EDA Technology

Chapter 5

Operators and Structural Description Statement

5.1 Operators of Operation

- Verilog is rich in operation operators. According to the number of operands that the operator takes, the operation operators can be divided into the following 3 categories.
- Unary operators: it can take one operand, such as logic inversion "~". For example, ~A .
- Binary operators: it can take two operands, such as AND operation "&". For example, A&B .
- Ternary operators: it can take three operands, such as condition operator "?:" (question mark and colon). For example, s? a:b .

- assign y= s? a:b;
- s? (question mark)
- This statement means:
- If s==1, then y=a;
- Else y=b;

This statement is very important, because it refers to the continuous assignment statement.

5.1.1 Bit Logical Operator

- In addition to the logical inversion operator "~", the bit logical operator belongs to the binary operator.
- The logical operations are performed separately according to bits.

Table Functional descriptions and usage examples of bit logic operators

Logic operator	Logic function	Logic operation results of A and B	Logic operation results of C and D	Logic operation results of C and E
~	Logic inversion	~A = 1'b1	~C = 4'b0011	~E = 6'b101001
|	Logic or	A | B = 1'b 1	C | D = 4'b1111	C | E = 6'b011110
&	Logic and	A & B = 1'b 0	C & D = 4'b1000	C & E = 6'b000100
^	Logic xor	A ^ B = 1'b 1	C ^ D = 4'b0111	C ^ E = 6'b011010
~^ or ^~	Logic xnor	A ~^ B = 1'b 0	C ~^ D = 4'b1000	C ~^ E =6'b100101

Assume: A=1'b0, B=1'b1, C[3:0]=4'b1100, D[3:0]=4'b1011, E[5:0]=6'b010110

5.1.2 Logical Operator

- The operators of logical operation have the following three types.
- Logic AND: &&.
- Logic OR: ||
- Logic INVERSE: !. For example, !A=0.

- "!" belongs to unary operators, "&&" and "||" both belong to binary operator.

- The difference between the logic operation operator and the bit logical operator in the above is that if the operand corresponding to the logical operator is a bit vector, then no matter how many bits, the output after the operation is only 1 bit.

- If A=4’b1001, B=4’b0001, then:
- A && B=(1|0|0|1) & (0|0|0|1)=1&1=1’b1

- Besides, if a vector contains z in addition to 0, it is considered to be logical z and has the following relations.
- 1&z = 1'bz, 0&z = 1'b0, 1|z = 1'b1, 0|z = 1'bz

- A=4'b1001, B=4'b0001
- Firstly, compute 1 | 0 | 0 | 1 =1
- Second, compute 0 | 0 | 0 | 1 =1
- Finally, 1&1 =1

EDA Lab

5.1.3 Arithmetical Operator

Table Arithmetic operator functions and their examples

Logic operator	Function	Instruction	Example
+	Addition		S = A + B = 8'b00011000
-	Subtraction		S = B - A = 8'b11111110
*	Multiplication		S = A * B = 8'b10001111=2'H8F
/	Division	Results: decimal fraction discarded	S = A / 3 = 8'b00000100
%	Remainder	Division to get remainder	S = A % 3 = 8'b00000001

Assume: A[3:0]=4'b1101, B[3:0]=4'b1011, define S as S[7:0]

- All arithmetic operations are performed by unsigned operands, and if they are subtractive operations, the result of output is complemental code.

[Example]

```
module test1 (A,B,C,D,RCD,RAB,RM1,RM2,S,C0,R1,R2);
input [3:0] C,D ;  input signed [3:0] A,B;
output [3:0] RCD;  output [3:0] RAB;
output [7:0] RM1;  output [7:0] RM2;
output [3:0] S;   output C0;  output R1,R2;
 reg [3:0] S ;      reg C0;
 reg [3:0] RCD ;    reg [7:0] RM1 ;
 reg signed [3:0] RAB;  reg signed [7:0] RM2;
 reg R1,R2;
  always@(A,B,C,D) begin
   RCD <= C+D ;   RAB <= A+B;
   RM1 <= C*D ;   RM2 <= A*B;
    {C0,S} <= {1'b0,C} - {1'b0,D};// notice parallel connection operator
      R1  <= (C>D); R2<=(A>B);    end
 endmodule
```

