Final Lab Report: Coffee Machine UI, Recipe Lookup, Production Engine, and LCD Controller Integration

Alexander Nicolazzo
Partner(s): *Alex Nicoazzo, Carlo Cortez*

December 5, 2025

1 Introduction

The purpose of this final lab was to integrate the complete coffee machine system into a single, demonstrable design that supports: (1) user-interface (UI) selection of drink parameters, (2) recipe-table lookup and capture (commit) of the looked-up recipe into a production (brew) engine, (3) simulated actuation using board lights for motors/grinders/water/heat, and (4) LCD status messaging for selection, brew progress, warnings, and errors.

Relative to the earlier LCD-focused lab work, this final lab expands the system into a full end-to-end coffee production controller. The system uses a recipe table (`cmach_recipes.svh` via `cmach_recp`) where the time fields and product equivalence must be maintained, an interactive selection interface (flavor/type/size + start), a timed brew engine, and LCD rendering that includes both normal operation (selection + progress) and error/warning cycling.

2 System Overview and Architecture

Figure 1 summarizes the final system. A top-level module (`lcdIp_Top`) connects the physical inputs (buttons, sensors) to:

- the recipe provider (`cmach_recp`) and recipe table,
- the coffee production engine (`coffeeSystem`),
- the LCD write engine (`lcdIp`),
- discrete output enables used as indicator lights to simulate motors/valves/heater.

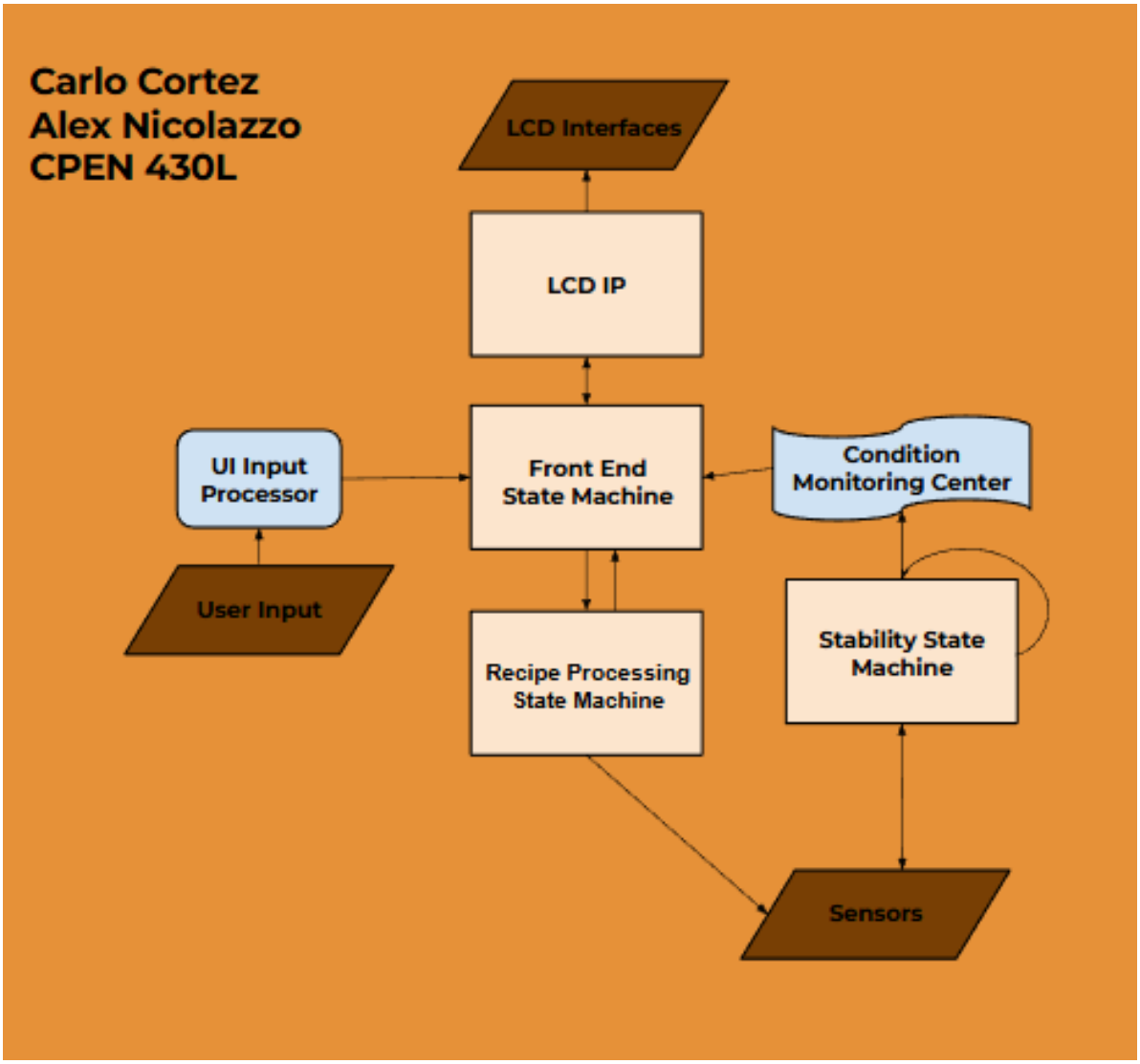

Figure 1: Final system architecture: UI selection → recipe lookup → recipe commit → brew engine phases and LCD reporting.

3 Recipe Table and Lookup Integration

3.1 Recipe Format

Recipes are provided through `cmach_recipes.svh` and delivered to the system using `cmach_recp`. Each recipe entry is a packed record (`coffee_recipe_t`) containing timed fields and option bits. In the production engine, the latched recipe is unpacked positionally into:

- `r_load_filter` (filter/paper load phase),
- `r_grinder_time` (seconds),
- `r_cocoa_time` (seconds),

- `r_pour_time` (seconds),
- `r_hot_water_time` (seconds),
- `r_add_creamer` (creamer enable during brew),

with an additional packed bit present in the record field ordering (captured in code as `_unused_HP`). The recipe times and product behavior were preserved to match the required equivalence constraints for this lab.

3.2 Recipe Indexing

The recipe lookup uses an index computed from the current UI selection:

$$\texttt{rIndex} = (\texttt{dType} \times 3) + \texttt{dSize}$$

This maps the Cartesian product of `type` and `size` into a contiguous table range `0..14`, matching the array bounds used in the top-level integration.

UI Field	Signal	Coding Used in Design
Flavor	`cur_flavor`	0 = Flavor 1, 1 = Flavor 2
Type	`cur_type[2:0]`	0=Mocha, 1=Latte, 2=Espresso, 3=Americano, 4=Drip
Size	`cur_size[1:0]`	0=10oz, 1=16oz, 2=20oz

Table 1: UI selection fields displayed on the LCD (normal screen) and used for recipe lookup.

3.3 Recipe Commit to Production Engine

A key requirement for the final lab is that the lookup result is *captured and committed* to the production engine. This is implemented by latching:

- the selection (`flavor_run`, `dType_run`, `dSize_run`),
- the looked-up recipe record (`recipe_lat`),

on a successful `START` event (no blocking ingredient/system errors at the time of start). After commit, the brew engine runs using `recipe_lat` even if the UI selection changes afterward.

4 Input Conditioning: Synchronization and Debounce

4.1 Metastability Protection (Synchronizer)

Asynchronous UI inputs (switches/buttons) are synchronized to the system clock using a two-flop synchronizer. The design uses the following module:

```
module sync(
    input  swIn,
    input  clk,
    output syncSignal
);
reg ff1;
reg ff2;
always @(posedge clk) begin
    ff1 <= swIn;
    ff2 <= ff1;
end
assign syncSignal = ff2;
endmodule
```

This ensures stable sampling on clock edges and prevents metastability from propagating into state logic.

4.2 Debounce for Mechanical Buttons

Mechanical buttons bounce and can produce multiple transitions per press. A debounce module is included for the UI buttons and placed in the button input path so that each physical press produces a clean digital action. The debounced outputs are then used for edge-detection in `coffeeSystem` to guarantee one selection increment per press.

5 Coffee Production Engine (`coffeeSystem`)

5.1 Top-Level State Machine

The production engine is organized into three primary states:

- **SELECT** (`S_SELECT`): user cycles flavor/type/size; LCD shows selection; system checks allow start gating.
- **WAIT** (`S_WAIT`): after a successful start, the engine waits for temperature readiness (`W_TEMP`).
- **BREW** (`S_BREW`): timed phase execution using the latched recipe.

