Imperial College London

Lecture 3

Verilog HDL – Part 1

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/ee2_digital/
E-mail: p.cheung@imperial.ac.uk

This and the next lectures are about Verilog HDL, which, together with another language VHDL, are the most popular hardware languages used in industry.

Verilog is only a tool; this course is about digital electronics. Therefore, I will NOT be going through Verilog as in a programming course - it would have been extremely boring for both you and me if I did. Instead, you will learn about Verilog through examples. I will then point out various language features along the way. What it means is that the treatment of Verilog is NOT going to be systematic – there will be lots of features you won't know about Verilog. However, you will learn enough to specify and design reasonably sophisticated digital circuits, and you should gain enough confidence to learn the rest by yourself.

There are many useful online resources available on details of Verilog syntax etc.. Look it up as you need to and you will learn how to design digital circuit using Verilog through designing real circuits.

The problem sheets are mostly about circuits and concepts, with occasional Verilog exercises. You will be doing lots of Verilog coding during the four weeks of Lab Experiment in the second half of the term.

Lecture Objectives

- By the end of this lecture, you should understand:
  - Why a Hardware Description Language (HDL) is a better design entry method than schematic entry?
  - The basic structure of a module specified in Verilog HDL
  - Commonly used syntax of Verilog HDL
  - Continuous vs Procedural Assignments
  - **always** block in Verilog
  - Sensitivity list
  - Nets vs variables

Here is a list of lecture objectives. They are provided for you to reflect on what you are supposed to learn, rather than an introduction to this lecture.

I want, by the end of this lecture, to give you some idea about the basic **structure** and **syntax** of Verilog. I want to convince you that schematic capture is NOT a good way to design digital circuits. Finally, I want you to appreciate how to use Verilog to specify a piece of hardware at **different levels of abstraction**.

Schematic vs HDL

Schematic	HDL
✓ Good for multiple data flow	✓ Flexible & parameterisable
✓ Give overview picture	✓ Excellent input to optimisation & synthesis
✓ Relate directly to hardware	✓ Direct mapping to algorithms
✓ Don't need good programming skills	✓ Excellent for datapaths
✓ High information density	✓ Easy to handle electronically (only needing a text editor)
✓ Easy back annotations	
✓ Useful for mixed analogue/digital	
× Not good for algorithms	× Serial representation
× Not good for datapaths	× May not show overall picture
× Poor interface to optimiser	× Need good programming skills
× Poor interface to synthesis software	× Divorce from physical hardware
× Difficult to reuse	
× Difficult to parameterise	

PYKC 14 Oct 2019 E2.1 Digital Electronics Lecture 3 Slide 3

You are very familiar with schematic capture. However modern digital design methods in general DO NOT use schematics. Instead an engineer would specify the design requirement or the algorithm to be implemented in some form of computer language specially designed to describe hardware. These are called "Hardware Description Languages" (HDLs).

The most important advantages of HDL as a means of specifying your digital design are: 1) You can make the design take on parameters (such as number of bits in an adder); 2) it is much easier to use compilation and synthesis tools with a text file than with schematic; 3) it is very difficult to express an algorithm in diagram form, but it is very easy with a computer language; 4) you can use various datapath operators such as +, * etc.; 5) you can easily edit, store and transmit a text file, and much hardware with a schematic diagram.

For digital designs, schematic is NOT an option. Always use HDL. In this lecture, I will demonstrate to you why with an example.

Verilog HDL

- Similar to C language to describe/specify hardware
- Description can be at different levels:
  - **Behavioural level**
  - **Register-Transfer Level (RTL)**
  - **Gate Level**
- Not only a specification language, also with associated **simulation environment**
- Easier to learn and "lighter weight" than its competition: VHDL
- Very popular with chip designers

- For this lecture, we will:
  - ❑ Learn through examples and practical exercises
  - ❑ Use two examples: 2-to-1 multiplexer and 7 segment decoder

PYKC 14 Oct 2019 E2.1 Digital Electronics Lecture 3 Slide 4

I have chosen to use Verilog HDL as the hardware description language for this module. Verilog is very similar to the C language, which you should already know from last year. However, you must always remember that YOU ARE USING IT TO DESCRIBE HARDWARE AND NOT AS A COMPUTER PROGRAMME.