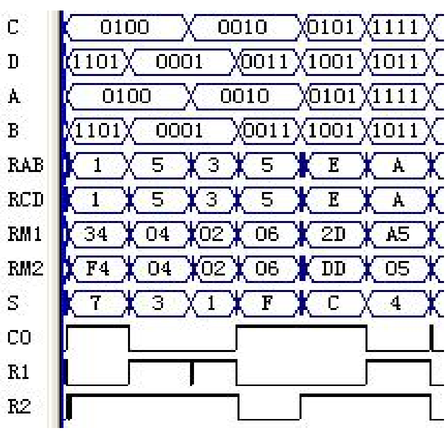


Figure: The simulation waveform of Example

5.1.4 Relational Operator

Table Equality operators and their examples

Equality operator	Meaning	Equality operation example
==	Equal	(3==4)=0，(A==4'b1011)=1，(B==4'b1011)=0
!=	Unequal	(D!=C)=1，(3!=4)=1
===	Identically equal	(D===C)=1，(E===4'b0x10)=0
!==	Not identically equal	(E!==4'b0x10)=1

Assume: A=5'b01011, B=4'b0010, C=4'b0z10, D=4'b0z10, E=3'bx10

Table Inequality operators and their examples

Inequality operator	Meaning	Operation examples
>	Greater than	$(A<B)=0$, $(A>B)=1$ $(A<20)=1$, $(A>12)=1$ $(A>=14)=0$, $(A<=13)=1$
<	Less than	
<=	Less than or equal	
>=	Greater than or equal	

Assume: A=4'B1101, B=4'B0110

Example

```
module BCD_ADDER (A,B,D) ;
   input [7:0] A,B;   output [8:0] D;
   wire [4:0] DT0, DT1 ; reg [8:0] D; reg S;
  always@ (DT0)
    begin  if (DT0[4:0] >= 5'b01010 )
     // If the sum of the low bit BCD codes is greater than or equal to 10, then
     //6 is added to the sum and there is also carry, so that the carry flag S
     //equals to 1.
             begin D[3:0] = (DT0[3:0]+4'b0110); S=1'b1; end
             else  begin D[3:0] = DT0[3:0] ; S=1'b0; end
     end // Otherwise, the low bit value is assigned to the low bit BCD code D[3:0]
         //and outputs without carry, so that the carry flag S is equal to 0.
 always@ (DT1)   begin
    if (DT1[4:0]>=5'b01010)
  begin D[7:4] = (DT1[3:0]+4'b0110); D[8]=1'b1; end
      else begin D[7:4] = DT1[3:0] ; D[8]=1'b0;  end
    end
   assign DT0 = A[3:0] + B[3:0] ;         //Assume there is no carry from the low
                                          //bit
   assign DT1 = A[7:4] + B[7:4] + S;      //S is the carry from the sum of the BCD
                                          //codes of the low bits.
endmodule
```

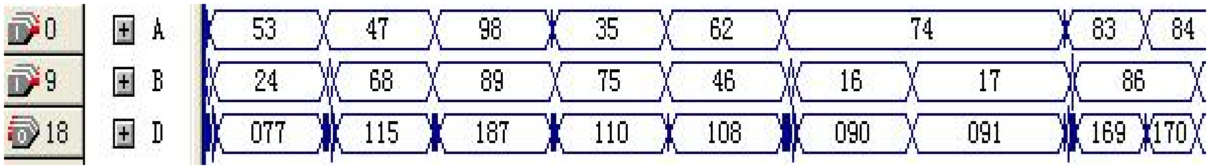


Figure: The simulation waveform of the Example

5.1.6 Contraction Operators

- There are six types of contraction operators, including & (AND), -& (NAND), | (OR), ~| (NOR), ^ (XOR), ^~, ~^ (XNOR). The contraction operator belongs to the unary operator, and the output result of its operation is also one bit.

- For example, if A=8’b11101111, then &A=1&1&1&0&1&1&1&1=0; this is because only when every bit of A is 1, their reduced operation value of AND is 1.

