VLSI front-end flow

Phu Nguyen

January 2024

1 Introduction

In the VLSI design field, there are several steps to developing a chip prior to manufacturing. Below is the flow chart that describes the VLSI design flow:

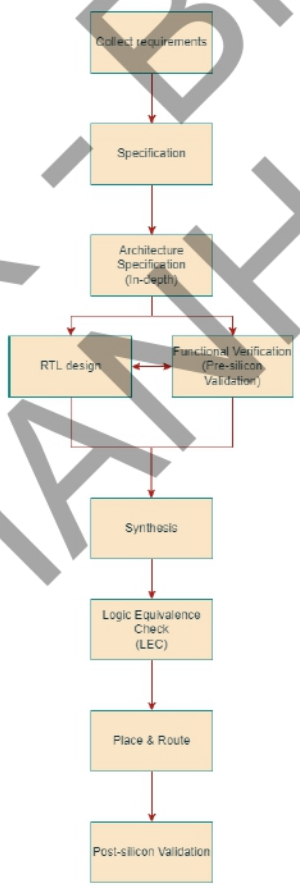

Figure 1: VLSI design flow

In the below sections, we will focus on the specification, RTL design, and functional verification steps.

2 Specification

This is the first stage of the VLSI flow. Normally, the design department will get the detailed requirements from the customer, including functionality, performance, power consumption, etc. Then the RTL designer will start making the design specification, and then the designer will create the RTL code.

The Specification will contain:

- Introduction about the design.
- Functional description (clock, reset, block diagram,...).
- Interface (input, output of the design).
- Configuration.

Here is an example of a simple 4-bit counter specification:

The 4-bit counter samples at the positive edge of the clock signal have an asynchronously active-low reset. When rst_n is asserted, the output is 0. It can either count up or count down based on the user's needs.

Signal	Width	Type	Description
clk	1	input	Clock signal
rst_n	1	input	Negative edge reset. If rst_n = 0, output will be set to 0. Else, it will start the normal operation.
sel	1	input	Mode selection signal. If sel = 1, the design will start counting up else, it will start counting down from the current output value.
out	4	output	Result of the counter.

Table 1: Port definitions

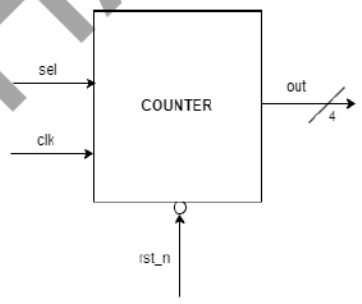

Figure 2: 4-bit counter diagram

As an RTL design engineer, you will start to design the component based on the specification. On the verification side, the DV engineer will create a detailed verification plan and environment in order to verify the RTL design.

3 RTL design

After receiving the specifications, RTL designers will model the functional block using the HDL language (Verilog, VHDL, System Verilog). Synthesizable constructs are used so that the RTL can map to the actual gates via "synthesis tool" (DC, Genus,...).

3.1 Verilog HDL

The Hardware Description Language (HDL) is widely used for designing digital components. Verilog and System Verilog

3.1.1 Verilog Syntax

Below is some Verilog syntax that is frequently used when writing the code.

- All lines in Verilog code should be terminated by a semi-colon ";".

- There are 2 ways to write comments in Verilog:

+ Use "//" to comment a single line. Ex: //Verilog comment

+ A multiple-line comment starts with "/*" and ends with "*/" and cannot be nested.

- In Verilog, number can be represented in *binary*, *octal*, *hexadecimal*, *decimal*:

$$[size]'[format][number] \tag{1}$$

size is the bit-width of the number; *format* can be *b* (binary), *o* (octal), *h* (hex), *d* (dec); *number* is the value of the number.

Example: 3'b011 (size is 3, format is binary, value is 3).

3.1.2 Module

```
module <module name> #(<param list>) (<port list>);
  <Declarations>
  <Instantiations>
  <Data flow statements>
  <Behavioral blocks>
  <task and functions>
endmodule
```

Figure 3: Module Declaration

To create a digital component using Verilog HDL, we must declare the component inside the *module* and *endmodule* keyword.

In the figure 3, the *param list* is the parameter list which will be reused inside the module. *port list* is the list of *input* and *output* ports.

