), .rst_ni(rst_ni), .in_i(first),     .out_o(second));

endmodule // top
```

A word of caution: Internet may not be your friend

- **There are many alternative ways of doing the same thing**
  - In our book, exercises and lectures we try to use a *consistent style*
  - We emphasize methods that have *proven to simplify your life*
  - A google search is not always going to give you good results
- **Consult trusted sources first…**
  - The textbook by Hubert Kaeslin "*Top Down Digital VLSI Design*"
  - Exercise notes and example files located under **`/home/vlsi1`**
  - The EDA wiki page: **`eda.ee.ethz.ch`** (an ETHZ internal web page)
- **… before you venture on the Internet**

More ways to do the same thing

```
module top (
  input  logic        clk_i,
  input  logic        rst_ni,
  input  logic        mode_i,
  input  logic [15:0] data_in_i,
  output logic [15:0] result_o
);


// instantiate one block
  ffs #(.Width(16)) i_reg_1 (
      .clk_i(clk_i),
      .rst_ni(rst_ni),
      .in_i(data_in_i),
      .out_o(first));

endmodule // top
```

```
module top (clk_i, rst_ni, mode_i,
            data_in_i, result_o);

// declare types of I/O later
  input  logic        clk_i;
  input  logic        rst_ni;
  input  logic        mode_i;
  input  logic [15:0] data_in_i;
  output logic [15:0] result_o;

// instantiate one block
  ffs #(.Width(16)) i_reg_1 (
      .clk_i(clk_i),
      .rst_ni(rst_ni),
      .in_i(data_in_i),
      .out_o(first));

endmodule // top
```

Possible:
But we do not use it

Two ways to instantiate components

```
module top (
  input  logic        clk_i,
  input  logic        rst_ni,
  input  logic        mode_i,
  input  logic [15:0] data_in_i,
  output logic [15:0] result_o
);


// instantiate one block
  ffs #(.Width(16)) i_reg_1 (
      .clk_i(clk_i),
      .rst_ni(rst_ni),
      .in_i(data_in_i),
      .out_o(first));

// pin assignments can be made in any order
// more robust

endmodule // top
```

```
module top (
  input  logic        clk_i,
  input  logic        rst_ni,
  input  logic        mode_i,
  input  logic [15:0] data_in_i,
  output logic [15:0] result_o
);


// instantiate one block
  ffs #(16) i_reg_1 (clk_i, rst_ni,
                    data_in_i, first);

// pin assignments are made in declaration
// order. This is very easy to make mistakes
// Especialy if you end up making changes
// to the original module.


endmodule // top
```

Dangerous !!
NEVER use it

There is a history to most *strange* things in SysVerilog

- **Verilog was initially developed just to describe schematics**
  - Other uses came later (in some cases much later)
  - Verilog was modified (became SystemVerilog) to support new ideas
- **Example:**
  - The original description of Verilog defined signals as `wire` and `reg`
    - These could only have the value `0` and `1`
  - But the name `reg` is very confusing. It makes it sound as if it is a **register**, a FF.
    - This was not the case
    - Signals that were assigned in a process (we will talk about it next lecture) had to be `reg`
    - All **registers** were in fact `reg` type, but not all `reg` declared signals were **registers**
  - Now we use the type `logic` (well most of the time)

Anatomy of a SystemVerilog file: comments

SystemVerilog uses C++ style comments, everything after // is comment.

- You can also use /* */ but this we do not recommend it.

```
module top #(
  parameter int Width = 16                // parameter Width, default value 16
) (
  input  logic             clk_i,         // clock input
  input  logic             rst_ni,        // reset active low
  input  logic             mode_i,        // mode select 1: combine and, 0: or
  input  logic [Width-1:0] data_in_i,     // data input
  output logic [Width-1:0] result_o       // data output
);

// Declare signals to be used in the module
  logic  [Width-1:0] first, second;
  logic  [Width-1:0] combine_and, combine_or;

