# Lab 2: Introduction to *Verilog*

## CS207: Digital Logic

Jialin Liu

Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

16 September 2022

# Outline

# Outline of This Lecture

| | |
|---|---|
| 需求分析 | Specification |
| 高层次模块设计 | High-level Design |
| 低层次模块设计 | Low-level Design |
| 寄存器传输级（HDL描述电路） | RTL-coding |
| 电路逻辑仿真（前仿真） | Functional Verification by Simulations |
| 电路逻辑综合 | Logic Synthesis |
| 门级验证 | Gate-level Verification |
| 电路优化及布局布线 | Placement & Routing (PnR) |
| 电路时序仿真 | Post PnR Verification |
| 制造工艺 | Fabrication |
| 硅后验证 | Post Silicon Verification |

# Verilog HDL

Verilog is a hardware description language (HDL) (硬件描述语言) that describes the behaviour or functionality of digital circuits (behaviour modelling).

- Logic simulation: verify the functionality by simulations
- Logic synthesis: transfer the circuits described by HDL to actual circuits composed by basic logic components.

Syntax similar to `C`.

Top-down design methodologies.

# Level of Abstraction

In Verilog a module can be defined using various levels of abstraction. There are four levels of abstraction in *Verilog*. They are:

**Behavioural or algorithmic level**: This is the highest level of abstraction. A module can be implemented in terms of the design algorithm. The designer no need to have any knowledge of hardware implementation.
**Data flow level**: In this level the module is designed by specifying the data flow. Designer must how data flows between various registers of the design.
**Gate level**: The module is implemented in terms of logic gates and interconnections between these gates. Designer should know the gate-level diagram of the design.
**Switch level**: This is the lowest level of abstraction. The design is implemented using switches/transistors. Designer requires the knowledge of switch-level implementation details.

# Outline of This Lecture

# *Verilog* Syntax

- *Verilog* is **case-sensitive**.
- **Variable** names can contain letter, numbers, _ or $ and should start with a letter or _.
- **Keywords**:
  - Special identifiers reserved to define the language constructs and are in lower case.
  - Dozens of important keywords, such as `module`, `wire`, `assign`.
- **Comments**: `//` or `/*` `*/`
- **Operators**: unary, binary and conditional.
- **Number format**: `[size]'[base_format][number]`.
  - `[size]`: number of bits in the number, always decimal
  - `[base_format]`: specifies the base that the number part represents, among decimal (`d` or `D`), hexadecimal (`h` or `H`) and octal (`o` or `O`)
  - `[number]`: consecutive digits. 0,1,...,9 if decimal; 0, 1, ..., F if hexadecimal.
  - Example: `3'b010`, `-6'd2`, `-6'sd9`
- **String**: between “ ”. It can not be split into multiple lines.

# Outline of This Lecture

# Verilog Module (模块)

- A block of Verilog code that implements a certain functionality.
- Modules can be embedded within other modules.
- Modules can communicate with their inputs and outputs.
- A module should be created in a Verilog file (`.v`). The filename should match the module name.
- Enclosed with `module` and `endmodule`
- Module name right after `module`. The filename should match the module name.
- An optional list of `ports` (端口). Ports declared in the list of port declarations cannot be redeclared within the body of the module.
  - `input`: cannot be written inside the module.
  - `output`: cannot be read inside the module.
  - `inout`: can receive data or send data.
  - Data type: `wire` by default.

```verilog
module <name> ([port_list]);
  // content of the module
  /* This is a comment
     ...
     and another line */
endmodule
```

# Structure of Module

```
module 模块名[端口名1，端口名2，…];
  端口类型说明;                    //端口声明, input、output, inout
  参数定义;                        //参数声明。可选
  数据类型定义;                    //变量定义，wire、reg等

  //主体部分
  调用低层次模块和基本门级元件;
  连续赋值语句;                    //assign
  过程块                           //initial、always
  任务和函数;
endmodule
```

Figure: Figure from [1].

