UNIVERSITY OF BRITISH COLUMBIA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EECE 479: Introduction to VLSI Systems
Winter 2015

Final Course Project

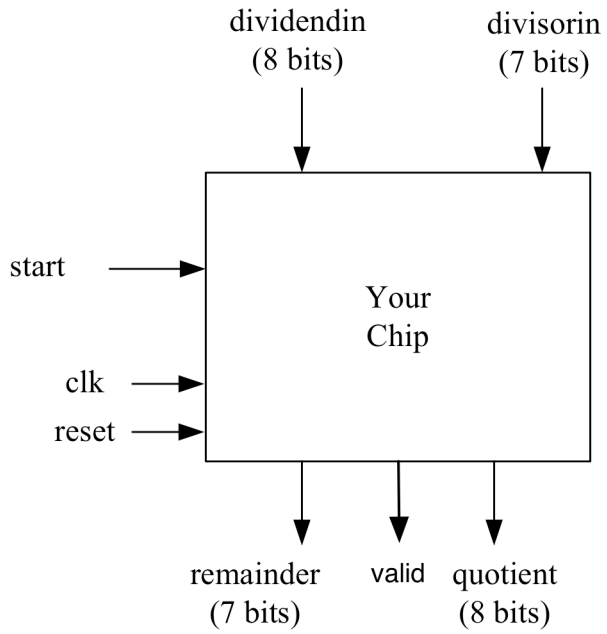
Due: April 10, 2013: 11:59pm
Work on this project in groups of two or three[1]

In the project for this course, you will design, lay out, and simulate an 8-bit divider. Dividers are critical in high-speed compression and encryption systems, and an efficient divider implementation is very important. This project will be a lot of work, so it is highly recommended that you get working on it early. A suggested schedule of tasks will be given; we won't be checking your progress, so it is up to you to make sure you meet the milestones. If you don't, you'll have a hard time completing the assignment.

Remember that you need to hand in the project to pass the course.

Black Box View

The following shows a black-box view of the circuit you will implement:

