Prelab Report: Design and Implementation of a Programmable Digital Clock

Arye Mindell, LMU EECE 3200

***Abstract*—This report outlines the development of a programmable digital clock using ARM Assembly Language on a Raspberry Pi. The clock operates in "military" time and is updated every second. The program consists of modular subroutines for setting the clock, updating the time, implementing delays, and displaying the output. This project demonstrates the integration of hardware and software in embedded systems, emphasizing modular programming and real-time operation.**

I. Introduction

THIS lab focuses on creating a programmable digital clock on a Raspberry Pi using ARM Assembly Language. The clock allows the user to set the initial time in 'military' format (e.g., "23:59") via a keyboard and continuously updates and displays the time every second on the terminal. The design employs modular subroutines for input, time updating, delay implementation, and output display. This prelab outlines the design and implementation approach, as well as the testing methodology for validating the functionality.

II. System Description

THE system is implemented on a Raspberry Pi microcontroller. The key hardware components include:

- Raspberry Pi 4B
- USB keyboard and monitor for input/output

The software consists of four assembly source files:

- `clock.s`: The main routine that calls subroutines for input, updating, and output.
- `setClock.s`: A subroutine to input the initial time (hours and minutes).
- `delay.s`: A subroutine implementing a one-second delay loop.
- `display.s`: A subroutine to display the current time on the terminal.

III. Program Code Overview

THE modular structure of the clock program begins with the main routine, which initializes the clock by invoking the `setClock` subroutine to capture user input. A continuous loop is then used to update the time by incrementing seconds, minutes, and hours as needed. To maintain accurate timing, the `delay` subroutine is employed to create a one-second delay between updates. Finally, the `display` subroutine outputs the current time in the "HH:MM:SS" format on the terminal.

The program is developed and simulated using Geany, GNU Assembler, and GNU Project Debugger.

IV. Flowchart

THE flowchart in Fig. 1 outlines the overall structure of the clock program, including initialization, updating, and displaying time.

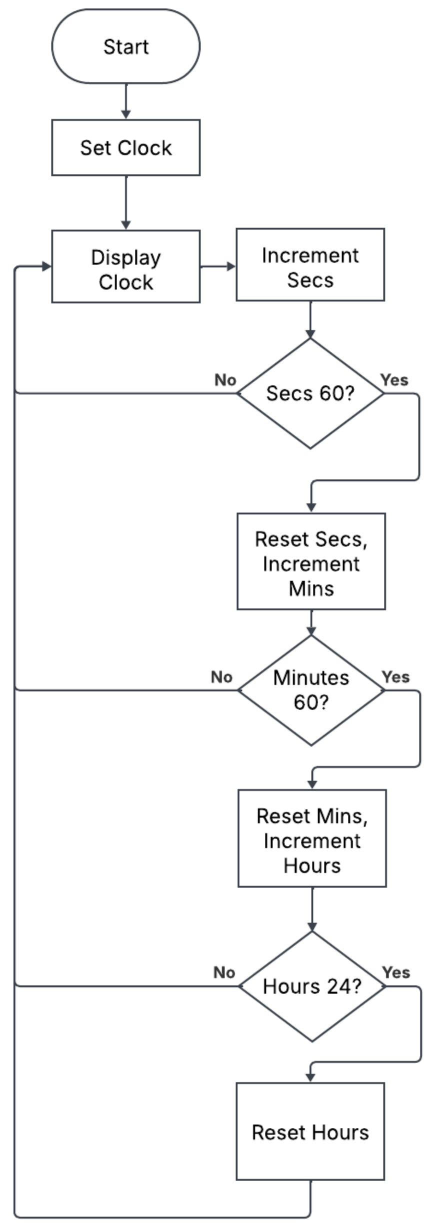

Fig. 1. Flowchart of the clock program algorithm.

V. Test Cases

To validate the clock program, the following critical test cases will be simulated:

- Transition from `12:59:59` to `13:00:00`.
- Transition from `23:59:59` to `00:00:00`.
- Normal operation over several minutes to ensure accuracy.

An additional script, `clockSim.s` simulates each subroutine for testing. Simulation results will be captured using CPUlator.

VI. Limitations and Challenges

Potential limitations in this project include the accuracy of the delay subroutine, which is highly dependent on the timing of instruction cycles. Another challenge is ensuring robust user input validation to accept only valid time formats. Additionally, if the clock loses 10 microseconds per second, it will accumulate a loss of 864 milliseconds per day, resulting in a total loss of 5.26 minutes per year. For the clock to maintain an accuracy of one second per year, it must lose less than 0.032 microseconds per second.

References

[1] Loyola Marymount University, Department of Electrical Engineering and Computer Science, “EECE 3200: Lab 2 - Programmable Digital Clock,” Spring 2025.

