CSEC 793 Capstone in Computing Security
Project Report

OPTIMIZING HARDWARE ARCHITECTURE IN HYBRID CHECKPOINTING DESIGNS FOR INTERMITTENT DEVICES

April 10, 2023

Christabelle Alvares
Department of Computing Security
College of Computing and Information Sciences
Rochester Institute of Technology
cda5542@rit.edu

1 Abstract

The work in this project discusses the challenges of implementing checkpointing in batteryless devices, which rely on intermittent power sources that may cause a shutdown mid-computation. The current literature on the topic is reviewed, highlighting the advantages and drawbacks of various approaches, including security features, hardware solutions and software checkpointing. This project proposes a hybrid approach that supplements the CPU with hardware architecture to support efficient volatile memory checkpointing. This hardware-optimized checkpoint scheme conserves time and energy for the actual checkpoint process and includes a mechanism to trigger checkpoint generation based on a threshold value of memory usage. The proposed design is flexible and is compatible with any software checkpointing that can offer further security for the device. The experimentation phase tested the design's effectiveness against a baseline checkpointing solution and includes measurement of various metrics such as hardware cost and energy efficiency. The results of this project conclude that a hybrid checkpointing scheme with both hardware and software solutions is more efficient for secure checkpointing in intermittent devices.

2 Introduction

Batteryless devices in the form of sensors and microcontrollers are emerging technologies in the IoT industry. They are cheaper, more eco-friendly than their battery-powered counterparts and can be deployed in the most remote places without requiring routine maintenance. These devices rely on power sources that may be sporadic and the likelihood of energy exhaustion causing a shutdown mid-computation is high. The concept of checkpointing is implemented in these intermittent devices to preserve the state of the device prior to energy depletion and to resume operations at a later time when enough energy has been harvested. Certain security features must be added to the checkpoint solution to guarantee that the device is not susceptible to attacks on confidentiality, integrity or availability.

There are different approaches to solving the problem of unsecured checkpoints in current literature. Certain approaches look to cryptography techniques to ensure confidentiality yet introduce time and energy overhead while requiring additional hardware support. Other hardware approaches involve the addition of memory protection units (MPUs) or tamper-free non-volatile memory hardware components to address confidentiality without the need for cryptography. On the other hand, the majority of research focuses on software checkpointing solutions that display higher effectiveness than prior hardware implementations. Even fewer approaches explore a hybrid scheme - a design that relies on a hardware/software co-design.

Despite their efficacy, existing software approaches suffer from several drawbacks:

- Reliance on software to address security issues with checkpointing comes with higher overhead. Efforts to reduce the overhead have been successful but it is yet to be

determined if they meet the required benchmark based on legitimate uses in the industry.

- Software solutions may require trust in the user program which could be insecure and a threat to the checkpointing software.
- With this solution, checkpointing the dynamically allocated areas of memory is difficult as the software has limited access to those portions of memory. A more optimized approach than simply replicating volatile memory to non-volatile memory during checkpoint generation should exist.

The above arguments call for a hybrid system to checkpoint security. In this project, a secure, hardware-optimized checkpoint scheme for intermittent computing was designed to address more efficient checkpointing for the volatile RAM. The design intends for flexible security as it is compatible with any other software checkpointing, a benefit for industry use.

[Metrics and Results will be added to this paragraph]

3 Background

Energy harvesting techniques have advanced far enough that low energy-consuming, wireless devices can be sufficiently powered to run computations and transmissions as necessary. Various sources of energy such as wind, sound, vibration, solar energy and radio frequency (RF) are being considered in the designs of these technologies [1]. Although wind and sound are uncommon sources for energy harvesting devices, vibration energy is capable of powering sensors in fields, for example. A newer venture, RF energy harvesting is the method of utilizing either ambient or produced electromagnetic waves for power generation [2].

Today, several industries require sensing and embedded technologies in their operations. They are deployed to monitor infrastructure in factories, in autonomous vehicles, in spacecrafts and satellites, to observe and examine wildlife, etc. [3]. Recently, these self-powered devices have been explored even in the wearable technology market [4], their ability to solve energy supply obstacles in a sustainable manner proving to be a more flexible and environment-friendly solution. The most noteworthy application of sensors is in the field of Internet of Things (IoT), whose devices can be deployed for general or highly specific needs. The past decade has seen the rapid development of IoT and a main concern of researchers is sustainability, leading to a proliferation of studies in batteryless IoT devices.

Although batteryless devices have proven to be beneficial in many processes, they are subject to circumstances in the surrounding environment. Battery-free products require effective power management, but the frequency of intermittent program execution is high due to power loss. At this point, the device powers off and seeks to harvest enough energy

to effectively continue the computation. This necessitates a method to save the efforts accomplished by the device before running out of power and into the energy-harvesting state. Central to the entire discipline of intermittent computing is the concept of checkpoints [5, 6, 7, 8, 9, 10, 11, 12, 13] to deal with the loss of program state and erasing of volatile memory during power-disruptive events. When these devices gain power, the checkpoint restoration process will reinstate the system and memory based on the last generated checkpoint. Checkpoints prevent batteryless devices from having to restart an application or computation if power is lost, saving processing time and energy in an efficient manner.

However, an unsecured checkpoint system can be vulnerable to attacks on the confidentiality and integrity of data, thereby causing interference in the correct operation of batteryless devices. Examples of attacks are checkpoint snooping, spoofing and replay, as described by Krishnan and Schaumont [14]. Apart from attacks, unintentional programmer errors can also cause the system to malfunction and corrupt checkpoints. As such, securing the checkpointing process must ensure the integrity of the checkpoint, the confidentiality of data in memory and the efficacy of checkpointing in terms of optimizing memory space and energy consumption.

