Support for Accessible Bitsliced Software

Thomas J. Conroy

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

February 4, 2021

Blacksburg, Virginia

Keywords: Bitsliced Software, Code Generation, Direct Memory Access, Neural Network Acceleration

Support for Accessible Bitsliced Software

Thomas J. Conroy

(ABSTRACT)

The expectations on embedded systems have grown incredibly in recent years. Not only are there more applications for them than ever, the applications are increasingly complex, and their security is essential. To meet such demanding goals, designers and programmers are always looking for more efficient methods of computation. One technique that has gained attention over the past couple of decades is bitsliced software. In addition to high efficiency in certain situations, including block ciphers computation, it has been used in designs to resist hardware attacks. However, this technique requires both program and data to be in a specific format. This requirement makes writing bitsliced software by hand laborious and adds computational overhead to transpose the data before and after computation. This work describes a code generation tool that produces it from a higher-level description in Verilog. By supporting the synthesis of sequential circuits, this tool extends bitsliced software to parallel synchronous software. This tool is then used to implement a method for accelerating software neural network processing with reduced-precision computation on highly constrained devices. To address the data transposition overhead and to support a hardware attack-resistant architecture, a custom DMA controller is introduced that efficiently transposes the data as it transfers along with dedicated hardware for masking and redundancy generation. In combination, these tools make bitsliced software and its benefits more accessible to system designers and programmers.

Support for Accessible Bitsliced Software

Thomas J. Conroy

(GENERAL AUDIENCE ABSTRACT)

Small computers embedded in devices, such as cars, smart devices, and other electronics, face many challenges. Often, they are pushed to their limits by designers and programmers to reach acceptable levels of performance. The increasing complexity of the applications they run compounds with the need for these applications to be secure. The programmers are always looking for better, more efficient methods of doing computations. Over the past two decades bitsliced software has gained attention as a technique that can, in certain situations, be more efficient than standard software. It also has properties that make it useful for designs implementing secure software. However, writing bitsliced software by hand is a laborious task, and the data input to the software needs to be in a specific format. To make writing the software easier, a tool that generates it from the well-known Verilog hardware description language is discussed in this work. This tool is then used to implement a method to accelerate artificial intelligence calculations on highly constrained computers. A custom hardware module is also introduced to speed up the formatting of data for bitsliced processing. In combination, these tools make bitsliced software and its benefits more accessible.

to my parents and my sister

Acknowledgments

I would like to thank my advisor, Dr. Schaumont, for guiding me through this program and being a wonderful mentor and teacher. I would also like to thank the other members of my committee, Drs. Diehl and Plassmann, for supporting me in this and for being part of my educational experience going back as far as my second semester as an undergraduate.

To the members of the Secure Embedded Systems lab, thank you for being there when I needed help and for making the lab a welcoming place. To Richa Singh and Pantea Kiaei in particular, I am thankful for being able to work with you.

I would like to thank Drs. Burbey and Tront for their help and advice and for running the CyberCorps: SFS program, which has supported me greatly. I always enjoyed dropping by your offices pre-pandemic.

Finally, to the other SFS students, my friends, and everyone else who has made the past couple of years outstanding, thank you.

This material is based upon work supported by the National Science Foundation under Grant Number 1303297. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
AXI	Advanced eXtensible Interface
CPA	Correlation Power Analysis
DES	Data Encryption Standard
DFA	Differential Fault Analysis
DMA	Direct Memory Access
DPA	Differential Power Analysis
FFs	Flip-Flops
FIFO	First In, First Out
FSMD	Finite State Machine with Datapath
GPIO	General Purpose Input/Output
HDL	Hardware Description Language
IRQ	Interrupt Request
LUTs	Look-Up Tables
MAC	Multiplier-Accumulator
NIST	National Institute of Standards and Technology

NN	Neural Network
PRNG	Pseudo-Random Number Generator
PSP	Parallel Synchronous Programming
PSPCG	Parallel Synchronous Programming Code Generator
RTL	Register-Transfer Level
SIMD	Single Instruction, Multiple Data
SoC	System-on-Chip
SPN	Substitution-Permutation Network
tDMA	transposing-Direct Memory Access
XOR	Exclusive OR

Chapter 1

Introduction

With every passing year in recent memory, the unremitting march of technological progress makes more powerful computing machines available—and last year's models cheaper. When the Commodore 64 was announced in early 1982 its 64 kibibytes of memory were an upgrade over other home computers at the time. Today, it is common to own a cell phone with many thousands of times the memory capacity.[1] By modern standards, its memory alone makes the Commodore 64 ill-suited as a home computer, but it becomes even less attractive due to its $595 price (initially, in 1982 dollars) [1]. Computing machines have become cheaper and can have the resources and speed directly out of the dreams of a Commodore 64 user, but products more like it than a modern home computer did not simply cease to exist. As rose the performance-to-cost ratio of home computers, so it did for lower-cost options. In fact, products with similar performance to the Commodore 64 can be found for less than a dollar. It was no longer prohibitively expensive to embed these in other products and devices, thus becoming embedded systems.

Embedded systems are more capable than ever, and the growth of expectations for them never halts. With the recent explosion of "smart devices" becoming available and embedded systems' continually increasing ubiquity, the tasks asked of embedded systems increase in complexity. One task is deep learning neural network processing, which is usually done with expensive and powerful hardware available in contrast to the low-power, high-constraints

[1]One gibibyte is 16,384 times larger than the memory capacity of the Commodore 64

environment of embedded systems [2]. Hence, there has been great interest in the efficiency of such computing tasks [3].

Beyond resource constraints, by nature embedded systems designers and programmers must also deal with the problems arising from the technology interacting with the physical world more than other computing machines. One problem this thesis focuses upon is that bad actors can have easier physical access to embedded systems. This allows an adversary to subject the device to powerful attacks on the device's security, such as power-based side-channel analysis and fault injection attacks. Power-based side channel analysis can recover secret information, such as cryptographic keys, from a system by passively measuring and analyzing one or more traces of the power usage of the device during computation with secret data [4]. Fault attacks, on the other hand, actively cause—or "inject"—faults in the device (*i.e.*, cause incorrect results via violating hardware timing requirements) and relate the result to secret data [5]. Extensive research has explored these attacks and countermeasures to prevent the reveal of secret information. Among these countermeasures include masking [6] and intra-instruction redundancy [7]—for power side-channel analysis and fault attacks, respectively.

One technique that has been studied that supports both masking and intra-instruction redundancy in a flexible manner is bitslice programming [8]. This technique treats every bit in a processor word as a separate parallel processor via only using bitwise instructions. For example, a bitsliced program running on a 32-bit processor would be computing 32 parallel instances of the same program in lockstep.

1.1 Contributions

Though bitslice programming has its benefits, in both security and performance, nevertheless there are issues facing programmers who wish to employ the technique. First, writing bitsliced programs by hand is inefficient and time consuming. As will be explained in Chapter 2, it is at least as tedious as designing a hardware module entirely at the gate level. Second, because bitsliced programs treat every bit of a processor word separately, typical data inputted to the program must be transposed into a correct format for computation and transposed a second time afterwards. These transpositions, when done in software, can be costly and represent a considerable portion of the total computation in some cases.

The primary objective of this work is to improve the accessibility of bitslice programming by providing software and hardware tools to programmers and designers that make writing bitsliced programs easier and transposing data more efficient. To achieve this, the following contributions are made:

- To support the creation of efficient bitsliced code, the Parallel Synchronous Programming Code Generator (PSPCG) was developed. This tool accepts a Verilog hardware design as input and produces bitsliced C code as output. The bitsliced code it generates, however, can be seen as part of a new paradigm of Parallel Synchronous Programming [9] since the stateful elements of the hardware design are also synthesized into software. Bitslice programming traditionally focuses only upon representing combinational logic in software. Including support for sequential logic expands the capabilities of the generated software to include emulation of Finite State Machines with Datapath (FSMDs).
- As an application of Parallel Synchronous Programming, we investigate how PSPCG can be applied to accelerate matrix-vector multiplication in software neural networks

by utilizing the full width of the processor for reduced-width operands.

- A custom, special-purpose DMA controller is described with the capability to accelerate the conversion of data into and out of the bitsliced format as well as support countermeasures for side-channel and fault attacks. This DMA controller, the transposing-DMA (tDMA), was designed specifically to support the bitslice representations from the SKIVA architecture [8], but it can be integrated in traditional System-on-Chips (SoCs) as well. The tDMA was integrated for testing and performance measurements with both a MicroBlaze soft processor core [10] and an ARM Cortex A-9 in a Zynq-7000 SoC [11]. To show the correctness of the implementation, major parts of the controller's algorithm were formally verified using model-checking.

1.2 Organization

The rest of the thesis is organized as follows. In Chapter 2 the background of relevant topics is discussed. Afterwards, Chapter 3 gives an overview of related work found in literature. The PSPCG is discussed in Chapter 4 along with its use in implementation of fast block ciphers. Chapter 5 discusses a method of efficient matrix-vector multiplication for neural networks using software created with the PSPCG. In Chapter 6 the design, implementation, and performance of the tDMA are analyzed and discussed. Chapter 7 concludes the work with a summary of the results and high-level discussion.

Chapter 2

Background

We first discuss some preliminaries that give context to the contributions in this work. These topics include an overview of bitslice programming and its history, cryptographic block ciphers, masking, intra-instruction redundancy, and model checking.

2.1 Bitslice Programming

Bitslice programming is a technique originally introduced by Biham in 1997 [12] for creating high-performance cryptographic software. That paper describes an implementation of the Data Encryption Standard (DES) block cipher using bitslice programming that at the time was the fastest known. By computing 64 parallel instances of the cipher on a 64-bit processor, the processor is better utilized than by a standard implementation. That is, due to how DES is designed a standard implementation may operate on data smaller than 64 bits. For example, each DES S-box takes a 6-bit input and produces a 4-bit output [13]. Each S-box could be implemented as a look-up table, and the processor is being under-utilized. In bitslice programming, each of those 6 input bits are stored in 6 separate processor words, and the S-box is computed on all 64 parallel instances. Bitslice programming requires data to be represented such that the multiple bits of a data word are each located in separate processor words—called *slices*. This allows each bit of a slice to represent a different data word. A transformation from a standard representation to the bitsliced representation is shown in

Figure 2.1. This transformation is referred to as a transposition, like matrix transposition, since considering each word to be a row in a matrix and each bit a column, the transformation flips the matrix over its minor diagonal.

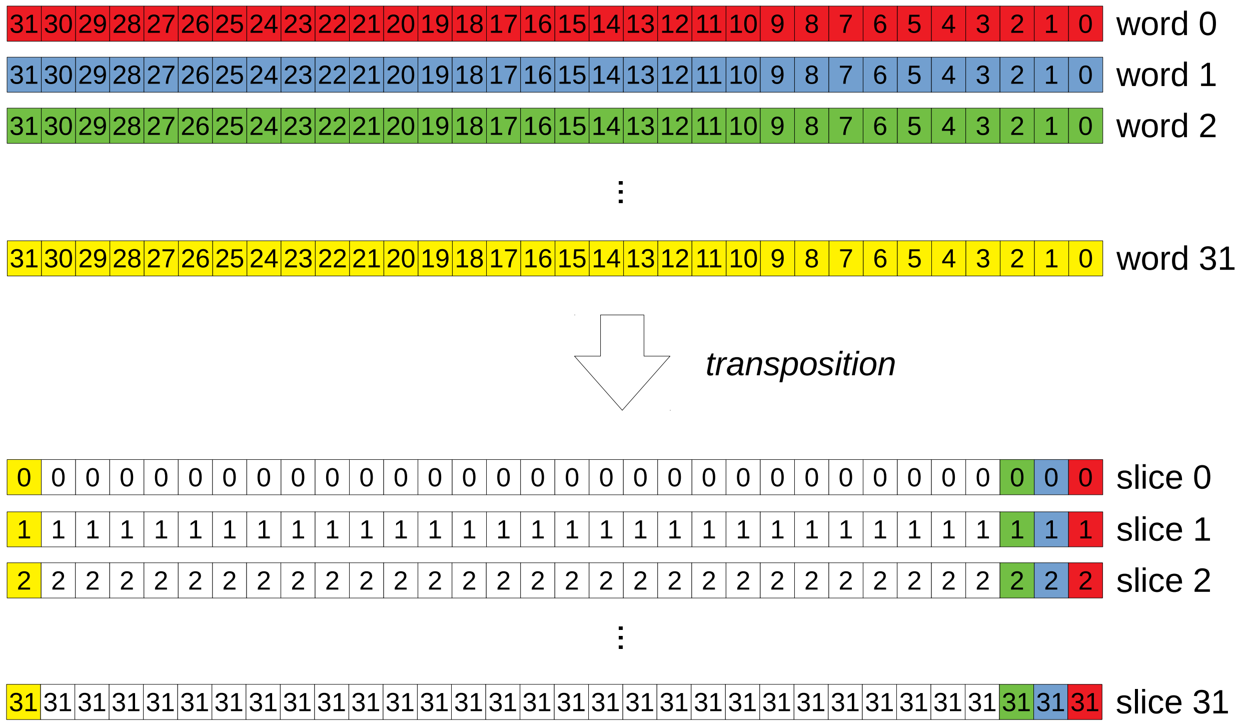
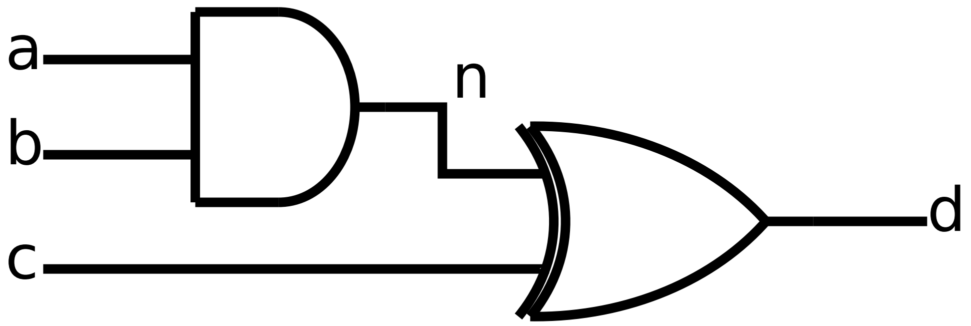


Figure 2.1: A transposition of 32 words of 32 bits into slices

Instead of using a look-up table, the bitsliced S-box is computed as a series of logical bitwise operations (`AND`, `OR`, `NOT`, `XOR`, *etc.*) that directly follow the structure of the S-box implemented in hardware as logic gates. That is, each logic gate in the hardware description is converted into a bitwise instruction in software. While the width of inputs to logic gates is a single bit, this is not the case for bitwise instructions. For example, consider Figure 2.2, which shows a simple circuit comprising two gates and the bitsliced equivalent in C-code. In Figure 2.2b, `MDTYPE` is the type that represents a full-width unsigned integer. This code computes multiple copies of the circuit in parallel, again depending on the size of `MDTYPE`.

One caveat, however, when considering mapping logic gates to logic instructions is that although logic gates can be specified an any order (since all “execute” at the same time), the logic operations in software must satisfy the dataflow dependencies created by the structure of the circuit. In our toy example, the `AND` operation must be computed before the `XOR`, otherwise `n` would be used before its initialization.



(a) Toy logic gate circuit

```
void toy_circuit(MDTYPE a, MDTYPE b, MDTYPE c, MDTYPE* d) {
    MDTYPE n;
    n  = a & b;
    *d = c ^ n;
}
```

(b) The logic gate circuit implemented as a C bitsliced function

Figure 2.2: Toy example of a circuit and possible bitsliced function implementation in C

In [12], the DES gate-level circuit is computed using this technique. It is clear to see how any cycle-free combinational circuit can be implemented this way in software. However, this form of computation is not always preferable to standard software implementations. Many algorithms are better suited for software rather than hardware for one reason or another.

Nevertheless, as bitsliced functions directly follow the hardware design, more efficient hardware leads to faster bitsliced code, and there are reasons to believe block ciphers are good candidates for bitslice programming:

1. Block ciphers often contain elements that are efficiently implemented in hardware. One popular class of block cipher designs are Substitution-Permutation Networks (SPNs), which, as the name implies, contain permutations. These are "free" in hardware (meaning only wires are required)—so also very low-cost in bitslice programming.

2. Similarly, block ciphers designers often consider hardware implementations during the design process.

3. They may use small-width S-boxes or otherwise work on words smaller than optimal for software implementations [12].

4. The algorithms are often not control-intensive. Because bitsliced functions emulate the entire gate-level circuit, algorithms that require branching logic lead to a selection at the end between pre-computed results, which is largely performance disadvantage.

