VIPERS II

A Soft-core Vector Processor with Single-copy Data Scratchpad Memory

by

Christopher Han-Yu Chou

B.A.Sc., The University of British Columbia, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2010

Abstract

Previous work has demonstrated soft-core vector processors in FPGAs can be applied to speed up data-parallel embedded applications, while providing the users an easy-to-use platform to tradeoff performance and area. However, its performance is limited by load and store latencies, requiring extra software design effort to optimize performance. This thesis presents VIPERS II, a new vector ISA and the corresponding microarchitecture, in which the vector processor reads and writes directly to a scratchpad memory instead of the vector register file. With this approach, the load and store operations and their inherent latencies can often be eliminated if the working set of data fits in the vector scratchpad memory. Moreover, with the removal of load/store latencies, the user doesn't have to use loop unrolling to enhance performance, reducing the amount of software effort required and making the vectorized code more compact. The thesis shows the new architecture has the potential to achieve performance similar to that of the unrolled versions of the benchmarks, without actually unrolling the loop. Hardware performance results of VIPERS II demonstrated up to 47× speedup over a Nios II processor with only 13× more resources used.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Guy Lemieux, for providing me directions and support in my research project. Also, thank you for the time you spent proofreading my thesis.

Thanks to Altera and NSERC for providing the funding that made this project possible, and to Altera for donating the DE3 development system.

I would also like to thank Aaron Severance for his work, which helped speed up the system level development of VIPERS II.

Finally, I would like to thank my family and my fiance Stella for their support over the years.

Glossary

MACL - Multiply Accumulate Chain Length, refers to the number of sum-accumulate units connected in a chain for sum-reduction operations.

MemWidth - Memory Width, refers to the width of the vector scratchpad memory in context of VIPERS II; refers to the width of main memory in context of VIPERS.

MemWidthByte - Memory width expressed in number of bytes.

MVL - Maximum Vector Length, refers to the maximum number of elements allowed in a data vector. In VIPERS, MVL is proportional to NLane. VIPERS II has no natural limit on MVL.

NLane - Number of vector lanes in the vector processor.

OpSize - Operand Size, refers to the size of the data elements in the vector. VIPERS II supports operand size of byte (8b), halfword (16b), and word (32b).

VL - Vector Length, refers to the number of elements in a data vector.

VPUW - Vector processing unit width, refers to the width of the processing unit in each vector lane.

Chapter 1

Introduction

The demand for high-performance embedded systems is driven by today's embedded applications, many of which involve heavy signal processing and multimedia algorithms. Moreover, the wide variety of products and requirements such as low cost and short time-to-market call for a rapid development platform. Embedded systems employ microprocessors, digital signal processors (DSPs), or field-programmable gate arrays (FPGAs) to perform intensive computations. However, hardware design expertise is required to implement embedded systems using FPGA platforms. This presents a barrier to productivity and cost, as hardware design is usually slower and require engineers with specialized skills.

Modern FPGAs offer soft processor cores, such as the Nios II [2] by Altera and the MicroBlaze by Xilinx, which can be incorporated into the embedded system to help simplify the design process. These soft-core processors, however, are not ideal for the heavy computational workload required by signal processing-rich embedded applications. Signal processing and multimedia algorithms typically involve only a few operations performed over a large amount of data. The soft-core processors achieve this by iteration, which often fails to deliver the desired performance. Many solutions have been proposed to improve soft processor performance, but most of them require the user to have parallel programming and hardware design skills.

Recent works on VIPERS [29–31] and VESPA [26–28], proposed to utilize vector processing to enhance the performance of soft-core processors in FPGA-based embedded systems. The idea of vector processing is to operate on vectors, a collection of data elements, by using multiple parallel datapaths. This can be applied to exploit the data-level parallelism

that exists in signal processing and multimedia algorithms.

1.1 Motivation

VESPA, Vector-Extended Soft Processor Architecture, provides an average speedup over a single lane vector processor from 1.8× up to 6.3×, when scaling the design from 2 to 16 lanes. It was also demonstrated in [26], when comparing against custom accelerators, VESPA can reduce the performance gap of 432× between scalar soft processor and custom hardware down to 17× with only software development effort.

VIPERS, Vector ISA Processors for Embedded Reconfigurable Systems, demonstrated how vector processing can help enhance the performance of soft processors on FPGAs. With the 16 vector lane instance of the design, the VIPERS achieved a speedup of 25× over the Nios II processor with 14× more resources. Not only does VIPERS provide better performance, it also offers many advantages over other performance enhancement solutions. The highly parameterized Verilog source code used to implement VIPERS provides the benefit of soft processor configurability. This allows the user to accelerate an application without any hardware design knowledge; all the user needs is to set the desired configuration and develop a vectorized version of the software. Moreover, VIPERS, a general-purpose architecture, can be used for a wide range of applications. However, there is also the option for application-specific customization to reduce the amount of resource usage.

Despite the many advantages VIPERS offers, there are still areas in which it could be improved. A major performance bottleneck of the VIPERS architecture is the high load and store latencies that occurs when copying data from memory to the register file. The speedup improvement shown by loop-unrolled versions over the original rolled up codes [30] prove that the load and store latencies are big factors in performance, even when the vector operations and vector memory accesses execute concurrently. On top of that, although loop unrolling improves performance, it also generates much larger code and takes up more

instruction memory space.

Another imperfection is the data duplication in memory. The vector register file data in VIPERS is duplicated to allow for simultaneous reads of two vectors as input operands. This traditional way of providing dual read ports is adequate for scalar processors due to their small register files. However, for a vector processor, the register file is much larger, making the duplication extremely costly in terms of using precious on-chip memory. Furthermore, if the main memory or a data cache is implemented on the FPGA as well, each vector will then reside in three different locations in the on-chip memory blocks, which is very inefficient usage of the limited on-chip memory resources.

Finally, the scalar core of the soft vector processor was implemented with the UTIIe [13], an open source processor, and it was modified to work with the vector processor. Unfortunately, the UTIIe was pipelined for multithreaded use and does not contain forwarding multiplexers needed to operate as a single thread processor. Therefore, it takes four cycles to execute each scalar instruction, which degrades the performance of the entire system. More significantly, unlike the Nios II, the UTIIe is not equipped with a debug-core, making validation of the VIPERS hardware system much more difficult.

The work in this thesis focuses on reducing, if not eliminating, the effects of these shortcomings on the performance and resource usage of VIPERS, while keeping all of its positive features.

1.2 Contribution

The main contribution of this research is the development of VIPERS II, the second generation of configurable soft-core vector processor. The VIPERS II architecture contains modifications intended to overcome several shortcomings of the original VIPERS.

As mentioned in the previous section, load and store latencies when copying data from memory to register file can limit performance in the rolled-up loops with the original

VIPERS. Since modern FPGAs offer integrated memory blocks with high operating frequencies, the VIPERS II architecture takes advantage of the fast on-chip memory and reads vector data directly from a scratchpad memory, whose content is loaded via a DMA engine. This removes the vector data register files and eliminates the need for the slow load and store operations when the working set fits inside the scratchpad buffer. The removal of the register file also resolves the data duplication issue and helps VIPERS II to achieve more efficient memory usage. Moreover, this also allows the size of the scratchpad buffer to be scaled up very easily by moving the design to a FPGA device with higher capacity.

The VIPERS II architecture accesses the scratchpad memory through address registers, which point to the starting address of each data vector. The architecture offers 32 address registers with post-increment, pre-decrement, and circular buffer features. These features modify the address register contents in preparation for subsequent instructions, thus lowering the loop overhead required to setup the address registers. This allows rolled-up code to achieve the performance of unrolled code without the instruction bloat or creating pressure on the number of vector data registers needed.

Along with the direct accessing of vector scratchpad memory, fracturable ALUs are introduced into the vector datapath to execute vector instructions with varying operand size. This provides more computational power for smaller operand sizes.

Performance of VIPERS II is maximized when vector data are aligned in the scratchpad memory. In cases where the vectors are not aligned, they are first moved to aligned locations in the scratchpad memory prior to execution. This is accomplished by a single data alignment crossbar network, which replaces the function provided by two separate read and write crossbars in the original VIPERS. The single crossbar saves area compared to two separate crossbars.

Outside of the scope of this thesis, two other major modifications were done by a fellow student, Aaron Severance, to complete the VIPERS II system. One is to use the VIPERS II as an extended custom instruction of Nios II. This allows an actual Nios II to be used

as the scalar core for the soft vector processor, providing a fully pipelined scalar unit with debug capability. The other is the development of a direct memory access (DMA) engine in charge of populating the vector scratchpad memory from off-chip DDR2 memory.

1.3 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 provides an overview of vector processing and presents the original VIPERS architecture. Chapter 3 describes in detail the new VIPERS II architecture, focusing on the major changes made, the advantages of these changes and how they are implemented. Chapter 4 provides experimental results to demonstrate the impact of the individual changes and discusses the overall resource usage of VIPERS II. Chapter 5 describes the benchmark preparation and presents the simulated and hardware performance results. Chapter 6 summarizes the work in this thesis and provides suggestions for future work.

Chapter 2

Background

This chapter provides some background information relevant to the work in this thesis. The chapter starts by giving an overview of vector processing and how it helps to accelerate data-parallel computations. Next, an overview of the original VIPERS architecture [31] and its implementation is presented. Finally, previous work in on-chip scratchpad memory and address registers are examined.

2.1 Vector Processing Overview

Vector processing has been applied in supercomputers for computing scientific workloads since the 1970s. The key advantage of vector processing over traditional computing methods is its ability to exploit the data-level parallelism readily available to scientific and engineering applications.

The basis of vector processing, as indicated by the name, is to operate on vectors of data. Each vector instruction defines a single operation to be executed on all data elements of the source vectors. Being able to perform a single operation on multiple data elements makes vector processing the ideal candidate for exploiting data-level parallelism, which has the same properties. The inherently parallel vector operations can be realized in vector processors by having multiple datapaths, called vector lanes, that execute in parallel.

A typical vector architecture contains a vector processing unit and a scalar core. The scalar core is required to execute the control flow instructions, such as loops, and the non-vectorizable code. The vector processing unit, containing the vector lanes, performs the

data processing operations by executing each instruction over all vector elements.

Each vector instruction can specify tens of operations and produce tens of results at once. Vector instructions are governed by the vector length register, VL, which specifies the number of elements to operate on. The VL register can be modified at run time between vector instructions. However, most architectures are also subject to a maximum vector length, or MVL, usually in the range of 32 to 256 elements. When software requires even larger vectors, the data must be broken up into sizes the hardware can handle.

To demonstrate how vector processing exploits data parallelism, let's look at the example of an 8-tap finite impulse response (FIR) filter, which is described by:

$$y[n] = \sum_{k=0}^{7} x[n-k]h[k].$$

This can be implemented in a scalar processor using two nested loops: the inner loop performs eight multiply-add operations using the input data and filter coefficients, and the outer loop iterates over the entire input data set. With the Nios II assembly, 65 instructions are required to produce one output datum.