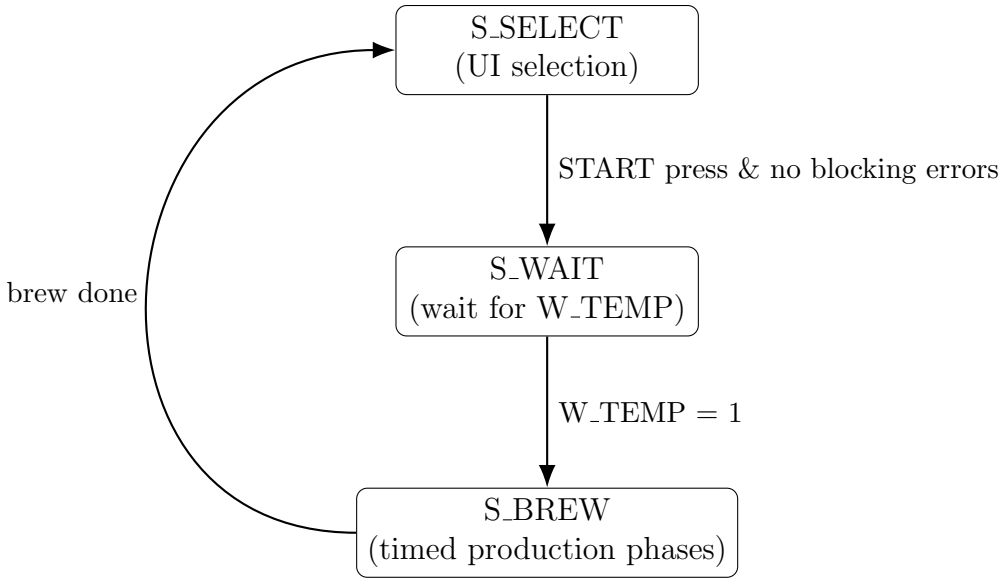

Figure 2: Coffee production engine top-level state machine.

5.2 Phase Machine and Timing

Within `S_BREW`, the engine runs a phase machine:

`PH_PAPER` → `PH_GRIND` → `PH_COCOA` → `PH_POUR` → `PH_WATER` → `PH_DONE`

Each phase duration is defined by the latched recipe fields. Zero-duration phases are skipped via `first_nonzero_phase()`, allowing the recipe table to define which operations are included for each drink.

5.3 Progress Reporting (0..16)

The system computes:

- `total_sec` = sum of recipe seconds (with a minimum of 1),
- `elapsed_sec` incremented once per second,
- `brew_progress16` = scaled progress from 0 to 16 via integer division.

This progress value is used to render a 12-character progress bar on the LCD during brewing.

6 System Errors, Ingredient Checks, and LCD Messaging

6.1 System Error Mask (Continuous)

System-level errors are continuously evaluated and force the controller back to `SELECT` while stopping production timing. The system error bits include:

- paper not installed / paper empty,
- pressure error / high pressure warning condition,
- status/hardware fault input.

6.2 Ingredient Blocking Errors (Latched on Start)

Ingredient availability is checked against the *live recipe* at the moment `START` is pressed. If the recipe requires grinder time, cocoa time, or creamer and the corresponding bin level indicates empty/not installed, the start is blocked and an ingredient error latch is held for a timed display window.

6.3 Warning Cycling

Warnings (e.g., "almost empty" levels or low pressure warning) are collected into a separate warning mask and displayed by alternating warning screens at a 1-second cadence when no blocking error is active.

6.4 LCD Message Selection

The LCD logic supports:

- **Normal screen** (selection + state/progress),
- **Warnings** (cycled),
- **Errors** (cycled),
- **Enjoy screen** displayed briefly after brew completion.

Msg Sel	Displayed Message (2 lines)	Category
0	Normal UI + progress (generated dynamically)	Normal
1–7	Ingredient/paper "almost empty" warnings	Warning
8–11	Pressure/status-related messages	Error/Warning
12–15	Blocking ingredient errors (no coffee bin, no cocoa, no creamer)	Error

Table 2: LCD message selection groups used by the final system. Exact strings are defined in `lcdIp_Top`.

7 Output Lights (Simulated Actuation)

The design uses discrete output enables as indicator lights to simulate actuators:

- HEAT_EN indicates heater active (waiting for temperature readiness),
- GRINDER_0_EN / GRINDER_1_EN indicate grinder motor operation (selected by flavor),
- COCOA_EN indicates cocoa dispensing,
- PAPER_EN indicates paper/filter handling,
- POUROVER_EN indicates low-pressure pour operation (used by recipes for drip-style pours),
- WATER_EN indicates the hot/high-pressure water stage when defined by the recipe,
- CREAMER_EN indicates creamer enabling during brewing when required.

Phase	Enabled Light(s)	Meaning
PH_PAPER	PAPER_EN	Filter/paper handling stage (if enabled by recipe)
PH_GRIND	GRINDER_0_EN/GRINDER_1_EN	Grinder motor active (depends on selected flavor)
PH_COCOA	COCOA_EN	Cocoa dispensing stage (if enabled by recipe)
PH_POUR	POUROVER_EN	Low-pressure pour stage (commonly used for drip recipes)
PH_WATER	WATER_EN	Hot/high-pressure water stage (as defined by recipe)
WAIT (pre-brew)	HEAT_EN	Heating until W_TEMP asserts

Table 3: Light outputs used to simulate operations during the production engine timeline.

8 LCD IP (lcdIp) and Top Integration (lcdIp_Top)

8.1 LCD Write Engine

The LCD engine is an 8-bit HD44780-compatible writer that supports instruction/data writes using:

- userOp (00=instruction, 01=data),
- send handshake,
- busy status output.

It implements timing guards (setup time, E pulse width, command delays, and extended clear delay).

8.2 Top-Level LCD Formatting

The top-level LCD logic:

- initializes the LCD using the required power-up command sequence,
- writes both lines (16 chars each),
- selects between normal/warn/error/enjoy messages,
- detects changes (selection, progress, state transitions) and triggers a clear+rewrite update.

8.3 Post-Brew "Please Enjoy" Screen

After brew completion (transition from BREW to SELECT), the top-level asserts a timed two-second override that displays:

- Line 1: "Please Enjoy!"
- Line 2: "Thank you"

Any new user input cancels the enjoy screen immediately.

9 Results and Demonstration

The final integrated design demonstrated the following required behaviors:

- UI selection of flavor/type/size with debounced single-step actions.
- Recipe lookup from the table using `rIndex = type*3 + size`.
- Recipe commit on successful `START`: latched recipe and selection remain stable throughout brewing.
- Brew sequence with phase skipping for zero-duration steps (recipe-defined behavior).
- Indicator lights correctly track heater, grinders, paper, cocoa, pour-over, and water stages.
- LCD normal screen displays selection and progress; errors/warnings cycle at 1 Hz; enjoy screen appears after completion.

Figure 3: Representative simulation or observed behavior: selection → start gating → timed phases → completion and enjoy screen.

A video demonstration of the coffee production sequence was recorded and uploaded to the course submission site as required.

10 Conclusion

This final lab successfully integrated the coffee machine UI, recipe lookup and commit, timed production engine, LCD reporting, and simulated actuation via indicator lights. The design supports user-driven selection, deterministic recipe execution after commit, robust handling of system/ingredient faults, and clear user feedback through LCD messages and phase indicators. The completed system meets the final lab requirements and provides a complete end-to-end demonstration of the coffee machine controller.

