In this lecture, we will go beyond the basic Verilog syntax and examine how flipflops and other clocked circuits are specified.

I will also introduce the idea of a "testbench" as part of a design specification.
Verilog is very much like C. However, the declaration of a, b and sum in the module add32 specifies the data width (i.e. number of bits in each signal a, b or sum). This is often known as a “vector” or a “bus”. Here the data width is 32-bit, and it is ranging from bit 31 down to bit 0 (e.g. sum[31:0]).

You can refer to individual bits using the index value. For example, the least-significant bit (LSB) of sum is sum[0] and the most-significant bit (MSB) is sum[31]. sum[7:0] refers the the least-significant byte of sum.

The ‘+’ operator can be used for signals of any width. Here a 32-bit add operation is specified. sum is also 32-bit in width. However, if a and b are 32-bit wide, the sum result could be 33-bit (including the carry out). Therefore this operation could result in a wrong answer due to overflow into the carry bit. The 33th bit is truncated.

The second example module add32_carry shows the same adder but with carry input and carry output. Note the LHS of the assign statement. The (cout, sum) is a concatenation operator – the contents inside the brackets () are concatenated together, with cout is assigned the MSB of the 33th bit of the result, and the remaining bits are formed by sum[31:0].

There are three different types of Boolean operators:

- **Bitwise operators** perform what you would expect as if there are parallel gates used for each bit of the operands. Therefore a&b means that each bit from a and b is passed through an AND-gate.

- **Logical operators** only result in 0 or 1 (i.e. 1-bit result) In this example la (not a) where a = 0101, will result in first, a being evaluated as a logical value (i.e. logical ‘1’ or true). Therefore the result “a is logical 0 (or false).

- **Reduction operators** is applied to a single operand (and sometimes known as unary operators). It performs the operation one-bit at a time to the operand.
Assume that we want to specify a 3-to-1 multiplexer as shown on the left. On the right is an attempt to specify this using the always + case construct in Verilog.
The case variable ‘sel’ is 2-bit wide, and therefore has 4 possibilities. The case statement only specifies three of the four possible cases.
This is known as an “incomplete specification”.

In Verilog, there is this rule:
If something is not completely specified, the output must retain its previous value when the unspecified condition occurs.

The consequence of this is an unexpected extra latch being added to the hardware.
In order to cope with the unspecified condition of sel = 2'b11, the output of the MUX is fed to be latch.
Noted that a latch is level-triggered; a flipflop is edge-triggered. A latch has the property that when the gate input G is high, Q = D (i.e. it is transparent: input goes straight to output). If G is low, the latch become opaque, meaning that it retains the previous value.
The green shaded latch in the diagram and the controlling NAND gate are the unintended consequences of this incompletely specified 3-to-1 multiplexer.
Always avoid incomplete specification

- Solution 1: Precede all conditionals with a default assignment for all signals:

```verilog
always @(a or b or c or sel) begin
  out = 1'b0;
  case (sel)
    2'b00: out = a;
    2'b01: out = b;
    2'b10: out = c;
    default: out = 1'b0;
  endcase
endmodule
```

- Solution 2: Fully specify all branches of if-else construct, or include a default statement in case construct:

```verilog
always @(a or b or c or sel) begin
  case (sel)
    2'b00: out = a;
    2'b01: out = b;
    2'b10: out = c;
    default: out = 1'b0;
  endcase
endmodule
```

There are two solutions to avoid the unintended latch being added.

**Solution 1** is to put outside the case statement a "default" value for out. Here $1'b0$ (i.e. `.'0`) means **undefined**.

**Solution 2** is better: inside the case statement block, always add the default line. This will catch ALL the unspecified cases and avoid the introduction of the spurious unintended latches.

Lesson: always include a default assignment in any case statement to capture unintended incomplete specification.

How to specify a sequential circuit?

- Edge-triggered flipflop is specified with: `always @ (posedge clk):

```verilog
module combinational(a, b, sel, out);
  input a, b;
  input sel;
  output out;
  reg out;
  always @(a or b or sel) begin
    if (sel) out = a;
    else out = b;
  end
endmodule
```

- Sequential circuit using always block:

```verilog
module sequential(a, b, sel, clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;
  always @(posedge clk) begin
    if (sel) out <= a;
    else out <= b;
  end
endmodule
```

We have previously seen the 2-to-1 MUX being specified as combinational circuit in Verilog using the `always` construct with the **sensitivity list**.

The right hand diagram shows how a clocked **sequential circuit** is being specified using `always` block, but with a sensitivity list that includes the keyword `posedge` (or `negedge`). Note that the clocking signal `clk` is an arbitrary name – you could call it "fred" or anything else!

The sensitivity list NO LONGER contains the input signals `a`, `b` or `sel`. Instead the hardware is specified to be sensitive the positive edge of `clk`. When this happens, the output changes according to the specification inside the `always` block.

Two assignments ("=" and "<=") are shown here. I will explain the difference between these later.
Therefore in Verilog, you specify flipflops using **always block** in conjunction with the keyword **posedge** or **negedge**.

Here is a specification for a D-flipflop with synchronous clear which is low active (i.e. clear the FF when clearb is low).

You may have more than one **always** block in a module. But if this is the case, beware that the two **always** blocks will **execute in parallel**. Therefore they must NOT specify the same output, otherwise a **race condition** exists and the result is unpredictable.

