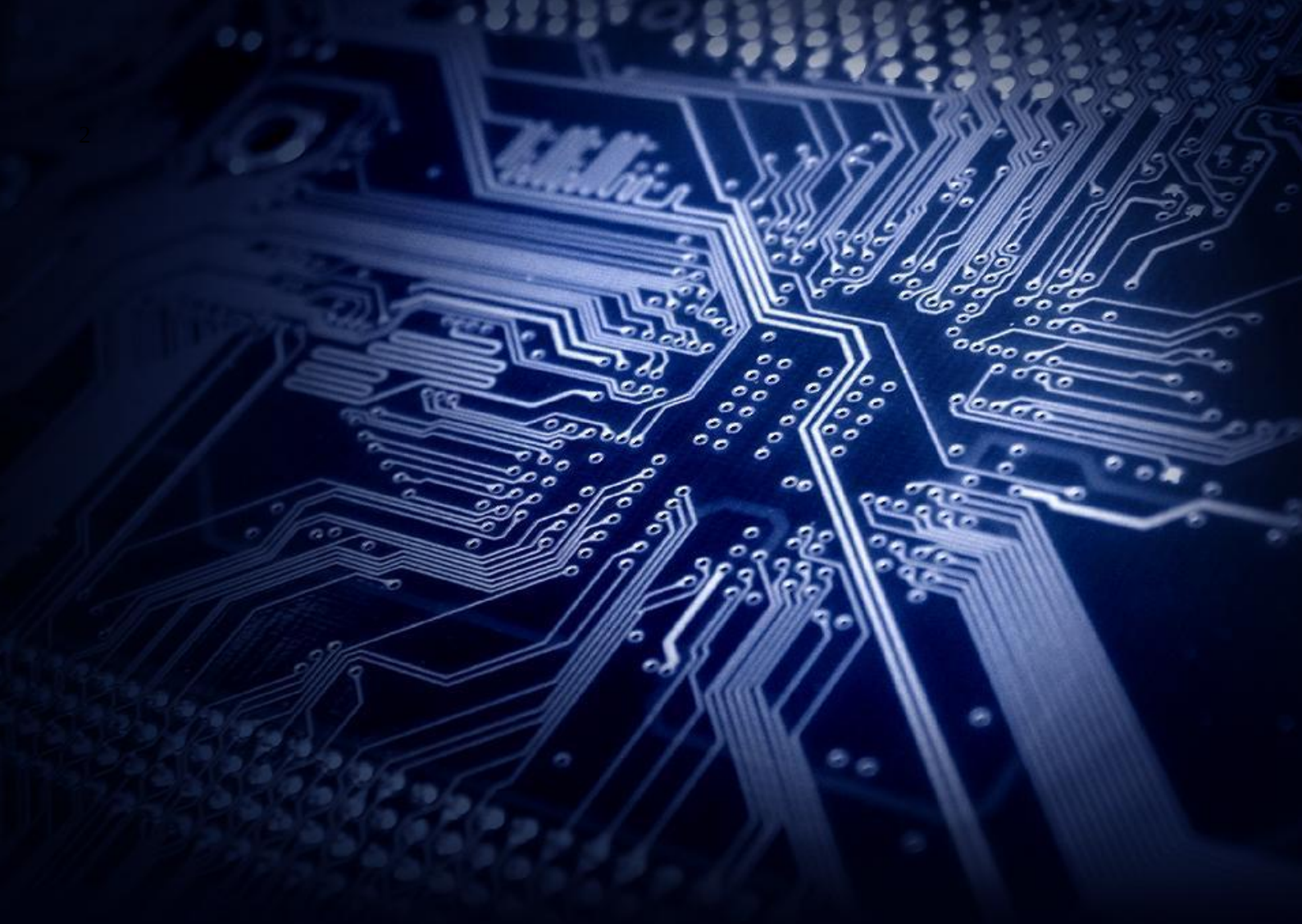


Digital Design & Computer Architecture

CIE 239
Fall 2022
Final Project Report

Team:
1- Adham Saad 202100163
2- Mohamed Morshedy 202100372
3- Aser Osama 202101266
4- Mohamed Magdy 202101520

Contents

I. Abstract.

II. Introduction.

III. Methodology.

IV. References.

- Phase One:
    1- Control Unit.
    2- Counters.
    3- Clock Divider.

- Phase Two:
    1- System Integration.
    2- Seven Segments Display.
    3- Add 2 Minutes.
    4- Subtract 2 Minutes.
    5- Error Code 55:55 Implementation.
    6- Error Correction and Bug Fixation of Phase One.

Note: All source code files are attached along with the report as a separate ZIP file as well as a video link for our project undergoing our testing at the end of the document.

Abstract.

- This is the report for the documentation of the final project for course CIE 239: Digital Design & Computer Architecture.

- The goal is to implement and verify the functionality of a stopwatch system using HDL (SystemVerilog IEEE 1800-2017) and the De0-Nano FPGA development platform.

- All modules/sub-modules are built from the ground up.

- This report will document our thought process and decision in building our project stopwatch.

Introduction.

The project is broken down into two phases:

- Phase one: Implementation and simulation of fully functional stand-alone modules/submodules of the stopwatch.

- Phase two: Integration of these modules/sub-modules together to build up the full system of the stopwatch.

- Main specifications:
  1- Count up and down.
  2- Set and reset timer.
  3- Minimum value 10:20.
  4- Maximum value 49:30.
  5- Add and subtract two minutes.
  6- Speed counting up and down (2x).
  7- Resume counting after a pause.

- Extra features:
  1- Seven segment display instead of LEDs. **(Done)**
  2- Handle the case when the counting mode is changed while counting. **(Not Done)**
  3- Handle the case when both speed up and speed down are pressed. **(Done)**

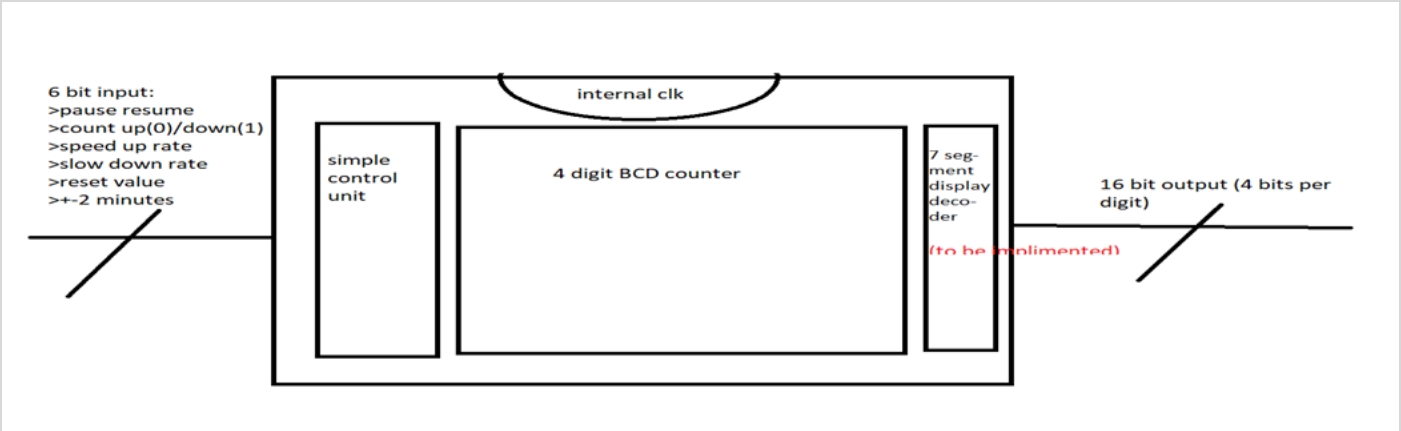
Methodology.
——----Phase One Summary, Updated ——

Why did we choose our design?

