MEF ÜNİVERSİTESİ

ÖĞRENMEYE SINIRSIZ ÖZGÜRLÜK

ARM Assembly Roulette Game

RAHIQ ISSAM MOHAMMED ALI AL HUSLAN - 042101140

RIZA ASLAN - 042001010

Abstract

This report presents the development of an interactive roulette game implemented in ARMv7 Assembly Language on the DE1-SoC platform. The game simulates a spinning roulette wheel using ten LEDs that illuminate sequentially. Players place bets by selecting one of ten switches (SW0-SW9) before gameplay begins. The LED sequence cycles continuously until the player presses KEY0 to stop the wheel. The final illuminated LED determines the game outcome—if it matches the player's chosen switch, "YES" displays on the 7-segment displays; otherwise, "END" appears. The corresponding LED number (0-9) is shown on the HEX3 display. This project demonstrates key embedded systems programming concepts including memory-mapped I/O, bit-level operations, and direct hardware control.

---

1. Introduction

Project Overview

Our embedded systems project implements a digital roulette game that bridges software logic with hardware control. The system creates an engaging gambling simulation where users interact directly with DE1-SoC board components to place bets and control game flow.

System Architecture

The game architecture consists of four main hardware interfaces:

- LED Array: Ten LEDs (LEDR0-LEDR9) create the visual roulette wheel
- Input Switches: Ten switches (SW0-SW9) allow bet placement
- Control Interface: KEY0 button provides game control
- Display System: 7-segment displays provide game feedback

Design Objectives

We aimed to create a system that demonstrates fundamental embedded programming concepts while providing an intuitive user experience. The design emphasizes efficient bit manipulation, real-time input processing, and direct hardware control.

---

2. System Configuration and Setup

2.1 Memory-Mapped I/O Configuration

The DE1-SoC platform uses memory-mapped I/O architecture, where hardware components are controlled through specific memory addresses. Our system utilizes four primary memory regions:
Hardware Component Mapping:

- LED Base Address (0xFF200000): Controls the ten red LEDs through a 32-bit register where bits 0-9 correspond to LEDR0-LEDR9

- Switch Base Address (0xFF200040): Reads switch states through a 32-bit register where bits 0-9 represent SW0-SW9 positions
- Button Base Address (0xFF200050): Monitors push button states where bit 0 represents KEY0 status
- 7-Segment Base Address (0xFF200020): Controls four 7-segment displays (HEX0-HEX3) through separate 8-bit segments

