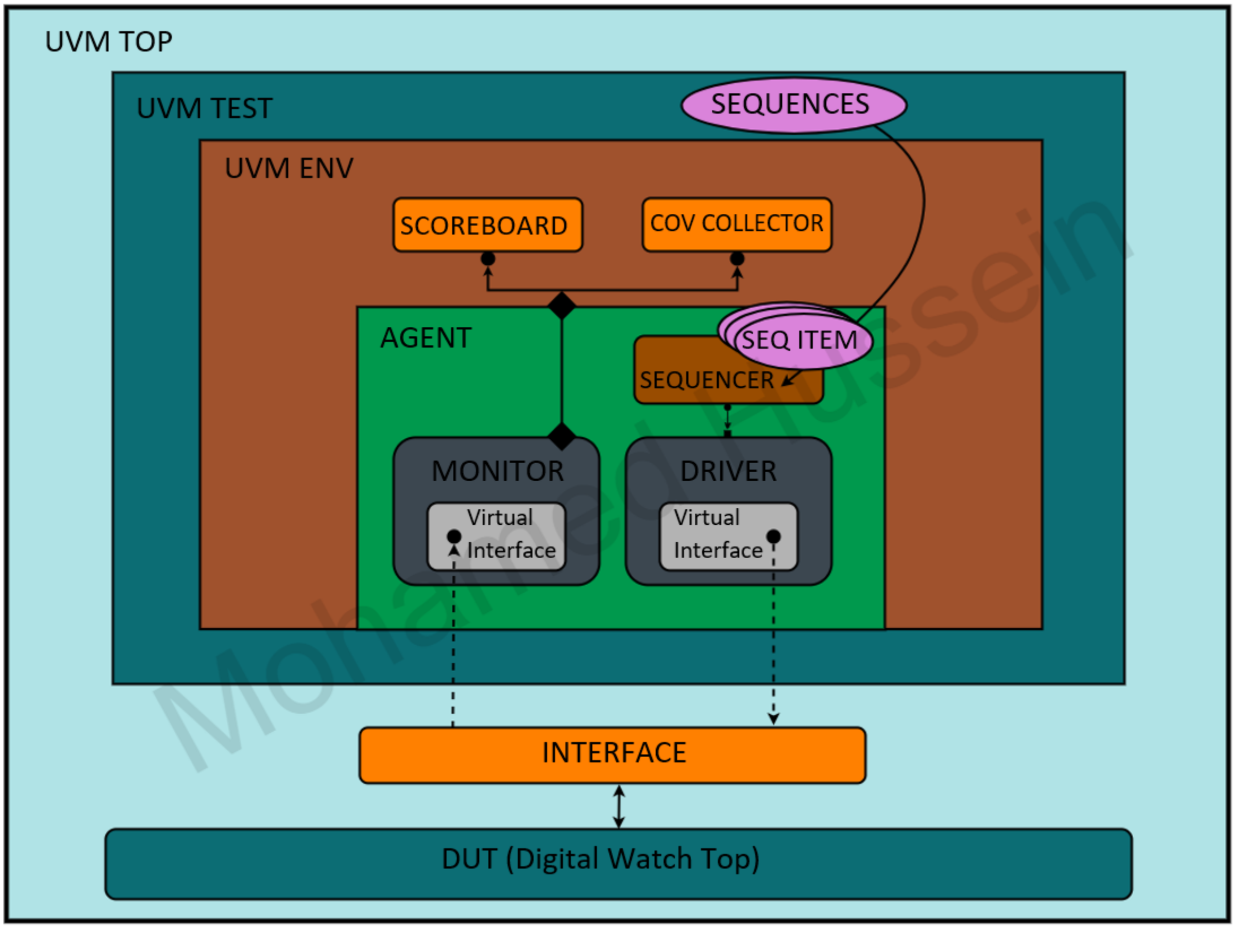
ASIC Design and Automation



Digital Watch Front-End Design

Students		Department
Name	Mohamed Ahmed Mohamed Hussein	ECE
ID	2100924	
Email	2100924@eng.asu.edu.eg	
Name	Kirollos Rafat Fouad	ECE
ID	2100491	
Email	2100491@eng.asu.edu.eg	
Name	Hanin Ahmed Abdelfattah Ahmed	ECE
ID	2100661	
Email	2100661@eng.asu.edu.eg	
Name	Habiba Essam Eldin Abdelhamid	ECE
ID	2100321	
Email	2100321@eng.asu.edu.eg	

Chapter 1 “Introduction”

- **Project Idea:**

The project’s main purpose is to develop and mimic a real-world Digital Watch implementation that provides multiple functionalities to the user.

- **Problem description:**

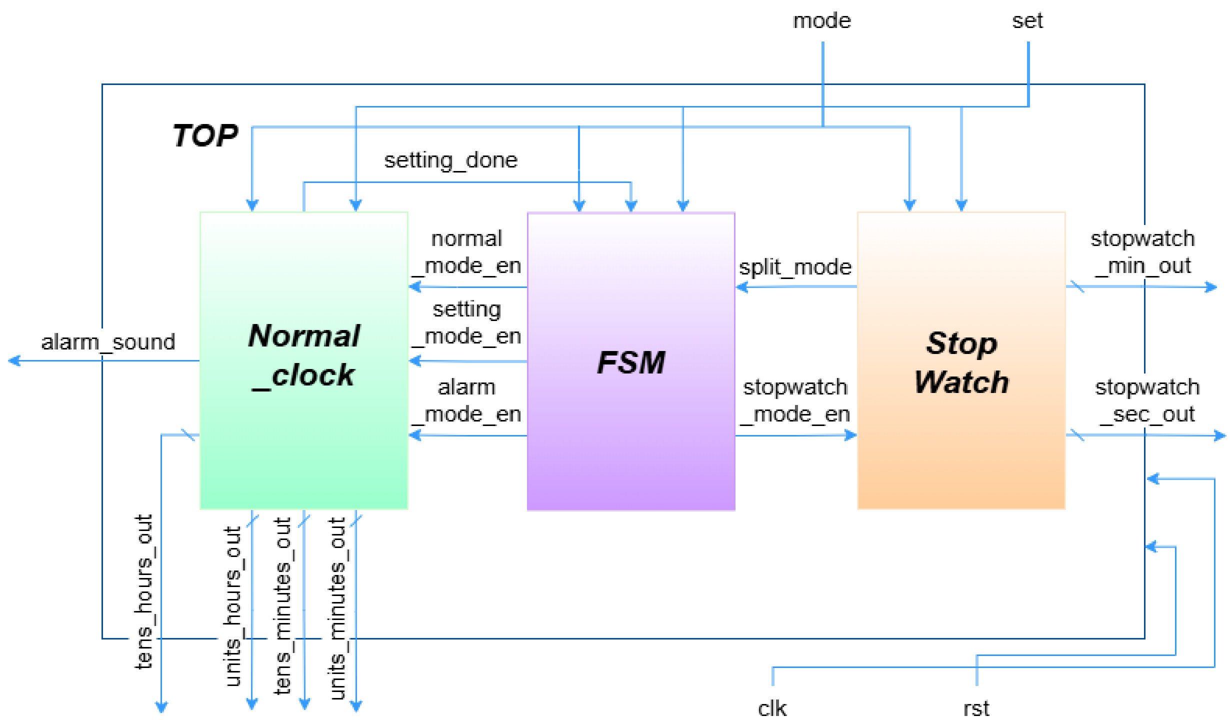
The digital watch is merely designed for normal clock operation, stopwatch with two modes of operations and an alarm. These functions resemble the core operations needed for any user of a digital watch. This part aims to go through the front end process of the implementation, the RTL development and the Verification flow.

Chapter 2: “FSM design”

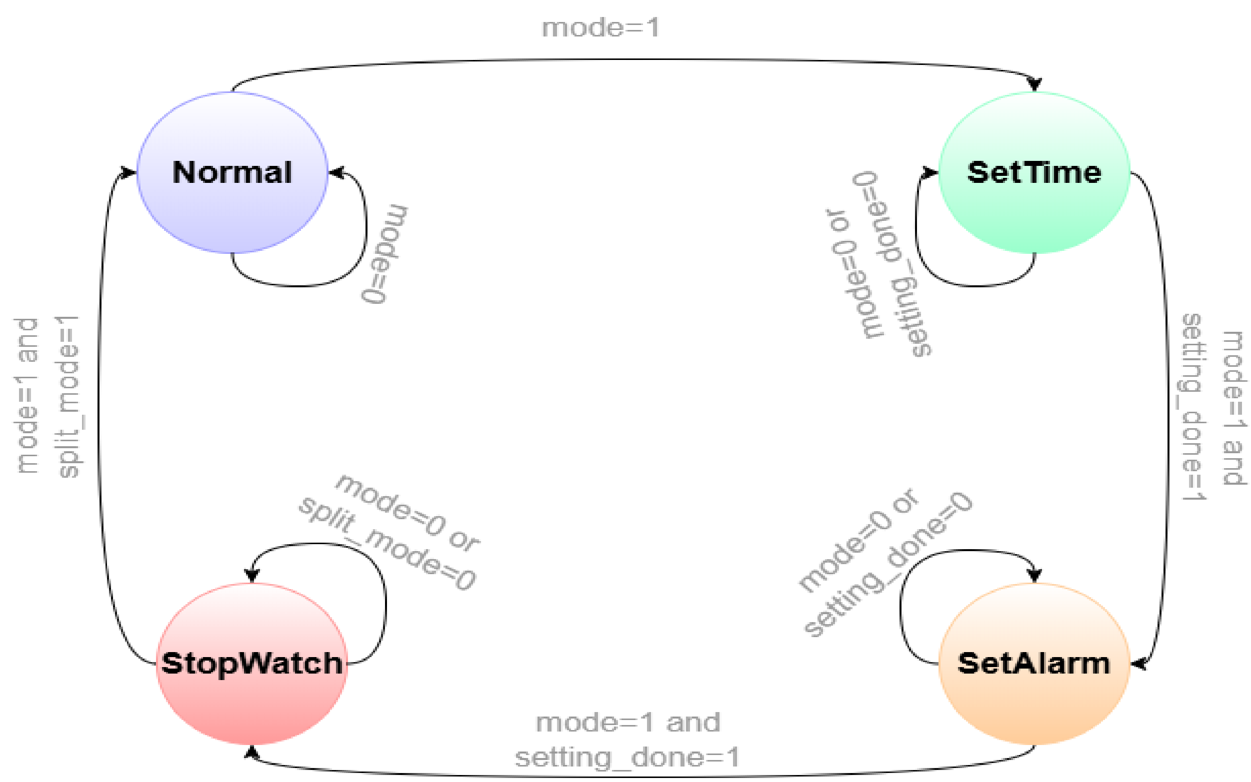
- **Design details:** The FSM orchestrates between different operations and the next table is dedicated to elaborate more on the side of the chosen specs for the design of interest.

Module	Description
Normal Clock	• **Time Generation & Display:** Continuously generates and updates the current hours and minutes in normal mode with correct rollover behavior, and provides digit-wise outputs for display. • **Time & Alarm Setting:** Allows the user to set the current time and the alarm time digit-by-digit, storing the programmed values for later use.Each time set is asserted by the user the digit gets incremented by 1 • **Alarm Activation:** Compares the current time with the saved alarm time and activates the alarm sound for a fixed duration when a match occurs. A.for 20 seconds and then stops ringing is adjusted to keep larm
Stopwatch	• **Elapsed Time Mode:** Operates as a standard stopwatch with start, stop, resume, and clear functions, counting minutes and seconds from 00:00 up to 59:59. • **Split Time Mode:** Allows capturing and holding a split (lap) time while the internal stopwatch continues running in the background and continue back from the split time. • **Mode Switching Control:** Mode button toggles between elapsed time and split time modes using the mode input, with split_mode indicating the current operating mode. Each time mode is asserted by the user we traverse to another digit say (from tens of hours to units of hours).
FSM Module	• **Central Mode Controller:** Acts as the main control unit that manages all operating modes of the digital watch: Normal display, Time Setting, Alarm Setting, and Stopwatch. • **Enable Control:** Generates enable signals (normal_mode_en, setting_mode_en, alarm_mode_en, stopwatch_mode_en) to activate the appropriate module at the correct time. • **User Interaction :** Coordinates user inputs (mode, set) with internal module feedback to ensure correct sequencing of time setting, alarm setup, and stopwatch operation.

