



Australian  
National  
University

# Lecture 06

## Essential sequential designs, Introduction to Verilog HDL

ENGN3213 Digital Systems and Microprocessors

Semester 1, 2013

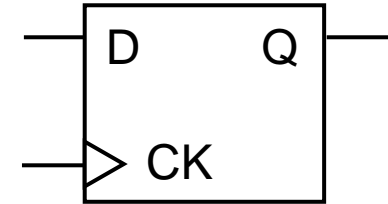
# What's in this lecture

- Essential sequential designs
  - Counters
  - Clock dividers
  - Shift registers
- Verilog HDL basics
  - Basic declarations and variable types
  - Procedural blocks
  - Blocking and nonblocking assignments

# Resources

- Wakerly 8.4.1-8.4.2 for counters
- Wakerly 8.5.1 for shift registers
- Wakerly 5.4 for Verilog HDL
- There is a section on Verilog in the 2008 Brick Appendix
- Verilog code examples can be found throughout Chapters 6, 8
  - Usually the book explains a basic design, e.g., a multiplexer, and then shows a possible Verilog HDL code description. Don't look at these until after the lecture. It might be a bit hard to understand.
  - Note that the book describes 3 different languages: Verilog, VHDL and ABEL. **We are only interested in *Verilog*.**

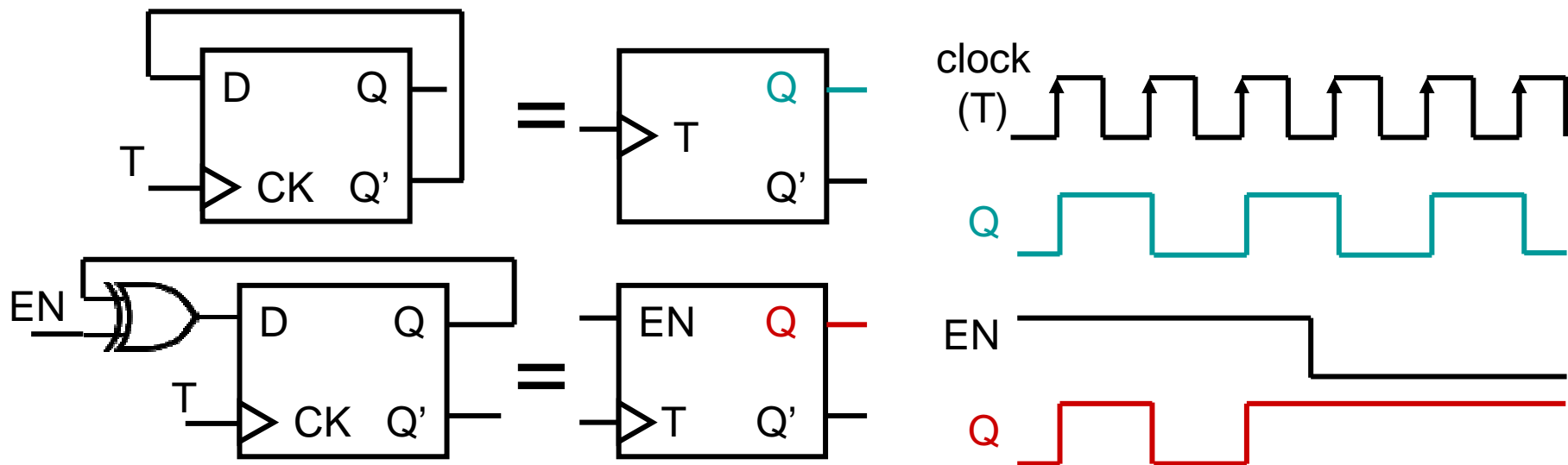
## Some uses of flip-flops



- Last week we discussed D flip-flops
  - Circuits which sample the value at their D input as an active edge is presented to their *clock* input port
- D flip-flops are the type used in FPGA hardware
- Today we start off with some basic but essential and widespread designs which use flip-flops
  - Today we will only give a cursory look at this material. Much more detailed insights on sequential circuit design will come when we discuss finite-state machines.

# The “T” flip flop

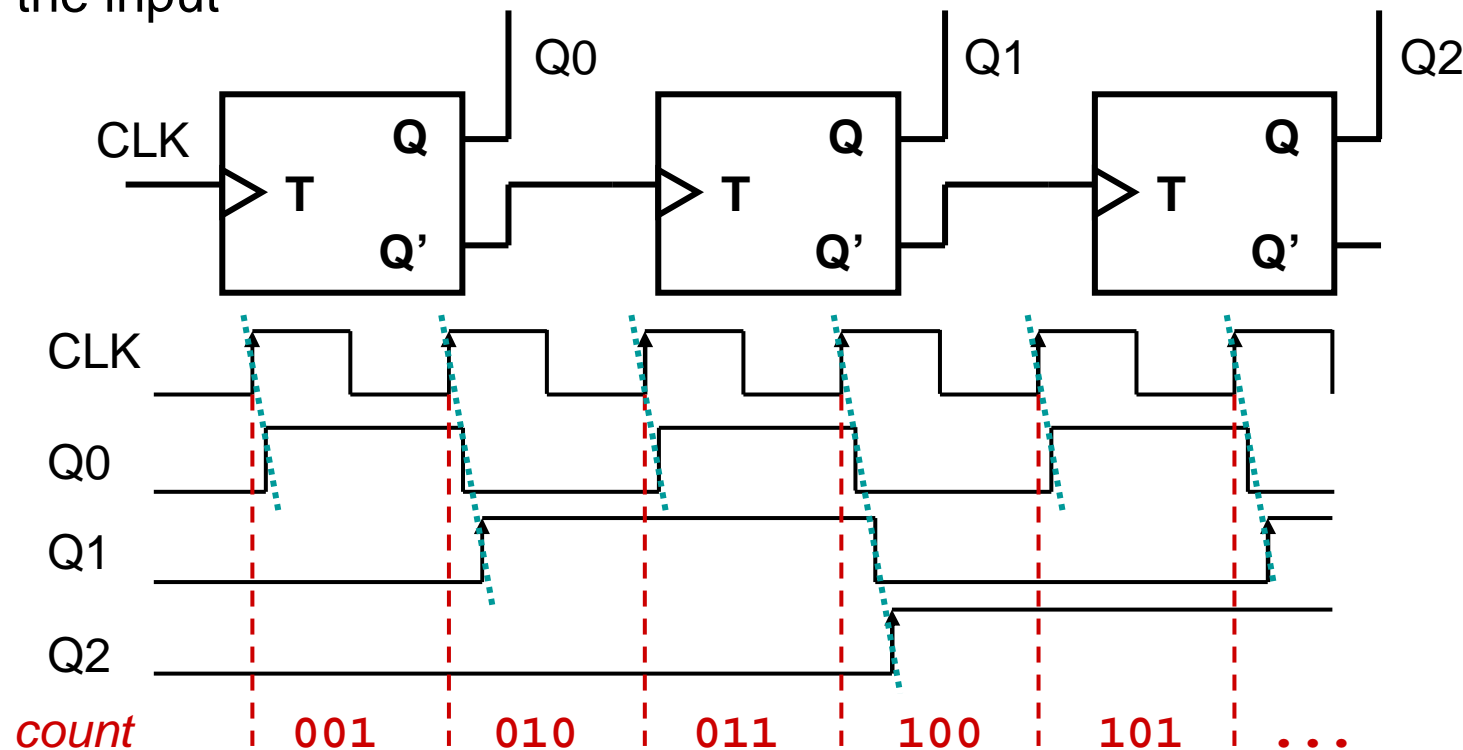
- I promised that we would only learn D flip flops. This is not really a different type, just a variation.
- T stands for **T**oggle. This design switches between logic values each time an active edge appears at the input T.