4 Related Work

Recent studies focus on certain critical aspects of checkpointing - data confidentiality, the integrity of the checkpoint and reduction of software or hardware overhead, depending on the approach of the implementation. Furthermore, differences in static and just-in-time checkpointing also play a part in the designs of these implementations. Static or manual checkpointing involves predetermined locations in the program to complete checkpoint generation. This technique helps the checkpointing process avoid interruption of access by peripherals. Time-based or just-in-time checkpointing [11] starts checkpointing based on the results of a power-monitoring mechanism, only checkpointing with impending power failure. This technique helps avoids unnecessary checkpointing as opposed to the static method, conserving time and energy, and removing the requirement of programming checkpoints. General characteristics of solutions in literature have been determined as shown in Figure 1.

Among hardware-based and hybrid approaches, the work in [15], [16], [17] and [10] target the security issues of data confidentiality with their checkpointing process. An early approach by Ghodsi, Garg and Karri [15] uses the conventional checkpointing scheme - the state of volatile memory is copied into a shadow portion of the non-volatile memory - with added encryption/decryption using "using a light-weight block cipher PRINCE 11". The design requires hardware support for the encryption keys. Some disadvantages of this system include the cryptography overhead, the lack of checkpoint integrity and the risk of chosen plaintext attacks on the key after a physical capture of the device.

Similarly, the work by Suslowicz et al. [16] attempted to improve on the cryptographic

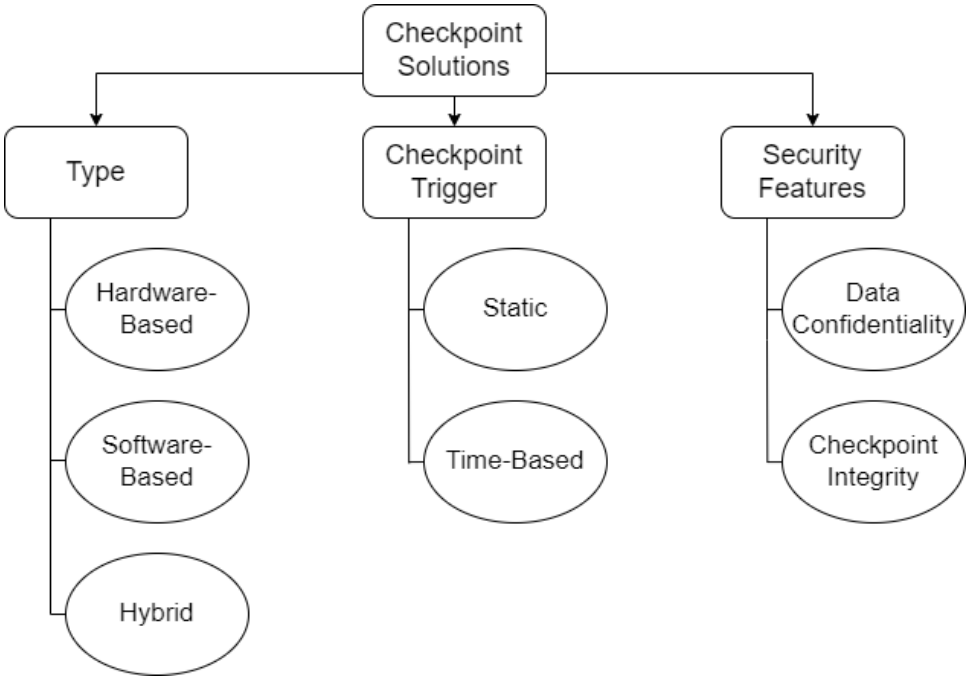

Figure 1: Characteristics of current checkpoint solutions in the literature

theme by developing 2 designs - a hardware-accelerated AES scheme and a software-based implementation of a lightweight cryptographic protocol. Suslowicz et al. highlighted the need for checkpoint integrity, authenticity and freshness, where [15] failed. However, their designs have the same shortcomings as the work by Ghodsi, Garg and Karri [15]. Another hybrid approach by Hardin et al. [17] proposes a memory isolation system for an ultra-low-power microcontroller using the minimal protections of the MPU along with a compiler that restricts access to memory portions based on the user program. The compiler inserts address bounds and then verifies access by the program to memory addresses. However, this work is not compatible with intermittent computing devices and the MPUs themselves are not sufficient to fully protect the RAM as they are subject to misconfigurations with repetitive power failures.

As discussed above, all these schemes [15, 16, 17] are vulnerable to memory corruption and remote-based software attacks. Dinu, Krishnan and Schaumont [10] designed a secure intermittent architecture (SIA) with no hardware modifications, equipped with self-attestation and remote attestation features. However, similar to the work by Hardin et al. [17], SIA's [10] implementation is dependent on the intermittent microcontrollers having MPUs and therefore exclude devices that are poorer in hardware security. This hybrid approach is additionally vulnerable to the same problems as [17].

