Signed Arithmetic in Verilog 2001 – Opportunities and Hazards

Dr. Greg Tumbush, Starkey Labs, Colorado Springs, CO

Introduction

Starkey Labs is in the business of designing and manufacturing hearing aids. The new digital hearings aids we design at the Starkey Labs Colorado IC Design Center utilize very complex DSP algorithms implemented in both software and hardware accelerators. The predominant data type used in these algorithms is signed. The format of the signed type is two's complement. The designation of signed and two's complement is used interchangeably throughout this document.

Verilog 2001 provides a very rich set of new signed data types. However, there are issues when performing operations such as sign extension, truncation or rounding, saturation, addition, and multiplication with signed values. These new data types (in theory) free the designer from worrying about some of these signed data type issues. More compact and readable code should result. However, in the spirit of Verilog, usage of this new functionality is "user beware"! Arithmetic manipulation between mixes of signed and unsigned may simulate and synthesize in unintended ways. Assignments between differently sized types may also not result in what the designer intended. Does the usage of signed data types in arithmetic operations result in smaller or larger circuits?

Verilog 1995 provides only one signed data type, integer. The rule is that if any operand in an expression is unsigned the operation is considered to be unsigned. The rule still applies for Verilog 2001 but now all regs, wires, and ports can be signed. In addition, a numeric value can be designated with a 's similar to the 'h hex designation. Signed functions are also supported as well as the type casting operators *$signed* and *$unsigned*. There are many new rules about when an operation is unsigned, and some may surprise you!

In this paper I will provide code examples of how the new signed data types can be used to create more compact code if some simple rules are followed. RTL and gate level simulation results of add and multiply operations using mixtures of signed and unsigned data types will be provided. Area results from synthesis using Design Compiler 2003.12 will be presented to compare efficiencies of these operations. Synthesis warnings that should be investigated thoroughly will be explained. Suggestions for improvement in the Verilog 2001 language, such as saturation support, will also be provided.

Signed Data Types

Table 1 demonstrates the conversion of a decimal value to a signed 3-bit value in 2's complement format. A 3-bit signed value would be declared using Verilog 2001 as *signed [2:0] A;*.

Decimal Value	Signed Representation
3	3'b011
2	3'b010
1	3'b001
0	3'b000
-1	3'b111
-2	3'b110
-3	3'b101
-4	3'b100

Table 1: Decimal to 3-bit Signed

Type Casting

The casting operators, *$unsigned* and *$signed*, only have effect when casting a smaller bit width to a larger bit. Casting using *$unsigned(signal_name)* will zero fill the input. For example *A = $unsigned(B)* will zero fill B and assign it to A. Casting using *$signed(signal_name)* will sign extend the input. For example, *A = $signed(B)*. If the sign bit is X or Z the value will be sign extended using X or Z, respectively. Assigning to a smaller bit width signal will simply truncate the necessary MSB's as usual. Casting to the same bit width will have no effect other than to remove synthesis warnings.

Signed Based Values

The only way to declare a signed value in Verilog 1995 was to declare it as an integer which limited the size of the value to 32-bits only[1]. Verilog 2001 provides the *'s* construct for declaring and specifying a sized value as signed. For example, 2 represented as a 3-bit signed hex value would be specified as *3'sh2*. Somewhat confusing is specifying negative signed values. The value -4 represented as a 3-bit signed hex value would be specified as *-3'sh4*. A decimal number is always signed.

Signed Addition

Adding two values that are n-bits wide will produce a n+1 bit wide result. The signed values must be sign

extended. For example, adding -2 (3'b110) to 3 (3'b011) will result in 1 (4'b0001). See the example in Figure 1.

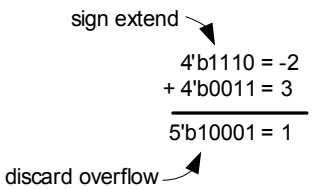

Figure 1: Basic Signed Addition Example

To do this addition using Verilog-1995 constructs we could use the code in Code Example 1.

```
module add_signed_1995 (
                        input [2:0] A,
                        input [2:0] B,
                        output [3:0] Sum
                        );
  assign                Sum = {A[2],A} + {B[2],B};
endmodule // add_signed_1995
```

Code Example 1: Addition - Verilog 1995

Or we can use the new signed type and get the code in Code Example 2.

```
module add_signed_2001 (
                        input signed [2:0] A,
                        input signed [2:0] B,
                        output signed [3:0] Sum
                        );
  assign                Sum = A + B;
endmodule // add_signed_2001
```

Code Example 2: Addition - Verilog 2001

Both adders are exactly the same size. So you will get the same results without having to worry about manually doing the sign extension.