Bitslice programming has been applied to ciphers other than DES to great effect. Matsui and Nakajima [14] in 2007 used bitslice programming to create a high-performance Advanced Encryption Standard (AES) [15] implementation targeting an Intel Core2 processor. This version set a record for software AES speed at the time by targeting 128-bit Single Instruction, Multiple Data (SIMD) XMM instructions. These instructions operate on registers a multiple of times larger than the processor word size—allowing for greatly increased parallelism and performance for bitsliced programs. Simply put, doubling the word size doubles the performance for bitsliced programs.

Beyond using bitslice programming to implement ciphers, some block cipher designers have tailored their algorithms with bitsliced computation in mind. Notable examples include Serpent [16], RECTANGLE [17], and GIFT [18]. Serpent, a finalist in the NIST AES selection process (by which Rijndael was selected) [19], was designed to fully utilize the parallelism of bitslice programming on a 32-bit processor. Instead of computing 32 blocks at the same time, however, a single block's encryption was split into 4 slices. This meant 32 S-boxes could be computed in parallel and a linear transformation layer computed using just bit shifts, bit rotations, and XOR of the slices [16].

2.2 Masking

Masking is a typical software countermeasure against differential power analysis (DPA) attacks. As introduced by Kocher et al. [20], DPA is based on statistically distinguishing an intermediate value in security-critical computation. An attacker must collect many traces and corresponding output of the computation—usually ciphertext. By guessing part of the key (a small number of bits, *e.g.*, 6) and using that to recreate an intermediate value of the computation, the traces can be separated into groups based on that value. The difference of the averages of these two groups will show large spikes when the guess is correct and will be flat otherwise.

The general idea behind the masking countermeasure to this attack is to decouple the dynamic power of the device from the computation on secret data. The way this is accomplished is through splitting the data and computation into several separate parts—or *shares*—that do not give information about the original data unless all are known. The standard way to split a bit variable v is to create a set of d shares $v_1, v_2, \ldots, v_d$ such that $v = v_1 \oplus v_2 \oplus \cdots \oplus v_d$ [21, 22]. The desired property is satisfied if $v_1 = v \oplus v_2 \oplus v_3 \oplus \cdots \oplus v_d$ and $v_2, \ldots, v_d$ are random. Unless the attacker has information about all the shares, no information about the original v is known. The number $d-1$ is referred to as the masking order, which refers to the number of intermediate values that an attacker could target and still be thwarted. A masked implementation of the algorithm is then used to compute the result. What creating a masked implementation of an algorithm entails largely is algorithm dependent, but in general linear operations are simple to mask since a linear operation f by definition $f(v) = f(v_1) \oplus f(v_2) \oplus \cdots \oplus f(v_d)$. Nevertheless, block ciphers contain non-linear elements by design; the traditional example being S boxes. Research has been done into masked implementations of these elements (full S boxes [21] or at a gate level [23, 24]).

In 2004, a powerful side-channel attack was introduced named Correlation Power Analysis (CPA) [25]. This attack has the same general setup as DPA, in that the attacker measures a number of traces of computation with secret data and records the output, but instead uses a Hamming distance leakage model. That is, the power usage does not just depend on the value of the data being processed but the number of bits flipped when a constant but unknown reference state changes to a value of interest. A correlation factor for the power and Hamming distance is computed for guesses of the unknown values and maximized. In [25], Brier et al. explain that CPA requires fewer traces than DPA and does not suffer from some of the problems and limitations that were identified with DPA. However, masking is still an effective countermeasure against this form of power analysis. In fact, there has been work showing that masking, under certain conditions, is provably secure [6]. Translating the information theoretic security to a real-world implementation, however, has been shown to have serious issues for designers of those systems [26, 27].

2.3 Intra-Instruction Redundancy

Another class of attacks to which embedded systems are vulnerable because an attacker may have physical access is fault attacks. Injecting errors by various means into computation using secret data can lead to the extraction of that secret information. The seminal paper on fault attacks on symmetric key cryptographic algorithms by Biham and Shamir [28] introduced Differential Fault Analysis (DFA). In this attack, an attacker encrypts the same plaintext twice. During one of the encryptions a fault is injected into the device and a single bit is flipped, which causes the corresponding ciphertext to be incorrect. The attacker does not tamper with the second encryption and the correct ciphertext is produced. Information about the internals of the algorithm, such as part of the key, may be extracted by comparing the two ciphertexts even if the specific location in which the fault was injected in the algorithm is not known. With enough pairs of ciphertexts the attacker may be able to extract the entire key.

The standard countermeasure against fault attacks includes some form of redundancy. That is, after computation the results are checked for faults against some secondary data. This may include designs that compute the results twice and compare at the end, take the results and undo the computation to see if it matches the original input, or check that the processor jumps to the correct location in code based on a precomputed signature [29]. One specific countermeasure of interest (because work in this thesis supports it) is intra-instruction redundancy [7]. Like other countermeasures based on redundancy, this countermeasure computes multiple copies of the data and compares the results. This, however, is done synchronously via the use of bitslicing. Multiple copies of the data are transposed into slices and computed on in parallel. If a fault is injected and flips a bit, the difference in results is detected. Additionally, data with known ciphertext is also included in the slices so that if an instruction is skipped the result will not match the expected value.

2.4 Model Checking

Model checking [30] is a verification technique that shows a design satisfies a set of requirements. By modeling a finite-state system at a reasonable level of abstraction with these tools the desired properties can be verified in *all* possible executions. This makes model checking a powerful tool when designing systems, especially when systems are safety critical, and provides a higher confidence of correctness than standard testing methods. Requirements are specified using temporal logic, which allows future behavior to be specified along with the current state. This lets properties like "the algorithm will always finish" to be formalized and checked. One other strength of model checking is that when checking properties fails, the tool can present the violating behavior of the system as a counterexample.

The model checker used in this thesis is the TLC model checker for the TLA$^+$ language [31].

Chapter 3

Related Work

In this chapter we will discuss previous research on the problems that this thesis focuses upon. The primary categories of works are generation of bitsliced code, use of DMA controllers for data processing, and generation of slices.

3.1 Bitsliced Code Generation

One of the issues facing programmers wishing to use bitslice programming is the creation of bitsliced code itself. Early bitsliced implementations often took parts of algorithms described as Boolean functions and manually converted them to bitsliced code, such as how [32] compared different DES S-box implementations. Or, in the case of [33, 34], extensive search through possible Boolean S-box functions to find efficient implementations. Others, such as [12], start with a gate-level hardware design and then move that implementation to software. In either case a manual conversion to software is required. Writing bitsliced code by hand, however, is no less tedious as it is equivalent to describing the algorithm as an ordered hardware circuit. For large designs, this becomes very time consuming and the chance of errors increases. This problem sets the scene for automated bitsliced code generation tools.

The first automatic bitsliced code generator was the BitSlice Compiler (`bsc`) [35, 36]. To use `bsc`, a user would specify their algorithm in a custom language, and the compiler would generate bitsliced code in C. The language was designed with block cipher implementation in mind and provides, among other things, the user the ability to define variable-size registers, permutations, and look-up tables, and to apply logical operations, concatenation, permutations, and table look-ups to those registers. All are standard design elements of block ciphers. Like software, the compiler treats the actions in the language as ordered, and so the user is responsible for incorrect ordering of instructions in the generated C code.

Close to two decades passed before the problem of automated bitsliced code generation was studied again. In 2017, Xu and Gregg [37] developed efficient bitsliced code for operations—addition, multiplication, shifts, *etc.*—on customizable data precisions of integer and floating-point types. In some applications, it is often the case that the data does not need the full width of a processor word, so full width operations include some wasted computation. Via bitslicing, creating words of variable precision, such as 3-bit, 5-bit operands, is as straightforward as choosing the number of slices to represent the data. To operate on these vectors of slices, Xu and Gregg first wrote bitsliced C code by hand, in some cases, "using circuit techniques that were originally developed for hardware" [37]. However, to further optimize the code, they traced the execution of logical operations in their code, and from that, generated a hardware design in Verilog, which could be put through a logic optimization tool. Finally, the optimized Verilog was converted back to C by mapping the hardware to a library of cells describing supported bitwise instructions in the processor's instruction set.

The following year, Mercadier et al. introduced a successor to Pornin's `bsc` named `Usubac` for the USUBA language [38]. USUBA includes many of constructs that were supported by `bsc` for cryptographic implementation while also providing support for hierarchical designs via defining "nodes," which are compiled into individual functions. This allows repeated sections of bitsliced code, common in block cipher designs, to be replaced by a function call with inputs and outputs as arguments. One significant difference between the `bsc` language and Usuba is that Usuba is a "synchronous dataflow language" [38], meaning that all the orderings of the equations are equivalent. `Usubac` schedules the bitsliced code such that it does not violate the dataflow structure of the design. This also provides the benefit of efficient scheduling since the compiler can optimize instruction ordering to prevent register contents from spilling into memory.

3.2 DMA Data Processing

DMA controllers are a class of hardware peripherals that—as the name implies—have direct access to memory. In the standard design of a microcontroller, a processor is surrounded by a number of hardware modules, including memories and peripherals. The processor commonly communicates with these devices over a bus. Peripherals, such as timers, GPIO, various interfaces, and hardware accelerators, often are not able to have control over the bus and only send and receive data as directed. DMA controllers, however, are an exception because having a hardware module dedicated to moving data around—in memory or between memory and peripherals—can be more efficient than the processor managing the transfers [39, 40]. The processor may work on some other task or be put into a low-power state while the transfer completes. Alternatively, the DMA can be transferring data for the next computation step from slow memory to faster memory while the processor computes—effectively removing the latency of the slow memory from the processor's perspective. This technique can be referred to as double buffering or a ping-pong scheme [41].

As shown in [39], introducing a DMA to a system with a high-performance accelerator/co-processor can reduce data transfer overhead for a small increase in area. This is taken further in works like [42] where the accelerator itself includes a DMA controller. In that work a binary convolutional neural network accelerator includes a feature map memory for temporary internal storage of layer inputs and partial results since the task is so data heavy. A DMA controller is included to manage access to this internal memory. The DMA controller also packs results into words before writing to memory. As such, this is an example of how DMA controllers can also be used for some data processing/reorganizing tasks, though this is not very common outside of ones from digital signal processors.

A DMA controller not part of a larger accelerator with this capability has been introduced in [43]. This work is the closest to the DMA controller presented in this thesis. A number of different transfer modes are supported to accelerate common scientific operations, including a “reshape” transfer to change format of two-dimensional arrays. Moreover, it includes a “matrix transposition module” for accelerating that operation, though the elements in this case are words rather than individual bits. Words are written into an internal buffer row-wise and then read out column-wise. This is the same approach used in this thesis.

3.3 Slice Generation

One drawback of bitslice programming is that transforming data in and out of a bitsliced representation (see Section 2.1) incurs a performance overhead. This penalty can be a significant fraction of the total computation depending on the algorithm. Consider the task of generating slices for n words of n bits each as the transposition of a $n \times n$ bit matrix. A naïve algorithm for this builds each row in the output matrix one bit from each input row at a time, which is $O(n^2)$ operations.

Better algorithms for this operation have been identified. In [35, 36], Pornin describes a recursive algorithm that only requires $O(n \log n)$ operations. With an n-bit processor, many smaller transpositions are computed in parallel, which build up to a full $n \times n$ transposition. This algorithm has become the standard for slice generation, though it has a few different formulations at this point. Once such formulation uses the SWAPMOVE technique [44].

Although the recursive transposition algorithm has a better algorithmic complexity, that does not imply it is the most efficient in all cases. Matsui and Nakajima [14] provided an assembly transposition algorithm for the Intel Core2 processor using XMM SIMD instructions for 128 words of 128-bits. This method, despite not being based on the recursive algorithm, achieved speeds of sub-1 cycle per byte.

Until now we have only discussed software algorithms for transposition. The SKIVA architecture from [8] includes hardware support for slice generation in the form of the `TR2` and `INVTR2` custom instructions. These instructions interleave the bits of two registers and stores the results in two other registers. Implementing the "balanced two-way merge sorting" algorithm from [45]—which is the basis for Pornin's recursive algorithm [35, 36]—using these instructions is also $O(n \log n)$ but requires fewer bit manipulation instructions than standard software implementations.

Chapter 4

Software Support for Bitslicing

In this chapter we will discuss the PSPCG. The PSPCG is a software tool for creating bitsliced code that fits in the Parallel Synchronous Programming (PSP) paradigm [9]. The user of the tool describes their design in the well-known Verilog hardware description language (HDL), and PSP C-code is produced as output. Verilog allows description of the bitsliced design at the Register-Transfer Level (RTL) or behavioral level rather than specifying at the gate level. The tool is built as a back-end for the Yosys Open Synthesis Suite (Yosys) [46]. This open-source software tool synthesizes the design into logic gates, which are converted into the C language by the back-end. This tool supports architectures of any bit-width and with varying bitwise instructions. For example, later sections of this chapter discuss code generated for ARM, ARM NEON, and MicroBlaze architectures. Chapter 5 discusses code for RISC-V.

4.1 Functionality

This section describes the steps a user takes to synthesize their design from Verilog into C-code using Yosys and the PSPCG and the relevant options the tools provide to the user. This general procedure and necessary inputs are shown in Figure 4.1. As shown in that figure, there are two previously undiscussed inputs needed in the process that depend on the architecture being targeted. Assume for example's sake that the user has a design to be run

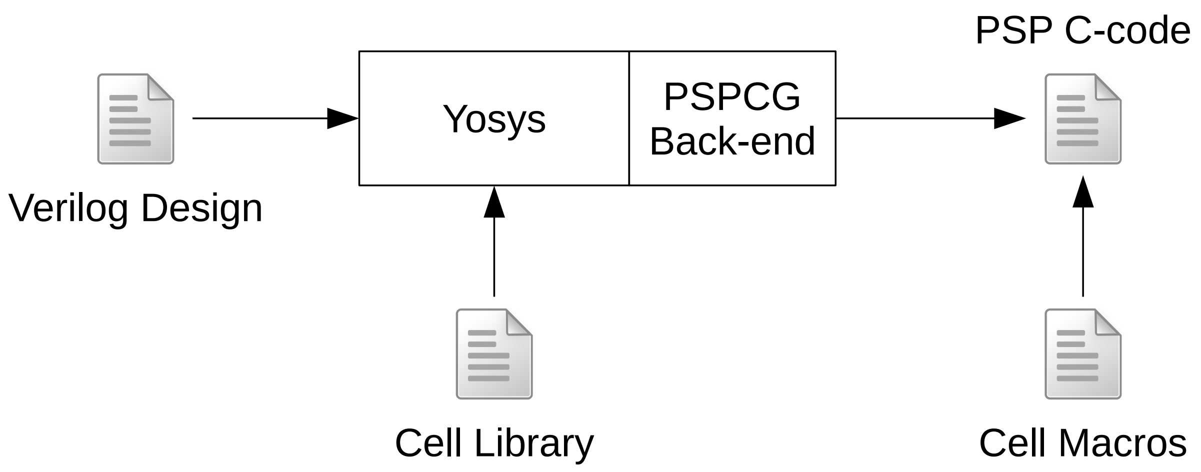


Figure 4.1: Procedure of using the PSPCG

on architecture A. The width of a processor word in A does not matter until later, but the supported bitwise instructions do. Say A has the following standard bitwise instructions in its instruction set:

- `not rd, rs` (the complement of `rs` is stored in `rd`)
- `and rd, rs, rt` (the AND of `rs` and `rt` is stored in `rd`)
- `or rd, rs, rt` (the OR of `rs` and `rt` is stored in `rd`)
- `xor rd, rs, rt` (the XOR of `rs` and `rt` is stored in `rd`)

The first step in using the PSPCG is synthesis of the design into a netlist of wires and cells. These wires and cells become the variables and instructions in PSP. However, the cells that should be used in the netlist depends solely on the instructions A supports. The "Cell Library" input in Figure 4.1 is the file that provides Yosys with the cell definitions from which it should construct the netlist. The cell library for A (only one "Cell Library" and "Cell Macros" file needs to be written for each architecture) would define the above bitwise instructions as equivalent one (for `not`) and two input logic gates. The library would also include definitions for Flip-Flops. It should be noted that Yosys is also used in this step to optimize the design using a variety of techniques. In this step the user has the option of flattening their design into a single module. Typical hardware designs have a hierarchical structure of instantiated modules. Hardware modules are converted into functions by the PSPCG. Flattening in this step causes the PSPCG to generate a single C function in the end for the entire design. As will be explained in Subsection 4.3.2, this may decrease the performance of the code and increase code size, but it allows the PSPCG to support more types of designs.

The next step in the process is done by the PSPCG back-end. It converts the netlist (represented in the internal Yosys language) into PSP C-code. Modules are converted into C functions. Their inputs and outputs become function parameters. Wires are converted directly into local variables and cells are converted into C macros and written into the function. These macros are defined in the "Cell Macros" input (typically a C header file) from Figure 4.1. An example of these header files is shown in Figure 4.2 for the A architecture. Since the bitwise instructions A supports are easily defined using C bitwise operators, they suffice. However, these macros could also be expanded into inline assembly to force the compiler to use a specific instruction. The macros' names need to be the same as the corresponding cell in the "Cell Library."

