Digital Design Project Report: Digital Alarm Clock

Farida Said, Hoda Hussein, Laila Sayed, and Zeina Elsawy
900221087, 900223388, 900223389, 900223285
CSCE230102 - Digital Design I (2024 Spring)
Section 01 and 02
Department of Computer Science and Engineering, AUC

Abstract

In this report, we will discuss all the details of the process of reaching our project goal. We will outline the design, giving a brief explanation of the usages of the modules and their implementations. We will then discuss the issues that we faced during implementation, and what we did to fix our errors, and test or validate our outputs. We will then finally state the contributions of each group member in the project.

Introduction

The design and implementation of an alarm-equipped digital clock on the BASYS3 FPGA development board is the main goal of this project. The clock has two main modes that it uses a finite state machine (FSM) to control: "clock/alarm" and "adjust." Simple push button interactions allow users to set the alarm and current time. The clock has a 7-segment display that shows the hours and minutes. Blinking decimal points and LEDs provide extra visual cues. When the current time coincides with the set up alarm time, the alarm initiates both a blinking LED and a sound output.The project uses Verilog to describe the hardware and implements exact state control and timing.

Outlining the design

For our clock alarm simulation, we adopted the Mealy machine approach and divided the process into distinct states. We identified six states in total: clock, alarm, adjust time hour, adjust time minute, adjust alarm hour, and adjust alarm minute. We then created a state diagram table to transition between these states.

Present State	Next State					Output
	BTNC	BTNL	BTNR	BTNU	BTND	
Clock	Adjust Time Hour	Clock	Clock	Clock	Clock	Behavior of next stage State assignment LED assignment Enable assignment Updown assignment Clock Frequency
Adjust Time Hour	Clock	Adjust Alarm Minute	Adjust Time Minute	Adjust Time Hour	Adjust Time Hour	
Adjust Time	Clock	Adjust Time	Adjust Alarm	Adjust Time	Adjust Time	

Minute		Minute	Hour	Minute	Minute	
Adjust Alarm Hour	Clock	Adjust Time Minute	Adjust Alarm Minute	Adjust Alarm Hour	Adjust Alarm Hour	
Adjust Alarm Minute	Clock	Adjust Alarm Hour	Adjust Time Hour	Adjust Alarm Minute	Adjust Alarm Minute	
Alarm	Clock	Clock	Clock	Clock	Clock	

To represent and navigate through the states in our Mealy finite state machine, we utilized if statements. In the Mealy model, outputs are not associated with the current state but are instead determined during transitions. We reflected this in our code by placing transition conditions within if statements, with the corresponding outputs immediately following in the block. The outputs varied depending on the state, so we categorized them into five primary components: behavior of the next stage, state assignment, LED assignment, enable assignment, up/down assignment, and clock frequency. Within each if statement, we detailed the outputs for that specific state. For instance, here's a code snippet from the case "Adjust time minute" to illustrate our approach:

```
if (BTNC)
            begin
                next_state = CLOCK_ALARM;
                enable=1'b1;
                LD0 = 0; LD12 = 0; LD13 = 0; LD14 = 0; LD15 = 0;
                Updown =1'b1;
                clockinput = clk1HZ;
                countHours = 1'b0;
                countMIn = 1'b0;
                alarm_count_hours = 1'b0;
                alarm_count_min = 1'b0;
             end
```

In our implementation using a Mealy FSM, the if statement determines the transition condition, such as pressing BTNC in this example. Following the if condition, the block specifies the setup for the subsequent state, here transitioning to "Clock Alarm." The outputs for the next state are defined as follows:

- Behavior of next stage:
  countHours = 1'b0;

countMIn = 1'b0;
alarm_count_hours = 1'b0;
alarm_count_min = 1'b0;

- State assignment: next_state = CLOCK_ALARM;
- LED assignment: LD0 = 0; LD12 = 0; LD13 = 0; LD14 = 0; LD15 = 0;
- Enable assignment: enable=1'b1;
- Updown assignment: Updown =1'b1;
- Clock frequency: clockinput = clk1HZ;

At the conclusion of each case, the final else block details the outputs for the default state. We previously defined this default state within the code as holding the current state it is in. This approach maintains the behavior of the current state without changing its status explicitly within the FSM itself.

Our Binary_Counter module is a flexible parameterized module designed to count up or down based on a control signal. The module parameters allow it to be adapted for different bit widths and maximum values.

Parameters:

x: Specifies the bit width of the counter.

n: Specifies the maximum count value.

