

Nguyễn Đình Dương – 20225966 – Lab10

Assignment 1

Code:

```
# RISC-V Assembly Code - Assignment 1
# Hiển thị studentID "66" trên LED 7 đoạn

.data
DIGIT_TO_7SEG:
    .byte 0x3F # 0: 00111111
    .byte 0x06 # 1: 00000110
    .byte 0x5B # 2: 01011011
    .byte 0x4F # 3: 01001111
    .byte 0x66 # 4: 01100110
    .byte 0x6D # 5: 01101101
    .byte 0x7D # 6: 01111101
    .byte 0x07 # 7: 00000111
    .byte 0x7F # 8: 01111111
    .byte 0x6F # 9: 01101111

.text
.global main
main:
    # Hiển thị số "6" trên LED bên trái
    li    a0, 125          # 0x7D trong hệ thập phân
    jal   SHOW_7SEG_LEFT   # Gọi hàm hiển thị bên trái

    # Hiển thị số "6" trên LED bên phải
    li    a0, 125          # 0x7D trong hệ thập phân
    jal   SHOW_7SEG_RIGHT  # Gọi hàm hiển thị bên phải

    # Đọc một ký tự từ người dùng
    li    a7, 12           # ecall code cho đọc ký tự
    ecall

    # Mã ASCII của ký tự nằm trong a0

    # Lấy 2 chữ số cuối của mã ASCII
    li    t3, 100          # t3 = 100
    rem   t0, a0, t3       # t0 = a0 % 100 (lấy 2 chữ số cuối)

    # Tách chữ số hàng chục và hàng đơn vị
```

```

li    t3, 10          # t3 = 10
div   t1, t0, t3      # t1 = t0 / 10 (chữ số hàng chục)
rem   t2, t0, t3      # t2 = t0 % 10 (chữ số hàng đơn vị)

# Lấy mã 7 đoạn cho chữ số hàng chục
la    t3, DIGIT_TO_7SEG # t3 = địa chỉ bảng mã 7 đoạn
add   t4, t3, t1       # t4 = địa chỉ mã 7 đoạn cho chữ số hàng chục
lbu   a0, 0(t4)        # a0 = mã 7 đoạn cho chữ số hàng chục
jal   SHOW_7SEG_LEFT   # Hiển thị trên LED bên trái

# Lấy mã 7 đoạn cho chữ số hàng đơn vị
la    t3, DIGIT_TO_7SEG # t3 = địa chỉ bảng mã 7 đoạn
add   t4, t3, t2       # t4 = địa chỉ mã 7 đoạn cho chữ số hàng đơn vị
lbu   a0, 0(t4)        # a0 = mã 7 đoạn cho chữ số hàng đơn vị
jal   SHOW_7SEG_RIGHT  # Hiển thị trên LED bên phải

# Kết thúc chương trình
li    a7, 10          # ecall code cho thoát
ecall

```

```

# -----
# Hàm SHOW_7SEG_LEFT: Hiển thị trên LED 7 đoạn bên trái
# param[in] a0 - giá trị cần hiển thị
# -----
SHOW_7SEG_LEFT:
    lui    t0, 1048560    # t0 = 1048560 << 12
    addi   t0, t0, 17     # t0 = t0 + 17 (địa chỉ SEVENSEG_LEFT)
    sb     a0, 0(t0)      # Ghi giá trị vào địa chỉ LED bên trái
    jr     ra             # Quay lại hàm gọi

# -----
# Hàm SHOW_7SEG_RIGHT: Hiển thị trên LED 7 đoạn bên phải
# param[in] a0 - giá trị cần hiển thị
# -----
SHOW_7SEG_RIGHT:
    lui    t0, 1048560    # t0 = 1048560 << 12
    addi   t0, t0, 16     # t0 = t0 + 16 (địa chỉ SEVENSEG_RIGHT)
    sb     a0, 0(t0)      # Ghi giá trị vào địa chỉ LED bên phải
    jr     ra             # Quay lại hàm gọi

```

1. Hiển thị số "66" trên LED 7 đoạn:

- Gọi hàm SHOW_7SEG_LEFT để hiển thị số 6 trên LED bên trái.
- Gọi hàm SHOW_7SEG_RIGHT để hiển thị số 6 trên LED bên phải.

2. Đọc ký tự từ người dùng:

- Thực hiện lệnh ecall để đọc mã ASCII của ký tự từ bàn phím.

3. Lấy 2 chữ số cuối của mã ASCII:

- Tính **phần dư** khi chia mã ASCII cho 100 để lấy hai chữ số cuối.
- Tách **hàng chục** và **hàng đơn vị**:
 - Hàng chục = Chia hai chữ số cuối cho 10.
 - Hàng đơn vị = Phần dư khi chia hai chữ số cuối cho 10.

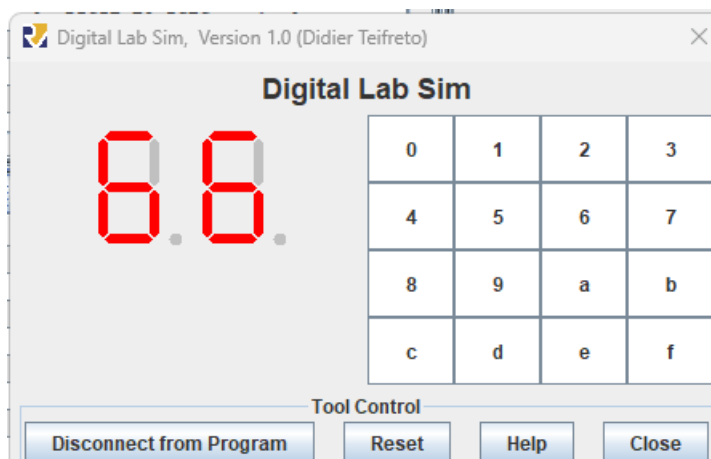
4. Hiển thị 2 chữ số cuối trên LED 7 đoạn:

