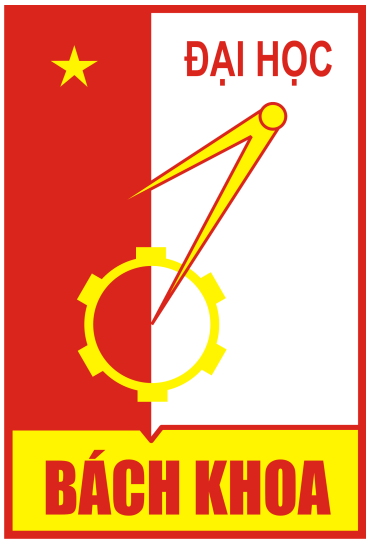

Assembly Lang. & Computer Arch. Lab
Final Project Report

GROUP 01

Khuat The Anh - 20226008
Nguyen Tran Ngan Ha - 20225969

2024-12-17

Contents

1 Problem 6: RAID 5 Disk Simulation

1.1 Problem Statement

Simulate a RAID 5 disk system with three disks using RISC-V assembly language.

- Input a string where the length must be a multiple of 8.
- Divide the input string into 4-byte blocks and store on disk depends on Round-Robin method.
  - Block 1 is stored on Disk *i*.
  - Block 2 is stored on Disk *i + 1*.
  - Parity Block is stored on Disk *i + 2*. Parity is calculated as the XOR of ASCII values of Block *i* and Block *i + 1*.
- Display the data in both ASCII and Hexadecimal formats for each disk.
- Allow the program to restart if the input string length does not meet the requirements.

1.2 Implementation

1.2.1 Data Section

Defines memory for input, messages, and utility markers.

```
    .data
input_buffer:    .space 1024           # Allocate 1024 bytes for input buffer
temp_reg:       .word 1                # Temporary register storage
prompt_msg:     .asciz "Please input a string (length must be multiple of 8): \n"
error_msg:      .asciz "Error: Length must be divisible by 8. Try again! \n"
header:         .asciz "     Disk 1              Disk 2              Disk 3      \n"
divider:        .asciz " --------------      --------------      -------------- \n"
start_ascii:    .asciz "|     "          # Starting marker for ASCII output
end_ascii:      .asciz "     |     "      # Ending marker for ASCII output
start_hex:      .asciz "[[ "            # Starting marker for Hexadecimal output
end_hex:        .asciz "]]     "        # Ending marker for Hexadecimal output
hex_map:        .asciz "0123456789abcdef" # Hexadecimal character map
```

Where:

- *input_buffer*: location that user's input will be stored, and maximum length is 256 bytes
- *hex_map*: a map for printing out hexadecimal number

1.2.2 Input and Validation

Prompt the user to input a string.

```
# Display prompt message to the user
    la a0, prompt_msg
    li a7, 4
    ecall
```

Read the input string into a buffer, and store it in the end of previous string.

```
# Input string into buffer
    la a0, input_buffer                 # Load address of input buffer
    add a0, a0, s11                     # Get the end of previous string
    li a1, 1024                         # Buffer size
    sub a1, a1, s11
    li a7, 8                            # Syscall for string input
    ecall
```

Count characters to ensure the length is divisible by 8.

```
# Count characters in the string
    la t0, input_buffer                 # Load input buffer address
    li t1, 0                            # Character count = 0

count_chars:
    lb t2, 0(t0)                        # Load byte from buffer
    beqz t2, validate_length            # If null terminator, validate length
    addi t1, t1, 1                      # Increment character count
    addi t0, t0, 1                      # Move to the next character
    j count_chars
```

If the length is not divisible by 8, restart the program.

```
    validate_length:
    li t3, 8                            # Length must be divisible by 8
    addi t1, t1, -1                     # Exclude null terminator
    rem t4, t1, t3                      # t4 = t1 % 8
    beqz t4, process_string             # If remainder is 0, continue

# Display error message and restart
    la a0, error_msg
    li a7, 4
    ecall
    j main
```

1.2.3 Process Input String into RAID 5 Blocks

Split input into 8-byte chunks, load data and compute parity:

```
    lw t0, 0(s0)                    # Load first 4-byte
    lw t1, 4(s0)                    # Load second 4-byte

    xor t2, t0, t1                  # XOR first and second word for redundancy
    li t6, 3                        # Disk selection modulus
```

