Cooper Mattern
Prof. Paul Hummel
CPE233-05
March 14, 2022

OTTER Calculator

Introduction

Thank you for purchasing the OTTER Calculator! A simple, no frills, quick calculator to get you through the day!

The OTTER Calculator is a four-operation calculator (addition, subtraction, multiplication, and division) where all you need to do is hook up a keyboard and display and you're ready to go! This calculator supports up to **six** total numbers and can display negative numbers that are produced by subtracting. The calculator has safeguards built in to prevent you from inputting too many digits (remember, no more than **six** total numbers to be input), too many operands (e.g., "1++2" and "-1+-2" are not possible), and too many overall inputs (**seven** is the maximum. E.g., "1+23456"). When dividing, the OTTER Calculator only outputs the rounded down, truncated quotient (E.g., "109/10" will output "10"). This type of division was used to keep the outputs simple and easy to comprehend. With the information I've given in hand, how more excited could you be to own an OTTER Calculator!

Owner's Manual

When you first start up the OTTER Calculator you will be greeted with a blank white bar on your screen. This is where you will see all your inputs into the calculator, and after you hit enter, the answer of your requested calculation.

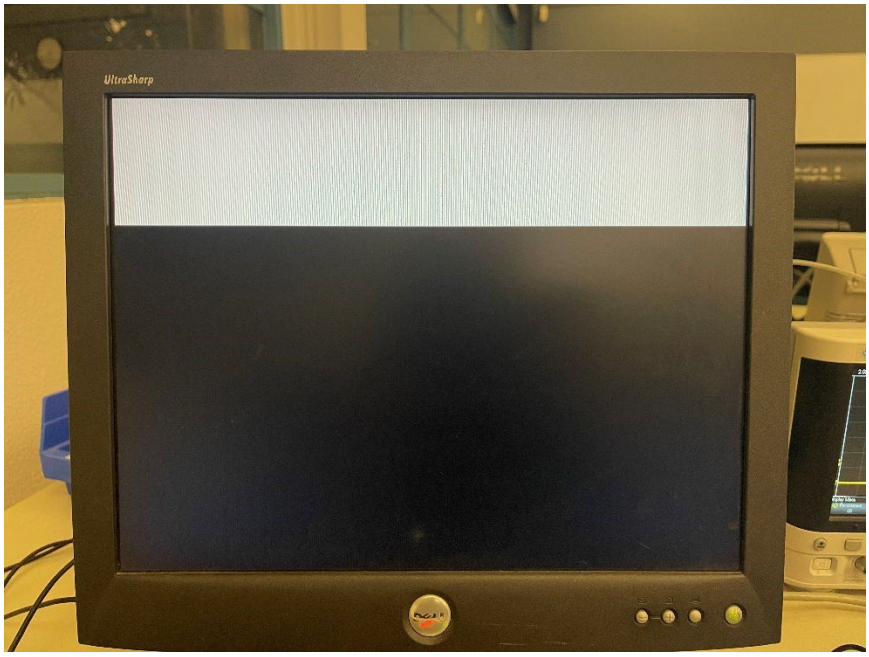

Figure 1. Start screen of OTTER Calculator

After you start typing on the keyboard, your inputs will start to show up in the white bar. You enter a calculation on the calculator by entering the first number on the number row of the keyboard, the operation you want to do, and then the second number. Below is the table of keyboard keys that correspond to their operations.

Table 1. Operations of the OTTER Calculator

Keyboard Key	Operation
“-/_”	Subtract
“=/+”	Add
“X”	Multiply
“//?”	Divide

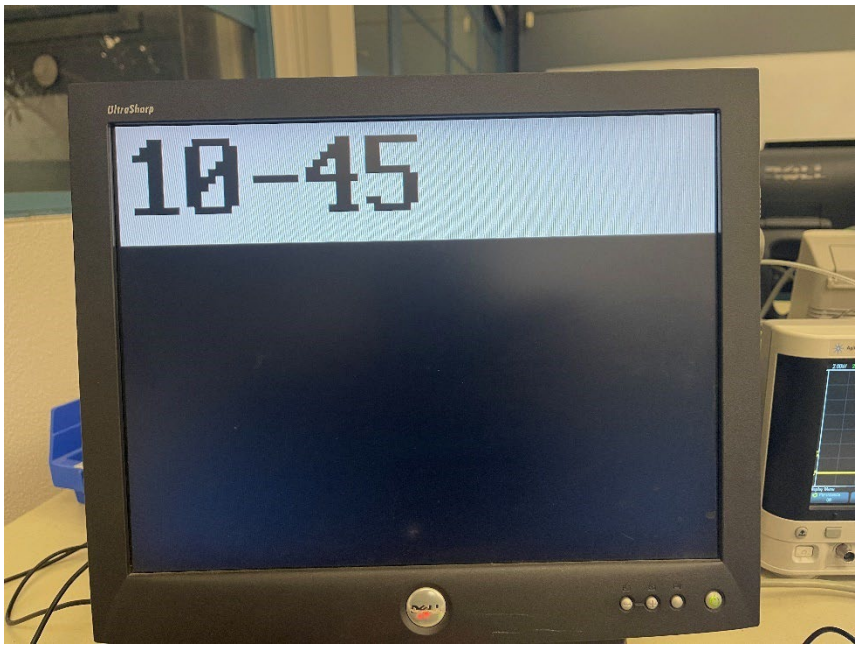

Figure 2. Example input of OTTER Calculator

Once you are finished entering your input, go ahead and press the enter key to view the answer!

