Digital Systems Design ELEC373/473

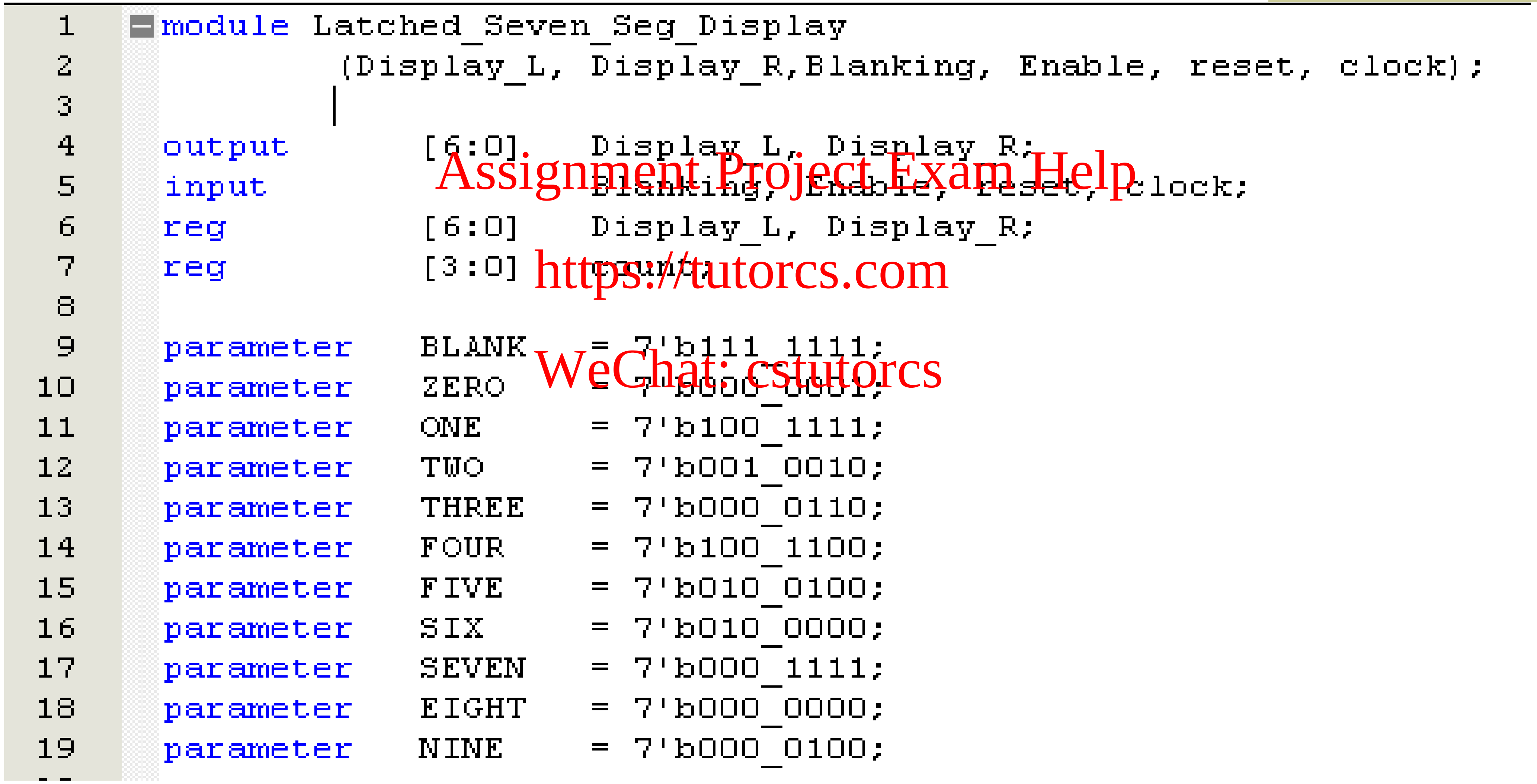
Synthesis Tips

Extracted from Verilog HDL by M.D. Ciletti

Exploiting Logical Don’t-Cares in Synthesis

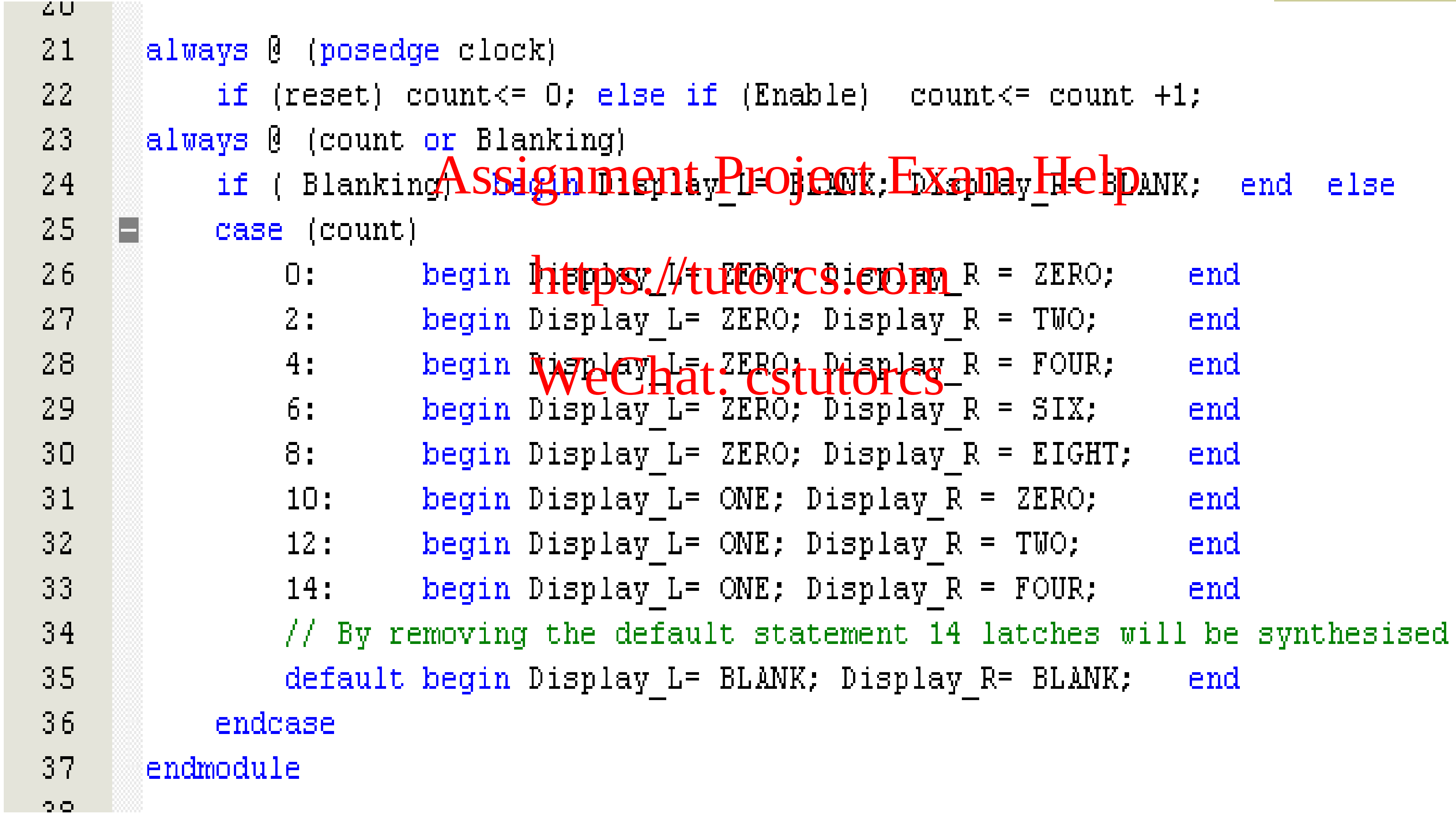
- The absence of a “**default**” assignment in “case” statements, and incomplete “**if**” statements will cause latches to be instantiated by a synthesis tool. (Example 1)

Example 1 – Counter & Seven segment decoder

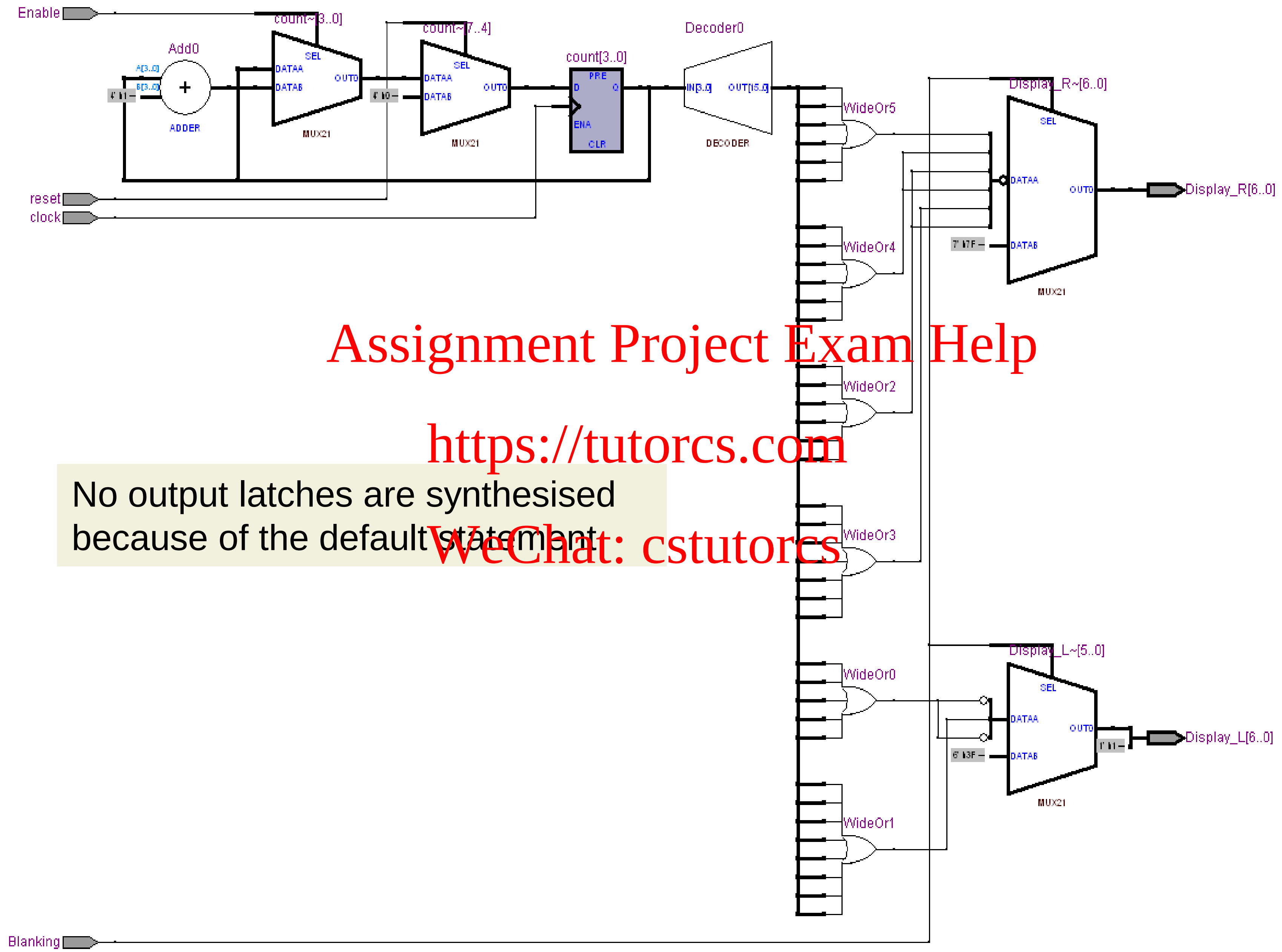


```
module Latched_Seven_Seg_Display
            (Display_L, Display_R,Blanking, Enable, reset, clock);

output          [6:0]    Display_L, Display_R;
input                    Blanking, Enable, reset, clock;
reg             [6:0]    Display_L, Display_R;
reg             [3:0]    count;

parameter       BLANK    = 7'b111_1111;
parameter       ZERO     = 7'b000_0001;
parameter       ONE      = 7'b100_1111;
parameter       TWO      = 7'b001_0010;
parameter       THREE    = 7'b000_0110;
parameter       FOUR     = 7'b100_1100;
parameter       FIVE     = 7'b010_0100;
parameter       SIX      = 7'b010_0000;
parameter       SEVEN    = 7'b000_1111;
parameter       EIGHT    = 7'b000_0000;
parameter       NINE     = 7'b000_0100;
```

