HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Computer Architecture Lab Final Project

Group 2

Supervised by:
Master Le Ba Vui

Teaching Assistant:
Nguyen Trung Kien

Group Members:
Hoang Ba Bao - 20226015
Dinh Ngoc Cam - 20226016

Hanoi, 18th December 2024

1. Problem 1: RAID 5 Simulation

1.1. Program Description

The RAID5 drive system requires at least 3 hard disks, in which parity data will be stored on 3 drives as shown below. Write a program to simulate the operation of RAID 5 with 3 drives, assuming each data block has 4 characters. The interface is as shown in the example below. Limit the length of the input string to a multiple of 8.

1.2. Method

The method of this program involves of generating the title for each disk, generate the upper boundary lines, generate space between title, generate space between boundary line, store input value to sp address, compute the xor value of each pair, get the least significant bit, then print them all out according to the requirements. This can be explained as follow:

1.2.1. Setup

First we initialize the value of input buffer, input information, disk titles, many kinds of line bound and spacing, comma space, symbols for printing.

1.2.2. Input handling

Then we display the input information by reading the string into the buffer, then counting the input string and checking whether its length is multiple of 8, if not we will ask the user to re enter the string.

1.2.3. Title and Upper bound generation

We now draw the title and upper bound using the prepared data segments.

1.2.4. Loop Preparing

In this step, we prepare for the stopping condition based on the length of the input string, we load all the 8 bytes-part of the input string, xor them to get the xor value. After that, we check the condition to choose the appropriate printing strategy.

1.2.5. Loop Printing

We will then start printing with load balance between 3 strategies, for each 4 bytes of input string we load them from the memory and print step by step, then we take the least significant bits of the xor result and print them out with prepared format. Then we traverse to the next row and not forget to restore the state of sp pointer and check the condition before looping again

1.2.6. Print ending bound after loop

After we have done all the printing, now we print the ending bound and exit the program

1.3. Algorithms

1.3.1. Initialization

- We setup initial parameters like the input inform, the disk title, the space and lind bound, assign rs t6 equals 3, rs s8 1, rs s9 2 for 3 strategies checking

1.3.2. Input Handling

- We count the input string length and check it remainder with 8 if it not a remainder of 8 we ask the user to re enter the value

1.3.3. Print the title upper bound

- We start print the disk title, the upper boundary line

1.3.4. Loop Start and Preparing

- We first take the length of the string divide it by 8 to get the number of line we need to print out, then we start prepare for each line

- For each line, we extract 8 bytes part in a loop and xor them bytes by bytes then store all in sp pointer address part in the data segments

- After prepare all the values, we checking the current variables t5 which is the remainder of current row index and 3 to get the appropriate printing strategies
- After that we will start printing that row

1.3.5. Loop Print

- For each printing strategy, we have different orders for printing
- For the first strategy, we will print 8 bytes part first with 4 bytes each cell, then print the xor part
- For the second strategy, we will print 4 bytes first, then print the xor part, then print remaining 4 bytes
- For the third strategy, we will print the xor part first, then print the 8 bytes part with 4 bytes each cell
- When print the xor part, we have to print the LSB, so i first and the xor part with 0x0f to get the last 4 bits, then checking whether is a number of not, if it a number i just print it out, if not i convert it to appropriate characters by add it asci code with 87. Then we extract the remained 4 bits of LSB by and it with 0xf0 and srli 4 bits before convert similarly like the first part
- For the input 8 bytes part, we print 4 bytes part a time by just using load byte and print them consecutively
- After done printing 1 line, we restore the sp pointer, increase index by 1 and move the the next line with the next 8 bytes part

1.3.6. Printing the end bound

- After done printing all the line, now we print the end bound with prepared part in the data segments and end the program