Our design initially seemed more complicated than other choices such as the ripple counter as it could require more sequential logic. However, it in return gave us a lot more freedom to control it more specifically which gave us the ability to do things such as use the same counter for counting up and down which is a very useful feature not only to save the cost of the flipflops and other components but to make our design even more compact. Also, using a synchronous design means we will have to worry less about the delays in the timer, as in the ripple counter we were expecting around 1 second of delay as it got more complicated due to the nature of the components. We also had the choice between many different flipflops for our designs, we initially tried to use T-Flip-Flops asynchronously but after getting halfway through the design we decided to move to a synchronous design due to the previously mentioned issues with asynchronous. When we made that switch, however, we realized that using T-Flip-Flops would be overly limiting and exhausting to try and make work correctly so we moved on onto the better option of D-Flip Flops (we avoided JK-Flip Flops as they would be way too complicated for our application of a stopwatch.)

The nature of our design:

Inside our design, we used many different components. This will be represented by the diagram shown below.



1- The control unit:

The control unit contains a clock divider and some combinational logic that manipulates the input.

-The clock divider (not yet designed) will take 2 inputs, speed up and down, and output the clock signal itself. This is the truth table which will represent all possibilities for it:

2x	1/2x	Clk
0	0	2Hz (default speed for our counter)
1	0	4Hz
0	1	1Hz
1	1	2Hz

2- The Counter's building blocks:

The counter is the heart of our design as everything else is designed behind how it itself works. We made our counter using *D-Flip Flops, combinational logic gates, 2:1 Multiplexers, 1:2 Demultiplexer, 3 and 4 input comparators*, and our own custom components made using a mix of the rest such as:

"e-not": this is a not gate with an enable signal, this is used to cause some changes to the counter if a signal of 1 is sent to it, this signal can be the "count mode" signal such that when that is 1 it means we are counting down so this modifies certain wires accordingly.

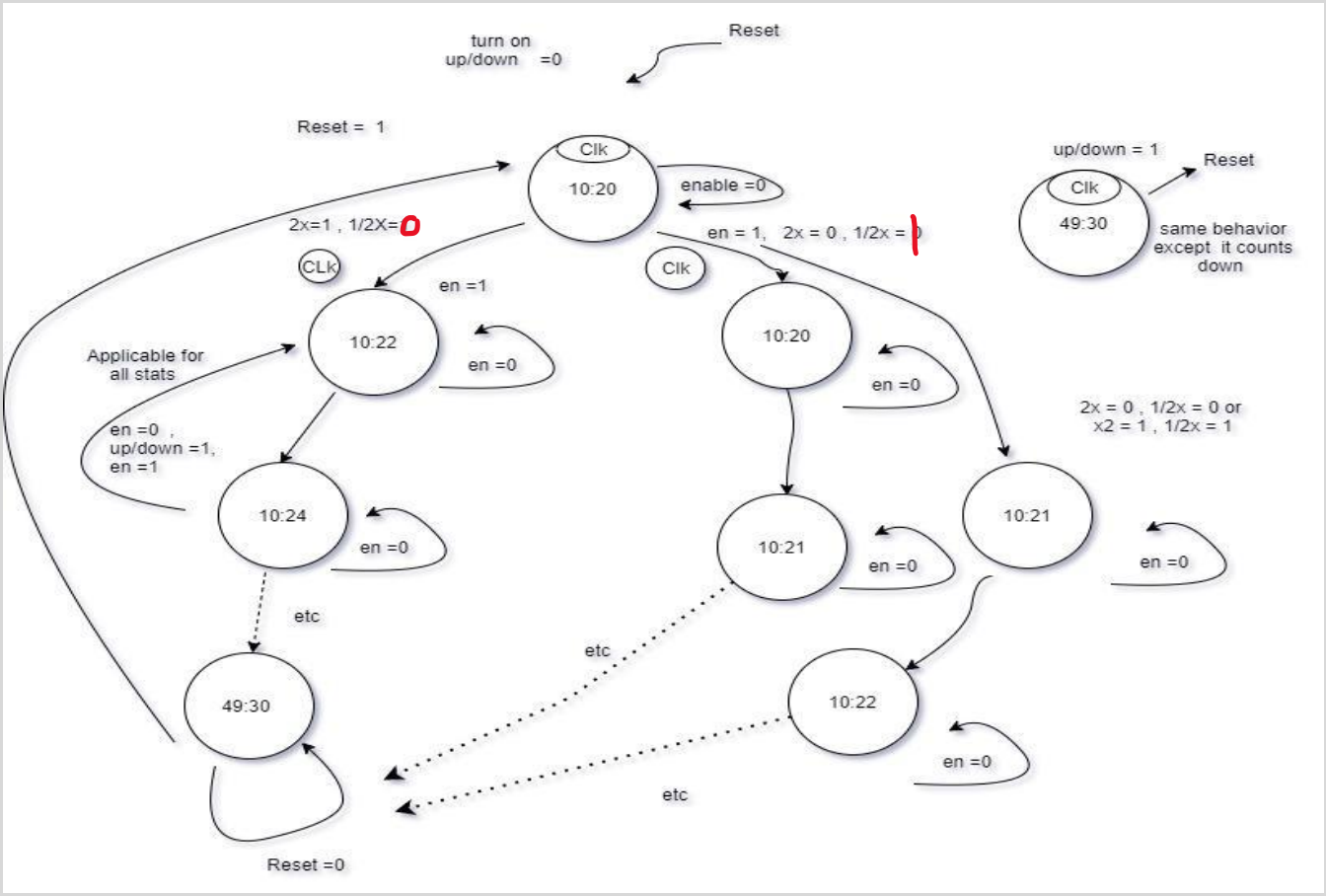
"*MaxComp":* this is a 14-bit comparator that checks if the current value is while counting up, the max value of the counter (49:30) as when this is reached it outputs a signal which is later used to force the counter to pause itself.

"MinComp": same as MaxComp however its signal is only considered when the counter's counting mode is = 1 which means its counting down. It checks the current value vs (10:20) instead.