- This example, incidentally, shows that **feedback is allowed** in sequential circuits, can you explain why?

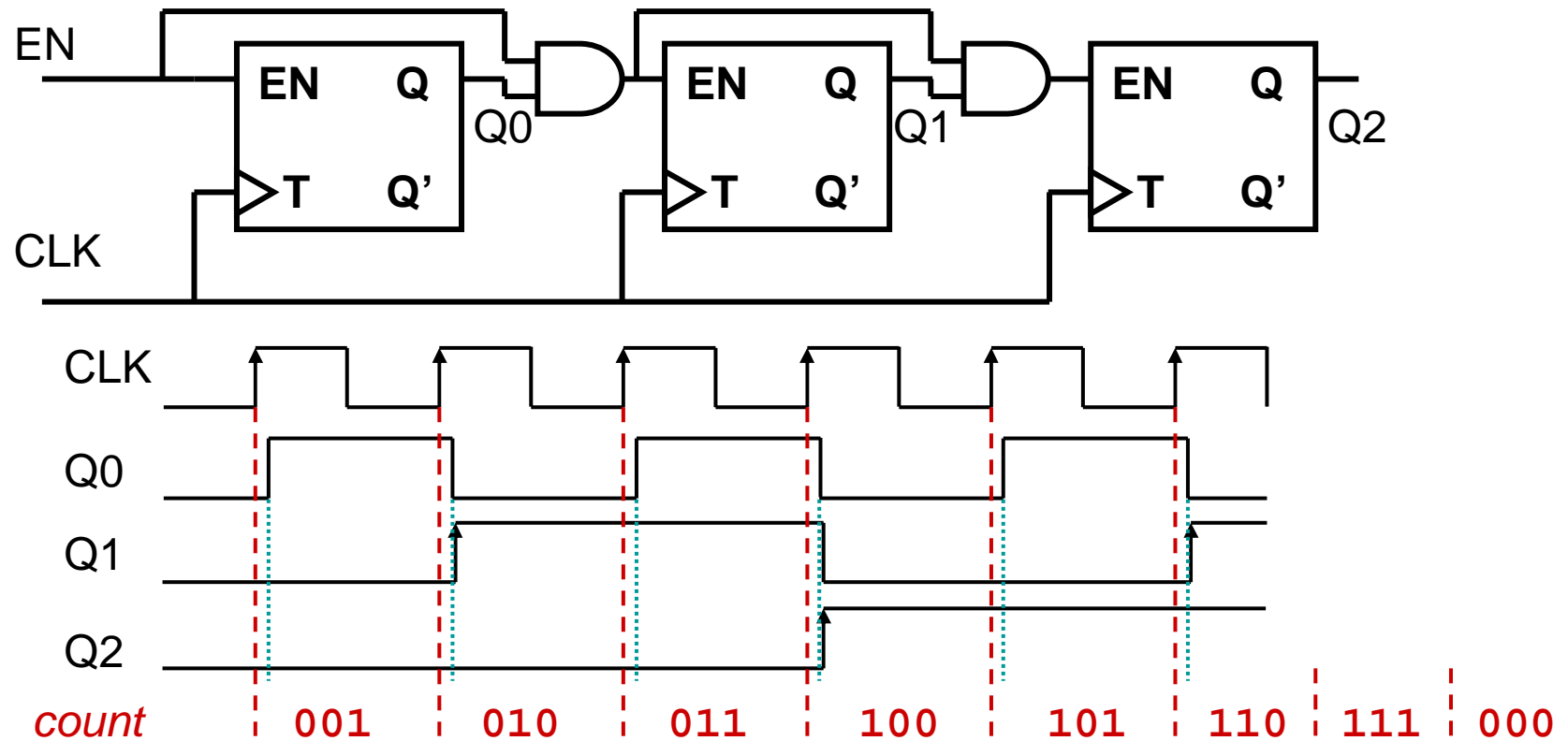
## Asynchronous counter (“ripple” counter)

- **Counter:** a circuit which returns the number of active edges observed at the input



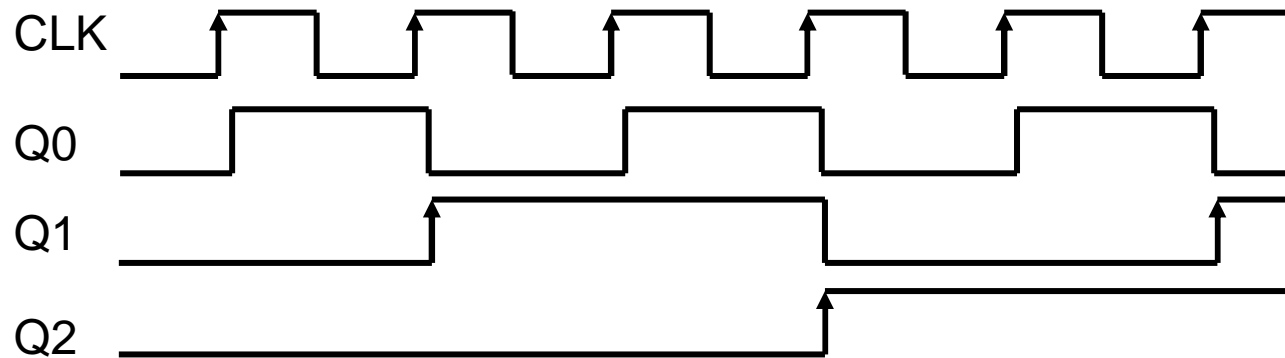
- **Asynchronous:** the update of the count bits is not simultaneous with the input clock (due to a “ripple” effect). This can be undesirable in some applications.