1.4. Source code

1.4.1. Setup Program

```
.data
buffer: .space 1000
input_display: .asciz "Enter the input string: "
input_warning: .asciz "Length of the string must be a multiple 8\n"

disk_title_1: .asciz "    DISK 1    "
disk_title_2: .asciz "    DISK 2    "
disk_title_3: .asciz "    DISK 3    "

begin_line_bound: .asciz "\n --------------"
space_line_bound: .asciz "        "
space_title_bound: .asciz "         "
space_large_title_bound: .asciz "          "
line_bound: .asciz " --------------"

loop_begin_first: .asciz "\n|     "
loop_begin: .asciz "|     "
loop_end: .asciz "     |"

partition_begin_first: .asciz "\n[[ "
partition_begin: .asciz "[[ "
partition_end: .asciz "]]"

comma_space: .asciz ","
```

Printing format preparation

```
.text
input:
  li a7, 4
  la a0, input_display
  ecall

  li a7, 8
  la a0, buffer
  li a1, 100
  ecall

  li t6, 3
  li s8, 1
  li s9, 2
  li t1, 10
  li t2, 0
```

String Input and constant declaration

1.4.2. Input Handling

```
  li t2, 0

count_char:
  la s1 buffer

count_char_loop:
  lb s2 0(s1)
  beq s2, t1, exit_loop
  addi t2, t2, 1
  addi s1, s1, 1
  j count_char_loop

exit_loop:
  li t4, 8
  rem t4, t2, t4
  bnez t4 re_input
  li t4, 8
  div t4, t2, t4
  j disk_title

re_input:
  li a7, 4
  la a0, input_warning
  ecall
  j input
```

Input Handling

1.4.3. Title and Upper bound generation

```
#Print the disk title
disk_title:
  li a7, 4
  la a0, disk_title_1
  ecall

  li a7, 4
  la a0, space_large_title_bound
  ecall

  li a7, 4
  la a0, disk_title_2
  ecall

  li a7, 4
  la a0, space_large_title_bound
  ecall

  li a7, 4
  la a0, disk_title_3
  ecall
```

Disk title printing

```
#Print the begin line of boundary
begin_bound:
  li a7, 4
  la a0, begin_line_bound
  ecall

  li a7, 4
  la a0, space_title_bound
  ecall

  li a7, 4
  la a0, line_bound
  ecall

  li a7, 4
  la a0, space_title_bound
  ecall

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

Upper boundary printing

1.4.4. Loop Value Preparation

```
prepare_loop:
  li t3, 0
  la s1, buffer
  j start_loop

start_loop:
  beq t3, t4, end_bound
  li s3, 0
  li s4, 7

first_load_prepare:
  blt s4, s3, end_prepare
  lb s5,0(s1)
  addi sp, sp, -1
  sb s5, 0(sp)
  addi s3, s3, 1
  addi s1, s1, 1
  j first_load_prepare

end_prepare:
```

Loop checking condition and value loop preparation

```
start_xor:
  lb s6, 3(sp)
  lb s7, 7(sp)
  xor s7, s7, s6
  sb s7 -1(sp)

  lb s6, 2(sp)
  lb s7, 6(sp)
  xor s7, s7, s6
  sb s7 -2(sp)

  lb s6, 1(sp)
  lb s7, 5(sp)
  xor s7, s7, s6
  sb s7 -3(sp)

  lb s6, 0(sp)
  lb s7, 4(sp)
  xor s7, s7, s6
  sb s7 -4(sp)
```

Parity Code generation and store

```
li t5, 0
rem t5, t3, t6
add s10 zero sp
beqz t5, print_1
beq t5, s8, print_2
beq t5, s9, print_3
```

Print Strategy Choosing

1.4.5. Print Strategy 1

```
print_1:
start_print:
  li a7, 4
  la a0, loop_begin_first
  ecall

  li a7, 11
  lb a0, 7(sp)
  ecall

  li a7, 11
  lb a0, 6(sp)
  ecall

  li a7, 11
  lb a0, 5(sp)
  ecall

  li a7, 11
  lb a0, 4(sp)
  ecall

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

Print first 4 bytes part

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

li a7, 4
la a0, loop_begin
ecall

li a7, 11
lb a0, 3(sp)
ecall

li a7, 11
lb a0, 2(sp)
ecall

li a7, 11
lb a0, 1(sp)
ecall

li a7, 11
lb a0, 0(sp)
ecall

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

Print second 4 bytes part

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

li a7, 4
la a0, partition_begin
ecall

li s3, 0
li s4, 3
j xor_print_loop

xor_print_loop:
li a7, 4
blt s4, s3, end_xor_print
lb a1, -1(s10)
andi s6, a1, 0xf0
srli s6, s6, 4

bge s6, t1, print_char_first
blt s6, t1, print_num_first

print_num_first:
li a7, 1
add a0, zero ,s6
ecall
j after_first_print
```

Load parity code part