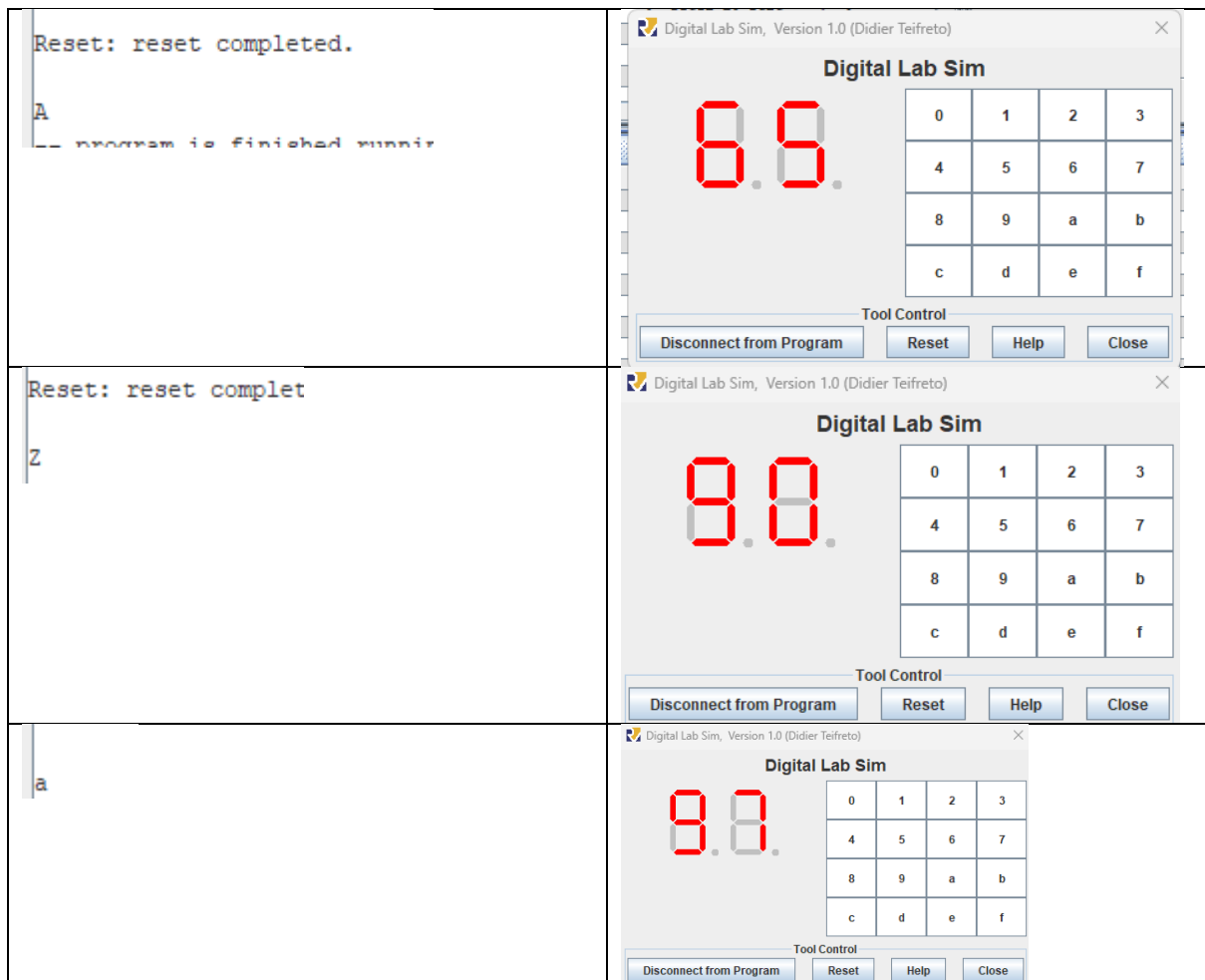
- Lấy mã 7 đoạn cho **chữ số hàng chục** và gọi SHOW_7SEG_LEFT để hiển thị trên LED bên trái.
- Lấy mã 7 đoạn cho **chữ số hàng đơn vị** và gọi SHOW_7SEG_RIGHT để hiển thị trên LED bên phải.

5. Kết thúc chương trình:

- Thực hiện lệnh ecall để thoát chương trình.

Output:





Assignment 2

Code:

```
# RISC-V Assembly Code - Assignment 1
# Hiển thị studentID "66" trên LED 7 đoạn
```

```
.data
DIGIT_TO_7SEG:
    .byte 0x3F # 0: 00111111
    .byte 0x06 # 1: 00000110
    .byte 0x5B # 2: 01011011
    .byte 0x4F # 3: 01001111
    .byte 0x66 # 4: 01100110
    .byte 0x6D # 5: 01101101
    .byte 0x7D # 6: 01111101
    .byte 0x07 # 7: 00000111
```

```

.byte 0x7F # 8: 01111111
.byte 0x6F # 9: 01101111

.text
.global main
main:
    # Đọc một ký tự từ người dùng
    li    a7, 12          # ecall code cho đọc ký tự
    ecall

    # Mã ASCII của ký tự nằm trong a0

    # Lấy 2 chữ số cuối của mã ASCII
    li    t3, 100          # t3 = 100
    rem    t0, a0, t3      # t0 = a0 % 100 (lấy 2 chữ số cuối)

    # Tách chữ số hàng chục và hàng đơn vị
    li    t3, 10           # t3 = 10
    div    t1, t0, t3      # t1 = t0 / 10 (chữ số hàng chục)
    rem    t2, t0, t3      # t2 = t0 % 10 (chữ số hàng đơn vị)

    # Lấy mã 7 đoạn cho chữ số hàng chục
    la    t3, DIGIT_TO_7SEG # t3 = địa chỉ bảng mã 7 đoạn
    add    t4, t3, t1       # t4 = địa chỉ mã 7 đoạn cho chữ số hàng chục
    lbu    a0, 0(t4)        # a0 = mã 7 đoạn cho chữ số hàng chục
    jal    SHOW_7SEG_LEFT   # Hiển thị trên LED bên trái

    # Lấy mã 7 đoạn cho chữ số hàng đơn vị
    la    t3, DIGIT_TO_7SEG # t3 = địa chỉ bảng mã 7 đoạn
    add    t4, t3, t2       # t4 = địa chỉ mã 7 đoạn cho chữ số hàng đơn vị
    lbu    a0, 0(t4)        # a0 = mã 7 đoạn cho chữ số hàng đơn vị
    jal    SHOW_7SEG_RIGHT  # Hiển thị trên LED bên phải

    # Kết thúc chương trình
    li    a7, 10           # ecall code cho thoát
    ecall

# -----
# Hàm SHOW_7SEG_LEFT: Hiển thị trên LED 7 đoạn bên trái
# param[in] a0 - giá trị cần hiển thị
# -----
SHOW_7SEG_LEFT:
    lui    t0, 1048560      # t0 = 1048560 << 12
    addi    t0, t0, 17      # t0 = t0 + 17 (địa chỉ SEVENSEG_LEFT)

```

```

sb    a0, 0(t0)      # Ghi giá trị vào địa chỉ LED bên trái
jr    ra              # Quay lại hàm gọi

# -----
# Hàm SHOW_7SEG_RIGHT: Hiển thị trên LED 7 đoạn bên phải
# param[in] a0 - giá trị cần hiển thị
# -----
SHOW_7SEG_RIGHT:
    lui    t0, 1048560    # t0 = 1048560 << 12
    addi   t0, t0, 16      # t0 = t0 + 16 (địa chỉ SEVENSEG_RIGHT)
    sb     a0, 0(t0)      # Ghi giá trị vào địa chỉ LED bên phải
    jr     ra              # Quay lại hàm gọi

```

1. Đọc ký tự từ người dùng:

- Gọi ecall với mã 12 để đọc một ký tự từ bàn phím.
- Lưu mã ASCII của ký tự vào thanh ghi a0.