- ## System diagram:



- ## FSM diagram:



Chapter 3 "RTL design"

- ## RTL codes:

Normal_Clock Module:

```
// this module is to generate the seconds, minutes and hours for normal display
mode in the digital watch
module Normal_Clock (
    input wire clk,               // system clock
    input wire rst,            // active low reset
    input wire mode,            // mode signal to switch between normal mode and
setting mode
    input wire set,             // set signal to increment the time in setting
mode

    // fsm control signals
    input wire normal_mode_en,    // normal_mode_enble signal
    input wire alarm_mode_en,     // alarm_mode_enable signal
    input wire setting_mode_en,   // setting_mode_enable signal

    // outputs for fsm
    output reg setting_done,      // to indicate if setting is done (we are in
the last digit, most right)

    // display outputs
    output [1:0] tens_hours_out,  // tens hours output (0-2)
    output [3:0] units_hours_out, // units hours output (0-9)
    output [2:0] tens_minutes_out, // tens minutes output (0-5)
    output [3:0] units_minutes_out, // units minutes output (0-9)
    output alarm_sound              // alarm sound output
);
    // Digits internal signals
    reg [1:0] tens_hours; // represents the left digit of hours (0-2) (most left)
    reg [3:0] units_hours; // represents the right digit of hours (0-9)
    reg [2:0] tens_minutes; // represents the left digit of minutes (0-5)
    reg [3:0] units_minutes; // represents the right digit of minutes (0-9) (most
right)

    reg [1:0] setting_digit; // to indicate which digit is being set from left to
right


    // seconds counter
```

```
    reg [5:0] sec_count;    // seconds output (0-59)

    // Setting mode internal signals
    reg [5:0] set_hours;
    reg [6:0] set_minutes;

    // Alarm mode internal signals
    reg [5:0] alarm_hours;
    reg [6:0] alarm_minutes;
    reg [4:0] alarm_counter; // alarm should sound for a 20 seconds duration
    reg alarm_active;       // to indicate if alarm is currently active
    reg alarm_done;         // to indicate if alarm setting is done, use this to
begin the normal time with value saved in set_hours and set_minutes
    reg first_alarm;    // to indicate the first time alarm is set, to keep alarm
shut when reset

    // Seconds counter
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            sec_count <= 6'd0;
        end
        else if (setting_mode_en) begin
            // In setting mode, seconds reset to 0 and the time is paused until
setting is done
            sec_count <= 6'd0;
        end
        else begin // if any mode but setting the watch will work
            if (sec_count == 6'd59) begin
                sec_count <= 6'd0;
            end else begin
                sec_count <= sec_count + 6'd1;
            end
        end
    end

    // Hours and Minutes counter (both in the same always block)
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            tens_hours <= 2'd0;
            units_hours <= 4'd0;
            units_minutes <= 4'd0;
            tens_minutes <= 3'd0;
            setting_digit <= 2'b00;
            setting_done <= 1'b0;
        end
```

```
        else begin
            if (setting_mode_en || alarm_mode_en) begin
                // In setting mode, the time is paused until setting is done
                // choosing the digit being set
                if (!setting_digit) begin
                    setting_done <= 1'b0; // reset setting done flag
                    tens_hours <= 2'd0;
                    units_hours <= 4'd0;
                    units_minutes <= 4'd0;
                    tens_minutes <= 3'd0;
                    if (mode) begin // transition to the next digit
                        setting_digit <= 2'b01; // move to next digit
                    end
                    else if (set && tens_hours < 2) begin // make sure tens of
hours does not exceed 2
                        tens_hours <= tens_hours + 1;
                    end
                    else if (set) begin
                        tens_hours <= 0;
                    end
                end
                else if (setting_digit == 2'b01) begin
                    if (mode) begin // transition to the next digit
                        setting_digit <= 2'b10; // move to next digit
                    end
                    else if (set && units_hours < 9) begin // make sure units of
hours does not exceed 9
                            units_hours <= units_hours + 1;
                    end
                    else if (set) begin
                        units_hours <= 0;
                    end
                end
                else if (setting_digit == 2'b10) begin
                    if (mode) begin // transition to the next digit
                        setting_digit <= 2'b11; // move to next digit
                    end
                    else if (set && tens_minutes < 5) begin // make sure tens of
minutes does not exceed 5
                        tens_minutes <= tens_minutes + 1;
                    end
                    else if (set) begin
                        tens_minutes <= 0;
                    end
                end
```

```
                else if (setting_digit == 2'b11) begin
                    setting_done <= 1'b1; // indicate setting is done
                    if (mode) begin
                        setting_digit <= 2'b00; // reset to first digit
                    end
                    else if (set && units_minutes < 9) begin // make sure units
of minutes does not exceed 9
                        units_minutes <= units_minutes + 1;
                    end
                    else if (set) begin
                        units_minutes <= 0;
                    end
                end
            end
            else if (normal_mode_en) begin
                if (alarm_done) begin
                    {tens_minutes, units_minutes} <= set_minutes; // load the set
minutes
                    {tens_hours, units_hours} <= set_hours;       // load the set
hours
                end
                else begin
                    if (sec_count == 6'd59) begin
                        if (units_minutes == 4'd9 && tens_minutes == 3'd5) begin
                            units_minutes <= 4'd0;
                            tens_minutes <= 3'd0;
                        end
                        else if (units_minutes < 9) begin
                            units_minutes <= units_minutes + 1; // increment
units of minutes
                        end
                        else begin
                            tens_minutes <= tens_minutes + 1; // increment tens
of minutes
                        end
                    end
                    if (sec_count == 6'd59 && units_minutes == 4'd9 &&
tens_minutes == 3'd5) begin
                        if (tens_hours == 2'd2 && units_hours == 4'd3) begin
                            tens_hours <= 2'd0;
                            units_hours <= 4'd0;
                        end
                        else if (units_hours < 4'd9) begin
                            units_hours <= units_hours + 4'd1;
                        end
```

```
                        else begin
                            tens_hours <= tens_hours + 2'd1;
                        end
                    end
                end
            end
        end
    end

    // Setting time mode/Alarm mode always block to save the setting time/alarm
time
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            alarm_hours <= 6'd0;
            alarm_minutes <= 7'd0;
            set_hours <= 6'd0;
            set_minutes <= 7'd0;
            alarm_done <= 1'b0;
            first_alarm <= 1'b0;
        end else if  (setting_mode_en) begin
            set_hours <= {tens_hours, units_hours}; // combine tens and units
hours
            set_minutes <= {tens_minutes, units_minutes}; // combine tens and
units minutes
        end else if (alarm_mode_en) begin
            alarm_hours <= {tens_hours, units_hours}; // combine tens and units
hours
            alarm_minutes <= {tens_minutes, units_minutes}; // combine tens and
units minutes
            alarm_done <= 1'b1;
            first_alarm <= 1'b1;
        end
        else begin // if normal or stopwatch mode turn off the alarm done signal
            alarm_done <= 1'b0; // reset alarm done flag
        end
    end

    // Alarm sound control
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            alarm_counter <= 5'd0;
            alarm_active <= 1'b0;
        end else if ((alarm_hours == {tens_hours, units_hours}) && (alarm_minutes
== {tens_minutes, units_minutes})) begin
```

```
            if (normal_mode_en && !alarm_mode_en && first_alarm && alarm_counter
< 5'd22) begin
                if (alarm_counter > 1) alarm_active <= 1'b1;
                alarm_counter <= alarm_counter + 5'd1; // count up to 20 seconds
            end
            else begin
                alarm_active <= 1'b0;
            end
        end
        else begin
            alarm_active <= 1'b0;
            alarm_counter <= 5'd0; // reset counter when time does not match
alarm time
        end
    end

    // Output assignments
    assign tens_hours_out = tens_hours;
    assign units_hours_out = units_hours;
    assign tens_minutes_out = tens_minutes;
    assign units_minutes_out = units_minutes;
    // alarm should sound when current time matches alarm time
    assign alarm_sound = alarm_active;
endmodule
```

Digital_Watch_FSM Module:

```
// this module acts as the controlling unit for a digital watch which can display
hours, minutes and set alarm and make a stop watch function
module Digital_Watch_FSM (
    input clk,
    input rst,
    input mode, // to transition between time display, set time, alarm set, and
stopWatch
    input set,  // to set the time or alarm

    // inputs from Normal_Clock
    input setting_done, // to indicate if setting is done (we are in the last
digit, most right)

    // inputs from StopWatch
    input split_mode, // to indicate if it's in split time mode as if mode=1 we
will go back to normal time

    // control signals to Normal_Clock
```

```verilog
    output reg normal_mode_en, // to enable normal clock display mode
    output reg setting_mode_en, // to enable setting mode to set the time
    output reg alarm_mode_en, // to enable alarm setting mode
    output reg stopwatch_mode_en // to enable stopwatch mode
);
    localparam Normal = 2'b00,
               SetTime = 2'b01,
               SetAlarm = 2'b10,
               StopWatch = 2'b11;

    reg [1:0] current_state, next_state;

    // state memory
    always @(posedge clk or negedge rst) begin
        if (!rst)
            current_state <= Normal; // start in SetTime mode
        else
            current_state <= next_state;
    end

    // next state logic
    always @(*) begin
        case (current_state) // use mode to transition between modes
            Normal: begin
                if (mode) next_state = SetTime;
                else next_state = Normal;
            end
            SetTime: begin
                if (mode && setting_done) next_state = SetAlarm;
                else next_state = SetTime;
            end
            SetAlarm: begin
                if (mode && setting_done) next_state = StopWatch;
                else next_state = SetAlarm;
            end
            StopWatch: begin
                if (mode && split_mode) next_state = Normal;
                else next_state = StopWatch;
            end
            default: next_state = Normal;
        endcase
    end

    // output logic
    always @(*) begin
```

```
        // Default values for all outputs
        normal_mode_en = 1'b0;
        setting_mode_en = 1'b0;
        alarm_mode_en = 1'b0;
        stopwatch_mode_en = 1'b0;

        case(current_state)
            Normal: begin
                normal_mode_en = 1'b1;
            end
            SetTime: begin
                setting_mode_en = 1'b1;
            end
            SetAlarm: begin
                alarm_mode_en = 1'b1;
            end
            StopWatch: begin
                stopwatch_mode_en = 1'b1;
                normal_mode_en = 1'b1;
            end
            default: begin  // Default case handles Normal mode
                normal_mode_en = 1'b1;
            end
        endcase
    end
endmodule
```