You can use Verilog to describe your digital hardware in three different level of abstraction:

1) Behavioural Level – you only describe how the hardware should behave without ANY reference to digital hardware.

2) Register-Transfer-Level (RTL) – Here the description assumes the existence of registers and these are clocked by a clock signal. Therefore digital data is transferred from one register to the next on successive clock cycles. Timing (in terms of clock cycles) is therefore explicitly defined in the Verilog code. This is the level of design we use most frequently in this course.

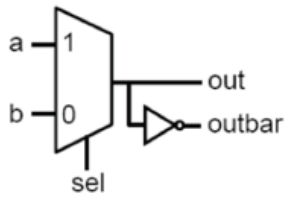
3) Gate Level – this is the lowest level description where each gate and its interconnection are explicitly specified.

Verilog is not only a specification language which tells the CAD system what hardware is suppose to do, it also includes a complete simulation environment. A Verilog compiler does more than mapping your code to hardware, it also can **simulate** (or execute) your design to predict the behaviour of your circuit. It is the predominant language used for chip design.

You will learn Verilog through examples and exercises, not through lecture. However, I will spend just two lectures to cover the basics of Verilog.

Structure of a Module

- Verilog design contains **interconnected modules**
- A module has collections of low-level gates, statements, and other modules
- Here is an example of a simple module that describes a 2-to-1 multiplexer:



sel	out
0	b
1	a

```
// Function: 2-to-1 multiplexer
module mux2to1 (out, outbar,
                a, b, sel);

    output  out, outbar;
    input   a, b, sel;

    assign  out = sel ? a : b;
    assign  outbar = ~out;

endmodule
```

- // to end-of-line is comment. Can also use /* … */ for multiline comments
- Declare and name module; list its ports; terminate with ';'
- Specify port as input, output (or inout if bidirectional)
- Express modules behaviour; each statement executes in parallel; ORDER DOES NOT MATTER

This is a Verilog module that specifies a 2-to-1 multiplexer. It is rather similar to a C function (except for the **module** keyword).

It is important to remember the basic structure of a Verilog module. There is a module name: **mux2to1**. There is a list of interface ports: 3 inputs **a**, **b** and **sel**, and 2 outputs **out** and **outbar**. Always use meaningful names for both module name and variable names.

You must specify which port is input and which port is output, similar to the data type declaration in a C programme.

Finally, the 2-to-1 multiplexing function is specified in the **assign** statement with a construct that is found in C. This is a **behavioural** description of the multiplexer – no gates are involved.

The last statement specifies the relationship between **`out`** and **`outbar`**. It is important to remember that Verilog describes HARDWARE not instruction code. The two **assign** statements specify hardware that "execute" or perform the two hardware functions in parallel. Therefore their **order does not matter**.

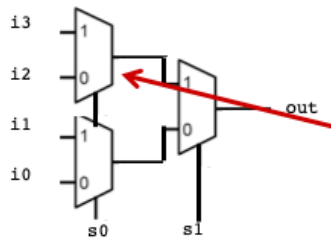
Continuous Assignment

```
module mux2to1 (out, outbar,
                a, b, sel);

   output  out, outbar;
   input   a, b, sel;

   assign  out = sel ? a : b;
   assign  outbar = ~out;

endmodule
```



- Keyword **assign** specifies **continuous assignment** to describe combinational logic
- Right-hand expression continuously evaluated …. responding to input changes immediately
- Left-hand is a net driven with evaluated value
- Left side must be a scalar, a net or a concatenation of nets and vector nets. (nets, vectors etc. will be described later)
- All continuous assignments execute in parallel
- Operators in expressions are low-level:
  - Conditional assignment: (cond)**?** vTrue**:** vFalse
  - Boolean: ~, &, |,
  - Arithmetic: +, -, *
- Operators can be nested. For example, here is a 4-to-1 mux:

```
assign   out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);
```


Continuous assignment specifies combinational circuits – output is continuously reflecting the operations applied to the input, just like hardware.