2. Tính 2 chữ số cuối của mã ASCII:

- Lấy **phần dư** của mã ASCII khi chia cho 100 để giữ lại hai chữ số cuối. Kết quả lưu vào t0.

3. Tách chữ số hàng chục và hàng đơn vị:

- **Hàng chục:**
 - Chia t0 cho 10 và lưu kết quả vào t1.
- **Hàng đơn vị:**
 - Lấy phần dư của t0 khi chia cho 10 và lưu kết quả vào t2.

4. Hiển thị chữ số hàng chục trên LED bên trái:

- Lấy địa chỉ của bảng DIGIT_TO_7SEG và tính địa chỉ tương ứng với chữ số hàng chục (t1).
- Nạp giá trị mã 7 đoạn từ bảng và lưu vào a0.
- Gọi hàm SHOW_7SEG_LEFT để hiển thị giá trị trên LED bên trái.

5. Hiển thị chữ số hàng đơn vị trên LED bên phải:

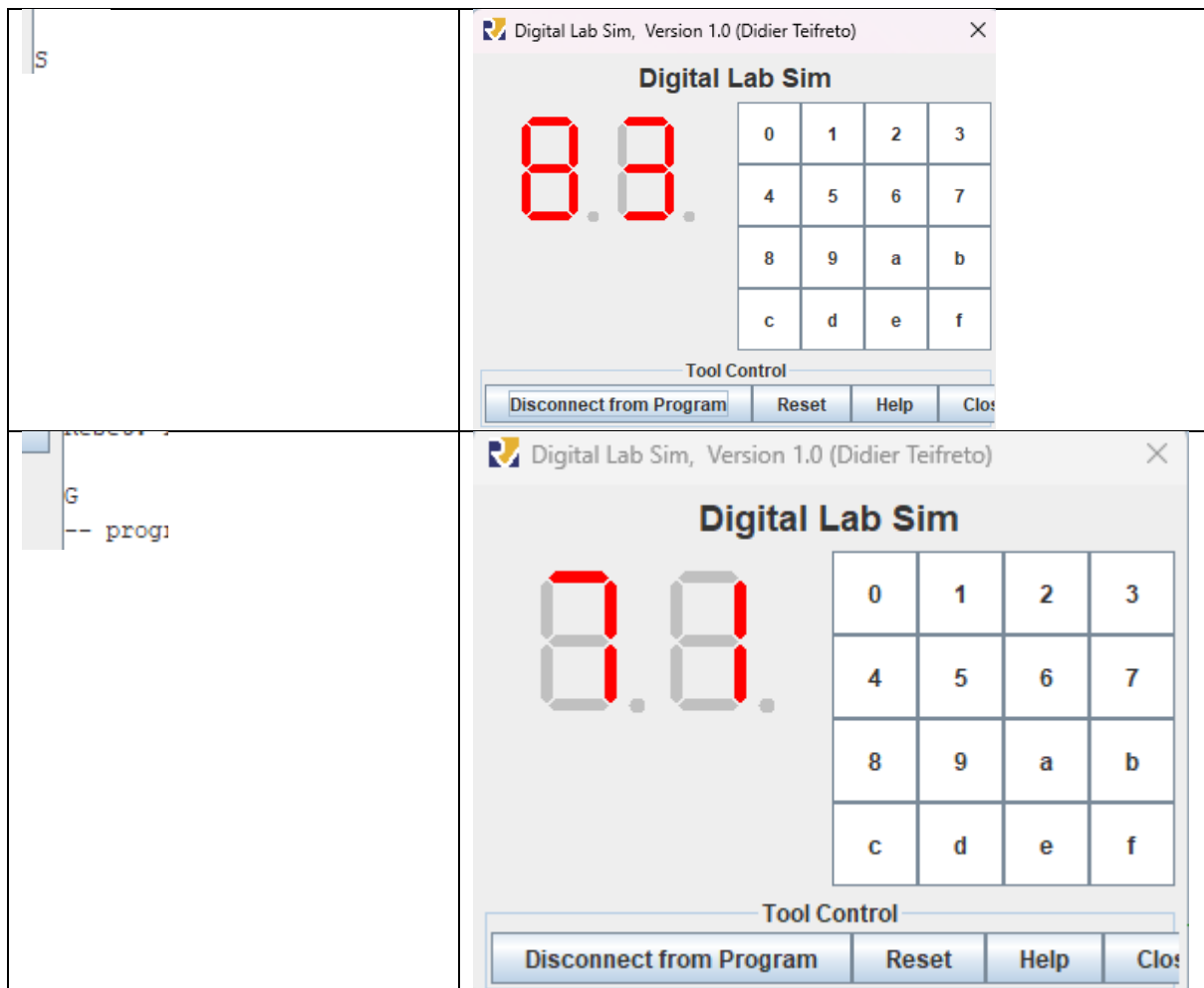
- Lấy địa chỉ của bảng DIGIT_TO_7SEG và tính địa chỉ tương ứng với chữ số hàng đơn vị (t2).
- Nạp giá trị mã 7 đoạn từ bảng và lưu vào a0.

