Two Function Calculator

Final Project Report
December 9, 1999
E157

Mark Holland

Abstract:

This project prototypes a two function calculator consisting of a 16 key keypad, FPGA, and 8 digit LED display. The two targeted operations are addition and subtraction. Inputted numbers are positive base ten integers of 8 or fewer digits. The user types in an arithmetic equation in four steps: they type the first number, an operator, the second number, and lastly an equals operator. Each inputted number is placed in a shift register in the FPGA while also being displayed on the LEDs. The operator is also placed in a register. When an equals sign is inputted, the proper operation is performed on the two numbers and the resulting number is put in a register and displayed on the LED display.

Introduction

Mathematics is a building block upon which all of the sciences are dependent. Being able to perform simple arithmetic operations quickly and efficiently is a necessary tool in all scientific fields. Calculators were created in order to give people a simple, fast, and error free method of doing these calculations.

I chose to prototype a calculator because they are one of the most basic and important tools for an engineer such as myself. Being able to design and understand the hardware of a calculator is a good starting point from which I can go on to design and understand more complicated devices.

My calculator performs the operations of addition and subtraction. It operates in base 10 and is capable of accepting, displaying, and operating on any numbers in the range 0 to 99,999,999. If answers occur that are not in this range an overflow/negative indicator light is used to show that the outputted number is not in the acceptable data range. A clear button is used to clear the memory and the display of any contents.

From the cleared (empty) state, my calculator works by accepting four inputs from a user: a number (from 0 to 99,999,999), an operator (+ or -), a second number (from 0 to 99,999,999), and lastly an equals operator. The inputs are made on a 16 key keypad, which is shown in Figure 1.

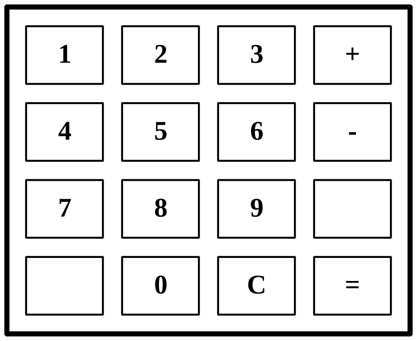

Figure 1: 16 Key Keypad

Each inputted number is displayed on the LED display as the number is being inputted, and the answer is displayed after the equals operator is entered. The outputted numbers are displayed on four dual 7-segment displays, combined as shown in Figure 2.

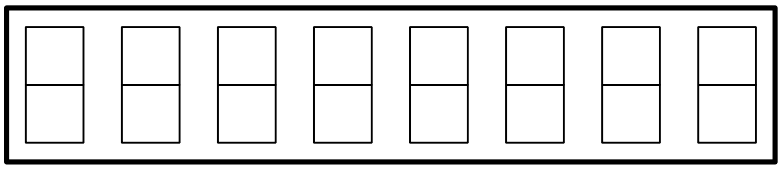

Figure 2: LED Display

My calculator uses an FPGA almost exclusively, with a small amount of external logic on my board. Inputs come from the keypad and outputs go to the display. Inputted numbers and operators are shifted into registers on the FPGA as they are entered on the keypad. Inputted numbers and the eventual answer of the operation are displayed on the LED display at the proper times. The FPGA multiplexes the 8-digit display such that each digit is only on 1/8 of the time.

The only external logic that I use is 8 PNP transistors that control the switching of the 8 digits in the display. The FPGA was inadequate in its current sinking/sourcing capabilities, which required the use of these transistors.

The most basic block diagram for the design appears in Figure 3. From the keypad a button is pressed, the proper operations are done inside the FPGA, and the proper segment mappings are outputted to the display.

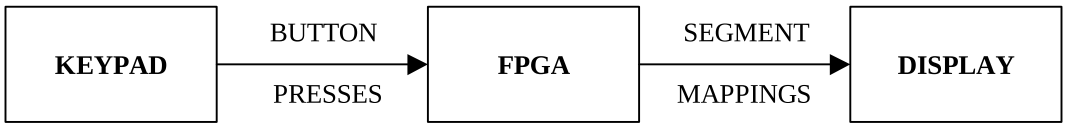

Figure 3: Basic Design Block Diagram

Another basic block diagram can be made just to describe the FPGA. This block diagram appears in Figure 4. The inputted number is first decoded into either a number or operation input and put into a register. By deciphering the states of the registers the proper number can be chosen and mapped to LED mappings. These mappings tell the display which segments to turn on.

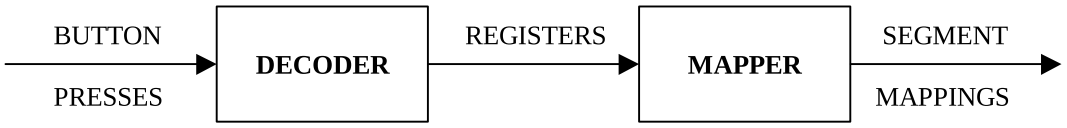

Figure 4: Basic FPGA Block Diagram

Much more goes on in the FPGA but a lot of it is regulation or control of the items shown in the block diagrams and not actual blocks. The entire FPGA design is covered in a later section.

Schematics

The schematics for my calculator circuit appear in Figure 5. The wires are named after the FPGA inputs and outputs to which they are connected (the names are the same as those that appear in the Verilog code in the Appendix). Poll[3:0] are inputs to the FPGA and the rest of the wires are outputs of the FPGA.

