Intelligent Digital Credit Management System for University Mess/Canteen

November 2, 2025

Team Members

1. 241CS206, Shanthi Alluri, alurishanthi.241cs206@nitk.edu.in,
2. 241CS224, Deekshitha Gowda, deekshithaum.241cs224@nitk.edu.in,
3. 241CS257, Somyak Priyadarshi Mohanta, somyakpriyadarshimohanta.241cs257@nitk.edu.in,

GitHub Link

`https://github.com/somyaknotfound/S2-18-MessCreditManagement`

Abstract

Traditional university mess and canteen systems rely on manual or basic digital billing, lacking real-time recommendation and dynamic credit management. This project presents an **Intelligent Digital Credit Management System** implemented using **Logisim** to automate student meal authentication, credit validation, and financial transactions with flexible action-based credit adjustment. The system employs a **Finite State Machine (FSM)** with five sequential states (IDLE, AUTH, RECOMMEND, TRANSACTION, UPDATE) controlling the entire process from meal request to balance modification. The FSM uses custom **Next State Logic** derived from minimized truth tables, integrating a **counter-comparator unit** for precise authentication delays. A modular **Arithmetic Logic Unit (ALU)** performs operations based on 2-bit action type: deducting credits for mess consumption (00) or canteen purchases (10), and adding credits when mess is skipped (01). The ALU includes real-time credit verification and balance calculation. A **Display Driver** circuit handles BCD digit extraction for 7-segment display of remaining balance. The design demonstrates

secure authentication, automatic credit validation with action-aware processing, and accurate transaction handling.

1 Introduction

The management of financial transactions within university campuses presents significant operational challenges. Traditional methods—manual token systems, outdated card readers, or basic point-of-sale terminals—are prone to human error, lack security, and offer no intelligent decision support [1]. These limitations result in extended queuing times, credit disputes, and inefficient resource allocation. Existing systems fail to accommodate flexible operations like refunds for skipped meals or differential pricing between mess and canteen services. An efficient, reliable, and automated system is essential to enhance user experience for students and streamline canteen management operations.

The solution demands strict adherence to sequential logic principles to ensure financial data integrity [2]. The transaction workflow is inherently sequential: authentication, credit verification, meal option presentation, transaction type decoding (ate mess, skipped mess, or ate canteen), transaction execution with appropriate add/subtract operation, and secure balance update—all without timing errors or data corruption. This necessitates a **Finite State Machine (FSM)** for disciplined control flow, orchestrating interactions between the credit register, meal cost memory (ROM), and arithmetic processing unit [4]. The system supports bidirectional credit operations: debit transactions for meal consumption and credit additions for skipped pre-paid meals. This dual-mode operation requires sophisticated ALU control logic with a 2-bit action-type control signal enabling differentiation between three transaction modes.

Contemporary digital design emphasizes modularity, reusability, and formal verification [2]. Discrete logic blocks for arithmetic, control, and display functions simplify testing and debugging. Tools like Logisim facilitate rapid prototyping and structural verification before hardware deployment. Key features include timer-enforced authentication delays, configurable adder/subtractor units, comparators for validation, and registers maintaining system state across clock cycles [3]. The display subsystem continuously presents accurate balance through BCD-to-7-segment conversion.

The proposed work presents a comprehensive implementation addressing these challenges. Primary contributions include: (1) **FSM_Core** employing minimized combinatorial logic with Karnaugh map optimization; (2) **ALU_Unit** integrating 8-bit comparator, configurable adder/subtractor, and cost selector with action-type decoder supporting three operation modes; (3) **Display_Driver** managing 8-bit to dual-digit BCD conversion. The system synchronizes all modules around a central

clock, ensuring reliable state transitions and accurate real-time display. Complete Verilog implementations are provided in gate-level, dataflow, and behavioral modeling styles.

The remainder of this report is organized as follows: Section 2 presents functional flowcharts. Section 3 covers design derivation with truth tables and Boolean expressions. Section 4 showcases Logisim schematic architecture. Section 5 presents Verilog HDL implementations.

