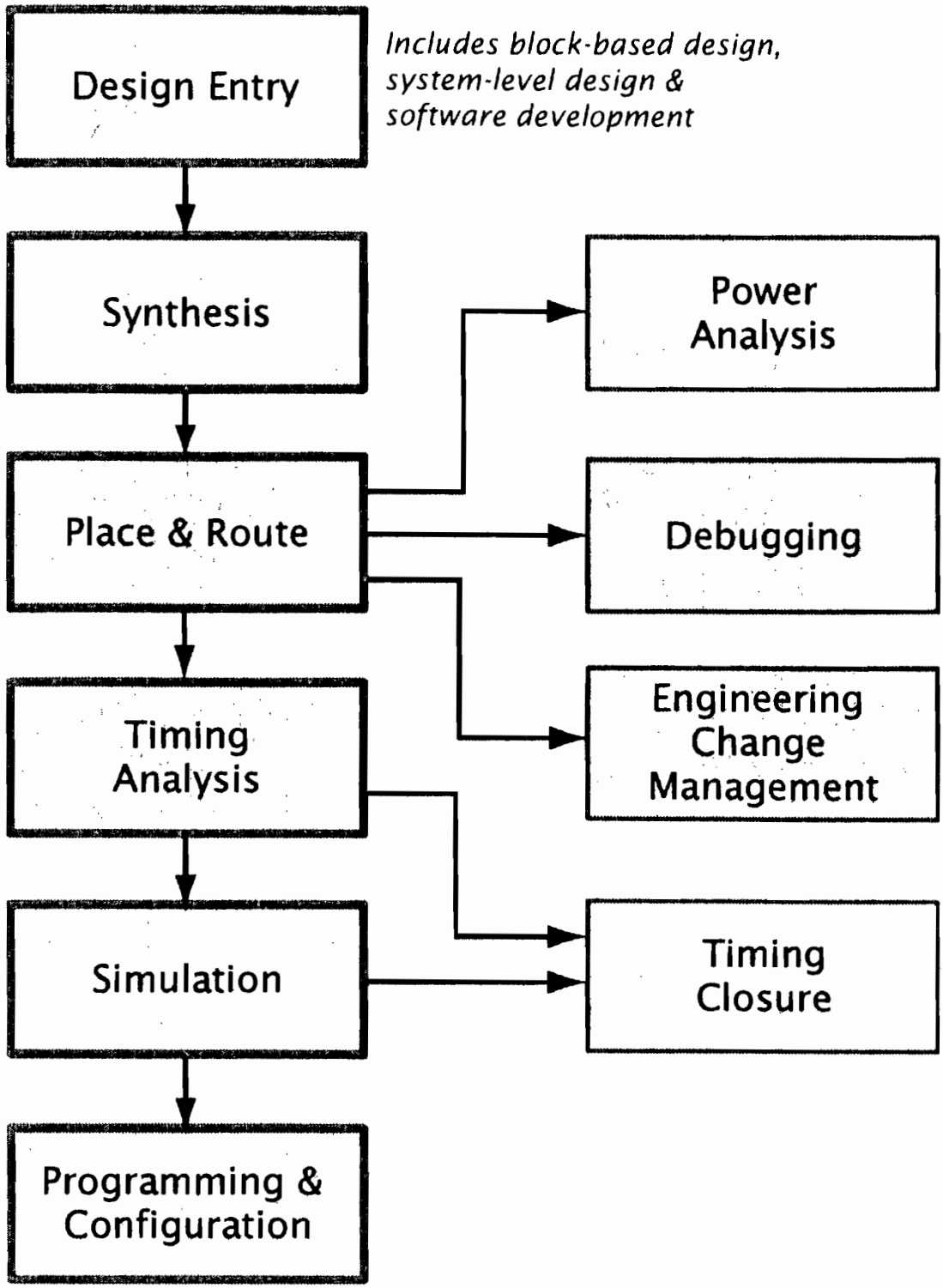
Chapter 3 · Synthesis of Verilog HDL Procedures

Figure 1. Quartus II Design Flow



Quick overview of the Synthesis Process

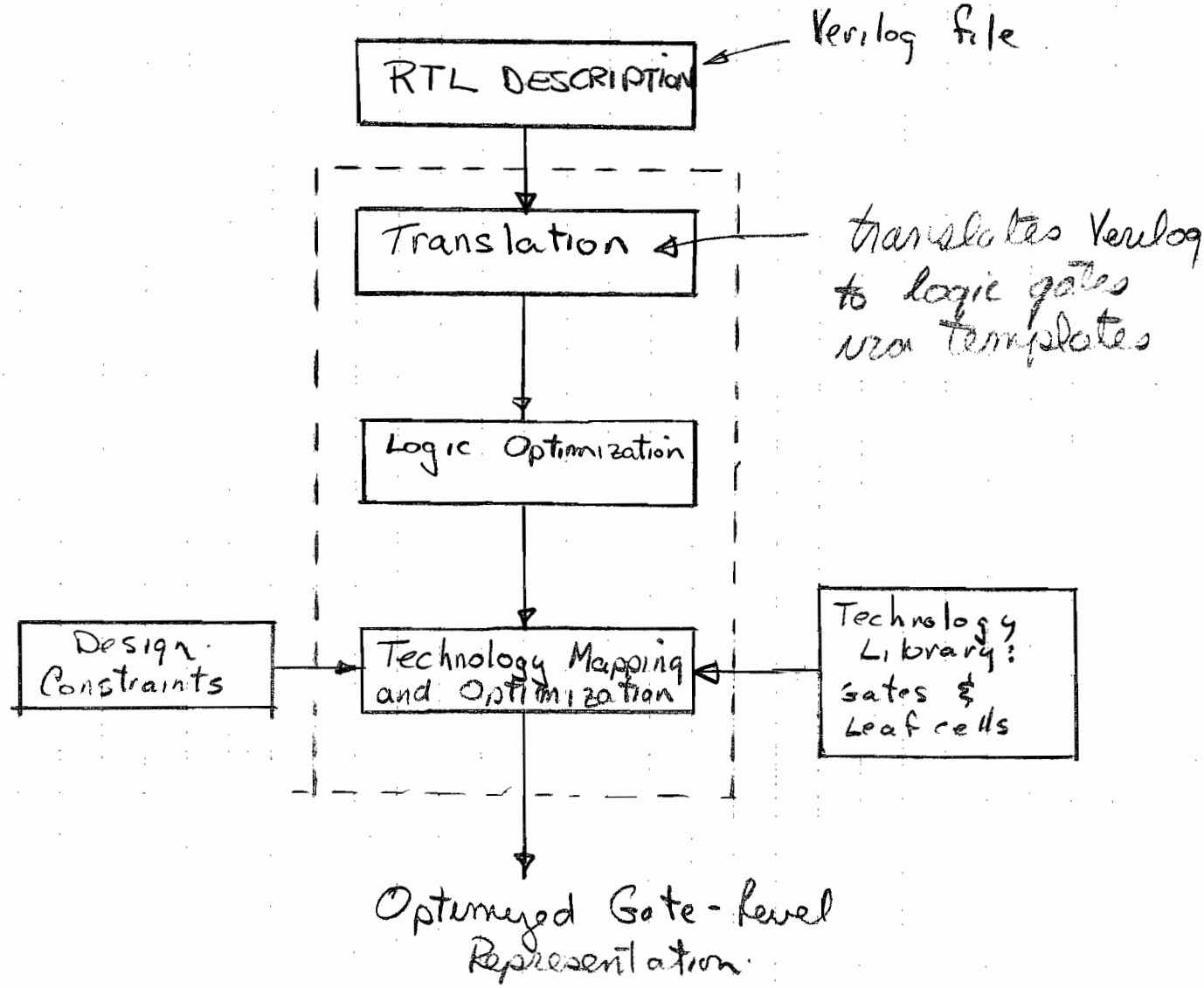
The compiler uses the Verilog HDL file as input.

Using templates for if-else statement, case statements etc it generates a circuit from standard logic gates.

Then an "optimizer" program is executed. If the "optimize for speed" option is selected it finds the fastest possible equivalent circuit.

Then the optimized output is processed by another program called a "Technology mapper". This program converts the logic in the circuit into the technology available on the device. For example a "sea of nand gate" IC, all logic circuits would be converted to equivalent circuits built entirely from "nand" gates. In the case of FPGAs logic is constructed from look-up tables.

Synthesis


RTL DESCRIPTION
Verilog file
Translation
translates Verilog to logic gates using templates
Logic Optimization
Design Constraints
Technology Mapping and Optimization
Technology Library: Gates & Leaf cells
Optimized Gate-level Representation

Designing Combinational Logic using Procedures

```
input  [3:0]  a, b;
output [3:0]  c;
reg    [3:0]  c;
```

NOTE: Any outputs generated in a procedure must be declared "reg"

indicates procedural statement is to follow

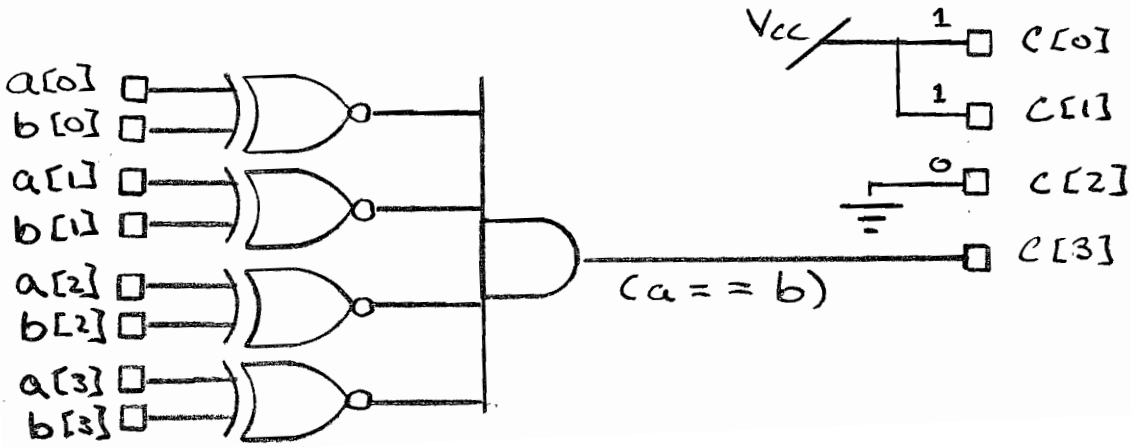
```
always @ (a or b)
begin
    if (a == b)
        c = 4'b1011;
    else
        c = 4'b0011;
end
```

Sensitivity list - used only by simulators but many synthesizers check for it so should be included.

wildcard (* used instead of typing complete list)

* begin-end not needed here - only needed when more than 1 statement is used in a procedure.

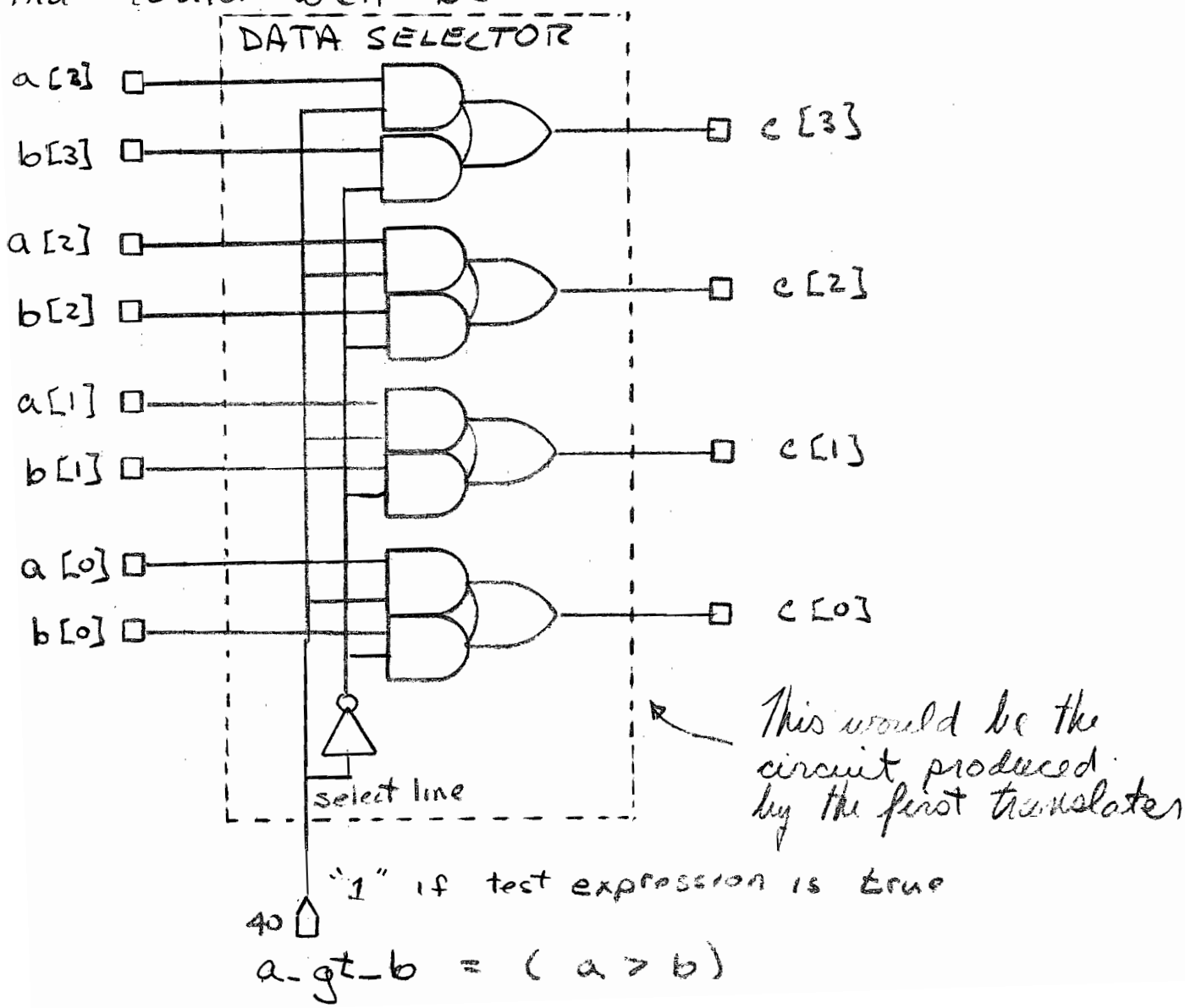
One circuit the synthesizer (compiler) may produce after optimization



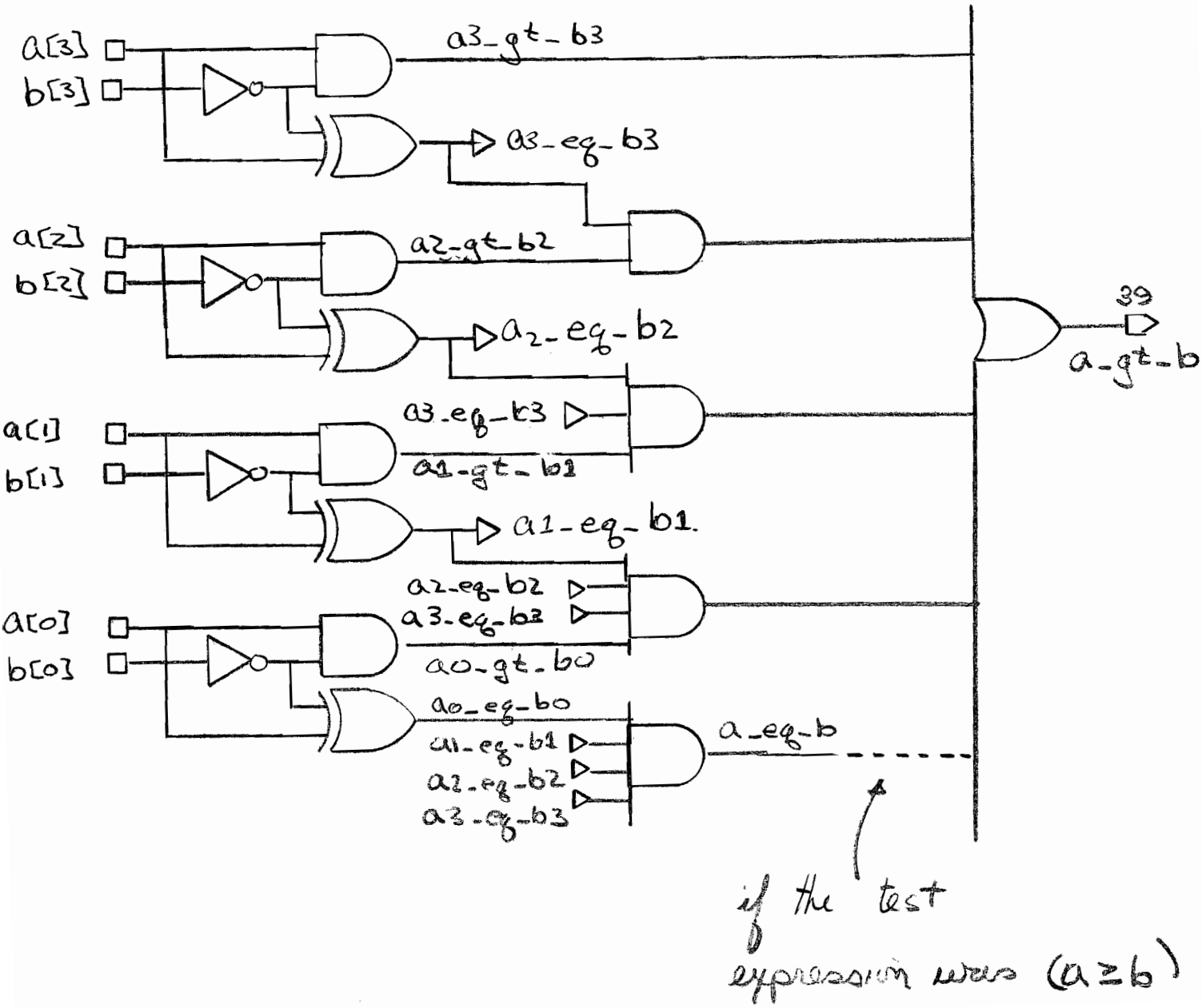
Circuit Produced by Translater

```
always @ (a or b)    // could use always @ *
    if (a > b)  c = a;
    else        c = b;
```

A compiler would translate this into a circuit that could well be

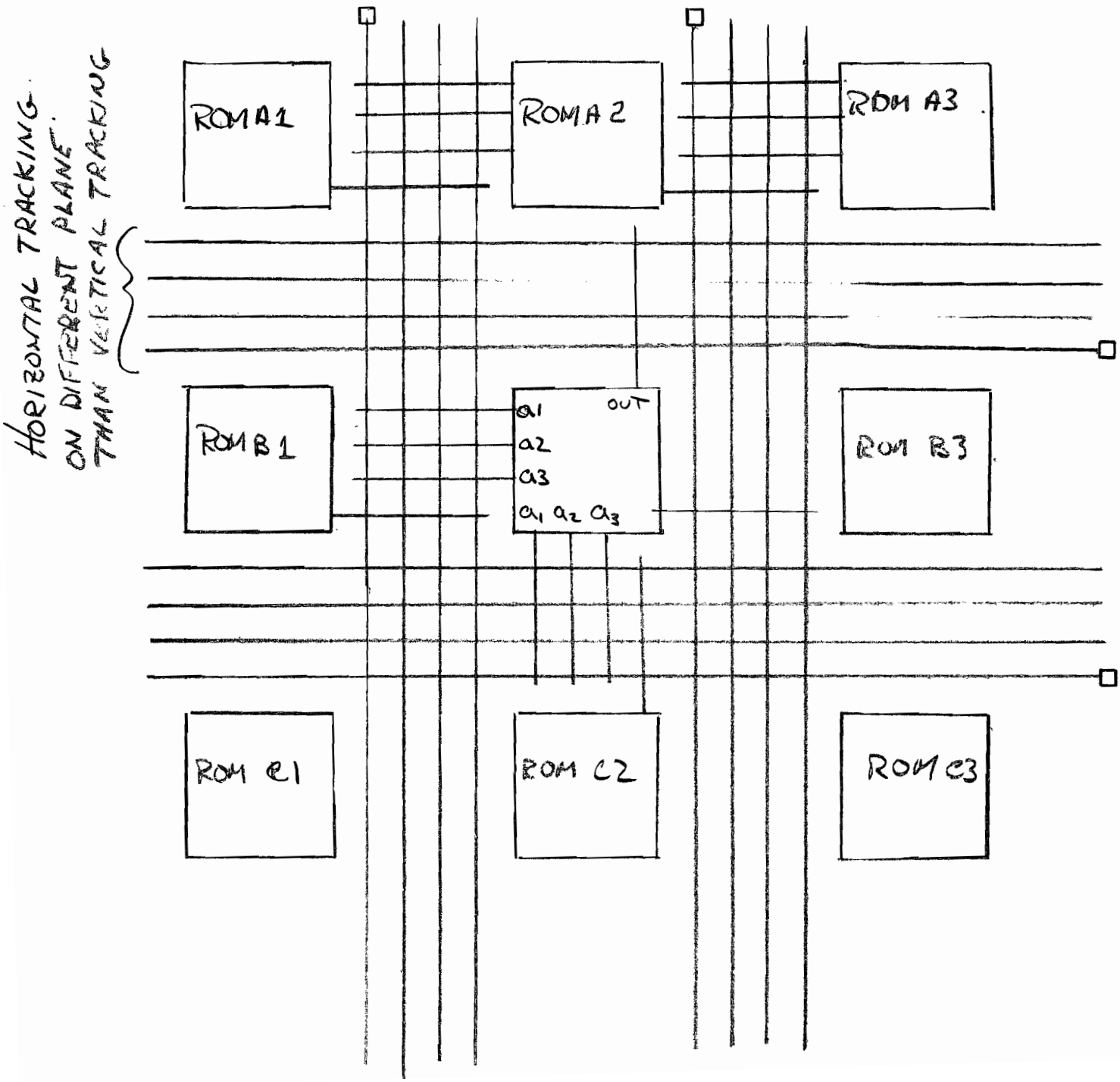


Circuit that evaluates the test expression (a > b)

