CME341 HDL Class Notes

Dr. Eric Salt

Originally created over 1 year ending September 8, 2016
Latest revision: September 3, 2021

Revised on:

October 2, 2016:	added verilog 2001 synthesis directive for full case
September 5, 2017:	Revised chapter 1 up to end of demo. Updated procedures for the new version of Quartus, Quartus Prime, and the new version of modelsim-altera, Modelsim-Altera 10.5b. Also removed references made to the old DE2 boards.
September 6, 2017:	Corrected a few typos
November 7, 2017:	Extended these typed notes to include the first part of the section on Sequential Circuits. The remainder of the notes are in the hand written version of Chapter 3
December 29, 2017:	Added instructions on saving and restoring the wave window in modelsim.
September 4+5, 2020:	Initial modifications for SystemVerilog support (Brian Berscheid)
September 21, 2020:	Fixed a few minor typos in Chapter 3. (Brian Berscheid)
September 3, 2021:	Added instructions to generate timing simulation netlist in Quartus 20. (Brian Berscheid)

Contents

Chapter 3

Synthesis of Verilog HDL Procedures

3.1 Quartus Design Flow and Synthesis

Quick overview of the synthesis process:

The compiler uses the Verilog HDL file as an input.

Using templates for if-else statements, 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"technology mapper". This program converts the logic in the circuit into the technology available on the device. For example, if only nand gates are available on the FPGA, logic circuits using logic other than nand gates would be converted to circuits built entirely of nand gates. The logic in the Cyclone family of FPGAs is implemented with lookup-tables. Therefore, for Cyclone FPGAs the mapper maps logic to lookup-tables.

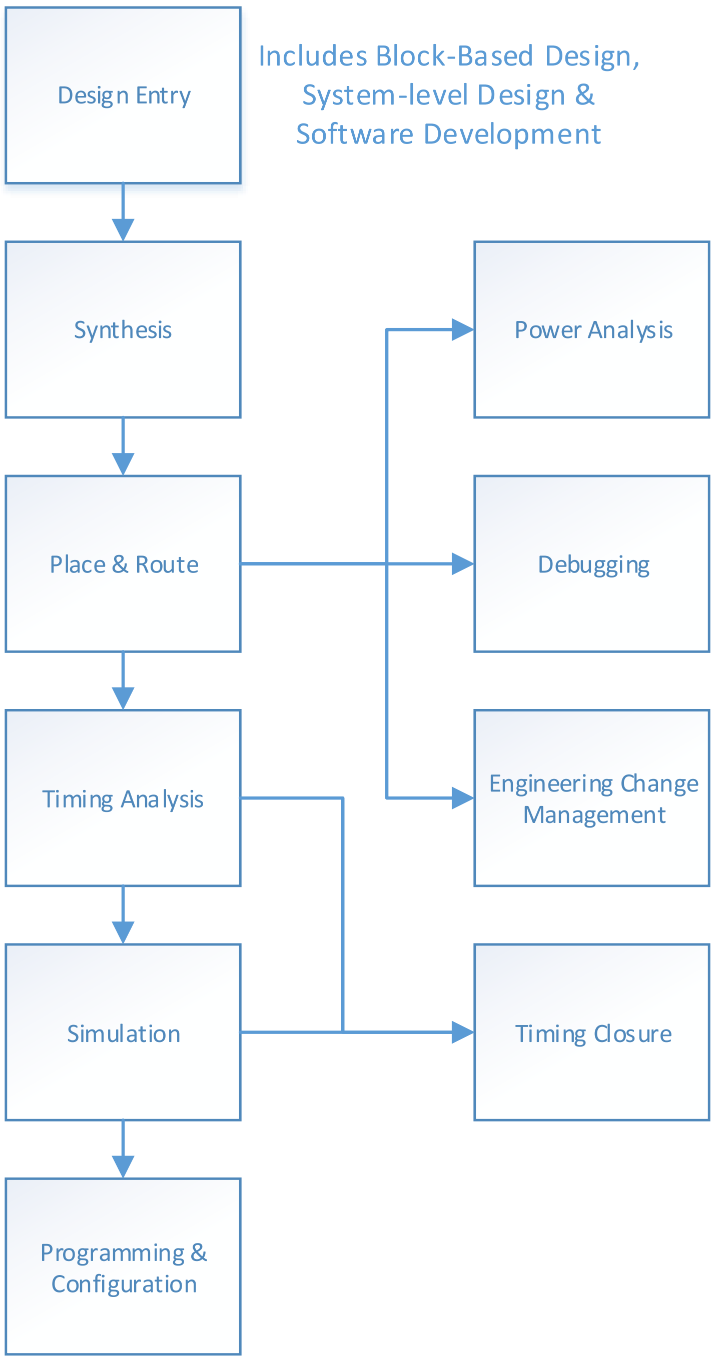
Design Entry
Includes Block-Based Design, System-level Design & Software Development
Synthesis
Place & Route
Timing Analysis
Simulation
Programming & Configuration
Power Analysis
Debugging
Engineering Change Management
Timing Closure

Figure 3.1: Quartus II Design Flow

3.1.1 Synthesis Process

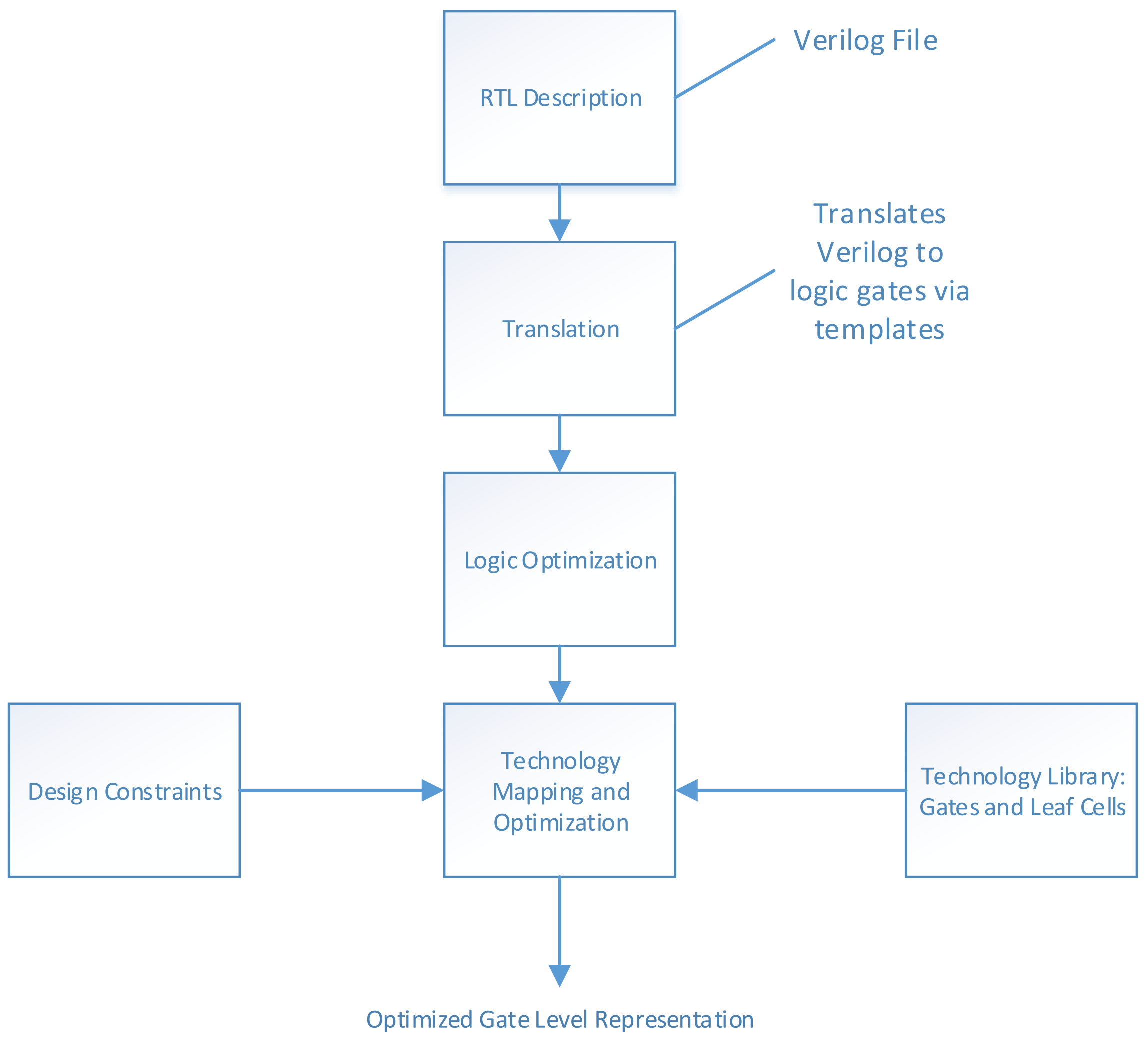
RTL Description
Verilog File
Translation
Translates Verilog to logic gates via templates
Logic Optimization
Design Constraints
Technology Mapping and Optimization
Technology Library: Gates and Leaf Cells
Optimized Gate Level Representation

Figure 3.2: Synthesis Process

3.2 Designing Combinational Logic using Procedures

3.2.1 always blocks

Procedures in Verilog begin either with an “initial” or an “always” statement. Both “initial” and “always” procedures can be used in Verilog descriptions written for simulators. However, only the “always” procedure can be synthesized. This means the “initial” procedure can not be used in designs compiled in Quartus, but can be used in test benches written for simulation.