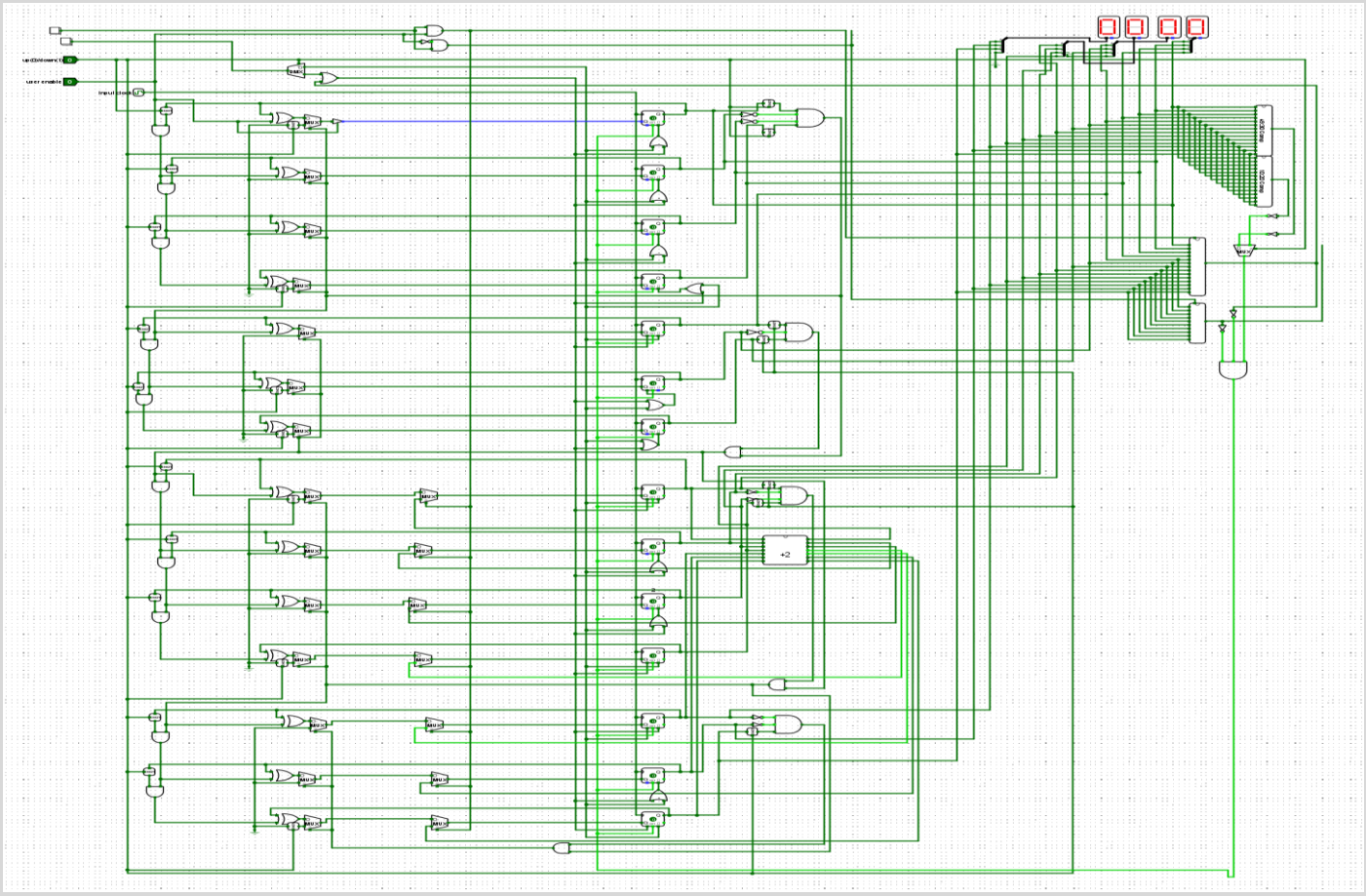
"UpOf": used alongside the "add 2 minutes button", it checks if when we add 2 minutes, the counter overflows and if it can detect the said condition, its output signal is then used to force the value of the counter to its maximum (49:30) value and pauses it.

"*DownOf":* it is the same as the UpOf however it only operates when the counter is in the counting down mode and its output signal is used to force the value of the counter to its minimum instead (10:20) and pause it.

State Diagram:

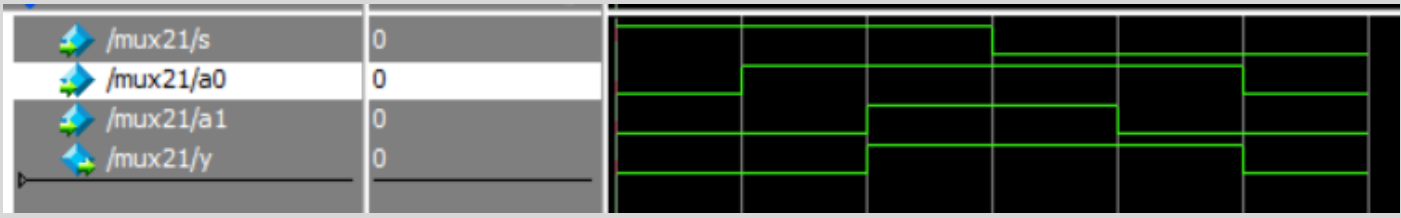


- The Counter's schematic:



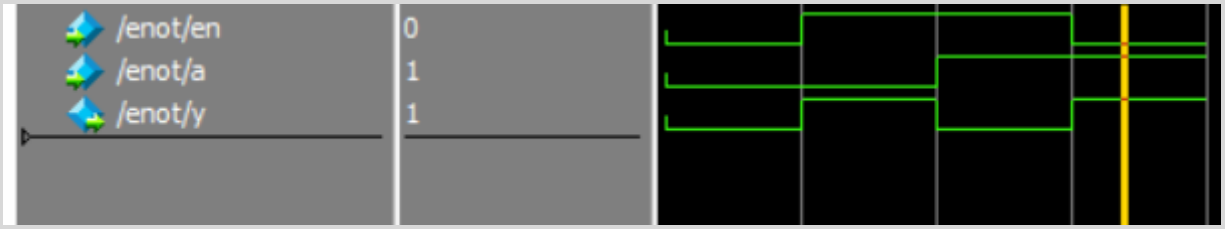
Our Made Components

1. **2:1 multiplexer**:
   If the selection input is 1, the output will be the second input. If the selection input is 0, the output will be the first input.



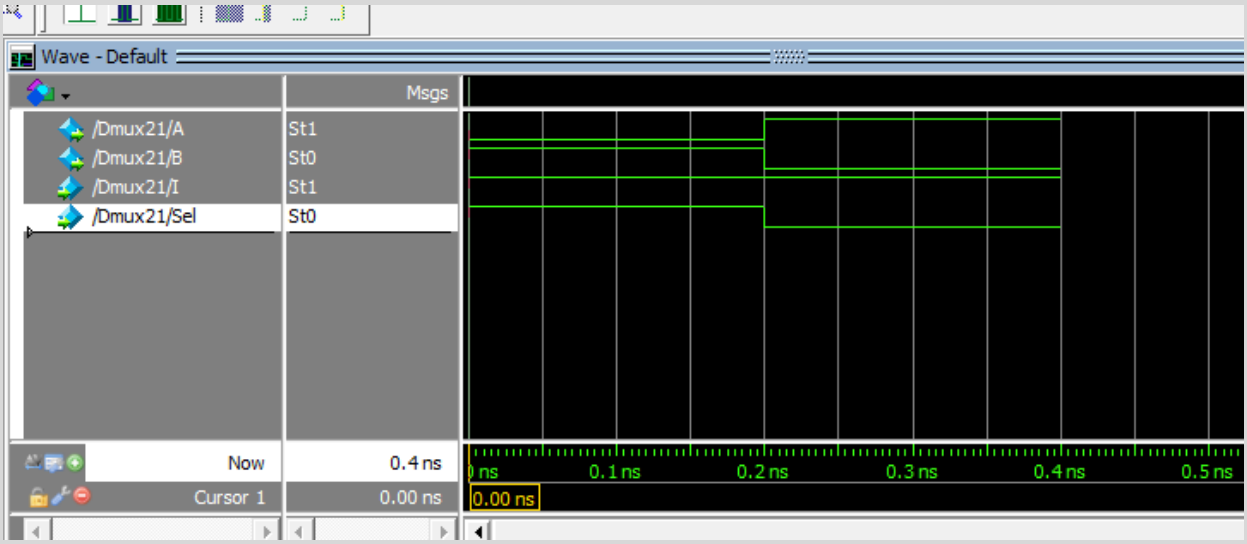
2. **E- not** (not-with-enable):

   This takes an input (a) and a selection input (en), and produces an output (y) that is the inverse of (a) if en=1, and (a) if en=0. The module is implemented using a not gate and a 2-to-1 multiplexer.



