AUC The American University in Cairo

Project 2

Digital Alarm Clock on the BASYS3 FPGA Board

CSCE 2301 - Digital Design I

Dr. Mohamed Shalan

Basmala Abdelkader

MagdElDin AbdalRaaof

Mohamed Elshemy

Due Date: December 17th 2022

Submission Date: December 17th 2022

Block Diagram Design (contributed by all three members):

The design of the alarm clock was a joint effort between all three of us. We initially had the 7 segment display be controlled by a decoder and counter that work together to display the output in such a way that it would feel as if all displays were lit up simultaneously. Scanning to determine the output was initially done twice, once at the end of each module, before the mode determined whether the output of adjust or the clock would be displayed. Additionally, we used an XOR gate and a D-flip flop for the implementation of the circuit to determine the mode of the alarm clock. Furthermore, we processed the inputs of the buttons using debouncers, synchronizers, and edge detectors. The input processing circuit was used for every button input.

In adjust mode, the modification of numbers was initially done using accumulators such that what value got loaded to be added to the accumulator was dependent on whether the up or down buttons (left or right in the case of the selector portion) were pressed. This module used comparators to identify which portion was selected at the moment. The module operates on a 100 Hz clock that was made possible using a clock divider that outputs a 100 Hz clock signal.

The clock portion was designed much like how it was designed in class using counters with differing enable signals to count seconds, minutes, and hours. This circuit operates on a 1 Hz clock that is a result of using a clock divider that outputs a 1 Hz clock. The scanning done at the end of both the adjust and the clock modules were managed by the same 2 bit counter. The alarm was supposed to be activated via the output of a 4 bit comparator.

This initial iteration of the design was a struggle to create, as it was our first time designing a system of this size. Additionally, initial attempts at integrating finite state machines made matters more complicated. This initial design was missing requirements such as the ability to set the time of the clock and the decimal point blinking. Other features were implemented in a sub-optimal manner such as using accumulators as opposed to counters for the adjust module, and using an XOR gate and a D-flip flop as opposed to using a T-flip flop.

There was a later iteration of the design that implemented the missing features and streamlined the implementation of already existing features. In adjust, counters were now utilized and a 2x4 binary decoder is used to allow selection of the individual portion to modify. These from top to bottom were the adjust of the alarm hours, alarm minutes, clock hours, and clock minutes. Clearly, this order is swapped from the specified order in the project requirements, but it is fixed in later iterations. The times for the adjusted hours, ten hours, minutes, and ten minutes of both the clock and the alarm are saved in registers for later use.

The enables of the counters in the clock module were modified to account for the fact that resets occur when ten hours is 2 and hours is 3 and to account for the fact that values can be loaded the moment the circuit enters clock mode. Moreover, the data to be loaded was either 4 bit binary 0 or the value saved from the adjusted clock time. Which of the two was loaded depended on if the center button was pressed to change mode.

For the circuit responsible for taking control of the alarm, it is only enabled when the alarm time matches the time on the running clock (checked using 4 separate 4-bit comparators) and the mode is set to clock, this enabled output is then passed to a T-flip flop to create the blinking effect on LD0 and the buzzing sound from the buzzer. In the overall circuit, the screening is now done using two selection lines (mode and the output of the 2 bit counter connected to a high clock speed). Adjust takes in the 100 Hz clock and the button inputs after they are processed. Meanwhile, the clock module takes in the processed button input of BTNC and the times from the adjusted clock. LD0 is activated when the mode is set to adjust or the alarm signal is on, but not when both are on at once.

The diagram did get changed again as a consequence of going through the code we used to implement the project on the FPGA board then realizing it was not quite the same. Changes include the use of only one modulo 24 counter when adjusting alarm hours or clock hours, then utilizing an LUT-like structure to assign the correct values to the alarm hours and alarm ten hours values. Furthermore, the clock module now takes in mode as well to be able to properly identify when to load in the values into the clock. There was previously a separate processing circuit dedicated to making BTNC work on 1 Hz for the clock circuit, but that was a failure that proceeded to be scrapped later on. Reset signals were also added to almost every module in the circuit.

Term Legend in the block diagram:

ATH: Alarm Ten Hours

AH: Alarm Hours

ATM: Alarm Ten Minutes

AM: Alarm Minutes

AJTH: Adjust Clock Ten Hours

AJH: Adjust Clock Hours

AJTM: Adjust Clock Ten Minutes

AJM: Adjust Clock Minutes

CTH: Clock Ten Hours

CH: Clock Hours

CTM: Clock Ten Minutes

CM: Clock Minutes

LDA: the alarm signal

Sel: the selection line from the 2 bit counter to screen the value inserted into the decoder

