Hiện đang hiểu là, chương trình cứ loop mỗi lần 1s, sau mỗi 1s sẽ kiểm tra kí tự nhập và in ra số kí tự nhập. Nếu có interrupt từ bàn phím (lúc này là ENTER nhưng chưa biết locate ở đâu) => sẽ nhảy vào interrupt. Trong interrupt sẽ hiển thị ra những kí tự nhập vào trên MMIO, và in ra số kí tự đúng, cũng như cho phép người dùng lặp lại chương trình hay k

**Bài 3: Kiểm tra tốc độ và độ chính xác khi gõ văn bản**

Sinh viên xử lí bài tập: **Phạm Lê Đức – 20161128**

**Đề bài**: Chương trình sau sẽ đo tốc độ gõ bàn phím và hiển thị kết quả bằng 2 đèn led 7 đoạn. Nguyên tắc:

- Cho một đoạn văn bản mẫu, cố định sẵn trong mã nguồn. Ví dụ “bo mon ky thuat may tinh” - Sử dụng bộ định thời Timer (trong bộ giả lập Digi Lab Sim) để tạo ra khoảng thời gian để đo. Đây là thời gian giữa 2 lần ngắt, chu kì ngắt.

- Trong thời khoảng đó, người dùng nhập các kí tự từ bàn phím. Ví dụ nhập “bo mOn ky 5huat may tinh”. Chương trình cần phải đếm số kí tự đúng (trong ví dụ trên thì người dùng gõ sai chữ O và 5) mà người dùng đã gõ và hiển thị lên các đèn led.

**Cách dùng:**

* Mở MARS, sau đó mở Digital Git Lab, MMIO và connect với MIPS.
* Chạy chương trình
* **Note**: MMIO connection có thể bị lỗi, để khắc phục ta bấm reset để có thể vừa type string vừa hiện ra output luôn !

**Source code**:

.eqv SEVENSEG\_LEFT 0xFFFF0011 # LED LEFT

.eqv SEVENSEG\_RIGHT 0xFFFF0010 # LED RIGHT

.eqv KEY\_CODE 0xFFFF0004 # ASCII code from keyboard, 1 byte

.eqv KEY\_READY 0xFFFF0000 # =1 if has a new keycode

# Auto clear after lw

.eqv DISPLAY\_CODE 0xFFFF000C # ASCII code to show, 1 byte

.eqv DISPLAY\_READY 0xFFFF0008 # =1 if the display has already to do

# Auto clear after sw

.eqv MASK\_CAUSE\_KEYBOARD 0x0000034 # Keyboard Cause

.data

LEDAscii : .byte 63,6,91,79,102,109,125,7,127,111 # this is the decimal representation of numbers from 0 to 9 for 7 segments. 63 is 0 and 111 is 9

storestring : .space 1000 # maximum 25 words to store in this array

stringsource : .asciiz "Bo mon ky thuat may tinh"

numCorrectChar: .asciiz "\n The number of matched characters: "

notification: .asciiz "\n Continue using or quit? "

testing: .asciiz "\n length of the typed string: "

string: .asciiz "\n String: "

startMain: .asciiz "\n Start main. "

startReadingKey: .asciiz "\n Start reading key. "

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# MAIN Procedure

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

.text

li $k0, KEY\_CODE

li $k1, KEY\_READY

li $s0, DISPLAY\_CODE

li $s1, DISPLAY\_READY

MAIN:

li $s4,0 # length of storestring later on

li $t9,10

li $v0, 4

la $a0, startMain

syscall

nop

li $s6, 0 # flag var to turn on / off ask loop

li $t3, 0 # the number of matched characters

li $s7, 0 # count time

LOOP:

nop

WAIT\_FOR\_KEY:

lb $t5, 0($k1) # check if a key is pressed or not

beq $t5, $zero, CHECK # if no key then we check condition

nop

READ\_KEY:

lb $t0, 0($k0) # $t0 = [$k0] = KEY\_CODE. Here we use lb for efficient data storage because ASCII only goes to 127 max, which is 8 bits

SHOW\_KEY:

sb $t0, 0($s0) # show the input key when key ready has a signal 1 on MMIO

la $s5,storestring # the address of our input string is stored in $s5

add $s5,$s5,$s4

sb $t0,0($s5) # store the input key into the string

addi $s4,$s4,1 # i = i + 1 so

nop

CHECK:

addi $s7, $s7, 1 # increase time by one

teqi $s7, 500 # if finish 500ms get into interrupt

SLEEP:

addi $v0,$zero,32 # sleep service, just to get back to loop, used to avoid some unpexpected bugs

li $a0, 5 # $a0 = sleep length, which is 5ms in this case

syscall

nop # WARNING: nop is mandatory here.

j LOOP # Loop

j LOOP

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# INTERRUPT SERVICE

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

.ktext 0x80000180 # this will always be called when it comes to interrupt.

IntSR: #--------------------------------------------------------

# Temporary disable interrupt

#--------------------------------------------------------

dis\_int:

mfc0 $t1, $13 # the cause of interrupt is stored inside $t1

KEYBOARD:

li $t2, MASK\_CAUSE\_KEYBOARD

and $at, $t1, $t2

beq $at, $t2, END

nop

j END\_PROCESS

END: #PRINTING

li $v0,11

li $a0,'\n' # new line

syscall

li $v0, 4

la $a0, testing

syscall

li $v0, 1

add $a0, $s4, $zero

syscall

li $v0, 4

la $a0, string

syscall

li $v0, 4

la $a0, storestring

syscall

li $v0,11

li $a0,'\n' # new line

syscall

nop

li $t1,0 # $t1 - increment index when looping

li $t3,0 # stores the number of correct characters

li $t8,24 # Length of the sourcestring.

beq $s4, 0, PRINT

nop

slt $t7,$s4,$t8 # We compare two lengths: sourcestring and storestring

# We get the smaller length.

bne $t7,1, CHECK\_STRING # If $s4 > $t8 then we use $t8 as the length

nop

add $t8,$0,$s4 # If not we set $s4 as $t8, so we only need to use $t8 as our final length everytime