The same FIR filter can be implemented with the VIPERS instruction set as shown in Figure 2.1. The inner loop of the scalar code is replaced by the VMAC/VCCZACC instruction pair, which computes a single result by multiplying a vector all filter coefficients with a data sample vector of length eight. The vectorized FIR filter requires only 10 instructions per result, and performs better than the scalar version.

2.2 VIPERS Architecture

This section provides details on the architecture and the implementation of the original VIPERS soft-core vector processor [29–31]. The VIPERS soft vector architecture defines a family of soft vector processors with variable performance and resource utilization, and a set of configurable features to suit different applications. The users can configure the

```
        movi    r12, N                  ; Total samples to process
        movi    r6,  NTAPS              ; Number of filter taps
        vld     v2, vbase1              ; Load filter coefficients once
.L4:
        movi    r11, 2*NTAPS            ; Window size of samples to load
        vmstc   VL, r11                 ; Set VL to window size
        vld     v1, vbase0, vinc0       ; Load x[] data vector
.L5:
        vmstc           VL, r6          ; Reset VL to number of taps
        VMAC            v1, v2          ; Multiply-accumulate VL values
        VCCZACC         v3              ; Copy result from accumulator and zero
        vmstc           vindex, zero    ; Set vindex to 0
        vext.vs         r3, v3          ; Extract sum result from element 'vindex'
        stw             r3, 0(r10)      ; Store filtered result
        VUPSHIFT        v1, v1          ; Vector element shift by 1 position
        addi            r10, r10, 4     ; Increment y[] buffer position
        addi            r11, r11, -1
        bne             r11, r6, .L5

        sub     r12, r12, r6            ; Repeat for next NTAPS samples
        bne     r12, zero, .L4
```

Figure 2.1: VIPERS Vector Assembly code for 8-tap FIR Filter [30]

highly parameterized Verilog source code and generate an application-specific instance of the processor.

The VIPERS instruction set architecture (ISA) borrows heavily from the VIRAM [16] instruction set, except a few extensions are made to make use of FPGA resources to accelerate certain applications. Table 2.1 lists the additional instructions that are included in the VIPERS ISA.

2.2.1 Architecture Overview

Figure 2.2 gives a high-level view of the VIPERS architecture, consisting of a scalar core, a vector core, and a memory interface unit. The scalar core is implemented with the UTIIe [13], a 32-bit soft processor compatible with the Nios II instruction set. The VIPERS instructions are 32-bit and can be mixed with scalar instructions in the instruction stream.

Instruction	Description	Application
VMAC	Multiply-accumulate	FIR, motion estimation
VCCZACC	Compress copy from accumulator and zero	FIR, motion estimation
VUPSHIFT	Vector element up-shift	FIR
VABSDIFF	Vector absolute difference	motion estimation
VLDL	Load from local memory	AES
VSTL	Store to local memory	AES

Table 2.1: VIPERS Specific Instructions (Source: [29])

The different scalar and vector instructions can be executed concurrently. For operations that require both processors, the FIFO queues are used to transfer and synchronize data between the two cores.

The pretested UTIIe was chosen to speed up the development of VIPERS. Also, since its Verilog source is available, it was modified to allow close coupling between the scalar and vector cores, which share the same instruction memory. However, the multithreaded design of the UTIIe has no hazard detection or forwarding, so it can only issue a new instruction after the previous one has completed, causing it to take 4 cycles per scalar instruction. Moreover, the lack of a JTAG debug core made it hard to debug the system in hardware.

Traditional vector processors tend not to have a large number of vector lanes; therefore, high performance is achieved through pipelining and instruction chaining. Figure 2.3(a) illustrates instruction chaining, which refers to the passing of partial results between functional units for data-dependent instructions. This adds to the size and complexity of the vector register file as multiple read and write ports cannot be implemented efficiently on FPGAs.

The VIPERS architecture takes advantage of the programmable fabric on FPGAs to provide a large number of vector lanes, allowing the vector instructions to be executed across the vector lanes in SIMD fashion, and pipelined over several cycles in traditional vector fashion. Figure 2.3(b) illustrates this hybrid vector-SIMD execution model. The number of clock cycles required to execute a vector instruction is the vector length divided

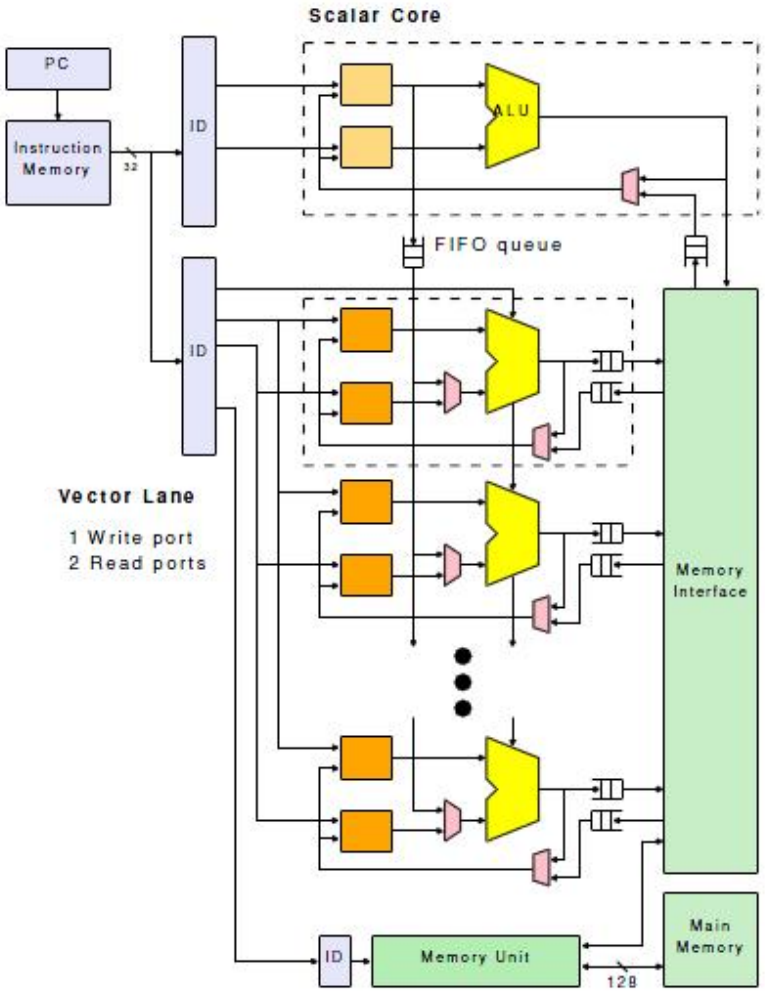


Figure 2.2: VIPERS Architecture (Source: [29])

by number of vector lanes. With a large number of vector lanes, the number of cycles required to execute a vector operation is decreased. As a result, instruction chaining can be removed, simplifying the design of the vector register file.

2.2.2 Vector Lane Datapath

Figure 2.4 illustrates the vector lane datapath of the vector core. The vector unit consists of a configurable number of vector lanes, as defined by the NLane parameter; each lane is

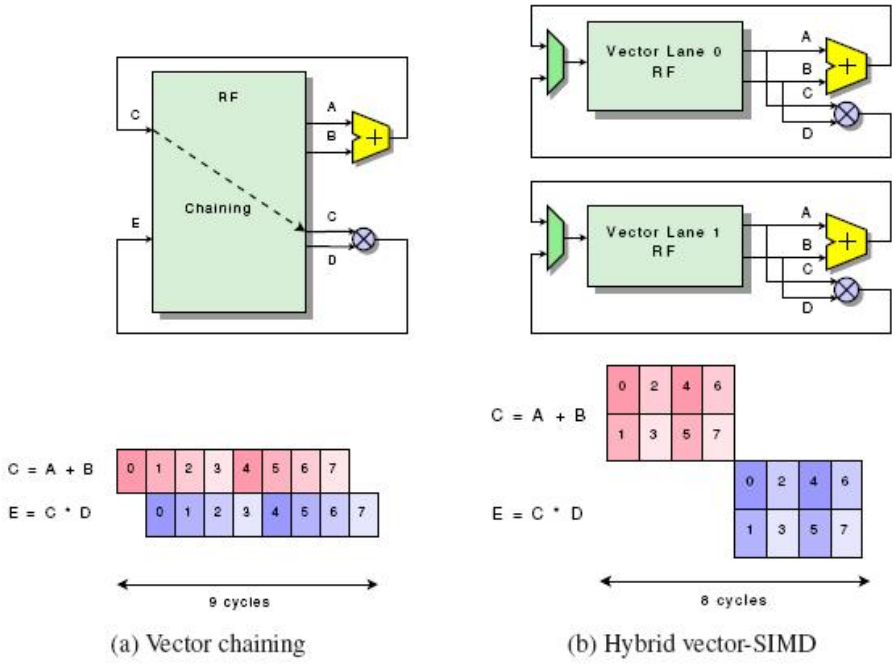


Figure 2.3: Chaining and Hybrid Vector-SIMD Model (Source: [29])

completed with a full set of functional units, a partitioned vector register file, vector flag registers, a load-store unit, and an optional local memory. NLane is the primary parameter in scaling the vector processor's performance and area. A vector processor instance of more lanes can process the same vector in fewer clock cycles, but will require more resources to implement.

The vector core has a 4 stage pipeline plus the shared instruction fetch stage in the scalar core. As shown in Figure 2.2, the vector unit has two instruction decoders (ID), one to decode all vector instructions, and another to decode the vector memory instructions. The extra decoder allows for concurrent execution of vector computation and vector memory access to lower the effects of the slower load and store operations. Read after write (RAW) hazards are resolved by pipeline interlocking; the decode stage stalls the later instructions if it detects data dependency between the instructions.

