Chapter 5. Verilog Hardware Description Language

A hardware description language (HDL) is the counterpart of a high-level programming language in software design.

It is used for describing hardware.
A set of related software tools allow us to automatically simulate and synthesize the design.

A typical tool suite includes:

A *text editor* for writing, editing and saving an HDL program.

A *compiler* that produces an intermediate description that is technology independent.
It also finds syntax errors.

A *synthesis tool* produces a design that targets a specific hardware technology.
It refers to one or more *libraries* that contains building blocks of the final design.
The result is a design that is ready to be implemented.
For custom design of large ICs, synthesis may be partitioned into gate-level synthesis and physical design (place and route).

A *simulator* accepts the HDL program and inputs.
The simulator determines values of internal signals and outputs.
This is useful for *verification* of the design, which is the process of ensuring that the design is functionally correct.

A *timing analyzer* calculates the worst-case paths in the design and their delays.

A *power estimation tool* calculates the worst-case power that the design may consume.

5.1. Verilog Models and Modules

The basic unit of design in Verilog is a *module*. This is a text file that contains declarations and statements.

The *declarations* describe the names and types of the inputs, outputs, local signals, variables, constants, and functions that are internal to the module.

The *statements* specify the operation of the module.

Verilog statements can specify the operation of a module *behaviorally*, based on its function.

The operation of a module can also be specified *structurally* by describing which other modules or individual components (such as gates) are used, and how they are interconnected.
This is equivalent to providing a logic diagram.

When a module refers to another module, it is said to instantiate the other module.
The modules are defined hierarchically.
A higher-level module may insantiate several lower-level modules, each one several times.
Several higher-level modules may instantiate the same lower-level module.

The definitions inside a module remain local to the module.
Values are passed between modules using declared input and output signals.

Short comments begin with // and end at the end of the line.
Longer comments can start with /* and end with */.

Syntax for a module declaration:

```
module module-name(port-name,port-name,...);
    input declarations
    output declarations
    inout declarations
    net declarations
    variable declarations
    parameter declarations
    function declarations
    task declarations
    concurrent statements
endmodule
```

Example:

```
module VrInhibit(X,Y,Z); // X and not Y
    input X,Y;
    output Z;
    assign Z=X&~Y;
endmodule
```

The strings module, input, output, assign and endmodule are keywords, and have special meaning in Verilog.

User-defined identifiers start with a letter or underscore and can contain letters, digits, underscores, and dollar signs.
Verilog is case-sensitive.

input specifies that the signal is an input to the module.

output specifies that the signal is an output of the module.
Its value cannot be read inside the module.

inout specifies that the signal can be used as both an input and an output.

Without a range, a signal has a single bit.
To define a vector, it is possible to specify a range of indices for bits within the vector:
input [msb:lsb] identifier;
msb specifies the index of the leftmost signal.
lsb specifies the index of the rightmost signal.

5.2. Logic System, Nets, Variables and Constants

A 1-bit signal can take on one of four possible values:
0 Logical 0, or false
1 Logical 1, or true
x An unknown logical value
z High impedance

Verilog has the following bitwise Boolean operators:

operator	operation
&	AND
|	OR
~	NOT
^	Exclusive OR
~^,^~	Exclusive NOR

A net corresponds to a wire.
It provides connectivity between modules as well as between elements inside a structural model.
There are several types of nets.

wire provides connectivity, and no functionality is implies.
This is the defaul net type.
A signal name that appears in the input/output list but not declared is assumed to be of type wire.

supply0 and supply1 are permanently wired to the corresponding power-sypply rail (ground or V_{CC}).
These are used for constant 0 and 1 signals.

tri is used for three-state wires.

Other types exist.

Syntax:
wire identifier, identifier, ...;
wire [msb:lsb] identifier, identifier, ...;
tri identifier, identifier, ...;
tri [msb:lsb] identifier, identifier, ...;

Variables are used during execution of Verilog code.
They do not necessarily have physical counterparts in a circuit.
They cannot be changed from outside the module.
This implies that they cannot be of type input or inout.

Commonly used variable types: reg and integer.