CHECK\_STRING: # when the original string has a greater length

la $t2,storestring # handle the typed string

add $t2,$t2,$t1

lb $t5,0($t2) # The character index i of storestring will be compared with the character at index i of sourcestring

la $t4,stringsource # handle the source string

add $t4,$t4,$t1

lb $t6,0($t4) # extract the character of source string at index i

bne $t6, $t5, CONTINUE # if they are not similar then we skip increment of corrected characters

nop

addi $t3,$t3,1 # if it is similar then we increase the counting register by one

CONTINUE:

addi $t1,$t1,1 # increase i to continue looping

beq $t1,$t8,PRINT # if i reach the length of the string we are looping then we stop

nop

j CHECK\_STRING # Continue checking if we havent finished

PRINT: li $v0,4

la $a0,numCorrectChar

syscall

li $v0,1

li $a0, 0

add $a0, $a0, $t3

syscall

DISPLAY\_DIGITAL:

li $t9, 10 # set t9 to 10 for certainty that it is 10 to divide

div $t3,$t9 # divide by 10. If the total is >= 100 then it will be wrong

mflo $t8 # quotient at the left LED

la $s2,LEDAscii # store the address of the array containing the values of 7 segment numbers

add $s2,$s2,$t8 # get the correct address of the value we want

lb $a0,0($s2) # get that value

jal SHOW\_7SEG\_LEFT # show it on Digital Lab

#------------------------------------------------------------------------

mfhi $t7 # the right side similar with remainder

la $s2,LEDAscii

add $s2,$s2,$t7

lb $a0,0($s2) # set value for segments

jal SHOW\_7SEG\_RIGHT # show

#------------------------------------------------------------------------

li $s6, 1 # turn on ask loop

beq $s4, 0, END\_PROCESS

li $t3, 0

nop

SMALL\_LOOP:

li $s6, 0 # used as a value to clean buffer

la $s5, storestring

add $s5, $s5, $t3

sb $s6, 0($s5) # set the whole string to 0

add $t3, $t3, 1 # increase index

bne $t3, $s4, SMALL\_LOOP # finish when i = n

nop

li $s6, 1

j END\_PROCESS

nop

SHOW\_7SEG\_LEFT:

li $t0, SEVENSEG\_LEFT # assign port's address

sb $a0, 0($t0) # assign new value

jr $ra

SHOW\_7SEG\_RIGHT:

li $t0, SEVENSEG\_RIGHT # assign port's address

sb $a0, 0($t0) # assign new value

jr $ra

END\_PROCESS: .

mtc0 $zero, $13 # need to clear the cause of interrupt here ($13 = 0)

beq $s6, 1, ASK\_LOOP

nop

# Evaluate the return address of main routine

# epc <= epc + 4

#--------------------------------------------------------

NEXT\_PC:

mfc0 $at, $14 # $at <= Coproc0.$14 = Coproc0.epc

addi $at, $at, 4 # $at = $at + 4 (next instruction)

mtc0 $at, $14 # Coproc0.$14 = Coproc0.epc <= $at

nop

RETURN:

eret # return to the next instruction after interrupt

nop

ASK\_LOOP:

li $v0, 50

la $a0, notification

syscall

beq $a0,0,RESET\_DATA

nop

j EXIT

RESET\_DATA:

li $t3, 0 # reset corrected counter register

li $v0, 32 # change back to sleep service due to strange bug

li $s4, 0 # reset index

li $s6, 0 # reset ask loop

li $s7, 0 # reset

j NEXT\_PC

EXIT:

li $v0, 10

syscall

Phân tích: Để làm được bài này, thì cần ba giai đoạn lớn:

* Giai đoạn 1: Thiết lập dữ liệu cần thiết cho chương trình. Cụ thể, ta cần setup những symbols như SEVENSEG\_LEFT, SEVENSEG\_RIGHT, DISPLAY\_CODE cũng như 1 vài symbols nữa để sử dụng cho Digital Lab cũng như MMIO. Như đã biết SEVENSEG\_LEFT lưu địa chỉ của LED bên trái, và khi ta set giá trị tại địa chỉ này => sẽ hiện lên Digital lab. Bởi vì ta không biết ta cần numbers gì, vậy nên các số từ 0 -> 9 sẽ được lưu vào 1 array.

Ngoài ra, ta cũng cần sử dụng 1 số registers để làm các variables increment cho loop, cũng như 2 registers để chứa length của source string, và typed string.

* Giai đoạn 2: Lắng nghe input keyboard từ người dùng đồng thời đếm thời gian tạo ngắt (interrupt)

Bởi vì Time Counter của MARS có nhiều bugs, ngoài ra việc chạy đồng thời sleep service cho Time Counter (bắt buộc bởi vì nó là bugs, không kiểm soát được) và phải chạy vòng lặp liên tục để lấy input của users là rất khó. Vậy em xin phép được sử dụng 1 phương pháp tương tự, cơ chế chạy hoàn toàn giống nhau, vẫn đếm thời gian nhưng không sử dụng công cụ Time Counter của Digital Lab.

Đầu tiên ta sẽ check tín hiệu input của người dùng ($k1, KEY\_READY), nếu nó bằng 0 tức không có input truyền vào, thì ta jump đến vị trí sử dụng biến đếm thời gian. Ở đây ta coi thời gian như những vòng lặp với giới hạn là số lần lặp của nó. Ở đây interval của nó là 500 lần lặp – 1 khoảng tgian vừa đủ để nhập 1 đoạn string không quá ngắn cũng không quá dài. Vậy khi nhảy đến vị trí sử dụng biến đếm, ta increment biến đếm lên 1 và check, nếu như đã reach 500 thì ta sử dụng ngắt mềm thông qua lệnh teqi để nhảy vào interrupt.

Nếu như bằng 1 thì tức là có input, lúc này ta sẽ đọc input key, sau đó ta show lên MMIO, đồng thời lưu nó vào trong 1 ô của variable string ta đã setup từ đầu. Lưu ý, chỗ này ta sử dụng byte hay lb sb thay vì lw sw. Lý do bởi vì ASCII table chỉ tới 8 bits mà max, ngoài ra ta cần lưu từng character, vậy nên data string của ta phải là .space và dùng hoàn toàn với 1 byte thay vì 4 bytes (1 word). Sau khi lưu, ta tăng biến đếm length của string lên 1 và tiếp tục loop để kiểm tra kí tự đầu vào tiếp theo.

* Giai đoạn 3: xử lí interrupt

Khi đã vào interrupt, ta sẽ check xem có đúng là interrupt do ta vừa call từ trên không (thông qua mfc0 $t1, $13). Con processor 13 sẽ lưu cause, và truyền nó vào trong $t1. Sau đó ta kiểm tra xem cause này có khớp với cause ta muốn (đó chính là MASK\_CAUSE\_KEYBOARD lưu ở 0x0000034).

Nếu đúng thì ta sẽ in 1 số dòng output nhất định cho người dùng (string đã nhập, số kí tự đã nhập). Nếu sai thì sẽ nhảy tới END\_PROCESS luôn và tiếp tục đọc lại.

Sau khi in xong, ta kiểm tra sự giống nhau của 2 strings theo thuật toán sau đây (viết bằng ngôn ngữ C. Thuật toán sẽ không khớp với trình bày ở source code 100%, nhưng ý tưởng và cách thức thực hiện tương đồng):

srcString = lengthString (lengthString đã được fixed từ trước);

desString = countedLength(countedLength đã được incremented ở giai đoạn 2)

int checkString(char \*srcString, char\* desString) {

If (desString == 0) // nếu không nhập gì)

Return -1; // không thực sự return gì ở source code, mà nhảy tới lúc in kết quả luôn

Else {

Int total = 0; // biến lưu tổng những kí tự đúng

Int length = 0;

If (srcString < desString) // ý tưởng là xét length nhỏ hơn.

