HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



BÁO CÁO CUỐI KỲ

**THỰC HÀNH KIẾN TRÚC MÁY TÍNH**

**NHÓM 14**

**Giảng viên hướng dẫn**

Ngô Lam Trung

**Trợ Giảng**

Đỗ Quốc Minh

**Thành viên nhóm**

Phan Khánh Vũ

Hoàng Văn Thắng

***Mục lục***

[1. Assignment 6: Mô phỏng ổ đĩa RAID 5 3](#_Toc199347229)

[**1.1.** **Mô tả bài toán:** 3](#_Toc199347230)

[**1.2.** **Thuật toán sử dụng:** 4](#_Toc199347231)

[1.2.1. Khởi tạo 4](#_Toc199347232)

[1.2.2. Nhập liệu và kiểm tra độ dài : 4](#_Toc199347233)

[1.2.3. Thuật toán 3: Xử lý và phân phối dữ liệu (RAID 5 Logic): 4](#_Toc199347234)

[1.2.4. Thủ tục HEX (Chuyển đổi sang Hexa): 5](#_Toc199347235)

[**1.3.** **Đoạn mã lệnh và giải thích** 6](#_Toc199347236)

[1.3.1. Khai báo dữ liệu (.data) 6](#_Toc199347237)

[1.3.2. Main và input 7](#_Toc199347238)

[1.3.3. Kiểm tra độ dài chuỗi (length, check\_char, test\_length, error) 8](#_Toc199347239)

[1.3.4. Thủ tục HEX (Chuyển byte sang 2 ký tự Hexa ASCII) 10](#_Toc199347240)

[1.3.5. Mô phỏng RAID 5 (block1, block2, block3) 11](#_Toc199347241)

[1.4. Chạy thử mã lệnh và kết quả: 28](#_Toc199347242)

[1.4.1. Kết quả hiển thị ảnh: 28](#_Toc199347243)

[1.4.2. Kết quả hiển thị ảnh đã xóa màu cho các chữ cái DCE trong ảnh: 28](#_Toc199347244)

[1.4.3. Kết quả hiển thị ảnh đã hoán đổi vị trí các chữ cái: 29](#_Toc199347245)

[1.4.4. Kết quả hiển thị ảnh các chữ cái DCE với giá trị màu mới: 29](#_Toc199347246)

[2. Assignment 9: Kiểm thử các thuật toán sắp xếp 30](#_Toc199347247)

[2.1. Mô tả bài toán: 30](#_Toc199347248)

[2.2. Thuật toán sử dụng: 31](#_Toc199347249)

[2.2.1. Thuật toán 1 31](#_Toc199347250)

[2.2.2. Thuật toán 2 31](#_Toc199347251)

[2.2.3. Thuật toán 3 31](#_Toc199347252)

[2.3. Đoạn mã lệnh và giải thích: 32](#_Toc199347253)

[2.3.1. Khởi tạo các mảng ban đầu: 32](#_Toc199347254)

[2.3.2. Khởi tạo ngẫu nhiên các giá trị màu: 33](#_Toc199347255)

[2.3.3. Đầu vào và xử lý đầu vào: 35](#_Toc199347256)

[2.3.4. Xử lý gán màu và hiển thị màu: 37](#_Toc199347257)

[2.3.5. Thắng trò chơi: 41](#_Toc199347258)

[2.4. Kết quả chạy mẫu: 42](#_Toc199347259)

# **Assignment 6: Mô phỏng ổ đĩa RAID 5**

## **Mô tả bài toán:**

Hệ thống ổ đĩa RAID5 cần tối thiểu 3 ổ đĩa cứng, trong đó phần dữ liệu parity sẽ được chứa lần lượt lên 3 ổ đĩa như trong hình bên. Hãy viết chương trình mô phỏng hoạt động của RAID 5 với 3 ổ đĩa, với giả định rằng, mỗi block dữ liệu có 4 kí tự. Giao diện như trong minh họa dưới. Giới hạn chuỗi kí tự nhập vào có độ dài là bội của 8.

Trong ví dụ sau, chuỗi kí tự nhập vào từ bàn phím (DCE.\*\*\*\*ABCD1234HUSTHUST) sẽ được chia thành các block 4 byte. Block 4 byte đầu tiên “DCE.” sẽ được lưu trên Disk 1, Block 4 byte tiếp theo “\*\*\*\*” sẽ lưu trên Disk 2, dữ liệu trên Disk 3 sẽ là 4 byte parity được tính từ 2 block đầu tiên với mã ASCII là 6e=’D’ xor ‘\*’ ; 69=’C’ xor ‘\*’; 6f=’E’ xor ‘\*’ ; 04=’.’ xor ‘\*’

A close-up of a computer code

AI-generated content may be incorrect.

Sơ đồ lưu trữ:

|  |  |  |  |
| --- | --- | --- | --- |
| **Stripe** | **Disk 1** | **Disk 2** | **Disk 3** |
| 1 | Data Block 1 | Data Block 2 | Parity (1,2) |
| 2 | Data Block 3 | Parity (3,4) | Data Block 4 |
| 3 | Parity (5,6) | Data Block 5 | Data Block 6 |
| ... | ... | ... | ... |

## **Thuật toán sử dụng:**

### Khởi tạo

* Định nghĩa các chuỗi thông báo, vùng nhớ cho đĩa (disk1, disk2, disk3), mảng lưu trữ parity (array), và buffer cho chuỗi nhập liệu (string).
* Thiết lập các con trỏ thanh ghi ban đầu cho các vùng nhớ đĩa và mảng parity.

### Nhập liệu và kiểm tra độ dài :

* Hiển thị lời nhắc "Nhap chuoi ki tu : ".
* Đọc chuỗi ký tự từ người dùng.
* Kiểm tra độ dài chuỗi:

+ Đếm số ký tự trong chuỗi (không bao gồm ký tự xuống dòng \n).

+ Nếu chuỗi rỗng (chỉ có \n) hoặc độ dài không phải là bội số của 8, hiển thị thông báo lỗi "Do dai chuoi khong hop le! Chieu dai cua chuoi phai chia het cho 8. Hay nhap lai." và quay lại bước nhập liệu.

+ Nếu độ dài hợp lệ, tiếp tục.

### Thuật toán 3: Xử lý và phân phối dữ liệu (RAID 5 Logic):

* Chương trình xử lý dữ liệu theo từng cụm 24 byte (tương ứng với 3 stripe, mỗi stripe xử lý 8 byte dữ liệu đầu vào).
* Vòng lặp chính (nextloop) sẽ lặp qua các cụm 24 byte này. Bên trong mỗi cụm, có 3 giai đoạn (block1, block2, block3) tương ứng với 3 cách phân phối parity:
* Giai đoạn 1 (block1): Parity trên Disk 3

+ Đọc 4 ký tự đầu tiên của 8 byte hiện tại vào disk1.

