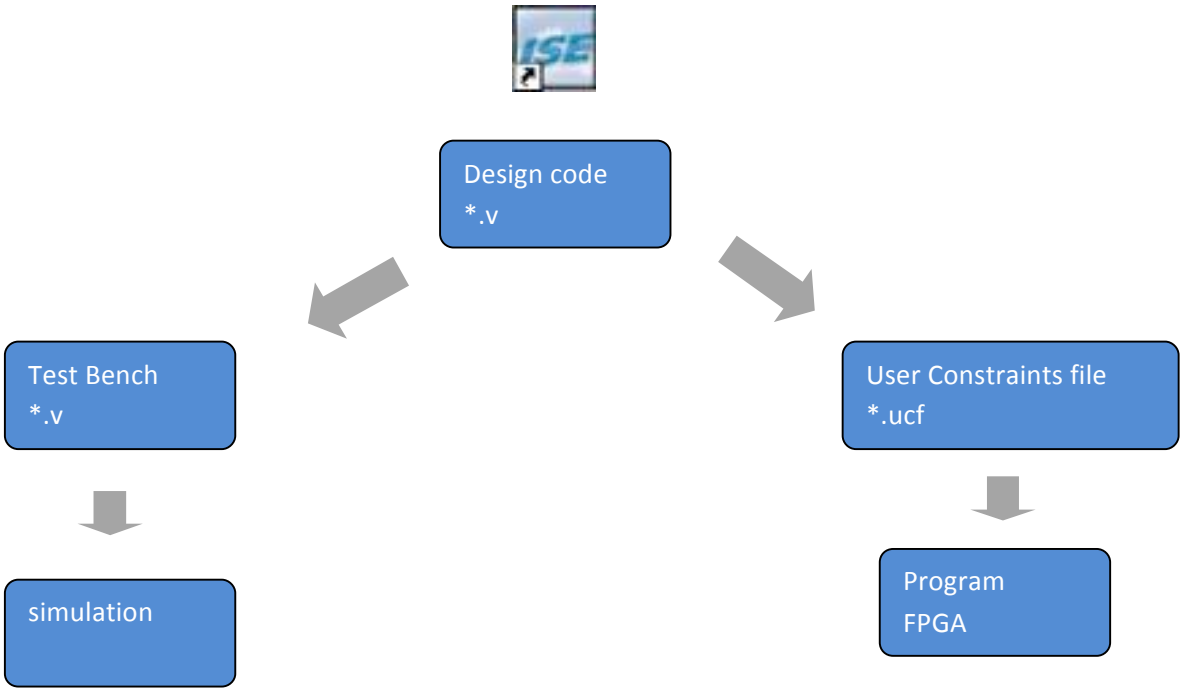
IntroGuide to FPGA's Design Flow



Software WebPack Xilinx

Xilinx provides an integrated software environment (ISE) that can be used to program the FPGA's (field-programmable gate arrays) like the Spartan-3 which we'll use in lab. It has a smart text editor for Verilog, a simulator, a compiler, and will create the .bit-file that we'll use to burn the FPGA.

The help menu within ISE is quite helpful and it is recommended that you use it.

Before you start **Fill in the Table on the right**

A	B	OR	AND	XOR
0	0			
0	1			
1	0			
1	1			

Exercise 1 Combinational Logic: Open Xilinx-ISE Design Suite→ ISE Design Tools → 32-bit Project Navigator

Close any open programs (File → Close Project)

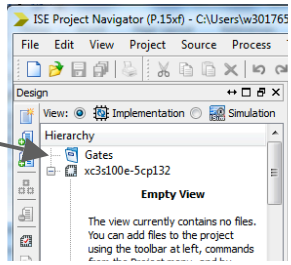
Open a new Project (File → New Project)

Call the project Gates, and save it in a folder with your name, and press *Next*

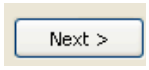
Make sure that the properties are the same as these and press *NEXT* and then *FINISH*

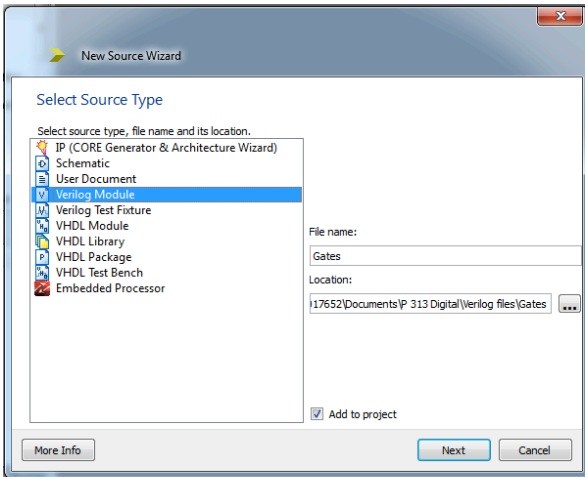
Property Name	Value
Evaluation Development Board	None Specified
Product Category	All
Family	Spartan3E
Device	XC3S100E
Package	CP132
Speed	-5
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	Verilog
Property Specification in Project File	Store all values
Manual Compile Order	☐
VHDL Source Analysis Standard	VHDL-93

Right-click on the project name and select *New Source*

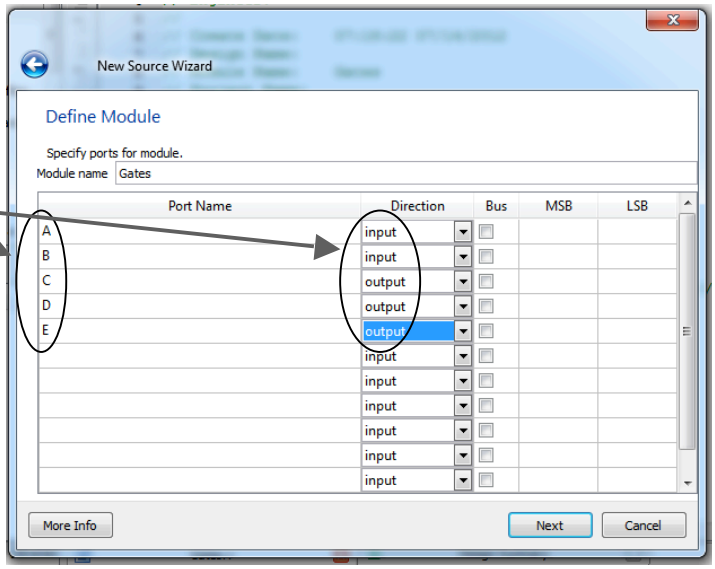


Choose “Verilog Module,”call it Gates and press *Next*

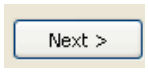


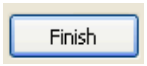


Give it 2 inputs (A and B) and three outputs (C, D, and E)



Click through all of the other options
(Next, Finish)





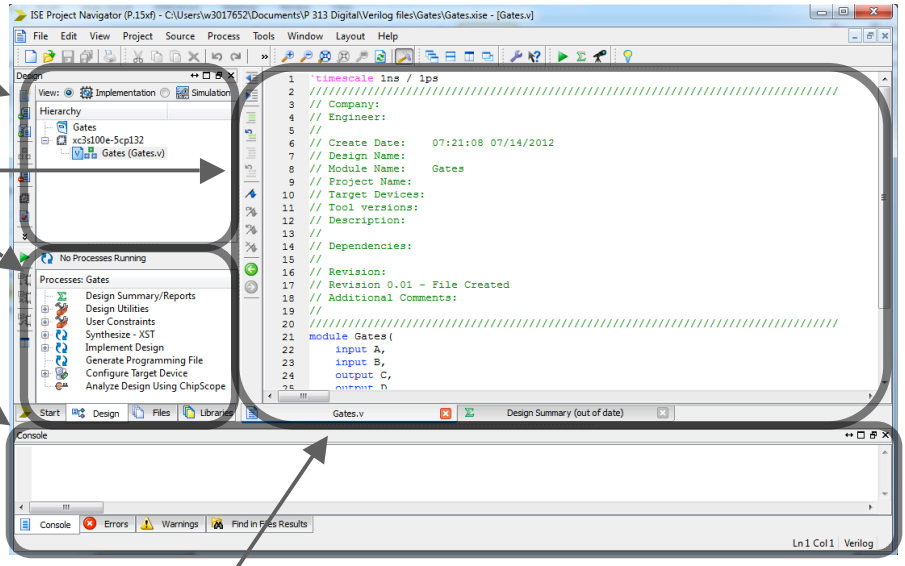
Notice that the screen that is divided into four parts.

Hierarchy

Processes

Main window

Console



To see your Verilog program, click the Gates.v tab at the bottom of the main window.

