HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

****

ASSEMBLY LANGUAGE AND COMPUTER ARCHITECTURE LAB

MINI PROJECT REPORT

Lecturer: **Le Ba Vui**

Group Number: **7**

Group Member: **Trinh Thi Thuy Duong 20226034**

Nguyen Thi Thu Huyen 20220073

TASK DELEGATION

Student name	Student id	Project id
Trinh Thi Thuy Duong	20226034	13
Nguyen Thi Thu Huyen	20220073	4

PROBLEM 4: Memory allocation malloc()

1.1. Methods

The program below implements the malloc function in RISC-V assembly language to allocate memory for pointer variables. It handles both one-dimensional and two-dimensional arrays effectively. The program displays a menu in the I/O Run, allowing users to select tasks based on the number they input:

```
1. Handling one-dimensional array
2. String copy
3. Handling two-dimensional array
4. Free the memory allocated
Select:
```

If the user selects 1, they can input a one-dimensional array. The program will compute and display the pointer's value, its address, and the allocated memory size.

If the user selects 2, they can input a string, and the program will create and display a copied version of it.

If the user selects 3, they can input a two-dimensional array.The program will display and modify the value of the umbrella required by the user.

If the user selects 4, the program will free all memory allocated to the pointers.

1.2. Detailed Implementation

1.2.1. Setup Program

```
.data
CharPtr1: .word 0          # Biến con trỏ kiểu asciiz
CharPtr2: .word 0          # Biến con trỏ kiểu asciiz
ArrayPtr: .word 0          # Biến con trỏ mảng 1 chiều
Array2Ptr: .word 0         # Biến con trỏ mảng 2 chiều
message1: .string "\n\n1. Xu ly mang mot chieu\n"
message2: .string "2. Sao chep mang ky tu\n"
message3: .string "3. Xu ly mang hai chieu\n"
message4: .string "4. Giai phong bo nho\n"
message0.1: .string "So phan tu: "
message0.2: .string "So byte moi phan tu (1 hoac 4): "
message0.3: .string "Nhap phan tu: "
message1.1: .string "Gia tri cua con tro: "
message1.2: .string "\nDia chi cua con tro: "
message1.3: .string "\nTong dia chi da cap phat: "
message2.1: .string "So ky tu toi da: "
message2.2: .string "\nNhap chuoi ky tu: "
message2.3: .string "\nChuoi ky tu duoc copy: "
message3.1: .string "\nSo hang: "
message3.2: .string "\nSo cot: "
message3.3: .string "\n1. getArray[i][j]\n"
message3.4: .string "2. setArray[i][j]\n"
message3.5: .string "3. Thoat\n"
message3.6: .string "\nGia tri cua phan tu: "
message3.01: .string "i = "
message3.02: .string "j = "
message4.1: .string "Da giai phong toan bo bo nho cap phat.\n"
select: .string "Lua chon: "
errmessage: .string "\nSo vua nhap khong hop le.\n"


Sys_TheTopOfFree: .word 1          # Vùng không gian tự do, dùng để cấp bộ nhớ cho các biến con trỏ
Sys_MyFreeSpace:        .space 1024
```

This assembly code defines the data section of a program, setting up various memory locations, strings, and pointers used in its operation. The pointers *CharPtr1*, *CharPtr2*, *ArrayPtr*, and *Array2Ptr* are initialized to 0 and will later point to dynamically allocated memory. The strings (e.g., *message1*, *message2*, etc.) are defined to provide user prompts and messages during program execution. These messages cover operations like processing one-dimensional and two-dimensional arrays, copying character arrays, and releasing allocated memory.

Sys_TheTopOfFree holds the starting address of the custom free memory region, while *Sys_MyFreeSpace* reserves a block of 1024 bytes as a simulated heap. This space is used for manual memory allocation and deallocation by the program. The code sets up an interactive program that can perform various memory operations, with error handling and user feedback via the defined strings.

1.2.2. Dynamic-Allocated Memory and Menu Displaying

```
# Khởi tạo vùng nhớ cấp phát động
jal SysInitMem

menu:
    li a7, 4                       # syscall number for printing string
    la a0, message1                 # Load address of message1
    ecall                           # Make syscall

    la a0, message2                 # Load address of message2
    ecall                           # Make syscall

    la a0, message3                 # Load address of message3
    ecall                           # Make syscall

    la a0, message4                 # Load address of message4
    ecall                           # Make syscall

    la a0, select                   # Load address of select
    ecall                           # Make syscall

    li a7, 5                        # syscall number for reading integer
    ecall                           # Make syscall (input is in a0)
```

The program first calls a subroutine that initializes the custom memory system for dynamic memory allocation. The *SysInitMem* function initializes the custom memory management system. It sets up a pointer (*Sys_TheTopOfFree*) to point to the starting address of the free memory region (*Sys_MyFreeSpace*). This ensures that the program can track and allocate memory from the defined free space. Finally, it returns to the calling function using *jr ra*.

```
SysInitMem:
    la t0, Sys_TheTopOfFree         # Lấy con trỏ tới địa chỉ đầu tiên của vùng bộ nhớ tự do
    la t1, Sys_MyFreeSpace          # Lấy địa chỉ của vùng bộ nhớ tự do hiện tại
    sw t1, 0(t0)                    # Lưu địa chỉ của vùng bộ nhớ tự do vào Sys_TheTopOfFree
    jr ra                           # Quay lại địa chỉ trả về
```

This code segment displays a menu with several options and prompts the user for input in an assembly program. It utilizes system calls to print predefined string messages (message1, message2, message3, message4, and select) that represent menu options. This code forms the interactive interface of the program, allowing the user to select an option from the menu.

