硬體描述語言

Chapter 05

Synthesis of Sequential Logic and Language Construct

Ren-Der Chen (陳仁德)
Department of Computer Science and Information Engineering
National Changhua University of Education
E-mail: rdchen@cc.ncue.edu.tw

Fall, 2024

Synthesis of Sequential Logic with Latches (1/2)

- Latches are synthesized
  - Intentionally
  - Accidentally
- Latch-free logic
  - A feedback-free netlist of combinational primitives will synthesize into latch-free combinational logic.
  - A set of feedback-free continuous assignments will synthesize into latch-free combinational logic.
  - A description of combinational logic must assign value to the outputs for ***all*** possible values of the inputs.

Synthesis of Sequential Logic with Latches (2/2)

- Latch-inferred logic
  - A continuous assignment using a conditional operator with feedback will synthesize into a latch.

Ex. **assign** data_out = (CS_b == 0) ?

(WE_b == 0) ? data_in : data_out : 1’bz;

- *Transparent mode* (CB_b == 0 and WE_b == 0): data_out follows data_in.
- *Latched mode* (CB_b == 0 and WE_b == 1): data_out = data_out.
- *Three-state, high-impedance* (CB_b == 1).

Synthesis of Latches (1/3)

- Latch is level-sensitive.
- The output is affected by the input only when a control signal is asserted.
- At other times, the input is ignored and the output retains its residual value.
- A latch is inferred by the synthesis tool when it detects a level-sensitive behavior in which a register variable is assigned value in some threads of activity, but not others (e.g., an incomplete **if** statement).

Synthesis of Latches (2/3)

- If the activity flow assigns value to a given register variable in all possible threads of the activity, a latch will be inferred if a path assigns a variable its own value, i.e., self-feedback.
- In synthesis, latches implement incompletely-specified assignments to register variables in **case** and **if** statements in a level-sensitive cyclic behavior.
- If a **case** statement has a default assignment with feedback (the variable is explicitly assigned to itself), the synthesis tool will choose a mux structure with feedback.

Synthesis of Latches (3/3)

- If the behavior is edge-sensitive, incomplete **case** and **if** statements synthesize register variable to flip-flops.
- If the statements are completed with feedback, the result is a register whose output is fed back through a mux at its data path.
- The functionality of a latch is also inferred when the conditional operator (? ... :) is implemented with feedback.

Synthesis of Latches - case (1/3)

```
module latch_case_assign (latch_in, enable, set, clear, latch_out);
    input       latch_in, enable, set, clear;
    output      latch_out;
    reg         latch_out;
    //  Efficient in simulation, but does not synthesize with some tools
    always @ (enable or set or clear)
        case ({enable, set, clear})
            3'b100: assign latch_out = latch_in;     // Transparent mode
            3'b110: assign latch_out = 1'b1;         // Set
            3'b010: assign latch_out = 1'b1;         // Set
            3'b101: assign latch_out = 1'b0;         // Clear
            3'b001: assign latch_out = 1'b0;         // Clear
            default: deassign latch_out;             // Holds residual value
        endcase
endmodule
```

Synthesis of Latches - case (2/3)

```
module latch_case1 (latch_in, enable, set, clear, latch_out);
    input           latch_in, enable, set, clear;
    output          latch_out;
    reg             latch_out;
    //  Less efficient in simulation, but can be synthesized by all tools
    always @ (latch_in or enable or set or clear)
        case ({enable, set, clear})
            3'b100: latch_out = latch_in;      // Transparent mode
            3'b110: latch_out = 1'b1;          // Set
            3'b010: latch_out = 1'b1;          // Set
            3'b101: latch_out = 1'b0;          // Clear
            3'b001: latch_out = 1'b0;          // Clear
            default: latch_out = latch_out;    // Holds residual value
        endcase   // The case statement is completed with default feedback.
endmodule
```

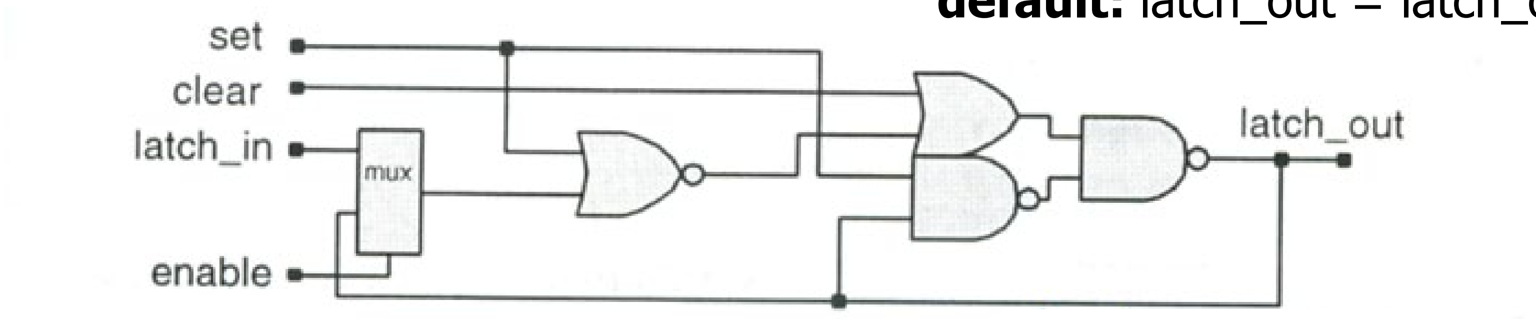
Synthesis of Latches - case (3/3)

```
module latch_case2 (latch_in, enable, set, clear, latch_out);
   input        latch_in, enable, set, clear;
   output       latch_out;
   reg          latch_out;

   always @ (latch_in or enable or set or clear)
      case ({enable, set, clear})
         3'b100: latch_out = latch_in;     // Transparent mode
         3'b110: latch_out = 1'b1;         // Set
         3'b010: latch_out = 1'b1;         // Set
         3'b101: latch_out = 1'b0;         // Clear
         3'b001: latch_out = 1'b0;         // Clear
         // No default value
      endcase   // Incompletely-specified case statement => latch
endmodule
```

All the three designs exhibit correct transparent behavior!

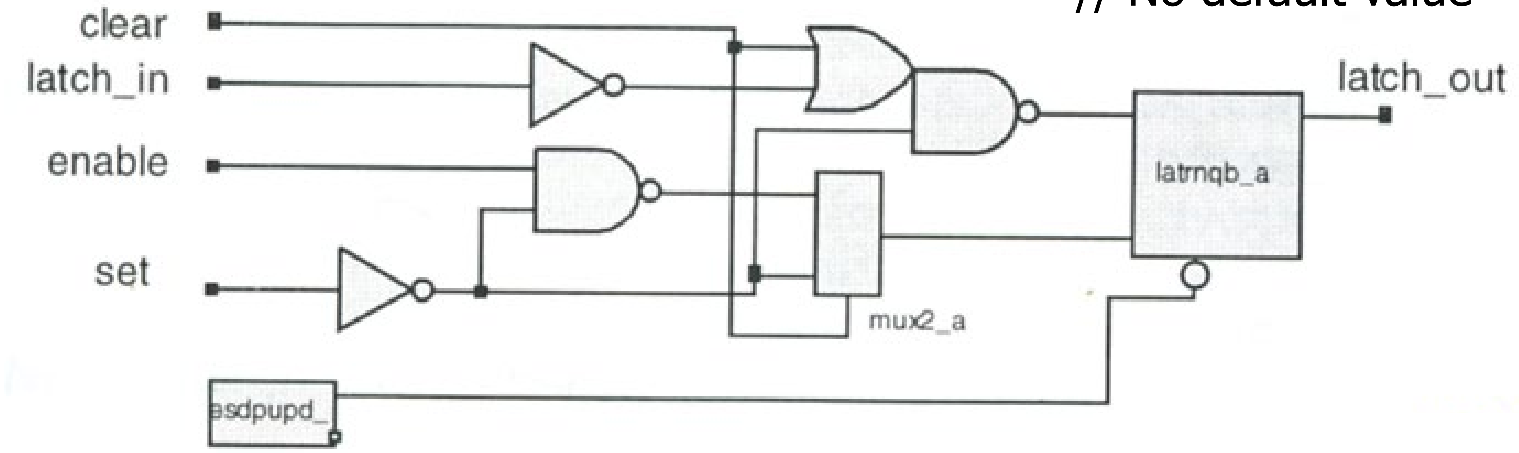
Synthesis Results

default: latch_out = latch_out;



latch_case1

// No default value



latch_case2

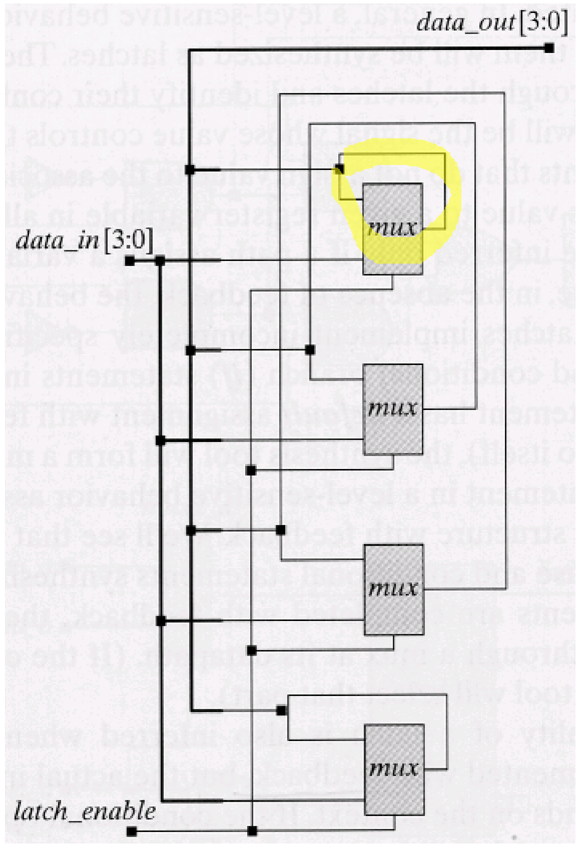
Under some specific synthesis tool and technology

Synthesis of Latches - if (1/3)

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

    always @ (latch_enable)
        if (latch_enable)
            assign      data_out = data_in;
        else
            deassign    data_out;
endmodule
```

Synthesis of Latches - if (2/3)

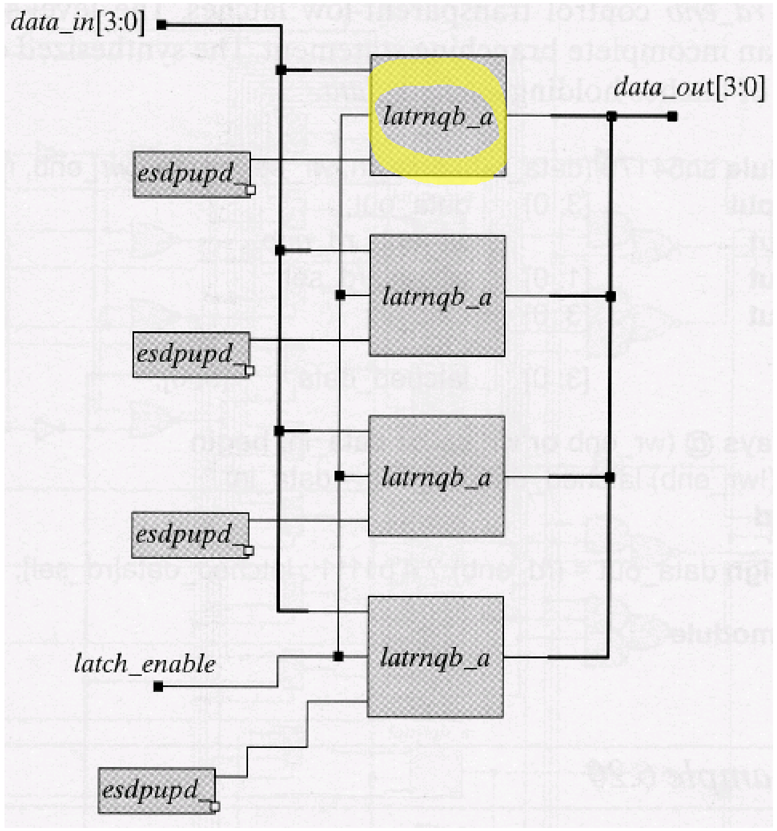
```
module latch_if1 (data_in, latch_enable, data_out);
    input     [3:0]   data_in;
    input             latch_enable;
    output    [3:0]   data_out;
    reg       [3:0]   data_out;
    // Designed with feedback
    always @ (data_in or latch_enable)
        if (latch_enable)
            data_out = data_in;
        else
            data_out = data_out;
endmodule
```



```
assign data_out[3:0] = latch_enable ? data_in[3:0] : data_out[3:0];
```

Synthesis of Latches - if (3/3)

```
module latch_if2 (data_in, latch_enable, data_out);
    input     [3:0]   data_in;
    input             latch_enable;
    output    [3:0]   data_out;
    reg       [3:0]   data_out;
    // Incompletely specified
    always @ (data_in or latch_enable)
        if (latch_enable)
            data_out = data_in;
endmodule
```



Latch_if1 and latch_if2 are equivalent in simulation, but the physical circuit will have different area/speed trade-offs.

A Four-Bit OR Gate without Latch

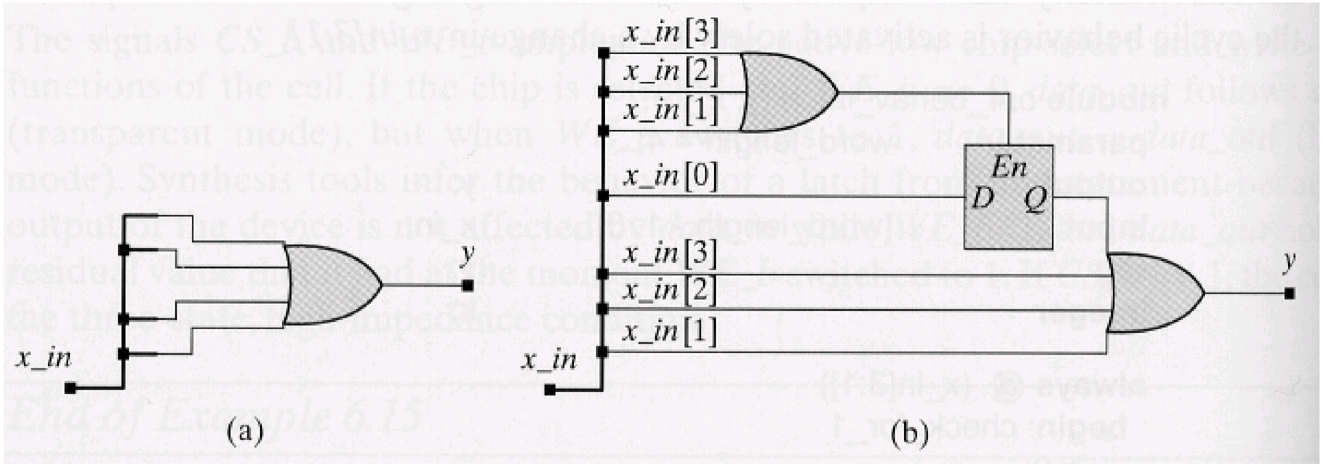
```
module or4_behav (x_in, y);
    parameter     word_length = 4;
    input         [word_length-1:0] x_in;
    output        y;
    reg           y;
    integer       k;

    always @ (x_in) begin: check_for_1
      y = 0;
      for (k = 0; k <= word_length -1; k = k+1)
        if (x_in[k] == 1) begin
          y = 1;
          disable check_for_1;
        end
    end
endmodule
```

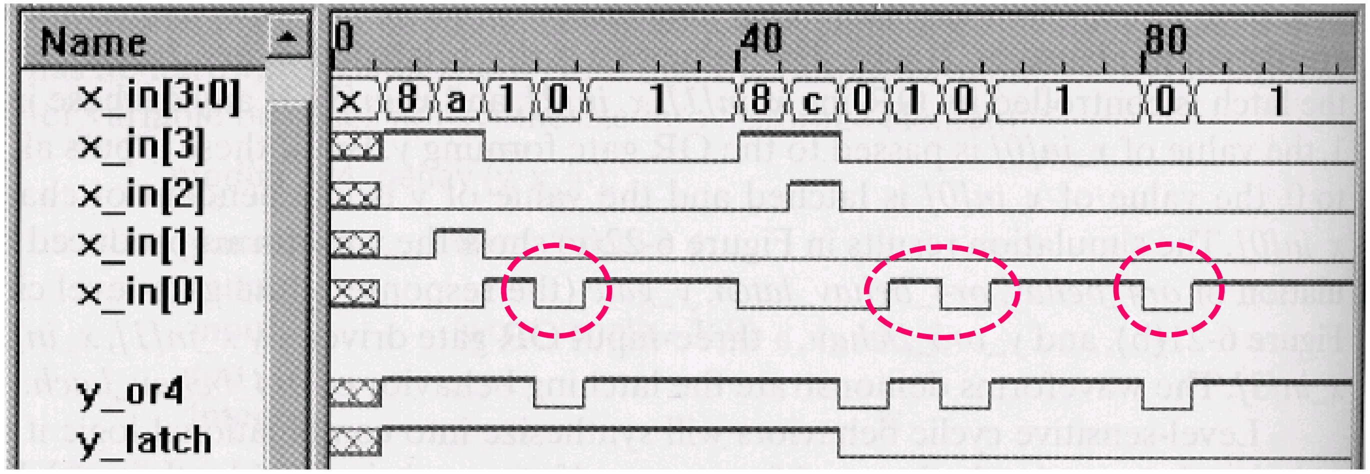
A Four-Bit OR Gate with Latched Output

```
module or4_behav_latch (x_in, y);
    parameter    word_length = 4;
    input        [word_length-1:0] x_in;
    output       y;
    reg          y;
    integer      k;

    always @ (x_in [3:1]) begin: check_for_1
      y = 0;
      for (k = 0; k <= word_length -1; k = k+1)
        if (x_in[k] == 1) begin
          y = 1;
          disable check_for_1;
        end
    end
endmodule
```

Accidental synthesis of latches !!

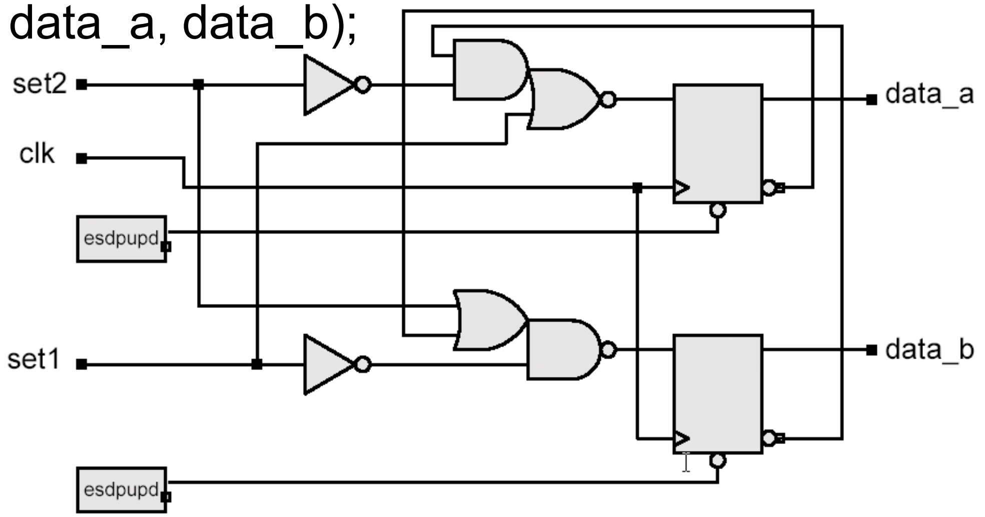
Synthesis and Simulation Results





Two Flip-Flops for Swapping Values

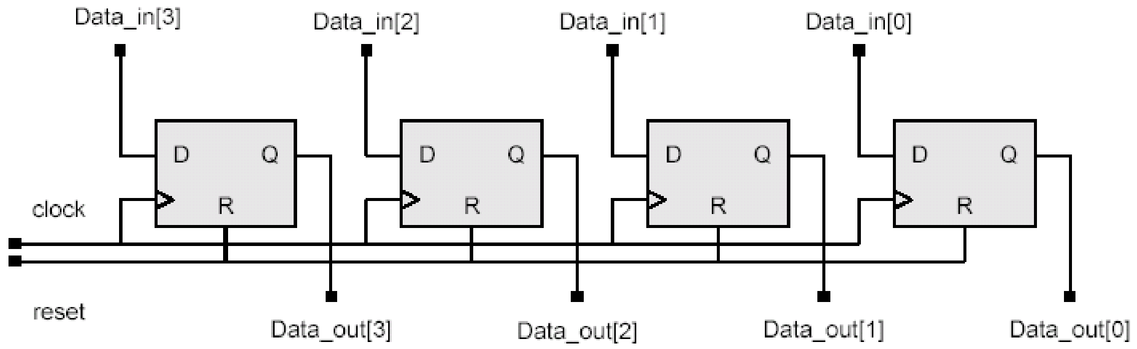
```
module swap_synch (clk, set1, set2, data_a, data_b);
    input       clk, set1, set2;
    output      data_a, data_b;
    reg         data_a, data_b;

    always @ (posedge clk) begin
      if (set1) begin
          data_a <= 0; data_b <= 1;
      end
      else if (set2) begin
          data_a <= 1; data_b <= 0;
      end
      else begin
          data_b <= data_a; data_a <= data_b;
      end
    end
endmodule
```



- A register variable will be synthesized as the output of a flip-flop when its value is assigned synchronously with *the edge of a signal*. (edge-triggered)

Four-Bit Register (1/2)