Inputs:

clk: The clock signal that drives the counter.

reset: A signal to reset the counter to zero.

en: Enable signal to control whether the counter is active.

updown: Determines the counting direction (up or down).

Output:

count: The current count value.

A clock divider is a circuit that receives a clock with a specific frequency *fin* and outputs a clock with a lower frequency *fout*. To compute the frequency of the output clock, use the formula

```
fout = fin / 2n
```

Where n is the number of cycles in the input clock before the output clock flips from 0 to 1 or 1 to 0. This module is essential to the project; therefore, we decided to remove the clock dividers from any modules which already had clock dividers implemented inside them so that they could

be implemented directly into the FSM. We declared two clock outs at frequencies of 200Hz and 1Hz, which were called depending on the state we were in for "Adjust" and "Clock" respectively. The module had a parameter of 50000000, which was set as our default, and when we created it, we altered the frequency based on the desired output.

The pushbutton main function is to detect the input signal under conditions of the clock divider, debouncer and rising edge detector to produce a clean output signal z. First the clock divider takes the input clk signal and reduces its frequency by the parameter n=500000, to produce a slower clock which is translated as a clk_out output signal. Consequently, the signal that the pushbutton receives is now slower, so we can detect its behavior. After that, the debouncer takes the clk_out signal, the reset, and the “in” signal as its inputs and filters any noise that could be involved with the pushbutton, and sends the filtered signal as an output signal w1. the debouncer initializes three wires to zero at reset signal equals one, then it makes a non-blocking assignment of our input signal to the first wire at the first positive edge of our clock, which is then assigned as an input to the next wire, and the second wire is assigned as another input to the third wire, so we can ensure that the signal is transmitted in a clear process without any distortions and that the signal changes with every positive edge of the clock. Then the synchronizer takes the debouncer’s output signal w1, and the clk_out as input and produces a synchronized output w2. A synchronized output means that the output signal is aligned correctly with the clock to avoid any timing distortions like the metastable state. Then finally the rising edge detector takes the synchronizer output signal w2, the clk_out and reset and matches the positive rising edge of the clock with the external push of our pushbutton, and outputs a push which is aligned with the rising edge of the clock, so the button is pressed successfully.

In the clock state, we initially set the clock_alarm_display value with 0, to indicate to the digital clock that we are in the clock mode and not the alarm mode, distinguishing the states from each other. We first check ‘zflag’, which checks whether the clock is equal to the set alarm. If it is true, it enters the alarm state. We set the enable to 1 so the clock would be running. All the LEDs are set as 0 so they would be turned off, except for LED 0. We assign to it the 1Hz clock that we got from the clock divider so it would blink with a frequency of 1Hz when the clock is equal to the set alarm. Updown is set with 1, but it is a don’t care variable in this case anyway, so it does not differ. We also do not take the count of the hours and minutes of the clock or the alarm as we do not need them. They are needed just when incrementing and decrementing. It then checks if the center button is pressed. If this is the case, it enters the first state in the adjust mode, which is the adjust time hour state. The enable that keeps the clock running, that was previously set as 1, is set to 0. LED 0 is set to 1 so it would be on as long as we are in the adjust mode. LED 1 also is set to 1 so it would be indicated which case in the adjust mode we are in. All the other variables are kept as they were in the clock state. The final condition is if the zflag was false, meaning that the clock was not equal to the set alarm, and the center button is not pressed. In this case, we want to stay in the clock state, so we set all the LEDs as 0 so they would all be turned off. Letting LED 0 off indicates that we are in the normal clock state. The enable is

also set to 1 so the clock would run. The next state is set as the current state, keeping it in the clock state as required. We change the digital clock input to the 1Hz clock we got from the clock divider, so it would operate with a speed as a normal clock.

The increment and decrement operations are managed through specific states within the FSM, responding to user inputs via buttons. The up button (BTNU_b) and the down button (BTND_b) are used to increase or decrease the selected parameter, respectively. These parameters include the hour and minute for both the time and alarm settings. The FSM transitions between different states based on the button presses, updating the corresponding parameter and using LEDS as a signal of which parameter is currently adjusted.