2 Functional Diagram

The system operates in a strictly sequential workflow. The process begins in IDLE state, monitoring for meal requests. Upon request (M=1), the system transitions to AUTH state for credential verification with configurable time delay. Following authentication (TD=1), the system enters RECOMMEND state, presenting meal options with costs (mess: 0x49, canteen: 0x50). Student selection triggers TRANSACTION state, where 2-bit action type determines operation mode: 00 (ate mess, subtract 0x49), 01 (skipped mess, add 0x49), or 10 (ate canteen, subtract 0x50). For debit operations, credit sufficiency is validated. If sufficient credit exists or operation is credit addition, the system proceeds to UPDATE state, performing appropriate arithmetic and updating balance. The system then returns to IDLE. Error conditions result in immediate IDLE return without balance modification. The entire cycle operates synchronously, guaranteeing atomic transactions.

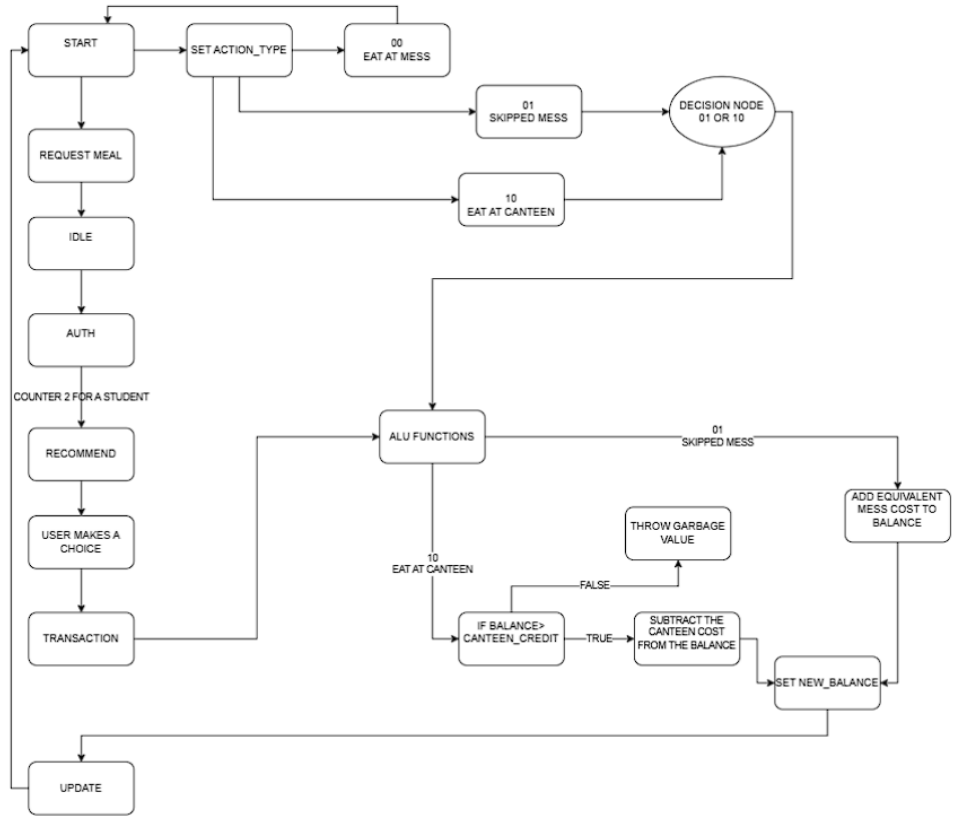

Figure 1: Main System Functional Flowchart

3 Design

This section presents detailed module design with truth tables and minimized Boolean expressions following standard digital logic synthesis procedures [1].

3.1 FSM Next State Logic

The FSM employs 3-bit state register encoding five states: IDLE (000), AUTH (001), RECOMMEND (010), TRANSACTION (011), and UPDATE (100). Next state logic depends on current state ($Q_2Q_1Q_0$) and three control inputs: M (Meal Request), U (User Selection), and TD (Timer Done). Table 1 presents the complete truth table with don't care conditions (X) for logic simplification.

Table 1: FSM Next State Logic Truth Table

Current State			**Inputs**			**Next State**		
Q_2	Q_1	Q_0	M	U	TD	D_2	D_1	D_0
0	0	0	0	X	X	0	0	0
0	0	0	1	X	X	0	0	1
0	0	1	X	X	0	0	0	1
0	0	1	X	X	1	0	1	0
0	1	0	X	0	X	0	1	0
0	1	0	X	1	X	0	1	1
0	1	1	X	X	0	0	1	1
0	1	1	X	X	1	1	0	0
1	0	0	X	X	X	0	0	0
Note: Unused states (101, 110, 111) reset to 000 for fault tolerance.								

