Verishort Language Reference Manual

Programming Languages and Translators
Professor Stephen Edwards
11/3/2010

Dictator: Anish Bramhandkar
Minions: Elba Garza
Scott Rogowski
Rujie Song

Introduction

Verilog is a very popular hardware description language (HDL) which is widely utilized by the electronics hardware design industry. First invented and used in the early 80s at Automated Integrated design Systems, Verilog was put into the public domain and standardized by the IEEE in 1995. This initial public version of Verilog became known as Verilog-95. The language was later expanded in 2001 and 2005 to address deficiencies and add features resulting in Verilog-2001 and Verilog-2005, the most recent version. (A combined hardware description / verification language known as SystemVerilog was extended from the 2005 standard but goes beyond the scope of this manual.)

Despite its popularity, Verilog is infamous for its repetitiveness, strange grammar, and ease of bug insertion. Part of this is a factor of the nature of low-level hardware design. There is a difference between languages meant to be run using gates and latches rather than processors and memory. However, we believe that another part of this simply poor language design and can be improved.

VeriShort HDL is meant to simplify the Verilog-2005 language to make it easier to read and write. First, we have reduced repetitiveness in accordance with the DRY (Don't repeat yourself) philosophy by simplifying module input/output syntax and instantiation. Next, we introduced some C-language features such as brackets and array-like bus descriptions. We substantially simplified synchronous logic by doing away with 'always' syntax and replacing it with simple 'if' statements. The list of reserved keywords has been substantially shortened in order to make VeriShort completely synthesizable and to remove rarely used features. Finally, we added a standard library of commonly used electronic components like latches, multiplexers, and decoders to further reduce the Verilog tedium.

Because of the wide adoption of Verilog and the existence of many verifiers and hardware synthesizers specific to the IEEE standards, the initial goal of VeriShort will not be to exist as a self contained HDL but rather to translate into clean synthesizable Verilog code. In support of these efforts, a translator has been started and is expected to be running by the date of December 22nd 2010.

Lexical conventions

Tokens

There are 5 classes of tokens: identifiers, keywords, numbers, operators, and other separators. Blanks, tabs, and newlines (collectively, whitespace) are ignored, except when they serve to separate tokens.

Comments

The characters `/*` introduce a comment, which is terminated by the characters `*/`.
The characters `//` also introduce a comment, which is terminated by the newline character.
Comments do not nest. Lines marked as comments are discarded by the compiler.

Data types

The primary data type is the bit, which may store the value `0` or `1`. A group of bits comprises a bus. All multibit binary values are treated as two's complement numbers.

In for-loops (see the corresponding section), the loop variable is assumed to be a simple integer (i.e., natural number).

Identifiers

An identifier is a sequence of characters that represent a wire, bus, register, parameter, or module. An identifier may only include alphanumerical characters or the underscore character (_). The first character of an identifier may not be a number.

Keywords

The following identifiers are reserved as keywords and may not be used for any other purpose: In this manual, keywords are **bolded**.

- **`case`**
- **`clock`**
- **`else`**
- **`for`**
- **`if`**
- **`input`**
- **`module`**
- **`negedge`**
- **`output`**
- **`parameter`**
- **`posedge`**

- **register**
- **return**
- **reset**
- **wire**

Numbers

Numbers can be either binary or integer values and are specified as follows:

- A sequence of digits, followed by a radix suffix (`b` for binary, or `d` for integer)
- If there is only a single `0` or `1`, no suffix needs to be provided because 0 and 1 are equivalent in integer and binary
- The characters `0` and `1` are valid binary digits.
- The characters `0-9` are valid integer digits.
- Extended binary numbers are like normal binary numbers, but may also use the character ‘x’ as a binary digit. They may only be used in case structures.

Operators

An operator is a token that specifies an operation on at least one operand. The operand may be an expression or a constant.

Bitwise operators:

- `!`
- `&`
- `!&`
- `|`
- `!|`
- `^`
- `!^`
- `>>`
- `<<`

Comparison operators:

- `==`
- `>=`
- `<=`
- `>`
- `<`

Arithmetic operators:

- `+`
- `-`
- `*`
- `/`
- `%`

Assignment operator:

- `=`

Sign extension operator:

- `'` (apostrophe)

The following operators are only valid within a for-loop:

- `++`
- `--`

Buses

A bus represents a multibit wire. The number of bits in a bus must be determinable at compile time. Buses are declared using the syntax `data_type bus_name[number_of_bits];`
Where data_type is either **`wire`**, **`register`** or assumed to be input or output by its position in a module declaration. The number of bits must be a constant. From here, any bit in the bus may be referred to using the subscript syntax: `bus_name[bit_index],` where `bit_index` is a constant or expression that yields an integer value less than or equal to the size of the bus.