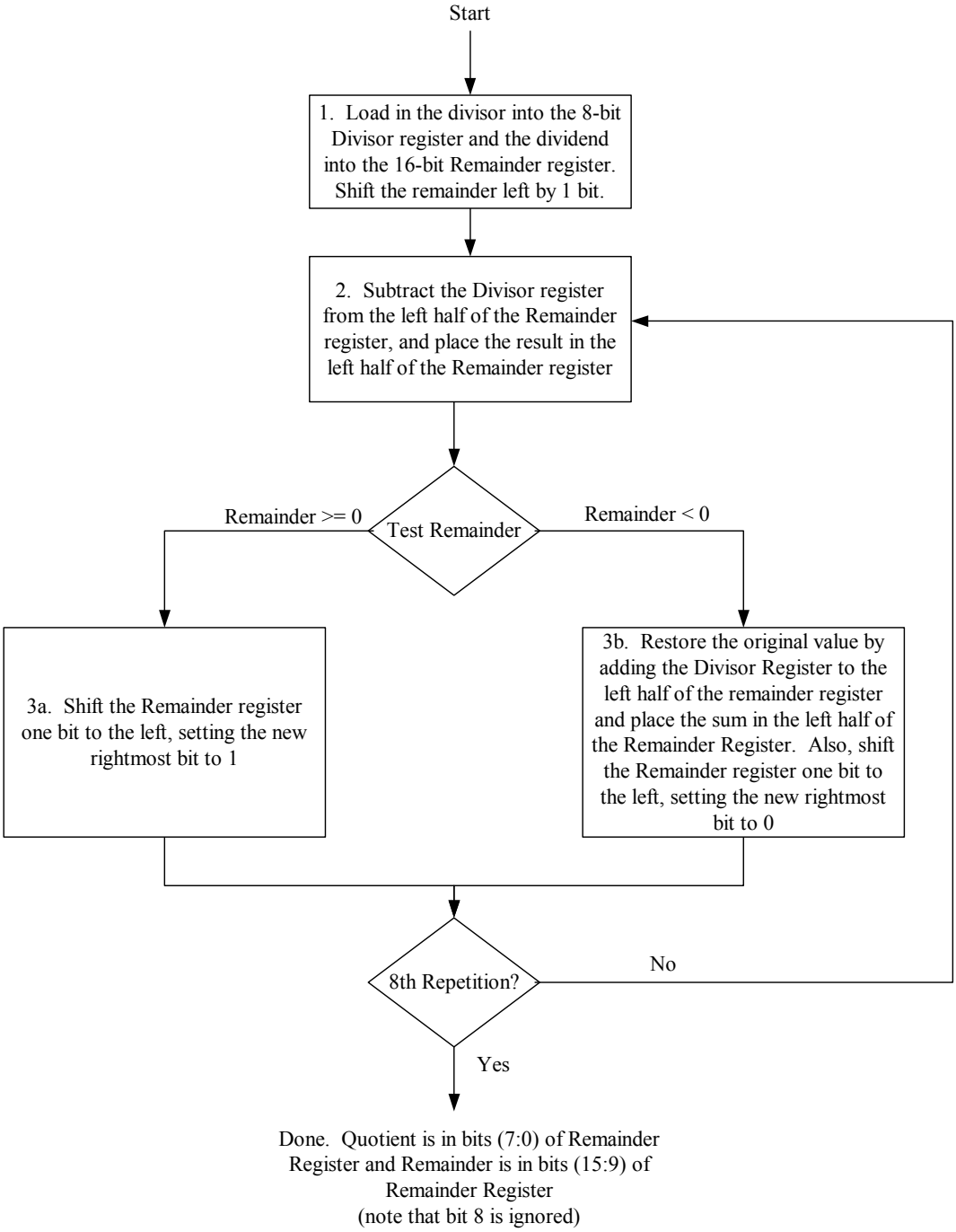


The operation of this circuit is as follows: the user drives the start cycle high for one cycle. During the next cycle, the user lowers start, and drives the dividend and divisor (in the equation a/b, a is the dividend and b is the divisor) on the appropriate inputs. After 17 further cycles, the remainder and quotient are available at the output of the chip. This condition is marked with a pulse of the valid signal.

[1] If anyone wants to work alone, they need to implement the entire circuit as described here (I don't recommend this)

Algorithm

The algorithm that you will employ is outlined in the following flowchart (the implementation details will be discussed soon; the flowchart is just to give you an idea of the algorithm).



An example of how this algorithm works is shown in the following table (a 4 bit divisor and an 8-bit dividend is shown for clarity, but it should be obvious how this scales up to an 8-bit divisor and 16-bit dividend). In this example, the number 0000 0111 (binary) is divided by 0010 (binary). The resulting quotient is 0011 and the resulting remainder is 001 (in the three MSB's of the Remainder register). Stare at this example and at the accompanying flowchart until you understand how it works.

Iteration	Step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem – Div	0010	1110 1110
	3b: Rem < 0, so add Div, and shift left, bit0=0	0010	0001 1100
2	2: Rem = Rem – Div	0010	1111 1100
	3b: Rem < 0, so add Div, and shift left, bit0=0	0010	0011 1000
3	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem >=0, so shift left, bit0=1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem >=0, so shift left, bit0=1	0010	0010 0011

For more information on algorithms to perform division, see "Computer Organization and Design" by David A. Patterson, John L. Hennessy (that is where this example came from).