// instantiate two blocks with different names i_reg_1 and frank
  ffs #(.Width(Width)) i_reg_1 (
      .clk_i(clk_i), .rst_ni(rst_ni), .in_i(data_in_i), .out_do(first));
  ffs #(.Width(Width)) frank   (
      .clk_i(clk_i), .rst_ni(rst_ni), .in_i(first),     .out_do(second));

// combine outputs
  assign combine_and = first & second ;
  assign combine_or  = first | second ;

/* Possible also to use this style of comment
   but we will prefer the C++ style */
  assign result_o = mode_i ? combine_and : combine_or ;
endmodule // top
```

Structural HDL describes hierarchical connections

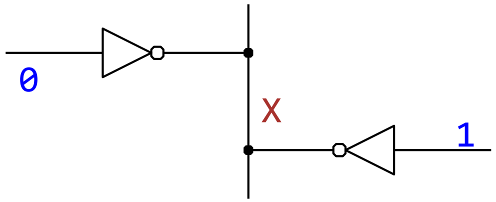
- **What we have looked at now is called structural HDL**
  - The circuit includes other circuits which are interconnected
  - There is no other information than interconnections and instantiations
    - No function – or just simple Boolean functions
- **A netlist is a structural HDL that instantiates only library components**
  - The library components are simple pre-designed circuits with physical properties
  - Once we have a circuit netlist, it has a corresponding physical form, it can be manufactured.
  - Once we are done with this part, we will discuss how we map a netlist into a chip!
- **But there are other types of HDL descriptions**

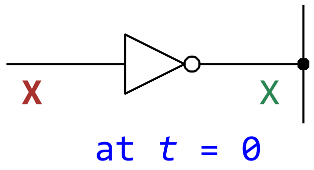
Different HDL styles

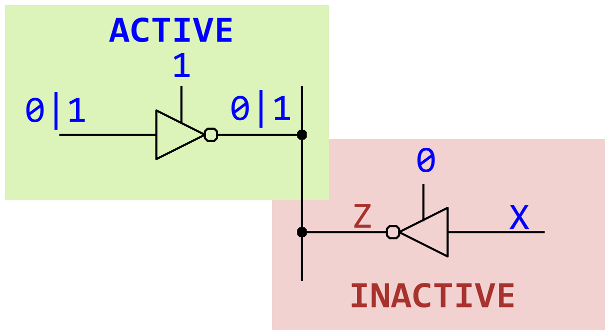
- **Structural HDL**
  - Interconnections between components
- **Behavioral HDL, RTL description**
  - Describes the functionality of the circuit
- **Timing Information / Model**
  - Contains information on different timing properties of signals of a circuit
  - Only used for simulation.
- **Test environment (testbenches), verification**
  - Support environment to verify circuits we have designed. Allows inputs to be generated for inputs, and checks the outputs.
  - Is not meant for actual circuit design, only used for simulation.

It is not only logic-1 and logic-0 we need…

- **What happens when a wire is driven by two sources at the same time?**
  - Drive conflict: ’**X**’
- **What is the initial value of a wire ?**
  - We do not know really => ‘**X**’
- **What if we do not care what the input is ?**
  - ‘**X**’ can also be used as don't care.
- **What if we do not drive the output**
  - The output has three states, 0,1 and **inactive**
  - This inactive logic state is denoted as ‘**Z**’







Figures taken from Hubert Kaeslin, “Top Down Digital VLSI Design: from Architectures to Gate-Level Circuits and FPGAs”

Main data type we use in SystemVerilog is logic

- If we are defining wires in an electrical circuit we will use logic
- It can be used to define: inputs, outputs, signals and constants
- If a constant is only used in one module, declare as `localparam`
- We use the `assign` statement to connect these signals/constants

```
module top (
  input  logic data_in_i,
  output logic results_ready_o
);

  logic  first, second;                   // if the type is the same multiple definitions possible
  logic  third;
  localparam ZERO         = 1'b0 ;        // constant definition

  assign first            = data_in_i ;   // this connects first to data_in_i
  assign results_ready_o  = third ;       // connections represent electrical wires
  assign third            = first ;       // the order you define them is not important
  assign second           = ZERO ;        // assign a constant value to a wire
```

If we need more than one bit, logic can be an array

- It can be as large as you want, you specify the range
- We define the array from MSB to LSB (like we write numbers)
- We call this (standard) format packed
- Can also have multiple dimensions `logic [15:0][7:0] varname`