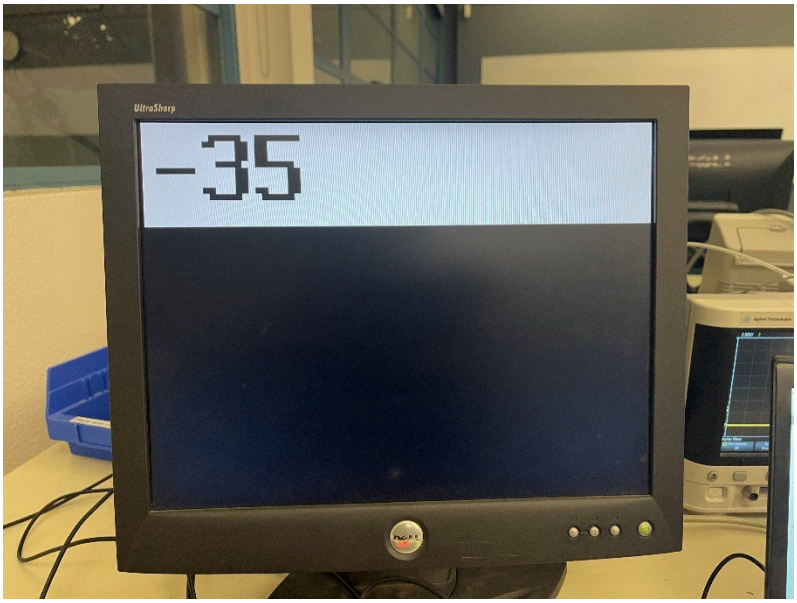

Figure 3. Result of previous expression on OTTER Calculator

After getting your answer, putting in a new calculation is as easy as typing the first number of the next calculation. The OTTER Calculator will clear the screen and show the first number you typed in. Follow the steps above to get your new answer and repeat.

If you want to clear the screen for any reason, press the center button on the OTTER Calculator circuit board.

Some notes:

1. The OTTER Calculator **will not** allow a first input to be an operand.
2. The OTTER Calculator **will not** allow two operands to be input.
3. The OTTER Calculator **will not** allow more than **six** total numbers to be input; split between the first number and the second number.
4. The OTTER Calculator **will not** allow more than **seven** total inputs for a calculation.
5. The OTTER Calculator performs what is called **"floor division"**. This means the answer is only the quotient **with no remainder or decimal** and is **rounded down** to the **nearest quotient**.
6. The OTTER Calculator **does not** support deleting. If you mess up when typing in an expression, **you must clear the screen and start again**.

As a result of the first two points, **negative numbers cannot be input**. However, a negative number can still be a result of subtraction, as seen above. Examples of points three, four, and five can be seen on the next page.

