N/A

Embedded Systems Using FPGA

Multifunction Digital Clock - Report

Student ID: 106033467

Abstract

This report contains the design, implementation and testing elements of developing a non-trivial embedded system – a multifunction digital clock. The XUPV5-LX110T FPGA Development board will serve as a platform for investigating and implementing the embedded system design.

Contents

1Introduction

1 Aims and Objectives

- Implement a digital multifunction clock.
- Software design should make use of interrupts.
- Learn about Hardware/Software design decisions
- Consider Reliability of system

2 User Specification

The features of the multifunction digital clock:

Digital Clock • Set time (hh:mm:ss) • 24/12 hr clock format • Alarm status (on/off)	**Alarm Function** • Alarm on/off • Set Alarm (hh:mm) • Indication: LED and Buzzer • Snooze
Stopwatch • Millisecond granularity • Start • Stop	**Timer** • Set timer value • On/Off • Expiry indication – LED and Buzzer

Table 1.1 - Digital Multifunction Clock Specification

3 Functional and Operational Specification

Figure 1.1 shows the basic operational and functional aspects of the system.

- I/O peripherals on the board used – pushbuttons, dip switches, leds, piezo buzzers, GPIO, Watchdog timers and the LCD panel.

- The LCD display is simple, uncluttered and easily readable by the user. It is different for each screen, and the first row shows the user which screen/function is currently selected. Each screen will show information only related to that particular function.

- Consistency is Key to this design; therefore the navigation/display/interface should be intuitive and simple.

- For example the West push button is used to set (increment) the hours – in the alarm, timer and clock screens. The center button sets the minutes and the east button sets the seconds and snooze function in the alarm.

- The North/South push buttons can be used to easily switch back/forward between the screens/functions.
- The Dip switches will be used to switch on/off the alarm and timer, as well as change the formatting of the clock display (24/12 hr format)

