University of Massachusetts Dartmouth

Department of Electrical and Computer Engineering

Processor-in-Memory Computer Architectures

A Thesis in

Computer Engineering

by

Richard Muri

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

January 2019

We approve the thesis of Richard Muri

Date of Signature

Paul Fortier
Professor, Department of Electrical and Computer Engineering
Thesis Advisor

David Rancour
Associate Professor, Department of Electrical and Computer Engineering
Thesis Committee

Gaurav Khanna
Professor, Department of Physics
Thesis Committee

Liudong Xing
Graduate Program Director, Department of Electrical and Computer Engineering

---

Antonio H. Costa
Chairperson, Department of Electrical and Computer Engineering

---

Jean VanderGheynst
Dean, College of Engineering

Tesfay Meressi
Associate Provost for Graduate Studies

Abstract

Processor-in-Memory Computer Architectures

by Richard Muri

A processor-in-memory (PIM) computer architecture is any design that performs some subset of logical operations in the same location as memory. The traditional model of computing involves a processor loading data from memory to perform operations, with a bus connecting the processor and memory. While this technique works well in many situations, a growing gap between memory performance and processor performance has led some researchers to develop alternative architectures.

This thesis includes a discussion of what is a PIM architecture, as well as motivations, applications, and limitations of PIM. After providing background information on the subject, a Field Programmable Gate Array (FPGA) implementation of a PIM enhanced microcontroller is presented. Using an Artix-7 FPGA, an ATmega103 microcontroller soft core is modified to include a PIM core as an accelerator. The sample application of AES encryption provides a comparison between the baseline processor and the PIM enhanced machine.

Acknowledgements

Completing a graduate degree requires not only dedication from an individual, but support from a wide group of people. I would first like to thank my adviser, Professor Paul Fortier, for his expertise and patience. Thank you also to Professor David Rancour and Professor Gaurav Khanna for serving on my thesis committee. Dr. Benjamin Viall and Patrick DaSilva both contributed significant technical advice which accelerated my research. The graduate students of the Electrical and Computer Engineering department provided both camaraderie and commiseration, and I am grateful for the lasting friendships formed while working on this thesis. The faculty and staff of Electrical and Computer Engineering department have helped guide me through college and turn me into an engineer. In particular I would like to thank Ms. Fernanda Botelho, without whom the whole department could not run. The final people I must thank are my parents, James and Nancy Muri. My parents have nurtured and encouraged me from day one. They have fed me, housed me, loved me, and listened to me complain about how much I hate writing for my entire life. It is impossible to know how much my parents have done for me, but I hope they know I appreciate it.

Contents

List of Figures viii

List of Tables x

Acronyms xi

Chapter 1 Introduction 1

1.1 Problem Statement 1

1.2 Approach 2

Chapter 2 Background Information 6

2.1 Applications 7

2.2 Motivations 9

2.2.1 Performance Improvements 9

2.2.2 Energy Efficiency 10

2.2.3 Limitations of Traditional Architectures 11

2.3 Limitations 12

2.4 Why Now? 14

Chapter 3 Survey of PIM Literature 16

3.1 DRAM 16

3.1.1 Computational RAM 17

3.1.2 IRAM 20

3.1.3 Bulk Bitwise Operations in DRAM 23

3.2 SRAM 26

3.2.1 DRC^2 . . . 28
3.2.2 Terasys . . . 31
3.2.3 Intelligent SRAM . . . 32
3.2.4 Recryptor . . . 35
3.3 Resistive Memories . . . 39
3.3.1 PRIME . . . 41
3.3.2 Spin-Transfer Torque Magnetic RAM . . . 43
3.3.3 Computation-in-Memory Parallel Adder . . . 47
3.4 Three Dimensional Integration . . . 49
3.4.1 Tesseract . . . 50
3.4.2 HAMLeT . . . 52

Chapter 4 FPGA-based PIM Simulation 55
4.1 Hardware Architecture . . . 55
4.1.1 ATmega103(L) Microcontroller . . . 56
4.1.2 PIM Modifications . . . 57
4.2 Processor in Memory Instructions . . . 60
4.2.1 Instruction Format . . . 60
4.2.2 Instruction Pipeline . . . 62
4.2.3 Typical PIM Operation Usage . . . 63
4.3 Example Application . . . 64
4.4 Results . . . 67
4.4.1 Software Comparison . . . 67
4.4.2 Hardware Comparison . . . 73

Chapter 5 Conclusions and Future Work 75

5.1 Implementation Limitations 75
5.2 Future Work 77

Appendix A C Source Code 78
A.1 AES.c 78
A.2 PIM_AES.c 84
A.3 AES.h 108
A.4 softcore_util.c 109
A.5 softcore_util.h 114

References 116

List of Figures

Figure 1.1: DRAM performance improvement versus processor performance 3

Figure 1.2: The von Neumann architecture 4

Figure 3.1: DRAM Array 17

Figure 3.2: CRAM bit processor 19

Figure 3.3: VIRAM1 floorplan 21

Figure 3.4: VIRAM1 vector lane 22

Figure 3.5: Standard six transistor SRAM cell 27

Figure 3.6: Example SRAM array 28

Figure 3.7: Ten transistor static random access memory (SRAM) cell with separate read and write ports 29

Figure 3.8: Additional column hardware enabling logic operations 30

Figure 3.9: ISRAM hardware architecture 33

Figure 3.10: ISRAM performance summary 34

Figure 3.11: Ten transistor SRAM cell used in Recryptor 36

Figure 3.12: Relative performance comparison of Recryptor 38

Figure 3.13: ReRAM cell 40

Figure 3.14: ReRAM crossbar 40

Figure 3.15: STT-MRAM cell 44

Figure 3.16: Tesseract architecture 51

Figure 3.17: HAMLeT architecture 53

Figure 4.1: ATmega103 architecture and PIM modifications 58

Figure 4.2: ATmega103 data memory with PIM modifications 59

Figure 4.3: Format of a PIM instruction 61

Figure 4.4: Example pipeline of a PIM instruction 63
Figure 4.5: Example usage 64
Figure 4.6: AES round steps illustrated 66
Figure 4.7: Function to xor two 16 byte arrays 69
Figure 4.8: Generated assembly for function in Figure 4.7 70
Figure 4.9: PIM implementation of function in Figure 4.7 71
Figure 4.10: Logical values in the mix_sub_columns function are dependent on run time addresses, preventing the use of PIM operations 72
Figure 4.11: FPGA resource utilization 74

List of Tables

Table 3.1: VIRAM1 performance relative to other embedded microcontrollers (higher is better) 23
Table 3.2: Truth table for in-DRAM bitwise operations 24
Table 3.3: Throughput comparison of bitwise AND operations 26
Table 4.1: Artix-7 FPGA Specifications 56
Table 4.2: Opcode List 62
Table 4.3: Performance comparison of AES encryption on base ATmega103 and PIM enhanced ATmega103 68

List of Acronyms

AES Advanced Encryption Standard

ADC analog-to-digial converter

ASIC application specific integrated circuit

ALU arithmetic logic unit

BL bit line

CIM Computing-in-Memory

CMOS Complementary Metal Oxide Semiconductor

CPU central processing unit

CRAM Computational RAM

CSB crypto-SRAM bank

dbC data-parallel bit C

DMA direct memory access

DAC digital-to-analog converter

DRAM dynamic random access memory

DRC^2 Dynamically Reconfigurable Computing Circuit

ECC Elliptic curve cryptography

EEMBC Embedded Microprocessor Benchmark Consortium

FPGA field programmable gate array

GPR general purpose register

GPU graphical processing unit

GPIO general purpose input/output

HAMLeT Hardware Accelerated Memory Layout Transform

HRS high resistance state

HMC Hybrid Memory Cube

IRAM Intelligent RAM

ISRAM Intelligent SRAM

ISA instruction set architecture

LRS low resistance state

MTJ magnetic tunnel junction

NN neural network

PIM processor-in-memory

RAM random access memory

RBL read bit line

RBLB read bit line bar

RBLF Read Bit Line False

RBLT Read Bit Line True

RISC Reduced Instruction Set Computer

RWLF Read Word Line False

RWLT Read Word Line True

ReRAM Redox Random Access Memory

RTL register transfer level

SIMD single instruction, multiple data

SL source line

STT-CIM spin-transfer torque compute-in-memory

STT-MRAM spin-transfer torque magnetic RAM

SRAM static random access memory

S-box substitution box

TSV through silicon via

VIRAM1 vector IRAM

VLSI Very Large Scale Integration

VHDL Very High Speed Integrated Circuit Hardware Description Language

WL word line

WBLF Write Bit Line False

WBLT Write Bit Line True

WWL Write Word Line

Chapter 1 Introduction

1.1 Problem Statement

The limiting factor for computer performance is often the ability to transfer data between memory and the processor. From 1980 to 2000, microprocessor performance improved by approximately 60% per year, while dynamic random access memory (DRAM) access time improved by less than 10% per year [1, 2] (Figure 1.1). Due to the processor and memory performance gap, developing strategies to ensure a processor has work to perform during memory latency has been a major area of research in computer engineering [3].

The divergence between processor and memory performance creates further issues as hardware scales in performance. Memory bandwidth problems become more apparent as processor throughput increases. As more instructions are completed, more data is needed to match the processor execution rate. If the memory bandwidth is unable to keep up, latency increases as the response to processor request slows [1]. Equation 1.1 describes the average memory access time of a system,

$$t_{avg} = p * t_c + (1 - p) * t_m \tag{1.1}$$

where p is the probability of a cache hit, t_c is the time to access the cache, and t_m is the time to access main memory.

Assuming that processor requests are matched by the cache memory, it is apparent that as t_c and t_m diverge, main memory access time will have a greater effect on both t_{avg} and system performance. Depending on the frequency of memory references and the cache hit rate, t_{avg} becomes the bottleneck. Even with conservative assumptions like a cache hit rate

of 99% and an average of one memory reference per five instructions, t_{avg} quickly becomes the limiting factor [4].

DDR4 random access memory (RAM), made commercially available in 2014, has an access latency of approximately 45 nanoseconds [5]. The Intel i7-6700 processor, commercially available in 2015, has a reported 4640 million instructions per second performance on the single-threaded LZMA compression benchmark [6]. Assuming one in five instructions are memory accesses, the cache hit rate is 99%, and cache hits add negligible additional time to instruction throughput, running LZMA compression for one second results in 0.4176 seconds of time added by memory access latency, and 0.5824 seconds of cache hits or non memory access instructions. More than 40% of the run time in this scenario is devoted to memory access. If the memory latency trend depicted in Figure 1.1 continues, performance improvements to processors will not significantly improve system performance.

1.2 Approach

The memory bus as a limiting factor in computer efficiency is commonly referred to as the von Neumann bottleneck, after the ubiquitous computer architecture of the same name (Figure 1.2). In a 1977 lecture, John Backus of IBM states "Not only is this [memory bus] a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck..." [7], implying the von Neumann architecture has inherent shortcomings.

The von Neumann bottleneck exists as a result of the difference between the rate processors can handle data versus the rate data can be transferred from memory. Processors are much faster, leading to wait cycles where the processor must delay execution while data is transferred. Modified architectures, such as the Harvard architecture, and memory hierarchy, such as associative caches, improve the overall throughput of a processor, but the bottleneck of memory transfer is fundamentally unchanged [8].

Many techniques have been developed to improve the efficiency of modern computers.

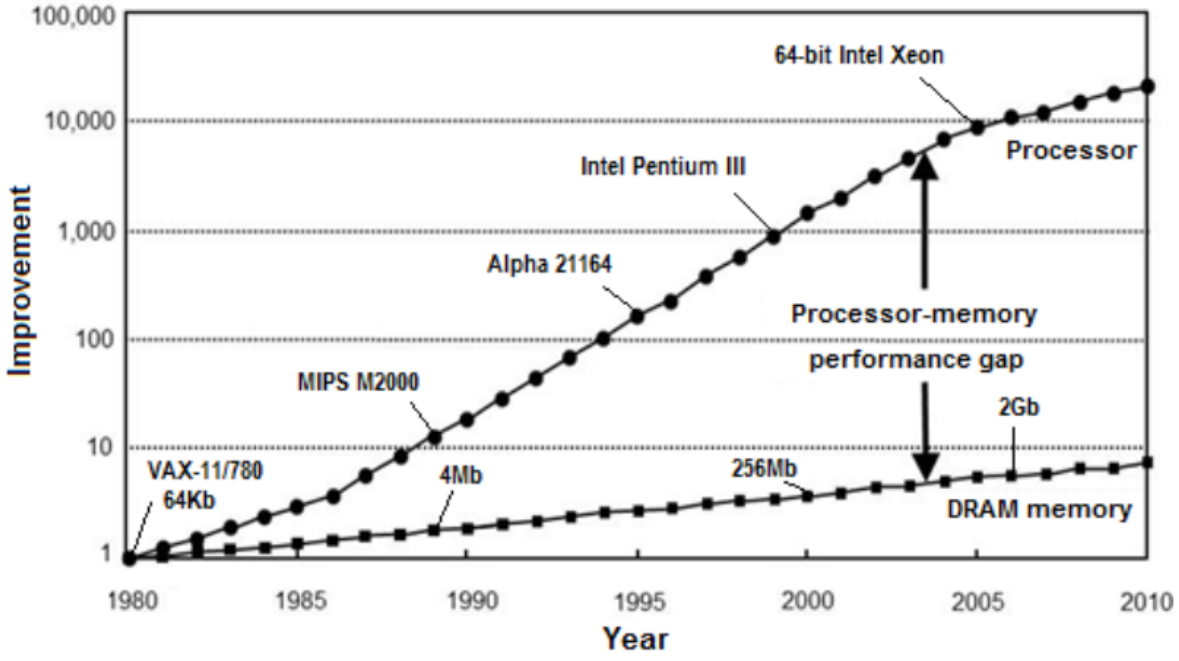


Figure 1.1: DRAM performance improvement versus processor performance [2]

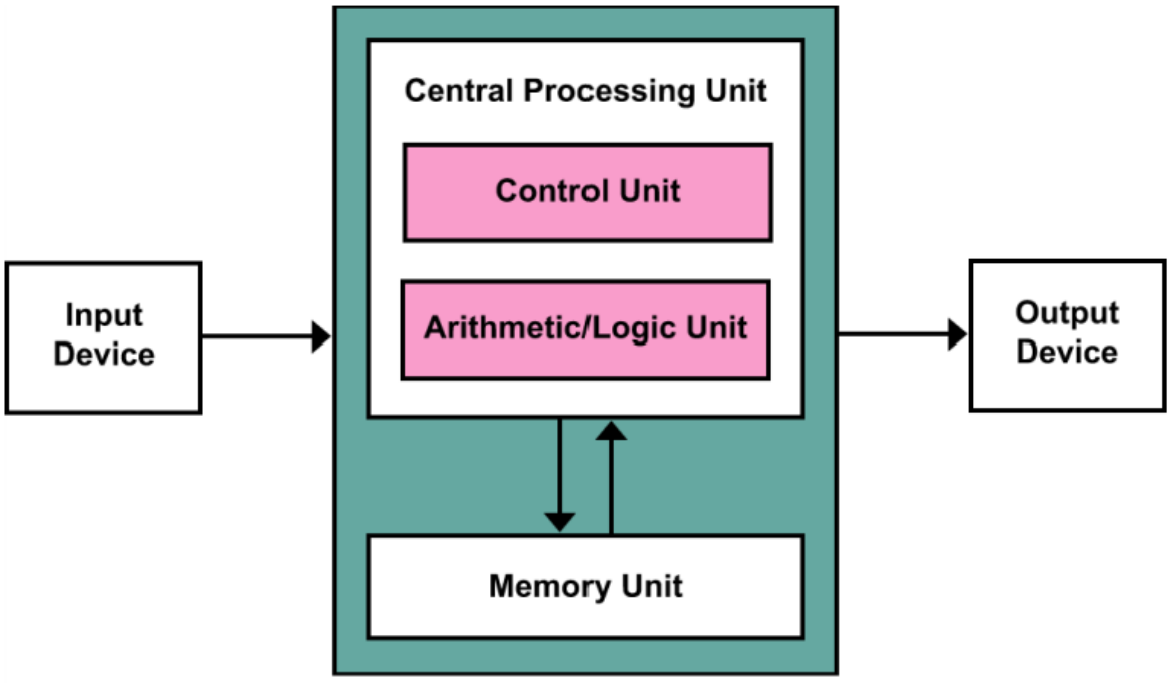


Figure 1.2: The von Neumann architecture

Architectural features such as pipelining, memory caching, and branch prediction improve the throughput of a processor by working around the bottleneck [8,9]. Alternative programming paradigms such as functional programming or self-modifying code have been proposed to avoid passing as many instructions over the data bus [7]. Parallel computing and multiple core systems increase the number of processors executing code in order to accomplish more instructions in the same amount of time. Recently reconfigurable hardware has emerged as a means to produce application specific hardware not requiring a traditional data bus [9]. All of these approaches improve computing efficiency, potentially with trade-offs, and may be employed in some combination given the design requirements.

This thesis will evaluate the use of a processor-in-memory (PIM) architecture to improve computer efficiency. PIM is another technique used to mitigate the von Neumann bottleneck by reducing instruction data that must travel over the memory bus. PIM represents a change to the fundamental computer architecture used for years. Where the data is stored, hardware exists to perform logical and arithmetic operations on the stored data. Computations in memory potentially reduces the number of required instructions in terms of load and store operations. Additionally, if a general purpose processor is used in conjunction with PIM, the main processor is free to perform other operations while the PIM unit provides in memory computations.

The rest of this thesis is organized as follows: Chapter 2 defines PIM and discusses motivations, applications, and limitations of PIM, Chapter 3 includes a survey of literature on PIM and classifies it into four main categories based on the memory type used, Chapter 4 details this thesis' contributions simulating PIM on an field programmable gate array (FPGA), and Chapter 5 includes conclusions and future work.

Chapter 2 Background Information

The logical solution to the von Neumann bottleneck is to significantly improve memory transfer bandwidth. Bandwidth inside a memory chip is several orders of magnitude higher than on a bus between memory and a processor [10]. Memory-focused architectures that perform logical operations in the same location where memory is stored exploit on-chip memory bandwidth for performance improvements.

PIM architectures seek to mitigate the von Neumann bottleneck by placing logical operations and memory in the same physical location [11]. Many different names have been applied to the basic strategy of combining logic and memory, including smart/intelligent memory, Intelligent RAM (IRAM) [12], merged DRAM/Logic, and Computing-in-Memory (CIM) [11]. This thesis will use PIM as the preferred terminology for any architecture involving interleaved logic and memory circuitry, or any strategy involving using memory to perform logical operations. Some PIM strategies discussed include using different types of modified RAM, interleaving logic circuitry with memory circuitry, and exploiting properties of traditional RAM to perform logical operations. Relatively new strategies have been proposed using memristor based RAM [13], 3D integration of logic and memory [14, 15], reconfigurable hardware [16, 17], and RAM powered by spin transfer torque [18]. Chapter 3 contains a detailed description of various PIM implementations and proposals.

There are two broad approaches to creating a closer coupling of memory and logic: in-memory computing and near memory computing [18]. Both of these approaches fall under the classification of PIM, although this thesis focuses mainly on in-memory computing and near-memory techniques performing fine-grained operations on the memory. For instance, Recryptor [19], a PIM implementation of a cryptography accelerator, includes both classifications. The RAM is modified to allow bitwise operations (in-memory computing), and some

peripherals are added nearby, such as a shifter, rotator, and S-box (near memory computing). 3D integration using stacked memory and logic chips fits the description of near memory computing, but is still discussed in this thesis because of the similarities in outcomes with in-memory computing, namely the ability to perform logical operations on arbitrary words in memory while minimizing bus traffic.

While there are many strategies that could be classified as PIM, most implementations share an approach that revolves around selecting multiple words in memory simultaneously. In a typical processor, words are selected and loaded from data memory one at a time to a register file. The values are operated on from the register file, and eventually stored back to data memory. In PIM, multiple words are selected and operated on directly in memory, with the results being stored in place. Depending on the PIM implementation, logic may rely on electrical properties of the memory itself, additional logic gates attached to the sense amplifiers, or functional units contained by the same die as the memory. Specific PIM implementations are discussed in more detail in Chapter 3.

2.1 Applications

The primary reason to introduce a new style of computer architecture is to facilitate problem solving. Without an application in mind, an architecture is useless. In their 2018 lecture, Turing award winners John Hennessy and David Patterson predict "new advances could create compilers and domain-specific architectures that deliver tenfold or more jumps [in performance]" [20]. Hennessy and Patterson note that while single core performance improvements have slowed down in recent years, as opposed to the historical trend where processors improved by 60% a year, significant progress may still be made in domain specific areas. PIM architectures are one such style of architecture, well-suited to a specific domain, instead of serving as an overall improvement to general computing.

PIM architectures are best applied to "memory wall" problems [4], or applications that

perform poorly on modern architectures due to the von Neumann bottleneck. In particular applications that are scalable to parallel processors [21], or that require repeated memory accesses with relatively simple logic operation, are well suited to a PIM approach. The reason for this is because of two common limiting factors improved by a PIM architecture, limited general purpose registers (GPRs) [2] and imperfect caches. Programs that load more data than could fit in the register file can be improved by performing operations directly in memory rather than loading and storing data as needed using the limited number of available GPRs. Programs that cannot utilize a cached memory hierarchy to hide access latency, such as sparse matrix operations with irregular data access patterns [21], also greatly benefit from performing operations in memory.

A common theme among many example applications for PIM architectures is the size of the data set involved. Hamdiou et al. [22] discuss a DNA analysis problem, involving comparing 200 GB of DNA against a 3 GB reference. The analysis is computational simplistic and well suited to parallelization, but across a huge data set. The authors estimate orders of magnitude better performance using a PIM architecture than a traditional von Neumman machine.

PIM can also be applied to smaller scale problems that are punished by a small number of GPRs, as opposed to frequent cache misses. One such application is accelerating cryptographical functions, such as block ciphers or hashing algorithms [19]. On a microprocessor with a small register file, a cipher algorithm might involve frequently loading segments of the cipher block into the register file, performing the necessary logical operations, and storing the result back to memory to free the register file for the next segment. PIM allows instead the operations to be performed directly in memory, without worrying about the number of GPRs available.

Sample applications for PIM architectures from the literature include graph processing

[14, 23], neural networks [24], cryptography [19], sparse matrix operations [15]. Li et al. [23] note PIM greatly increases performance of bitwise vector operations, commonly used in databases, graph processing, bio-informatics, and image processing.

2.2 Motivations

The ultimate purpose of using a PIM architecture is to mitigate the von Neumann bottleneck, however, there are other motivating factors. The benefits of PIM can be divided into three major categories of performance improvement, energy efficiency, and overcoming limitations of traditional computer architectures.

2.2.1 Performance Improvements

Internal memory bandwidth is typically much larger than external memory bandwidth [10, 23]. PIM makes use of internal memory bandwidth, as operations are also internal to the memory. Vector operations wide as the memory row offer massive parallelism. In situations where multiple rows are operated on, using the full bit width of the memory, data is processed with multiple orders of magnitude more bandwidth than the equivalent DDR3 RAM bus [23]. In problems that benefit from parallelization, such as graph processing, PIM enables memory capacity proportional performance [14], whereas traditional architectures fail to scale with capacity.

