5

Figure 5-0
Example 5-0
Syntax 5-0
Table 5-0

Assignments

The assignment is the basic mechanism for getting values into nets and registers. There are two basic forms of the assignment:

- the *continuous assignment*, which assigns values to *nets*
- the *procedural assignment*, which assigns values to *registers*

An assignment consists of two parts, a left-hand side and a right-hand side, separated by the equal (=) character. The right-hand side can be any expression that evaluates to a value. The left-hand side indicates the variable that the right-hand side is to be assigned to. The left-hand side can take one of the following forms, depending on whether the assignment is a continuous assignment or a procedural assignment.

Statement type	Left-hand side
continuous assignment	net (vector or scalar) constant bit-select of a vector net constant part-select of a vector net concatenation of any of the above 3
procedural assignment	register (vector or scalar) bit-select of a vector register constant part-select of a vector register memory element concatenation of any of the above 4

Table 5-1: Legal left-hand side forms in assignment statements

5.1
Continuous Assignments

Continuous assignments drive values onto nets, both vector and scalar. The significance of the word “continuous” is that the assignment occurs whenever simulation causes the value of the right-hand side to change. Continuous assignments provide a way to model combinational logic without specifying an interconnection of gates. Instead, the model specifies the logical expression that drives the net. The expression on the right-hand side of the continuous assignment is not restricted in any way. It can even contain a reference to a function. Thus, the result of a `case` statement, `if` statement, or other procedural construct can drive a net.

The syntax for continuous assignments is as follows:

```
<net_declaration>
        ::= <NETTYPE> <expandrange>? <delay>? <list_of_variables> ;
        ||= trireg <charge_strength>? <expandrange>? <delay>? <list_of_variables> ;
        ||= <NETTYPE> <drive_strength>? <expandrange>? <delay>?
                        <list_of_assignments> ;

<continuous_assign>
        ::= assign <drive_strength>? <delay>? <list_of_assignments> ;

<expandrange>
        ::= <range>
        ||= scalared <range>
        ||= vectored <range>

<range>
        ::= [ <constant_expression> : <constant_expression> ]

<list_of_assignments>
        ::= <assignment> <,<assignment>>*

<charge_strength>
        ::= ( small )
        ||= ( medium )
        ||= ( large )

<drive_strength>
        ::= ( <STRENGTH0> , <STRENGTH1> )
        ||= ( <STRENGTH1> , <STRENGTH0> )
```

Syntax 5-1: Syntax for <net_declaration>

5.1.1
The Net Declaration Assignment

The first two alternatives in the <net_declaration> are discussed in Chapter 3, *Data Types* (see Section 3.2.3). The third alternative, the net declaration assignment, allows a continuous assignment to be placed on a net in the same statement that declares that net. The following is an example of the <net_declaration> form of a continuous assignment:

```
wire (strong1, pull0) mynet = enable;
```

Please note: Because a net can be declared only once, only one net declaration assignment can be made for a particular net. This contrasts with the continuous assignment statement; one net can receive multiple assignments of the continuous assignment form.

5.1.2
The Continuous Assignment Statement

The <continuous_assign> statement places a continuous assignment on a net that has been previously declared, either explicitly by declaration or implicitly by using its name in the terminal list of a gate, a user-defined primitive or module instance. The following is an example of a continuous assignment to a net that has been previously declared:

```
assign (strong1, pull0) mynet = enable;
```

Assignments on nets are continuous and automatic. This means that whenever an operand in the right-hand side expression changes value during simulation, the whole right-hand side is evaluated and assigned to the left-hand side.

The following is an example of the use of a continuous assignment to model a four bit adder with carry. Note that the assignment could not be specified directly in the declaration of the nets because it requires a concatenation on the left-hand side.

```
module adder (sum_out, carry_out, carry_in, ina, inb) ;
output [3:0]sum_out;
input [3:0]ina, inb;
output carry_out;
input carry_in;
wire carry_out, carry_in;
wire[3:0] sum_out, ina, inb;
   assign
       {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

Example 5-1: Use of continuous assign statement

The following example describes a module with one 16-bit output bus. It selects between one of four input busses and connects the selected bus to the output bus.

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
    parameter n = 16;
    parameter Zee = 16'bz;
    output [1:n] busout;
    input [1:n] bus0, bus1, bus2, bus3;
    input enable;
    input [1:2] s;


    tri [1:n] data;                          // net declaration.
    tri [1:n] busout = enable ? data : Zee;// net declaration with
                                             // continuous assignment.
    assign                                 // assignment statement with
        data = (s == 0) ? bus0 : Zee,      // 4 continuous assignments.
        data = (s == 1) ? bus1 : Zee,
        data = (s == 2) ? bus2 : Zee,
        data = (s == 3) ? bus3 : Zee;
endmodule
```

Example 5-2: Net declaration assignment and continuous assign statement

The following sequence of events is experienced during simulation of the description in Example 5-2:

1. The value of `s`, a bus selector input variable, is checked in the assign statement; based on the value of `s`, the net `data` receives the data from one of the four input busses.
2. The setting of `data` triggers the continuous assignment in the net declaration for `busout`; if `enable` is *set*, the contents of `data` are assigned to `busout`; if `enable` is *clear*, the contents of `Zee` are assigned to `busout`.

Note that the parameter `Zee` has an *explicit* width specification on the high impedance value. This is recommended practice, because it avoids mistakes where extra bits of a value would cause erroneous results. The default width of the high-impedance value (`z`) is the word size of the host machine, typically 32 bits.

Please note: There is a functional difference between a net declaration assignment and a continuous assignment statement. In net declaration assignments, all changes during a time unit in the expression on the right-hand side of the assignment operator (=) propagate to the net. In continuous assignment statements, the value in the expression on the right-hand side of the assignment operator (=) propagates to the net after the final change to the value of the expression.

5.1.3 Delays

A delay given to a continuous assignment specifies the time duration between a right-hand side operand value change and the assignment made to the left-hand side. If the left-hand side references a scalar net, then the delay is treated in the same way as for gate delays—that is, different delays can be given for the output rising, falling, and changing to high impedance (see Chapter 6, *Gate and Switch Level Modeling*).

If the left-hand side references a vector net, then up to three delays can also be applied. The following rules determine which delay controls the assignment:

- If the right-hand side LSB is non-zero or becomes zero, then the falling delay is used.
- If the right-hand side LSB is `z` or becomes `z`, then the turn-off delay is used.
- If the right-hand side LSB is a one or becomes a one, then the rising delay is used.
- If the right-hand side LSB is an `x` or becomes an `x`, then the lesser of the delay values is used.

