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 reasonably sophisticated digital circuits, and you should gain enough confidence to learn the rest by yourself.

There are many useful online resources available on the 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.
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**.
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.
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.
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

- 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:

```
module mux2to1 (out, outbar, a, b, sel);
output out, outbar;
input a, b, sel;
assign out = sel ? a : b;
assign outbar = ~out;
endmodule
```

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
```
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.
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:

```verilog
..... 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 represent 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 synthesized to store signal_a or not would depend on the rest of the Verilog code.

Note that Verilog keyword reg does not imply 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.
This slide shows how the procedural statement is mapped to the basic MUX circuit. The continuous assignment statement corresponds to the NOT gate.
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.
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:

\[
\text{assign } b[12:0] = \{5'b0, a[7:0]\};
\]
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!
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.
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.

```verilog
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);
```
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.
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.
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?
   \[ x = (c +1)? b = a+1 : b = a - 1; \]
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.