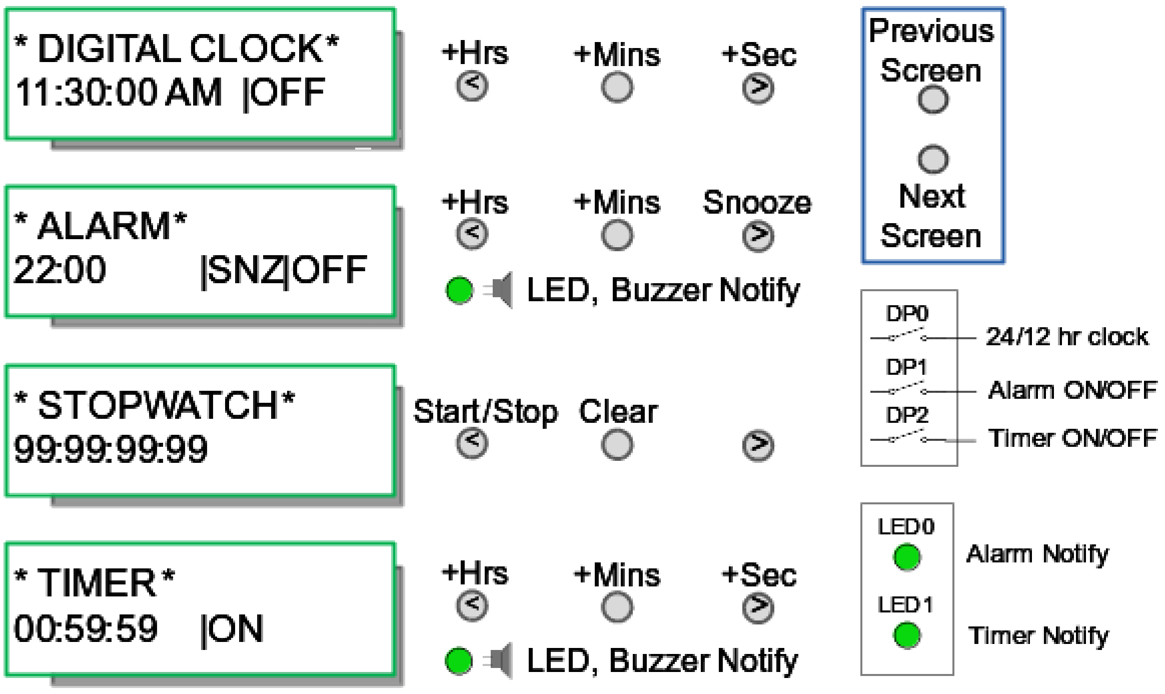


Figure 1.1 - Functional Aspects of the Multifunction Digital Clock

2 High Level Design

4 Hardware Design

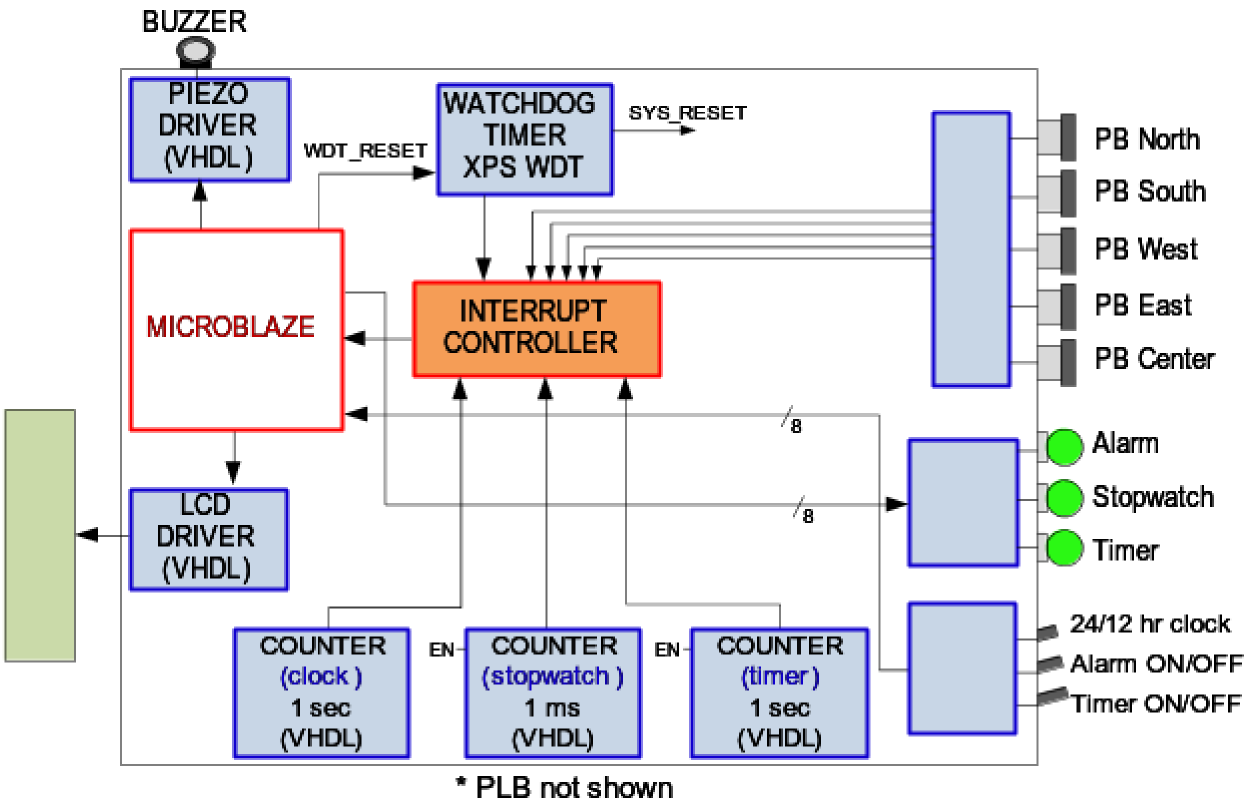


Figure 2.2 - High Level Hardware Design

1 LCD Driver

- Please refer Appendix XX for Source listing of the LCD Driver module.
- Implemented using Hardware (VHDL). Source Code written by Dr. Andy Pomfet. The LCD device can only operate in 4-bit mode, so the driver has been written as seperate two components.
  - A 4-bit VHDL driver, which is essentially a 5-state FSM. The data strobe line – E will be controlled and used to produce a busy signal for the duration given in the datasheet. Data(4bit) is loaded onto the LCD lines in the 'SETUP' state. The 'E' line on the LCD module is pulled high in the 'ACTIVE' state and low again for the 'HOLD' state.

- The 8-bit VHDL driver used contains 14 states (but only 6 are used after the device is initialised). The 8-bit driver is used to wrap up the 4-bit component and is used to above the timing constraints of the LCD.

Reasons for Choice and Implementing in Hardware:

This was a difficult design choice – as the purely Hardware version of the design consisted of multiple state machines with many states. Another design solution would have been to write the LCD driver in Assembler, making use of the PicoBlaze microcontroller (a firmware solution). Development time would have been shorter and debugging would have been easier.

However the LCD driver implemented in purely hardware is much efficient (in terms of resources it may take up) and also faster than the assembly instructions that the picoblaze will execute. And since in this instance – the VHDL code was provided to experiment with during the lab session, it was chosen to use the pure VHDL and implement the LCD driver in hardware.

2 Debounce Logic

- Please refer Appendix XX for full code listings of the Debounce peripheral implemented in VHDL.
- The debouncer design is essentially a counter, which samples the bouncy input after a given timeout period.

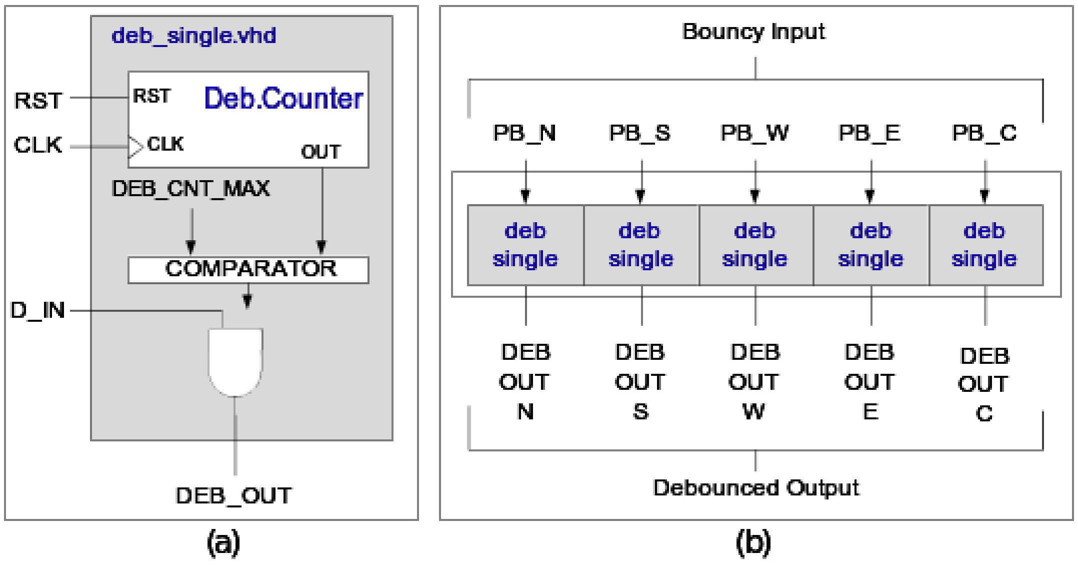


Figure 2.3 - (a) a single debounce unit (b) A single debouncer circuit implemented using an array of deb_single units

- The difficulty was to obtain a suitable value for DEB_CNT_MAX (up-counter overflow) – which varied greatly depending on the dev. board.