The *Declarations* is used to declare the *wire*, *reg* or *multi − dimension* array. Example:

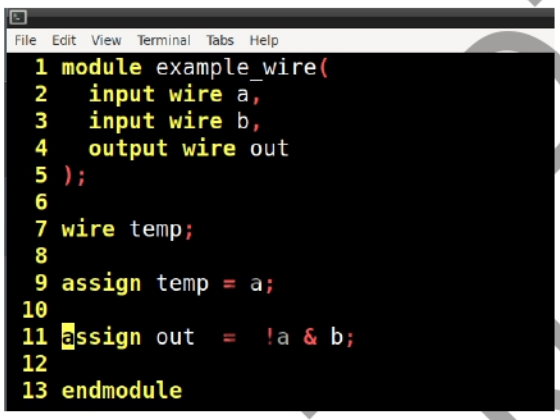

Figure 4: Example for wire inside a module

The *wire temp* is the internal wire inside the module, it is used for creating logic operations or connections.

The *reg* is a variable used to model the "hold" value component (register, flip-flop,...). the *reg* type variable must be used inside the *always block*.

```
1 module example_reg(
2   input wire a,
3   input wire b,
4   output reg out
5 );
6
7   always @(*) begin
8     out = !a & b;
9   end
10
11 endmodule
```

Figure 5: Example for reg inside a module

Also, we may need to instantiate other modules inside one module (*Instantiation*).

The *Data flow statements* is normally used to describe the operations that do not depend on the clock signal (Use *assisgn* keyword). For example:

$$assign\ out = !a\ \&\ b; \tag{2}$$

```
my_mod #(.W(1), .N(4)) U1 (.in1(a), .in2(b),
                           .out(c));
```

Figure 6: Instantiate

The *Behavioral blocks* in Verilog contain procedural statements, that control the simulation and manipulate variables of the data types. There are 2 commonly used behavioral blocks: *always block* and *initial blocks*. The second one is a non-synthesizable construct.

```
always @(<sensitivity list>) begin
  <statements>;
end


initial begin
  <statements>;
end
```

Figure 7: Behaviour Block

The *task* and *function* are normally used in testbench to simulate the design.

```
File  Edit  View  Terminal  Tabs  Help
 1 module counter #(parameter WIDTH=4)(
 2   input wire clk,
 3   input wire rst_n,
 4   input wire sel,
 5   output reg [WIDTH-1:0] out
 6 );
 7
 8   always @(posedge clk or negedge rst_n) begin
 9     if (!rst_n) begin
10       out <= 0;
11     end
12     else begin
13       if (sel == 0) begin
14         out <= out - 1;
15       end
16       else begin
17         out <= out + 1;
18       end
19     end
20   end
21 endmodule
```

Figure 8: Verilog Code - 4bit counter

Figure 8 is a simple verilog module for the 4-bit counter using the *always* behaviour block.

3.1.3 Control statement

There are several conditional constructs in Verilog, these constructs are written inside the *behaviour block*:

- if...else block:

```
if (<condition>) begin
    <statements> ;
end
else if (<condition>) begin
    <statements> ;
end
else begin
    <statements> ;
end
```

Figure 9: If - else statement

+ The statement occurs if the expressions controlling the if statement evaluate to true. (True: 1 or non-zero value; False: 0 or ambiguous (X)).

- Case block:

+ The statement occurs when the expressions match the corresponding alternative.

```
case (<expression)
    <alternative 1> : <statement 1> ;
    <alternative 2> : <statement 2> ;
    ...
    default: <default statement> ;
endcase
```

Figure 10: Case statement

- There are some other control constructs like *for* and *while* have the same operation as *C* code.

3.1.4 Blocking & Non-blocking assignment

The blocking assignment ("=") is used in the assignment block (*assign*) and *always* block (not contain the clock signal).

The non-blocking assignment ("<=") is used in an *always* block (contains the clock signal). You can refer to the Figure 8.

3.2 Verilog Operators

The figures 11 and 12 are the basic Verilog Operators that can be used to describe logical behavior.

Operator types	Operators	Description
Relational	<, >	A </> B: A less/greater than B ? Returns 1 if true.
	<=, >=	A <=/>= B: A less/greater or equal B ? Returns 1 if true.
Arithmetic	*	A * B: A is multiplied by B.
	/	A / B: A is divided by B.
	+	A + B: A is added by B.
	-	A - B: A is subtracted by B.
	%	A % B: A mod(B).
Logical	&&	A && B: Both A and B true ? Returns 1 if true.
	||	A || B: A or B true ? Returns 1 if true.
	!	!A: A is not true ? Returns 1 if not true.