Adjust Time Hour/Time minute States

When the FSM is in the ADJUST_TIME_HOUR/ADJUST_TIME_MINUTE state, pressing the up button (BTNU_b) or down button (BTND_b) will adjust the clock's hour/minute setting. The logic is as follows:

Increment Time Hour: If BTNU is pressed, the state remains in ADJUST_TIME_HOUR/ADJUST_TIME_Minute, and a control signal called Count Hours/Count Minutes is enabled to indicate that we are currently adjusting the time hour/time minute. Another control signal called Updown is set to 1 to indicate that we are incrementing. This results in the hour/minute value increasing by one. Finally, the input clock is adjusted to 200 Hz to make it slow enough to allow us to adjust and see the adjustments.

Decrement Time Hour: If BTND is pressed, the state remains in ADJUST_TIME_HOUR/ADJUST_TIME_Minute, and a control signal called Count Hours/Count Minutes is enabled to indicate that we are currently adjusting the time hour/time minute. Another control signal called Updown is set to 0 to indicate that we are decrementing. This results in the hour/minute value decreasing by one. Finally, the input clock is adjusted to 200 Hz to make it slow enough to allow us to adjust and see the adjustments.

The corresponding LED (LD12) is turned on to indicate that the hour is being adjusted, and LED (LD13) is lit to show that the minute is being adjusted, while other LEDs remain off.

Adjust Alarm Hour/Alarm Minute States

In the ADJUST_ALARM_HOUR state, the alarm hour can be adjusted, while in the ADJUST_ALARM_MINUTE state, the minute setting for the alarm is adjusted:

Increment Alarm Hour/Alarm Minute: If BTNU is pressed, the state remains in ADJUST_ALARM_HOUR/ADJUST_ALARM_MINUTE, and a control signal called alarm_count_hours/alarm_count_min is enabled to indicate that we are currently adjusting the alarm hours/minutes. Another control signal called Updown is set to 1 to indicate that we are incrementing. This results in the hour/minute value increasing by one. Finally, the input clock is adjusted to 200 Hz to make it slow enough to allow us to adjust and see the adjustments.

Decrement Alarm Hour/Alarm Minute: If BTND is pressed, the state remains in ADJUST_ALARM_HOUR/ADJUST_ALARM_MINUTE, and a control signal called alarm_count_hours/alarm_count_min is enabled to indicate that we are currently adjusting the alarm hours/minutes. Another control signal called Updown is set to 0 to indicate that we are decrementing. This results in the hour/minute value decreasing by one. Finally, the input clock is adjusted to 200 Hz to make it slow enough to allow us to adjust and see the adjustments.

LED (LD14) indicates the alarm hour adjustment mode, and LED (LD15) indicates the alarm minute adjustment mode.

The MinHour module is responsible for deriving and displaying our digital clock. The MinHour module uses three counters for seconds, minutes, and hours, which are synchronized and controlled to provide accurate time updates. The module interfaces include inputs for clock signals, reset, enable, and control signals for counting direction. The outputs represent the time units in decimal format, divided into units and tens.

Ports and Signals

Inputs:

clk_out_seconds: Clock signal of 1 HZ.

reset: Resets all counters.

enable_seconds: Enables counting of seconds.

Updown: Controls the counting direction (up or down).

count_hours: Enables adjustment of hours.

count_min: Enables adjustment of minutes.

Outputs:

seconds_units: Units digit of seconds (0-9).

seconds_tens: Tens digit of seconds (0-5).

minutes_units: Units digit of minutes (0-9).

minutes_tens: Tens digit of minutes (0-5).

hours_units: Units digit of hours (0-9).

hours_tens: Tens digit of hours (0-2).

Wires

countsec: 6-bit wire for counting seconds.

countMin: 6-bit wire for counting minutes.

counthours: 5-bit wire for counting hours.

Functionality Breakdown

Counting Logic:

Seconds Counter: Uses a 6-bit counter with a range of 0-59. It increments with each clk_out_seconds pulse unless reset.

Minutes Counter: Also a 6-bit counter for 0-59. It increments when countsec reaches 59 and enable_seconds is 1 or when count_min is 1.

Hours Counter: A 5-bit counter for 0-23. It increments when both countsec and countMin reach their maximum values and enable_seconds is 1 or when count_hours is 1.

Decimal Conversion:

The module converts binary counts to decimal format for each time unit by separating the digits for each counter :

Seconds: Units and tens.

Minutes: Units and tens.

Hours: Units and tens.

Efficiency Considerations

Using 6-bit counters for seconds and minutes, and a 5-bit counter for hours is more efficient than using 3 or 4-bit counters because Using larger counters (6 and 5-bit) reduces the number of counters required to manage the time units. Moreover, using multiple 3 or 4-bit counters would increase complexity, resource usage, and potentially the cost.