Syntax:
reg identifier, identifier, ...;
reg [msb:lsb] identifier, identifier, ...;
integer identifier, identifier, ...;

In the case of reg, the value of a bit can be 0, 1, x or z.

The value of an integer variable is a two's-complement number with 32 bits or more, depending on the word length used by the Verilog tools.

Outputs can have type net or reg.
Procedural (behavioral) code can assign values only to variables, not to nets.
An output of type reg can be assigned values in the procedural code of the module.
The value is always present on the output port.

To write procedurel code for the value of an output of a module:

1. If the output is declared as type reg, procedural code can write to it directly.

2. If the output needs to have type net (for example, tri), it is possible to define an internal reg variable, and use procedural code to write to it.
Then assign the value of the internal reg variable to the output net.

A number such as 1234 or -5678 is interpreted as a signed decimal number.

The general format for a number:
$n'Bdd\cdots d$

n is a decimal number equal to the number of bits in the representation of the number.
If it is missing, the default is 32 bits or the word width used by the Verilog tools.

B is a single letter specifying the base.

b or B	binary
o or O	octal
h or H	hexadecimal
d or D	decimal

$dd\cdots d$ is a string of one or more digits in the specified base.
In hexadecimal, $a-f$ may be lower or upper case.
If the number is longer than n bits, leftmost bits are discarded.
If the number is shorter than n bits, leftmost bits are zeroes.

Examples:

1'b0	0
1'b1	1
1'bx	x
8'b00000111	00000111
8'b0	00000000
8'b1	00000001
8'h17	00010111
2'b1011	11
4'b1zzz	1zzz
4'b1x1	01x1

A parameter declaration allows contants to be defined.

Syntax:

```
parameter identifier = value,
          identifier = value, ...;
```

The value can be an expression.

Examples:

```
parameter BUS_SIZE = 32,
          MSB = BUS_SIZE-1, LSB = 0;
parameter BELL = 7'b0111000; // ASCII code
```

5.3. Vectors and Operators

reg [msb:lsb] identifier;
msb specifies the index of the leftmost signal.
lsb specifies the index of the rightmost signal.

Examples:
reg [7:0] byte1, byte2, byte3;
reg [15:0] word1, word2;
reg [1:16] Zbus;

Selecting a single bit:

byte1[7]	leftmost bit of byte1
byte1[6]	right of leftmost bit of byte1
Zbus[16]	rightmost bit of Zbus
Zbus[15]	left of rightmost bit of Zbus

Selecting a range of bits:

Zbus[1:8]	leftmost byte of Zbus
Zbus[9:16]	rightmost byte of Zbus
byte1[5:2]	midle four bits of byte1

It is possible to concatenate bits or parts of vectors to form larger vectors.
The concetenation operator uses braces {}.

Examples:

{2'b00,2'b11}	4'b0011
{byte1,byte1,byte2,byte3}	32-bit vector
	byte1byte1byte2byte3

The replication operator *n*{} can be used within a concatenation to replicate a bit or vector *n* times.

Example:

{2{byte1},byte2,byte3} 32-bit vector
byte1byte1byte2byte3

The bitwise Boolean operators can be applied to vectors.
With vectors of different sizes, the shorter vector is padded on the left with 0s, and the vectors are aligned on their leftmost bits.

Example: 2’b11 & 4’b1101

2’b11	0	0	1	1
4’b1101	1	1	0	1
2’b11 & 4’b1101	0	0	0	1

Padding with zero applies to numbers as well.
The only exception is if the leftmost bit is x or z.
In this case, the padding is with x or z.

Example: 4'bz00 is equivalent to 4'bzz00

Sign extension is used only for integer variables.

Arithmetic operators treat vectors as unsigned integers by default.

+	addition
-	subtraction
*	multiplication
/	division
%	modulus (remainder)
**	exponentiation
<<	logical shift left
>>	logical shift right
<<<	arithmetic shift left
>>>	arithmetic shift right

An addition operator will cause an adder to be synthesized.

A multiplication operator will cause a multiplier to be synthesized.
It is also possible to write a module for a multiplier if the performance and size of the multiplier are imporatnt.