```
print_char_first:
addi s6, s6, 87
li a7, 11
add a0, zero ,s6
ecall
j after_first_print

after_first_print:
andi s7, a1, 0x0f
bge s7, t1, print_char_second
blt s7, t1, print_num_second

print_num_second:
li a7, 1
add a0, zero ,s7
ecall
j after_second_print

print_char_second:
addi s7, s7, 87
li a7, 11
add a0, zero ,s7
ecall
j after_second_print
```

Print and convert parity code part

```
after_second_print:
  addi s10, s10, -1
  bne s3, s4, add_comma_space
  addi s3, s3, 1
  j xor_print_loop

add_comma_space:
  li a7, 4
  la a0, comma_space
  ecall
  addi s3, s3, 1
  j xor_print_loop

end_xor_print:
  li a7, 4
  la a0, partition_end
  ecall
  j end_current_loop
```

Print and convert parity code part

1.4.6. Print Strategy 2

```
print_2:
start_print_2:
  li a7, 4
  la a0, loop_begin_first
  ecall

  li a7, 11
  lb a0, 7(sp)
  ecall

  li a7, 11
  lb a0, 6(sp)
  ecall

  li a7, 11
  lb a0, 5(sp)
  ecall

  li a7, 11
  lb a0, 4(sp)
  ecall

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

Print 4 bytes first part

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

  li a7, 4
  la a0, partition_begin
  ecall

  li s3, 0
  li s4, 3
  j xor_print_loop_2

xor_print_loop_2:
  blt s4, s3, end_xor_print_2
  lb a1, -1(sp)
  andi s6, a1, 0xf0
  srli s6, s6, 4

  bge s6, t1, print_char_first_2
  blt s6, t1, print_num_first_2

print_num_first_2:
  li a7, 1
  add a0, zero ,s6
  ecall
  j after_first_print_2
```

Load parity code part

```
print_char_first_2:
  addi s6, s6, 87
  li a7, 11
  add a0, zero ,s6
  ecall
  j after_first_print_2

after_first_print_2:
  andi s7, a1, 0x0f
  bge s7, t1, print_char_second_2
  blt s7, t1, print_num_second_2

print_num_second_2:
  li a7, 1
  add a0, zero ,s7
  ecall
  j after_second_print_2

print_char_second_2:
  addi s7, s7, 87
  li a7, 11
  add a0, zero ,s7
  ecall
  j after_second_print_2
```

Print and convert parity code part

```
  li a7, 11
  lb a0, 3(sp)
  ecall

  li a7, 11
  lb a0, 2(sp)
  ecall

  li a7, 11
  lb a0, 1(sp)
  ecall

  li a7, 11
  lb a0, 0(sp)
  ecall

  li a7, 4
  la a0, loop_end
  ecall

  j end_current_loop
```

Print 4 bytes last part

1.4.7. Print Strategy 3

```
## PRINT STRATEGY 3
print_3:
start_print_3:
  li a7, 4
  la a0, partition_begin_first
  ecall

  li s3, 0
  li s4, 3
  j xor_print_loop_3

xor_print_loop_3:
  blt s4, s3, end_xor_print_3
  lb a1, -1(sp)
  andi s6, a1, 0xf0
  srli s6, s6, 4

  bge s6, t1, print_char_first_3
  blt s6, t1, print_num_first_3

print_num_first_3:
  li a7, 1
  add a0, zero ,s6
  ecall
  j after_first_print_3
```

Load parity code part