Remember that unlike a programming language, the two continuous assignment statements here ARE specifying hardware in PARALLEL, not in series.

Here we also see the conditional statement that is found in C. This maps perfectly to the function of a 2-to-1 multiplexer in hardware and is widely used in Verilog.

Furthermore, there are many other Boolean and arithmetic operators defined in Verilog (as in C). Here is a quick summary of all the Verilog operators (used in an expression).

(What is "reduction and &"? I want you to find this yourself online.)

{}, {{}}	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
~	bit-wise negation
&	bit-wise and
|	bit-wise inclusive or
^	bit-wise exclusive or
^~ or ~^	bit-wise equivalence

&	reduction and
~&	reduction nand
|	reduction or
~|	reduction nor
^	reduction xor
~^ or ^~	reduction xnor
<<	left shift
>>	right shift
? :	condition
or	event or

Description at Gate Level

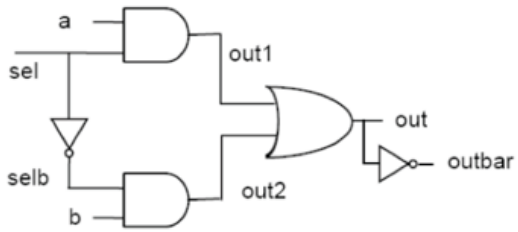
```
// Function: 2-to-1 multiplexer as gates

module mux_gate (out, outbar,
                 a, b, sel);

    output  out, outbar;
    input   a, b, sel;
    wire    out1, out2, selb;

    not i1 (selb, sel);
    and a1 (out1, a, sel);
    and a2 (out2, b, selb);
    or o1 (out, out1, out2);
    not i2 (outbar, out);

endmodule
```



- Built-in logic gate primitives:
  - and, nand, or, nor, xor, xnor, not, buf
- Can use arbitrary number of inputs, e.g. and gate_name (out, in1, in2, in3, …)
- Tri-state buffers: bufif1 and bufif0
- Connect gates with nets using declaration keyword wire
- **and a1 (out1, a, sel);**
- This statement is an **instantiation** of an AND gate. **a1** is the name of THIS particular AND gate.

While the previous Verilog code for the 2-to-1 mux only specifies "**behaviour**", here is one that specifies a gate implementation of the same circuit. Three types of gates are used: **and**, **or** and **not** gates. There are internal nets (declared as **wire**) which must also be declared and are used to connect gates together.

Keywords such as **and**, **or** and **xor** are special – they specify actual logic gates. They are also special in that the number of inputs to the and-gate can be 2, 3, 4, ….. Any length!

Note that this module uses TWO AND gates, and they have different names: a1 and a2. These are TWO **separate instances** of the AND gate. In software, "calling" a function simple execute the same piece of programme code. Here the two statements "**and a1 (out1,** …" and "**and a2 (out2 …**" produce two separate piece of hardware. We say that each line is "**instantiating**" an AND gate.

Wiring up the gates is through the use of ports and wires, and depends on the positions of these "nets". For example, **out1** is the output net of the AND gate **a1**, and it is connected to the input of the **OR** gate **it**by virtual of its location in the gate port list.

Procedural Assignment using "always"

- Keyword **always** and **initial** are used to define **procedural assignment**, similar to procedures in software
- Good for behavioural description of hardware including combinational and sequential logic
- Support richer C-like constructs such as **if, for, while, case** etc.

```
module mux_2_to_1 (out, outbar, a, b, sel);
    output  out, outbar;
    input   a, b, sel;

    reg out, outbar;

    always @ (a or b or sel)

    begin
        if (sel) out = a;
        else  out = b;

        outbar = ~out;
    end
endmodule
```

always @ *

- Declarations same as before
- Assignment inside an always block must be declared as variable data type such as **reg** (see later)
- **always** block runs once whenever a signal in the **sensitivity list** changes value
- Statements inside the always block are **executed sequentially**. ORDER MATTERS!
- Sandwiched between begin/end

PYKC 14 Oct 2019 E2.1 Digital Electronics Lecture 3 Slide 8

So far we have used Verilog in very hardware specific way. "assign" and using gate specification are special to Verilog. You do not find these in C.