When different rise and fall delays are specified for a vector net, the actual delay choosen is based on the value or value change of the least significant bit. An example of this is shown in Example 5-3.

```
module least_significant_bit (out);
output [3:0] out;
reg [3:0] a;
wire [3:0] b;

  assign #(10,20) b = a;

  initial
      begin
           a = 'b0000;
      #100 a = 'b1101;
      #100 a = 'b0111;
      #100 a = 'b1110;
      end

  initial
      begin
      $monitor($time, , "a=%b, b=%b",a, b);
      #1000 $finish;
      end
endmodule

Compiling source file
Highest level modules:
least_significant_bit

   0 a=0000, b=xxxx
  20 a=0000, b=0000
 100 a=1101, b=0000
 110 a=1101, b=1101 // LSB is high so uses a rise delay
 200 a=0111, b=1101
 210 a=0111, b=0111 // LSB is high so uses a rise delay
 300 a=1110, b=0111
 320 a=1110, b=1110 // LSB is low so uses a fall delay
```

Example 5-3: Delay based on the value or value change of the least significant bit

In order to model rise and fall delay for individual bits, you need to expand the register expression to a single bit expression as shown in Example 5-4.

```
module least_significant_bit (out);
output [3:0] out;
reg [3:0] a;
wire [3:0] b;
   assign #(10,20) b[0] = a[0],
                   b[1] = a[1],
                   b[2] = a[2],
                   b[3] = a[3];

   initial
      begin
           a =‘b0000;
      #100 a =’b1101;
      #100 a =’b0111;
      #100 a =’b1110;
       end

   initial
      begin
         $monitor($time, , “a=%b, b=%b”,a, b);
         #1000 $finish;
      end

endmodule
```

```
Compiling source file
Highest level modules:
least_significant_bit

 0 a=0000, b=xxxx
 20 a=0000, b=0000
 100 a=1101, b=0000
 110 a=1101, b=1101  // rise delay of 10 time units
 200 a=0111, b=1101
 210 a=0111, b=1111  // rise delay of 10 time units
 220 a=0111, b=0111  // fall delay of 20 time units
 300 a=1110, b=0111
 310 a=1110, b=1111  // rise delay of 10 time units
 320 a=1110, b=1110  // fall delay of 20 time units
```

Example 5-4: Delay based on the rise and fall delay for individual bits

Note that specifying the delay in a continuous assignment that is part of the net declaration is different from specifying a net delay and then making a continuous assignment to the net. A delay value can be applied to a net in a net declaration, as in the following example:

```
wire #10 wireA;
```

This syntax, called a *net delay*, means that any value change that is to be applied to `wireA` by some other statement is delayed for ten time units before it takes effect. When there is a continuous assignment in a declaration, the delay is part of the continuous assignment and is *not* a net delay. Thus, it is not added to the delay of other drivers on the net. Furthermore, if the assignment is to an expanded vector net (a net not specified with the keyword `vectored`), then the rising and falling delays are not applied to the individual bits if the assignment is included in the declaration.

In situations where a right-hand side operand changes before a previous change has had time to propagate to the left-hand side, then the latest value change is the only one to be applied. That is, only one assignment occurs. This effect is known as *inertial delay*.

The following example implements a vector exclusive `OR`. The size and delay of the operation are controlled by parameters, which can be changed when instances of this module are created. See Section 12.2 for details on *Overriding Module Parameter Values*.

```
module modxor(axorb, a, b);
   parameter size = 8, delay = 15;
   output [size-1:0] axorb;
   input [size-1:0] a, b;
   wire [size-1:0] #delay axorb = a ^ b;
endmodule
```

Example 5-5: Use of delays with assignments

5.1.4 Strength

The driving strength of a continuous assignment can be specified by the user. This applies only to assignments to scalar nets of the types listed below:

```
wire      wand      tri       trireg
          wor       triand    tri0
                    trior     tri1
```

Continuous assignments driving strengths can be specified in either a net declaration or in a stand-alone assignment, using the `assign` keyword. The strength specification, if provided, must immediately follow the keyword (either the keyword for the net type or the `assign` keyword) and must precede any delay specified. Whenever the continuous assignment drives the net, the strength of the value will simulate as specified.

A <drive_strength> specification contains one strength value that applies when the value being assigned to the net is `1` and a second strength value that applies when the assigned value is `0`. The following keywords specify the strength value for an assignment of `1`:

```
supply1  strong1  pull1  weak1  highz1
```

The following keywords specify the strength value for an assignment of `0`:

```
supply0  strong0  pull0  weak0  highz0
```

The order of the two strength specifications is arbitrary. The following two rules constrain the use of drive strength specifications:

- The strength specifications (`highz1`, `highz0`) and (`highz0`, `highz1`) are illegal language constructs.
- When the keyword `vectored` is specified together with a specification of strength on a continuous assignment, the keyword `vectored` is ignored.

5.2 Procedural Assignments

The primary discussion of procedural assignments is in Section 8.2. However, a description of the basic ideas here will highlight the differences between continuous assignments and procedural assignments.

As stated above, continuous assignments drive nets in a manner similar to the way gates drive nets. The expression on the right-hand side can be thought of as a combinatorial circuit that drives the net continuously. The word continuous is important; continuous assignments cannot be disabled.

In contrast, procedural assignments put values in registers. The assignment does not have duration; instead, the register holds the value of the assignment until the next procedural assignment to that register.

Procedural assignments occur within procedures such as `always`, `initial`, `task`, and `function` (these procedures are described in later chapters), and can be thought of as "triggered" assignments. The trigger occurs when the flow of execution in the simulation reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements. Event controls, delay controls, `if` statements, `case` statements, and looping statements can all be used to control whether assignments get evaluated. Chapter 8, *Behavioral Modeling*, gives details and examples.

5.3 Accelerated Continuous Assignments

This section describes how you can accelerate continuous assignments to make your designs simulate faster. This chapter also explains the following:

- the restrictions on accelerated continuous assignments
- how to accelerate continuous assignments
- the kinds of designs that simulate faster with this feature and the kind of design that simulates slower
- how accelerated continuous assignments affect simulation

5.3.1 The Restrictions on Accelerated Continuous Assignments

You can accelerate continuous assignments only if they meet the restrictions described in this section. These restrictions apply to the following syntax elements of a continuous assignment statement:

- the types of nets on the left-hand side of the assignment operator
- the operators and operands in the expressions on the right-hand side of the assignment operator
- the contents and use of a delay expression

Left-hand side restrictions

You can accelerate a continuous assignment if it assigns a value to one of the following types of nets:

- a scalar net
- a expanded vector net that contains less than 64 bits
- a bit-select of an expanded vector net
- a part-select that is less than 64 bits of an expanded vector net

You can also accelerate a continuous assignment if it assigns a value to a concatenation of these types of nets, provided that the concatenation contains fewer than 64 bits.

