6.2.1.1 Understanding Types and Numbers

Verilog defines two data types: **nets**(or *wire*) and ***reg***. A *net* represents a physical connection between structural entities and is of a resolved data type, meaning that the final value will be the resolution of all the sources (i.e., drivers) asserting a value onto the net. A *net* object is treated as an UNSIGNED number object and loses the significance of the SIGN bit.

A *reg* is an "abstraction of a data storage element" and may store a value. For synthesis, a *reg* type includes the *reg* and i*nteger* declarations. An object asserted a value in an *always* or *initial* block must be of type *reg (*synthesis ignores the *initial statement).* In addition, a local object of a *task* or *function* must also be of type *reg.* A *reg* object is unresolved and can be assigned a value from different *always* or *initial* blocks. However, for synthesis, a *reg* type can only be assigned in only one *always* block (excluding *task* and *function*). A *reg* object can be a discrete one bit, or a vector (e.g., an array of bits), or a memory (i.e., one-dimensional array of vectors).

An object of type *reg* is processed by Verilog as an UNSIGNED number, however, it can be assigned a negative constant. Negative numbers are represented in 2's complement form. A *reg* object loses the significance of the SIGN bit. An object of type *integer* is processed by Verilog as a SIGNED number and retains the significance of the sign. The difference between SIGNED and UNSIGNED number is in SIGN or ZERO extension of the left most bits when arithmetic and logical operations are performed. Section 4.4.1 of the Verilog LRM specifies the rules for expression bit lengths. For "+ - / % & | ^ ^~ ~^" arithmetic operations, and for "=== !== == != && || > >= < <=" logical operations, the number of bits used in the expression evaluations is the maximum of the length of the left operand and the right operand. Remember that an *integer* value is a 32-bit value, whereas a sized value is defined by the value of the size. Decimal numbers are signed. Based-numbers (e.g., 4'h21) are unsigned. An UNSIZED value (e.g., *'h5*) is 32 bits. Unsized unsigned constants, where the high order bit is unknown (e.g., X or x) or tri-state (Z or z), are extended to the size of the expression containing the constant. If the size of the unsigned number is smaller than the size specified for the constant (e.g. intA32bits = 'hF;), the unsigned number is padded to the left with zeros (e.g., intA32bits = 32'h0000_000F;). If the leftmost bit in the unsigned number is an *x* or a *z*, then an *x* or a *z* is used to pad to the left respectively. These concepts are demonstrated in Figure 6.2.1.1-1 and 6.2.1.1-2. Table 6.2.1.1 provides an explanation of the results for the simulation of Figure 6.2.1.1-1.

```
module arith2;
 integer intA;
 reg [15:0] regA;

 always @ (intA or regA)
    $display($time, "   intA = %h, regA= %h", intA, regA);
```

```
  initial
     begin
      #50 intA = -4'd12;
      #50 regA = intA / 3; // expression result is -4,
      // intA is an integer data type, regA is 65532

      #50 regA = -4'd12; // regA is 65524

      #50 intA = regA / 3; // expression result is 21841,
      // regA is a reg data type

      #50 intA = -4'd12 / 3;// expression result is 1431655761.
      // -4'd12 is effectively a 32-bit reg data type

      #50 regA = -12 / 3; // expression result is -4, -12 is effectively
      // an integer data type. regA is 65532

      #50 regA = 'h1z;
      #50 intA = 'h1z;

      #50 regA = 'hz;
      #50 intA = 'hz;

      #50 regA ='hf;
      #50 intA = 'hf;

    #500 regA = 'h0;
     end
endmodule //

/* -----\/----- EXCLUDED -----\/-----
NC-Sim> run 1 us
              50   intA = fffffff4, regA= xxxx
             100   intA = fffffff4, regA= fffc
             150   intA = fffffff4, regA= fff4
             200   intA = 00005551, regA= fff4
             250   intA = 55555551, regA= fff4
             300   intA = 55555551, regA= fffc
             350   intA = 55555551, regA= 001z
             400   intA = 0000001z, regA= 001z
             450   intA = 0000001z, regA= zzzz
             500   intA = zzzzzzzz, regA= zzzz
             550   intA = zzzzzzzz, regA= 000f
             600   intA = 0000000f, regA= 000f
-----/\----- EXCLUDED -----/\----- */
```