```
print_char_first_3:
  addi s6, s6, 87
  li a7, 11
  add a0, zero ,s6
  ecall
  j after_first_print_3

after_first_print_3:
  andi s7, a1, 0x0f
  bge s7, t1, print_char_second_3
  blt s7, t1, print_num_second_3

print_num_second_3:
  li a7, 1
  add a0, zero ,s7
  ecall
  j after_second_print_3

print_char_second_3:
  addi s7, s7, 87
  li a7, 11
  add a0, zero ,s7
  ecall
  j after_second_print_3
```

Print and convert parity code part

```
after_second_print_3:
  addi s10, s10, -1
  bne s3, s4, add_comma_space_3
  addi s3, s3, 1
  j xor_print_loop_3

add_comma_space_3:
  li a7, 4
  la a0, comma_space
  ecall
  addi s3, s3, 1
  j xor_print_loop_3

end_xor_print_3:
  li a7, 4
  la a0, partition_end
  ecall

  li a7, 4
  la a0, space_line_bound
  ecall

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

Print and convert parity code part

```
li a7, 11
lb a0, 7(sp)
ecall

li a7, 11
lb a0, 6(sp)
ecall

li a7, 11
lb a0, 5(sp)
ecall

li a7, 11
lb a0, 4(sp)
ecall

li a7, 4
la a0, loop_end
ecall

li a7, 4
la a0, space_line_bound
ecall

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

Print first 4 bytes code part

```
li a7, 11
lb a0, 3(sp)
ecall

li a7, 11
lb a0, 2(sp)
ecall

li a7, 11
lb a0, 1(sp)
ecall

li a7, 11
lb a0, 0(sp)
ecall

li a7, 4
la a0, loop_end
ecall

li s3, 0
li s4, 3
j end_current_loop
```

Print second 4 bytes code part

1.4.8. Loop restore and traversing

```
end_current_loop:
  addi t3, t3, 1
  addi sp, sp, 8
  j start_loop
```

Restore stack pointer and move to next element

1.4.9. Ending bound generation

```
#Print the end line of boundary
end_bound:
  li a7, 4
  la a0, begin_line_bound
  ecall

  li a7, 4
  la a0, space_title_bound
  ecall

  li a7, 4
  la a0, line_bound
  ecall

  li a7, 4
  la a0, space_title_bound
  ecall

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

Ending bound generation

1.5. Simulation Result

```
Enter the input string: DCE.****ABCD1234HUSTHUST
    DISK 1                  DISK 2                  DISK 3
 --------------          --------------          --------------
|     DCE.     |        |     ****     |        [[ 6e,69,6f,04]]
|     ABCD     |        [[ 70,70,70,70]]        |     1234     |
[[ 00,00,00,00]]        |     HUST     |        |     HUST     |
 --------------          --------------          --------------
-- program is finished running (dropped off bottom) --
```