Here is something that is more like C – and it is called "**procedural assignment**". Typically we use something called "**always**" block to specify a "**procedure**", i.e. a collection of sequential statements which are sandwiched between the **begin-end** construct.

The **always** block needs a **sensitivity list** – a list of signals which, if ANY of these signals changes, the **always** block will be invoked. You may read this block as:

"always at any changes in nets **a**, **b** or **sel**, do the bits between **begin** and **end**".

Actually, if you are defining a combinational circuit module, an even better way to define the always block is to use:

..... always @ * .... // always at any change with any input signal

Inside the **begin-end** block, you are allowed to use C-like statements. In this case, we use the **if-else** statement. All statements inside the **begin-end** block are **executed sequentially**.

Verilog "register" is NOT what it appears!

- Registers normally represents storage elements in digital logic, they need clock signals to update their output value
- In Verilog **reg** is NOT the same as a digital register, it is used only to declare a variable that holds a value
- Values of variables (declared as **reg**) can be changed anytime in a simulation, and can be used for nets of a combinational circuit
- In other words, in Verilog, **reg** is similar to declarations such as **int, real** etc. When you use:

```
reg    signal_a;
```

whether a physical register (or D-FF) is sythesized to store **signal_a** or not would depend on the rest of the Verilog code.

Note that Verilog keyword **reg** does not implies that there is a register created in the hardware. It is much more like declaring a variable that holds a value. It is a rule in Verilog that if you perform an **assignment** to a variable **INSIDE** an **always** block, that variable **MUST be declared reg**, and NOT a net (**wire**). This is one of the few peculiarities of Verilog that can be confusing to students.

Mixing procedural & continuous assignments

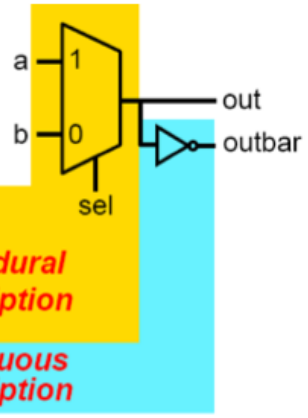
- Procedural and continuous assignments can co-exist within a module
- In procedural assignments, the value of variables declared as **reg** are changed only once when the procedural block is invoked by changes in the sensitivity list
- In continuous assignments, the right-hand expression is constantly evaluated and the left-side net is updated all the time

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```



This slide shows how the procedural statement is mapped to the basic MUX circuit. The continuous assignment statement corresponds to the NOT gate.

case statement – better alternative to if-else

- **case** statement can often replace **if-else** construct within an **always** block, and provides better abstraction
- Here is an example using the mux_2_to_1 module:

```
always @ (a or b or sel)
begin
    case (sel)
      1'b0: out = b;
      1'b1: out = a;
    endcase
end
```

```
if (sel) out = a;
else  out = b;
```

Notation for numbers:
<size> ‘ <base> <number>

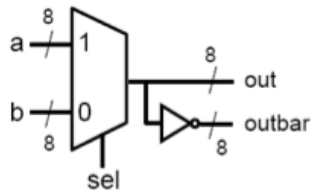
2’b10	2 bit binary, v=2
‘b10	Unsized binary 32-bit, v=2
31	Unsized decimal, v=31
8’hAf	8-bit hex, v=175
-16’d47	16-bit negative decimal, v=-47

This is yet another way to specify the MUX circuit. It is still a procedural assignment with the **always** block. However, we replace the **if-else** statement with a “**case**” statement. The case variable is **sel**. Since **sel** is a 1-bit signal (or net), it can only take on 0 or 1.