# Synchronous counter



- **Synchronous:** all flip-flops in the chain switch at the same time.

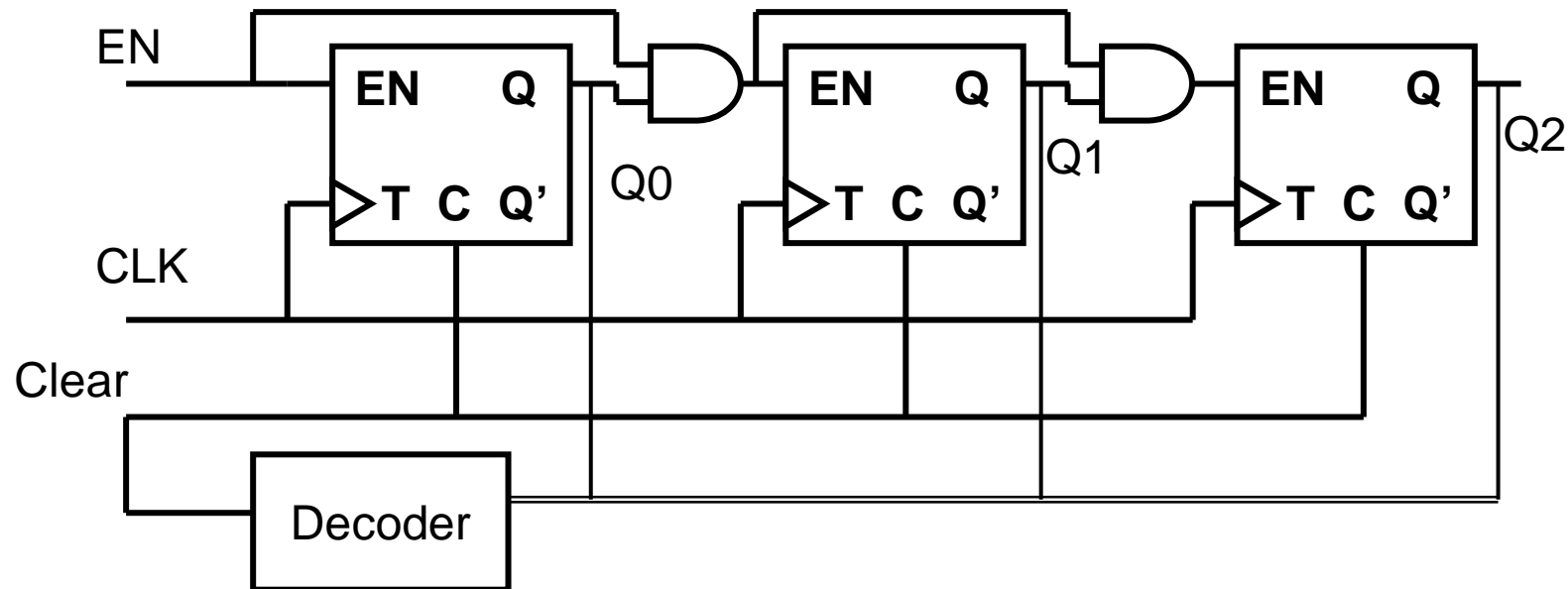
## Counters as clock dividers



- A binary counter can be used as a  $2^n$  **clock divider**. You can see that the Q0 output toggles at  $\frac{1}{2}$  the frequency of CLK, Q1 at  $\frac{1}{4}$ , etc.
  - Sometimes called *modulo-m counters* as they count up to  $m$
- We will use this in some applications where we want the system (or parts of it) to run at slower clock speeds.
- Class exercise. How could we design a counter that counts up to an arbitrary number? (not a power of 2)



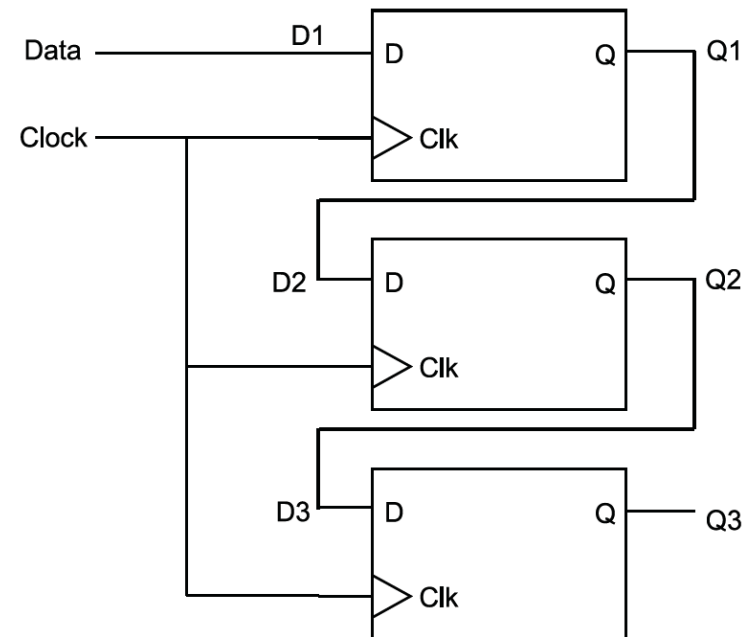
## Arbitrary $m$ up-counter



- The decoder associates the binary sequence for  $m$  with a 1 output.
- Instead of connecting the equality result to the counter reset, it can be used to activate some other circuit event. Such use is called an **event sequencer**.