```
logic    [7:0] eight_bit_bus ;
logic   [15:0] big_bus ;
logic    [0:0] tiny_bus ;
logic          first, second, third ;

assign eight_bit_bus = 8'b0000_0000;     // assigning a constant
assign big_bus[15:8] = eight_bit_bus;    // partial assignment
assign first         = big_bus[3];       // picking out single bits
                                         // more examples of combining
                                         // will come in a few slides
```

How to access/assign part of a vector

- **Use the { } to combine vectors together**

```
// You can assign partial busses
  logic [15:0] longbus;
  logic  [7:0] shortbus;
  logic        first, second;
  logic  [3:0] x, y, z;

  assign shortbus = longbus[12:5];
  assign first    = shortbus[1];
  assign longbus[2] = second;

// Concatenating is by {}
  assign x = {a[2], a[1], a[0], a[0]};

// Possible to define multiple copies
  assign y = {a[0], a[0], a[0], a[0]};
  assign z = { 4{a[0]} };
```

How to express numbers in SystemVerilog

N' Bxx

8' b0000_0001

- **(N) Number of bits**
  - Expresses how many bits will be used to store the value
- **(B) Base**
  - Can be **b** (binary), **h** (hexadecimal), **d** (decimal), **o** (octal)
- **(xx) Number**
  - The value expressed in base
  - Apart from numbers it can also have **X** and **Z** as values.
  - Underscore _ can be used to improve readability

Examples for numbers

- **SystemVerilog is not strongly typed**
  - Source of many many errors
  - You must make sure that left- and right-hand side of assignments have same size

```
localparam CONST_A = 4'b1001;    // stores  4 bit value 1001
localparam CONST_B = 8'b1001;    // stores  8 bit value 0000 1001
localparam CONST_C = 8'b0;       // stores  8 bit value 0000 0000
localparam CONST_D = 12'hFA7;    // stores 12 bit value 1111 1010 0111
localparam CONST_E = 4'hFA7;     // stores  4 bit value 0111
localparam CONST_F = 4'bx0XZ;    // stores  4 bit value X0XZ
localparam CONST_G = 4'b00_11;   // stores  4 bit value 0011
localparam CONST_H = 8'd42;      // stores  8 bit value 0010 1010 (dec 42)

assign some_signal = 16'b0;      // 16bit some_signal is set to all zeroes
```

Naming Conventions used at IIS

- **Basic idea: Add a _ and a regular suffix to identify special signals**

  `data_i, result_o, clk_c, grant_n, rst_ni, tristate_io`
- **Use meaningful names with** `lower_snake_case`

inputs	`_i`	Signals that are declared input in the module
outputs	`_o`	Signals that are declared output in the module
types	`_t`	Type definitions
reset	`_r`	Asynchronous reset used for flip-flops and latches
clock	`_c`	Clock signals for flip-flops and latches
active low	`_n`	Signals that are active when they have the value `0`

How to implement basic logic functions

- **So far we talked only about connections**
- **SystemVerilog allows you to define simple Boolean functions directly**
  - `not (~), and (&), or (|), xor (^)` are supported.
  - In theory, if you can define a NAND, you can define any Boolean function.
- **We will discuss, arithmetic functions (+ - * / % << >>) a bit later**

```
assign y0 =  ~a;         // NOT
assign y1 =   a & b;     // AND
assign y2 =   a | b;     // OR
assign y3 =   a ^ b;     // XOR
assign y4 = ~(a & b);    // NAND
assign y5 = ~(a | b);    // NOR
assign y6 =  (a & b | ~(a & ~b)); // more complex
```

Simple Multiplexers

- **Multiplexers are very common building blocks in digital design**
- **In system Verilog we use the ternary operator**

```
assign value = condition ? true : false ;
```

```
module mux2(
  input  logic [3:0] data0_i, data1_i,
  input  logic       select_i,
  output logic [3:0] result_o
);

 assign result_o = select_i ? data1_i : data0_i;
   // if (select) then result = data1 else result = data0;

endmodule
```

Multi-level multiplexers can also be defined

- **You can use ternary operators hierarchically**