The library also includes definitions for Flip-Flops, which in PSP store their value between function calls. Each time the PSP function is called is equivalent to a new clock cycle in hardware. The default way the PSPCG creates values that persistent between calls is by having the target of each Flip-Flop declared a local `static` variable. The user also has the option named "external-state." Instead of static variables, the state of each function is is stored in a C `struct` that is passed to the function as an argument each call. This has a

```
#ifndef _CELLS_H_
#define _CELLS_H_

#define MDTYPE unsigned int

#define NOT1(a,r) { r = ~a; }
#define AND2(a,b,r) { r = a & b; }
#define OR2(a,b,r) { r = a | b; }
#define XOR2(a,b,r) { r = a ^ b; }
#define DFF(clk,d,q) { q = d; }

#endif
```

Figure 4.2: Example contents of a “Cell Macros” header file for the A architecture

number of benefits:

- The function is not tied to a specific instance of the design. Multiple “states” of the design could be held separately and interleaved if desired.
- A typical PSP design needs to be reset to a known state before execution. Instead of doing a function call to reset the state, which needlessly evaluates the full function, external state allows the state to be reset externally. This avoids a full function call and reduces the size of the function as reset logic can be removed.
- External state is required to support some hierarchical designs. Similar to the first benefit, if a hardware submodule is instantiated multiple times, external-state allows the reuse of single function definition. The states `struct` of submodules are included in the state `struct` of their parent. This allows a hierarchy of state to match the design's module hierarchy. See Subsection 4.3.2 for more details.

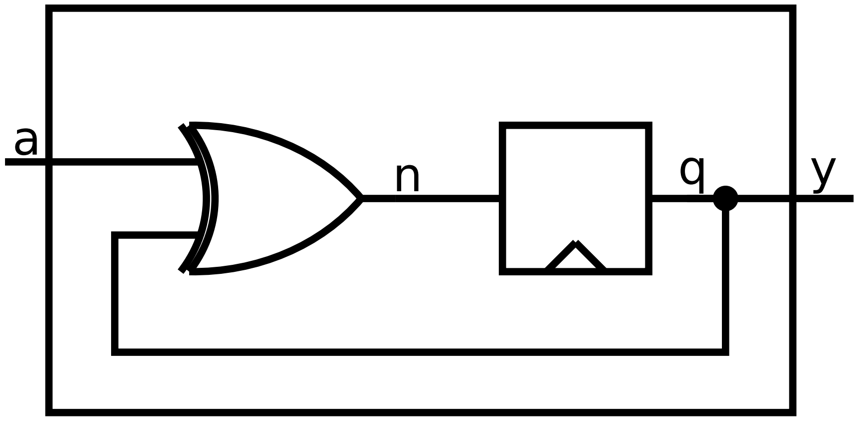
The one final definition included in the “Cell Macros” header file is `MDTYPE`. This is the type of a slice, and the PSPCG uses it for all variables. For typical C implementations `MDTYPE` may be an `unsigned int`, `uint64_t`, or similar depending on the width of the processor. It may also be types for targeting SIMD instructions for example.

4.2 Implementation Details

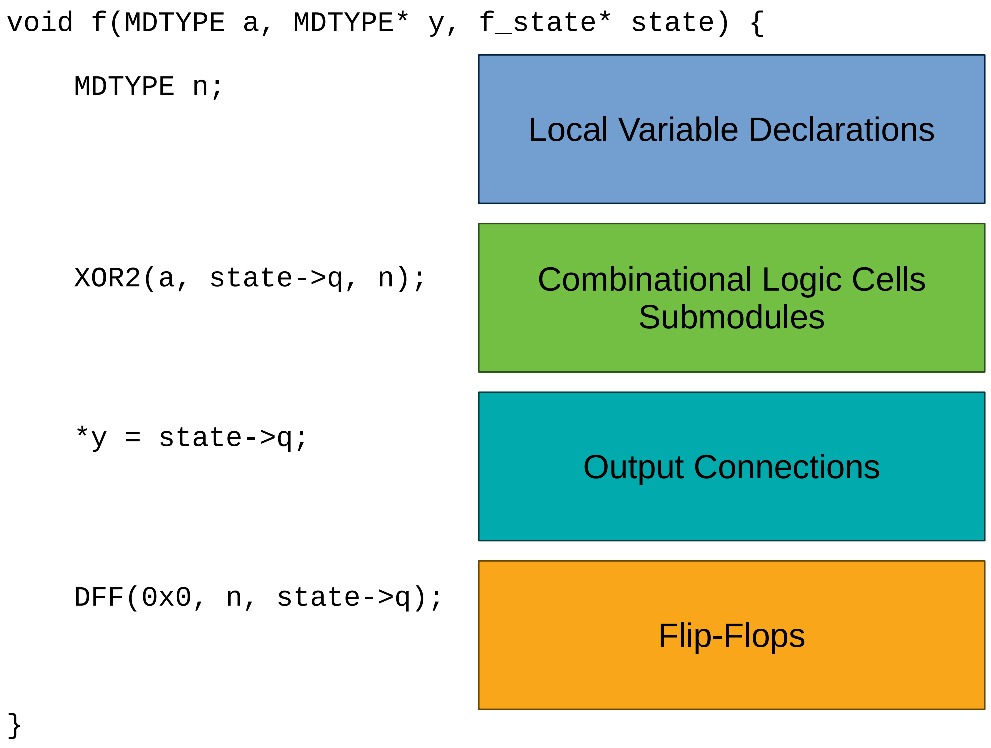
This section will discuss some of the implementation details of how the C++ PSPCG back-end produces correct C-code. Though producing syntactically correct code is important, it is not sufficient for the back-end to directly dump the netlist as provided by Yosys synthesis in "C format" for the code to have correct behavior. The function needs to be structured and cells need to be ordered for sequential evaluation. The general structure of a module is shown in Figure 4.3 along with an example of a simple hardware design for the A architecture with "external-state" enabled. This structure supports a single-pass sequential evaluation of a hardware module. All local variables are declared first. Then non-Flip-Flop cells are evaluated in topologically sorted order. This group of cells also includes submodule function calls. Next, any direct connections from Flip-Flop outputs to module outputs are written. Finally, Flip-Flops are evaluated, and the module's state is updated.

4.2.1 Identification of State

One crucial step before declaring local variables is identifying which signals constitute the state of the module. These signals are either declared `static` so that they persist between function calls or are omitted entirely if "external-state" is used. In this case they appear as fields in the state `struct`. The PSPCG identifies these signals via their connection to the `\Q` port on a cell. In Yosys, a cell has a `\Q` port if and only if it is a Flip-Flop. If "external-state"



(a) Example sequential circuit



```
void f(MDTYPE a, MDTYPE* y, f_state* state) {
    MDTYPE n;

    XOR2(a, state->q, n);

    *y = state->q;

    DFF(0x0, n, state->q);
}
```

Local Variable Declarations
Combinational Logic Cells Submodules
Output Connections
Flip-Flops

(b) The logic gate circuit implemented as a PSP function and sections

Figure 4.3: Structure of a module implemented as a PSP function with example

is specified and a hierarchical design is being synthesized, then the state of a module also includes any state of any submodules.

4.2.2 Netlist Sorting

The non-Flip-Flop and Flip-Flop cells are also ordered, separately. For combinational cells, no cell appears in the function until its inputs have been produced (either as input to the function, state, or by another cell). The algorithm used for this sorting is based directly on the Yosys "torder" pass, which recursively works backwards through the circuit until an acceptable order is found. Any combinational loops are detected and reported along with the involved cells to the designer. For Flip-Flop sorting, the opposite relationship is needed: each Flip-Flop's output needs to be fully consumed before it produces the next. This orders them such that their values are not overwritten. Similarly, any loop caused by the Flip-Flops is reported to the designer.

4.3 Discussion and Analysis

4.3.1 Design Considerations

Code produced with the PSPCG is originally described in Verilog. However, designers should not confuse writing Verilog that becomes hardware with writing it for PSP code generation. The main distinction between the two is that hardware is massively parallel while software is primarily sequential. As such, not all techniques for enhancing the performance of hardware designs have a benefit when targeting software. Pipelining is a good example of this. Pipelining can increase throughput in hardware by introducing delay stages to decrease the critical path and increase the operating frequency at the cost of increased latency. However, the benefit of this is lost when computing the design sequentially in software. Instead of computing each pipeline stage in parallel, each must be computed sequentially, which is equivalent to having no pipelining at all. In fact, the addition of delays may hinder performance.

When designing the circuit to run in software, area is doubly important. Not only does the code size increase with the number of gates but also the computation time. Specifically, the computation time is proportional to the number of gates times the number of calls of the PSP function (number of clock cycles in hardware). Though they are often competing goals when targeting hardware, reducing area also often increases throughput in this situation. The typical way of improving performance in hardware of utilizing more spatial resources to increase parallelism does not translate to sequential software evaluation. Instead, designers should try to find a balance between area and number of calls. Fully unrolling to a design that requires only a single call may reduce the overhead associated with calling, but it incurs other performance penalties. Namely, increased code size, register pressure, and instruction cache misses may occur. Such problems are detailed in later sections.

Performance can be gained, however, by using the software memory model. With the "external-state" option, software outside of the PSP function can read and modify the state of the design, which leads to a number of optimizations:

- There is no need for reset logic to be included in designs since any state can be initialized externally. This saves a full call to the PSP function to reset the state. It also saves area, which decreases the computation time of each call, since any reset logic, even if not being used, is still evaluated.

- In some cases, there is no need to include logic that "loads" a value into the state of a design. This could be, for example, loading plaintext or key into a cipher design. By loading values directly into state through external means, the area can be reduced by removing this load logic.

- There is no need to have an output that is simply part of the state. The calling code can read the state directly out of the `struct` at the end of computation.

4.3.2 Hierarchical Designs

Table 4.1: Code size and computation time for hierarchical design vs. flattened design of AES-128 PSP code on a MicroBlaze soft processor core

	`.text` Size (bytes)	Computation Clock Cycles	Cycles / byte
Flattened	289,740	2,450,191	4,786
Hierarchical	113,600	1,089,326	2,128
Relative Change	−60.8%	−55.5%	

Designers should also consider making their designs compatible with hierarchical PSP synthesis. For supported designs the PSPCG can create separate PSP functions for submodules, including those that have associated state. Hierarchical synthesis can greatly decrease the code size and increase the performance of the design in software. To demonstrate this, two PSP versions (one hierarchical, one flattened) of AES were synthesized from a design from OpenCores [47]. The target architecture was the 32-bit MicroBlaze soft processor core, which has a typical set of bitwise instructions. The results of this experiment are shown in Table 4.1. The reduced code size (a smaller `.text` memory segment) of the hierarchical code has a straightforward explanation: the reuse of submodule functions means less code needs to be generated. In the AES example there are, in total, 20 instantiated S-box modules. The flattened design makes the equivalent of 20 copies of this module in its single function while the hierarchical design calls the S-box function 20 times. This, however, does not explain the drastic increase in performance. One typically useful analysis technique for PSP functions is to create a breakdown of the instructions present. The number of instructions directly reflect the gates of the circuit along with overhead from managing memory. When the circuit is wide enough, the compiler can no longer store all intermediate values in registers, and values “spill” into memory. The instruction breakdown in Table 4.2 shows the counts of instructions the processor does during a call to the PSP code. That is, it is the effective instruction breakdown rather than a breakdown of the instructions present in the `.text` segment. Because PSP functions contain no branching instructions, finding the effective counts is just a matter of counting the instructions in each function and multiplying by the number of times that function is called. The hierarchical design shows a modest improvement in the breakdown over the flattened design, but this cannot be solely responsible for the performance increase.

Table 4.2: Effective instruction breakdown for hierarchical design vs. flattened design of AES-128 PSP code on a MicroBlaze soft processor core

	Logic					Overhead			
	`not`*	`xor`	`and`	`or`	**Total**	`lwi`	`swi`	Other	**Total**
Flattened	793	1,373	8,851	11,677	22,694	22,749	6,897	3	29,649
Hierarchical	546	1,253	8,888	11,866	22,553	19,736	6,926	190	26,852
Change					−0.62%				−9.4%

(*) `not` implemented as `xori` with `0xFFFFFFFF` immediate

To further investigate, MicroBlaze profiling was done for the two implementations (Table 4.3). The standout difference was the cycles per instruction metric. The flattened implementation had over twice the cycles/instruction despite having a similar instruction makeup and data cache miss rate. The primary explanation for this—and the performance difference—comes from differences in instruction cache use. Though both designs are too large to fit entirely in the 16KiB instruction cache, the hierarchical design better utilizes it due to the reuse of the S-box function (which contains 88% of the total effective instructions over its 20 calls). Through its reuse, the function remains in instruction cache, and cycles are not wasted—as happens in the flattened design—fetching from main memory what are effectively the same instructions 20 times per execution.

Table 4.3: Profiling for hierarchical design vs. flattened design of AES-128 PSP code on a MicroBlaze soft processor core

	Cycles/Instruction	Data Cache Miss Rate
Flattened	3.16	1.19%
Hierarchical	1.43	0.49%

One instruction cache feature we found useful was speculative prefetching of instructions. The MicroBlaze can be configured to prefetch cache lines that correspond to linear code execution. Because PSP code contains very few jumps or branches, this feature provides a large boost in performance to both flattened and hierarchical designs but does not solve the flattened design's instruction cache inefficiencies in this experiment.

Supported Designs

One important topic we have heretofore neglected to discuss is what makes a design compatible with hierarchical synthesis. The decider is if all submodules are used in a combinational-like manner. That is, synthesis will succeed if you assume all submodules are purely combinational and there are no combinational loops introduced. The submodules need not be purely combinational, but this requirement is introduced to allow evaluation and update of state in the same step for each module. The AES design is an example of a reasonably large design that has this property. On the other hand, the circuit shown in Figure 4.4 does not. The submodule `s` has a combinational path from its output `q` to its input `n` through the XOR gate. The issue this causes for synthesis targeting sequential evaluation is that there is no solution to the question "should the XOR gate or `s` be evaluated first?" The XOR gate needs `q` to be present before evaluation, but `s` needs `n` to be present.

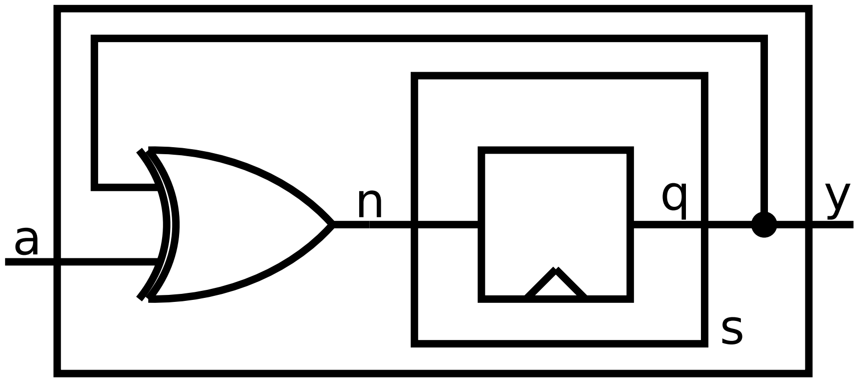


Figure 4.4: A circuit for which the PSPCG does not support hierarchical synthesis

There is, however, a work around to this problem. During the design process, if the designer raises the state in the hierarchy (that is, move state to a higher module) then the "combinational loops" will be broken. This does lead to changes in port definitions of modules. The designer would need to add an input for the state and output for the next state, but no combinational logic needs changing.

4.3.3 Applications

To evaluate how well the generated code performs, we chose to implement the GIFTb cipher on a couple of architectures. This block cipher was chosen, in part, because it is used in the GIFT-COFB [48] algorithm, which is a Round 2 candidate in NIST's Lightweight Cryptography Standardization Project. Additionally, as GIFT was designed with bitslicing in mind, it should be a good candidate for a PSP implementation. The Verilog design used for PSP synthesis is based off the one presented and described in [49] with the optimizations described in Subsection 4.3.1 applied. More specifically, "load" logic was removed along with direct output of the ciphertext since "external-state" was used. The "done" signal was also removed as the calling function can count the rounds more easily. One other change to the design was the exclusion of round constant generation. Because each of the bits uses the same round constant, these are more efficiently passed in via an argument than generated in the PSP function.

For benchmarking the code, the Cortex-M4F of a Texas Instruments MSP432P401R was targeted. This device could be considered a "highly-constrained device" [50] like those for which the NIST Lightweight Cryptography Standardization Project exists. The code was compiled with GCC optimizing for size. Size optimization often produces smaller *and* faster PSP code than other optimization choices. The benchmarking results are shown in Table 4.4 along with other implementations' results from other works. The computation clock cycle counts are for encryption of 32 blocks (with round key generation). It should be noted that the data is assumed to be in the correct transposed format for PSP. For comparison of performance, the Fixsliced implementation is the fully unrolled, high-performance ARM assembly implementation.