BTNL: Button Left

BTNR: Button Right

BTNU: Button Up

BTND: Button Down

BTNC: Button Center

INL: Processed input of BTNL

INR: Processed input of BTNR

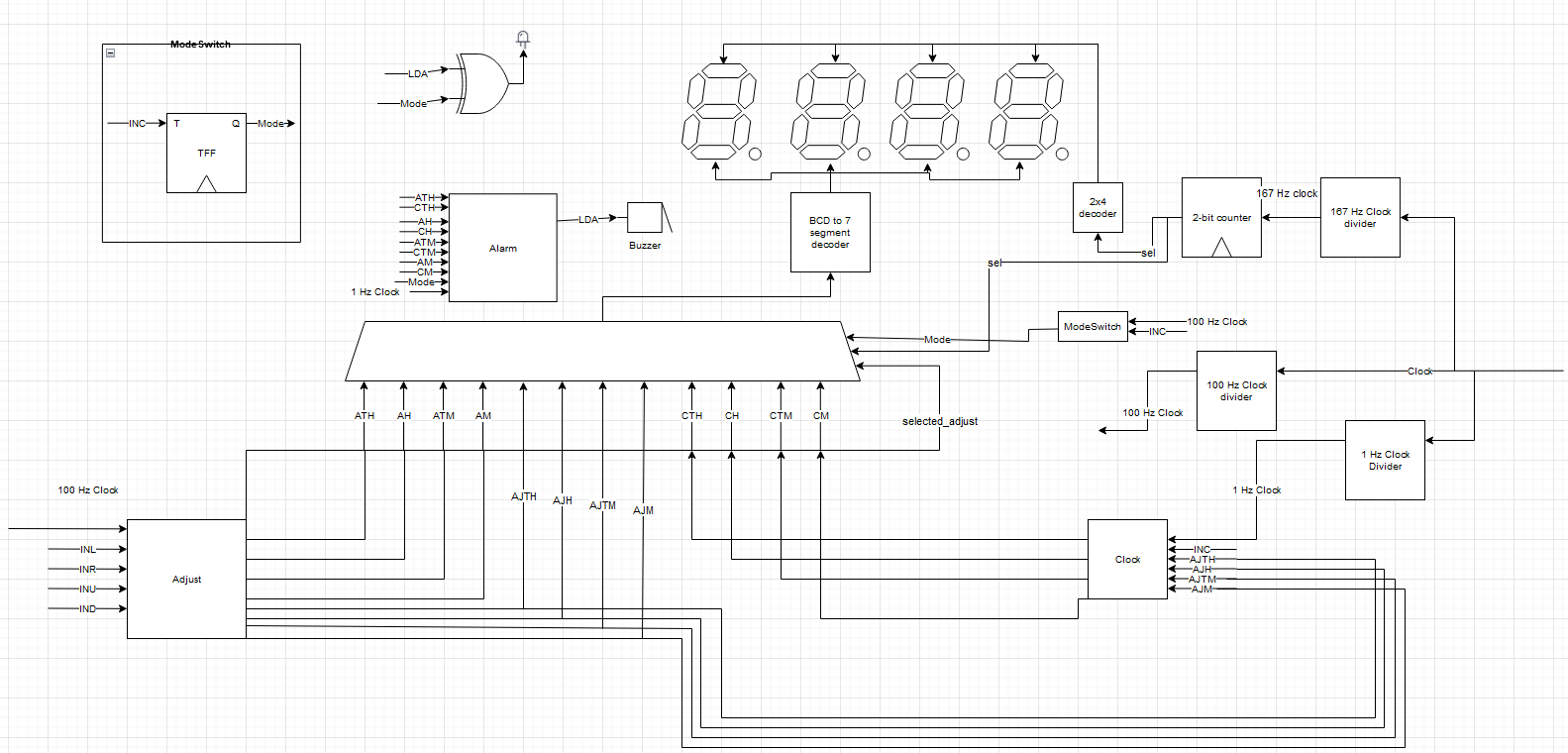
INU: Processed input of BTNU

IND: Processed input of BTND

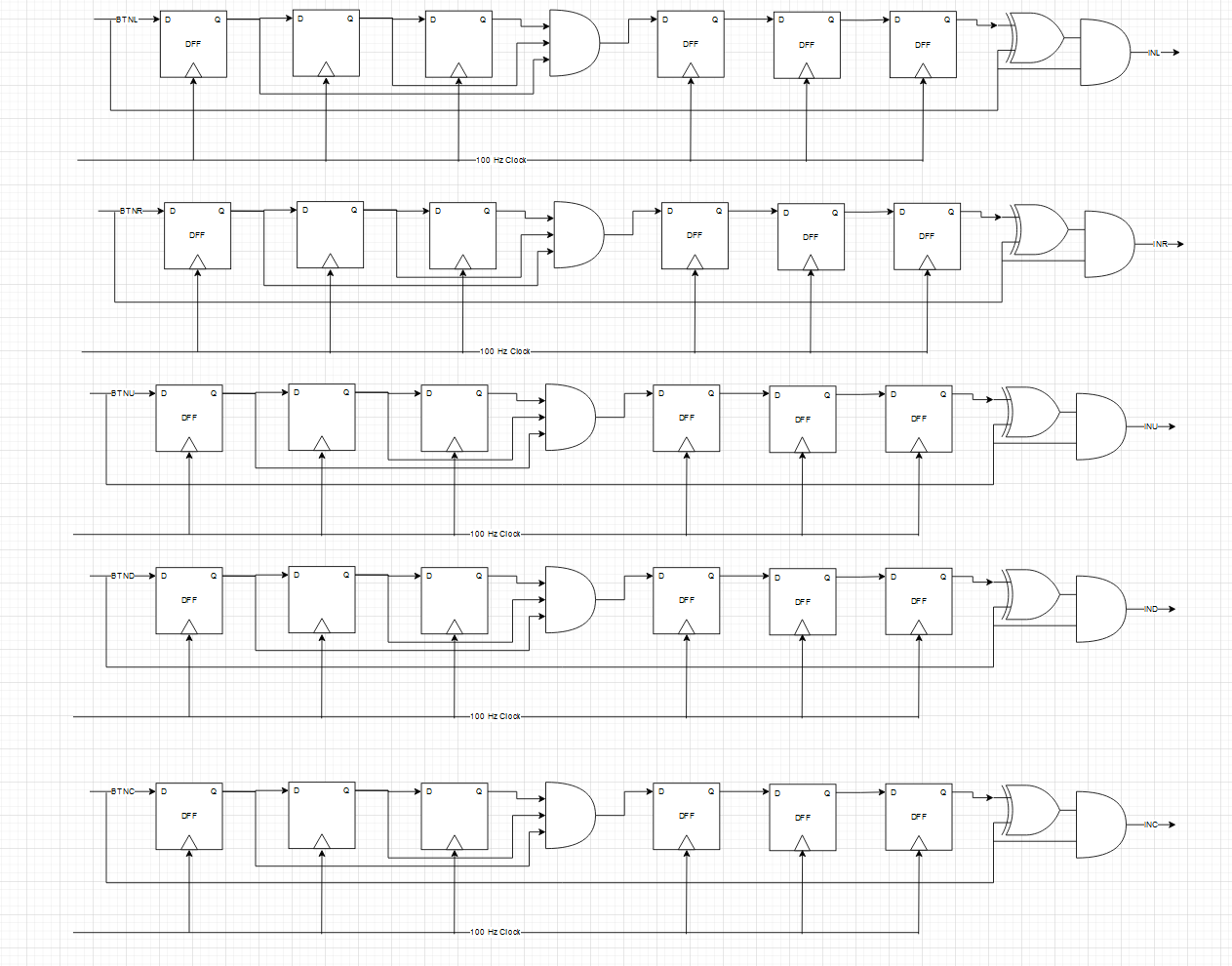
INC: Processed input of BTNC

Pictures of the block diagram:

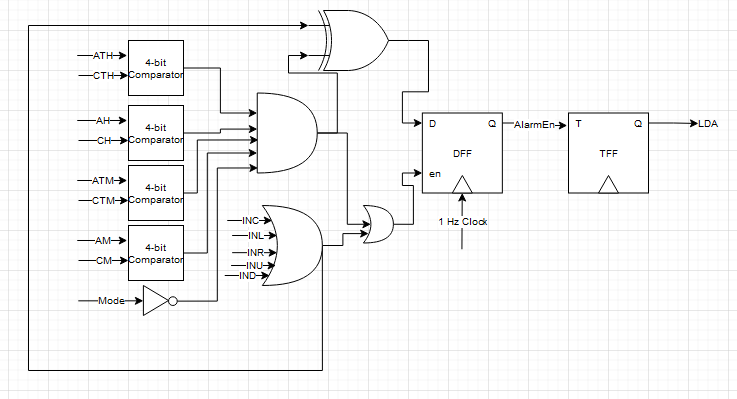
Main Circuit:



Picture of the button adjustment block diagrams:

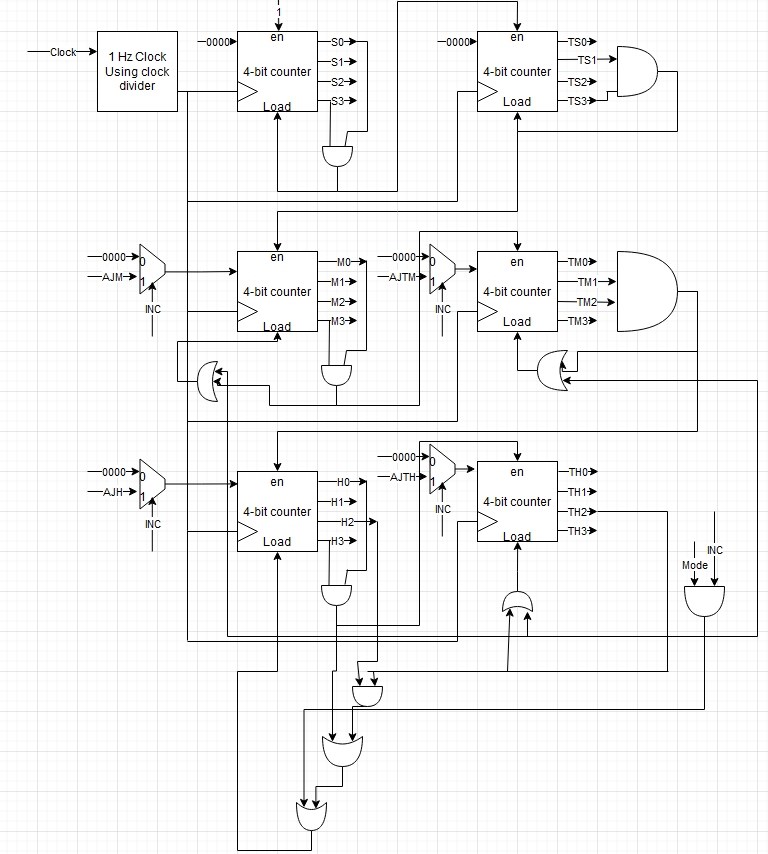


Picture of the alarm circuit:



Picture of the Adjust circuit would have been provided, but draw.io is not being cooperative and is the images are all disfigured, one would have to go to the website and use the GitHub repo to properly see it.

Picture of the Clock circuit:



The Adjust Module Code (Contributed by MagdElDin AbdalRaaof)

To explain this part, we first need to explain the implementation of an x-bit counter module that we created. The module has 2 parameters, x for the number of bits and n to indicate the maximum number the counter is allowed to reach. Its inputs are a clock signal (clk), a reset signal (rst), an enable signal (en), and a signal that indicates if it should count up or count down (UpDown). It outputs the n-bit number in the counter (out). Upon each rising edge of either the clock or the reset, the counter first checks if the reset signal is enabled. If it is, then the counter outputs an x-bit 0. Otherwise, it checks if the counter is enabled. If the counter is enabled, it proceeds to check if the counter should count up or down using the UpDown signal. If UpDown is 1, then the counter should count down, but if the counter is at 0, then the counter loops to the maximum possible value instead. If UpDown is 0, then the counter should count up, but if the counter is at (n-1), the counter loops back to 0. It is assumed that if none of these conditions are met that the counter maintains its old value.