StopWatch Module:

```
// this module is to make a stopwatch which has elapsed time and split time
functions
module StopWatch (
    input clk,
    input rst,
    input mode, // to transition between elapsed time and split time
    input set, // to start and stop the stopwatch

    // input from Digital_Watch FSM
    input stopwatch_mode_en, // to indicate if it's in stopwatch mode


    // outputs for display (mm:ss)
    output reg [5:0] min_out, // minutes output (0-59)
    output reg [5:0] sec_out,  // seconds output (0-59)
```

```
    output reg split_mode // to indicate if it's in split time mode as if mode=1
we will go back to normal time
);
    // Stopwatch internal signals
    reg stopwatch_mode; // if it's Elapsed time (default) or Split Time
    reg split_captured; // to indicate if split time is captured
    reg split_pulse; // to maintain a pulse for split time capture (for timing
purposes)
    reg split_pulse_captured; // to indicate if split pulse is captured

    // internal seconds and minutes
    reg [5:0] sec_count;
    reg [5:0] min_count;
    reg [5:0] sec_count_split;
    reg [5:0] min_count_split;

    // counters for different states in each mode
    reg [2:0] Elapsed_state; // states for elapsed time mode
    reg [2:0] Split_state;   // states for split time mode

    // Stopwatch seconds counter
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            sec_count <= 6'd0;
            min_count <= 6'd0;
            stopwatch_mode <= 1'b0; // default to elapsed time
            split_captured <= 1'b0;
            sec_count_split <= 6'd0;
            min_count_split <= 6'd0;
            split_mode <= 1'b0;
            split_pulse <= 1'b0;
        end
        else if (stopwatch_mode_en) begin
            if (mode) begin
                stopwatch_mode <= ~stopwatch_mode; // toggle between elapsed and
split time
            end
            // In elapsed time mode, do not increment the stopwatch when stop,
incerement it otherwise
            else if (!stopwatch_mode) begin
                split_mode <= 1'b0; // indicate we are not in split mode
                if (Elapsed_state == 3'b001 || Elapsed_state == 3'b011) begin //
running states
                    if (sec_count == 6'd59) begin
                        sec_count <= 6'd0;
```

```
                        if (min_count == 6'd59) begin
                            min_count <= 6'd0;
                        end else begin
                            min_count <= min_count + 6'd1;
                        end
                    end else begin
                        sec_count <= sec_count + 6'd1;
                    end
                end else if (Elapsed_state == 3'b000) begin
                    // In cleared state, reset the stopwatch
                    sec_count <= 6'd0;
                    min_count <= 6'd0;
                end
            end
            // In split time mode, increment the stopwatch even when splitting
            else begin
                split_mode <= 1'b1; // indicate we are in split mode
                if (split_pulse) begin
                    split_pulse <= 1'b0; // reset the split pulse after one cycle
                end
                if (Split_state == 3'b010 && !split_captured) begin // in split
state, we should keep the time we are at without stopping incrementing
                    sec_count_split <= sec_count;
                    min_count_split <= min_count;
                    split_captured <= 1'b1;
                    split_pulse <= 1'b1;
                end
                if (Split_state == 3'b001 || Split_state == 3'b010 || Split_state
== 3'b011) begin // running states (start and split released)
                    if (sec_count == 6'd59) begin
                        sec_count <= 6'd0;
                        if (min_count == 6'd59) begin
                            min_count <= 6'd0;
                        end else begin
                            min_count <= min_count + 6'd1;
                        end
                    end else begin
                        sec_count <= sec_count + 6'd1;
                    end
                end else if (Split_state == 3'b000) begin
                    // In cleared state, reset the stopwatch
                    sec_count <= 6'd0;
                    min_count <= 6'd0;
                    split_captured <= 1'b0;
                end
```

```
            end
        end
    end

    // which state we are in
    always @(posedge clk or negedge rst) begin
        if (!rst) begin
            Elapsed_state <= 3'b000;
            Split_state <= 3'b000;
        end
        else if (stopwatch_mode_en) begin
            if (!stopwatch_mode) begin
                // Elapsed time mode state transitions
                case (Elapsed_state)
                    3'b000: if (set) Elapsed_state <= 3'b001; // start
                    3'b001: if (set) Elapsed_state <= 3'b010; // stop
                    3'b010: if (set) Elapsed_state <= 3'b011; // resume
                    3'b011: if (set) Elapsed_state <= 3'b100; // stop again
                    3'b100: if (set) Elapsed_state <= 3'b000; // clear
                    default: Elapsed_state <= 3'b000;
                endcase
            end else begin
                // Split time mode state transitions
                case (Split_state)
                    3'b000: if (set) Split_state <= 3'b001; // start
                    3'b001: if (set) Split_state <= 3'b010; // split
                    3'b010: if (set) Split_state <= 3'b011; // split release
                    3'b011: if (set) Split_state <= 3'b100; // stop
                    3'b100: if (set) Split_state <= 3'b000; // clear
                    default: Split_state <= 3'b000;
                endcase
            end
        end
    end

    // output assignments
    always @(*) begin
        if (split_pulse) begin // the whole split pulse and split pulse captured
thing is for timing purposes
            min_out = min_count_split;
            sec_out = sec_count_split;
            split_pulse_captured = 1;
        end
        else if (split_pulse_captured && Split_state == 3'b010) begin
            min_out = min_count_split;
```

```
            sec_out = sec_count_split;
            split_pulse_captured = 1;
        end
        else begin
            min_out = min_count;
            sec_out = sec_count;
            split_pulse_captured = 0;
        end
    end

    //assign min_out = (Split_state == 3'b010) ? min_count_split : min_count;
    //assign sec_out = (Split_state == 3'b010) ? sec_count_split : sec_count;
endmodule
```

Digital_Watch_Top Module:

```
/ // this module is the top module that integrates Normal_Clock, StopWatch, and
Digital_Watch FSM

module Digital_Watch_Top (
    input clk,
    input rst,
    input mode, // to transition between time display, set time, alarm set, and
stopWatch
    input set,  // to set the time or alarm

    // outputs for display from Normal_Clock
    output [1:0] tens_hours_out,  // tens hours output (0-2)
    output [3:0] units_hours_out, // units hours output (0-9)
    output [2:0] tens_minutes_out, // tens minutes output (0-5)
    output [3:0] units_minutes_out, // units minutes output (0-9)
    output alarm_sound,              // alarm sound output

    // outputs for display from StopWatch
    output [5:0] stopwatch_min_out, // stopwatch minutes output (0-59)
    output [5:0] stopwatch_sec_out  // stopwatch seconds output (0-59)
);
    // Control signals from Digital_Watch FSM
    wire normal_mode_en;
    wire setting_mode_en;
    wire alarm_mode_en;
    wire stopwatch_mode_en;

    // Signals from Normal_Clock
    wire setting_done;
```

```
// Signals from StopWatch
wire split_mode;

// Instantiate Digital_Watch FSM
Digital_Watch_FSM fsm_inst (
    .clk(clk),
    .rst(rst),
    .mode(mode),
    .set(set),
    .setting_done(setting_done),
    .split_mode(split_mode),
    .normal_mode_en(normal_mode_en),
    .setting_mode_en(setting_mode_en),
    .alarm_mode_en(alarm_mode_en),
    .stopwatch_mode_en(stopwatch_mode_en)
);

// Instantiate Normal_Clock
Normal_Clock clock_inst (
    .clk(clk),
    .rst(rst),
    .mode(mode),
    .set(set),
    .normal_mode_en(normal_mode_en),
    .alarm_mode_en(alarm_mode_en),
    .setting_mode_en(setting_mode_en),
    .setting_done(setting_done),
    .tens_hours_out(tens_hours_out),
    .units_hours_out(units_hours_out),
    .tens_minutes_out(tens_minutes_out),
    .units_minutes_out(units_minutes_out),
    .alarm_sound(alarm_sound)
);

// Instantiate StopWatch
StopWatch stopwatch_inst (
    .clk(clk),
    .rst(rst),
    .mode(mode),
    .set(set),
    .stopwatch_mode_en(stopwatch_mode_en),
    .min_out(stopwatch_min_out),
    .sec_out(stopwatch_sec_out),
    .split_mode(split_mode)
```

```
    );
endmodule
```

- ## Comments:

  - ### Normal_Clock Module:

The **Normal_Clock** module implements the main time-keeping and time-setting logic of the digital watch. It is responsible for maintaining the current **hours** and **minutes**, managing **time-setting**, and supporting an **alarm feature**. The module operates under the control of external mode signals from the FSM, which determine whether the watch is in **normal mode**, **setting mode**, or **alarm-setting mode**.

  - ### Digital_Watch_FSM Module:

The **Digital_Watch_FSM** module is the central control unit of the digital watch. It decides which functional block of the system is active based on user inputs and internal completion flags. Through mode transitions, it orchestrates how the system moves between **normal display**, **time setting**, **alarm setting**, and **stopwatch operation**.

  - ### StopWatch Module:

The **StopWatch** module implements the stopwatch feature of the digital watch. It supports two major operating modes:

1. **Elapsed Time Mode** — traditional stopwatch (start → stop → resume → stop → clear)
2. **Split Time Mode** — allows capturing intermediate “split” times while the stopwatch continues running in the background

The module keeps track of minutes and seconds, manages a multi-step user interaction flow, and generates display values depending on whether the user is viewing live elapsed time or a frozen split time snapshot.

  - ### Digital_Watch_Top Module:

The **Digital_Watch_Top** module integrates the three main subsystems of the digital watch:

1. **Digital_Watch_FSM** (controls modes)
2. **Normal_Clock** (timekeeping + alarm)
3. **StopWatch** (stopwatch counter)

The top module only connects signals between these submodules. It does **not** implement logic itself.

Chapter 4 “TB design”

• Test scenario:

Testbench Verification Flow

The testbench simulates the complete functionality of the **Digital Watch Top Module** by generating a 1-Hz clock, applying mode transitions, and stimulating the set push button to emulate real user interaction. The following steps represent the overall verification flow:

1. **Clock and Reset Initialization**
    - A 1 Hz clock is generated (clk toggles every 0.5 seconds).
    - Global reset is asserted and then released to initialize all counters.
2. **Sequential Mode Testing**
   The testbench covers **all four modes** of the watch in the same order a real user would operate it:
    - Normal Mode
    - Set Time Mode
    - Set Alarm Mode
    - Stopwatch Mode (Elapsed)
    - Stopwatch Mode (Split)
    - Return to Normal Mode
3. **Automatic User-Behavior Simulation**
    - Two tasks (Set_Time_Alarm and StopWatch) emulate the required press sequences of the **mode** and **set** buttons.
    - No manual waveform inspection is needed for inputs—the testbench generates all required patterns automatically.
4. **Functional Output Observation**
    - After each test scenario, the output time digits and stopwatch counters update according to the expected behavior.
    - The alarm time is set and later verified when the current time reaches it.

Test Scenarios Executed in the Testbench

1. Normal Mode Test

- Watch starts in normal mode after reset.
- The clock runs for **300 seconds (5 minutes)**.
- Purpose:
    - Validate minute/hour counting.
    - Verify that when the watch later reaches **02:05**, the alarm will ring.

2. Set Time Mode

- The mode button is clicked once to enter Set Time mode.
- The task Set_Time_Alarm is used to set the time to **02:00**.
- Purpose:
  - Validate digit-by-digit entry (tens hours → units hours → tens minutes → units minutes).
  - Confirm "set" button increments only the selected digit.

3. Set Alarm Mode

- Mode pressed again to reach Set Alarm mode.
- The same task sets the alarm to **02:05**.
- Purpose:
  - Confirm the alarm time is stored correctly.
  - Later verify alarm triggering.

4. Stopwatch Mode — Elapsed Time Test

- Enter Stopwatch Mode using the mode button.
- The task StopWatch performs:
  - Start stopwatch at t=0
  - Stop at 7 seconds
  - Resume at 12 seconds
  - Final stop at 17 seconds
  - Reset at 22 seconds
- Purpose:
  - Verify start / stop / resume / final stop / reset events.
  - Confirm the internal stopwatch counter behaves correctly.

5. Stopwatch Mode — Split Time Test

- Again enter Stopwatch Mode.
- Using the same timings, the sequence now emulates **split** operation:
  - Start → Split → Split Release → Final Stop → Reset
- Purpose:
  - Validate split functionality
  - Ensure the stopwatch freezes displayed time but internal counter continues

6. Back to Normal Mode

- Mode pressed to return to normal mode.
- Clock runs for another **300 seconds**.
- Purpose:

  - Observe alarm activation at the previously set alarm time.
  - Ensure normal time counting resumes after stopwatch usage.

Task Summaries

1. *Set_Time_Alarm* Task

This task simulates the real user behavior for setting time/alarm by pressing the **set** and **mode** buttons.

Function:

- Receives 4 input digits:
  - Tens of hours
  - Units of hours
  - Tens of minutes
  - Units of minutes
- For each digit:
  - Presses **set** repeatedly until the digit reaches the desired value.
  - Presses **mode** once to move to the next digit.
- Includes special handling for digits that need to be set to 0.

Purpose:

- Automates correct digit-cycling behavior.
- Ensures realistic user interaction for time/alarm configuration.

2. *StopWatch* Task

This task emulates start/stop/split operations of the stopwatch.

Inputs:

1. start_time — when to start the stopwatch
2. stop_split_time — when to stop or split
3. resume_splitRelease_time — when to resume or release split
4. final_stop_time — when to perform the final stop
5. reset_time — when to reset the stopwatch counters

Function:

- Waits until the appropriate time, then toggles the **set** button to trigger:
  - Start
  - Stop (or Split)

- Resume (or Split Release)
- Final Stop
- Reset

- Simulates realistic button presses with exact cycle-accurate timing.

