Australian National University

Week 3
Verilog HDL (continued)
Video Lecture 5

ENGN4213/6213
Digital Systems and Microprocessors

What's in this lecture

- Verilog HDL
  - More on blocking and nonblocking assignments
  - Procedural keywords and examples
  - Instantiations
  - Parameters
  - Timing specifications and test bench files
  - Examples

Resources

- Wakerly (5th edition) Chapter 5 for Verilog HDL
- Verilog code examples of combinational circuits can be found throughout Wakerly (5th edition) Chapters 6,7,8
- Verilog code examples of counters and shift registers can be found in Wakerly (5th edition) Chapter 11
- Many Verilog examples ready to compile and run with Icarus Verilog at the following website http://www.asic-world.com/examples/verilog/index.html

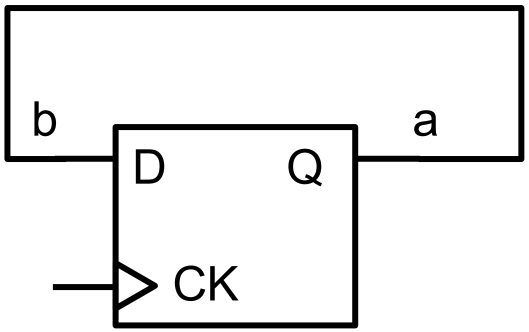
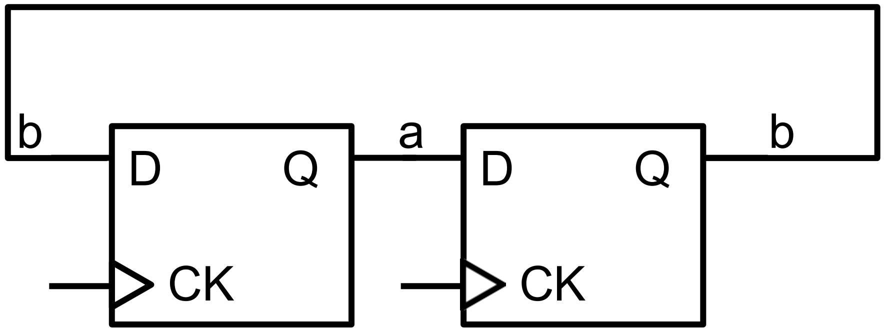
Blocking vs Nonblocking assignments (again)

- *Review from last lecture*
- Two ways to assign values inside an **`always`** block
- **Blocking assignment**
  - Are "executed" at the end of each line.
  - The result of one assignment affects the next.
  - Normally used for combinational designs with *level sensitivity lists* [e.g., **`always@(<signal_names>)`**, **`always@(*)`**]
- **Nonblocking assignments**
  - Are read in sequence but "executed" at the end of the **`always`** block.
  - The result of one assignment does not affect the next.
  - Normally used with *edge sensitivity list* [**`always@(posedge x)`**]

Blocking vs Nonblocking assignments (2)

- Typical example

`…`
`always` `@(posedge clk)` **`begin`**
`    a=b;`
`    b=a;`
`end` ✗
`…`
	`…`
	`always` `@(posedge clk)` **`begin`**
	`    a<=b;`
	`    b<=a;`
	`end` ✓
	`…`

Relatively useless  	Intended behaviour 

Blocking vs Nonblocking assignments (3)

- **Basic rules** are:
  - **Blocking assignments for combinational logic**
  - **Nonblocking assignments for sequential logic**
- Other important rules
  - **Don’t mix blocking and nonblocking** assignments in the same **`always`** block
  - Don’t assign the same variable in multiple **`always`** blocks

Procedural keywords

- Used only inside always blocks
- Allows you to give a *behavioural description* of a circuit rather than a *structural* one
- Does look a lot like programming
  - But remember that your "program" must be synthesisable in hardware!

