CSE 460: VLSI Design

Lab Experiment 3: Blocking and Non-blocking Statements in Verilog

BRAC UNIVERSITY

Inspiring Excellence

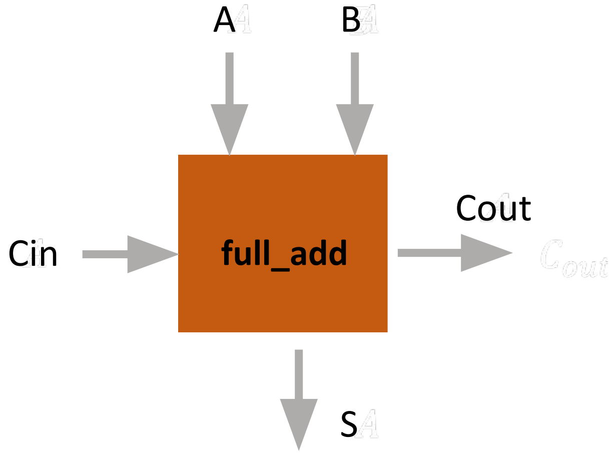
Concurrent Statements

In any HDL, concurrent statement means the code may include a number of statements and each represent a part of the circuit.

What concurrent means:

Concurrent is used because the statements are considered in parallel and the ordering of statements in the code doesn't matter.

```
module full_add(S, Cout, A, B, Cin);
// This module implements a 1-bit full adder
   input A, B, Cin;
   output S, Cout;

   assign S = A ^ B ^ Cin;
   assign Cout = (A & B) | (Cin & (A ^ B));
endmodule
```



Procedural Statements

- These statements are inside **always @()** block.
- ✓ **The If-else statement:**

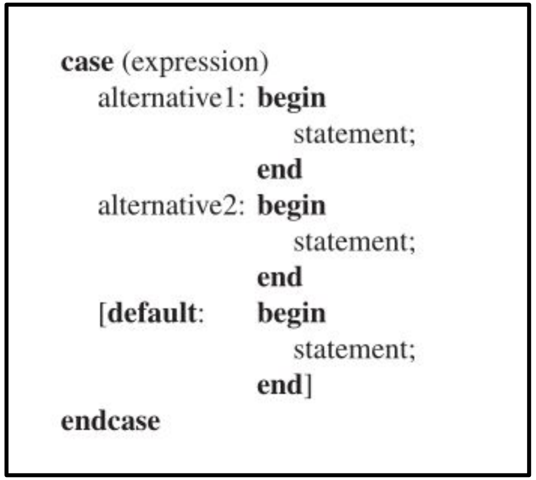
- If expression1 is True then the statement1 is evaluated.
- If not , then the compiler will consider other expressions.
- The else if and else clauses are optional.
- When multiple statements are involved, they have to be included inside a begin-end block

```
always @(*)
    if(expression1)
    begin
     statement1;
    end
    else if(expression2)
    begin
     statement2;
    end
    else
    begin
        statement3;
    end
end
```

Procedural Statements

✓ **The case statement**

- The bits in *expression* are called the *controlling expression*.
- *Controlling expression* are checked for a match with each alternative.
- The first successful match causes the associated statements to be evaluated.
- Default case evaluates only when no other alternative matches.



```
case (expression)
    alternative1: begin
                     statement;
                  end
    alternative2: begin
                     statement;
                  end
    [default:     begin
                     statement;
                  end]
endcase
```

wire vs reg

- <u>Nets</u>

  - Nets represent connections between hardware elements. Nets are continuously driven by the outputs of the devices they are connected to.

  - Nets are declared with the keyword ***wire****.* A net is assigned the value *z* by default.

- <u>Registers</u>

  - In verilog register means a variable that can hold a value. Unlike net, a register doesn't need a driver.

  - Registers are declared with the keyword ***reg****.* The default value of a *reg* data type is *x*.

```
wire a, b; // wire declaration
reg clock; // register declaration
```

wire vs reg

When to use which?