A range of bits in a bus is represented by using the index of the most significant bit in the range, followed by the colon character (`:`), followed by the index of the least significant bit in the range, as the subscript. E.g. wires 4-8 would be referred to by `bus_name[7:3]`. Reversing this order is invalid.

Assignment

All assignments in Verishort bind wires and ports to other wires, binary values, or decimal values. These can be done en masse, such as in buses (multi-bit wires) or one by one.

Assigning wires

The most basic assignment is of a single wire to a single bit value:

```
wire w1 = 0;
wire w2 = !w1; // w2 == 1
```

A bundle of wires can be assigned to a multi-bit value, as long as the number of bits matches the number of wires in the bundle:

```
wire w3[5] = 01010b;    // assigning a binary value
wire w4[4] = 10d; // assigning a decimal value
wire w5[10] = {1,8{0},1}; // 1000000001b using concatenation

//This does not work and will result in an error because the left hand side
and the right hand side are not the same size.
wire w6[10] = 10b;
```

Note that the number of bits must *always* be specified for more than one bit.

Subsets of buses can be assigned:

```
wire w6[5];
w6[3:0] = 4d;
w6[4] = 1b; // w6 == 10100b
```

The two sides of an assignment must have the same number of bits.
The assignment operator returns the right operand and associates from right to left, making the following possible:

```
wire w7[5], w8[3];
w7[2:0] = w[8] = 010b;
```

Binding ports

Ports are bound to wires when instantiating a module by setting the module's parameter name equal to the wire or value to which it should be bound. Like assigning wires, these bindings can be in whole or in part. The value of a port, or part of a port, that has not been bound is assumed to be 0.

```
module m1(input in1[5], in2; output out[5]) { ... } // declared somewhere
// now, inside of a calling module
wire w7[5];
m1(in1[3:0] = w6[4:1], in2 = 1b; out = w7);
```

Expressions & Operators

The physical logic of VeriShort is described using expressions which are made up of one or more operators and operands. An operand can be either a single bit or a bus. All expressions will return a bit or bus that is then be assigned to a wire or output (see Assignment) or returned in an output. This section will detail operators ordered from the most basic building blocks to complex operations.

Concatenation, Replication, Splitting

To place two or more bits or buses together into a single bus, the concatenation syntax is used.

```
wire a = 0;
wire b = 1;
wire c[2] = 01b;
//{a,b,c,01b} results in 010101b
wire a1 = 1;
wire b1[4];
```

```
b1 = {4{a1}}; //results in 1111b, which is equivalent to {a1,a1,a1,a1}
```

Bitwise

Bitwise operators represent the primitive AND, OR, and NOT gates. All other logical are a combination of these operations. Every bitwise operation with the exception of the NOT gate is a binary operation in the “operand operation operand” style with both operands being the same size which is also the size of the return value. A NOT operation will return the same number of bits as its single operand.

Primitive bitwise operators ordered by precedence.

```
wire a = 01b;
wire b = 11b;
```

Operator	Example	Result
NOT	!a	10b
AND	a&b	01b
OR	a|b	11b

Full range of bitwise operators:

Operator	Example	Equivalency	Result
NAND	a!&b	!a & !b	00b
NOR	a!|b	!a | !b	10b
XOR	a^b	(!a & b) | (a & !b)	
XNOR	a!^b	(!a | b) & (a | !b)	

Parenthesis

Parenthesis has the highest precedence.

```
1|(1&0) //1
(1|1)&0 //0
```

Reduction

Reduction operators take only a single operand on their right hand side (a bus) and result in a

single bit result.

```
wire a[3] = 010b;
```

Operator	Example	Equivalency	Result
AND	&a	example[0] & example[1] & example[2]	-
NAND	!&a	!example[0] & !example[1] & !example[2]	0
OR	|a	example[0] | example[1] | example[2]	1
NOR	!|a	!example[0] | !example[1] | !example[2]	1
XOR	^example	example[0] ^ example[1] ^ example[2]	1
XNOR	!^example	!example[0] ^ !example[1] ^ !example[2]	1

Arithmetic

Arithmetic operators are shorthand for common equivalent but complex operations. They operate on two bits or buses which do not have to be the same size. They will return a bit or bus. All operations are done in two's complement.