Table 4.4: Code size and computation time of GIFTb-128 PSP code vs. reference implementations on an ARM Cortex-M4F

	`.text` Size (bytes)	Computation Clock Cycles	Cycles / byte
Naïve [51]	-	-	523.4
Tornado [52]	-	-	358
Fixsliced* [51]	29,536	84,191	164.4
PSP	25,976	74,121	144.8

(*) Our measurement

Another GIFTb benchmark was also done to showcase the capabilities of the PSPCG. This benchmark targets the ARM NEON SIMD architecture [53] of a Cortex-A9 in a Zynq-7000 SoC. This architecture offers greater parallelism (and performance) for PSP functions due to its 128-bit quad registers. To target these registers and use NEON instructions, `MDTYPE` was defined to be `uint32x4_t` and the cell definitions used NEON intrinsics. However, these are the only changes required since the ARM NEON logic instructions match the base ARM instructions. This means the same PSP function could be used for both the Cortex-M4 and NEON (with a different `cells.h`). The results are shown in Table 4.5. For a baseline, the reference implementation is the fixsliced C implementation from [51], which uses only standard ARM instructions. Once again only PSP computation is included in the cycle counts. For analysis of the cycles spent transposing data, see Chapter 6. As expected, the SIMD implementation is very fast in comparison to a standard software implementation.

Table 4.5: Code size and computation time for 128 blocks of GIFTb-128 NEON PSP code vs. reference implementation on an ARM Cortex-A9

	`.text` Size (bytes)	Computation Clock Cycle	Cycles / byte
Reference* [51]	63,964	288,858	141.0
PSP	67,760	171,614	83.8
Relative Change	5.9%	−40.6%	

(*) Our measurement

Chapter 5

Application of Software Tools to Neural Networks

This chapter is based on work presented in:

R. Singh, T. Conroy, and P. Schaumont, "Variable Precision Multiplication for Software-Based Neural Networks," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, Sep. 2020, pp. 1-7. [Online]. Available: https://ieeexplore.ieee.org/document/9286170/ © 2020 IEEE

Up until now we have focused on applying PSP code generation for block cipher applications. However, the technique is more general purpose than this. In this chapter, we will discuss its application to neural network (NN) processing on embedded devices. Recent years have seen massive growth in the field of deep learning and its uses in solving difficult problems. Along with this growth has been a push to move computation away from centralized servers to the edge of networks where data can be gathered and processed (at least partially) by low-cost, highly constrained devices. One task these devices might do is inference, which evaluates a pre-trained NN on the collected data. Naturally, the efficiency of this task has garnered interest from the academic community and private enterprise. Among many others, one technique of interest to this paper that has been studied is the use of reduced-precision operands [3, 54]. It has been shown that full-width (say 16-bit) operands are not needed in many cases to achieve results with similar accuracy [55].

The particular operation we use PSP to accelerate is at the center of deep neural networks. Each neuron with k inputs $i_1, i_2, \ldots, i_k$ in such a network computes its activation y as

$$y = f\left(\sum_{n=1}^{k} w_n i_n + b\right)$$

where b is the neuron's bias, w_n is the weight for input i_n, and f is its activation function. At the heart of this equation is the weighted sum of the input neurons. When viewed at a network level, multiple neurons do this operation in parallel on the same inputs with different weights. An example of this shown in Figure 5.1. In general, this operation can be described as parallel dot products (of weight vectors with the same input vector), or a matrix-vector multiplication. For the example the operation is compactly shown as

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1k} \\ w_{21} & w_{22} & \cdots & w_{2k} \\ w_{31} & w_{32} & \cdots & w_{3k} \\ w_{41} & w_{42} & \cdots & w_{4k} \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_k \end{bmatrix}.$$

Highly constrained embedded devices sometimes lacking a hardware multiplier, however, are not well equipped to benefit from matrix-vector multiplication with reduced-precision operands using standard software techniques. Bitsliced software, and more generally PSP, is very capable with data of unusual precision. A set of 5-bit values, for example, is stored in 5 processor words. This sets the scene for software acceleration via PSP. The question then becomes "what is the appropriate design to synthesize?" Focusing on the parallel dot product view of matrix-vector multiplication hints at what operation the "thread" of the PSP function could be doing in parallel. For example, one "thread" should compute the dot product highlighted in red:

$$s_1 = w_{11} \times i_1 + w_{12} \times i_2 + \cdots + w_{1k} \times i_k.$$

Hence, each bit—which executes an independent instance of the design—should compute a dot product using a specific row. The chosen design is a multiplier-accumulator (MAC), which supports vectors of variable dimension. Figure 5.2 shows a block diagram of the MAC. To support variable precision operands, the design accepts a `WIDTH` Verilog module parameter, which defines the width of `a`, `b`, `out`, and other relevant signals. By passing each column of the weight matrix along with the inputs as arguments to the MAC PSP function, the accumulated value at the end will be the desired weighted sum for all the neurons.

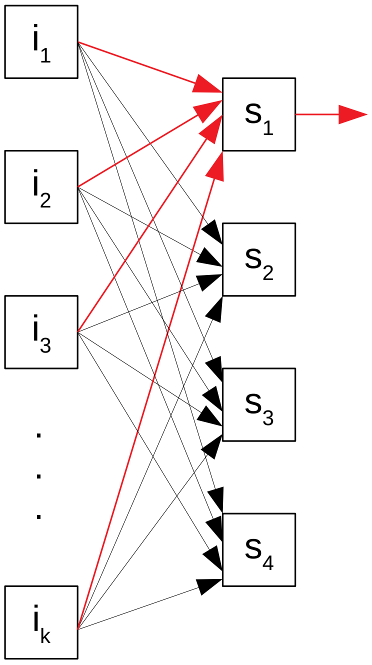


Figure 5.1: Fully-connected layer of a neural network with four weighted sums

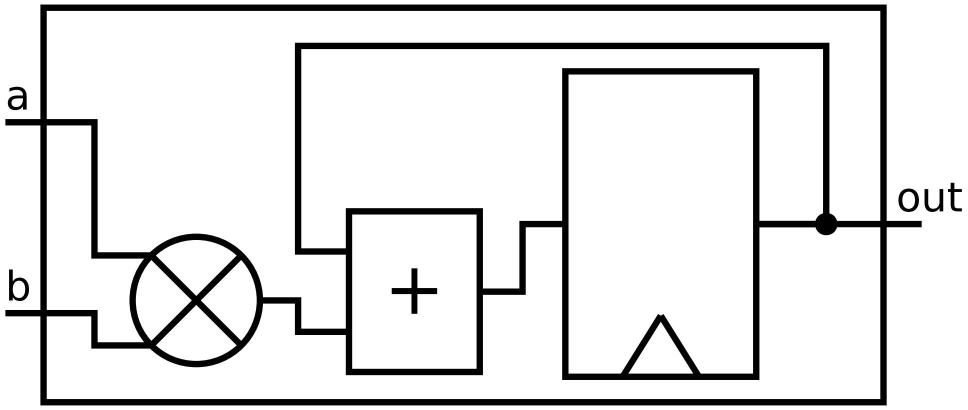


Figure 5.2: Multiplier-accumulator block diagram

5.1 Results

The target platform for the design was the PULPissimo SoC [56] implemented within a Zynq-7000. This platform includes a 32-bit processor compliant with the RISC-V instruction set architecture. The PSPCG was used to synthesize the MAC design for it like any other architecture—a cell library and cell definitions were created. Multiple syntheses were done for various operand bit precisions. To compare this method to standard implementations, we target the multiplication of a 32×32 matrix and a 32×1 vector since this corresponds to every bit in the PSP function doing useful work. The matrix and vector elements were pseudorandomly generated integers of a given precision. The platform includes a hardware multiplier, but by compiling code targeting the `rv32i` instruction set (the base RISC-V integer instruction set), the compiler will instead use software multiplication only. The average cycle count over 100 trials for each bit precision and implementation is presented in Table 5.1. The implementation with a hardware multiplier naturally requires a set number of cycles independent of the precision. This implementation outperforms the "no hardware multiplier" implementation in all cases but is slower than the PSP implementation for very small precisions (≤ 3). When comparing standard software without a hardware multiplier to the PSP implementation, the PSP function is better for small precisions with a crossover point between 8 and 16 bits, which is sufficient to support its use in reduced-precision NNs. Note that the number of cycles is consistent with a quadratic complexity with the bit precision.

Table 5.1: Cycle counts of PSP matrix-vector multiplication vs. standard software with and without a hardware multiplier for various bit precisions

[used with permission] © 2020 IEEE

	Cycle Counts for Bit Precisions				
	2-bit	4-bit	8-bit	16-bit	32-bit
Standard Software (no HW mult.)	119,711	149,667	175,594	212,845	282,649
Standard Software (HW mult.)	10,418	10,418	10,418	10,418	10,418
PSP	2,613	12,309	66,870	345,174	1,492,374

5.2 Overhead Analysis

As we have found with other designs, register spilling adds a significant amount of overhead to the design. Instead of keeping all intermediates in registers the compiler uses memory (specifically the stack) to hold values it needs later, as needed. In the RISC-V architecture, the performance overhead of this comes from load (`lw`) and store (`sw`) instructions. The larger the fraction of instructions these are, the less efficient the design. Table 5.2 shows an instruction breakdown for 5 MAC PSP functions of different precisions. The overhead percentage (calculated as the number of overhead instructions over the total) remains in the range of 34% to 50% despite the total number of instructions increasing by a factor of 772× from 2-bit to 32-bit precision.

To better understand the need for these overhead instructions, we also analyze how well

Table 5.2: Instruction breakdown of PSP matrix-vector multiplication of various bit precisions

[used with permission] © 2020 IEEE

	Logic					Overhead				
Bits	`not`	`or`	`xor`	`and`	**Total**	`lw`	`sw`	**Total**	Other	Overhead
2	6	5	9	15	35	17	8	25	0	41.67%
4	23	77	36	100	236	82	43	125	2	34.44%
8	119	451	196	489	1,255	605	192	797	2	38.80%
16	394	2,211	848	2,109	5,562	3,885	1,227	5,112	2	47.88%
32	1,968	9,499	3,009	8,727	23,203	17,369	5,729	23,098	2	49.88%

the compiler is using the registers. Ideally all registers are being used—at least, holding intermediate values. To see if this is the case, we analyze the lifetimes of variables in the assembly code of the PSP function. Due to the linear nature of the code, static analysis is sufficient to fully understand how registers and the stack are used over time. A visualization of the usage is shown in Figure 5.3 for the 4-bit precision function. Each instruction in the function is numbered and placed on the x-axis. Because PSP functions have no jumps or branches, this is equivalent to time being on the axis. The registers are shown on the top half of the y-axis while the bottom half shows stack memory addresses (relative to the stack pointer).

A colored line segment at a specific y-value corresponds to a value being held in a specific register/memory address, otherwise it is not being used at that time. Of course, each register and memory address always holds some value, but we only consider a value present if it is used at a later time. A change in color corresponds to a new value being loaded by a specific instruction. The register usage appears very dense for the 4-bit function as well as the 8-bit function in Figure 5.4, and no registers are being completely unused. It is clear the 8-bit function uses the stack much more extensively, but that is to be expected with a wider design. Note that the line segments in memory are usually longer than those in registers,

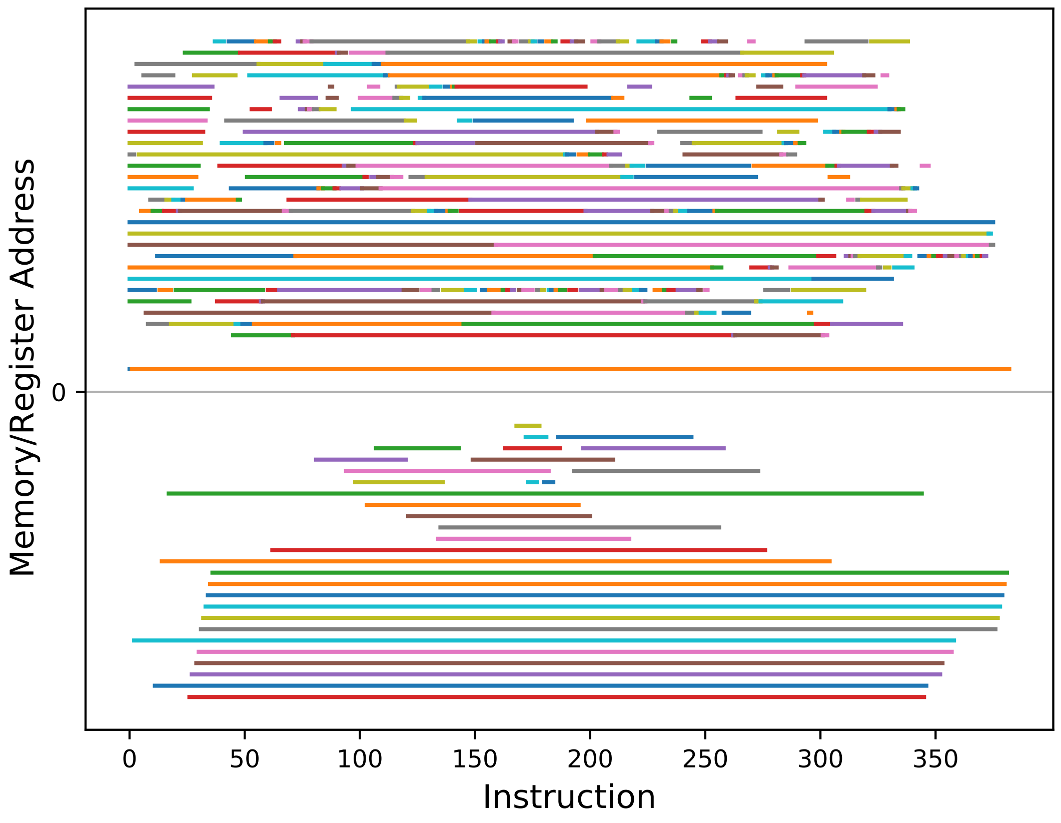


Figure 5.3: Register/Memory usage of the 4-bit MAC PSP function
[used with permission] © 2020 IEEE

which could reflect the compiler storing long-lived variables or those that are not used soon after assignment. The very long-lived values at the bottom of the memory addresses are the function saving callee-saved registers. Overall, we see no reason from this to believe the compiler is producing sub-optimal results.

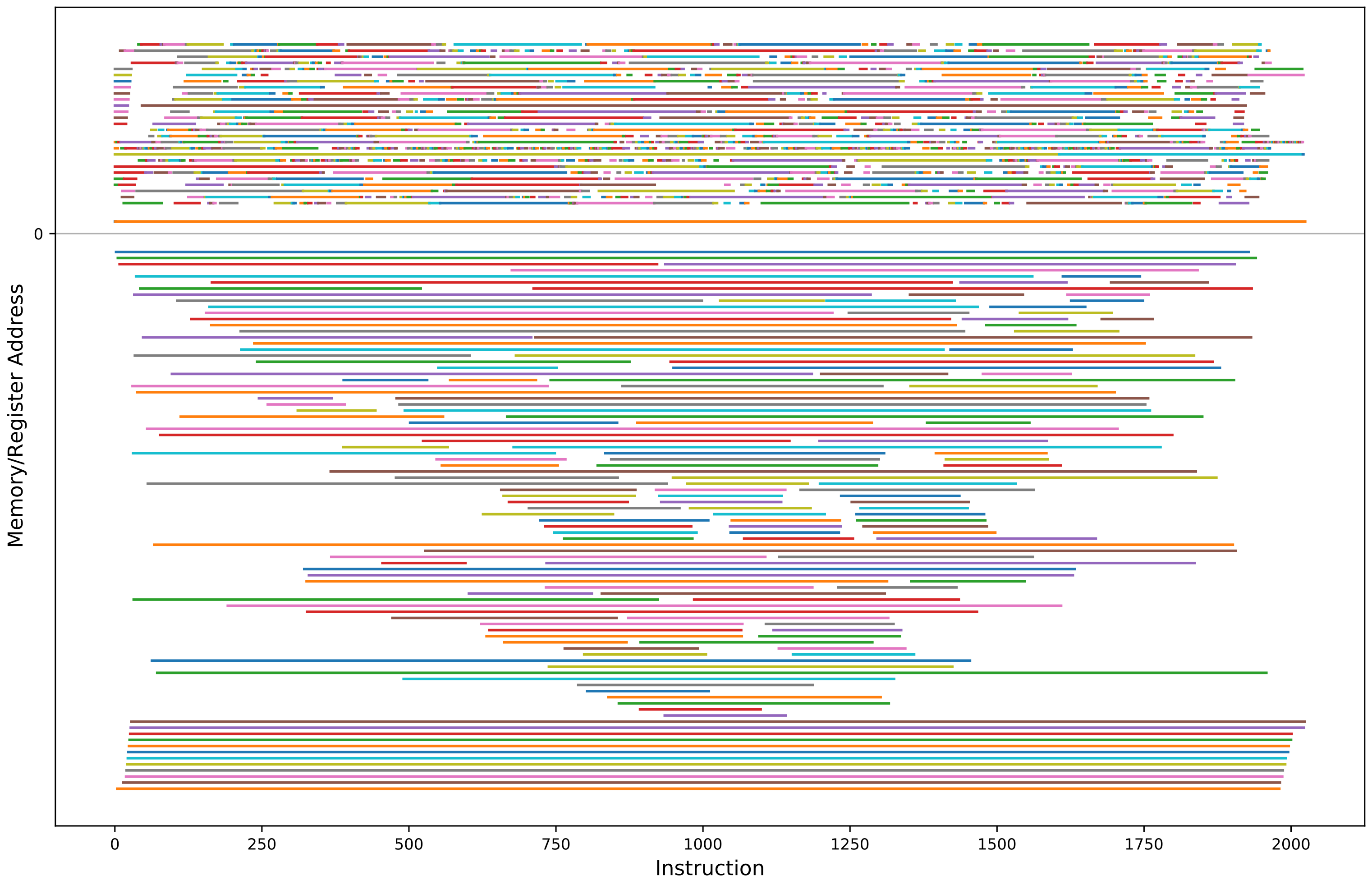


Figure 5.4: Register/Memory usage of the 8-bit MAC PSP function [used with permission] © 2020 IEEE

Chapter 6

Hardware Support for Bitslicing

Another problem facing programmers wanting to use bitsliced and PSP software is the transformation of data to its standard representation into slices. The operation can take a significant amount of time, which reduces overall performance. To improve the performance of transposing into and out of bitsliced format, we introduce a hardware accelerator with DMA capability called the tDMA. That is, the tDMA is given a source memory address of data in one format and produced data in the other format at a destination address. The processor needs only to specify the location, format, and shape of the data; start the transfer; and wait for an interrupt when the transfer is complete. Beyond accelerating the transposition, the tDMA also supports the creation of slices with masking and redundancy that are compatible with the SKIVA architecture [8]. This chapter will discuss the design, implementation and performance details, and algorithmic analysis of the tDMA.