```
module D_reg4_a (Data_in, clock, reset, Data_out);
    input       [3:0]   Data_in;
    input               clock, reset;
    output      [3:0]   Data_out;
    reg         [3:0]   Data_out;

    always @ (posedge clock or posedge reset)
    if (reset == 1'b1)
      Data_out <= 4'b0;
    else
      Data_out <= Data_in;
endmodule
```



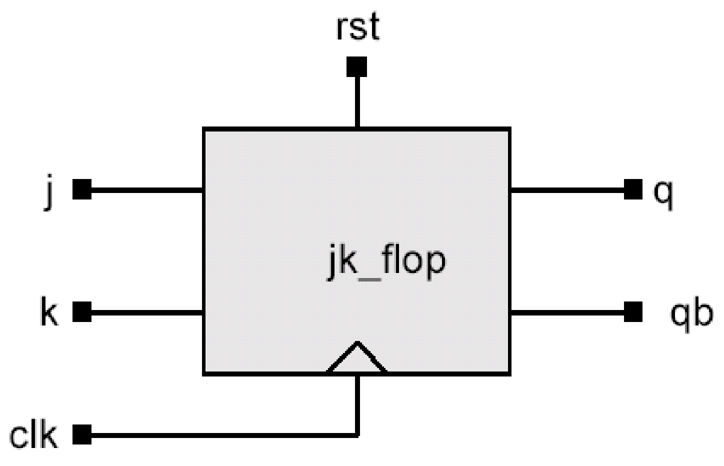
Four-Bit Register (2/2)

```
module D_reg4_b (Data_in, clock, reset, Data_out);
    input       [3:0]   Data_in;
    input               clock, reset;
    output      [3:0]   Data_out;
    reg         [3:0]   Data_out;

    always @ (posedge clock)
        Data_out <= Data_in;
    always @ (reset)
        if (reset)
            assign Data_out = 4'b0;
        else
            deassign Data_out;
endmodule
```

Maybe an unsupported description !!

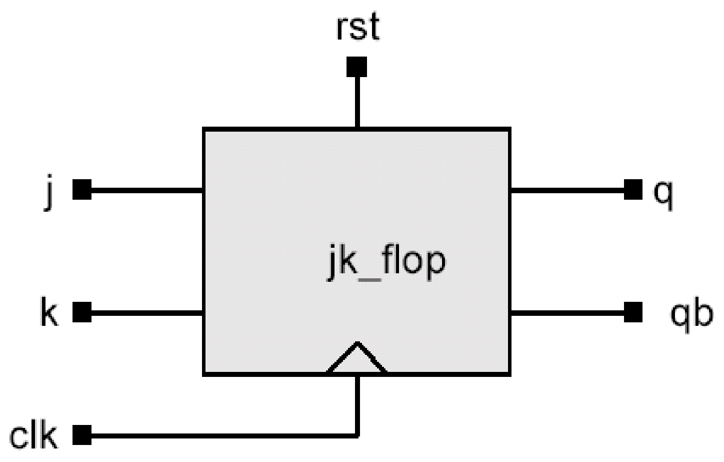
JK Flip-Flops (1/2)

```
module jk_flop_1 (j, k, clock, rst, q, qb);
    input        j, k, clock, rst;
    output       q, qb;
    reg          q;
    assign       qb = ~q;

    always @ (posedge clock or posedge rst) begin
        if (rst == 1'b1) q = 1'b0;
        else if (j == 1'b0 && k == 1'b0) q = q;
        else if (j == 1'b0 && k == 1'b1) q = 1'b0;
        else if (j == 1'b1 && k == 1'b0) q = 1'b1;
        else if (j == 1'b1 && k == 1'b1) q = ~q;
    end
endmodule
```



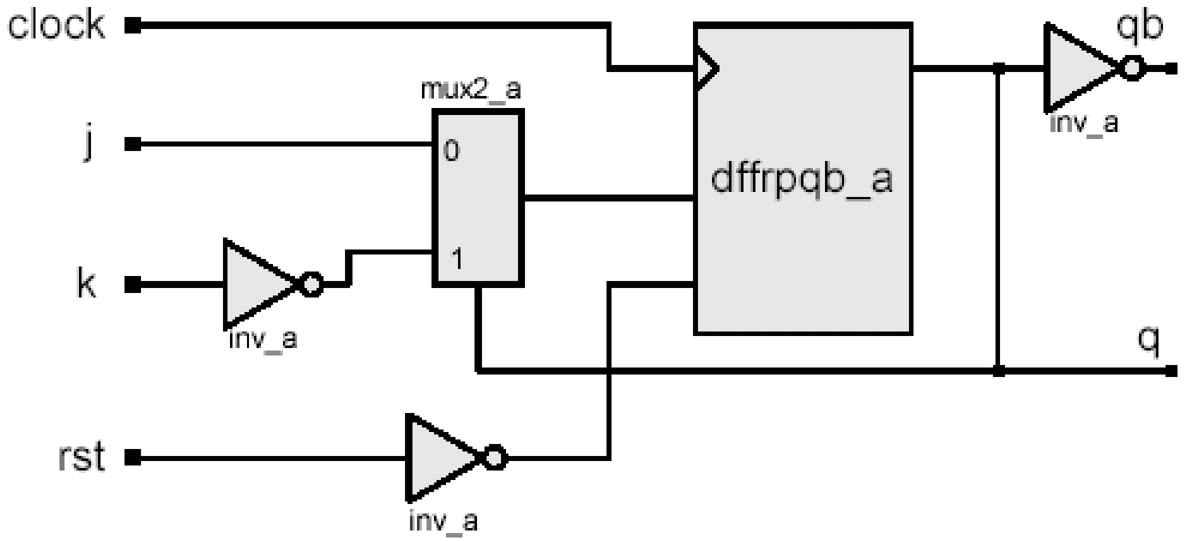
JK Flip-Flops (2/2)

```
module jk_flop_2 (j, k, clock, rst, q, qb);
    input           j, k, clock, rst;
    output          q, qb;
    reg             q;
    assign          qb = ~q;
    always @ (posedge clock or posedge rst) begin
        if (rst == 1'b1)
          q = 1'b0;
        else
          case ({j, k})
            2'b00: q = q;
            2'b01: q = 1'b0;
            2'b10: q = 1'b1;
            2'b11: q = ~q;
          endcase
    end
endmodule
```



Synthesis Result

```
case ({j, k})
    2'b00: q = q;
    2'b01: q = 1'b0;
    2'b10: q = 1'b1;
    2'b11: q = ~q;
```

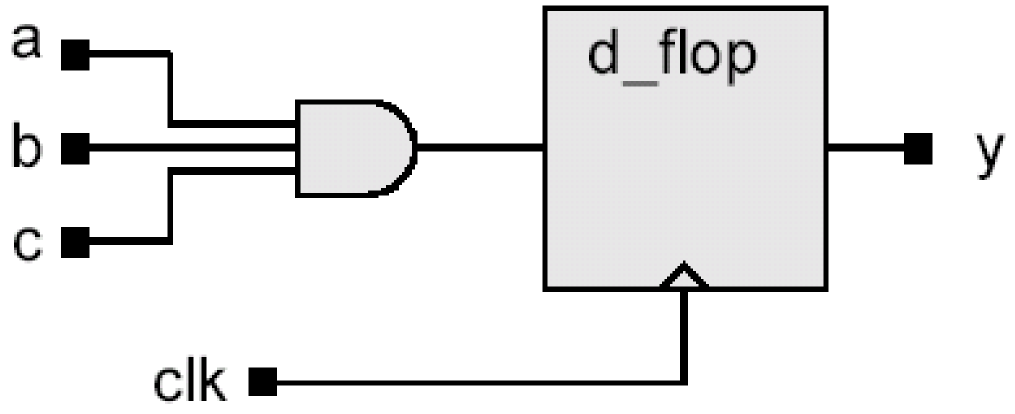


The synthesis tool uses a D-type flip-flop with input logic to decode JK inputs.

Under some specific synthesis tool and technology

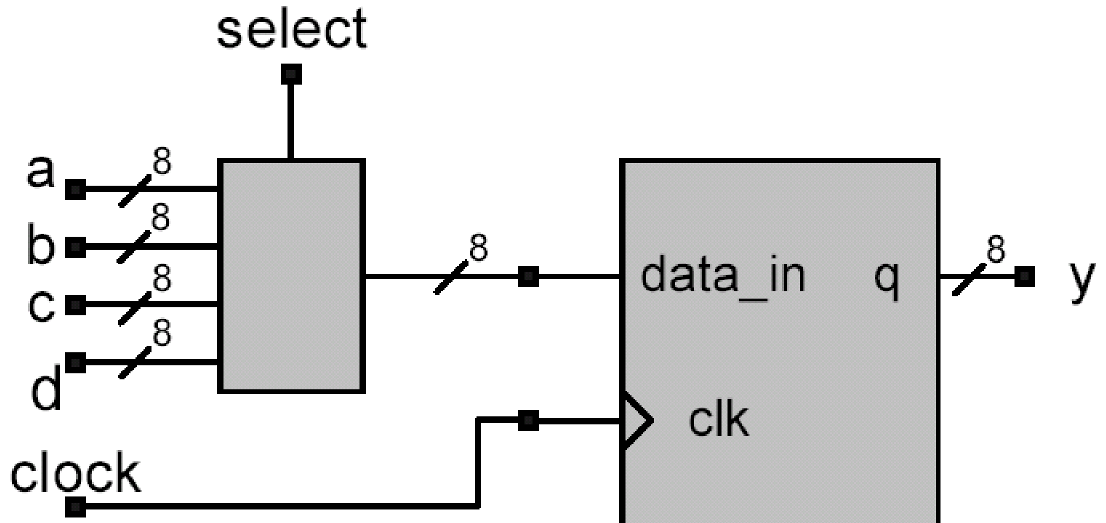
Registered AND Gate

```
module reg_and (a, b, c, clk, y);
    input       a, b, c, clk;
    output      y;
    reg         y;

    always @ (posedge clk)
        y <= a & b & c;
endmodule
```



Registered Multiplexer

```
module mux_reg (a, b, c, d, select, clock, y);
    input          [7:0]     a, b, c, d;
    input          [1:0]     select;
    input          clock;
    output         [7:0]     y;
    reg            [7:0]     y;

    always @(posedge clock)
      case (select)
          0:          y <= a;
          1:          y <= b;
          2:          y <= c;
          3:          y <= d;
          default:  y <= 8'bx;
      endcase
endmodule
```



Physical Size from Synthesis

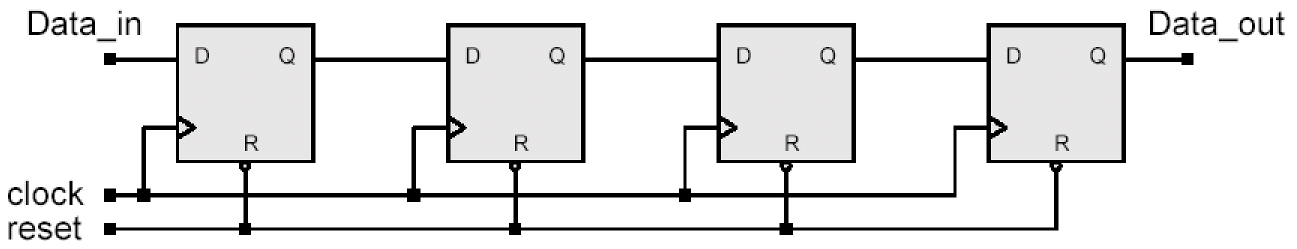
- The physical size of an ASIC synthesized from Verilog is not necessarily proportional to the size of the source code.

```
always @ (posedge clock) begin
    data_register <= data_bus;
end
```

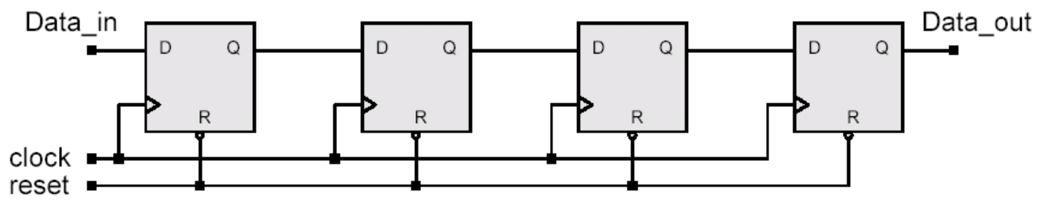
- data_register could be a single bit or a 32-bit register.
- The physical part selected in the technology mapping depends on the robustness of the target library.

Shift Register 1

```
module Shift_reg4_1_1 (Data_in, clock, reset, Data_out);
    input           Data_in, clock, reset;
    output          Data_out;
    reg       [3:0] Data_reg;

    assign          Data_out = Data_reg[0];

    always @ (posedge clock or negedge reset)
        if (reset == 1'b0) Data_reg <= 4'b0;
        else Data_reg <= {Data_in, Data_reg[3:1]};
endmodule
```



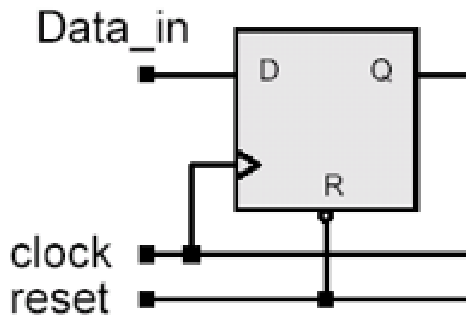
Shift Register 2



```
module Shift_reg4_1_2 (Data_in, clock, reset, Data_out);
    input          Data_in, clock, reset;
    output         Data_out;
    reg      [3:0] Data_reg;

    assign         Data_out = Data_reg[0];

    always @ (posedge clock or negedge reset)
        if (reset == 1'b0) Data_reg <= 4'b0;
        else begin // in the order of row by row
            Data_reg[3] <= Data_in;      Data_reg[2] <= Data_reg[3];
            Data_reg[1] <= Data_reg[2]; Data_reg[0] <= Data_reg[1];
        end
endmodule
```

Shift Register 3

```
module Shift_reg4_2_2 (Data_in, clock, reset, Data_out);
    input           Data_in, clock, reset;
    output          Data_out;
    reg       [3:0] Data_reg;

    assign          Data_out = Data_reg[0];

    always @ (posedge clock or negedge reset)
        if (reset == 1'b0) Data_reg = 4'b0;
        else begin // in the order of row by row
            Data_reg[3] = Data_in;      Data_reg[2] = Data_reg[3];
            Data_reg[1] = Data_reg[2];  Data_reg[0] = Data_reg[1];
        end
endmodule
```



Only one register is synthesized!

Parallel Load Register (1/2)

```
module Par_load_reg4_1_1 (Data_in, load, clock, reset, Data_out);
    input       [3:0]   Data_in;
    input               load, clock, reset;
    output      [3:0]   Data_out;
    reg         [3:0]   Data_out;

    always @ (posedge clock or posedge reset)
      if (reset == 1'b1)
          Data_out <= 4'b0;
      else if (load == 1'b1)
          Data_out <= Data_in;
endmodule
```

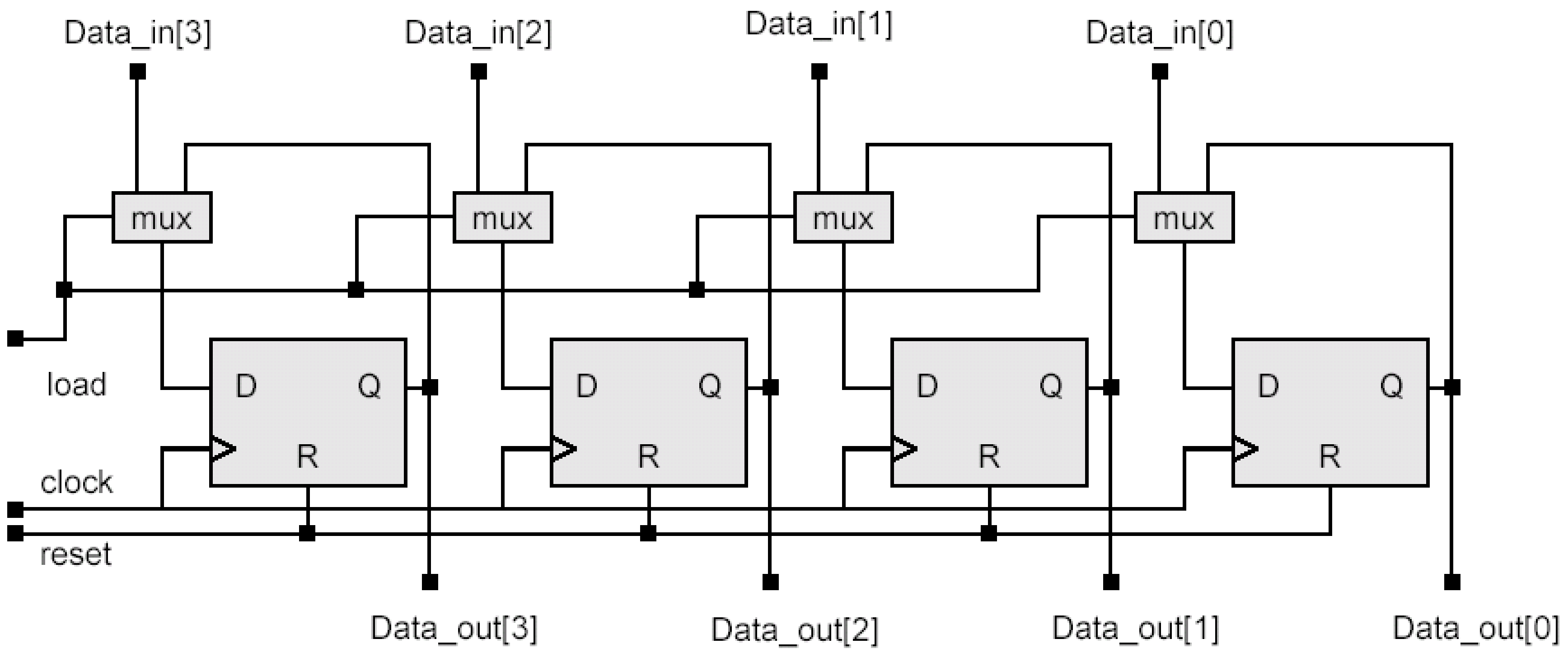
```
always @ (posedge clock or posedge reset)
  if (reset == 1'b1)
    Data_out = 4'b0;
  else if (load == 1'b1)
    Data_out = Data_in;
```

Parallel Load Register (2/2)