- If a signal needs to be assigned inside an ***always block***, it must be declared as a ***reg***.
- If a signal is assigned using a ***continuous*** assignment statement, it must be declared as a ***wire***.
- By **default**, module input and output ports are ***wires***; if any output ports are assigned in an *always* block, they must be explicitly declared as *reg*: ***output reg <signal name>***

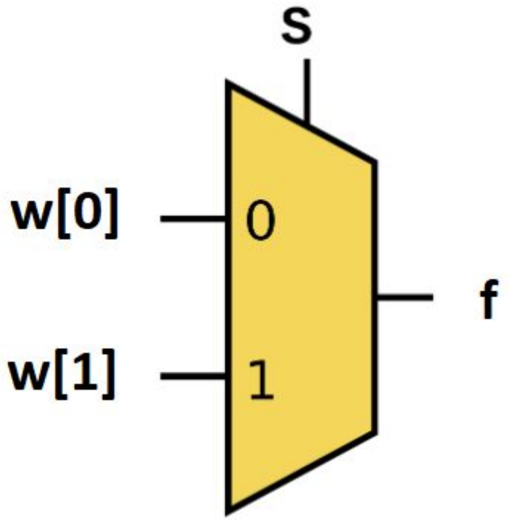
How to know if a net represents a register or a wire?

- A *wire* net always represents a combinational link
- A *reg* net represents a wire if it is assigned in an ***always @ (*)* block**
- A *reg* net represents a register if it is assigned in an ***always @ (posedge/negedge clock)*** block

Procedural Statements

- ✓ 2 to 1 Mux:

When s=0, f = w[0]
When s=1, f = w[1]

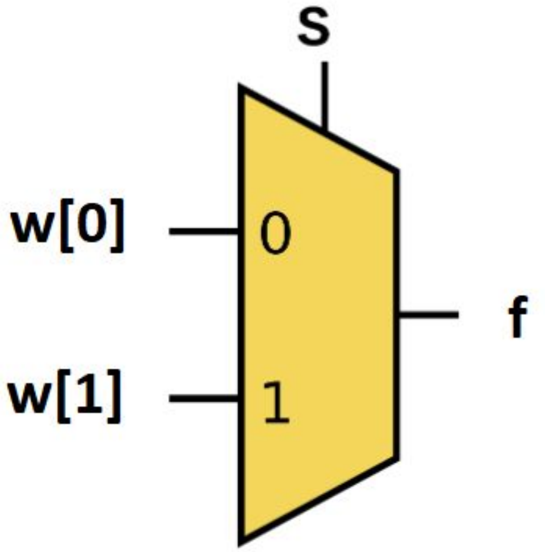


BRAC UNIVERSITY
Inspiring Excellence

Procedural Statements

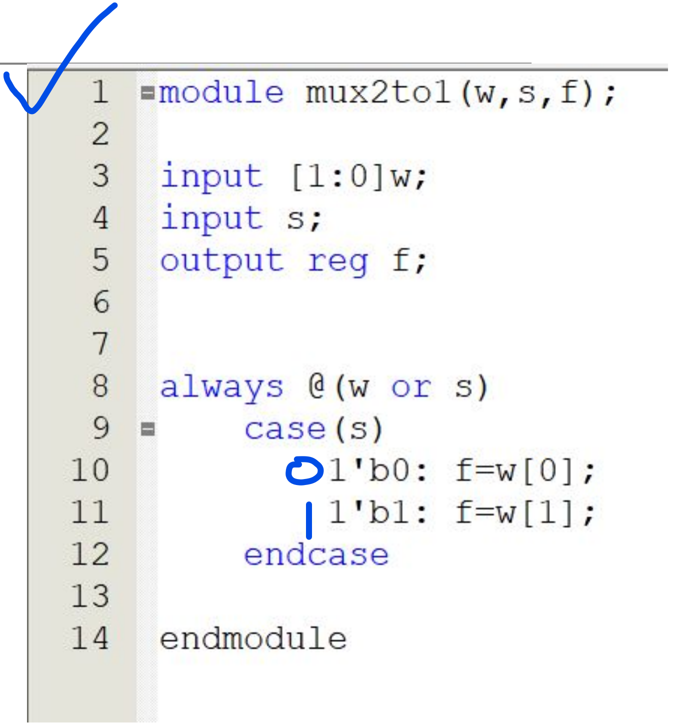
The If-else statement:

```
module mux2to1(w, S, f);
   input S;
   input [1:0]w;
   output reg f;

   always @(w, S)  // always @(*)
   begin
        if(S == 0)
           f = w[0];
        else
           f = w[1];
   end
endmodule
```



Procedural Statements

- This is the code of 2 to 1 Mux using case statement.
- The mux can have two possible outputs because "*s*" is only 1 bit.
- Which is why the case statement has two alternatives.
- We could have included a default case because "*s*" can also have values of "*x*" and "*z*". But we will learn about them soon.
- We can also use "*1*" as alternative instead of "*1'b0*".
- If a statement in an alternative has multiple line it must be included in Begin-end block.

