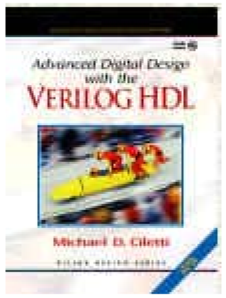
Advanced Digital Design with the Verilog HDL



M. D. Ciletti

Department
of
Electrical and Computer Engineering
University of Colorado
Colorado Springs, Colorado

ciletti@vlsic.uccs.edu

Draft: Chap 6c: Synthesis of Combinational and Sequential Logic (Rev 9/17/2003)

Synthesis of Implicit State Machines, Registers and Counters

- Implicit state machines are described by one or more edge-sensitive event-control expressions in the same behavior
- Clock edges define the boundaries of the states
- Machine remains in a fixed state between clock edges
- All transitions must be synchronized to the same edge of the clock
- Do not declare a state register – it is implicit
- Limitation: Each state of an implicit state machine may be entered from only one other state

Example: Implicit State Machines

- Synthesis tools infer the existence of an implicit FSM when a cyclic (***always***) behavior has more than one embedded, clock-synchronized, event control expression
- The multiple event control expressions within an implicit finite state machine separate the activity of the behavior into distinct clock cycles of the machine

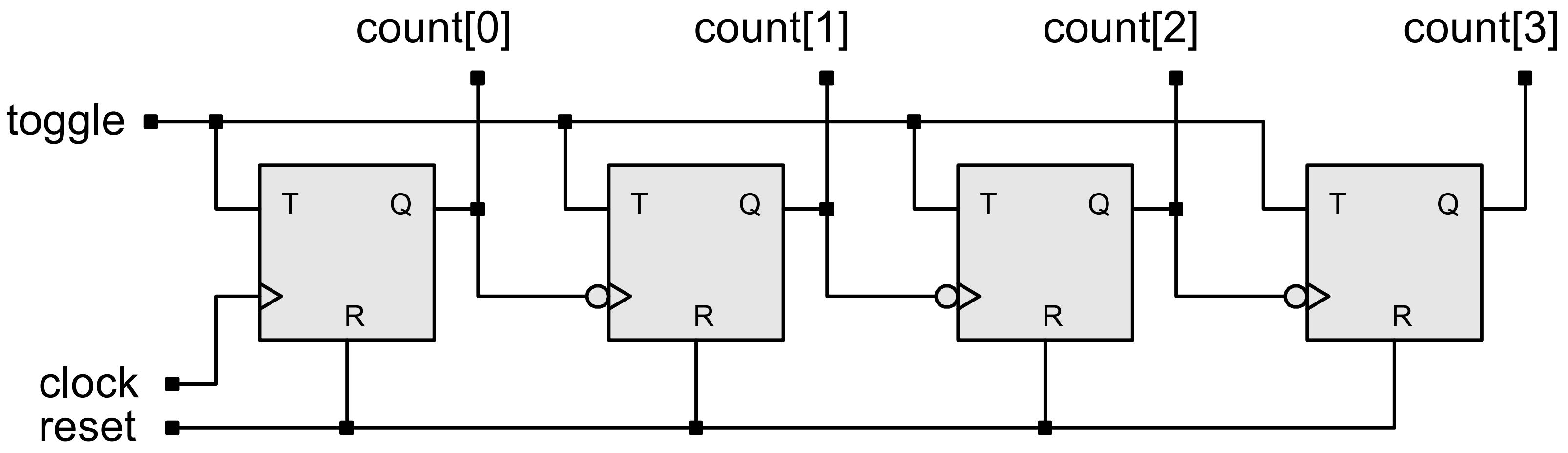
```
always @  (posedge clk)  // Synch event before first assignment
  begin
    reg_a <= reg_b;          // Executes in first clock cycle
    reg_c <= reg_d;          // Executes in first clock cycle.
    @  (posedge clk)         // Begins second clock cycle.
      begin
        reg_g <= reg_f;      // Executes in second clock cycle.
        reg_m <= reg_r;      // Executes in second clock cycle.
      end
  end
```

Synthesis of Implicit State Machines

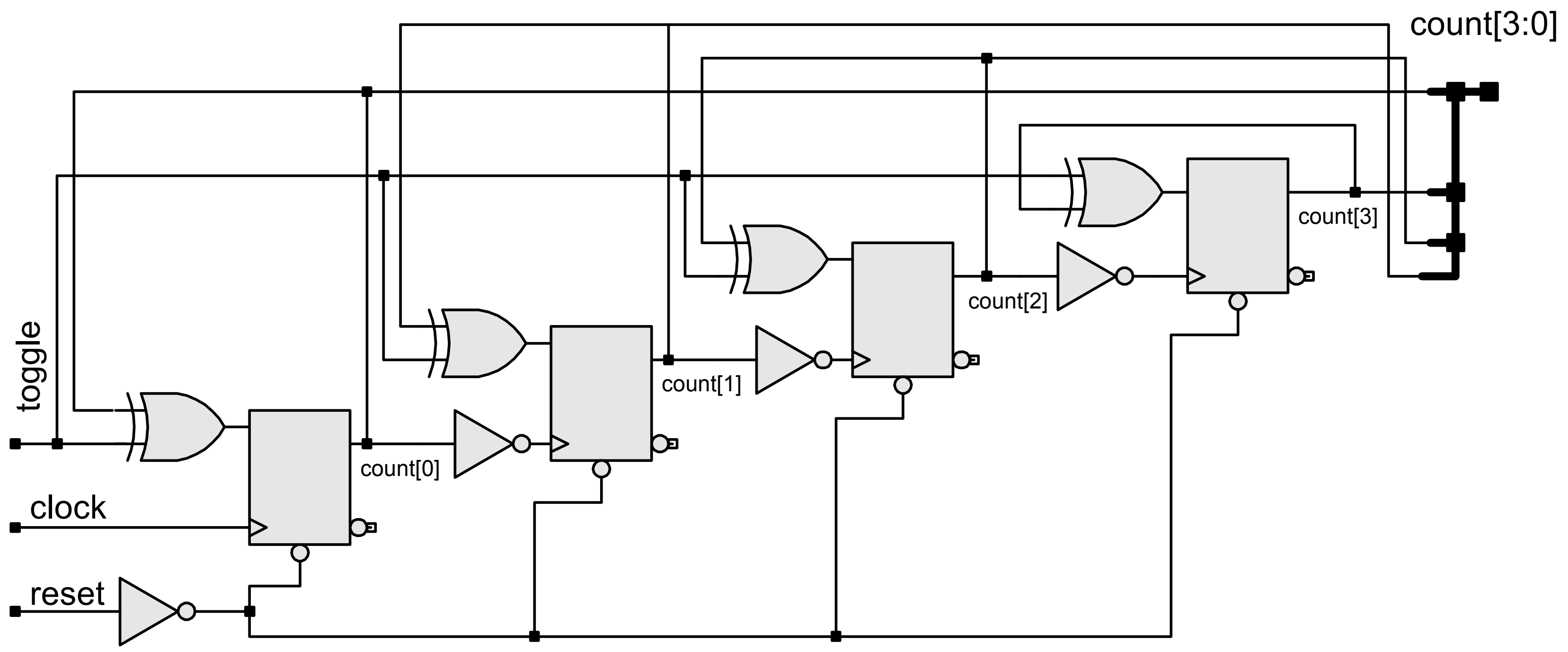
- Synthesis tool infers the size of the state
- Registers are allocated to hold "multiple wait states"
- Synthesis tool infers the logic governing state transitions

Example: Ripple Counter

```
module ripple_counter (count, toggle, clock, reset,);
 output      [3: 0]    count;
 input                 toggle, clock, reset;
 reg         [3: 0]    count;
 wire                  c0, c1, c2;

 assign c0 = count[0];
 assign c1 = count[1];
 assign c2 = count[2];

 always @  (posedge reset or posedge clock)
  if (reset == 1'b1) count[0] <= 1'b0; else
   if (toggle == 1'b1) count[0] <= ~count[0];

 always @  (posedge reset or negedge c0)
  if (reset == 1'b1) count[1] <= 1'b0; else
   if (toggle == 1'b1) count[1] <= ~count[1];
  always @  (posedge reset or negedge c1)
    if (reset == 1'b1) count[2] <= 1'b0; else
      if (toggle == 1'b1) count[2] <= ~count[2];

  always @  (posedge reset or negedge c2)
    if (reset == 1'b1) count[3] <= 1'b0; else
      if (toggle == 1'b1) count[3] <= ~count[3];
endmodule
```



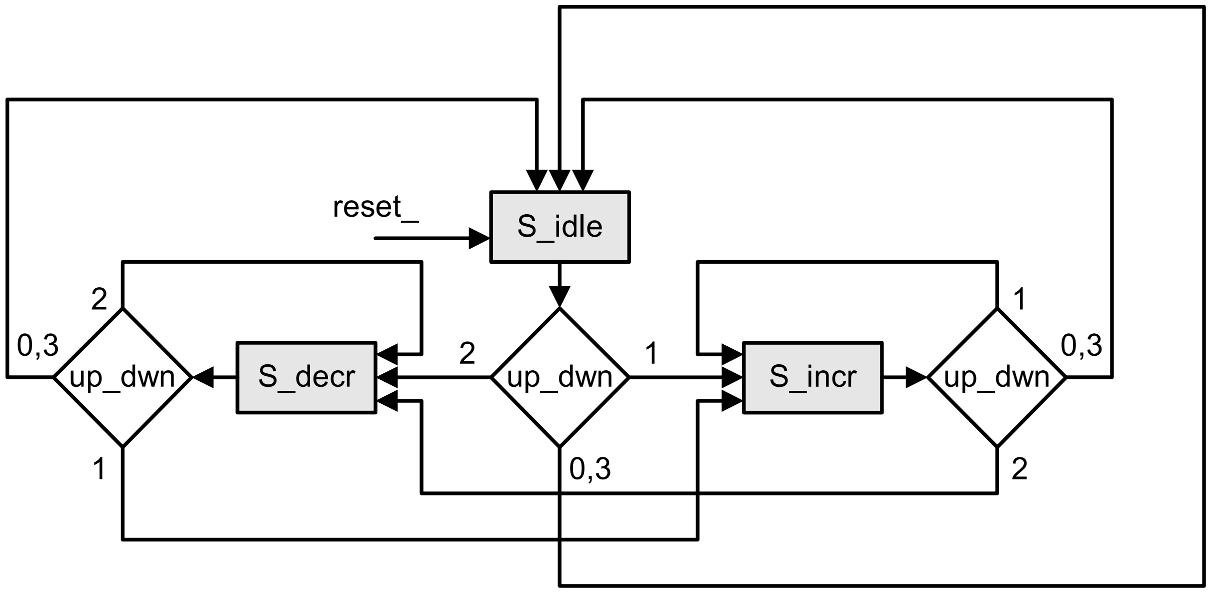
Synthesized circuit:

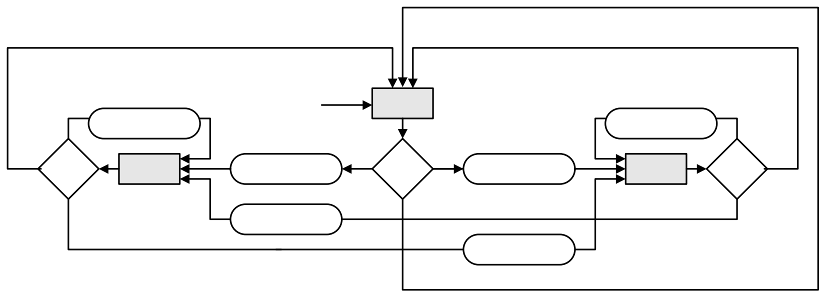


Example: Ring Counter

- A sequential machine having an identical activity flow in every cycle is a one-cycle implicit state machine
- One state: "running"

Revisit Example 5.40

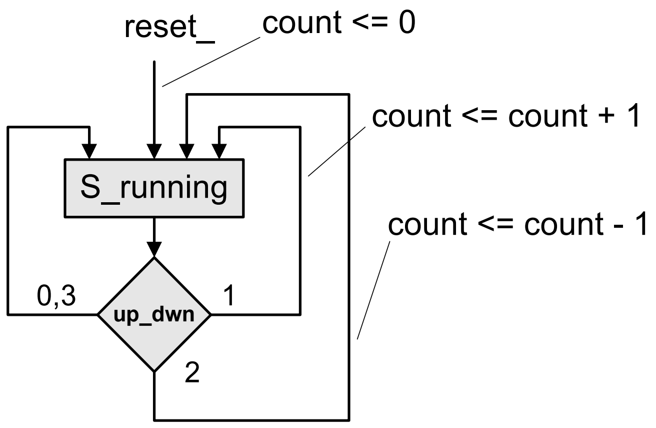


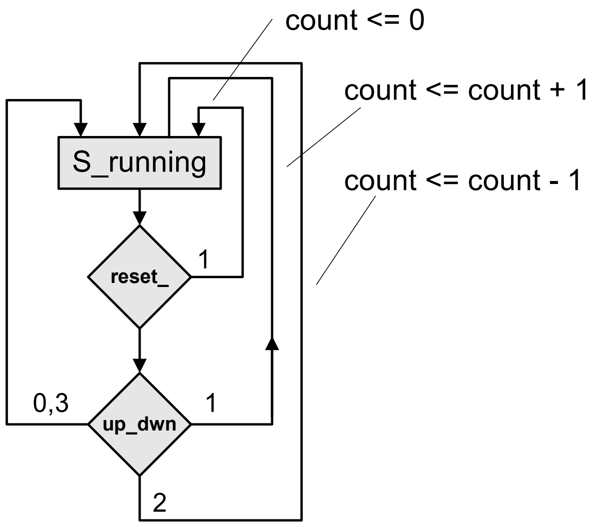


2

0,3
up_

1





Note: modeling the machine with a single implicit state is simpler than modeling a machine whose state is the count.