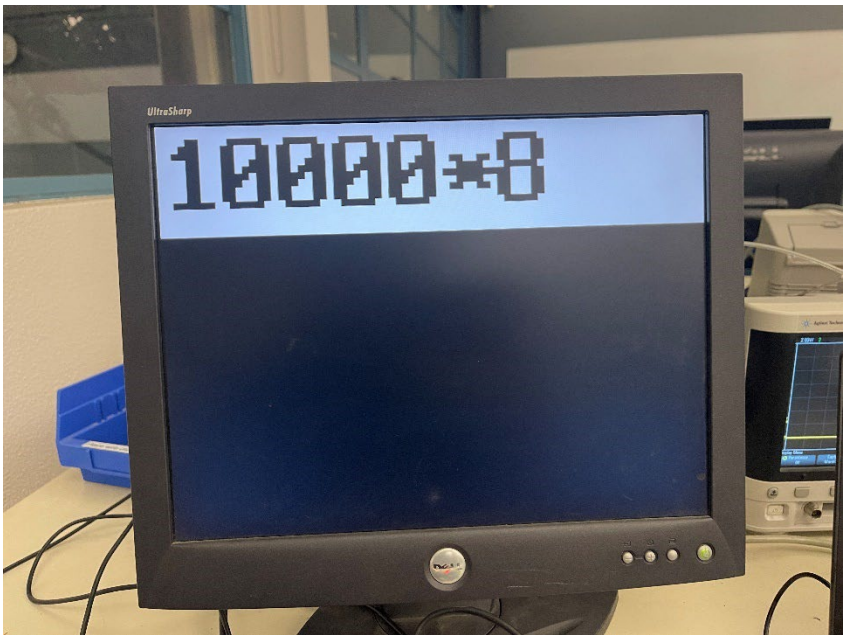

Figure 4. Six total numbers being input

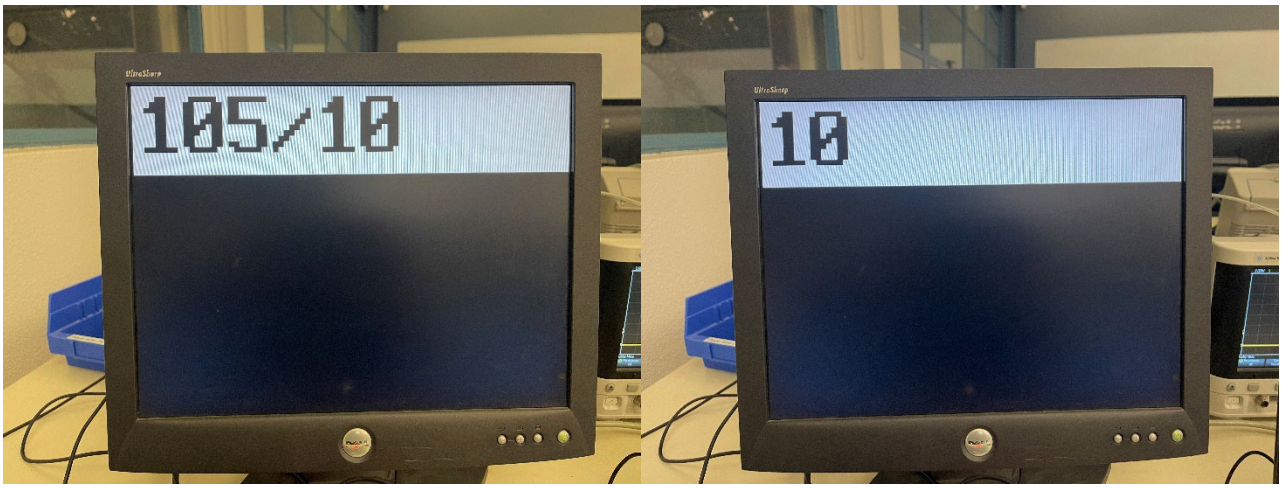

Figure 5. Example of "floor division"

Software Design

The OTTER Calculator functions by waiting for an input from the keyboard, processing the input, and then displaying the input on the screen.

When waiting for an input, the OTTER Calculator simply waits for an interrupt to be called, stores the scancode put into the MMIO address from the keyboard driver, and then saves the scancode in the data segment for later retrieval. The OTTER Calculator then returns to the main loop to call the process input subroutine.

When processing the input, the OTTER Calculator retrieves the scancode input, then checks if we have typed all seven inputs or all parts of the expression (first number, operand, second number) have been typed in. If either are true, it checks for an enter press. If only the first is true, the only valid input is the enter key. This is achieved with return address manipulation. After checking for an enter press, it goes into a loop where it loads scancodes from the data segment and checks against the scancode that was input to find if a valid input was made. If a number was input, it will concatenate the number input to the current number on the screen using multiplication and addition and add the result to the **expression stack**. If an operand was input, it will add a coded operand to the expression stack. Both previous statements result in the number or operand being displayed on the screen. The expression stack is three data values stored in the stack to represent the expression entered that are manipulated and retrieved during processing and calculation. The stack is heavily used in the program as it allows for quick, non-destructive, storage and retrieval of values.

When displaying the input or output, the OTTER Calculator uses visual representation data stored in the data segment. The visual representation of numbers and operands were derived from this source. I needed to write a short Python script to process the data and convert it from a text document to an array of word separated values. Each number or operand takes up 4 words and is read row-by-row which are byte-indexed in the data segment. Conveniently, with each number and operand being an 8x16 resolution, the indices in the data segment are easy to understand with “0” being at 0x6000, “1” being at 0x6010, “2” being at 0x6020 and so on. The operands are stored at the end of the segment and are coded as “0xA” being addition, “0xB” being subtraction, “0xC” being multiplication, and “0xD” being division. The use of this coding of operands allows for easy looping during processing and simple concatenation during display. During the display function, each row is read from the byte index that is incremented by 1 as we move from top to bottom on the visual representation. Each byte is then masked starting with the MSB of the byte to reveal if a pixel should be black or white, depending on if the bit is a 1 or a 0, respectively. Horizontal pixel position is tracked by a globally saved register, and we add the current column index to the overall horizontal position to correctly display pixel in the right spot. To increment through the columns of the number or operand, we mask the 7^{th} bit of the data and then shift the data to the right to traverse through the data. The display function calls J. Callenes and Prof. Hummel’s “draw_dot” subroutine to write the pixel position and color to the VGA output. Once a number or operand is displayed, we increment the horizontal position to get ready for the next display call. To clear the entire display, the background color is written to each pixel of the display area and the horizontal position is reset.

During calculation, the OTTER Calculator pops the expression stack and decodes the operand into the correct operation. I increment down from “0xD” for division and to “0xA” for addition. I start with division because the division subroutine easily creates the most calls as it is a simple division-by-repeated-subtraction method, and I don’t want to run anymore instructions than necessary. I used a multiplication method that only uses addition and bitwise operators from this source. One of my goals of the project was to reduce function calls and create efficiency where needed. During subtraction, the result is checked for sign and if the result is negative, a minus sign is first displayed, followed by negation of the result before returning to the calculation subroutine. Once the result is calculated, we pass the result into a modified version of my BCD program from SW7-2. Using a BCD will allow us to then get each decimal digit out of the result, again using bit masking and shifting. Each digit is ORd with “0x6000” and stored in the stack for looping through and displaying each digit. Because I am bit masking from LSD to MSD, when it comes time to pop each value, the output produces digits in the correct order. To clear the screen, instead of using the instruction-heavy method of clearing the entire screen and then displaying each digit, I opted for just displaying the digits of the result and then writing blank characters to the remainder of the screen until I’ve gotten to 7 total display operations.

Once calculation is done, any number input from the user will completely clear the display, display the number, and then wait for more input.

Overall, I am proud with the final product and felt that I found interesting solutions to the simple problem of making a calculator.