Purpose:

- Fully verify stopwatch operation including both elapsed-time and split-time modes.
- Guarantees correct control flow through the stopwatch sub-FSM.

Tested feature	*rst*	*mode*	*set*	Delay (s)	Tens hours out	units hours out	Tens minutes out	units minutes out	Alarm sound	Stopwatch min out	Stopwatch sec out
Initialization (active reset)	0	0	0	Negedge Clk	0	0	0	0	0	0	0
Normal mode	1	0	0	300 s	0	0	0	5	0	0	0
Set time mode	1	1	0	Negedge clk	0	0	0	0	0	0	0
	1	0	1	Negedge clk	0	0	0	0	0	0	0
	1	0	0	Negedge clk	0	1	0	0	0	0	0
	1	0	1	Negedge clk	0	1	0	0	0	0	0
	1	0	0	Negedge clk	0	2	0	0	0	0	0
Set alarm	1	1	0	Negedge clk	0	0	0	0	0	0	0
	1	0	1	Repeat same as before	0	2	0	5	0	0	0
	1	0	0		0	2	0	0	0	0	0
Stopwatch Elapsed time	1	1	0	Negedge clk	0	2	0	0	0	0	0
	1	0	1	7s	0	2	0	0	0	0	7
	1	0	0	Negedge clk	0	2	0	0	0	0	7
	1	0	1	6s	0	2	0	0	0	0	8
	1	0	0	Negedge clk	0	2	0	0	0	0	8
	1	0	1	5s	0	2	0	0	0	0	12

	1	1	0	Negedge clk	0	2	0	0	0	0	12
	1	1	1	5s	0	2	0	0	0	0	12
	1	1	0	Negedge clk	0	2	0	0	0	0	12
Stopwatch split time	1	0	0	Negedge clk	0	2	0	0	0	0	0
	1	0	1	7s	0	2	0	0	0	0	7
	1	0	0	Negedge clk	0	2	0	0	0	0	7
	1	0	1	6s	0	2	0	0	0	0	12
	1	0	0	Negedge clk	0	2	0	0	0	0	12
	1	0	1	5s	0	2	0	0	0	0	17
	1	1	0	Negedge clk	0	2	0	0	0	0	17
	1	0	1	5s	0	2	0	0	0	0	17
Back to normal mode	1	1	0	Negedge clk	0	2	0	0	0	0	17
	1	0	0	300s	0	2	0	5	1	0	17

- **TB code:**

```
// this module is to test the Digital_Watch_Top module
// we are acting as the clock divider which supports 1Hz clock for the
Digital_Watch_Top
`timescale 1s / 1ms
module Digital_Watch_tb;
    // Inputs
    reg clk;
    reg rst;
    reg mode;
    reg set;

    // Outputs
```

```
wire [1:0] tens_hours_out;
wire [3:0] units_hours_out;
wire [2:0] tens_minutes_out;
wire [3:0] units_minutes_out;
wire alarm_sound;
wire [5:0] stopwatch_min_out;
wire [5:0] stopwatch_sec_out;

// Instantiate the Digital_Watch_Top module
Digital_Watch_Top uut (
    .clk(clk),
    .rst(rst),
    .mode(mode),
    .set(set),
    .tens_hours_out(tens_hours_out),
    .units_hours_out(units_hours_out),
    .tens_minutes_out(tens_minutes_out),
    .units_minutes_out(units_minutes_out),
    .alarm_sound(alarm_sound),
    .stopwatch_min_out(stopwatch_min_out),
    .stopwatch_sec_out(stopwatch_sec_out)
);

// Clock generation
initial begin
    clk = 0;
    forever #0.5 clk = ~clk;  // 1 Hz clock (toggle every 0.5 sec)
end

// Test sequence
```

```
    initial begin
        // Initialize inputs
        rst = 0;
        mode = 0;
        set = 0;
        @(negedge clk);
        rst = 1;

        if(tens_hours_out == 0 && units_hours_out ==0 &&
tens_minutes_out==0 && units_minutes_out ==0 &&stopwatch_min_out==0 &&
stopwatch_sec_out==0 )
        $display("Correctly initialized");
        else
        $display ("ERROR");

        // we will test the different modes of the digital watch here
in this sequence
        // Normal Mode -> Set Time -> Set Alarm -> Stopwatch Mode ->
Back to Normal Mode

        //*************************** first case: Normal Mode
***************************//
        mode = 0;
        // Let the clock run to 02:05 to trigger the alarm
        repeat (300) @(negedge clk); // wait for 300 seconds (5
minutes)
```

```
   if (tens_hours_out == 0 && units_hours_out ==0 && tens_minutes_out
== 0 && units_minutes_out == 'd5)
        $display("Correct: Actual time = 0 0:0 5 , Time = %d%d:%d%d",
tens_hours_out, units_hours_out , tens_minutes_out ,
units_minutes_out);
        else
        $display("Error: Actual time = 0 0:0 5 , Time =
%0d%0d:%0d%0d", tens_hours_out,units_hours_out , tens_minutes_out ,
units_minutes_out);


//**************************** second case: Set Time Mode
****************************//
        mode = 1; // move to Set Time Mode
        @(negedge clk);
        mode = 0;
        Set_Time_Alarm(2'd0, 4'd2, 3'd0, 4'd0); // Set time to 02:00

        if (tens_hours_out == 0 && units_hours_out =='d2 &&
tens_minutes_out == 0 && units_minutes_out == 0)
        $display("Correct: Actual time = 0 2:0 0 , Time = %d%d:%d%d",
tens_hours_out, units_hours_out , tens_minutes_out ,
units_minutes_out);
        else
        $display("Error: Actual time = 0 2:0 0 , Time =
%0d%0d:%0d%0d", tens_hours_out,units_hours_out , tens_minutes_out ,
units_minutes_out);
```

```
        //*************************** third case: Set Alarm Mode
***************************//
        mode = 1; // move to Set Alarm Mode
        @(negedge clk);
        mode = 0;
        Set_Time_Alarm(2'd0, 4'd2, 3'd0, 4'd5); // Set alarm to 02:05


        //*************************** fourth case: Stopwatch Mode
(Elapsed Time) ***************************//
        mode = 1; // move to Stopwatch Mode
        @(negedge clk);
        mode = 0;
        StopWatch(0, 7, 12, 17, 22); // start at 0s, stop at 7s,
resume at 12s, final stop at 17s, reset at 22s

        if (stopwatch_min_out == 0 && stopwatch_sec_out == 'd12)
        $display("Correct: Actual stop = 0:12 , Stop time =
%d:%d",stopwatch_min_out , stopwatch_sec_out);
        else
        $display("Error: Actual stop = 0:12 , Stop time = %d:%d at
t=%d",stopwatch_min_out , stopwatch_sec_out , $time);


        //*************************** fifth case: Stopwatch Mode
(Split Time) ***************************//
        mode = 1; // move to Stopwatch Mode (split time)
        @(negedge clk);
        mode = 0;
        StopWatch(0, 7, 12, 17, 22); // start at 0s, split at 7s,
split release at 12s, final stop at 17s, reset at 22s
```

```
        if (stopwatch_min_out == 0 && stopwatch_sec_out == 'd17)
        $display("Correct: Actual stop = 0:17 , Stop time =
%d:%d",stopwatch_min_out , stopwatch_sec_out);
        else
        $display("Error: Actual stop = 0:17 , Stop time = %d:%d at
t=%d",stopwatch_min_out , stopwatch_sec_out , $time);


        //*************************** sixth case: Back to Normal Mode
***************************//
        mode = 1; // move to Normal Mode
        @(negedge clk);
        mode = 0;
        repeat (300) @(negedge clk); // let it run for a while in
normal mode
        // note: we will notice the alarm ringing at 02:05 if we set
the time and alarm correctly
         if (tens_hours_out == 0 && units_hours_out =='d2 &&
tens_minutes_out == 0 && units_minutes_out == 'd5 )
        $display("Correct: Actual time = 0 2:0 5 , ring ime =
%d%d:%d%d " , tens_hours_out, units_hours_out , tens_minutes_out ,
units_minutes_out );
        else
        $display("Error: Actual time = 0 2:0 5 , ring time =
%0d%0d:%0d%0d", tens_hours_out,units_hours_out , tens_minutes_out ,
units_minutes_out );


        // Finish simulation
        $stop;
```

```
    end

    // Set_Time_Alarm Task (takes 4 inputs for each digit) and it
clicks the suitable number of times on set signal to set the time
    task Set_Time_Alarm;
        input [1:0] tens_hours_in;
        input [3:0] units_hours_in;
        input [2:0] tens_minutes_in;
        input [3:0] units_minutes_in;
        integer i;
        begin
            // Set Tens Hours
            for (i = 0; i < tens_hours_in; i = i + 1) begin
                @(negedge clk);
                set = 1;
                @(negedge clk);
                set = 0;
            end
            if (i == 0) @(negedge clk); // to handle the case of
setting to 0
            mode = 1; // move to next digit
            @(negedge clk);
            mode = 0;
            // Set Units Hours
            for (i = 0; i < units_hours_in; i = i + 1) begin
                @(negedge clk);
                set = 1;
                @(negedge clk);
                set = 0;
            end
```

```
            if (i == 0) @(negedge clk); // to handle the case of 
setting to 0
            mode = 1; // move to next digit
            @(negedge clk);
            mode = 0;
            // Set Tens Minutes
            for (i = 0; i < tens_minutes_in; i = i + 1) begin
                @(negedge clk);
                set = 1;
                @(negedge clk);
                set = 0;
            end
            if (i == 0) @(negedge clk); // to handle the case of 
setting to 0
            mode = 1; // move to next digit
            @(negedge clk);
            mode = 0;
            // Set Units Minutes
            for (i = 0; i < units_minutes_in; i = i + 1) begin
                @(negedge clk);
                set = 1;
                @(negedge clk);
                set = 0;
            end
            if (i == 0) @(negedge clk); // to handle the case of 
setting to 0
        end
    endtask

    // StopWatch task should take 5 inputs:
```

```
    // 1- when to start
    // 2- when to stop, split if (split_mode)
    // 3- when to resume, split release if (split_mode)
    // 4- when to stop finally
    // 5- when to reset (clear)
    task StopWatch;
        input integer start_time;
        input integer stop_split_time;
        input integer resume_splitRelease_time;
        input integer final_stop_time;
        input integer reset_time;
        integer t; // time counter
        begin
            for (t = 0; t < start_time; t = t + 1) begin
                @(negedge clk);
            end
            set = 1; // start
            @(negedge clk);
            set = 0;
            for (t = 0; t < (stop_split_time - start_time) -1; t = t +
1) begin
                @(negedge clk);
            end
            set = 1; // stop or split
            @(negedge clk);
            set = 0;
            for (t = 0; t < (resume_splitRelease_time -
stop_split_time) -1; t = t + 1) begin
                @(negedge clk);
            end
```

```
            set = 1; // resume or split release
            @(negedge clk);
            set = 0;
            for (t = 0; t < (final_stop_time -
resume_splitRelease_time) -1; t = t + 1) begin
                @(negedge clk);
            end
            set = 1; // final stop
            @(negedge clk);
            set = 0;
            for (t = 0; t < (reset_time - final_stop_time) -1; t = t +
1) begin
                @(negedge clk);
            end
            set = 1; // reset
            @(negedge clk);
            set = 0;
        end
    endtask
endmodule
```

• Comments:

This testbench verifies all functional modes of the **Digital_Watch_Top** module:

1. **Normal time mode**
2. **Set Time mode**
3. **Set Alarm mode**
4. **Stopwatch elapsed-time mode**
5. **Stopwatch split-time mode**
6. **Return to normal mode**

It also checks:

- Time setting logic
- Alarm setting logic
- Alarm triggering
- Stopwatch start/stop/resume/reset
- Split and split-release behavior

Chapter 5 “Simulation”

