Xuan Yuan Huang

Alvin Lim

Group 11

CS M152A

Lab 4

**Introduction**

In this lab, we implemented a simple integer calculator, with the addition of four registers to allow switching between calculations. Our calculator has the following functions:: (a) integer addition, subtraction, multiplication, and division; (b) “registers”, which may hold up to 4 results concurrently and can be reset; and (c) a display that supports display registers, their values, and editing input operands. When using only one register, the calculator displays the result of the last calculation and allows the user to perform four basic operations (addition, subtraction, multiplication, and division) with its current value and a new input as operands. The input is selected after the operation using a basic digit editor, with left and right buttons scrolling between digits and up and down buttons incrementing and decrementing the chosen digit.

Since the seven-segment display only supports four digits at a time, the calculator will show ERR if a result is larger than 9999 or smaller than -9999. The calculator also shows ERR if the user attempts to divide by zero. Any operation with ERR will result in ERR. Negative values are represented by turning on all decimal points on the seven-segment display; user inputs can also be flipped between positive and negative using the first switch.

Users can employ the four registers by entering memory mode with the middle button, then scrolling between registers with the left and right buttons. The down button resets the current register from any value, including ERR, to 0. The display flashes between the number of the current register and its value for readability. Registers save the result of the latest calculation performed on them, and if returned to will restore that value. For example, if one adds 5 to 0 in register 0, register 0 will remember 5. If the user switches to other registers and performs calculations on them, register 0 will still contain 5. When the user switches back to register 0, the calculator will once again display 5.

Our original stretch goal was a “PEMDAS” mode that allowed users to operate on all four registers at once, with order of operations taken into account – this did not make it into the final implementation.

**Design**

***Inputs:***

**clk** - Used to make clocks for other functionality (display, counters, etc.)

Operations (Default Mode)

**btnU** - *Multiplication:* enter digit editing mode and multiply the current register by the input

**btnD** - *Division:* enter digit editing mode and divide the current register by the input

**btnL** - *Subtraction:* enter digit editing mode and subtract the input from the current register

**btnR** - *Addition:* enter digit editing mode and add the input to the current register

**btnS** - *Memory:* enter memory mode

Operations (Digit Editing Mode)

**sw[0]** - set to 1 to use a negative number; 0 for positive number

**btnU** - Increase the current digit by one

**btnD** - Decrease the current digit by one

**btnL** - Move the cursor by one digit to the left

**btnR** - Move the cursor by one digit to the right

**btnS** - Exit digit editing mode with the currently displayed number as input to the calculator and return

to default mode

Operations (Memory Mode)

**btnL** - Change to the register to the left

**btnR** - Change to the register to the right

**btnD** - Reset the currently displayed register to zero

**btnS** - Select the current register and exit memory mode to return to default mode

Operations (PEMDAS, Stretch Goal)

**sw[1]** - set to 1 to immediately switch into PEMDAS mode.

**btnU** - *Multiplication:* request calculator multiply the two registers displayed

**btnD** - *Division:* request calculator divide the two registers displayed

**btnL** - *Subtraction:* request calculator subtract the two registers displayed

**btnR** - *Addition:* request calculator add the two registers displayed

*\*\* Once 3 operations are requested, the calculator executes the operations in the correct order and*

*briefly displays the result*

***Outputs*:**

**Seven-Segment Display** - Displays value of current register, all decimals lit up for negative #s

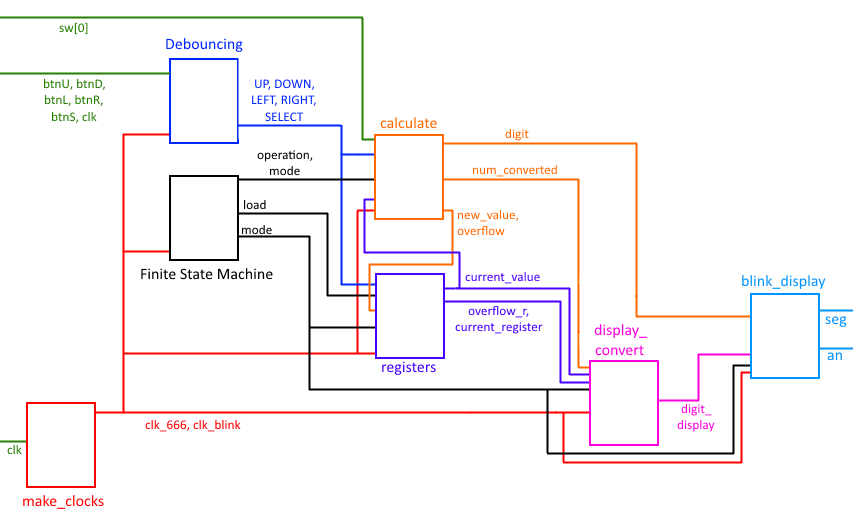
Alternates between register number and value if in Memory Mode