On the next page is the flowchart for the OTTER Calulator.

Appendix

```
.data
Visrep: .word
#load in visual representations of 0-9,+,-,*,/. The 1's and 0's (for writing black and write to a
#pixel region) are stored in 4 word chunks, where each byte is one line. The final numbers are 8x16 in
#resolution, so I can fit 7 numbers/symbols per line.
0xc67c0000,0xf6decec6,0x7cc6c6e6,0x0,0x38180000,0x18181878,0x7e181818,0x0,0xc67c0000,0x30180c06,0xfec6
c060,0x0,0xc67c0000,0x63c0606,0x7cc60606,0x0,0x1c0c0000,0xfecc6c3c,0x1e0c0c0c,0x0,0xc0fe0000,0x6fcc0c0
,0x7cc60606,0x0,0x60380000,0xc6fcc0c0,0x7cc6c6c6,0x0,0xc6fe0000,0x180c0606,0x30303030,0x0,0xc67c0000,0
xc67cc6c6,0x7cc6c6c6,0x0,0xc67c0000,0x67ec6c6,0x780c0606,0x0,0x0,0x7e181800,0x1818,0x0,0x0,0x7e000000,
0x0,0x0,0x0,0xff3c6600,0x663c,0x0,0x0,0x180c0602,0x80c06030,0x0, 0x0, 0x0, 0x0, 0x0
#representation of negative sign, this is a half width symbol as we only have space for 7.5 symbols
#per line
.byte
Codes: 0x45, 0x16, 0x1e, 0x26, 0x25, 0x2e, 0x36, 0x3d, 0x3e, 0x46, 0x55, 0x4e, 0x22, 0x4a
SCANCODE:
#set addresses for the symbols and numbers
.eqv disp_zero, 0x6000
.eqv disp_one, 0x6010
.eqv disp_two, 0x6020
.eqv disp_three, 0x6030
.eqv disp_four, 0x6040
.eqv disp_five, 0x6050
.eqv disp_six, 0x6060
.eqv disp_seven, 0x6070
.eqv disp_eight, 0x6080
.eqv disp_nine, 0x6090
.eqv disp_plus, 0x60a0
.eqv disp_sub, 0x60b0
.eqv disp_div, 0x60c0
.eqv disp_mult, 0x60d0
.eqv disp_blank, 0x60e0
.eqv disp_BGCOL, 0xff
.eqv disp_BLACK, 0x0

#set addresss for MMIO
.eqv MMIO, 0x11000000

.text
```

```
main:   li sp, 0x10000          #initialize stackpointer
        li s0, MMIO             #load MMIO address and SCANCODE address
        la s1, SCANCODE

        la t0, ISR              #set ISR address in mtvec
        csrw t0, mtvec
        addi t0, zero, 1        #enable interrupts
        csrw t0, mie

        jal clrscn              #clear the screen
        lui t0, 0               #initialize interrupt flag
        lui s2, 0               #initialize total input count
        lui s3, 0               #initialize number input
        lui s4, 0               #initialize operand count

loop:   beq t0, zero, loop      #check for interrupts
        jal proc
        addi t0, zero, 1
        csrw t0, mie            #re-enable interrupts
        lui t0, 0               #reset interrupt flag
        j loop

#load in code from keyboard
ISR:    lw  t0, 0x100(s0)       #load scancode
        sb t0, 0(s1)            #save to scancode
        addi t0, zero, 1        #set interrupt flag
        mret

####################process input function##############################################
proc:   lbu t0, (s1)            #load in scancode
        addi t1, zero, 7        #initalize value to check if we've typed
                                #the max inputs for the line
        bgeu s2, t1, enter      #check enter press if max inputs first so we dont
                                #run through the process loop unnecessarily
        addi t1, zero, 3        #initalize value to check if we've typed
                                #3 operands so we have a valid expression
        blt s4, t1, ckinpt      #dont check enter if havent typed in 3 operands
        addi sp, sp, -4         #save ra
        sw ra, (sp)
```

```
        jal enter               #check if enter was pressed
        lw ra, (sp)             #reload ra if enter not pressed
        addi sp, sp, 4
ckinpt: lui t1, 0               #initialize counter for loop
        la t2, Codes            #initialize scancode array address for loop
        la t3, Visrep           #intialize visrep array address for loop
        ploop:  lb t4, (t2)             #load scancode to check from address
                addi a0, t3, 0          #send visrep addresss to argument
                addi t5, zero, 0xa      #initialize value for checking symbols
                addi a1, t1, 0          #send counter to argument
                blt t1, t5, num         #if counter is below 10, we are checking numbers
                ########symbols############
                beqz s3, return         #return to main loop if we havent typed a number
                addi t5, zero, 1        #value for checking if we have already typed 3 operands and try to
                bne s4, t5, return      #input another symbol
                addi s2, s2, 1          #increment inputs
                addi t6, s3, 0          #temp store numbers typed in
                lui s3, 0               #clear numbers typed in
                addi s4, s4, 1          #increment operand count
                addi sp, sp, -4                 #increment sp since we are at next operand
                sw t1, (sp)             #store encoded operande in stack
                beq t0, t4, disp        #check symbol value
                addi s2, s2, -1                 #decrement inputs
                addi s3, t6, 0          #restore numbers typed in
                addi s4, s4, -1                 #decrement operand count
                sw zero, (sp)           #reset value in stack and back off stack bc incorrect symbol
                addi sp, sp, 4
                #######numbers############
        num:    beq t0, t4, add_num     #check number
                addi t1, t1, 1          #increment counter
                addi t2, t2, 1          #increment scancode address
                addi t3, t3, 0x10       #increment visrep address