1.2.3. Malloc functions

```
malloc:
    la t0, Sys_TheTopOfFree         # Lấy con trỏ tới địa chỉ đầu tiên của vùng bộ nhớ tự do
    lw t1, 0(t0)                    # Lấy giá trị tại địa chỉ Sys_TheTopOfFree vào t1 (địa chỉ đầu tiên của vùng bộ nhớ tự do)

    li t2, 4                        # Tải giá trị 4 vào t2 (đại diện cho kích thước của 1 từ - word)
    bne a2, t2, initialize          # Nếu a2 (kích thước phần tử) không phải là 4, nhảy đến phần khởi tạo

    andi t3, t1, 0x03               # Lấy phần dư khi chia địa chỉ bộ nhớ tự do cho 4
    beq t3, x0, initialize          # Nếu phần dư = 0, tức là địa chỉ đã đúng, bỏ qua bước điều chỉnh

    addi t1, t1, 4                  # Nếu không, điều chỉnh địa chỉ đến phần bộ nhớ chia hết cho 4
    sub t1, t1, t3                  # Điều chỉnh địa chỉ (trừ t3 từ t1)

initialize:
    sw t1, 0(a0)                    # Lưu địa chỉ vào biến con trỏ (a0 chứa địa chỉ trả về của malloc)
    addi a0, t1, 0                  # Lưu địa chỉ đã điều chỉnh vào a0 để trả về

    mul t4, a1, a2                  # Tính kích thước của mảng cần cấp phát (a1 là số phần tử, a2 là kích thước mỗi phần tử)
    add t5, t1, t4                  # Tính địa chỉ của con trỏ tiếp theo (địa chỉ mới sau khi cấp phát)

    sw t5, 0(t0)                    # Cập nhật con trỏ Sys_TheTopOfFree với địa chỉ mới sau khi cấp phát
    jr ra                           # Quay lại hàm gọi
```

The *malloc* function in this code is responsible for allocating memory dynamically. It first loads the current free memory address from *Sys_TheTopOfFree* and checks if the element size (*a2*) is 4. If the size is not 4, the function adjusts the memory address to be properly aligned to a 4-byte boundary, as certain memory operations may require this alignment. The adjustment involves checking the remainder of the address when divided by 4 and adjusting the address if necessary.

Once the address is aligned, the function calculates the total memory size needed by multiplying the number of elements (*a1*) by the size of each element (*a2*). It then updates the free memory pointer (*Sys_TheTopOfFree*) with the new address after memory allocation. Finally, the allocated memory address is returned to the caller by storing it in the address passed in *a0*, completing the memory allocation process. The function then returns control back to the calling function.

1.2.4. Handling one-dimensional array

```
case_1:
    li t0, 1                    # Load immediate value 1 into t0
    bne a0, t0, case_2          # If a0 != 1, jump to case_2

    # Print message0_1
    li a7, 4                    # syscall number for printing string
    la a0, message0.1           # Load address of message0.1
    ecall                        # Make syscall (print string)

    # Read integer input into a0
    li a7, 5                    # syscall number for reading integer
    ecall                        # Make syscall (input will be stored in a0)

    bltz a0, error              # If value in a0 is < 0, jump to error

    addi a1, a0, 0              # Copy value of a0 into a1 using addi

    # Print message0_2
    li a7, 4                    # syscall number for printing string
    la a0, message0.2           # Load address of message0.2
    ecall                        # Make syscall (print message0.2)

    # Read another integer input into a0
    li a7, 5                    # syscall number for reading integer
    ecall                        # Make syscall (input will be stored in a0)

is1:
    li t0, 1                     # Load immediate value 1 into t0
    beq a0, t0, ready            # If a0 == 1, jump to ready

is4:
    li t1, 4                     # Load immediate value 4 into t1
    beq a0, t1, ready            # If a0 == 4, jump to ready
```

```
ready:
    addi a2, a0, 0              # Copy value from a0 to a2 (no 'move' in RISC-V)
    la a0, ArrayPtr             # Load the address of ArrayPtr into a0
    jal malloc                   # Call malloc to allocate memory
    mv t0, a0                  # Sao chép kết quả malloc (địa chỉ cấp phát) vào t0
    mv a3, t0                  # Khởi tạo a3 bằng địa chỉ bộ nhớ cấp phát (a3 = t0)


    addi t0, a0, 0              # Store the result of malloc into t0 (copying address)

    # Print message0.3
    li a7, 4                    # Syscall number for printing a string
    la a0, message0.3           # Load address of message0.3
    ecall                        # Make syscall (print string)

    addi a0, t0, 0              # Copy the allocated address from t0 into a0
    addi t0, x0, 0              # Set t0 to 0 (clear t0)

input_loop:
    beq t0, a1, input_end       # Nếu t0 == a1, thoát khỏi vòng lặp
    li a7, 5                    # Syscall đọc số nguyên (read integer)
    ecall                        # Gọi syscall, kết quả sẽ lưu vào a0

    li t1, 1                    # Gán t1 = 1 để so sánh
    bne a2, t1, byte_4          # Nếu a2 != 1, nhảy tới byte_4

byte_1:
    sb a0, 0(a3)               # Lưu 1 byte từ a0 vào địa chỉ a3
    addi a3, a3, 1             # Tăng con trỏ a3 lên 1 byte
    addi t0, t0, 1             # Tăng bộ đếm t0
    j input_loop               # Quay lại đầu vòng lặp
```

```
byte_4:
    sw a0, 0(a3)               # Lưu 4 byte từ a0 vào địa chỉ a3
    addi a3, a3, 4             # Tăng con trỏ a3 lên 4 byte
    addi t0, t0, 1             # Tăng bộ đếm t0
    j input_loop                # Quay lại đầu vòng lặp

input_end:
    li a7, 4                    # Syscall: in chuỗi (print string)
    la a0, message1.1           # Nạp địa chỉ chuỗi message1.1
    ecall                       # Gọi syscall để in chuỗi

    la a0, ArrayPtr             # Nạp địa chỉ của ArrayPtr vào a0
    jal getValue                # Gọi hàm getValue
    #mv a0, a0                   # Truyền giá trị trả về của hàm vào a0
    li a7, 1                  # Syscall: in địa chỉ (print address)
    ecall                       # Gọi syscall

    li a7, 4                    # Syscall: in chuỗi (print string)
    la a0, message1.2           # Nạp địa chỉ chuỗi message1.2
    ecall                       # Gọi syscall để in chuỗi

    la a0, ArrayPtr             # Nạp địa chỉ của ArrayPtr vào a0
    jal getAddress              # Gọi hàm getAddress
    #mv a0, a0                   # Truyền giá trị trả về của hàm vào a0
    li a7, 1                  # Syscall: in địa chỉ (print address)
    ecall                       # Gọi syscall

    li a7, 4                    # Syscall: in chuỗi (print string)
    la a0, message1.3           # Nạp địa chỉ chuỗi message1.3
    ecall                       # Gọi syscall để in chuỗi

    jal memoryCalculate         # Gọi hàm memoryCalculate
    mv a0, a0                  # Truyền giá trị trả về của hàm vào a0
    li a7, 1                   # Syscall: in số nguyên (print integer)
    ecall                       # Gọi syscall
```