```
always @ (posedge clock or posedge reset)
  if (reset == 1'b1)
      Data_out <= 4'b0;
  else if (load == 1'b1) begin
      Data_out[3] <= Data_in[3];        Data_out[2] <= Data_in[2];
      Data_out[1] <= Data_in[1];        Data_out[0] <= Data_in[0];
  end
```

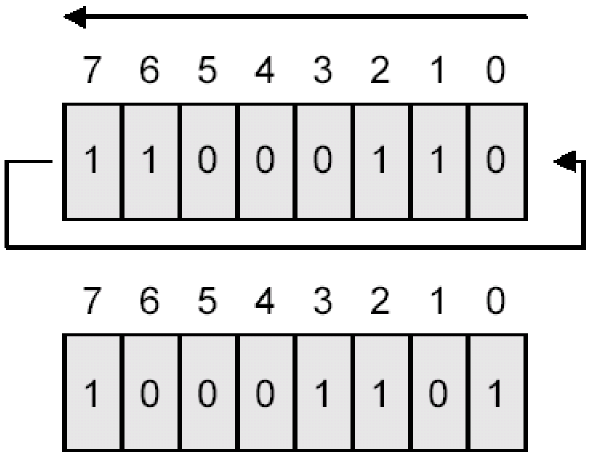
```
always @ (posedge clock or posedge reset)
  if (reset == 1'b1)
      Data_out = 4'b0;
  else if (load == 1'b1) begin // in the order of row by row
      Data_out[3] = Data_in[3];         Data_out[2] = Data_in[2];
      Data_out[1] = Data_in[1];         Data_out[0] = Data_in[0];
  end
```

Synthesis Result

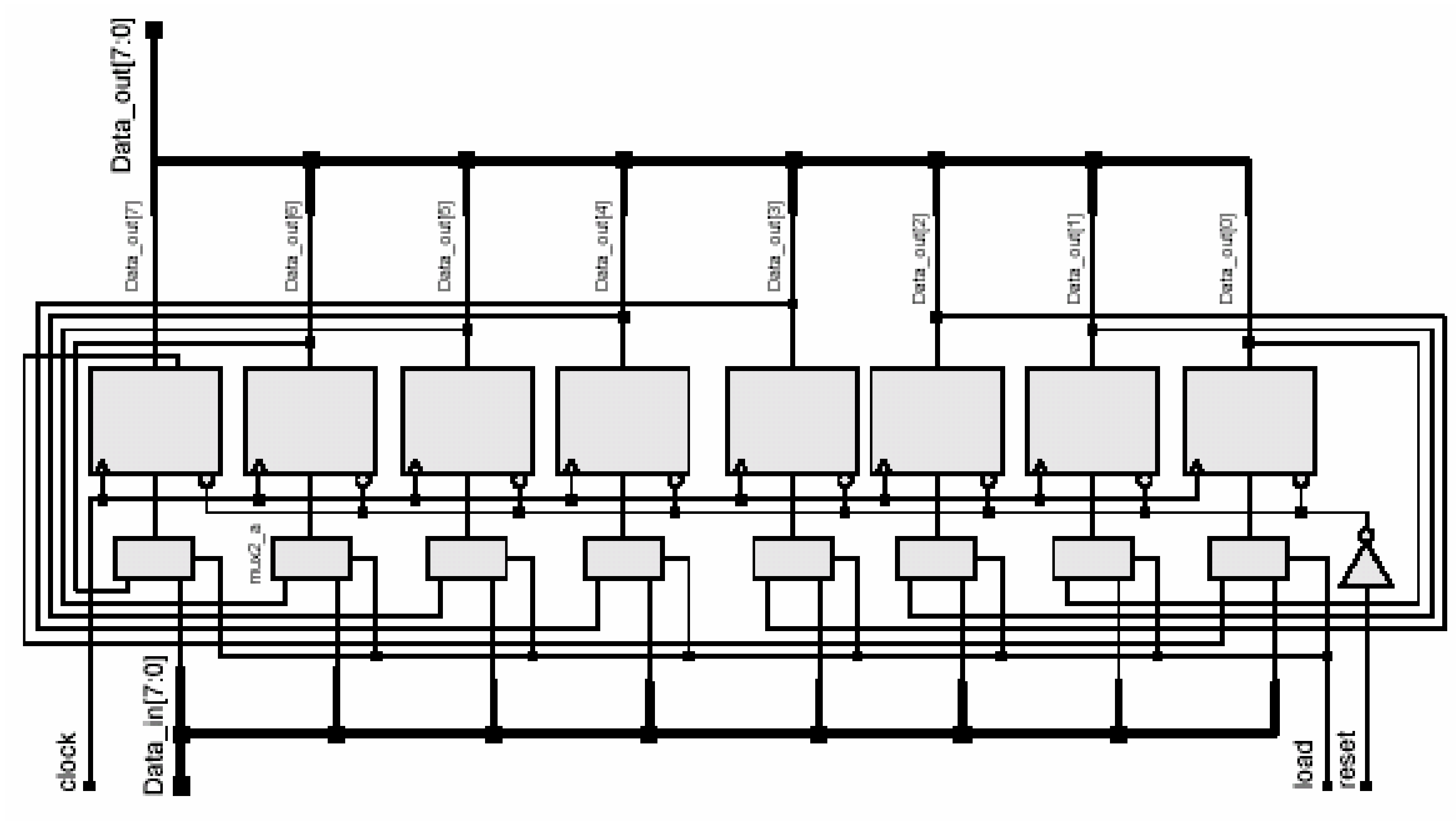


Barrel Shifter

```
module barrel_shifter (Data_in, load, clock, reset, Data_out);
    input         [7:0]    Data_in;
    input                  load, clock, reset;
    output        [7:0]    Data_out;
    reg           [7:0]    Data_out;

    always @ (posedge clock or posedge reset)
      if (reset == 1'b1)
          Data_out <= 8'b0;
      else if (load == 1'b1)
          Data_out <= Data_in;
      else
          Data_out <= {Data_out[6:0], Data_out[7]};
endmodule
```



Synthesis Result



Under some specific synthesis tool and technology

Ripple Counter (1/2)

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

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

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

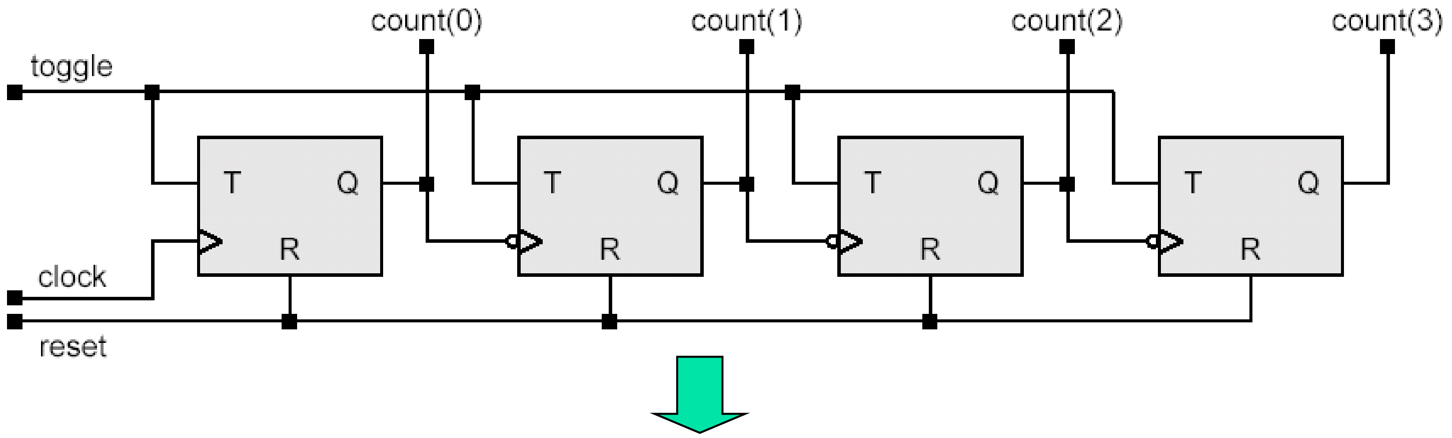
```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
0000
…
```

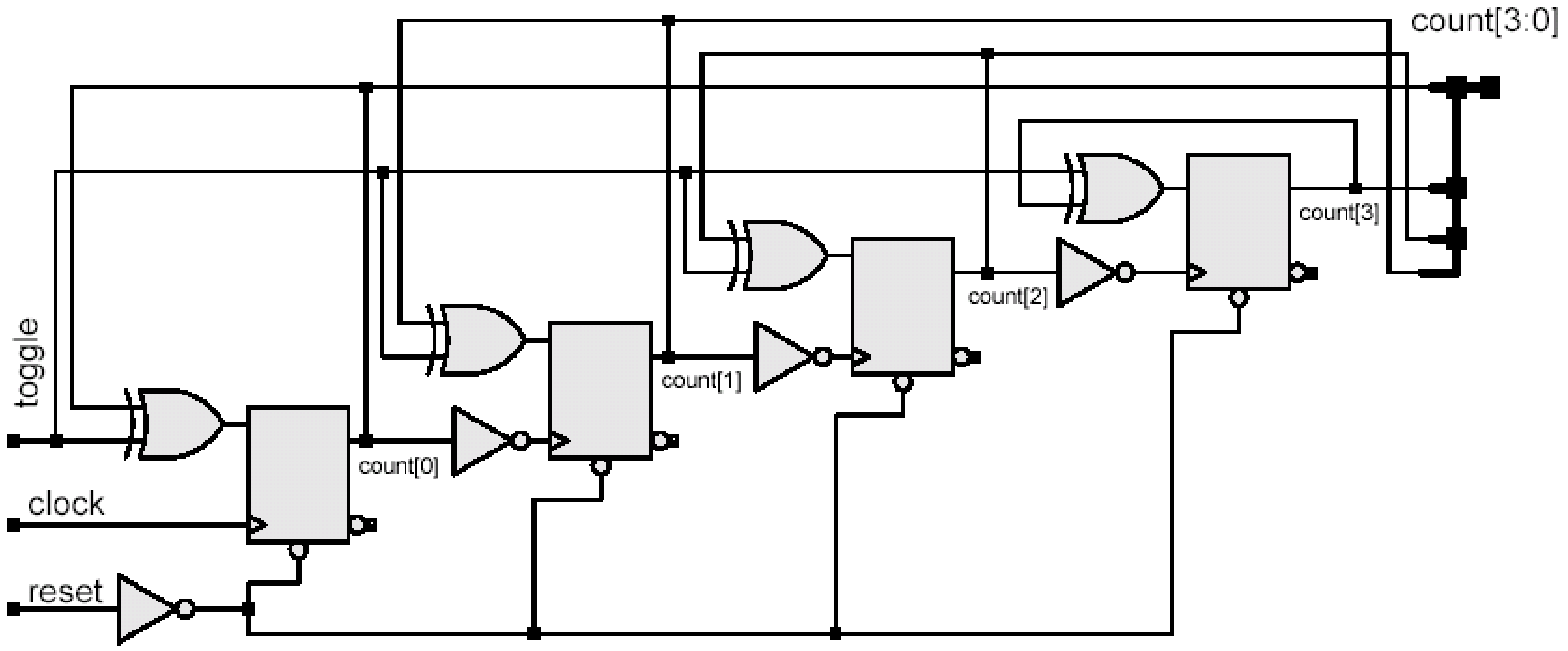
Ripple Counter (2/2)

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

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

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

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
0000
…
```

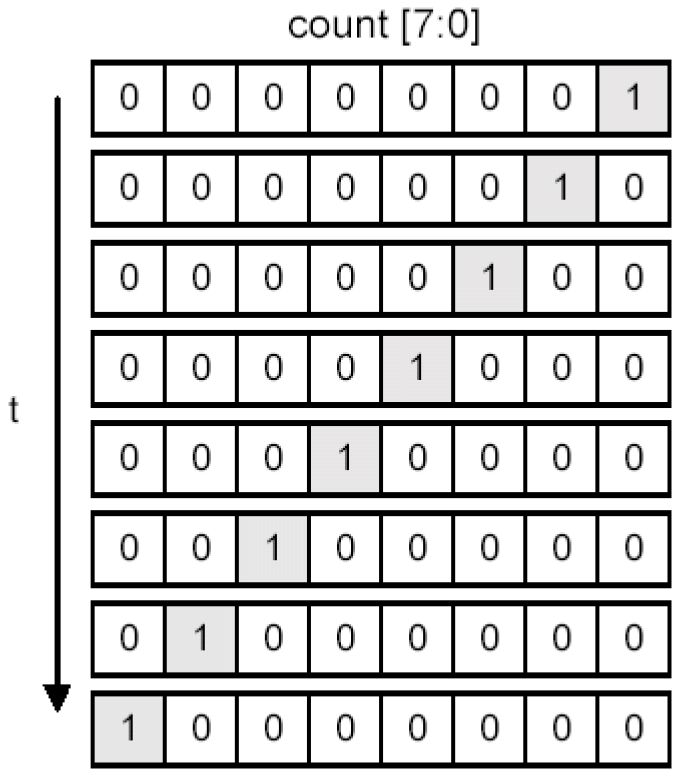
Synthesis Result





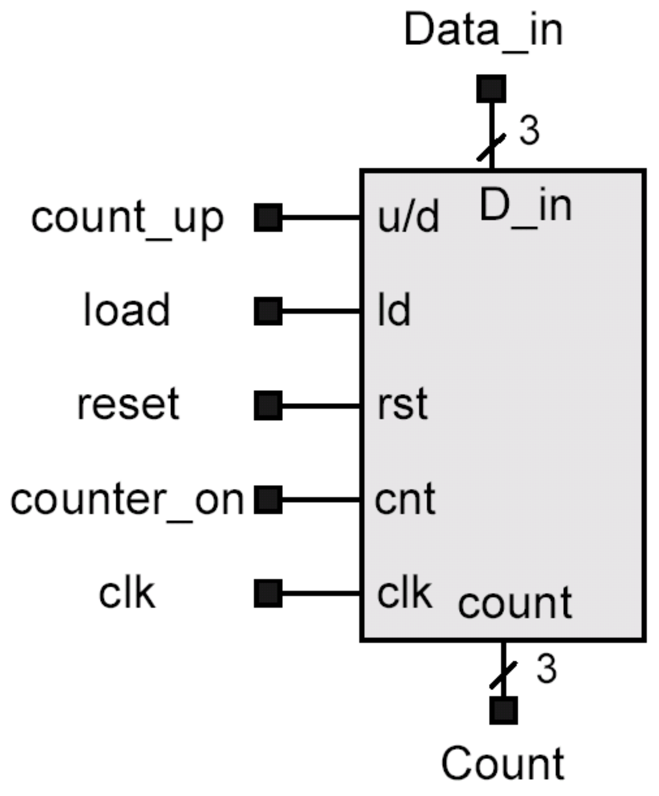
Ring Counter

```
module ring_counter (enable, clock, reset, count);
   input              enable, clock, reset;
   output    [7:0]    count;
   reg       [7:0]    count;

   always @ (posedge clock or
                posedge reset)
     if (reset == 1'b1)
          count <= 8'b0000_0001;
     else if (enable == 1'b1)
          count <= {count[6:0], count[7]};
endmodule
```

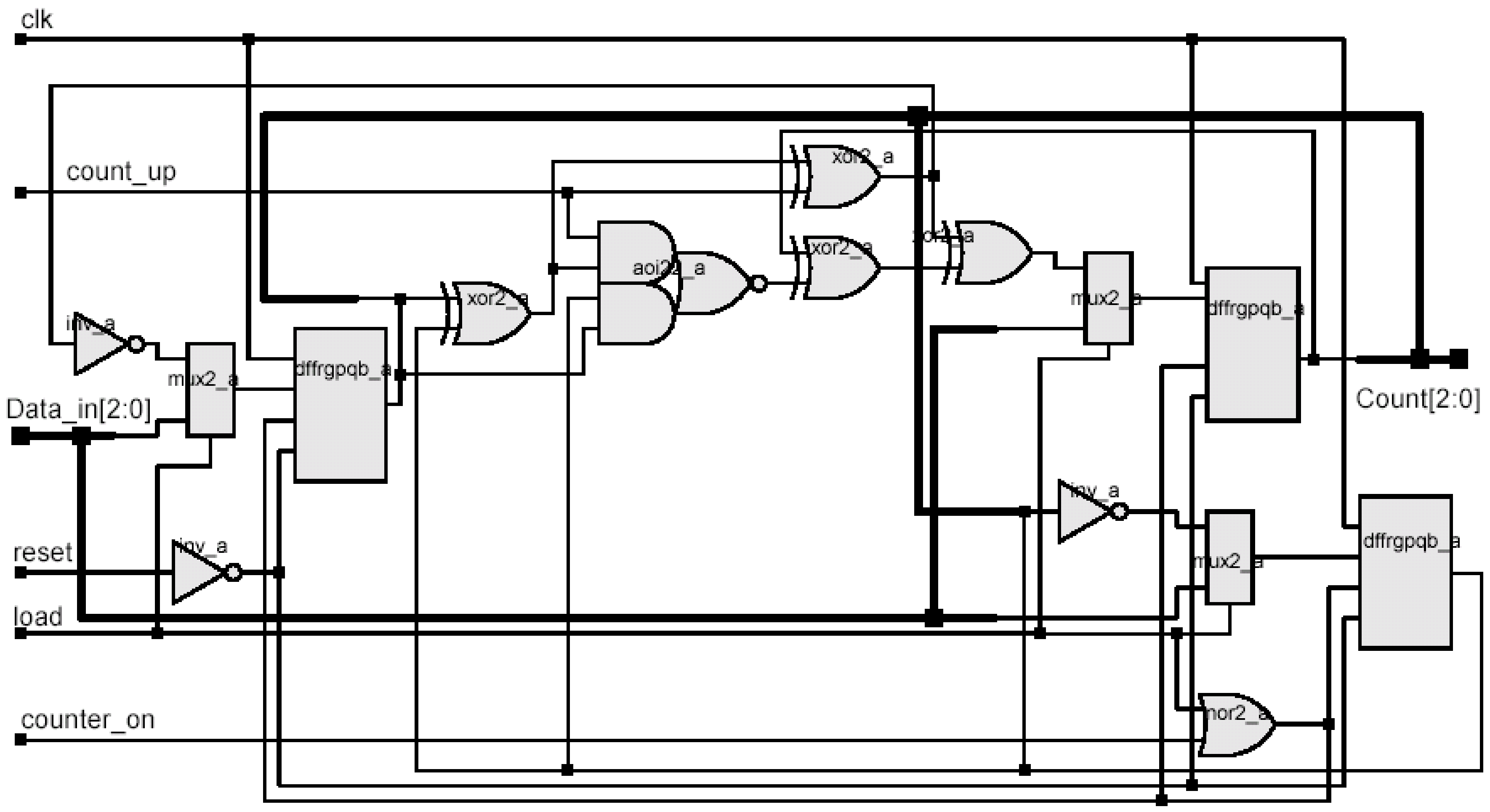


Up-Down Counter

```
module up_down_counter (load, count_up, counter_on, clk, reset, Data_in, Count);
    input                load, count_up, counter_on, clk, reset;
    input         [2:0]  Data_in;
    output        [2:0]  Count;
    reg           [2:0]  Count;

    always @ (posedge clk or posedge reset)
      if (reset == 1'b1)        Count = 3'b0;
      else if (load == 1'b1)    Count = Data_in;
      else if (counter_on == 1'b1) begin
            if (count_up == 1'b1)
              Count = Count + 1;
            else
              Count = Count - 1;
      end
endmodule
```