The set of functional units inside each vector lane includes an ALU, a barrel shifter,

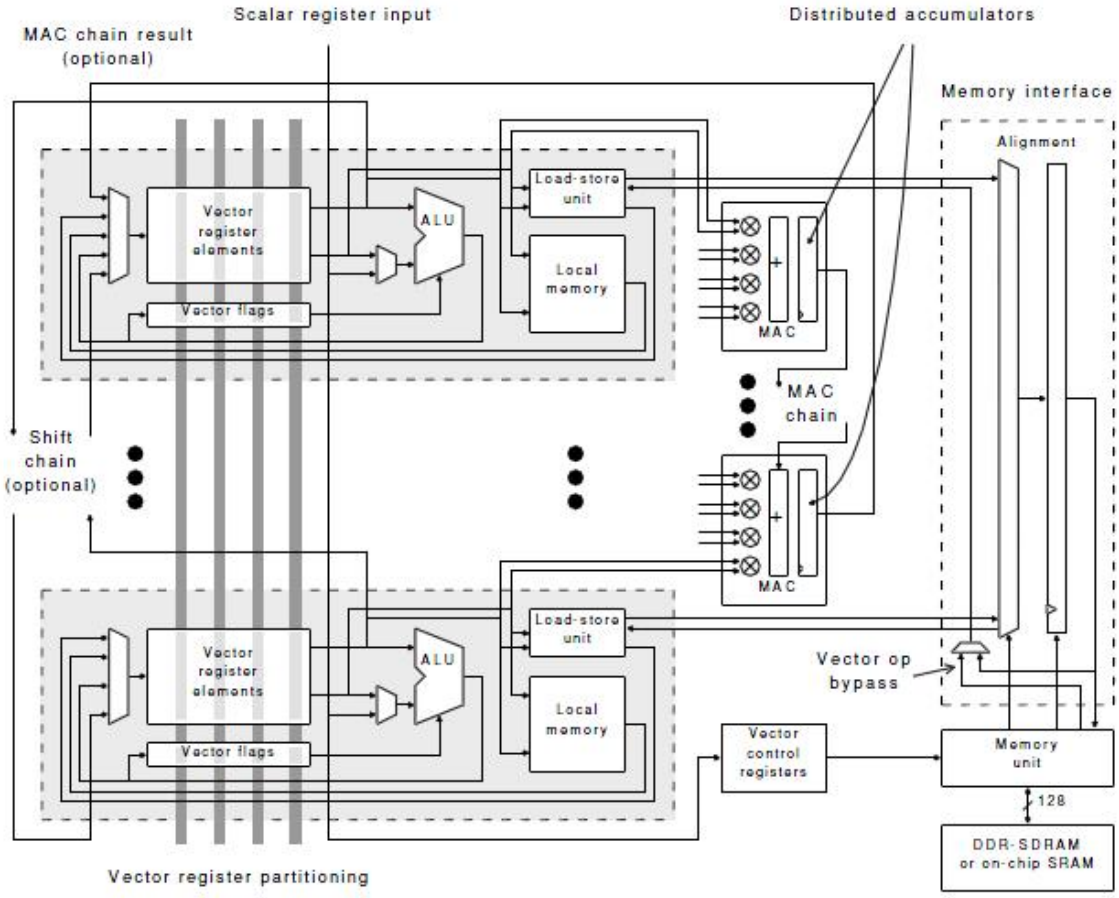


Figure 2.4: VIPERS Vector Lane Datapath (Source: [29])

and a optional hardware multiplier. The ALU supports arithmetic, logical, and comparison operations; extensions such as maximum/minimum, merge, absolute value, and absolute difference are implemented as well. Figure 2.5 shows the implementation of the ALU. The first adder performs most of the arithmetic operations, and the second adder is used to execute the logical and the min/max, absolute value/difference operations. The barrel shifter is implemented by log(n) levels of multiplexers controlled using a dedicated shift decoder. The hardware multiplier is implemented using the DSP blocks in Stratix III [4]. The multiply-accumulate (MAC) unit is another piece of logic that utilizes the DSP blocks; it is used to realize the multiply-accumulate operations, which are quite common in signal

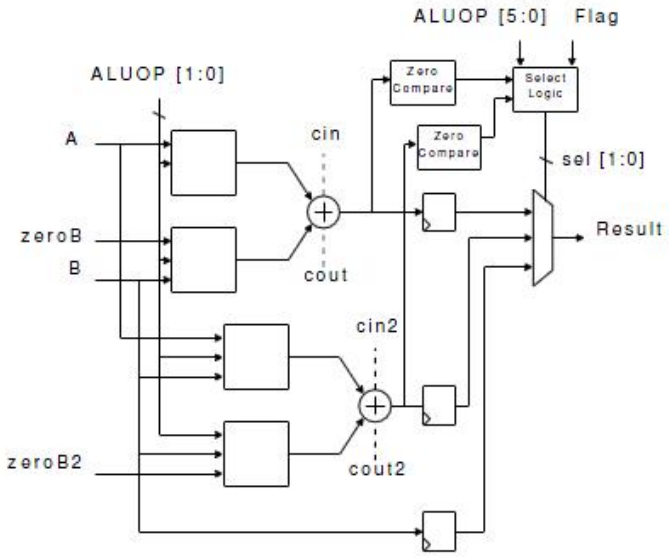


Figure 2.5: ALU Implementation (Source: [29])

processing algorithms.

VIPERS features a distributed vector register file, where a partition of the register file resides in each vector lane. The register file is element-partitioned, so each vector lane has access to all registers in the vector register file, but only a few elements of each register. This divides the register file so they can fit nicely into the on-chip memory blocks. The VIPERS ISA defines 64 vector registers. The Altera M9K memory blocks are used in 256×32 mode to implement the vector register file, allowing four 32-bit elements to fit naturally in each lane. Hence, the default maximum vector length is $4 \times NLane$. The register file data in each lane is duplicated to provide two read ports.

Two FIFO queues are used to buffer load and store data between the vector lanes and the memory crossbar. For vector memory store, the datapath can accept the next instruction after data is transferred from the register file to the store data buffer. For vector memory load, the memory interface loads data into the load queue independently from the vector lane operations. Once all data has been loaded into the buffer, a non-pipelined micro-operation interrupts the current vector operation and moves the data from the load buffers

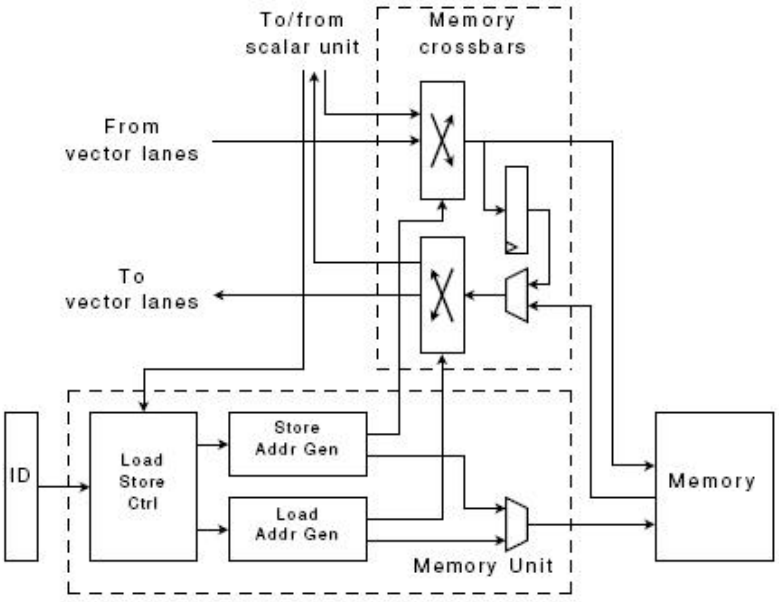


Figure 2.6: Memory Interface Unit (Source: [29])

to the vector register file.

2.2.3 Memory Interface Unit

The memory interface unit, shown in Figure 2.6, is in charge of memory accesses for both the scalar and vector cores. The memory unit consists of a load/store controller, a read interface and a write interface. It supports all three vector addressing modes: unit stride, constant stride and indexed access. This is implemented with separate read and write crossbars to align the data from memory to vector lanes and vice versa.

The load/store controller is a state machine controlling the memory unit operations. It handles memory access requests from both the scalar and vector cores sequentially, and stalls the requesting core if it is busy. A FIFO queue is used to order the memory access request coming from the two cores. The controller also generates control signals for the load and store address generators and the memory crossbars.

The read interface includes the read address generator and the read crossbar, which is

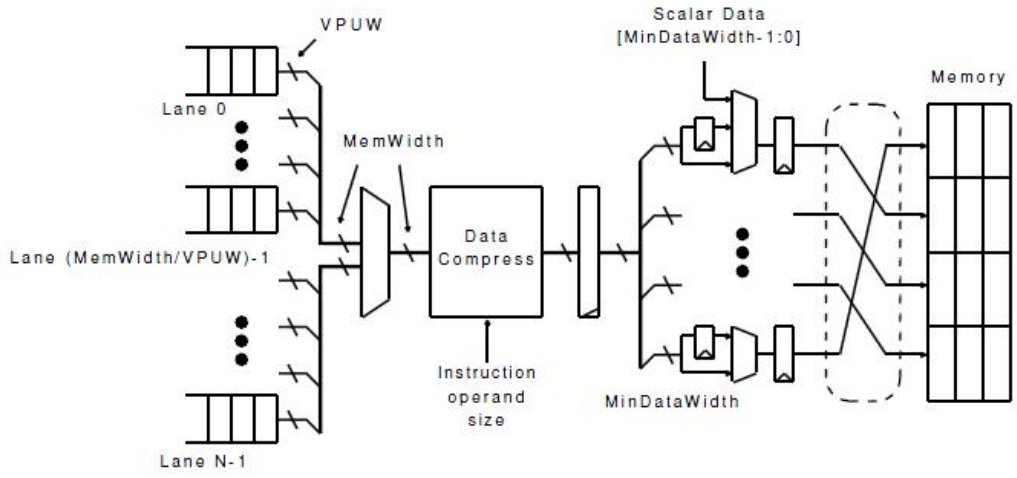


Figure 2.7: VIPERS Write Interface Datapath (Source: [29])

MemWidth bits wide and *MinDataWidth* bits in granularity. *MemWidth* defines the width of external memory and *MinDataWidth* defines the smallest data size that can be used for memory access. The address generator calculates how many data elements can be aligned and stored into the vector lane. For unit and constant stride loads, the read interface aligns up to $\frac{MemWidth}{MemDataWidth}$ number of elements per cycle, where *MemDataWidth* is the operand size as defined by the vector load instruction. For example, at the maximum memory width of 128-bit, the unit stride and constant stride loads can align 16 byte-size data elements per cycle.