Problems creep up when mixing signed and unsigned. Consider adding two 3-bit values with a 1-bit carry in. See Code Example 3 for a valid solution using Verilog 1995

```
module add_carry_signed_1995 (
                        input [2:0] A,
                        input [2:0] B,
                        input carry_in,
                        output [3:0] Sum
                        );
  assign                Sum = {A[2],A} + {B[2],B} + carry_in;
endmodule // add_carry_signed_1995
```

Code Example 3: Add with Carry - Verilog 1995

Intuitively we would create Code Example 4 to use signed types. However, when synthesized the following warning occurs: *signed to unsigned conversion occurs. (VER-318)* In addition there is a functional error. Due to the *carry_in* being unsigned the operation is unsigned and neither the *A* nor *B* operand is sign extended properly as in Figure 1.

```
module add_carry_signed_2001 (
                        input signed [2:0] A,
                        input signed [2:0] B,
                        input carry_in,
                        output signed [3:0] Sum
                        );
  assign                Sum = A + B + carry_in;
endmodule // add_carry_signed_2001
```

Code Example 4: Addition with Carry – Incorrect

We can avoid the synthesis warning by using *assign Sum = A + B + $signed(carry_in).* But this creates a different functional error. What happens if carry_in = 1? In this case the *$signed* operator sign extends the carry_in so it now equals 4'b1111 and we would have been subtracting 1 instead of adding 1. A similar functional error occurs if we use Code Example 4 but declare *carry_in* to be a signed input. See Code Example 5 for a valid solution. Using this code we avoid the synthesis warning and sign extend *carry_in* correctly with 0's.

```
module add_carry_signed_final (
                        input signed [2:0] A,
                        input signed [2:0] B,
                        input carry_in,
                        output signed [3:0] Sum
                        );
  assign      Sum = A + B + $signed({1'b0,carry_in});
endmodule // add_carry_signed_final
```

Code Example 5: Add with Carry - Correct

The code in Code Example 1 and Code Example 2 simulate the same with both RTL and gate level verilog. They are also the same size. The code in Code Example 3 and Code Example 5 simulate the same using both RTL and synthesized gate level verilog. They are also the same size. Code Example 4 is smaller in area but functionally incorrect.

Signed Multiplication

Multiplying two values that are n-bits wide will produce a 2n bit wide result. For example, multiplying -3 (3'b101) by 2 (3'b010) should result in -6 (6'b111010). The multiplier (second factor) is examined bit by bit right to left (least significant to most significant bit) to determine if the multiplicand (first factor) is to be added to the partial result. If so, the multiplicand is aligned so that the least significant bit is under the correct multiplier bit position. If

the multiplicand is negative it must be sign extended. However, if the MSB of the multiplier is 1, the multiplicand is actually subtracted. Recall that subtraction is the same as invert and increment. See the example in Figure 2.

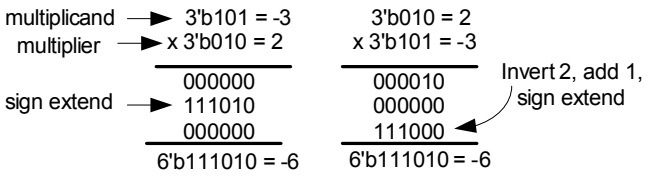

Figure 2: Signed Multiply Examples

Using Verilog-1995 constructs the code required to multiply two 3-bit signed values is in Code Example 6.

```
module mult_signed_1995 (
                input [2:0] a,
                input [2:0] b,
                output [5:0] prod
                );
  wire [5:0] prod_intermediate0;
  wire [5:0] prod_intermediate1;
  wire [5:0] prod_intermediate2;
  wire [2:0] inv_add1;
  assign     prod_intermediate0 = b[0] ? {{3{a[2]}}, a}       : 6'b0;
  assign     prod_intermediate1 = b[1] ? {{2{a[2]}}, a, 1'b0} : 6'b0;
  // Do the invert and add1 of a.
  assign     inv_add1 = ~a + 1'b1;
  assign     prod_intermediate2 = b[2] ? {{1{inv_add1[2]}},
                                          inv_add1, 2'b0} : 6'b0;
  assign     prod =  prod_intermediate0 +  prod_intermediate1 +
                                           prod_intermediate2;
endmodule
```

Code Example 6: Signed Multiply - Verilog 1995

Or we can use the new signed type and write the code in Code Example 7.

```
module mult_signed_2001 (
                input  signed [2:0] a,
                input  signed [2:0] b,
                output signed [5:0] prod
                );
   assign       prod = a*b;
endmodule
```