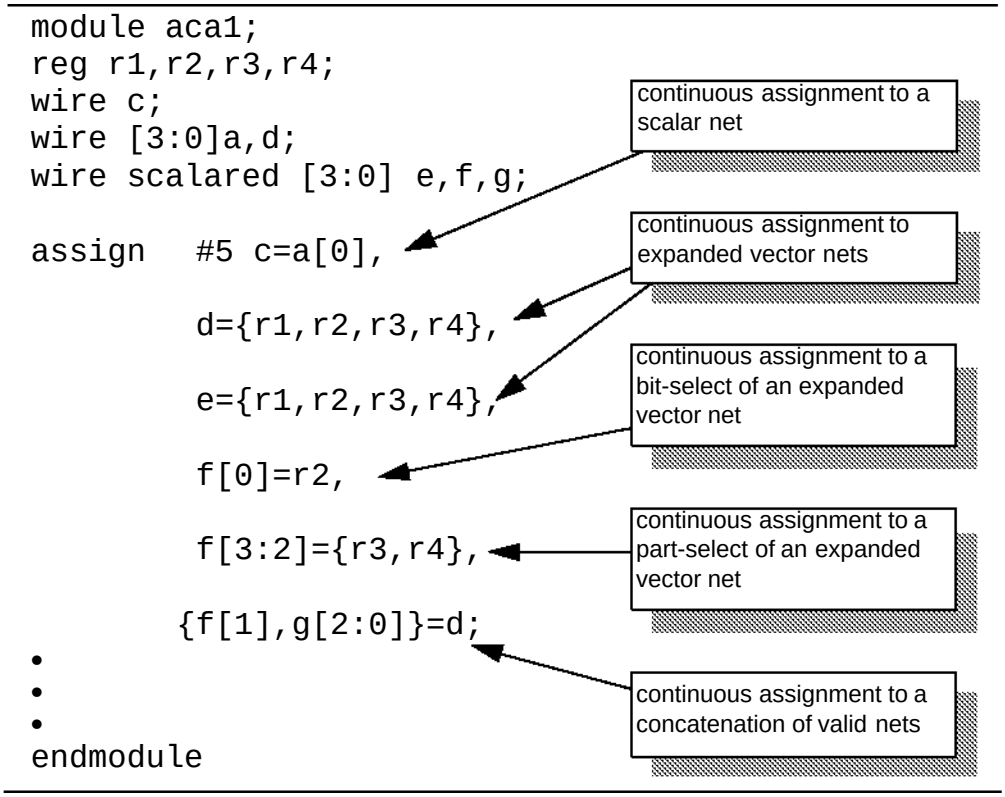
An expanded vector net is a vector net that Verilog-XL converts to a group of scalar nets. This group contains one scalar net for each bit of the vector net. Verilog-XL automatically converts or “expands” a vector net for a number of reasons, which include the following:

- to handle bit-selects and part-selects
- to improve performance and to accelerate continuous assignments

You can require Verilog-XL to expand a vector net by including the keyword `scalared` in the net's declaration.

An unexpanded vector net is a vector net that Verilog-XL does not convert to scalar nets. You can prevent Verilog-XL from expanding a vector net by including the keyword `vectored` in its declaration.

Example 5-6 shows continuous assignments that you can accelerate because the left-hand side meets these restrictions.



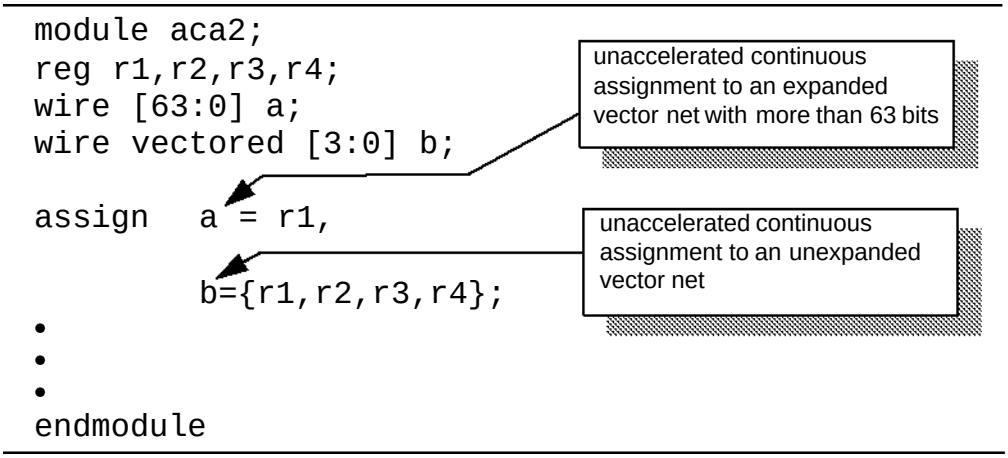
Example 5-6: Left-hand sides of continuous assignments that can be accelerated

Verilog-XL cannot accelerate a continuous assignment to the following types of vector nets:

- vector nets with 64 or more bits
- vector nets declared with the keyword `vectored`

To accelerate a continuous assignment to a vector net, Verilog-XL must expand that vector net. If you declare a vector net with the keyword `vectored`, Verilog-XL cannot accelerate a continuous assignment to it.

Example 5-7 shows continuous assignments that you cannot accelerate because the left-hand side does not meet these restrictions.



Example 5-7: Left-hand side of continuous assignments that cannot be accelerated

Right-hand side restrictions

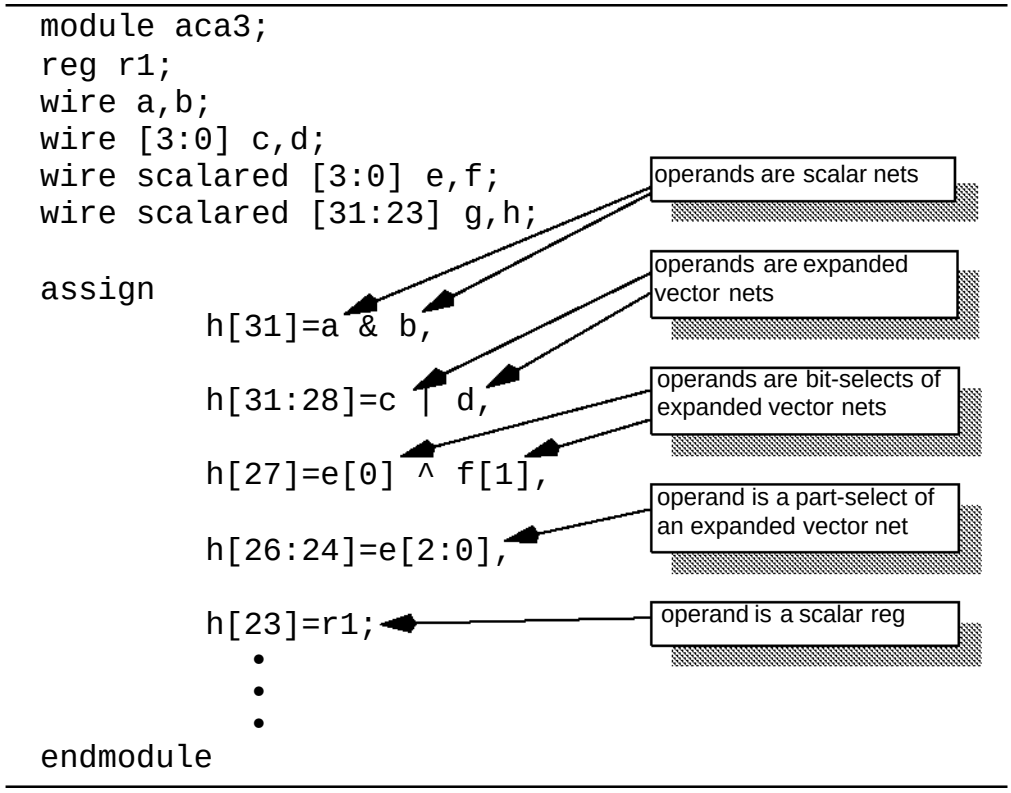
You can accelerate a continuous assignment only if the expression on the right-hand side contains certain operands and operators.

The right-hand expression of a continuous assignment can contain any of the following operands:

- scalar nets
- expanded vector nets that contain less than 64 bits
- bit-selects of expanded vector nets
- part-selects that are less than 64 bits of expanded vector nets
- scalar registers
- constants

You can also accelerate a continuous assignment where the right-hand side is a concatenation of these types of nets, provided that the concatenation contains fewer than 64 bits.

Example 5-8 shows continuous assignments that you can accelerate because the operands in the expression on the right-hand side meet these restrictions.



