DESIGN AND CODE OPTIMIZATION WHEN WORKING WITH TIME-CRITICAL DATA

Betina-Adriana TUREAC

"Transilvania" University of Brasov, Romania (betina-adriana.tureac@student.unitbv.ro)

DOI: 10.19062/1842-9238.2022.20.1.4

Abstract: *FPGA modeling techniques for DUT's are currently the state-of-the-art in the testing and verification industry for ASIC designs and with a growing interest in smart house solutions, a quick time to market is a benefit for any company. The system presented in this paper consists of an FPGA-based multifunctional digital clock with a temperature sensor connected to a Microblaze CPU and a control and monitoring unit consisting of a software running on a Raspberry PI 3B with an enhanced version of Linux with PreemptRT patch. The paper presents the design and testing of the system and also an analysis of the optimizations considered for the proposed system.*

Keywords: *FPGA, DUT, RaspberryPi, Linux, PreemptRT, Microblaze, Smart house.*

1. INTRODUCTION

In today's world, the interest over quality of life is becoming more and more important. This interest is placing pressure on technology companies to develop more and more efficient solutions for home automation. The current solutions, involve a growing number of sensors, actuators and CPU-s to automate a house[1] , Embedded systems have a major importance in this due to low power consumption and high configurability.

When it comes to the home comfort, probably the most important factor is the temperature. In this case, home automation solutions take this factor very seriously by offering ways to monitor the ambient temperature and control the methods of generating heat or air flow. The sensors used to monitor the environment parameters like temperature, humidity or gasses, can be most of the time hidden in typical household electronic devices, mostly due to esthetics, like digital clocks. The modern digital clock can hold a variety of sensors and can display the values but can also contain a Bluetooth, WiFi or RF transmitter to send the data to central units for processing. While the array of sensors located in different parts of the building provide the monitoring aspect, these central units handle data collection and control aspects, by collecting the data from the connected sensors and taking appropriate measures to improve the conditions by controlling devices in the house like valves, actuators or pumps.

Since the creation of the RaspberryPi single core platform in 2012, development of home automation applications has grown exponentially due to the low cost, modularity, open design and high processing power offered by the processors used, capable of running Operating Systems. The RaspberryPi boards are taking the place of the central units while also being capable of holding sensors on their own.

In [1], the authors propose and interesting collection of aspects that are not present in the state-of-the-art of home automation or are simply issues still not addressed by solution

developers. These aspects are: Control mode, communication, scalability and power consumption.

In this paper, the proposed system tries to offer a partial solution to these issues by implementing a monitoring system that offers an additional use for the gadget by adding the clock aspect, designed using a FPGA with different peripherals, and using Verilog and VHDL languages to create a running, stand alone architecture used to access the data on the peripherals. Using a Raspberry Pi[4] board as the central unit, the data from the monitoring station is processed using a software application written in C, running on a real time version of Linux, making use of the OS built in functions like the scheduler and memory management.

2. DESIGN AND HARDWARE IMPLEMENTATION

The proposed system is composed of a modern digital clock and a central processing unit. The clock system is implemented on a Digilent Nexys 3 FPGA board to which is connected the MCP 9808 temperature sensor. The display used is a VMOD TFT from Digilent, with a 480×272 native resolution, connected to the VHDCI FPGA port. The central processing unit is represented by a Raspberry Pi 3B board containing a 64-bit Broadcom BCM2837 CPU.

The architecture on the FPGA[5] is mainly composed of a Microblaze CPU, connected via AXI bus to an I2C[9] master module for the temperature sensor, a time control module used to control the values for the clock itself and a serial control module for the communication.

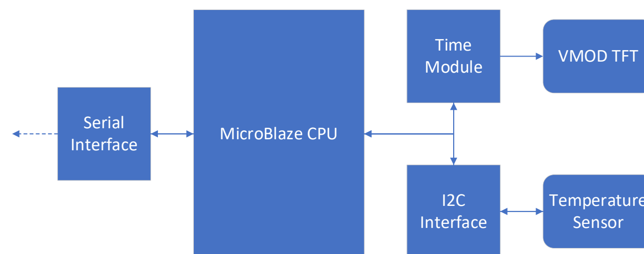

FIG. 1 Block diagram of the FPGA Architecture.

The Microblaze processor is running its own application which is capable of taking commands over the serial interface and acting on them, either to write or to read registers via the AXI Bus. In fact, the software is capable to read the temperature from the temperature sensor and set the time and date on the time control module.