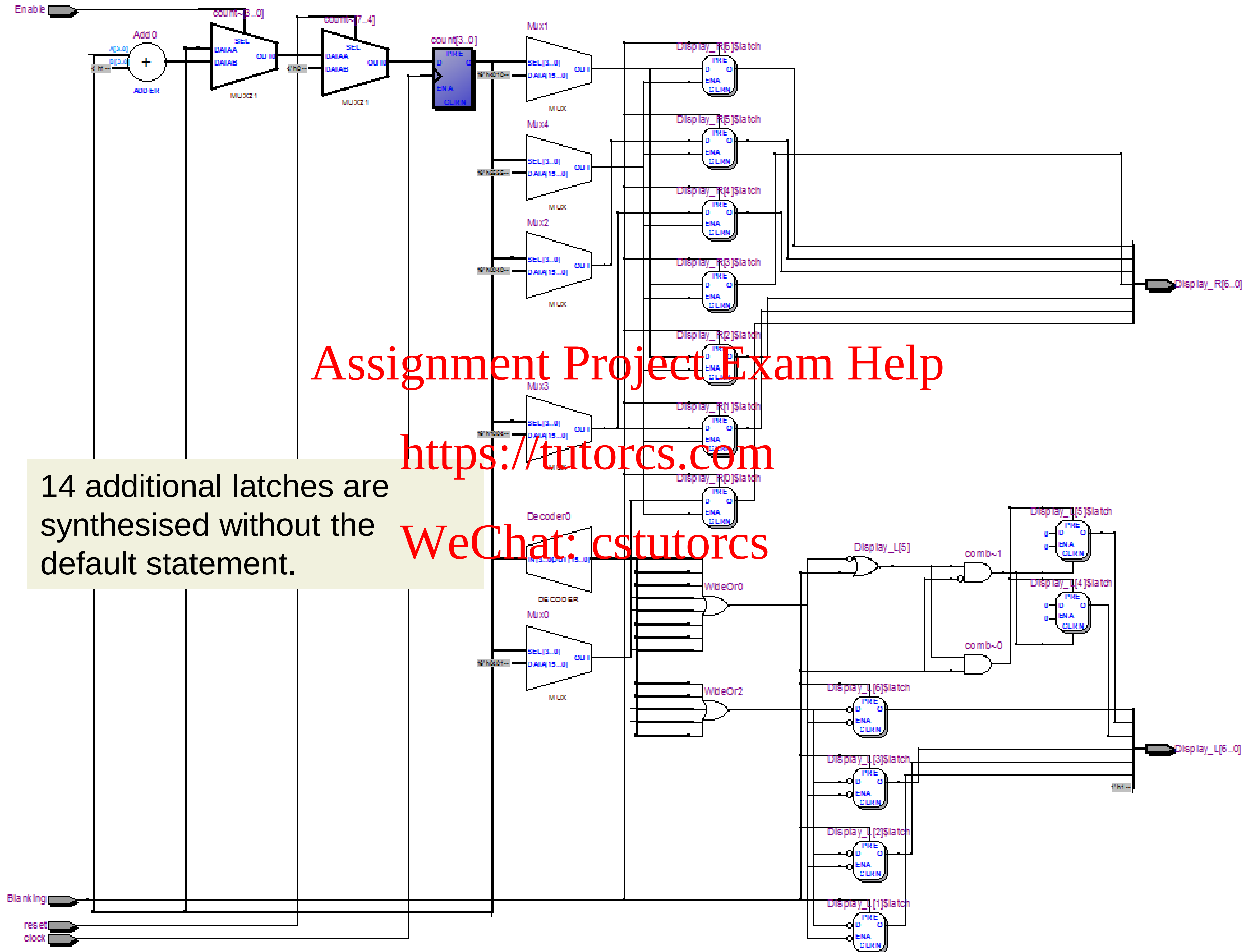
Example 1 (cont.)



```
always @ (posedge clock)
    if (reset) count<= 0; else if (Enable)  count<= count +1;
always @ (count or Blanking)
    if ( Blanking) begin Display_L= BLANK; Display_R= BLANK;  end  else
    case (count)
        0:      begin Display_L= ZERO; Display_R = ZERO;    end
        2:      begin Display_L= ZERO; Display_R = TWO;     end
        4:      begin Display_L= ZERO; Display_R = FOUR;    end
        6:      begin Display_L= ZERO; Display_R = SIX;     end
        8:      begin Display_L= ZERO; Display_R = EIGHT;   end
        10:     begin Display_L= ONE; Display_R = ZERO;     end
        12:     begin Display_L= ONE; Display_R = TWO;      end
        14:     begin Display_L= ONE; Display_R = FOUR;     end
        // By removing the default statement 14 latches will be synthesised
        default begin Display_L= BLANK; Display_R= BLANK;   end
    endcase
endmodule
```



No output latches are synthesised because of the default statement



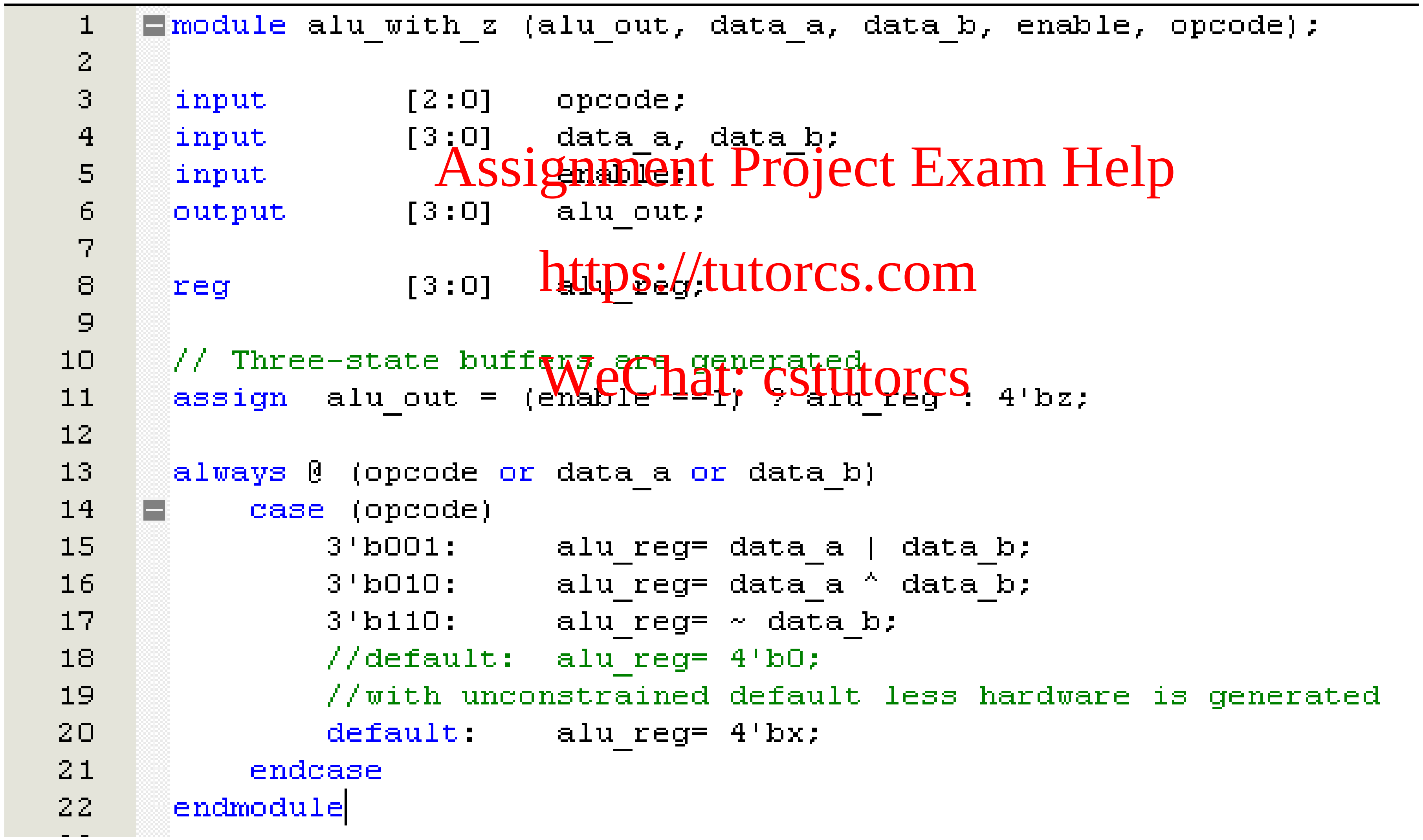
14 additional latches are synthesised without the default statement.

Don’t cares & Tri-State Outputs

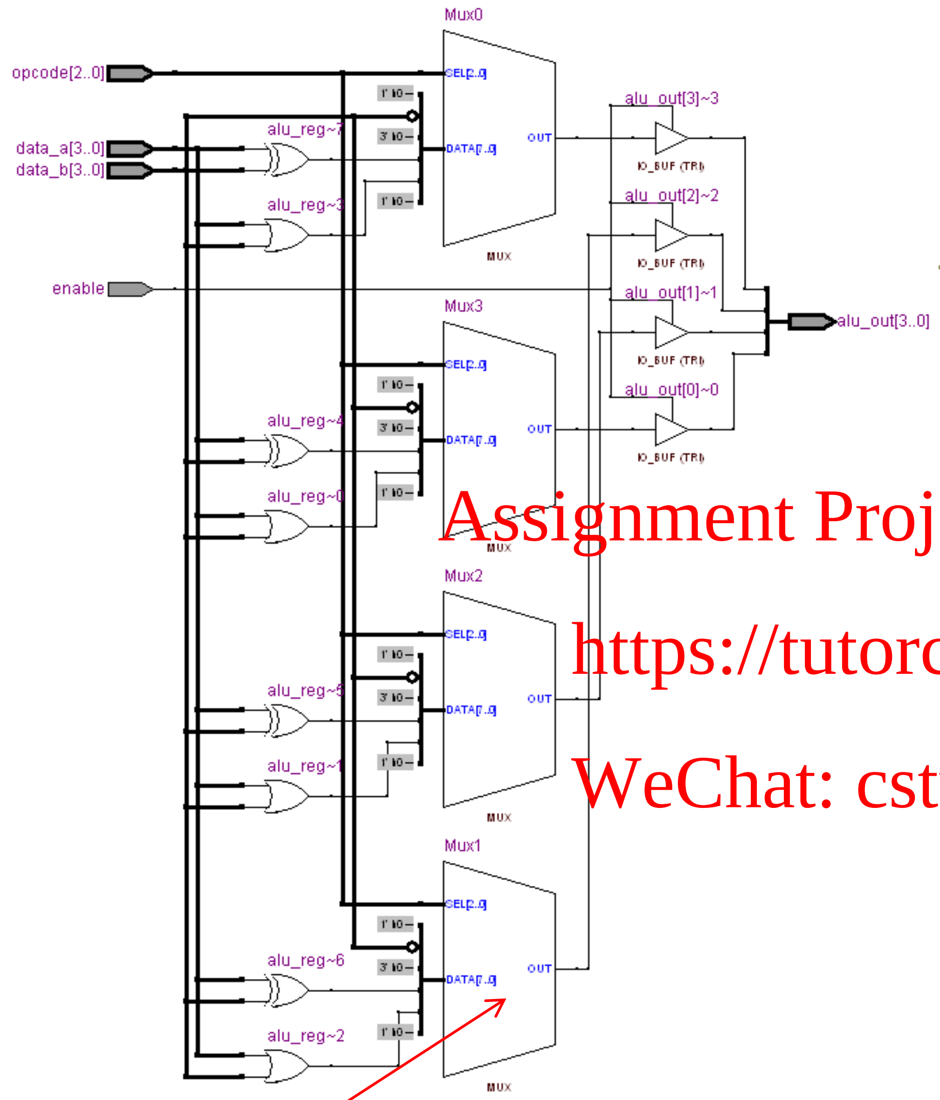
- An assignment to “**x**” in a “**case”** or an “**if”** assignment will be treated as a don’t-care condition in synthesis. This will be used in logic minimisation. (Example 2)