Length = srcString;

else

Length = desString;

// loop qua 2 string để check sự tương đồng

For (int I = 0; I < length; i++) {

If (srcString[i] == desString[i])

Total = total + 1;

}

}

Return total;

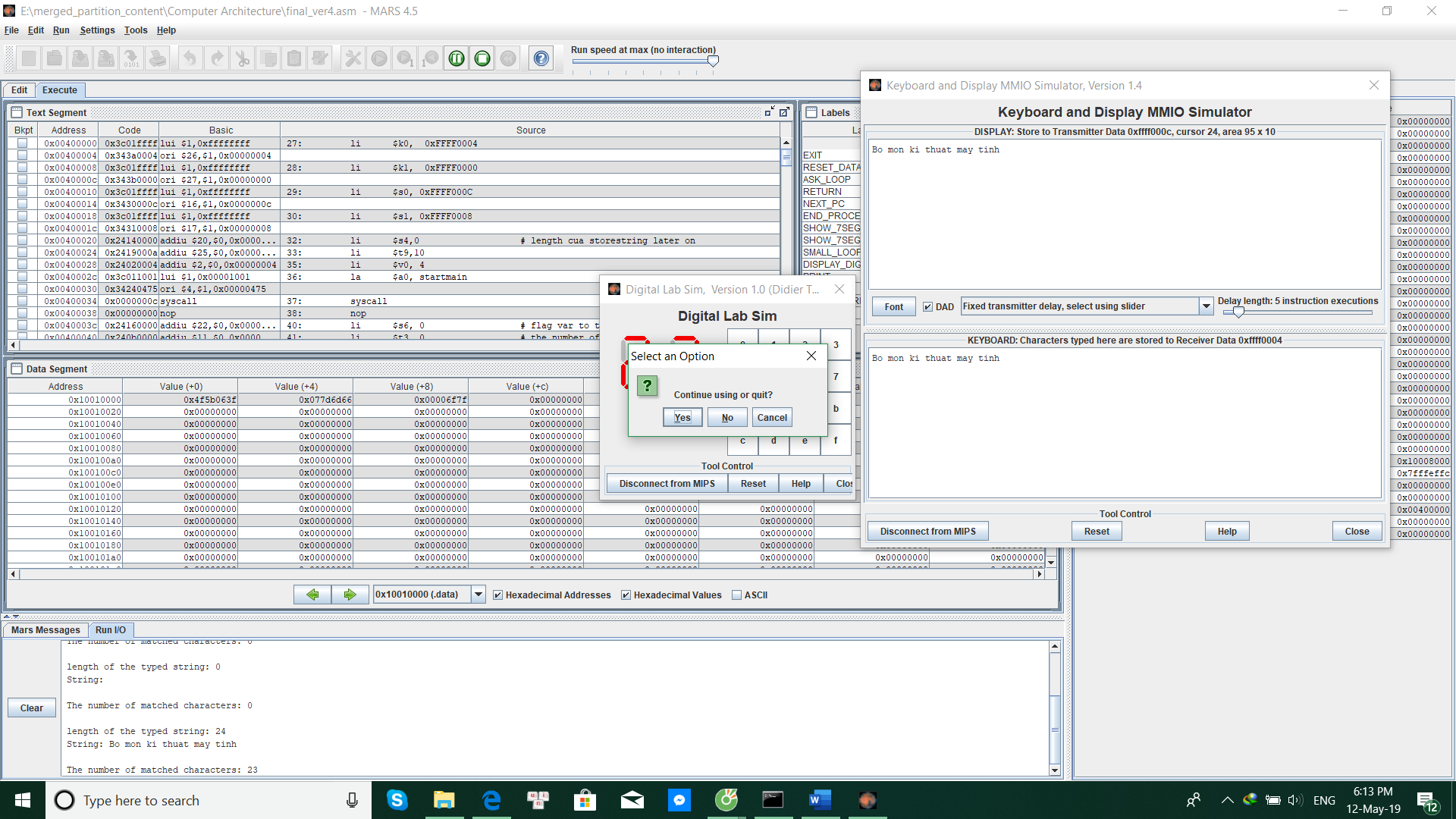
}

Sau khi thu được số lượng characters khớp, ta in ra và sử dụng nó để hiển thị lên Digital Lab.

Cách thức hiển thị: Vì giới hạn Digital Lab chỉ có 2 LED, vậy nên thuật toán này cũng chỉ đúng trong phạm vi từ 0 -> 99, nếu số characters >= 100 sẽ sai. Thuật toán sử dụng biến total / 10, phần nguyên sẽ là phần hiển thị lên đèn LED bên trái, và phần dư sẽ được hiển thị lên đèn LED bên phải (sử dụng mflo và mfhi sau khi chia). Sau đó, ta sử dụng 1 array gồm các số từ 0 đến 9 (lưu dưới dạng bytes or giống với ASCII, chuyển từ binary ra decimal sẽ thu được). Lấy địa chỉ của array này (la $s2,bytehex) sau đó cộng address với phần nguyên (cho LED trái) và phần dư (cho LED phải) để thu được chính xác số mà ta mong muốn. Lý do cộng được bởi vì như giải thích ở trên – array này được lưu dưới dạng byte thay vì word, vậy khi increment từng ô thì ta +1 thay vì +4.

Sau khi hiển thị xong, ta clean buffer (input string) bằng cách loop qua nó và set từng ô kí tự bằng 0. Rồi dùng popup service để hỏi người dùng xem có muốn tiếp tục không. Nếu tiếp tục thì sẽ retrieve instruction được lưu ở Coproc $14 khi interrupt xảy ra. Để tiếp đến instruction tiếp theo thì ta increment lên 4 (addi $at, $at, 4), và sau đó lại truyền giá trị vào $14, và gọi eret (giống như $ra) để nhảy tới địa chỉ ta vừa set – next instruction.

Sau khi quay trở về, ta lại có 1 vòng lặp mới và chương trình cứ thế tiếp diễn. Nếu không tiếp tục thì chương trình sẽ kết thúc.