We use the AlarmMinHour module in the digital clock module to separate between the clock mode and the alarm mode. This module stores the values of the set alarm and has the ‘updown’ parameter to increment and decrement the hours and minutes of the alarm when adjusting it. This module also helps us in separating the adjusted time from the adjusted alarm. It uses a modulo 60 counter to display the minutes and a modulo 24 counter to display the hours. Unlike the MinHour module that is used for the clock, we do not need to have a counter for the seconds. This module has only 2 enables, 1 for the hours and 1 for the minutes. We also do not need to check the minutes or seconds reached to reset the counter, as done in the MinHour counter. Finally, we output the hours units, hour tens, minutes units, and minutes tens using the modulus (%) operator and division(/) to separate the value that we got from the counters, which we then use to compare with the hours and the minutes of the clock, and that we also use in the display.

The digital clock module was the module that combined both the MinHour module needed for the clock state and the adjust time hour and minute states, and the AlarmMinHour needed for the adjust alarm hour and minute states. It has as inputs a clock signal that counts the seconds, a higher frequency clock signal that is needed to have a multiplexed display, ensuring that each digit is displayed sequentially and rapidly enough that they all appear to be shown simultaneously to the human eye. It also has an active high reset signal that initializes the clock and the alarm. A display input signal is used to control whether the clock is being displayed or the alarm. Four input signals are also used for the time hours, time minutes, clock hours, and clock minutes to control their adjustments and settings. An ‘updown’ signal is also used as it is needed for incrementing and decrementing. An enable signal is used to control when the clock is counting. The module displays the required output to the seven-segment display and has another output signal that controls which digit (anode) of the display is active. A 4-bit register ‘num’ is used to hold the current digit to be displayed. The module uses a binary counter, driven by a 200 Hz clock, to have a multiplexed display, which outputs ‘count’, the multiplexing signal. The ‘num’ register is updated by the always @(count) block, which uses the count's current value to determine which digit is displayed. This multiplexing signal with the ‘num’ register and the control signal for the anodes drive the seven segment display. As for the ‘zflag’, which tells us whether the alarm time has been reached or not, it is set as high when the alarm time matches the current time. We compare the tens and units of the alarm hours and minutes that we get from the alarm module to the tens and units of the clock hours and minutes that we get from the ‘MinHour’ module. We also check that the seconds in the clock are equal to zero, as we do not check for the seconds in the alarm module. If these cases are all true, then the ‘zflag’ is true.

A seven-segment display (SSD) is a common electronic display device used for showing decimal numerals, crucial for projects like digital clocks due to its simple yet effective way of presenting information. Each SSD comprises seven Light Emitting Diodes (LEDs) arranged to represent various digits. In the context of our project, the display not only shows clock and alarm values but also incorporates a control mechanism for display selection. This is achieved using two inputs and two outputs:

- 2-bit **en** input for enabling specific anode
- 4-bit **num** input representing the number to be displayed
- 7-bit **segments** to individually control each segment of the display
- 4-bit **anode_active** for selecting active anodes

The module translates each input number ranging from 0 to 9 to corresponding segment patterns on the display, represented in 7 bit binary code. Similarly, the input determines which display anode is active at any one moment, allowing distinct numbers to be shown on many screens in succession. This arrangement allows us to manipulate the clock frequency in order to appear

although all 4 anodes are constantly active. The code snippet down below details how the digits are depicted on the SSD, with each segment's state being defined in a case structure. Notably, the segments are active-low, meaning a '0' turns a segment on (LED lights up), and a '1' turns it off (LED remains dark). This module is later utilized and instantiated within the “Digital Clock” module in the project, where each digit is represented as follows:

```
0: segments = 7'b0000001;
1: segments = 7'b1001111;
2: segments = 7'b0010010;
3: segments = 7'b0000110;
4: segments = 7'b1001100;
5: segments = 7'b0100100;
6: segments = 7'b0100000;
7: segments = 7'b0001111;
8: segments = 7'b0000000;
9: segments = 7'b0001100;
```