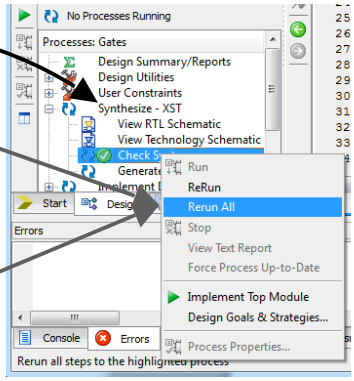
```
20  ////////////////////////////////
21  module Gates(
22      input A,
23      input B,
24      output C,
25      output D,
26      output E
27      );
28
29    assign C = A | B; //A OR  B
30    assign D = A & B; //A AND B
31    assign E = A ^ B; //A XOR B
32
33  endmodule
```

Add these three lines to Gates.v

In the *Processes* panel, choose *Synthesize-XST* right-click on *Check Syntax* and select *Rerun All*. It will ask if you want to save the unsaved files. Click *YES*

If *Check Syntax* doesn't appear in the *Processes* panel make sure Gates.v is selected in the *Sources* panel

If there is any problem with the syntax a note will appear in the *Console* panel below



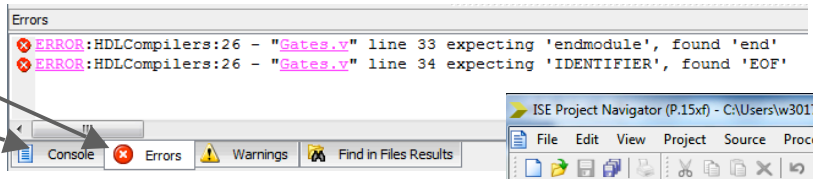
In your program, change "endmodule" to "end module" to introduce an error on purpose

Under *Check Syntax*, right-click and select *Rerun All*,

In the *Console Panel*, choose the *Error* tab

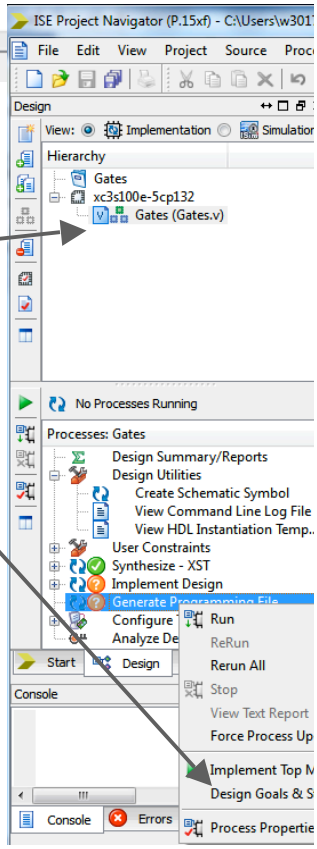
Read the errors

Change the command back to "endmodule"



SELECT JTAG CLOCK INSTEAD OF CCLK

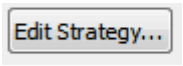
JTAG (Joint Testing Action Group) and CCLK (Composite Clock) are two IEEE standards for coordinating timing between devices. The default setting for WebPack is CCLK and the default for the Digilent Boards is JTAG. If you don't make them consistent then you'll get a warning message when you program the FPGA. It's never caused a problem for me but the warning get annoying, so if you want to avoid it, change the setting on WebPack to JTAG. You only need to make this change once. Here's how:



Make sure Gates.v is selected

Right click on *Generate Programming File* and select *Design Goals & Strategies…*

Select the Edit Strategy button

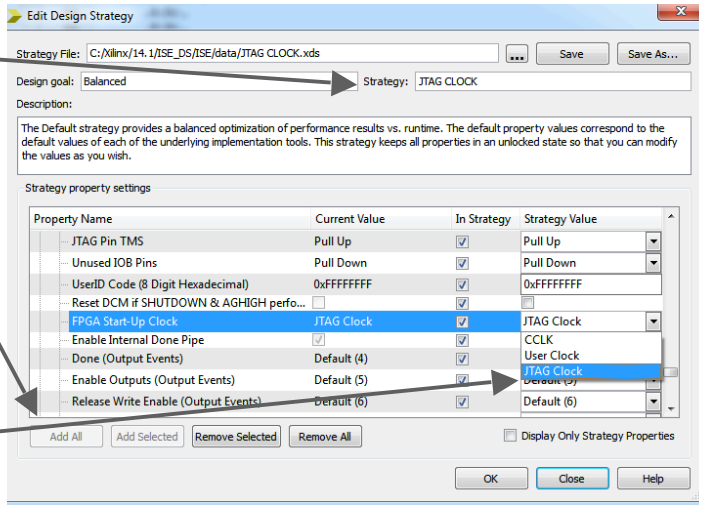


Name your strategy JTAG CLOCK

Select the Add ALL button

If it's grayed out you've probably already added all of the properties.

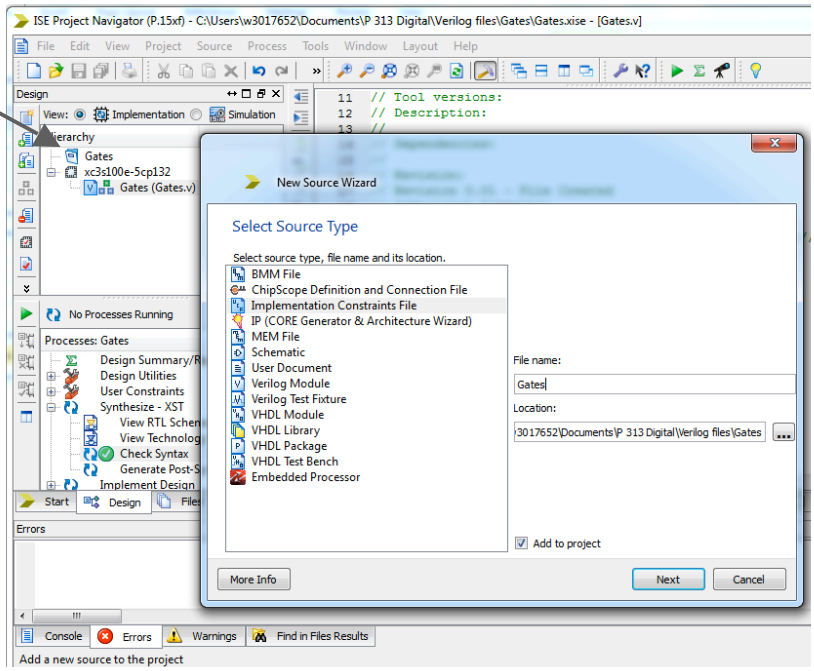
Scroll most of the way down and change CCLK to JTAG Clock



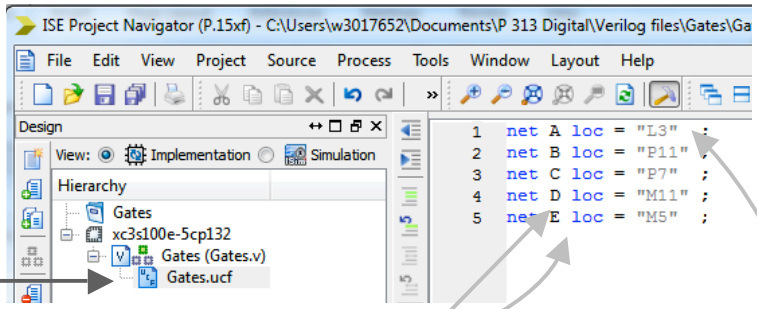
Hit Save and then OK and then make sure that JTAG CLOCK is selected as your preferred design goal strategy.