Logic shift operators fill new bits with 0s.
Examples of logical shift operators:

4’b1101<<2 0100
4’b1101>>2 0011

The same result is obtained with unsigned numbers and the arithmetic shift operators.

Arithmetic shift right for a signed number performs sign extension.
Example:

4’b1001>>1 1100

The bitwise Boolean operators can be applied to a single vector operand.
In this case they will combine all the bits in the vector using the corresponding operation, and return a 1-bit result.

Examples:

Zbus	&Zbus	|Zbus
0000	0	0
0010	0	1
1111	1	1
1x1x	x	1
1x10	0	1
0xx0	0	x
xxxx	x	x

5.4. Arrays

An array is an ordered set of variables of the same type.
Each element is selected by an array index.

Syntax:
reg identifier [start:end];
reg [msb:lsb] identifier [start:end];
integer identifier [start:end];
wire identifier [start:end];
wire [msb:lsb] identifier [start:end];

Example:
reg [7:0] mem [0:255]
mem is an array of 256 8-bit vectors.

Accessing array elements or ranges of elements:
mem[5] is the element with index 5.
mem[10][3:0] is the vector consisting of the four rightmost bits of mem[10].

5.5. Logical Operators and Expressions

The 1-bit value 1’b1 is considered true.
The 1-bit values 1’b0, 1’bx and 1’bz are considered false.

A multi-bit value with all known bits is true if it is nonzero.
4’b0100 is true, 4’b1011 is true, 4’b0000 is false.
An operand that contains x or z bits is considered as 1’bx, which is treated as false.

Logical operators combine true and false values.
A logical operator yields a value 1'b1 if the result is true.
A logical operator yields a value 1'b0 if the result is false.
If one of the operands contains x or z bits, a logical operator yields a value x, which is treated as false.

operator	operation
&&	logical AND
||	logical OR
!	logical NOT
==	logical equality
!=	logical inequality
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

Example: 4’b0100&4’b1011 = 1’b1
4’b0100 evalutes to 1’b1.
4’b1011 evaluates to 1’b1.
The AND yields 1’b1.

With at least one unsigned operand, both operands are considered unsigned.
A shorter operand is padded with 0s on the left.

Signed numbers are considered only if both operands are signed.
In this case, a shorter operand is sign extended.

When synthesized, the last six operators create comparators.
For performance and size it is better to write modules for comparators.

operator	operation
===	case equality
!==	case inequality

These operators compare the operands bit by bit and produce either 0 or 1 (x is avoided).

Example:
4’b01xx == 4’b01xx evalutes to x
4’b01xx === 4’b01xx evalutes to 1

These should not be used in synthesizable modules since there is no circuit element that distinguishes an x or z from a 0 or 1.

The syntax for the conditional operator is
logical-expression ? true-expression : false-expression

Example:
(A>B)?A:B;
Select the maximum of A and B.

5.6. Compiler Directives

‘include filename
The file is read immediately and processed as if it had been included in the current file.
This is typically used to read in definitions that are common to multiple modules.
The included file can contain include directive.

‘define identifier text
The compiler replaces each appearance of identifier with text.
There is no evaluation or any other processing of text.
The definition is in effect for the current file as well as subsequent files that are processed during the same compiler run.
This includes files that are processed using include.

5.7. Structural Models

Syntax for a module declaration:

```
module module-name(port-name,port-name,...);
    input declarations
    output declarations
    inout declarations
    net declarations
    variable declarations
    parameter declarations
    function declarations
    task declarations
    concurrent statements
endmodule
```

concurrent statement types: instance, continuous-assignment, always, and initial.

In the structural style of circuit description, individual gates and other components are instantiated and connected using nets.
Each component has a module declaration in a library.

Available gate type:

and	nand	or	nor
xor	xnor	buf	not
bufif0	bufif1	notif0	notif1

The last four have enable inputs, and three-state outputs.
For example, bufif0 acts as a buffer when the enable input is 0.

Syntax for instantiating a gate or component:
component-name instance-identifier (expr, expr, ..., expr);
component-name instance-identifier
(.port-name(expr),
.port-name(expr), ...,
.port-name(expr));

Only the first format is allowed for built-in gates.
The ports need to be given in the order
(output, input, input, ...)

