Verilog: a brief history

c.1977 First HDLs developed
1985 Verilog introduced by Gateway Design Automation
1990 Gateway purchased by Cadence Design Systems
1991 ‘Opened’ as a language
1995 Made into IEEE standard #1364
2001 Second IEEE standard including extensions and improvements
2005 A new standard with some minor improvements
2005 SystemVerilog introduced: Verilog-2005 plus major extensions ⇒ IEEE #1800
2009 SystemVerilog IEEE Std. 1800-2009 which merges 1364-2005 and 1800-2005

The version used in these modules is Verilog-2001.

[The schematic capture tools used are not compatible with all the features of SystemVerilog.]

Verilog versions

Most of the information in this module is based on Verilog-2001. This is probably, currently (2010) the most common version although SystemVerilog is gaining ground.

These pages cover some of the significant differences in the major versions.

Verilog-95 is becoming obsolescent. Examples of features absent include:
- Fewer syntactic options e.g. always @ (a or b or c) 2001 added always @ (a, b, c) and always @ (*)
- No signed registers, shifts etc.
- I/O type declarations inside port list
  module and2 (a, b, q) // This form for V-95
  input a, b;
  output q;
...
- Multi-dimension arrays
- Generate blocks
- Parameter passing by name
- Most of the file support

This is not an exhaustive list; rather it is intended to be indicative of the type of enhancements. Verilog-2001 is backwards compatible.

Some of these features may be unfamiliar, at least in Verilog, the first time you read this. You don’t need to use them, although there will be some brief coverage, shortly.

SystemVerilog

SystemVerilog attempts to combine the merits of Verilog HDL with a Hardware Verification Language (HVL). It is somewhat C++-like. A selection of additional features include:
- Enumerated data types
- Multi-dimensional arrays (bundles of buses)
- Structures/Unions
- Interfaces (related collections of I/O signals)
- Object-oriented programming
- Assertions
- Coverage features
- Improved thread synchronization
- ...

Other ‘existing’ Verilog features have also been enhanced/regularised in places. Some of these features are for synthesizable RTL code, others are there to improve the modelling and verification features.

The current standard is 2009; not all toolsets have caught up with this yet.

We don’t use SystemVerilog in the lab. because the Cadence implementation does not interoperate with some of the other tools we want to use. (As far as we can make them work!)

VHDL

VHDL is a different HDL which has some popularity. In principle it works largely in the same way as Verilog. It has some advantages, such as stronger typing, and disadvantages, such as verbosity. However converting from one to the other is largely a matter of new syntax.

For further opinions there are numerous VHDL vs Verilog discussions out there in Internet Land.
Verilog …

- … should be, to some extent, already familiar
- … is a Hardware Description Language
  - for *modelling* hardware
- … is designed for parallel programming
- … can be synthesized into circuits
  - providing an *appropriate subset* is used
- … is used as a portable design format

In this module we use Verilog for:

- writing tests
- implementing hardware
- (invisibly) as part of the tool flow

---

### Verilog: a reminder

These pages are provided as a brief *aide memoire*. There are many books and web tutorials which give a more complete description. A suggested, on-line tutorial may be found at: [http://www.asic-world.com/verilog/](http://www.asic-world.com/verilog/)

This not meant to be comprehensive; you may meet some new material later!

#### Declarations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire aaa;</td>
<td>A simple, combinatorial logic variable.</td>
</tr>
<tr>
<td>reg bbb;</td>
<td>A variable which may (or may not) be state holding.</td>
</tr>
<tr>
<td>reg [7:0] ccc;</td>
<td>As above, but an eight bit value.</td>
</tr>
</tbody>
</table>

#### Values

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>A decimal number.</td>
</tr>
<tr>
<td>16'hABCD</td>
<td>A 16-bit hexadecimal number.</td>
</tr>
<tr>
<td>4'b1010</td>
<td>A 4-bit binary number.</td>
</tr>
<tr>
<td>x</td>
<td>The ‘unknown’ value.</td>
</tr>
<tr>
<td>z</td>
<td>The high-impedance state.</td>
</tr>
</tbody>
</table>

#### Operators

Operators are, in general the same as in C or Java. There are some additional features:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;ccc</td>
<td>A (unary) reduction operator; ANDs all the bits in the variable together. Others include ‘|’, ‘^’, ‘~&amp;’, etc.</td>
</tr>
<tr>
<td>{ddd, eee}</td>
<td>Concatenate two (or more) variables.</td>
</tr>
<tr>
<td>{4{ddd}}</td>
<td>Concatenate four copies of ddd.</td>
</tr>
</tbody>
</table>