Assigning Pins

Right-click on *Gates.v* and choose *New Source -> Implementation Constraints file* and name it *Gates* and then hit *Next* and *Finish*



Double-click your new .ucf (user constraints file)

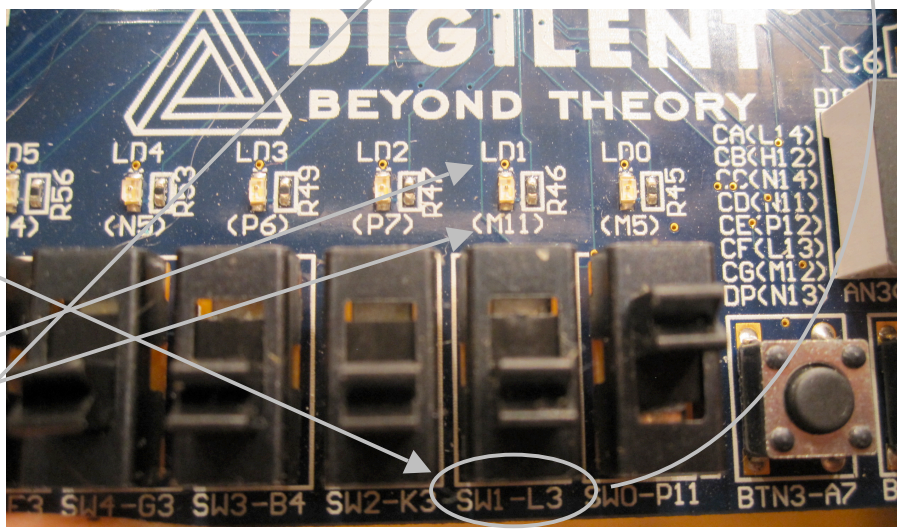


And enter this text in the main window

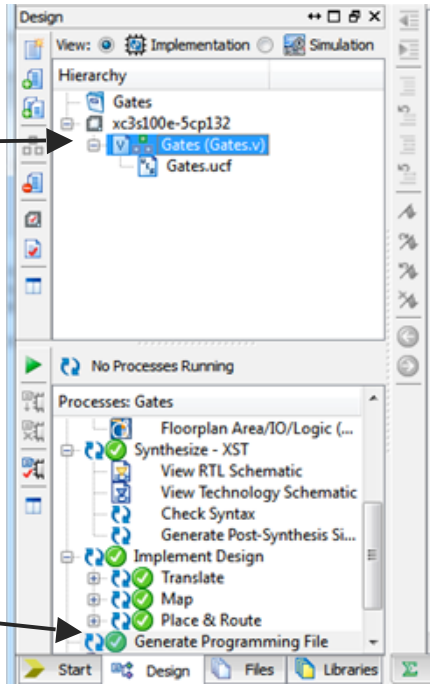
Note: Verilog is a case sensitive language
When it comes to variable names
NAME ≠ Name ≠ name
However, the .UCF file is not case sensitive

Verify that the board name (*SW1*) matches the FPGA name (L*3)* and also matches the name used in your Verilog module (A)

Also verify that LED#1 (LD1) corresponds to the FPGA pin called M11 which the .ucf file associates with the variable name D



Choose Gates.v

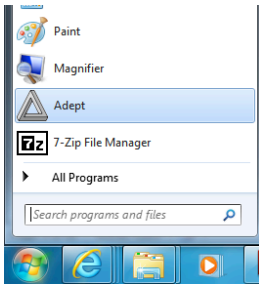


Right-click on *Generate Programming File*
and select *Rerun All.*
This will convert your program from .v to .bit
So that it can be burned onto the FPGA

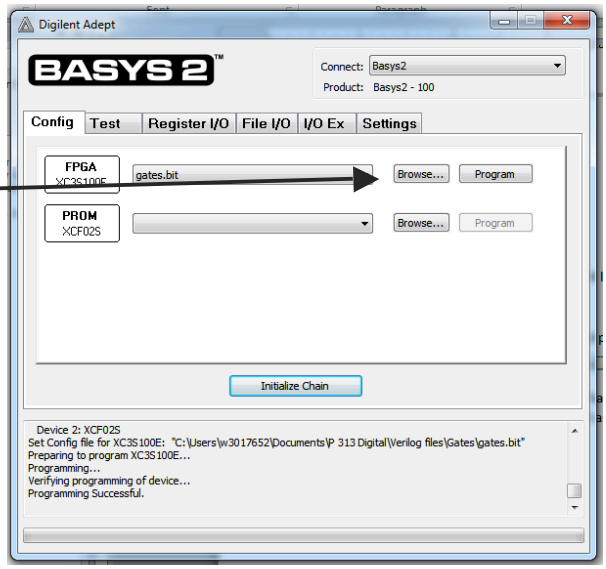
Program your FPGA

Connect your FPGA to the computer, turn it on, and install the drivers automatically if prompted.

Open the program Adept from the start menu.



Browse for your program *Gates.bit,*
(it should be in the folder that you created in part 1) and press
Program

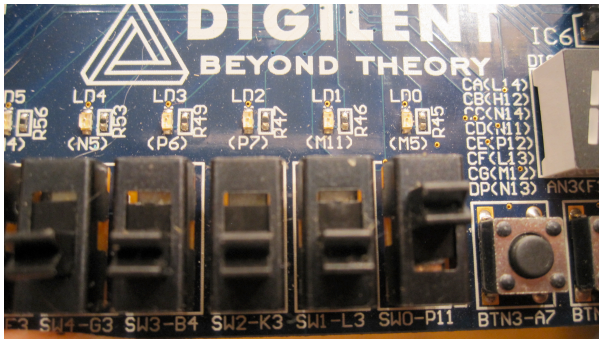


On the physical BASYS 2 Board, activate A and B (the switches SW1 & SW0) and verify that the LED's act correctly:

LD2 should be "on" when SW1 OR SW0 are on

LD1should be "on" when SW1 AND SW0 are on

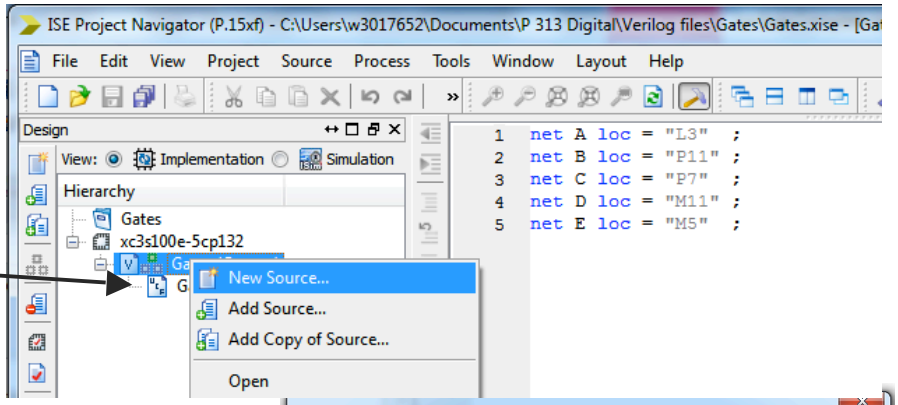
LD0 should be "on" when SW1 XOR SW0 are on (but not when both of them are on)



Create a test bench.

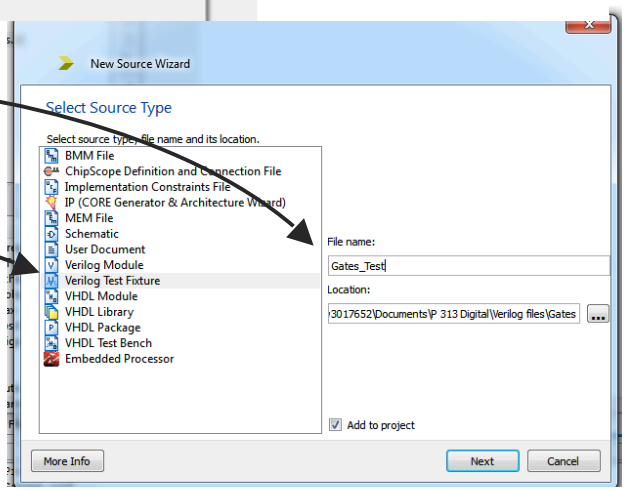
Go back to the ISE window

Right-click on Gates.v and select *New Source.*

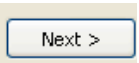


Select Verilog Test Fixture

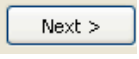
Call it *Gates_Test*
(spaces are not allowed)



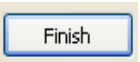
Choose *Next*



Next



and *Finish*:



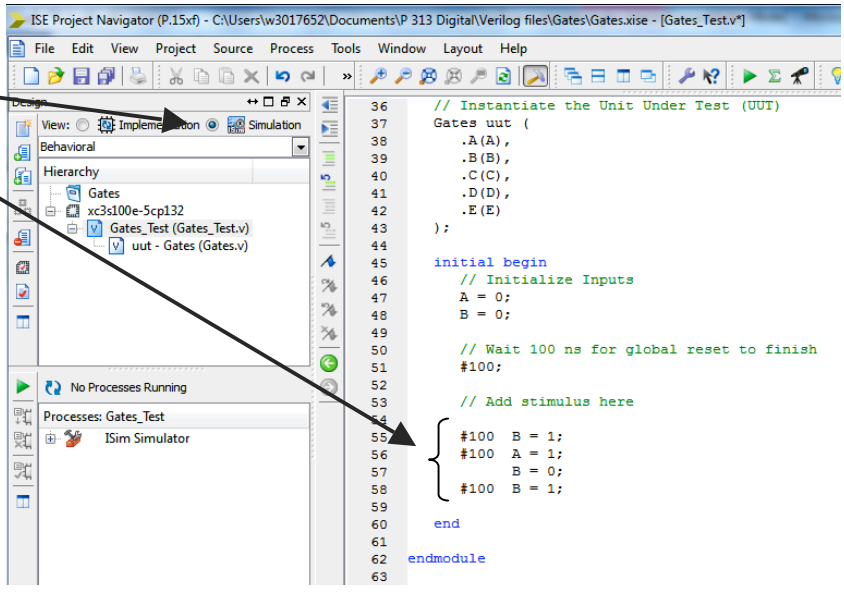
Select the Simulation radio-button

Open *Gates_Test* and add these 4 lines before *end*

This means that the test bench will wait 100 ns, and then assign B a value of 1.