Figure 11: Logic Operator

Operator types	Operators	Description
Equality and Identity	==, !=	A ==/!= B: A equal/not equal to B ? Returns 1 if true.
	===, !==	A ===/!== B: A case-equal/not case-equal to B ? Returns 1 if true.
Bitwise and Reduction	&	&A: AND all bits of A. A & B: AND each bit of A with each bit of B.
	|	|A: OR all bits of A. A | B: OR each bit of A with each bit of B.
	^	^A: XOR all bits of A. A ^ B: XOR each bit of A with each bit of B.
	~&, ~|, ~^	Same as &, |, ^ but inversed (~) (NAND, NOR, XNOR). ~A: inverse all bits of A.
Shift and other operator	<<, >>	A <</>> B: A shift left/right by B-bit.
	<<<, >>>	A <<</>>> B: A arithmetic shift left/right by B-bit.
	?:	C ? A : B: If C true, returns A. Else, returns B.
	{}	{A, B}: returns the concatenation of A and B.
	{{}}	{3{A}}: replicate A 3 times. Similar to {A, A, A}.

Figure 12: Logic Operator

4 Functional Verification

Verification is a crucial step in the development process of any product or system, particularly in fields such as software engineering, hardware design, and manufacturing. It involves the process of evaluating whether a product, system, or component meets its specified requirements and functions correctly.

The main goal of verification is to ensure that the product or system behaves as intended and fulfills the expectations outlined in its design specifications. By performing verification, potential defects, errors, or deviations from the desired behavior can be identified and corrected early in the development lifecycle, reducing the likelihood of costly issues arising later during testing or deployment.

In real chip design, the verification effort needed to fully verify the design is huge. Normally, one design engineer needs three verification engineers.

4.1 Verification Plan

After reviewing clearly the specification, we will start to develop the verification plan.

Here is the basic format for the verification plan:

Section	Item	Description	Testcase name	Owner	Status
1	Reset	When rst_n is asserted, the output is 0, when it de-asserts, the output start to count up each positive edge of clock	cnt_rst_test	Phu	PASS
2	Max count	When output is 4'b1111 and the counter is counting up, the next positive edge clock, output will be 4'b0000	cnt_max_test	Phu	RUNNING

Table 2: Basic Verification Plan

The verification plan contains a detailed description of what we want to test on the DUT (Design Under Test). The verification engineer will start creating the verification plan along with the RTL designer. This plan was created based on the specifications.

In the table 2, *Status* can be PASS (the test passed), FAIL (test failed), RUNNING (testcase is running) or PENDING (Test is pending or not started yet).

4.2 Building verification environment

After finishing the verification plan, we will start to develop the environment, including the verification component (normally written in System Verilog-UUVM) and testbench. In this section, we will focus on the testbench.

4.2.1 Task & Function

Task & function are commonly used inside testbench to control the behaviour of the DUT.

- Task: A task need not have a set of arguments in the port list, we can keep it empty. Note that the task can contain the timing argument # while the function cannot (because function costs no time to execute).

```
//Task with port list
task [name] (input [port_list], output [port_list]);
    begin
        [statements];
    end
endtask
```

```
//Empty task
task [name];
    begin
        [statements];
    end
endtask
```

- Function: Function must have the port list.

```
function [return type] [name] (input [port_list]);
    begin
        [statements];
    end
endfunction
```

Here are some notes about tasks and functions that we need to remember:
- Function is not time-consuming, while task is time-consuming.
- A function cannot execute a task. It can only execute other functions.
- A task can execute both other functions and tasks.
- A function must have an input list and cannot have an output port list.
- A task can either have an input-ooutput port list or be empty.

4.2.2 Testbench

In the Testbench, we need to declare the variable that control the DUT operation (clk, rst_n, sel), wire out to observe the output and instantiate the DUT.

The First *initial* block is used as a clock generator.

The Second *initial* block is used to run the test, we normally declare the task and function inside this block.

The Third *initial* block is used to create the wave file which can be viewed using Verdi, DVE, Simvision, ...

The figure 13 is the testbench of the 4-bit counter.

- Delay (#) is used to specify the delay time units before a statement that is executed during the simulation. Example:

$$\#[number]\ y = 1; \tag{3}$$

- The $finish keyword is used to end the simulation. If $finish is not declared, the simulation will hang.

4.3 Verification

We will start verifying the DUT when we have a verification plan and environment. There are several EDA tools that can be used to run simulations, like VCS, Xcelium, and Questa Sim.

```
module testbench;
  parameter WIDTH=4;
  reg clk;
  reg rst_n;
  reg sel;
  wire [WIDTH-1:0] out;


  counter u_counter (
    .clk (clk),
    .rst_n (rst_n),
    .sel (sel),
    .out (out)
);

  initial begin
    #0;
    clk = 0;
    forever #5 clk = ~clk;
  end

  initial begin
    run_test();
  end

  initial begin
    $shm_open("tb.shm");
    $shm_probe("AS");
  end

  task run_test();
    #0;
    rst_n = 0;
    repeat (5) @(posedge clk);
    rst_n = 1;
    sel   = 1;

    #500;

    $finish;
  endtask
endmodule
```

Figure 13: Basic Testbench

5 Labs

5.1 Running Xcelium simulator

Working directory tree:

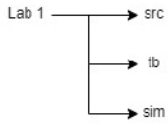

Figure 14: Directory Structure

The *src* directory contains the RTL source code, the *tb* directory contains the testbench and the *sim* directory contains Makefile, list file and simulation result.