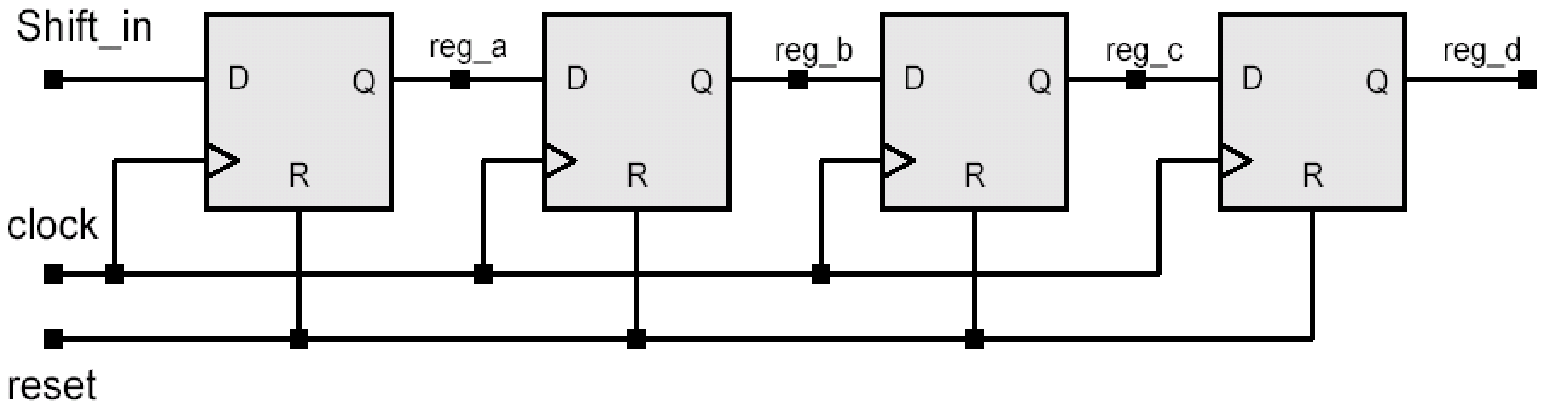
Synthesis Result



Under some specific synthesis tool and technology

Shift Register

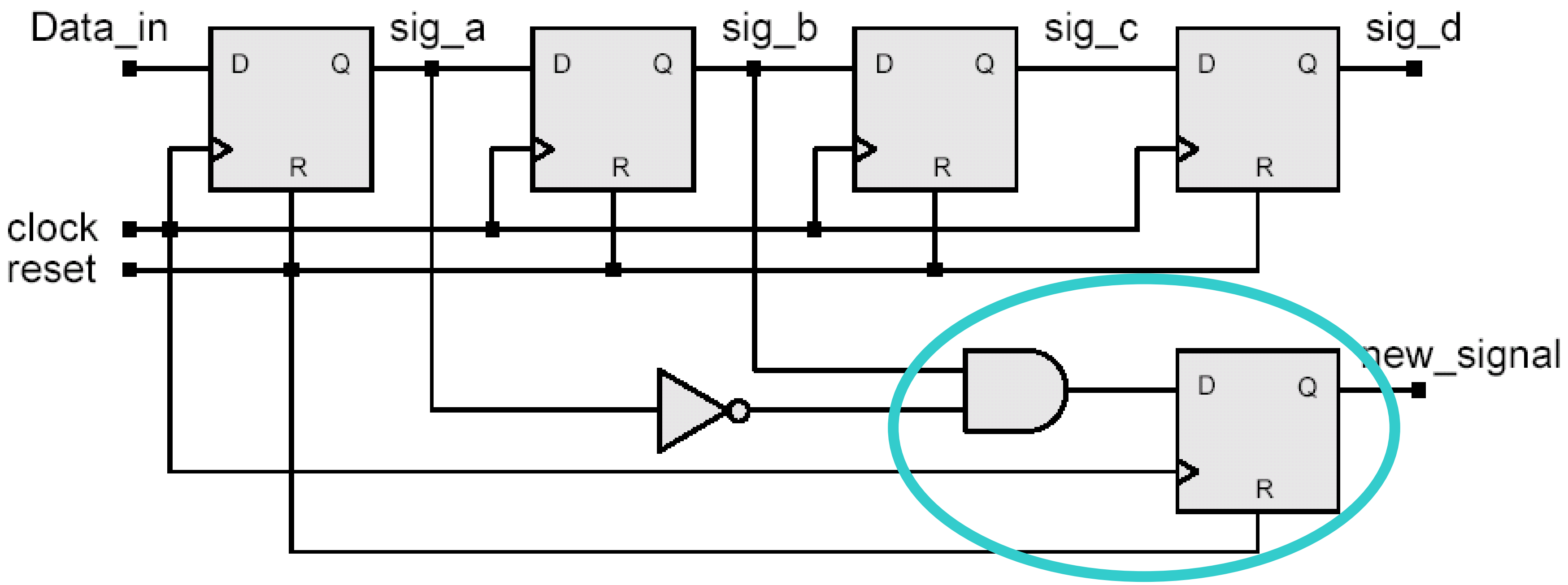
```
always @ (posedge clock)
    if (reset == 1'b1) begin
        reg_a <= 1'b0;
        reg_b <= 1'b0;
        reg_c <= 1'b0;
        reg_d <= 1'b0;
    end
    else begin
        reg_a <= Shift_in;
        reg_b <= reg_a;
        reg_c <= reg_b;
        reg_d <= reg_c;
    end
```



Shift Register with Registered Combinational Logic (1/4)

```
module shifter_1 (Data_in, clock, reset, sig_d, new_signal);
    input       Data_in, clock, reset;
    output      sig_d, new_signal;
    reg         sig_a, sig_b, sig_c, sig_d, new_signal;

    always @ (posedge clock or posedge reset) 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
```

Shift Register with Registered Combinational Logic (2/4)

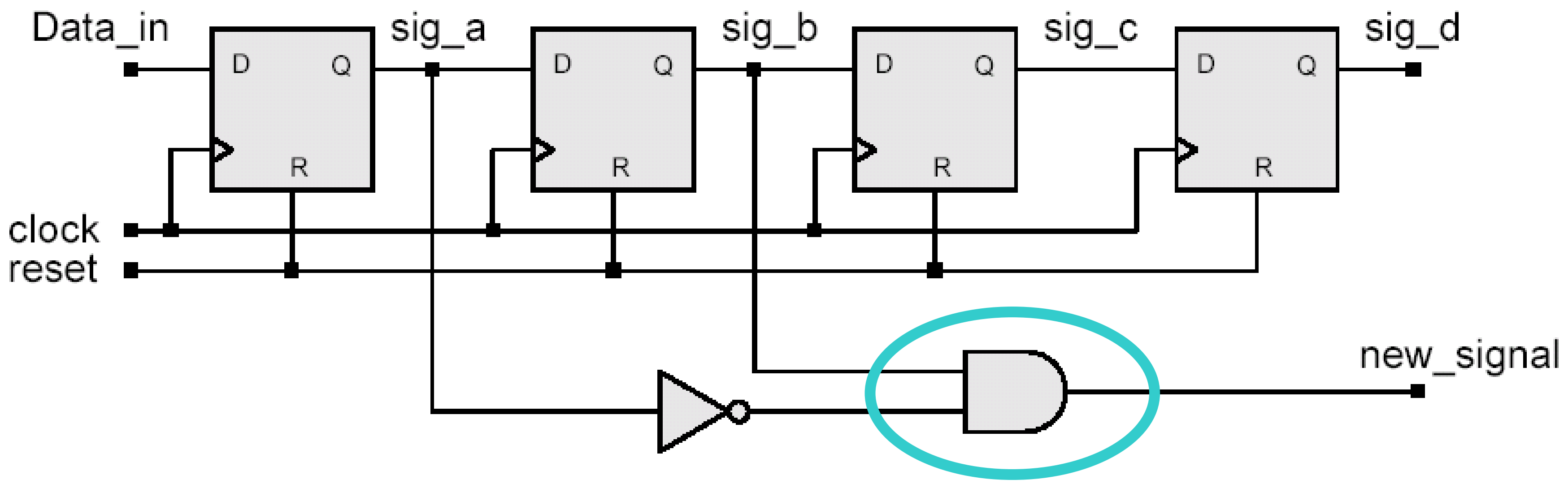


An output that is assigned new value within a synchronized behavior will be synthesized as the output of a flip-flop.

Shift Register with Registered Combinational Logic (3/4)

```
module shifter_1 (Data_in, clock, reset, sig_d, new_signal);
    input       Data_in, clock, reset;
    output      sig_d, new_signal;
    reg         sig_a, sig_b, sig_c, sig_d; wire new_signal;

    always @ (posedge clock or posedge reset) 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;
            end
          end
      assign new_signal <= (~ sig_a) & sig_b;
      endmodule
```

Shift Register with Registered Combinational Logic (4/4)

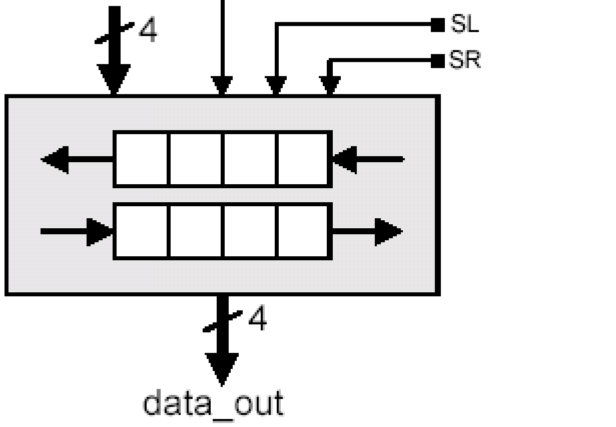


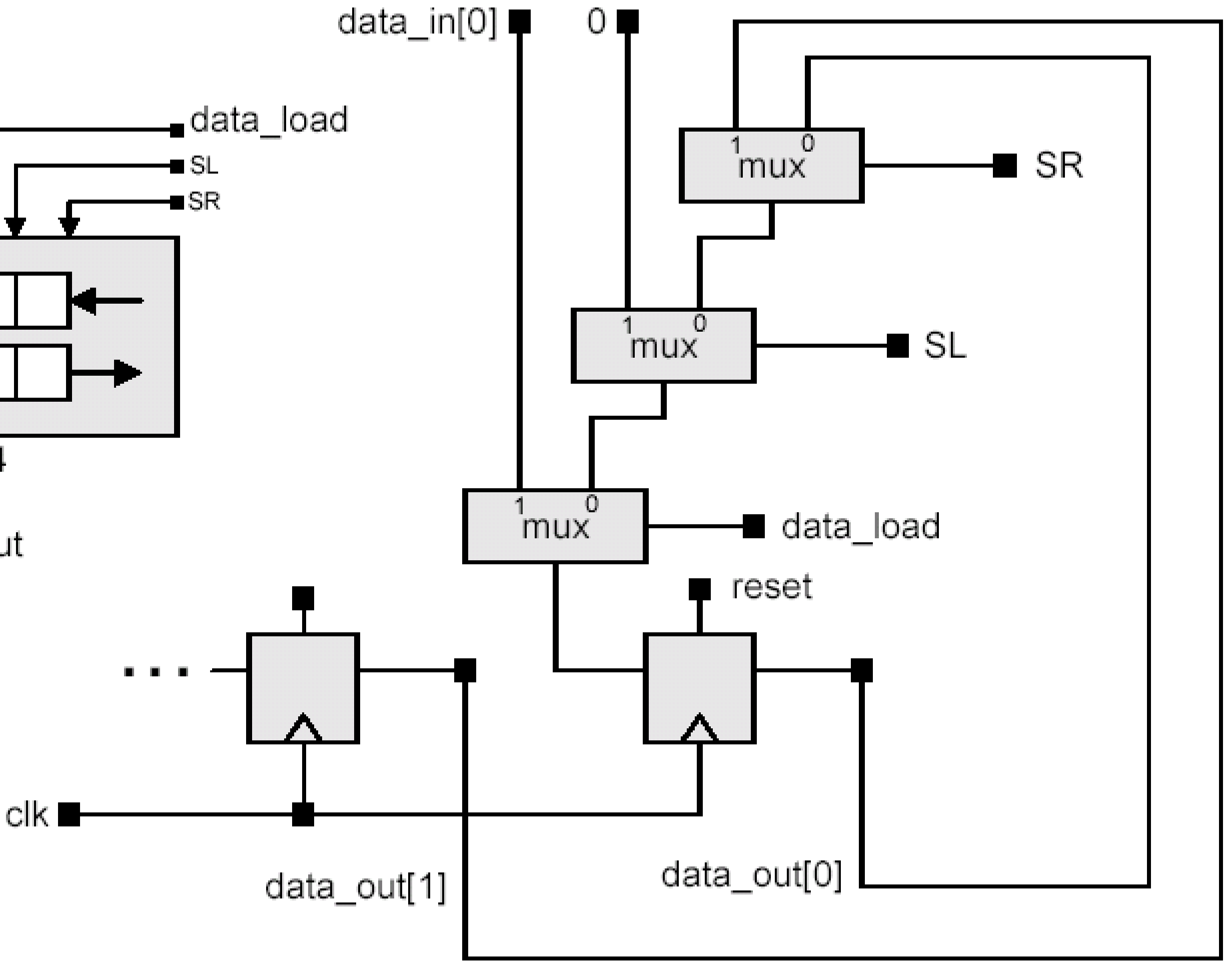
Signals that are assigned new values outside a behavior or in a behavior that does not include a synchronizing signal in its event control expression will be synthesized as combinational logic.

Data Shifter (1/2)

```
module data_shift ( data_out, data_in, clk, data_load, SL, SR);
 input     [3:0] data_in;
 input           clk, data_load, SL, SR;
 output    [3:0] data_out;

 reg       [3:0] data_out;

always @ (posedge clk)
 if (data_load) data_out = data_in;
 else if (SL) data_out = data_out << 1;
 else if (SR) data_out = data_out >> 1;
endmodule
```

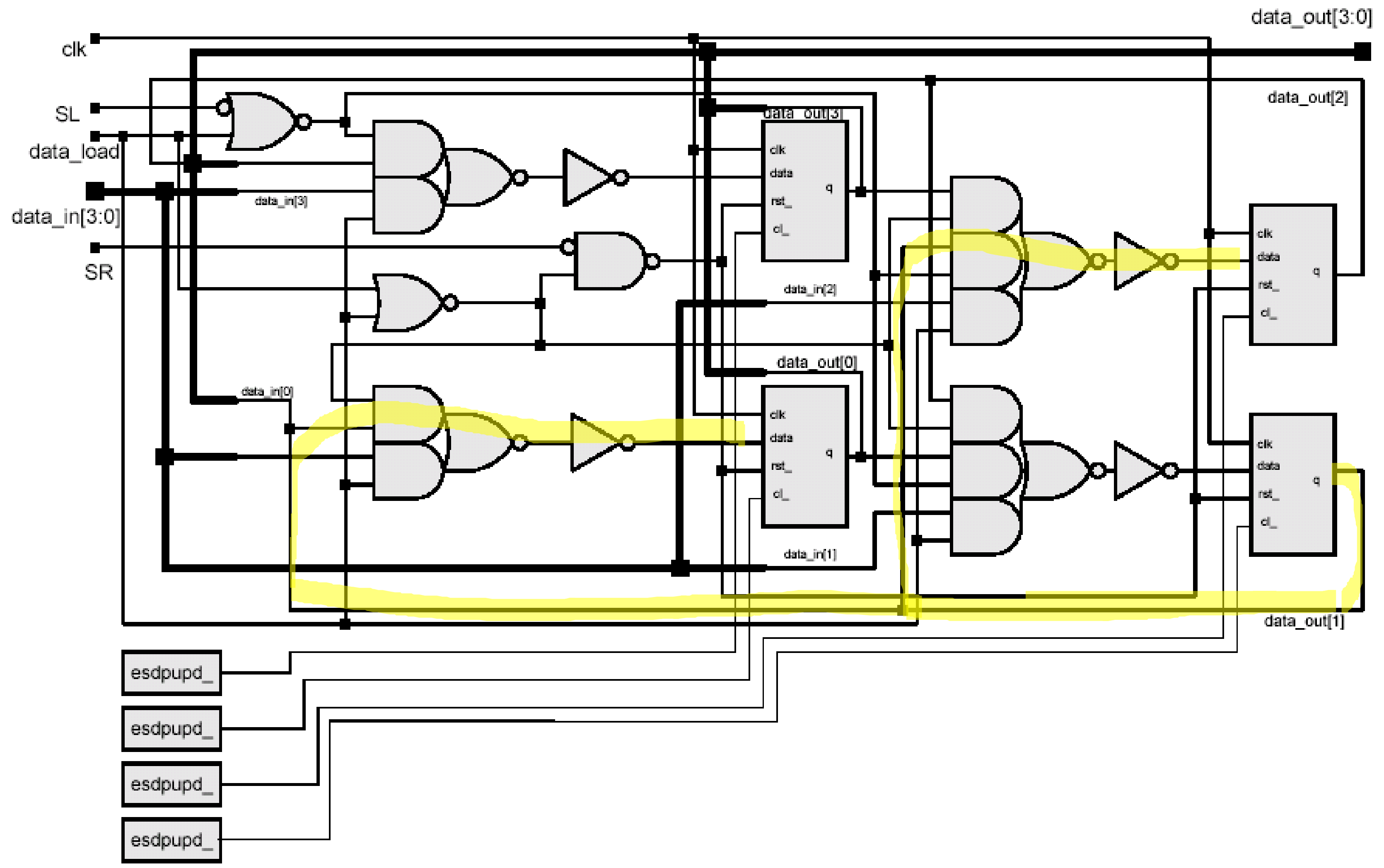
Data Shifter (2/2)





Data path for LSB

Synthesis Result



Accumulator 1

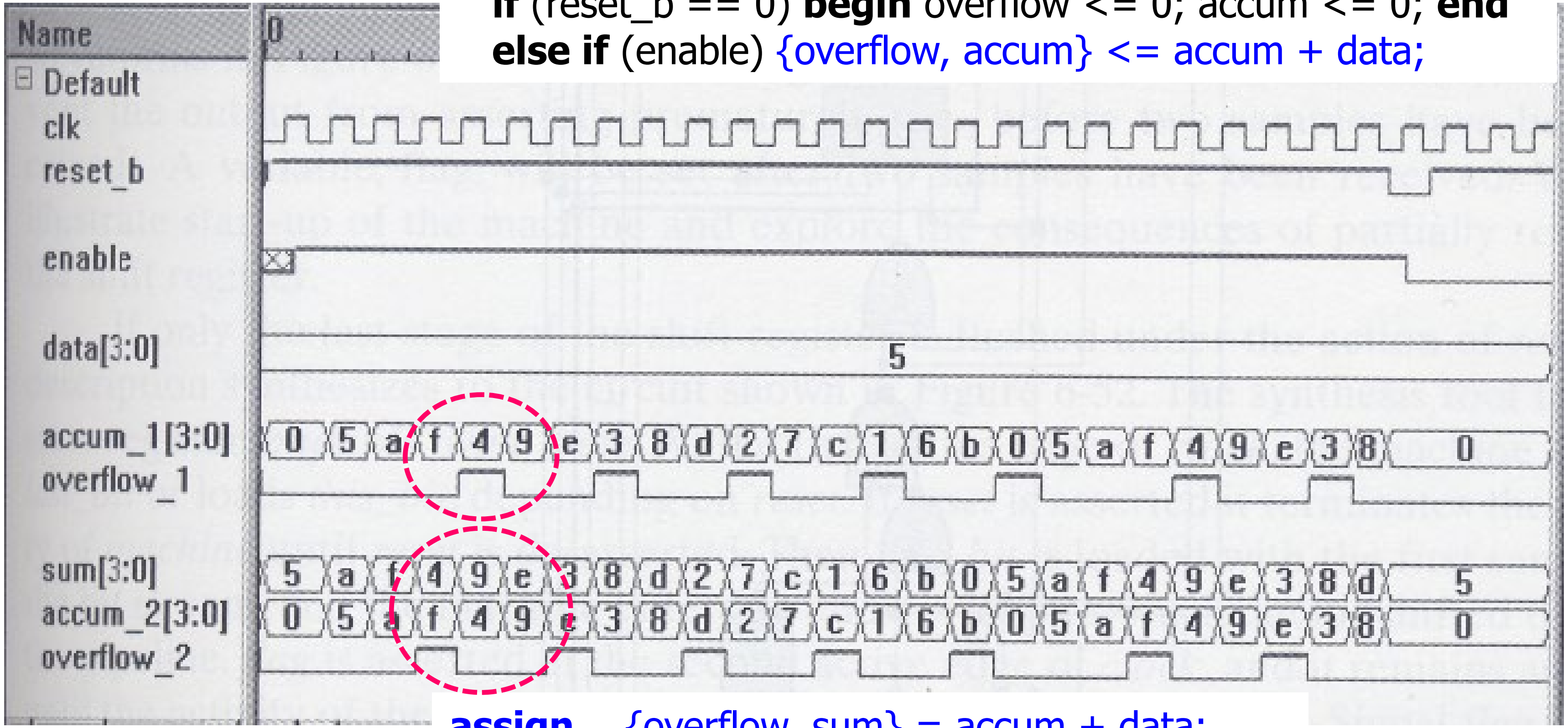
```
module Add_Accum_1 (data, enable, clk, reset_b, overflow, accum);
   input        [3:0]   data;
   input                enable, clk, reset_b;
   output               overflow;
   output       [3:0]   accum;
   reg                  overflow;
   reg          [3:0]   accum;

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

Accumulator 2