3. **2:1 Demux:**

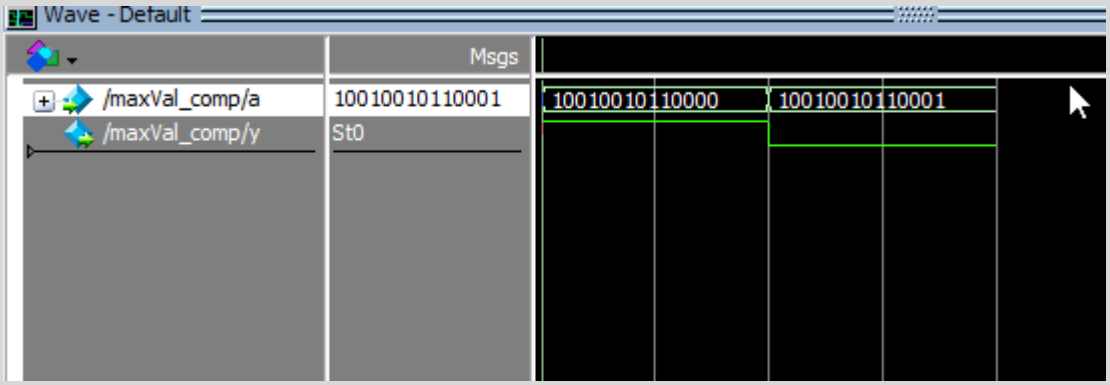
   When the selector is (0), (A= the input), and (B=0). If the (selector =1) then, (A=0) and (B = the input). This is internally just a bunch of NOTs and ANDs.



4. **Maximum value comparator:**

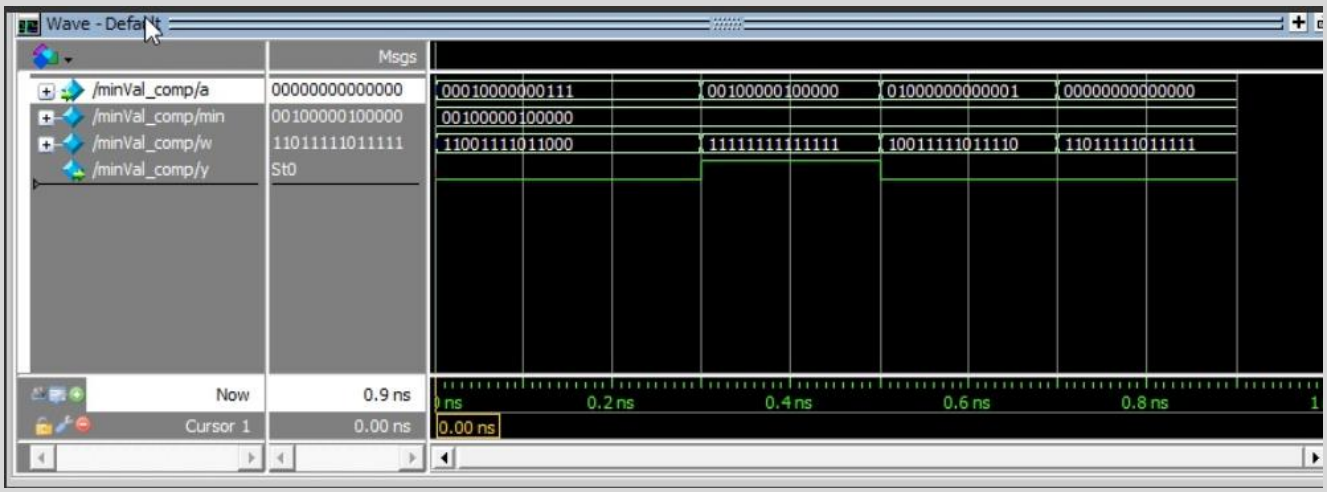
If the entered value does not equal 49:30, the output will be false.

This module takes two inputs: a 14-bit input (a) and produces one output (y). The module contains a 14-bit register (max) initialized to the constant value "10010010110000" and 14 XNOR gates (XNOR). The function of the module is to compare the input (a) to the value stored in the max register and set the output (y) to a logic 1 if all the bits in the input vector are equal to the corresponding bits in the max register, and a logic 0 otherwise. To do this, the XNOR gates are used to compare each bit of the input to the corresponding bit in the max register, and the AND gate at the end combines the results of these comparisons. If all the bits in the input vector match the corresponding bits in the max register, the output of each XNOR gate will be a logic 1 and the AND gate will produce a logic 1 output. If any of the bits do not match, at least one of the XNOR gates will have a logic 0 output, and the AND gate will produce a logic 0 output. This module is like the min_Val_comp module but compares the input to a different constant value stored in the max register.



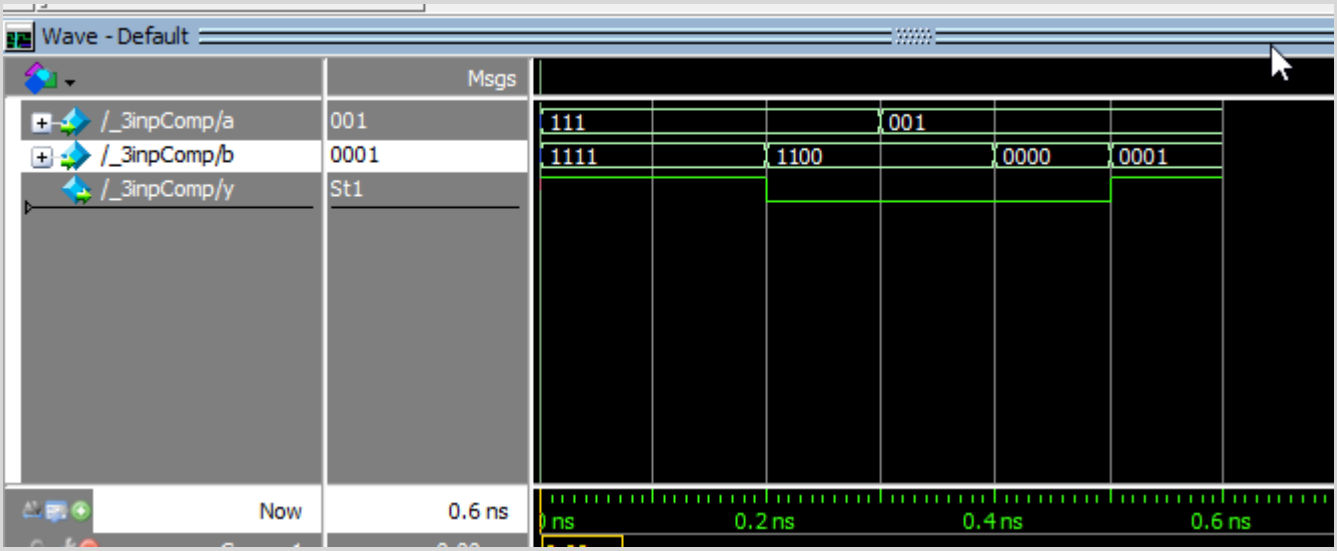
5. **Minimum value comparator**:

This module has two inputs: a 14-bit input (a) and produces one output (y). The module contains a 14-bit register (min) initialized to the constant value "00100000100000" and 14 XNOR gates (XNOR). The function of the module is to compare the input (a) to the value stored in the min register, and set the output (y) to a logic 1 if all the bits in the input vector are equal to the corresponding bits in the min register, and a logic 0 otherwise. To do this, the XNOR gates are used to compare each bit of the input vector to the corresponding bit in the min register, and the AND gate at the end combines the results of these comparisons. If all the bits in the input vector match the corresponding bits in the min register, the output of each XNOR gate will be a logic 1 and the AND gate will produce a logic 1 output. If any of the bits do not match, at least one of the XNOR gates will have a logic 0 output, and the AND gate will produce a logic 0 output.