Appendix: RTL Code Listings (PDF) and Supporting Figures

A. lcdIp_Top.sv

```
1  `timescale 1ns/1ps
2  `include "cmach_recipes.svh"
3
4  module lcdIp_Top (
5      input  wire        CLOCK_50,
6      input  wire        KEY,            // KEY0 (active-low) -> START
7
8      input  wire        BTN_FLAVOR,      // if these are also KEY buttons, they are active-low
9      input  wire        BTN_TYPE,
10     input  wire        BTN_SIZE,
11     input  wire        BTN_START,       // unused (KEY0 is start), kept for pin-compat
12
13     // 2-bit "level" inputs (same encoding as PAPER_LEVEL):
14     // 00=NOT INSTALLED (or hard empty), 01=EMPTY, 10=ALMOST EMPTY, 11=OK
15     input  wire [1:0]  PAPER_LEVEL,
16     input  wire [1:0]  BIN0_LEVEL,
17     input  wire [1:0]  BIN1_LEVEL,
18     input  wire [1:0]  ND_LEVEL,
19     input  wire [1:0]  CH_LEVEL,
20
21     input  wire [1:0]  W_PRESSURE,
22     input  wire        W_TEMP,
23     input  wire        STATUS,
24
25     output wire        HEAT_EN,
26     output wire        POUROVER_EN,
27     output wire        WATER_EN,
28     output wire        GRINDER_0_EN,
29     output wire        GRINDER_1_EN,
30     output wire        PAPER_EN,
31     output wire        COCOA_EN,
32     output wire        CREAMER_EN,
33
34     output wire [7:0]  LCD_DATA,
35     output wire        LCD_RS,
36     output wire        LCD_RW,
37     output wire        LCD_EN,
38     output wire        LCD_ON,
39     output wire        LCD_BLON
40 );
41
42     assign LCD_ON   = 1'b1;
43     assign LCD_BLON = 1'b1;
44
45     //============================================================
46     // Internal power-on reset (POR)
47     //============================================================
48     localparam [21:0] POR_CYCLES = 22'd1_000_000; // ~20ms @50MHz
49     reg [21:0] por_cnt = 22'd0;
50     wire rst = (por_cnt < POR_CYCLES);
```

```
51
52      always @(posedge CLOCK_50) begin
53          if (por_cnt < POR_CYCLES)
54              por_cnt <= por_cnt + 22'd1;
55      end
56
57      //=============================================================
58      // Buttons: treat board keys as active-low => invert to active-high
59      // KEY0 is START
60      //=============================================================
61      wire btn_flavor_lvl = ~BTN_FLAVOR;
62      wire btn_type_lvl   = ~BTN_TYPE;
63      wire btn_size_lvl   = ~BTN_SIZE;
64      wire btn_start_lvl  = ~KEY;         // KEY0 start (active-low on board)
65
66      //=============================================================
67      // Recipes
68      //=============================================================
69      logic [$bits(coffee_recipe_t)-1:0] recipes [0:14];
70      cmach_recp u_recipes (.recipes(recipes));
71
72      //=============================================================
73      // Coffee control system
74      //=============================================================
75      wire       cur_flavor;
76      wire [2:0] cur_type;
77      wire [1:0] cur_size;
78      wire [1:0] sys_state;
79
80      wire [15:0] coffee_err_mask;
81      wire [2:0]  brew_phase;
82      wire [4:0]  brew_progress16;
83
84      coffeeSystem #(
85          .CLK_HZ(50_000_000),
86          .SPEEDUP_DIV(1)
87      ) u_coffee (
88          .clk(CLOCK_50),
89          .rst(rst),
90
91          .btn_flavor(btn_flavor_lvl),
92          .btn_type  (btn_type_lvl),
93          .btn_size  (btn_size_lvl),
94          .btn_start (btn_start_lvl),
95
96          .PAPER_LEVEL(PAPER_LEVEL),
97          .BIN0_LEVEL (BIN0_LEVEL),
98          .BIN1_LEVEL (BIN1_LEVEL),
99          .ND_LEVEL   (ND_LEVEL),
100         .CH_LEVEL   (CH_LEVEL),
```

```
101
102         .W_PRESSURE(W_PRESSURE),
103         .W_TEMP(W_TEMP),
104         .STATUS(STATUS),
105
106         .recipes(recipes),
107
108         .cur_flavor(cur_flavor),
109         .cur_type(cur_type),
110         .cur_size(cur_size),
111         .sys_state(sys_state),
112
113         .HEAT_EN(HEAT_EN),
114         .POUROVER_EN(POUROVER_EN),
115         .WATER_EN(WATER_EN),
116         .GRINDER_0_EN(GRINDER_0_EN),
117         .GRINDER_1_EN(GRINDER_1_EN),
118         .PAPER_EN(PAPER_EN),
119         .COCOA_EN(COCOA_EN),
120         .CREAMER_EN(CREAMER_EN),
121
122         .err_mask(coffee_err_mask),
123         .brew_phase(brew_phase),
124         .brew_progress16(brew_progress16)
125     );
126
127     //============================================================
128     // LCD IP interface
129     //============================================================
130     reg  [1:0] op;       // 2'b00 = INSTR (RS=0), 2'b01 = DATA (RS=1)
131     reg        send;
132     reg  [7:0] din;
133     wire       busy, systemReady;
134
135     lcdIp u_lcd (
136         .clk          (CLOCK_50),
137         .userOp       (op),
138         .send         (send),
139         .reset        (rst),
140         .inputCommand (din),
141         .lcd_data     (LCD_DATA),
142         .lcd_rs       (LCD_RS),
143         .lcd_rw       (LCD_RW),
144         .lcd_e        (LCD_EN),
145         .busy         (busy),
146         .systemReady  (systemReady)
147     );
148
149     //============================================================
150     // PLEASE ENJOY screen (shows briefly after brew completes)
```

```
151     //============================================================
152     reg [7:0] enjoy_l1 [0:15] = '{"P","l","e","a","s","e"," ","E","n","j","o","y","!"," ","
    "," "};
153     reg [7:0] enjoy_l2 [0:15] = '{"T","h","a","n","k"," ","y","o","u"," "," "," "," "," ","
    "," "};
154 
155     // 1-second tick (also used for err/warn cycling)
156     localparam [31:0] DISP_TICKS = 32'd50_000_000;
157     reg [31:0] disp_cnt;
158     wire disp_tick = (disp_cnt == (DISP_TICKS-1));
159 
160     always @(posedge CLOCK_50) begin
161         if (rst) disp_cnt <= 32'd0;
162         else if (disp_tick) disp_cnt <= 32'd0;
163         else disp_cnt <= disp_cnt + 32'd1;
164     end
165 
166     reg [1:0] sys_state_d;
167     reg       enjoy_active;
168     reg [1:0] enjoy_secs_left;
169 
170     always @(posedge CLOCK_50) begin
171         if (rst) begin
172             sys_state_d     <= 2'd0;
173             enjoy_active    <= 1'b0;
174             enjoy_secs_left <= 2'd0;
175         end else begin
176             sys_state_d <= sys_state;
177 
178             // detect transition BREW(2) -> SELECT(0)
179             if ((sys_state_d == 2'd2) && (sys_state == 2'd0)) begin
180                 enjoy_active    <= 1'b1;
181                 enjoy_secs_left <= 2'd2; // show for 2 seconds
182             end else if (enjoy_active && disp_tick) begin
183                 if (enjoy_secs_left <= 2'd1) begin
184                     enjoy_active    <= 1'b0;
185                     enjoy_secs_left <= 2'd0;
186                 end else begin
187                     enjoy_secs_left <= enjoy_secs_left - 2'd1;
188                 end
189             end
190 
191             // any new selection input cancels enjoy screen immediately
192             if (enjoy_active && (btn_flavor_lvl || btn_type_lvl || btn_size_lvl ||
    btn_start_lvl))
193                 enjoy_active <= 1'b0;
194         end
195     end
196 
197     //============================================================
198     // ERROR/WARN cycling selection
```

```
199     //=============================================================
200     // Must match coffeeSystem.sv error bit numbers
201     localparam int E_PAPER_NOT_INST = 0;
202     localparam int E_PAPER_EMPTY    = 1;
203     localparam int E_NO_COFFEE0     = 2;
204     localparam int E_NO_COFFEE1     = 3;
205     localparam int E_NO_CREAMER     = 4;
206     localparam int E_NO_CHOC        = 5;
207     localparam int E_PRESS_ERR      = 6;
208     localparam int E_PRESS_HIGH     = 7;
209     localparam int E_STATUS_ERR     = 8;
210 
211     // warnings we want to cycle (not blocking)
212     localparam int W_FILTER_ALMOST  = 0;
213     localparam int W_LOW_PRESSURE   = 1;
214     localparam int W_BIN0_EMPTY     = 2;
215     localparam int W_BIN1_EMPTY     = 3;
216     localparam int W_ND_EMPTY       = 4;
217     localparam int W_CH_EMPTY       = 5;
218 
219     logic [15:0] warn_mask;
220 
221     always @(*) begin
222         warn_mask = 16'b0;
223 
224         if (PAPER_LEVEL == 2'b10) warn_mask[W_FILTER_ALMOST] = 1'b1; // almost empty
225         if (W_PRESSURE  == 2'b00) warn_mask[W_LOW_PRESSURE]  = 1'b1; // low pressure
226 
227         // ingredient "almost empty" warnings are ONLY when level==2'b10
228         if (BIN0_LEVEL == 2'b10) warn_mask[W_BIN0_EMPTY] = 1'b1;
229         if (BIN1_LEVEL == 2'b10) warn_mask[W_BIN1_EMPTY] = 1'b1;
230         if (ND_LEVEL   == 2'b10) warn_mask[W_ND_EMPTY]   = 1'b1;
231         if (CH_LEVEL   == 2'b10) warn_mask[W_CH_EMPTY]   = 1'b1;
232     end
233 
234     wire err_present  = |coffee_err_mask;
235     wire warn_present = |warn_mask;
236 
237     // --------- Quartus-safe priority encoder (NO LOOPS) ---------
238     function automatic [3:0] first_set16(input logic [15:0] m);
239         begin
240             if      (m[0])  first_set16 = 4'd0;
241             else if (m[1])  first_set16 = 4'd1;
242             else if (m[2])  first_set16 = 4'd2;
243             else if (m[3])  first_set16 = 4'd3;
244             else if (m[4])  first_set16 = 4'd4;
245             else if (m[5])  first_set16 = 4'd5;
246             else if (m[6])  first_set16 = 4'd6;
247             else if (m[7])  first_set16 = 4'd7;
248             else if (m[8])  first_set16 = 4'd8;
```

```
            else if (m[9])  first_set16 = 4'd9;
            else if (m[10]) first_set16 = 4'd10;
            else if (m[11]) first_set16 = 4'd11;
            else if (m[12]) first_set16 = 4'd12;
            else if (m[13]) first_set16 = 4'd13;
            else if (m[14]) first_set16 = 4'd14;
            else if (m[15]) first_set16 = 4'd15;
            else            first_set16 = 4'd0;
        end
    endfunction

    function automatic [3:0] next_set16(input logic [15:0] m, input logic [3:0] cur);
        logic [31:0] mm;
        logic [31:0] rot;
        logic [15:0] r16;
        logic [4:0]  sh;
        logic [3:0]  idx;
        begin
            if (m == 16'b0) begin
                next_set16 = cur;
            end else begin
                mm = {m, m};
                sh = {1'b0, cur} + 5'd1;
                rot = (mm >> sh);
                r16 = rot[15:0];

                if (r16 == 16'b0) begin