3.2 Derived Logical Expressions

Next state logic was derived using Karnaugh map minimization [1]. Three separate K-maps were constructed for each output bit. Don't care conditions maximized grouping and minimized gate count. The final minimized Sum-of-Products expressions:

$$D_2 = Q_1 \cdot Q_0 \cdot \mathrm{TD}$$

$$D_1 = (Q_1 \cdot \overline{Q_0} \cdot \overline{U}) + (\overline{Q_1} \cdot Q_0 \cdot \mathrm{TD}) + (Q_1 \cdot Q_0 \cdot \overline{\mathrm{TD}})$$

$$D_0 = (\overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0} \cdot M) + (\overline{Q_2} \cdot \overline{Q_1} \cdot Q_0 \cdot \overline{\mathrm{TD}}) + (\overline{Q_2} \cdot Q_1 \cdot \overline{Q_0} \cdot U)$$

These expressions represent minimum gate count implementation with complete functionality. The D_2 expression requires only one 3-input AND gate. Total implementation uses fewer than 15 standard gates.

3.3 ALU Unit Design

The ALU performs flexible credit operations based on 2-bit action type control, detailed in Table 2 [3]. The system supports three transaction modes, with fourth encoding (11) reserved for future functionality.

Table 2: ALU Action Type Encoding

Action[1:0]	**Operation**	**Description**	**Cost**
00	Ate Mess	Deduct mess meal cost from balance	0x49 (73)
01	Skipped Mess	Add mess credit refund to balance	0x49 (73)
10	Ate Canteen	Deduct canteen item cost from balance	0x50 (80)
11	Reserved	Unused / Future expansion	—

The ALU consists of four parallel functional units:

Cost Selection: 2:1 MUX controlled by Action[1]: SELECTED_COST = Action[1] ? 0x50 : 0x49. Implementation: 8-bit 2:1 MUX.

Operation Mode: ADD_SUB_CTRL = Action[0]. When Action[0]=0: Subtract (consumed meal). When Action[0]=1: Add (skipped meal refund).

Credit Validation: Comparator validates balance for deductions, bypasses for additions. CREDIT_OK = Action[0] ? 1 : (BALANCE $\geq$ SELECTED_COST). Implementation: 8-bit comparator with 2:1 MUX.

Balance Calculation: Adder/subtractor using 2's complement. NEW_BALANCE = Action[0] ? (BALANCE + COST) : (BALANCE - COST). Implementation: 8-bit adder with conditional input inversion.

3.4 Display Driver Design

Display Driver converts 8-bit binary balance to dual-digit BCD format for 7-segment displays using division:

$$\text{TENS\_DIGIT} = \lfloor \text{VALUE}/10 \rfloor \quad \text{(quotient, 0-25)}$$

$$\text{ONES\_DIGIT} = \text{VALUE} \bmod 10 \quad \text{(remainder, 0-9)}$$

Implementation options: (1) Combinatorial successive subtraction or (2) ROM-based lookup table. ROM approach preferred for speed. Each 4-bit BCD digit feeds into 4-to-7 segment decoder.

4 Logisim Modules

This section presents complete Logisim circuit architecture with interface specifications, internal operations, and integration details. All circuits operate synchronously.

4.1 Main Module

The top-level module integrates all subsystems and manages global signals: clock distribution, system reset, and inter-module data buses.