5.1.7 Parallel Connection Operator

- If s1=1'b0, s0=1'b0, then {s1, s0}=2'b00, here the parenthesis "{}" is the parallel connection operator. "{}" can splice two or more signals in binary bits and use them as a data signal.

- {a1, b1, 4{a2,b2}} = { a1, b1, {a2,b2}, {a2,b2}, {a2,b2}, {a2,b2}} = {a1,b1,a2,b2,a2,b2,a2,b2,a2,b2}

5.1.8 Shift Operator

- ">>" is a rightward shift operator, "<<" is a leftward shift operator, and their general formats are given as follows:
- V >> n or V << n

- The data in the operands or variables V is shifted to the right or left by n bits.
- For example, if V=8'b11001001, then:
- the value of V>>1 is 8'b01100100
- the value of V<<3 is 8'b01001000

- “>>>” is as the right shift operator, “<<<” is as the left shift operator. Their general form are given as follows:
- V >>> n or V <<< n

- The above expressions mean that the data (signed number) in the operand or variable V is shifted to the right or left by n bits. And for the right shift operation, the symbol bit, that is, the highest position, is filled with the removed bits, and the left shift operation is the same with the ordinary left shift operator “<<”.

output signed[7:0] y;
input signed[7:0] a;
assign y = (a<<<2);
if a=10101011,then y=10101100 is output
if a=10001111,then y=00111100 is output
	parameter C=8'sb10101011;
	parameter D=8'sb01001110;
	output [7:0] Y1,Y2;
	assign Y1=(C>>>2); //result: Y1=11101010
	assign Y2=(D>>>2); //result: Y2=00010011

5.1.9 Example of Shift Operator

[Example]

```
module MULT4B(R,A,B);
  parameter S=4;
  output[2*S:1] R ;
  input[S:1] A,B ;
   reg[2*S:1] R;
     integer i;
     always @(A or B)
      begin
      R = 0 ;
      for(i=1; i<=S; i=i+1)
      if(B[i]) R=R+(A<<(i-1));
      end
endmodule
```

[Example]

```
module MULT4B (R,A,B);
  parameter S=4;
  output[2*S:1] R;
   input[S:1] A,B;
  reg[2*S:1] R,AT;
  reg[S:1] BT,CT;
  always @(A,B )
    begin
    R=0; AT = {{S{1'B0}},A};
    BT = B;  CT = S;
    for(CT=S; CT>0; CT=CT-1)
      begin if(BT[1]) R=R+AT;
       AT = AT<<1;  BT = BT>>1;
     end
 end   endmodule
```

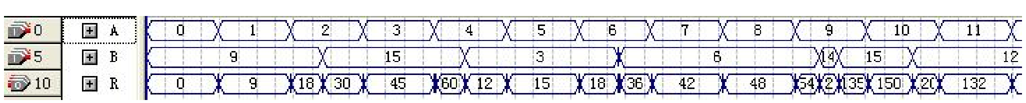


Figure: The timing simulation of 4-bit multiplier

⌘Shifter

[Example]	[Example]

`module SHIF4(DIN,CLK,RST,DOUT);`
` input CLK,DIN,RST;`
` output DOUT;`
` reg [3:0] SHFT;`
` always@(posedge CLK or posedge RST)`
`     if(RST) SHFT<=4'B0;`
`     else begin  SHFT[3]<=DIN;`
`       SHFT[2:0] <= SHFT[3:1];`
`     end`
`    assign DOUT=SHFT[0];`
`endmodule`
	`module SHIF5 (DIN,CLK,RST,DOUT);`
	` input CLK,DIN,RST;  output DOUT;`
	` reg [3:0] SHFT;`
	` always@(posedge CLK or posedge RST)`
	`  if(RST) SHFT<=4'B0;`
	`     else begin`
	`       SHFT  <= (SHFT >> 1);`
	`       SHFT[3] <= DIN;`
	`     end`
	`  assign DOUT = SHFT[0];`
	`endmodule`

5.1.10 Conditional Operator

- The general format of the conditional operator usage is given as follows:
- conditional expression ? expression 1: expression 2

[Example]

```
module DFF2 (input CLK, input D, input RST , output reg Q );
    always @(posedge CLK )
      Q <= RST ? 1'b0 : D;
endmodule
```

5.2 Continual Assignment Statement

- `assign   target variable name = drive expression;`

- When any signal variable in the driving expression on the right side of the equal sign changes, the expression is calculated once and the obtained data is immediately assigned to the target variable marked by the variable name on the left side of the equal sign.