- Firstly a single debounce unit for a single bounce signal was implmented and ‘generated’ multiple times to design a single debouncer peripheral, able to debounce multiple bouncy signals.

Reasons for Choice and Implementing in Hardware:

Debounce can be performed using simple hardware constructs such as counters, and it seems only logical to solve the debounce issue closest to the pushbutton.

3 Counters

- The clock, timer and stopwatch features of the device are operated via interrupts.
- Two different counter types are used in the hardware design – freerunning counters and gated counters.
- Free-running counters – counts up until overflow, and then produces an output signal (this signal is connected to the interrupt controller)
- Gated counters – has an enable line which is connected to a slave register. When EN=1, the counter runs until overflow and generated an interrupt. When EN=0, the count is idle and in a RESET state.

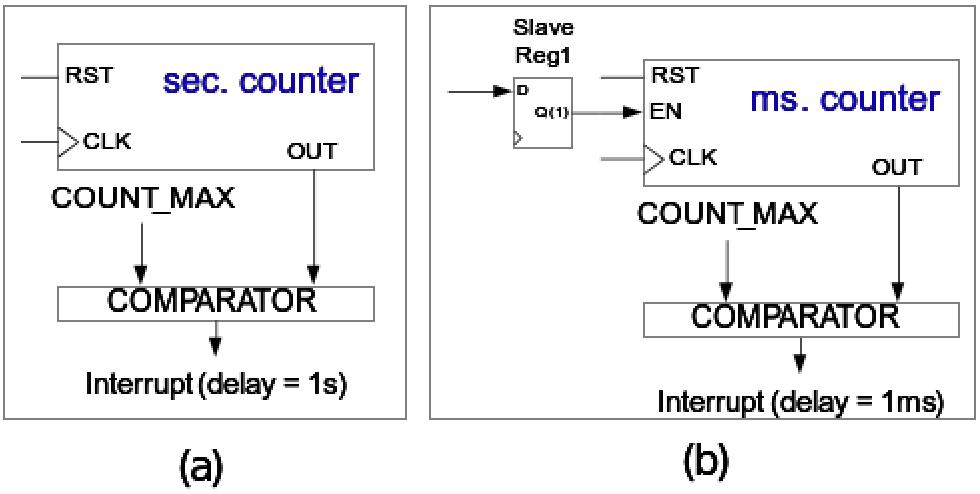


Figure 2.4 - (a) Free running counter (b) Gated – counter

- The difficulty was calculating an accurate value for COUNT_MAX (overflow) calue of the counters.
- The free running counter needed to produce an output high signal every 1 second (for the clock feature), while another feature of the embedded system needed an interrupt produced at every milisecond (i.e stopwatch feature)

Reasons for choosing Multiple Interrupts (as opposed to a single interrupt)

- Timing accuracy is critical. Interrupts should occur every 1 second, and every 1 milisecond (when needed) for the embedded device to function properly.

- Having a single interrupt driving the clock, stopwatch and timer functionality, will delay the interrupt handling and is difficult to scale/add new functions.
- By having multiple periodic, controllable interrupts – the accuracy of the stopwatch/clock/timer will be more reliable. And also the Interrupt handlers will be much faster (as they are shorter)
- Disadvantage: interrupts overlapping each other – as the number of periodic interrupts increase.

4 Piezo Buzzer Driver

- Is basically a frequency generator (clock divider)
- A simple counter – easily implemented in hardware.
- Frequency : 260Hz (Tone – Middle C)
- Has an enable line (gated counter) connected to slave registers, so can switch on/off when needed.