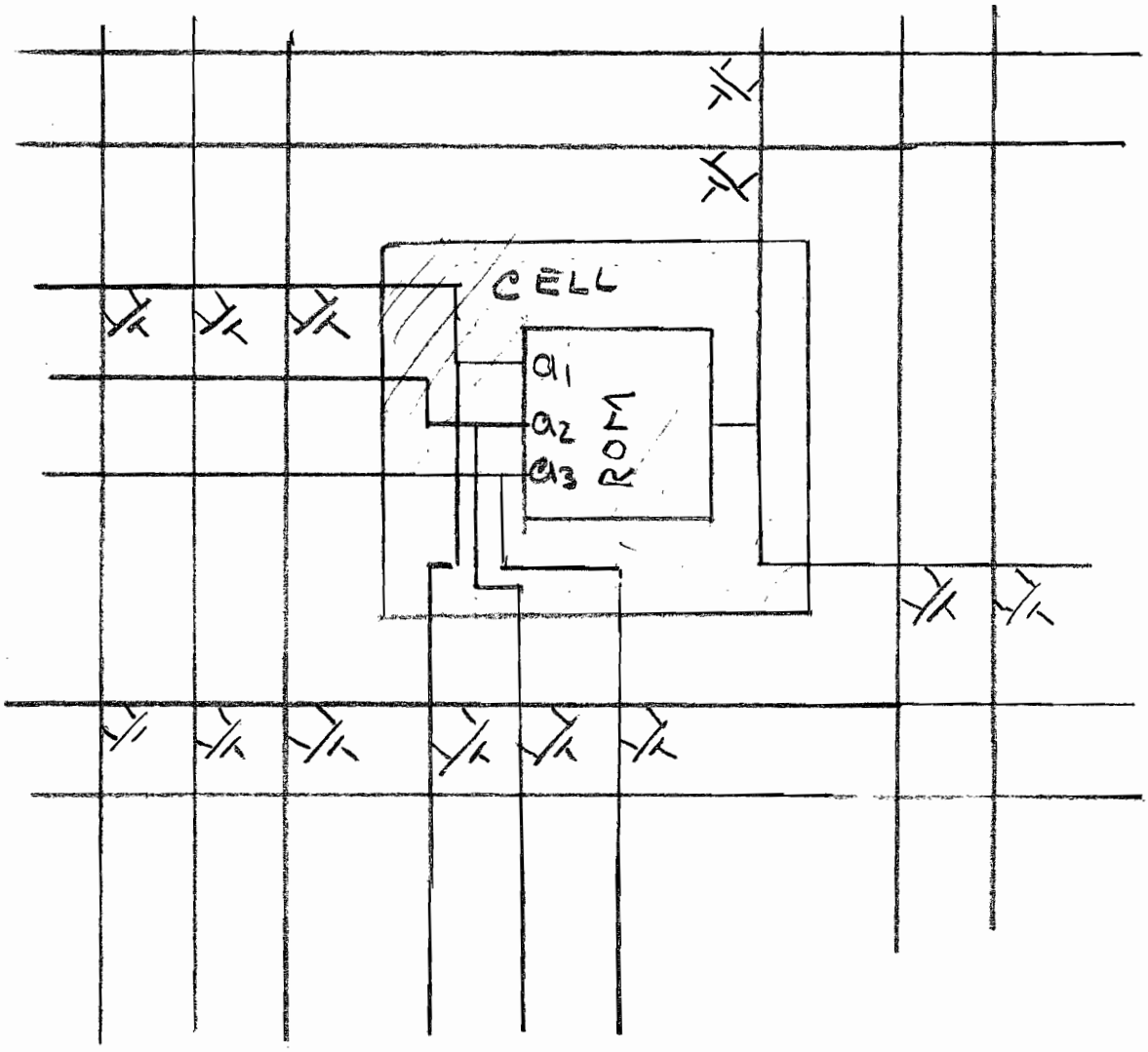


Technology Mapping

Suppose the techology is a contrived EE431 FPGA with the following architecture.

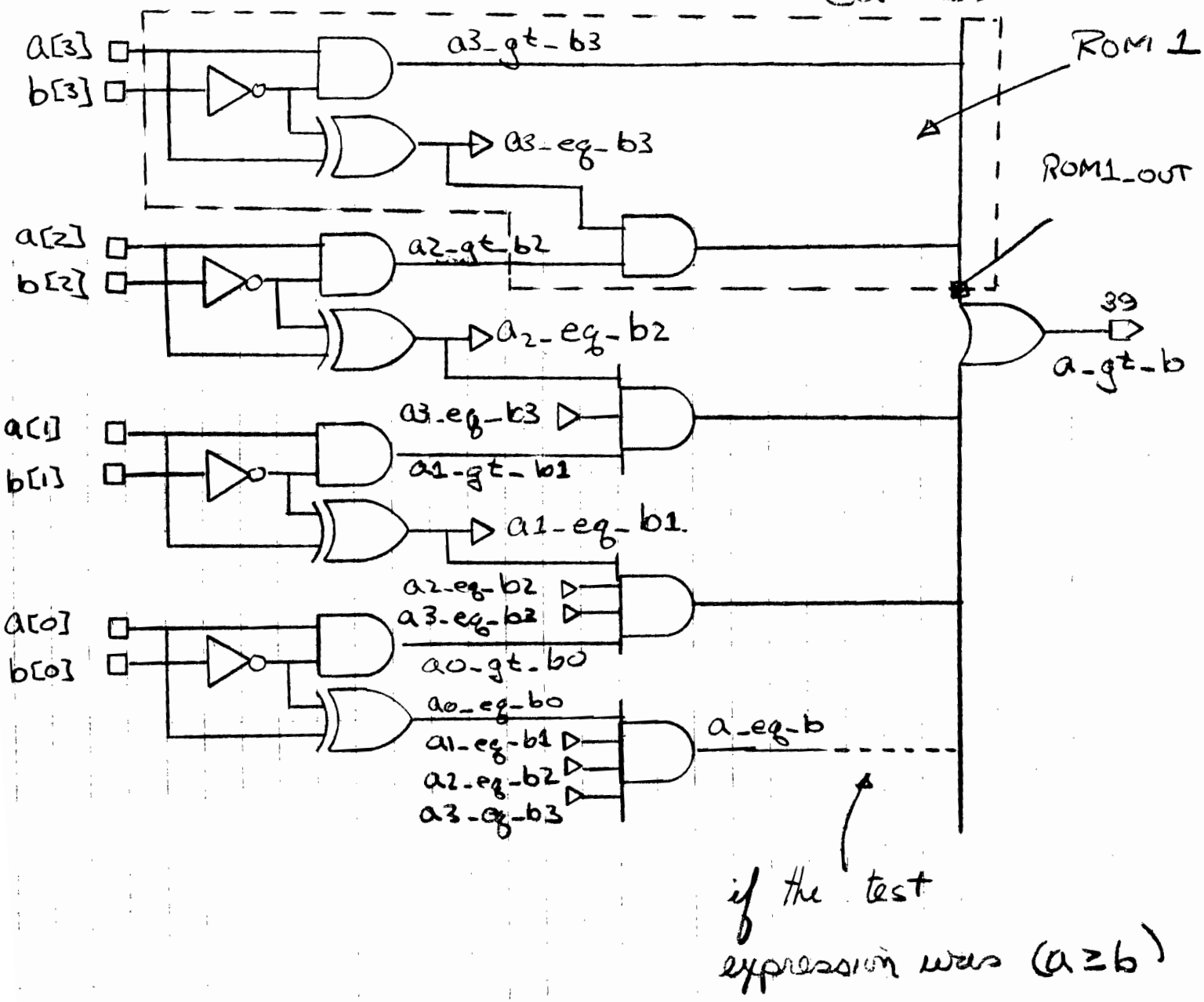


HORIZONTAL & VERTICAL TRACKING ARE ON DIFFERENT PLANES. CONNECTIONS ARE PROGRAMED WITH TRANSISTOR SWITCHES.



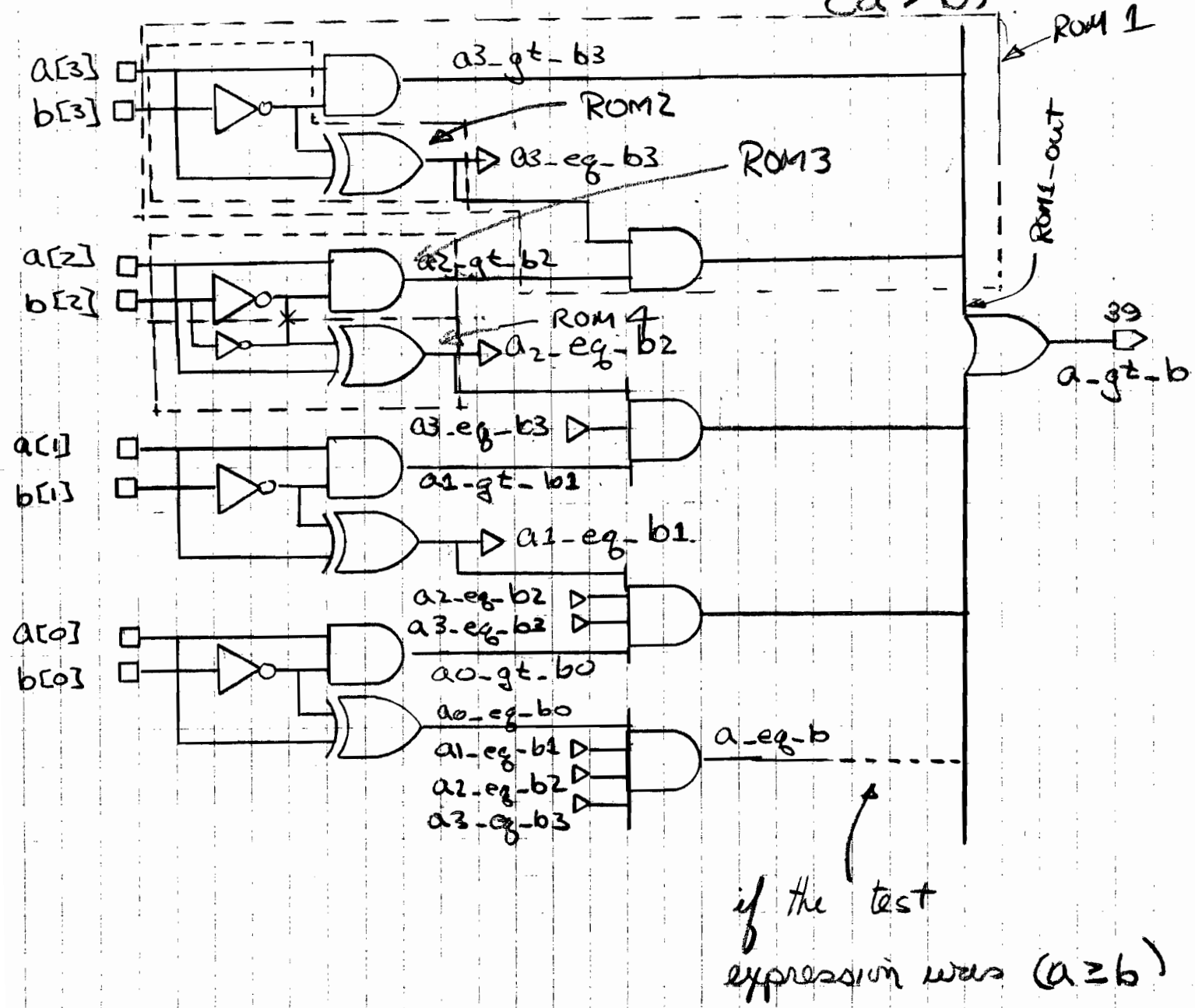
The three inputs and one output of each ROM. can be connected to either the horizontal or vertical Tracks

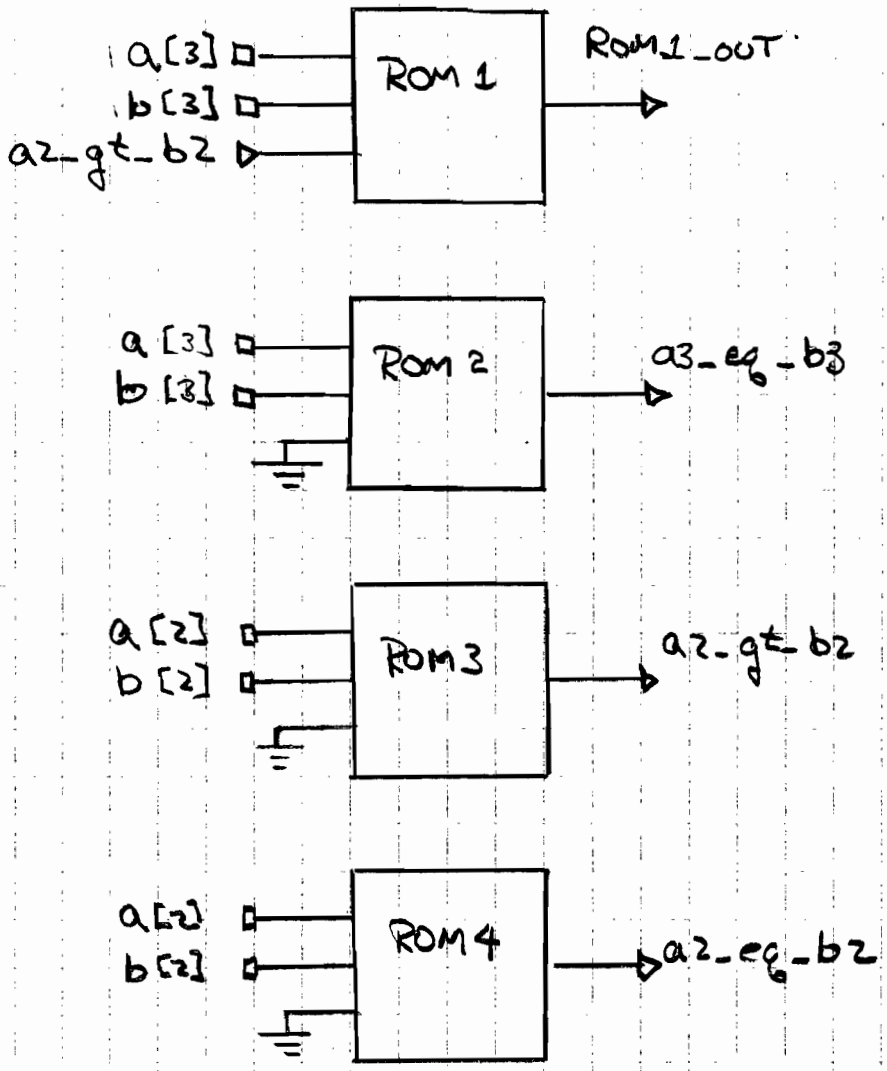
Circuit that evaluates the test expression (a > b) 40 C)



if the test expression was (a ≥ b)

Circuit that evaluates the test expression (a > b) 40d)



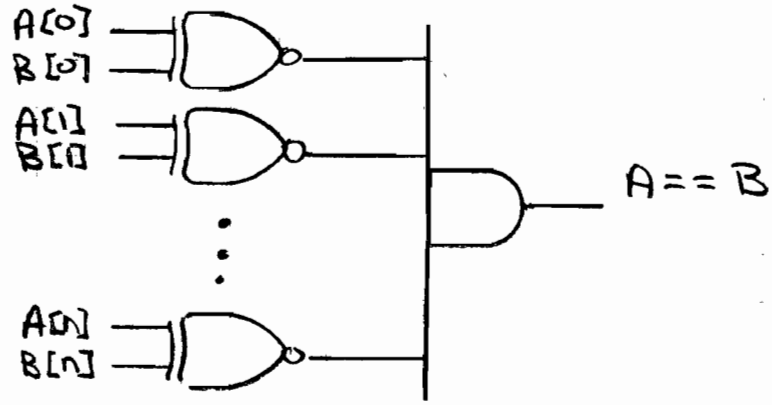


After the LOGIC IS MAPPED TO ROMS, the ROMS are placed on the FPGA (using a complicated algorithm) and then the routing is done.
(wires are connected via transistor switches to connect all signals with the same name).

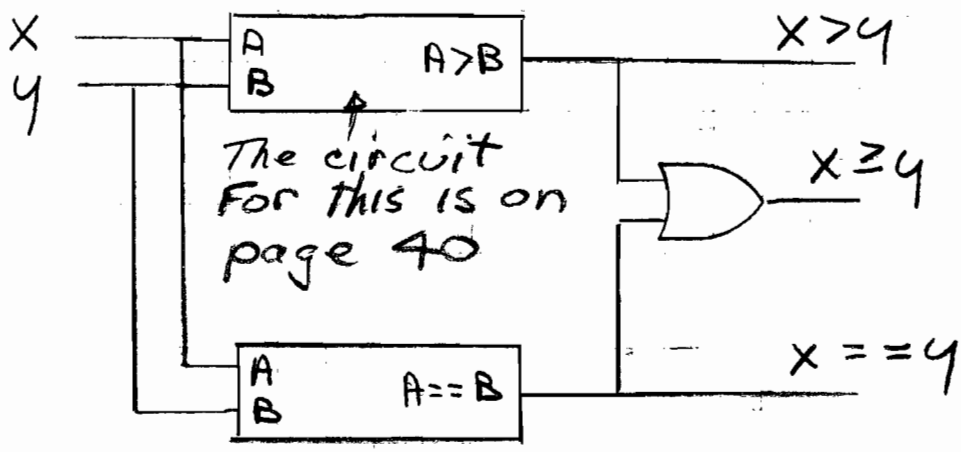
Sometimes it is not possible to route signals for a particular placing of ROM, in which case the placement of ROMs is changed.

TEMPLATES FOR RELATIONAL OPERATORS

A == B



X > y , X ≥ y , X < y , X ≤ y .



For X < y and X ≤ y the inputs are reversed. X is connected to the B inputs and y is connected to the A inputs

Quartus Netlist viewer tools.

Quartus provides tools to view the output of the synthesizer after translating Verilog to logic and optimizing the logic. It provide a schematic diagram at the Register Transfer Level and the tool that does this is called the RTL viewer. This schematic is generated by selecting

TOOLS → NETLIST VIEWERS → RTL VIEWER.