```
module Add_Accum_2 (data, enable, clk, reset_b, overflow, accum);
   input        [3:0]   data;
   input                enable, clk, reset_b;
   output               overflow;
   output       [3:0]   accum;
   reg          [3:0]   accum;
   wire                 overflow;
   wire         [3:0]   sum;
   assign       {overflow, sum} = accum + data;
   always @ (posedge clk or negedge reset_b)
      if (reset_b == 0) begin accum <= 0; end
      else if (enable) accum <=  sum;
endmodule
```

Simulation Result

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



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
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

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

Expression Substitution 1

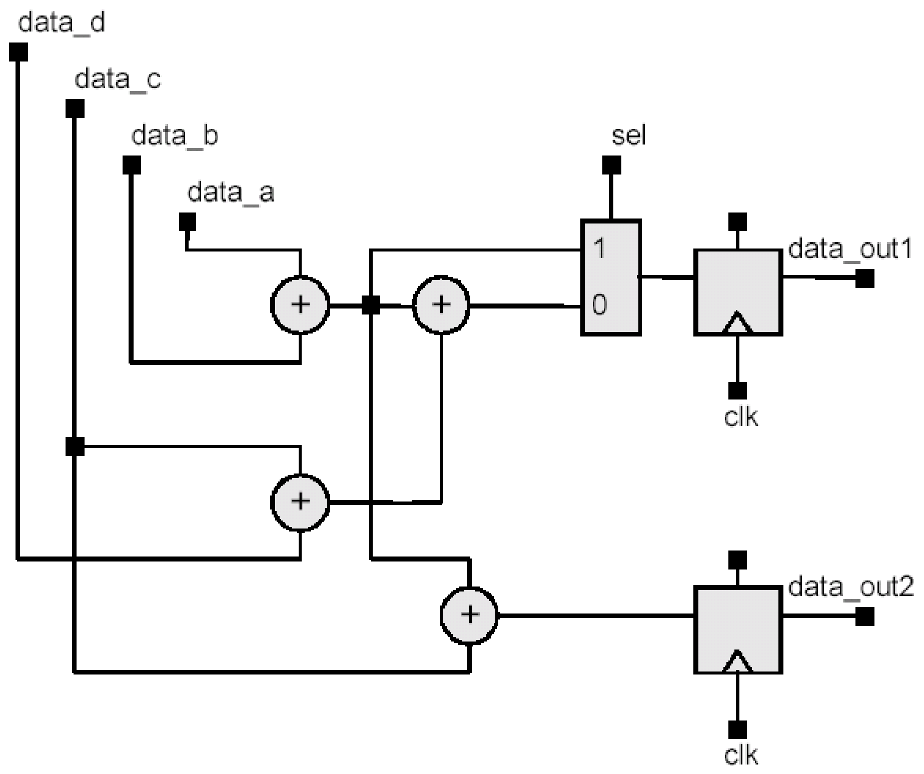
```
module  multiple_reg_assign
  (data_out1, data_out2, data_a, data_b, data_c, data_d, sel, clk);

  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
```

Expression Substitution 2

```
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                 clk, sel;
  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
```

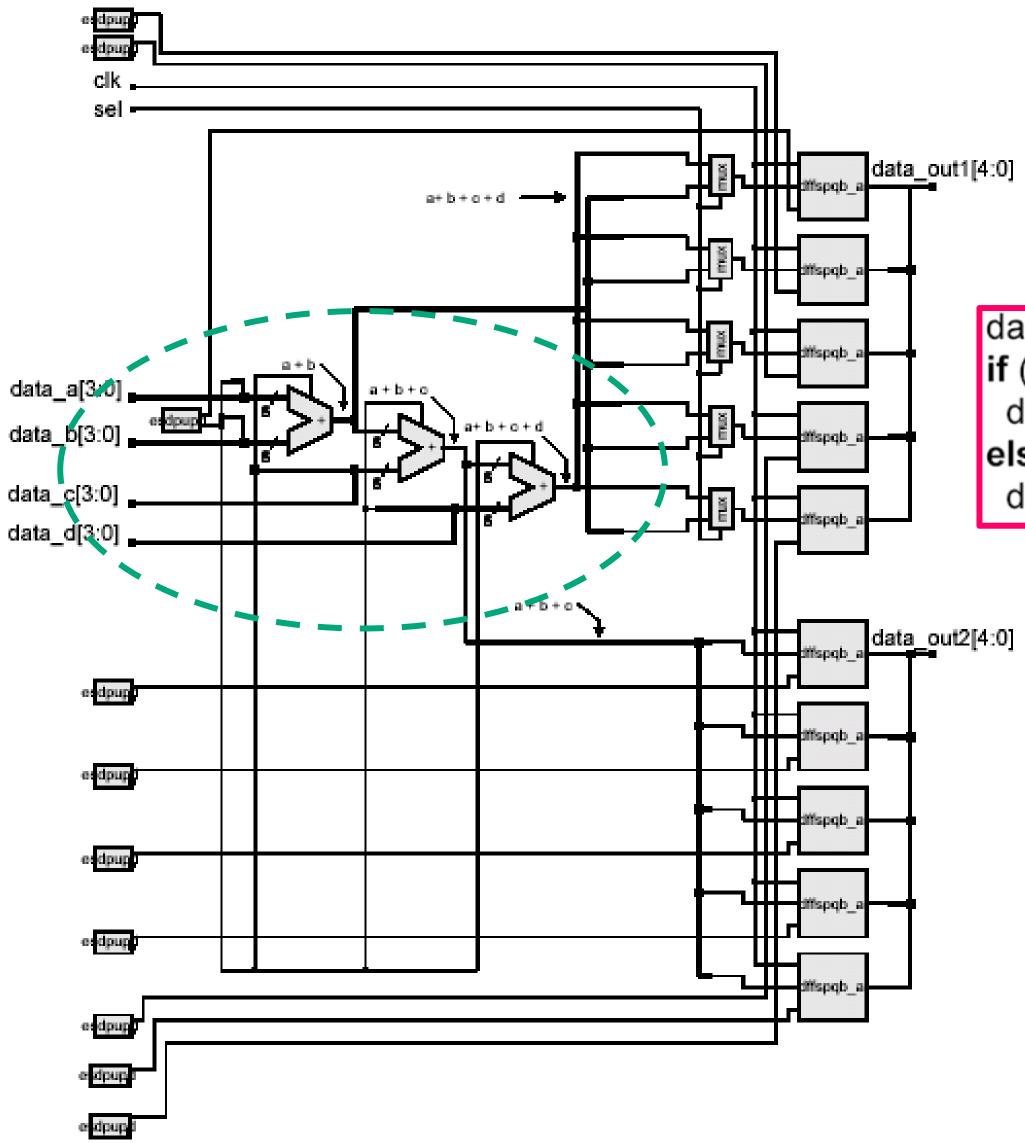
```
data_out1 = data_a + data_b ;
data_out2 = data_out1 + data_c;
if (sel == 1'b0)
  data_out1 = data_out2 + data_d;
```

Data Flow Graph



```
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;
```

Synthesis Result



```
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;
```

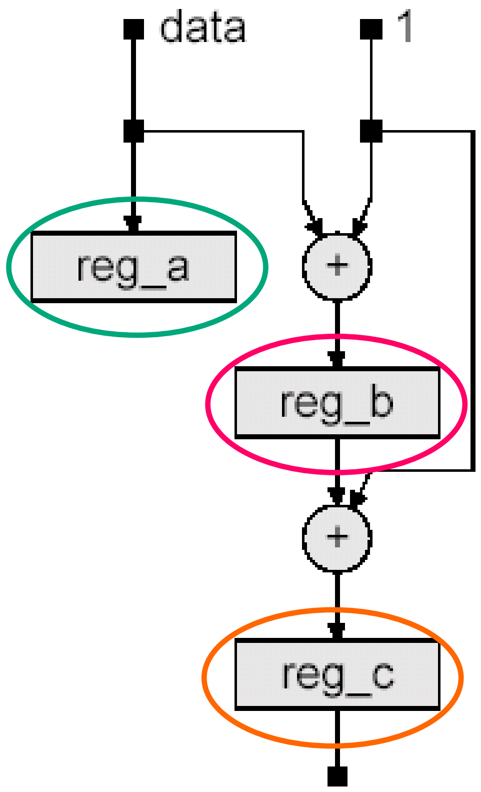
Note: The adder units will be optimized, depending on the performance constraints and the available resources (e.g. carry look-ahead vs ripple-carry).

Synthesis of Non-Blocking Assignments

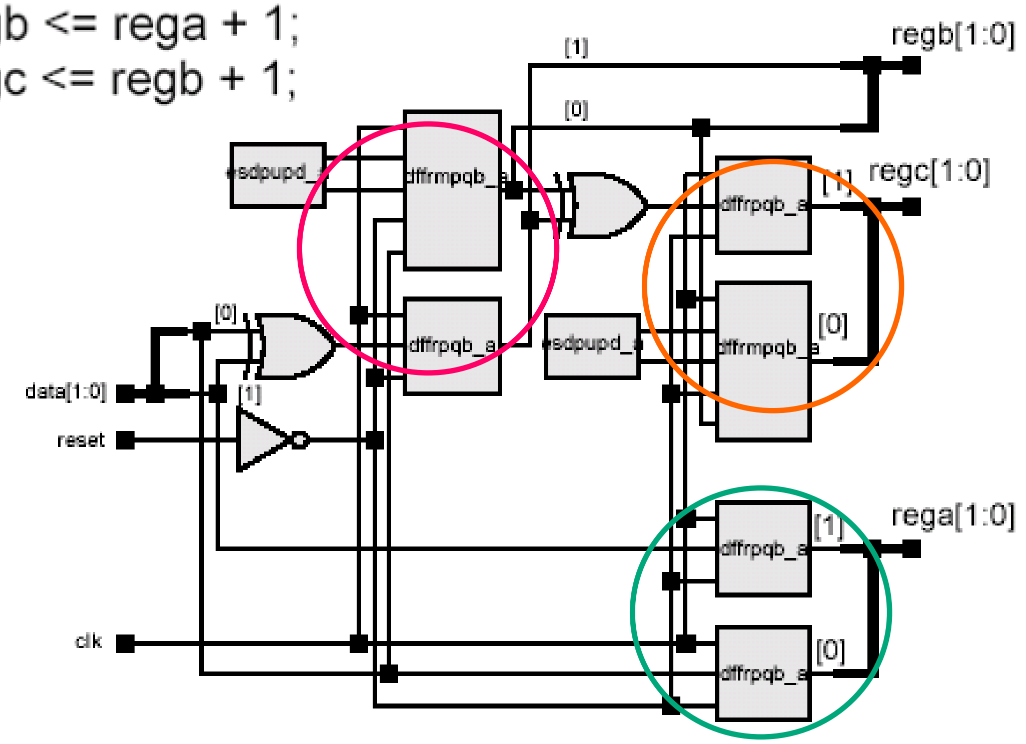
```
module pipe1 (data, rega, regb, regc, clk, reset);
 input          [1:0] data;
 input                clk, reset;
 output         [1:0] rega, regb, regc;
 reg            [1:0] rega, regb, regc;

 always @  (posedge clk or posedge reset)
  if (reset)
    begin
      rega = 2'b0;
      regb <= 2'b0;
      regc <= 2'b0;
    end
  else
    begin
     rega = data;
     regb <= rega + 1;
     regc <= regb + 1;
    end
endmodule
```

Data Flow Graph and Synthesis Result

Note: The data with non-blocking assignments implies a pipeline structure, and regc gets the value of regb from the previous cycle.



```
rega = data;
regb <= rega + 1;
regc <= regb + 1;
```

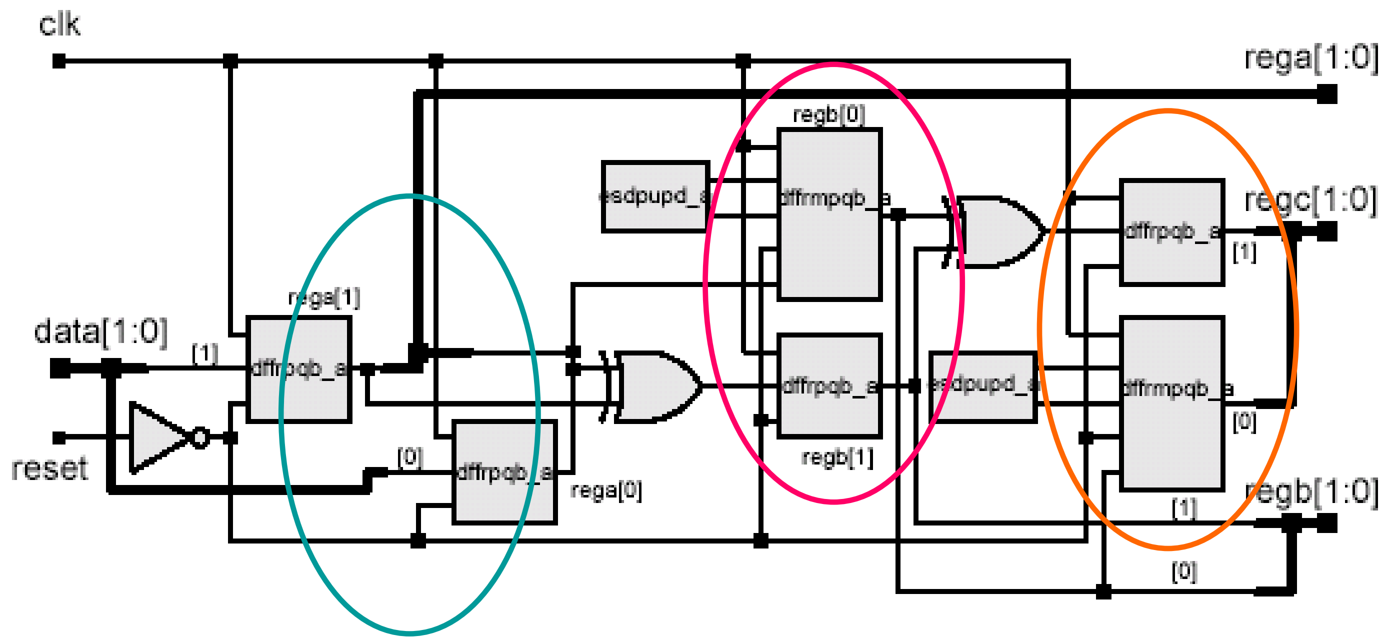


The activity associated with procedural assignments is processed before the activity associated with non-blocking assignments.

Synthesis of Non-Blocking Assignments

```
module pipe1_alt (data, rega, regb, regc, clk, reset);
 input          [1:0] data;
 input                clk, reset;
 output         [1:0] rega, regb, regc;
 reg            [1:0] rega, regb, regc;

always @  (posedge clk or posedge reset)
   if (reset) rega = 2'b0; else rega = data;

always @  (posedge clk or posedge
reset)
  if (reset)
     begin
       regb <= 2'b0;
       regc <= 2'b0;
     end
  else
     begin
       regb <= rega + 1;
       regc <= regb + 1;
     end
endmodule
```

Synthesis Result



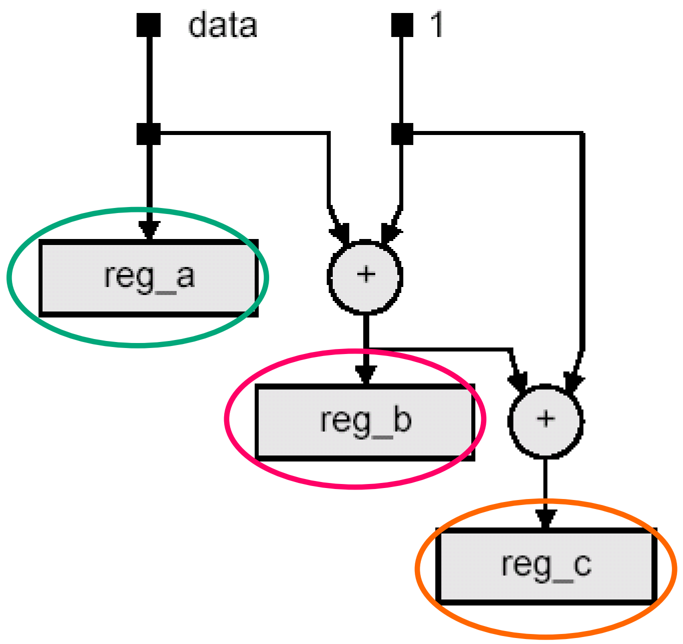
always @ (**posedge** clk **or posedge** reset)
if (reset) rega = 2'b0; **else** rega = data;

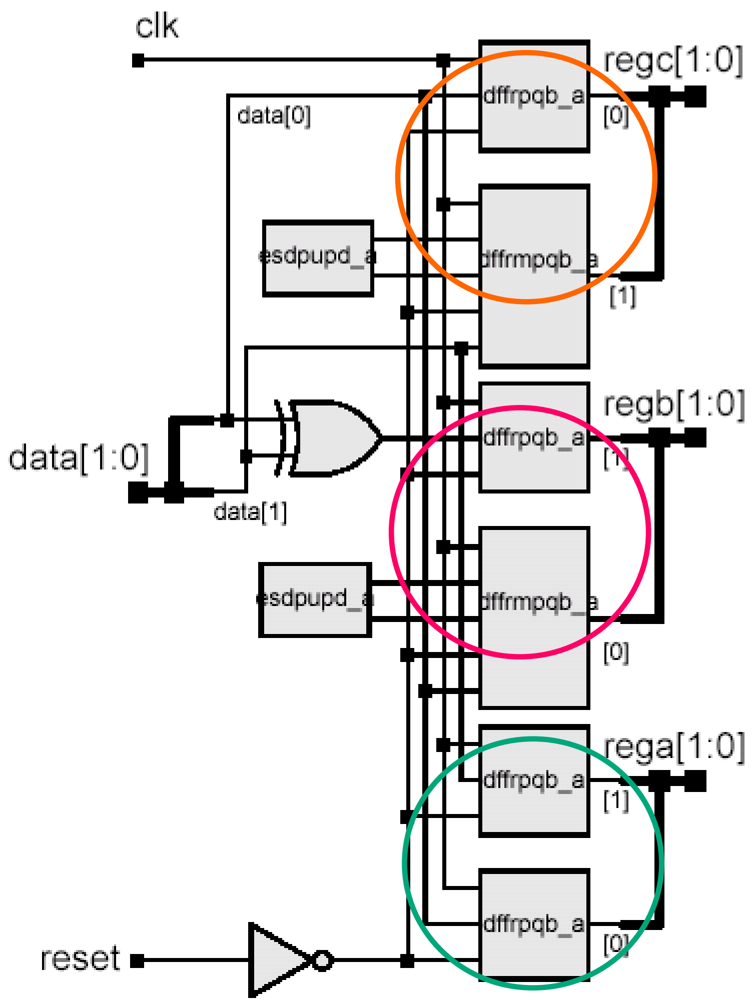
regb <= rega + 1;
regc <= regb + 1;

Synthesis of Blocking Assignments

