Project Report

Digital Clock with Stopwatch

Mohammad Fares Aljamous 1088672

Omar Mohammad 1088546

Hadi Albanna 1088677

Supervised By: Eng. Gasm and Dr. Sajid

جامعة أبوظبي
ABU DHABI UNIVERSITY

Submitted: December 2, 2024

Contents

List of Figures

List of Tables

Abstract

This article describes the design and implementation of a digital clock/stopwatch on an FPGA, demonstrating its potential to support real-time digital systems. The goal was to design a clock that accurately shows hours, minutes, and seconds while leveraging the parallel processing capability of FPGA technology. The system was built with hardware description language (HDL) to implement essential capabilities such as a frequency divider for timekeeping, a counter system, and a seven-segment display driver for visual output. Testing on FPGA hardware revealed perfect timing and consistent performance, verifying the design's efficiency and precision. The research illustrates the usefulness of FPGAs for flexible and low-latency performance designs, with the possibility of future expansions such as the addition of warning functionalities or user interfaces for customization. This project helped us realize the power of the FPGA as a tool that can be used in many different industries.

1 Introduction

The use of a digital clock in an FPGA exemplifies the practical use of digital design and embedded systems. Digital clocks require perfect synchronization, efficient hardware utilization, and reliable visual output, making them an ideal assignment for testing FPGA capabilities. This project focuses on creating a clock that displays hours, minutes, and seconds by using a frequency divider for timekeeping, logic counters, and a seven-segment display driver. Unlike typical microcontroller-based designs, the FPGA offers high-speed performance and versatility, making it ideal for real-time applications. The goal is to demonstrate the effectiveness of FPGA-based systems in real-time operations and investigate their potential for future improvements, such as alerts or user-controlled features.

1.1 Motivation

The idea of developing a digital clock/stopwatch on an FPGA comes from an urgent need for a precise, adaptable, and efficient means to keep track of time. Using FPGA technology, we can create a clock that is both exact and programmable, with features such as alarms, various time zones, and even synchronization with other devices. This study demonstrates how hardware design can address real-world problems by demonstrating that FPGAs can provide faster and more efficient solutions than standard microcontrollers. A clock like this might be beneficial in smart homes, IoT devices, and even industrial systems, contributing to the development of smarter, more connected, and energy-efficient technology.

1.2 Problem Statement

The issue we want to address is the lack of a flexible, efficient, and high-precision digital clock/stopwatch system suited for modern applications. Traditional digital clocks and stopwatches frequently lack the flexibility, scalability, and integration capabilities necessary for smart homes, IoT devices, and advanced industrial systems. Furthermore, many existing solutions are based on set hardware or microcontrollers, which might limit performance and flexibility. We solve these restrictions by developing a digital clock/stopwatch on an FPGA, resulting in a solution that is not only accurate and dependable but also capable of incorporating sophisticated features and supporting

a wide range of use cases. This ensures interoperability with new technology while yet preserving efficiency and precision.

1.3 Literature Review

Existing digital clock systems are typically built using microcontrollers, application-specific integrated circuits (ASICs), or software-based approaches, each with its advantages and limitations. Microcontroller-based clocks are affordable and straightforward but lack the flexibility to be upgraded or expanded. Software-based clocks, while adaptable and easy to update, are often less energy-efficient and may introduce latency. On the other hand, ASICs offer high precision and efficiency but are complex and costly to design. FPGA-based digital clocks combine the advantages of hardware precision and reprogrammability, offering a high degree of flexibility and customization. However, depending on the implementation, they may require specialized expertise and could be more energy-intensive and expensive [1].

In counter-based digital clocks, incremental counters are used to keep track of time. Typically, a clock signal drives a counter that updates each second, with separate counters for seconds, minutes, and hours. The FPGA or microcontroller handles the logic for carry-over when the minutes or hours exceed their respective limits (60 for minutes and hours, or 12/24 for hours depending on the clock format) [2].

Real-time clock (RTC) modules, such as the DS3231 and DS1307, are standalone circuits that use a crystal oscillator to maintain accurate time. These modules communicate with the main controller, usually through I2C or SPI protocols, and feature integrated functions like time storage, alarms, and temperature compensation [2, 3].

For FPGA-based digital clocks, the clock signal is typically generated by the FPGA's internal oscillator or an external crystal oscillator. With the FPGA's resources, additional logic such as phase-locked loops (PLLs) and frequency dividers can be incorporated to provide highly accurate clock signals that drive the timekeeping counters [1].

2 Design

To address the project constraints, we began by brainstorming and organizing our ideas into a well-structured plan. Afterward, we sought feedback from

Eng. Gasm to refine these concepts and ensure that the design is aligned with the required specifications. Our first step was to create the basic structure of the digital clock, including counters for seconds, minutes, and hours. We tested the counters thoroughly and explored different VHDL implementations to determine the most efficient method for handling time increments and managing overflow conditions. Once the core clock functionality was established, we focused on adding the stopwatch feature. This involved designing the reset, start, and stop functions, which required careful attention to state transitions and handling user inputs. We then integrated the display system, including the 7-segment display and the logic for converting binary values to Binary-Coded Decimal (BCD) for accurate output. Throughout the design process, we followed logical design principles and ensured proper timing constraints were maintained, enabling smooth interaction between all functional blocks. Using VHDL, we translated the design into code, applying the techniques and knowledge we had gained during our coursework. Finally, we synthesized the design and tested it on the FPGA, making necessary refinements to ensure the system functioned correctly in real-time.

2.1 Requirements, Constraints, and Considerations

- Objective: The goal of this project was to design and implement a digital clock with stopwatch functionality on an FPGA. The system displays real-time hours, minutes, and seconds on a 7-segment display and supports stopwatch operations like start, stop, and reset. It includes overflow logic to handle 60 seconds incrementing minutes and 60 minutes incrementing hours. Developed using VHDL, the design efficiently manages time-keeping and display tasks while processing user inputs. By leveraging the FPGA's parallel processing capabilities, the system ensures real-time performance and seamless updates for both clock and stopwatch features.

- Constraints:

    1. **Input Data and Control Mechanism:** The system's control inputs are managed through switches and push buttons on the FPGA board. These inputs allow users to toggle between displaying seconds with minutes or minutes with hours on the digital clock and to switch to the stopwatch mode. Stopwatch functionality is controlled using push buttons—one for starting and stopping

the stopwatch and another for resetting it. Additionally, users can adjust the clock's time via these inputs. The system is designed to process these inputs without delay, ensuring responsive user interaction. Handling these manual inputs introduces design challenges, such as addressing switch debouncing and ensuring smooth transitions between clock and stopwatch modes. Accurate mapping of inputs is essential to enable seamless functionality.

2. **Time Management Logic:** The digital clock's functionality is centered around counters that increment every second, with overflow signals cascading to update the minute and hour counters. Each counter is designed to operate efficiently, ensuring smooth functionality. Once 60 seconds are reached, the minute counter increases by one, and similarly, after 60 minutes, the hour counter advances. The design must handle carry-over signals between counters to avoid errors in the time display caused by overflow issues. Maintaining accurate and reliable timekeeping over extended periods is critical to minimize drift and ensure consistent performance.
3. **Stopwatch Functionality:** A key aspect of this project is the independent operation of the stopwatch from the digital clock. The stopwatch should enable users to start, pause, and reset the timer without interrupting the digital clock, which continues to display the current time. The design must handle various stopwatch states, including running, pausing, and reset, ensuring the clock's functionality remains unaffected. Additionally, the stopwatch should be able to resume timing from where it was paused, preserving the elapsed time without restarting.
4. **Display Specifications:** The system utilizes a 7-segment display to present the time for both the digital clock and stopwatch. One of the key challenges involves converting the internal binary time values into easily readable decimal digits for display. This requires implementing a Binary Coded Decimal (BCD) to a 7-segment converter. The display must update continuously in real-time, accurately reflecting both the clock's current time and the stopwatch's progress. The conversion logic must ensure that the displayed digits are clear and free of errors or overlaps. For example, when the timer reaches 59 seconds, the display should correctly show "59"

and then seamlessly reset to "00" upon overflow.

5. **System Implementation and Real-Time Operation:** The entire system, including time-keeping and stopwatch functionalities, must be implemented on an FPGA, specifically the Cyclone V board. This requires synthesizing the VHDL code to handle time data, manage user inputs, and control the output displays. The FPGA's ability to execute multiple tasks in parallel ensures that the clock's counters are updated every second, and inputs are processed without delay. Real-time operation is vital to maintain the accuracy of both the clock and stopwatch. The VHDL code will exploit the FPGA's parallel processing capabilities to ensure smooth, uninterrupted operation.

- **Considerations:**

1. **User Interface:** The user interface design is vital to provide an intuitive way to control both the digital clock and the stopwatch. The system must ensure that the switches are easy to use and that the current mode of the system is clearly indicated. Switch debouncing must be handled properly, and the user should receive clear feedback on the status of the stopwatch or digital clock through the seven-segment display.
2. **Accuracy and Reliability:** The system must ensure the digital clock maintains accurate time, and the stopwatch operates without errors. The FPGA design supports reliable time-keeping by updating the clock in fixed intervals, which prevents the inaccuracies typically seen with software-based timers. The real-time processing ability of the FPGA ensures both the clock and stopwatch function simultaneously and without drift.
3. **Scalability and Future Enhancements:** While the initial design focuses on basic time-keeping and stopwatch functions, future versions of the system can include additional features such as an alarm, the ability to customize time formats (12-hour vs. 24-hour), or the addition of multiple stopwatch channels. The system should be designed with scalability in mind, allowing for easy upgrades and additions without significant changes to the overall architecture.

2.2 Design Process

We followed a structured process to design and implement the digital clock and stopwatch on an FPGA. The first step involved creating timekeeping counters for the clock, managing seconds, minutes, and hours, and properly handling overflow conditions. Next, we developed the stopwatch module, including a state machine to handle running, pausing, and resetting states independently from the clock. Both modules were implemented in VHDL to ensure accuracy and reliability. The display logic was then designed, starting with a Binary to BCD conversion, followed by BCD to 7-segment display logic to present time in a readable format. Control logic was added to process inputs from switches and buttons for toggling modes, adjusting the clock, and operating the stopwatch. Finally, all components were integrated, rigorously tested, and optimized to achieve a functional, real-time system that demonstrates the capabilities of VHDL programming and FPGA technology.

2.2.1 Basic Components: Counter, and Comparator:

The design of the digital clock/stopwatch system relies on two fundamental components: the counter, and comparator. These components form the foundation of the system, enabling efficient time-keeping, and logical decision-making. Each component was implemented in VHDL to ensure modularity, accuracy, and reliable functionality.

1. **Counter:** The counter serves as the backbone for time progression in both the digital clock and stopwatch functionalities. It increments its output value on each clock cycle and resets when the reset signal is activated. This component is used to manage time units such as seconds, minutes, and hours. Below is the VHDL code for the counter:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY accu IS
    GENERIC (
        n: integer
    );
    PORT(
```

```
        reset, clk: IN std_logic;
        output: BUFFER std_logic_vector(n-1 DOWNTO 0)
    );
END ENTITY;

architecture behv OF accu IS
BEGIN
    process(clk, reset)
    BEGIN
        IF reset = '1' THEN
            output <= (OTHERS => '0');
        ELSIF rising_edge(clk) THEN
            output <= output + 1;
        END IF;
    END process;
END architecture;
```

The counter is essential for accurate time-keeping and overflow handling, ensuring seamless transitions between time units.

2. **Comparator:** The comparator is responsible for evaluating two input values and determining their equality. This logic is crucial for detecting conditions such as counter overflow or user-triggered resets. When the two inputs match, the comparator outputs a high signal (1); otherwise, it outputs a low signal (0). The VHDL implementation is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY comparator IS
    GENERIC (
        n: integer
    );
    PORT(
        c1, c2: IN std_logic_vector(n-1 DOWNTO 0);
        output: OUT std_logic
    );
END ENTITY;
```

```
15
16 architecture behv OF comparator IS
17 BEGIN
18     process(c1, c2)
19     BEGIN
20         IF c1 = c2 THEN
21             output <= '1';
22         ELSE
23             output <= '0';
24         END IF;
25     END process;
26 END architecture;
```

The comparator is utilized to monitor conditions such as counter limits, enabling precise control of state transitions within the system.