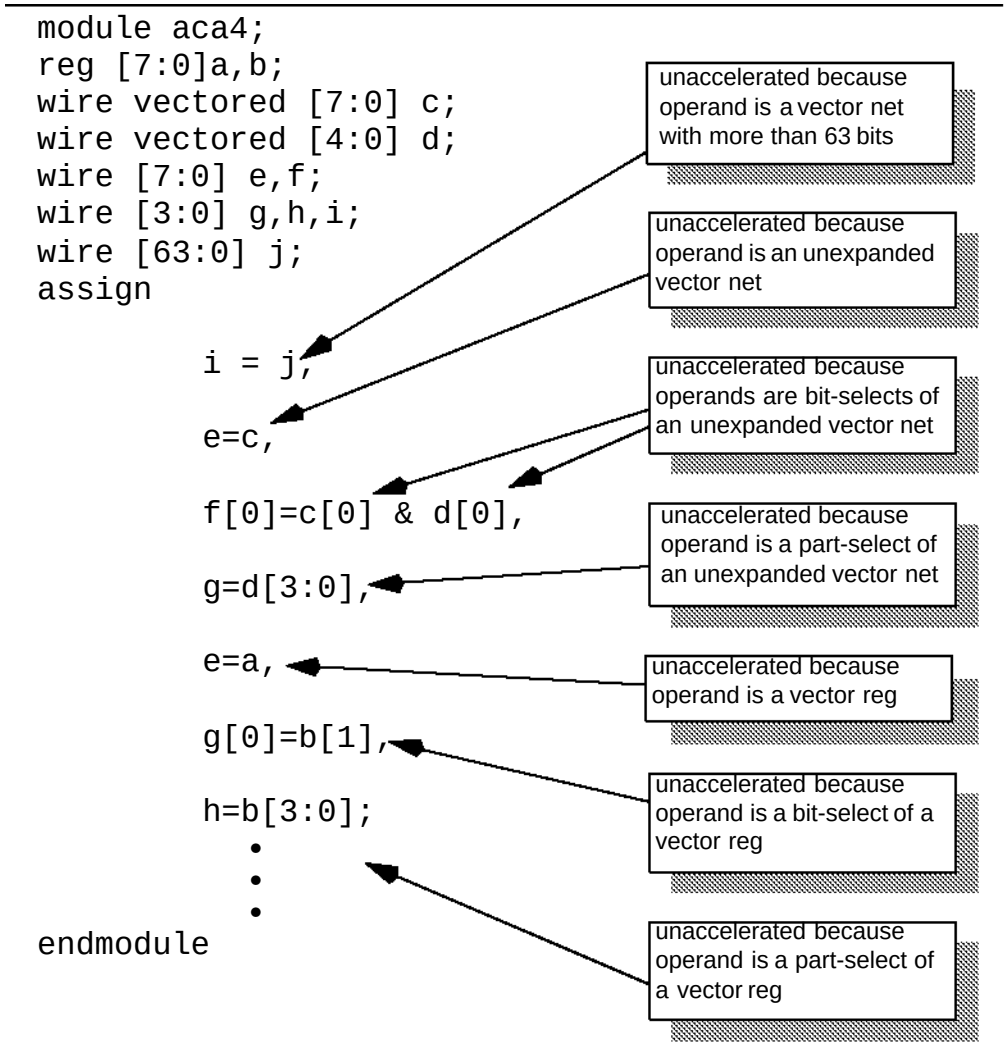
Example 5-8: Operands in continuous assignments that can be accelerated

In Example 5-8, all operands are less than 64 bits.

The prohibited operands are as follows:

- expanded vector nets that contain more than 63 bits
- unexpanded vector nets
- bit-selects of unexpanded vector nets
- part-selects of unexpanded vector nets
- vector registers
- bit-selects of vector registers
- part-selects of vector registers

Example 5-9 shows continuous assignments that you cannot accelerate because the operands in the expression of the right-hand side do not meet these restrictions.



Example 5-9: Operands in continuous assignments that cannot be accelerated

The expression on the right-hand side of a continuous assignment can only contain the following operators:

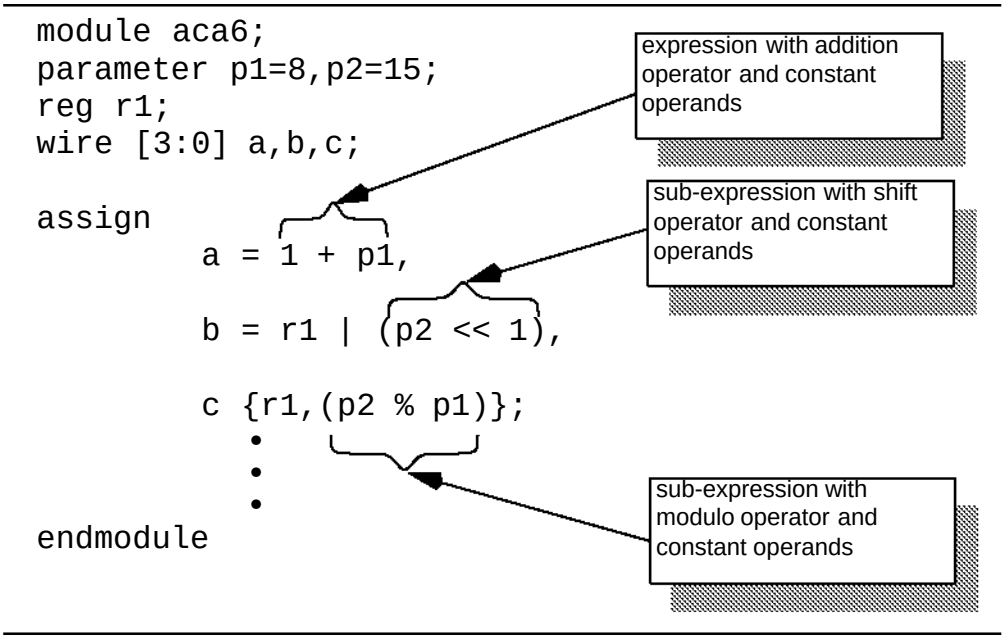
&	bit-wise and reduction AND
&&	logical AND
~&	reduction NAND
|	bit-wise and reduction OR
||	logical OR
~|	reduction NOR
^	bit-wise and reduction XOR
~^	bit-wise and reduction XNOR
~	bit-wise NOT
!	logical NOT
{}	concatenation
{{}}	duplicate concatenation
?:	conditional
==	logical equality
!=	logical inequality

Example 5-10 shows continuous assignments that you can accelerate because the operators in the expression on the right-hand side meet these restrictions.

```
module aca5;
reg r1,r2,r3,r4,r5,r6,r7;
wire a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,s,t;
wire [1:0]u,v,w,y;
assign
        a=r1 & r2,          bit-wise AND operator
        b=&s,               unary reduction AND operator
        c=r1 && r2,         logical AND operator
        d=~&u,              unary reduction NAND operator
        e=r2 | r3,          bit-wise OR operator
        f=r3 || r4,         logical OR operator
        g=|u,               unary reduction OR operator
        h=~|u,              unary reduction NOR operator
        i=r4 ^ r5,          bit-wise XOR operator
        j=^u,               unary reduction XNOR operator
        k=r5 ~^r6,          bit-wise XNOR operator
        l=~^v,              unary reduction XNOR operator
        m=~j,               bit-wise NOT operator
        n = !r1,            logical NOT operator
        w={a,b},            concatenation operator
        y={2{r7}},          duplicate concatenation operator
        q=r1 ? a : b,       conditional operator
        s= r1 == r2,        logical equality operator
        t= r3 != r4;        logical inequality operator
endmodule
```

