Digital Systems Design ELEC373/473

Behavioural Modelling

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

Overview of HDLs

- Provides many descriptive styles
  - Structural
  - Register Transfer Level (RTL)
  - Behavioral
- HDL based designs are highly portable and independent of technology.
- HDLs are not like procedural software and cannot run on processors.
- HDLs are a convenient medium for integrating intellectual property (IP) from a variety of sources with a proprietary design.

Learning Objectives

- Explain the "**always**" and "**initial**" procedures in behavioural modelling.
- Define *blocking* and *nonblocking* procedural assignments.
- Explain conditional statements using "**if**" and "**else**".
- Describe multiway branching using "**case**", "**casex**", and "**casez**" statements.
- Understand the usage of the *blocking* and *nonblocking* assignments via examples.

Why Behavioural Modelling

- Behavioural modelling describes the functionality of a design. It describes
  - what the circuit will do
  - not how to build it in hardware.
- Designers need to evaluate the tradeoffs of various architectures and algorithms before they implement the optimum one in hardware.
- Behavioural modelling represent the circuit at a very high level of abstraction.
- Behavioural modelling provides *flexibility* by allowing part of the design to be modelled at different levels of abstraction.
- Modern synthesis tools are now able to generate RTL circuits from behavioural models:
  - Its “like” using a C compiler rather than working with assemble language (RTL) – But remember its hardware.

Structured Procedures

- ***always*** and ***initial*** are the two most basic statements in behavioural modelling.
  - All other behavioural statements can only appear inside these structured procedure statements.
- Verilog is a *concurrent* simulation language.
  - Activity flows in Verilog run in parallel rather than in sequence.
- Each activity flow starts at simulation time 0.
- Each ***always*** and ***initial*** statement represent a separate activity flow.
- The ***always*** and ***initial*** statements cannot be nested.

always and *initial* statements

- The ***always*** statement starts at time 0 and executes the statements in a looping fashion.
- The ***always*** statement is used to model a block of activity that is repeated continuously in a digital circuit.
- The ***initial*** statement executes exactly once during a *simulation*.
- The ***initial*** blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire *simulation* run.
  - However the ***initial*** block in Quartus will set the output values of pins after programming
- The ***initial*** blocks have no meaning for synthesis all initialisation in synthesis is done via the reset signal.
- The statements inside the ***always*** block are executed sequentially and are "procedural" statements.

Procedural Assignments

- Procedural assignments update values of ***reg***, ***integer***, ***real***, or ***time*** variables.
- The value placed on a variable will remain unchanged until another procedural statement updates the variable.
- The left-hand side (LHS) of the assignment must be a variable and the RHS can be any expression that evaluates to a value.
- The (=) operator is used to specify blocking statements.
  - Blocking statements are executed sequentially.
- The (<=) operator is used for nonblocking statements.
  - The nonblocking assignments allow scheduling of assignments without blocking the execution of the statements that follow in a sequential block.

Verilog Registers

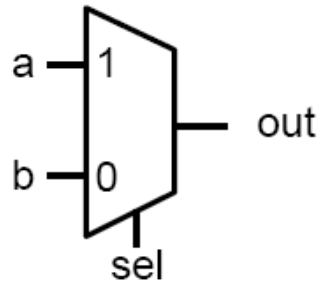
- In digital design, registers represent memory elements.
- Digital registers need a clock to operate and update their state on a certain phase or edge
- Registers in Verilog should not be confused with hardware registers.
- In Verilog the term register (reg) simply means a variable that can hold its value.
- Verilog registers don’t need a clock and don’t need to be driven by a net. Values of registers can be changed anytime in a simulation by assigning a new value to the register.

The Sequential always Block

- Edge-triggered circuits are described using a sequential `always` block

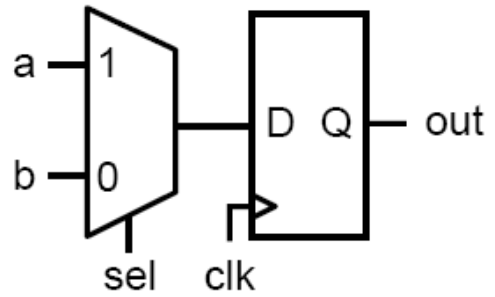
Combinational

```
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

```
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
```



Event Control Statements

- Event Control
  - Edge Triggered Event Control
  - Level Triggered Event Control
- Edge Triggered Event Control

  @ (posedge CLK) //Positive Edge of CLK

  Curr_State <= Next_state;
- Level Triggered Event Control

  @ (A or B) //change in values of A or B

  Out = A & B;

@ negedge	@ posedge
1 → x	0 → x
1 → z	0 → z
1 → 0	0 → 1
x → 0	x → 1
z → 0	z → 1

Importance of the Sensitivity List

- The use of `posedge` and `negedge` makes an `always` block sequential (edge-triggered)
- Unlike a combinational `always` block, the sensitivity list does determine behavior for synthesis!

D Flip-flop 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
```

`always` block entered only at each positive clock edge

D Flip-flop 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
```

`always` block entered immediately when (active-low) clearb is asserted

Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`

If one signal in the sensitivity list uses posedge/negedge, then all signals must.

- **Assign any signal or variable from only one always block, Be wary of race conditions: always blocks execute in parallel**

Cyclic Behavioural Models of Flip-flops and Latches

- Continuous assignments cannot model an element that has edge-sensitive behaviour, such as a flip-flop.
- Verilog uses a cyclic behaviour to model edge-sensitivity behaviour.
- Cyclic behaviours are abstract (they do not use hardware to specify signal values).


