Imperial College London

Lecture 4

Verilog HDL – Part 2

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

Lecture Objectives

- By the end of this lecture, you should understand:
  - The use of arithmetic and logic operations in Verilog
  - The danger of **incomplete specification**
  - How to specify **clocked circuits**
  - How to specify asynchronous and synchronous **set/reset** in flip-flops
  - Differences between **blocking** and **nonblocking** assignments
  - The use of **testbenches**

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.

Power of Verilog: Integer Arithmetic

- Arithmetic operations make computation easy:

```
module add32(a, b, sum);
  input[31:0] a,b;
  output[31:0] sum;
  assign sum = a + b;
endmodule
```

- Here is a 32-bit adder with carry-in and carry-out:

```
module add32_carry(a, b, cin, sum, cout);
  input[31:0] a,b;
  input cin;
  output[31:0] sum;
  output cout;
  assign {cout, sum} = a + b + cin;
endmodule
```


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]**.

Different types of Boolean Operators

- **Bitwise operators**: perform bit-sliced operations on vectors bit by bit
  - **~(4'b0101) = {~0,~1,~0,~1} = 4'b1010**
  - **4'b0101 & 4'b0011 = 4'b0001**
- **Logical operators**: return true or false (1-bit) results
  - **!(4'b0101) = ~1 = 1'b0**
- **Reduction operators**: act on each bit of a SINGLE input vector
  - **&(4'b0101) = 0 & 1 & 0 & 1 = 1'b0**

Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b	XNOR

Logical

!a	NOT
a && b	AND
a || b	OR

Note distinction between ~a and !a

Reduction

&a	AND
~&	NAND
|	OR
~|	NOR
^	XOR

There are three different types of Boolean operators:

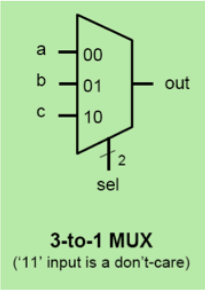
Bitwise operators perform what you would expect as if there are parallel gates used for each bit of the operands. Therefore **a&b** means that each bit from **a** and **b** is passed through an AND-gate.

Logical operators only result in 0 or 1 (i.e. 1-bit result) In this example !a **(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.

Beware of Incomplete Specification

Intention



What you may write, but wrong!

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

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


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.

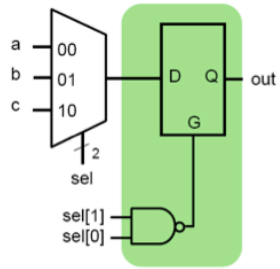
Incomplete specification: adds unwanted latch circuit

if `out` is not assigned during any pass through the always block, then the previous value must be retained!

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

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

Synthesized Circuit



- When sel = 2'b11, G = 0, therefore the latch stores the previous output value as required by Verilog in this situation.

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:

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

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

```
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'bx;
   endcase
 end
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'bx** (i.e. 'x') 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.

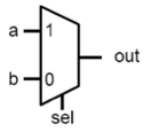
Lesson: always include a default assignment in any **case** statement to capture unintended incomplete specification.

How to specify a sequential circuit?

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

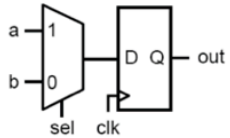
Combinational cct

```
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 cct

```
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.

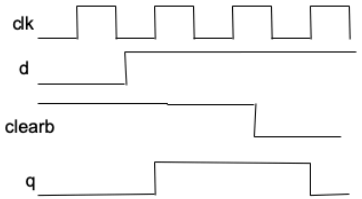
Two assignments ("**=**" and "**<=**") are shown here. I will explain the difference between these later.

Synchronous clear in D-flipflop

- **posedge** and **negedge** makes an always block sequential and edge-triggered
- Sensitivity list in a sequential **always** block determines what circuit is synthesized

D flipflop with **synchronous** clear

```
module dff_sync_clear(d, clearb,
clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```



+ve edge on clock triggers action in always block

- Beware of **race condition** if you have two or more always blocks – they execute in parallel!

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.

