ECE 482: Digital Systems: High Level Synthesis and Codesign

Digital Timer and Alarm Clock

Grant Abella

Contact Information

Grant Abella
Electrical and Computer Engineering Student
Caterpillar College of Engineering and Technology
Bradley University

1821 W. Callender Ave.
Peoria, IL, 61606, USA

Phone: +1 (630) 776-4026
Email: gabella@mail.bradley.edu

Table of Contents

1 Introduction

Digital clocks and displays have been in wide use since the 1970s, and since then their adoption has grown almost exponentially. Digital displays provide users with faster recognition and processing of information than analog displays such as the traditional wall clock and wrist watch. Smart watches such as the Fitbit and Apple Watch have grown in popularity over the past few years because of a combination of factors including their ease of use, functionality, and compatibility with devices that most people use daily such as smart phones and computers.

Microcontrollers and FPGA boards are suitable candidates for digital clock and watch hardware because of their size and efficiency. Many of these pieces of hardware contain components that go beyond simple timekeeping capabilities. Things like temperature sensors, gyroscopes, and accelerometers can be utilized to provide detailed information for things such as weather, step-tracking, and even medical information.

With this said, the intention of this project was to provide the user with a catalogue of clocks and alarms, similar to what one would see on a smartphone's clock application. Ideally, the user would be able to interface with the FPGA board via a web server, and perform tasks such as setting or changing an alarm.

2 Project Overview

This project evolved over the course of its development. In order to express the changes from the original plan to the final product, both will be described in this section. Additionally a brief explanation as to why certain changes were made will be provided.

The project was originally split into 5 main objectives:

A. Obtain input from the user via a web server console
B. Store the received information into the board's memory
C. Process the data
D. Output the processed information to the board's display in real-time
E. Wait for further input

Objective A was to be achieved using the development board's built-in RJ-45 Ethernet jack, with send and receive functions of the Ethernet frame being handled by an onboard Ethernet physical transceiver (PHY). The interfacing between the board and the user would be accomplished with a terminal emulator such as Tera Term, as used in one of Digilent's tutorials[i]. Objective B is straightforward as all it would entail is storing the input that the user enters into the terminal into the DDR2 memory on the FPGA. From there, the data processing (Objective C) would be handled using simple programming logic and math, depending on what information is being processed. Output to the display (Objective D) would be done in real-time using the results from the aforementioned data processing. This entire process (Objectives A through D) would be repeated, providing a fluid experience for the user.

The data gathered from the user would consist mainly of integer values for choices related to operations on the clock or alarm, as well as values to set for them.

The project was eventually composed of the following objectives:

A. Obtain input from the user via the board's GPIO components
B. Store the data
C. Process the information
D. Wait for further input

These changes were made mainly because of complications that arose when trying to set up and connect to the web server. The complications were mainly related to MAC and IP address conflicts that would have been too time consuming to fix by the project's deadline. As a result, all of the actions that were originally planned to be performed over the server were instead moved to a more local setup using the board and it's GPIO.

2.1 User Input

Input from the user is completely handled through the FPGA's DIP switches and buttons. Although the board has 16 DIP switches and 5 push buttons, only 4 switches (SW0, SW1, SW2, and SW15) and 2 buttons (BTNU and BTND) were needed to adequately get information from the user. The first three switches are used to select which aspect of the clock is to be programmed. SW0 corresponds to the seconds, SW1 to the minutes, SW2 to the hour. By default, the clock's alarm is turned off, however, when SW15 is enabled, the alarm function becomes active. Programming for the alarm is done in the same manner as the clock, with SW0, SW1, and SW2 pertaining to the values of the alarm's seconds, minutes, and hours respectively.

When in programming mode for either the clock or the alarm, the user simply has to press BTNU to increase the selected value, and BTND to decrease the value. The table below describes every possible behavior of the program in response to user input.

SW0	SW1	SW2	SW15	BTNU	BTND	Clock Result	Alarm Result
0	0	0	0	X	X	Runs normally	Disabled
0	0	0	1	X	X	Runs normally	Enabled
1	0	0	0	1	0	Seconds increase by 1	None
1	0	0	0	0	1	Seconds decrease by 1	None
1	0	0	1	1	0	None	Seconds increase by 1
1	0	0	1	0	1	None	Seconds decrease by 1
0	1	0	0	1	0	Minutes increase by 1	None
0	1	0	0	0	1	Minutes decrease by 1	None
0	1	0	1	1	0	None	Minutes increase by 1
0	1	0	1	0	1	None	Minutes decrease by 1
0	0	1	0	1	0	Hours increase by 1	None
0	0	1	0	0	1	Hours decrease by 1	None
0	0	1	1	1	0	None	Hours increase by 1
0	0	1	1	0	1	None	Hours decrease by 1

Table 1: Program response to user input from GPIO peripherals.