Quartus also provides a tool for viewing the technology-mapped schematic produced by the synthesizer. This is before the placing and routing has been done. This schematic is generated by selecting

TOOLS → NETLIST VIEWERS →

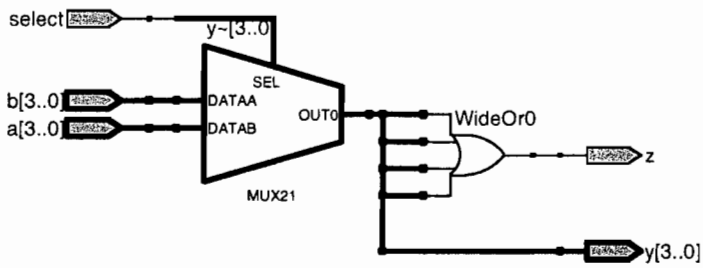
TECHNOLOGY MAP VIEWER (POST MAPPER)

The post place & route map (i.e. the post fitter map) is displayed by selecting

TOOLS → NETLIST VIEWERS → TECNOLOGY MAP VIEWER

CIRCUIT #1 4(a)

RTL VIEWER.




```
module circuit_1 (
    input [3:0] a, b,
    input select,
    output reg z,
    output reg [3:0] y );
always@ *
  if (select == 1'b1)
      y = a;
  else
      y = b;
always@ *
      z = |y;
endmodule
```

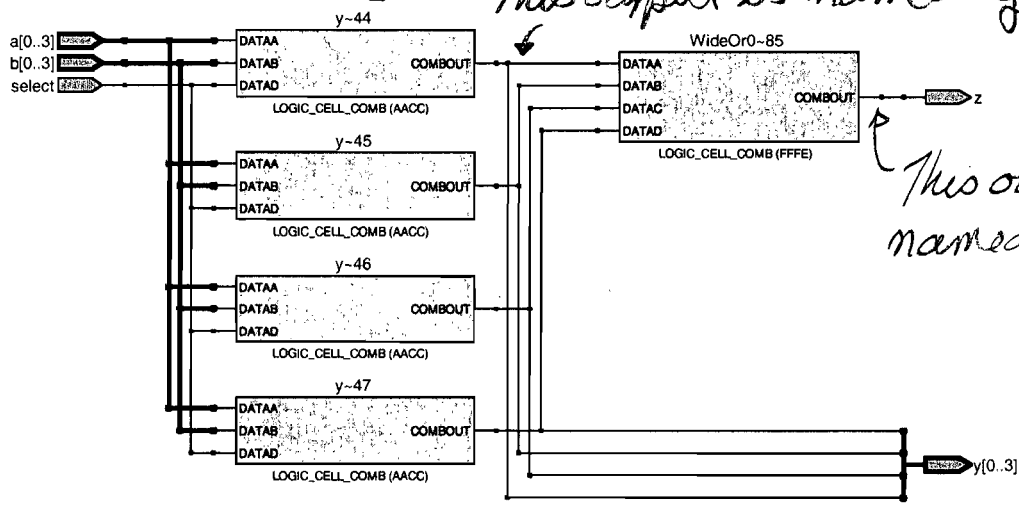
Verilog 2001 allows the signal types to be declared in the port list

 [redacted]

CIRCUIT #1 41 b)

POST MAPPER but before fitter

This output is named y~44



This output is named WideOr0~85

Each block is a 16X1 ROM (i.e. Look-up-table)
It has four inputs DATAA, DATAB, DATAC, DATAD
The output is called COMBOUT (the COMB stands for combinatorial logic.

If the mouse is hovered over a LUT it gives the logic equation implemented by that block.

The logic equation for the y~44 LUT is y~44 = DATAD & DATAA # !DATAD & DATAB

means OR

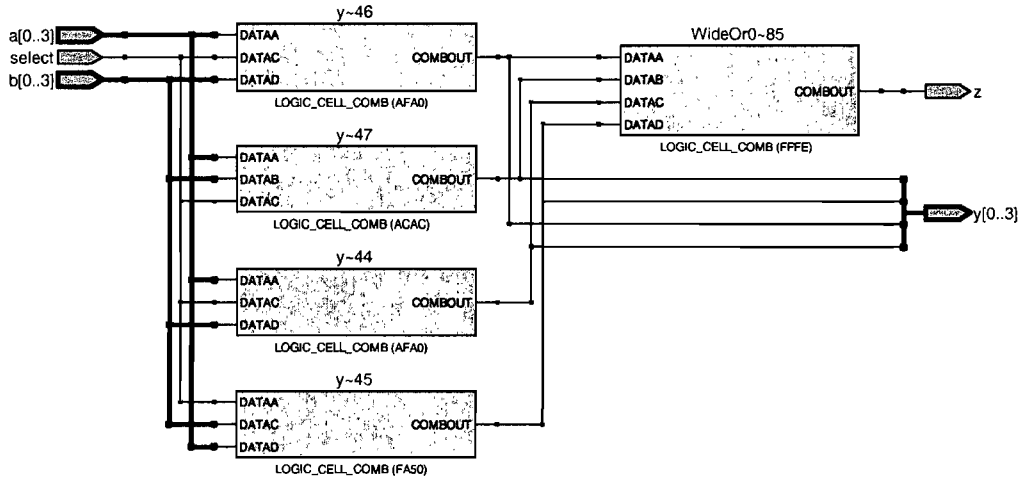
The logic equation for the WideOr0~85 LUT is

WideOr0~85 = DATAA # DATAB # DATAC # DATAD

 [redacted]

Circuit #1
Post Fitter

41 c)

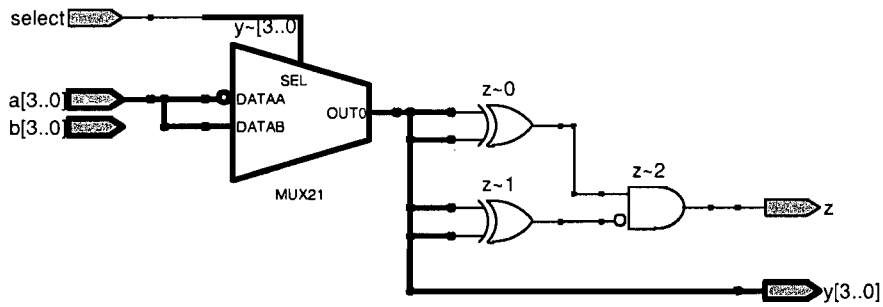


Notice that the fitter changed the wiring. For example the "select" input now goes to input C on LUTs y~46, y~47 and y~44 and goes to input D on LUT y~45

 [redacted]

Circuit_2 4(d)

RTL VIEWER

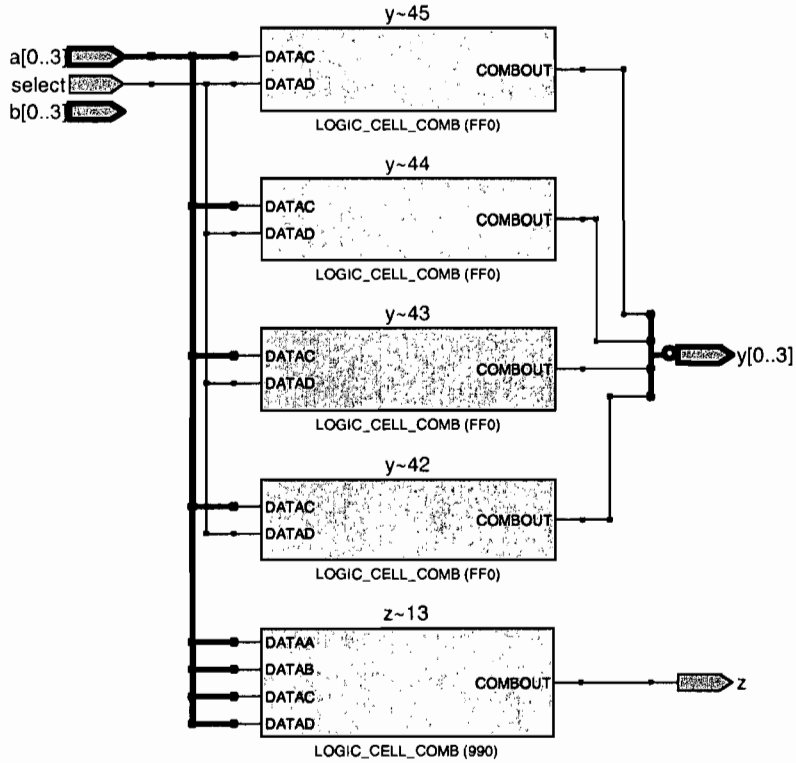



```
Module circuit_2 (
    input [3:0] a, b,
    input select,
    output reg z,
    output reg [3:0] y;
always @ *
  if (select == 1'b1)
        y = a;
  else
        y = ~a;
always @ *
    z = (y[0] ^ y[1]) & (y[2] ~^ y[3]);
end module
```

 [redacted]

Circuit_2 4(e)

Post mapping but before fitting



Logic in LUT y~45

y~45 = DATAC $ DATAD

↑ This symbol means XOR.

Logic in LUT z~13

or . not

z~13 = DATA & DATAC & (DATAB $ DATAD) # !DATAA

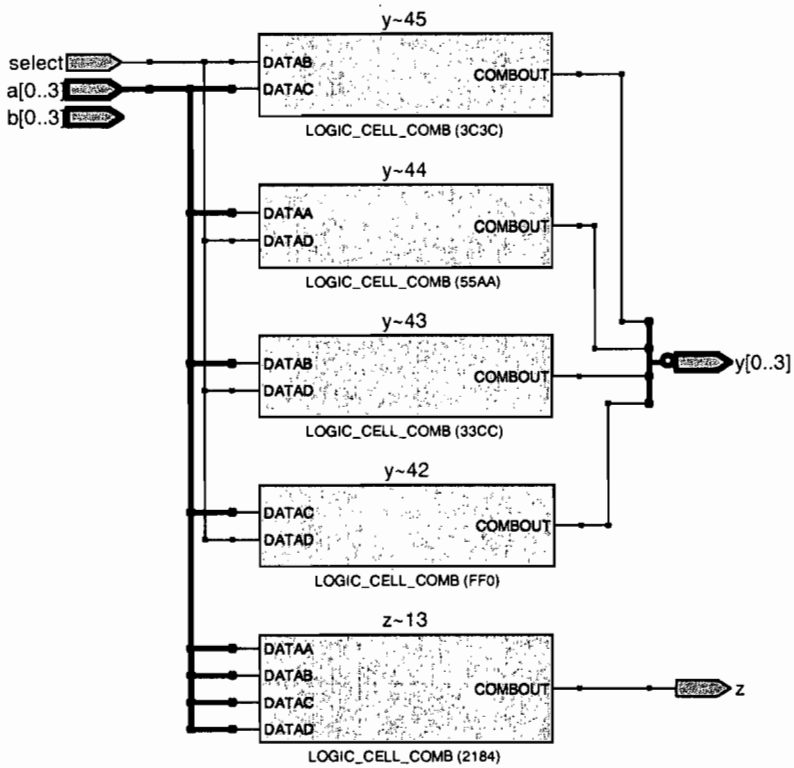
↑ XOR

& !DATAC & (DATAB $ DATAD)

 [redacted]

CIRCUIT #2 41 f)

Post - fitter

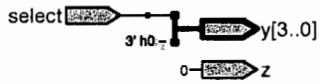


Again some wiring has changed.

 [redacted]

Circuit 3. 41 g)

RTL VIEWER

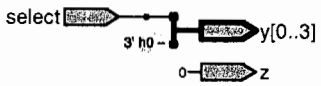



```
module circuit_3 (
    input select,
    output reg z,
    output reg [3:0] y );
always @ *
    if (select == 1'b1)
        y = 4'b1000;
    else
        y = 4'b0000;
always @ *
    z = & y;
end module
```

 [redacted]

CIRCUIT_3 41 h)

Post mapper.

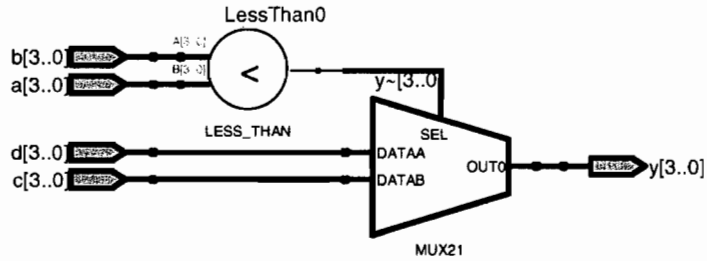


The fitter does not change this so the post fitter technology map is identical to the one above.

 [redacted]

Circuit_4 41 i)

RTL VIEWER



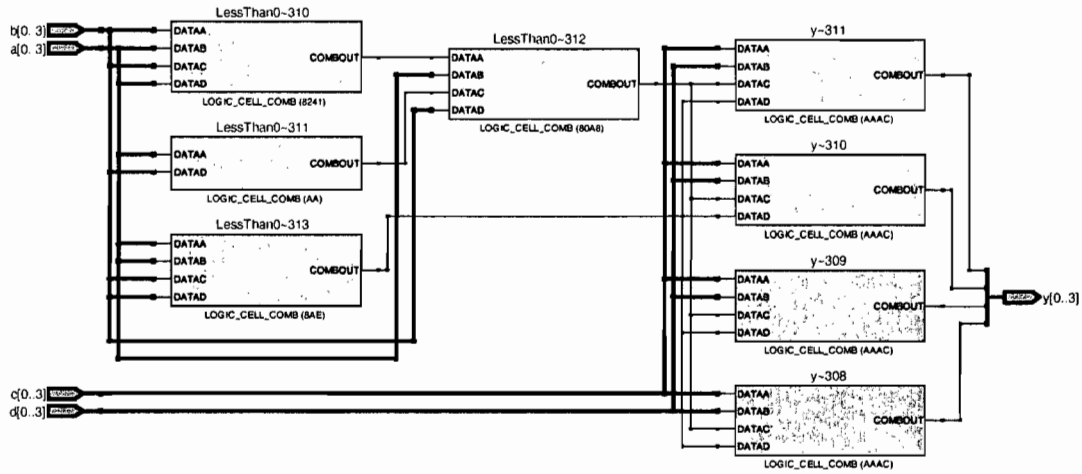

```
module Circuit_4 (
    input [3:0] a, b, c, d,
    output reg [3:0] y );

always @ *
    if (a > b)
        y = c;
    else
        y = d;
endmodule
```

 [redacted]

Circuit_4 4(j)

Post mapper

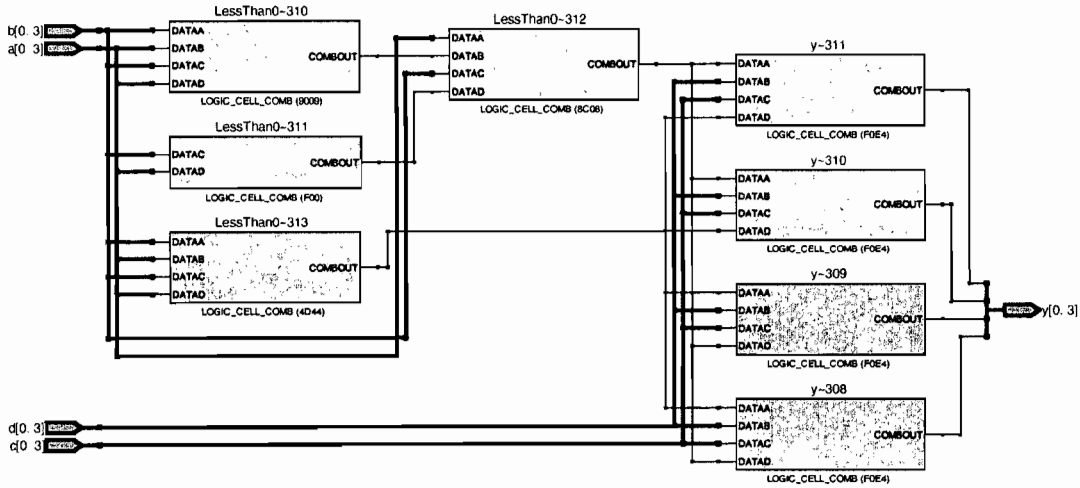


The circuit that evaluates (a > b) takes more than 4 LUT.

It requires LUTs lessthan0~310, lessthan0~311, lessthan0~312 and lessthan0~313. There is also some logic in LUTs y~308, y~309, y~310, y~311. It is clear that y~308 to y~311 LUTs contain more than a 2:1 data selector as 2:1 data selectors have only 3 inputs.

Circuit - 4 41K

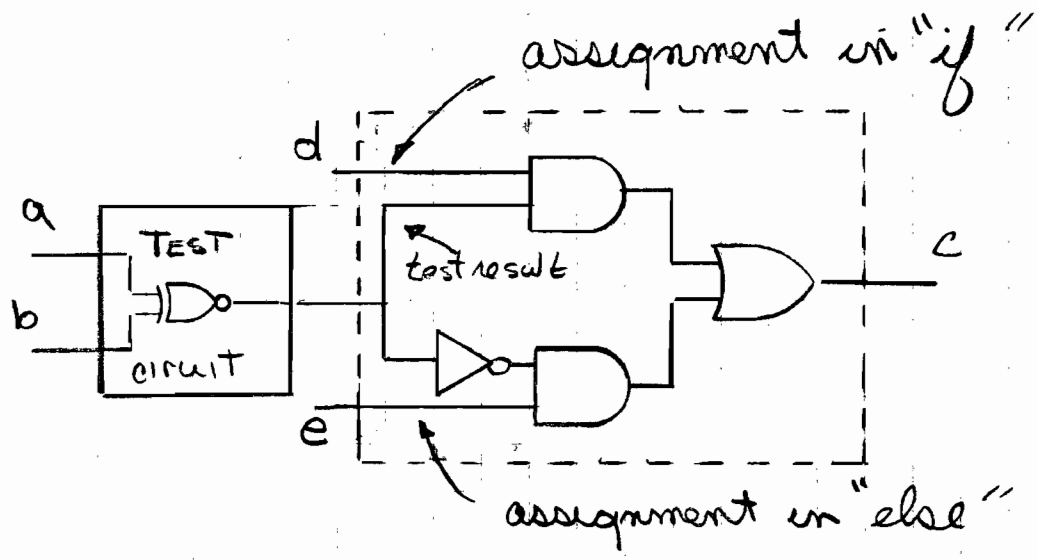
Post fitter. (Post place & route)



The fitter has changed the wiring

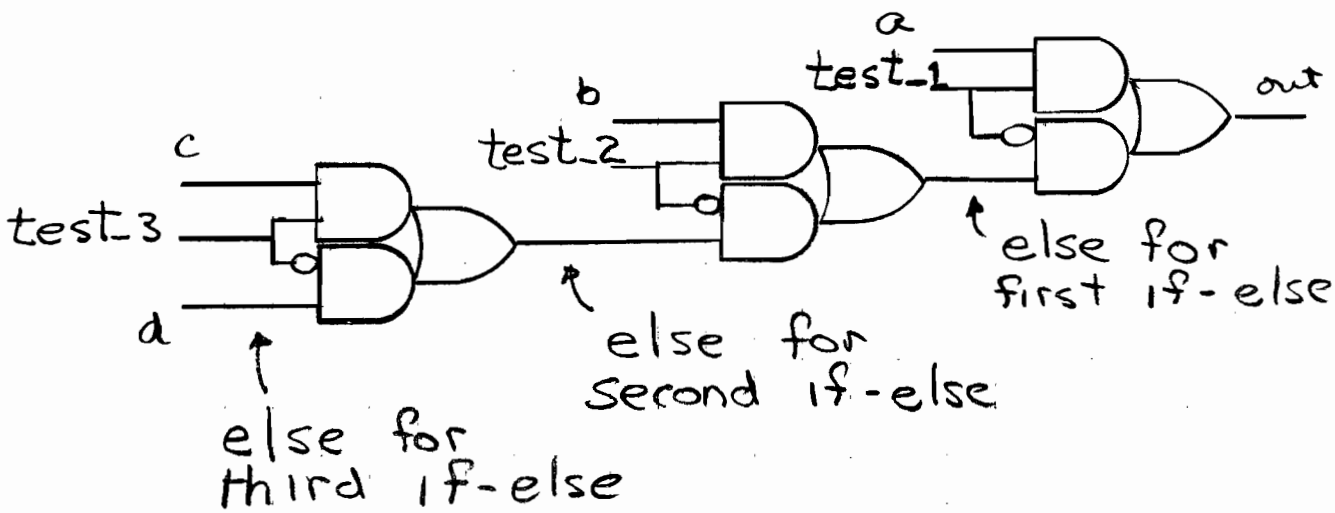
Compilers translate if-else statements into the following hardware

```
reg c;
always @ ( a or b or d or e)
if ( a == b)    c = d;
else            c = e;
```



Nested if-else statement are build by stacking the circuit above.

Nested if-else statements




```
reg out;
always @ (test_1 or test_2 or test_3
          or a or b or c or d)
if (test_1) out = a;
else if (test_2) out = b;
     else if (test_3) out = c;
          else out = d;
```

NOTE: Propagation Time from "a" to "out" is 1/3 that of Time from "c" to "out"

First if assignment is fast

Last if assignment is slow

EXAMPLE

Design a circuit that does the following.

1. operates on two 4-bit inputs, x and y, to produce a 1 bit output Q.
2. The circuit performs a bit reversal on x, compares the result with y and makes Q "high" if and only if the bit-reversed X is equal to y.

```
module compare (x, y, Q);
input [3:0] x, y;
output Q;

reg [3:0] xr;
reg Q;

always @ *
begin
    xr [3] = x [0];
    xr [2] = x [1];
    xr [1] = x [2];
    xr [0] = x [3];
end

always @ *
    if (xr == y)  Q = 1'b1;
    else          Q = 1'b0;
endmodule
```

x_r & Q are built in an always procedure so must be type reg

There must be a begin - end wrapper when more than one statement is used in an always procedure.

Comments on relational expressions.

relation expression.

```
if ( in_1 == 4'b1x11 ) ...
else ...
```

this means ignore in_1[2] in the comparison

The relational expression

(in_1 == 4'b1x11) is equivalent to

((in_1[3] == 1'b1) && (in_1[1:0] == 2'b11))

precedence of relational operator higher than logic operator so do not need brackets

It is also equivalent to

result is a 4 bit vector.

((in_1 & 4'b1011) == 4'b1011)

bit wise and.

need brackets because the bitwise operator has lower precedence than the relational operator.