# Simple Example

```
module setbit(output A);
wire A;
assign A = 1;
endmodule
```

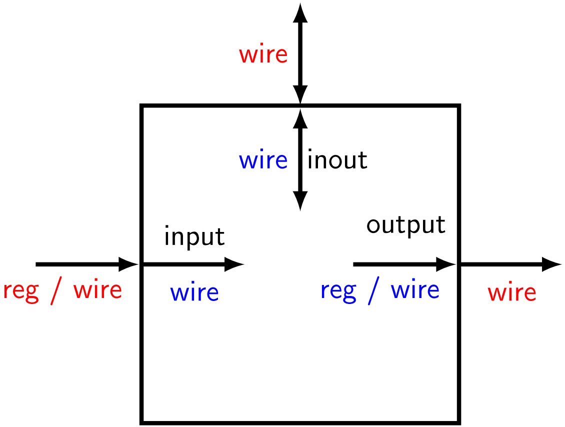
```verilog
module fport(output [3:0] data);
//-- Module output is a 4 wire bus.
wire [3:0] data;
//-- Output the value through that 4-bit bus.
assign data = 4'b1010; //-- 4'hA
endmodule

module fport_tb;
//-- 4-wire bus, to connect it to the Fport component output.
wire [3:0] DATA;
//--Instantiating the component. Connect output to DATA.
fport FP1 (.data (DATA));
//-- Begin the test
initial begin
  //-- After 10 time units we check whether the cable
  //-- has the previously given pattern or not.
  # 10 if (DATA != 4'b1010)
      $display("---->ERROR!");
    else
      $display("Component works!");
  //-- Finish the simulation 10 time units after that.
  # 10 $finish;
end
endmodule
```

# Connection of Ports

`input`: cannot be written inside the module.

`output`: cannot be read inside the module.

`inout`: can receive data or send data.



# Outline of This Lecture

# Four-Valued Logic

Almost all Verilog data types are 4-state:

- `0`: a low level signal, or a logic `0`, or a condition `false`
- `1`: a high level signal, or a logic `1`, or a condition `true`
- `x`: an unknown logic value, or don't care
- `z`: a high-impedance state (高阻态)

In *Verilog*, all data take values from the above four logic states.

**Format**: `[+/-]` `[size]`'`[signed]` `[base_format]` `[number]`

- `[+/-]`: positive / negative
- `[size]`: number of bits in the number, always decimal
- `[signed]`: signed number
- `[base_format]`: specifies the base that the number part represents, among decimal (`d` or `D`), hexadecimal (`h` or `H`) and octal (`o` or `O`)
- `[number]`: consecutive digits. 0,1,...,9 if decimal; 0, 1, ..., F if hexadecimal.

# Constants

## Examples of Integers

```
4'b1001      // 4位二进制数
5'D3         // 5位十进制数
3'b01x       // 3位二进制数，最低位为x
12'hx        // 12位数据均为x
8'd-6        //非法表示，数值<number>不能为负
-8'd6        //位宽为8，十进制数-6
4'shf        //4位有符号数1111，可表示为-4'h1（即-1）
-4'sd15      //等价于-(-4'd1)，即0001
27_195_000   //十进制数27195000，用"_"增加可读性
```

# Constants

## parameter

`parameter` to define an identifier for a constant.

**Syntax**: `parameter [signed][range] param1 = const_expr1, param2 = const_expr2, ... ;`

Examples:

```
parameter msb = 7;     //定义msb为常值7
parameter e = 25, f = 9;     //定义两个常值
parameter average_delay = (r + f) / 2;
     //带有表达式的参数常量
parameter signed [3:0] mux_selector = 0;
     //signed参数常量 hexadecimal.
```

# Categories of Data Types

Data storage elements and their physical connections.

Two categories of data types:

- `net`: physical connections. The mostly used one is `wire`.
- `variable`: data storage elements and connections. The mostly used one is `reg`.

# Data Type

- `wire`: usually used for the digital signal specified by `assign`. The default data type of `port` is `wire`.
- `reg`: data type for register
- `memory`: an array of registers
- `parameter`: used to define a constant
- `integer`
- `real`: double
- `realtime`: store time with `real`
- `wand`: `net` data type
- `wor`: `net` data type
- ... ...

# wire

The mostly used net data type. The data type of port is wire by default.

Used for the signal specified by assign, or input to expressions, or output of a component.

A wire variable: `wire data1, data2, ..., data9;`

```
module setbit(output A);
wire A;
assign A = 1;
endmodule
```

A vector of wire variables: `wire [n-1:0] data1, data2, ..., data5;`

```
module fport(output [3:0] data);
//-- Module output is a 4 wire bus.
wire [3:0] data;
//-- Output the value through that 4-bit bus.
assign data = 4'b1010; //-- 4'hA
endmodule
```

# reg

The mostly used `variable` data type. The data type of `port` is `wire` by default.

Usually be assigned by `always` or `initial` blocks.