2.2 Program Logic

The actual logic coding for the program was simple and minimal. The first task was to set up a timer that would trigger an interrupt every second. Since the generated hardware design

contained an interrupt controller and two timers (all running on the AXI bus), it was just a matter of initializing and configuring the controller and one of the AXI timers (*axi_timer_0*). Once this was accomplished, the interrupt handler was written and was comprised of four things:

A. Increment the value corresponding to seconds for the clock by one
B. Stop the timer
C. Reset the timer to its starting value
D. Start the timer

It is now important to focus on the handling of the data associated with the clock and the alarm. Because the both need to hold information such as seconds, minutes, hours, etc. it made sense to program a C structure to store these variables in a neatly contained object. In addition to this, functions for handling the changing of these values would be necessary in order for the time programming capabilities to work, as well as the interrupt handler for *axi_timer_0* to function properly. The Time structure for this project contained the following members:

- isHour12 - a Boolean denoting if the clock is 12 hour or 24 hour
- alarmOn - a Boolean denoting if an alarm is enabled for the clock
- hour - an integer corresponding to the hour value of the clock
- minute - an integer corresponding to the minute value of the clock
- second - an integer corresponding to the second value of the clock
- alarm - a pointer to another Time structure

While the purposes of most of the variables above are clear, elaboration on the alarm member is needed to fully describe the overall functionality of the structure. Although much of the logic used to check whether a clock has reached its alarm time could have be hardcoded, it made more sense to simply tie two Time objects together using pointers. This allows for more freedom when coding, as making changes to the alarm can be done using a reference to the alarm object. Without this, when editing any code that deals directly with the alarm (and referring to it using its variable name), the programmer would need to be sure to change any uses of the alarm Object name accordingly. Also, using references leaves open the possibility for future improvements to the project. The programmer could potentially design a system which supports multiple alarms, cycling through them using a pointer/reference system to assign one of many different alarms to the clock.

Seven functions work with the members of Time objects. These functions are listed and described below, with some being given more detail as the logic for several of the functions is similar.

```
void setTime(struct Time *t, int hr, int min, int sec)
```

This function was used to directly set the time of a Time object. The object is passed to the function by reference, and as a result, the function can be used with both the main clock object and the alarm associated with the clock. The remaining arguments are the values to be assigned to the object's corresponding members.

```
void incrementSecond(struct Time *t)
```

The interrupt handler assigned to *axi_timer_0* calls this function to increment the second value for the clock. The clock object is passed by reference. The function begins by incrementing the value, then checking whether a rollover should occur. Rollovers occur when the second value increments from 59. At this point, second is set to zero, and the following function is called.

```
void incrementMinute(struct Time *t)
```

Only called by `incrementSecond`, this function's logic is similar in nature. When a rollover happens, the next function is called.

```
void incrementHour(struct Time *t)
```

Again, the role of this function is similar to the past two. The notable way this function differs is in the value at which rollovers occur. If the hour value of the object increments from twelve, a comparison is made to check whether the object is set as a twelve or 24 hour clock. If it is a twelve-hour clock, the hour is set to one, otherwise the value increments to thirteen.

```
void progSecond(struct Time *t, int val)
```

This function is related to the programming functionality described in the previous section on user input. A Time object and signed integer are passed to the function, and the integer is added to the second member of the respective object. This method was necessary because of the user's ability to either increment or decrement whichever value he or she is programming. For ease of use, one of two rollover values is checked in this function depending on whether an increment or decrement is occurring. This allows the user to jump from 59 to zero when BTNU is pressed, and from zero to 59 when BTND is pressed.

```
void progMinute(struct Time *t, int val)
```

This function behaves in the exact same manner as above, with the only difference being that it changes the value of the minute member.

```
void progHour(struct Time *t, int val)
```

This function also behaves similarly, with rollover values changing based on the twelve-hour setting of the Time object.

2.3 System Output

The output of this system is performed by utilizing the board's eight-digit common anode seven-segment display. This peripheral runs on the AXI bus and is dual channel in nature, with channel 1 controlling the cathode, and channel 2 controlling the anode. By changing the data direction of the cathode channel, one of the eight display digits is selected for display. Through changing the direction of the anode channel, a combination of the seven segments for the selected display digit are illuminated. Although only a single digit can be displayed at one time, quickly cycling between which digits are selected for display can give the illusion of several digits being displayed at once.

