GROUP 5 REPORT FOR FINAL PROJECT

Student_No1: Tran Quang Hung 20226045

Student_No2: Tran Anh Dung 20226031

# Exercise 13: SIMON GAME

## Table of content



## INTRODUCTION

**The main tasks of this exercise are:**

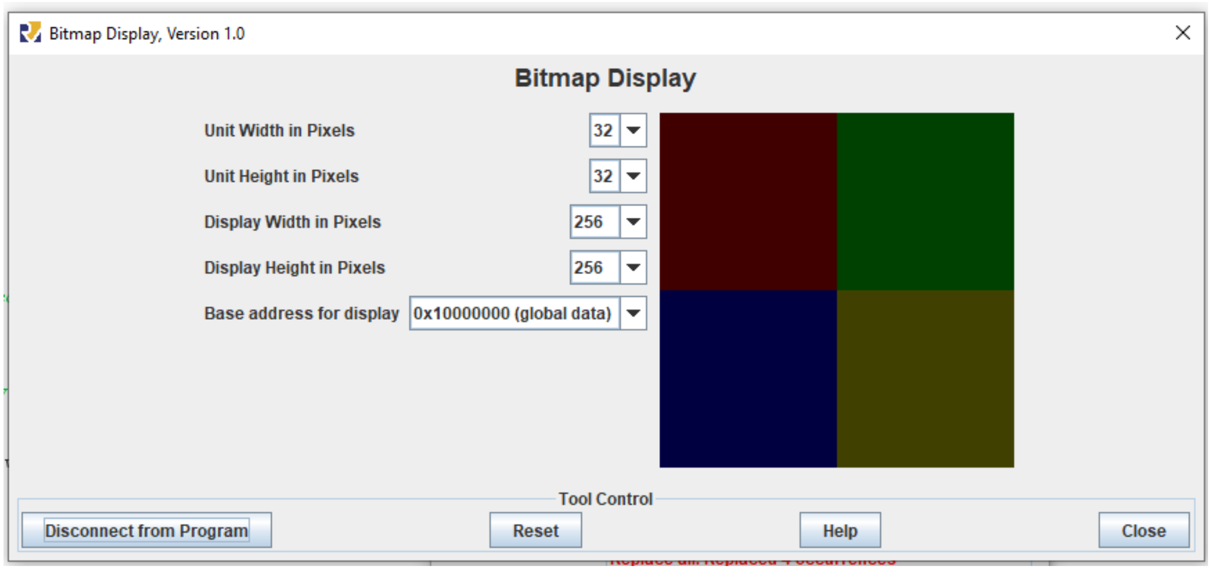
- Display 4 squares with different colors on the bitmap display.
- Able to highlight one square given its ID (1-4).
- Create an algorithm to let the program generate a list of random numbers corresponding to the list of square IDs that is going to appear on the bitmap display in order.
- Scan and check the user input to see if it follows the pattern created by the algorithm or not.

**Bonus:**

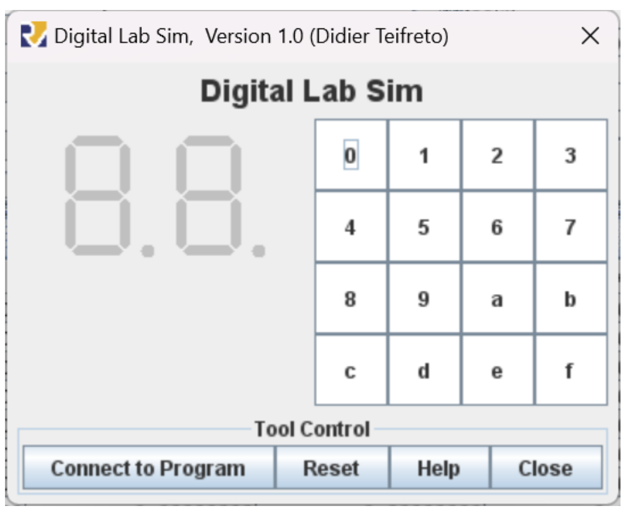
- Implement an interrupt-driven method to scan for user input to decrease load and has an algorithm to prevent multiple interrupt signals from getting called.
- Display the amount of points the user currently has on the bitmap seven segment display (number of rounds completed).

**Configuration:**

- The bitmap display is configured as follow:




- UI for user input:



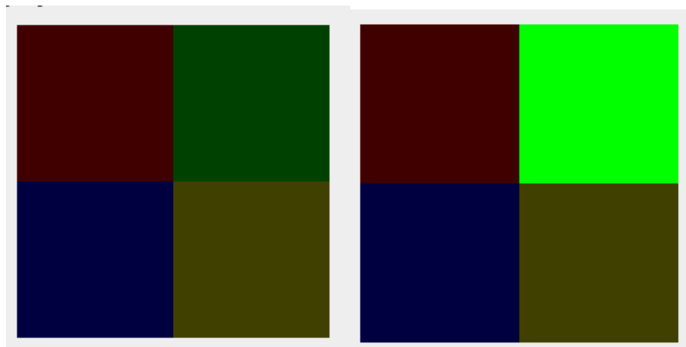

RED: 0 GREEN: 1 BLUE: 2 YELLOW: 3

# MAIN IDEAS

**Color Config**

Offset is used to signify the starting bit position of each color on the bitmap display, combining with pixel_len will create a unique area for each color where each color will start at the offset bits and span the height and width of pixel_len.

Masks are used to control the dimness of each square color, at each step, the color and the mask (semi-transparent dark grayish-blue) will be AND together, decreasing the overall brightness of the square color.