Flashes the digit being adjusted in digit editing mode

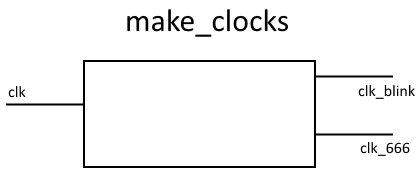
Displays the number of the two registers currently being operated on in PEMDAS mode

***High-Level Schematic:***

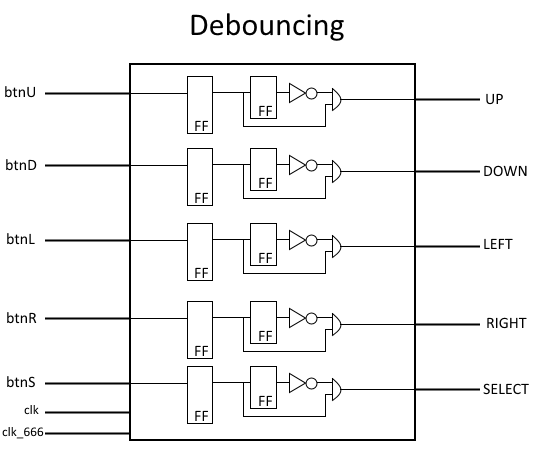
The following is a high-level schematic, with several abstractions made for readability due to the complex interactions between modules. Wires are combined, wire sizes are omitted, and not all inputs may be used (most modules don’t use clk\_666 and clk\_blink, for example).



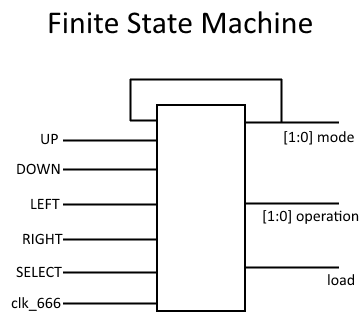
***Module-by-module Breakdown:***



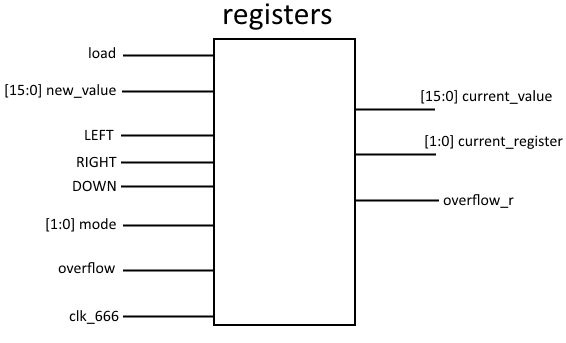
Divides the 100 MHz onboard clock signal “clk” into clk\_666 and clk\_blink using counters.



An abstraction of the actual implementation in calculator.v, Debouncing cleans input by feeding it through a flip-flop, then converts the signals (which are 1 if buttons are held down and 0 otherwise) to “posedges” (bursts one clock-cycle long) to pass to other modules. The clocks are used to time the flip-flops, with clk timing the first column of flip-flops and clk\_666 the second (not drawn in schematic for clarity).

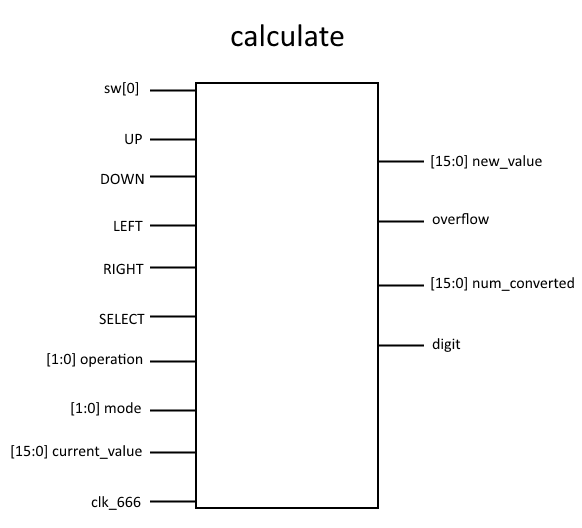


The Finite State Machine acts like a state transition table between three states: default mode (the calculator is simply displaying a register), digit edit mode (the calculator is waiting for user input), and memory mode (the calculator is displaying registers and their contents and allowing them to be changed or cleared). Mode changes are as described previously; the FSM is additionally responsible for changing the current operation and reporting mode and load events, so that calculate knows which operation to perform and registers know when to save.