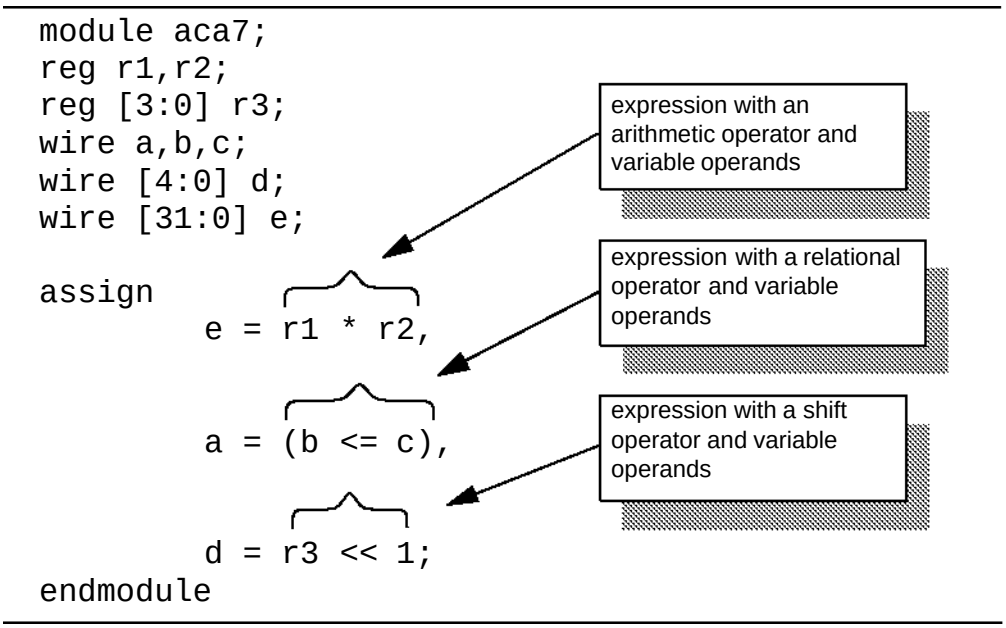
Example 5-10: Operators in continuous assignments that can be accelerated

You can enter other operators in the right-hand side of accelerated continuous assignments, but only in an expression or sub-expression whose operands are constants. (A sub-expression is a part of an expression that Verilog-XL can evaluate separately.) The prohibition against other operators does not apply in these expressions or sub-expressions because Verilog-XL evaluates them at compile time. Example 5-11 shows how you can use other operators in accelerated continuous assignments.



Example 5-11: Other operators in accelerated continuous assignments

Example 5-12 shows continuous assignments that you cannot accelerate because they use other operators with variable operands.

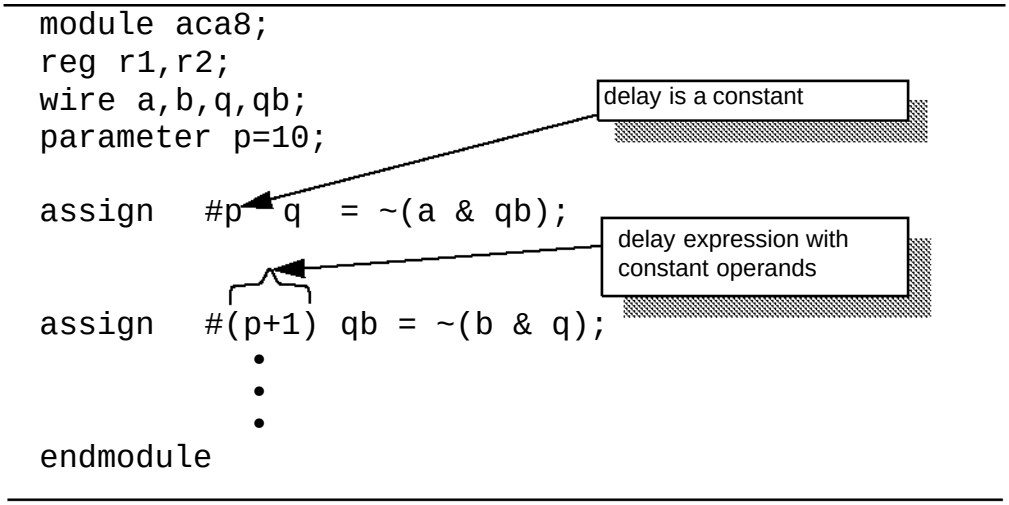


Example 5-12: Operators in continuous assignments that cannot be accelerated

Delay expression restrictions

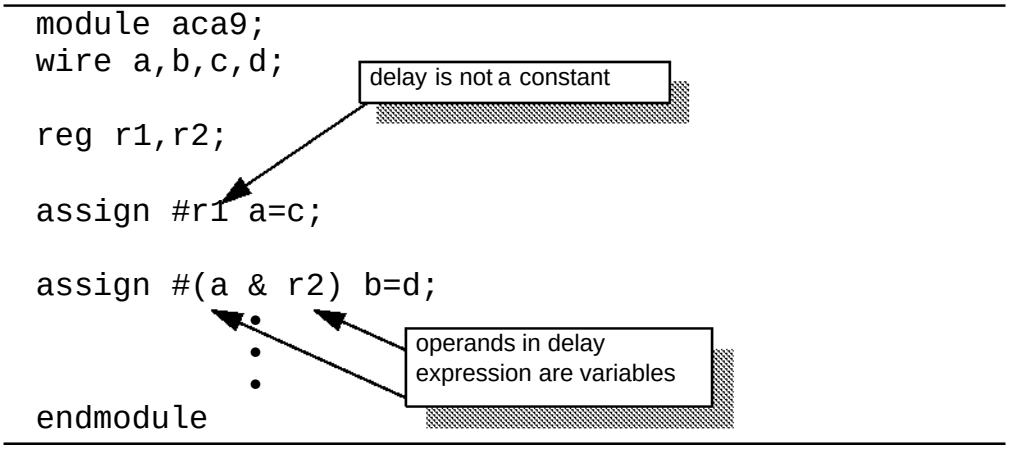
You can accelerate a continuous assignment that includes a delay only if that delay is a constant or an expression whose operands are constants.

Example 5-13 shows continuous assignments that you can accelerate because the delay expression meets this restriction.



Example 5-13: Delay expressions in continuous assignments that can be accelerated

Example 5-14 shows continuous assignments that you cannot accelerate because the delay expression does not meet this restriction.



Example 5-14: Delay expressions in continuous assignments that cannot be accelerated

Restriction summary

Figure 5-1 summarizes the valid syntax elements in accelerated continuous assignments.