NOTE: Some compilers do not support don't cares in operands of relational operators. Leonardo Spectrum (2001) Quartus 2009 (ver 8.1) evaluate "in_1 == 4'b1x11" to false because in_1[2] is not equal to "x"

Quartus does not issue a warning

Use of don't cares

Example.

```
wire [3:0] y;
                    don't care
assign  y = 4'b11xx;
```

This assigns y[3] and y[2] a value of 1 and lets the compiler assign y[1] and y[0] either 0 or 1.

Another Example

```
always @ *
if (test_1)  z = a;
else  if (test_2)  z = b;
      else         z = 4'b11xx;
```

When don't cares are used in this situation it means the compiler can assign any value it wants so it can reduce the circuit.

NB: In my opinion assigning "don't cares" to give the compiler more freedom for optimization is <u>very bad practice</u>.

It does not help much but makes debugging a bit more difficult. When debugging it is very helpful to know what the output should be for every combination of inputs.

I strongly recommend assigning values of zero in don't care situations.

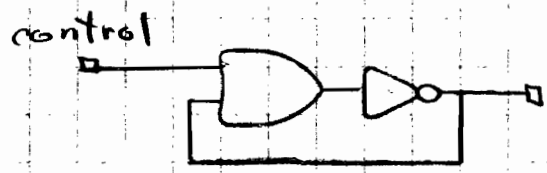
TRANSPARENT LATCHES

```
always @ (Gate or d)
    if (Gate == 1'b1)
        q = d;
    else  q = q;
```

- Ask students to generate schematics, give them about 5 minutes then proceed.

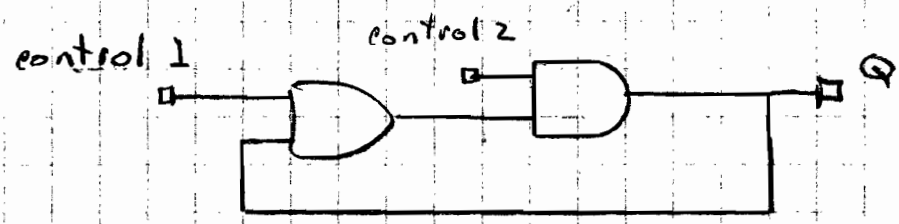
Unstable and Semi Stable Circuits

Unstable Circuits (negative feedback)



When the control is high this circuit will oscilate because there is negative feedback.

Semi stable circuits (positive feedback)

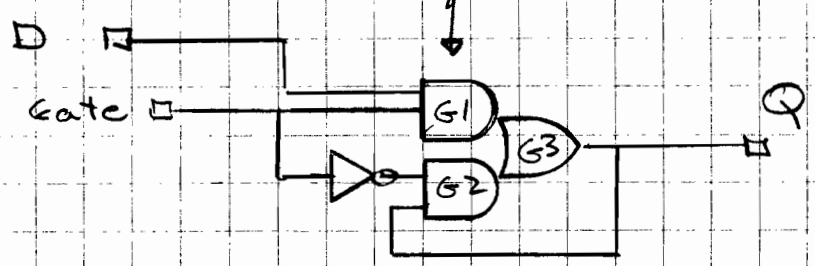


If "control-1" is low and "control 2" is high the output can be stable in either a high or low or low state. (positive feedback)

If the output is low while control-2 is high and a narrow positive pulse is applied to control-1 That pulse can circulate (theoretically forever). The width of the pulse must be less that the progation delay through the two gates.

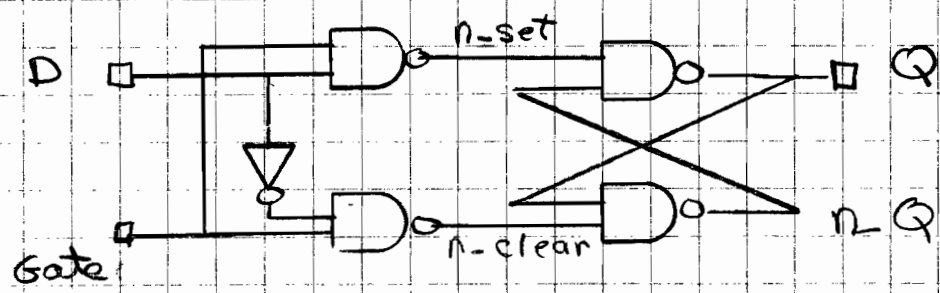
CIRCUIT # 1 (will oscillate)

problem is gate-control turns off G1 before it turns on G2

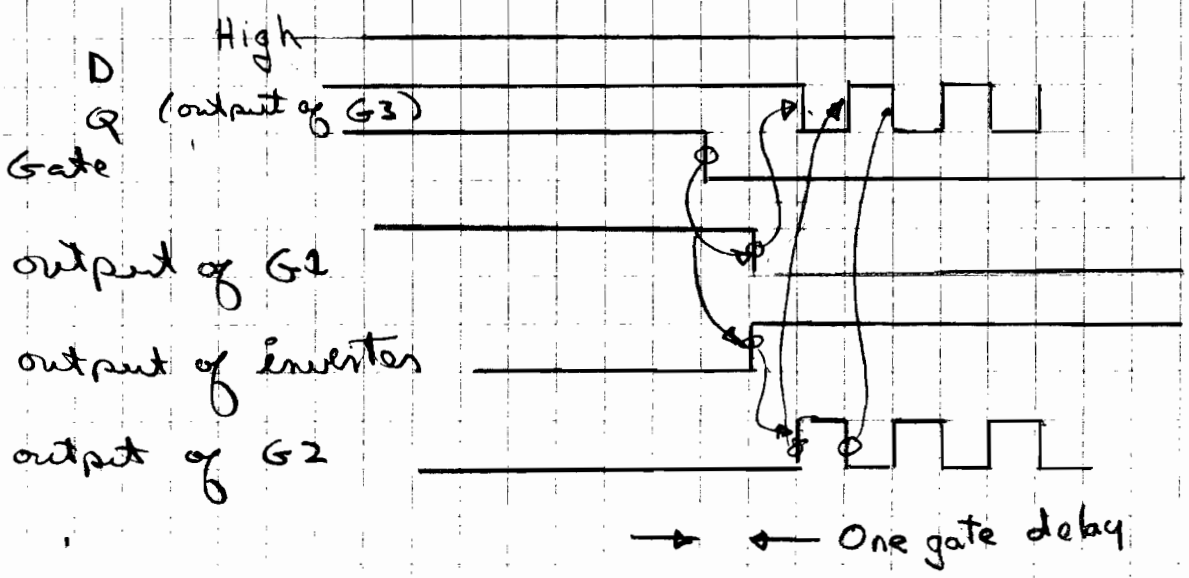


Suppose D & Q are High when Gate goes low. The output of G1 goes low before the output of G2 goes high

CIRCUIT # 2 (GOOD)



TIMING FOR CIRCUIT # 1 (each square = 1 gate delay)

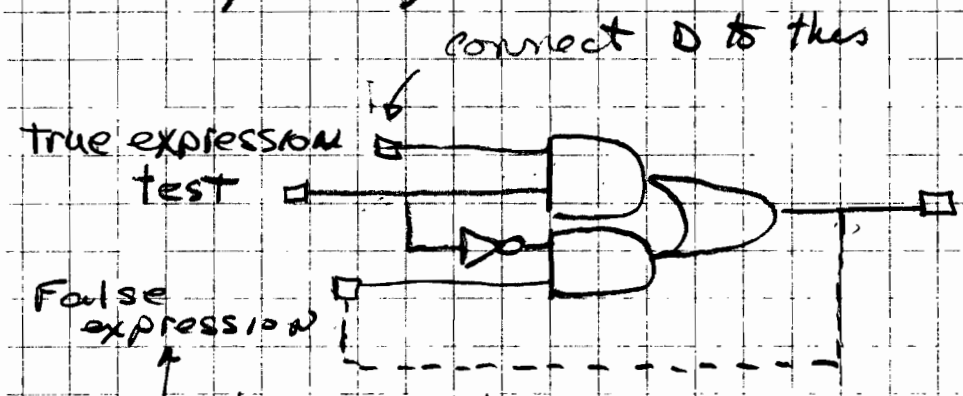


Verilog Procedure for Circuit #1

```
reg Q;
always @ (Gate or D)
if (Gate == 1'b1)    Q = D;
else                 Q = Q;
```

Same as description given as in class assignment.

Notice that circuit 1 fits the template used for if-else statements.



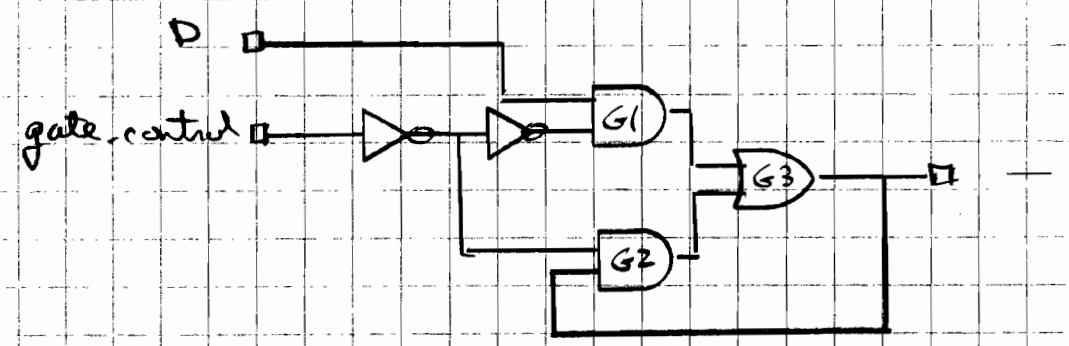
connect Gate to the test input.

Building Circuit #2 from verilog.

NB: This circuit does not fit the if-else template. This structure will not be constructed from if-else statements.

To build this circuit a structural description (e.g. using primitives) must be used.

Modified CIRCUIT #1 (GOOD)



With this circuit the gate turns on G2 before it turns off G1 ∴ the "or" gate is kept on and the output remains high

NB Compiler must recognize the description as a latch and then go to the library & get a latch.

NB If the "else" clause in an if statement is omitted, the compiler assumes the else clause is

output variable

else $q_0 = q_0$;

Procedural statements describe the circuit in terms of how it is to function rather than in terms of how it is to be structured. (Structured means showing the blocks or primitives and how they are to be connected)

Procedural statements describe the behavior of a circuit and not its structure so are called <u>behavioral descriptions</u>.

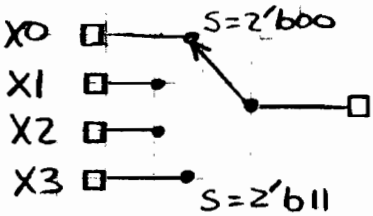
Case Statements

```
module data_selector (x0, x1, x2, x3, s, y);
                                      select
input x0, x1, x2, x3;
input [1:0] s;
output y;
reg y;
always @ *
    case (s)
        2'b00:  y = x0;
        2'b01:  y = x1;
        2'b10:  y = x2;
        2'b11:  y = x3;
        default:  y = 1'b0;
    endcase

endmodule
```

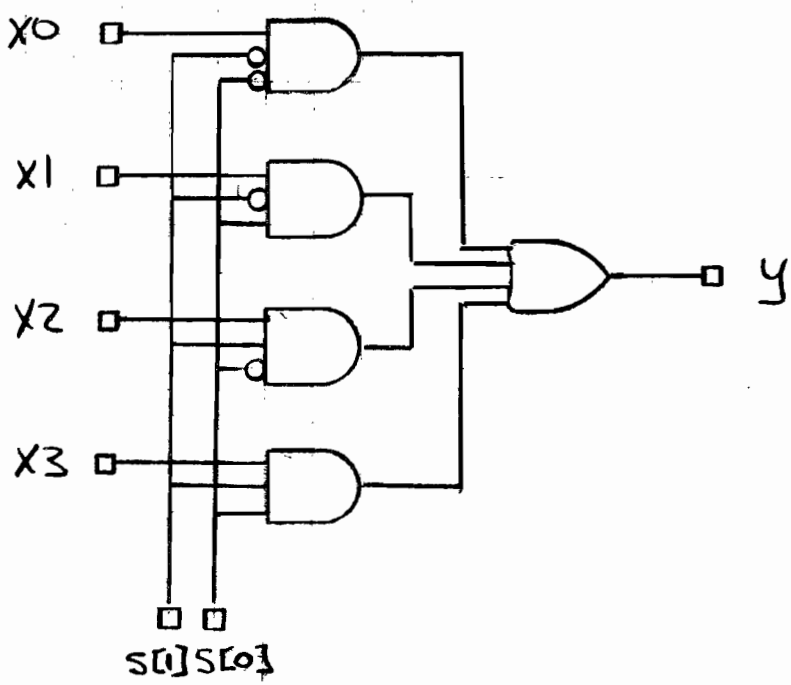
Operation

The output, y, is connected to one of the four inputs depending on the value of the 2-bit input S. This circuit is basically a 4-pole single through switch.



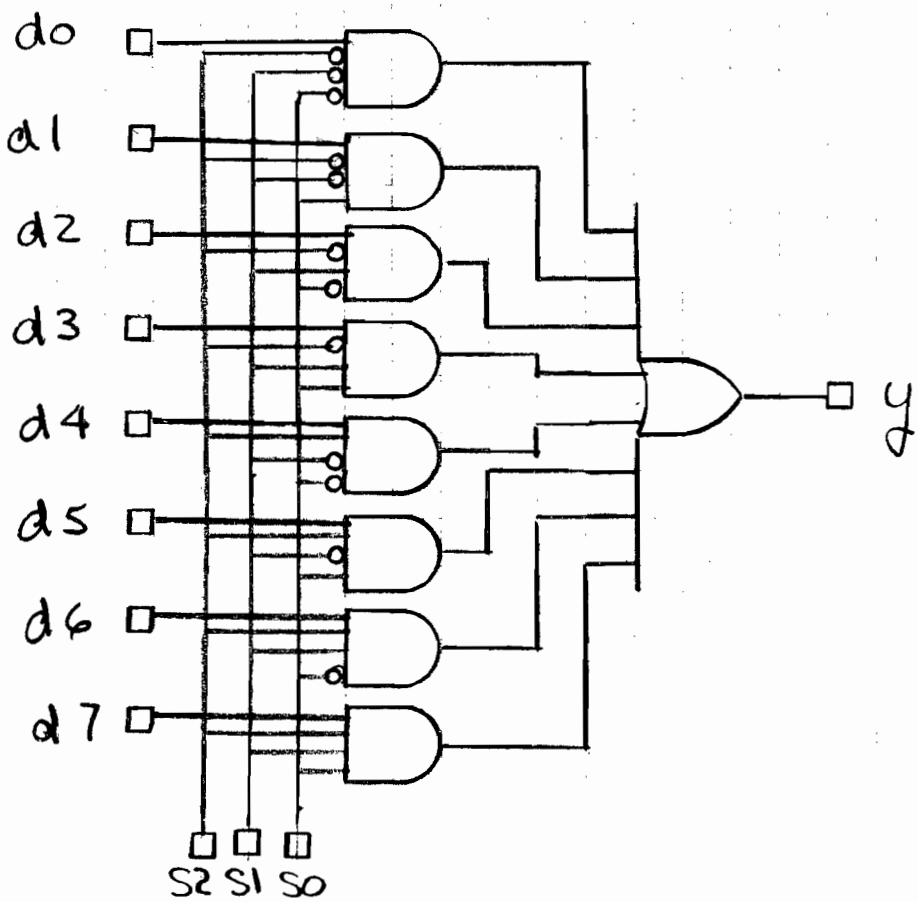
position of switch controlled by S.

Circuit that is built



A compiler translates a case statement into the hardware for a data selector and then connects the signals

```
Case ( {s2, s1, s0} )
3'b000:  y = d0;
3'b001:  y = d1;
3'b010:  y = d2;
3'b011:  y = d3;
3'b100:  y = d4;
3'b101:  y = d5;
3'b110:  y = d6;
3'b111:  y = d7;
end case
```



Structure of the Case Statement

```
case (expression)
    alternative 1:  statement 1 ;
    alternative 2 :  statement 2 ;
        :                :
    default:            default statement.
endcase
```

The expression is evaluated to a vector. The connection to the output is decided by the statement corresponding to the alternative that matches the evaluated expression. If two alternatives match (because of two repeated alternatives or because of don't cares in the alternative) the connection is made using statement associated with the matching alternative that is closest to the top of the list. If no alternative is found then the connection is established based on the default statement.

NB If the default statement is not used and there are missing alternatives then for those alternatives the statement

"out_variable = out_variable"

is implied. This will build some sort of latch.

If/else statements can be used as statement_1, statement_2, etc. Also another case statement could be used for statement_1, statement_2 etc.

NOTE: The case alternatives can be specified using any base, i.e. 3'd0, 3'd1, etc or. 4'H0, 4'H1, etc

Case Statements with "don't cares"

Should use casex if "don't cares" are used in case alternatives

4 bits

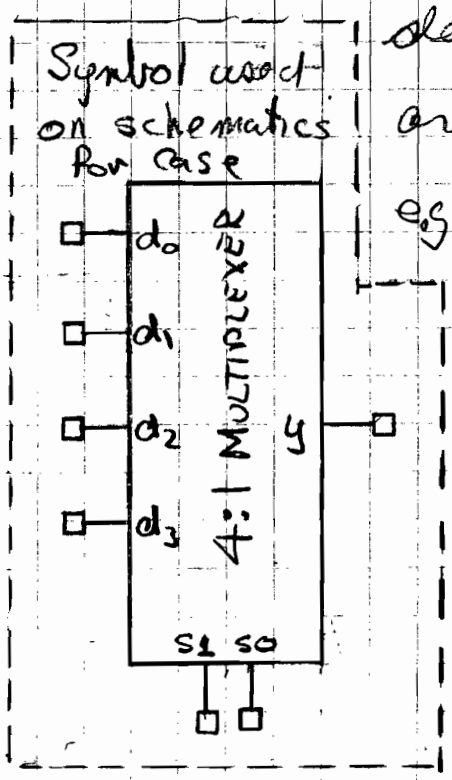
```
casex (select)
  4'b1xxx :   y = data_0;
  4'bx1xx :   y = data_1;
  4'bxx1x :   y = data_2;
  default :   y = 8'bxxxxxxxx;
endcase
```

but most compilers allow don't cares with the case statement.

not good practice use 8'b0

NOTE: the functionality of the circuit depends on order that case items are listed

e.g. As is, a value of 1100 for select steers data_0 to y. However if the first two case items were listed in reverse order then data_1 would be steered to y.