The always procedure has syntax `always @ (a or b)`. The argument is a list of signals entered between the brackets ( ). In this example the signals are separated “or”. This argument is called the sensitivity list.

The sensitivity list is only used by compilers for simulators, but compilers for synthesizers check the list for syntax to make sure the Verilog HDL will compile for a simulator.

```
input [3:0] a, b;
output [3:0] c;
reg [3:0] c; // The output of a procedure must
             // be declared type reg

always @ (a or b) // could use wildcard * in
                  // place of ‘‘a or b’’.
begin  // begin-end are unnecessary if the
          procedure has a single statement
if(a == b)
c = 4’b1011;
else
c = 4’b0011;
end
```

Key rules for procedures:

- The outputs generated in a procedure must be declared type “reg”.
- The keyword “always” indicates that a procedural statement is to follow.
- The list of items in parenthesis after the ”always” statement is called the sensitivity list. These specify when the outputs in the procedure can change.
- The ”begin – end” wrapper is only needed when more than one statement is used in a procedure.

After Synthesis (compilation) and optimization the “always” procedure above could result in the circuit in Figure 3.3:

For synthesis a complete sensitivity list is always used. That is, every input signal used in the procedure must be listed in the sensitivity list. To save the time of typing all the signals, Verilog provided a wild card “*” to indicate all inputs. Therefore in synthesis “always @ *” is used to generate combinational logic.