                    next_set16 = first_set16(m);
                end else begin
                    idx = first_set16(r16);
                    next_set16 = (cur + 4'd1 + idx) & 4'hF;
                end
            end
        end
    endfunction

    reg warn_show;
    reg [3:0] err_code_cur;
    reg [3:0] warn_code_cur;

    always @(posedge CLOCK_50) begin
        if (rst) begin
            warn_show     <= 1'b0;
            err_code_cur  <= 4'd0;
            warn_code_cur <= 4'd0;
        end else if (disp_tick) begin
            if (err_present) begin
                warn_show <= 1'b0;

                if (!coffee_err_mask[err_code_cur])
```

```
299                     err_code_cur <= first_set16(coffee_err_mask);
300                 else
301                     err_code_cur <= next_set16(coffee_err_mask, err_code_cur);
302 
303             end else if (warn_present) begin
304                 warn_show <= ~warn_show;
305 
306                 if (!warn_show) begin
307                     if (!warn_mask[warn_code_cur])
308                         warn_code_cur <= first_set16(warn_mask);
309                     else
310                         warn_code_cur <= next_set16(warn_mask, warn_code_cur);
311                 end
312 
313             end else begin
314                 warn_show <= 1'b0;
315             end
316         end
317     end
318 
319     function automatic [3:0] map_err_code_to_msg(input logic [3:0] code);
320         begin
321             case (code)
322                 E_PAPER_EMPTY:    map_err_code_to_msg = 4'd2;
323                 E_PAPER_NOT_INST: map_err_code_to_msg = 4'd3;
324                 E_PRESS_ERR:      map_err_code_to_msg = 4'd8;
325                 E_PRESS_HIGH:     map_err_code_to_msg = 4'd9;
326                 E_STATUS_ERR:     map_err_code_to_msg = 4'd11;
327 
328                 E_NO_COFFEE0:     map_err_code_to_msg = 4'd12;
329                 E_NO_COFFEE1:     map_err_code_to_msg = 4'd13;
330                 E_NO_CHOC:        map_err_code_to_msg = 4'd14;
331                 E_NO_CREAMER:     map_err_code_to_msg = 4'd15;
332 
333                 default:          map_err_code_to_msg = 4'd11;
334             endcase
335         end
336     endfunction
337 
338     function automatic [3:0] map_warn_code_to_msg(input logic [3:0] code);
339         begin
340             case (code)
341                 W_FILTER_ALMOST: map_warn_code_to_msg = 4'd1;
342                 W_LOW_PRESSURE:  map_warn_code_to_msg = 4'd10;
343 
344                 W_BIN0_EMPTY:    map_warn_code_to_msg = 4'd4;
345                 W_BIN1_EMPTY:    map_warn_code_to_msg = 4'd5;
346                 W_ND_EMPTY:      map_warn_code_to_msg = 4'd6;
347                 W_CH_EMPTY:      map_warn_code_to_msg = 4'd7;
348 
```

```
                default:         map_warn_code_to_msg = 4'd1;
            endcase
        end
    endfunction

    wire [3:0] msg_sel =
        (enjoy_active)              ? 4'd0 : // enjoy overrides inside get_byte()
        (err_present)               ? map_err_code_to_msg(err_code_cur) :
        (warn_present && warn_show) ? map_warn_code_to_msg(warn_code_cur) :
                                      4'd0;

    //============================================================
    // LCD message ROMs
    //============================================================
    reg [7:0] m1_l1 [0:15]  = '{"P","a","p","e","r"," ","F","i","l","t","e","r","
","A","l","m"};
    reg [7:0] m1_l2 [0:15]  = '{"s","t","r","E","m","p","t","y"," "," "," "," "," "," "," ","
"};
    reg [7:0] m2_l1 [0:15]  = '{"P","a","p","e","r"," ","F","i","l","t","e","r"," "," "," ","
"};
    reg [7:0] m2_l2 [0:15]  = '{"E","m","p","t","y"," "," "," "," "," "," "," "," "," "," ","
"};
    reg [7:0] m3_l1 [0:15]  = '{"P","a","p","e","r"," ","F","i","l","t","e","r","
","N","O","T"};
    reg [7:0] m3_l2 [0:15]  = '{"I","N","S","T","A","L","L","E","D"," "," "," "," "," "," ","
"};
    reg [7:0] m4_l1 [0:15]  = '{"C","o","f","f","e","e"," ","B","i","n"," ","0","
","A","l","m"};
    reg [7:0] m4_l2 [0:15]  = '{"s","t","r","E","m","p","t","y"," "," "," "," "," "," "," ","
"};
    reg [7:0] m5_l1 [0:15]  = '{"C","o","f","f","e","e"," ","B","i","n"," ","1","
","A","l","m"};
    reg [7:0] m5_l2 [0:15]  = '{"s","t","r","E","m","p","t","y"," "," "," "," "," "," "," ","
"};
    reg [7:0] m6_l1 [0:15]  = '{"N","o","n","-","D","a","i","r","y"," ","C","r","m","r","
","A"};
    reg [7:0] m6_l2 [0:15]  = '{"l","m","s","t","r","E","m","p","t","y"," "," "," "," "," ","
"};
    reg [7:0] m7_l1 [0:15]  = '{"C","h","o","c","o","l","a","t","e"," ","P","w","d","r","
","A"};
    reg [7:0] m7_l2 [0:15]  = '{"l","m","s","t","r","E","m","p","t","y"," "," "," "," "," ","
"};
    reg [7:0] m8_l1 [0:15]  = '{"W","a","t","e","r"," ","P","r","e","s","s","u","r","e","r","
"};
    reg [7:0] m8_l2 [0:15]  = '{"E","r","r","o","r","!"," "," "," "," "," "," "," "," "," ","
"};
    reg [7:0] m9_l1 [0:15]  = '{"H","i","g","h","r","W","a","t","e","r"," "," "," "," "," ","
"};
    reg [7:0] m9_l2 [0:15]  = '{"P","r","e","s","s","u","r","e","r"," "," "," "," "," "," ","
"};
    reg [7:0] m10_l1[0:15]  = '{"L","o","w","r","W","a","t","e","r"," "," "," "," "," "," ","
"};
```

```
382     reg [7:0] m10_l2[0:15]  = '{"P","r","e","s","s","u","r","e"," "," "," "," "," "," "," "," "
    "};
383     reg [7:0] m11_l1[0:15]  = '{"H","a","r","d","w","a","r","e"," ","S","t","a","t","u","s","
    "};
384     reg [7:0] m11_l2[0:15]  = '{"E","r","r","o","r"," ","-"," ","S","e","r","v","i","c","e","
    "};
385
386     reg [7:0] m12_l1[0:15]  = '{"N","o"," ","C","o","f","f","e","e"," ","B","i","n"," ","0","
    "};
387     reg [7:0] m12_l2[0:15]  = '{"C","a","n","n","o","t"," ","e","x","e","c","u","t","e"," ","
    "};
388
389     reg [7:0] m13_l1[0:15]  = '{"N","o"," ","C","o","f","f","e","e"," ","B","i","n"," ","1","
    "};
390     reg [7:0] m13_l2[0:15]  = '{"C","a","n","n","o","t"," ","e","x","e","c","u","t","e"," ","
    "};
391
392     reg [7:0] m14_l1[0:15]  = '{"N","o"," ","C","h","o","c","o","l","a","t","e"," "," "," ","
    "};
393     reg [7:0] m14_l2[0:15]  = '{"C","a","n","n","o","t"," ","e","x","e","c","u","t","e"," ","
    "};
394
395     reg [7:0] m15_l1[0:15]  = '{"N","o"," ","C","r","e","a","m","e","r"," "," "," "," "," ","
    "};
396     reg [7:0] m15_l2[0:15]  = '{"C","a","n","n","o","t"," ","e","x","e","c","u","t","e"," ","
    "};
397
398     //============================================================
399     // Normal 2-line message generator (selection + state/progress)
400     //============================================================
401     function automatic [39:0] drink5(input [2:0] t);
402         begin
403             case (t)
404                 3'd0: drink5 = {"M","O","C","H","A"};
405                 3'd1: drink5 = {"L","A","T","T","E"};
406                 3'd2: drink5 = {"E","S","P","R"," "};
407                 3'd3: drink5 = {"A","M","E","R"," "};
408                 3'd4: drink5 = {"D","R","I","P"," "};
409                 default: drink5 = {"U","N","K","N"," "};
410             endcase
411         end
412     endfunction
413
414     function automatic [31:0] size4(input [1:0] s);
415         begin
416             case (s)
417                 2'd0: size4 = {"1","0","o","z"};
418                 2'd1: size4 = {"1","6","o","z"};
419                 2'd2: size4 = {"2","0","o","z"};
420                 default: size4 = {"?","?","o","z"};
421             endcase
```

```
422         end
423     endfunction
424 
425     function automatic [127:0] mk_line1(input logic f, input logic [2:0] t, input logic [1:0]
    s);
426         logic [7:0] fch;
427         begin
428             fch = (f) ? "2" : "1";
429             mk_line1 = {"C", fch, " ", drink5(t), " ", size4(s), " ", " ", " "};
430         end
431     endfunction
432 
433     function automatic [31:0] phase4(input logic [2:0] ph);
434         begin
435             case (ph)
436                 3'd0: phase4 = {"P","A","P","R"};
437                 3'd1: phase4 = {"G","R","N","D"};
438                 3'd2: phase4 = {"C","O","C","O"};
439                 3'd3: phase4 = {"P","O","U","R"};
440                 3'd4: phase4 = {"W","A","T","R"};
441                 default: phase4 = {"D","O","N","E"};
442             endcase
443         end
444     endfunction
445 
446     function automatic [95:0] bar12(input logic [3:0] filled);
447         begin
448             bar12 = {
449                 (filled> 0 ? "#" : "-"),
450                 (filled> 1 ? "#" : "-"),
451                 (filled> 2 ? "#" : "-"),
452                 (filled> 3 ? "#" : "-"),
453                 (filled> 4 ? "#" : "-"),
454                 (filled> 5 ? "#" : "-"),
455                 (filled> 6 ? "#" : "-"),
456                 (filled> 7 ? "#" : "-"),
457                 (filled> 8 ? "#" : "-"),
458                 (filled> 9 ? "#" : "-"),
459                 (filled>10 ? "#" : "-"),
460                 (filled>11 ? "#" : "-")
461             };
462         end
463     endfunction
464 
465     function automatic [127:0] mk_line2(input logic [1:0] st, input logic [2:0] ph, input
    logic [4:0] prog16);
466         logic [3:0] filled12;
467         logic [7:0] tmp_fill;
468         begin
469             case (st)
```

```
                2'd0: mk_line2 = {"P","r","e","s","s"," ","S","T","A","R","T"," "," "," "," "," "};
                2'd1: mk_line2 = {"H","E","A","T","I","N","G",".",".",".","."," "," "," "," "," "};
                2'd2: begin
                    if (prog16 >= 5'd16) filled12 = 4'd12;
                    else begin
                        tmp_fill = (prog16 * 8'd12) / 8'd16;
                        filled12 = tmp_fill[3:0];
                    end
                    mk_line2 = { bar12(filled12), phase4(ph) };
                end
                default: mk_line2 = {"R","E","A","D","Y"," "," "," "," "," "," "," "," "," "," "," "," "};
            endcase
        end
    endfunction

    function automatic [7:0] byte16(input [127:0] s, input [4:0] k);
        begin
            byte16 = s[8*(15-k) +: 8];
        end
    endfunction

    function automatic [7:0] get_byte;
        input [3:0] sel;
        input       line; // 0=line1, 1=line2
        input [4:0] k;
        logic [127:0] L1, L2;
        begin
            // enjoy override
            if (enjoy_active) begin
                get_byte = line ? enjoy_l2[k] : enjoy_l1[k];
            end else if (sel == 4'd0) begin
                L1 = mk_line1(cur_flavor, cur_type, cur_size);