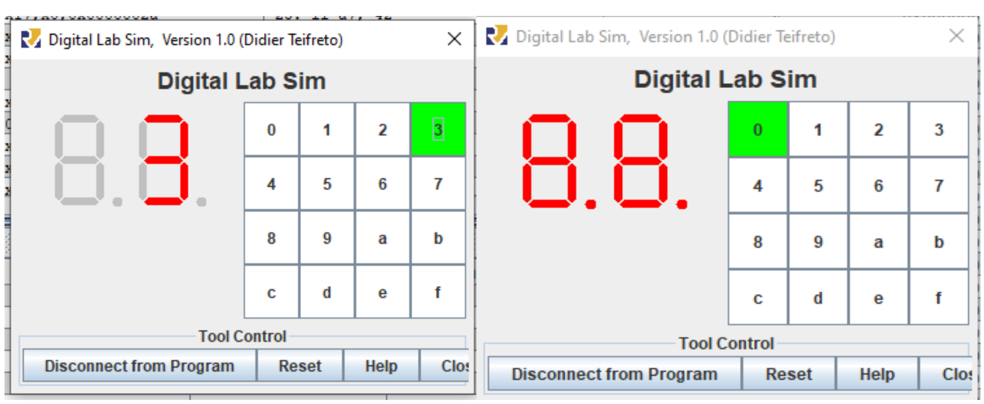


**How does the game run ?**

To implement the algorithm that allows interaction between the program and the user, there will be 2 separate loops running depending on the current state of the program stored in the a4 register.

- Firstly, a4 is set to 0 and the machine_loop will run continuously to show the sequence of highlighted squares to the user. While in this loop, any interrupt signals from the user will be ignored.
- After that, a4 is set to 1 and the human_loop starts, the program will start receiving interrupt signals and process/check user input to see if it similar to the sequences output by the program or not.

After each successful round, the player scores will increase by one (up to maximum of 9) and the program will exit when the input is incorrect, if it is, the number in Lab Sim is 8.8. !



## IMPLEMENTATION

**Initialization**

```
color: .word 0x00FF0000, 0x0000FF00, 0x000000FF, 0x00FFFF0F ## R G B Y
offset: .word 0, 16,128,144
base_add: .word 0x10000000
pixel_len: .word 4
masks: .word 0x00404040, 0x00404040, 0x00404040, 0x00404040
scores: .word 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x67
number_sequence: .word
```

a4: state of the programming (0 or 1 corresponding to machine and human)

s8: address of number sequence (this value will be increased by 4 for each square IDs append to the list)

s9: the base address for the list of number sequences, this value will not change throughout the program.

s7: the current score of the user.

**Machine code section**

*This code block includes the monitor setup and algorithm to light_up a LED*

```
machine_code_section:
##always push a new value to the stack in the machine_code_section
    li a7, 42
    li a0, 0
    li a1, 4
    ecall
    addi a0,a0, 1
    sw a0, 0(s8)
    addi s8, s8, 4
```

```
    addi a5, s9, 0
machine_loop:
### set bit to 0 to prevent human_code_section
    addi a4,zero,0
    beq a5, s8, end_machine_loop
    lw a0, 0(a5)
    jal set_highlight_color
    jal display_all_color
    li a7, 32
    li a0, SPEED
    ecall
    addi a0, a0, 5
    jal set_highlight_color
    jal display_all_color
    li a7, 32
    li a0, SPEED
    ecall
    addi a5,a5, 4
    j machine_loop
end_machine_loop:
    addi a2, s9, 0
    addi a4,zero, 1
    j human_code_section
```

This section is run every time it is the machine turn to display the square sequences on the bitmap display. At each run, a new random square ID will be appended to the list of existing square IDs and the machine_loop will proceed to display the highlighted squares one by one. (There is a delay of around 1s between square IDs)

```
set_highlight_color:
	addi sp, sp, -16
	sw ra, 12(sp)
	sw a0, 8(sp)
	sw s5, 4(sp)
	sw a7, 0(sp)
### RESET hightlight array ( to all be dim )
	lui a7, 0x00404
	addi a7, a7, 0x040
	la s5, masks
	sw a7, 0(s5)
	sw a7, 4(s5)
	sw a7, 8(s5)
	sw a7, 12(s5)
### SET highlight color ( set 1 square to be highlighted )
	addi a7, zero, 0xFFFFFFFF
	addi a0, a0, -1
	slli a0,a0, 2
	la s5, masks
	add s5, s5, a0
	sw a7, 0(s5)
	lw a7, 0(sp)
	lw s5, 4(sp)
	lw a0, 8(sp)
	lw ra, 12(sp)
	addi sp,sp, 16
	jr ra
```

```
display_all_color:
	addi sp, sp, -40
	sw a0, 36(sp)
	sw ra, 32(sp)
	sw s0, 28(sp)
	sw s1, 24(sp)
	sw s2, 20(sp)
	sw s3, 16(sp)
	sw s4, 12(sp)
	sw s5, 8(sp)
	sw t5, 4(sp)
	sw t6, 0(sp)

	la s3, color ### LOAD the color
	la s4, offset ### load the offset
	la s5, masks
	addi t5, zero, 0
	addi t6, zero, 4
```

The set_highlight_color procedure will proceed to set the next highlighted square ( stored in a0, for example, if a0 has the value of 2, the second square will be highlighted)

Following set_highlight_color will be display_all_color procedure, which will load all the color, their offset and their mask (which was modified by set_highlight_color) and proceed to set the corresponding bits in the global data segment for displaying the color for each pixel.

**Human code section**

*This code block handle the input from user*

```
human_code_section:
enable_keyboard_interrupt:
	la s11, handler
	csrrs zero, utvec, s11
	li s10, 0x100
	csrrs zero, uie, s10
	csrrsi zero, ustatus, 0x1
	li s10, IN_ADDRESS_HEXA_KEYBOARD
	li t3, 0x80
	sb t3, 0(s10)
human_inner_loop:
	beqz a4, increase_segment_display
	j human_inner_loop
```

In the human code section, firstly, interrupt will be enabled to receive user input as interrupt.

After that, a loop will continuously run to check the state of the program. If a4 is changed back to 0 (machine state), this means the user input has been correct and the score will be incremented by 1.

For the interrupt handler:

```
handler:
disable_user_interrupt:
	li s10, 0x100
	csrrc zero, uie, s10
	csrrci zero, ustatus, 0x1
```

```
save_context:
        beqz a4, handler_exit
        addi sp, sp, -28
        sw a1, 0(sp)
        sw a7, 4(sp)
        sw t1, 8(sp)
        sw t2, 12(sp)
        sw a0, 16(sp)
        sw t5, 20(sp)
        sw t6, 24(sp)
```

Firstly, interrupt is disabled to prevent the program from getting more interrupts while the handler is running.