A `reg` variable:

```
reg data1, data2, ..., data6;
```

A vector of `reg` variables:

```
reg [n-1:0] data1, data2, ..., data7;
```

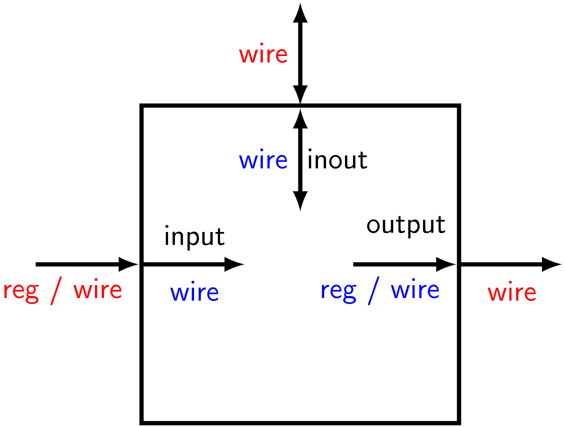
```
reg a;                          //a是1位的reg型变量
reg [7:0] qout;                 //qout是8位的reg型向量
reg signed [3:0] signed_reg;    //signed_reg是4位reg型向量，范围-8~7
```

# Connection of Ports

`input`: cannot be written inside the module.

`output`: cannot be read inside the module.

`inout`: can receive data or send data.



# Memory

Verilog uses an array of `reg` to define all types of computer memory.

`reg [n-1:0] memoryname[m-1:0];`

```
reg [7:0] mem[255:0];        //定义了名为mem的存储器，
                             //地址范围0~255，位宽8位
```

Comparing memory to an array of `reg`:

```
reg[7:0] rega;               //表示rega是1个8位的寄存器
reg memb[7:0];               //表示memb是8个1位的存储器
                             //地址范围从0~7
rega = 0;
memb[2] = 0;                 //如直接令memb = 0，则是非法的
```

# integer

Belongs to `variable`, often used as a couner.

`integer name1, name2, ..., name10;`

```
integer A, B;          //定义两个整型变量
A = 6 ;                //A的值为32 'h0000_0006
B = -6;                //B的值为32 'hFFFF_FFFA
```

# Outline of This Lecture

Introduction to *Verilog*

Syntax

Module

Data Types

Operations

# Operations

<table>
<tr><th>运算符</th><th>类别</th><th>备注</th></tr>
<tr><td>{} {{}}</td><td>拼接，复制</td><td></td></tr>
<tr><td>+ -</td><td>符号运算符（一元运算）</td><td></td></tr>
<tr><td>+ - * / ** %</td><td>算术运算符</td><td></td></tr>
<tr><td>> >= < <=</td><td>关系运算符</td><td></td></tr>
<tr><td>! && || == !=</td><td>逻辑运算符</td><td></td></tr>
<tr><td>=== !==</td><td>全等比较运算符</td><td>不可综合</td></tr>
<tr><td>~ & | ^ ^~或~^</td><td>按位运算符</td><td></td></tr>
<tr><td>& ~& | ~| ^ ^~或~^</td><td>归约运算符（一元运算）</td><td></td></tr>
<tr><td><< >> <<< >>></td><td>移位操作符</td><td></td></tr>
<tr><td>? :</td><td>条件运算符（三元运算）</td><td></td></tr>
</table>

# Concatenation

```
module concatenations;
reg [2:0] a,b;
initial begin
  a = 3'b100;
  b = 3'b111;
  $displayb({a,b[1:0]});  // 5'b100_11
  $displayb({2{a,b}});  // 12'b100_111_100_111
end
endmodule
```

concatenations.v

```
liu$ iverilog -o concatenations.o concatenations.v
liu$ vvp concatenations.o
10011
100111100111
```

# Arithmetic Operations I

```
module arithmetic_with_integer_reg;
integer intA;
reg [15:0] regA;
reg signed [15:0] regS;

initial begin
  intA = -4'd12;
  regA = intA / 3; // the value is -4, regA is 65532
  $displayb("regA: ", regA);
  regA = -4'd12; // regA is 65524
  $displayb("regA: ", regA);
  intA = regA / 3;  // the value and intA are both 21841
  $displayb("intA: ", intA);
  regA = -12 / 3; // the value is -4, regA is 65532
  regS = -12 / 3; // the value is -4, regS is -4
  $displayb("regA: ", regA);
  $displayb("regS: ", regS);
end
endmodule
```

arithmetic_with_integer_reg.v

# Arithmetic Operations II

```
liujl$ vvp arithmetic_with_integer_reg.o
regA: 1111111111111100
regA: 1111111111110100
intA: 00000000000000000101010101010001
regA: 1111111111111100
regS: 1111111111111100
```