2.2 Register Allocation Strategy

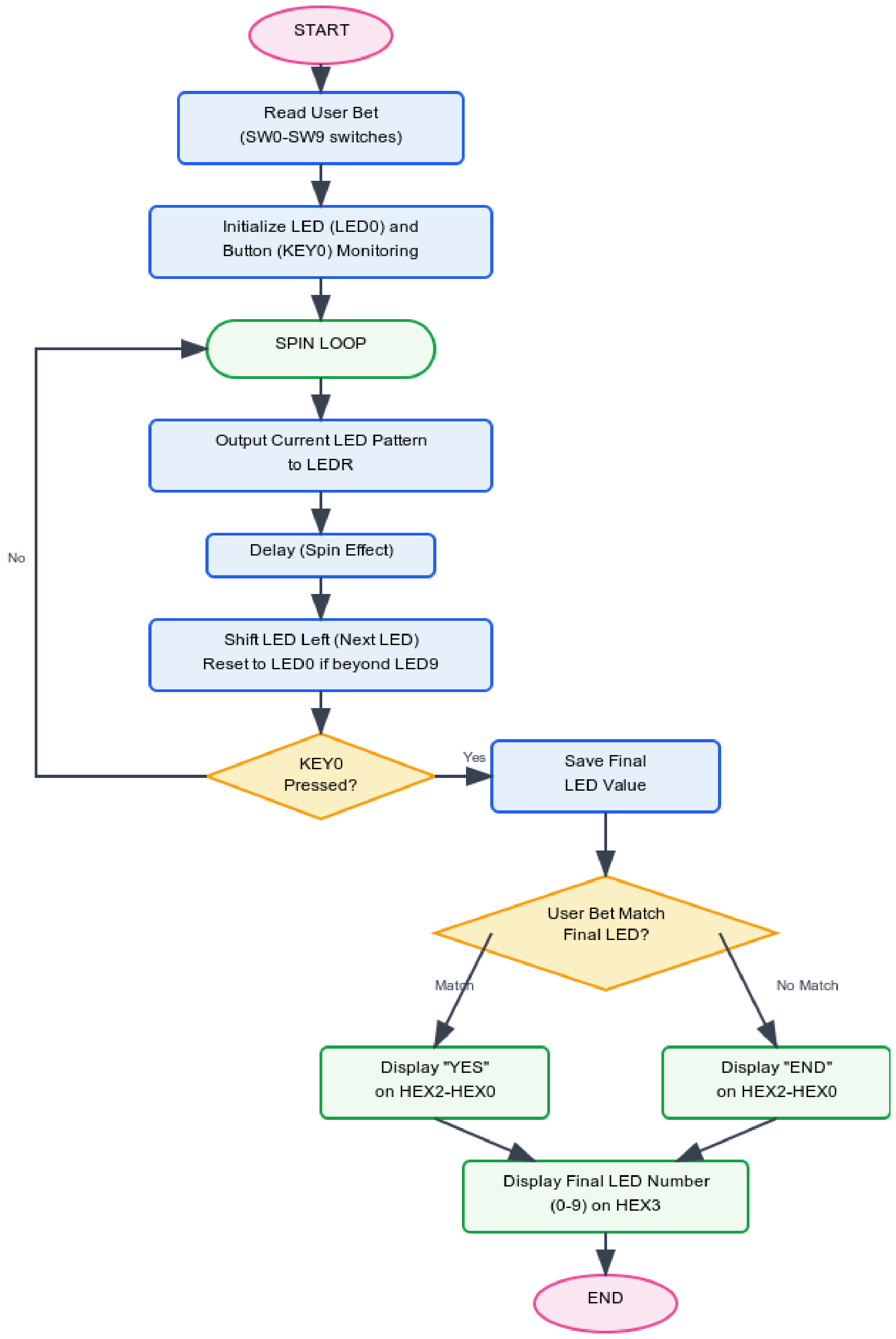
We designed a systematic register allocation approach to manage game state efficiently:

- R1: Base address storage for hardware components
- R2: Temporary data storage for I/O operations
- R3: Player bet storage (switch configuration)
- R4: LED base address (persistent)
- R5: Current LED pattern (spinning state)
- R6: LED index counter for result calculation
- R7: Final LED pattern (game result)
- R8: Delay counter for timing control
- R10: Button base address (persistent)
- R11: Button state reading
- R12: Temporary calculation register

This allocation ensures efficient data flow while maintaining clear separation between different system functions.

---

3. Flowchart and Game Algorithm Implementation


START
Read User Bet (SW0-SW9 switches)
Initialize LED (LED0) and Button (KEY0) Monitoring
SPIN LOOP
Output Current LED Pattern to LEDR
Delay (Spin Effect)
Shift LED Left (Next LED) Reset to LED0 if beyond LED9
KEY0 Pressed?
No
Yes
Save Final LED Value
User Bet Match Final LED?
Match
No Match
Display "YES" on HEX2-HEX0
Display "END" on HEX2-HEX0
Display Final LED Number (0-9) on HEX3
END

3.1 Bet Placement Algorithm

Input Acquisition Process: The betting phase implements a straightforward input reading algorithm. The system reads the 32-bit value from the switch base address and stores it for later comparison. This approach captures all switch states simultaneously, allowing players to place multiple bets or change their selection before starting the game.
Data Storage Method: The switch configuration is preserved in a dedicated register throughout the entire game cycle. This ensures the player's original bet remains unchanged during the spinning phase, preventing accidental modifications due to switch movements during gameplay.

3.2 LED Spinning Algorithm

Sequential Illumination Logic: The spinning mechanism uses a bit-shifting algorithm to create smooth LED transitions. We start with a binary pattern of 0000000001 (only LED0 lit) and apply left logical shifts to move the active bit position.
Pattern Progression:

- Initial state: 0000000001 (LED0)
- After 1st shift: 0000000010 (LED1)
- After 2nd shift: 0000000100 (LED2)
- ...continuing until LED9

Wraparound Implementation: When the pattern reaches 10000000000 (decimal 1024), indicating movement past LED9, the algorithm resets to 0000000001. This creates a continuous cycling effect without gaps.
Timing Control: Each LED transition includes a calibrated delay to ensure visibility. The delay algorithm counts down from a predetermined value (0x8000), creating approximately 100ms intervals between LED changes. This timing provides optimal visual feedback while maintaining responsive user control.