After that, the context is saved and restored later.

```
get_user_input:
        li t1, IN_ADDRESS_HEXA_KEYBOARD
        addi t2, zero, 0x81
        sb t2, 0(t1) # Must reassign expected row
        li t1, OUT_ADDRESS_HEXA_KEYBOARD
        lb a1, 0(t1)


        andi a1, a1, 0xF0
        srli a1, a1, 4
        addi a0,zero,0


count_loop:
        beq a1, zero, end_count
        addi a0, a0, 1
```

```
        srli a1, a1, 1
        j count_loop


end_count:
        addi a1, a0, 0
        jal set_highlight_color
        jal display_all_color
        li a7, 32
        li a0, SPEED
        ecall
        addi a0, a0, 5
        jal set_highlight_color
        jal display_all_color
        li a7, 32
        li a0, SPEED
        ecall
        lw t5, 0(a2)
        xor t6, a1, t5
        bnez t6, exit
        addi a2,a2, 4
        beq a2, s8, next_puzzle
```

After that, the user input is read, since we are only reading the first row, so we only concern with the leftmost 4 bits of a1 -> performs an ANDI with 0xF0 and shift right by 4 bits to get the value of the key the user pressed (1,2,4,8 corresponding to keys 0,1,2,3).

After that, continuously shift a1 to the left 1 bit and add a0 by 1 to get the a0 as the final ID of the squares (1-4).

After that, the program will also display the square corresponding with the key the user just pressed.

If the amount of keys read from the user is equal to the amount of squares shown by the program and no mistakes from the user have been detected yet -> the user is correct -> move to the next round.

```
next_puzzle:
        addi a4, zero, 0
        lw a1, 0(sp)
        lw a7, 4(sp)
        lw t1, 8(sp)
        lw t2, 12(sp)
        lw a0, 16(sp)
        lw t5, 20(sp)
        lw t6, 24(sp)
        addi sp, sp, 28

        li s10, 0x100
        csrrs zero, uie, s10

        csrrsi zero, ustatus, 0x1
        li a7, 32
        li a0, SPEED
        ecall
        j increase_segment_display

handler_exit:
        li s10, 0x100
        csrrs zero, uie, s10
```

```
    csrrsi zero, ustatus, 0x1
    uret
```

In the next_puzzle segment, the previous value of the registers are restored and interrupt is enabled again to let the user continue the input.

Finally, if the user input incorrectly, the program will exit and display 88 on the scoreboard.

```
exit:
    addi s7, zero, 0xFF
    li t4, SEVENSEG_RIGHT
    li t5, SEVENSEG_LEFT
    sb s7, 0(t4)
    sb s7, 0(t5)
```

