LSU EE 4755 **Synthesis Study Guide** Fall 2015

David Koppelman

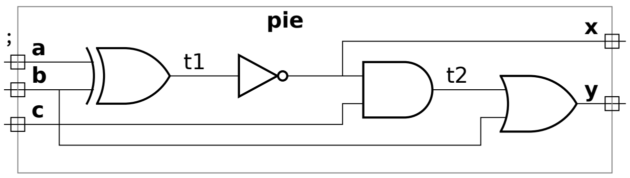
Updated 17 September 2015, 19:17:21 CDT

1.1 Introduction

Synthesis is the converting of a design specified in a hardware description language, such as Verilog or VHDL, into a form suitable for manufacture or into a form that can be loaded to a programmable device such as an FPGA. Synthesis is performed by a synthesis program, or by a collection of synthesis programs. The one used for LSU EE 4755 is Cadence Encounter.

This study guide describes the kind of hardware that one gets for a given piece of Verilog. It supplements the material provided in the course references. Some coverage of synthesis is provided in Brown & Vranesic 3rd Edition in Section 4.6 (for combinational logic) and Section 5.12 (for sequential logic), and these sections are probably the best place for a student unfamiliar with the material to look (other than this guide). The coverage in the Ciletti text is in chapters 5 and 6, but this coverage contains unnecessary detail and is not recommended for a beginning student.

This guide will first provide a brief overview of synthesis, with a level of detail appropriate for EE 4755. The rest of the guide will describe the kind of hardware a typical synthesis programs generates for a given piece of Verilog code. That is, given some Verilog code the guide will show how to sketch a logic diagram of the *inferred* hardware. For example, the Verilog modules below might each be synthesized into the logic shown to the right.



```
// Module is written in Explicit Structural Form
module pie( output wire x, y,  input wire a, b, c );
   wire   t1, t2;

   xor x1(t1,a,b);
   not n1(x,t1);
   and a1(t2,x,c);
   or  o1(y,t2,b);
endmodule

// Module is written in Implicit Structural Form
module pie( output wire x, y,  input wire a, b, c );
   assign x = ~ ( a ^ b );
   assign y = x & c | b;
endmodule

// Module is written in a Synthesizable Behavioral Style
module pie( output logic x, y,  input wire a, b, c );
   always_comb
     begin
        x = ~ ( a ^ b );
        y = x & c | b;
     end
endmodule

// Module is written in Implicit Structural Form
module pie( x, y, a, b, c );
   input wire a, b, c;
   output wire x, y;

   assign x = ~ ( a ^ b );
   assign y = x & c | b;
endmodule

// Module is written in a Synthesizable Behavioral Style
```

```
module pie( x, y, a, b, c );
   input wire a, b, c;
   output logic x, y;

   always @( a or b or c )
     begin
        x = ~ ( a ^ b );
        y = x & c | b;
     end
endmodule
```

Determining the hardware that will be synthesized for a given piece of Verilog is an important skill since the cost and performance of a circuit is determined by what the synthesis program generates.

One might wonder: Why is this hard? Afterall, Verilog is a hardware *description* language. Actually, Verilog is two languages in one. Verilog written in an *explicit structural form* describes exactly how components are interconnected. So the synthesis program (at least after the initial inference step, explained below) generates exactly what it read. But Verilog is also a simulation language, in which you can describe what hardware will do by writing *behavioral code*. Though behavioral code describes what a circuit will do, it does not describe what components are needed to do it. The last module above uses behavioral code.

The Verilog language describes exactly what behavioral code should do during simulation. The language says almost nothing about what kind of hardware corresponds to the behavioral code.

All this suggests that one should use explicit structural code for designing hardware and use behavioral code for writing *testbenches* (a module that tests other modules by providing inputs to those modules and verifying that their outputs are correct). The problem is that explicit structural code is tedious to write, and so synthesis programs have been developed which can synthesize hardware from behavioral code. They do this by *inferring* the hardware that corresponds to some piece of Verilog behavioral code.

Current synthesis programs cannot infer hardware for *any* behavioral code. The behavioral code must follow certain rules, and we will call such code *synthesizable behavioral code*. These rules are defined by the synthesis programs, they are not part of the Verilog standard. Each synthesis program has different rules but there is a great deal of commonality.

This guide will describe rules which work for the Cadence Encounter synthesis program and the rules should work for many other synthesis programs.