The `if … else` keyword

- **Some circuit event only occurs *if* a particular condition is verified, *else* some other event will take place**
- Syntax

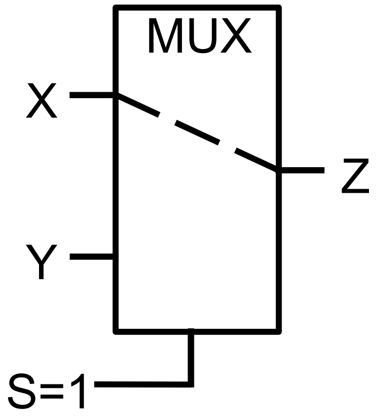
```
if (condition) begin
    <statement events if condition true>; end
else begin
    <statement events if condition false>;
end
```

- In **combinational designs**, you **must contemplate all *if* cases** in your code, otherwise the synthesiser may generate ***inferred latches*** (common mistake, also wastes hardware resources).
- **`begin … end`** constructs are used if there are multiple statements within following *if* or *else*

Example, the multiplexer

```
module MUX (
   input wire X,S,
   output reg Z);

   always @(*)
      if(S==1) Z=X;
      else Z=Y;
endmodule
```



The `case` keyword

- **Similar to `if` (for conditional statements), but useful when there are *multiple cases*, not just a single T/F condition.**
- Syntax

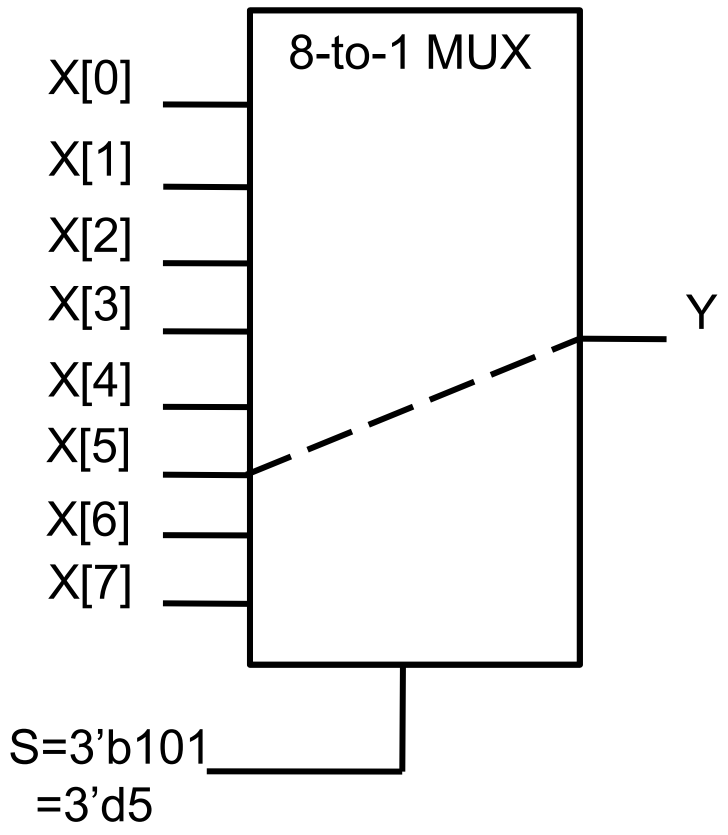
```
case (case variable)
    value#1  :  statement#1;
    value#2  :  statement#2;
    …
    default  :  statement if none of the above occur;
endcase
```

- **`begin`** … **`end`** constructs are used if there are multiple statements for any one case
- We have already seen examples of this as very useful to do MUX, DEMUX and DECODERS

Example: a bigger MUX (8-to-1)

```
module eight_to_1_MUX (
   input wire [7:0] X,
   input wire [2:0] sel,
   output reg Y);

   always @(*) begin: my_cases
     case(sel)
       3'd0 : Y = X[0];
       3'd1 : Y = X[1];
       3'd2 : Y = X[2];
       3'd3 : Y = X[3];
       3'd4 : Y = X[4];
       3'd5 : Y = X[5];
       3'd6 : Y = X[6];
       3'd7 : Y = X[7];
       default : Y=1'bx;
     endcase
   end
endmodule
```