3.3 Input Monitoring Algorithm

Continuous Polling Method: During the spinning phase, the system continuously monitors the button base address for KEY0 activation. The polling frequency is synchronized with the LED update cycle to ensure responsive control.
Button State Detection: The algorithm uses bitwise testing to isolate KEY0 status from the 32-bit button register. By testing bit 0 specifically, we avoid false triggers from other buttons while maintaining efficient processing.
Immediate Response Logic: Upon detecting KEY0 activation, the spinning loop terminates immediately, preserving the current LED pattern as the final game result. This ensures fair gameplay where the stopping moment directly determines the outcome.

---

4. Result Determination and Display System

4.1 Win/Loss Logic Algorithm

Bitwise Comparison Method: The result determination uses bitwise AND logic to compare the player's bet pattern against the final LED pattern. This approach efficiently handles multiple bet scenarios and provides accurate matching.
Algorithm Steps:

1. Perform bitwise AND between bet pattern and result pattern
2. If result is non-zero, at least one bit matches (win condition)
3. If result is zero, no bits match (loss condition)

Advantage of Bitwise Approach: This method automatically handles complex betting scenarios. Whether a player bets on one LED or multiple LEDs, the same algorithm correctly determines wins and losses.

4.2 7-Segment Display Control

Character Encoding System: The 7-segment displays require specific bit patterns for each character. We developed a character encoding system where each segment (a, b, c, d, e, f, g, dp) corresponds to one bit in an 8-bit pattern.
Display Message Implementation:

- "YES" Message: Three separate 8-bit codes combined and positioned across HEX2-HEX0
    - Y = 0x6E (segments b, c, d, f, g active)
    - E = 0x79 (segments a, d, e, f, g active)
    - S = 0x6D (segments a, c, d, f, g active)
- "END" Message: Similar encoding for loss condition
    - E = 0x79, N = 0x37, D = 0x5E

Multi-Display Coordination: The algorithm combines multiple character codes into a single 32-bit value using bit shifting and OR operations. This allows simultaneous update of all three displays with one memory write operation.

4.3 LED Number Display Algorithm

Pattern-to-Number Conversion: Converting the final LED pattern to a decimal number requires finding the position of the active bit. Our algorithm uses a counting approach:
Bit Position Detection:

1. Initialize counter to 0
2. Check if bit 0 is active using bitwise AND
3. If not active, shift pattern right by 1 and increment counter
4. Repeat until active bit is found or counter reaches 10
5. Counter value represents LED number (0-9)

Display Output: The calculated LED number is converted to its corresponding 7-segment code using a lookup table containing pre-calculated patterns for digits 0-9. This code is then positioned in the HEX3 display location.