Research into software solutions for checkpointing has grown at an increased rate in the past decade. Early work in checkpoint designs such as Mementos [5], [6], Hibernus++, [7], HarvOS [8], [9], DICE [18], [11], [19], [12] focus on improving the efficiency of checkpointing but are insecure, i.e. they do not protect data confidentiality and checkpoint integrity. Recently, a fully software-based solution was proposed by Grisafi et al. [13] which ensures both these security features. MPI (Memory Protection for Intermittent Computing) [13] uses a hypervisor for lightweight memory protection based on the trusted computing module (TCM). The implementation works without cryptographic mechanisms or hardware modifications. MPI also employs a compiler toolchain that rewrites and instruments the user program based on the memory protection policy, similar to the compiler explored by Hardin et al. [17]. This solution also differs from other secure checkpoint approaches by removing trust in the user programs or any assumptions that leave the device vulnerable to software-based attacks. Comparing MPI to the hybrid schemes using common metrics, the time and energy consumption is much lower than encryption-based schemes but is very close to those dependent on partial MPU protection. The MPI scheme trades in security for performance as the runtime overhead is significant. This motivates the idea that an optimized hybrid approach could be more efficient.

Existing hybrid and hardware approaches currently do not achieve the security features or performance to the same extent as that of software solutions. It can be observed from analyzing the current literature that a holistic view of checkpoint security is lacking. Based on this examination, the following are the contributions of this project:

- This project develops a clean-slate approach to checkpointing by supplementing the CPU with architecture that supports efficient volatile memory checkpointing.
- This reduces the software overhead with a more optimized checkpointing system for volatile memory, specifically the heap or the dynamically allocated memory spaces of the RAM.
- The hardware designed in this project can track used memory portions or blocks better, conserving time and energy for the actual checkpoint process.
- The design also includes a mechanism to trigger checkpoint generation based on a determined threshold of memory usage which can be utilized to accurately decide when to checkpoint.
- This hardware approach fits into a hybrid checkpointing scheme which is flexible and allows general or purpose-specific software solutions to work with higher optimization.

5 Project Description

As discussed in previous sections, current solutions in the literature trade in performance for security or vice-versa. This project idea focuses on exploring a hardware-software

co-design that involves optimizing checkpointing the volatile RAM. Prior work has focused on improving security but inadvertently add software overhead or hardware support for cryptographic implementations that bloat the checkpoint solution and make it unfeasible. The overall goal of this project was to design, implement and examine the benefits of a hybrid checkpoint model that can take advantage of the security provided by software solutions and the performance boost of added hardware.

In the memory checkpointing process, it is evident that copying the entire contents of RAM into non-volatile memory was not effective. The requirements of this method are not feasible - the size of non-volatile memory needs to be big enough to fit a sizable portion of RAM during checkpoint generation. To optimize this process, the proposed solution considers slicing volatile memory into '**N**' number of blocks. A table saved in external peripheral memory keeps track of which blocks of the memory have been written to after the device has been booted or reset. For every block in memory, there is a corresponding flag in the table. This table, called '**D_Table**' in this design, is implemented as an array of '**N**' bits.