3. **Integration of Components** The counter, and comparator work together to ensure the seamless operation of the digital clock and stopwatch. The counter tracks time increments, and the comparator manages critical decision-making processes. Together, these components provide a robust foundation for the system, enabling reliable, real-time performance and user interaction.

2.2.2 Digital clock Counters:

- The development of the time-keeping counters for the digital clock and stopwatch aimed to ensure accurate real-time operation, smooth transitions between time units, and reliable user interaction. These counters are central to the system, managing the sequential progression of seconds, minutes, and hours while effectively integrating with user inputs and display components.

- The design revolves around the principle of incremental time tracking, with each counter dedicated to a specific time unit, seconds, minutes, or hours. These counters increase their value based on clock pulses and reset upon reaching their maximum capacity, triggering the subsequent counter in the sequence. For example, the seconds counter increments with each clock pulse and resets to zero after reaching 60, signaling the minutes counter to increment by one. Likewise, the minutes counter

resets at 60 and prompts the hours counter to increase. This cascading configuration ensures consistent and uninterrupted timekeeping. Managing overflow is a crucial aspect of the design. Comparators are used to monitor the values of each counter, generating a reset signal when a counter reaches its limit (59 seconds). This reset signal also carries forward to the next counter, enabling smooth and continuous transitions between time units. For instance, when the seconds counter resets to zero, the carry-out signal triggers the minutes counter to increment, ensuring a reliable and error-free time progression.

- The block diagram which is shown in Figure 1 represents the architecture of a digital clock system that accurately tracks and displays time in real time using cascading counters for seconds, minutes, and hours. The process begins with a high-frequency clock signal (50 MHz) that is divided down to a 1 Hz signal using a frequency divider. This 1 Hz clock drives the seconds counter, which increments every second. When the seconds counter reaches 60, a comparator detects the overflow, resets the counter to zero, and sends a carry-out signal to increment the minutes counter. This cascading system continues as the minutes counter increments every 60 seconds and resets upon reaching 60 minutes, triggering the hours counter to increment. The hours counter operates in a 24-hour format, resetting to zero after reaching 24 hours. To ensure human-readable output, the binary values from the counters are converted into Binary-Coded Decimal (BCD) format using Binary to BCD converters. The BCD values are then processed by BCD to 7-segment converters, which generate signals to drive 7-segment displays for each time unit: seconds, minutes, and hours. The cascading nature of the counters, along with comparators for overflow management and conversion logic for display, ensures seamless and accurate timekeeping. This modular design enables real-time updates while maintaining synchronization between time units, providing an efficient and reliable solution for digital clock functionality.

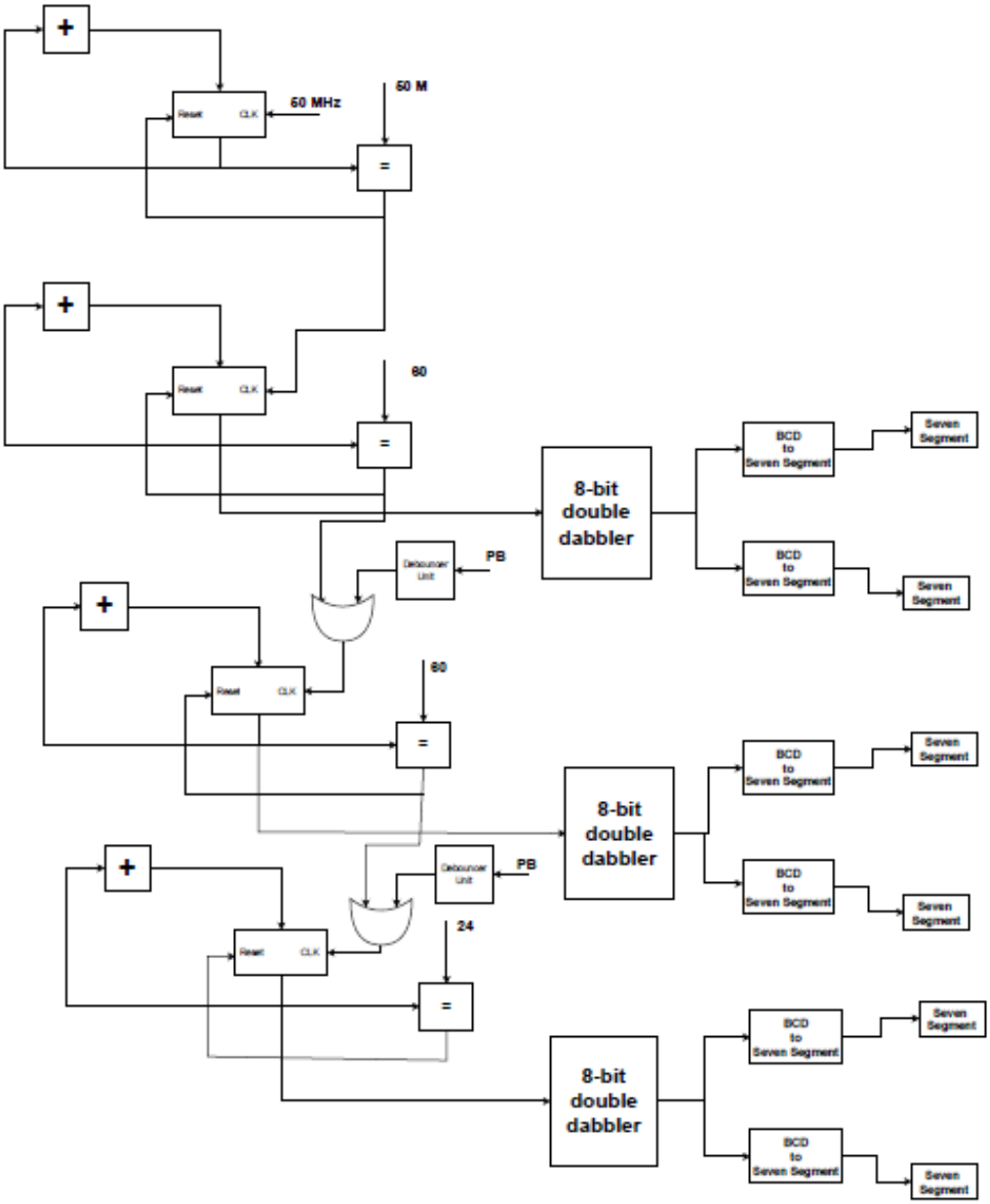


Figure 1: Digital Clock Diagram.

- The VHDL code provided implements the core functionality of the digital clock, involving key components such as comparators, accumulators (counters), debouncers, binary-to-BCD converters, and BCD-to-7-segment converters. Each component has a specific function that ensures smooth and accurate timekeeping. The comparators (comp1, comp2, comp3, and comp4) monitor the counter values for seconds, minutes, and hours. When a counter reaches its maximum value, a comparator triggers a reset and sends a carry-out signal to the next counter.

  The accumulators (acc1, acc2, acc3, and acc4) function as the counters for number of clock pulses, seconds, minutes, and hours. Each accumulator increments its value with every clock cycle. The reset signals generated by the comparators control the counters, ensuring they increment correctly.

  Debouncers (mdebounce and hdebounce) handle user input by stabilizing the signals from push buttons. These components ensure that each button press (e.g., for incrementing minutes or hours) is registered only once, even if the button creates multiple unintended signals due to mechanical bouncing.

  The binary-to-BCD converters (btobcd1, btobcd2, btobcd3) convert the binary outputs of the counters into Binary-Coded Decimal (BCD), which is necessary for accurate display. These BCD values are then passed to the BCD-to-7-segment converters (bcdtoseven1 to bcdtoseven6), which drive the 7-segment displays to show the current time for seconds, minutes, and hours.

  This modular design ensures that the digital clock operates efficiently, with precise time increments and a clear, human-readable display.

```
1     -- Digital Clock
2  comp1: comparator generic map (n => 26) port map (c1 => f1,
       c2 => c1, output => o1);
3  comp2: comparator generic map (n => 8) port map (c1 => f2, c2
       => c2, output => o2);
4  comp3: comparator generic map (n => 8) port map (c1 => f3, c2
       => c3, output => o3);
5  comp4: comparator generic map (n => 8) port map (c1 => f4, c2
       => c4, output => o4);
```

```
6  mdebounce: debouncer port map(clk => clock, pb => incm,
       output => mpb);
7  hdebounce: debouncer port map(clk => clock, pb => inch,
       output => hpb);
8
9  acc1: accu generic map (n => 26) port map (reset => o1, clk
       => clock, output => f1);
10 acc2: accu generic map (n => 8) port map (reset => o2, clk =>
       o1, output => f2);
11 acc3: accu generic map (n => 8) port map (reset => o3, clk =>
       (o2 or mpb), output => f3);
12 acc4: accu generic map (n => 8) port map (reset => o4, clk =>
       (o3 or hpb), output => f4);
13 btobcd1: binaryToBCD port map(input => f2, d1 => bcd1, d2 =>
       bcd2);
14 btobcd2: binaryToBCD port map(input => f3, d1 => bcd3, d2 =>
       bcd4);
15 btobcd3: binaryToBCD port map(input => f4, d1 => bcd5, d2 =>
       bcd6);
16 bcdtoseven1: bcdtoseven port map(input => bcd1, output =>
       seven1);
17 bcdtoseven2: bcdtoseven port map(input => bcd2, output =>
       seven2);
18 bcdtoseven3: bcdtoseven port map(input => bcd3, output =>
       seven3);
19 bcdtoseven4: bcdtoseven port map(input => bcd4, output =>
       seven4);
20 bcdtoseven5: bcdtoseven port map(input => bcd5, output =>
       seven5);
21 bcdtoseven6: bcdtoseven port map(input => bcd6, output =>
       seven6);
```

2.2.3 Stopwatch Design:

- The design process for the stopwatch system aimed at providing accurate time-tracking functionality independently from the digital clock. The stopwatch must allow users to start, stop, reset, and display elapsed time using a similar modular approach as the digital clock, but with additional state control to manage its independent operation.

- The stopwatch features separate counters responsible for tracking seconds and hundredths of a second. These counters increment while the stopwatch is active and reset when the reset button is pressed. The counters' operation is controlled by comparators and accumulators, ensuring seamless transitions between time units for accurate timekeeping.

- The stopwatch counters are designed in a cascading manner. The hundredths of a second counter increments with each clock cycle and triggers the seconds counter to increment when it reaches 100 hundredths of a second, when the seconds counter reaches 60 then it resets. When the stopwatch is reset, the counters reset to zero, and the stopwatch can be restarted from the beginning. These transitions are managed by comparators (compstop1, compstop2, and compstop3), which monitor each counter and generate reset signals when necessary.