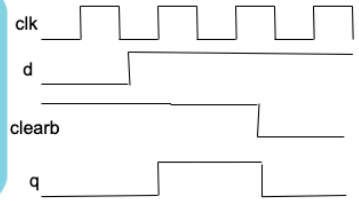
Asynchronous clear in D-flipflop

- If one signal in the sensitivity list uses **posedge** or **negedge**, ALL signals must also specify an edge. This syntax is wrong: **always @ (clearb or posedge clock)**

D flipflop with **asynchronous** clear

```
module dff_async_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;

always @ (negedge clearb or posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```



+ve edge on clock OR clearb trigger action in always block

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.

Blocking vs Non-blocking Assignments

- Verilog has two different types of assignments: **blocking** & **nonblocking**.
- Blocking assignments **=** are executed in the order they appear, therefore they are done one after another. Therefore the first statement "blocks" the second until it is done, hence it is called blocking assignments.

```
a = b;
b = a;
// both a & b = b
```

blocking

```
always @ (a or b or c)
begin
  x = a | b;          1. Evaluate a | b, assign result to x
  y = a ^ b ^ c;      2. Evaluate a^b^c, assign result to y
  z = b & ~c;         3. Evaluate b&(~c), assign result to z
end
```

blocking

- Non-blocking assignments **<=** are executed in parallel. Therefore an earlier statement does not block the later statement. Note the subtle effect this has within **always** block:

```
a <= b;
b <= a;
// swap a and b
```

Non-blocking

```
always @ (a or b or c)
begin
  x <= a | b;         1. Evaluate a | b but defer assignment of x
  y <= a ^ b ^ c;     2. Evaluate a^b^c but defer assignment of y
  z <= b & ~c;        3. Evaluate b&(~c) but defer assignment of z
end                   4. Assign x, y, and z with their new values
```

Non-blocking

PYKC 15 Oct 2019 E2.1 Digital Electronics Lecture 4 Slide 11

In Verilog '**=**' is known as **blocking assignment**. They are **executed in the order** they appear within the Verilog simulation environment. So the first '=' 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.

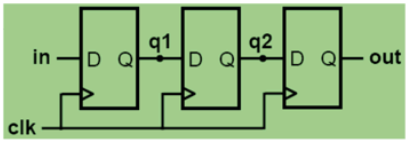
Non-block assignment is '**<=**', and statements with this assignments are **executed in parallel** (i.e. order do not matter).

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.

Be careful to use the correct assignment

- Here are two versions of a 3-stage shift register consisting of 3 flipflops using blocking and nonblocking assignments.
- Will they give the same results?




```
module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end
endmodule
```

```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end
endmodule
```

PYKC 15 Oct 2019 E2.1 Digital Electronics Lecture 4 Slide 12

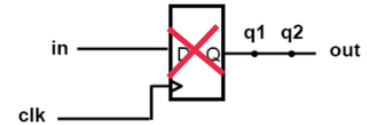
Understanding the difference between '**=**' and '**<=**' is important. Suppose we want to specify a three-stage shift register (i.e. three D-FF in series as shown in the schematic).

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

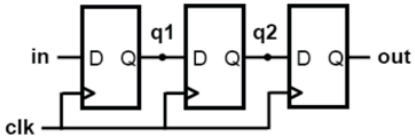
```
always @ (posedge clk)
begin
   q1 = in;
   q2 = q1;
   out = q2;
end
```

- At each rising clock edge:
  q1 = in,
  then q2 = q1 = in,
  then, out = q2 = q1 = in.
- Therefore out = in, which is NOT the intention.




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

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



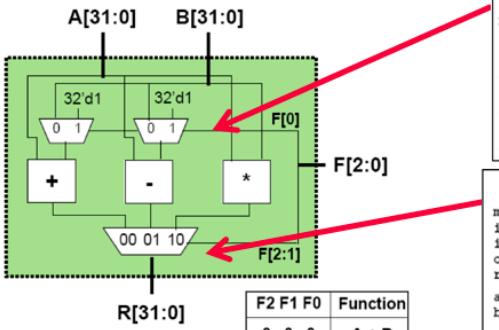
The left hand specification is **wrong**. Since the three assignments are performed in sequence, **out = q2 = q1 = in**. Therefore the resultant circuit is ONE D-flipflop.

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

- Here is an 32-bit ALU with 5 simple instructions:



F2 F1 F0	Function
0 0 0	A + B
0 0 1	A + 1
0 1 0	A - B
0 1 1	A - 1
1 0 X	A * B

2-to-1 MUX

