Verilog and Altera Crash Course

Verilog Introduction:

Verilog is a hardware description language that couples standard programming language semantics with hardware constructs to facilitate the simulation and synthesis of circuits. However, while Verilog at times may look like a C-style language, one must remember that your code will ultimately be used to generate hardware. As a result, a variety of constructs that are common in programming languages like C or Java (eg. dynamically sized arrays, creation and destruction of objects, recursion, non-constant loop iterations) do not easily translate into hardware representations. Thus, we will restrict you to a powerful subset of Verilog that will hopefully prevent you from wasting time agonizing over the intricacies of Hardware Description Languages.

Modules:

Typically, your logic should be encompassed within a variety of modules. An example of an 8-bit adder module is shown below:

```
module Adder#(
        parameter width=8
)
(
        //input declarations
        input [width-1:0] A_in,
        input [width-1:0] B_in,

        //output declarations
        output [width-1:0] C_out
);

//Internal Logic
assign C_out = A_in + B_in;

endmodule
```

As you can see, modules consist simply input and output declarations and internal logic. One can also declare parameters, which in the case above function as constants, although further information on parameters will be discussed later. One can also define the bit length of any signal, as shown by the *input [width-1:0] A_in* statement, which consists of 8 bits indexed from 0 to 7 (since *width* is defined to be 8). If one does not specify a width (for example *input A_in*), Verilog semantics dictate the number of bits that comprise *A_in* is 1.

Modules can be combinational (the internal logic is not reliant on a clock) or sequential (the internal logic reliant on a clock), and the above adder is an example of a combinational module. With combinational modules, one can express the internal logic with assign statements, to an extent. Unfortunately, control flow (if-else statements) cannot be expressed simply through assign statements. For example, a selector requires the selection of two inputs based on the value a third input, and the

resulting module is shown below:

```
module Selector#(
        parameter width=8
)
(
        //input declarations
        input [width-1:0] A_in,
        input [width-1:0] B_in,
        input select,

        //output declarations
        output reg [width-1:0] C_out
);

//Internal Logic
always @(*)
begin
        if (select)
        begin
                C_out = B_in;
        end
        else
        begin
                C_out = A_in;
        end
end

endmodule
```

Here, the internal logic is enclosed within an *always @(*)* block, which simply means that the statements within the *begin* and *end* tokens constitute a combinational block.

Not all circuits, however, are combinational. Sequential circuits (those that depend on a clock input) can be constructed as shown below:

```
module Register
(
        input CLK,
        input D_in,

        output reg O_out
);

always @(posedge CLK)
begin
        O_out <= D_in;
end

endmodule
```

As you have probably noticed, sequential modules require an *always @(posedge CLK)* declaration, which means that the statements within the *begin* and *end* tokens will occur when the *CLK* signal transitions from 0 to 1.

Blocking vs. Nonblocking Assignments:

You may have also noticed one other difference, namely the "nonblocking assignment" *O_out <= D_in*. This looks different from what you are probably used to, which would probably look something like *O_out = D_in*. In Verilog, these two types of assignments have very different semantics. Take for example the following:

```
always @(posedge CLK)
begin
        a = b;
        b = a;
end
```

The above code is an example of "blocking assignments." Let's assume that, before we execute the *always* block, *a = 1*, and *b = 0*. If the above code was executed in a language like C, the first assignment would happen, resulting in *a* being set to *0*, and the subsequent assignment of *a* to *b* will result in *b* being set back to its original value of *0*. This is because the blocking semantics of the "=" operator results in lines being executed in order.

However, if we turn our attention to the following line of code, we will see some different behavior:

```
always @(posedge CLK)
begin
        a <= b;
        b <= a;
end
```

Here, the "<=" operator results in nonblocking semantics, which means that all assignments within the block occur in parallel. This means that, if *a = 1*, and *b = 0* before the block executes, after the first clock edge, the assignment of *b* to *a* will occur at the same time. This will result in *a* obtaining the value of *0* and *b* obtaining the value of *1*.