The adjust module itself takes in the clock signal (clk), reset signal (rst), enable signal (en), processed signals of the button input from the left, right, up, and down buttons respectively (INL, INR, INU, IND). It then outputs the values of the alarm hours (tens (ATH) and units (AH)), alarm minutes (tens (ATM) and units (AM)), clock hours (tens (AJTH) and units (AJH)), clock minutes (tens (AJTM) and units (AJM)), and the select line that chooses between which one of the aforementioned outputs is changed (selected\_adjust). It also outputs the data necessary to turn on the LEDs that indicate which part is being adjusted right now (LD12, LD13, LD14, and LD15). Inside the module, there are wires such as enable\_alarm\_hours that activates when the adjust module is enabled, and selected\_adjust is 2’b10, enable\_alarm\_minutes that activates when the module is enabled and selected\_adjust is 2’b11, enable\_clock\_hours that enables when module is enabled and selected\_adjust is 2’b00, and enable\_clock\_minutes that enables when module is enabled and selected\_adjust is 2’b01. ATH, AH, AJTH, and AJH have all been declared as registers for a reason that will be explained later. Our first counter appears and that is a 2-bit, modulo 4 counter that outputs selected\_adjust to choose which of the next counters to activate. Then we have a 5-bit, modulo 24 counter that outputs a 5-bit number that represents the current hour that the alarm is set to. Its output is AH\_temp, and it is used in a case statement to select the value of ATH and AH. The same applies for modifying the clock hours using AJH\_temp. For modifying alarm minutes, two counters are used, both of which are 4-bit, but the first one is modulo 10 and the second one is modulo 6. The modulo 6 counter only activates if the first one is active and AM is 0 and UpDown is 1, or when the first one is active and AM is 9 and UpDown is 0. A similar system is used for the adjusting of clock minutes.

Difficulties faced during the creation of this module included the module initially only showing binary numbers as opposed to denary due to faulty connection with the main decoder. This was fixed by explicitly declaring wires and registers. Additionally, the 24 hour counter was a particularly challenging aspect. By trying to implement it in the same way as the minute counters, the counter could go all the way up to 29. Attempts to remedy this using separate counters only made matters worse as numbers would shift around, seemingly at random because the enable conditions for the counters would be fulfilled unwittingly. Additionally, the adjust portion was swapped, where the alarm adjustment took the first 2 LEDs and the clock adjustment took the last 2 LEDs. This was eventually fixed.

The Clock Module Code / FSM diagrams (Contributed by Basmala Abdelkader)

This module, called *Clock\_Circuit*, was done by integrating another module and using it inside this module. The module we used was *Counter4bit*. *Counter4bit* is a 4 bit up counter that takes the following inputs and outputs:

**Inputs:** Data of 4 bits, and a clock, reset, enable, load\_zero, and load\_value (will be explained).

**Outputs:** Count which is 4 bits, and declared as a register (used in always block).

For the Inputs, the Data is the number transferred to the counter and the starting point of the count, and the load\_value input allows the data to be transferred to the counter depending on whether it is high or low. The reset and load\_zero inputs set the counter back to 0. The enable allows the counter to start counting if it’s high, and if low it stops the counter at its last value.

The output is a single 4 bit variable named count which changes depending on the input as explained above.

In the *Clock\_Circuit* module, we have included six such counters, each for a section of the clock. The first two counters are for counting seconds. The first counter, named c0, counts the first digit, where the data input is 4 bit 0, clock, reset, the enable set to 1, load\_zero is high when seconds reach 9, load\_value is set to 0, and the output is saved in the wire S. The second counter, named c1, counts the tens digit of seconds, where all the inputs of the first counter are the same in the second except for the enable which becomes high when S reaches 9, load\_zero becomes high when the variable TS (representing the tens digit of seconds) reaches 10, load\_value is set to 0, and the output is saved in the TS wire.

The next two counters are the minutes counters. The source of their data input is the Adjust mode to set the values to specific starting points if needed, and the data inputs are called AJM and AJTM, for Adjust Minutes and Adjust Tens Minutes (each digit). The first counter, called c2, also takes the inputs clock, reset, the enable value being high or low depending on whether TS has reached 10, the load\_zero being high or low depending on whether CM (counter for minutes digit) has reached 10, the load\_value being high or low depending on whether both INC2 and Mode are high or not, and the output CM as a wire. The second counter, named c3, has the data input AJTM, clock, reset, the enable becoming high only when CM reaches 10, the load\_zero becoming high if CTM reaches 6, the load\_value becoming high if INC2 and Mode are both high, and the output being the wire CTM.

The last two counters are responsible for counting the hours. The first counter, named c4, counts the first digit, taking in the data input AJH for adjust hour, clock, reset, enable set to high only if CTM reaches 6, load\_zero set to high only if one of the following conditions are met: CH reaches 4 and CTH reaches 2, or CH reaches 10, the load\_value set high only if both INC2 and Mode are high, and the output wire CH. The second and last counter, called c5, takes the data input AJTH for adjust tens hour, clock, reset, enable set to high if CH reaches 10, load\_value set to high if CTH reaches 2 and CH is 4, the load\_value set to high if INC2 and Mode are both high, and the output wire CTH. This concludes the module *Clock\_Ciruit*.

Difficulties regarding this module included the clock not having a proper reset system in place, and it failing to adjust when exiting adjust mode. That second error was especially difficult and was fixed by making the clock detect the rising edge of the processed center button input, which was processed at 100 Hz even though the clock circuit operates with 1 Hz.