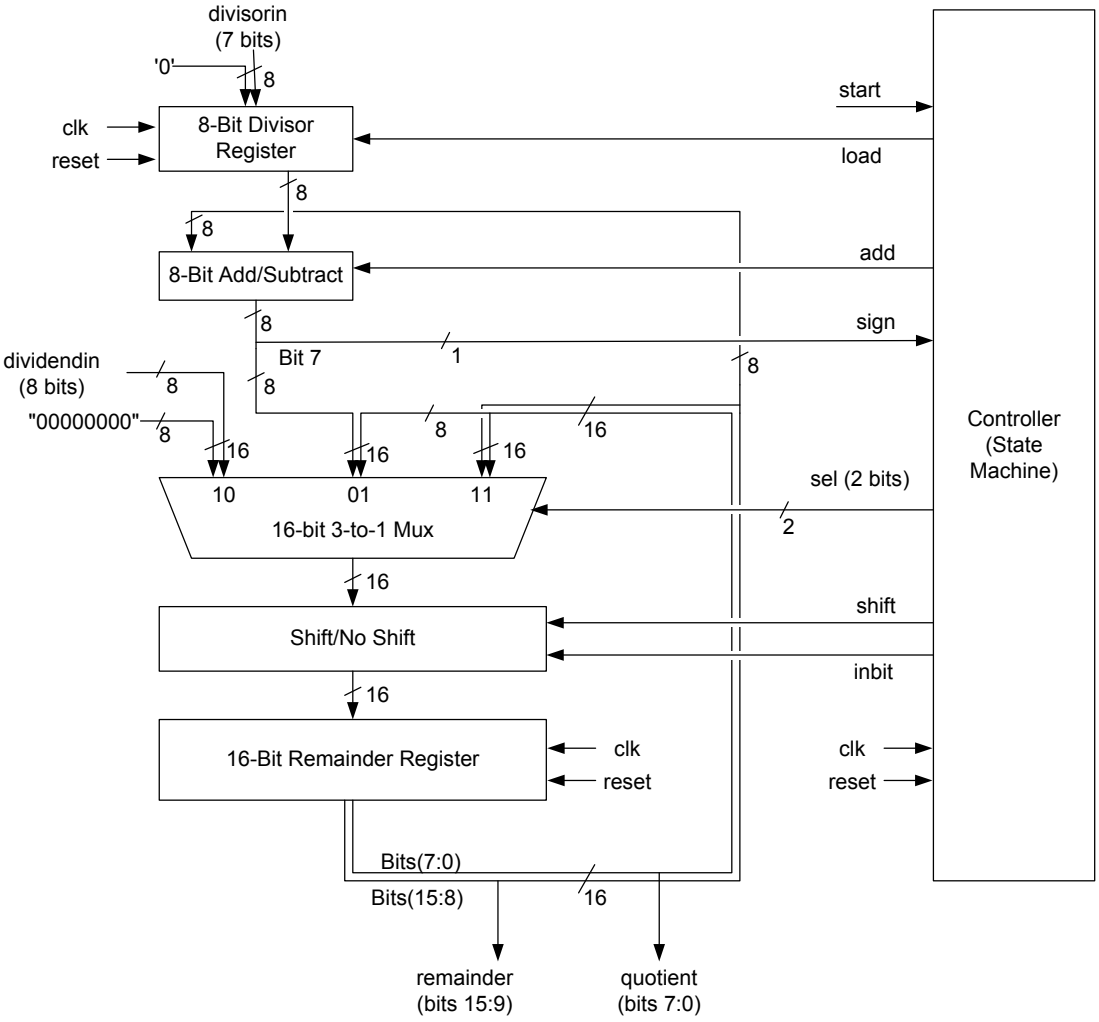
Implementation in Hardware

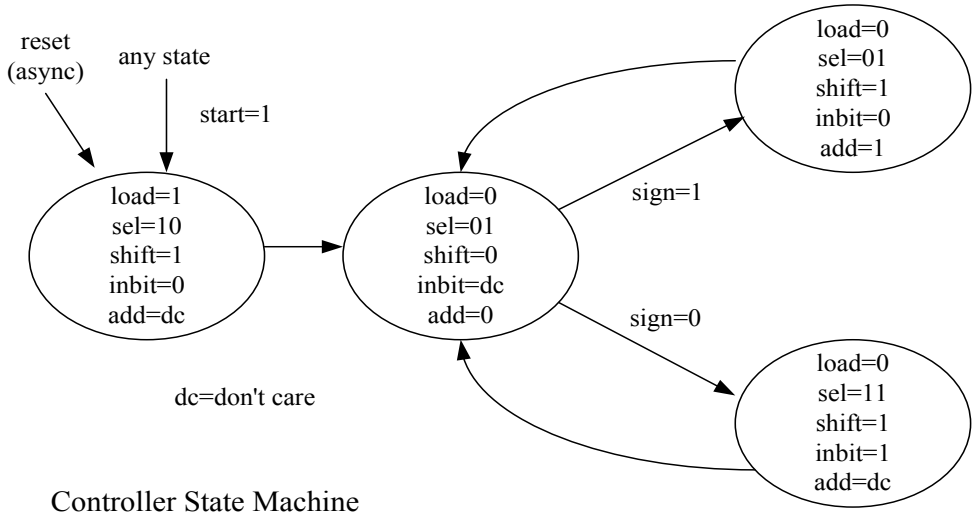
The previous section described the algorithm the chip will employ. The next task is to implement this algorithm in hardware. It may be tempting to just write some Verilog code that follows the flowchart exactly (it would then look like a piece of software). This may be useful to help you understand the algorithm, but it doesn't help with the design of the hardware. Instead, we need to think about what sort of datapath and controller would make sense to implement this algorithm.