We'll see later that this problem can be partitioned into a datapath and a controller.

```
module Up_Down_Implicit1 (count, up_dwn, clock, reset_);
  output   [2: 0]      count;
  input    [1: 0]      up_dwn;
  input            clock, reset_;

  reg [2: 0] count;

  always @  (negedge clock or negedge reset_)
    if (reset_ == 0)                              count <= 3'b0; else
      if (up_dwn == 2'b00 || up_dwn == 2'b11)  count <= count; else
        if (up_dwn == 2'b01)                      count <= count + 1; else
          if (up_dwn == 2'b10)                    count <= count –1;

endmodule
```

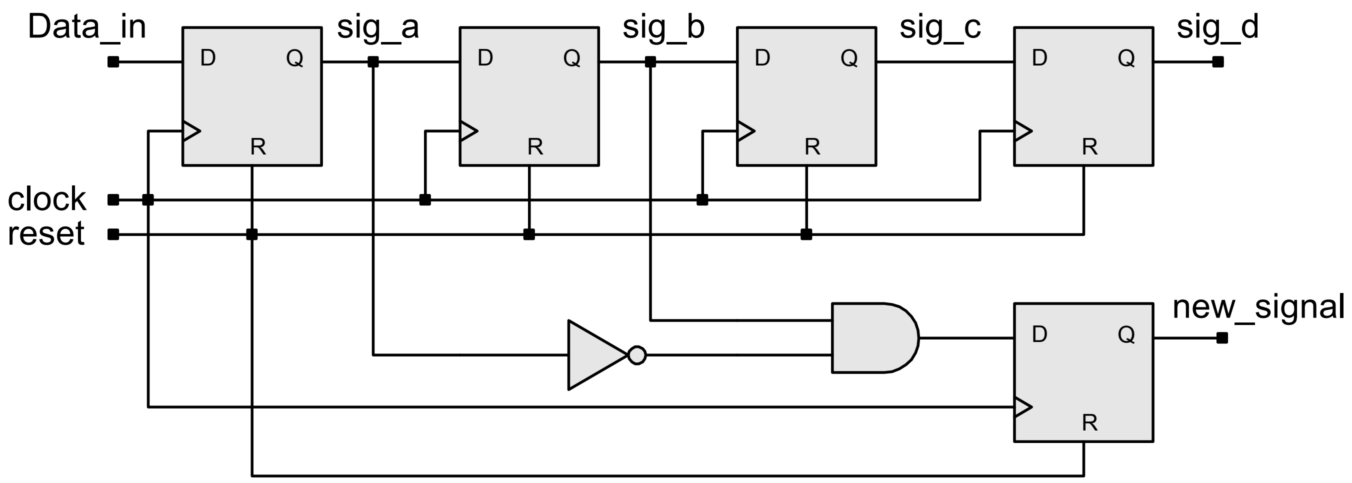
Synthesis of Registers

Register: Array of D-type flip-flops with a common clock

Example 6.31

```
module shifter_1 (sig_d, new_signal, Data_in, clock, reset);
 output      sig_d, new_signal;
 input       Data_in, clock, reset;

 reg         sig_a, sig_b, sig_c, sig_d, new_signal;
 always @  (posedge reset or posedge clock)
  begin if (reset == 1'b1)
   begin
    sig_a <= 0;
    sig_b <= 0;
    sig_c <= 0;
    sig_d <= 0;
    new_signal <= 0;
   end
  else
   begin
    sig_a <= Data_in;
    sig_b <= sig_a;
    sig_c <= sig_b;
    sig_d <= sig_c;
    new_signal <= (~ sig_a) & sig_b;
   end
 end
endmodule
```

Note: *new_signal* receives value within a synchronous behavior and appears as an output port. It will be synthesized as the output of a flip-flop.

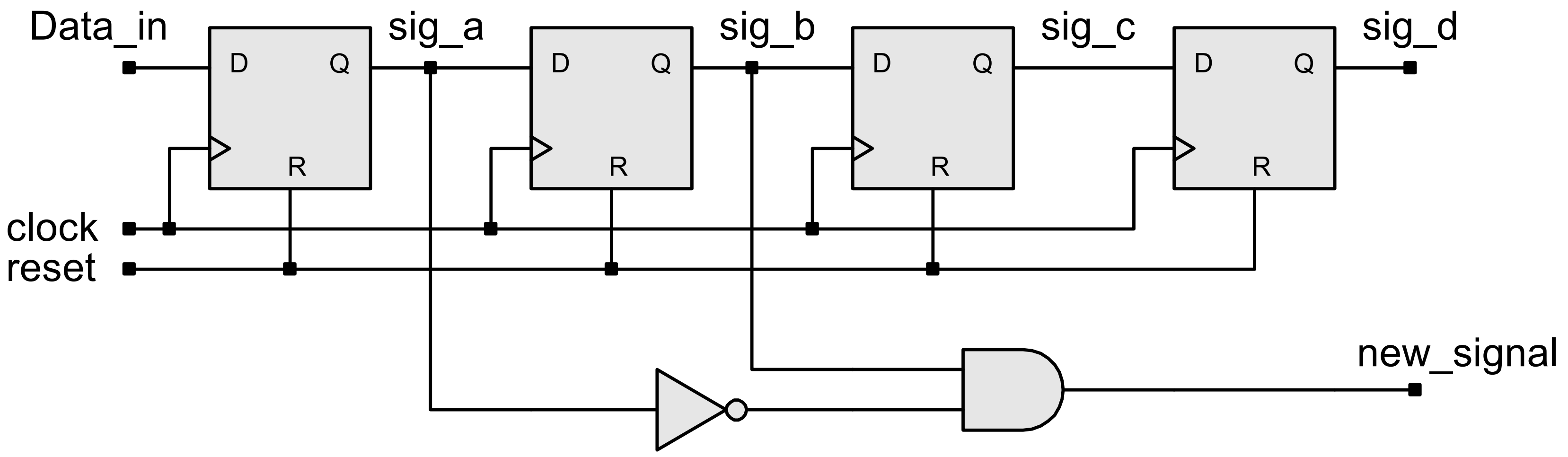


Example 6.32

- *new_signal* is formed outside of the behavior in a continuous assignment
- *new_signal* appears as an output port
- *new_signal* is synthesized as the output of combinational logic.

```
module shifter_2 (sig_d, new_signal, Data_in, clock, reset);
  output       sig_d, new_signal;
  input        Data_in, clock, reset;

  reg          sig_a, sig_b, sig_c, sig_d, new_signal;
```

```
 always @  (posedge reset or posedge clock)
  begin
   if (reset == 1'b1)
    begin
     sig_a <= 0;
     sig_b <= 0;
     sig_c <= 0;
     sig_d <= 0;
    end
   else
    begin
     sig_a <= shift_input;
     sig_b <= sig_a;
     sig_c <= sig_b;
     sig_d <= sig_c;
    end
  end
  assign new_signal = (~ sig_a) & sig_b;
endmodule
```


Data_in
sig_a
sig_b
sig_c
sig_d
D
Q
R
clock
reset
new_signal

Example 6.33 Two versions of an accumulator forming the running sum of two samples of an input.

```
module Add_Accum_1 (accum, overflow, data, enable, clk, reset_b);
  output [3: 0]      accum;
  output        overflow;
  input [3: 0]  data;
  input         enable, clk, reset_b;
  reg           accum, overflow;

  always @ (posedge clk or negedge reset_b)
    if (reset_b == 0) begin accum <= 0; overflow <= 0; end
    else if (enable) {overflow, accum} <= accum + data;
endmodule
```

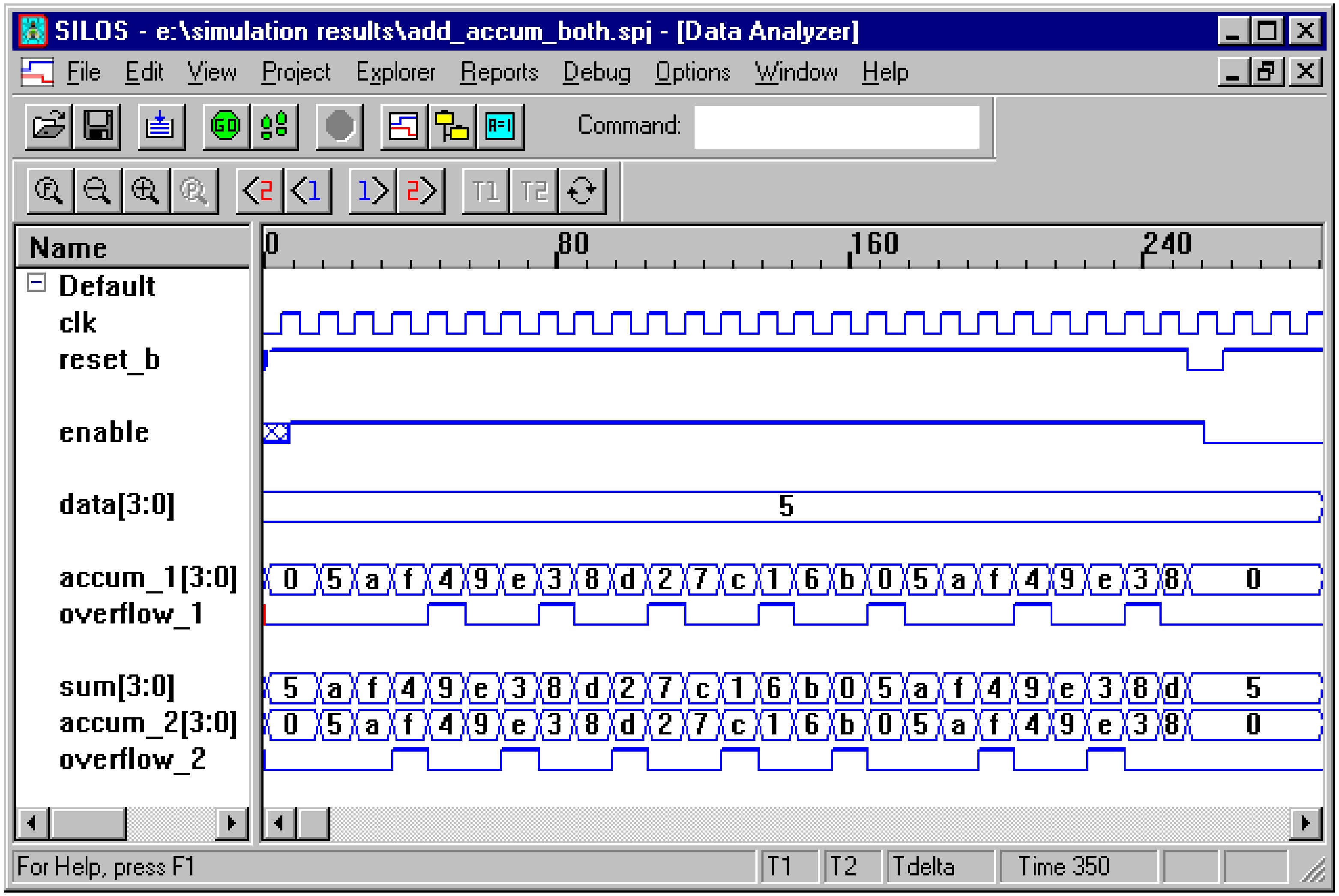
```
module Add_Accum_2 (accum, overflow, data, enable, clk, reset_b);
 output [3: 0]      accum;
 output        overflow;
 input [3: 0]  data;
 input         enable, clk, reset_b;
 reg           accum;
 wire [3:0]    sum;
 assign        {overflow, sum} = accum + data;

 always @ (posedge clk or negedge reset_b)
   if (reset_b == 0) accum <= 0;
   else if (enable) accum <=  sum;
endmodule
```

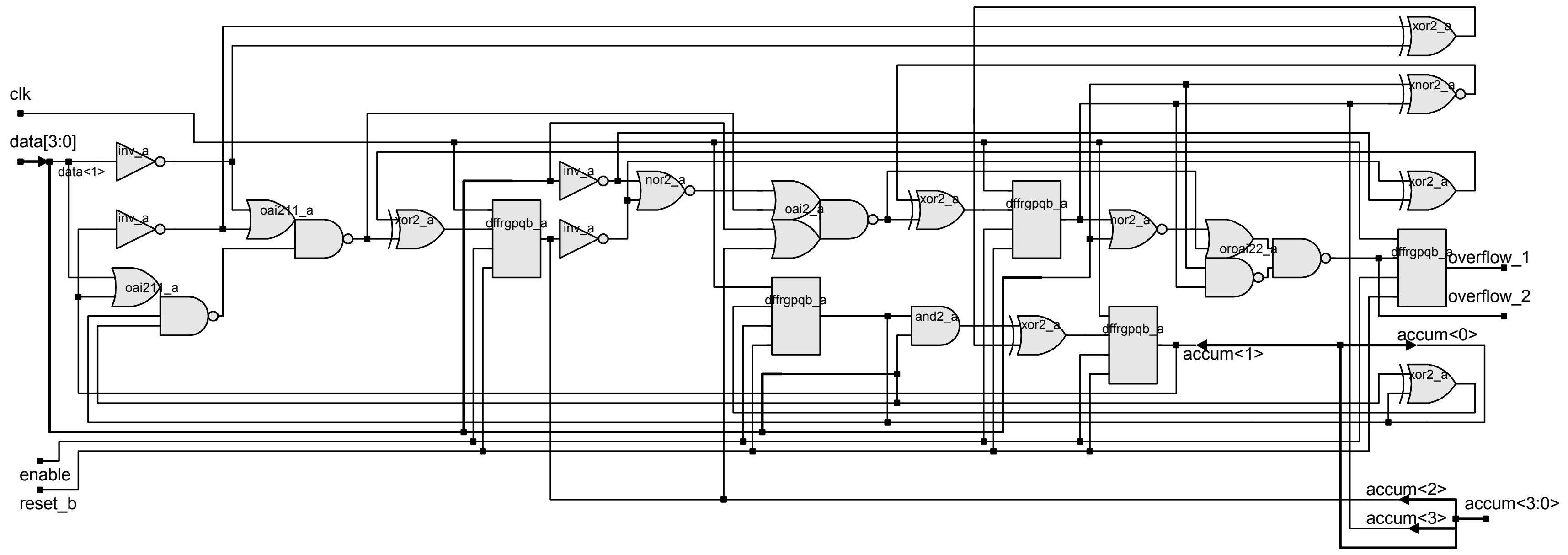
Simulation Results:

- *Add_Accum_1* forms *overflow_1* one cycle after storing the results of an overflow condition
- *Add_Accum_2* forms *overflow_2* as an unregistered Mealy output

Synthesis Results:

- *overflow_1* is formed in *Add_Accum_1 as a registered version of overflow_2*, which is formed in *Add_Accum_2*.