Then it will wait another 100 ns and then put A to 1 and return B to 0;

Then it will wait another 100 ns and put B to 1.



Save the changes.

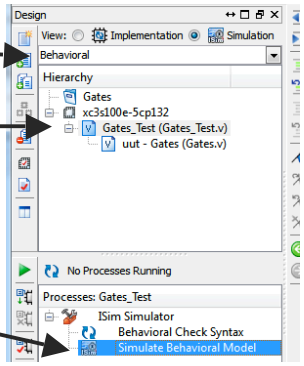
Select *Behavioral* from the drop down menu

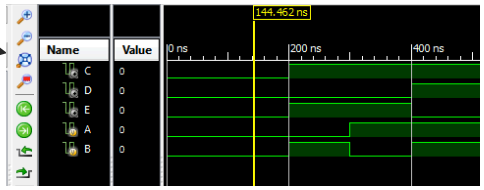
Select *Gates_Test*

Right-Click on *Simulate Behavioral Model*
and select *Rerun All*

When the ISim window opens,
click here to see the whole simulation.

Verify that the outputs make sense.





Exercise 2 Put one module into another

Before you start: Fill in the Truth Tables on the right based on the circuit below.

In this exercise, we are going to create a module (ANDOR) and then use two *instances* of it in a larger module called LOGIC.

A	B	C	F
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

J	K	L	M	N	P	Q
0	0	0	0	0		
0	0	0	0	1		
0	0	0	1	0		
0	0	0	1	1		
0	0	1	0	0		
0	0	1	0	1		
0	0	1	1	0		
0	0	1	1	1		
0	1	0	0	0		
0	1	0	0	1		
0	1	0	1	0		
0	1	0	1	1		
0	1	1	0	0		
0	1	1	0	1		
0	1	1	1	0		
0	1	1	1	1		
1	0	0	0	0		
1	0	0	0	1		
1	0	0	1	0		
1	0	0	1	1		
1	0	1	0	0		
1	0	1	0	1		
1	0	1	1	0		
1	0	1	1	1		
1	1	0	0	0		
1	1	0	0	1		
1	1	0	1	0		
1	1	0	1	1		
1	1	1	0	0		
1	1	1	0	1		
1	1	1	1	0		
1	1	1	1	1		

ANDOR has 3 inputs and one output. LOGIC has 5 inputs and 2 outputs.

Close any open projects and follow the instructions from exercise 1 to create a new project in ISE® called LOGIC.

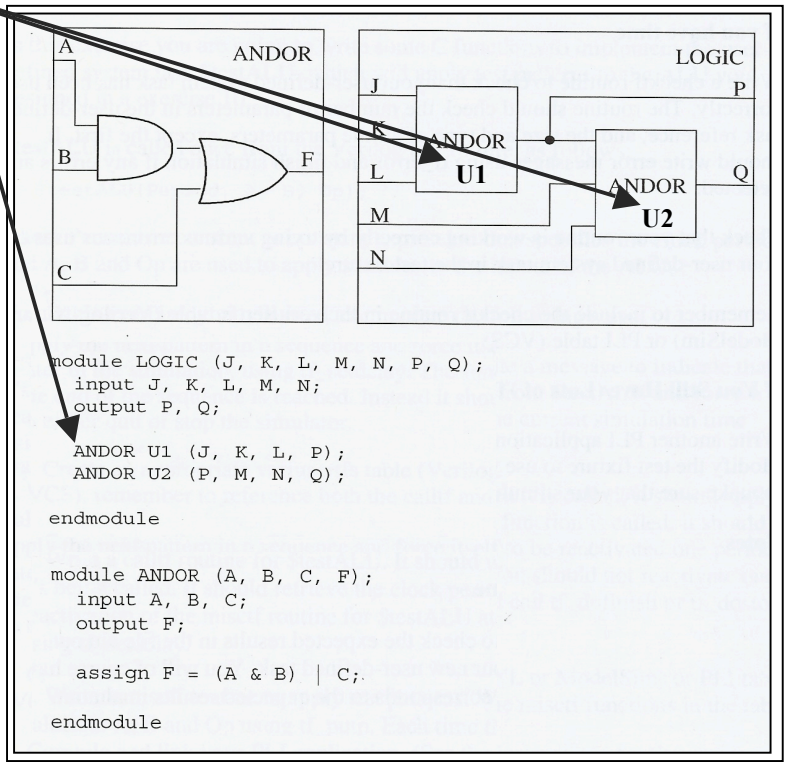
Right-click on the name →Create a new source →Verilog module
Call it LOGIC

Your project should have 5 inputs (J, K, L, M, N) and two outputs(P, Q)

Enter the rest of the program as shown in the box below.

The command *ANDOR U1 (J, K, L, P);* means:
Create an *instance* of the *ANDOR* module within the module *LOGIC*
Call this *instance* U1.
ANDOR has three inputs and one output that are called (A, B, C, F).
Let's connect ANDOR using the names from LOGIG in order. Verify that the connections are the same as the schematics shows.
Check the syntax and save the file.

In the sources panel, note that you can click the + sign on the side of LOGIC to see the two instances of ANDOR.



Two ways to connect modules: ordered mapping or named mapping

In the above example we used ordered mapping to make the connection between the names of ANDOR (which are A, B, C, F) and the names of LOGIC (J, K, L, P) and (P, M, N, Q). We used ordered mapping so J, K L, P are assigned in the same order as A, B, C, D.

The other method is named mapping.

ANDOR U1 (.A(J), .B(K), .C(L), .F(P)); //this means that the output F of instance U1 of the module ANDOR

ANDOR U2 (.A(P), .B(M), .C(N), .F(Q)); //is equal to wire P in Module LOGIC. Verify this with the schematic above.

NOTE: I like using ordered mapping for simple cases. If the modules are more complex it's safer to use named mapping.

Software vs. Hardware.

Now we are going to create a test bench to verify that the program is functioning correctly (just like we did for the Gates module), but before doing so I'd like to talk a little bit about the difference between writing a program in software and coding hardware. We will use the same tool (Verilog) to do both of them, but there are some things that you can do in hardware that you can't do in software and visa versa.

The program LOGIC that we just made is synthesizable, which means that it can be converted into hardware. The test bench that we will write now (LOGIC_TEST), won't be synthesized into hardware but rather it's software that we will use to verify that LOGIC works.

Unlike synthesizable programs, LOGIC_TEST doesn't have inputs or outputs because all of its variables will be generated internally. What's more, we can use commands that don't make sense in hardware, for example, the command *initial* doesn't make sense in hardware. The hardware can't start over, it simply *is*.

Also, we can take certain liberties with software that we can't take with hardware. For example, FOR loops can be synthesized as repeated structures in hardware, but one has to be sure that the limits of the loop are fixed. For example, Verilog can realize this "for" loop

for (i = 0, i<=7, i=i+1)

in hardware by copying the circuit the correct number of times (in this case 8).

On the other hand, although it's ok to use "while" in simulations (software) in general it's not synthesizable and will not work in hardware because there's no way to know how many times it has to run.

while (error_flag == 0)

Create a Test Bench. In the *Hierarchy* panel make sure that the *Implementation* radio button is selected and then right-click on the Verilog program (LOGIC.v) and choose *New Source*. Choose *Verilog Test Fixture* and call it *LOGIC_TEST*. Then push *Next, Next, Finish, Finish*.

Note that *LOGIC_TEST* does not appear in the *Hierarchy* panel when the *Implementation* radio-button is selected. This is because *LOGIC_TEST* isn't a synthesizable program, but rather a test bench. Now choose *Behavioral Simulation* from the drop-down menu in the *Hierarchy* panel, find *LOGIC_TEST*, and right-click on the plus (+) next to it, and verify that the module *LOGIC* is below it (and therefore part of it).