For three-state buffers and inverters the order is
(output, data-input, enable-input)

Earlier we saw the module:

```
module VrInhibit(X,Y,Z); // X and not Y
    input X,Y;
    output Z;
    assign Z=X&~Y;
endmodule
```

The same module using structural style:

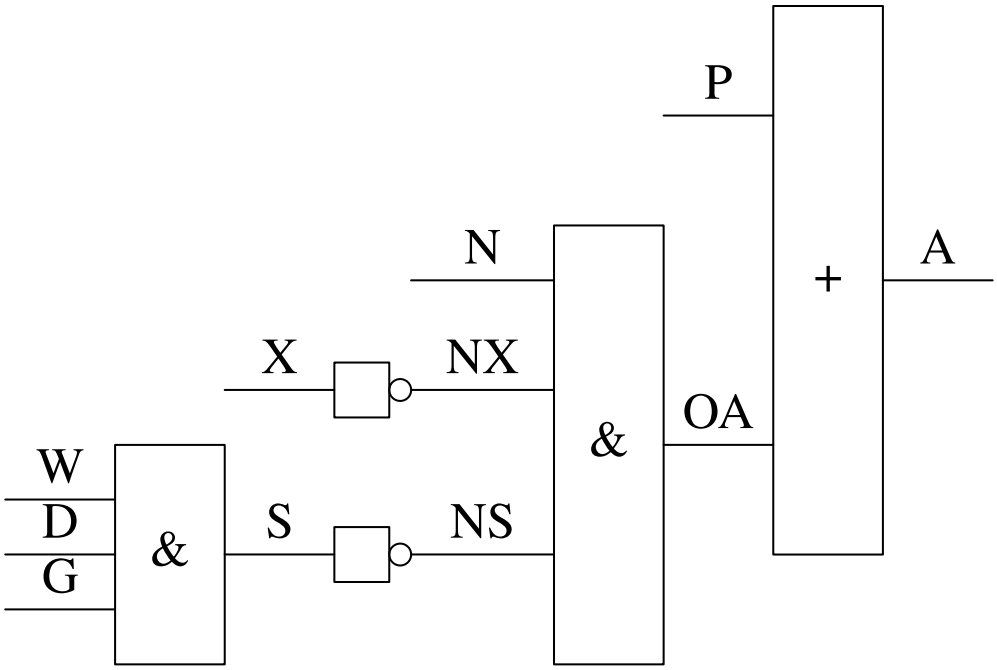
```
module VrInhibit(in,invin,out);
    input in,invin;
    output wire out;
    wire notinvin;
    not U1 (notinvin,invin);
    and U2 (out,in,notinvin);
endmodule
```

The order of the instance statements is not important since the statements execute concurrently.

The same circuit would be synthesized regardless of the order of the statements.

Simulation would produce the same results.

Another example:




```
module VrAlarmCkt (input P,N,X,W,D,G,
        output A);
    wire S,NS,NX,OA;
    or U1 (A,P,OA);
    and U2 (OA,N,NX,NS);
    not U3 (NX,X);
    not U4 (NS,S);
    and U5 (S,W,D,G);
endmodule
```

Parameters can be used to parametrize structural modules to handle inputs and outputs of any width.

Example: A 3-bit majority function implements the function

$$OUT = (I0 \cdot I1) + (I0 \cdot I2) + (I1 \cdot I2)$$

```
module Maj(OUT,I0,I1,I2);
    parameter WID=1;
    input [WID-1:0] I0,I1,I2;
    output [WID-1:0] OUT;
    assign OUT = (I0&I1)|(I0&I2)|(I1&I2);
```

Suppose that W, X, Y and Z are 8-bit vectors.
It is possible to instantiate an 8-bit majority function for X, Y and Z, with output W, by using:
Maj #(8) U1(.OUT(W),.I0(X),.I1(Y),.I2(Z));

If a module has several parameters, they need to be listed in the order by which they appear in the module.
If only some of them need to be changed, it is necessary to specify all the parameters until the last one that needs to be changed.