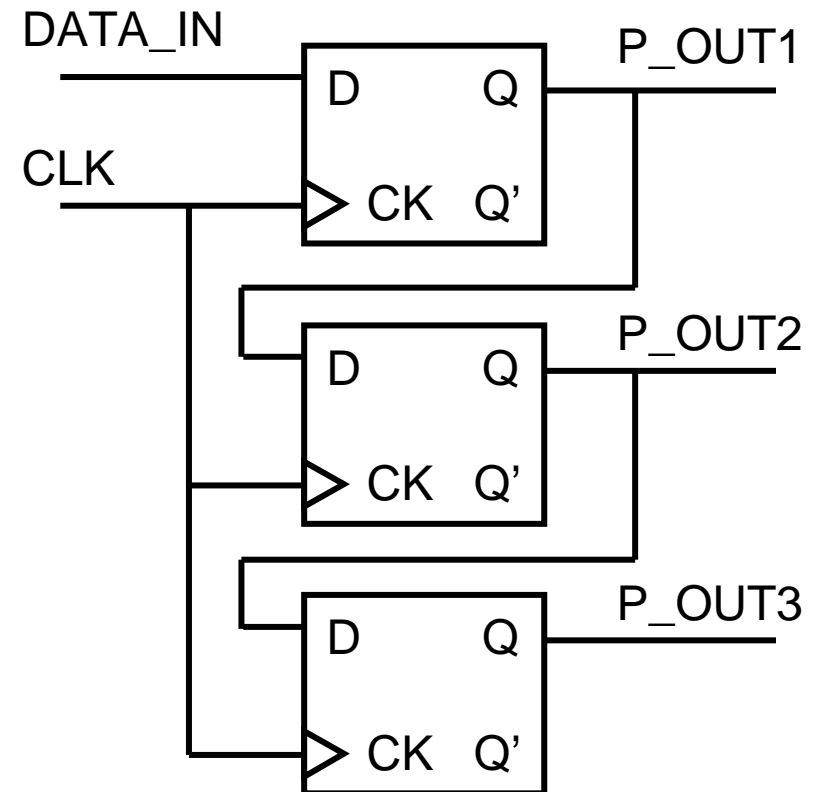
# Shift registers

- An assembly of  $n$  flip-flops with a provision for shifting data between them. E.g.,
- At each active clock edge, data is shifted from a FF to the next
  - The one on the right is called a **serial-in, serial-out** shift register. Data words are read at the input and produced at the output one bit at a time.



## Shift registers (2)

- As well as serial-in serial-out (SISO) register, there are also others where the data word is read in or out with all bits at once (**parallel**).
  - Parallel operation is faster but requires more wires.
  - The circuit on the right can be used either as SISO or as SIPO (serial-in, parallel-out).
  - What would PIPO and PISO look like?

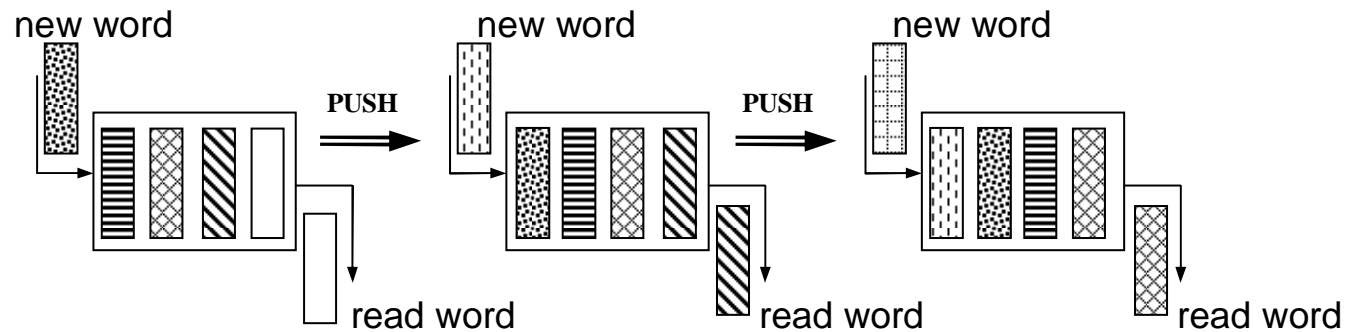


## Shift registers (3)

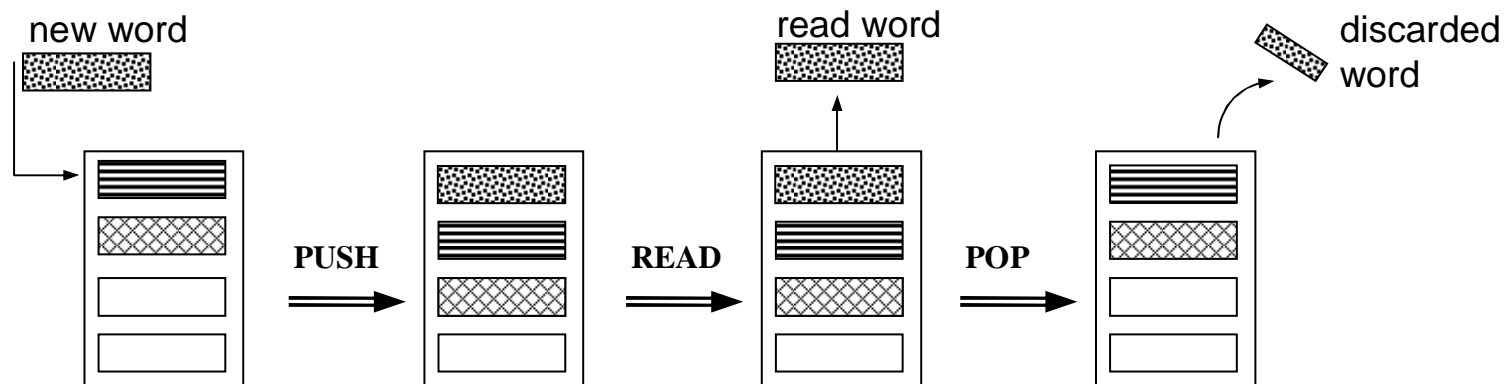
- Another distinction between registers is given by the possible shift directions of data.
  - In *first-in, first-out* (FIFO) registers, data flows through in a queue fashion
  - In *last-in, first-out* (LIFO) registers, you can imagine data as stacked in a pile, with the last added item being on top
  - You will encounter FIFO and LIFO registers in HLAB2
- Figure 8-41 of Wakerly shows a universal shift register, a versatile commercial design that can do it all: SISO, PISO, SIPO PIPO, FIFO and LIFO (!!)