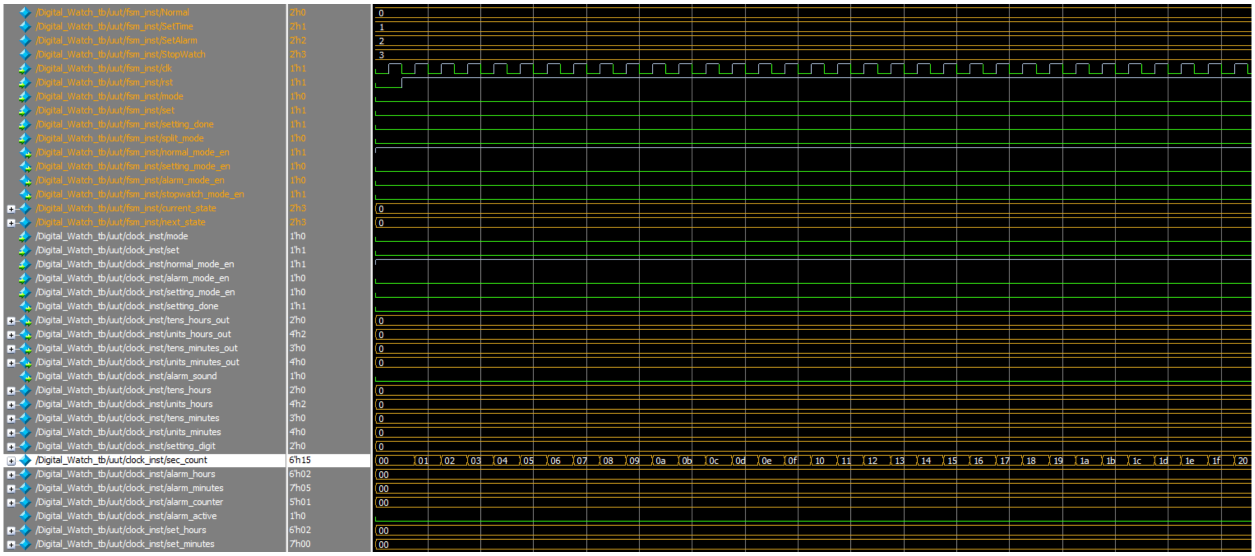
• Simulation waveforms:

In this section we will verify our scenarios discussed in **“Test Scenario”** section in **“Chapter 4”**

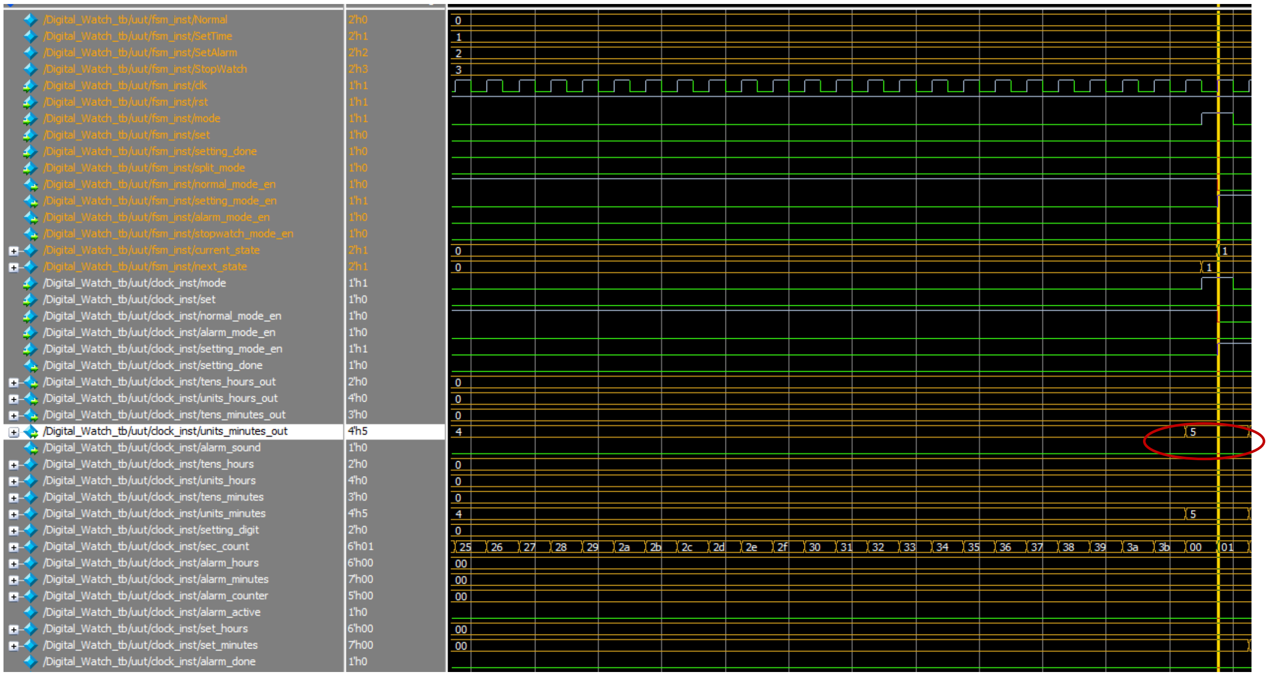
We will verify using snapshots from QuestaSim in the following **Sequence**:

Normal Mode -> Set Time -> Set Alarm -> Stopwatch Mode -> Back to Normal Mode

• 1st Case “Timekeeping mode for 300 seconds (5 Minutes)”:

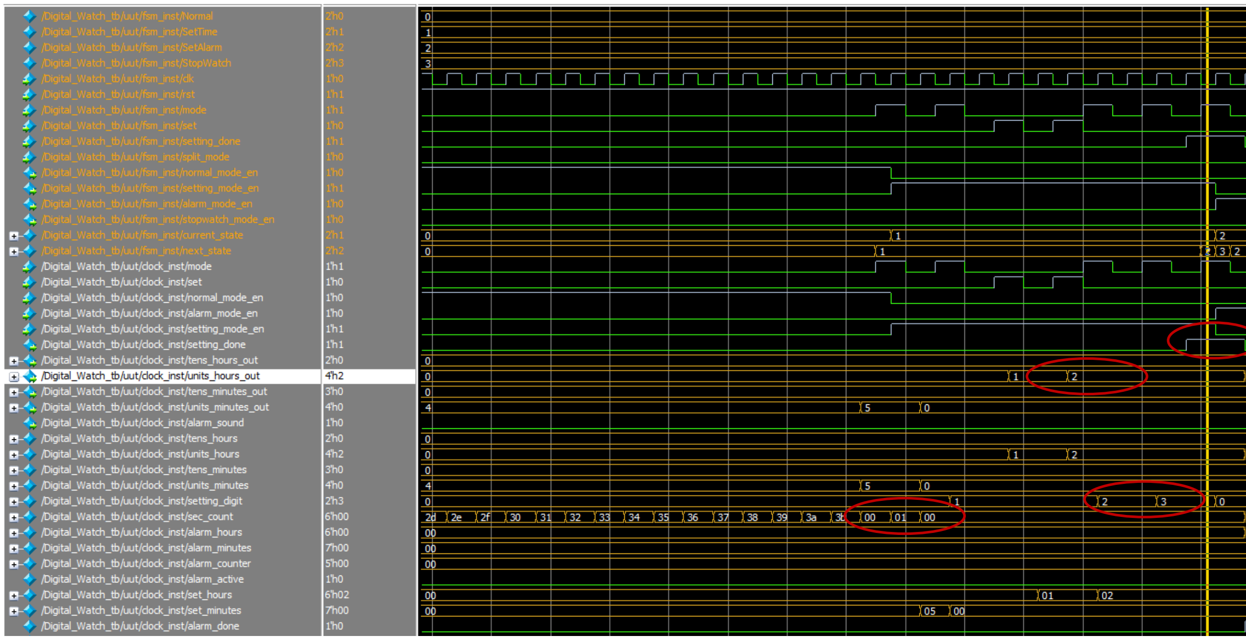


After reset we see the **seconds counter is running** from the beginning.



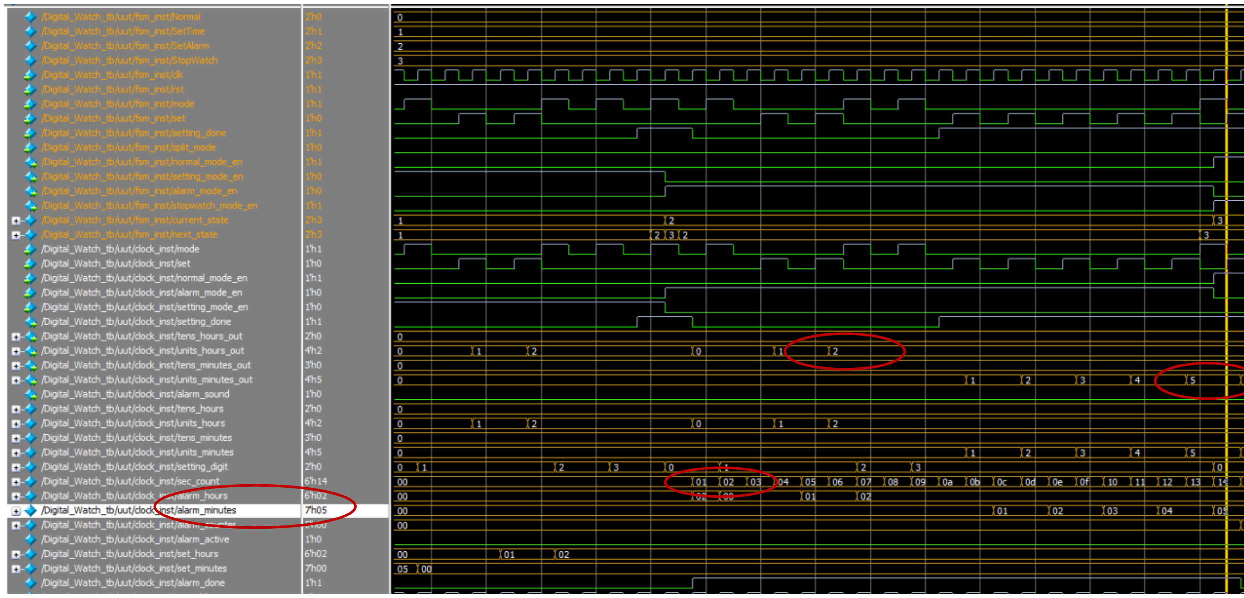
After quit time “300 seconds” we can see that the minutes’ digit **(rightest digit**) had reached 5 minutes **as we expected.**

- ## 2nd Case “Time setting mode (Set the time at 02:00)”:



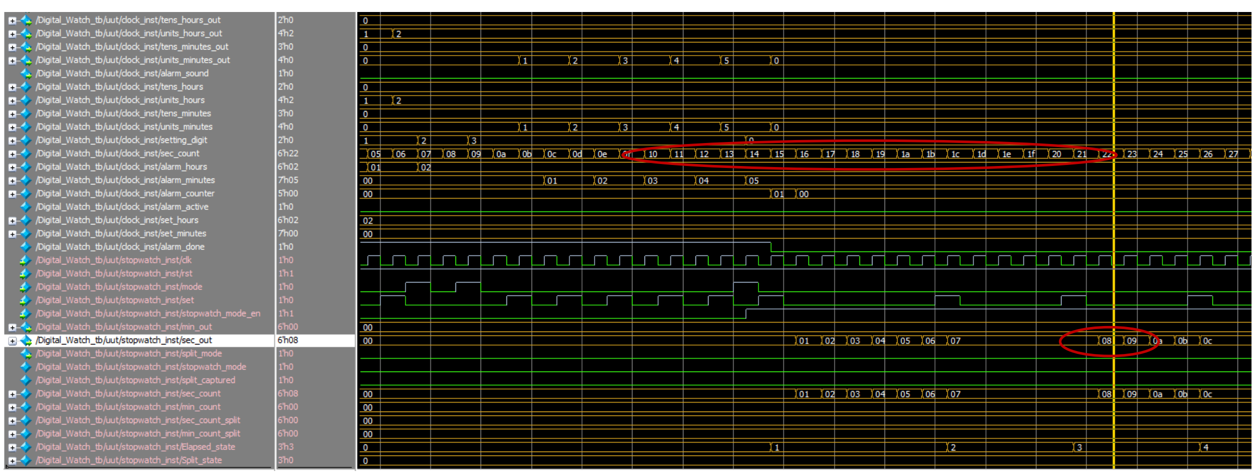
- Notice that the units of hours’ digit (**second digit from left**) is **equal to 2** and the **rest of digit has 0 in their value** **as we expected.**
- Also notice that the **seconds counter has stopped** once we entered the setting time mode as when we are setting the time, **the time is actually being set.**
- After we are done setting the time, the ***setting_done*** flag raises to indicate that the next ***“mode”*** press will take us to the next mode which is the “**setting alarm mode**”.
- Notice how the ***Setting_Digit*** signal is changing with every ***“mode”*** press.

