Digital System Design ELEC373/473

UNIVERSITY OF LIVERPOOL

Dataflow or RTL Modelling

Prof J.S. Smith
Room A515;
E-mail: j.s.smith@liv.ac.uk

Module Styles

- Modules can be specified in different ways
  1) **Structural**: connect primitives and modules
  2) **RTL** (Register Transfer Level) :
     - use continuous assignments
  3) **Behavioral**: use **initial** and **always** blocks
     - Note that "initial" is primarily for simulation rather than for synthesis.
- A single module can use more than one method.

Learning Objectives

- Describe the continuous assignment (assign) statement.
- Define expressions, operators and operands.
- List operators for arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation and conditional operations.
- Use dataflow constructs to model practical combinational and sequential digital circuits in Verilog.
- Timing constraints in edge triggered FFs.

Nets and Variables

- The net data types represent physical connections between structural entities, such as gates.
- Nets get the output value of their drivers.
- Nets are declared primarily with the keyword *wire*.
- Net is not a keyword but represents a class of data types such as *wire*, *wor*, *wand*, *trior* and *triand*.
- A *variable* is an abstraction of a data storage element. A variable shall store a value from one assignment to the next.

Nets and Variables

- A variable can have reg, time and integer data types.
- Do not confuse the term *register* in Verilog with hardware registers built from edge triggered flip-flops.
- In Verilog, the term *register* merely means a variable that can hold a value until another value is placed onto it.
- Registers can also be declared as signed variables.
  - reg signed [63:0] m; // 64-bit signed value
  - integer i ;   // 32-bit signed value as default

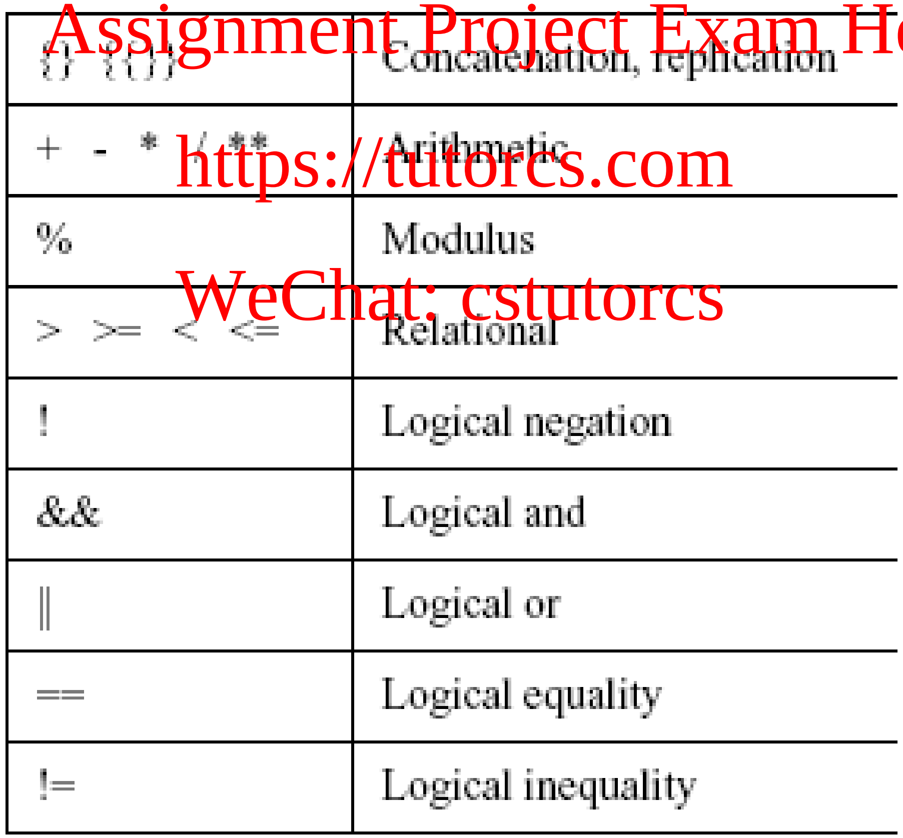
Arrays and Memories

- An array declaration for a **net** or a **variable** declares an element type which is either scalar or vector.
  - reg z[11:0];   //scalar reg
  - wire [0:7] y[5:0];   //8-bit-wide vector wire of 6 elements
  - reg [31:0] x [127:0]; //32-bit-wide 128 element vector register
  - It is important not to confuse arrays with vectors.
  - x[3] = 32'h4532 ; // set 4th element of the x array
  - y[6] = 25;   // Illegal syntax. The index is out of range
- Memories are modelled as a one-dimensional array of registers.

```
reg [7:0] membyte [0:1023]; // 8-bit memory
reg mem1bit [0:1023];          // 1-bit memory
```

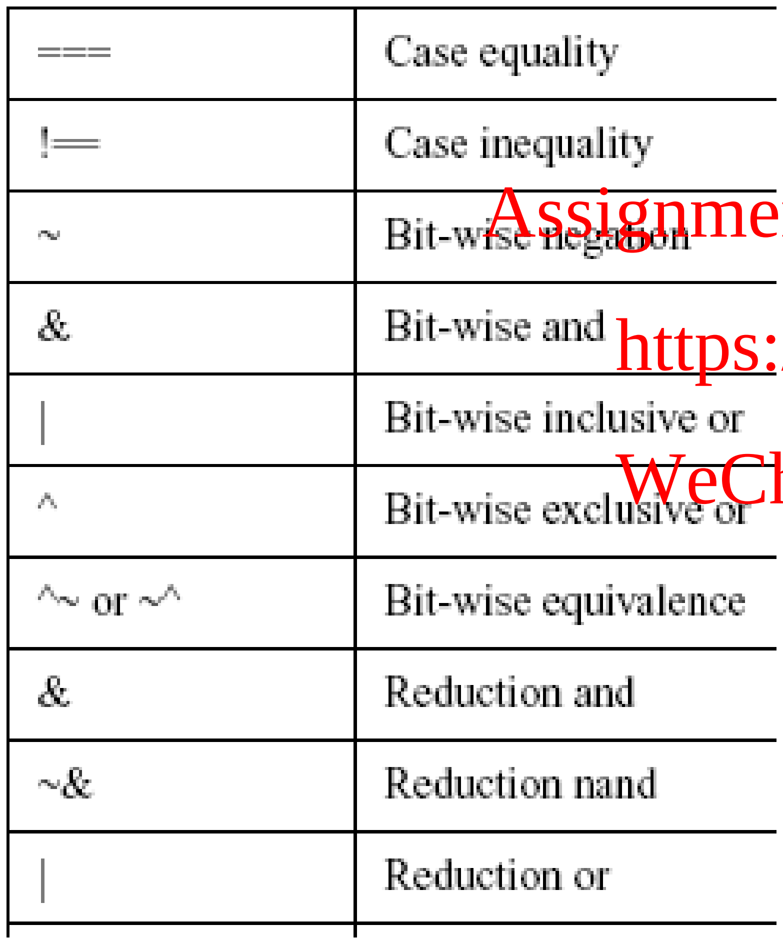
Operators in the Verilog HDL(1)

The symbols for the Verilog HDL operators are similar to those in the C programming language.

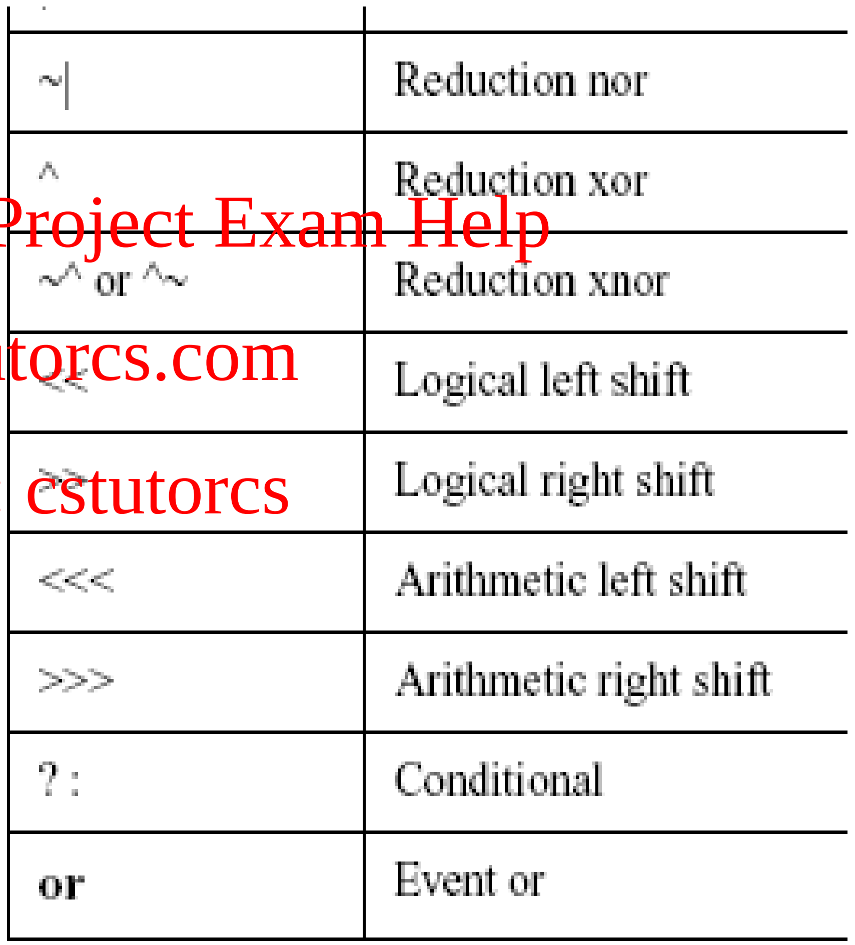


{} {{}}	Concatenation, replication
+ - * / **	Arithmetic
%	Modulus
> >= < <=	Relational
!	Logical negation
&&	Logical and
||	Logical or
==	Logical equality
!=	Logical inequality

Operators in the Verilog HDL (2)



===	Case equality
!==	Case inequality
~	Bit-wise negation
&	Bit-wise and
|	Bit-wise inclusive or
^	Bit-wise exclusive or
^~ or ~^	Bit-wise equivalence
&	Reduction and
~&	Reduction nand
|	Reduction or



~|	Reduction nor
^	Reduction xor
~^ or ^~	Reduction xnor
<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift
? :	Conditional
or	Event or

Reduction Operators

- Reduction operators perform a bitwise operation on a vector operand and yield a 1-bit result.
- Reduction operators take only one operand.
- Reduction operators are *and* (&), *nand* (~&), *or* (|), *nor* (~|), *xor* (^), and xnor (~^, ^~).

```
          //  X= 4'b1010;
&X        // Equivalent to                    Results in 1'b
|X        // Equivalent to                    Results in 1'b
^X        // Equivalent to                    Results in 1'b
```

Reduction Operators

- Reduction operators perform a bitwise operation on a vector operand and yield a 1-bit result.
- Reduction operators take only one operand.
- Reduction operators are *and* (&), *nand* (~&), *or* (|), *nor* (~|), *xor* (^), and *xnor* (~^, ^~).

```
         //  X= 4'b1010;
&X       // Equivalent to 1 & 0 & 1 & 0.  Results in 1'b0.
|X       // Equivalent to 1 | 0 | 1 | 0.       Results in 1'b1.
^X       // Equivalent to 1 ^ 0 ^ 1 ^ 0.      Results in 1'b0.
```

Shift Operators

- Logical shift operators shift a vector operand to the right or to the left by a specified number of bits.
- In logical shifts the vacant bits are filled with 0.
- Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

```
X=4'b1100;
Y = X >> 1          // Y is 4'b
Y = X << 2          // Y is 4'b
integer a, b, c;    // signed data types
a = -10;            // a = 11…10110
b = a >>> 3;        // b =                    ,  b = -    decimal
c = a <<< 2;        // b =                    ,  b = -   decimal
```

Shift Operators

- Logical shift operators shift a vector operand to the right or to the left by a specified number of bits.
- In logical shifts the vacant bits are filled with 0.
- Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

```
X=4’b1100;
Y = X >> 1          // Y is 4’b0110
Y = X << 2          // Y is 4’b0000
integer a, b, c;    // signed data types
a = -10;            // a = 11…10110
b = a >>> 3;        // b = 11 …11110  ,  b = -2 decimal
c = a <<< 2;        // b = 11 …1011000  ,  b = -40 decimal
```