SILOS - e:\simulation results\add_accum_both.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
clk
reset_b
enable
data[3:0]
accum_1[3:0]
overflow_1
sum[3:0]
accum_2[3:0]
overflow_2
0
80
160
240
5
0 5 a f 4 9 e 3 8 d 2 7 c 1 6 b 0 5 a f 4 9 e 3 8 0
5 a f 4 9 e 3 8 d 2 7 c 1 6 b 0 5 a f 4 9 e 3 8 d 5
0 5 a f 4 9 e 3 8 d 2 7 c 1 6 b 0 5 a f 4 9 e 3 8 0
For Help, press F1
T1
T2
Tdelta
Time 350


clk
data[3:0]
data<1>
enable
reset_b
inv_a
oai211_a
xor2_a
dffrgpqb_a
nor2_a
oai2_a
and2_a
oroai22_a
xnor2_a
overflow_1
overflow_2
accum<0>
accum<1>
accum<2>
accum<3>
accum<3:0>

Resets

- Provide a reset signal to every sequential module that is to be synthesized
  - Initialize the state
  - Support testing
- Provide an external signal to reset an implicit state machine
- Provide reset logic to every cycle of the behavioral model
  - Return to the top of the multi-cycle behavior from any cycle
  - Return to the same state independently of when reset is asserted
  - Synthesis will produce extra logic to accommodate incomplete resets
- May have to synchronize the reset signal

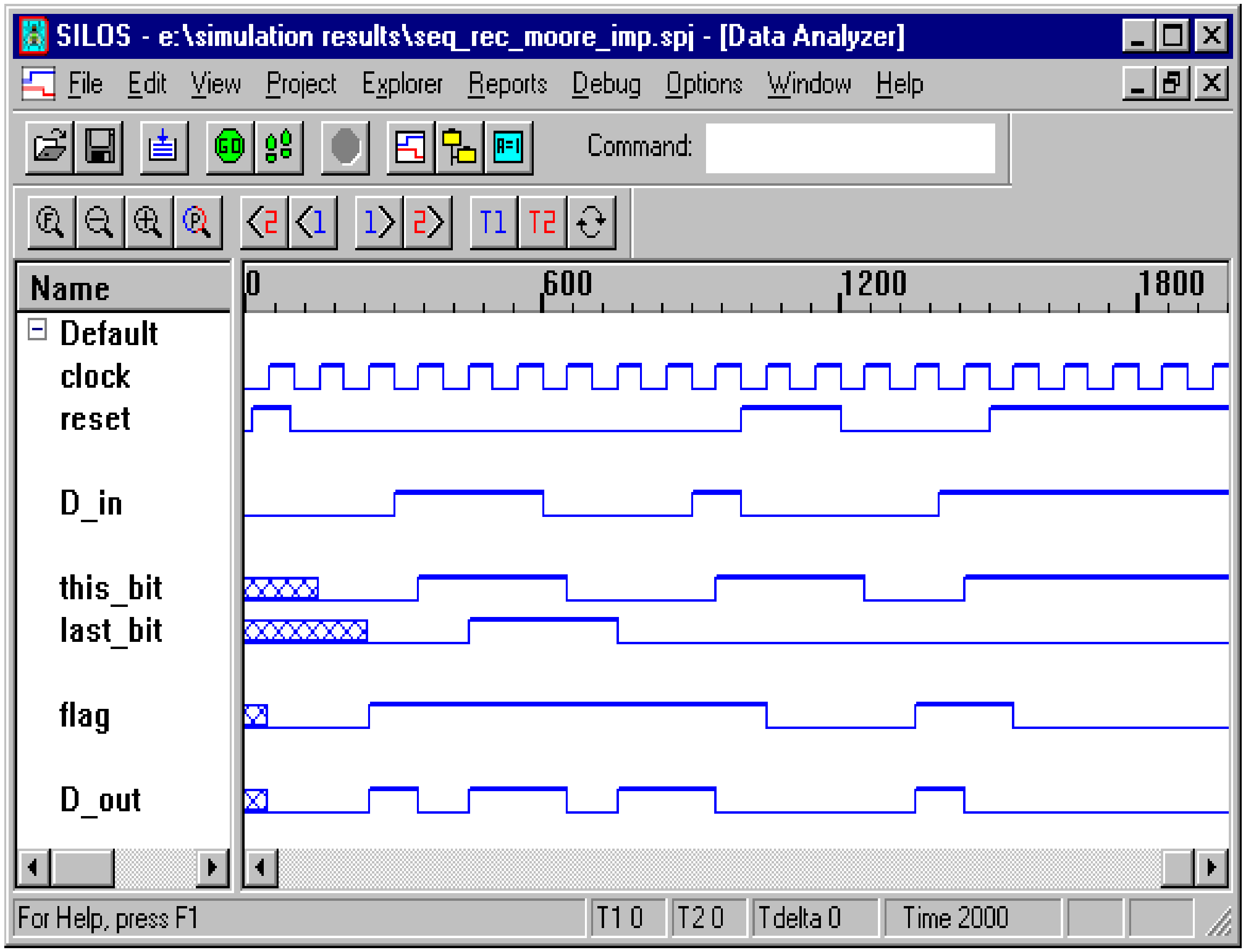
Partial Resets

- Implicit state machine to asserts *D_out* after two successive samples of *D_in* are both either 1 or 0
- Hold samples in a two-stage shift register
- Prevent premature assertion by having a state machine set *flag* after two samples have been received
- Examine consequences of partially resetting the data register

```
module Seq_Rec_Moore_imp (D_out, D_in, clock, reset);
 output          D_out;
 input           D_in;
 input      clock, reset;
 reg        last_bit, this_bit, flag;
 wire       D_out;
 always begin: wrapper_for_synthesis
  @ (posedge clock /* or posedge reset*/)
  begin: machine
     if (reset == 1) begin
        last_bit <= 0;
        //  this_bit <= 0;
        // flag <= 0;
        disable machine; end
    else begin
     //  last_bit <= this_bit;
     this_bit <= D_in;
```

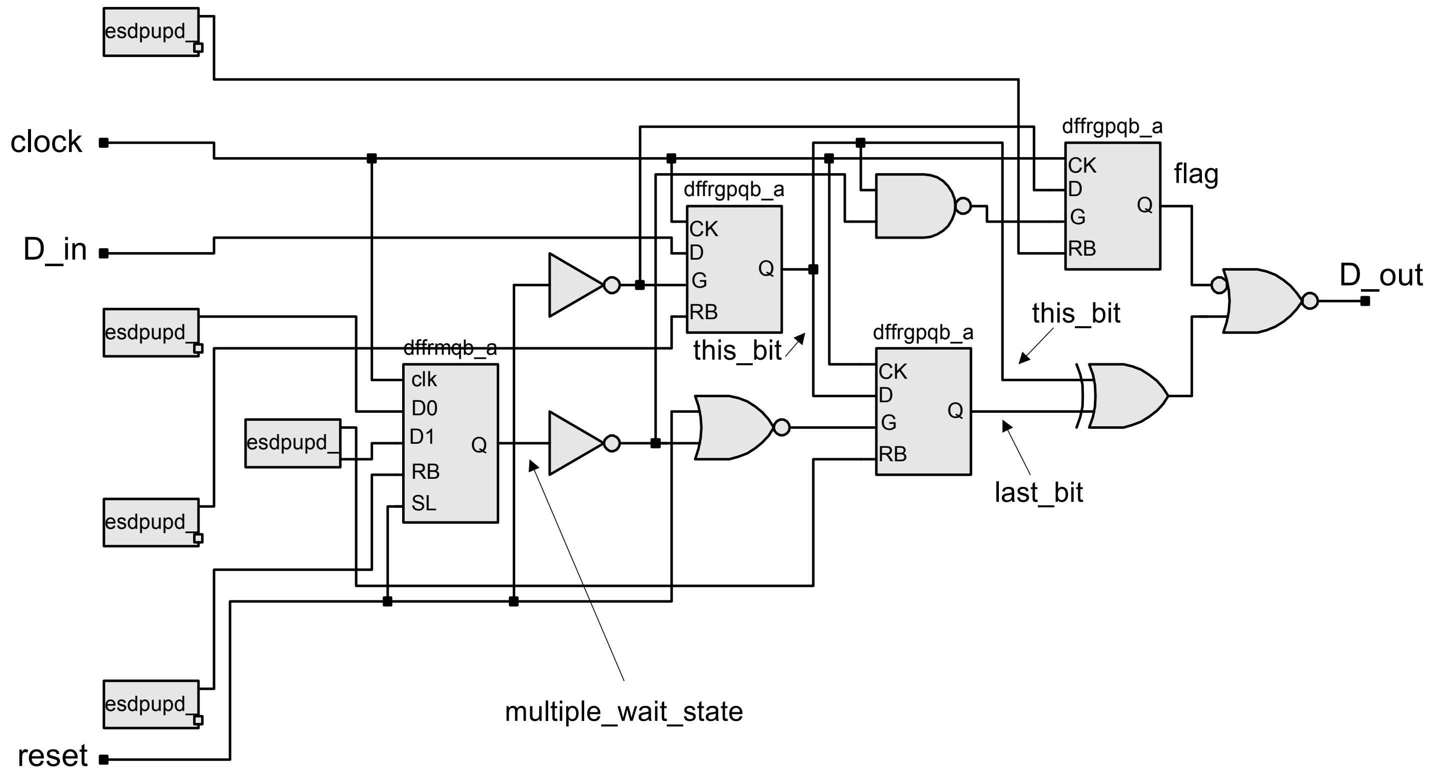
```
        forever
          @ (posedge clock  /* or posedge reset */) begin
            if (reset == 1) begin
              // last_bit <= 0;
              // this_bit <= 0;
              flag <= 0;
              disable machine; end
            else begin
              last_bit <= this_bit;
              this_bit <= D_in;
              flag <= 1; end          // second edge
        end
      end
    end // machine
  end  // wrapper_for_synthesis

  assign D_out = (flag && (this_bit == last_bit));
endmodule
```

Simulation Results:


SILOS - e:\simulation results\seq_rec_moore_imp.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
clock
reset
D_in
this_bit
last_bit
flag
D_out
0
600
1200
1800
For Help, press F1
T1 0
T2 0
Tdelta 0
Time 2000

Note: synthesis depends on how the reset is implemented

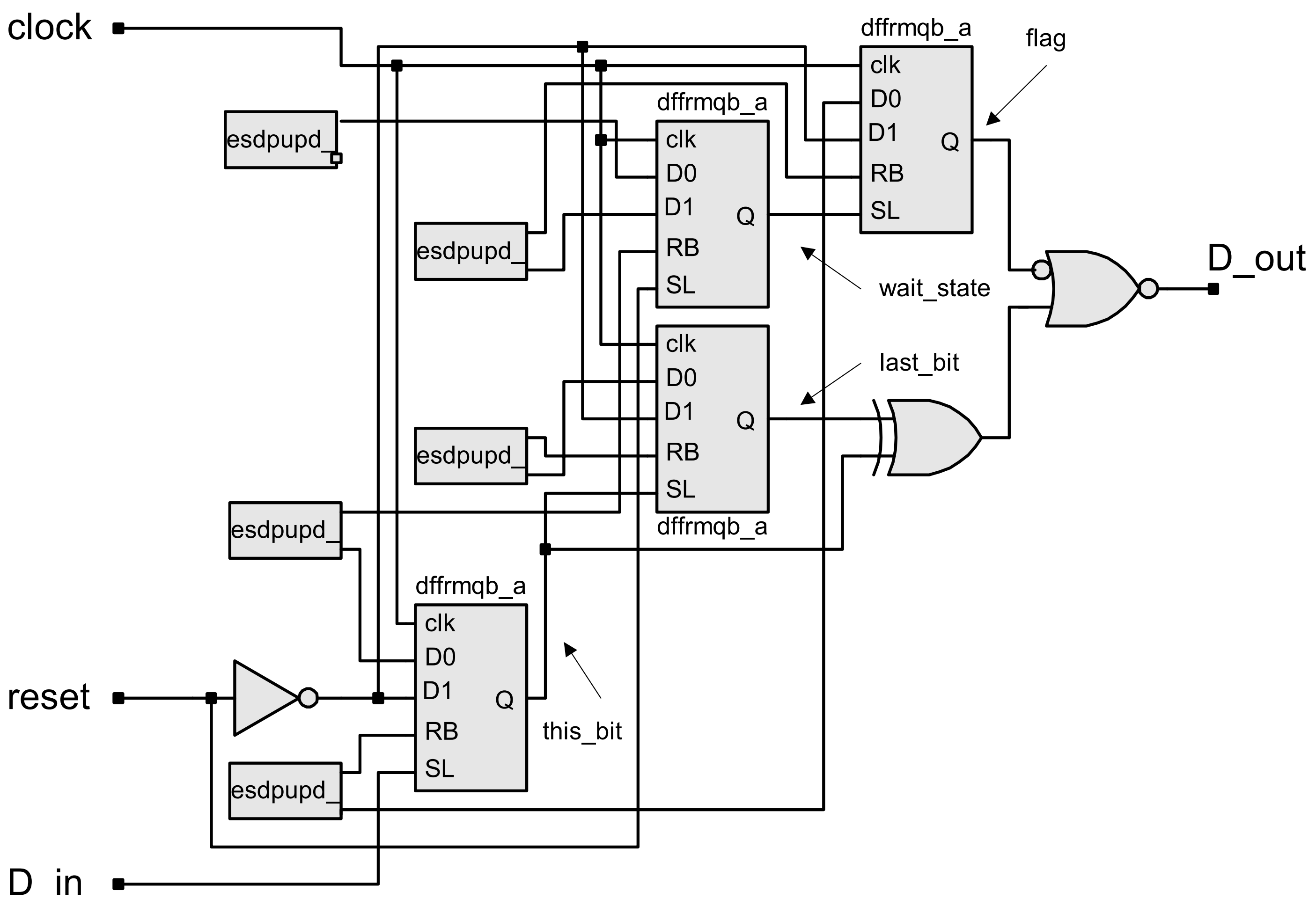
Case 1: Flush only last_bit (earliest bit received)



Features to note:

- dffrgpqb_a is a gate-input flip-flop with an internal datapath from *Q* to *D*
  - *Q* is connected to *D* when *G* is low
  - *Q* is connected to external datapath (*D*) when *Q* is high
- dffmpqb_a is a dual multiplexed input flip-flop
  - Holds multiple_wait_state indicating two samples received
  - *RB* (reset) input is disabled by espupd
  - *SL* (set) is active low and wire to reset
  - *D0* and *D1* are wired to power and ground respectively
  - *SL* low selects *D0*; *SL* high selects *D1*
- Assertion of reset:
  - Causes *this_bit* and *last_bit* to hold their values
  - Steers the external input (0) to the *flag* register
- De-assertion of reset:
  - *multiple_wait_state* is 1 after first clock
  - Steers *this_bit* to *last_bit*
  - *this_bit* gets *in_bit* after reset is de-asserted
- Only *flag* is flushed in the loop

Case 2: Flush the pipeline



When registers are not flushed on reset, additional logic is required to feed their outputs back to their inputs to retain their state under the action of the clock.

Option: for simpler hardware, drive the register to a known value on reset.

Synthesis of Gated Clocks and Clock Enables

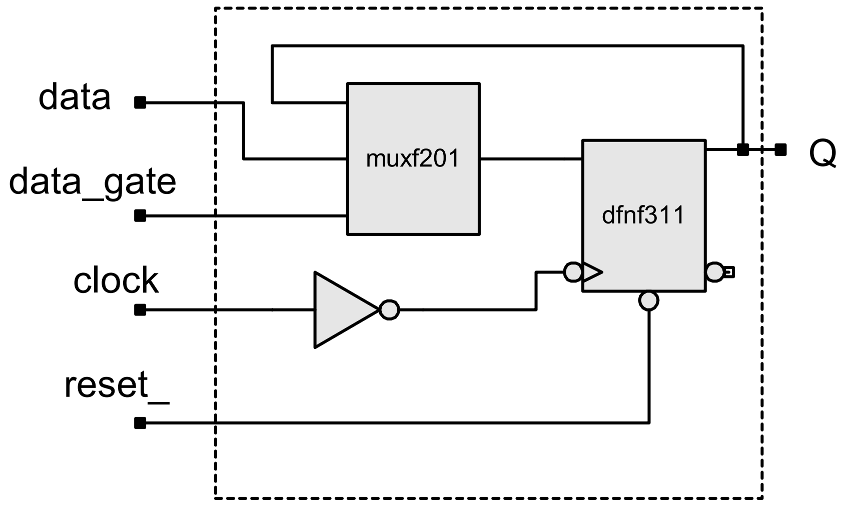
Caution: Gated clocks can add skew to the clock path, and cause the clock signal to violate a flip-flop's constraint on the minimum width of the clock pulse.