Alarm Module (Contribution by Mohammed ElShemy)

Initially, we tried creating this part using structural logic, but this proved too much of a hassle to do. Eventually, we had it so that the code for detecting when the alarm should be enabled should simply be a part of the finite state machine in the main circuit while the alarm circuit is simply a signal being alternated depending on whether or not the alarm is active. Its inputs are the 100 Hz clock (clk), a reset signal (rst), an enable signal (en), and it output the signal that acts as the alarm (LDA). If rst was 1 or en was 0 then LDA would be 0, otherwise, the value of LDA would alternate each clock cycle.

Main Circuit (Contribution by all members):

Variable legend (anything not written here was referenced back in the block diagram legend with the same meaning):

I/O signals:

rst: Reset signal

clk: Clock signal

seg: 7-bit binary number to indicate the lights needed to be turned on in the 7 segment display (active low)

LD0: signal for LD0 to indicate alarm going off and adjust mode

LD12: signal for LD12 to indicate adjusting of alarm minutes

LD13: signal for LD13 to indicate adjusting of alarm hours

LD14: signal for LD14 to indicate adjusting of clock minutes

LD15: signal for LD15 to indicate adjusting of clock hours

DP: signal for the decimal point to light up (active low)

alarm\_signal: signal used to activate the buzzer and LD0

anodes: a 4-bit binary number to indicate which one of the anodes is active

Internal wires and registers:

selected\_adjust: a 2-bit binary number to indicate which time / part of time the user is adjsuting at the moment.

not\_turned\_on: a signal to indicate if the alarm was already turned on and then diabled

DP\_count: a number to allow for turning the decimal point on and off at a rate of 1 Hz

Num\_out: the number input to the BCD to 7 segment decoder.

state: current state

NextState: Next state

One\_Hundred\_Hz: a signal that acts as a clock with a frequency of 100 Hz

One\_Hz: a signal that acts as a clock with a frequency of 1 Hz

select\_out: a number to indicate which one of the four 7 segment displays is activated

Modules:

PushButton: the module that processes the button presses

Adjust: the module that allows for adjusting the clock time and the alarm time

Clock\_Circuit: the module that drives the clock

alarm: the module that drives the alarm signal

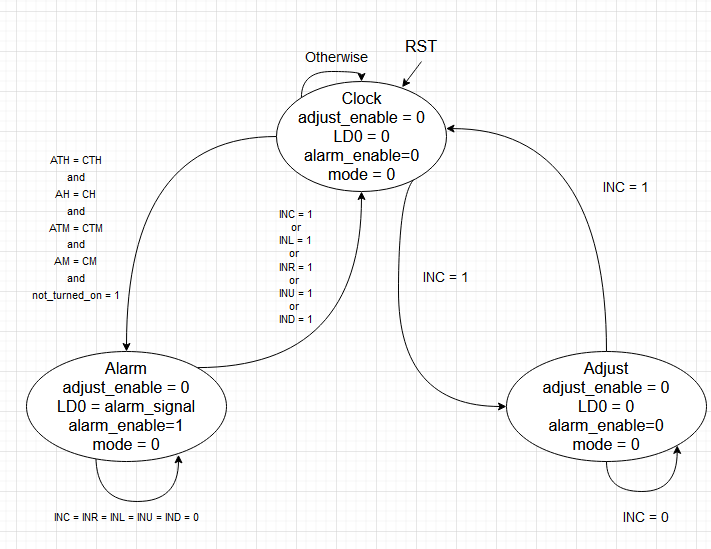
ClockDivider: the clock divider module

SelectCounter: the module that controls what anode to activate and what number to be displayed on the 7 segment display

decoder4x7p: a BCD to 7 segment decoder

X\_Bit\_Counter: a counter module that can count up and down

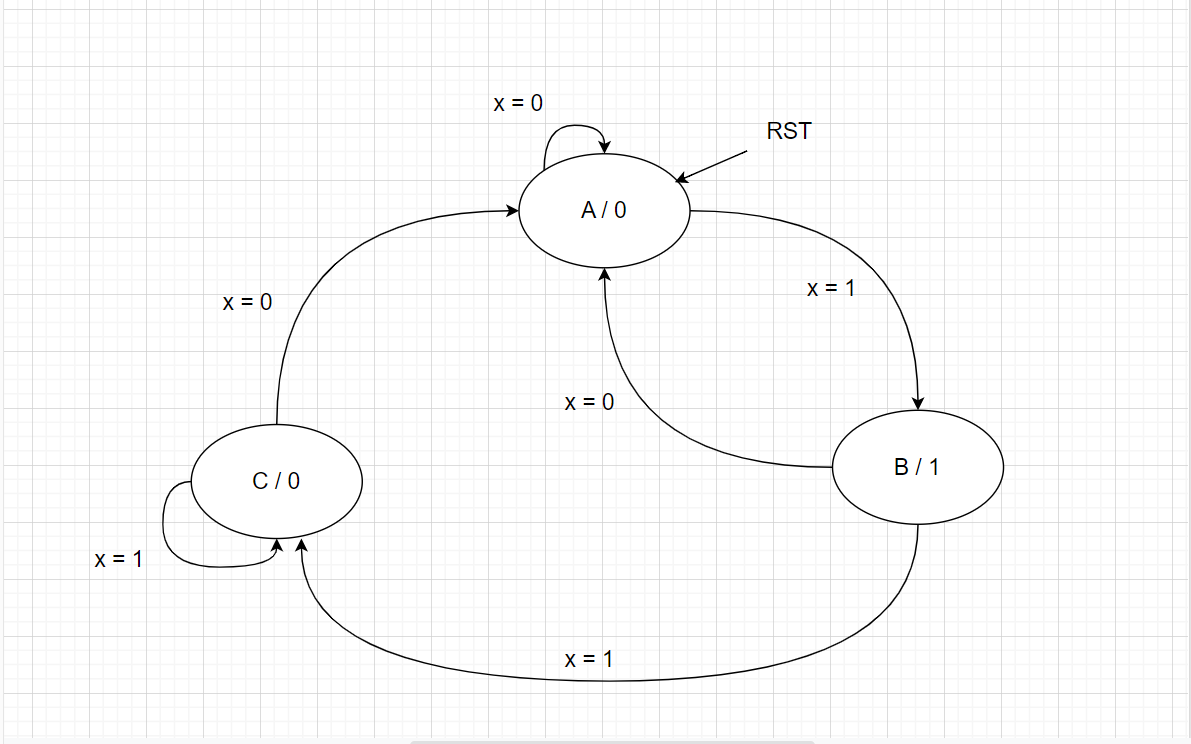
We have used three finite state machines in the project: the Main Circuit FSM (Moore Machine), the Rising Edge Detector FSM (Moore Machine), the Adjusting Circuit FSM, and the Hours Counter FSM. Below is a screenshot of the Main Circuit FSM.



