Summary of Synthesisable Verilog 2001

Numbers and constants

Example: 4-bit constant 11 in binary, hex and decimal:

```
  4'b1011 == 4'hb == 4'd11
```

Bit concatenation using `{}`:

```
  {2'b10,2'b11} == 4'b1011
```

Note that numbers are unsigned by default.

Constants are declared using `parameter` vis:

```
  parameter foo = 42
```

Operators

Arithmetic: the usual `+` and `-` work for add and subtract. Multiply (`*`) divide (`/`) and modulus (%) are provided by remember that they may generate substantial hardware which could be quite slow.

Shift left (<<) and shift right (>>) operators are available. Some synthesis systems will only shift by a constant amount (which is trivial since it involves no logic).

Relational operators: equal (==) not-equal (`!=`) and the usual `< <= > >=`

Bitwise operators: and (&) or (`|`) xor (ˆ) not (˜)

Logical operators (where a multi-bit value is false if zero, true otherwise): and (&&) or (`||`) not (`!`)

Bit reduction unary operators: and (&) or (`|`) xor (ˆ)
Example, for a 3 bit vector `a`:

```
   &a == a[0] & a[1] & a[2]
```
and `|a == a[0] | a[1] | a[2]`

Conditional operator `?` used to multiplex a result
Example: `(a==3'd3) ? formula1 : formula0`
For single bit formula, this is equivalent to:

```
     ((a==3'd3) && formula1)
  || ((a!=3'd3) && formula0)
```

Registers and wires

Declaring a 4 bit wire with index starting at 0:

```
  wire [3:0] w;
```

Declaring an 8 bit register:

```
  reg [7:0] r;
```

Declaring a 32 element memory 8 bits wide:

```
  reg [7:0] mem [0:31]
```

Bit extract example:

```
  r[5:2]
```
returns the 4 bits between bit positions 2 to 5 inclusive.

Assignment

Assignment to wires uses the `assign` primitive *outside* an `always` block, vis:

```
  assign mywire = a & b
```

This is called *continuous assignment* because `mywire` is continually updated as `a` and `b` change (i.e. it is all combinational logic).

Registers are assigned to *inside* an `always` block which specifies where the clock comes from, vis:

```
always @(posedge clock)
  r<=r+1;
```

The `<=` assignment operator is none blocking and is performed on every positive edge of `clock`. Note that if you have whole load of none blocking assignments then they are *all updated in parallel*.

Adding an *asynchronous reset*:

```
  always @(posedge clock or posedge reset)
    if(reset)
      r <= 0;
    else
      r <= r+1;
```

Note that this will be synthesised to an asynchronous (i.e. independent of the clock) reset where the reset is connected directly to the clear input of the DFF.

The *blocking assignment* operator (=) is also used inside an `always` block but causes assignments to be performed as if in sequential order. This tends to result in slower circuits, so we *do not used it for synthesised circuits*.

Case and if statements

`case` and `if` statements are used inside an `always` block to conditionally update state.

Example:

```
  always @(posedge clock)
    if(add1 && add2) r <= r+3;
    else if(add2) r <= r+2;
    else if(add1) r <= r+1;
```

Note that we don't need to specify what happens when `add1` and `add2` are both false since the default behaviour is that r will not be updated.

Equivalent function using a `case` statement:

```
  always @(posedge clock)
    case({add2,add1})
      2'b11  : r <= r+3;
      2’b10  : r <= r+2;
      2’b01  : r <= r+1;
      default: r <= r;
    endcase
```

And using the conditional operator (?):

```
  always @(posedge clock)
    r <= (add1 && add2) ? r+3 :
                   add2 ? r+2 :
                   add1 ? r+1 : r;
```

Which because it is a contrived example can be shortened to:

```
  always @(posedge clock)
    r <= r + {add2,add1};
```

Note that the following would not work:

```
  always @(posedge clock) begin
    if(add1) r <= r + 1;
    if(add2) r <= r + 2;
  end
```

The problem is that the none blocking assignments must happen in parallel, so if `add1==add2==1` then we are asking for `r` to be assigned `r+1` and `r+2` simultaneously which is ambiguous.

Module declarations

Modules pass inputs and outputs as wires only. If an output is also a register then only the output of that register leaves the module as wires.

Example:

```
module simpleClockedALU(
        input clock,
        input [1:0] func,
        input [3:0] a,b,
        output reg [3:0] result);
  always @(posedge clock)
    case(func)
      2’d0    : result <= a + b;
      2’d1    : result <= a - b;
      2’d2    : result <= a & b;
      default : result <= a ˆ b;
   endcase
endmodule
```

Example in pre 2001 Verilog:

```
module simpleClockedALU(
     clock, func, a, b, result);
  input  clock;
  input  [1:0] func;
  input  [3:0] a,b;
  output [3:0] result;
  reg    [3:0] result;
  always @(posedge clock)
    case(func)
      2’d0    : result <= a + b;
      2’d1    : result <= a - b;
      2’d2    : result <= a & b;
      default : result <= a ˆ b;
   endcase
endmodule
```

Instantiating the above module could be done as follows:

```
wire clk;
wire [3:0] data0,data1,sum;

 simpleClockedALU myFourBitAdder(
    .clock(clk),
    .func(0),     // constant function
    .a(data0),
    .b(data1),
    .result(sum));
```

Notes:

- `myFourBitAdder` is the name of this instance of the hardware
- the `.clock(clk)` notation refers to:
  `.port_name(your_name)`
  which ensures that values are wired to the right place.
- in this instance the function input is zero, to the synthesis system is likely to simplify the implementation of this instance so that it is only capable of performing an addition (the zero case)

Simulation

Example simulation following on from the above instantiation of `simpleClockeALU`:

```
reg clk;
reg [7:0] vals;
assign data0=vals[3:0];
assign data1=vals[7:4];

// oscillate clock every 10 simulation units
always #10 clk <= !clk;

// initialise values
initial #0 begin
  clk = 0;
  vals=0;
// finish after 200 simulation units
  #200 $finish;
end

// monitor results
always @(negedge clk)
  $display("%d + %d = %d",data0,data1,sum);
```

Simon Moore