1.2 Overview and Terminology

For purposes of designing hardware Verilog (and other HDLs including VHDL) code is written in different *styles*. A style is a set of rules specifying how the code can be written. In the *explicit structural style* code cannot contain behavioral code (code following an `always` or `initial`) and it cannot contain continuous assignments (`assign` statements). Such modules only consist of instantiations of other modules and primitives. Code in the *implicit structural style* cannot contain behavioral code but can contain continuous assignments. Code in a *synthesizable behavioral style* can contain behavioral code but must follow certain rules, which are described in this guide.

A *primitive gate* is a gate recognized by Verilog. Examples include `and`, and `xor`. A *technology module* is a module which has been identified as a component in the target technology. For example, `INVX1`, is a NOT gate defined in the OAU technology library.

Two important kinds of programs for reading HDL code are *simulators* and *synthesis programs*. The Verilog code tells the simulator how to determine the values on wires (and registers, etc) over time, and it also directs the simulator to print messages, write files, and so on. Simulators are run to verify the correctness of the design. A synthesis program reads the HDL and applies several *passes* to convert the HDL into a manufactureble form.

The following is a simplified description of synthesis, describing the steps which we will focus on in this class.

The starting point for synthesis is an HDL model of the design (which might consist of a file or files of Verilog code) which has already been debugged using simulation tools.

The first step is *inference*. In the inference step the HDL is converted to an explicit structural form. In explicit structural form the design uses only primitive gates, technology modules, or modules that themselves are in explicit structural form. For the `pie` example above inference is a straightforward process, but for other code things aren't so simple. Understanding how inference works is important and the primary purpose of this study guide.

The next step is *optimization*. The goal of the optimization step is to reduce the area of the design, improve the speed of the design, or achieve some other goal. Optimization may be performed multiple times. It is typically done initially after inference, and again after subsequent steps that modify the design, such as after technology mapping (see below).

After optimization the synthesis program will perform *technology mapping*. The goal of technology mapping to replace all primitive gates with technology modules. A technology module is a component that can be manufactured on the chip, a programmable part of an FPGA, or something similar.

Following technology mapping another optimization step may be performed.

The last step (or steps, but we'll consider it one step) is *place and route*. In place and route the synthesis program (usually a separate place and route program) first decides where on the chip to place the technology modules. Then it decides how to run connections between them.

After place and route the design could be sent to fabrication, but before doing that post-synthesis simulations are performed to verify timing and other characteristics. The post-synthesis simulation uses timing data provided by the synthesis program which is based on the capabilities of technology modules taking into account the impact of fan-out and the length of the wiring between the modules.

EE 4755 students need to know the definitions of the terms described above, that's the easy part. They also need to be able to look at a Verilog description and determine the hardware that would be inferred, and how the hardware might be simplified after optimization. That's the interesting part.

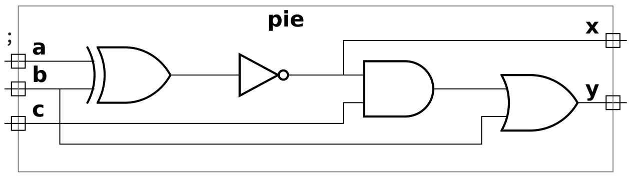
1.3 Synthesis of Simple Expressions in Implicit Structural Code

Inference of code in implicit structural form is easiest to understand because things declared as `wire` are wire in the synthesized design and Verilog operators map to hardware in a straightforward way. The Boolean operators are the simplest, since they map to primitive gates. For arithmetic operators, such as `+`, synthesis programs will use modules from a library (usually provided with the synthesis software). The conditional operator synthesizes into a multiplexor.

1.3.1 Inference of Boolean Operators

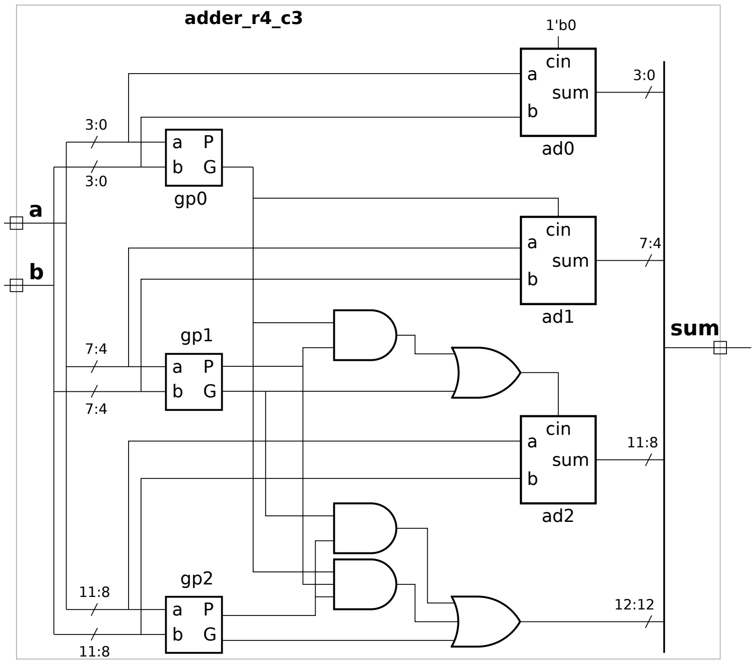
Boolean operators are inferred to the respective primitive gates. In the example below the `^` operator maps to an XOR, the `~` to a NOT gate, etc. Notice that `t1` is declared as a wire in the module and appears as a wire in the inferred hardware. (Things aren't so simple in behavior code with variables declared `logic`.)

```
// Module is written in Implicit Structural Form
module pie( output wire x, y,  input wire a, b, c );
   assign x = ~ ( a ^ b );
   assign y = x & c  b;
endmodule
```