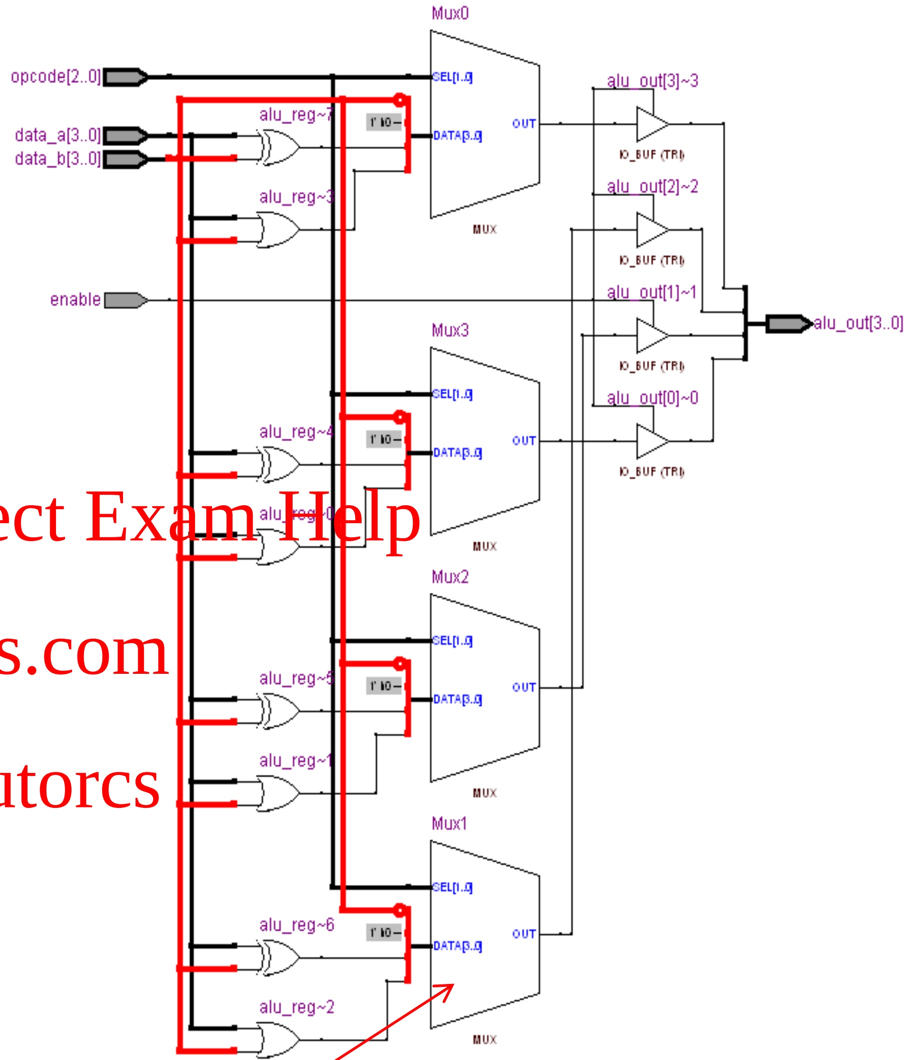
- If a conditional operator assigns the value **“z”** to the right hand side expression of a continuous assignment in a level sensitive behaviour, the statement will synthesise to a three-state device driven by combinational logic. (Example 2)

Example 2



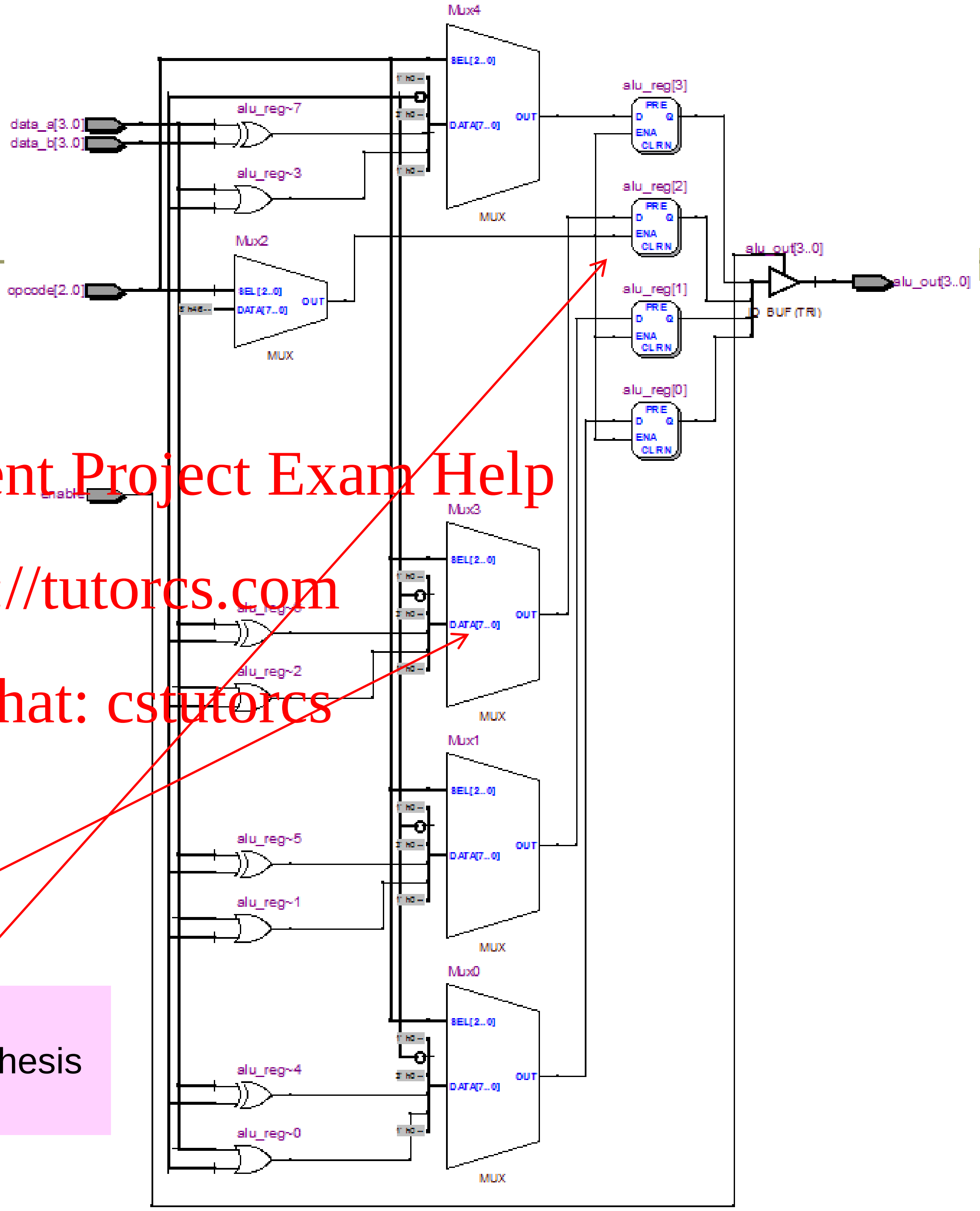
```
module alu_with_z (alu_out, data_a, data_b, enable, opcode);

input         [2:0]   opcode;
input         [3:0]   data_a, data_b;
input                 enable;
output        [3:0]   alu_out;

reg           [3:0]   alu_reg;

// Three-state buffers are generated
assign  alu_out = (enable ==1) ? alu_reg : 4'bz;

always @ (opcode or data_a or data_b)
    case (opcode)
        3'b001:     alu_reg= data_a | data_b;
        3'b010:     alu_reg= data_a ^ data_b;
        3'b110:     alu_reg= ~ data_b;
        //default:  alu_reg= 4'b0;
        //with unconstrained default less hardware is generated
        default:    alu_reg= 4'bx;
    endcase
endmodule
```





Constrained default assignment has used **8-1 multiplexer** blocks for synthesis. 8 Logic Elements of a Cyclone II device will be used.

Unconstrained default assignment has used **4-1 multiplexer** blocks for synthesis. 4 Logic Elements of a Cyclone II device will be used.



With no default assignment has used **8-1 multiplexer** blocks for synthesis and also synthesised latches.

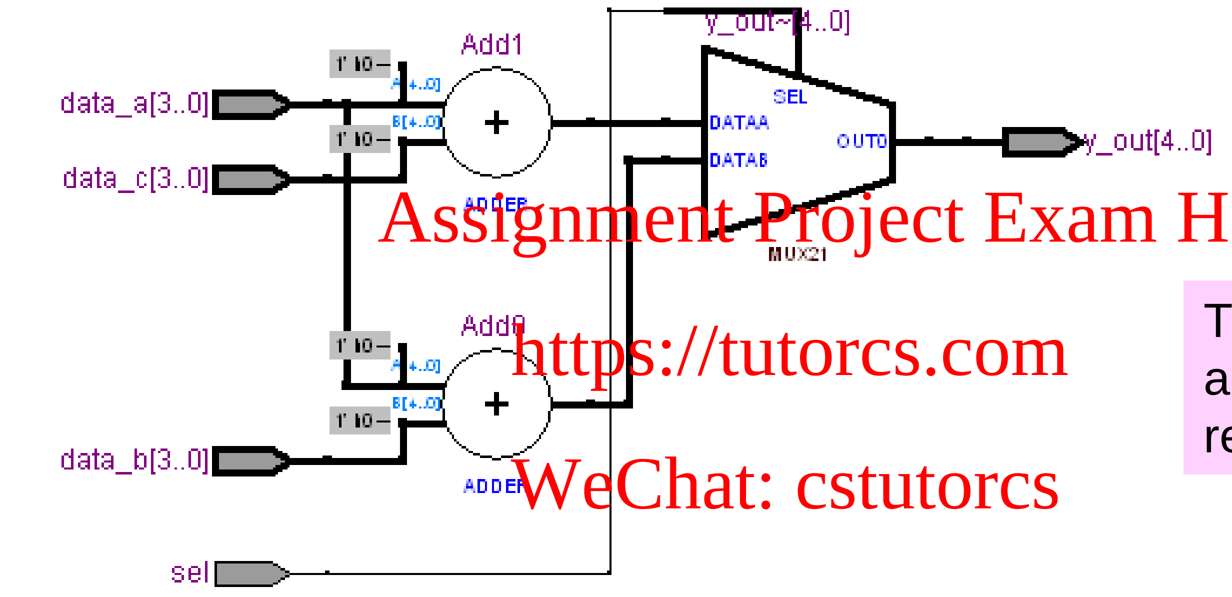
Resource Sharing

- A synthesis tool must recognise whether the physical resources required to implement complex behaviours (e.g. Adders) can be shared.
- If the data flows within a behaviour do not conflict, the resource can be shared between one or more paths.

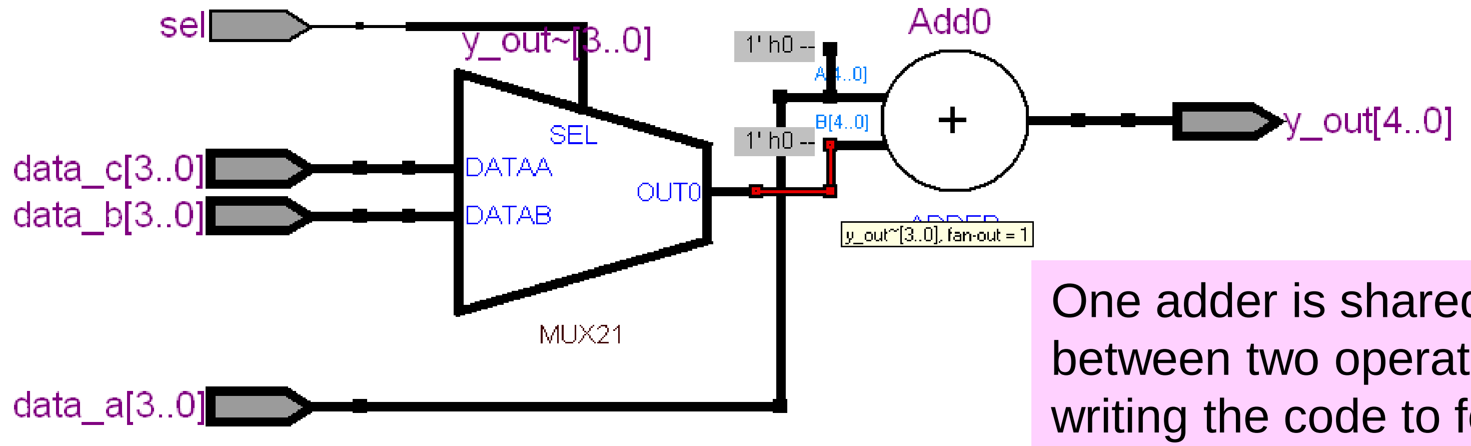
```
module res_share  (y_out, sel, data_a, data_b, data_c);
        output          [4:0]       y_out;
        input           [3:0]       data_a, data_b, data_c;
        input           sel;
        // The synthesis tool couldn't detect the resource sharing
        // assign       y_out  =  sel ?  (data_a +  data_b) : (data_a + data_c);
        //  Forced resource sharing
        assign          y_out  =  data_a + ( sel ?  data_b : data_c);
endmodule
```

The important design trade off is that the **mux** will use significantly less resources than the adder that it replaces. (Depends on the Architecture)

Forced Resource Sharing



The compiler did not automatically implement resource sharing.