FULL CODE

```
.eqv IN_ADDRESS_HEXA_KEYBOARD 0xFFFF0012
.eqv OUT_ADDRESS_HEXA_KEYBOARD 0xFFFF0014
.eqv SEVENSEG_LEFT 0xFFFF0011
.eqv SEVENSEG_RIGHT 0xFFFF0010
.eqv SPEED 1000
.data
    color: .word 0x00FF0000, 0x0000FF00, 0x000000FF, 0x00FFFF0F ## R G B Y
    offset: .word 0, 16,128,144
    base_add: .word 0x10000000
    pixel_len: .word 4
    masks: .word 0x00404040, 0x00404040, 0x00404040, 0x00404040
    scores: .word 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x67
    number_sequence: .word
.text
```

```
main:
init_variables:
        addi a4, zero, 0 ### STATE OF THE PROGRAM 0 FOR MACHINE AND 1 FOR USER
        la s8, number_sequence ### ADDRESS TO STORE THE SEQUENCE OF COLOR
        la s9, number_sequence ### STARTING ADDRESS OF THE LIST OF SQUARE IDS
        la s7, scores ## USER SCORES


machine_code_section:
##always push a new value to the stack in the machine_code_section
        li a7, 42
        li a0, 0
        li a1, 4
        ecall
        addi a0,a0, 1
        sw a0, 0(s8)
        addi s8, s8, 4
        addi a5, s9, 0
        machine_loop:
### set bit to 0 to prevent human_code_section
        addi a4,zero,0
        beq a5, s8, end_machine_loop
        lw a0, 0(a5)
        jal set_highlight_color
        jal display_all_color
        li a7, 32
        li a0, SPEED
        ecall
        addi a0, a0, 5
        jal set_highlight_color
        jal display_all_color
        li a7, 32
        li a0, SPEED
        ecall
        addi a5,a5, 4
        j machine_loop
```

```
end_machine_loop:
        addi a2, s9, 0
        addi a4,zero, 1
        j human_code_section

human_code_section:
enable_keyboard_interrupt:
        la s11, handler
        csrrs zero, utvec, s11
        li s10, 0x100
        csrrs zero, uie, s10
        csrrsi zero, ustatus, 0x1
        li s10, IN_ADDRESS_HEXA_KEYBOARD
        li t3, 0x80
        sb t3, 0(s10)
human_inner_loop:
        beqz a4, increase_segment_display
        j human_inner_loop

increase_segment_display:
        addi s7,s7,4
        li t4, SEVENSEG_RIGHT
        lw s6, 0(s7)
        sb s6, 0(t4)
        li t4, SEVENSEG_LEFT
        addi s6, zero, 0
        sb s6, 0(t4)
        j machine_code_section

set_highlight_color:
        addi sp, sp, -16
        sw ra, 12(sp)
        sw a0, 8(sp)
        sw s5, 4(sp)
        sw a7, 0(sp)
```

```
### RESET hightlight array ( to all be dim )
        lui a7, 0x00404
        addi a7, a7, 0x040
        la s5, masks
        sw a7, 0(s5)
        sw a7, 4(s5)
        sw a7, 8(s5)
        sw a7, 12(s5)
### SET highlight color ( set 1 square to be highlighted )
        addi a7, zero, 0xFFFFFFFF
        addi a0, a0, -1
        slli a0,a0, 2
        la s5, masks
        add s5, s5, a0
        sw a7, 0(s5)
        lw a7, 0(sp)
        lw s5, 4(sp)
        lw a0, 8(sp)
        lw ra, 12(sp)
        addi sp,sp, 16
        jr ra


display_all_color:
        addi sp, sp, -40
        sw a0, 36(sp)
        sw ra, 32(sp)
        sw s0, 28(sp)
        sw s1, 24(sp)
        sw s2, 20(sp)
        sw s3, 16(sp)
        sw s4, 12(sp)
        sw s5, 8(sp)
        sw t5, 4(sp)
        sw t6, 0(sp)
```

```
        la s3, color ### LOAD the color
        la s4, offset ### load the offset
        la s5, masks
        addi t5, zero, 0
        addi t6, zero, 4
init_loop:
        beq t5, t6, exit_init_loop
        lw s0, 0(s3)
        lw t3, 0(s4)
        lw s1, 0(s5)  ### load the mask ( signify the intensity of color )
        lw s2, base_add
        add s2, s2, t3
        jal display_color
        addi t5, t5, 1
        addi s3,s3, 4
        addi s4, s4, 4
        addi s5,s5,4
        j init_loop

exit_init_loop:
        lw t6, 0(sp)
        lw t5, 4(sp)
        lw s5, 8(sp)
        lw s4, 12(sp)
        lw s3, 16(sp)
        lw s2, 20(sp)
        lw s1, 24(sp)
        lw s0, 28(sp)
        lw ra, 32(sp)
        lw a0, 36(sp)
        addi sp, sp, 40
        jr ra

display_color:
        addi sp, sp, -36
```

```
        sw a0, 32(sp)
        sw s2, 28(sp)
        sw ra, 24(sp)
        sw t3, 20(sp)
        sw t2, 16(sp)
        sw t1, 12(sp)
        sw t0, 8(sp)
        sw s1, 4(sp)
        sw s0, 0(sp)

        and s0,s0,s1 ## appli mask
        addi t2, zero, 0x0  ### loop for column
        addi t3, zero, 0x4

display_column:
        beq t2,t3, end_display_color
        addi t0, zero, 0x0 ## loop for row
        addi t1, zero, 0x4

display_row:
        beq t0, t1, end_display_row
        sw s0, 0(s2)
        addi s2, s2, 0x4
        addi t0,t0, 1
        j display_row

end_display_row:
        addi t2, t2, 0x1
        addi s2, s2, 0x10
        j display_column

end_display_color:
        lw s0, 0(sp)
        lw s1, 4(sp)
        lw t0, 8(sp)
```

```
        lw t1, 12(sp)
        lw t2, 16(sp)
        lw t3, 20(sp)
        lw ra, 24(sp)
        lw s2, 28(sp)
        lw a0, 32(sp)
        addi sp,sp, 36
        jr ra

handler:
disable_user_interrupt:
        li s10, 0x100
        csrrc zero, uie, s10
        csrrci zero, ustatus, 0x1
save_context:
        beqz a4, handler_exit
        addi sp, sp, -28
        sw a1, 0(sp)
        sw a7, 4(sp)
        sw t1, 8(sp)
        sw t2, 12(sp)
        sw a0, 16(sp)
        sw t5, 20(sp)
        sw t6, 24(sp)
get_user_input:
        li t1, IN_ADDRESS_HEXA_KEYBOARD
        addi t2, zero, 0x81
        sb t2, 0(t1) # Must reassign expected row
        li t1, OUT_ADDRESS_HEXA_KEYBOARD
        lb a1, 0(t1)

        andi a1, a1, 0xF0
        srli a1, a1, 4
        addi a0,zero,0
```

```
count_loop:
	beq a1, zero, end_count
	addi a0, a0, 1
	srli a1, a1, 1
	j count_loop

end_count:
	addi a1, a0, 0
	jal set_highlight_color
	jal display_all_color
	li a7, 32
	li a0, SPEED
	ecall
	addi a0, a0, 5
	jal set_highlight_color
	jal display_all_color
	li a7, 32
	li a0, SPEED
	ecall
	lw t5, 0(a2)
	xor t6, a1, t5
	bnez t6, exit
	addi a2,a2, 4
	beq a2, s8, next_puzzle

end_check:
	lw a1, 0(sp)
	lw a7, 4(sp)
	lw t1, 8(sp)
	lw t2, 12(sp)
	lw a0, 16(sp)
	lw t5, 20(sp)
	lw t6, 24(sp)
	addi sp, sp, 28
	j handler_exit
```

```
next_puzzle:
        addi a4, zero, 0
        lw a1, 0(sp)
        lw a7, 4(sp)
        lw t1, 8(sp)
        lw t2, 12(sp)
        lw a0, 16(sp)
        lw t5, 20(sp)
        lw t6, 24(sp)
        addi sp, sp, 28

        li s10, 0x100
        csrrs zero, uie, s10

        csrrsi zero, ustatus, 0x1
        li a7, 32
        li a0, SPEED
        ecall
        j increase_segment_display

handler_exit:
        li s10, 0x100
        csrrs zero, uie, s10
        csrrsi zero, ustatus, 0x1
        uret
exit:
        addi s7, zero, 0xFF
        li t4, SEVENSEG_RIGHT
        li t5, SEVENSEG_LEFT
        sb s7, 0(t4)
        sb s7, 0(t5)
```

# Project 4: Memory Allocation malloc()

## Key Ideas:

- Break the original tasks down into these tasks:
    1. Allocate Char array
    2. Allocate Byte array
    3. Allocate Word array
    4. Assign values to the arrays
    5. Display arrays' values
    6. Display arrays' addresses
    7. Free allocated memory
    8. Check allocated memory size
    9. Allocate a 2D word array
    10. Set *A[i][j]* for the 2D array
    11. Get *A[i][j]* for the 2D array
    12. Copy content from Char Array to Byte Array

## Memory Layout and Data Initialization:

*.data*, we initialize:

- Pointer variables: *CharPtr*, *BytePtr*, *WordPtr*, and *Array2dPtr*, all start at 0 (NULL) because no memory is allocated.

- Counters: *CharCount*, *ByteCount*, *WordCount* track the number of elements allocated in each corresponding array.

- *Sys_TheTopOfFree* and *Sys_MyFreeSpace*: A simple custom "heap" is simulated by reserving 1024 bytes (*.space 1024*). *Sys_MyFreeSpace* marks the start of this heap, and *Sys_TheTopOfFree* tracks the current top of the free memory region.

**Memory Initialization (*SysInitMem*):** - *SysInitMem* sets *Sys_TheTopOfFree* to the start of *Sys_MyFreeSpace*, initializing the heap before any allocations occur.