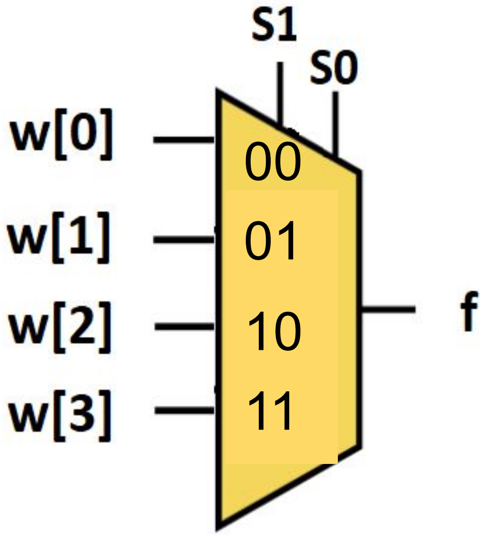


```
module mux2to1(w,s,f);

input [1:0]w;
input s;
output reg f;


always @(w or s)
      case(s)
         1'b0: f=w[0];
         1'b1: f=w[1];
      endcase

endmodule
```

Procedural Statements

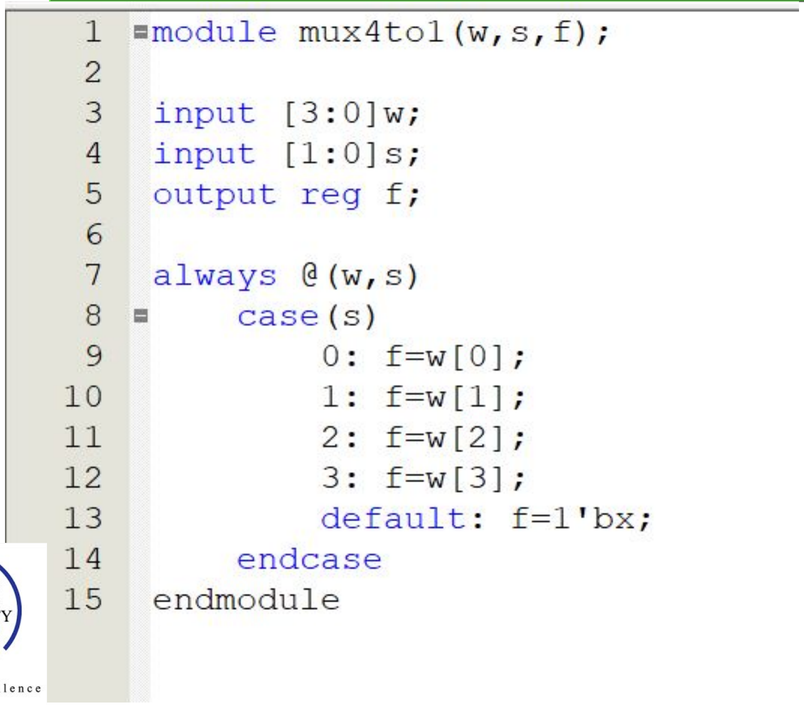
- ✓ 4 to 1 Mux:

When s=00, f = w[0]
When s=01, f = w[1]
When s=10, f = w[2]
When s=11, f = w[3]



BRAC UNIVERSITY
Inspiring Excellence

Procedural Statements



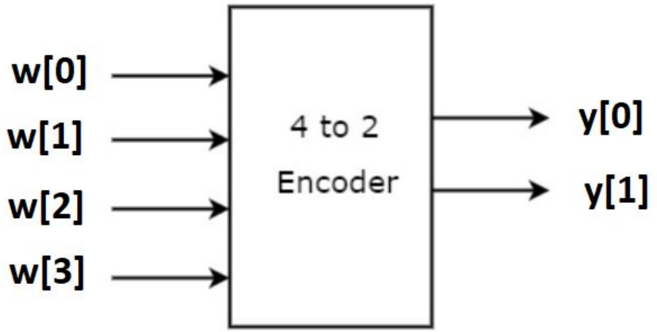
```verilog
module mux4to1(w,s,f);

 input [3:0]w;
 input [1:0]s;
 output reg f;

 always @(w,s)
     case(s)
         0:  f=w[0];
         1:  f=w[1];
         2:  f=w[2];
         3:  f=w[3];
         default: f=1'bx;
     endcase
 endmodule
```