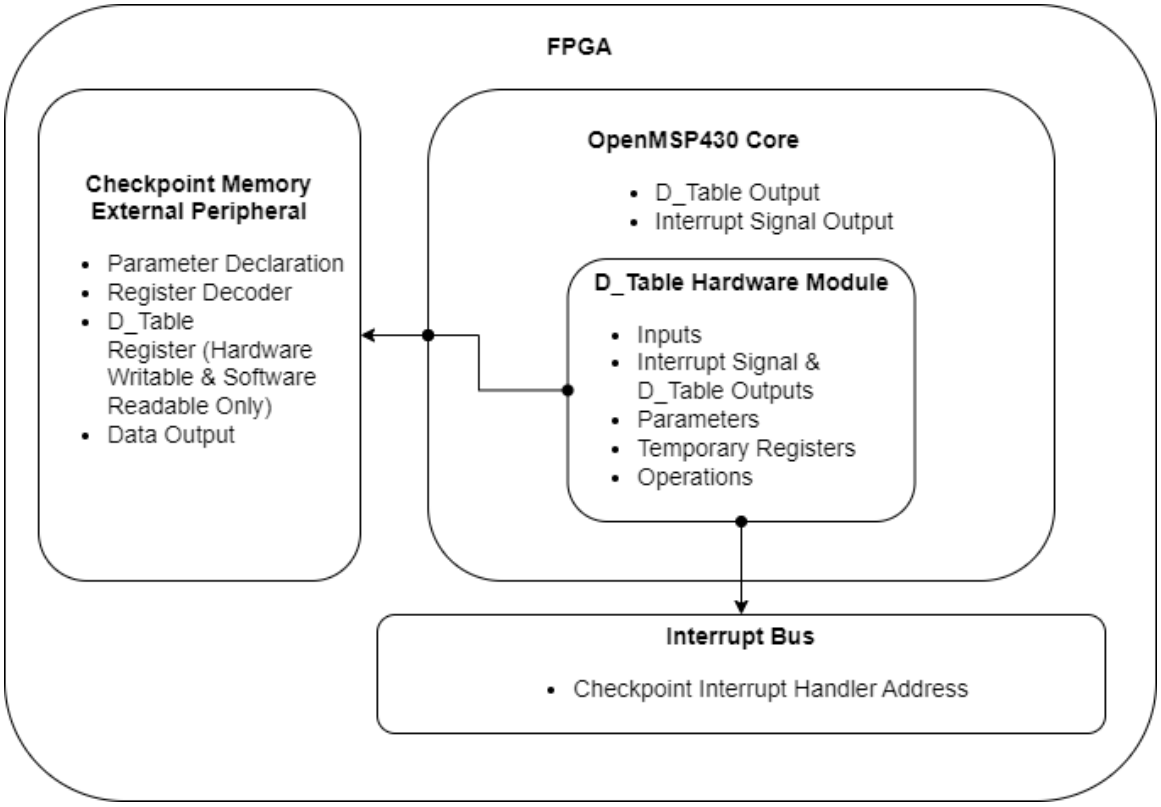

Figure 2: High-Level Hardware Design of the Solution

Figure 2 shows the high-level design of the solution for the MSP430 microcontroller based on OpenMSP430 open-source microcontroller core [20]. The figure shows the additions made to the core to accommodate the new operations needed to implement the '**D_Table**' module, as well as the introduction of an external peripheral for the actual '**D_Table**' array of '**N**' bits to be stored in non-volatile memory. This external peripheral holds '**D_Table**', which can only be written by hardware and is read-only by software for security purposes to preserve the integrity of the table.

The '**D_Table**' module also triggers an interrupt to begin the checkpoint generation step when a certain threshold is passed. This threshold value is the maximum count of blocks that have been written to, which must be copied during checkpointing. The value can be predetermined like a static trigger or perhaps calculated depending on the amount of energy available to the device to successfully checkpoint, like a time-based trigger. This offers flexibility to the user to choose whichever trigger method they would like to be configured.

Once set, the interrupt signal in the interrupt bus interrupts the program and executes the corresponding function handler. This function will begin the checkpoint generation and shutdown process of the device. In the next section, the details of the design are described in detail along with visuals.

6 Project Implementation

In this section, details about the design and implementation are described, as well as the challenges faced during the process.

6.1 Hardware Design & Implementation

This solution was implemented using the OpenMSP430 [20] open-source microcontroller core written in Verilog (a hardware description language) using the software Vivado [21], in which the designed hardware was synthesized and analyzed. Figure 3 is a visual representation of the architecture implemented in Vivado for the '**D_Table**' module, as mentioned in the previous section. The actual Verilog modules and modifications to the original OpenMSP430 modules can be found in the Appendix.

Figure 3 shows the connections, operations, inputs and outputs in the '**D_Table**' module. As shown in the figure, the module takes 5 inputs - clock, program counter (PC), reset bit, memory address (MA) and a memory write (MW) bit indicating if the memory address is being written to at a certain clock signal. All displayed operations occur at the positive edge of the clock input; this is not shown in the figure to help readability.

There are 3 configurable parameters that are set at boot:

- Block Size Shift (BSS) - The BSS value is associated with '**N**', the number of blocks that volatile memory has been split into and consequently the number of bits in