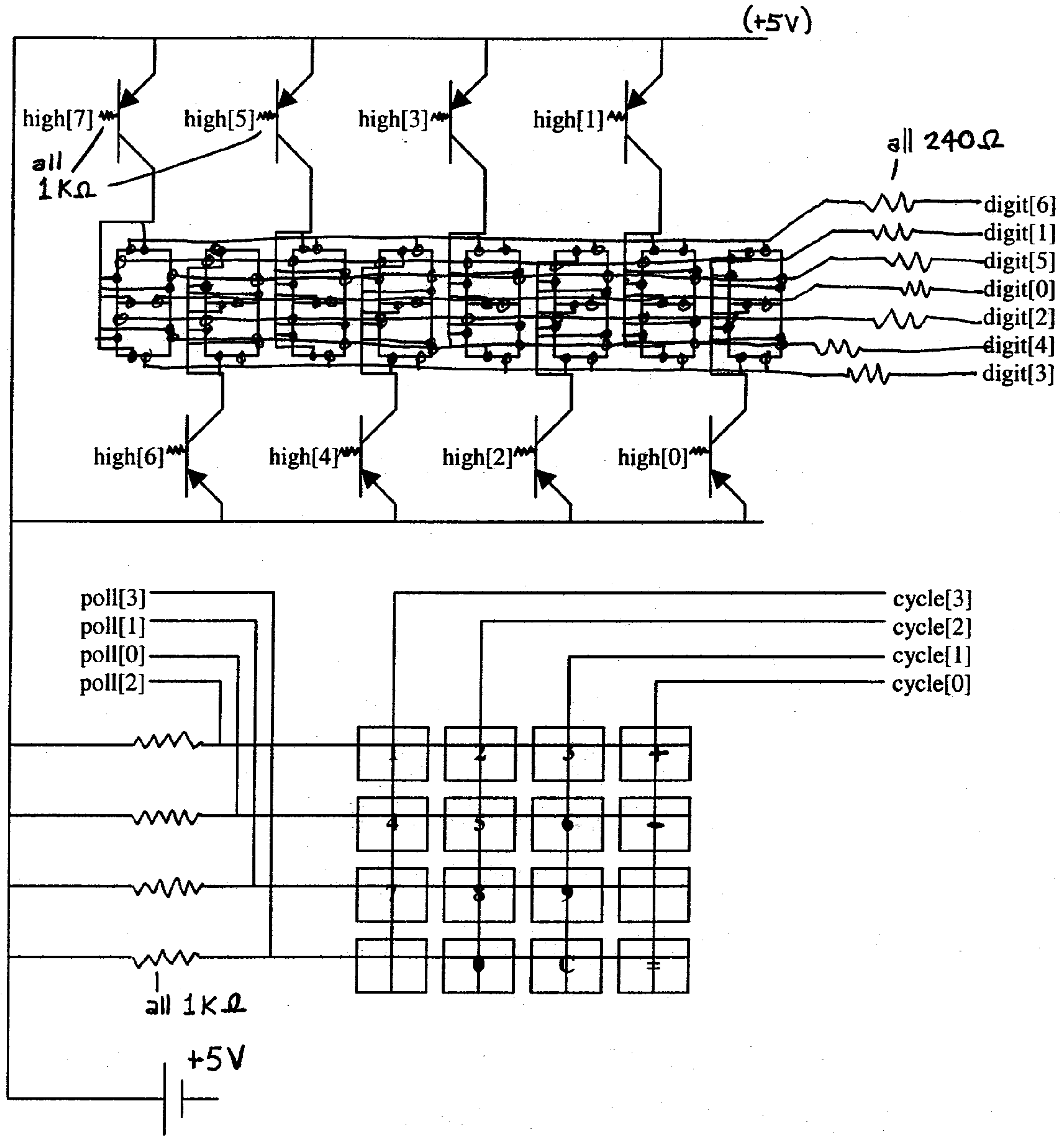

Figure 5: Board Schematics

The keypad contains a matrix of wires that can be shorted by pressing the buttons. For example, pressing button 1 would short the wires CYCLE[3] and POLL[2]. The FPGA cycles the columns one at a time so that there is always only one column wire that is low. If a button in that column is being pressed, the corresponding row wire is pulled low and the FPGA recognizes that a button has been pressed. By knowing which row wire and column wire are low it can then decode the occurrence into a specific button press. This is how the polling circuit works.

The LED display is a common anode display. This means that the anodes (positive side) of each of the seven LEDs on a digit are tied together. The cathodes of like segments (for example the cathodes from the top LED on each digit) are then tied together as well.

In my design, the anodes are pulled high to turn on a digit. While a digit's anodes are pulled high, the LEDs that have low cathodes will light up. This is how my display works.

Ideally, the LEDs would like to see 5-20mA of current, but my FPGA cannot source this much current. Because of this I use PNP transistors (see Figure 5) as switches to source the necessary current. To turn on a PNP transistor I drop the base voltage of the transistor to 0V. The current that the LEDs see is then determined by the value of the base resistor (here 1-Kohm), where a lower resistance would allow more current to go through the transistor. I also have 240-ohm resistors in series with the LEDs to limit the current to 21mA so that the LEDs are not damaged.

As an example, to display the number 3 in the least significant bit I would set the following levels: high[0] = 0, digit[0] = 0, digit[3] = 0, digit[4] = 0, digit[5] = 0, digit[6] = 0, and the rest of the digit and high variables would = 1.

FPGA Design

My Verilog code is divided into seven modules, each of which appears in the Appendix. I will attempt to introduce and describe them in order from input to output.

main.v

Main.v is the top-level module that organizes my overall design. It contains all of the FPGA inputs and outputs and also contains wires for variables that are used across multiple modules. I will describe most of these variables in the specific modules in which they are used.

The two inputs that I will describe here are *clk* and *reset*, because they are special cases. *Clk* is the 2Mhz clock that is generated by the FPGA. *Reset* is a global reset variable that appears in all of the modules. When *reset* is entered, all of the variables in all of the modules get reset to their default values. This is shown by the fact that every always statement within my design has logic designed to reset all variables.

slow_clock.v

The clock that the FPGA runs off of is a 2 MHz clock, which means that it has a cycle time of 500ns. I have two different applications in my design that set upper and lower limits on the frequency that I can operate my hardware at. The debouncing circuit that appears in decoder.v (to be discussed later) requires a clock with cycles of approximately 5ms (200Hz). The display must multiplex its eight digits such that each digit is flashing at around 50Hz or faster. To accommodate both of these requirements I chose to use a clock that operates at 488Hz with a cycle time of 2ms. This clock gives me 61Hz flashing for my display (which is acceptable), and gives me a 2ms-cycle time for my debouncing circuit which I experimentally found to be an adequate cycle time.

Slow_clock.v is the module that creates a 488Hz clock. It inputs the FPGA clock (*clk*) and, using a counter, creates a system clock named *trigger* that operates at 488Hz. If *clk* were the least significant bit of a multiple bit signal, *trigger* would be the twelfth bit of the signal. Slow_clock also creates a three-bit clock, *clk2*, which is used in the other modules to control the timing of hardware operations.

decoder.v

Decoder.v is the module that is used to handle keypad inputs. It works by cycling the bits *cycle[3:0]* such that the four bits alternate at being 0, and then monitors the bits *poll[3:0]* to see if any of them are pulled low. When a button is pressed (see Figure 5) and the column and row wires are shorted, the row gets pulled to the value of the corresponding column.

The basic hardware flow implied by decoder.v appears in Figure 6. In the figure, boxes represent hardware, lines entering from the tops are inputs, lines entering from the sides are enable lines, and lines coming out of the bottoms are outputs.

The debouncing circuit in decoder.v is designed such that an input must be sensed on two consecutive clock cycles for it to be valid. This is because when a key is pressed the contacts between the row and column wires may bounce. The debouncer must be able to ignore inputs that happen on only one clock edge and realize that they were due to bouncing and not an actual input. This way each button press is sensed only once and the keypad's bouncing does not effect the input.

The variables *hit [1:0]* and *miss* are used to keep track of the consecutiveness of inputs. *Hit* is fed to the decoding hardware as enabling lines, so that inputs are only decoded after being debounced. The decoder code starts on line 132 of decoder.v. Once the enable line (*hit)*, is sensed, two logic blocks take the *cycle* and *poll* lines and decipher what the input was, outputting a new control line *num[3:0]*.

The *num* control line is fed as an enable line to five hardware units depending on the type of input that was sensed: numeral input, addition operator, subtraction operator, equals operator, or clear button.

In the numerical input unit there are two more units which are controlled by the signals *plus* and *minus*. Logic determines whether *plus* or *minus* is true and enters the first unit if it is true (which means that the entered digit is part of the second inputted number). If it is not true then we enter the second hardware unit, which decodes digits for the first inputted number.

Both numerical input units (for first and second number) operate the same way. Upon numerical inputs the first thing that happens is that the previous numerical inputs are shifted down a register in the shift register. The input is then decoded into the low end of the shift register by the use of a multiplexer. These numbers are held in *dig0[3:0]* through *dig7[3:0]* variables for the first inputted number and *dig2_0[3:0]* through *dig2_7[3:0]* variables for the second inputted number. The wires for these variables are also made accessible to the hardware in other modules.

If an addition, subtraction, or equals operator is sensed, the corresponding control line is set high to indicate which part of the equation we are in and what operator we are using.