To avoid this, it is possible to list the parameters by name.
For the majority module:
Maj #(.WID(8)) U1 (.OUT(W), .I0(X), .I1(Y), .I2(Z));

A module called Mod with three parameters, called P1, P2 and P3:
Mod #(.P3(8), .P1(4)) U2(.OUT(F), .I0(A), .I1(B), .I2(C));

Parameters can be substituted only when modules are instantiated structurally.

5.8. Dataflow Models

Dataflow models use the continuous-assignment statement.
Syntax:
assign net-name = expression;
assign net-name[bit-index] = expression;
assign net-name[msb:lsb] = expression;
assign net-concatenation = expression;

The statement causes the value of the righthand side to be evaluated, and assigned to the lefthand side continuously.

The order of assign statements is not important.
If a later statement changes a net value use by an earlier statement, the simulator will re-evaluate the earlier statement and update its result.

Example using the conditional operator:

```
module Vrbytes1(A,B,C,selA,selB,selC,Z);
    input [7:0] A,B,C;
    input selA,selB,selC;
    output [7:0] Z;
    assign Z = selA ? A:
        (selB ? B:
            (selC ? C:8'b0));
endmodule
```

If selA=true, Z=A.
Else if selB=true, Z=B.
Else if selC=true, Z=C.
Else, Z=0.

5.9. Behavioral Models (Procedural Code)

Behavioral modeling uses the always statement.
Syntax:

always @(signal-name or signal-name or ... or signal-name)
    procedural statement

always @(signal-name, signal-name, ... signal-name)
    procedural statement

always @(*)
    procedural statement

always @(posedge signal-name)
    procedural statement

always @(negedge signal-name)
    procedural statement

always
    procedural statement

procedural statement is typically a block of statements.

Procedural statements in an always block execute sequentially.

The always block executes concurrently with other concurrent statements in the same module.

This implies that if a procedural statement changes the value of a net or variable used by another concurrent statement, it will cause the other statement to be re-executed.

always @(signal-name or signal-name or ... or signal-name)
    procedural statement
always @(signal-name, signal-name, ... signal-name)
    procedural statement

The signals in parentheses are called the sensitivity list.
The sensitivity list should specify all the signals whose values may affect the results in the always block.

always @(*)

    procedural statement

This form of a sensitivity list stands for every signal that may change a result.

During simulation, an always block is initially suspended.

When any signal in its sensitivity list changes value, the block resumes execution.

A resumed always block executes its statements sequentially starting from the first one.

If any signal in the sensitivity list changes value as a result of executing the always block, the block is excuted again.

This continues until none of the signals change value.

It is possible to write statements that will exceutive forever.

Example: not(X,˜X) assigns X=X'. X will change its value forever.

Simulators terminate such statements after a number of passes.

Procedural statements available for use within an always block:
assignment, begin-end block, if, case, while and repeat.

When using conditional statements, it is important to ensure that every variable is assigned a value in every execution pass.
Otherwise Verilog assumes that the previous value of the variable needs to be stored, and adds a memory element for the variable.

Assignment statements have the following syntax:
variable-name = expression;
variable-name <= expression;

The first is referred to as a blocking assignment.
It behaves as expected of an assignment.
It evaluates the righthand side, and assigns it to the variable on the lefthand side.
This creates combinational logic.

The second is referred to as a nonblocking assignment.
It evaluates the righthand side.
It assigns the value to the variable on the lefthand side only a small delay after the entire always block has been executed.
Until then the previous value of the variable is used.
This creates sequential logic.

A begin-end block has the following syntax:

```
begin
    procedural statement
    . . .
    procedural statement
end

begin: block-name
    variable decparations
    parameter declarations
    procedural statement
    . . .
    procedural statement
end
```

Earlier we had the module:

```
module VrAlarmCkt (input P,N,X,W,D,G,
        output A);
    wire S,NS,NX,OA;
    or U1 (A,P,OA);
    and U2 (OA,N,NX,NS);
    not U3 (NX,X);
    not U4 (NS,S);
    and U5 (S,W,D,G);
endmodule
```

Using procedural code for the same module:

```
module VrAlarmCktB (input P,N,X,W,D,G,
        output reg A);
    always @(P,N,X,W,D,G)
        begin : Ablk
        reg S;
            S = W&D&G;
            A = P | (N & ~X & ~S);
        end
endmodule
```

Note that the output A has to be defined as a reg since its value is assigned by procedural code.