5. Visual System Demonstration

5.1 Winning Scenario Analysis

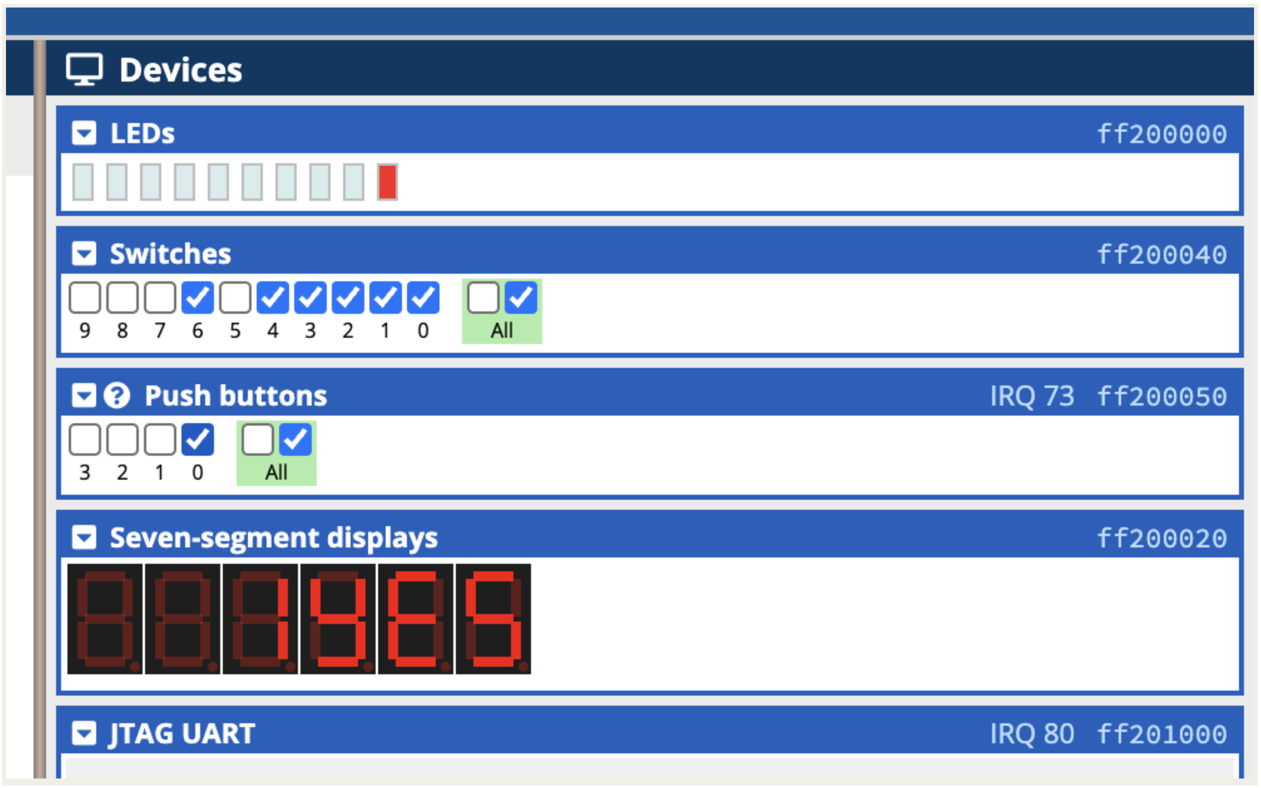

Devices
LEDs ff200000
Switches ff200040
9 8 7 6 5 4 3 2 1 0
All
Push buttons IRQ 73 ff200050
3 2 1 0
All
Seven-segment displays ff200020
JTAG UART IRQ 80 ff201000

Figure 1: Successful Win Condition *System state showing winning game outcome with LED9 illuminated, matching player's bet configuration, resulting in "YES" display and number "9" on HEX3.*

Technical Analysis: The screenshot demonstrates correct implementation of our win detection algorithm. LED9 is active (rightmost position), the switch configuration shows the corresponding bet was placed, and the display system correctly shows "YES" across HEX2-HEX0 with "9" on HEX3.

