مدينة زويل للعلوم والتكنولوجيا
Zewail City of Science and Technology

# Stopwatch System using HDL

## CIE 239

Prepared For
Dr. Mohamed Samir

Prepared By


| | |
|---|---|
| Ibrahim Hamada | 201800739 |
| Ehab Mansour | 201800506 |
| Areej Mohamed | 201801902 |
| Nasrah Mohamed | 202000925 |
| Yasmeen Wael Abosaif | 202001116 |

# Table of content :

# Introduction

In this project, we are going to design, implement, and verify the functionality of a stopwatch system using HDL (SystemVerilog IEEE 1800-2017). The stopwatch is designed to display 4 digits, 2 for minutes and 2 for seconds: M1M0:S1S0 . The minimum starting value of the stopwatch is 10:20, while the maximum value of this stopwatch is 49:30.

The stopwatch has multiple inputs as follows:

a) DIP switch for starting and pausing the count operation.
b) DIP switch for toggling between the count up and count down modes.
c) DIP switch for speeding up the watch rate so that the watch will count two digits each second.
d) DIP switch for slowing down the watch rate so that the watch will advance one digit each two seconds.
e) Push button for resetting.
f) Push button for adding (or subtracting) 2 minutes each time it is pressed (add2 button). The adding or subtracting is allowed while counting and while paused.
    - Count-up mode: adding 2 minutes each time it is pressed.
    - Count-down mode: subtracting 2 minutes each time it is pressed.

# Possible Design Approaches

There are different design approaches to implement the digital stopwatch system that arise from having different types of counters. Moreover, the digital stopwatch control mechanism can be implemented using different approaches.

## ● Types of Counters:

Counters are sequential digital logic devices that are used to count the number of produced pulses by the input clock. The counters should possess memory elements, therefore they can be implemented by using register type circuits like flip-flops. Since the counters are sequential digital logic devices, they can be designed using two approaches: synchronous or Asynchronous.

## 1) Synchronous Counter:

Synchronous Counter is a digital circuit that counts in binary numbers with the use of flip-flops where the input clock for all used flip-flops is the same. Accordingly, the flip-flops are all clocked at the same time with the same signal.

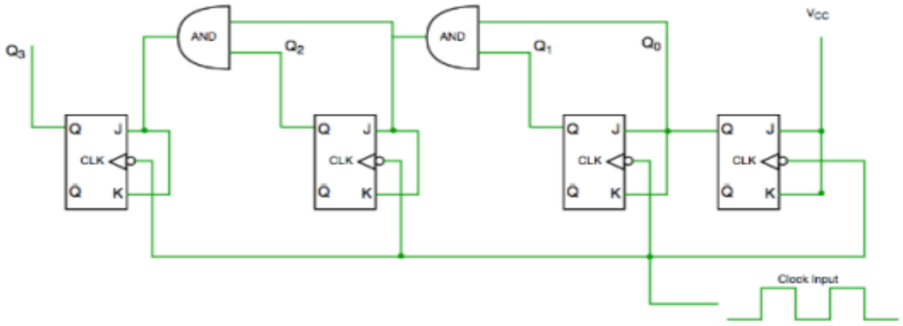


**Figure 1. Synchronous counter circuit**

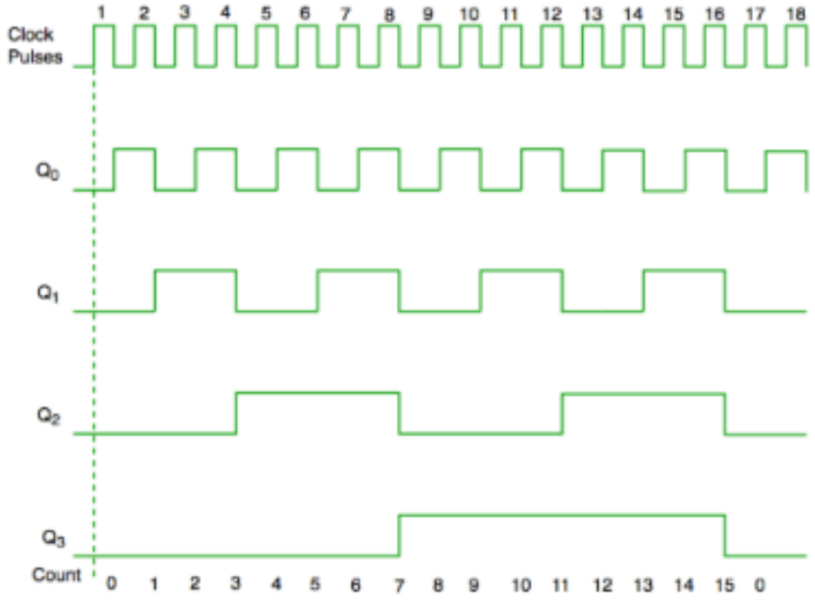


**Figure 2. Timing Diagram of Synchronous counter**

- **Important points about synchronous counter:**

a) The flip flops in a synchronous counter are triggered simultaneously by the same clock
b) In operation, synchronous counters are faster than asynchronous counters
c) There are no decoding errors with Synchronous Counters
d) Parallel counters are also referred to as synchronous counters
e) Due to the increasing number of states, designing and implementing synchronized counters can be complex.

f) Any count sequence can be used with a synchronized counter.
g) Examples of synchronized counters are: **Ring counter and Johnson counter**.
h) In synchronous counters, propagation delay is less.

- **Process of design synchronous counter:**

a) Decide the number of flip-flops.
b) Excision table of flip-flops.
c) State diagram and excitation table.
d) Obtain simplified eq. using k maps.
e) Draw the logic diagram.

- **JK flip-flop:**

JK flip-flops are a type of synchronous counter component that prevent invalid output conditions by incorporating clock input circuitry. They have four input combinations and can offer advantages over clocked SR flip-flops, but may experience timing issues known as "races" if the output Q changes before the clock pulse ends. These issues can be avoided by using high frequency clock pulses, or by implementing master-slave (edge-triggered) flip-flops, which are more stable.

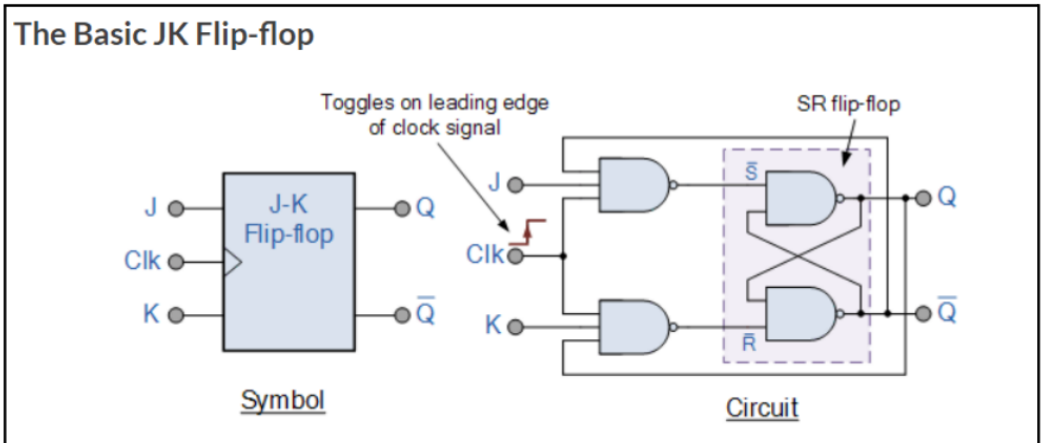


**Figure 3. JK flip flop circuit**

- **Master-Slave JK Flip-flop:**

A master-slave flip-flop is a type of digital circuit that uses two SR flip-flops connected in series to prevent timing issues. One flip-flop, known as the "master," is triggered on the rising edge of the clock pulse, while the other flip-flop, known as the "slave," is triggered on the falling edge. This allows the circuit to operate smoothly and avoid problems such as "races," in which the output Q changes state before the clock pulse has ended. Master-slave flip-flops are often used in cases where high-frequency timing pulses are required, but are not possible with basic NAND or NOR gates.

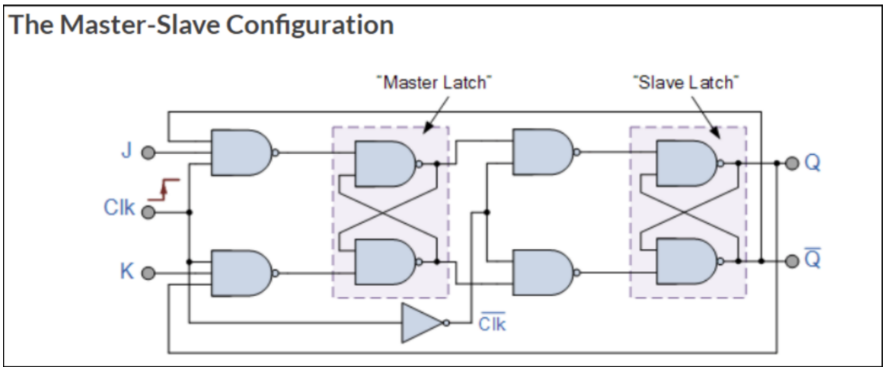


**Figure 4. Master-Slave JK Flip-flop circuit**

JK flip-flops are popular components in synchronous counters and consist of gated SR flip-flops with clock input circuitry that prevents invalid output conditions caused by logic level 1 inputs. They have four input combinations due to the clocked input: logic 1, logic 0, no change, and toggle. Master-slave flip-flops were developed to improve stability, as JK flip-flops can sometimes suffer from timing issues called "races" when the output Q changes state before the clock pulse goes "off". In a master-slave flip-flop, two SR flip-flops are connected in series to eliminate timing problems. The "master" flip-flop triggers on the leading edge of the clock pulse and the "slave" flip-flop triggers on the falling edge. When the clock input is "HIGH", the inputs J and K are "locked" to the gated "master" SR flip flop and the "slave" flip flop does not toggle. Master-slave JK flip-flops transmit data only at the timing of the clock signal and are therefore synchronous devices.

2) **Ring Counter:**

A ring counter is a type of shift register in which the output of the last flip-flop is connected to the input of the first flip-flop. It functions similarly to a shift counter, but has a looped structure. The number of states in a ring counter is equal to the number of flip-flops used in the counter. For example, a 4-bit ring counter would require 4 flip-flops.

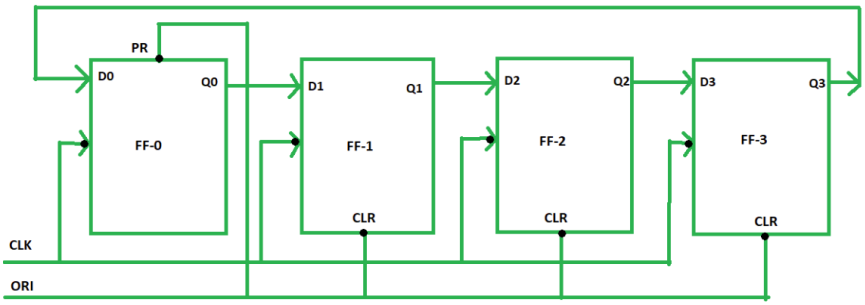


**Figure 5. Ring Counter Circuit**

If PR is 1, the preset input has no effect on the output of the flip-flop. Similarly, when CLR is 1, the clear input has no effect on the output of the flip-flop. A synchronous counter using flip-flops with preset and clear inputs is called a programmable counter. This allows the user to set the initial count value and reset the counter to zero whenever desired.

- PR = 0, Q = 1
- CLR = 0, Q = 0

- **Truth Table:**

| ORI | CLK | $Q_0$ | $Q_1$ | $Q_2$ | $Q_3$ |
|---|---|---|---|---|---|
| Low | X | 1 | 0 | 0 | 0 |
| High | low | 0 | 1 | 0 | 0 |
| High | low | 0 | 0 | 1 | 0 |
| High | low | 0 | 0 | 0 | 1 |
| High | low | 1 | 0 | 0 | 0 |

**Table 1. Ring Counter Truth Table**

The ring counter is a type of digital circuit that is created using a series of flip-flops connected in a ring configuration. It is similar to a shift register, with the output of the last flip-flop being connected to the input of the first flip-flop. The ring counter has a number of states equal to the number of flip-flops used in the circuit, and can be preset to a particular state by using the override input (ORI). The clock pulse is applied to all flip-flops simultaneously, making the ring counter a synchronous circuit. The ORI input, when made low, generates a "1" and the clock becomes a "don't care" signal. After ORI is made high and a low clock pulse is applied, the "1" is shifted to the next flip-flop, creating a ring of states. In a 4-bit ring counter, there are 4 states: "1000", "0100", "0010", and "0001".

### 3) Asynchronous Counter:

An asynchronous counter is a type of digital circuit that uses flip-flops to count, but each flip-flop receives its clock signal from the output of the previous flip-flop rather than from a shared system clock. This means that the outputs of the flip-flops do not change at the same time, resulting in different propagation delays for each flip-flop. There are several types of asynchronous counters, including 4-bit synchronous up counters, 4-bit synchronous down counters, and 4-bit synchronous up/down counters.

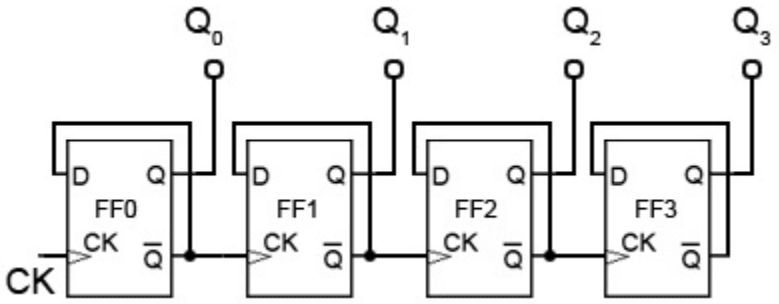


**Figure 6. Asynchronous Counter Circuit**

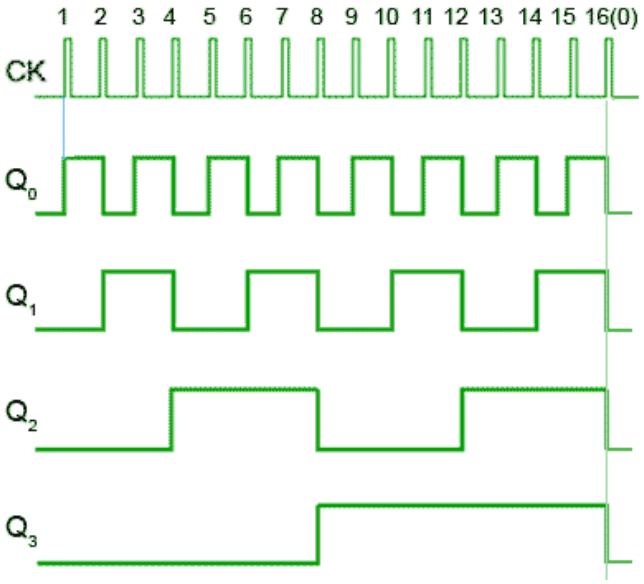


**Figure 7. Timing diagram of Asynchronous counter**

- **Clock Ripple**

In a circuit driven by the time delay of individual clock pulses, called a "clock ripple," the propagation delays of logic gates add to the delay of the next flip flop. The total of these delays is known as the propagation delay of the circuit. As the outputs of all flip-flops change at different time intervals, a new value occurs at the output each time there is a different input at the clock signal.

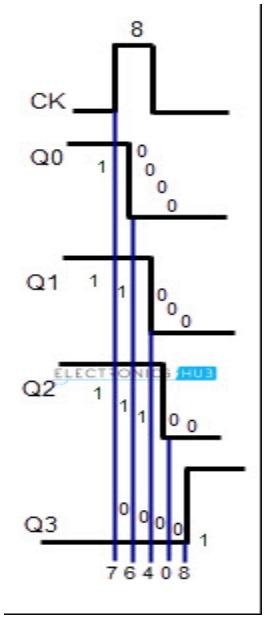


**Figure 8. flip flop's propagation delay**

| Advantage | Disadvantage |
|---|---|
| • The T flip flop and D flip flop can easily be used to design asynchronous counters. | • It is possible to require an extra flip flop during “Re synchronization ”. |
| • In low-speed circuits, these counters are also called ripple-counters. | • Additional feedback logic is needed to count the sequence of truncated counters (mod is not equal to 2n). |
| • A Divide by-n counter divides an input by n, where n is an integer. | • When counting large numbers of bits, asynchronous counters have a very long propagation delay. |

**Table 2. Advantages and disadvantages of asynchronous counters**

# Selected Approach

As per the information introduced in the previous section, we decided to use Synchronous counters. It is not only for the sake of synchronization and less propagation delay, but also it is more feasible and doable to design using an FPGA. When we searched about the implementation of Asynchronous circuits using FPGA, we found that it is preferable and necessary to implement synchronous circuits on FPGA because most synthesis and verification tools are based on Static Timing Analysis (STA) and the vast majority of libraries are built with synchronous techniques.

Moreover, Synchronous counters have several advantages over asynchronous counters:

1. **Speed:** Synchronous counters are faster than asynchronous counters because they are clocked, which means that all the flip-flops in the counter are triggered by a common clock signal. This allows the counter to operate at the maximum speed of the clock, whereas asynchronous counters have to wait for the input signal to propagate through the entire circuit before the output is updated.

2. **Timing:** Synchronous counters have a predictable timing behavior because they are clocked, which means that the output changes at specific intervals based on the clock frequency. This makes it easier to design and debug the counter. Asynchronous counters, on the other hand, have unpredictable timing because the output changes based on the arrival of the input signal, which can vary depending on the input frequency and the circuit delay.

3. **Synchronization:** Synchronous counters are easier to synchronize with other circuits because they are clocked, which means that they can be easily aligned to a common clock signal. Asynchronous counters, on the other hand, can be more difficult to synchronize because they do not have a common clock reference.

4. **Glitch-free operation:** Synchronous counters are less prone to glitches because the outputs are updated simultaneously on the rising edge of the clock. Asynchronous counters, on the other hand, can experience glitches if the input signal changes during the propagation delay through the circuit.

Accordingly, we decided to design a counter that is based on a 4-bit register with a Full adder that increments the current value of the register with one at each clock positive edge. Moreover, we depended on the combinational logic components to design the control unit in order to satisfy all the requirements of the digital stopwatch .

The following components are used in our design:

1. D-flip-flop→ to build up the 4-bit register that is used in each of the 4 main elements in the counter (Minute_1 - Minute_0 : Second_1 - Second_1)
2. JK-flip-flop→ the flip flop is used to produce 4 bits each of 4 seconds when we have error 0 1 0 1, we used 0 1 to have flashing where 0 to don't show anything 1 to show f3:f3
3. Register → to store the count values.
4. Adder→ to increment 1 to the current value in the register.
5. Subtructor → to find the result of subtracting two 4-bits numbers and it is used in the comparator module
6. Comparator → to check whether we reached the max count value or not.
7. Mux → to choose among different options in the circuit.
8. Clock divider → to speed up and slow down the clock of the FPGA.
9. AND - OR Gates → to build up the control unit to handle all requirements.
10. Error Code 1 → to detect whether the mode is changed from up to down or from down to up while counting.
11. Seven Segment → to display the output of each clock's digit (S0, S1, M0, M1)
12. Seven Segment Decoder → to decode the 4-bits output of each register to 7-bits output in order to be displayed on the seven segment display.

The previously mentioned components are used as the main building blocks of the system as they are used in nearly each module. In the following sections, a brief description of each component is introduced.

1. **D flip-flop :**

Digital electronic circuits (D or Delay) Flip Flop delay the state change of their output signals (Q) until the next rising edge on the clock timing input signal. Since the D Flip Flop's output remains constant unless the D input is altered and a rising clock signal is applied, it acts as an electronic memory component. The D Flip Flop is the register building piece. A simple divide-by-two circuit is created by connecting the inverting output of the D Flip Flop to the D input, where the D output changes state at half the frequency of the clock signal. A countdown timer can be constructed by cascading D flip flops and using combinational logic gates.

| D | CLK | $\bar{Q}$ |
|---|---|---|
| 0 | Rising Edge | 0 |
| 1 | Rising Edge | 1 |

**Figure 9. D flip-flop truth table**

2. **Mux:**

A data selector that takes several inputs and generates a single output is called a multi-input selector. In this case, the input lines are 2n, and the output lines are 1. The number n refers to the total number of selected lines. Despite accepting multiple data inputs, a multiplexer produces only one output. Multiplexing involves transmitting various digital input signals, analog signals, or streams of data over a single channel. This technique combines several low-speed channels into one high-speed channel. As a result, the high-speed channel is effectively utilized. A carrier can save operating costs by eliminating the maintenance of multiple lines by using multiplexing communication. Multiplexers are electronic devices that perform multiplexing. This component combines multiple analog or digital signals (inputs) into one transmission line.

3. **Register:**

A register consists of flip-flops that store binary data. Shift registers are capable of shifting bits either towards the right or left sides. The information stored within these registers can be transferred with the help of shift registers. Shift Register is a group of flip flops used to store multiple bits of data. The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses. An n-bit shift register can be formed by connecting n flip-flops where each flip-flop stores a single bit of data. The registers which will shift the bits to the left are called “Shift left registers”. The registers which will shift the bits to the right are called “Shift right registers”. Shift registers are basically of 4 types: Serial In Serial Out shift register, Serial In parallel Out shift register, Parallel In Serial Out shift register, Parallel In parallel Out shift register. We are using a Parallel In parallel Out shift register which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and also produces a parallel output known as the Parallel-In parallel-Out shift register.

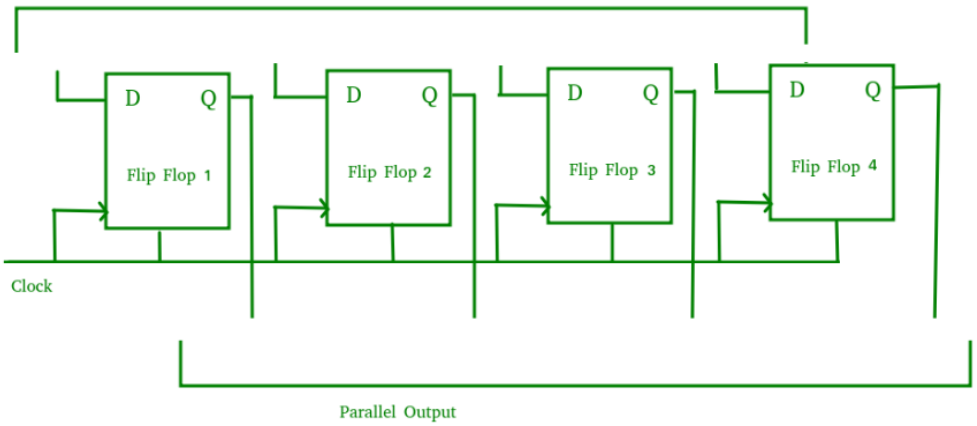


**Figure 10. Parallel-In parallel-Out shift register**

4. **Seven Segment :**

Seven segment displays are commonly used in electronic devices to display numerical information. They are called "seven segment" displays because they are made up of seven separate segments that can be lit up to form numerical digits. Each segment is a separate LED (light emitting diode) or LCD (liquid crystal display) element, and the combination of lit segments can be used to display all ten digits from 0 to 9, as well as a few letters and special characters. Seven segment displays are often used in devices such as digital clocks, calculators, and electronic displays for automobiles. They are simple, easy to read, and relatively inexpensive, which makes them a popular choice for displaying numerical information in a variety of applications.

- **Seven Segment Displays:**

A seven-segment display is an alternative to dot matrix displays for the display of images, text, or decimal numbers as well as many other types of information. Many electronic devices that display numerical information use it, including digital clocks, basic calculators, electronic meters, and other devices that display numerical data. The light-emitting diodes (LEDs) are arranged in seven segments, just like numerical 8.