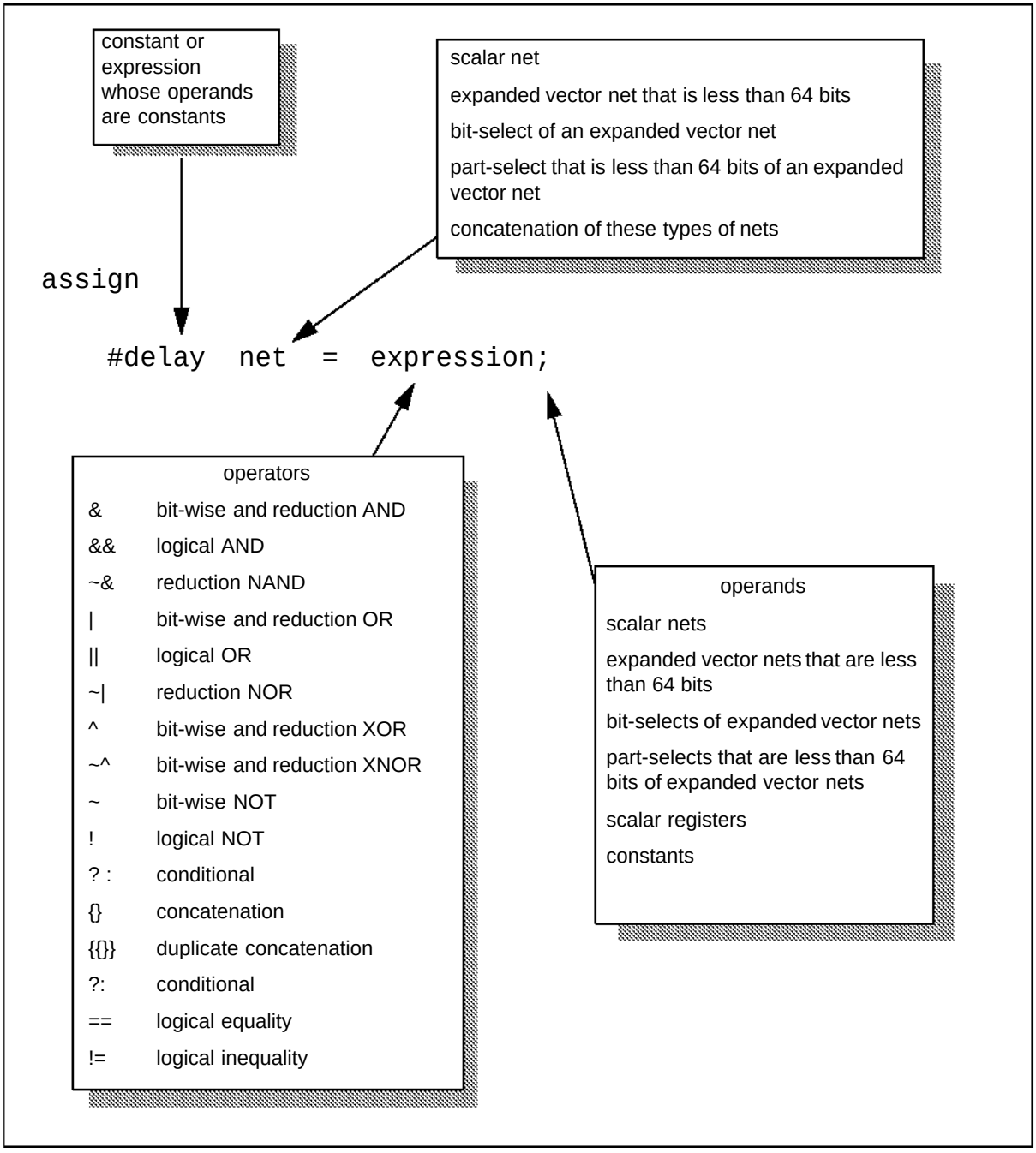


Figure 5-1: Syntax elements of an accelerated continuous assignment

5.3.2 How to Control the Acceleration of Continuous Assignments

Accelerate continuous assignments in your design by entering the `+caxl` command line option. When you enter this option, you accelerate the continuous assignments in the regions of your design that can contain accelerated primitives. You specify these regions with the following mechanisms:

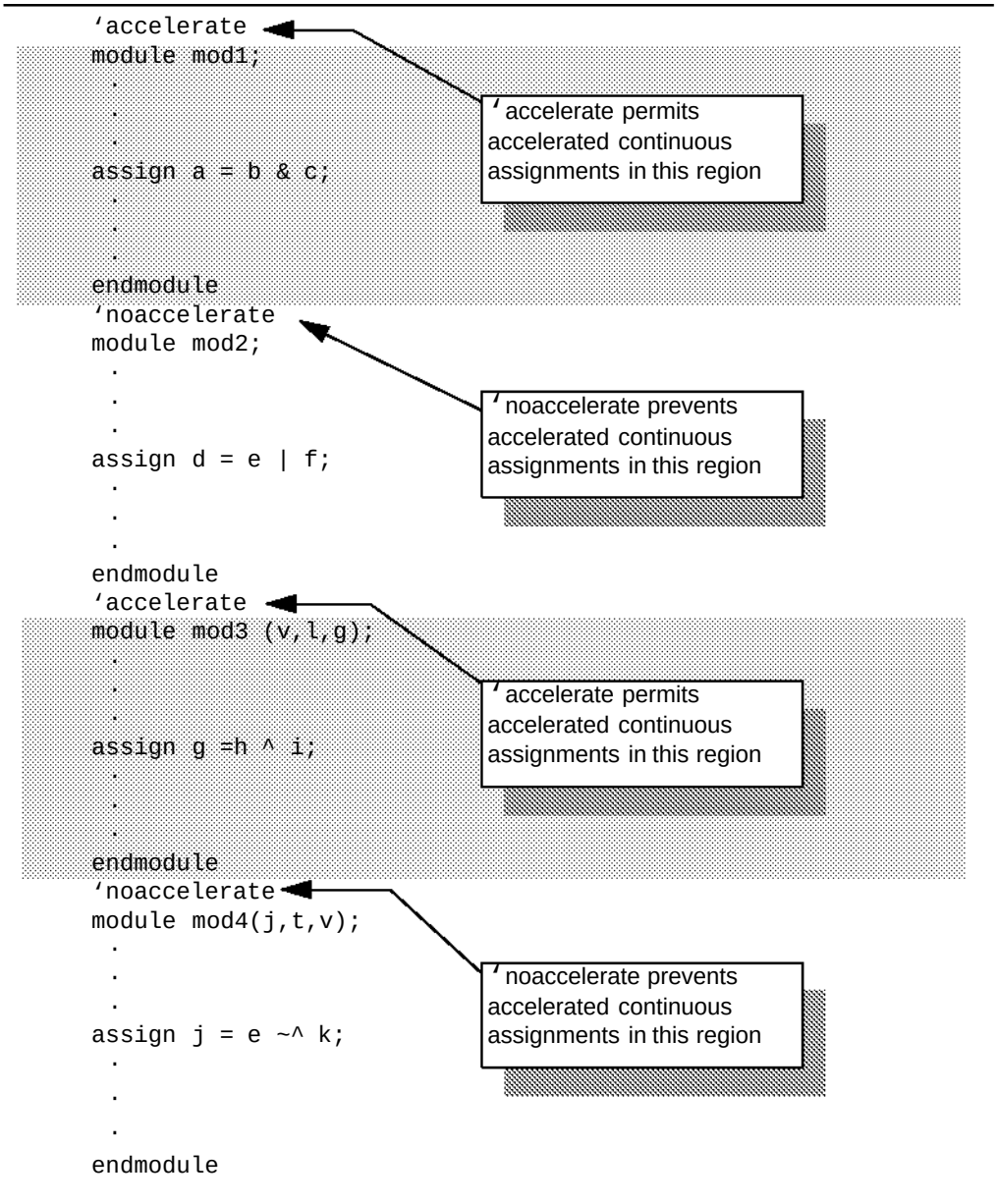
- `-a` command line option
- `‘accelerate` compiler directive
- `‘noaccelerate` compiler directive

The following command line shows how the `-a` option works with the `+caxl` option:

```
verilog source.v -a +caxl
```

This command line tells Verilog-XL to accelerate all the primitives and continuous assignments in `source.v` that it can.

