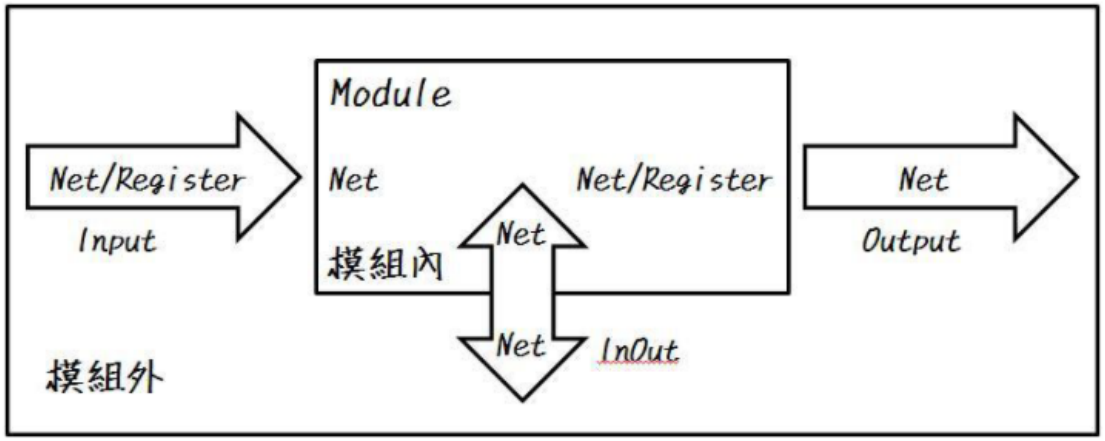
https://hom-wang.gitbooks.io/verilog-hdl/content/Chapter_01.html

https://www.nandland.com/verilog/tutorials/index.html

http://www.utdallas.edu/~akshay.sridharan/index_files/Page5212.htm

http://www.darwin.esys.tsukuba.ac.jp/home/ohyou/verilog/operator



http://www.darwin.esys.tsukuba.ac.jp/home/ohyou/verilog/operator

0001000000000000000000000000000

https://www.hdlworks.com/hdl_corner/verilog_ref/items/Instantiation.htm

Module

A module is the basic unit in Verilog and represents hardware components by describing the interface and design behaviour.

Syntax:

```
module_word module_name [ ( port_list ) ];
   module_items;
endmodule
```

module_word = **module** | **macromodule**

Modules can be used to represent hardware components by describing the hardware interface and behaviour, or can be used to declare parameters, tasks and functions that are used elsewhere in the design.

The keyword **macromodule** is a synonym for **module**. Some EDA tools compile macromodules differently from modules, for example by flattening macromodule hierarchy. This might make simulation more efficient in terms of speed or memory.

Example:

```
module Mod1(A, B, C);
   input A, B;
   ouput C;
   assign C = A & B;
endmodule
```

Port Declaration

The ports of a module declare the interface of the module.

Syntax:

port_direction [port_size] port_name, port_name, ...;
port_direction data_type [port_size] port_name, port_name, ...;

port_direction = **input** | **output** | **inout**

Description:

The module *ports* model the pins of hardware components. The *port declaration* specifies the port direction of the ports listed in the *module declaration*.

Verilog requires that signals connected to the input or output of a module have two declarations: the port direction, and the data type of the signal. If no data type is declared, it is implicity declared as a wire with the same size as the corresponding port.

In Verilog-2001 the two declarations, direction and data type, may be combined in one statement.

The port declaration and the port list in the module declaration may even be combined in one statement in Verilog-2001. This combination is known as the ANSI-style. The syntax for this combination is:

module module_name (port_direction data_type [port_size] port__name, port_name, ...);

Example:

```
input Clk;
output [7:0] Q;
input wire Clk;                                  // Verilog-2001
output reg [7:0] Q;                              // Verilog-2001

module Cnt (output reg [7:0] Q,
            input wire Clk, Reset, Enable,
            input wire [7:0] D );                // Verilog-2001 ANSI-style
```