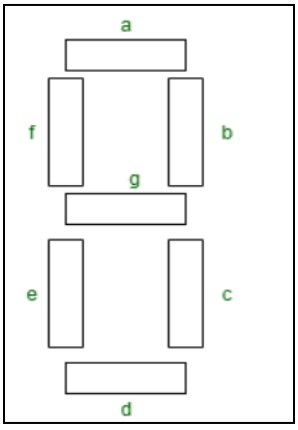


**Figure 11. Seven segment display**

- **Working of Seven Segment Displays:**

Different pins of a seven-segment display can be powered (or energized) at the same time, so multiple numerical combinations can be displayed. When all segments are powered, the number 8 is displayed, but if you disconnect the power for 'g', the number 0 is displayed. As seven-segment displays cannot form alphabets like X and Z, they cannot be used for alphabets and can only display decimal numerical magnitudes.

DPs are usually located to the left or to the right of the display pattern on each display unit. This type of pattern can be used to display numerals from 0 to 9 and letters from to F hexadecimal digits.

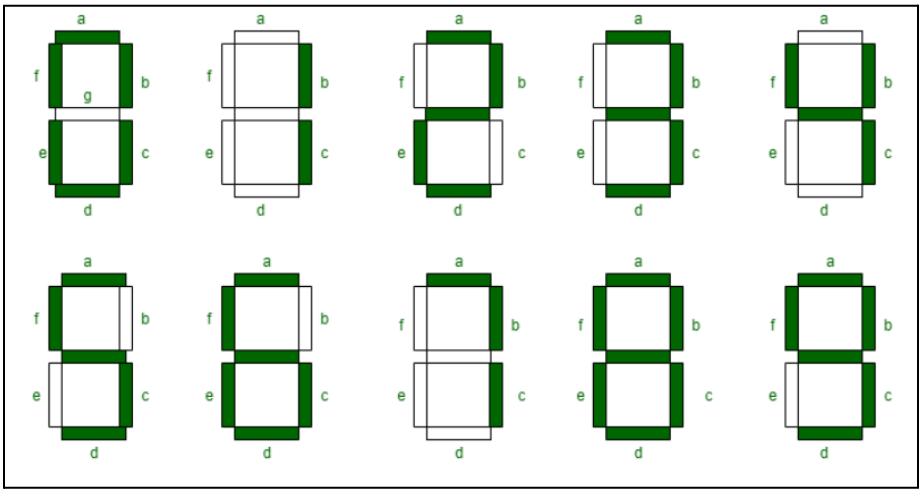


*figure* 12 : *Working of Seven Segment Displays*:

- **Truth table**

| Decimal Digit | Individual Segments Illuminated | | | | | | |
|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | f | g |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

*Table* 3 : *truth table of seven segment display*

There are seven segment displays that show the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 by turning on and off light-emitting diodes (LEDs). To communicate with these external devices, such as switches, keypads, and memory, microcontrollers are used. LEDs require the same number of segments for each digit to function as there are segments per digit.

- **Types of Seven Segment Displays:**

Seven-segment displays can be configured in two ways depending on the application: as a common anode display or as a common cathode display.

1. LED segments are connected together to logic 0 or ground in ***common cathode seven segment displays***. When biasing the anode terminals "a" to "g", logic 1 is used through a current limiting resistor.
2. All LED anode connections in a ***common anode seven segment display*** are connected to logic 1. The cathode of segments a to g is applied to logic 0 via a current limiting resistor.

- **The hardware component:**

We used **5011AS** seven segments, which is a common cathode seven segment. We connected the common cathode to the GND Pin of the FPGA, while the other pins (a, b, c, d, e, f, g) are attached to series resistors to limit the current and then connected to GPIO pins.

The link for datasheet

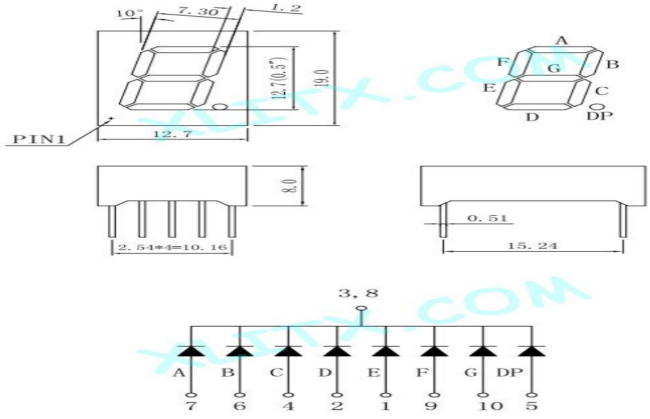


*figure* 13: 5011*as* 7 *segment*

5. **Seven Segment Decoder:**

A seven segment decoder is a digital circuit that converts a binary or BCD input into a form that can be displayed on a seven segment display. Seven segment displays are commonly used in digital clocks, calculators, and other electronic devices to display numerical information. The input to a seven segment decoder is typically a binary number, which is represented by a group of digital signals. Each of these signals corresponds to a single digit in the binary number. For example, a 4-bit binary number can be represented by four digital signals, each of which can be either high (1) or low (0).

The output of a seven segment decoder is a set of digital signals that control the segments of a seven segment display. Each segment of the display corresponds to a different part of the number being displayed, such as the top, bottom, left, or right part of a digit. By activating the appropriate segments, the decoder can display any number from 0 to 9 on the seven segment display. There are several types of seven segment decoders, including combinational and sequential decoders. Combinational decoders provide a direct mapping from the input signals to the output signals, while sequential decoders use memory elements such as flip-flops to store and output the decoded information. Seven segment decoders are widely used in digital electronics because they provide a simple and efficient way to display numerical information on a seven segment display. They are commonly used in a variety of applications, including digital clocks, calculators, and other electronic devices that require the display of numerical data.

# System Design

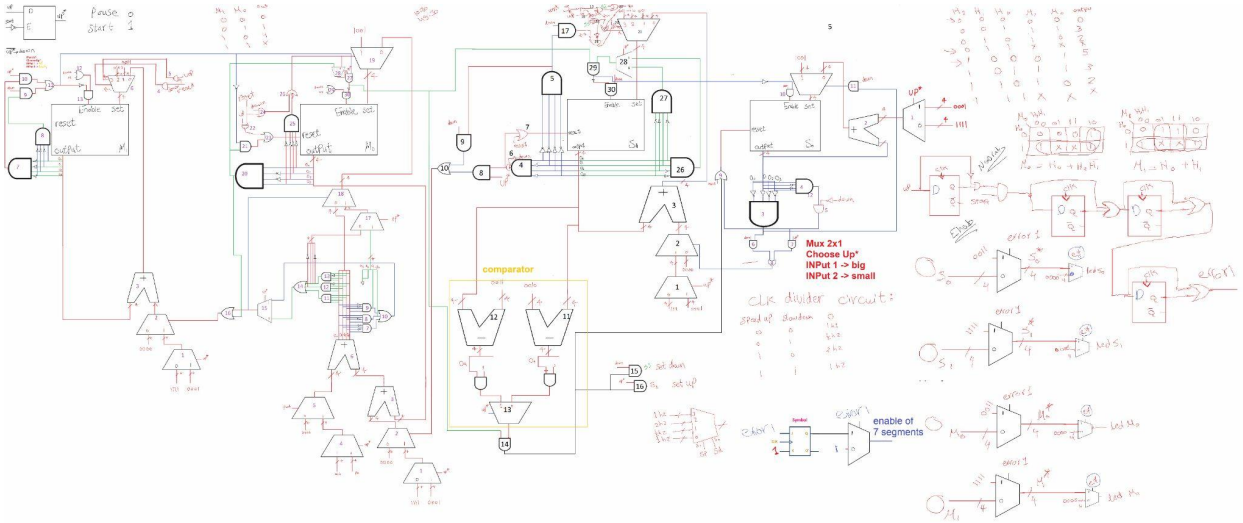


**Figure 14. System Circuit**

There are four 4-bit registers: S0, S1, M0, M1. S0 is responsible for counting the units of seconds, while S1 is responsible for counting the tens of seconds. On the other hand, M0 is responsible for counting the units of minutes, and M1 is responsible for counting the tens of minutes. For each one of the four registers, there is a certain combinational logic for Enable, Set, and Reset signals. The stopwatch counts from 10:20 to 49:30 and vice versa based on the selected option (up/down).

1. S0:
    a) **Enable:** if **start (not pause)** switch is pressed and we are in the allowed range (10:20 - 49:30)

    b) **Reset:** if the **reset** switch is pressed, or the counter reaches 9 if we are counting up, or we are out of the allowed range (10:20 - 49:30).

    c) **Set:** In **Up** Mode it starts with 0 and increments 1 at each clock positive edge. In **down** mode, it starts with 9 and decrements 1 at each clock positive edge.

2. S1:

    a) **Enable:** if **start (not pause)** switch is pressed and we are in the allowed range (10:20 - 49:30)

    b) **Reset:** if the **reset** switch is pressed, or the counter reaches 6 if we are counting up, or we are out of the allowed range (10:20 - 49:30).

    c) **Set:** In **Up** Mode it starts with 2 and increments 1 when it receives signal from S0 that it reached 9. In **down** mode, it starts with 3 and decrements 1 when it receives a signal from S0 that it reached 0.

3. M0:
    a) **Enable:** if **start (not pause)** switch is pressed and we are in the allowed range (10:20 - 49:30)

    b) **Reset:** if the **reset** switch is pressed. Or the counter reaches 9 if we are counting up, and we are out of the allowed range (10:20 - 49:30).

    c) **Set:** In **Up** Mode it starts with 0 and increments 1 when it receives a signal from S1 that it reached 6. In **down** mode, it starts with 9 and decrements 1 when it receives a signal from S1 that it reached 0.

4. M1:
    a) **Enable:** if **start (not pause)** switch is pressed and we are in the allowed range (10:20 - 49:30)

    b) **Reset:** if the **reset** switch is pressed. Or the counter reaches 9 if we are counting up, and we are out of the allowed range (10:20 - 49:30).

    **c) Set:** In **Up** Mode it starts with 1 and increments 1 when it receives a signal from M0 that it reached 9. In **down** mode, it starts with 4 and decrements 1 when it receives a signal from M0 that it reached 0.

5. Clock divider:

    This module is responsible for implementing speed up and slow down features. It selects between 3 clocks: normal clock (1 Hz), fast clock (0.5 Hz), slow clock (2 Hz).

**6. STOPWATCH:**

After implementing the above modules, we integrated them together in the STOPWATCH module. This module has the main system's inputs: input_clk, reset, start, up, plus_minus_2, speedup, slowdown. On the other hand, it outputs the following signals: output_S0, output_S1, output_M0, and output_M1 where each of them is a 4-bits output.

**7. Error Code 1:**

The implementation of the error code is separated from the implementation of the circuit where :

a) The first D flip flop used to identify if there is change in the value of the up value button.
b) Then it is anded with the start to give "1" just in case the up button is changed when it is counting
c) It is followed by 3 flip flops in order to send four "1" signals in four second ; two seconds off and two on, to represent flashing
d) This output "error output" is used as a selector for four muxes
e) The first mux takes " the output of s0" and " 0011"
f) The second mux takes "the output of s1" and "1111"
g) The third mux takes "the output of m0" and "0011"
h) The fourth mux takes "the output of m1" and "1111"
i) If the "error output" is 1 , so it will choose "0011" in "s0 and m0" and " 1111" in "s1 and m1"
j) Continuously, it gives output "1" four times without stopping the clock.

## 8. Seven Segment Decoder:

While implementing the decoder, we took the "error code 1" message (F3:F3) into consideration. That's because our seven segment will not only be displaying values from 0 to 9, but also it needs to provide the flashing feature (F3:F3 → OFF → F3:F3 → OFF) if an error is detected.

Therefore, the decoder's truth table is shown below. Two more values are added to the truth table rather than values 0-9. These values represent the 4-bits output of the Error detection module and they are encoded as follows:

1. **1010**: represents **OFF**.
2. **1111**: represents "**F**".

| **Input signal** | **A3** | **A2** | **A1** | **A0** | **a** | **b** | **c** | **d** | **e** | **f** | **g** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 10 **[error]** | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 **[error]** | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

*Table 4 : truth table of seven segment decoder*

After constructing the truth table, the equations for each input (a, b, c, d, e, f, g) is obtained using K-maps. The seven equations are listed below:

$$a = A_3` A_1 + A_2 A_0 + A_3` A_2` A_0`$$
$$b = A_3` A_2` + A_2` A_1` + A_3` A_1 A_0 + A_1` A_0`$$
$$c = A_1` + A_3` A_0 + A_3` A_2$$
$$d = A_3 A_1` + A_3` A_2` A_0` + A_3` A_2` A_1 + A_3` A_1 A_0` + A_2 A_1`$$
$$e = A_3 A_2 + A_2` A_1` A_0` + A_3` A_1 A_0`$$
$$f = A_1` A_0` + A_2 A_1` + A_2 A_0` + A_3 A_0$$
$$g = A_2 A_1` + A_2 A_0` + A_3 A_1` + A_3 A_0 + A_3` A_2` A_1$$

9. **STOPWATCH with Seven Segment Decoder:**

After testing the functionalities of the **Error code 1** module and the seven segments decoder separately, they are combined together with the stopwatch module in a way that converts the 4-bits outputs of each register to 7-bits in order to be displayed on the seven segment while taking the error code condition into account. Therefore, this module calls the STOPWATCH module and passes its outputs to the decoder to be decoded.

# Implemented Modules

After designing the circuit, we started implementing and simulating the needed modules using systemVerilog language and Modelsim software. The implemented modules along with simulation results are listed below.

1) **4-Input AND Gate:**

This module is used to find the result of ANDing 4 bits. The module stores the AND of two inputs in a wire firstly using the 2-input AND, then the other two inputs. Finally, it ANDs the resulting two wires together using another 2-input AND gate.

```
/* This module implments a 4-inputs AND Gate that and the result is stored in y */
module AND4 (a0, a1, a2, a3, y);
input logic a0, a1, a2, a3; //Input 4 bits a0-a3
output logic y;             //Output y = a0 & a1 & a2 & a3

wire w1, w2;

// 3 2-inputs AND gates are used to implment the 4-inputs AND gate following the structural design.
and M1(w1, a0, a1);
and M2(w2, a2, a3);
and M3(y, w1, w2);

endmodule
```

**Figure 15. AND4 Module Implementation**

- Simulation Output:

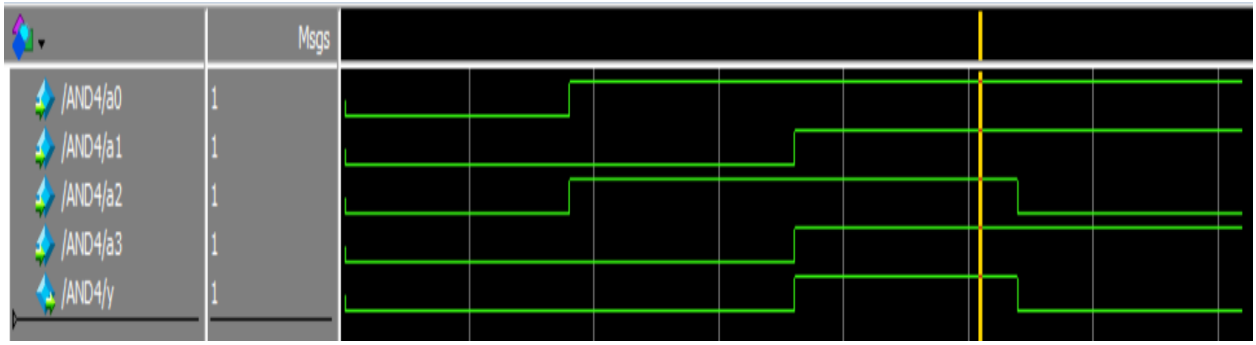


**Figure 16. AND4 Module Simulation**

As shown in the figure, the module outputs ‘1’ if and only if the 4 inputs are ‘1’. Otherwise, it outputs ‘0’.

2) **3-Input OR Gate:**

This module is used to find the result of ORing 3 bits. The module stores the OR of two inputs in a wire firstly using the 2-input OR, then ORing the result with the third input.

```
/* This module implments a 3-inputs OR Gate that stores the result is stored in y */
module OR3(A, B, C, y);

input logic A, B, C;  //Inputs
output logic y;       //Output y = A | B | C
wire w1;

// 2 2-input or gates are used to find the result of ORing 3 bits
or M1(w1, A, B);
or M2(y, w1, C);

endmodule
```

**Figure 17. OR3 Module Implementation**

- Simulation Output:

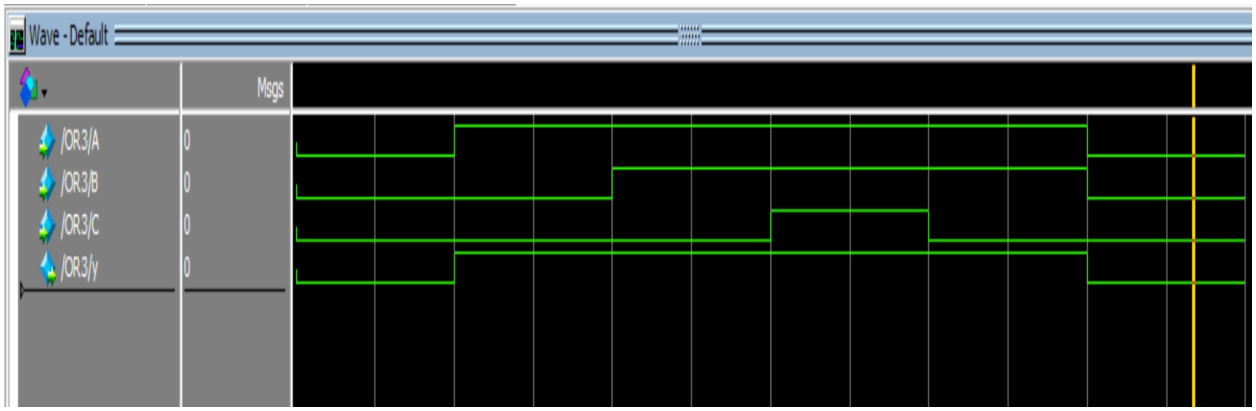


**Figure 18. OR3 Module Simulation**

As shown in the figure, the module outputs ‘0’ if and only if the 3 inputs are ‘0’. Otherwise, it outputs ‘1’.

### 3) 4-Input NOR Gate:

This module is used to find the result of NORing 4 bits. The module uses 3 2-input OR gates to find the ORing of the 4 inputs, then it inverts the result using a NOT gate. The module stores the ORing of two inputs in a wire firstly using the 2-input OR, then the other two inputs. After that, it ORs the resulting two wires together using another 2-input OR gate. Finally, it inverts the result to find the NOR.

```
/* This module implments a 4-inputs NOR Gate that stores the result is stored in y */
module NOR4 (a0, a1, a2, a3, y);
input logic a0, a1, a2, a3;  //Input 4 bits
output logic y;              //Output: y = (a0 + a1 + a2 + a3)'

wire w1, w2, w3;

//The module uses 3 2-input OR to find the result of ORing 4 bits
or M1(w1, a0, a1);
or M2(w2, a2, a3);
or M3(w3, w1, w2);
//OR of the four inputs is inverted to find the NOR
not M4(y, w3);

endmodule
```

**Figure 19. NOR4 Module Implementation**

- Simulation Output:

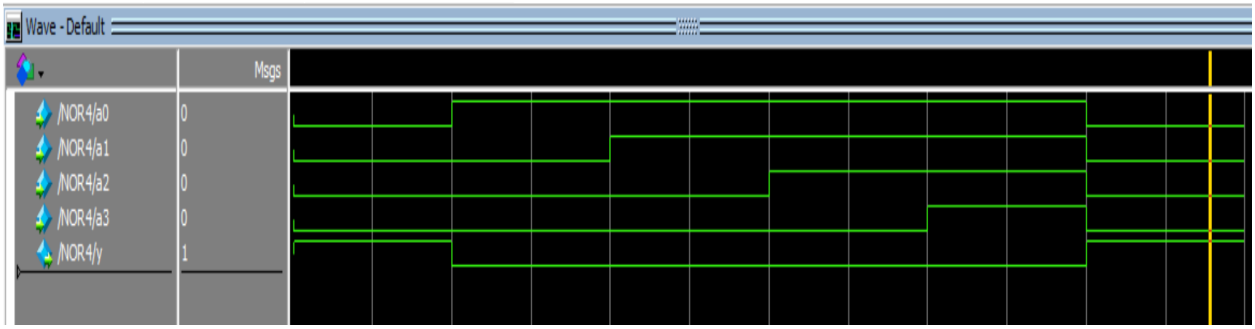


**Figure 20. AND4 Module Simulation**

As shown in the figure, the module outputs ‘1’ if and only if the 4 inputs are ‘0’. Otherwise, it outputs ‘0’.

4) **MUX2:**

This module is used to select between 2 inputs. The module is parameterized using parameter WIDTH in order to use it with any n-bits inputs. The module outputs ‘a1’ if the select line is ‘1’ and outputs ‘a0’ if the select line is ‘0’. This module will be used to implement the MUX4 module following the structural design approach.

```
/* This module implments a 2:1 MUX that selects between 2 1-bit inputs  */
module MUX2_1S(a, b, sel, y);
input logic a, b, sel;   //inputs
output logic y;          //outputs
wire w1, s_bar, w2;
// Select between 2 bits
and M1(w1, b, sel);
not M2(s_bar, sel);
and M3(w2, a, s_bar);
or M4(y, w1, w2);

endmodule
```

**Figure 21a. MUX2 Module Implementation (Select between 1-bit 2 inputs)**

```
/* This module implments a 2:1 MUX that selects between 2 4-bits inputs  */
module MUX2_14S(a, b, sel, y);
input logic [3:0] a, b;  //Inputs
input logic sel;         //Selection
output logic [3:0] y;    //Output
//Selecting between 4-bits
MUX2_1S M1(a[3], b[3], sel, y[3]);
MUX2_1S M2(a[2], b[2], sel, y[2]);
MUX2_1S M3(a[1], b[1], sel, y[1]);
MUX2_1S M4(a[0], b[0], sel, y[0]);

endmodule
```

**Figure 21b. MUX2 Module Implementation (Select between 4-bits 2 inputs)**

- Simulation Output:

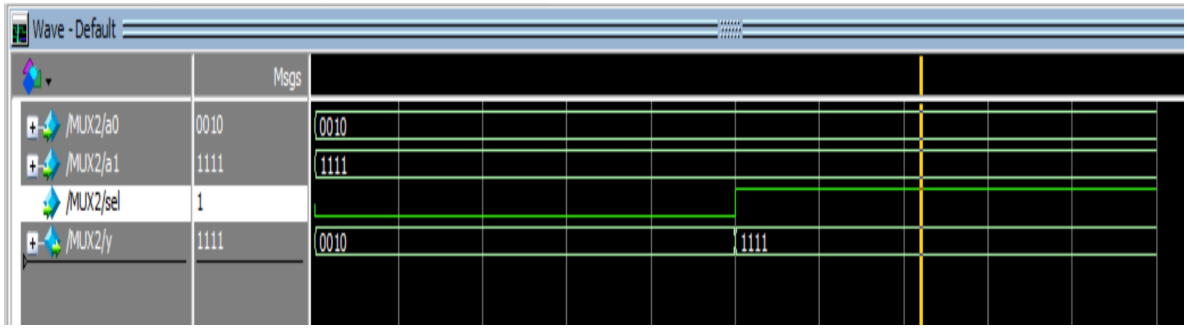


**Figure 22. MUX Module Simulation**

As shown in the figure 8, the module outputs (‘a0’ =‘0010’) when the select is ‘0’ and outputs (‘a1’ = ‘1111’) when the select is ‘1’.