The registers module contains four registers and remembers which register is currently active.

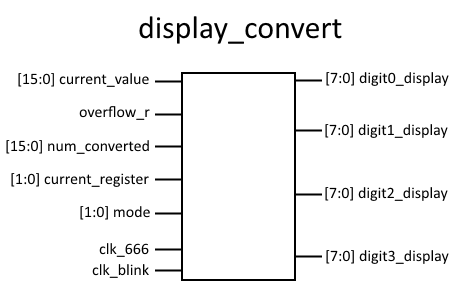
* Inputs
  + It loads new\_value into the currently active register if load is triggered by the FSM.
  + When the calculator is in memory mode, registers checks LEFT and RIGHT to change the currently active register and DOWN to reset (wipe) the active register.
  + If calculate reports an overflow, registers remembers that the register active for that calculation overflowed.
* Outputs
  + The register persistently outputs the value stored in the active register as current\_value, as well as the numeric ID of that register as current\_register. If a register overflowed in some prior calculation and has yet to be reset, it reports so in overflow\_r.



The calculate module is highly complex, but fundamentally it is responsible for taking user inputs during calculation mode and determining the result.

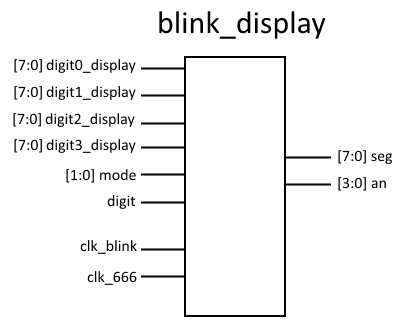
* Inputs
  + sw[0] simply tells calculate whether or not to treat the current user input as a positive or negative operand
  + UP, DOWN, LEFT, and RIGHT are used as previously described to allow the user to edit their input
  + SELECT tells calculate to finalize its calculation and report the results via new\_value and (if the calculation overflowed) overflow
  + operation informs the module whether to add, subtract, multiply, or divide
  + mode informs the module whether to do anything at all (calculate only operates when the calculator as a whole is in calculation mode)
  + current\_value is the value stored in the active register, for use as the first operand
* Outputs
  + new\_value is the finalized result of the calculation, directed to registers for storage
  + overflow indicates if the final result overflowed, and reports to registers
  + num\_converted is the current user input, and is directed to display\_convert to allow the user to see their operand
  + digit indicates which digit in the input the user is currently editing, and is directed to blink\_display so that it can be blinked for clarity

Internally, the calculate model repeatedly calculates the numeric value of the user’s inputs from the four digits being edited, and carries out the actual operation when the user confirms with SELECT.



Based on what mode the calculator is in, display\_convert must generate the raw bitstrings digit0\_display through digit3\_display for each digit on the display:

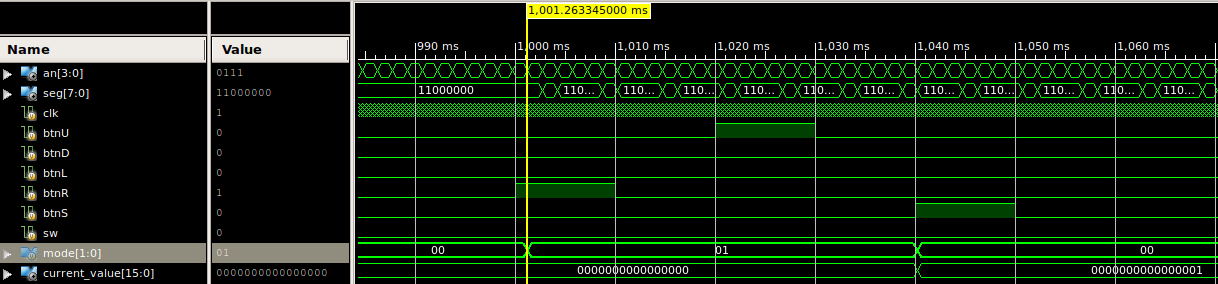
* In normal mode, display\_convert converts current\_value from the registers to digit\_display
  + If the registers module enables overflow\_r, then display\_convert instead produces the bitstrings to display “ERR”
* In calculate mode, display\_convert instead converts num\_converted from the calculate module, allowing users to see their input as they modify it
* In memory mode, display\_convert uses clk\_blink to alternate between converting current\_value and current\_register to display - for the end user, this lets them see both what register is active and its ID