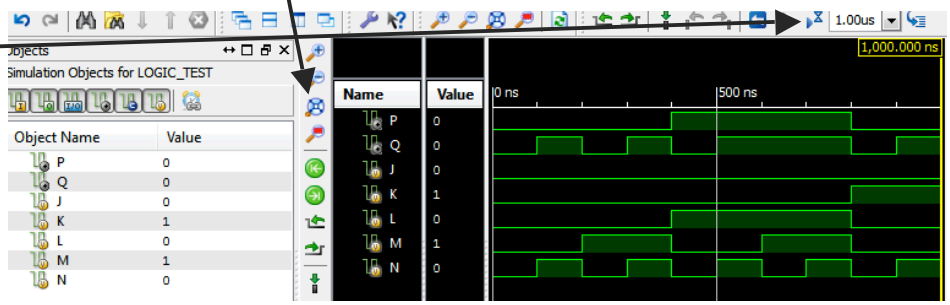
In the main window, click on the *LOGIC_TEST* tab, and note that ISE has created a framework. Add the following command just before *endmodule* at the bottom **always #100 {J, K, L, M, N} = {J, K, L, M, N} + 1;**

This command means, wait 100 units of time (in this case nanoseconds) and then add one to the concatenation of the variables J, K, L, M, N. The symbol {} represents concatenation which means that the program will treat the 5 variables as a single variable with 5 digits. This allows us to cycle through every possible input value so that we can compare the output with the truth table.

In the *Hierarchy* panel verify that the *Simulation* radio button is selected and that *LOGIC_TEST* is highlighted. In the *Processes* panel, make sure that the *Design* tab is selected and then choose *Behavioral Check Syntax*. If there is any error, a note will appear in the *Console* panel at the bottom of the screen indicating the line number of the program where (or near where) the error is located.

When you have corrected all of the errors, choose *Simulate Behavioral Model* which will open a new window with the values of all of the inputs and outputs. Press this button to see the whole simulation.

You can add time to the simulation using this button.



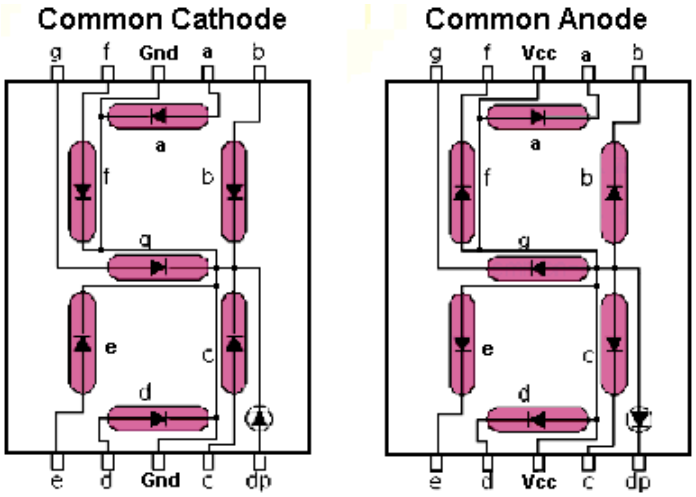
Verify that your program is working correctly by comparing the values to those found in the "previous work" truth tables at the beginning of exercise 2.

Note that all of the options change depending on what is selected in the various panels. If for example LOGIC is selected in the Sources panel, ISE will try to run the simulation using LOGIC instead of LOGIC_TEST and it will not work.

Exercise 3: 7-Segment Displays

Introduction In this exercise we will create a Verilog module to control a 7-segment display.

There are two types of displays. In order to illuminate a segment on a common-Cathode-display one must write a 1 to that segment. But if the display is a Common-Anode-type, then you must write a 0 to illuminate each segment.



Before you start: The displays on the Basys2 board have common anodes so we will write 0's to turn on each segment. So to display the number 2 all of the segments should be low (0) except for segments "c" and "f" which should be high (1). Fill in the truth table below.

	a	b	c	d	e	f	g
0	0	0	0	0	0	0	1
1							
2							
3							
4							
5							
6							
7							
8							
9							
A							
b							
C							
d							
E							
F							

Create an ISE project → call it Seven_Segmen t→ right-click on the name choose "Add copy of source…" and select hex7seg.v (it's shown below). You'll have to change the X's to represent the values in your truth table.

New keyword: **`always @(*)`** this means that the block will execute whenever any of the variables in the block changes (in the case below the block will run every time there's a change in one of the switches sw.)

wire and reg are the two most common variable types. In Verilog. Wires do not store values so they can only be used for combinational logic. reg stands for Register and can be used to model a flip flop.

[3:0] indicates that the variable sw is a 4-bit bus. sw[3] is the most significant bit and sw[0] is the least significant bit. a_to_g is a 7-bit bus (one for each segment on the display.)

Module is a keyword that indicates the start of a new program (called a module) in Verilog

Each command ends in a semi-colon in Verilog. I've broken this command into three lines of code to make it easier to read.

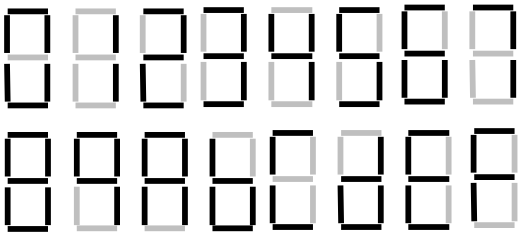
```
module hex7seg (
input wire [3:0] sw,
output reg [6:0] a_to_g );

always @(*)                  //abc_defg
a_to_g =       (sw==0)? 7'b000_0001 :
               (sw==1)? 7'bXXX_XXXX :
               (sw==2)? 7'bXXX_XXXX :
               (sw==3)? 7'bXXX_XXXX :
               (sw==4)? 7'bXXX_XXXX :
               (sw==5)? 7'bXXX_XXXX :
               (sw==6)? 7'bXXX_XXXX :
               (sw==7)? 7'bXXX_XXXX :
               (sw==8)? 7'bXXX_XXXX :
               (sw==9)? 7'bXXX_XXXX :
             (sw=='hA)? 7'bXXX_XXXX :
             (sw=='hB)? 7'bXXX_XXXX :
             (sw=='hC)? 7'bXXX_XXXX :
             (sw=='hD)? 7'bXXX_XXXX :
             (sw=='hE)? 7'bXXX_XXXX :
                        7'bXXX_XXXX ;

endmodule
```

Keywords appear in blue in the ISE smart editor.

Commented text starts with // and appears in green in the ISE smart editor.
You can comment large sections of code by highlighting it and right clicking

This is one way to make an if/than structure in Verilog. If the switches (sw) have a value of 0 then the 7-segments (a_to_g) will have a value of 000 0001 i.e., all "on" except for the center segment.

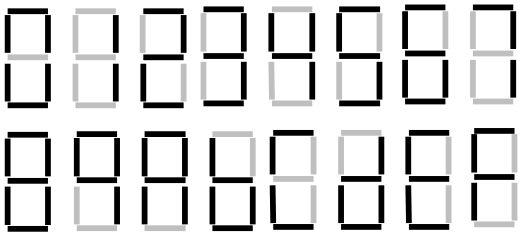
7'b000_0001 means we have a 7-bit number in binary whose value is 000 0001 (which corresponds to the top line in the truth table). The underscore in the number is ignored but makes the number easier to read.



- Right-click on the file name select “Add Copy of Source…” and choose “Universal.ucf”
- Uncomment sw[3:0] and a_to_g[6:0] by right clicking.
- Create the .bit file by clicking on “Generate Program File”
- Program your FPGA, and verify that it’s working correctly.