                L2 = mk_line2(sys_state, brew_phase, brew_progress16);
                get_byte = (line) ? byte16(L2, k) : byte16(L1, k);
            end else begin
                case (sel)
                    4'd1:  get_byte = line ? m1_l2[k]  : m1_l1[k];
                    4'd2:  get_byte = line ? m2_l2[k]  : m2_l1[k];
                    4'd3:  get_byte = line ? m3_l2[k]  : m3_l1[k];
                    4'd4:  get_byte = line ? m4_l2[k]  : m4_l1[k];
                    4'd5:  get_byte = line ? m5_l2[k]  : m5_l1[k];
                    4'd6:  get_byte = line ? m6_l2[k]  : m6_l1[k];
                    4'd7:  get_byte = line ? m7_l2[k]  : m7_l1[k];
                    4'd8:  get_byte = line ? m8_l2[k]  : m8_l1[k];
                    4'd9:  get_byte = line ? m9_l2[k]  : m9_l1[k];
                    4'd10: get_byte = line ? m10_l2[k] : m10_l1[k];
                    4'd11: get_byte = line ? m11_l2[k] : m11_l1[k];
                    4'd12: get_byte = line ? m12_l2[k] : m12_l1[k];
```

```
518                     4'd13: get_byte = line ? m13_l2[k] : m13_l1[k];
519                     4'd14: get_byte = line ? m14_l2[k] : m14_l1[k];
520                     4'd15: get_byte = line ? m15_l2[k] : m15_l1[k];
521                     default: get_byte = " ";
522                 endcase
523             end
524         end
525     endfunction
526 
527     //============================================================
528     // LCD init/write FSM
529     //============================================================
530     localparam [5:0]
531         S_PWRUP     = 6'd0,
532         S_WAIT1     = 6'd2,
533         S_WAIT2     = 6'd4,
534         S_WAIT3     = 6'd6,
535         S_WAIT_DISPOFF   = 6'd8,
536         S_WAIT_CLR        = 6'd10,
537         S_WAIT_ENTRY      = 6'd12,
538         S_WAIT_DISPON     = 6'd14,
539         S_WAITON          = 6'd16,
540         S_SET_L1    = 6'd17,  S_WAIT_L1 = 6'd18,
541         S_WRITE_L1  = 6'd19,  S_WAIT_WL1= 6'd20,
542         S_SET_L2    = 6'd21,  S_WAIT_L2 = 6'd22,
543         S_WRITE_L2  = 6'd23,  S_WAIT_WL2= 6'd24,
544         S_IDLE      = 6'd25,
545         S_CLR_SW    = 6'd26,  S_WAIT_ENTRY_SW= 6'd27;
546 
547     reg [5:0]  state = S_PWRUP;
548     reg [31:0] dly = 0;
549     reg [4:0]  idx = 0;
550 
551     localparam DLY_15MS   = 32'd750_000;
552     localparam DLY_5MS    = 32'd250_000;
553     localparam DLY_100US  = 32'd5_000;
554     localparam DLY_CMD    = 32'd5_000;
555     localparam DLY_CLEAR  = 32'd75_000;
556 
557     // Change detection
558     reg [3:0] prev_msg_sel;
559     reg       prev_flavor;
560     reg [2:0] prev_type;
561     reg [1:0] prev_size;
562     reg [1:0] prev_sys_state;
563     reg [2:0] prev_brew_phase;
564     reg [4:0] prev_brew_prog;
565 
566     reg       prev_enjoy_active;
567     wire      enjoy_changed = (enjoy_active != prev_enjoy_active);
```

```
568
569     wire msg_changed = (msg_sel != prev_msg_sel);
570
571     wire normal_changed = (msg_sel == 4'd0) &&
572                           ((prev_flavor     != cur_flavor) ||
573                            (prev_type       != cur_type)   ||
574                            (prev_size       != cur_size)   ||
575                            (prev_sys_state  != sys_state)  ||
576                            (prev_brew_phase != brew_phase) ||
577                            (prev_brew_prog  != brew_progress16));
578
579     reg change_detected;
580
581     always @(posedge CLOCK_50) begin
582         if (rst) begin
583             prev_msg_sel    <= 4'd0;
584             prev_flavor     <= 1'b0;
585             prev_type       <= 3'd0;
586             prev_size       <= 2'd0;
587             prev_sys_state  <= 2'd0;
588             prev_brew_phase <= 3'd0;
589             prev_brew_prog  <= 5'd0;
590             prev_enjoy_active <= 1'b0;
591             change_detected <= 1'b0;
592         end else begin
593             if (msg_changed || normal_changed || enjoy_changed)
594                 change_detected <= 1'b1;
595             else if (state == S_CLR_SW)
596                 change_detected <= 1'b0;
597
598             prev_msg_sel    <= msg_sel;
599             prev_flavor     <= cur_flavor;
600             prev_type       <= cur_type;
601             prev_size       <= cur_size;
602             prev_sys_state  <= sys_state;
603             prev_brew_phase <= brew_phase;
604             prev_brew_prog  <= brew_progress16;
605             prev_enjoy_active <= enjoy_active;
606         end
607     end
608
609     always @(posedge CLOCK_50) begin
610         if (rst) begin
611             state <= S_PWRUP;
612             dly   <= 0;
613             idx   <= 0;
614             op    <= 2'b00;
615             din   <= 8'h00;
616             send  <= 1'b0;
617         end else begin
```

```
618             send <= 1'b0;
619
620             case (state)
621                 S_PWRUP: begin
622                     if (dly < DLY_15MS) dly <= dly + 1;
623                     else begin
624                         din   <= 8'h30; op <= 2'b00; send <= 1'b1;
625                         dly   <= 0; state <= S_WAIT1;
626                     end
627                 end
628
629                 S_WAIT1: begin
630                     if (!busy && dly >= DLY_5MS) begin
631                         din   <= 8'h30; op <= 2'b00; send <= 1'b1;
632                         dly   <= 0; state <= S_WAIT2;
633                     end else dly <= dly + 1;
634                 end
635
636                 S_WAIT2: begin
637                     if (!busy && dly >= DLY_100US) begin
638                         din   <= 8'h30; op <= 2'b00; send <= 1'b1;
639                         dly   <= 0; state <= S_WAIT3;
640                     end else dly <= dly + 1;
641                 end
642
643                 S_WAIT3: begin
644                     if (!busy && dly >= DLY_100US) begin
645                         din   <= 8'h38; op <= 2'b00; send <= 1'b1;
646                         dly   <= 0; state <= S_WAIT_DISPOFF;
647                     end else dly <= dly + 1;
648                 end
649
650                 S_WAIT_DISPOFF: begin
651                     if (!busy && dly >= DLY_CMD) begin
652                         din   <= 8'h08; op <= 2'b00; send <= 1'b1;
653                         dly   <= 0; state <= S_WAIT_CLR;
654                     end else dly <= dly + 1;
655                 end
656
657                 S_WAIT_CLR: begin
658                     if (!busy && dly >= DLY_CMD) begin
659                         din   <= 8'h01; op <= 2'b00; send <= 1'b1;
660                         dly   <= 0; state <= S_WAIT_ENTRY;
661                     end else dly <= dly + 1;
662                 end
663
664                 S_WAIT_ENTRY: begin
665                     if (!busy && dly >= DLY_CLEAR) begin
666                         din   <= 8'h06; op <= 2'b00; send <= 1'b1;
667                         dly   <= 0; state <= S_WAIT_DISPON;
```

```
668                     end else dly <= dly + 1;
669                 end
670 
671                 S_WAIT_DISPON: begin
672                     if (!busy && dly >= DLY_CMD) begin
673                         din   <= 8'h0C; op <= 2'b00; send <= 1'b1;
674                         dly   <= 0; state <= S_WAITON;
675                     end else dly <= dly + 1;
676                 end
677 
678                 S_WAITON: begin
679                     if (!busy && dly >= DLY_CMD) begin
680                         state <= S_SET_L1;
681                     end else dly <= dly + 1;
682                 end
683 
684                 S_SET_L1: begin
685                     if (!busy) begin
686                         din <= 8'h80; op <= 2'b00; send <= 1'b1;
687                         dly <= 0; idx <= 0; state <= S_WAIT_L1;
688                     end
689                 end
690 
691                 S_WAIT_L1: begin
692                     if (!busy && dly >= DLY_CMD) state <= S_WRITE_L1;
693                     else dly <= dly + 1;
694                 end
695 
696                 S_WRITE_L1: begin
697                     if (!busy) begin
698                         din <= get_byte(msg_sel, 1'b0, idx);
699                         op  <= 2'b01; send <= 1'b1;
700                         dly <= 0; state <= S_WAIT_WL1;
701                     end
702                 end
703 
704                 S_WAIT_WL1: begin
705                     if (!busy && dly >= DLY_CMD) begin
706                         if (idx < 5'd15) begin
707                             idx <= idx + 5'd1; state <= S_WRITE_L1;
708                         end else begin
709                             idx <= 0; state <= S_SET_L2;
710                         end
711                     end else dly <= dly + 1;
712                 end
713 
714                 S_SET_L2: begin
715                     if (!busy) begin
716                         din <= 8'hC0; op <= 2'b00; send <= 1'b1;
717                         dly <= 0; state <= S_WAIT_L2;
```

```
                    end
                end

                S_WAIT_L2: begin
                    if (!busy && dly >= DLY_CMD) state <= S_WRITE_L2;
                    else dly <= dly + 1;
                end

                S_WRITE_L2: begin
                    if (!busy) begin
                        din <= get_byte(msg_sel, 1'b1, idx);
                        op  <= 2'b01; send <= 1'b1;
                        dly <= 0; state <= S_WAIT_WL2;
                    end
                end

                S_WAIT_WL2: begin
                    if (!busy && dly >= DLY_CMD) begin
                        if (idx < 5'd15) begin
                            idx <= idx + 5'd1; state <= S_WRITE_L2;
                        end else begin
                            idx <= 0; state <= S_IDLE;
                        end
                    end else dly <= dly + 1;
                end

                S_IDLE: begin
                    if (change_detected && !busy) state <= S_CLR_SW;
                end

                S_CLR_SW: begin
                    if (!busy) begin
                        din <= 8'h01; op <= 2'b00; send <= 1'b1;
                        dly <= 0; state <= S_WAIT_ENTRY_SW;
                    end
                end

                S_WAIT_ENTRY_SW: begin
                    if (!busy && dly >= DLY_CLEAR) state <= S_SET_L1;
                    else dly <= dly + 1;
                end

                default: state <= S_PWRUP;
            endcase
        end
    end

endmodule
```