- ## 3^{rd} Case "Alarm setting mode (Set alarm at 02:05)":



- Notice that the units of hours' digit (**second digit from left**) is **equal to 2** and the units of minutes' digit (**rightest digit**) is **equal to 5** and the **rest of digit has 0 in their value** **as we expected.**
- We are saving the alarm time in ***alarm_hours*** and ***alarm_minutes*** signals **as the alarm to sound when the time matches their values** "we will see this case in the **6^{th} scenario**".
- Notice that the **seconds counter starts to increment again** as we are not the **setting time mode.**
- Notice how the ***Setting_Digit*** signal is changing with every ***"mode"*** press.

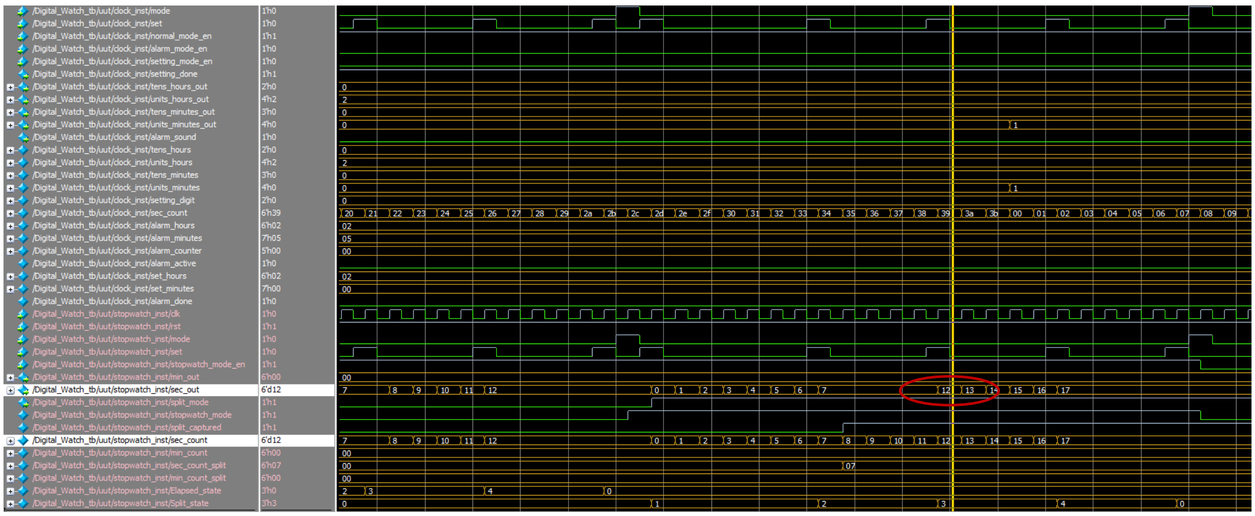
- ## 4^{th} Case "Stopwatch mode (Elapsed time)":



- In elapsed time, when we stop the time and then resume it, **it will continue from where we stopped it before.**
- So in our case we **start** at **"second 0"**, **stop** at **"second 7"**, **resume** at **"second 12"**, **stop again** at **"second 17"**, and **clear** at **"second 22".**

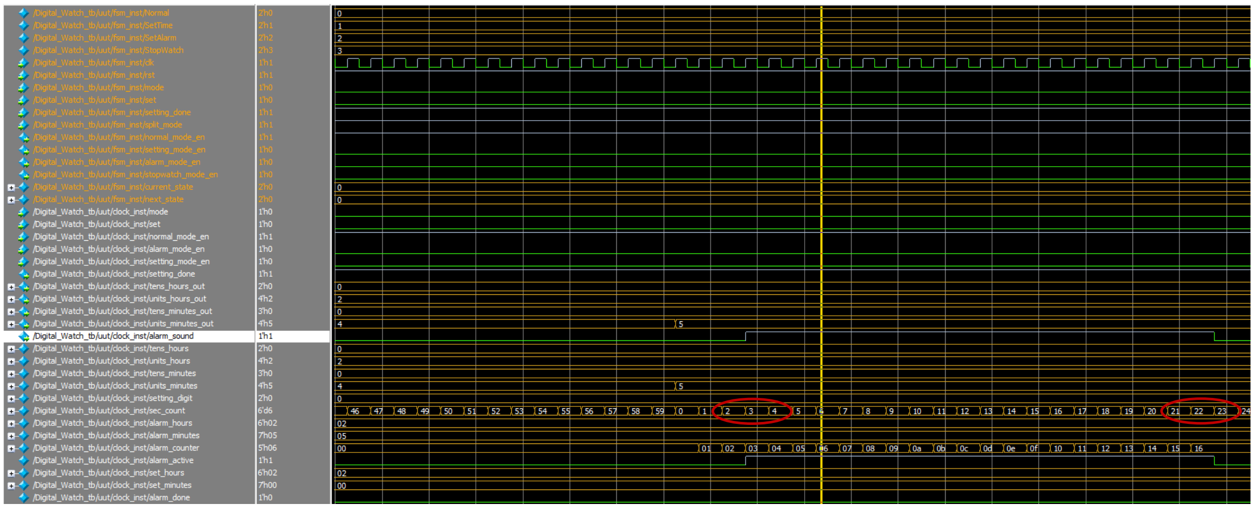
- So this snapshot verifies the **correctness** of out design as even **while we resumed the stopwatch at the second 12** the **stopwatch continued at 8, as we had stopped it at 7.**
- I used ***mode*** signal to **switch modes** in the stopwatch, while I used ***set*** signal to **do the actions in each mode.**
- In stopwatch mode the **normal time shouldn't stop** so, you'll notice the **seconds counter** from the normal clock module is **running** while we are in the stopwatch mode.

• 5th Case "Stopwatch mode (Split time)":

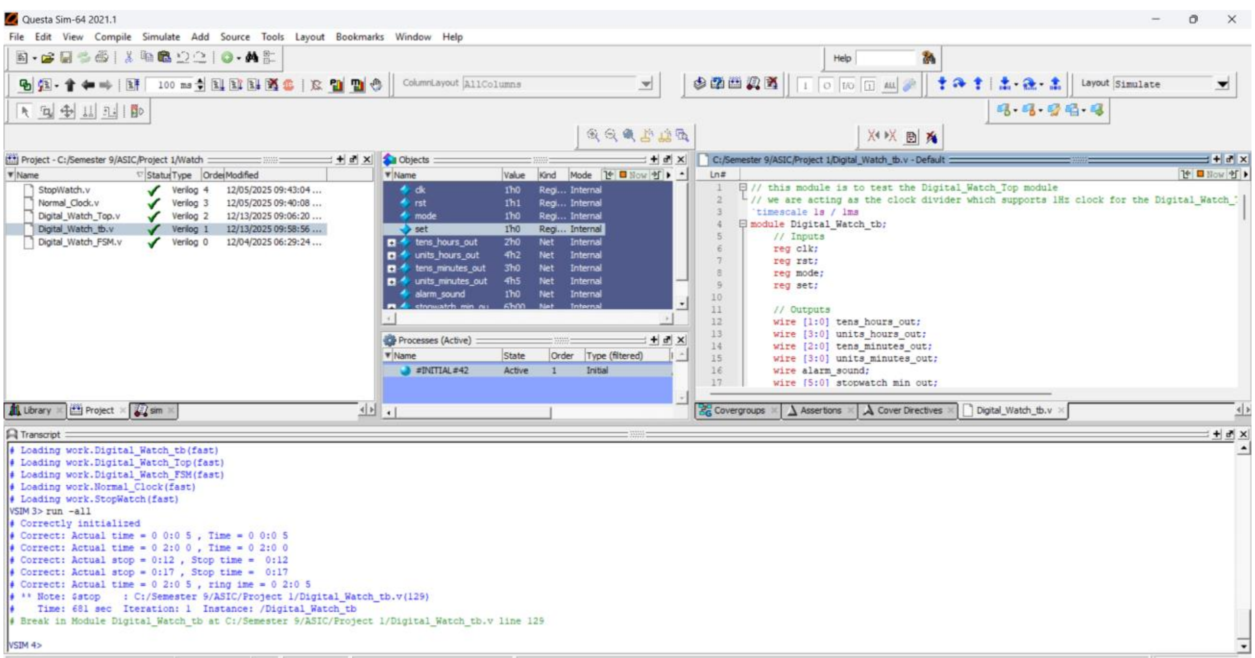


- In split time, when we split the time and then release split it, **it will continue on the actual time we release split at.**
- So in our case we **start** at **"second 0"**, **split** at **"second 7"**, **release split** at **"second 12"**, **stop again** at **"second 17"**, and **clear** at **"second 22".**
- So this snapshot verifies the **correctness** of out design as when we split release at **"second 12"** the displayed time then should be **12** as illustrated by the **red circle.**
- I used ***mode*** signal to **switch modes** in the stopwatch, while I used ***set*** signal to **do the actions in each mode.**
- **Note**: I changed the radix to be **decimal** in this snapshot.
- **seconds counter** is **still running**.

- **6th Case "Back to Normal mode (to check on the Alarm)":**



- After returning to normal mode again the **clock will continue** from the **time we had set it on while we were in the time setting mode** which was **"02:00".**
- Recall that we **had set our alarm at "02:05".**
- So **after 5 minutes** the alarm will **sound** as illustrated by the snapshot.
- The sound will last for **exactly 20 second** as the "**Casio guide**" documented.



• Comments:

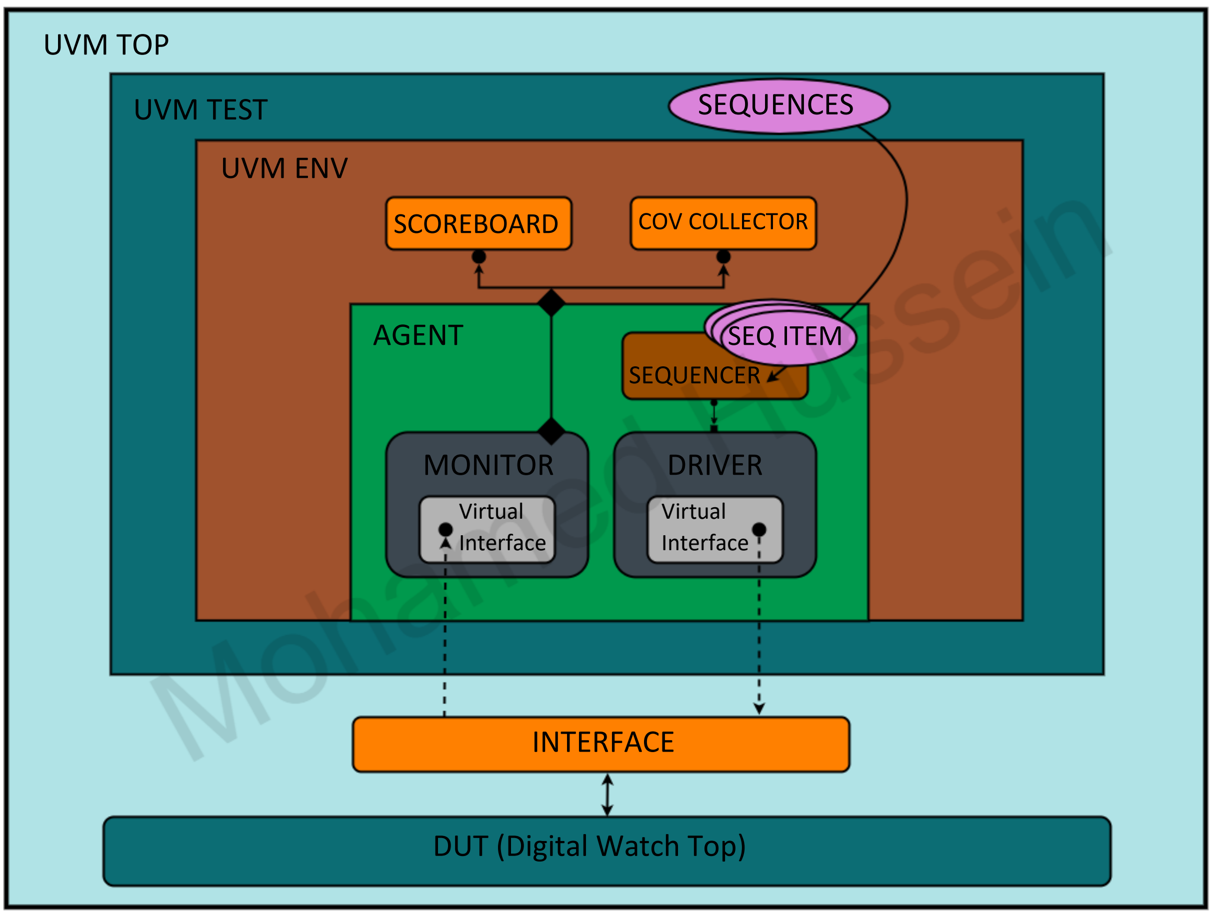
Comments and explanation are under every snapshot