Concatenation Operator

- Concatenation operator ({ }) provides a mechanism to append multiple operands.
- The operands must be sized.

```
// A= 1'b1, B = 2'b00 , C = 2'b10, D = 3'b110
Y = { B, C};                    // Y is
Y = { A,  B, C, D};             // Y is
Y = { B[0], D[2], 2'b11};       // Y is
```

Concatenation Operator

- Concatenation operator ({ }) provides a mechanism to append multiple operands.
- The operands must be sized.

```
// A= 1'b1, B = 2'b00 , C = 2'b10, D = 3'b110
Y = { B, C};                  // Y is 4'b0010
Y = { A,  B, C, D};           // Y is 8'b10010110
Y = { B[0], D[2], 2'b11};     // Y is 4'b0111
```

Replication Operator

- Repetitive concatenation of the same operand can be expressed by using a replication constant.

```
reg A;
reg [1:0]  B, C;
reg [2:0]  D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y = { 4{A} };            // Results in Y =
Y = { 4{A} , 2{B} };     // Results in Y =
Y = { 4{A}, 2{B}, C };// Results in Y =
```

Replication Operator

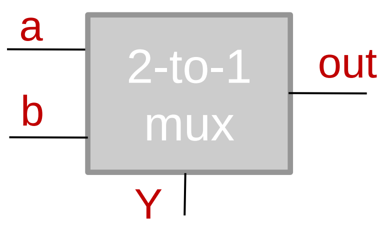
- Repetitive concatenation of the same operand can be expressed by using a replication constant.

```
reg A;
reg [1:0]  B, C;
reg [2:0]  D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y = { 4{A} };              // Results in Y = 4'b1111
Y = { 4{A} , 2{B} };       // Results in Y = 8'b11110000
Y = { 4{A}, 2{B}, C };     // Results in Y = 10'b1111000010
```

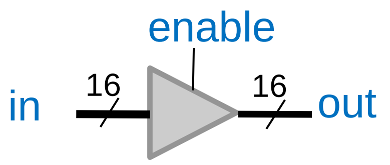
Conditional Operator

- Usage: *condition expr* ? *true_expr* : *false_expr* ;
- The conditional expression is first evaluated.
  - If the result is true the true expression is evaluated.
  - If the result is false the false expression is evaluated.

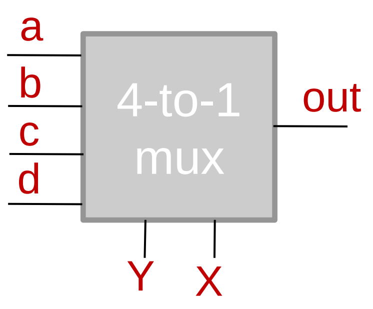
```
// model functionality of a 2-to-1 mux
assign out = Y ? b : a;
```



```
// model functionality of a tri state buffer
assign out = enable ? in : 16'bz;
```



```
// nested conditional operators. 4-to-1 mux
assign out = Y ? ( X ? d : c) : ( X ? b : a);
```



Continuous Assignment

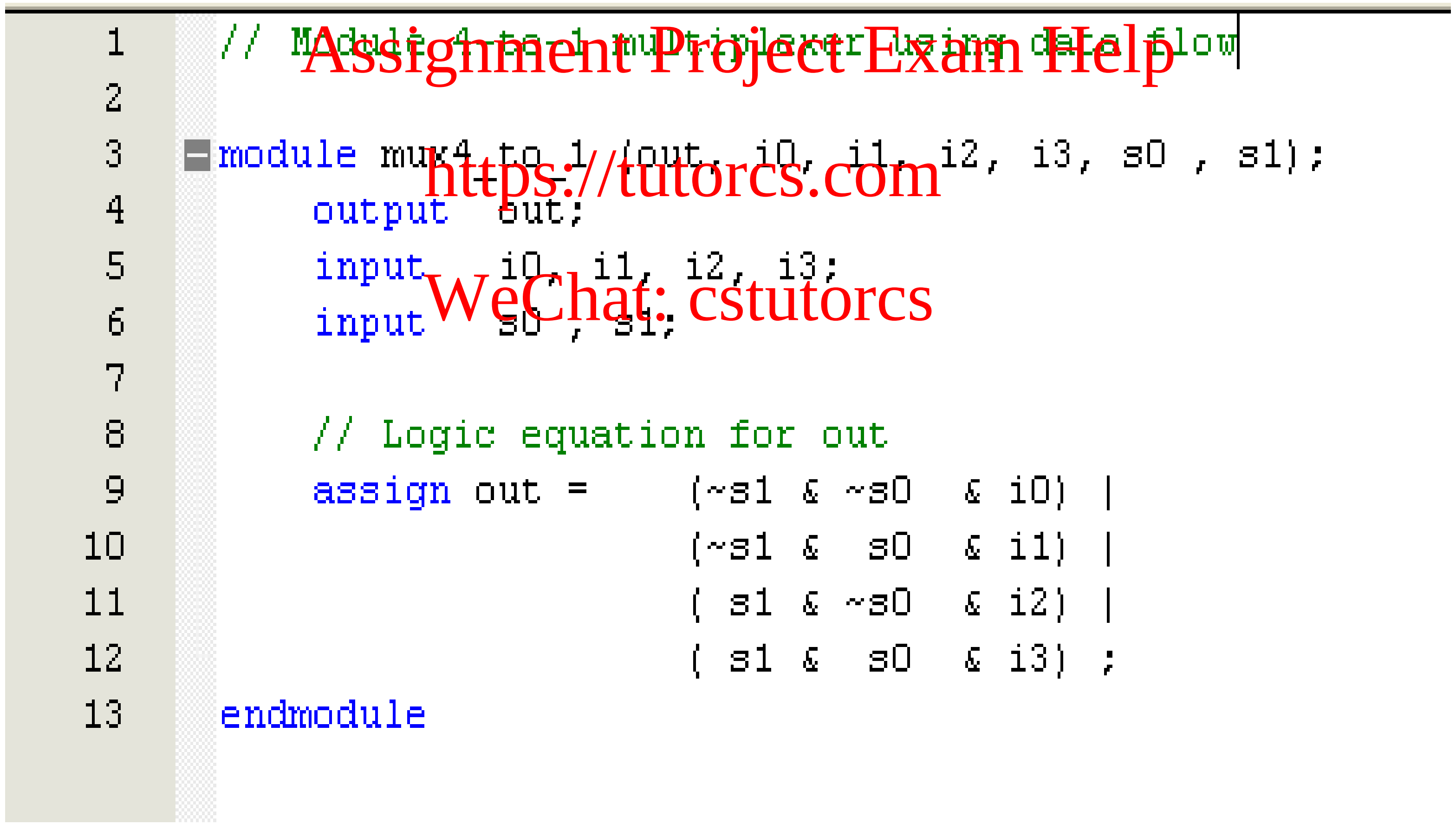
- The LHS of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets.
- The operands on the RHS can be registers or nets or function calls.
- Continuous assignments are always active. The assignment expression is evaluated as soon as one of the RHS operands changes.
- assign out = i1 & i2 ;
- assign addr[15:0] = addr1[15:0] ^ addr2[15:0] ;
- assign {c_out , sum[3:0]} = a[3:0] + b[3:0] + c_in ;

XOR

Concatenation

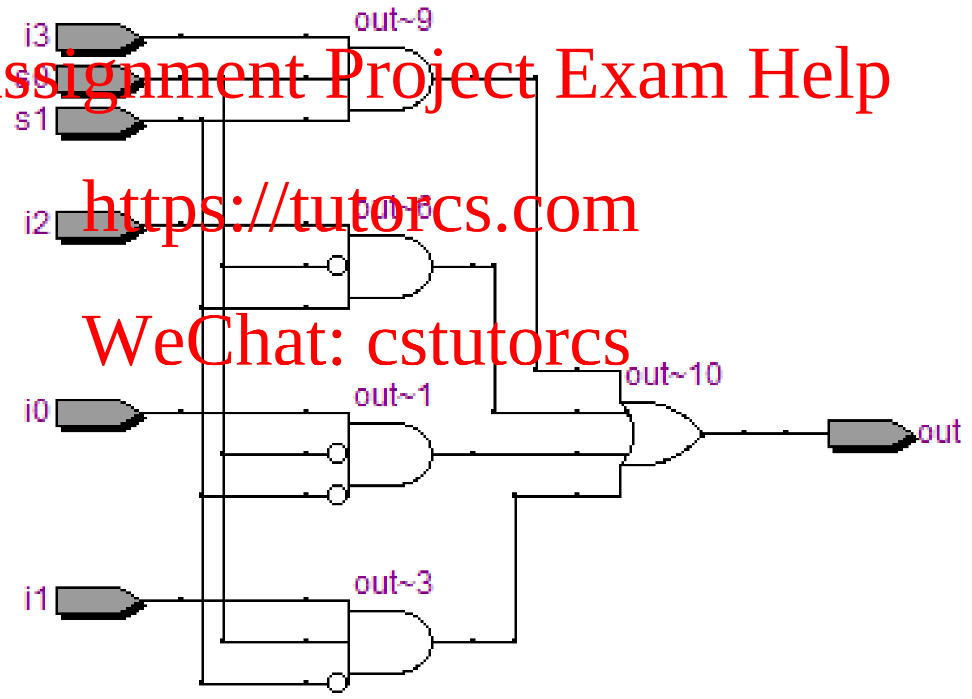
4-to-1 Multiplexer: Method 1: Logic Equation

- In this method operators have been used instead of individual gate instantiations.

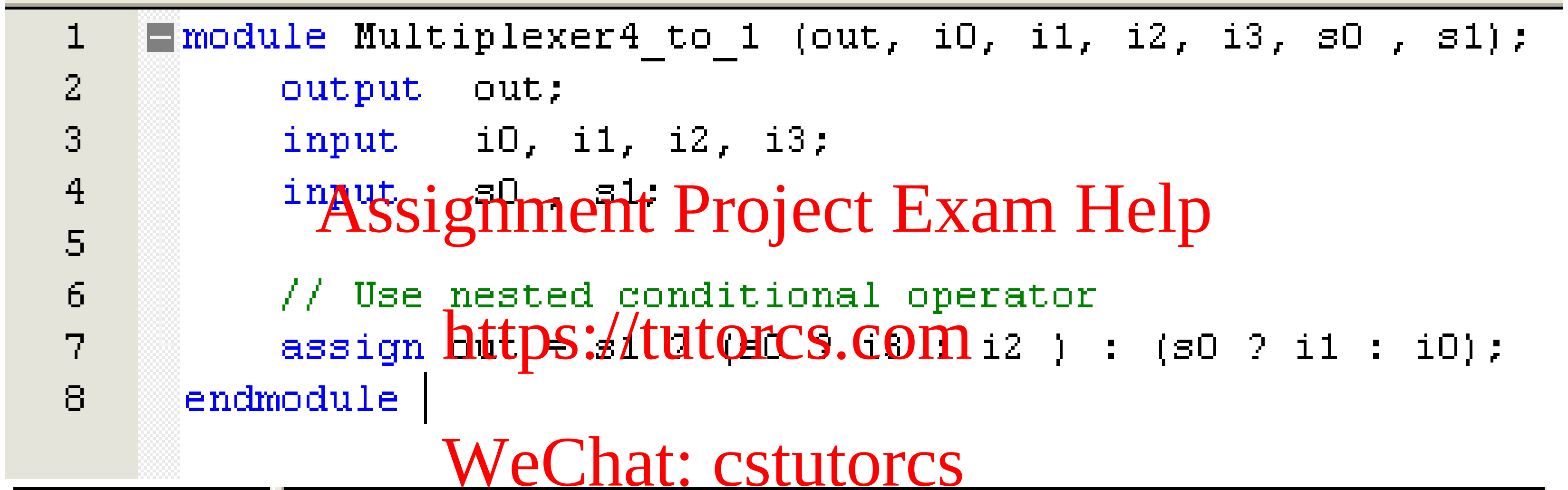


```
// Module 4-to-1 multiplexer using data flow

module mux4_to_1 (out, i0, i1, i2, i3, s0 , s1);
    output  out;
    input   i0, i1, i2, i3;
    input   s0 , s1;

    // Logic equation for out
    assign out =    (~s1 & ~s0  & i0) |
                    (~s1 &  s0  & i1) |
                    ( s1 & ~s0  & i2) |
                    ( s1 &  s0  & i3) ;
endmodule
```

