UCD School of Electrical and Electronic Engineering

EEEN30190 Digital System Design

Calculator Design

Name: Jack Coleman **Student Number: 21207103**

Name: Ruairi Hogan **Student Number: 21432816**

Lab Session: Thu 10 **Brightspace Group: CR 5**

By submitting this report through Brightspace, we certify that ALL of the following are true:

1. We have read the *Student Guide to the UCD Student Plagiarism Policy*. (The full policy document is available on Brightspace). We understand the definition of plagiarism and the consequences of plagiarism.
2. We recognise that any work that has been plagiarised (in whole or in part) may be subject to penalties, as outlined in the documents above. That includes work that is the result of collaboration, where that is not acknowledged.
3. We have not plagiarised any part of this report. The work described was done by the people named above, and this report is all our own original work, except where otherwise acknowledged in the report.

Calculator Functionality (Ruairi Hogan)

Our calculator is a 5-digit hexadecimal calculator with 2 main operators, addition and multiplication. Two 5-digit hexadecimal numbers can be summed or multiplied together and, as long as the resulting answer is less than “FFFFF”, the calculator will display the correct calculation after the equal key is pressed. At this point, the user can choose to perform a new calculation, using the answer to the previous as their first value. If the user wants, they can also chain operations together without pressing the equal key; For example, “4*5+2=” will display the correct value of 16. Further, when the “+” button is pressed in that sequence, the correct calculation “14” (4*5=14) will be displayed to the user.

It is important to note that this calculator does not follow Order of Operations- on this calculator, “2+4*5” will display 1E (6*5), when a scientific calculator might display 16 (2+14).

The calculator offers an assortment of additional quality of life features to make this interaction more seamless. This include:

- The display will continue to display the first number entered after the operator of choice is selected, and only change when the first digit of the new number is selected.
- If the user selects the wrong operator by mistake, they can change their selection to the other without having to re-start the calculation. (“6*+2=” will display 8)

- If someone has already entered 5 digits into the display and attempts to enter a 6th, no change will happen, and the entered number will not change. (an input of “123456” will display “12345”, not “23456”)
- If the user mis-enters the current number, they can press the *CE* button, which will clear the current entry but keep the rest of the calculation stored.
- If the user mis-enters the entire calculation, they can press the *CA* button, and re-start the calculation.
- If the user wants to square the current number, they can press the x^2 button and the currently displayed number will immediately be squared.
- If the Equal button is pressed multiple times in a row, the display will remain the same.
- If the user wants to store a number for later use, they can press the *MS* button, which will immediately store the currently displayed number. Pressing *MR* will immediately recall this number to the display to be used. Neither *CA* nor *CE* will clear the stored value- only a full reset will.

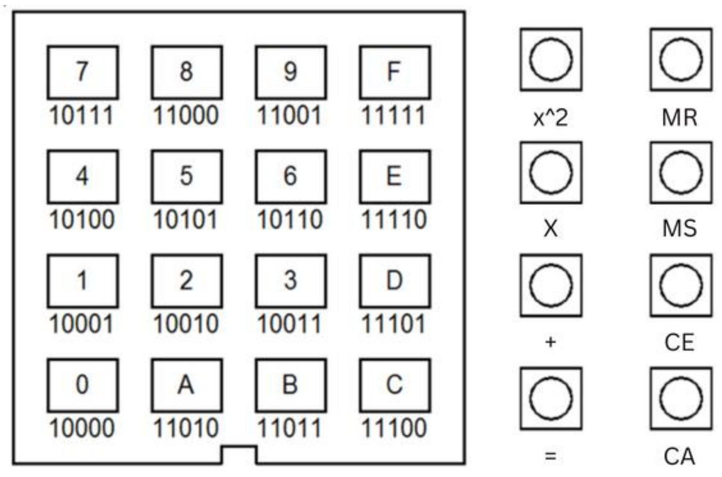
As this calculator is limited to only a 5-digit display, it has a built in Overflow alert system. If an operation is attempted in which the result cannot be shown on the display, 5 dots are turned on. These dots will not disappear until the *CA* button is pressed, in which they will immediately dis-appear.

Key Assignment

Our design uses the 16 hexadecimal keys for their obvious hexadecimal equivalents, and each of the 8 function keys for a unique purpose. We initially intended for the left column of the function keys to be “operator” keys (equal, multiply, add, etc.). This would allow us to quickly detect if an operator key was pressed by checking if the first two bits of the key were “01”. However, we eventually decided to have 3 “operator” keys and 5 “special” keys (x^2, MR, MS, CE, CA), and so “operator” and “special” detection became slightly more complicated, as will be explained later. For the most part though, we were able to distinguish between an “operator” or “special” key with only the first two bits of the keycode. Below is a chart with the keycode corresponding to each function.

KEYCODE	FUNCTION
01001	Square Current Value
01010	Multiply Operator
01011	Add Operator
01100	Equal
00001	Memory Recall
00010	Memory Save
00011	Clear Entry
00100	Clear All

Below is a diagram of each button in relation to where they are on the keypad themselves.

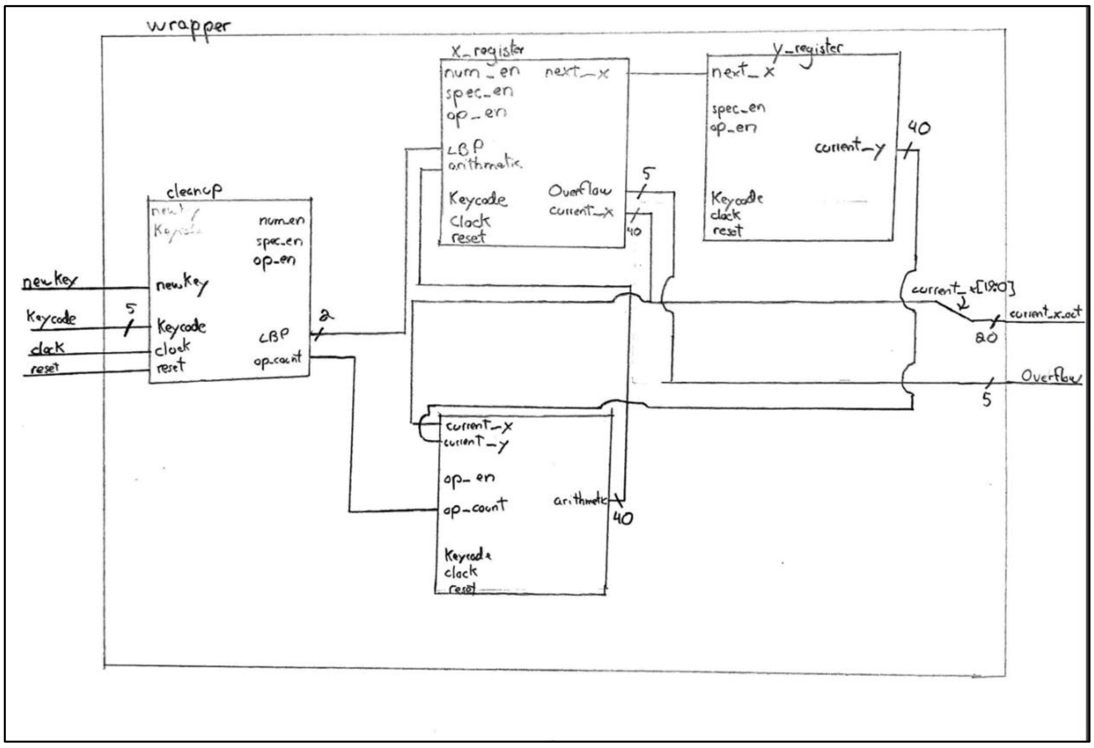


Calculator Hardware (Jack Coleman)

To implement a calculator with these features, we designed 4 unique modules, which were then instantiated into one larger wrapper module, which linked the four modules as needed and contained the initial inputs and final outputs. The four unique modules were:

- Cleanup- a module that took the *newKey* and *Keycode* signals from the "keypad interface" module and produced three enables: one indicating a number was selected, one indicating an operator was selected, and one indicating a special function. The module also contained two registers storing the calculator's "state": one indicating the type of the last button pressed (from now on known as the *LBP* for last button pressed) and one counting the amount of operations in the current calculation, which is useful for correctly chaining operations.
- Operations- a module that took the current values for x and y, the operation enable and Keycode signals, and the operation count value, and outputted an "arithmetic" signal.
- X_register- a module that inputted the three unique enables and Keycode signal, the LBP and operation count signals, and the arithmetic signal, and outputted the current value of x and an overflow signal. This module also outputs the next value for x, which is used solely in determining the value for the y register.
- Y_register- the final module, which took as inputs the operator and special enables, the Keycode signal, and the next value for x, and simply outputs the current value for y.