# FIFO and LIFO

- FIFO**



- LIFO**



# Summing up and moving on

- We have seen and discussed some basic uses of flip-flops
  - Counters
  - Clock dividers
  - Shift registers
- We have seen that **feedback is admissible** in sequential circuits and in fact it can be rather useful.
- We have again encountered matters of **synchrony and timing**. Synchrony errors are by far the most common issue with designs so I will insist on timing-related concepts over and over throughout the course.

# Verilog HDL

- *HDL = Hardware Description Language*
- A **BIG** topic in the course.
- A new language that focuses on ***describing hardware modules***, both *structurally* (what they are made of) and *behaviourally* (what they do).
- **It is not a programming language.**
  - Although there is a parallel in the development chain
    - Program → Compiler → Assembly code → Machine code
    - HDL → Synthesiser → Gate-level netlist → Hardware

# HDL is not a programming language

- **This is very important** and I will repeat it *ad nauseam*.
  - Still, you will hear the expressions “programming in HDL”, “Verilog programs”, etc. I prefer using the words “coding in Verilog”, “Verilog module description” etc. for clarity.
- Programs as you know them represent step-by-step algorithms
- HDLs are a description of hardware. The order of operations is dictated by the interconnection of components, not by the order of code lines
- HDL are not hugely dissimilar from schematics, but they have advantages, especially for complex designs:
  - Instantiation of sub-components in hierarchical designs is easy
  - Behavioural description of blocks is faster as it allows some abstraction
  - Timing modelling can be specified for use in simulations
  - Easy integration with simulators and synthesisers

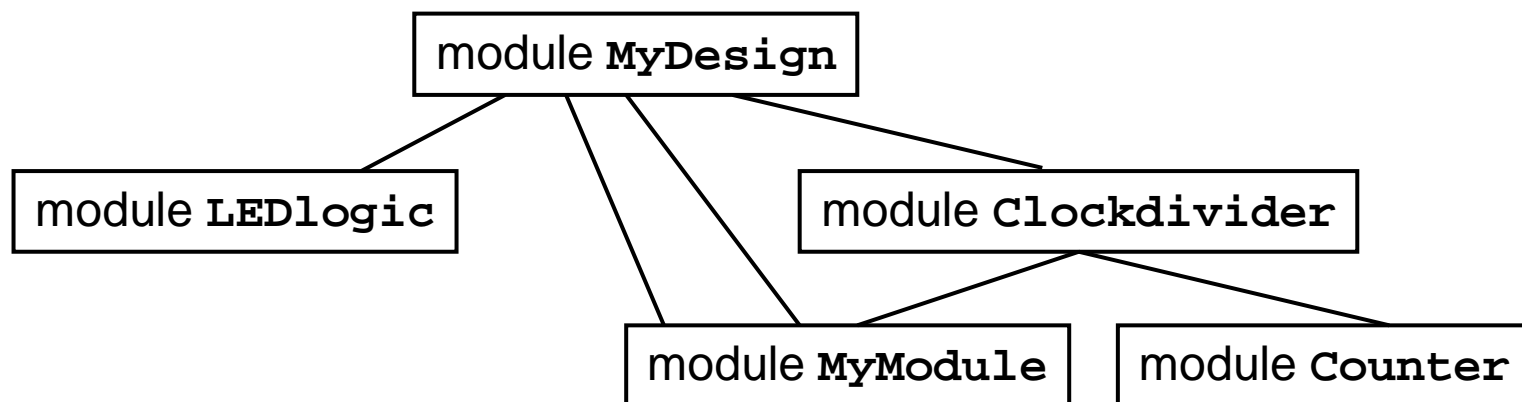


## Bad reasons to use HDL

- To avoid understanding the hardware
- Because I can't draw a schematic
- If you don't know what you are doing and could not draw a schematic, it is very unlikely that your Verilog design will work.

# Structure of a verilog file (.v)

- It is a standard text file
- Each file describes 1 entity (*module*)
- In a hierarchical design, higher level modules can *instantiate* lower level modules
  - Facilitates the re-use of code for modules performing similar operations
  - Makes the overall design easier to read and understand





## Structure of a verilog file (.v) (2)

```
module module name(ports);  
    input declarations  
    output declarations  
    net declarations  
    variable declarations  
    parameter declarations  
    instantiations (if any)  
  
    statements  
endmodule
```

```
module My_and (X,Y,Z);  
    input X, Y;  
    output Z;  
  
    assign Z = X & Y;  
endmodule
```

**Note:** your textbook will have some extra constructs, e.g. functions, tasks, etc.  
We don't cover everything so refer to the slides when you study.

**If I haven't explained some methods in class and you want to use them ask me first!**

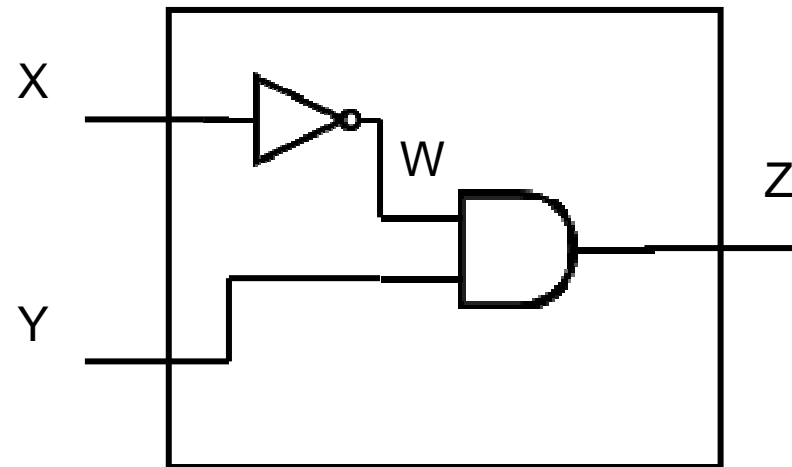
# Some very general Verilog

- It is case-sensitive (A and a are different)
- It has reserved keywords (`module` is one)
- Lines of code are generally terminated by a semicolon ;
- Notes can (should!) be added to your code
  - in-line (using `//this is a note`)
  - as a chunk of note text (`/* like this note blah blah blah */`)
- Logic values are 0 (false), 1 (true), x (unknown)
- Bitwise boolean operators are
  - & (AND),
  - | (OR),
  - ^ (XOR),
  - ~ (NOT)