```
module dff_behav( q, qbar, data, set, reset, clk);
input    data, set, reset, clk;
output   q, q_bar;
reg      q;
assign   q_bar = ~q;
always @ (posedge clk) // Flip-flop with synchronous set/reset
    begin
        if (reset == 0) q<=0;  else if (set == 0) q<=1; else q<= data;
    end
endmodule
```

Conditional Statement

- The *conditional* *statement (or* *if-else* *statement) is used to make a decision as to whether a statement is executed* or not.
- **if** (expression) statement_or_null
  [**else** statement_or_null]
  - If the expression evaluates to true the first statement shall be executed.
  - If it evaluates to false (has a zero value or the value is x or z), the first statement shall not execute.
  - If there is an **else** statement and the expression is false, the **else** statement shall be executed.

Conditional Statement (cont.)

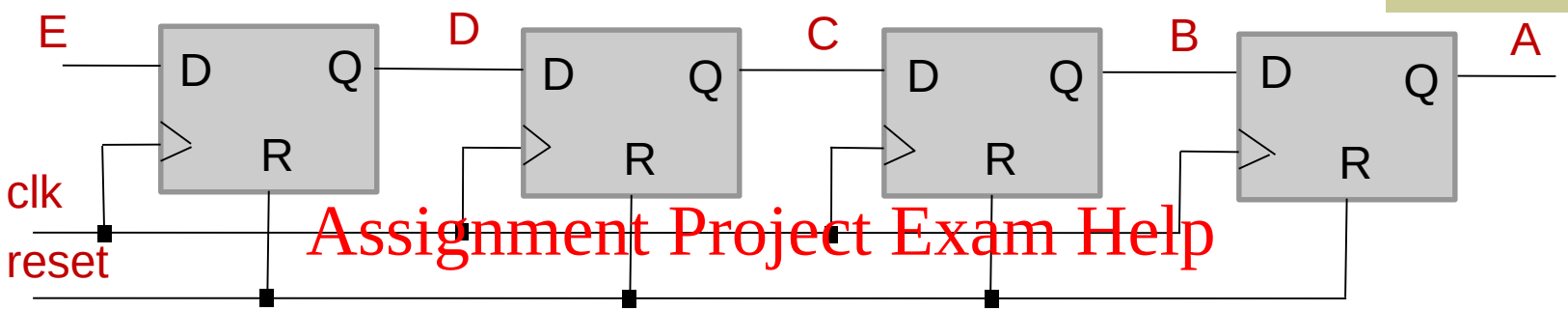
```
if (index > 0)
    if (rega > regb)
        result = rega;
    else
        result = regb;
```

else applies to preceding **if** as both **if**s do not have the optional **else**.


```
if (index > 0)          begin
    if (rega > regb)
        result = rega;
    end
else    result = regb;
```

begin-end block statement shall be used to force the proper association. Now **else** applies to the first **if**.

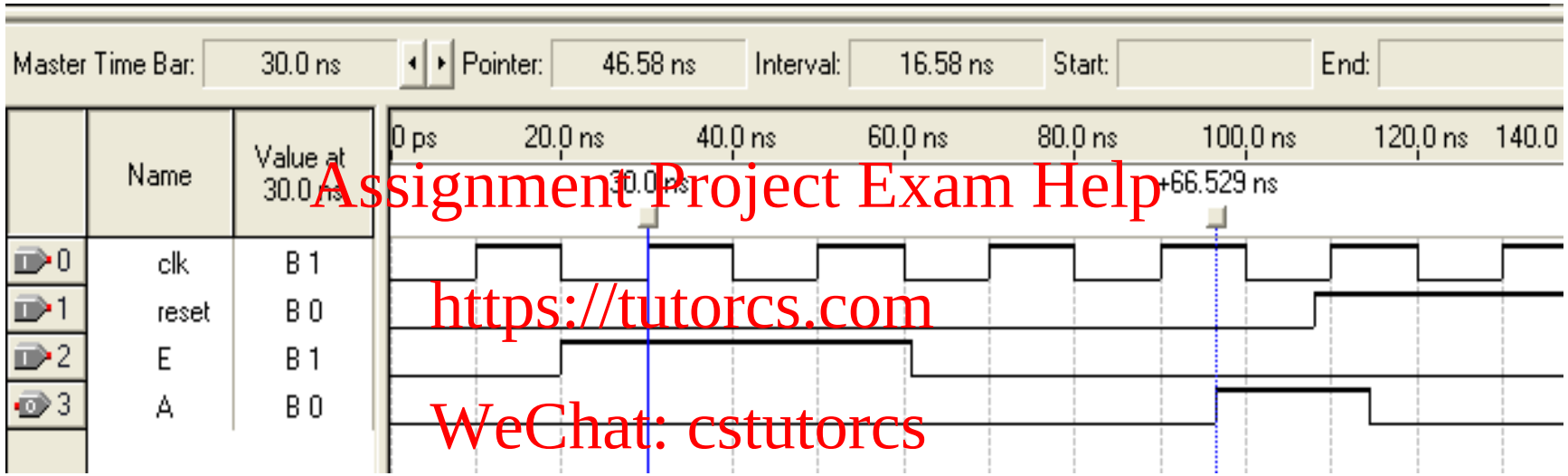
Shift Register Model using Blocking Assignment

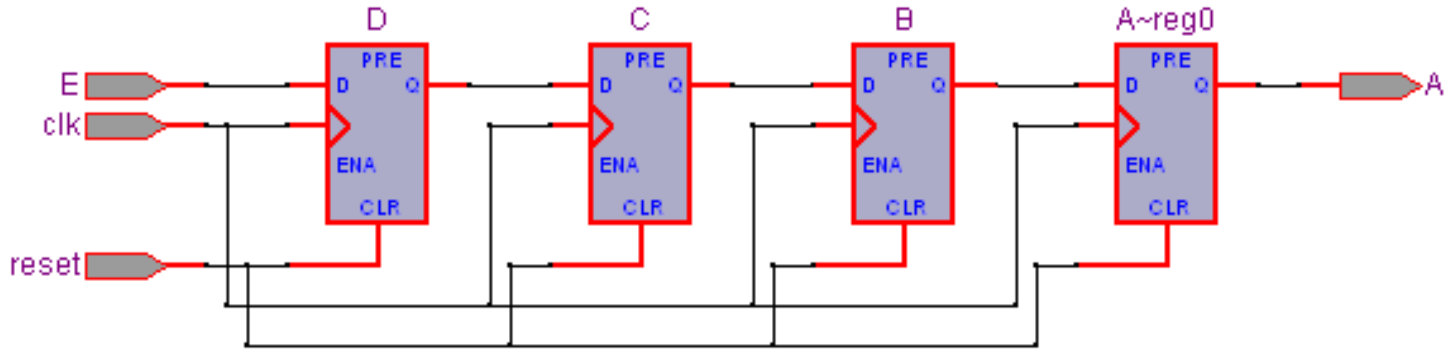