Intermediate Signals:

Like all C-style languages, Verilog provides the ability to declare intermediate variables of a variety of types within a module. We recommend you restrict yourself to two types, *reg* and *wire*. The major difference between these two types is that signals of type *wire* cannot be assigned within a *begin-end* block, whereas signals of type reg can only be assigned within a *begin-end* block. For example, the following code is valid:

```
wire a;
wire b;
reg c;
assign a = b;

always @(*)
begin
        c = a;
end
```

On the other hand, the following code is invalid, as it contains a reg type being assigned outside of a begin-end block and a wire type being assigned within one.

```
wire a;
wire b;
reg c;
assign c = a;

always @(*)
begin
        b = a;
end
```

Instantiating Modules:

At some point, you may want to use a module that you write. For example, let's say we wanted to use the adder and selector that we defined earlier in a new module. An example of how to do this is shown on the next page.

```
module AdderShift#(
        parameter width=11
)
(
        input [width-1:0] a_in,
        input [width-1:0] b_in,
        input shift_in,
        output [width-1:0] o_out
);

wire [width-1:0] adder_out;
wire [width-1:0] adder_out_shift = adder_out << 2;

Adder #(
        .width(width)
) adder (
        .A_in(a_in),
        .B_in(b_in),
        .C_out(adder_out)
);

Selector#(
        .width(width)
) selector (
        .A_in(adder_out),
        .B_in(adder_out_shift),
        .select(shift_in),
        .C_out(o_out)
);

endmodule
```

As we can see, module declarations require both a module type and name. One can also set the parameters of the module externally. For example, the parameter *width* for both the Adder and Selector modules is set to be *11*, even though when we declared it that parameter was set to *8*. Verilog semantics dictate that the outermost module's parameter definition will propagate into the child module, which means that the instantiation of our *Adder* and *Selector* will have its width parameter set to *11* since the outermost module, *AdderShift*, set it to be *11*.

To connect the inputs, outputs, and parameters, it is more reliable to utilize the syntax used above. For example, when connecting the *adder_out* wire to the *A_in* pin of our Selector, we use the following syntax: *.A_in(adder_out)*. This will unambiguously specify which inputs and wires are connected, and it can automatically adjust the bit-width if the sizes do not match exactly.

Altera Tools Introduction:

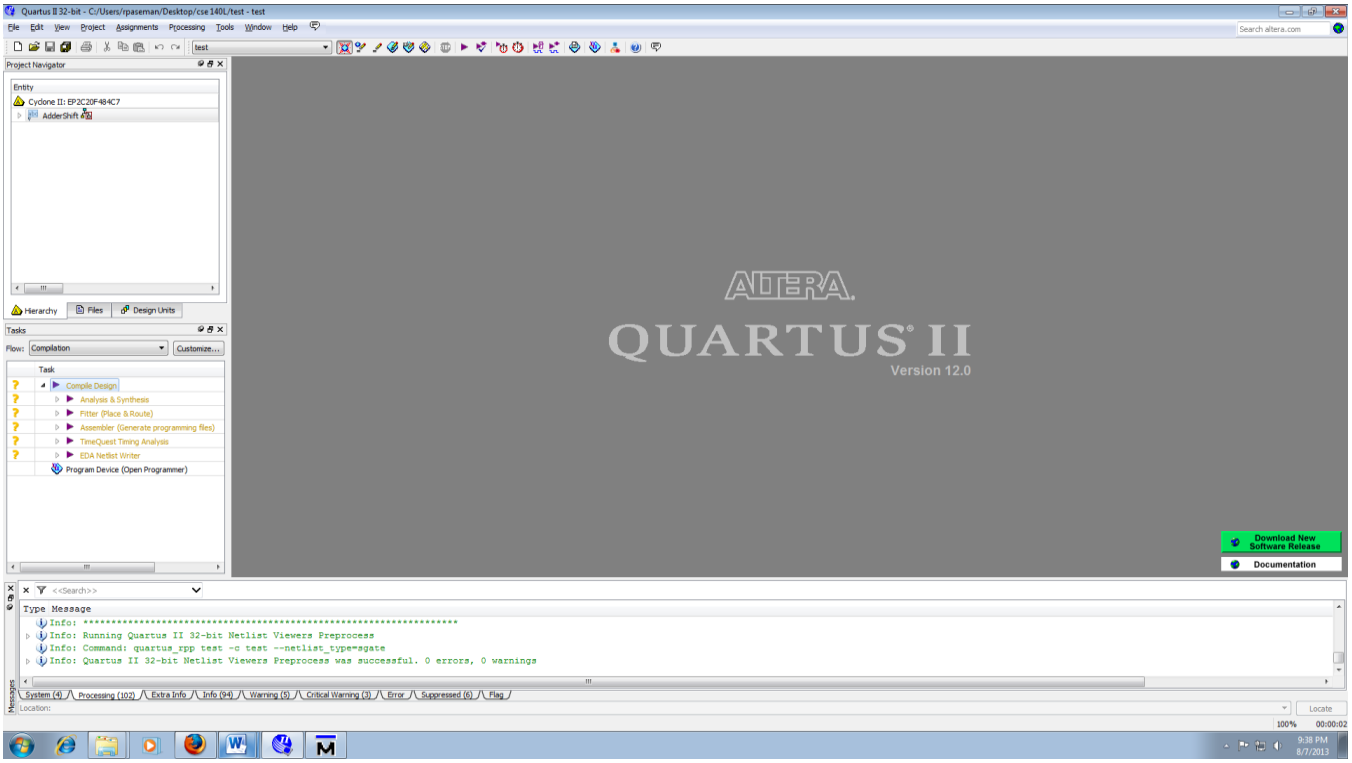
Now that you have gotten acquainted with Verilog, it is time to learn how you can verify that your Verilog code is well-formed. In this class, you will use the Altera tools, which should be in the lab computers in the basement. The two tools you will use will be Quartus and Modelsim. While the tools are quite powerful, they have a rather steep learning curve, so we hope that the following tutorial will alleviate this somewhat.

Quartus Introduction:

If you are in the lab basement computers, you can find Quartus by going to Start → All Programs → Altera 12.0 Build 178 → Quartus II 12.0sp2 → Quartus II 12.0sp2 (32-bit). When the Quartus window comes up, perform the following steps:

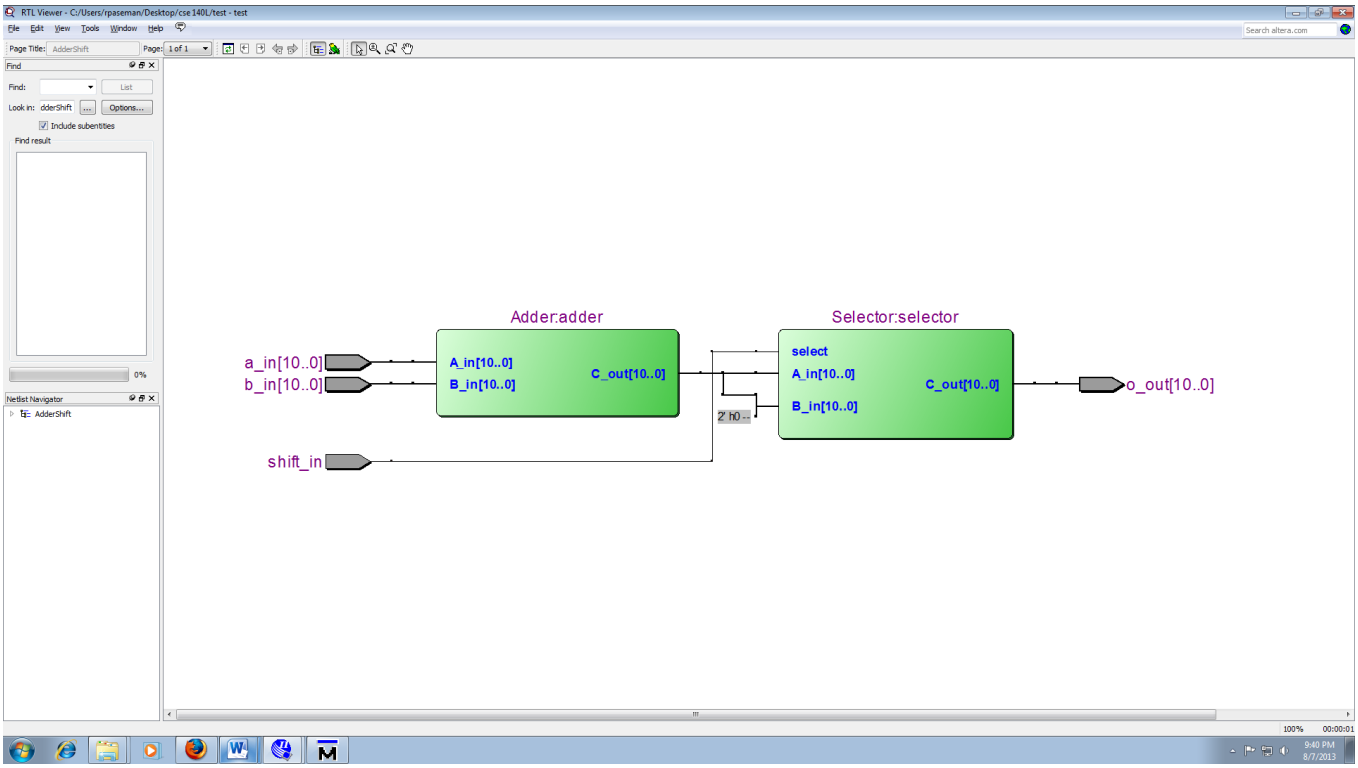
1. Select the New Project Wizard.
2. Click Next
3. Specify the project directory (probably a directory on your Desktop) and project name. Click Next.
4. If you have Verilog files, you can add them, but if not, just click Next.
5. In this class, we will target the Cyclone II EP2C20F484C7, so enter that into the window, select that FPGA, and click Next.
6. Make sure that the Simulation Tool Name is “ModelSim-Altera” and the Format is “Verilog.” Click Next.
7. Click Finish.

You should now have a window like the following:



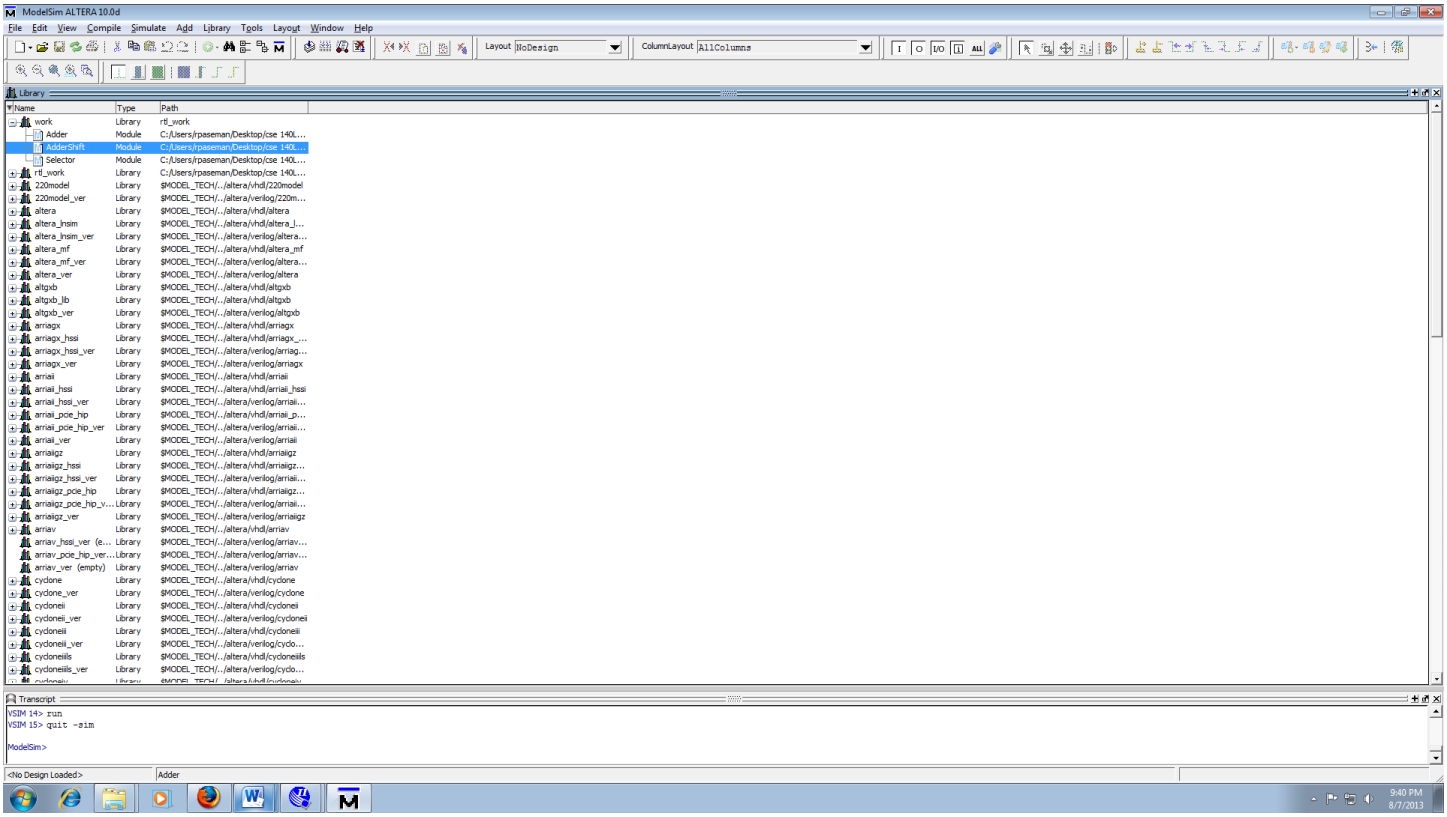
To create a Verilog file, go to File → New, and then select Verilog HDL File. You can then write your Verilog through Quartus' IDE. For example, you can copy the *Adder* code that is shown on Page 1. Save your file in the same directory that you specified in step 3 from above, and you will now see your Verilog file listed under the Files tab on the left. Right click the file and click "Set as Top-Level Entity" so that you can synthesize it. To synthesize your file, go to Processing → Start Compilation. Check the console at the bottom for any errors to fix if you have them.

If you want to include other modules in your project, you can create or import them. For example, if you create two new files and copy in the *Selector* and *AdderShift* modules to each of them, and if you set the top level module to *AdderShift*, then you will be able to compile all three modules into a single circuit. You can view a schematic of your code by selecting Tools → Netlist Viewers → RTL Viewer. For example, the RTL viewer for my *AdderShift* module looks as follows:

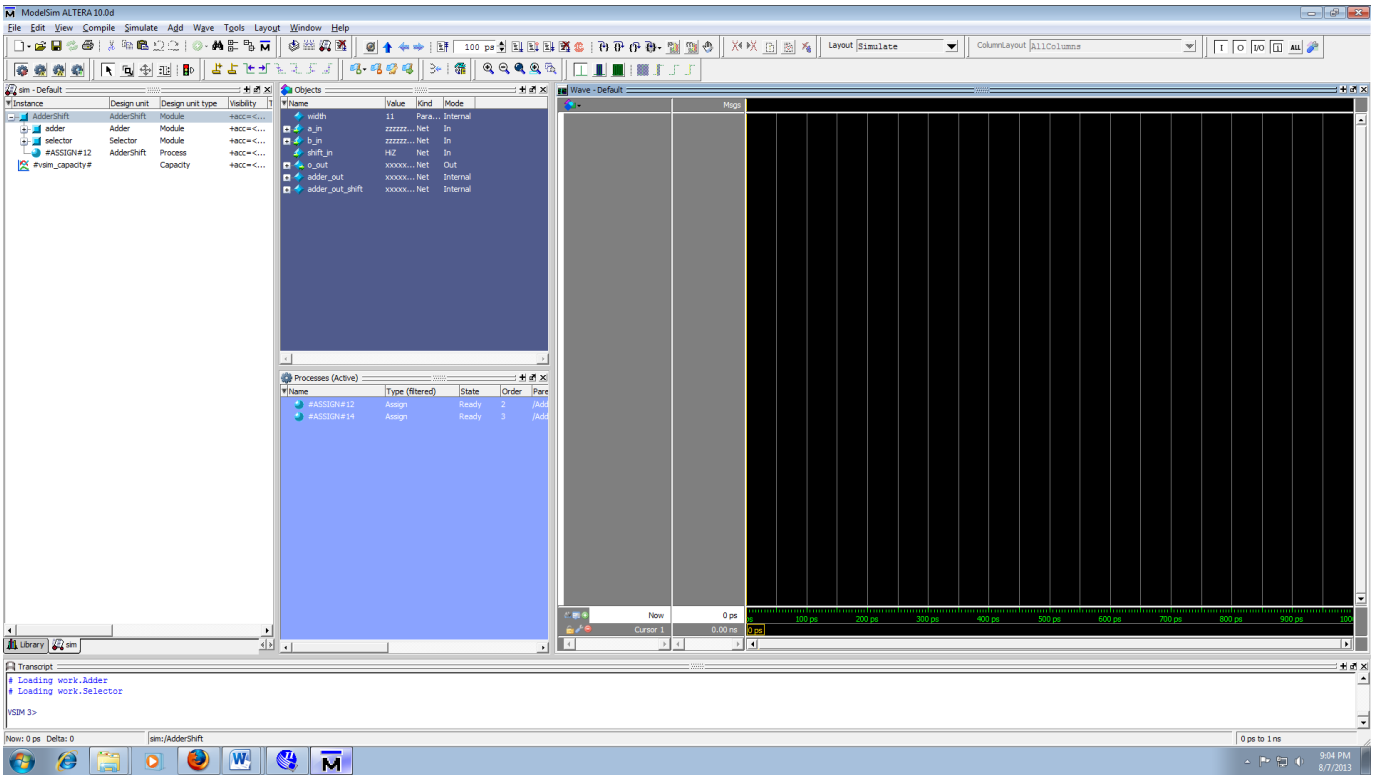


Modelsim Introduction:

There are many ways to start Modelsim, but the easiest way is through Quartus. After you synthesize your circuit, go to Tools → Run Simulation → RTL Simulation. When you do this, Quartus will supply Modelsim with the appropriate commands to create directories and set paths, mitigating the need for you to handle such tasks. If you do so for the aforementioned AdderShift module, you should see something like the following on your screen:

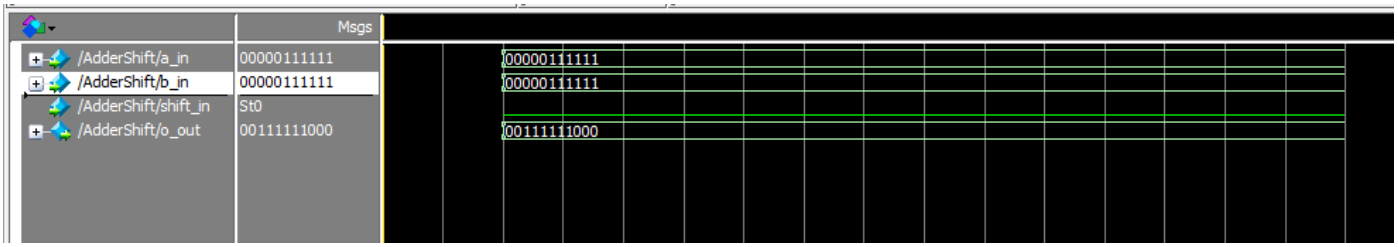


In the Modelsim window, expand the “work” library, and you should see your Verilog files. Right click the top-most module and click “Simulate” in order to simulate your Verilog. Click View → Wave to see the waveform window. Your resulting window should look something like this:



If you do not see this, check your Transcript window at the bottom to verify that all of your code was able to compile without errors.

To actually view the signals on the waveform, you need to drag them from the Objects window onto the Wave window. You can then set values for the inputs by right clicking them, clicking “Force,” and setting the value. To simulate, go to Simulate → Run → Run 100. You should see green lines and white numbers appear on the waveform, similar to the following:

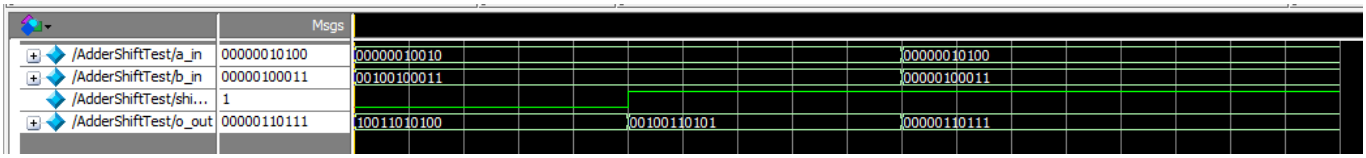


Testbenches:

Forcing signals every time you simulate can be quite cumbersome, especially if your circuit has a large number of inputs. You can automate the testing of your circuit by writing testbenches, which are simply wrapper modules that instantiate and supply inputs to your circuit. Here is an example of a testbench for the *AdderShift* module previously provided:

```
`timescale 1ns/1ps

module AdderShiftTest();

parameter width = 11;
reg [width-1:0] a_in;
reg [width-1:0] b_in;
reg shift_in;
wire [width-1:0] o_out;


AdderShift#(
        .width(width)
) addershift
(
        .a_in( a_in),
        .b_in(b_in),
        .shift_in(shift_in),
        .o_out(o_out)
);

initial begin
        a_in <= 11'h012;
        b_in <= 11'h123;
        shift_in <= 0;
        #1
        shift_in <= 1;
        #1
        a_in <= 11'h014;
        b_in <= 11'h023;

end

endmodule
```

You can add this module in Modelsim by clicking Compile → Compile… and navigating to the Verilog file containing the above code. You can them right click it and simulate it. After dragging the relevant signals to the Wave window, you should be able to simulate it and obtain the following in your waveform:



Further reading:

This handout is in no way comprehensive regarding the intricacies of Quartus, Modelsim, or Verilog, but it should provide a solid starting point for writing the Verilog that you will need to generate in this course. If you want a more comprehensive tutorial, you can check out the following links:

ftp://ftp.altera.com/up/pub/Altera_Material/10.1/Tutorials/Verilog/Quartus_II_Introduction.pdf

http://cseweb.ucsd.edu/classes/wi13/cse140L-a/modelsim_tut.pdf

http://cseweb.ucsd.edu/classes/wi13/cse140L-a/02-VerilogFundametals.pdf