The module below includes both instantiations and implicit structural logic. The illustration for the inferred hardware shows the instantiated modules as boxes even though hardware would be inferred for them too. A test or homework question should specify whether to show the logic inside of instantiated modules. For the example below the problem might read: *Show the hardware that a synthesis program would infer for the module below. Show the instantiated modules as boxes.*

```
module adder_r4_c3(sum,a,b);
   input wire [11:0]  a, b;
   output wire [12:0] sum;
   wire [2:0]  P, G, carry;
   wire [2:0]  CO; // Unused.

   ripple_4_block ad0(sum[3:0],  CO[0], a[3:0],  b[3:0],  carry[0]);
   ripple_4_block ad1(sum[7:4],  CO[1], a[7:4],  b[7:4],  carry[1]);
   ripple_4_block ad2(sum[11:8], CO[2], a[11:8], b[11:8], carry[2]);

   gen_prop_4 gp0(G[0], P[0], a[3:0],  b[3:0]);
   gen_prop_4 gp1(G[1], P[1], a[7:4],  b[7:4]);
   gen_prop_4 gp2(G[2], P[2], a[11:8], b[11:8]);

   assign      carry[0] = 1'b0;
   assign      carry[1] = G[0];
   assign      carry[2] = G[0] & P[1] | G[1];
   assign      sum[12] = G[0] & P[1] & P[2] | G[1] & P[2] | G[2];
endmodule
```



1.3.2 Inference of Arithmetic Operators

Arithmetic operators include the four basic operations, as well as comparisons. That is, the comparison in `a == b` is considered an arithmetic operator. When a synthesis program encounters an arithmetic operator it will substitute (or infer) a corresponding module from a library of arithmetic operators. Most synthesis programs come with such a library, and many let you substitute your own modules. The substituted modules, of course, must be synthesizable.

For our show-the-synthesized hardware problems it will be sufficient to show boxes for the modules performing arithmetic operations (see the `borc` example below).

The synthesis program too might leave the arithmetic modules as module instantiations (as opposed to showing, say, all the gates that make up the adder). This would make it easier for the synthesis program to optimize circuits containing multiple arithmetic operations.

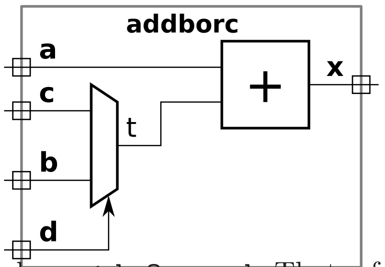
1.3.3 Inference of the Conditional Operator (`c ? a : b`)

The synthesis program will infer a multiplexor for the conditional operator. An expression of the form `c ? a : b` will synthesize to a two-input multiplexor, with one input connected to `a` and the other to `b`, the control input will connect to `c`.

When drawing diagrams of the inferred hardware we will show the multiplexor as a multiplexor. As with arithmetic operations, this is both for clarity and as a reminder that the synthesis program too might not replace these modules with gates until after it has made an optimization pass.

Here is a module using the conditional operator and addition:

```
module addborc
 ( output wire [15:0] x,
   input wire [15:0] a, b, c,
   input wire d );

   wire [15:0] t = d ? b : c;
   assign x = a + t;
endmodule
```



Often the conditional operator includes a comparison, for example, `a < b ? c : d`. That, of course, is handled by connecting `a` and `b` to a comparison unit.

1.4 Synthesis of Combinational Logic from Behavioral Code

This section describes how combinational logic is inferred from behavioral Verilog code. The Verilog code must be in a synthesiable form suitable for combinational logic, that form is described in the next section. That is followed by a description of the kind of hardware that is inferred for various pieces of code.