Not all applications benefit from massive parallelism. One of the major challenges of realizing a single instruction, multiple data (SIMD) machine is that for less well structured applications, performance degrades. PIM architectures have the benefit of also being memory. For applications solvable in parallel, PIM operations are available. In situations where parallel operations are unnecessary, or not useful, PIM hardware still functions as traditional memory. Instead of using die area for hardware that is not always useful, PIM provides a parallel processing accelerator that functions as memory while not in use [25].

2.2.2 Energy Efficiency

The traditional memory hierarchy is a major power consumer in a microprocessor. By changing the frequency of data transfers, or the memory model, PIM can lead to great improvements in energy efficiency. Computation is the primary purpose of a system, but consumes much less chip area and power than communication and memory access; cache accesses and communication can take easily between 70% and 90% of energy consumption in a microprocessor [22]. A memory-centric processor architecture is a natural next step towards producing energy efficient systems. The current processor-centric approach relies on a memory model that not only creates a performance bottleneck, but also significant power consumption overhead.

The trend of memory access requiring more power than computation is apparent in many systems. In some modern graphical processing units (GPUs), fetching operands is more costly than operations [26]. Transferring data between the microprocessor and off-chip memory is even worse, requiring up to two orders of magnitude more energy than a floating point operation [23, 24]. For applications requiring high memory bandwidth, such as machine learning applications, memory access can grow to consume almost all of the energy in a system. DRAM access is responsible for 95% of the power consumption in the DianNao neural net accelerator [24].

PIM architectures solve these memory related energy efficiency problems by greatly reducing or eliminating the need to transfer data between chips. When operations are performed directly in memory, there is no need to use high capacitance buses to transfer data to and from the processor [26, 27]. Additionally in PIM systems it may be possible to streamline the memory interface, depending on the design. If in memory operations are largely sufficient for the applications of a PIM system, wide parallel memory interfaces using

the majority of the pins on a microprocessor can be swapped for serial interfaces [27], as data transfer is less necessary when the majority of operations are performed in place.

2.2.3 Limitations of Traditional Architectures

The gap in processor and memory performance creates a need for cached memory hierarchies to overcome the memory access latency. A memory hierarchy places smaller, high-speed caches physically close to the processor, often tiered into multiple levels. The first level is the smallest, fastest, and closest to the processor, with subsequent cache levels having a greater capacity at the expense of speed and distance from the processor. Cache accesses are much faster than memory accesses, but require data to be copied from main memory. Caching is far from a perfect solution, as caches represent significant die area and complexity added to a processor. Sometimes more than 40% of the area in a processor die is taken up by the cache [2, 12].

Caches consist of redundant data made necessary by the performance gap between memory and processors. If memory could keep up with the processor, there would be no need to include a cache hierarchy [2,27]. Maintaining a cache hierarchy contributes to system complexity and energy consumption. Data modified in the cache must be modified in main memory to match, and old data in the cache must be updated from the memory.

Main memory design further complicates issues related to the processor versus memory performance gap. The focus of DRAM improvements have been on capacity and bandwidth, at the cost of latency [5,28]. This increases the penalty of a cache miss, and severely degrades the performance of applications where cacheing is less effective. Techniques to mitigate the cost of such latency, including out of order execution and speculation, introduce non-trivial complexity and area overhead, and do not scale well. Two MIPS processors, the R5000 and R10000, are good examples. The R5000 is a simple RISC machine, while the R10000

includes out of order execution and speculation. The R10000 performed 1.64 times better on the SPECint95 test bench, but requires 3.43 times more chip area [27].

Systems utilizing multiple processor cores on a die exacerbate the bottleneck even more. Sharing a die means each processor is limited in cache size; a smaller cache requires more memory accesses, and places a greater demand on the main memory [27]. One method of improving modern data intensive applications is to perform operations in parallel using multiple processor cores. Adding additional cores allows more parallel operations, but also increases memory access requirements as the cache size must go down to fit more cores on the same die. In order to scale, PIM architectures do not need to make the same trade offs between cache size and functional units as a traditional system.

In the modern age, a series of applications sometimes referred to as big data problems are prevalent. Big data problems are especially data intensive, and include applications such as large scale scientific computing, business analytics, machine learning, or bio-informatics. The data size of these problems has grown at a greater rate than hardware improvements have scaled processor performance. For applications where the primary goal is to process vast amounts of data for hidden trends, the volume of storage and processing acts as a barrier for traditional architectures. Memory size and access negatively impact both performance and energy consumption, requiring new architectures designed with big data in mind [22]. Supercomputers designed to handle data intensive problems are expensive, power hungry, and area inefficient [29]. PIM is an alternative providing parallelism inside the memory, mitigating problems caused by a bus bottleneck and complex cache hierarchies.

2.3 Limitations

The primary challenge developing a PIM architecture is to realize a useful logic system without significantly reducing the function of standard memory. One of the limiting factors in building such architectures is the required number of transistors for logic circuits, making it

impractical to incorporate into a memory device, although this has become less of a problem as transistor density has improved [1]. Incorporating logic into memory requires a trade-off potentially with either memory density, or efficiency [18]. As a result, PIM is typically limited to simple operations, such as bitwise logic. Some implementations do include simple arithmetic operations such as addition [18,19,26,29], but not all operations can be performed in memory efficiently. Most PIM implementations must function as an accelerator or co-processor to a more general computer, or as an application specific architecture.

A major limiting factor in developing PIM is the manufacturing process. The semiconductor industry is split: microprocessors and memory are designed by different parties [28,30]. Early PIM efforts failed because design/manufacture of performance optimized logic with density optimized memory is not cost effective. Logic designs take extra metal layers, and are not desirable for memory manufacturers [23]. PIM designers are either limited to existing memory technologies, or are faced with costly custom solutions. The process of memory manufacture is highly optimized to produce economical, dense memory chips, and requires compromise either in price or in memory performance to produce a mixed logic and memory design.

Introducing a new source of complexity into memory design, logic, complicates the design requirements and test requirements. Thermal efficiency is especially important in PIM architectures. Data retention rates are affected negatively as die temperature increases [12, 27], and thermal requirements limit the complexity of logic in PIM [31]. High speed switching logic also adds additional noise into the memory system [27]. Testing the whole system becomes more difficult. Adding processor elements increases the time and cost of testing to memory [12].

Finally, as with any new architecture, in order for widespread adoption, PIM must be supported with software. PIM operations need to be used by compilers that are capable

of recognizing program parallelism, and are capable of generating instructions that make effective use of the internal memory bandwidth [2]. For people to use PIM, it must run a significant body of software, and it must be sufficiently easy to program. Widespread adoption will occur when some critical value of cost-performance benefit, ease of use, and availability of useful programs becomes available [27].

2.4 Why Now?

PIM is not a new concept, with significant research being performed on the subject in the 1990s [23]. PIM has seen a recent resurgence in the literature for three main reasons: modern improvements to hardware technologies, greater demand for data intensive applications, and economic incentives.

New memory technologies and improvements to existing technologies improve the viability of PIM. One of the greatest barriers of entry for PIM was the challenge of fitting memory and logic into the same die. In modern times, smaller feature sizes and greater transistor density make it easier to fit useful logic and sufficiently dense memory into the same die [1]. Additionally new memory materials and manufacturing techniques show promise for use in PIM applications. Resistive memories based around memristors arranged in crossbars enable in memory operations. Three dimensional integration using through silicon vias (TSVs) allows logic and memory dies to be stacked on top of each other, effectively combining into a singular chip.

The emergence of big data applications incentivized research for accelerators. PIM offers the potential for memory capacity proportional performance, making it an attractive option for big data applications.

The final reason PIM is being worked on again now is Moore's Law has finally slowed down. Massive improvements to system performance that drove economic growth over the past 20 year was largely driven by hardware improvements, such as denser transistors and

higher clock rates [20]. This trend has disappeared, leaving space in the field of computer architecture to search for alternative improvements. PIM was never widely adopted in part because of memory density problems which are now less of an issue, and because of the cost involved with developing custom memory. Now that there are other incentives to develop PIM, the costs associated are more acceptable.

Chapter 3 Survey of PIM Literature

This chapter contains a survey of literature summarizing published PIM proposals, strategies, and implementations. The chapter is divided into four sections based on the memory technology described: Section 3.1 DRAM, Section 3.2 SRAM, Section 3.3 resistive memories, and Section 3.4 three dimensional integration.

3.1 DRAM

DRAM is ubiquitous in modern computer systems, serving as the foundational technology of main memory for decades [5]. DRAM is desirable for its density and cost. The manufacturing process has been optimized to produce high-capacity, inexpensive memory. The trade-off is that DRAM has higher latency than other memory technology, such as SRAM [30]. Latency is one of the primary motivations for PIM, in an effort to avoid the bottleneck it creates. A basic understanding of the architecture of DRAM is necessary to understand how PIM is applied to improve it.

DRAM consists of a two dimensional array of cells, each of which store a singular bit (Figure 3.1). A standard DRAM cell contains a capacitor connected to a transistor, controlled via a word line and bit line. Word lines are connected to a row decoder that corresponds to memory address, and bit lines are connected to sense amplifiers capable of latching digital values. A cell is accessed by setting the word line high, connecting the capacitor to the bit line. The bit line is pre-charged; when the word line goes high, the capacitor shares charge with the bit line. A sense amplifier at the end of the bit line determines if the capacitor was charged, representing a logic '1', or discharged, representing a logic '0'. Such a read is destructive, requiring a rewrite to preserve the bit value. Additionally, the capacitor will leak charge over time, even when the access transistor is turned off.

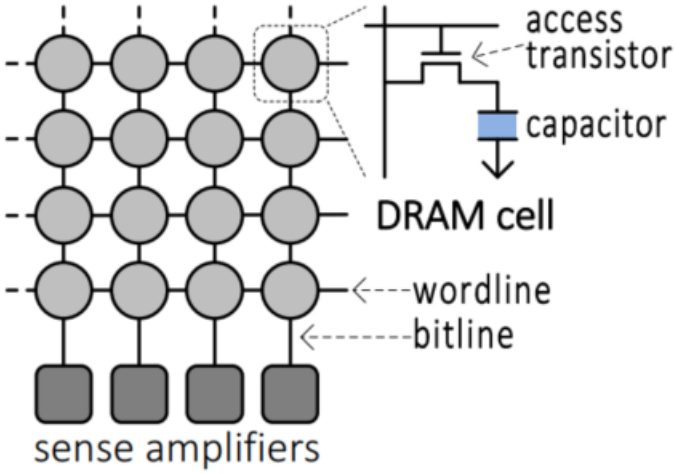


Figure 3.1: DRAM Array [5]

Peripheral circuitry that periodically restores every charged capacitor is necessary to avoid data loss [30].

DRAM is typically arranged into several hierarchical levels: a chip contains multiple banks (often eight), each of containing rows of cells [5]. In order to avoid problems with layout, long wires, and degraded performance due to large bit lines and word lines, banks contain multiple subarrays. A typical size for a subarray is 256 words by 512 bits [30].

3.1.1 Computational RAM

Early PIM implementations sought to incorporate SIMD into memory. In 1992, Elliot et al. [10] prototyped a PIM implementation using DRAM called Computational RAM (CRAM). CRAM places processing elements directly into main memory in line with memory columns. It exploits internal memory bandwidth available at the sense amplifiers to improve performance. The authors note that CRAM could replace conventional memory to add array processing capabilities to standard hardware, or be used as a standalone processor for embedded video or signal processor.

CRAM uses 1-bit processors located at the bottom of memory columns (Figure 3.2). There are three inputs, two registers and memory, to an arbitrary 8-bit truth table microcoded by an off chip controller. Inter-processor communication is done with shift registers between adjacent processors.

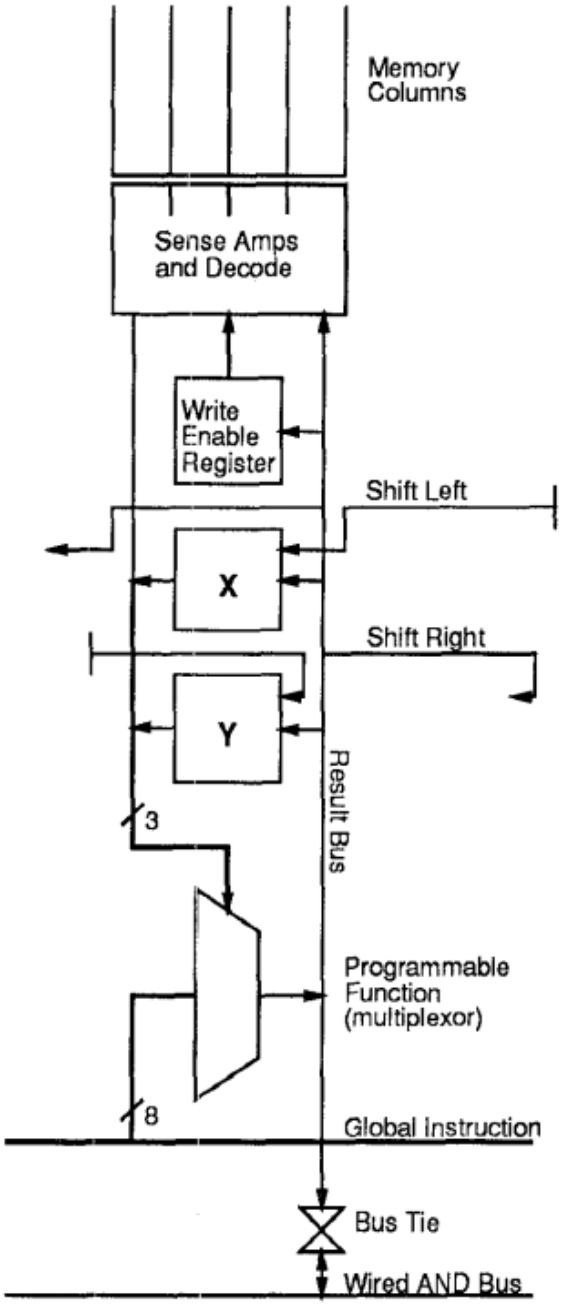


Figure 3.2: CRAM bit processor [10]

A prototype CRAM was fabricated using 8 Kbits of SRAM as the memory. In the

prototype, processor elements make up 9% of the chip area. An extension of the design is to use DRAM instead of SRAM for the implementation. CRAM is able to read, perform two ALU operations, and write-back in 150 ns, meaning the speed of DRAM operation is not impacted. A DRAM design has lower performance than SRAM, but is more desirable due to better memory density. 32 Mbytes of CRAM is capable of performing 13 billion 32 bit additions per second. This result is unrealistic, as it is the theoretical maximum and also reliant on data locale within the memory, but even utilizing a fraction of the theoretical performance is impressive.

3.1.2 IRAM

University of California Berkeley's IRAM project focused on identifying the design issues involved with a merged processor and DRAM system [12, 27, 28, 32, 33]. The project resulted in the vector IRAM (VIRAM1) processor, which the research group designed and fabricated [34]. VIRAM1 combines PIM with vector instructions to exploit internal memory bandwidth for massive parallelism. Vector instructions operate on linear arrays of numbers, whereas a typical scalar processor operates on one piece of data at a time. VIRAM1 targeted embedded media applications because media processing benefits from vectorization, and the PIM nature of the architecture enabled a low power design.

A vector architecture was chosen for its parallelism and relative simplicity. VIRAM1 was designed by a team of researchers and graduate students, while most commercial microprocessors are designed by a large contingent of engineers over years time. Vectors inherently guarantee mutual independence of data elements. There is no need to use complex structures like speculation, prediction, or re-ordering to expose parallelism as vectors are explicitly parallel. Avoiding these more complex architectural features saves power, and limits design complexity. Operations are executed on parallel data paths, referred to as lanes. Performance

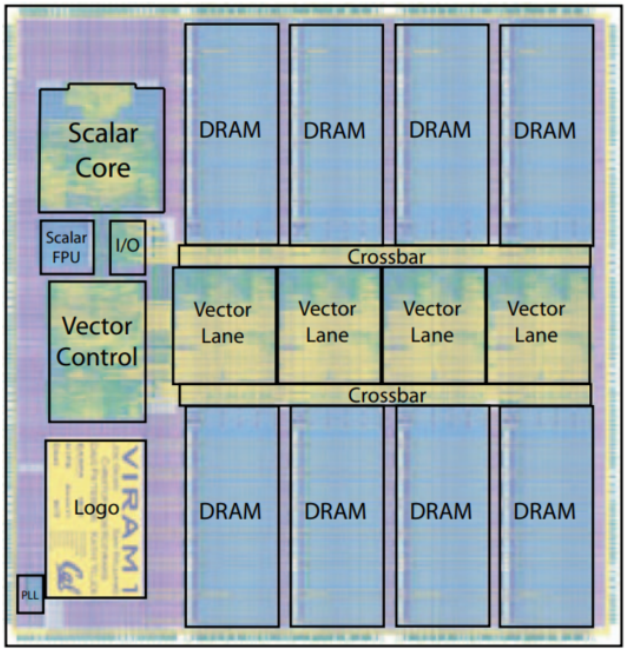


Figure 3.3: VIRAM1 floorplan [34]

in terms of both energy efficiency and processing throughput is improved by operating on full vectors. Each vector is fetched and decoded as a unit, and operated on wholly.

Figure 3.3 shows an overview of VIRAM1. The design consists of four key components: a scalar core, a vector control unit, four vector lanes, and eight embedded DRAM modules. The scalar core is a MIPS M5Kc 64-bit microprocessor with 8 KBytes of instruction and data caches. Interfaced with the core are a vector control unit and a floating point unit. The DRAM modules are each 13 MBits with an eight byte interface connecting to vector lanes.

Figure 3.4 shows a single vector lane. Each lane contains one quarter of an eight Kbyte register file, a floating point arithmetic unit, a fixed point arithmetic unit, and a flag unit. Single-precision floating point and integer multiplication are supported, as well as fixed-point multiply-add instructions targeted at media applications. Fixed-point operations can be partitioned into one 64-bit operation, two 32-bit operations, or four 16-bit operations. The

flag unit contains flag registers holding masks for vector instructions, enabling conditional execution on individual vector elements.

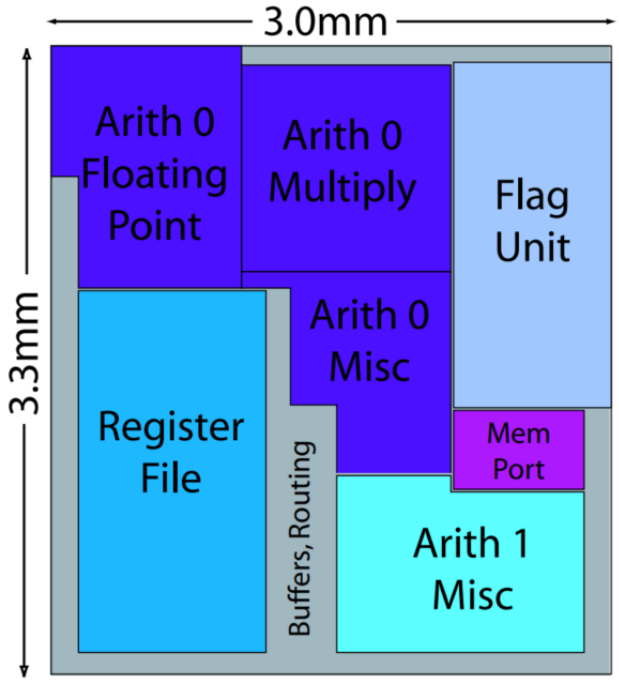


Figure 3.4: VIRAM1 vector lane [34]

VIRAM1 was evaluated using the Embedded Microprocessor Benchmark Consortium (EEMBC) suite for comparison. Table 3.1 contains the results for the VIRAM1 and several contemporary embedded processors in the Telecom and Consumer application areas. VIRAM1 significantly outperforms the listed processors. Not only is the performance higher, but the VIRAM1 uses minimal power, and runs at a relatively low clock frequency.

IRAM is shown to be a promising area of computer architecture research. The VIRAM1 outperformed contemporary embedded processors, and it was designed and implemented by a research lab. A commercial manufacturer with substantially more resources could produce

Table 3.1: VIRAM1 performance relative to other embedded microcontrollers (higher is better)

Processor	Clock Frequency	Power	ConsumerMark	TeleMark
VIRAM1	200 MHz	2 W	201.4	61.7
Motorola MPC7455	1000 MHz	21.3 W	122.6	27.2
AMD K6-III+	550 MHz	21.6 W	34.2	8.7
TI C6203	300 MHz	1.7 W	n/a	44.6
NEC MIPS VR500	250 MHz	12.1 W	14.5	2.0
Trimedia TM1300	166 MHz	2.7 W	110.0	n/a

an even better design. VIRAM1 proves PIM is a viable commercial strategy, provided economic and manufacturing limitations are overcome [12].

3.1.3 Bulk Bitwise Operations in DRAM

A common strategy in PIM is to modify sense amplifiers or add logic near the amplifiers to enable bitwise operations between words. Seshadri et al. [35] describe a method to use DRAM to perform bitwise AND and OR operations, with only minor modifications. In a traditional system, a bitwise operation requires loading multiple words from main memory, processing them, and writing the result back into memory. This procedure is undesirable and inefficient, particularly for larger applications with lots of data, such as databases.

The proposal allows bitwise operations to be performed in bulk by exploiting the fact that a sense amplifier is connected to many DRAM cells. Three rows of DRAM are activated simultaneously, causing a multiple kilobyte wide bitwise AND or OR operation on two of the rows; the third row is used to select between bitwise operations.

The principal behind in-DRAM bitwise operations is based on charge sharing during memory access. The logical value of a cell is determined by the voltage of a bit line after charge sharing. If the voltage at the sense amplifier is greater than $\frac{1}{2}V_{DD}$ (where V_{DD} is the operating voltage of the DRAM), a logic '1' is recorded, and a logic '0' is recorded if the

Table 3.2: Truth table for in-DRAM bitwise operations

R	A	B	$Result$
0	0	0	**0**
	0	1	**0**
	1	0	**0**
	1	1	**1**
1	0	0	**0**
	0	1	**1**
	1	0	**1**
	1	1	**1**

voltage is less than $\frac{1}{2}V_{DD}$. The voltage level of the bit line when three cells are active is greater than $\frac{2}{3}V_{DD}$ if at least two cells are charged, resulting in a logic '1'. If no more than one cell is active, the bit line voltage is no greater than $\frac{1}{3}V_{DD}$, resulting in a logic '0'. A truth table for the method is shown in Table 3.2, where R refers to the control row, and A and B are the operand rows. If R is '0', then the result is A AND B; else if R is '1', the result is A OR B.

A glaring issue with the design is that the bitwise operations are destructive. To overcome this problem, first two source rows are copied into two temporary rows, then a third temporary control row is initialized to '0' or '1' depending on the operation. Finally the results are written back to a sixth destination row. Copying this much data would ordinarily add more overhead than the bulk operation is worth, but the authors propose another method called RowClone [36] for efficiently copying rows within the same subarray or bank. The details of RowClone are beyond the scope of this summary. Two rows initialized to all '0' or all '1' are reserved to copy into the control row; additionally, the three temporary rows are reserved in each subarray.

Modifications to DRAM required to implement bulk bitwise operations include the ad-

dition of a small decoder in each subarray to control the reserved rows, additional control signals in the memory controller to interface with the decoders, support for RowClone [36]. The estimate for a subarray with 1024 rows is no more than 0.5% less memory capacity.