When the user chooses option 1, the program will jump to *case_1*. This code manages a one-dimensional array. First, it prompts the user to input the array's length, select the byte size for each element, and enter the array elements. During input, the program checks conditions such as whether the array length is negative or the byte size is neither 1 nor 4. Once the input is validated, the program proceeds to the "*ready*" stage, where it calls the malloc subroutine to dynamically allocate memory and stores the address of the *ArrayPtr* pointer for future use.

In the *input_loop* stage, depending on the byte size chosen by the user, the array memory is updated with the appropriate size to accommodate new elements. After the loop finishes, the program calls *getValue* to retrieve the pointer's value, *getAddress* to get the pointer's address, and *memoryCalculate* to calculate the total memory allocated.

```
# getValue: Lấy giá trị của biến con trỏ
getValue:
    lw a0, 0(a0)        # Lấy giá trị của biến con trỏ trong ô nhớ có địa chỉ lưu trong $a0
    jr ra                # Quay lại hàm gọi (return)

# getAddress: Lấy địa chỉ từ $a0 và trả về trong $a0
getAddress:
    add a0, x0, a0       # Lấy địa chỉ từ $a0 và lưu vào $a0 (trả về địa chỉ)
    jr ra                # Quay lại hàm gọi (return)
```

```
memoryCalculate:
    la t0, Sys_MyFreeSpace        # Tải địa chỉ của Sys_MyFreeSpace vào t0
    la t1, Sys_TheTopOfFree       # Tải địa chỉ của Sys_TheTopOfFree vào t1
    lw t2, 0(t1)                  # Tải giá trị tại địa chỉ Sys_TheTopOfFree vào t2 (địa chỉ đầu tiên con trống)
    sub a0, t2, t0                # Tính hiệu giữa hai địa chỉ (t2 - t0), kết quả vào a0
    jr ra                         # Quay lại địa chỉ gọi hàm
```

In *getValue* and *getAddress*, *a0* stored the address of pointer *ArrayPtr*. The *memoryCalculate* function calculates the amount of free memory available by subtracting the address of the starting point of the allocated memory (*Sys_MyFreeSpace*, stored in *t0*) from the address of the first available free memory block (*Sys_TheTopOfFree*, stored in *t1*). The value at *Sys_TheTopOfFree* is loaded into *t2*, and the difference between *t2* and *t0* is computed and stored in *a0*, representing the amount of free memory. After calculating this value, the function returns to the caller. This operation helps track the available memory within a system.

1.2.5. String copy

```
case_2:
    li t1, 2                  # Tải hằng số 2 vào t1
    bne a0, t1, case_3        # Nếu a0 != 2 thì nhảy đến case_3

    li a7, 4                  # Syscall: in chuỗi
    la a0, message2.1         # Nạp địa chỉ của message2.1
    ecall                     # Gọi syscall để in chuỗi

    li a7, 5                  # Syscall: nhập số nguyên
    ecall                     # Kết quả sẽ nằm trong a0
    mv a1, a0                 # Lưu số nguyên vào a1
    li a2, 1                  # Gán giá trị 1 cho a2 (sử dụng làm tham số)

    la a0, CharPtr1           # Nạp địa chỉ CharPtr1 vào a0
    jal malloc                # Gọi hàm malloc để cấp phát bộ nhớ
    mv s0, a0                 # Lưu địa chỉ trả về của malloc vào s0

    la a0, CharPtr2           # Nạp địa chỉ CharPtr2 vào a0
    jal malloc                # Gọi hàm malloc để cấp phát bộ nhớ
    mv s1, a0                 # Lưu địa chỉ trả về của malloc vào s1

    li a7, 4                  # Syscall: in chuỗi
    la a0, message2.2         # Nạp địa chỉ của message2.2
    ecall                     # Gọi syscall để in chuỗi
```

```
    mv a0, s0                 # Truyền địa chỉ CharPtr1 vào a0
    li a7, 8                  # Syscall: nhập chuỗi ký tự
    ecall                     # Gọi syscall để nhập chuỗi

    mv a1, s1                 # Truyền địa chỉ CharPtr2 vào a1
    jal strcpy                # Gọi hàm strcpy để sao chép chuỗi

    li a7, 4                  # Syscall: in chuỗi
    la a0, message2.3         # Nạp địa chỉ của message2.3
    ecall                     # Gọi syscall để in chuỗi

    mv a0, s1                 # Truyền địa chỉ CharPtr2 vào a0
    li a7, 4                  # Syscall: in chuỗi
    ecall                     # Gọi syscall để in chuỗi đã sao chép

    j menu                    # Quay lại menu
```

When the user selects option 2, the program jumps to *case_2*, where it handles string copying using two pointers. The program first prompts the user to input the string's length, followed by the string itself. It then calls the *malloc* subroutine to allocate memory for two pointers, *CharPtr1* and *CharPtr2*. Once the memory is allocated, the program prepares to call the *strcpy* subroutine, which is responsible for copying the input string from the source to the destination using the two pointers.

```
# strcpy: Sao chép chuỗi từ $a0 (nguồn) sang $a1 (đích)
strcpy:
    add a2, x0, a0        # Khởi tạo $a2 ở đầu chuỗi nguồn (nguồn)
    add a3, x0, a1        # Khởi tạo $a3 ở đầu chuỗi đích (đích)

cpyLoop:
    lb a4, 0(a2)          # Đọc ký tự từ chuỗi nguồn (a2)
    beq a4, x0, cpyLoopEnd  # Nếu ký tự là '\0' (end of string), dừng vòng lặp
    sb a4, 0(a3)          # Lưu ký tự vào chuỗi đích (a3)

    addi a2, a2, 1        # Chuyển đến ký tự tiếp theo trong chuỗi nguồn
    addi a3, a3, 1        # Chuyển đến ký tự tiếp theo trong chuỗi đích
    j cpyLoop             # Quay lại vòng lặp

cpyLoopEnd:
    jr ra                 # Trở về
```