5 GPIO Peripherals

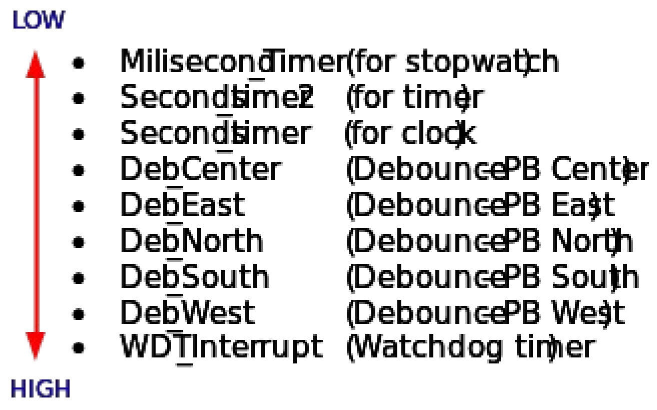
- Xilinx XPS GPIO Peripheral.
- 32-bit, and attaches itself to the PLB.
- Used in the design to read/write to peripherals such as – DipSwitches/LEDs

6 Watch Dog Timer

- Connected to the System Reset line and resets the system after a given timeout.
- Needs to be continuously refreshed (stopped from overflowing)

7 Microblaze and Interrupt Controller

- Microblaze processor - heart of the embedded device. Executes the Software components.
- The Interrupt controller (XPS Interrupt Controller) – detects the various interrupts generated by the peripherals and relays them to the Microblaze. Able to manage multiple interrupt sources.
- Each interrupt is assigned a priority within the Interrupt Controller. Asynchronous interrupts are given higher priority – to improve system responsiveness.



Milisecond_Timer	(for stopwatch)
Seconds_Timer2	(for timer)
Seconds_Timer	(for clock)
Deb_Center	(Debounce PB Center)
Deb_East	(Debounce PB East)
Deb_North	(Debounce PB North)
Deb_South	(Debounce PB South)
Deb_West	(Debounce PB West)
WDT_Interrupt	(Watchdog timer)

Figure 2.5 - Interrupt Controller - Interrupt Priorities

5 Software Design

- The Software aspect of the design will make use of the Microblaze processor.
- Software will make use of the interrupt generated by the different peripherals mentioned in Section 2.
- Software will be written in C and will be used to control the overall functionality of the design. This involves a large state machine, which is best implemented in software (less complex)
- PicoBlaze microcontroller – was another software option, but was not selected, because it is relatively much complicated to code and debug in Assembler than C code.