The write interface consists of the write address generator and the write interface datapath, shown in Figure 2.7. The write interface datapath consists of three stages: input data selection, data compression, and data alignment. The multiplexer between the vector lanes and the data compress block is only needed when $NLane \times VPUW > MemWidth$, because the amount of data to store will be larger than the available memory space that can be written to in one cycle. The write interface concatenates the selected *VPUW* wide elements into a *MemWidth* wide memory word, where *VPUW* is the width of the vector lane processors. The data compress block truncates the memory word based on the operand

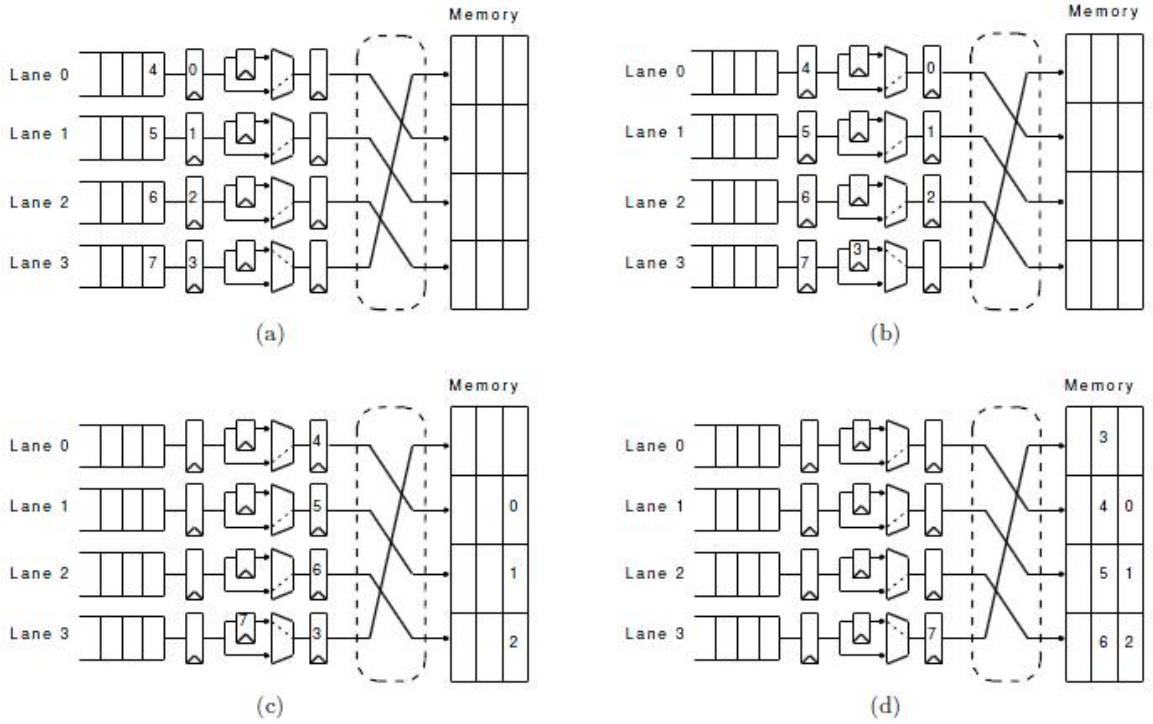


Figure 2.8: Data Alignment Example (Source: [29])

size. The memory word is then aligned by passing through the selectable delay network [5] and the crossbar before written into memory.

The write address generator can generate a memory address each cycle, writing up to as many data elements as can be fit into a single memory slice. With unit stride and constant stride access, not all data from the vector lanes are written in the same cycle due to the offset of memory location. Therefore, the selectable delay network is introduced to buffer the data accordingly so they can be written to the correct memory slice. The example in Figure 2.8 illustrates how data element 3 is delayed to be written to the same memory slice as elements 4, 5 and 6. As the selectable delay network controls the timing of memory writes, the crossbar is in charge of the rearranging of data elements. The unit and constant stride stores can store up to $\frac{MemWidth}{MemDataWidth}$ number of elements per cycle. The indexed load and store operations executes at one element per cycle. Note the alignment crossbar control

logic contains the critical path of the system.

The memory interface connects to an on-chip main memory, located on the bottom right of Figure 2.6, implemented with several Altera M144K memory blocks. Depending on the Stratix III device used, the M144Ks offer an overall capacity between 96kB and 768kB, which should be sufficient to buffer most embedded applications. In cases where higher capacity is required, the memory interface can be connected to an off-chip 128-bit SDRAM, which is capable of burst reading and writing of long vectors.

2.3 Related Works

VIPERS II utilizes on-chip scratchpad memory to replace the vector register file. It also utilizes address registers to index the scratchpad memory. This section considers the prior use of these two architectural features.

2.3.1 Scratchpad Memory

In today's embedded system, memory plays a major role in both cost and performance [11]. Moreover, as the demand for low power portable devices increases, researchers have considered replacing cache with on-chip scratchpad memories. Software-managed scratchpad memory, without the added hardware for tag matching, has area and power advantages over cache, provided that there is an efficient compiler for allocating data in the scratchpad memory. The area and power savings were quantified in [7], where scratchpad memory demonstrated an average energy reduction of 40% and average area-time reduction of 46% over cache memory.

Since then, many scratchpad-based embedded processors have been developed in both academia and industry. The CELL processor [10, 12, 23] from IBM, which is designed for streaming multimedia computations, features 8 synergistic processor elements each having its own SRAM scratchpad which is filled and emptied using DMA operations. The

Signal-Processing On-Demand Architecture (SODA) [18, 19] for 3G wireless protocols has a global scratchpad memory and local scratchpad memory for each of its 4 SIMD processors. ARM processors, such as the ARM1136, ARM926, and Cortex-R4, support both cache and scratchpad memory, so the user can tune the processors based on the needs of a specific application.

With the introduction of scratchpad memories, researchers also investigated in efficient data allocation strategies and on-chip memory organization to enhance performance. For a joint cache and scratchpad system, [21] presents a scheme for partitioning the scalar and array variables into the off-chip memory (cached) and scratchpad memory to minimize cache misses. In [6], a technique for static data allocation to heterogeneous memory units at compile time is presented. Dynamic memory allocation approaches are also discussed in [14, 25].

Due to the large size of vector register files, it makes sense to have it replaced by a vector scratchpad memory. To the best of our knowledge, VIPERS II is the first vector processor to feature an on-chip scratchpad memory. The scratchpad memory in the VIPERS II microarchitecture is most similar to the local memory of the CELL processor. The scratchpad memory in these two architectures are distributed and local to each of the processing units, and both are accessed using address registers and are populated via DMA operations. However, the VIPERS II offers variable vector lengths, which can lower loop overhead.

Due to the large size of vector register files, it makes sense to have it replaced by a vector scratchpad memory. It was also demonstrated in [20], that using software managed data allocation with array variable rotation makes a hardware register file unnecessary.

2.3.2 Address Registers

Address registers have long been used in processors for indirect accessing of memories, where the effective address of an operand is computed based on the address register content and

the addressing mode. There are many addressing modes and the available addressing modes depend on the processor architecture. Auto-increment and auto-decrement, which modify the address register contents as a side effect, can be used in a loop to step forward/backward through all elements in an array or vector; together they can also implement a stack.

Indirect access of memory via address registers can also be found in vector processors. The Cray-1 [24] uses eight dedicated address registers for memory accesses. The Torrent-0 [5] supports unit-stride (with auto-increment) and strided memory accesses by computing the effective address from its scalar registers, and indexed memory accesses by computing the effective address from its vector registers. The VIRAM [15, 16] also supports unit-stride, strided, and indexed accesses, but the base addresses and stride values are stored in a separate register file to comply with the MIPS ISA.

The register pointer architecture (RPA) [22] proposed using register pointers to indirectly access a larger register file without modifying the instruction set architecture. It also demonstrated that by changing the register pointer contents dynamically, the need for loop unrolling to exploit data locality can be reduced. The same technique can be applied to address registers, as they are essentially pointers to the memory.

The address registers in VIPERS II architecture and RPA have very similar features. Both architectures can dynamically modify the pointer values using auto-increment or circular addressing, which decrease the number of pointer value update thus lowering the loop overhead. The implementation of the circular addressing mode is different in the two architectures. The RPA uses two registers to store the begin and end addresses of the circular addressing region, where as the circular buffer feature in VIPERS II is implemented with only one register defining the size of the circular buffer region.

Chapter 3

VIPERS II Architecture

This chapter describes the design and implementation of VIPERS II, targeted to the Altera Stratix III family of FPGAs. The chapter first gives an overview the VIPERS II architecture, outlining the system-level connections to the Nios II scalar processor and the external DDR2 memory. Then, each of the major architectural changes are examined, explaining why the changes were made and what advantages they provide, and describing the design and implementation of these modifications.

3.1 System Overview

The previous chapter described the original VIPERS architecture and how it was able to accelerate data-parallel operations. It also pointed out some areas that the VIPERS could be improved upon, such as the load/store latencies and inefficient memory usage. The design of the new VIPERS II architecture aims to reduce, if not eliminate, the effects of the above mentioned shortcomings of the original VIPERS. This is accomplished with the use of a scratchpad memory that can be accessed directly by the vector core. Since on-chip memory resources are scarce, data are transferred by DMA into the scratchpad buffer instead of using a vector data cache.

Figure 3.1 shows the system-level view of the VIPERS II architecture, consisting of a Nios II processor, a vector processor, an external DDR2 memory, and a DMA engine. VIPERS II is designed to work as a coprocessor accelerator to Altera's Nios II soft processor. The Nios II is in charge of dispatching vector instructions to VIPERS II and controlling the

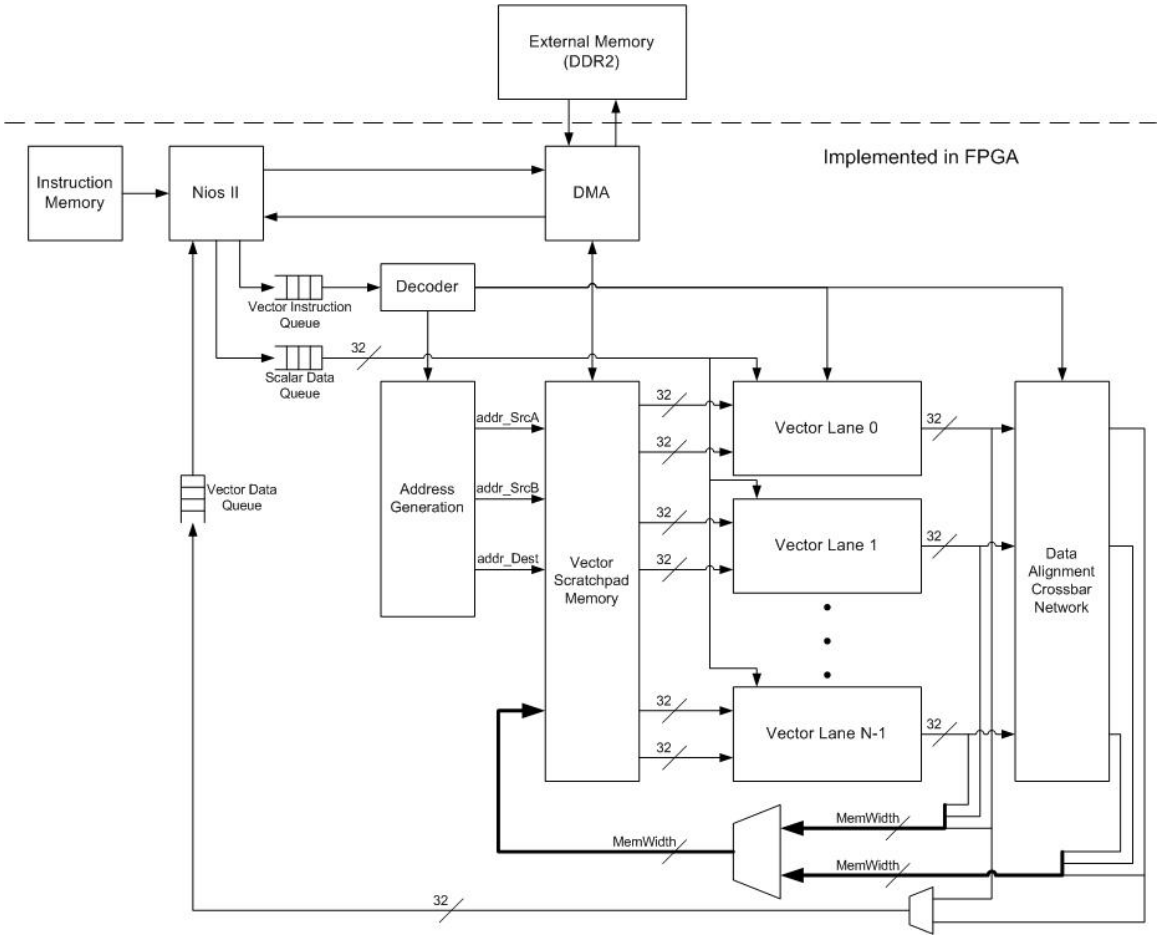


Figure 3.1: VIPERS II Architecture

DMA engine to load the appropriate data from the external DDR2 memory into the on-chip vector memory. The vector processor does not access the instruction memory; vector instructions are treated as extended custom instructions of the Nios II processor. Vector instructions are usually multi-cycle, depending on the vector length, so a vector instruction queue is used to dispatch the vector instructions from the Nios II to allow concurrent execution of scalar and vector instructions. The scalar and vector data queues are used to synchronize the data transfers between the scalar and vector cores for vector instructions that requires both cores. For vector instructions that write back to the Nios II register file, the Nios II processor is stalled until the result is generated by the vector core.