The *strcpy* function in this code is responsible for copying a string from a source (held in register *a0*) to a destination (held in register *a1*). The function first initializes two pointers: *a2* points to the source string, and *a3* points to the destination string. It then enters a loop where it reads each byte (character) from the source string. If the byte is the null terminator ('\0'), which marks the end of the string, the loop exits. Otherwise, the character is stored in the destination string, and both the source and destination pointers are incremented to move to the next character. The process continues until the entire string is copied. Once the loop finishes, the function returns control to the calling function, completing the string copy operation.

1.2.6. Handling two-dimensional array:

This part is designed to handle dynamic 2D arrays in RISC-V assembly. It allows users to create a 2D array, input values into its elements, and Interact with It Through a Menu-Driven Interface. Below is a detailed analysis of the program, Breaking it into logical components with an emphasis on understanding its functionality.

1.2.6.1. Handling main menu selection

```
case_3:
        li t1, 3
        bne a0, t1, case_4
```

- **Purpose**: To handle the user's selection from the main menu.
- **Details**:
  - The program checks if the user chose option **3** (the option for creating a 2D array).
  - If the user didn't select **3**, the program jumps to the label *case_4* to handle other choices.

1.2.6.2. Input the number of rows and columns

```
li a7, 4                    #nhap hang
la a0, message3.1
ecall

li a7, 5
ecall
mv a1, a0

li a7, 4                    #nhap cot
la a0, message3.2
ecall

li a7, 5
ecall
mv a2, a0
```

- **Purpose:** To ask the user for the dimensions of the 2D array (rows and columns) and store these values in *a1* (rows) and *a2* (columns).
- **Details:**
  - The system uses two system calls:
    - 4: To display messages to the user.
    - 5: To read inputs from the user.
  - The number of rows (*a1*) and columns (*a2*) will be used later to allocate memory and calculate indices.

1.2.6.3. Allocate memory for the 2D array

```
# Call malloc2 to allocate memory
la a0, Array2Ptr
jal malloc2
mv t0, a0
mv a3, t0

# Display message0.3
li a7, 4
la a0, message0.3
ecall

# Store the base address of Array2Ptr in a0
mv a0, t0
```

- **Purpose:** To dynamically allocate memory for the 2D array and save its base address for further operations.
- **Details:**
  - The *malloc2* function calculates the required memory based on the number of rows and columns (*rows × columns × 4 bytes*) and allocates it.
  - The base address of the allocated memory is stored in *t0* (and copied into *a3* for later use).

```
# malloc2
malloc2:
    addi sp, sp, -12
    sw ra, 8(sp)
    sw a1, 4(sp)
    sw a2, 0(sp)

    mul a1, a1, a2
    addi a2, x0, 4
    jal malloc

    lw ra, 8(sp)
    lw a1, 4(sp)
    lw a2, 0(sp)
    addi sp, sp, 12
    jr ra
```

The *malloc2* function is a custom memory allocation routine for two-dimensional arrays. It calculates the total memory required for a matrix (based on the number of rows and columns) and calls the standard *malloc* function to allocate memory. After allocation, it restores the saved registers to ensure the program's state remains consistent.

1.2.6.4. Input values into the 2D array

```
# Initialize the loop
mv t0, x0
mv t1, a1
mul a1, a1, a2

# input_loop2
input_loop2:
    beq t0, a1, input_end2
    li a7, 5
    ecall
    sw a0, 0(a3)      # Store the value at the current address
    addi a3, a3, 4    # Move to the next memory location
    addi t0, t0, 1    # Increment counter
    j input_loop2

input_end2:
    mv a1, t1          # Restore the value of a1
```

Purpose: To fill the 2D array with user-provided values.

Details:

- A loop iterates through the total number of elements in the array (rows × columns).
- For each iteration:

    - The program reads a value from the user.
    - The value is stored at the current memory location pointed to by *a3*.
    - The pointer (*a3*) is incremented by 4 bytes to move to the next element.
- The loop exits when all elements are entered.

1.2.6.5. Display the ***submenu***

```
# sub_menu
sub_menu:
    li a7, 4
    la a0, message3.3
    ecall
    la a0, message3.4
    ecall
    la a0, message3.5
    ecall
    la a0, select
    ecall

    li a7, 5
    ecall
```

Purpose: To present the user with a *submenu* and read their choice.
Details:

- The *submenu* typically provides options like:
    1. Retrieve a value from the array.
    2. Update a value in the array.
    3. Exit the *submenu*.

1.2.6.6. Retrieve an element from the array (*getArray*)

```
# sub_case_1
sub_case_1:
    li t1, 1
    bne a0, t1, sub_case_2

    # Display message3.01
    li a7, 4
    la a0, message3.01
    ecall

    # Read integer input for $s0
    li a7, 5
    ecall
    mv s0, a0

    # Display message3.02
    li a7, 4
    la a0, message3.02
    ecall

    # Read integer input for $s1
    li a7, 5
    ecall
    mv s1, a0

    # Load the array pointer
    la a1, Sys_MyFreeSpace
    jal getArray
    mv s2, a0

    # Display message3.6
    li a7, 4
    la a0, message3.6
    ecall

    # Print the value in $s2
    li a7, 1
    mv a0, s2
    ecall
    j sub_menu
```

Purpose: To retrieve a specific element from the 2D array based on user-provided row and column indices.