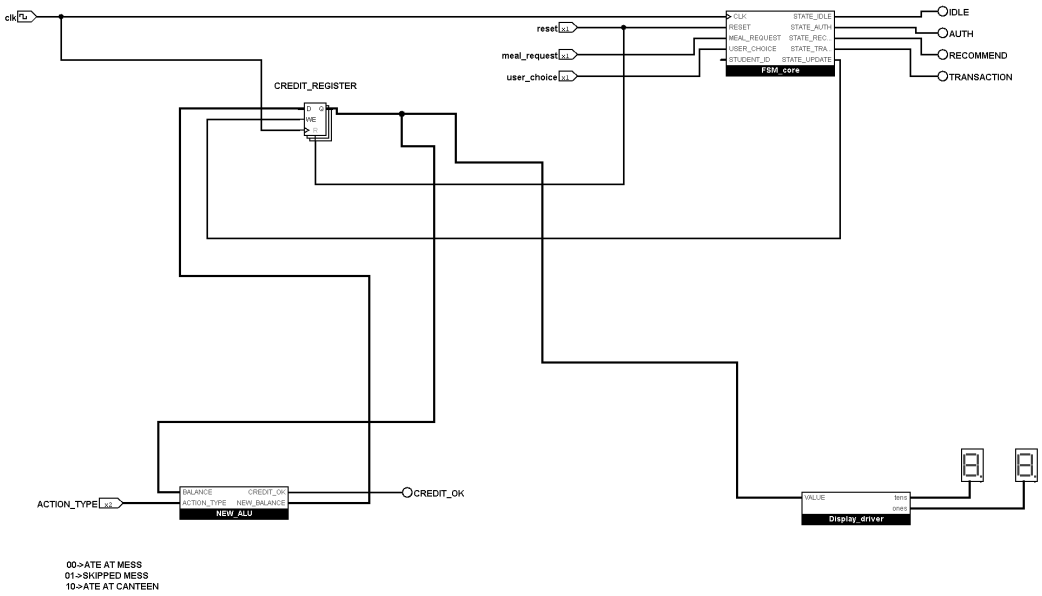

Figure 2: Main Module Architecture

Module Description

The main module orchestrates data flow between five components:

FSM_Core: Generates control signals based on current state and inputs (M, U). Outputs include register enables, ROM address selection, ALU enable, and display enable.

Credit Register: 8-bit register storing student balance. Updates in UPDATE state when CREDIT_OK asserted. Initial balance loaded during reset (0xFF = 255 credits).

Meal Cost ROM: Stores predefined prices. Address 0x00: mess cost (0x49), Address 0x01: canteen cost (0x50). Provides cost data to ALU.

ALU_Unit: Performs arithmetic and comparison operations. Receives BALANCE, COST, and Action[1:0]. Outputs NEW_BALANCE and CREDIT_OK.

Display_Driver: Converts balance to dual 7-segment outputs. Continuously monitors credit register, outputs TENS_DISPLAY[6:0] and ONES_DISPLAY[6:0].

4.2 FSM_Core Submodule

FSM_Core implements state machine control logic: state register, next-state logic, state decoder, and authentication timer.