- `assign [delay] target variable name = drive expression;`
- 'timescale 10ns/100ps;
- assign #6 R1 = A & B;

- #:number sign

【Example】

```
module MUX41a (A,B,C,D,S1,S0,Y);
    input A,B,C,D,S1,S0;
   output Y;
    assign AT = S0 ? D : C ;
    assign BT = S0 ? B : A ;
    wire  Y = (S1 ? AT : BT);
endmodule
```

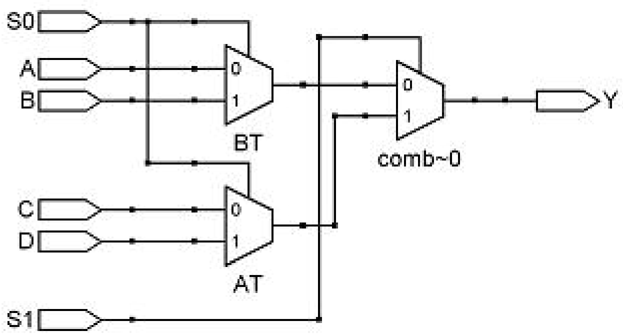


Figure: The RTL diagram of the Example

wire Y= (S1? AT: BT);   is equal to
wire Y; assign Y= S1? AT: BT

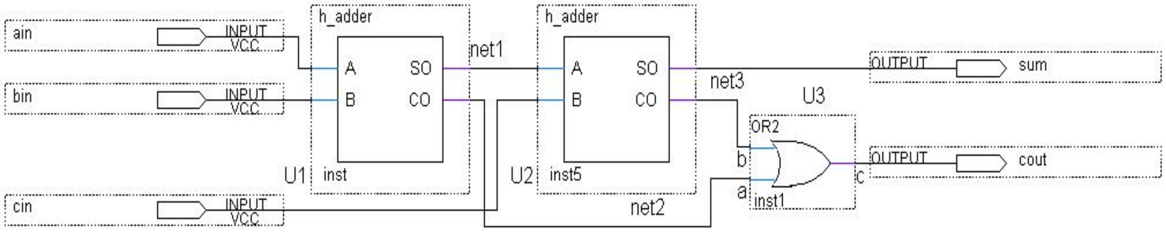
5.3 Instantiation Statement

5.3.1 Half-adder Design

```
module h_adder (A,B,SO,CO);
    input A,B;
    output SO,CO;
     assign SO = A ^ B;     // After the XOR logic is executed between variables
                            //A and B, the result is assigned to the output signal
                            //SO.
     assign CO = A & B;     //After the AND logic is executed between variables
                            //A and B, the result is assigned to the output signal
                            //CO.
 endmodule
```

5.3.2 Full-adder Design

```
module f_adder(ain,bin,cin,cout,sum);
    output cout,sum;
     input ain,bin,cin;
    wire  net1,net2,net3;
    h_adder  U1(  ain, bin, net1, net2);
    h_adder  U2(.A(net1), .SO(sum), .B(cin), .CO(net3) );
         or  U3(cout, net2, net3);
 endmodule
```



5.3.3 Verilog Instantiation Statement and Its Usage

1. Port name correlation method of instantiation statement

- the general format of the commonly used port name correlation method is as follows:
- < module component name > <instantiated component name > (.instantiation component port (instantiation element external port name),...);
- h_adder U2(.A(net1), .SO(sum), .B(cin),.CO(net3));
- h_adder U2(.B(cin), .CO(net3), .A(net1), .SO(sum));

2. Instantiation statement location correlation method

- There is also a corresponding way of linking expression called “location correlation method”. The so-called location correlation is to connect the corresponding ports based on the relevant position.
- The location of signal is very important and cannot be misplaced.