B. debounce.sv

```
module debounce #(
    parameter int CLK_HZ = 50_000_000,
    parameter int DEBOUNCE_MS = 10
)(
    input  logic clk,
    input  logic rst,
    input  logic noisy,
    output logic clean
);

    localparam int COUNT_MAX = (CLK_HZ/1000)*DEBOUNCE_MS;

    logic [$clog2(COUNT_MAX):0] count;
    logic state;

    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= 1'b0;
            clean <= 1'b0;
            count <= '0;
        end else begin
            if (noisy != state) begin
                if (count == COUNT_MAX) begin
                    state <= noisy;
                    clean <= noisy;
                    count <= '0;
                end else begin
                    count <= count + 1;
                end
            end else begin
                count <= '0;
            end
        end
    end
endmodule
```

C. coffeeSystem.sv

src\coffeeSystem.sv

```
1  `timescale 1ns/1ps
2  `include "cmach_recipes.svh"
3
4  //============================================================
5  // Debounce Module Import
6  //============================================================
7  module coffeeSystem #(
8      parameter int CLK_HZ      = 50_000_000,
9      parameter int SPEEDUP_DIV = 1
10 )(
11     input  logic        clk,
12     input  logic        rst,
13
14     input  logic        btn_flavor,
15     input  logic        btn_type,
16     input  logic        btn_size,
17     input  logic        btn_start,
18
19     input  logic [1:0]  PAPER_LEVEL,
20     input  logic [1:0]  BIN0_LEVEL,
21     input  logic [1:0]  BIN1_LEVEL,
22     input  logic [1:0]  ND_LEVEL,
23     input  logic [1:0]  CH_LEVEL,
24
25     input  logic [1:0]  W_PRESSURE,
26     input  logic        W_TEMP,
27     input  logic        STATUS,
28
29     input  logic [$bits(coffee_recipe_t)-1:0] recipes [0:14],
30
31     output logic        cur_flavor,
32     output logic [2:0]  cur_type,
33     output logic [1:0]  cur_size,
34
35     output logic [1:0]  sys_state,
36
37     output logic        HEAT_EN,
38     output logic        POUROVER_EN,
39     output logic        WATER_EN,
40     output logic        GRINDER_0_EN,
41     output logic        GRINDER_1_EN,
42     output logic        PAPER_EN,
43
44     output logic        COCOA_EN,
45     output logic        CREAMER_EN,
46
47     output logic [15:0] err_mask,
48
```

```
49      output logic [2:0]  brew_phase,
50      output logic [4:0]  brew_progress16
51  );
52
53  //=============================================================
54  // Add Debounced Button Signals
55  //=============================================================
56  logic db_flavor, db_type, db_size, db_start;
57
58  debounce #(.CLK_HZ(CLK_HZ)) DB0 (.clk(clk), .rst(rst), .noisy(btn_flavor),
    .clean(db_flavor));
59  debounce #(.CLK_HZ(CLK_HZ)) DB1 (.clk(clk), .rst(rst), .noisy(btn_type),   .clean(db_type));
60  debounce #(.CLK_HZ(CLK_HZ)) DB2 (.clk(clk), .rst(rst), .noisy(btn_size),   .clean(db_size));
61  debounce #(.CLK_HZ(CLK_HZ)) DB3 (.clk(clk), .rst(rst), .noisy(btn_start),  .clean(db_start));
62
63  //=============================================================
64  // Error bit mapping
65  //=============================================================
66  localparam int E_PAPER_NOT_INST = 0;
67  localparam int E_PAPER_EMPTY    = 1;
68  localparam int E_NO_COFFEE0     = 2;
69  localparam int E_NO_COFFEE1     = 3;
70  localparam int E_NO_CREAMER     = 4;
71  localparam int E_NO_CHOC        = 5;
72  localparam int E_PRESS_ERR      = 6;
73  localparam int E_PRESS_HIGH     = 7;
74  localparam int E_STATUS_ERR     = 8;
75
76  //=============================================================
77  // Rising edge detection for *debounced* buttons
78  //=============================================================
79  logic bf_d, bt_d, bs_d, bstart_d;
80  logic bf_p, bt_p, bs_p, bstart_p;
81  logic btns_armed;
82
83  always_ff @(posedge clk or posedge rst) begin
84      if (rst) begin
85          bf_d       <= 1'b0;
86          bt_d       <= 1'b0;
87          bs_d       <= 1'b0;
88          bstart_d   <= 1'b0;
89          btns_armed <= 1'b0;
90      end else begin
91          bf_d     <= db_flavor;
92          bt_d     <= db_type;
93          bs_d     <= db_size;
94          bstart_d <= db_start;
95          btns_armed <= 1'b1;
96      end
97  end
```

```
98
99  assign bf_p     = btns_armed & (db_flavor & ~bf_d);
100 assign bt_p     = btns_armed & (db_type   & ~bt_d);
101 assign bs_p     = btns_armed & (db_size   & ~bs_d);
102 assign bstart_p = btns_armed & (db_start  & ~bstart_d);
103
104 //============================================================
105 // MAIN STATE MACHINE
106 //============================================================
107 typedef enum logic [1:0] { S_SELECT=2'd0, S_WAIT=2'd1, S_BREW=2'd2 } state_t;
108 state_t state;
109
110 logic        flavor;
111 logic [2:0]  dType;
112 logic [1:0]  dSize;
113
114 logic        flavor_run;
115 logic [2:0]  dType_run;
116 logic [1:0]  dSize_run;
117
118 always_comb begin
119     sys_state = state;
120     if (state == S_SELECT) begin
121         cur_flavor = flavor;
122         cur_type   = dType;
123         cur_size   = dSize;
124     end else begin
125         cur_flavor = flavor_run;
126         cur_type   = dType_run;
127         cur_size   = dSize_run;
128     end
129 end
130
131 //============================================================
132 // Recipe lookup
133 //============================================================
134 logic [3:0] rIndex;
135 coffee_recipe_t recipe_live;
136
137 always_comb begin
138     rIndex      = ({1'b0,dType} * 4'd3) + {2'b0,dSize};
139     recipe_live = coffee_recipe_t'(recipes[rIndex]);
140 end
141
142 coffee_recipe_t recipe_lat;
143
144 //============================================================
145 // Unpack recipe
146 //============================================================
147 logic       r_load_filter, r_add_creamer;
```

```
148 logic [3:0] r_pour_time, r_hot_water_time, r_grinder_time, r_cocoa_time;
149 logic       _unused_HP;
150
151 always_comb begin
152     {r_load_filter, _unused_HP, r_pour_time, r_hot_water_time,
153      r_grinder_time, r_cocoa_time, r_add_creamer} = recipe_lat;
154 end
155
156 logic       lv_load_filter, lv_add_creamer;
157 logic [3:0] lv_pour_time, lv_hot_water_time, lv_grinder_time, lv_cocoa_time;
158 logic       lv_unused_HP;
159
160 always_comb begin
161     {lv_load_filter, lv_unused_HP, lv_pour_time, lv_hot_water_time,
162      lv_grinder_time, lv_cocoa_time, lv_add_creamer} = recipe_live;
163 end
164
165 //============================================================
166 // System error mask
167 //============================================================
168 logic [15:0] sys_err_mask;
169
170 always_comb begin
171     sys_err_mask = 16'b0;
172
173     if (PAPER_LEVEL == 2'b00)      sys_err_mask[E_PAPER_NOT_INST] = 1'b1;
174     else if (PAPER_LEVEL == 2'b01) sys_err_mask[E_PAPER_EMPTY]    = 1'b1;
175
176     if (W_PRESSURE == 2'b11)      sys_err_mask[E_PRESS_ERR]  = 1'b1;
177     else if (W_PRESSURE == 2'b10) sys_err_mask[E_PRESS_HIGH] = 1'b1;
178
179     if (STATUS == 1'b0) sys_err_mask[E_STATUS_ERR] = 1'b1;
180 end
181
182 wire sys_error_condition = |sys_err_mask;
183
184 //============================================================
185 // Ingredient empty decode
186 //============================================================
187 wire bin0_empty = (BIN0_LEVEL == 2'b00) || (BIN0_LEVEL == 2'b01);
188 wire bin1_empty = (BIN1_LEVEL == 2'b00) || (BIN1_LEVEL == 2'b01);
189 wire nd_empty   = (ND_LEVEL   == 2'b00) || (ND_LEVEL   == 2'b01);
190 wire ch_empty   = (CH_LEVEL   == 2'b00) || (CH_LEVEL   == 2'b01);
191
192 //============================================================
193 // Ingredient errors
194 //============================================================
195 logic [15:0] ing_fail_mask;
196 logic [15:0] ing_err_latch;
197
```

```
198 always_comb begin
199     ing_fail_mask = 16'b0;
200 
201     if (lv_grinder_time != 4'd0) begin
202         if (!flavor && bin0_empty) ing_fail_mask[E_NO_COFFEE0] = 1'b1;
203         if ( flavor && bin1_empty) ing_fail_mask[E_NO_COFFEE1] = 1'b1;
204     end
205 
206     if ((lv_cocoa_time != 4'd0) && ch_empty) ing_fail_mask[E_NO_CHOC] = 1'b1;
207     if (lv_add_creamer && nd_empty)          ing_fail_mask[E_NO_CREAMER] = 1'b1;
208 end
209 
210 assign err_mask = sys_err_mask | ing_err_latch;
211 
212 //============================================================
213 // Time constants
214 //============================================================
215 localparam int TICKS_PER_SEC = (CLK_HZ / SPEEDUP_DIV);
216 localparam int ING_ERR_HOLD_SECS = 2;
217 
218 logic [31:0] ing_tick_cnt;
219 logic [3:0]  ing_secs_left;
220 
221 //============================================================
222 // Brewing phase machine
223 //============================================================
224 typedef enum logic [2:0] {
225     PH_PAPER=3'd0, PH_GRIND=3'd1, PH_COCOA=3'd2, PH_POUR=3'd3, PH_WATER=3'd4, PH_DONE=3'd5
226 } phase_t;
227 
228 phase_t phase;
229 assign brew_phase = phase;
230 
231 logic [31:0] tick_cnt;
232 logic [3:0]  sec_left;
233 
234 //============================================================
235 // Duration helper functions
236 //============================================================
237 function automatic [3:0] phase_duration(input phase_t p);
238     case (p)
239         PH_PAPER: phase_duration = (r_load_filter) ? 4'd1 : 4'd0;
240         PH_GRIND: phase_duration = r_grinder_time;
241         PH_COCOA: phase_duration = r_cocoa_time;
242         PH_POUR:  phase_duration = r_pour_time;
243         PH_WATER: phase_duration = r_hot_water_time;
244         default:  phase_duration = 4'd0;
245     endcase
246 endfunction
247 
```

```
248 function automatic phase_t next_phase(input phase_t p);
249     case (p)
250         PH_PAPER: next_phase = PH_GRIND;
251         PH_GRIND: next_phase = PH_COCOA;
252         PH_COCOA: next_phase = PH_POUR;