Recommended style:

```
module best_gated_clock (clock, reset_, data_gate, data, Q);
 input          clock, reset_, data, data_gate;
 output         Q;
 reg            Q;

 always @  (posedge clock or negedge reset_)
   if (reset_ == 0) Q <= 0; else if (data_gate) Q <= data;
endmodule
```

Features:

- Input data and flip-flop output are multiplexed at D input
- Clock synchronizes the circuit
- *data_gate* gates the action of the clock
- Clock slivers are not a problem



Typical model for clock enable logic:

```
always @ (posedge clk)
if (enable == 1) q_out <= Data_in;
```

Anticipating the Results of Synthesis

- Data types
- Operator grouping
- Expression Substitution
- Loops

Additional details in "Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL"

Synthesis of Data Types

- The identity of nets that are primary inputs or outputs are retained in synthesis
- Internal nets may be eliminated by synthesis
- Integers are stored as 32-bit words (minimum per IEEE 1364)
- Used sized parameters rather than integer parameters
- Explicit use of x or z in logical tests will not synthesize

Synthesis of Operators

Some operators may map directly into library cells (e.g. +, 1, <, >, =)
Synthesis might impose restrictions on an operator

- Shift operators is allowed only if the shift index is a constant
- Reduction, bitwise, and logical operators synthesize to equivalent gates
- Conditional operator synthesizes to a mux structure
- If both operands of * are constant the tool produces the constant result
- If one operand of * is a power of 2 the synthesis tool forms the result by left-shifting the other operand
- If one operand of / is a power of 2 the result will be formed by right-shifting the other operand
- If the divisor is a power of 2 the % operator will be formed as a part-select

See "Modeling, Synthesis and Rapid Prototyping with the Verilog HDL for more details

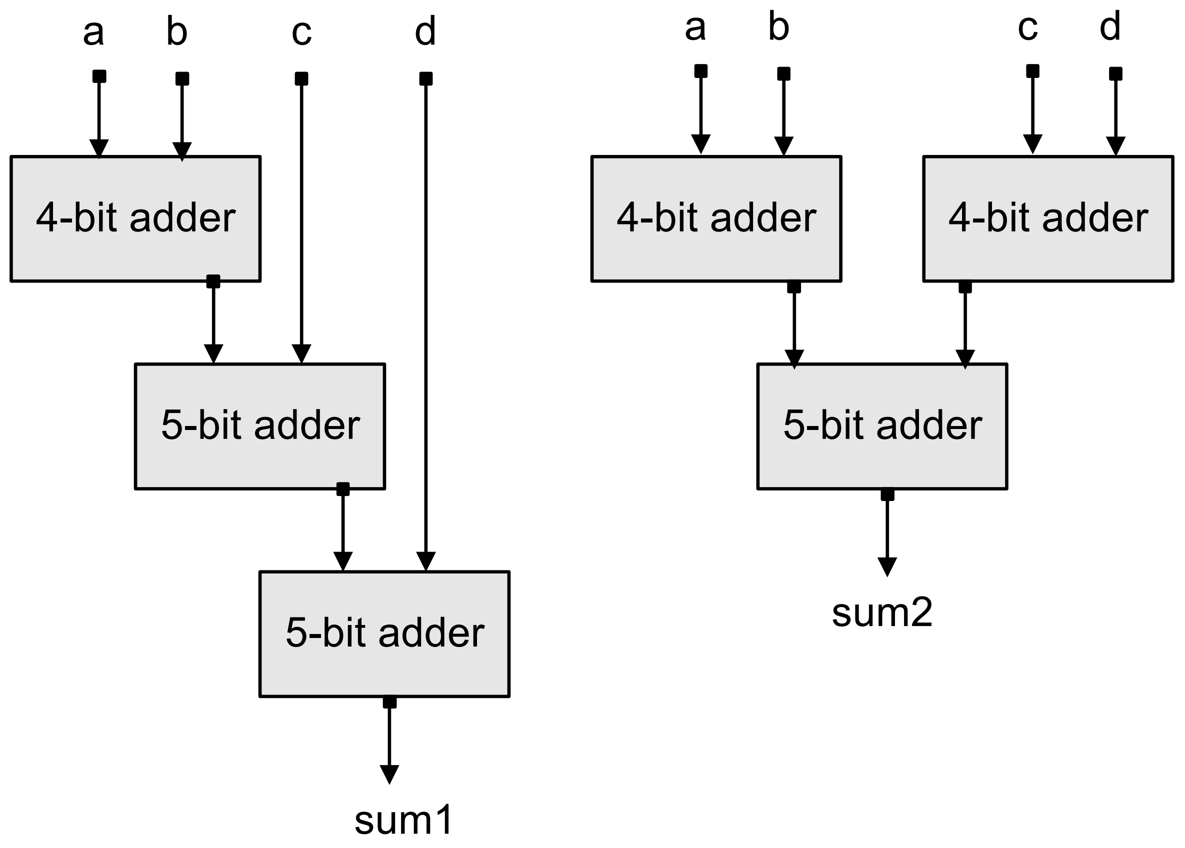
Operator Grouping

Use parentheses within expressions to influence the outcome of synthesis

Example 6.35

```
module operator_group (sum1, sum2, a, b, c, d);
  output [4: 0]      sum1, sum2;
  input   [3: 0]      a, b, c, d;


  assign sum1 = a + b + c + d;         // 3 levels of logic
  assign sum2 = (a + b) + (c + d);    // 2 levels of logic

endmodule
```

Synthesis results:



Note: Use sum1 with input d to accommodate a late signal

Expression Substitution

- Synthesis tools perform expression substitution to determine the effective logic of a sequence of procedural assignments

- Remember: procedural assignments have immediate effect on the value of the target variable

Example 6.36

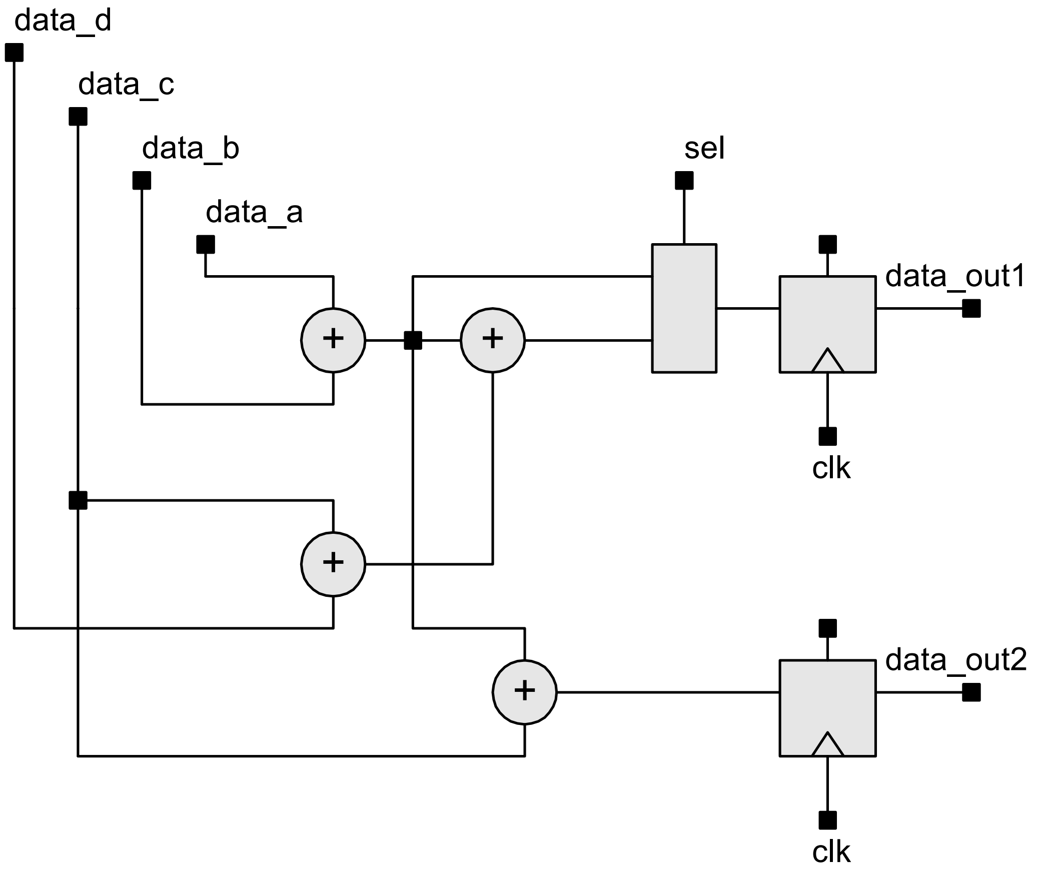
```
module  multiple_reg_assign
 (data_out1, data_out2, data_a, data_b, data_c, data_d, sel, clk);

 output       [4: 0]     data_out1, data_out2;
 input        [3: 0]     data_a, data_b, data_c, data_d;
 input               clk;

 reg          [4: 0]     data_out1, data_out2;

 always @ (posedge clk)
  begin
   data_out1 = data_a + data_b ;
   data_out2 = data_out1 + data_c;
   if (sel == 1'b0)
    data_out1 = data_out2 + data_d;
  end
endmodule
```

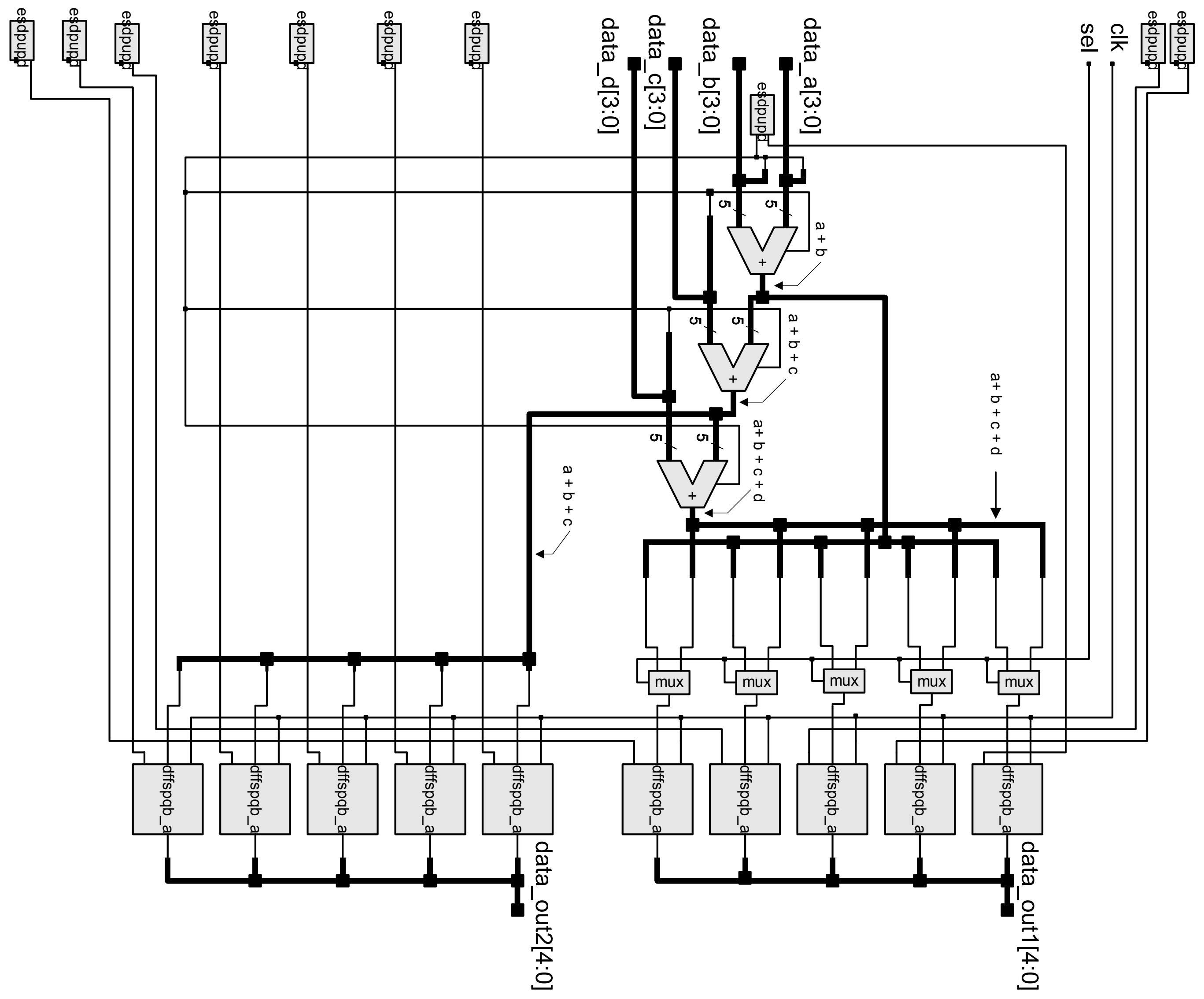
More readable alternative:

```
module  expression_sub
  (data_out1, data_out2, data_a, data_b, data_c, data_d, sel, clk);

  output       [4: 0]      data_out1, data_out2;
  input        [3: 0]      data_a, data_b, data_c, data_d;
  input                 sel, clk;
  reg          [4: 0]      data_out1, data_out2;

  always @ (posedge clk)
    begin
      data_out2 = data_a + data_b + data_c;
      if (sel == 1'b0)
        data_out1 = data_a + data_b + data_c + data_d;
      else
        data_out1 = data_a + data_b;
    end
endmodule
```

Equivalent logic with nonblocking assignments

```
module  expression_sub_nb
 (data_out1nb, data_out2nb, data_a, data_b, data_c, data_d, sel, clk);

 output      [4: 0]    data_out1nb, data_out2nb;
 input       [3: 0]    data_a, data_b, data_c, data_d;
 input                 sel, clk;
 reg         [4: 0]    data_out1nb, data_out2nb;

 always @ (posedge clk)
  begin
   data_out2nb <= data_a + data_b + data_c;
   if (sel == 1'b0)
    data_out1nb <= data_a + data_b + data_c + data_d;
   else
    data_out1nb <= data_a + data_b;
  end
endmodule
```


data_d
data_c
data_b
data_a
sel
data_out1
clk
data_out2
clk
+
+
+
+


esdpupd
data_d[3:0]
data_c[3:0]
data_b[3:0]
data_a[3:0]
sel
clk
5
a + b
a + b + c
a + b + c + d
mux
dffspqb_a
data_out2[4:0]
data_out1[4:0]