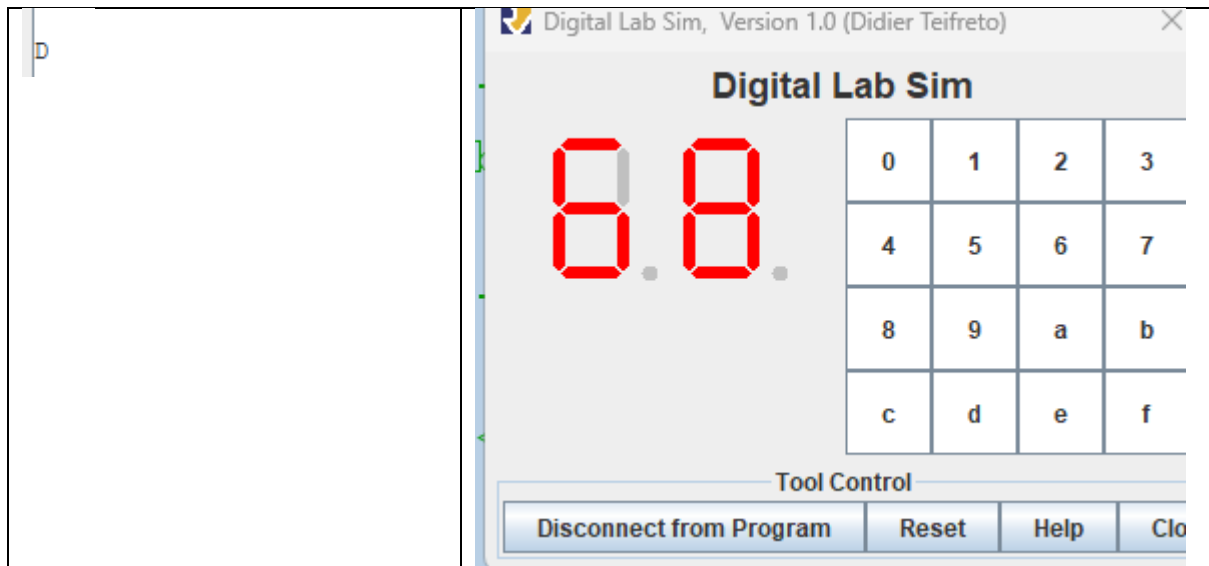
- Gọi hàm SHOW_7SEG_RIGHT để hiển thị giá trị trên LED bên phải.

6. Kết thúc chương trình:

- Gọi ecall với mã 10 để thoát chương trình

Output:





Assignment 3

Code:

```
# RISC-V Assembly Code - Assignment 3
# Vẽ bàn cờ vua trên Bitmap Display
# Lưu ý setting bitmap sẽ là:
# Unit width, height in pixels: 2
# Display width, height in pixels: 256

.data
# Định nghĩa các màu sắc
WHITE:    .word 0x00FFFFFF # Màu trắng
BLACK:    .word 0x00000000 # Màu đen

.text
.global main
main:
    # Cấu hình các tham số của Bitmap Display
    # Giả sử Display Width và Height là 128 đơn vị (unit)
    # Bạn có thể điều chỉnh tùy theo cấu hình của bạn

    # Địa chỉ cơ sở của màn hình
    li    t0, 0x10010000    # MONITOR_SCREEN

    # Tải màu trắng và đen vào thanh ghi
    la    t1, WHITE         # Địa chỉ của màu trắng
    lw    t2, 0(t1)         # t2 = màu trắng
```



```

la    t1, BLACK      # Địa chỉ của màu đen
lw    t3, 0(t1)      # t3 = màu đen

# Thiết lập các tham số bàn cờ
li    t4, 8          # Số ô trên mỗi hàng và cột (8x8)
li    t5, 16         # Kích thước mỗi ô (16 đơn vị)

# Bắt đầu vòng lặp để vẽ bàn cờ
li    s0, 0          # y = 0 (hàng đầu tiên)

```

draw_row:

```

bge    s0, t4, end_program # Nếu y >= 8, kết thúc
li     s1, 0               # x = 0 (cột đầu tiên)

```

draw_column:

```

bge    s1, t4, next_row    # Nếu x >= 8, chuyển sang hàng tiếp theo

```

Tính toán màu cho ô hiện tại

```

add    t6, s0, s1          # t6 = x + y
andi   t6, t6, 1           # t6 = (x + y) % 2
beq     t6, zero, use_white # Nếu t6 == 0, dùng màu trắng

```

Dùng màu đen

```

mv     s5, t3              # s5 = màu đen
j      draw_square

```

use_white:

```

# Dùng màu trắng
mv     s5, t2              # s5 = màu trắng

```

draw_square:

```

# Vẽ ô vuông kích thước t5 x t5
# Tính toán địa chỉ bắt đầu của ô
li     s6, 128             # s6 = Display Width (số đơn vị trên mỗi hàng)
mul     s7, s0, t5          # s7 = y * kích thước ô
mul     s7, s7, s6          # s7 = s7 * Display Width (tính số dòng bỏ qua)
mul     s8, s1, t5          # s8 = x * kích thước ô (tính số cột bỏ qua)
add     s7, s7, s8          # s7 = tổng offset đơn vị

```

Vòng lặp qua các dòng của ô vuông

```

li     s3, 0               # s3 = i = 0

```

draw_square_row:

```

bge    s3, t5, increment_column # Nếu i >= kích thước ô, tăng cột
# Cập nhật offset cho dòng hiện tại
mul    s9, s3, s6               # s9 = i * Display Width
add    s10, s7, s9              # s10 = offset dòng hiện tại

# Vòng lặp qua các cột của ô vuông
li     s4, 0                    # s4 = j = 0

draw_square_column:
    bge    s4, t5, next_square_row # Nếu j >= kích thước ô, chuyển sang dòng
    tiếp theo

    # Tính địa chỉ pixel
    add    s11, s10, s4          # s11 = offset đơn vị hiện tại
    slli   s11, s11, 2           # s11 = s11 * 4 (mỗi đơn vị 4 byte)
    add    s11, t0, s11          # s11 = địa chỉ pixel hiện tại

    # Ghi màu vào pixel
    sw     s5, 0(s11)           # [địa chỉ pixel] = màu

    # Tiếp tục đến pixel tiếp theo
    addi   s4, s4, 1             # s4 += 1
    j      draw_square_column

next_square_row:
    addi   s3, s3, 1             # s3 += 1
    j      draw_square_row

increment_column:
    addi   s1, s1, 1             # x += 1
    j      draw_column

next_row:
    addi   s0, s0, 1             # y += 1
    j      draw_row

end_program:
    # Kết thúc chương trình
    li     a7, 10                # ecall code cho thoát
    ecall

```

1. Khởi tạo các tham số và màu sắc

1. Định nghĩa hai màu sắc:

- **Trắng** (WHITE) với mã màu 0x00FFFFFF.
- **Đen** (BLACK) với mã màu 0x00000000.

2. Cấu hình tham số bitmap display:

- **Địa chỉ cơ sở** của màn hình tại 0x10010000.
- Tải màu trắng và đen vào các thanh ghi (t2, t3).
- Thiết lập các tham số:
 - Kích thước bàn cờ: 8x8 ô.
 - Kích thước mỗi ô vuông: 16x16 pixel.