```
module pipe2 (data, rega, regb, regc, clk, reset);
 input          [3:0] data;
 input                clk, reset;
 output   [3:0] rega, regb, regc;
 reg            [3:0] rega, regb, regc;

 always @  (posedge clk or posedge reset)
  if (reset)
    begin
      rega = 4'b0;
      regb = 4'b0;
      regc = 4'b0;
    end
  else
    begin
      rega = data;          // = data;
      regb = rega + 1;      // = data + 1;
      regc = regb + 1;      // = data + 1 + 1;
    end
endmodule
```

Data Flow Graph and Synthesis Result

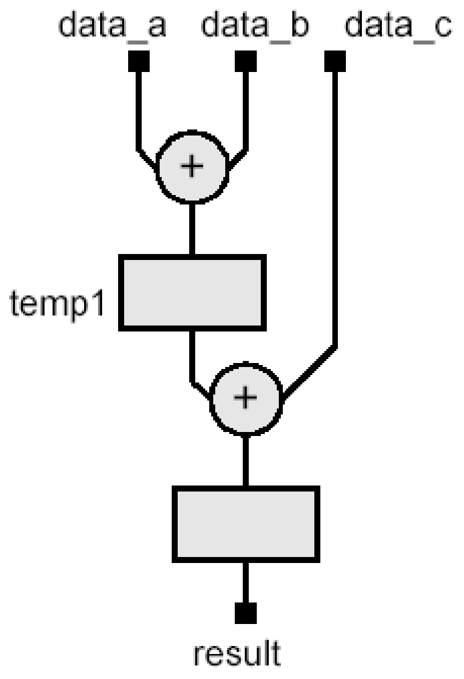




Note: regc gets the new value of regb in the same clock cycle.

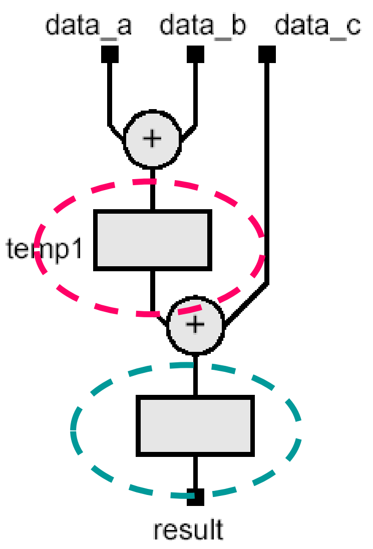
Synthesis of Multi-Cycle Operations

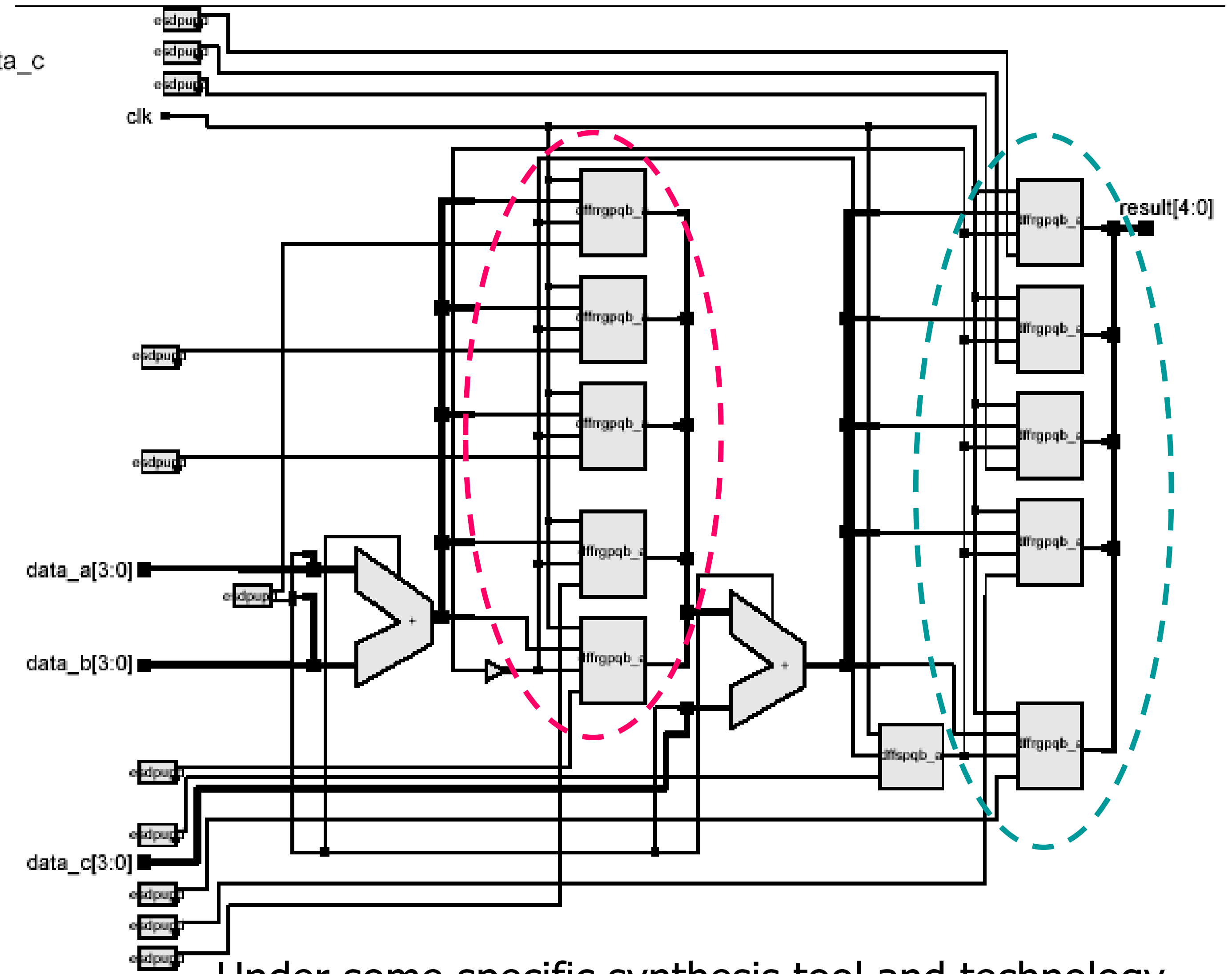
```
module m_cycle (result, data_a, data_b, data_c, clk);
  input     [3:0] data_a, data_b, data_c;
  input           clk;
  output    [4:0] result;
  reg       [4:0] result, temp1;

  always @  ( posedge clk)
   begin
    temp1 = data_a + data_b;

    @  (posedge clk)
     result = temp1 + data_c;
   end
endmodule
```



An operation that cannot complete in one cycle must be distributed over multiple clock cycles, i.e. pipelined,

Synthesis Result

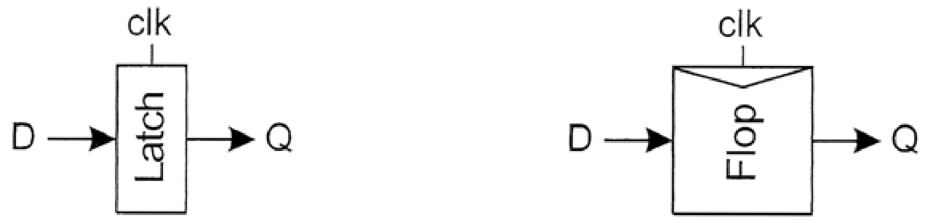


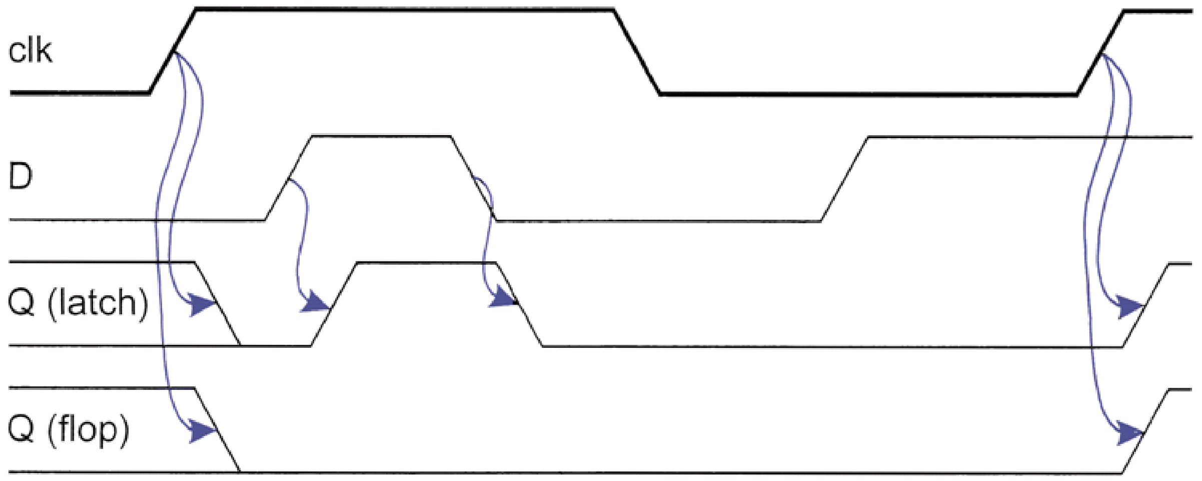


Under some specific synthesis tool and technology

Storage Elements

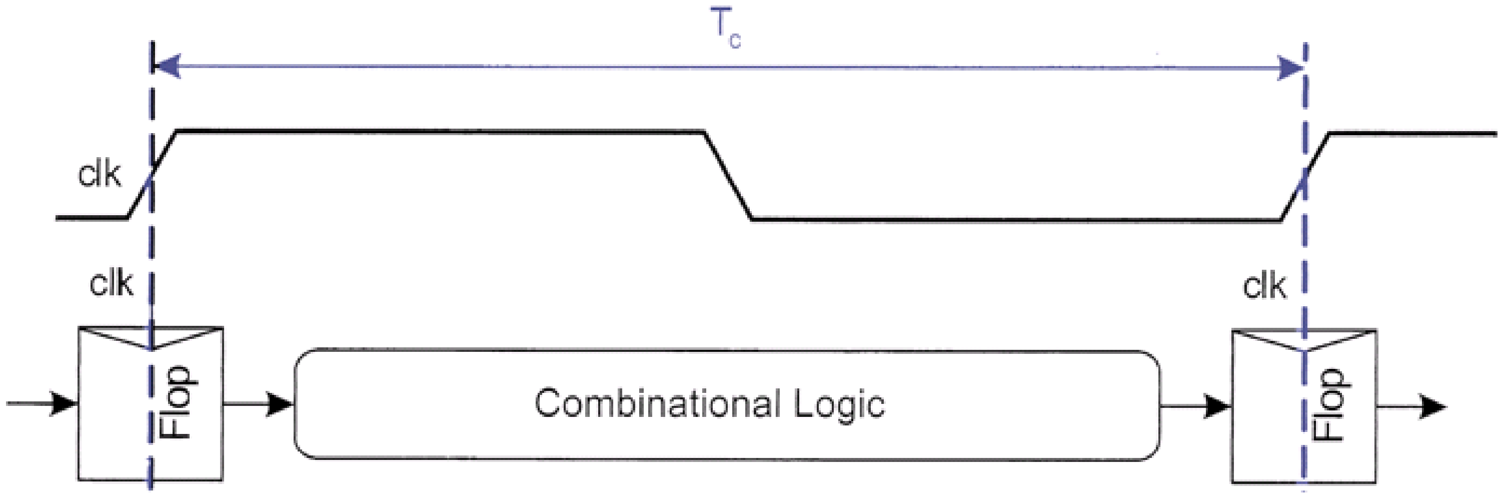
- Latch: level-sensitive
- Flip-flop: edge-triggered
  - D-type, T-type, SR-type, and JK-type.





Flip-Flops

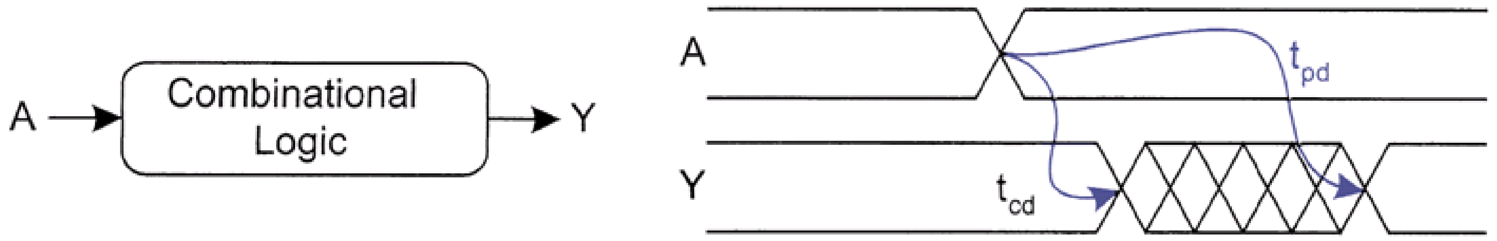
Clock Period

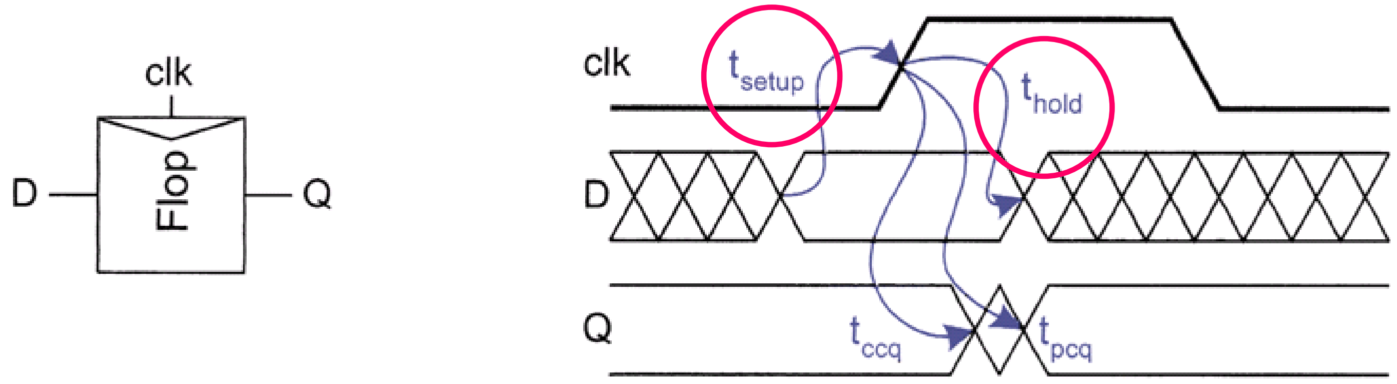


Timing Notation

- t_{pd}: logic propagation delay
- t_{cd}: logic contamination delay
- t_{pcq}: flop clock-to-Q propagation delay
- t_{ccq}: flop clock-to-Q contamination delay
- t_{setup}: flop setup time
- t_{hold}: flop hold time

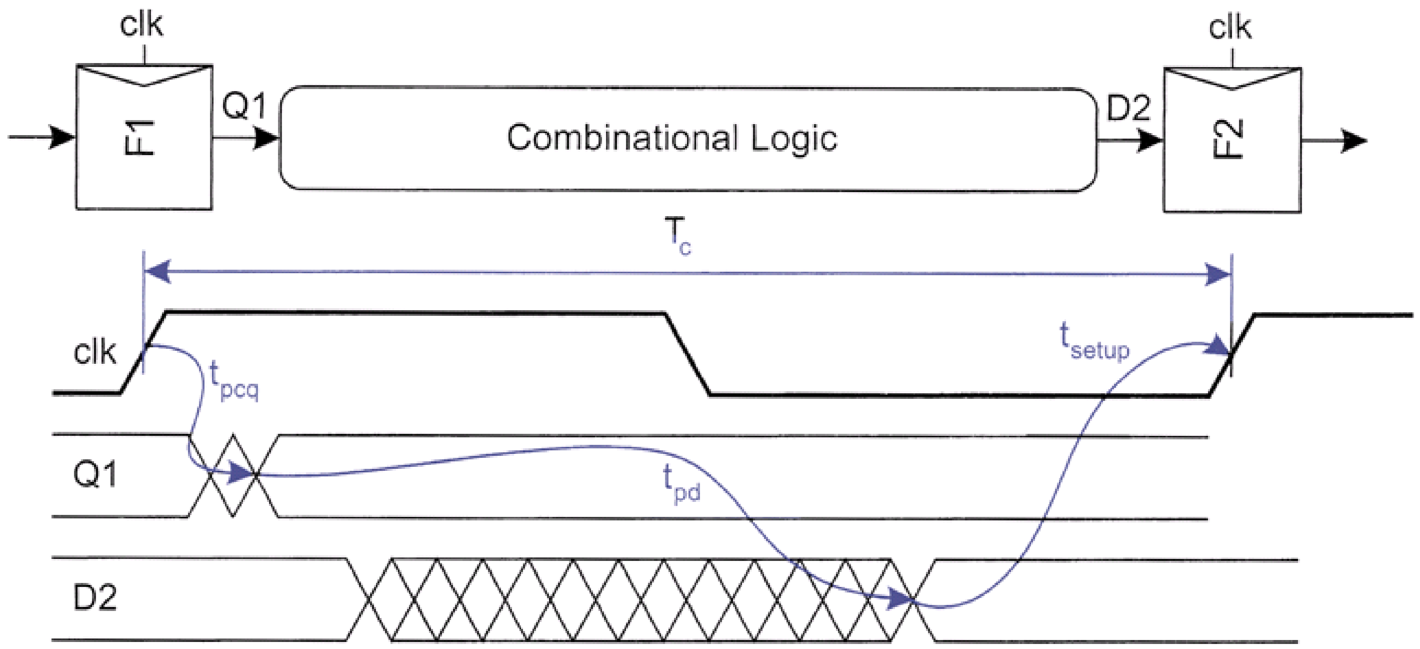
Timing Diagrams





Max-Delay Constraint of Flip-Flops (1/2)

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}$$

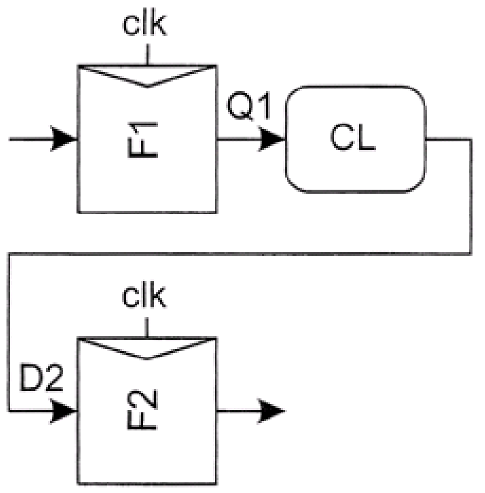


Max-Delay Constraint of Flip-Flops (2/2)

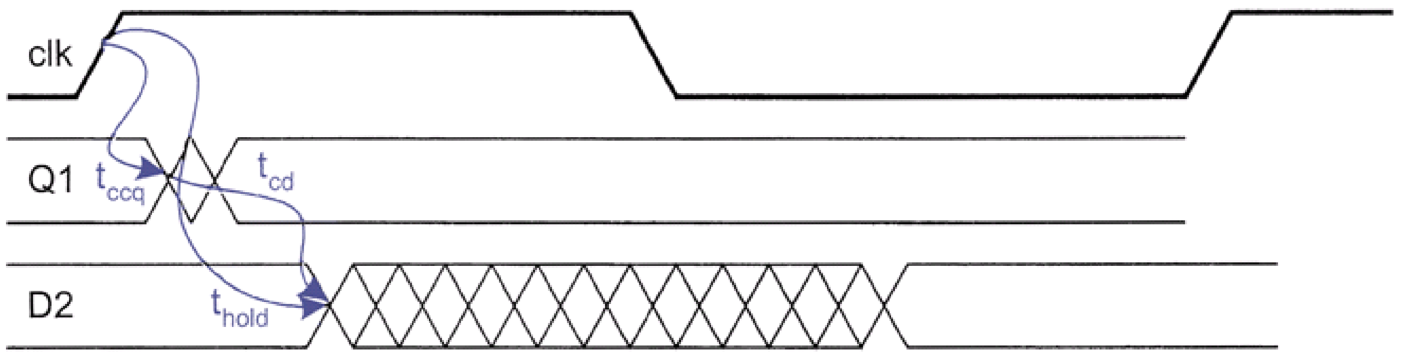
- Setup time failure (max-delay failure)
  - If the combinational logic delay is too long, the receiving element will miss its setup time and sample the wrong value.
- Solution
  - Redesign the logic to be faster.
  - Increase the clock period.

$$T_c \geq t_{pcq} + t_{pd} + t_{\mathrm{setup}}$$

Min-Delay Constraint of Flip-Flops (1/2)



$$t_{ccq} + t_{cd} \geq t_{\text{hold}}$$



Min-Delay Constraint of Flip-Flops (2/2)

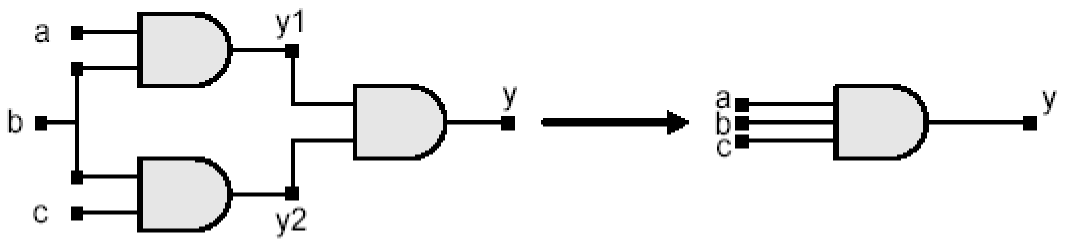
- Hold time failure (min-delay failure, race condition)
  - If the hold time is large and the contamination delay is small, data can incorrectly propagate through two successive elements on one clock edge, corrupting the system state.
- Solution
  - Redesign the logic to be slower.

$$t_{ccq} + t_{cd} \geq t_{\text{hold}}$$

Collapsing of Nets

An internal net declared in the source description may be eliminated by the optimization process.

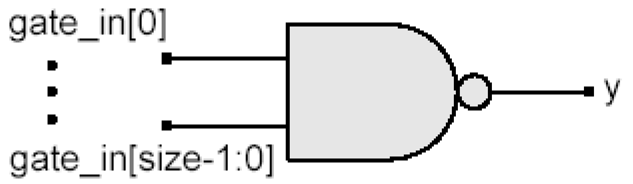
```
module and_gate(y, a, b, c);
  input       a, b, c;
  output      y;
  wire        y1, y2;