The RaspberryPi is connected to the FPGA using a USB to MicroUSB cable. The Raspberry Pi[2] is running a modified version of Linux, enhanced with the Preempt-RT[10] patch for real time applications. On the operating system, the control application runs a series of BIST type tasks to verify the correct functionality of the peripherals and then launches threads for data collection, processing and control.

For the proposed system, as the controlled device, a LED has been attached to the Raspberry Pi pins which in this case will be active when the temperature from the sensor passes a certain threshold. Other peripherals may be added like a buzzer for the clock alarm which is not a part of the smart house concept so it will not be controlled externally but is an integral part of the clock. On the other hand, the alarm can be set as needed by the control unit.

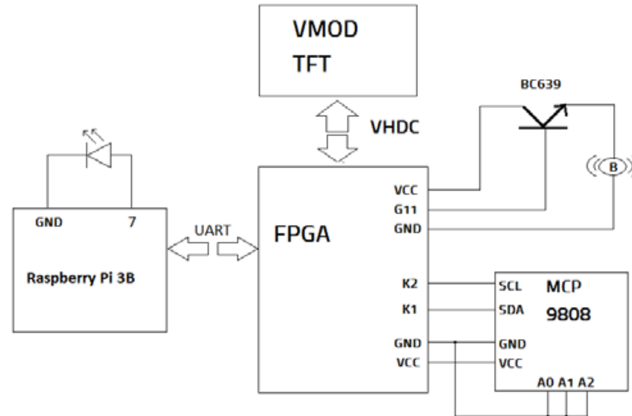

FIG. 2 General diagram for the proposed system.

3. SOFTWARE IMPLEMENTATION

The software running on the MicroBlaze processor is minimized to be as fast as possible, acting like a switch between the peripherals and the control unit. The serial interface is scanned and data is present from the control unit, the decoded instruction is compared against the set of instructions contained. For simplification purposes, the instruction is delivered as a lower-case letter. This is to separate the instructions from the numeric values transmitted to the time module in order to set the clock, the alarm or the current date. There are different letters used for every action, including setting or getting a time value. In this case, a similar module may contain 26 instructions and is capable of expanding with another 26 instructions if upper-case letters are also used.

```
read = read_uart_lock();
if (read == 'a') {
    a = read_uart_lock();
    ceas[0] = a;
}
if (read == 'z') {
    a = ceas[0];
    write_uart_u(a);
}
if (read == 's') {
    b = read_uart_lock();
    ceas[1] = b;
}
```

FIG. 3 Example of Microblaze code

The hardware must always have the values at hand in case of a processor access. In this case, all sensor modules must be implemented using interface registers. Only by accessing these registers, the processor communicates with the modules and must not be permitted to act on the internal values of the modules. In Fig. 3, the read variable receives the value from UART as an instruction. In this example, letter “a” is used to signal that the control unit is requesting to set the value of the hour value on the clock. The software understands the instruction and waits for the numeric value from the control unit after which it is written to the access register of the time module. In case the control unit requests the current time, for synchronization porpoises for example, the letter “z” is used to get the value from the access registers and sent to the control unit. The table of instructions continues for every parameter that the control unit can access.

The software written for the FPGA MicroBlaze processor[7] contains 2 functions, both related to the communication over UART. The read function is blocking the flow and waits for data inside the function. This is the most efficient way to address the instructions sent by the control unit since the processor does not have any other duty. The function is used every time the processor is done with an instruction and waits for the next one, or based on a set instruction, it waits for the numeric value.

The main function contains the instruction table inside an infinite loop but before this, on startup, the software sets the default time and date on the clock screen as 11.00 25.10.2020, and the alarm as 9.00. This process can also be considered a BIST since on startup, a visual inspection can be done by an observer.

In the case of the application running on the control unit, the software is more complex, having to deal with all the sensors and control devices. The smart house[3] general idea is that data is periodically read from the sensors, processed, and based on the processing result, a control action is, or not, taken. In this case, we can determine 3 phases of the process:

-Data collection (DC)

-Data processing (DP)

-Control action (CA)

When dealing with cause-and-effect actions where action A leads to action B, and most importantly, action A must occur periodically, parallelism is considered the best course of action since there is no way to determine when action B is finalized or if there are any actions after B.