1.4.1 Restrictions on Verilog Code to Synthesize Combinational Logic

Behavioral code following the rules below (and additional rules, because the list is not complete) will synthesize into combinational logic.

- The code must be in an `always` or `always_comb` block.
- The sensitivity list must not contain a `posedge` or `negedge`.
- The sensitivity list must explicitly list all referenced live-in variables, or consist of a `*` which does the same thing.
- A variable must either be always assigned in a block, or never assigned in a block.

There are other restrictions which apply to things not covered in class, such as `wait` statements or the use of event controls, like `@`, within a block. Additional restrictions will be given in the sections that follow.

The module below follows these rules:

```
// Will synthesize to combinational logic.
module pie( x, y, a, b, c );
   input wire a, b, c;
   output logic x, y;
   always @( a or b or c )
     begin
        x = ~ ( a ^ b );
        y = x & c | b;
     end
endmodule
```

Here is how the code above follows the rules: The behavioral code is in an `always` block. (As opposed to an `initial` block.) The sensitivity list (the stuff after the `@`) does not have an edge trigger, and does list each live-in variable that has been used, `a`, `b`, and `c`. (The variable `x` is referenced but it is not live in because its value is assigned before it is used.) Both `x` and `y` are always assigned.

In contrast, the module below breaks some of these rules and so will not synthesize to combinational logic:

```
// Will NOT synthesize to combinational logic.
module pie( x, y, a, b, c );
   input wire a, b, c;
   output logic x, y;
   always @( a or b )
     begin
        x = ~ ( a ^ b );
        if ( c ) y = x & c | b;
     end
endmodule
```

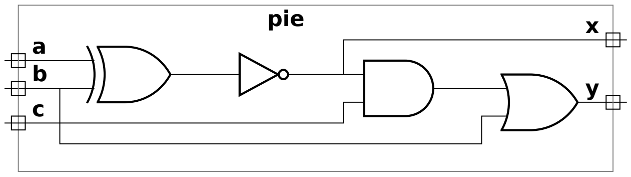
Here is how the code above violates the rules: The sensitivity list omits `c`. The variable `y` is only sometimes assigned. The synthesis program might reject this code for the incomplete sensitivity list. Even if that were corrected, the conditional assignment would result in a latch being used for `y`.

1.4.2 Inference of Simple Assignments, and the Handling of Var (non-net) Type Objects

For procedural code consisting of assignments, such as the code below, the method to determine the inferred hardware is the same as the method used for implicit structural code.

Consider again the behavioral `pie` module below.

```
module pie( x, y, a, b, c );
   input wire a, b, c;
   output logic x, y;

   always @( a or b or c )
     begin
        x = ~ ( a ^ b );
        y = x & c | b;
     end

endmodule
```



Notice that `x` and `y` are declared `logic` but they appear as wire. In this case there is exactly one wire for each `logic`. If there are loops then a `logic` can synthesize into multiple wires.

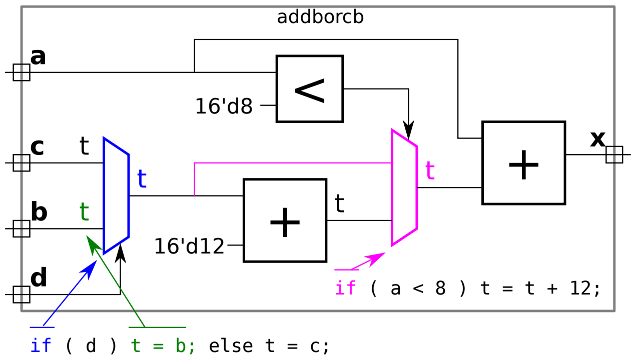
1.4.3 Inference of `if` Statements

One or more multiplexors will be inferred for an `if` statement in behavioral code. To be exact, there will be one multiplexor for each distinct register assigned in the `if` or `else` part. One mux input is for the value produced in the `if` part, the other is for the value produced in the `else` part. If no value is produced in either the `if` or `else` part then the value before the `if` is used.

In the example below there are two `if` statements. In the first `if` statement, `if ( d ) t = b; else t = c;`, `t` is assigned in both the `if` and `else` parts, and so both multiplexor inputs come from value assigned inside the respective parts. Notice that the statement creates three versions of `t`, one each in the `if` and `else` statements, and a third version at the output of the multiplexor. In the synthesized hardware each of these is a separate wire.