There are 3 states in this FSM, and they are named Clock, ADJUST, and ALARM. The default state is the Clock, where if none of the other specified conditions are met, the next state is the Clock (otherwise), and the outputs are adjust\_enable which is set to 0, LD0 set to 0, and alarm\_enable set to 0. The Clock is also the next state if the reset is set to high. However, if the input INC is set to 1, the next state is the Adjust mode. So the output of the Adjust state is the adjust\_enable set to high. If INC is then set to 1, the next state becomes clock. If INC is 0, then the next state remains Adjust.

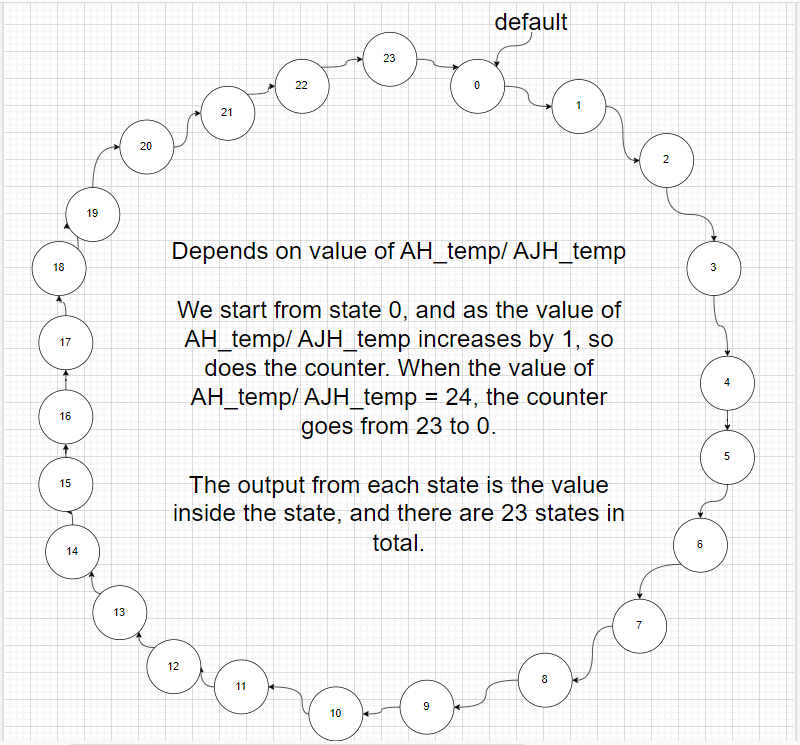
If the current state is Clock, the next state becomes Alarm if the conditions mentioned above are met (ATH = CTH and AH = CH … not\_turned\_on = 1). The outputs in the Alarm state are adjust\_enable set to 0, LD0 set to the alarm\_signal, and alarm\_enable set to 1. The next state then becomes Clock if the conditions above are met (INC = 1 or INL = 1… IND = 1). Otherwise, the current state remains Alarm. The code for the module functions effectively the same with more details present in the code, such as the 2nd decimal point being assigned a value of 0 (the decimal point is active low) whenever a special counter called DP\_counter reaches, select is set to 2’b10, and the circuit is in clock mode. Additionally, not\_turned\_on is set to 0 whenever the alarm state is exited.

Below is a screenshot of the Rising Edge Detector FSM.