Chapter 6 “UVM Verification Environment”

This chapter describes the Universal Verification Methodology (UVM) environment used to verify the Digital Watch design. The verification strategy combines **constrained-random testing** and **multiple directed sequences** to ensure full functional coverage and correct behavior across all operating modes.

1. UVM Structure

The verification environment follows the standard UVM layered architecture to ensure modularity, reusability, and scalability.



The main components are summarized as follows:

- **UVM Test (`uvm_test`)**
  Controls the overall verification flow. It is responsible for configuring the environment, starting sequences in a specific order, and managing resets between different testing phases.

Here are the sequences ordering from code:

```
task run_phase(uvm_phase phase);
    super.run_phase(phase);
    phase.raise_objection(this);

    // 1. Initial Reset to start the simulation
    `uvm_info("run_phase", "Initial reset asserted", UVM_LOW)
    reset_seq.start(env.agt.sqr);
    `uvm_info("run_phase", "Initial reset deasserted", UVM_LOW)

    // 2. Main Random/Standard Sequence
    `uvm_info("run_phase", "Starting Main Sequence", UVM_LOW)
    main_seq.start(env.agt.sqr);
    `uvm_info("run_phase", "Main Sequence finished", UVM_LOW)

    // 3. SECOND RESET: Prepare for Directed Testing
    // This ensures the FSM starts from IDLE for your directed scenarios
    `uvm_info("run_phase", "Intermediate reset for Directed Tests", UVM_LOW)
    reset_seq.start(env.agt.sqr);
    `uvm_info("run_phase", "Intermediate reset deasserted", UVM_LOW)

    // 4. Executing Directed Sequences in Order
    `uvm_info("run_phase", "Executing Directed Test Battery", UVM_LOW)

    `uvm_info("run_phase", "Starting Directed Seq 1", UVM_LOW)
    directed_1st_seq.start(env.agt.sqr);

    `uvm_info("run_phase", "Starting Directed Seq 2", UVM_LOW)
    directed_2nd_seq.start(env.agt.sqr);

    `uvm_info("run_phase", "Starting Directed Seq 3", UVM_LOW)
    directed_3rd_seq.start(env.agt.sqr);

    `uvm_info("run_phase", "Starting Directed Seq 4", UVM_LOW)
    directed_4th_seq.start(env.agt.sqr);

    `uvm_info("run_phase", "Starting Directed Seq 5", UVM_LOW)
    directed_5th_seq.start(env.agt.sqr);

    `uvm_info("run_phase", "Starting Directed Seq 1", UVM_LOW)
    directed_1st_seq.start(env.agt.sqr);

    phase.drop_objection(this);
endtask
```

- **UVM Environment (`uvm_env`)**
  Acts as the container for all verification components. It instantiates and connects the agent, coverage collector, and scoreboard.
- **UVM Agent (`uvm_agent`)**
  Represents the interface to the DUT and includes:
  - **Sequencer**: Generates sequence items from sequences.
  - **Driver**: Drives stimulus onto the DUT interface.
  - **Monitor**: Observes DUT inputs, outputs, and internal signals (via virtual interfaces) and forwards transactions to coverage and checking components.
- **Sequence Item (`uvm_sequence_item`)**
  Encapsulates one transaction, including control signals such as `rst`, `mode`, and `set`, as well as observed DUT outputs. Constrained randomization is applied at this level.

Here are the constraints I used from code:

```
// constraints
constraint reset_con {
    rst dist {0:/1, 1:/99};
}

constraint mode_con {
    if (!stopwatch_task)
        mode dist {1:/11, 0:/89};
    else
        mode dist {1:/1, 0:/150};  // make mode signal happen less in stopwatch
}

constraint set_con {
    if (!stopwatch_task)
        set dist {1:/75, 0:/25};
    else {
        set dist {1:/15, 0:/150}; // make the set happen less when in stopwatch
        !(set && consecutive); // no set after mode
        !(set && consecutive_set); // no double set in stopwatch
    }
}

// mode and set must not be high together
constraint no_mode_and_set_together {
    !(mode && set);
}

// mode must not be high in two consecutive items
constraint no_consecutive_mode_high {
    !(mode && consecutive);
}
```

- **Coverage Collector**
  Collects functional coverage based on transactions received from the monitor. Coverage includes operating modes, state transitions, and key signal combinations.

Here is the cross coverage I used:

```
// cross coverage
almost_new_day_C: cross cp_tens_hours_out, cp_units_hours_out,
cp_tens_minutes_out, cp_units_minutes_out{ // when the clock is 23:59
    bins day_23_59 = binsof(cp_tens_hours_out) intersect {2} &&
                        binsof(cp_units_hours_out) intersect {3} &&
                        binsof(cp_tens_minutes_out) intersect {5} &&
                        binsof(cp_units_minutes_out) intersect {9};
                        option.cross_auto_bin_max = 0;
} // cross coverage to indicate an end of a day

day_start_day_C: cross cp_tens_hours_out, cp_units_hours_out,
cp_tens_minutes_out, cp_units_minutes_out{ // when the clock is 23:59
    bins day_00_00 = binsof(cp_tens_hours_out) intersect {0} &&
                        binsof(cp_units_hours_out) intersect {0} &&
                        binsof(cp_tens_minutes_out) intersect {0} &&
                        binsof(cp_units_minutes_out) intersect {0};
                        option.cross_auto_bin_max = 0;
} // cross coverage to indicate a start of a day
```

- **Scoreboard**
  Used to check correctness of DUT behavior by comparing expected and actual results.

Here is the report function:

```
// report
function void report_phase(uvm_phase phase);
    int total_success_rand;
    int total_fail_rand;
    real success_rate;

    super.report_phase(phase);

    // Calculate totals
    total_success_rand = normal_correct_count + stopwatch_correct_count;
    total_fail_rand    = normal_error_count + stopwatch_error_count;

    // Calculate Success Rate (Avoid division by zero)
    if ((total_success_rand + total_fail_rand) > 0) begin
```

```
        success_rate = (real'(total_success_rand) / (total_success_rand +
total_fail_rand)) * 100.0;
    end else begin
        success_rate = 0.0;
    end

    // Display formatted summary
    `uvm_info("STIM_SUMMARY", $sformatf("\n\n*** Randomization + Directed
Stimulus Summary ***\ntotal successful transactions: %0d\nSuccess Rate: %0.2f%%",
            total_success_rand, success_rate), UVM_MEDIUM)
endfunction
```

This structure cleanly separates stimulus generation, signal driving, monitoring, and checking, following UVM best practices.

2. Flow of Testing

To achieve **100% functional coverage**, a hybrid verification approach was adopted, combining **large-scale constrained random testing** with **targeted directed sequences**.

2.1 Randomized Testing Phase

The verification process begins with constrained-random stimulus generation:

- A **main random sequence** is executed for **21,000 transactions**
- Randomization explores a wide range of input combinations and state transitions
- Constraints ensure legal stimulus (e.g., preventing illegal signal combinations)
- This phase efficiently exposes corner cases and unintended behaviors

Random testing alone, however, is not sufficient to guarantee coverage of all specific functional modes and corner scenarios. Therefore, directed testing is applied next.

2.2 Directed Testing Phase

After completing the randomized phase, the test transitions to a set of **directed sequences**, executed in a controlled and deterministic order. A reset is applied before directed testing to ensure the DUT starts from a known state.

The sequence execution order from the `uvm_test` run phase is as follows:

1. **Initial Reset**
    - Reset is asserted at the beginning of simulation
    - Ensures the DUT starts from a clean and deterministic state
2. **Main Random Sequence**

- Executes 21,000 constrained-random transactions
- Targets broad functional coverage

```
task body;
    repeat(21000)begin
        seq_item = dw_seq_item::type_id::create("seq_item");
        start_item(seq_item);
        assert(seq_item.randomize());
        finish_item(seq_item);
    end
endtask
```

3. **Second Reset (Pre-Directed Testing)**
    - A second reset is applied
    - Guarantees the internal FSM starts from the IDLE/normal state
    - Prevents state carry-over from random testing into directed scenarios
4. **Directed Sequence Execution**

    The following directed sequences are executed sequentially:

    - **Directed Sequence 1 – Normal Clock Mode**
      Verifies correct time counting behavior in normal clock operation, including hour and minute progression.

```
task body;
    repeat(350)begin
        directed_begin = 1;
        seq_item = dw_seq_item::type_id::create("seq_item");
        start_item(seq_item);
        seq_item.rst = 1;
        seq_item.mode = 0;
        seq_item.set = 0;
        finish_item(seq_item);
    end
endtask
```

- **Directed Sequence 2 – Time Setting Mode**
  Validates the functionality of manual time adjustment, ensuring correct response to control signals and proper value updates.
- **Directed Sequence 3 – Alarm Setting Mode**
  Tests alarm configuration behavior, including alarm time programming and correctness of stored alarm values.
- **Directed Sequence 4 – Stopwatch Elapsed Mode**
  Verifies stopwatch start, stop, and elapsed time counting behavior.

