Lab 5 Report

By *Automagically*

Sean Murphy
U49850246
(33%)

Jesse Walton
U89823440
(33%)

Austin Matthew
U89904116
(33%)

Introduction:

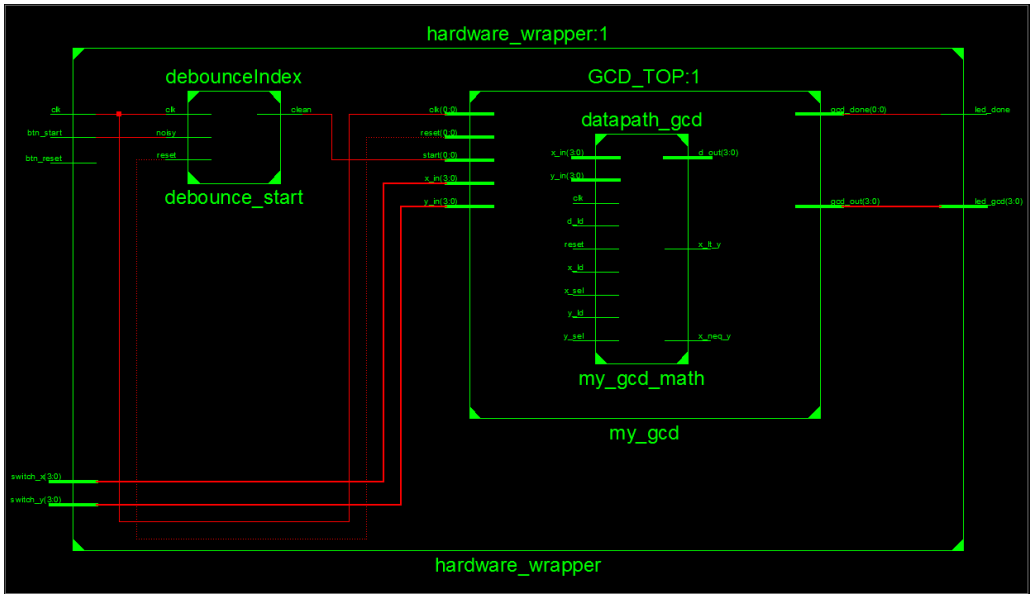
The purpose of this lab is to implement an algorithm to compute the greatest common denominator of two values on an FPGA. To that end, start and reset buttons are needed as well as x and y inputs that will come from switches 0-3 and 4-7 respectively. The outputs required for the module are named GCD_OUT and DONE. GCD_OUT displays the greatest common denominator of the two values and the DONE output goes high when the computation is complete. These outputs are displayed via leds.

The correct operation of this lab will allow the user to power on the FPGA, load values x and y by setting the switches and then begin execution by pressing the run button. The algorithm will cycle through until the greatest common denominator is computed and then the program halts, displaying the correct information. The results will stay displayed until the start button is pressed, at which point the program will cycle through again.