+ Đọc 4 ký tự tiếp theo của 8 byte hiện tại vào disk2.

+ Tính toán parity: Với mỗi cặp ký tự tương ứng từ disk1 và disk2, + thực hiện phép XOR. Lưu 4 byte kết quả vào mảng array.

Hiển thị nội dung disk1 (dạng ký tự), disk2 (dạng ký tự), và parity từ array (dạng hexa, sử dụng thủ tục HEX).

* Giai đoạn 2 (block2): Parity trên Disk 2

+ Con trỏ chuỗi đầu vào được dịch chuyển 8 byte so với đầu cụm 24 byte.

+ Đọc 4 ký tự đầu tiên của 8 byte tiếp theo vào disk1.

+ Đọc 4 ký tự tiếp theo của 8 byte này vào disk3.

+ Tính toán parity giữa dữ liệu trên disk1 và disk3. Lưu 4 byte kết quả vào mảng array.

+ Hiển thị nội dung disk1 (dạng ký tự), parity từ array (dạng hexa) cho vị trí Disk 2, và disk3 (dạng ký tự).

* Giai đoạn 3 (block3): Parity trên Disk 1

+ Con trỏ chuỗi đầu vào được dịch chuyển 16 byte so với đầu cụm 24 byte.

+ Đọc 4 ký tự đầu tiên của 8 byte cuối cùng trong cụm 24 byte vào disk2.

+ Đọc 4 ký tự tiếp theo của 8 byte này vào disk3.

+ Tính toán parity giữa dữ liệu trên disk2 và disk3. Lưu 4 byte kết quả vào mảng array.

+ Hiển thị parity từ array (dạng hexa) cho vị trí Disk 1, nội dung disk2 (dạng ký tự), và disk3 (dạng ký tự).

* Sau khi xử lý một cụm 24 byte (3 stripe), con trỏ chuỗi đầu vào (s0) được cập nhật để trỏ đến cụm 8 byte tiếp theo cho block1 của chu kỳ mới (nếu còn dữ liệu).

### Thủ tục HEX (Chuyển đổi sang Hexa):

* Nhận một byte dữ liệu trong thanh ghi s8.
* Chuyển đổi byte này thành 2 ký tự ASCII biểu diễn dạng hexa.
* Ví dụ: byte 0x6E sẽ được chuyển thành hai ký tự '6' và 'e'.
* In hai ký tự này ra màn hình.
  + 1. Lặp lại hoặc kết thúc:
* Sau khi xử lý toàn bộ chuỗi đầu vào, chương trình hiển thị một dòng gạch ngang phân cách.
* Hỏi người dùng "Try another string?".
* Nếu người dùng chọn "Yes", xóa nội dung chuỗi cũ và quay lại bước Nhập liệu.
* Nếu người dùng chọn "No" hoặc "Cancel", kết thúc chương trình.

## **Đoạn mã lệnh và giải thích**

### Khai báo dữ liệu (.data)

|  |
| --- |
| .data  prompt: .asciz "Nhap chuoi ki tu : "  # ASCII into hexa  hex: .byte '0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'  disk1: .space 4  disk2: .space 4  disk3: .space 4  array: .space 32 # Store parities (results for data XOR)  string: .space 5000 # Input string  newline: .asciz "\n" # Ký tự xuống dòng  error\_message: .asciz "Do dai chuoi khong hop le! Chieu dai cua chuoi phai chia het cho 8. Hay nhap lai.\n"  disk: .asciz " Disk 1 Disk 2 Disk 3\n"  msg1: .asciz " -------------- -------------- --------------\n"  msg2: .asciz "| "  msg3: .asciz " | "  msg4: .asciz "[[ "  msg5: .asciz "]] "  comma: .asciz ","  message: .asciz "Try another string?" |

- Giải thích: Đoạn này khai báo các chuỗi hằng, bảng chuyển đổi hexa, và các vùng nhớ sẽ được sử dụng để lưu trữ dữ liệu tạm thời cho các đĩa, parity, và chuỗi đầu vào. array: .space 32 có vẻ hơi lớn so với nhu cầu thực tế của mỗi lần tính parity (chỉ cần 4 byte), nhưng đủ để chứa parity cho vài stripe nếu cần mở rộng.

### Main và input

|  |
| --- |
| .text  main: # Bắt đầu của chương trình chính  la s1, disk1 # s1 = address of disk 1  la s2, disk2 # s2 = address of disk 2  la s3, disk3 # s3 = address of disk 3  la a2, array # Address of parities    j input  nop    input: # Bắt đầu phần nhập liệu  li a7, 4 # Print " Nhap chuoi ky tu"  la a0, prompt  ecall    li a7, 8 # Get string (Syscall num 8)  la a0, string  li a1, 1000 # maximum number of characters to read  ecall    mv s0, a0 # s0 = address of input string |

- Giải thích:

- main khởi tạo các thanh ghi s1, s2, s3 trỏ tới vùng nhớ của disk1, disk2, disk3 và a2 trỏ tới array (dùng để lưu các byte parity).

- input hiển thị lời nhắc và đọc chuỗi ký tự từ người dùng, lưu địa chỉ bắt đầu của chuỗi vào s0.

### Kiểm tra độ dài chuỗi (length, check\_char, test\_length, error)

|  |
| --- |
| # ---------------- Check whether input string's length is multiple of 8 ----------------  length: # Kiểm tra độ dài chuỗi  addi t3, zero, 0 # t3 = length  addi t0, zero, 0 # t0 = index  check\_char:  # Check \n?  add t1, s0, t0 # t1 = address of string[i]  lb t2, 0(t1) # t2 = string[i]  li s4, 10 # '\n' = 10 ASCII  beq t2, s4, test\_length # if string[i] = '\n', then jump test\_length  nop    addi t3, t3, 1 # length++  addi t0, t0, 1 # index++  j check\_char  nop    test\_length: # Kiểm tra tính hợp lệ của độ dài chuỗi  mv t5, t3 # t5 = string length  beq t0, zero, error # If only '\n' -> error    andi t1, t3, 0x0000000f # t1 = (t3 & 0xF) last byte  bne t1, zero, test1 # if (t3 & 0xF) != 0, then jump 'test1'  j input\_prompt # else jump 'block1'  nop  test1:  li s11, 8 # s11 = 8  beq t1, s11, input\_prompt # if (t3 & 0xF) == 8, then jump 'block1'  j error # else jump 'error'  nop    error: # Xử lý lỗi độ dài không hợp lệ  li a7, 4 # Print error\_message  la a0, error\_message  ecall    j input  nop    input\_prompt:  li a7, 4  la a0, disk  ecall  li a7, 4  la a0, msg1  ecall  j block1 |

- Giải thích:

- length và check\_char lặp qua chuỗi đầu vào để tính độ dài thực tế (t3) cho đến khi gặp ký tự \n.