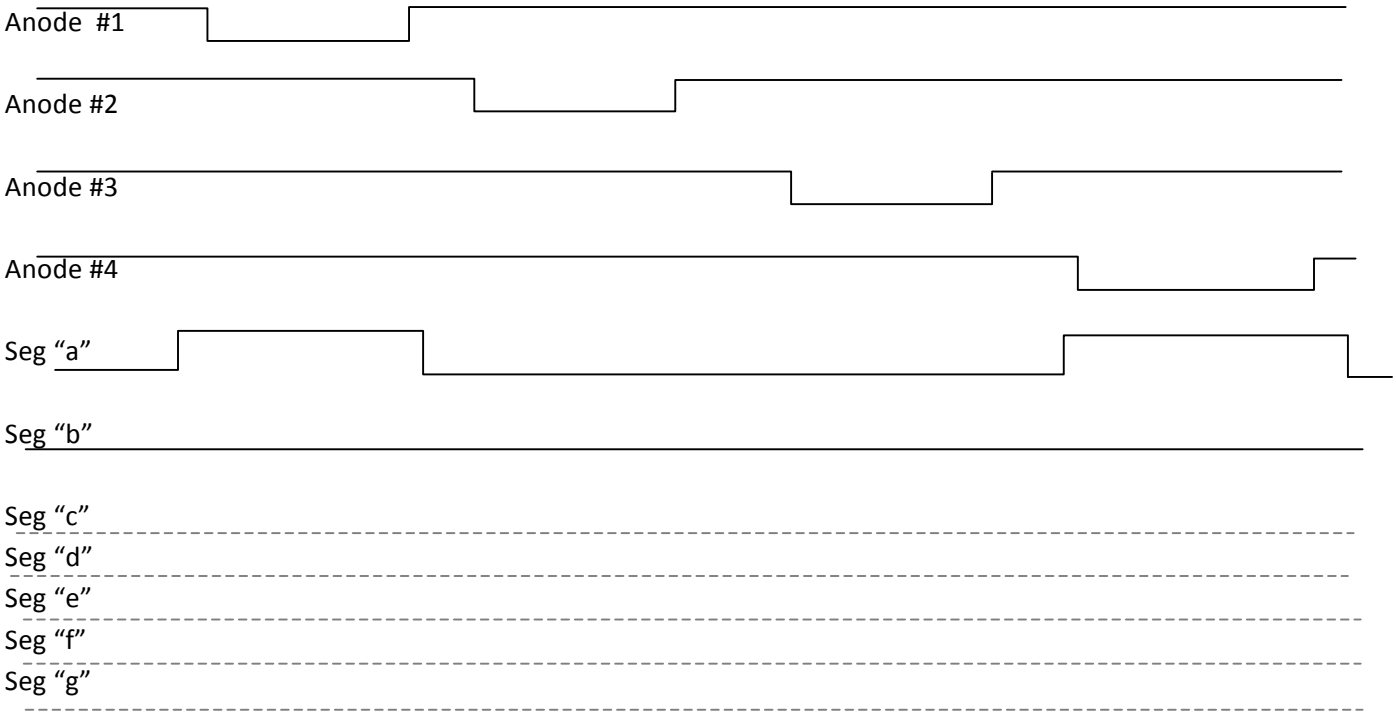
Exercise 3b: Writing four different values to the displays: Normally we will not want all four displays to show the same number. Now we are going to figure out how to write different values to each display.

In order to control the four displays separately we should have 32 outputs (7 segments + 1 decimal point) * 4 displays = 32. But in order to conserve inputs/outputs on the FPGA we will multiplex the controls so that the displays can share the same controls. The basic idea is that we will use one line to turn on the first display then we will write the desired number to it. Then we’ll wait a few microseconds, turn off the first display, turn on the second display, write its value, wait, etc.



For example, in order to write the number 1234 to the display, we’ll do the following. Segment “a” is off for display’s 1 and 4 and is “on” for numbers 2 and 3. This means that we’ll put segment “a” high when anode 1 and anode 4 are on, and we’ll put segment “a” low when anode 2 and 3 are active.

Draw the waveforms below so that the number 1234 will appear in the four 7-segment displays. Segments “a” and “b” are done for you. In reality, it doesn’t matter exactly when the transitions are as long as the segments have their correct value when the corresponding anode is active.



Create a new project called x7segb *(the b stands for the fact that this version will blank leading zeros).*

Right-click on the file name, select "Add copy of source" and choose x7segb_top.v

This is the TOP module. All it does is pass a 16-bit number (x) to the sub-module (x7segb) to be displayed.

```
module x7segb_top(                  //You can download this program from D2L.
input wire mclk ,                   //After opening a new Project in ISE right-click on D
input wire [3:0] btn ,              // the name of your Project and select "Add Copy of Source"
input wire [7:0] sw ,               // Also add the sub-module .UCF universal below.
output wire [6:0] a_to_g ,
output wire [3:0] an ,
output wire dp );

wire [15:0] x;      // the number that appears on the display is a concatenation of the switches,
                    // three of the buttons, and one constant with a value of A.

assign x = {sw, btn[2:0], 5'b01010};  //digit 0 = A

x7segb X2 (.x(x) , .mclk(mclk), .clr(btn[3]), .a_to_g(a_to_g), .an(an), .dp(dp) );  // this submodule is copied below

endmodule
```

Then right-click on the file name, select "Add copy of source" and choose x7segb.v which is shown on the next page.

Add another source (Univeral.ucf) and uncomment all of the inputs and outputs that appear in the top module.

Generate Programming File

Install onto your BASYS2 Boards

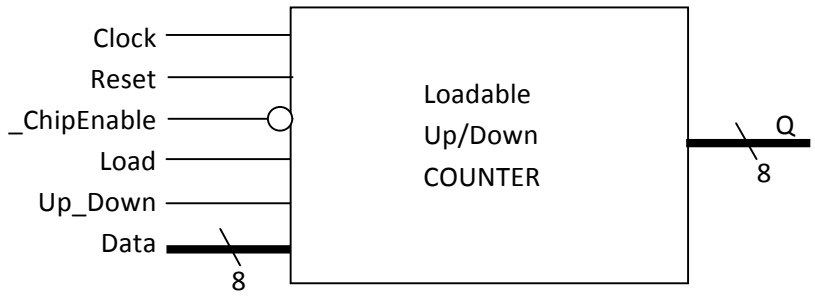
Verify that the correct numbers appear on the 7-segment displays.

What position do the switches and buttons have to be so that the display reads *CACA*?

How would you change how "x" is assigned in the top module to spell out the word *FACE*?

```
module x7segb(                          //this program can be downloaded from D2L
input wire [15:0] x ,
input wire mclk ,                       //This is the 50 MHz  clock
input wire clr ,
output reg [6:0] a_to_g ,
output reg [3:0] an ,
output wire dp  );

wire [1:0] s;
reg [3:0] digit ;
wire [3:0] aen ;
reg [19:0] clkdiv ;

assign dp = 1 ;                         //Turn off the decimal point
assign s = clkdiv[19:18] ;              //Count every 5.2 ms = 190.7 Hz  = 50,000,000 Hz / 2^18 = 50,000,000 Hz / 262,144
assign aen[3] = | x[15:12] ;            //Does the first digit have a value?  (If not, let's turn it off, called "Blanking".)
assign aen[2] = | x[15:8] ;             //Do the first two digits have a value?
assign aen[1] = | x[15:4] ;             //Do the first three digits have a value?
assign aen[0] = 1 ;                     //We will not turn off the 4th digit even if it's zero.  So we'll always leave it on.

// Quad 4-to-1 MUX:  mux44
always @(*)
        case(s)                         // "s" serves as a clock with two bits that is a lot slower (190.7 Hz) than mclock (50MHz)
                0: digit = x[3:0];      //  there are four possibilities for "s"   00,  01,  10,  y 11
                1: digit = x[7:4];      //  "x" is a 16 bit number.  Every 4 bits group of "x" represents on of the numbers displayed.
                2: digit = x[11:8];     // when "s" has a value of 10, we'll copy bits [11:8] to the variable digit.
                3: digit = x[15:12];
                default: digit = x[3:0];
        endcase

// 7-segment decoder: hex7seg
always @ (*)
        case(digit)
                0: a_to_g = 7'b0000001;   //when "digit" has a value of zero, we will turn on (write to zero) all of the segments except "g".
                1: a_to_g = 7'b1001111;
                2: a_to_g = 7'b0010010;
                3: a_to_g = 7'b0000110;
                4: a_to_g = 7'b1001100;
                5: a_to_g = 7'b0100100;
                6: a_to_g = 7'b0100000;
                7: a_to_g = 7'b0001111;
                8: a_to_g = 7'b0000000;
                9: a_to_g = 7'b0000100;
                'hA: a_to_g = 7'b0001000;
                'hB: a_to_g = 7'b1100000;
                'hC: a_to_g = 7'b0110001;
                'hD: a_to_g = 7'b1000010;
                'hE: a_to_g = 7'b0110000;
                'hF: a_to_g = 7'b0111000;
                default: a_to_g = 7'b0000001;
        endcase

always @(*)
        begin
        an = 4'b1111;                   //  We'll start by turning all of the display's off.  .
        if(aen[s] == 1)                 //  If the value of the "s" bit isn't blanked
        an[s] = 0;                      // then we'll turn it on
        end

//Clock divider
        always @(posedge mclk or posedge clr)
                begin                                                           //  this is the big counter
                        if(clr==1) clkdiv <= 0;                                 //  in this case we are using bits
                        else                    clkdiv <= clkdiv + 1;           //  [19:18] to make our slow clock
                end
endmodule
```

Exercise 4: Reg's (make a counter) **Introduction:** This circuit is a counter. It has 6 inputs and one output "Q". The input with the highest priority is "Reset."




```
module COUNTER(
   input Clock,
   input Reset,
   input _ChipEnable //active low
   input Load,
   input Up_Down,
   input [7:0] Data,
   output reg [7:0] Q
   );

always @(posedge Clock or
            posedge Reset)
if (Reset)    Q = 0;
 else
   begin
   if (~_ChipEnable)
      begin
      if (Load) Q = Data;
      else
         begin
         if (Up_Down) Q=Q + 1;
         else Q = Q - 1;
         end
      end
   end
endmodule
```