If the clear button is sensed, all of the variables in the module are reset to their default values. This means that *plus*, *minus*, *equals*, and all of the digits are reset to zero.

Logic also exists to prevent the user from giving bad inputs. Once a plus or minus operator has been chosen, the unchosen logic block of the two becomes disabled so that the calculator cannot try to do addition and subtraction. Also, the equals logic unit cannot be entered until an operator has been chosen. The only debugging logic that did not get entered into the module (due to a lack of space on the FPGA) is logic that would prevent the user from inputting numbers after = has been entered. In the completed calculator, the user can continue to edit the second inputted number (and therefore the answer) after the equals button has already been hit.

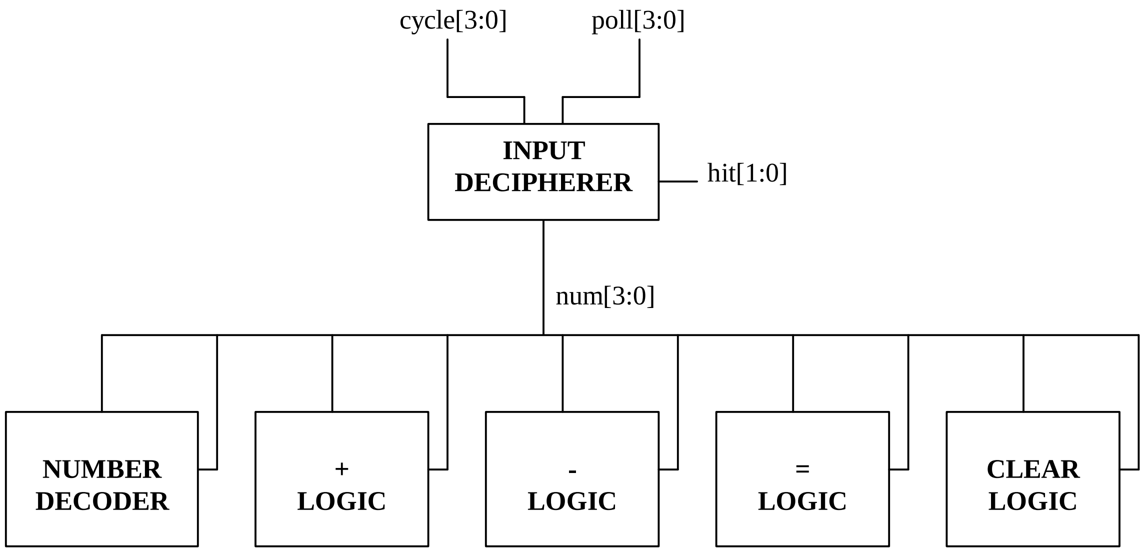

Figure 6: Decoder.v Hardware Flow

operator.v

The operator.v module is necessary for performing operations on the two inputted numbers. It inputs the 8 digits of each of the first two numbers (*dig0 – dig7*, *dig2_0 – dig2_7)*, the operators *plus*, *minus*, and *equals*, and it outputs the answer variables *ans0[4:0] – ans7[4:0]* and the wire *LED* which indicates an overflowing/negative answer.

The hardware implied by operator.v is all enabled by the wire *equals*. The two numbers are not operated upon until the equals button is pressed and *equals* is set to one. The hardware for operator.v is shown in Figure 7.

Multiplexers are used to choose between the digits of the first and second numbers depending on the state of *clk2[2:0]*. A multiplexer also sets *enable[7:0]* so that *sum[4:0]* can be mapped to the correct answer register. On any given clock cycle the two inputted numbers will be added or subtracted along with *c_in*, their carry out (*c_out)* will be determined, and the result will be put in the correct *ans* variable depending on the state of *enable*. Note that for the least significant bit addition there is a multiplexer on the carry line to make sure that no carry in is added or subtracted.

The wire *LED* is also set in operator.v. *LED* is set to go high whenever there is a carry out (*c_out)* at the same time that *clk2* = 7 and *equals* is high. This indicates either an overflow for addition or a negative number for subtraction.

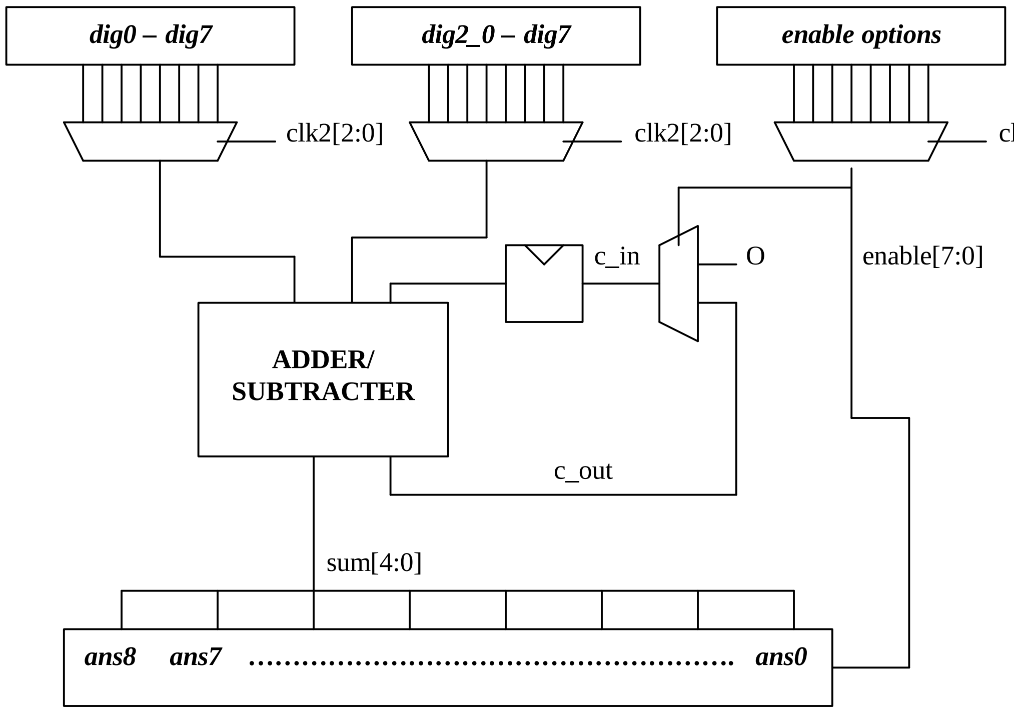

Figure 7: Hardware for operator.v

mapper.v

Mapper.v is responsible for taking in the input and output numbers of the equation and mapping them to the LEDs. It inputs both of the user-inputted numbers, the answer, *plus, minus,* and *equals,* and it outputs the display mappings, *key0[6:0] – key7[6:0]*.

Mapper.v starts by using *equals* to enable logic that will map the *ans* variables into *temp* variables for later use. If *equals* is not high, it then has an enable line of *plus* or *minus* enabling logic that maps the *dig2_* variables (from the second digit) into the *temp* variables. If neither of the first two mappings occurred it maps the *dig* variables (from the first digit) to the *temp* variables.

Once the eight *temp[3:0]* variables are set, a multiplexer uses *clk2[2:0]* to choose between them and to put the proper one into *hold[3:0]*. Depending on the value of *hold*, another multiplexer then puts the correct LED mapping into a *key* variable, which is then outputted to the display modules.

display.v

Display.v is responsible for taking the *key* LED mappings from mapper.v and outputting them to the display. It inputs *key0 – key7* and outputs a variable *digit[6:0]*.