- test\_length kiểm tra:

+ Nếu độ dài là 0 (chỉ có \n), báo lỗi.

+ Sử dụng phép toán bitwise andi t1, t3, 0x0000000f để lấy 4 bit cuối của độ dài. Nếu kết quả này là 0 (ví dụ: độ dài 16, 32,...) hoặc 8 (ví dụ: độ dài 8, 24,...), thì độ dài được coi là hợp lệ (là bội của 8). Nếu không, nhảy đến error.

+ Nếu độ dài hợp lệ, in ra tiêu đề bảng Disk và nhảy đến block1 để bắt đầu xử lý.

- error in thông báo lỗi và quay lại input.

### Thủ tục HEX (Chuyển byte sang 2 ký tự Hexa ASCII)

|  |
| --- |
| HEX: # Nhãn: Thủ tục chuyển đổi một byte sang 2 ký tự hexa ASCII  # -------------------- Get parities --------------------  # Đầu vào s8 chứa byte parity, chuyển từ số sang hexa (ASCII)  li t4, 7 # Khởi tạo biến đếm t4 = 7 (cho 8 nibble, nhưng chỉ in 2 cuối)    loopH:  blt t4, zero, endloopH # t4 < 0 -> endloop  slli s6, t4, 2 # s6 = t4\*4  srl a0, s8, s6 # a0 = s8 >> s6  andi a0, a0, 0x0000000f # Get the last byte of a0  la s7, hex # s7 = adrress of hex  add s7, s7, a0  li a4, 1  bgt t4, a4, nextc # if t4 > 1 , jump to nextC  lb a0, 0(s7) # Print hex[a0]  li a7, 11  ecall  nextc:  addi t4, t4, -1 # t4 --  j loopH  nop  endloopH:  jr ra  nop |

- Giải thích:

- Thủ tục này nhận một byte trong s8.

- Nó tách byte đó thành hai nibble (4 bit).

- Nibble cao (4 bit đầu): Dịch phải s8 4 vị trí, sau đó dùng andi với 0x0f để lấy giá trị.

- Nibble thấp (4 bit cuối): Dùng andi với 0x0f trực tiếp trên s8.

- Mỗi giá trị nibble (0-15) được dùng làm chỉ số để tra cứu trong bảng hex và lấy ký tự ASCII tương ứng ('0'-'9', 'a'-'f').

- In ra 2 ký tự hexa này.

- jr ra quay lại nơi gọi.

*- Lưu ý:* Phần mã gốc của HEX có vẻ được thiết kế để xử lý một word 32-bit và in ra 8 ký tự hexa, nhưng trong ngữ cảnh này nó chỉ được gọi để in 2 ký tự hexa cuối cùng của byte trong s8 (do bgt t4, a4, nextc với a4=1). Đoạn mã giải thích ở trên đã được đơn giản hóa cho mục đích chuyển đổi 1 byte thành 2 ký tự hexa, điều này phù hợp hơn với cách nó được sử dụng. Nếu giữ nguyên mã gốc, t4 nên được khởi tạo là 1 để nó thực hiện 2 lần lặp cho 2 nibble.

### Mô phỏng RAID 5 (block1, block2, block3)

|  |
| --- |
| #------------------------------ RAID5 SIMULATION------------------------------------  RAID5: # Đánh dấu bắt đầu phần mô phỏng RAID5 (không được nhảy trực tiếp đến)  # Block 1 : byte parity is stored in disk 3  # Block 2 : byte parity is stored in disk 2  # Block 3 : byte parity is stored in disk 1  block1:  # Function block1: First 2 4-byte blocks are stored in disk1, disk2; parity is stored in disk3  addi t0, zero, 0  addi s9, zero, 0  addi s8, zero, 0  la s1, disk1  la s2, disk2  la a2, array    print11: # In phần mở đầu cho Disk 1  li a7, 4  la a0, msg2  ecall    b11: # Vòng lặp xử lý byte cho block1  # Store into disk1  lb t1, 0(s0) # t1 = first value of input string  addi t3, t3, -1 # t3 = length - 1  sb t1, (s1) # store t1 into disk1  b12: # Xử lý byte thứ hai cho cặp  # Store ịnto disk2  addi s5, s0, 4 # s5 = s0 + 4  lb t2, 0(s5) # t2 = string[5]  addi t3, t3, -1 # t3 = t3 - 1  sb t2, 0(s2) # store t2 into disk2  b13: # Tính và lưu parity  # Store XOR result into disk3  xor a3, t1, t2 # a3 = t1 xor t2  sw a3, 0(a2) # Store a3 into a2  addi a2, a2, 4 # Parity string  addi t0, t0, 1 # Next char  addi s0, s0, 1 # Eliminate considered char, eg : "D"  addi s1, s1, 1 # Address of disk 1 + 1  addi s2, s2, 1 # Address of disk 2 + 1  li a6, 3 # a6 = 3  bgt t0, a6, reset # 4 byte are considered --> reset disk  j b11  nop  reset: # Reset con trỏ buffer disk để chuẩn bị in  la s1, disk1  la s2, disk2    print12: # Vòng lặp in nội dung buffer disk1  lb a0, 0(s1) # Print each char in disk1  li a7, 11 # syscall 11 (print char)  ecall  addi s9, s9, 1 # Tăng biến đếm s9 (số ký tự đã in của disk1)  addi s1, s1, 1 # Tăng con trỏ buffer disk1  bgt s9, a6, next11 # Print 4 times --> end priting disk1  j print12  nop    next11: # Chuẩn bị in disk2  li a7, 4  la a0, msg3  ecall  li a7, 4  la a0, msg2  ecall    print13: # Vòng lặp in nội dung buffer disk2  lb a0, 0(s2) # Nạp byte từ buffer disk2 (s2) vào a0  li a7, 11 # syscall 11 (print char)  ecall  addi s8, s8, 1 # Tăng biến đếm s8 (số ký tự đã in của disk2)  addi s2, s2, 1 # Tăng con trỏ buffer disk2 (s2)  bgt s8, a6, next12 # Print 4 times --> end printing disk2  j print13  nop    next12: # Chuẩn bị in parity (disk3)  li a7, 4  la a0, msg3  ecall  li a7, 4  la a0, msg4  ecall  la a2, array # a2 = address of parity string[i]  addi s9, zero, 0 # Reset biến đếm s9 = 0 (cho việc in parity)    print14: # Convert parity string --> ASCII and print  lb s8, 0(a2) # s8 = adress of parity string[i]  jal HEX  nop  li a7, 4  la a0, comma  ecall    addi s9, s9, 1 # Parity string's index + 1  addi a2, a2, 4 # Tăng con trỏ mảng parity (a2) lên 4 byte  li a5, 2 # Nạp giá trị 2 vào a5 (để in 3 dấu phẩy cho 4 parity)  bgt s9, a5, endisk1 # Print first 3 parities with ','  j print14  endisk1: # In byte parity cuối cùng (không có dấu phẩy sau)  lb s8, 0(a2) # Nạp byte parity cuối cùng vào s8  jal HEX  nop  li a7, 4  la a0, msg5  ecall    li a7, 4  la a0, newline  ecall  beq t3, zero, exit1 # If string length = 0 --> exit  j block2 # else --> block2  nop |

- Giải thích block1:

- Vòng lặp b11 đến b13:

+ Đọc 1 byte từ s0 (chuỗi đầu vào) vào disk1.