Looping statements: `for, while, repeat, forever`

- **Carry out an operation over and over as long as**
  - **a looping index meets the requirements (`for`)**
  - **a counter is not exhausted (`repeat`)**
  - **a particular logical expression remains true (`while`)**
  - **the circuit is running (`forever`)**

- Syntax

`for` (*index=initial; expression; index=next*)
    procedural statements

`while` (*logical expression*)
    procedural statements

`repeat` (*integer number of times*)
    procedural statements

`forever`
    procedural statements

Looping statements (2)

- **Looping statements are dangerous:**
  - They can create *non-synthesisable designs*
  - They often introduce *feedback*
  - Other times designs will *blow out in terms of hardware space requirements*
- **We will limit their use or avoid them altogether**
  - Not particularly useful for hardware designs. If anything, for sequential designs only
  - Somewhat *useful in writing test benches*
  - This is an important practical difference with computer programming, where loops are very common.

Examples with looping statements

- An example that chews up hardware

```
…
always @(*) begin
    for(i=0;i<=63;i=i+1)
      Y[i]=X[i]+i;
end
…
```

This will create ***63 separate adders*** ("unrolling" the loop)

- Non-synthesisable code

```
…
always @(*) begin
    while(c<d) begin
      c=c+2;
      a=a+1;
    end;
end
…
```

This module, which uses *a* to count how many +2 operations are required in order for *c* to become greater than *d*, ***can't be synthesised*** because the number of repetitions in the loop is unknown *a priori*, so the loop can't be "unrolled".

Examples with looping statements (2)

- An example that is useful in a testbench file

```
…
forever begin
    #1 sysclk=1;
    #1 sysclk=0;
end
…
```

  - This creates a signal *sysclk* which switches value between 0 and 1 at every time increment.
  - This code *is not synthesisable* but it is useful to create a clock source for simulation purposes, for example.
  - It uses timing specifications (that `#1`), which we will discuss shortly.

Instantiation

- The hierarchical nature of the Verilog language can be exploited by coding sub-modules separately and then calling upon them (***instantiating***) from a higher-level module. Let's refer to the calling module as the *master (m)* and the called module as the *slave (s)*
  - Instantiation is routine in simulations where the test bench file has the master role and the module under test is *an instance* of the slave
- **Instantiation syntax:**

```
module_name instance_name ( .s1(m1), .s2(m2), … );
```

- Each slave module signal (`s#` stands for slave signal `#`) is listed preceded by a dot, while the corresponding signal in the master module is named in brackets (`m#` stands for master signal `#`).
  - **Paired signals need not have the same name**

Instantiation example

Slave module

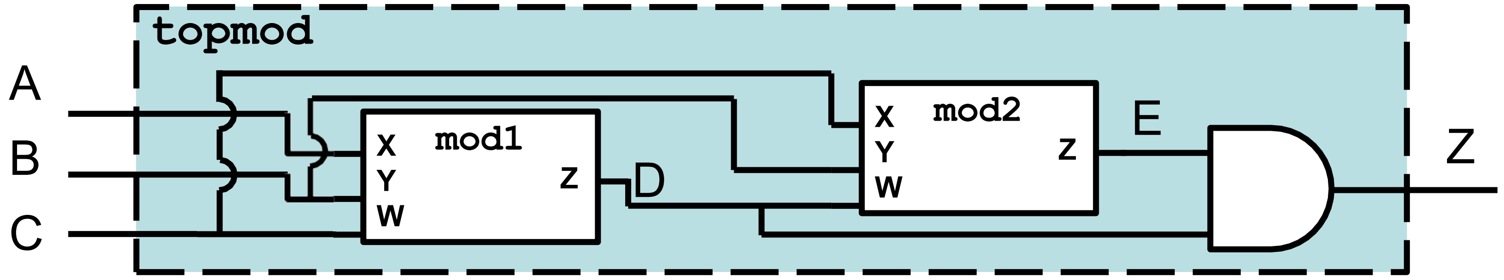
```
module boolmod  (
   input wire X, Y, W,
   output reg Z);

   always @(*)
        Z = (X & Y)|(X & W);
end
endmodule
```