## Menu and Tasks:

The *.text* section starts by calling *SysInitMem* to prepare the memory system, then enters *main*. In *main*, the user is shown a menu and can select from tasks:

1. **Allocate Char Array (*task1*)**:
   Prompts the user for the number of characters. If valid, calls *malloc* with element size = 1 byte. The allocated address is stored in *CharPtr*. The count is stored in *CharCount*. Prints success message and returns to main menu.

2. **Allocate Byte Array (*task2*)**:
   Similar to *task1* but for a "byte array" (also 1 byte per element). Stores address in *BytePtr* and count in *ByteCount*.

3. **Allocate Word Array (*task3*)**:

   Asks the user for the number of words, uses *malloc* with element size = 4 bytes. Stores address in *WordPtr* and count in *WordCount*.

4. **Assign Values to Array (*task4*)**:
   Prompts user to choose which array type (Char, Byte, or Word). Checks if that array is allocated. If allocated, reads values from the user:

   - For Char/Byte arrays: reads single characters.
   - For Word arrays: reads integers. Store these values directly into the allocated memory. Below is a revised report section to include an explanation about the word alignment fix, as required by the assignment:

**Word Alignment**: When allocating memory for word arrays (where each element is 4 bytes), if the *Sys_TheTopOfFree* pointer is not aligned on a 4-byte boundary, the newly allocated array could start at a misaligned address. This could cause incorrect memory access or simulation errors.

**The Fix:** In the *malloc* function, after obtaining *Sys_TheTopOfFree* and determining that *a2=4* (word-sized allocations), we add *3* and then apply a mask (*0xfffffffc*) using the *and* instruction to ensure the address is properly aligned to a 4-byte boundary. Specifically:

```
li      t0,4
bne     a2,t0,not_word
addi    s10,s10,3
li      t0,0xfffffffc
and     s10,s10,t0
not_word:
```

This sequence ensures that *s10* (the top of free memory pointer) is now rounded down to the nearest 4-byte boundary. So when we store *s10* as the allocated address and update *Sys_TheTopOfFree*, all word arrays begin at addresses divisible by 4.

1. **Display Array Values (*task5*)**:
   Prints values of Char, Byte, and Word arrays if allocated and not empty. If not allocated or empty, prints appropriate messages.

2. **Display Array Addresses (*task6*)**:
   Prints the current addresses stored in *CharPtr*, *BytePtr*, *WordPtr*, and *Array2dPtr*. Also shows the pointer variable addresses themselves for debugging.

3. **Free Allocated Memory (*task7*)**:
   Sets all pointers and counts to zero. Calls *SysInitMem* to reset the memory system. Prints a message indicating memory freed.

4. **Check Allocated Memory (*task8*)**:
   Calls *AllocatedMemory* to find out how many bytes have been allocated from *Sys_MyFreeSpace* and prints the value.

5. **Allocate a 2D Word Array (*task9*)**:
   Prompts for the number of rows and columns, then uses *Malloc2d* to allocate a word-based 2D array. The base address is stored in *Array2dPtr*. Prints success message.

6. **Set A[i][j] (*task10*)**:
   Reads row (i) and column (j), and a value. Calls *Set_func* to store the value into the 2D array at *Array2dPtr*.

7. **Get A[i][j] (*task11*)**:
   Reads row (i) and column (j). Calls *Get_func* to retrieve the value from the 2D array at *Array2dPtr*, then prints it.

8. **Copy Data from Char to Byte Array (*task12*)**:
   Checks if CharPtr and BytePtr are allocated and if BytePtr has enough space. If so, copies the exact number of elements from CharPtr to BytePtr using *copy_char_arrays*. Prints a success message.

## Malloc Implementation:

- The *malloc* function takes a pointer variable address (in *a0*), number of elements (*a1*), and element size (*a2*).

- It aligns addresses if *a2=4* (word alignment). - Allocates *a1*a2* bytes from *Sys_MyFreeSpace*.

- Updates *Sys_TheTopOfFree*.

- Returns the allocated address in *a0* and also stores it into the pointer variable passed in *a0*.

## Malloc2d Function:

The *Malloc2d* function is a specialized allocator for two-dimensional word arrays. It takes three parameters: the pointer variable address for the array (*Array2dPtr*), the number of rows, and the number of columns.

1. It stores the number of rows and columns into memory variables *row* and *col*.
2. It then calculates the total number of elements (*rows * columns*).
3. Since the array consists of words (4 bytes each), it multiplies the number of elements by 4 to determine the total size in bytes.
4. It calls *malloc* with this total size, which returns a base address for the 2D array.
5. This base address is then stored in *Array2dPtr*.

When indexing the 2D array, given indices *i* (row) and *j* (column), the effective element address is computed as:

address = Array2dPtr + ((i * col_count) + j) * 4

This ensures that each element of the 2D word array can be accessed and manipulated using the *Get_func* and *Set_func* functions

## Error Handling:

- If invalid input (negative number), prints an error message and returns to main.

- If pointers are NULL (not allocated), prints “array not allocated” or “2d array not allocated” and returns to main.

- Out-of-bounds accesses in *Get_func* and *Set_func* print a segmentation fault message and return to main.