+ Đọc 1 byte từ s0+4 (chuỗi đầu vào) vào disk2.

+ Tính XOR của 2 byte này, lưu kết quả vào array.

+ Tăng con trỏ s0, s1 (disk1), s2 (disk2), a2 (array).

+ Biến t3 (độ dài còn lại của chuỗi) được giảm 2 lần cho mỗi cặp byte xử lý (1 lần ở b11, 1 lần ở b12 trong mã gốc). Điều này có nghĩa là sau khi xử lý 4 cặp byte (1 stripe), t3 sẽ giảm đi 8.

- Sau khi lặp 4 lần (xử lý 4 byte cho mỗi block), các con trỏ s1, s2 được reset.

- In nội dung disk1 (4 ký tự).

- In nội dung disk2 (4 ký tự).

- In 4 byte parity từ array, mỗi byte được chuyển sang hexa bằng HEX và cách nhau bằng dấu phẩy (trừ byte cuối).

- Con trỏ s0 đã tự động dịch chuyển 4 byte trong quá trình đọc vào disk1. *Quan trọng:* Để block2 bắt đầu từ dữ liệu mới, s0 cần được điều chỉnh thêm ở đầu block2 để bỏ qua 4 byte đã được đọc vào disk2 trong block1. Mã gốc thực hiện điều này.

|  |
| --- |
| #----------------------------------------  block2: # Funtion block2: Next 2 4-byte blocks are stored in disk1, disk3; parity is stored in disk2  la a2, array # Nạp địa chỉ mảng parity 'array' vào a2 (parity cho Disk2)  la s1, disk1 # Nạp địa chỉ buffer disk1 vào s1  la s3, disk3 # Nạp địa chỉ buffer disk3 vào s3  addi s0, s0, 4 # Tăng con trỏ chuỗi đầu vào (s0) lên 4 (bỏ qua 4 byte đã xử lý ở block1 cho disk1/disk2)  addi t0, zero, 0 # Reset biến đếm vòng lặp t0 = 0    print21: # print "| "  li a7, 4  la a0, msg2  ecall  b21: # Store 4 bytes into disk1  lb t1, 0(s0) # Nạp byte từ chuỗi đầu vào (s0) vào t1 (cho Disk1)  addi t3, t3, -1 # string\_length --  sb t1, 0(s1) # Lưu byte t1 vào buffer disk1  b23: # Store next 4 bytes into disk3  addi s5, s0, 4 # string addr + 4  lb t2, 0(s5)  addi t3, t3, -1 # length --  sb t2, 0(s3)    b22: # Store XOR result into disk2  xor a3, t1, t2 # Tính XOR của t1 và t2 -> a3 (parity)  sw a3, 0(a2) # Lưu byte parity a3 (như word) vào mảng 'array' (cho Disk2)  addi a2, a2, 4 # Tăng con trỏ mảng parity (a2)  addi t0, t0, 1 # Tăng biến đếm vòng lặp t0  addi s0, s0, 1 # Tăng con trỏ chuỗi đầu vào (s0)  addi s1, s1, 1 # Tăng con trỏ buffer disk1 (s1)  addi s3, s3, 1 # Tăng con trỏ buffer disk3 (s3)  bgt t0, a6, reset2 # Nếu t0 > 3, nhảy đến 'reset2'  j b21  nop  reset2: # Reset disks  la s1, disk1 # Nạp lại địa chỉ buffer disk1 vào s1  la s3, disk3 # Nạp lại địa chỉ buffer disk3 vào s3  addi s9, zero, 0 # Reset biến đếm s9 = 0 (cho việc in)    print22: # In nội dung buffer disk1  lb a0, 0(s1) # Nạp byte từ buffer disk1 (s1) vào a0  li a7, 11 # syscall 11 (print char)  ecall  addi s9, s9, 1 # Tăng biến đếm s9  addi s1, s1, 1 # Tăng con trỏ buffer disk1 (s1)  bgt s9, a6, next21 # Nếu s9 > 3, nhảy đến 'next21'  j print22  nop    next21: # Chuẩn bị in parity (Disk2)  li a7, 4  la a0, msg3  ecall  la a2, array # Nạp lại địa chỉ mảng parity 'array' vào a2  addi s9, zero, 0 # Reset biến đếm s9 = 0  li a7, 4  la a0, msg4  ecall    print23: # Vòng lặp in các byte parity (Disk2)  lb s8, 0(a2) # Nạp byte parity từ mảng (a2) vào s8  jal HEX  nop  li a7, 4  la a0, comma  ecall  addi s9, s9, 1 # Tăng biến đếm s9  addi a2, a2, 4 # Tăng con trỏ mảng parity (a2)  bgt s9, a5, next22 # Nếu s9 > 2, nhảy đến 'next22'  j print23  nop    next22: # In byte parity cuối cùng (Disk2)  lb s8, (a2) # Nạp byte parity cuối cùng vào s8  jal HEX  nop    li a7, 4  la a0, msg5  ecall    li a7, 4  la a0, msg2  ecall  addi s8, zero, 0 # Reset biến đếm s8 = 0 (cho việc in Disk3)    print24: # Vòng lặp in nội dung buffer disk3  lb a0, 0(s3) # Nạp byte từ buffer disk3 (s3) vào a0  li a7, 11  ecall  addi s8, s8, 1 # Tăng biến đếm s8  addi s3, s3, 1 # Tăng con trỏ buffer disk3 (s3)  bgt s8, a6, endisk2 # Nếu s8 > 3, nhảy đến 'endisk2'  j print24  nop  endisk2: # Kết thúc in block2  li a7, 4  la a0, msg3  ecall  li a7, 4  la a0, newline  ecall  beq t3, zero, exit1  j block3  nop |

- Giải thích block2:

* + addi s0, s0, 4: Quan trọng! Con trỏ s0 được dịch 4 byte để bỏ qua phần dữ liệu đã được đọc vào disk2 ở block1. Bây giờ s0 trỏ đến đầu của segment 8 byte tiếp theo trong chuỗi đầu vào.
  + Vòng lặp b21 đến b22:

+ Đọc 1 byte từ s0 vào disk1.