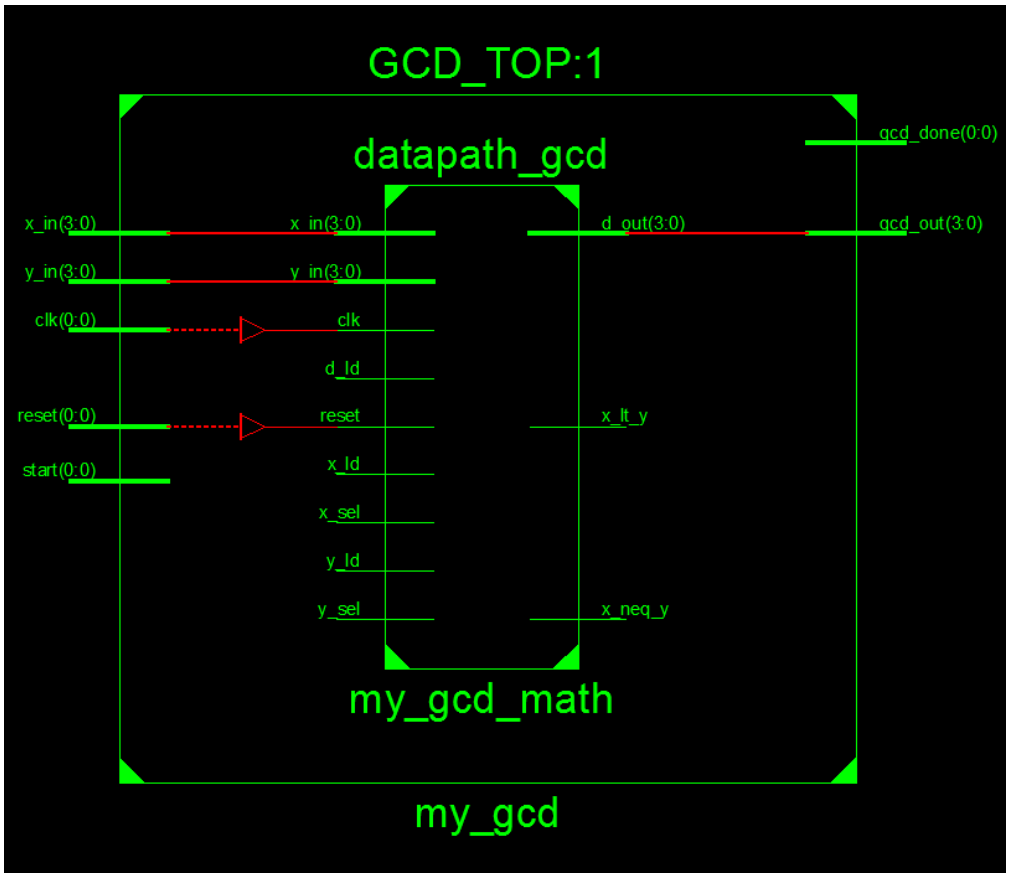
This lab requires two components implemented independently. These components are known as the datapath and the controller. Although we were given the algorithm to calculate the greatest common denominator, implementing it in hardware using this split method looks to be both powerful and require a bit more planning to execute correctly.

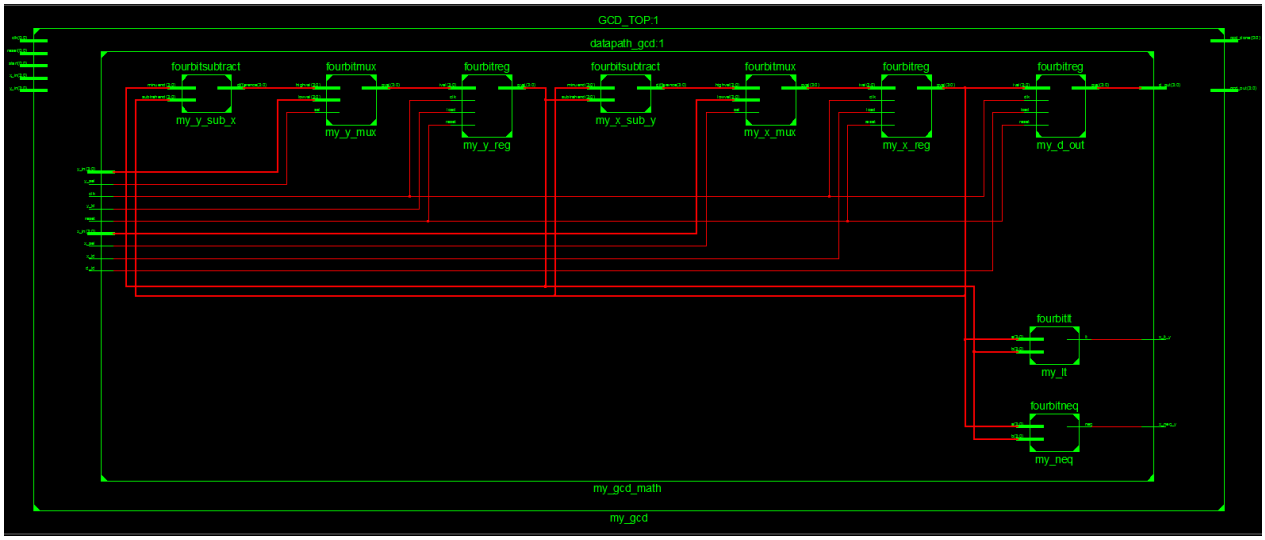
Design:

The design of this lab is broken down into one top level module with inputs for the x and y values (4 bits) coming from the DIP switches as well as the start button and reset buttons. The top module also has two outputs, routed to the LEDs. The two outputs display when the program is finished executing and also displays the GCD that was calculated.



One level beneath the top module there are three sub-modules consisting of the button debounce, the datapath and the controller.