One adder is shared between two operations by writing the code to force it.

Synthesis of Sequential Logic with Latches

- A set of feedback-free continuous assignments will synthesise into latch free combinational logic.

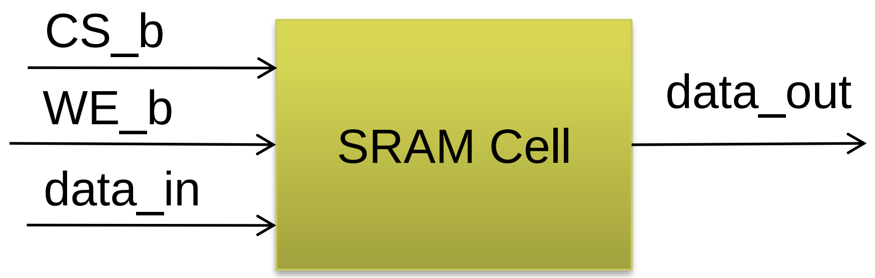
- A continuous assignment using a conditional operator with feedback will synthesise into a latch.

```
assign data_out = ( CS_b ==0) ? (WE_b ==0) ? data_in : data_out  : 1'bz;
```

This is better written as

```
assign data_out = ( CS_b ==0) ? ((WE_b ==0) ? data_in : data_out ) : 1'bz;
```

The above assignment synthesises an SRAM cell

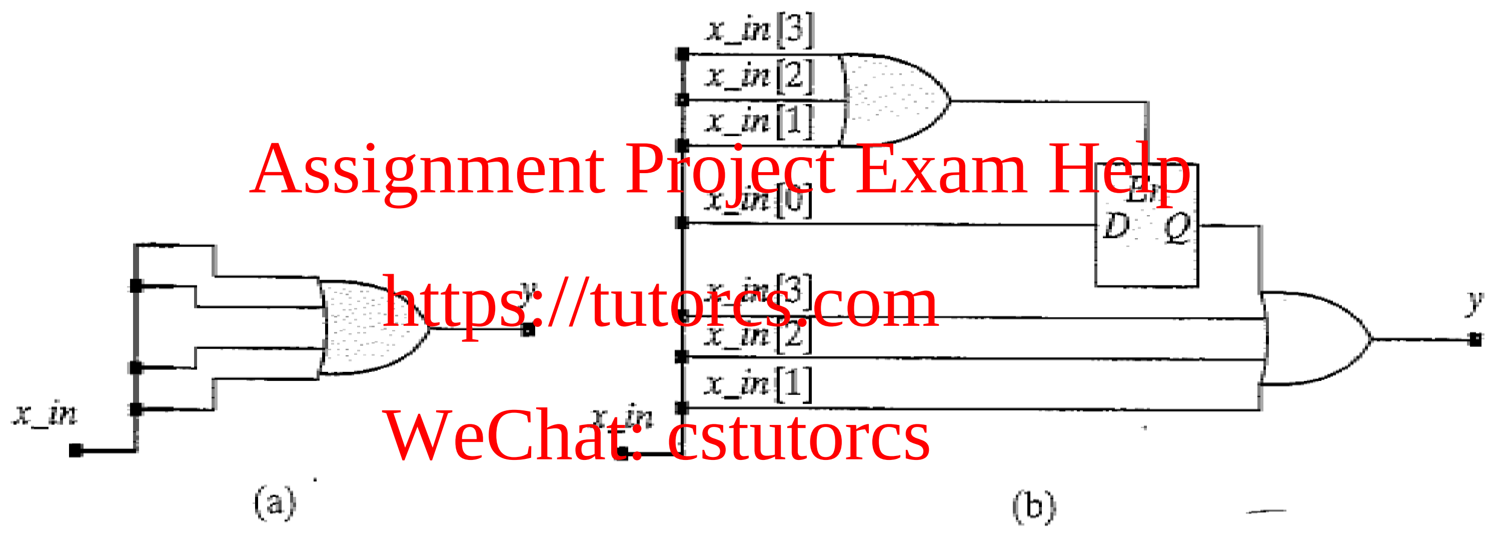


Accidental Synthesis of Latches

- A Verilog description of combinational logic must assign values to the outputs for all possible values of the inputs.
- The event control expression must be sensitive to all the inputs otherwise unintentional latches are synthesised.

```
module or4_behav      (y, x_in);
        output                  y;
        input        [3:0]      x_in;
        reg                     y;      // No register is synthesised for y
        integer                 k;
        always @  (x_in)        //  Only a 4-input OR gate is synthesised
        // always @  (x_in [3:1])       // If x_in[0] input is excluded from the sensitivity list of the
        //  cyclic behaviour an unintentional latch for x_in[0] is synthesised too.
                begin: check_for_1
                      y=0;
                      for (k = 0; k <= 3; k = k+1)
                                if (x_in[k] ==1 )
                                        begin  y =1;    disable   check_for_1;     end
                end
endmodule
```

Synthesised Circuits



- (a) Circuit synthesised from a level-sensitive cyclic behaviour
- (b) Circuit synthesised from a latch-inducing model

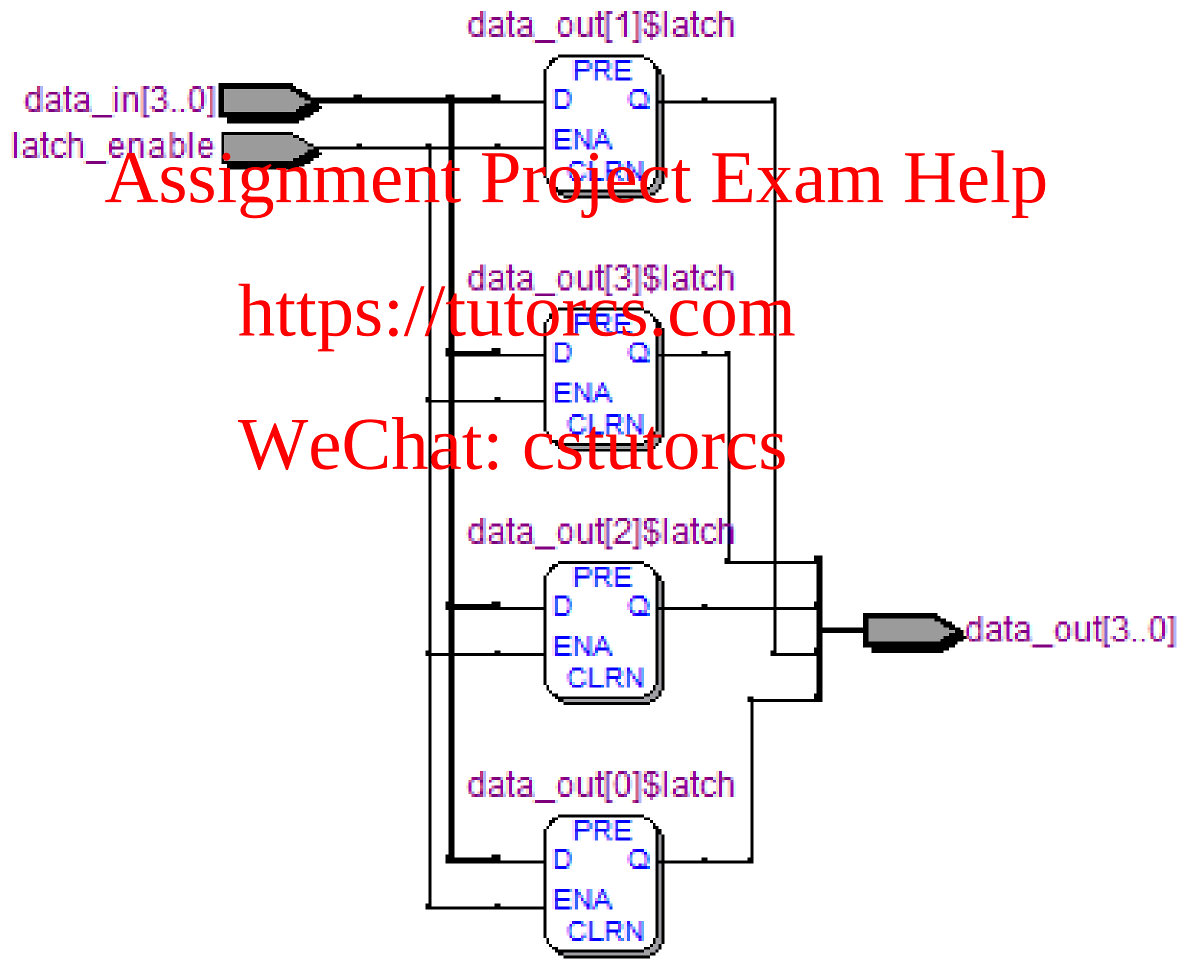
Note: Quartus will actually synthesise circuit (a) for either definition.
However not all synthesisers behave the same.

Intentional Synthesis of Latches

- An **if** statement in a level-sensitive behaviour will synthesise to a latch if the statement assigns value to a register variable in some, but not all, branches (i.e., the statement is incomplete).

```
module   latch_if2          (data_out, data_in, latch_enable);
      output        [3:0]     data_out;
      input         [3:0]     data_in;
      input                   latch_enable;
      reg           [3:0]     data_out;

      always @ (latch_enable  or  data_in)
           if (latch_enable)  data_out  = data_in; // incompletely specified
endmodule
```

Synthesised Circuit



Synthesis of Loops

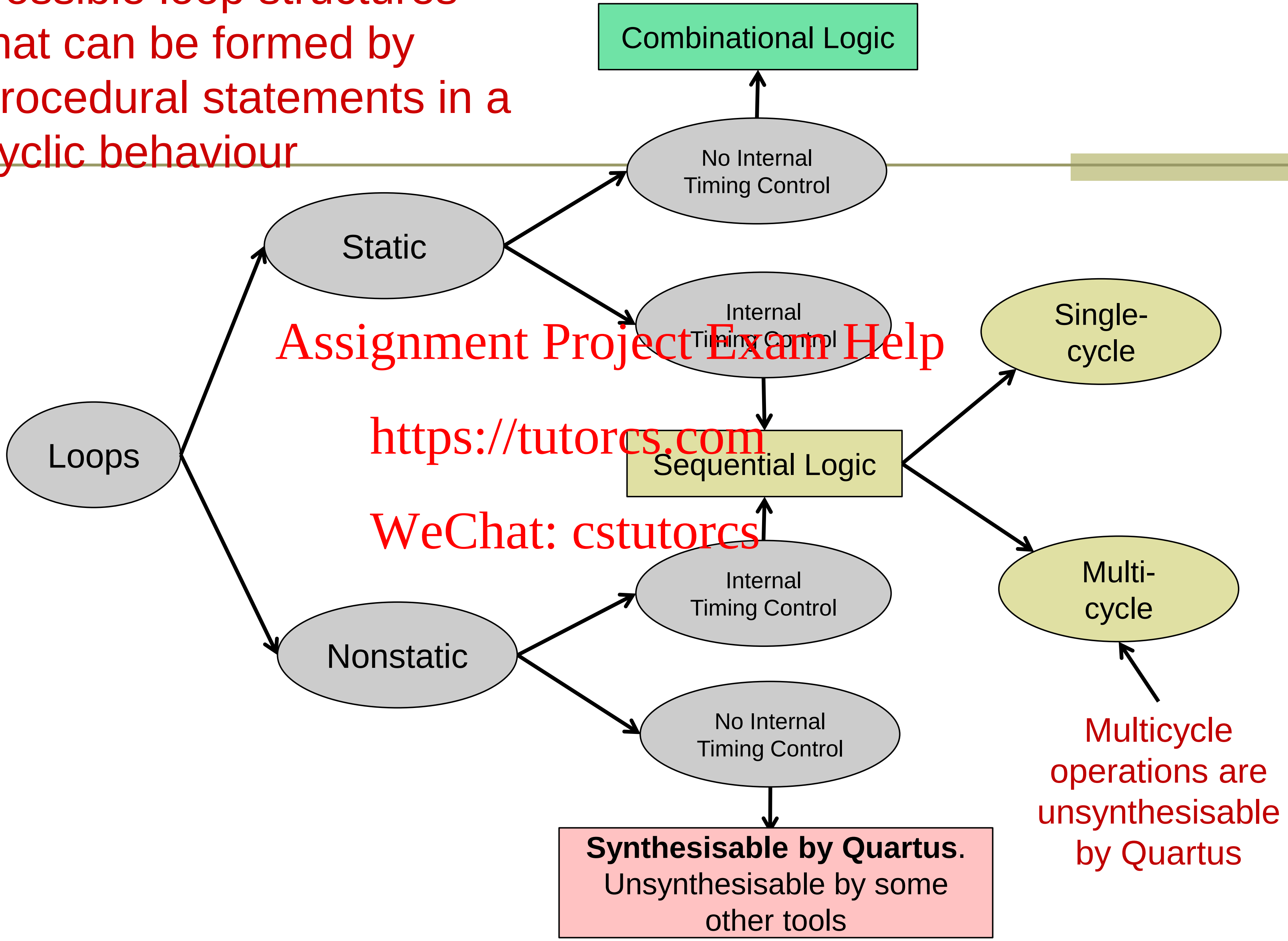
- In static or data-independent loops the number of iterations can be determined by the compiler before simulation.