Code Example 7: Signed Multiply - Verilog 2001

Now, lets multiply a signed value by an unsigned value. Using Verilog 1995 constructs the code in Code Example 8 results. Now if we multiply –3 (3'b101) by 2 (3'b010) as usual with Code Example 8 we get –6 (6'b111010). When using a multiplier with one operand unsigned be sure of the range of input to the unsigned operand. If we tried to multiply 2 (3'b010) by –3 (3'b101) we would get 0xA because –3 is actually 5 unsigned. Note that because the multiplicand is unsigned this code is more compact and results in a smaller size multiplier.

```
module mult_signed_unsigned_1995 (
                input [2:0] a,
                input [2:0] b,
                output [5:0] prod
                );
  wire [5:0] prod_intermediate0;
  wire [5:0] prod_intermediate1;
  wire [5:0] prod_intermediate2;
  assign     prod_intermediate0 = b[0] ? {{3{a[2]}}, a}       : 6'b0;
  assign     prod_intermediate1 = b[1] ? {{2{a[2]}}, a, 1'b0} : 6'b0;
  assign     prod_intermediate2 = b[2] ? {{1{a[2]}}, a, 2'b0} : 6'b0;
  assign     prod =  prod_intermediate0 +  prod_intermediate1 +
                                           prod_intermediate2;
endmodule
```

Code Example 8: Signed by Unsigned Multiply - Verilog 1995

After migrating to Verilog 2001 we might be tempted to use Code Example 9. However, recall the rule that if any operand of an operation is unsigned the entire operation is unsigned. When synthesized the following warning occurs: *signed to unsigned conversion occurs. (VER-318).* Now if we multiply –3 (3'b101) by 2 (3'b010) as usual with this code we get 0xA (6'b001010). The reason for this is that since we mixed signed with unsigned we actually multiplied 5 by 2 and got 0xA since the operation is considered unsigned.

```
module mult_signed_unsigned_2001 (
                input  signed [2:0] a,
                input  [2:0] b,
                output signed [5:0] prod
                );
  assign       prod = a*b;
endmodule
```

Code Example 9: Signed by Unsigned Multiply - Incorrect

How about trying Code Example 10? This works for multiplying -2x3=-6 but what about if the MSB of our unsigned number = 1? In this case the multiplier is sign extended which is also incorrect. For the operation -2x7 we would get actually get 2 while the correct answer is 0x-E (6'b110010).

```
module mult_signed_unsigned_2001 (
                input  signed [2:0] a,
                input  [2:0] b,
                output signed [5:0] prod
                );
  assign       prod = a*$signed(b);
endmodule
```

Code Example 10: Signed by Unsigned Multiply - Still Incorrect

The correct answer to this problem follows from Code Example 5. Using this code we avoid the synthesis warning and sign extend *b* correctly with 0's. The correct code is in Code Example 11.

```
module mult_signed_unsigned_2001 (
                    input  signed [2:0] a,
                    input  [2:0] b,
                    output signed [5:0] prod
                    );
  assign          prod = a*$signed({1'b0,b});
endmodule
```

Code Example 11: Signed by Unsigned Multiply - Correct

Code Example 7 synthesizes to about 18% smaller than Code Example 6. I believe that this is because synthesis found a better implementation. There is no reason why we cannot replicate this size by more careful hand coding. The RTL and gate level implementation simulate the same.

Code Example 11 synthesizes to about 6% smaller than the code in Code Example 8. Once again, I believe that this is because synthesis found a better implementation. There is no reason why we cannot replicate this size by more careful hand coding. The RTL and gate level implementation simulate the same. The code in Code Example 9 and Code Example 10 are smaller in area but are functionally incorrect.

What is an expression?

The *Verilog-2001 LRM* states that to evaluate an expression "Coerce the type of each operand of the expression (excepting those which are self-determined) to the type of the expression"[2]. The question is what is an expression? Consider Code Example 12 which is directly from a Synopsys's SolvNet article 002590[3]. There are two ways to look at this code. It could be considered as two expressions, a signed multiply and then an unsigned addition. It can also be considered as one expression, an unsigned multiply followed by an unsigned addition. Results will differ in each case.

```
module mult_add (
              input signed [3:0] in1, in2,
               input [3:0] in3,
               output [7:0] o1;
                );
assign o1 =  in1 * in2 + in3;
endmodule
```

Code Example 12: Multiply and Add

It was reported that older versions of Design Compiler considered this code as two expressions while some simulators at the time considered it as one. Newer version of Design Compiler and ModelSim consider this code as one expression, alleviating a very worrisome simulation/synthesis mismatch. This issue is slated to be clarified in the upcoming Verilog 2005 LRM.