6.1 Design Overview

The primary motivation for the tDMA is improving the interface between the standard representation of data and the bitsliced representation in SKIVA though it can be used effectively in many different architectures. The improvements it makes in particular are:

1. Acceleration of the transposition. Because the transformation is a permutation of

bits, a hardware implementation is very efficient. In software the operation is done via bit manipulation instructions while a permutation of wires is all that is needed in hardware. This operation becomes the core structure of the tDMA datapath: the transposer.

2. Masking and redundancy creation and removal. To achieve its security goals, the SKIVA architecture utilizes masking and redundancy built into the bitsliced format. See Figure 2 of [8] for details about such aggregations. The tDMA supports the automatic and efficient production of these aggregations. It also handles the transformation back to standard representation, which for SKIVA includes a check for redundancy errors that could be caused by fault injection.

To accomplish these goals, the design shown in Figure 6.1 was created. The tDMA was designed to strike a balance between performance and resource utilization so that it is suitable for inclusion in embedded designs. As such, AXI4-Lite [57] interfaces were chosen to connect the peripheral to a SoC's memory bus. Two interfaces are included in the design. The slave interface receives programming from the processor, such as the source address, destination address, and other parameters. The master interface is how the tDMA accesses memory for the bitslicing transformation. The final port on the device is an IRQ that signals to the processor that a transfer has completed or an error has occurred. Errors can either be "Redundancy Errors" where a redundancy check failed or "AXI Errors" that happen if an error occurs on the AXI4-Lite bus. The processor can read the tDMA's status register to confirm no errors occurred.

The IRQ signal is generated by the Controller module. This module directs the operation of the other submodules and also generates addresses for the AXI interface. The controller implements the "stride algorithm" for address generation that is shown in Figure 6.2 and

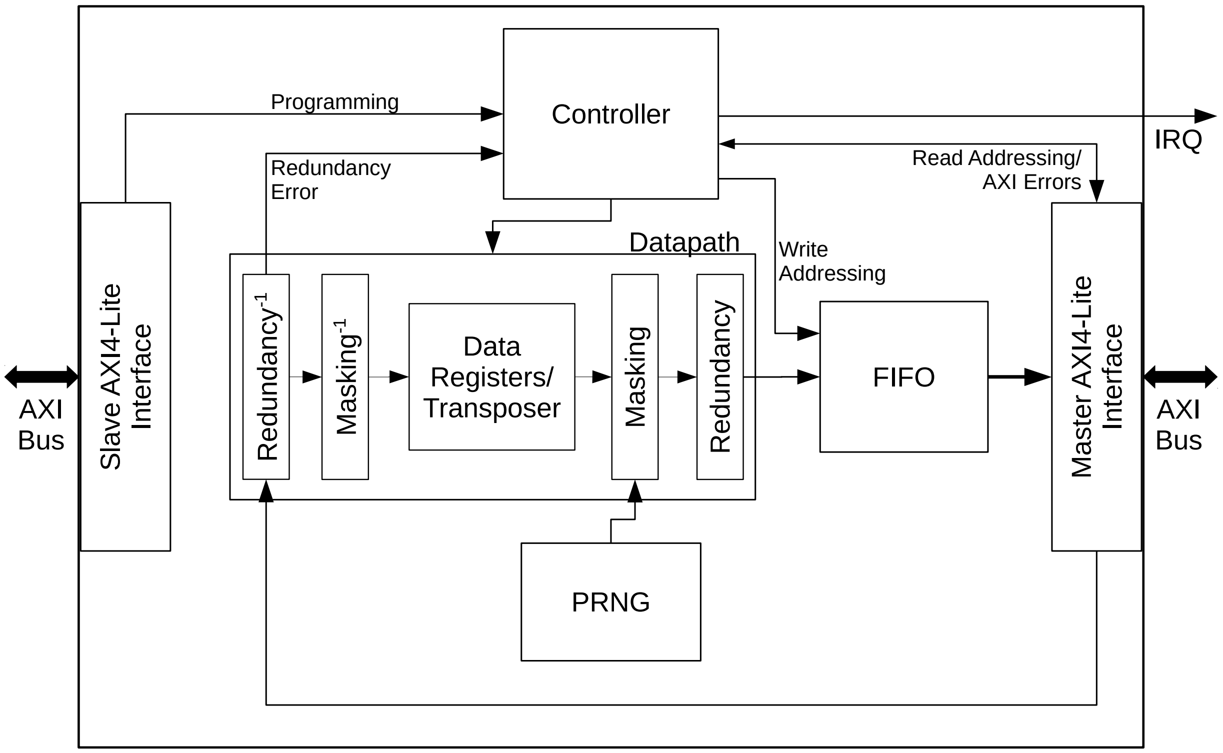


Figure 6.1: tDMA block diagram

explained in detail in the next section.

Let us now turn our attention to the datapath. The core of it is the transposer, which also includes a file of 32 registers each 32 bits wide. The file is this size because the tDMA is designed for 32-bit systems. Unlike traditional DMA controllers that typically try to push data words through the device as quickly as possible, because bits for each slice come from multiple words in memory, a file of registers is needed to store all the words before a single slice is created. Once the words for transposition are present, multiple slices can be pulled from the registers and written into memory. Though before this happens, the slices individually pass through a masking stage and then a redundancy stage.

In the masking stage three possible outputs are possible based on the parameter $D \in \{1, 2, 4\}$ (which corresponds to the variable d from Section 2.2) from SKIVA. If $D = 1$ then no masking is done. Because random values are needed (up to 24 bits per 32-bit slice) to create masks, a PRNG from [58] is included in the design. The processor seeds the PRNG before masking is needed. The output of this stage is passed to the redundancy stage.

In the redundancy stage, the parameter $R_s \in \{1, 2, 4\}$ (as defined in the SKIVA architecture) decides how much of the slice is redundant. Specifically, the 32-bit slice holds $32/R_s$ bits of information. The rest is redundant copies of those bits in one of two forms. The tDMA supports both "direct" redundancy, which is a direct copy of the bits, or "complementary" redundancy, which stores the original as well as a bitwise inverted copy. The slice is then passed into a FIFO along with its address for the AXI interface to process.

Because D and R_s are controlled independently and masking and redundancy do not interfere with each other, these two stages are independent. This is also true for the reverse operation by which slices are transposed back to words. First, redundancy is checked (that is, checking if the copies agree with each other) and discarded. Next, masks are recombined, and the slice is written into the register file.

6.2 Formal Verification

Verifying the correctness of hardware is extremely important. Unlike software, making changes to hardware, if already fabricated, involves replacing the physical device. To gain a high confidence in the correctness of the tDMA, core parts of the Controller's algorithm were verified using the TLC model checker. During the design process and before any HDL for the Controller was written, the algorithm for slice and address generation was described in the TLA$^+$ language. This description is fully included in Appendix A.

6.2.1 Stride Algorithm

The stride algorithm is the name given to the algorithm the tDMA uses to generate slices and addresses. Its name comes from the strides an address pointer takes through memory when processing words longer than 32 bits. Note that *words* in this section refer to distinct elements to transpose rather than fixed-size 32-bit words. The size is specified when needed. For example, a 128-bit plaintext is what we call a word despite it requiring four 32-bit memory locations to store. See how words are identified in Figure 6.2. One of the core features of the tDMA is that it produces slices for any number of words less than 2^{16} of any length less than 2^{12} bits despite only having a 32×32-bit register file. It accomplishes this by working in chunks of 32-bit words called *offsets*. Figure 6.2 shows six offsets as lightly shaded boxes. Note that these 32-bit words are not at consecutive memory addresses. They occur, rather, at an "offset" of the starting address. Because the tDMA is designed for 32-bit systems, the 32-bit word is the base unit of memory that is designed around. It only accesses/writes memory as 32-bit words. The tDMA starts its processing by loading from the source buffer. Once the register file has been filled with an offset, the tDMA stops loading and switches to an unload state in which the registers contents are transposed and written into the destination buffer at consecutive memory addresses. Once all the data has been unloaded, the algorithm resumes loading.

However, it is not sufficient to only stride through memory. When the source buffer's words are longer than 32 bits (this length is referred to as *WordLength*), it takes up more than one 32-bit word in memory. The tDMA assumes a little-endian data format for such occasions. That is, bits 31 down to 0 are at the 0th address, bits 63 down to 32 are at the 1st address, . . . bits $32n+31$ down to $32n$ are at the nth address. Since the processor is expected to work on 32 words at a time (it has 32 bits for PSP after all), once an offset has been transposed and unloaded, the algorithm loads the next offset from the same 32 words. These 32 words are referred to as a *block*. Only once all the offsets of a block have been processed does the algorithm move to the next block.

The stride algorithm also works in the reverse direction, but the source buffer is read consecutively while the algorithm strides through the destination buffer.

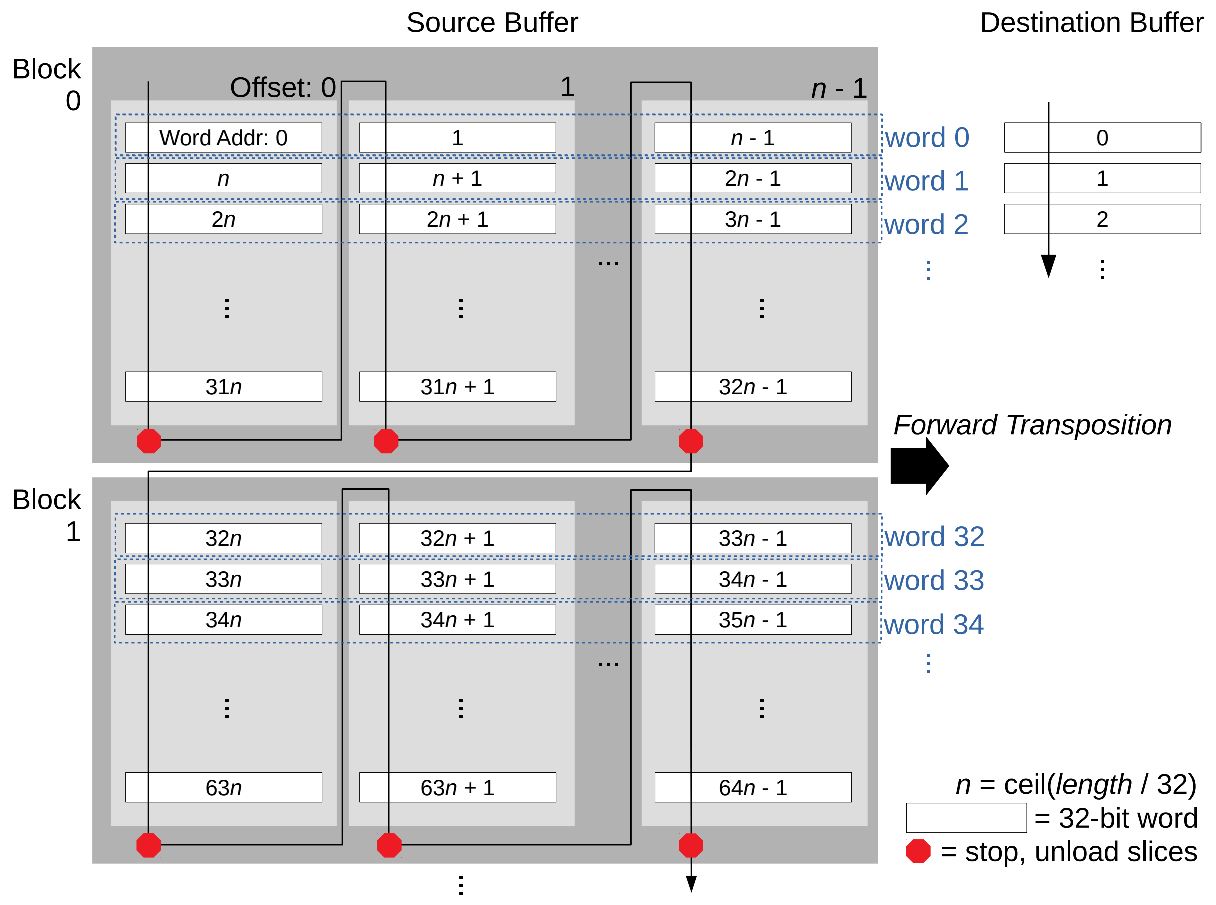


Figure 6.2: tDMA stride algorithm

6.2.2 Correctness

To show that the stride algorithm is correct for both forward and backward transformations, the formula *Correctness* in Figure 6.3 is shown to be an invariant of the model.

```
$Correctness \triangleq$
    $DMAState =$ "DONE" $\implies$
        $\land \forall addr \in$ DOMAIN $SrcMemReadCount$ :
            $SrcMemReadCount[addr] = 1$
        $\land \forall addr \in$ DOMAIN $DstMem$ :
            $DstMem[addr] \neq \{\}$
        $\land$ IF $Direction =$ "FORWARD" THEN
              $\forall addr \in$ DOMAIN $SrcMemReadCount$ :
                 $\land \forall dstaddr \in MappedDstAddrs(addr)$ :
                     $addr \in DstMem[dstaddr]$
                 $\land \forall dstaddr \in$ DOMAIN $DstMem \setminus MappedDstAddrs(addr)$ :
                     $addr \notin DstMem[dstaddr]$
            ELSE    $Direction =$ "REVERSE"
              $\forall addr \in$ DOMAIN $DstMem$ :
                 $\land \forall dstaddr \in MappedDstAddrs(addr)$ :
                     $dstaddr \in DstMem[addr]$
                 $\land \forall dstaddr \in$ DOMAIN $SrcMemReadCount \setminus MappedDstAddrs(addr)$ :
                     $dstaddr \notin DstMem[addr]$
```

Figure 6.3: Correctness TLA$^+$ formula

This invariant states that when the tDMA has finished a transfer, all source memory addresses have been read exactly once; every destination address has been written to; and data from the correct source addresses have ended up in the correct destination addresses and no incorrect data is present. These last two requirements use the *MappedDstAddrs* function which maps a source address to a set of destination addresses where bits from that source address should be located. The model abstracts the actual words and slices to just their memory addresses, which is sufficient to track data through the algorithm. By running the model checker on many possible *WordLength*s and word counts, we can be confident that the algorithm is correct. Unfortunately, due to the explosion of the state space for large *WordLength*s and *WordCount*s, all possible combinations of input configurations could not be checked. However, the most important section of the initial state space (identified through reasoning in later sections), the transposition of a $m \times n$-bit matrix where $m \leq 32$, was verified for $n \leq 512$ and all values of D, R_s in both directions.