Note that the various case values can be expressed in different number formats as shown in the slide. For example, consider **2’b10**. The 2 is the number of bits in this number. **‘b** means it is specified in binary format. The value of this number is **10** in binary.

n-bit signals - buses

- Verilog is powerful in specifying multi-bit signals and buses.
- Here is an example for 8-bit wide 2-to-1 multiplexer:




```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input[7:0] a, b;
  input sel;
  output[7:0] out, outbar;
  reg[7:0] out;
  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end
  assign outbar = ~out;
endmodule
```

Concatenate signals using the { } operator

```
assign {b[7:0],b[15:8]} = {a[15:8],a[7:0]};
```

effects a byte swap

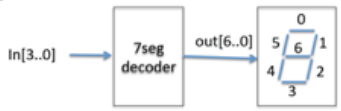
This slide demonstrates why language specification of hardware is so much better than schematic diagram. By simply declaring the signals as a multi-bit bus (8 bits [7:0]), we change this module to one that specifies 8 separate 2-to-1 multiplexers.

Another useful way to specify a bus is using the concatenation operator: { .... } as shown above.

The concatenation operator is particularly useful in converting digital signals from one word length (i.e. number of bits in a word) to another. For example, to convert an 8-bit unsigned number a[7:0] to a 13-bit unsigned number b[12:0], you can simple do this:

```
assign   b[12:0] = {5'b0, a[7:0]};
```

Putting everything together – 7 seg decoder



In3 : in2

In1 : in0 \ out6	00	01	11	10
00	1	0	1	0
01	1	0	0	0
11	0	1	0	0
10	0	0	0	0

in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7

in[3..0]	out[6:0]	Digit
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

out5 = /in3*/in2*in0 + /in3*/in2*in1 + /in3*in1*in0 + in3*in2*/in1*in0

out4 = /in3*in0 + /in3*in2*/in1 + in3*/in2*/in1*in0

out3 = /in3*in2*/in1*/in0 + /in3*/in2*/in1*in0 + in2*in1*in0 + /in2*in1*/in0

out2 = /in3*/in2*in1*/in0 + in3*in2*/in0 + in3*in2*in1

out1 = in3*in2*/in0 + /in3*in2*/in1*in0 + in3*in1*in0 + in2*in1*/in0

out0 = /in3*/in2*/in1*in0 + /in3*in2*/in1*/in0 + in3*in2*/in1*in0 + in3*/in2*in1*in0

Here is a simple example: the design of a 4-bit hex code to 7 segment decoder. You can express the function of this 7-segment decoder in three forms: 1) as a truth table (note that the segments are low active); 2) as 7 separate K-maps (shown here is for **out[6]** segment only); 3) as Boolean equations.

This is probably the last time you see K-maps. In practical digital design, you would rely heavily on CAD tools. In which case, logic simplifications are done for you automatically – you never need to use K-maps to do Boolean simplification manually!

Method 1: Schematic Entry Implementation

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

out5 = /in3*/in2*in0 + /in3*/in2*in1 + /in3*in1*in0 + in3*in2*/in1*in0

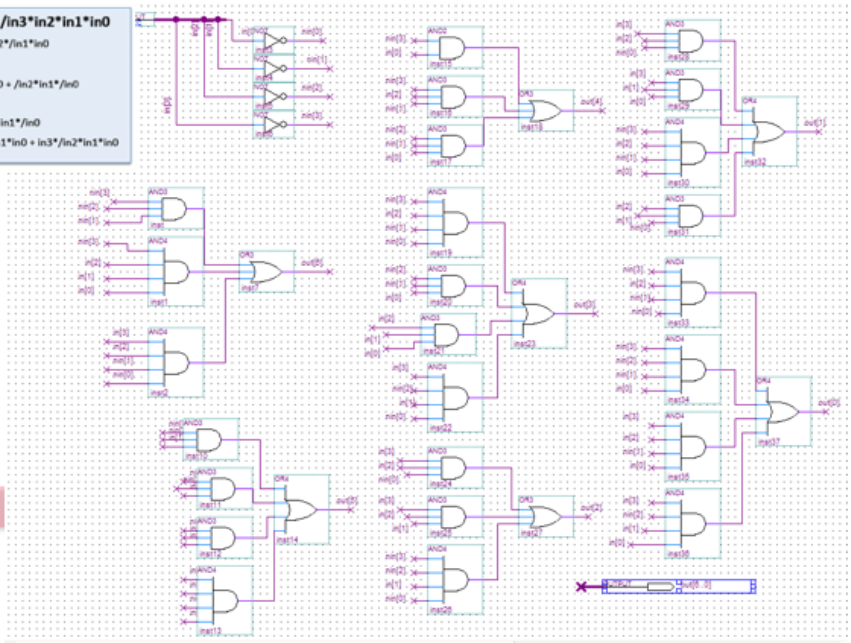
out4 = /in3*in0 + /in3*in2*/in1 + in3*/in2*/in1*in0