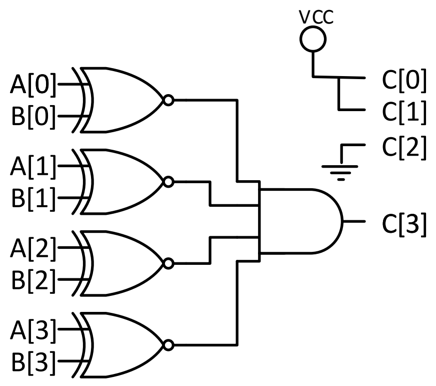

Figure 3.3: Circuit resulting from the simple always procedure in the notes

For example,

```
always @ (a or b)
   if(a > b)
      c = a;
   else
      c = b;

Is equivalent to:

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

3.2.2 Role of the Translator in Synthesis

The translator converts Verilog HDL to a logic circuit. It does this by first associating a Verilog HDL with a logic circuit template and then connects signals to that template. For example the Verilog HDL procedure

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

is translated in two steps: First the if-else section is translated to the circuit shown in Figure 3.4.

Then the comparison (a > b) is translated to a circuit that produces a 1 if (a > b) evaluates to

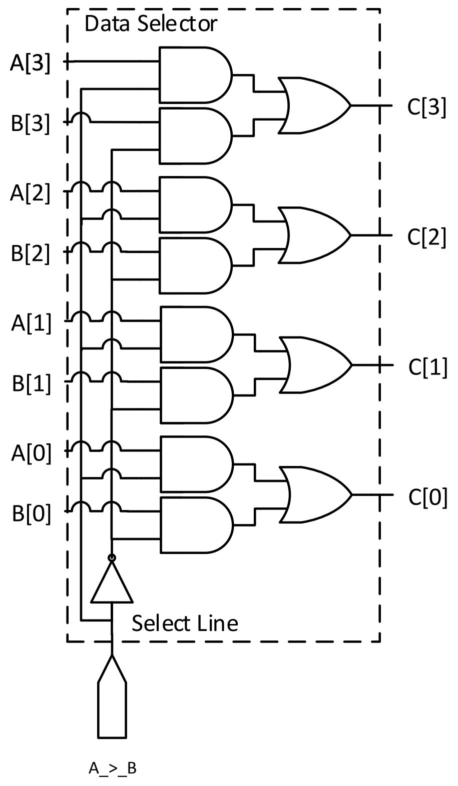

Figure 3.4: Translation for if-else

"true" and a 0 if it evaluates to "false". The circuit that evaluates the expression (a > b) is given in Figure 3.5. Then the output of Figure 3.5 is connected to the A_>_B input in Figure 3.4.

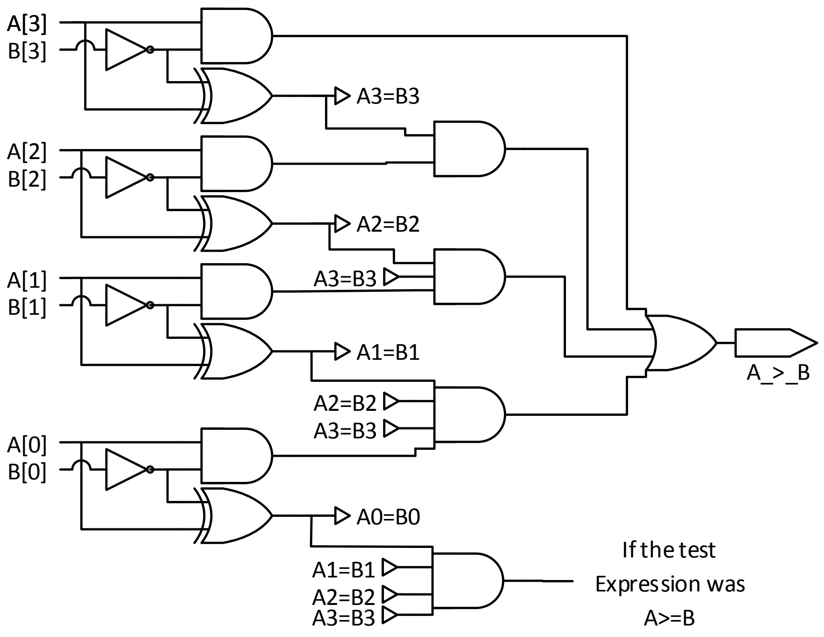

Figure 3.5: Translation of A > B to a logic 1 or logic 0.

3.3 Technology Mapping

The function of the technology mapper is to convert the logic circuits to the fabric used in the FPGA. Suppose the hardware in an FPGA has the architecture shown in Figure 3.6.

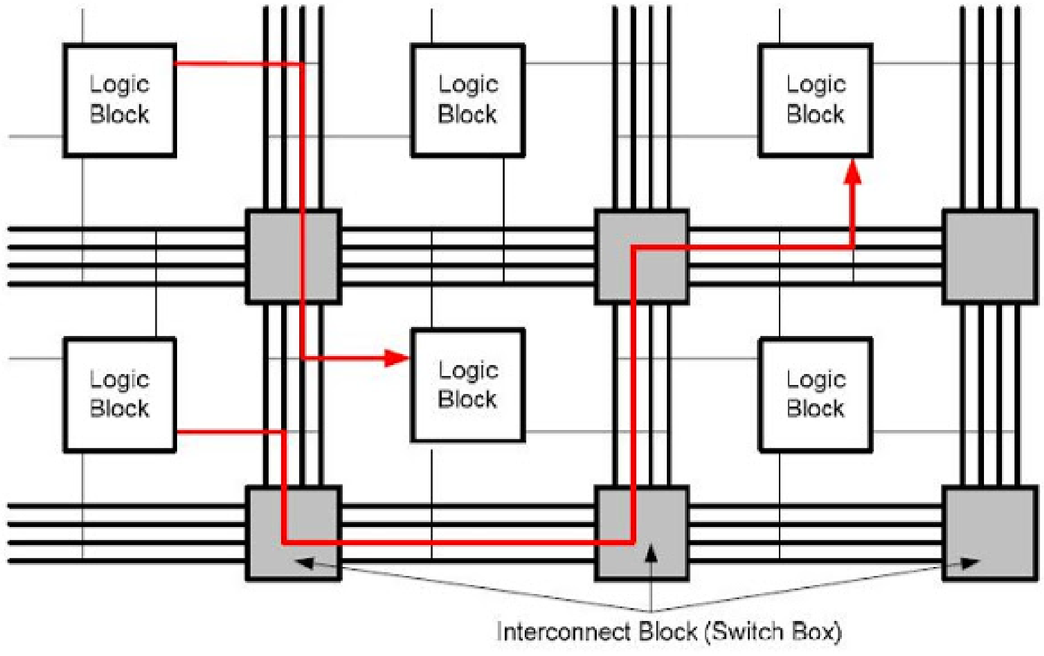

Figure 3.6: Simplistic View of the Architecture for an FPGA

The horizontal and vertical tracking are on different planes, but can be connected together by transistor switches that are programmed at compile time to be on or off. Each logic block a ROM with 4 inputs and 1 output. The inputs and output of each logic block can be connected to any of several horizontal and vertical tracks. The grey boxes are comprised of an array of transistor switches that can connect one horizontal to another horizontal track, one vertical track to another vertical track or one horizontal track to one vertical track.

The technology mapper constructs the logic circuit as a network of ROMs without regard for the placement of ROM in the FPGA or the wiring needed to connect them. The way this is done is illustrated in Figure 3.7.

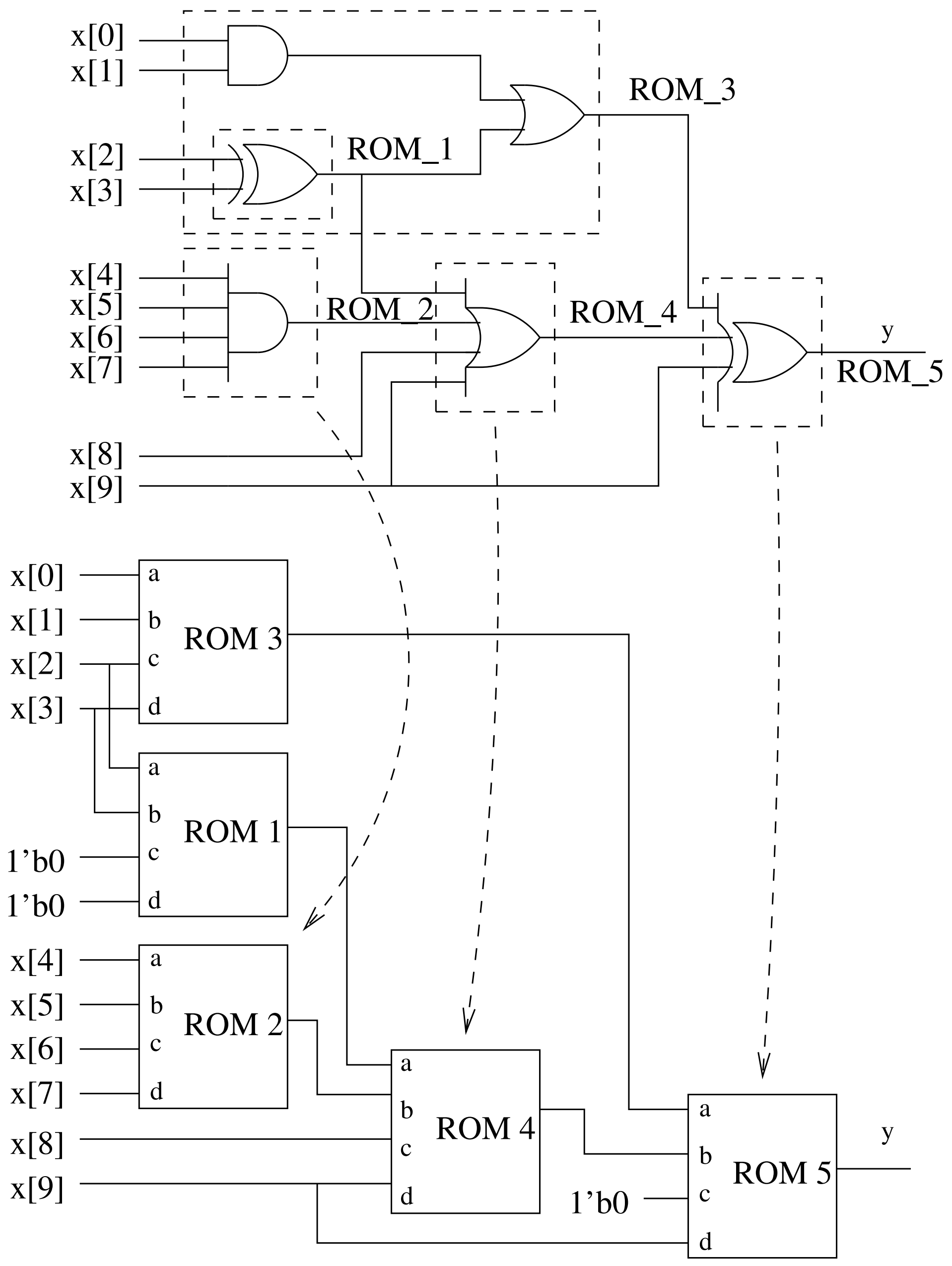

Figure 3.7: Illustration of the technology mapping process.

3.4 Place and Route

After the technology mapper has created a network of 4-input 1-output ROMs that constructs the circuit, the ROMs are placed (i.e. assigned positions) in the FPGA. The process of doing this is called fitting and the software that does it is called the fitter in Quartus. After the ROMs have been placed by the fitter a router tries to wire the ROMS together as they were placed in the FPGA by the fitter. If the placement is poor there will be wiring bottlenecks and the router will fails to find wiring paths to connect all the ROMS. In this case the ROMs in the network are refitted, meaning they are assigned different positions in the FPGA and then the router tries again.

After the place and route is finished, Quartus generates a report indicating how much of the FPGA is used.

More on this later.

3.5 Templates for Relational Operators

Verilog constructs the logic circuits that implement relational operators using one of several in a family of templates stored in a data base. The template that implements a “>” when the arguments are 4-bit unsigned vectors is given in Figure 3.5. Two families of such templates are generated for vectors of different length. One family for signed number and one family for unsigned numbers.

The logic-circuit template use to implement == is given in Figure 3.8.

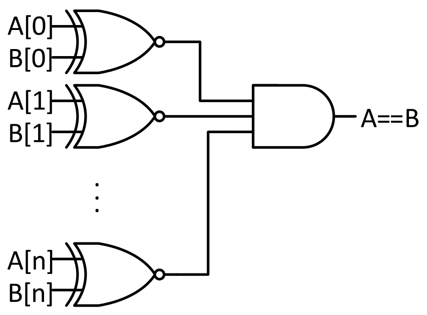

Figure 3.8: Logic circuit to determine if A == B

.

The templates for the 5 relational operators, $<$, $\leq$, $==$, $\geq$ and $>$ are obtained by combining the logic for $>$ and $==$ operators as shown in Figure 3.9.
To change from $>$ to $<$ the inputs to the logic-circuit template are reversed.

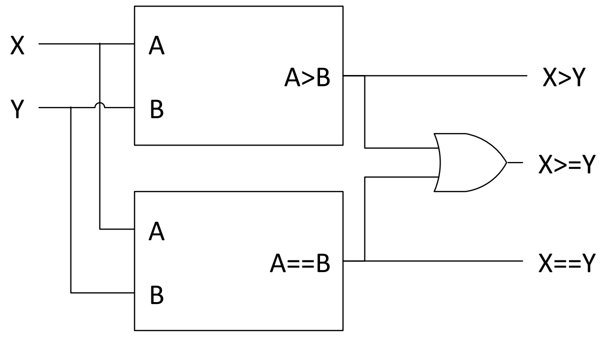

Figure 3.9: Template that covers all relational operators

3.6 Quartus Netlist Viewer Tools

Quartus provides tools to view the output of the synthesizer after translating Verilog to logic and optimizing the logic. It provides 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 and route map (i.e. the post filter map) is displayed by selecting:

Tools → Netlist Viewers → Technology Map Viewer

3.6.1 Circuit 1