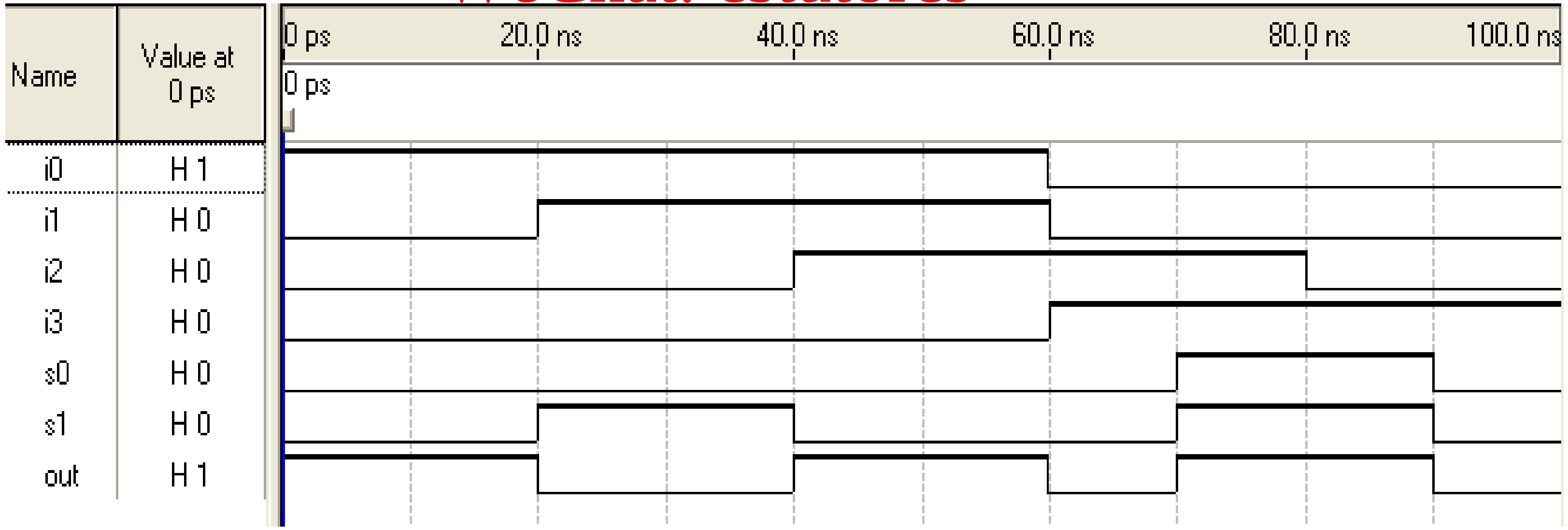
Synthesised circuit



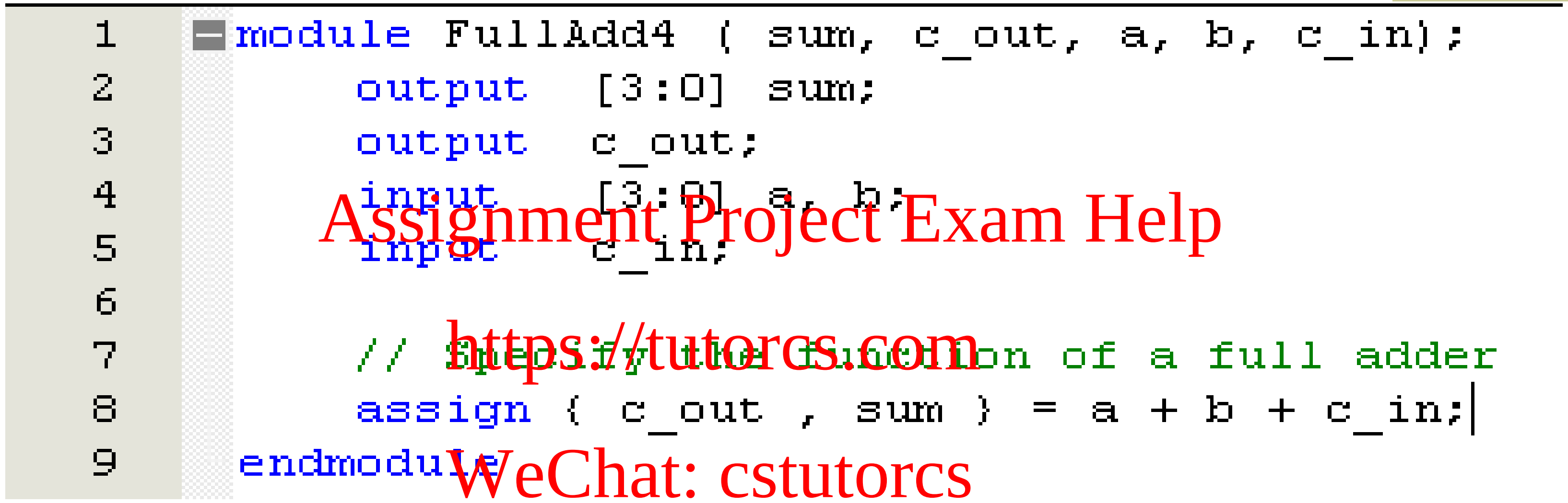
4-to-1 Multiplexer: Method 2: Conditional Operator



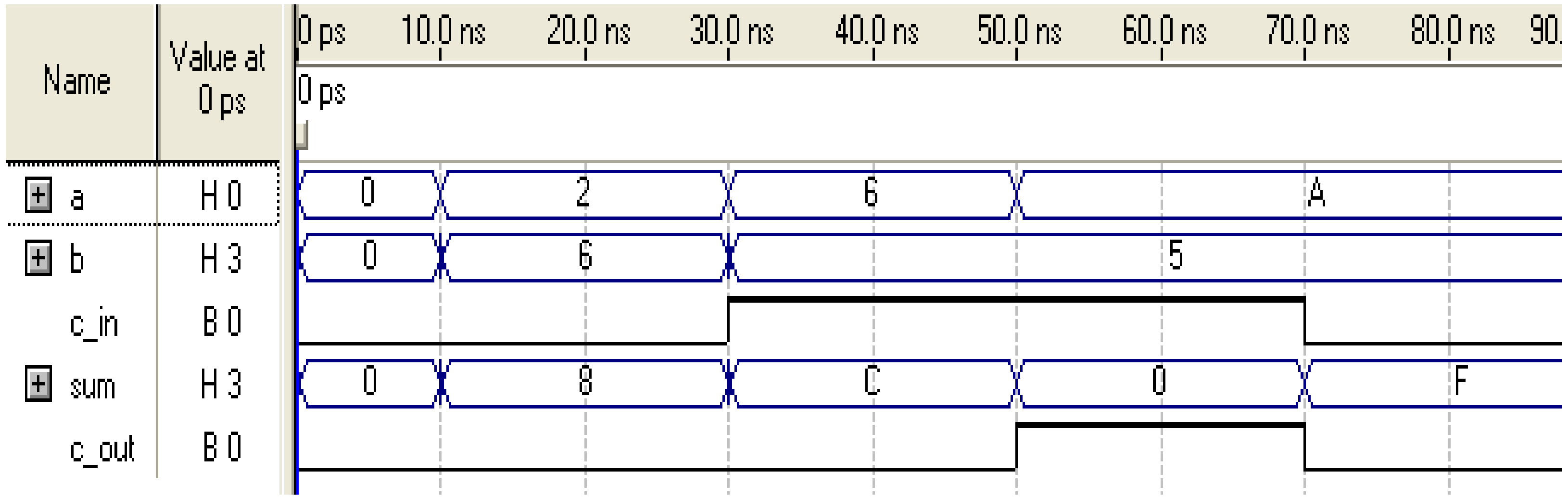
```
module Multiplexer4_to_1 (out, i0, i1, i2, i3, s0 , s1);
    output  out;
    input   i0, i1, i2, i3;
    input   s0 , s1;

    // Use nested conditional operator
    assign out = s1 ? (s0 ? i3 : i2 ) : (s0 ? i1 : i0);
endmodule
```



4-bit Full Adder

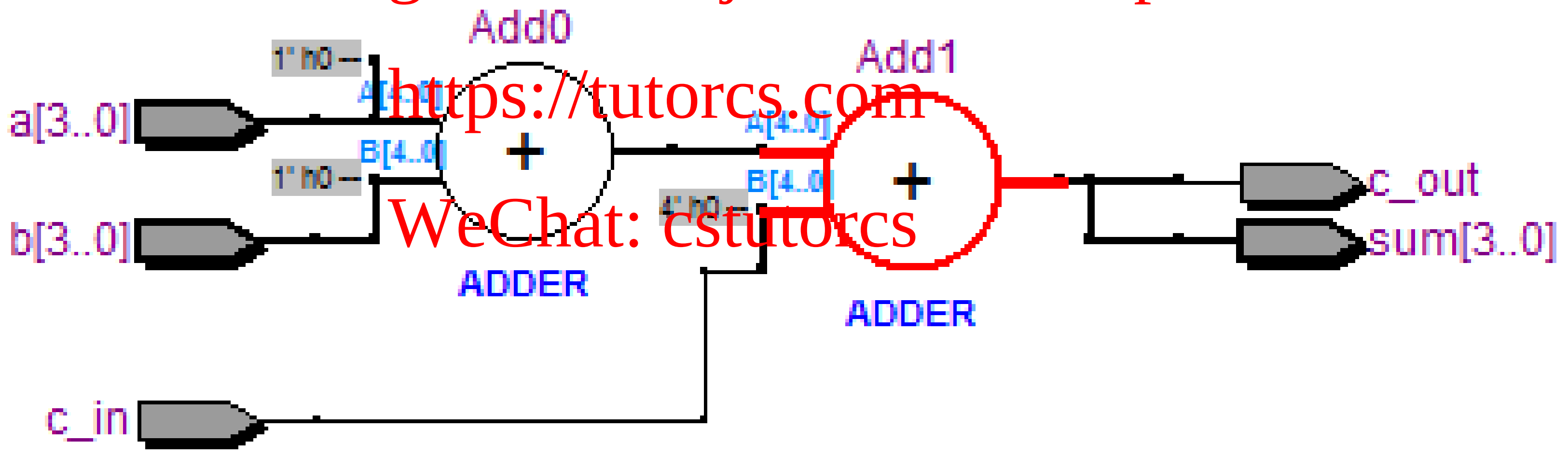


```
module FullAdd4 ( sum, c_out, a, b, c_in);
      output  [3:0] sum;
      output  c_out;
      input   [3:0] a, b;
      input   c_in;

      // Specify the function of a full adder
      assign { c_out , sum } = a + b + c_in;
endmodule
```



Synthesised Circuit

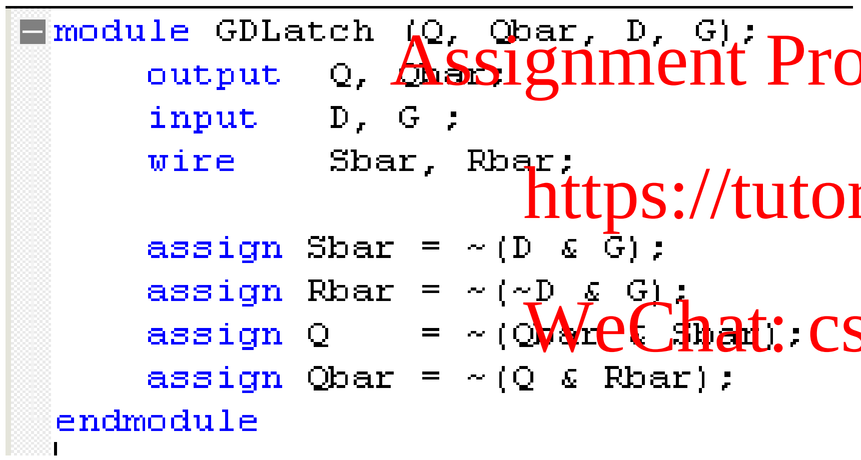
- This synthesis uses two 5-bit full adders.