Display.v is simply a multiplexer that uses *clk2[2:0]* to choose between the 8 mapped *key* variables and sends the proper mapping to the display.

display2.v

Display2.v is responsible for pulling the anodes of a digit high at the correct time. It outputs *high[7:0]* to the 8 digits of the display.

Display2.v is simply a multiplexer that uses *clk2[2:0]* to cycle through all of the display digits, pulling one of them high at a time.

Results

My initial proposal was to design and build a three-function calculator that would perform addition, subtraction, and multiplication, free of bugs. My finished calculator was a two-function calculator that performed addition and subtraction, and had one known bug.

The reason that the multiplication function did not make it into the calculator is because I ran out of room on my FPGA. The FPGA has a total of 196 configurable logic blocks, and my completed design used 193 of those blocks. In order to meet my original objectives of making a three-function calculator I wrote a module that would perform operation, but this module could not be incorporated into the design. The multiplication module appears in Appendix B.

I also proposed to have a calculator that would not accept bad inputs or display any errant behavior. As I mentioned during my discussion of the module decoder.v, I was unable to fit all of my debugging logic onto the FPGA. The result is that after the user hits equals, he can still edit the second inputted number (and therefore the answer) by pressing digits on the keypad.

I would be able to make enough room for this last piece of debugging logic if I were to replace lines 163 and 177 of operator.v with the code if (sum[3] && (sum[2] || sum[1])). This would have eliminated two adders and given me room for my last piece of debugging.

I also did not maximize the brightness of my LED display. I left 1Kohm resistors in the transistor circuits, which prevented the LEDs from seeing the maximum amount of current (21mA) that the 240ohm resistors would have allowed them to see. The LED displays are robust enough that I could have taken the 1Kohm resistors out and had a brighter display at no harm to my hardware.

The most difficult part of my design was optimizing my hardware. I began writing my Verilog code without any consideration for optimization because I was unaware that I would run out of room on my FPGA. When I ran out of room I then had to take the code I had written and I had to reduce it to the simplest hardware possible. While I believe I accomplished this in many cases, there were pieces of hardware that I did not fully optimize, like the adders mentioned above.

Another problem that I had during my design process was that I was not initially writing my code with the implied hardware in mind. I was simply writing code that would accomplish the tasks I needed, which is one of the reasons that my original code implied unnecessary hardware. Getting myself to write code with hardware in mind was somewhat difficult, but once I started doing it my code writing improved greatly.