An RTL of the wrapper module is below, with the more interesting wire connections drawn in; please note any inputs or outputs with the same name should be connected, but the diagram would be effectively unreadable. The wrapper module outputs a 20 bit signal *current_x_out* which is the truncated value of *current_x* and will feed the display signal *digits*. The 5 bit signal *Overflow* will feed the display *dots* signal



The final design contains 7 registers: the LBP, the operation count, the x register, the y register, the operator register, the memory value register, and the Overflow detection register.

Multiple signals in this design used numeric values to represent non-numeric concepts (*Keycode* values representing functions, *LBP* signal representing a button type, the *operation* signal representing a type of operator). For that reason, we often used local parameters that converted the concept to their numeric values. These local parameters will appear on some RTL diagrams and in the Verilog description; below are the most prevalent ones that were used.

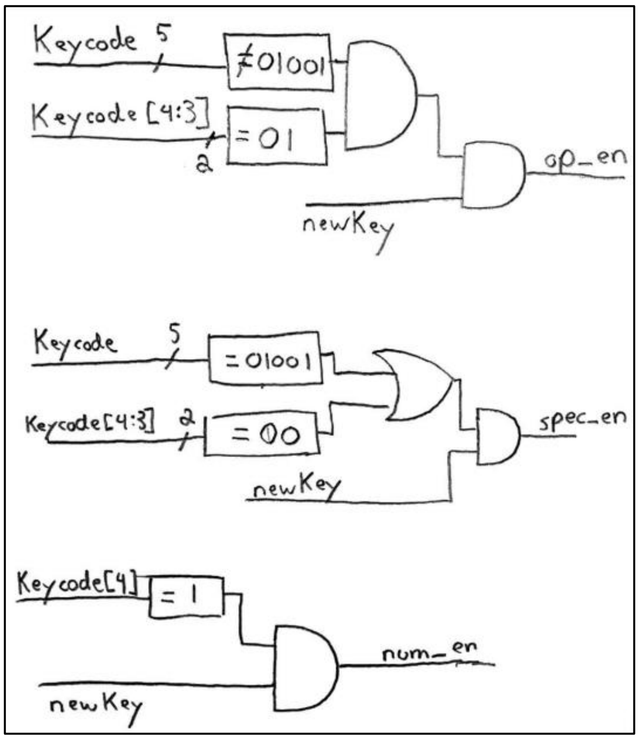
```
//localparameters of buttons
localparam MEMRECALL_KEY=5'b00001;
localparam MEMSAVE_KEY=5'b00010;
localparam EQUAL_KEY=5'b01100;
localparam MULTIPLY_KEY=5'b01010;
localparam ADD_KEY=5'b01011;
localparam CE_KEY=5'b00011;
localparam CA_KEY=5'b00100;
localparam SQUARE_KEY=5'b01001;
```

```
//localparameters of LBP
localparam OPERATOR=2'd1;
localparam NUMBER=2'd2;
localparam EQUAL=2'd3;

//localparamaters of operator
localparam EQUAL_OP=2'b0;
localparam ADD_OP=2'b1;
localparam MULTIPLY_OP=2'd2;
```

Cleanup (Jack Coleman)

During initial drafting stages, in which many additional features were not yet included, this module was quite simple. However, as time went on and more modifications were added, this module specifically became more complex.

The most important aspect of this module are the three enable signals *op_en*, *spec_en*, and *num_en*. All three enables are high only when the *newKey* signal is high, along with other individual requirements. The signal *num_en* also requires the most significant bit of the *Keycode* signal is 1. *op_en* requires the 2 most significant bits of *Keycode* to be “01”, but *Keycode* itself is not “01001”. Finally, *spec_en* requires the 2 most significant bits of *Keycode* to be “00”, or that *Keycode* itself is “01001”. As mentioned before, the mapping of the keys was intentional to only require these enables to check the 2 most significant digits at max. However, the assortment keys we needed meant this could not be done, and key “01001” needs to be checked individually.

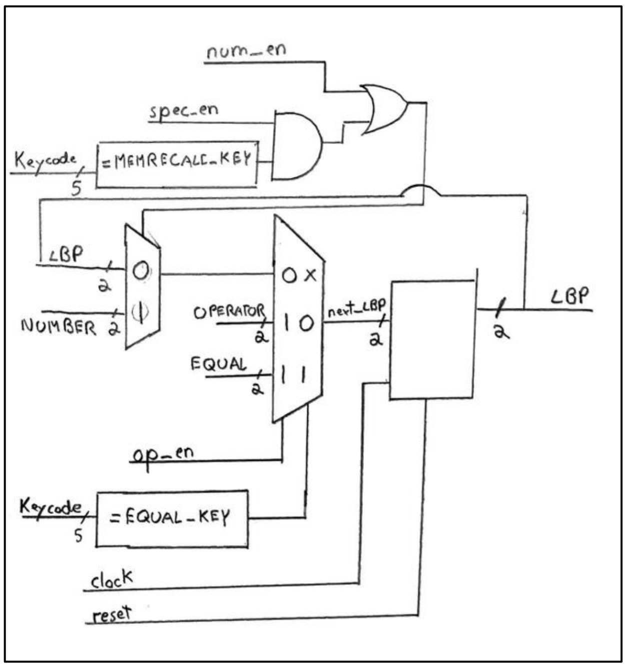


```
assign op_en = newKey & (Keycode[4:3]==2'b01 && Keycode!=5'b01001); //newkey pressed and
button was one of the op buttons
  assign num_en = newKey & (Keycode[4]==1'b1);//newkey pressed and button was a number
  assign spec_en = newKey & (Keycode[4:3]==2'b00 || Keycode==5'b01001); //newkey pressed
and button was one of the spec buttons
```

In the rest of the modules, care has been taken to ensure that 1) no *Keycode* value is used in register value calculation without first checking the enable (for reliabilities sake), but also 2) the enables are only being checked when needed and not redundantly (for efficiency’s sake).

The second and third part of the “cleanup” module, the *LBP* and *op_count* registers, were designed to allow for additional features we wanted to include. We found the LBP useful for a number of reasons, but it was originally designed as a method of delaying the wiping of the X register until the user began entering a new number. (“4 *” should still display the number 4 here, rather than 0.) Instead of wiping the X register whenever an operator was pressed, knowing the LBP allows the calculator to only clear the X register when a number is pressed AFTER an operator was pressed.