- In nonstatic loops the number of iterations depends on some variable during operation.

- In addition to having a dependency on data, a loop may have a dependency on embedded timing controls.

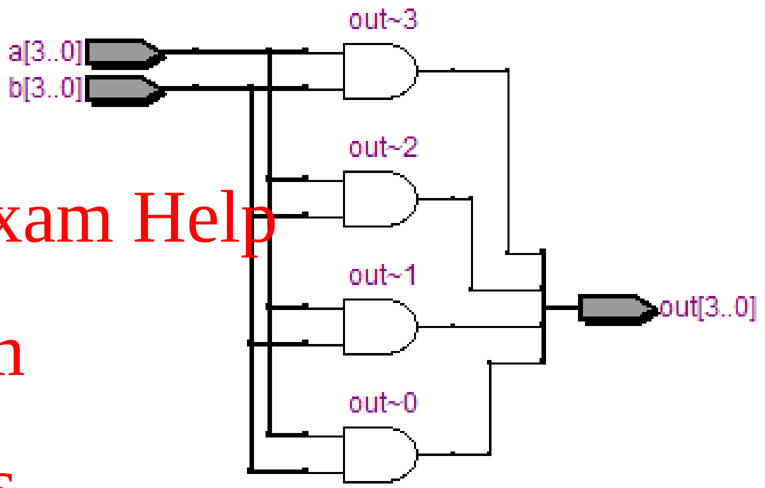
- Nonstatic loops that do not have internal timing controls are not synthesisable by some tools.

Possible loop structures that can be formed by procedural statements in a cyclic behaviour


Combinational Logic
No Internal Timing Control
Static
Internal Timing Control
Single-cycle
Loops
Sequential Logic
Internal Timing Control
Multi-cycle
Nonstatic
No Internal Timing Control
Synthesisable by Quartus. Unsynthesisable by some other tools
Multicycle operations are unsynthesisable by Quartus

Static Loops without Embedded Timing Controls

```
module for_and_loop_com  (out, a,b);
    output      [3:0]     out;
    input       [3:0]     a, b;
    reg         [2:0]     i;
    reg         [3:0]     out;
    wire        [3:0]     a, b;
    always @ (a or b)
        begin
            for (i=0; i<=3; i=i+1)
                          out[i] = a[i] & b[i];
        end
endmodule
```

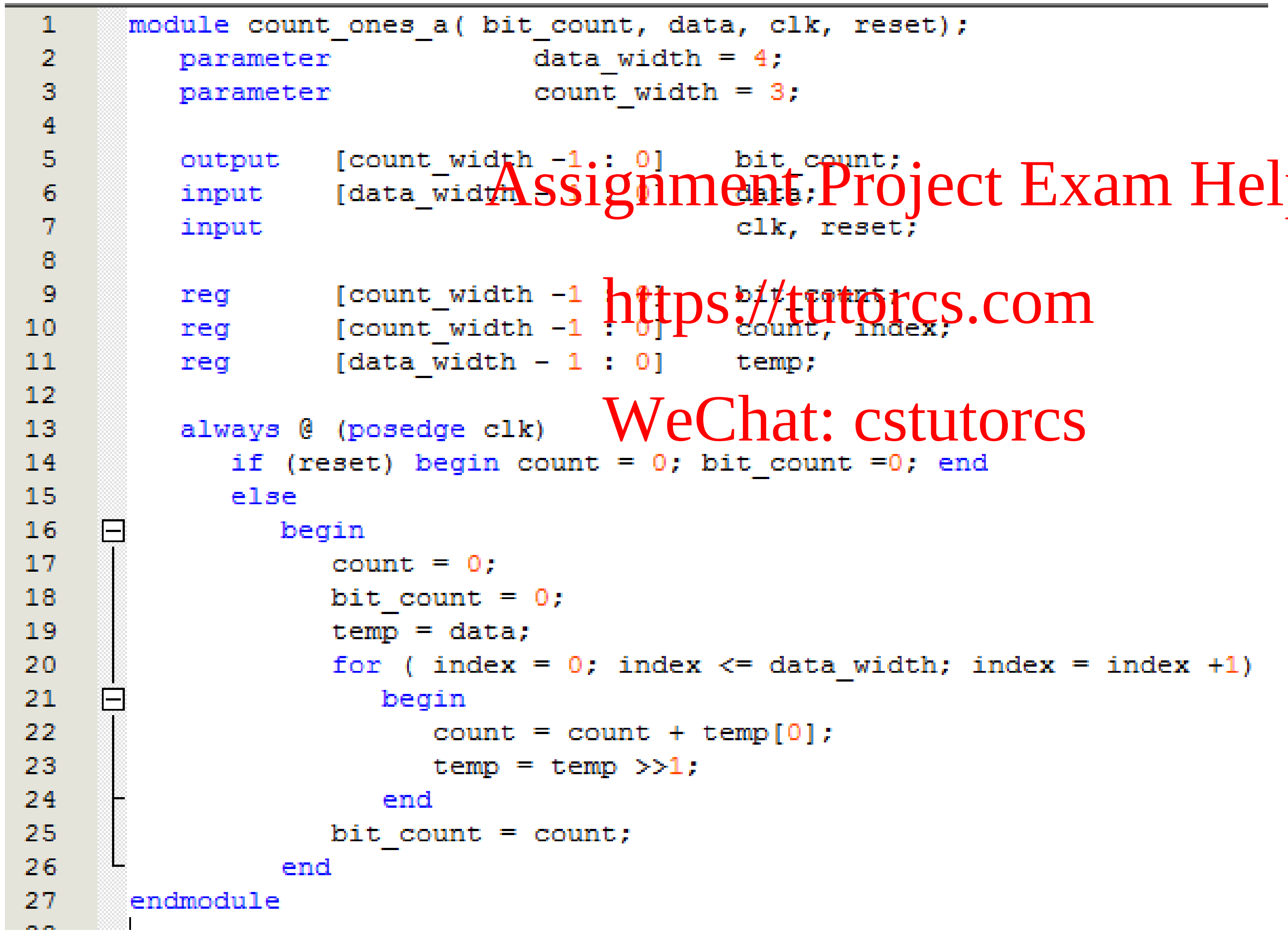


The loop does not depend on the data and does not have embedded event controls. Therefore it produces combinational logic.

Example for the next sections

- The following examples all aim to count the number of 1’s in a word received in a parallel data format.

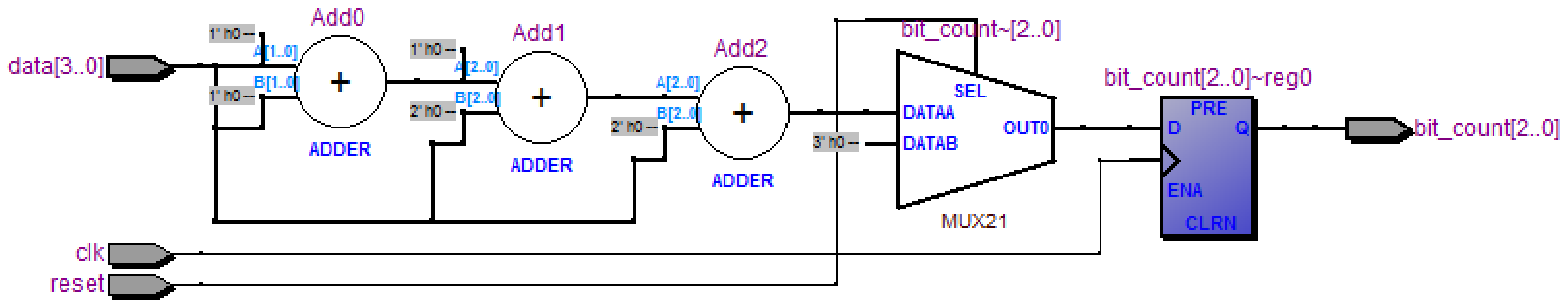
Example count_ones_a



```
module count_ones_a( bit_count, data, clk, reset);
   parameter                data_width = 4;
   parameter                count_width = 3;

   output    [count_width -1 : 0]    bit_count;
   input     [data_width - 1 : 0]    data;
   input                             clk, reset;

   reg       [count_width -1 : 0]    bit_count;
   reg       [count_width -1 : 0]    count, index;
   reg       [data_width - 1 : 0]    temp;

   always @ (posedge clk)
      if (reset) begin count = 0; bit_count =0; end
      else
         begin
            count = 0;
            bit_count = 0;
            temp = data;
            for ( index = 0; index <= data_width; index = index +1)
               begin
                  count = count + temp[0];
                  temp = temp >>1;
               end
            bit_count = count;
         end
endmodule
```

Notes on count_ones_a

- The contents of the register variables index and temp do not have a lifetime outside the cyclic behaviour in which they are assigned a value.
  - Both variables are eliminated by the synthesis tool.
- The only synthesised register is bit_count.
  - Signal bit_count is registered because its value is assigned with an edge sensitive cyclic behaviour. bit_count is also an output port.



Static Loops with Embedded Timing Controls

```
module count_ones_b2  ( bit_count, data, clk, reset);
    parameter                                   data_width = 4;
    parameter                                   count_width = 3;
    output          [count_width -1 : 0]        bit_count;
    input           [data_width - 1 : 0]        data;
    input                                       clk, reset;
    reg             [count_width -1 : 0]        count, bit_count;
    reg             [data_width - 1 : 0]        temp;
    integer                                     index;
    always begin : machine
          for ( index = 0; index <= data_width; index = index +1)
             begin
                @ (posedge clk)             // embedded timing control
                 if (reset) begin    bit_count = 0; disable machine;     end
                 else if (index == 0) begin count = 0; bit_count =0; temp = data; end
                 else if (index < data_width)
                     begin    count = count + temp[0]; temp = temp >>1; end
                  else  bit_count = count + temp[0];
              end
     end  // machine
endmodule
```

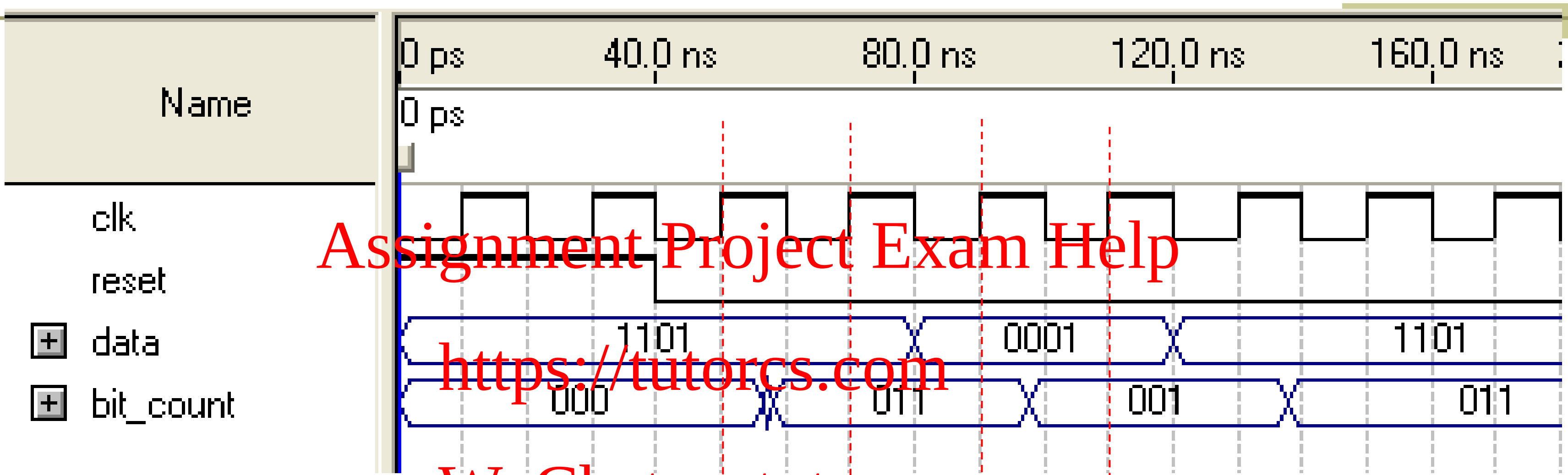
Will not synthesise in Quartus: Multiple event control statements are not supported for synthesis