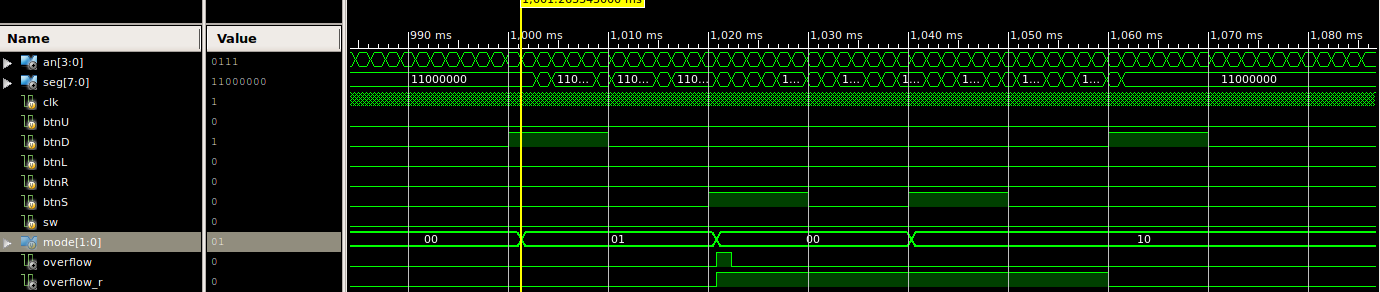
The module blink\_display generally has the simple task of alternating between the four digit\_display inputs from display\_convert at 666 Hz to determine seg (the seven-segment display) and an (which digit on the seven-segment display to use with seg).

* When in calculate mode, blink\_display uses clk\_blink to stop displaying the digit being edited for 0.75s, effectively blinking that digit

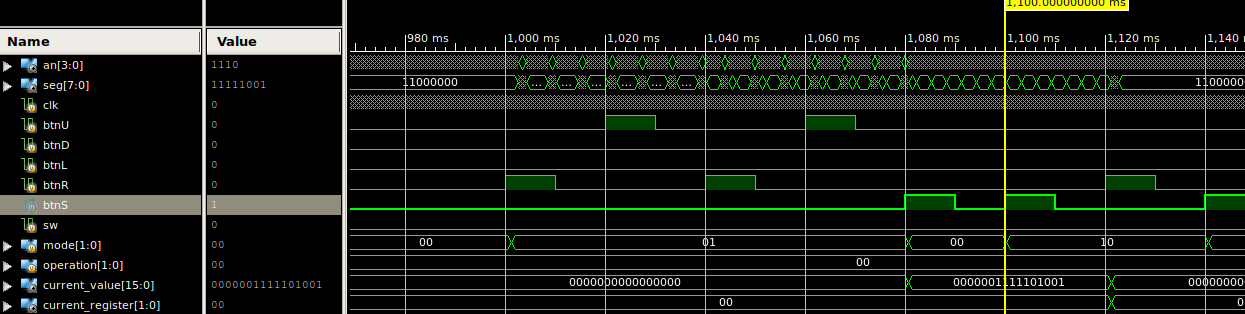
**Simulation**



As evident in this simulation, when users press btnR (add), a mode variable changes, to switch to Digit Editing Mode (01). In there, the directional buttons can be used to edit what number to add to the register. When btnS is then pressed, one can see that current\_value[] is updated to reflect the result: 1.



This simulation shows the implementation and functionality of overflow and reset. When btnD is pressed, the calculator waits for a number to divide the register by. Since btnS is pressed, the calculator tries to divide by 0, which results in an undefined result, so we set the overflow flag, which prompts the screen to output Err (Error). This output is also displayed when the result is for numbers larger than 9999 or numbers less than -9999. The register keeps this value until users reset the register: btnS goes into Memory Mode, in which btnD resets the value of the register to 0 and clears the overflow flag.

The following shows our implementation of changing registers:

(1) btnR: Enter digit edit mode, “add...”