Details:

- The program prompts the user for the row and column indices.
- The *getArray* function calculates the memory address of the requested element and retrieves its value.
- The retrieved value is displayed to the user.

```
getArray:
    mul t0, s0, a2          # i * số cột
    add t0, t0, s1          # Thêm j vào
    slli t0, t0, 2          # Nhân với 4 để có byte offset
    add t0, t0, a1          # Cộng với địa chỉ đầu màng
    lw a0, 0(t0)            # Lấy giá trị phần tử
    jr ra
```

The *getArray* function is used to retrieve the value of an element in a **two-dimensional array**. The function uses the row index (*s0*) and column index (*s1*) to calculate the correct memory address of the requested element and then loads the value stored at that memory address.

1.2.6.7. Update an element in the array (*setArray*)

```
# sub_case_2
sub_case_2:
    li t1, 2
    bne a0,t1, sub_case_3

    # Display message3.01
    li a7, 4
    la a0, message3.01
    ecall

    # Read integer input for $s0
    li a7, 5
    ecall
    mv s0, a0

    # Display message3.02
    li a7, 4
    la a0, message3.02
    ecall

    # Read integer input for $s1
    li a7, 5
    ecall
    mv s1, a0

    # Move $v0 to $s2
    mv s2, a0

    # Display message0.3
    li a7, 4
    la a0, message0.3
    ecall

    # Read integer input for $v0
    li a7, 5
    ecall

    # Load the array pointer
    la a1, Sys_MyFreeSpace

    jal setArray
    j sub_menu
```

Purpose: To update the value of a specific element in the 2D array.

Details:

- The user is prompted for the row and column indices and the new value.
- The *setArray* function calculates the memory address of the element and updates its value.

```
setArray:
    mul t0, s0, a2          # t0 = i * số cột
    add t0, t0, s1          # t0 = (i * số cột) + j
    slli t0, t0, 2          # t0 = (i * số cột + j) * 4 (byte offset)
    add t0, t0, a1          # t0 = Địa chỉ của phần tử (địa chỉ đầu màng + offset)
    sw a0, 0(t0)            # Lưu giá trị vào màng tại vị trí tính toán
    jr ra                   # Trở lại
```

The *setArray* function is used to **set** or **store** a value in a specific element of a 2D array. Similar to *getArray*, it calculates the memory address of the target array element based on the row and column indices, but instead of loading the value, it **stores** the value into that memory address.

1.2.6.8. Exit

```
# sub_case_3
sub_case_3:
li t1, 3
    bne a0, t1, error
    j menu
```

It ensures that the program either exits the *submenu* gracefully or jumps to an error-handling routine if the user's choice is invalid.

1.2.7. Free the memory allocated

```
case_4:
    li t1, 4                        # Tải giá trị 4 vào t1
    bne a0, t1, error               # Nếu a0 != 4, nhảy đến error

    jal free                        # Gọi hàm free

    # In message4.1
    li a7, 4                        # Syscall: in chuỗi
    la a0, message4.1               # Nạp địa chỉ message4.1
    ecall                           # Thực thi syscall

    # In message1.3
    li a7, 4                        # Syscall: in chuỗi
    la a0, message1.3               # Nạp địa chỉ message1.3
    ecall                           # Thực thi syscall

    jal memoryCalculate             # Gọi hàm memoryCalculate

    mv a0, a0                       # Sử dụng lệnh mv thay vì move
    li a7, 1                        # Syscall: in số nguyên
    ecall                           # Thực thi syscall

    j menu                          # Quay lại menu
```

When the user selects option 4, the program will enter the *case_4* subroutine. It will first verify whether the value in *a0* is 4. If this condition is met, the program will then proceed to call the free subroutine.

```
free:
    addi sp, sp, -4        # Tạo không gian 4 byte trên stack
    sw ra, 0(sp)           # Lưu giá trị của $ra vào stack (để quay lại sau khi thực hiện)
    jal SysInitMem         # Gọi hàm SysInitMem (để khởi tạo lại vị trí bộ nhớ)
    lw ra, 0(sp)           # Lấy lại giá trị của $ra từ stack
    addi sp, sp, 4         # Khôi phục lại stack (xóa không gian đã sử dụng)
    jr ra                  # Trở về điểm gọi
```

The free subroutine is responsible for deallocating memory by resetting the memory system. The process starts by creating 4 bytes of space on the stack and saving the return address (*ra*) to ensure the program can return to the correct location after the function call. Then, the free function calls the *SysInitMem* subroutine, which is responsible for resetting or re-initializing the memory system, by updating pointers or clearing allocated memory. After the *SysInitMem* function finishes executing, the return address (*ra*) is restored from the stack to ensure the program returns to the right point. The stack pointer is restored to its original state by adjusting *sp*, and the subroutine returns to the caller using the *jr ra* instruction. This ensures that memory is freed, and the program can continue from the point where the free function was called.

After that, the program will again the total memory allocated by program. If the above logic works correctly, the total amount of memory would be 0.

1.3. Simulation Results

1.3.1. Handling one-dimensional array

```
1. Handling one-dimensional array
2. String copy
3. Handling two-dimensional array
4. Free the memory allocated
Select: **** user input : 1
Number of Element **** user input : 3
Number of bytes per element (1 or 4) **** user input : 4
Enter the element **** user input : 3
**** user input : 4
**** user input : 5
The value of the pointer: 268501596
The address of the pointer: 268501000
the amount of allocated memor: 12
```

```
1. Handling one-dimensional array
2. String copy
3. Handling two-dimensional array
4. Free the memory allocated
Select: **** user input : 1
Number of Element **** user input : 3
Number of bytes per element (1 or 4) **** user input : 1
Enter the element **** user input : 1
**** user input : 2
**** user input : 3
The value of the pointer: 268501608
The address of the pointer: 268501000
the amount of allocated memor: 15
```

1.3.2. String copy

```
1. Handling one-dimensional array
2. String copy
3. Handling two-dimensional array
4. Free the memory allocated
Select: **** user input : 2
Maximum number of characters: **** user input : 30

Enter the string of characters: **** user input : chuc mot giang sinh an lanh ha

Character series is copied: chuc mot giang sinh an lanh h
```