Determine disk storage depending on the round-robin method.

```
    rem t3, t3, t6                  # Determine disk based on block counter
    li t5, 1                        # Constant for Disk 2

# Reset disk flags
    li s1, 0                        # Disk 1 flag
    li s2, 0                        # Disk 2 flag
    li s3, 0                        # Disk 3 flag

# Determine which disk to use
    beqz t3, disk_3                 # If t3 == 0 -> Disk 3
    beq t3, t5, disk_2              # If t3 == 1 -> Disk 2
    j disk_1                        # Else -> Disk 1
```

Process loaded data in each case:

```
disk_3:
    li s3, 1                        # Set Disk 3 flag
    j display_data

disk_2:
    mv t4, t1                       # Swap t1 and t2
    mv t1, t2
    mv t2, t4
    li s2, 1                        # Set Disk 2 flag
    j display_data

disk_1:
    mv t4, t0                       # Swap t0 and t2
    mv t0, t2
    mv t2, t4
    li s1, 1                        # Set Disk 1 flag
```

1.2.4 Display data

Load data in each disk into a temporary register and then call output_data to print data on the disk.

```
display_data:
# Output Disk 1 data
    la a1, temp_reg
    sw t0, 0(a1)                    # Store t0 in temp_reg
    mv a2, s1                       # Disk 1 flag
    jal output_data

# Output Disk 2 data
    la a1, temp_reg
    sw t1, 0(a1)                    # Store t1 in temp_reg
    mv a2, s2                       # Disk 2 flag
    jal output_data

# Output Disk 3 data
    la a1, temp_reg
    sw t2, 0(a1)                    # Store t2 in temp_reg
    mv a2, s3                       # Disk 3 flag
    jal output_data

    j next_block                    # Move to the next block
```

Create a stack frame to store data from the other disk and the counter value.

```
    addi sp, sp, -16                # Create stack frame
    sw t0, 0(sp)
    sw t1, 4(sp)
    sw t2, 8(sp)
    sw t3, 12(sp)
```

Check if the disk flag is 0 or 1 to print the hexadecimal value or the ASCII character.

```
    beqz a2, ascii_output           # If disk flag is 0, print ASCII
```

If the value of the disk flag is 0, then jump to the *ASCII_output* function. This function is used to print each byte in a 4-byte block on the screen.

```
    ascii_output:
    la a0, start_ascii              # Print start of ASCII block
    li a7, 4
    ecall

    li t0, 4                        # Process 4 bytes

ascii_print_loop:
    beqz t0, ascii_done             # Done printing the loading block
    lb a0, 0(a1)                    # Load byte
    li a7, 11                       # Print character
    ecall
    addi t0, t0, -1                 # Decrement counter
    addi a1, a1, 1                  # Move to next byte
    j ascii_print_loop

ascii_done:
    la a0, end_ascii                # Print end of ASCII block
    li a7, 4
    ecall
```

If it is a storage disk, we call the *hex_output* function to print the value of the parity odd of 2 loaded disk.

```
    hex_output:
        li t0, 4                    # Process 4 bytes
        la a0, start_hex            # Print start of hex block
        li a7, 4
        ecall

    hex_print_loop:
        lb t1, 0(a1)                # Load byte
        andi t2, t1, 0xF0           # Extract high nibble
        srli t2, t2, 4
        add t3, s5, t2              # Map to hex_map
        lb a0, 0(t3)                # Print high nibble
        li a7, 11
        ecall

        andi t2, t1, 0x0F           # Extract low nibble
        add t3, s5, t2
```

```
        lb a0, 0(t3)                # Print low nibble
        li a7, 11
        ecall

        addi t0, t0, -1             # Decrement byte counter
        addi a1, a1, 1              # Move to next byte
        beqz t0, hex_done           # Done printing the storing block

        li a7, 11                   # Print ',' separator
        li a0, ','
        ecall
        j hex_print_loop

    hex_done:
        la a0, end_hex              # Print end of hex block
        li a7, 4
        ecall
        j restore_stack
```

In this function, we process 4 bytes one by one, convert each byte to hexadecimal format, and print it. We load each byte to register t1, extract high and low nibble value, then mapping to the *hex_map* to print the hexadecimal value. The loop continues until we print all data in storage disk on the screen.