2. Vòng lặp vẽ bàn cờ

Vẽ từng hàng (row)

1. Bắt đầu từ hàng 0 (thanh ghi s0).
2. Nếu $s0 \geq 8$, kết thúc vẽ.

Vẽ từng cột (column) trong hàng

1. Bắt đầu từ cột 0 (thanh ghi s1).
2. Nếu $s1 \geq 8$, chuyển sang hàng tiếp theo.

Xác định màu của ô

1. Cộng chỉ số hàng (s0) và cột (s1) để tính tổng.
2. Lấy $(s0 + s1) \% 2$ để xác định màu:
 - Nếu bằng 0: Ô màu **trắng**.
 - Nếu khác 0: Ô màu **đen**.

3. Vẽ ô vuông (square)

1. Tính tọa độ bắt đầu của ô vuông:
 - Dòng đầu tiên của ô vuông = (hàng * kích thước ô) * số đơn vị trên mỗi hàng.
 - Cột đầu tiên của ô vuông = cột * kích thước ô.
 - Tổng offset = tổng dòng và cột (đơn vị).

2. Vẽ từng dòng (row) trong ô:

- Lặp từ 0 đến kích thước ô (16 pixel).
- Tính địa chỉ dòng trong ô.

3. Vẽ từng cột (column) trong dòng:

- Lặp từ 0 đến kích thước ô (16 pixel).
- Tính địa chỉ pixel hiện tại.
- Ghi màu vào pixel.

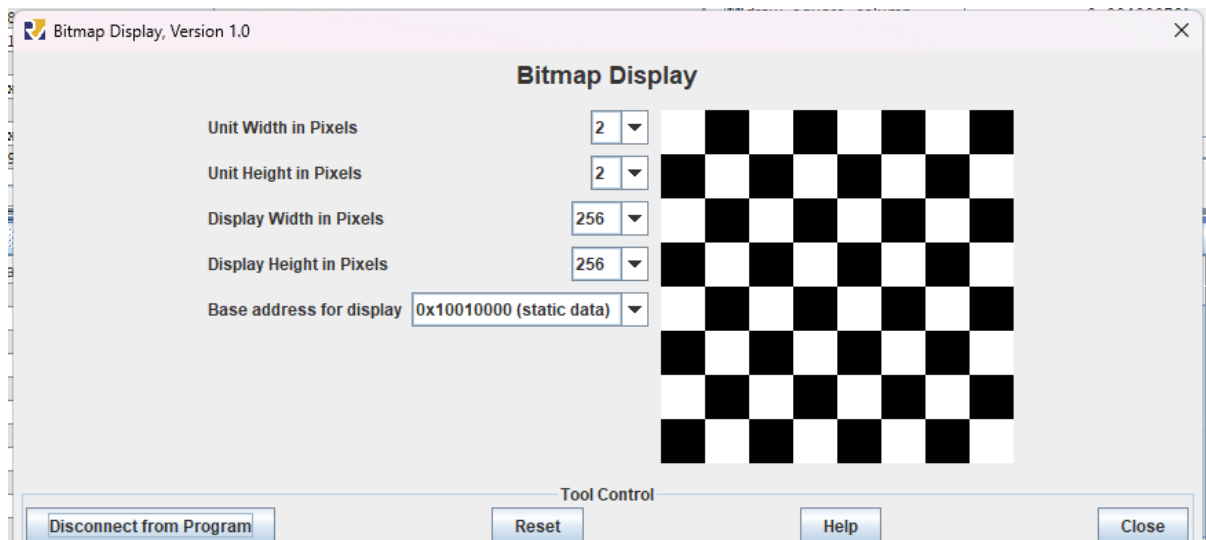
4. Chuyển sang ô và hàng tiếp theo

- Sau khi vẽ xong một ô:
 - Tăng chỉ số cột ($s1 += 1$).
- Sau khi vẽ xong một hàng:
 - Tăng chỉ số hàng ($s0 += 1$).

5. Kết thúc chương trình

- Sau khi hoàn thành vẽ bàn cờ (8x8 ô), gọi `ecall` để thoát chương trình.

Output:



Assignment 4

Code:

```
.data
buffer: .space 4      # Dành 4 byte bộ nhớ để lưu 4 ký tự cuối cùng được nhập
vào.

.text
.globl main
main:
    # Địa chỉ bộ nhớ vào/ra (MMIO)
    .eqv KEY_CODE      0xFFFF0004 # Địa chỉ nhận mã ASCII từ bàn phím,
1 byte
    .eqv KEY_READY     0xFFFF0000 # =1 khi có mã phím mới sẵn sàng
(tự động xóa sau khi đọc)
    .eqv DISPLAY_CODE  0xFFFF000C # Địa chỉ ghi mã ASCII để hiển thị,
1 byte
    .eqv DISPLAY_READY 0xFFFF0008 # =1 khi màn hình sẵn sàng (tự
động xóa sau khi ghi)

    # Khởi tạo địa chỉ và hằng số
    li    a0, KEY_CODE      # Lưu địa chỉ của KEY_CODE vào thanh ghi
a0
    li    a1, KEY_READY     # Lưu địa chỉ của KEY_READY vào thanh
ghi a1
    li    s0, DISPLAY_CODE  # Lưu địa chỉ của DISPLAY_CODE vào
thanh ghi s0
    li    s1, DISPLAY_READY # Lưu địa chỉ của DISPLAY_READY vào
thanh ghi s1
    la    s2, buffer        # Lưu địa chỉ của buffer vào thanh ghi s2
    li    s3, 'e'           # Mã ASCII của ký tự 'e'
    li    s4, 'x'           # Mã ASCII của ký tự 'x'
    li    s5, 'i'           # Mã ASCII của ký tự 'i'
    li    s6, 't'           # Mã ASCII của ký tự 't'

loop:
    # Chờ một phím được nhấn
WaitForKey:
    lw     t1, 0(a1)         # Đọc KEY_READY vào thanh ghi t1
    beq    t1, zero, WaitForKey # Nếu KEY_READY == 0 thì tiếp tục chờ

    # Đọc mã phím
```

ReadKey:

```
lb    t0, 0(a0)      # Đọc mã phím vào t0 (1 byte)
andi  t0, t0, 0xFF    # Đảm bảo t0 chỉ chứa 8 bit
mv     s7, t0         # Lưu mã phím gốc vào thanh ghi s7
```