253         PH_POUR:  next_phase = PH_WATER;
254         PH_WATER: next_phase = PH_DONE;
255         default:  next_phase = PH_DONE;
256     endcase
257 endfunction
258 
259 function automatic phase_t first_nonzero_phase(input phase_t start_p);
260     phase_t p = start_p;
261     for (int k = 0; k < 6; k++) begin
262         if (p == PH_DONE) begin end
263         else if (phase_duration(p) != 4'd0) begin end
264         else p = next_phase(p);
265     end
266     return p;
267 endfunction
268 
269 //============================================================
270 // Progress 0..16
271 //============================================================
272 logic [7:0] total_sec;
273 logic [7:0] elapsed_sec;
274 
275 function automatic [7:0] calc_total_seconds(input coffee_recipe_t r);
276     logic ld, ac;
277     logic [3:0] pt, hw, gt, ct;
278     logic hp;
279     {ld, hp, pt, hw, gt, ct, ac} = r;
280     calc_total_seconds = (ld ? 8'd1 : 8'd0) + pt + hw + gt + ct;
281     if (calc_total_seconds == 8'd0) calc_total_seconds = 8'd1;
282 endfunction
283 
284 function automatic [4:0] calc_progress16(input [7:0] el, input [7:0] tot);
285     int unsigned q = (el * 16) / ((tot == 0) ? 1 : tot);
286     if (q > 16) q = 16;
287     return q[4:0];
288 endfunction
289 
290 //============================================================
291 // MAIN sequential block
292 //============================================================
293 always_ff @(posedge clk or posedge rst) begin
294     if (rst) begin
295         state          <= S_SELECT;
296 
297         flavor         <= 1'b0;
```

```
298         dType          <= 3'd0;
299         dSize          <= 2'd0;
300
301         flavor_run     <= 1'b0;
302         dType_run      <= 3'd0;
303         dSize_run      <= 2'd0;
304
305         recipe_lat     <= '0;
306
307         phase          <= PH_PAPER;
308         tick_cnt       <= 32'd0;
309         sec_left       <= 4'd0;
310
311         total_sec      <= 8'd1;
312         elapsed_sec    <= 8'd0;
313         brew_progress16<= 5'd0;
314
315         ing_err_latch  <= 16'd0;
316         ing_tick_cnt   <= 32'd0;
317         ing_secs_left  <= 4'd0;
318
319     end else begin
320
321         // INGREDIENT ERROR LATCH LOGIC
322         if (sys_error_condition) begin
323             ing_err_latch <= 16'd0;
324             ing_tick_cnt  <= 32'd0;
325             ing_secs_left <= 4'd0;
326         end else if (state != S_SELECT) begin
327             ing_err_latch <= 16'd0;
328             ing_tick_cnt  <= 32'd0;
329             ing_secs_left <= 4'd0;
330         end else begin
331             if (bf_p || bt_p || bs_p) begin
332                 ing_err_latch <= 16'd0;
333                 ing_tick_cnt  <= 32'd0;
334                 ing_secs_left <= 4'd0;
335             end else if (ing_err_latch != 16'd0) begin
336                 if (ing_secs_left == 4'd0) begin
337                     ing_err_latch <= 16'd0;
338                     ing_tick_cnt  <= 32'd0;
339                 end else begin
340                     if (ing_tick_cnt == (TICKS_PER_SEC-1)) begin
341                         ing_tick_cnt <= 32'd0;
342                         if (ing_secs_left <= 4'd1) begin
343                             ing_err_latch <= 16'd0;
344                             ing_secs_left <= 4'd0;
345                         end else begin
346                             ing_secs_left <= ing_secs_left - 4'd1;
347                         end
```

```
348                     end else begin
349                         ing_tick_cnt <= ing_tick_cnt + 32'd1;
350                     end
351                 end
352             end else begin
353                 ing_tick_cnt  <= 32'd0;
354                 ing_secs_left <= 4'd0;
355             end
356         end
357 
358         // SYSTEM ERROR → RESET TO SELECT
359         if (sys_error_condition) begin
360             state           <= S_SELECT;
361             phase           <= PH_PAPER;
362             tick_cnt        <= 32'd0;
363             sec_left        <= 4'd0;
364             elapsed_sec     <= 8'd0;
365             brew_progress16 <= 5'd0;
366 
367         end else begin
368 
369             // SELECTION HANDLING
370             if (state == S_SELECT) begin
371                 if (bf_p) flavor <= ~flavor;
372                 if (bt_p) dType  <= (dType == 3'd4) ? 3'd0 : (dType + 3'd1);
373                 if (bs_p) dSize  <= (dSize == 2'd2) ? 2'd0 : (dSize + 2'd1);
374             end
375 
376             case (state)
377 
378                 //------------------------------------------------------------
379                 // SELECT
380                 //------------------------------------------------------------
381                 S_SELECT: begin
382                     if (bstart_p) begin
383                         if (ing_fail_mask != 16'b0) begin
384                             ing_err_latch <= ing_fail_mask;
385                             ing_tick_cnt  <= 32'd0;
386                             ing_secs_left <= ING_ERR_HOLD_SECS[3:0];
387 
388                             elapsed_sec      <= 8'd0;
389                             brew_progress16  <= 5'd0;
390                             phase            <= PH_PAPER;
391                             sec_left         <= 4'd0;
392                             tick_cnt         <= 32'd0;
393 
394                         end else begin
395                             flavor_run <= flavor;
396                             dType_run  <= dType;
397                             dSize_run  <= dSize;
```

```
398
399                         recipe_lat      <= recipe_live;
400                         total_sec       <= calc_total_seconds(recipe_live);
401                         elapsed_sec     <= 8'd0;
402                         brew_progress16 <= 5'd0;
403
404                         phase           <= PH_PAPER;
405                         sec_left        <= 4'd0;
406                         tick_cnt        <= 32'd0;
407
408                         ing_err_latch   <= 16'd0;
409                         ing_tick_cnt    <= 32'd0;
410                         ing_secs_left   <= 4'd0;
411
412                         state           <= S_WAIT;
413                     end
414                 end
415             end
416
417             //------------------------------------------------------------
418             // WAIT FOR WATER TEMP
419             //------------------------------------------------------------
420             S_WAIT: begin
421                 if (W_TEMP == 1'b1) begin
422                     state       <= S_BREW;
423                     tick_cnt    <= 32'd0;
424                     elapsed_sec <= 8'd0;
425                     brew_progress16 <= 5'd0;
426
427                     phase    <= first_nonzero_phase(PH_PAPER);
428                     sec_left <= phase_duration(first_nonzero_phase(PH_PAPER));
429                 end
430             end
431
432             //------------------------------------------------------------
433             // BREW PROCESS
434             //------------------------------------------------------------
435             S_BREW: begin
436                 if (phase == PH_DONE) begin
437                     state           <= S_SELECT;
438                     phase           <= PH_PAPER;
439                     tick_cnt        <= 32'd0;
440                     sec_left        <= 4'd0;
441                     elapsed_sec     <= total_sec;
442                     brew_progress16 <= 5'd16;
443
444                 end else begin
445                     if (tick_cnt == (TICKS_PER_SEC - 1)) begin
446                         tick_cnt <= 32'd0;
447
```

```
448                             if (elapsed_sec < total_sec)
449                                 elapsed_sec <= elapsed_sec + 8'd1;
450 
451                             brew_progress16 <= calc_progress16(
452                                 ((elapsed_sec < total_sec) ? (elapsed_sec + 8'd1) :
    elapsed_sec),
453                                 total_sec
454                             );
455 
456                             if (sec_left != 4'd0)
457                                 sec_left <= sec_left - 4'd1;
458 
459                             if (sec_left == 4'd1) begin
460                                 phase    <= first_nonzero_phase(next_phase(phase));
461                                 sec_left <=
    phase_duration(first_nonzero_phase(next_phase(phase)));
462                             end
463 
464                         end else begin
465                             tick_cnt <= tick_cnt + 32'd1;
466                         end
467                     end
468                 end
469 
470             endcase
471         end
472     end
473 end
474 
475 //============================================================
476 // Output logic
477 //============================================================
478 always_comb begin
479     HEAT_EN      = 1'b0;
480     POUROVER_EN  = 1'b0;
481     WATER_EN     = 1'b0;
482     GRINDER_0_EN = 1'b0;
483     GRINDER_1_EN = 1'b1;
484     PAPER_EN     = 1'b0;
485     COCOA_EN     = 1'b0;
486     CREAMER_EN   = 1'b0;
487 
488     if (!sys_error_condition) begin
489 
490         if ((state != S_SELECT) && (W_TEMP == 1'b0))
491             HEAT_EN = 1'b1;
492 
493         if (state == S_BREW) begin
494             if (sec_left != 4'd0) begin
495                 case (phase)
```

```
496                     PH_PAPER: PAPER_EN = 1'b1;
497                     PH_GRIND: begin
498                         if (!flavor_run) GRINDER_0_EN = 1'b1;
499                         else             GRINDER_1_EN = 1'b1;
500                     end
501                     PH_COCOA: COCOA_EN = 1'b1;
502                     PH_POUR:  POUROVER_EN = 1'b1;
503                     PH_WATER: WATER_EN = 1'b1;
504                 endcase
505             end
506 
507             if (r_add_creamer)
508                 CREAMER_EN = 1'b1;
509         end
510     end
511 end
512 
513 endmodule
514 
```