```
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
```

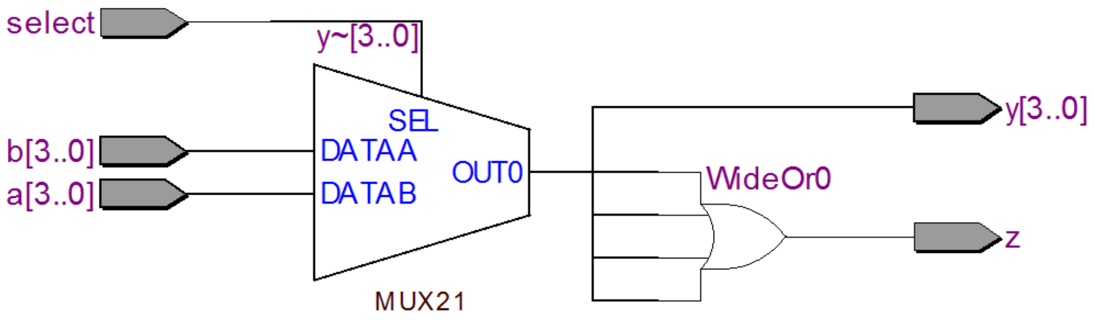

Figure 3.10: RTL View of Circuit 1

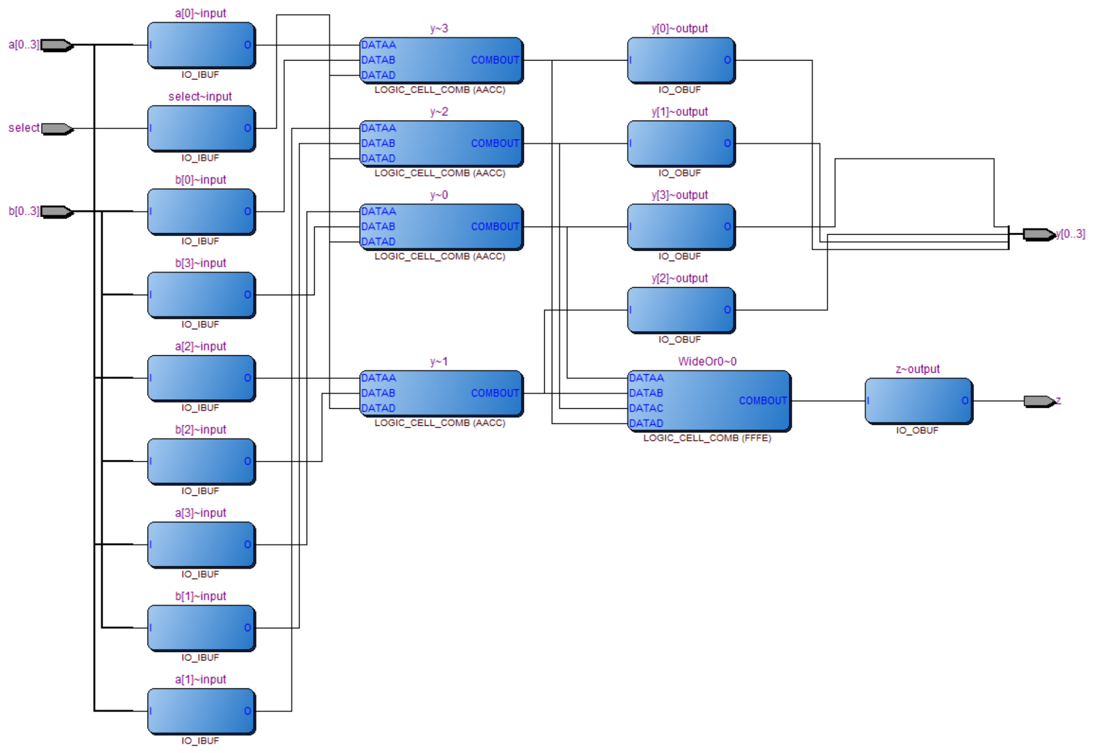

Figure 3.11: Post Mapper But Before Fitter

Each block is a 16x1 ROM (lookup table). It has four inputs: DataA, DataB, DataC, DataD. The output is called COMBOUT (COMB stands for combinational logic.
If the mouse is hovered of a LUT, it gives the logic equation implemented by that block. The logic Equation for the y∼0 LUT is:

```
y~0 = DATAD & DATAA # !DATAD & DATAB
NOTE: # means OR
      $ means XOR
```

The logic equation for the WideOr0∼0 is:

```
WideOr0~0 = DATAA # DATAB # DATAC # DATAD
```

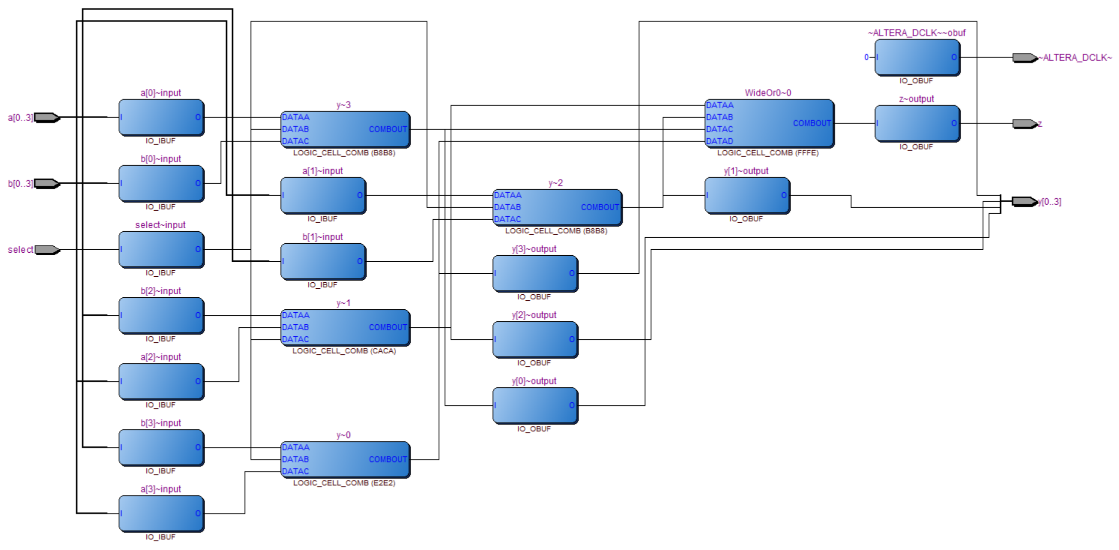

Figure 3.12: Post Mapper After Fitter

Notice that the fitter changed the wiring. For example, the select line used to go to only DATAD inputs, but after the fitter it goes to DATAB and DATAC inputs.

3.6.2 Circuit 2

```
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]);

endmodule
```

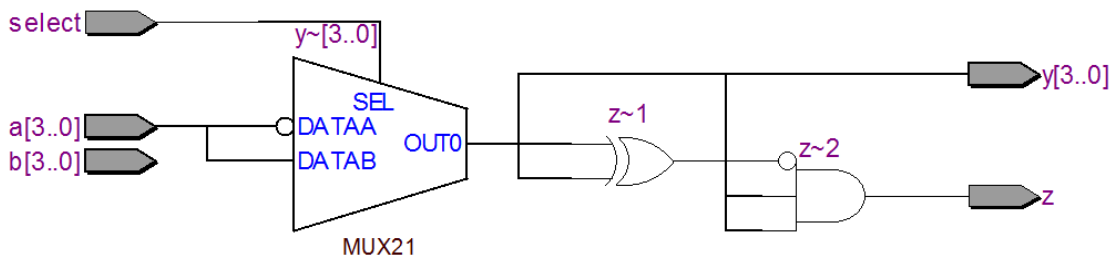

Figure 3.13: RTL View of Circuit 2

Does the post mapping differ from the post fitter on this circuit?

(yes)

3.6.3 Circuit 3

(Exercise - Try to draw the RTL of the following circuit)

```
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;

endmodule
```

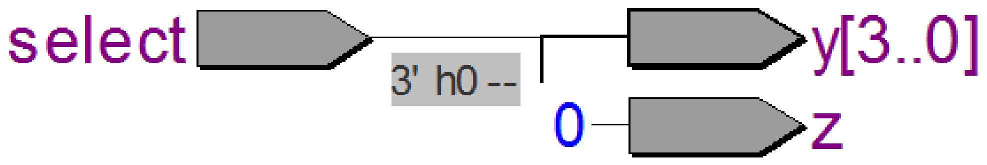

Figure 3.14: RTL View of Circuit 3

Does the post mapping differ from the post fitter on this circuit?

(No)

3.7 Coding Structures

3.7.1 If-Else Statements

Compilers translate the if-else statement in a procedure into logic circuits using templates. The if-else statement below

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

is translated into hardware using the template circuit in Figure 3.15.

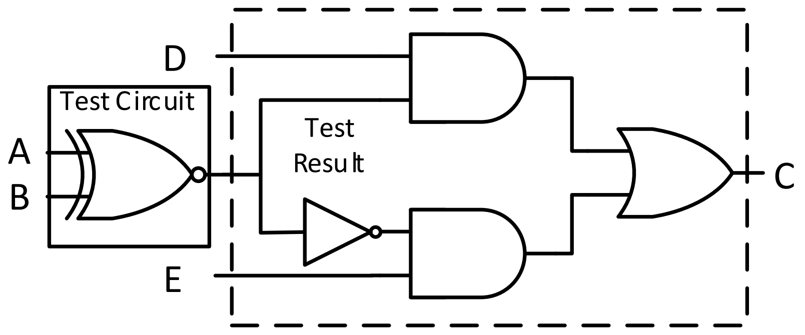

Figure 3.15: Hardware Implementation of an IF-Else statment

Nested If-Else Statements

The translation of nested if-else statements into hardware is achieved by the repeated translation of if-else statements. The way this is done is illustrated with an example. The procedure