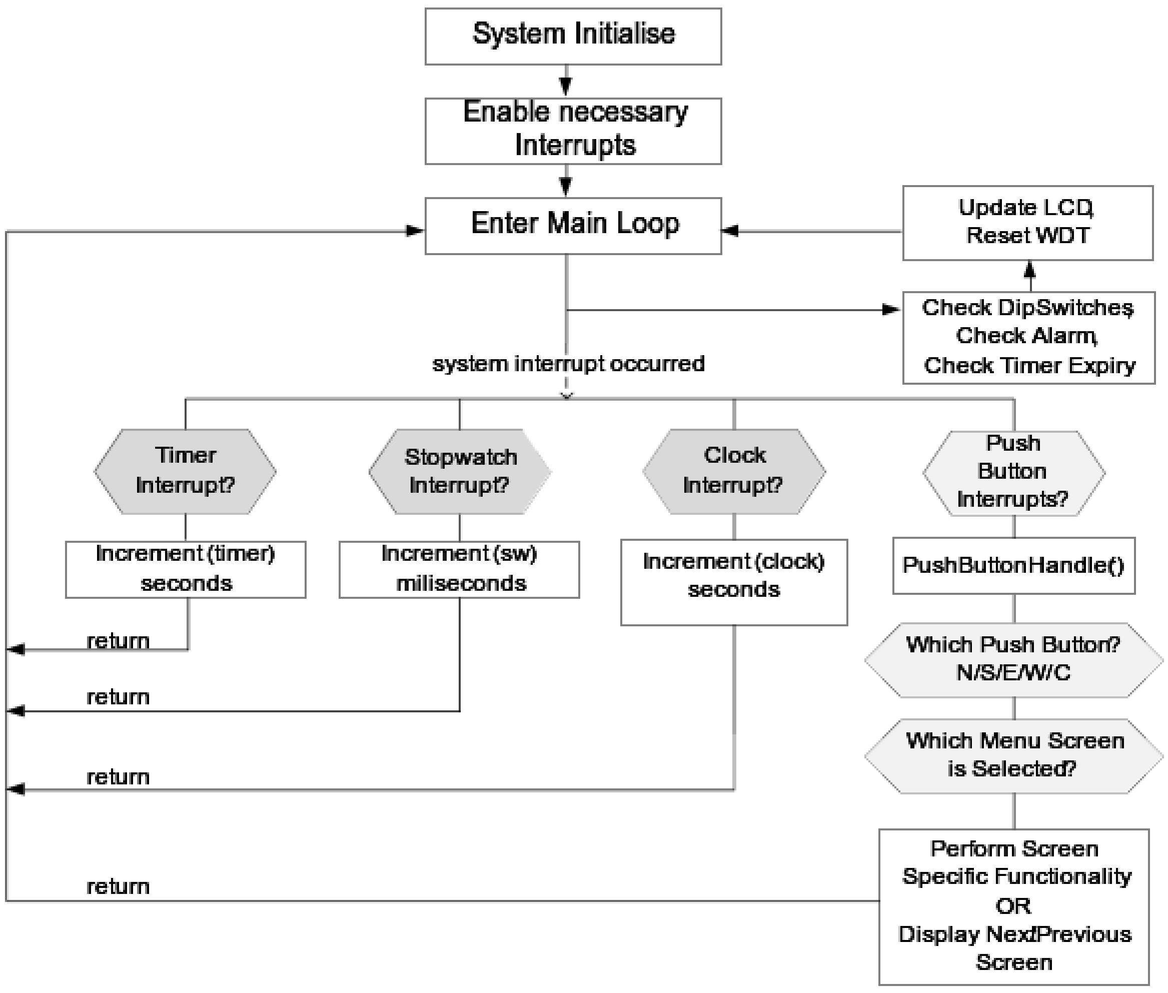

System Initialise
Enable necessary Interrupts
Enter Main Loop
Update LCD, Reset WDT
Check DipSwitches, Check Alarm, Check Timer Expiry
system interrupt occurred
Timer Interrupt?
Stopwatch Interrupt?
Clock Interrupt?
Push Button Interrupts?
Increment (timer) seconds
Increment (sw) miliseconds
Increment (clock) seconds
PushButtonHandle()
Which Push Button? N/S/E/W/C
Which Menu Screen is Selected?
Perform Screen Specific Functionality OR Display Next/Previous Screen
return
return
return
return

Figure 2.6 - High level Software Flow

6 Modular Approach

- The functions of the system was implemented in seperate modules (.c and .h files)
    - Eg: clock.c, clock.h, alarm.c, alarm.h, stopwatch.c, stopwatch.h, menu.c, menu.h etc.
- Separate Interrupt handlers for each interrupt (thus keeping the interrupt handling more organised and fast)
- This approach makes it easier to add/remove functionality. More organised – easier to fix bugs, and more readable.
- Getters/Setters/Increment which manipulate the local variables. For example, the seconds value for the clock, timer and stopwatch will be managed by independent variables, which can be manipulated by functions of that particular c file.
- Figure 3.1 shows a high level design of the software flow.
- <u>Disadvantage:</u> overall footprint of the code (memory used) will be more.

1 Getter/Setter Functions

Getter: Returns a local variable, that is local to that particular C file.