out3 = /in3*in2*/in1*/in0 + /in3*/in2*/in1*in0 + in2*in1*in0 + /in2*in1*/in0

out2 = /in3*/in2*in1*/in0 + in3*in2*/in0 + in3*in2*in1

out1 = in3*in2*/in0 + /in3*in2*/in1*in0 + in3*in1*in0 + in2*in1*/in0

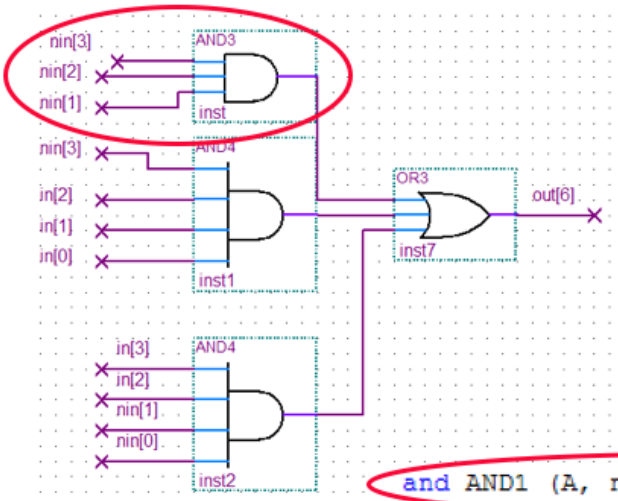
out0 = /in3*/in2*/in1*in0 + /in3*in2*/in1*/in0 + in3*in2*/in1*in0 + in3*/in2*in1*in0



TEDIOUS!!!!!

Here is a tedious implementation in the form of schematic diagram of the 7 segment decoder as interconnected gates. Very hard to do and very prone to errors.

Method 2: Use primitive gates in Verilog



Equally TEDIOUS!!!!!

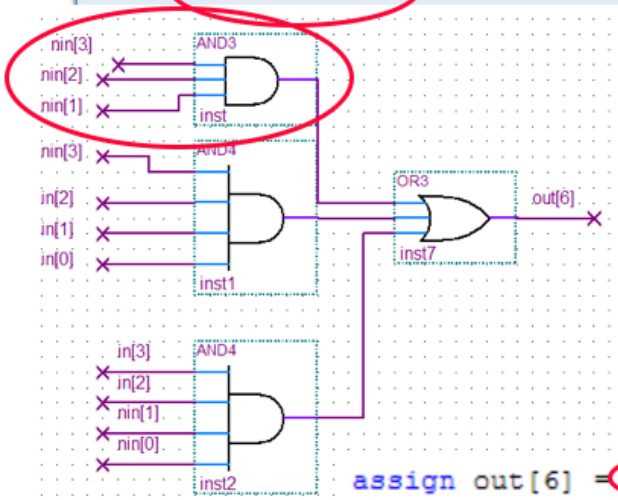
Direct mapping of gates to primitives

```
and AND1 (A, nin[3], nin[2], nin[1]);
and AND2 (B, nin[3], in[2], in[1], in[0]);
and AND3 (C, in[3], in[2], nin[1], nin[0]);
or  OR1  (out[6], A, B, C);
```


One could take a group of gates and specify the gates in Verilog gate primitives such as **and**, **or** etc. Still very tedious. Here is the implementation for the **out[6]** output.

Method 3: Use continuous assignment in Verilog

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0



Much Better?