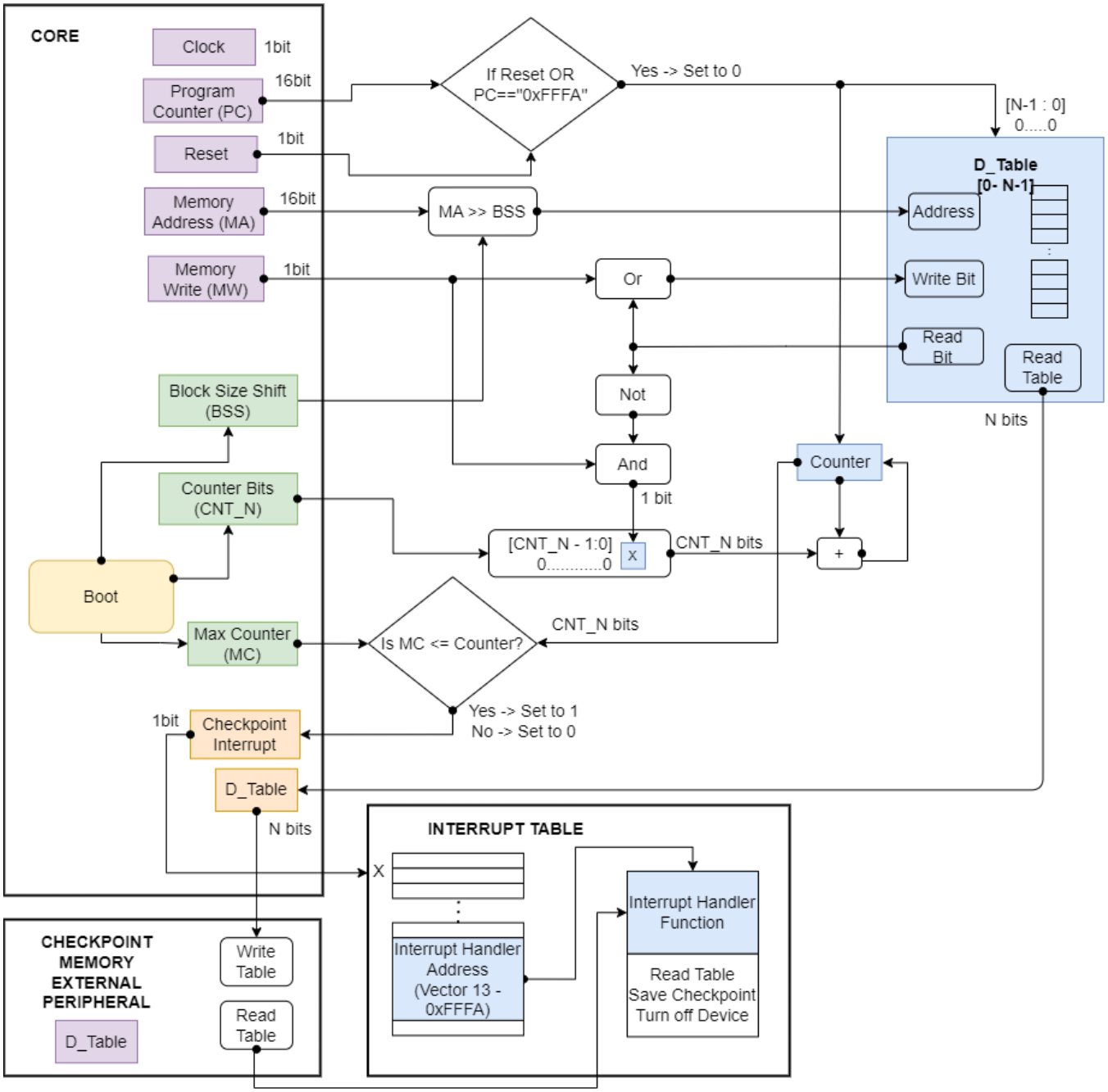

Figure 3: Visual Representation of the Hardware Architecture

'**D_Table**'. '**N**' is 2 to the power of BSS, a natural number. In this design, '**D_Table**' needs enough values in its array to keep track of all the blocks in memory.

$$N = 2^{BSS}$$

.

- Max Counter (MC) - MC is the count of blocks in '**D_Table**' that when reached sets the interrupt signal to 1. The threshold value that triggers the interrupt signalling the checkpoint generation process.

- Counter Bits (CNT_N) - This value is the number of bits required to represent the number '**N**' in binary.

The '**D_Table**' module has multiple internal registers used for temporary values for the different operations. The main registers are the temporary '**D_Table**' and the counter, shown in blue in Figure 3. There are 2 outputs of the module - '**D_Table**' table and the interrupt signal.

The following points describe the main operations in the module:

- Writing to the '**D_Table**' Register: To find the corresponding flag in the '**D_Table**' register that should be set for writing, the memory address (MA) is shifted right the number BSS times. This value gives the index in the '**D_Table**' register. Additionally, this flag is only set to 1 if the memory write (MW) is also 1, as shown with the OR operator in the figure.

- Incrementing the Counter: Figure 3 demonstrates that the counter only increments if the memory address is the first address in the block to be written to. This means that the corresponding bit for the block in '**D_Table**' has been set to 1 for the first time.

- Resetting the Registers: The registers for the temporary '**D_Table**' and the counter are set to 0 bits when the reset signal is set. This also happens right after the interrupt function has begun executing. This second scenario is tracked based on the specific program counter "0xFFFA" which is the address associated with the function handler.

- Triggering the Interrupt: When the value in the Counter register reaches the MC value, the interrupt signal is set to 1. The interrupt bus receives the signal and the following PC is set to the address of the interrupt function handler.

The '**D_Table**' output from the module is sent to the external peripheral as shown in Figure 3. The table in the peripheral is only writable by hardware. It has read only access by software for security purposes.

6.2 Challenges Faced During the Design & Implementation Process

The following are some of the challenges faced:

- Software programming is very different from designing hardware in HDL (hardware description language). It is very easy to fall into the trap of designing hardware using programming logic. Hardware requires a different line of thinking and has its own best practices. For example, when you can simply divide an integer by 2 in code, in hardware using shift registers could be more appropriate. The design process must factor in feasibility and hardware cost, using the least costly gates, etc.
- While implementing the design in Vivado, a previous version of the architecture did not account for resetting the Counter register and so the interrupt kept interrupting itself endlessly without executing the function. This took a while to debug but it was resolved upon clearing the the register after the interrupt had been triggered.

7 Testing and Experiments

7.1 Experiments on Performance & Computing Metrics

[Results will be added post-completion]

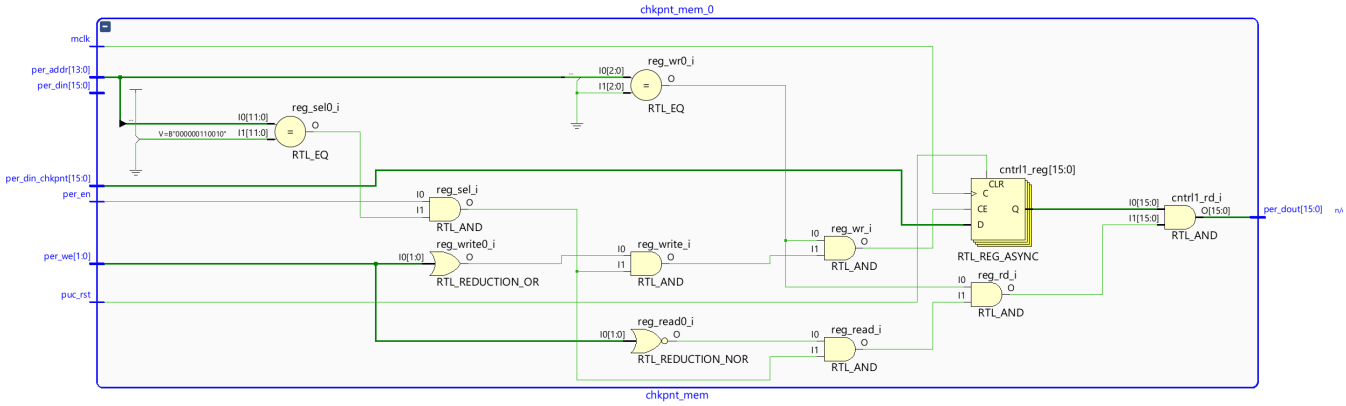

Figure 4: Schematic of the External Peripheral

8 Conclusion

This project addressed various checkpointing solutions and examined their advantages and drawbacks. A hybrid checkpoint model was proposed by designing an optimization for