Instead of using a vector register file to hold the working set as vector data, VIPERS II uses address registers to directly access the vector scratchpad memory. The data read from vector memory are sent directly to the vector lanes for processing and the results are written back into the memory. This direct coupling of memory and vector lanes simplifies the memory access as it does not require the read crossbar. A data alignment crossbar network replaces the write crossbar and is used automatically whenever vector data alignment is required.

The system-level connection between the VIPERS II vector processor, the Nios II scalar processor, and the external DDR2 memory was designed by a fellow student, Aaron Severance. The Nios II processor communicates with VIPERS II via its custom instructions [3]. The Nios II provides a synchronous custom instruction interface that can be used to implement external hardware with low latency access to the Nios II register file and immediate values encoded into the custom instruction. Since there are not enough immediate bits in a Nios II custom instruction to encode the VIPERS II ISA, each VIPERS II instruction is issued by executing two Nios II custom instructions, the first sending the least significant 16 bits, the second sending the most significant 16 bits. This extended custom instruction includes an 8-bit field to specify the vector operation to execute; therefore, allowing up to 256 different operations to be defined. In the VIPERS II architecture, other than the dispatching of vector instructions, custom instructions are also defined for writing to and reading from VIPERS II control registers and DMA Engine control registers.

The DMA Engine encapsulates a queue of outstanding DMA requests and hardware to service them. DMA requests are serviced in order, but not sequentially with respect to the VIPERS II instruction stream, allowing computation to overlap with communication. The user can guarantee ordering if needed by polling the number of outstanding DMA requests until it reaches 0, at which point all memory transfers have completed, before issuing a VIPERS II instruction; or by issuing a vector control register read that sends a scalar result to a Nios II register, which will synchronize the two instruction streams, before issuing a

DMA request. The DMA engine does not have access to the Nios II data cache, so the user is responsible for either allocating data as uncached or flushing the data from its cache before initiating a DMA transfer. The user is also responsible for aligning the start addresses in VIPERS II local memory and external memory to the correct alignment. This alignment is adjustable by a top level parameter from 1 byte, allowing for any alignment but with the most resource usage, to the memory width in bytes. The DMA engine generates a full set of the memory signals including: data input/output, write enable, and byte enable, which are connected to dedicated ports on the scratchpad memory. Having a dedicated channel for external access simplifies the interfacing of DMA and the scratchpad memory, as there is no contention or waiting when the DMA needs to access vector memory.

Just like VIPERS, the new VIPERS II architecture offers scalable performance and area by allowing the user to set the primary parameter *NLane*, the number of vector lanes, to include in the design. However, as the vector lanes are coupled directly to the vector memory, the number of lanes directly determines the memory width, *MemWidth*. For a four lane vector processor, the memory width is fixed at $4 \times 32 = 128$ bits. Other fine-tuning parameters will be pointed out in the following sections.

3.2 New Pipeline Structure

As can be seen from Figure 3.1, quite a bit has changed from the original VIPERS architecture to the new VIPERS II architecture. The motivation behind all these changes was the performance bottleneck in the original design: the high latency that occurs when copying data from memory to the vector register file [30].

This disadvantage can be reduced or eliminated by changing the pipeline structure of VIPERS. Looking at modern FPGAs, most of the devices offer high speed on-chip memories, which can be accessed without much concern over memory latency. For example, the TriMatrix memory in Altera's Stratix III and IV family of devices can operate up to 600MHz;

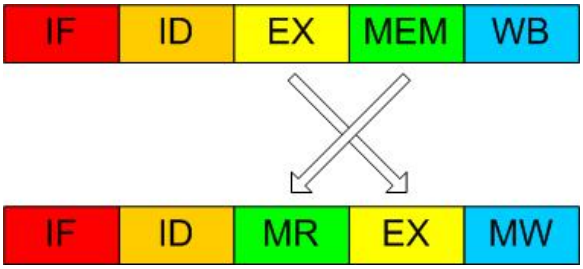


Figure 3.2: New Pipeline Structure

the integrated block memory in Xilinx's Virtex 4 and 5 can be accessed at 500MHz and 550MHz respectively. Utilizing these on-chip memories, the vector processor can access data directly from the memory instead of going through the vector register file.

In VIPERS II, the general pipeline structure is changed from the classical 5-stage pipeline as shown in Figure 3.2 to allow direct access of the vector scratchpad memory. The change involves swapping the execute and memory access stages. In the resulting pipeline, the instructions are decoded and the source operand locations are fetched from the address registers during the instruction decode stage. The memory read stage retrieves the vectors from memory over several cycles and send them to the execution stage for processing in a pipelined fashion. The results are written directly back into the scratchpad memory at the memory write stage. The actual number of pipeline stages in VIPERS II varies from five to sixteen or more because the depth of the data alignment network changes with the memory width. Figure 3.3 shows where the pipeline registers (indicated by the dash lines) are located in each of the major components in the VIPERS II architecture.

The scratchpad memory operates at twice the frequency of the vector core, allowing both the read and write to occur in a single cycle. The FPGA memories are naturally dual ported, so this double pumping allows a total of four memory ports. These are allocated as shown in Figure 3.3: memory read 1 and 2, memory write, and DMA (read or write).

The VIPERS II resolves read after write (RAW) hazards through pipeline interlocking. The decode stage detects data dependency by comparing the address register **names** used

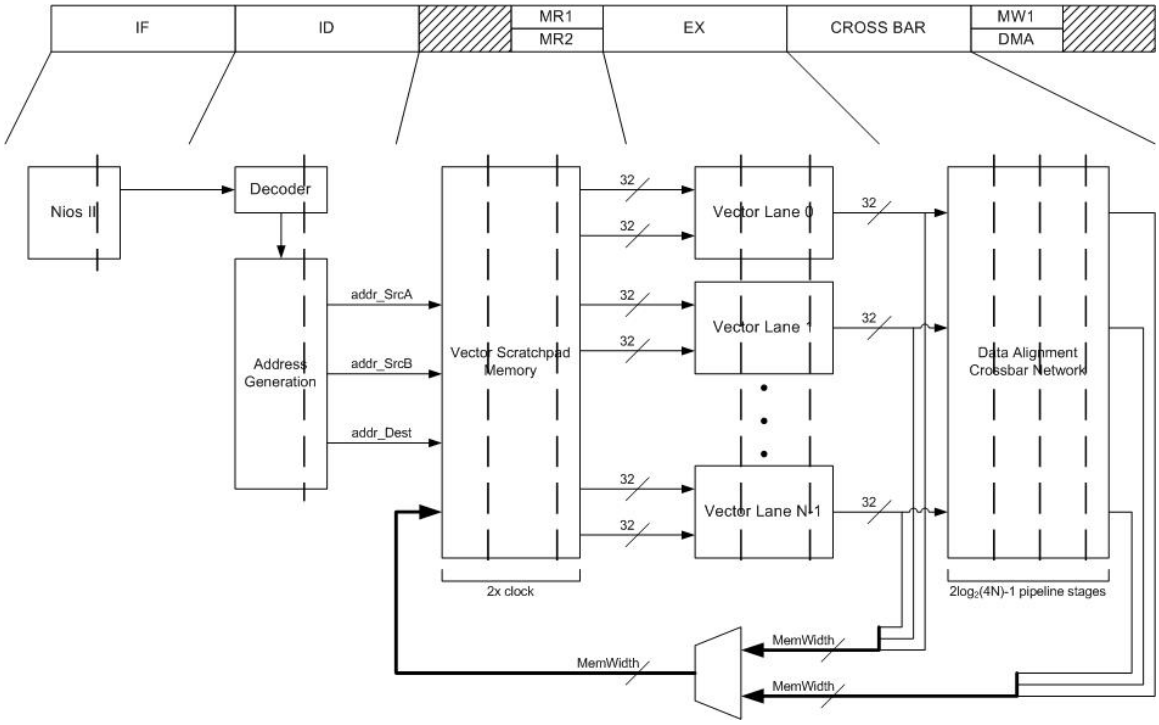


Figure 3.3: VIPERS II Pipeline

in the instructions. If an instruction reads from an address register that is being written to by a previous instruction that has yet to pass through the pipeline, the later instruction is stalled until the hazard is resolved. Note that the actual read and write addresses are not compared, only the address register names; therefore, it is up to the user to ensure that address registers are pointing to non-overlapping regions in the vector scratchpad memory.

To realize this new pipeline structure, four major architectural changes are made in the VIPERS II:

1. Use of a vector address register file instead of a vector data register file,
2. Vector scratchpad memory,
3. Data alignment crossbar network (DACN), and
4. Fracturable ALUs.

The subsequent sections will provide details on the implementation of these changes.

3.3 Address Registers

In the original VIPERS, each vector lane holds a partition of the vector data register file. Vector load and store instructions are used to copy data from memory and distribute them into the vector register file. Other than the latency of the load/store operations, the register file also causes unnecessary data duplications. Most vector instructions require two input operands, so an extra copy of the register file was implemented to provide dual read ports for the vector lanes. As a result, prior to being processed, each datum in the working set resides in 3 different locations: 2 copies in the register file plus the original in the main memory or a cache. This is an inefficient usage of the precious on-chip memory resources.

3.3.1 Address Register Usage

The VIPERS II architecture replaces the partitioned vector register file with a set of 32 address registers for accessing the scratchpad memory. This provides us with three key advantages:

1. Performance gain by eliminating the slow load/store operations,
2. Only one copy of the vector data resides in the memory, and
3. Greatly increased maximum vector length, which can grow to fill the entire scratchpad.