- To ensure the time is displayed in a human-readable format, the binary outputs from the counters are converted into Binary-Coded Decimal (BCD) using the binary-to-BCD converters (btobcdstop1 and btobcdstop2). The BCD values are then converted into signals that drive the 7-segment displays, showing the elapsed time in hundredths of a second and seconds.

```
1  -- Stop Watch
2
3  -- Comparator for Stop Watch
4  compstop1: comparator generic map (n => 20)
5      port map (c1 => fstop1, c2 => cstop, output => ostop1);
6
7  compstop2: comparator generic map (n => 8)
8      port map (c1 => fstop2, c2 => cstop2, output => ostop2);
9
10 compstop3: comparator generic map (n => 8)
11     port map (c1 => fstop3, c2 => c2, output => ostop3);
12
13 -- Accumulators for Stop Watch Counters
14 accstop1: accu generic map (n => 20)
15     port map (reset => ostop1, clk => (clock and (not
           startwatch)), output => fstop1);
16
```

```
accstop2: accu generic map (n => 8)
    port map (reset => (ostop2 or (not resetwatch)), clk =>
        ostop1, output => fstop2);

accstop3: accu generic map (n => 8)
    port map (reset => (ostop3 or (not resetwatch)), clk =>
        ostop2, output => fstop3);

-- Binary to BCD Conversion for Stop Watch
btobcdstop1: binaryToBCD
    port map(input => fstop2, d1 => bcdstop1, d2 => bcdstop2);

btobcdstop2: binaryToBCD
    port map(input => fstop3, d1 => bcdstop3, d2 => bcdstop4);

-- BCD to Seven-Segment Conversion for Stop Watch Display
bcdtosevenstop1: bcdtoseven
    port map(input => bcdstop1, output => seven7);

bcdtosevenstop2: bcdtoseven
    port map(input => bcdstop2, output => seven8);

bcdtosevenstop3: bcdtoseven
    port map(input => bcdstop3, output => seven9);

bcdtosevenstop4: bcdtoseven
    port map(input => bcdstop4, output => seven10);
```

- The block diagram in Figure 2 shows how the stopwatch system interacts with the rest of the components. The process begins with the clock signal, which increments the counters when the stopwatch is active. The comparators monitor the counters, ensuring they reset at the appropriate values, while the accumulators manage the actual counting. The BCD converters and 7-segment display logic then process the time for display, ensuring it is visible in a human-readable format. Overall, the stopwatch system is designed to be independent of the digital clock, allowing users to track time separately and resume from where they left off, or reset it entirely. This modular approach makes it easy to test and extend the system while providing reliable, real-time

performance.

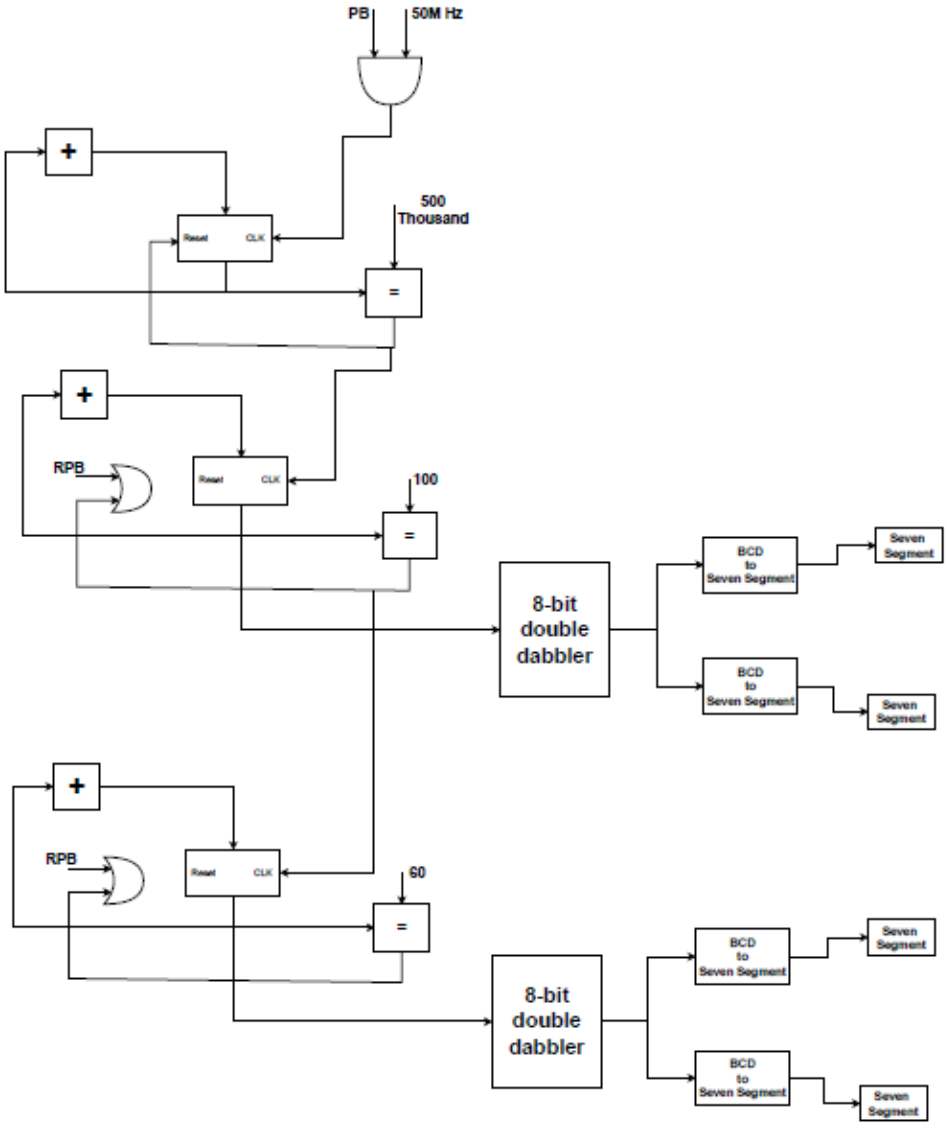


Figure 2: Stop Watch Diagram.

2.2.4 Debounce Design:

- The debouncer design is crucial for ensuring reliable user input processing by eliminating errors caused by the mechanical bouncing of push buttons. When a button is pressed or released, the mechanical contacts often generate unintended fluctuations or "bounces" in the signal. These fluctuations can cause the system to register multiple signals from a single button press, leading to unreliable behavior. The debouncer module addresses this problem by stabilizing the input signal and ensuring that only a single, valid signal is registered for each user input.

- The debouncer module operates by using accumulators and comparators to monitor and filter the input signal. When a button is pressed, the signal typically fluctuates before it stabilizes. The accumulator tracks the signal and holds it stable for a set period, while the comparator monitors the output to determine whether the signal has remained stable long enough to be considered valid. This ensures that only a consistent button press or release is processed, avoiding the noise caused by mechanical bouncing.

- In the VHDL implementation, the debouncer is built with two main components: the accumulator and the comparator. The accumulator stores the signal and ensures that the state change is sustained long enough to be considered a legitimate input. The comparator checks whether the input signal has stabilized, and once the signal is deemed valid, it passes the result to the rest of the system. This process filters out unwanted noise and ensures that each button press is registered correctly.

  The VHDL code for the debouncer is structured as follows:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_arith.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity debouncer is
7        port(
8                clk, pb: in std_logic;
```

```
9               output: out std_logic
10              );
11  end entity;
12
13  architecture behv of debouncer is
14          component accu is
15                  generic (n: integer);
16                  port(reset, clk: in std_logic; output: buffer
                        std_logic_vector(n-1 downto 0));
17          end component;
18
19          component comparator is
20                  generic (n: integer);
21                  port(c1, c2: in std_logic_vector(n-1 downto 0);
                        output: out std_logic);
22          end component;
23
24          signal f1: std_logic_vector(19 downto 0);
25          signal f2, f: std_logic_vector(1 downto 0);
26          signal o1, o2: std_logic;
27
28          begin
29                  comparator1: comparator generic map(n => 20)
                        port map(C1 => f1, C2 =>
                        "01111010000100100000", output => o1);
30                  comparator2: comparator generic map(n => 2)
                        port map(C1 => f2, C2 => "01", output =>
                        o2);
31
32                  acc1: accu generic map(n => 20) port map(reset
                        => (o1 or pb), clk => (clk and not pb),
                        output => f1);
33                  acc2: accu generic map(n => 2) port map(reset
                        => f2(0), clk => (o1 and (not f(0))),
                        output => f2);
34                  acc3: accu generic map(n => 2) port map(reset
                        => pb, clk => o2, output => f);
35
36                  output <= f(0);
37  end architecture;
```

- The diagram of the debouncer operation is shown in Figure 3, illustrating how the input signal is processed through the accumulators and comparators to ensure it is stable before being passed on to the system. The debouncer ensures that user interactions with the digital clock or stopwatch are accurate and error-free, even in the presence of noisy input signals.

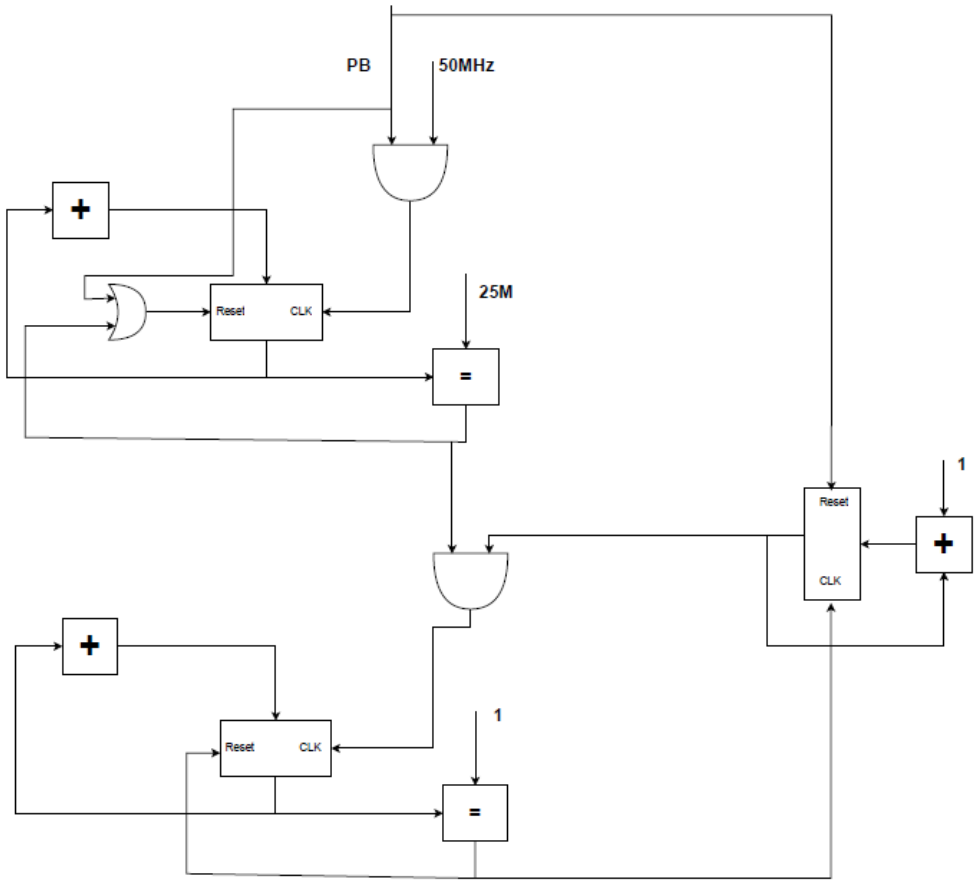


Figure 3: Debouncer Diagram.

By utilizing a modular approach with accumulators and comparators, the debouncer can easily be tested and maintained, guaranteeing that it works reliably in the overall system. This approach allows the user to interact with the stopwatch and digital clock without encountering issues from mechanical button noise. The debouncer design is an

essential part of achieving a smooth, accurate user experience in the system.

2.2.5 Choosing Display Mode:

- Allowing the user to choose what numbers or counters to display is very crucial, as we only posses 4 seven segment displays, and for the digital clock we would need an minimum of six seven segments displays (2 for seconds, 2 for minutes, 2 for hours), and we also need four to display the stopwatch (2 for the hundredths of a second, and 2 for seconds). So we would need to allow the user to choose which counters to see. To solve this problem, we wrote the following VHDL:

```
process(seven1, seven2, seven3, seven4, seven5, seven6,
    seven7, seven8, seven9, seven10, mode)
        begin
           case mode is
                       when "00" => output1 <= seven1; output2
                           <= seven2; output3 <= seven3;
                           output4 <= seven4;
                       when "01" => output1 <= seven3; output2
                           <= seven4; output3 <= seven5;
                           output4 <= seven6;
                       when "10" => output1 <= seven7; output2
                           <= seven8; output3 <= seven9;
                           output4 <= seven10;
                       when "11" => output1 <= seven7; output2
                           <= seven8; output3 <= seven9;
                           output4 <= seven10;
           end case;
end process;
```

- This code takes the input 2-bit binary number "mode" and selects which which counters are displayed on the seven segments, we can see that when the mode is "00", then only seconds and minutes of the digital clock are displayed, when mode is "01", only the minutes and hours of the digital clock are displayed, and lastly when mode is "10" or "11", then the stop watch is displayed. This code does not follow the trend we already set, which is making everything as modular as possible, because as we can see, it is inside a process statement which models the behavior of some hardware, and it is using a switch case statement, which will then be turned into multiplexers when the

hardware is generated.

2.3 System Overview

The design of the digital clock and stopwatch system follows a modular framework, where different components work in unison to manage timekeeping, and accurate display updates. The system is organized into several key modules: the timekeeping counters, comparators, debouncers, and display driver (Binary to BCD and BCD to Seven-Segment). The timekeeping counters are responsible for tracking seconds, minutes, and hours, each in its own module. The counters increment with each clock pulse and handle overflow conditions by resetting to zero after reaching their respective limits (60 for seconds, 60 for minutes, and 24 for hours). These transitions are managed by comparators, which monitor the counter values and trigger reset signals when necessary.

In the digital clock, the seconds counter increments every 50 million clock cycles and triggers the minute counter once it reaches 60 seconds. Similarly, the minute counter increments when it reaches 60, and the hour counter increments when the minutes reach 60. Similarly, for the stopwatch, the hundredths of a second counter increments every clock cycle and triggers the seconds counter after 100 hundredths of a second. The seconds counter increments after reaching 60.

The debouncers are crucial for user interaction, stabilizing input signals from buttons and preventing errors due to mechanical button bouncing. Once stable, these signals are used to control the stopwatch start, stop, and reset functions. The output of the counters is converted into Binary Coded Decimal (BCD) format for easier display handling. The BCD values are then passed through a BCD to Seven-Segment display driver, converting them into signals that drive the seven-segment displays for seconds, minutes, and hours in the digital clock and hundredths of a second and seconds in the stopwatch.

All of these components are linked through clear, defined interfaces that allow smooth interaction between counters, comparators, debouncers, and the display system. This design ensures seamless communication between the modules, guaranteeing accurate, real-time timekeeping and reliable user input handling. The modular approach implemented using VHDL provides a scalable, efficient system that is easy to test and extend for future improvements.

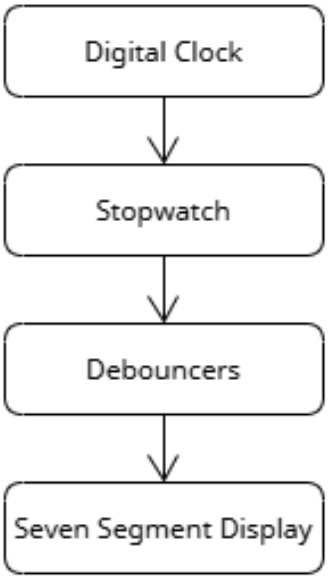


Figure 4: System Overview Diagram.

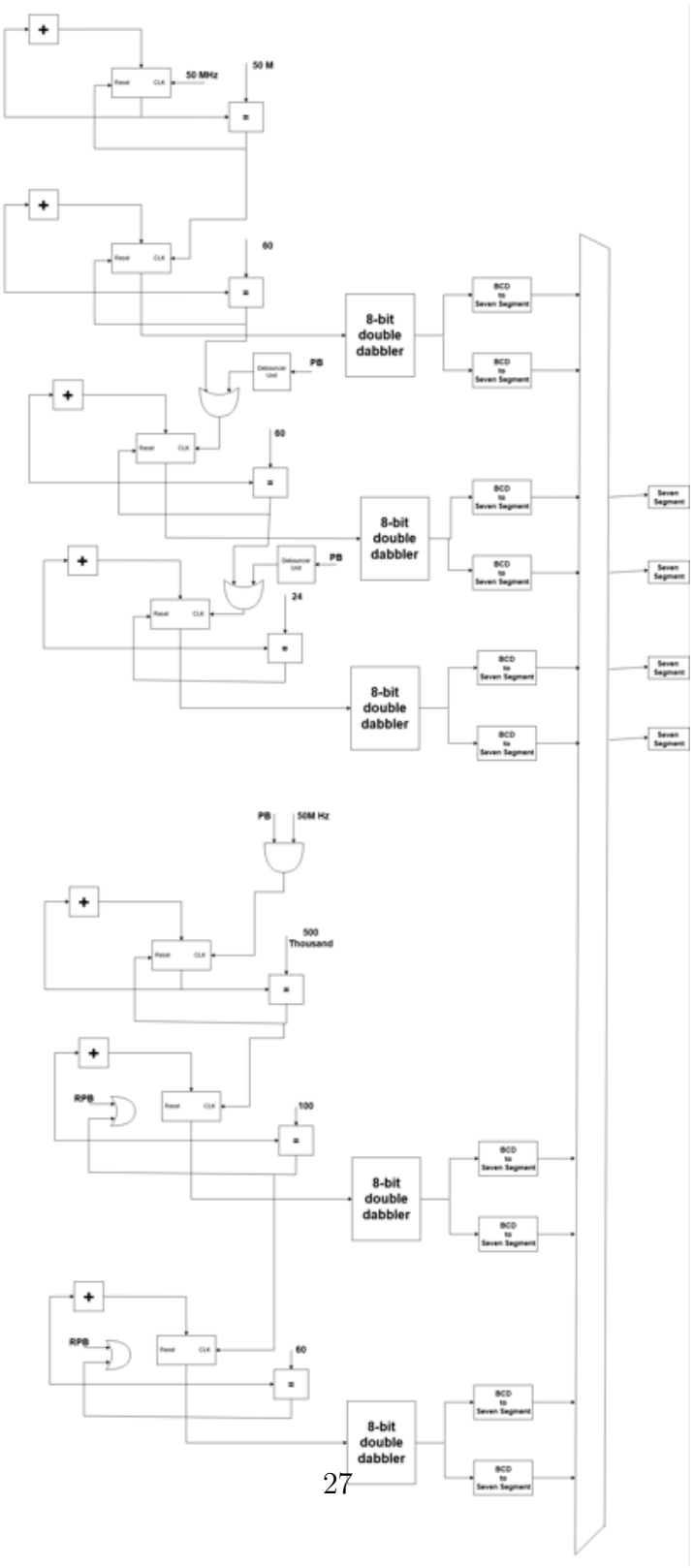
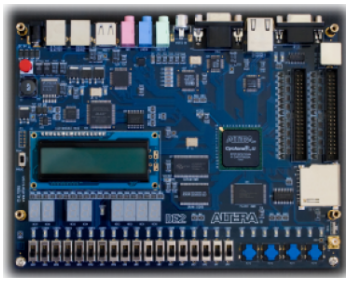


Figure 5: Full System Diagram.

2.4 Component Design

The primary equipment used in this system is the Cyclone V (5CGXFC5C6F27C7N) FPGA, which functions as the processing unit for the digital clock and stopwatch. The FPGA processes user inputs through switches and provides real-time updates on timekeeping by controlling the various components involved. The FPGA drives the time-keeping counters for seconds, minutes, and hours in the digital clock, as well as the stopwatch counters for seconds and hundredths of a second. It manages the flow of time by incrementing these counters, handling user interactions such as start, stop, and reset, and ensuring smooth transitions between time units. The results are displayed on 7-segment displays, which show the current time or elapsed time for both the clock and stopwatch functionalities.

In addition to the FPGA, the modular design of the system is supported by VHDL code that ensures accurate and real-time functionality. The code handles the operations of counters, comparators, accumulators, debouncers, and display drivers, which all work together to ensure the correct operation of the clock and stopwatch. As illustrated in Figure 6, the FPGA board is equipped with switches, push buttons, and 7-segment displays, offering a comprehensive solution for precise and dependable time management. The switches enable users to adjust the time, switch between clock and stopwatch modes, and reset the system. The push buttons are used to manually increment time and control the stopwatch, allowing it to start, pause, and reset. The 7-segment displays show the time in a clear, readable format, ensuring the system is efficient and easy to use.



(a) Diagram of the Cyclone V FPGA with 7-segment displays

(b) Altera Quartus II Software

Figure 6: List of Equipment Used

The core of the digital clock and stopwatch system is built using an FPGA, which serves as the processing unit to manage timekeeping and handle user inputs efficiently. The FPGA manages the operation of the timekeeping counters for seconds, minutes, and hours, while also supporting the stopwatch function with its independent counters for seconds and hundredths of a second. The system is designed in a modular fashion, with distinct components working together to ensure accurate real-time time tracking and display updates.

- **Digital Clock and Stopwatch Counters:** The counters for the digital clock and stopwatch are the heart of the timekeeping process. These counters are implemented in a cascading structure, where the seconds counter increments with each clock pulse and triggers the next counter once it reaches its maximum value (60 for seconds, 60 for minutes, and 24 for hours). The stopwatch follows a similar structure but tracks hundredths of a second, and seconds, with the hundredths of a second counter triggering the seconds counter when it reaches 100 hundredths of a second. Similarly, the seconds counter triggers the minutes counter after reaching 60 seconds, and the minutes counter triggers the hour counter when it reaches 60 minutes. The counters are controlled by accumulators, which increment their values based on clock cycles and reset when the specified limits are reached, providing a seamless flow of time.

- **Debouncing and User Interaction:** User interaction is handled

through push buttons to control the digital clock and stopwatch, and through switches to chose the display mode. The debouncer component is implemented to prevent errors caused by mechanical bouncing of the buttons, ensuring that each button press is registered only once. The debouncers stabilize the input signals for controlling the stopwatch's start, stop, reset, and the clock's manual time adjustments. The debouncer design ensures reliable and consistent user interaction by eliminating the possibility of multiple unintended signals.

- **Display and Time Conversion:** To display the time in a human-readable format, the binary outputs of the counters are converted to Binary-Coded Decimal (BCD) using Binary to BCD converters. These BCD values are then processed by BCD to Seven-Segment converters, which after checking the mode of operation that the user chose, activate the appropriate segments of the 7-segment displays to show the current time. The system's modular design allows for real-time updates of the display, ensuring that the time is shown accurately and promptly.

- **Stopwatch and Digital Clock Integration:** The stopwatch operates independently of the digital clock, with its counters for tracking seconds and hundredths of a second. These counters reset when the user presses the reset button, and the stopwatch can be restarted from the beginning. Comparators monitor the stopwatch counters, ensuring the correct resetting of values and smooth transitions between time units. The modular design allows the stopwatch to function independently, providing a reliable way to track elapsed time without affecting the digital clock's operation.

- **VHDL Implementation:** The system's functionality is implemented in VHDL, with each component—such as counters, debouncers, and display drivers—working together to achieve the desired outcome. The VHDL code for the counters, comparators, and display drivers ensures accurate timekeeping and smooth user interaction. Each module is designed with clear interfaces, allowing for efficient communication between the components. Below is the VHDL code that demonstrates the implementation of the digital clock's core functionalities:

```
1    --Digital Clock
```

```
2                comp1: comparator generic map (n => 26) port
                     map (c1 => f1, c2 => c1, output => o1);
3                comp2: comparator generic map (n => 8) port map
                     (c1 => f2, c2 => c2, output => o2);
4                comp3: comparator generic map (n => 8) port map
                     (c1 => f3, c2 => c3, output => o3);
5                comp4: comparator generic map (n => 8) port map
                     (c1 => f4, c2 => c4, output => o4);
6                mdebounce: debouncer port map(clk => clock, pb
                     => incm, output => mpb);
7                hdebounce: debouncer port map(clk => clock, pb
                     => inch, output => hpb);
8
9
10               acc1: accu generic map (n => 26) port map
                     (reset => o1, clk => clock, output => f1);
11               acc2: accu generic map (n => 8) port map (reset
                     => o2, clk => o1, output => f2);
12               acc3: accu generic map (n => 8) port map (reset
                     => o3, clk => (o2 or mpb), output => f3);
13               acc4: accu generic map (n => 8) port map (reset
                     => o4, clk => (o3 or hpb), output => f4);
14               btobcd1: binaryToBCD port map(input => f2, d1
                     => bcd1, d2 => bcd2);
15               btobcd2: binaryToBCD port map(input => f3, d1
                     => bcd3, d2 => bcd4);
16               btobcd3: binaryToBCD port map(input => f4, d1
                     => bcd5, d2 => bcd6);
17               bcdtoseven1: bcdtoseven port map(input => bcd1,
                     output => seven1);
18               bcdtoseven2: bcdtoseven port map(input => bcd2,
                     output => seven2);
19               bcdtoseven3: bcdtoseven port map(input => bcd3,
                     output => seven3);
20               bcdtoseven4: bcdtoseven port map(input => bcd4,
                     output => seven4);
21               bcdtoseven5: bcdtoseven port map(input => bcd5,
                     output => seven5);
22               bcdtoseven6: bcdtoseven port map(input => bcd6,
                     output => seven6);
```

```
23
24              --Stop Watch
25              compstop1: comparator generic map (n => 20)
                    port map (c1 => fstop1, c2 => cstop, output
                    => ostop1);
26              compstop2: comparator generic map (n => 8) port
                    map (c1 => fstop2, c2 => cstop2, output =>
                    ostop2);
27              compstop3: comparator generic map (n => 8) port
                    map (c1 => fstop3, c2 => c2, output =>
                    ostop3);
28              accstop1: accu generic map (n => 20) port map
                    (reset => ostop1, clk => (clock and (not
                    startwatch)), output => fstop1);
29              accstop2: accu generic map (n => 8) port map
                    (reset => (ostop2 or (not resetwatch)), clk
                    => ostop1, output => fstop2);
30              accstop3: accu generic map (n => 8) port map
                    (reset => (ostop3 or (not resetwatch)), clk
                    => ostop2, output => fstop3);
31              btobcdstop1: binaryToBCD port map(input =>
                    fstop2, d1 => bcdstop1, d2 => bcdstop2);
32              btobcdstop2: binaryToBCD port map(input =>
                    fstop3, d1 => bcdstop3, d2 => bcdstop4);
33              bcdtosevenstop1: bcdtoseven port map(input =>
                    bcdstop1, output => seven7);
34              bcdtosevenstop2: bcdtoseven port map(input =>
                    bcdstop2, output => seven8);
35              bcdtosevenstop3: bcdtoseven port map(input =>
                    bcdstop3, output => seven9);
36              bcdtosevenstop4: bcdtoseven port map(input =>
                    bcdstop4, output => seven10);
37
38              process(seven1, seven2, seven3, seven4, seven5,
                    seven6, seven7, seven8, seven9, seven10,
                    mode)
39              begin
40                      case mode is
41                              when "00" => output1 <= seven1;
                                    output2 <= seven2; output3 <=
```