```
module mux4(
  input logic  [3:0] data0_i, data1_i, data2_i, data3_i,
  input logic  [1:0] select_i,
  output logic [3:0] result_o
);
 assign result_o = (select_i == 2'b11) ? data3_i :
                   (select_i == 2'b10) ? data2_i :
                   (select_i == 2'b01) ? data1_i :
                   data_0;
// if      (select = "11" ) then result= data3
// else if (select = "10" ) then result= data2
// else if (select = "01" ) then result= data1
// else                          result= data0
endmodule
```

Devil is in the details: bitwise vs logical operators

- **Bitwise operators (~ & | ^ =)**
  - Will work on all bits of a vector and result in a vector of same size
  - Used to implement simple Boolean functions
- **Logical operators (! && || == !=)**
  - Will generate a true or false result
  - Used as a condition in (for example) ternary operators
- **Tricky part**
  - Technically if you use a single bit operator both bitwise and logical operators will give identical results.
  - Don't do this!

Sequential circuits == Combinational circuit + State

- **Largest part of a sequential circuit is actually combinational**
  - The only additional thing we need to learn is to store the state
  - Defining flip-flops (latches), registers should do the trick
- **Sequential circuits divide the operation into time slots**
  - At every time slot inputs (if there are any) are taken
  - Present state and inputs are used to calculate the next state
  - The next state is saved in the flip-flops (registers)
- **How fast we can finish the operation will tell us how fast we can work**
  - The clock signal is used to tell when to move from one state to the next state
  - I.e. a 2 GHz clock has time steps of 500ps. One operation is finished every 500ps.
  - The actual work within a time slot is done by a combinational circuit.

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 evaluated
- **Process body is evaluated sequentially**
  - Unlike normal descriptions in Verilog
- **Multiple processes will run in parallel**
  - Only the process body runs sequentially
- **This description is not physical**
  - We describe the behavior of the circuit
  - This will be then translated into hardware

```
module example (
//…
);
  logic state_q, state_d, a_i, cnt;

  always @ (state_q, a_i, cnt) begin
    state_d = 1’b0;
    if (cnt == 1’b1) begin
      state_d = state_q;
    end else begin
      state_d = a_i;
    end
  end // always

// other parts of the description
endmodule
```

Using begin .. end for longer statements

- **It is possible to use multiple statements in your code**
  - Use `begin .. end` for long statements
  - You can use multiple nested statements
  - Indent by 2 chars when you do so
- **If you have only a single statement**
  - Verilog allows you to skip `begin .. end` if there is a single statement.
  - Not recommended
- **Don't start line with `begin`**
  - This is a style decision we use at ETH

```
logic state_q, state_d, a_i, cnt;

  always @ (state_q, a_i, cnt) begin
    state_d = 1'b0;
    if (cnt == 1'b1) begin
      state_d = state_q;
    end else begin
      state_d = a_i;
    end
  end // always


// Possible, but not recommended
// always @ (state_q, a_i, cnt) begin
//     state_d = 1'b0;
//     if (cnt == 1'b1) state_d=state_q;
//     else state_d = a_i;
//   end // always

endmodule
```

Multiple processes can be in parallel, but be careful!!

- **You can not assign to the same signal in multiple processes!!**
  - Basically one process determines the value of a signal. (single driver)
- **A signal assigned in a process can not be assigned to by a concurrent statement.**
- **Only one of the three statements in the example can be used to determine the value of `state_d`**
  - Does not matter which one!

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

logic a, state_d;

  always @ (a) begin
    state_d = ~a;
  end

  always @ (a) begin
    state_d <= a;
  end

  assign state_d = 1’b0;

endmodule
```

SystemVerilog introduced intent to `always` statements

- **There are three different `always` statements in SystemVerilog**
  - Combinational circuit: `always_comb`
  - Latches: `always_latch`
  - Flip-flops: `always_ff`
- **The tools will know what you want, you state it with the flavor.**
  - There is no '*I deliberately forgot a case to make the description sequential*'
- **`always_comb` has no sensitivity list**
  - The process will be triggered when signals change,
  - Replaces `always @ (*)`
- **No need to use the regular `always` with SystemVerilog**
  - It will work, but we advise against it.