## Source Code:

```
    .data

# Pointer variables
    CharPtr:
    .word     0
    BytePtr:
    .word     0
    WordPtr:
    .word     0
    Array2dPtr:
    .word     0

    CharCount:
    .word     0
    ByteCount:
    .word     0
    WordCount:
    .word     0

    Sys_TheTopOfFree:
    .word     1
    Sys_MyFreeSpace:
    .space    1024

# Messages
    space:
    .asciz    "  "
    prompt_array_not_allocated:
    .asciz    "array not allocated\n"
    all_assigned:
    .asciz    "All elements assigned.\n"
    prompt_array_menu:
    .asciz    "Choose which array to assign values:\n1. Char Array\n2. Byte Array\n3. Word Array\n"
    enter_char_msg:
    .asciz    "Enter a character: "
    enter_int_msg:
```

```
    .asciz    "Enter an integer: "
    freeMem:
    .asciz    "Memory freed\n"
    message_char:
    .asciz    "Enter the number of characters : "
    message_byte:
    .asciz    "Enter the number of characters for the byte array : "
    message_word:
    .asciz    "Enter the number of words : "
    message_success:
    .asciz    "Memory allocated. The starting address is: "
    message_lessthan0:
    .asciz    "Number of elements cannot be less than 0 \n"
    endl:
    .asciz    "\n"
    address_str:
    .asciz    "\nAddress of CharPtr, BytePtr, WordPtr, 2dArrayPtr are: \n"
    value_str:
    .asciz    "\nValues stored in arrays:\n"
    da_cap_phat:
    .asciz    "Allocated bytes: "
    message_row:
    .asciz    "\nEnter the number of rows: "
    message_col:
    .asciz    "\nEnter the number of columns: "
    row:
    .word     1
    col:
    .word     1
    input_row:
    .asciz    "\nEnter row i (first row i=0): "
    input_col:
    .asciz    "\nEnter column j (first column j=0): "
    input_val:
    .asciz    "\nEnter value for 2d array element : "
    output_val:
    .asciz    "\nReturn value : "
    bound_error:
    .asciz    "\nError: Segmentation fault. Index out of bounds.\n"
    arrMes1:
    .asciz    "A["
    arrMes2:
    .asciz    "] = "
    arrMin:
    .asciz    "Min element in the array: "
    cpyMess:
    .asciz    "Copied from CharPtr to BytePtr.\n"
    no_space_msg:
    .asciz    "no space left to copy\n"

# Messages for Task5
```

```
    char_array_values_msg:
    .asciz    "Char array values: "
    char_not_allocated_msg:
    .asciz    "Char array not allocated\n"
    char_no_elements_msg:
    .asciz    "Char array has no elements\n"

    second_char_array_values_msg:
    .asciz    "Byte array values: "
    second_char_not_allocated_msg:
    .asciz    "Byte array not allocated\n"
    second_char_no_elements_msg:
    .asciz    "Byte array has no elements\n"

    word_array_values_msg:
    .asciz    "Word array values: "
    word_not_allocated_msg:
    .asciz    "Word array not allocated\n"
    word_no_elements_msg:
    .asciz    "Word array has no elements\n"

# Menu options
    menu:
    .asciz    "\n1. Allocate Char.\n2. Allocate Byte.\n3. Allocate word.\n4. Assign values to an array.\n5. Display array values.\n6. Display array addresses.\n7. Free allocated memory.\n8. Check allocated memory.\n9. Allocate a 2D word array.\n10. Set Array[i][j].\n11. Get Array[i][j].\n12. Copy data from CharPtr to BytePtr.\n0. Exit the program"

    .text
    jal       SysInitMem

    .global   main
    main:
    print_menu:
    la        a0, menu
    jal       take_the_int
    mv        s0,a0

    li        t0,1
    beq       s0,t0,task1
    li        t0,2
    beq       s0,t0,task2
    li        t0,3
    beq       s0,t0,task3
    li        t0,4
    beq       s0,t0,task4
    li        t0,5
    beq       s0,t0,task5
    li        t0,6
    beq       s0,t0,task6
    li        t0,7
```

```
    beq     s0,t0,task7
    li      t0,8
    beq     s0,t0,task8
    li      t0,9
    beq     s0,t0,task9
    li      t0,10
    beq     s0,t0,task10
    li      t0,11
    beq     s0,t0,task11
    li      t0,12
    beq     s0,t0,task12
    li      t0,0
    beq     s0,t0,terminated

# task1: allocate char array
    task1:
    la      a0, message_char
    jal     take_the_int
    slt     s10,a0,zero
    beq     s10,zero,done_check_char
    la      a0,message_lessthan0
    li      a7,4
    ecall
    jal     main

    done_check_char:
    mv      a1,a0           # no +1 for null terminator
    la      a0,CharPtr
    li      a2,1
    jal     malloc
    mv      s0,a0
    la      t0,CharCount
    sw      a1,0(t0)

    la      a0, message_success
    li      a7,4
    ecall
    mv      a0,s0
    li      a7,34
    ecall
    la      a0,endl
    li      a7,4
    ecall
    jal     main

# task2: allocate byte array
    task2:
    la      a0, message_byte
    jal     take_the_int
    slt     s10,a0,zero
    beq     s10,zero,done_check_Byte
```

```
    la      a0,message_lessthan0
    li      a7,4
    ecall
    jal     main

    done_check_Byte:
    mv      a1,a0               # no +1 for null terminator
    la      a0,BytePtr
    li      a2,1
    jal     malloc
    mv      s0,a0
    la      t0, ByteCount
    sw      a1, 0(t0)

    la      a0, message_success
    li      a7,4
    ecall
    mv      a0,s0
    li      a7,34
    ecall
    la      a0,endl
    li      a7,4
    ecall
    jal     main

# task3: allocate word array
    task3:
    la      a0,message_word
    jal     take_the_int
    slt     s10,a0,zero
    beq     s10,zero,done_check_Word
    la      a0,message_lessthan0
    li      a7,4
    ecall
    jal     main

    done_check_Word:
    mv      a1,a0
    la      a0,WordPtr
    li      a2,4
    jal     malloc
    mv      s0,a0
    la      t0, WordCount
    sw      a1, 0(t0)

    la      a0,message_success
    li      a7,4
    ecall
    mv      a0,s0
    li      a7,34
    ecall
```

```
    la      a0,endl
    li      a7,4
    ecall
    jal     main