After that, we restore value that we saved in stack register.

```
    restore_stack:
    lw t0, 0(sp)                    # Restore registers
    lw t1, 4(sp)
    lw t2, 8(sp)
    lw t3, 12(sp)
    addi sp, sp, 16                 # Restore stack pointer
    jr ra
```

When a 8-byte block is completely printed, we move to the next block until it is not the end of string.

```
    next_block:
    addi s0, s0, 8                  # Move to next 8-byte block
    addi t3, t3, 1                  # Increment block counter

    li a0, '\n'                    # Print newline
```

```
    li a7, 11
    ecall

    lb t1, 0(s0)                    # Check for end of input
    li t6, '\n'
    bne t1, t6, data_loop #If it is not the end of string, move back to data processing

end_program:
    la a0, divider
    li a7, 4
    ecall #Print divider line

    j main #Jump back to continue getting user input
```

Finally, we jump back to main to get other user's input.

2 Problem 17: Digital Clock

2.1 Problem Statement

Design and implement a Digital Clock program in RISC-V Assembly to:

- Display current time using 7-segment displays: Allow toggling between display modes (hours, minutes, seconds, day, month, year) via the keypad of Digital Lab Sim.
- Update the time every second.
- Play a sound every full minute.

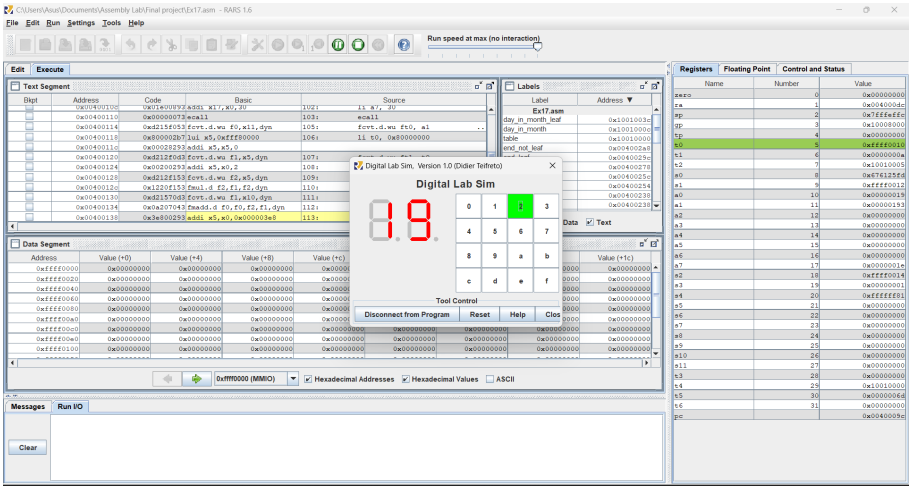

Figure 1: Example for Digital Clock

2.2 Implementation

2.2.1 Memory Configuration and Data Section

Define some constant address and value.

```
.eqv IN_ADDRESS_HEXA_KEYBOARD 0xFFFF0012
.eqv OUT_ADDRESS_HEXA_KEYBOARD 0xFFFF0014
.eqv SEVENSEG_LEFT 0xFFFF0011
.eqv SEVENSEG_RIGHT 0xFFFF0010

.data
table: .byte 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F
day_in_month: .word 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
day_in_month_leap: .word 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
```

Where:

- *table:* List of digits from 0 to 9 used for seven-segment display.
- *day_in_month:* Number of days in each month of a normal year
- *day_in_month_leap:* Number of days in each month of a leap year

2.2.2 Main function

The program continuously call *get_time* and *get_sec* to check if a full minute passed. After every minutes, a sound is played with *play_sound* function.

It also continuously check if interrupt occurs (Digital Lab Sim key press). If keys from 1 to 6 is pressed, corresponding sub function will be called.

```
.text
main:
    li s1, IN_ADDRESS_HEXA_KEYBOARD
    li s2, OUT_ADDRESS_HEXA_KEYBOARD
polling:
check_full_min:
    jal get_time
    jal get_sec
    beq a0, zero, play_sound
continue_polling:
    li s3, 0x01
    sb s3, 0(s1)
    lb s4, 0(s2)
    bne s4, zero, perform
    li s3, 0x02
    sb s3, 0(s1)
    lb s4, 0(s2)
    bne s4, zero, perform
    j back_to_polling
perform:
display_time:
    jal get_time
    # If press 1
    li t0, 0x21
    beq s4, t0, display_hour
    # If press 2
    li t0, 0x41
    beq s4, t0, display_min
    # If press 3
    li t0, 0xffffff81
    beq s4, t0, display_sec
    # If press 4
    li t0, 0x12
    beq s4, t0, display_day
    # If press 5
    li t0, 0x22
    beq s4, t0, display_month
    # If press 6
```

```
    li t0, 0x42
    beq s4, t0, display_year
back_to_polling:
    j polling
```