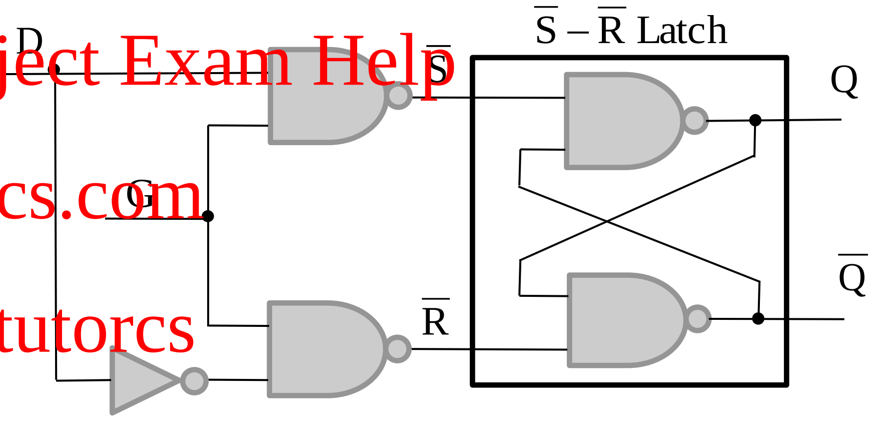
Gated D-Latch

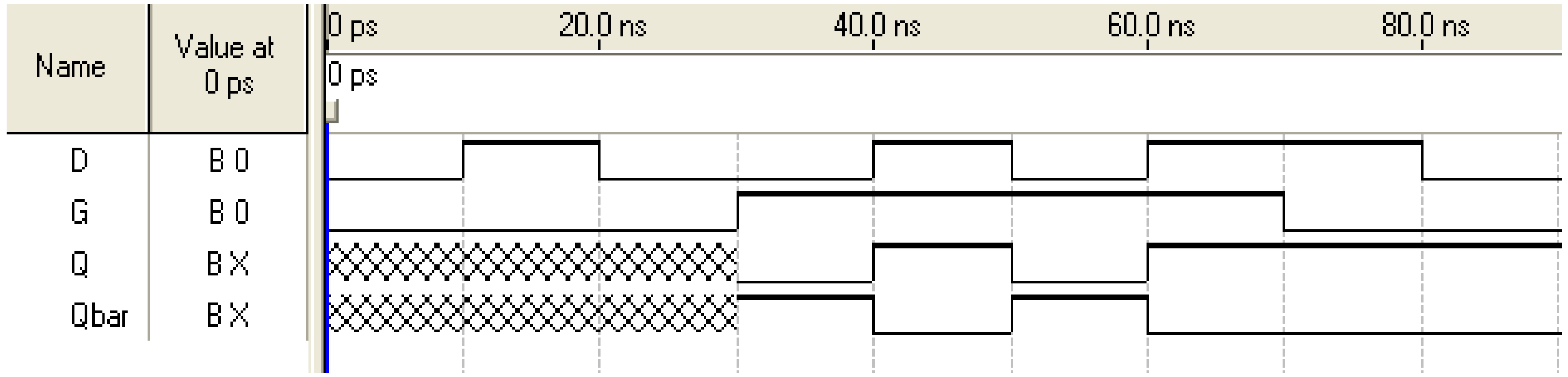
When G=1 then Q=D .

When G=0 then Q holds the last value of D (no state change).



```
module GDLatch (Q, Qbar, D, G);
    output  Q, Qbar;
    input   D, G ;
    wire    Sbar, Rbar;

    assign Sbar = ~(D & G);
    assign Rbar = ~(~D & G);
    assign Q    = ~(Qbar & Sbar);
    assign Qbar = ~(Q & Rbar);
endmodule
```

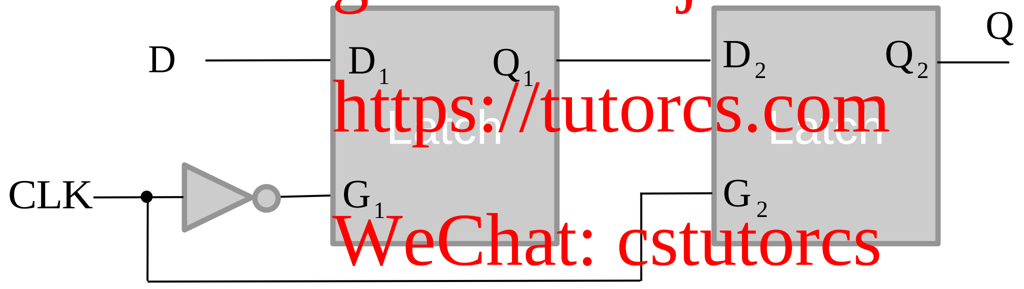




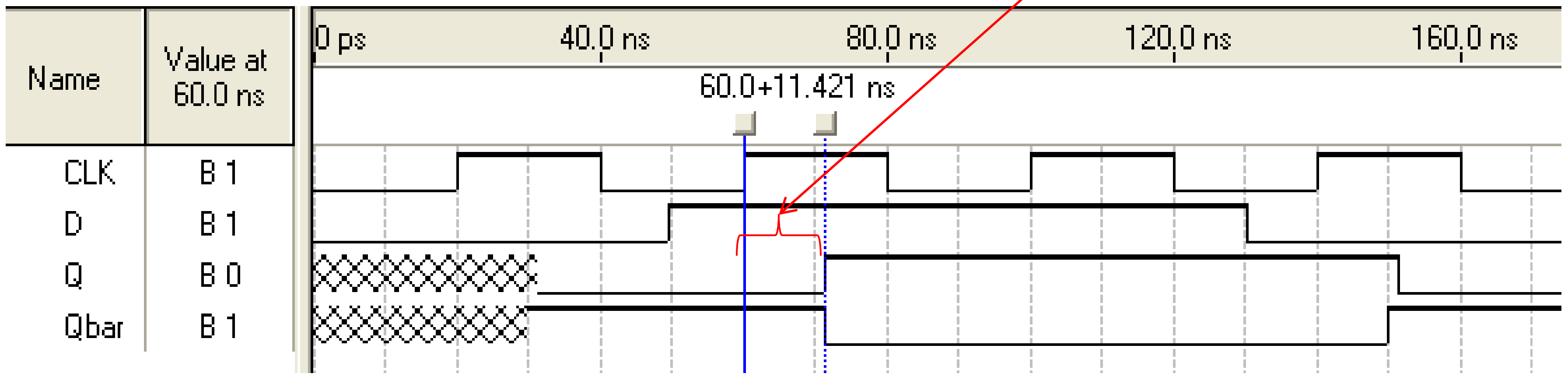
Edge Triggered D Type Flip-Flop

When CLK=0 then the value of D is latched in Q_1 and the second latch output does not change.

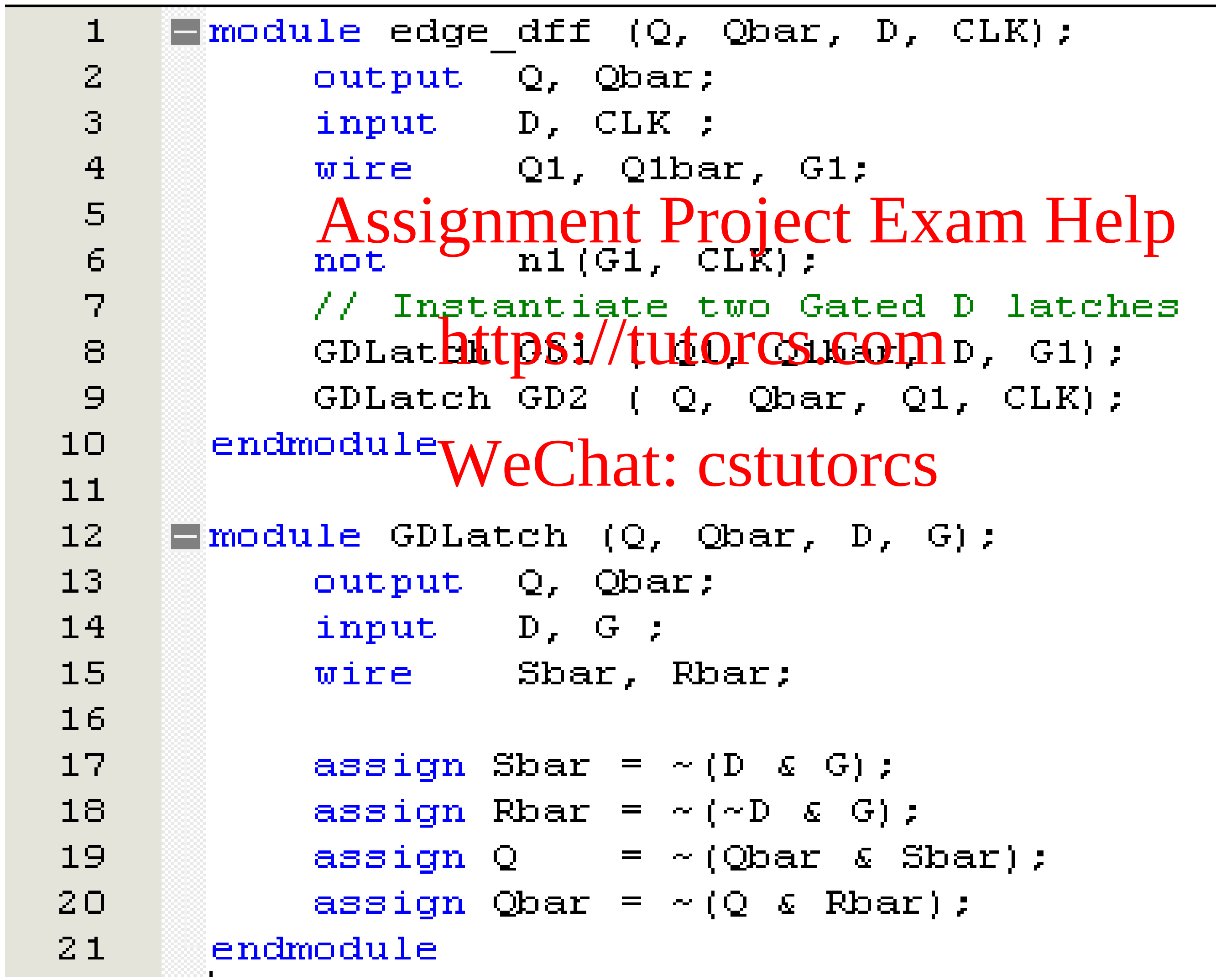
When CLK=1 then the stored value of D IN Q_1 is latched in Q_2.