# task4: Assign values to an array (char/byte/word)
    task4:
    li      a7,4
    la      a0, endl
    ecall

    li      a7,4
    la      a0, prompt_array_menu
    ecall

    li      a7,5
    ecall
    mv      t3,a0

    li      t1,1
    beq     t3,t1,assign_char
    li      t1,2
    beq     t3,t1,assign_byte
    li      t1,3
    beq     t3,t1,assign_word

    jal     main

    assign_char:
    la      t0,CharPtr
    lw      t0,0(t0)
    beq     t0,zero,no_alloc
    la      t1,CharCount
    lw      t1,0(t1)
    j       start_assign

    assign_byte:
    la      t0,BytePtr
    lw      t0,0(t0)
    beq     t0,zero,no_alloc
    la      t1,ByteCount
    lw      t1,0(t1)
    j       start_assign

    assign_word:
    la      t0,WordPtr
    lw      t0,0(t0)
    beq     t0,zero,no_alloc
    la      t1,WordCount
    lw      t1,0(t1)
    j       start_assign
```

```
no_alloc:
li       a7,4
la       a0, prompt_array_not_allocated
ecall
jal      main

start_assign:
li       t6,1                 # default: char arrays
li       t5,3
beq      t3,t5,set_word_size
# If not word, keep t6=1 for char/byte arrays
j        assign_loop

set_word_size:
li       t6,4                 # word array

assign_loop:
li       t4,0

assign_loop_start:
beq      t4,t1,assign_done

li       t5,4
beq      t6,t5,is_word

# char/byte arrays read char
li       a7,4
la       a0, enter_char_msg
ecall
li       a7,12
ecall
mv       s1,a0
add      t2,t0,t4
sb       s1,0(t2)
j        next_element

is_word:
# word array read int
li       a7,4
la       a0,enter_int_msg
ecall
li       a7,5
ecall
mv       s1,a0
slli     t2,t4,2
add      t2,t2,t0
sw       s1,0(t2)

next_element:
addi     t4,t4,1
```

```
    j        assign_loop_start

    assign_done:
# No null terminator logic, just done
    li       a7,4
    la       a0, all_assigned
    ecall
    jal      main

# task5: Display array values
    task5:
    la       a0, value_str
    li       a7,4
    ecall

# Print Char array
    la       t0,CharCount
    lw       t2,0(t0)
    la       t0,CharPtr
    lw       t0,0(t0)
    beq      t0,zero,char_not_alloc
    beq      t2,zero,char_no_elem
    li       a7,4
    la       a0,char_array_values_msg
    ecall
    li       t3,0
    char_loop:
    beq      t3,t2,char_done
    add      t4,t0,t3
    lb       a0,0(t4)
    li       a7,11              # print char
    ecall
    li       a7,4
    la       a0,space
    ecall
    addi     t3,t3,1
    j        char_loop

    char_done:
    li       a7,4
    la       a0,endl
    ecall
    j        char_skip

    char_not_alloc:
    li       a7,4
    la       a0,char_not_allocated_msg
    ecall
    j        char_skip

    char_no_elem:
```

```
    li      a7,4
    la      a0,char_no_elements_msg
    ecall

    char_skip:

# Print Byte array (second char array)
    la      t0,ByteCount
    lw      t2,0(t0)
    la      t0,BytePtr
    lw      t0,0(t0)
    beq     t0,zero,second_char_not_alloc
    beq     t2,zero,second_char_no_elem
    li      a7,4
    la      a0,second_char_array_values_msg
    ecall
    li      t3,0
    byte_loop:
    beq     t3,t2,byte_done
    add     t4,t0,t3
    lb      a0,0(t4)
    li      a7,11               # print char
    ecall
    li      a7,4
    la      a0,space
    ecall
    addi    t3,t3,1
    j       byte_loop

    byte_done:
    li      a7,4
    la      a0,endl
    ecall
    j       byte_skip

    second_char_not_alloc:
    li      a7,4
    la      a0,second_char_not_allocated_msg
    ecall
    j       byte_skip

    second_char_no_elem:
    li      a7,4
    la      a0,second_char_no_elements_msg
    ecall

    byte_skip:

# Print Word array
    la      t0,WordCount
    lw      t2,0(t0)
```

```
    la        t0,WordPtr
    lw        t0,0(t0)
    beq       t0,zero,word_not_alloc
    beq       t2,zero,word_no_elem
    li        a7,4
    la        a0,word_array_values_msg
    ecall
    li        t3,0
    word_loop:
    beq       t3,t2,word_done
    slli      t4,t3,2
    add       t4,t4,t0
    lw        a0,0(t4)
    li        a7,1              # print int
    ecall
    li        a7,4
    la        a0,space
    ecall
    addi      t3,t3,1
    j         word_loop

    word_done:
    li        a7,4
    la        a0,endl
    ecall
    j         word_skip

    word_not_alloc:
    li        a7,4
    la        a0,word_not_allocated_msg
    ecall
    j         word_skip

    word_no_elem:
    li        a7,4
    la        a0,word_no_elements_msg
    ecall

    word_skip:
    jal       main

# task6: Display array addresses
    task6:
    la        a0, address_str
    li        a7,4
    ecall

    la        a0,endl
    li        a7,4
    ecall
```

```
# Print the address stored in CharPtr
    la      t0, CharPtr          # t0 = address of CharPtr variable
    lw      a0,0(t0)             # a0 = *CharPtr (the pointer value)
    li      a7,34                # print pointer in hex
    ecall

    la      a0,endl
    li      a7,4
    ecall

# Print the address stored in BytePtr
    la      t0, BytePtr
    lw      a0,0(t0)
    li      a7,34
    ecall

    la      a0,endl
    li      a7,4
    ecall

# Print the address stored in WordPtr
    la      t0, WordPtr
    lw      a0,0(t0)
    li      a7,34
    ecall

    la      a0,endl
    li      a7,4
    ecall

# Print the address stored in Array2dPtr
    la      t0, Array2dPtr
    lw      a0,0(t0)
    li      a7,34
    ecall

    la      a0,endl
    li      a7,4
    ecall

    jal     main

    la      a0, address_str
    li      a7,4
    ecall

    la      a0,endl
    li      a7,4
    ecall

    la      a0,CharPtr
```

```
    li      a7,34
    ecall

    la      a0,endl
    li      a7,4
    ecall

    la      a0,BytePtr
    li      a7,34
    ecall

    la      a0,endl
    li      a7,4
    ecall

    la      a0,WordPtr