S	00	01	10	11
f	W[0]	W[1]	W[2]	W[3]

Procedural Statements

- ✓ 4 to 2 encoder (one-hot encoding):

Inputs				Outputs	
w[3]	w[2]	w[1]	w[0]	y[1]	y[0]
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1



Procedural Statements

- ✓ 4 to 2 priority encoder:

When multiple input lines are active high at the same time, the output is generated by the the input with the highest priority.

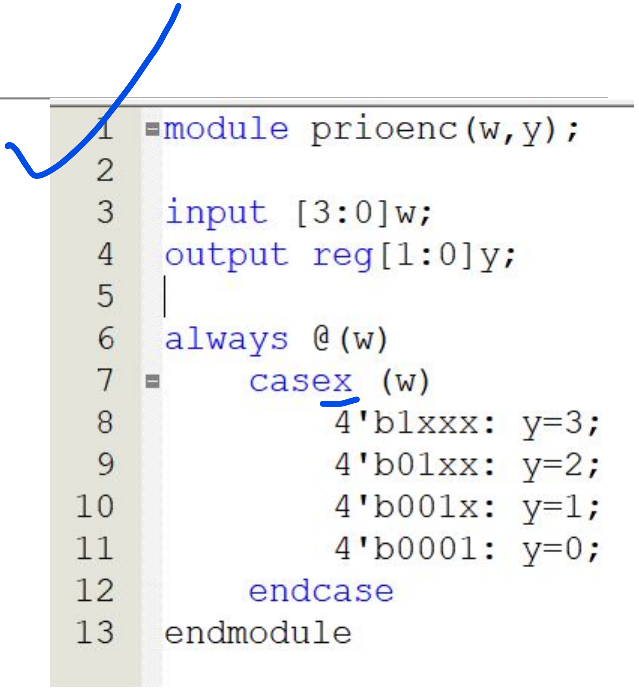
For 3>2>1>0 priority:

w[3]	w[2]	w[1]	w[0]	y[1]	y[0]
0	0	0	0	X	X
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

BRAC UNIVERSITY
Inspiring Excellence

Procedural Statements

- In the "*case*" statement , controlling bits can also have value of "*x*" and "*z*".
- The values of "*x*" and "*z*" are also checked for exact match with the same values in the controlling expressions.
- The "*casex*" statement treats both "*x*" and "*z*" as don't cares.
- That means when they are present as input , code won't check for their alternatives.
- In the right there is a Verilog code of priority encoder with 4 bit input "*w*" and output "y".
- The first alternative "1xxx" specifies that if w[3] has the value of 1 , then the other inputs are treated as don't cares and so the output is set to "*y=3*"



```
module prioenc(w,y);

input [3:0]w;
output reg[1:0]y;

always @(w)
      casex (w)
            4'b1xxx: y=3;
            4'b01xx: y=2;
            4'b001x: y=1;
            4'b0001: y=0;
      endcase
endmodule
```

Procedural Assignment Statements

- A value is assigned to a variable with a *procedural assignment statement*.
- There are two kinds of assignment statements.
  1. Blocking assignments
  2. Non-blocking assignments.

  - Blocking assignments are denoted by the "=" symbol.
  - Blocking means that first the assignment statement completes and updates it's left-hand side first.
  - This updated left-hand side value is then used for evaluation of subsequent statements.

```
S = X + Y;
p = S[0];
```

Procedural Assignment Statements

- At simulation time t_i the statements are evaluated in order.
- The first statement sets “*S*” to have the summation of current values of “*X*” and “*Y*” .
- Then the second statement sets “*p*” according to this current value of “*S*”

```
S = X + Y;
p = S[0];
```

Procedural Assignment Statements

- 2nd types of assignment statement is non-blocking assignments.
- Non-blocking assignments use the “<=“ symbol.
- At simulation time t_i the statements still are evaluated in order but they both use the value of the variables that exist at the start of simulation time.
- The first statement assigns a new value to “*S*” based on the current value of “*X*” and “*Y*” .
- But “*S*” is not actually changed to this value until all statements in the always block have been evaluated.
- For this , the value of “*p*” at time t_i is based on the value of “S” at time t_{i-1}.

```
S <= X + Y;
p <= S[0];
```