The basic premise of the LBP register is that if *op_en* is high and *Keycode* represents the equal key, the LBP register will store 3, which represents an equal. For all other cases of *op_en* being high, the LBP register will store 1, representing an operator. If *num_en* is high, OR *spec_en* is high and *Keycode*=MEMRECAL_KEY, *LBP* will store 2, representing a number. It is important to incorporate the memory recall key here, as, within the concept of "last button pressed" it represents a number being selected and not a special button.

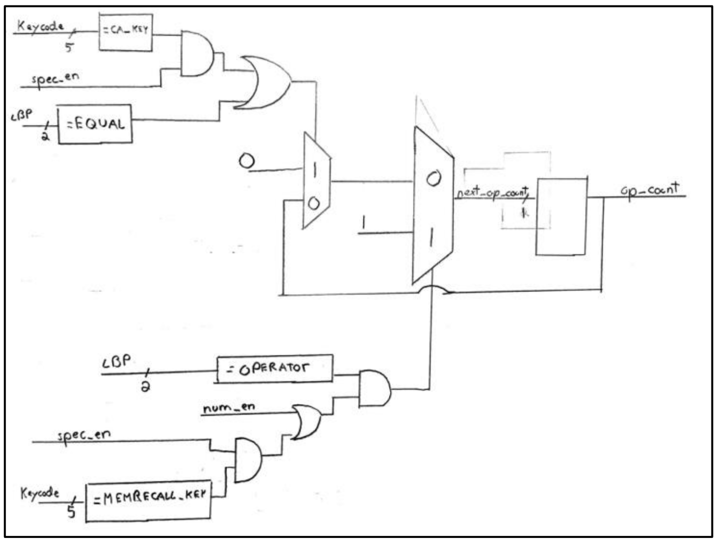


```
//LBP Register
    always@(posedge clock)
    if(reset) LBP<=1'b0;
    else LBP<= next_LBP;

    //Multiplexor for LBP Register
    always@(op_en, num_en, Keycode, LBP, spec_en)
    if((op_en)&& Keycode==EQUAL_KEY) //LBP=Equal if Equal was hit
        next_LBP=EQUAL;
    else if(op_en) //LBP= operator if operator pressed (but not equal)
        next_LBP=OPERATOR;
    else if(num_en||((spec_en)&&(Keycode==MEMRECALL_KEY))) //LBP=number if number
pressed OR memrecall was pressed (techincally a number)
        next_LBP=NUMBER;
    else next_LBP=LBP;
```

The operation count register is only used to correctly implement the chaining operations feature. Take the input sequence "1+1+1=". On the first press of the + key, the display should continue to display "1". However, on the second press of the + key, the display should read "2". The + key has different functionality depending on whether it is the first calculation in a chain or not. The first idea was to, after every calculation, wipe Y to 0, and leave +'s functionality static. This would mean that "1+" would correctly display "1". However, this method would cause "1*" to display 0, which is very wrong. Instead, this design will count if the first operation has occurred yet.

If the LBP was an operator, and *num_en* is high (or *spec_en* and *Keycode* is the memory recall key) the operator count is set to 1. 1 represents the chain being past the first operation (in the "1+1+1" example, 1+1 has been entered). The reason the register waits for *num_en* to be high, rather than acting on *op_en* or *LBP*=OPERATOR is to maintain support of another feature, the ability to change operator selection mid-operation. If a user presses "4*+3=", the display should read "7", the result of "4+3". However, “4*+3” is only one operation even though *op_en* will be high twice; waiting for a number to be pressed after an operator will correctly identify all operations without sending any false positives.

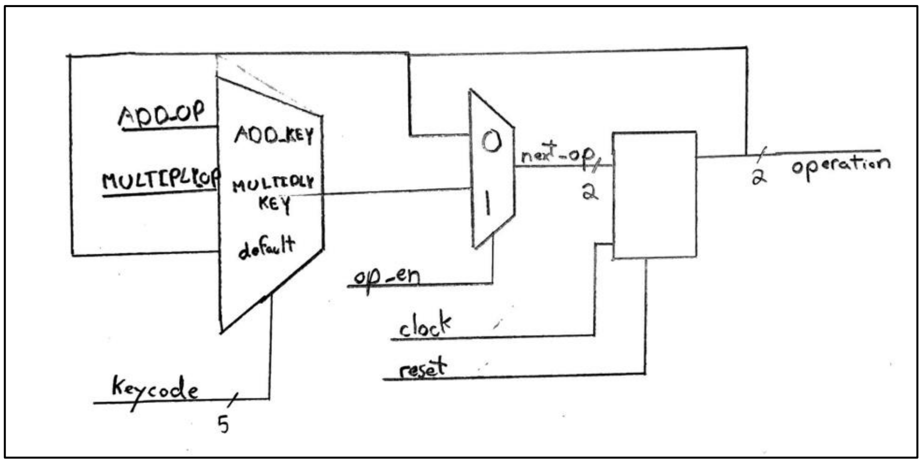
If *LBP*=EQUAL, or *spec_en* is high and *Keycode*=CA_KEY, the operator count is returned to 0. This means a operation chain has just been terminated, and the next operator will be the first in a chain.



```
reg next_op_count;

   //Operation Count (For Operation Chaining) Register
    always@(posedge clock)
    if(reset) op_count<=1'b0;
    else op_count<=next_op_count;

  //multiplexor for Operation Count Register
    always@(LBP, num_en, spec_en, Keycode, op_count)
    if((LBP==OPERATOR)&&((num_en)||(spec_en==1)&&(Keycode==MEMRECALL_KEY)))//if an
operation has just happened this 'iteration', remember
        next_op_count=1'b1;
    else if((LBP==EQUAL) ||((spec_en)&&(Keycode==CA_KEY)))//if CA or equal is hit, its a
new iteration and 0 operations have happened
        next_op_count=1'b0;
    else next_op_count=op_count;
    endmodule
```

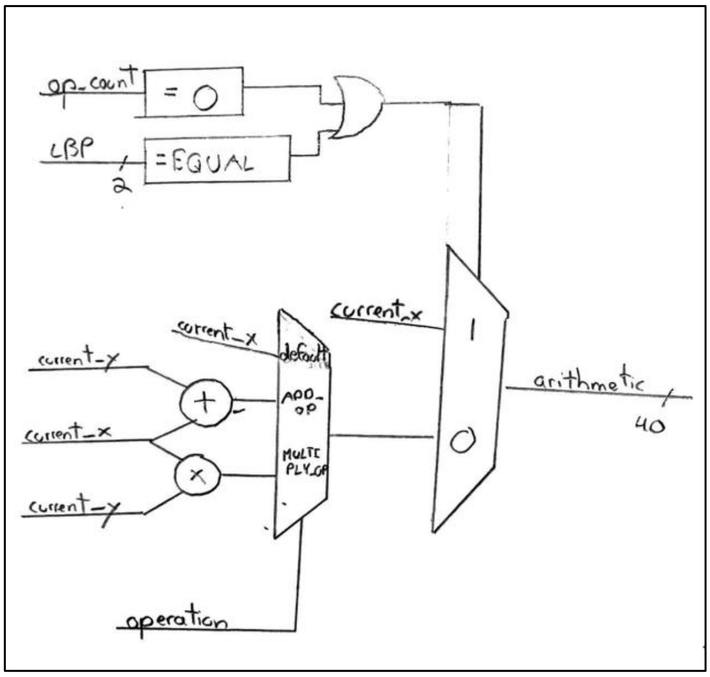
Operations (Ruairi Hogan)