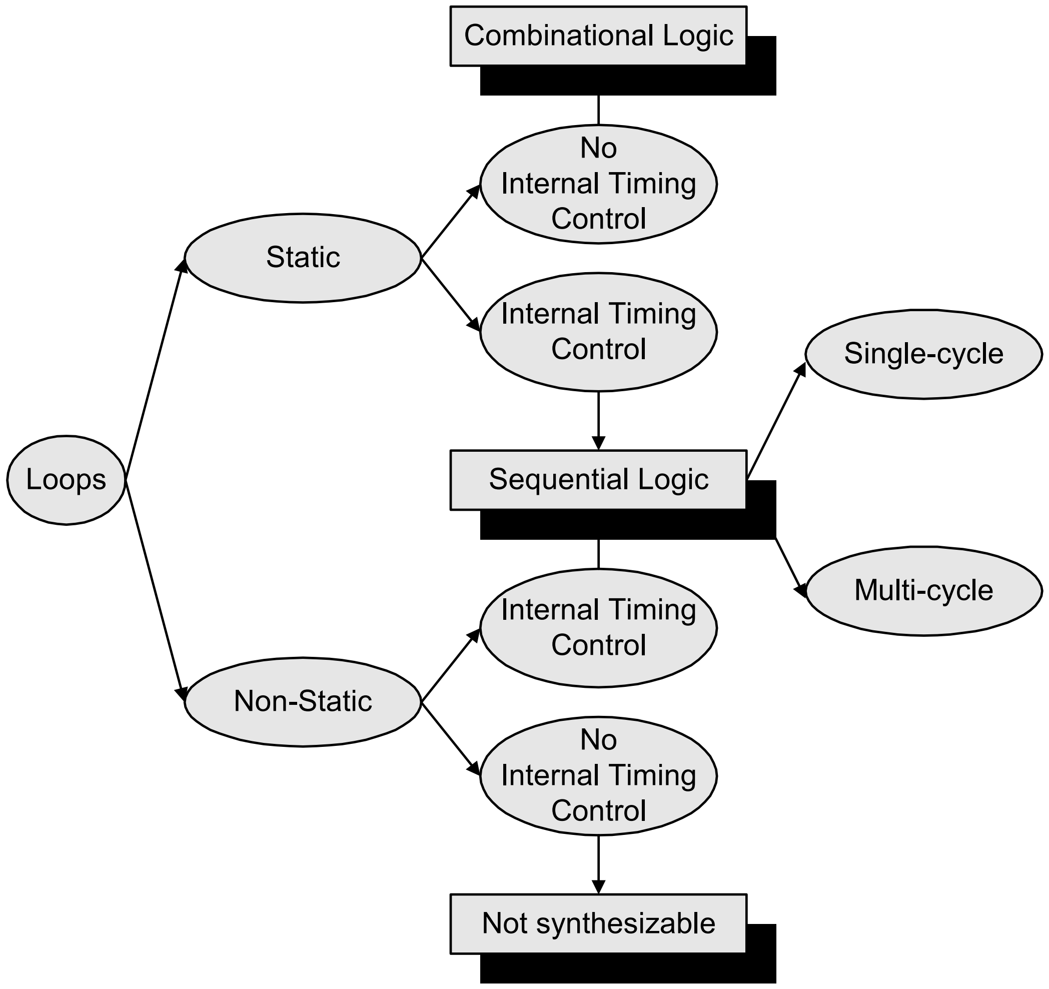
Synthesis of Loops

Classifications of loops according to data dependency and timing controls

Static (data-independent) loop: the number of its iterations can be determined by the compiler *before* simulation

Non-static (data-dependent) loop: the number of iterations depends on some variable during operation.

Note: Non-static loops that do not have internal timing controls cannot be synthesized directly.


Combinational Logic
No Internal Timing Control
Static
Internal Timing Control
Single-cycle
Loops
Sequential Logic
Multi-cycle
Internal Timing Control
Non-Static
No Internal Timing Control
Not synthesizable

Static Loops – No Internal Timing Control

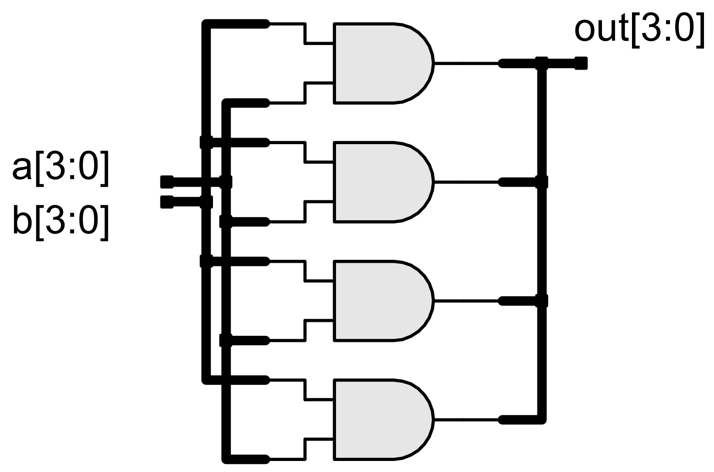
If a loop has no internal timing controls and no data dependencies, its computational activity is implicitly combinational.

Example 6.37

```
module for_and_loop_comb (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
```

Equivalent unrolled loop:

```
out[0] = a[0] & b[0];
out[1] = a[1] & b[1];
out[2] = a[2] & b[2];
out[3] = a[3] & b[3];
```



There are no dependencies in the datapath, and the order in which the statements are evaluated does not affect the outcome of the evaluation.

Example 6.38 Count the 1s in a word received in parallel format

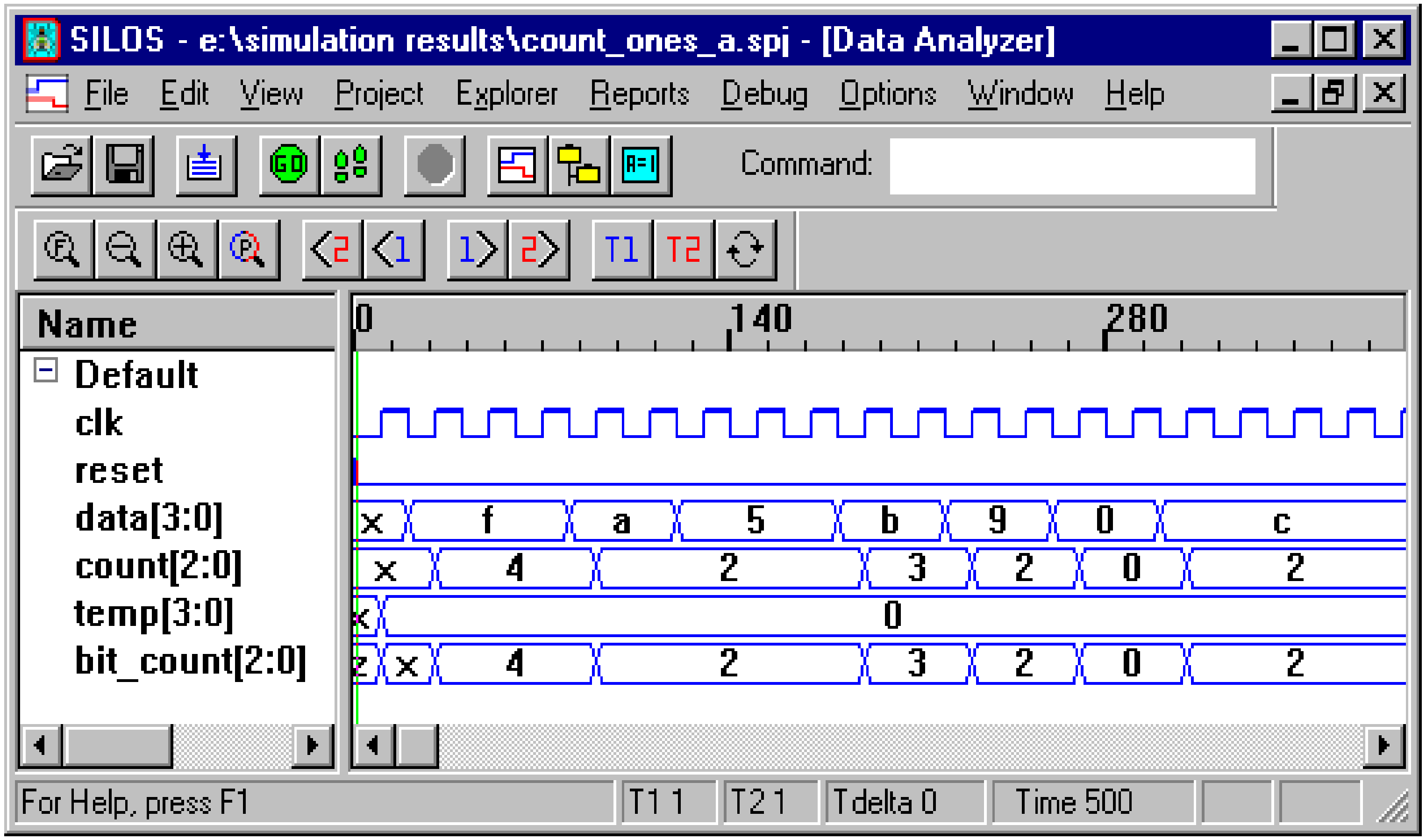
```
module count_ones_a (bit_count, data, clk, reset);
 parameter                         data_width = 4;
 parameter                         count_width = 3;
 output       [count_width-1: 0]       bit_count;
 input        [data_width-1: 0]        data;
 input                             clk, reset;
 reg          [count_width-1: 0]       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;
    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
```

Note:

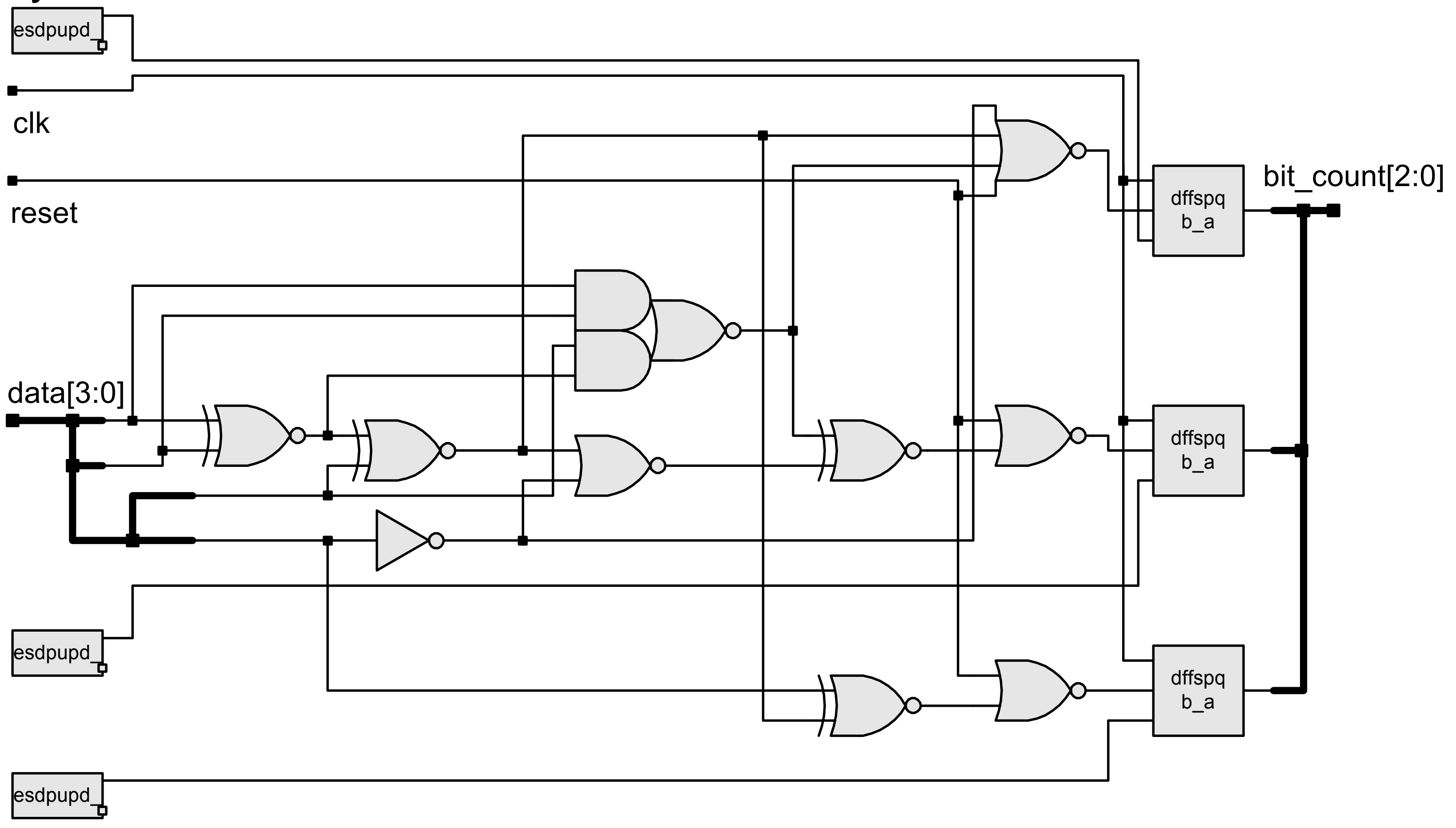
- The loop in the Verilog model *count_ones_a* below has no internal timing controls and is static
- The order in which the statements in the cyclic behavior execute is important
- The model uses the blocked assignment operator (=)
- *bit_count* asserts after the loop has executed, within the same clock cycle that the loop executes

Simulation results:



Note: the displayed values of *temp*, *count* and *bit_count* are final values that result after any intermediate values have been overwritten.