### 5) MUX4:

This module is used to select between 4 inputs. The module is parameterized using parameter WIDTH in order to use it with any n-bits inputs. The module outputs ‘a0’ if the select line is ‘00’, ‘a1’ if the select line is ‘01’, ‘a2’ if the select line is ‘10’, and ‘a3’ if the select line is ‘11’. This module uses 3 MUX2 blocks to select the output. The first select bit is used to select between a0, a1 and a2, a3. After that, the second select bit selects between the result of the previous 2 blocks.

```
/* This module implments a parameterized 4:1 MUX that selects between 4 n-bits inputs  */
module MUX4(a0, a1, a2, a3, sel, y);

parameter WIDTH=4; //Number of bits of the inputs

input logic [WIDTH-1:0] a0, a1, a2, a3;  //Inputs
input logic [1:0] sel;                   //Select
output logic [WIDTH-1:0] y;              //Output

wire [3:0] y1, y2;

//3 MUX2 blocks are used to select 1 out of the 4 inputs.
MUX2 #(WIDTH) M1(a0, a1, sel[0], y1);
MUX2 #(WIDTH) M2(a2, a3, sel[0], y2);
MUX2 #(WIDTH) M3(y1, y2, sel[1], y);

endmodule
```

**Figure 23. MUX4 Module Implementation**

- Simulation Output:

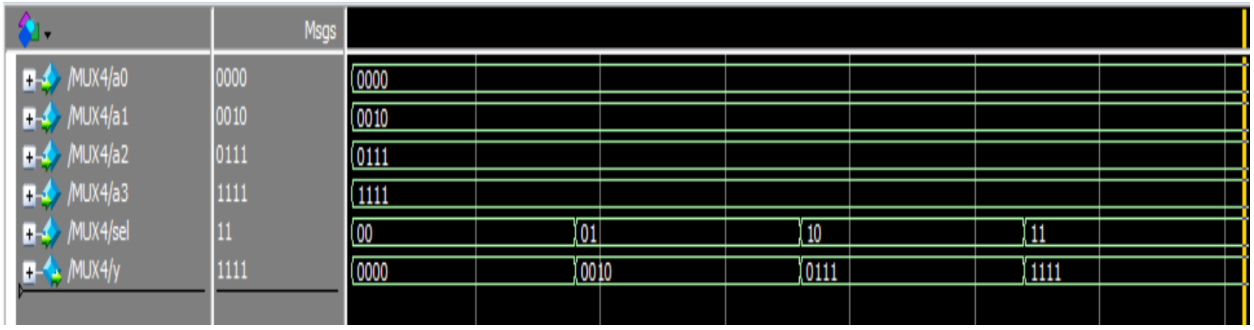


**Figure 24. MUX4 Module Simulation**

As shown in the figure 10, the module outputs (‘a0’ = ‘0000’) when the select is ‘00’, outputs (‘a1’ = ‘0010’) when the select is ‘01’, outputs (‘a1’ = ‘0111’) when the select is ‘10’, and outputs (‘a1’ = ‘1111’) when the select is ‘11’.

## 6) Full Adder:

This module is used to find the result of Adding 2 bits while having a carry input value. The module implements the equations of sum and carry values as follows:

- Sum = A xor B xor Cin
- Carry_Out = (A and B) or (A and Cin) or (B and Cin)

There are 2-input xor gates to find the sum of the bits. 3 2-input AND gates and 2 2-input OR gates are used to find the output carry.

```
/* This module implments a Full Adder that stores the sum and Cout of the inputs */
module FA(Cin, A, B, Cout, S);
input logic Cin, A, B;   //Inputs
output logic Cout, S;    //Outputs: Sum and Carry
wire w1, w2, w3, w4, o5;

//Find the Sum = A ^ B ^ Cin
xor M1(w1, A, B);
xor M2(S, w1, Cin);

//Find the Carry out = A*B + A*Cin + B*Cin
and M3(w2, A, B);
and M4(w3, A, Cin);
and M5(w4, B, Cin);

or M6(o5, w2, w3);
or M7(Cout, o5, w4);

endmodule
```

**Figure 25. FA Module Implementation**

- Simulation Output:

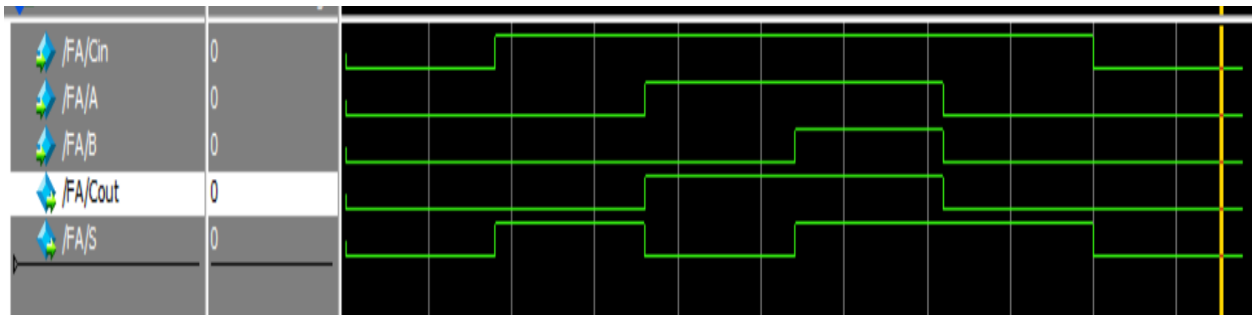


**Figure 26. FA Module Simulation**

As shown in the figure, the module outputs

- S= ‘1’ and Cout = ‘1’ if A and B and Cin = ‘1’.
- S= ‘0’ and Cout = ‘0’ if A and B and Cin = ‘0’.
- S= ‘1’ and Cout = ‘0’ if only one input is ‘1’.
- S= ‘0’ and Cout = ‘1’ if only two inputs are ‘1’.

### 7) 4-input Full Adder Gate:

This module is used to find the result of Adding 2 inputs each of them with 4 bits. There are 4 FA blocks that are used to add each 2 bits as follows.

- FA0 : A0 + B0 + Cin = S0, C0
- FA1: A1 + B1 + C0 = S1, C1
- FA3: A2 + B2 + C1 = S2, C2
- FA3: A3 + B3 + C2 = S3, Cout

The underlined symbols correspond to the outputs of the block: Sum and Output Carry.

```
/* This module implments a 4-bit Full Ripple Carry Adder that stores the sum and Cout of the inputs */
module FA4(Cin, A, B, Cout, S);
input logic [3:0] A, B;  //Input 4-bits
input logic Cin;         //Carry in
output logic [3:0] S;    //Sum
output logic Cout;       //Carry out

wire c1, c2, c3;

//The module uses 4 FA blocks to add each corresponding 2 bits from the two inputs while rippling the carry
FA A1(Cin, A[0], B[0], c1, S[0]);   //FA(Cin, A, B, Cout, S);
FA A2(c1, A[1], B[1], c2, S[1]);
FA A3(c2, A[2], B[2], c3, S[2]);
FA A4(c3, A[3], B[3], Cout, S[3]);

endmodule
```

**Figure 27. FA Module Implementation**

- Simulation Output:

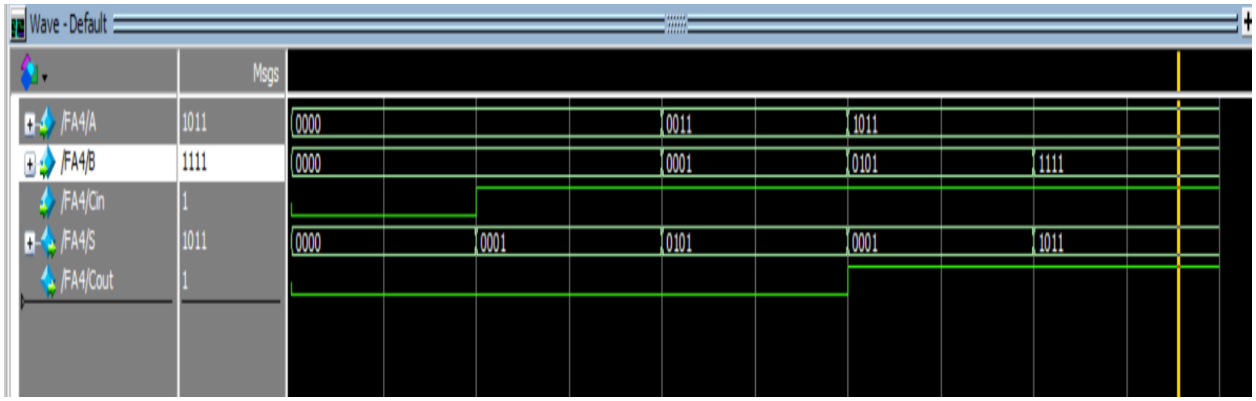


**Figure 28. FA Module Simulation**

As shown in the figure, the module outputs the sum and carry of adding 4-bits inputs and a carry as follows: A + B + Cin = Sum , Carry

- '0000' (0) + '0000' (0) + '0'= '0000' , '0' (0)
- '0011' (3) + '0001' (1) + '1' (1) = '0101' , '0' (5)
- '1011' (11) + '0101' (5) + '1' (1) = '0001' , '1' (10001 = 17)
- '1011' (11) + '1111' (15) + '1' (1) = '1011' , '1' (11011 = 27)

## 8) 4-bits Subtractor:

This module is used to find the result of Subtracting 2 inputs each of them with 4 bits. The module uses 4-bit Full Adder to add the first input with the two's complement of the second input.

```
/*This module implments a 4-bits subtractor that subtracts B from A and the answer is stored in y*/
module Subtractor4(A, B, y);
//A-B
input logic [3:0] A, B;   //Inputs A and B

output logic [3:0] y;     //Output y = A-B

wire Cout;
wire [3:0] B_bar;

//Finding B' (First Complment)
not(B_bar[0], B[0]);
not(B_bar[1], B[1]);
not(B_bar[2], B[2]);
not(B_bar[3], B[3]);

//Using the 4-bits Adder to add A and the two's complment of B.
FA4 A1(1'b1, A, B_bar, Cout, y);   // FA4(Cin, A, B, Cout, S);

endmodule
```

**Figure 29. Subtractor4 Module Implementation**

- Simulation Output:

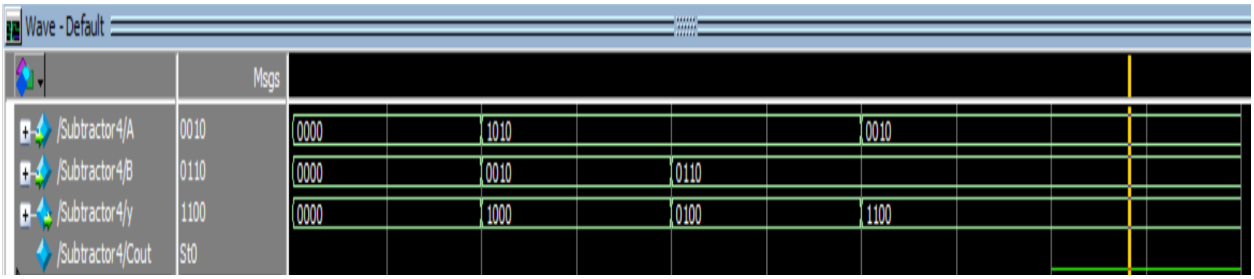


**Figure 30. Subtractor4 Module Simulation**

As shown in the figure, the module outputs the result of subtracting 4-bits inputs and a carry as follows:

| | A | - | B | = | Result | |
|---|---|---|---|---|---|---|
| ● | '0000' (0) | - | '0000' (0) | = | '0000' | (0) |
| ● | '1010' (-6) | - | '0010' (2) | = | '1000' | (-8) |
| ● | '1010' (-6) | - | '0110' (6) | = | '0100' , '1' | (10100 = -16+4 = -12) |
| ● | '0010' (2) | - | '0110' (6) | = | '1100' | (-4) |

## 9) Comparator:

This module is used to find the result of comparing 2 inputs each of them with 4 bits. The module outputs y which is the signal to determine whether B>A or not. If y = 1, then B>A. The implemented Subtractor4 Module is used to subtract the two inputs and finds whether the MSB is '1' or not. If the result of subtraction is negative (MSB = '1'), it means that B is greater than A.

```
/*This module implments a 4-bits Comarator compares between 2 4-bits inputs*/
module Comparator4(A, B, y);
input logic [3:0] A, B;  //Inputs
output logic y;

wire [3:0] res;          //Subtraction result

Subtractor4 M1(A, B, res);   //A-B

assign y = res[3];           //y is a signal to determine whether B>A or not. if y = 1 then B>A

endmodule
```

**Figure 31. Comparator Module Implementation**

- Simulation Output:

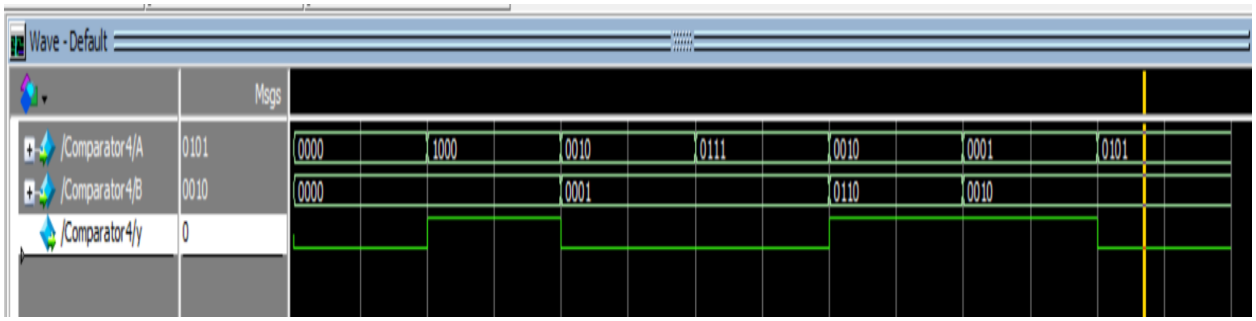


**Figure 32. Comparator Module Simulation**

As shown in the figure, the module outputs the result of Comparing between 4-bits inputs as follows:

| | A - B | = | Result | y |
|---|---|---|---|---|
| ● | '0000' (0) - '0000' (0) | = | '0000' (0) | '0' |
| ● | '1000' (-8) - '0000' (0) | = | '1000' (-8) | '1' |
| ● | '0010' (2) - '0001' (1) | = | '0001' (1) | '0' |
| ● | '0010' (2) - '0110' (6) | = | '1100' (-4) | '1' |

**10) <u>D Flip-flop:</u>**

This module is used to implement a DQ Flip Flop with an Enable signal and a Synchronous Reset. At the positive edge of the clock, it checks if the reset signal is ‘1’, the output will be ‘0’. Otherwise, the input value of ‘D’ will be passed to ‘Q’ if the Enable signal is ‘1’.

```
/* This module implments a DQ Flip Flop with a Enable and Synchrouns Reset */
module D_FF_SyncR( clk, reset, EN, D, Q);
input logic clk, reset, EN, D;
output logic Q;

// At Positive edge of the clock, check the reset and if it euqals '0', pass D to Q if it is enabled.
always_ff @(posedge clk)
if (reset) Q <= 0;
else if (EN) Q <= D;
endmodule
```

**Figure 33. D_FF_SyncR Module Implementation**

- Simulation Output:

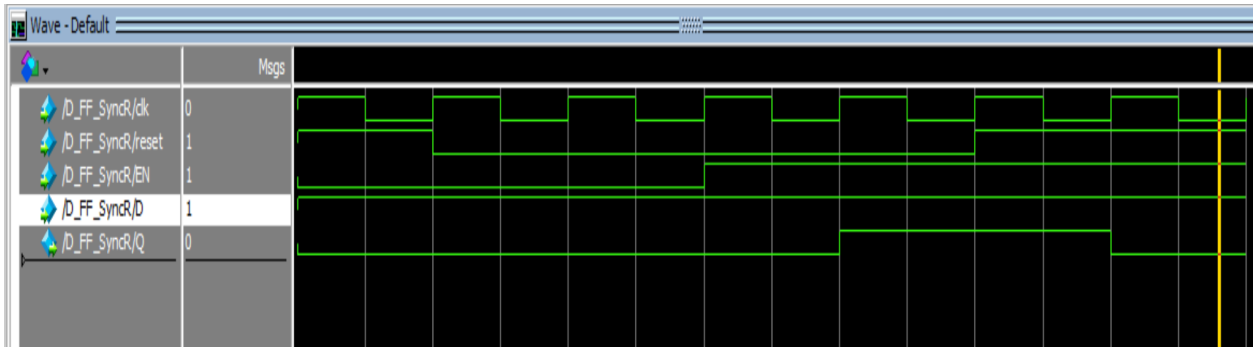


**Figure 34. D_FF_SyncRModule Simulation**

As shown in the figure, the module passes the input value ‘D’ to ‘Q’ if and only if the enable signal is ‘1’ and reset signal is ‘0’.

### 11) 4-bit register:

This module is used to implement a 4-bit register that will deal with each of the 4 counters (S0, S1, M0, M1). The module uses 4 DQ Flip Flops where each one of them is responsible for dealing with 1 bit. The module has an Enable Signal, Set Signal and a Synchronous Reset Signal. At the positive edge of the clock, it checks if the reset signal is '1', the output will be '0000'. Otherwise, the input value of 'D' will be passed to 'Q' if the Enable signal is '1'.

```
/* This module implments a 4-bit register with a Enable, Set, and Synchrouns Reset */
module Register4_SyncR( clk, reset, EN, set, out);

input logic clk, reset, EN;
input logic [3:0] set;
output reg [3:0] out;
//4 D_FF are used to deal with each bit of the 4 bits, where they are all synchronised
D_FF_SyncR M1(clk, reset, EN, set[0], out[0]);
D_FF_SyncR M2(clk, reset, EN, set[1], out[1]);
D_FF_SyncR M3(clk, reset, EN, set[2], out[2]);
D_FF_SyncR M4(clk, reset, EN, set[3], out[3]);

endmodule
```

**Figure 35.register Module Implementation**

- Simulation Output:

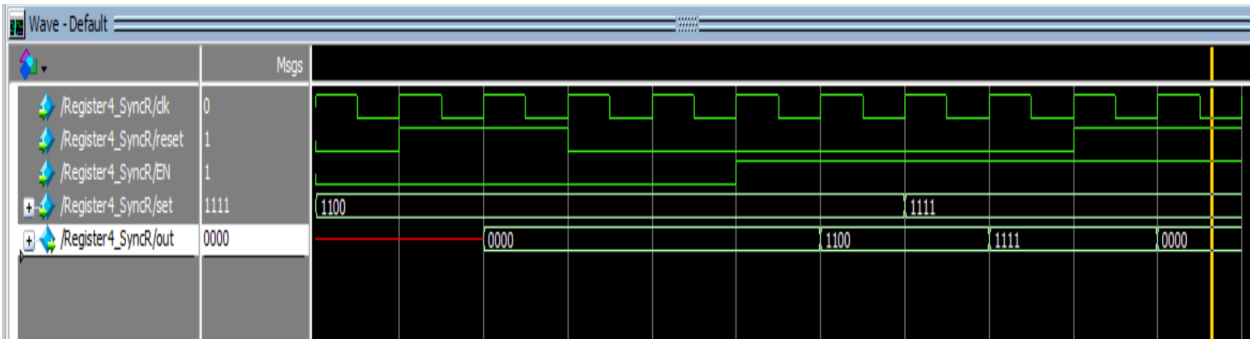


**Figure 36. registerModule Simulation**

As shown in the figure, the module passes the input value 'D' to 'Q' if and only if the enable signal is '1' and reset signal is '0'.

## 12) Clock Divider:

This module is used to be able to get a clock that can drive the stop watch with having a clock every 1 second to increase the stop by one second for the normal state, and we have other two states which is we are increasing the stop watch with 2 every one second which means every half second we increase one on the stopwatch and the last mode is increasing the stop watch with 1 every two seconds.
The clock is coming with a specific speed which is 1 Hz which is for the normal mode, 2 Hz for the fast mode and half Hz for the slow mode.
Also there should be a control unit which decides the clock speed that should supply the stop watch. This can be achieved by using the pushed buttons of slowing or fasting the stop watch, We will have a Mux which have 4 inputs and 1 output and 2 select lines, the select lines are the push buttons and the inputs will be normal clock, fast clock, slow clock and the other input is also the normal clock where when it doesn't have pushed button it select the normal clock and when both buttons are pressed then we have also the normal clock. But when the fast button is pressed the Mux chooses the fast clock, and when the slow button is pressed the Mux chooses the slow clock.
There are three modules which get the 1 Hz, 2 Hz and half Hz and a fourth module which the controller one.
For simulation we used a 50MHz speed clock and we divided it to get a 25MHz clock which is fast clock, 12.5MHz which is the normal one and 6.25MHz which is the slow clock.

```
//This module is used to generate a 1Hz clock out of the 50MHz input clock from the FPGA
module clckdivider1hz(clock_in,clock_out);
input clock_in;       //input clock
output reg clock_out; //output clock

reg[27:0] counter=28'd0;  //counter to increment with each 50MHz clock
parameter division = 28'd50000000; // replace d4 to be d50000000 if the clock is 50Mhz
always @(posedge clock_in)
//check if the counter reaches 50000000, the output clock will generate one clock
begin
 counter <= counter + 28'd1;
 if(counter>=(division-1))
  counter <= 28'd0;
 clock_out <= (counter<division/2)?1'b1:1'b0;
end
endmodule
```

**Figure 37. ClckDivder1Hz Implementation**

```
//This module is used to generate a 2 Hz clock (for speed up mode) out of the 50MHz input clock from the FPGA
module clckdivider2hz(clock_in,clock_out);
input clock_in;  //input clock
output reg clock_out;  //output clock

reg[27:0] counter=28'd0;            //counter to increment with each half cycle of the 50MHz clock
parameter division = 28'd25000000;   //replace d4 to be d25000000 if the clock is 50Mhz
always @(posedge clock_in)
//check if the counter reaches 25000000, the output clock will generate one clock
begin
 counter <= counter + 28'd1;
 if(counter>=(division-1))
  counter <= 28'd0;
 clock_out <= (counter<division/2)?1'b1:1'b0;
end
endmodule
```

**Figure 38. ClckDivder2Hz Implementation**

```
//This module is used to generate a 0.5 Hz clock (for slow down mode) out of the 50MHz input clock from the FPGA
module clckdividerhalfhz(clock_in,clock_out);
input clock_in;   //input clock
output reg clock_out;  //output clock

reg[27:0] counter=28'd0;               //counter to increment with each 2 cycles of the 50MHz clock
parameter division = 28'd100000000;    //replace d4 to be d100000000 if the clock is 50Mhz
always @(posedge clock_in)
begin
//check if the counter reaches 100000000, the output clock will generate one clock
 counter <= counter + 28'd1;
 if(counter>=(division-1))
  counter <= 28'd0;
 clock_out <= (counter<division/2)?1'b1:1'b0;
end
endmodule
```

**Figure 39. ClckDivderHalfHz Implementation**

```
// This module is used to determine the input clock to S0, S1, M0, M1 based on the selected mode
module clock_dividor(clk,speedup,slowdown,outputclk);
input clk, speedup, slowdown;  //inputs
output reg outputclk;          //output
wire  w0,w1,w2;                //wires

clckdivider1hz clk1(clk,w0);     //Normal clock
clckdividerhalfhz clk2(clk,w1);  //Speed up clock
clckdivider2hz clk3(clk,w2);     //Slow down clock
//Mux 4 is used to select from different clocks based on speedup,slowdown
MUX4 #(1) mux(w0,w1,w2,w0,{speedup,slowdown},outputclk);

endmodule
```

**Figure 40. ClckDividerController Implementation**

- Simulation Output:

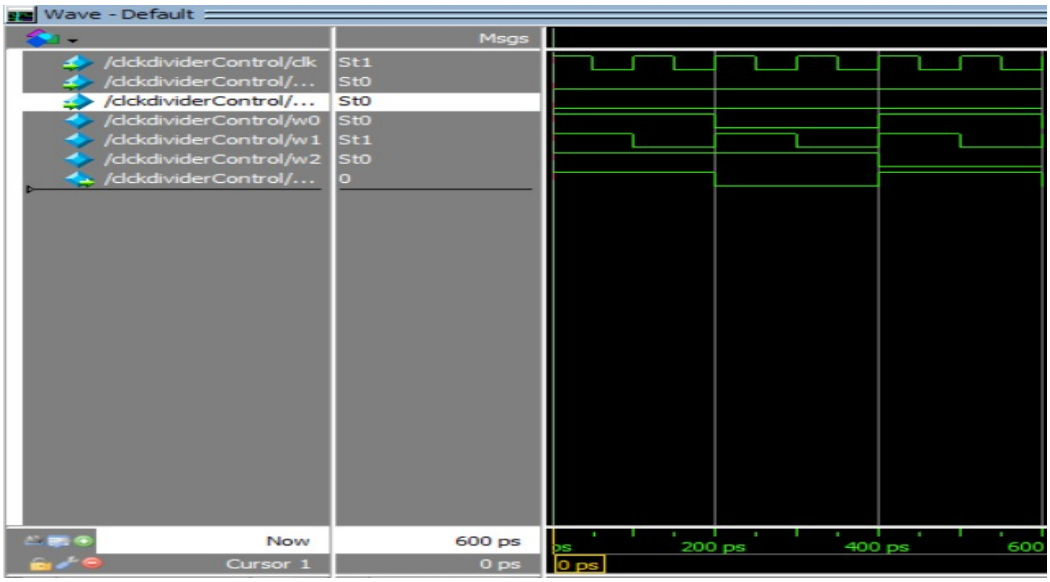


**Figure 41. ClkDividerController Simulation normal clock output**

As we can see that we got three clocks, one is normal which is w0 wire, fast one which is w1 wire and slow one which is  w2 wire, and we can see that the output is the normal one as no push button is pressed.