The “operations” module has one main role- it outputs a signal *arithmetic*, which the x register will be assigned whenever *op_en* is high. To do so, a register labelled *operation* remembers the last operator (not including equals) that was hit. Every clock cycle, *operation* is set to the value of *next_op*. If *op_en* is high, *next_op*=ADD_OP if *Keycode*=ADD_KEY, and MULTIPLY_OP if *Keycode*=MULTIPLY_KEY. If *op_en* is high and *Keycode* is neither of those things, *next_op* equals the value of *operation.*



```
reg[1:0] operation;
  reg[1:0] next_op;

  // operator register
  always@(posedge clock)
    if(reset) operation<=1'b0;
  else operation<=next_op;


  //multiplexor for operator register
  always@(op_en, Keycode, operation)
    if(op_en) //if operation enable
        case (Keycode)
        ADD_KEY: next_op= ADD_OP; //store add
        MULTIPLY_KEY: next_op=MULTIPLY_OP; //store multiply
        default: next_op=operation;
        endcase
    else next_op=operation;
```

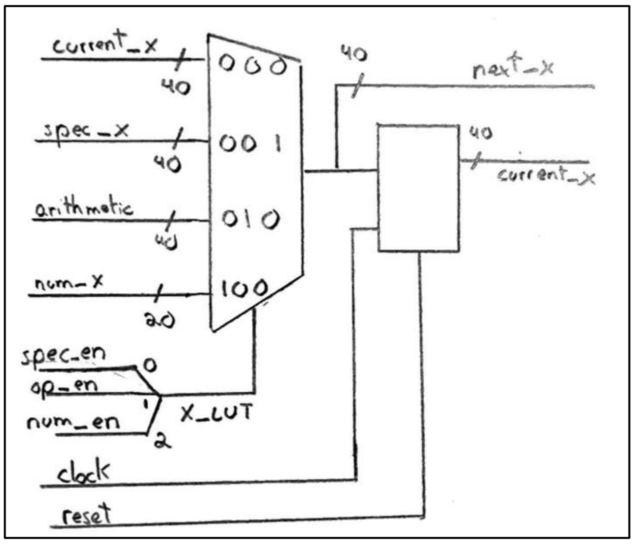
This register is used in the generation of *arithmetic* signal. As the X register is given the value of *arithmetic* every time *op_en* is high, many edge cases must be accounted for in the calculation of this signal. As said before, if the operation chain count is at 0 (“1+”), the display should remain the value of x. Similarly, if the user enters “1+2=+”, the display should not change after the + is selected. Therefore, if the *op_count*=0, or *LBP*=EQUAL, *arithmetic* should be set to the signal *current_x*. Otherwise, *arithmetic* is assigned according to the value of *operation*: *current_x* times *current_y* if *operation*=MULTIPLY_OP, or *current_x* plus *current_y* if *operation*=ADD_OP.



```
always@(operation, current_x, current_y, op_count, LBP, op_en, Keycode)
    if(op_count==1'b0 || LBP==EQUAL) //if on first operation of chain, dont want to
affect x
       arithmetic=current_x;
  else  //else do the operation on x
    case (operation)
        ADD_OP: arithmetic=current_x+current_y;
        MULTIPLY_OP: arithmetic=current_x*current_y;
        default: arithmetic=current_x;
        endcase
endmodule
```

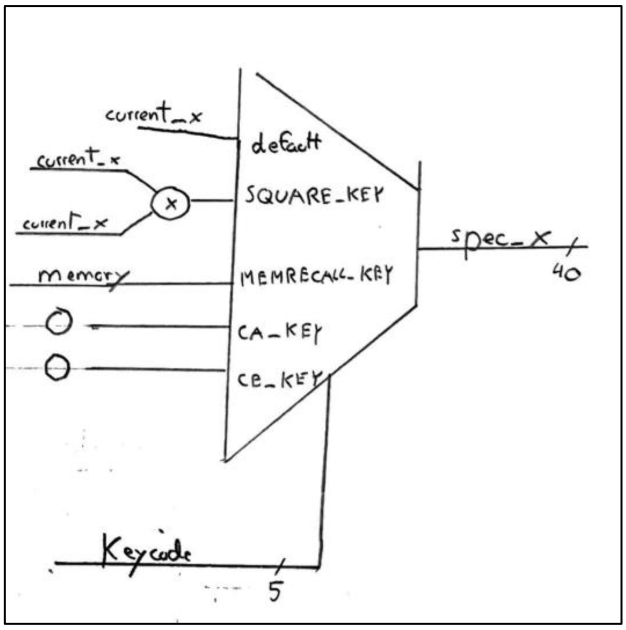
X Register (Jack Coleman)

The “x_register” module is the module that eventually decides what the display reads. The main register of the module, *current_x*, is assigned the value of the signal *next_x* every clock cycle. The signal *next_x* is determined by a multiplexor controlled by the signal *X_LUT*, which is simply a concatenation of {*num_en*, *op_en*, *spec_en*}. If *num_en* is high, *next_x* is assigned the value of signal *num_x*. If *op_en* is high, *next_x* is assigned *arithmetic*, which is calculated in the “operations” module. If *spec_en* is high, *next_x* is assigned the value of *spec_x*. Finally, if no enables are active, *next_x* simply equals *current_x*. This design was one of the first to be created at the onset of the project (without the inclusion of *spec_en* or *spec_x*, of course), and hence is one of the simpler designs, even with the inclusion of new features.



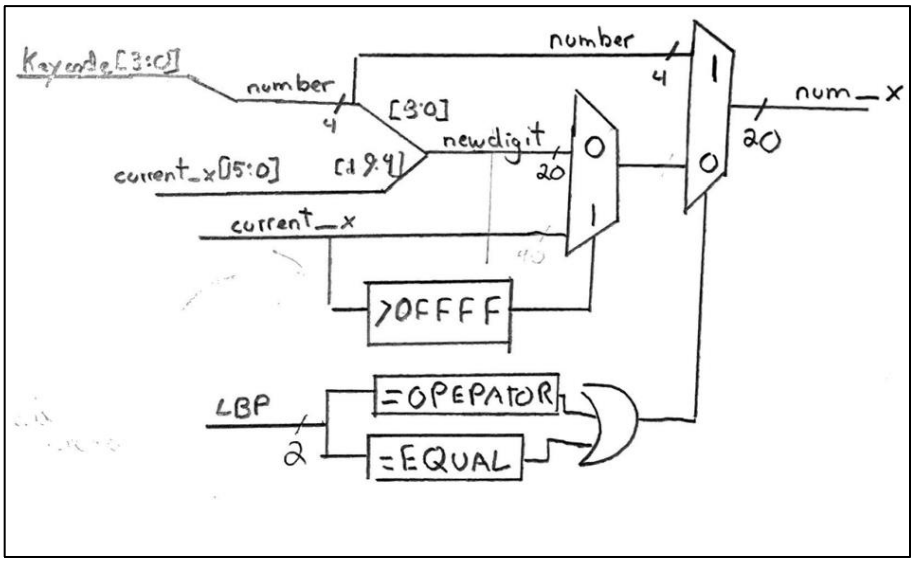
```
// Verilog for register
    always @(posedge clock)
        if(reset)
            current_x <= 1'b0; //on reset, resets counter to 0
        else current_x <= next_x;


  // Creating control signal for multiplexer
  wire [2:0] x_LUT;
  assign x_LUT = {num_en,op_en,spec_en};


  // Multiplexer for choosing next X value
    always @ (x_LUT, current_x, spec_x, arithmetic, num_x)
      case(x_LUT)
       3'b000: next_x= current_x;
       3'b001: next_x=spec_x;
       3'b010: next_x=arithmetic;
       3'b100: next_x=num_x;
       default: next_x=current_x;

endcase
```

The signal *spec_x* is assigned using just a multiplexor controlled by the *Keycode* signal. For each special value of *Keycode* (x^2, MR, CA, CE), *spec_x* is assigned the corresponding function (*current_x* times *current_x*, the signal *memory*, 0, and 0 respectively.)