Example 5-15 shows the regions of a sample design, delimited by `‘accelerate` and `‘noaccelerate`, whose continuous assignments you can accelerate if you enter the `+caxl` option, without the `-a` option, on the command line. In Example 5-15, the continuous assignments in the grey regions can be accelerated, and the other continuous assignments cannot be accelerated.



Example 5-15: Design regions that you can accelerate

5.3.3
The Effects of Accelerated Continuous Assignments

Accelerating continuous assignments can have the following effects on your simulation:

- faster simulation
- slightly slower compilation
- slightly more memory use
- simulation results that are different from the results when you do not accelerate continuous assignments

These effects are described in the following subsections.

Simulation speed

Accelerating continuous assignments does not increase the simulation speed of all designs. The types of designs that simulate faster, and the one type that simulates slower, are described in this subsection.

Designs that simulate faster

The following is a list of the kinds of designs that simulate faster when you accelerate continuous assignments:

- designs that consist entirely of accelerated continuous assignments to scalar nets
- designs that are a combination of gate-level and accelerated continuous assignments
- gate-level designs that are stimulated by accelerated continuous assignments
- designs that consist of accelerated continuous assignments to large vector nets

The following are examples of these designs and an explanation of how accelerated continuous assignment increases their simulation speed.

1. Accelerating continuous assignments is what most increases the simulation speed of designs that consist entirely of accelerated continuous assignments to scalar nets. These designs simulate approximately eight times faster when you accelerate all their continuous assignments. The following source description shows a design of a multiplexer that consists of accelerated continuous assignments to scalar nets:

```
module aca10 (op1,op2,s1,s2,out,cr);
input op1,op2,s1,s2;
output out,cr;
wire nop1,nop2,mx1,mx2;
assign
        nop1 = ~op1,
        nop2 = ~op2,
        mx1  = ((op1 & s1)|(nop1 & ~s1)),
        mx2  = ((op2 & s2)|(nop2 & ~s2)),
        out  = mx1 ^ mx2,
        cr   = mx1 & mx2;
endmodule
```

Example 5-16: Design that consists entirely of accelerated continuous assignments

In this source description, data flows through a path of accelerated continuous assignments.

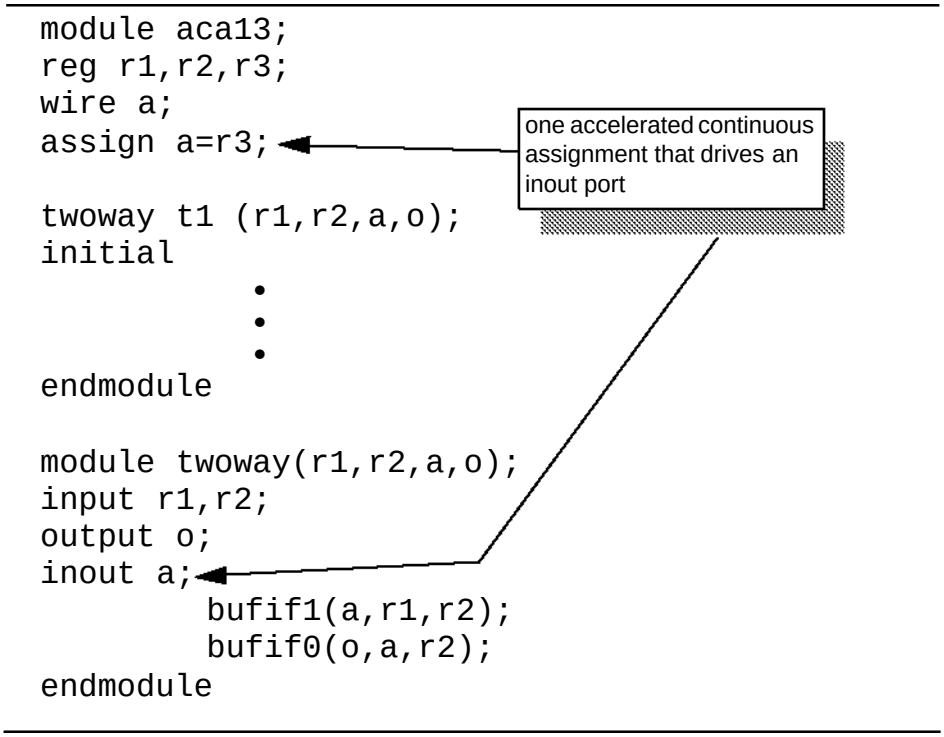
2. Accelerating continuous assignments also increases the simulation speed of designs whose logic is a combination of gate-level and accelerated continuous assignments. How much the acceleration of the continuous assignments increases the simulation speed depends on the proportion of continuous assignments to gate instances. The following source description shows a design that is a combination of accelerated continuous assignments and gate instances:

```
module aca11 (op1,op2,s1,s2,out,cr);
input op1,op2,s1,s2;
output out,cr;
wire nop1,nop2,mx1,mx2;
assign
        mx1  = ((op1 & s1)|(nop1 & ~s1)),
        mx2  = ((op2 & s2)|(nop2 & ~s2));

        not nt1 (nop1,op1),
            nt2 (nop2,op2);
        xor xr1 (out,mx1,mx2);
        and ad1 (cr,mx1,mx2);
endmodule
```

Example 5-17: Design that consists of accelerated continuous assignments and gate instances

In this source description, data flows from gates to continuous assignments and back to gates.

3. Accelerating continuous assignments also increases the simulation speed of gate-level designs that are stimulated by accelerated continuous assignments. How much the acceleration of the continuous assignments increases the simulation speed of these designs also depends on the proportion of continuous assignments to gate instances, as in the following source description:



Example 5-18: Design that contains only one accelerated continuous assignment

This design includes one accelerated continuous assignment. Accelerating this continuous assignment does little to increase the design's simulation speed because the accelerated continuous assignment is such a small proportion of this design.

4. Accelerating the continuous assignments in designs that consist of continuous assignments to large vector nets results in the smallest increase in simulation speed. Continuous assignments to vector nets 64 bits wide and larger cannot be accelerated. The closer the left-hand side of a continuous assignment comes to this limit of 63 bits, the more time the XL algorithm needs to simulate the continuous assignment, as in the following source description:

```
module aca14;
wire [30:0]
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t;
wire [61:0] m1,m2,m3,m4,m5;
assign   m1=~(({a,b}&{d,e})|({c,d}^{e,f})),
         m2={e,f}&{h,i},
         m3=~{i,j},
         m4=~({m,n}|{a,b}),
         m5=((q & r)^(p | t)~^{q,r});
endmodule
```

Example 5-19: Design that consists of continuous assignments to large vector nets

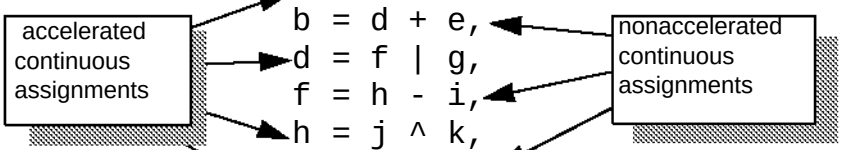
This source description shows the continuous assignment of expressions with large operands and several operators to very large vector nets. The greater the complexity of the expression on the right-hand side and the larger the vector net on the left-hand side, the more time the XL algorithm needs to simulate the continuous assignment.