2.2.3 Play_sound function

The syscall number 31 is **MidiOut**, whose outputs simulated MIDI tone to sound card. The values in register *a0, a1, a2* and *a3* represent pitch, duration, instrument and volume of the sound.

```
play_sound:
# Play a sound
    li a7, 31
    li a0, 69
    li a1, 100
    li a2, 7
    li a3, 50
    ecall
    j continue_polling
```

2.2.4 Get_time function

The syscall number 30 is **Time**, which get the current time (milliseconds since 1 January 1970). The lower 32 bits are stored in *a0* and the higher ones are in *a1*.

The function *get_time* use floating points registers to deal with 64-bit numbers. It firstly stores 32 higher bits in a 64-bit floating point register *ft0*.
Then, this value is multiplied with 2^{32} to shift it 32 bits to the left. The value in register *a0* is added to *ft0* to get the complete value, which indicates the total of milliseconds since 1 January 1970 up to present.
Finally, to get the number of seconds since 1 January 1970, we divides it by 1000.
After this process, we get the total number of seconds, which fits in 32 bits. Therfore, we convert it back to unsigned integer and stores in register *s0*.

```
get_time:
# Get the time using syscall and convert to seconds and assign to s0
    li a7, 30
    ecall
    # Load the time into floating point register
    fcvt.d.wu ft0, a1 # ft0 stores 32 higher bits
    li t0, 0x80000000
    fcvt.d.wu ft1, t0
    li t0, 2
```

```
    fcvt.d.wu ft2, t0
    fmul.d ft2, ft1, ft2 # ft2 = 2^32
    fcvt.d.wu ft1, a0 # ft1 stores 32 lower bits
    fmadd.d ft0, ft0, ft2, ft1 # ft0 = ft0 * 2^32 + ft1 is the complete time
    li t0, 1000
    fcvt.d.wu ft1, t0 # ft1 = 1000
    fdiv.d ft0, ft0, ft1 # ft0 = ft0 / 1000 to convert millisecond to second
    fcvt.wu.d s0, ft0
```

2.2.5 Get hour, minute, second, day, month and year functions

After *get_time*, we have already had the number of seconds from 1 January 1970 stored in *s0*. Functions *get_sec*, *get_min* and *get_hour*

To get the current second, we take the remainder of *s0* dividing by 60 (60 seconds in a minute).

```
get_sec:
# Get the second from total second in s0 and assign to a0
    li t0, 60
    remu a0, s0, t0
    jr ra
```

To get the current minute, we take the remainder of *s0* dividing by 3600 (3600 seconds in an hour) and then divide it by 60.

```
get_min:
# Get the minute from total second in s0 and assign to a0
    li t0, 3600
    li t1, 60
    remu a0, s0, t0
    divu a0, a0, t1
    jr ra
```

To get the current hour, we take the remainder of *s0* dividing by 86400 (86400 seconds in an hour) and then divide it by 3600. However, the time is taken at GMT+0 and Vietnam is at GMT+07, so we need to add 7 to the result.
This leads to a situation that the hour we get greater than 24, so we have to check whether it's the next day or not. If the next day comes in our timezone, we set a new hour and load the value 1 to *s11*, which will be used as a flag for the counting day part.

```
get_hour:
# Get the hour from total second in s0 and assign to a0
    li t0, 86400
```

```
    li t1, 3600
    remu a0, s0, t0
    divu a0, a0, t1
    li t0, 16
    bgt a0, t0, next_day
    addi a0, a0, 7
    j continue_get_hour
next_day:
    addi a0, a0, -17
    li s11, 1
continue_get_hour:
    jr ra
```