+ Đọc 1 byte từ s0+4 vào disk3.

+ Tính XOR, lưu vào array (đây sẽ là parity cho Disk2).

+ t3 lại giảm 2 cho mỗi cặp byte, tổng cộng 8 cho stripe này.

* + In disk1 (dữ liệu), array (parity cho vị trí Disk2), disk3 (dữ liệu).
  + Con trỏ s0 lại dịch thêm 4 byte.

|  |
| --- |
| #--------------------------------  block3: # Funtion block3: Next 2 4-byte blocks are stored in disk2, disk3; parity is stored in disk1  la a2, array  la s2, disk2  la s3, disk3  addi s0, s0, 4 # Tăng con trỏ chuỗi đầu vào (s0) lên 4  addi t0, zero, 0 # Reset biến đếm vòng lặp t0 = 0  print31: # Print '[['  li a7, 4  la a0, msg4  ecall  b32: # Byte stored in Disk 2  lb t1, 0(s0) # Nạp byte từ chuỗi đầu vào (s0) vào t1 (cho Disk2)  addi t3, t3, -1 # string\_length --  sb t1, 0(s2) # Lưu byte t1 vào buffer disk2  b33: # Store in Disk 3  addi s5, s0, 4 # Tính địa chỉ byte tương ứng (s0+4)  lb t2, 0(s5) # Nạp byte từ (s0+4) vào t2 (cho Disk3)  addi t3, t3, -1 # Giảm độ dài còn lại t3  sb t2, 0(s3) # Lưu byte t2 vào buffer disk3    b31: # Store XOR result into disk1  xor a3, t1, t2 # Tính XOR của t1 và t2 -> a3 (parity)  sw a3, 0(a2) # Lưu byte parity a3 (như word) vào mảng 'array' (cho Disk1)  addi a2, a2, 4 # Tăng con trỏ mảng parity (a2)  addi t0, t0, 1 # Tăng biến đếm vòng lặp t0  addi s0, s0, 1 # Tăng con trỏ chuỗi đầu vào (s0)  addi s2, s2, 1 # Tăng con trỏ buffer disk2 (s2)  addi s3, s3, 1 # Tăng con trỏ buffer disk3 (s3)  bgt t0, a6, reset3 # Nếu t0 > 3, nhảy đến 'reset3'  j b32  nop  reset3: # Reset con trỏ buffer disk để chuẩn bị in  la s2, disk2  la s3, disk3  la a2, array  addi s9, zero, 0 # Index - Reset biến đếm s9 = 0 (cho việc in)    print32: # Vòng lặp in các byte parity (Disk1)  lb s8, 0(a2) # Nạp byte parity từ mảng (a2) vào s8  jal HEX  nop  li a7, 4  la a0, comma  ecall    addi s9, s9, 1 # Tăng biến đếm s9  addi a2, a2, 4 # Tăng con trỏ mảng parity (a2)  bgt s9, a5, next31 # Nếu s9 > 2, nhảy đến 'next31'  j print32  nop    next31: # In byte parity cuối cùng (Disk1)  lb s8, 0(a2) # Nạp byte parity cuối cùng vào s8  jal HEX  nop  li a7, 4  la a0, msg5  ecall  li a7, 4  la a0, msg2  ecall  addi s9, zero, 0    print33: # Vòng lặp in nội dung buffer disk2  lb a0, 0(s2) # Nạp byte từ buffer disk2 (s2) vào a0  li a7, 11 # syscall 11 (print char)  ecall  addi s9, s9, 1 # Tăng biến đếm s9  addi s2, s2, 1 # Tăng con trỏ buffer disk2 (s2)  bgt s9, a6, next32 # Nếu s9 > 3, nhảy đến 'next32'  j print33  nop    next32: # Chuẩn bị in Disk3  addi s9, zero, 0 # Reset biến đếm s9 (không thực sự dùng ngay sau)  addi s8, zero, 0 # Reset biến đếm s8 (cho việc in Disk3)  li a7, 4  la a0, msg3  ecall  li a7, 4  la a0, msg2  ecall  print34: # Vòng lặp in nội dung buffer disk3  lb a0, (s3) # Nạp byte từ buffer disk3 (s3) vào a0  li a7, 11 # syscall 11 (print char)  ecall  addi s8, s8, 1 # Tăng biến đếm s8  addi s3, s3, 1 # Tăng con trỏ buffer disk3 (s3)  bgt s8, a6, endisk3 # a6 is still 3 - Nếu s8 > 3, nhảy đến 'endisk3'  j print34  nop  endisk3: # Kết thúc in block3  li a7, 4  la a0, msg3  ecall    li a7, 4  la a0, newline  ecall  beq t3, zero, exit1 # Nếu độ dài còn lại t3 = 0, nhảy đến 'exit1'  j nextloop # Nhảy đến 'nextloop' để xử lý cụm 3 block tiếp theo  nop |

- Giải thích block3:

* + addi s0, s0, 4: Con trỏ s0 lại được dịch 4 byte, bỏ qua dữ liệu đã đọc vào disk3 ở block2. s0 trỏ đến đầu segment 8 byte thứ ba trong cụm 24 byte.
  + Vòng lặp b32 đến b31:

+ Đọc 1 byte từ s0 vào disk2.

+ Đọc 1 byte từ s0+4 vào disk3.

+ Tính XOR, lưu vào array (parity cho Disk1).

+ t3 giảm 8 cho stripe này.

* + In array (parity cho vị trí Disk1), disk2 (dữ liệu), disk3 (dữ liệu).
  + Con trỏ s0 lại dịch thêm 4 byte.
    1. Vòng lặp và Kết thúc (nextloop, exit1, ask, clear, exit)