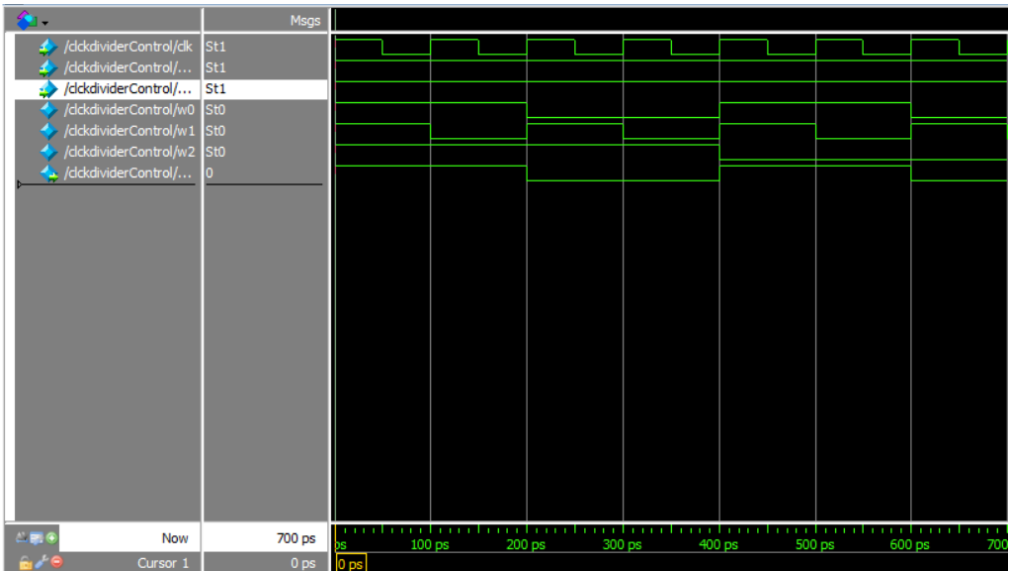


**Figure 42. ClkDividerController Simulation normal clock output**

We can see here when the two push buttons are pressed it made the output to be also the normal clock

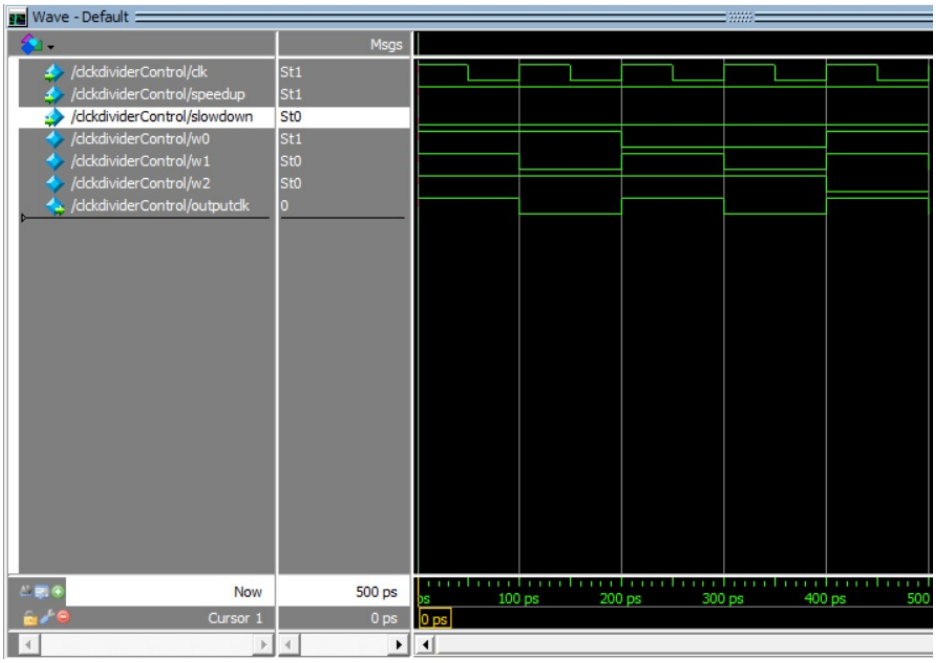


**Figure 43. ClkDividerController Simulation fast clock output**

We can see here when only the speedup push button is pressed, the controller choosed the fast clock to be the output.

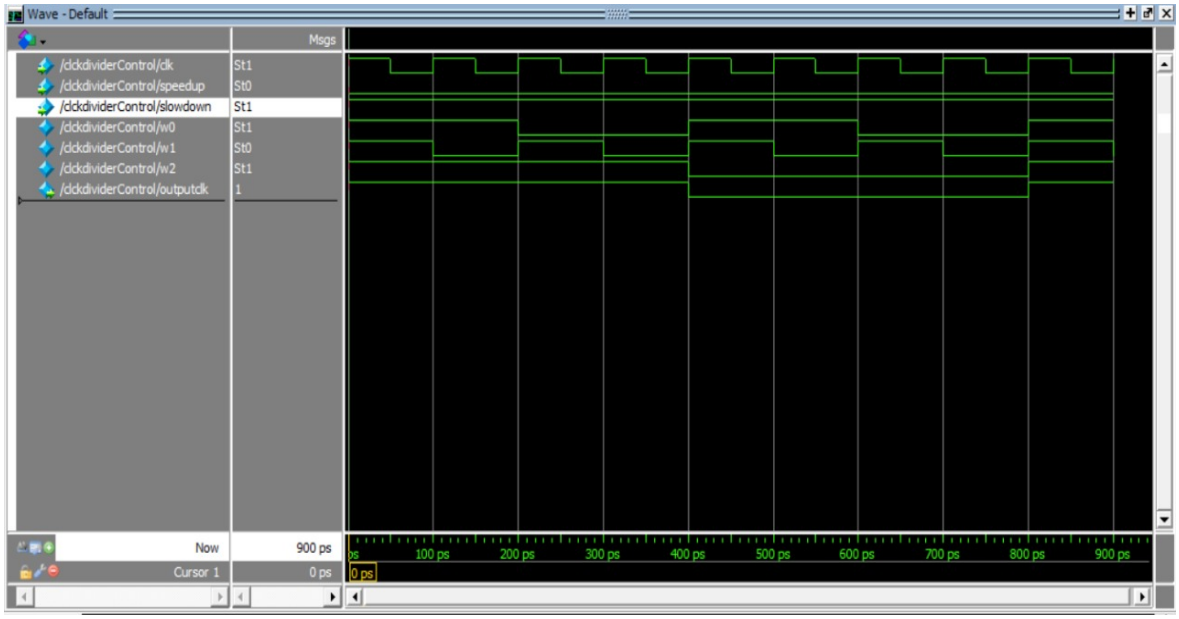


**Figure 44. ClkDividerController Simulation slow clock output**

We can see here when only the slowdown push button is pressed, the controller choosed the slow clock to be the output.

### 13) S0:

This module is used to implement the first digit of seconds block (S0). The input signals to the block are: clk, reset, start, down, ups, sig1_S1, comp_sig (which states whether we are out of the allowed range or not). The outputs are: out_S0 (Counter value), out_sig_S0 (Output signal to increment S1).

```
1  //This module implments the S0 digit in the stopwatch.
2  module S0_block(clk, reset, start, ups, in_sig_S1, in_comp_sig_S1, out_sig_S1, out_S0);
3  input logic clk, reset, start, ups, in_sig_S1, in_comp_sig_S1;   //inputs
4  output reg [3:0] out_S0;                                         //output register 4 bits
5  output logic out_sig_S1;                                         //outputs
6
7  //wires
8  wire [3:0] M1_out, M2_out;
9  wire cout_M2, M3_out, M4_out, M5_out, M6_out, M7_out, M11_out, M13_out, M14_out, M15_out,M16_out;
10 wire down, reset_signal, enable_signal;
11 wire [3:0] set_reg;
12 assign down = ~ups;
13
14 //Set Signals based on the selected mode (up/down)
15 MUX2 #(4) M1(4'b1111, 4'b0001, ups, M1_out);
16 FA4 M2(1'b0, out_S0, M1_out, cout_M2, M2_out);
17 AND4_mod M3(~out_S0[0], ~out_S0[1], ~out_S0[2], ~out_S0[3], M3_out);
18 AND4_mod M4(out_S0[0], ~out_S0[1], ~out_S0[2], out_S0[3], M4_out);
19 and M5(M5_out, ~down, M4_out);
20 and M6(M6_out, down, M3_out);
21 and M7(M7_out, ups, M5_out);
22 or M8(M16_out, M7_out, M6_out); //out_sig_S1
23 and M16(out_sig_S1,M16_out,enable_signal);
24 OR3_mod M9(M5_out, reset, in_comp_sig_S1, reset_signal);  //Reset Signal
25 and M13(M13_out, ups, in_sig_S1);
26 and M14(M14_out, M6_out, in_sig_S1);
27 or M15(M15_out, M13_out, M14_out);
28 and M10(enable_signal, start, ~M15_out);  //Enable Signal
29 and M11(M11_out, down, M3_out);
30 MUX2 #(4) M12(M2_out, 4'b1001, M11_out, set_reg);
31
32
33 //Register Output Signal
34 Register4_SyncR RR(clk, reset_signal, enable_signal, set_reg, out_S0);
35 endmodule
```

**Figure 45. S0 Module Implementation**

- Simulation Output:

1)[10:20] stop counting when receives signal from s1

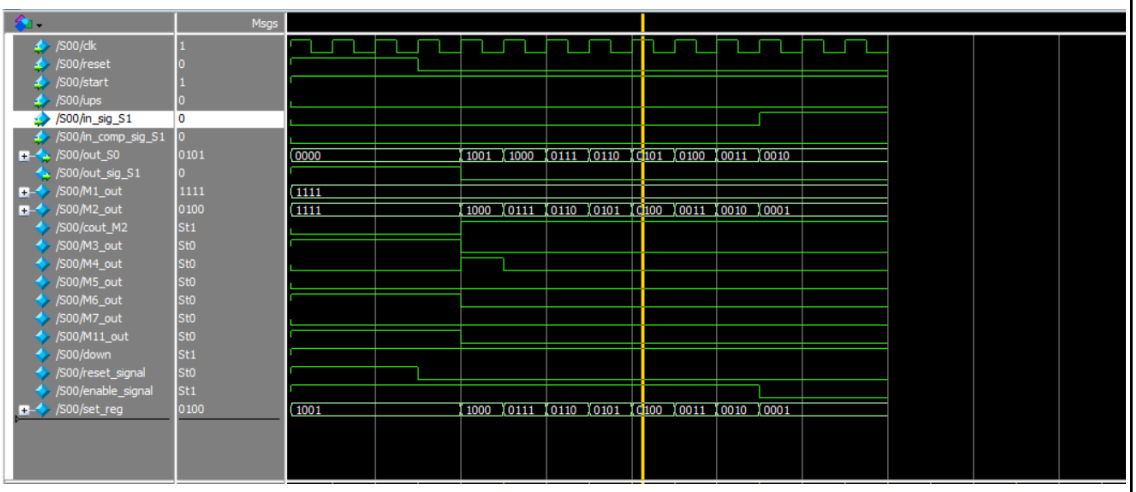


**Figure 46. S0 Module simulation**

2) [10:20] stop counting when reach (0) down

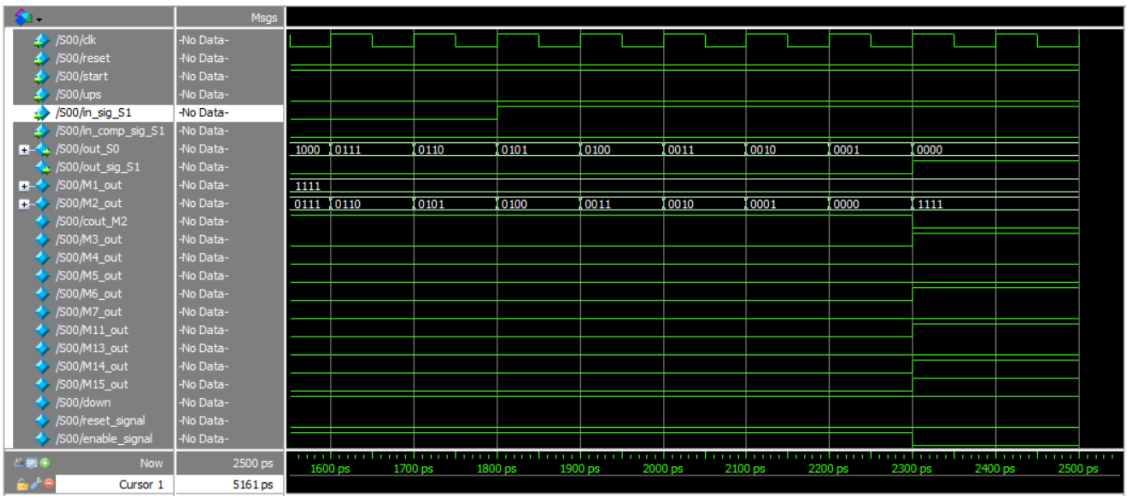


**Figure 47. S0 Module simulation**

3)[49:30] stop counting when receives signal from s1

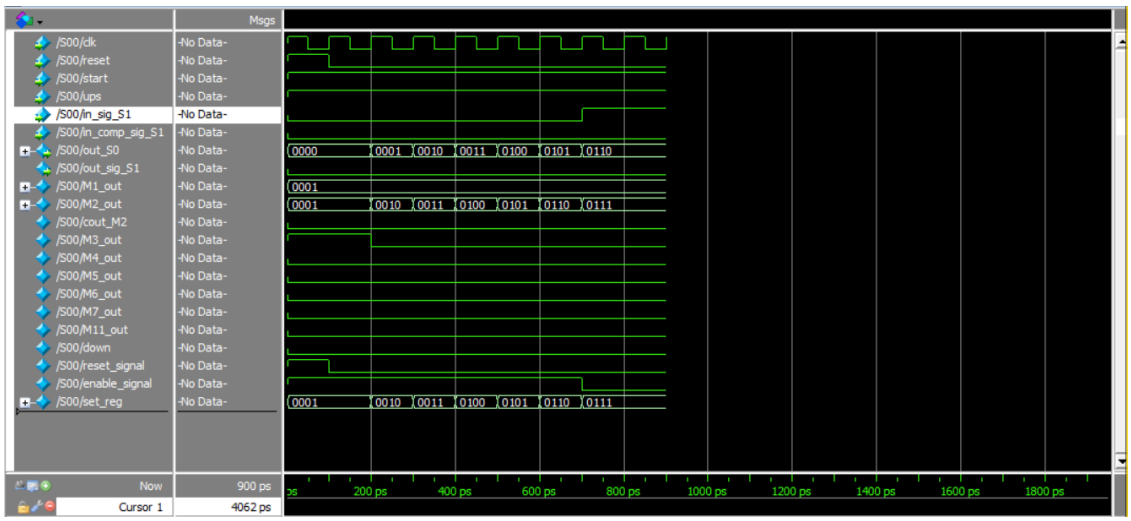


**Figure 47. S0 Module simulation**

4) down form any start reach limit (0) reset ( 9)

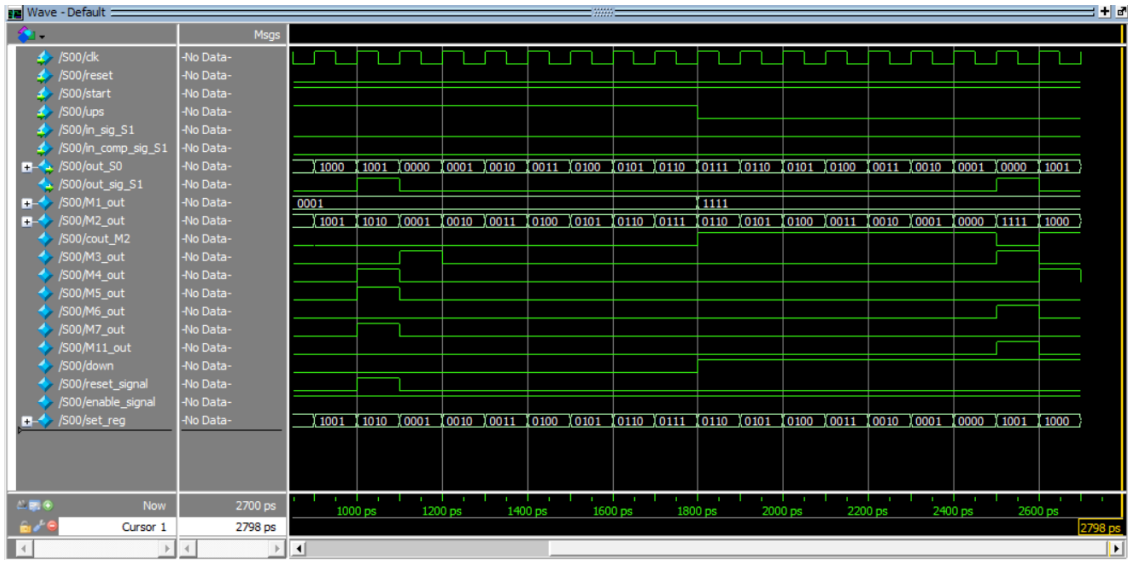


**Figure 48. S0 Module simulation**

5)More than or equal [49:30] stop and reset counting up

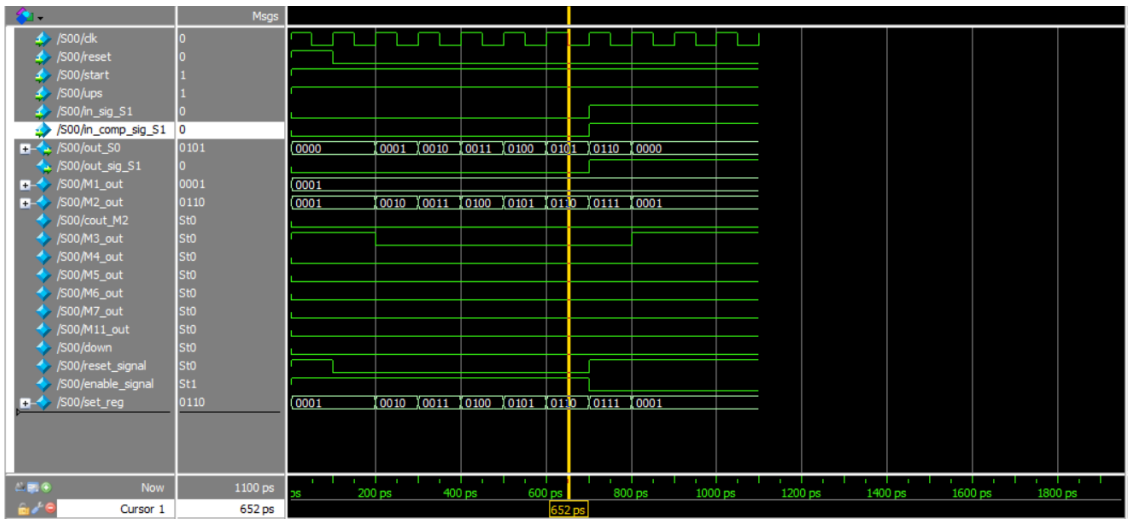


**Figure 49. S0 Module simulation**

6)pause counting stop _start count a gain

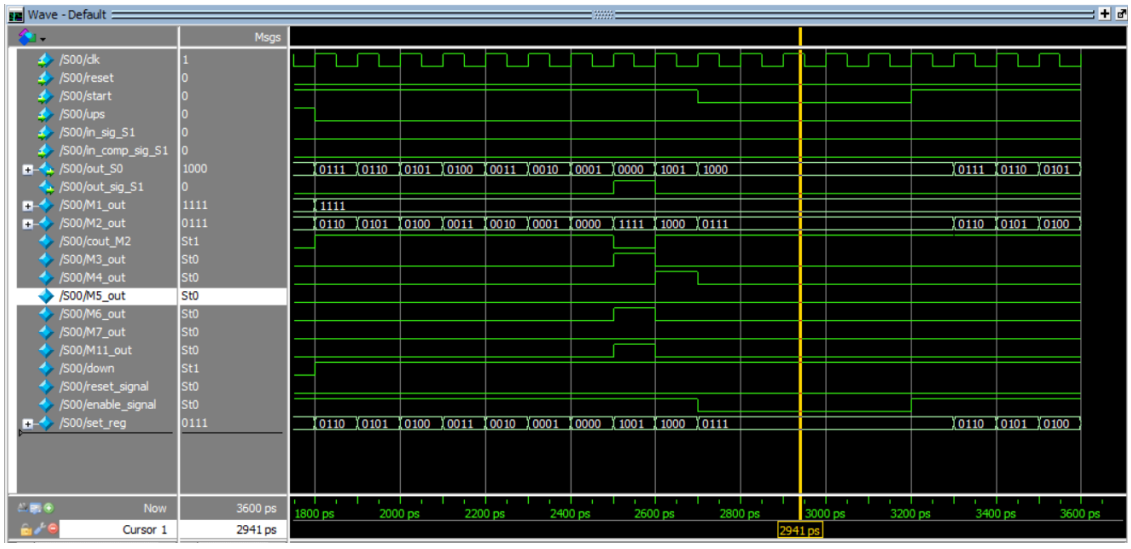


**Figure 50. S0 Module simulation**

7)reset_ start count (0)_until remove reset start counting

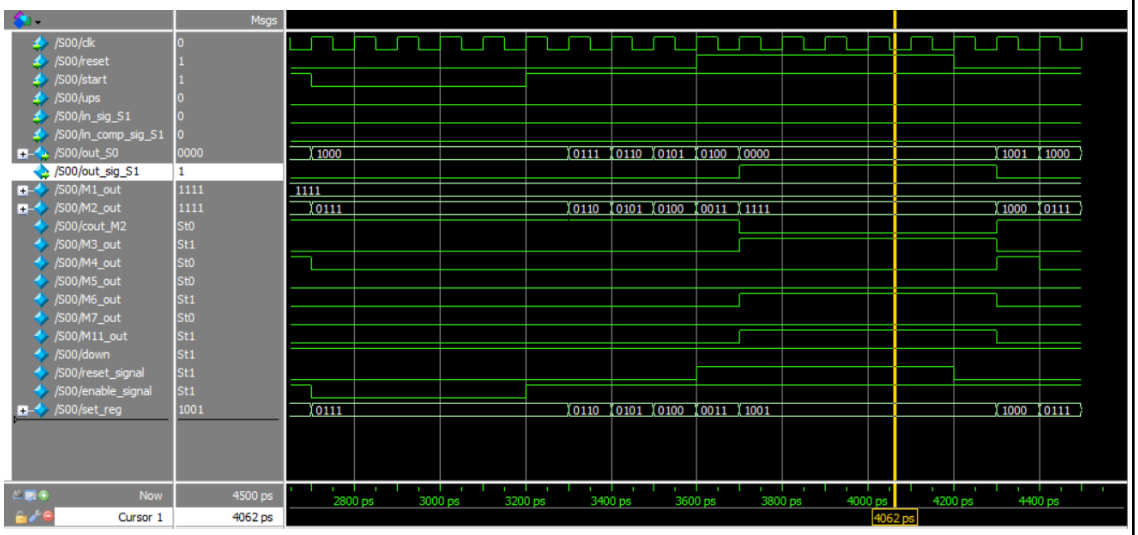


**Figure 51. S0 Module simulation**

8)smaller than [10:20] reset to 0 and stop counting

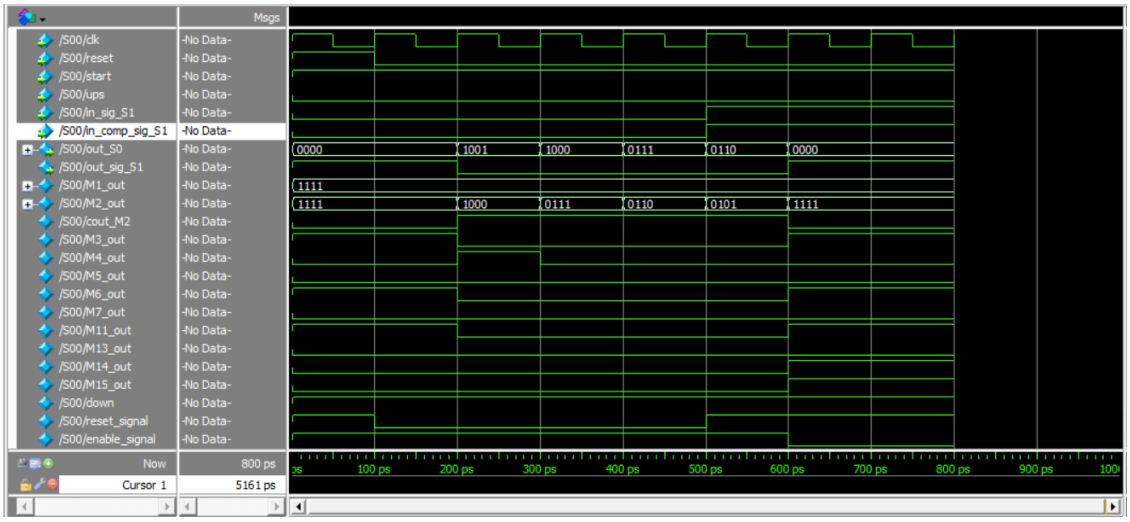


**Figure 52. S0 Module simulation**

9) up reach limit (9 ) reset (0)