Blocking vs. Non-blocking assignments

Blocking assignment: evaluation and assignment are immediate

```
always @ (a or b or c)
begin
  x = a | b;
  y = a ^ b ^ c;
  z = b & ~c;
end
```

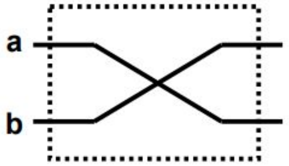
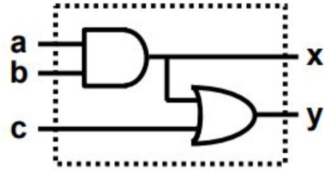
1. Evaluate *a* | *b*, assign result to *x*
2. Evaluate *a*^*b*^*c*, assign result to *y*
3. Evaluate *b*&(~*c*), assign result to *z*

Nonblocking assignment: all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

```
always @ (a or b or c)
begin
  x <= a | b;
  y <= a ^ b ^ c;
  z <= b & ~c;
end
```

1. Evaluate *a* | *b* but defer assignment of *x*
2. Evaluate *a*^*b*^*c* but defer assignment of *y*
3. Evaluate *b*&(~*c*) but defer assignment of *z*
4. Assign *x*, *y*, and *z* with their new values

Why we need them?

	Swapping a,b 	
Blocking: Evaluation and assignment are immediate	✗ a = b b = a	✓ x = a & b y = x | c
Non-Blocking: Assignment is postponed until all r.h.s. evaluations are done	✓ a <= b b <= a	✗ x <= a & b y <= x | c
When to use (inside always block)	**Sequential Circuits**	**Combinational Circuits**

Combinational vs. Sequential

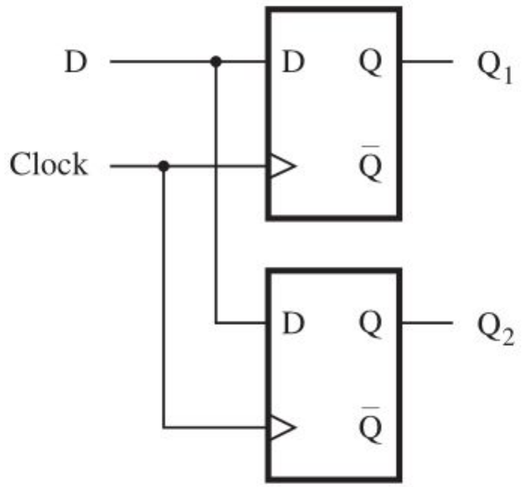
A combinational circuit is one in which the output is independent of time and solely depends on the present input. Example: Encoder, Decoder, Multiplexer, Demultiplexer

A sequential circuit is one in which the output is dependent not only on the current input but also on the past ones. Examples: Flip-flops, counters.

Procedural Assignment Statements

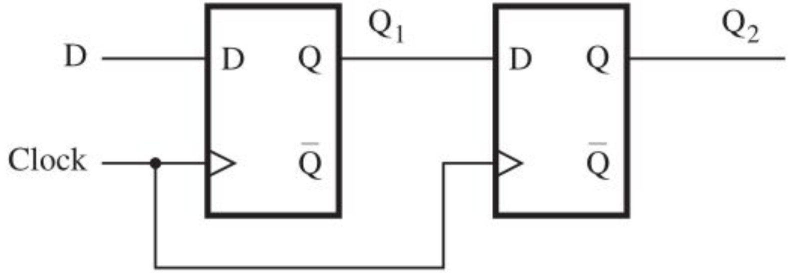
✓

```
module example7_3 (D, Clock, Q1, Q2);
    input D, Clock;
    output Q1, Q2;
    reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 = D;
        Q2 = Q1;
    end

endmodule
```