The estimated results of bulk bitwise operations are promising. Two estimates are provided, an "aggressive" estimate where the overall latency of a bitwise operation is assumed to be 200 ns, and a "conservative" estimate where latency is assumed to be 340 ns (the aggressive method uses the additional decoder to speed up copying rows). An Intel i7-4790k processor with two 8 GB DDR3-1333 DIMMs on separate channels was used as a baseline for a comparison of bitwise AND operation throughput, measured in GB/s. The benchmark involved performing AND operations on two vectors of sizes ranging from 8 KB to 32 MB. The baseline processor performance significantly drops off as vectors grow to the point that they no longer fit in caches. Table 3.3 includes a summary of bitwise operation throughput. The rows labeled with a cache name describe the baseline performance when the working set fits entirely within the cache.

The greatest benefit of in-memory bitwise operations is the scalability. The throughput does not change with the size of the vector processed. Additionally bulk bitwise operations scale linearly with the inclusion of additional specially modified banks, whereas additional processor cores will not improve the baseline as the bottleneck is memory access. Providing additional processors does not change the speed of the DRAM bus, but modifying more DRAM banks allows bitwise operations in multiple banks simultaneously.

Energy consumption was estimated based solely on the cost of DRAM accesses. The conservative estimate had a 31.6X improvement over the baseline, and the aggressive estimate had a 50.5X improvement.

Table 3.3: Throughput comparison of bitwise AND operations

Device	Throughput (GB/s)
Aggressive, 2 banks	76.4
L1 cache	71
Aggressive	38.2
L2 cache	25
Conservative	22.4
L3 cache	15
Baseline	3.9

The authors also provide a more realistic benchmark. The FastBit bitmap index library is used in physics simulations and network analysis. Performance on a standard data set where the baseline spent an average of 31% of time on bitwise operations resulted in a 30% improvement using four aggressive estimate banks. A single bank with the conservative estimate still had an 18% performance improvement.

3.2 SRAM

Static random access memory (SRAM) is a volatile memory array that does not require refreshing. SRAM is commonly used in applications such as caches or register files. SRAM is faster than DRAM, and requires less peripheral circuitry, but is less dense and more expensive [30].

Figure 3.5 shows a standard six transistor SRAM cell. It consists of cross coupled inverters, each with an access transistor. Three signal wires control each SRAM cell: a word line, a bit line, and a bit bar line. The bit bar line is associated with the inverted section of the memory cell. The word line is connected to the gate terminal of both access transistors, and is used to select a word for operations. The bit line and bit bar line connect to sense amplifiers for reads and write drivers for writes.

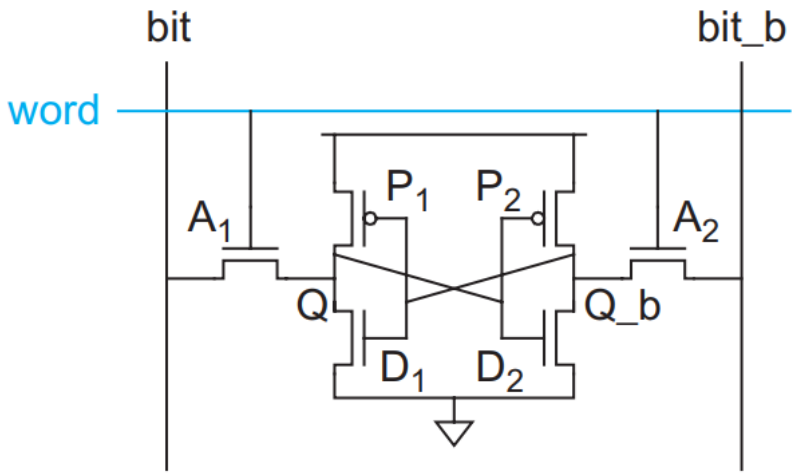


Figure 3.5: Standard six transistor SRAM cell [30]

Both bit lines are pre-charged high to perform a read operation. The bit line connected to the side of the cell storing a logic 1 will remain charged, while the other bit line will form a path towards ground through the access transistor and discharge.

SRAM cells are arranged in a two dimensional array (Figure 3.6). Word lines are selected by row decoder, and bit lines are selected by column decoder. Each column contains sense amplifier used to read the bit lines. Similar to DRAM (Section 3.1), SRAM is often divided into a hierarchy [30].

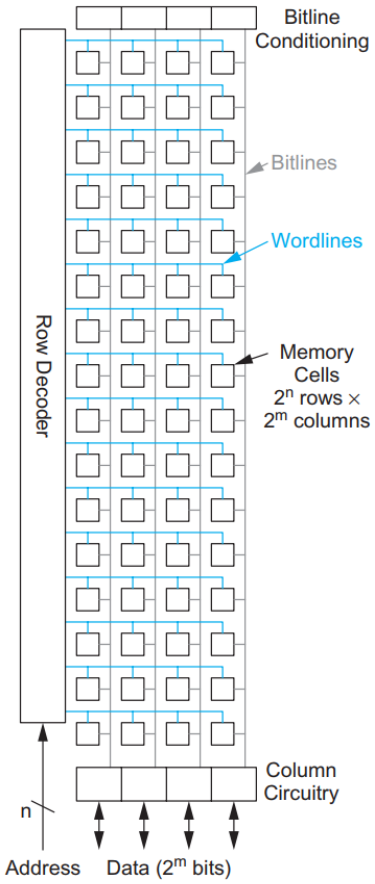


Figure 3.6: Example SRAM array [30]

3.2.1 DRC²

Akyel et al. [26] propose a PIM-implementation based on SRAM using a ten transistor cell called Dynamically Reconfigurable Computing Circuit (DRC²) [26]. The goal is to reduce power consumption caused by data transfer between memory and processor. In order to accomplish this, DRC² uses a highly parallel pipelined architecture. The provided example application is merging image buffers used in robotics to detect obstacles and estimate their velocities. DRC² is estimated to perform the image processing in as few as half as many clock cycles. An alternative implementation using a standard six transistor SRAM cell is

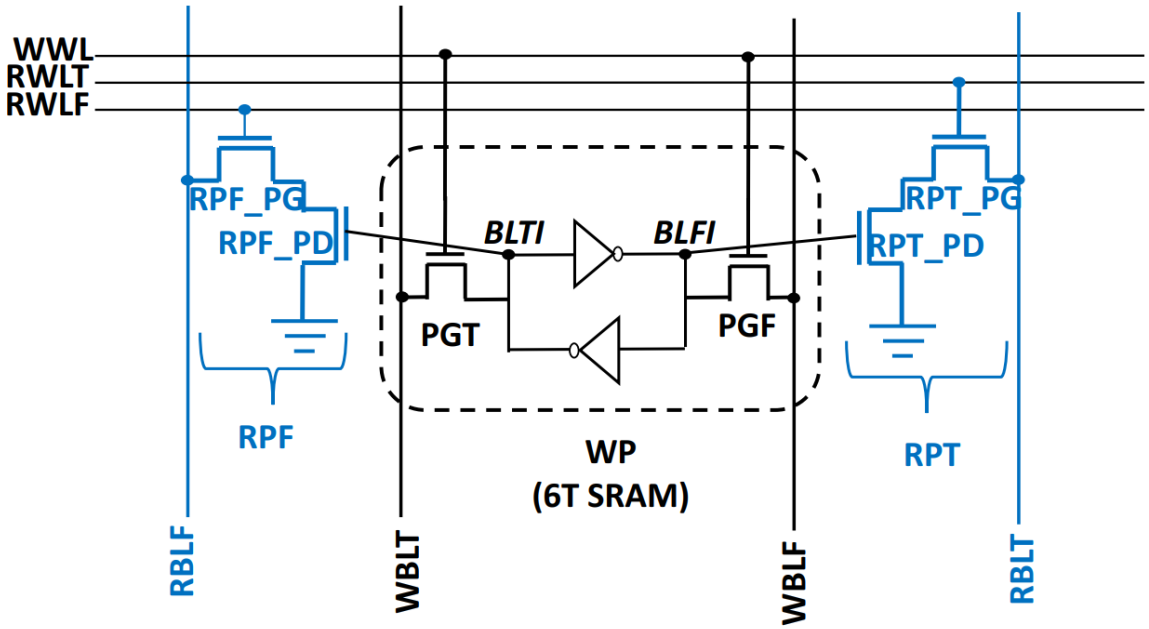


Figure 3.7: Ten transistor SRAM cell with separate read and write ports [26]

estimated to have a 1.5X performance improvement, but has less area overhead than a ten transistor cell. The authors also predict significant energy savings, but are unable to provide a quantitative value.

The SRAM cell (Figure 3.7) in DRC^2 uses two extra pairs of transistors to provide separate read and write ports. There are three word lines and two pairs of bit lines. Write Word Line (WWL) in conjunction with Write Bit Line True (WBLT) and Write Bit Line False (WBLF) allow independent writes; Read Word Line True (RWLT) works with Read Bit Line True (RBLT) and Read Word Line False (RWLF) works with Read Bit Line False (RBLF) in the same manner to allow independent reads.

DRC^2 follows the approach of selecting multiple word lines to perform computations in SRAM along a column. The architecture supports operations a basic ALU provides,

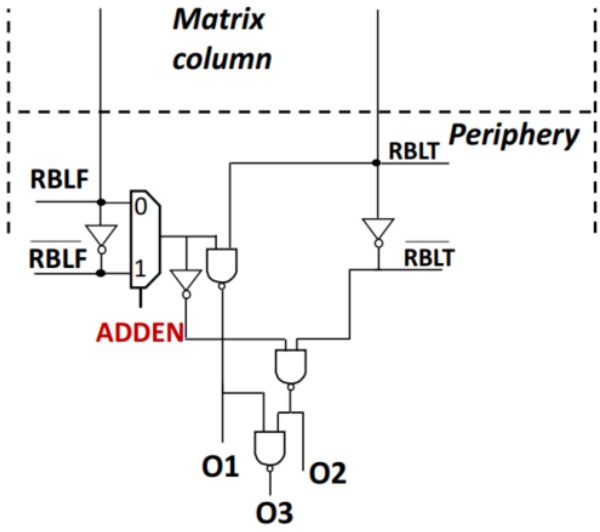


Figure 3.8: Additional column hardware enabling logic operations [26]

including bitwise operations like XOR, NOR, NAND, OR, and AND, but it also includes more complex instructions such as addition, subtraction, and shifting.

Bitwise operations including AND, OR, NAND, and NOR are performed using a similar technique as described in Section 3.1.3. Multiple word lines are selected in a column, and the final result on the bit line is equal to a logical operation between the selected words. For example, cells A and B in contain the logical values 1 and 0 respectively. If RBLT is charged, and RWLT is active for both cells, RBLT has a path to ground and goes to a logic '0'. Repeating the exercise for all values of A and B it become apparent that this operation is a logical AND of A and B. SRAM has an inverse bit line in addition to the normal bit line, allowing potentially two separate simultaneous bitwise operations from each column of cells. Additionally, the operations scale with multiple operands. N word lines in a column can be selected to perform a bitwise operation on N bits.

Other operations require additional hardware at the bottom of each column. XOR/NXOR, comparison, material implication, less than, and greater than use a multiplexer and a handful

of logic gates as pictured in Figure 3.8. Addition and subtraction require a full adder inside the SRAM, comprised of a multiplexer and three additional gates beyond the peripheral circuitry. The carry out signal of each adder is connected to the carry in of the adder in the next most significat column. For use in high frequency systems, the adder must be pipelined into three clock cycles to allow time for the carries to propagate.

The architecture presented is promising, with notable issues. The concept of memory configured to perform computation opens the door to exciting strategies such as exploiting parallelism and greater bandwidths inherent inside memory. Applications using a large amount of data implemented with parallel bit operations could benefit greatly from an architecture such as DRC^2. The caveat is that the authors do not provide an area overhead or cost analysis. Adding a full adder to the bottom of every SRAM column could come with significant overhead. DRC^2 is an interesting concept that needs more research.

3.2.2 Terasys

In 1995, Gokhale et al. [25] created an alternative to a high-performance SIMD computer using a PIM architecture. The motivation behind the design was to overcome the inherent limitations of SIMD systems: performance degrades on applications not well suited to parallel execution. Terasys aims to provide a platform with the advantages of SIMD for applications that can benefit from parallelism, without degrading for other applications. A flexible host processor allows general purpose computing, while SIMD processor arrays are designed as PIM that can be used as additional memory for operations that do not require SIMD instructions.

Terasys PIM chips are designed using a 4-bit memory with a single bit arithmetic logic unit (ALU) at the controlling each memory column. Eight megabytes of address space consisting of 32K single-bit ALUs were placed in a Sparcstation-2 workstation. The workstation consists of a Sparc-2 processor with a bus connecting to an external enclosure

with an interface board, and up to PIM array units containing 4K single bit ALUs each. PIM array units are made up of eight banks of eight chips each containing 64 ALUs.

Each PIM chip contains 2K by 64 bit SRAM, with a parallel prefix network and a logical OR network to allow communication between ALUs. An ALU can load data from memory or store it on each clock cycle, and produces three outputs for storage, recirculation, or routing. The networked architecture of Terasys allows the use of microcode to support complex operations. The SPARC processor is capable of sending two commands per 200 nanoseconds, each command consisting of 25 bits of microcode instruction. The interface board holds a 4K lookup table that is filled with instructions generated for each program.

Terasys is strongly coupled to a parallel programming language devised with the architecture in mind called data-parallel bit C (dbC). Included is a microcode assembler that will generate a lookup table for every program written. dbC includes special instructions for memory allocation, interprocessor communication, and data movement to take advantage of Terasys.

The performance of Terasys is compared to the Cray-YMP supercomputer. Programs developed for Terasys ranged from 5 to 50 Cray-YMP single processor equivalent performance. For example, Terasys was able to generate $2 * 10^10$ pseudorandom bits per second (20 Cray-YMP equivalents). The authors claim to "deliver supercomputer performance for a small fraction of supercomputer cost". Additionally, each PIM bank can be used as additional memory for the SPARC host processor to perform operations that do not benefit from parallel execution.

3.2.3 Intelligent SRAM

Embedded Intelligent SRAM (ISRAM) is a near-memory enhancement of SRAM proposed by Jain et al. [37] targeted at pipelined Reduced Instruction Set Computer (RISC) processors used in embedded systems. ISRAM places an ALU and an accumulator near on-chip SRAM,

and provides several additional instructions to allow data movement between the processor register file, accumulator, and SRAM. The design relies on the fact that words in the on-chip SRAM are stored with multiple words to the same row; in order to read a single word, the whole row is read, and then the desired word is selected. This allows the ISRAM ALU to read two words in a single clock cycle with minimal additional hardware. Figure 3.9 contains an overview of the ISRAM architecture.

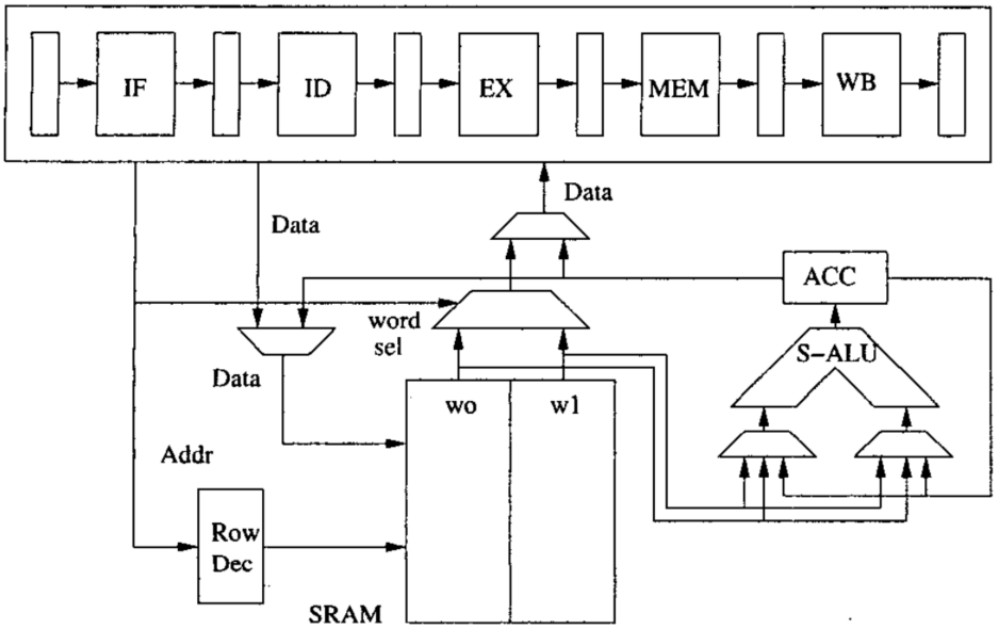


Figure 3.9: ISRAM hardware architecture [37]

The hardware is designed to accelerate an ARM9TDMI processor, which has a five stage pipeline. The pipeline is unmodified, save to include direct memory access (DMA) style instructions to control the ISRAM ALU. Assuming an SRAM latency of one cycle, ISRAM calculations do not need to stall the processor pipeline. In an unmodified system, a load instruction followed by an operation on the loaded data could require a pipeline stall, because

the load instruction requires a cycle to produce valid data. ISRAM allows operands to be loaded from memory and used in calculations during a single instruction with no stalling.

ISRAM requires operations to be partitioned between the host processor ALU and the added hardware. The authors present algorithms for dividing instructions, and for mapping data so the ISRAM ALU can make the best use of it. They estimate the additional hardware requires about 3000 gates, and does not include multiplication. The results are shown in Figure 3.10, using eight example applications, and differing assumptions about the length of SRAM latency.

Benchmark	SRAM latency = 1 cycle			SRAM latency = 2 cycles			SRAM latency = 3 cycles		
	Orig (cycles)	New (cycles)	Speedup	Orig (cycles)	New (cycles)	Speedup	Orig (cycles)	New (cycles)	Speedup
`plus`	1438	1438	1.00	1738	1638	1.06	2040	1840	1.10
`VADD`	803	603	1.33	1105	805	1.37	1407	1007	1.39
`AVG`	609	409	1.48	710	510	1.39	811	611	1.32
`leaf_comp`	904	704	1.28	1207	907	1.33	1510	1110	1.36
`MatrixAdd`	2437	1925	1.26	3208	2440	1.31	3979	2955	1.34
`SHA1`	3002	2746	1.09	3339	3083	1.08	3676	3418	1.07
`MD5`	896	832	1.07	1024	960	1.06	1152	1088	1.05
`GSM_DEC`	800	600	1.33	1100	900	1.22	1400	1200	1.16

Figure 3.10: ISRAM performance summary [37]

ISRAM offers modest improvements to the chosen benchmarks. In particular it is well suited to applications such as matrix and vector addition. The near-memory approach of a building a complete ALU inside a memory bank is suited to embedded applications where capacity is a major concern, however the architecture probably will not scale. For larger applications, architectures that take better advantage of in-memory bandwidth by placing hardware in memory columns include designs described in Sections 3.2.1 or 3.2.4.

3.2.4 Recryptor

Zhang et al. [19] implemented a PIM accelerator for cryptographic applications for Internet of Things devices called Recryptor. Recryptor is a solution that replaces other accelerators, such as application specific integrated circuits (ASICs), reconfigurable hardware, or a traditional co-processor. Cryptographic operations typically have a high bit width, and make frequent use of bitwise operations, making them an ideal candidate for PIM architectures.

The authors propose a new architecture, dubbed "Recryptor", which is essentially a co-processor using a memory-centric architecture. Instead of a traditional processor, with registers, ALU, and program counter, it uses a specialized SRAM bit cell with interspersed logic. Recryptor was tested with Elliptic curve cryptography (ECC), a form of public key cryptography, Advanced Encryption Standard (AES), a form of secret key cryptography, and the Keccak hash SHA-3 hashing algorithm with significant energy efficiency improvements over standard approaches. Additionally, the authors claim to be the first to accelerate public and secret key and hash functions at the same time.

Recryptor is designed using an ARM Cortex-M0 microcontroller with 32 kB of memory, with one 8 kB memory bank replaced with a custom SRAM implementation referred to as a crypto-SRAM bank (CSB). It uses a ten transistor bit cell to support dual read ports; by accessing two words simultaneously, bitwise logical operations can be performed in one clock cycle. These bitwise operations are referred to as in-memory computation, while a shifter, rotator, and substitution box (S-box) placed physically near the CSB provide near memory computation.

Figure 3.11 is a diagram of the ten transistor bit cell used in the CSB. For a read operation, only the read bit line (RBL) is pre-charged; for a logical 1, the line will be discharged, and for a logical 0 it will remain high. In order to perform bitwise logic, multiple word lines are selected at the same time. A or B is performed by charging the read bit line,

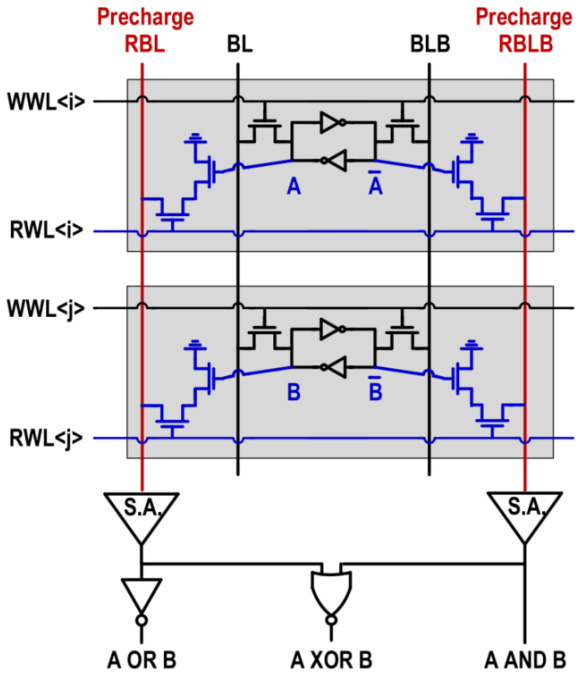


Figure 3.11: Ten transistor SRAM cell used in Recryptor [19]

and if either is a logic 1, then the line will discharge, the equivalent of A NOR B. An inverter at the bottom of the column turns this into A OR B. A AND B works in a similar fashion, using instead the read read bit line bar (RBLB). A NOR gate between the two bit lines allows A XOR B if both lines are charged high. All of these operations are performed in a single cycle.

Another innovation to speed up cryptographic computations is how memory "sub"-banks are configured. The CSB is arranged in 16 slices with 128 32 bit words. Sub banks can be enabled to allow for up to 512 bit single cycle logic operations. This gives a clever programmer the means to parallelize cryptographic algorithms with high bit width words.

The "near memory" devices allow single cycle operations important for cryptography beyond bitwise operations. A shifter allows for various shift operations based on the sub bank used; the design is accomplished with multiplexers and is area efficient, but wiring intensive. A rotator allows arbitrary 64-bit rotation using two stages, a 0-7 stage and a multiples of 8 stage. It uses a standard barrel shifter. An S-box is also included; an S-box is a cryptography specific component used for byte substitution in block cyphers. These near memory devices function like accelerators, providing functionality that would otherwise require multiple clock cycles to perform with a general purpose central processing unit (CPU).

The authors describe programming cryptography algorithms on Recryptor. Modularity of the design, allowing multiple algorithms of different general classes, is a focal point of the design. The implementation details are less exciting. The results are more interesting. For the ECC algorithm, Recryptor performs in 330 estimated operations what takes a standard Cortex-M0 software implementation an order of magnitude more operations. Specific performance improvement results are not listed for AES or Keccak functions, but implementations are provided. Recryptor programming is done using what are essentially direct memory access instructions to a memory-mapped decoder.