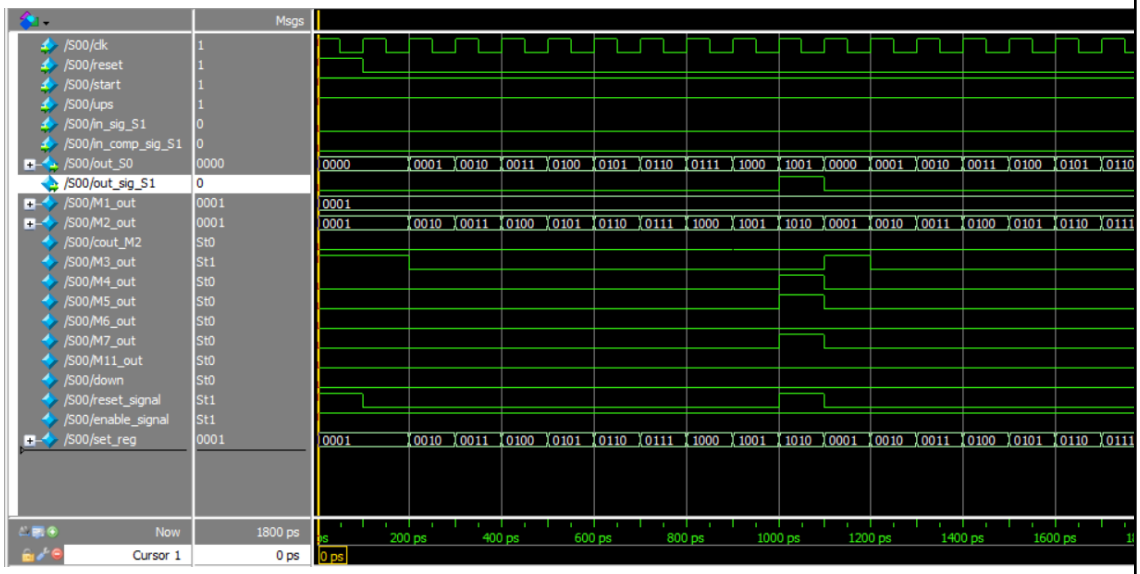


**Figure 53. S0 Module simulation**

**S1:**

This module is used to implement the second digit of seconds digit block (S1). The input signals to the block are: clk, reset, start, down, ups, sig_S0 (to increment), sig_M0 (to enable). The outputs are: out_S1 (Counter value), out_sig_M0(Output signal to increment S1), out_comp_S0, out_sig_S0

```
Ln#
 1  //This module implments the S1 digit in the stopwatch.
 2  module S1_block(clk, reset, start, ups, force_reset, in_sig_S0, in_sig_M0, out_sig_S0, out_comp_sig_S0, out_sig_M0, out_S1,M_set, plus_min2);
 3  input logic clk, reset, start, ups, force_reset, in_sig_S0, in_sig_M0,M_set, plus_min2;  //inputs
 4  output reg [3:0] out_S1;                                                                 //output register 4 bits
 5  output logic out_sig_S0, out_comp_sig_S0, out_sig_M0;                                    //outputs
 6
 7  //wires
 8  wire [3:0] M1_out, M2_out, M3_out;
 9  wire cout_M3, M4_out, M5_out, M6_out, M8_out, M9_out, M10_out, M11_out, M12_out, M13_out, S_3, S_2,M50_out,M66_out;
10  wire M17_out, M18_out, M19_out, M20_out, M21_out, M22_out, M23_out, M24_out, M26_out, M27_out, M28_out, M30_out, M7_out, M72_out, M71_out;
11  wire down, reset_signal, enable_signal;
12  wire [3:0] set_reg;
13  assign down = ~ups;
14
15  //Set Signals based on the selected mode (up/down)
16  MUX2 #(4) M1(4'b1111, 4'b0001, ups, M1_out);
17  MUX2 #(4) M2(4'b0000, M1_out, in_sig_S0, M2_out);
18  FA4 M3(1'b0, out_S1, M2_out, cout_M3, M3_out);
19  AND4_mod M4(out_S1[0], ~out_S1[1], out_S1[2], ~out_S1[3], M4_out);
20  AND4_mod M5(~out_S1[0], ~out_S1[1], ~out_S1[2], ~out_S1[3], M5_out);
21  and M6(M6_out, ~down, M4_out);
22  and M7(M7_out, in_sig_S0, M6_out);
23  or M70(reset_signal, M7_out, force_reset);   //Reset Signal
24  and M8(M8_out, ups, M6_out);
25  and M9(M9_out, down, M5_out);
26  or M10(M10_out, M9_out, M8_out);  //out_sig_M0
27  and M50(M50_out,  M10_out, in_sig_S0);
28  and M51(out_sig_M0,M50_out,enable_signal);
29  ///comparator Circuit
30  Comparator4 M11(out_S1, 4'b0010, M11_out);
31  Comparator4 M12(4'b0010, out_S1, M12_out);
32  MUX2 #(1) M13(M11_out, M12_out, ups, M13_out);
33  and M14(M66_out, M13_out, in_sig_M0);
34  or M66(out_comp_sig_S0,M66_out,M_set);
35  //S3, S2
```

```
//S3, S2
and M15(S_3, down, out_comp_sig_S0);  //S3
and M16(S_2, ups, out_comp_sig_S0);    //S2
and M17(M40_out, down, M5_out);
and M40(M17_out, M40_out, in_sig_S0);
and M18(M18_out, ups, reset);
and M19(M19_out, down, reset);
or M20(M20_out, S_2, M19_out);
or M22(M22_out, M18_out, S_3);
or M21(M21_out, M22_out, M20_out);  //M1
and M23(M23_out, ~M22_out, M17_out);
or M24(M24_out, M23_out, M20_out);  //M0

MUX4 #(4) M25(M3_out, 4'b0101, 4'b0010, 4'b0011, {M21_out, M24_out}, set_reg);  //Set Register

AND4_mod M26(out_S1[0], out_S1[1], ~out_S1[2], ~out_S1[3], M26_out);
AND4_mod M27(~out_S1[0], out_S1[1], ~out_S1[2], ~out_S1[3], M27_out);

MUX2 #(1) M28(M27_out, M26_out, ups, M28_out);
and M29(out_sig_S0, M28_out, in_sig_M0);  //out_sig_S0

or M71(M71_out, plus_min2, start);
and M30(M30_out, ~out_sig_S0, start);  //Enable Signal
wire M31_out;
or M31(enable_signal, M30_out, reset);

//Register Output Signal
Register4_SyncR RR(clk, reset_signal, 1'b1, set_reg, out_S1);
endmodule
```

**Figure 54. S1 Module implementation**

- Simulation Output:

1)don’t stop when 10 signal come and greater 2

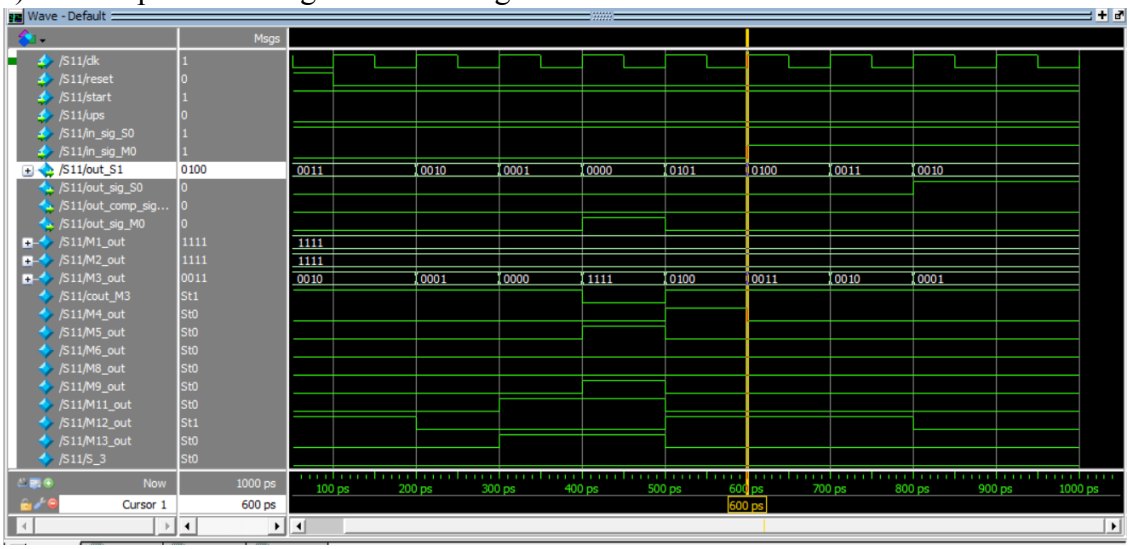


**Figure 54. S1 Module Simulation**

2)don’t stop when [49]signal come and smaller 3

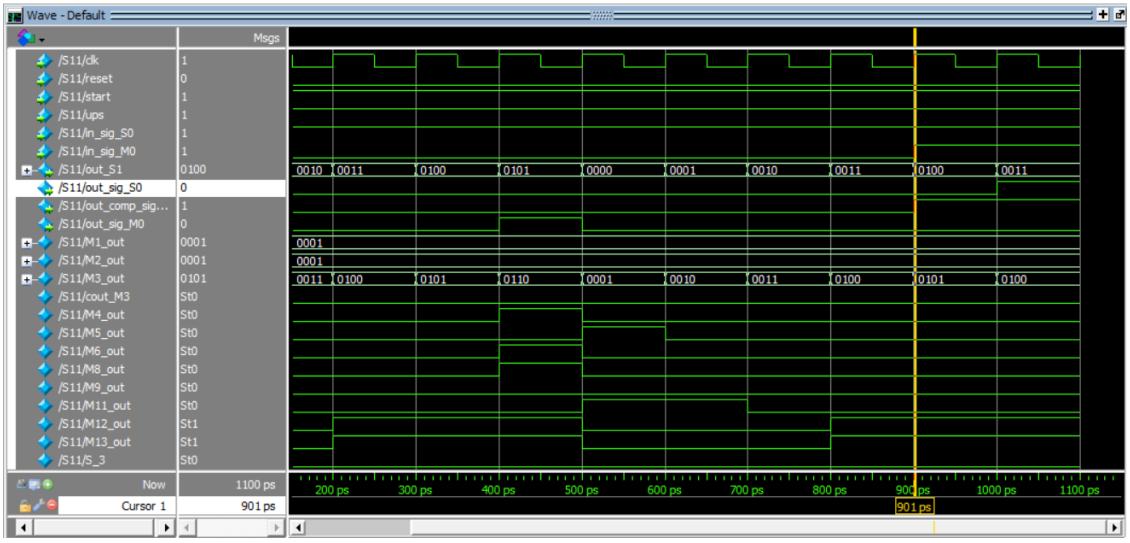


**Figure 55. S1 Module Simulation**

3)set to 2 if 10 signal come and greater 2 and stop

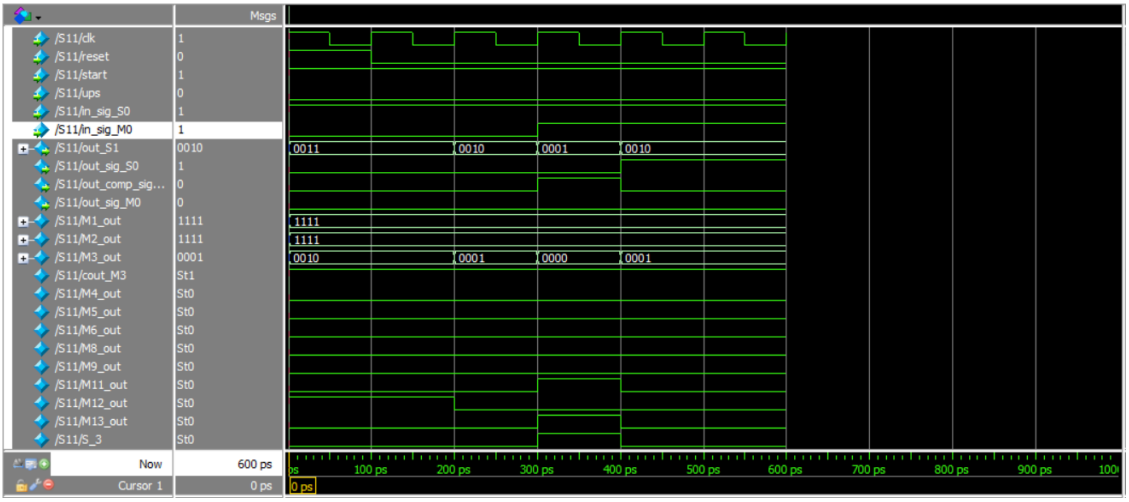


**Figure 56. S1 Module Simulation**

4)set to 3 if [49]signal greater [3] and stop:

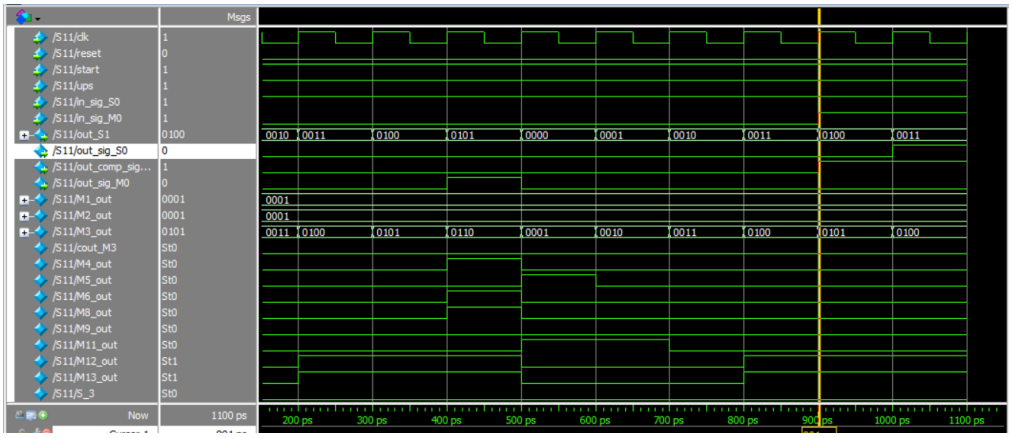


**Figure 57. S1 Module Simulation**

5)start [2] when up till[5] and reset [0] then count_up

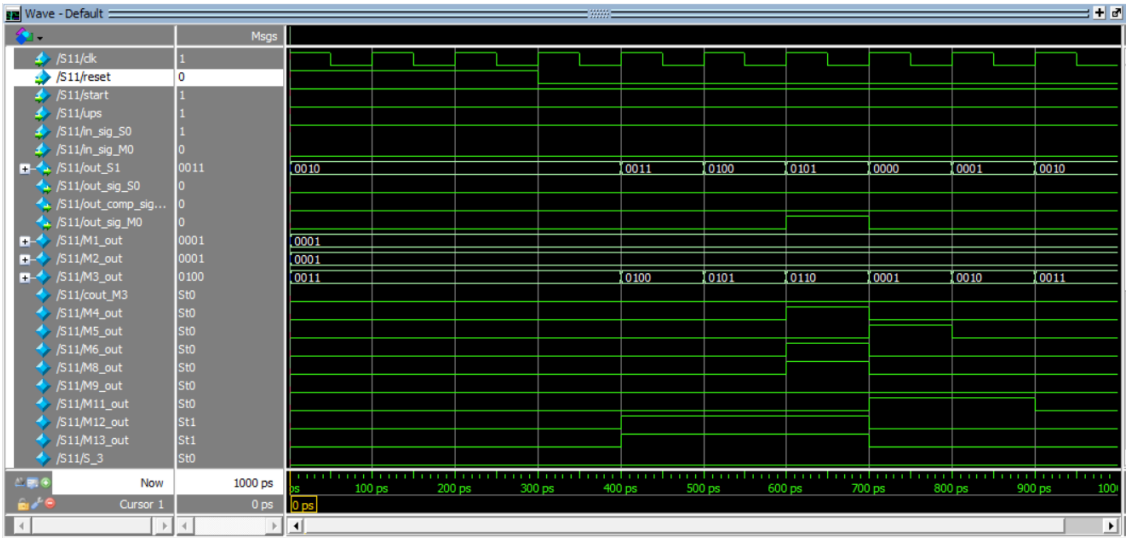


**Figure 58. S1 Module Simulation**

6)start [3] when down till[0] and reset [5] then count_down

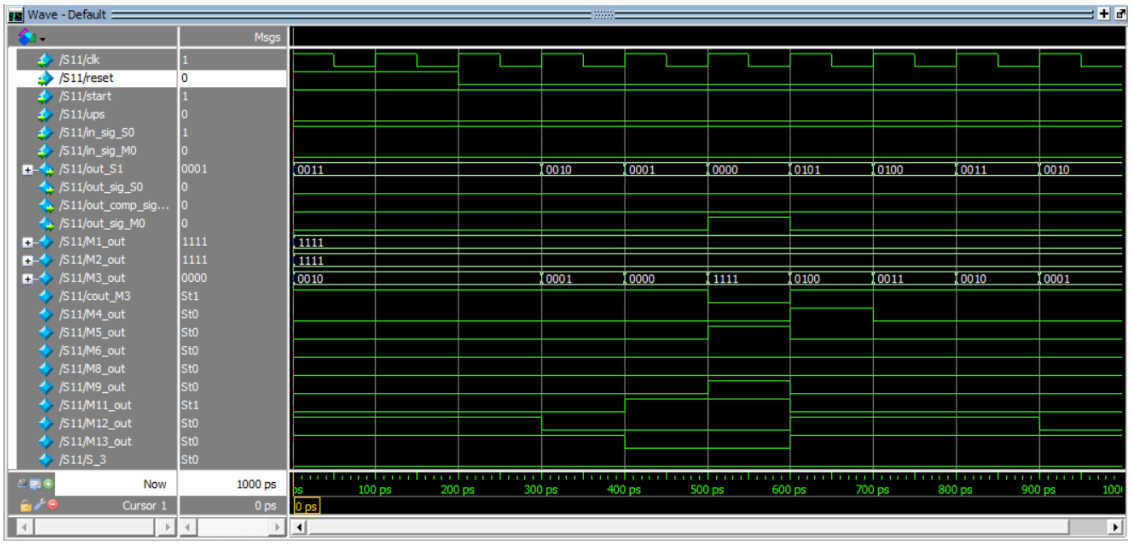


**Figure 59. S1 Module Simulation**

**M0:**

```
Ln#
 1  //This module implments the M0 digit in the stopwatch.
 2  module M0_block(clk, reset, start, ups, plus_min2, force_reset, in_sig_S1, in_sig_M1, out_sig_S1, out_sig_M1, out_M0,M_set);
 3  input logic clk, reset, start, ups, plus_min2, force_reset, in_sig_S1, in_sig_M1;
 4  output reg [3:0] out_M0;
 5  output logic out_sig_S1, out_sig_M1,M_set;
 6
 7  //wires
 8  wire [3:0] M1_out, M2_out, M3_out, M4_out, M5_out, M6_out, M17_out, M18_out;
 9
10  wire cout_M3, cout_M6, M7_out, M8_out, M9_out, M10_out, M11_out, M12_out, M13_out, M14_out, M15_out, M16_out, M20_out,M44_out, M50_out,M67_out;
11  wire M21_out, M22_out, M23_out, M24_out, M25_out, M26_out,M35_out ,M28_out, M29_out,M36_out,M38_out,M39_out,M40_out,M41_out,M42_out,M43_out;
12  wire down, reset_signal, enable_signal,M66_out,M70_out;
13  wire [3:0] set_reg;
14
15  assign down = ~ups;
16  //Set Signals based on the selected mode (up/down)
17  MUX2 #(4) M1(4'b1111, 4'b0001, ups, M1_out);
18  MUX2 #(4) M2(4'b0000, M1_out, in_sig_S1, M2_out);
19  FA4 M3(1'b0, out_M0, M2_out, cout_M3, M3_out);
20
21  MUX2 #(4) M4(4'b1110, 4'b0010, ups, M4_out);
22  MUX2 #(4) M5(4'b0000, M4_out, plus_min2, M5_out);
23  FA4 M6(1'b0, M3_out, M5_out, cout_M6, M6_out);
24
25  AND4_mod M7(~M6_out[0], M6_out[1], ~M6_out[2], M6_out[3], M7_out);
26  AND4_mod M8(M6_out[0], M6_out[1], ~M6_out[2], M6_out[3], M8_out);
27  AND4_mod M9(~M6_out[0], ~M6_out[1], M6_out[2], M6_out[3], M9_out);
28  OR3_mod M10(M7_out, M8_out, M9_out, M10_out);
29
30  AND4_mod M11(M6_out[0], M6_out[1], M6_out[2], M6_out[3], M11_out);
31  AND4_mod M12(M6_out[3], M6_out[2], M6_out[1], ~M6_out[0], M12_out);
32  AND4_mod M13(M6_out[3], M6_out[2], ~M6_out[1], M6_out[0], M13_out);
33  OR3_mod M14(M11_out, M12_out, M13_out, M14_out);
34  MUX2 #(1) M15(M14_out, M10_out, ups, M15_out);
35  and (M39_out,M20_out,ups);
36  or M16(out_sig_M1, M39_out, M15_out);
```

```
37
38  MUX2 #(4) M17({~M13_out, M13_out, M13_out, ~M12_out} , {1'b0,1'b0, M9_out, M8_out}, ups, M17_out);
39  MUX2 #(4) M18(M6_out, M17_out, M15_out, M18_out);
40  or M67(M67_out,in_sig_S1,reset);
41  and M66(M66_out,M40_out,M67_out);
42  or M43(M43_out,M66_out,M41_out);
43  MUX2 #(4) M19(M18_out, 4'b1001, M43_out, set_reg); //Set Register
44
45  AND4_mod M20(out_M0[0], ~out_M0[1], ~out_M0[2], out_M0[3], M20_out);
46  and M38(M38_out,M50_out,~plus_min2);
47  and M21(M21_out, M38_out, ~in_sig_M1);
48  and M36(M36_out,M21_out,ups);
49  and M22(M22_out, ups, reset);
50  and M44(M44_out,M41_out,down);
51  OR3_mod M23(M44_out, M36_out, M22_out,M23_out);   //Reset Signal
52  or O1(reset_signal, force_reset, M23_out);
53  and M24(M24_out, down, reset);
54
55  AND4_mod M25(~out_M0[0], ~out_M0[1], ~out_M0[2], ~out_M0[3], M25_out);
56  or M26(M26_out, M25_out, M24_out);
57  //and M51(M26_out, M51_out, in_sig_S1);
58  and M35(M35_out,M26_out,down);
59  or M70(M70_out,~plus_min2,down);
60  and M40(M40_out,M35_out,M70_out);
61  and M41(M41_out,out_sig_M1,in_sig_M1);
62  or M42(M42_out,M20_out,M41_out);
63  MUX2 #(1) M28(M40_out, M20_out, ups, M28_out);
64
```

```
AND4_mod M25(~out_M0[0], ~out_M0[1], ~out_M0[2], ~out_M0[3], M25_out);
or M26(M26_out, M25_out, M24_out);
//and M51(M26_out, M51_out, in_sig_S1);
and M35(M35_out,M26_out,down);
or M70(M70_out,~plus_min2,down);
and M40(M40_out,M35_out,M70_out);
and M41(M41_out,out_sig_M1,in_sig_M1);
or M42(M42_out,M20_out,M41_out);
MUX2 #(1) M28(M40_out, M20_out, ups, M28_out);

and M27(out_sig_S1, M28_out, in_sig_M1);
and M50(M50_out, out_sig_S1, in_sig_S1);

or M29(M29_out, start, plus_min2);
and M30(enable_signal, ~M50_out, M29_out);  //Enable Signal

and Mset(M_set,M15_out,in_sig_M1);   //out set signal

//Register Output Signal
Register4_SyncR RR(clk, reset_signal, enable_signal, set_reg, out_M0);
endmodule
```