Ảnh output: 

**Problem 10: Máy tính bỏ túi**

Student: **Nguyễn Đức Anh – 20160125**

**Problem:**

Sử dụng 2 ngoại vi là bàn phím và led 7 thanh để xây dựng một máy tính bỏ túi đơn giản. Hỗ trợ các phép toán +, -, \*, /. Do trên bàn phím không có các phím trên nên sẽ dùng các phím

* Bấm phím a để nhập phép tính +
* Bấm phím b để nhập phép tính –
* Bấm phím c để nhập phép tính \*
* Bấm phím d để nhập phép tính /
* Bấm phím f để nhập phép =

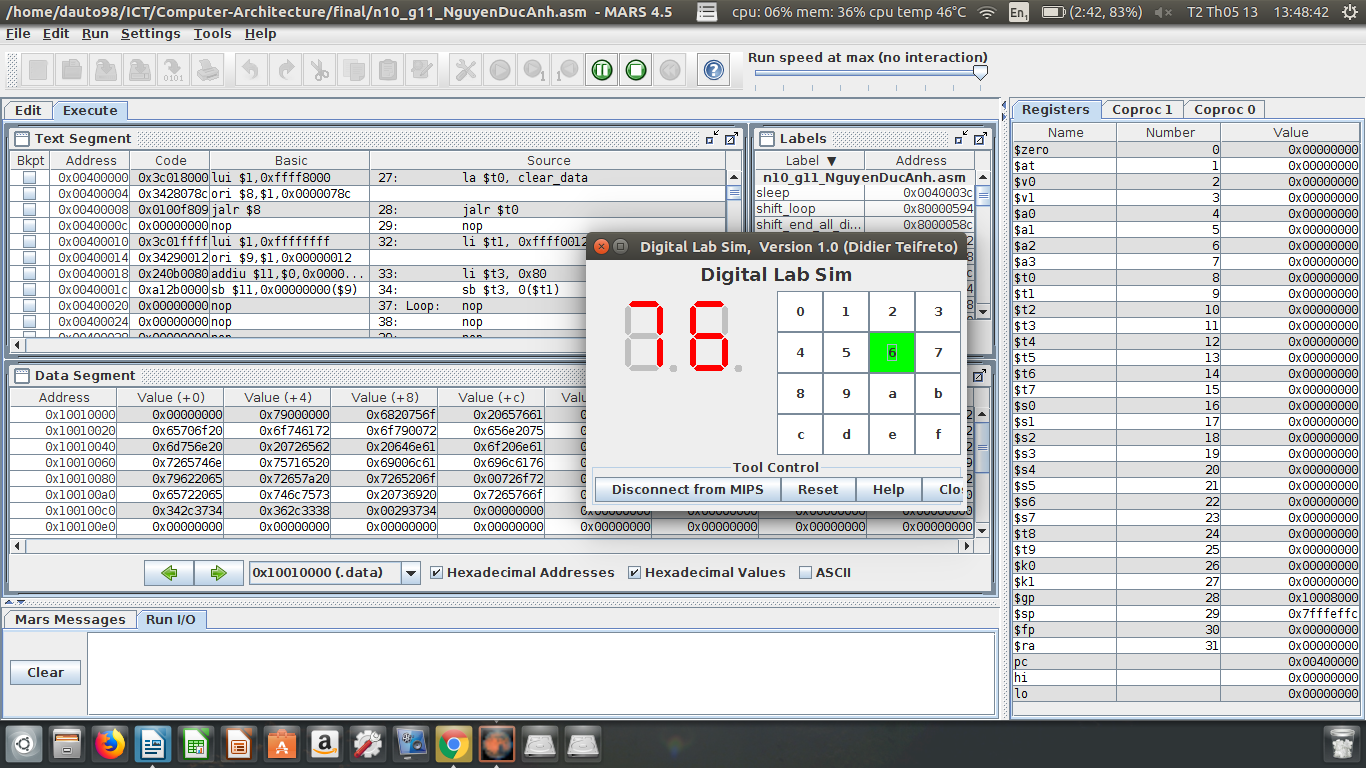
Yêu cầu cụ thể như sau:

* Khi nhấn các phím số, hiển thị lên LED, do chỉ có 2 LED nên chỉ hiện thị 2 số cuối cùng. Ví dụ khi nhấn phím 1 à hiện thị 01. Khi nhấn thêm phím 2 à hiển thị 12. Khi nhấn thêm phím 3 à hiển thị 23.
* Sau khi nhập số, sẽ nhập phép tính + - \* /
* Sau khi nhấn phím f (dấu =) , tính toán và hiển thị kết quả lên LED.

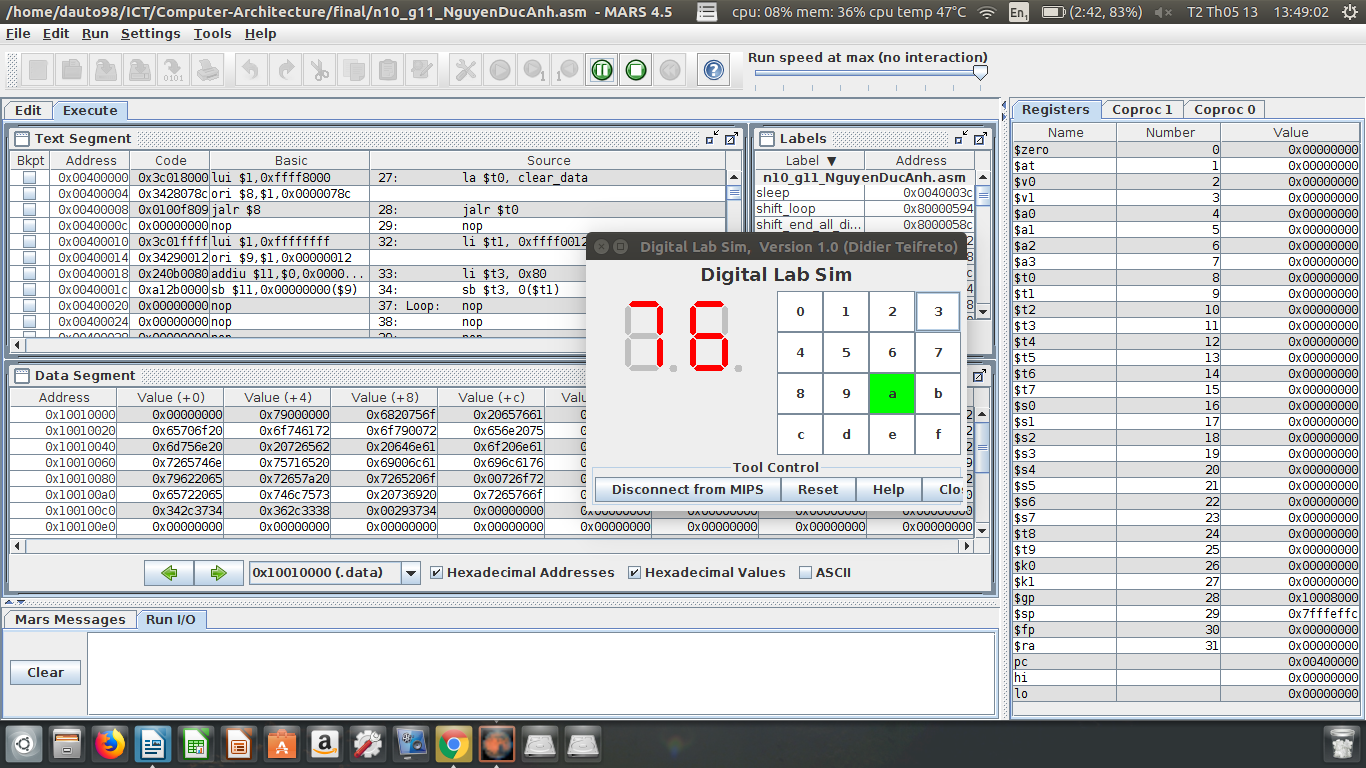
Chú ý: Do bài toán sẽ có rất nhiều trường hợp xảy ra, yêu cầu cơ bản là thực hiện được phép tính và hiển thị lên LED. Các yêu cầu về bắt lỗi, các trường hợp tràn số, … là mở rộng, không bắt buộc.