Verilog Operators

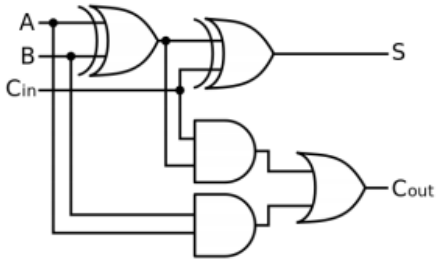
Operator Type	Symbol	Operation Performed
Arithmetic	+	Add
	–	Substract
	*	Multiply
	/	Divide
	%	Modulus
Logical	!	Logical **negation**
	&&	Logical **and**
	‖	Logical **or**
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	Inequality
Bitwise	~	Bitwise **negation**
	&	Bitwise **AND**
	|	Bitwise **OR**
	^	Bitwise **XOR**
Reduction	&	Reduction **AND**
	|	Reduction **OR**
	~&	Reduction **NAND**
	~|	Reduction **NOR**
	^	Reduction **XOR**
	~^ or ^~	Reduction **XNOR**
Concatenation	{}	
Replication	{{}}	{4{3'b011}}，{2{4'hA}}
Shift	<<	Left shift (i.e. *2^n, n moving bits)
	>>	Right shift (i.e. /2^n, n moving bits)
Conditional	? :	If … else … (e.g. out=A==1? B: C)

```
Out = ( A>B ) ? A : B;   //若 A>B, Out = A 反之 若 A<=B, Out = B
If( A>B )       // 使用 if-else
      Out = A;
else
      Out = B;


A = { 1'b0, 1'b1 };              // A = 2'b01
A = { B[1:0], C[0], D[2] };      // A = B[1], B[0], C[0], D[2]
A = { 2{2'b01} };                // A = 4'b0101
A = { 3'b101, 2{1'b0} }          // A = 5'b10100
```

Reduction

Verilog has six reduction operators, these operators accept a single vectored (multiple bit) operand, performs the appropriate bit-wise reduction on all bits of the operand, and returns a single bit result. For example, the four bits of A are **AND**ed together to produce Y1.

```
module Reduction (A, Y1, Y2, Y3, Y4, Y5, Y6);
        input [3:0] A;
        output Y1, Y2, Y3, Y4, Y5, Y6;
        reg Y1, Y2, Y3, Y4, Y5, Y6;

        always @(A)
        begin
                Y1=&A; //reduction AND
                Y2=|A; //reduction OR
                Y3=~&A; //reduction NAND
                Y4=~|A; //reduction NOR
                Y5=^A; //reduction XOR
                Y6=~^A; //reduction XNOR
        end
endmodule
```



Gate-Level Circuit

Using Structural Verilog

```
module full_adder (input a, input b, input cin, output s, output cout);
xor(s, a, b, cin);

wire xor_a_b;
wire cin_and;
wire and_a_b;

xor(xor_a_b, a, b);
and(cin_and, cin, xor_a_b);
and(and_a_b, a, b);
or(cout, and_a_b, cin_and);
endmodule
```

Using Dataflow Verilog

```
module full_adder (input x, input y, input cin, output s, output cout);
assign s = x ^ y ^ cin;
assign cout = (a && b) || (cin && (a ^ b));        //a&&b || cin&&(a || b)
endmodule
```

Using Behavioral Verilog

```
module Full_Adder( A, B, Cin, Sum, Cout );
      input A, B, Cin;
      output Sum, Cout;

      wire W1, W2, W3;

      always @( A, B, Cin ) begin
            { Cout, Sum } = A + B + Cin;
      end
endmodule
```

http://www.utdallas.edu/~akshay.sridharan/index_files/Page5212.htm
https://www.itread01.com/content/1519893495.html
https://www.fpga4student.com/2017/08/verilog-code-for-pwm-generator.html

Parameters are bound during static elaboration creating flexible modules

```
module vcMux2
#( parameter WIDTH = 1 )
(
  input  [WIDTH-1:0] in0, in1,
  input  [1:0]       sel,
  output [WIDTH-1:0] out
);

  always @(*)
  begin
   case ( sel )
     1'd0 : out = in0;
     1'd1 : out = in1;
     default : out = {WIDTH{1'bx}};
   endcase
  end

endmodule
```

Instantiation Syntax

```
vcMux2#(32) alu_mux
(
  .in0 (op1),
  .in1 (bypass),
  .sel (alu_mux_sel),
  .out (alu_mux_out)
);
```

http://vcomp.sourceforge.net/doc-vcomp-10.shtml

Verilog 'generate' statements allow conditional declarations of variables, instantiation of other modules or of always statements.