Here is the Makefile contents:

```
1 sim:
2   xrun +xm64bit -sv +access+rw -timescale 1ns/10ps -f list.f -l sim.log
3 wave:
4   simvision -64bit ./*.shm &
```

Figure 15: Makefile

Note that the *list.f* file contains the link to all the RTL and testbench source code that will be simulated by Xcelium.

After finishing the RTL and testbench code and including all the codes inside the list file, we go to the *sim* directory and run the following commands *make sim* and *make wave*

The *make sim* will compile our design and testbench code, it will detect some errors inside the module. After the compilation is completed, xcelium will execute the simulation and dump the waveform.

make wave will open the simvision to observe the waveform as the figure 16

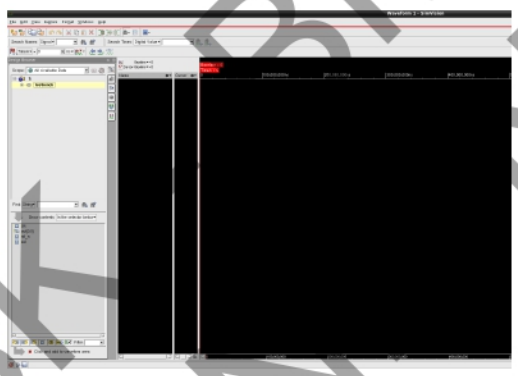

Figure 16: Simvision GUI 1

After invoking the simvision (figure 17, we click on the *testbench*, it will show us the DUT and all the signals. At step 2, click on those signals to observe them on the waveform.

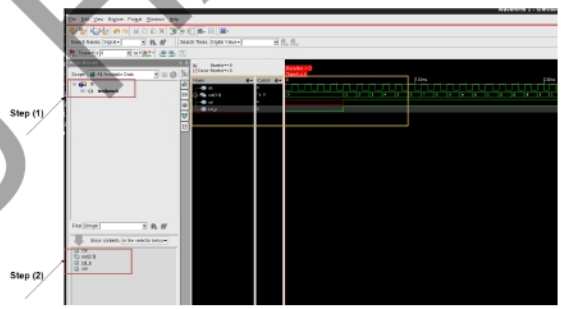

Figure 17: Simvision GUI 2

5.2 Lab 1: 4-bit Counter

Requirements:

- Design a 4-bit counter, sampled at the positive edge of the clock signal, with an asynchronously active-low reset. When rst_n is asserted, the output is 0. It can either count up or count down based on the user's needs.

Signal	Width	Type	Description
clk	1	input	Clock signal
rst_n	1	input	Negative edge reset. If rst_n = 0, output will be set to 0. Else, it will start the normal operation.
sel	1	input	Mode selection signal. If sel = 1, the design will start counting up else, it will start counting down from the current output value.
out	4	output	Result of the counter.

Table 3: Port definitions

2. Create a simple verification plan to test basic operation of the counter.
3. Simulate and fill in the verification plan.

5.3 Lab 2: ALU

Requirements:

1. Design a basic ALU, sample at positive edge of clock, asynchronous reset, when reset is asserted, result is 0.

The ALU can perform these operations:

- Addition, Subtraction.
- Bitwise AND, OR, XOR, NOT.
- Logical shift left, logical shift right.

Signal	Width	Type	Description
clk	1	input	Clock signal
rst_n	1	input	Reset signal. If rst_n = 0, result and carry will be 0 else the design will work normally based on the input arguments
a	4	input	First argument
b	4	input	Second argument
op	3	input	Select the operation
result	4	output	Result of ALU
carry	1	output	Carry flag

Table 4: Port definitions

Below table is the detailed function of the ALU design.

Operation (op)	**Description**
3'b000	{carry, result} = a + b
3'b001	{carry, result} = a - b
3'b010	carry = 0; result = and(a,b)
3'b011	carry = 0; result = or(a,b)
3'b100	carry = 0; result = xor(a,b)
3'b101	carry = 0; result = not(a)
3'b110	carry = 0; result = a » b
3'b111	carry = 0; result = a « b

Table 5: Functional description

2. Create a simple verification plan to test basic operation of the ALU.
3. Simulate the design and fill in the verification plan.