Designs that simulate slower

Not all designs with continuous assignments that can be accelerated simulate faster with the XL algorithm. XL speeds up the simulation when it processes a continuous assignment, but transitions between the XL and non-XL algorithms slow down the simulation. A large number of transitions can make a simulation run slower than if no part of it is simulated by the XL algorithm. The following is a list of designs that contain continuous assignments that you can accelerate, but which simulate faster without accelerating these continuous assignments.

1. Designs whose data flows many times from accelerated to nonaccelerated continuous assignments simulate at a slower speed than if you did not accelerate any continuous assignment. This slower speed is caused by the performance cost of a large number of transitions between algorithms. The following source description shows data flowing through a path of continuous assignments that cause Verilog-XL to transition frequently between algorithms.

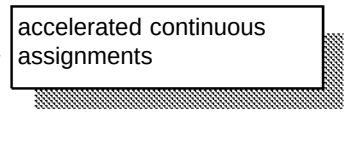
```
module aca15;
wire a,b,c,d,e,f,g,h,i,j,k,l,m,n,o;
assign
                a = b & c,
                b = d + e,
                d = f | g,
                f = h - i,
                h = j ^ k,
                j = l * m,
                l = n ~^ o;
                •
                •
                •
endmodule
```

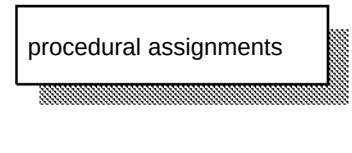


Example 5-20: Design whose data flows from accelerated to nonaccelerated continuous assignments

2. Designs whose data flows many times from accelerated continuous assignments to procedural assignments also simulate at a slower speed that if you did not accelerate any continuous assignment. This slower speed is also caused by transitions between algorithms. In the following source description, data flows between both kinds of assignments.

```
module aca16;
reg r1,r2,r3,r4,r5;
wire a,b,c,d,e;
assign
        a = r1,
        b = r2,
        c = r3,
        d = r4,
        e = r5;

always
        begin
        #10 r1 = b;
        #10 r2 = c;
        #10 r3 = d;
        #10 r4 = e;
        #10 r5 = ~r5;
        end
initial
        begin
        r5=1;
          •
          •
          •

endmodule
```





Example 5-21: Design whose simulation causes transitions between algorithms

In this source description, a value of `1` propagates through several wires and registers. Data flow begins with a procedural assignment to reg `r5`, then through a path of registers and wires that are driven by alternating continuous and procedural assignments.

Compilation speed

During compilation, Verilog-XL processes accelerated continuous assignments so that they can be simulated by the XL algorithm. Therefore, compilation time increases as the number of accelerated continuous assignments increases. A design that consists entirely of continuous assignments that can be accelerated takes approximately twice as long to compile if you accelerate these continuous assignments. (In a typical worst-case design, compilation without accelerated continuous assignments took 19 seconds; compilation with accelerated continuous assignments took 41 seconds.)

Memory usage

Accelerated continuous assignments cause Verilog-XL to use more memory at compile time, but less memory at run time.

Verilog-XL needs more memory to compile a design with accelerated continuous assignments. A design that consists entirely of accelerated continuous assignments needs 20% more memory to compile.

Accelerated continuous assignments reduce Verilog-XL's memory requirements during simulation.

The possibility of different results

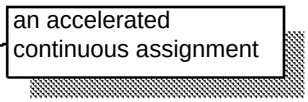
Accelerating continuous assignments to vector nets when these continuous assignments include delay expressions can produce simulation results that differ from the results produced without accelerating these continuous assignments. This possible difference is caused by the difference between how the XL and non-XL algorithm simulate these continuous assignments.

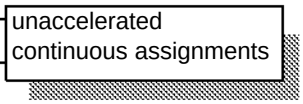
In both the XL and non-XL algorithms, when a continuous assignment statement includes a delay expression, Verilog-XL evaluates the right-hand side and schedules the assignment to occur after the delay elapses. In the non-XL algorithm, if any of the bits of the right-hand side change before the delay elapses, Verilog-XL re-evaluates the entire right-hand side and reschedules the assignment. In the XL algorithm, if any of the bits of the right-hand side change before the delay elapses, Verilog-XL schedules a subsequent assignment to those bits.

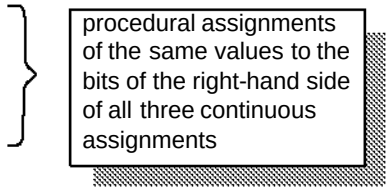
Example 5-22 and Example 5-23 show how accelerating continuous assignments can produce different simulation results.

Example 5-22 shows a module that contains accelerated and unaccelerated continuous assignments that assign the same values and include the same delay expression. The accelerated continuous assignments propagate value changes at simulation times when the unaccelerated continuous assignments do not propagate these value changes.

```
module dif;
wire [1:0] a1, a2;
wire vectored [1:0] b1;
reg c1,c2;
reg [1:0] d1;

assign   #10 a1 = {c1,c2};

assign
         #10 b1 = {c1,c2},
             a2 = d1;

initial
         begin
         $monitor("At simulation time %0d\n",
         $time,
         "  accelerated a1=%b\n",a1,
         "unaccelerated b1=%b a2=%b\n\n",b1,a2);
         #25 c1 = 0;
             d1[1] = 0;
         #5  c2 = 0;
             d1[0] = 0;
         end
endmodule
```







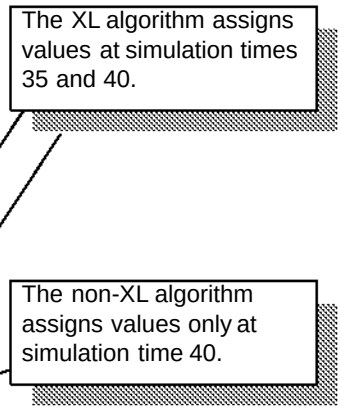
Example 5-22: Module with accelerated and unaccelerated continuous assignments

In Example 5-22, the continuous assignment to wire `a1` of the concatenation of scalar registers `c1` and `c2` can be accelerated. The continuous assignment to wire `b1` cannot be accelerated because it assigns a value to an unexpanded vector net; the continuous assignment to wire `a2` cannot be accelerated because its operand is a vector reg. The delay expression in these continuous assignments is 10 time units.

Procedural assignments assign the same values to the right-hand sides of these continuous assignments. These procedural assignments specify a five time unit interval between bit changes of the right-hand sides of the continuous assignments.

The XL algorithm schedules the propagation of all bit changes as they occur; the non-XL algorithm does not. The difference in simulation results between the accelerated and unaccelerated continuous assignments is shown in Example 5-23.

```
Highest level modules:
dif

At simulation time 0
  accelerated a1=xx
unaccelerated b1=xx a2=xx


At simulation time 35
  accelerated a1=0x
unaccelerated b1=xx a2=xx


At simulation time 40
  accelerated a1=00
unaccelerated b1=00 a2=00
```



Example 5-23: Different simulation results

In Example 5-23, the XL algorithm assigns values to `a1` at simulation times 35 and 40. The non-XL algorithm waits until simulation time 40 to assign values.