D. lcdIp.sv

```
1  //=============================================================
2  // lcdIp: simple 8-bit HD44780 write engine (instruction/data)
3  // RS=0 for instructions, RS=1 for data. R/W is always 0.
4  //=============================================================
5  module lcdIp (
6      input  wire       clk,
7      input  wire [1:0] userOp,         // 00=INSTR, 01=DATA
8      input  wire       send,
9      input  wire       reset,
10     input  wire [7:0] inputCommand,
11     output reg  [7:0] lcd_data,
12     output reg        lcd_rs,
13     output reg        lcd_rw,
14     output reg        lcd_e,
15     output reg        busy,
16     output reg        systemReady
17 );
18     localparam OP_INSTR = 2'b00;
19     localparam OP_DATA  = 2'b01;
20
21     typedef enum logic [2:0] { INIT_WAIT, IDLE, LOAD, SETUP, E_HIGH, E_LOW, WAIT_DONE }
   state_t;
22     state_t st;
23
24     reg [7:0] cmd_latched;
25     reg [1:0] op_latched;
26     reg [19:0] t; // enough for ~1.5ms at 50MHz
27
28     // timing @50MHz
29     localparam integer T_INIT   = 750_000;  // ~15ms after power-up
30     localparam integer T_SETUP  = 4;        // >=80ns setup (4 cycles @20ns)
31     localparam integer T_E_PW   = 1000;      // 2 µs E high
32     localparam integer T_E_LOW  = 1000;      // 2 µs guard before wait
33     localparam integer T_CMD    = 2_500;    // 50 µs (>37 µs)
34     localparam integer T_CLEAR  = 100_000;  // 2 ms (>1.52 ms)
35
36     wire is_clear_or_home = (op_latched==OP_INSTR) && ((cmd_latched==8'h01) ||
   (cmd_latched==8'h02));
37
38     // FSM
39     always @(posedge clk or posedge reset) begin
40         if (reset) begin
41             st <= INIT_WAIT; t <= T_INIT;
42             lcd_data <= 8'h00;
43             lcd_rs   <= 1'b0;
44             lcd_rw   <= 1'b0;
45             lcd_e    <= 1'b0;
46             busy     <= 1'b1;
47             systemReady <= 1'b0;
48             cmd_latched <= 8'h00;
```

```
49              op_latched  <= OP_INSTR;
50          end else begin
51              // defaults
52              lcd_rw <= 1'b0; // always write
53              systemReady <= (st==IDLE);
54
55              case (st)
56                  INIT_WAIT: begin
57                      busy <= 1'b1; lcd_e <= 1'b0;
58                      if (t!=0) t <= t-1;
59                      else begin st <= IDLE; busy <= 1'b0; end
60                  end
61
62                  IDLE: begin
63                      busy <= 1'b0; lcd_e <= 1'b0;
64                      if (send) begin
65                          cmd_latched <= inputCommand;
66                          op_latched  <= userOp;
67                          st <= LOAD;
68                      end
69                  end
70
71                  LOAD: begin
72                      // Place RS and DATA and then wait a few cycles before E↑
73                      busy     <= 1'b1;
74                      lcd_rs   <= (op_latched==OP_DATA);
75                      lcd_data <= cmd_latched;
76                      t        <= T_SETUP;
77                      st       <= SETUP;
78                  end
79
80                  SETUP: begin
81                      // Meet HD44780 address/data setup time before toggling E
82                      if (t!=0) t <= t-1;
83                      else begin
84                          t  <= T_E_PW;
85                          st <= E_HIGH;
86                      end
87                  end
88
89                  E_HIGH: begin
90                      lcd_e <= 1'b1;
91                      if (t!=0) t <= t-1;
92                      else begin
93                          t  <= T_E_LOW;
94                          st <= E_LOW; // falling edge latches into LCD
95                      end
96                  end
97
98                  E_LOW: begin
```

```
                    lcd_e <= 1'b0;
                    if (t!=0) t <= t-1;
                    else begin
                        t  <= is_clear_or_home ? T_CLEAR : T_CMD;
                        st <= WAIT_DONE;
                    end
                end

                WAIT_DONE: begin
                    if (t!=0) t <= t-1;
                    else begin
                        st <= IDLE; busy <= 1'b0;
                    end
                end
            endcase
        end
    end
endmodule
```