Propagation delay time

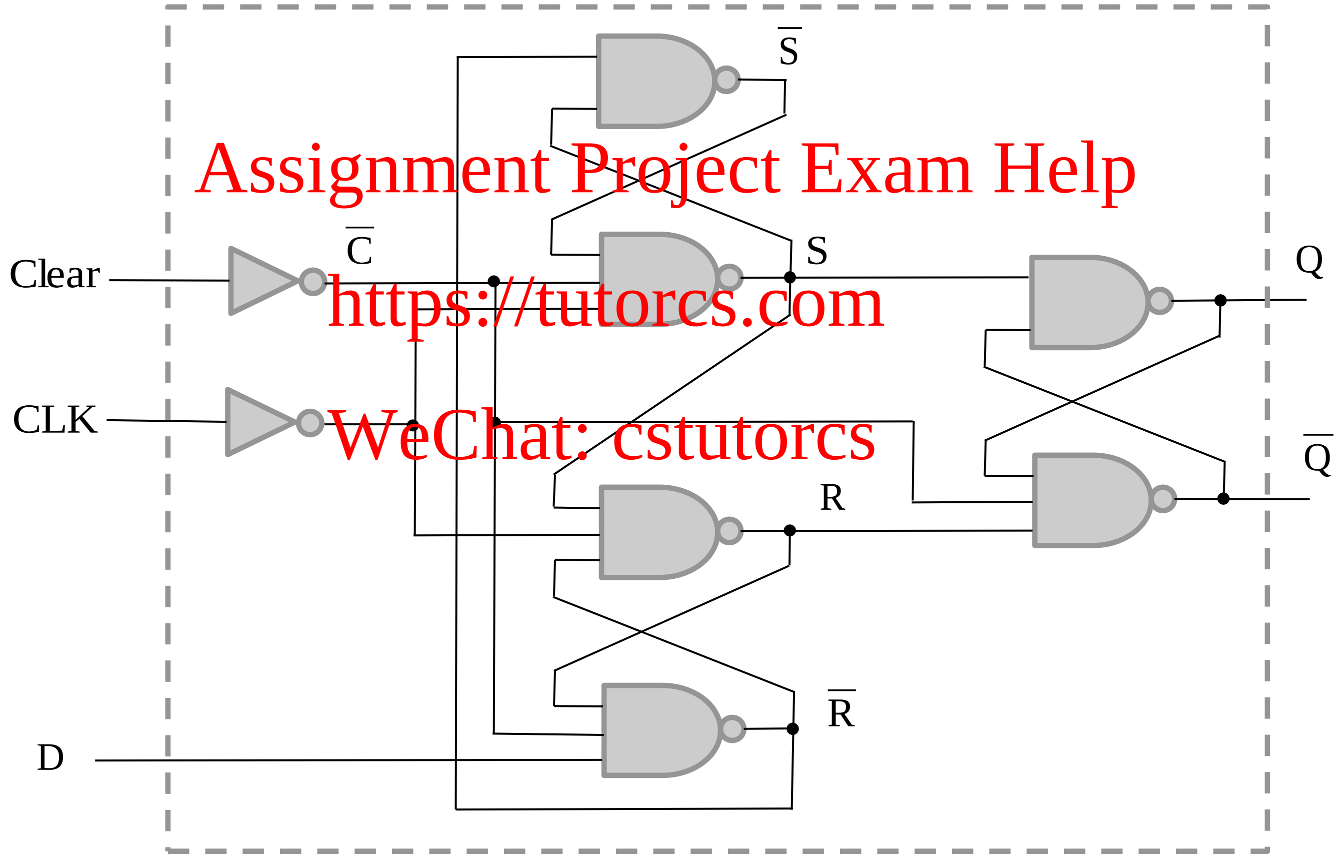


Verilog Code for D-Type Flip-flop

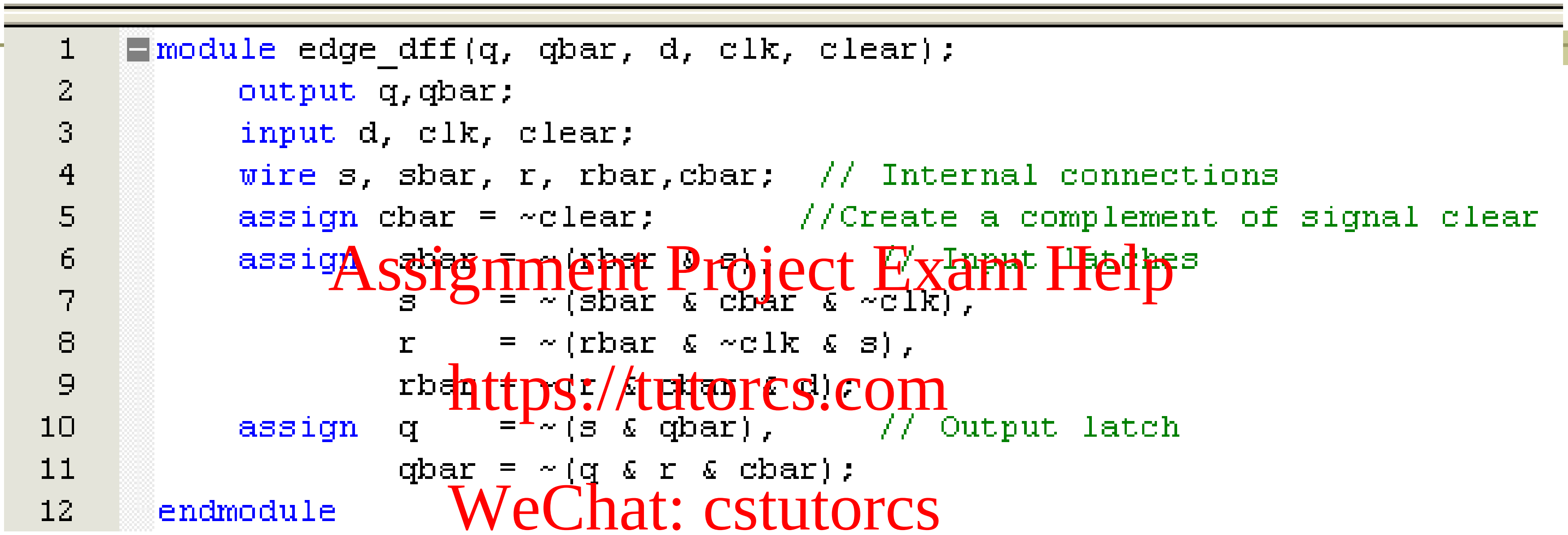


```verilog
module edge_dff (Q, Qbar, D, CLK);
        output  Q, Qbar;
        input   D, CLK ;
        wire    Q1, Q1bar, G1;

        not     n1(G1, CLK);
        // Instantiate two Gated D latches
        GDLatch GD1 ( Q1, Q1bar, D, G1);
        GDLatch GD2 ( Q, Qbar, Q1, CLK);
endmodule

module GDLatch (Q, Qbar, D, G);
        output  Q, Qbar;
        input   D, G ;
        wire    Sbar, Rbar;

        assign Sbar = ~(D & G);
        assign Rbar = ~(~D & G);
        assign Q    = ~(Qbar & Sbar);
        assign Qbar = ~(Q & Rbar);
endmodule
```

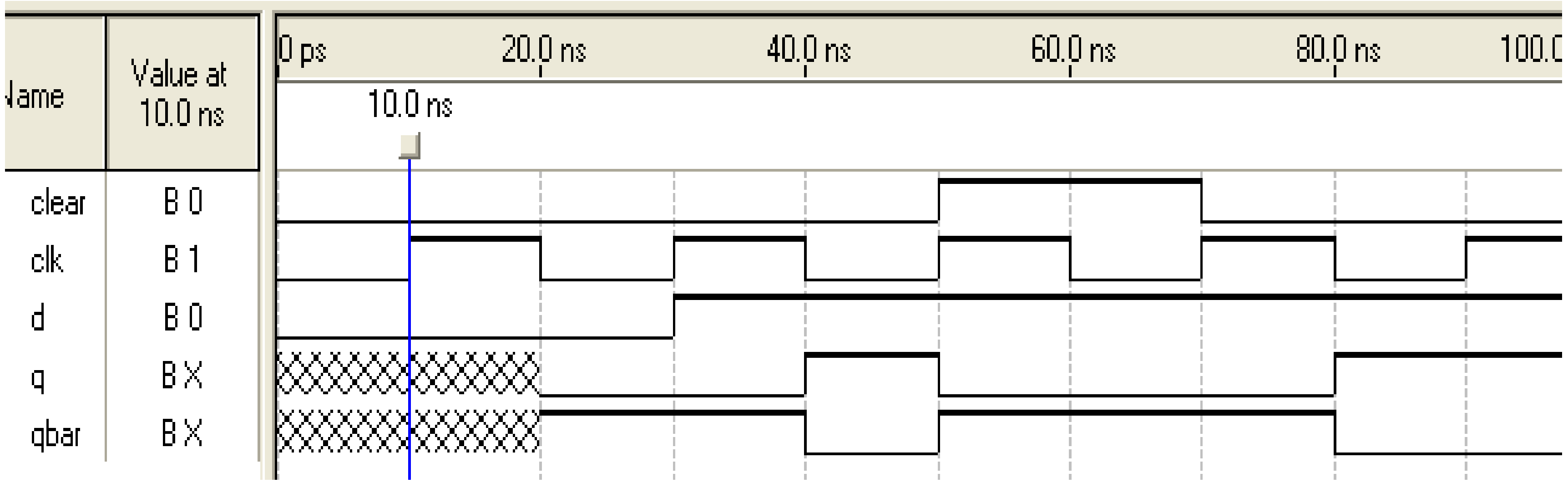
Efficient Negative Edge-Triggered D-type Flip-flop with Asynchronous Clear



Verilog code and Functional Simulation

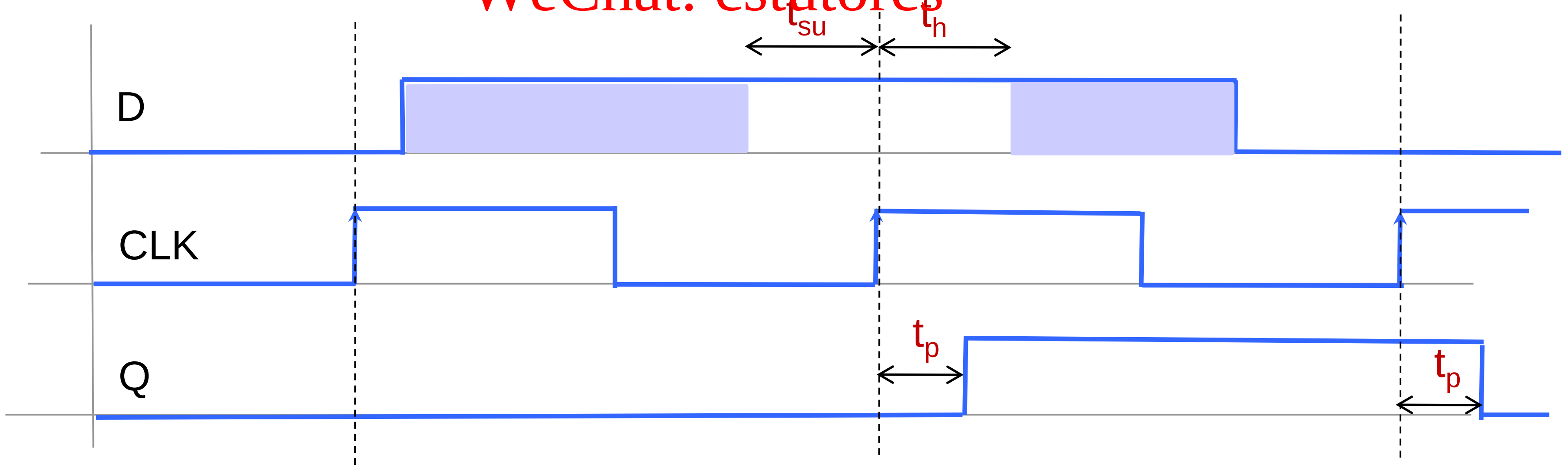


```verilog
module edge_dff(q, qbar, d, clk, clear);
      output q,qbar;
      input d, clk, clear;
      wire s, sbar, r, rbar,cbar;  // Internal connections
      assign cbar = ~clear;        //Create a complement of signal clear
      assign sbar = ~(rbar & s),   // Input latches
             s    = ~(sbar & cbar & ~clk),
             r    = ~(rbar & ~clk & s),
             rbar = ~(r & clear & d);
      assign q    = ~(s & qbar),     // Output latch
             qbar = ~(q & r & cbar);
endmodule
```

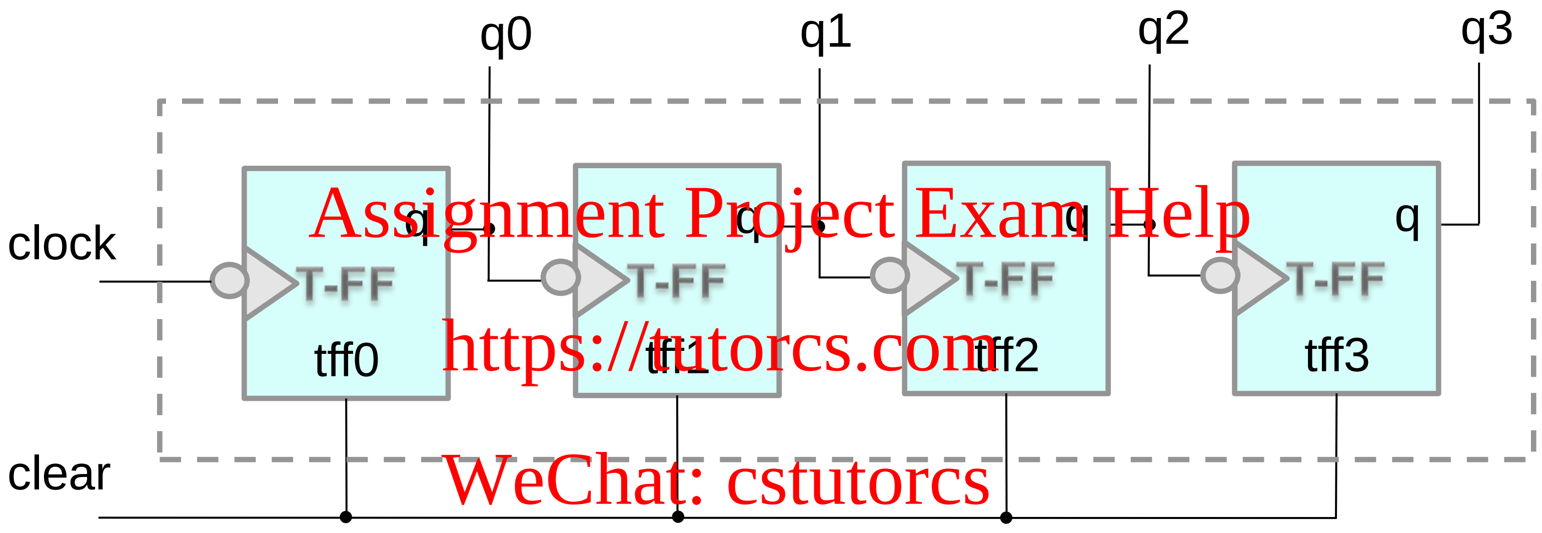


Setup and Hold Times for D Flip-flops

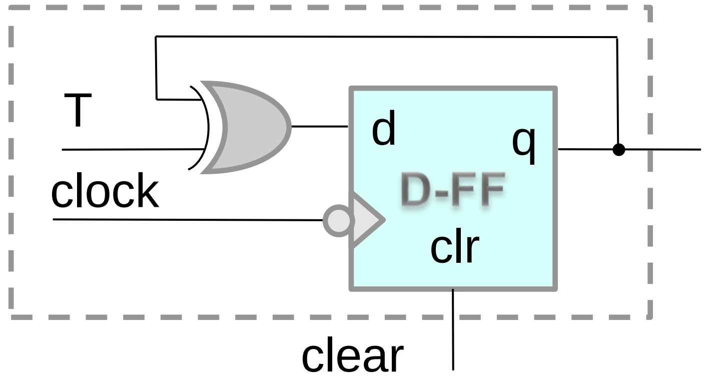
- For the proper function of the edge triggered FF
  - D input must be stable for t_{su} time (setup time) before the active edge of the clock.
  - D input must be stable for t_h time (hold time) after the active edge of the clock.