Xử lý mã phím

Kiểm tra nếu là chữ cái viết thường

```
li     s8, 'a'
```

```
li     s9, 'z'
```

```
blt    t0, s8, check_uppercase # Nếu t0 < 'a', kiểm tra chữ viết hoa
```

```
bgt    t0, s9, check_uppercase # Nếu t0 > 'z', kiểm tra chữ viết hoa
```

```
addi   t0, t0, -32      # Chuyển sang chữ hoa (giảm 32)
```

```
j      after_processing
```

check_uppercase:

Kiểm tra nếu là chữ cái viết hoa

```
li     s8, 'A'
```

```
li     s9, 'Z'
```

```
blt    t0, s8, check_digit    # Nếu t0 < 'A', kiểm tra chữ số
```

```
bgt    t0, s9, check_digit    # Nếu t0 > 'Z', kiểm tra chữ số
```

```
addi   t0, t0, 32          # Chuyển sang chữ thường (tăng 32)
```

```
j      after_processing
```

check_digit:

Kiểm tra nếu là chữ số

```
li     s8, '0'
```

```
li     s9, '9'
```

```
blt    t0, s8, set_star      # Nếu t0 < '0', gán ký tự '*'
```

```
bgt    t0, s9, set_star      # Nếu t0 > '9', gán ký tự '*'
```

```
j      after_processing      # Nếu là chữ số, không thay đổi
```

set_star:

```
li     t0, '*'              # Gán t0 bằng ký tự '*'
```

after_processing:

Chờ màn hình sẵn sàng

WaitForDisplay:

```
lw     t2, 0(s1)           # Đọc DISPLAY_READY vào t2
```

```
beq    t2, zero, WaitForDisplay # Nếu DISPLAY_READY == 0 thì chờ
```

Hiển thị ký tự đã xử lý

```
sb     t0, 0(s0)           # Ghi ký tự vào DISPLAY_CODE
```

```

# Cập nhật buffer với mã phím gốc
# Dịch nội dung buffer
lb    s8, 1(s2)
sb    s8, 0(s2)
lb    s8, 2(s2)
sb    s8, 1(s2)
lb    s8, 3(s2)
sb    s8, 2(s2)
sb    s7, 3(s2)          # Thêm mã phím gốc vào cuối buffer

# Kiểm tra nếu 4 ký tự cuối là "exit"
lb    s8, 0(s2)
bne   s8, s3, loop      # So sánh với 'e'
lb    s8, 1(s2)
bne   s8, s4, loop      # So sánh với 'x'
lb    s8, 2(s2)
bne   s8, s5, loop      # So sánh với 'i'
lb    s8, 3(s2)
bne   s8, s6, loop      # So sánh với 't'

# Thoát chương trình nếu nhập "exit"
j     exit_program

j     loop              # Lặp lại vòng lặp

exit_program:
li    a7, 10            # Mã syscall để thoát
ecall                      # Gọi syscall để kết thúc chương trình

```

1. Khởi tạo

- Khai báo các địa chỉ MMIO:
 - **KEY_CODE**: Lưu mã ASCII từ bàn phím.
 - **KEY_READY**: Cờ kiểm tra nếu có ký tự mới.
 - **DISPLAY_CODE**: Lưu mã ASCII để hiển thị.
 - **DISPLAY_READY**: Cờ kiểm tra nếu màn hình sẵn sàng hiển thị.
- Cài đặt:
 - **buffer**: Bộ nhớ 4 byte để lưu trữ 4 ký tự cuối cùng từ bàn phím.
 - Các thanh ghi (s3, s4, s5, s6) lưu ASCII của 'e', 'x', 'i', 't'.

2. Vòng lặp chính (loop)

a) Chờ người dùng nhấn phím:

- Kiểm tra KEY_READY:
 - Nếu KEY_READY chưa bật (0), tiếp tục chờ.
 - Khi KEY_READY bật (1), đọc ký tự từ KEY_CODE vào thanh ghi t0.

b) Xử lý ký tự:

- Kiểm tra và xử lý các trường hợp:
 - **Ký tự chữ thường (a-z):** Chuyển thành chữ hoa (bằng cách trừ 32).
 - **Ký tự chữ hoa (A-Z):** Chuyển thành chữ thường (bằng cách cộng 32).
 - **Ký tự số (0-9):** Giữ nguyên.
 - **Các ký tự khác:** Đặt ký tự thành '*'.

c) Chờ màn hình sẵn sàng hiển thị:

- Kiểm tra DISPLAY_READY:
 - Nếu chưa sẵn sàng (0), tiếp tục chờ.
 - Khi sẵn sàng (1), ghi ký tự đã xử lý (t0) vào DISPLAY_CODE.

d) Cập nhật buffer:

- Dịch các ký tự trong buffer sang trái:
 - Byte 1 → Byte 0.
 - Byte 2 → Byte 1.
 - Byte 3 → Byte 2.
- Thêm ký tự gốc (chưa xử lý) vào cuối buffer (Byte 3).