Direct mapping of Boolean equation using continuous assignment..

```
assign out[6] = ~in[3]&~in[2]&~in[1] |
                in[3]&in[2]&~in[1]&~in[0] |
                ~in[3]&in[2]&in[1]&in[0];
```

PYKC 14 Oct 2019 E2.1 Digital Electronics Lecture 3 Slide 16

Instead of specifying each gate separately, here is using continuous assignment statement, mapping the **Boolean equation** direction to a single Verilog statement. This is better.

module & **endmodule** sandwich the content of this hardware module

Hex_to_7seg.v

```
//-----------------------------
// Module name: hex_to_7seg
// Function: convert 4-bit hex value to drive 7 segment display
//           output is low active
// Creator:  Peter Cheung
// Version:  1.0
// Date:     22 Oct 2011
//-----------------------------

module hex_to_7seg  (out,in);

    output  [6:0]   out;    // low-active output to drive 7 segment display
    input   [3:0]   in;     // 4-bit binary input of a hexademical number

    assign out[6] = ~in[3]&~in[2]&~in[1] | in[3]&in[2]&~in[1]&~in[0] |
                    ~in[3]&in[2]&in[1]&in[0];
    assign out[5] = ~in[3]&~in[2]&in[0] | ~in[3]&~in[2]&in[1] |
                    ~in[3]&in[1]&in[0] | in[3]&in[2]&~in[1]&in[0];
    assign out[4] = ~in[3]&in[0] | ~in[3]&in[2]&~in[1] | in[3]&~in[2]&~in[1]&in[0];
    assign out[3] = ~in[3]&in[2]&~in[1]&~in[0] | ~in[3]&~in[2]&~in[1]&in[0] |
                    in[2]&in[1]&in[0] | ~in[2]&in[1]&~in[0];
    assign out[2] = ~in[3]&~in[2]&in[1]&~in[0] | in[3]&in[2]&~in[0] |
                    in[3]&in[2]&in[1];
    assign out[1] = in[3]&in[2]&~in[0] | ~in[3]&in[2]&~in[1]&in[0] |
                    in[3]&in[1]&in[0] | in[2]&in[1]&~in[0];
    assign out[0] = ~in[3]&~in[2]&~in[1]&in[0] | ~in[3]&in[2]&~in[1]&~in[0] |
                    in[3]&in[2]&~in[1]&in[0] | in[3]&~in[2]&in[1]&in[0];
endmodule
```

good header helps documenting your code

specify interface to this module as viewed from outside

specify a 7-bit output bus, out[6] … out[0]

declaration of input and output ports

assign used to specify combinational circuit

PYKC 14 Oct 2019 | E2.1 Digital Electronics | Lecture 3 Slide 17

Here is the complete specification of the hex_to_7seg module using **continuous assignment** statements. It shows how one should write Verilog code with good comments and clear documentation of input and output ports.

Method 4: Power of behavioural abstraction

```
module hex_to_7seg  (out,in);

    output  [6:0]   out;    // low-active output to
    input   [3:0]   in;     // 4-bit binary input o

    reg     [6:0]   out;    // make out a variable

    always @ (in)
      case (in)
        4'h0: out = 7'b1000000;
        4'h1: out = 7'b1111001;     // -- 0 ---
        4'h2: out = 7'b0100100;     // |      |
        4'h3: out = 7'b0110000;     // 5      1
        4'h4: out = 7'b0011001;     // |      |
        4'h5: out = 7'b0010010;     // -- 6 ---
        4'h6: out = 7'b0000010;     // |      |
        4'h7: out = 7'b1111000;     // 4      2
        4'h8: out = 7'b0000000;     // |      |
        4'h9: out = 7'b0011000;     // -- 3 ---
        4'ha: out = 7'b0001000;
        4'hb: out = 7'b0000011;
        4'hc: out = 7'b1000110;
        4'hd: out = 7'b0100001;
        4'he: out = 7'b0000110;
        4'hf: out = 7'b0001110;
      endcase
endmodule
```

BEAUTIFUL !!!