In the case of processors, parallelism is represented by the use of application threads[6]. Using threads for each action is crucial for a correct functionality of the entire system. Of course, the main disadvantage of using threads is when a large number of threads is created and can stall the processor. This is the system designer attribution, to check the optimal time between launching two thread series.

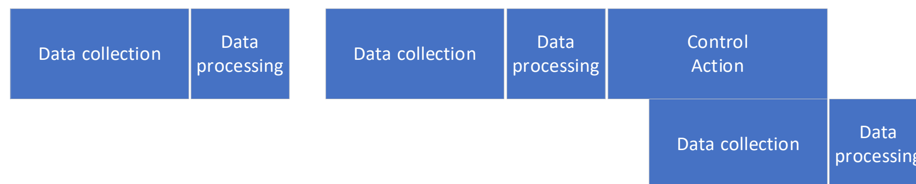

FIG. 4 Thread series example.

In Fig. 4, the collected data from the second thread series generates a control action, but in the same time, the periodical data collection is launched for a new series. In this case the scheduler controls what is occupying the processor at a certain time. In this way, the processor will complete both tasks.

The software running on the control unit for the proposed system contains the main function which starts up with a BIST portion of code which is taking advantage of the built-in WiFi capability of the Raspberry Pi. Once the device itself starts up, it searches for an internet connection over LAN or wireless and if successful, sets its internal clock and date from the internet. Taking advantage of this, the first thing the control unit will do is to set the time and date on the clock using the SET instructions and requests the temperature as a first value to verify the connectivity. In the same time, the thread that handles the control device (Thread 3) is started for a visual inspection by an observer. After the BIST is finished, the first thread is started as the main thread for the application.

Thread 1 handles the serial communication with the sensor. At a specified interval, the thread launches a request for the temperature value and once received, the value is passed through a queue to the second thread. Thread 1 is the Data collection thread and is controlling the timing of when data is requested from the monitoring device.

Thread 2 receives data from the queue and comparing it against a hardcoded threshold value. Based on the result of the comparison, thread 3 is either launched or not. If launched, thread 3 will send a PWM signal lasting 5 seconds, to the LED with a frequency of 1 second. The process is repeated until the temperature from the sensor is below the threshold value.

FIG. 5 Thread series for the proposed system

It is still unclear the optimal thread series process and how is should occur or how many threads can run at one point in time until the processor stalls.

4. EFFICIENCY IMPROVEMENT

Scenario 1: Data collection thread is running in an infinite loop and the related data processing thread is launched when the data is available. This is similar to the diagram in Fig. 4. For every monitoring device (MD) there must be a thread series. This kind of architecture may be used when:

-there is a small number of monitoring devices

-the time between requesting data from the devices is bigger.

The only potential benefit in this case is an ease of scalability since for every new device a thread series is added to the code, but the issues with this architecture are substantial. In first place, a minimum number of threads are running on the CPU equal to the number of MDs connected. Furthermore, accessing the same communication medium by 2 different threads may result in an access conflict. Lastly, if the data collection threads launch their own data processing thread, in a worst-case scenario, the load on the CPU would be close to 2x the number of MDs. In the case of a small control unit with limited computing power, this is an issue.

Scenario 2: One way to ease the load on the CPU is to have a general data processing thread. The initial load is the number of MDs +1. The communication between the DC and DP may be done by queue. DC threads are pushing data in the queue and DP thread will treat each data and launch the appropriate CA thread. In a worst-case scenario, the load would be 2x MDs +1, in the case of all existing CA threads are launching close to the same time, depending on the length of all CA threads and the processing speed in the DC thread. The benefit of implementing such an architecture is having more control over the back-end of the application, but this complicates the scalability having to add the DC and CA threads for a new MD while complicating the CA to integrate the new function. Also, the queue may contain the data from the DC thread but having multiple DC threads, the main DP will not know the source DC. A signaling system must also be implemented complicating even more the scalability. This scenario is represented in Fig. 5.

Scenario 3: To improve on the benefits from Scenario 2 to ease the load would be to also control the front end of the application.

The software will launch a master DP thread that will, in turn, control the moment each DC thread is launched. In this way, the issue of the source of the data is solved if the DP launches one DP thread at a time, collects the data. It and then launches a new DC thread and at the same time, launching a response CA thread. Having this kind of control is very beneficial for the load aspect since the initial load is of one single thread and the worst-case scenario is three threads. This scenario is most optimal for central units with low processing power, but still maintaining scalability. Since each MD is requested data in a serial mode, there is no possible overlap on the communication devices and no heavy load on the CPU. The drawback of this architecture is actually what draws in the benefits. The request on each MD being done in a serial manner, with a large number of MDs, the time between accessing the same MD rises but the issues presented, from [1] of scalability, communication and control are solved but the architecture may not be efficient on a large scale.