```
module ShiftRegister( A, E, reset, clk);
input    E, reset, clk;
output   A;
reg      A, B, C, D;
always @ (posedge clk or posedge reset)
begin
    if (reset) begin A=0; B=0; C=0; D=0; end
          else begin A=B; B=C; C=D; D=E;  end
end
endmodule
```

The list of statements are executed in sequential order. Blocking assignments.

Simulation and Synthesis

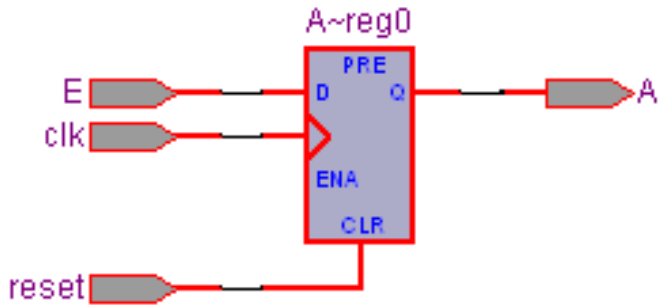




Order of Execution in Blocking Assignments

```
module ShiftRegister( A, E, reset, clk);
input    E, reset, clk;
output   A;
reg      A, B, C, D;
always @ (posedge clk or posedge reset)
begin
    if (reset) begin A=0; B=0; C=0; D=0; end
        else begin D=E; C=D; B=C; A=B; end // Equivalent to A=E
end
endmodule
```

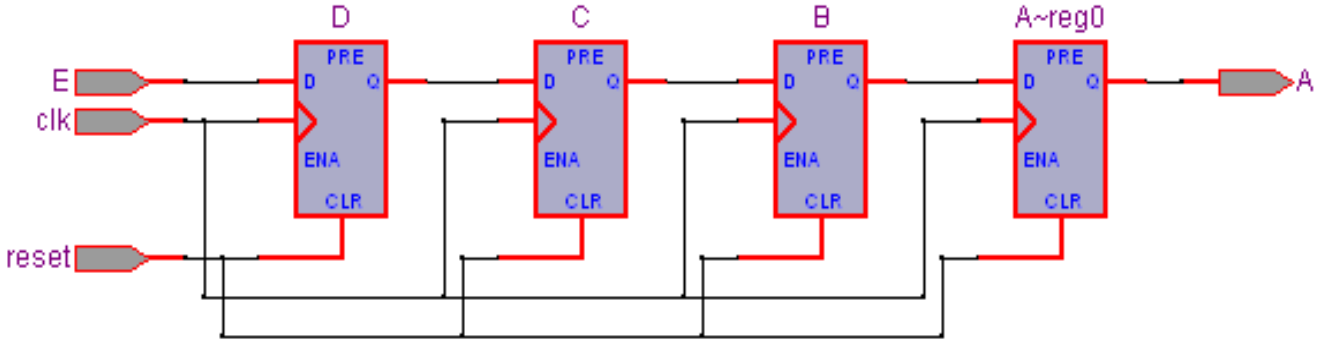

Only one flip-flop is synthesised.



Shift Register using Nonblocking Statements