Master module

```
module topmod (
   input wire A, B, C,
   output wire Z);
   wire D, E;
   boolmod mod1(.X(A),.Y(B),
     .W(C), .Z(D));
   boolmod mod2(.X(C),.Y(B),
     .W(D), .Z(E));

   assign Z = D & E;
endmodule
```



Parameters

- A clever tool to create modules which can be instantiated with varying bit-widths for inputs and outputs.
  - Saves time as it is not necessary to re-code the module each time

- **Syntax:**
  - Inside the module which will be instantiated
    `parameter par_name = value;`
    - The parameter value inside the module declaration is the *default value* of the parameter
  - When instantiation is performed at master module level
    `module_name #(par_value) instance_name (i/o signals);`
    - When used in an instantiation statement, the parameter value inside the `#( )` placeholder will override the default parameter value for that particular instance only.

Parameterisation example

- A variable-width adder (default width 2 bits)

```
module cooladder (
      input wire cin,
      input wire [WID-1:0] X,
   input wire [WID-1:0] Y,
   output reg cout,
      output reg [WID-1:0] Z);


   parameter WID=2;
   reg sum [WID:0];

   always @(*) begin
      sum = X+Y+cin;
      Z=sum[WID-1:0];
      cout=sum[WID];
   end
endmodule
```

- Three instantiations where a 2, 4 and 8-bit adders are instantiated

```
… //2-bit adder (default WID)
cooladder add2 (.cin( ),.X( ),
  .Y( ),.cout( ),.Z( ));
…

… //4-bit adder (WID=4)
cooladder #(4) add4 (.cin( ),
  .X( ), .Y( ),.cout( ),.Z( ));
…

… //8-bit adder (WID=8)
cooladder #(8) add8 (.cin( ),
  .X( ), .Y( ),.cout( ),.Z( ));
…
```

Directives

- High-level commands which inform compilation
  - **`` `include “*filename*” ``**
    - The named file is read and processed as part of the file containing the directive. It is a convenient way to refer programmatically to module definitions located in different folders.
  - **`` `define *identifier text* ``**
    - Used to rename things. Every time *identifier* is used in the code, it will be replaced by text. I could, for example, rename the net type *wire [3:0] to bus4.*
  - **`` `timescale *ts / tp* ``**
    - Used to specify the time step (*ts*) and precision *(tp)* to be adopted NOTE: this can be used in simulations only!