# Inputs / outputs and nets

- **Inputs and outputs** represent the “ports” through which a module communicates with the outside world.
- **Nets** are interconnections between different components, we will mostly use the net type `wire`, which offers only basic connectivity.
  - Inputs are generally of type `wire`

```
module Mymod (X,Y,Z);  
  input wire X, Y;  
  output wire Z;  
  wire W;  
  assign Z = X & W;  
  assign W = ~(X); //order?  
endmodule
```



# “reg” variables and buses

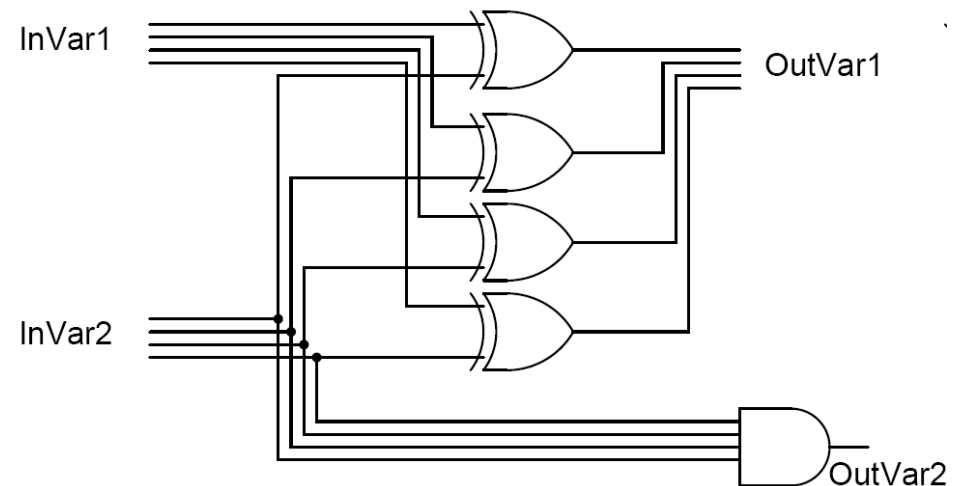
- **reg** is a particular type of variable
  - it's only *subtly different* from a **wire**
  - its value can only be assigned by a *procedural block* (we'll see this shortly) and only inside the module where it is declared
  - outputs can be of type **reg**
  - a **reg** is *not* a register or a flip-flop. It can be used in combinational circuits.
- both **reg** and **wire** types can be declared as vectors (buses)
  - examples

```
wire [3:0] mywire;      reg [15:0] mybigbus;
```
  - these are a 4-bit and a 16-bit bus of type **wire** and **reg**, respectively
  - note the use of numerals in the declaration. It seems strange at first but it makes sense if you think of how we write binary numbers in a positional number system framework.  
e.g. number N=6 in binary is N=110 and N[3]=1 N[2]=1 N[1]=0

## An example with buses

```
module Module_Name (           //note declaration style
    input wire [3:0] InVar1,
    input wire [3:0] InVar2,
    output wire [3:0] OutVar1,
    output wire OutVar2);

    assign OutVar1 = InVar1^InVar2; //bit-by-bit XOR
    assign OutVar2 = &InVar2; //peculiar use of &
endmodule
```



# Number literals

- Verilog has a unique way of representing numbers.
- **The format is  $n' B d d d \dots$ , where**
  - $n$  is the number of bits required to describe the number
  - $B$  is the base, chosen between  $b$  (binary),  $d$  (decimal),  $o$  (octal),  $h$  (hexadecimal)
  - $d d d d \dots$  is a string of digits in the base  $B$
- examples
  - Number 10 (decimal) is  $4'd10$  or  $4'b1010$  or  $4'ha$
  - Number 1000 (binary) is  $4'b1000$  or  $4'd8$  or  $4'h8$  or  $4'o10$
  - etc.



# The `assign` keyword

- **`assign`** describes an interconnection between wires.
  - NOTE: you cannot use `assign` to allocate a value to a `reg`
- it describes a ***continuous assignment***, not a sequence of commands
  - so the **order does not matter**
  - and **you must avoid feedback**
- **syntax:** `assign net_name = expression`
- in the expression you can use boolean operators and numeric literals (also some basic arithmetic)

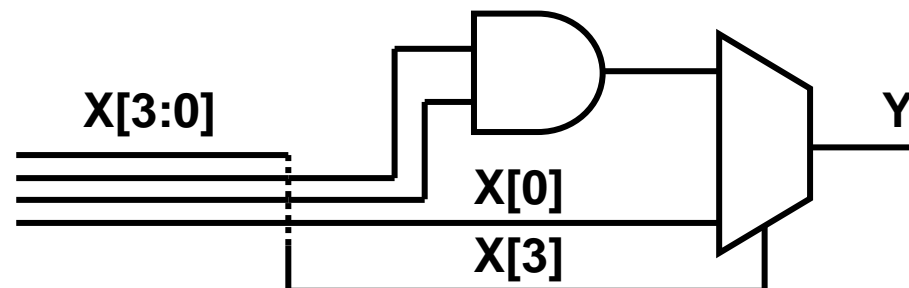
```
...  
wire X, Y;  
assign Y=X;  
assign Y=~X; /* this code has  
issues! */  
...
```

```
...  
wire K, W, X, Y, Z;  
assign W=4'd15;  
assign X=Y+Z; //examples  
assign K=Y&Z;  
...
```

# Advanced use of `assign`

- `assign` is straight forward in its meaning and usage
- There is one advanced usage feature: ***conditional assignment*** to be made, i.e. different values are assigned to the wire according to a ***select signal (SS)***.
- The `? ... :` operator is used.
- **Syntax:** `assign wirename = SS ? (value_if_true) : (value_if_false)`

```
module lovely_MUX (input wire [3:0] X, output wire Y);  
    assign Y = X[3] ? (X[2]&X[1]) : (X[0]);  
endmodule
```





## Behavioural designs (**always** block)

- So far we have mostly seen how to describe interconnections of gates and wires.
  - It is quicker than drawing but might not seem worth the effort yet
- Verilog also supports *behavioural designs*, i.e. a description of what a module *does*
- This is done through blocks of *procedural code*, i.e. lines which are processed sequentially (unlike what we saw previously).
- The keyword **always** introduces these blocks.
  - **NOTE:** this part of the Verilog language looks a lot like standard programming. I will stress again that you should *not* treat HDL as a programming language or your designs may fail.