```
module ShiftRegister( A, E, reset, clk);
input    E, reset, clk;
output   A;
reg      A, B, C, D;
always @ (posedge clk or posedge reset)
begin
    if (reset) begin A=0; B=0; C=0; D=0; end
        else begin D<=E; C<=D; B<=C; A<=B; end // The order is unimportant
end
endmodule
```

All of the variables in the RHS of the nonblocking assignments (<=) are sampled and held in memory to update the RHS variables concurrently.



Case Statement

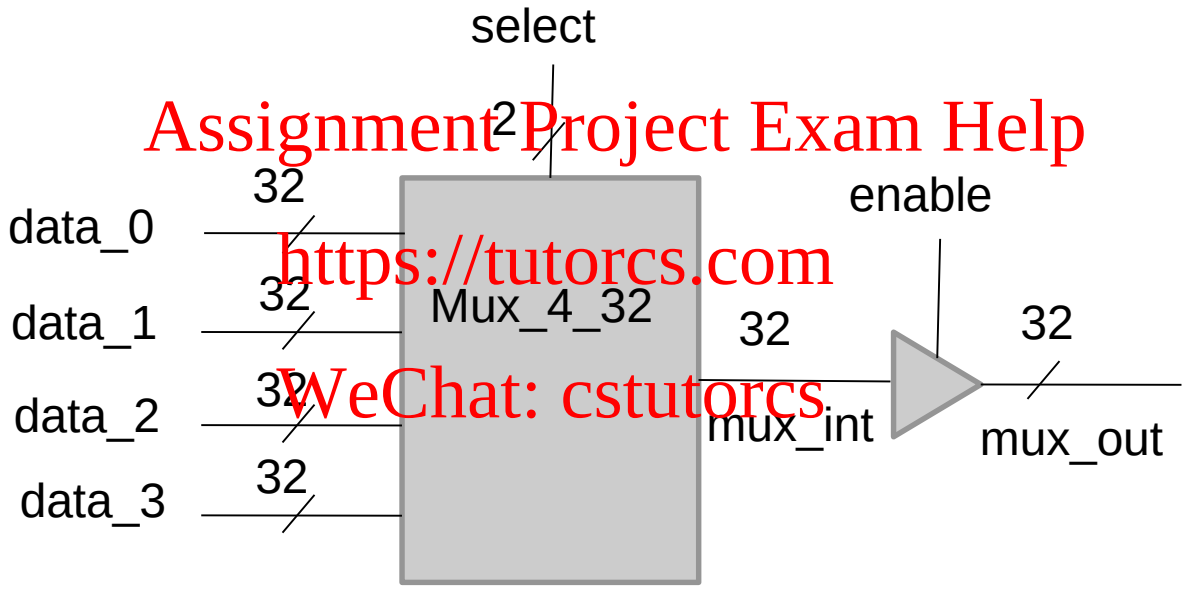
- The *case statement is a multiway decision statement.*
- **case** (expression)

  expression { , expression } : statement_or_null

  | **default** [**:**] statement_or_null

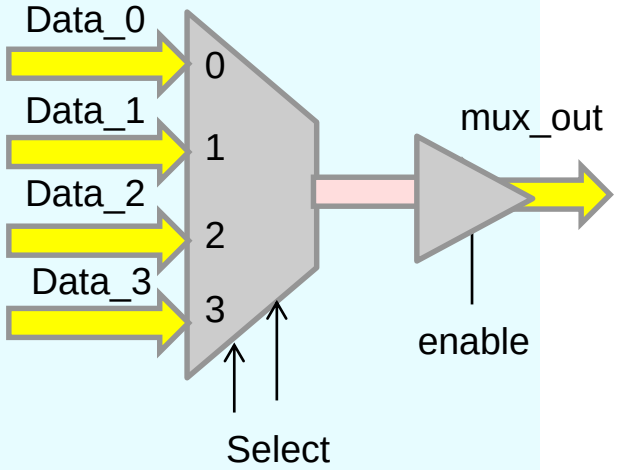
  endcase
- The *case item expressions shall be evaluated and compared with the case expression in the exact order in which they are given.*
- During the linear search, if a match occurs then the statement associated with that case item shall be executed.
- The default statement executes if all comparisons fail.
- The default statement is optional.

4-channel 32-bit Multiplexer with tri-state

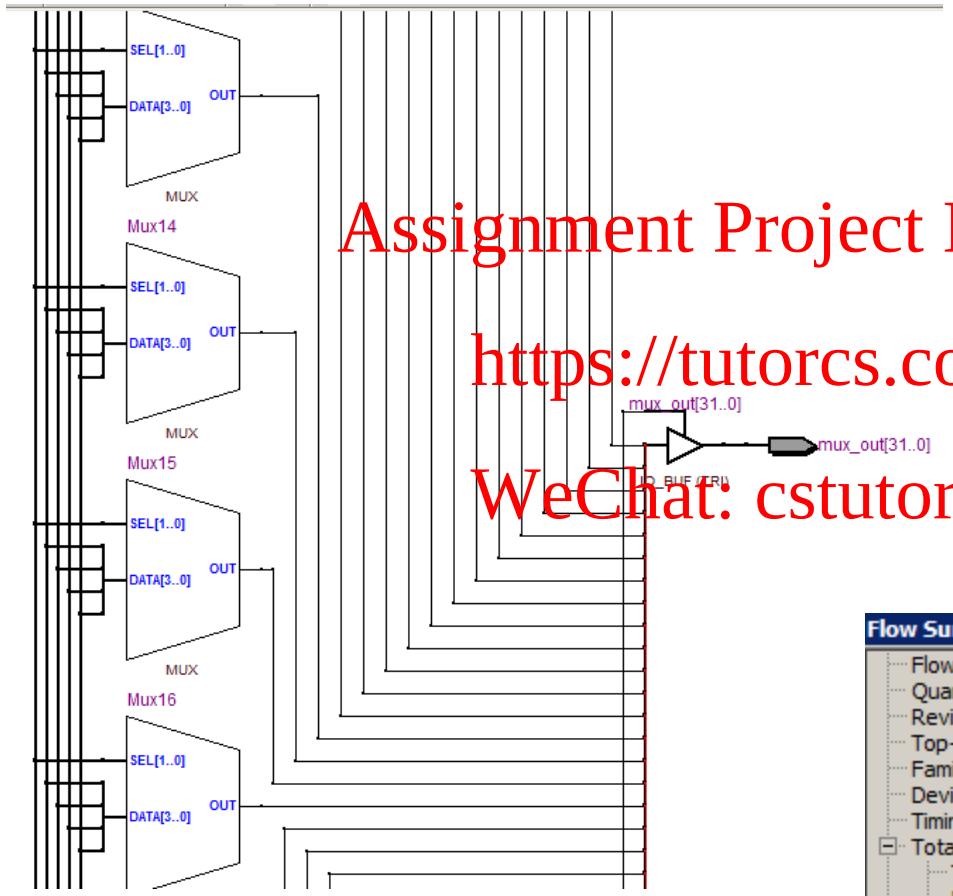


Behavioural Model of the 4-channel 32-bit Multiplexer

```
module Mux_4_32_case (mux_out, data_3, data_2, data_1, data_0, select,
    enable);
output   [31:0]  mux_out;
input    [31:0]  data_3, data_2, data_1, data_0;  [1:0] select ;
input    enable;
reg      [31:0] mux_int;
assign   mux_out = enable ? mux_int : 32'bz;
always @ (data_3 or data_2 or data_1 or data_0 or select)
    case (select)
        0:          mux_int = data_0;
        1:          mux_int = data_1;
        2:          mux_int = data_2;
        3:          mux_int = data_3;
        default     mux_int = 32'bx;
    endcase
endmodule
```

Synthesised Design



Flow Summary

Flow Status	Successful - Mon Oct 10 10:19:23 2011
Quartus II 64-Bit Version	11.0 Build 208 07/03/2011 SP 1 SJ Full Version
Revision Name	Mux_4_32_case
Top-level Entity Name	Mux_4_32_case
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	64 / 33,216 (< 1 %)
Total combinational functions	64 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	163 / 475 (34 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

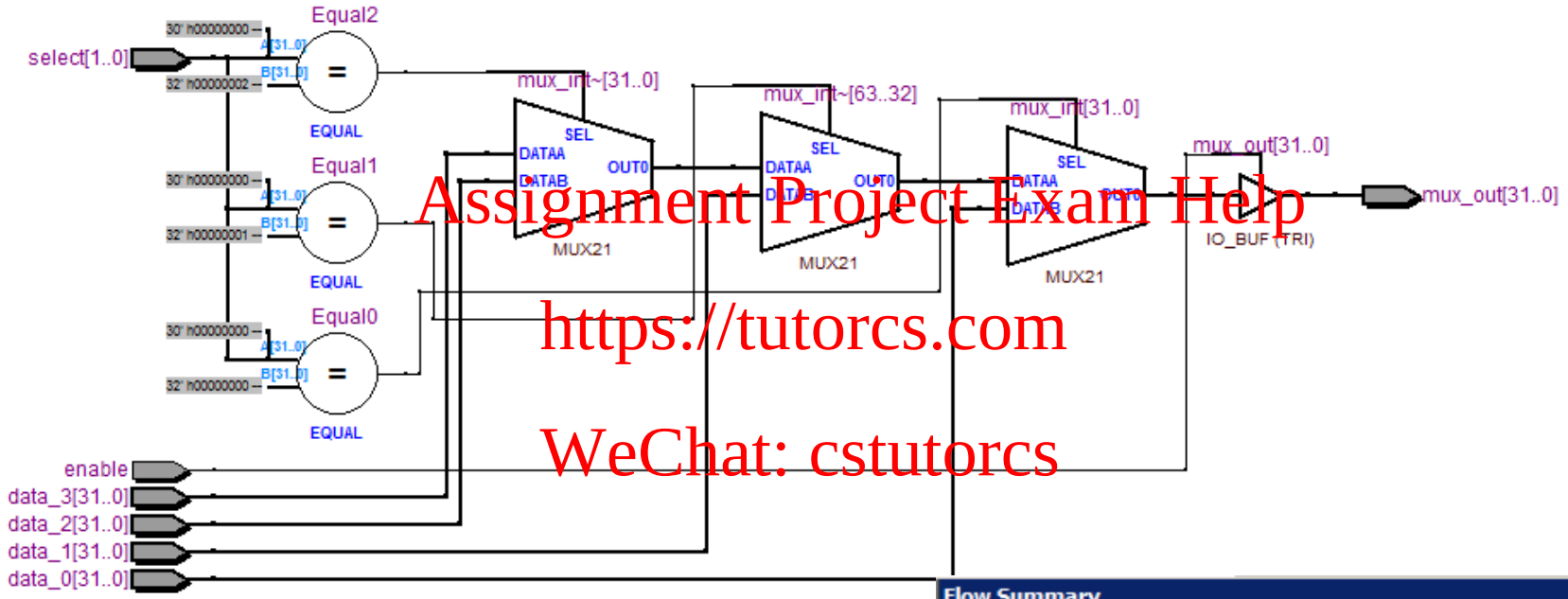
Multiplexer Model using Nested Conditional Statements