Figure 6.2.1.1-1 Numbers on Registers and Integer Objects (*ch6/arith2.v*)

Table 6.2.1.1 Register and Integer Objects

OPERATION	VALUE	COMMENTS
integer intA; reg [15:0] regA;	IntA is 32 bits regA is 16 bits	Object decalration

intA = -4'd12;	intA = FFFF_FFF4	4'd12 = 0000_000C // unsigned number in hex, extended
		// to 32 bits because it will be assigned onto a
		// a 32-bit vector
		-4'd12 = 2's complement of the 32-bit vector
		= 2's complement(0000_000C) = FFFF_FFF4

regA = intA / 3;	regA= FFFC	intA is a SIGNED integer number
		intA / 3 = FFFF_FFFC
		// regA is a 16-bit UNSIGNED register
		regA = intA[15:0] = FFFC

regA = -4'd12;	regA= FFF4	4'd12 = 000C // unsigned number in hex, extended
		// to 16 bits because it will be assigned onto a
		// a 16-bit vector
		-4'd12 = 2's complement of the 16-bit vector
		= 2's complement(000C) = FFF4

intA = regA / 3;	intA = 0000_5551	// regA is UNSIGNED FFF4 or 65524, positive number
		regA /3 = FFF4 / 3 = 5551
		// regA is zero extended to 32 bits because it will be
		// assigned onto a 32-bit register. Thus,
		intA = 0000_5551

intA = -4'd12 / 3;	intA = 5555_5551	4'd12 = 0000_000C // unsigned number in hex, extended // to 32 bits because it will be assigned onto a // a 32-bit vector -4'd12 = 2's complement of the 32-bit vector = 2's complement(0000_000C) = FFFF_FFF4 -4'd12/3 = FFFF_FFF4/3 = 5555_5551
regA = -12 / 3	regA= FFFC	// -12 is an integer 32-bit number -12 = FFFF_FFF4 (in hex) -12/3 = FFFF_FFFC
regA = 'h1z;	regA= 001z	//regA is 16 bits. 'h1z = 0001_zzzz, zero extended
intA = 'h1z;	intA = 0000001z	//intA is 32 bits. 'h1z = 0001_zzzz, zero extended
regA = 'hz;	regA= zzzz	Automatic left padding `{16{1'hz}}`
intA = 'hz;	intA = zzzzzzzz	Automatic left padding `{32{1'hz}}`
regA ='hf;	regA= 000f	Unsized padding, zero padding
intA = 'hf;	intA = 0000000f	Unsized padding, zero padding