6.2.3 Liveness

We have shown that the algorithm produces correct results when it finishes, but how do we know it will ever finish? The temporal formula in Figure 6.4 states that "eventually the model will finish."

$$\begin{array}{l} \mathit{Liveness} \triangleq \\ \quad \Diamond(\mathit{DMAState} = \text{"}\mathsf{DONE}\text{"}) \end{array}$$

Figure 6.4: Liveness TLA$^+$ temporal formula

One of the strengths of model checking is the ability to show this is the case. That is, in any state the model checker reaches, it eventually progresses to completion. There are no deadlocks or cycles in the algorithm that prevent it from ending and interrupting the processor.

6.2.4 Implementation

The implementation of the tDMA's Controller is of interest because the TLA$^+$ description of the algorithm is model checked, but it is implemented in SystemVerilog. How can we be confident the implementation has the same behavior as the model? Model checking shows that the algorithm is well-considered and assists the design process, but we admit there is no proof that shows the implementation behaves the same. However, there are reasons that support belief in the correctness of the implementation:

- Both TLA$^+$ and SystemVerilog describe the algorithm as a state machine with similar semantics. That is, there are similar ways to express the same meaning in both languages in many cases. Thus, the TLA$^+$ description of the algorithm nearly can be directly translated into SystemVerilog with only syntax changes. For example, compare the code snippets in Figure 6.5 that contain the same part of the algorithm in the two languages.

```
IF $Direction = \text{"FORWARD"}$ THEN
    $\land$ $DstMemPtr' = DstMemPtr + 1$
    $\land$ IF $DstBit + 1 < Min(32, DataWordLength - (32 * Offset))$ THEN
        $\land$ $DstBit' = DstBit + 1$
        $\land$ UNCHANGED $\langle InternalPtr, DataWordCount, DMAState, Offset,$
                    $InternalMem, BasePtr\rangle$
      ELSE
        $\land$ $DstBit' = 0$
        $\land$ IF $InternalPtr + WordsUsedPerSlice < BlockSize$ THEN
            $\land$ $InternalPtr' = InternalPtr + WordsUsedPerSlice$
            $\land$ UNCHANGED $\langle DataWordCount, DMAState, Offset, InternalMem\rangle$
```

(a) TLA$^+$ implementation

```
if(direction_i == 1'b0) // forward
begin
    next_dst_mem_ptr = dst_mem_ptr + 30'd1;
    if(dst_bit + 6'd1 < bits_left_in_offset)
    begin
        next_dst_bit = dst_bit + 6'd1;
    end
    else
    begin
        next_dst_bit = 6'd0;
        if(internal_ptr + words_per_slice < block_size)
        begin
            next_internal_ptr = internal_ptr + words_per_slice;
        end
```

(b) SystemVerilog implementation

Figure 6.5: Code snippets of the same part of the stride algorithm in TLA$^+$ and SystemVerilog

This similarity allows some visual verification of the implementation. Though this is a weak form of verification, having the semantic similarity lets the designer more easily implement the algorithm and reason about what is must be different in the implementation.

- Traditional verification of the tDMA supports its correctness via testing with a variety of test cases. The tDMA was integrated with a MicroBlaze soft processor core and Zynq-7000. Some of these tests will be detailed in the next section.

6.3 Implementation Results

This section will go over results from implementing the tDMA, including performance metrics of its operation. It was implemented for Artix-7 FPGAs in all the cases the section will discuss. The implementation shows reasonable resource utilization in Table 6.1 for multiple FIFO depth options. Most of the FF usage comes from the internal register file and the FIFO while the LUT usage primarily comes from the controller and datapath. The number of LUTs and FFs can be reduced by reducing the depth of the FIFO at a cost to performance. A FIFO depth of 32 was used for all performance numbers presented in this work.

Table 6.1: Resource utilization of tDMA with various FIFO depths on an Artix-7 FPGA

FIFO Depth	Look-Up Tables (LUTs)	Flip-Flops (FFs)
8	1,259	1,866
16	1,402	2,383
32	1,704	3,414

To benchmark the performance of this implementation, the tDMA was integrated with a MicroBlaze soft processor core on an Arty A7 [59]. The processor was configured for performance with instruction and data cache enabled along with a barrel shifter. In this experimental setup (displayed in Figure 6.6) the processor and tDMA have access to main memory located in DDR3L and a smaller on-chip Block RAM memory. This smaller memory is where the `.data` code segment was configured to be stored.

On this setup we compare the most basic form of the bitsliced transposition (32 words of 32 bits each with $D = 1, R_s = 1$) in software with the tDMA. The results of this experiment (Table 6.2) show the tDMA outperforming both the naïve and the recursive algorithm (discussed in Section 3.3). Naturally, the recursive algorithm is much faster than the naïve approach, but it still outpaced by the single-cycle slice generation of the tDMA. The primary source of latency of the tDMA is reading and writing memory. In this case, the transposition itself only takes 32 cycles. The rest of the cycles come from programming the transfer, memory access, and the MicroBlaze responding to the interrupt at the end. It should be noted that if the transfer is done only in DDR3L instead of using on-chip memory then the recursive algorithm using cache is slightly faster. All the implementations have similar code. The tDMA does not have a smaller `.text` segment than the rest due to programming transfers and configuring the interrupt controller.

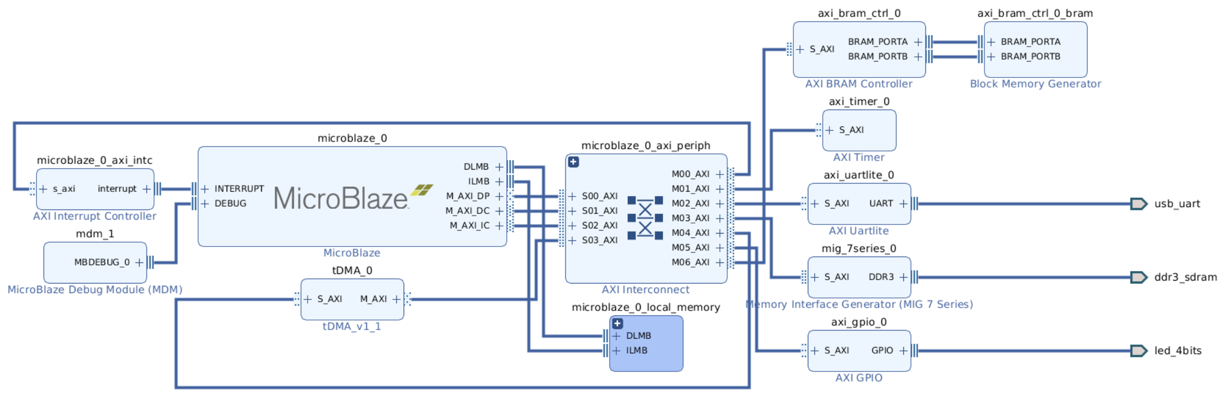


Figure 6.6: tDMA integrated with a MicroBlaze soft processor core on an Arty A7

The next experiment we will discuss is a more realistic case. We accelerate the transformation

Table 6.2: 32 × 32-bit matrix transposition for tDMA vs. software on a MicroBlaze soft processor core (with enabled barrel shifter)

	`.text` Size (bytes)	Transposition Cycle Count
Naïve	78,432	14,690
Recursive	83,316	1,478
tDMA	80,208	1,010

of plaintext, key, and ciphertext for the MicroBlaze AES example presented in Chapter 4. In total there are two 32 × 128-bit forward transpositions (plaintext and key) and one reverse (ciphertext) that are included in the measurement. We compare the tDMA with the recursive software algorithm. For this experiment, both of the `.data` and `.bss` segments were put in on-chip memory. This produced the best results for both implementations. Table 6.3 shows the tDMA outperforming the software transposition by a larger margin with similar code sizes. The overhead for the transposition is reduced by the tDMA to nearly negligible levels.

Table 6.3: Transposition using tDMA vs. software for AES-128 bitsliced code on a MicroBlaze soft processor core (with enabled barrel shifter)

	`.text` Size (bytes)	Transposition Cycle Count	Overhead (% of total cycles)
Recursive	113,416	44,629	3.9%
tDMA	113,600	7,063	0.6%
Relative Change	0.2%	−84.2%	-

The last experiment shows a limitation of the tDMA when used with processors with a word width greater than 32 bits. The tDMA was integrated with the ARM Cortex-A9 in a Zynq-7000 on a Cora Z7 [60] as shown in Figure 6.7 for the GIFTb-128 NEON SIMD PSP example in Chapter 4. This experiment is like the one for AES transposition in that plaintext, key, and ciphertext are transposed. However, instead of each of these being 32 × 128-bit transpositions, each is 128 × 128-bit. The tDMA is compared to a naïve implementation of the transposition using standard ARM instruction operating on 32-bit words and the recursive algorithm implemented using 128-bit NEON SIMD instructions. Once again, the `.data` and `.bss` segments are mapped to on-chip memory. The results shown in Table 6.4 demonstrate that the tDMA is outmatched by the recursive algorithm when the data word is wider than 32 bits but not wider than the processor's word, though the naïve algorithm still performs significantly worse and is included just for context. See the following section about algorithmic analysis for details. The tDMA is also limited in this experiment due to operating frequency. The Cortex-A9 was clocked at 650 MHz while the tDMA was at its maximum operating frequency of 120 MHz. The clock cycles counted in Table 6.4 are processor cycles. Another source of inefficiency in this experiment is the tDMA's AXI4-Lite interfaces. It could benefit through the use of burst transfers, such as those in the full AXI4 standard. AXI4-Lite interfaces were chosen for their extremely low area usage and because the TLA$^+$ model considers single transfers only.

Table 6.4: GIFTb-128 bitsliced transposition on an ARM Cortex-A9 targeting NEON SIMD instructions using tDMA vs. software

	`.text` Size (bytes)	Transposition Cycle Count	Overhead (% of total cycles)
Naïve	67,296	408,516	70.4%
Recursive	67,760	37,636	18.0%
tDMA	73,964	179,166	51.1%

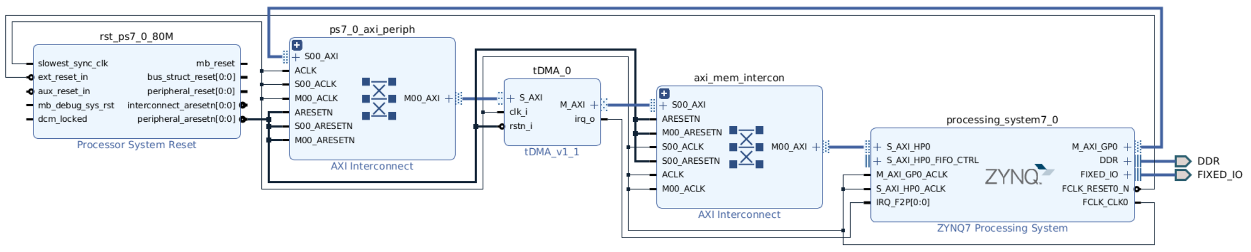


Figure 6.7: tDMA integrated in a Zynq-7000

Though the tDMA had sub-optimal performance in the last experiment, there are situations in which designers might still include it in this setup. One such case is using it for share generation for masked implementations. In place of accessing another hardware RNG or generating pseudorandom values in software, the tDMA's internal PRNG is integrated with its transposition algorithm. This eliminates any need for preprocessing of data by the processor.

6.4 Algorithmic Analysis

The computational complexity of the stride algorithm (and the tDMA in general) deserves some analysis. On one hand the stride algorithm has the property that each memory location in the source and destination buffers is only accessed a single time. On the other, we saw in the last section how it was outperformed by the recursive algorithm in software. As mentioned in Section 3.3, the recursive algorithm has a $O(n \log n)$ time complexity for an $n \times n$-bit matrix transposition. This result, however, is predicated upon the availability of operations on n-bit (or wider) registers. Assume, however, we are executing this algorithm on an m-bit processor and that $n > m$. We are no longer interested in the $n \times n$-bit matrix transposition, but rather the $m \times n$-bit case since we can only execute m PSP "threads" at a time. Each pass through the matrix until all $m \times m$ matrices are transposed now includes a factor of n/m since the processor can only work on m bits of each row at a time. The time complexity of this part of the algorithm is thus

$$O\left(\frac{n}{m} m \log m\right) = O(n \log m).$$

The remaining passes of the algorithm no longer use bitwise instructions and shifts to transpose sub-matrices. Instead, the processor must swap m-bit words. It is clear upon review of the algorithm that one half of the bits are swapped each pass, meaning one half of the m-bit words are swapped in these passes. Each requires

$$\frac{1}{2}\left\lceil\frac{n}{m}\right\rceil m = O(n)$$

swaps. Each pass, whether in the first group of $\log_2(m)$ passes or the second, requires $O(n)$ operations. Because there are $\log_2(\lceil n/m\rceil)$ in the second group of passes, in total the recursive algorithm has $O(n\log n)$ complexity for the $m \times n$-bit transposition on an m-bit processor. If, once all $m\times m$ sub-matrices were transposed, the m-bit words could be directly reordered in a single $O(n)$ pass—which is what the stride algorithm does—the complexity becomes $O(n\log m)$.

Turning now to the same problem using the tDMA with the stride algorithm, assume an m-bit tDMA. That is, instead of a set 32-bit design, extend the tDMA to be based on an m-bit word. The tDMA's transposer allows the transposition of any matrix of size $m \times m$ in $O(m)$ operations. An $m \times n$ matrix requires $\lceil n/m\rceil$ applications of the transposer. Hence, an m-bit tDMA completes the transposition in

$$O\left(\frac{n}{m}m\right) = O(n)$$

time. There are no additional passes after the $m \times m$ transposition due to the stride algorithm. This is the true purpose of the stride algorithm: to process $m\times m$ sub-matrices in the order that produces a *fully* transposed matrix after the first pass. For larger $n\times n$ matrices, it transposes in groups of m rows of the matrix ("blocks" in Subsection 6.2.1), which is a more convenient format for an m-bit processor to use than the regular $n \times n$ transposed matrix.

This complexity result is supported by measurements of the tDMA in Figure 6.8 showing a linear asymptotic behavior. Note the steps in the function occurring every 32 bits. These are caused by the need for another application of the transposer, which requires loading 32 additional 32-bit words.

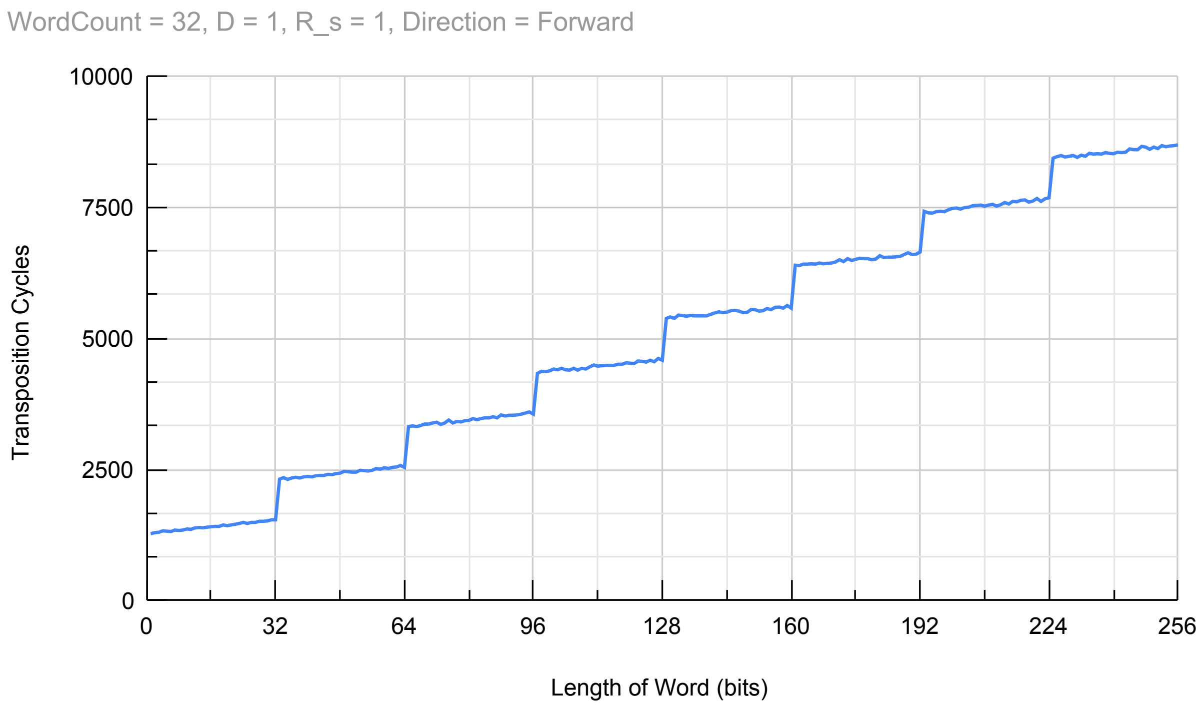


Figure 6.8: Transposition cycles vs. word length for 32 words in the forward direction

We have not yet discussed the performance of the tDMA when $D \neq 1$ or $R_s \neq 1$ in this section. Changing these parameters does not change the single pass behavior of the tDMA. It only changes how many 32-bit words are written to memory (in the forward transposition) or read from memory (in the reverse transposition). Specifically, define the constant security factor SF as the product $SF = D \cdot R_s$. For the $m \times n$-bit transposition discussed above, the tDMA writes m words of m-bits to memory each application of the transposer. In the more general case this becomes $SF \cdot m$ words since each of these only contains m/SF bits from the original matrix. Reading/writing $SF \cdot m$ words is still a $O(m)$ operation so the tDMA, in total, has a $O(n)$ time complexity for a constant SF. If this is the case, we might expect to see a $O(1)$ factor when comparing the same transposition with different values of SF. Indeed, Figure 6.9 shows this behavior for each of the possible values of SF. Note that transposing (as in this case of this figure) 256 words is equivalent to transposing eight $32 \times n$-bit blocks for the stride algorithm.