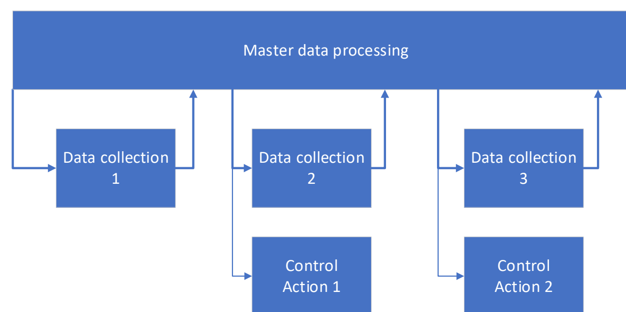

FIG. 6. Thread series for scenario 3 architecture

Scenario 4: The remaining efficiency issue may be resolved by a combination of the first and third scenarios. Implementing a single Master thread that controls all thread series.

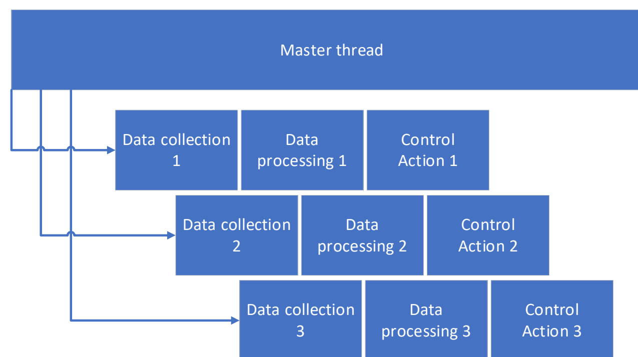

FIG. 7 Thread series for scenario 4 architecture

Analyzing this scenario, CPU load in a worst case scenario is the number of MDs +1, being the Master thread. This value of load is much higher than scenario 3 but still lower than the other scenarios, thus solving the issue of balancing efficiency against the possibility of stalling the CPU. Having control directly from the Master thread, each thread series can be launched to avoid same communication channel overlap. Scalability is solved by retaining the simplified variant of implementing the 3 threads[8] as independent functions and only modifying the master thread to allocate each new MD a time slot to be launched in. In addition, a more important MD may receive more time slots in a specified time than less important MDs.

5. EXPERIMENTAL RESULTS

As mentioned in this paper, an experimental design has been created to test these hypotheses. The main focus is to test the timing difference between Scenario 2 and 4 since these versions of the architectures can be considered the most un-optimized and optimized versions.

Starting with the architecture proposed at scenario 2, where threads 1 and 2 are launched at the start of the application and only thread 3 is conditioned we get the time values presented in Table 1.

Table 1. Average execution time for scenario 2

T1 exec. Time + 5s	T2 Q empty + 2s	T2 temp < th.	T2 temp > th.	T3 exec. Time + 5s	T1 start to T3 start
13551 us	64 us	2 us	636 us	496 us	1947662 – 936386 us

While these are the values under normal functionality, some anomalies have been observed.

1. In some cases, the time reported from start to finish for T3 is reported as being under 1 second. This is practically impossible since T3 has an average functionality time of 5000496 us. The cause is because in special cases, “T1 start to T3 start” is smaller than T3 start to T3 finish. In these cases, a second T3 is launched before the first T3 is finished.

2. The large values of time reported for “T1 start to T3 start” are due to T1 and T2 not being synchronized.

For the implementation of the architecture for scenario 4, a more simplified technique is applied as mentioned, by writing the independent threads as stand-alone functions. In this scenario, only the Master thread is launched in a loop and every 6 seconds, T1 is started. T2 is launched only when the data is received by T1. In this manner, the memory is freed and the synchronization issue between T1 and T2 is solved. In addition, a more precise time line can be observed thus preventing an unnecessary overlap of two T3 threads.

The time values obtained from the test run can be observed in table 2.