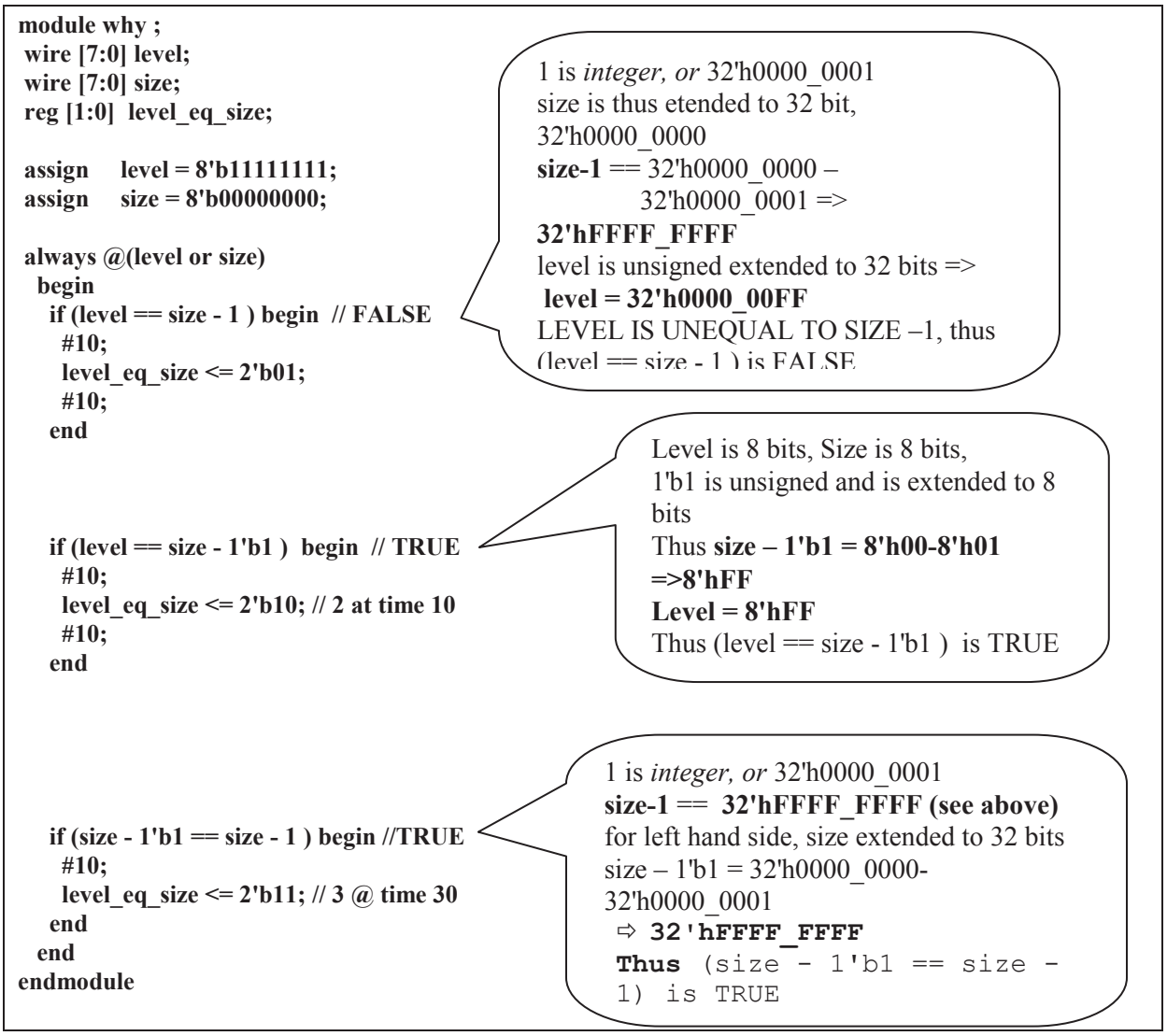

Figure 6.2.1.1-2 Comparison Operations on Registers and Integer Objects (*ch6/why.v*)

6.2.1.2 Signed Operations with Unsigned Registers

As mentioned in previous subsections, Verilog'95 does not support SIGNED registers, but current synthesis tools do support the Verilog'95 HDL. When SIGNED numbers need to be used, the users have two options:

1. Use *Integer* type: This approach is only available for numbers whose vector equivalent can be represented in 32-bit numbers extending in range from -2^{31} to + $(2^{31} - 1)$.
2. Use *reg* type with caution: reg vectors can be of any size, and are not limited to 32 bits. Since the interpretation of *reg* types is UNSIGNED, users must perform all sign extensions and must modify the comparison operators to properly interpret the signs of the numbers. For example, *a =4'b0101*; // +5, *b=4'b1011*; // -5, *if (a>b)* will be FALSE since *a* and *b* are UNSIGNED *reg*, an error if the user's interpretation of *a* and *b* are SIGNED numbers.