These address registers point to locations in the scratchpad buffer where the vector data are stored. This allows the vector data to be fetched directly from the scratchpad memory, eliminating the need for load and store operations, which results in an improvement in performance. Moreover, since there is no register file and data are read directly from memory, the vector processor would need just one set of data to reside in the on-chip memory, resulting in a much more efficient usage of the limited on-chip memory resources.

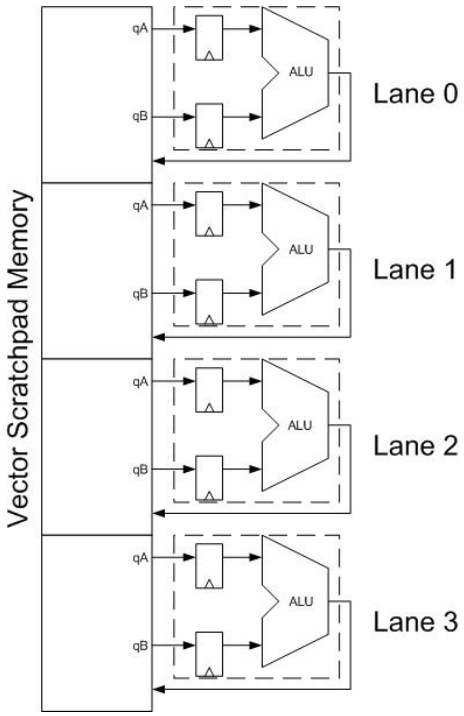


Figure 3.4: Vector Memory & Vector Lane Connections

The output of the vector scratchpad memory feeds directly into the vector datapath. Each 32 bit sector of the memory outputs to a corresponding vector lane as shown in Figure 3.4; therefore, VIPERS II does not require a read crossbar to distribute data among the vector lanes. Figure 3.5 shows an example of vector operations with the new VIPERS II instruction set architecture.

Performance gain from the elimination of load/store operations is maximized when the vector sources and destination are residing in aligned locations in the scratchpad memory. As shown in Figure 3.5, the elements of vectors pointed to by address registers va1 and va2 are aligned, so their first element is feed into lane 1 for processing, second element to lane 2, etc. Elements 3 to 6 are processed on the next cycle and the final element on the third cycle. With this architecture, each instruction will take $\frac{VL \times OpSize + Offset}{MemWidth}$ cycles to finish, where *VL* stands for vector length, *OpSize* is the operand size, and the *Offset* is the starting location from *MemWidth* in number of bits.

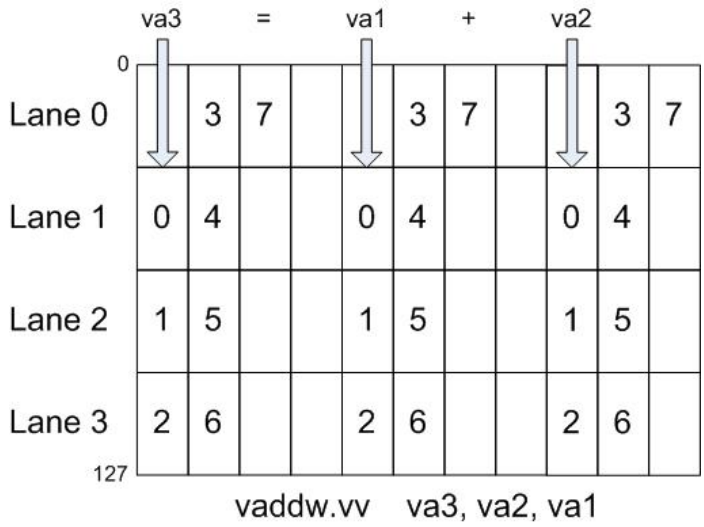


Figure 3.5: Example of VIPERS II Operations

When vectors are not aligned, a move instruction, similar to the load/store operation, is necessary to move the data into alignment. This will be described later in Section 3.5.

To efficiently fit the vector registers into on-chip memory blocks, VIPERS limits the maximum vector length to a multiple of $4 \times NLane$, where $NLane$ represents the number of lanes in the design. For VIPERS II, since all data are stored in the vector memory and address registers are used instead, the limit on vector length is the size of the entire scratchpad memory. This makes VIPERS II more favourable in applications where long vectors can be used.

However, in VIPERS II there is another shorter limit on the maximum vector length determined by the length of the vector flag registers. The vector flag registers store the 1-bit result from conditional operations on vectors. In the current implementation, vector flag registers are implemented using MLABs, and are limited to 32-elements long because of the number of available MLABs. Although there are many MLABs on the chip, the current implementation uses each one as a single 16×1 memory. Even though they are capable of implementing a 16×20 memory, Quartus does not automatically add the logic needed to modify this into a 320×1 memory as needed by the vector flag registers. With more time

and effort, surrounding logic can be built to better utilize the MLAB resources, allowing much longer vector flag registers to be implemented.

3.3.2 Auto-Increment/Decrement and Circular Buffer

The address registers are designed with some useful features: automatic post-increment or pre-decrement of addresses and the capability to be used as a circular buffer. Both of these features provide more flexibility to the address registers and help lower loop overhead in the vector code.

To use the address registers, their contents need to be set and changed by the user at run time using the VMSTC instruction, which copies a given scalar register value to the vector control registers (including address registers). The VMSTC instruction takes 4 cycles to execute and is not pipelined.

In many embedded applications, although the data can be vectorized to exploit data-level parallelism, iterations are still needed to go over the entire range of data due to limitations on vector length. For these cases, address registers need to be set at the start of each loop, which adds significant amounts of overhead. To avoid this overhead, the post-increment and pre-decrement allows the address registers to be set automatically in preparation for the next instruction.

The post-increment and pre-decrement happens at the beginning or at the end of a vector operation, and should not be confused with the increment that is needed to step through the entire vector. As mentioned in the previous section, vector instructions may take several cycles to finish because the vectors can span over a few memory slices. The reading of successive memory slices is accomplished by temporarily incrementing the address register by *MemWidth* and takes place during the execution of an instruction.

To demonstrate the use of post-increment, Figure 3.6 shows the inner loop of the median filter benchmark. This code implements the bubble sort algorithm, in VIPERS and VIPERS II assembly language. In the VIPERS assembly code, the vector is loaded into register v2

```
L14:
vld.b     v2, vbase2, vinc0
vmax      v31, v2, v4
vmin      v4, v2, v4
vst.b     v31, vbase2, vinc1
addi      r2, r2, 1
bge       r6, r2, .L14
```

(a)

```
.L14
vmstc     va2, r4
vmaxb     va31, va2, va4
vminb     va3, va2, va4
vmovab    va2+, va31
addi      r2, r2, 1
bge       r3, r2, .L14
```

(b)

Figure 3.6: Median Filter Example: (a) VIPERS assembly; (b) VIPERS II assembly

at the beginning of the loop, then the maximum and minimum values are written into v31 and v4, respectively. Finally, the maximum result in v31 is stored back into memory and vbase2, which points to the memory location, is incremented. In the VIPERS II assembly, with direct accessing of the scratchpad memory, the load operation at the start of the loop is no longer required. The VMAXB and VMINB instructions perform the exact same operations as their counter parts, with the B suffix indicating the operand size is a byte. The VMOVAB instruction replaces the vector store operation. It performs an aligned move, which moves the maximum data at aligned location va31 to va2; the plus sign denotes post-increment of va2, so va2 would point to the next vector in memory in preparation for the next loop iteration. The auto-increment feature sets up va2 so no VMSTC instruction is required, thus reducing the amount of loop overhead. A more detailed explanation on the implementation of the median filter benchmark is presented in Section 4.1.

Note that writing the VMINB result is achieved with the address registers va4 and va3 pointing to the same location in the vector scratchpad memory. This is because the contents of the address register change as it steps through a vector in memory. Since the write back happens later in the pipeline, va4 is no longer pointing to the start of the vector, so address register va3 is used to ensure the VMINB result is written into the correct location.

The auto-increment/decrement feature also provides the potential to use much fewer address registers than needed by the loop-unrolled code. This is demonstrated later in Section 4.1.

The circular buffer feature resets the address register to its starting location when its

incremented value exceeds the user-determined window size. This is useful when a set of vectors needs to be reused as instructions execute in a loop. Examples of how these features are applied will be shown later with benchmark development in Section 5.1. By decreasing the number of VMSTC instructions required, the auto-increment/decrement and circular buffer features help reduce the loop overheads, as well as the instruction code size.

3.3.3 Implementation

To implement the address registers and their features, four sets of registers are used in the design as shown in Figure 3.7. The *base* register holds the vector memory address of the vector it points to. The *incr* register stores the amount, in number of bytes, to increment or decrement the content of an address register between vector operations. The *window* register indicates the size of the circular buffer. The *next* register stores the value of the incremented/decremented address. Each address register has its corresponding *incr* and *window* registers, so the vector instructions do not require extra bits to indicate which *incr* or *window* registers to use, keeping the vector instruction width to 32-bits.

In addition to the address registers, the scratchpad vector memory is accessed using three address generators, one for each of the two source operands and one for the destination operand. After a vector instruction is decoded, the correct set of *base* address, *incr*, and *window* values are selected and sent to the address generators for computation. The resulting addresses are then output to the vector scratchpad memory so the correct read and write sequence can be carried out.

Typical vector instructions are multi-cycle and the address generators perform different computations at different stages of the instruction. During the instruction execution, the *base* register value is directly incremented by *MemWidth* each cycle to step through the vector stored in the scratchpad memory. This eliminates the need for an extra copy of the address to be kept in the address generator itself. The auto increment/decrement feature is accomplished with the *next* register. In the first cycle of execution, the *next* register content

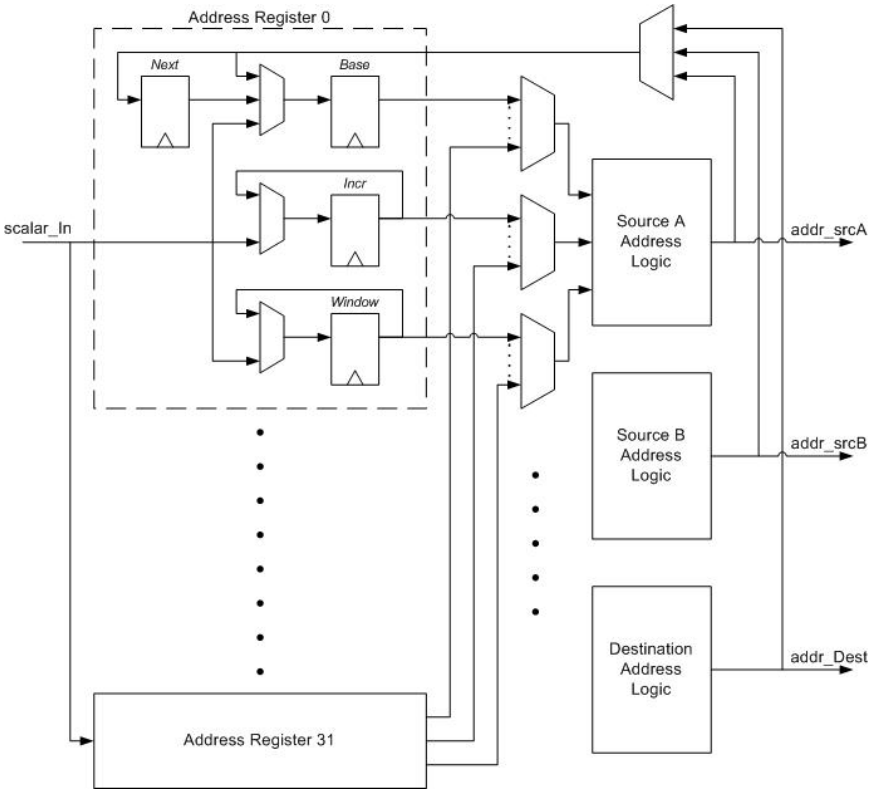


Figure 3.7: Address Generation Logic

is computed based on the unmodified *base* register value and the *incr* register value; if the address register is to be pre-decremented, the decremented value is used as the starting address of the read/write sequence. In the final cycle of the execution, the address value stored in the *next* address register is loaded into the *base* register to set the address register pointing to the leading element of the next vector in memory.