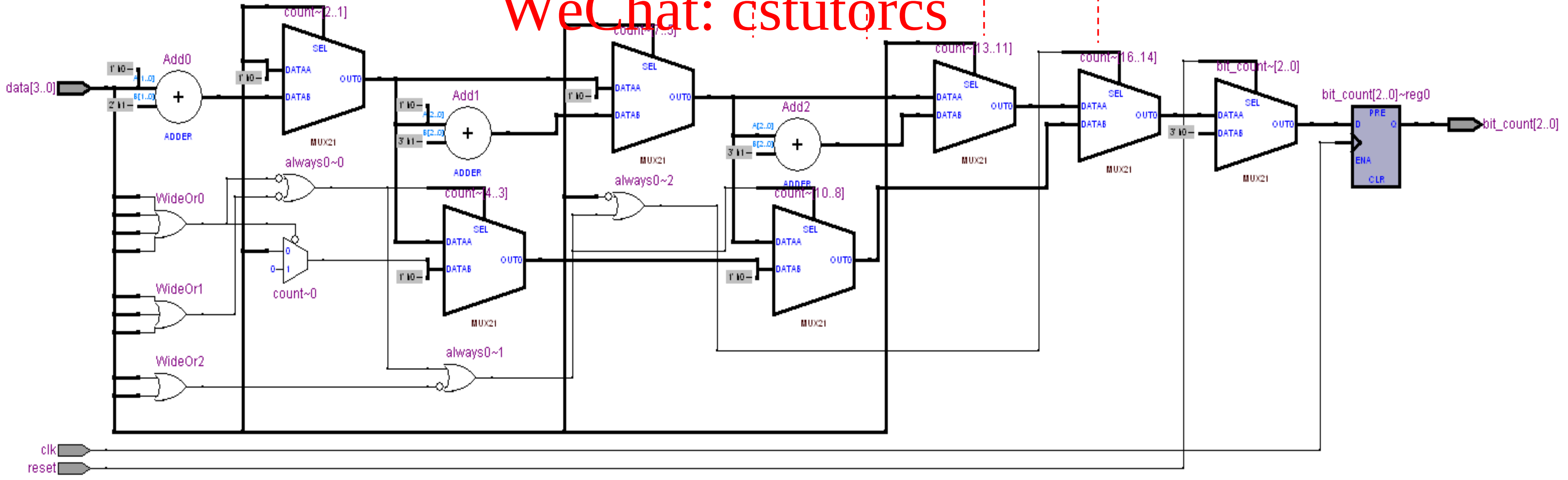
Nonstatic Loops without Embedded Timing Controls

```
module count_ones_c ( bit_count, data, clk, reset);
    parameter                                   data_width = 4;
    parameter                                   count_width = 3;
    output        [count_width -1 : 0]          bit_count;
    input         [data_width - 1 : 0]          data;
    input                                       clk, reset;
    reg           [count_width -1 : 0]          count, bit_count, index;
    reg           [data_width - 1 : 0]          temp;
    always @ (posedge clk)
        if (reset) begin count = 0; bit_count = 0; end
        else begin
            count = 0;       temp = data;
            for ( index = 0; | temp; index = index +1) begin
                if (temp[0])   count = count + 1;
                temp = temp >>1;                end
            bit_count = count;
        end
    endmodule
```

This loop's length is data dependent but it was synthesised by Quartus II

Nonstatic Loops without Embedded Timing Controls Simulation and RTL View





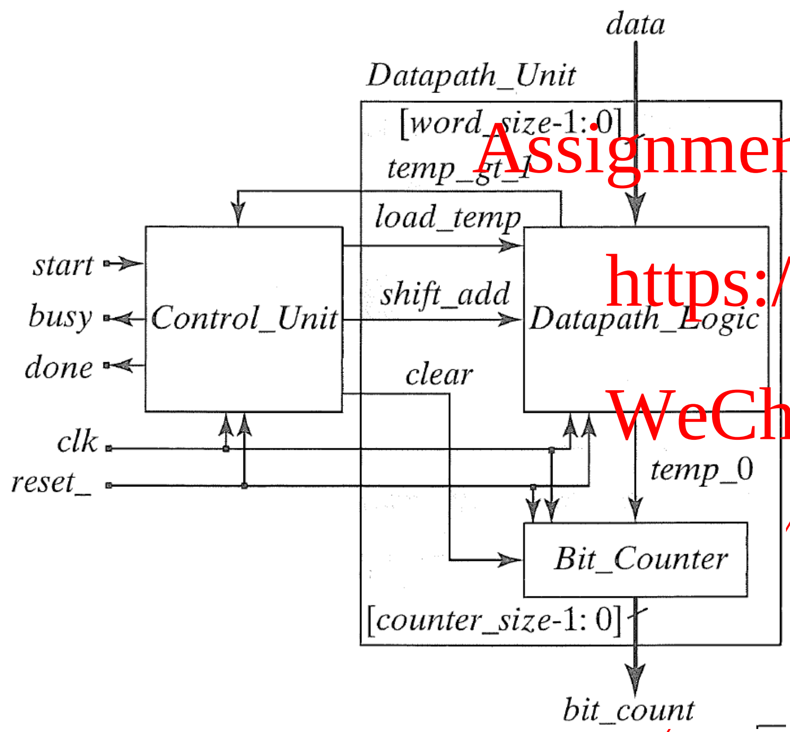
Nonstatic Loops with Embedded Timing Controls

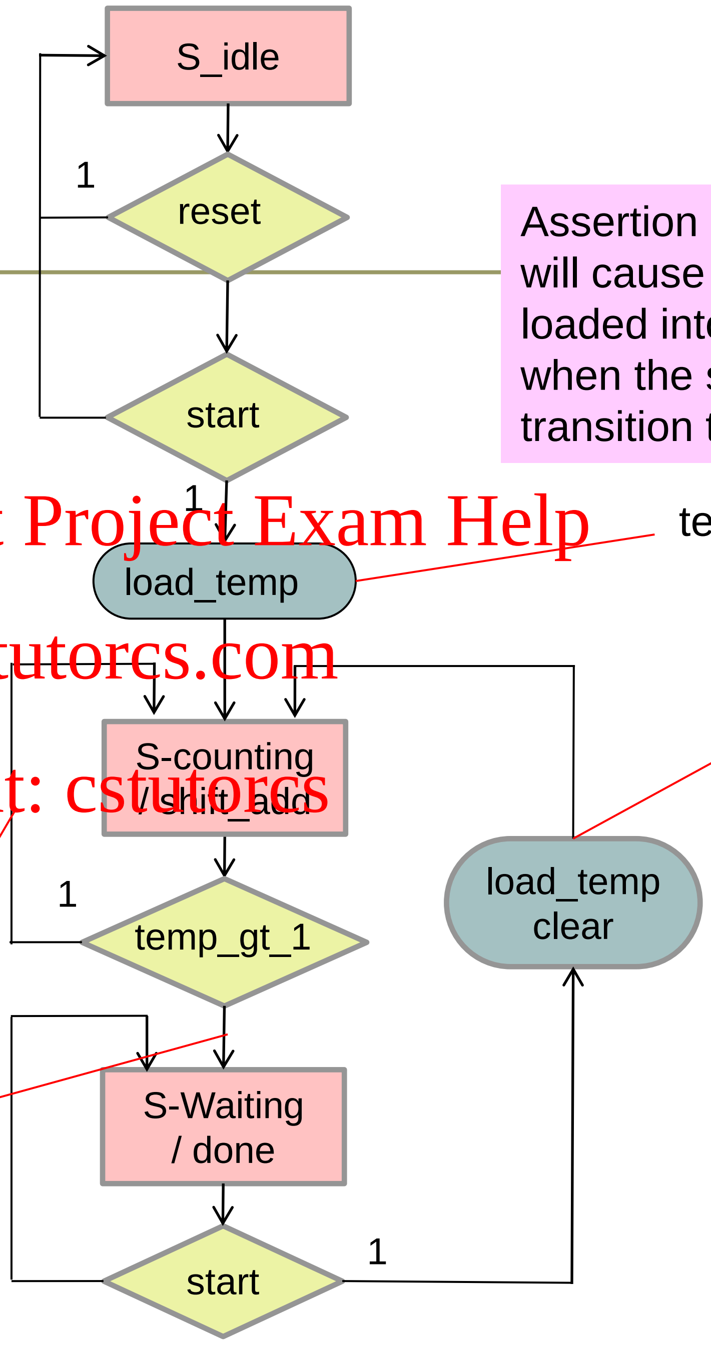
```
module count_ones_d   ( bit_count, data, clk, reset);

   parameter                        data_width = 4;
   parameter                        count_width = 3;

   output     [count_width -1 : 0]  bit_count;
   input      [data_width - 1 : 0]  data;
   input                            clk, reset;

   reg        [count_width -1 : 0]  count,   bit_count;
   reg        [data_width - 1 : 0]  temp;
```

```
    always  begin:    wrapper_for_synthesis
    @ (posedge clk)
          if (reset)    begin   count = 0;     bit_count = 0;    end
          else      begin:  bit_counter
                    count = 0;
                    temp = data;
                    while ( temp )  // nonstatic loop with embedded timing control
                     @ (posedge clk)
                          if (reset)
                             begin   count = 0;   disable bit_counter;   end
                          else
                             begin count = count + temp [0]; temp = temp >>1;  end
                    @ (posedge clk)
                    if  (reset)     begin count = 0;   disable  bit_counter;    end
                             else      bit_count = count;
                    end       // bit_counter
     end           // wrapper_for_synthesis
endmodule
```

Multiple event control statements are not supported by Quartus

ASMD replacement for unsynthesisable loops





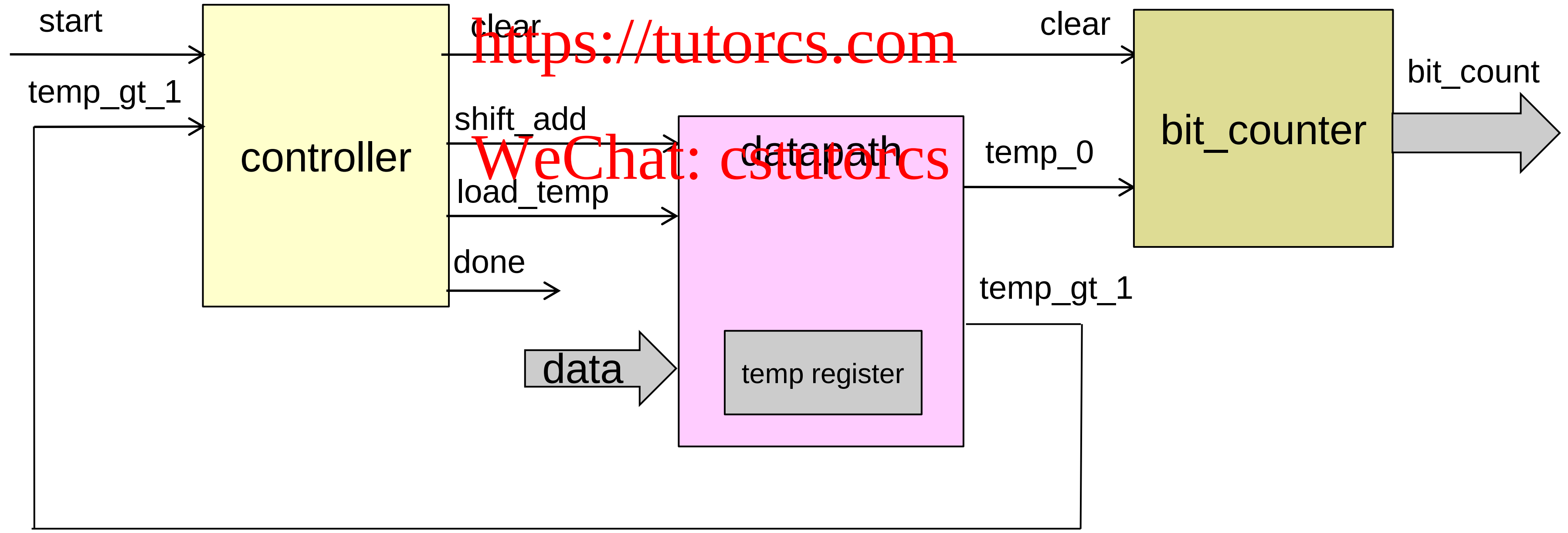
Assertion of load_temp will cause data to be loaded into register *temp* when the state makes a transition to *S_counting*.