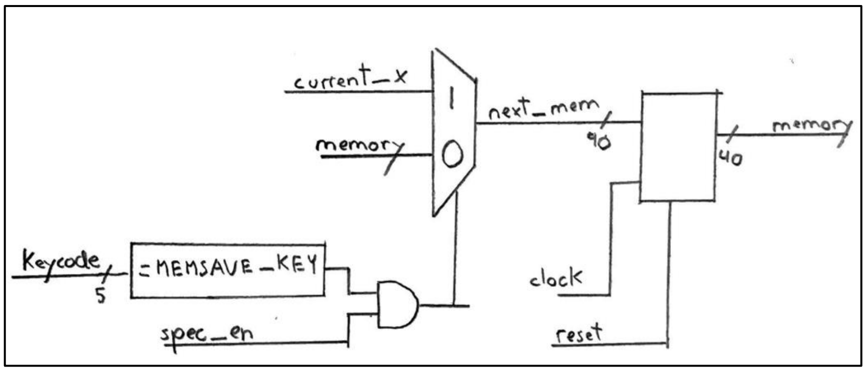
```
reg [39:0] spec_x; //deciding value for spec_x
  always@(Keycode, current_x, memory)
  case(Keycode)
  SQUARE_KEY: spec_x=current_x*current_x; //square key function
  MEMRECALL_KEY: spec_x=memory; //memrecal function
  CA_KEY: spec_x=1'b0; //CA function
  CE_KEY: spec_x=1'b0; //CE function
  default: spec_x=current_x; //anything else
  endcase
```

The *num_x* signal is slightly more complicated. If the *LBP* signal represents an operator or equals, *num_x* will simply become the signal *number*, the first 4 bits of *Keycode*. This is because the user has just started a new number. If *current_x* is currently greater than “0FFFF”, *num_x* is assigned *current_x*. This is to avoid overflow and ensures a rogue 6th digit will not change a 5-digit number. Otherwise, *num_x* is assigned the signal *newdigit*. *newdigit* is calculated by placing the 16 right-most bits of the *current_x* in the left-most 16 bits of *newdigit* and assigning the first 4 with the *number*. This allows the user to enter in multi-digit numbers one digit at a time.



```
assign newDigit = {current_x[15:0],number[3:0]}; //wire representing adding the digit
to current number


  always@(LBP, number, current_x, newDigit)
  if(LBP==OPERATOR || LBP==EQUAL) // if this is start of new number, begin a new
number with this digit
    num_x=number;
  else if(current_x>20'H0FFFF) //if current x is already 5 digits, do nothing
    num_x=current_x;
  else num_x=newDigit; //otherwise add digit to current number
```

The final two parts of the "x_register" module are additional features: the memory register, and the overflow register. Both are relatively simple. The signal *memory* is assigned the value of *next_mem* every clock cycle. If *spec_en* is high and *Keycode*=MEMSAVE_KEY, *next_mem* is set to *current_x*. Otherwise, *next_mem* is set to the value of *memory*.

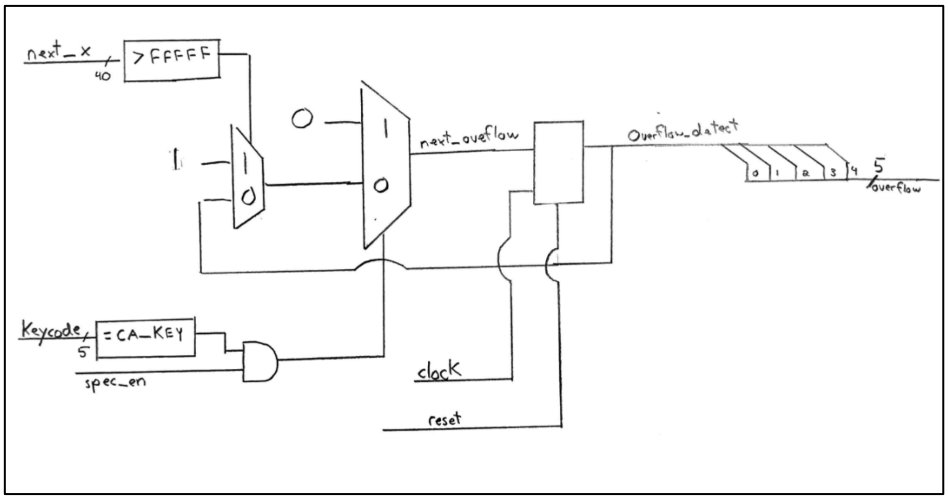


```
//register for memory value
always@(posedge clock)
if(reset) memory<=1'b0;
else memory<=next_mem;

//multiplexor and combinational logic for next_mem
always@(spec_en, Keycode, current_x, memory)
if((spec_en) && (Keycode==MEMSAVE_KEY))//if Memsave key is pressed, and enable high, set
memory to current x
next_mem=current_x;
else next_mem=memory;
```

The register *Overflow_detect* operates off the value of *next_overflow* at every clock cycle. If *next_x* is greater than "FFFFF", *next_overflow* is set to 1. If the CA key is ever pressed, *next_overflow* is set to 0. This creates a flip-flop of sorts- the overflow register is "off" until the value of x goes above 5 digits- in which case the register is forever "on", and only turns off when the CA button is pressed (or the reset button, obviously).

The signal *next_x* is used here for timing purposes. It is safe to use here, as it is still being used on the same clock cycle as in the *current_x* register. We want the Overflow dots to turn on, on the same clock cycle that *current_x* is in an overflow state- using *current_x* would mean the dots turn on 1 clock cycle later.

One may notice that many signals (*next_x, arithmetic, spec_x, current_x, current_y)* were 40 bits long, when the this calculator design can only display 20. This is because 40 bits can contain the largest possible result of the multiplication of two 5-digit numbers (FFFFF*FFFFF= FFFFE00001). Making these key signals 40 bits ensures that the calculator will always catch an overflow operation, no matter the size. The *current_x* signal is truncated to only its rightmost 20 bits in the "wrapper" module.



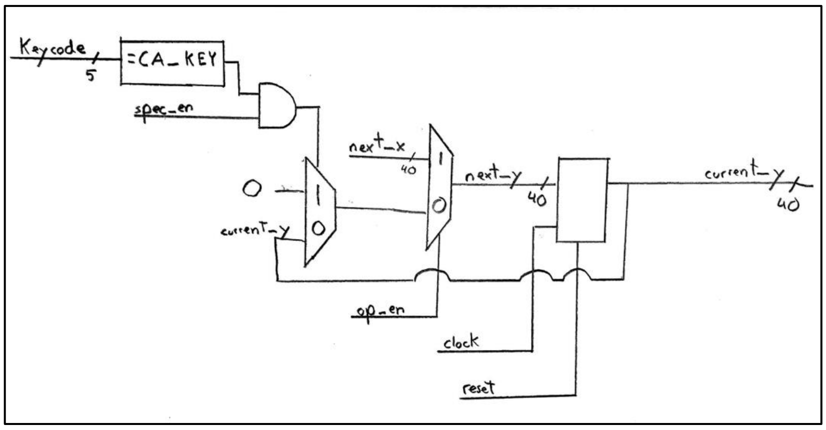
```
//register for overflow detection
  always@(posedge clock)
    if(reset)
      Overflow_detect<=1'b0;
  else Overflow_detect<=next_overflow;


  //multiplexor to decide next_overflow
  always@(next_x, op_en, Keycode, spec_en, Overflow_detect)
    if((spec_en) &&(Keycode==CA_KEY)) //if CA key and CA enable, reset overflow to 0
    next_overflow=1'b0;
  else if (next_x>20'hFFFFF)// if next_x on clock cycle is too big, overflow turns on
      next_overflow=1'b1;
  else next_overflow=Overflow_detect; //otherwise keep the register the same as

 //turns Overflow from one bit wide to 5, for display dot control
  assign Overflow={Overflow_detect, Overflow_detect, Overflow_detect,
Overflow_detect, Overflow_detect};
```

The signal *Overflow* is a signal consisting of 5 iterations of *Overflow_detect*. This is to allow *Overflow* to control the 5 dots on the display with a single bit.