Figure 6.2.1.2-1 represents a model of a multiplier that checks signs of operands before performing the multiplication.

```
module signmult95cc (k_64c, k_32a, k_32b);
  output [63:0] k_64c;
  input  [31:0] k_32a, k_32b;
  reg    [63:0] k_64c;
  reg    [31:0] op32a, op32b;
  reg           neg_product;

  // (1) if only one of the operand MSBs is set,
  //     the product will be negative.
  // (2) generate the absolute value of each multiplicand.
  // (3) generate the appropriate positive or
  //     negative signed output product
  always @(k_32a or k_32b) begin
    neg_product = (k_32b[31] ^ k_32a[31]);
    if (k_32a[31]) op32a = -k_32a; // absolute value
    else           op32a =  k_32a;
    if (k_32b[31]) op32b = -k_32b; // absolute value
    else           op32b =  k_32b;
    if (neg_product) k_64c = -(op32a * op32b);
    else             k_64c =  (op32a * op32b);
  end
endmodule
// Cliff Cummings      Sunburst Design, Inc.
// cliffc@sunburst-design.com
```

Figure 6.2.1.2-1 Model of a Multiplier (*ch6/signmult95cc.v*)

Figure 6.2.1.2-2 represents a trivial testbench to quickly evaluate the multiplier. Figure 6.2.1.2-3 shows the test results.

```
module signmult95_tb;
  parameter SIZE = 32;
  wire [2*SIZE-1:0]         k_64c;
  reg [SIZE-1:0] k_32a;
  reg [SIZE-1:0] k_32b;


  signmult95cc signmult95cc(
                                    // Outputs
                                    .k_64c              (k_64c[2*SIZE-1:0]),
                                    // Inputs
                                    .k_32a              (k_32a[SIZE-1:0]),
                                    .k_32b              (k_32b[SIZE-1:0]));

   initial begin
       #40;
       k_32a <= 5;
       k_32b <= 6;
       #50;
       k_32a <= 32'hFFFF_FFFB;
       k_32b <= 32'h0000_0004;
       #50;
       k_32b <= 32'hFFFF_FFFC;
       #50;
   end

  always @ (k_32a or k_32b or k_64c)
      $display ($time, "  k_32a=%h; k_32b=%h; k_64c=%h", k_32a, k_32b, k_64c);
endmodule // signmult95_tb
```

Figure 6.2.1.2-2 Simple Multiplier Testbench (*ch6/signmult95_tb.v)*

```
NC-Sim> run 1 us
              40  k_32a=00000005; k_32b=00000006; k_64c=xxxxxxxxxxxxxxxx
              40  k_32a=00000005; k_32b=00000006; k_64c=000000000000001e
              90  k_32a=fffffffb; k_32b=00000004; k_64c=000000000000001e
              90  k_32a=fffffffb; k_32b=00000004; k_64c=ffffffffffffffec
             140  k_32a=fffffffb; k_32b=fffffffc; k_64c=ffffffffffffffec
             140  k_32a=fffffffb; k_32b=fffffffc; k_64c=0000000000000014
Ran until 1 US + 0
NC-Sim>
```

Figure 6.2.1.2-3 Multiplier Test Results (with Cadence *NC-Sim*)

Figure 6.2.1.2-4 is an RTL view of the synthesized multiplier. The two's complement inverters, multiplexers, and multipliers are clearly demonstrated with this vectored SIGNED multiplier defined in Verilog'95.