Syntax of an if statement:
if(condition) procedural-statement

if(condition) procedural-statement
else procedural-statement

The first procedural statement is executed if the condition is true (1'b1).
Else (inculding the case where the condition evaluates to x), the second procedural statement is executed.

Example: identifying prime numbers in the range [1:16].

```
module Vrprimei(N,F);
input [3:0] N;
output reg F;
parameter OneIsPrime=1;
always @(*)
    if(N==1) F=OneIsPrime;
    else if((N%2)==0)
        begin
            if(N==2) F=1;
            else F=0;
        end
    else if(N<=7) F=1;
    else if((N==11)||(N==13)) F=1;
    else F=0;
endmodule
```

A case statement can be used when nested if statements test the same variable.

Syntax:

```
case(selection-expression)
    choice, · · ·, choice: procedural-statement
    . . .
    choice, · · ·, choice: procedural-statement
    default: procedural-statement
endcase
```

The selection expression is evaluated.

The choices are compared one by one to the value of the selection expression.

Only the first match is considered. The corresponding procedural statement is executed.

If a case statement omits some of the possible choices, the synthesis tool will infer latches to hold the previous values of the outpts for the missing choices.
It is therefore important to cover all the choices.

Example: A module that selects between A, B and C based on the value of a select input called sel.

```
module Vrbytecase(A,B,C,sel,Z);
input [7:0] A,B,C;
input [1:0] sel;
output reg [7:0] Z;
    always @(*)
        case (sel)
            2'd0: Z=A;
            2'd1: Z=B;
            2'd2: Z=C;
            2'd3: Z=8'b0;
            default: Z=8'bx;
        endcase
endmodule
```

The default is important in case sel contains x or z bits.

In this case the output will be all-x.

Syntax of a for statement:

```
for(loop-index=first-expr; logical-expression;
    loop-index=next-expr)
        procedural-statement
```

For synthesis to be possible:
first-expr must be a constant.
It must be possible to determine the value of logical-expression at compilation time.
next-expr needs to be limited to simple incrementing or decrementing.

This results in the following syntax:

```
for(loop-index=first-expr; loop-index <= last;
    loop-index=loop-index+1)
        procedural-statement
```

Example: A module that compares two 8-bit inputs X and Y, and assigns $gt = 1$ if $X > Y$.

Using combinational logic, it is necessary to compare $X[i]$ and $Y[i]$ for $i = 0, 1, \cdots, 7$.
If $X[i]Y[i] = 01$, $X < Y$ up to bit i.
If $X[i]Y[i] = 10$, $X > Y$ up to bit i.

$$gt0 = X[0] \cdot Y'[0]$$
$$gt1 = X[1] \cdot Y'[1] + (X'[1] \cdot Y[1])' \cdot gt0$$
$$gt2 = X[2] \cdot Y'[2] + (X'[6] \cdot Y[6])' \cdot gt1$$
$$\cdots$$
$$gt7 = X[7] \cdot Y'[7] + (X'[7] \cdot Y[7])' \cdot gt6$$
$$gt = gt7$$

The order of writing these equations does not affect the logical circuit implementing them.

```
module Vrcomp(X,Y,gt);
input [7:0] X,Y;
output reg gt;
integer i;
    always @(X,Y)
        begin
            gt=0;
            for(i=0;i<=7;i=i+1)
                if(X[i]&~Y[i]) gt=1;
                else if (~X[i]&Y[i]) gt=0;
                // else there is no change to gt
        end
endmodule
```

Other looping statements:
repeat, while, and forever.

Read:

5.10. Functions and Tasks

5.11. The Time Dimension

5.12. Simulation

5.13. Test Benches

5.14. Verilog Features for Sequential Logic Design

5.15. Synthesis