Approach that compiler will probably take to solving the case statement below:

```
case ({s2, s1, s0})
   2'bx11:  y = z;
   2'bxx1:  y = ~z;
   2'b00x:  y = 1'b1;
   default: y = 1'b0;
endcase
```

The compiler would generate a full case statement without any don't cares by expanding the alternatives starting with the first one. Conflicts generated from later alternatives are ignored.

```
3'b000:  y = 1'b1;  from 3rd
3'b001:  y = ~z;    from 2nd (3rd ignored)
3'b010:  y = 1'b0;  from default
3'b011:  y = z;     from 1st alternative (2nd ignored)
3'b100:  y = 1'b0;  from default
3'b101:  y = ~z     from 2nd
3'b110:  y = 1'b0;  from default
3'b111:  y = z;     from 1st alternative (2nd ignored)
```

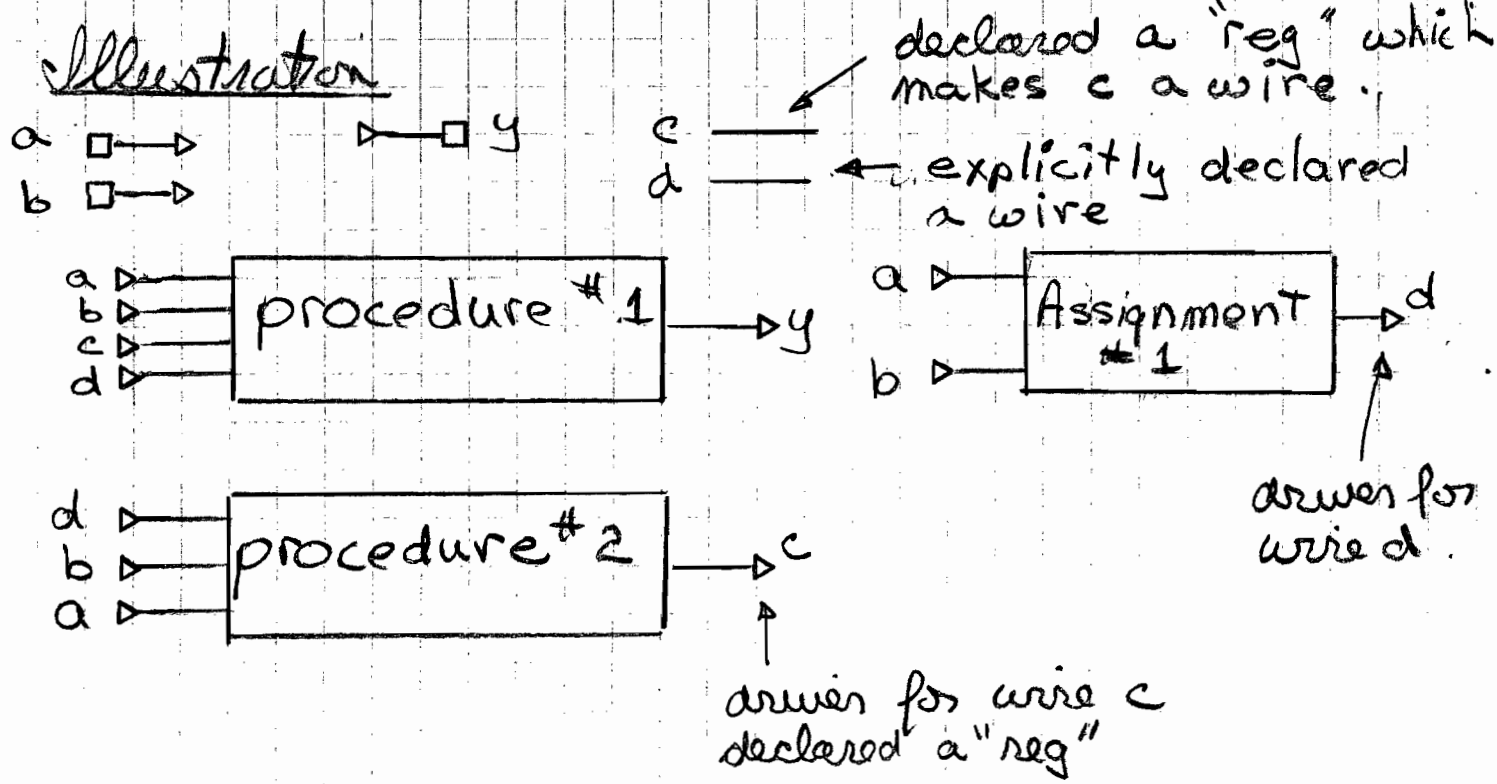
NB: Using don't cares in assignments is bad practice

NB: Using don't cares in case alternatives is not recommended. Occasionally it will clarify the intent (making the description more logical)

Gluing Procedures and Assignments Together

The compiler constructs circuits for each procedure and assignment statement independently. The inputs and outputs for each of the circuits are then connected to wires. Input are assumed to be connected to wires of the same name so there is no need to explicitly declare a wire. Similarily variables of type "reg" are assumed to be connected to a wire with the same name.

Illustration



Example.

Give the students 5 minutes to construct the circuit for module "useless_circuit".

```
module useless_circuit (
input  a, b,
output reg y );
 reg c;
 wire d;
always @ *
  case ({a, b})
  2'b 1x :  y = c;
  2'b x0 :  y = d;
  default:  y = 1'b0;
  end case
always @ *
  if (a == b)   c = d;
  else          c = ~d;
assign  d = a ^ b;
endmodule
```

procedure 1

procedure 2

assign statement

Important rules

1. If the "else" part of an if-else statement is omitted then the compiler inserts the statement

   else Q = Q;

   and prints a warning to the effect a "else Q = Q" was inserted and another warning that a combinational circuit contains feedback and the circuit may result in a latch.

2. If a case item is missing and there is no default listed then the compiler inserts a Q = Q statement and prints the feedback and latch warnings.

3. An output can not be assigned values in two different always procedures.
Each output must be completely defined in one always statement.

4. If an output is defined more than once in a single always procedure the compiler will use the last one.

e.g.:

```
begin
    if (x==a)  y = 4'b1010;   <- this statement is ignored
    if (x==b)  y = 4'b0011;
    else       y = 4'b0000;
end
```

NOTE TO SELF - Feb 2003
= put questions on an assignment for the students to experience these.

Things to Watch for.

1) setting an output variable in two separate always statements — the compiler will flag this

2) Setting an output variable in two or more separate places in the same always statement — the compiler will probably not flag this. The rule is that the last statement is synthesized.

Examples:

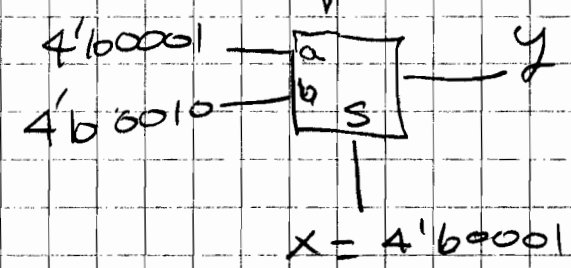
```
input [3:0] x; output [3:0] y.

always @ x
begin
        y = 4'b0101;
        if (x == 4'b0001)  y = 4'b0001;
        else  y = 4'b0010;
end
```

This statement is ignored by synthesizer.

The instructions to the synthesizer are

- wire y to 4'b0101
- also wire y to mux



The synthesizer can not do both so only implements the last one.

<u>Example 2</u>

```
input [3:0] x;   output [3:0] y;

always @ x
begin
        y = 4'b0101;
        if (x == 4b0001) y = 4'b0010;
end
```

bad practice

This is really two statements but the newer compilers will recognize this as

```
if (x == 4'b0001) y = 4'b0010;
else y = 4'b0101;
```

Example 3

```
always @ x
begin
    if(x == 4'b0001) y = 4'b0001;
    if(x == 4'b0010) y = 4'b0011;
    if (x == 4'b1000) y = 4'b0100;
    else y = 4'b0000;
end
```

only this statement is used

In the above there are three places that y is connected. Suppose that x = 4'b0001,

a) The first statement wants to change y to 4'b0001.

b) The second statement, through an implied "else y = y" wants y to retain it value

c) The third statement wants to change y to 4'b0000.

<u>Example</u> (will compile properly)

```
always @ x
begin
        if (x ==4'b0000 ) y = 4'b0001;
        else if (x == 4'b0001) y = 4'b0011;
        else if (x == 4'b0010) y = 4'b0111;
        else y = 4'b1111;
end
```

do not need begin-end here as because there is only one statement

In the above, the instructions are clear. For each value of x, y is to be assigned a unique value.

Nesting "if's" as above will compile properly – But compilers work better with case statements and so case statements are preferred

For loops & integers

- integers are used for indices in loops.

NB - For loops are not used like if/else or case statements.
- They are "precompiled". They are processed before the compile begins.
- They are expanded into synthesizable verilog HDL prior to compilation

** - They are used to save writing. They allow certain algorithms to be expressed in a compact way.

Structure of a for loop

variable must be of type integer or type reg

no semicolon

```
integer i;

for ( i = 0; i <= 6; i = i+2)
        Statement;
```

initializes index

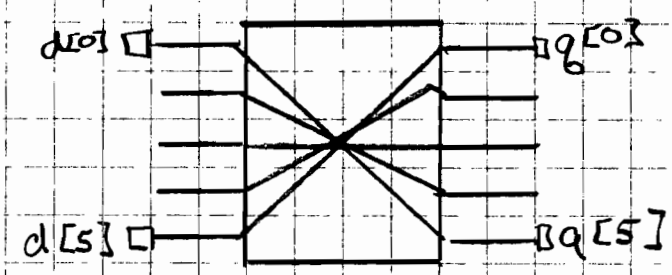
termination condition

statement that determines how the index variable is changed normally incremented or decremented

single statement, to use a block of statements need a begin - end.

NOTE: The loop variable (the index) can be of type "integer" or "reg". If it is of type reg, it must be a positive integer and is limited in size by the "reg" declaration

<u>Example</u>. a circuit that flips bits



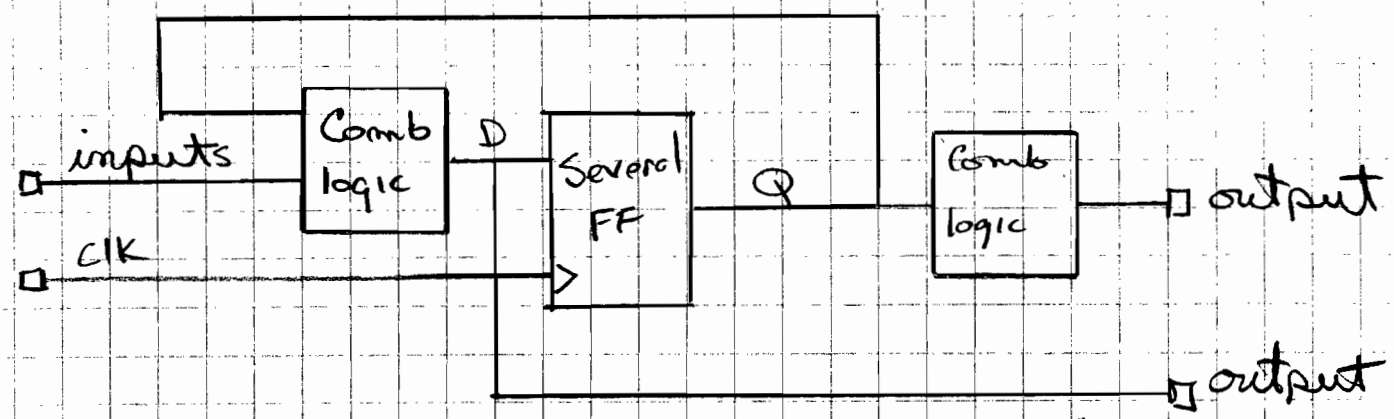

```
module bit_flipper ( data_in, data_out)
input [15:0] data_in;
output [15:0] data_out;
reg [15:0] data_out;
integer i;
always @ ( data_in)
    for ( i=0; i<=15; i=i+1)
        data_out [15-i] = data_in [i]

endmodule
```

Sequential Logic

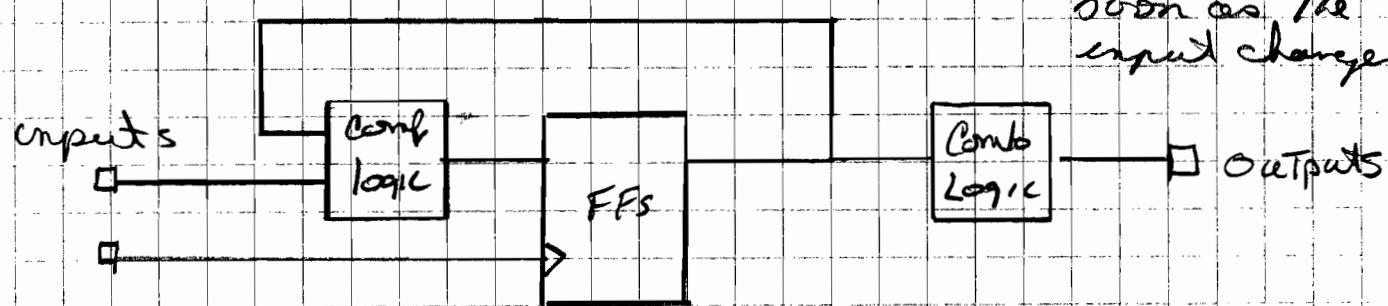
- Driven by a clock
- always contains flip/flops.
- feedback from output to input of flip/flops is very useful. It does not cause uncontrolable oscillation because of sampling action of the clock.
- feedback causes the output to change on the next clock edge. The value of the output after the next edge depends on the current value.
- every clock edge changes the output, thus generating a sequence, which is why it is called sequential logic

General sequencial Machine

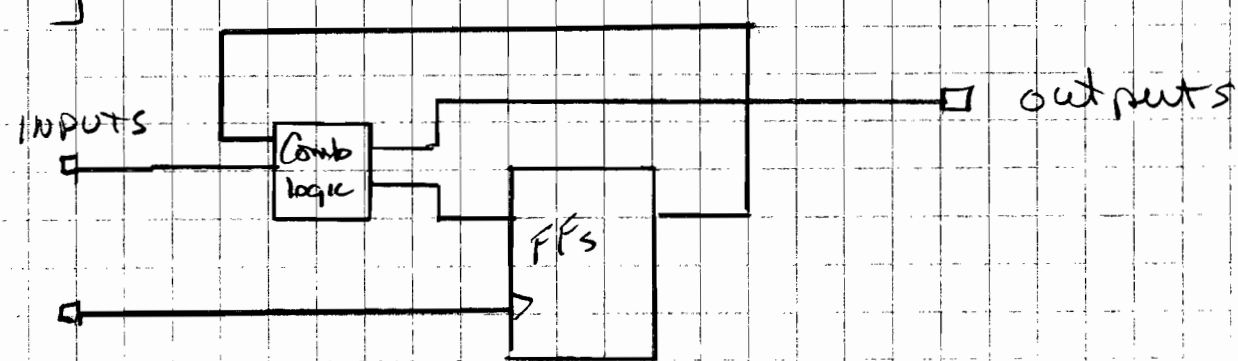


these outputs change as soon as the input changes

Moore Machine



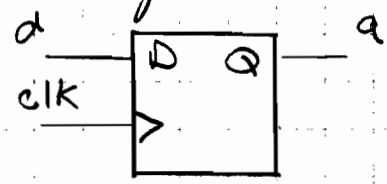
Mealy Machine



Behavioral descriptions of Sequential Circuits.

```
reg q;                                  could be keyword
                                        "negedge"
always @ (posedge clk)
        q = d;
```

Such a procedure generates a D flip flop



When a signal is used for a clock, in this case "clk" is used as a clock, the compiler issues a warning "Found pins functioning as undefined clocks and/or memory enables". The circuit will compile properly but the compiler would like the engineer to provide information about the clock. This is done by selecting "Assignments". Then expanding "Timing analysis settings" to get the submenu. Select "Classic timing Analyzer Settings". In the window to the right click "Individual clocks" and select "new"

In this window you have to associate the clock node used in the design with another name for the clock. The clock node in the design is called "clk" (i.e. always @ (posedge clk)). You can name the clock input anything, say clock_1. Then select the node that is used for the clock i.e. select clk. The timing analyser need to know how fast your circuit should run so that it can print a warning if the propagation delays through the logic cause the signal to arrive at the "d" input to late to meet the set up time. In this class we will use "1MHz" for the "Required Max" setting.

Examples of Sequential Circuits

```
always @ (posedge clk)
    q = d;
```

d-Flip/Flop
DFF primitive
or DFFE primitive

```
always @ (posedge clk)
    if (clk_enable) q = d;
    else q = q;
```

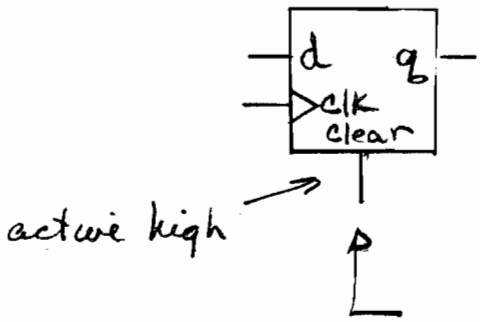
d-Flip/Flop with clock enable
DFFE primitive

```
always @ (posedge clk)
    if (clear == 1'b1) count = 8'b0;
    else count = count + 1'b1;
```

counter with synchronous clear

```
always @ (posedge clk or posedge clear)
    if (clear == 1'b1) q = 1'b0;
    else q = d;
```

d-flip/flop with asynchronous clear
DFF or DFFE primitive



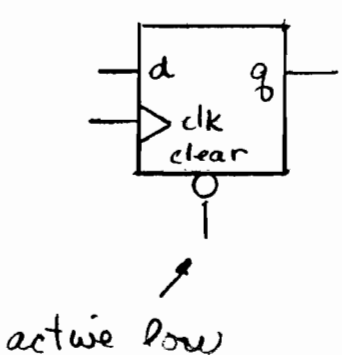
clear is active high :.

```
always @(posedge clk or posedge clear)
if (clear == 1'b1)   q = 1'b0;
else                 q = d;
```

Asyncronous clear :. need to have add "clear" to sensitivity list and test for clear in the first statement. Since clear is active high the inclusion in the sensitivity list must be "posedge clear" and the test must be if (clear == 1'b1).

If the test is "if (clear == 1'b0)" when "posedge clear" is in the sensitivity list, then the compiler prints an error message.

clear is active low




```
always @(posedge clk or negedge clear)
if (clear == 1'b0)   q = 1'b0
else                 q = d;
```

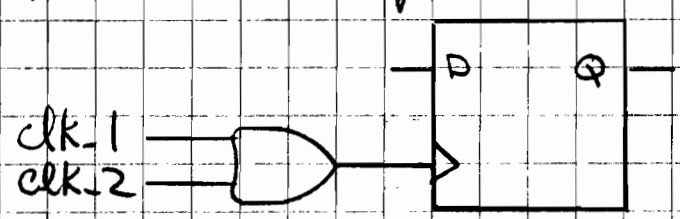
NOTE If sensitivity list has two edge sensitive signals then one must be tested in always statement and the test should be for the assertion after the edge.

for posedge - test for signal == 1;
for negedge - test for signal == 0;

Example of unbuildable circuit

```
always @ (posedge clk 1 or posedge clk 2)
        q = d;
```

a flip flop only has one clock input so only choice is



but if clk_1 is high when clk_2 ↑ the posedge will be masked.

An example of something that was unbuildable until Dual Edge flip/flops were used (to save power)

```
reg q;
always @ (posedge clk or negedge clk)
        q = d;
```

The instruction is clear, update q at both the positive and negative edge of the clock. However, until the dual edge clocked flip flop was implemented, it made no sense to support this.

Now back to example

```
always @ (posedge clk 1 or posedge clk 2)
        q = d;
```

Even though the circuit is unbuildable, the intention is very clear, transfer d to q at the instant either clk 1 or clk 2 has a positive edge.

The problem is that technology can't support it.

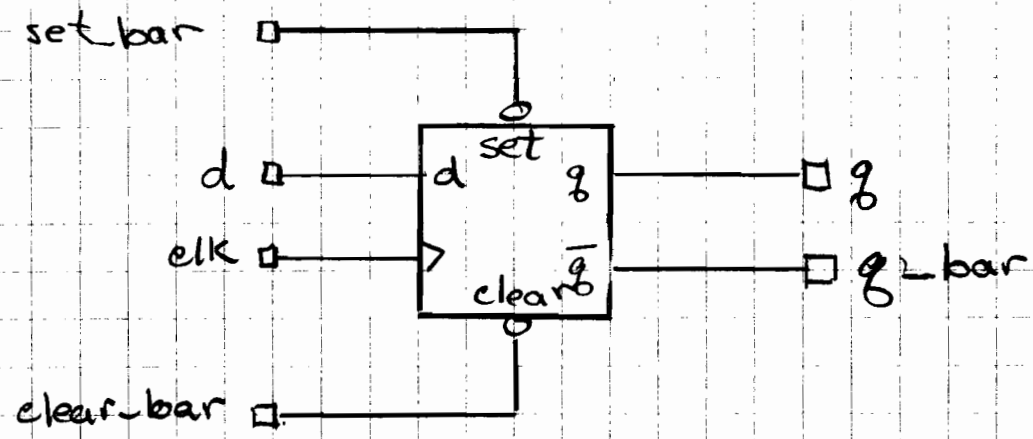
<u>Example</u>.

Write a Verilog HDL of a 3 bit counter that is preloadable (asynchronous) and has a countenable.

```
always @ (posedge clk or posedge preload)
    if (preload == 1'b1)  count = data;
    else if (count_enable == 1'b1) count = count + 1'b1;
    else  count = count;
```

NOTE: This feedback will not generate a warning as it only occurs on a clock edge

Building asynchronously loaded D flip flops



Truth Table.