```
module Mux_4_32_if (mux_out, data_3, data_2, data_1, data_0, select,
    enable);
output   [31:0]   mux_out;
input     [31:0]  data_3, data_2, data_1, data_0;  [1:0] select ;
input     enable;
reg       [31:0]   mux_int;

assign   mux_out = enable ? mux_int : 32'bz;

always @ (data_3 or data_2 or data_1 or data_0 or select)
    if (select ==0) mux_int = data_0; else
       if (select ==1) mux_int = data_1; else
          if (select ==2) mux_int = data_2; else
             if (select ==3) mux_int = data_3; else mux_int = 32'bx;
endmodule
```

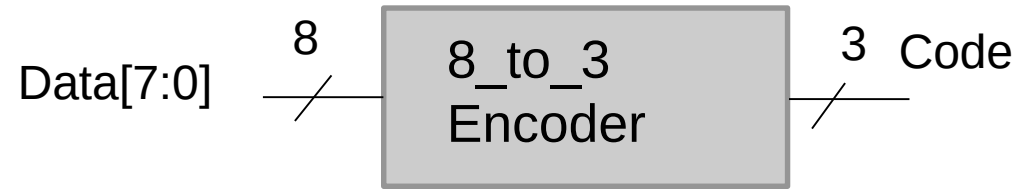
Synthesised Design



Flow Summary

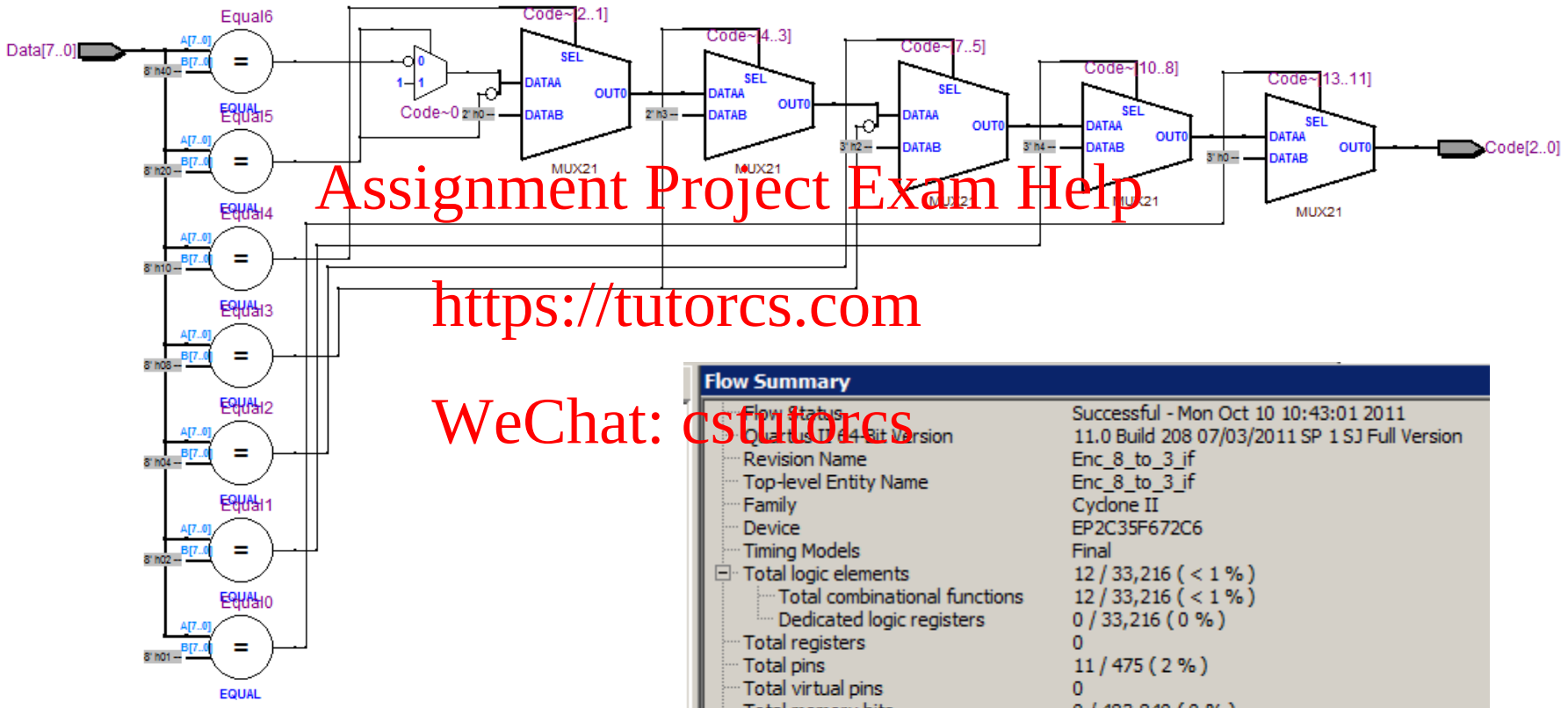
Flow Status	Successful - Mon Oct 10 10:13:14 2011
Quartus II 64-Bit Version	11.0 Build 208 07/03/2011 SP 1 SJ Full Version
Revision Name	Mux_4_32_if
Top-level Entity Name	Mux_4_32_if
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	64 / 33,216 (< 1 %)
Total combinational functions	64 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	163 / 475 (34 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Encoder