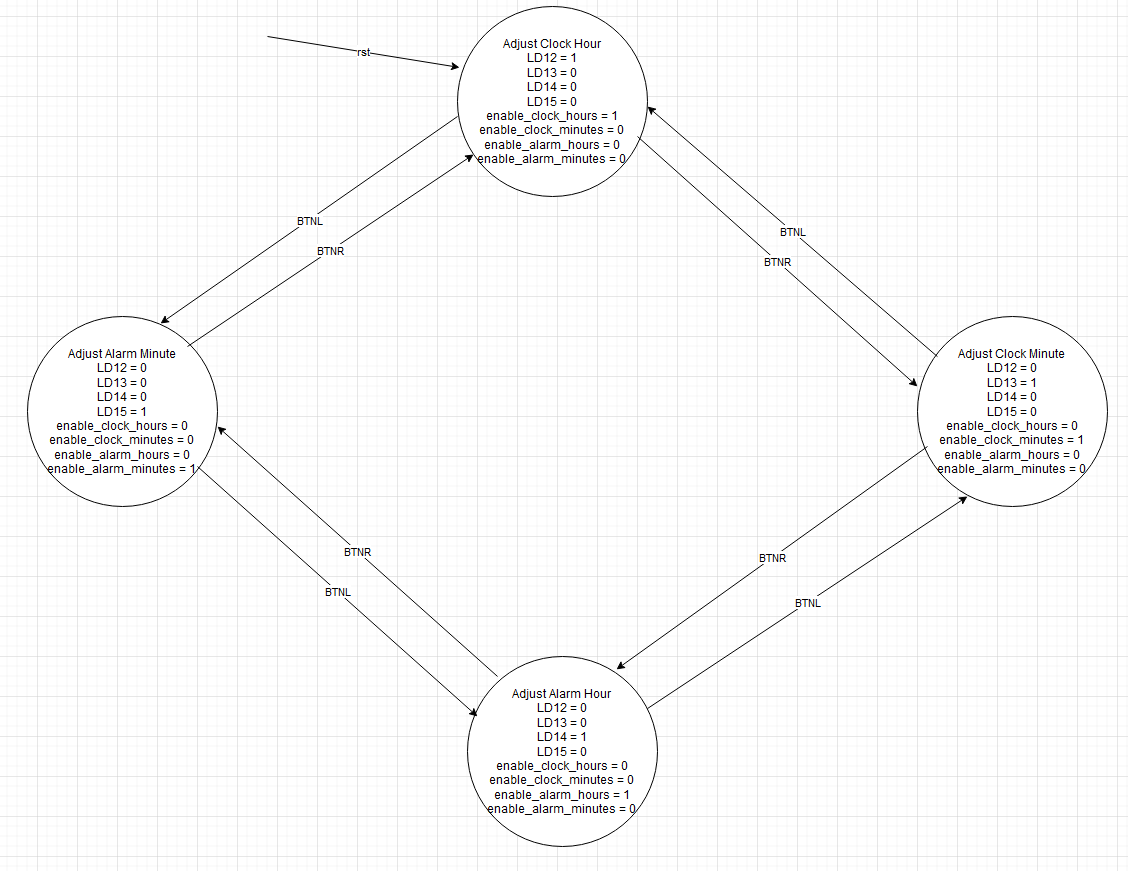
For this FSM, there are 3 states named A, B and C. The default state is A. When reset is set to 1 the current state is set to A, and the output is 0. The next state is B if the input x is set to 1 and the current state is A, otherwise (if x = 0) then the next state remains A. The output is 1 whenever the current state is B. If x = 1 and the current state is B, then then the next state is C, but if x = 0 then the next state is A. The output is 0 when the current state is C. If x is set to 1, the next state remains C, but if x is set to 0 the next state is A.

The hours counter is necessary for the Adjust module to function. It is not a separate counter, but can be considered a finite state machine as will be demonstrated. Below is a screenshot of the Hours Counter FSM.



The output would have been difficult to write for each state, but this is explained in the diagram. Since this FSM is for both AH\_temp and AJH\_temp, each state depends on the current value of the AH\_temp or AJH\_temp, depending on the FSM we are using. The significance of AJ\_temp and AJH\_temp are explained in the code. The default state is 0, where the FSM starts, so there is no reset, but the default state is set to state 0. The special condition here is that if the value of AJ\_temp or AJH\_temp is 23, it returns to 0. This concludes the FSM section.

The adjust circuit’s FSM goes as follows:

There are four states, AdjustClockHour, AdjustClockMinute, AdjustAlarmHour, and AdjustAlarmMinute. The user cycles between these 4 states by pressing the left and right buttons while in active mode. AdjustClockHour is the default mode of operation for the adjust circuit when the circuit is reset.

The Adjust Module Code (Contributed by MagdElDin AbdalRaaof)

To explain this part, we first need to explain the implementation of an x-bit counter module that we created. The module has 2 parameters, x for the number of bits and n to indicate the maximum number the counter is allowed to reach. Its inputs are a clock signal (clk), a reset signal (rst), an enable signal (en), and a signal that indicates if it should count up or count down (UpDown). It outputs the n-bit number in the counter (out). Upon each rising edge of either the clock or the reset, the counter first checks if the reset signal is enabled. If it is, then the counter outputs an x-bit 0. Otherwise, it checks if the counter is enabled. If the counter is enabled, it proceeds to check if the counter should count up or down using the UpDown signal. If UpDown is 1, then the counter should count down, but if the counter is at 0, then the counter loops to the maximum possible value instead. If UpDown is 0, then the counter should count up, but if the counter is at (n-1), the counter loops back to 0. It is assumed that if none of these conditions are met that the counter maintains its old value.

The adjust module itself takes in the clock signal (clk), reset signal (rst), enable signal (en), processed signals of the button input from the left, right, up, and down buttons respectively (INL, INR, INU, IND). It then outputs the values of the alarm hours (tens (ATH) and units (AH)), alarm minutes (tens (ATM) and units (AM)), clock hours (tens (AJTH) and units (AJH)), clock minutes (tens (AJTM) and units (AJM)), and the select line that chooses between which one of the aforementioned outputs is changed (selected\_adjust). It also outputs the data necessary to turn on the LEDs that indicate which part is being adjusted right now (LD12, LD13, LD14, and LD15). Inside the module, there are wires such as enable\_alarm\_hours that activates when the adjust module is enabled, and selected\_adjust is 2’b10, enable\_alarm\_minutes that activates when the module is enabled and selected\_adjust is 2’b11, enable\_clock\_hours that enables when module is enabled and selected\_adjust is 2’b00, and enable\_clock\_minutes that enables when module is enabled and selected\_adjust is 2’b01. ATH, AH, AJTH, and AJH have all been declared as registers for a reason that will be explained later. We alternate between which counter is going to be active using states marked by selected\_adjust. We have a 5-bit, modulo 24 counter that outputs a 5-bit number that represents the current hour that the alarm is set to. Its output is AH\_temp, and it is used in a case statement to select the value of ATH and AH. The same applies for modifying the clock hours using AJH\_temp. For modifying alarm minutes, two counters are used, both of which are 4-bit, but the first one is modulo 10 and the second one is modulo 6. The modulo 6 counter only activates if the first one is active and AM is 0 and UpDown is 1, or when the first one is active and AM is 9 and UpDown is 0. A similar system is used for the adjusting of clock minutes.