In general, the bus that receives the result must contain enough bits to hold all bits in the result, or the result of the arithmetic operation may be undefined.

```
wire e0[3] = 011b //equivalent to 3d and can be expanded to 0011b
wire e1[3] = 111b //equivalent to -1d and can be expanded to 1111b
wire e3[3];
wire e4[4];
```

Operator	Example	Notes	Result
Plus	e3=e0+e1 e4=e0+e1	If there is overflow and the bus holding the result has insufficient bits, the result of the operation may be incorrect. If the resultant has n bits and the addends fewer, the addends will both be extended to n bits before addition occurs.	e3=010b e4=0010b
Minus	e3=e0-e1 e4=e0-e1	Equivalent to e0+{!e1[n],e1[n-1:0]}.	e3 = 100b e4 = 0100b
Multiplication	e0*e1	Returns a bus that is n+m-1 long	10101b

Division	e0/e1	not implemented	
Modulus	e0%e1	Returns a bus that is n long where n is the size of the first operand	0b11

Shifting

Shifting operations will literally shift the entire bus to the left or right and will discard the bits shifted off the end.

```
e0 = 0111b; //7d
e1 = 1111b; //-1d
```

Operator	Example	Note	Result
Left-shift	e0<<2 e1<<2	Left shift will always fill with zeros	1100b //-4d 1100b
Right-shift	e0>>2 e1>>2	Right shift will always fill with the most significant bit to preserve sign	0001b //1d 1111b //-1d

Sign extension

The sign extension operator sign-extends the right operand to the number of bits specified by the left hand side operand in decimal. The left operand must be determinable at compile time. Attempts to specify a number of bits that is smaller than the number of bits in the right operand is a syntax error. The operation is done in two's complement.

```
e0 = 0111b; //7d
e1 = 1111b; //-1d
```

Operator	Example	Note	Result
Sign extension	8'e0 8'e1	Left hand side is always a decimal number.	00000111b 11111111b

Comparison

Comparison operators will compare the values of two buses which do not need to be equally sized and will return a one bit true or false result

```
e0 = 0111b; //7d
e1 = 0111b; //7d
e2 = 1111b; //-1d
```

Operator	Example	Result
Less than	`e0 < e1`	0
Less than or equal to	`e0 <= e1`	1
Greater than	`e0 > e2`	1
Greater than or equal to	`e0 >= e2`	1
Equal to	`e0 == e1`	1
Not equal to	`e0 != e1`	0

Precedence and associativity

Operators, in order of decreasing precedence	Associativity
++ -- (postfix, for-loops only)	left to right
! + - & !& ^ !^ | !|(unary) ' (sign extension) ++ -- (prefix, for-loops only)	right to left
* / %	left to right
+ - (binary)	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right

& !& (binary)	left to right
^ !^ (binary)	left to right
| !| (binary)	left to right
=	right to left

Declarations

There are three types of declarations in Verishort: wires, for passing values between modules; registers, for storing values between clock cycles; and modules, blocks of Verishort code offering specific functionality.

Identifiers

Identifiers are user-friendly names, much like variable names in most programming languages. They must start with an upper- or lower-case letter followed by any sequence of letters, numbers, and underscores. No other characters are allowed in identifier names.

Wire declarations

Wires can be single or multi-bit (buses/bundles). They are declared using the keyword **`wire`** followed by a space, then an identifier, then an optional number inside square brackets. The number inside the brackets is the number of bits to be used for the bus, using 0-indexing. The MSB is at the highest bit value, i.e., at index 15 for a 16-bit wire. If the brackets and enclosed number are omitted, the wire is assumed to hold a single bit.

```
wire w1; // single bit
wire w2[5]; // five bits with MSB at index 4, LSB at index 0
```

In addition, the declared wires can take values immediately during the declaration. Following the identifier (or optional bracketed expression) another space, an equals sign, a space, a binary, decimal, or logical value can be used:

```
wire w3 = !w2[0];
wire w4[3] = 4d;
```

The shorthand for digit repetition can also be used:

```
wire w5[16] = {1,14{0},1};
```

Subsets of buses can be assigned:

```
wire w6[8] = w5[0:7];
```

Equivalently:

```
for(i = 0; i < 8; i++) {
  w6[i] = w5[i];
}
```

Register declarations

Registers are declared exactly like wires but with the **`register`** keyword. The main difference between registers and wires are that registers can be reassigned at will at any point in your code. They can be used to store values between clock cycles, for example.