Y Register (Ruairi Hogan0

The final module designed in the "y_register" module. This module is the simplest of the 4, as it consists of just one register with combinational logic. The signal *current_y* is assigned the value of *next_y* every clock cycle. If *op_en* is active (whenever an operator is pressed), *next_y* is assigned the value of *next_x*. If *spec_en* is high and *Keycode*=CA_KEY, *next_y* is set to 0. This represents the CA key wiping the contents of the Y register. Otherwise, *next_y* is simply set as *current_y*.

The signal *next_x* is used here for similar reasons as it is in the Overflow detection hardware. We want, on *op_en* high, *current_x* and *current_y* to be labelled the same thing. During verification, we foud cases where *current_y* was assigned values we did not intend when *current_x* was used instead of *next_x*, due to timing related problems. We believe this design is safe, as *next_x* is still operating on the same clock cycle in both registers. In this case, *next_y* could also be assigned to *arithmetic* instead of *next_x* to implement the same function, but we found this method to better represent our mental thought process of assigning X and Y the same value.



```
//register for storing y value
  always@(posedge clock)
    if(reset) current_y<=1'b0;
  else current_y<=next_y;

 //combinational logic to decide next_y
  always@(op_en, spec_en, Keycode, current_y, next_x)
    if(op_en) //on every operator select, set y to x
      next_y=next_x;
else if((spec_en) && (Keycode==CA_KEY)) //set y to 0 if clear all is pressed
    next_y=1'b0;
  else next_y=current_y;
```

Verification

After implementing our design in Verilog, we recognized that Verification at this point had two unique angles to it- does each module independently do what we want them to, but more importantly, does the combination of each module functioning the way we want them to, correlate to a calculator working. We decided that due to the complex nature of how each module interacted with each other, it would be beneficial to verify the calculator as a whole very thoroughly, and spend less time on each module.

The goal of each module-specific verification phase was to ensure that the modules were working generally as we intended them to. For the most part, this went without a hitch. The purpose of the main system verification, and the one that took the most time, was ensuring that every function worked when the modules came together. We often forgot small inclusions for a feature that rendered the feature completely useless. These are bugs that are very hard to find at a module level, but become obvious at the higher levels of abstraction.

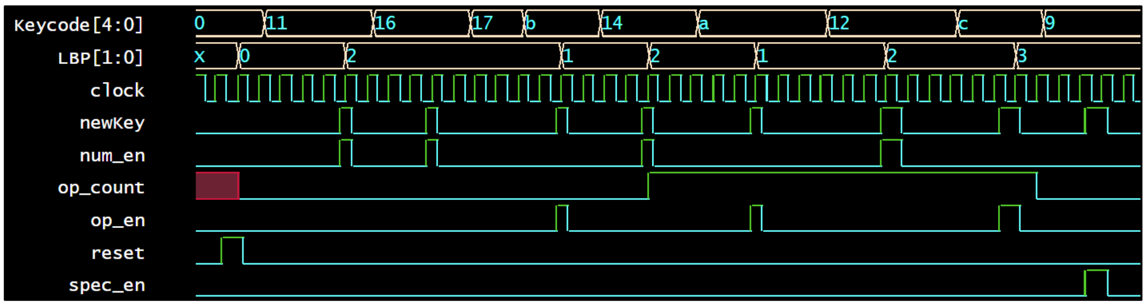
We also attempted a form of black-box verification for the module-specific phases, where we verified the modules that we had the least involvement in. However, due to the abstract nature of each module, and the high level of involvement we each had in all four modules, this was not perfectly effective.

Cleanup (Ruairi Hogan)

Verifying the cleanup module involved checking the 3 most important aspects:

- Do all three enables function properly?
- Does the *LBP* register function properly?
- Does the *op_count* module correctly recognize operations?

Below is an example of a testbench we used to verify this module. This timing diagram represents the input "16+4*2=[x^2]".



If you follow the enables, they are only high when *newKey* is high, and correlate with the number pressed at that point; starting from the beginning, there is 2 *num_en* highs, a *op_en* high, another *num_en* high, another *op_en*, another *num_en*, an *op_en*, and finally a *spec_en*, which matches the input.

The signal *op_count* only becomes active on the start of the number after an operator, as wanted. Similarly, *LBP* correctly identifies what the last button pressed is.

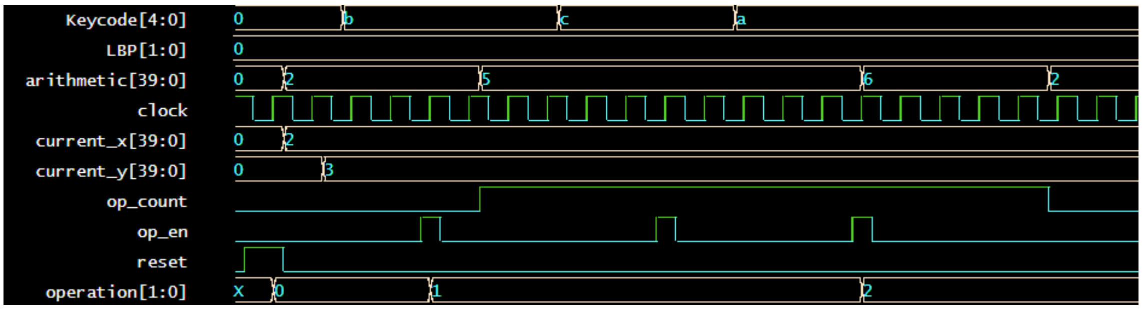
This verification does not cover every possible condition and functionality of the module (two operators pressed in a row, using the MR key, using the CA key to reset *op_count*, to name a few). This is because we knew these tests would be done using the self-written testbench on the design as a whole. This verification is more basic, to ensure the module is generally functioning and ready for the full test.

Operations (Jack Coleman)

The key functions of the "operations" module that I wanted to verify are:

- *arithmetic* is the value of *current_x* when *op_count* is 0
- *arithmetic* is the correct calculation when *op_count* is 1
- *operation* is assigned correctly, at the correct moment

Below is an example of a testbench used to verify this module. As can be seen, *operation* is correctly assigned to either 1 (addition) or 2 (multiplication) when *op_en* is high, and successfully ignores the equal key selection. Further, *arithmetic* is acting differently depending on the state of *op_count* and *operation*, either equalling 2, 5 (2+3), or 6 (2*3) depending on the circumstance.