Australian National University

System tasks

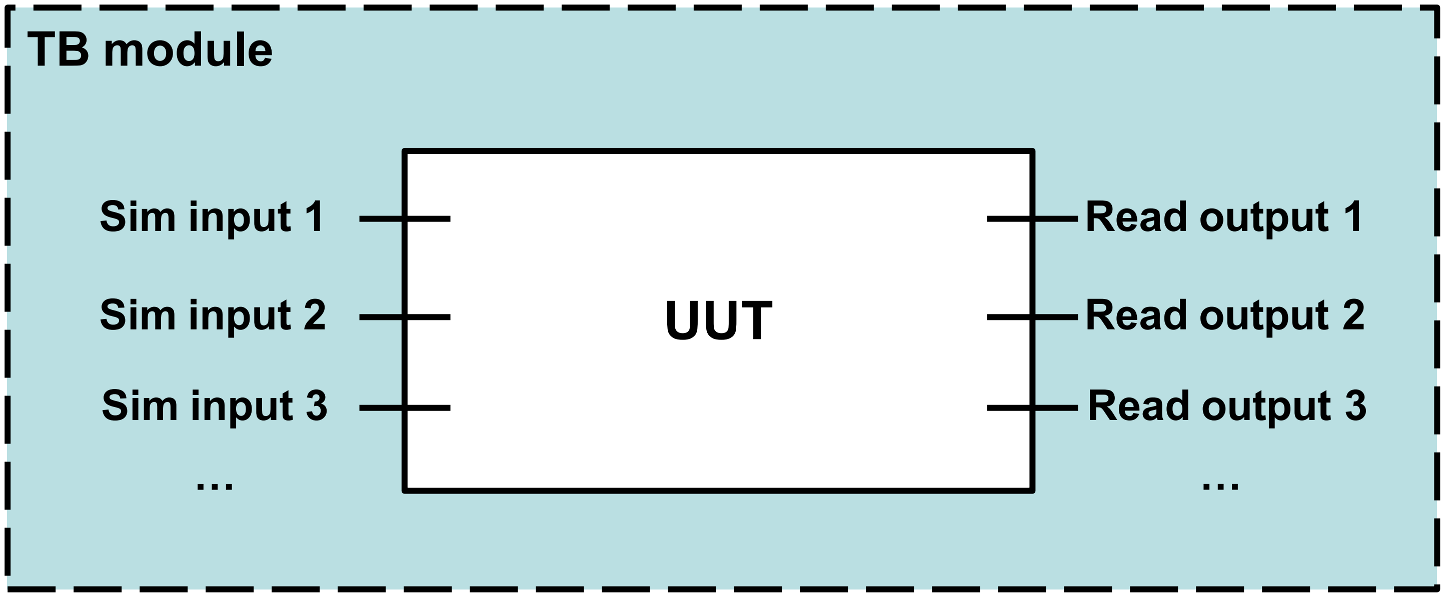
- A number of predefined system tasks affect the behaviour of the simulator. There are many. Only a handful of useful ones are listed here. We will use some in test benches.
  - **`$dumpfile`**`("`*filename*`")` is used to specify the name of the file in which the simulation results will be saved (generally a .vcd file)
  - **`$dumpvars`**`(`*mode*`,` *list_of_variables_or_modules*`);` specifies that changes in the variables need to be saved in the dump file
    - *mode* and *list* are optional. If *mode* is 0, all variables of the listed modules will be dumped; if 1, only the listed variables from the listed modules will be dumped.
  - **`$display`**(*string*)`;` can be used to print a text-based output in the terminal console as the simulation is running
    - the format of the string is given in C-language-like syntax.
  - **`$monitor`**(*string*) `;` like $display but always active so prints the output at every signal change.
  - **`$finish`** forces a simulation to terminate

Timing specifications

- Can be used to specify time delays in circuit operations
  - Time delay specifications are ***not synthesisable***
  - They are commonly used in test benches
- **Syntax:**
  - Basic construct: `#X` where `X` is to be replaced by the required number of time delay steps
    - Remember: the length of a time step is specified by `` `timescale ``
  - Depending on where the construct is placed in a statement, different timing effects can be achieved. The most common:
    - `wire #10 A;` all assignments which affect A will only take effect after 10 time steps
    - `assign #5 a = b+c;` the value of `b+c` is evaluated instantly but assigned to `a` after 5 time steps
    - *inside procedural blocks* `#7 q = x+y;` executes the assignment after 7 time steps

Test benches

- A test bench is a top module with no external inputs or outputs
- It is only valuable for simulation
- It provides a description of test signals which will be fed to the inputs of a design under test and lists the test outputs which will be read (UUT= Unit Under Test)



Test benches: typical structure

Essential test bench structure (code very similar to CLAB1)

```
`timescale 1ns / 1ps //specifies simulation time steps (default 1ns)
module TB_mux1;        //A TB is a normal module, just with no i/o
   reg X;
   reg S;              //Inputs: note they are of type reg
   reg Y;
   wire Z;             //Outputs: note they are of type wire

   mux mx(.X(X), .S(S), .Y(Y), .Z(Z)); //instantiation of module under test

   initial begin
         $dumpfile ("mux.vcd");      //dumpfile name specification
         $dumpvars;                  //activate variable change dump (all)
         #20; $finish;               //specifies simulation duration (20ns)
   end

   initial begin
         #1X = 1'b0;
         #1Y = 1'b1;
         #1S = 1'b1;       //procedural block (initial) to specify signal changes
         ...
         #1Y = 1'b0;
         #1S = 1'b0;
   end
endmodule
```

