Verilog: `wire` vs. `reg`

Chris Fletcher
UC Berkeley
Version 0.2008.9.25

August 27, 2009

1 Introduction

Sections 1.2 to 1.4 discuss the difference between `wire` and `reg` in Verilog, and when to use each of them.

1.1 Conventions

Throughout this tutorial, similar terms are used to describe different concepts. These possible sources of confusion are explained below:

Module Instantiation : Refer to modules instantiated in other modules. Program 1 shows examples of input/output ports for a simple module instantiation.

Module Declaration : Refer to the actual Verilog code written for a module. Program 2 shows examples of inputs/outputs within a module declaration. Notice that each input and output is declared twice in the code. This is important so that inputs and outputs can be assigned **p**arameterizable widths (in this example, 'Width' is used as an example parameter).

Program 1 Module instantiation.

```
Register #(      .Width( ...))
         SentReg(.Clock( ...), // Input port
                 .Reset( ...), // Input port
                 .Set(   ...), // Input port
                 .Enable(...), // Input port
                 .In(    ...), // Input port
                 .Out(   ...)); // OUTPUT port
```

Program 2 Module declaration.

```
module  MyModule(In, Out);
    ...
        parameter Width = 32;
    ...
    input [Width-1:0] In, Out;
    ...
endmodule
```

1.2 `wire` Elements (Combinational logic)

`wire` elements are simple wires (or busses/bit-vectors of arbitrary width) in Verilog designs. The following are syntax rules when using `wire`s:

1. `wire` elements are used to connect `input` and `output` **p**orts of a module **i**nstantiation together with some other element in your design.
2. `wire` elements are used as `input`s and `output`s within an actual module declaration.
3. `wire` elements must be driven by something, and cannot store a value without being driven. In other words, `wire` elements are a stateless way of connecting two peices in a Verilog-based design.
4. `wire` elements cannot be used as the left-hand side of an `=` or `<=` sign in an `always@` block.
5. `wire` elements are the only legal type on the left-hand side of an `assign` statement.
6. `wire` elements can only be used to model **c**ombinational logic.

Program 3 shows various legal uses of the `wire` element.

Program 3 Legal uses of the `wire` element

```
wire        A, B, C, D, E; // simple 1-bit wide wires
wire [8:0] Wide;     // a 9-bit wide wire
reg I;

assign A = B & C;    // using a wire with an assign statement

always @(B or C) begin
    I = B | C;       // using wires on the right-hand side of an always@
                     // assignment
end

mymodule MyModule(.In (D), // using a wire as the input of a module
                  .Out(E)); // using a wire as the output of a module
```

1.3 `reg` Elements (Combinational and Sequential logic)

`reg` are similar to wires, but can be used to store information ('state') like registers. The following are syntax rules when using `reg` elements.

1. `reg` elements **c**an be connected to the input **p**ort of a module **i**nstantiation. Note that `reg` **c**annot connect to the **o**utput port of a module instantiation.
2. `reg` elements **c**an be used as `output`s within an actual module declaration. Note that `reg` **c**annot be used as `inputs`s within an actual module declaration.
3. `reg` is the only legal type on the left-hand side of an `always@` block `=` or `<=` sign.
4. `reg` is the only legal type on the left-hand side of an `initial` block `=` sign (used in Test Benches).
5. `reg` **c**annot be used on the left-hand side of an `assign` statement.
6. `reg` can be used to create registers when used in conjunction with `always@(posedge Clock)` blocks.
7. `reg` can, therefore, be used to create both **c**ombinational and sequential logic.

Program 4 shows various legal uses of the `reg` element.

Program 4 Legal uses of the `reg` element

```
wire      A, B;
reg       I, J, K;    // simple 1-bit wide reg elements
reg[8:0] Wide;        // a 9-bit wide reg element

always @(A or B) begin
    I = A | B;        // using a reg as the left-hand side of an always@
                      // assignment
end

initial begin         // using a reg in an initial block
    J = 1'b1;
    #1
    J = 1'b0;
end

always @(posedge Clock) begin
    K <= I;           // using a reg to create a positive-edge-triggered register
end
```

1.4 When `wire` and `reg` Elements are Interchangable

`wire` and `reg` elements can be used interchangably in certain situations:

1. Both can appear on the right-hand side of `assign` statements and `always@` block `=` or `<=` signs.
2. Both can be connected to the input **p**orts of module instantiations.