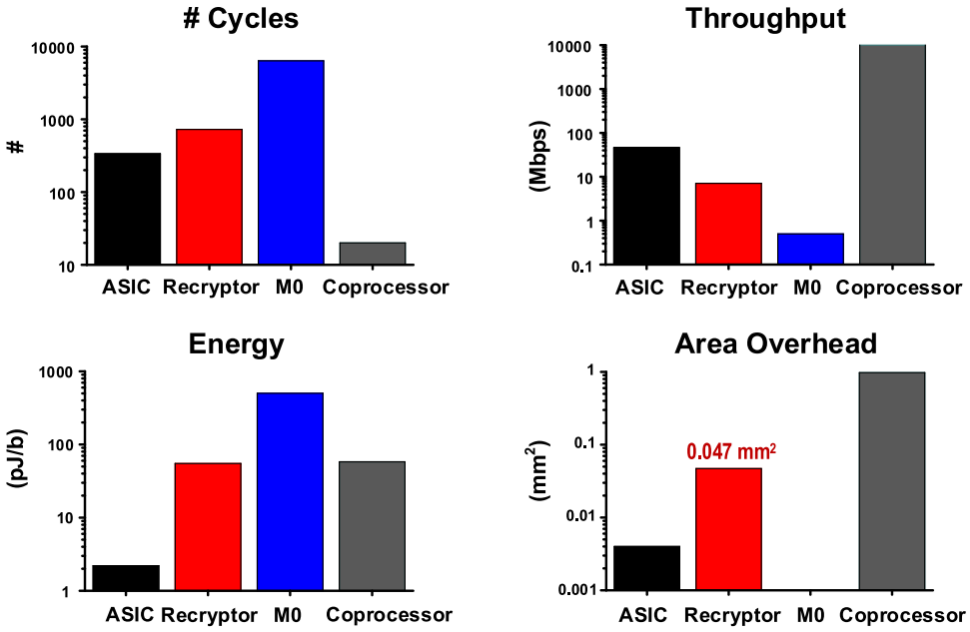


Figure 3.12: Relative performance comparison of Recryptor [19]

Testing was done using real chips fabricated with a 40-nm Complementary Metal Oxide Semiconductor (CMOS) process. The CSB resulted in an increase in memory area from 55.000 µm^2 to 180.000 µm^2, or an area overhead of 36% beyond a standard ARM Cortex-M0. Bit cells used were from the standard cell library. The authors speculate they could achieve an 18% area overhead given a custom bit cell design.

Recryptor shows promise accelerating cryptographic applications for embedded devices. Figure 3.12 shows relative results. Recryptor significantly improves performance over the baseline while using less energy and area than a co-processor. ASICs outperform Recryptor, as expected, but are not programmable, and are expensive to develop. The concept behind computation in memory is exciting for applications beyond just cryptography. Recryptor is particularly noteworthy because results are based on a fabricated chip, not just simulations.

The design could be applied as an alternative computer architecture to the standard Harvard model being used for general purpose applications.

3.3 Resistive Memories

Resistive memories are an emerging nonvolatile memory technology relying on variable resistance cells to represent logical values. Cells change resistance based on polarity, magnitude, and duration of a current or voltage applied, typically between a high resistance and low resistance to represent a binary value. A typical configuration includes a word line connected to an access transistor, a bit line for sensing data, and source line to allow write operations [23].

Redox Random Access Memory (ReRAM) cells are memristive devices using an oxidation/reduction reaction as the switching mechanism. ReRAM is desirable for its scaling, endurance, and retention properties [22] and is a strong candidate for PIM due to its capacity, fast read speed, and computation capability [24]. A ReRAM cell consists of two metal electrodes with an insulating oxide layer as the switching material (Figure 3.13). By applying an external voltage, a cell may be switched between a high resistance state (HRS) and a low resistance state (LRS). Changing a cell to HRS (a SET operation), representing a logical '1', is requires a positive voltage high enough to generate a sufficient write current. Switching a cell to LRS (RESET operation), representing a logical '0', is done by applying a voltage of high enough magnitude negative voltage [24].

ReRAM used in PIM is frequently arranged in a crossbar structure [22,23,38]. A crossbar contains two sets of parallel lines, with each set placed perpendicular to the other (Figure 3.14). The lines are conductive metal that act as electrodes for memristive material that forms a junction. Each junction is a singular ReRAM cell accessed via the crossbar. Writes are performed by applying the correct voltage bias across the junction, while reads involve applying a relatively low voltage (low enough to avoid an accidental write) and differentiating

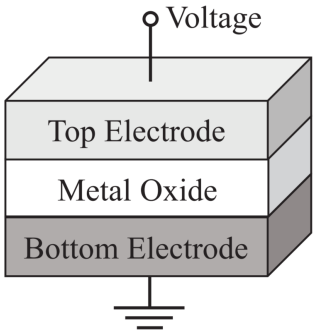


Figure 3.13: ReRAM cell [23]

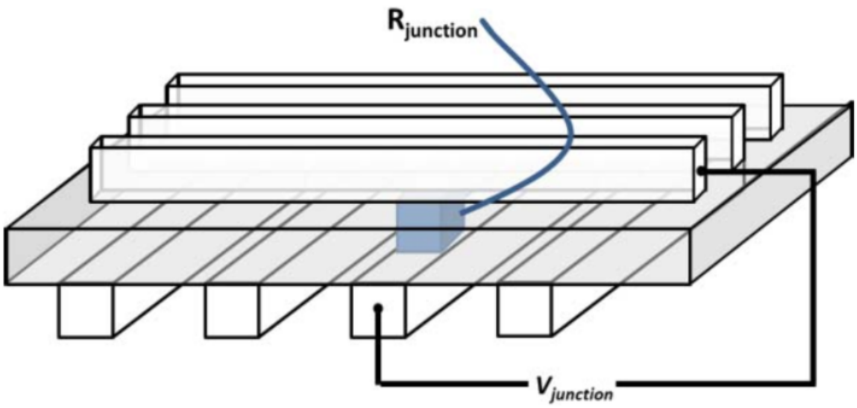


Figure 3.14: ReRAM crossbar [23]

between LRS and HRS using Ohm's law. If multiple cells are selected simultaneously, logical operations can be performed by exploiting the electrical properties inherent to the crossbar, similar to the techniques using multiple SRAM word lines described in Sections 3.2.1 or 3.2.4.

ReRAM has been considered as a replacement technology for DRAM because ReRAM potentially requires less area and energy for the same performance [22]. ReRAM has a comparable read latency, but significantly longer write latency. Recent improvements to

ReRAM architectures have bridged the gap to as small as a 10% write performance difference between ReRAM and DRAM [24].

3.3.1 PRIME

PIM is a strong candidate for use as a neural network (NN) accelerator because neural networks fit the criteria of requiring large amounts of memory access with relatively small computational effort on each access. Chi et. al. [24] propose PRIME, a PIM architecture used for neural network computation inside ReRAM. PRIME seeks to maintain large memory capacity and memory bandwidth by moving computation into the memory chip.

Artificial neural networks are a class of machine learning algorithms consisting of a network of neurons, with multiple layers, and weighted synapses. Traditional CMOS-based NNs have substantial design challenges related to the area occupied by the thousands of transistors making up neurons and synapses. ReRAM crossbars have been proposed as a denser method of constructing synaptic arrays required for NN computations. A ReRAM crossbar represents input data as analog input voltages on the word lines. Synaptic weights are programmed into cell conductances of the crossbar. Current flowing through each bit line is a matrix vector multiplication. A non-linear function is applied after sensing the bit line.

Additional peripheral hardware is required to realize a ReRAM NN. Digital-to-analog converters (DACs) and analog-to-digial converters (ADCs) are needed for analog computing, a sigmoid performs the non-linear function, and a subtraction unit combines positive and negative weights implemented as separate crossbars. PRIME is unique as a ReRAM NN implementation because it allows ReRAM crossbar arrays to operate either for NN computation, or to be used as memory when other functions are not necessary.

PRIME partitions ReRAM banks into three regions: memory subarrays, full function (FF) subarrays, and buffer subarrays. Memory subarrays are used for data storage only,

and are the same as conventional ReRAM subarrays. FF subarrays can be used for either NN computation, or data storage. Buffer subarrays are caches for FF subarrays, located physically nearby, and can also be used for storage when not in use as a buffer. An additional controller is necessary to control FF and buffer subarrays.

To realize PRIME FF subarrays, the decoder and driver, column multiplexer, and sense amplifier of a traditional ReRAM cell are modified. The decoder and drivers include additional hardware to allow switching between computation and memory mode, and to support more levels of input voltage. The column multiplexers incorporate analog processing units in the form of a subtraction unit, and a sigmoid unit. Sense amplifiers are configured to support up to eight bits of precision, and include logic circuitry to perform NN computations. Buffer subarrays include additional decoders and multiplexers to allow FF subarrays to access any memory location, and include an intermediate storage register to allow conditional bypass of buffer subarrays. Modified sense amplifiers and write drivers serve the same purpose as ADCs and DACs would in other NN ReRAM designs, saving space by incorporating signal conversion functionality into existing peripheral circuitry.

PRIME was evaluated using a test bench of several well known NNs of varying size (small, medium, and large) and a suite of simulation softwares. The baseline is a four core, 3 GHz processor with out-of-order execution, two levels of cacheing and 16 GB of ReRAM-based main memory. Also included are a PIM solution using 3D integration, with 64 parallel neural processing units, and a co-processor accelerator. Across NNs, the PIM solution achieves an average speedup of 9.1X over the co-processor, and PRIME achieves a speedup of 4.1X over the PIM solution (37.3X over the co-processor). In terms of energy efficiency, PRIME is between two and five orders of magnitude better than the baseline, and two orders of magnitude better than the PIM solution.

The area overhead of PRIME is minimal. In a bank of 64 subarrays, including two

FF subarrays and one buffer is a 5.76% overhead. Given the impressive performance and minimal area overhead, PRIME could become an effective architecture for NN acceleration if ReRAM becomes a viable memory technology.

3.3.2 Spin-Transfer Torque Magnetic RAM

One of the primary motivations for PIM is energy efficiency. Modern multicore systems have high memory demands, devoting significant area and power to memory. Jain et al. [18] propose a PIM system based on spin-transfer torque magnetic RAM (STT-MRAM) for energy efficient, high performance computing called spin-transfer torque compute-in-memory (STT-CIM).

STT-MRAM is a form of resistive memory that changes resistance based on the orientation of a magnetic tunnel junction (MTJ) accessed via a transistor (Figure 3.15). Each MTJ has two layers, one with a fixed magnetic orientation, one with a changeable orientation. The layers are separated by an oxide, and the MTJ changes between resistances based on whether the layers share orientations or oppose each other. Opposite orientations have higher resistance, representing logic '0', while similar orientations have lower resistance, representing logic '1'.

Read operations are performed by applying a voltage bias between bit line (BL) and source line (SL) while the word line (WL) is selected. The resulting current is compared to a reference to determine the state of the MTJ. Write operations are performed by passing a sufficient current through the MTJ. The direction of the current determines the logical value written.

Similar to many in-memory PIM implementations using traditional memory, STT-CIM relies on selecting multiple word lines simultaneously to perform logical operations. Enabling multiple word lines causes the resulting current flowing through the SL to be the summation of the current flowing through each bit cell. There are three possible SL currents: if

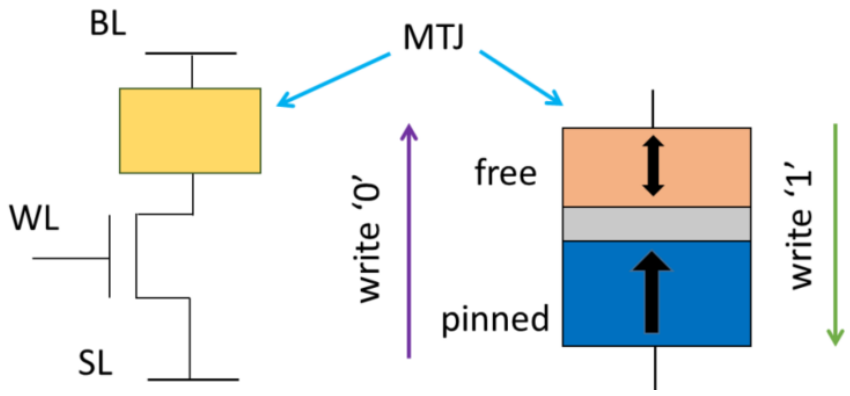


Figure 3.15: STT-MRAM cell [18]

both MTJs are '0', the current is a relatively low value, and if both MTJs are '1', the current is relatively high, with a mixture resulting in a current halfway between. Instead of comparing to the standard reference current, a special current is used for each logical operation. Comparing SL to a reference current halfway between the low current and the middle current using a fully differential amplifier results in a logical OR on the positive output of the amplifier and a NOR on the negative output. The only case where the SL current is lower than the reference current is when both MTJs are '0', resulting in an OR operation. The paper also details a similar method for logical AND/NAND operations, and describes using additional gates in conjunction with the current sensing method to realize bitwise XOR and in memory addition. Addition operations require three additional logic gates placed near the sense amplifiers, while XOR operations require a singular OR gate.

The benefits of using STT-MRAM for PIM operations as opposed to similar techniques using traditional memory such as SRAM based implementations (Section 3.2) include energy efficiency and memory density. STT-MRAM is non-volatile, and does not suffer from leakage currents or high power consumption typical of traditional memory. The modifications necessary for STT-CIM do not affect the core bit cells or memory arrays. All modifications

are made to peripheral circuitry, allowing PIM operations without affecting standard memory design parameters such as density or read/write operation efficiency. Additionally read currents used for PIM operations are significantly lower than write currents, preventing accidental writes while performing PIM operations.

To support PIM operations, STT-CIM modifies peripheral circuitry. A special "CiM-Type" input signal is fed to a decoder to signify the PIM operation being used. Read circuitry is modified to support additional reference currents, and row decoders are modified to support selecting multiple WLs. While row decoders require approximately double the area to a conventional STT-MRAM system, the authors estimate that only contributes a 1.8% increase in overall chip area. Write operations are unaffected and do not need additional hardware.

Sensing circuitry is also extended. Each column of memory requires two fully differential sense amplifiers, four logic gates, and three multiplexers. Again the authors note that the power and area overhead associated with enhancements are minimal compared to the overall system.

STT-CIM has tighter read margins during PIM operations than a standard read on STT-MRAM. Monte Carlo simulations of the proposed circuit considering process variations in each MTJ show that an error correction scheme is necessary to reliably use STT-CIM. Triple error correction, quadruple error detection using Hamming codes is sufficient for reliable operation. Error correction does introduce overhead in the form of additional hardware, and reduces overall performance. Each error correction requires three memory array accesses. Even with the overhead, errors are infrequent enough that the performance benefit of STT-CIM is still worthwhile.

To further improve performance, vector operations are supported. STT-CIM exploits internal memory bandwidth to speed up data-parallel computations. For instance, in the

case where every element of two size N arrays are added together, rather than use N PIM operations, a single vector PIM operation is used. A near-memory reducer supports operations such as summation and Euclidean distance to address bandwidth issues returning vector values. The reducer reduces results to a scalar value when possible before storing the result. In order to limit overhead, vector operations are limited to vectors of either size four or eight elements.

STT-CIM is controlled by a host processor using an extended instruction set architecture (ISA). The host controls PIM operations by sending control signals including the operation code, the operand addresses, and the destination address.

There are three criteria to operate on elements in STT-CIM. Elements must be in the same bank, they must be in separate data rows, and they must be in the same column. The authors describe techniques for data placement to facilitate using PIM operations. Three methods for data mapping are presented, and shown to be useful in real-world applications. While mapping data results in some small overhead, the assumption is that on a large enough problem arranging the data once in the beginning is negligible compared to the overall run time.

STT-CIM performance was measured using a cycle-accurate register transfer level (RTL) simulation. A suite of 12 algorithms from various applications such as cryptography, text processing, and optical character recognition forms the benchmark for an Intel Nios II processor with a one megabyte STT-CIM scratchpad and eight element vector operations. The baseline system assumed a standard STT-MRAM as main memory. Area overhead was 16.6% larger than the baseline, mostly because of the additional error correction hardware. Memory access latency overhead was 0.8% due to greater WL and BL delays. Overall performance improvements included an average of 3.83 times energy improvement and 3.93 times system performance at an application level.

The results Jain et al. present are promising. The area overhead introduced by STT-CIM is minimal, and memory function is largely unaffected because modifications focus on peripheral circuitry as opposed to the memory array. If STT-MRAM becomes a viable alternative to DRAM for main memory, STT-CIM is a viable architecture for data-parallel loads.

3.3.3 Computation-in-Memory Parallel Adder

There are many criteria used to evaluate a computer architecture - power consumption, manufacturing cost, and relative speed, among others. Major improvements to computers, as measured by these criteria, have largely been due to improvements in the electrical characteristics as opposed to improvements in designs. As the limits of materials used in manufacturing computers are reached, in order to continue to create better devices, different architectures must be designed. The obvious improvement, parallelizing the computing process, comes with its own set of difficulties. Nguyen et al. [29] address some of those problems through the use of memristors in order to perform computation in memory.

The Von Neumann gap is infamous in the world of computer architecture. It refers to the memory bottleneck that exists within modern computers. Modern processors are much faster than modern memory, leading to a bottleneck and wasted time while the processor waits for memory to catch up with a request. Du Nguyen et al. identify "a need for a novel architecture that significantly reduces the memory bottleneck, massively supports parallelism, and is energy efficient". A recent technology, the memristor, may offer a partial solution. The electrical characteristics of a memristor are less important for the purposes of understanding this paper; it is enough to know memristors are nanometer scale devices that hold non-volatile memory in an analog manner by changing resistance values. A computation in memory (CIM) architecture is described with an implementation of a parallel addition algorithm to sum inputs in chunks. Considered are the passive memristors used to implement the adder, active CMOS circuitry required to control and communicate with the adder, and

a peripheral interface to act between the CMOS and memristor components. Two types of memristor cells are used: complementary resistive switching (CRS) and bipolar resistive switching (BRS). BRS uses one memristor and a high or low resistance value to represent logical 1 or 0. CRS contains two stacked memristors and alternates the resistances values to represent logical 0 or 1. CRS cells require CMOS logic to transfer values between cells as the total resistance of low or high is the same, while BRS cells may suffer from sneak path currents, where bits may be misread as resistance values on separate paths sum to an equivalent R value.

The read operation for a memristor is destructive; the act of reading changes its value. Thus a "write-back" instruction must be issued to restore the value after a read. This requires CMOS control logic to trigger. Depending on what kind of cell is used, more or less additional circuitry will be required to use a memristor array. CRS cells require more sense amplifiers (one for each crossbar), and muxes to distinguish which bit on a word is being read. It takes more BRS cells overall to realize an adder circuit than CRS, thus BRS cells require more voltage drives with more strength. The largest factor in the delay and area of the proposed memristor adder circuits is the CMOS controllers required to use them. Because of this, the designs scale well to an increased size, as the majority of space/time tradeoffs are a result of the overhead. Overall delay is about one order of magnitude worse than a traditional CMOS multicore design, although authors explain this as partially due to optimistic results regarding caching in their simulation. The memristor designs however are up to two orders of magnitude more energy efficient, and five orders of magnitude more area efficient. The memristor parallel addition design shows promise for in memory logic. It is shown to be scalable, and to apply to other forms of highly parallel logic aside from addition. The main caveats that come with these exciting results is that memristors are new, and largely unresearched. They do not have the same standard libraries as CMOS designs. They

are not suitable for general purpose computing at the moment, as their overall endurance is poor, but some researchers are optimistic about improvements to endurance. Fabrication processes still need to be researched in order to physically realize these devices. With enough additional research, it is conceivable that memristor based memory architectures will become useful.

3.4 Three Dimensional Integration

Three dimensional (3D) integration is a manufacturing technology that involves stacking multiple layers of connected integrated circuits. Chips are stacked on top of each other and joined using through silicon vias (TSVs). Benefits to 3D integration include reduced power consumption, reduced noise, improved memory density and performance, and the ability to create new architectures [39]. 3D integration uses shorter wires, lowering load capacitance and resistance to improve energy efficiency and noise tolerance. Stacking chips allows greater memory capacity in the same sized footprint to traditional two dimensional designs. Finally, stacked chips enable PIM by tightly coupling logic and memory together.

Connecting logic to memory with a TSV facilitates PIM. TSV bandwidth is greater than bandwidth available between chips connected by pins. Stacking chips eliminates pin count concerns, and allows for high bandwidth, low latency memory [40]. Additionally, DRAM peripheral circuitry can be moved to the logic layer, where timings are better. While 3D integrated solutions do not place logic and memory in the same wafer like other PIM implementations, the overall effect is similar. TSVs allow computer architects to take advantage of internal memory bandwidth to perform data parallel operations, much like any other PIM strategy.

3.4.1 Tesseract

The massive memory bandwidth available to 3D integrated architectures lends itself to big data applications. Ahn et al. [14] propose Tesseract, a design which applies PIM concepts to a 3D integrated memory design for accelerating large scale graph processing. Tesseract seeks to enable "memory-capacity-proportional performance", a feat which traditional multi core architectures fail to provide.

Tesseract utilizes the Hybrid Memory Cube (HMC), a 3D-stacked DRAM standard. HMC provides up to 320 GB/s of external memory bandwidth through eight high-speed serial links. Multi core machines fail to scale with such a large bandwidth and are unable to utilize the full potential performance. Motivated by the challenge of a system that scales, Tesseract applies concepts of PIM to utilize the bandwidth of HMC. Internal memory bandwidth of a HMC is even greater than external bandwidth, at 512 GB/s, further motivating PIM.

Tesseract is functionally a network of HMCs. Figure 3.16 shows a conceptual overview of Tesseract. It is built on using HMCs with eight, eight GB DRAM layers. Each cube contains 32 vertical slices referred to as a vault connected via a crossbar network. Each vault contains a 16 bank DRAM partition with a dedicated memory controller and an in-order core placed on the logic layer, for a total of 32 cores per cube. The majority of total die area is taken by DRAM, with all 32 cores taking up an order of magnitude less area. ARM Cortex A5 processors are used for vault cores.

Tesseract acts as an accelerator to a host processor. Tesseract is memory-mapped to the host processor, allowing the host to access any region of the total system. The memory-mapped region contained by the cubes is not cached or virtualized to avoid complexity and overhead. The host processor is responsible for distributing graphs to vaults using a specialized malloc call.