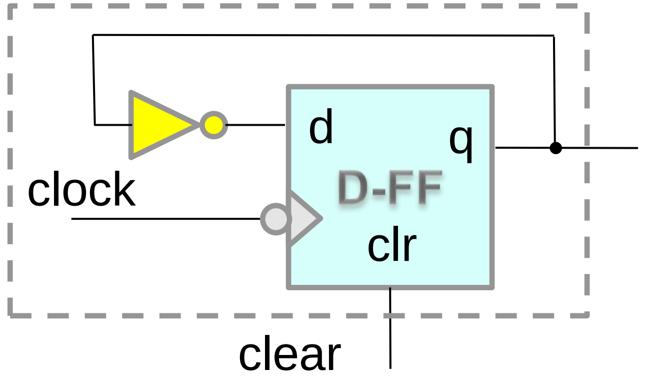
Example: 4_bit Ripple Counter
NB: This is bad it’s Asynchronous



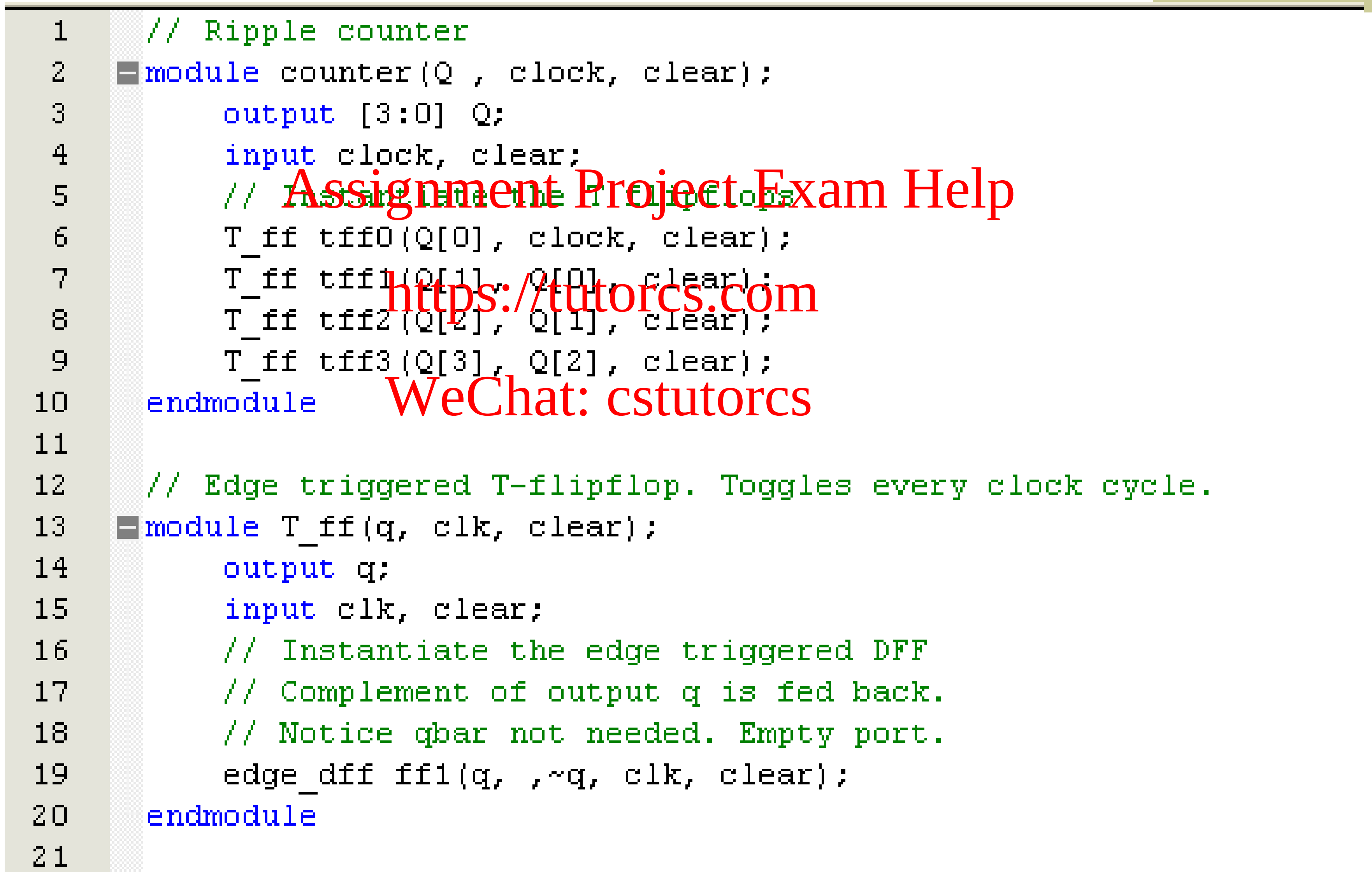
T-FF



T-FF with T=1

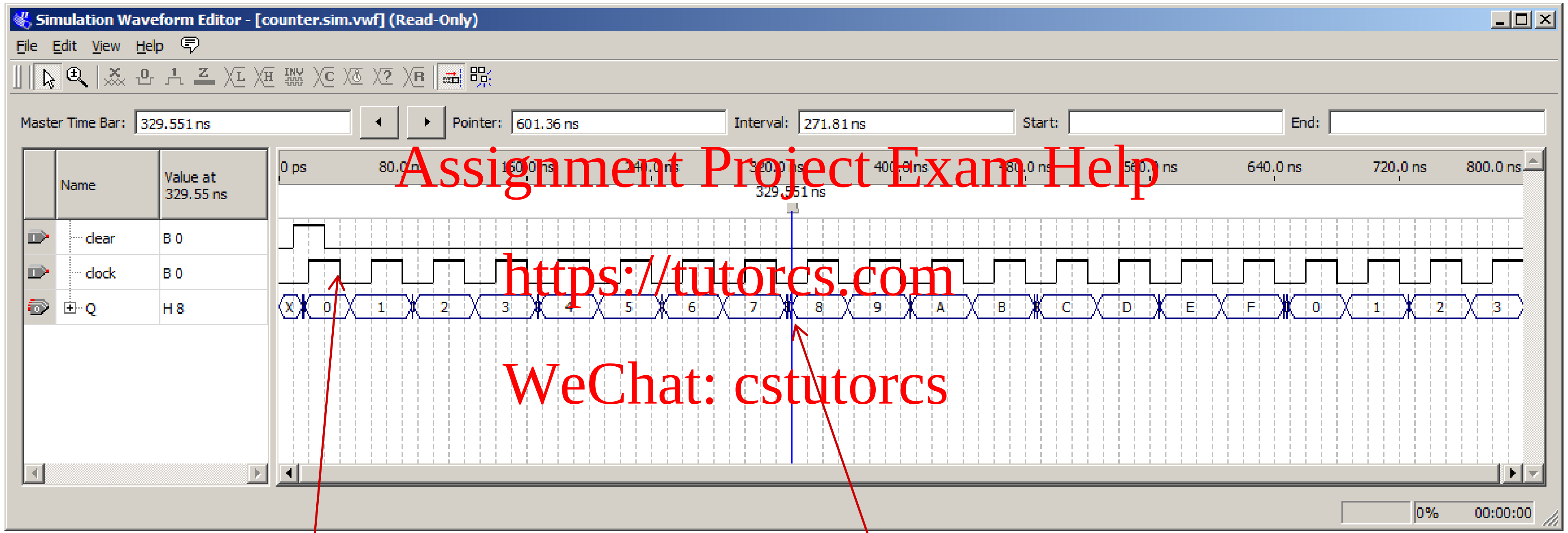


Verilog Code
NB: This is bad it's Asynchronous



```
// Ripple counter
module counter(Q , clock, clear);
    output [3:0] Q;
    input clock, clear;
    // Instantiate the flipflops
    T_ff tff0(Q[0], clock, clear);
    T_ff tff1(Q[1], Q[0], clear);
    T_ff tff2(Q[2], Q[1], clear);
    T_ff tff3(Q[3], Q[2], clear);
endmodule

// Edge triggered T-flipflop. Toggles every clock cycle.
module T_ff(q, clk, clear);
    output q;
    input clk, clear;
    // Instantiate the edge triggered DFF
    // Complement of output q is fed back.
    // Notice qbar not needed. Empty port.
    edge_dff ff1(q, ,~q, clk, clear);
endmodule
```