Synthesis results:



Static Loops with Embedded Timing Control

Embedded event control expressions synchronize and distribute the computational activity of a loop over one or more cycles of the clock (implicit state machine with a cycle for each iteration)

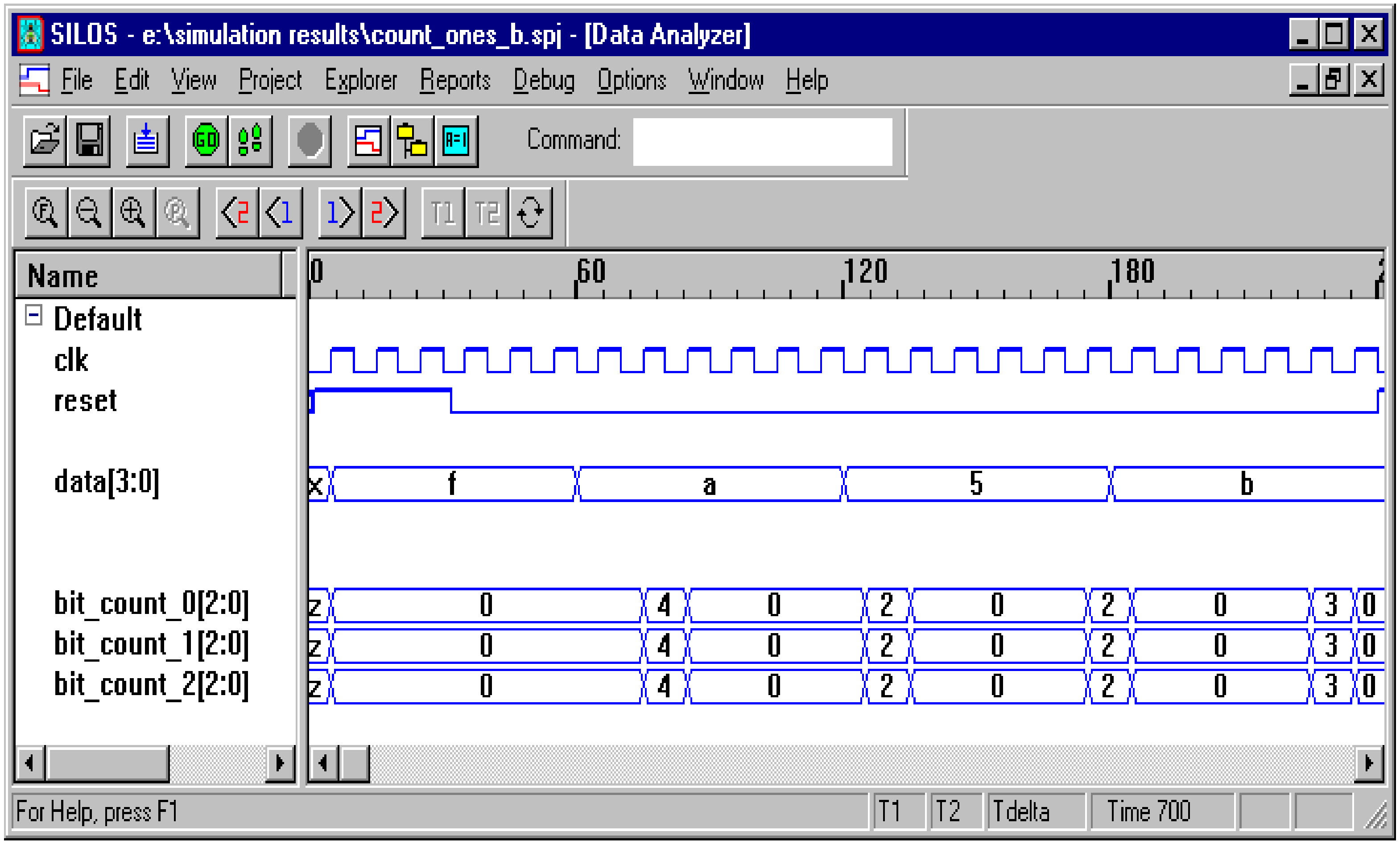
Example 6.39

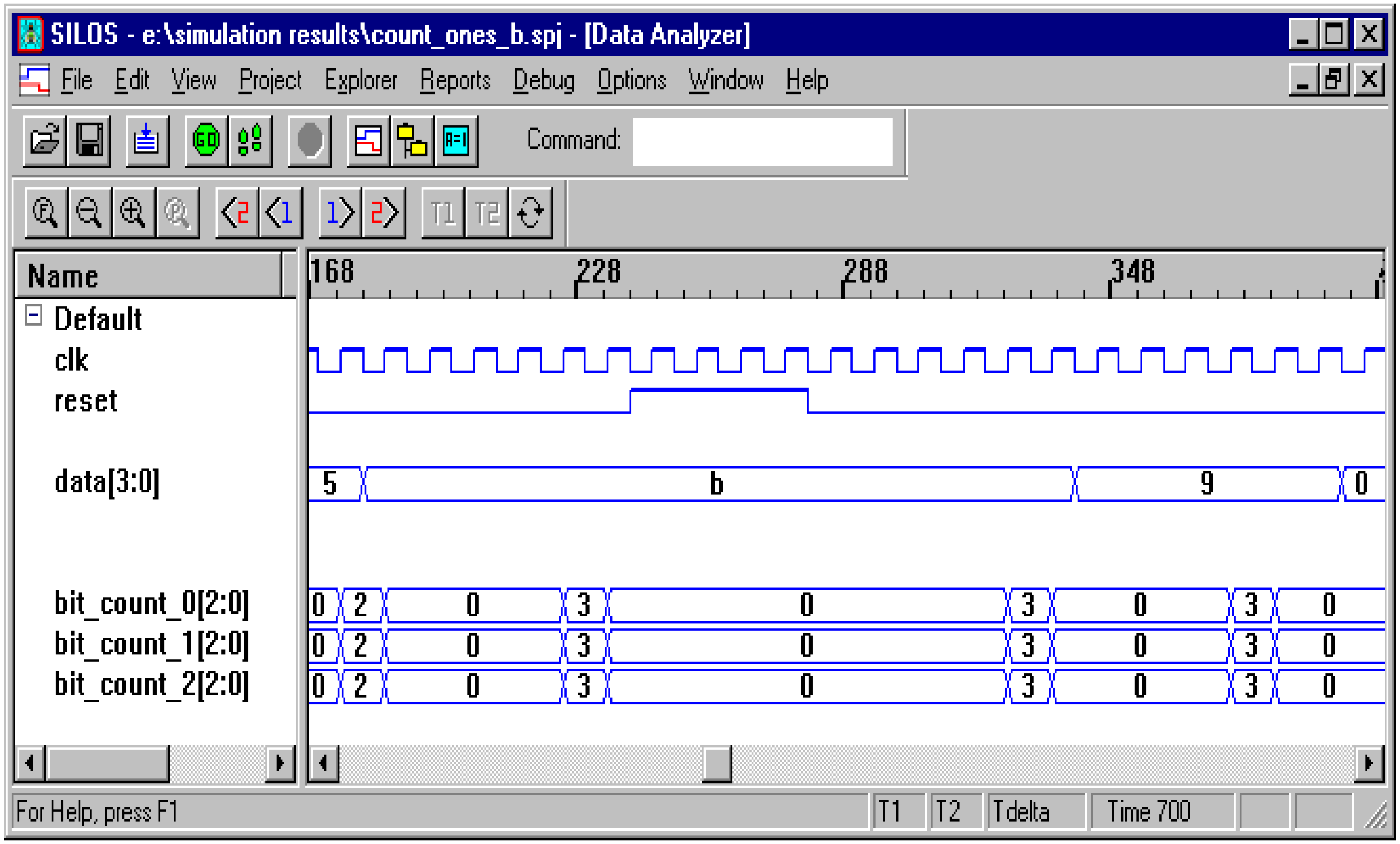
```
module count_ones_b0 (bit_count, data, clk, reset);
  parameter                              data_width = 4;
  parameter                              count_width = 3;
  output       [count_width-1: 0]        bit_count;
  input        [data_width-1: 0]         data;
  input                                  clk, reset;
  reg          [count_width-1: 0]        count, bit_count;
  reg          [data_width-1: 0]         temp;
  integer                                index;
```

```
 always begin: wrapper_for_synthesis
  @ (posedge clk) begin: machine
   if (reset) begin bit_count = 0;  disable machine; end
   else
    count = 0; bit_count = 0; index = 0; temp = data;
    forever  @ (posedge clk)
     if (reset) begin bit_count = 0; disable machine; end
     else if (index < data_width-1) begin
      count = count + temp[0];
      temp = temp >> 1;
      index = index + 1;
         end
         else begin
      bit_count = count + temp[0];
      disable machine;
     end
  end // machine
 end // wrapper_for_synthesis
endmodule
```

```
module count_ones_b1 (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: wrapper_for_synthesis
  @ (posedge clk) begin: machine
   if (reset) begin bit_count = 0; disable machine; end
   else  begin
    count = 0; bit_count = 0; index = 0;temp = data;
    while (index < data_width) begin
     if (reset) begin bit_count = 0; disable machine; end
     else if ((index < data_width) && (temp[0] ))
      count = count + 1;
      temp = temp >> 1;
      index = index +1;
      @ (posedge clk);
    end
    if (reset) begin bit_count = 0; disable machine; end
         else bit_count = count;
    disable machine;
   end
  end // machine
 end // wrapper_for_synthesis
endmodule
```

```
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)
        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
```


SILOS - e:\simulation results\count_ones_b.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
clk
reset
data[3:0]
bit_count_0[2:0]
bit_count_1[2:0]
bit_count_2[2:0]
0
60
120
180
x
f
a
5
b
z
0
4
0
2
0
2
0
3
0
For Help, press F1
T1
T2
Tdelta
Time 700


SILOS - e:\simulation results\count_ones_b.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
clk
reset
data[3:0]
bit_count_0[2:0]
bit_count_1[2:0]
bit_count_2[2:0]
168
228
288
348
5
b
9
0
0 2 0 3 0 3 0 3 0
For Help, press F1
T1
T2
Tdelta
Time 700

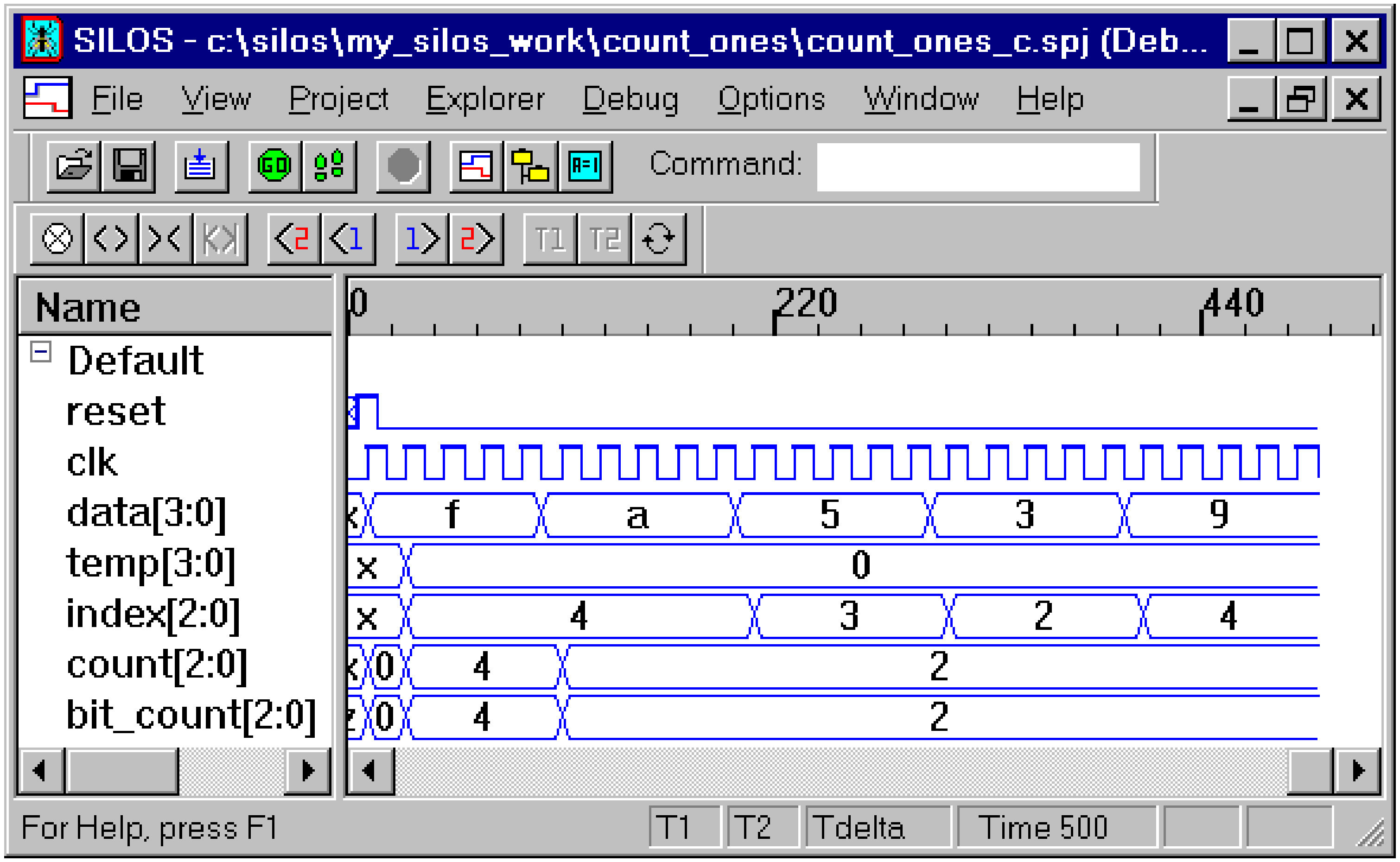
Nonstatic Loops without Embedded Timing Control

The number of iterations to be executed by a loop having a data dependency cannot be determined before simulation.