In implementing a controller and datapath to compute the greatest common denominator, we needed to first write out the pseudocode for the controller and design the structural components for the datapath. The idea behind the controller and the datapath is to separate the calculations from the current state. As a result, the datapath calculates the modules asynchronously, determined only by the current value of the registers while the controller routes the signals and register values.

controller intro

The controller is responsible for the current state of the machine. It was designed by taking the greatest common denominator algorithm and breaking down each step into a specific state. If statements, switch statements and while loops were implemented as conditional jump statements. In those states, the next state is determined based on the values returned from the datapath.

Initially, the controller is in a wait state that loops to itself until the start button is pressed. When the button is pressed, the second state loads the x and y registers with their respective muxes set to read from the external inputs. After loading the registers, they are compared together. If equal, the program jumps to the end. If not equal, the algorithm subtracts the smaller number from the larger number and checks again if they are now equal. This process repeats until they are equal, at which point the GCD is found. The GCD value is output and the algorithm is reset to the beginning, awaiting the start button to trigger the process again.

```
state01:                    // initial state, wait here until start button pressed
        d_load = 0;         // reset GCF found line
        if (start_button)   // loop until start button is pressed
                state = state01;
        else
                state = state02;

state02:
        x_mux_sel = 0;      // change reg mux to read from external input
        y_mux_sel = 0;      //
        x_load = 1;         // load current input into register
        y_load = 1;         //
        state = state03;    // jump to state 3 unconditionally

state03:
        x_load = 0;         // stop loading register
        y_load = 0;         //
        x_mux_sel = 1;      // switch mux to internal input (vs. external)
        y_mux_sel = 1;      //

        if(!(x != y))       // if x == y
                gcd_done = 1;           // GCD is found
                state = state06;
        else if (x < y)     // else if x < y
                state = state04;
        else                // else if y < x
                state = state05;

state04:                    // x <  y, subtract x from y, put on y input
        y_load = 1;         // load current input into y register
        state = state03;

state05:                    // y < x, subtract y from x, put on x input
        x_load = 1;         // load current input into x register
        state = state03;

state06:
        d_load = 1;         // output GCD found to the top level wrapper
        state = state01;    // jump to beginning
```

datapath intro

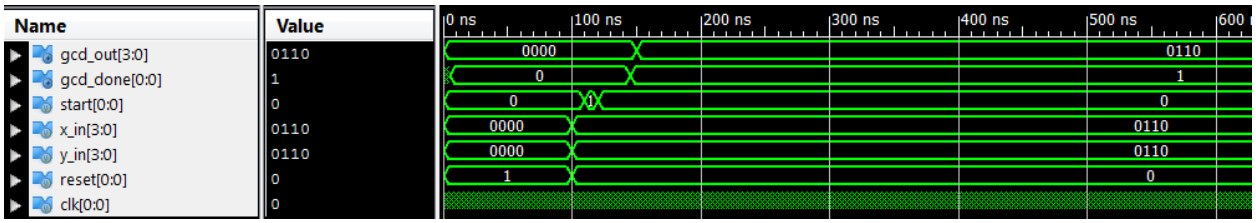
The datapath is responsible for storing inputs, performing calculations on them and outputting them back to the controller. Submodules within the datapath consist of ‘not equal to’, ‘x subtract y’, ‘y subtract x’ and ‘x < y’. These modules have their inputs hardwired from the registers. The registers, however, receive their inputs from a multiplexer that selects from external input or the internal input, after a calculation has been completed on it.

The purpose of the datapath in this project is to store the current values in registers x and y. The datapath is also responsible for comparing x and y and subtracting x from y or y from x, based on the inputs received from the controller. Essentially, the datapath does computations as soon as the values appear in the registers, but the outputs are only sampled as needed by the controller.

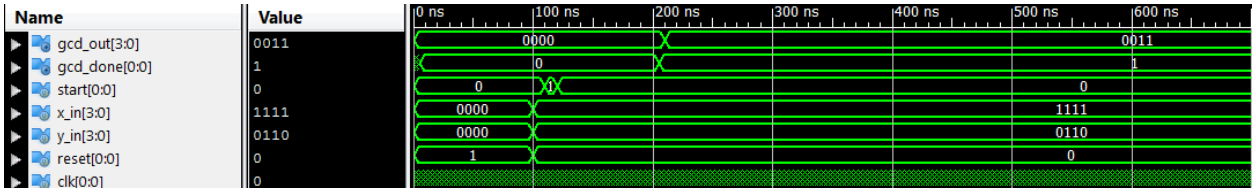
Testing:

We took the following scenarios and tested them in the simulator and then again on the physical board.