                addi t4, zero, 0xe      #set value for checking if we dont have a valid input
                bltu t1, t4, ploop
        jr ra

enter:  addi t1, zero, 0x5a     #checking for "enter" which is the call to calculate
        beq t0, t1, calc
        jr ra
```

```
###################adding a number from intput function#################################
add_num:addi t0, zero 5      #load in value checking number input
        beq s3, t0, return   #check if we are at our max number input
        addi s2, s2, 1       #increment inputs
        addi t0, a0, 0       #load in display address
        addi t1, a1, 0       #load in number to add in
        bnez s4, skip0       #if this is the first input of the line clear the screen
        addi sp, sp -12      #save ra, t0, t1
        sw ra, (sp)
        sw t0, 4(sp)
        sw t1, 8(sp)
        jal clrscn
        lw ra, (sp)          #reload ra, t0, t1
        lw t0, 4(sp)
        lw t1, 8(sp)
        addi sp, sp, 12
skip0:  bnez s3, skip1       #increment sp if we typing the first number
        addi s4, s4, 1       #increment operand count on first number typed
        addi sp, sp, -4      #increment stack pointer on first number input
        sw zero, (sp)        #and store a zero
skip1:  addi s3, s3, 1       #increment numbers input
        lw t2, (sp)          #grab current number
        addi sp, sp, -8      #save ra, display address, and add number
        sw ra, 8(sp)
        sw t0, 4(sp)
        sw t1, 0(sp)
        addi a0, t2, 0       #multiply current number by 10
        jal ten
        lw t1, 0(sp)         #reload number to add in
        addi sp, sp, 4
        add t2, t1, a1       #load result from multiplication and add new number
        lw a0, (sp)          #load in display address to argument register
        addi sp, sp, 4
        lw ra, (sp)          #reload ra
        sw t2, (sp)          #store new number in sp
        j disp
```

```
#################################calculation function#################################
calc:   addi t2, zero, 7      #checking if we have maximum inputs
        beq s2, t2, cont      #if not, the proper ra was saved in the stack
        lw ra, (sp)           #load ra
        addi sp, sp, 4
cont:   addi s5, zero, 1      #clear horizontal display position
        lw a0, 8(sp)          #first number
        lw t0, 4(sp)          #operand
        lw a1, (sp)           #second number
        addi sp, sp, 12
        addi t1, zero, 0xd    #checking for divide
        beq t0, t1, divid
        addi t1, t1, -1       #checking for multiply
        beq t0 t1, multi
        addi t1, t1, -1       #checking for subtraction
        beq t0, t1, subt
        add a1, a0, a1        #add numbers and save to bcd argument input
calcret:addi sp, sp, -12      #save ra, scancode address, and horizontal display pos
        sw ra, (sp)
        sw s1, 4(sp)
        sw s5, 8(sp)
        jal bcd               #generate bcd from answer
        lw ra, (sp)           #reload ra, scancode address, and horz disp pos
        lw s1, 4(sp)
        lw s5, 8(sp)
        addi sp, sp, 12
        addi t0, a0, 0        #load in result from bcd

        #######decoding calculation bcd result##################
        li t1, 0xF                    #bitmask for first number
        lui t2, 0                     #initialize temp for calc lp for number
        addi t4, zero, 7              #nitalize counter for number of white spaces needed
        addi sp, sp, -4               #save ra
        sw ra, (sp)
        calclp:and t2, t0, t1         #get number with bitmask and save to argument
               slli t2, t2, 4         #put number in 2 LSB hex digit
               li t3, 0x6000          #0x6000 + 0xx0 = 0x60x0 == address
               or t2, t2, t3
               addi sp, sp, -4        #increment stack pointer
```

```
            sw t2, (sp)           #save decoded address in stack
            srli t0, t0, 4        #shift number for mask
            addi t4, t4, -1       #decrement whitespace needed
            bnez t0, calclp       #check if we are done loading the stack

        #########displaying number#################################
        addi t0, t4, 0            #mv t4 to t0 and t1
        addi t1, t4, 0
        displp: lw a0, (sp)       #load number address
            addi sp, sp, 4
            addi sp, sp, -8       #save counters for blanks
            sw t0, (sp)
            sw t1, 4(sp)
            jal disp              #display number
            lw t0, (sp)           #reload counter
            lw t1, 4(sp)
            addi sp, sp, 8        #move back stack pointer
            addi t0, t0, 1
            addi t2, zero, 7      #value to check if we are done displaying number
            bltu t0, t2, displp

        ##########adding blankspace#################################
        blnklp: beqz t1, endcal   #put as many blank spaces are needed to get to the end
            li a0, disp_blank     #of the line
            addi sp, sp, -4       #save t1
            sw t1, (sp)
            jal disp
            lw t1, (sp)           #load t1
            addi sp, sp, 4
            addi t1, t1, -1
            j blnklp

endcal: lw ra, (sp)           #reload ra
        addi sp, sp, 4
        lui s2, 0             #reset input count
        lui s3, 0             #reset number input count