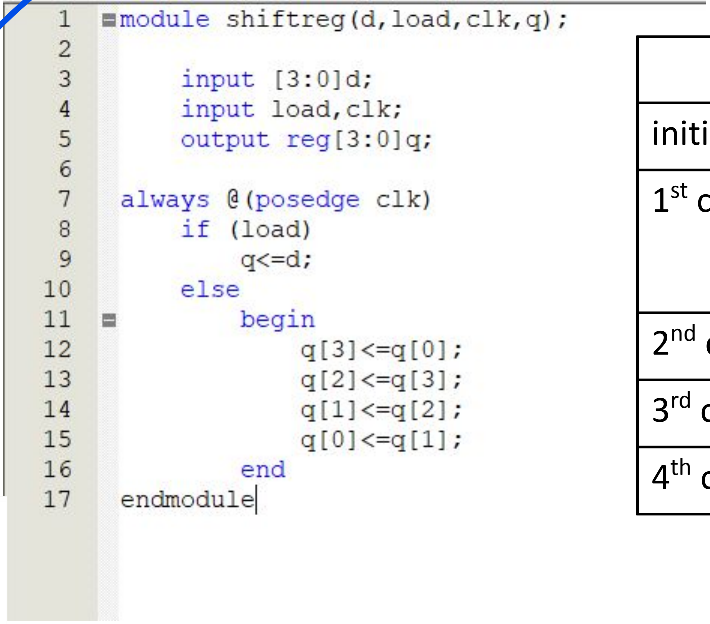


Procedural Assignment Statements

✓
```
module example7_4 (D, Clock, Q1, Q2);
    input D, Clock;
    output Q1, Q2;
    reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 <= D;
        Q2 <= Q1;
    end

endmodule
```