In the second `if` statement there is no `else` part (which is the same as an else part in which `t` was not assigned). So for the second mux the condition-false (top) input to the mux is the "old" value of `t`. Also notice that a comparison unit is used to provide the condition for the mux.

```
module addborcb
   ( output logic [15:0] x,
     input wire [15:0] a, b, c,
     input wire d );

   logic [15:0] t;

   always @* begin
      if ( d ) t = b; else t = c;
      if ( a < 8 ) t = t + 12;
      x = a + t;
   end
endmodule
```



1.4.4 Inference of `case` Statements and Multiple `if` Statements

For a case statement with consecutive case constants the synthesis program will infer a multiplexor. For an example, see the `imult_ord_radix_4` in the sequential logic section below.

Though each `if` statement will be inferred as a two-input multiplexor, under the right conditions an optimization pass can combine these into a single multiplexor.

1.4.5 Inference of Iteration Constructs (Loops)

Iteration constructs include `for` and `repeat`. For these to be synthesizable the number of iterations must be an elaboration-time constant.

For example, `for( int i=0; i<10; i++)` is fine. However the example below won't synthesize because the synthesis program does not know the value of `myluckynumber`.

```
// Won't synthesize because of myluckynumber
module simple(x,myluckynumber,a);
   input wire [7:0] myluckynumber;
   input wire [15:0] a;
   output logic [15:0] x;

   always @* begin
      x = 0;
      for ( int i=0; i<myluckynumber; i++ ) x = x + a;
   end
endmodule
```

If you need to synthesize something like the module above and there is a reasonable maximum to the number of iterations, you can guard the loop body with an `if`:

```
// Will synthesize.
module var_iter(x,myluckynumber,a);
   input wire [7:0] myluckynumber;
   input wire [15:0] a;
   output logic [15:0] x;
   localparam int MAX_LUCKY = 15;

   always @* begin
      x = 0;
      for ( int i=0; i<MAX_LUCKY; i++ ) if ( i < myluckynumber ) x = x + a;
   end
endmodule
```

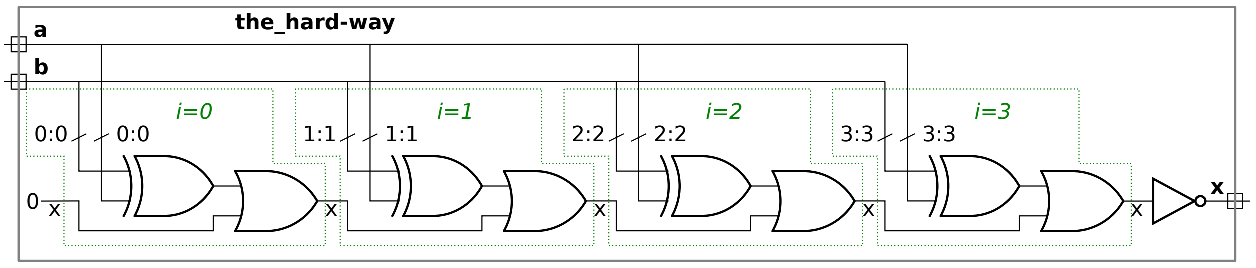
To determine the what hardware will be inferred for a loop just duplicate the loop body by the number of iterations, and make the loop variable (`i` in the example above) a constant in each body. (Duplicating the body of a loop is called *loop unrolling* and is an important technique in code optimization which you might see in other courses.) For example, consider:

```
module the_hard_way( output logic x, input wire [3:0] a, b );

   always_comb
     begin
        x = 0;
        for (int i=0; i<4; i++) x = x | ( a[i] ^ b[i] );
        x = ~x;
     end
endmodule
```

The number of iterations is 4. So to the synthesis program this is equivalent to:

```
module the_hard_way_unrolled( output logic x, input wire [3:0] a, b );
   always_comb
     begin
        // Note: This is not something you will actually see because
        //       the synthesis program unrolls the loop, not you.
        x = 0;
        x = x | ( a[0] ^ b[0] );
        x = x | ( a[1] ^ b[1] );
        x = x | ( a[2] ^ b[2] );
        x = x | ( a[3] ^ b[3] );
        x = ~x;
     end
endmodule
```

The inferred hardware (before optimization) will be:



Here are some important things to observe about this example:

- The variable `i` has been replaced by constants, in this example the bit positions for `a` and `b`.