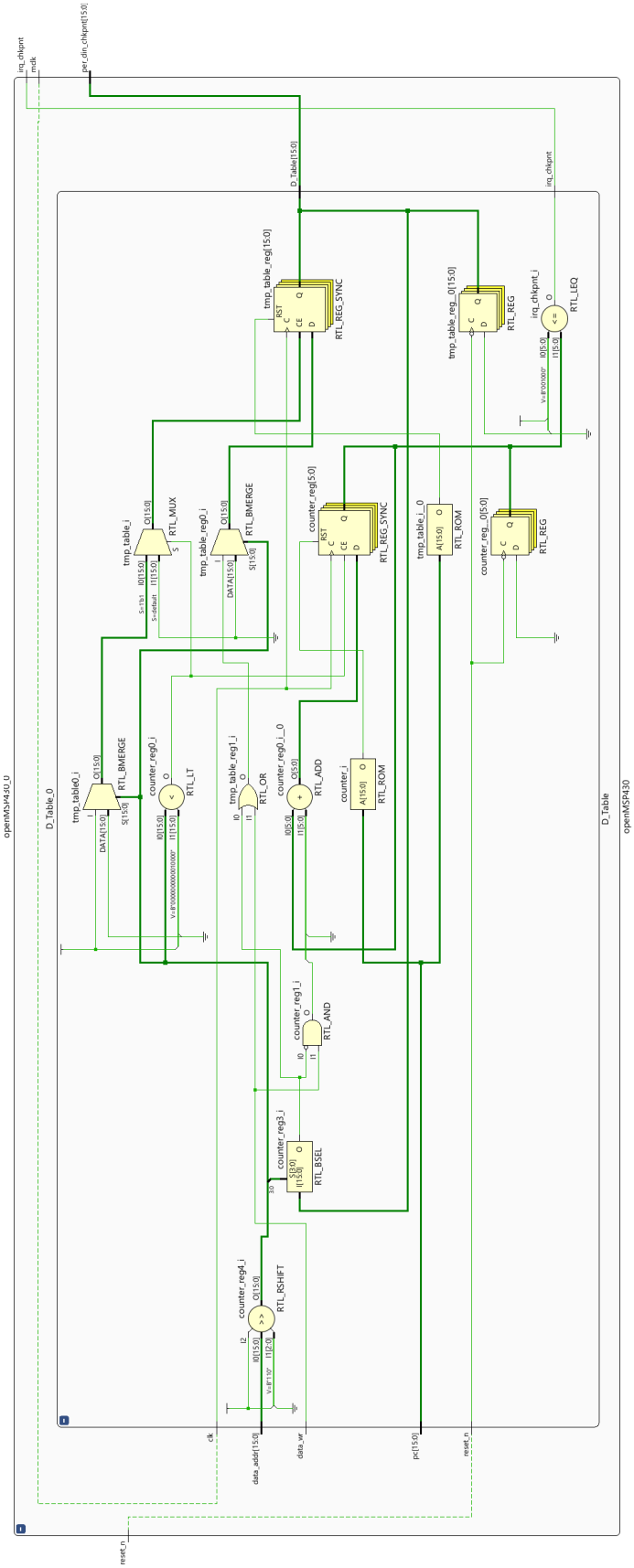

Figure 5: Schematic of the D_Table Module

hardware architecture to supplement a secure software solution. The goal of this clean-slate method was to more efficiently track volatile memory usage to reduce the work during the checkpoint generation phase. [Results]

In the experimentation phase, a basic software checkpointing solution was programmed as the baseline. The proposed design was used in conjunction with that software checkpointing to form a hybrid solution. For future work, this optimized design could be tested along with leading solutions in the literature. It would be interesting to investigate if the design proposed in this project can assist in reducing the overheads introduced by these software solutions. Furthermore, the security of the resulting hybrid solution can be examined as well. Overall, the security, efficiency and feasibility of the hybrid model will determine the viability of the solution and progress the research in checkpointing for intermittent devices. Additionally, future work in the area could include research into more effectively determining the threshold value based on estimating power supply and energy consumption.

9 Acknowledgment

I would like to thank my Capstone course advisor Dr Sumita Mishra and Capstone advisor Dr Ivan De Oliveira Nunes for their guidance, direction and helping me to finalize this Capstone project. I'd also like to extend a special thanks to Antonio Joia Neto and Adam Caulfield for their advice and tips on checkpointing, Vivado and the best practices of hardware design. I appreciate my family for financing my education and supporting me throughout my studies.

References

[1] S. Chalasani and J. M. Conrad, "A survey of energy harvesting sources for embedded systems," in *IEEE SoutheastCon 2008*, 2008, pp. 442–447.

[2] T. Soyata, L. Copeland, and W. Heinzelman, "Rf energy harvesting for embedded systems: A survey of tradeoffs and methodology," *IEEE Circuits and Systems Magazine*, vol. 16, no. 1, pp. 22–57, 2016.

[3] J. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3131672.3131699

[4] C. Xu, Y. Song, M. Han, and H. Zhang, "Portable and wearable self-powered systems based on emerging energy harvesting technology," *Microsystems & Nanoengineering*, vol. 7, no. 1, p. 25, 2021.

[5] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 159–170.

[6] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 575–585, 2015.