|  |
| --- |
| #-----------End first 6 4-byte blocks----------------------------- (Thực ra là 3 block 8 byte = 24 byte)  #-----------Next 6 4-byte blocks---------------------------------- (Tương tự, là các cụm 24 byte tiếp theo)  nextloop: # Bắt đầu một chu kỳ mới của 3 block RAID  addi s0, s0, 4 # Tăng con trỏ chuỗi đầu vào (s0) lên 4 (để block1 tiếp theo bắt đầu từ đoạn 8 byte mới)  j block1  nop    exit1: # Print ------ and end RAID simulation  li a7, 4  la a0, msg1  ecall  j ask  nop    #--------------------END RAID 5 SIMULATION-------------------------  #--------------------TRY ANOTHER STRING----------------------------  ask: # Hỏi người dùng có muốn thử chuỗi khác không  li a7, 50 # syscall 50 (message dialog yes/no/cancel)  la a0, message  ecall  beq a0, zero, clear # a0: 0 = YES; 1 = NO; 2 = CANCEL - Nếu a0 = 0 (Yes), nhảy đến 'clear'  nop  j exit # Nếu không phải Yes (No hoặc Cancel), nhảy đến 'exit'  nop    # clear function: Return string to original state  clear: # Xóa nội dung chuỗi đầu vào cũ để chuẩn bị nhập mới  la s0, string # Nạp địa chỉ của buffer 'string' vào s0  add s3, s0, t5 # Tính địa chỉ cuối của chuỗi cũ (string + length\_old\_string)  li t1, 0 # Set t1 = 0  goAgain: # Return string to empty state to start again  sb t1, (s0) # Ghi byte null (t1) vào địa chỉ s0 trong buffer 'string'  nop  addi s0, s0, 1 # Tăng con trỏ s0 lên byte tiếp theo  bge s0, s3, input # Nếu s0 >= s3 (đã xóa hết), nhảy đến 'input'  nop  j goAgain  nop  #-----Exit program----------  exit:  li a7, 10  ecall |

- Giải thích:

* + nextloop: Sau khi hoàn thành 3 block (tức là 3 stripes, xử lý 24 byte dữ liệu đầu vào), s0 được dịch thêm 4 byte. Điều này là để s0 trỏ đến điểm bắt đầu của dữ liệu cho disk1 trong block1 của chu kỳ tiếp theo. Chu trình 3-block (block1, block2, block3) lặp lại.
  + exit1: Khi t3 (độ dài còn lại) bằng 0, in dòng gạch ngang cuối cùng và nhảy đến ask.
  + ask: Sử dụng syscall 50 để hiển thị hộp thoại Yes/No. Nếu Yes (a0=0), nhảy đến clear. Ngược lại, thoát.
  + clear và goAgain: Xóa nội dung của string bằng cách ghi các byte null vào đó, dựa trên độ dài t5 đã lưu trước đó. Sau đó quay lại input.
  + exit: Kết thúc chương trình.

## Chạy thử mã lệnh và kết quả:

### Kết quả hiển thị ảnh:

A screenshot of a computer

AI-generated content may be incorrect.

### Kết quả hiển thị ảnh đã xóa màu cho các chữ cái DCE trong ảnh:

A screenshot of a computer

AI-generated content may be incorrect.

### Kết quả hiển thị ảnh đã hoán đổi vị trí các chữ cái:

A screenshot of a computer

AI-generated content may be incorrect.

### Kết quả hiển thị ảnh các chữ cái DCE với giá trị màu mới:

A screen shot of a computer

AI-generated content may be incorrect.

# Assignment 9: Kiểm thử các thuật toán sắp xếp

## Mô tả bài toán:

- Tìm hiểu hàm hệ thống để đọc, ghi file văn bản.

- Cho trước file văn bản chứa các số nguyên ngẫu nhiên, phân cách bởi dấu cách. Số lượng phần tử có thể lớn, tối đa 10000 phần tử.

- Tạo giao diện cho phép người dùng nhập tên file để mở, các số trong file được đọc và lưu vào bộ nhớ.

- Người dùng chọn thuật toán sắp xếp cần thực hiện (Nổi bọt, Chèn, Lựa chọn). Được điểm cộng nếu thực hiện thêm các thuật toán khác.

- Chương trình chạy thuật toán và in ra thời gian thực hiện.

- Chương trình ghi kết quả sắp xếp vào file kết quả.

## Thuật toán sử dụng:

### Thuật toán 1

* Sử dụng 2 mảng, mỗi mảng 8 phần tử, một mảng lưu giá trị hexa của màu, một mảng lưu số lần màu đấy còn có thể được xuất hiện. Số thứ tự của màu trong mảng lưu giá trị hexa chính là số thứ tự của số lần màu đến còn có thể được xuất hiện trong mảng còn lại.
* Sử dụng một mảng để lưu màu (các màu ở mỗi lần sinh khác nhau sẽ ở vị trí khác nhau trong mảng – tăng dần theo số lần sinh)
* Sử dụng hàm hệ thống để sinh tự động số thứ tự của màu trong mảng để lấy được màu từ mảng rồi lưu vào mảng lưu màu. Sau đó giảm giá trị của mảng lưu số lần xuất hiện còn lại ở vị trí màu đấy. => Tránh lặp lại màu quá 2 lần.
* Thực hiện vòng lặp cho đến khi mảng lưu màu có đủ 16 màu.

### Thuật toán 2

* Nhấn phím trên keyboard của digital lab sim để chọn ô trên grid (4x4).
* Sử dụng polling để nhận tín hiệu phím được bấm.
* Trong lúc đợi phím được nhả thì kiểm tra lần lượt từng hàng trên bàn phím digital lab sim để xem xem phím ở hàng nào và cột nào được bấm.
* Nếu phím bấm ánh xạ sang bit map có ô đấy đã mở thì báo lỗi và bấm lại.

### Thuật toán 3

* Sau khi lấy được phím bấm thì thực hiện so sánh kiểm tra xem thẻ ở vị trí bấm đấy đã mở hay chưa.
  + Lấy màu được lưu từ mảng lưu màu với giá trị màu được lấy ở vị trí mà nút bấm ánh xạ tới (0 - 15).
  + Lấy màu từ vị trí bấm ở trên bit map để so sánh với màu được lấy từ mảng. (cùng vị trí ấy mà chỗ đấy đã có màu tức là ô đấy đã được mở).
* Nếu tại vị trí bấm đấy mà chưa mở thì thực hiện lưu màu và hiển thị màu trên bit map.
* Lưu chỉ số thẻ và màu thẻ lại nếu đó là lần đầu mở thẻ trong 2 lần của 1 lượt.
* Thực hiện lấy màu của lần bấm thứ hai của lượt chơi đó (cũng kiểm tra có chọn phải ô đã mở chưa)
  + So sánh màu ở 2 lần mở. Nếu giống thì chuyển sang lượt chơi tiếp theo, nếu không giống thì đóng 2 thẻ mở lại.
* Nếu đã mở đủ cả 8 cặp màu thì dừng chương trình.