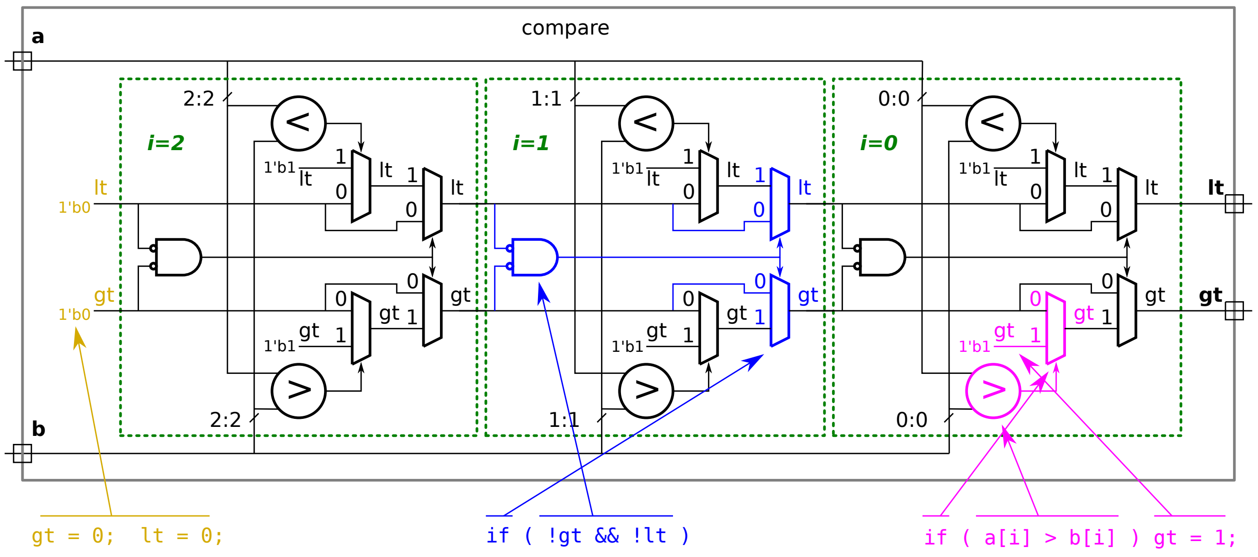
- For each time `x` was assigned a wire was synthesized.

- This module is practical because the number of iterations was small. If it were, say, 100,000, the cost of the hardware would probably be too high.

- Some optimizations can be applied, most obviously eliminating the leftmost OR gate.

The module below compares two signed numbers one pair of bits at a time, starting from the most-significant bit. The behavioral code should be reasonably easy to follow, but the inferred hardware in the illustration that follows might be considered a mess. Mess or not, it's ripe for optimization.

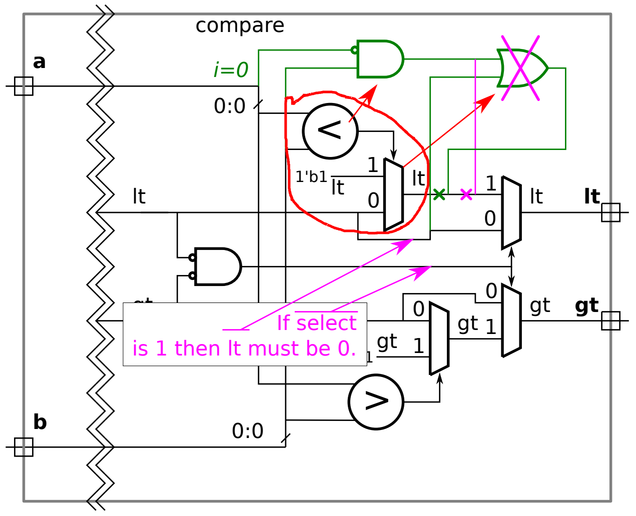
```
module compare( output logic gt, lt,  input wire [2:0] a, b );
   always @* begin
      gt = 0; lt = 0;
      for ( int i=2; i>=0; i-- )
        if ( !gt && !lt ) begin
           if ( a[i] < b[i] ) lt = 1;
           if ( a[i] > b[i] ) gt = 1;
        end
   end
endmodule
```