[7] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[8] N. A. Bhatti and L. Mottola, "Harvos: Efficient code instrumentation for transiently-powered embedded sensing," in *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2017, pp. 209–219.

[9] G. Luan, Y. Bai, C. Wang, J. Zeng, and Q. Chen, "An efficient checkpoint and recovery mechanism for real-time embedded systems," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*. IEEE, 2018, pp. 824–831.

[10] D. Dinu, A. S. Khrishnan, and P. Schaumont, "Sia: Secure intermittent architecture for off-the-shelf resource-constrained microcontrollers," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 208–217.

[11] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1101–1116.

[12] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 85–99.

[13] M. Grisafi, M. Ammar, K. S. Yildirim, and B. Crispo, "Mpi: Memory protection for intermittent computing," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 3597–3610, 2022.

[14] A. S. Krishnan and P. Schaumont, "Exploiting security vulnerabilities in intermittent computing," in *Security, Privacy, and Applied Cryptography Engineering: 8th International Conference, SPACE 2018, Kanpur, India, December 15-19, 2018, Proceedings 8*. Springer, 2018, pp. 104–124.

[15] Z. Ghodsi, S. Garg, and R. Karri, "Optimal checkpointing for secure intermittently-powered iot devices," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 376–383.

[16] C. Suslowicz, A. S. Krishnan, D. Dinu, and P. Schaumont, "Secure application continuity in intermittent systems," in *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, 2018, pp. 1–8.

[17] T. Hardin, R. Scott, P. Proctor, J. Hester, J. Sorber, and D. Kotz, "Application memory isolation on ultra-low-power mcus," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 127–132.

[18] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Efficient intermittent computing with differential checkpointing," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019, pp. 70–81.

[19] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention." in *OSDI*, 2016, pp. 17–32.

[20] Olgirard, "Olgirard/openmsp430: The openmsp430 is a synthesizable 16bit microcontroller core written in verilog." [Online]. Available: https://github.com/olgirard/openmsp430

[21] "Vivado ml overview." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

10 Appendix

This appendix contains original files added to the Vivado OpenMSP430 project for this project's hardware for optimized checkpointing. These original files are D_Table.v and chkpnt_mem.v.

10.1 D_Table.v

```
//-----------------------------------------------------------------------------
//
// *File Name: D_Table.v
//
// *Module Description: Module for Dirty bit Table or D_Table
//
//
```

```
// *Author(s): Christabelle Alvares
// Advisors: Dr. Ivan De Oliveira Nunes, Antonio Joia Neto, Adam Caulfield
//
//-------------------------------------------------------------------------
// $Rev: ?
// $LastChangedDate: 4th April 2023
//-------------------------------------------------------------------------

`include "openMSP430_defines.v"

module D_Table(
    //INPUTS
clk,
data_addr,
data_wr,
reset_n,
pc,
//max_counter, ? Q: max counter as input or parameter?

//OUTPUTS
D_Table,
irq_chkpnt
);

// PARAMETERs
//============
parameter CNTR_MSB   =  6;         // MSB of the address bus
parameter BLK_SIZE   =  64;       // Memory size in bytes
parameter BLK_SIZE_SHIFT = 6;      // Constant for shift instead of division by 64
parameter [5:0] MAX_COUNTER = 8;         // THRESHOLD VALUE triggers the interrupt

// OUTPUTs
//============
// ? Size of counter&temp changes based on # of blocks
// output [5:0] counter;
output [15:0] D_Table;
output irq_chkpnt;

// INPUTs
//============
```

```
// ? [15:0]?
input [15:0] data_addr;       // RAM address
input clk;        // RAM clock
input data_wr;        // RAM write enable (low active)
input reset_n;
input [15:0] pc;

// RAM
//============

reg [15:0] i;

// ? Size of D_Table/tmp_table depends on Dmem size
reg [15:0] tmp_table;
reg [5:0] counter;

initial begin
    counter = 0;
    i = 0;
    tmp_table = 0;
end
always @(negedge reset_n)
    begin
       counter = 0;
       tmp_table=0;
    end
always @(posedge clk)
    begin
      if(pc==16'hFFFA)
       begin
        counter=0;
        tmp_table=0;
       end else begin
          i = data_addr >> BLK_SIZE_SHIFT;
          if(i< 16) // ADDED TEMPORARILY to address only fewer mem blocks
          begin
            counter = counter + {{CNTR_MSB-1{1'b0}},{((~tmp_table[i]) & data_wr)}};
            tmp_table[i] = tmp_table[i] | data_wr;
         end
      end
    end
```

```
assign irq_chkpnt = (MAX_COUNTER<=counter);
assign D_Table = tmp_table;
endmodule
```