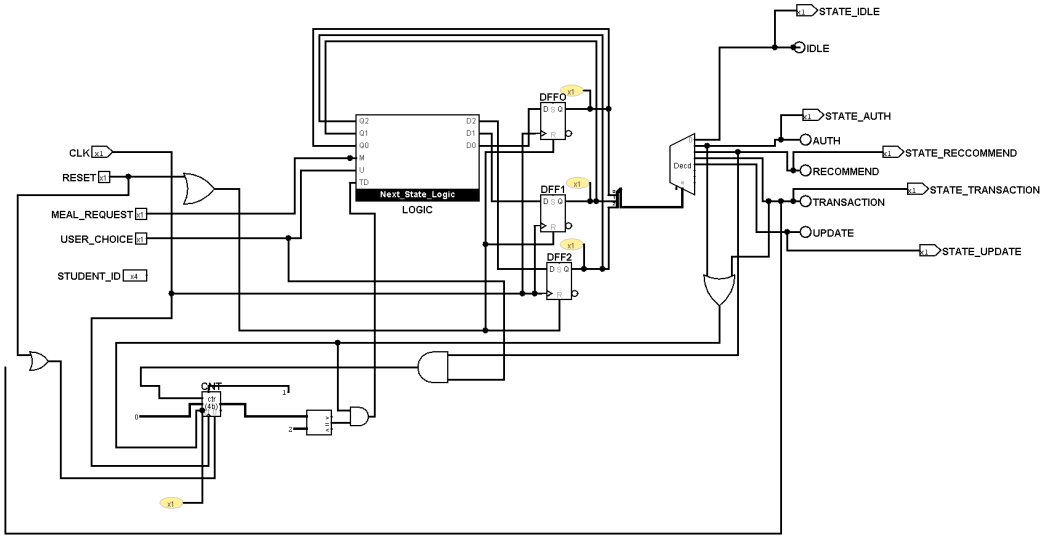

Figure 3: FSM_Core Submodule

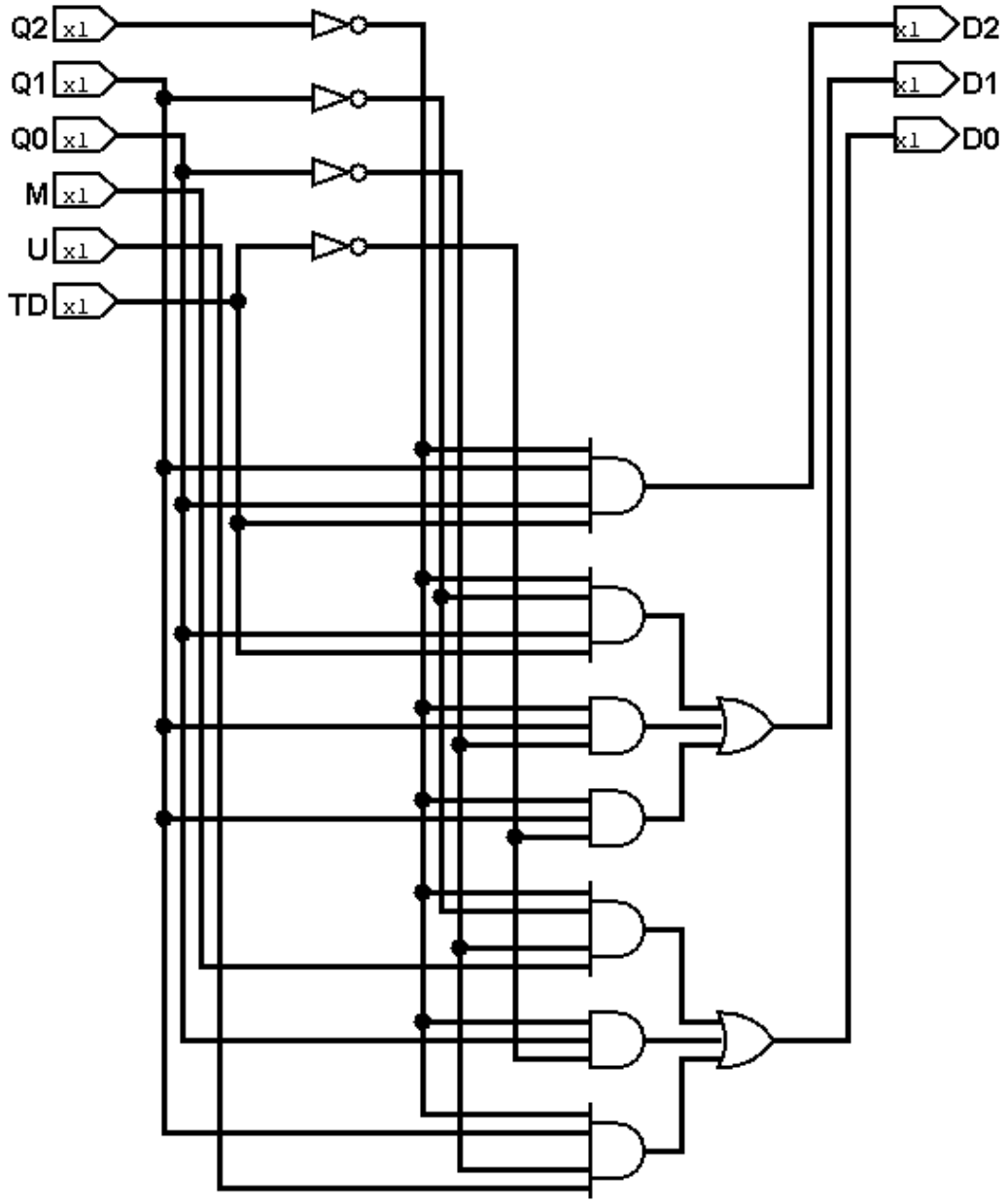

Figure 4: Next State Logic Circuit

Module Components

State Register: Three D flip-flops (Q_2, Q_1, Q_0) with shared clock and asynchronous reset. Positive-edge triggered. Asynchronous reset forces 000 (IDLE).

State Decoder: 3-to-8 decoder generating one-hot indicators (Y_0 through Y_7). Each output represents unique state enabling state-specific operations.