```
                                    seven3; output4 <= seven4;
42                                when "01" => output1 <= seven3;
                                    output2 <= seven4; output3 <=
                                    seven5; output4 <= seven6;
43                                when "10" => output1 <= seven7;
                                    output2 <= seven8; output3 <=
                                    seven9; output4 <= seven10;
44                                when "11" => output1 <= seven7;
                                    output2 <= seven8; output3 <=
                                    seven9; output4 <= seven10;
45                        end case;
46        end process;
```

This design, implemented in VHDL, ensures that the digital clock and stopwatch work efficiently and accurately, allowing for precise timekeeping and reliable user interaction through simple push-button controls.

3 Experimental Testing and Results

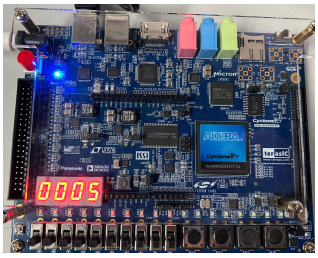
3.1 Testing Plan and Acceptance Criteria

This project was completed using the Quartus II program, so we just uploaded the code to the FPGA for testing. We used pushbuttons to test if the clock would correctly increment, in addition, we used switches to switch between the two different displays we had and checked the seven-segment display to see if it produced the correct numbers.

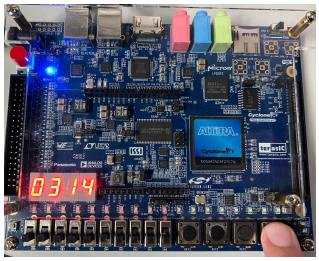
3.2 Results

3.2.1 Testing the digital clock

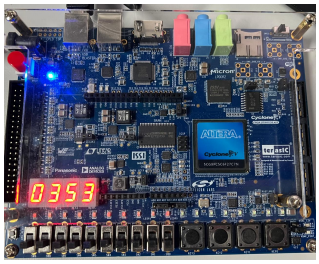
As can be seen in Figure 7, we were able to build a fully functional digital clock that will increment the seconds, minutes, and hours at the correct time. We were able to verify our design by comparing it to a real clock to check whether the incrementing was correct or not.



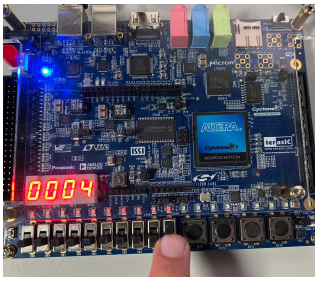
(a) The seven-segment displays zero minutes and 5 seconds (00:05).



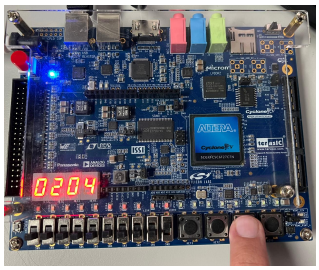
(b) The seven-segment displays three minutes and 14 seconds (03:14) after using the push button (which increments the minutes) to test it and see the minute.



(c) The seven-segment displays three minutes and 53 seconds (03:53) by incrementing alone by time.



(d) The seven-segment displays zero hours and 4 minutes (00:04) after using the switch to convert from the minute-second display to the hours-minute display.

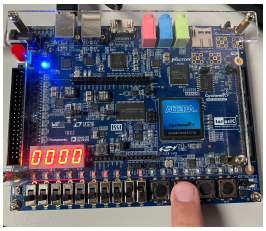


(e) The seven-segment displays two hours and 4 minutes (02:04) after using the push button (which increments the hours) to test it and see the hours increasing.

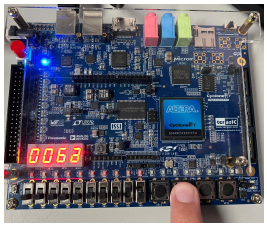
Figure 7: Testing Our Digital Clock System

3.2.2 Testing the Stopwatch

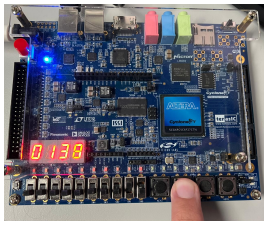
After various trials and errors, we were able to build a successful stopwatch that would start counting the moment we pressed a pushbutton and stop whenever the pushbutton was released. In addition, pushing another push-button will reset the stopwatch. All of this happens while the original clock is still working. This can be seen in the images.



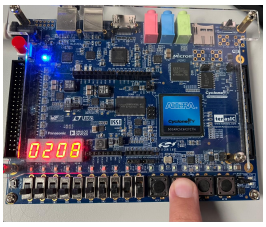
(a) After converting from the Digital clock to a stopwatch using the switch, we can use the push button which is shown to start the stopwatch.



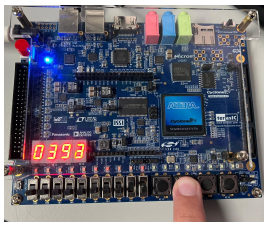
(b) The stopwatch is showing zero seconds and 63 hundredths of a second (00:63).



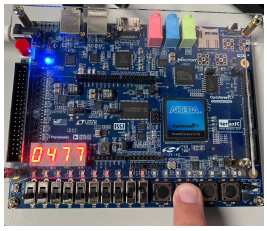
(c) By keeping pressing the push button, it will keep running the stopwatch, and here it shows one second and 38 hundredths of a second (01:38)



(d) The stopwatch is showing two seconds and eight hundredths of a second (02:08).



(e) The stopwatch is showing three seconds and 93 hundredths of a second (03:93).

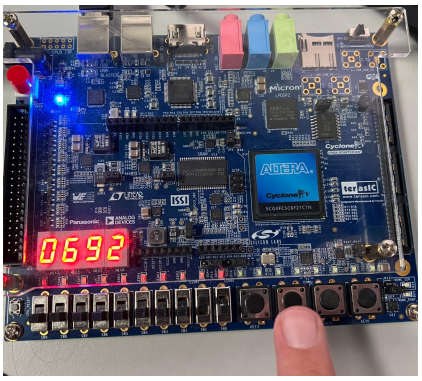


(f) Keep pressing the push button to keep it running, but whenever you remove your finger from the push button, it will stop immediately

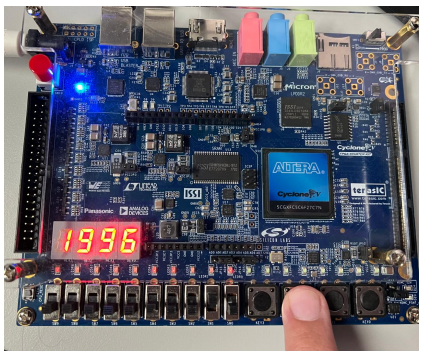
Figure 8: Testing Our Stopwatch System

3.2.3 Testing the manual increment

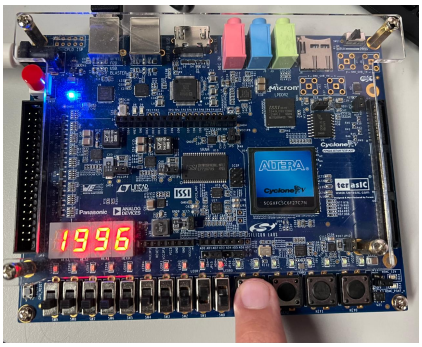
We decided to add an extra feature whereby, by pressing a certain push button, we can manually increment seconds, minutes, and hours, and the clock will start counting from there instead of waiting to set it to your real-life time, as can be seen in Figure 7.



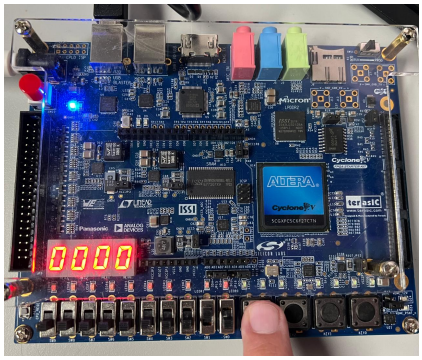
(a) Here as we can see, our finger is removed, and the stopwatch stopped (pause)



(b) After pausing the stopwatch, we can continue with it again



(c) The reset push button to reset the stopwatch timer, as can be seen in Figure 9d



(d) As it shows, the stopwatch has been rested after pressing the push button

Figure 9: Testing Our Stopwatch System

3.3 Analysis and Interpretation of Data

The digital clock/stopwatch design on FPGA produced results that matched expectations, demonstrating FPGA's ability to efficiently handle timekeeping systems with precise counters for seconds, minutes, and hours. The time formatting logic worked as expected, and the system performed admirably for routine timekeeping activities. The architecture exhibited FPGA's ability in handling parallel processes, allowing for optimal resource utilization while maintaining consistent performance. This study demonstrated the adaptability of FPGA-based designs for digital clocks, as the system's modularity allowed for straightforward feature modification and expansion. The implementation's success confirmed the viability of FPGA-based solutions in timekeeping applications, laying the groundwork for future enhancements such as incorporating more complex features or refining the design for specific applications.

4 Conclusion

4.1 Summary

To summarize everything that has been stated so far, this project created a digital clock/Stopwatch with an FPGA (Cyclone V). The system used accumulator circuits combined with a comparator circuit The FPGA's parallel processing capabilities allowed for effective real-time data handling, achieving the goal of developing to accurately achieve a fully functional clock.. Digital clocks on FPGA are appropriate for applications that demand a simple, dependable timekeeping system, and its modular design allows for easy customization and expansion with new capabilities.

4.2 Future Improvements and Takeaways

Designing a digital clock/stopwatch on an FPGA demonstrated the versatility and efficiency of FPGAs in timekeeping applications. We learned the value of managing system resources, especially as the design expands with new features like as alerts and alternative time formats. Future enhancements could include integrating RTC modules for increased accuracy, optimizing resource utilization, and incorporating advanced displays such as LCDs for a

more user-friendly interface. Overall, the project improved our understanding of FPGA design by emphasizing flexibility and striking a balance between complexity, performance, and resource efficiency.

4.3 Lessons Learned

In this project, we learned about FPGA design, specifically how to develop timekeeping devices such as digital clocks and stopwatches. We used our knowledge of counters, time formatting, and logic design to create a working clock system. While we did not use any new hardware technologies, we did go deeper into VHDL/Verilog to build logic on the FPGA and improve temporal synchronization. We investigated FPGA clock generation strategies and how to handle time overflow in digital clocks, and we used this knowledge to ensure precise time tracking. We also learned how to optimize FPGA resource utilization while maintaining functionality and performance. Some of the lessons learned were through trial and error, particularly when attempting to accurately format time and manage clock signal synchronization, which initially caused design challenges.

4.4 Team Dynamics

- **Who was the team leader? What evidence of good leadership can you provide?**

  Mohammad Fares was the team leader. He demonstrated strong leadership by delegating tasks, maintaining clear communication, addressing challenges promptly, and ensuring everyone's input was valued, which kept the team on track.

- **How did you create a collaborative and inclusive environment? Were all members engaged? How did the team communicate/-collaborate? Did you create a WhatsApp group? Met in the library? Met in the lab? Communicated by emails? Were all members in attendance? Did you take meeting minutes?**

  We fostered a collaborative environment by encouraging open communication and ensuring everyone contributed their ideas. We set up a WhatsApp group for quick updates and used the library for brainstorming sessions, while the lab served as the space for practical work.

Regular meetings were held, and we communicated through email as well, with meeting minutes taken to monitor our progress.

- **What goals did you set for your team?**

  The main goals were to deliver a working digital clock and stopwatch system on time, meeting technical requirements, and to collaborate efficiently to complete all tasks, including design, coding, testing, and final presentation.

- **How did you plan the tasks? Did you use a Gantt Chart? Did you track your progress and update your tasks?**

  We used a Gantt Chart which is shown in Figure 10 to plan and track tasks, breaking the project into manageable phases. Progress was reviewed during meetings, and the chart was updated regularly to adjust for any delays or changes.

- **Did you meet your objectives?**

  Yes, we met our objectives. The system was delivered on time, fully functional, and the final presentation effectively showcased our work.

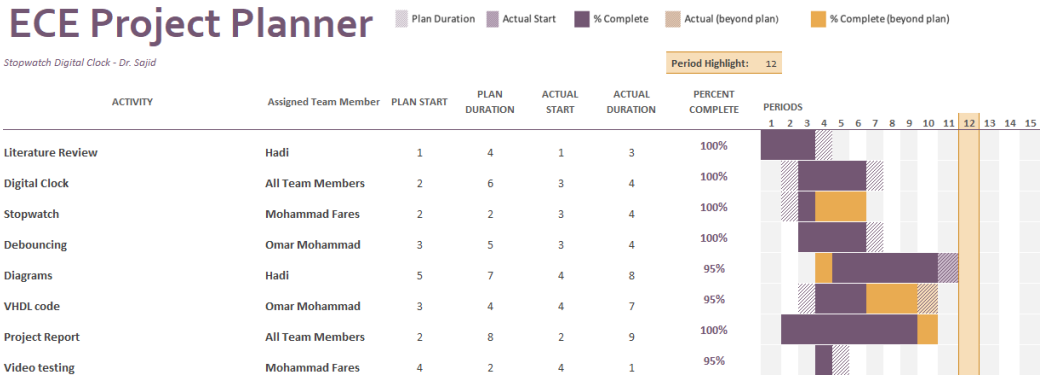


Figure 10: Gantt Chart.

4.5 Impact Statement

- **Does your solution help society? How?**
  Yes, it provides accurate timekeeping for everyday use, benefiting industries like healthcare and education.

- **Are there any privacy concerns for the users?**
  No, the system doesn't store any personal data, so privacy is not an issue.

- **Will your solution create jobs? Remove jobs from the market?**
  It may create jobs in manufacturing and maintenance but won't remove existing jobs.

- **Will the electronics from your project end up being electronic waste? How did you reduce this? What impact does your system have on the environment? Does it reduce dependency on fossil fuel? Is it sustainable?**
  The system uses modular components for easy recycling. Its low energy consumption reduces environmental impact and fossil fuel reliance, making it more sustainable.

- **Does your system reduce energy demand? Increase energy demand? By how much? Is it still worth it?**
  The system uses minimal energy, cutting demand compared to traditional devices. It's energy-efficient and worth the minimal energy cost.

- **Does your system provide comparable performance to existing ones while reducing the cost? Reducing the power consumption? Reducing the environmental footprint?**
  It performs similarly to existing systems but at a lower cost and with reduced power use and environmental impact.

Environmental Impact Analysis								
Impact of your project	**Nature**	**Extent**	**Timing**	**Severity**	**Duration**	**Reversibility**	**Uncertainty**	**Significance**
The climate Example: *Does the project affect the emission of greenhouse gases into the atmosphere?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Use of Energy Example: *Does your project affect the energy consumption of the economy? How?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on detecting phones.							
Air quality Example: *Does the project have an effect on emissions of harmful air pollutants that might affect human health, damage crops or buildings or lead to deterioration in the environment (soil or rivers)?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Biodiversity, flora, fauna and landscapes Example: *Does it affect endangered species, their habitats or ecologically-sensitive areas?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							

Water quality and resources Example: *Does the project decrease or increase the quality or quantity of freshwater and groundwater?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Renewable or non-renewable resources Example: *Does the project reduce or increase use of non-renewable resources?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Sustainability Example: Does the option lead to more sustainable production and consumption? How?	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Waste production/generation/recycling Example: *Does the project affect waste production (solid, urban, agricultural, industrial, mining, radioactive or toxic waste) or how waste is treated, disposed of or recycled?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							

Economic Impact Analysis								
Impact of your project	**Nature**	**Extent**	**Timing**	**Severity**	**Duration**	**Reversibility**	**Uncertainty**	**Significance**
Economic Prosperity Example: *Does the project affect the GDP/capita, employment rate, household savings?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: It might affect it because its innovative companies might buy it							
Investment Flows Example: *Does your project affect the flow of investment from outside the country? Does it encourage local investment in it?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: yes, because of the nature of this project, many people who are interested in keeping the workplace time strict, this project might useful							
Public Budgets or Services Example: *Does the project affect the budgets of hospitals, community services, older people services, transport services, service quality, schools, policing, municipality services..etc?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: it might affect it because it might help people to pay attention to time,so big companies might buy it.							
Market Mechanisms Example: *Does it affect the private sector business opportunities? Help companies reach more costumers? Change how business is done?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							

Innovation, Research and Development Example: *Does the project have commercialization potential, lead to a potential patent? Does it allow others to innovate/research through it?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: yes, many engineers might develop it and make it more efficient,by improving the range and the quality of the components.							
Sustainable Consumption and Production Example: Does the project produce a sustainably consumed product or service? Can it be produced sustainably?	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							

Social Impact Analysis								
Impact of your project	**Nature**	**Extent**	**Timing**	**Severity**	**Duration**	**Reversibility**	**Uncertainty**	**Significance**
Health and Longevity Example: *Does the project impact health and longevity? Does it affect physical activity, nutrition, chronic diseases, accidental injuries, independent living, mental wellbeing?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Safety Example: *Does your project affect safety of social environment, protection of older people against abuse, protection against risks, response to emergency cases, feelings of safety, physical safety?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Productive and Valued Activities Example: *Does the project increase leisure time, reduce stress, lead to positive behavior, increase productivity?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							

Standard of Living Example: *Does it affect the quality of life? Make lives easier? Reduce poverty and deprivation? Increase life choices and opportunities?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Education/Life-long Learning Example: *Does the project affect literacy, use of ICT, chances of higher education, quality of education, life-long learning? Improve attainment of learning outcomes?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: it might help students manage their time ,thus thriving in education.							
Quality of Social Interaction Example: *Does the project affect social connectedness, social participation, volunteering?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							

Privacy and Personal Data Example: *Does the project reveal the user identities? Create potential private data leaks or identity theft?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							
Social Reasonability Example: *Does the project affect access to products and services for people of determination? Does it affect their integration into society? Does it affect their participation in the economy? Does it address their needs?*	Direct Positive	Local	Immediate	High	Temporary	Reversible	Low Likelihood	Unimportant
	Justification/Explanation: no, our project focuses on building a digital clock							

A RTL Representation of Each Component

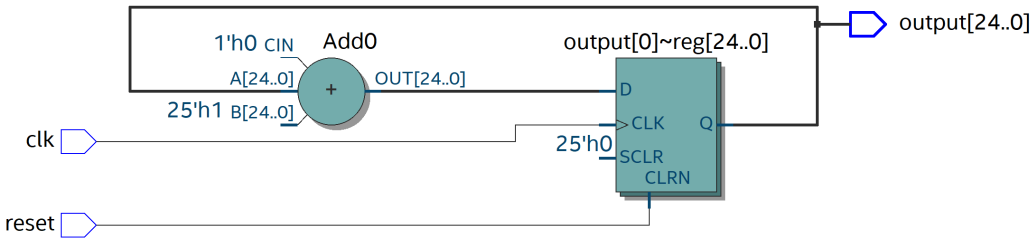


Figure 11: Counter.

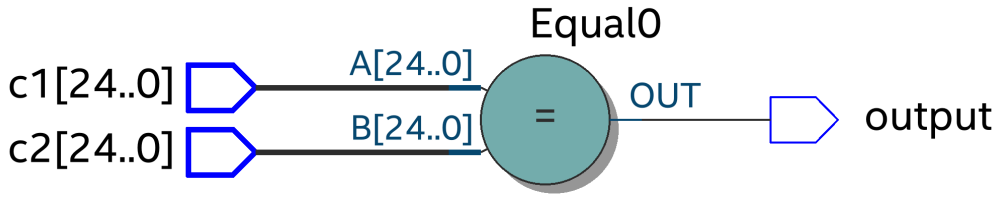


Figure 12: Comparator.

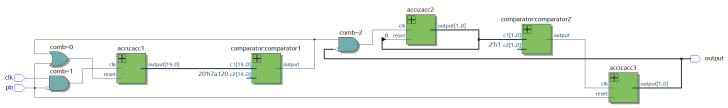


Figure 13: Debouncer Unit.

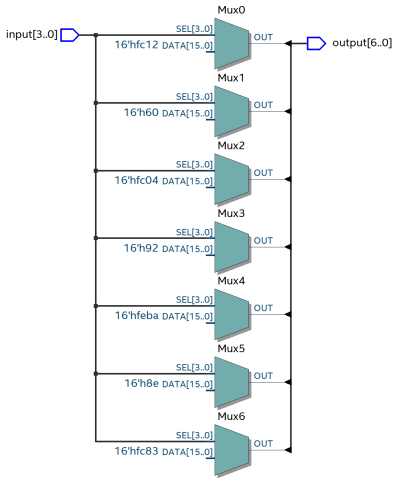


Figure 14: BCD to Seven Segments.

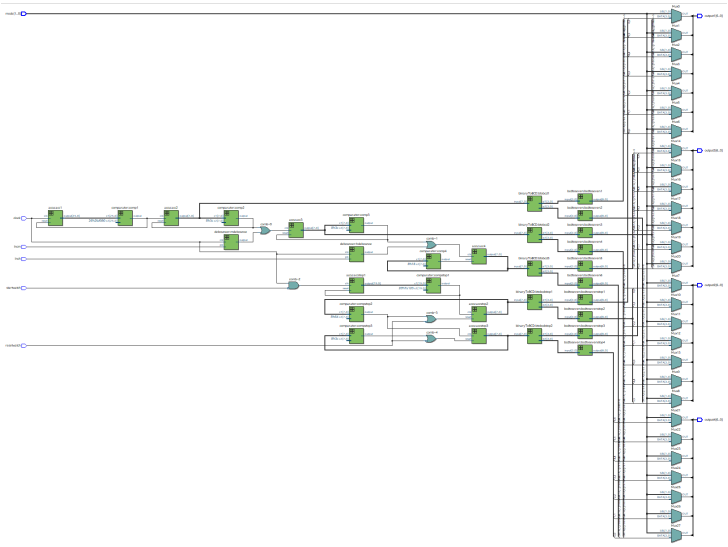


Figure 15: Full System.

B Code for Accumulator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY accu is
    generic (
        n: integer
    );
    port(
        reset, clk: in std_logic;
        output: buffer std_logic_vector(n-1 downto 0)
    );
end ENTITY;

architecture behv of accu is
begin
    process(clk, reset)
    begin
        if reset = '1' then
            output <= (others => '0');
        elsif rising_edge(clk) then
            output <= output + 1;
        end if;
    end process;
end architecture;
```