## Đoạn mã lệnh và giải thích:

### Khởi tạo các mảng ban đầu:

|  |
| --- |
| .eqv IN\_ADDRESS\_HEXA\_KEYBOARD 0xFFFF0012  .eqv OUT\_ADDRESS\_HEXA\_KEYBOARD 0xFFFF0014  .eqv MONITOR\_SCREEN 0x10040000  .eqv PAIR\_COUNT 8  .eqv TILE\_COUNT 16  .data  info: .asciz "Hãy lựa chọn thẻ\n"  error: .asciz "Thẻ đã được lật! Hãy chọn thẻ khác\n"  finish: .asciz "Bạn đã thắng!!!\n"  Colors: .word 0x00FF0000, 0x0000FF00, 0x000000FF, 0x00FFFF00,  0x0000FFFF, 0x00FF00FF, 0x00FFFFFF, 0x00808080  Mark: .word 2, 2, 2, 2, 2, 2, 2, 2 # Mỗi màu xuất hiện 2 lần  Temp: .space 64 # Lưu màu của 16 thẻ (16\*4 byte)  FirstIdx: .word 0 # Vị trí thẻ đầu tiên  FirstCol: .word 0 # Màu thẻ đầu tiên  # Ánh xạ phím sang chỉ số thẻ  keymap:  .word 0x00000011, 0  .word 0x00000021, 1  .word 0x00000041, 2  .word 0xFFFFFF81, 3  .word 0x00000012, 4  .word 0x00000022, 5  .word 0x00000042, 6  .word 0xFFFFFF82, 7  .word 0x00000014, 8  .word 0x00000024, 9  .word 0x00000044, 10  .word 0xFFFFFF84, 11  .word 0x00000018, 12  .word 0x00000028, 13  .word 0x00000048, 14  .word 0xFFFFFF88, 15  .word 0xFFFFFFFF, -1 # Kết thúc bảng ánh xạ |

* info, error, finish lưu các chuỗi sẽ hiển thị ra màn hình
* Colors là mảng lưu giá trị hexa của các màu
* Mark là mảng lưu số lần xuất hiện của các màu đó
* Temp là mảng lưu vị trí của các màu vào mỗi thẻ
* FirstIdx lưu vị trí của màu đầu tiên được mở trong 1 lượt chơi
* FirstCol lưu giá trị hexa của màu đầu tiên được mở trong 1 lượt chơi
* keymap là mảng 2 chiều lưu giá trị là tọa độ của nút được bấm cùng với vị trí của tọa độ đó trong mảng Temp.
* ***Địa chỉ của BITMAP DISPLAY là ở heap.***

### Khởi tạo ngẫu nhiên các giá trị màu:

|  |
| --- |
| .text  start:  li t0, 0  li t1, TILE\_COUNT  la t3, Temp  # Khởi tạo bảng thẻ với màu ngẫu nhiên  random\_loop:  beq t0, t1, start\_game  li a7, 42 # random  li a1, PAIR\_COUNT # giới hạn từ 0 đến 7  ecall  la t4, Colors  la t5, Mark  slli t6, a0, 2 # offset = a0 \* 4  add t5, t5, t6  lw t2, 0(t5) # số lần còn lại của màu này  beqz t2, random\_loop  add t4, t4, t6  lw s5, 0(t4) # lấy giá trị màu  sw s5, 0(t3) # lưu vào Temp  addi t3, t3, 4  addi t2, t2, -1 # giảm số lần sử dụng màu  sw t2, 0(t5)  addi t0, t0, 1  j random\_loop |

* Khởi tạo giá trị đếm các màu đã được khởi tạo đúng (hợp lý) là 0, đếm tăng dần cho đến khi đủ 16 (TILE\_COUNT) màu đã được khởi tạo.
* Hàm hệ thống khởi tạo số nguyên ngẫu nhiên từ 0 đến bound trừ 1 (PAIR\_COUNT - 1)
* Thực hiện lấy địa chỉ màu và số lần màu đó còn có thể xuất hiện từ mảng Colors (t4 - vị trí 0) và Mark (t5 - vị trí 0) bằng cách lấy địa chỉ tại ví trí 0 cộng với 4 lần giá trị số nguyên ngẫu nhiên vừa khởi tạo (a0).
* Nếu số lần xuất hiện của màu đã là 2 thì lặp lại để lấy giá trị mới còn không thì lưu màu vào mảng Temp ở vị trí mà mỗi lần lặp sẽ tăng lên (t3), giảm số lần màu đó còn có thể xuất hiện rồi lưu vào mảng Mark, và tăng giá trị đếm lên 1.
* Thực hiện lại vòng lặp cho đến khi đủ 16 màu được khởi tạo.

### Đầu vào và xử lý đầu vào:

|  |
| --- |
| start\_game:  li s0, 0 # số cặp đã khớp  li s1, PAIR\_COUNT  li s3, 0 # lượt chọn (0 = lần 1, 1 = lần 2)  game\_loop:  beq s0, s1, game\_done  li a7, 4  la a0, info  ecall  # Đợi người chơi nhấn phím  wait\_key:  li t1, 0x01  wait\_loop:  li t2, IN\_ADDRESS\_HEXA\_KEYBOARD  li t3, OUT\_ADDRESS\_HEXA\_KEYBOARD  sb t1, 0(t2)  lb a0, 0(t3)  beqz a0, next\_row  la t4, keymap  search\_keymap:  lw t5, 0(t4)  li t6, -1  beq t5, t6, wait\_key # không hợp lệ  beq t5, a0, found\_key  addi t4, t4, 8  j search\_keymap  found\_key:  lw s4, 4(t4) # lấy chỉ số thẻ  # Đợi người chơi nhả phím  wait\_key\_release:  li t2, OUT\_ADDRESS\_HEXA\_KEYBOARD  lb t3, 0(t2)  bnez t3, wait\_key\_release  jal retrieve # kiểm tra thẻ đã mở chưa  bnez t2, invalid\_input  beq s3, zero, save\_first  j check\_match  next\_row:  slli t1, t1, 1  li t4, 0x10  blt t1, t4, wait\_loop  j wait\_key |

* Bắt đầu trò chơi, khởi tạo giá trị số cặp đã khớp là 0 (s0) và lưu tổng số cặp là 8 (PAIR\_COUNT), và lưu số lần mở trong 1 lượt chơi là 0 (s3).
* Thực hiện vòng lặp trò chơi, lặp qua từng số cặp đã khớp cho đến khi bằng tổng số cặp màu thì kết thúc trò chơi.
* Thực hiện polling, chờ người chơi bấm phím. Khởi tạo giá trị hàng kiểm tra đầu tiên là 0x01 (t1). Lấy địa chỉ đầu vào (t2) và đầu ra (t3) của bàn phím digital lab sim, truyền vào địa chỉ đầu vào giá trị hàng đang kiểm tra để lấy giá trị hàng và cột kiểm tra ra từ địa chỉ đầu ra (a0) của bàn phím digital lab sim.
  + Nếu gía trị lấy từ địa chỉ đầu ra của bàn phím digital lab sim bằng 0 có nghĩa là người dùng không bấm ở hàng đấy, nhảy đến ***next\_row*** để thực hiện tăng hàng lên gấp đôi rồi quay về ***wait\_loop*** để lấy được phím bấm.
  + Nếu giá trị lấy từ địa chỉ đầu ra của bàn phím digital lab sim khác 0 có nghĩa là người dùng đã bấm nút ở hàng đấy và chương trình bắt đầu thực hiện ánh xạ sang vị trí của bitmap.
    - Thực hiện lặp để so sánh, tìm vị trí phím được lấy ra với các giá trị của mảng 2 chiều keymap, nếu phím bấm trùng hợp với giá trị hexa map vị trí các phần tử của mảng keymap thì sẽ lưu giá trị (s4) là chỉ số của vị trí đó trên bitmap ở địa chỉ &keymap[i + 4].
    - Thực hiện tăng giá trị địa chỉ trỏ tới các phần tử của mảng keymap lên 8 để truy cập đến các giá trị keymap[i] để so sánh trong lúc lặp mà chưa tìm thấy phần tử khớp với phím được bấm.
* Thực hiện kiểm tra xem phím bấm đã được nhả ra hay chưa. Nếu phím chưa được nhả ra thì ta chưa thực hiện gán màu cũng như các hoạt động sau.