The function of the alarm state is to make LD0 blink under a condition, this condition is to check whether the current time matches the set alarm, if it matches then the alarm function will display and the LD0 will blink, if not then the LD0 will not blink. In addition, the LD0 exits the blinking condition when we press any of the five buttons we have/: BTNC, BTNU, BTND, BTNL, BTNR. In the main alarm function, first the display alarm is initially set to 1’b1 which is off because it is active low. Then we have the if condition which indicates that the clock will exist the blinking state if any of the buttons is pushed, and if any button is pushed the alarm will go back to the default state which is CLOCK_ALARM, and in this state all LEDs are set to zero and the Up/Down for increment and decrement is enabled however all the hours-minutes counters are disabled. Then we have the else condition which enables the state of the ALARM. In the alarm mood all the LEDs are off except LED0 it is on and blinks with a frequency of 1HZ. All the counters for hour-minute are enabled but the Up/Down for increment and decrement is disabled. Then looping again if we pressed any of our buttons, we will go back to the default CLOCK_ALARM state.

```
clock_alarm_display = 1'b0;
        if(BTNC | BTNU | BTND | BTNR | BTNL)begin
            next_state = CLOCK_ALARM;
            enable=1'b1;
            LD0 = 0; LD12 = 0; LD13 = 0; LD14 = 0; LD15 = 0;
            Updown =1'b1;
```

```
            clockinput = clk1HZ;
            countHours = 1'b0;
            countMIn = 1'b0;
            alarm_count_hours = 1'b0;
            alarm_count_min = 1'b0;
            end
        else begin
            next_state = ALARM;
            enable=1'b1;
            LD0 = clk1HZ; LD12 = 0; LD13 = 0; LD14 = 0; LD15 = 0;
            clockinput = clk1HZ;
            Updown = 1'd1;
            countHours = 1'b0;
            countMIn = 1'b0;
            alarm_count_hours = 1'b0;
            alarm_count_min = 1'b0;
        end
      end
  endcase
end
```

Implementation issues and validation activities

The first implementation issue we faced was clock divider precision. We had to correctly distinguish when we should take the output of which clock divider, in order for the alarm clock to work correctly, and we had to match the frequency of the clock divider used for the clock with the frequency of the push buttons so there would be no overlapping. The second implementation issue we faced was multiplexing the seven-segment display, which we then solved by adding a binary counter to the digital_clock module. We also first started our FSM with two states only: clock/alarm state and adjust state. This then seemed impractical, so we separated the adjust states into 4 states: adjust time hour, adjust time minute, adjust alarm hour, and adjust alarm minute. After finishing most of our module, we decided to separate the clock/alarm state into a state for the clock and the state for the alarm. This would help us control when the LED 0 blinking and the buzzer stop while the clock continues operating normally. While doing so, we encountered a difficulty with appropriately implementing the necessary layout of the FSM based on the structure we chose. We were able to solve our problem by utilizing if statements, but we needed to make use of the 'begin' and 'end' functions to block out our outputs because we had a lot of different things we needed to display all at once. Another crucial implementation issue that we faced is that we first used in the min_hour module 6 counters instead of 3. The counters for the hours were a mod 3 counter and a mod 10 counter. However, using this approach later brought up the issue that we were not able to accurately control when the mod 10 counter resets, as it resets once after 9 and once after 4. So, we changed our approach in this module to use 3 counters instead: 2 mod 60 counters and one mod 24 counter.

As for the validation activities we used through our process, we created testbenches when doing smaller modules, like the rising edge detector, to ensure that they are working correctly. When we first implemented the base of our FSM module, without logic, we also created a testbench that just checks if it operates generally as desired, with error messages as outputs to determine where the error occurred. We also generated a clock only first, and created for it a constraint file and generated it on the FPGA. This was to ensure that the clock works correctly and counts as desired so we would then move forward to the next step. When testing the clock, we adjusted the frequency using the clock divider so the clock would be much faster than the normal clock, so we could check how it operates during the 24 hours. To avoid many running mistakes when creating the FSM module, we created the clock state first and ensured that it works correctly and moves to the next state correctly. We then implemented the first adjust state, adjust time hour, and made sure that the increment and decrement and working and the left and right buttons change the state to the next desired state correctly. After we made sure this implementation is correct, we copied it for the three other adjust modes, but edited the variables in it as needed. We finally added the alarm mode, that was also a copy of the rest of the states with a few adjustments.

]

Contributions

Farida Said led the main FSM implementation and integration for the digital alarm clock project. Hoda Hussein managed the increment and decrement functions along with the related state transitions. Laila Sayed focused on input handling and synchronization, ensuring smooth operation and interaction. Zeina Elsawy was responsible for the clock divider logic and display management, crucial for the functionality and user interface of the clock.

References

1. *http://www.sunburst-design.com/papers/CummingsSNUG1998SJ_FSM.pdf*