# Relational Operations

```
module relational;
initial begin
  $displayb("2 > 1 is ", 2 > 1);
  $displayb("2 > 1'bx is ", 2 > 1'bx);
  $displayb("2 > -1 is ", 2 > -1);
  $displayb("2'd2 > 3'd1 is ", 2'd2 > 3'd1);
  $displayb("3'sd2 > -2'sd1 is ", 3'sd2 > -2'sd1);
  $displayb("2.0 > 1 is ", 2.0 > 1);
end
endmodule
```

relational.v

```
liujl$ vvp relational.o
2 > 1 is 1
2 > 1'bx is x
2 > -1 is 1
2'd2 > 3'd1 is 1
3'sd2 > -2'sd1 is 1
2.0 > 1 is 1
```

# Logical Operations

```verilog
module logical;
initial begin
  $displayb(!2'b10); // 0
  $displayb(!2'b00); // 1
  $displayb(!2'bx0); // x

  $displayb(2'b10 && 2'b10); // 1
  $displayb(2'b00 && 2'b10); // 0
  $displayb(2'bx0 && 2'b10); // x

  $displayb(2'b10 || 2'b00); // 1
  $displayb(2'b00 || 2'b00); // 0
  $displayb(2'bx0 || 2'b00); // x

  $displayb(2'b10 != 2'b00); // 1
  $displayb(2'b00 != 2'b00); // 0
  $displayb(2'bx0 != 2'b00); // x

  wire a=1;
  wire b=1;
  reg f;
  and and1(f, a, b);
  $displayb(and1(a,b));
end
endmodule
```

logical.v

# 全等比较运算符

```
module case_equality;
initial begin
  $displayb(2'b10 === 2'b00);   // 0
  $displayb(2'b00 === 2'b00);   // 1
  $displayb(2'bx0 === 2'bx0); // 1
  $displayb(2'bz0 === 2'bz0);   // 1
end
endmodule
```

case_equality.v

# 按位操作符

```
module bitwise;
initial begin
  $displayb(~3'b101);                    // 3'b010
  $displayb(3'b101 & 3'b100);   // 3'b100
  $displayb(3'b101 | 3'b100);      // 3'b101
  $displayb(3'b101 ^3'b100);      // 3'b001
  $displayb(3'b101 ^~ 3'b100);  // 3'b110
end
endmodule
```

bitwise.v

# 规约操作符

```
module reduction;
initial begin
  $displayb(&4'b0000);      // 0
  $displayb(&4'b1111);      // 1
  $displayb(~&4'b0000);     // 1
  $displayb(~&4'b1111);     // 0
  $displayb(|4'b0000);        // 0
  $displayb(|4'b1111);        // 1
  $displayb(~|4'b0000);       // 1
  $displayb(~|4'b1111);       // 0
  $displayb(^4'b0000);        // 0
  $displayb(^4'b1111);        // 0
  $displayb(^4'b1000);        // 1
  $displayb(~^4'b0000);       // 1
  $displayb(~^4'b1111);       // 1
  $displayb(~^4'b1000);       // 0
end
endmodule
```

reduction.v

# 移位操作符

```
module shift;
initial begin
  $displayb(4'sb1001 << 2 );   // 4'sb0100
  $displayb(4'sb1001 >> 2 );   // 4'sb0010
  $displayb(4'sb1001 <<< 2);   // 4'sb0100
  $displayb(4'sb1001 >>> 2);   // 4'sb1110
end
endmodule
```

shift.v

# Conditional Operation (条件运算符)

Syntax: `d= a?b:c`

```
wire [15:0] busa = busa_en ? data : 16'bz;
```

# Priority of Operations

<table>
<tr><th>运算符</th><th>优先级</th></tr>
<tr><td>+ - ! ~ & ~& | ~| ~^或^~（一元运算）</td><td>最高优先级</td></tr>
<tr><td>**</td><td rowspan="12"></td></tr>
<tr><td>* / %</td></tr>
<tr><td>+ - （二元运算）</td></tr>
<tr><td><< >> <<< >>></td></tr>
<tr><td>< <= > >=</td></tr>
<tr><td>== != === !==</td></tr>
<tr><td>& （二元运算）</td></tr>
<tr><td>^ ^~或~^ （二元运算）</td></tr>
<tr><td>| （二元运算）</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>? :</td></tr>
<tr><td>{} {{}}</td><td>最低优先级</td></tr>
</table>

# References


1 Textbook by 薛一鸣，文娟（出版社：清华大学出版社）：

高等学校电子信息类专业系列教材

**FPGA数字系统设计**

薛一鸣 文娟 编著


2 Woo, Jeong-Ho et al. "Mobile 3D Graphics SoC: From Algorithm to Chip." (2010).

3 `https://verilogguide.readthedocs.io/en/latest/index.html`

4 `https://www.chipverify.com/verilog/verilog-introduction`

5 Verilog Tutorial by Deepak Kumar Tala `http://classweb.ece.umd.edu/enee359a/verilog_tutorial.pdf`