Timing Simulation

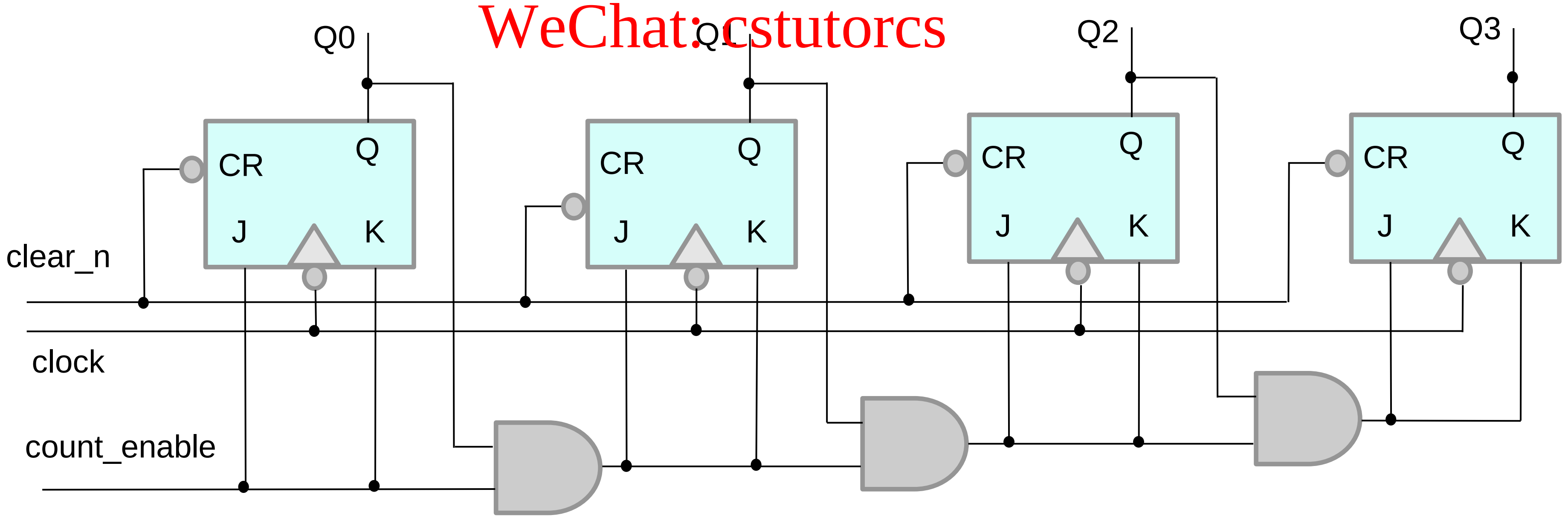


Falling edge trigger

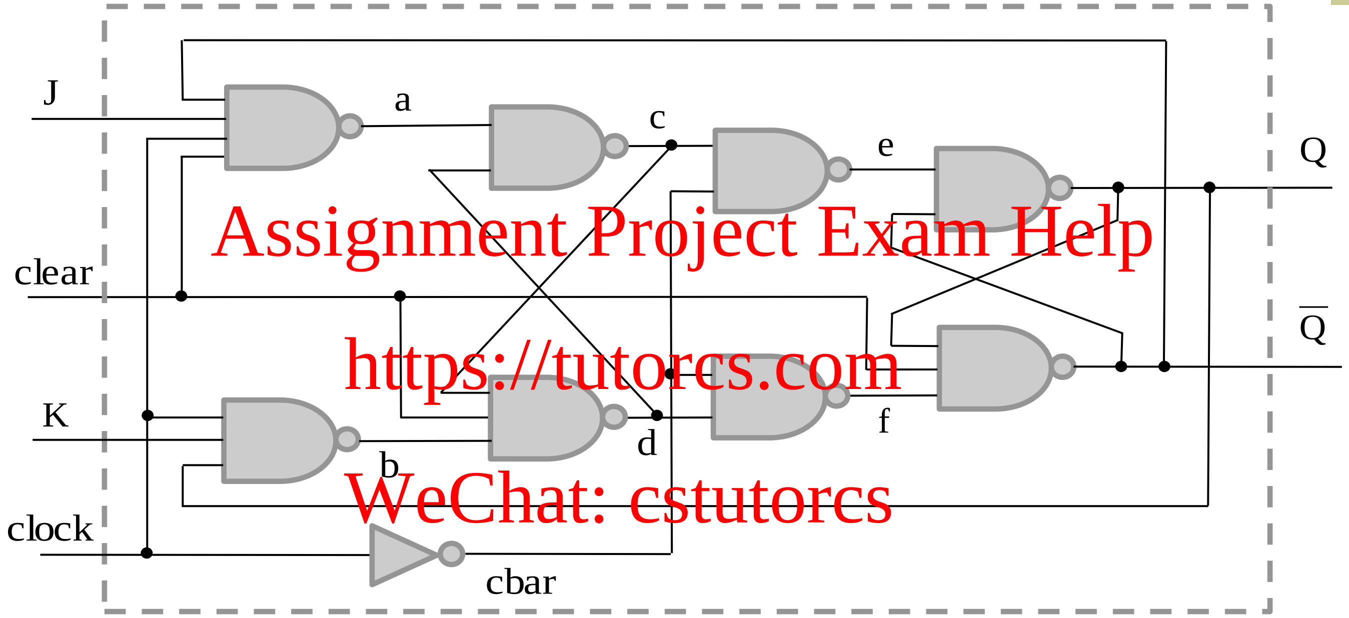
9.551 ns is the propagation delay

Exercise: Synchronous Counter using Master-Slave J-K Flip-flops

- Write a Verilog data flow description of the following synchronous counter and simulate it using Quartus II.
- The J-K flip-flops are reset to 0 when clear_n becomes low.
- The flip-flops change state on the falling edge of the clock.
- Counting is disabled when count_enable signal is low.

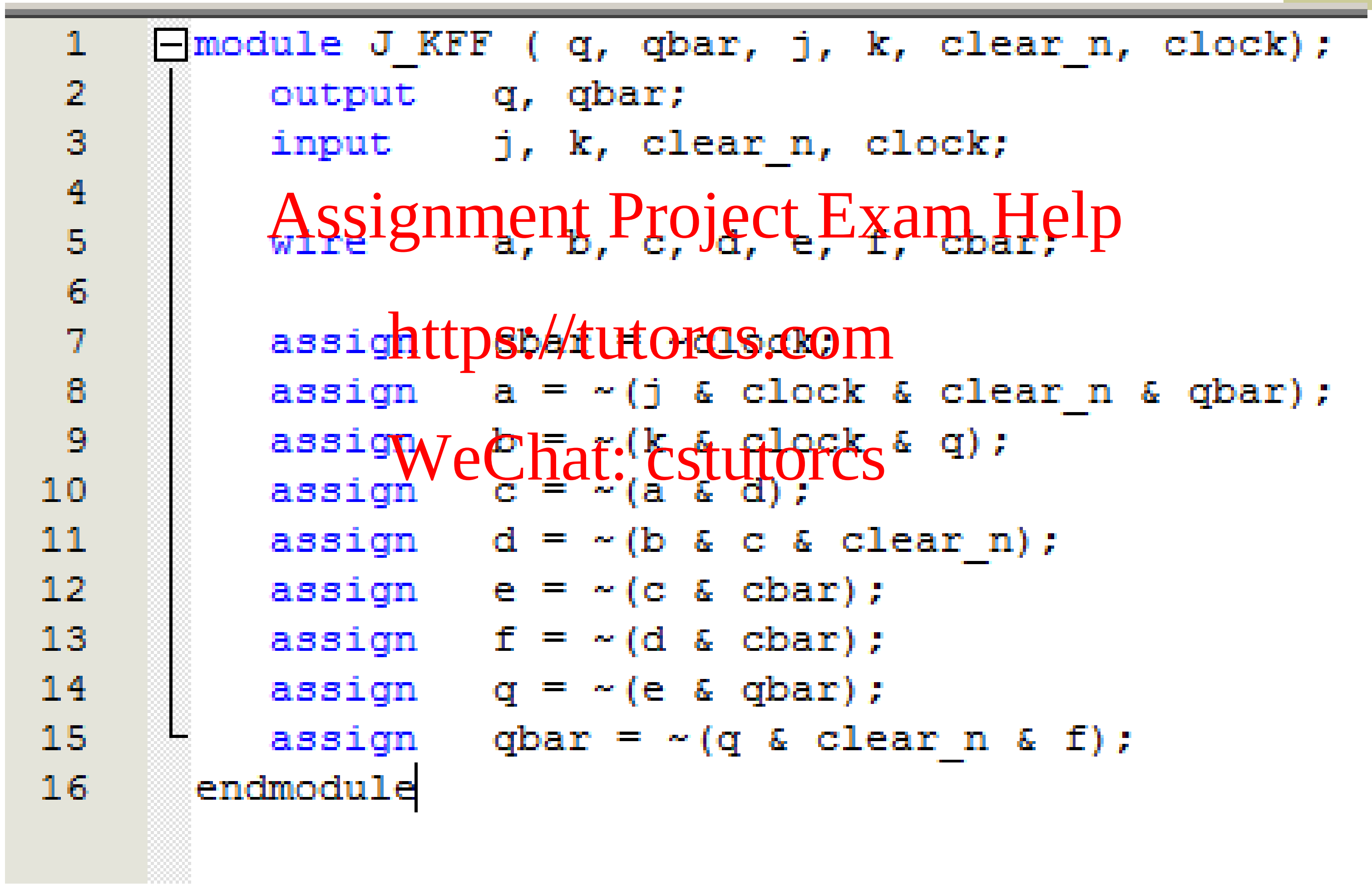


Master-Slave J-K Flip-flop



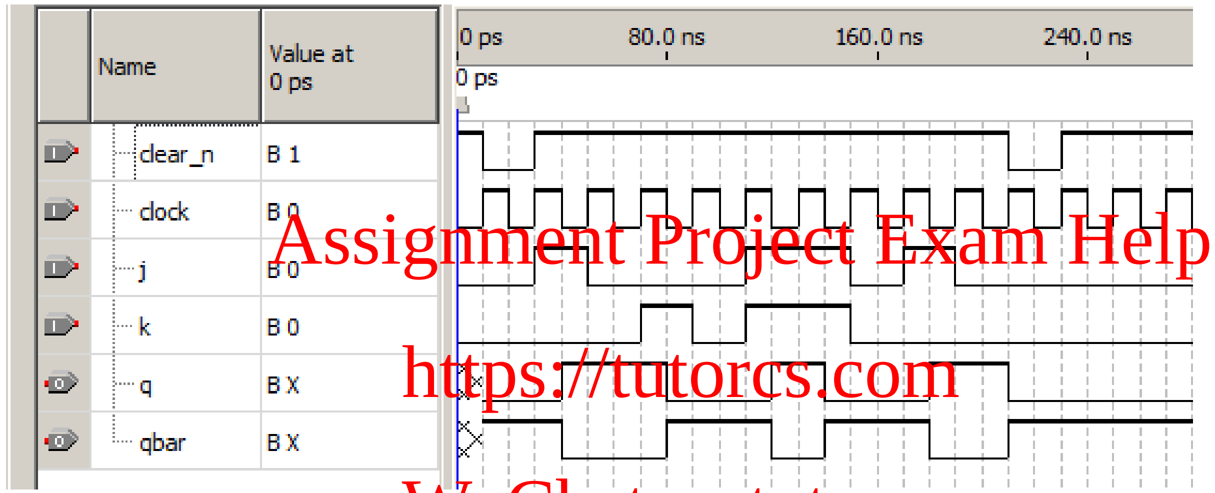
You may encounter a timing issue in the timing simulation. In this case use the JKFF flip-flop from the library. From the Edit menu select Insert Template then expand the Verilog HDL \ Altera Primitives\Registers and Latches and select the JKFF

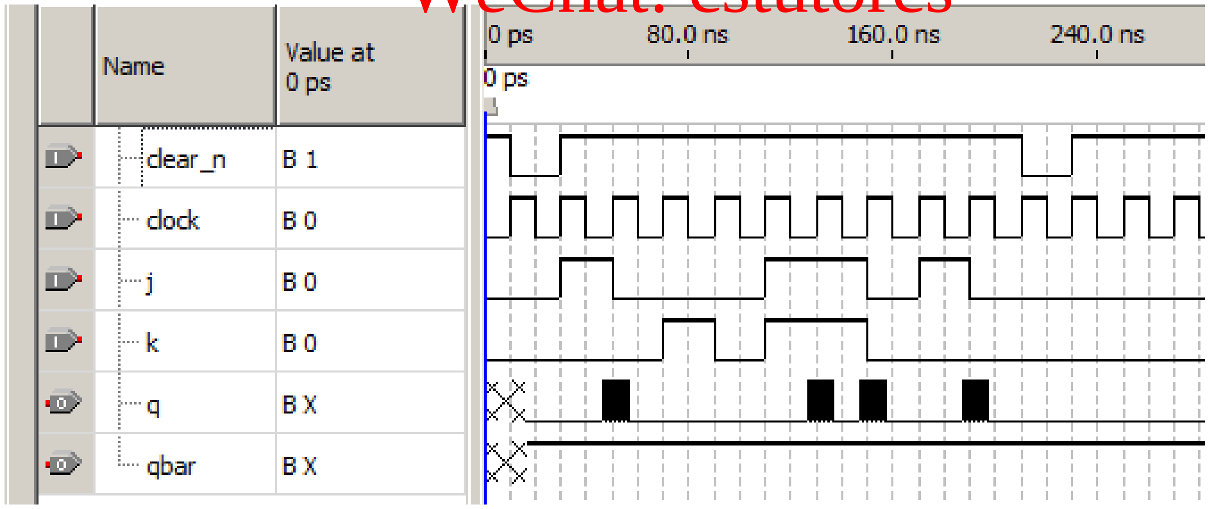
JK Flip-flop



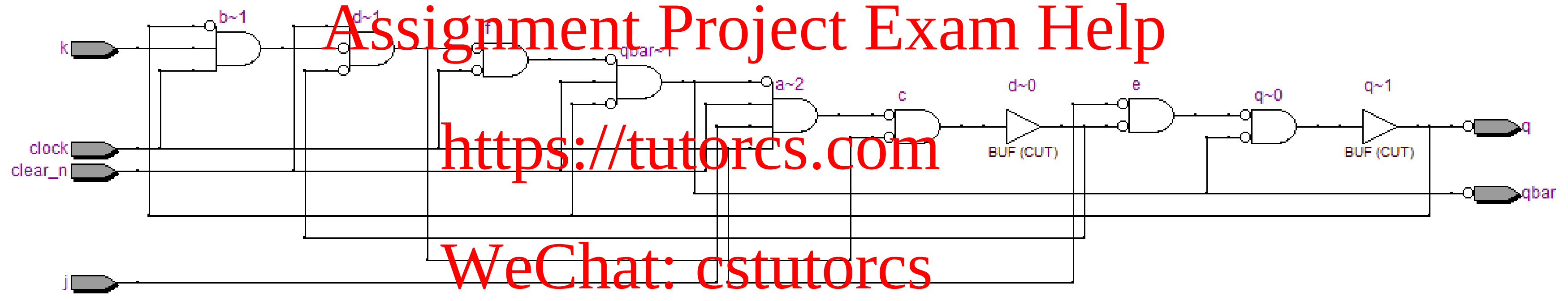
```
module J_KFF ( q, qbar, j, k, clear_n, clock);
    output   q, qbar;
    input    j, k, clear_n, clock;

    wire     a, b, c, d, e, f, cbar;

    assign   cbar = ~clock;
    assign   a = ~(j & clock & clear_n & qbar);
    assign   b = ~(k & clock & q);
    assign   c = ~(a & d);
    assign   d = ~(b & c & clear_n);
    assign   e = ~(c & cbar);
    assign   f = ~(d & cbar);
    assign   q = ~(e & qbar);
    assign   qbar = ~(q & clear_n & f);
endmodule
```