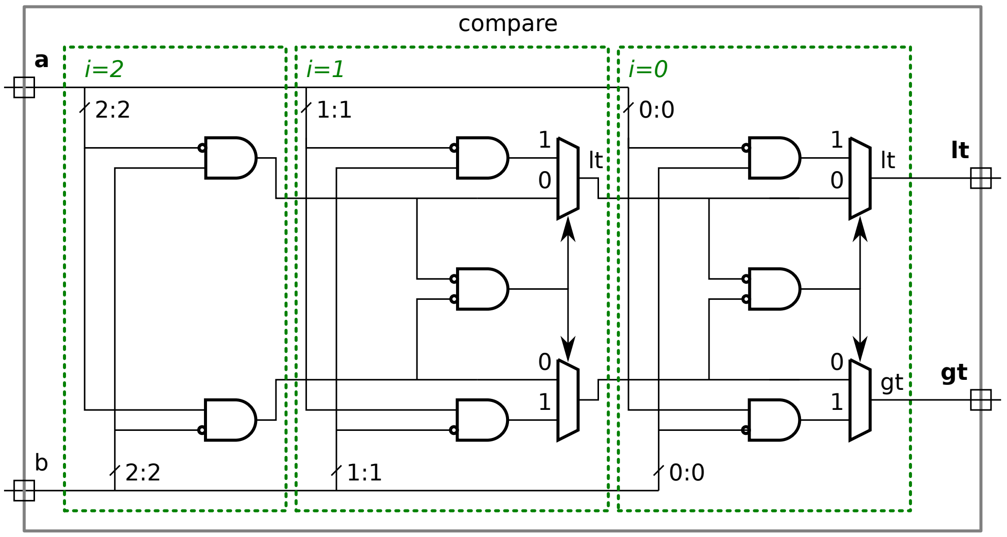
The initial assignments to `gt` and `lt` are synthesized as wires driven by constant values, that's shown in gold above. Each of the three `i` loop iteration bodies is shown in a green box. An entire loop body is guarded by an `if (!gt && !lt )`, that's shown in blue for the second iteration: if the condition is false the values of `lt` and `gt` pass through unchanged to the next iteration or to the module outputs. Within an iteration `gt` is set to `1` if the $>$ comparison is true, that's shown in purple for the last iteration.

Test your understanding by tracing the changes in `lt` for some sample numbers, both in the Verilog code and in the diagram of the inferred hardware.

Because the loop body operates on one-bit quantities many optimizations can be applied. Notice that the `<` and `>` comparisons can be replaced by AND gates with one input inverted, the replacement for the `<` is shown in the diagram below. The mux used for the `if ( a[i] < b[i] ) lt = 1;` comparison can be replaced by an OR gate. (If you don't see it draw a truth table.) These two optimizations appear in green. The OR gate can itself be eliminated, that's shown in purple. It can be eliminated because if the select signal for the mux is 1, choosing the OR-gate output, the value of `lt` must be zero.



The completed hand-optimized module is shown below:



1.5 Synthesis of Sequential Logic

Sequential logic (logic which includes a clock and edge-triggered registers or flip flops) is inferred for behavioral Verilog in the following form:

- The code must be in an `always` or `always_ff` block.
- The sensitivity list must contain a `posedge` or a `negedge` and nothing else.

There are other restrictions which apply to topics not covered in class, such as `wait` statements or the use of event controls, like `@`, within a block.

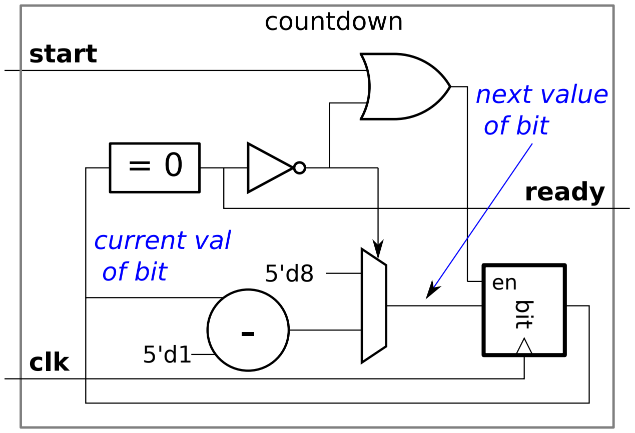
For code in this form an edge-triggered register will be inferred for each variable assigned in such an `always` block. An inferred register will be removed during optimization if the corresponding variable is not *live out* with respect to the block.

A variable is considered *live out* with respect to a block is the value assigned in the block is used outside the block. In module `imult_ord_radix_4` (further below) variables `im` and `pp` are not live out, but `bit` and `product` are live out.

If a variable is conditionally assigned then either an enable signal or a multiplexor will be used.

The example below shows a simple count down timer. Variable `bit` is live out because it is used to determine the value of `ready` and as an `if` condition, and so a register is inferred for it. Because it is not always assigned an enable signal is generated. The inferred logic for the if/else chain has simplified from two multiplexor to one, this is possible because of the enable signal.