- h_adder (A, B, SO, CO) in Example 5-9 can no longer be changed to module h_adder (A, B, CO, SO).

5.4 Application of Parameter Transmission Statement

```
module MULT4B(R,A,B);
  parameter S=4;
  output[2*S:1] R ;
  input[S:1] A,B ;
   reg[2*S:1] R;
     integer i;
     always @(A or B)
      begin
      R = 0 ;
      for(i=1;  i<=S; i=i+1)
      if(B[i])  R=R+(A<<(i-1));
      end
endmodule
```

```
module MULT4B (R,A,B);
  parameter S=4;
  output[2*S:1] R;
   input[S:1] A,B;
  reg[2*S:1] R,AT;
  reg[S:1] BT,CT;
  always @(A,B )
    begin
    R=0; AT = {{S{1'B0}},A};
    BT = B;  CT = S;
    for(CT=S; CT>0; CT=CT-1)
      begin  if(BT[1]) R=R+AT;
       AT = AT<<1;  BT = BT>>1;
     end
 end   endmodule
```

- To achieve this goal, the expression way of *parameter* in Example 5-3 should be firstly rewritten. That is to say, the top two statements in the example module MULT4B (R, A, B) and parameter S=4 are only needed to be rewritten into the following forms:

- module MULT4B #(parameter S=4)(R,A,B);
- or: module MULT4B #(parameter S)(R,A,B);

- #number sign

Bottom design	Top layer design

`module MULT4B #(parameter S)(R,A,B);`
`output[2*S:1] R ;`
`input[S:1] A,B ;`
`reg[2*S:1] R;   integer i;`
`...     //The following is the same as example 5-3`
	`module MULTB (RP,AP,BP);`
	`output[15:0] RP ;`
	`input[7:0] AP,BP ;`
	`MULT4B #(.S(8))`
	`U1(.R(RP), .A(AP), .B(BP) );`
	`endmodule`

- For example, if the module statements and parameters of the original underlying file are expressed as:
- module SUB_E
-     #(parameter S1=4, parameter S2=5, parameter S3=2)(A,B,C);
- then in the instantiation statement, a similar statement should be made as follows:
- SUB_E #(.S1(8), .S2(9), .S3(7)) U1(.C(CP), .A(AP), .B(BP));

- In Verilog, there is also a parameter transmission statement similar to *parameter* function, that is, *defparam*. Its detailed usage will be introduced in Chapter 6 through examples.

5.5 Structural Description with Library Component

- Gate level components can be divided into 3 categories: multiple input gates, multiple output gates, and three-state gates. There are 12 most commonly used gates, and their functions and keywords include:
- (1) There are 6 multiple input gates: AND gate *and*, NAND gate *nand*, OR gate *or*, NOR gate *nor*, XOR gate *xor*, XNOR gate *xnor*.
- (2) There are 2 multiple output gates: buffer gate *buf*, NOT gate *not*.
- (3) There are 4 three-states gates: three-state gate with high level enabling *bufif1*, three-state gate with low level enabling *bufif0*, three-state non-gate with low level enabling *notif0*, three-state non-gate with high level enabling *notif1*.

```
module LOGICGATE (input A,B,C,S , output OUT);
wire a1,a2,a3,a4;
    not u1 (a1,B);
    and u2 (a2,A,a1);
    or  u3 (a3,C,B);
    xor u4 (a4,a3,a2);
 notif1 u5 (OUT,a4,S);
endmodule
```

- The format of invoking gate element is:
- Component name of basic gate <gate instantiation name> (<Port correlation list >)