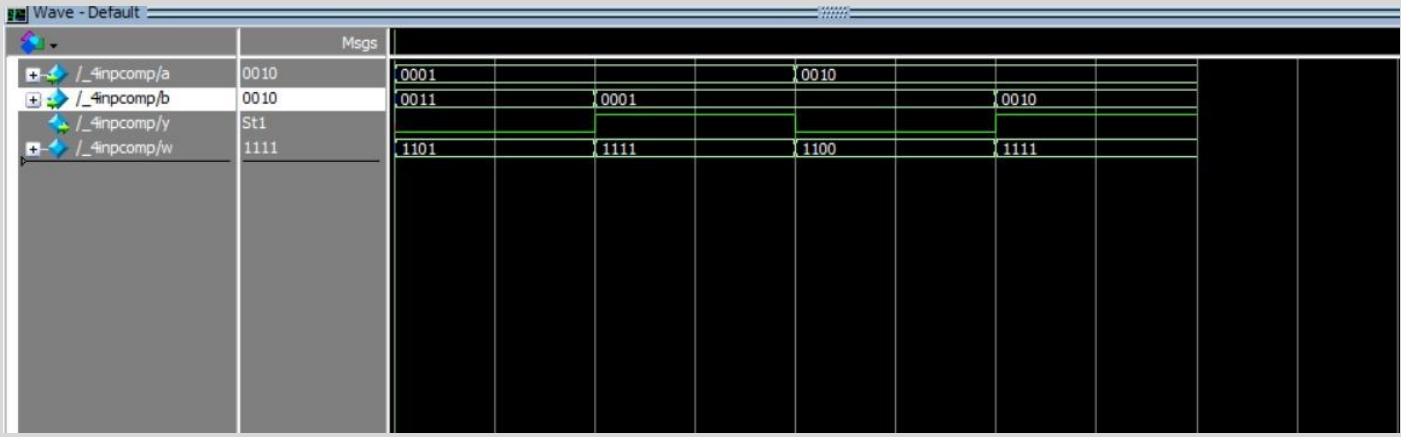
6. **3-Input comparator**, outputs true only when a = b.

   This module compares two 3-bit vectors, a and b, and produces a single output y. The output is 1 if all of the corresponding bits in a and b are equal, and 0 otherwise.

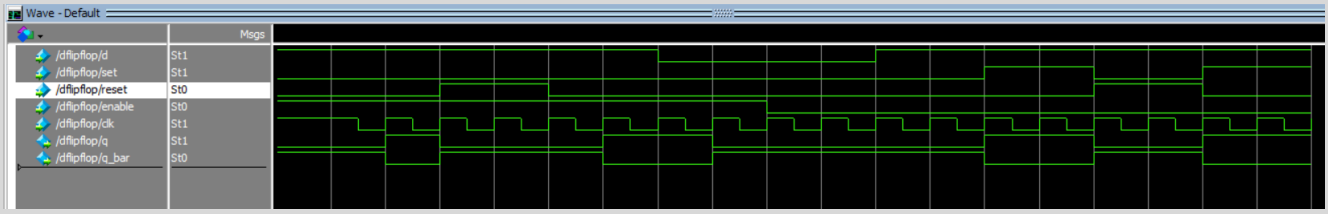


7. **4-Input comparator,** outputs true only when a = b.

   This module compares two 4-bit vectors, a and b, and produces a single output y. The output is 1 if all of the corresponding bits in a and b are equal, and 0 otherwise.



8. **D-flip-flop** This is a standard D-flip-flop with inputs for data, set, reset, enable, and clock, and outputs for the stored value and its inverse.



10. & 11: overflow detectors for the add and subtract 2 minutes: (Completed in Phase 2)

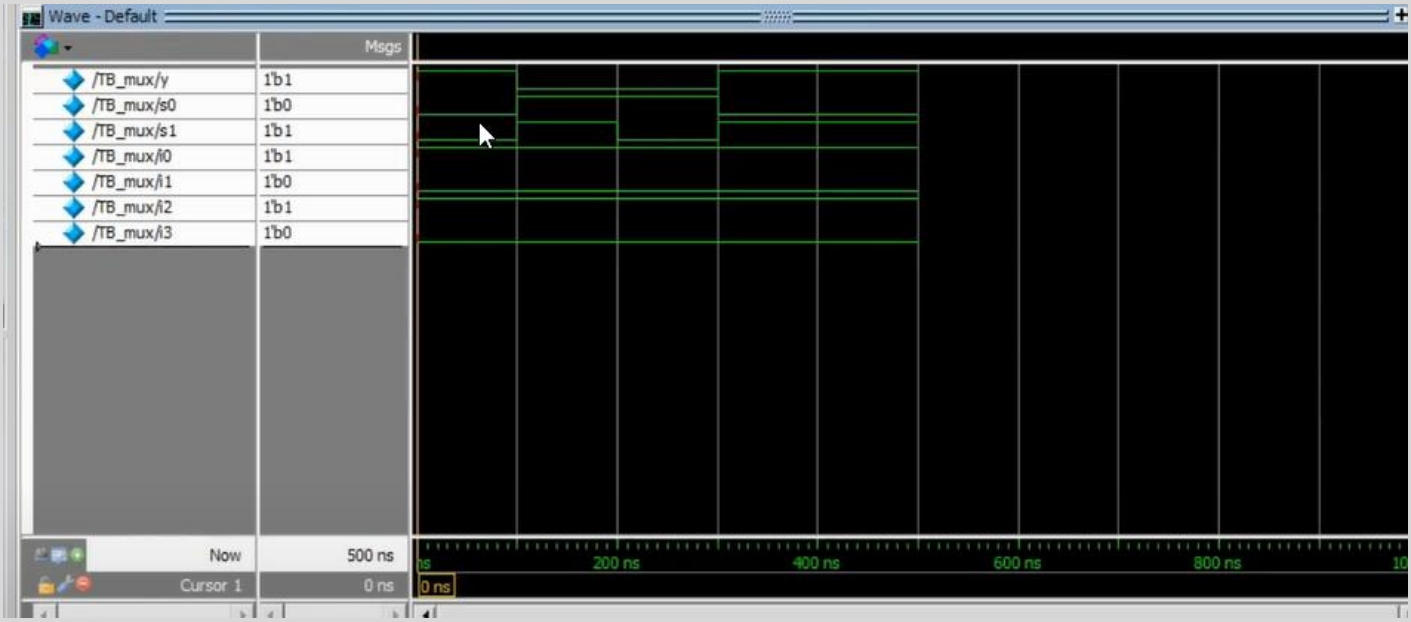
They are modules that detect if when pressing add 2 or subtract 2, the counter will overflow (bypass 4930 or 1020 in it's respective counting mode). When it detects that, it outputs a signal of 1 which is used in the counter to call the reset signal of the opposite counting mode so it hits and stops at maximum (so, when time is 4731 and counting up, it outputs 1, and when user presses plus 2, it hits the reset button for count down to

force the value to be 4930). It works by checking digit by digit if the value would cause an overflow, prioritizing the most significant digits first. So, in the count up if it is 48:XX it ignores the seconds and instantly outputs a signal of 1. In count down if it is 2X:XX it instantly outputs a 0 without considering what the other digits could possibly have. It is compatible with edge cases such as 47:30 being accepted and 47:31 being denied. It also accepts edge cases such as 49:XX and instantly outputs a 1. It is simply a column of comparators that all go into an OR gate per digit and a bit AND gate for all digits. However, there are separate ANDs for more significant digits such as when it is 48 or 49 that AND is active therefore all other digits are invalid and a 1 is outputted. It all goes into a mux that takes a user input selector. When user selector is 1, aka the add 2 button is pressed and it's in the correct count mode (simple and gate), the value of the overflow detector is outputted, else, it constantly outputs a 0.