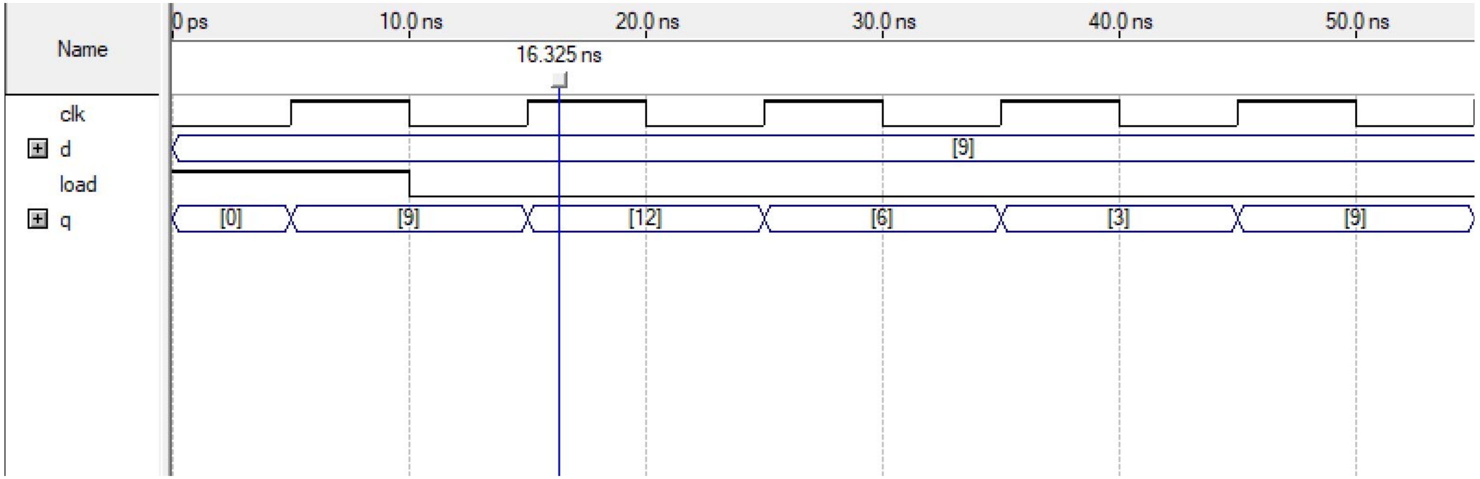
Procedural Assignment Statements



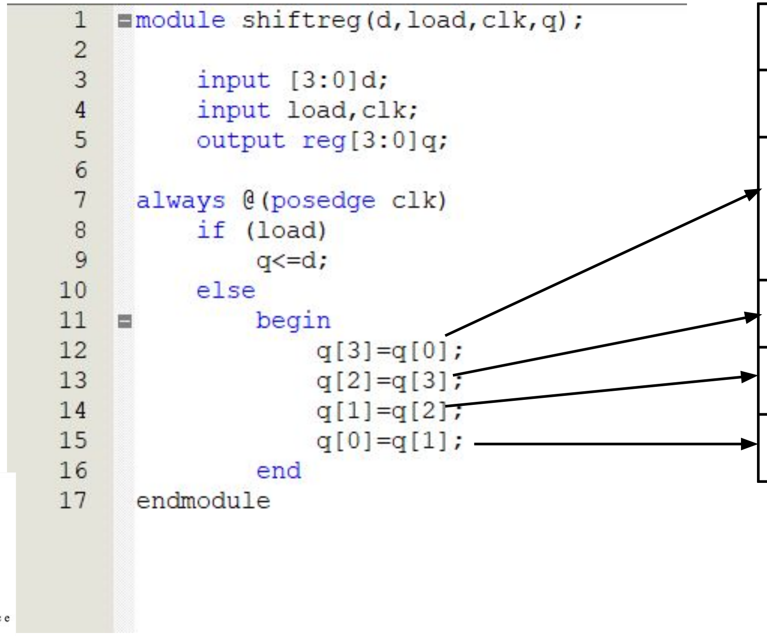
```
module shiftreg(d,load,clk,q);

    input [3:0]d;
    input load,clk;
    output reg[3:0]q;

always @(posedge clk)
    if (load)
        q<=d;
    else
        begin
            q[3]<=q[0];
            q[2]<=q[3];
            q[1]<=q[2];
            q[0]<=q[1];
        end
endmodule
```

	q[3]	q[2]	q[1]	q[0]
initial	1	0	0	1
1st cycle	1	1	0	0
2nd cycle	0	1	1	0
3rd cycle	0	0	1	1
4th cycle	1	0	0	1

Procedural Assignment Statements
(Non-Blocking)



Procedural Assignment Statements

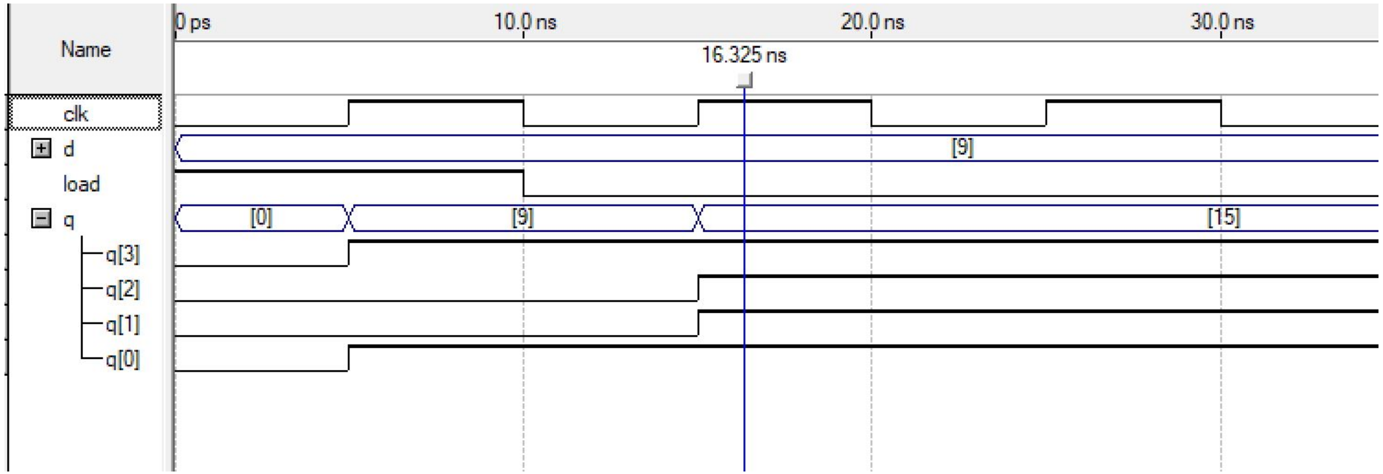


```
module shiftreg(d,load,clk,q);

    input [3:0]d;
    input load,clk;
    output reg[3:0]q;

always @(posedge clk)
    if (load)
        q<=d;
    else
        begin
            q[3]=q[0];
            q[2]=q[3];
            q[1]=q[2];
            q[0]=q[1];
        end
endmodule
```

	q[3]	q[2]	q[1]	q[0]
initial	1	0	0	1
1st statement	1	0	0	1
2nd statement	1	1	0	1
3rd statement	1	1	1	1
4th statement	1	1	1	1

All of these happened within one cycle!

Procedural Assignment Statements
(Blocking)



Thank you!