The first diagram on the next page shows one circuit which can be used to implement the algorithm. The left side of the diagram shows the datapath, which includes the adder/subtractor, shifter, a multiplexer, and the two registers. The right side is the controller, which asserts the control signals load, add, shift, inbit, and sel (this latter control signal is actually two bits wide). The controller is where the "brains" are: it uses a state machine to assert these control signals at the correct times to cause the datapath to perform the required operations on the data. A bubble diagram showing the operation of the state machine is also shown on the next page.

Note that the state machine does not actually stop after 8 iterations as suggested by the flowchart in the previous section. However, to assist debug and usage the valid signal marks the first cycle on which the output is valid.

Also note that you might be able to find a more efficient way to implement the algorithm. If so, feel free to use it. But again, remember that you have to lay out the circuit, so be careful what you choose.





Controller State Machine
Bubble Diagram

Task 1: Set up your Account

The files you need are in ~eece479/proj_files. Copy these files to your own directory using:

```
cp –r ~eece479/proj_files  ~/project
```

(the –r copies the directories recursively).

Task 2: Describe Behaviour in Verilog

The first thing to do for the project is to describe the behaviour of the circuit using Verilog. It would be tedious to describe every block in the circuit on the previous page in a separate Verilog module. Yet, as outlined earlier, describing the entire circuit in a single module doesn't give any insight into what the hardware will look like. Therefore, I'd like you to implement the circuit using three Verilog modules: one to describe the controller (state machine) and one to describe the datapath, and one to combine the two into a top-level design.

Here are the specifics. You can find template files in ~/project/specification (you should use these template files to ensure your design works with our tester).

Controller: Filename: controller.v
I/O: (in this order):
load: output
add: output
shift: output
inbit: output
sel: output (2 bits)
valid: output
start: input
sign: input
clk: clock input
reset: async reset

Datapath: Filename: datapath.v
I/O: (in this order):
remainder: output (7 bits)
quotient: output (8 bits)
sign: output
divisorin: input (7 bits)
dividendin: input (8 bits)
load: input
add: input
shift: input
inbit: input
sel: input (2 bits)
clk: clock input
reset: async reset

Top Level: Filename: divider.v
I/O: (in this order):
remainder: output (7 bits)
quotient: output (8 bits)
valid: output
divisorin: input (7 bits)
dividendin: input (8 bits)
start input

clk: clock input
reset: async reset

Once you have completed your design, you must test it. A testbench file is in ~/project/specification/test_divider.v. The tests provided only test a few sample divisions. This probably isn't sufficient. When we test your circuit, we will use a much more extensive set of test vectors (and we won't give you these test vectors). Thus, it would be a good idea to add enough test vectors until you are sure your specification is correct. Note also that this testbench does **not** check the valid output. You should modify the testbench to do this checking as we will be checking this in our testbench when we mark your project.

Task 3: Floorplan your Design

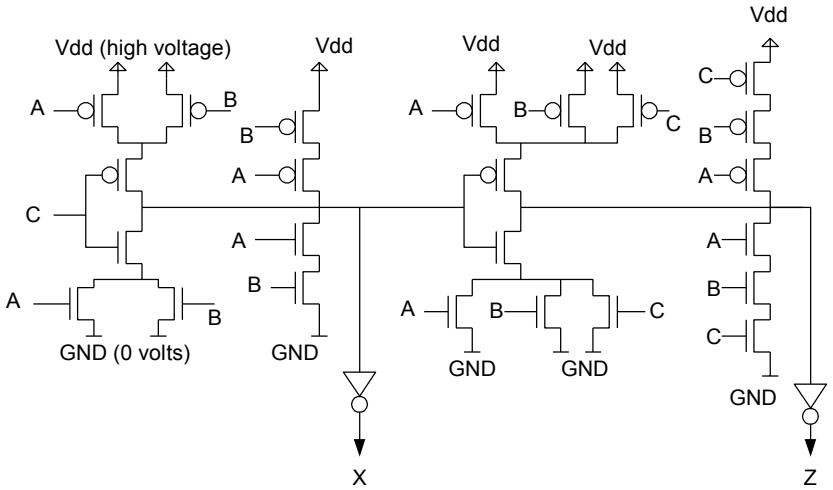
In the next few tasks, you are going to lay out this circuit. In order to plan your wiring, you should first create a floorplan. On a piece of paper, draw an outline of how you plan to lay out the cells. You should not provide the details of each cell; rather, each cell should be drawn as a box. At the periphery of each box, label the inputs and outputs of each cell, along with the metal/poly layers that these signals will use. Use the datapath techniques discussed in class, and attempt to make sure that as many connections as possible can be made by abutting cells. Also draw in the wires that must travel over the cells. Think about how you plan to lay out the controller logic. Also think about how you will distribute power, ground and the clocks. Note that even though you will not hand in this floorplan, the quality of the layout you produce will be significantly impacted by how carefully you plan your design in this task. In the worst case, if you don't plan well, you might find yourself laying out cells more than once if you get the wrong aspect ratio/IO locations the first time. This would be really bad.