Appendix A

This Appendix includes the Verilog files for my project. They are in the order in which they are introduced in the FPGA design section of this paper.

```
//author: Mark Holland
//program: main.v
//purpose: top level module for calculator program

module main(clk, reset, poll, cycle, high, digit, plus, minus, equals,
LED);

input clk;                    //the FPGAs clock
input reset;                        //global reset
input [3:0] poll;             //polling variables
output [3:0] cycle;                 //polling variables
output [7:0] high;                  //multiplexing variables
output [6:0] digit;                 //multiplexing variables

output plus;                        //operators, + - =
output minus;
output equals;

output LED;                   //LED for overflow/negative

wire [3:0] dig0;              //wires for the first number
wire [3:0] dig1;
wire [3:0] dig2;
wire [3:0] dig3;
wire [3:0] dig4;
wire [3:0] dig5;
wire [3:0] dig6;
wire [3:0] dig7;

wire [3:0] dig2_0;                  //wires for the second number
wire [3:0] dig2_1;
wire [3:0] dig2_2;
wire [3:0] dig2_3;
wire [3:0] dig2_4;
wire [3:0] dig2_5;
wire [3:0] dig2_6;
wire [3:0] dig2_7;

wire [4:0] ans0;              //wires for the answer
wire [4:0] ans1;
wire [4:0] ans2;
wire [4:0] ans3;
wire [4:0] ans4;
wire [4:0] ans5;
wire [4:0] ans6;
wire [4:0] ans7;


wire [6:0] key0;              //wires for the key mappings
wire [6:0] key1;
wire [6:0] key2;
wire [6:0] key3;
wire [6:0] key4;
wire [6:0] key5;
wire [6:0] key6;
wire [6:0] key7;
wire [2:0] clk2;               //wires for the slow clock
wire trigger;                        //wire for the trigger clock



//I call each of the submodules

slow_clock slow_clock(clk, reset, clk2, trigger);

decoder decoder(trigger, reset, poll, dig0, dig1, dig2, dig3, dig4,
dig5, dig6, dig7, dig2_0, dig2_1, dig2_2, dig2_3, dig2_4, dig2_5,
dig2_6, dig2_7, cycle, plus, minus, equals);

display display(clk2, trigger, reset, key0, key1, key2, key3, key4,
key5, key6, key7, digit);

display2 display2(clk2, trigger, reset, high);

mapper mapper(clk2, trigger, reset, dig0, dig1, dig2, dig3, dig4, dig5,
dig6, dig7, dig2_0, dig2_1, dig2_2, dig2_3, dig2_4, dig2_5, dig2_6,
dig2_7, ans0, ans1, ans2, ans3, ans4, ans5, ans6, ans7, key0, key1,
key2, key3, key4, key5, key6, key7, plus, minus, equals);

operator operator(clk2, trigger, reset, plus, minus, equals, dig0,
dig1, dig2, dig3, dig4, dig5, dig6, dig7, dig2_0, dig2_1, dig2_2,
dig2_3, dig2_4, dig2_5, dig2_6, dig2_7, ans0, ans1, ans2, ans3, ans4,
ans5, ans6, ans7, LED);

endmodule
```

```
//author: Mark Holland
//program: slow_clock.v
//purpose: to provide a slowed down clock for debouncing and
multiplexing

module slow_clock(clk, reset, clk2, trigger);

input clk;                    //the board's clock
input reset;                        //my reset
output [2:0] clk2;                  //my slow clock
output trigger;                     //my trigger for the always blocks
reg [14:0] count;             //my 13-bit counter

assign clk2 = count[14:12];         //assign clk2

assign trigger = count[11];         //this gives me approximately a 4ms
cycle time
                              //which will be good for debouncing


always @(posedge clk or posedge reset)    //on clk or reset...

      if(reset)               //reset resets the counting
            count = 0;

      else
            count = count + 1;      //the actual counting

endmodule
```

```
//author: Mark Holland
//program: decoder.v
//purpose: to debounce and decode keypad inputs, and to set the
//       necessary corresponding logic

module decoder(trigger, reset, poll, dig0, dig1, dig2, dig3, dig4,
dig5, dig6, dig7, dig2_0, dig2_1, dig2_2, dig2_3, dig2_4, dig2_5,
dig2_6, dig2_7, cycle, plus, minus, equals);

input trigger;                          //the trigger for operation
(clock)
input reset;                            //my global reset
input [3:0] poll;                 //polling vars for keypad

output [3:0] dig0;                      //the digits of the first
number,
output [3:0] dig1;                      //in binary encoded decimal
output [3:0] dig2;
output [3:0] dig3;
output [3:0] dig4;
output [3:0] dig5;
output [3:0] dig6;
output [3:0] dig7;

output [3:0] dig2_0;                    //the digits of the second
number,
output [3:0] dig2_1;                    //in binary encoded decimal
output [3:0] dig2_2;
output [3:0] dig2_3;
output [3:0] dig2_4;
output [3:0] dig2_5;
output [3:0] dig2_6;
output [3:0] dig2_7;


output [3:0] cycle;                     //cycle vars for polling
keypad

output plus;                            //logic for the operators
output minus;
output equals;

reg plus;                         //various necessary registers...
reg minus;
reg equals;

reg [3:0] cycle;
reg [3:0] num;                          //num deciphers keypad
entries
reg [1:0] hit;                          //hit used in debouncing
reg miss;                         //used in debouncing too

reg [3:0] dig0;                         //registers for numbers...
reg [3:0] dig1;
reg [3:0] dig2;
reg [3:0] dig3;
reg [3:0] dig4;
reg [3:0] dig5;
reg [3:0] dig6;
reg [3:0] dig7;

reg [3:0] dig2_0;
reg [3:0] dig2_1;
reg [3:0] dig2_2;
reg [3:0] dig2_3;
reg [3:0] dig2_4;
reg [3:0] dig2_5;
reg [3:0] dig2_6;
reg [3:0] dig2_7;





parameter _zer = 4'b0000;                 //some parameters for the
numbers
parameter _one = 4'b0001;
parameter _two = 4'b0010;
parameter _thr = 4'b0011;
parameter _fou = 4'b0100;
parameter _fiv = 4'b0101;
parameter _six = 4'b0110;
parameter _sev = 4'b0111;
parameter _eig = 4'b1000;
parameter _nin = 4'b1001;


parameter col1 = 4'b0111;                 //parameters for the polling
parameter col2 = 4'b1011;                 //circuit
parameter col3 = 4'b1101;
parameter col4 = 4'b1110;

parameter line1 = 2'b00;                  //parameters for mapping from
parameter line2 = 2'b01;                  //the polling circuit
parameter line3 = 2'b10;
parameter line4 = 2'b11;



always @(posedge trigger or posedge reset)      //at trigger or
reset...

      if(reset) begin                     //if reset...

            cycle = col1;                 //set variables to defaults

            num[3:0] = 4'b0000;
            hit = 0;
            miss = 0;

            plus = 0;
            minus = 0;
            equals = 0;
            dig0 = _zer;
            dig1 = _zer;
            dig2 = _zer;
            dig3 = _zer;
            dig4 = _zer;
            dig5 = _zer;
            dig6 = _zer;
            dig7 = _zer;

            dig2_0 = _zer;
            dig2_1 = _zer;
            dig2_2 = _zer;
            dig2_3 = _zer;
            dig2_4 = _zer;
            dig2_5 = _zer;
            dig2_6 = _zer;
            dig2_7 = _zer;

      end

      //if the polling circuit senses an entry we enter the  debouncing
      //and decoding logic

      else if(poll != 4'b1111) begin

            if(hit == 1) begin            //if second straight
occurence...

                  //find the inputted column and row in case statements

                  case(cycle)
                  7:          num[3:2] = line1;
                  11:         num[3:2] = line2;
                  13:         num[3:2] = line3;
                  14:         num[3:2] = line4;
                  endcase

                  case(poll)
                  7:          num[1:0] = line4;
                  11:         num[1:0] = line1;
                  13:         num[1:0] = line3;
                  14:         num[1:0] = line2;
                  endcase

                  //if it was a number input...

                  if(num < 11 && num != 3) begin

                        if(plus || minus) begin  //if on 2nd number

                              dig2_7 = dig2_6; //shift all numbers over
                              dig2_6 = dig2_5;
                              dig2_5 = dig2_4;
                              dig2_4 = dig2_3;
                              dig2_3 = dig2_2;
                              dig2_2 = dig2_1;
            dig2_1 = dig2_0;

            //and map new number

            case(num)

                  0:          dig2_0 = _one;
                  1:          dig2_0 = _fou;
                  2:          dig2_0 = _sev;
                  4:          dig2_0 = _two;
                  5:          dig2_0 = _fiv;
                  6:          dig2_0 = _eig;
                  7:          dig2_0 = _zer;
                  8:          dig2_0 = _thr;
                  9:          dig2_0 = _six;
                  10:         dig2_0 = _nin;

            endcase

      end

      else begin //if on first number

            dig7 = dig6;      //shift numbers over
            dig6 = dig5;
            dig5 = dig4;
            dig4 = dig3;
            dig3 = dig2;