```
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;
```

translates to the hardware shown in Figure 3.16.

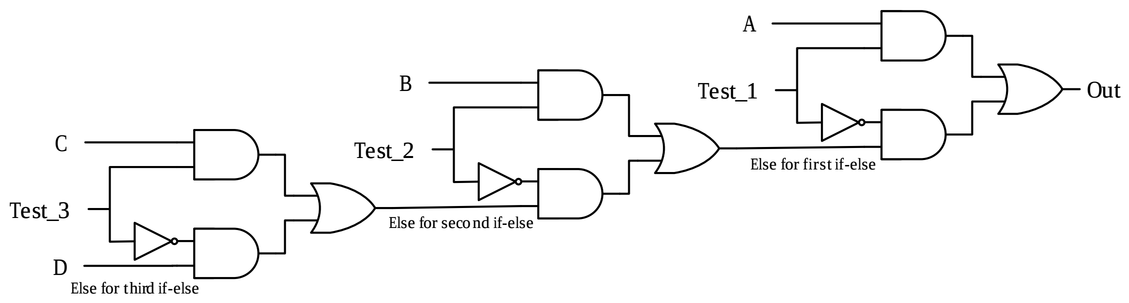

Figure 3.16: Hardware Implementation of Nested If-Else statements

The circuit produced by translation is not "flat" in that the propagation time from input to output depends on the order of testing. For the circuit of Figure 3.16 the propagation time from 'A' to 'Out' is 1/3 that of the time from 'C' to 'Out'. In the early years, before the logic optimizers were very good, the engineers had to be cognisant of how the nested if-else worked to get their designs to run fast. Now it is less of an issue as the optimizers take care of the timing issues.

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.

Solution:

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

reg [3:0] Xr;

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
```

Note: The outputs constructed in an always procedure must be of type "reg" even though the output is from a logic element, like a look-up-table ROM, and not a flip/flop.

Before closing this subsection on IF-ELSE statements a very important point must understood. Verilog HDL must have the "else" clause to translate the construct to hardware. If the else clause is omitted Verilog HDL appends one, but does issue a warning in blue print. If the output in the if clause is say Y, and the else clause is omitted from the Verilog HDL, then the compiler assumes the else clause is Y=Y. That is, the translator will label the "else" input of the template "Y" and then in the wiring stage the output will be connected to the else input.

For example the Verilog HDL

```
input gate, d;
output Q;
always @ *
if (gate == 1'b1)
Q = D;
```

will be interpreted by the compiler to be

```
input gate, d;
output Q;
always @ *
if (gate == 1'b1)
   Q = D;
else
   Q = Q;
```

which is the Verilog HDL for a transparent latch. This means if the else clause is inadvertently omitted the Verilog compiler will build a latch. When this happens the compiler prints a warning (blue writing) saying something to the effect that a circuit was built with a feedback loop and a latch of some sort was created.

Exercise: (give students 5 minutes in class to complete)
First append the "else" clause that will automatically be appended by the compiler and then draw the circuit produced by the translator for the following Verilog HDL

```
input a, x;
output y;
```

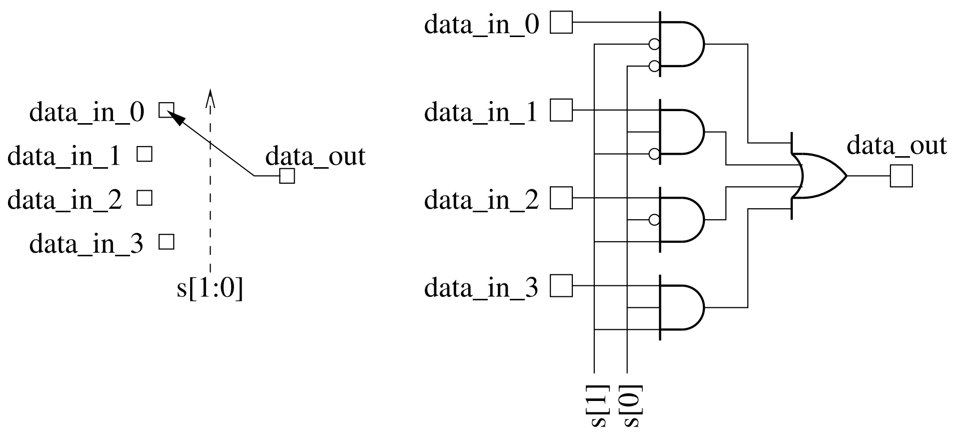

Figure 3.17: The logic used to implement a multi-throw switch and also a case statement

```
always @ *
if (a == 1'b1)
y = x;
```

3.7.2 Case Statements

Case statements build multi-throw switches that connect one of several inputs to an output. The position of the switch is controlled by a vector called the select input, which is usually symbolized with the letter `s`. The multi-throw switch and the logic template used for its implementation is shown in Figure 3.17.

The Verilog HDL that builds the logic shown in Figure 3.17 is as follows:

```
module data_selector (
input data_in_0, data_in_1, data_in_2, data_in_3;
input [1:0] s;
output reg data_out  );

always @ *
case(s)
2'b00: data_out = data_in_0;
2'b01: data_out = data_in_1;
2'b10: data_out = data_in_2;
2'b11: data_out = data_in_3;
endcase
```

The syntax of the case statement is as follows:

```
case (select_expression)
alternative_1: output=expression_1;
alternative_2: output=expression_2;
  .
```

```
   .
   .
alternative_N: output=expression_N;

default:       output=default_expression;

endcase
```

where

select_expression is any expression that in the end becomes a vector. The resulting vector is interpreted as an unsigned number. This number controls which of the expressions in the list of alternatives is steered to the output.

alternative is a number in any valid Verilog format, e.g. 2'b00. The result of the expression associated with the alternative that equals `select_expression` is steered to the output of the circuit.

default is used to cover all the alternatives that are not explicitly listed. If the `select_expression` evaluates to a number that is not in the list of alternatives then the expression associated with the default alternative is steered to the output.

If the "default" alternative is not entered in the Verilog HDL then the compiler assumes it to be `default: output = output;`. The default will not be used unless one or more of the possible alternatives are omitted from the list. In that case, as previously explained for the if-else structure, the circuit will build a latch of some sort.

The way the translator builds the circuit for a case statement is explained through an example.

Consider the following Verilog HDL

```
module data_selector (
input d0, d1, d2, d3, d4, d5;
input s0, s1, s2;
output reg y; // y must be type reg as it is
              // built in a procedure
                     );
always @ *
case( { s2, s1, s0 } )
3'b010:  y = d2;
3'b100:  y = d4;
3'b101:  y = d5;
3'b101:  y = d0; // this will be ignored
3'b011:  y = d3;
3'b000:  y = d0;
3'b001:  y = d1;
default: y = 1'b0;
endcase
```

```
endmodule
```

The translator first looks at the number of bits in the select input. In this case the select input has 3 bits. The translator then gets the generic template for a data selector with 3 select lines, which is an eight-to-one data selector. That template is shown of the left side in Figure 3.18.

After the template is selected the three bit select vector in the case statement argument is assigned to the 3 select line inputs in the generic template. In this example the three elements making up the select vector in the argument of the case have the same names as the select lines in the template. Therefore, the signal names for select lines are the same before and after the assignment.

Next the translator has to change the template inputs as dictated by the case statement. The order in which the inputs are determined does not matter, however, the process through which they are determined does matter. The process that determines the inputs is described below:

Start of process for determining inputs and output

Rename the output: The output in the case statement is `y` so the output `out` in the template circuit is replaced with `y`. This is shown in Figure 3.18.