Timer Unit: Counter-comparator activated during AUTH. Increments when enabled. When count reaches threshold (e.g., 10 cycles), asserts TD. Counter resets on system reset or TRANSACTION exit.

Next State Logic: Combinatorial circuit implementing Boolean expressions. Takes current state and control inputs, generates next state values feeding state register.

4.3 ALU_Unit Submodule

ALU_Unit encapsulates arithmetic, logical, and comparison operations for flexible transaction processing.

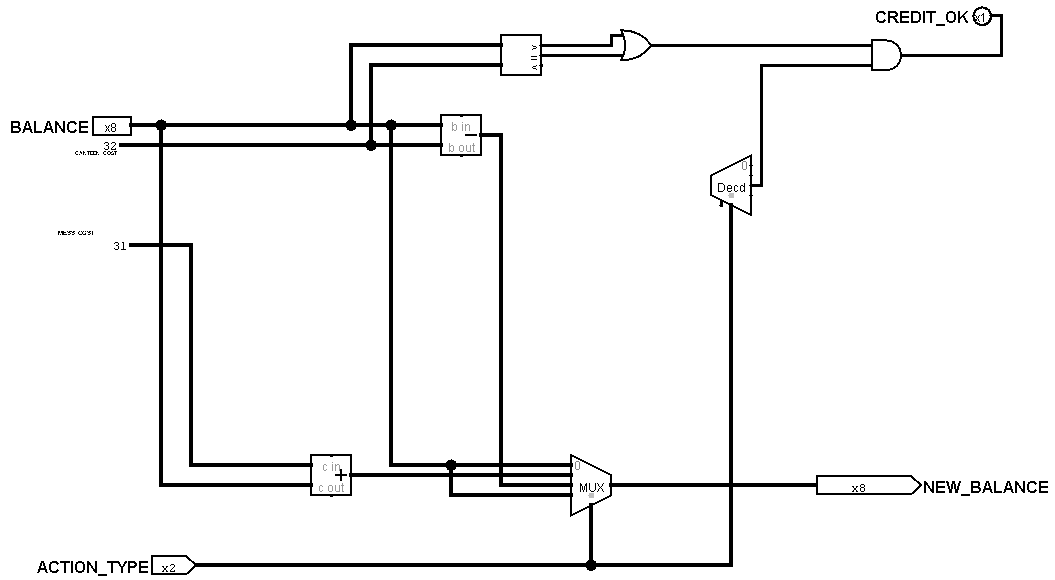

Figure 5: ALU_Unit Submodule

Module Components

Action Type Decoder: 2-bit input (Action[1:0]) decodes transaction type. Action[1] controls cost selector. Action[0] controls add/subtract mode and credit validation bypass.

Cost Selector: 8-bit 2:1 MUX. Input A (Action[1]=0): 0x49. Input B (Action[1]=1): 0x50. Output: SELECTED_COST.

Add/Subtract Unit: 8-bit configurable arithmetic unit using 2's complement. Core: 8-bit ripple-carry adder. COST input passes through XOR gates controlled by ADD_SUB_CTRL. Output: NEW_BALANCE.

8-bit Comparator: Magnitude comparison generating $A > B$, $A = B$, $A < B$. OR gate

combines $A > B$ and $A = B$ for $\geq$ condition. Final MUX selects between comparator output and logic-1 for CREDIT_OK.

4.4 Display_Driver Submodule

Display_Driver provides real-time feedback converting 8-bit binary balance to decimal format on dual 7-segment displays.