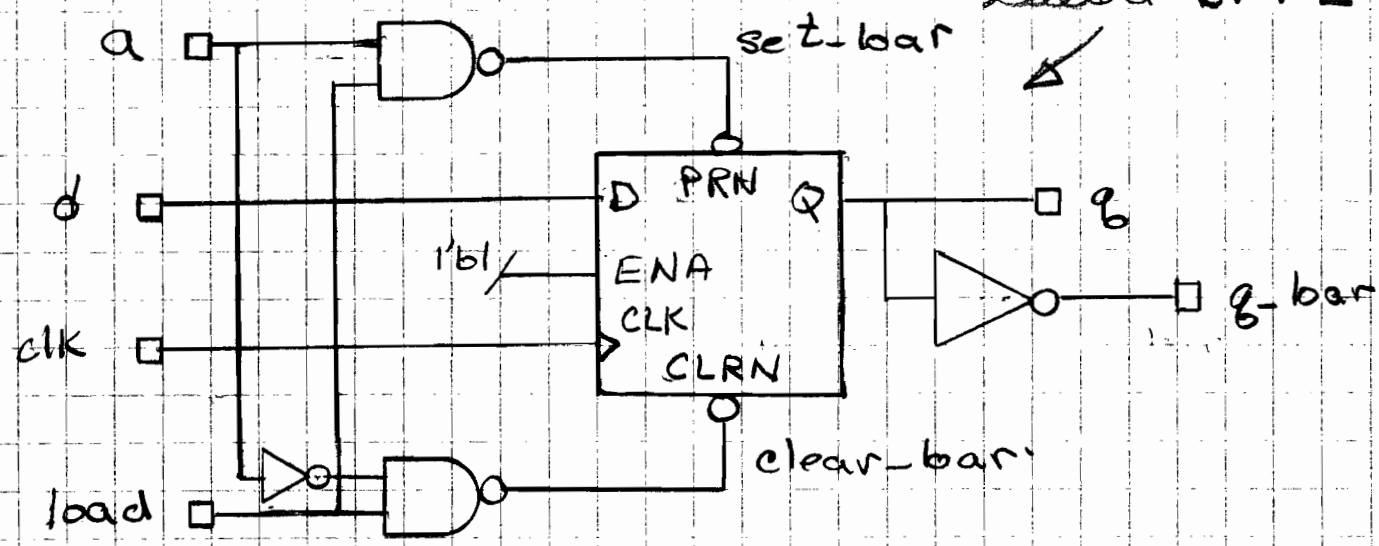
$\overline{set}$	$\overline{clear}$	clk	q	q_bar
0	1	X	1	0
1	0	X	0	1
1	1	↑	d	$\overline{d}$
0	0	X	undefined	

↳ Could be.
Q = Q_bar = 1
Q = Q_bar = 0
Q = 0, Q_bar = 1
Q = 1, Q_bar = 0

<u>NB</u>: It is not possible to implement the "undefined" state in verilog.
∴ A particular state must be chosen.
The compiler will insert logic to make sure this happens.

For example

This f/f is an Altera primitive called DFFE




```
module   async_load ( clk, a, load, d.
                      q, q_bar );
input  clk, a, load, d;
output q, q_bar;
wire   set_bar, clear_bar;
assign q_bar = ~ q;
assign set_bar = ~( a & load);
assign clear_bar = ~(~a & load);
DFFE   flip_flop_1 ( .D(d),
                     .CLK(clk),
                     .ENA(1'b1),   ← could omitt this line
                     .Q(q),
                     .PRN (set_bar),
                     .CLRN (clear_bar) )
end module
```

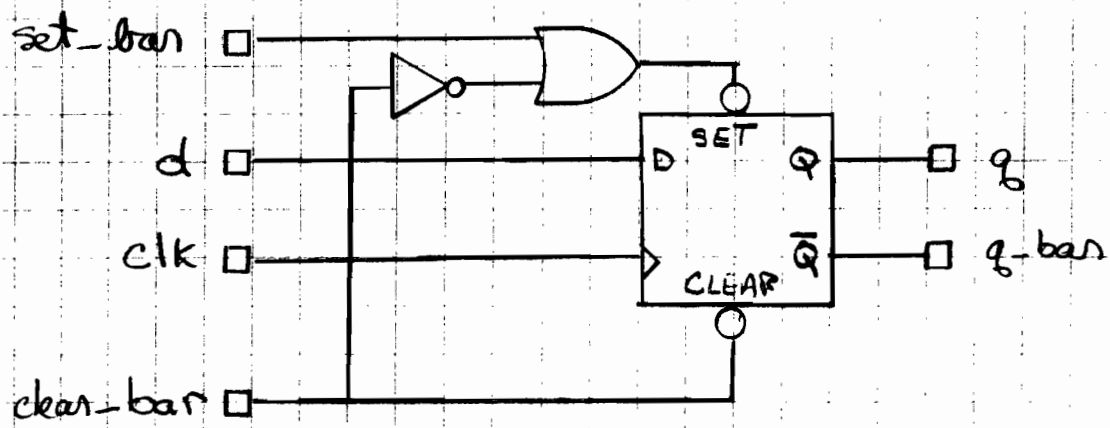
```
Always @(posedge clk or negedge set_bar
                 or negedge clear_bar)

  if (clear_bar == 1'b0)
      begin  q = 1'b0; q_bar = 1'b1; end.

else    if (set_bar == 1'b0)
        begin q = 1'b1; q_bar = 1'b0; end.

        else
        begin q = d;  q_bar = ~d; end.
```

This builds a "clear over rides set" flip flop. It generates the following hardware.



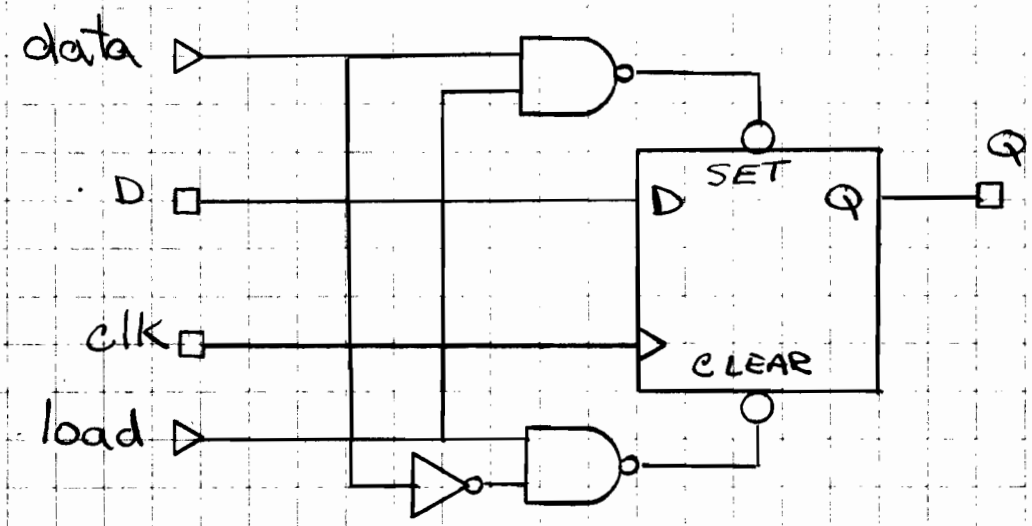
Preloadable flip/flop with 3 preloads.

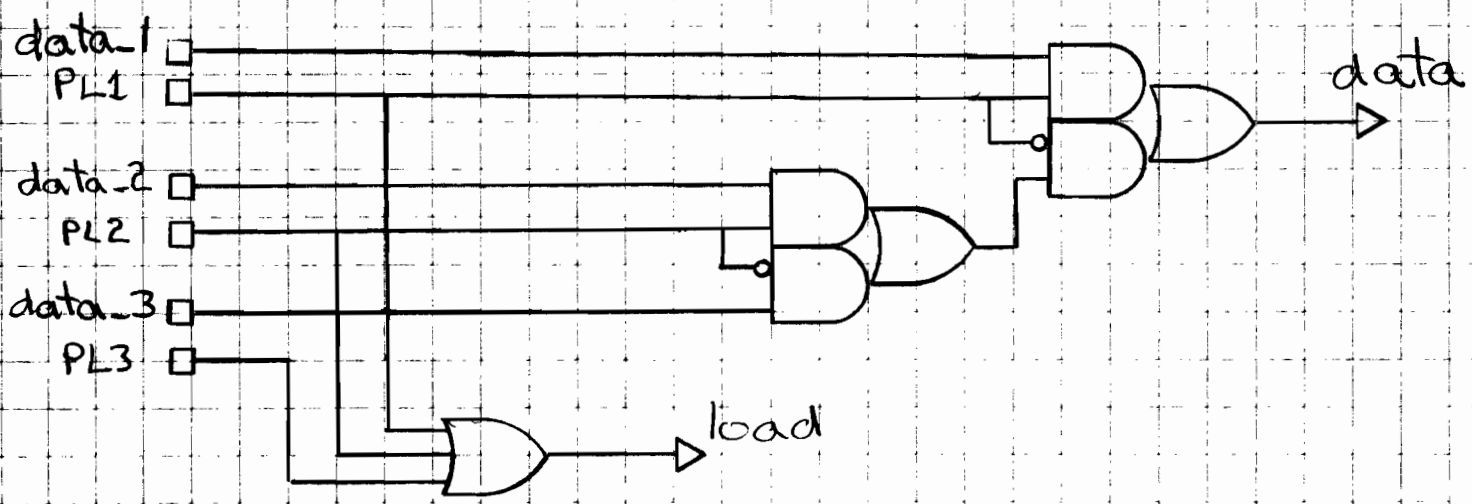
preload data_1 if PL1 is high
preload data_2 if PL2 is high
preload data_3 if PL3 is high

PL1 overrides both PL2 and PL3.
PL2 overrides PL3

```
always @ ( posedge clk
           or posedge PL1
           or posedge PL2
           or posedge PL3 )
  if ( PL1 == 1'b1 )  Q = data_1;
  else if (PL2 == 1'b1) Q = data_2;
       else if (PL3 == 1'b1) Q = data_3;
            else  Q = D; // PL1, PL2, PL3
                         // are all low so
                         // the "positive edge"
                         // is from clk.
```

Template used by synthesizer

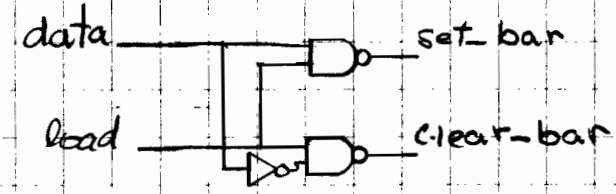




The synthesizer's algorithm is

1. If there is only one "posedge" in the sensitivity list, then it is the clock

2. If there are two or more "posedge" declarations in the sensitivity list then an asynchronous set, clear, or load is required. Use the data-load template.



For a set, data = 1'b1,
For a clear, data = 1'b0,
For a load, data = variable.

3. For each "posedge" declaration after the second, i.e. for 3rd, 4th etc, add a 2:1 mux to data and an "or gate" to "or" the loads.

Flip/Flops with only an asynchronous clear

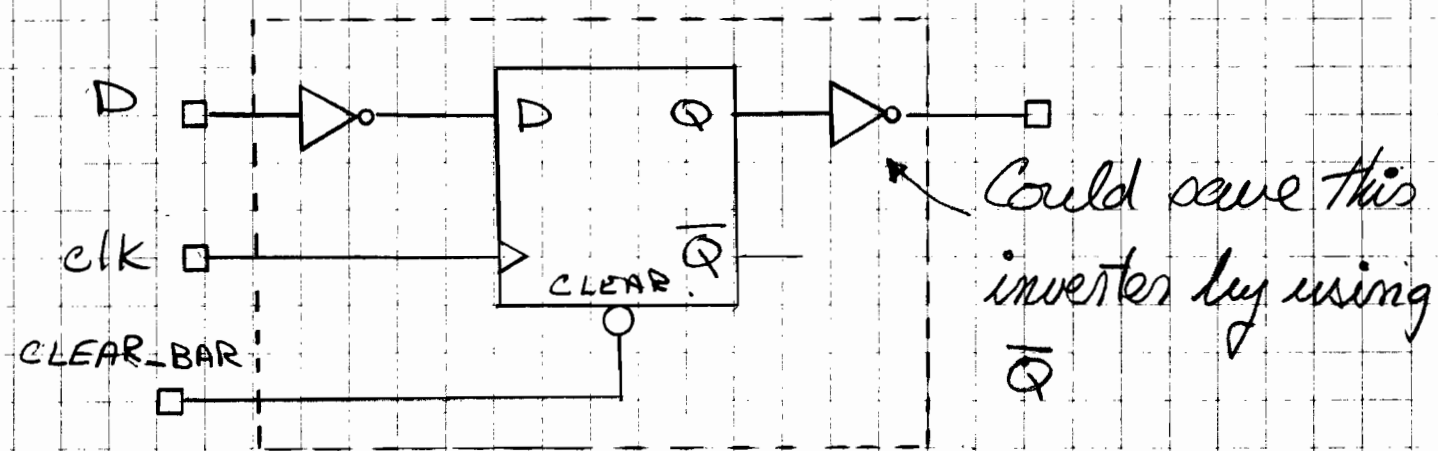
Most of the newer FPGAs use flip flop that only have a clear. The reasons for this are.

1. Asynchrous loads are very rarely used.
2. The flip/flops often need to be set or cleared at startup and most of them are cleared
3. *** The extra tracking required to use both set and clear input is quite costly and is rarely used.

Reason #3 is the main reason.

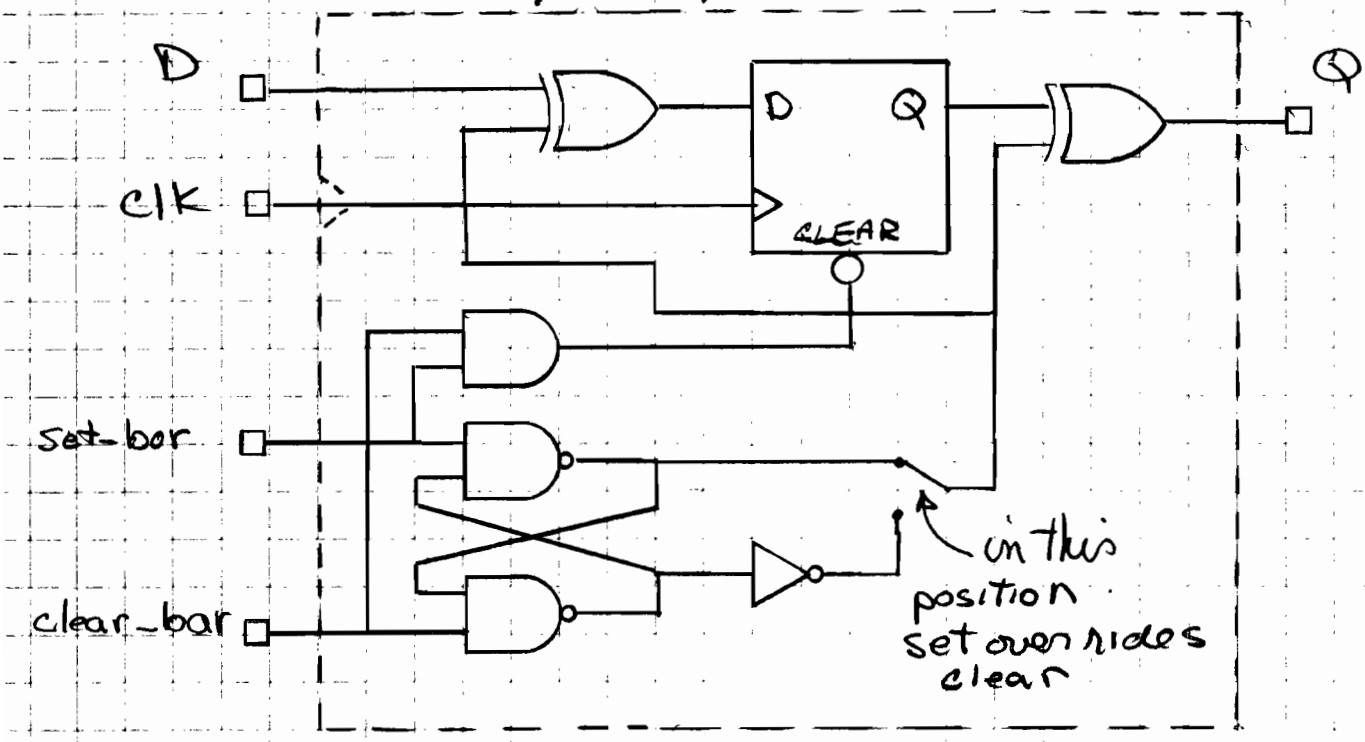
Asynchronous "set" using a FF with only an asynchronous clear.

A flip flop with an asynchronous "clear" input is converted to a flip/flop with an asynchronous "set" input by placing inverters before D and after Q.



Whenever the synthesizer does this it prints a warning. The key phase in the warning is "not push back". Placing the inverters before D and after Q is referred to as "not push back".

Converting an asynchronous clear flip flop to an asynchronous set-clear flip flop



Synthesizer algorith for asynchrous set, clear, load for FF with only asynchronous clear

1. If a variable is loaded, then build a FF with asynchronous clear and set inputs and use the algorithm described for that type of FF.

2. If a constant is loaded, then less circuitry is required. Use an inverters before and after the input and output for an asynchonous load of 1'b1 (i.e. a set) and just use the F/F as is for an asynchronous load of 1'b0

Synchronous Loads

Most applications require that a flip/flop be loaded on a clock edge. A Verilog description of such a circuit would be

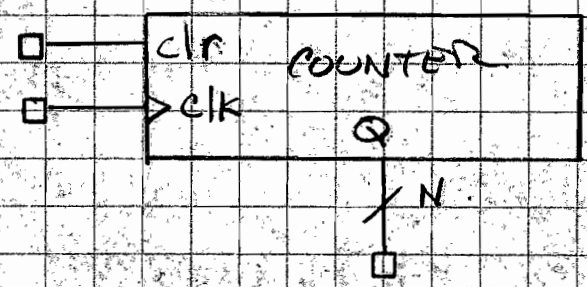
```
always @ (posedge clk)
if (load == 1'b1)
        y = 4'b1010;
else    y = y + 4'b0001;
```

The circuit built by the synthesizer does not use the asynchronous "set" or "clear" inputs.

Examples

1. Synchronous counter with synchronous clear.
2. single f/f that divides frequency by 2. (augment with "freeze" control) (XOR with D input)
3. Ripple Counter
4. Counter that counts to 9 and then starts at zero
5. 3 f/f ring tail counter built with 3 always statements
6. 3 bit synchronous counter built from 3 always statements.

Synchronous counter with synchronous clear

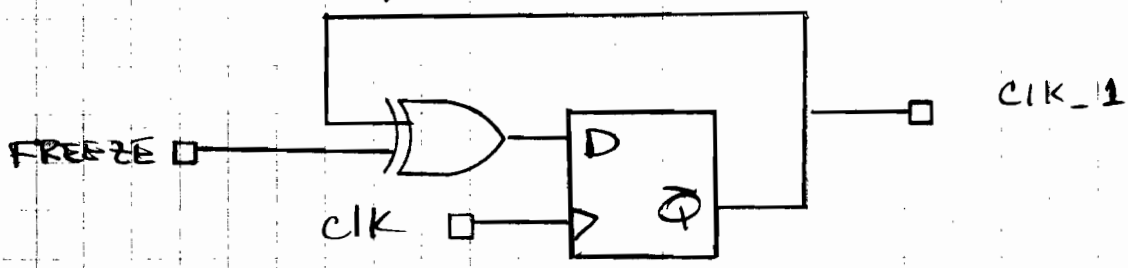



```
module counter_sync_clr (clk, clr, Q);
input clk, clr;
output [7:0] Q;
reg    [7:0] Q;
always @(posedge clk)
  if (clr == 1'b1)  Q = 8'b0;
  else  Q = Q + 1'b1;
endmodule
```

NOTE: if the "else" statement was Q = Q + 1; the compiler would expand 1 to 32 bits by replacing it with 32'b1. Q would be expanded to 32 bits, the addition would take place and then result would be truncated to fit into Q, which is 8 wires. The hardware that is built is the same, but the compiler generates a "truncation" warning.