        lui s4, 0             #reset operand count
        jr ra
```

```
        ############calculation subroutines####################
subt:   sub a1, a0, a1          #do subtraction and send to arguemnt
        bgez a1, calcret        #check if result is postive
        li a0, disp_sub         #display negative sign
        addi sp, sp, -8         #save ra, a1
        sw ra, (sp)
        sw a1, 4(sp)
        jal disp
        lw ra, (sp)             #reload ra, a1
        lw a1, 4(sp)
        addi sp, sp, 8
        neg a1, a1              #negate result of subtraction
        j calcret

#########################################################################################
#Algorithm for multiply adapted from                                                    #
#https://www.techiedelight.com/multiply-two-numbers-without-using-multiplication-operator-loops/#
#########################################################################################
multi:  addi t0, a0, 0                  #load in first number
        addi t1, a1, 0                  #load in second number
        lui a1, 0                       #reset result
        multlp:beqz t1, calcret
               andi t2, t1, 1
               beqz t2, mskip
               add a1, a1, t0
        mskip: slli t0, t0, 1
               srli t1, t1, 1
               j multlp

divid:  addi t0, a0, 0                  #load in dividend
        addi t1, a1, 0                  #load in divisor
        lui a1, 0                       #reset quotient
        beqz t1, calcret                #checking divide by zero
        divilp:sub t0, t0, t1           #sub t1 from t0
               bltz t0, calcret         #if the subtractions produces a number less than 0
               addi a1, a1, 1           #increment quotient
               j divilp
```

```
#########################display function#####################################
disp:   addi t0, a0, 0                      #load in address from argument
        lui t2, 0                           #initialize row counter for display output
        addi t3, zero, 8                    #initialize max amount for column
        addi t4, zero, 16                   #initialize max amount for row
        rowlp: lui t1, 0                    #initialize column counter
               add t5, t0, t2               #get address of current row
               lbu t6, (t5)                 #load in data from that row
               collp: add t5, s5, t1        #add column counter to horizontal position
                      addi a0, t5, 0        #send column and row to arguments
                      addi a1, t2, 0
                      andi t5, t6, 0x80     #use bitmask to get value at front of byte
                      li a3, disp_BGCOL     #set dot color to background
                      beqz t5, dspskip      #if read value is 1, change color to black
                      li a3, disp_BLACK
               dspskip:addi t1, t1,  1      #increment column counter
                      addi sp, sp, -12      #store ra, t0, t1
                      sw ra,  (sp)
                      sw t0, 4(sp)
                      sw t1, 8(sp)
                      jal draw_dot          #draw dot at location
                      lw ra,  (sp)          #reload ra, t0, t1
                      lw t0, 4(sp)
                      lw t1, 8(sp)
                      addi sp, sp, 12
                      slli t6, t6, 1        #shift data to the left to move right along the data
                      blt t1, t3, collp
               addi t2, t2, 1
               blt t2, t4, rowlp
        addi s5, s5, 8                      #move horizontal position
        jr ra

#######################clear screen subroutine#####################################
clrscn: lui t1, 0                           #intialize row counter
        addi t2, zero, 80                   #intialize max column
        addi t3, zero, 16                   #intialize max row
        crowlp: lui t0, 0                   #initialize column counter
               ccollp: addi a0, t0, 0       #set arguments as row and column counter
                      addi a1, t1, 0
```

```
                        li a3, disp_BGCOL
                        addi sp, sp, -12        #save ra, t0, t1
                        sw ra, (sp)
                        sw t0, 4(sp)
                        sw t1, 8(sp)
                        jal draw_dot            #draw dot of background color
                        lw ra, (sp)
                        lw t0, 4(sp)
                        lw t1, 8(sp)
                        addi sp, sp, 12         #load ra, t0, t1
                        addi t0, t0, 1          #increment column
                        bne t0, t2, ccollp      #check if we are dont printing to the row
                addi t1, t1, 1                  #increment row counter
                bne t1, t3, crowlp              #check if we are done printing to the area
        addi s5, zero, 1                        #reset horizontal display counter
        jr ra


#CREDIT: J. CALLENES AND PUAL HUMMEL
#draws a dot on the display at the given coordinates:
#       (X,Y) = (a0,a1) with a color stored in a3
#       (col, row) = (a0,a1)
# Modifies (directly or indirectly): t0, t1
draw_dot:
        andi t0,a0,0x7F         # select bottom 7 bits (col)
        andi t1,a1,0x3F         # select bottom 6 bits  (row)