Demo Result

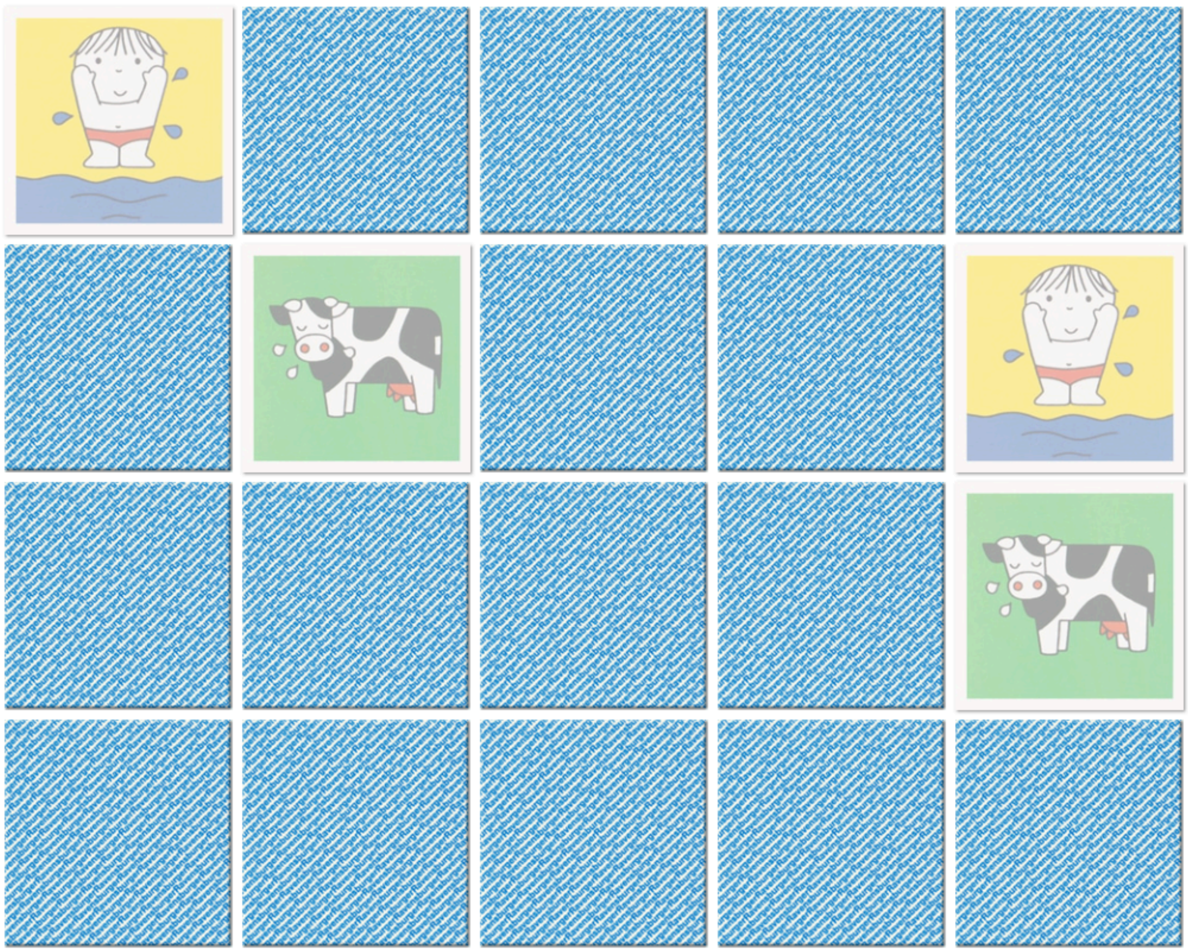
2. Problem 2: Flip Card Game

2.1. Description:

14. Flip card game

+ Reference https://www.memozor.com/memory-games/for-toddlers-babies/dick-bruna
+ Research about the system call to generate a random number.
+ On the Bitmap Display window, 4x4 square cards with 8 pairs of different colors are displayed in random positions, in a face-down state.
+ The user opens the card by pressing the corresponding button on the key matrix. If the opened pair of cards has the same color, they will be in the open state, if the pair of cards has a different color, they will return to the face-down state.
+ The game ends if all the cards are opened.

Requirement



How this game look like

Two images above provide a comprehensive view of the task. When player choose the same type for two items, it will be considered correct.

2.2. Algorithm

2.2.1. Generate Random Color

Initially, we have an array store 8 different colors and another array store the amount of each individual color we need to fill into the Bitmap Display (all are 2).

```
Colors: .word RED, GREEN, BLUE, YELLOW, CYAN, MAGENTA, WHITE, GRAY
Mark: .word 2, 2, 2, 2, 2, 2, 2, 2
Temp: .word 0, 0, 0, 0, 0, 0, 0, 0
```

Three important Arrays

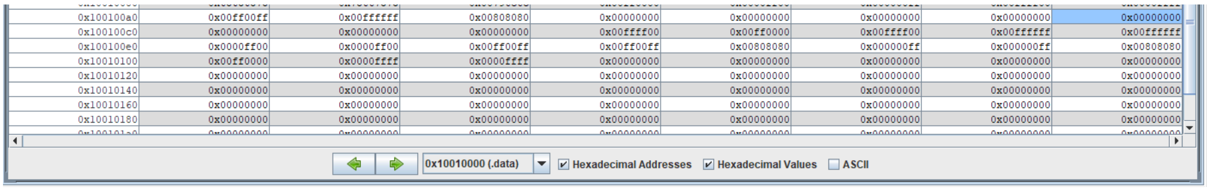
After that, we use **system call 42** to generate a random integer in the boundary of [0, 7]. When having a random number, we first check in the **Mark** array whether this color is generated enough? If the value is greater than 0, we store this color to the corresponding address of the **Temp** array.