### Assignment

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign a = b + c;</td>
<td>A continuous assignment; combinatorial logic. Assigns to wire.</td>
</tr>
<tr>
<td>d = e &amp; f;</td>
<td>Blocking assignment. d may be reassigned in block, reading top-to-bottom. Use for combinatorial logic. Assigns to reg.</td>
</tr>
<tr>
<td>g &lt;= h;</td>
<td>Non-blocking assignment. g should be assigned at most once at any time. Use for registers. Assigns to reg.</td>
</tr>
</tbody>
</table>

### Execution

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>Execute following statement once, starting at time = 0.</td>
</tr>
<tr>
<td>always</td>
<td>Run following statement continuously. (Needs a delay?)</td>
</tr>
<tr>
<td>always @ {...}</td>
<td>Run following statement whenever an event in the parenthesized list occurs.</td>
</tr>
</tbody>
</table>

### Control

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (...) ... [else] ...</td>
<td>C/Java-like choice.</td>
</tr>
<tr>
<td>case</td>
<td>Multi-way ‘switch’ choice.</td>
</tr>
<tr>
<td>repeat (10)</td>
<td>Perform following statement ten times.</td>
</tr>
<tr>
<td>while (...) ...</td>
<td>C-like repetition.</td>
</tr>
<tr>
<td>for (......) ...</td>
<td>C-like repetition.</td>
</tr>
</tbody>
</table>

### Timing

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>#20</td>
<td>Wait for 20 time units.</td>
</tr>
</tbody>
</table>

Some of these constructs are useful for testbenches but are not synthesizable.

---

Verilog reminder: a testbench

```verilog
initial clk = 0; // Set clock initial value
always #5 clk = ~clk; // Oscillate with 10 unit period
initial #1000 $stop; // Limit simulation run time

initial
begin // Bracket following into one statement
reset = 1; // Initialise the Device Under Test
en = 1; // Set up an initial input value
#10 // Wait one clock cycle
reset = 0; // Remove reset
while (c == 0) #10; // Wait until output 'c' asserted
en = 1'x; // Make input 'en' undefined
end // End of statement (but time continues)
```

Verilog for testbenches

This testbench assumes the existence of some design which has defined the variables. It is reminiscent of the stimulus files used in testing in Cadence.

In the example on the slide there are four separate blocks. The three ‘initial’ blocks start at time = 0 and run once. Thus the first statement initialises an input and stops. The other ‘initial’ statements contain delays which can deschedule them for periods of (simulation) time.

The always block has no sensitivity list so it runs immediately. A conflict with the preceding ‘initial’ assignment is avoided by the delay. Following the delay the clock is inverted and the always runs again; thus the clock oscillates. The second ‘initial’ block waits for a (long) time and then tells the simulator to halt. Without this the simulation would run forever – or until it is halted by user intervention.

Note: it would be a mistake to write something like:

```verilog
always clk = ~clk;
```

What would happen?

Would the simulation terminate?

The method shown is not the only way to achieve the desired effect. For example the clock (and termination) may be produced by:

```verilog
initial
begin
clk = 0;
repeat (200) #5 clk = ~clk;
$stop;
end
```

You may prefer this style.

Verilog language

Choices

As in most programming languages, there are multiple ways in which a particular effect can be achieved. The choice of structure will be influenced by the author’s stylistic prejudices. The best advice is to try and make things as simple and obvious as you can.

```verilog
wire p; assign p = !a; // This?
reg q; always @ (a) q = !a; // or this?
```

Because Verilog syntax has evolved from (arguably) crude beginnings but maintained backward compatibility there are sometimes multiple ways to specify the same thing.

Example

```verilog
module inv1(input wire a, output wire q);
assign q = !a;
endmodule;
module inv2(a, q);
input a;
output q;
wire a, q;
assign q = ~a;
endmodule;
```

Both these modules have the same effect.

Notes

- These notes are not intended as a comprehensive guide to Verilog.
- Introductions to Verilog features may be found in a number of books and web tutorials.
  - e.g. http://www.asic-world.com/verilog/
- For a definitive guide, consult the IEEE standard document.
Verilog reminder: a circuit

module counter (input wire clk, input wire en,
        output reg [3:0] count, output wire c);

reg [3:0] next; // Internal variable (combinatorial)

always @ (count) // Run block when ‘count’ changes
        begin
    next = count + 1; // Blocking assignment
    if (count == 9) next = 0; // Variable may be reassigned
end

always @ (posedge clk, posedge reset)
        if (reset) count <= 0; // Asynchronous reset
        else if (en)
            if (count == 9) count <= 0;
            else count <= next;
        else count <= count; // This clause may safely be omitted

assign c = (count == 9); // TRUE is a ‘1’ value
endmodule

Verilog for synthesis

The example is contrived to illustrate a number of different Verilog features.