This process was handled by two functions. The first function, `displayTime`, receives reference to a Time object, and a reference to a GPIO display instance. `displayTime` is only ever used for outputting the normal clock to the seven-segment display. Ensuring accurate output is accomplished through a two-step process:

A. Extract the ones-place and tens-place digits of the second, minute, and hour members of the clock
B. Loop through the first six digits of the display, adjusting anode and cathode values accordingly

Step A is achieved by performing simple division and modulus operations on the members of the Time object. The extracted digits are then stored in a six integer array, in the order that each would appear on the display.

Step B is a simple loop comprised of setting the direction for the anode channel of the display to pick a digit of the display to work with. After that, the direction for the cathode channel is changed, depending on the value of whatever extracted digit is to be displayed. A small delay of two milliseconds is then applied, as recommended by Digilent to avoid a flickering effect on the display[ii]. After this, the next digit is handled until all six are fully displayed.

When the user selects a programming mode, the selected value toggles on and off on the display to signal which value is going to be changed. To accomplish this, a separate display function was needed.

`displayTimeBlink` takes the same two arguments as its sister function, as well as an additional argument of type u32. This is the value of the board's DIP switches, and was needed so that the function would know which value the user has selected for programming. The function first extracts the digits in the exact same manner as `displayTime`. After this, the value of the AXI timer is captured. The function checks whether the timer's value is greater than a certain hex value (corresponding to 500 milliseconds), and branches into two sections depending on the result of that comparison.
If the value is greater than the number corresponding to 500 millisecond, the time is displayed as usual by calling `displayTime`. Otherwise, a process begins in order to handle the blinking of the two seven-segment display digits relating to whichever selection has been made by the user. This process is accomplished by simply assigning the cathode values to each of the two anodes a binary value of 11111111, meaning all segments are off. The remaining four digits are then displayed normally. This process creates the blinking effect for user time programming.

3 System Design

The procedure of programming the FPGA board was simple and allowed for fast debugging. Using a Digilent USB cable, the board was able to interface with a laptop. This allowed for programming the .bit file to the board and running the program, with any variable or register values able to be printed to the SDK's console window for easy debugging, value-checking, etc.

3.1 The Development Board

The choice to use the Digilent Nexys 4 DDR was made because of its wide array of GPIO. Having the seven-segment display built into the board allowed for easy control over the peripheral. Had it been necessary to interface with an external display through PMOD or some other bus, the general process for handling output would have been much longer or may have required additional IPs and libraries.

3.2 Software Environments

Vivado 2016.2 was used to create the block design for the system, as well as create the HDL wrapper, bitstream, and SDK files for the project. This process was both smooth and intuitive, which makes sense as Xilinx will be moving towards dropping support for the Xilinx EDK in favor of Vivado.

When the Vivado design was complete, the SDK files were exported for use with Xilinx SDK 2016.2. Creating an application using the generated board support package allowed access to a plethora of libraries for using the Nexys. All of the coding for this project was done in C, and was spread across two .c and two .h files for organizational purposes. The final line count spread across all four files came to 604 lines (including comments and line breaks).

4 Results and Analysis

4.1 Accuracy

The accuracy of the timekeeping for the project was tested using Google's built-in timer[iii], and multiple trials were performed. The project kept time with Google very accurately, which was not surprising, as care was taken when coding to avoid any code-segments that would significantly bog down the AXI timer. The only deviation between the project's clock and Google's was when programming mode occurred, as the board's timer is paused. This was expected, as users could have potentially experienced issues when trying to set a time if the time was not paused.

4.2 Overall Functionality

This project functions as intended, and is fairly easy to use once the user understands how to input information to the device. The only glaring issue with the project is that once the alarm has been triggered, the RGB LED with continue to flash until the board is reset or reprogrammed. This is something that could easily be fixed in future iterations of the project by designing a snooze button component.

5 Summary and Conclusion

In conclusion, the Nexys 4 DDR provided a great hardware platform for this project. It possessed all the GPIO peripherals needed for the I/O, and interfaced smoothly with Xilinx SDK 2016.2. The entire development of this project was around two weeks, although about half of that time was wasted or scrapped due to evolutions in the objectives for the project.

There are many areas in which this project could be improved and built upon, however with the project deadline it was impossible to make any of these changes.

6 References

[i] Nexys 4 DDR - Getting Started with Microblaze Servers, https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-getting-started-with-microblaze-servers/start

[ii] Nexys 4 DDR Reference Manual, https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual

[iii] Google timer and stopwatch, https://www.google.com/search?q=timer