Tesseract cores are restricted to accessing only their own DRAM. Cores communicate

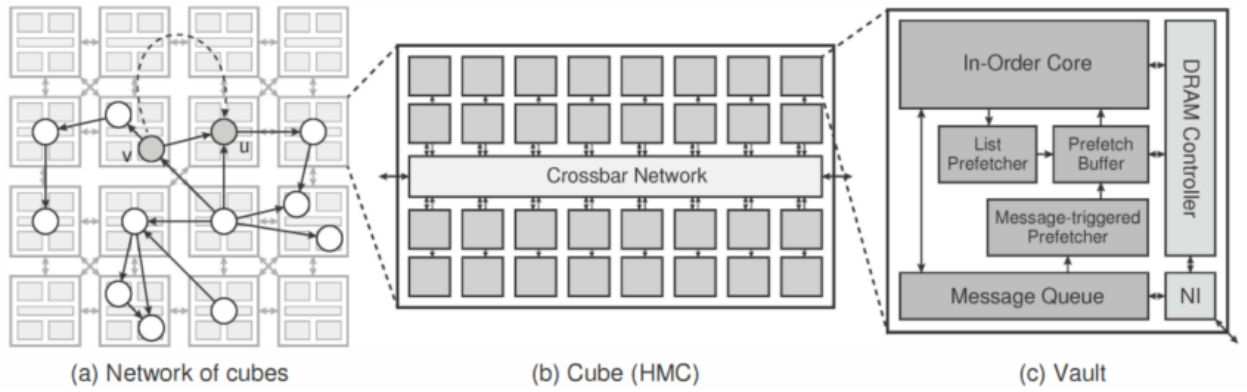


Figure 3.16: Tesseract architecture (not to scale) [14]

via a message passing mechanism. Computation is implemented using remote function calls. Remote memory access is disallowed. Unlike traditional architectures, rather than passing data for processing, Tesseract passes code for execution.

In order to make use of the full HMC memory bandwidth, Tesseract uses two hardware prefetch mechanisms. Vault cores prefetch using a standard stride prefetcher, and through a message-triggered prefetch. When a function is passed to a core, a message is included to trigger prefetch before the function is executed.

Tesseract was evaluated using a cycle accurate x86-64 simulator. The baseline high-performance system used 32 four GHz four core out of order processors with 128 GB of DDR3 DRAM for a total of 102.4 GB/s of memory bandwidth accessible by the cores. The simulation of Tesseract used a network of 16 HMCs with a total of 512 2 GHz in order cores. The benchmarks used were a series of five graph algorithms implemented in C++: Average Teenage Followers, Conductance, PageRank, Single-Source Shortest Path, and Vertex Cover. Three input graphs were used, with the largest containing 7.4 million vertices and 194 million edges, for a total of up to 5 GB of memory footprint. Tesseract's

average system performance was found to be ten times better than the baseline, with an 87% reduction in energy consumption.

Tesseract's model of communicating code instead of communicating data between processor cores is innovative, and creates a system that scales well with additional cores. One of the shortcomings of traditional multi core architectures is a failure to scale as more cores are added. Placing cores on top of data, and using a high bandwidth 3D integrated memory allows near linear scaling with additional cores. Tesseract is a promising solution to large graph processing problems, and demonstrates the value of PIM concepts applied to 3D integration for supporting big data applications.

3.4.2 HAMLeT

Providing higher bandwidth, lower latency memory on its own is not sufficient to solve the memory wall problem. 3D die integration provides scalable main memory for big data applications, but for an application to run efficiently, it must take advantage of the memory architecture. Akin et al. propose Hardware Accelerated Memory Layout Transform (HAMLeT) [31, 40], a hardware accelerated memory transform framework. By efficiently reorganizing data, HAMLeT makes use of a 3D integrated memory architecture to provide energy efficient, low latency computation.

HAMLeT transforms memory layout to allow a host processor to more efficiently compute. A high level summary of HAMLeT's operation is reading data into the logic layer, reorganizing it locally, and then writing it back into stacked DRAM. Figure 3.17 shows a typical 3D stacked DRAM (using HMC as an example) next to the HAMLeT modifications. Three main components handle layout transformation operations: local buffers, buffer interconnections, and a control unit. Many existing 3D integrated DRAM systems already include an interconnection fabric in the form of a crossbar switch. HAMLeT extends the crossbar with fast SRAM buffers and a control unit.

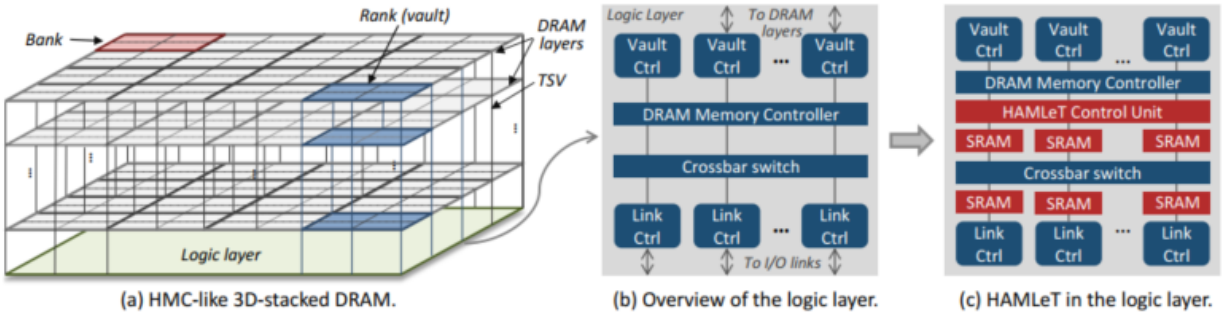


Figure 3.17: HAMLeT [40]

SRAM buffers act essentially as caches for data transformation. Buffers are sized according to the DRAM configuration to minimize row buffer misses and avoid penalties for common memory access patterns. HAMLeT and the host processor access memory in parallel, so that transformation and computation do not interrupt each other. Parallel memory access is achieved through carefully planned access patterns, such as block by block reorganization where the host and HAMLeT operate on separate blocks at the same time.

Some data intensive applications require data transformation as part of computation. For example, linear algebra or signal processing applications often use matrix transpositions or 3D matrix rotations. These operations are expensive to perform due memory access patterns that are inefficient on traditional memory hierarchies. HAMLeT provides methods for matrix transposition, matrix blocking, and 3D-matrix rotation. By utilizing buffers capable of keeping pace with the massive memory bandwidth, and exploiting fine-grained parallelism inherent with 3D-integrated memory with multiple vaults, HAMLeT offers efficient forms of matrix transformations.

HAMLeT was modeled using CACTI-3DD, a modeling framework designed for 3D-integrated DRAM. The three matrix transpositions described above were simulated on HAMLeT, an Intel Xeon E3-1230 processor, and an Nvidia GTX 670 GPU. The Xeon

processor has a peak memory bandwidth of 21 GB/s, the GPU has a bandwidth of 192 GB/s, and the HAMLeT implementation has a peak bandwidth of 201 GB/s. HAMLeT performed on average an order of magnitude better than either the processor or GPU. Both the processor suffer from high memory access latency, and were only able to utilize a fraction of their peak memory bandwidth. HAMLeT however is better suited to utilize high memory bandwidth, and maintained above 90% peak bandwidth usage throughout the benchmarks.

The overhead involved with HAMLeT is minimal. The authors estimate 2-7% of the area of each logic layer is required for a HAMLeT implementation, with 3-5.5% increase in power consumption based on an HDL synthesized implementation produced in the Synopsis Design Compiler.

HAMLeT presents an application specific accelerator using 3D stacked DRAM. For computing requiring high bandwidth, low latency memory access, 3D-integration is a suitable technology. HAMLeT is an example of leveraging the benefits for 3D-integration with specialized logic for a specific application.

Chapter 4 FPGA-based PIM Simulation

This thesis's contributions are to create an FPGA implementation of a PIM architecture that extends an existing microcontroller. It is important to note the implementation models the control flow and design of a PIM architecture, but does not include modified memory. Traditional block memory is attached to a module performing PIM operations, as opposed to a true PIM implementation, where the memory itself is partially used to perform logical operations. Presented is a proof of concept design focusing on architectural features necessary to realize a PIM system, without focusing on the detailed hardware implementation.

The organization of this chapter is as follows: Section 4.1 contains an overview of the base microcontroller architecture and the PIM modifications, Section 4.2 describes the added instructions, Section 4.3 discusses the example application of AES encryption, and Section 4.4 contains a comparison of the performance of the base microcontroller versus the PIM-enabled implementation. Additionally, Chapter 5 goes into greater detail concerning design trade offs surrounding the results, including a discussion of limitations of the design and future work to mitigate some of the limitations.

4.1 Hardware Architecture

The physical hardware this project runs on is a Nexys DDR4 development board with a Xilinx Artix-7 FPGA. Table 4.1 includes an overview of the technical specifications of the Artix-7. The Artix-7 was chosen because the performance versus price met project requirements, and for the Vivado Design Suite software's availability for academic use. The hardware for the project is implemented using the Very High Speed Integrated Circuit Hardware Description Language (VHDL).

Table 4.1: Artix-7 FPGA Specifications

Part Number	XC7A100T
Logic Cells	101,440
Slices	15,850
CLB Flip Flops	126,800
Distributed RAM (Kb)	1,188
Block RAMs (36 Kb each)	135
Clock Management Traces	6

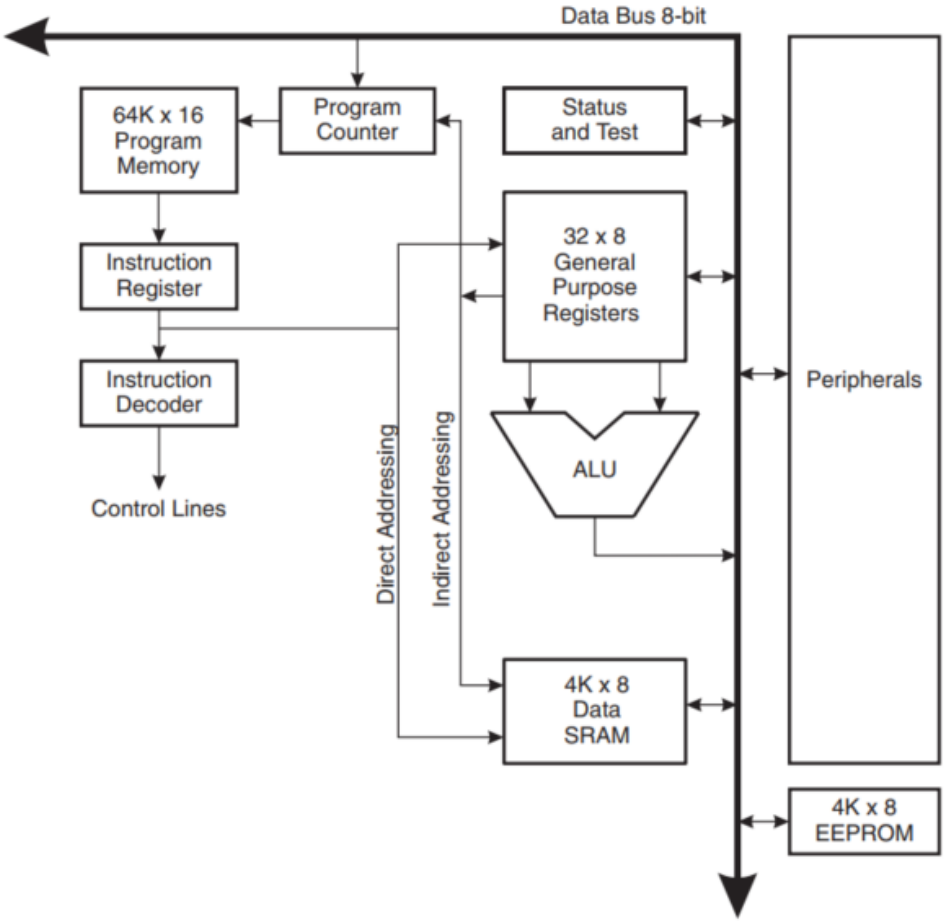
4.1.1 ATmega103(L) Microcontroller

An ATmega103(L) microcontroller [41] forms the basis of the PIM implementation, and as a baseline comparison. The ATmega103 is a CMOS 8-bit microcontroller based on the AVR RISC ISA [42]. The microcontroller is a modified Harvard architecture with separate program and data memory sized, 128 kilobytes and 4 kilobytes respectively, and 32 general purpose registers. Instructions follow a two stage pipeline, where on the first clock cycle instructions are fetched and decoded, and on the second clock cycle are executed and written back to the destination. The ATmega103 is well suited as a base for highlighting new architectural features due to its relatively small footprint, and the relative simplicity of the two stage RISC pipeline. Additionally, the AVR ISA is well documented, with strong community support including an open source tool chain in the AVR gcc compiler and related softwares.

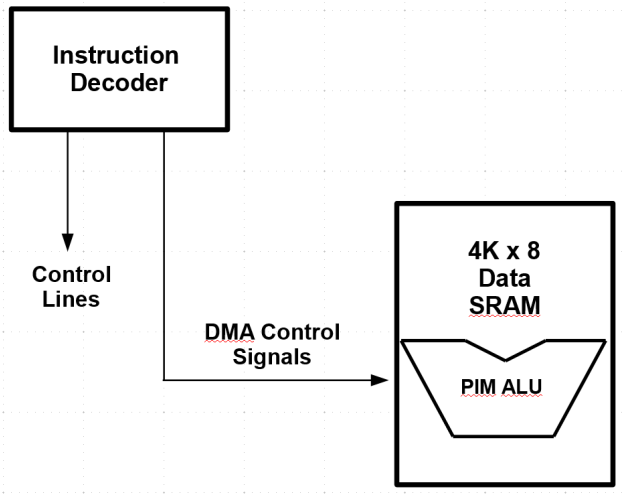
The actual FPGA implementation of the microcontroller, referred to from this point on as a soft core, was taken from an open source soft core by Ruslan Lepetenok [43]. Additionally the project includes a counter for tracking clock cycles. The counter is enabled by setting the output value of a general purpose input/output (GPIO) pin, allowing for software run time information accurate to the clock cycle. Figure 4.1 (a) includes an example of the soft core top level architecture.

4.1.2 PIM Modifications

In order to realize PIM instructions, the ATmega103 is modified with a special Data SRAM containing an ALU. The SRAM is controlled by the instruction decoder, modified to provide DMA style instructions. When a PIM operation is detected, control signals are generated to directly interface with the memory from the decoder. The control signals govern the operations, performed directly inside the memory and stored internally without passing information along the data bus. Figure 4.1 (b) highlights the components modified from the original soft core to create the PIM extended soft core.



(a) ATmega103 block diagram [41]



(b) PIM modifications

Figure 4.1: ATmega103 architecture and PIM modifications

Figure 4.2 shows a more detailed view of the PIM extended SRAM alluded to in 4.1 (b). There are seven input signals: the address used by the unmodified portion of the soft core, input data, write enable, two addresses used for PIM operations, a PIM enable signal, and the PIM operation code. The included memory is a true dual port RAM with two read and write capable ports. The "A" port is used as the memory access port for the microcontroller. The "B" port is used exclusively internally for performing PIM operations. A PIM operation functions by selecting the two PIM addresses for ports A and B. The memory feeds the output of both ports to an internal ALU, performing an operation based on the op code selected. The PIM address A and the ALU output are latched, and on the next clock cycle fed to port B to write the result into memory.

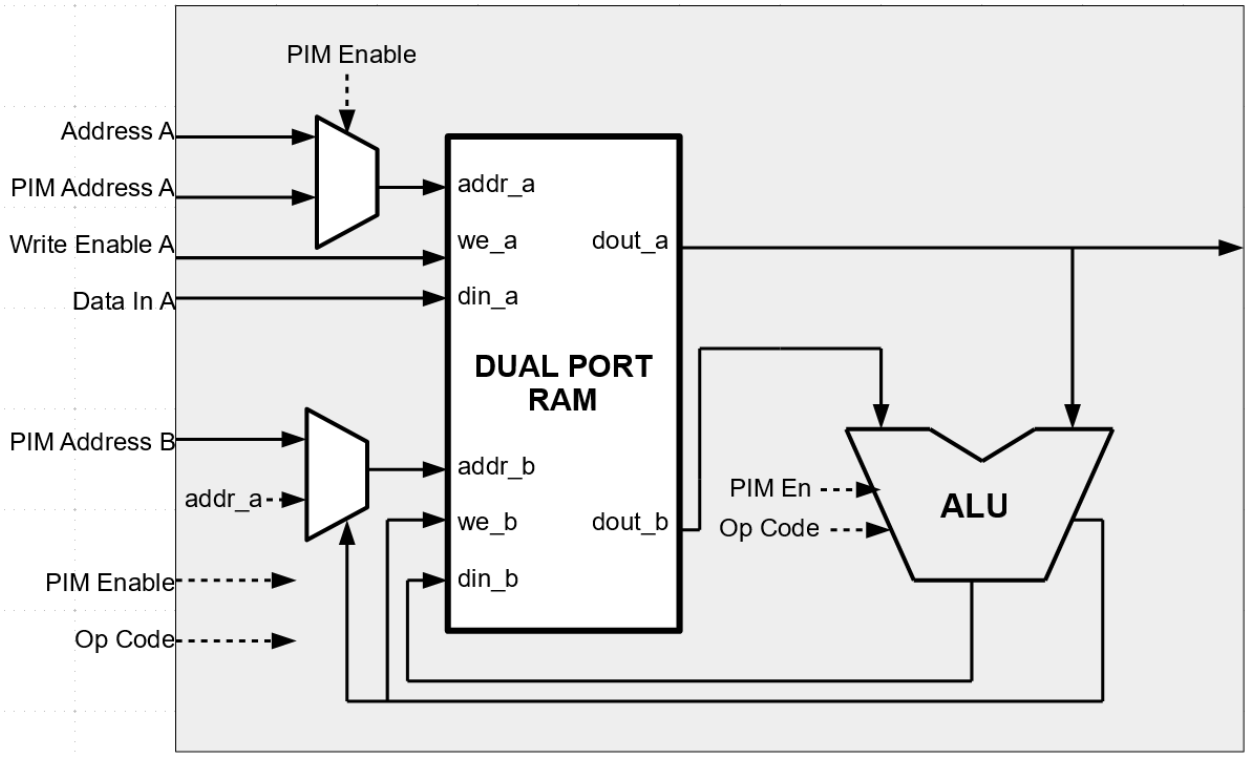


Figure 4.2: ATmega103 data memory with PIM modifications

4.2 Processor in Memory Instructions

4.2.1 Instruction Format

All PIM instructions take the form of a two word instruction. The ATmega103 uses two byte words, therefore PIM instructions are four bytes long in total. Every instruction contains two 12 bit addresses, and a four bit op code used by the PIM ALU. Figure 4.3 describes the instruction format. In order to integrate with the existing microcontroller, reserved op codes are re-purposed as PIM instructions. The AVR ISA has reserved words from op codes `0x0001` to `0x00FF`, so the first byte of every PIM instruction is `0x00`. The next byte of the first word is divided into the upper nibble of address A and the PIM op code. The second word is divided into the lower byte of both addresses A and B. The final four bits required, the upper nibble of address B, are stored in register 30. This means that PIM operations can only be performed on 255 address blocks before register 30 needs to be set to a new value, an operation that requires one clock cycle. There are 16 such blocks in the ATmega103 SRAM.

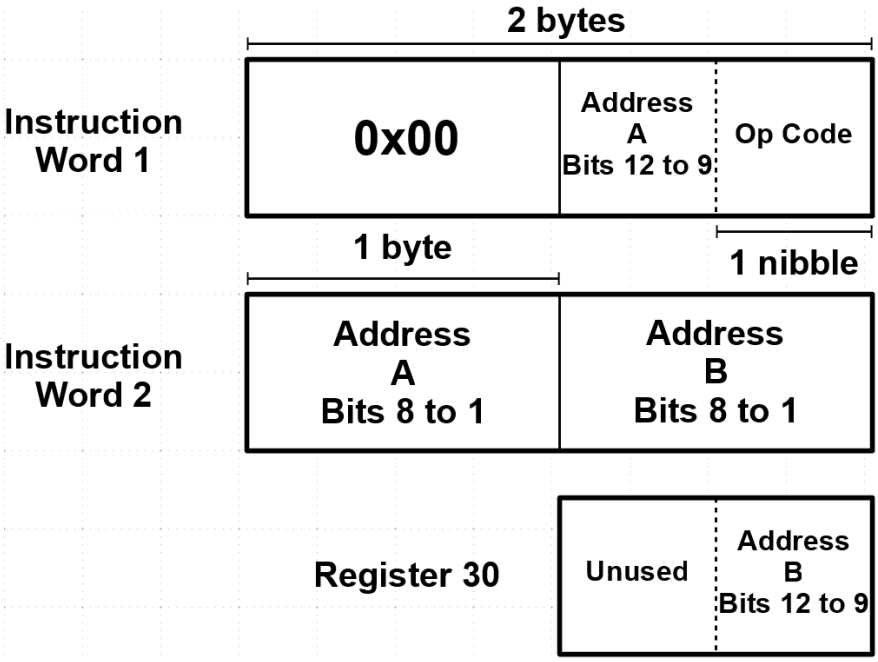


Figure 4.3: Format of a PIM instruction

A list of all supported PIM operations and their opcodes is detailed in Table 4.2. Operations are limited to bitwise logic, addition, subtraction, and one special operation: Move. Move writes the value of operand B to address A, effectively performing an in-place copy and paste of memory location B to memory location A. It is important to note that a hardware implementation of addition and subtraction would require a full adder to be placed beneath every column of SRAM, with the carries chained from the least significant bit to the most significant bit. This thesis' implementation operates under the assumption that given the relatively slow clock frequency of the microcontroller (16 Mhz), the carries in the adder chain have plenty of time to propagate and produce a result before the writeback on the next clock cycle. In systems with a clock frequency on the order of GHz, addition/subtraction would need to be done in multiple steps with latches for intermediate values in order to allow the carries to propagate.

Table 4.2: Opcode List

Operation	Opcode	Operands	Description
Nop	0000	No operands	No operation
Not	0001	Operand A	Return the negation of the value at address A
Move	0010	Operand B	Write data word B to address A
XOR	0011	Operands A & B	XOR data words at addresses A and B
NXOR	0100	Operands A & B	Negation of XOR of data words A and B
NOR	0101	Operands A & B	NOR data words at addresses A and B
NAND	0110	Operands A & B	NAND data words at addresses A and B
OR	0111	Operands A & B	OR data words at addresses A and B
AND	1000	Operands A & B	AND data words at addresses A and B
Add	1001	Operands A & B	Add data words at addresses A and B
Subtract	1010	Operands A & B	Subtract data words at addresses A and B
Increment	1011	Operand A	Increment data word at address A
Decrement	1100	Operand A	Decrement data word at address A

4.2.2 Instruction Pipeline

In the AVR ISA, most instructions take one clock cycle, with some multi-word instructions requiring additional clock cycles. PIM instructions follow suit, requiring two clock cycles for a two word instruction. In reality, instructions take more clock cycles, as they are divided into a two stage pipeline, with a fetch/decode stage, and an execute/write-back stage, but instruction throughput is as though instructions took fewer cycles due to pipelining. Figure 4.4 illustrates a sample pipeline containing a PIM instruction. On the first cycle, the instruction is decoded. On the next cycle, the second word is decoded, and execution begins. On the third clock cycle, the result of the PIM operation is written back into memory. Performing 16 PIM operations in a row takes 33 clock cycles, two cycles per operation plus an extra at the start of the pipeline when the first word is decoded.