(2) btnU: Increment ones place by 1, “add by 1” not yet confirmed

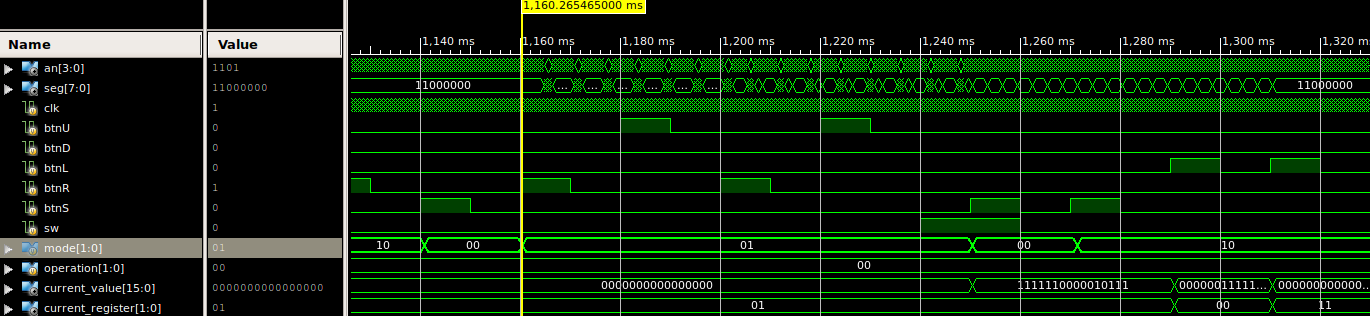
(3) btnR: Move to tens place

(4) btnU: Increment tens place by 1, “add by 11” not yet confirmed

(5) btnS: Confirm “add by 11”, reflected in current\_value[], which shows that current\_register 0 contains 11.

(6) btnS: Enter memory mode. current\_register = 0, current\_value = 11.

(7) btnR: Switch to next register. current\_register = 1, current\_value = 0.



(8) btnS: Select current register. current\_register = 0, current\_value = 0.

(9) btnR: Enter digit edit mode “add…”

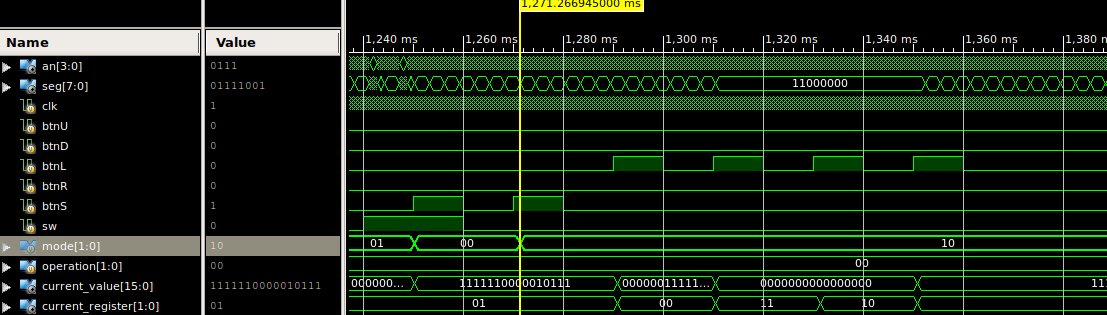
(10) btnU: Increment ones place by 1, “add by 1” not yet confirmed

(11) btnR: Move to tens place

(12) btnU: Increment tens place by 1, “add by 11” not yet confirmed

(13) sw[0]: Mark input as negative. “add by -11” not yet confirmed

(14) btnS: Confirm “add by -11”, reflected in current\_value[] which shows that current\_register 1 contains -11.



(15) btnS: Enter memory mode. current\_register = 1, current\_value = -11

(16) btnL: Change to previous register. current\_register = 0, current\_value = 11

(17) btnL: Change to previous register (0 wraps to 3). current\_register = 3, current\_value = 0

(18) btnL: Change to previous register. current\_register = 2, current\_value = 0

(19) btnL: Change to previous register. current\_register = 1, current\_value = -11

**Conclusion**

This lab served as a capstone for the class, so unsurprisingly we encountered many difficulties. One of the greatest problems was timing – the calculator is essentially a very complex finite state machine, and things often behaved in the wrong order. At the heart of the calculator was a state transition table - understanding sequential versus combinatorial assignment and the nature of the Verilog sensitivity list for always blocks was key to getting it to work as expected. We initially tried to use posedge of buttons for the sensitivity lists, but they caused timing issues since there were so many of them, and so we had to implement our a different type of debounced signal that was more like a “blip” that would last long enough for us to check the signal in a clock cycle (with posedge of the clock as the sensitivity list).

We also found low-level design problems as we worked. For example, it would likely have been better to combine all the registers into a single module with array storage, as is abstracted in this report. Overall, the lab was a good test of skills developed throughout the course.