C Code for Comparator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY comparator is
```

```
    generic (
        n: integer
    );
    port(
        c1, c2: in std_logic_vector(n-1 downto 0);
        output: out std_logic
    );
end ENTITY;

architecture behv of comparator is
begin
    process(c1, c2)
    begin
        if c1 = c2 then
            output <= '1';
        else
            output <= '0';
        end if;
    end process;
end architecture;
```

D Code for Debouncer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity debouncer is
    port(
        clk, pb: in std_logic;
        output: out std_logic
    );
end entity;

architecture behv of debouncer is

    component accu is
```

```
16          generic (
17              n: integer
18          );
19          port(
20              reset, clk: in std_logic;
21              output: buffer std_logic_vector(n-1 downto 0)
22          );
23      end component;
24
25      component comparator is
26          generic (
27              n: integer
28          );
29          port(
30              c1, c2: in std_logic_vector(n-1 downto 0);
31              output: out std_logic
32          );
33      end component;
34
35      signal f1: std_logic_vector(19 downto 0);
36      signal f2, f: std_logic_vector(1 downto 0);
37
38      signal o1, o2: std_logic;
39
40  begin
41
42      comparator1: comparator generic map(n => 20) port map(C1 => f1,
            C2 => "01111010000100100000", output => o1);
43      comparator2: comparator generic map(n => 2) port map(C1 => f2,
            C2 => "01", output => o2);
44
45      acc1: accu generic map(n => 20) port map(reset => (o1 or pb),
            clk => (clk and not pb), output => f1);
46      acc2: accu generic map(n => 2) port map(reset => f2(0), clk =>
            (o1 and (not f(0))), output => f2);
47      acc3: accu generic map(n => 2) port map(reset => pb, clk => o2,
            output => f);
48
49      output <= f(0);
50
```

```
end architecture;
```

E Code for Binary to BCD Conversion

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY binaryToBCD is

        port( input: in std_logic_vector(7 downto 0);
                        d1, d2: out std_logic_vector(3 downto 0)
        );
end ENTITY;

architecture behv of binaryToBCD is


        begin

                process(input)
                        begin
                                                                CASE
                                                                    input
                                                                    IS
    when "00000000" => d1 <= "0000"; d2 <= "0000"; -- 0
    when "00000001" => d1 <= "0001"; d2 <= "0000"; -- 1
    when "00000010" => d1 <= "0010"; d2 <= "0000"; -- 2
    when "00000011" => d1 <= "0011"; d2 <= "0000"; -- 3
    when "00000100" => d1 <= "0100"; d2 <= "0000"; -- 4
    when "00000101" => d1 <= "0101"; d2 <= "0000"; -- 5
    when "00000110" => d1 <= "0110"; d2 <= "0000"; -- 6
    when "00000111" => d1 <= "0111"; d2 <= "0000"; -- 7
    when "00001000" => d1 <= "1000"; d2 <= "0000"; -- 8
    when "00001001" => d1 <= "1001"; d2 <= "0000"; -- 9
    when "00001010" => d1 <= "0000"; d2 <= "0001"; -- 10
    when "00001011" => d1 <= "0001"; d2 <= "0001"; -- 11
```

```
33      when "00001100" => d1 <= "0010"; d2 <= "0001"; -- 12
34      when "00001101" => d1 <= "0011"; d2 <= "0001"; -- 13
35      when "00001110" => d1 <= "0100"; d2 <= "0001"; -- 14
36      when "00001111" => d1 <= "0101"; d2 <= "0001"; -- 15
37      when "00010000" => d1 <= "0110"; d2 <= "0001"; -- 16
38      when "00010001" => d1 <= "0111"; d2 <= "0001"; -- 17
39      when "00010010" => d1 <= "1000"; d2 <= "0001"; -- 18
40      when "00010011" => d1 <= "1001"; d2 <= "0001"; -- 19
41      when "00010100" => d1 <= "0000"; d2 <= "0010"; -- 20
42      when "00010101" => d1 <= "0001"; d2 <= "0010"; -- 21
43      when "00010110" => d1 <= "0010"; d2 <= "0010"; -- 22
44      when "00010111" => d1 <= "0011"; d2 <= "0010"; -- 23
45      when "00011000" => d1 <= "0100"; d2 <= "0010"; -- 24
46      when "00011001" => d1 <= "0101"; d2 <= "0010"; -- 25
47      when "00011010" => d1 <= "0110"; d2 <= "0010"; -- 26
48      when "00011011" => d1 <= "0111"; d2 <= "0010"; -- 27
49      when "00011100" => d1 <= "1000"; d2 <= "0010"; -- 28
50      when "00011101" => d1 <= "1001"; d2 <= "0010"; -- 29
51      when "00011110" => d1 <= "0000"; d2 <= "0011"; -- 30
52      when "00011111" => d1 <= "0001"; d2 <= "0011"; -- 31
53      when "00100000" => d1 <= "0010"; d2 <= "0011"; -- 32
54      when "00100001" => d1 <= "0011"; d2 <= "0011"; -- 33
55      when "00100010" => d1 <= "0100"; d2 <= "0011"; -- 34
56      when "00100011" => d1 <= "0101"; d2 <= "0011"; -- 35
57      when "00100100" => d1 <= "0110"; d2 <= "0011"; -- 36
58      when "00100101" => d1 <= "0111"; d2 <= "0011"; -- 37
59      when "00100110" => d1 <= "1000"; d2 <= "0011"; -- 38
60      when "00100111" => d1 <= "1001"; d2 <= "0011"; -- 39
61      when "00101000" => d1 <= "0000"; d2 <= "0100"; -- 40
62      when "00101001" => d1 <= "0001"; d2 <= "0100"; -- 41
63      when "00101010" => d1 <= "0010"; d2 <= "0100"; -- 42
64      when "00101011" => d1 <= "0011"; d2 <= "0100"; -- 43
65      when "00101100" => d1 <= "0100"; d2 <= "0100"; -- 44
66      when "00101101" => d1 <= "0101"; d2 <= "0100"; -- 45
67      when "00101110" => d1 <= "0110"; d2 <= "0100"; -- 46
68      when "00101111" => d1 <= "0111"; d2 <= "0100"; -- 47
69      when "00110000" => d1 <= "1000"; d2 <= "0100"; -- 48
70      when "00110001" => d1 <= "1001"; d2 <= "0100"; -- 49
71      when "00110010" => d1 <= "0000"; d2 <= "0101"; -- 50
72      when "00110011" => d1 <= "0001"; d2 <= "0101"; -- 51
```