        slli t1,t1,7            #  {a1[5:0],a0[6:0]}
        or t0,t1,t0             # 13-bit address
        sw t0, 0x120(s0)        # write 13 address bits to register
        sw a3, 0x140(s0)        # write color data to frame buffer
        jr ra
```

```
#############################################################################
#Below is a slightly modified version of my BCD code from SW7-2          #
#I have added support for 100,000's place and changed the inputs and outputs to #
#work with my current program. Algorithm is the same                       #
#############################################################################
bcd:    addi s2, zero, 10     #value for checking if we are done with divison
        addi s1, a1, 0        #load in value from argument
        lui s3, 0             #reset internal output
        addi sp, sp -4        #save ra
        sw ra, (sp)
        la ra, write          #put the write lable as the current ra (end of function)
bcdloop:bltu s1, s2, ones     #checking if the number loaded in is less than 10
        addi t0, s1, 0        #set temporary for number loaded in
        lui t1, 0             #reset number of divisions
        chklp: add a0, t0, zero      #send value in to function
               add a1, t1, zero      #send current number of divisions to function
               j div10
        divret:addi t0, a0, 0        #get return qotient
               addi t1, a1, 0        #get new number of divisions
        bge t0, s2, chklp     #checking if we need to divide again
        add a0, t0, zero      #loading division result into function input
        addi s2, zero, 5      #value for checking number of divisions
        beq t1, s2, hundth    #multiply quoteint by 100,000 if numdiv = 5
        addi s2, s2, -1       #t2 = 4
        beq t1, s2, tenth     #multiply quoteint by 10,000 if numdiv = 4
        addi s2, s2, -1       #t2 = 3
        beq t1, s2, thou      #multiply quotient by 1,000 if numdiv = 3
        addi s2, s2, -1       #t2 = 2
        beq t1, s2, hund      #mutliply quotient by 1,000 if numdiv = 2
        addi s2, s2, -1       #t2 = 1
        beq t1, s2, ten       #multiply quotient by 10 if numdiv = 1
        ones:  add s3, s3, s1
               addi a0, s3, 0        #load result into argument return
               lw ra, (sp)
               addi sp, sp, 4
               jr ra
        write: addi t0, a0, 0        #get BCD digit
               addi t1, a1, 0        #get new subtract value
```

```
                add s3, s3, t0          #put BCD in final answer
                sub s1, s1, t1          #subtract from number to get new dividend
                addi s2, zero 10        #reset division checking number
        bgez s1, bcdloop

div10:  add t0, a0, zero        #loading in number to divide
        add t1, a1, zero        #load in number of divisions
        lui t2, 0               #reset t2
        lui t3, 0               #reset t3
        addi t4, zero, 1        #number of shifts
        addi t5, zero, 16       #divide by 10 needs 5 shifts
        addi t6, zero, 4        #change from shifting input to current quotient
        divlp: blt t4, t6, in           #check for what we are shifting
                srl t3, t2, t4          #shift current quotient
                j addchk
        in:     srl t3, t0, t4          #shift input
        addchk: add t2, t3, t2          #add shifted value to quotient
                slli t4, t4, 1          #multiply shift amount by 2
                ble t4, t5, divlp       #check if we have gone over 16 for shift amnt
        srli t2, t2, 3          #final quotient shift
        slli t3, t2, 3          #multiply quotient by 10
        slli t4, t2, 1
        add t3, t3, t4
        sub t0, t0, t3          #remainder (r = x - q*10)
        addi t3, zero, 9
        sgt t3, t0, t3          # (r > 9)
        add a0, t2, t3          # q = q + (r>9)
        addi a1, t1, 1          #increment number of divisions
        j divret

hundth: add t0, a0, zero        #load in quotient
        lui a1, 0               #reset subtract value
        slli t1, t0, 16         #x*10^5 = (x<<16) + (x<<15) + (x<<10) + (x<<9) + (x<<7) + (x<<5)
        add a1, a1, t1
        slli t1, t0, 15
        add a1, a1, t1
        slli t1, t0, 10
        add a1, a1, t1
        slli t1, t0, 9
```

```
        add a1, a1, t1
        slli t1, t0, 7
        add a1, a1, t1
        slli t1, t0, 5
        add a1, a1, t1          #final amount to be subtracted
        slli a0, t0, 20         #shift 100,000's number into 100,000's nibble
        jr ra

tenth:  add t0, a0, zero        #load in quotient
        slli t1, t0, 13         #x*10^4 = (x<<13) + (x<<10) + (x<<9) + (x<<8) + (x<<4)
        slli t3, t0, 10
        add t4, t1, t3          #(x<<13) + (x<<10)
        slli t1, t0, 9
        slli t3, t0, 8
        add t6, t1, t3          #(x<<9) + (x<<8)
        add t6, t4, t6          #adding previous two to release t4
        slli t1, t0, 4
        slli a0, t0, 16         #shift 10,000's number into 10,000's nibble
        add a1, t1, t6          #final amount to subtract from total
        jr ra

thou:   add t0, a0, zero        #load in quotient
        slli t1, t0, 9          #x*10^3 = (x<<9) + (x<<8) + (x<<7) + (x<<6) + (x<<5) + (x<<3)
        slli t3, t0, 8
        add t4, t1, t3          #(x<<9) + (x<<8)
        slli t1, t0, 7
        slli t3, t0, 6