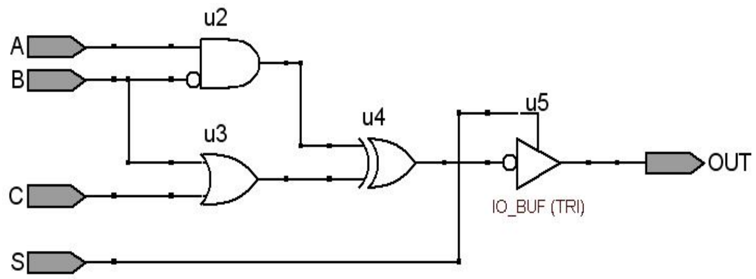


Figure: The logic circuit described in Example 5-13

- The instantiation statements of 3-input AND gate and 2-input AND gate are as follows:
- and U1 (out,in1,in2,in3); //3-input AND gate, and the instantiation name is U1
- and U2 (out,in1,in2); //2-input AND gate, and the instantiation name is U2
- For the three-state gate, the input/output ports are listed in the following order, for example:
- bufif1 U1(out,in,enable); //three-state gate with high level enabling
- bufif2 U2(out,a,ctr1); //three-state gate with low level enabling
- As for the invoking of two components *buf* and *not*, it should be noted that they allow multiple outputs, but only one input, for example:
- not IC1 (out1,out2,in); //1 input in, 2 output out1,out2
- buf IC2 (out1,out2, out3,in); //1 input in, 3 output out1,out2, out3

5.6 Compiling Directive Statement

- In the expression way of program, the compiling directive statements and the macro names that have been defined begin with the symbol "'".
- Verilog provides multiple compiling directive statements, such as macro definition statement *'define*, conditional compilation statement *'ifdef*, *'else*, *'endif*, *'restall*, etc.
- The most commonly used statements are *'define*, *'include*, *'ifdef*, *'else* and *'endif*.

- ': apostrophe

5.6.1 Macro Definition Statement

- The general usage format of *'define* statement is:
-     'define macro name (identifier) macro content (string)

- 'define s   A+B+C+D
- “assign DOUT='s + E”   is equivalent to the statement “assign DOUT = A+B+C+D+E;”.

- The specific application of *'define* should also be noted that:
- (1) A semicolon is not added to the macro definition statement at the end of the line.
- (2) When a defined macro name is quoted in a program, the symbol "'" must be added to the identifier that defines the macro name to show that the identifier is a macro definition name.

5.6.2 File Inclusive Statement, *'include*

- The function of the file inclusive statement *'include* is to include all of a file in another file, and its format is:
- 'include "file name"

- "": double quotation marks

```
'include "h_adder.v"
'include "or2a.v"
 module f_adder(input ain,bin,cin,output cout,sum );
   wire  e,d,f ;
 h_adder  u1(  ain, bin,  e,  d );
 h_adder  u2(.a(e), .so(sum), .b(cin),.co(f) );
     or2a  u3(.a(d), .b(f),    .c(cout) );
 endmodule
```

- When using a file inclusive statements, it should be noted that:
- (1) a *'include* statement can only specify a contained file, giving full name and suffix in the statement.
- (2) The *'include* statement can appear anywhere in the program.
- (3) If the included file is not in the folder where the current project is located, it must indicate the path of the file. For example, 'include "e:/ADDER/h_adder.v".
- (4) The file inclusion of the *'include* statement allows multilevel inclusions. For example, file 1 contains file 2, file 2 contains file 3, etc.
- (5) Different compilers and synthesizers have different requirements on *'include* statements, so they need to be treated differently.

5.6.3 Conditional Compilation Statement, *'ifdef, 'else, 'endif*

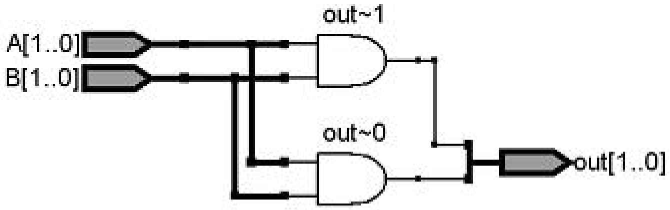
- The function of conditional compilation command statement *'ifdef, 'else* and *'endif* is to direct synthesizer to make the part specified in the statement participate in the Verilog source program and be compiled and synthesized simultaneously.

Format 1 of conditional compilation command statement	Format 2 of conditional compilation command statement
'ifdef macro name statement block 'endif	'ifdef macro name Statement block1 'else statement block2 'endif

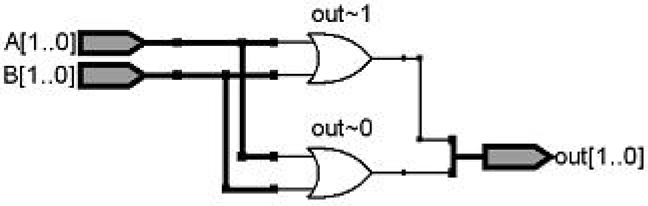
Format 1: If the macro name is defined in the program, the statement block is executed.