Determine the input that is selected when the select lines are 3'b000: The alternative for this case is obviously 3'b000. The corresponding input in the generic template is the signal named `in0`. The determination of the value or expression to be assigned to the input labeled `in0` begins with a search of the list of alternatives starting from the top and proceeding downward toward the bottom. As soon as alternative "3'b000" is found the search stops and the corresponding expression is used for the input `in0`. For the Verilog HDL under consideration the search would have encountered 3'b010, 3'b100, 3'b101, 3'b101 (again), and 3'b011 before reaching 3'b000. Once 3'b000 is reached the search stops and the expression associated with alternative 3'b000, which is `y=d0`, is used to determine input `in1`. This simple expression amounts to renaming `in1 d0`.

Determine the input that is selected when the select lines are 3'b001: The same process is followed. Alternative 3'b001 is found sixth in the search list. The associated expression is `y = d1;` so the compiler simply renames `in1 d1`.

⋮

Determine the input that is selected when the select lines are 3'b101: The same process is followed. What is different is alternative 3'b101 appears twice in the list and the two occurrences have different assignment expressions. Only the first alternative encounter in the search is used so input `in5` is renamed `d5`.

Determine the input that is selected when the select lines are 3'b110: Alternative 3'b110 does not appear in the list of alternatives so the compiler uses the expression associated with the default alternative. Since the expression for the default alternative is `y = 1'b0`, input `in6` is assigned 1'b0.

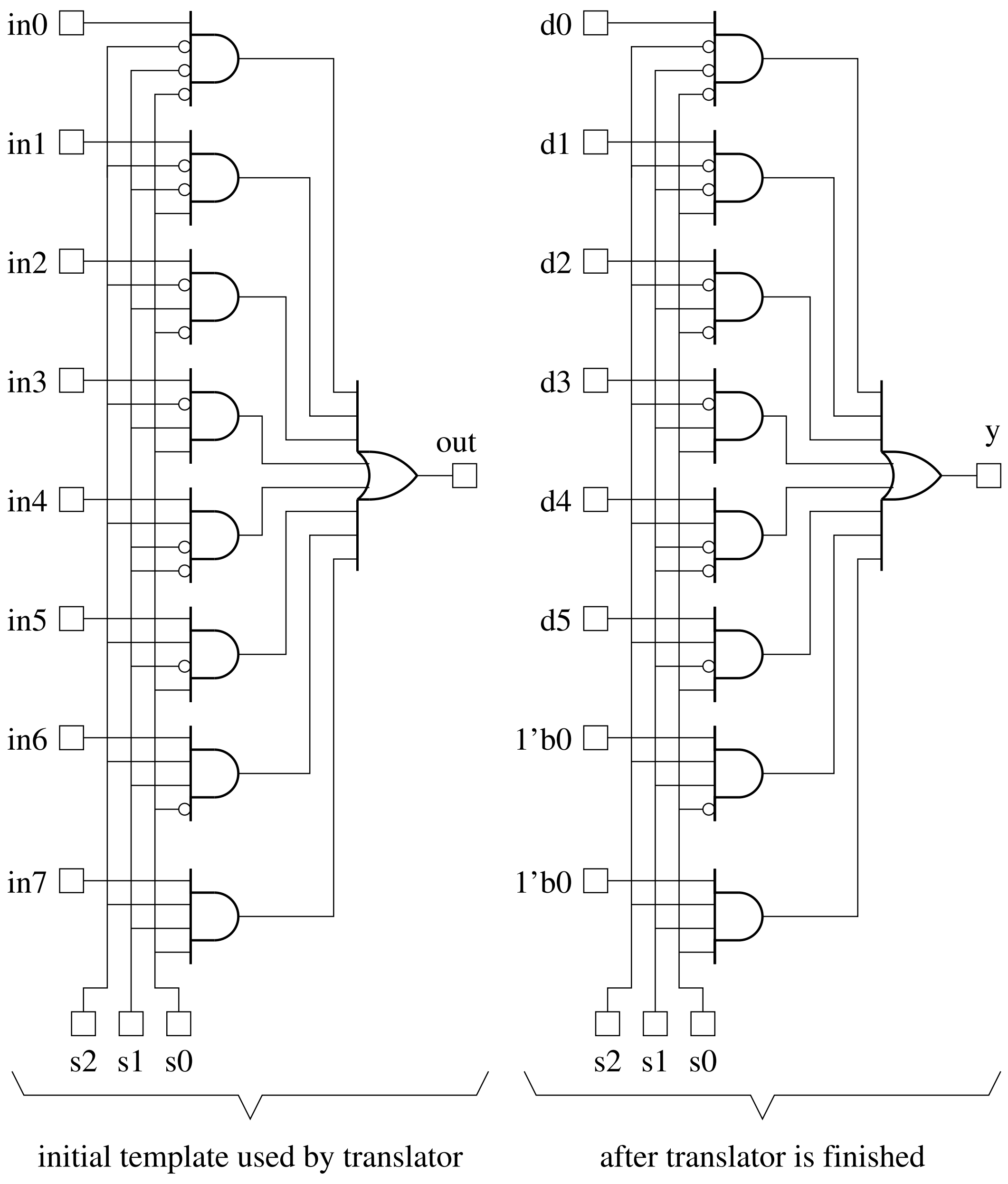
in0
in1
in2
in3
in4
in5
in6
in7
out
s2
s1
s0
initial template used by translator
d0
d1
d2
d3
d4
d5
1'b0
1'b0
y
s2
s1
s0
after translator is finished

Figure 3.18: Hardware generated by the translator for the case statement in the example

Determine the input that is selected when the select lines are 3'b111: The result is the same as for select lines 3'b110.

End of process for determining inputs and output

After the translator has finished determining all the inputs the circuit becomes that shown on the right side of Figure 3.18.

3.7.3 casex statements

Verilog HDL allows for "don't cares" to be used in case alternatives. When that is done the "case" in the case statement should be "casex", but many compilers will allow don't cares if "case" is used.

An example casex application is given below:

```
module data_selector (
input a, b, c, d, e;
input [2:0] sel;
output reg y; // y must be type reg as it is
              // an output in a procedure
                     );
always @ *
casex ( sel ) // should use casex if don't cares appear
              // in alternatives
3'b1xx:  y = a;
3'bxx0:  y = b;
3'b101:  y = c;
3'b001:  y = d;
3'b010:  y = e;
endcase
endmodule
```

The "`x's`" in the alternatives means that it could be either a 1 or a 0. This means the first alternative expands to 4 alternatives: 3'b1xx $\longrightarrow$ 3'b100, 3'b101, 3'b110, 3'b111 and the second alternative expands into 4 alternatives: 3'bxx0 $\longrightarrow$ 3'b000, 3'b010, 3'b100, 3'b110.

Before the compiler starts synthesizing it pre-compiles casex statements, that may have alternatives containing don't cares, into case statements that do not have don't care in the alternatives. The pre-compilation replaces the casex statement in the procedure above with the case statement in the procedure below:

```
module data_selector (
input a, b, c, d, e;
input [2:0] sel;
output reg y          );
```

```
always @ *
case ( sel ) // casex is changed to case

3'b100:  y = a;
3'b101:  y = a;
3'b110:  y = a;
3'b111:  y = a;

3'b000:  y = b;
3'b010:  y = b;
3'b100:  y = b;
3'b110:  y = b;

3'b101:  y = c;
3'b001:  y = d;
3'b010:  y = e;
endcase
endmodule
```

Using the rules for the search produces the equivalent case statement listed below.

```
module data_selector (
input a, b, c, d, e;
input [2:0] sel;
output reg y          );

always @ *
case ( sel ) // equivalent case statement
3'b000:  y = b;
3'b001:  y = d;
3'b010:  y = b;
3'b011:  y = y; // not in alternative list and
                // default is not listed therefore
                // use the assumed default of y = y
3'b100:  y = a;
3'b101:  y = a;
3'b110:  y = a;
3'b111:  y = a;
endcase
endmodule
```

Notice that inputs `c` and `e` are not used. The compiler will issue some sort of warning saying that inputs `c` and `e` are not used.

Exercise (give the students 5 minutes in class to complete)
Generate the equivalent case statement with alternatives organized in increasing order for the casex statement in the Verilog HDL module given below. Once that is done translate the equivalent case statement to a schematic diagram like the one in Figure 3.18.

```
module data_selector (
input a, b, c, d;
input [2:0] sel;
output reg y;        );

always @ *
casex ( sel )
3'b1x1:  y = a;
3'b0x0:  y = b;
3'b101:  y = c;
3'b00x:  y = d;
endcase
endmodule
```

3.7.4 Ill constructed synthesis directive - full_case

The standard "Verilog 2001" allows special directives to be given to synthesizers and ignored by simulators. The directives are enclosed in `(* directive *)`. One such directives is `(* full_case *)`. While many directives are valued this particular directive is viewed by some (perhaps many) to be evil. This directive instructs the synthesizer to act in certain way if the default case item is missing. It tells the synthesizer to assume the missing default case item assigns don't cares to the output.