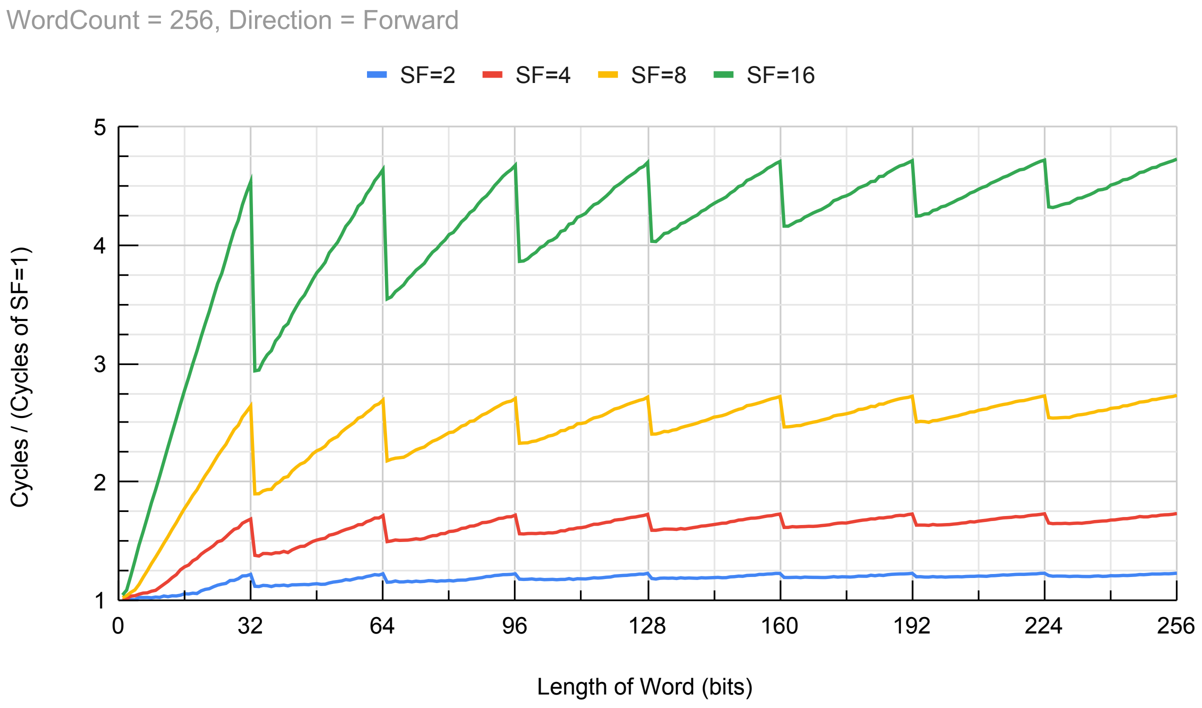


Figure 6.9: Security factor impact vs. word length for 256 words in the forward direction

Chapter 7

Conclusion

This thesis has presented two contributions that support designers and programmers wishing to use bitsliced and PSP software. Bitsliced software continues to be a field with active research, and these contributions make the technique more accessible. We have shown how programmers can describe their programs in Verilog and create well-performing PSP functions in C using the PSPCG. This includes hierarchical synthesis of some sequential logic designs. This code generation tool's value was demonstrated by its use in the creation of efficient block cipher implementations and a software matrix-vector multiplication scheme for reduced-precision operands fit for embedded devices. For the latter, we provide an analysis of the overhead created during compilation of the PSP functions. In addition to the code generation problem, the thesis addresses slice generation by introducing the tDMA to accelerate the transposition of bit-matrices. It also includes support for the formats of slices used for side-channel and fault attack resistance in the SKIVA architecture [8]. With the correctness of hardware being critical, the core parts of the control algorithm were described in TLA$^+$ and verified using model checking. The benefits of this hardware accelerator over software methods were shown in concrete examples and algorithmic analysis.

The results in this thesis give some promise that the techniques and tools described herein are worthy of further study. In the future, we intend to explore the following research directions:

- Inclusion of the tDMA in a RISC-V based system. Firstly, this would integrate the tDMA with the matrix-vector multiplication work from Chapter 5. Secondly, the argument for the inclusion of a hardware accelerator for a specific type of processing into a system is more convincing when the architecture itself is open and flexible, like RISC-V.

- Use of the PSPCG and PSP techniques for other applications. There are many applications, such as those with some form of internal parallelism or work well with reduced-precision operands, that are suitable for PSP implementation. Among these include new algorithms for information security in the fields of post-quantum and lightweight cryptography.

Bibliography

[1] T. S. Perry and P. Wallich, “Design case history: the Commodore 64,” *IEEE Spectrum*, vol. 22, no. 3, pp. 48–58, Mar. 1985.

[2] M. Verhelst and B. Moons, “Embedded Deep Neural Network Processing: Algorithmic and Processor Techniques Bring Deep Learning to IoT and Edge Devices,” *IEEE Solid-State Circuits Magazine*, vol. 9, no. 4, pp. 55–65, Nov. 2017.

[3] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8114708/

[4] M. Randolph and W. Diehl, “Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman,” *Cryptography*, vol. 4, no. 2, May 2020, art. no. 15. [Online]. Available: https://www.mdpi.com/2410-387X/4/2/15

[5] C. Giraud and H. Thiebeauld, “A Survey on Fault Attacks,” in *Smart Card Research and Advanced Applications VI*, ser. IFIP International Federation for Information Processing, J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A. A. El Kalam, Eds. Boston, MA, USA: Springer, 2004, vol. 153, pp. 159–176.

[6] E. Prouff and M. Rivain, “Masking against Side-Channel Attacks: A Formal Security Proof,” in *Advances in Cryptology – EUROCRYPT 2013*, ser. Lecture Notes in Computer Science, T. Johansson and P. Q. Nguyen, Eds., vol. 7881. Berlin, Heidelberg, Germany: Springer, 2013, pp. 142–159. [Online]. Available: http://link.springer.com/10.1007/978-3-642-38348-9_9

[7] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont, "Lightweight Fault Attack Resistance in Software Using Intra-instruction Redundancy," in *Selected Areas in Cryptography – SAC 2016*, ser. Lecture Notes in Computer Science, R. Avanzi and H. Heys, Eds., vol. 10532. Cham, Switzerland: Springer International Publishing, 2017, pp. 231–244. [Online]. Available: http://link.springer.com/10.1007/978-3-319-69453-5_13

[8] P. Kiaei, D. Mercadier, P.-E. Dagand, K. Heydemann, and P. Schaumont, "Custom Instruction Support for Modular Defense against Side-channel and Fault Attacks," in *Constructive Side-Channel Analysis and Secure Design (COSADE 2020)*, Oct. 2020. [Online]. Available: https://eprint.iacr.org/2020/466

[9] P. Kiaei and P. Schaumont, "Synthesis of Parallel Synchronous Software," *IEEE Embedded Systems Letters*, pp. 1–1, 2020, early access. [Online]. Available: https://ieeexplore.ieee.org/document/9085919/

[10] Xilinx, *MicroBlaze Processor Reference Guide (UG984)*, San Jose, CA, USA, Jun. 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug984-vivado-microblaze-ref.pdf

[11] ——, *Zynq-7000 SoC Technical Reference Manual (UG585)*, San Jose, CA, USA, Jul. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[12] E. Biham, "A fast new DES implementation in software," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, E. Biham, Ed., vol. 1267. Berlin, Heidelberg, Germany: Springer, 1997, pp. 260–272. [Online]. Available: http://link.springer.com/10.1007/BFb0052352

[13] *Data Encryption Standard (DES)*, ser. Federal Information Processing Standard (FIPS). National Institute of Standards and Technology, Oct. 1999, no. 46-3 (Withdrawn). [Online]. Available: https://csrc.nist.gov/publications/detail/fips/46/3/archive/1999-10-25

[14] M. Matsui and J. Nakajima, "On the Power of Bitslice Implementation on Intel Core2 Processor," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds., vol. 4727. Berlin, Heidelberg, Germany: Springer, 2007, pp. 121–134. [Online]. Available: http://link.springer.com/10.1007/978-3-540-74735-2_9

[15] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. Dray, Jr., *Advanced Encryption Standard (AES)*, ser. Federal Information Processing Standard (FIPS). National Institute of Standards and Technology, Nov. 2001, no. 197. [Online]. Available: https://www.nist.gov/publications/advanced-encryption-standard-aes

[16] E. Biham, R. Anderson, and L. Knudsen, "Serpent: A New Block Cipher Proposal," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., vol. 1372. Berlin, Heidelberg, Germany: Springer, 1998, pp. 222–238. [Online]. Available: http://link.springer.com/10.1007/3-540-69710-1_15

[17] W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede, "RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms," *Science China Information Sciences*, vol. 58, no. 12, pp. 1–15, Dec. 2015. [Online]. Available: http://link.springer.com/10.1007/s11432-015-5459-7

[18] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "GIFT: A Small Present," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, ser.

Lecture Notes in Computer Science, W. Fischer and N. Homma, Eds., vol. 10529. Cham, Switzerland: Springer International Publishing, 2017, pp. 321–345. [Online]. Available: https://eprint.iacr.org/2017/622

[19] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback, "Report on the development of the Advanced Encryption Standard (AES)," *Journal of Research of the National Institute of Standards and Technology*, vol. 106, no. 3, p. 511, May 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/jres/106/3/j63nec.pdf

[20] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology — CRYPTO' 99*, ser. Lecture Notes in Computer Science, M. Wiener, Ed., vol. 1666. Berlin, Heidelberg, Germany: Springer, 1999, pp. 388–397. [Online]. Available: http://link.springer.com/10.1007/3-540-48405-1_25

[21] L. Goubin and J. Patarin, "DES and Differential Power Analysis The "Duplication" Method," in *Cryptographic Hardware and Embedded Systems (CHES 1999)*, ser. Lecture Notes in Computer Science, Ç. K. Koç and C. Paar, Eds., vol. 1717. Berlin, Heidelberg, Germany: Springer, 1999, pp. 158–172. [Online]. Available: http://link.springer.com/10.1007/3-540-48059-5_15

[22] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks," in *Advances in Cryptology — CRYPTO' 99*, ser. Lecture Notes in Computer Science, M. Wiener, Ed., vol. 1666. Berlin, Heidelberg, Germany: Springer, 1999, pp. 398–412. [Online]. Available: http://link.springer.com/10.1007/3-540-48405-1_26

[23] Y. Ishai, A. Sahai, and D. Wagner, "Private Circuits: Securing Hardware against Probing Attacks," in *Advances in Cryptology - CRYPTO 2003*, ser.

Lecture Notes in Computer Science, D. Boneh, Ed., vol. 2729. Berlin, Heidelberg, Germany: Springer, 2003, pp. 463–481. [Online]. Available: http://link.springer.com/10.1007/978-3-540-45146-4_27

[24] E. Trichina, *Combinational Logic Design for AES SubByte Transformation on Masked Data*. Cryptology ePrint Archive, 2003, no. 236. [Online]. Available: https://eprint.iacr.org/2003/236

[25] E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds., vol. 3156. Berlin, Heidelberg, Germany: Springer, 2004, pp. 16–29. [Online]. Available: http://link.springer.com/10.1007/978-3-540-28632-5_2

[26] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert, "On the Cost of Lazy Engineering for Masked Software Implementations," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, M. Joye and A. Moradi, Eds., vol. 8968. Cham, Switzerland: Springer International Publishing, 2015, pp. 64–81. [Online]. Available: http://link.springer.com/10.1007/978-3-319-16763-3_5

[27] S. Gao, B. Marshall, D. Page, and E. Oswald, "Share-slicing: Friend or Foe?" *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 1, pp. 152–174, Nov. 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8396

[28] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology — CRYPTO '97*, ser. Lecture Notes in Computer Science, B. S. Kaliski, Ed., vol. 1294. Berlin, Heidelberg, Germany: Springer, 1997, pp. 513–525. [Online]. Available: http://link.springer.com/10.1007/BFb0052259

[29] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2013, pp. 404–409. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6513538

[30] O. Grumberg, D. Kroening, D. Peled, H. Veith, E. M. Clarke, and E. M. Clarke, Jr., *Model Checking, Second Edition*. Cambridge, MA, USA: MIT Press, 2018.

[31] L. Lamport, *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Boston, MA, USA: Addison-Wesley, 2003. [Online]. Available: https://lamport.azurewebsites.net/tla/book.html

[32] L. May, L. Penna, and A. Clark, "An Implementation of Bitsliced DES on the Pentium MMXTM Processor," in *Information Security and Privacy*, ser. Lecture Notes in Computer Science, E. P. Dawson, A. Clark, and C. Boyd, Eds., vol. 1841. Berlin, Heidelberg, Germany: Springer, 2000, pp. 112–122. [Online]. Available: http://link.springer.com/10.1007/10718964_10

[33] B. Gladman, "Serpent S Boxes as Boolean Functions," 2000. [Online]. Available: http://brg.a2hosted.com//oldsite/cryptography_technology/serpent/index.php

[34] Z. Bao, P. Luo, and D. Lin, "Bitsliced Implementations of the PRINCE, LED and RECTANGLE Block Ciphers on AVR 8-Bit Microcontrollers," in *Information and Communications Security*, ser. Lecture Notes in Computer Science, S. Qing, E. Okamoto, K. Kim, and D. Liu, Eds., vol. 9543. Cham, Switzerland: Springer International Publishing, 2016, pp. 18–36. [Online]. Available: http://link.springer.com/10.1007/978-3-319-29814-6_3

[35] T. Pornin, "Implantation et optimisation des primitives cryptographiques," Ph.D.

Thesis, Département d'Informatique de l'École Normale Supérieure, 2001. [Online]. Available: http://www.bolet.org/~pornin/2001-phd-pornin.pdf

[36] ——, "Automatic Software Optimization of Block Ciphers using Bitslicing Techniques," *Not published*, 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.3085&rep=rep1&type=pdf

[37] S. Xu and D. Gregg, "Bitslice Vectors: A Software Approach to Customizable Data Precision on Processors with SIMD Extensions," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug. 2017, pp. 442–451. [Online]. Available: http://ieeexplore.ieee.org/document/8025318/

[38] D. Mercadier, P.-É. Dagand, L. Lacassagne, and G. Muller, "Usuba: Optimizing & Trustworthy Bitslicing Compiler," in *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, Vienna, Austria, Feb. 2018, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3178433.3178437

[39] L. Wang, H. Zhao, and G. Bai, "A cost-Efficient Implementation of Public-key Cryptography on Embedded Systems," in *2007 International Workshop on Electron Devices and Semiconductor Technology (EDST)*, Jun. 2007, pp. 194–197. [Online]. Available: http://ieeexplore.ieee.org/document/4289808/

[40] D. Rossi, I. Loi, G. Haugou, and L. Benini, "Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, May 2014, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/2597917.2597922

[41] C. Zinner and W. Kubinger, "ROS-DMA: A DMA Double Buffering Method for Embedded Image Processing with Resource Optimized Slicing," in *12th IEEE*

Real-Time and Embedded Technology and Applications Symposium (RTAS'06), San Jose, CA, USA, Apr. 2006, pp. 361–372. [Online]. Available: http://ieeexplore.ieee.org/document/1613350/

[42] R. Andri, G. Karunaratne, L. Cavigelli, and L. Benini, "ChewBaccaNN: A Flexible 223 TOPS/W BNN Accelerator," *arXiv*, no. 2005.07137, Nov. 2020, preprint. [Online]. Available: http://arxiv.org/abs/2005.07137

[43] S. Ma, L. Huang, Y. Lei, Y. Guo, and Z. Wang, "An Efficient Direct Memory Access (DMA) Controller for Scientific Computing Accelerators," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, May 2019, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/8702172/

[44] G. Gaubatz and B. Sunar, "Leveraging the Multiprocessing Capabilities of Modern Network Processors for Cryptographic Acceleration," in *Fourth IEEE International Symposium on Network Computing and Applications*, Jul. 2005, pp. 235–238. [Online]. Available: http://ieeexplore.ieee.org/document/1565960/

[45] D. E. Knuth, "ANSWERS TO EXERCISES," in *The art of computer programming*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1997, vol. 3, p. 587.