A question may arise is why **Temp** array? This is because we do not want to store the color to the Bitmap Display at the beginning, we have to hide it as the game's rule. When player hits the keyboard, the color with corresponding key will be taken from **Temp** array, then save to **Bitmap Display** to describe the color that the user are choosing.

```
loop:
        beq s0, s1, exit
generate_random:
        li a7, 42 # System call to generate random integer
        li a1, 8  # It should be in the range 0 ~ 7
        ecall
load_color:
        la t1, Colors # Load address of Color
        la t2, Mark   # Load address of counter for each color
        addi a2, zero, 4 # Value of each word in array
        mul a2, a2, a0 # Calculate address of element from random number by a0 * 4
        add t2, t2, a2 # Add its address to base Mark's address
        lw a3, 0(t2) # Load the counter of this color
        jal check_slot # Check if we have enough pair?
        add t1, t1, a2 # Add its address to base Colors' address
        lw a4, 0(t1) # Load color
        sw a4, 0(t3) # Store color to Temp
        addi t3,t3, 4 # Point Temp to the next address
        addi a3, a3, -1 # Minus counter of this color by 1
        sw a3, 0(t2) # Store again counter to Mark
        addi s0, s0, 1
        j loop
check_slot:
        beq a3, zero, generate_random # If a color have enough pair, continue random
        jr ra # If not, continue to assign color
exit:
```

Functions generate random color

We can see the output in the data segment with 8 pair of color:



0x100100a0	0x00ff00ff	0x00ffffff	0x00808080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00ffff00	0x00ff0000	0x00ffff00	0x00ffffff	0x00ffffff
0x100100e0	0x0000ff00	0x0000ff00	0x00ff00ff	0x00ff00ff	0x00808080	0x000000ff	0x000000ff	0x00808080
0x10010100	0x00ff0000	0x0000ffff	0x0000ffff	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

2.2.2. Input and Retrieve Color

As mentioned above, when the player hits the key, a new color will be displayed in the Bitmap. If the user hits two times, we have to check whether those colors make a pair. In the worst case, we need to reset those colors - in other words to hide them from Bitmap. The ***input_loop*** function will be called until we reach 8 pairs of colors.

Another important thing is when the player hits the key and the color at the corresponding address is visible, we have to jump to the ***notify*** function to let them know, then remove color for the single color which is displayed in the Bitmap (if exist) by calling ***reset_color_for_first_input***.

```
input_loop:
        beq s3, s2, is_match # If user click 2 times, then check if match
        beq s0, s1, done # If couple = 8, done
        li t0, MONITOR_SCREEN # Load address of MONITOR
        la t3, Temp # Load random array color
        add s4, zero, a3 # Save first color
        li a7, 51
        la a0, info
        ecall
        add a5, zero, a4  # Save number of first color in Random Array
        jal retrieve # Display color to bitsmap
        j input_loop # Print message and try again
retrieve:
        add a4, zero, a0        # Save current color number in Random Array
        addi a2, zero, 4        # Store the size of each word
        mul a2, a2, a0          # Multiply to find the corresponding index
        add t0, t0, a2          # Add to the MONITOR base's address
        add t3, t3, a2          # Add to the Random Array Color base Address
        lw a6, 0(t0)            # Load current color of this address
        bnez a6, notify         # If this address already have color, next
        lw a3, 0(t3)            # Load color
        sw a3, 0(t0)            # Save to Minitor to display
        addi s3, s3, 1
        jr ra
```

Input looping and Display color

```
notify:
        li a7, 55
        la a0, error
        li a1, 0
        ecall
        bgt s3, zero, reset_color_for_first_input
        j input_loop
```