Module declarations

Modules are analogous to classes in Java. Modules allow code to be grouped to offer specific functionality.

They are declared with the **`module`** keyword, followed by an identifier, followed by a parenthesized expression. The parenthesized expression contains a comma-separated list of input parameters preceded by the **`input`** keyword, followed by a semi-colon, followed by a comma-separated list of output parameters preceded by the **`output`** keyword. The body of the module (discussed below) followed inside braces

```
module m1(input in1, in2, in3[3]; output out[5]) { ... }
```

In addition to the user-specified inputs and outputs (see the *Assignment* section for information on binding parameters), each module has an implicit reset input (keyword **`reset`**) and clock input (keywords **`posedge`** and **`negedge`**).

Modules can be declared in any order but all code must reside in one file. Modules cannot be nested, i.e., one module cannot be declared inside of another module. However, a module can be instantiated (*Instantiating a module* below) inside of another module as long as there are no infinitely recursive references.

Building a module

A module is a series of declarations, assignments, logic, and conditional statements.
All declarations in a module must appear at the beginning of the module, before any other statements.
Here is a brief example:

```
module m2(input enable, in[4]; output out[4]) {
  register r[4] = 0d;
  if(posedge) {
    if(reset) {
      r = 0000b; // equivalent to r = 0d;
    }
  }
  else if(enable == 1b) {
    r = in;
```

```
    out = r;
  }
}
```

This is a simple four-bit latch. On every positive clock edge, if the enable bit is set to 1, the output will be set to the input. If the enable bit is set to 0, the output will remain unchanged. However, if the module's reset bit is 1 when the clock edge is detected, the register values and output will be reset to `0d`.

This module is reusable. Here is an example of instantiating the module from within another module:

```
module m3(input check1, check2, in[4]; output out[4]) {
  wire enabler, resetter;
  m2(enable = enabler, in = in; out = out; reset = resetter);

  if(posedge) {
    enabler = check1;
    resetter = check1 & check2;

  }
}
```

The enable bit of module m2 is 1 if input check1 in module m3 is 1. If both check1 and check2 are 1, then the module m2 is reset. check2 on its own does nothing.

Notice that though reset is not listed as an input of module m2, it is implicit; it can be referred to by keyword reset.

In addition to binding wires to outputs, subsets of outputs can be returned as in traditional languages. To indicate that a module returns *n* bits, put the number of bits to be returned (as in a **`wire`** declaration) in square brackets after the identifier, including for a single bit returned, unlike wires.

If you wish to return all or a subset of outputs (replacing or in addition to the normal outputs via binding), list them after the **`return`** keyword anywhere in a block. A single block may not have multiple **`return`** keywords.

```
module m10[5] (input in[5]; output out[3]) {
  ...
  out = ... ;
  return {out,in[1],in[2]};
}
```

These can then be used as follows:

```
wire w11[5] = m10(...);
```

Statements

A semicolon is necessary after a statement in Verishort. Because whitespace has no effect, it is necessary to have semicolons to signal the end of a statement.

Examples from previous sections:

```
wire w3 = !w2[0];
wire c[2] = 01b;
```

For some statements, such as connectivity, a statement is inherently acknowledged through the brackets.