Notes

‘next’ is sometimes reassigned in its block but it is always assigned, thus the combinatorial block always calculates its result(s) from its input(s). This guarantees it is combinatorial and does not need any state holding.

The description of the register, with its asynchronous clear, is one which will be recognised by the synthesizer.

The register (‘count’) does not need the second else; if it is not enabled it will do nothing. There is already a state-holding element here so this doesn’t affect the circuit.

The carry output (‘c’) is assigned combinatorially from a comparator. In practice this may ‘glitch’ as its input bits change. This doesn’t matter as long as it’s only sampled on a clock edge. It should not be used as a clock, itself.

It is also possible, and sometimes clearer, to incorporate some of the logic in the same statement as the register assignment.

always @ (posedge clk, posedge reset)
        if (reset) count <= 0; // Asynchronous reset
        else if (en)
            if (count == 9) count <= 0;
            else count <= count + 1;

The choice of style can depend on the author. This is shorter than the example on the slide although the variable ‘next’ or equivalent – the state that is about to be entered – is sometimes a convenient input for other logic … or for viewing for debug purposes.

Guidelines

*** You should read this page! ***

Some ‘features’ of Verilog are open to abuse. It is possible to write Verilog code in numerous ways. Some ways are error-prone; they ‘look’ right but behave oddly, or are unduly expensive to build. Some models will run on a simulator. Some will run on particular simulators, but not on others. Some will run when synthesised into hardware. Some will be reliable across all tools and implementations.

Here are some guidelines to help you achieve the last of these cases.

- Verilog is case sensitive and ‘wire’ variables do not have to be declared. Thus:
  
  wire this, that;
  assign This = a && b;
  assign that = this;

  will have an undefined output.

- Use blocking assignments for combinatorial logic.

- Ensure ‘combinatorial’ outputs are assigned under all conditions. E.g. include ‘default’ by habit in all ‘case’ statements.

- Use non-blocking assignments for latching circuits.

- Make variable sizes match. Assigning a variable of one size to one of another is syntactically legal.

Verilog is case sensitive and ‘wire’ variables do not have to be declared. Thus:

```verilog
wire this, that;
assign This = a && b;
assign that = this;
```

will have an undefined output.

Use blocking assignments for combinatorial logic.

Ensure ‘combinatorial’ outputs are assigned under all conditions. E.g. include ‘default’ by habit in all ‘case’ statements.

Use non-blocking assignments for latching circuits.

Make variable sizes match. Assigning a variable of one size to one of another is syntactically legal.
Structural Verilog

Verilog can also be used to specify hierarchical designs.

Example

\[
\text{and2 gate}_1 (.a(x), .b(y), .q(z));
\]

where

- `and2` is the part (module) name
- `gate_1` is the instance name
- \{a, b, q\} are the inputs/output of the module
- \{x, y, z\} are wires at the higher level

- Structural Verilog may be written by hand – but it is often easier to draw schematics
  - It is sometimes useful for including an existing module in a behavioural description
- It is often machine-written as a netlist format
  - e.g. the output from synthesizing behavioural code may be in this form
  - As text it is more portable than schematics.

- Structural Verilog allows the explicit inclusion of certain cells within a design. The example below shows a simple module with a schematic equivalent.

```
module demo (input wire clk, input wire [7:0] a, input wire b, input wire c, output wire p, output wire [15:0] q);
wire d, e, f;
wire [7:0] w;
assign d = b || c; // Synthesizable statement
block_1 I1(.clk(clk), .a(a), .b(d), .i(e), .j(w));
block_2 I2(.clk(clk), .a(w), .k(f), .l(q));
and2 I3 (.a(e), .b(f), .q(p));
endmodule
```

Schematic

This example is contrived to show some features:

- I/O wires and buses connected directly
- Internal wire and bus declarations
- Naming of instances
- Mixture of structural and synthesized code allowed
  - Not necessarily Good Practice though!

Syntax

Assume we have a module declaration:

```
module and2 (input a, input b, output q);
```

The Verilog 2001 syntax

```
and2 gate_1 (.a(x), .b(y), .q(z));
```

explicitly connects wires/buses to ports on the module by associating their names. The order in which these are written is irrelevant. Ports may be omitted (although this is only sensible for outputs; inputs should always be defined).