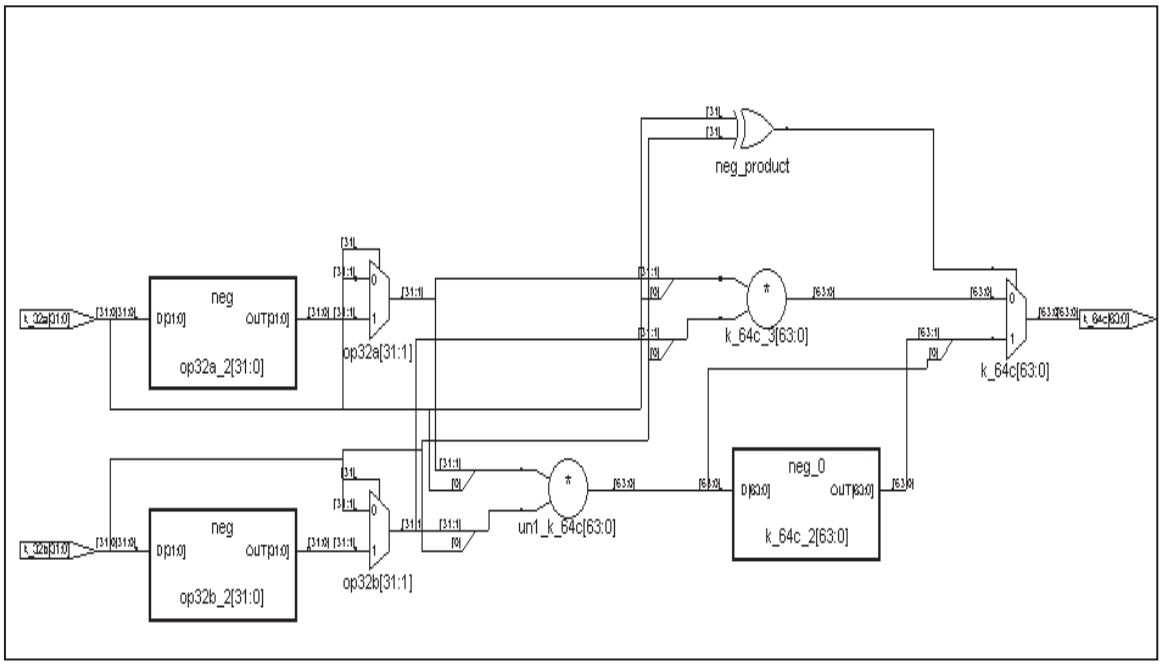

Figure 6.2.1.2-4 Synthesized Multiplier RTL View (with Synplicity *Synplify Pro*)

6.2.2 Verilog 1364-2001 Arithmetic

Many new features were added in Verilog 2001 version to support arithmetic operations. These are summarized below. New Verilog 2001 are summarized in the reference shown in the footnote, and of course, are defined in the Language Reference Manual.

- Verilog 2001[51] added the concept of SIGNED registers and nets:
  reg signed [63: 0] data;
  wire signed [11: 0] address;

- Function returns can be declared as signed:
  function signed [128: 0] function signed [128: 0] alu;

- Literal integer numbers can be declared as signed:
  16'shC501 // a signed 16- bit hex

- New arithmetic shift operators (<<< and >>>) maintain the sign of a value.

- New $signed() and $unsigned() system functions can "cast" a value to signed or unsigned:
  data_signed = $signed(data_unsigned);

---

[51] *The IEEE Verilog 1364- 2000 Standard, What's New, and Why You Need It*, Stuart Sutherland, Sutherland HDL, Inc., Presented at the HDLCON- 2000 Conference March 10, 2000 San Jose, California (http://www.sutherland-hdl.com)

- Automatic Width Extension Past 32 bits:
  1. In Verilog- 1995:
  - Verilog assignments zero fills when the left-hand side is wider than the right- hand side
  - Unsized integers default to 32- bits wide; therefore, the widths of integers must be hard-coded

  Verilog- 1995
  parameter WIDTH = 64;
  reg [WIDTH- 1: 0] data;
  data = 'bz; // fills with 'h00000000zzzzzzzz
  data = 64'bz; // fills with 'hzzzzzzzzzzzzzzzz