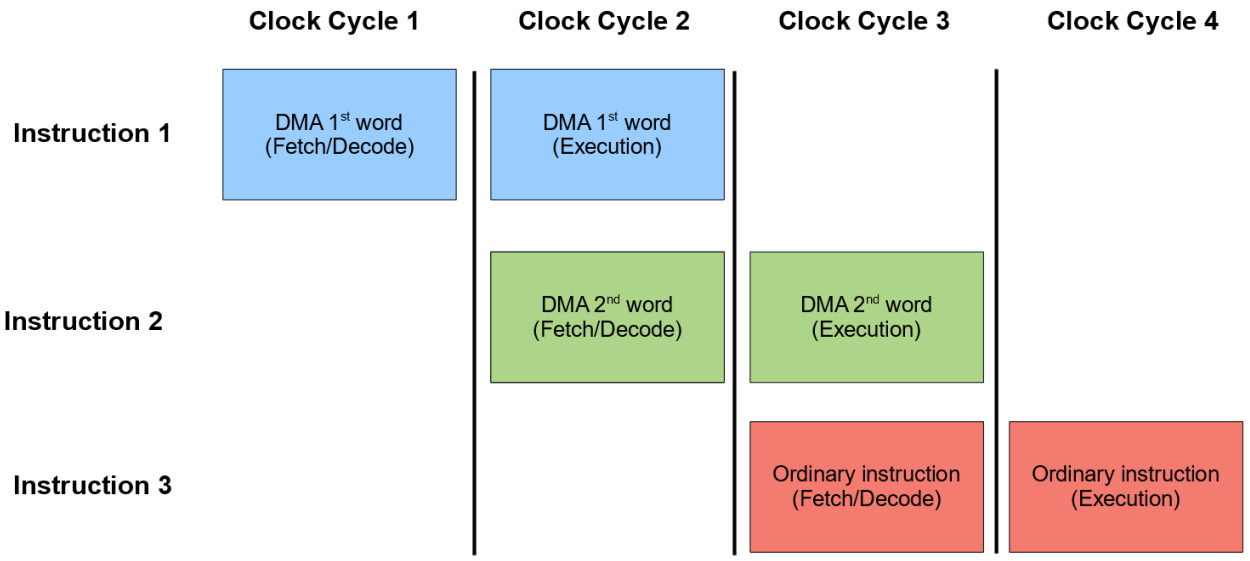


Figure 4.4: Example pipeline of a PIM instruction

4.2.3 Typical PIM Operation Usage

Figure 4.5 contains an AVR assembly snippet for a typical programming pattern easily replaced with a PIM operation. Values are loaded into registers 0 and 1 from memory locations 0x100 and 0x200 respectively. The registers are added together, and then stored back in memory location 0x100. Each load or store operation takes two clock cycles, while the addition takes one, for a total of seven clock cycles. This entire process can be replicated in a single PIM operation, which adds two memory locations together directly without loading or storing values. The PIM operation takes two clock cycles, for a savings of five cycles. In the case that register 30 does not already contain the upper nibble of address B (in this case 0x2), one clock cycle is required to set the proper value in the register, for a savings of only four cycles.

In some cases, the final goal is to load a set of values into the registers. If there are enough general purpose registers to contain all the values for a given program or function,

PIM operations do not save any time. Some applications, however, operate on more values, and require frequent load/store operations. One such application, Advanced Encryption Standard (AES) encryption, which operates on 16 byte blocks, is detailed in Section 4.3.

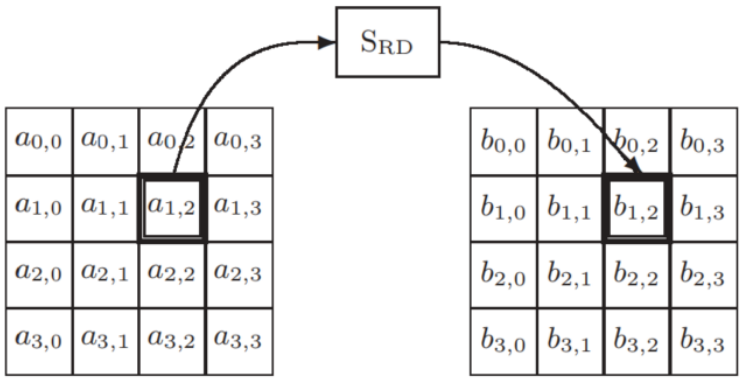
```
LDS R0, 0x100 ;Load direct from data space
              ;memory location 0x100 to
              ;register 0 (2 word instruction)
LDS R1, 0x200 ;Load direct from data space
              ;memory location 0x200 to
              ;register 1 (2 word instruction)
ADD R0, R1 ;Add registers 0 and 1 and store
           ;the result in register 0
STS 0x100, R0 ;Store direct to data space
              ;register 0 to memory location
              ;0x100 (2 word instruction)
```

Figure 4.5: Example usage

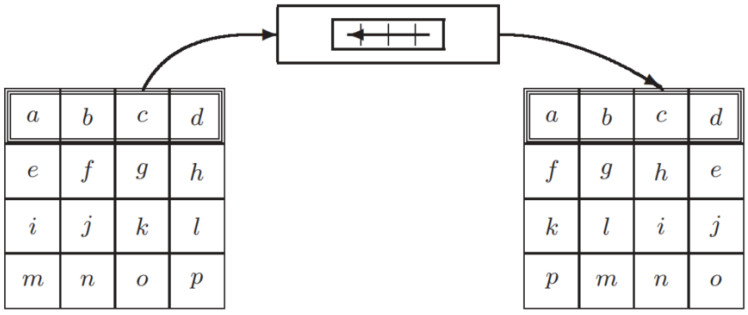
4.3 Example Application

AES, also referred to as a Rijndael block cipher, is one of the most widely used symmetric key encryption algorithms [44]. AES can be performed using a 128, 192, or 256 bit key in 10, 12, or 14 rounds respectively, although this implementation only uses a 128 bit key. To start, the initial 128 bit key is used to derive a separate key for each round. The plaintext message to be encrypted is stored in a 4x4 matrix referred to as the state. The state is combined with the first round key using a bitwise **xor**. Then a series of transformations are performed on

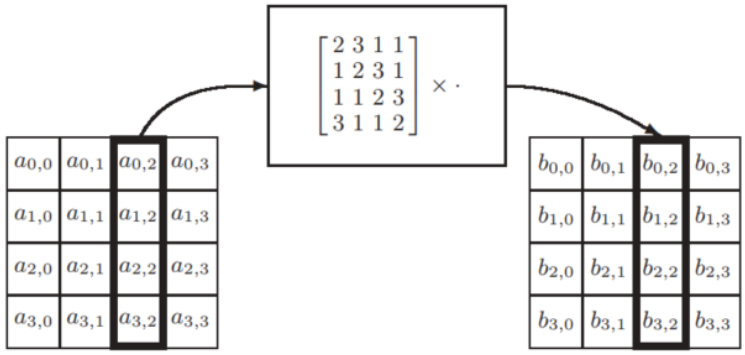
the state in each round: SubBytes (Figure 4.6 (a)), ShiftRows (Figure 4.6 (b)), MixColumns (Figure 4.6 (c)), and AddRoundKey (Figure 4.6 (d)). SubBytes involves substituting each byte of the state with a byte from a pre-computed lookup table referred to as a substitution box, or S-box. ShiftRows shifts each byte in a row cyclically by an offset. MixColumns transforms the columns of the state by multiplication with a matrix (this can be performed as a series of shifts followed by a conditional xor with 0x1B). Finally, AddRoundKey xors the state with the current round key. In the final round, the MixColumns step is omitted. The resulting state is the cipher.



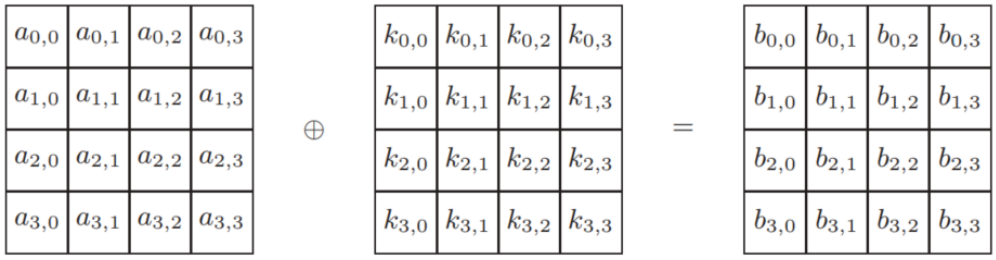
(a) SubBytes



(b) ShiftRows



(c) MixColumns



(d) AddRoundKey

Figure 4.6: AES round steps illustrated [44]

AES is well suited to PIM operations because it satisfies two basic criteria: AES operates on more values than can be stored in the register file, and relies on a series of simple bitwise operations. Both the state and each round key are 16 bytes long, meaning during each round 32 bytes must be loaded and stored. For example, the final step of each round, AddRoundKey, would require loading each byte from memory for both the state and the round key, xoring the values together, and storing the result back to memory. Using PIM operations, the operation is performed in just two clock cycles per byte, as opposed to five. Section 4.4 compares the performance of AES using the base ATmega103 versus a PIM extended microcontroller.

4.4 Results

4.4.1 Software Comparison

One 16 byte cipher was calculated on both an ATmega103 soft core, and the PIM enhanced implementation. The AES code is written in C, based on a GPL licensed library [45]. In both cases the code was compiled using the AVR-gcc compiler. PIM operations are inserted as inline assembly using the .word directive to define individual words of memory with the PIM instruction. This is necessary because the compiler and assembler have no knowledge of PIM operations. The source code for both the standard AES program and the PIM AES program can be found in Appendix A.

PIM operations significantly improve the performance of AES encryption, at the cost of program size. Table 4.3 shows the base system compared to the PIM enhanced implementation. Discrepancies in clock cycle totals come from minor differences in how the compiler optimizes code while profiling because the hardware based counter still requires code to enable and disable the GPIO pin controlling the counter. Key expansion for all 10 rounds takes the majority of the time in this implementation of AES, and does not include any

PIM operations. Key expansion is only performed once; subsequent blocks use the same key. After the key schedule is created, encryption using PIM operations takes 38.7% less time. Encrypting more blocks of plaintext will result in similar time savings per block, and similar memory overhead per block.

While the memory overhead is significant, both programs require small amounts compared to the overall available program memory. The ATmega103 has 128Kb of program memory, and the PIM encryption requires just under 3% of the total program memory. In modern times where memory capacity is larger and cheaper, requiring a larger program could be considered an acceptable trade off.

Table 4.3: Performance comparison of AES encryption on base ATmega103 and PIM enhanced ATmega103

	ATmega103	PIM	Difference
Total AES (clock cycles)	16,069	12,418	22.7% less time
AES encryption only (clock cycles)	9,461	5,797	38.7% less time
Key Expansion clock (cycles)	6,610	6,619	–
Program Memory size (bytes)	2,424	3,740	54.3% more memory

The performance presented in 4.3 is not entirely fair. In order to offer a true comparison, one must consider that PIM operations must be inlined. Most of the time savings from PIM operations occur when xoring the key with the state at the end of each round. Figure 4.7 contains the xor_block function, and Figure 4.8 contains the generated assembly (function overhead is omitted for the purpose of illustration). In total, xor_block takes 181 clock cycles to run, and requires 13 words of program memory. The PIM equivalent xor_block (Figure 4.9) operates directly on the memory locations of the state and key, takes 34 clock cycles to

run, and requires 33 words of program memory. The standard implementation uses pointers to operate on arrays, and loops to increment the pointer. If instead it is unrolled by using a pointer to every single array location, it takes 117 clock cycles to perform the equivalent of the xor_block function. The trade off between the PIM approach and the standard approach is that the standard approach can be called on any two arrays via a function call at the cost of some overhead, whereas the PIM approach must be repeated inline in memory with direct addresses that are known at compile time.

```
void xor_block (uint8_t * d, uint8_t * s)
{
    for (uint8_t i = 0 ; i < N_BLOCK ; i++)
    {
        *d++ ^= *s++ ;
    }
}
```

Figure 4.7: Function to xor two 16 byte arrays

```
ldi r18, 0 ; initialize iterator
ld  r24, Z ; load destination value
ld  r25, X+ ; load source value
eor r24, r25 ; xor values together
st  Z+, r24 ; store xor result in destination memory location
subi    r18, 0xFF ; increment r18 by 1
cpi r18, 0x10 ; compare iterator to 16
brne    .-14  ; branch if not equal (branch goes to ld r24, Z)
ret ; return from function
```

Figure 4.8: Generated assembly for function in Figure 4.7

```
//set register 30 for upper nibble of B address
asm volatile("ldi r30, 0""\n\t":::"r30");
//0x3 is the PIM op code for xor
asm volatile(".word 0x0003, 0x6080""\n\t"::);
asm volatile(".word 0x0003, 0x6181""\n\t"::);
asm volatile(".word 0x0003, 0x6282""\n\t"::);
asm volatile(".word 0x0003, 0x6383""\n\t"::);
asm volatile(".word 0x0003, 0x6484""\n\t"::);
asm volatile(".word 0x0003, 0x6585""\n\t"::);
asm volatile(".word 0x0003, 0x6686""\n\t"::);
asm volatile(".word 0x0003, 0x6787""\n\t"::);
asm volatile(".word 0x0003, 0x6888""\n\t"::);
asm volatile(".word 0x0003, 0x6989""\n\t"::);
asm volatile(".word 0x0003, 0x6A8A""\n\t"::);
asm volatile(".word 0x0003, 0x6B8B""\n\t"::);
asm volatile(".word 0x0003, 0x6C8C""\n\t"::);
asm volatile(".word 0x0003, 0x6D8D""\n\t"::);
asm volatile(".word 0x0003, 0x6E8E""\n\t"::);
asm volatile(".word 0x0003, 0x6F8F""\n\t"::);
```

Figure 4.9: PIM implementation of function in Figure 4.7

PIM operations requiring direct addressing is a major limitation in the implementation. All memory locations to be accessed through a PIM operation must be known at compile time, preventing the use of common programming constructs such as loops or array offsets. Consider the code snippet in Figure 4.10; it appears to be a strong candidate for improvement with PIM operations, as it contains more values than could fit in the register file, and uses

```
f2(x) ((x) & 0x80 ? (x << 1) ^ 0x011B : x << 1);
void mix_sub_columns (int dt[16], int st[16])
{
    int j = 5 ;
    int k = 10 ;
    int l = 15 ;
    for (int i = 0 ; i < 4 ; i += 4)
    {
        int a = st [i] ;
        int b = st [j] ;  j = (j+N_COL) & 15 ;
        int c = st [k] ;  k = (k+N_COL) & 15 ;
        int d = st [l] ;  l = (l+N_COL) & 15 ;
        int a1 = s_box(a);
        int b1 = s_box (b);
        int c1 = s_box (c);
        int d1 = s_box (d);
        int a2 = f2(a1);
        int b2 = f2(b1);
        int c2 = f2(c1);
        int d2 = f2(d1);
        dt[i]   = a2    ^  b2^b1  ^  c1     ^   d1;
        dt[i+1] = a1    ^  b2     ^  c2^c1  ^   d1;
        dt[i+2] = a1    ^  b1     ^  c2     ^d2^d1;
        dt[i+3] = a2^a1 ^  b1     ^  c1     ^   d2;
    }
}
```

Figure 4.10: Logical values in the mix_sub_columns function are dependent on run time addresses, preventing the use of PIM operations

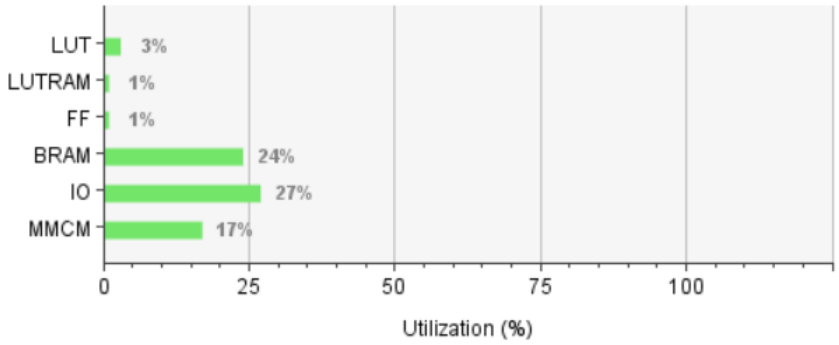
a series of xor operations. The problem is that the s_box function returns a value from an in-memory lookup table, accessed using the current state as the index. The state value is only available at run time, thus a PIM operation cannot be addressed to use a value from the s_box lookup table. All of the logical operations in the function rely on values indexed by the state, preventing the use of PIM operations. Alternatives to mitigate the limitations introduced by the addressing mode of PIM operations are discussed in Chapter 5.

4.4.2 Hardware Comparison

The other design comparison to consider is the differences in the hardware itself. Adding PIM operations to a microcontroller results in area overhead on the chip in the form of additional logic gates. The hardware overhead of the PIM enhanced ATmega103 (Figure 4.11 (b)) is minimal compared to the baseline (Figure 4.11 (a)). Neither design uses significant FPGA resources compared to the total available, and the only resource the PIM implementation uses significantly more of is IO blocks. For FPGA based soft core microcontrollers, given the limited resource overhead for noticeable performance improvements, PIM is a viable architectural strategy.

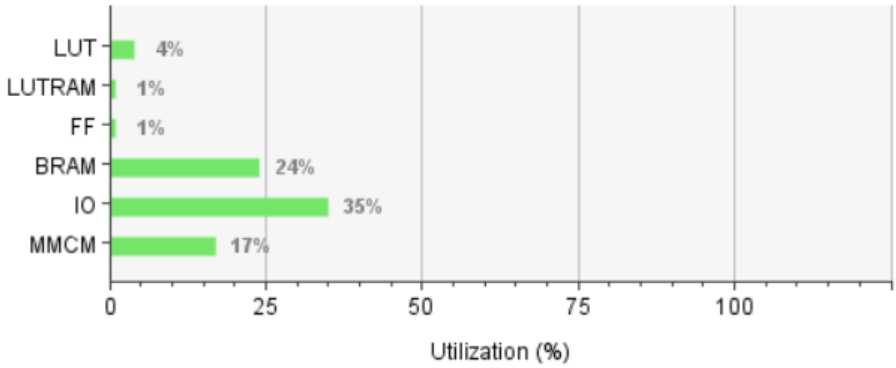
It is important to note that on an ASIC, the hardware area overhead is more significant. Based on similar works that fabricated PIM ASICs (for example 3.2.4), 10-20% area overhead is a reasonable estimate for an ASIC implementation.

Resource	Utilization	Available	Utilization %
LUT	2153	63400	3.40
LUTRAM	4	19000	0.02
FF	1294	126800	1.02
BRAM	33	135	24.44
IO	57	210	27.14
MMCM	1	6	16.67



(a) Base ATmega103

Resource	Utilization	Available	Utilization %
LUT	2242	63400	3.54
LUTRAM	4	19000	0.02
FF	1337	126800	1.05
BRAM	33	135	24.44
IO	73	210	34.76
MMCM	1	6	16.67



(b) PIM enhanced ATmega103

Figure 4.11: FPGA resource utilization

Chapter 5 Conclusions and Future Work

PIM computer architectures show promise in two main problem domains: applications bottlenecked by memory latency, and applications bottlenecked by register availability. Modern applications involving large data sets, such as machine learning or DNA sequencing, do not scale well with traditional von Neumann architectures, where the amount of data accesses overwhelms the computational effort. PIM provides a mechanism for memory bandwidth proportional performance for data hungry applications. On a smaller scale, PIM also shows promise to accelerate applications that fill a register file. In the case where a value must be loaded to a register, operated on, and stored back to memory in order to free the register, PIM speeds up execution by performing operations directly in memory. This thesis's contributions focused on this second domain, enhancing an existing microcontroller with a set of PIM operations. Section 5.1 discusses limitations of the implementation and potential ways to mitigate them, and Section 5.2 provides suggested future work.

5.1 Implementation Limitations

The major limitation of this thesis's implementation of PIM operations on an ATmega103 is it only supports direct addressing. PIM operations can only be performed on addresses known at compile time, and do not support accessing an array by an offset. For instance, the common construct of iterating over an array is impossible. A ten element array could only be operated on by placing ten sequential PIM instructions in program memory, instead of using a few instructions to iterate over any size array with a for loop. Two potential solutions to introduce indirect addressing are changing the instruction format of PIM operations, or using a PIM program memory.

The instruction format for PIM operations does not lend itself to run time addressing

because addresses are stored in program memory, which cannot be changed. If the addresses were instead stored in a location capable of changing at run time, such as the register file, indirect addressing would be possible. The trade off is that some amount of the register file must be reserved, and there is overhead involved setting up address registers. Storing two 12 bit addresses would require three registers. Using the base ISA, it would take a clock cycle to set each register, nullifying much of the time savings involved by performing in memory operations. Alternatively, if only 8 bits of each address were stored in registers, PIM operations could be performed on 255 address long blocks. The block position would need to be known at run time, but the offset within the block could be set at run time. In that case, adding every value of two arrays would involve a PIM operation followed by incrementing each address register, repeated in a loop until the end of the array is reached (or the end of the block). Partitioning PIM operations into blocks would necessitate a PIM aware compiler tool chain in order to intelligently place data structures within blocks.

The other alternative method of implementing indirect addressing is to use a PIM capable program memory. In the same example as before, the program memory would contain a PIM operation on the data memory, followed by a PIM operation to increment the instruction affecting data memory. The major trade off in this situation is additional hardware for the program memory, and in compiler complexity. Self-modifying code greatly increases the complexity of a program, and strictly defined rules for how program memory PIM operations could be used would be required. Extra hardware could be kept to a minimum by designating a section of the program memory as PIM; a block of program memory addresses would be enhanced with PIM hardware, while the rest of the program memory is constructed as normal. All PIM operations would need to be placed within the PIM section.

The challenges of supporting indirect addressing make another inherent challenge for PIM architectures apparent. In memory operations increase performance, but addressing

multiple locations with a single instruction introduces overhead. Performing an in memory operation requires providing as many addresses as there are operands. If calculating those addresses takes as long as loading the values instead, there is no point to PIM operations. When designing a PIM system, one must make trade offs in the address space accessible, the time spent creating addresses, and the hardware reserved for PIM operations.

5.2 Future Work

The extra FPGA resource utilization of the PIM enhanced ATmega103 was insignificant compared to the baseline. For this reason it is a strategy worth considering for future FPGA based systems. An interesting topic for future research would be to compare those results in an ASIC. By creating a VLSI implementation of the design, with the memory hardware details defined explicitly rather than using block RAM, a better estimate of both hardware area and power consumption can be made.

The other great challenge for any new computer architecture is to support the architecture with programming tools. In order to be used in a practical system, PIM operations must be supported by a compiler. Development of the PIM AES program was slow because of the challenges of interacting with an optimizing compiler not aware of the full ISA. Any memory location touched by a PIM operation is marked volatile to let the compiler know memory location values need to be reloaded every time because the values might have changed. Inline assembly PIM operations were added by compiling a program, checking the memory map file, and adding the PIM operations using the locations from the map, and re-compiling with the PIM operations. Beyond just being inconvenient, this process makes optimization more difficult, as it adds constraints to the compiler and programmer. Future work would involve adding the PIM instructions to the assembler and compiler, and updating the linker and loader to be aware of PIM and plan the memory mapping to facilitate PIM operations.