Rules for Expression Types

Located in the Verilog 2001 LRM but worth repeating here are the rules for determining the resulting type of an expression. The following operations are unsigned regardless of the operands.

1. Bit-select results
2. Part-select results, even if the entire vector is selected.
3. Concatenation results
4. Comparison results

Signed Shifting

Shifting of signed values creates another problem for Verilog 1995. Consider a signed negative value that is right shifted. The positions vacated by the right shift will be filled in with zeros which is incorrect. Instead, the sign bit should be used for vacated bits. A new operator >>> is introduced in Verilog 2001 to accomplish exactly this. A signed left shift operator (<<<) is also provided for language consistency[1].

Signed Saturation

In this section we present a concept that is widely used in DSP math but is not easily accomplished in Verilog. While sign extension is used when assigning a smaller bit-width variable to a larger bit-width variable, the opposite is accomplished using saturation. The possible outcomes of saturation are max positive indicating positive overflow, max negative indicating negative underflow, and simply dropping the appropriate number of bits starting at the MSB.

Saturation is accomplished by examining the number of bits to saturate plus 1 starting at the MSB. If all of these bits are the same drop the number of bits to saturate. If these bits are different examine the MSB. If the MSB is 0 go to max positive, else go to max negative. A module *sat* to accomplish this is in Code Example 13. Usage of this module is in Code Example 14.

```
module sat (sat_in, sat_out);

parameter IN_SIZE = 21;  // Default is to saturate 22 bits to 21 bits
parameter OUT_SIZE = 20;
input      [IN_SIZE:0] sat_in;
output reg [OUT_SIZE:0] sat_out;

wire [OUT_SIZE:0]   max_pos = {1'b0,{OUT_SIZE{1'b1}}};
wire [OUT_SIZE:0]   max_neg = {1'b1,{OUT_SIZE{1'b0}}};

always @* begin
   // Are the bits to be saturated + 1 the same?
   if ((sat_in[IN_SIZE:OUT_SIZE]=={IN_SIZE-OUT_SIZE+1{1'b0}}) ||
     (sat_in[IN_SIZE:OUT_SIZE]=={IN_SIZE-OUT_SIZE+1{1'b1}}))
           sat_out = sat_in[OUT_SIZE:0];
    else if (sat_in[IN_SIZE])  // neg underflow.  go to max neg
           sat_out = max_neg;
    else // pos overflow, go to max pos
           sat_out = max_pos;
  end
endmodule
```

Code Example 13: Saturation Module

```
wire signed [4:0] A, B, C;
reg signed [2:0] D, E, F;

A = 5'sb11101;
B = 5'sb01001;
C = 5'sb10001;

// Drop two MSB's. D will equal 3'sb101
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satA (.sat_in(A), .sat_out(D));

// Go to max positive.  E will equal 3'sb011
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satB
                (.sat_in(B), .sat_out(E));

// Go to max negative.  F will equal 3'sb100
sat  #(.IN_SIZE(4), .OUT_SIZE(2))  satC
                (.sat_in(C), .sat_out(F));
```

Code Example 14: Use of Saturation Module

Summary

This paper strove to give the user a strong background on the use of signed types using the Verilog 2001 language. The proper use of type casting, addition, multiplication, shifting, and truncation was presented. In addition, an example of a signed saturation module along with examples of it's use were included.

Proper use of the new signed capability in Verilog 2001 can be summarized by a few basic rules.

1. If any operand in an operation is unsigned the entire operation is unsigned[2].
2. Investigate fully all *signed to unsigned conversion occurs. (VER-318)* synthesis warnings. These point to incorrect functionality
3. All signed operands will be signed extended to match the size of the largest signed operand.
4. Type casting using *$unsigned* will make the operation unsigned. The operand will be sign extended with 0's if necessary.
5. Type casting using *$signed* make the operand signed. The operand will be sign extended with 1's if necessary. Pad the operand with a single 0 before the cast if this is not desired.
6. Expression type depends only on the operands or operation, it does not depend of the LHS of the expression.

References

1. S. Sutherland. *Verilog 2001 A Guide to the New Features of the Verilog Hardware Description Language.* Kluwer Academic Publishers

2. IEEE P1364-2005/D3. *Draft Standard for Verilog® Hardware Description Language.*

3. Synopsys Inc, *Synopsys Solvnet,* solvnet.synopsys.com