Setter: Used to set a local variable to a particular value. The new value is passed into the function as an argument.

2 Increment/Decrement Functions

Increment: adds +1 to the local variable. It is important that this function also updates other related variables accordingly (eg: CLOCK_IncrementSecs(), used to increment the seconds variable of the clock by +1, the mins and hours local variables will be incremented accordingly).

Decrementer: deducts -1 from the local variable. All related variables should be updated accordingly. (eg: TIMER_DecrementSecs(), decrements the seconds value, and also updates the mins and hours variables accordingly)

3 Get Display Functions

Used to create the character array (string) that will be displayed on the LCD panel. Currently implemented for 16 ascii characters. Integer values are converted to char, by taking their corresponding ascii values. (eg: CLOCK_GetDisplayString()). These functions will handle any other notifications displayed on the respective screen (eg: ALARM_GetDisplayString() shows '|SNZ' if snooze was pressed)

4 Switching ON/OFF LED

Bit Masking is used to get the LED GPIO peripheral values, that will switch ON/OFF the respective LED (8 lights).

5 Alarm Functional Flow

The Alarm and Alarm Snooze functions needed a proper design flow, to ensure minimum amount of bugs and ease of implementation. Since it involved certain conditions (at which point does the alarm need to be disabled/enabled?), a flow diagram needed to be drawn.

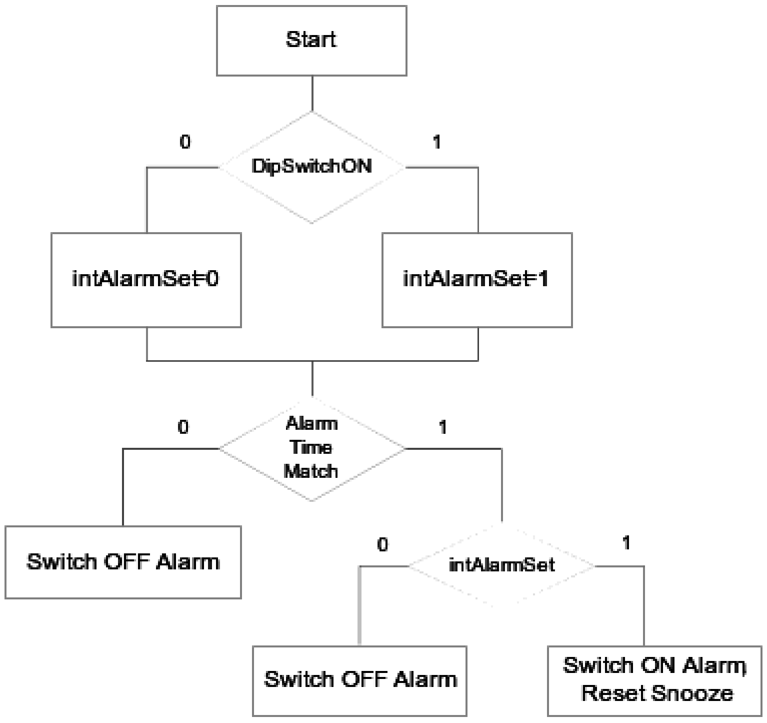


Figure 2.7 - Alarm Notification Switch ON/OFF Design Flow

Snooze Functionality

If snooze is pressed by the user, the alarm time is incremented by 'ALARM_SNOOZE_DELAY_MINS', which is a constant value that can be set within the code. Thereby - basically extending the alarm expiry time.

7 Future Extensions to the Design

- Clock: Ability to set and display the date/month/year
- Alarm: Low power mode. Multiple Alarms
- Timer: Different sound from the peizzo buzzer when timer has reached expiry. Reset timer back to original value after overflow occurs)
- Display: show notifications of the alarm, timer, stopwatch status across all screens (will take up more screen space)

3 Conclusion and Summary

All aims set in this assignment has been completed successfully accept for the “Low Power” mode function.