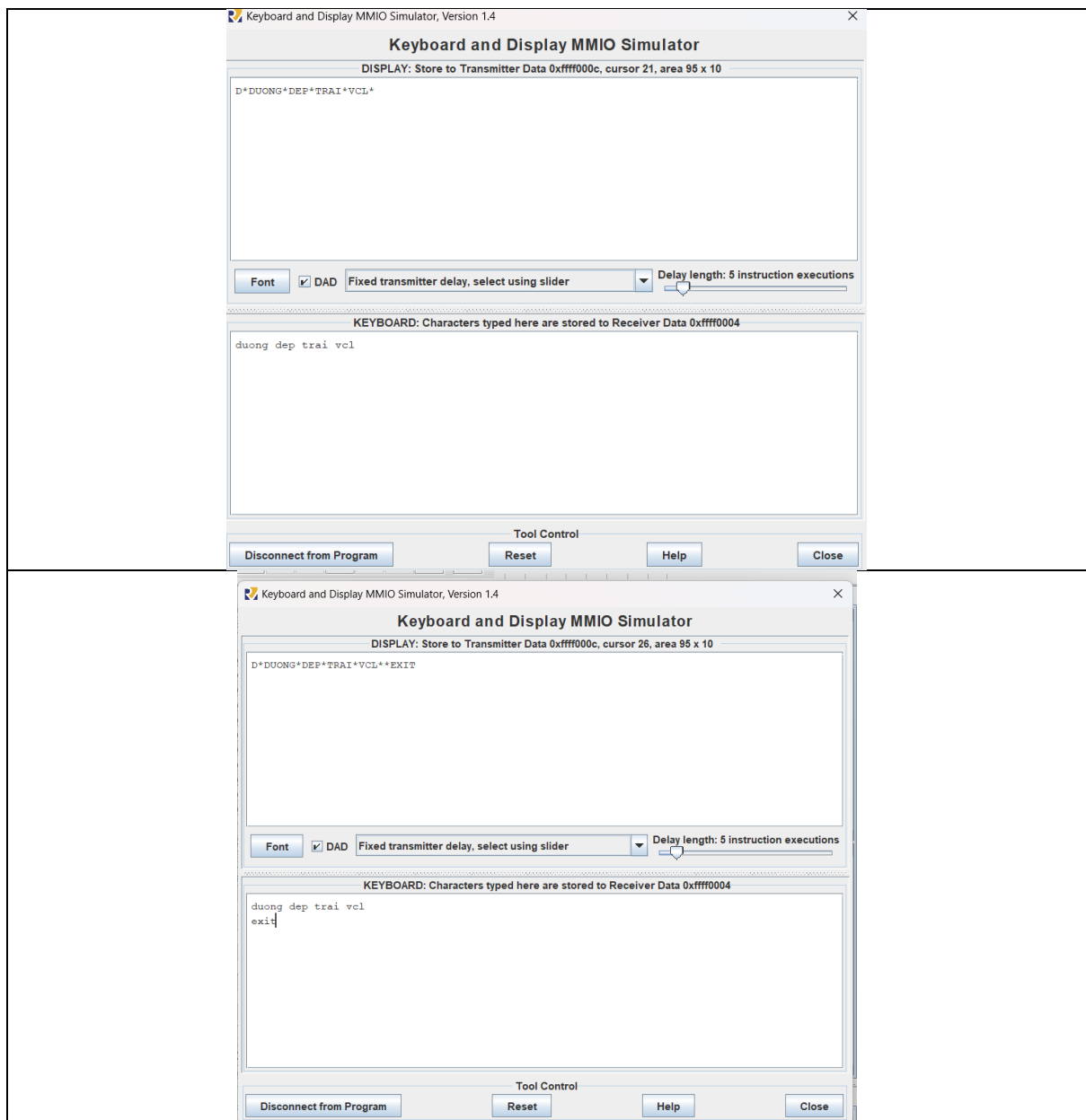
e) Kiểm tra chuỗi "exit" trong buffer:

- So sánh từng byte của buffer với các ký tự 'e', 'x', 'i', 't':
 - Nếu không khớp, quay lại **vòng lặp chính**.
 - Nếu khớp, thoát chương trình.

3. Kết thúc chương trình (exit_program)

- Gọi syscall 10 để thoát chương trình.

Output:





Assignment 5

Code:

```
# Định nghĩa các hằng số
# Setting: Unit: 1, Display: 256
.equ MONITOR_SCREEN, 0x10010000 # Địa chỉ bắt đầu của màn hình
bitmap
```

```

.eqv RED,          0x00FF0000 # Giá trị màu đỏ
.eqv GREEN,        0x0000FF00 # Giá trị màu xanh lá
.eqv WIDTH,        256      # Chiều rộng màn hình (theo pixel)
.eqv UNIT_SIZE,    4        # Đơn vị chiều rộng và chiều cao của 1 pixel

.text
.global main
main:
    # Nạp các hằng số vào thanh ghi
    li    s4, MONITOR_SCREEN # s4 = địa chỉ màn hình
    li    s5, WIDTH          # s5 = chiều rộng màn hình
    li    s6, UNIT_SIZE       # s6 = kích thước đơn vị (4 pixel)

    # Đọc giá trị x1 từ người dùng
    li    a7, 5               # ecall để đọc số nguyên
    ecall
    mv    s0, a0              # s0 = x1

    # Đọc giá trị y1 từ người dùng
    li    a7, 5
    ecall
    mv    s1, a0              # s1 = y1

    # Đọc giá trị x2 từ người dùng
    li    a7, 5
    ecall
    mv    s2, a0              # s2 = x2

    # Đọc giá trị y2 từ người dùng
    li    a7, 5
    ecall
    mv    s3, a0              # s3 = y2

    # Kiểm tra nếu x1 = x2 hoặc y1 = y2 thì thoát
    beq    s0, s2, exit        # Nếu x1 == x2, thoát
    beq    s1, s3, exit        # Nếu y1 == y2, thoát

    # Xác định minX và maxX
    blt    s0, s2, set_minX_s0 # Nếu x1 < x2, gán minX = x1
    mv     t0, s2              # t0 = minX = x2
    mv     t1, s0              # t1 = maxX = x1
    j      set_minY            # Nhảy đến xử lý minY

```

```

set_minX_s0:
    mv    t0, s0          # t0 = minX = x1
    mv    t1, s2          # t1 = maxX = x2

set_minY:
    # Xác định minY và maxY
    blt    s1, s3, set_minY_s1 # Nếu y1 < y2, gán minY = y1
    mv     t2, s3          # t2 = minY = y2
    mv     t3, s1          # t3 = maxY = y1
    j      start_drawing   # Nhảy đến phần vẽ hình chữ nhật

set_minY_s1:
    mv     t2, s1          # t2 = minY = y1
    mv     t3, s3          # t3 = maxY = y2

start_drawing:
    mv     s7, t2          # s7 = y hiện tại = minY

outer_loop:
    bgt     s7, t3, end_drawing # Nếu y > maxY, thoát vòng lặp ngoài

    mv     s8, t0          # s8 = x hiện tại = minX

inner_loop:
    bgt     s8, t1, end_inner_loop # Nếu x > maxX, thoát vòng lặp trong

    # Khởi tạo dy (hàng pixel)
    li     s9, 0          # s9 = dy = 0

dy_loop:
    bge     s9, s6, end_dy_loop # Nếu dy >= UNIT_SIZE, thoát vòng lặp dy

    # Khởi tạo dx (cột pixel)
    li     s10, 0         # s10 = dx = 0

dx_loop:
    bge     s10, s6, end_dx_loop # Nếu dx >= UNIT_SIZE, thoát vòng lặp dx

    # Tính pixel_x = x * UNIT_SIZE + dx
    mul     s11, s8, s6    # s11 = x * UNIT_SIZE
    add     s11, s11, s10  # s11 = pixel_x

    # Tính pixel_y = y * UNIT_SIZE + dy