            dig2 = dig1;
            dig1 = dig0;

            //and map new number

            case(num)

                  0:          dig0 = _one;
                  1:          dig0 = _fou;
                  2:          dig0 = _sev;
                  4:          dig0 = _two;
                  5:          dig0 = _fiv;
                  6:          dig0 = _eig;
                  7:          dig0 = _zer;
                  8:          dig0 = _thr;
                  9:          dig0 = _six;
                  10:         dig0 = _nin;

            endcase

      end

      hit = 2;    //increment hit so that we only
                  //sense one input

end

else if (num == 11) begin     //logic for clear
```

```
      cycle = col1;           //set all variables
                        //to default values
      dig0 = _zer;
      dig1 = _zer;
      dig2 = _zer;
      dig3 = _zer;
      dig4 = _zer;
      dig5 = _zer;
      dig6 = _zer;
      dig7 = _zer;

      dig2_0 = _zer;
      dig2_1 = _zer;
      dig2_2 = _zer;
      dig2_3 = _zer;
      dig2_4 = _zer;
      dig2_5 = _zer;
      dig2_6 = _zer;
      dig2_7 = _zer;


      hit = 0;
      miss = 0;

      plus = 0;
      minus = 0;
      equals = 0;

end

else if (num == 12) begin     //logic for +

      if(!minus) begin  //if we aren't already
                        //doing subtract...
      plus = 1;         //we do add
      hit = 2;          //and increment hit

      end

end

else if (num == 13) begin     //logic for -

      if(!plus) begin         //if we aren't already
                        //doing an add...
      minus = 1;        //we do subtract
      hit = 2;          //and increment hit

      end

end

else if (num == 15) begin     //logic for =

      if(plus || minus) begin //if we already chose
                        //to add or subtract...
      equals = 1;       //we do equals
```

```
                        hit = 2;          //and increment hit

                        end

                  end





            end
            else if(hit == 0) begin       //if the first occurence of
entry,
                              //we get ready for second occurence
                  hit = 1;          //this is debouncing logic
                  miss = 0;

            end

      end
      else if(miss == 1) begin            //if no entry is seen we keep
                              //cycling and polling
            case(cycle)
            14:         cycle = col1;
            7:          cycle = col2;
            11:         cycle = col3;
            13:         cycle = col4;
            endcase

      end
      else if(miss == 0) begin            //if no entry is seen for
first
                              //consecutive time, we reset
            miss = 1;               //polling variables miss and hit
            hit = 0;

      end

endmodule
```

```
//author: Mark Holland
//program: operator.v
//purpose: to perform either addition or subtraction on two inputted
numbers.

module operator(clk2, trigger, reset, plus, minus, equals, dig0, dig1,
dig2, dig3, dig4, dig5, dig6, dig7, dig2_0, dig2_1, dig2_2, dig2_3,
dig2_4, dig2_5, dig2_6, dig2_7, ans0, ans1, ans2, ans3, ans4, ans5,
ans6, ans7, LED);

input [2:0] clk2;                     //my clock
input trigger;                              //the trigger for operation

input reset;                                //global reset

input plus;                           //variable for adding
input minus;                                //variable for subtracting
input equals;                               //variable for equals

input [3:0] dig0;                     //the first inputted number, in
input [3:0] dig1;                     //binary encoded decimal
input [3:0] dig2;
input [3:0] dig3;
input [3:0] dig4;
input [3:0] dig5;
input [3:0] dig6;
input [3:0] dig7;

input [3:0] dig2_0;                         //the second inputted number,
in
input [3:0] dig2_1;                         //binary encoded decimal
input [3:0] dig2_2;
input [3:0] dig2_3;
input [3:0] dig2_4;
input [3:0] dig2_5;
input [3:0] dig2_6;
input [3:0] dig2_7;

output [4:0] ans0;                          //the answer, in binary
encoded
output [4:0] ans1;                          //decimal
output [4:0] ans2;
output [4:0] ans3;
output [4:0] ans4;
output [4:0] ans5;
output [4:0] ans6;
output [4:0] ans7;

output LED;                           //overflow and negative indicator

reg [4:0] ans0;                             //registers for ans
reg [4:0] ans1;
reg [4:0] ans2;
reg [4:0] ans3;
reg [4:0] ans4;
reg [4:0] ans5;
reg [4:0] ans6;
reg [4:0] ans7;


reg [3:0] num1;                          //other necessary registers
reg [3:0] num2;
reg [7:0] enable;
reg c_in;
reg c_out;
reg [4:0] sum;



//logic for overflow (adding) and negative (subtracting)

assign LED = clk2[2] && clk2[1] && clk2[0] && c_out && equals;

parameter def = 5'b0_0000;               //default for ans



always @(posedge trigger or posedge reset)

      if(reset) begin                    //if reset...

            ans0 = def;            //reset all variables
            ans1 = def;
            ans2 = def;
            ans3 = def;
            ans4 = def;
            ans5 = def;
            ans6 = def;
            ans7 = def;

            num1 = 4'b0000;
            num2 = 4'b0000;
            sum = 5'b0_0000;
            enable = 8'b0000_0000;
            c_in = 0;
            c_out = 0;


      end

      else if(equals) begin              //if equals, perform
operation

      //on each of eight clock cycles in clk2[2:0] I choose a different
decimal
      //place to operate on.

      //choose decimal digit from first number

                  case(clk2[2:0])

                        0:          num1 = dig0;
                        1:          num1 = dig1;
                        2:          num1 = dig2;
                        3:          num1 = dig3;
                        4:          num1 = dig4;
                        5:          num1 = dig5;
                        6:          num1 = dig6;
                        7:          num1 = dig7;

                  endcase

            //choose decimal digit from second number

                  case(clk2[2:0])

                        0:          num2 = dig2_0;
                        1:          num2 = dig2_1;
                        2:          num2 = dig2_2;
                        3:          num2 = dig2_3;
                        4:          num2 = dig2_4;
                        5:          num2 = dig2_5;
                        6:          num2 = dig2_6;
                        7:          num2 = dig2_7;

                  endcase

            //set the one hot encoded enable line for controlling the answer
      register

                  case(clk2[2:0])

                        0:          enable = 8'b0000_0001;
                        1:          enable = 8'b0000_0010;
                        2:          enable = 8'b0000_0100;
                        3:          enable = 8'b0000_1000;
                        4:          enable = 8'b0001_0000;
                        5:          enable = 8'b0010_0000;
                        6:          enable = 8'b0100_0000;
                        7:          enable = 8'b1000_0000;

                  endcase

            //If working on first digit there is no carry in

                  case(enable[0])

                        0:          c_in = c_out;
                        1:          c_in = 0;

                  endcase




                  if(plus) begin                      //if doing plus...

                        sum = num1 + num2 + c_in;     //add the numbers along
                        c_out = 0;              //with c_in, set c_out
```

```
			if(sum > 9) begin	//if we have carry...

				sum = sum - 10;		//subtract 10 from sum
				c_out = 1;	//set c_out = 1

			end

		end

		else if(minus) begin			//if doing minus...

			sum = num1 - num2 + 10 - c_in;		//add by assuming
a borrow
			c_out = 1;

			if(sum > 9) begin	//if don't need to borrow

				sum = sum - 10;		//make necessary
corrections
				c_out = 0;

			end

		end

	//These if statements map the answers to the proper output
register (digit)

			if(enable[0])

				ans0 = sum;

			if(enable[1])

				ans1 = sum;

			if(enable[2])

				ans2 = sum;

			if(enable[3])

				ans3 = sum;

			if(enable[4])

				ans4 = sum;

			if(enable[5])

				ans5 = sum;

			if(enable[6])

				ans6 = sum;

			if(enable[7])
```

```
                                        ans7 = sum;

                  end

            endmodule
```

```
//author: Mark Holland
//program: mapper.v
//purpose: to map numbers to display segments for the display

module mapper(clk2, trigger, reset, dig0, dig1, dig2, dig3, dig4, dig5,
