EE 306-03: Microprocessors

Term Project - Report

« DnDicer »

Bogdán Itsám Dorantes-Nikolaev - 042101002

Onur Keleş - 042001049

Ömer Mert Yıldız - 042002008

Department of Engineering, MEF University

Prof. Tuba Ayhan

May 17th, 2024

1. Abstract

This ARM assembly program operates as a digital dice roller on a microcontroller platform, interfacing with hardware peripherals. It supports 11 distinct dice configurations: 2-sided, 4-sided, 6-sided, 8-sided, 10-sided, 12-sided, 20-sided, 32-sided, 64-sided, and 99-sided. Users select the dice type via switches, with each switch corresponding to one dice type. The program includes error checking, displaying "Er" on the HEX display for multiple toggles and "SL" for no selection. The ready state or initial display is indicated by "rdy" on the HEX display. A timer provides seed values for the random number generator, with results shown on HEX displays, ensuring clear and immediate visual feedback of the dice roll outcome.

2. Flowchart

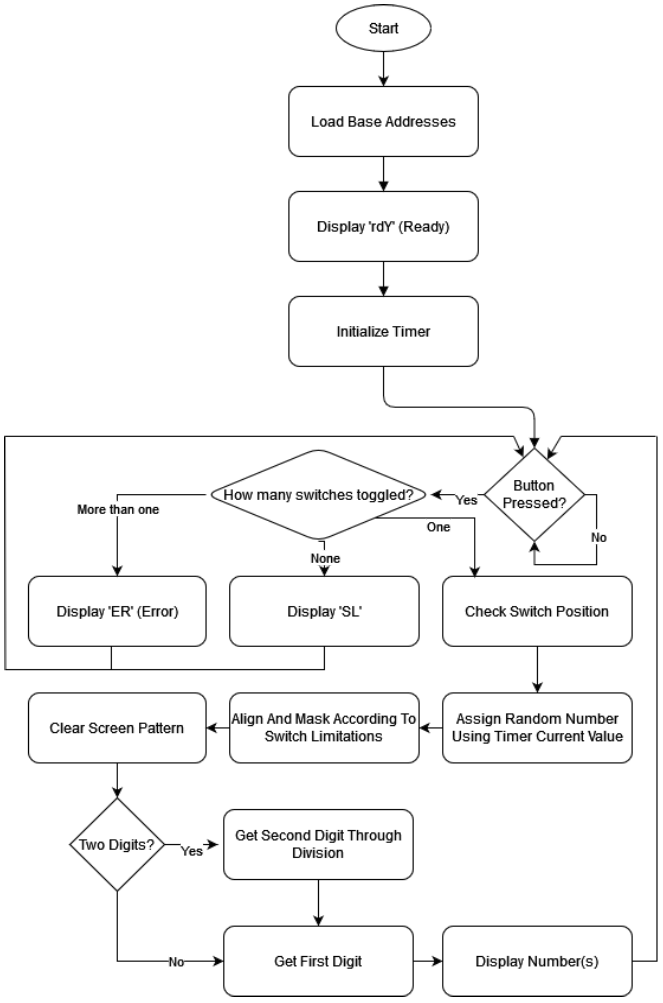

Figure 1. Flowchart of "DnDicer"

3. User Manual

Normal Usage:

Upon launch, the device prompts that it is in a ready state:

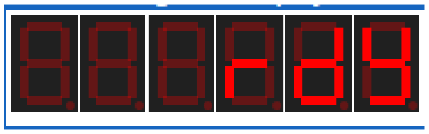

Figure 2. Ready message indicating a successful program launch.

Choose a desired dice (or coin) using the toggle switches:

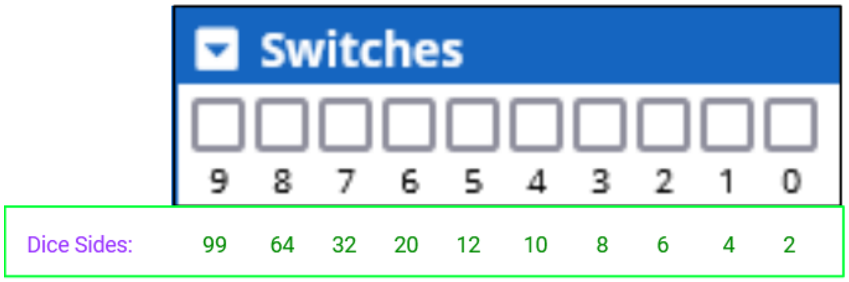

Figure 3. Toggle-switch legend for their corresponding dice side values.