```
module mux32two(i0,i1,sel,out);
input [31:0] i0,i1;
input sel;
output [31:0] out;

assign out = sel ? i1 : i0;

endmodule
```

3-to-1 MUX

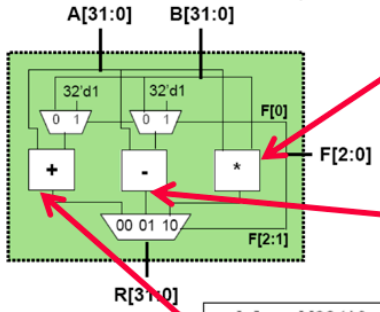
```
module mux32three(i0,i1,i2,sel,out);
input [31:0] i0,i1,i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;
always @ (i0 or i1 or i2 or sel)
begin
  case (sel)
    2'b00: out = i0;
    2'b01: out = i1;
    2'b10: out = i2;
    default: out = 32'bx;
  endcase
end
endmodule
```


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).

The arithmetic modules

- Here is an 32-bit ALU with 5 simple instructions:




```
module mul16(i0,i1,prod);
input [15:0] i0,i1;
output [31:0] prod;

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;

endmodule
```

```
module sub32(i0,i1,diff);
input [31:0] i0,i1;
output [31:0] diff;

assign diff = i0 - i1;

endmodule
```

```
module add32(i0,i1,sum);
input [31:0] i0,i1;
output [31:0] sum;

assign sum = i0 + i1;

endmodule
```


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

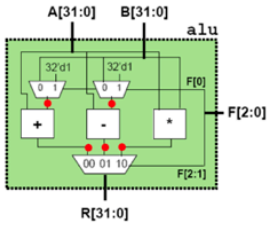
add32 – a 32-bit binary adder

sub32 – a 32-bit binary subtractor

Top-level module – putting them together

- Given submodules:

```
module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);
```




```
module alu(a, b, f, r);
  input [31:0] a, b;
  input [2:0] f;
  output [31:0] r;

  wire [31:0] addmux_out, submux_out;
  wire [31:0] add_out, sub_out, mul_out;

  mux32two    adder_mux(b, 32'd1, f[0], addmux_out);
  mux32two    sub_mux(b, 32'd1, f[0], submux_out);
  add32       our_adder(a, addmux_out, add_out);
  sub32       our_subtracter(a, submux_out, sub_out);
  mul16       our_multiplier(a[15:0], b[15:0], mul_out);
  mux32three  output_mux(add_out, sub_out, mul_out, f[2:1], r);

endmodule
```

intermediate output nodes

module names

(unique) instance names

corresponding wires/regs in module alu

Now let us put all these together.

Note that **mxu32two** 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], c1, sum[1], c2);
  full_adder FA2(a[2],b[2], c2, sum[2], c3);
  full_adder FA3(a[3],b[3], c3, 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.

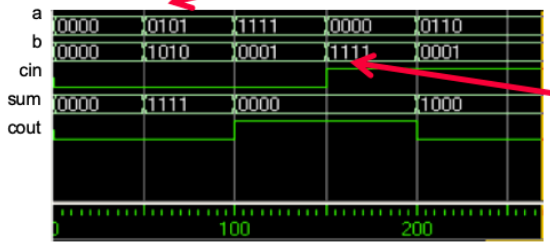
Testbench to test the 4-bit full adder

- **Initial** block together with **#<time>** define input vectors at different times to test circuit:

```
module test_adder;
  reg [3:0]  a, b;
  reg        cin;
  wire [3:0] sum;
  wire       cout;

  full_adder_4bit dut(a, b, cin,
                      sum, cout);
```

```
initial
  begin
    a = 4'b0000;
    b = 4'b0000;
    cin = 1'b0;
    #50;
    a = 4'b0101;
    b = 4'b1010;
    // sum = 1111, cout = 0
    #50;
    a = 4'b1111;
    b = 4'b0001;
    // sum = 0000, cout = 1
#50;
    a = 4'b0000;
    b = 4'b1111;
    cin = 1'b1;
    // sum = 0000, cout = 1
    #50;
    a = 4'b0110;
    b = 4'b0001;
    // sum = 1000, cout = 0
  end // initial begin
endmodule // test_adder
```



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 !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)

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;
```

Answers are all in the notes.