Concurrent access to 3 of the 32 address registers is required for most vector instructions; therefore, flip-flops are used for the implementation of these registers to meet the accessibility demands. However, the flip-flop implementation consumes more logic resources than if the address registers are implemented with memory blocks. To save logic, it is also possible for the VIPERS II processor to be constructed with fewer than 32 address registers. However, this feature is not currently implemented.

3.4 Vector Scratchpad Memory

In the previous section, we demonstrated how the vector scratchpad memory works with the address registers in eliminating load and store operations and resolving the data duplication issues. This section explains the performance gain from eliminating load/store operations in terms of the memory interface design. On top of that, the use of a scratchpad buffer provides a much more flexible storage medium for vector data, allowing the VIPERS II architecture to deliver both performance and memory efficiency.

3.4.1 Memory Interface

By replacing the vector register file with a single scratchpad memory, the interaction between the vector core and main memory in the VIPERS II design is also very different from the original VIPERS.

In VIPERS, the main memory interfaces with the vector lanes via separate read and write crossbars, as shown in Figure 3.8(a). The read crossbar distributes the vector elements to the partitioned register files inside each vector lane, and based on the operand size, the elements are zero/sign-extended to fit the *VPUW*. The write crossbar concatenates the data from the distributed register file and stores them into the main memory. When the size of the elements to store is smaller than *VPUW*, the write crossbar also compresses the data by discarding the most significant bits. With the read and write crossbars, VIPERS can handle three types of memory accesses, unit stride, constant stride, and indexed, along with three memory data sizes, byte, halfword, and word.

VIPERS II, on the other hand, uses a DMA engine to load the vector data from main memory into the vector scratchpad memory, as shown in Figure 3.8(b). The DMA engine acts as a buffer, it collects data from the main memory and then writes them to specific addresses in the scratchpad memory. Different from the crossbars, the DMA engine does not perform strided or indexed access of the main memory, it simply copies blocks of data

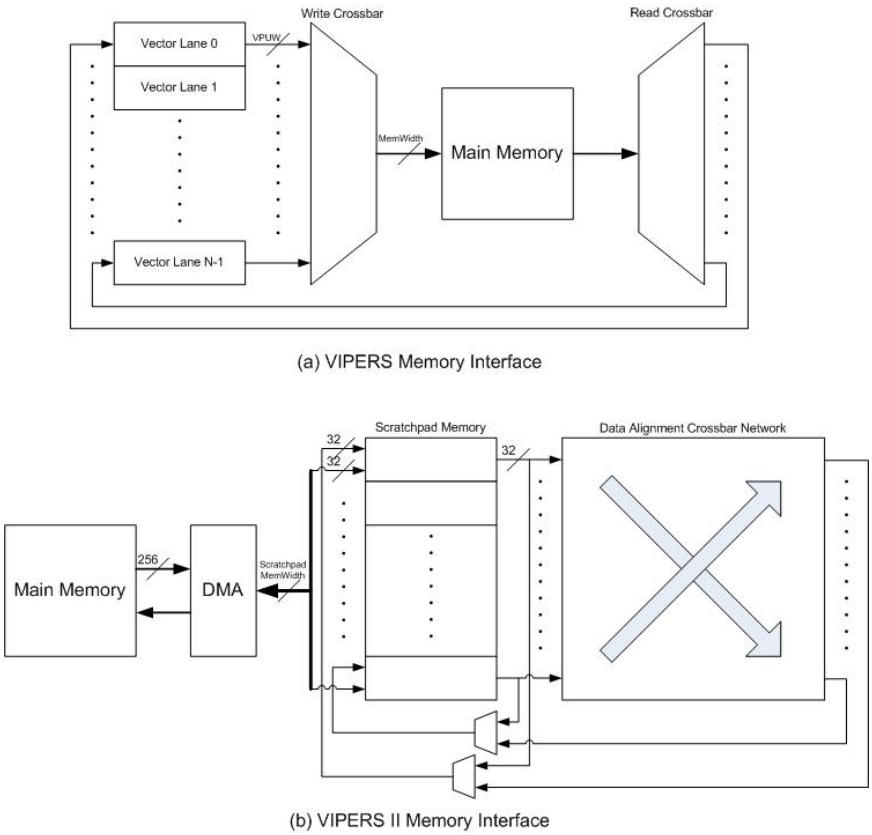


Figure 3.8: Memory Interface Comparison

into the scratchpad memory. In cases where rearrangement of data is required, e.g. vector elements are scattered, the data in the vector scratchpad memory are passed through a data alignment crossbar network (DACN) to form continuous vectors. The data alignment crossbar network is explained in Section 3.5.

3.4.2 Performance Advantage

The scratchpad memory offers improved performance potential over VIPERS in three ways: reduced load and store latencies, fewer load and store instructions, and improved on-chip data capacity through single-copy data storage and the ability to operate directly on byte or halfword data instead of sign-extending it to fit *VPUW*. These features are discussed further below.

The latency in the load and store operations in VIPERS is partly due to the complexity of the memory interface design; in order to implement the strided and indexed accesses, each load/store instruction requires multiple cycles to execute. In comparison, the VIPERS II memory interface, which accesses memory via a DMA engine, is much simpler in design and operation, and offers much faster data transfer between the external main memory and the on-chip scratchpad buffer. Moreover, VIPERS II, which uses the DACN to accomplish strided and indexed moves, does not require data to pass through the slower DACN for all operations. This allows the common case, where vector elements reside in contiguous locations in the memory, to execute without any added delays, and thus resulting in better performance.

Just like VIPERS, the performance of VIPERS II scales with the number of vector lanes in the design; more vector lanes means more data elements can be executed in parallel. The number of vector lanes defines the memory width requirement for the vector scratchpad memory. Moreover, the performance of VIPERS II is maximized when the entire working set fits inside the scratchpad memory. Therefore, both the width and size of the vector scratchpad memory contributes to the performance of VIPERS II. The configurability of FPGAs allows the scratchpad memory to be scaled as needed to deliver the desired performance. In cases where the scratchpad memory cannot fit in a particular FPGA device, there is the option to move to a device with higher capacity.

The configurability of the vector scratchpad memory allows VIPERS II to have as many data vectors as can be fit into an FPGA device. This provides an advantage over the original VIPERS, which can only have a maximum of 64 vectors before resorting to vector load or store operations. For example, in the unrolled version of the median filter benchmark, 25 vectors were used to store the data needed for a 5×5 window. Therefore, the original VIPERS would not be able to unroll median filter benchmarks with a window size larger than 7×7, unless the ISA is changed, which is infeasible. On the other hand, VIPERS II is not limited by its ISA, so it can be used for a median filter benchmark with a large window

size, as long as there is an FPGA with a big enough on-chip memory.

The vector elements are stored in the scratchpad memory in their natural length of 8 bits, 16 bits, or 32 bits. As these data elements are fetched directly from the scratchpad, fracturable ALUs are used to operate on elements with varying sizes. This allows more results to be computed in parallel when the data is halfword or byte-sized, enhancing the performance of VIPERS II. Details on the fracturable ALUs are presented in Section 3.6.

3.4.3 Implementation

The direct coupling between vector scratchpad memory and vector lanes makes the design of vector memory extremely important. The vector memory must have enough bandwidth to meet the demands of the vector datapath. Typical vector instructions require two reads and 1 write in each operation. A three-port memory can be used to meet this requirement. However, because a three-port memory is typically implemented with two identical copies of the memory, it wastes precious on-chip memory.

Another way to achieve the required bandwidth is to operate the memory at twice the clock frequency of the vector processor. Using the original VIPERS's operating frequency of 100 MHz as an approximation, the vector memory would need to operate at 200 MHz, which is well below the advertised frequency of 600 MHz for TriMatrix RAM blocks. These RAM blocks are already dual-ported, so operating at twice the frequency provides four memory ports to VIPERS II.

The vector memory uses two bidirectional ports to support both reads and writes. Combined with the doubled clock frequency, the vector memory allows four accesses per cycle. The reading of operands and writing of result vector uses three of the four channels. The last channel is reserved for loading and storing of data to the external memory via DMA. Currently, the reading of two operands takes place in the second half of the cycle, whereas the writing back of one result and DMA access (read or write) takes place in the first half of the cycle.

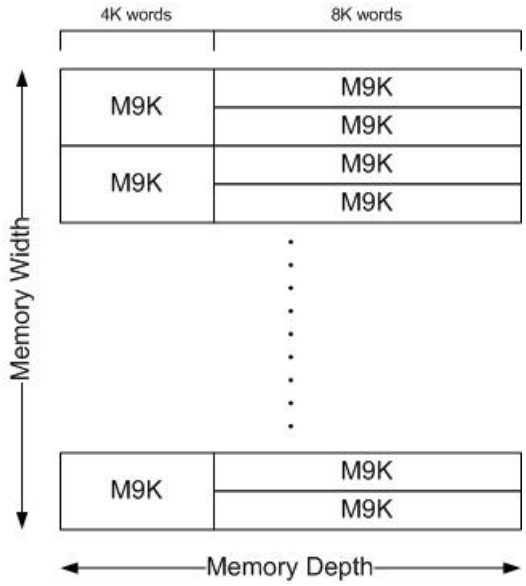


Figure 3.9: Memory block with depth of 12K words

Register banking is another alternative to provide two read ports to the register file without duplication, but it adds restrictions to the register usage. Moreover, by running the scratchpad memory at twice the frequency provides a dedicated channel for DMA access; therefore, the over clocking of memory is selected over register banking to achieve the necessary bandwidth..