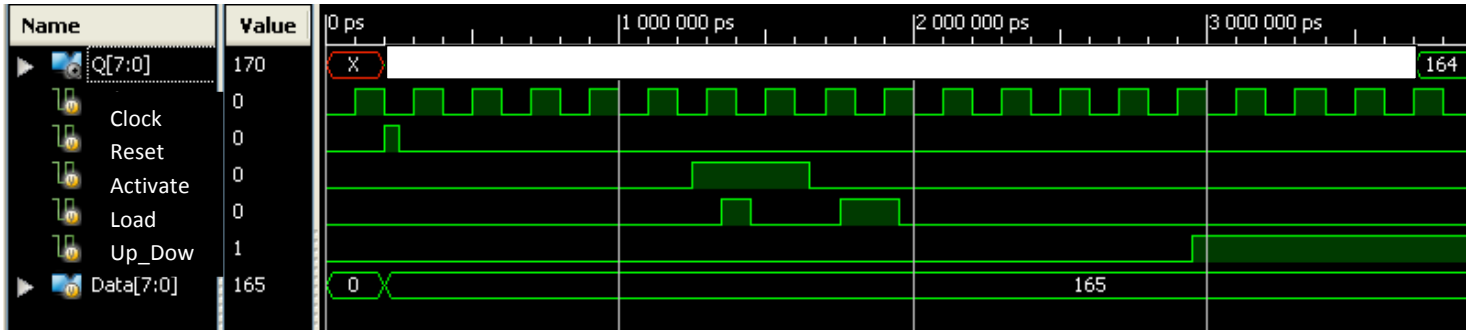
Up until this point we have been dealing with combinational logic where the output only depends on the current value of the inputs. Now we are going to **sequential logic** where the outputs depend not only on the current value of the inputs, but also on their past values. In order to remember what the past value was we'll use a new variable type called a reg (short for register and pronounced that way).

The default variable type is *wire*.

Whenever, "Reset" is 1,"Q" will be set to zero.

If "Reset" is zero and the circuit is enabled it is possible to load a number into the counter. If "Load" is high then the value of "Data" will be transferred to "Q".

Before you start: Sketch-in below the value of the output for the inputs shown.



A note on Regs: If you want to assign a value to a variable in an always or initial block you have to use a reg. On the other hand, inputs have to be wires. Also, if something receives its value using the keyword *assign* (like we used in exercise 3) then it has to be a wire. My program for the counter is on the right. Note that Verilog uses the key words *begin* and *end* to form hierarchies.

Create a Test Bench

Right-Click on the name of your file and select New Source. Select Verilog Test Fixture. You can add the code on the right to the test bench. Choose Simulate Behavioral Model. Look carefully at the result. Was your prediction correct?

Program your FPGA: As always, you must create a .ucf to define the pins. I used the switches for Data, the LED's for Q,C8 for Clock, and the buttons for Reset, _ChipEnable, Load, and Up_Down. I called my file COUNTER.ucf

```
            // Add stimulus here
            #100 Reset = 1;
                  Data = 8'b1010_0101;
            #50 Reset= 0;
            #1000 _ChipEnable = 1;
            #100 Load = 1;
            #100 Load = 0;
            #200 _ChipEnable= 0;
            #100 Load = 1;
            #200 Load = 0;
            #1000 Up_Down = 1;
      end

      always #100 Clock= !Clock;

endmodule
```

Exercise 4b: Blocking vs. non-blocking. Verilog uses two signs for equals. They are called blocking (=) and non-blocking (<=). The difference is that all non-blocking assignments in a block occur at the same time (usually with the clock), while blocking assignments happen in sequential order.

In the example on the right, the values of flop1 and flop2 will interchange. But if we had used the *blocking* "=" then the program would execute in order and both flops would end with the value of flop2.

It's good practice to use *non-blocking* assignments (<=) with triggering on a positive or negative edge (*posedge and negedge*)

Create a .ucf file and install the program onto your board. Does it work?

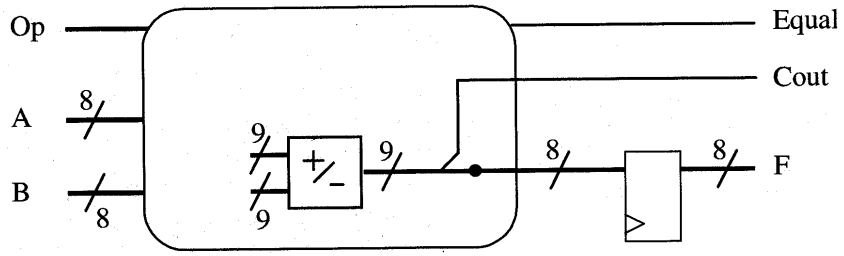
```
module nonb(
input wire clock,
input wire reset,
input wire enable,
output reg flop1,
output reg flop2
    );

always @ (posedge reset or posedge clock)
       if (reset)
               begin
                       flop1 <= 0;
                       flop2 <= 1;
               end
       else if (enable)
               begin
                       flop1 <= flop2;
                       flop2 <= flop1;
               end

endmodule
```

Save your Project as BLOCKING and change all of the **<=** to = Did the behavior change?

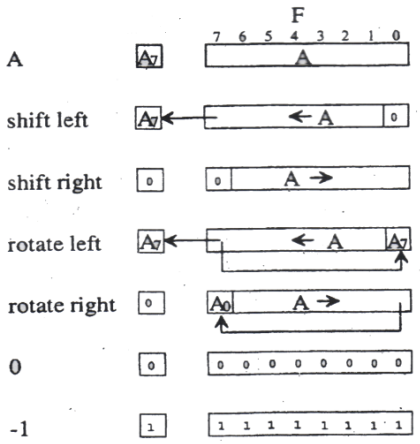
Exercise #5: Design an arithmetic logic unit (ALU)

Write a module for an 8-bit ALU that completes the following arithmetic and logical operations. The output of the ALU (F) should be registered (made from clock-triggered flip flops). The outputs Cout and Equal should not be registered (they should be purely combinational and change asynchronously with A and B).

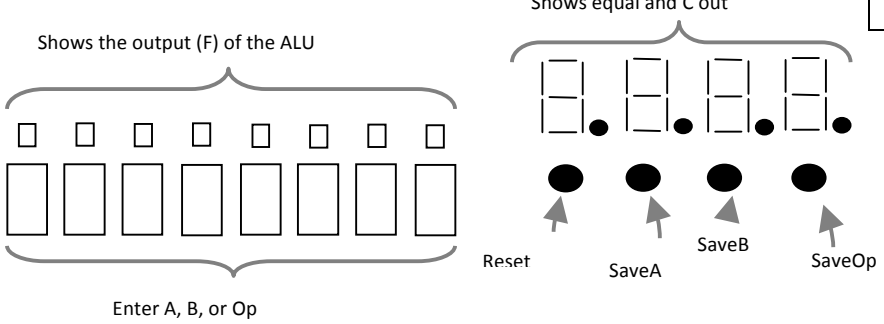
The use of case is used to select the different operations of the ALU. It is good practice to create a default-case so that the output will be predictable even when none of the case values matches. Note that parameters have been used to make the code more readable.




```
module ALU(
   input Clock,
   input [7:0]A,
   input [7:0]B,
   input [3:0]Op,
   output reg [7:0]F,
   output Cout,
   output Equal );
          reg [8:0]Tmp;
          parameter Three = 3;          //When the word Three is used in this program
                                        //it will represent the number 3.
          parameter Twelve = 12;        //When the word Twelve is used
                                        //it will represent the number 12.
          assign Cout = Tmp[8];         //Unlike regs, wires like Cout and Equal can
          assign Equal = (A==B);        //be continuously assigned outside of always blocks

          always @(A or B or Op)
                case (Op)
                4'b0000: Tmp <= {1'b0,A} + {1'b0,B};      //although not strickly necessary,
                                                          // I wanted to make
                4'b0001: Tmp <= {1'b0,A} - {1'b0,B};      //explicit the fact that Tmp has
                                                          //one more bit than A or B
                2:       Tmp <= {1'b0,B} - {1'b0,A};      //decimal is the default format for numbers
                Three:   Tmp <= {!A[7], ~A};              //the word Three was declared as a parameter
                                                          //that represents he number 3
                4'b0100: Tmp <= {A[7],A};                 //repeating A[7] maintains the sign
                4'b0101: Tmp <= {B[7],B};
                4'b0110: Tmp <= -A;
                4'b0111: Tmp <= -B;
                4'b1000: Tmp <= {A[7:0], 1'b0};
                4'b1001: Tmp <= {2'b00, A[7:1]};
                4'b1010: Tmp <= {A, A[7]};
                4'b1011: Tmp <= {1'b0, A[0], A[7:1]};
                Twelve:  Tmp <= {!A[7],~A} + 1'b1;        //finding the 2's complement
                13:    Tmp <= {!A[7],!A} + 1'b1;          //notice the difference between logical negation (!)
                                                          //and bitwise negation (~)
                4'b1110: Tmp <= 9'b000000000;
                default: Tmp <= 9'b111111111;             //it's always a good idea to use default as the last case
                endcase
          endmodule
```