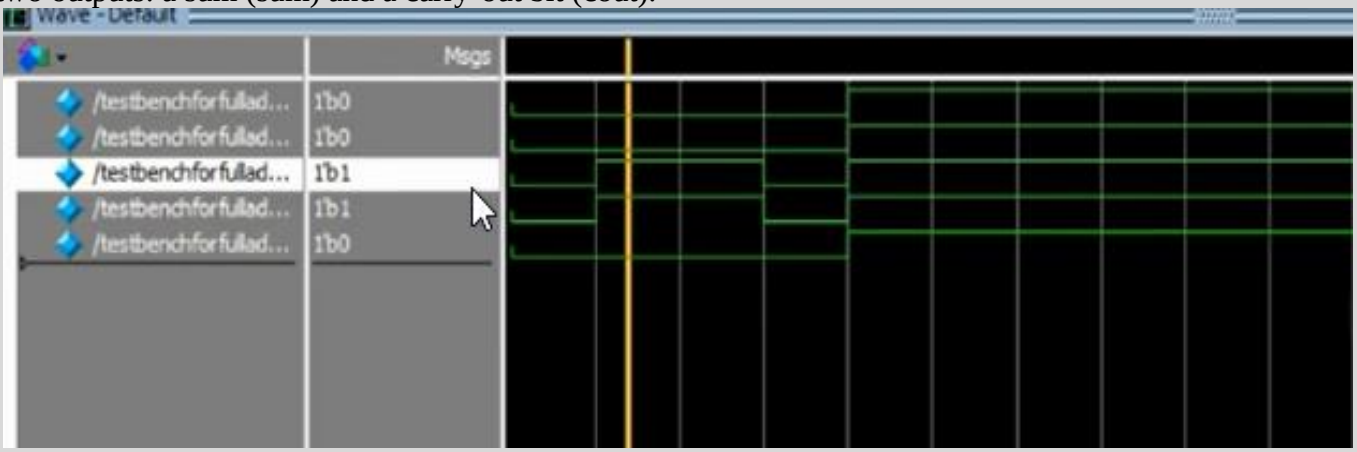
Kindly view them in working in the counter simulation below as due to the nature of Verilog their inputs and outputs had to be changed around so it is difficult to test independantly

——-————————Phase Two——————————

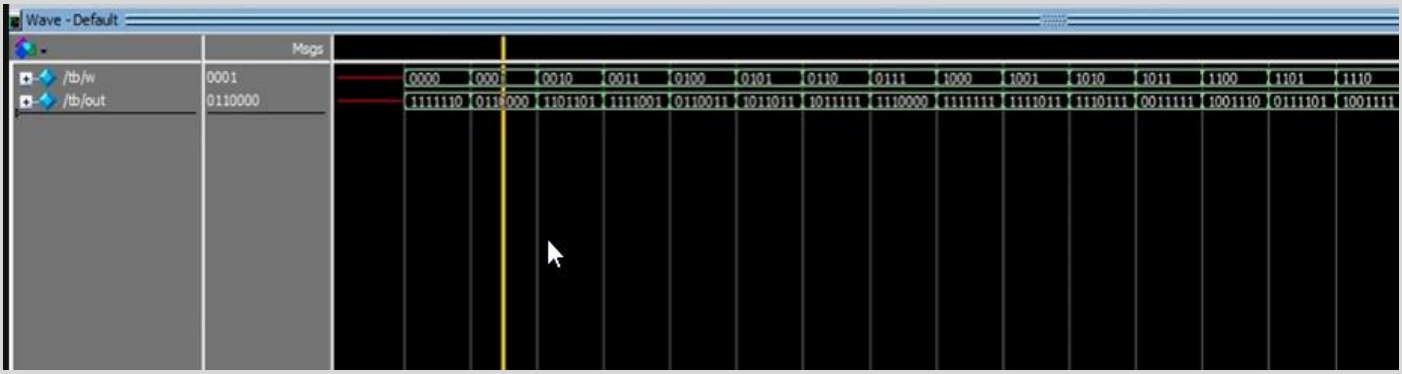
12- **Mux41**: The multiplexer has four inputs (a0, a1, a2, a3) and one output (y). It also has two select lines (s0 and s1) that determine input should be routed to the output.



13- **Full Adder**: The full adder has three inputs: two binary digits (a and b) and a carry-in bit (cin). It has two outputs: a sum (sum) and a carry-out bit (cout).



14- **7-Segment-Decoder**: decoder that converts a binary-coded decimal (BCD) input into a 7-segment display output



15- **Add-2-Minutes**: The "add2" module performs an 8-bit addition of the "in" input vector and a constant value. The constant value is generated by using a series of AND and OR gates to set the carry input for the first full adder (i0) to 1, and all the other carry inputs to 0.

The full adders (i0 to i7 and o0 to o7) are connected in a ripple carry configuration, where the carry output of each full adder is passed as the carry input to the next full adder. This allows the addition to be performed across all 8 bits. I0 to I7 are responsible for the actual addition where o0 to o7 are responsible for dealing with converting the outputs to bcd, this is done by a set of gates that output a flag which add “6” to the value if its invalid in terms of bcd values (ie 10, 11, 12.. etc). This same flag logic is implemented in the subtract 2 minutes. The first adder get an initial carry of 0.

The inputs and output digits are disorganized due to technical reasons so it is unreadable/unusable independently therefore we cannot provide a screenshot of it working as a standalone module and we kindly request you see it working in the simulations of the counter

16- **Subtract-2-Minutes**: The "sub2" module performs an 8-bit subtraction of the "in" input vector and a constant value and returns the result in the "out" output vector.

The constant value is generated in the same way as in the "add2" module, by using a series of AND and OR gates to set the carry inputs for the full adders.

The full adders (i0 to i7 and o0 to o7) are still connected in a ripple carry configuration, but the constant value being subtracted is different from the one in the "add2" module. This results in a different carry input sequence for the full adders. The inputs are added to the 9’s compliment of 2 to be able to subtract rather than add. This concept was found by research and trial and error. The first adder gets an initial carry of 1.

The inputs and output digits are disorganized due to technical reasons so it is unreadable/unusable independently therefore we cannot provide a screenshot of it working as a standalone module and we kindly request you see it working in the simulations of the counter

17- **Clock divider:.**

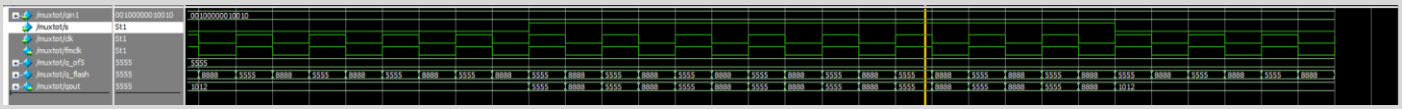
The "clk_dvdr" module is a simple clock divider circuit that takes an input clock signal and divides it by a factor to generate a slower clock signal. The input clock signal and the desired division factor are specified as inputs to the module, and the divided clock signal is the output. This module contains a behavioral case statement. This statement has a table of what speed it should be on depending on what mode we are on. So, if the current “speed” is default it outputs the default clock of 2hz, case of 2x, outputs 4hz, case of 1/2x, outputs 1hz, and finally, if both speed up and speed down are on, it outputs a slower clock of 1hz so it can be used to help with the flashing of the error code. There will be no simulation of this seen as it’s a standard clockdivider and modelsim is not capable of simulating a 5Mhz clock.