1.3.3. Handling two-dimensional array

```
1. Handling one-dimensional array
2. String copy
3. Handling two-dimensional array
4. Free the memory allocated
Select: **** user input : 3

Row number: **** user input : 2

Column number: **** user input : 2
Enter the element **** user input : 1
**** user input : 2
**** user input : 3
**** user input : 4
```

```
1. getArray[i][j]
2. setArray[i][j]
3. Exit
Select: **** user input : 1
i = **** user input : 1
j = **** user input : 1

The value of element: 4



1. getArray[i][j]
2. setArray[i][j]
3. Exit
Select: **** user input : 2
i = **** user input : 1
j = **** user input : 1
Enter the element **** user input : 5


1. getArray[i][j]
2. setArray[i][j]
3. Exit
Select: **** user input : 1
i = **** user input : 1
j = **** user input : 1

The value of element: 5
```

```
1. getArray[i][j]
2. setArray[i][j]
3. Exit
Select: **** user input : 3


1. Handling one-dimensional array
2. String copy
3. Handling two-dimensional array
4. Free the memory allocated
```

1.3.4. Free the memory allocated

```
1. Xu ly mang mot chieu
2. Sao chep mang ky tu
3. Xu ly mang hai chieu
4. Giai phong bo nho
Lua chon: **** user input : 4
Da giai phong toan bo bo nho cap phat.

Tong dia chi da cap phat: 0
```

PROBLEM 15: Simon Game

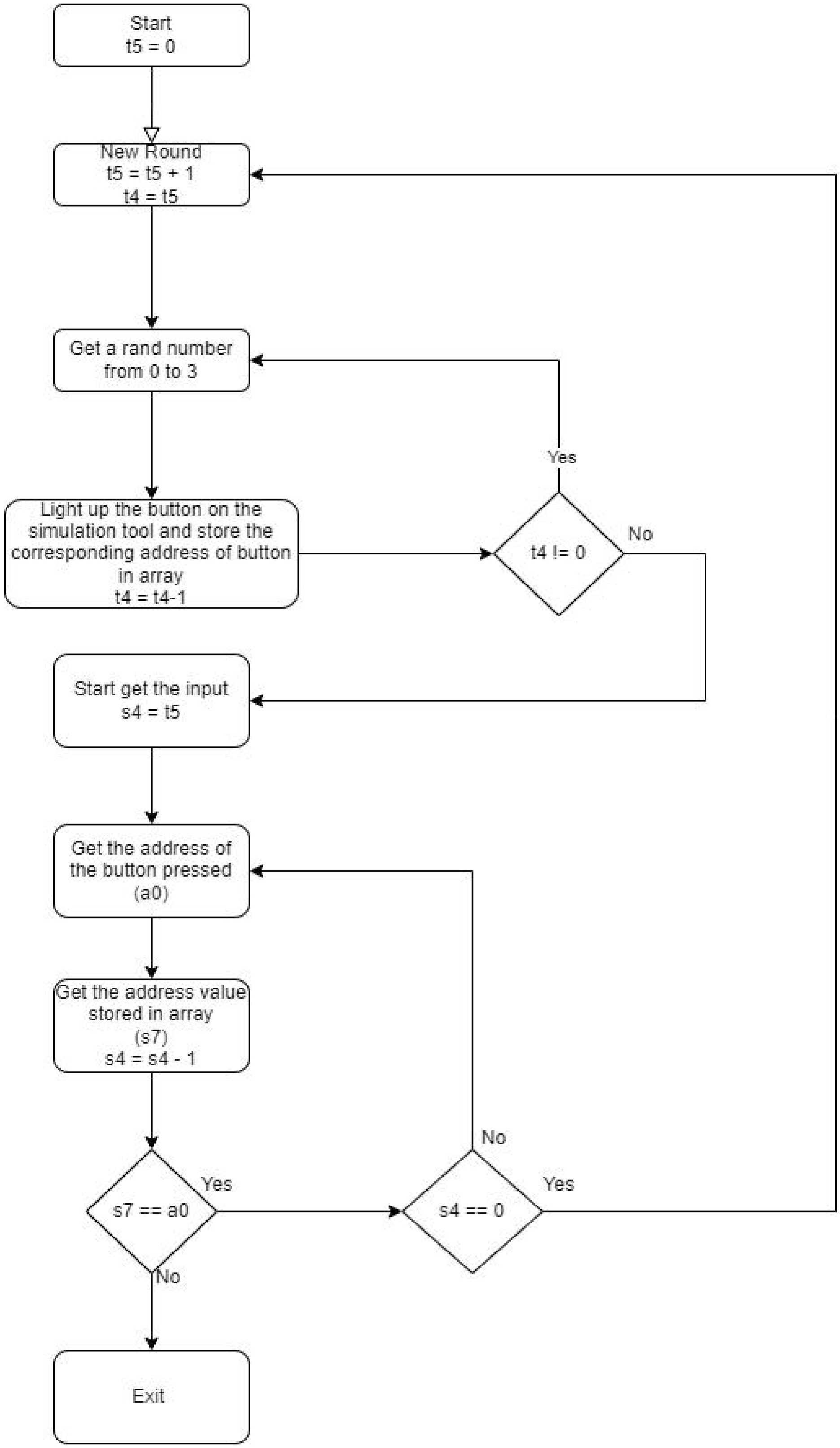
1.1. Project Description

The Simon Game is a simple and fun memory game that tests how well you can remember patterns. It has four colored buttons that light up and in a certain order. At the start, the game shows a short pattern by lighting up the buttons, and the player has to copy it by pressing the same buttons in the correct order. As the game goes on, the length of pattern gets longer and more difficult to remember. If you make a mistake, the game ends, and you can try again to beat your high score. It’s a great game for people of all ages to improve memory and concentration while having fun.