**How to use:**

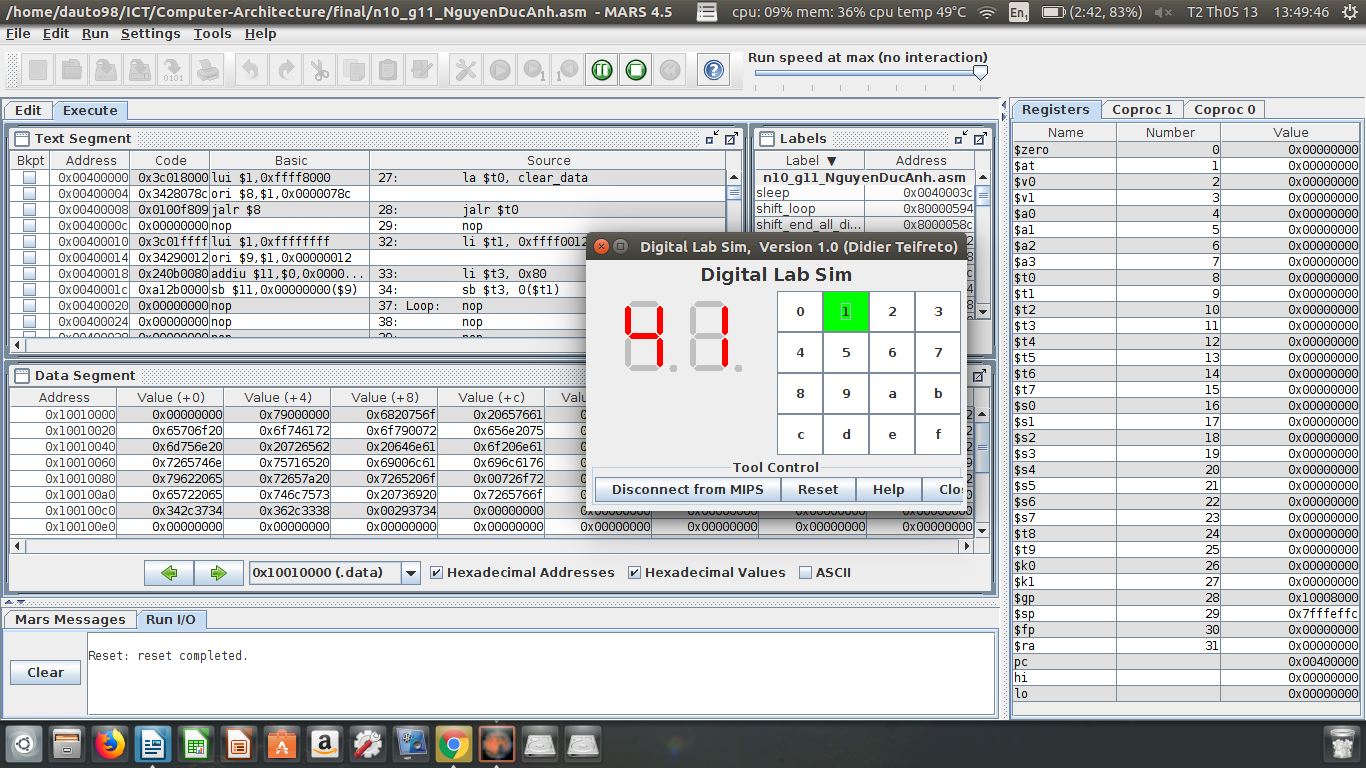
* open digital lab sim tool and run the program
* click the numbers in hexa keyboard and the corresponding number will appear on the seven segment display

  
Image 1: user click 7, then 6

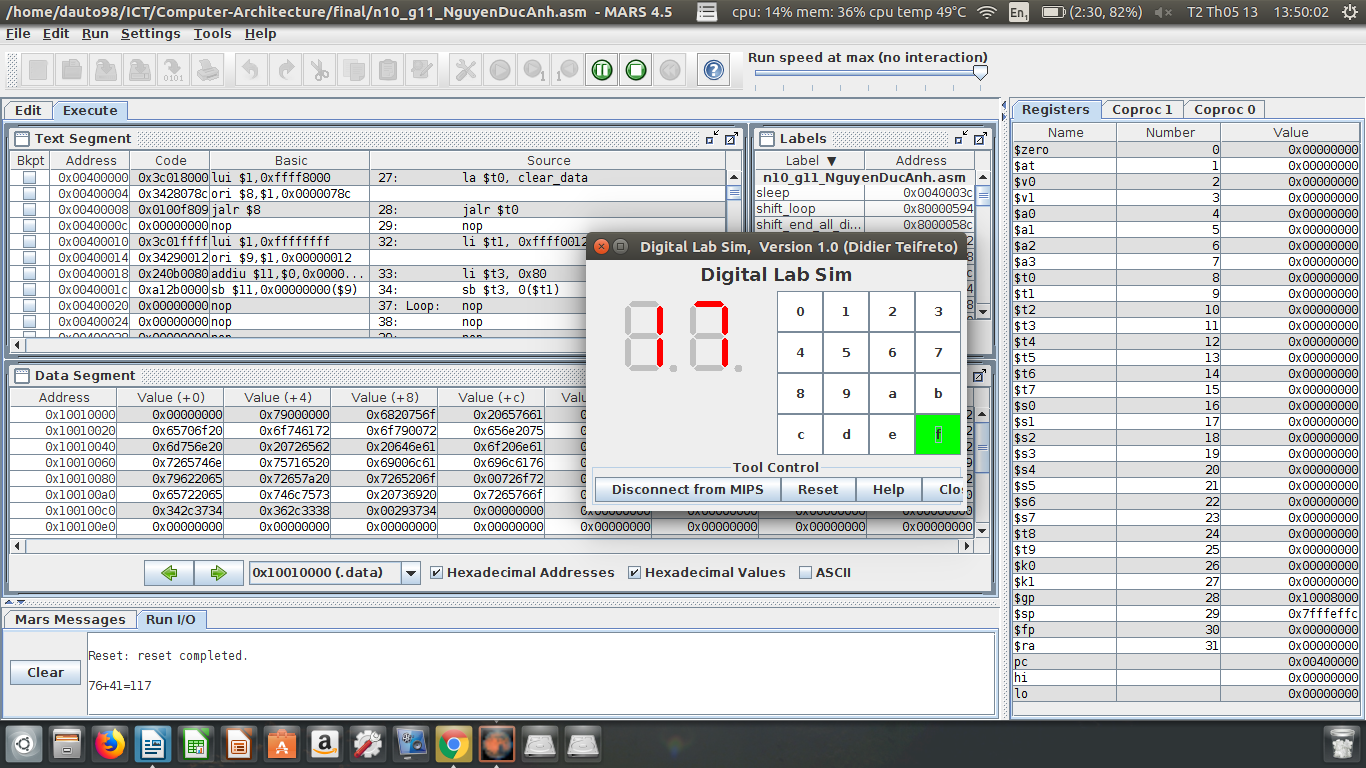
* choose an operator (see the problem description for what key is equal to what operator)