[46] C. Wolf and J. Glaser, "Yosys - A Free Verilog Synthesis Suite," in *21st Austrian Workshop on Microelectronics (Austrochip 2013)*, Linz, Austria, Oct. 2013, pp. 1–6. [Online]. Available: http://www.clifford.at/yosys/files/yosys-austrochip2013.pdf

[47] R. Usselmann, "AES (Rijndael) IP Core," 2018. [Online]. Available: https://opencores.org/projects/aes_core

[48] S. Banik, A. Chakraborti, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "GIFT-COFB v1.0,"

2019, submission to the NIST Lightweight Cryptography project. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/gift-cofb-spec-round2.pdf

[49] P. Kiaei, A. S. Krishnan, and P. Schaumont, "Parallel Synchronous Code Generation for Second Round Light Weight Candidates," 2020, submission to the NIST Lightweight Cryptography project. [Online]. Available: https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/parallel-synchronous-code-generation-lwc2020.pdf

[50] Information Technology Laboratory Computer Security Division, "Lightweight Cryptography | CSRC | CSRC," Jan. 2017. [Online]. Available: https://csrc.nist.gov/Projects/Lightweight-Cryptography

[51] A. Adomnicai, Z. Najm, and T. Peyrin, "Fixslicing: A New GIFT Representation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 402–427, Jun. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8595

[52] S. Belaïd, P.-É. Dagand, D. Mercadier, M. Rivain, and R. Wintersdorff, "Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations," in *Advances in Cryptology – EUROCRYPT 2020*, ser. Lecture Notes in Computer Science, A. Canteaut and Y. Ishai, Eds., vol. 12107. Cham, Switzerland: Springer International Publishing, 2020, pp. 311–341. [Online]. Available: http://link.springer.com/10.1007/978-3-030-45727-3_11

[53] ARM, "NEON Programmer's Guide," 2013. [Online]. Available: https://static.docs.arm.com/den0018/a/DEN0018A_neon_programmers_guide_en.pdf

[54] M. G. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine Learning at the Network Edge: A Survey," *arXiv*, no. 1908.00080 [cs, stat], Sep. 2020. [Online]. Available: http://arxiv.org/abs/1908.00080

[55] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/document/7783722/

[56] "pulp-platform/pulpissimo," Dec. 2020. [Online]. Available: https://github.com/pulp-platform/pulpissimo

[57] ARM, "AMBA AXI and ACE Protocol Specification," 2013. [Online]. Available: https://static.docs.arm.com/ihi0022/d/IHI0022D_amba_axi_protocol_spec.pdf

[58] J. Strömbergson, "secworks/ca_prng," Oct. 2020. [Online]. Available: https://github.com/secworks/ca_prng

[59] Digilent, "Arty A7 [Digilent Documentation]," Jan. 2021. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/arty-a7/start

[60] ——, "Cora Z7 [Digilent Documentation]," Jan. 2021. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/cora-z7/start

Appendices

Appendix A

TLA+ Specification

This TLA$^+$ specification describes the algorithm the tDMA uses to generate slices, including where it reads from and writes to memory.

```
──────────────────── MODULE tDMA ────────────────────
EXTENDS Naturals

CONSTANTS DataWordLengths, DataWordCounts

VARIABLES InitialDataWordCount, DataWordCount, DataWordLength, SecurityFactor,
          SrcMemReadCount, DstMem, DMAState, InternalMem,
          InternalPtr, SrcMemPtr, DstMemPtr, BasePtr, Offset,
          DstBit, Direction

constantvars ≜ ⟨InitialDataWordCount, DataWordLength, SecurityFactor, Direction⟩

loadvars ≜ ⟨SrcMemReadCount, SrcMemPtr⟩

unloadvars ≜ ⟨DataWordCount, DstMemPtr, DstMem, DstBit, Offset⟩

vars ≜ ⟨DMAState, InternalMem, InternalPtr, BasePtr,
        constantvars, loadvars, unloadvars⟩

NextMultipleOf(i, j) ≜
   IF i%j = 0 THEN i
    ELSE i + j − (i%j)

Ceil(i, j) ≜
    NextMultipleOf(i, j) ÷ j

WordsUsedPerSlice ≜ 32 ÷ SecurityFactor

Stride ≜
   Ceil(DataWordLength, 32)

SrcMemSize ≜
   IF Direction = "FORWARD" THEN
```

```
      Stride ∗ InitialDataWordCount
    ELSE
      DataWordLength ∗ Ceil(InitialDataWordCount, WordsUsedPerSlice)

DstMemSize ≜
  IF Direction = "FORWARD" THEN
      DataWordLength ∗ Ceil(InitialDataWordCount, WordsUsedPerSlice)
    ELSE
      Stride ∗ InitialDataWordCount

TypeOK ≜
    ∧  InitialDataWordCount ∈ DataWordCounts
    ∧  DataWordCount ∈ (0 . . InitialDataWordCount)
    ∧  DataWordLength ∈ DataWordLengths
    ∧  SecurityFactor   ∈ {1, 2, 4, 8, 16}
    ∧  Direction ∈ {"FORWARD", "REVERSE"}
    ∧  SrcMemReadCount ∈ [(0 . . SrcMemSize − 1) → Nat]
    ∧  DstMem ∈ [(0 . . DstMemSize − 1) → SUBSET Nat]
    ∧  DMAState ∈ {"LOAD", "UNLOAD", "DONE"}
    ∧  InternalMem ∈ [(0 . . 31) → (0 . . SrcMemSize)]
    ∧  InternalPtr   ∈ (0 . . 31)
    ∧  SrcMemPtr ∈ (0 . . SrcMemSize)
    ∧  DstMemPtr ∈ (0 . . DstMemSize)
    ∧  BasePtr ∈ Nat
    ∧  Offset  ∈ Nat
    ∧  DstBit ∈ (0 . . 31)

Init ≜
    ∧ InitialDataWordCount ∈ DataWordCounts
    ∧ DataWordCount = InitialDataWordCount
    ∧ DataWordLength ∈ DataWordLengths
    ∧ SecurityFactor   ∈ {1, 2, 4, 8, 16}
    ∧ Direction ∈ {"FORWARD", "REVERSE"}
    ∧ SrcMemReadCount = [i ∈ (0 . . SrcMemSize − 1) ↦ 0]
    ∧ DstMem = [i ∈ (0 . . DstMemSize − 1) ↦ {}]
    ∧ DMAState = "LOAD"
    ∧ InternalMem = [i ∈ (0 . . 31) ↦ SrcMemSize]
    ∧ InternalPtr   = 0
    ∧ SrcMemPtr = 0
    ∧ DstMemPtr = 0
    ∧ BasePtr = 0
    ∧ Offset  = 0
    ∧ DstBit = 0
```

```
$ReadMem(addr) \triangleq$
    $SrcMemReadCount' = [SrcMemReadCount$
                              $\text{EXCEPT } ![addr] = SrcMemReadCount[addr] + 1]$

$LoadInternalMem(memAddr) \triangleq$
    $InternalMem' = [InternalMem \text{ EXCEPT } ![InternalPtr] = memAddr]$

$BlockSize \triangleq$
    $\text{IF } DataWordCount \geq 32 \text{ THEN } 32$
     $\text{ELSE } DataWordCount$

$Min(a, b) \triangleq$
     $\text{IF } a < b \text{ THEN } a$
      $\text{ELSE } b$

$UpdateLoadState \triangleq$
    $\text{IF } Direction = \text{“FORWARD” THEN}$
        $\text{IF } InternalPtr + 1 < BlockSize \text{ THEN}$
            $\land InternalPtr' = InternalPtr + 1$
            $\land SrcMemPtr' = SrcMemPtr + Stride$
            $\land \text{UNCHANGED } \langle DMAState, BasePtr \rangle$
         $\text{ELSE}$
            $\land InternalPtr' = 0$
            $\land DMAState' = \text{“UNLOAD”}$
            $\land \text{IF } Offset + 1 < Stride \text{ THEN}$
                $\land SrcMemPtr' = BasePtr + Offset + 1$
                $\land \text{UNCHANGED } \langle BasePtr \rangle$
               $\text{ELSE}$
                $\land SrcMemPtr' = SrcMemPtr + 1$
                $\land BasePtr' = SrcMemPtr + 1$
     $\text{ELSE}$    $Direction = \text{“REVERSE”}$
        $\land SrcMemPtr' = SrcMemPtr + 1$
        $\land \text{IF } InternalPtr + 1 < Min(32, DataWordLength - (32 * Offset)) \text{ THEN}$
              $\land InternalPtr' = InternalPtr + 1$
              $\land \text{UNCHANGED } \langle DMAState, BasePtr \rangle$
           $\text{ELSE}$
              $\land InternalPtr' = 0$
              $\land DMAState' = \text{“UNLOAD”}$
              $\land \text{UNCHANGED } BasePtr$

$Load \triangleq$
    $\land DMAState = \text{“LOAD”}$
    $\land ReadMem(SrcMemPtr)$
    $\land LoadInternalMem(SrcMemPtr)$
    $\land$ $UpdateLoadState$
    $\land$ UNCHANGED $\langle unloadvars \rangle$

$WriteSlice(addr) \triangleq$
   $DstMem' = [DstMem$ EXCEPT $![addr] =$
      $\{InternalMem[i]$ :
         $i \in (InternalPtr$ ..
            $Min(InternalPtr + WordsUsedPerSlice - 1, 31))\}]$

$WriteRegular(addr) \triangleq$
   $DstMem' = [DstMem$ EXCEPT $![addr] =$
      $\{InternalMem[i] : i \in (0 \,..\, 31)\}]$

$ClearInternalMem \triangleq$
   $InternalMem' = [i \in (0 \,..\, 31) \mapsto SrcMemSize]$

$UpdateUnloadState \triangleq$
   IF $Direction =$ "FORWARD" THEN
      $\land$ $DstMemPtr' = DstMemPtr + 1$
      $\land$ IF $DstBit + 1 < Min(32, DataWordLength - (32 * Offset))$ THEN
            $\land$ $DstBit' = DstBit + 1$
            $\land$ UNCHANGED $\langle InternalPtr, DataWordCount, DMAState, Offset,$
                           $InternalMem, BasePtr \rangle$
         ELSE
            $\land$ $DstBit' = 0$
            $\land$ IF $InternalPtr + WordsUsedPerSlice < BlockSize$ THEN
                  $\land$ $InternalPtr' = InternalPtr + WordsUsedPerSlice$
                  $\land$ UNCHANGED $\langle DataWordCount, DMAState, Offset, InternalMem \rangle$
               ELSE
                  $\land$ $InternalPtr' = 0$
                  $\land$ $ClearInternalMem$
                  $\land$ IF $Offset + 1 < Stride$ THEN
                        $\land$ $Offset' = Offset + 1$
                        $\land$ $DMAState' =$ "LOAD"
                        $\land$ UNCHANGED $DataWordCount$
                     ELSE
                        $\land$ $Offset' = 0$
                        $\land$ $DataWordCount' = DataWordCount - BlockSize$
                        $\land$ IF $DataWordCount - BlockSize = 0$ THEN
                              $DMAState' =$ "DONE"
                           ELSE
                              $DMAState' =$ "LOAD"
      $\land$ UNCHANGED $BasePtr$
   ELSE    $Direction =$ "REVERSE"
```

```
    IF $DstBit + InternalPtr + 1 < BlockSize$ THEN
        $\land DstMemPtr' = DstMemPtr + Stride$
        $\land$ IF $InternalPtr + 1 < WordsUsedPerSlice$ THEN
            $\land InternalPtr' = InternalPtr + 1$
            $\land$ UNCHANGED $\langle DstBit, DMAState, InternalMem \rangle$
          ELSE
            $\land InternalPtr' = 0$
            $\land DstBit' = DstBit + WordsUsedPerSlice$
            $\land DMAState' =$ "LOAD"
            $\land ClearInternalMem$
        $\land$ UNCHANGED $\langle DataWordCount, Offset, BasePtr \rangle$
    ELSE
        $\land InternalPtr' = 0$
        $\land DstBit' = 0$
        $\land ClearInternalMem$
        $\land$ IF $Offset + 1 < Stride$ THEN
            $\land Offset' = Offset + 1$
            $\land DstMemPtr' = BasePtr + Offset + 1$
            $\land DMAState' =$ "LOAD"
            $\land$ UNCHANGED $\langle DataWordCount, BasePtr \rangle$
          ELSE
            $\land Offset' = 0$
            $\land DstMemPtr' = DstMemPtr + 1$
            $\land DataWordCount' = DataWordCount - BlockSize$
            $\land$ IF $DataWordCount - BlockSize = 0$ THEN
                $\land DMAState' =$ "DONE"
                $\land$ UNCHANGED $BasePtr$
              ELSE
                $\land DMAState' =$ "LOAD"
                $\land BasePtr' = DstMemPtr + 1$

$Unload \triangleq$
    $\land\ DMAState =$ "UNLOAD"
    $\land$ IF $Direction =$ "FORWARD" THEN
        $WriteSlice(DstMemPtr)$
      ELSE  $Direction =$ "REVERSE"
        $WriteRegular(DstMemPtr)$
    $\land\ UpdateUnloadState$
    $\land$ UNCHANGED $\langle loadvars \rangle$

$Next \triangleq$
    $\land \lor Load$
       $\lor Unload$
    $\land$ UNCHANGED $constantvars$

$Fairness \triangleq$
    $\text{WF}_{vars}(Next)$

$Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness$

$SizeOfFullBlock \triangleq$
    $SecurityFactor * DataWordLength$

$SizeOfRegularOffset(block) \triangleq$
    LET $blockWordCount \triangleq Min(32, InitialDataWordCount - block * 32)$IN
        $32 * Ceil(blockWordCount, WordsUsedPerSlice)$

$DataWordAddr(addr) \triangleq$
    $addr \div Stride$

$WordAddrInBlock(addr) \triangleq$
    $DataWordAddr(addr)\%32$

$SlicesPerWord(offset) \triangleq$
    IF $offset = Stride - 1$ THEN
        IF $DataWordLength\%32 = 0$ THEN
            $32$
          ELSE
            $DataWordLength\%32$
      ELSE
        $32$

$StartDstAddress(addr) \triangleq$
    LET $offset \triangleq addr\%Stride$IN
        LET $blockAddr \triangleq addr \div (32 * Stride)$IN
            $blockAddr * SizeOfFullBlock +$
            $offset * SizeOfRegularOffset(blockAddr) +$
            $(WordAddrInBlock(addr) \div WordsUsedPerSlice)$
                $* SlicesPerWord(offset)$

$MappedDstAddrs(addr) \triangleq$
    $(StartDstAddress(addr)$
        $\,..\, StartDstAddress(addr) + SlicesPerWord(addr\%Stride) - 1)$

$Correctness \triangleq$
    $DMAState = \text{“DONE”} \implies$
        $\land \forall\, addr \in$ DOMAIN $SrcMemReadCount :$
            $SrcMemReadCount[addr] = 1$
        $\land \forall\, addr \in$ DOMAIN $DstMem :$
            $DstMem[addr] \neq \{\}$
    $\land$ IF $Direction = $ "FORWARD" THEN
          $\forall\, addr \in$ DOMAIN $SrcMemReadCount$ :
             $\land\ \forall\, dstaddr \in MappedDstAddrs(addr)$ :
                 $addr \in DstMem[dstaddr]$
             $\land\ \forall\, dstaddr \in$ DOMAIN $DstMem \setminus MappedDstAddrs(addr)$ :
                 $addr \notin DstMem[dstaddr]$
       ELSE   $Direction = $ "REVERSE"
          $\forall\, addr \in$ DOMAIN $DstMem$ :
             $\land\ \forall\, dstaddr \in MappedDstAddrs(addr)$ :
                 $dstaddr \in DstMem[addr]$
             $\land\ \forall\, dstaddr \in$ DOMAIN $SrcMemReadCount \setminus MappedDstAddrs(addr)$ :
                 $dstaddr \notin DstMem[addr]$

$Liveness \triangleq$
    $\Diamond(DMAState = $ "DONE"$)$
```

Appendix B

Citations of Copyrighted Works

Table 5.1, Table 5.2, Figure 5.3, Figure 5.4 [used with permission]

R. Singh, T. Conroy, and P. Schaumont, “Variable Precision Multiplication for Software-Based Neural Networks,” in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, Sep. 2020, pp. 4-6. [Online]. Available: https://ieeexplore.ieee.org/document/9286170/ Used with permission of the IEEE; see supporting documentation.