Example 6.40

```
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
```


SILOS - c:\silos\my_silos_work\count_ones\count_ones_c.spj (Deb...
File View Project Explorer Debug Options Window Help
Command:
Name
Default
reset
clk
data[3:0]
temp[3:0]
index[2:0]
count[2:0]
bit_count[2:0]
0
220
440
x f a 5 3 9
x 0
x 4 3 2 4
x 0 4 2
z 0 4 2
For Help, press F1
T1
T2
Tdelta
Time 500

Nonstatic Loops without Embedded Timing Control

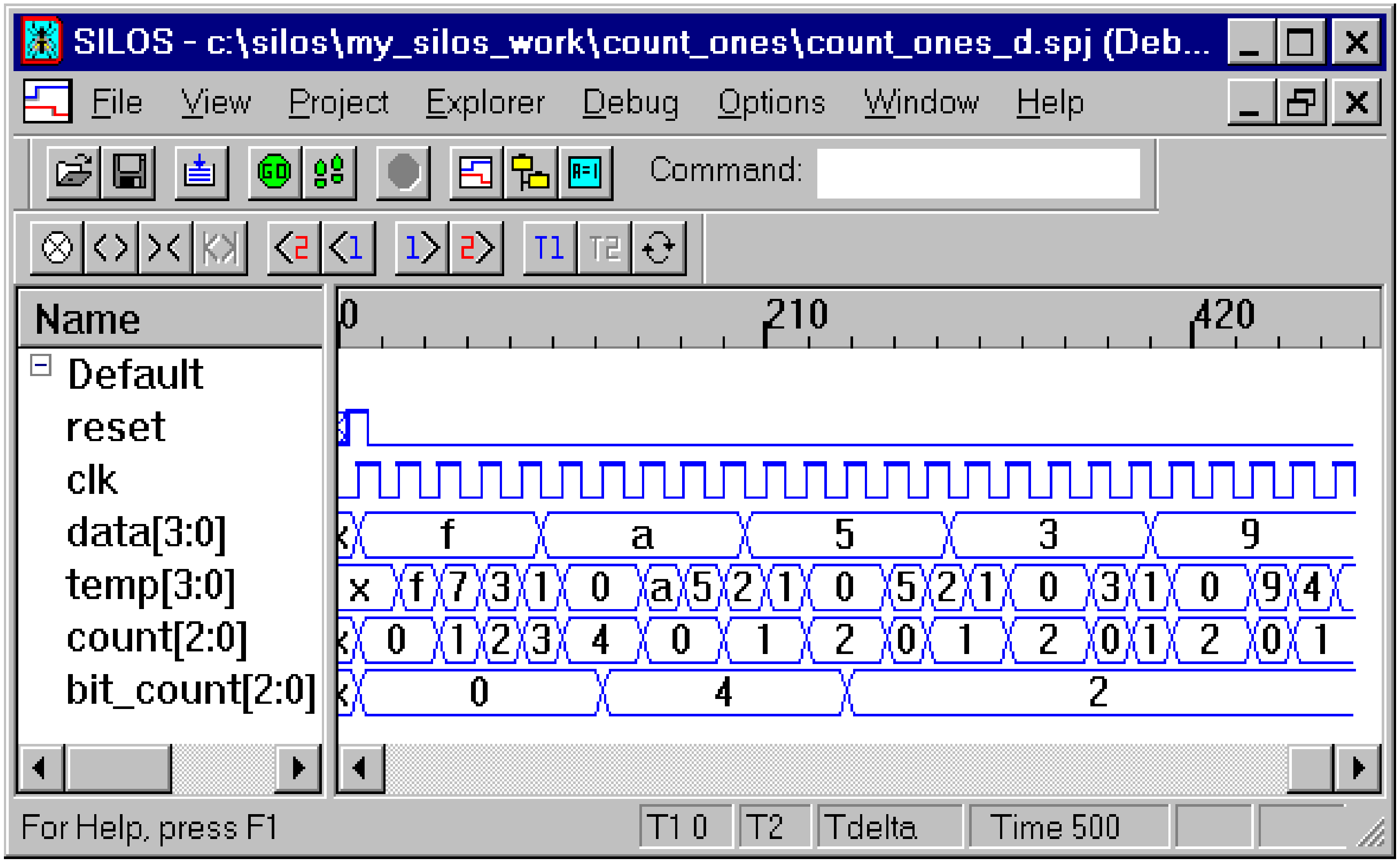
For synthesis , the iterations of a non-static loop must be separated by a synchronizing edge-sensitive control expression.

Example 6.41

```
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;
```

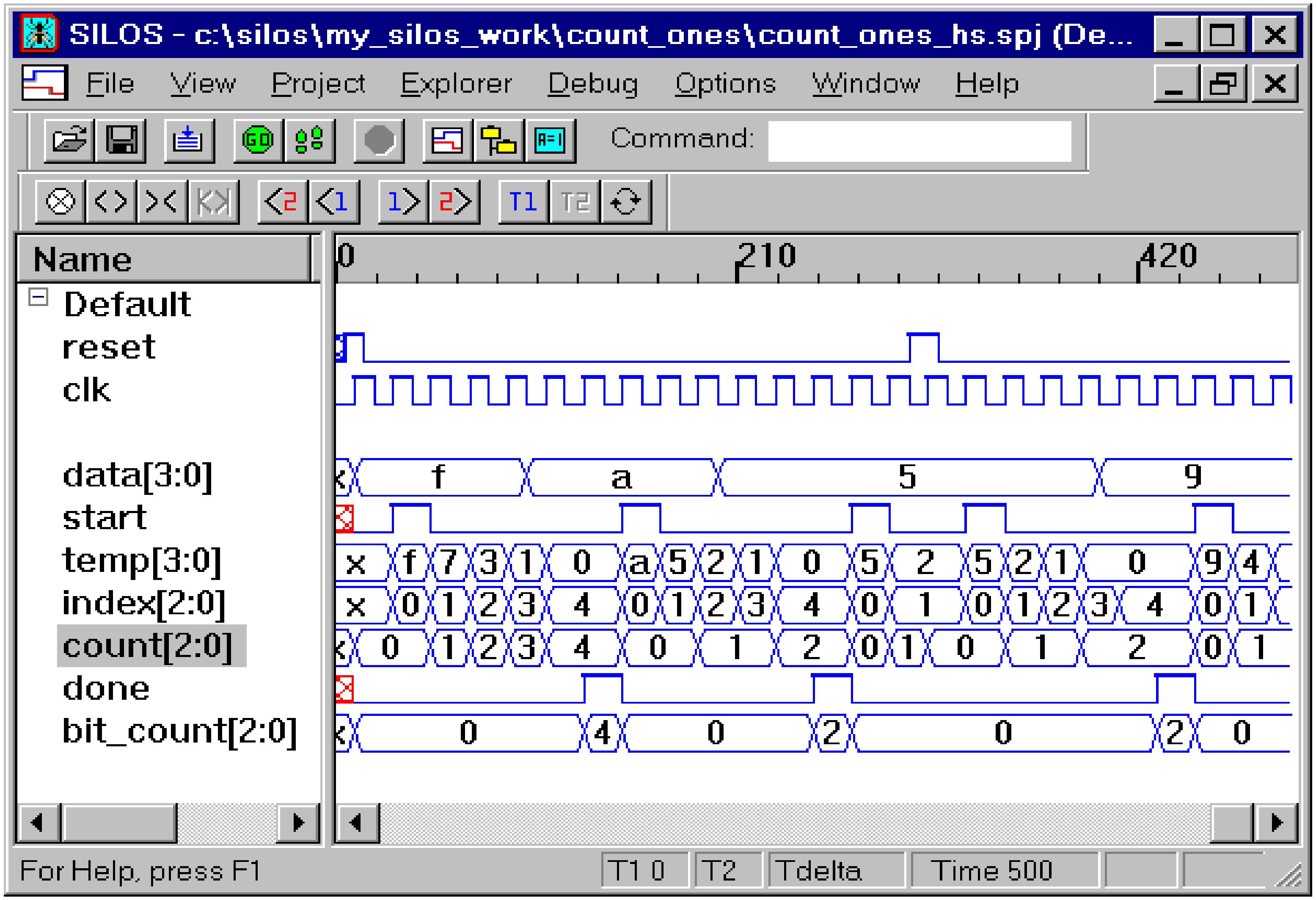
```
always begin: wrapper_for_synthesis
 @ (posedge clk)
  if (reset) begin count = 0; bit_count = 0; end
  else begin: bit_counter
   count = 0;
   temp = data;
   while (temp)
    @  (posedge clk)
     if (reset) begin
      count = 2'b0;
      disable bit_counter; end
     else begin
      count = 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
  endmodule
```


SILOS - c:\silos\my_silos_work\count_ones\count_ones_d.spj (Deb...
File View Project Explorer Debug Options Window Help
Command:
Name
Default
reset
clk
data[3:0]
temp[3:0]
count[2:0]
bit_count[2:0]
0
210
420
x f a 5 3 9
x f 7 3 1 0 a 5 2 1 0 5 2 1 0 3 1 0 9 4
x 0 1 2 3 4 0 1 2 0 1 2 0 1 2 0 1
x 0 4 2
For Help, press F1
T1 0
T2
Tdelta
Time 500

```
module count_ones_SD (bit_count, done, data, start, clk, reset);
 parameter  data_width = 4;
 parameter  count_width = 3;
 output     [count_width-1: 0]     bit_count;
 output                            done;
 input      [data_width-1: 0]      data;
 input                             clk, reset;
 reg        [count_width-1: 0]     count, bit_count, index;
 reg        [data_width-1: 0]      temp;
 reg                               done, start;


 always @ (posedge clk) begin: bit_counter
  if (reset) begin count = 0; bit_count = 0; done = 0; end
  else if (start) begin
   done = 0;
   count = 0;
   bit_count = 0;
   temp = data;
```

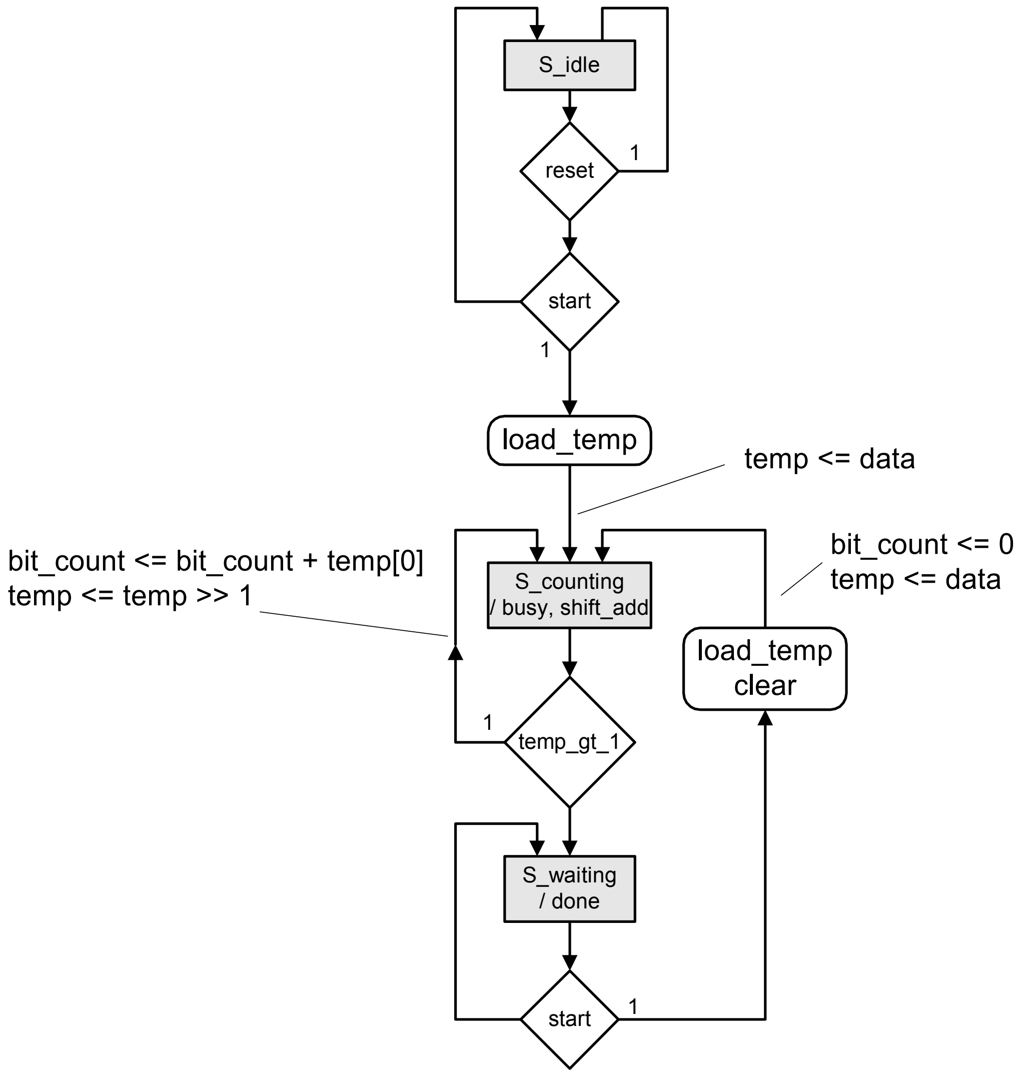
```
      for (index = 0; index < data_width; index = index + 1)
        @ (posedge clk)
        if (reset) begin count = 0; bit_count = 0; done = 0;
          disable bit_counter; end
        else begin
          count = count + temp[0];
          temp = temp >> 1;
        end

      @  (posedge clk)  // Required for final register transfer
       if (reset) begin  count = 0; bit_count = 0; done = 0;\
        disable bit_counter; end
      else begin
        bit_count = count;
        done = 1; end
    end
   end
endmodule
```


SILOS - c:\silos\my_silos_work\count_ones\count_ones_hs.spj (De...
File View Project Explorer Debug Options Window Help
Command:
Name
Default
reset
clk
data[3:0]
start
temp[3:0]
index[2:0]
count[2:0]
done
bit_count[2:0]
0
210
420
For Help, press F1
T1 0
T2
Tdelta
Time 500

State Machine Replacements for Unsynthesizable Loops

Replace loop structures of non-static loops that don't have embedded timing controls by equivalent synthesizable sequential behavior.

Example 6.42


S_idle
reset
1
start
1
load_temp
temp <= data
bit_count <= bit_count + temp[0]
temp <= temp >> 1
S_counting
/ busy, shift_add
bit_count <= 0
temp <= data
load_temp
clear
1
temp_gt_1
S_waiting
/ done
start
1