Dialog box to notify

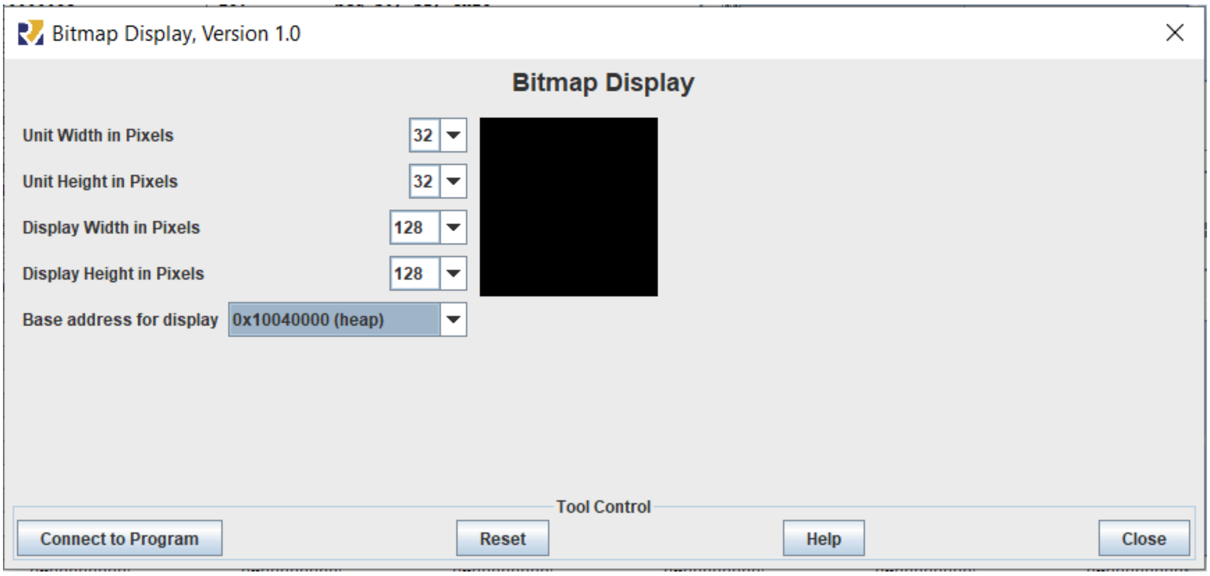
2.2.3. Matching Color

To check matching color, we have to store them in two different registers, then **xor** them with each other. If the result is equal to 0, then they have the same color. The procedure of removing current color is also done in the *not_match* function.

```
is_match:
	xor s5, s4, a3 # Compare two color
	add s3, zero, zero # Reset s3
	beqz s5, match # If they color have the same color, then result is 0 and match
	j not_match
match:
	addi s0, s0, 1 # Increase pair by 1
	j input_loop # Continue to find and check if done
not_match:
	sw zero, 0(t0) # Because when we are in this function, MONITOR SCREEN current point to second color, so we reset
	addi a2, zero, 4
	mul a2, a2, a5 # Address of first number
	li t0, MONITOR_SCREEN # Load base address again
	add t0, t0, a2 # Index of first color
	sw zero, 0(t0) # Reset
	j input_loop
```