  2. Verilog- 2001 automatically extends a logic Z or X to the full width of the left-hand side

  Verilog- 2001
  parameter WIDTH = 64;
  reg [WIDTH- 1: 0] data;
  data = 'bz; // fills with 'hzzzzzzzzzzzzzzzz

More on CADENCE Signed Objects[52]

Cadence *NC-Sim* currently supports SIGNED objects and SIGNED arithmetic. The values of signed quantities are represented with two's complement notation. A signed value will not cross hierarchical boundaries. If a signed value is needed in other modules of a hierarchy, then it must be declared in each of the modules where signed arithmetic is necessary. The following example shows some sample declarations.

wire signed [3:0] signed_wire; // range -8 <-> +7
reg signed [3:0] signed_reg; // range -8 <-> +7
reg signed [3:0] signed_mem [99:0] // 100 words range -8 <-> +7
function signed [3:0] signed_func; // range -8 <-> +7

A based constant can be typed by prepending the letter `s` to the base type as shown in Figure 6.2.2.1-1.

[52] NC-Sim Reference, Product Version 3.2 Cadence Design Systems, Inc.
http://www.cadence.com This arithmetic is complient to Verilog 2001

```
module test_signed1;
  reg signed [3:0] sig_reg;
  reg [3:0] unsig_reg;

  initial
    begin
      #10  sig_reg   = -4'd1;
      unsig_reg = -4'd1;
      #1  $display ($time,,"sig_reg=%d unsig_reg=%d (-4'd1)=%d (-4'sd1)=%d",
            sig_reg, unsig_reg, -4'd1, -4'sd1);
      #10 sig_reg   = -4'sd1;
      unsig_reg = -4'sd1;
      #1  $display ($time,,"sig_reg=%d unsig_reg=%d (-4'd1)=%d (-4'sd1)=%d",
            sig_reg, unsig_reg, -4'd1, -4'sd1);
    end
endmodule
```

Figure 6.2.2.1-1 Aplication of SIGNED Type (Ch6/test_signed1.v)

The simulation output with Cadence *NC-Sim* is:

11 sig_reg= -1 unsig_reg=15 (-4'd1)=15 (-4'sd1)= -1
22 sig_reg= -1 unsig_reg=15 (-4'd1)=15 (-4'sd1)= -1

The following rules determine the resulting type of an expression:

- The expression type depends only on the operands. It does not depend on the left-hand side (LHS) (if any).
- Decimal numbers are signed.
- If any operand is real, the result is real.
- If all operands are signed, the result is signed, regardless of operator.
- The following list shows objects that are unsigned regardless of the operands:
  - The result of any expression where any operand is unsigned
  - Based numbers
  - Comparison results (1, 0)
  - Bit select results
  - Part select results
  - Concatenate results
- If a signed operand is to be resized to a larger signed width and the value of the sign bit is `x` or `z`, the resulting value will be a bit filled with an `x` value.
- If any non-logical operation has a bit with a signed value of `x` or `z`, then the result is `x` for the entire value of the expression.

Nets as signed objects only have significance in an expression, in which case the entire expression is considered a signed value.

Expressions on ports are typed, sized, evaluated, and assigned to the object on the other side of the port using the same rules as expressions in assignments. Verilog-XL

uses the following steps for evaluating an expression:

1. Determine the right-hand side (RHS) type, then coerce all RHS operands to this type.
2. Determine the largest operand size, including the LHS (if any), then resize all RHS operands to this size.
3. Evaluate the RHS expression, producing a result of the type found in step 1 and the size found in step 2.
4. If there is a LHS,
   - Resize the result to the LHS size.
   - Coerce the result to the LHS type.