Difficulties arose, when trying to obtain a close enough accurate value for the seconds and millisecond interrupts – still slightly slower than required.

Testing showed a few bugs, which have been fixed – related to the Alarm Snooze functionality.

4References

- Dr. A. Pomfret, “Embedded Systems using FPGAs” Lab Scripts, University of York, Electronics Department, 2011
- B. W. Kernighan, D. Ritchie, “The C Programming Language”, Prentice Hall, 1988.

5 Appendix

8 Functional Testing

First each separate functional unit (eg: clock.c, alarm.c, stopwatch.c etc.) was tested, then all components were integrated and tested simultaneously.

1 Digital Clock Testing

- Incrementing seconds using East Push Button: check if minutes + hours update accordingly.
- Turn ON/OFF 24/12 hour formatting Dip Switch – check if display updated accordingly.
- Measure Clock accuracy using a real a stopwatch.
- Test what happens when the East+West push buttons are pressed simultaneously.

Results

- All tests were successful – except that the clock timing accuracy was not correct – this is because the Seconds Timer (counter) Max value was not set correctly (relative to the PLB clock speed).

2 Alarm Function Testing

- Increment minute’s value (using Center button) – check if the Minutes + Hours get updated.
- Set Alarm, Set clock time – check if alarm notification (led+buzzer) is active when Alarm Time=Clock Time.
- Check if alarm notifications are switched off when dipswitch is turned off.
- Test Snooze functionality
- Reset alarm again and repeat test process.

Results

- All tests passed, except snooze functionality – when snooze button is pressed, the alarm is switched off, and after X minutes the alarm reoccurs, but turning off dipswitch does not switch off the alarm. Further investigation into the ‘logic flow’ of the operation of the snooze helped to fix the logical problem.

3 Stopwatch Function Testing

- Check if Start/Stop button works as expected (millisecond timer interrupt enable/disable correctly).

- Check if the stopwatch accuracy was correct, by comparing against a known working stopwatch.
- Clear button – resets the stop watch to it’s original state ’00:00:00:00’.

Results

- On occasion – when the clear button is pressed without stopping the stopwatch the milliseconds value would be ’00:00:00:01’, which means that the interrupt occurs once, after the stopwatch has been cleared. This bug was later fixed by changing the Clear() routine.
- Stopwatch timing wasn’t perfect – slow by about 20-30 milliseconds.

4 Timer Function Testing

- Test Start/Stop Timer, works as expected.
- Test if able to set an initial Timer value as expected.
- Timer values decrement as expected, when active.

Results

- All functions operated as expected, but the timer did not reset itself to the original value when the overflow occurred.

5 Overall Integrated Testing

- Test concurrent use of the interrupts – switch on stopwatch, timer and clock and see if the system will still cope.
- Set alarm and set timer check if both LEDs get turned on at expiry and if there are any conflicts with the buzzer.

Results

- All tests resulted as expected. The Piezzo buzzer did not sound different, which does not differentiate between the alarm notification and timer notification.

9 Reliability Considerations

Most of the reliability issues were considered from a software aspect. Since the software design was relatively more complicated than the simple hardware components used.

1 Fault Avoidance

- Variables and are validated before using them.
- Invalid/Unknown states inside ‘case’ statements are checked and handled (debug warning written out and execute default state)
- Peripheral statuses are checked before using them.

2 Handling Deadlocks

- Effort has been made to make the PushButton Interrupts Mututally exclusive.
- Semaphore flags (Boolean) is tested as soon as a PushButton interrupt is entered and set when exiting the routine.
- This is to avoid corrupting the local variables.

3 Fault Recovery

- The **XPS Watchdog timer** is used – with a timeout of about 16 seconds (2^30).
- When the watchdog timer expires, the entire system resets and starts in the initial state (eg: clock is 12:00:00)
- WDT is reset, from overflowing every time the WD_Timer interrupt occurs.