```
module   encoder (Code, Data);
output   [2:0]   Code;
Input    [7:0]   Data;
reg      [2:0]   Code;
always   @   ( Data )
    begin
      if (Data == 8'b00000001 )  Code = 3'b000; else
      if (Data == 8'b00000010 )  Code = 3'b001; else
      if (Data == 8'b00000100 )  Code = 3'b010; else
      if (Data == 8'b00001000 )  Code = 3'b011; else
      if (Data == 8'b00010000 )  Code = 3'b100; else
      if (Data == 8'b00100000 )  Code = 3'b101; else
      if (Data == 8'b01000000 )  Code = 3'b110; else
      if (Data == 8'b10000000 )  Code = 3'b111; else
          Code = 3'bx;
    end
endmodule
```

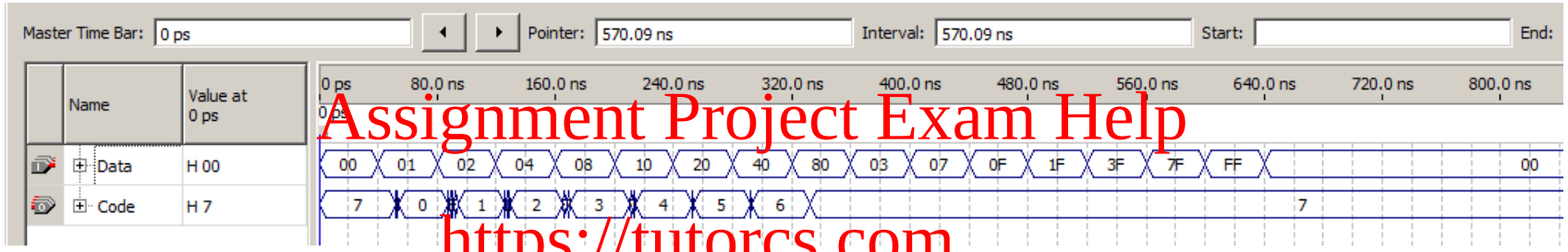
Synthesised design



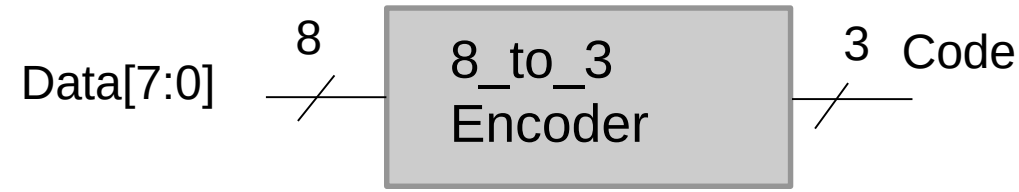
Flow Summary

Flow Status	Successful - Mon Oct 10 10:43:01 2011
Quartus II 64-Bit Version	11.0 Build 208 07/03/2011 SP 1 SJ Full Version
Revision Name	Enc_8_to_3_if
Top-level Entity Name	Enc_8_to_3_if
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	12 / 33,216 (< 1 %)
Total combinational functions	12 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	11 / 475 (2 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Simulation Results

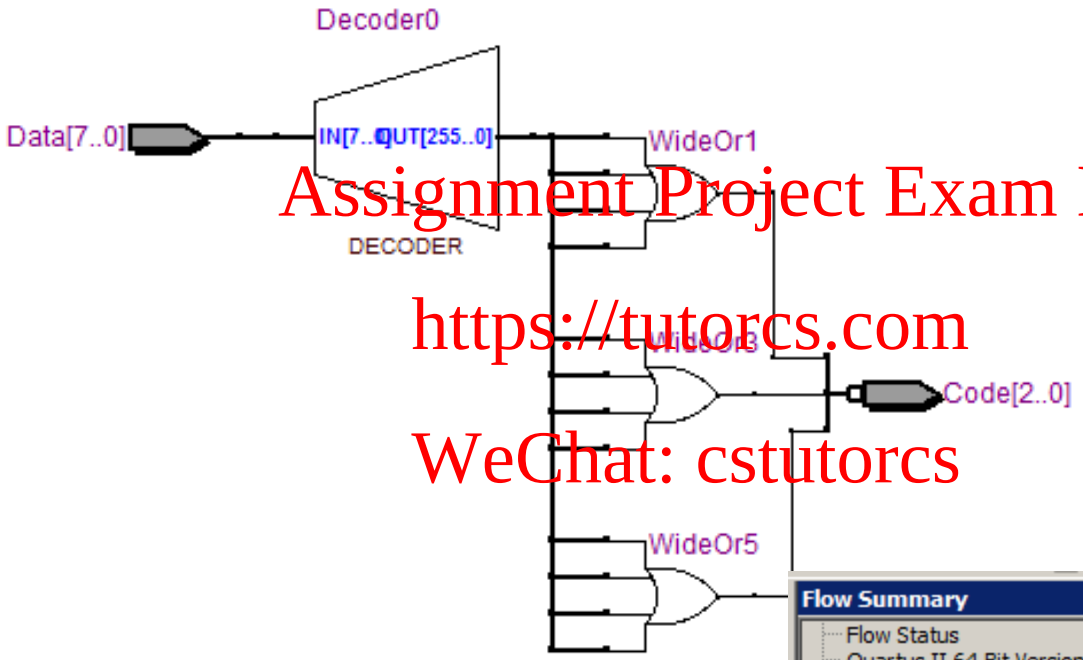


Encoder



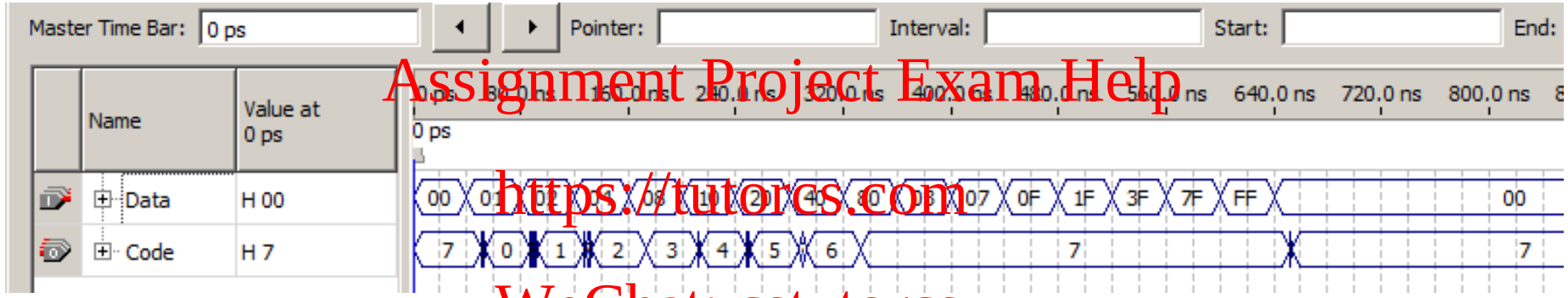