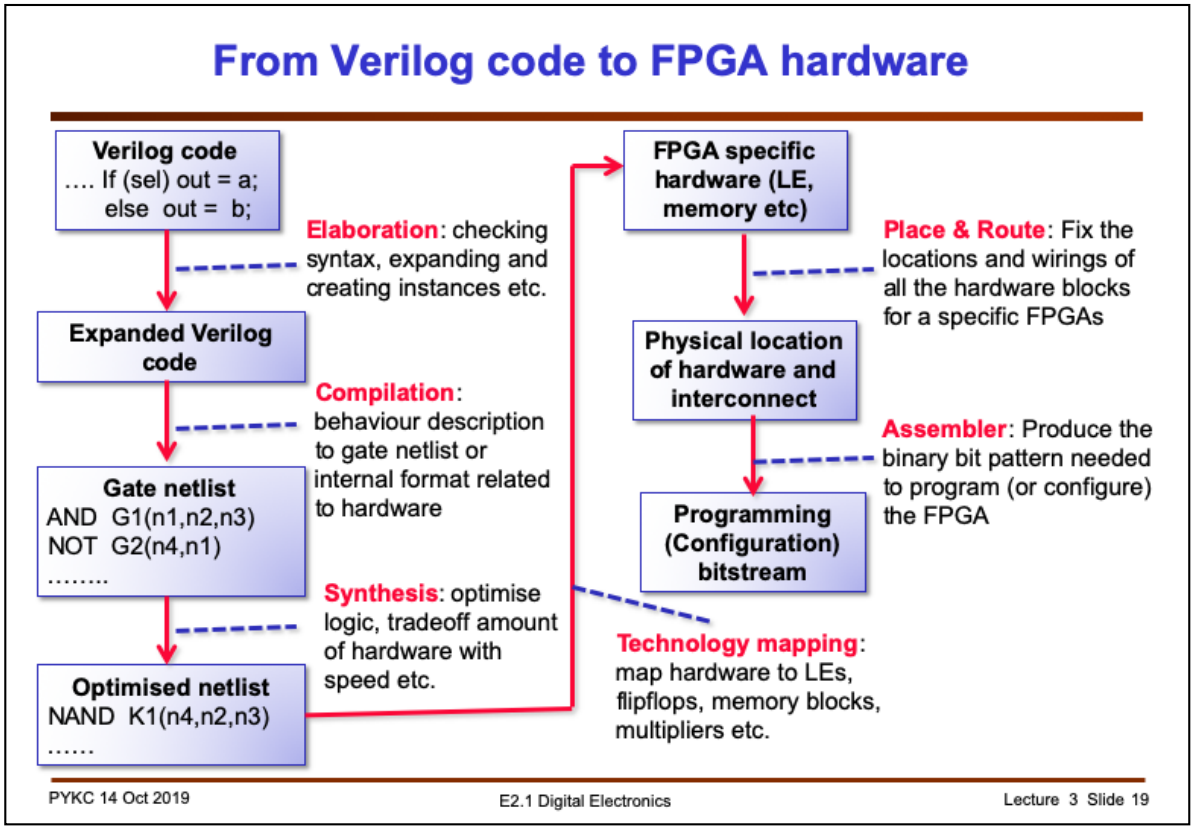
- Direct mapping of truth table to case statement
- Close to specification, not implementation

in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

PYKC 14 Oct 2019 E2.1 Digital Electronics Lecture 3 Slide 18

Finally the 4th method is the best. We use the case construct to specify the behaviour of the decoder. Here one directly maps the truth table to **the case statement** – easy and elegant.

Instead of using: always @ (in), you could also use always @*



How is a Verilog description of a hardware module turned into FPGA configuration? This flow diagram shows the various steps taken inside the Quartus Prime CAD system.

Quiz

1. What are inside a Verilog module?
2. Where does one specify the interface to a module?
3. What are port declarations?
4. When a variable is declared as type *reg*, must there always be some registers (flipflops) in the circuit?
5. What is the meaning of this statement?

   x = (c) ? b = a+1 : b = a – 1;
6. What is a continuous assignment?
7. How does a continuous assignment differ from a procedural assignment?
8. What is the value of a number written as: 9'hA3D? How many bits are used to present this number?
9. When are case statements used?
10. Why is behavioural description generally better than gate level description?

Answers are all in the notes.