5.2 Losing Scenario Analysis

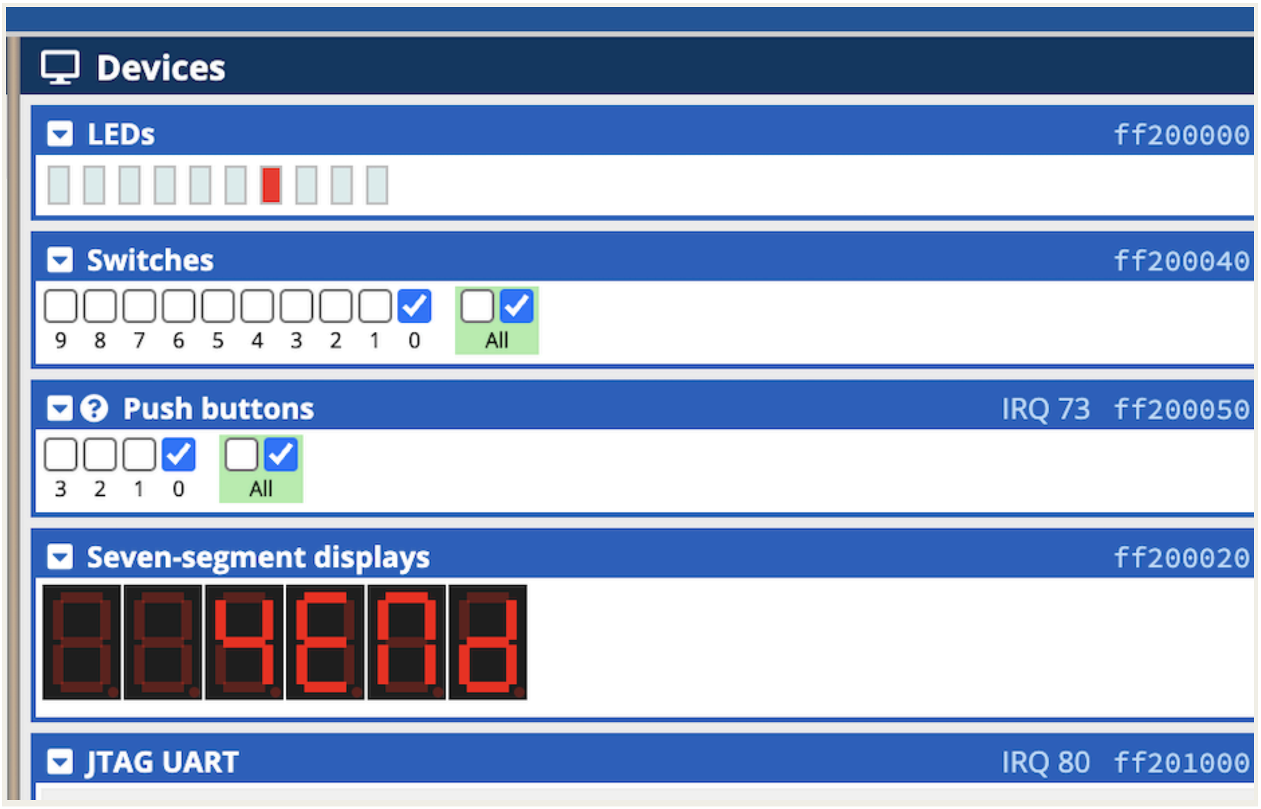


Figure 2: Loss Condition Display *System state showing losing game outcome with LED6 illuminated, not matching SW0 bet selection, resulting in "END" display and number "6" on HEX3.*
Technical Analysis: This screenshot validates our loss detection logic. LED6 is illuminated while SW0 was selected, creating a mismatch condition. The system correctly displays "END" and shows "6" on HEX3, confirming proper pattern-to-number conversion.

5.3 Hardware Validation

Both screenshots confirm proper implementation of:

- Memory-mapped I/O: Correct addresses (ff200000, ff200040, ff200050, ff200020) are active
- Bit manipulation: LED patterns correctly correspond to physical LED states
- Display encoding: 7-segment characters properly formed
- Algorithm flow: Complete game cycle from bet to result display

---

6. Conclusion

This project successfully implemented a functional and interactive roulette game on the DE1-SoC platform using ARMv7 Assembly language. Through direct manipulation of hardware components such as switches, LEDs, buttons, and 7-segment displays, the project demonstrated core concepts of embedded systems programming, including memory-mapped I/O, bit-level operations, and loop control structures.

Beyond technical execution, the project offered valuable insight into the practical challenges of low-level programming. It required precise timing control, real-time user input handling, and efficient display management—skills fundamental to embedded system design. By bridging software logic with hardware behavior, the game served as both a learning tool and a proof of concept for real-world embedded applications. Ultimately, the project deepened our understanding of how hardware and software interact at the most fundamental level.

Appendix

A.1 Memory Map Reference

Component	Base Address	Data Width	Function
LED Array	0xFF200000	32-bit	LED0-LED9 control
Switch Array	0xFF200040	32-bit	SW0-SW9 input
Push Buttons	0xFF200050	32-bit	KEY0-KEY3 input
7-Segment Displays	0xFF200020	32-bit	HEX0-HEX3 output

A.2 Algorithm Complexity Analysis

- LED Spinning: O(1) per cycle (constant time bit shifting)
- Result Determination: O(1) (single bitwise operation)
- Number Conversion: O(log n) where n = LED count (bit counting algorithm)
- Display Update: O(1) (direct memory write operations)

A.3 7-Segment Character Encoding Reference

Character	Hex Code	Binary Pattern	Active Segments
0	0x3F	00111111	a,b,c,d,e,f
1	0x06	00000110	b,c
Y	0x6E	01101110	b,c,d,f,g
E	0x79	01111001	a,d,e,f,g
S	0x6D	01101101	a,c,d,f,g
N	0x37	00110111	a,b,c,e,f
D	0x5E	01011110	b,c,d,e,g