Here is a specification for asynchronous clear of the D-flipflop. Either positive edge on **clock** or negative edge on **clearb** will cause the statements inside the **always** block to take effect.

I must remind everyone that the code shown here is a **specification**. They are **synthesised** into logic circuits – they are NOT executed as in a C programme.
In Verilog 'a=' is known as **blocking assignment**. They are executed in the order they appear within the Verilog simulation environment. So the first 'a=' assignment blocks the second one. This is very much like what happens in C codes.

In the top left example, both `a` and `b` eventually have the value `b`.

In the top right example, each statement is evaluated in turn and assignment is performed immediately at the end of the statement.

**Non-block assignment** is '<=' and statements with this assignments are executed in parallel (i.e. order do not matter).

In the bottom left example, `a` and `b` are swapped over because you can view that the two assignments happen at the same time.

In the bottom right example, three evaluations are made, and the assignment to `x`, `y` and `z` happens at the same time on exiting from the **always** block.

Understanding the difference between '=' and '<=' is important. Suppose we want to specify a three-stage shift register (i.e. three D-FF in series as shown in the schematic).

Here are two possible specification. Which one do you think will create the correct circuit and which one is wrong?
Use NONBLOCKING assignment for sequential logic

always @(posedge clk)
begin
    q1 = in;
    q2 = q1;
    out = q2;
end

always @(posedge clk)
begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
end

- At each rising clock edge:
  q1 = in,
  then, out = q1 = in.
  Therefore out = in, which is NOT the intention.

- At each rising clock edge, q1, q2 and out simultaneously receive the old values of in, q1, q2 respectively.

The left hand specification is wrong. Since the three assignments are performed in sequence, out = q2 = q1 = in. Therefore the resultant circuit is ONE D-flipflop.

The right hand side is correct. q1, q2 and out are updated simultaneously on exit from the always block. Therefore their "original" values MUST be retained. Hence this will result in three D-flipflops being synthesised (i.e. created).

In general, you should always use <= inside an always block to specify your circuit.

A larger example – 32-bit ALU in Verilog

Now let us put all you have learned together in specifying (or designing) a 32-bit ALU in Verilog.

There are five operators in this ALU. We assume that there are three arithmetic blocks, and three multiplexers (two 2-to-1 MUX and one 3-to-1 MUX).
Each hardware block is defined as a Verilog module. So we have the following modules:

- **mux32two** – a 32-bit multiplexer that has TWO inputs
- **mux32three** – a 32-bit multiplexer that has THREE inputs
- **mul16** – a 16-by-16 binary multiplier that produces a 32-bit product
- **add32** – a 32-bit binary adder
- **sub32** – a 32-bit binary subtractor

Now let us put all these together.

Note that **mux32two** is being used twice and therefore this is instantiated two times with two different instance names: **adder_mux** and **sub_mux**.

Connections between modules are implicit through the use of signal names. For example, the 16-bit inputs to the multiplier are taken from the lower 16-bits of **a** and **b** inputs (i.e. **a[15:0]** and **b[15:0]**).
Testbench – Better than waveform editor

- Testbench is a module NOT for hardware synthesis, but for testing and debugging only.
- Verilog has behavioural statements to help implementing testbench.
- Here is an example of a 4-bit full adder defined from low-level up:

```
Full Adder (1-bit)
module full_adder (a, b, cin, sum, cout);
    input a, b, cin;
    output sum, cout;
    reg sum, cout;
    always @(a or b or cin)
        begin
            sum = a ^ b ^ cin;
            cout = (a & b) | (a & cin) | (b & cin);
        end
endmodule
```

```
Full Adder (4-bit)
module full_adder_4bit (a, b, cin, sum, cout);
    input [3:0] a, b;
    input cin;
    output [3:0] sum;
    output cout;
    wire c1, c2, c3;
    // Instantiate 1-bit adders
    full_adder FA0(a[0],b[0],cin,sum[0],c1);
    full_adder FA1(a[1],b[1],cin,sum[1],c2);
    full_adder FA2(a[2],b[2],cin,sum[2],c3);
    full_adder FA3(a[3],b[3],cin,sum[3],cout);
endmodule
```

Instead of specifying the adder through the ‘+’ operator, here is an example of a 4-bit adder specified as low level logic operations.

Testbench to test the 4-bit full adder

- `initial block together with #<time>` define input vectors at different times to test circuit.

To test this module, we can use the `behavioural` feature of Verilog and specify a test module known as `testbench`.

The first statement instantiates the `full_adder_4bit` module.

The `initial` block and the `#<time>` keywords specify how the module would be exercised or tested.

The idea is that once you have created this `testbench`, you could change the design of the `full_adder_4bit` modules and have it tested in exactly the same way without touching the `testbench` again.
Quiz

1. What is the three types of logical operators?
2. What is the difference between \(~a\) and \(\neg a\)?
3. What is the common consequence of incompletely specify a combinational logic circuit?
4. How do you describe a simple D-flipflop in Verilog?
5. How would you describe a D-flipflop with asynchronous clear input?
6. What is wrong with this:
   
   ```
   always @ (clear or posedge clk)
   ```
    
   Answers are all in the notes.
7. What is the difference between blocking and nonblocking assignments?
8. If \(a = 4'h5\), \(b = 4'h3\), \(c = 4'h9\), what are the results after the following code segment is executed?
   ```
   a = b; b = c; c = a;
   ```
9. Same as above, but for the following code segment:
   ```
   a <= b; b <= c; c <= a;
   ```