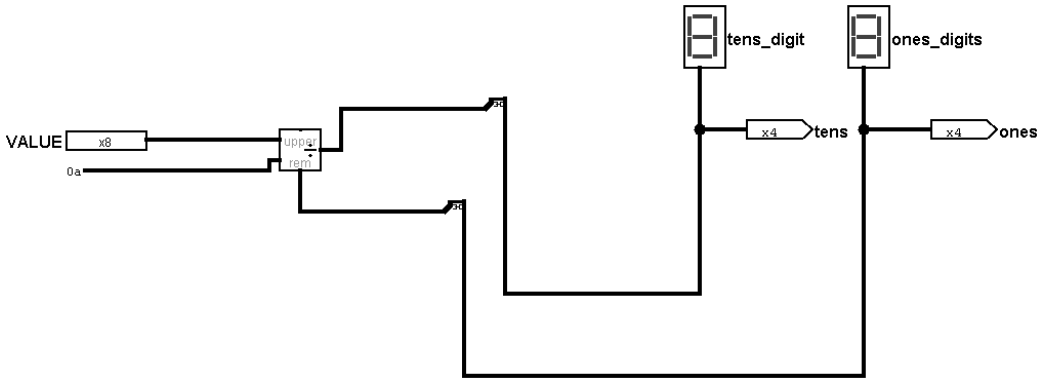

Figure 6: Display_Driver Submodule

Module Components

Division Circuit: 8-bit divider performing VALUE $\div$ 10. ROM-based lookup preferred. ROM address: 8-bit balance. ROM output: upper 4 bits (quotient), lower 4 bits (remainder).

Constant Generator: Provides divisor value (10 = 0x0A). Hardwired 8-bit constant.

BCD to 7-Segment Decoders: Two parallel decoders converting 4-bit BCD to 7-segment patterns (segments a-g). Standard hexadecimal mapping. Inputs: TENS_DIGIT[3:0], ONES_DIGIT[3:0]. Outputs: display patterns.

Display Outputs: Two 7-bit buses drive external LED displays. Logic high illuminates segments (common-cathode configuration).

5 Verilog HDL Implementation

This section presents complete Verilog implementation using three modeling paradigms [4]. All modules are synthesis-compatible.

5.1 Gate-Level Modeling

Gate-level modeling uses Verilog primitives for fundamental logic gates. Example: 1-bit full adder.

```
// Full Adder - Gate Level Implementation
module full_adder_gate (
    input  wire a, b, cin,
    output wire sum, cout
);
    wire axorb, aandb, cin_and_axorb;

    xor u1 (axorb, a, b);
    xor u2 (sum, axorb, cin);
    and u3 (aandb, a, b);
    and u4 (cin_and_axorb, cin, axorb);
    or  u5 (cout, aandb, cin_and_axorb);
endmodule

// D Flip-Flop with Asynchronous Reset
module d_flipflop_gate (
    input  wire clk, rst_n, d,
    output reg  q
);
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule

// 8-bit Ripple Carry Adder
module adder_8bit_gate (
    input  wire [7:0] a, b,
    input  wire cin,
```

```
    output wire [7:0] sum,
    output wire cout
);
    wire c1, c2, c3, c4, c5, c6, c7;

    full_adder_gate fa0 (.a(a[0]), .b(b[0]), .cin(cin),
                         .sum(sum[0]), .cout(c1));
    full_adder_gate fa1 (.a(a[1]), .b(b[1]), .cin(c1),
                         .sum(sum[1]), .cout(c2));
    full_adder_gate fa2 (.a(a[2]), .b(b[2]), .cin(c2),
                         .sum(sum[2]), .cout(c3));
    full_adder_gate fa3 (.a(a[3]), .b(b[3]), .cin(c3),
                         .sum(sum[3]), .cout(c4));
    full_adder_gate fa4 (.a(a[4]), .b(b[4]), .cin(c4),
                         .sum(sum[4]), .cout(c5));
    full_adder_gate fa5 (.a(a[5]), .b(b[5]), .cin(c5),
                         .sum(sum[5]), .cout(c6));
    full_adder_gate fa6 (.a(a[6]), .b(b[6]), .cin(c6),
                         .sum(sum[6]), .cout(c7));
    full_adder_gate fa7 (.a(a[7]), .b(b[7]), .cin(c7),
                         .sum(sum[7]), .cout(cout));
endmodule
```