```
73      when "00110100" => d1 <= "0010"; d2 <= "0101"; -- 52
74      when "00110101" => d1 <= "0011"; d2 <= "0101"; -- 53
75      when "00110110" => d1 <= "0100"; d2 <= "0101"; -- 54
76      when "00110111" => d1 <= "0101"; d2 <= "0101"; -- 55
77      when "00111000" => d1 <= "0110"; d2 <= "0101"; -- 56
78      when "00111001" => d1 <= "0111"; d2 <= "0101"; -- 57
79      when "00111010" => d1 <= "1000"; d2 <= "0101"; -- 58
80      when "00111011" => d1 <= "1001"; d2 <= "0101"; -- 59
81          when "00111100" => d1 <= "0000"; d2 <= "0110"; -- 60
82      when "00111101" => d1 <= "0001"; d2 <= "0110"; -- 61
83      when "00111110" => d1 <= "0010"; d2 <= "0110"; -- 62
84      when "00111111" => d1 <= "0011"; d2 <= "0110"; -- 63
85      when "01000000" => d1 <= "0100"; d2 <= "0110"; -- 64
86      when "01000001" => d1 <= "0101"; d2 <= "0110"; -- 65
87      when "01000010" => d1 <= "0110"; d2 <= "0110"; -- 66
88      when "01000011" => d1 <= "0111"; d2 <= "0110"; -- 67
89      when "01000100" => d1 <= "1000"; d2 <= "0110"; -- 68
90      when "01000101" => d1 <= "1001"; d2 <= "0110"; -- 69
91      when "01000110" => d1 <= "0000"; d2 <= "0111"; -- 70
92      when "01000111" => d1 <= "0001"; d2 <= "0111"; -- 71
93      when "01001000" => d1 <= "0010"; d2 <= "0111"; -- 72
94      when "01001001" => d1 <= "0011"; d2 <= "0111"; -- 73
95      when "01001010" => d1 <= "0100"; d2 <= "0111"; -- 74
96      when "01001011" => d1 <= "0101"; d2 <= "0111"; -- 75
97      when "01001100" => d1 <= "0110"; d2 <= "0111"; -- 76
98      when "01001101" => d1 <= "0111"; d2 <= "0111"; -- 77
99      when "01001110" => d1 <= "1000"; d2 <= "0111"; -- 78
100     when "01001111" => d1 <= "1001"; d2 <= "0111"; -- 79
101     when "01010000" => d1 <= "0000"; d2 <= "1000"; -- 80
102     when "01010001" => d1 <= "0001"; d2 <= "1000"; -- 81
103     when "01010010" => d1 <= "0010"; d2 <= "1000"; -- 82
104     when "01010011" => d1 <= "0011"; d2 <= "1000"; -- 83
105     when "01010100" => d1 <= "0100"; d2 <= "1000"; -- 84
106     when "01010101" => d1 <= "0101"; d2 <= "1000"; -- 85
107     when "01010110" => d1 <= "0110"; d2 <= "1000"; -- 86
108     when "01010111" => d1 <= "0111"; d2 <= "1000"; -- 87
109     when "01011000" => d1 <= "1000"; d2 <= "1000"; -- 88
110     when "01011001" => d1 <= "1001"; d2 <= "1000"; -- 89
111     when "01011010" => d1 <= "0000"; d2 <= "1001"; -- 90
112     when "01011011" => d1 <= "0001"; d2 <= "1001"; -- 91
```

```
    when "01011100" => d1 <= "0010"; d2 <= "1001"; -- 92
    when "01011101" => d1 <= "0011"; d2 <= "1001"; -- 93
    when "01011110" => d1 <= "0100"; d2 <= "1001"; -- 94
    when "01011111" => d1 <= "0101"; d2 <= "1001"; -- 95
    when "01100000" => d1 <= "0110"; d2 <= "1001"; -- 96
    when "01100001" => d1 <= "0111"; d2 <= "1001"; -- 97
    when "01100010" => d1 <= "1000"; d2 <= "1001"; -- 98
    when "01100011" => d1 <= "1001"; d2 <= "1001"; -- 99
    when others =>
        d1 <= "0000"; d2 <= "0000";
      end case;
        end process;
end architecture;
```

F Code for BCD to Seven-Segment Conversion

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity bcdtoseven is
    port(
        input: in std_logic_vector(3 downto 0);
        output: out std_logic_vector(6 downto 0)
    );
end ENTITY;

architecture behv of bcdtoseven is
begin
    process(input)
    begin
        case input is
            when "0000" => output <= "0000001"; -- 0
            when "0001" => output <= "1001111"; -- 1
            when "0010" => output <= "0010010"; -- 2
```

```
            when "0011" => output <= "0000110"; -- 3
            when "0100" => output <= "1001100"; -- 4
            when "0101" => output <= "0100100"; -- 5
            when "0110" => output <= "0100000"; -- 6
            when "0111" => output <= "0001111"; -- 7
            when "1000" => output <= "0000000"; -- 8
            when "1001" => output <= "0000100"; -- 9
            when others => output <= "1010101"; -- Error state
        end case;
    end process;
end architecture;
```

G Code for ADSD (Digital Clock and Stopwatch)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ADSD is
    port(
        clock, incm, inch, startwatch, resetwatch: in std_logic;
        mode: in std_logic_vector(1 downto 0);
        output1, output2, output3, output4: out std_logic_vector(6
            downto 0)
    );
end ENTITY;

architecture behv of ADSD is
    component comparator is
        generic (n: integer);
        port(c1, c2: in std_logic_vector(n-1 downto 0); output: out
            std_logic);
    end component;

    component accu is
```

```
21          generic (n: integer);
22          port(reset, clk: in std_logic; output: buffer
                std_logic_vector(n-1 downto 0));
23      end component;
24
25      component binaryToBCD is
26          port(input: in std_logic_vector(7 downto 0); d1, d2: out
                std_logic_vector(3 downto 0));
27      end component;
28
29      component bcdtoseven is
30          port(input: in std_logic_vector(3 downto 0); output: out
                std_logic_vector(6 downto 0));
31      end component;
32
33      component debouncer is
34          port(clk, pb: in std_logic; output: out std_logic);
35      end component;
36
37      -- Signals for Clock and Stopwatch
38      signal c1: std_logic_vector(25 downto 0) :=
            "10111110101111000010000000";
39      signal cstop: std_logic_vector(19 downto 0) :=
            "01111010000100100000";
40      signal cstop2: std_logic_vector(7 downto 0) := "01100100";
41      signal c2, c3, c4: std_logic_vector(7 downto 0) := "00111100",
            "00111100", "00011000";
42      signal f1: std_logic_vector(25 downto 0);
43      signal fstop1: std_logic_vector(19 downto 0);
44      signal f2, f3, f4, fstop2, fstop3: std_logic_vector(7 downto 0);
45      signal o1, o2, o3, o4, ostop1, ostop2, ostop3: std_logic;
46      signal bcd1, bcd2, bcd3, bcd4, bcd5, bcd6, bcdstop1, bcdstop2,
            bcdstop3, bcdstop4: std_logic_vector(3 downto 0);
47      signal seven1, seven2, seven3, seven4, seven5, seven6, seven7,
            seven8, seven9, seven10: std_logic_vector(6 downto 0);
48      signal mpb, hpb: std_logic;
49
50  begin
51      -- Digital Clock
```

```
52      comp1: comparator generic map (n => 26) port map (c1 => f1, c2
            => c1, output => o1);
53      comp2: comparator generic map (n => 8) port map (c1 => f2, c2
            => c2, output => o2);
54      comp3: comparator generic map (n => 8) port map (c1 => f3, c2
            => c3, output => o3);
55      comp4: comparator generic map (n => 8) port map (c1 => f4, c2
            => c4, output => o4);
56
57      -- Debouncing for Digital Clock
58      mdebounce: debouncer port map(clk => clock, pb => incm, output
            => mpb);
59      hdebounce: debouncer port map(clk => clock, pb => inch, output
            => hpb);
60
61      -- Accumulators for Clock
62      acc1: accu generic map (n => 26) port map (reset => o1, clk =>
            clock, output => f1);
63      acc2: accu generic map (n => 8) port map (reset => o2, clk =>
            o1, output => f2);
64      acc3: accu generic map (n => 8) port map (reset => o3, clk =>
            (o2 or mpb), output => f3);
65      acc4: accu generic map (n => 8) port map (reset => o4, clk =>
            (o3 or hpb), output => f4);
66
67      -- Binary to BCD for Clock
68      btobcd1: binaryToBCD port map(input => f2, d1 => bcd1, d2 =>
            bcd2);
69      btobcd2: binaryToBCD port map(input => f3, d1 => bcd3, d2 =>
            bcd4);
70      btobcd3: binaryToBCD port map(input => f4, d1 => bcd5, d2 =>
            bcd6);
71
72      -- BCD to Seven-Segment for Clock
73      bcdtoseven1: bcdtoseven port map(input => bcd1, output =>
            seven1);
74      bcdtoseven2: bcdtoseven port map(input => bcd2, output =>
            seven2);
75      bcdtoseven3: bcdtoseven port map(input => bcd3, output =>
            seven3);
```

```
76      bcdtoseven4: bcdtoseven port map(input => bcd4, output =>
            seven4);
77      bcdtoseven5: bcdtoseven port map(input => bcd5, output =>
            seven5);
78      bcdtoseven6: bcdtoseven port map(input => bcd6, output =>
            seven6);
79
80      -- Stopwatch
81      compstop1: comparator generic map (n => 20) port map (c1 =>
            fstop1, c2 => cstop, output => ostop1);
82      compstop2: comparator generic map (n => 8) port map (c1 =>
            fstop2, c2 => cstop2, output => ostop2);
83      compstop3: comparator generic map (n => 8) port map (c1 =>
            fstop3, c2 => c2, output => ostop3);
84      accstop1: accu generic map (n => 20) port map (reset => ostop1,
            clk => (clock and (not startwatch)), output => fstop1);
85      accstop2: accu generic map (n => 8) port map (reset => (ostop2
            or (not resetwatch)), clk => ostop1, output => fstop2);
86      accstop3: accu generic map (n => 8) port map (reset => (ostop3
            or (not resetwatch)), clk => ostop2, output => fstop3);
87
88      -- Binary to BCD for Stopwatch
89      btobcdstop1: binaryToBCD port map(input => fstop2, d1 =>
            bcdstop1, d2 => bcdstop2);
90      btobcdstop2: binaryToBCD port map(input => fstop3, d1 =>
            bcdstop3, d2 => bcdstop4);
91
92      -- BCD to Seven-Segment for Stopwatch
93      bcdtosevenstop1: bcdtoseven port map(input => bcdstop1, output
            => seven7);
94      bcdtosevenstop2: bcdtoseven port map(input => bcdstop2, output
            => seven8);
95      bcdtosevenstop3: bcdtoseven port map(input => bcdstop3, output
            => seven9);
96      bcdtosevenstop4: bcdtoseven port map(input => bcdstop4, output
            => seven10);
97
98      -- Mode Control
99      process(seven1, seven2, seven3, seven4, seven5, seven6, seven7,
            seven8, seven9, seven10, mode)
```

```
    begin
        case mode is
            when "00" =>
                output1 <= seven1;
                output2 <= seven2;
                output3 <= seven3;
                output4 <= seven4;
            when "01" =>
                output1 <= seven3;
                output2 <= seven4;
                output3 <= seven5;
                output4 <= seven6;
            when "10" =>
                output1 <= seven7;
                output2 <= seven8;
                output3 <= seven9;
                output4 <= seven10;
            when "11" =>
                output1 <= seven7;
                output2 <= seven8;
                output3 <= seven9;
                output4 <= seven10;
        end case;
    end process;
end behv;
```

References

[1] P. S. W. S. Jr., *FPGA-Based Clock Systems*. Berlin, Germany: Springer, 2018.

[2] M. I. Products, "Ds3231 real-time clock (rtc) module," *Maxim Integrated Products*, 2022. [Online]. Available: https://www.maximintegrated.com/en/products/analog/real-time-clocks/DS3231.html

[3] M. Technology, "Ds1307 real-time clock (rtc) module," *Microchip Technology*, 2021. [Online]. Available: https://www.microchip.com/wwwproducts/en/DS1307

Youtube Demo Link