### Xử lý gán màu và hiển thị màu:

* Nếu phím đã được nhả thì bắt đầu thực hiện kiểm tra vị trí đấy đã được mở hay chưa rồi gán màu và in màu ra bit map display.

|  |
| --- |
| retrieve:  la t0, Temp  slli t1, s4, 2  add t2, t0, t1  lw a3, 0(t2) # lấy màu  li t0, MONITOR\_SCREEN  add t3, t0, t1  lw t4, 0(t3)  bnez t4, already\_chosen  sw a3, 0(t3)  li t2, 0  li t5, 1  beq s3, t5, delay\_display  ret |

* Lấy địa chỉ của mảng Temp(t0), tính địa chỉ của mảng ở vị trí nút bấm (địa chỉ là 4 bytes nên nhân vị trí nút bấm đã được ánh xạ từ keymap với 4) bằng cách lấy địa chỉ của mảng Temp cộng với giá trị của 4 lần vị trí nút bấm. Ta lấy màu (a3) đã được lưu từ quá trình khởi tạo màu ban đầu ở Temp.
* Lấy địa chỉ của màn hình bit map, tính địa chỉ của vị trí tô màu trong bit map ở vị trí nút bấm (địa chỉ là 4 bytes nên nhân vị trí nút bấm đã được ánh xạ từ keymap với 4) bằng cách lấy địa chỉ của màn hình bit map cộng với giá trị của 4 lần vị trí nút bấm. Lấy màu (t4) từ vị trí đó trên bit map, thực hiện kiểm tra xem ô đấy đã được tô màu chưa.
  + Nếu màu ở vị trí đó trên bit map mà là màu đen (giá trị 0x00000000 trong bảng màu hexa) thì thực hiện gán màu (a3) vào chính vị trí đó trên bit map và gán cờ báo lỗi (t2) bằng 0. Kiểm tra nếu đã bấm 1 lần trong lượt chơi thì nhảy đến ***delay\_display*** để trì hoãn hiển thị.

|  |
| --- |
| delay\_display:  li a7, 32  li a0, 1000 # delay 1000ms  ecall  ret |

* + - Sau khi thực hiện trễ thì nhảy ngược trở về câu lệnh sau khi gọi chương trình con ***retrieve.***
  + Nếu màu ở vị trí đó trên bit map mà không là màu đen thì nhảy đến ***already\_choosen*** rồi hiển thị thông báo lỗi.

|  |
| --- |
| already\_chosen:  li t2, 1  ret |

* + - Sau khi thực hiện gán cờ báo lỗi (t2) bằng 1 thì nhảy ngược trở về câu lệnh sau khi gọi chương trình con ***retrieve.***
* Nếu cờ báo lỗi (t2) mà khác 0 thì nhảy đến chương trình con ***invalid\_input*** để báo lỗi.

|  |
| --- |
| invalid\_input:  li a7, 55  la a0, error  li a1, 0  ecall  j game\_loop |

* Nếu cờ báo lỗi (t2) mà bằng 0 thì thực hiện so sánh số lần màu đó đã được bấm trong một lượt chơi (s3) với 0.
  + Nếu số lần đã bấm trong một lượt chơi bằng 0 thì thực hiện nhảy đến ***save\_first*** để lưu các thông số về màu đó ở lần bấm đầu tiên này.

|  |
| --- |
| save\_first:  la t0, FirstIdx  sw s4, 0(t0)  la t0, FirstCol  sw a3, 0(t0)  li s3, 1  j game\_loop |

* + - Lấy địa chỉ của ***FirstIdx*** để lưu giá trị vị trí đã được ánh xạ vào địa chỉ này rồi lấy địa chỉ của ***FirstCol*** để lưu giá trị màu ở vị trí được ánh xạ vào địa chỉ này và gán số lần đã bấm trong một lượt chơi =1.
  + Nếu số lần đã bấm trong một lượt chơi khác 0 (bằng 1) thì thực hiện nhảy đến ***check\_match*** để so sánh cặp màu trong lượt chơi.

|  |
| --- |
| check\_match:  la t0, FirstCol  lw t1, 0(t0)  xor t2, t1, a3  li s3, 0  beqz t2, matched  j not\_matched |

* + - Lấy màu được lưu từ lần đầu bấm (t1) từ địa chỉ ***FirstCol*** rồi dùng phép toán xor của màu đấy với màu vừa bấm. Nếu hai màu giống nhau thì giá trị của kết quả sẽ là 0 còn ngược lại là khác 0. Sau đó reset lại số lần bấm nút của người dùng. So sánh kết quả với 0.
      * Nếu bằng 0 thì nhảy đến nhãn ***match*** để tăng số cặp giống lên 1 đơn vị rồi quay lại lặp lấy đầu vào.

|  |
| --- |
| matched:  addi s0, s0, 1  j game\_loop |

* + - * Nếu khác 0 thì nhảy đến nhãn ***not\_match*** để xóa màu từ vị trí trước và xóa thẻ vị trí rồi nhảy lặp lại vòng lặp.

|  |
| --- |
| not\_matched:  li t0, MONITOR\_SCREEN  slli t1, s4, 2  add t1, t0, t1  sw zero, 0(t1) # xóa màu hiện tại  la t2, FirstIdx  lw t3, 0(t2)  slli t4, t3, 2  add t4, t0, t4  sw zero, 0(t4) # xóa thẻ trước đó  j game\_loop |

### Thắng trò chơi:

|  |
| --- |
| game\_done:  li a7, 55  la a0, finish  li a1, 1  ecall  li a7, 10  ecall |

* Khi mà số cặp giống nhau bằng số cặp màu thì trò chơi kết thúc, chương trình nhảy đến nhãn ***game\_done*** để in ra màn hình thông báo chiến thắng và kết thúc trò chơi.

## Kết quả chạy mẫu:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.