  assign y1 = a & b;
  assign y2 = c & b;
  assign y = y1 & y2;
endmodule
```



A net that is declared as a primary input or output will be retained in the synthesized design.

Elimination of Register Variables

A register variable declared in the source description may be eliminated by the optimization process.



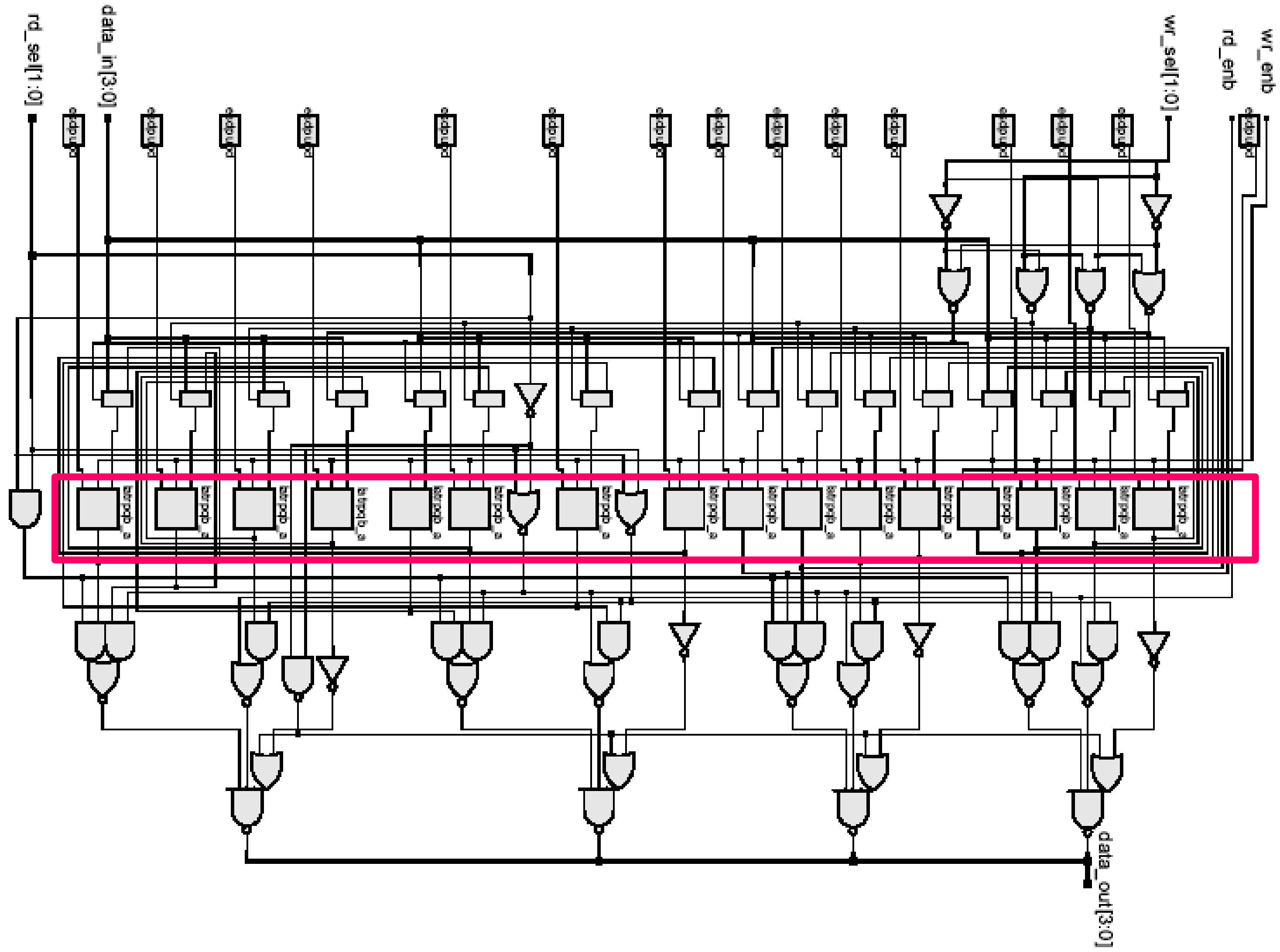
```
module n_gate1 (y, gate_in);
  parameter          size = 3;
  input [size –1: 0] gate_in;
  output             y;
  reg                y_int;
  integer            k;    // Loop control
  assign y = y_int;

always @ (gate_in)
   begin : look_for_0
    y_int = 0;
    for (k=0; k <= size-1; k=k+1)
      if (gate_in[k] == 0) begin
        y_int = 1;
        disable look_for_0;
      end
   end
endmodule
```

Note: Both k and y_int are eliminated.
The same result is obtained if *reg [size-1:0] k* is declared for the loop control.

Memory Synthesis

```
module sn54170 (data_in, wr_sel, rd_sel, wr_enb, rd_enb, data_out);
  input              wr_enb, rd_enb;
  input        [1:0] wr_sel, rd_sel;
  input        [3:0] data_in;
  output       [3:0] data_out;

  reg      [3:0] latched_data   [3:0];  // 4*4 bits

  always @ (wr_enb or wr_sel or data_in) begin
    if (!wr_enb) latched_data[wr_sel] = data_in;
  end                     // write

  assign data_out = (rd_enb) ? 4'b1111 : latched_data[rd_sel];
                                        // read
endmodule
```

Synthesis Result



// 4*4 bits

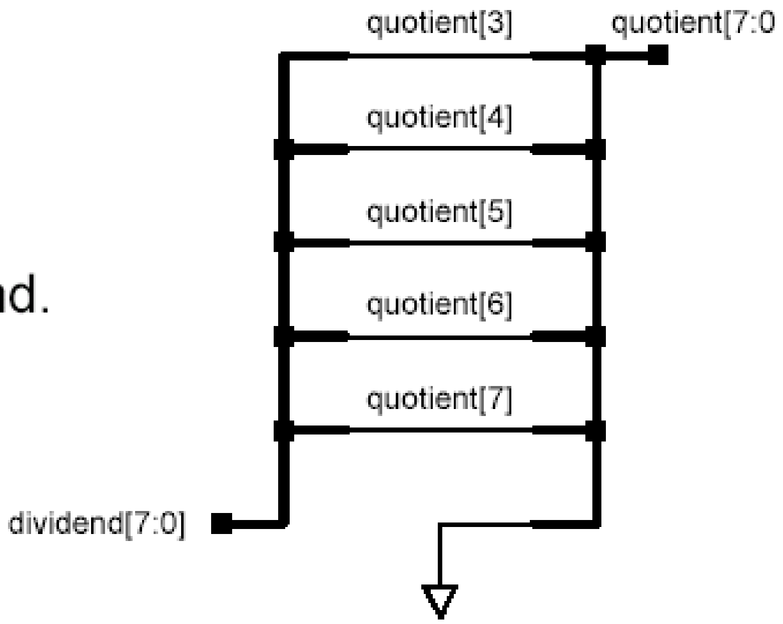
Synthesis of Arithmetic Operators (1/2)

assign quotient [7:0]=dividend [7:0] / 8;

<u>EXAMPLE</u>

1000 >> 3 = 0001

Zeros automatically fill from behind.



Division by a power of 2 will be recognized as a right-shift and synthesized as a re-wired bus, without gates.

Synthesis of Arithmetic Operators (2/2)

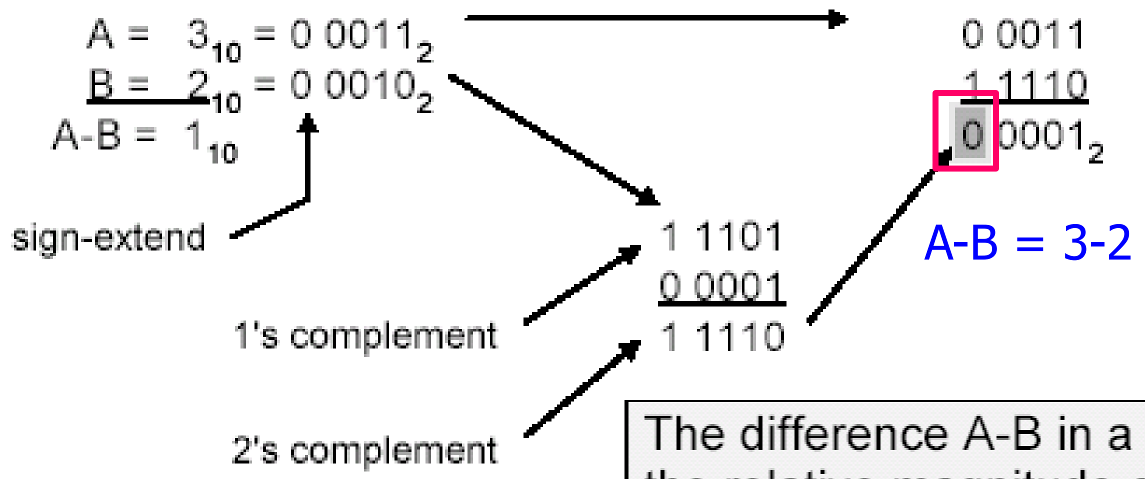
REMAINDER (%) WILL BE IMPLEMENTED AS A PART-SELECT IF THE DIVISOR IS A POWER OF 2.

EXAMPLE: $100\boxed{1}_2 \ \% \ 0010_2 = 0001_2$ All but the LSB are divisible by 2.

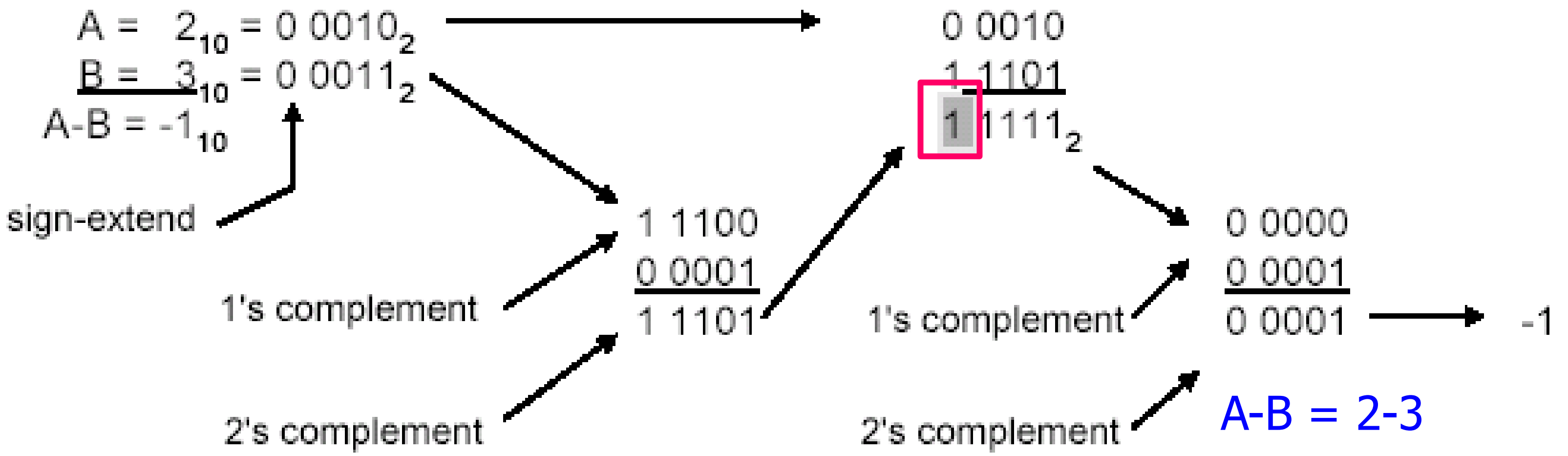
EXAMPLE: $10\boxed{10}_2 \ \% \ 0100_2 = 0010_2$ The bits to the left of 2^2 are divisible by 2^2.

PART-SELECT DOES NOT REQUIRE GATES.

Synthesis of Relational Operators (1/2)



The difference A-B in a sign-extended format reveals the relative magnitude of A and B in finite word length arithmetic. A > B if the extended bit is 0.

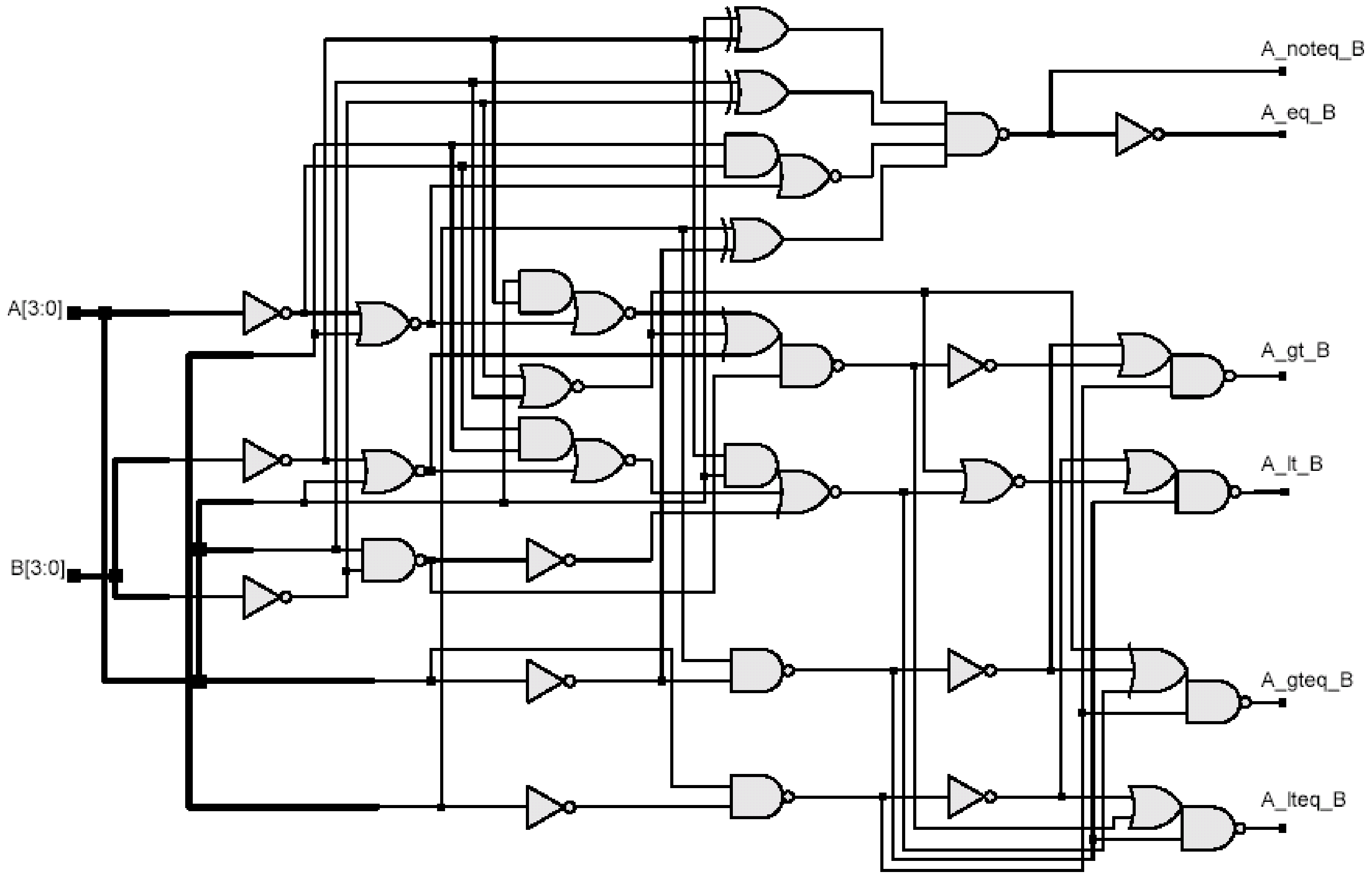


Synthesis of Relational Operators (2/2)

```
module relational_identity_4bit
  (A_eq_B, A_noteq_B, A_gt_B, A_gteq_B, A_lt_B, A_lteq_B, A, B);
  output        A_eq_B, A_noteq_B, A_gt_B, A_gteq_B, A_lt_B, A_lteq_B;
  input     [3:0] A, B;

  assign A_eq_B = (A == B);
  assign A_noteq_B = (A != B);
  assign A_gt_B = (A > B);
  assign A_gteq_B = (A >= B);
  assign A_lt_B = (A < B);
  assign A_lteq_B = (A <= B);

endmodule
```

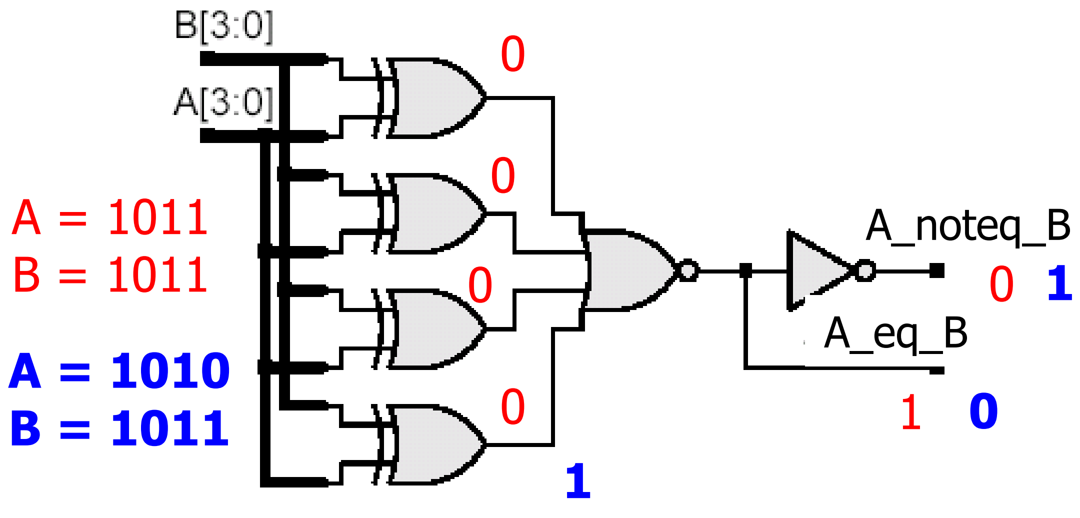
Note shared logic in synthesis result.

Synthesis Result



Synthesis of Equality Operators

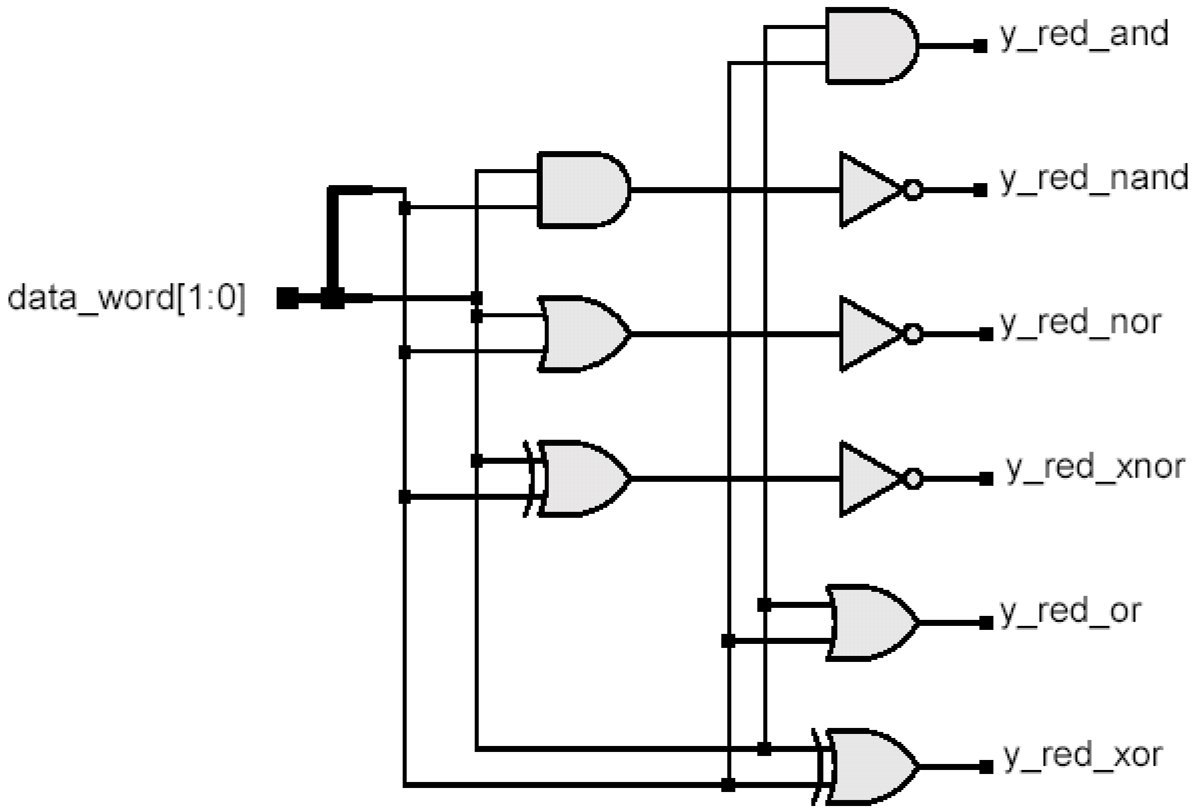
```
module identity_4bit (A_eq_B, A_noteq_B, A, B);

  output         A_eq_B, A_noteq_B;
  input     [3:0] A, B;

  assign A_eq_B = (A == B);
  assign A_noteq_B = (A != B);

endmodule
```



Synthesis of Reduction Operators (1/2)