    li      a7,34
    ecall

    la      a0,endl
    li      a7,4
    ecall

    la      a0,Array2dPtr
    li      a7,34
    ecall

    la      a0,endl
    li      a7,4
    ecall

    jal     main

# task7: Free all pointers and reset memory
    task7:
# Set all pointers to zero
    la      t0,CharPtr
    sw      zero,0(t0)
    la      t0,BytePtr
    sw      zero,0(t0)
    la      t0,WordPtr
    sw      zero,0(t0)
    la      t0,Array2dPtr
    sw      zero,0(t0)

# Also reset counts to zero
    la      t0,CharCount
    sw      zero,0(t0)
    la      t0,ByteCount
    sw      zero,0(t0)
    la      t0,WordCount
```

```
    sw      zero,0(t0)

# Re-initialize memory
    jal     SysInitMem

    la      a0,freeMem
    li      a7,4
    ecall

    la      a0,endl
    li      a7,4
    ecall
    jal     main

    la      t0,CharPtr
    sw      zero,0(t0)
    la      t0,BytePtr
    sw      zero,0(t0)
    la      t0,WordPtr
    sw      zero,0(t0)
    la      t0,Array2dPtr
    sw      zero,0(t0)

    jal     SysInitMem

    la      a0,freeMem
    li      a7,4
    ecall

    la      a0,endl
    li      a7,4
    ecall
    jal     main

# task8: Print how many bytes allocated
    task8:
    la      a0, da_cap_phat
    li      a7,4
    ecall
    jal     AllocatedMemory
    li      a7,1
    ecall

    la      a0,endl
    li      a7,4
    ecall

    jal     main

# task9: Allocate 2D word array
    task9:
```

```
    la      a0, message_row
    jal     take_the_int
    mv      t0,a0
    la      a0,message_col
    jal     take_the_int
    mv      a1,t0
    mv      a2,a0
    la      a0,Array2dPtr
    jal     Malloc2d
    mv      s0,a0
    la      a0, message_success
    li      a7,4
    ecall
    mv      a0,s0
    li      a7,34
    ecall
    jal     main

# task10: Set element in 2D array
    task10:
    la      a0,Array2dPtr
    lw      s2,0(a0)
    la      a0,input_row
    jal     take_the_int
    mv      s0,a0
    la      a0,input_col
    jal     take_the_int
    mv      s1,a0
    la      a0,input_val
    jal     take_the_int
    mv      a3,a0
    mv      a1,s0
    mv      a2,s1
    mv      a0,s2
    jal     Set_func
    jal     main

# task11: Get element from 2D array
    task11:
    la      a0,Array2dPtr
    lw      s1,0(a0)
    la      a0,input_row
    jal     take_the_int
    mv      s0,a0
    la      a0,input_col
    jal     take_the_int
    mv      a2,a0
    mv      a1,s0
    mv      a0,s1
    jal     Get_func
    mv      s0,a0
```

```
    la       a0,output_val
    li       a7,4
    ecall
    mv       a0,s0
    li       a7,1
    ecall
    jal      main

# task12: Copy data from CharPtr to BytePtr using counts
    task12:
    la       t0,CharPtr
    lw       t1,0(t0)              # Load source address
    beq      t1,zero,no_src_alloc

    la       t0,BytePtr
    lw       t2,0(t0)              # Load dest address
    beq      t2,zero,no_dest_alloc

# Load counts
    la       t0,CharCount
    lw       t3,0(t0)              # t3 = CharCount
    la       t0,ByteCount
    lw       t4,0(t0)              # t4 = ByteCount

    blt      t4,t3,no_space_left      # If ByteCount < CharCount, no space

# ByteCount >= CharCount, can copy
    mv       a0,t2                 # dest
    mv       a1,t1                 # src
    mv       a2,t3                 # number of chars to copy
    jal      copy_char_arrays         # copy exactly t3 chars

    la       a0, cpyMess
    li       a7,4
    ecall
    jal      main

    no_src_alloc:
    la       a0,message_lessthan0
    li       a7,4
    ecall
    jal      main

    no_dest_alloc:
    la       a0,message_lessthan0
    li       a7,4
    ecall
    jal      main

    no_space_left:
    la       a0,no_space_msg
```

```
li       a7,4
ecall
jal      main

terminated:
li       a7,10
ecall

SysInitMem:
la       s11,Sys_TheTopOfFree
la       s9,Sys_MyFreeSpace
sw       s9,0(s11)
ret

malloc:
la       s11,Sys_TheTopOfFree
lw       s10,0(s11)
li       t0,4
bne      a2,t0,not_word
addi     s10,s10,3
li       t0,0xfffffffc
and      s10,s10,t0
not_word:
sw       s10,0(a0)
mv       a0,s10
mul      s9,a1,a2
add      s8,s10,s9
sw       s8,0(s11)
ret

Malloc2d:
addi     sp,sp,-4
sw       ra,0(sp)
la       s0,row
sw       a1,0(s0)
sw       a2,4(s0)
mul      a1,a1,a2
li       a2,4
jal      malloc
lw       ra,0(sp)
addi     sp,sp,4
ret

error_lessthan0:
la       a0,message_lessthan0
li       a7,4
ecall
jal      main

Take_ptr_value:
la       t0,CharPtr
```

```
slli    t1,a0,2
add     t0,t0,t1
lw      a0,0(t0)
ret

print_task5:
li      a7,34
ecall
la      a0,endl
li      a7,4
ecall
ret

Take_ptr_address:
la      t0,CharPtr
slli    t1,a0,2
add     a0,t0,t1
ret

take_the_int:
addi    t1,a0,0
li      a7,51
ecall
beq     a1,zero,got_the_int
li      t0,-2
beq     a1,t0,terminated
addi    a0,t1,0
jal     main
got_the_int:
ret

AllocatedMemory:
la      s11,Sys_TheTopOfFree
lw      s11,0(s11)
la      s10,Sys_MyFreeSpace
sub     a0,s11,s10
ret

Set_func:
la      s0,row
lw      s1,0(s0)
lw      s2,4(s0)
bge     a1,s1,bound_err
bge     a2,s2,bound_err
mul     s0,s2,a1
add     s0,s0,a2
slli    s0,s0,2
add     s0,s0,a0
sw      a3,0(s0)
ret
```

```
Get_func:
la      s0,row
lw      s1,0(s0)
lw      s2,4(s0)
bge     a1,s1,bound_err
bge     a2,s2,bound_err
mul     s0,s2,a1
add     s0,s0,a2
slli    s0,s0,2
add     s0,s0,a0
lw      a0,0(s0)
ret

bound_err:
la      a0,bound_error
li      a7,4
ecall
jal     main

# copy_char_arrays: Copies exactly a2 chars from source(a1) to dest(a0)
copy_char_arrays:
li      t0,0                # counter i=0
copy_loop:
beq     t0,a2,done_copy
lb      t1,0(a1)            # load byte from source
sb      t1,0(a0)            # store byte to dest
addi    a1,a1,1
addi    a0,a0,1
addi    t0,t0,1
j       copy_loop

done_copy:
ret
```