**Figure 60. M0 Module code implementation**

**The output simulation:**

**1)1_signal comes to stop (0) only countdown_till 0 stop**

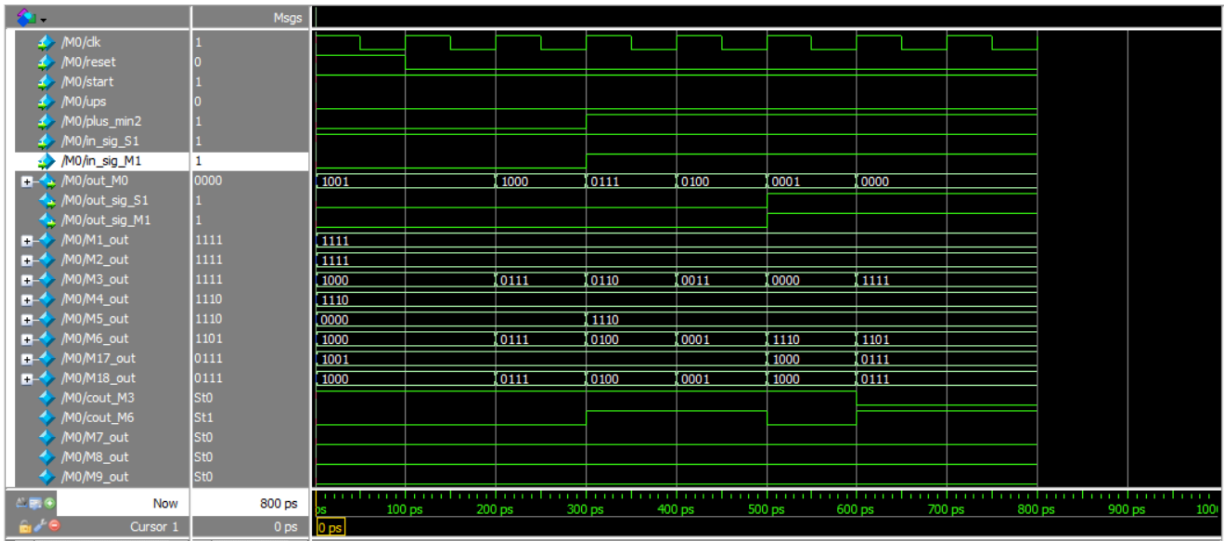


**Figure 61. M0 Module Simulation**

**2)4_signal comes stop 9 only count till 9 stop**

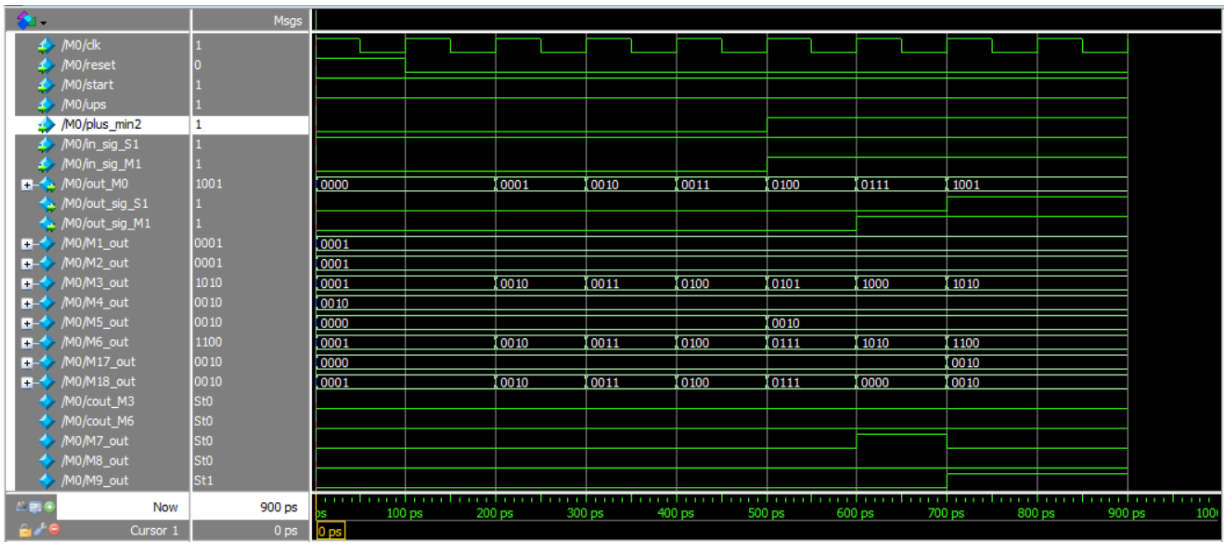


**Figure 62. M0 Module Simulation**

**3)add 2_and coming signal don't exceed 9_1_2**

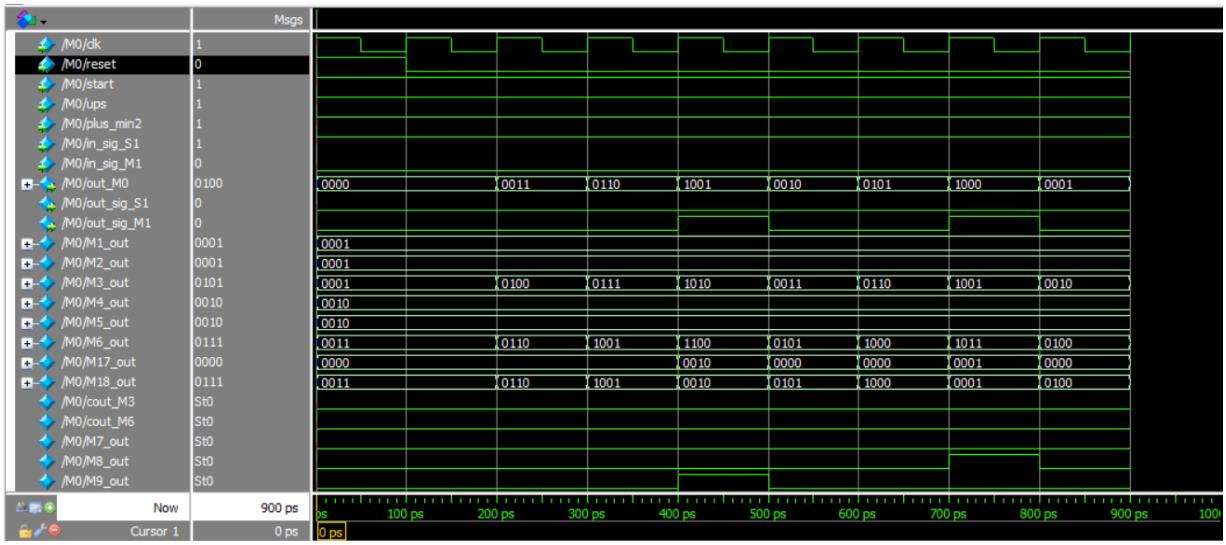


**Figure 63. M0 Module Simulation**

**4)add 2 when up_reach 8 then go zero output 1**


**Figure 64. M0 Module Simulation**

**5)addup_start with (0) till 9 then go to 0**

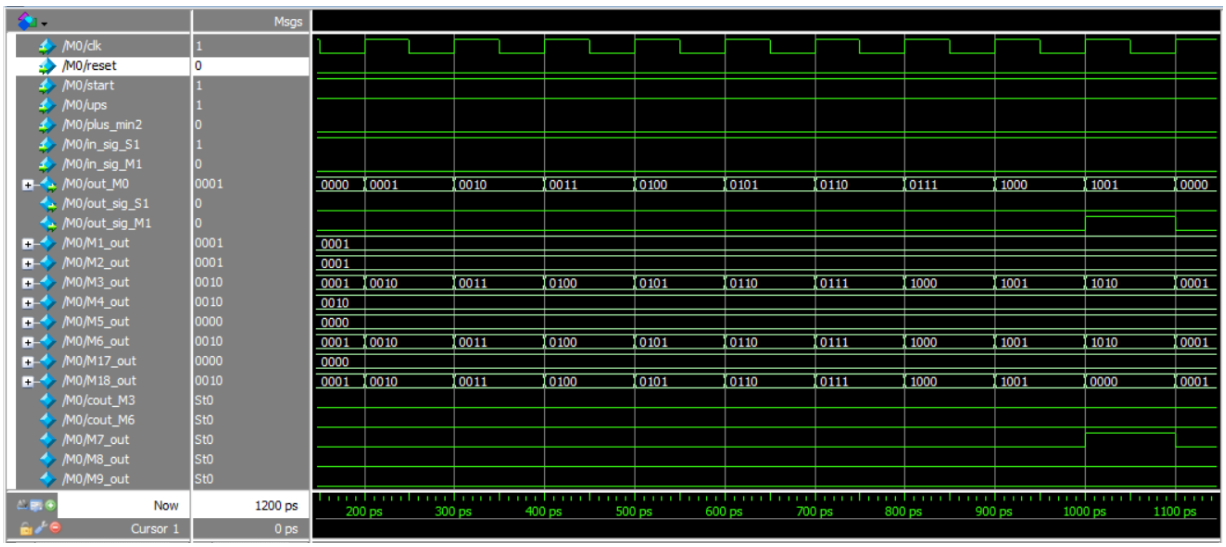


**Figure 65. M0 Module Simulation**

**6)sub(2) and coming signal don't exceed 0_7_8**

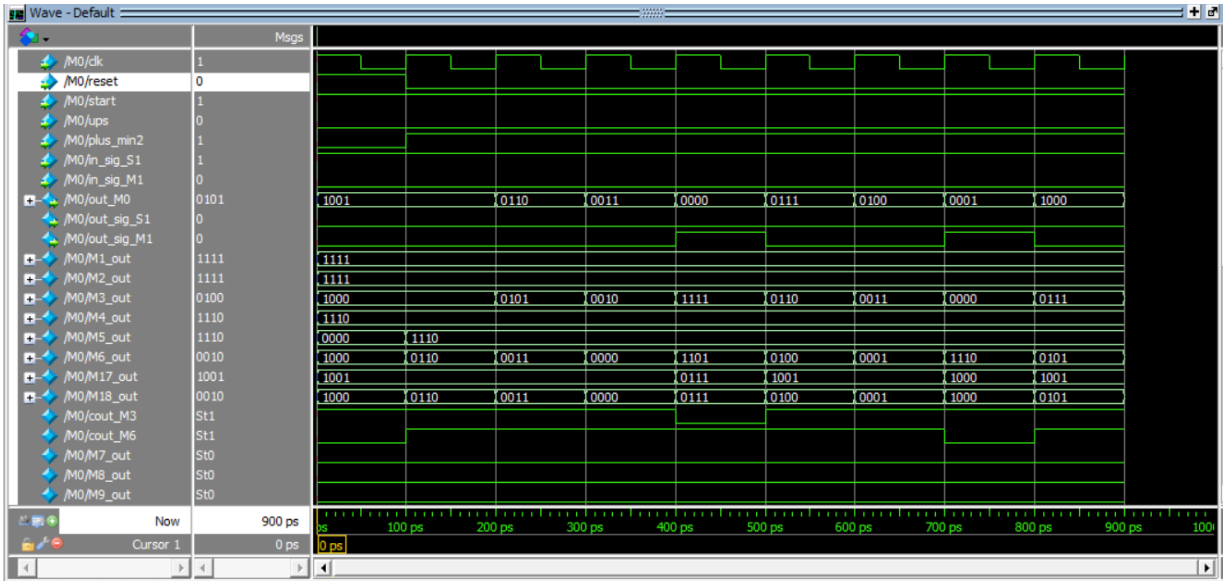


**Figure 66. M0 Module Simulation**

**7)subtract (2)when down reach (1) then go 9 output (1)**

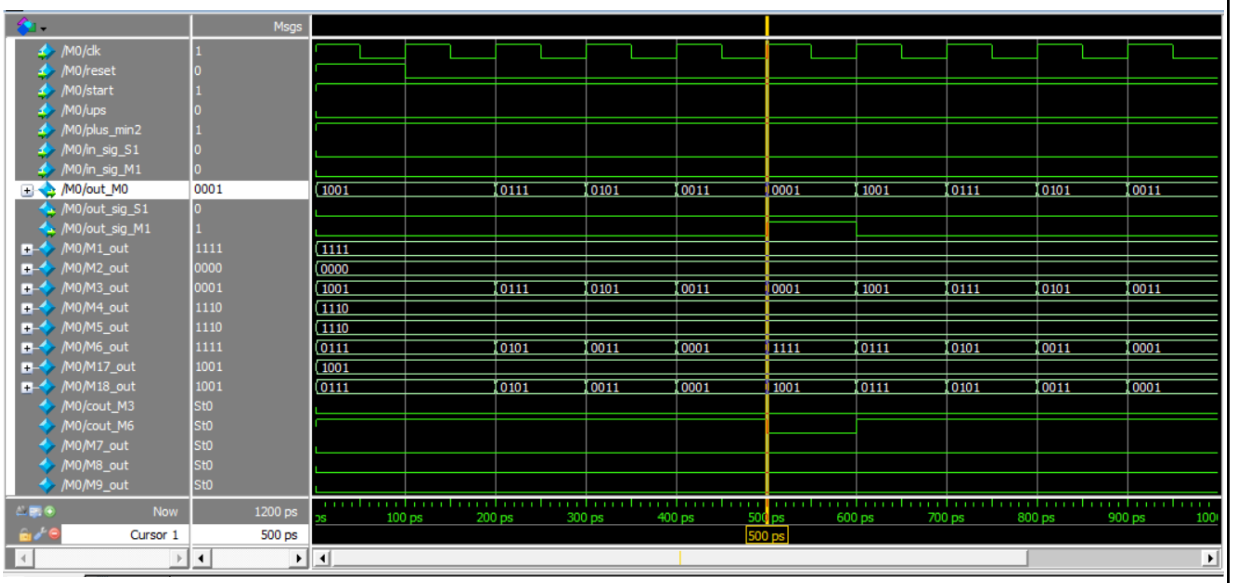


**Figure 67. M0 Module Simulation**

**8)subtract down _start with(9)till (0)then go to (9):**

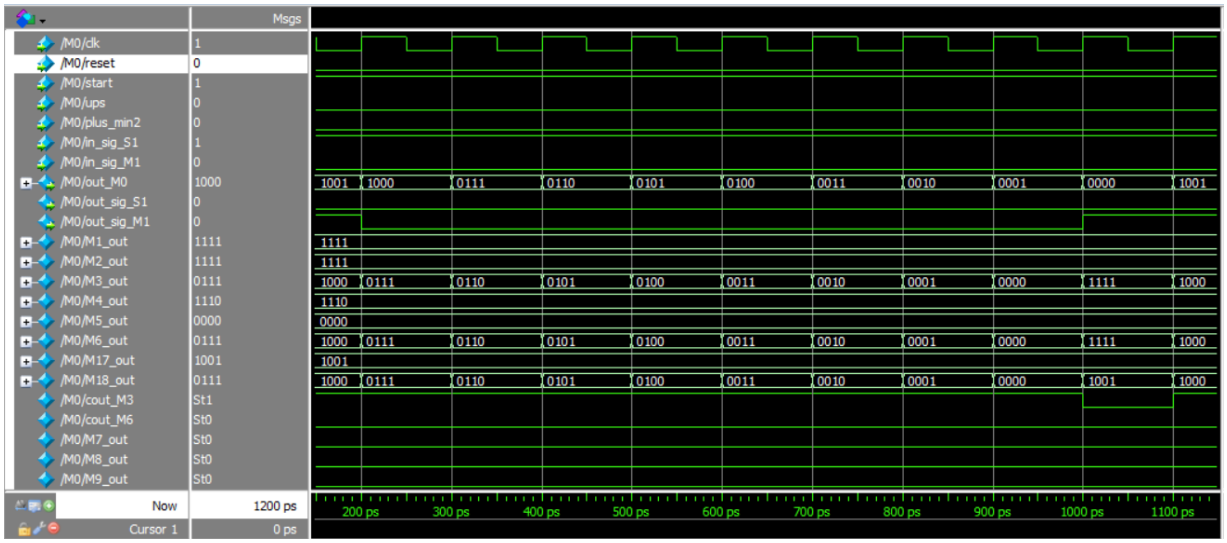


**Figure 68. M0 Module Simulation**

**M1:**

```
Ln#
 1  //This module implments the M1 digit in the stopwatch.
 2  module M1_block(clk, reset, start, ups, plus_min2, force_reset, in_sig_M0, out_sig_M0, out_M1);
 3  input logic clk, reset, start, ups, plus_min2, force_reset, in_sig_M0;
 4  output reg [3:0] out_M1;
 5  output logic out_sig_M0;
 6
 7  //wires
 8  wire [3:0] M1_out, M2_out, M3_out;
 9  wire cout_M3, M4_out, M5_out, M7_out, M8_out, M9_out, M10_out, M12_out,M13_out;
10
11  wire down, reset_signal, enable_signal;
12  wire [3:0] set_reg;
13  assign down = ~ups;
14
15  assign reset_signal = force_reset;
16  //Set Signals based on the selected mode (up/down)
17  MUX2 #(4) M1(4'b1111, 4'b0001, ups, M1_out);
18  MUX2 #(4) M2(4'b0000, M1_out, in_sig_M0, M2_out);
19  FA4 M3(1'b0, out_M1, M2_out, cout_M3, M3_out);
20
21  and M4(M4_out, ups, reset);  //M0
22  and M5(M5_out, down, reset);   //M1
23
```

```
MUX4 #(4) M6(M3_out, 4'b0001, 4'b0100, 4'b0000, {M5_out, M4_out}, set_reg);  //Set Signal

AND4_mod M7(~out_M1[0], ~out_M1[1], out_M1[2], ~out_M1[3], M7_out);
AND4_mod M8(out_M1[0], ~out_M1[1], ~out_M1[2], ~out_M1[3], M8_out);
and M9(M9_out, down, M8_out);

and M10(M10_out, ups, M7_out);

or M11(out_sig_M0, M10_out, M9_out);
and M41(M41_out, out_sig_M0, in_sig_M0);

or M12(M12_out, plus_min2, start);
and M13(M13_out, ~M41_out, M12_out);  //Enable Signal
or (enable_signal,M13_out,reset);

//Register Output Signal
Register4_SyncR RR(clk, reset_signal, enable_signal, set_reg, out_M1);
endmodule
```

**Figure 69. M1 Module implementation**

**The output simulation:**

**1)Countdown from 4 till 1 then stop**

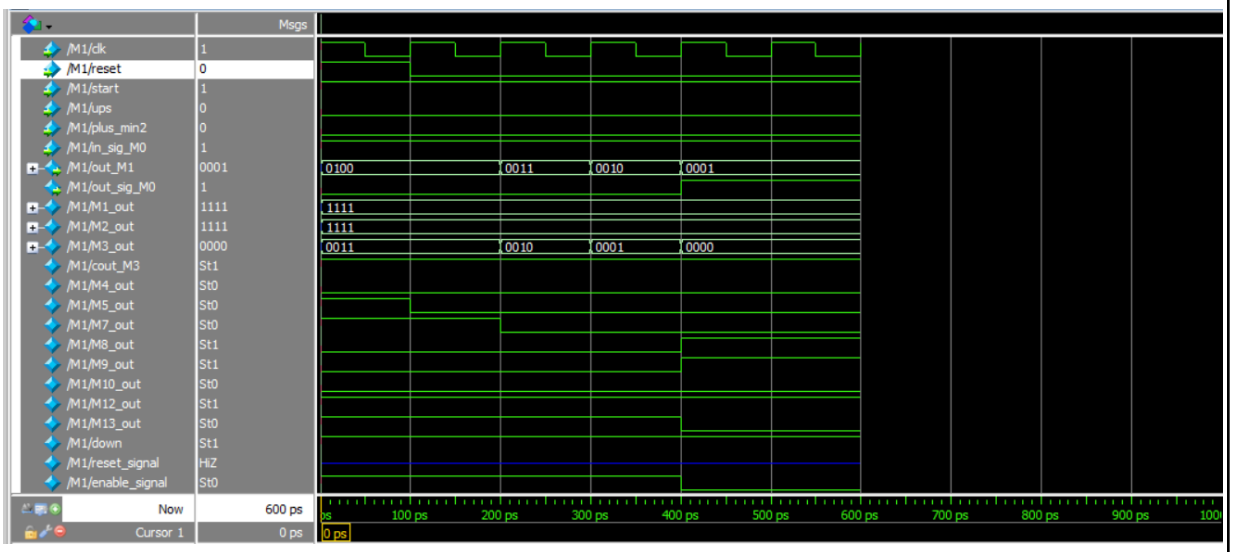


**Figure 70. M1 Module Simulation**

**2) count up from (1) till (4) then stop**

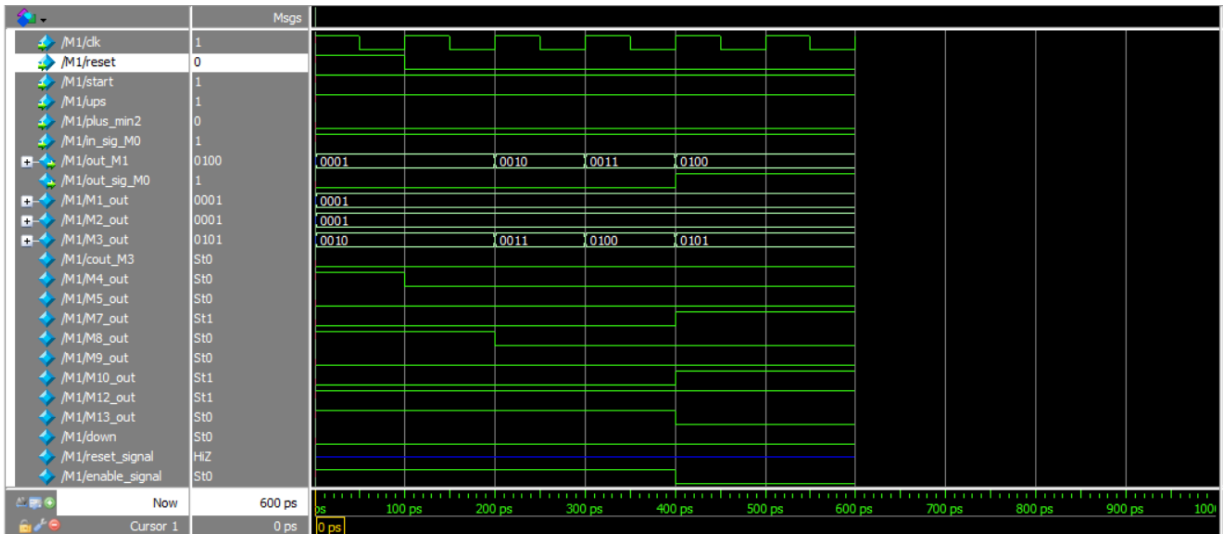


**Figure 71. M1 Module Simulation**

## Stop watch ( simulation of whole circuit):

```
Ln#
1  module STOPWATCH(in_clk, reset, start, up, plus_min2, speedup, slowdown , SS0, SS1, MM0, MM1);
2
3  input logic in_clk, reset, start, up, plus_min2, speedup, slowdown;  //inputs
4  output reg [3:0] SS0, SS1, MM0, MM1;                                 //outputs 4-bits
5  //wires
6  wire ups, sig_S1_S0, sig_S0_S1, comp_sig_S1_S0, sig_S1_M0, sig_M0_M1, sig_M1_M0, sig_M0_S1,setting,M_set, force_reset;
7  //wire clk;
8  wire [3:0] out_S0, out_S1, out_M0, out_M1;
9  assign force_reset = 0;
10 clock_dividor H1(in_clk,speedup,slowdown,clk);
11 wire seven_segment_flag;
12 assign seven_segment_flag = 0;
13 //Finds the value of up* (ups) that will be passed to all modules
14 and (setting,up,~reset);
15 DFF_mod H2(clk, ~reset, setting, ~start, up, ups);
16 // 4 blocks for S0, S1, M0, M1
17 S0_block H3(clk, ~reset, start, ups, sig_S1_S0, comp_sig_S1_S0, sig_S0_S1, out_S0);
18 S1_block H4(clk, ~reset, start, ups, force_reset, sig_S0_S1, sig_M0_S1, sig_S1_S0, comp_sig_S1_S0, sig_S1_M0, out_S1,M_set
19 M0_block H5(clk, ~reset, start, ups, ~plus_min2, force_reset, sig_S1_M0, sig_M1_M0, sig_M0_S1, sig_M0_M1, out_M0,M_set);
20 M1_block H6(clk, ~reset, start, ups, ~plus_min2, force_reset, sig_M0_M1, sig_M1_M0, out_M1);
21
22 //Error module to check the error code 1
23 Error1_out H7(clk, ~reset, up, start, seven_segment_flag, out_S0, out_S1, out_M0, out_M1, SS0, SS1, MM0, MM1);
24 endmodule
25
26
```