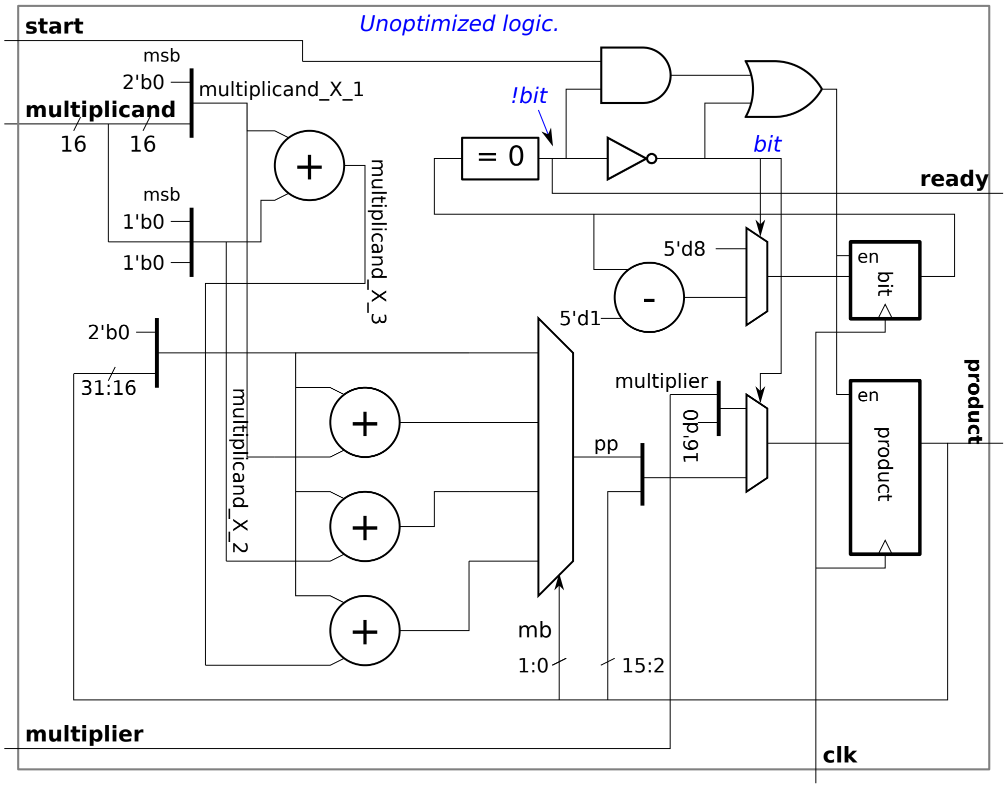
```
module countdown(ready,start,clk);
   input wire start, clk;
   output wire ready;
   logic [4:0]     bit;

   assign        ready = bit == 0;

  // Needed for simulation, ignored for synthesis.
   initial bit = 0;

   always @( posedge clk )
      if ( start ) begin
         bit = 8;
```



```
      end else if ( bit != 0 ) begin
         bit = bit - 1;
      end

endmodule
```

The example below is more complex, but follows the same rules. Notice here that variables `mb` and `pp` are assigned but because neither is live-out, no registers are synthesized for them.

```
module imult_ord_radix_4(product,ready,multiplicand,multiplier,start,clk);

   input wire [15:0] multiplicand, multiplier;
   input wire start, clk;
   output logic [31:0] product;
   output wire      ready;

   logic [4:0]     bit;
   assign        ready = !bit;

   // cadence translate_off
   initial bit = 0;
   // cadence translate_on

   wire [17:0]   multiplicand_X_1 = {2'b0,multiplicand};
   wire [17:0]   multiplicand_X_2 = {1'b0,multiplicand,1'b0};
   wire [17:0]   multiplicand_X_3 = multiplicand_X_2 + multiplicand_X_1;
   logic [17:0]    pp;  // Partial Product
   logic [1:0]     mb;  // Multiplier Bits

   always @( posedge clk )
     if ( ready && start ) begin
        bit     = 8;
        product = { 16'd0, multiplier };
     end else if ( bit ) begin
        mb = product[1:0];
        case ( mb )
          2'd0: pp = {2'b0, product[31:16] };
          2'd1: pp = {2'b0, product[31:16] } + multiplicand_X_1;
          2'd2: pp = {2'b0, product[31:16] } + multiplicand_X_2;
          2'd3: pp = {2'b0, product[31:16] } + multiplicand_X_3;
        endcase
        product = { pp, product[15:2] };
        bit     = bit - 1;
     end
endmodule
```



Unoptimized logic.
start
msb
2'b0
multiplicand_X_1
multiplicand
16
16
msb
1'b0
1'b0
multiplicand_X_3
!bit
= 0
bit
ready
5'd8
-
5'd1
en
bit
2'b0
31:16
multiplier
16'd0
pp
en
product
product
multiplicand_X_2
mb
1:0
15:2
multiplier
clk