dig6, dig7, dig2_0, dig2_1, dig2_2, dig2_3, dig2_4, dig2_5, dig2_6,
dig2_7, ans0, ans1, ans2, ans3, ans4, ans5, ans6, ans7, key0, key1,
key2, key3, key4, key5, key6, key7, plus, minus, equals);

input [2:0] clk2;              //slow clock
input trigger;                       //trigger clock

input reset;                         //global reset

input [3:0] dig0;              //binary encoded decimal digits for
input [3:0] dig1;              //first number
input [3:0] dig2;
input [3:0] dig3;
input [3:0] dig4;
input [3:0] dig5;
input [3:0] dig6;
input [3:0] dig7;

input [3:0] dig2_0;                  //binary encoded decimal digits for
input [3:0] dig2_1;                  //second number
input [3:0] dig2_2;
input [3:0] dig2_3;
input [3:0] dig2_4;
input [3:0] dig2_5;
input [3:0] dig2_6;
input [3:0] dig2_7;

input [4:0] ans0;              //binary encoded decimal digits for
input [4:0] ans1;              //answer
input [4:0] ans2;
input [4:0] ans3;
input [4:0] ans4;
input [4:0] ans5;
input [4:0] ans6;
input [4:0] ans7;

output [6:0] key0;                   //mappings for what I am currently
output [6:0] key1;                   //displaying
output [6:0] key2;
output [6:0] key3;
output [6:0] key4;
output [6:0] key5;
output [6:0] key6;
output [6:0] key7;

input plus;                    //operators, + - =
input minus;
input equals;

reg [6:0] key;                       //register for key currently being
                               //mapped
reg [6:0] key0;
reg [6:0] key1;
reg [6:0] key2;
reg [6:0] key3;
reg [6:0] key4;
reg [6:0] key5;
reg [6:0] key6;
reg [6:0] key7;

reg [3:0] temp0;             //register for holding the numbers
reg [3:0] temp1;             //to map
reg [3:0] temp2;
reg [3:0] temp3;
reg [3:0] temp4;
reg [3:0] temp5;
reg [3:0] temp6;
reg [3:0] temp7;

reg [3:0] hold;                    //register for holding specific
temp var
reg [7:0] enable;            //enable contols the writing to key0-7



parameter def  = 7'b111_1111;      //   the mappings for each digit,
as
parameter _one = 7'b100_1111;      //   they are sent to the display
parameter _two = 7'b001_0010;
parameter _thr = 7'b000_0110;
parameter _fou = 7'b100_1100;
parameter _fiv = 7'b010_0100;
parameter _six = 7'b010_0000;
parameter _sev = 7'b000_1111;
parameter _eig = 7'b000_0000;
parameter _nin = 7'b000_1100;
parameter _zer = 7'b000_0001;


always @(posedge trigger or posedge reset)      //on trigger or
reset...

      if(reset) begin                     //if reset...

            temp0 = 4'b0000;        //set all variables
            temp1 = 4'b0000;        //to default
            temp2 = 4'b0000;
            temp3 = 4'b0000;
            temp4 = 4'b0000;
            temp5 = 4'b0000;
            temp6 = 4'b0000;
            temp7 = 4'b0000;

            key = def;

            key0 = def;
            key1 = def;
            key2 = def;
            key3 = def;
            key4 = def;
            key5 = def;
            key6 = def;
            key7 = def;

            hold = 4'b0000;
            enable = 8'b0000_0000;

      end

      else begin

      if (equals) begin             //if operation is complete...

            temp0 = ans0;                 //map answer to temp
            temp1 = ans1;
            temp2 = ans2;
            temp3 = ans3;
            temp4 = ans4;
            temp5 = ans5;
            temp6 = ans6;
            temp7 = ans7;

      end

      else if (plus || minus) begin //if on second number

            temp0 = dig2_0;         //map second number to temp
            temp1 = dig2_1;
            temp2 = dig2_2;
            temp3 = dig2_3;
            temp4 = dig2_4;
            temp5 = dig2_5;
            temp6 = dig2_6;
            temp7 = dig2_7;

      end

      else begin                    //if on first number

            temp0 = dig0;                 //map first number to temp
            temp1 = dig1;
            temp2 = dig2;
            temp3 = dig3;
            temp4 = dig4;
            temp5 = dig5;
            temp6 = dig6;
            temp7 = dig7;

      end

      case(clk2[2:0])

            0:          hold = temp0;     //depending on clock, will
            1:          hold = temp1;     //operate on specific digit,
so
            2:          hold = temp2;     //put proper digit in hold
            3:          hold = temp3;
            4:          hold = temp4;
            5:          hold = temp5;
            6:          hold = temp6;
            7:          hold = temp7;

      endcase

      case(clk2[2:0])

            0:          enable = 8'b0000_0001;  //set enable for
mapping
            1:          enable = 8'b0000_0010;  //the output according
to
            2:          enable = 8'b0000_0100;  //the input
            3:          enable = 8'b0000_1000;
            4:          enable = 8'b0001_0000;
            5:          enable = 8'b0010_0000;
            6:          enable = 8'b0100_0000;
            7:          enable = 8'b1000_0000;

      endcase

      case(hold)

            0:          key = _zer;       //map the number to the
            1:          key = _one;       //display logic
            2:          key = _two;
            3:          key = _thr;
            4:          key = _fou;
            5:          key = _fiv;
            6:          key = _six;
            7:          key = _sev;
            8:          key = _eig;
            9:          key = _nin;

      endcase

      //map the display logic to the proper display variable so that it
puts
      //the number in the right digit place

      if(enable[0])

            key0 = key;

      if(enable[1])

            key1 = key;

      if(enable[2])

            key2 = key;

      if(enable[3])

            key3 = key;
```

```
        if(enable[4])
                key4 = key;
        if(enable[5])
                key5 = key;
        if(enable[6])
                key6 = key;
        if(enable[7])
                key7 = key;
        end
endmodule
```

```
//author: Mark Holland
//program: display.v
//purpose: to send the display information/mappings to the display  LEDs

module display(clk2, trigger, reset, key0, key1, key2, key3, key4,
key5, key6, key7, digit);

input [2:0] clk2;              //clock for multiplexing
input trigger;                       //trigger for operation (clock)
input reset;                         //global reset
input [6:0] key0;              //the display mappings, for 8 digits
input [6:0] key1;
input [6:0] key2;
input [6:0] key3;
input [6:0] key4;
input [6:0] key5;
input [6:0] key6;
input [6:0] key7;
output [6:0] digit;                  //the current outputted display
mapping,
                               //to proper display digit

reg [6:0] digit;

parameter def = 7'b011_0110;         //display mapping if in reset



always @(posedge trigger or posedge reset)      //at trigger or
reset...

      if(reset)                      //if reset...

            digit = def;                   //set variable to defaults

      else                           //otherwise...

      case(clk2[2:0])
                  3'b000:     digit = key0;     //assign the
                  3'b001:           digit = key1;     //mapping of
                  3'b010:           digit = key2;     //segments to
                  3'b011:           digit = key3;     //the newly
                  3'b100:           digit = key4;     //pressed button
                  3'b101:           digit = key5;
                  3'b110:           digit = key6;
                  3'b111:           digit = key7;
      endcase

endmodule
```

```
//author: Mark Holland
//program: display2.v
//purpose: to turn the proper digit on in the multiplexed display

module display2(clk2, trigger, reset, high);

input [2:0] clk2;              //clock for multiplexing
input trigger;                       //trigger for operation (clock)
input reset;                         //global reset
output [7:0] high;                   //for pulling a digit high


reg [7:0] high;




always @(posedge trigger or posedge reset)      //on trigger or
reset...

      if(reset)                      //if reset...

            high = 8'b1111_1110;           //set to default

      else                           //otherwise...

      case(clk2[2:0])

            3'b000:           high = 8'b1111_1110;    //pull the
correct
            3'b001:           high = 8'b1111_1101;    //digit high (0
here
            3'b010:           high = 8'b1111_1011;    //for high)
            3'b011:           high = 8'b1111_0111;
            3'b100:           high = 8'b1110_1111;
            3'b101:           high = 8'b1101_1111;
            3'b110:           high = 8'b1011_1111;
            3'b111:           high = 8'b0111_1111;
            default:    high = 8'b1111_1110;

      endcase


endmodule
```