Example from previous sections:

```
wire a = 1;
wire b[4];
wire c[16] = {0{b},0,0,10b,4{0,a},b} // results in 0010010101011111b
b = {4{a}}; // results in 1111b
```

Conditional statements

Conditional statements work just as they do in the C programming languages with **`if`** and **`else`**. Following an **`if`**, an expression is placed within parenthesis. This expression must evaluate to a bit value. An expression returning 1 evaluates as the equivalent of ‘true’ and 0 as ‘false’. An **`else`** block attaches itself to the closest **`else`**-less **`if`** block.

```
wire gate = 1;
wire b[2];
if (gate & 1b > 0b) {
      b = 10b;
      }
else {
      b = 01b;
      }
```

The power of conditional statements come in their ability to use the clock. For example, an incrementer:

```
register a[8] = 0;
if (posedge) {
      a = a + 1b;
```

```
      }
```

Asynchronous and synchronous logic can also be combined together seamlessly:

```
wire gate = 1;
register a[8] = 0;
if (posedge & gate) {
      a = a + 1b;
      }
```

Case/switch structures

Case structures use the **case** keyword and work similarly to the switch statement in C. The main difference is that the case structure in Verishort does not provide fall-through or default behavior.

This is especially useful when the user wants to test for conditions on certain bits but doesn't care about the value of other bits. Those bits can be replaced with **x** instead of 1 or 0.

Inside of a case block, the condition is followed by a colon, then followed by the resulting statement, followed by a semicolon.

If-else statements and for-loops cannot appear inside case structures.

```
wire w[3] = ... // also assume a 3-bit output 'out'
case(w) {
  1x1b : out = 111b;
  x00b : out = 000b;
}
```

For-loops

For loops in Verishort are different from for loops in C. Instead of being used to repeat a task multiple times, they are instead used to repeat tedious code. For example, to wire every other bit of the wire 'a' to a module and output the result to b:

```
wire a[32];
wire b[16];
for (i = 0; i < 16; i++) {
      example_module(in=a[i*2],out=b[i]);
}
```

This code has the effect of creating 16 example_modules which are all wired.

Scope

VeriShort tends to be a linear language, with very little dependence on lexical scope and linkage. That being said, VeriShort still limits scope for certain conditions. Importantly, data declared within one module is not available outside of that module. Similarly, data declared within blocks (between the brackets in 'if/else' blocks and 'for' loops) is not available outside of those blocks.

Preprocessor

Like the #define directive in C, the **parameter** keyword can be used in VeriShort to replace numbers before compile time. For example, in the following code:

```
parameter const = 1010b;
wire c[4] = const;
```

The const identifier is literally replaced with the number 1010b before compile time. As with any assignment, notice that `c` contains the correct number of bits to hold the constant.

To reduce the repetitiveness of Verilog, we automatically assume clock and reset ports in all modules that use their functionality. So, a clock input port is included in any module that uses the **posedge** or **negedge** keywords. Similarly, a reset input port is included in any module that uses registers to the effect that asserting a reset will set all registers back to 0. These ports can be overloaded by simple specifying a different value for them. This is useful for gating the clock like in the following example:

```
module gater(input gate, i; output o) {
      add_on_clk(in=i;out=o);
      clock = clock & gate;
      }
```

Assuming that the add_on_clk module actually uses a clock, this also has the effect of changing the clock defintion in the add_on_clock and any other module instantiated within the gater module.

Standard Library Modules

Because of the repetitiveness of many aspects of hardware design, the Verishort language includes standard modules of many commonly used electronic components in the hope of reducing some tedium. The definitions given below are pseudo-module declarations. The appropriate value of n will be selected at compile time. Note that, like any module port, clk and

reset do not need to be specified. By Verishort convention, a clk is explicitly assumed in all modules that use the posedge/negedge keywords. A reset is explicity assumed in all modules that use the reset keyword.

Latches

```
module SRL[n] (input S[n], reset[n], E[n]; output Q[n], QNOT[n])
module JKL[n] (input J[n], K[n], E[n]; output Q[n], QNOT[n])
module DL[n] (input D[n], E[n]; output Q[n], QNOT[n])
module TL[n] (input T[n], E[n]; output Q[n], QNOT[n])
```

Flipflops

```
module DFF[n] (input D[n], S[n], reset[n], clk; output Q[n], QNOT[n])
module TFF[n] (input T[n], clk; output Q[n], QNOT[n])
module JKFF[n] (input J[n], K[n], clk; output Q[n], QNOT[n])
```

Multiplexers

```
module MUX[1] (input IN[n],SEL[log2 n]; output OUT)
module DEMUX[n] (input IN[n], SEL[log2 n]; output OUT[n])
```

Decoders

```
module DECODE[n] (input IN[log2 n]; output OUT[n])
module ENCODE[log2 n] (input IN[n]; output OUT[log2 n]) //Priority Encoder
```