In this project, we create a program to simulate the Simon Game using RARS tools, namely Bitmap Display and Digital Lab Sim. The Bitmap Display will visually represent the four colored buttons, while in the Digital Lab Sim, the first-row buttons (0, 1, 2, 3) correspond to the colored buttons in the Bitmap Display (Red, Green, Blue, and Yellow). Players must press the buttons in the correct order as shown by the lights on the Bitmap Display. If they succeed, the message “Round win!” will appear in the Run I/O section. However, if they make an error, the program will immediately terminate, displaying the message “You failed!”.

1.2. Methods

Below is the flowchart of the program. This flowchart is made for easier understanding of the source code implementation.



The program data consists of an array *mang* to store the value of corresponding address of the button on the Digital Lab Sim. This program is an infinite *round* loop, program exits only when players choose the incorrect button.

First, the program randomly generates an integer between 0 and 3. If the number is 0, the red button lights up and address value 0x11 is stored in *mang* array. If it is 1, the green button lights up and address value 0x21 is stored in *mang* array. For 2, the blue button lights up, address value 0x41 is stored, and for 3, the yellow button lights up, 0x81 is stored in *mang* array. All the coloured buttons are displayed on Bitmap Display.

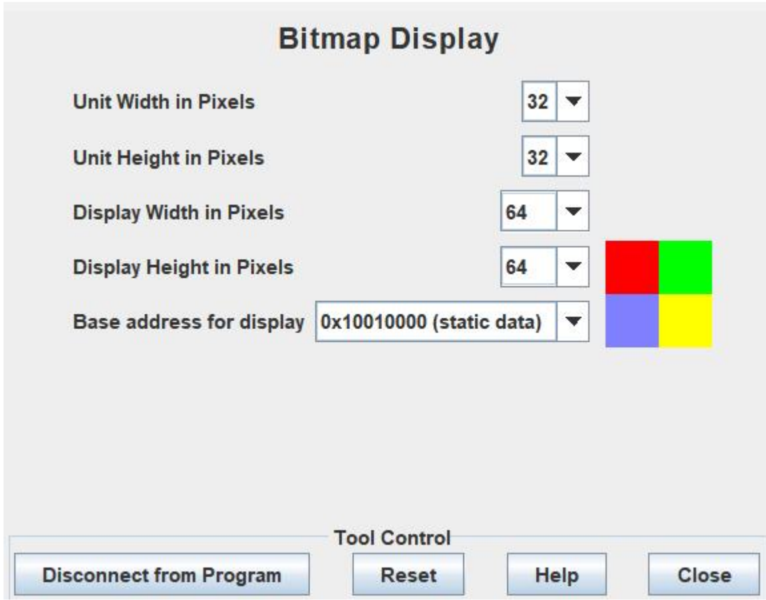
Next, the program reads the address of the button pressed in Digital Lab Sim and retrieves the corresponding values stored in the array mang to compare them with the expected button addresses. The program runs through several loops, checking conditions: if the returned value is zero (indicating no button was pressed), the loop continues. If the player presses the buttons in the correct sequence, they win the current round and proceed to the next iteration of the round loop. However, if even a single value is incorrect, the program will terminate immediately.

1.3. Detail Implementation

1.3.1. Set up program:

```
.eqv MONITOR_SCREEN 0x10010000 # Start address of the bitmap display
.eqv RED 0x00FF0000 # Common color values
.eqv LIGHTER_RED 0x00FF8080
.eqv GREEN 0x0000FF00
.eqv LIGHTER_GREEN 0x0080FF80
.eqv BLUE 0x000000FF
.eqv LIGHTER_BLUE 0x008080FF
.eqv YELLOW 0x00FFFF00
.eqv LIGHTER_YELLOW 0x00FFFF80
.eqv IN_ADDRESS_HEXA_KEYBOARD 0xFFFF0012
.eqv OUT_ADDRESS_HEXA_KEYBOARD 0xFFFF0014
.data
X: .space 16
mang: .space 400
string: .asciz "Round win!"
nline: .asciz "\n"
string2: .asciz "You lose!"
.text
set_up_program:
 li a0, MONITOR_SCREEN # Load address of the display
 li t0, RED
 sw t0, 0(a0)
 li t0, GREEN
 sw t0, 4(a0)
 li t0, BLUE
 sw t0, 8(a0)
 li t0, YELLOW
 sw t0, 12(a0)
 li t5, 0
```

This label is called every start of the round to set the default state of the program: the array of integer *mang* is used to store corresponding address values. We setup a 2x2 grid representing 4 colored buttons in Bitmap Display with unit width and height in pixels are both 32, while display width and height are both 64.



The *round* loop is the core loop of our program. It represents the number of rounds the player will play in the game. The loop only ends if the player presses an incorrect button; otherwise, it will continue indefinitely.

```
 round:
 la s6, mang
 addi t5, t5, 1
 add t4, zero, t5
 display:
 beq t4, zero, continue
 li a1, 4
 li a7, 42
 ecall
 beq a0, zero, call_subroutine0   # If a0 == 0, call subroutine0
 li t0, 1
 beq a0, t0, call_subroutine1     # If a0 == 1, call subroutine1
 li t0, 2
 beq a0, t0, call_subroutine2     # If a0 == 2, call subroutine2
 li t0, 3
 beq a0, t0, call_subroutine3
 # Exit program (in case of invalid a0)
continue:
 add s4, zero, t5
 la s6, mang
 loop:
 li t1, IN_ADDRESS_HEXA_KEYBOARD
 li t2, OUT_ADDRESS_HEXA_KEYBOARD
 li t3, 0x01 # check row 4 with key 0, 1, 2, 3
 lw s7, 0(s6)          # Lấy giá trị màu từ stack
 addi s6, s6, 4        # Tăng stack pointer để giải phóng
polling:
    sb t3, 0(t1)    # Gửi hàng mong muốn đến bàn phím
    lb a0, 0(t2)    # Đọc mã quét của phím bấm từ hàng đã chọn
```

Each iteration, we invoke system call number 42 with an upper bound of 4 to generate a random number, which is stored in register a0. The program then evaluates the value of a0 and directs the flow to the corresponding subroutine based on the result. Specifically, if a0 equals 0, the program jumps to call_subroutine0, and similarly, it follows this pattern, jumping to call_subroutine1 for a0 = 1, call_subroutine2 for a0 = 2, and call_subroutine3 for a0 = 3. Each subroutine corresponds to a different colored button.