temp <= data

bit_count<= 0
temp <= data

bit_count <= bit_count + temp[0]
temp <= temp >>1

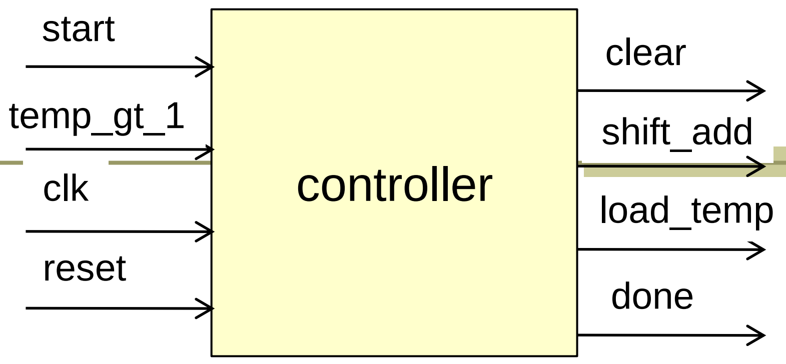
Block Diagram of the Loop



Verilog code

```
module count_ones_SM (bit_count, busy, done, data, start, clk, reset);
    parameter           word_size = 4;
    parameter           counter_size = 3;

    output              [counter_size-1:0]      bit_count;
    output              busy, done;
    input               [word_size-1: 0]        data;
    input               start, clk, reset;

    wire                load_temp, shift_add, clear;
    wire                temp_0, temp_gt_1;

    controller  M0  (load_temp, shift_add, clear, done, start, temp_gt-1, clk, reset);
    datapath    M1  (temp_gt_1, temp_0, data, load_temp, shift_add, clk, reset);
    bit_counter M2  (bit_count, temp_0, clear, clk, reset);
endmodule
```

Controller Module

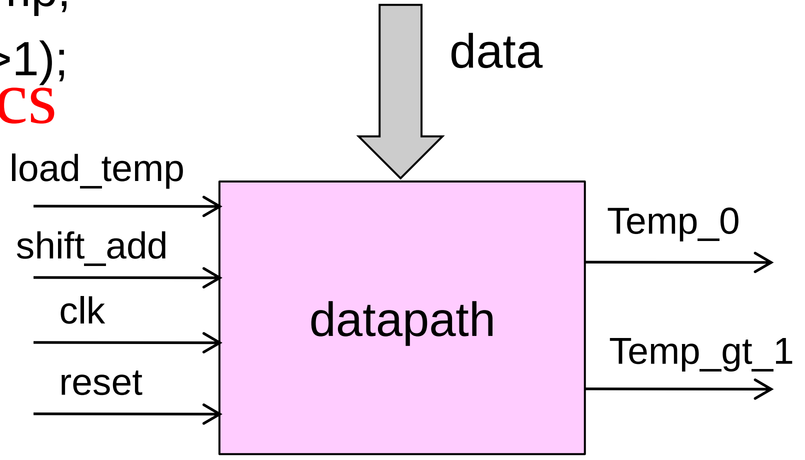


```
module controller
        ( load_temp, shift_add, clear, done, start, temp_gt_1, clk, reset);
  parameter                state_size = 2;
  parameter                S_idle = 0;
  parameter                S_counting = 1;
  parameter                S_waiting =  2;

  output                   load_temp,   shift_add,   clear,   done;
  input                    start,   temp_gt_1,   clk,    reset;

  reg   [state_size-1 : 0] state,   next_state;
  reg                      load_temp,   shift_add,  done,  clear;
```

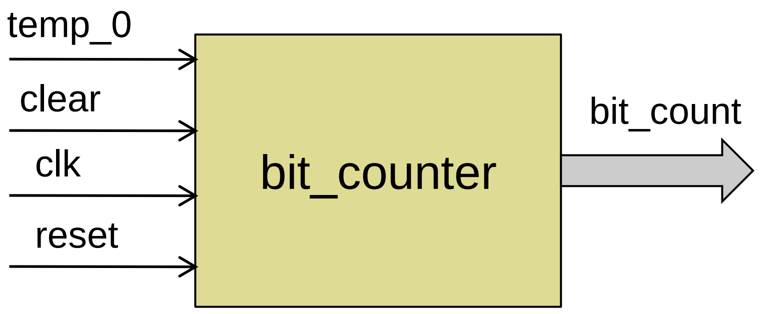
```
always @ (posedge clk)   // state transitions
        if (reset)    state= S_idle;  else              state <= next_state ;
// next state cyclic behaviour
 always @ ( state or start  or temp_gt_1 ) begin
        load_temp = 0;    shift_add = 0;    done = 0;
        clear =0;     next_state = S_idle;
        case (state)
              S_idle:            if ( start)   begin      next_state   = S_counting;
                                                          load_temp   = 1;   end
              S_counting:        begin
                                 shift_add =1;
                                 if ( temp_gt_1)         next_state = S_counting;
                                              else        next_state  = S_waiting;         end
              S_waiting:         begin
                                 done =1;
                                 if ( start)   begin      next_state = S_counting;
                                                          load_temp=1; clear =1; end
                                             else next_state = S_waiting;
                                 end
             default:            begin       clear =1;     next_state = S_idle;   end
        endcase
    end
endmodule
```

Datapath Module

```
module datapath    ( temp_gt_1, temp_0, data, load_temp, shift_add, clk, reset);
    parameter                word_size  =  4;
    output                   temp_gt_1, temp_0;
    input                    load_temp, shift_add, clk, reset;
    input                    [word_size-1 : 0]  data;
    reg                      [word_size-1 : 0]  temp;
    wire                     temp_gt_1 = (temp >1);
    wire                     temp_0 = temp[0];

    always @ (posedge clk)
        if (reset)           temp <= 0 ;
        else  begin
            if (load_temp) temp <= data;
            if (shift_add )  temp <= (temp >>1);
        end
endmodule
```



```
module   bit_counter   ( bit_count, temp_0, clear, clk, reset);
    parameter                                       counter_size = 3;
    output          [ counter_size - 1 : 0]         bit_count;
    input                                           temp_0, clear, clk, reset;

    reg             [ counter_size - 1 : 0]         bit_count;

    always @ (posedge clk)
          if (reset || clear)   bit_count <= 0 ;
          else                  bit_count <= bit_count + temp_0;
endmodule
```

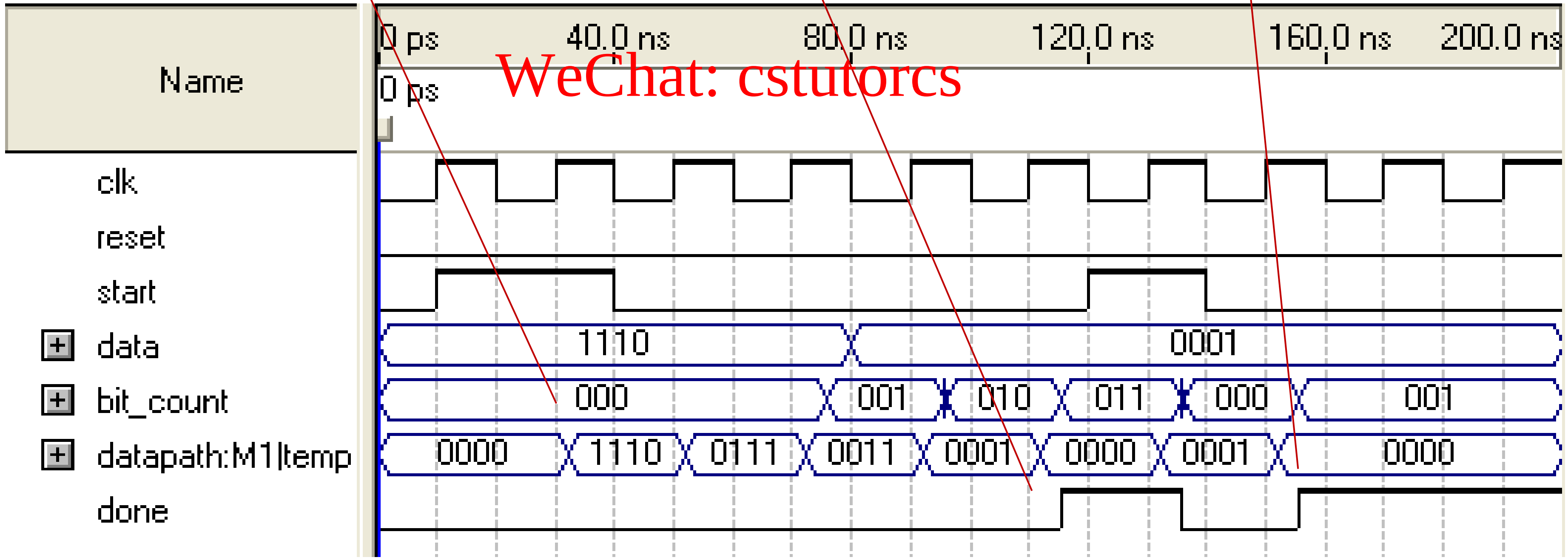


Annotated Simulation

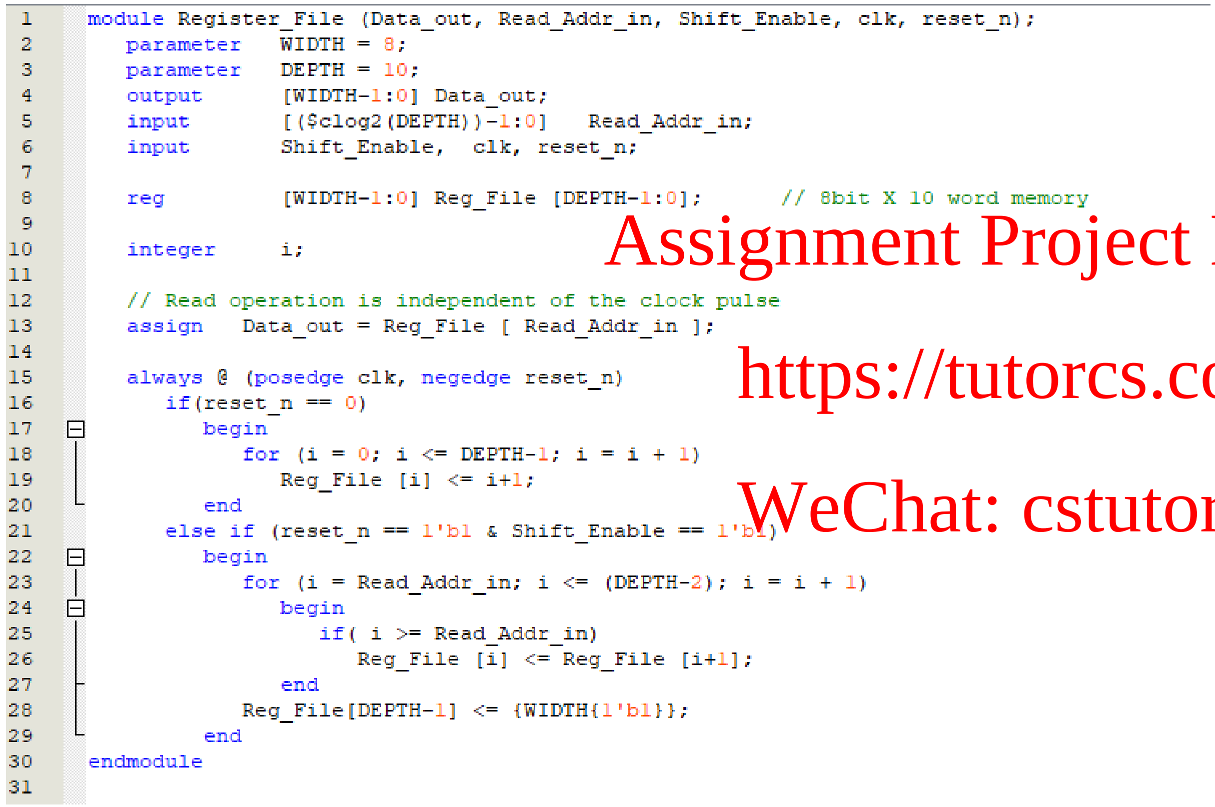
Data is loaded into the temp register in the following clock cycle