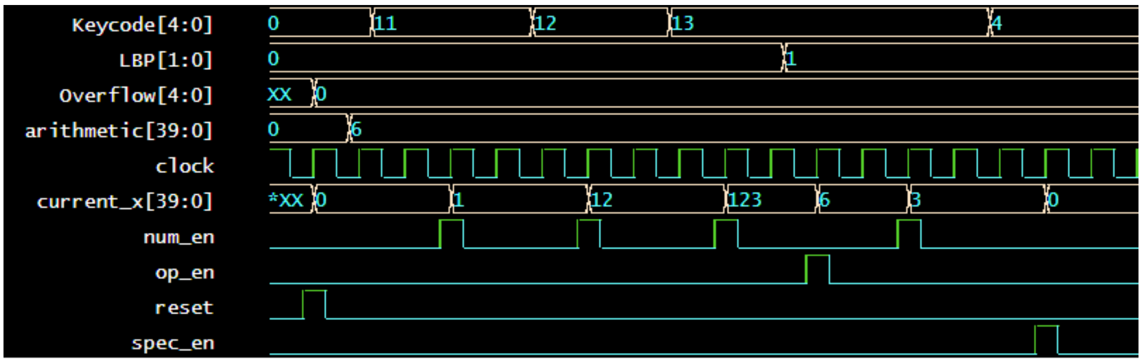
Again, this testbench dose not verify a lot of the more edge cases, such as interactions when the last button pressed is equal, or Overflow scenarios. A waveform testbench that covers all these scenarios would be quite big and cumbersome, and we intend to verify these edge cases at the end when the entire system is being verified more thoroughly in a much more precise and quick method.

X_Register (Ruairi Hogan)

Before beginning the verification process, I wrote down exactly what I wanted to test for:

- the ability for *x_register* to be affected by all three enables,
- The ability to insert values into *x_register*,
- The ability to create multi-digit numbers,
- The ability to start a new number on the keypress after an operator, equal, or clear key.

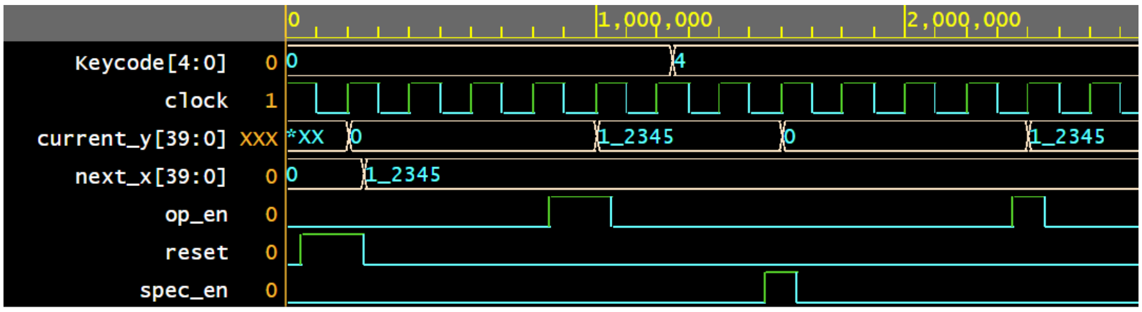
The example testbench below demonstrates how I tested for these functions. As can be seen, *x_register* can be set to “123” by inputting the three numbers sequentially with 3 *num_en* highs. An *op_en* correctly assigns *num_en* the value of *arithmetic*, and *spec_en* coupled with *Keycode*=CA sets *x_register* to 0. Finally, a *num_en* following *LBP* being 1 begins a new number, rather than adding the 3 digit to the 6 digit to get “63”.



Y_Register (Jack Coleman)

Verification of the Y-register was very simple, to the point that I was able to verify almost every feature of the module using only the testbench. Before preparing the waveform generator, I noted that I wanted to test that:

- An *op_en* set *current_y* to the value of *current_x*.
- *Current_y* will be set to 0 when *spec_en* is high, if *Keycode*=CA_KEY
- All other cases kept *current_y* the same.



This above testbench example tests for these functions. *Current_y* is not assigned a value until *op_en* is high, in which case it is immediately assigned the value of *next_x*, *12345*. Then, *current_y* is set to 0 on the *spec_en*, as *Keycode* is 0100, which is the keycode representation of the CA key. On the next *op_en*, *current_y* is returned to 12345.

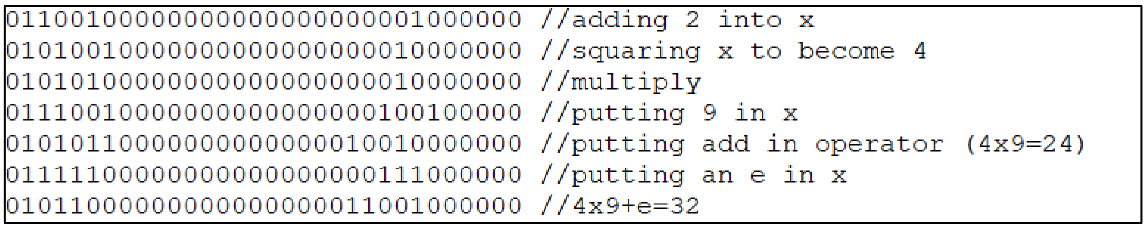
Main (Whole System) Verification (Joint)

The main part of the Verification system was the verification of the system as a whole. The first stage of the main verification plan was preparing the testbench test itself. We wrote down the most thorough list we could think of regarding the calculator's functions, and then prepared a test that covered all of these scenarios. Our test checked the calculator's ability to:

- Be properly reset using the reset button.
- Ignore an input when *newKey* is not high.
- Successfully add and multiply 2 multi-digit numbers together.
- Save a number to memory mid-calculation.
- Add or multiply a number with the answer to the previous calculation.
- Correctly chain 2 or more operations together in one calculation
- Continue displaying the first number until the second number is selected, and not upon operation selection.
- If more than 1 is operator is selected sequentially, utilize the last selected operator, but ensure operation chaining and display features still function.
- Square a number mid calculation.
- Recall a number from memory for use, both at the beginning and in the middle of a calculation.
- Clear only the X register with the *CE* key.
- Clear the X and Y register with the *CA* key, but not memory.
- Pressing the Equal button more than once in a row has no effect on the displayed number.

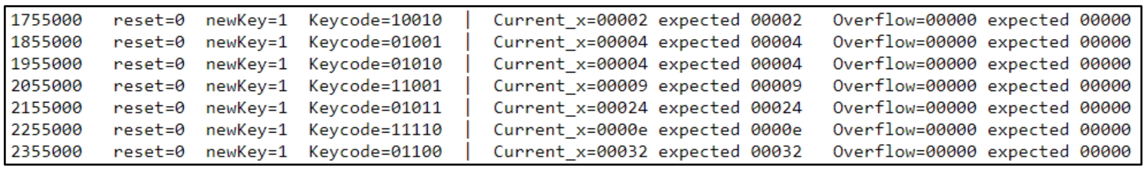
- Stop the user from entering more than a 5-digit number- (the inputs “123456” should cause the display to read “12345”- the 6 should be ignored.)
- Recognize when the display has reached a state of overflow, and do not leave that state until the CA key has been pressed.
- Leave the Overflow state when the CA key is pressed.
- function as intended after the Overflow state has been left.