# The `always` block

- **Syntax:** `always @ (sensitivity list)`
- The sensitivity list is a *list of signals*. The statements following `always` are executed **whenever one of the signals in the sensitivity list changes value**.
- Signals can be named explicitly or a special one-fits-all `always @ (*)` syntax can be used.
  - When `always @ (*)` is used, the sensitivity list automatically includes all signals found on the RHS of any assignment or as the argument of functions.

```
module module1 (  
  input wire [1:0] X,  
  output wire [1:0] Y);  
  
  always @(X[1]) Y[1:0]=X[1:0];  
endmodule
```

Y will only be updated if X[1] changes

```
module module2 (  
  input wire [1:0] X,  
  output wire [1:0] Y);  
  
  always @(*) Y[1:0]=X[1:0];  
endmodule
```

Y will be updated if either X[0] or X[1] change

## The `always` block (2)

- When you wish to include multiple lines of code following an `always` keyword, the `begin ... end` construct is used.

```
module new_module (  
  input wire [1:0] X,  
  output reg Y[1:0]);  
  
  always @(*) begin: mybeginend //the block name is optional  
    Y[1]~=X[1:0];  
    Y[0]=X[0]&X[1];  
  end  
endmodule
```

- Note: left hand terms in assignments inside an `always` block are **always of type reg**.



## The `always` block (3)

- The content of the `always` block is executed in zero “circuit time”
  - So you have to be careful not to create feedback loops.
  - example

```
always @( * ) begin
    X = Y | Z;
    Y = X ^ W;
end
```

- This makes general programming sense, but in Verilog it will create a circuit whose **simulation will loop indefinitely**.
  - This is because each time the block is executed the value of a RHS term is changed (actually, two: `X`, `Y`), causing the block to be executed again (over, and over, and over...).
  - The same would happen for a code line `assign X = ~X;`

## The `always` block (4)

- What makes the `always` block interesting is that we can describe quite complex behaviour, for example with `if` statements.

```
always @( * ) begin
    if (X==4'b1000);
        Y=16'habcd;
    else Y=16'd0;
end
```

- Note that when you use an `if` statement, all variables which are conditionally assigned a value should have all cases covered
  - hence the use of `else`
  - otherwise, what value should the circuit assign to Y if the condition is false?
    - many synthesisers will interpret this as a *requirement for a latch* to store the last value (inferred latches), this can lead to circuit behaviour problems and/or waste of hardware resources

## The `always` block (5)

- `if` statements are not the only procedural code options in Verilog. The following keywords might already mean something to you from previous programming experiences:
  - `case`
  - `while`
  - `for`
  - ...
- you will see examples and more description of these in your textbook, in tutorials, labs and upcoming lectures





## Example: a majority comparator

- An example of a combinational circuit with the **if** statement

```
module majority_comp (  
    input wire X, Y,  
    output reg W, Z);  
  
    always @(*) begin  
        if(X>Y) begin  
            W = 1'b1;  
            Z = 1'b0;  
        end  
        else begin  
            W = 1'b0;  
            Z = 1'b1;  
        end  
    end  
endmodule
```

- Note the use of **begin ... end** whenever multiple lines belong to a same block
- Note how all conditional cases are covered.

## Example: an 8-to-1 MUX

- An example of a combinational circuit with the **case** statement

```
module eight_to_1_MUX (  
  input wire [7:0] X,          //an 8-bit input X  
  input wire [2:0] sel,        //a 3-bit selector signal  
  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; //this should not occur
```

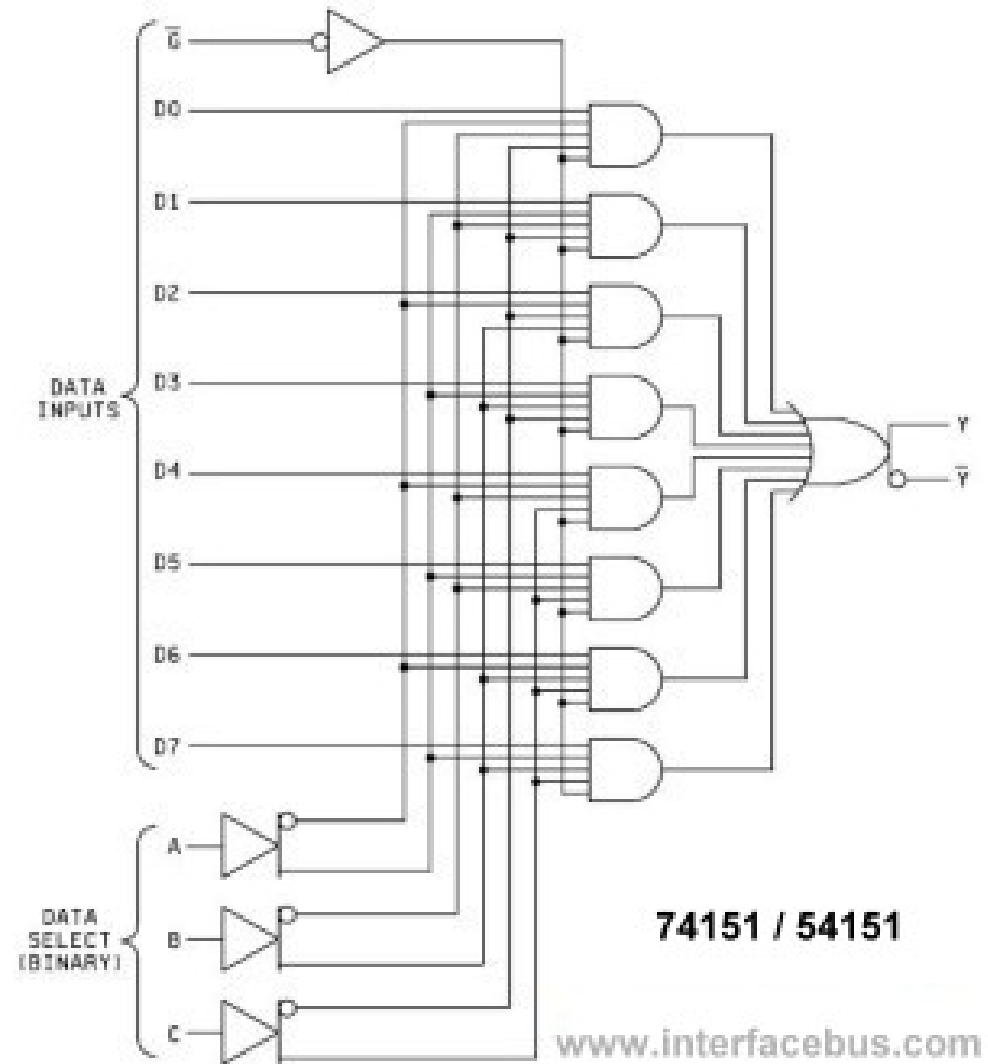
```
    endcase
```

```
  end
```

```
endmodule
```