Appendix A C Source Code

A.1 AES.c

This section includes the source code for AES encryption using the default ATmega103.

```
#include "AES.h"
#include <avr/pgmspace.h>
#include <stdint.h>
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "softcore_util.h"



extern unsigned char RAM_file[FILE_SIZE];

// Allocate variables on stack so address is assigned at compile time,
    ↪ appears in map file
static const uint8_t Rcon[11] = { 0x8b, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
    ↪ 0x1b, 0x36 };
//initialize vector
uint8_t iv [N_BLOCK] = {0x69, 0x6e, 0x69, 0x74, 0x69, 0x61, 0x6c, 0x69, 0x7a, 0x65,
    ↪ 0x76, 0x65, 0x63, 0x74, 0x6f, 0x72};
uint8_t cipher [16];
//this is my key!!
uint8_t key[] = { 0x74, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x6d, 0x79, 0x20,
    ↪ 0x6b, 0x65, 0x79, 0x21, 0x21 };
uint8_t key_sched[KEY_SCHED_BYTES];
```

```
//shh its a secret
uint8_t plaintext[] = { 0x73, 0x68, 0x68, 0x20, 0x69, 0x74, 0x73, 0x20, 0x61, 0x20,
    ↪ 0x73, 0x65, 0x63, 0x72, 0x65, 0x74 };

volatile uint8_t s1[N_BLOCK];
volatile uint8_t s2[N_BLOCK];



int main( void )
{
    int blocks = 1; // 1 block for every 16 bytes in plaintext

    //0x3EC5 total 16,069
    set_key(key, key_sched); //0x19d2 6610
    PORTA = 0x03;
    //0x24f5 9461
    cbc_encrypt(plaintext, cipher, blocks, iv, key_sched);
    PORTA = 0x02;
    Interrupt_Loop(0x060);
    return 0;
}

uint8_t s_box (uint8_t x)
{
  return (uint8_t) RAM_file[x];
}



uint8_t set_key(uint8_t key[], uint8_t key_sched[])
```

```
{
    uint8_t hi = HI; //(ROUND + 1) << 4
    int keylen = KEYLEN; //16 bytes long
    copy_n_bytes (key_sched, key, keylen) ;
    uint8_t t[4] ;
    uint8_t next = keylen ;
    for (uint8_t cc = keylen, rc = 1 ; cc < hi ; cc += N_COL)
    {
      //Iterate through key_sched 4 bytes at a time
      //starting with 12, 13, 14, 15, ending with
      //168, 169, 170, 171
      for (uint8_t i = 0 ; i < N_COL ; i++)
        t[i] = key_sched [cc-4+i] ;
      if (cc == next)
        {
          rc = Rcon[cc >> 4]; //Same as division by 16, goes 1 to 10
          next += keylen ;
          uint8_t ttt = t[0] ; //cannot be precomputed
          t[0] = s_box (t[1]) ^ rc ;
          t[1] = s_box (t[2]) ;
          t[2] = s_box (t[3]) ;
          t[3] = s_box (ttt) ;
        }

      uint8_t tt = cc - keylen ;
      for (uint8_t i = 0 ; i < N_COL ; i++)
        key_sched [cc + i] = key_sched [tt + i] ^ t[i] ;
    }
    return (uint8_t) SUCCESS;
}
```

```
uint8_t cbc_encrypt (uint8_t * plain, uint8_t * cipher, int n_block, uint8_t iv
    ↪ [N_BLOCK], uint8_t * key_sched)
{
  while (n_block--)
    {
      xor_block (iv, plain) ;
      if (encrypt (iv, iv, key_sched, ROUND) != SUCCESS)
        return FAILURE ;
      copy_n_bytes (cipher, iv, N_BLOCK) ; // copy iv into cipher?
      plain  += N_BLOCK ;
      cipher += N_BLOCK ;
    }
  return SUCCESS ;
}

uint8_t encrypt (uint8_t plain [N_BLOCK], uint8_t cipher [N_BLOCK], uint8_t
    ↪ key_sched[KEY_SCHED_BYTES], int round)
{
  if (round)
    {
      uint8_t r;
      copy_and_key ((uint8_t*)s1, plain, (uint8_t*) (key_sched)) ;

      for (r = 1 ; r < round ; r++)
        {
          mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1) ;
          copy_and_key ((uint8_t*)s1, (uint8_t*)s2, (uint8_t*) (key_sched + r *
    ↪ N_BLOCK)) ;
```

```
        }
      shift_sub_rows ((uint8_t*)s1) ;
      copy_and_key (cipher, (uint8_t*)s1, (uint8_t*) (key_sched + r * N_BLOCK)) ;
      return SUCCESS;
    }
  else
    return FAILURE;
}

void copy_n_bytes (uint8_t * d, uint8_t * s, uint8_t nn)
{
  while (nn >= 4)
    {
      *d++ = *s++ ;  // some unrolling
      *d++ = *s++ ;
      *d++ = *s++ ;
      *d++ = *s++ ;
      nn -= 4 ;
    }
  while (nn--)
    *d++ = *s++ ;
}

void copy_and_key (uint8_t * d, uint8_t * s, uint8_t * k)
{
  for (uint8_t i = 0 ; i < N_BLOCK ; i ++)
    {
      *d++ = *s++ ^ *k++ ;
    }
}
```

```
void mix_sub_columns (uint8_t dt[N_BLOCK], uint8_t st[N_BLOCK])
{
  uint8_t j = 5 ;
  uint8_t k = 10 ;
  uint8_t l = 15 ;
  for (uint8_t i = 0 ; i < N_BLOCK ; i += N_COL)
    {
      uint8_t a = st [i] ;
      uint8_t b = st [j] ;  j = (j+N_COL) & 15 ;
      uint8_t c = st [k] ;  k = (k+N_COL) & 15 ;
      uint8_t d = st [l] ;  l = (l+N_COL) & 15 ;
      uint8_t a1 = s_box(a);
      uint8_t b1 = s_box (b);
      uint8_t c1 = s_box (c);
      uint8_t d1 = s_box (d);
      uint8_t a2 = f2(a1);
      uint8_t b2 = f2(b1);
      uint8_t c2 = f2(c1);
      uint8_t d2 = f2(d1);
      dt[i]   = a2     ^  b2^b1  ^  c1     ^  d1 ;
      dt[i+1] = a1     ^  b2     ^  c2^c1  ^  d1 ;
      dt[i+2] = a1     ^  b1     ^  c2     ^  d2^d1 ;
      dt[i+3] = a2^a1  ^  b1     ^  c1     ^  d2 ;
    }
}

void shift_sub_rows (uint8_t st [N_BLOCK])
{
  st [0] = s_box (st [0]) ; st [4]  = s_box (st [4]) ;
```

```
  st [8] = s_box (st [8]) ; st [12] = s_box (st [12]) ;

  uint8_t tt = st [1] ;
  st [1] = s_box (st [5]) ;  st [5]  = s_box (st [9]) ;
  st [9] = s_box (st [13]) ; st [13] = s_box (tt) ;

  tt = st[2] ; st [2] = s_box (st [10]) ; st [10] = s_box (tt) ;
  tt = st[6] ; st [6] = s_box (st [14]) ; st [14] = s_box (tt) ;

  tt = st[15] ;
  st [15] = s_box (st [11]) ; st [11] = s_box (st [7]) ;
  st [7]  = s_box (st [3]) ;  st [3]  = s_box (tt) ;
}

void xor_block (uint8_t * d, uint8_t * s)
{
  for (uint8_t i = 0 ; i < N_BLOCK ; i++)
    {
      *d++ ^= *s++ ;
    }
}
```

A.2 PIM_AES.c

This section includes the source code for AES encryption using PIM operations.

```
#include "AES.h"
#include <avr/pgmspace.h>
#include <stdint.h>
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <util/delay_basic.h>
#include "softcore_util.h"

// Pre computed in memory s-box
extern unsigned char RAM_file[FILE_SIZE];

// Allocate variables on heap so address is assigned at compile time, appears
    ↪ in map file
static const uint8_t Rcon[11] = { 0x8b, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
    ↪ 0x1b, 0x36 };

// WARNING: THIS CODE IS NOT PORTABLE
// Special processor in memory instructions in this file make certain
    ↪ assumptions about
// memory locations of global variables. In order for this to work, all of
    ↪ the following
// global variables must be allocated at the location commented

//initial vector RAM address 0x060
//"initializevector"
volatile uint8_t iv [N_BLOCK] = {0x69, 0x6e, 0x69, 0x74, 0x69, 0x61, 0x6c, 0x69, 0x7a,
    ↪ 0x65, 0x76, 0x65, 0x63, 0x74, 0x6f, 0x72};

//cipher to be computed RAM address 0x15c
volatile uint8_t cipher [N_BLOCK];

//symmetric key RAM address 0x070
//"this is my key!!"
```

```
uint8_t key[] = { 0x74, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x6d, 0x79, 0x20,
    ↪ 0x6b, 0x65, 0x79, 0x21, 0x21 };

//temporary state variable RAM address 0x09c
volatile uint8_t s1[N_BLOCK];
//temporary state variable RAM address 0x16c
volatile uint8_t s2[N_BLOCK];

//key computed for all 10 rounds RAM address 0x0ac
volatile uint8_t key_sched[KEY_SCHED_BYTES];

//plaintext to hide RAM address 0x080
//"shh its a secret"
uint8_t plaintext[] = { 0x73, 0x68, 0x68, 0x20, 0x69, 0x74, 0x73, 0x20, 0x61, 0x20,
    ↪ 0x73, 0x65, 0x63, 0x72, 0x65, 0x74 };



int main( void )
{
    int blocks = 1; // 1 block for every 16 bytes in plaintext

    //3082 total 12,418
    PORTA = 0x03;
    set_key(key, (uint8_t *)key_sched); //19db 6,619
    PORTA = 0x02; //16a5 5797
    cbc_encrypt_PIM(plaintext, cipher, blocks, iv, (uint8_t*)key_sched);
    //PORTA = 0x02;
    Interrupt_Loop(0x0060);
    return 0;
}
```

```
// Function that combines the first three steps used in rounds
// 1-9 of AES
void mix_sub_columns (uint8_t dt[N_BLOCK], uint8_t st[N_BLOCK])
{
  uint8_t j = 5 ;
  uint8_t k = 10 ;
  uint8_t l = 15 ;
  for (uint8_t i = 0 ; i < N_BLOCK ; i += N_COL)
    {
      uint8_t a = st [i] ;
      uint8_t b = st [j] ;  j = (j+N_COL) & 15 ;
      uint8_t c = st [k] ;  k = (k+N_COL) & 15 ;
      uint8_t d = st [l] ;  l = (l+N_COL) & 15 ;
      uint8_t a1 = s_box(a);
      uint8_t b1 = s_box (b);
      uint8_t c1 = s_box (c);
      uint8_t d1 = s_box (d);
      uint8_t a2 = f2(a1);
      uint8_t b2 = f2(b1);
      uint8_t c2 = f2(c1);
      uint8_t d2 = f2(d1);

      dt[i]   = a2     ^  b2^b1  ^  c1     ^  d1 ;
      dt[i+1] = a1     ^  b2     ^  c2^c1  ^  d1 ;
      dt[i+2] = a1     ^  b1     ^  c2     ^  d2^d1 ;
      dt[i+3] = a2^a1  ^  b1     ^  c1     ^  d2 ;
    }
}
```

```
// Looks up s-box value from table in RAM
uint8_t s_box (uint8_t x)
{
  return (uint8_t) RAM_file[x];
}


// Calculate the keys for all ten rounds and store in an array
uint8_t set_key(uint8_t key[], uint8_t key_sched[])
{
    uint8_t hi = HI; //(ROUND + 1) << 4
    int keylen = KEYLEN; //16 bytes long
    copy_n_bytes (key_sched, key, keylen) ;
    uint8_t t[4] ;
    uint8_t next = keylen;
    for (uint8_t cc = keylen, rc = 1 ; cc < hi ; cc += N_COL)
    {
      //Iterate through key_sched 4 bytes at a time
      //starting with 12, 13, 14, 15, ending with
      //168, 169, 170, 171
      for (uint8_t i = 0 ; i < N_COL ; i++)
        t[i] = key_sched [cc-4+i] ;
      if (cc == next)
        {
          rc = Rcon[cc >> 4]; //Same as division by 16, goes 1 to 10
          next += keylen ;
          uint8_t ttt = t[0] ; //cannot be precomputed
          t[0] = s_box (t[1]) ^ rc ;
          t[1] = s_box (t[2]) ;
```

```
          t[2] = s_box (t[3]) ;
          t[3] = s_box (ttt) ;
        }

      uint8_t tt = cc - keylen ; //Starts at 0, goes to 152 increments
                                 // of 4
      for (uint8_t i = 0 ; i < N_COL ; i++)
        key_sched [cc + i] = key_sched [tt + i] ^ t[i] ;
    }
    return (uint8_t) SUCCESS;
}


// Encrypt in cypher block chaining mode
uint8_t cbc_encrypt_PIM (uint8_t * plain, volatile uint8_t * cipher, int n_block,
    ↪ volatile uint8_t * iv, uint8_t * key_sched)
{
  while (n_block--)
    {

    //xor_block ((uint8_t *)iv, plain); //takes c9 clocks (201)
    asm volatile("ldi r30, 0""\n\t":::"r30");
    asm volatile(".word 0x0003, 0x6080""\n\t"::);
    asm volatile(".word 0x0003, 0x6181""\n\t"::);
    asm volatile(".word 0x0003, 0x6282""\n\t"::);
    asm volatile(".word 0x0003, 0x6383""\n\t"::);
    asm volatile(".word 0x0003, 0x6484""\n\t"::);
    asm volatile(".word 0x0003, 0x6585""\n\t"::);
    asm volatile(".word 0x0003, 0x6686""\n\t"::);
    asm volatile(".word 0x0003, 0x6787""\n\t"::);
```

```
    asm volatile(".word 0x0003, 0x6888""\n\t"::);
    asm volatile(".word 0x0003, 0x6989""\n\t"::);
    asm volatile(".word 0x0003, 0x6A8A""\n\t"::);
    asm volatile(".word 0x0003, 0x6B8B""\n\t"::);
    asm volatile(".word 0x0003, 0x6C8C""\n\t"::);
    asm volatile(".word 0x0003, 0x6D8D""\n\t"::);
    asm volatile(".word 0x0003, 0x6E8E""\n\t"::);
    asm volatile(".word 0x0003, 0x6F8F""\n\t"::);



      // no need to copy back into cipher after like this
      if (encrypt ((uint8_t*)iv, (uint8_t*)cipher, key_sched, ROUND) != SUCCESS)
        return FAILURE ;
      //copy_n_bytes ((uint8_t *)cipher, (uint8_t*)iv, N_BLOCK) ; // copy iv
    ↪ into cipher?
      //plain  += N_BLOCK ;
      //cipher += N_BLOCK ;

    }
  return SUCCESS ;
}


// Perform actual AES on 16 byte block
uint8_t encrypt (uint8_t * iv, uint8_t* cipher, uint8_t key_sched[KEY_SCHED_BYTES], int
    ↪ round)
{
  if (round)
    {
```

```
  //iv starts at 0x060, key_sched at 0x0ac, s1 at 0x09c, cipher 0x15c
  //copy_and_key ((uint8_t*)s1, (uint8_t*)iv, (uint8_t*) (key_sched));

// Copy iv to s1
asm volatile("ldi r30, 0x00""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C60""\n\t"::);
asm volatile(".word 0x0002, 0x9D61""\n\t"::);
asm volatile(".word 0x0002, 0x9E62""\n\t"::);
asm volatile(".word 0x0002, 0x9F63""\n\t"::);
asm volatile(".word 0x0002, 0xA064""\n\t"::);
asm volatile(".word 0x0002, 0xA165""\n\t"::);
asm volatile(".word 0x0002, 0xA266""\n\t"::);
asm volatile(".word 0x0002, 0xA367""\n\t"::);
asm volatile(".word 0x0002, 0xA468""\n\t"::);
asm volatile(".word 0x0002, 0xA569""\n\t"::);
asm volatile(".word 0x0002, 0xA66A""\n\t"::);
asm volatile(".word 0x0002, 0xA76B""\n\t"::);
asm volatile(".word 0x0002, 0xA86C""\n\t"::);
asm volatile(".word 0x0002, 0xA96D""\n\t"::);
asm volatile(".word 0x0002, 0xAA6E""\n\t"::);
asm volatile(".word 0x0002, 0xAB6F""\n\t"::);


// s1 ^= key_sched
//asm volatile("ldi r30, 0x00""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9CAC""\n\t"::);
asm volatile(".word 0x0003, 0x9DAD""\n\t"::);
asm volatile(".word 0x0003, 0x9EAE""\n\t"::);
asm volatile(".word 0x0003, 0x9FAF""\n\t"::);
```

```
asm volatile(".word 0x0003, 0xA0B0""\n\t"::);
asm volatile(".word 0x0003, 0xA1B1""\n\t"::);
asm volatile(".word 0x0003, 0xA2B2""\n\t"::);
asm volatile(".word 0x0003, 0xA3B3""\n\t"::);
asm volatile(".word 0x0003, 0xA4B4""\n\t"::);
asm volatile(".word 0x0003, 0xA5B5""\n\t"::);
asm volatile(".word 0x0003, 0xA6B6""\n\t"::);
asm volatile(".word 0x0003, 0xA7B7""\n\t"::);
asm volatile(".word 0x0003, 0xA8B8""\n\t"::);
asm volatile(".word 0x0003, 0xA9B9""\n\t"::);
asm volatile(".word 0x0003, 0xAABA""\n\t"::);
asm volatile(".word 0x0003, 0xABBB""\n\t"::);



//ROUND 1
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);
//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
```

```
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);

//s1 ^= key_sched from 0x0bc
asm volatile("ldi r30, 0x00""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9CBC""\n\t"::);
asm volatile(".word 0x0003, 0x9DBD""\n\t"::);
asm volatile(".word 0x0003, 0x9EBE""\n\t"::);
asm volatile(".word 0x0003, 0x9FBF""\n\t"::);
asm volatile(".word 0x0003, 0xA0C0""\n\t"::);
asm volatile(".word 0x0003, 0xA1C1""\n\t"::);
asm volatile(".word 0x0003, 0xA2C2""\n\t"::);
asm volatile(".word 0x0003, 0xA3C3""\n\t"::);
asm volatile(".word 0x0003, 0xA4C4""\n\t"::);
asm volatile(".word 0x0003, 0xA5C5""\n\t"::);
asm volatile(".word 0x0003, 0xA6C6""\n\t"::);
asm volatile(".word 0x0003, 0xA7C7""\n\t"::);
asm volatile(".word 0x0003, 0xA8C8""\n\t"::);
asm volatile(".word 0x0003, 0xA9C9""\n\t"::);
asm volatile(".word 0x0003, 0xAACA""\n\t"::);
asm volatile(".word 0x0003, 0xABCB""\n\t"::);


//ROUND 2
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);
//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
```

```
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);

//s1 ^= key_sched from 0x0cc
asm volatile("ldi r30, 0x00""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9CCC""\n\t"::);
asm volatile(".word 0x0003, 0x9DCD""\n\t"::);
asm volatile(".word 0x0003, 0x9ECE""\n\t"::);
asm volatile(".word 0x0003, 0x9FCF""\n\t"::);
asm volatile(".word 0x0003, 0xA0D0""\n\t"::);
asm volatile(".word 0x0003, 0xA1D1""\n\t"::);
asm volatile(".word 0x0003, 0xA2D2""\n\t"::);
asm volatile(".word 0x0003, 0xA3D3""\n\t"::);
asm volatile(".word 0x0003, 0xA4D4""\n\t"::);
asm volatile(".word 0x0003, 0xA5D5""\n\t"::);
asm volatile(".word 0x0003, 0xA6D6""\n\t"::);
asm volatile(".word 0x0003, 0xA7D7""\n\t"::);
```

```
asm volatile(".word 0x0003, 0xA8D8""\n\t"::);
asm volatile(".word 0x0003, 0xA9D9""\n\t"::);
asm volatile(".word 0x0003, 0xAADA""\n\t"::);
asm volatile(".word 0x0003, 0xABDB""\n\t"::);


//ROUND 3
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);
//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);

//s1 ^= key_sched from 0x0dc
asm volatile("ldi r30, 0x00""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9CDC""\n\t"::);
```

```
asm volatile(".word 0x0003, 0x9DDD""\n\t"::);
asm volatile(".word 0x0003, 0x9EDE""\n\t"::);
asm volatile(".word 0x0003, 0x9FDF""\n\t"::);
asm volatile(".word 0x0003, 0xA0E0""\n\t"::);
asm volatile(".word 0x0003, 0xA1E1""\n\t"::);
asm volatile(".word 0x0003, 0xA2E2""\n\t"::);
asm volatile(".word 0x0003, 0xA3E3""\n\t"::);
asm volatile(".word 0x0003, 0xA4E4""\n\t"::);
asm volatile(".word 0x0003, 0xA5E5""\n\t"::);
asm volatile(".word 0x0003, 0xA6E6""\n\t"::);
asm volatile(".word 0x0003, 0xA7E7""\n\t"::);
asm volatile(".word 0x0003, 0xA8E8""\n\t"::);
asm volatile(".word 0x0003, 0xA9E9""\n\t"::);
asm volatile(".word 0x0003, 0xAAEA""\n\t"::);
asm volatile(".word 0x0003, 0xABEB""\n\t"::);



//ROUND 4
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);

//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
```

```
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);

//s1 ^= key_sched from 0x0ec
asm volatile("ldi r30, 0x00""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9CEC""\n\t"::);
asm volatile(".word 0x0003, 0x9DED""\n\t"::);
asm volatile(".word 0x0003, 0x9EEE""\n\t"::);//fine
asm volatile(".word 0x0003, 0x9FEF""\n\t"::);//fine
asm volatile(".word 0x0003, 0xA0F0""\n\t"::);//fine
asm volatile(".word 0x0003, 0xA1F1""\n\t"::);//fine
asm volatile(".word 0x0003, 0xA2F2""\n\t"::);//fine
asm volatile(".word 0x0003, 0xA3F3""\n\t"::);//fine
asm volatile(".word 0x0003, 0xA4F4""\n\t"::);//bad
asm volatile(".word 0x0003, 0xA5F5""\n\t"::);//bad
asm volatile(".word 0x0003, 0xA6F6""\n\t"::);
asm volatile(".word 0x0003, 0xA7F7""\n\t"::);
asm volatile(".word 0x0003, 0xA8F8""\n\t"::);
asm volatile(".word 0x0003, 0xA9F9""\n\t"::);
asm volatile(".word 0x0003, 0xAAFA""\n\t"::);
asm volatile(".word 0x0003, 0xABFB""\n\t"::);



//ROUND 5
```

```
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);
//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);

//s1 ^= key_sched from 0x0fc
asm volatile("ldi r30, 0x00""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9CFC""\n\t"::);
asm volatile(".word 0x0003, 0x9DFD""\n\t"::);
asm volatile(".word 0x0003, 0x9EFE""\n\t"::);
asm volatile(".word 0x0003, 0x9FFF""\n\t"::);
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0003, 0xA000""\n\t"::);
asm volatile(".word 0x0003, 0xA101""\n\t"::);
asm volatile(".word 0x0003, 0xA202""\n\t"::);
```

```
asm volatile(".word 0x0003, 0xA303""\n\t"::);
asm volatile(".word 0x0003, 0xA404""\n\t"::);
asm volatile(".word 0x0003, 0xA505""\n\t"::);
asm volatile(".word 0x0003, 0xA606""\n\t"::);
asm volatile(".word 0x0003, 0xA707""\n\t"::);
asm volatile(".word 0x0003, 0xA808""\n\t"::);
asm volatile(".word 0x0003, 0xA909""\n\t"::);
asm volatile(".word 0x0003, 0xAA0A""\n\t"::);
asm volatile(".word 0x0003, 0xAB0B""\n\t"::);



//ROUND 6
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);
//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
```

```
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);

//s1 ^= key_sched from 0x10c
//asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9C0C""\n\t"::);
asm volatile(".word 0x0003, 0x9D0D""\n\t"::);
asm volatile(".word 0x0003, 0x9E0E""\n\t"::);
asm volatile(".word 0x0003, 0x9F0F""\n\t"::);
asm volatile(".word 0x0003, 0xA010""\n\t"::);
asm volatile(".word 0x0003, 0xA111""\n\t"::);
asm volatile(".word 0x0003, 0xA212""\n\t"::);
asm volatile(".word 0x0003, 0xA313""\n\t"::);
asm volatile(".word 0x0003, 0xA414""\n\t"::);
asm volatile(".word 0x0003, 0xA515""\n\t"::);
asm volatile(".word 0x0003, 0xA616""\n\t"::);
asm volatile(".word 0x0003, 0xA717""\n\t"::);
asm volatile(".word 0x0003, 0xA818""\n\t"::);
asm volatile(".word 0x0003, 0xA919""\n\t"::);
asm volatile(".word 0x0003, 0xAA1A""\n\t"::);
asm volatile(".word 0x0003, 0xAB1B""\n\t"::);



//ROUND 7
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);
//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
```

```
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);

//s1 ^= key_sched from 0x11c
//asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9C1C""\n\t"::);
asm volatile(".word 0x0003, 0x9D1D""\n\t"::);
asm volatile(".word 0x0003, 0x9E1E""\n\t"::);
asm volatile(".word 0x0003, 0x9F1F""\n\t"::);
asm volatile(".word 0x0003, 0xA020""\n\t"::);
asm volatile(".word 0x0003, 0xA121""\n\t"::);
asm volatile(".word 0x0003, 0xA222""\n\t"::);
asm volatile(".word 0x0003, 0xA323""\n\t"::);
asm volatile(".word 0x0003, 0xA424""\n\t"::);
asm volatile(".word 0x0003, 0xA525""\n\t"::);
asm volatile(".word 0x0003, 0xA626""\n\t"::);
asm volatile(".word 0x0003, 0xA727""\n\t"::);
asm volatile(".word 0x0003, 0xA828""\n\t"::);
asm volatile(".word 0x0003, 0xA929""\n\t"::);
asm volatile(".word 0x0003, 0xAA2A""\n\t"::);
```

```
asm volatile(".word 0x0003, 0xAB2B""\n\t"::);


//ROUND 8
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);
//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);


//s1 ^= key_sched from 0x12c
//asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9C2C""\n\t"::);
asm volatile(".word 0x0003, 0x9D2D""\n\t"::);
asm volatile(".word 0x0003, 0x9E2E""\n\t"::);
asm volatile(".word 0x0003, 0x9F2F""\n\t"::);
asm volatile(".word 0x0003, 0xA030""\n\t"::);
```

```
asm volatile(".word 0x0003, 0xA131""\n\t"::);
asm volatile(".word 0x0003, 0xA232""\n\t"::);
asm volatile(".word 0x0003, 0xA333""\n\t"::);
asm volatile(".word 0x0003, 0xA434""\n\t"::);
asm volatile(".word 0x0003, 0xA535""\n\t"::);
asm volatile(".word 0x0003, 0xA636""\n\t"::);
asm volatile(".word 0x0003, 0xA737""\n\t"::);
asm volatile(".word 0x0003, 0xA838""\n\t"::);
asm volatile(".word 0x0003, 0xA939""\n\t"::);
asm volatile(".word 0x0003, 0xAA3A""\n\t"::);
asm volatile(".word 0x0003, 0xAB3B""\n\t"::);



//ROUND 9
mix_sub_columns ((uint8_t*)s2, (uint8_t*)s1);
//Copy s2 into s1
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0002, 0x9C6C""\n\t"::);
asm volatile(".word 0x0002, 0x9D6D""\n\t"::);
asm volatile(".word 0x0002, 0x9E6E""\n\t"::);
asm volatile(".word 0x0002, 0x9F6F""\n\t"::);
asm volatile(".word 0x0002, 0xA070""\n\t"::);
asm volatile(".word 0x0002, 0xA171""\n\t"::);
asm volatile(".word 0x0002, 0xA272""\n\t"::);
asm volatile(".word 0x0002, 0xA373""\n\t"::);
asm volatile(".word 0x0002, 0xA474""\n\t"::);
asm volatile(".word 0x0002, 0xA575""\n\t"::);
asm volatile(".word 0x0002, 0xA676""\n\t"::);
asm volatile(".word 0x0002, 0xA777""\n\t"::);
```

```
asm volatile(".word 0x0002, 0xA878""\n\t"::);
asm volatile(".word 0x0002, 0xA979""\n\t"::);
asm volatile(".word 0x0002, 0xAA7A""\n\t"::);
asm volatile(".word 0x0002, 0xAB7B""\n\t"::);

//s1 ^= key_sched from 0x13c
//asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0003, 0x9C3C""\n\t"::);
asm volatile(".word 0x0003, 0x9D3D""\n\t"::);
asm volatile(".word 0x0003, 0x9E3E""\n\t"::);
asm volatile(".word 0x0003, 0x9F3F""\n\t"::);
asm volatile(".word 0x0003, 0xA040""\n\t"::);
asm volatile(".word 0x0003, 0xA141""\n\t"::);//
asm volatile(".word 0x0003, 0xA242""\n\t"::);
asm volatile(".word 0x0003, 0xA343""\n\t"::);
asm volatile(".word 0x0003, 0xA444""\n\t"::);
asm volatile(".word 0x0003, 0xA545""\n\t"::);
asm volatile(".word 0x0003, 0xA646""\n\t"::);
asm volatile(".word 0x0003, 0xA747""\n\t"::);
asm volatile(".word 0x0003, 0xA848""\n\t"::);
asm volatile(".word 0x0003, 0xA949""\n\t"::);
asm volatile(".word 0x0003, 0xAA4A""\n\t"::);
asm volatile(".word 0x0003, 0xAB4B""\n\t"::);



  shift_sub_rows ((uint8_t*)s1) ;
  //copy_and_key ((uint8_t*)cipher, (uint8_t*)s1, (uint8_t*) (key_sched +
↪ 10 * N_BLOCK)) ;
```

```
//Copy s1 (0x9c) into cipher (0x15c)
asm volatile("ldi r30, 0x00""\n\t":::"r30");
asm volatile(".word 0x0012, 0x5C9C""\n\t"::);
asm volatile(".word 0x0012, 0x5D9D""\n\t"::);
asm volatile(".word 0x0012, 0x5E9E""\n\t"::);
asm volatile(".word 0x0012, 0x5F9F""\n\t"::);
asm volatile(".word 0x0012, 0x60A0""\n\t"::);
asm volatile(".word 0x0012, 0x61A1""\n\t"::);
asm volatile(".word 0x0012, 0x62A2""\n\t"::);
asm volatile(".word 0x0012, 0x63A3""\n\t"::);
asm volatile(".word 0x0012, 0x64A4""\n\t"::);
asm volatile(".word 0x0012, 0x65A5""\n\t"::);
asm volatile(".word 0x0012, 0x66A6""\n\t"::);
asm volatile(".word 0x0012, 0x67A7""\n\t"::);
asm volatile(".word 0x0012, 0x68A8""\n\t"::);
asm volatile(".word 0x0012, 0x69A9""\n\t"::);
asm volatile(".word 0x0012, 0x6AAA""\n\t"::);
asm volatile(".word 0x0012, 0x6BAB""\n\t"::);

//Cipher ^= key_sched from 0x14c
asm volatile("ldi r30, 0x01""\n\t":::"r30");
asm volatile(".word 0x0013, 0x5C4C""\n\t"::);
asm volatile(".word 0x0013, 0x5D4D""\n\t"::);
asm volatile(".word 0x0013, 0x5E4E""\n\t"::);
asm volatile(".word 0x0013, 0x5F4F""\n\t"::);
asm volatile(".word 0x0013, 0x6050""\n\t"::);
asm volatile(".word 0x0013, 0x6151""\n\t"::);
asm volatile(".word 0x0013, 0x6252""\n\t"::);
asm volatile(".word 0x0013, 0x6353""\n\t"::);
asm volatile(".word 0x0013, 0x6454""\n\t"::);
```

```
    asm volatile(".word 0x0013, 0x6555""\n\t"::);
    asm volatile(".word 0x0013, 0x6656""\n\t"::);
    asm volatile(".word 0x0013, 0x6757""\n\t"::);
    asm volatile(".word 0x0013, 0x6858""\n\t"::);
    asm volatile(".word 0x0013, 0x6959""\n\t"::);
    asm volatile(".word 0x0013, 0x6A5A""\n\t"::);
    asm volatile(".word 0x0013, 0x6B5B""\n\t"::);

      return SUCCESS;
    }
  else
    return FAILURE;
}



// Copy n bytes from location source to location destination
void copy_n_bytes (uint8_t * d, uint8_t * s, uint8_t nn)
{
  while (nn >= 4)
    {
      *d++ = *s++ ;  // some unrolling
      *d++ = *s++ ;
      *d++ = *s++ ;
      *d++ = *s++ ;
      nn -= 4 ;
    }
  while (nn--)
    *d++ = *s++ ;
}
```

```
// Copy 16 bytes from location source to location destination
// and xor source with key
void copy_and_key (uint8_t * d, uint8_t * s, uint8_t * k)
{
  for (uint8_t i = 0 ; i < N_BLOCK ; i++)
    {
      *d++ = *s++ ^ *k++ ;  // some unrolling
    }
}


// Shift rows stage of AES
void shift_sub_rows (uint8_t st [N_BLOCK])
{
  st [0] = s_box (st [0]) ; st [4]  = s_box (st [4]) ;
  st [8] = s_box (st [8]) ; st [12] = s_box (st [12]) ;

  uint8_t tt = st [1] ;
  st [1] = s_box (st [5]) ;  st [5]  = s_box (st [9]) ;
  st [9] = s_box (st [13]) ; st [13] = s_box (tt) ;

  tt = st[2] ; st [2] = s_box (st [10]) ; st [10] = s_box (tt) ;
  tt = st[6] ; st [6] = s_box (st [14]) ; st [14] = s_box (tt) ;

  tt = st[15] ;
  st [15] = s_box (st [11]) ; st [11] = s_box (st [7]) ;
  st [7]  = s_box (st [3]) ;  st [3]  = s_box (tt) ;
}
```

```
// XOR two 16 byte blocks
void xor_block (uint8_t * d, uint8_t * s)
{
  for (uint8_t i = 0 ; i < N_BLOCK ; i++)
    {
      *d++ ^= *s++ ;
    }
}
```