In the end, this verification plan required 57 unique tests to be ran. There might be a plan that tests the same functions in less unique tests, but we felt that efficiency of tests quantity was not the biggest concern. We first wrote the 57 line text file, with each line containing 7 input bits (*reset*, *newKey*, and 5 bits representing *Keycode)*, and 25 output bits (20 representing *current_x_out*, and 5 represening *Overflow)*. Below is one example from the text file. These 7 lines were included to specifically test the square functionality and operation chaining.



```
01100100000000000000000001000000 //adding 2 into x
01010010000000000000000010000000 //squaring x to become 4
01010100000000000000000010000000 //multiply
01110010000000000000000100100000 //putting 9 in x
01010110000000000000001001000000 //putting add in operator (4x9=24)
01111100000000000000000111000000 //putting an e in x
01011000000000000000001100100000 //4x9+e=32
```

We then created a test bench that, before every clock cycle, fed the inputs as specified in the text file. After each clock cycle, the testbench compared the actual results with the expected results as indicated in the text file, and outputted the 57 results lines to a new, results file. Below is the corresponding results lines for the test file show above.



```
1755000   reset=0  newKey=1  Keycode=10010  |  Current_x=00002 expected 00002   Overflow=00000 expected 00000
1855000   reset=0  newKey=1  Keycode=01001  |  Current_x=00004 expected 00004   Overflow=00000 expected 00000
1955000   reset=0  newKey=1  Keycode=01010  |  Current_x=00004 expected 00004   Overflow=00000 expected 00000
2055000   reset=0  newKey=1  Keycode=11001  |  Current_x=00009 expected 00009   Overflow=00000 expected 00000
2155000   reset=0  newKey=1  Keycode=01011  |  Current_x=00024 expected 00024   Overflow=00000 expected 00000
2255000   reset=0  newKey=1  Keycode=11110  |  Current_x=0000e expected 0000e   Overflow=00000 expected 00000
2355000   reset=0  newKey=1  Keycode=01100  |  Current_x=00032 expected 00032   Overflow=00000 expected 00000
```

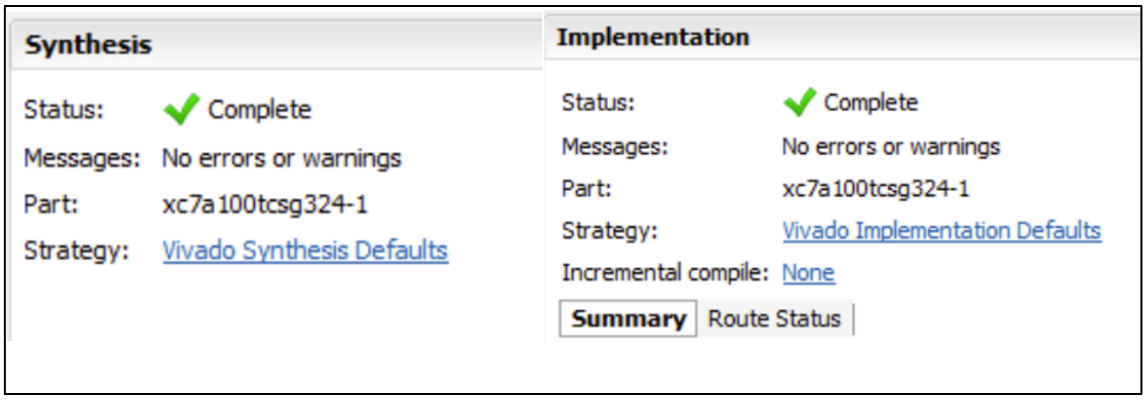
As can be seen, we displayed the *current_x* values in hexadecimal, which greatly helped in the de-bugging process as that is how we visualized the display. We kept every other input and output in binary, as that is how we were visualizing them mentally.

This verification method was very efficient in identifying problems with the design- as we expected, most bugs were due to not realizing how a fundamental part of a feature should work, which the module-specific verification method would not detect. During the first 3 or 4 verification cycles, over half of the tests were incorrect, due to the compounding of bugs For example, the input “5+MR=” (where MR is 3) should display an 8. However, in our original design, we did not realize to label the MR key as a number press, and so the calculator failed to recognize that an operation had occurred and simply displayed a 5. Our original design also failed to hold the same value on multiple presses of the EQUAL key and would instead repeat the operation, due to a mis-understanding of how the *arithmetic* signal should be generated (That could have been an accidental feature if we weren’t able to fix it). We also noticed that the y register was not storing what we thought it would be, a side-effect of the bug explained in that hardware sections. Other smaller bugs also occurred, and this method allowed us to catch all of them.

As a whole, verifying the whole calculator at once in this fashion felt far easier to comprehend than a massive waveform would be.

Synthesis and Implementation

As seen below, our design had no errors or warnings during synthesis and implementation.


Synthesis

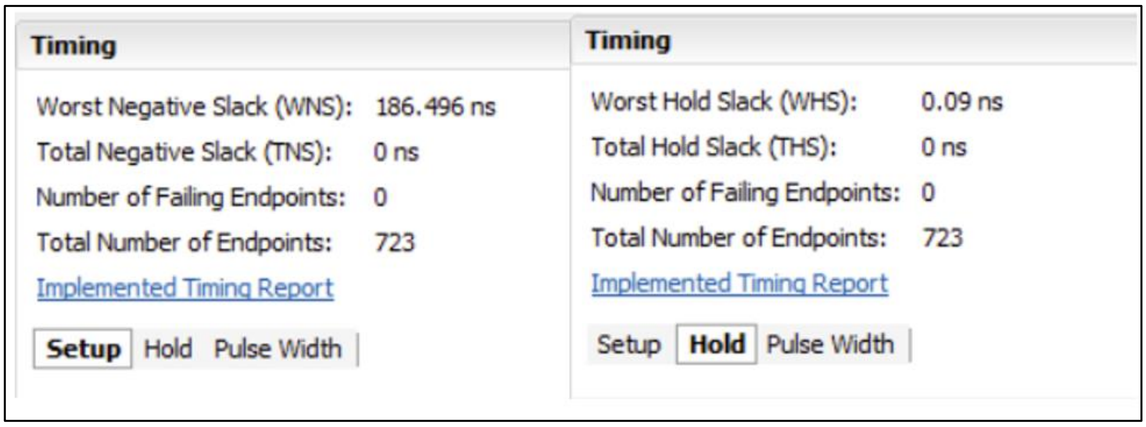
Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Our design had a Worst Negative Slack of 186.5ns. This means that in the worst case, our design finishes its combinational logic with 186.5ns to spare before setup time. This result is quite re-assuring, as it demonstrates that signals have quite a bit of buffer time to meeting timing constraints.

Our design's Worst Hold Slack is .09ns. This means that a register can change after a clock cycle at worst .09ns after the Hold time constraint concludes. A positive slack is of course good, but it is not a lot of slack. If we had more time with this project, we might look at the design and see if we could increase the Worst Hold Slack to maybe .15ns or .2ns, just for added stability.


Timing

Worst Negative Slack (WNS):	186.496 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	723

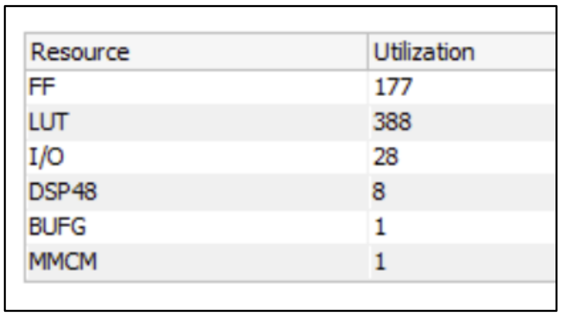
Implemented Timing Report

Timing

Worst Hold Slack (WHS):	0.09 ns
Total Hold Slack (THS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	723

Implemented Timing Report

Below are our resource utilization data. We did not use a large percentage of the available hardware resources on the board, and so we do not think there is much to worry about here.



Resource	Utilization
FF	177
LUT	388
I/O	28
DSP48	8
BUFG	1
MMCM	1

Hardware Testing and Conclusion

Every required and additional feature of our calculator functioned on the first attempt of hardware.

We tested the calculator on hardware by first simply using the device: entering operation chains, using the special keys, and doing complication calculations. We then checked the edge cases- overflow detection and 5-digit numbers, typing the enter button or an operator twice in a row, using the memory features in strange ways. In all cases, the device worked perfectly as intended. A picture of a calculator is not a good way of displaying its functionality, but below is a picture for completeness' sake.

We are happy with how the calculator product functioned. It felt like a complete product in every case (except perhaps the lack of subtraction and division), and at no point did the Quality of Life using product feel lacking, which was our goal.