Verilog 1995 syntax relies on ordering:

```
and2 gate_1 (x, y, z);
```

Backwards compatibility means that Verilog 2001 will accept either.

It is suggested that the later syntax is preferred because it is more robust against later edits, e.g. if another port is added to a module.
Synthesis: (possible) examples

```verilog
assign y = a & b | c;
assign q = s ? i1 : i0;
```

```verilog
always @(posedge clk, posedge rst)
if (rst) Q <= 0;
else Q <= D;
```

### Synthesis

Verilog is a Hardware Description Language but, if a subset of its features are used, the description can be turned into an implementation.

Just as a compiler takes a high-level language description of an algorithm and turns it into machine instructions for a chosen instruction set, a Verilog synthesizer turns a hardware description into an underlying technology.

The technology will vary according to what is on offer on the target process. **Cell libraries** are usually available from the **silicon foundry**.

A typical cell library will contain familiar gates such as AND2, AND3, OR2, … [and more - but more on that later].

During synthesis structures are identified in the source – and sometimes this fails. The bottom example on the slide shows a D-type register with an asynchronous clear. Reassure yourself that you follow the behaviour of this code.

Here’s a genuine example we found with one synthesizer:

```verilog
always @(posedge clk, posedge rst)
if (rst || clr) Q <= 0; // Failed to synthesize
else Q <= D;
```

Behaviourally these blocks are the same. They should simulate in the same way. However the author needed to separate the asynchronous input explicitly for synthesis.

### FPGA …

When synthesizing for an FPGA the tool will look for structures it can recognise and try to map these into the available resources.

Examples:

- A RAM – if specified appropriately – may be use dedicated RAM blocks.
- A statement such as “`A = B + C`” will use adjacent LUTs for its various bits to exploit fast carry logic.
- An “initial” statement may be applied to a registered signal. All flip-flops are zeroed when the FPGA’s are configured. “`initial x = 1;`” can be implemented by inverting the value around the actual flip-flop.
- An FPGA with multiplier blocks may be able to exploit these.

```
RAM
```

Logic functions will then be amalgamated to fit into LUTs of the appropriate size.

```
RAM
```

If a statement cannot be mapped into an available structure then the design may not fit the FPGA.

On the right are two RAM read structures. The upper one does not map to Xilinx block RAMs and therefore the RAM will be expanded into (a LOT of) combinatorial logic.

The lower structure looks bigger but exploits the available resources and thus is (much) more efficient.
States and truth tables

Verilog represents circuits digitally.

- There are four basic states which a digital wire can be in: \(\{0, 1, Z, X\}\)

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XOR</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- Switching between states is instantaneous
- There is no concept of ‘halfway’ or of ‘edge speed’
- ‘X’ (unknown) usually means either 0 or 1 but the simulator doesn’t know which
  - This is the default initial state of flip-flops, latches etc.
  - It may occur if two different outputs are connected together (mistake!)
- ‘Z’ (high-impedance, “Tristate™”) means there is no output driving the network
  - No longer relevant for on-chip signals
  - Default state for input wires

Tasks/Functions

A **task** is a procedure/method/subroutine/… structure in Verilog. They are local to modules (but could be ‘included’). Tasks may substitute for repetitive operations and may be useful in simplifying and clarifying a test bench. They may contain combinatorial or sequential logic and occupy (simulation) time in execution. [Syntax may vary; an alternative example is given in the lab. manual.]

```verilog
task something;
    input ...
    output ...
    begin
        <behavioural code>
    end
endtask
```

**Functions** are like simple tasks returning a value. They are typically used to represent some combinatorial function. For example:

```verilog
function adder; // Function name is also result
    input x, y;
    begin
        adder = x + y; // Hardly worthwhile!
    end
endfunction
```

Compiler directives

Verilog compiler directives are keywords preceded by the backtick character ‘`

Some of the more common ones are:

- `timescale` Set the timestep in simulation, i.e. how long #1 really represents.

- `include` Import another file into the source; useful for including common definitions amongst several files in a project.

- `define` Verilog allows test substitution in a similar manner to C’s “#define”.

```verilog
'define THING 99 // Define test substitution
x <= 'THING; // In use (note ‘)
```

- `undef` Remove a compile-time definition.

```verilog
'undef
```

- `ifdef, 'else, 'elsif, 'endif, 'ifndef` Used to control conditional compilation by “bracketing” parts of the code. The application should be obvious.
Generate

module big_memory (input wire CE, input wire WE, input wire OE, input wire A[11:0], input wire [7:0] Din, output wire [7:0] Dout);
reg [3:0] En;
wire [7:0] Data [0:3] // Array of four data buses
always @ (CE, A[11:10]) // Address decoder with enable
begin
En = 4’b0000;
if (CE) En[A[11:10]] = 1’b1;
end
generate
genvar i; // Variable used for compile-time iteration
for (i=0; i<4; i=i+1)
begin: block
memory mem((.CE(En[i]), .WE(WE), .OE(OE), .A(A[9:0]), .Din(Din), .Dout(Data[i])));
end
dengenerate
assign Dout = Data[A[11:10]]; // Output multiplexer
end

Generate

Generate allows multiple instances to be placed iteratively. The example in the slide shows how a 4 KB memory could be constructed from four, 1 KB blocks.