This is done as follows:

```
reg [3:0] q;
always @ *
(* full_case *) case (sel)
      2'b00: q = a;
      2'b01: q = b;
      2'b10: q = c;
endcase
```

For purposes of synthesis this is equivalent to appending the default statement `default: q = 4'bxxxx;`. If the .vo output produced by the synthesizer is used a simulator it will simulate as it was synthesized.

However, if the original .v (or .sv) is used in a simulator the directive is ignored and the simulator builds a latch for case item 3'b11.

Writing HDL that synthesizes differently than it simulates when it is unnecessary is not good practice. **It is recommended a default statement be used instead of the synthesis directive `(*`**

`full_case *).` It is further recommended the default statement assign a constant, say `q=4'b0000`, to the output rather than don't cares, i.e. `q=4'bxxxx`.

3.8 Gluing procedures and assignments together

The synthesizer constructs a circuit from a Verilog HDL in three steps. In the first step it breaks all procedures with multiple outputs into multiple procedures each with one output. The output may well be a vector.

A procedure or assignment with a single output can be multi-layered and quite complicated. For example the statement could be an if-else with case statements in both the if and else clauses. The alternatives within the case statement could contain an if-else statement or another case statement. The alternatives within the second layer of case statements could contain case statements. Etcetera.

For example the always procedure

```
always @ *
begin
   if (x==2'b10)
     case (m)
       3'b000:  y = 8'hBD;
       3'b001:  y = 8'h3F;
       default: y = 8'H44;
     endcase
   else
     y = b;

   case(x)
     1'b00:    w = c;
     default:  w = d;
   endcase
end
```

is broken into the two procedures

```
always @ *
   if (x==2'b10)
     case (m)
       3'b000:  y = 8'hBD;
       3'b001:  y = 8'h3F;
       default: y = 8'H44;
     endcase
   else
     y = b;

always @ *
```

```
   case(x)
     1'b00:    w = c;
     default:  w = d;
   endcase
```

The second step is translation. All the procedures and assignments, which now have a single output, are translated into circuits. This is done one procedure at a time using only the information within that procedure. The order in which the procedures are translated to circuits does not matter. At the end of this step there are a host of circuits with inputs and outputs that have names drawn from those within the associated procedures or assignments.

The third step is to wire all the circuits together. This is done by wiring the inputs and outputs of each circuit to tracks with the same name. These tracks, which are physical wires, are established and given a name by declaring a signal either a wire or a register or an input or an output. All declarations build a physical wire track and give that track a name and a type. The track type, which is either wire or reg, does not matter in the wiring step.

The wiring rule is quite simple: each track must have a single driver meaning it must be connected to one and only one output of a procedure or assignment. If after wiring a track does not have a driver (i.e. is not driven by an output) or if a track has more than one driver) (i.e. is connected to two or more outputs) the compiler issues an error stating as much and does not generate either the .sof or the .vo file.

For example consider the following Verilog HDL:

```
module wiring_example (
   input a, b,
   output reg y        );

   reg c;
   wire d;

   assign d = expression_1

   always @ * // procedure 1 - has output y
      procedure_1_statement

   always @ * // procedure 2 - has output c
      procedure_2_statement

   endmodule
```

After the second step of compilation the schematic diagram of the circuit looks like that shown in Figure 3.19. After the third step of compilation, which is the final step, the schematic diagram should look like that shown in Figure 3.20.

Exercise (give the students 5 minute in class to complete)
Generate the schematic diagram for the circuit described by the Verilog HDL below by following

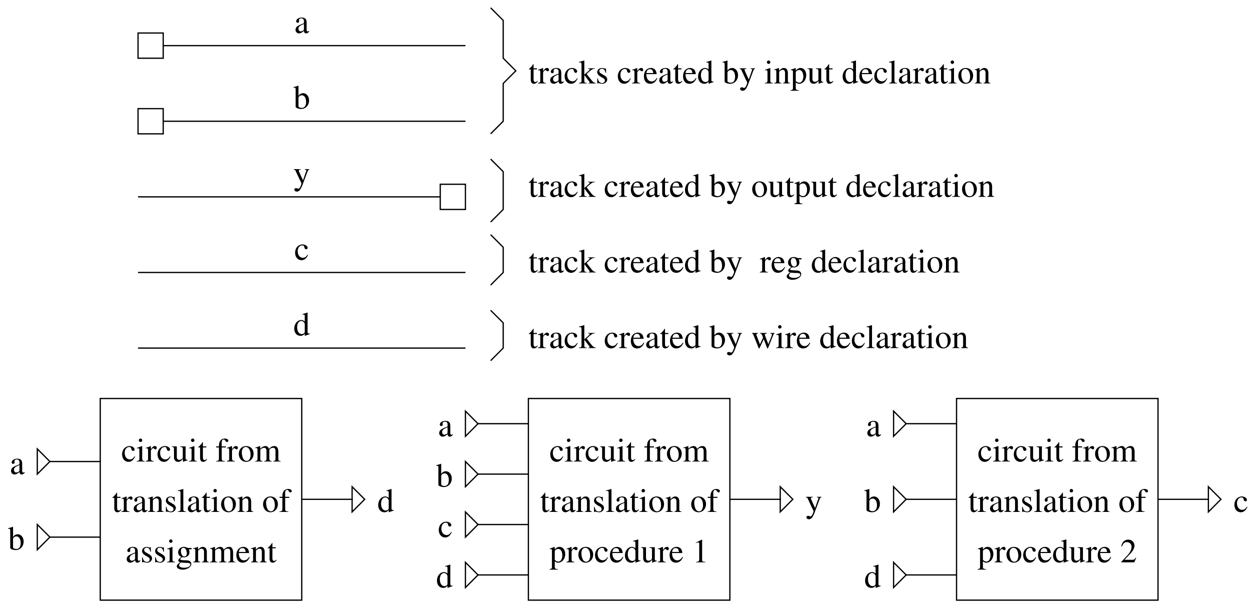
a
b
tracks created by input declaration
y
track created by output declaration
c
track created by reg declaration
d
track created by wire declaration
a
b
circuit from translation of assignment
d
a
b
c
d
circuit from translation of procedure 1
y
a
b
d
circuit from translation of procedure 2
c

Figure 3.19: The schematic diagram of the example after the second step of compilation

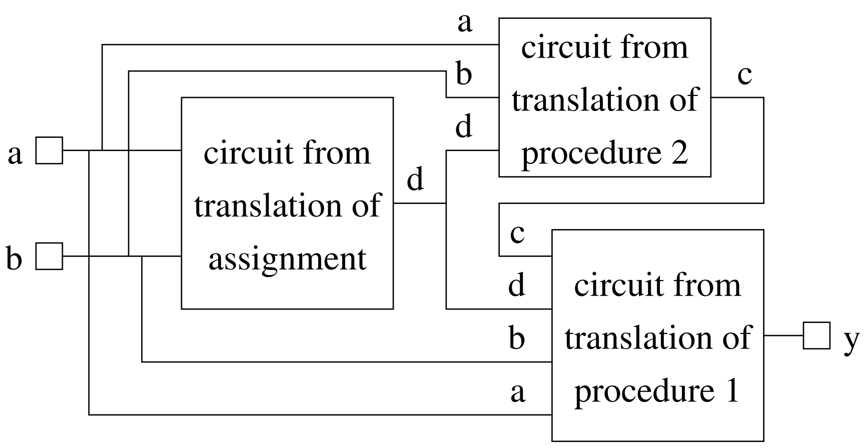
a
b
circuit from translation of assignment
d
a
b
d
circuit from translation of procedure 2
c
c
d
b
a
circuit from translation of procedure 1
y

Figure 3.20: The schematic diagram of the example after the final step of compilation

the three steps used in compilation.

```
module wiring_exercise (
   input a, b;
   output reg y         );

   reg c;
   wire d;

   assign d = a ^ b;

   always @ * // procedure 1
      casex({a, b})
        2'b1x:   y=c;
        2'bx0:   y=d;
        default: y=1'b0;
      endcase

   always @ * // procedure 2
      if (a==b)
        c = d;
      else
        c = ~d;

   endmodule
```

3.9 Summary of Important Rules and Quirks

1. If the else clause of an if-else statement is omitted the compiler inserts an else clause, which is `else output = output`. The compiler will print a warning indicating it has done this. The warning will say an else clause was inserted and a loop containing feedback was created possibly building a latch.
2. If a case alternative is missing and there is no default listed the compiler inserts an `output = output` for that alternative and issues a warning similar to the one described above.
3. **An output can not be assigned values in two different procedures, assignments or any combination there of.**

   EVERY OUTPUT MUST BE COMPLETELY DEFINED IN ONE ALWAYS PROCEDURE OR IN ONE ASSIGN.
4. If an output is defined more than once through multiple statements within the begin-end wrappers of an always procedure then the compiler will use the last one. For example, only the last if-else in the Verilog HDL will below will be used by the compiler.