```
module reduction
  (data_word, y_red_and, y_red_or, y_red_nand, y_red_nor, y_red_xor, y_red_xnor);
  input     [1:0] data_word;
  output          y_red_and, y_red_or, y_red_nand,
                  y_red_nor, y_red_xor, y_red_xnor;
  reg             y_red_and, y_red_or, y_red_nand,
                  y_red_nor, y_red_xor, y_red_xnor;

  always @ (data_word) begin
    y_red_and   <= &    data_word;
    y_red_or    <= |    data_word;
    y_red_nand  <= ~&   data_word;
    y_red_nor   <= ~|   data_word;
    y_red_xor   <= ^    data_word;
    y_red_xnor  <= ~^   data_word;
 end
endmodule
```

Synthesis of Reduction Operators (2/2)

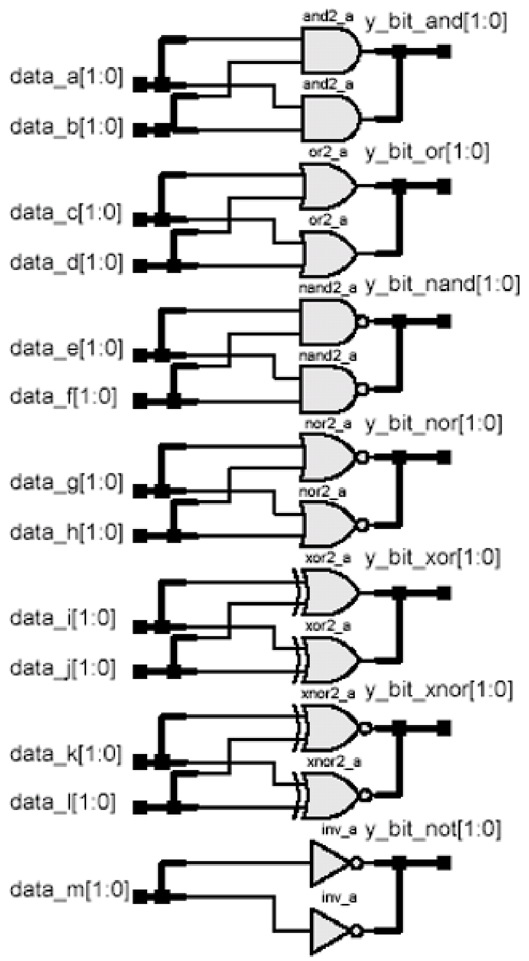


Synthesis of Bitwise Operators (1/2)

```
module bitwise (
      data_a,   data_b,   data_c,   data_d,   data_e,   data_f,   data_g,   data_h,
      data_I,   data_j,   data_k,   data_l,   data_m,
      y_bit_and,          y_bit_or,           y_bit_nand,
      y_bit_nor,          y_bit_xor,          y_bit_xnor, y_bit_neg);

 input        [1:0] data_a,   data_b,   data_c,   data_d,   data_e,   data_f,   data_g
                    data_h,   data_I,   data_j,   data_k,   data_l,   data_m;
 output  [1:0] y_bit_and,     y_bit_or,       y_bit_nand,     y_bit_nor,
               y_bit_xor,     y_bit_xnor,     y_bit_neg;
 reg    [1:0]  y_bit_and,     y_bit_or,       y_bit_nand,     y_bit_nor,
               y_bit_xor,     y_bit_xnor,     y_bit_neg;
```

Synthesis of Bitwise Operators (2/2)

```
                                                                  or
always @ (data_a or data_b or data_c or data_d    data_e or data_f  or
           data_g or data_h or data_l    data_j or data_k or data_l or data_m)
 begin                                        or
  y_bit_and   = data_a  &    data_b;
  y_bit_or    = data_c  |    data_d;
  y_band      = data_e  &    data_f;
  y_bit_nand = ~y_band;
  y_bor       = data_g  |    data_h;
  y_bit_nor   = ~y_bor;

  y_bit_xor   = data_i  ^    data_j;
  y_bit_xnor  = data_k  ~^   data_l;
  y_bit_neg   =         ~    data_mb;

end
endmodule
```



Synthesis of Conditional Operator (1/2)

```
module equality_if_4bit (data_out, data_a, data_b, data_c, data_d, sel);
  input [3:0]     data_a, data_b, data_c, data_d;
  input [1:0]     sel;
  output          data_out;
  reg             data_out;

  always @ (data_a or data_b or data_c or data_d or sel)
    begin
      if (sel == 2'b00) data_out = data_a; else
      if (sel == 2'b01) data_out = data_b; else
      if (sel == 2'b10) data_out = data_c; else
                        data_out = data_d;                // For illustration
    end
endmodule
```

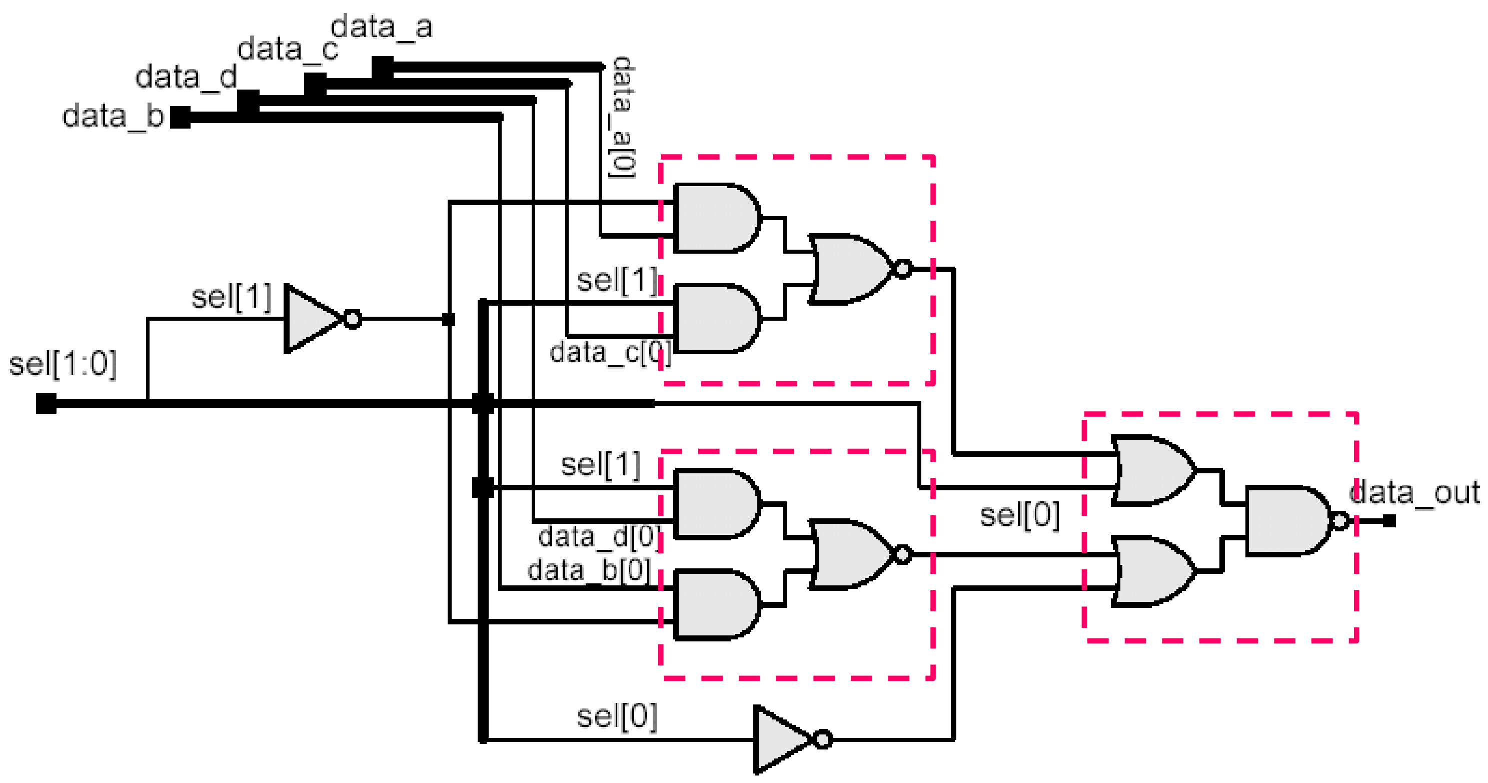
Synthesis of Conditional Operator (2/2)

```
module mux_conditional (data_out, data_a, data_b, data_c, data_d, sel);
  input     [3:0] data_a, data_b, data_c, data_d;
  input     [1:0] sel;
  output    [3:0] data_out;

  assign data_out = (sel == 2'b00) ? data_a :
                    (sel == 2'b01) ? data_b :
                    (sel == 2'b10) ? data_c : data_d;

endmodule
```

The conditional operator synthesizes into library muxes or equivalent gates. A conditional operator with feedback will synthesize into a latch. A conditional operator with an assignment of 'z'' will synthesize to a three-state gate.

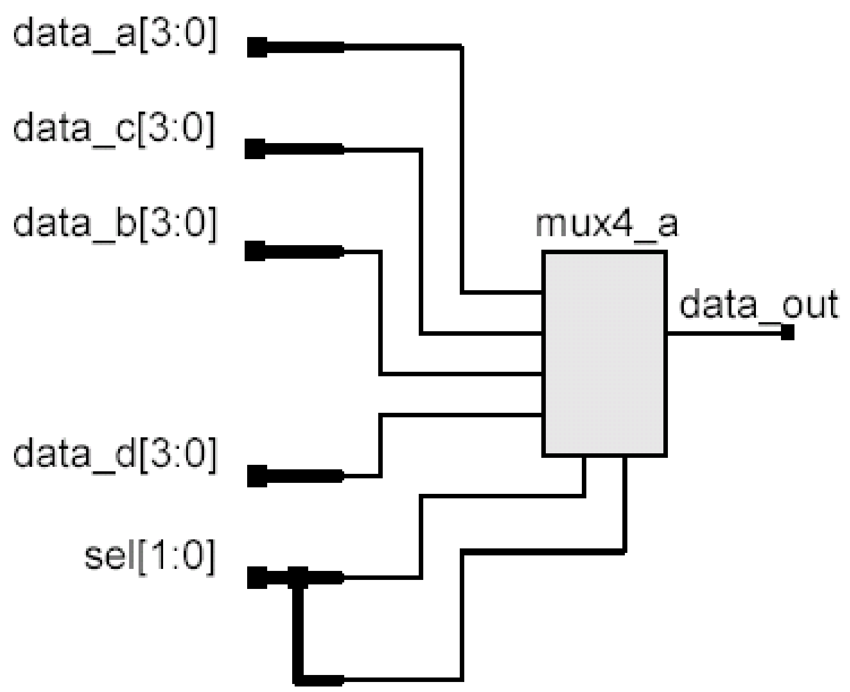
Synthesis Result (1/2)

Synthesis in library having only a two-channel mux:



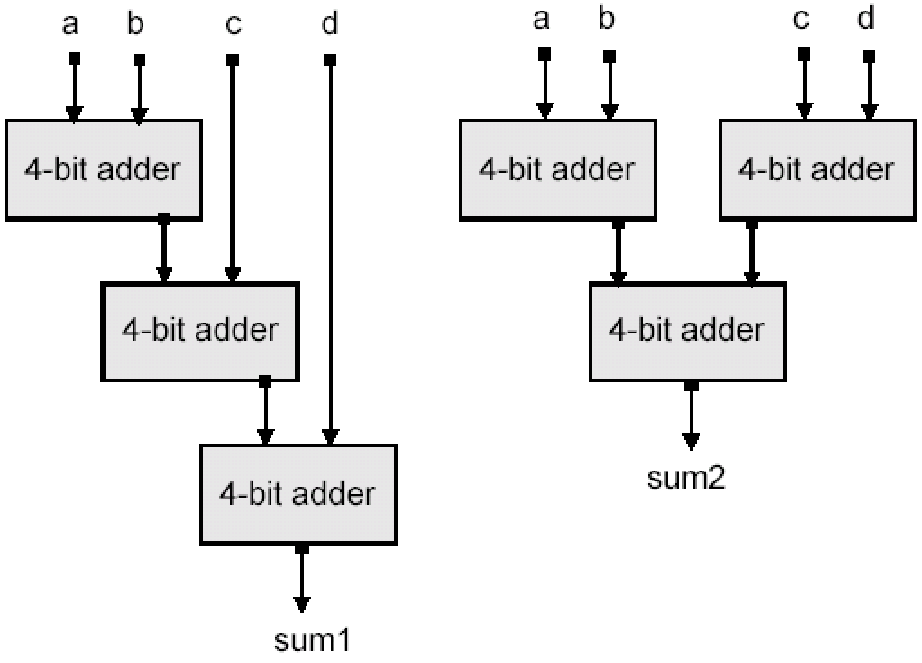
Synthesis Result (2/2)

Synthesis in a library having a four-channel mux:



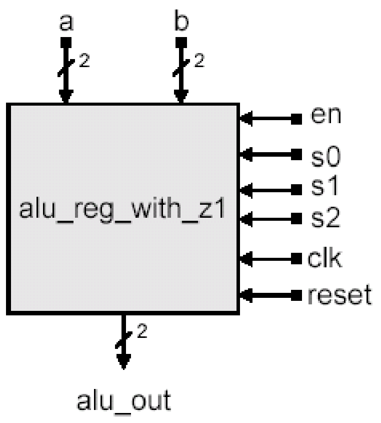
Grouping of Operators

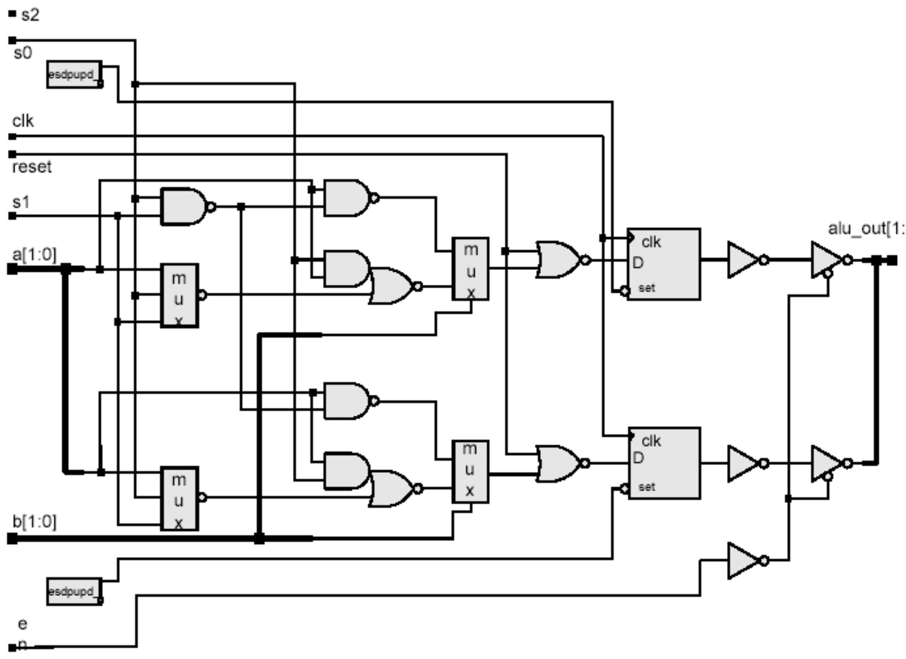
```
module operator_group
  (sum1, sum2, a, b, c, d);
  input    [3:0] a, b, c, d;
  output   [4:0] sum1, sum2;

  assign sum1 = a + b + c + d;
  assign sum2 = (a + b) + (c + d);

endmodule
```



Synthesis of “case” Statements (1/2)

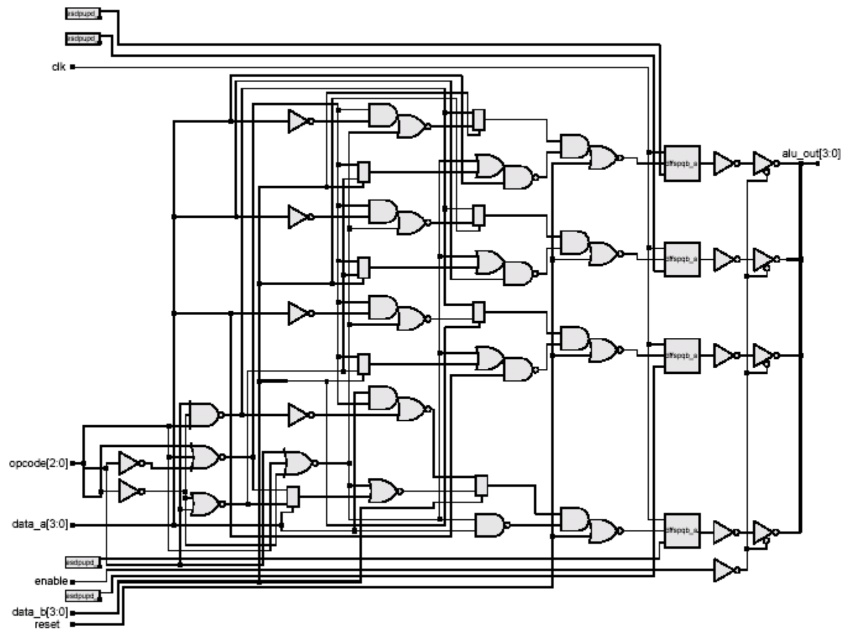
```
module alu_reg_with_z1 (alu_out, en, a, b, s0, s1, s2, clk, reset);
  output        [1:0] alu_out;
  input         [1:0] a, b;
  input               en, s0, s1, s2, clk, reset;
  reg           [1:0] out_int;

  assign alu_out = (en ==1) ? out_int : 2’bz;

  always @  (posedge clk)
   if (reset) out_int = 2’b0;
   else
    case ({s0,s1,s2})
     3’b111 :   out_int = a & b;
     3’b011 :   out_int = a | b;
     3’b001 :   out_int = a ^ b;
     default:   out_int = 2'bx;
   endcase
endmodule
```





Synthesis of “case” Statements (2/2)

```
module   alu_reg_with_z2 (alu_out, data_a, data_b, enable, opcode, clk, reset);
  input            [2:0] opcode;
  input            [3:0] data_a, data_b;
  input                  enable, clk, reset;
  output   [       3:0]  alu_out;
  reg              [3:0] alu_reg;

  assign alu_out = (enable ==1) ? alu_reg : 4'bz;

  always @  (posedge clk)
    if (reset) alu_reg = 0; else
    case (opcode)
      3'b001: alu_reg = data_a | data_b;
      3'b010: alu_reg = data_a ^ data_b;
      3'b101: alu_reg = data_a & data_b;
      3'b110: alu_reg = ~data_b;
      default: alu_reg = 0;
    endcase
endmodule              // Less flexibility
```



Reference

1. 教師自製
2. Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL, Michael D. Ciletti, ISBN: 0139773983, Prentice Hall, 1999.