Note that complete statements  
always end with a semicolon.  
(not required after “end” keywords)

- **The 8-to-1 MUX in schematics**  
(from <http://www.interfacebus.com/IC-Multiplexer-chips.html>)
- **Caution!** it is not exactly the same as the Verilog code
- You probably can see how drawing this is much more tedious than writing the code in the previous slide
- The schematic should still be *in your head*



# Blocking vs nonblocking assignments

- A peculiar concept in Verilog HDL.
- “**Blocking**” assignments have this name because the execution of procedural code stops until the assignment is made.
  - They are made **using the equal “=” sign** (inside `always` blocks)
  - Example

```
module blocking_assign(  
  input wire X[3:0],  
  output reg Z);  
  reg w;  
  always @(X) begin  
    w=X+1'b1;  
    Z=w+1'b1;  
  end  
endmodule
```

If we assume that *X* changes from 1 to 2, after the block has been executed we will have that *w* equals 3 and *Z* equals 4.

The result of earlier operations affects the result of the later ones.



## Blocking vs nonblocking assignments (2)

- In “**nonblocking**” assignments, instead, all values in consecutive evaluated but assigned “at once” at the end of the always block.
  - They are made **using the “<=” sign**
  - The “old” value remains current until the whole block has been evaluated
  - Used in sequential circuits to describe assignment events that are triggered by a transition of a signal in the sensitivity list (e.g. clock in flip-flops).

```
module nonblock_assign(  
  input wire A,B,clk,  
  output reg C,D);  
  always @(posedge clk) begin  
    C<=A&B;  
    D<=C&B;  
  end  
endmodule
```

In this example the value assigned to D does not depend on the value of C calculated in the first line of the **always** block.

It depends on the value of C and B *just before* the clock rising edge occurred.

Note the **posedge** keyword.

# Blocking vs nonblocking assignments (4)

- Timing diagram of blocking vs nonblocking

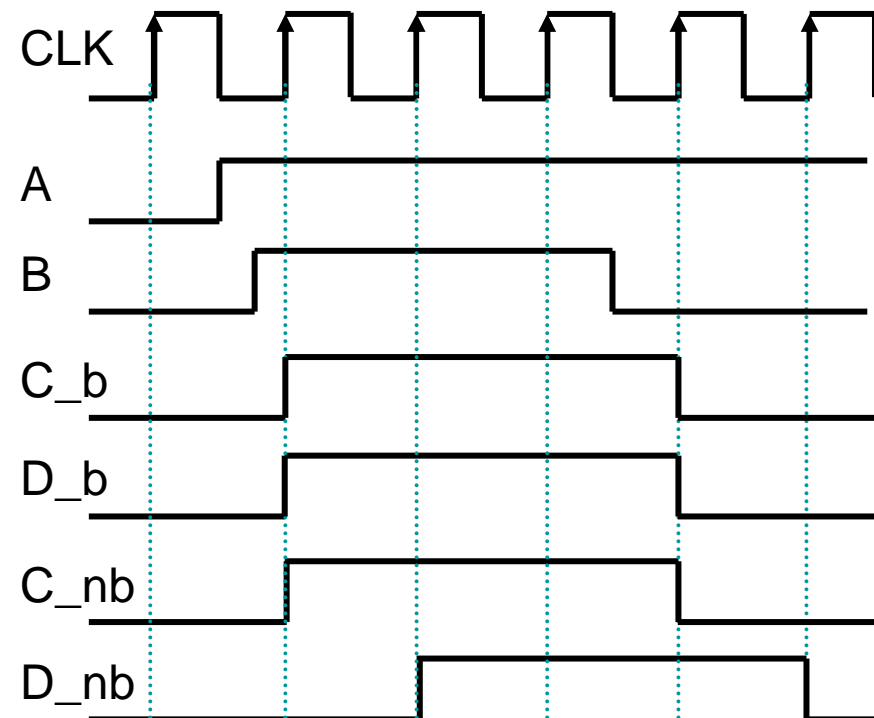
```

module nonblock_assign(
 A,B,clk,
	output reg C_b,D_b,C_nb,D_nb);

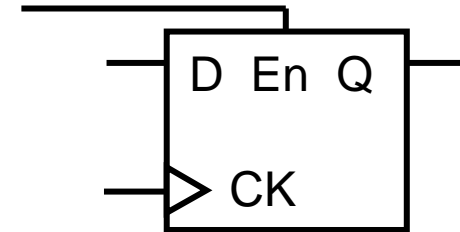
always @(posedge clk) begin
    C_b=A&B;
    D_b=C&B;
end

always @(posedge clk) begin
    C_nb<=A&B;
    D_nb<=C&B;
end
endmodule

```



## A D flip-flop example



With enable (En)

```
module Dff_En(  
  input wire Clk,  
  input wire D,  
  output reg Q);  
  always @(posedge Clk)  
    if(En)  
      Q <= D;  
endmodule
```

Without enable

```
module Dff_noEn(  
  input wire Clk,  
  input wire D,  
  output reg Q);  
  always @(posedge Clk)  
    Q <= D;  
endmodule
```

## Blocking vs nonblocking assignments (3)

- Getting your head around blocking and nonblocking assignments might be a bit tricky at first.
- We will discuss them in more detail in the next lecture.
- The basic rule of thumb is
  - Use **blocking assignments** for **combinational logic**
  - Use **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



# Summing up

- We have introduced **some of the key features of Verilog**.
  - Modules, i/o declarations
  - **wire** and **reg** variables
  - notations for buses and numeric literals
  - continuous assignments with **assign**
  - “procedural” **always** blocks
  - blocking and nonblocking assignments (briefly)
- There are some more advanced aspects and we will see those in the next lecture.
- I encourage you to look through the many examples in the textbook to improve your understanding.
  - Exposing yourselves to a variety of code examples is a good way to get your head around the new concepts.
  - After next week’s lab you will be able to code and simulate your own designs.