The basic syntax is:

```
generate

    .... generate statements

endgenerate
```

Valid 'generate' statements include:

```
if (<expression>) <generate statement>

if (<expression>); <generate statement> else
                   <generate statement>

begin : <block name>        <generate statements> end

case (<expression>)
<expression>: <generate statement>
....
default:    <generate statement>
endcase

for (<assign>; <expression>; <assign>)
begin : <block name>
end
```

```
module x(input in, output out);
    parameter p = 1;        // set when x is instantiated
    parameter y = 1;

    wire        w[p:0];     // an ARRAY of wires
    assign out = w[p];
    assign w[0] = in;

    generate

        if (p == 2)         reg x;                          // x is only declared if p = 2
        if (y > 8)   reg [y:0]r; else reg [7:0]r; // r is min 8 bits wide

        case (p)
        1:      initial r = 3;
        2:      initial begin x = 1; r = 4; end
        default:initial r = p;
        endcase

        genvar xx;

        for (xx = 0; xx < p; xx=xx+1) begin:block
            b #(xx+1)b(.in(w[xx]), .out(w[xx+1]));
        end
    endgenerate
```

Blocking/Non-Blocking 敘述

- Blocking (=)，具有順序性，敘述會與先後有關係 (順序處理)
- Non-Blocking (<=)，具有同時性，敘述與先後沒有關係 (平行處理)

範例：

```
/* Blocking ( = ) */
輸出 = 輸入邏輯;

/* Non-Blocking( <= ) */
輸出 <= 輸入邏輯;
```

範例：

```
input In;
reg [3:0] A, B, C;

always @( posedge CLK ) begin

    /* Blocking */      // 有順序執行，同 C 概念
    A[0] = In;          // 1CLK 後 A[0] = In
    A[1] = A[0];        // 1CLK 後 A[1] = A[0] = In
    A[2] = A[1];        // 1CLK 後 A[2] = A[1] = A[0] = In
    A[3] = A[2];        // 1CLK 後 A[3] = A[2] = A[1] = A[0] = In

    /* Non-Blocking */  // 同時執行，此範例有資料平移的效果
    B[0] <= In;         // 1CLK 後 B[0]存進 In, B[1]存進 B[0](存 In 之前的值)
    B[1] <= B[0];       // 2CLK 後 B[1]存進 In
    B[2] <= B[1];       // 3CLK 後 B[2]存進 In
    B[3] <= B[2];       // 4CLK 後 B[3]存進 In

    /* 混合 */          // 盡量不要常用
    C[0] <= In;         // 1CLK 後 C[0]存進 In, C[1], C[2]存進 C[0](存 In 之前的
值)
    C[1] = C[0];        // 2CLK 後 C[1] = C[2] = C[3] = In
    C[2] = C[1];        //
    C[3] <= C[2];       //

end
```

`define Directive :

A `` `define `` is a Verilog compiler directive which only substitutes the macro with the text associated with it. It is very similar to the `#define` compiler directive in C.

```
1.  `define SYNTHESIS                  // Uncommented for synthesis  (`define
    method 1)
2.  //`define SIMULATION                 //Uncommented for simulation
3.  `include some_header.v               // Header file
4.  module some_name #(                  // Module name declaration and start
    of parameters list in Verilog 2k syntax
5.  `ifdef SYNTHESIS                     // Condition check for macro
6.    `define myDef(a,b) (b > a) ? a : b // `define method 2
7.    parameter [3:0] INWIDTH = some_number_1, // parameter definition
8.    parameter [3:0] OUTWIDTH = some_number_2,
9.  `else
10.         `define myDef(a,b) (a > b) \      // `define method 3
11.                         ? a : b
12.         parameter INWIDTH = some_number_3,
13.         parameter OUTWIDTH = some_number_4,
14.       `endif
15.         parameter HALFWIDTH = INWIDTH >> 1, QUARTWIDTH = INWIDTH >> 2 //
    parameter definition with expression
16.       )
17.       (
18.         input  [INWIDTH-1:0]  myInput,    //Input port with
    parametrized width
19.         output [OUTWIDTH-1:0] myOutput    //Output port with
    parametrized width
20.         /*
21.           ,
22.           some_more_ports               //(if needed)
23.         */
24.       );
25.
26.         wire [HALFWIDTH-1:0]inLSB = myInput[(HALFWIDTH-1)-:HALFWIDTH];
27.         wire [HALFWIDTH-1:0]inMSB = myInput[(INWIDTH-1)-:HALFWIDTH];
28.         wire y = `myDef(inLSB,inMSB);
29.       /*
30.         Functionality
31.       */
32.       endmodule
```