Format 2: If the macro name is defined in the program, the statement block 1 is executed, otherwise statement block2 is executed.

```
'define AND
module andd (out,A,B);
input[1:0] A,B;
output [1:0] out;
 'ifdef AND
 assign out=A&B;
'else assign out=A|B;
 'endif
endmodule
```




```
'define OR1
module andd (out,A,B);
input[1:0] A,B;
output [1:0] out;
 'ifdef AND
 assign out=A&B;
'else assign out=A|B;
 'endif
endmodule
```



5.7 Application of Attribute of Keep

- Sometimes the designer hopes that, without increasing the signal connection that is not related to the design, the signal changes in a data channel defined within the module can also be understood in detail in the simulation, such as the signal net3 in Example 5-10.

- However, because this signal is a temporary signal or data channel inside the module, it is simplified and removed after the logic synthesis and optimization, so the signal cannot be found in the simulation signal, and cannot be observed in the simulation waveform.

- The *keep* attribute can be used to solve this problem.

```
module ff_adder(ain,bin,cin,cout,sum);
    output cout,sum ;     input ain,bin,cin ;
    (* synthesis, keep *)  wire  net3 ;
    wire  net2,net1 ;
...     //The following is the same as that of example 5-10
```

- (* synthesis, keep *) or (* synthesis, probe_port, keep *)

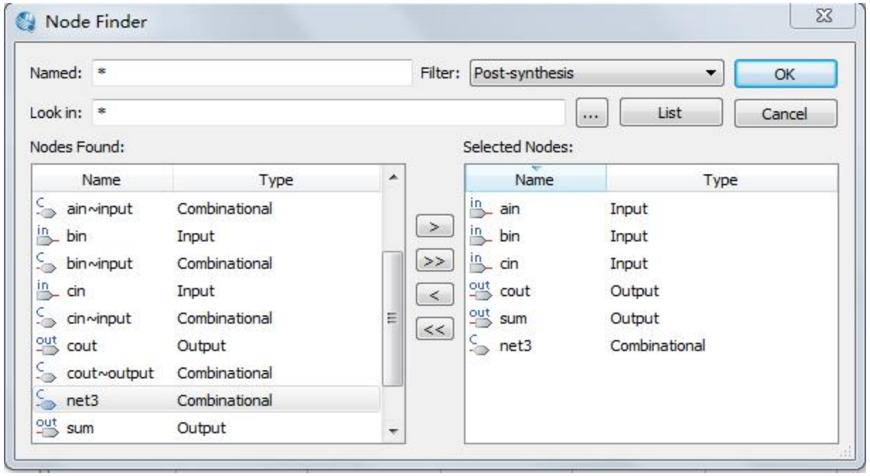


Figure: The addition of the simulation testing signal net3

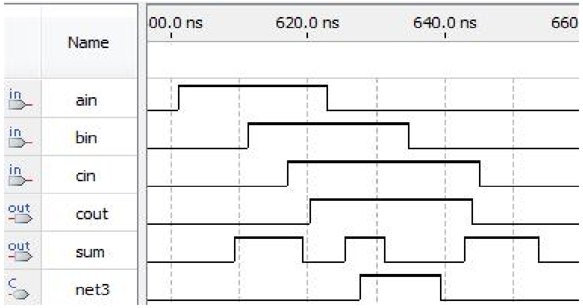


Figure: The simulation waveform of Example 5-17

- (* synthesis, probe_port, keep *) wire net3;
- For vector signals, such as A[7:0], it can be defined as follows:
- (* synthesis, probe_port, keep *) reg [7:0] A ;

5.8 Usage of SingalProbe

- In the process of hardware testing for FPGA development projects, in order to understand one or some of the signals within a design, the usual way is to add some external elicited ports, and to bring these internal signals to the outside for testing. These pin settings are deleted after the end of the test.
- However, the disadvantage of this approach is that the layout of the original design has been changed when leading the pin only to use for testing, and the system function after the deletion of these pins may not be able to return to the original functional structure.
- For this purpose, the SignalProbe signal detection function of Quartus II can be used to extract the internal signals needed by users from the FPGA, using the idle connections and ports in the FPGA without changing the original design layout.