Australian National University

The `initial` block

- It is a procedural block, so it can contain procedural statements like the **`always`** block.
- It's generally **not synthesisable** (there are exceptions)
  - **Only used in simulation** modules
- Unlike always, it has no associated sensitivity list.
  - the code is **executed *at the beginning*** of the simulation (at time 0)
- It is ok to have multiple initial blocks
- The **`begin`** … **`end`** construct can be used when multiple lines of procedural code are required

Concluding remarks

- This concludes our **quick overview** of Verilog HDL fundamentals
- There are a number of more complex advanced features which we have not discussed, but **what we have covered is sufficient to carry out moderately complex designs**.
- I recommend you **limit yourselves to learning these essentials** for the purpose of the course (to avoid confusion).
- The best way forward is to **write your own code and study already written examples** (plenty of them in your textbook).

- **The very essential things to remember:**
  - Verilog can do both **simulation and synthesis**, and **some constructs which work in simulations do not work in synthesis**
  - There are **general rules on how to do sequential and combinational designs**. Although they are not absolutes, **if you follow them you should be able to avoid common mistakes**.

Australian National University

The key coding rules: **Verilog for synthesis**

General rules

- No **`initial`** blocks
- Avoid looping statements whenever possible
  - Think that the synthesiser will work only if it can "unroll the loop"
- No delay specifications
- Specify the bit width of all numbers you use.
  - It's good practice and avoids involuntary hardware wastage
  - Example: `a[3:0] = b[3:0] + 8;` ***is 8 a 32-bit integer (default)?*

    viz: `a[3:0] = b[3:0] + 4'd8;`
- In conditional statements prefer **`case`** to nested **`if/else`** if there are multiple cases to prevent the synthesis of slow gate chains in hardware.

The key coding rules: **combinational logic**

- Value assignments inside the design module must be done with **all inputs on the RHS, and all outputs on the LHS**
- **No feedback loops** permitted (follows from the previous rule)
- Use **continuous assignment** **`assign`** for simple comb. logic
- Use a **level sensitivity list** **`always`** `@(*)` for procedural blocks
- Use **only blocking assignments** "="
- **`If`** **statements must be complete**, i.e., all variables changed within an **`if`** statement should have a value assigned to them regardless of whether the conditional clause is true or false (use of else or resort to default cases). This is to ***avoid inferred latches***.

The key coding rules: **sequential logic**

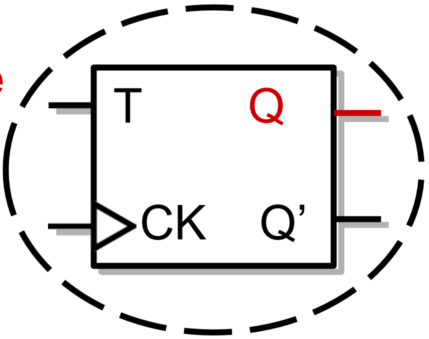
- Use an **edge sensitivity list** with a single clock entry, e.g., `always @(posedge sysclk)`
- Use only **non-blocking assignments** "**<=**"
- The clock triggers the execution of procedural blocks, but **the clock does not appear inside the block**
- **`If` statements do not need to be complete**
- **Feedback loops are fine**
- Any **external signals must be synchronised** (to avoid metastability)