Table 2. Average execution time for scenario 4

T1 exec. Time	T2 Q empty	T2 temp < th.	T2 temp > th.	T3 exec. Time + 5s	T1 start to T3 start
11080 us	N/A	21 us	559 us	367 us	18319 - 9658 us

First observation is that there is no execution time for when the Queue is empty between T2 and T1 since T1 launches T2 only when there is a value available in the Queue. Secondly, most of the values are significantly smaller since the threads do not have internal delays and are terminated once the tasks are done. A massive increase in efficiency can be seen in the “T1 start to T3 start” value since there is a synchronization. No anomalies have been observed during the test.

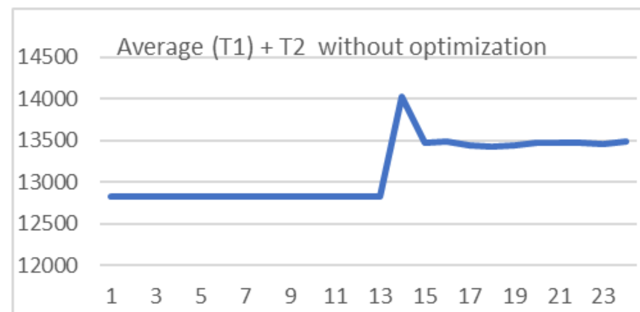

FIG.8 Timing values on unoptimized architecture.

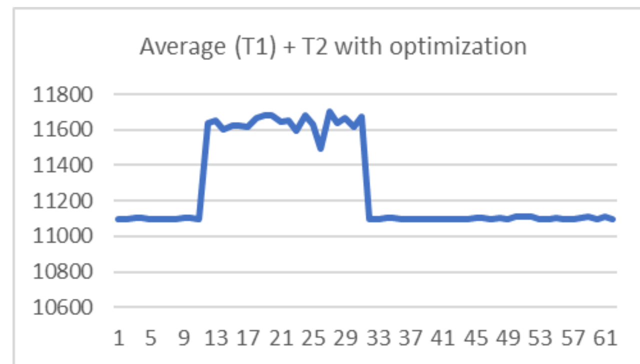

FIG.9 Timing values on optimized architecture.

Charts 1 and 2 contain the timing values collected during running. In both cases, the higher values coincide with the moments where high temperature values are received. Timing analysis suggests a decrease in general timing even when T3 is launched by 2 ms.

6. CONCLUSIONS

From this paper, we can conclude a well-defined architecture for a smart-house system and an efficient software for the control unit that is acceptable for different levels of processing power regardless of how many sensors or control devices are present. This presented software architecture solves all of the objectives planned.

REFERENCES

[1] M. Reaz, A. Assim, F. Choong, M. Hussain, and F. Mohd-Yasin, "*Proto-typing of smart home: A multiagent approach*,"WSEAS Transactions on Signal Processing, vol. 2, no. 5, pp. 805–810, 2006.
[2] N. Petrov, D. Dobrilovic, M. Kavali ́c, and S. Stanisavljev, "*Examples of raspberry pi usage in internet of things*," 2016.
[3] O.-V. Rusu and A.-V. Duka, "*Monitoring and control platform for homes based on fpga, soc and web technologies*,"Procedia Engineering, vol.181, pp. 588–595, 2017.
[4] PEREIRA, Renata IS, et al. *IoT embedded linux system based on Raspberry Pi applied to real-time cloud monitoring of a decentralized photovoltaic plant*. Measurement, 2018, 114: 286-297.
[5] M. Marufuzzaman, M. B. I. Reaz, and M. T. Islam, "*Fpga based distributed task organizing agents in smart home*," pp. 1–5, 2014.
[6] LEE, Edward A. *The problem with threads.* Computer, 2006, 39.5: 33-42.
[7] CHOI, Jongsok; BROWN, Stephen; ANDERSON, Jason. *From software threads to parallel hardware in high-level synthesis for FPGAs*. In: 2013 International Conference on Field-Programmable Technology (FPT). IEEE, 2013. p. 270-277.
[8] MOK, Aloysius K.; CHEN, Deji. *A multiframe model for real-time tasks. IEEE transactions on Software Engineering*, 1997, 23.10: 635-645.

[9] MANKAR, Jayant, et al. *Review of I2C protocol. International Journal of Research in Advent Technology*, 2014, 2.1.

[10] DE OLIVEIRA, Daniel Bristot; DE OLIVEIRA, Romulo Silva. *Timing analysis of the PREEMPT RT Linux kernel. Software:* Practice and Experience, 2016, 46.6: 789-819.