```
module   encoder (Code, Data);
output   [2:0]   Code;
input    [7:0]   Data;
reg      [2:0]   Code;
always   @   ( Data )
    case (Data)
        8'b00000001  :  Code = 3'b000;
        8'b00000010  :  Code = 3'b001;
        8'b00000100  :  Code = 3'b010;
        8'b00001000  :  Code = 3'b011;
        8'b00010000  :  Code = 3'b100;
        8'b00100000  :  Code = 3'b101;
        8'b01000000  :  Code = 3'b110;
        8'b10000000  :  Code = 3'b111;
         default          Code = 3'bx;
    endcase
endmodule
```

Synthesised design



Flow Summary	
Flow Status	Successful - Mon Oct 10 10:39:27 2011
Quartus II 64-Bit Version	11.0 Build 208 07/03/2011 SP 1 SJ Full Version
Revision Name	Enc_8_to_3_case
Top-level Entity Name	Enc_8_to_3_case
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	8 / 33,216 (< 1 %)
Total combinational functions	8 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	11 / 475 (2 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Simulation Results

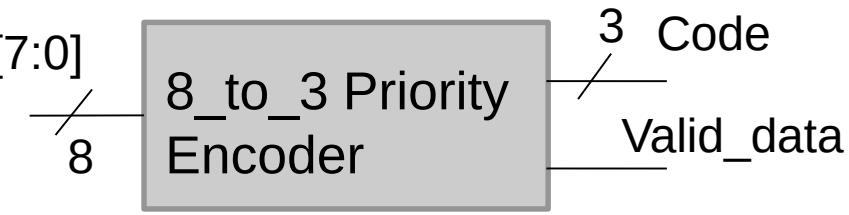


Case Statement with don’t-cares

- **casez** treats high-impedance values (z) as don’t-cares.
- **casex** treats both high-impedance (z) and unknown (x) values as don’t-cares.
- The use of **casex** and **casez** allows comparison of only non-x and non-z positions in the case expression.

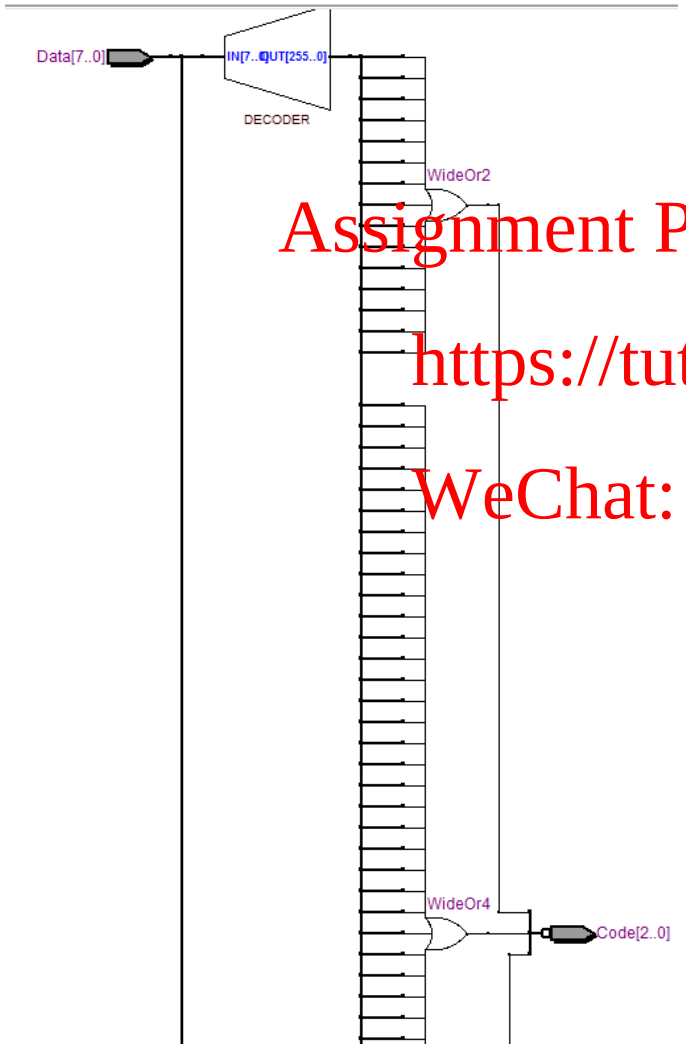