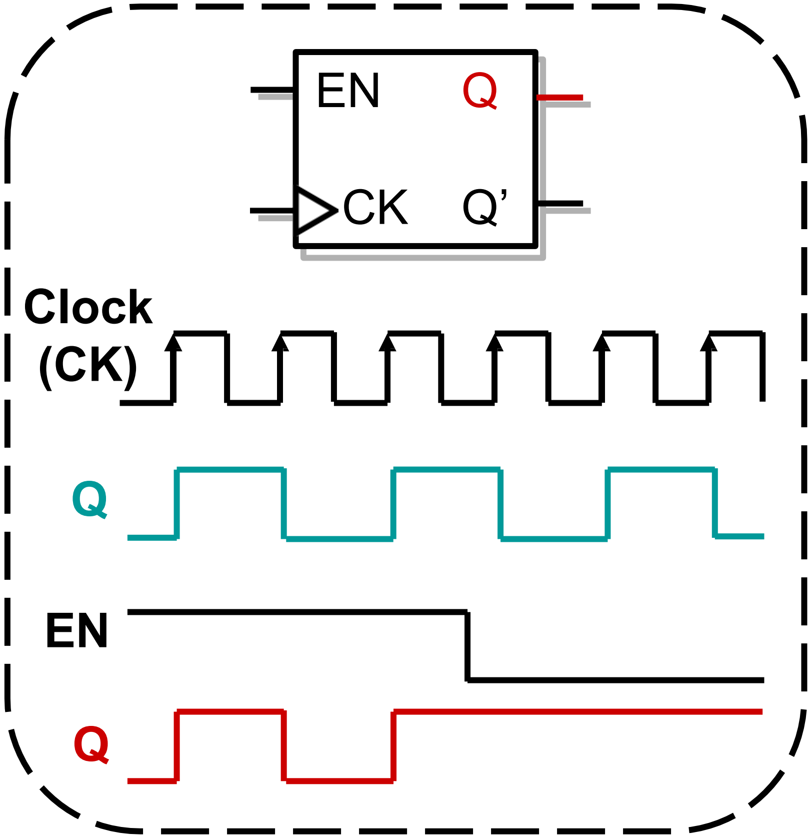
Examples

- The T-Flip flop

```
module tff(
input wire CK,
input wire EN,
output reg Q);
  always @(posedge CK) begin
     if (EN) Q <= ~Q;
  end
endmodule
```

Note possible alternative symbol

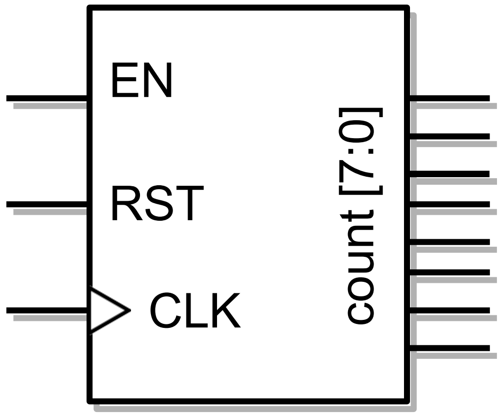




Examples (2)

- An 8-bit up-counter with enable and reset

```
module counter8(
input wire CLK, RST, EN,
output reg [7:0] count);
   always @(posedge CLK) begin
      if(reset) count <= 8’h00;
      else
         if(EN) count <= count + 1’b1;
   end
endmodule
```