The function *get_year_month_day* retrieves the current year, month and day, then stores them in register *a0, a2* and *a1*, respectively.
Firstly, it calls the *get_hour* to check the *next_day*. The number of day is the quotient of *s0* to 86400 plus 1 (plus 2 if the next day comes).
To get the current year, we first subtract the number of days by 10957 (to get the counting starts from 1 January 2000). This is just for easier calculation. Then we divide it into 400-year periods, 100-year periods, 4-year periods and 1-year periods. This is because of the criteria for a year to be leap year. To be specific, the current year will be $400x + 100y + 4z + t$ with x, y, z, t are the number of 400-year, 100-year, 4-year and 1-year period respectively. This value is stored in *a0*.
The remainder, i.e. the number of day left in this year is stored in register *a1* to calculate month and day. We have a *check_leap* function to check if current year is leap or not. If it's a leap year, the address of *day_in_month_leap* is loaded to *t2*. Otherwise, the address of *day_in_month* is loaded to *t2*. Then, a loop is created to loop through each element in the array, which indicates the number of days from January to December. By this way, we get the current day and month in registers *a1* and *a2*.

```
get_year_month_day:
# Get the year, month, day from total second in s0 and assign to a0, a2, a1
    addi sp, sp, -4
    sw ra, 0(sp)
    jal get_hour # Get_hour to check the flag s11 for next day
    lw ra, 0(sp)
    addi sp, sp, 4

    li t0, 86400 # 86400 seconds in a day
    div t1, s0, t0 # Get the number of days from 1970 to present
    addi t1, t1, 1
    beq s11, zero, continue
    addi t1, t1, 1 # Next day for GMT+7
```

```
continue:
    li t0, -10957
    add t1, t1, t0 # Get the number of days from 2000 to present
    # Consider 400 years
    li t0, 146097 # Number of days in 400 consecutive years
    div t2, t1, t0
    remu t1, t1, t0 # Number of days left after 400x years
    li t0, 400
    mul a0, t2, t0
    # Consider 100 years
    li t0, 36524 # Number of days in 100 consecutive years
    div t4, t1, t0
    remu t1, t1, t0 # Number of days left after 100y years
    li t0, 100
    mul t4, t4, t0
    add a0, a0, t4
    # Consider 4 years
    li t0, 1461 # Number of days in 4 consecutive years
    div t2, t1, t0
    slli t2, t2, 2
    add a0, a0, t2
    remu t1, t1, t0
    # Consider 1 years
    li t0, 365
    div t3, t1, t0
    add a0, a0, t3
    remu a1, t1, t0

    li t6, 0 # index
    j check_leap # load day list to t2
continue_get_day:
loop:
    slli t3, t6, 2
    add t4, t2, t3
    lw t5, 0(t4)
    ble a1, t5, end_get_day
    sub a1, a1, t5
    addi t6, t6, 1
    j loop
end_get_day:
```

```
    addi a2, t6, 1 # a2 store month
    jr ra
```

The *check_leap* function considers the criteria for a year to be leap year.

```
check_leap:
# Check year stored in a0
    li t0, 4
    remu t1, a0, t0
    bne t1, zero, end_not_leap
    li t0, 100
    remu t1, a0, t0
    bne t1, zero, end_leap
    li t0, 400
    rem t1, a0, t0
    bne t1, zero, end_not_leap
end_leap:
    la t2, day_in_month_leap
    j continue_get_day
end_not_leap:
    la t2, day_in_month
    j continue_get_day
```

2.2.6 Display number function

The function *display_number* displays the last 2 digits of the value stored in *a0*.

```
display_number:
# display the number in a0
    li t1, 10
    la t4, table
    # 2nd digit
    rem t2, a0, t1
    li t0, SEVENSEG_RIGHT
    add t2, t2, t4
    lb t5, 0(t2)
    sb t5, 0(t0)
    # 1st digit
    div a0, a0, t1
    rem t2, a0, t1
    li t0, SEVENSEG_LEFT
    add t2, a0, t4
```

```
    lb t5, 0(t2)
    sb t5, 0(t0)
    jr ra
```

2.2.7 Display hour, minute, second, day, month and year functions

These functions combine all the above to display the current hour, minute, second, year, month or day. Since the current day and month are stored in *a1* and *a2*, we have to move them to *a0* before calling the *display_number*.

```
display_hour:
    jal get_hour
    jal display_number
    j back_to_polling
display_min:
    jal get_min
    jal display_number
    j back_to_polling
display_sec:
    jal get_sec
    jal display_number
    j back_to_polling
display_day:
    jal get_year_month_day
    mv a0, a1
    jal display_number
    j back_to_polling
display_month:
    jal get_year_month_day
    mv a0, a2
    jal display_number
    j back_to_polling
display_year:
    jal get_year_month_day
    jal display_number
    j back_to_polling
```

3 Source code and Demonstration Videos

Note: The code of problem 17 will cause delay if running at speed of 30 ins/sec. However, max speed may cause lag and some unknown errors. Click me