Appendix B

This Appendix includes the Verilog file for the multiplier that did not get included into my design.

```
//author: Mark Holland
//program: operator.v
//purpose: to perform multiplication on two inputted numbers.

module operator(clk3, trigger, reset, equals, dig0, dig1, dig2, dig3,
dig4, dig5, dig6, dig7, dig2_0, dig2_1, dig2_2, dig2_3, dig2_4, dig2_5,
dig2_6, dig2_7, ans0, ans1, ans2, ans3, ans4, ans5, ans6, ans7, LED);

input [7:0] clk3;                    //my huge clock
input trigger;                             //the trigger for operation

input reset;                               //global reset

input equals;                              //variable for equals

input [3:0] dig0;                    //the first inputted number, in
input [3:0] dig1;                    //binary encoded decimal
input [3:0] dig2;
input [3:0] dig3;
input [3:0] dig4;
input [3:0] dig5;
input [3:0] dig6;
input [3:0] dig7;

input [3:0] dig2_0;                        //the second inputted number,
in
input [3:0] dig2_1;                        //binary encoded decimal
input [3:0] dig2_2;
input [3:0] dig2_3;
input [3:0] dig2_4;
input [3:0] dig2_5;
input [3:0] dig2_6;
input [3:0] dig2_7;

output [3:0] ans0;                         //the answer, in binary
encoded
output [3:0] ans1;                         //decimal
output [3:0] ans2;
output [3:0] ans3;
output [3:0] ans4;
output [3:0] ans5;
output [3:0] ans6;
output [3:0] ans7;

output LED;                          //overflow indicator

reg [3:0] ans0;                            //registers for ans
reg [3:0] ans1;
reg [3:0] ans2;
reg [3:0] ans3;
reg [3:0] ans4;
reg [3:0] ans5;
reg [3:0] ans6;
reg [3:0] ans7;

reg LED;
reg [3:0] temp_ans;                         //the temporary register that
                                    //maps to reg0 - reg7
reg [63:0] answer_binary;                   //the total answer in binary

reg [31:0] mult;                    //the multiplier (powers of 10)
                                    //in binary
reg [31:0] bin_num1;                        //the first inputted number
and
reg [63:0] bin_num2;                        //second inputted number in
binary

reg [31:0] sum;                             //sum maps to bin_num vars

reg [3:0] num1;                             //the current binary encoded
                                    //decimal I am working on



//parameters for all of the multipliers

parameter one = 32'b0000_0000_0000_0000_0000_0000_0000_0001;
parameter ten = 32'b0000_0000_0000_0000_0000_0000_0000_1010;
parameter one_hundred = 32'b0000_0000_0000_0000_0000_0000_0110_0100;
parameter one_thousand = 32'b0000_0000_0000_0000_0000_0011_1110_1000;
parameter ten_thousand = 32'b0000_0000_0000_0000_0010_0111_0001_0000;
parameter one_hundred_thousand =
32'b0000_0000_0000_0001_1000_0110_1010_0000;
parameter one_million = 32'b0000_0000_0000_1111_0100_0010_0100_0000;
parameter ten_million = 32'b0000_0000_1001_1000_1001_0110_1000_0000;

//parameters for the default values

parameter ans_def = 4'b0000;
parameter sum_def = 32'b0000_0000_0000_0000_0000_0000_0000_0000;
parameter big_def =
64'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_00
00_0000_0000;


//I input two 8-digit numbers where each digit is in binary encoded
decimal.  I then
//turn the two numbers into decimal numbers, multiply them by each
other, and map
//the result back into binary encoded decimal digits.  I reuse the same
hardware
//as often as possible which is why the clock is 8 bits wide.  A
mulitiplication
//would take approximately half a second to do.

always @(posedge trigger or posedge reset) //on trigger or reset

      if(reset) begin                       //if reset...

            ans0 = ans_def;                 //reset all variables
            ans1 = ans_def;
            ans2 = ans_def;
            ans3 = ans_def;
            ans4 = ans_def;
            ans5 = ans_def;
            ans6 = ans_def;
            ans7 = ans_def;

            temp_ans = ans_def;
            num1 = ans_def;
            sum = sum_def;
            bin_num1 = sum_def;
            bin_num2 = big_def;
            mult = sum_def;
            answer_binary = big_def;

            LED = 0;

      end

      else if(equals) begin               //if equals, perform
operation

            //during the first 32 clock cycles I map each binary
encoded
            //decimal digit to a straight binary number and keep a
running
            //sum of the complete number.
            //after every 8 cycles I complete one of the two digits

            if(!clk3[5] && !clk3[6] && !clk3[7]) begin

                  case(clk3[3:0])         //choose which digit

                        0:          num1 = dig0;
                        1:          num1 = dig1;
                        2:          num1 = dig2;
                        3:          num1 = dig3;
                        4:          num1 = dig4;
                        5:          num1 = dig5;
                        6:          num1 = dig6;
                        7:          num1 = dig7;
                        8:          num1 = dig2_0;
                        9:          num1 = dig2_1;
                        10:         num1 = dig2_2;
                        11:         num1 = dig2_3;
                        12:         num1 = dig2_4;
                        13:         num1 = dig2_5;
                        14:         num1 = dig2_6;
                        15:         num1 = dig2_7;

                  endcase


                  case(clk3[2:0])         //choose the proper
multiplier

                        0:          mult = one;
                        1:          mult = ten;
                        2:          mult = one_hundred;
                        3:          mult = one_thousand;
                        4:          mult = ten_thousand;
                        5:          mult = one_hundred_thousand;
                        6:          mult = one_million;
                        7:          mult = ten_million;

                  endcase


                  //map the digit to straight binary, add to running
sum

                  if(num1[0])

                        sum = sum + mult;

                  num1 = num1 >> 1;
                  mult = mult << 1;

                  if(num1[0])

                        sum = sum + mult;

                  num1 = num1 >> 1;
                  mult = mult << 1;

                  if(num1[0])

                        sum = sum + mult;

                  num1 = num1 >> 1;
                  mult = mult << 1;

                  if(num1[0])

                        sum = sum + mult;

                  num1 = num1 >> 1;
                  mult = mult << 1;


                  //after 8 cycles I store the sum as a completed
number

                  if(clk3[0] && clk3[1] && clk3[2]) begin

                        if(clk3[3])

                              bin_num1 = sum;

                        else

                              bin_num2 = sum;

                  end

            end
            //on the next 32 cycles I multiply the two binary numbers
            //together, giving me the result of the multiplication


            else if(clk3[5] && !clk3[6] && !clk3[7]) begin

                  if(bin_num1[0])

                        answer_binary = answer_binary + bin_num2;

                  bin_num1 = {0, bin_num1[31:1]};
                  bin_num2 = {bin_num2[30:0], 0};

            end

            //I use 128 clock cycles to map the binary answer back into
            //binary encoded decimal digits.  In all, I use 256 clock
cycles
            //of trigger, which takes about .5 seconds

            else if(clk3[7]) begin

                  case(clk3[6:4])         //choose proper multiplier

                        0:          mult = one;
                        1:          mult = ten;
                        2:          mult = one_hundred;
                        3:          mult = one_thousand;
                        4:          mult = ten_thousand;
                        5:          mult = one_hundred_thousand;
                        6:          mult = one_million;
                        7:          mult = ten_million;

                  endcase

                  //if I can pull out a multiplier I do so and
                  //increment the proper binary encoded decimal digit

                  if(answer_binary > mult) begin

                        answer_binary = answer_binary - mult;
                        temp_ans = temp_ans + 1;

                        if(temp_ans > 9)

                              LED = 1;

                  end

                  //At the proper times I map the digits back to the
output
                  //registers, ans0 - ans7

                  if(!clk3[6] && !clk3[5] && !clk3[4] && clk3[3] &&
clk3[2] && clk3[1] && clk3[0])
```

```
                        ans0 = temp_ans;

                  if(!clk3[6] && !clk3[5] && clk3[4] && clk3[3] &&
clk3[2] && clk3[1] && clk3[0])

                        ans1 = temp_ans;

                  if(!clk3[6] && clk3[5] && !clk3[4] && clk3[3] &&
clk3[2] && clk3[1] && clk3[0])

                        ans2 = temp_ans;

                  if(!clk3[6] && clk3[5] && clk3[4] && clk3[3] &&
clk3[2] && clk3[1] && clk3[0])

                        ans3 = temp_ans;

                  if(clk3[6] && !clk3[5] && !clk3[4] && clk3[3] &&
clk3[2] && clk3[1] && clk3[0])

                        ans4 = temp_ans;

                  if(clk3[6] && !clk3[5] && clk3[4] && clk3[3] &&
clk3[2] && clk3[1] && clk3[0])

                        ans5 = temp_ans;

                  if(clk3[6] && clk3[5] && !clk3[4] && clk3[3] &&
clk3[2] && clk3[1] && clk3[0])

                        ans6 = temp_ans;

                  if(clk3[6] && clk3[5] && clk3[4] && clk3[3] &&
clk3[2] && clk3[1] && clk3[0])

                        ans7 = temp_ans;

            end

      end


endmodule
```