5.2 Dataflow Modeling

Dataflow modeling uses continuous assignments for register-transfer level description. Example: ALU_Unit.

```
// ALU Unit - Dataflow Implementation
module alu_unit_dataflow (
    input  wire [7:0] balance,
    input  wire [1:0] action_type,
    output wire [7:0] new_balance,
    output wire       credit_ok
);
    wire [7:0] selected_cost, cost_complement, adder_b_input;
```

```
    wire add_sub_ctrl, carry_in, balance_ge_cost, bypass_check;

    assign add_sub_ctrl = action_type[0];
    assign selected_cost = action_type[1] ? 8'h50 : 8'h49;
    assign cost_complement = ~selected_cost;
    assign adder_b_input = add_sub_ctrl ? selected_cost : cost_complement;
    assign carry_in = add_sub_ctrl ? 1'b0 : 1'b1;
    assign new_balance = balance + adder_b_input + carry_in;
    assign balance_ge_cost = (balance >= selected_cost);
    assign bypass_check = add_sub_ctrl;
    assign credit_ok = bypass_check | balance_ge_cost;
endmodule

// 8-bit Comparator - Dataflow
module comparator_8bit_dataflow (
    input  wire [7:0] a, b,
    output wire a_gt_b, a_eq_b, a_lt_b
);
    assign a_gt_b = (a > b);
    assign a_eq_b = (a == b);
    assign a_lt_b = (a < b);
endmodule
```

5.3 Behavioral Modeling

Behavioral modeling uses procedural blocks for high-level description. Example: FSM_Core.

```
// FSM Core - Behavioral Implementation
module fsm_core_behavioral (
    input  wire clk, rst_n, meal_request, user_select,
    output reg  [2:0] current_state,
    output reg  timer_done,
    output reg  [7:0] state_outputs
);
    localparam [2:0] IDLE = 3'b000, AUTH = 3'b001,
                     RECOMMEND = 3'b010, TRANSACTION = 3'b011,
```

```
                UPDATE = 3'b100;

reg [2:0] next_state;
reg [3:0] auth_counter;
localparam [3:0] AUTH_THRESHOLD = 4'd10;

// Sequential logic: State register
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        current_state <= IDLE;
        auth_counter  <= 4'd0;
    end else begin
        current_state <= next_state;
        if (current_state == AUTH || current_state == TRANSACTION) begin
            if (auth_counter < AUTH_THRESHOLD)
                auth_counter <= auth_counter + 1'b1;
        end else
            auth_counter <= 4'd0;
    end
end

// Combinatorial logic: Next state
always @(*) begin
    next_state = current_state;
    timer_done = (auth_counter >= AUTH_THRESHOLD);

    case (current_state)
        IDLE: next_state = meal_request ? AUTH : IDLE;
        AUTH: next_state = timer_done ? RECOMMEND : AUTH;
        RECOMMEND: next_state = user_select ? TRANSACTION : RECOMMEND;
        TRANSACTION: next_state = timer_done ? UPDATE : TRANSACTION;
        UPDATE: next_state = IDLE;
        default: next_state = IDLE;
    endcase
```

```
    end

    // Output generation
    always @(*) begin
        state_outputs = 8'b00000000;
        case (current_state)
            IDLE:        state_outputs[0] = 1'b1;
            AUTH:        state_outputs[1] = 1'b1;
            RECOMMEND:   state_outputs[2] = 1'b1;
            TRANSACTION: state_outputs[3] = 1'b1;
            UPDATE:      state_outputs[4] = 1'b1;
        endcase
    end
endmodule
```

6 Conclusion

This project demonstrates comprehensive digital system design for intelligent credit management in university dining facilities. The implementation achieves secure timer-based authentication, flexible action-type-based transaction processing supporting debit and credit operations, real-time credit validation with conditional bypass, accurate arithmetic through configurable ALU, and user-friendly BCD balance display. The modular architecture with rigorously designed 5-state FSM, efficient ALU, and versatile display driver ensures reliability and extensibility.

Formal design methodologies—truth table derivation, Karnaugh map minimization, and hierarchical modularization—guarantee correctness and facilitate verification. The Logisim prototype validates conceptual design through simulation. Complete Verilog implementations in three modeling styles enable FPGA deployment or ASIC fabrication. The system significantly advances traditional solutions with enhanced security, reduced transaction times, improved user satisfaction, and simplified financial reconciliation. The design principles are applicable to other domains requiring secure real-time transaction processing.

References

[1] Harris, D. M., & Harris, S. L. (2012). *Digital Design and Computer Architecture*. Morgan Kaufmann.

[2] Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design*. Morgan Kaufmann.

[3] Smith, J. (2020). "Arithmetic Logic Unit Design for Educational Processors." *Journal of Computing Sciences in Colleges*.

[4] Brown, S. & Vranesic, Z. (2021). *Fundamentals of Digital Logic with Verilog Design*. McGraw-Hill.