Functional and Timing Simulation



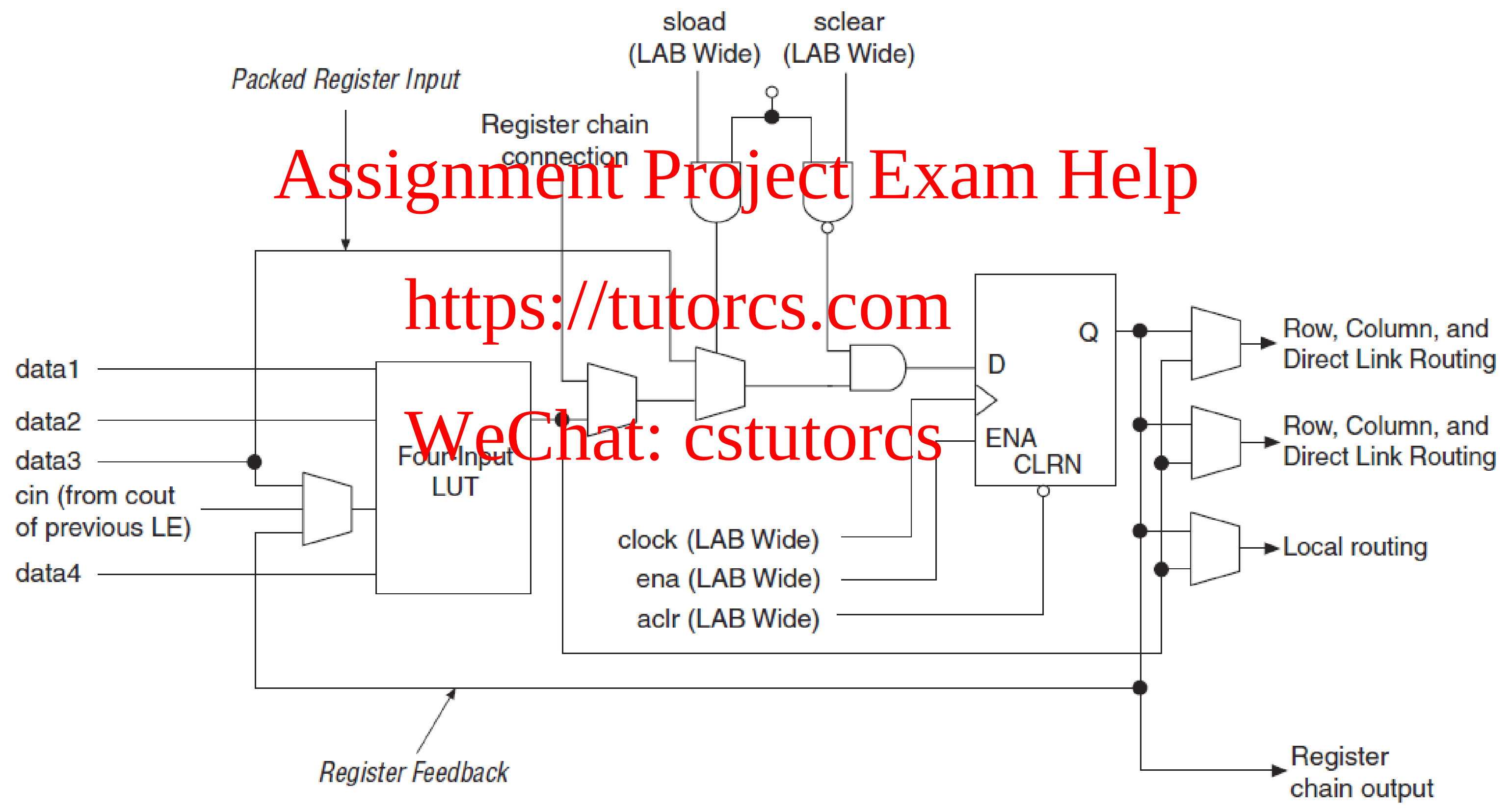


RTL Viewer

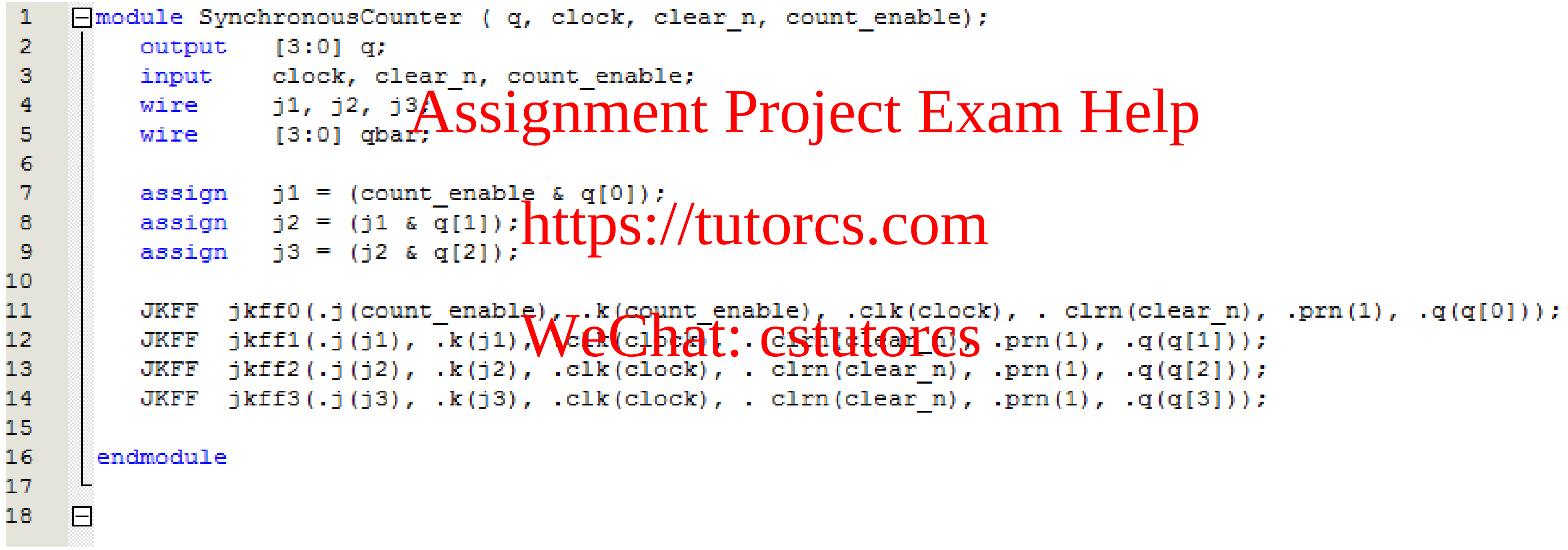


k
clock
clear_n
j
b~1
d~1
f
qbar~1
a~2
c
d~0
BUF (CUT)
e
q~0
q~1
BUF (CUT)
q
qbar

Cyclone II LE in Normal Mode

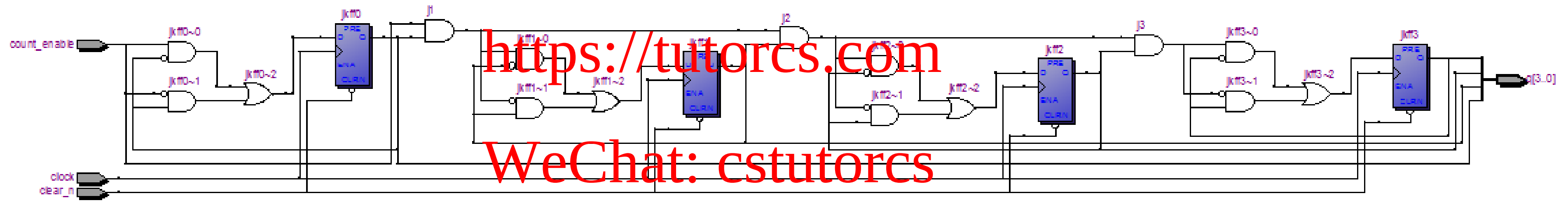


Using Altera JK Primitives

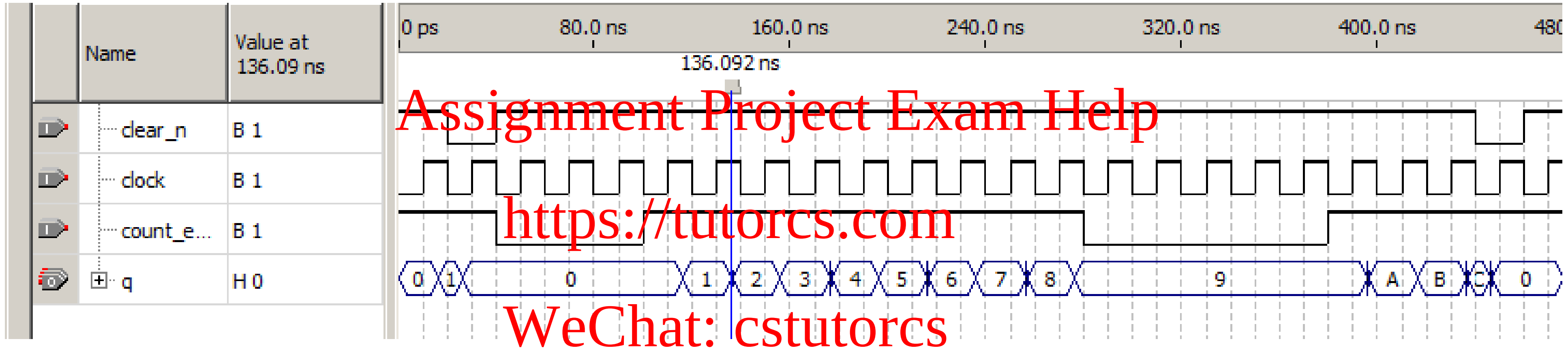


```
module SynchronousCounter ( q, clock, clear_n, count_enable);
    output    [3:0] q;
    input     clock, clear_n, count_enable;
    wire      j1, j2, j3;
    wire      [3:0] qbar;

    assign    j1 = (count_enable & q[0]);
    assign    j2 = (j1 & q[1]);
    assign    j3 = (j2 & q[2]);

    JKFF  jkff0(.j(count_enable), .k(count_enable), .clk(clock), . clrn(clear_n), .prn(1), .q(q[0]));
    JKFF  jkff1(.j(j1), .k(j1), .clk(clock), . clrn(clear_n), .prn(1), .q(q[1]));
    JKFF  jkff2(.j(j2), .k(j2), .clk(clock), . clrn(clear_n), .prn(1), .q(q[2]));
    JKFF  jkff3(.j(j3), .k(j3), .clk(clock), . clrn(clear_n), .prn(1), .q(q[3]));

endmodule
```

RTL Viewer of counter



Timing Simulation



RTL Summary

- The assign statement was explained.
- Some Verilog operators were explained via examples.
- 4-to-1 mux and full adder circuits were modeled as combinational circuit examples.
- Data flow modeling was used to model D-type flip-flop and a ripple counter.
- Setup and hold times in FFs were explained.
- Synchronous counters was introduced and the importance of the timing simulation was explained.