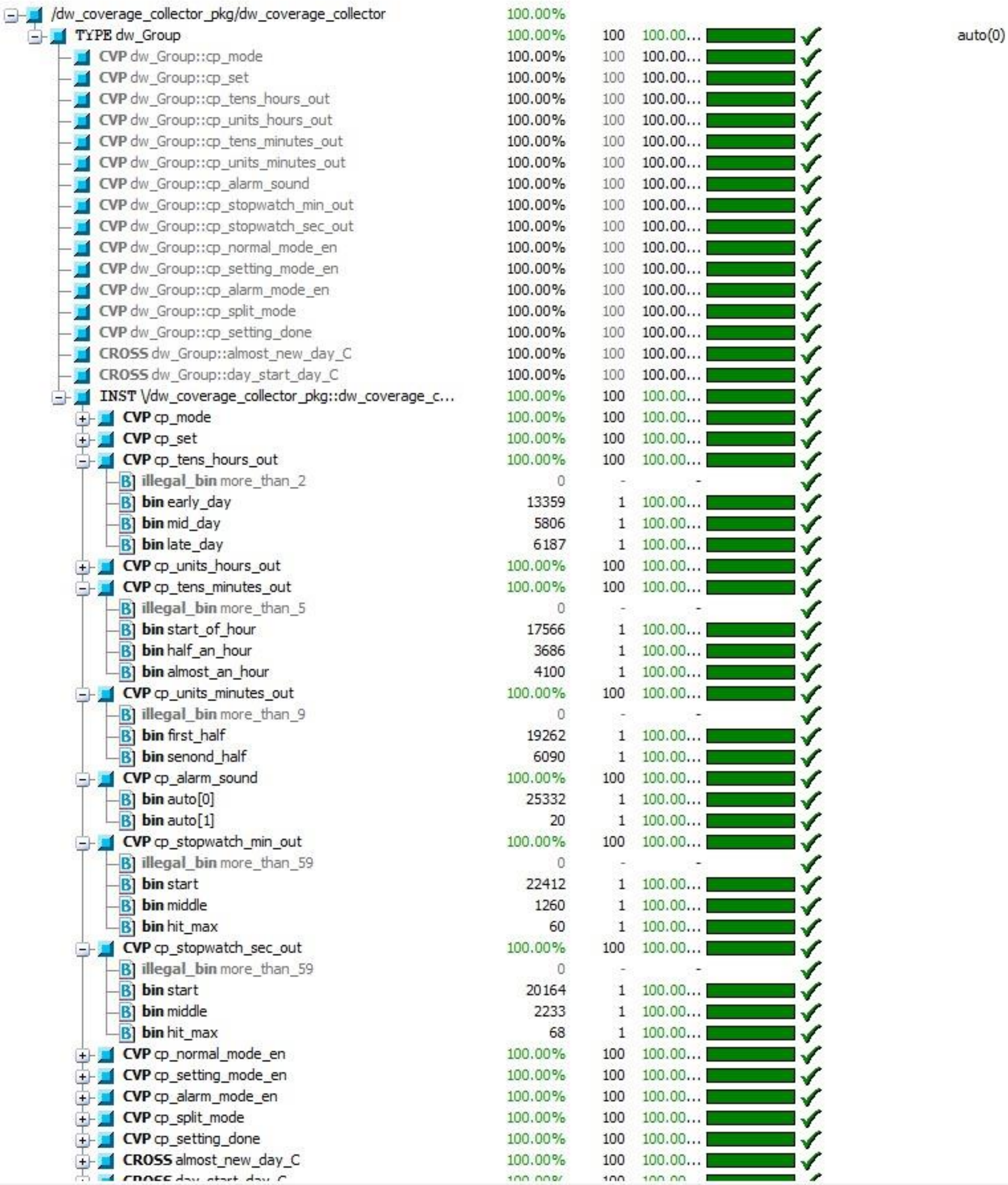
```
task body;
    repeat(4050)begin
        seq_item = dw_seq_item::type_id::create("seq_item");
        start_item(seq_item);
        seq_item.rst = 1;
        seq_item.mode = 0;
        seq_item.set = 0;
        if (z == 0) seq_item.mode = 1; // begin elapsed mode
        else if (z == i) seq_item.set = 1; // start
        else if (z == j) seq_item.set = 1; // stop
        else if (z == k) seq_item.set = 1; // resume
        else if (z == q) seq_item.set = 1; // stop again
        else if (z == v) seq_item.set = 1; // clear
        z++;
        finish_item(seq_item);
    end
endtask
```

- **Directed Sequence 5 – Stopwatch Split Mode**
  Checks the split (lap) functionality of the stopwatch, ensuring time capture without disturbing the main elapsed count.

5. **Final Normal Clock Sequence (Directed Sequence 1 Re-executed)**
   After completing all functional modes, the normal clock sequence is executed again to specifically verify:
   - Correct alarm triggering
   - Proper assertion of the `alarm_sound` signal when the programmed alarm time is reached

3. Coverage Closure Strategy

- **Random testing** provides broad exploration and uncovers unexpected interactions
- **Directed sequences** ensure deterministic coverage of:
  - All operating modes
  - Mode transitions
  - Special conditions such as alarm triggering and stopwatch behavior
- The combination of both approaches successfully achieves **100% functional coverage**



/dw_coverage_collector_pkg/dw_coverage_collector	100.00%			
TYPE dw_Group	100.00%	100	100.00...	auto(0)
CVP dw_Group::cp_mode	100.00%	100	100.00...	
CVP dw_Group::cp_set	100.00%	100	100.00...	
CVP dw_Group::cp_tens_hours_out	100.00%	100	100.00...	
CVP dw_Group::cp_units_hours_out	100.00%	100	100.00...	
CVP dw_Group::cp_tens_minutes_out	100.00%	100	100.00...	
CVP dw_Group::cp_units_minutes_out	100.00%	100	100.00...	
CVP dw_Group::cp_alarm_sound	100.00%	100	100.00...	
CVP dw_Group::cp_stopwatch_min_out	100.00%	100	100.00...	
CVP dw_Group::cp_stopwatch_sec_out	100.00%	100	100.00...	
CVP dw_Group::cp_normal_mode_en	100.00%	100	100.00...	
CVP dw_Group::cp_setting_mode_en	100.00%	100	100.00...	
CVP dw_Group::cp_alarm_mode_en	100.00%	100	100.00...	
CVP dw_Group::cp_split_mode	100.00%	100	100.00...	
CVP dw_Group::cp_setting_done	100.00%	100	100.00...	
CROSS dw_Group::almost_new_day_C	100.00%	100	100.00...	
CROSS dw_Group::day_start_day_C	100.00%	100	100.00...	
INST \/dw_coverage_collector_pkg::dw_coverage_c...	100.00%	100	100.00...	
CVP cp_mode	100.00%	100	100.00...	
CVP cp_set	100.00%	100	100.00...	
CVP cp_tens_hours_out	100.00%	100	100.00...	
illegal_bin more_than_2	0	-	-	
bin early_day	13359	1	100.00...	
bin mid_day	5806	1	100.00...	
bin late_day	6187	1	100.00...	
CVP cp_units_hours_out	100.00%	100	100.00...	
CVP cp_tens_minutes_out	100.00%	100	100.00...	
illegal_bin more_than_5	0	-	-	
bin start_of_hour	17566	1	100.00...	
bin half_an_hour	3686	1	100.00...	
bin almost_an_hour	4100	1	100.00...	
CVP cp_units_minutes_out	100.00%	100	100.00...	
illegal_bin more_than_9	0	-	-	
bin first_half	19262	1	100.00...	
bin senond_half	6090	1	100.00...	
CVP cp_alarm_sound	100.00%	100	100.00...	
bin auto[0]	25332	1	100.00...	
bin auto[1]	20	1	100.00...	
CVP cp_stopwatch_min_out	100.00%	100	100.00...	
illegal_bin more_than_59	0	-	-	
bin start	22412	1	100.00...	
bin middle	1260	1	100.00...	
bin hit_max	60	1	100.00...	
CVP cp_stopwatch_sec_out	100.00%	100	100.00...	
illegal_bin more_than_59	0	-	-	
bin start	20164	1	100.00...	
bin middle	2233	1	100.00...	
bin hit_max	68	1	100.00...	
CVP cp_normal_mode_en	100.00%	100	100.00...	
CVP cp_setting_mode_en	100.00%	100	100.00...	
CVP cp_alarm_mode_en	100.00%	100	100.00...	
CVP cp_split_mode	100.00%	100	100.00...	
CVP cp_setting_done	100.00%	100	100.00...	
CROSS almost_new_day_C	100.00%	100	100.00...	
CROSS day_start_day_C	100.00%	100	100.00...	

This structured flow ensures that all critical functionality of the Digital Watch design is thoroughly verified under both random and controlled scenarios.

3. Simulation Transcript and Waveform Analysis

To validate the correctness and completeness of the verification process, simulation **transcripts** and **waveform snapshots** were analyzed. These artifacts provide concrete evidence of successful test execution, error-free operation, and correct temporal behavior of the Digital Watch design.

3.1 Simulation Transcript Summary

The simulation transcript summarizes the overall verification results and confirms that all test phases executed successfully without any runtime issues.

```
# *** Randomization + Directed Stimulus Summary ***
# total successful transactions: 52704
# Success Rate: 100.00%
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   18
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [STIM_SUMMARY]     1
# [TEST_DONE]     1
# [run_phase]    13
```

Key observations from the transcript include:

- **Total successful transactions**: 52,704
- **Success rate**: 100.00%
- **No simulation errors or warnings were reported**

3.2 Waveform Snapshot Analysis

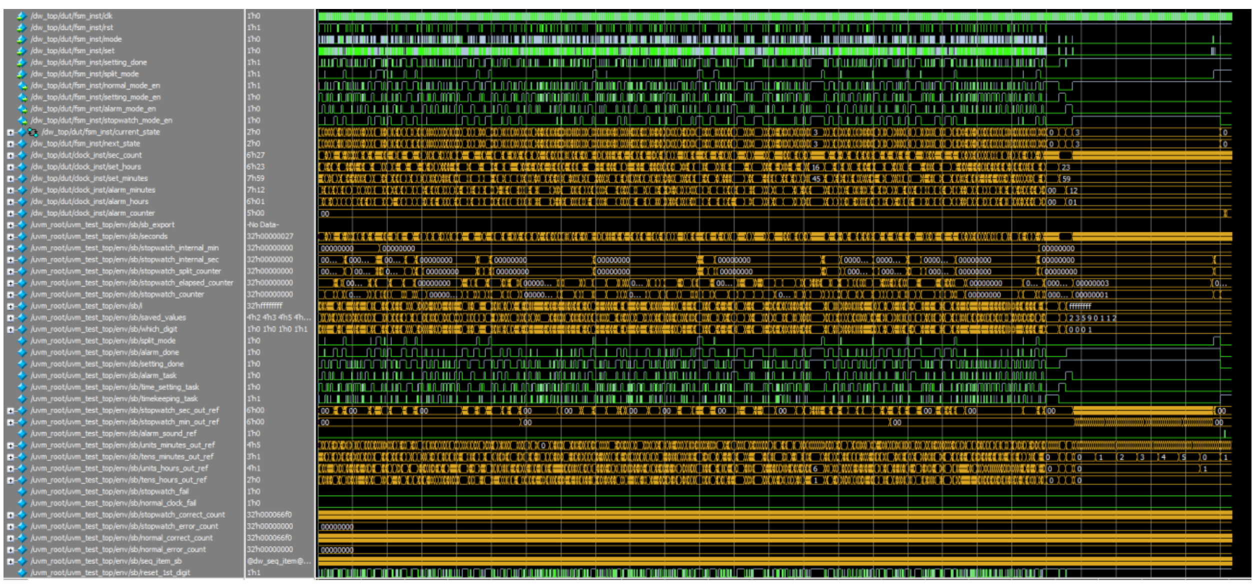
Waveform snapshots were captured during simulation to visually inspect and validate DUT behavior over time. These waveforms serve as a timing-level confirmation of the functional correctness observed in the transcript.

The waveform analysis confirms:

- **Correct reset behavior**
  - All outputs initialize to known states upon reset assertion
  - FSM returns to the normal clock state after reset deassertion

- **Mode transitions**
  - Proper transitions between:
    - Normal clock mode
    - Time setting mode
    - Alarm setting mode
    - Stopwatch elapsed mode
    - Stopwatch split mode
  - No illegal or unexpected state transitions observed
- **Clock and timing accuracy**
  - Time increments correctly in normal clock mode
  - Stopwatch minutes and seconds increment as expected
  - Split mode correctly freezes displayed values while internal counting continues
- **Alarm functionality**
  - Alarm signal (alarm_sound) asserts precisely when the current time matches the programmed alarm time
  - Alarm behavior remains correct after returning from other modes

Full waveform:



Waveform inspection complements the transcript results by confirming that signal transitions occur at the correct clock edges and follow the intended design timing, also we will notice **neither error counters in the scoreboard have been incremented** which proves the correctness of the design.

Repository and Reproducibility

To ensure reproducibility and facilitate further development, the complete design and verification environment for the Digital Watch project have been made publicly available on GitHub.

The repository includes:

- **Architecture**
- **RTL design files for the Digital Watch system**
- **Documentation**
- **UVM-based verification environment**
- **Vivado Documentation**
  Contains synthesis, elaboration, and implementation snapshots generated using Vivado, along with tool messages.

This allows reviewers and future developers to:

- Reproduce all simulation results
- Inspect the UVM architecture and test flow
- Extend the design or verification environment easily

GitHub Repository:
MohamedHussein27/Digital_Watch_Design_and_UVM_Verification