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

  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,  busy,  done,  start,  temp_gt_1,  clk,
reset);
  datapath M1 (temp_gt_1, temp_0, data, load_temp, shift_add, clk, reset);
  bit_counter_unit M2 (bit_count, temp_0, clear, clk, reset);

endmodule
```

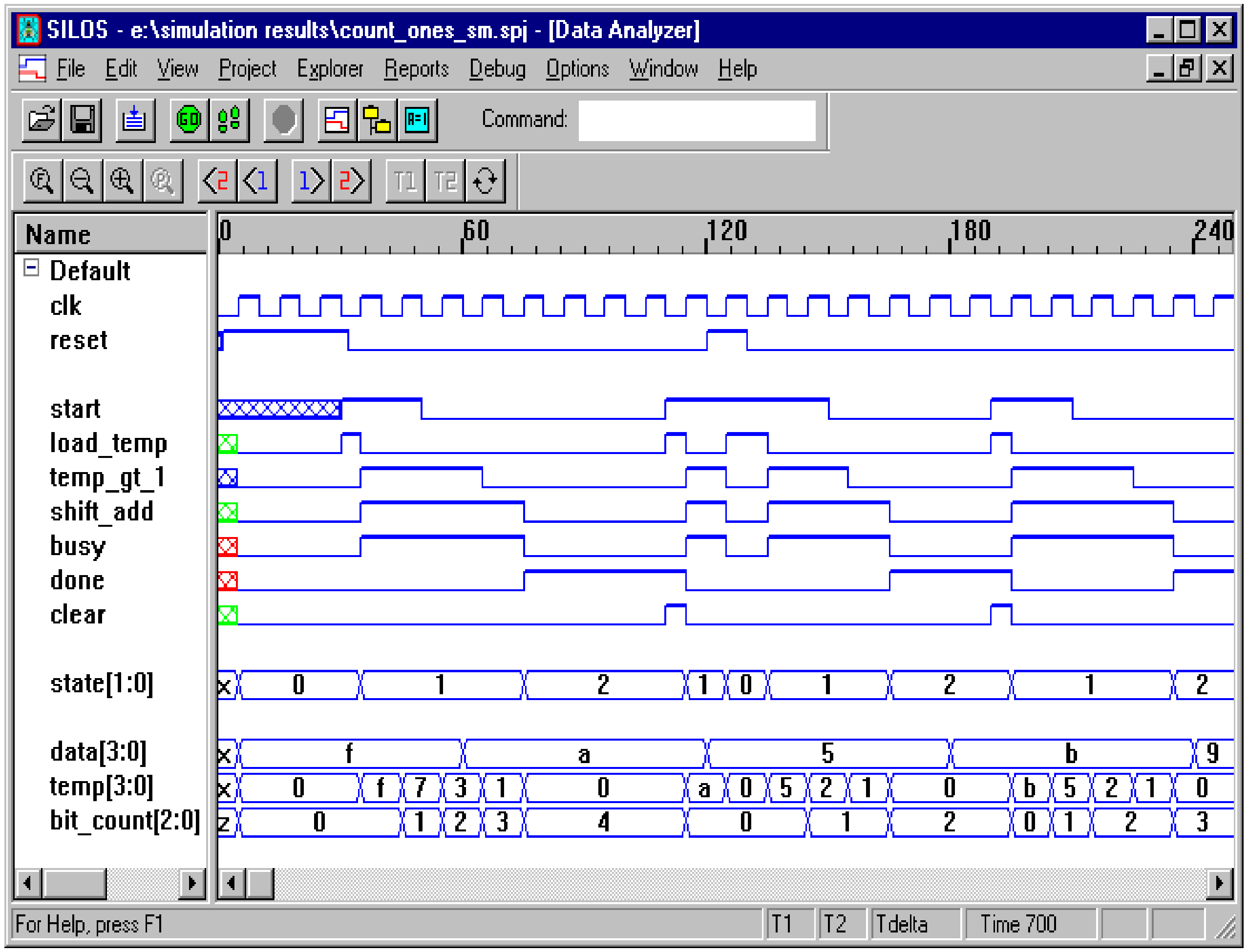
```
module controller (load_temp, shift_add, clear, busy, done, start, temp_gt_1, clk,
reset);
  parameter                        state_size = 2;
  parameter                        S_idle = 0;
  parameter                        S_counting = 1;
  parameter                        S_waiting = 2;

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

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

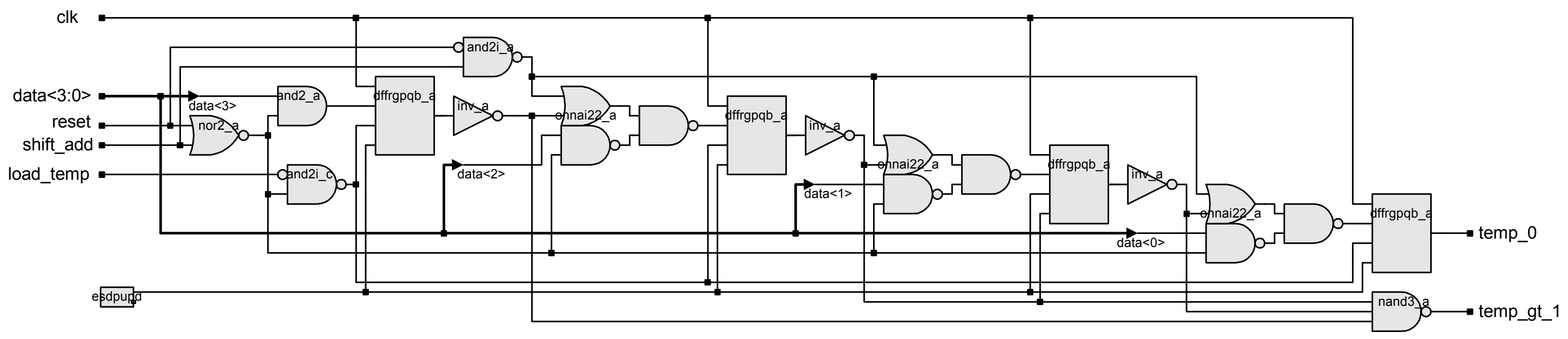
```
always @ (state or start or temp_gt_1) begin
  load_temp = 0; shift_add = 0; done = 0;
  busy = 0; clear = 0; next_state = S_idle;

  case (state)
   S_idle:    if (start) begin next_state = S_counting; load_temp = 1; end

   S_counting:    begin busy = 1; if (temp_gt_1) begin next_state = S_counting;
            shift_add = 1; end
          else begin next_state = S_waiting; shift_add = 1;  end
          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
```

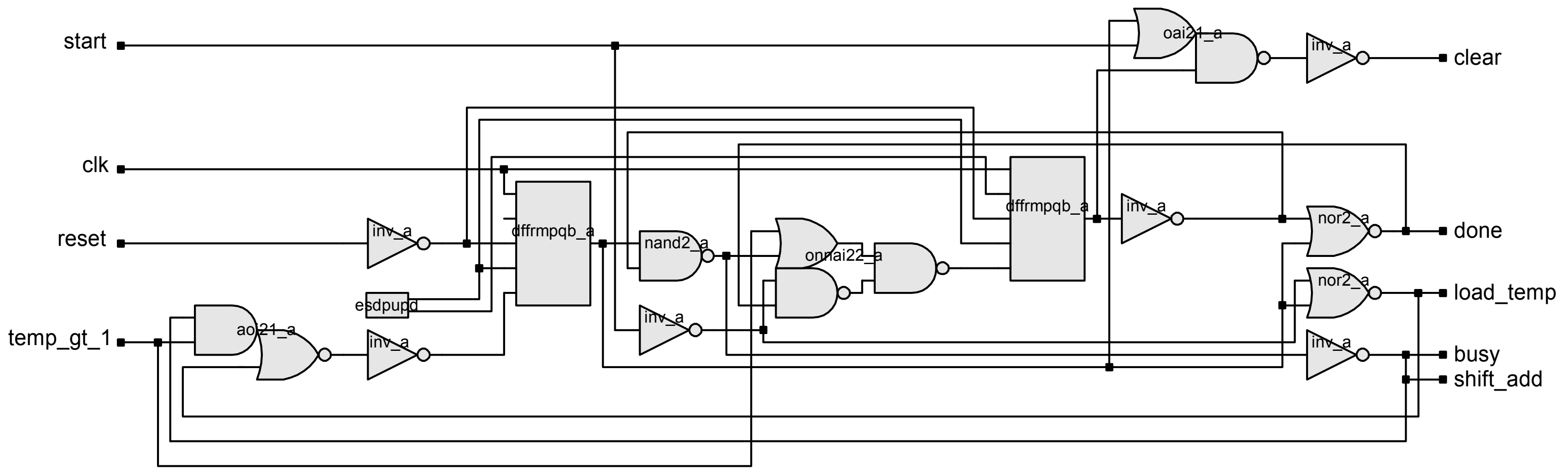
```
  always @ (posedge clk) // state transitions
    if (reset)
      state <= S_idle;
    else state <= next_state;
endmodule
```

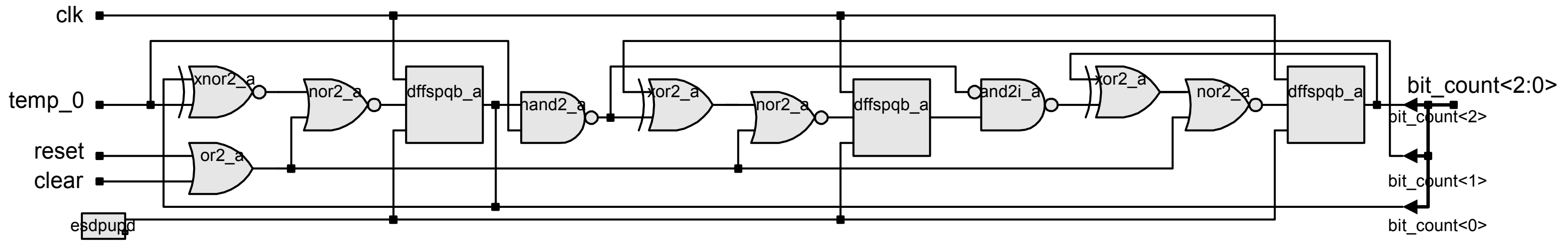
```
module datapath (temp_gt_1, temp_0, data, load_temp, shift_add, clk, reset);
 parameter                                word_size = 4;
 output                                   temp_gt_1, temp_0;
 input        [word_size-1: 0]            data;
 input                                    load_temp, shift_add,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

  reg         [word_size-1: 0]            temp;
 wire                                     temp_gt_1 = (temp > 1);
 wire                                     temp_0 = temp[0];

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

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

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


SILOS - e:\simulation results\count_ones_sm.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
clk
reset
start
load_temp
temp_gt_1
shift_add
busy
done
clear
state[1:0]
data[3:0]
temp[3:0]
bit_count[2:0]
0
60
120
180
240
For Help, press F1
T1
T2
Tdelta
Time 700


clk
data<3:0>
reset
shift_add
load_temp
data<3>
and2_a
nor2_a
and2i_a
and2i_c
dffrgpqb_a
inv_a
data<2>
oannai22_a
data<1>
data<0>
esdpupd
nand3_a
temp_0
temp_gt_1

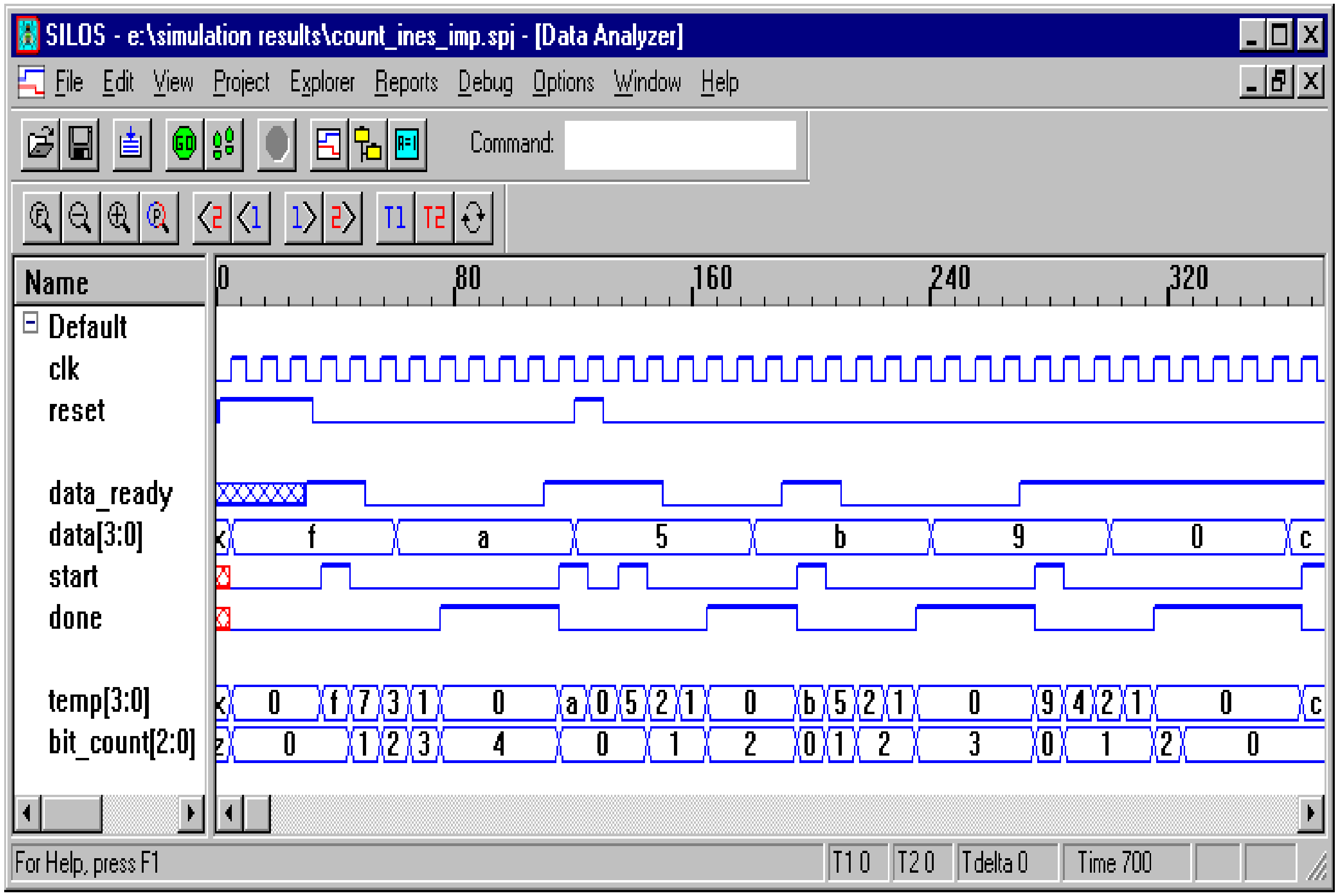

start
clk
reset
temp_gt_1
inv_a
esdpupd
aoi21_a
inv_a
dffrmpqb_a
nand2_a
inv_a
oai21_a
onnai22_a
dffrmpqb_a
inv_a
inv_a
nor2_a
nor2_a
inv_a
clear
done
load_temp
busy
shift_add


clk
temp_0
reset
clear
esdpupd
xnor2_a
or2_a
nor2_a
dffspqb_a
nand2_a
xor2_a
nor2_a
dffspqb_a
and2i_a
xor2_a
nor2_a
dffspqb_a
bit_count<2:0>
bit_count<2>
bit_count<1>
bit_count<0>

Example 6.43 Implicit State Machine

```
module count_ones_IMP (bit_count, start, done, data, data_ready, clk,
reset);
 parameter                                word_size = 4;
 parameter                                counter_size = 3;
 parameter                                state_size = 2;
 output     [counter_size -1 : 0]     bit_count;
 output                                   start, done;
 input      [word_size-1: 0]          data;
 input                                    data_ready, clk, reset;

 reg                                      bit_count;
 reg        [state_size-1 : 0]        state, next_state;
 reg                                      start, done, clear;
 reg        [word_size-1: 0]          temp;
```

```
always @ (posedge clk) if (reset)
    begin temp<= 0; bit_count <= 0; done <= 0; start <= 0; end
  else if (data_ready && data && !temp)
    begin temp <= data; bit_count <= 0; done <= 0; start <= 1; end
  else if (data_ready && (!data) && done)
    begin bit_count <= 0; done <= 1; end
  else if (temp == 1)
    begin bit_count <= bit_count + temp[0]; temp <= temp >> 1; done <=
1; end
   else if (temp && !done)
    begin start <= 0; temp <= temp >> 1; bit_count <= bit_count +
temp[0]; end
endmodule
```


SILOS - e:\simulation results\count_ines_imp.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
0 80 160 240 320
Default
clk
reset
data_ready
data[3:0]
start
done
temp[3:0]
bit_count[2:0]
For Help, press F1
T1 0
T2 0
Tdelta 0
Time 700