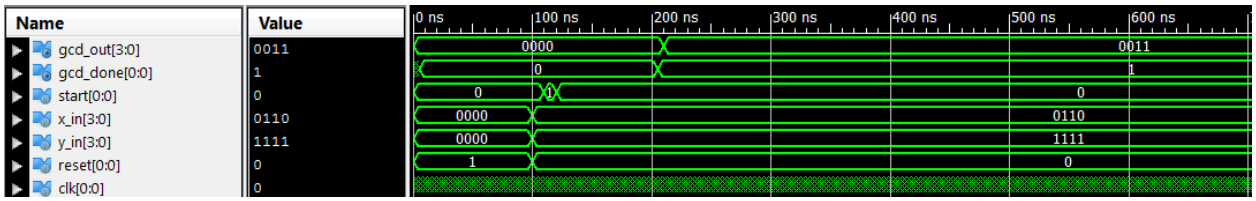
X=Y



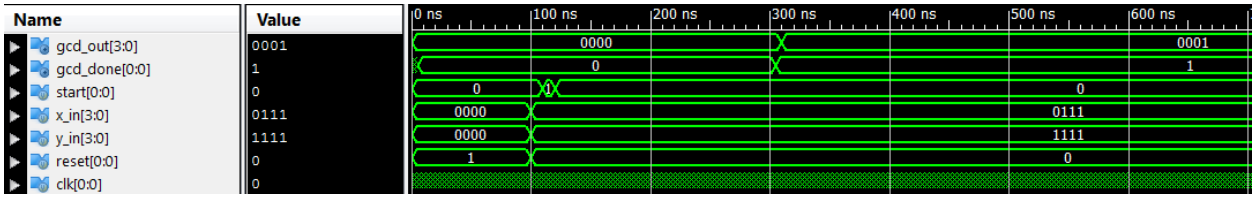
X>Y



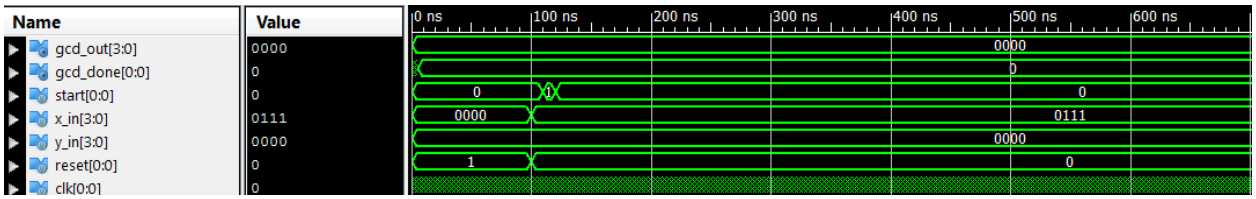
X<Y



Prime number



Zero as input



Results:

Upon uploading our program to the board we proceeded to try the inputs that we had previously simulated. All outputs matched their corresponding inputs from the simulation. We faced only one problem and that was when an input of zero was entered. This could have been resolved by a the addition of a behavior block where we assigned GCD_Done a high values as soon as the inputs were checked to see if either was zero. We decided to leave this part out as it was not included in our orginal assigned algorithm.

Conclusion:

In conclusion, this lab used a mixture of a finite state machine and an ALU type module to achieve the implementation of an algorithm on hardware. The finite state machine portion of this lab was almost identical to the last labs finite state machine, however, the inclusion of the datapath increased the complexity of the project while also increasing it's ability. The last lab was only able to change states based on the pre-determined next states while this lab is able to calculate the values as needed through the use of the data path.

The idea of having two separate modules interacting together to compute a final product was not a new one in hardware, but in separating the implementation from the data makes it seem closer to programming with functions and abstract data types. Although designing the project like this seems less intuitive, the more time we spent

with it, the more sense it makes. The controller is responsible for the state of the machine and the datapath is responsible for the inputs to that state machine. The separation of the two allows for creation and expandability by adding states, output/input triggers and datapath modules as needed. This is the case because the states and datapath are separate otherwise and are not dependent on any other past state or any other value in the datapath, you are able to explicitly state what values you need and what signals you need to send to get them. With that information, you are able to make decisions to determine the next state of the machine.