A.3 AES.h

This section includes the header file for AES source code.

```
#ifndef AES_H
#define AES_H
#define N_ROW 4
#define N_COL 4
#define N_BLOCK (N_ROW * N_COL)

#define KEYLEN 16 //key is 16 bytes long
#define ROUND 10  //128 bit encryption uses 10 rounds
#define HI ((ROUND + 1) << 4 )//I think evaluates to 176?
#define KEY_SCHED_BYTES ((ROUND + 1) * N_BLOCK)
#define SUCCESS (0)
#define FAILURE (-1)

#define WPOLY   0x011B
#define f2(x)   ((x) & 0x80 ? (x << 1) ^ WPOLY : x << 1);

uint8_t s_box(uint8_t);
uint8_t set_key(uint8_t [], uint8_t []);
```

```
void set_IV(uint8_t *iv, uint64_t ivc);
void get_IV(uint8_t *iv, uint64_t ivc);
void IV_inc(uint8_t *iv, uint64_t ivc);
uint8_t cbc_encrypt(uint8_t * plain, uint8_t * cipher, int n_block, uint8_t iv
    ↪ [N_BLOCK], uint8_t* key_sched);
uint8_t encrypt(uint8_t plain [N_BLOCK], uint8_t cipher [N_BLOCK], uint8_t
    ↪ key_sched[KEY_SCHED_BYTES], int round);
void copy_n_bytes(uint8_t * d, uint8_t * s, uint8_t nn);
void copy_and_key(uint8_t * d, uint8_t * s, uint8_t * k);
void mix_sub_columns(uint8_t dt[N_BLOCK], uint8_t st[N_BLOCK]);
void shift_sub_rows(uint8_t st [N_BLOCK]);
void xor_block(uint8_t * d, uint8_t * s);

#endif
```

A.4 softcore_util.c

This section includes utility functions used in AES code for interacting with the soft core.

```
#include "softcore_util.h"

#if ROM
// Program memory initialized variable
const unsigned char ROM_file[FILE_SIZE] PROGMEM =
{
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7,
    ↪ 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4,
    ↪ 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8,
    ↪ 0x31, 0x15,
```

```
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27,
    ↪ 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,
    ↪ 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c,
    ↪ 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,
    ↪ 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff,
    ↪ 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d,
    ↪ 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e,
    ↪ 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95,
    ↪ 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a,
    ↪ 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd,
    ↪ 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1,
    ↪ 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55,
    ↪ 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54,
    ↪ 0xbb, 0x16
};
// End program memory initialized variable

// Copy the program memory to SRAM
```

```
unsigned char RAM_file[FILE_SIZE];
#endif

unsigned char * volatile word_to_send;// volatile ptr to nonvolatile mem

// Functions for initializing RAM from ROM

void ROM_fill(void)
{
    memcpy_P(RAM_file, ROM_file, FILE_SIZE);
    USART_Init( MYUBRR );
    return;
}



// Functions for dumping RAM over UART




// UART data register empty interrupt service routine
ISR(UART_UDRE_vect) {

    if((int) word_to_send <= 4096)
    {
        USART_Transmit(*(word_to_send++)); // Transmit a word, increment pointer
        //Transmit_Address(word_to_send);
        PORTC |= 0x0F;
    }
    else
        PORTC = 0x01;
```

```
}


extern void Interrupt_Loop( uint8_t my_var )
{
    word_to_send = (unsigned char *) my_var;
    sei(); // Enable global interrupts
    while(1)
    {
        if(PINE) // Any of first 8 switches enabled, reset
        {
            cli();  // Disable interrupts
            word_to_send = (unsigned char *) my_var; // Reset pointer to first mem
    ↪ location
            //Transmit_Address(word_to_send);
            PORTC = 0x00; // Light LEDs, tell me where I am looping
        }
        else // Make sure interrupt is on
        {
            sei();
        }
    }
}


void USART_Init( unsigned int ubrr )
{
    /* Set baud rate */
    UBRR = (unsigned char)ubrr;
    /* Enable receiver and transmitter and data register empty interrupt */
    UCR = (1<<RXEN)|(1<<TXEN) | (1<<UDRIE); //ATmega103 defaults 1 start, 1 stop,
    ↪ 8 data
```

```
    DDRA = 0xFF; // Initialize Port A as output (used for cycle counting)
    PORTA = 0x00;// Reset when A(1) is low
    PORTA = 0x02;// Toggle reset of cycle counter
    DDRC = 0xFF; // Port C (LEDs) output
    DDRE = 0x00; // Initialize Port E (switches) as input
}

void USART_Transmit( unsigned char data )
{
    /* Wait for empty transmit buffer */
    while ( !( USR & (1<<UDRE)) );
    /* Put data into buffer, sends the data */
    UDR = data;
}

void Transmit_Address( unsigned char *pointer)
{
    // ASCII representation of address I'm printing next
    char address[4];
    // Integer to ASCII, tell me pointer mem location
    itoa( (int) pointer, address, 10);
    for(int i = 0; i < 4; i++) // Transmit address location string, CRLF
    {
        USART_Transmit(address[i]);
    }
    USART_Transmit('\r');
    USART_Transmit('\n');
}
```

A.5 softcore_util.h

This section includes the header file for utility functions.

```
#ifndef SOFTCORE_UTIL_H
#define SOFTCORE_UTIL_H

#define ROM 1
#define FILE_SIZE 0x100
#define OFFSET 0x003E // Start of pixel data in bmp file
#define BAUD 9600
#define FOSCFIRST 16000000
#if !defined(FOSC)
    #define FOSC 16000000 //Clock Speed
#endif
#define MYUBRR (FOSC/16/BAUD-1)

#if !defined(F_CPU)
    #define F_CPU 16000000UL //Clock Speed for delay.h
#endif

/* Data Direction Register, Port C */
#define DDRC    _SFR_IO8(0x14)
/* Data Direction Register, Port F */
#define DDRF    _SFR_IO8(0x08)

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <stdlib.h>
```

```
// Moves ROM file to RAM automagically by placing ROM_fill function
// in .init8 section of program memory before main is called

#if ROM
void ROM_fill(void) \
     __attribute__((naked)) \
     __attribute__((used))  \
     __attribute__((section(".init8")));
#else
void ROM_fill(void);
#endif




// RAM dump over UART
void USART_Init( unsigned int ubrr);
void USART_Transmit( unsigned char data );
void Transmit_Address( unsigned char *pointer);
extern inline void Interrupt_Loop(uint8_t) \
    __attribute__((always_inline));
#endif
```

References

[1] C. Carvalho, "The gap between processor and memory speeds," in *IEEE International Conference on Control and Automation*, Jan. 2002.

[2] D. Efnusheva, A. Cholakoska, and A. Tentov, "A survey of different approaches for overcoming the processor - memory bottleneck," *International Journal of Computer Science and Information Technology*, vol. 9, pp. 151–163, Apr. 2017.

[3] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis, "A combined DMA and application-specific prefetching approach for tackling the memory latency bottleneck," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 279–291, Mar. 2006.

[4] W. A. Wulf and S. A. McKee, "Hitting the memory wall," *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 20–24, Mar. 1995.

[5] K. K. Chang, *Understanding and improving the latency of DRAM-based memory systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2017.

[6] I. Pavlov, "7-zip LZMA benchmark." Online, 2018. Available at https://www.7-cpu.com/. Accessed 11/15/2018.

[7] J. Backus, "Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs," *Communications of the ACM*, vol. 21, pp. 613–641, Aug. 1978.

[8] J. Edwards and S. O'Keefe, "Eager recirculating memory to alleviate the von Neumann bottleneck," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, Dec. 2016.

[9] C.-H. Lu, C.-S. Lin, H.-L. Chao, J.-S. Shen, and P.-A. Hsiung, "Reconfigurable multi-core architecture – A plausible solution to the von Neumann performance bottleneck," in *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, IEEE, Sept. 2013.

[10] D. Elliott, W. Snelgrove, and M. Stumm, "Computational RAM: A memory-simd hybrid and its application to DSP," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, IEEE, 1992.

[11] L. Koskinen, J. Tissari, J. Teittinen, E. Lehtonen, M. Laiho, and J. H. Poikonen, "A performance case-study on memristive computing-in-memory versus von Neumann architecture," in *2016 Data Compression Conference (DCC)*, IEEE, Mar. 2016.

[12] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[13] Y. Zha and J. Li, "RRAM-based reconfigurable in-memory computing architecture with hybrid routing," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, Nov. 2017.

[14] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, New York: ACM Press, 2015.

[15] S. Srikanth, T. M. Conte, E. P. DeBenedictis, and J. Cook, "The superstrider architecture: Integrating logic and memory towards non-von Neumann computing," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, IEEE, Nov. 2017.

[16] B. Yu, M. Alawad, M. Riera, and M. Lin, "Improving memory performance in reconfigurable computing architecture through hardware-assisted dynamic graph," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, IEEE, Dec. 2013.

[17] X. Wen, F. Bensaali, and R. Sotudeh, "Dynamic co-operative intelligent memory," in *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*, IEEE, Jan. 2008.

[18] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, "Computing in memory with spin-transfer torque magnetic RAM," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–14, 2018.

[19] Y. Zhang, L. Xu, Q. Dong, J. Wang, D. Blaauw, and D. Sylvester, "Recryptor: A reconfigurable cryptographic Cortex-M0 processor with in-memory and near-memory computing for IoT security," *IEEE Journal of Solid-State Circuits*, vol. 53, pp. 995–1005, Apr. 2018.

[20] J. Hennessy and D. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, June 2018.

[21] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *2013 IEEE International 3D Systems Integration Conference (3DIC)*, IEEE, Oct. 2013.

[22] S. Hamdioui, L. Xie, A. N. H. Anh, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, "Memrisor based computation-in-memory architecture for data-intensive applications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, IEEE Conference Publications, 2015.

[23] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo," in *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, ACM Press, 2016.

[24] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 27–39, June 2016.

[25] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: the Terasys massively parallel PIM array," *Computer*, vol. 28, pp. 23–31, Apr. 1995.

[26] K. C. Akyel, H.-P. Charles, J. Mottin, B. Giraud, G. Suraci, S. Thuries, and J.-P. Noel, "DRC2: Dynamically reconfigurable computing circuit based on memory architecture," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, IEEE, Oct. 2016.

[27] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: IRAM," *Computer*, vol. 30, no. 9, pp. 75–78, 1997.

[28] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent RAM (IRAM): chips that remember and compute," in *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*, IEEE.

[29] H. A. D. Nguyen, L. Xie, M. Taouil, R. Nane, S. Hamdioui, and K. Bertels, "On the implementation of computation-in-memory parallel adder," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 2206–2219, Aug. 2017.

[30] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective (4th Edition)*. Pearson, 2010.

[31] B. Akin, F. Franchetti, and J. C. Hoe, "HAMLeT architecture for parallel data reorganization in memory," *IEEE Micro*, vol. 36, pp. 14–23, Jan. 2016.

[32] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick, "Intelligent RAM (IRAM): the industrial setting, applications, and architectures," in

Proceedings International Conference on Computer Design VLSI in Computers and Processors, IEEE Comput. Soc, 1997.

[33] C. Kozyrakis and D. Patterson, "A new direction for computer architecture research," *Computer*, vol. 31, no. 11, pp. 24–32, 1998.

[34] J. Gebis and S. Williams, "VIRAM1: A media oriented vector processor with embedded DRAM," in *41st Design Automation Student Design Contest*, 2004.

[35] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast bulk bitwise AND and OR in DRAM," *IEEE Computer Architecture Letters*, vol. 14, pp. 127–131, July 2015.

[36] V. Seshadri, M. A. Kozuch, T. C. Mowry, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, and P. B. Gibbons, "RowClone," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46*, ACM Press, 2013.

[37] P. Jain, G. E. Suh, and S. Devadas, "Embedded intelligent SRAM," in *Proceedings of the 40th conference on Design automation - DAC '03*, NACM Press, 2003.

[38] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, pp. 2054–2066, oct 2014.

[39] A. W. Topol, D. C. L. Tulipe, L. Shi, D. J. Frank, K. Bernstein, S. E. Steen, A. Kumar, G. U. Singco, A. M. Young, K. W. Guarini, and M. Ieong, "Three-dimensional integrated circuits," *IBM Journal of Research and Development*, vol. 50, pp. 491–506, July 2006.

[40] B. Akin, J. C. Hoe, and F. Franchetti, "HAMLeT: Hardware accelerated memory layout transform within 3D-stacked DRAM," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, Sep 2014.

[41] Atmel Corporation, *ATmega103(L) data sheet*, Feb. 2007.

[42] Atmel Corporation, *AVR instruction set manual*, 2016.

[43] R. Lepetenok, "AVR core." Online., Feb. 2017. Available at https://opencores.org/project/avrcore.

[44] J. Daemen and V. Rijmen, *The Design of Rijndael*. Berlin: Springer-Verlag, 2002.

[45] G. Spanos, "AES for microcontrollers." Online, Oct. 2018. Available at https://github.com/spaniakos/AES.