verilog HDL 程序中的兩個系统任務，$readmemb 和$readmemh，**從文件中讀取數據到暫存器**(存储器)。其格式如下：

（1）$readmemb("<数据文件名>",<存储器名>);

（2）$readmemb("<数据文件名>",<存储器名>,<起始地址>);

（3）$readmemb("<数据文件名>",<存储器名>,<起始地址>,<终止地址>);

（1）$readmemh("<数据文件名>",<存储器名>);

（2）$readmemh("<数据文件名>",<存储器名>,<起始地址>);

（3）$readmemh("<数据文件名>",<存储器名>,<起始地址>,<终止地址>);

被读取的文件中只能包含：空白位置（空格、换行、制表格（tab）），注释行（//形式的和/*...*/形式的都可以）、二进制和二六进制数据。

被读取的文件中不能包含位宽书名和格式说明，**對於$readmemb 系统任務，每個數必須是二進制**，对于$readmemh 系统任务，每个数必须是十六进制。**數字中可以有不定值 x 或X和高阻值 z 或Z，還可以有下畫線（_）**。另外，数字必须用空白位置或注释行来分隔。

任务会从指定的地址依次将读取到的数据存入寄存器（或寄存器数组），但当地址出现在数据文件中时，其格式是字符“@”后跟上十六进制数据，如：@hhhh。当读取中遇到地址说明符，会将地址后的数据存放到相应的地址中。

如：文件 init.dat 内容如下：

@002

11111111 01010101

00000000 10101010

@006

1111zzzz 00001111

verilog 程序如下：

```
reg [7:0] meme[0:7];
$readmemb("init.data",meme);
```

则寄存器中的内容如下：

meme[0]=xxxxxxxx;

meme[1]=xxxxxxxx;

meme[2]=11111111;

meme[3]=01010101;

meme[4]=00000000;

meme[5]=10101010;

meme[6]=1111zzzz;

meme[7]=00001111;

如果程序如下：

```
reg [15:0] meme[0:7]; //一个地址存储 1 6bit 数据
```

$readmemb("init.data",meme);
则结果如下：
meme[0]=xxxxxxxxxxxxxxxx;
meme[1]=xxxxxxxxxxxxxxxx;
meme[2]=11111111_01010101;
meme[3]=00000000_10101010;
meme[4]=xxxxxxxxxxxxxxxx;
meme[5]=xxxxxxxxxxxxxxxx;
meme[6]=1111zzzz_00001111;
meme[7]=xxxxxxxxxxxxxxxx;
则依次从文件中读取 16bit 的数据存储到寄存器的一个地址中。
补充说明：
（１）系统任务声明语句中和数据文件中都没有地址说明，则默认的存放地址为存储器定义语句中的起始地址，数据文件里的数据被连续存存放到该存储器中，直到存储单元存满为止或者数据文件里的数据存完。
（２）如果系统任务中说明了存放的起始地址，没有说明存放的结束地址，则数据从起始地址开始存放。
（３）如果数据文件里的数据个数和系统任务中起始地址和结束地址的数据个数不同的话，会提示出错信息。
（４）reg [7:0] meme[0:7] //地址为０－７　存储器定义的起始地址和结束地址
$readmemb("init.dat",meme,3,6) // 系统任务中定义的起始地址和结束地址
@006　　//数据文件中的地址
其中数据文件中地址必须在系统任务中定义的范围内，系统任务中定义的地址必须在存储器定义的地址范围内。优先考虑数据文件中的地址>系统任务中定义的起始地址和结束地址>存储器定义的起始地址和结束地址.