- The always block provides a 2:4 decoder from the upper two address lines.
- A 2-bit field taken from the address is used to index an En bit, which is set to '1'; all the (other) bits have been zeroed.
- The iterated instantiation is delimited by {generate, endgenerate}
- genvar declares a control variable; this only exists at 'compile time'
- The for loop iterates using the control variable
- Instances of memory called 'mem[0:3]' are produced
- Wiring is in common except for the different in chip selects (En[0:3])
- As an illustration, the output data buses are multiplexed explicitly.

This example instantiation iterates a structural Verilog block: it is possible to contain a variety of statements within the block.

The control for the generate needs to be determined when the blocks are to be instantiated. Hardware cannot be created dynamically 'at run time'. Obviously!

The real value of such constructs is apparent when the number of instantiations is parameterised. For example:

```verilog
always @ (CE, A[9+'BITS:10])
begin
En = 0;
if (CE) En[A[9+'BITS:10]] = 1’b1;
end
generate
genvar i;
for (i=0; i<(1<<('BITS-1)); i=i+1)
...
endgenerate
```

This allows the iterations to be specified with a single input definition.

Here 'BITS specifies the log₂ of the number of blocks and thus 1<<'BITS regenerates this number. It should be constrained to 1 because "A[10:10]" is legal but "A[9:10]" will cause problems.

The reverse approach is more difficult as taking a log is more tedious. However here is an appropriate function from the Verilog Standard document.

```verilog
function integer clogb2;
input [31:0] value;
for (clogb2=0; value>0; clogb2=clogb2+1;)
value = value>>1;
endfunction
```

If the log is non-integral, this finds the next largest integer, which is generally what is wanted.
Parameters

module adder (a, b, s);
parameter n = 16;
input wire [n-1:0] a, b;
output wire [n-1:0] s;
assign s = a + b;
endmodule

module example;
wire [15:0] pp, qq, rr;
wire [31:0] ss, tt, uu;
wire [7:0] vv, ww, xx;
adder add16(pp, qq, rr);
adder add32 #(32) (ss, tt, uu); // Verilog 95
adder add8 #(.n(8)) (.a(vv), .b(ww), .c(xx)); // Verilog 2001
endmodule

Parameters

There are several ways to define and modify parameters in Verilog. Only a selection is elucidated here but there should be enough to illustrate the concept.

The example on the slide specifies a (somewhat redundant) module and instantiates it three times. The module has a single parameter (n) which, in this case, specifies its bus widths. There is a ‘default’ value of 16 assigned to this parameter.

The first instantiation simply uses this default value. The other instantiations override the default. For clarity, in the first instantiation it is suggested that a ‘16’ should be passed anyway.

The add32 uses the older Verilog-95 syntax both the parameter list and the connections. This syntax works much like many programming languages in that associations are determined by the order of the lists. This syntax is also useable in later versions of Verilog.

add8 is instantiated using Verilog 2001 syntax. This specifies each parameter/connection explicitly.

..<name_inside_instance>(<local_name>)

In this syntax the elements can be specified in any order.

It is recommended that the Verilog 2001 syntax is used. It is more verbose but less prone to errors, especially if lists are edited after creation. In the case of parameters it also allows an arbitrary set of overrides to be used, not just those from the start of the list.

defparam

A parameter can be passed explicitly to a module using `defparam`.

The following could be added to the example above:

    defparam add16.n = 16;

where a value is assigned using its hierarchical name.
Summary

- Verilog is a Hardware Description Language
  - Quite good for modelling
  - Highly parallel
  - A subset can be used as source for RTL synthesis
- Verilog has evolved
  - Not always syntactically ‘clean’
  - Multiple ways to write the same thing
- There are ‘traps’ for the unwary
  - Synthesizers look for structures to implement
  - Simulated and synthesized functions may not match (if language ‘abused’)
  - Helpful to keep implementation technology in mind.
  - Care needed not to produce redundant hardware (e.g. latches)
- Useful … with a degree of care