- This function is different from the use of the *keep* attribute. Using the *keep* attribute simply tells the synthesizer not to optimize a signal, so that it can be invoked to observe in the simulation file. The use of SignalProbe detection function is to transmit the specified internal signals which does not belong to the port to the external of the device for testing. Of course, sometimes it must be combined with the application of *keep* attribute, so that SignalProbe can measure some internal signals that may be optimized on the device port.

- **1. Completing the design simulation and hardware test according to the routine process**
- **2. Setting up SignalProbe Pins**
- **3. Compiling SignalProbe Pins test information, downloading and testing**

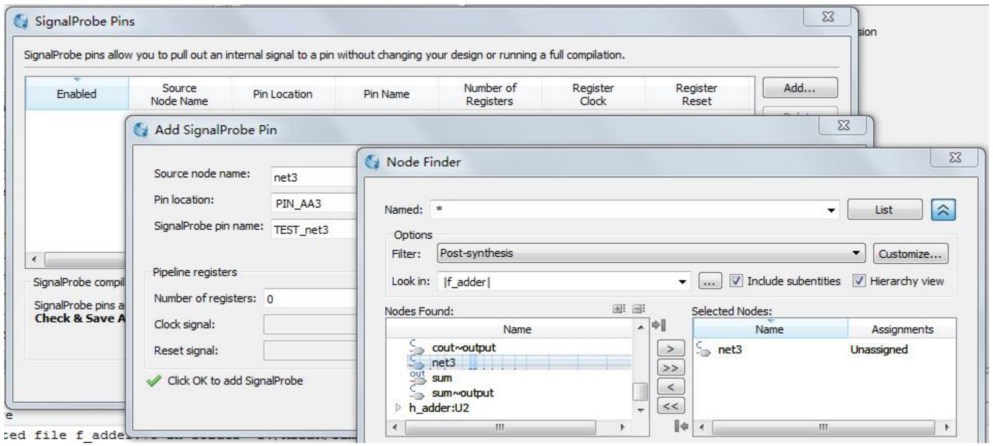


Figure: Setting the probe signal net3 in the SignalProbe dialog box

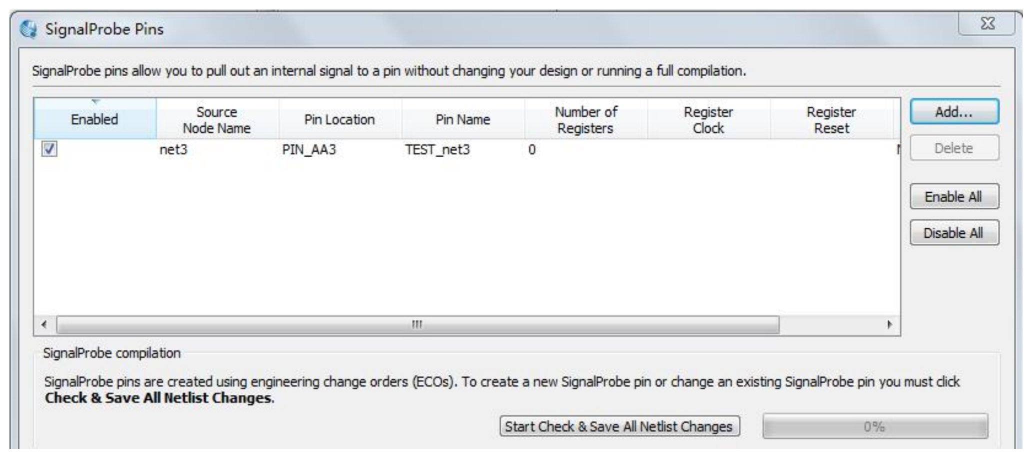


Figure: The settings of SignalProbe Pins dialog box