Difficulties faced during the creation of this module included the module initially only showing binary numbers as opposed to denary due to faulty connection with the main decoder. This was fixed by explicitly declaring wires and registers. Additionally, the 24 hour counter was a particularly challenging aspect. By trying to implement it in the same way as the minute counters, the counter could go all the way up to 29. Attempts to remedy this using separate counters only made matters worse as numbers would shift around, seemingly at random because the enable conditions for the counters would be fulfilled unwittingly. Additionally, the adjust portion was swapped, where the alarm adjustment took the first 2 LEDs and the clock adjustment took the last 2 LEDs. This was eventually fixed.

The Clock Module Code / FSM diagrams (Contributed by Basmala Abdelkader)

This module, called *Clock\_Circuit*, was done by integrating another module and using it inside this module. The module we used was *Counter4bit*. *Counter4bit* is a 4 bit up counter that takes the following inputs and outputs:

**Inputs:** Data of 4 bits, and a clock, reset, enable, load\_zero, and load\_value (will be explained).

**Outputs:** Count which is 4 bits, and declared as a register (used in always block).

For the Inputs, the Data is the number transferred to the counter and the starting point of the count, and the load\_value input allows the data to be transferred to the counter depending on whether it is high or low. The reset and load\_zero inputs set the counter back to 0. The enable allows the counter to start counting if it’s high, and if low it stops the counter at its last value.

The output is a single 4 bit variable named count which changes depending on the input as explained above.

In the *Clock\_Circuit* module, we have included six such counters, each for a section of the clock. The first two counters are for counting seconds. The first counter, named c0, counts the first digit, where the data input is 4 bit 0, clock, reset, the enable set to 1, load\_zero is high when seconds reach 9, load\_value is set to 0, and the output is saved in the wire S. The second counter, named c1, counts the tens digit of seconds, where all the inputs of the first counter are the same in the second except for the enable which becomes high when S reaches 9, load\_zero becomes high when the variable TS (representing the tens digit of seconds) reaches 10, load\_value is set to 0, and the output is saved in the TS wire.

The next two counters are the minutes counters. The source of their data input is the Adjust mode to set the values to specific starting points if needed, and the data inputs are called AJM and AJTM, for Adjust Minutes and Adjust Tens Minutes (each digit). The first counter, called c2, also takes the inputs clock, reset, the enable value being high or low depending on whether TS has reached 10, the load\_zero being high or low depending on whether CM (counter for minutes digit) has reached 10, the load\_value being high or low depending on whether both INC2 and Mode are high or not, and the output CM as a wire. The second counter, named c3, has the data input AJTM, clock, reset, the enable becoming high only when CM reaches 10, the load\_zero becoming high if CTM reaches 6, the load\_value becoming high if INC2 and Mode are both high, and the output being the wire CTM.

The last two counters are responsible for counting the hours. The first counter, named c4, counts the first digit, taking in the data input AJH for adjust hour, clock, reset, enable set to high only if CTM reaches 6, load\_zero set to high only if one of the following conditions are met: CH reaches 4 and CTH reaches 2, or CH reaches 10, the load\_value set high only if both INC2 and Mode are high, and the output wire CH. The second and last counter, called c5, takes the data input AJTH for adjust tens hour, clock, reset, enable set to high if CH reaches 10, load\_value set to high if CTH reaches 2 and CH is 4, the load\_value set to high if INC2 and Mode are both high, and the output wire CTH. This concludes the module *Clock\_Ciruit*.

Difficulties regarding this module included the clock not having a proper reset system in place, and it failing to adjust when exiting adjust mode. That second error was especially difficult and was fixed by making the clock detect the rising edge of the processed center button input, which was processed at 100 Hz even though the clock circuit operates with 1 Hz.

Alarm Module (Contribution by Mohammed ElShemy)

Initially, we tried creating this part using structural logic, but this proved too much of a hassle to do. Eventually, we had it so that the code for detecting when the alarm should be enabled should simply be a part of the finite state machine in the main circuit while the alarm circuit is simply a signal being alternated depending on whether or not the alarm is active. Its inputs are the 100 Hz clock (clk), a reset signal (rst), an enable signal (en), and it output the signal that acts as the alarm (LDA). If rst was 1 or en was 0 then LDA would be 0, otherwise, the value of LDA would alternate each clock cycle.

Testing (Contributed by Mohammed Elshemy):

Our testing methodology was predominantly brute force. After trying for about 7 hours to have a functional testbench, we realized it would be much quicker to try out every change on the FPGA board than to spend more time creating a functional testbench. This did, however, have the drawback of several minutes spent between each change and the testing phase as the program had to be synthesized, implemented, and have its bitstream generated before we could program the board. Additionally, although not stated, Mohammed Elshemy was responsible for any and all code documentation and cleanup. Effectively, he wrote down the function of each part of the code and removed any redundant code or code that was commented out. However, the Counter4bit module remains despite the existence of the X\_Bit\_Counter module because the latter does not have a load feature, and trying to implement it would have caused trouble in the code as much more troubleshooting and bug fixing would have been required to eliminate that redundancy, and we prioritized the completion of a functional result.