**Figure 72.stopwatch implementation**

## The output simulation:

**1)counting down (-2) while stopping:**

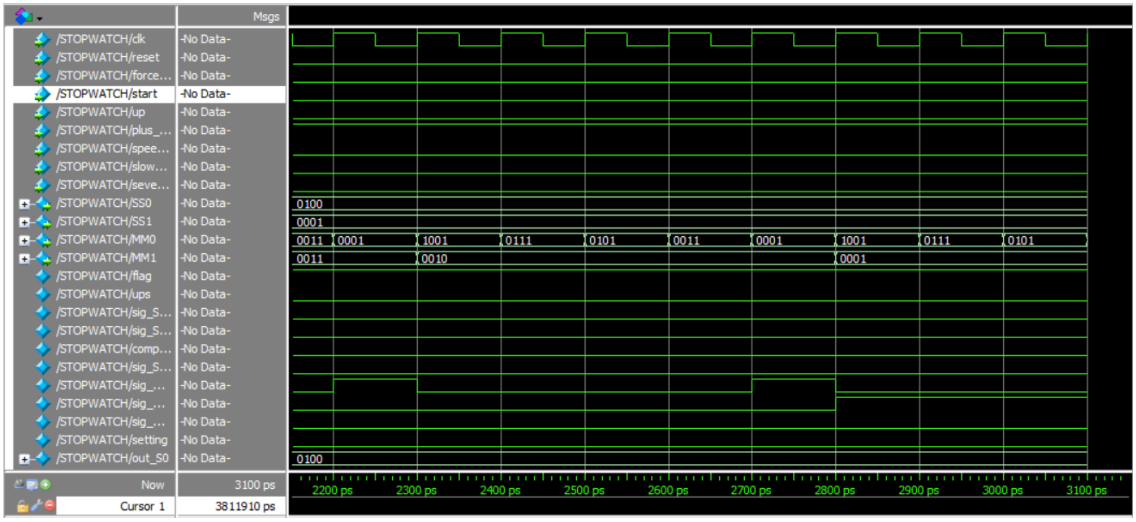


**Figure 73.stop watch**

**2)counting down (-2) while working**

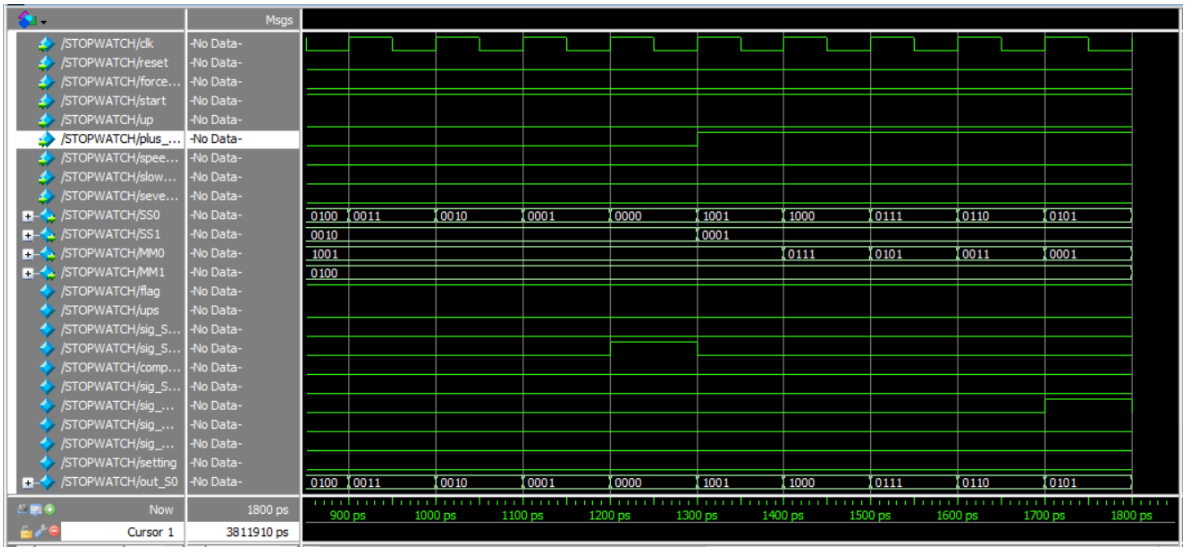


**Figure 74.stop watch simulation**

**3)counting down start from [49:30]:**

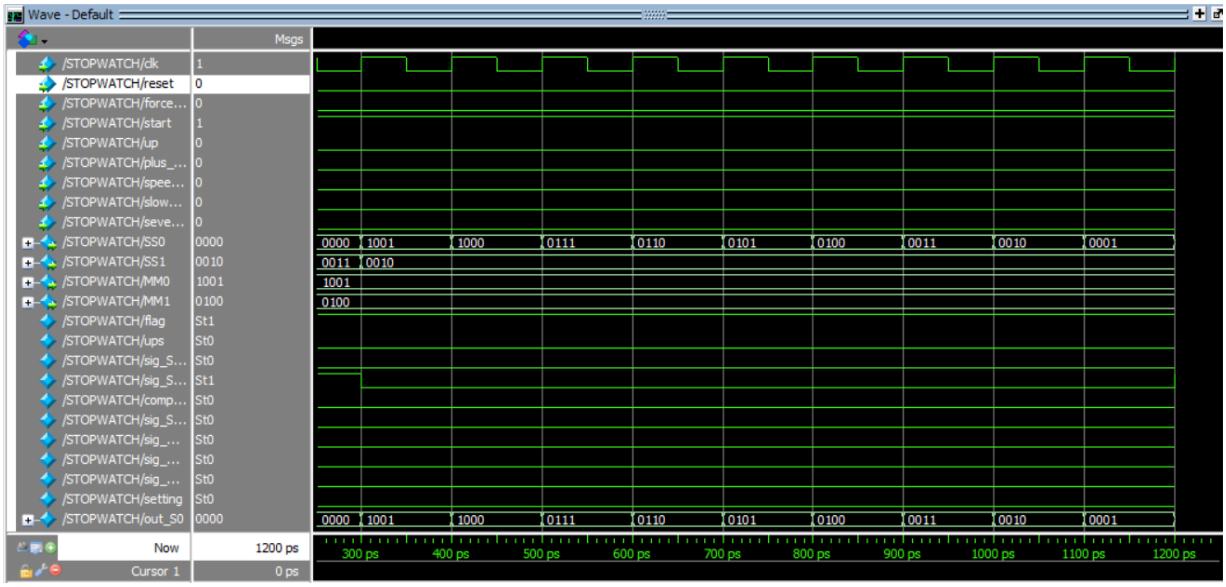


**Figure 75.stop watch simulation**

**4)counting down stop at [10:20]:**

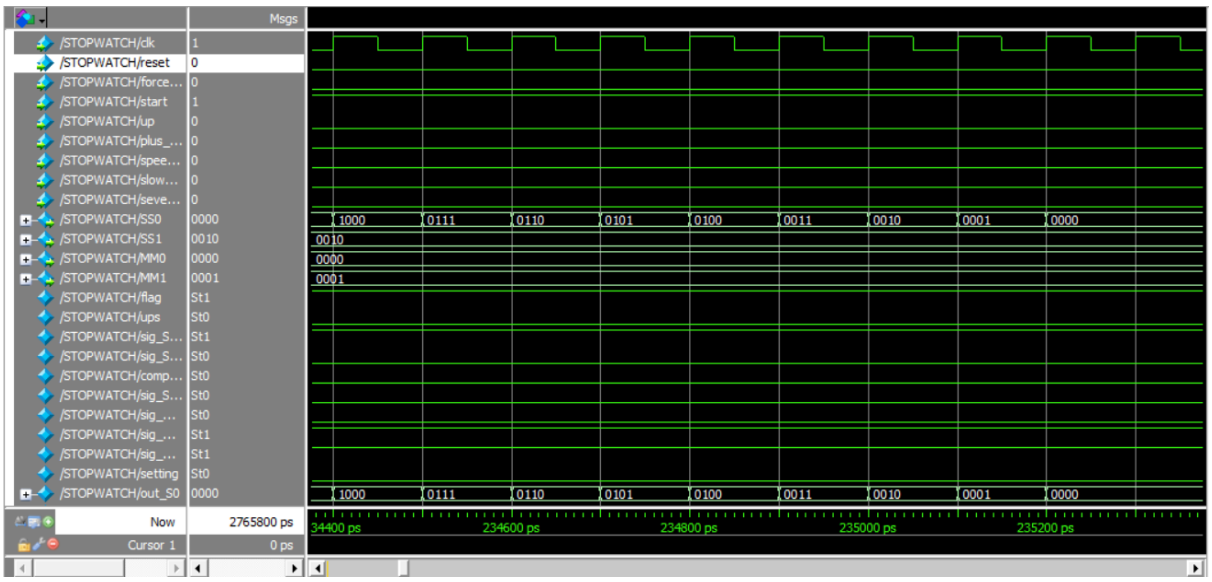


**Figure 76.stop watch simulation**

**5)counting up plus (+2) while stopping:**

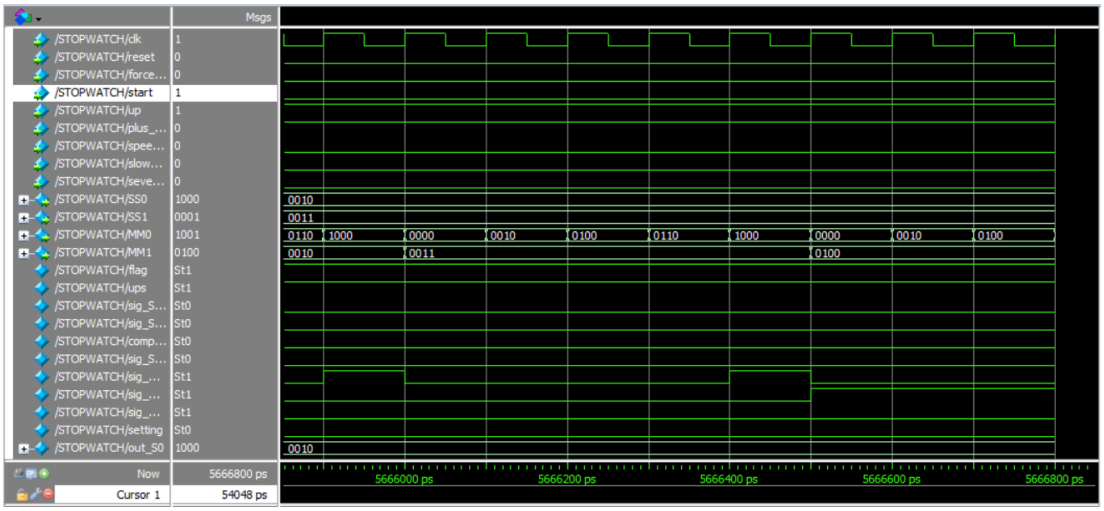


**Figure 77.stop watch simulation**

**6)counting up start from [10:20]:**

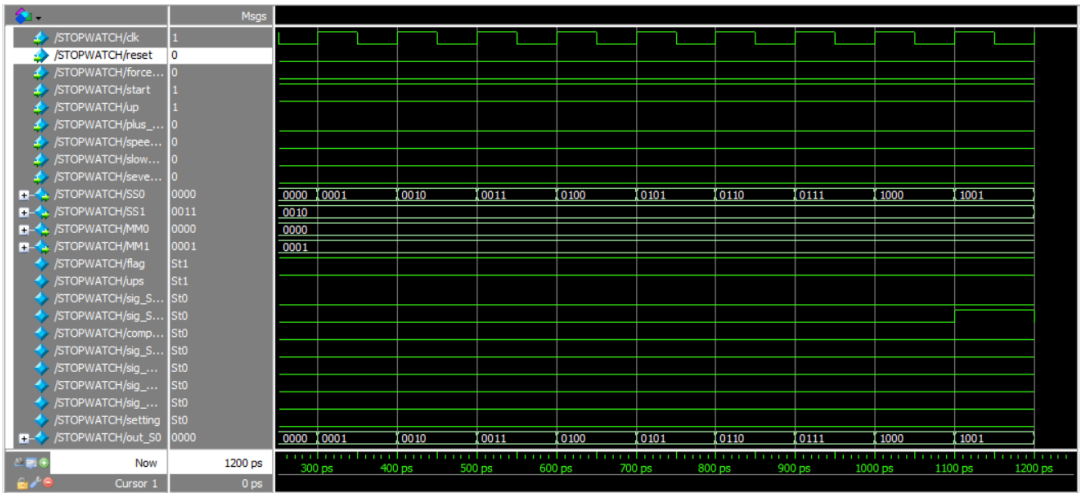


**Figure 78.stop watch simulation**

**7)counting up stop at [49:30]:**

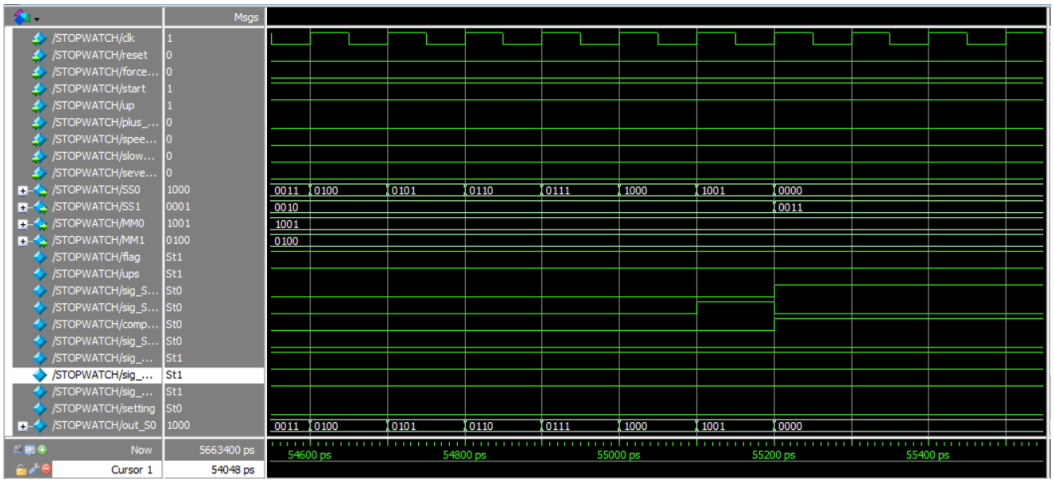


**Figure 79.stop watch simulation**

**8) Error when down change up without stop :**

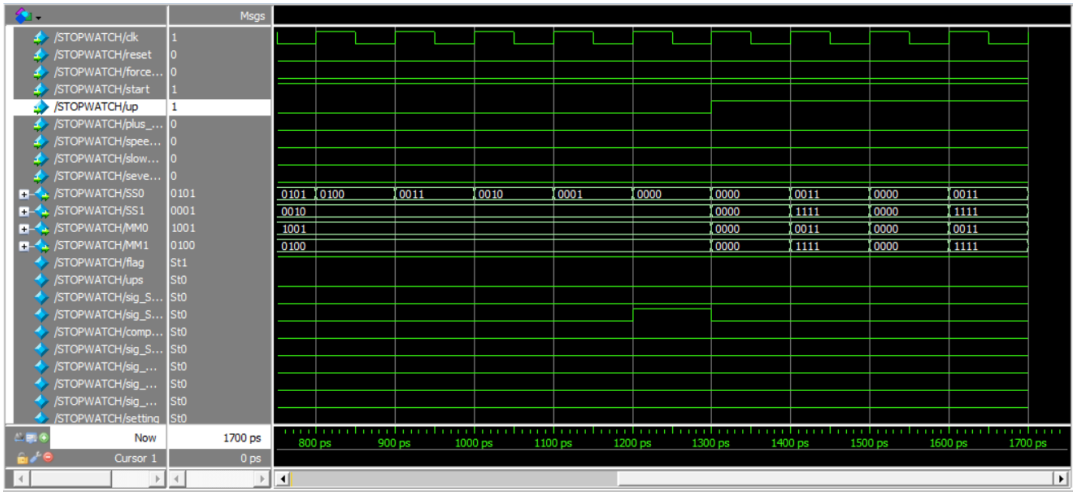


**Figure 80.stop watch simulation**

**9) Error when up change down without stop**

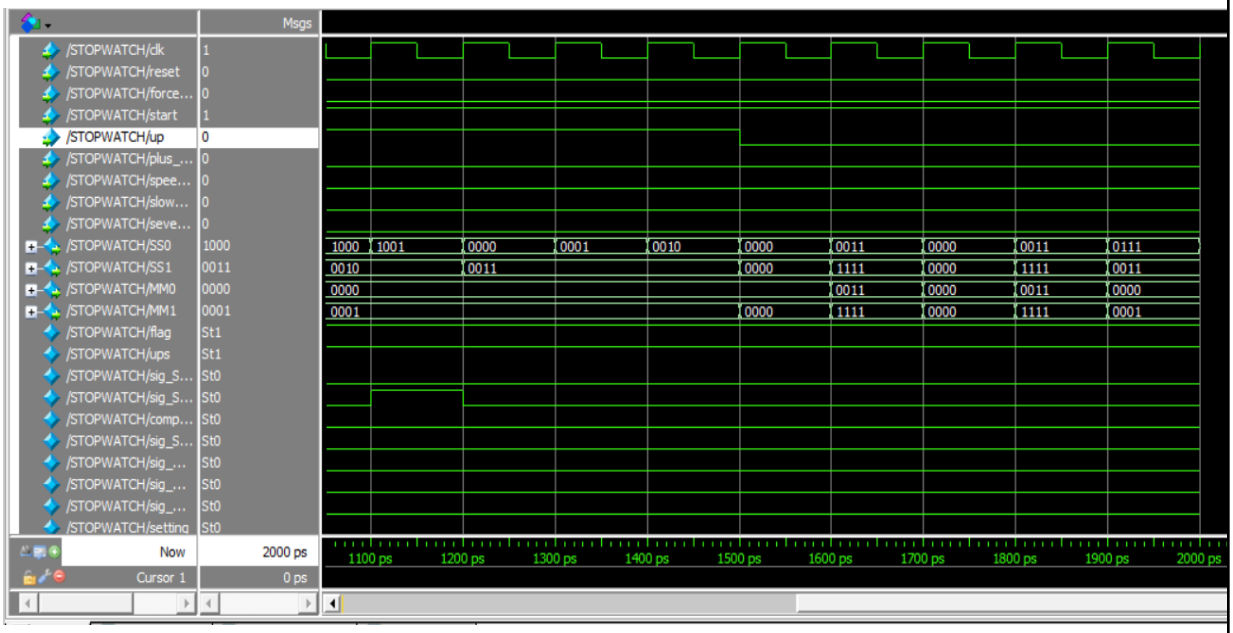


**Figure 81.stop watch simulation**

**10) counting up plus (+2) while working:**

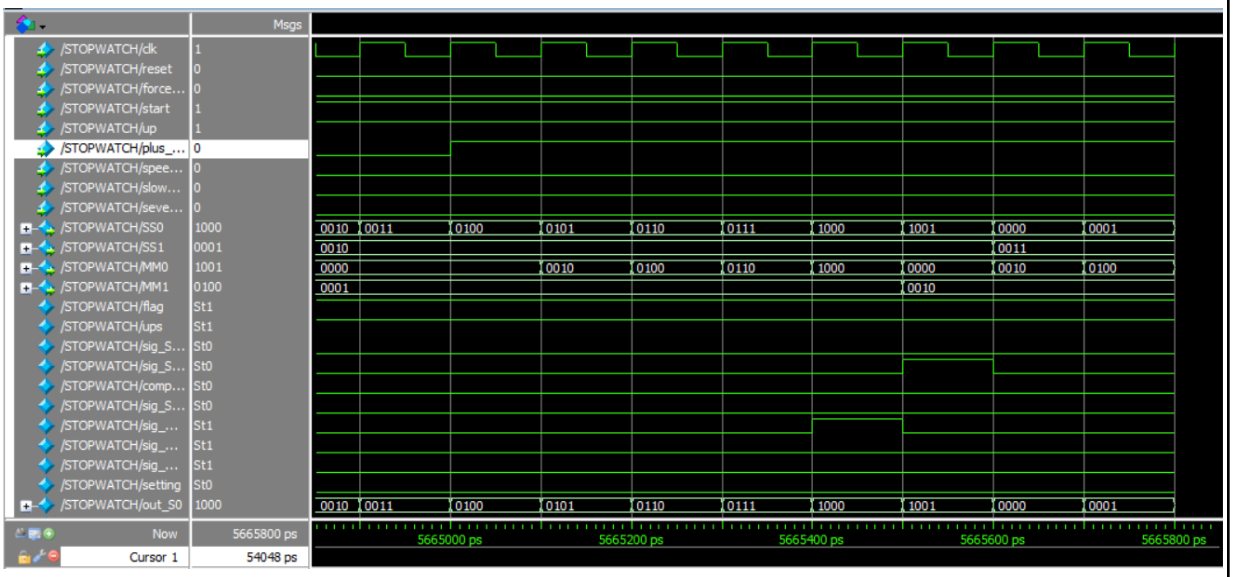


**Figure 82.stop watch simulation**

**11) setting up [49:30] and stop:**

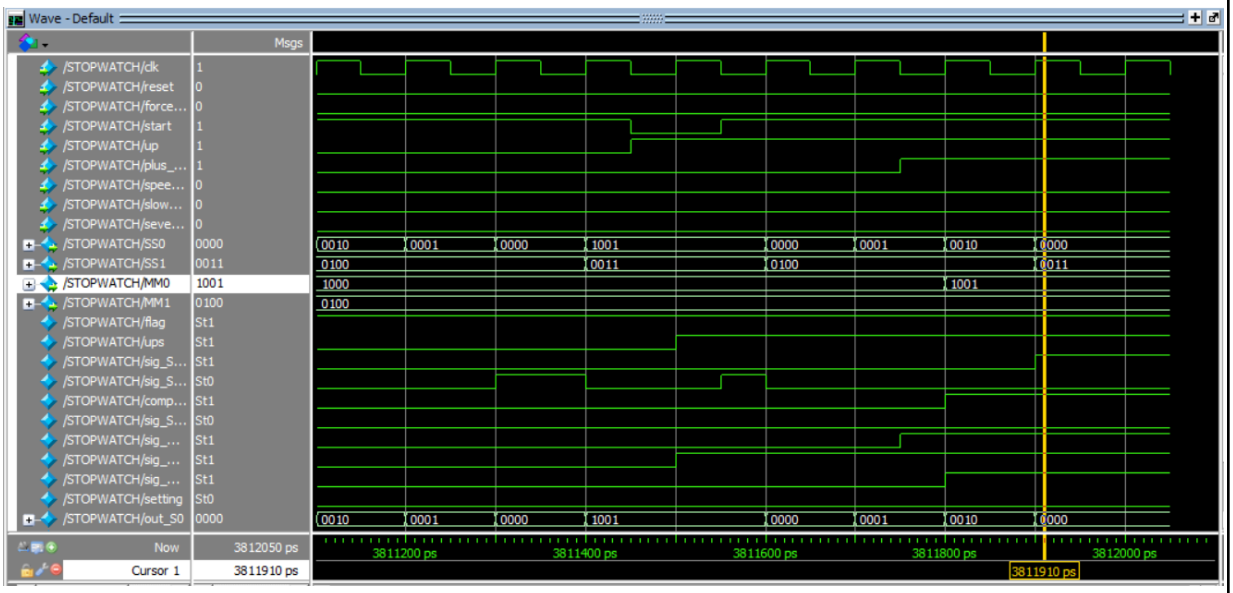


**Figure 83.stop watch simulation**

**12) setting down [10:20] and stop:**

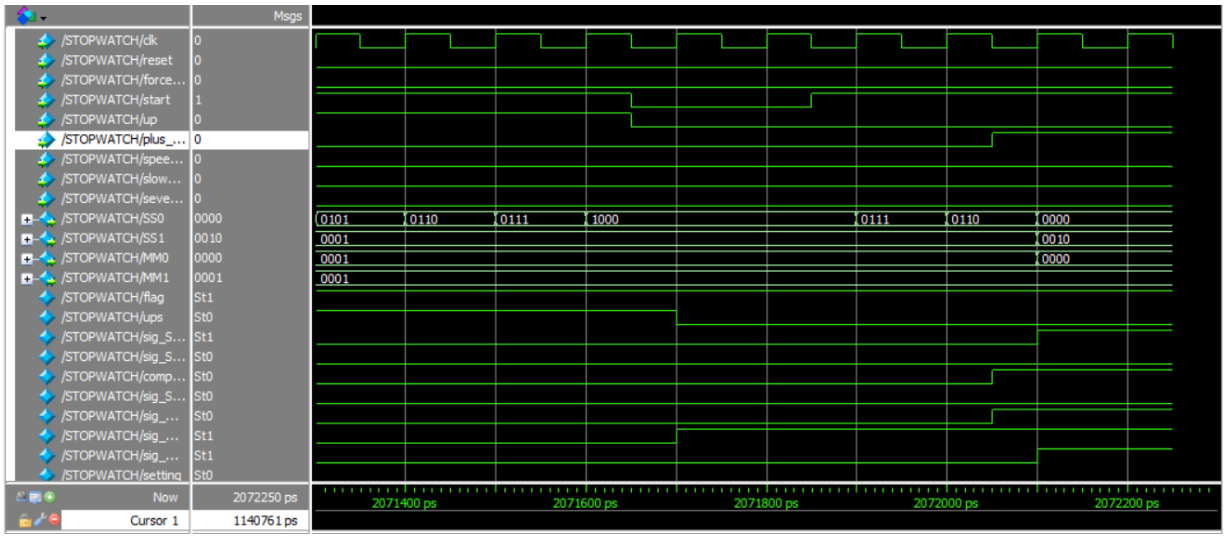


**Figure 80.stop watch simulation**

## Bonus Seven segment part:

Seven segment output simulation:


**Figure 81.seven segment simulation**

## Bonus Error 1 part:

Error code output simulation:

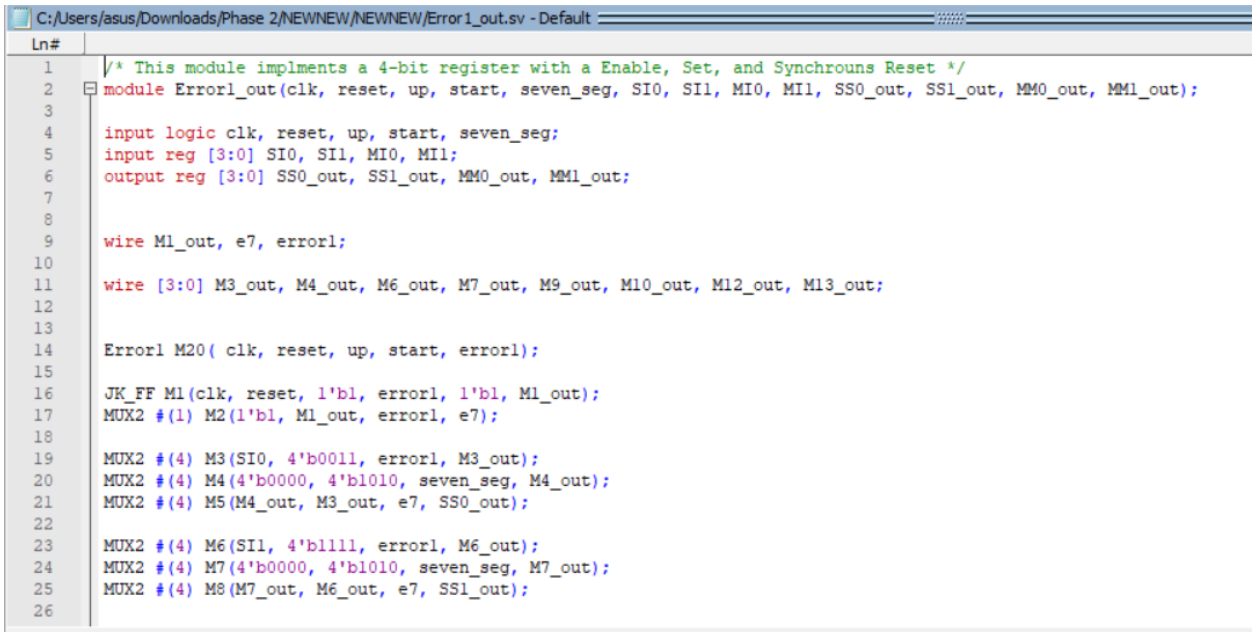


**Figure 82. Error Module implementation**

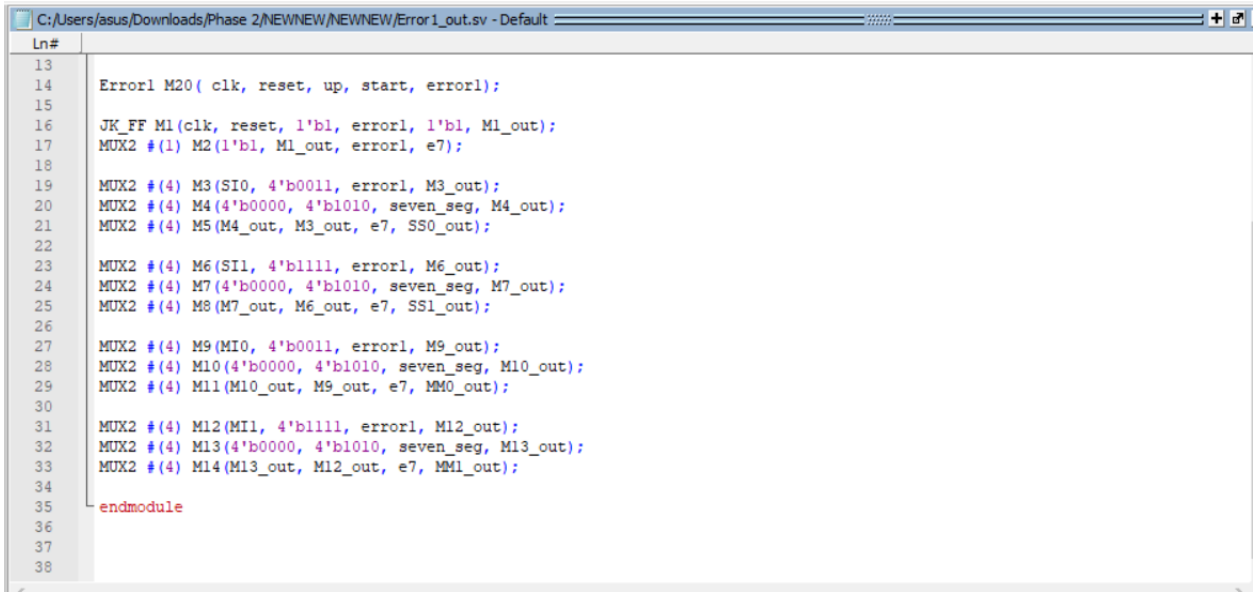


**Figure 82. Error Module implementation**

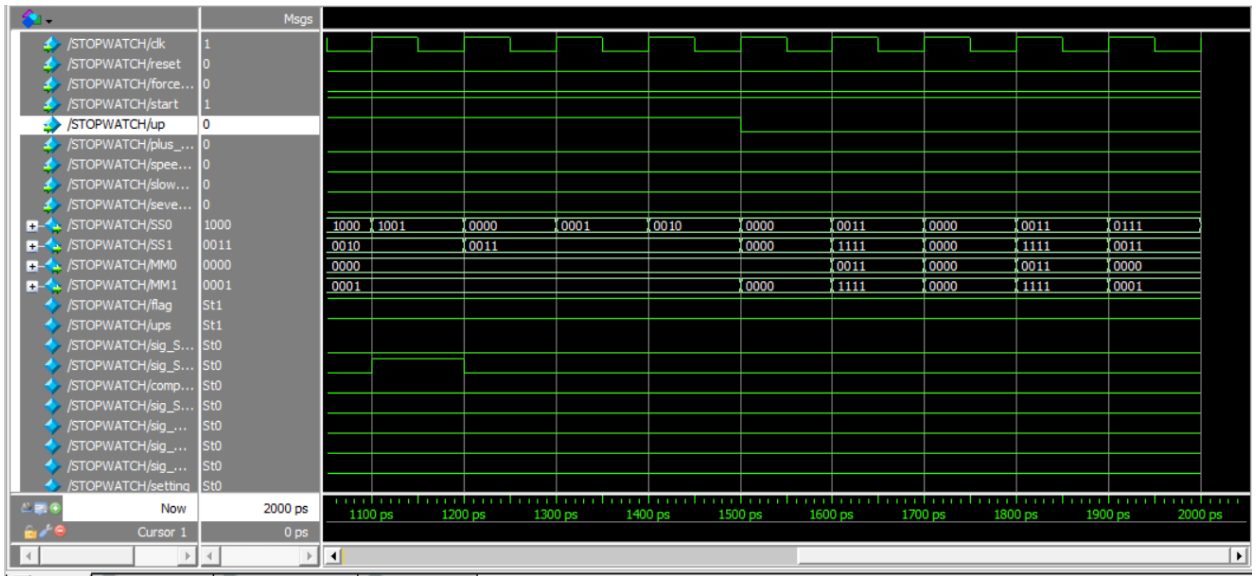


**Figure 83. Error when up changes to down without stop**

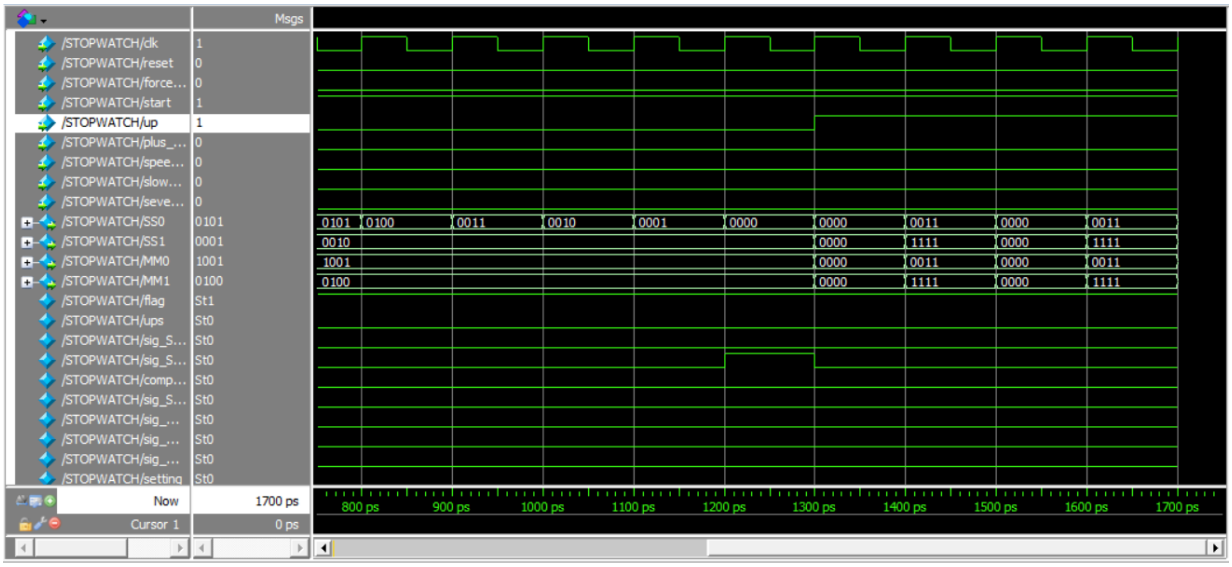


**Figure 84. Error when down changes to up without stop**

C:/Users/asus/Downloads/Phase 2/NEWNEW/NEWNEW/JK_FF.sv - Default

```
Ln#
 1  module JK_FF(clk,reset,EN,J,K,Q);
 2  input logic clk, J,K,reset,EN;
 3  output logic Q;
 4  wire D,and1,not_K,not_Q;
 5  D_FF_SyncR DFF(clk,reset,EN,D,Q);
 6  not gate0(not_K,K);
 7  not gate1(not_Q,Q);
 8  and gate2(and1,not_K,Q);
 9  and gate3(and2,not_Q,J);
10  or gate4(D,and1,and2);
11  endmodule
12
13
```

**Figure 85. JK flip flop Module implementation**

## Bonus Seven Segment Decoder:

```
/*This module implments a 4:7 decoder to convert the 4-bits output from registers into the 7 inputs of the 7 segment*/
module seven_seg_decoder(A, c, d, e, b, a, f, g);
//A-B
input logic [3:0] A;   //Inputs A

output logic a, b, c, d, e, f, g;     //Outputs

wire M1_out, M2_out, M3_out, M4_out, M6_out, M7_out, M8_out, M9_out, M10_out, M11_out, M13_out, M14_out, M16_out, M17_out, M18_out, M19_out, M20_out,
wire M23_out, M24_out, M25_out, M27_out, M28_out, M29_out, M30_out, M31_out, M33_out, M34_out, M35_out, M36_out, M37_out, M38_out;
// Segment a
xor M1(M1_out, A[3], A[1]);
and M2(M2_out, A[2], A[0]);
and M3(M3_out, ~A[3], ~A[2]);
and M4(M4_out, M3_out, ~A[0]);
OR3_mod M5(M1_out, M2_out, M4_out, a); //a
// Segment b
and M6(M6_out, ~A[3], ~A[2]);
and M7(M7_out, ~A[2], ~A[1]);
and M8(M8_out, ~A[3], A[1]);
and M9(M9_out, M8_out, A[0]);
and M10(M10_out, ~A[1], ~A[0]);
OR3_mod M11(M10_out, M9_out, M7_out, M11_out);
or M12(b, M11_out, M6_out);  //b
// Segment c
and M13(M13_out, ~A[3], A[0]);
and M14(M14_out, ~A[3], A[2]);
OR3_mod M15(M13_out, M14_out, ~A[1], c); //c
// Segment d
and M16(M16_out, A[3], ~A[1]);
AND4_mod M17(~A[3], ~A[2], ~A[0], 1'b1, M17_out);
AND4_mod M18(~A[3], ~A[2], A[1], 1'b1, M18_out);
AND4_mod M19(~A[3], A[1], ~A[0], 1'b1, M19_out);
AND4_mod M20(A[2], ~A[1], A[0], 1'b1, M20_out);
OR3_mod M21(M16_out, M17_out, M18_out, M21_out);
OR3_mod M22(M21_out, M19_out, M20_out, d); //d
// Segment e
and M23(M23_out, A[3], A[2]);
AND4_mod M24(~A[2], ~A[1], ~A[0], 1'b1, M24_out);
```

**Figure 86a. Seven Segment Decoder implementation**

```
AND4_mod M25(~A[3], A[1], ~A[0], 1'b1, M25_out);
OR3_mod M26(M23_out, M24_out, M25_out, e); //e
// Segment f
and M27(M27_out, ~A[1], ~A[0]);
and M28(M28_out, A[2], ~A[1]);
and M29(M29_out, A[2], ~A[0]);
and M30(M30_out, A[3], A[0]);
OR3_mod M31(M27_out, M28_out, M29_out, M31_out);
or M32(f, M31_out, M30_out); //f
// Segment g