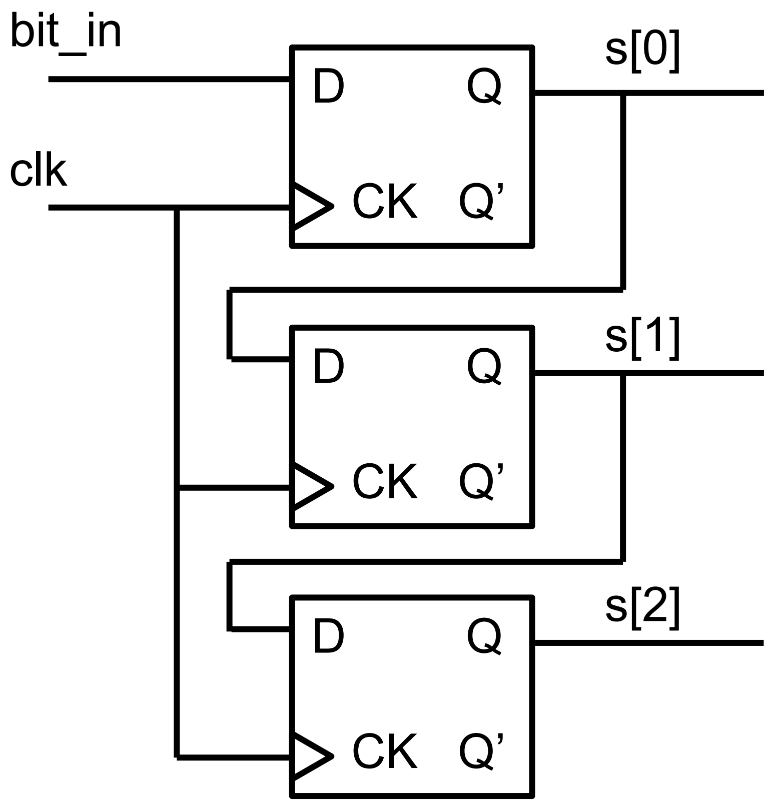


Examples (3)

- A 3-bit SIPO (serial-in parallel-out) shift register

```
module siporeg(
input wire clk, reset, bit_in,
output reg [2:0] s);

   always @(posedge clk) begin
      if(reset)
      s <= 3'b000;
      else begin
         s[0] <= bit_in;
         s[1] <= s[0];
         s[2] <= s[1];
      end
   end
endmodule
```

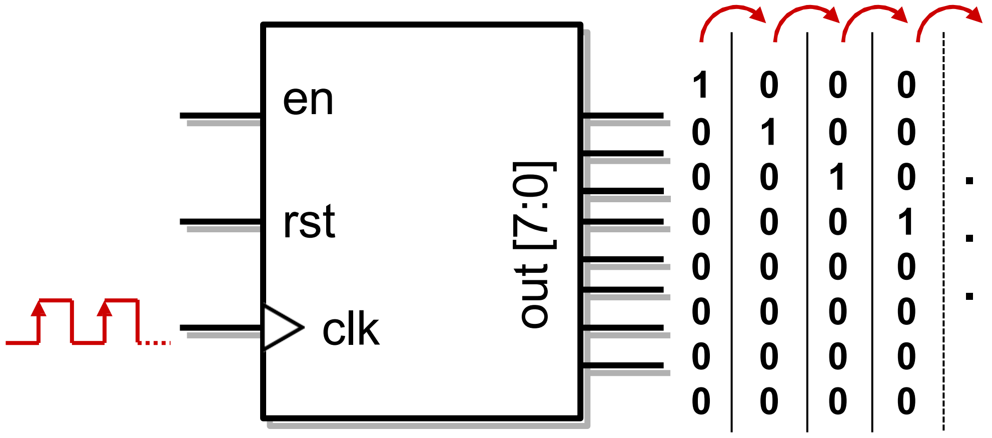


Example (4)

- An 8-bit “one-hot” counter

```
module one_hot_cnt8 (
input wire en, clk, rst,
output reg [7:0] out);

always @ (posedge clk)
   if (rst) begin
      out <= 8'b00000001 ;
   end else if (en) begin
      out <= {out[6],out[5],out[4],out[3],
              out[2],out[1],out[0],out[7]};
   end
endmodule
```



Summing up

- We have completed a full overview of the **main features of Verilog HDL.** Today's lecture introduced
  - Procedural statements
  - Instantiations
  - Timing specifications
  - Test bench files and their typical structure
- We have learned that the language can be used for circuit **simulation** and **synthesis**
  - The two uses have **some differences** in the way the code is written
  - There are **good coding practices** for **combinational** and **sequential** logic designs
- We have commented on a few significant examples
  - Now it is up to you to master the language through **practice, practice, practice.**