Task 4: Layout Datapath Cells (this task will take a lot of time)

Lay out one bit of each of the following datapath elements. Each of these files should be placed in the directory ~/project/cells

a) A one-bit rising-edge triggered register, with asynchronous reset. Use any flip-flop circuit design that matches this specification. Use the filename reg1.mag. The positions of the inputs/outputs should match your floorplan from Task 3.

b) A one-bit full-adder/subtractor. Use the filename addsub1.mag. A suggested schematic for the full adder is shown below: (you've seen this before!).



c) A one-bit 3-to-1 multiplexer. Use the filename mux1.mag

d) One bit of the shift/noshift block. Use the filename shift1.mag

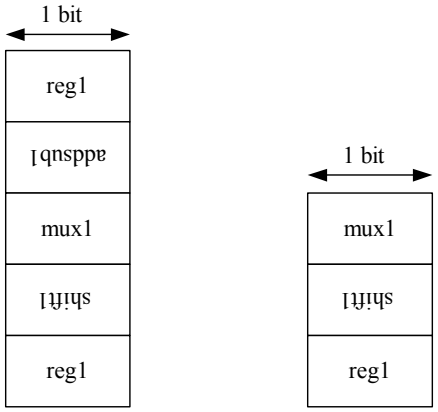
e) A single AND gate to create the qualified clock needed for the divisor register (Question to think about: Why does the divisor register need a qualified clock?)

For each of these blocks, I *strongly strongly strongly* urge you to extract the cell and simulate it using IRSIM. It won't take too much time, and will save much time later. Also, note that, depending on your floorplan, you may decide to leave extra space in each cell for metal strips to travel over top of the cell (you'd add these later, or else you can include them as part of the cell).

Task 5: Construct one bit of the datapath:

In this task, you will construct one bit of the datapath. Actually, since bits 0 through 7 are different than bits 8 through 15, you will design two bits: dbithigh and dbitlow. You should create these files in ~/project/layout/dbithigh.mag and ~/project/layout/dbitlow.mag.

Here's an example of what the two bits might look like (your layout may be different):

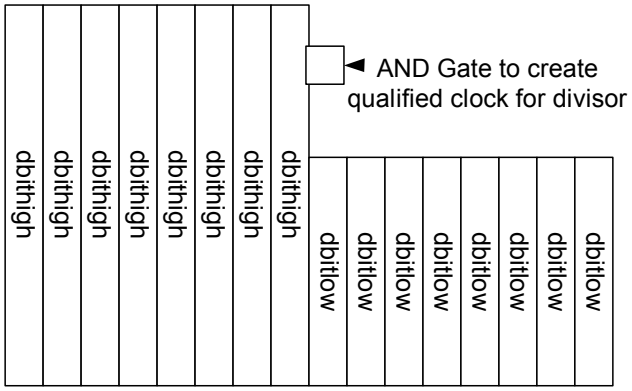


Note that, in some cases, not all high bits and not all low bits are identical (or they may be). Think about the best way to deal with this (hint: you probably don't want to create new layouts for each bit). At this point, I think you will agree that a few extra gate delays is less of a big deal that an extra hour of layout time.

It is likely that you will need to add some metal strips to connect cells that are not vertically adjacent (ie. the output of the bottom reg1 and the inputs to the mux). If you did your floorplanning carefully, you might be able to lay these wires over the cells. It may be that you actually included these strips as part of the layout of the reg1 and shift blocks; if so, lucky you!

Task 6: Construct the Datapath:

Now things really get fun. You are going to combine the bits to produce the datapath. Create the file ~/project/layout/datapath.mag and instantiate 8 copies of dbithigh and 8 copies of dbitlow. Your layout might look something like this:



Note that, if you did a good job in floorplanning, everything should abut together nicely. If you didn't do a good job, you'll have to manually connect the cell inputs and outputs. Note that, in any case, you probably need to add some connections.

Extract your design and simulate it. Note that the behaviour of this datapath should be identical to the behaviour of datapath.v (from Task 2), except for gate delays. If it is not, you need to fix it before moving on. I expect that most people will have to do *some* debugging at this point.