and M33(M33_out, A[2], ~A[1]);
and M34(M34_out, A[2], ~A[0]);
and M35(M35_out, A[3], ~A[1]);
and M36(M36_out, A[3], A[0]);
AND4_mod M37(~A[3], ~A[2], A[1], 1'b1, M37_out);
OR3_mod M38(M33_out, M34_out, M35_out, M38_out);
OR3_mod M39(M38_out, M36_out, M37_out, g);  //g

endmodule
```

**Figure 86b. Seven Segment Decoder implementation**

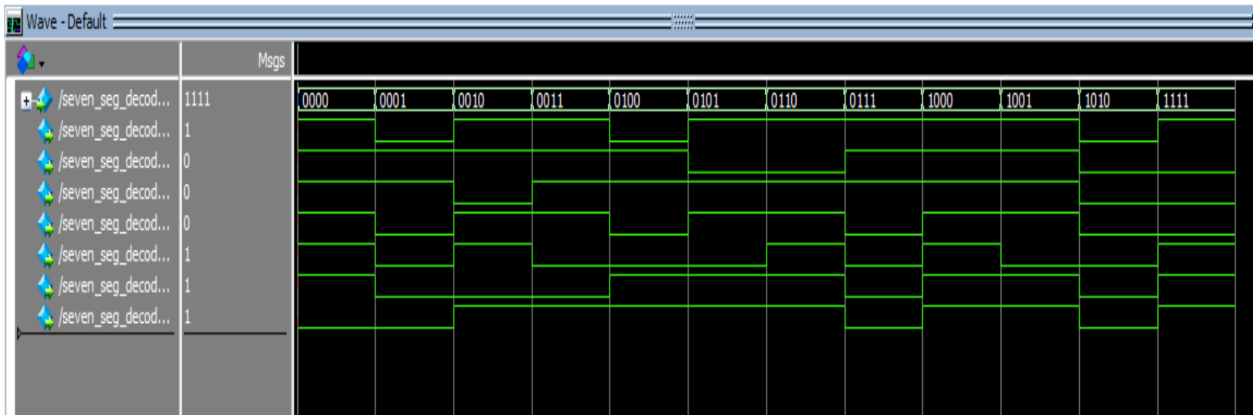


**Figure 87. Seven Segment Decoder Simulation**

## Bonus STOPWATCH with Seven Segment:

```
/*This module implments the functionality of stopwatch using 7 segments*/
module STOPWATCH_Seven_Seg(in_clk, reset, start, up, plus_min2, speedup, slowdown,  out_S0, out_S1, out_M0, out_M1);
input logic in_clk, reset, start, up, plus_min2, speedup, slowdown;  //inputs
output reg [6:0] out_S0, out_S1, out_M0, out_M1;                     //outputs
wire [3:0] SS0, SS1, MM0, MM1;
wire seven_segment_flag;
assign seven_segment_flag = 1'b1;
wire ups, sig_S1_S0, sig_S0_S1, comp_sig_S1_S0, sig_S1_M0, sig_M0_M1, sig_M1_M0, sig_M0_S1,setting,M_set, force_reset;
//Stopwatch that outputs 4 bits
STOPWATCH_LEDS K1(in_clk, reset, start, up, plus_min2, speedup, slowdown , SS0, SS1, MM0, MM1);
//Decoding each 4-bits output into 7 bits
seven_seg_decoder KS0(SS0, out_S0[0], out_S0[1], out_S0[2], out_S0[3], out_S0[4], out_S0[5] ,out_S0[6]);
seven_seg_decoder KS1(SS1, out_S1[0], out_S1[1], out_S1[2], out_S1[3], out_S1[4], out_S1[5] ,out_S1[6]);
seven_seg_decoder KM0(MM0, out_M0[0], out_M0[1], out_M0[2], out_M0[3], out_M0[4], out_M0[5] ,out_M0[6]);
seven_seg_decoder KM1(MM1, out_M1[0], out_M1[1], out_M1[2], out_M1[3], out_M1[4], out_M1[5] ,out_M1[6]);

endmodule
```

**Figure 88. STOPWATCH with Seven Segment implementation**

# Work Distribution:

The requirements of Phase 1 and how we distribute them over us are as follow:

1. **Identify more than one design approach.**
   We started by pointing out the possible approaches and found out that there are 2 main approaches for implementing counters: Synchronous and Asynchronous. We divided the two approaches among us to search and know more about the advantages and disadvantages of each approach as follows:

| | |
|---|---|
| **Ehab Mansour** | Synchronous Counters |
| **Ibrahim Hamada** | Asynchronous Counters |
| **Areej Mohamed** | Synchronous Counters |
| **Nasrah Mohamed** | Synchronous Counters |
| **Yasmeen Abosaif** | Asynchronous Counters |

2. **Select the most suitable approach, given the system's functional requirements.**
   After that we selected the Synchronous design approach based on the obtained advantages which were introduced previously.

3. **Clearly design of the system building blocks hierarchically**
   We collaborated together to design the circuit of the system so that it satisfies the system requirements.
   After that, we figured out the needed blocks and divided them among us so that each one is responsible for writing codes and simulating some SystemVerilog Modules.

4. **A simulation of each building block of the digital system working separately.**

| | |
|---|---|
| **Ehab Mansour** | Clock Divider Modules, S1 |
| **Ibrahim Hamada** | D_FF, Register4, S0 |
| **Areej Mohamed** | MUX2, MUX4 |
| **Nasrah Mohamed** | FA, FA4, Subtractor4, |
| **Yasmeen Abosaif** | AND4, OR3, NOR4, Comparator4 |

# Work distribution phase 2:

- We have completed what we started in the first stage, and we have divided the work. Modifications were made regarding the circle, and this was done with the participation of all team members.
- We discussed how the bonus part would be added, then it was divided into implementation and simulation.
- Ensure that team members are aware of how to use FPGA on their devices

| | |
|---|---|
| **Ehab Mansour** | M0, CIRCUIT DESIGN MODIFICATION and implementation |
| **Ibrahim Hamada** | STOP WATCH Combining Modules,FPGA |
| **Areej Mohamed** | M1,Error Code , review testing , documentation, FPGA |
| **Nasrah Mohamed** | STOP WATCH Combining Modules ,CIRCUIT DESIGN MODIFICATION,FPGA |
| **Yasmeen Abosaif** | Seven segments , draw a circuit ,review and modify codes regarding phase 1 that are suitable for new modification. |

## References :


- Harris, D., & Harris, S. (2016). *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann.
- Digital electronics and logic design tutorials. (n.d.). GeeksforGeeks. Retrieved December 28, 2022, from https://www.geeksforgeeks.org/digital-electronics-logic-design-tutorials/