The proper way to use always statements

- **Combinational: `always_comb`**
  - No sensitivity list needed
  - We will use blocking (**=**) statements
- **Latches: always_latch**
  - We will rarely use latches
  - This example without reset
  - We will use unblocking (**<=**) statements
- **Flip-flops: always_ff**
  - Our main state holding element
  - This example without reset
  - We will use unblocking (**<=**) statements

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

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

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

We will use blocking assignments within `always_comb`

- **and unblocking assignments within `always_ff` and `always_latch`**

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

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

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

- **These are from our SystemVerilog style guide**
  - Proven to make your life easier

Remember `always_ff` will describe a physical FF

- **The FFs will be available inside the FPGA or as a library component for ASIC**
  - Can be rising/falling edge triggered
  - Have a reset/set value
  - Some can have an enable (not updated on all clock cycles)
- **Do not add more functionality**
  - There are more structured ways of adding complex sequential behavior.

```
module example (
  input logic clk_ci,
  input logic rst_ni,
  //..
);
  logic [3:0] state_q, state_d;
  logic       en;

  always @ (posedge clk_ci,
            negedge rst_ni) begin
    if (rst_ni==1'b0) begin
      state_q <= 4'b0110;
    end else if (en==1'b1) begin
      state_q <= state_d;
    end
  end

endmodule
```

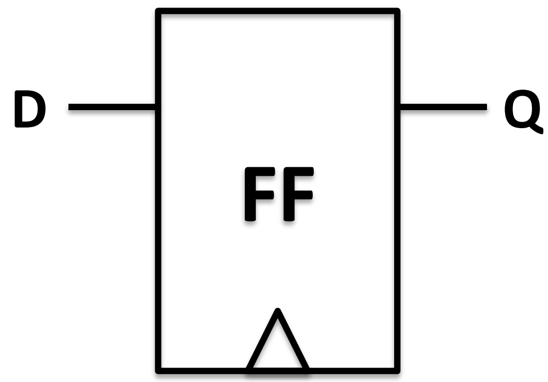
Keep FFs simple, do not add more functionality

- **All previous descriptions map to known hardware primitives.**
  - Additional code in this process will make mapping difficult
- **A process for a FF should:**
  - Only assign the next state to present state when clock comes
  - Have a reset condition
  - And at most an enable condition
  - Verilog will allow you to do more, but you will make more errors, and your circuit will not be better

```
always_ff @ (posedge clk_ci,
             negedge rst_ni) begin
  if rst_rbi == '0' begin
    result_q <= 5'b0;
  end else if (cond_i==1'b1) begin
    if (another_i == 1'b0) begin
      result_q  <= a_i ^ last_mul;
    end else begin
      if (something == 1'b0) begin
        result_q <= {b[3:0], b[0]};
      end else begin
        result_q <= x_i & result_d;
      end
    end
  end else if  (something == 1'b1) begin
      result_q <= result_d;
  end
end
```

Let us expand our naming conventions a bit

- **We will add _d to signals that will be the NEXT state signals of a FF**
  - Inspired by the input of the D type FF
- **We will add _q to signals that will be the PRESENT state signals of a FF**
  - Again inspired by the output of FFs
- **The _q signal will be assigned in an `always_ff` process**
- **The _d signal will be assigned in an `always_comb` process**



Basic SystemVerilog template for a FSM

- **Next state logic**
  - Calculates the next state `state_d` depending on inputs and present state `state_q`
- **State holding element**
  - One register with reset
- **Output calculation**
  - **Moore**: only dependent on state
  - **Mealy**: both on state and inputs
- **Only one process is mandatory**
  - Use whatever is simpler

```
// Next State
  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;
```