After selecting a dice, use any of the push-buttons to roll the dice:

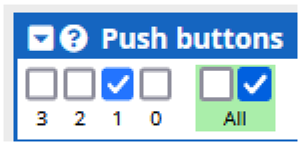

Figure 4. Push-buttons that perform the dice-rolling.

A usage example for the common 6-sided dice:

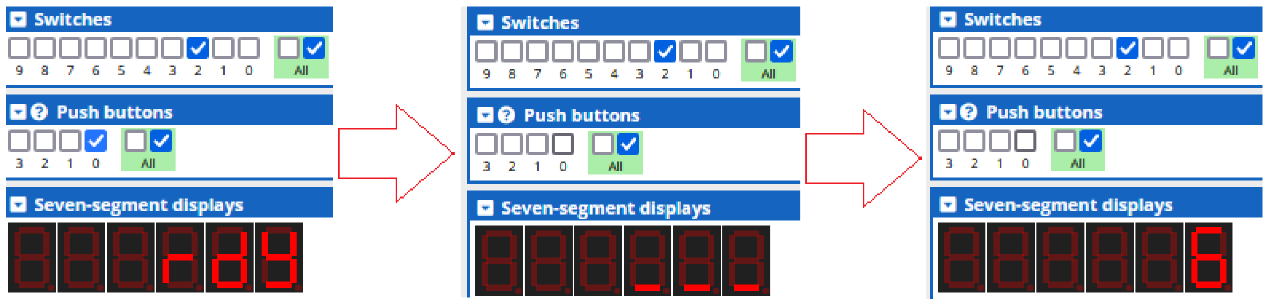

Figure 5. Program set to 6-sided dice and generated the value of 6.

Abnormal Usage:

Duplicate Switching:

When more than one switch is toggled and the dice are rolled using the push button, the device will prompt an error message:

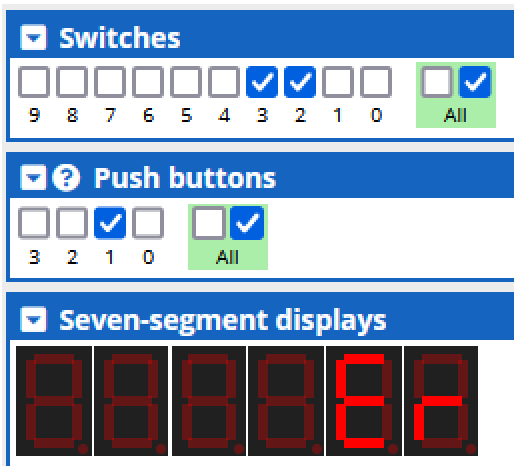

Figure 6. Error indication when more than one switch is toggled and dice rolled.

Undefined Switching:

When no switches are toggled and the dice are rolled using the push button, the device will prompt the user to make a selection:

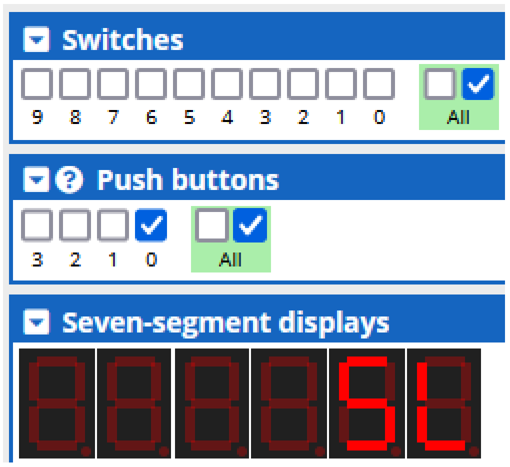

Figure 7. Selection prompting when no switches are toggled and dice rolled.

4. Technical Background

- **4.1 Hardware Background**

  The hardware setup includes push buttons, switches, HEX displays, and a private timer. Push buttons are mapped to the base address *0xFF200050* and are used to initiate the dice roll on press and release. Switches, mapped to the base address *0xFF200040*, are used to specify dice types (2-sided, 4-sided, 6-sided etc.). The 7 Segment Digital Display (SSD) in HEX, it is mapped to *0xFF200020*, and shows the results of the dice roll or gives warnings to the user depending on the action. The private timer, mapped to *0xFFFEC600*, provides the seed for the random number generator that is inspired from other high level compilers (such as Java).