18- **Mux_Total**:
This module represents a circuit that takes four inputs: a 14-bit input (qin1), a selection signal (s), and a clock signal (CLK), and produces a 16-bit output (qout). The module contains a 16-bit register (q_flash) and

a 16:1 multiplexer (mux21) for each bit of the output. This module partially is responsible for flashing the value of the error code (there is more implementation for the error code in BCD_4in1 due to the way Common Anode displays work.

The function of the module is to select either the input vector or the contents of the q_flash register and route it to the output vector, based on the value of the selection signal (s). When s is a logic 0, the default counter output is selected and passed through to the output. When s is a logic 1, the contents of the q_flash register ANDed with the clock (to simulate flashing) are selected and passed through to the output.

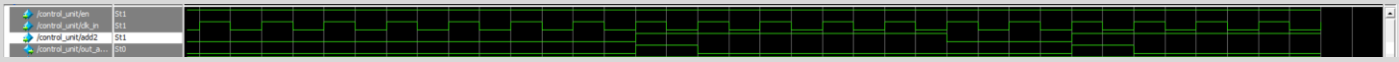
The q_flash register is loaded with the constant 16-bit value "0101010101010101" is updated on each rising edge of the clock signal. The multiplexers are used to route the appropriate input to each bit of the output. The output is made up of a combination of the input and the contents of the q_flash register. (the 8888 output represents all the seven segment display digit turned off)



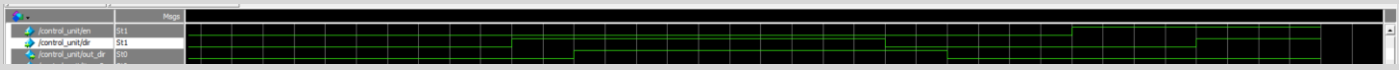
20- **Control Unit**:

This module is responsible for processing all the inputs coming into the stopwatch. It takes in (user_enable, direction, speed_times2, speed_div2, addsub2 ,clk_in,//comes from fpga//) and has outputs (out_clk//goes to components, out_dir, out_add2, out_en, flshmode). This module has many uses which will be listed and explained below:

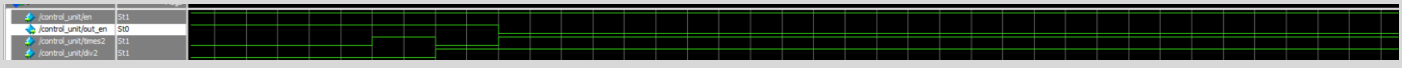
1; The add 2 button is supposed to be a push button where you can push it once and it adds 2, however, the control unit adds a restriction where you cannot simply keep holding down +2 and having it go till it reaches maximum value. This feature is useful for testing however it is not logical for the stopwatch design we have and it is required to be a singular click not a hold down. This is done by passing the +2 input of the user into an edge detector that updates almost instantly, gives out the signal 1, then turns back to 0. This edge detector is made using a single DFF and an AND gate. This uses the universal design for the edge detectors.



2; As per the project requirement, the user should not be allowed to switch the direction of counting while the stopwatch is unpaused, therefore, the control unit takes the enable and add2 signals from the user, manipulates them, and only changes the direction if the stopwatch is paused. This is done by taking the pause signal, getting it's compliment, and using it's compliment as the "Enable" signal for a DFF which stores the old output until the stopwatch is paused then starts accepting the new value of it's D aka the new direction. We are aware this simply simulates a latch however there were very weird bugs with modelsim when trying to simulate a latch so we went with this path instead.

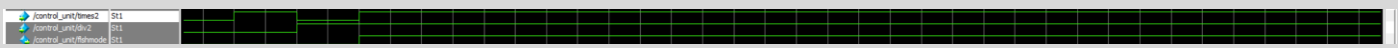


3;When speed up and slow down are both set to 1, the counter should be paused at it's current value and the flashing of the error code stops. This module also takes in the enable signal as well as the input of both speed up and slow down and manipulates the output signal which is Out_En to be 1 only if enable is 1 and both speeds aren't one. This is done but NANDing the signals of speed up and slow down and ANDing that signal with the signal of the enable and outputting that.



4;This module is what calls the module for the clock divider and outputs the new clock. It also sends it the speed. How the clock divider is made is described earlier in the document.

5;This module outputs the flag "flashMode" which notifies the according modules when they should be flashing an error code instead of counting. In our case, this simply ANDs both speed up and slow down.



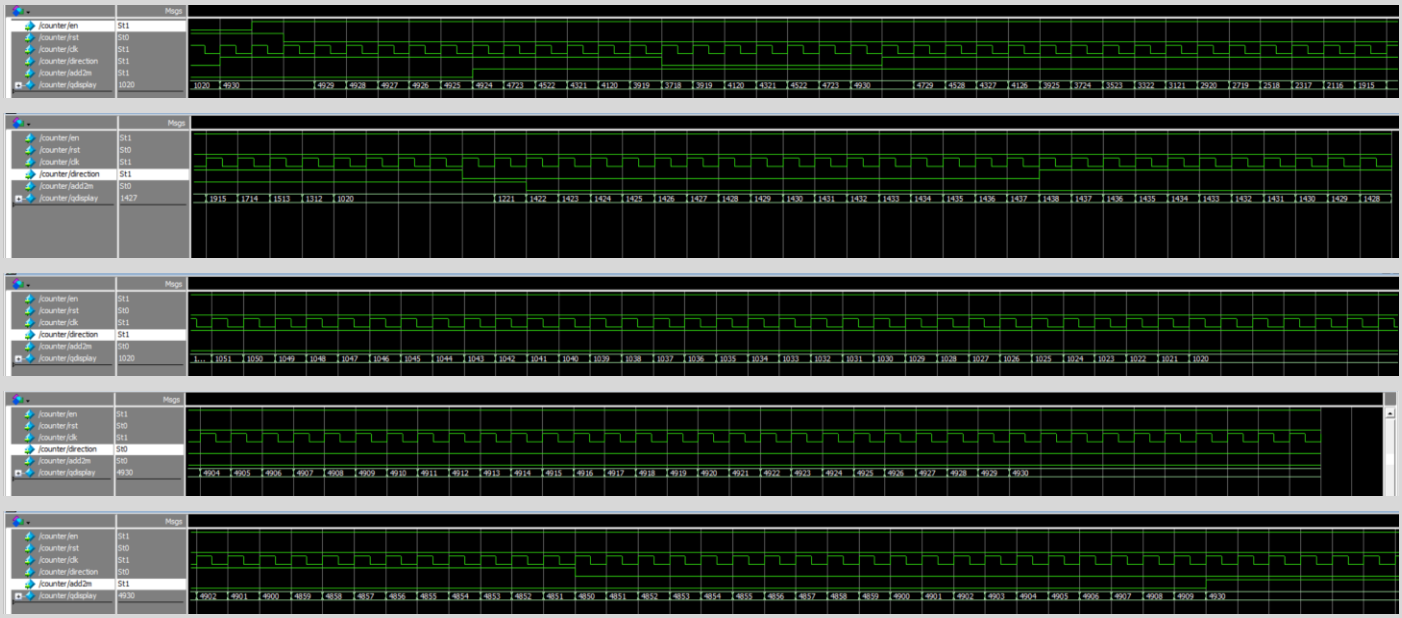
21- **Counter**

This is the core of our design and it's what is shown in the above schematic. This module does all the counting and is independent of any other blocks. It can count up to 49:30 then stops, can count down to 10:20 and stops, can add or subtract 2 given a signal and given what counting mode it's in, can reset when being given a reset signal. It takes in the inputs (enable, rst, clk, direction, addsub2) and it can output a 14 bit signal q (signal can be converted to 16 bit to be proper bcd) by grounding 2 digits which will never reach the value 1 anyways. For the testing, there will be a wire that represents the output value but in 16 bits. this will be to simplify the reading of the values and assure it's working properly by utilizing the Hexadecimal radix in modelsim however this does not affect the output in any way and will be commented out on submission (will not be deleted to ease testing for TAs).