It takes 4 clock cycles for calculation of the number of ones because the MSB of the data is 1

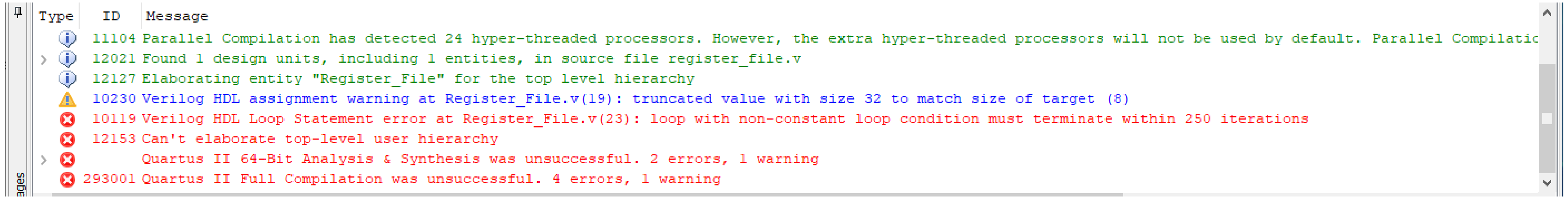
For the 0001 data it only takes one clock cycle because after the first shift the content of temp register becomes 0



Synthesising Loops



```
1   module Register_File (Data_out, Read_Addr_in, Shift_Enable, clk, reset_n);
2      parameter   WIDTH = 8;
3      parameter   DEPTH = 10;
4      output      [WIDTH-1:0] Data_out;
5      input       [($clog2(DEPTH))-1:0]   Read_Addr_in;
6      input       Shift_Enable,  clk, reset_n;
7
8      reg         [WIDTH-1:0] Reg_File [DEPTH-1:0];      // 8bit X 10 word memory
9
10     integer     i;
11
12     // Read operation is independent of the clock pulse
13     assign    Data_out = Reg_File [ Read_Addr_in ];
14
15     always @ (posedge clk, negedge reset_n)
16        if(reset_n == 0)
17           begin
18              for (i = 0; i <= DEPTH-1; i = i + 1)
19                 Reg_File [i] <= i+1;
20           end
21        else if (reset_n == 1'b1 & Shift_Enable == 1'b1)
22           begin
23              for (i = Read_Addr_in; i <= (DEPTH-2); i = i + 1)
24                 begin
25                    if( i >= Read_Addr_in)
26                       Reg_File [i] <= Reg_File [i+1];
27                 end
28              Reg_File[DEPTH-1] <= {WIDTH{1'b1}};
29           end
30  endmodule
31
```

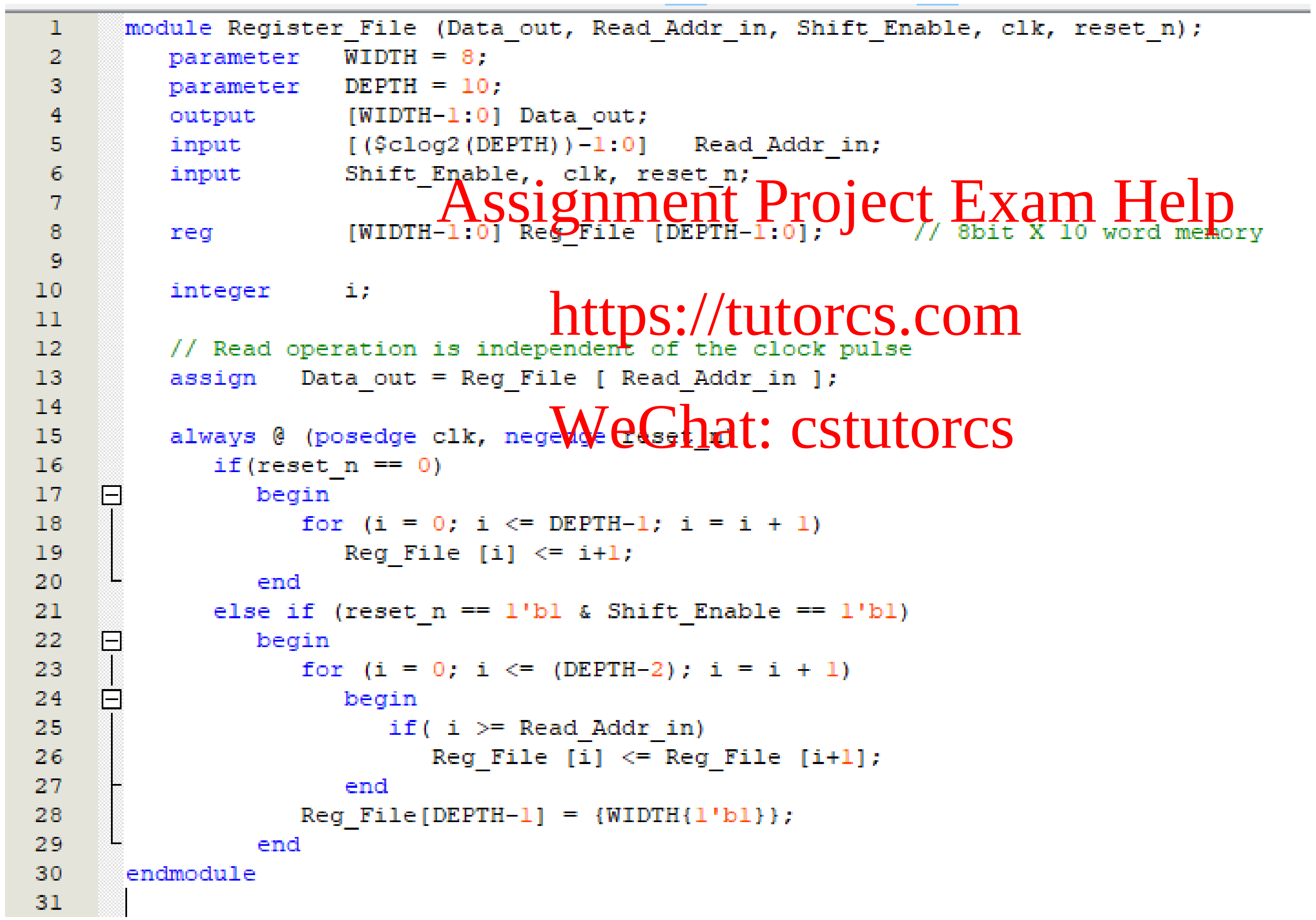


```
Type  ID     Message
      11104 Parallel Compilation has detected 24 hyper-threaded processors. However, the extra hyper-threaded processors will not be used by default. Parallel Compilatio
      12021 Found 1 design units, including 1 entities, in source file register_file.v
      12127 Elaborating entity "Register_File" for the top level hierarchy
      10230 Verilog HDL assignment warning at Register_File.v(19): truncated value with size 32 to match size of target (8)
      10119 Verilog HDL Loop Statement error at Register_File.v(23): loop with non-constant loop condition must terminate within 250 iterations
      12153 Can't elaborate top-level user hierarchy
            Quartus II 64-Bit Analysis & Synthesis was unsuccessful. 2 errors, 1 warning
     293001 Quartus II Full Compilation was unsuccessful. 4 errors, 1 warning
```

The error

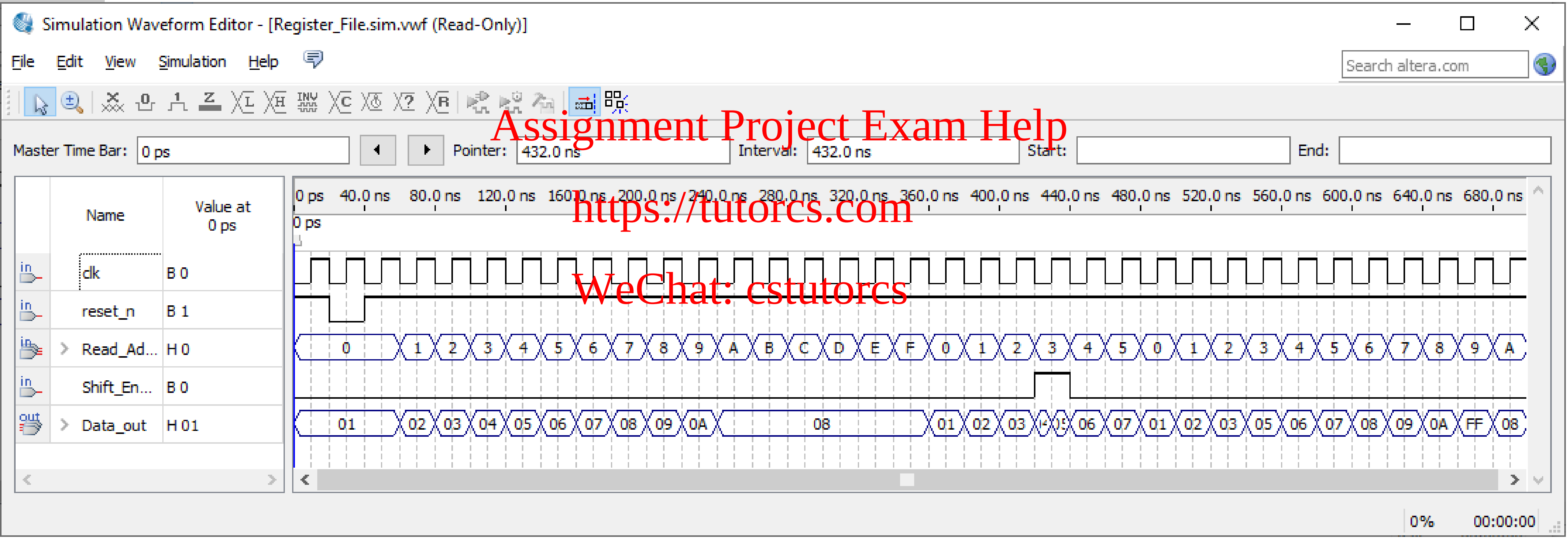
- The Error
  - Loop with none constant loop condition must terminate within 250 iterations
- The solution
  - Change from
    - For (i=variable; i<= variable; i = i+1)
      - Do something
  - To
    - For (i=fixed; i<= fixed; i = i+1)
      - If (i > variable) Do something

Revised Verilog code



```verilog
module Register_File (Data_out, Read_Addr_in, Shift_Enable, clk, reset_n);
   parameter   WIDTH = 8;
   parameter   DEPTH = 10;
   output      [WIDTH-1:0] Data_out;
   input       [($clog2(DEPTH))-1:0]   Read_Addr_in;
   input       Shift_Enable,  clk, reset_n;

   reg         [WIDTH-1:0] Reg_File [DEPTH-1:0];      // 8bit X 10 word memory

   integer     i;

   // Read operation is independent of the clock pulse
   assign   Data_out = Reg_File [ Read_Addr_in ];

   always @ (posedge clk, negedge reset_n)
      if(reset_n == 0)
         begin
            for (i = 0; i <= DEPTH-1; i = i + 1)
               Reg_File [i] <= i+1;
         end
      else if (reset_n == 1'b1 & Shift_Enable == 1'b1)
         begin
            for (i = 0; i <= (DEPTH-2); i = i + 1)
               begin
                  if( i >= Read_Addr_in)
                     Reg_File [i] <= Reg_File [i+1];
               end
            Reg_File[DEPTH-1] = {WIDTH{1'b1}};
         end
endmodule

```

Simulation



Simulation Waveform Editor - [Register_File.sim.vwf (Read-Only)]

File Edit View Simulation Help

Master Time Bar: 0 ps   Pointer: 432.0 ns   Interval: 432.0 ns   Start: End:

Name	Value at 0 ps
clk	B 0
reset_n	B 1
Read_Ad...	H 0
Shift_En...	B 0
Data_out	H 01

Search altera.com
0 ps 40.0 ns 80.0 ns 120.0 ns 160.0 ns 200.0 ns 240.0 ns 280.0 ns 320.0 ns 360.0 ns 400.0 ns 440.0 ns 480.0 ns 520.0 ns 560.0 ns 600.0 ns 640.0 ns 680.0 ns
0 ps
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 0 1 2 3 4 5 6 7 8 9 A
01 02 03 04 05 06 07 08 09 0A 08 01 02 03 06 07 01 02 03 05 06 07 08 09 0A FF 08
0%
00:00:00