```
reg [7:0] ir;
casez (ir)
    8’b1??????? : instruction1(ir); // ? represents the possibility of z
    // The instruction1 task is called if the MSB of ir is 1.
    8’b01?????? : instruction2(ir);
    8’b00010??? : instruction3(ir);
    8’b000001?? : instruction4(ir);
endcase
```

Priority Encoder



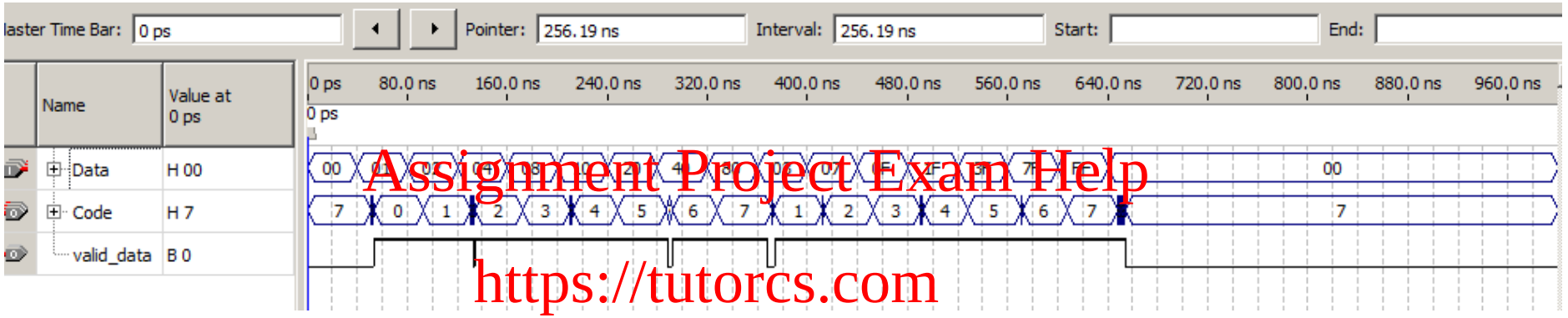

```
module   encoder (Code, valid_data, Data);
output   [2:0]      Code;
output              valid_data;
input    [7:0]      Data;
reg      [2:0]      Code;
assign   valid_data = |Data;
always   @ ( Data )
    casex (Data)
        8'b1xxxxxxx  :  Code = 7;
        8'b01xxxxxx  :  Code = 6;
        8'b001xxxxx  :  Code = 5;
        8'b0001xxxx  :  Code = 4;
        8'b00001xxx  :  Code = 3;
        8'b000001xx  :  Code = 2;
        8'b0000001x  :  Code = 1;
        8'b00000001  :  Code = 0;
         default        Code = 3'bx;
    endcase
endmodule
```

Synthesised Design

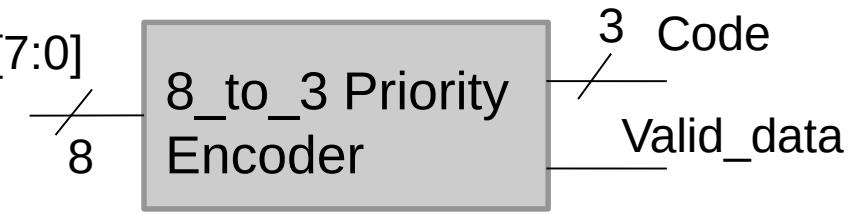


Flow Summary	
Flow Status	Successful - Mon Oct 10 11:09:02 2011
Quartus II 64-Bit Version	11.0 Build 208 07/03/2011 SP 1 SJ Full Version
Revision Name	PEnc_8_to_3_casex
Top-level Entity Name	PEnc_8_to_3_casex
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	10 / 33,216 (< 1 %)
Total combinational functions	10 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	12 / 475 (3 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Simulation Results

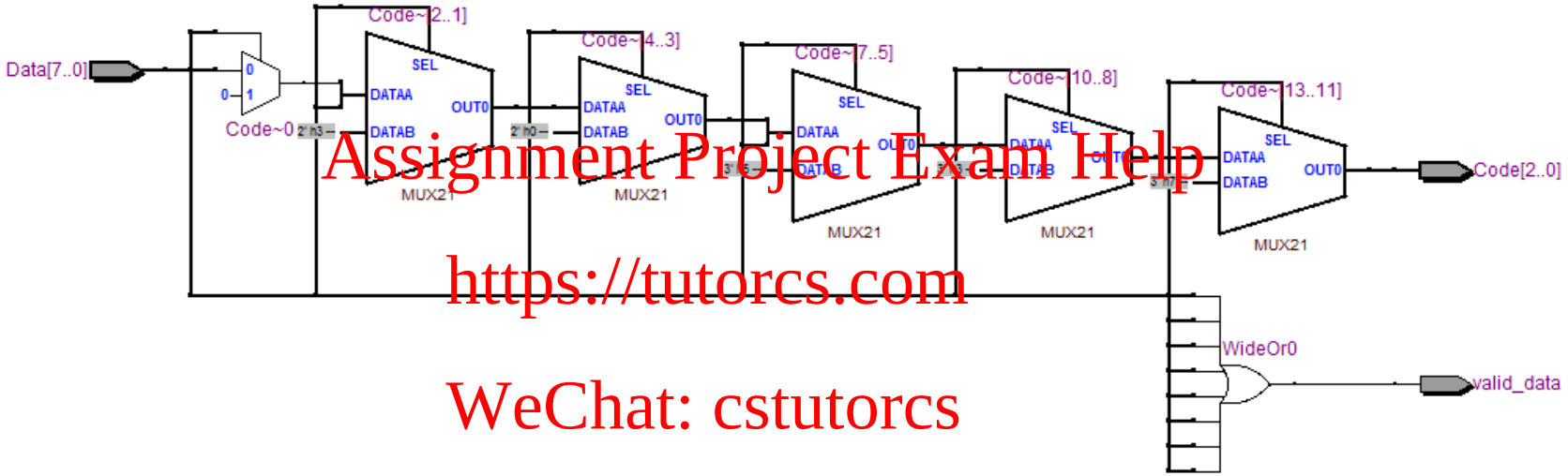


Priority Encoder




```
module   priority (Code, valid_data,  Data);
output   [2:0]       Code;
output               valid_data;
input    [7:0]       Data;
reg      [2:0]       Code;
assign   valid_data = |Data; // Reduction or operator
always   @  ( Data )
    begin
     if ( Data [7] )  Code = 7;  else
     if ( Data [6] )  Code = 6;  else
     if ( Data [5] )  Code = 5; else
     if ( Data [4] )  Code = 4; else
     if ( Data [3] )  Code = 3; else
     if ( Data [2] )  Code = 2; else
     if ( Data [1] )  Code = 1; else
     if ( Data [0] )  Code = 0; else    Code = 3'bx;
    end
endmodule
```

Synthesised Design



Flow Summary

Flow Status	Successful - Mon Oct 10 11:15:50 2011
Quartus II 64-Bit Version	11.0 Build 208 07/03/2011 SP 1 SJ Full Version
Revision Name	PEnc_8_to_3_if
Top-level Entity Name	PEnc_8_to_3_if
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	7 / 33,216 (< 1 %)
Total combinational functions	7 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	12 / 475 (3 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Simulation Results