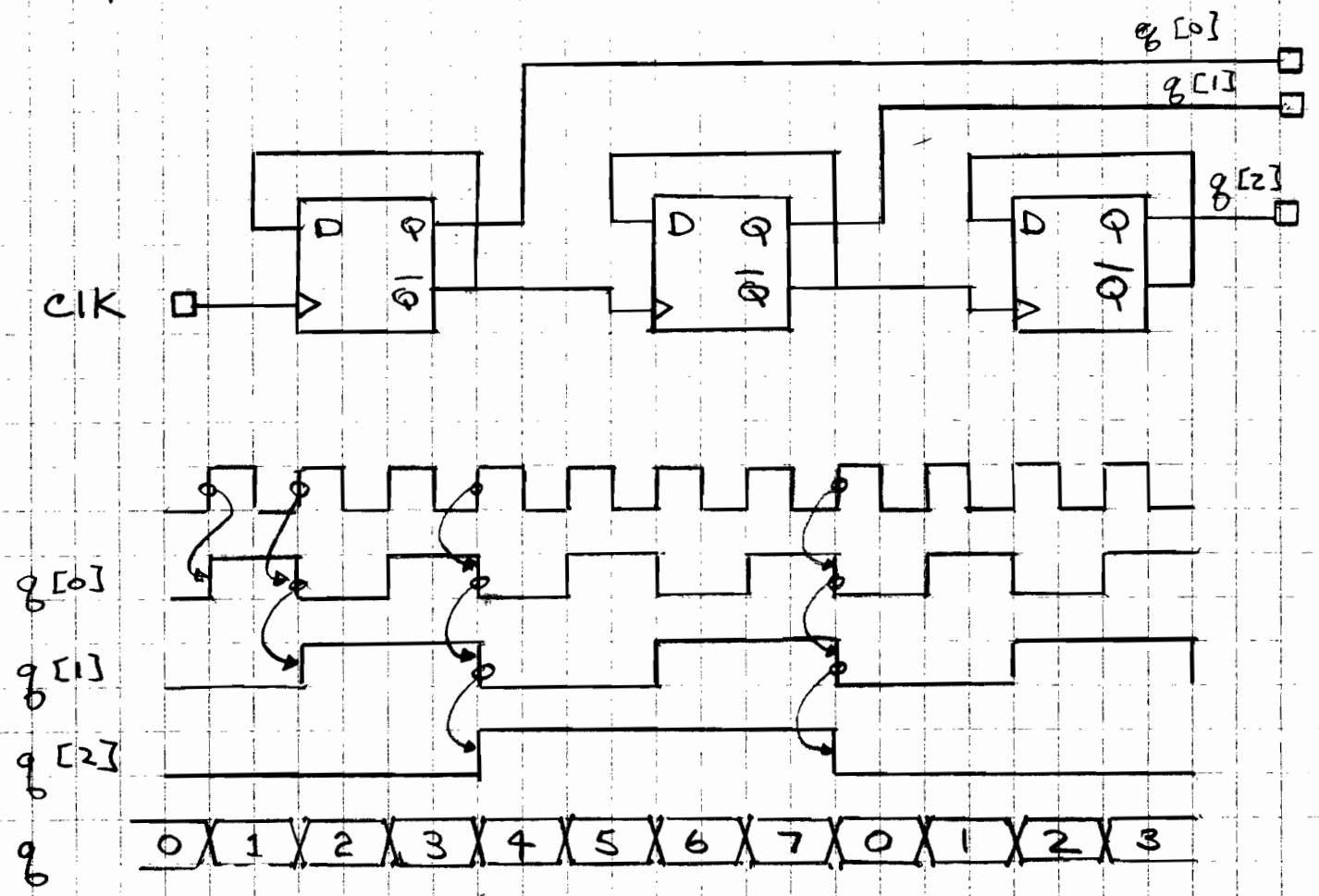
61 c)

Frequency Divider




```
always @ (posedge clk)
    if (FREEZE == 1'b1)   clk_1 = clk_1;
    else                  clk_1 = ~ clk_1;
```

Ripple Counter (rarely if ever used in FPGAs)




```
reg [2:0] q;
always @ (posedge clk) q[0] = ~q[0];
always @ (negedge q[0]) q[1] = ~q[1];
always @ (negedge q[1]) q[2] = ~q[2];
```

0 to 9 Counter

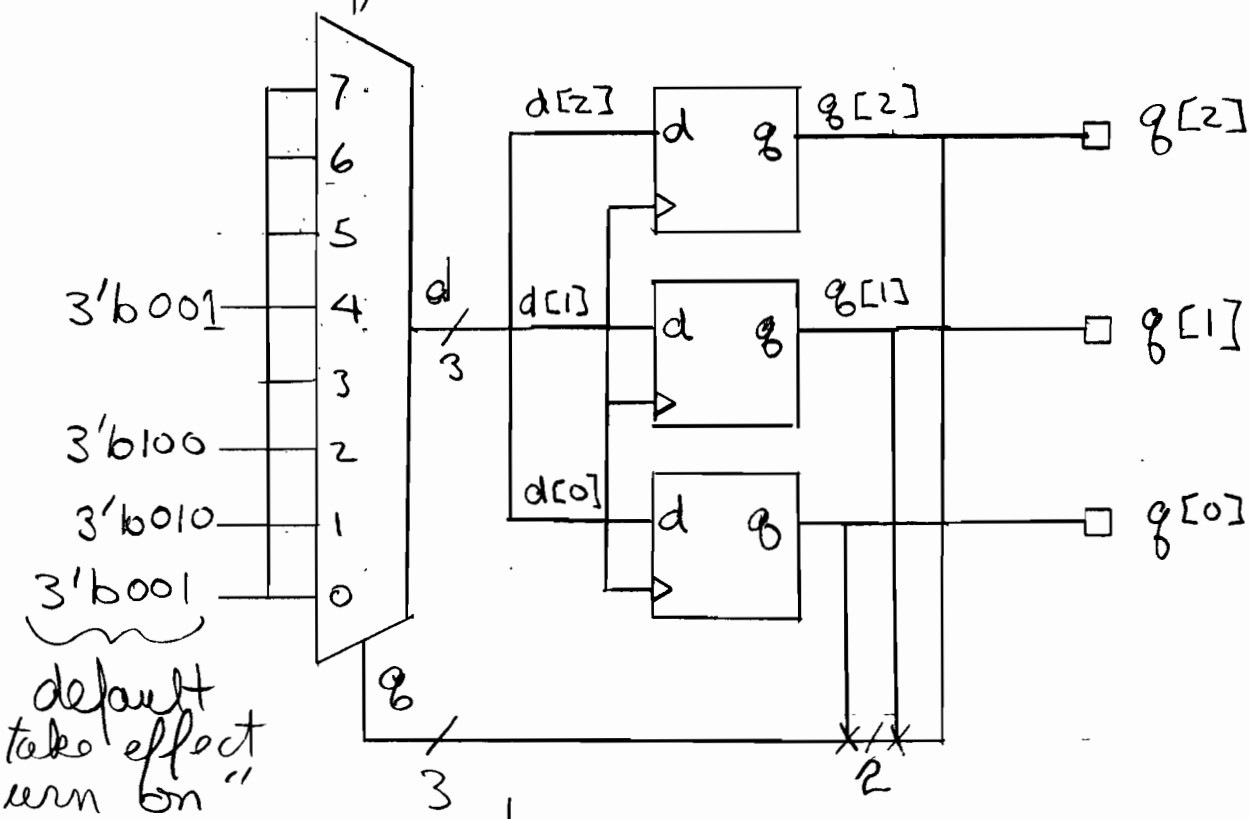
```
always @ (posedge clk)
    if (count == 4'd9) count = 4'd0;
    else count = count + 1'b1;
```

← This circuit could be 4'd10 at power up and then would take 6 clock cycles to get to 0.

```
always @ (posedge clk)
    if (count > 4'd8) count = 4'd0;
    else              count = count + 1'b1;
```

In the later, more hardware is required to build the circuit to evaluate the relation but, if the counter has value 4'd9, 4'd10, ... 4'd15 then it will change to 4'd0 on the next clock cycle.

Example Ring Counter




```
reg [2:0] q
always @ (posedge clk)
case (q)
3'b001:   q = 3'b010;
3'b010:   q = 3'b100;
3'b100:   q = 3'b001;
default:  q = 3'b001;
endcase
```

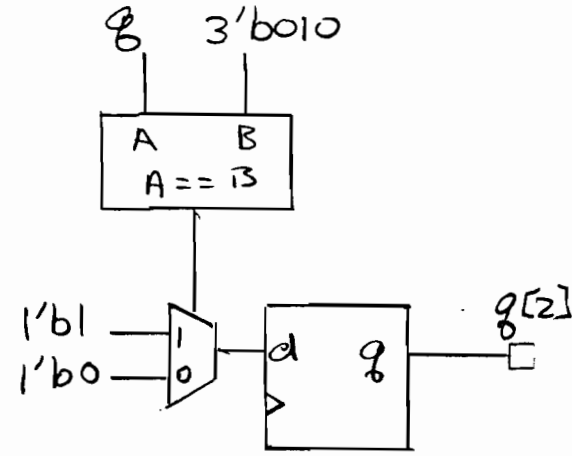
```
reg [2:0] d, q;
always @ (posedge clk)
        q = d;
always @ *
case (q)
3'b001:   d = 3'b010;
3'b010:   d = 3'b100;
3'b100:   d = 3'b001;
default:  d = 3'b001;
endcase
```

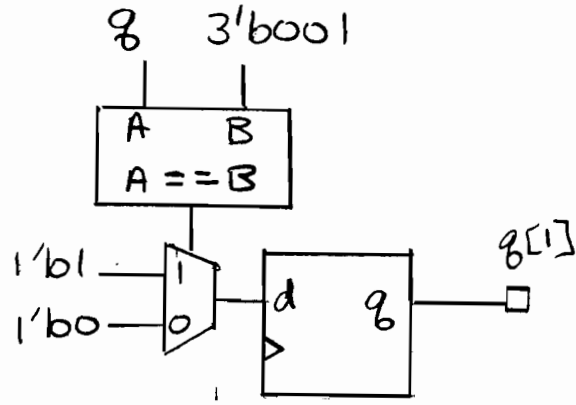
Alternate description

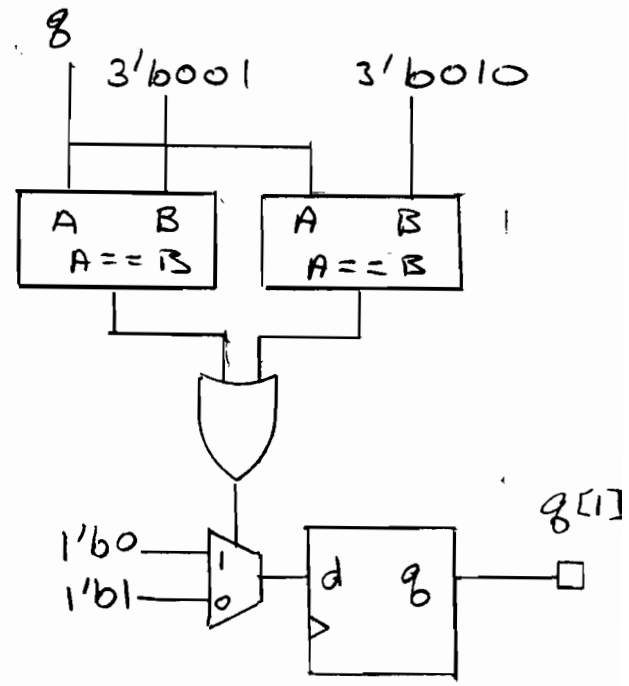
```
reg [2:0] q;
always @ (posedge clk)
if (q == 3'b010)
        q[2] = 1'b1;
else    q[2] = 1'b0;

always @ (posedge clk)
if (q == 3'b001)
        q[1] = 1'b1
else    q[1] = 1'b0

always @ (posedge clk)
if (q == 3'b001) ||
   q == 3'b010)
        q[0] = 1'b0;
else
        q[0] = 1'b1;
```

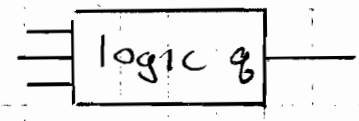




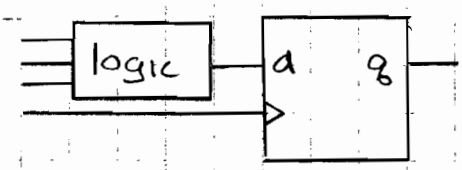


61.1

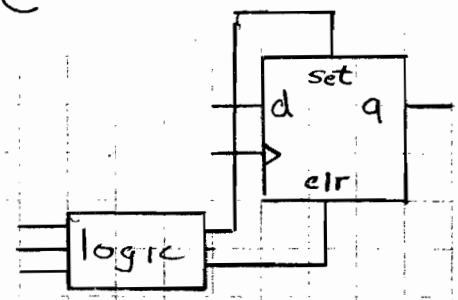
A



B



C



1.

```
always @(posedge clk)
if(clear==1'b1)  q = 1'b1;
else  q = d;
```

2.

```
always @ ( posedge clk or
           posedge clear)
if (clear == 1'b1)  q = 1'b0;
else  q = d;
```

3

```
always @ (posedge clk or
          posedge load)
if (load == 1'b1) q = d1;
else  q = d;
```

4

```
always @ *
if (clear == 1'b1)  q = 1'b1;
else  q = d;
```

5

```
always @ (posedge clk)
if (load == 1'b1) q = d1;
else  q = d
```

6

```
always @ *
if (load == 1'b1)  q = d1;
else  q = d;
```

PROCEDURE #	1	2	3	4	5	6
MATCHING CIRCUIT						

ANSWER
1B, 2C, 3C
4A, 5B, 6A

61.1 a)

Please check the appropriate boxes with a ✓

	Has feed back from output to 1 input	Builds logic in front of D input to DFF	Could use the enable input on DFF	Use DFF with set/clear inputs
always @(posedge clk) if (time == 3'b010) data = data-path-1 else data = data-path-2				

always @(posedge clk or posedge clear)
if (clear == 1) counter = 8'b0;
else if (set == 1)
counter = 8'HFF;
else counter = counter + 8'H01;				

always @ (posedge clk or posedge load)
if (load == 1'b1) timer = 16'd5760;
else if (pause == 1'b1)
timer = timer;
else timer = timer - 16'd1;				

always @ (posedge clk) case (s); 2'd0: state = state_2; 2'd1: state = state; 2'd2: state = state_1; 2'd3: state = 6'H37; end case				

answer

Feed back	front logic	enable	set/clear inputs
	✓		
✓	✓	✓	
✓	✓	✓	✓
	✓	✓	✓

<u>In class assignment.</u>

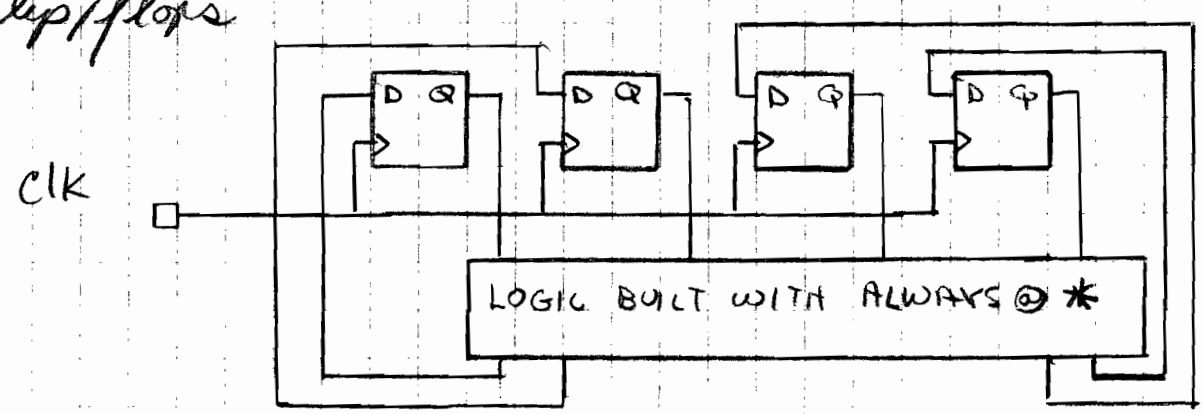
Build a 0 to 9 counter using an always @ * to build the logic and an always @ posedge clk to build the flip/flops.

In class assignment.

Design a 4-bit counter that that increments after each rising edge to generate the sequence 0, 1, 2, ... 8, 9, 0, ... Let Q be the output of the counter.

Design # 1.

Use two always statements.
a "always @ (posedge clk)" statement to make 4 d flipflops.
And an "always @ *" statement to make the logic that uses the Q outputs of the flip/flops to make the d inputs to the flip/flops



Design # 2

Use one "always @(posedge clk)" statement.

F. A. Q.

Q: Can always procedures be nested?

A: NO!

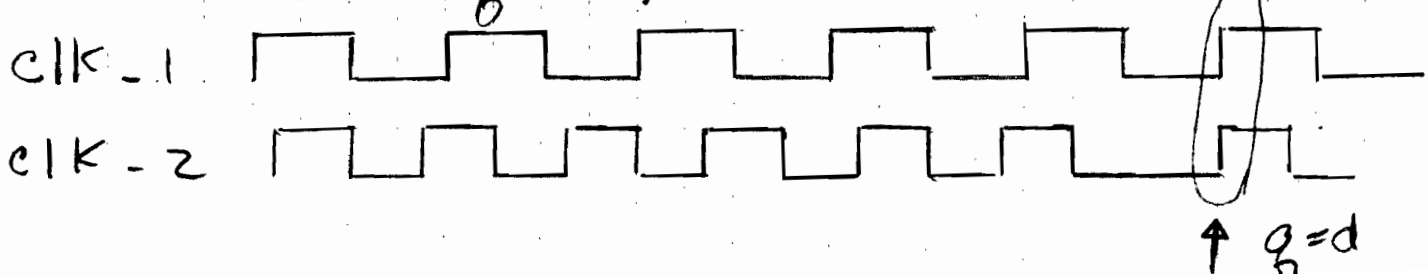
Combinational

```
always @ (a or b)
  begin
    y = a ^ b;
    always @ (a or b)
      z = a & b;
  end
```

Do not need this →

Sequential

```
always @ (posedge clk_1)
  always @ (posedge clk_2)
    q = d;
```



The description says make q = d when the clk_2 positive transition comes up. There is no hardware that can do that

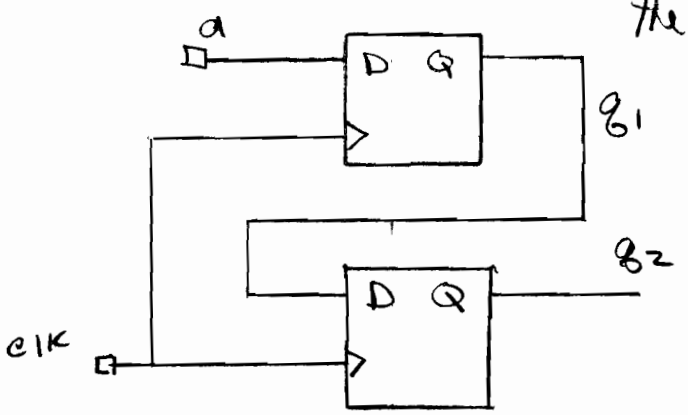
Prelude to Blocking / non-blocking assignments

```
always @ (posedge clk)
    q1 = d;

always @ (posedge clk)
    q2 = q1;
```

does not matter which always statement comes first

assigns q_2 the value of q_1 at the time of ↑ of clk.



NOTE

Block statements (begin ... end) in one always construct (more than one variable being assigned) can be rewritten as multiple always constructs with one variable being assigned in each.

BLOCKING & NONBLOCKING ASSIGNMENTS

There is only a difference between blocking and nonblocking assignments when more than one output is defined in a always @ (posedge ...) procedure.

e.g. always @ (posedge clk)
begin
$q_1 = d;$
$q_2 = q_1;$
end

= is a blocking assignment

<= is a nonblocking assignment.

For synthesizing circuit, the nonblocking assignment is used virtually all the time.

Blocked variables are updated in parallel. (i.e.) simultaneously but are updated as if they are update in the order they are listed between the "begin" "end" "end"

e.g.

```
always @ (posedge clk)
begin
        q1 = d;
        q2 = q1;   } ← blocking assignment
end
```

This instructs the compiler to update both q_1 and q_2 on the rising edge of the clock, but first figure out what q_1 is going to be after the rising edge of the clock and assign that value to q_2. In this case q_2 will be assigned d.

```
always @ (posedge clk)
begin
        q1 <= d;
        q2 <= q1;
end
```

nonblocking assignment.

This says update q_1 and q_2 simultaneously using the value of q_1 evaluated just prior to the rising edge of clk.

When blocking assignments are used the order of the statements matter. When non-blocking assignments are used the order of the statements does not matter

```
always @ (posedge clk)
begin   q1 = d;
        q2 = q1;
end
```

```
always @ (posedge clk)
begin   q2 = q1;
        q1 = d;
end
```

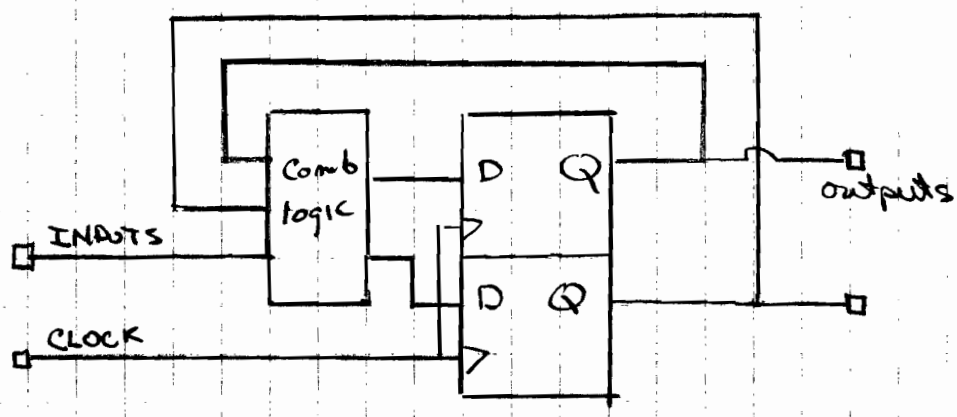
These two procedures build different circuits

```
always @ (posedge clk)
begin   q1 <= d;
        q2 <= q1;
end
```

```
always @ (posedge clk)
begin
    q2 <= q1;
    q1 <= d;
end.
```

These two procedures build the exact same circuit

A sequential circuit with synchronous outputs looks like this



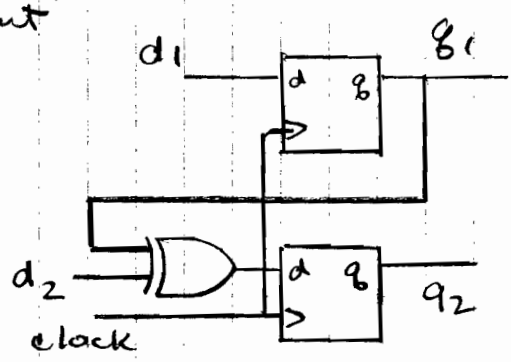
In practice all FFs are updated at the same time.