Figure 6.2.2.1-2 provides an example of a counter with SIGNED ports and registers.

```
module signed_counter (
                                    // Outputs
                                    count_out, termcount,
                                    // Inputs
                                    clk, rst_n
                                    ) ;
 parameter SIZE = 4;
 output          signed [SIZE - 1:0] count_out;// out : counter output
 output          termcount;
 input           clk;            // in : system clock
 input           rst_n;   // in : reset, active hi
 reg                 signed [SIZE-1:0] count_out;
 reg                 termcount;
 wire                signed [15:0] regS;
 assign          regS = -5;

 always @ ( posedge clk)
    begin
   if (! rst_n) begin
                   count_out <=regS ;
                   termcount <= 1'b0;
      end
      else begin
                   count_out <= count_out - 1;
                   if (count_out == -1)
                     termcount <= 1'b1;
                   else
                     termcount <= 1'b0;
      end
     $display($time,,"rst_n = %b, termcount=%b, count_out=%h",
         rst_n, termcount, count_out);
     end
endmodule // signed_counter
module signed_counter_tb;
 wire             signed count_out; // From signed_counter of signed_counter.v
 wire                     termcount; // From signed_counter of signed_counter.v
 reg                      clk;
```

```
reg                         rst_n;

initial begin
    clk = 1'b1;
    forever #50 clk = ~clk;
end

initial begin
    #10 rst_n = 1'b1;
    #100 rst_n = 1'b0;
    #100 rst_n = 1'b1;
end

signed_counter #(4) signed_counter(
                                        // Outputs
                                        .count_out(count_out),
                                        .termcount(termcount),
                                        // Inputs
                                        .clk   (clk),
                                        .rst_n (rst_n));
endmodule // signed_counter_tb
```

Figure 6.2.2.1-2 Counter and Testbench with SIGNED Ports and Registers (*ch6/signed_counter.v)*

The results of *NC-Sim* simulation is show in Figure 6.2.2.1-3 and Figure 6.2.2.1-4.

```
   NC-Sim> run 10 us
  0 rst_n = x, termcount=x, count_out=x
100 rst_n = 1, termcount=0, count_out=x
200 rst_n = 0, termcount=0, count_out=x
300 rst_n = 1, termcount=0, count_out=b
400 rst_n = 1, termcount=0, count_out=a
500 rst_n = 1, termcount=0, count_out=9
600 rst_n = 1, termcount=0, count_out=8
700 rst_n = 1, termcount=0, count_out=7
800 rst_n = 1, termcount=0, count_out=6
900 rst_n = 1, termcount=0, count_out=5
```

```
1000 rst_n = 1, termcount=0, count_out=4
1100 rst_n = 1, termcount=0, count_out=3
1200 rst_n = 1, termcount=0, count_out=2
1300 rst_n = 1, termcount=0, count_out=1
1400 rst_n = 1, termcount=0, count_out=0
1500 rst_n = 1, termcount=0, count_out=f
1600 rst_n = 1, termcount=1, count_out=e
1700 rst_n = 1, termcount=0, count_out=d
1800 rst_n = 1, termcount=0, count_out=c
1900 rst_n = 1, termcount=0, count_out=b
```

Figure 6.2.2.1-3 Signed Counter Simulation Results (Cadence *NC-Sim*)