```

```

mul    t4, s7, s6      # t4 = y * UNIT_SIZE
add    t4, t4, s9      # t4 = pixel_y

# Tính địa chỉ bộ nhớ: MONITOR_SCREEN + ((pixel_y * WIDTH +
pixel_x) * 4)
mul    t5, t4, s5      # t5 = pixel_y * WIDTH
add    t5, t5, s11     # t5 = pixel_y * WIDTH + pixel_x
slli   t5, t5, 2       # t5 = t5 * 4 (địa chỉ byte)
add    t5, s4, t5      # t5 = địa chỉ bộ nhớ

# Gán màu mặc định là xanh lá (GREEN)
li     t6, GREEN       # t6 = GREEN

# Kiểm tra nếu pixel thuộc viền
beq    s8, t0, set_color_red_unit
beq    s8, t1, set_color_red_unit
beq    s7, t2, set_color_red_unit
beq    s7, t3, set_color_red_unit
j      write_pixel_unit

set_color_red_unit:
li     t6, RED         # t6 = RED

write_pixel_unit:
sw     t6, 0(t5)       # Ghi màu vào pixel

# Tăng dx (cột pixel)
addi   s10, s10, 1     # dx = dx + 1
j      dx_loop

end_dx_loop:
# Tăng dy (hàng pixel)
addi   s9, s9, 1       # dy = dy + 1
j      dy_loop

end_dy_loop:
# Tăng x (đơn vị)
addi   s8, s8, 1       # x = x + 1
j      inner_loop

end_inner_loop:
# Tăng y (đơn vị)
addi   s7, s7, 1       # y = y + 1

```

```

j    outer_loop

end_drawing:
    # Thoát chương trình
    li    a7, 10          # ecall để thoát
    ecall

exit:
    li    a7, 10          # ecall để thoát
    ecall

```

1. Khởi tạo

- **Cấu hình hệ thống:**

- Định nghĩa các hằng số:
 - MONITOR_SCREEN: Địa chỉ bộ nhớ bắt đầu của màn hình bitmap.
 - RED và GREEN: Giá trị màu sắc.
 - WIDTH: Chiều rộng màn hình (256 pixel).
 - UNIT_SIZE: Kích thước mỗi đơn vị pixel (4x4 pixel).
- Nạp các giá trị vào thanh ghi (s4, s5, s6).

- **Nhập dữ liệu từ người dùng:**

- Nhập các tọa độ:
 - (x1, y1): Tọa độ góc trên bên trái.
 - (x2, y2): Tọa độ góc dưới bên phải.
- Lưu các giá trị vào các thanh ghi (s0, s1, s2, s3).

2. Kiểm tra và thiết lập giá trị

- **Kiểm tra tính hợp lệ:**

- Nếu $x1 == x2$ hoặc $y1 == y2$, thoát chương trình vì không tạo được hình chữ nhật.

- **Xác định minX, maxX, minY, maxY:**

- So sánh tọa độ đầu vào để xác định góc trên bên trái (minX, minY) và góc dưới bên phải (maxX, maxY).

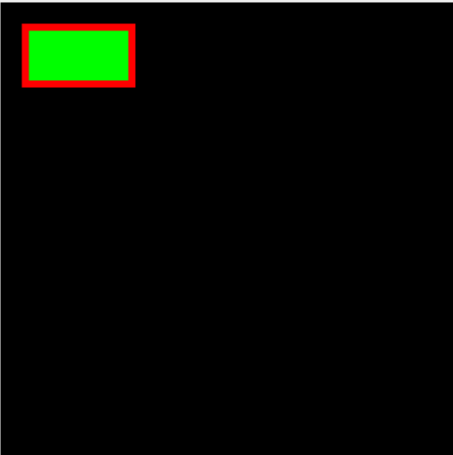
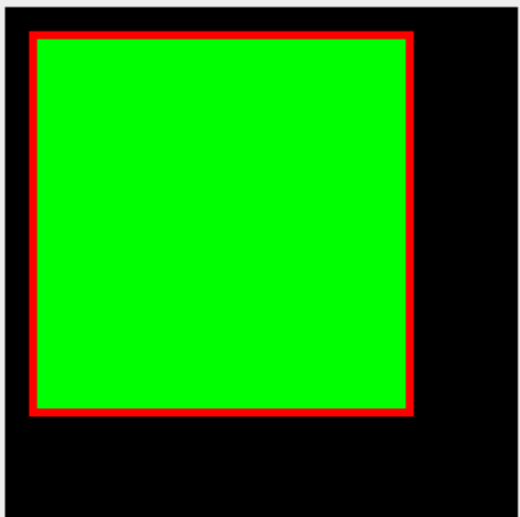
3. Vẽ hình chữ nhật

- **Thiết lập vòng lặp ngoài (theo hàng y):**
 - Bắt đầu từ $y = \min Y$ và lặp cho đến $y = \max Y$.
- **Thiết lập vòng lặp trong (theo cột x):**
 - Bắt đầu từ $x = \min X$ và lặp cho đến $x = \max X$.
- **Vẽ từng đơn vị pixel (4x4):**
 - **Vòng lặp qua các hàng pixel (dy):**
 - Lặp từ $dy = 0$ đến $dy = \text{UNIT_SIZE}$.
 - **Vòng lặp qua các cột pixel (dx):**
 - Lặp từ $dx = 0$ đến $dx = \text{UNIT_SIZE}$.
 - Tính địa chỉ của từng pixel dựa trên:
 - $(\text{pixel_x}, \text{pixel_y}) = (x * \text{UNIT_SIZE} + dx, y * \text{UNIT_SIZE} + dy)$.
 - Địa chỉ bộ nhớ = $\text{MONITOR_SCREEN} + ((\text{pixel_y} * \text{WIDTH} + \text{pixel_x}) * 4)$.
- **Kiểm tra viền của hình chữ nhật:**
 - Nếu pixel nằm trên viền:
 - $x == \min X$, $x == \max X$, $y == \min Y$, hoặc $y == \max Y$, đặt màu **RED**.
 - Ngược lại, đặt màu **GREEN**.
- **Ghi giá trị màu vào địa chỉ bộ nhớ.**

4. Kết thúc chương trình

- Sau khi hoàn thành vẽ hình chữ nhật:
 - Thoát chương trình bằng cách gọi syscall 10.

Ouput:

<div data-bbox="204 197 304 360"> <div></div> <div>3</div> <div>3</div> <div>18</div> <div>11</div> </div>	
<div data-bbox="204 667 316 887"> Reset: <div></div> <div>3</div> <div>3</div> <div>50</div> <div>50</div> </div>	
<div data-bbox="204 1216 304 1379"> <div></div> <div>0</div> <div>0</div> <div>30</div> <div>20</div> </div>	