1.3.2. Four subroutines corresponding to colored buttons

```
# Subroutine 0: Draw RED color at address MONITOR_SCREEN
call_subroutine0:
    li a0, MONITOR_SCREEN
    li t0, LIGHTER_RED
    sw t0, 0(a0)        # Store RED color at the first pixel
    li t1, 1000000       # Set counter for delay
    li a0, 1000 # sleep 100ms
    li a7, 32
    ecall
    # Change to Yellow after delay
    li a0, MONITOR_SCREEN
    li t0, RED
    sw t0, 0(a0)
    li t1, 0x11
    sw t1, 0(s6)          # Đẩy giá trị màu vào stack
    addi s6, s6, 4       # Giảm stack pointer để tạo không gian
    addi t4, t4, -1
    j display             # Return
```

```
# Subroutine 1: Draw GREEN color at address MONITOR_SCREEN + 4
call_subroutine1:
    li a0, MONITOR_SCREEN
    li t0, LIGHTER_GREEN
    sw t0, 4(a0)        # Store GREEN color at the second pixel
    li t1, 1000000       # Set counter for delay
    li a0, 1000 # sleep 100ms
    li a7, 32
    ecall
    # Change to Yellow after delay
    li a0, MONITOR_SCREEN
    li t0, GREEN
    sw t0, 4(a0)
    li t1, 0x21
    sw t1, 0(s6)          # Đẩy giá trị màu vào stack
    addi s6, s6, 4       # Giảm stack pointer để tạo không gian
    addi t4, t4, -1
    j display             # Return
```

```
# Subroutine 2: Draw BLUE color at address MONITOR_SCREEN + 8
call_subroutine2:
    li a0, MONITOR_SCREEN
    li t0, LIGHTER_BLUE
    sw t0, 8(a0)        # Store BLUE color at the third pixel
    li t1, 1000000       # Set counter for delay
    li a0, 1000 # sleep 100ms
    li a7, 32
    ecall
    # Change to Yellow after delay
    li a0, MONITOR_SCREEN
    li t0, BLUE
    sw t0, 8(a0)
    li t1, 0x41
    sw t1, 0(s6)          # Đẩy giá trị màu vào stack
    addi s6, s6, 4       # Giảm stack pointer để tạo không gian
    addi t4, t4, -1
    j display             # Return
```

```
# Subroutine 3: Draw BLUE color at address MONITOR_SCREEN + 12
call_subroutine3:
    li a0, MONITOR_SCREEN
    li t0, LIGHTER_YELLOW
    sw t0, 12(a0)        # Store YELLOW color at the third pixel
    # Delay loop (simulate 1 second)
    li t1, 1000000       # Set counter for delay
    li a0, 1000 # sleep 100ms
    li a7, 32
    ecall
    # Change to Yellow after delay
    li a0, MONITOR_SCREEN
    li t0, YELLOW
    sw t0, 12(a0)         # Update pixel to YELLOW
    li t1, 0xffffff81
    sw t1, 0(s6)          # Đẩy giá trị màu vào stack
    addi s6, s6, 4       # Giảm stack pointer để tạo không gian
    addi t4, t4, -1
    j display
```

Initially, the program lights up the button by displaying a lighter shade of the color (e.g., BLUE is shown as LIGHTER_BLUE). After 1 second, the color reverts to its original version, and the corresponding address values are stored in the mang array. The address values for each color are defined as follows:

- RED: 0x11
- GREEN: 0x21
- BLUE: 0x41
- YELLOW: 0xffffff81

The value in register t4 is used as a counter to determine how many buttons need to be predicted in each round. For each subroutine, t4 is decremented.

1.3.3. Let players press the buttons

```
continue:
 add s4, zero, t5
 la s6, mang
 loop:
 li t1, IN_ADDRESS_HEXA_KEYBOARD
 li t2, OUT_ADDRESS_HEXA_KEYBOARD
 li t3, 0x01 # check row 4 with key 0, 1, 2, 3
 lw s7, 0(s6)
 addi s6, s6, 4
polling:
    sb t3, 0(t1)
    lb a0, 0(t2)

    beq s7, a0, match
    beq a0, zero, polling
    j exit
match:
    addi s4, s4, -1
sleep:
 li a0, 100 # sleep 100ms
 li a7, 32
 ecall
 li a0, 0
 bne s4, zero, loop
```

The process starts by copying the value of t5 into s4 and loading the address of mang into s6.

In the loop, it sets up addresses for input (IN_ADDRESS_HEXA_KEYBOARD) and output (OUT_ADDRESS_HEXA_KEYBOARD) which are the essential addresses in Digital Lab Sim and checks for button presses on row 1 (with possible keys 0, 1, 2, 3). The value at s6 (the current address in mang) is loaded into s7 and then incremented.

The program polls for button input by writing a value (0x01) to t3 and checking the input at t2 using lb a0. If the pressed key matches the expected value (s7 == a0), it decrements s4 and proceeds. If no match is found, it jumps to exit. If the input is zero, it either keeps polling.

When a match is found, it waits for 100ms (sleep section) using the ecall system call and then checks if there are more buttons to predict. If s4 is not zero, it repeats the loop; otherwise, the program will continue.

1.3.4. Complete a round

```
Complete_a_round:
li a7, 4
la a0, string
ecall
la a0, nline
li a7, 4
ecall
li a0, 0
j round
```

If players complete one round (all of the correct buttons are pressed in a correct order way), the system will print message "Round win" and jump to the next round by *j round*:

```
You lose!
-- program is finished running (0) --

Round win!
Round win!
Round win!
Round win!
Round win!
```

1.3.5. Exit the program

```
exit:
la a0, string2
li a7, 4
ecall
li a7, 10
ecall
```

The program jump to *exit* tag just in case players choose the wrong buttons. Hence, system will print out the message "You lose!":

```
Round win!
Round win!
Round win!
You lose!
```

1.4. Simulation Results