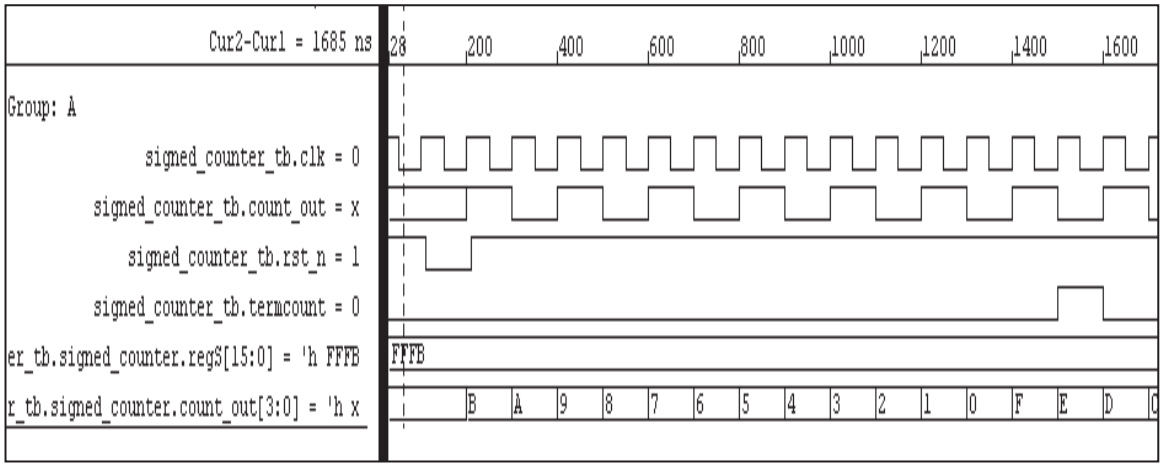

Figure 6.2.2.1-4 Signed Counter Simulation (Cadence *NC-Sim*)

The arithmetic shift operators (<<< and >>>) work the same as regular shift operators on unsigned objects. However, when used on *signed* objects, the following rules apply:

1. Arithmetic shift left ignores the signed bit and shifts bit values to the left (like a regular shift left operator), filling the open bits with zeroes.
2. Arithmetic shift right propagates all bits, including the signed bit, to the right while maintaining the signed bit value.

Figure 6.2.2.1-4 illustrates the Verilog 2001 shift concepts. Figure 6.2.2.1-5 demonstrates the simulation results.

```
module signed_shift;
  reg signed [3:0] start, result;
  initial
    begin
        $monitor($time,,"start=%b, result=%b", start, result);
      #10 start = -1;                  // Start is 1111
      #10 result = (start <<< 2);      // Result is 1100
      #10 result = (result <<< 1);     // Result is 1000
      #10 start = 5;                   // Start is 0101
      #10 result = (start <<< 2);      // Result is 0100
      #10 start = -3;                  // Start is 1101
      #10 result = (start >>> 1);      // Result is 1110
      #10 result = (result >>> 1);     // Result is 1111
      #10 result = (result >>> 1);     // Result is 1111
      #10 start = 3;                   // Start is 0011
      #10 result = (start >>> 1);      // Result is 0001
      #10 result = (result >>> 1);     // Result is 0000
    end
endmodule //signed_shift
```

Figure 6.2.2.1-4 Verilog 2001 Shift Concepts (*ch6/signed_shift.v*)

0 start=xxxx, result=xxxx	40 start=0101, result=1000	80 start=1101, result=1111
10 start=1111, result=xxxx	50 start=0101, result=0100	00 start=0011, result=1111
20 start=1111, result=1100	60 start=1101, result=0100	10 start=0011, result=0001
30 start=1111, result=1000	70 start=1101, result=1110	20 start=0011, result=0000

Figure 6.2.2.1-5 Simulation Results (Cadence *NC-Sim*)

7. MIXED SIMULATION AND SYNTHESIS

To enhance design reuse it is imperative to be able to perform mixed level simulation and synthesis of designs written in one HDL and incorporated into another level of hierarchy (i.e., instantiated) in another HDL. Verilog and VHDL design can be intermixed at the component or module level. This chapter addresses mixed mode simulation with Cadence *NC-Sim*[53], and mixed mode synthesis with Synplicity *Synplify Pro*. Other vendors perform similar integration methods, however users must check how each vendor implements this mixed mode integration.

There are two methods to import Verilog into VHDL or VHDL into Verilog:

1. With a shell
2. Without a shell.

A VHDL shell contains an entity/architecture pair in which the architecture consists of a *foreign* attribute that points to the compiled Verilog module. A Verilog shell is a Verilog module that contains a *foreign* attribute that points to the compiled VHDL architecture.

---

[53] Cadence NC-VHDL Simulator Help, Product Version 3.2