   ```
   always @ *
   begin
   ```

```
          if(x==4'b0001) y = 4'hD;
          if(x==4'b0010) y = 4'hE;
          if(x==4'b1000) y = 4'hA;
          else           y = 4'h0;
        end
```

What was probably intended in the Verilog HDL above could have and should have been written as the single statement given below

```
        always @ *
          if(x==4'b0001) y = 4'hD;
          else if(x==4'b0010) y = 4'hE;
          else if(x==4'b1000) y = 4'hA;
          else           y = 4'h0;
```

The compiler only allows one statement in an always procedure unless a begin-end wraps multiple statements. If a begin-end wrapper is not used the compiler flags the extra statement that cause multiple output assignments as being not allowed and prints an error message in red print that can not be ignored.

5. It is a good rule for novices and even somewhat experienced engineers to use begin-ends sparingly and write separate procedures for each output whenever possible.

3.10 For Loops

"For loops" statements are used in Verilog HDL, but they do not build combinational logic circuits so, from that perspective, do not belong in this section. However, "for" statements or "for loops" are used in procedures that do build combinational logic. On first reading, the two sentences above may seem contradictory, but they are not. The "for" statement or "for loop" is used to save writing many lines of Verilog HDL statements and at the same time clearly expresses the collective action of the many statements.

For loops are pre-compiled into Verilog HDL meaning. Before the compiler starts the process of converting Verilog HDL to hardware the "for loops" are expanded into the many statements they represent.

The syntax of a for loop is given in Figure 3.21.

The use of a for loop is best illustrated by way of an example. Suppose the task at hand is to construct a vector whose bits are the mirror image of the bits in another vector. A Verilog HDL for doing this is given below:

```
module flip (
input [15:0] data_in,
output reg [15:0] data out
            );
always @ *
```

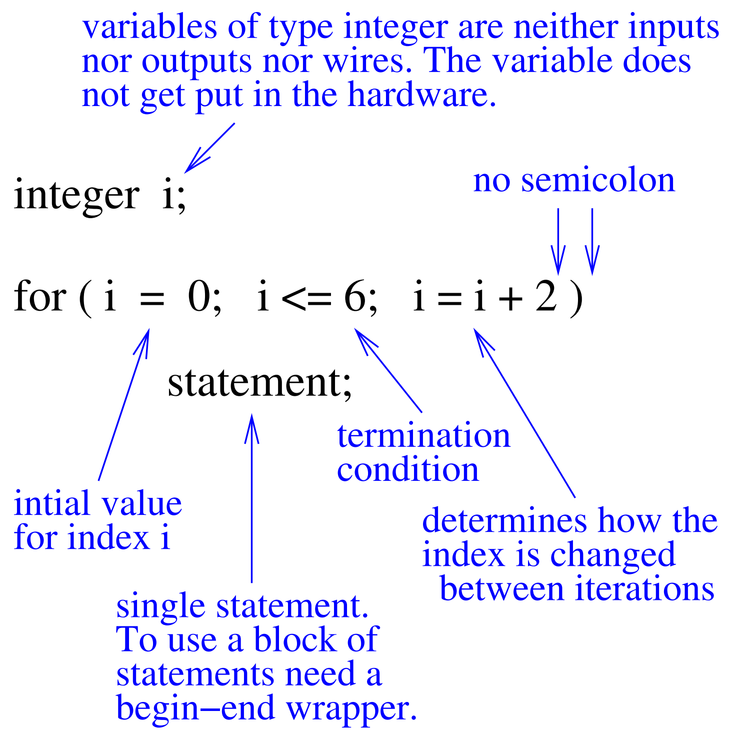
variables of type integer are neither inputs nor outputs nor wires. The variable does not get put in the hardware.
integer i;
no semicolon
for (i = 0; i <= 6; i = i + 2)
statement;
termination condition
intial value for index i
determines how the index is changed between iterations
single statement. To use a block of statements need a begin–end wrapper.

Figure 3.21: The syntax of a for loop

```
   begin
    data_out[15] = data_in[0];
    data_out[14] = data_in[1];
    data_out[13] = data_in[2];
    .
    :
    data_out[0] = data_in[15];
   end
endmodule
```

A for loop can be used to save the writing of the 16 statements in the begin-end wrapper. The Verilog HDL showing how this is done is given below.

```
module flip (
input [15:0] data_in;
output reg [15:0] data out
            );

integer i;
always @ * // do not need a begin-end as for is 1 statement
   for ( i = 0;  i <= 15;  i = i+1 )
      data_out[15-i] = data_in[i];
```

```
endmodule
```

3.11 Sequential Logic

3.11.1 Introduction

Circuits with sequential logic contain flip/flops and therefore, are driven by a clock. Unlike in combinational logic circuits, feedback from the output to the input of a flip/flop does not cause uncontrolled oscillation because of the sampling action of the clock.

Feedback from the output of a flip/flop to its D-input does not cause the output to change immediately. The D-input of the flip gets transferred to the output on the next positive clock edge. Therefore, the output can only change on positive clock edges.

Sequential circuits with feedback generate a sequence with values that changes on positive clock edges. For this reason circuits with flip/flops are called sequential circuits.

3.11.2 Behavioural Description of Sequential Circuits

The Verilog HDL is a behavioural description of a D-flip/flop that has input `d`, output `q` and is triggered by the positive edge of a clock named `clk` is given below.

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

The behavioural description of a D-flip/flop that is triggered by the negative edge of a clock named `clk` is given below:

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

When a signal is used as a clock, as is `clk`, the compiler will issue one or more warnings something to the effect

```
"Found pins functioning as undefined clocks and/or memory enables"
```

or

```
"Timing requirements not met"
```

The first warning is issued to inform the engineer that a signal is being used as an undeclared clock, which usually done by mistake. The second warning is issued to inform the engineer that the optimizer can not work properly without knowing the frequency of the clock.

The circuit will still compile properly, but the compiler would do a better job if the engineer provided the frequency of the clock. This clock information, as well as other timing information, can be provided in a separate file with extension `.sdc`. The .sdc file extension stands for Synopsys Design Constraints.

Synopsys Design Constraints (SDC) is an industry-standard language which is used to inform the Verilog compiler of any specific timing requirements that must be met when the design is running in hardware. The information takes the form of ”timing constraints” that are typically provided to the compiler in a .sdc file.

In order to create a .sdc file, select the menu option File -¿ New... and then choose ’Synopsys Design Constraints file’ from the pop-up window.

Type the following text into the file:

```
create_clock -name my_clk -period 37ns [get_ports clk]
```

Save the file as “project_name.sdc”. The file will be automatically imported into the project and used by Quartus to direct the compilation and implementation process.

The SDC language is very powerful and can be used to constrain designs with complex timing requirements such as input and output interfaces to external hardware components or multiple clocks that are not synchronized from a common source. However, for the designs in this course, an extremely simple .sdc file containing just a single ’create_clock’ command will suffice.

’create_clock’ allows the designer to specify the period of the clock being used to operate the logic within the FPGA, which in turn allows the compiler (Quartus) to ensure that the setup and hold requirements are met for each register in the final hardware design. The specific command provided above creates a clock named ’my_clk’ for the purposes of timing analysis which has a period of 37ns and is associated with the verilog input port ’clk’. Note that the name assigned to the clock constraint (’my_clk’ in the command above) doesn’t particularly matter and is mainly used to identify the clock in timing reports that are produced by Quartus.

Detailed information about the SDC language and the commands that can be placed in a .sdc file is provided in the Quartus SDC and TimeQuest API Reference Manual
(https://www.altera.com/en_US/pdfs/literature/manual/mnl_sdctmq.pdf).

3.11.3 Examples of Sequential Circuits

This is **not** the point where Chapter 3 ends. However it is the point where the type written notes for Chapter 3 give way to the hand written notes.

The continuation begins on page 61 of the 107 page hand written Chapter 3 that can be found on the class website. It has file name `Chapter_3_notes.pdf`. The page number that is hand written in the top right corner of the page is 58b.