  
Image 2: User choose a (addition)

* choose the number again, if the user doesn’t click a number key, then the number will be default to zero

  
Image 3: User click 4, then 1

* click f to compute the expression, the last 2 digit of the result will be display on the seven segment display. The whole equation is logged to the console

  
Image 4: User click f, the whole equation (76+41=117) is logged at the bottom left of the image, the last 2 digits (17) is displayed at the seven segments

Notes:

* the user cannot choose more than 1 operator for the expression, if not an error message will appear
* the user need to provide both numbers and an operator before click f (=), if not an error message will appear
* There is a constant NUMBER\_LENGTH defined at the start of the file, which specify how many digits of a number will be stored. Although only the last 2 digits will appear on the seven segments display, the rest is kept in the memory. When the number of digits exceed the NUMBER\_LENGTH, the oldest one get discarded to make place for the newest one
* User can change NUMBER\_LENGTH value, but it should not more than 10, since the maximum value can be stored in a register is 0x7fffffff = 2,147,483,647 in decimal, which has 10 digits

**How the program work:**

The main program is simply an infinite loop. A value, 0x80, is stored to the IN\_ADDRESS\_HEXA\_KEYBOARD to enable interrupt on each key press.

When a key is pressed, the program jump to the .ktext section. Here there are 2 subroutines to handle 2 exception: overflow exception and the keyboard matrix interrupt. There program first determine what exception is it by using MASK and branch instruction.

* Overflow\_exception: just display an error message and reset the program
* Keyboard interruption: the main part of the program, the subroutine go through the following step

1. Clear data if there is the previous calculation is finished
2. Scan the keyboard and get the code of the pressed keyboard
3. Convert the keyboard code to number and seven segment code
4. Check whether it is a number or an operator
   1. If it is a number:

* Push the number to the memory
* Display it at the seven segments display
* Exit the handler

4.2. If it is an operator:

* If it is not an “equal”, then put it in the memory, change the current\_number flag to 2 (indicate now the next number go in will be the second operand), then end the exception
* If it is an “equal”, proceed to next part

When the program receive “equal”, it go through the following step:

1. Get 2 operands from the memory and convert them to number and stored in registers (so both operands must be able to represent in 32 bit second complement form, if it fail to do so, an overflow exception is raised)
2. Compute the expression. Overflow may occur when the result cannot be represent in 32 bit. Divide by zero exception may also happend. At this point the program log the equation to the console so the user can see the whole expression
3. Get the last 2 digits of the result in decimal form, display it at the seven segments display
4. Set a flag so that on the next interruption, the handler will clear the display and all data of the previous computation

**Source code:**

.eqv IN\_ADDRESS\_HEXA\_KEYBOARD 0xffff0012

.eqv OUT\_ADDRESS\_HEXA\_KEYBOARD 0xffff0014

.eqv SEVENSEG\_LEFT 0xFFFF0011

.eqv SEVENSEG\_RIGHT 0xFFFF0010

.eqv NUMBER\_LENGTH 2 #number of digits stored before discarding oldest one for the newest one

.eqv MASK\_CAUSE\_KEYMATRIX 0x00000800

.eqv MASK\_OVERFLOW 0x00000030

.data

First\_Number: .space NUMBER\_LENGTH

Second\_Number: .space NUMBER\_LENGTH

current\_number: .space 1

operator: .space 1

clear\_before\_push: .space 1 # = 0 no effect, = 1 clear both segment, = 2 clear all data

Message\_multi\_operator: .asciiz "you have already enter an operator"

Message\_equal\_before\_finish: .asciiz "you need to enter both number and an operator before enter equal"

Message\_invalid\_operator: .asciiz "ivalid operator"

Message\_divide\_by\_zero: .asciiz "divide by zero error"

Message\_overflow: .asciiz "an operand or the result is overflow (max is 2,147,483,647)"

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# MAIN Procedure

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

.text

main:

# init data and clear old input

la $t0, clear\_data

jalr $t0

nop

# Enable the interrupt of Keyboard matrix 4x4 of Digital Lab Sim

li $t1, IN\_ADDRESS\_HEXA\_KEYBOARD

li $t3, 0x80

sb $t3, 0($t1)

# Loop forever waiting for keyboard input

Loop: nop

nop

nop

nop

nop

nop

nop

sleep: addi $v0, $0, 32 # v0 = 32

li $a0, 300 # a0 = 300

syscall # syscall 32: sleep

nop

b Loop # infinite loop

nop

end\_main:

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# GENERAL INTERRUPT

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

.ktext 0x80000180

# find out what kind of exception it is

mfc0 $t0, $13

li $t1, MASK\_OVERFLOW

and $t2, $t0, $t1

beq $t2, $t1, overflow\_exception

nop

li $t1, MASK\_CAUSE\_KEYMATRIX

and $t2, $t0, $t1

beq $t2, $t1, Keymatrix\_Intr

nop

# end exception handler

end\_exception:

# return to start of the loop instead of where the interrupt occur, since the loop doesn't do meaningful thing

la $s3, Loop

mtc0 $s3, $14

eret

overflow\_exception:

li $v0, 55

la $a0, Message\_overflow

li $a1, 0

syscall

jal clear\_data

nop

j end\_exception

nop

#----------------------------------------------------------------------

# interrupt handler for keymatrix

#---------------------------------------------------------------------

Keymatrix\_Intr:

# SAVE the current REG FILE to stack

# Save $ra, $at, $v0, $a0, $s0, $s1, $s2, $s3

addi $sp, $sp, -28

sw $ra, 24($sp)

sw $v0, 20($sp)

sw $a0, 16($sp)

sw $s0, 12($sp)

sw $s1, 8($sp)

sw $s2, 4($sp)

sw $s3, 0($sp)

#--------------------------------------------------------

# Processing

li $s0, IN\_ADDRESS\_HEXA\_KEYBOARD

li $s1, OUT\_ADDRESS\_HEXA\_KEYBOARD

# if the seven segments still display the previous expression

lb $t0, clear\_before\_push

bne $t0, 2, get\_code

nop

sb $0, clear\_before\_push

jal clear\_data

nop

#----------------

# scan keyboard

get\_code:

# scan row 1

li $s2, 0x81

sb $s2, 0($s0)

lbu $a0, 0($s1)

bne $a0, $0, process\_input

nop

# scan row 2

li $s2, 0x82

sb $s2, 0($s0)

lbu $a0, 0($s1)

bne $a0, $0, process\_input

nop

# scan row 3

li $s2, 0x84

sb $s2, 0($s0)

lbu $a0, 0($s1)

bne $a0, $0, process\_input

nop

# scan row 4

li $s2, 0x88

sb $s2, 0($s0)

lbu $a0, 0($s1)

bne $a0, $0, process\_input

nop

li $s2, 0x80

sb $s2, 0($s0)

#---------------

process\_input:

jal convert\_to\_number

nop

bge $v0, 0xa, process\_operator

nop

# process number digit

move $a0, $v0

lbu $a1, current\_number

move $s2, $v1

jal push\_to\_memory

nop

move $a0, $s2

lbu $a1, clear\_before\_push

jal push\_to\_seven\_seg

nop

j finish\_process

nop

process\_operator:

# check e, e button does nothing

beq $v0, 0x0e, finish\_process

nop

beq $v0, 0x0f, process\_equal

nop

lbu $s2, operator

# if user already choose an operator, then throw an error message

beq $s2, $0, process\_operator\_\_skip\_error

nop

li $v0, 55

la $a0, Message\_multi\_operator

li $a1, 0

syscall

j finish\_process

nop

process\_operator\_\_skip\_error:

sb $v0, operator

li $t1, 2

sb $t1, current\_number

li $t1, 1

sb $t1, clear\_before\_push

j finish\_process

nop

process\_equal:

lbu $t1, current\_number

# throw error mes if user click equal before entering 2 number and an operator

bgt $t1, 1, process\_equal\_\_skip\_error

nop

li $v0, 55

la $a0, Message\_equal\_before\_finish

li $a1, 0

syscall

j finish\_process

nop

process\_equal\_\_skip\_error: # Convert 2 number into correct form then calculating expression

la $a0, First\_Number

li $a1, NUMBER\_LENGTH

jal get\_number\_from\_bcd

nop

move $s2, $v0

la $a0, Second\_Number

jal get\_number\_from\_bcd

nop

move $a0, $s2

move $a1, $v0

lb $a2, operator

jal compute\_expression

nop

move $a0, $v0

abs $a0, $a0

li $a1, NUMBER\_LENGTH

jal get\_last\_2\_digit\_in\_seven\_seg\_code

nop

move $s3, $v0

move $a0, $v1

li $a1, 0

jal push\_to\_seven\_seg

nop

move $a0, $s3

li $a1, 0

jal push\_to\_seven\_seg

nop

li $t6, 2

sb $t6, clear\_before\_push

## RESTORE the REG FILE from STACK

finish\_process:

lw $ra, 24($sp)

lw $v0, 20($sp)

lw $a0, 16($sp)

lw $s0, 12($sp)

lw $s1, 8($sp)

lw $s2, 4($sp)

lw $s3, 0($sp)

addi $sp, $sp, 28

## return

j end\_exception

nop

#------------------------------------------------------------------------

# Convert digital lab sim cell coordinate into number and seven seg code

# $a0: digital lab sim cell coordinate

# $v0: number

# $v1: seven seg code, or 0 if it is an operator

#------------------------------------------------------------------------

convert\_to\_number:

li $v0, 0

li $v1, 0x3f

beq $a0, 0x11, convert\_to\_number\_exit

nop

li $v0, 1

li $v1, 0x06

beq $a0, 0x21, convert\_to\_number\_exit

nop

li $v0, 2

li $v1, 0x5b

beq $a0, 0x41, convert\_to\_number\_exit

nop

li $v0, 3

li $v1, 0x4f

beq $a0, 0x81, convert\_to\_number\_exit

nop

li $v0, 4

li $v1, 0x66

beq $a0, 0x12, convert\_to\_number\_exit

nop

li $v0, 5

li $v1, 0x6d

beq $a0, 0x22, convert\_to\_number\_exit

nop

li $v0, 6

li $v1, 0x7d

beq $a0, 0x42, convert\_to\_number\_exit

nop

li $v0, 7

li $v1, 0x07

beq $a0, 0x82, convert\_to\_number\_exit

nop

li $v0, 8

li $v1, 0x7f

beq $a0, 0x14, convert\_to\_number\_exit

nop

li $v0, 9

li $v1, 0x6f

beq $a0, 0x24, convert\_to\_number\_exit

nop

li $v0, 0x0a

li $v1, 0

beq $a0, 0x44, convert\_to\_number\_exit

nop

li $v0, 0x0b

li $v1, 0

beq $a0, 0x84, convert\_to\_number\_exit

nop

li $v0, 0x0c

li $v1, 0

beq $a0, 0x18, convert\_to\_number\_exit

nop

li $v0, 0x0d

li $v1, 0

beq $a0, 0x28, convert\_to\_number\_exit

nop

li $v0, 0x0e

li $v1, 0

beq $a0, 0x48, convert\_to\_number\_exit

nop

li $v0, 0x0f

li $v1, 0

beq $a0, 0x88, convert\_to\_number\_exit

nop

convert\_to\_number\_exit:

jr $ra

nop

#------------------------------------------------------------------------

# push number digit to space in data segment

# $a0: number digit

# $a1: number 1 or 2, indicate current input operand

# affected register: t0, t1, t2, t3, t4

#------------------------------------------------------------------------

push\_to\_memory:

# get the current input number

beq $a1, 2, load\_second\_space

nop

la $t0, First\_Number

j shift\_end\_all\_digit

nop

load\_second\_space:

la $t0, Second\_Number

# shift to the end all digit in the array space, creating new place at the start

# discard last digit

shift\_end\_all\_digit:

li $t1, NUMBER\_LENGTH

addi $t1, $t1, -1

shift\_loop:

add $t2, $t0, $t1 # t2 = add[t1]

addi $t3, $t1, -1 # t3 = t1 - 1

add $t3, $t0, $t3 # t3 = add[t3]

lb $t4, ($t3) # t4 = number[t1 - 1]

sb $t4, ($t2) # number[t1] = t4

addi $t1, $t1, -1

bne $t1, 0, shift\_loop # loop if $t1 > 0

nop

# store the new digit in a0 to the start of the array space

sb $a0, ($t0) # number[0] = a0

jr $ra

nop

#------------------------------------------------------------------------

# display new digit in seven seg display, push old one to the left

# $a0: number digit in seven seg code

# $a1: = 1 will clear the 2 segments before push new digit, = 0 will have no effect

# affected register:

#------------------------------------------------------------------------

push\_to\_seven\_seg:

li $t0, SEVENSEG\_LEFT

li $t1, SEVENSEG\_RIGHT

bne $a1, $0, push\_to\_seven\_seg\_\_clear

nop

lb $t2, ($t1)

sb $t2, ($t0) # move digit in right display to left display

j push\_to\_seven\_seg\_\_next

nop

push\_to\_seven\_seg\_\_clear:

sb $0, ($t0)

sb $0, clear\_before\_push

push\_to\_seven\_seg\_\_next:

sb $a0, ($t1)

jr $ra

nop

#------------------------------------------------------------------------------------------