        add t6, t1, t3          #(x<<7) + (x<<6)
        add t6, t4, t6          #adding previous two to release t4
        slli t1, t0, 5
        slli t3, t0, 3
        add t4, t1, t3          #(x<<5) + (x<<3)
        add a1, t4, t6          #final amount to subtract from total
        slli a0, t0, 12         #shift 1,000's nubmer into 1,000's nibble
        jr ra

hund:   add t0, a0, zero        #load in quotient
        slli t1, t0, 6          #x*10^2  = (x<<6) + (x<<5) + (x<<2)
        slli t3, t0, 5
```

```
        add t6, t1, t3          #(x<<6) + (x<<5)
        slli t1, t0, 2
        slli a0, t0, 8          #shift 100's number into 100's nibble
        add a1, t1, t6          #final amount to get subbed from total
        jr ra

ten:    add t0, a0, zero        #load in quotient
        slli t1, t0, 3          #x*10 = (x<<3) + (x<<1)
        slli t3, t0, 1
        slli a0, t0, 4          #shift 10's number into 10's nibble
        add a1, t1, t3          #final amount to get subbed from total
        jr ra

return: jr ra                   #branch for return address return
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: J. Calllenes
//           P. Hummel
//           C. Mattern
//
// Create Date: 01/20/2019 10:36:50 AM
// Module Name: OTTER_Wrapper
// Target Devices: OTTER MCU on Basys3
// Description: OTTER_WRAPPER with Switches, LEDs, and 7-segment display
//
// Revision:
// Revision 0.01 - File Created
// Revision 0.02 - Updated MMIO Addresses, signal names
//////////////////////////////////////////////////////////////////////////////////

module OTTER_Wrapper(
   input CLK,
   input BTNL,
   input BTNC,
   input [15:0] SWITCHES,
   input PS2CLK,
   input PS2DATA,
   output logic [15:0] LEDS,
   output [7:0] CATHODES, VGA_RGB,
   output [3:0] ANODES,
   output VGA_HS,
   output VGA_VS
   );

    // INPUT PORT IDS ////////////////////////////////////////////////////
    // Right now, the only possible inputs are the switches
    // In future labs you can add more MMIO, and you'll have
    // to add constants here for the mux below
    localparam SWITCHES_AD = 32'h11000000;
    localparam VGA_READ_AD = 32'h11000160;
```

```
 // OUTPUT PORT IDS ////////////////////////////////////////////////////
 // In future labs you can add more MMIO
 localparam LEDS_AD      = 32'h11000020;
 localparam SSEG_AD      = 32'h11000040;
 localparam KEYBOARD_AD  = 32'h11000100;
 localparam VGA_ADDR_AD  = 32'h11000120;
 localparam VGA_COLOR_AD = 32'h11000140;

// Signals for connecting OTTER_MCU to OTTER_wrapper ////////////////////
logic clk_50 = 1'b0;

logic [31:0] IOBUS_out, IOBUS_in, IOBUS_addr;
logic s_reset, IOBUS_wr, dbBTNL, s_interrupt, keyboard_int, btn_int;

// Signals for connecting VGA Framebuffer Driver
logic r_vga_we;             // write enable
logic [12:0] r_vga_wa;      // address of framebuffer to read and write
logic [7:0] r_vga_wd;       // pixel color data to write to framebuffer
logic [7:0] r_vga_rd;       // pixel color data read from framebuffer

// Registers for buffering outputs  ////////////////////////////////////
logic [15:0] r_SSEG;
logic [7:0] s_scancode;

// Declare OTTER_CPU ///////////////////////////////////////////////////
OTTER_MCU CPU (.RST(s_reset), .INTR(s_interrupt), .CLK(clk_50),
               .IOBUS_OUT(IOBUS_out), .IOBUS_IN(IOBUS_in),
               .IOBUS_ADDR(IOBUS_addr), .IOBUS_WR(IOBUS_wr));

// Declare Seven Segment Display ///////////////////////////////////////
SevSegDisp SSG_DISP (.DATA_IN(r_SSEG), .CLK(CLK), .MODE(1'b0),
                     .CATHODES(CATHODES), .ANODES(ANODES));

//Declare button debouncer//////////////////////////////////////////////
debounce_one_shot DBOS(.CLK(clk_50), .BTN(BTNL), .DB_BTN(btn_int));

//Declare keyboard driver///////////////////////////////////////////////
KeyboardDriver KEYBD(.CLK(CLK), .PS2DATA(PS2DATA), .PS2CLK(PS2CLK),
```

```
                    .INTRPT(keyboard_int), .SCANCODE(s_scancode));
// Declare VGA Frame Buffer ////////////////////////////////////////////
vga_fb_driver_80x60 VGA(.CLK_50MHz(clk_50), .WA(r_vga_wa), .WD(r_vga_wd),
                        .WE(r_vga_we), .RD(r_vga_rd), .ROUT(VGA_RGB[7:5]),
                        .GOUT(VGA_RGB[4:2]), .BOUT(VGA_RGB[1:0]),
                        .HS(VGA_HS), .VS(VGA_VS));

// Clock Divider to create 50 MHz Clock ////////////////////////////////
always_ff @(posedge CLK) begin
    clk_50 <= ~clk_50;
end

// Connect Signals /////////////////////////////////////////////////////
assign s_reset = BTNC;
assign s_interrupt = keyboard_int | btn_int;

// Connect Board input peripherals (Memory Mapped IO devices) to IOBUS
always_comb begin
     case(IOBUS_addr)
         SWITCHES_AD: IOBUS_in = {16'b0,SWITCHES};
         KEYBOARD_AD: IOBUS_in = {24'b0, s_scancode};
         VGA_READ_AD: IOBUS_in = {24'b0, r_vga_rd};
         default:     IOBUS_in = 32'b0;    // default bus input to 0
     endcase
 end


// Connect Board output peripherals (Memory Mapped IO devices) to IOBUS
 always_ff @ (posedge clk_50) begin
     if(IOBUS_wr)
         case(IOBUS_addr)
             LEDS_AD: LEDS   <= IOBUS_out[15:0];
             SSEG_AD: r_SSEG <= IOBUS_out[15:0];
             VGA_ADDR_AD: r_vga_wa <= IOBUS_out[12:0];
             VGA_COLOR_AD: begin
                     r_vga_wd <= IOBUS_out[7:0];
                     r_vga_we <= 1;
                 end
```

```
            endcase
    end
endmodule
```