10.2 chkpnt_mem.v

```
    //------------------------------------------------------------------------
// Copyright (C) 2009 , Olivier Girard
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//     * Redistributions of source code must retain the above copyright
//       notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above copyright
//       notice, this list of conditions and the following disclaimer in the
//       documentation and/or other materials provided with the distribution.
//     * Neither the name of the authors nor the names of its contributors
//       may be used to endorse or promote products derived from this software
//       without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
// OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
// THE POSSIBILITY OF SUCH DAMAGE
//
//------------------------------------------------------------------------
// *File Name: chkpnt_mem.v

// *Original File Name: template_periph_16b.v
//
// *Module Description:
//                       16 bit peripheral for D_Table memory.
```

```
//
// *Author(s):
//              - Olivier Girard,    olgirard@gmail.com
//
//------------------------------------------------------------------------------
// $Rev$
// $LastChangedBy$
// $LastChangedDate$
//------------------------------------------------------------------------------

module chkpnt_mem(

// OUTPUTs
    per_dout,                       // Peripheral data output

// INPUTs
    mclk,                           // Main system clock
    per_addr,                       // Peripheral address
    per_din,                        // Peripheral data input
    per_en,                         // Peripheral enable (high active)
    per_we,                         // Peripheral write enable (high active)
    per_din_chkpnt,
    puc_rst                         // Main system reset
);

// OUTPUTs
//=========
output       [15:0] per_dout;       // Peripheral data output

// INPUTs
//=========
input               mclk;           // Main system clock
input        [13:0] per_addr;       // Peripheral address
input        [15:0] per_din;        // Peripheral data input
input               per_en;         // Peripheral enable (high active)
input         [1:0] per_we;         // Peripheral write enable (high active)
input        [15:0] per_din_chkpnt;
input               puc_rst;        // Main system reset


//=============================================================================
```

```
// 1)  PARAMETER DECLARATION
//============================================================================

// Register base address (must be aligned to decoder bit width)
parameter       [14:0] BASE_ADDR   = 15'h0190;

// Decoder bit width (defines how many bits are considered for address decoding)
parameter              DEC_WD      =  3;

// Register addresses offset
parameter [DEC_WD-1:0] D_Table      = 'h0;
                       //CNTRL2      = 'h2,
                       //CNTRL3      = 'h4,
                       //CNTRL4      = 'h6;

// Register one-hot decoder utilities
parameter              DEC_SZ      =  (1 << DEC_WD);
parameter [DEC_SZ-1:0] BASE_REG    =  {{DEC_SZ-1{1'b0}}, 1'b1};

// Register one-hot decoder
parameter [DEC_SZ-1:0] CNTRL1_D    = (BASE_REG << D_Table);
                       //CNTRL2_D    = (BASE_REG << CNTRL2),
                       //CNTRL3_D    = (BASE_REG << CNTRL3),
                       //CNTRL4_D    = (BASE_REG << CNTRL4);


//============================================================================
// 2)  REGISTER DECODER
//============================================================================

// Local register selection
wire              reg_sel   =  per_en & (per_addr[13:DEC_WD-1]==BASE_ADDR[14:DEC_WD]);

// Register local address
wire [DEC_WD-1:0] reg_addr  =  {per_addr[DEC_WD-2:0], 1'b0};

// Register address decode
wire [DEC_SZ-1:0] reg_dec   =  (CNTRL1_D  &  {DEC_SZ{(reg_addr == D_Table )}});  //|
                               //(CNTRL2_D  &  {DEC_SZ{(reg_addr == CNTRL2 )}})  |
                               //(CNTRL3_D  &  {DEC_SZ{(reg_addr == CNTRL3 )}})  |
                               //(CNTRL4_D  &  {DEC_SZ{(reg_addr == CNTRL4 )}});
```

```
// Read/Write probes
wire              reg_write =  |per_we & reg_sel;
wire              reg_read  = ~|per_we & reg_sel;

// Read/Write vectors
wire [DEC_SZ-1:0] reg_wr    = reg_dec & {DEC_SZ{reg_write}};
wire [DEC_SZ-1:0] reg_rd    = reg_dec & {DEC_SZ{reg_read}};


//============================================================================
// 3) REGISTERS
//============================================================================

// D_Table Register
//-----------------
reg  [15:0] cntrl1;

wire        cntrl1_wr = reg_wr[D_Table];

always @ (posedge mclk or posedge puc_rst)
  if (puc_rst)        cntrl1 <=  16'h0000;
  else if (cntrl1_wr) cntrl1 <=  per_din_chkpnt;


// CNTRL2 Register
//-----------------
/*
reg  [15:0] cntrl2;

wire        cntrl2_wr = reg_wr[CNTRL2];

always @ (posedge mclk or posedge puc_rst)
  if (puc_rst)        cntrl2 <=  16'h0000;
  else if (cntrl2_wr) cntrl2 <=  per_din;


// CNTRL3 Register
//-----------------
reg  [15:0] cntrl3;
```

```
wire        cntrl3_wr = reg_wr[CNTRL3];

always @ (posedge mclk or posedge puc_rst)
  if (puc_rst)        cntrl3 <=  16'h0000;
  else if (cntrl3_wr) cntrl3 <=  per_din;


// CNTRL4 Register
//-----------------
reg  [15:0] cntrl4;

wire        cntrl4_wr = reg_wr[CNTRL4];

always @ (posedge mclk or posedge puc_rst)
  if (puc_rst)        cntrl4 <=  16'h0000;
  else if (cntrl4_wr) cntrl4 <=  per_din;
*/

//============================================================================
// 4) DATA OUTPUT GENERATION
//============================================================================

// Data output mux
wire [15:0] cntrl1_rd  = cntrl1  & {16{reg_rd[D_Table]}};
/*
wire [15:0] cntrl2_rd  = cntrl2  & {16{reg_rd[CNTRL2]}};
wire [15:0] cntrl3_rd  = cntrl3  & {16{reg_rd[CNTRL3]}};
wire [15:0] cntrl4_rd  = cntrl4  & {16{reg_rd[CNTRL4]}};
*/

wire [15:0] per_dout   =  cntrl1_rd;  //|
                          /*cntrl2_rd  |
                          cntrl3_rd  |
                          cntrl4_rd;*/


endmodule
```