Describing combinational circuits in `always` processes

- **We already saw that you can use a process for combinational circuits**
  - There is an overhead of defining the process structure
  - Mistakes in the sensitivity list can cause major headache
  - It should not be first choice to implement combinational circuits
- **There are many cases where processes simplify life**
  - Especially in control (FSM) next state calculations
    - "*when in this state, if this happens go to this state, if not go to this other state*"
  - Operations on vectors with variable size
    - i.e. "*AND all bits of a 32 bit vector*"
  - Usually these are not 'critical' functions
    - They have many cases, and generate a lot of code.
    - Most code == most mistakes

Enumerated types can be very handy for states

- **In an enumerated type you provide a list of names**
  - These will internally map to a binary code, it is up to the tool to decide.
- **Can test and assign these types**
  - Makes code more readable
  - Unless you use names like `S1,S2..`
- **Very useful for next state code**
  - Notice the initial statement to prevent accidental seq. behavior
  - By default state stays same

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

state_t state_q, state_d; // instances of 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
```

You can also use case statements in a process

- **case can help a lot**
  - More readable code in next state calculations.
  - Multiple cases can be combined with ,
- **default is important**
  - it maps all parasitic states
  - default of case statement
- **Init assignment still useful**
  - The if cont does not have an else statement !!

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

state_t state_q, state_d; // instances of 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
```

This covers the basics of SystemVerilog

- **There are some more tricks left**
  - Did not cover all statements
  - This is a refresher, refer to VLSI1 lectures for more information
- **You can basically design any digital circuit now**
  - Before you start coding in HDL you need to have
    - a clean architecture
    - a good block diagram that describes this architecture, complete with signal names
    - a state transition diagram for the FSM
- **Starting point of back-end design is a netlist (next lecture) in Verilog**
  - Even if you do not plan to design, you will need to deal with (System)Verilog files

RTL

Register Transfer Level

Register Transfer Level (RTL) Design

- **RTL is a generic way of defining 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 datapath circuits …**
  - They process data and do the main work
- **… and Finite State machines**
  - Generate control signals for the datapath, define what operation will be done.
- **Most people do not realize the distinction, makes life easier**

Your typical digital circuit will have two parts

Control (FSM)

- **Generates control signals for datapath**
  - Reacts to inputs
  - Decides what will happen in next cycle
  - Generally a small part of the circuit
- **Operations used**
  - Generally `if .. else` type of operations
  - Should not have datapath components
- **Uses a register to determine the current state**

Data processing (Datapath)

- **Does the actual calculation**
  - Determines 90% of the area
  - Should ideally determine operation speed
- **Operations used**
  - Mostly arithmetic and logic operations
  - Multiplexers
- **Registers are used to store (partially) processed data**

Try to keep these two operations separate

RTL uses hierarchy to manage complexity

- **Hierarchy is used to divide complex circuits into smaller sub-circuits**
  - Divide and conquer is our main tool to manage complexity
- **There might be different reasons for using hierarchy**
  - Design re-use: if the same module is instantiated many times (reduces effort)
  - Dividing work: different teams can work on it in parallel
  - Managing EDA flow: allowing parts of the circuit to be treated differently
- **Signs your module is not helping you**
  - The module definition is longer than the module body
  - You need to connect too many signals between two modules
  - You need to update the module definition frequently as you develop

Tip: datapath and control should be in the same module

- **Common mistake is to make two separate Verilog modules**
- **The control and datapath interact closely**
  - Throughout the lifetime of the circuit updates are very likely
    - Will add/modify/remove control signals
  - Simpler to debug, problem will be the interaction between control and datapath
- **Consider what you need to do to add a signal**
  - Add a port to the module definition of FSM as an output
  - Add a port to the module definition of the Datapath as an input
  - Change the instantiation at the higher level to contain the new ports
  - Add a wire to connect them
- **Use hierarchy where it makes your life simpler**

What to remember

- **HDL helps us essentially draw circuit schematics**
  - Describes the connections of a module
  - Allows hierarchy
  - Functionality inside the body
  - Allows you to control the hardware tightly (structural) or loosely (behavioral)
- **RTL is an expression of the design idea in HDL**
  - Divides operations into distinct steps (clock cycles)
  - Keeps the state in registers
  - Transforms (transfers) the present state to the next using combinatinal circuits