Consider an always construct that assigns two variables

```
always @ (posedge clk)
begin
    q1 <= d1 ;
    q2 <= q1 ^ d2
end
```

input

non-blocking assignment



This can be rewritten as

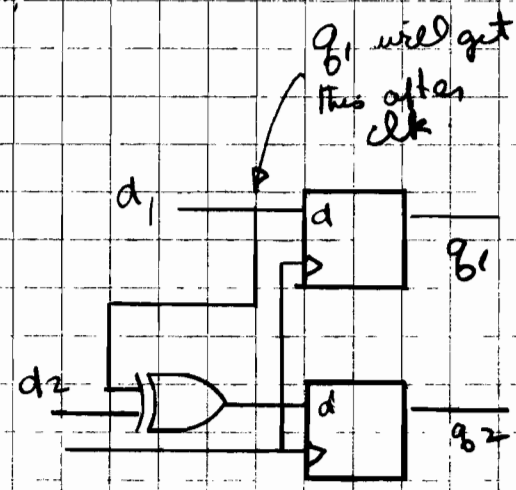
```
always @(posedge clk)  q1 = d;
always @(posedge clk)  q2 = q1 ^ d2;
```

```
always @ (posedge clk)
begin q1 = d1;
      q2 = q1 ^ d2;
end
```

blocking assignment

For a simulator the instruction is to update q_1 first and then use that value to update q_2. The hardware does not support clocking q_1 just before q_2. FF q_1 and q_2 are clocked with the same clock and therefore at the same time. The compiler reformulates the HDL with non blocking assignments.

```
always @ (posedge clk)
begin
      q1 <= d1;
      q2 <= d1 ^ d2;
end
```



This can be rewritten as

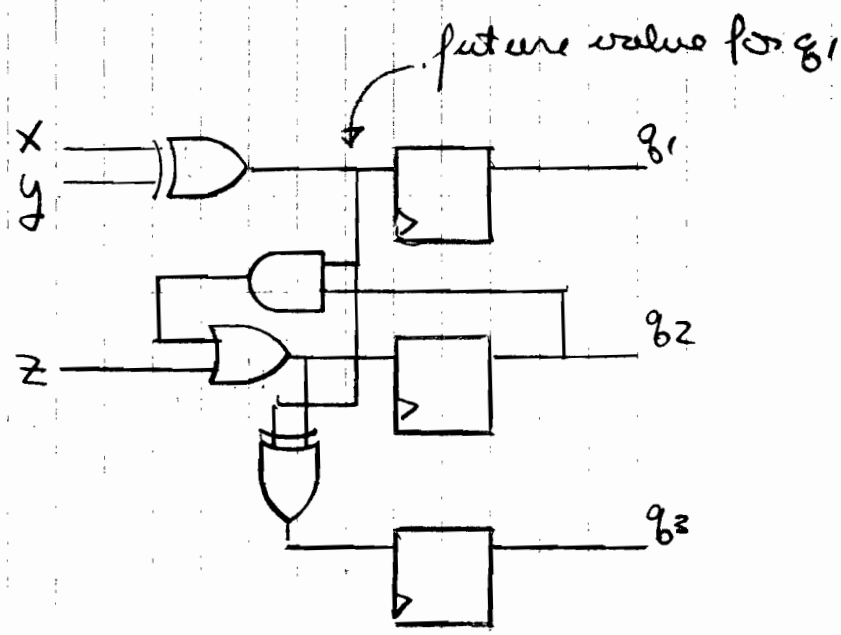
```
always @ (posedge clk) q1 = d1;
always @ (posedge clk) q2 = d1 ^ d2;
```

Example.

```
always @ (posedge clk)
begin
    q1 = x ^ y;
    q2 = q1 & q2 | z;
    q3 = q2 ^ q1;
end
```

TRANSLATES TO THIS

```
begin
    q1 <= x ^ y;
    q2 <= (x ^ y) & q2 | z;
    q3 <= ((x ^ y) & q2 | z) ^ (x ^ y);
end
```



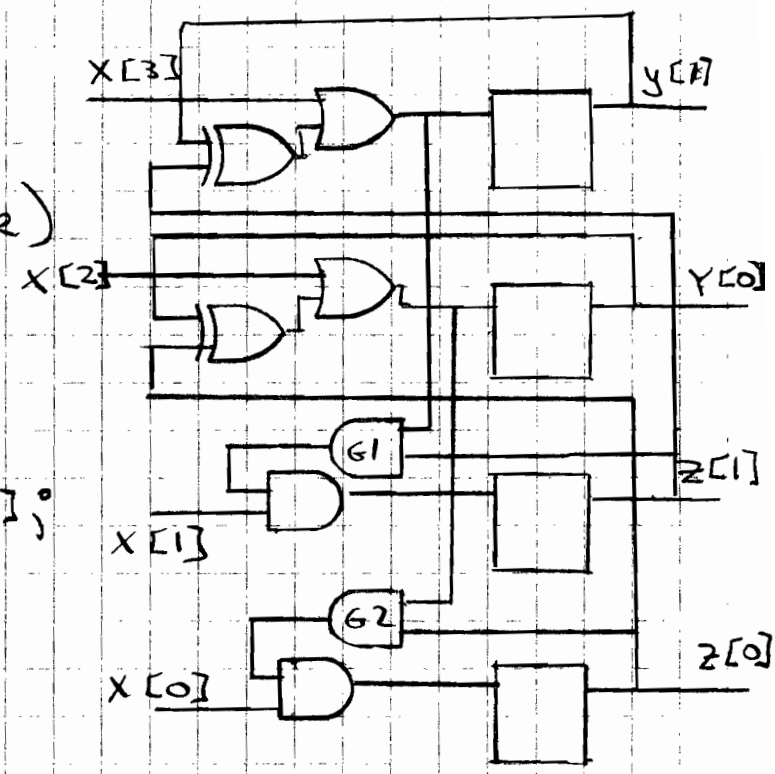
Example

generate schematics for the following Verilog descriptions

```
input [3:0] x;
output [1:0] y, z;
Reg [1,0]  y, z;

always @ (posedge clk)
begin
    y = (z ^ y) | x[3:2];

    z = (y & z) & x[1:0];
end
```



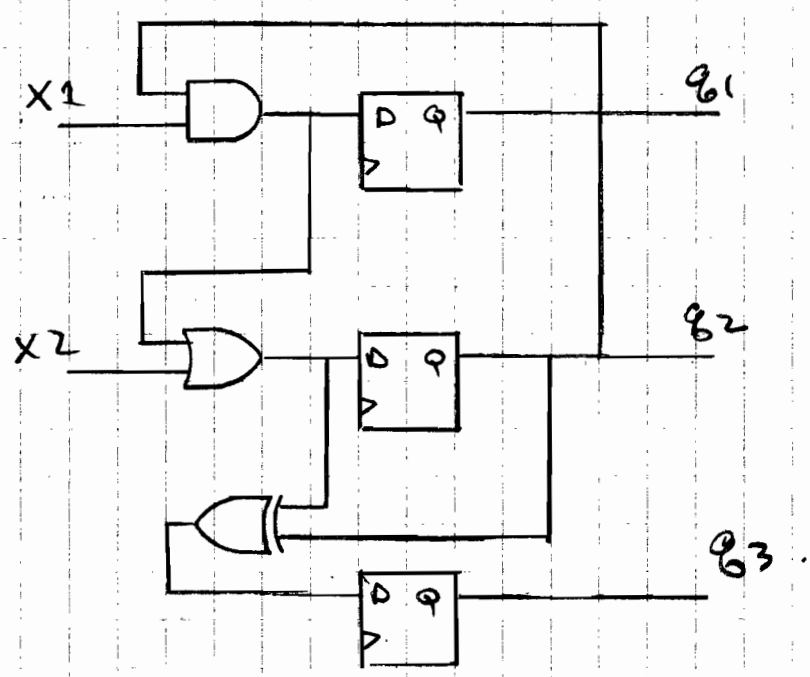
Example

```
always @ (posedge clk)
begin
    y <= (z ^ y) | x[3:2];
    z <= (y & z) & x[1:0];
end
```

Circuit is the same as above except: y inputs to "and" gates G1 & G2 are taken from the other side of f/f.

Example.

generate two Verilog descriptions of the following circuit: in one use blocking assignments in the other use non blocking

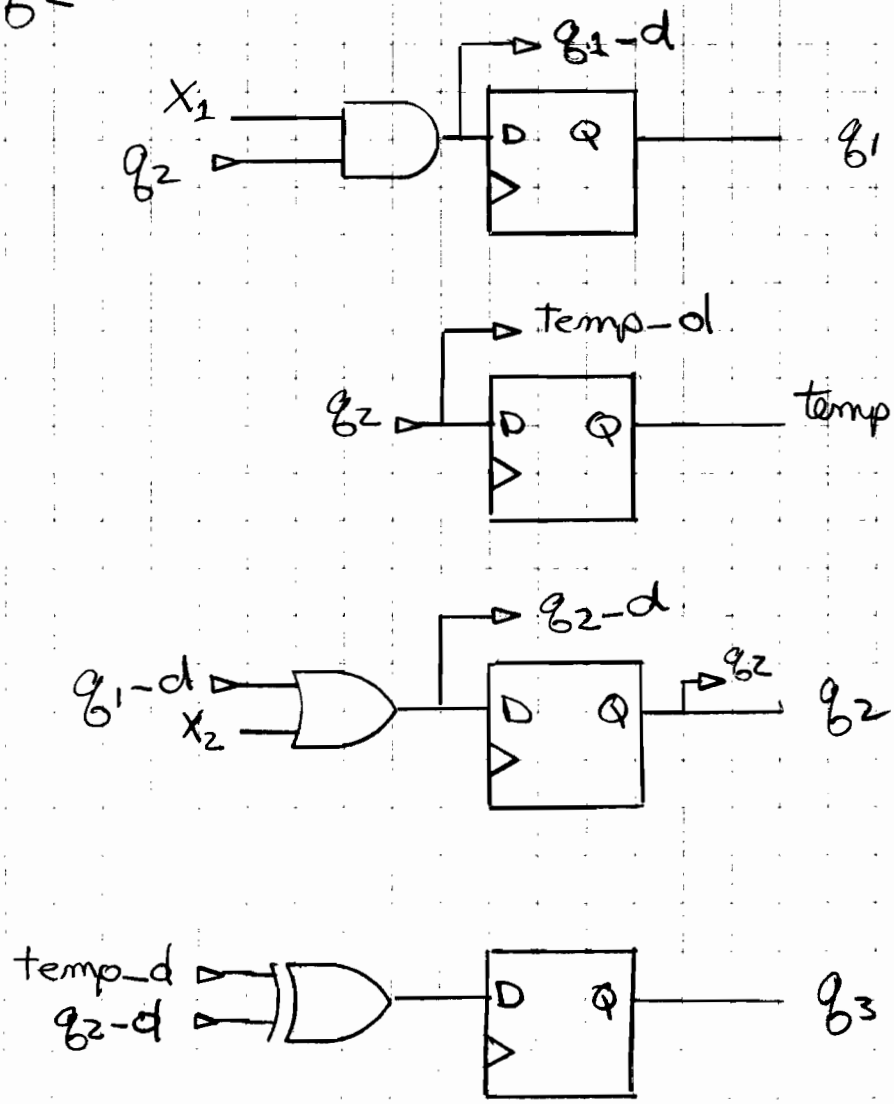



```
always @ (posedge clk)
begin
    q1 = x1 & q2 ;
    temp = q2 ;      <- need old value of q2
    q2 = q1 | x2 ;
    q3 = temp ^ q2 ;
end
```

```
always @ (posedge clk)
begin
    q1 <= x1 & q2 ;
    q2 <= (q2 & x1) | x2 ;
    q3 <= q2 ^ ((q2 & x1) | x2) ;
end.
```

Constructing the hardware from the blocking description.

```
Always @ (posedge clk)
begin
    q1 = x1 & q2;
    temp = q2;
    q2 = q1 | x2;
    q3 = temp ^ q2.
end.
```



Procedure

First draw the four flip flops. Then make the logic that goes in front of the "d" inputs starting with the first, which is q_1 and progressing to temp, then q_2 and then q_3

Notice that the solution for the "blocking assignment" formulation requires the adding another variable, which was called "temp". This is because the "old value" of q_2 is required after q_2 is assigned. To remember the old value of q_2 it is assigned to temp, which must be done prior to assigning q_2.

<u>NB:</u> If you change the order of blocking assignments within a begin-end wrapper, the circuit could change.

The order of non-blocking assignments within a begin-end wrapper does not matter.

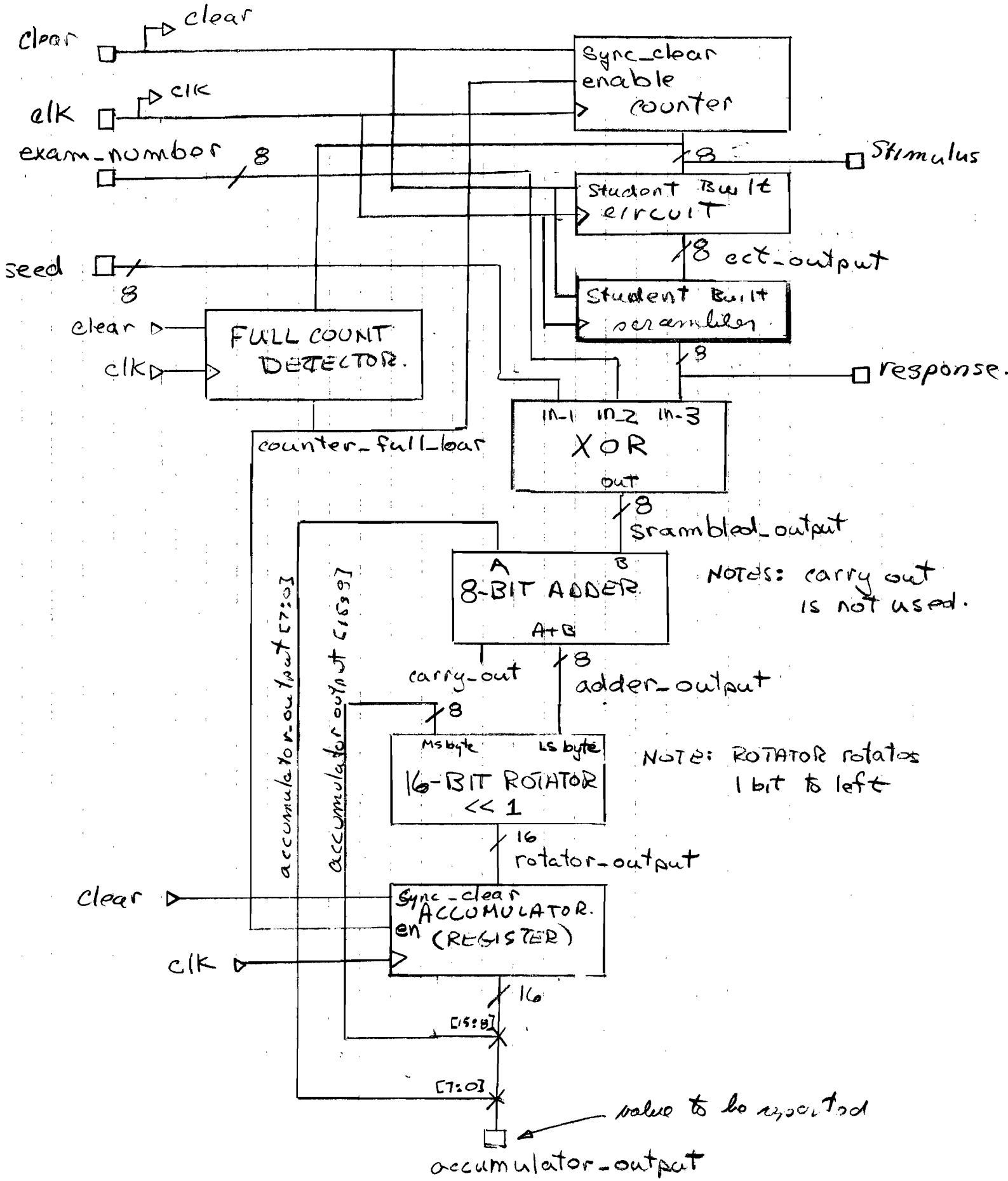
Example.

Design a circuit that can test an 8-input, 8-output circuit with a high degree of certainty. The circuit is to be used in midterm exams to evaluate circuits that were designed during the exam.

The students use the test bench during the exam to test their circuit. They report a 16-bit hex number, which can be decoded by the instructor to see if the circuit operates correctly.

To avoid one student copying another's result, there are two levels of security. First, there are eight different scramblers used. Part of the scrambler is verilog HDL written by the student from instructions provided in the exam.

Second there are two keys used to scramble the output. The keys are inputs to the circuit so are defined at simulation time. The two keys are 8 bit inputs called "seed" and "exam_number". The students enter values for these in the vector waveform file. The value to be entered are given in the exam.


clear
clear
clk
clk
exam_number
8
sync_clear
enable
counter
8
Stimulus
Student Built
circuit
8
ect_output
seed
8
Student Built
scrambler
8
response.
clear
clk
FULL COUNT
DETECTOR.
counter_full_load
IN-1 IN_2 IN-3
XOR
out
8
srambled_output
A
B
8-BIT ADDER
A+B
NOTES: carry out
is not used.
carry_out
8
adder_output
accumulator_output [7:0]
accumulator output [15:8]
8
MS byte
LS byte
16-BIT ROTATOR
<< 1
NOTE: ROTATOR rotates
1 bit to left
16
rotator_output
clear
clk
Sync_clear
ACCUMULATOR.
en
(REGISTER)
16
[15:8]
[7:0]
value to be accepted
accumulator_output

```
module exam_test_bench_2009 (
  input clk, clear,
  input [7:0] seed, exam_number,
  output reg [7:0] stimulus,
  output wire [7:0] response,
  output reg counter_full_bar,
  output reg [15:0] accumulator_output
                                    ); // verilog 2001 extention

reg [7:0] scrambled_output;
reg [7:0] adder_output;
reg [15:0] rotator_output;

wire [7:0] cct_output;

always @ (posedge clk)  //design the accumulator
  if (clear == 1'b1)
     accumulator_output <= 8'b0;
  else if (counter_full_bar == 0)
     accumulator_output <= accumulator_output;
  else
     accumulator_output <= rotator_output;


always @ (posedge clk)  //design the counter
  if (clear == 1'b1)
     stimulus <= 8'b0;
  else if (counter_full_bar == 0)
     stimulus <= stimulus;
else
  stimulus <=
       stimulus + 1'b1;

always @ * //design the rotator
  rotator_output = {accumulator_output[14:8],
          adder_output, accumulator_output[15]};
```

```
always @ * //design the adder
  adder_output = accumulator_output[7:0]+scrambled_output;

always @ * // design the scrambler
  scrambled_output = seed ^ exam_number ^ response;

always @ (posedge clk) // design the counter full dectector
  if (clear == 1'b1)
            counter_full_bar <= 1'b1;
  else if (& stimulus == 1'b1) // counter is full
            counter_full_bar <= 1'b0;
  else
            counter_full_bar <= counter_full_bar;

/* ****************************
 prototype for ``student_circuit''
 is provided by the student
*******************************/

 student_circuit cct_1 (.clk(clk),
                        .clear(clear),
                        .exam_number(exam_number),
                        .cct_input(stimulus),
                         .cct_output(cct_output) );
 student_scambler scmblr_1(.clk(clk),
                         .clear(clear),
                        .exam_number(exam_number),
                        .scrambler_input(cct_output),
                        .scrambler_output(response) );

endmodule
```

Critical Mistakes made on past midterms

1. Specified clock period to be 1 ms instead of 1 μs. Since the cyclone II is not fast enough for a 1 GHz clock it can not simulate the circuit. It points an error – but the students didn't understand it.

2. The display for the waveform report was zoomed so much the scroll bar was so small that it was not noticed. It appeared that all answers were 0.

3. The simulation end time was set incorrectly and the counter never reached 8'HFF.

4. A different verilog file was used for question #2. The top entity of the project was not changed so the compiler used the verilog file for Question #1 on all the compiles.

5. 8 bit registers were used in the design but they were declared as single bit registers
e.g. reg Q; instead of reg[7:0] Q;

6. The project was set up on the h drive in the top directory. A quartus project must be inside a folder.

Quartus Error Messages

Quartus error messages and warning messages are cryptic (i.e. obscure and curt).

However, Quartus does provide a good explanation of the cause of each error or warning as well the action that must be taken to correct the cause.

To get the explanation select

HELP → messages

then scroll down the list of message until you find the warning or error that you have. Double click on that message. The explanations of Cause and Action will then appear in the window pane on the right.