In the simulations, you see that the value reaches maximum with add2 1 clock-cycle before it should, this is due to add2 being forced constantly to 1 while the logic it's made around is centered around treats it as a singular button push, so this issue will not be seen in real world application

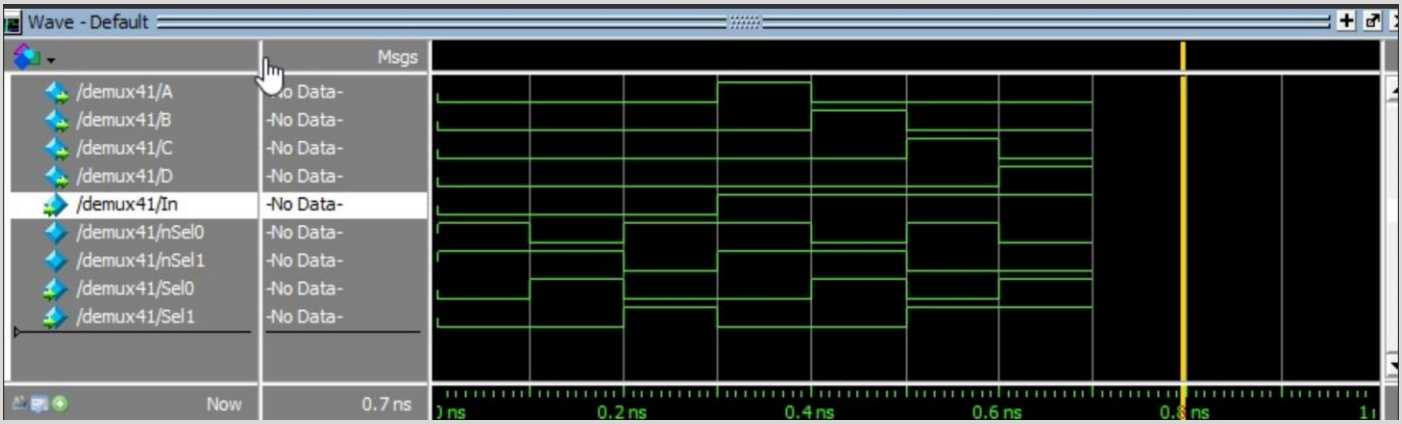
We tried to include as many edge cases as possible in the simulation however there will obviously be cases we did not include in the screenshots. However, we are certain there are no edge cases we are not dealing with correctly.

How it exactly is connected or functions is too complicated to explain over text however it is shown in the schematic above (although the schematic is not competed as some changes were made as code without drawing such as subtract 2, reset and stop at sub2 of, etc)



22- Demux41:

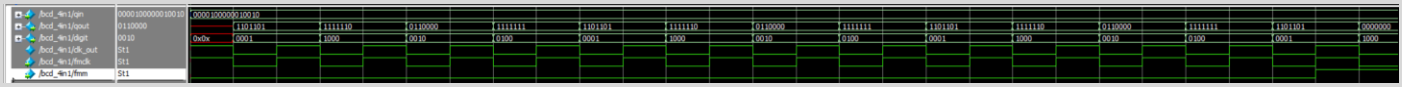
A 4-to-1 demux that routes an input signal to one of four outputs based on two selection signals. Its made by a bunch of combinational gates (ANDs and NOTs)



23- **Mini_Clock_Divider**:
A clock divider that takes an input clock signal clk_in and generates a divided clock signal c_out. The clock divider works by counting the rising edges of the input clock signal and toggling the output signal c_out whenever the counter reaches a certain value. This value in this case forces the clock output to be 400hz (This is the clock used to cycle through all 4 bcd decoded values and digits to be able to use the 4 digit bcd display provided to us). Note: the value of 400hz can be finetuned later at will as it's set depending on conditions of the external circuit which may be unstable.

24- **BCD_4in1**
This module decodes outputs for a 4-digit 7-segment display. It takes the input "output" as an input signal ([16:0]qin) and an outputs signal ([7:0]qout) as well as ([2:0]digit). (qin) is a 16-bit binary number that represents the digits to be displayed, and (qout) is a 7-bit signal that drives a 7-segment display. And display represents which "display" is currently active. It takes it's clock straight from the FPGA as it's division is done by "miniclkdiv" due to it needing a much faster frequency than any other component to be able to simulate continuity in all 4 displays and avoid any flicker. This module cycles through each digit in the input q such that it outputs the output of 4 of them in sequence, however due to the very fast refresh rate it will look like it's statically outputting every digit. It also has a 4 input demux which uses the same as the output muxes to be able to identify which digit belongs to which of the decoded bits, simply, it is the 1 hot encoded value of every bit. The counter which cycles through all selectors/digits consists of 2 flipflops and it unfortunately requires an "initalize 0 block" to function properly. It also takes in the (flashmode clock and flashmode flag) as inputs therefor when it should be flashing, it completely shuts down all leds or outputs a 5, this is done with simple muxes and their selector being the flashmode clock AND flashmode flag (if flag = 0 it just outputs the normal expected output) {mux21 m0(tqout[0],1'b0,fm,qout[0]);}

The flashing is near impossible to test in modelsim however it and other features will be shown in out attached video.



OUR ATTACHED VIDEO FOR PROJECT TESTING ON THE DATE 9/01/2023:
https://youtu.be/kGOckyLHlZM

- Work Distribution:

Module/Component	**Done by**
Writing Report (phase one)	Adham Saad Aser Osama Mohamed Morshedy
Clock Divider	Mohamed Magdy Aser Osama
Counters	Aser Osama Mohamed Morshedy Mohamed Magdy
3-input comparator	Adham Saad
4-input comparator	Adham Saad
D flip flop	Adham Saad
Demux	Mohamed Magdy
Comparator max	Mohamed Morshedy
Comparator min	Mohamed Morshedy
Of min	Aser Osama
Of max	Aser Osama
State Diagram	Mohamed Magdy
Mux21	Mohamed Magdy
E-Not	Aser Osama
Mux41	Mohamed Magdy
Muxtot	Mohamed Morshedy Aser Osama

Count_Mode	Mohamed Morshedy
Full Adder	Adham Saad
Add 2	Adham Saad Mohamed Morshedy Aser Osama Mohamed Magdy
Subtract 2	Adham Saad Mohamed Morshedy Aser Osama Mohamed Magdy
BCD Decoder	Adham Saad
Control Unit	Aser Osama
Mux21 (mymux)	Aser Osama
Writing Report (Phase Two)	Adham Saad Mohamed Morshedy Aser Osama Mohamed Magdy
Counter	Aser Osama
Demux41	Mohamed Morshedy
Miniclkdiv	Mohamed Magdy
BCD_4in1	Aser Osama

References:

1- R. Muley, B. Patil, and R. Henry, "Design and implementation of digital clock with stopwatch on FPGA," 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), 2017, pp. 1033-1036, DOI: 10.1109/ICCONS.2017.8250622.

2- Mano, M.M. (2008) Computer System Architecture. New Delhi: Prentice-Hall of India.

3- Harris, S.L. and Harris, D.M. (2022) Digital Design and computer architecture. Amsterdam etc.: Elsevier.