Appendix A
Assembly Code Files

```

@EECE 3200
@Arye Mindell
@clock main function: runs the main count loop
    of the clock. calls set_clock to initialize
    the clock and display_clock to print the
    clock output

.global _start
.extern set_clock
.extern sleep_one_second
.extern display_clock

_start:
    bl set_clock

count_loop:
    CMP r0, #60                 @ Check if the
        seconds counter has reached 60
    BNE check_minutes           @ If not, check
        minutes

    ADD r6, r6, #1              @ Increment
        minutes counter
    MOV r0, #0                  @ Reset the
        seconds counter

check_minutes:
    CMP r6, #60                 @ Check if the
        minutes counter has reached 60
    BNE check_hours             @ If not, check
        hours

    ADD r7, r7, #1              @ Increment hours
        counter
    MOV r6, #0                  @ Reset the
        minutes counter

check_hours:
    CMP r7, #24                 @ Check if the
        hours counter has reached 24
    BNE continue_count          @ If not, continue
         counting

    MOV r7, #0                  @ Reset the hours
        counter

continue_count:
    bl sleep_one_second         @ Pause for 1
        second
    ADD r0, r0, #1              @ Increment the
        seconds counter
        bl display_clock
    B count_loop                @ Repeat the loop
```

Listing 1. clock.s

```
@EECE 3200
@Arye Mindell
@setClock subroutine: accepts input from user to
     set the initial values of the clock

.text
.global set_clock
set_clock:
    MOV r0, #0                  @ Initialize
        seconds counter to 0
    MOV r6, #58                  @ Initialize
        minutes counter to 58
    MOV r7, #23                  @ Initialize
        hours counter to 23
bx lr
```

Listing 2. setClock.s

```
@EECE 3200
@Arye Mindell
@sleep_one_second subroutine: waits roughly one
    second based on processor clock rate. must
    be adjusted based on system

.section .text
.global sleep_one_second

sleep_one_second:
    PUSH {r4, r5, lr}       @ Save registers and
        return address

    LDR r4, =0x00100000      @ Load 1,000,000 (1
         MHz clock, 1 second delay)
loop_outer:
    SUBS r4, r4, #1         @ Decrement outer
        loop counter
    BNE loop_outer          @ If not zero, keep
        looping

    POP {r4, r5, lr}        @ Restore registers
        and return address
    BX lr                   @ Return to caller
```

Listing 3. delay.s

```
@EECE 3200
@Arye Mindell
@display_clock subroutine: converts clock values
     to ascii and outputs them to terminal


.section .text
.global display_clock

display_clock:
    PUSH {r0, r5, r6, r7, lr}                    @
        Save the current seconds counter value
```

```

    @ Calculate seconds for the display
    BL get_led_code           @ Call the subroutine to calculate display value
    MOV r5, r1                @ Store the seconds display code in r5

    @ Calculate minutes for the display
    MOV r0, r6                @ Load the minutes counter into r0
    BL get_led_code           @ Call the subroutine to calculate display value
    LSL r1, r1, #8           @ Shift the minutes display code left by 16 bits
    ORR r5, r1, r5            @ Combine minutes and seconds into r1

    @ Calculate hours for the display
    MOV r0, r7                @ Load the hours counter into r0
    BL get_led_code           @ Call the subroutine to calculate display value
    LSL r1, r1, #16           @ Shift the hours display code left by 16 bits
    ORR r1, r1, r5            @ Combine hours with minutes and seconds in R1

    @ Write the clock value to display

    POP {r0, r5, r6, r7, lr}                    @ Restore the seconds counter value
        Bx lr

@ Subroutine to calculate the seven-segment display value
get_led_code:
    PUSH {r4, lr}                 @ Save return address

    BL divide_by_10               @ Call the division subroutine
    @ After the call:
    @ r1 = tens digit (quotient)
    @ r0 = ones digit (remainder)

    @ Convert the ones digit to ascii
    MOV r2, r1                    @ Move tens digit to r2
    BL number_to_ascii         @ Call the function to get the segment code
    MOV r4, r1                    @ Store converted ones digit in R4

    @ Convert the tens digit to ascii
    MOV r0, r2
    BL number_to_ascii         @ Call the function to get the segment code
    LSL r1, r1, #4                @ Shift the tens digit value left by 4 bits

    @ Combine tens and ones digits into a single word
    ORR r1, r1, r4                @ Combine tens (shifted) and ones into r1

    POP {r4, lr}                  @ Restore return address
    BX lr                         @ Return to caller

number_to_ascii:
    PUSH {lr}                    @ Save return address
    CMP r0, #9                   @ Check if the number is greater than 9
    BHI invalid_input            @ Branch if invalid input

    ADD R1, R0, #48
    B return_value               @ Skip invalid case

invalid_input:
    MOV r1, #0                   @ Set r1 to 0 for invalid input (no segments lit)

return_value:
    POP {lr}                     @ Restore return address
    BX lr                        @ Return to caller

@subroutine to divide by 10 and save remainder (used to split digits)
divide_by_10:
    PUSH {lr}                 @ Save return address
    MOV r2, #10               @ Divisor (10)
    MOV r1, #0                @ Initialize quotient (r1) to 0

divide_loop:
    CMP r0, r2                @ Compare dividend (r0) with divisor (10)
    BLT end_divide            @ If dividend < 10, stop the division
    SUB r0, r0, r2            @ Subtract 10 from the dividend
    ADD r1, r1, #1            @ Increment quotient
    B divide_loop             @ Repeat the loop

end_divide:
    @ At this point:
    @ r1 = quotient (tens digit)
    @ r0 = remainder (ones digit)
    POP {lr}                  @ Restore return address
    BX lr                     @ Return to caller
```

Listing 4. display.s