  Memory initialization has, *dice_sides_array*, which contains values for each dice type, and the seven-segment table, that contains hex codes for displaying numbers on the SSD. Additionally, a variable (*variable_x*) is used to store intermediate values during computation.

  Upon startup, the program loads base addresses for peripherals into registers, configures the timer with a load value, enables auto-reload, and displays the initial "ready" state ("rdY") on the SSD.

- **4.2 Displaying**
  - **4.1.1 Number Results**

  The display result subroutine handles both single-digit and two-digit results. For single-digit results, it clears the previous display just in case there was a two digit display before, fetches the relevant seven-segment code, and displays the number then returns to the main loop. For two-digit results, it separates the result into two distinct digits, fetches and combines their respective seven-segment codes, and displays the combined result and returns to the main loop.

  - **4.1.2 Delay Result**

  A delay subroutine displays a loading pattern on the SSD to simulate processing time, adding realism and a sense of anticipation to the user during the dice roll.

- **4.2 Calculations**
  - **4.2.1 Switch Counter**
- Error checking is performed by the *is_switch_error* subroutine, which ensures that only one switch is activated at a time. This subroutine uses a bit counting algorithm to count the number of activated switches. It starts by initializing a counter register to zero. The switch state, stored in a register, is checked in a loop where the least significant bit (LSB) is masked and tested. If the LSB is 1, indicating an activated switch, the counter is incremented. The switch state is then shifted right by one bit, and the process repeats until all bits are checked. If the counter exceeds one, indicating that more than one switch is activated, the subroutine displays an error message ("Er") on the HEX display and returns to the main loop. This method ensures that the system correctly handles multiple switch activations, which is not in the scenario.

- **4.2.2 Switch Decider**

- The *which_switch* subroutine determines the specific dice type selected by the user. It compares the switch state to predefined values representing each possible dice type. Starting with the lowest switch value, it checks each switch in sequence. If a match is found, the relevant dice type value is loaded into a register at the end of the subroutine. This value is then stored in a predetermined memory location (*variable_x*) for use in the random number generation process.

- **4.2.3 Random Number Generator**

- First, the *random_number* subroutine loads the seed value from the timer register at *0xFFFEC600* into a register. Next, it uses the switch value, previously determined by the *which_switch* subroutine, to fetch the relevant number of sides from the *dice_sides_array*.

  The fetched timer value undergoes a modulo operation with the number of sides of the selected dice. This operation ensures that the random number falls within the valid range of the dice sides. The modulo operation is performed using an AND instruction to mask the number of sides of the dice to the corresponding number determined before.

  Finally, to avoid a result of zero, which is not a valid dice roll, one is added to the result. This part is optional but implemented as is because physical dice do not tend to have 0. The resulting random number is then stored in a predetermined memory location (*variable_x*) for further use in displaying the result.

5. Tackled Problems

Random Number Generation:

Our team began by exploring various techniques for creating random numbers. Initially, we experimented with exclusive-OR (EOR) operations and a seed value derived from a register. But after some trials, we decided to use a private timer instead. This choice was made because the timer provided a more reliable and efficient way to achieve randomness.

Switch Addressing:

While working on assigning each switch to correspond with a specific dice, we ran into a bit of a snag with the incrementation steps. At first, we tried lining up the switch addresses with each potential position through direct comparison. However, this led to some hiccups during the incrementation phase. We found a neat workaround by shifting the comparison value instead of multiplying it, which smoothly clarified the issues. This method ensured flawless switch addressing and enabled precise dice selection without any unexpected complications.

7-Segment Display Addressing:

Our first attempts at showing the randomly generated numbers faced a hurdle: they were limited to the rightmost segment of the HEX display. This became problematic when the numbers extended beyond two digits, mostly due to the differences between decimal and binary representations. To tackle this, we introduced a digit counter mechanism. By deducting the base decimal value from our random number, we could figure out the necessary number of digits and adjust the display accordingly. This clever strategy allowed us to display multi-digit numbers effectively on the HEX display, significantly enhancing the system's capability and user experience.

Avoiding Clobbered Registers:

We had hard times with determining where to use methods to avoid clobberation in the program. We tried to use every register that has been modified in a subroutine that was accessed. Our first version of the code had no problems with using registers directly for passing values as variables where we disabled the warning for clobberation. But our code was not passing the value we wanted to have in multiple subroutines. Therefore we used a predetermined area in the memory and used that area so we could pass values between subroutines.