Checking functions

2.3. Guideline

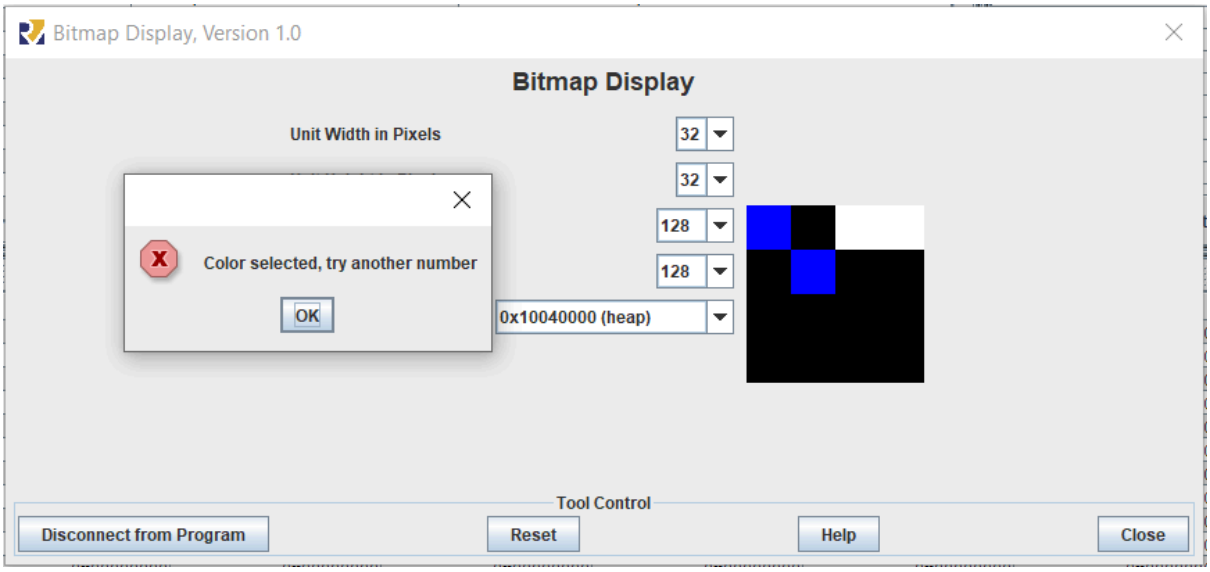


Bitmap Display setting

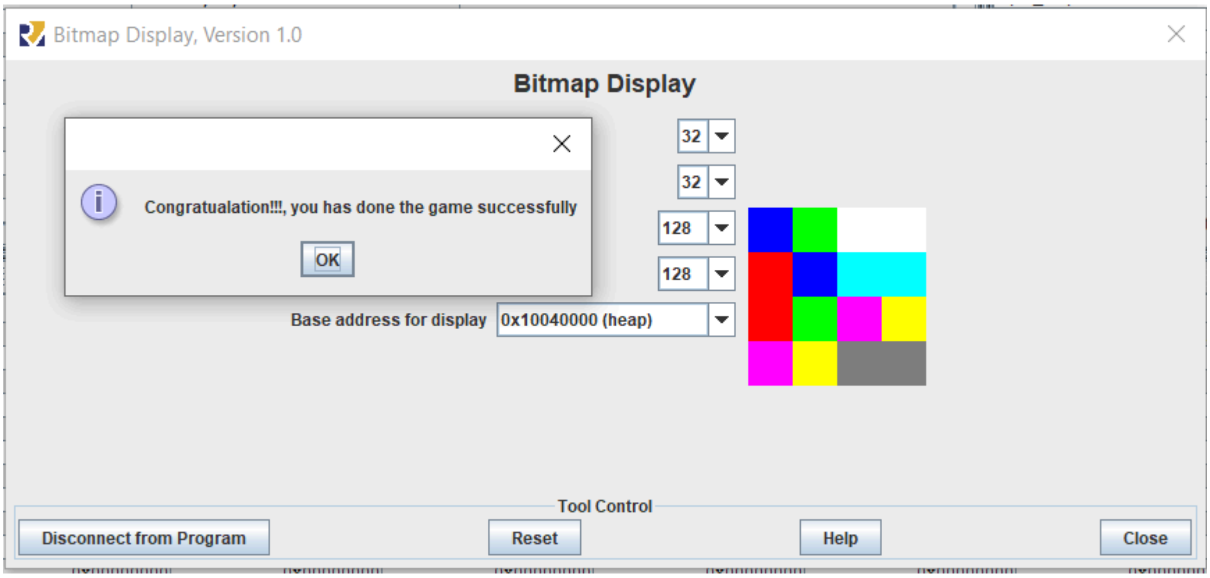
Note: Base address for displaying must be **0x10040000 (heap)**. This setting aims to avoid overlapping data segments and we run with the speed of **30 inst/sec**.

After setting it completely, run the program and wait until it requires an input number. Input number will be in range **0** -> **15**, representing 16 colors - 8 pairs. The player has to find 8 pairs of colors to win this game.

2.4. Demo



Player choose found color



Player winning the game