Operation	Code
A+B	0000
A-B	0001
B-A	0010
Your choice	0011
A	0100
B	0101
-A	0110
-B	0111
Shift left A	1000
Shift right A	1001
Rotate left A	1010
Rotate right A	1011
Your choice	1100
Your choice	1101
F=0	1110
F= -1	1111



```
module ALU_Top(
        input mclk,
        input [7:0]sw,
        input [3:0]btn,
        output [7:0]ld,
        output [6:0]a_to_g
    );

        wire [7:0]F;
        wire Cout, Equal, Clock, SaveA, SaveB, SaveOp;
        reg [7:0]A, B;
        reg [3:0] Op;
        assign Clock = mclk;
        assign Reset = btn[3];
        assign SaveA = btn[2];
        assign SaveB = btn[1];
        assign SaveOp = btn[0];
        assign ld = F[7:0];
        assign a_to_g[0] = !Cout ;
        assign a_to_g[1] = !Equal;
        assign a_to_g[6:2] = 5'b1_1111;   // to turn them off.

        ALU U2 (.Clock(mclk), .A(A), .B(B), .Op(Op), .
                F(F), .Cout(Cout), .Equal(Equal)) ;

                always @ (posedge Clock or posedge Reset)
        if (Reset)   begin
                                A    <= 8'b0000_0000;
                                B    <= 8'b0000_0000;
                                Op   <= 4'b0000;
                        end
                else if (SaveA)  A  <= sw;
                else if (SaveB)  B  <= sw;
                else if (SaveOp) Op <= sw[3:0];

endmodule
```

Exercise 6: Understanding For Loops

For Loops can be created in hardware (synthesized) only if they execute a fixed number of times. In the case on the left, the loop executes six times. Since "a" has a value of 3 in my test bench, we would expect that "d" would have a value of $(3)^6 = 729 = 2D9_{hex}$. But since "d" only has 8-bits, we only have D9. Likewise $(4)^4 = 256 = 100_{hex}$ which is rounded to 00 and so everything after that will be zero too. You might think that executing the loop six times would require six clock cycles, but since it is performed by six separate circuits they all happen in one clock cycle.

```
module Loop_4(a, b, c, reset, clock, d);
input [3:0] a, b;
input [1:0] c;
input reset, clock;
output reg [7:0] d;
integer i,j;
  always @ (posedge reset or posedge clock)
     if (reset)  d = 0;
     else case (c)
        00: d = a + b;
        01: d = a * b;
         2: begin
              d = 1;
              for (i = 0; i <= 5; i = i+1) d = d * a;
            end
  default: begin
              d = 1 ;
              for (j = 0; j <= 5; j = j+1) d = d * b;
            end
     endcase
  endmodule
```

Change the end of your Test bench to match this

```
      // Wait 100 ns for global reset to finish
      #100;

      // Add stimulus here
      #100 reset = 1;
      #100 reset = 0;
      #100 a = 3;
      #100 c = 1;
      #100 b = 4;
      #100 c = 2;
      #100 c = 3;

   end
   always #50 clock = ~clock;

endmodule
```

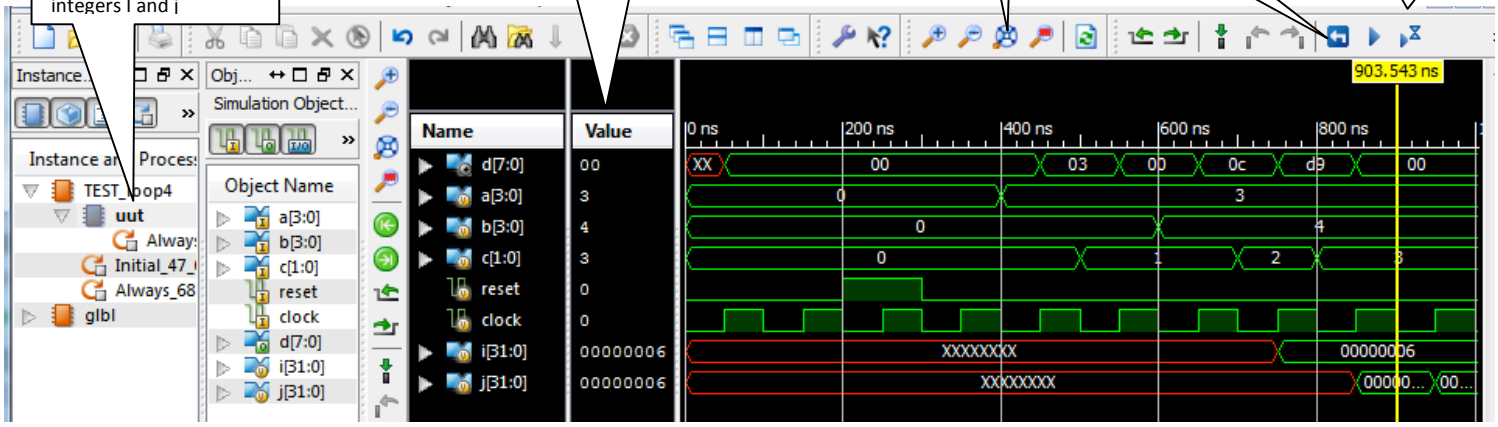
Click on UUT (unit under test) to show integers I and j

Right-click on selected variables to change the Radix to Hexadecimal. (or make the change your default under Edit -> Preferences -> ISim Simulator

Show the whole simulation

Go back to the start of the simulation

Run simulation for X amount of time (X = value on the right). This is often safer the "run" button which may get caught in an endless loop



Verilog Instruction QuickSheet

Xilinx-ISE is available free (search for ISE WebPack at www.xilinx.com)
Adept is also available free at www.digilent.com

Names

May contain letters (a-z, A-Z) digits (0-9), underscores (_) and dollar signs ($)
Must begin with a letter or an underscore.
Verilog is case sensitive so lower case and uppercase letters are seen as different
Can have up to 1024 characters.

MISC.

Verilog doesn't distinguish between different white spaces (spaces, tabs, and carriage returns are all equal).
Every command in Verilog ends with a semicolon (;) But often you can make code more readable by making a single command span several lines.
The keywords *always, if, else, else if* set up a single command. To introduce several commands, they must be bookended with the keywords *begin* and *end*.
It's a good idea to represent numbers like this 8'b0000_1111 (which means 8-bits of binary with a value of 15) because it forces you to think about how exactly the number will be represented in hardware (with 8-bits). Other formats include *h* (hexadecimal), *d (*decimal), o *o* (octal). The default is decimal.
The symbol // introduces single-line comments. (except in the .ucf files where comments start with the number sign (#)
For comments that require more than one line use /* to start and */ to end the comment.

Algebraic operations.

+ - * /	Arithmetic	4'b1111 + 4'b0001 = 4'b0000
%	Modulus (what remains after division).	
**	Raising to a power.	

Bitwise and reduction operators

~	bitwise negation (complement)	~4'b1010 =4'b 0101 (if there is only one bit ~ and ! are the same)
!	logical negation	!4'b1010 = 1'b0 (since every number except for zero is logically true) !4'b000 = 1'b1
|	or	4'b0101 | 4'b1100 = 4'b1101 | 4'b0101 = 1b1
&	and	4'b0101 & 4'b1100 = 4'b0100 & 4'b0101 = 1'b0 If there is only one bit then there is no difference between & and &&
^	X OR	4'b0101 ^ 4'b1100 = 4'b1001 ^4'b0101 = 1'b0 (Parity)
~&	NAND	
~|	NOR	
~^	XNOR (same as a ^~)	
<<	Shift left	
<<<	Signed shift left (maintains sign)	
>>	Shift right	
>>>	Signed shift right (maintains sign)	

Binary Operators

Every number except for zero is considered logically true

&&	logical and	4'b0101 && 4'1100 = 1'b1
||	logical or	
>	greater than	
>=	greater than or equal to	
<	less than	
<=	less than or equal to	
==	equal to	
!=	logical inequality	
===	case equality	== is different than === which compares x (unknown) and z (high impedance) states
!==	case inequality	x and z states are used only in simulations.
A ? B : C	conditional	if A then B else C
{A, B, C}	concatenation	A = 2'b10, B = 2'b00, C= 2'b11 {A,B,C} = 6'b10_0011
{N{A}}	replication	{N{A}} repeats the number A, N times