# convert array in data segment into number, overflow if it could not fit into one register

# $a0: the starting address

# $a1: the length of the segment

# $v0: the number

#------------------------------------------------------------------------------------------

get\_number\_from\_bcd:

addi $t0, $0, 0

addi $t1, $0, NUMBER\_LENGTH

addi $t1, $t1, -1

addi $t5, $0, -1

get\_number\_from\_bcd\_\_loop:

beq $t1, $t5, get\_number\_from\_bcd\_\_exit\_loop

nop

add $t2, $a0, $t1

lbu $t2, ($t2) # t2 = add[t1]

mul $t3, $t0, 10

# check overflow

blt $t3, 0, overflow\_exception

nop

add $t0, $t3, $t2

addi $t1, $t1, -1

j get\_number\_from\_bcd\_\_loop

nop

get\_number\_from\_bcd\_\_exit\_loop:

move $v0, $t0

jr $ra

nop

#------------------------------------------------------------------------------------------

# compute expression

# $a0: the 1st number

# $a1: the 2nd number

# $a2: the operator

# $v0: the result

# $v1: upper 32 bit for multiplication and remainder for division

# exeption: overflow, divide by 0

#------------------------------------------------------------------------------------------

compute\_expression:

# save register: $ra, $a0

addi $sp, $sp, -4

sw $ra, 0($sp)

beq $a2, 0xa, compute\_addition

nop

beq $a2, 0xb, compute\_subtraction

nop

beq $a2, 0xc, compute\_multiplication

nop

beq $a2, 0xd, compute\_division

nop

li $v0, 55

la $a0, Message\_invalid\_operator

li $a1, 0

syscall

j compute\_expression\_\_end

nop

compute\_addition:

add $v0, $a0, $a1

j compute\_expression\_\_end

nop

compute\_subtraction:

sub $v0, $a0, $a1

j compute\_expression\_\_end

nop

compute\_multiplication:

mul $v0, $a0, $a1

mfhi $v1

bne $v1, $0, overflow\_exception

nop

j compute\_expression\_\_end

nop

compute\_division:

bne $a1, $0, divide\_not\_by\_zero

nop

li $v0, 55

la $a0, Message\_divide\_by\_zero

li $a1, 0

syscall

jal clear\_data

nop

lw $ra, 0($sp)

addi, $sp, $sp, 4

j finish\_process

nop

divide\_not\_by\_zero:

div $a0, $a1

mflo $v0

mfhi $v1

compute\_expression\_\_end:

# logging

move $a3, $v0

jal logging\_expression

nop

# restore and return

lw $ra, 0($sp)

addi, $sp, $sp, 4

jr $ra

nop

#------------------------------------------------------------------------------------------

# get the last 2 digits in seven segment code of the input number

# $a0: the input number

# $a1: the number of digit

# $v0: the code of the least significant digit

# $v1: the code of the second least significant digit

#------------------------------------------------------------------------------------------

get\_last\_2\_digit\_in\_seven\_seg\_code:

# save register: $ra, $a0

addi $sp, $sp, -8

sw $ra, 4($sp)

sw $a0, 0($sp)

li $t0, 100

div $a0, $t0

mfhi $t1 # t1 = a0 % 100

li $t0, 10

div $t1, $t0

mflo $a0 # a0 = second least siginificant digit

mfhi $t2 # t2 = least significant digit

jal get\_seven\_seg\_code

nop

move $t1, $v0

move $a0, $t2

jal get\_seven\_seg\_code

nop

move $v1, $t1

# restore and return

lw $ra, 4($sp)

lw $a0, 0($sp)

addi, $sp, $sp, 8

jr $ra

nop

#------------------------------------------------------------------------------------------

# clear all data for the fresh new calculation

#------------------------------------------------------------------------------------------

clear\_data:

la $t0, First\_Number

la $t1, Second\_Number

addi $t2, $0, 0

clear\_data\_\_loop:

beq $t2, NUMBER\_LENGTH, clear\_data\_\_end\_loop

add $t3, $t2, $t0

add $t4, $t2, $t1

sb $0, ($t3)

sb $0, ($t4)

addi $t2, $t2, 1

j clear\_data\_\_loop

nop

clear\_data\_\_end\_loop:

li $t2, 1

sb $t2, current\_number

sb $0, operator

sb $0, clear\_before\_push

li $t1, SEVENSEG\_LEFT

sb $0, 0($t1)

li $t1, SEVENSEG\_RIGHT

sb $0, 0($t1)

jr $ra

nop

#------------------------------------------------------------------------

# Convert number into seven seg code

# $a0: number

# $v0: seven seg code

#------------------------------------------------------------------------

get\_seven\_seg\_code:

li $v0, 0x3f

beq $a0, 0, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x06

beq $a0, 1, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x5b

beq $a0, 2, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x4f

beq $a0, 3, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x66

beq $a0, 4, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x6d

beq $a0, 5, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x7d

beq $a0, 6, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x07

beq $a0, 7, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x7f

beq $a0, 8, get\_seven\_seg\_code\_\_exit

nop

li $v0, 0x6f

beq $a0, 9, get\_seven\_seg\_code\_\_exit

nop

get\_seven\_seg\_code\_\_exit:

jr $ra

nop

#------------------------------------------------------------------------

# Log the expression to console for debugging

# $a0: 1st number

# $a1: 2nd number

# $a2: operator

# $a3: result

#------------------------------------------------------------------------

logging\_expression:

# save register: $ra, $v0

addi $sp, $sp, -8

sw $ra, 4($sp)

sw $v0, 0($sp)

# log expression

move $t0, $a0

move $t1, $a1

move $t2, $a2

move $t3, $a3

li $v0, 1

move $a0, $t0

syscall

li $v0, 11

li $a0, '+'

beq $t2, 0x0a, logging\_cont

nop

li $a0, '-'

beq $t2, 0x0b, logging\_cont

nop

li $a0, '\*'

beq $t2, 0x0c, logging\_cont

nop

li $a0, '/'

beq $t2, 0x0d, logging\_cont

nop

logging\_cont:

syscall

li $v0, 1

move $a0, $t1

syscall

li $v0, 11

li $a0, '='

syscall

li $v0, 1

move $a0, $t3

syscall

li $v0, 11

li $a0, '\n'

syscall

# restore

lw $ra, 4($sp)

lw $v0, 0($sp)

addi $sp, $sp, 8

jr $ra

nop