Task 7: Layout of the Controller

Examine the state machine of the controller. When you specified the Verilog behaviour, you did not need to worry about the exact state encoding or the logic equations. Now, however, you do.

a) Choose a suitable state encoding and work out the next state and output equations. Don't skimp here; try several state encodings. Think hard; the smaller your next state and output equations, the less layout you will have to do. After getting through Tasks 4-6, I'm sure you can appreciate the desire for small equations.

b) Find a way to lay out the controller. You may want to lay out a few cells and replicate the cells to implement the logic function. You might want to use a PLA-approach. Absolutely anything is allowed here, as long as you wind up with a layout that correctly implements the controller. Note that this task can either be fairly easy, or really really hard, depending on how hard you think about your strategy before you start. Any cells you need within your controller should be placed in ~/project/cells (use appropriate file names). Note that you already have a cell reg1.mag, which will come in handy here. The top-level layout of the controller should be placed in ~/project/layout/controller.mag

c) Extract the controller, and simulate it to make sure it works exactly as in Task 2. Again, you should not go on before the simulations of the Verilog and layout match.

Task 8: Putting it all Together

This is it! Create a final layout in cell ~/project/layout/top.mag. Include your controller and datapath cells. The inputs and outputs of the chip should be placed on the periphery of the chip, and given *exactly the same names as in Task 2*. Note that only one instance of each input/output should appear. You will also have Vdd and GND inputs; you can have multiple Vdd and GND inputs if you want.

Extract your chip, and test it using IRSIM. A sample .cmd file is in ~/project/layout/divider.cmd. This file tests only one division; be sure to test many different inputs, to convince yourself your design is correct. If all tests work, you are done! If not, you have some debugging to do.

Suggested Milestones:

The following are some suggested milestones for this project. We aren't going to be checking your progress, but you can use this list to see if you are falling behind.

Milestone 1: Finish Tasks 1, 2 and 3: Mar. 11, 2015

Milestone 2: Finish Tasks 4: Mar. 18, 2015

Milestone 3: Finish Tasks 5 and 6: Mar. 25, 2015

Milestone 4: Finish Task 7: Mar. 27, 2015

Milestone 5: Finish Task 8: April 8, 2015

What to hand in:

If you have followed the above instructions correctly, all your files will be in the right place. The only thing you need is an info.txt file in **~/project/info.txt**. This file has four parts. In the first part, include the names, numbers, and email addresses of each of your group members. Be sure your email addresses are valid, because that is where your mark will be sent. Separate your names from your email address by a comma.

The second part of the file asks you to specify whether your design works or not. You should enter a **Y** or an **N** (only **Y** or **N**). If your design worked, use **Y**. If your design didn't work, use **N**. Note that if you say **Y,** and your design doesn't work, you will lose many more marks than if it doesn't work and you say **N**, so make sure you test your design thoroughly before saying **Y**. Also, notice you have to say **Y** or **N**; no Maybe's. You *cannot* get full marks if you leave this line blank.

The third part of the file can contain a text message explaining either why the design did not work (what you think the problem is: the more specific you can be, the more part marks you will get) and/or anything exceptional about your design you want the markers to know. Filling out this part of the file is optional.

The fourth part of the file asks you to disclose any help you might have received in completing the assignment. If you received any help (other than from the professor, the TA, or other members of your group) disclose it here. If you have not received any help, enter **NONE**. Do not leave this section blank. We will be carefully comparing assignments to look for unauthorized collaboration. Any un-disclosed collaboration will be reported to the Dean for possible referral to the discipline committee. Disclosing any help you may have received protects you from this (of course, you may not receive full marks if you copied someone's cell, but that is better than having the case referred to the Dean). If this is unclear, please contact the professor or TA *before* handing in your assignment.

PLEASE NOTE THAT WE WILL BE CAREFULLY COMPARING YOUR PROJECT VERSUS OTHERS IN THE CLASS (OR EVEN OTHER YEARS, ALTHOUGH THEIRS PROJECTS ARE SLIGHTLY DIFFERENT). MULTIPLE STUDENTS HAVE BEEN CAUGHT IN THE PAST BECAUSE OF THIS AND FAILED THE COURSE AS A RESULT. THE SOLUTION IS TO MAKE SURE THAT EVERY TRANSISTOR IN YOUR DESIGN WAS CREATED BY YOU DIRECTLY.

Once you are done, and have created the info.txt file, zip up the "project" directory you've been working on, and submit both the ZIP and TXT files to UBC Connect.

Then you are done!!!