The vector memory is implemented using the M9K RAM blocks that are integrated on-chip in the Stratix III family of FPGAs. In bidirectional dual port mode, the widest possible configuration of the M9K is 512×16 bits; therefore, the smallest scratchpad memory depth is set to 512 words to fully occupy a M9K block. This provides a vector memory with capacity of $512 \times MemWidth$ bits, which is 8kB for a four lane vector processor. The widest configuration was selected to keep the M9K usage minimal; however, should an application require more memory, the depth could be easily increased up to 8192 memory words. Even deeper scratchpad memory can be realized by stacking the M9K blocks. Figure 3.9 illustrates how a memory with 12K words can be implemented by stacking three M9Ks, two in 8192×1 configuration and one in 4096×2 configuration.

3.5 Data Alignment Crossbar Network

Since the vector scratchpad memory couples directly to the vector lanes in VIPERS II as shown in Figure 3.4, the vectors involved (sources and destination) in a single vector instruction must reside in aligned locations in the scratchpad memory. When the vectors are not aligned, a move instruction, similar to the load/store operation, is necessary to move the data into alignment.

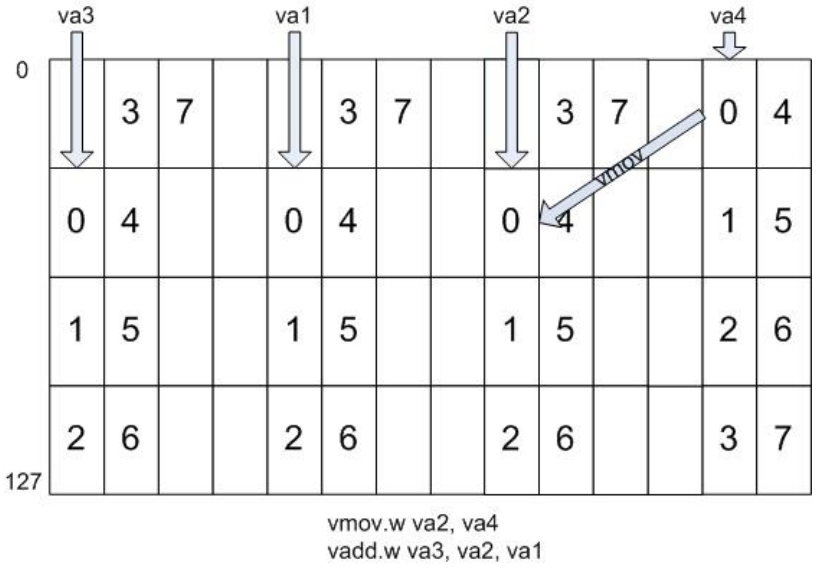


Figure 3.10: Example of Misaligned VIPERS II Operation

Figure 3.10 illustrates how misaligned vectors can be operated on by adding a VMOV instruction. In this example, we want to achieve a vector add of misaligned vectors va1 and va4. If the vectors referenced by va1 and va4 are added without moving, the first element of va1 will be added to the second element of va4 which produces an incorrect result. Therefore, the VMOV instruction is required to move the vector at va4 into an aligned location referenced by va2, which is then operated on to produce the correct results. Although the misaligned operation would suffer a loss in performance due to the extra move instruction, it is expected to be similar in performance to that of the original VIPERS, which needs a vector load and vector store pair to achieve the same result.

The vector move instruction can be achieved using a full crossbar, but these are costly in resource usage. This rearrangement of data can also be accomplished by multistage switching networks, such as the Clos network [9], Benes network [8], or Omega network, which trades off performance for resources. These networks can achieve all (or some) of the permutations possible with a crossbar, but they require less resources to implement. However, the multistage nature of these networks tend to result in longer delays, and it is harder to generate the control signals.

3.5.1 Permutation Requirements

The VIPERS II employs a fixed set of permutations with its move instructions. The switching network must be able to achieve the permutation needed for these three types of move operations:

1. Offset Moves
2. Strided Moves
3. Indexed Moves

Figure 3.11 gives an example of the permutation needed for each of these three types of moves. The offset moves (equivalent to unit stride load/store) are the most common case; it moves contiguous vector elements to an offset (unaligned) location. The strided move collects elements that are scattered by a fixed distance and stores them in contiguous locations to form a vector. The indexed move is a random move of elements to an indexed location. Due to time constraints, the strided and indexed moves are not implemented in this work.

3.5.2 Implementation

The data alignment crossbar network consists of a tag generation logic and a multistage switching network. In the current VIPERS II microarchitecture, the Benes network is chosen

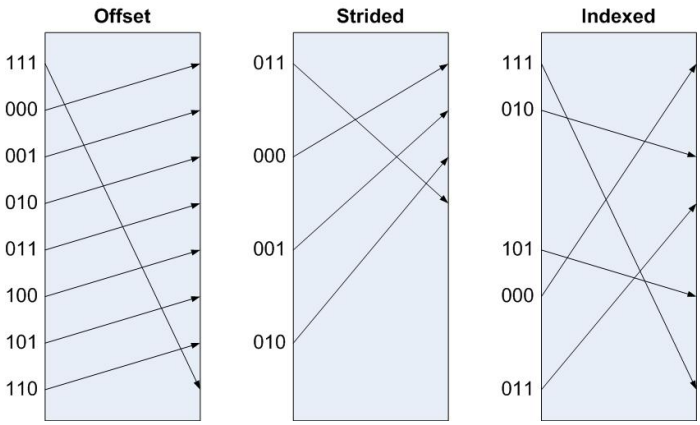


Figure 3.11: VIPERS II Move Operations

to implement the multistage switching network because of its rearrangeably non-blocking characteristic, which allows it to satisfy all three permutation requirements listed above. However, the Benes network requires a control algorithm to effectively route data through the network without blocking. Due to time constraints, only control for vector offset moves is implemented in this work.

Since the minimum operand size is a byte, the granularity of the Benes network is set to byte-size. The tag generation logic computes the destination tags, which are required for controlling the Benes network, for each byte-size element of the input data. A destination tag is the numerical designation for an output, in binary representation. For example, a $N \times N$ network would have its outputs labeled from 0 to N-1, and each valid input is assigned a destination tag showing which output it is going to. Details on tag generation and the control algorithm for a Benes network can be found in Appendix B.

Alternative Implementations

The Benes network is not the smallest switching network available. It uses $\frac{N}{2}(2\log_2 N - 1)$ 2-by-2 switching elements, where N is the number of inputs/outputs of the network. Smaller networks such as the Omega network and Banyan network, which have $\frac{N}{2}(\log_2 N)$ switching elements, typically have blocking characteristics [1] and cannot realize all permutations

listed in Section 3.5.1. However, performance can be traded off for area by lowering the permutation requirements, allowing the data alignment network to be implemented by smaller networks. For example, if the VIPERS II requires only the offset moves, it can be easily realized by the Omega network, which is about half the size of the Benes network. The Omega network also offers simpler control, as offset moves can be achieved without blocking by bit-controlling the switching elements with the destination tag.

Moreover, since the Omega network has identical connection patterns in every stage, if a longer delay through the data alignment network is tolerable, the data alignment network can even be implemented with a time-multiplexed Omega network. This lowers the resource cost to as low as $\frac{N}{2}$ switching elements. Despite the benefits, an Omega network was not implemented because it cannot realize strided vector moves.

3.5.3 Misalignment Detection and Auto Correct Mechanism

As mentioned in Section 3.3.1, the performance of VIPERS II is maximized when the vectors are stored in aligned locations in the scratchpad buffer. Moreover, incorrect results would be computed if the data alignment is not satisfied. To help lower the learning curve in software development, misalignment detection and auto-correction logic is implemented in the VIPERS II design.

The misalignment detection logic compares the lower $\log_2 MemWidth$ bits of the source and destination addresses, and generates three signals: misaligned_src, misaligned_dstA, and misaligned_dstB based on the comparison results. If the two source addresses are misaligned, misaligned_src is asserted; if the destination and source A is misaligned, misaligned_dstA is asserted; if the destination and source B is misaligned, misaligned_dstB is asserted. Based on the state of these signals, the vector data are moved into alignment to enable the vector core to carry out the correct computation.

Figure 3.12 shows in a flowchart the actions taken by the VIPERS II to correct the misalignment. At the top of the flow chart, if the two sources are misaligned with each

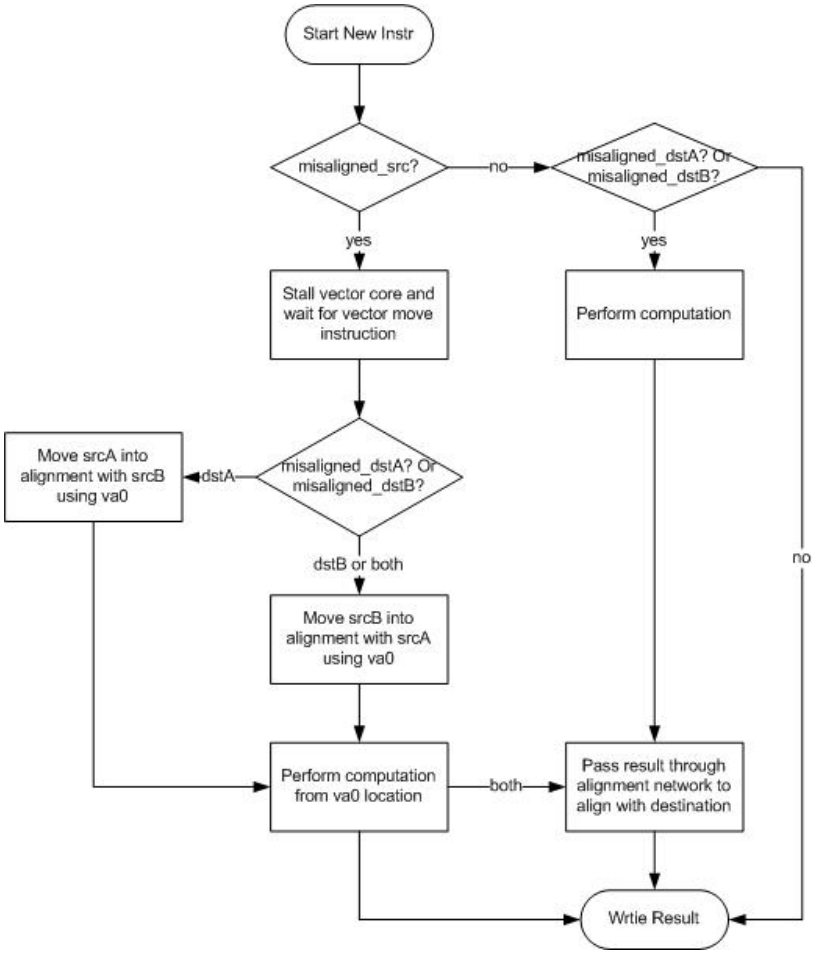


Figure 3.12: Misalignment Correction Logic

other, a vector move must take place so the processor is stalled and waits for a move instruction to be automatically inserted by the correction logic. When the sources are not aligned, if the destination is not aligned with source A only, it means the destination and source B are in alignment so source A needs to be moved; the opposite applies for misaligned destination and source B. If all three addresses are different, the sources are first moved into alignment for computation, then the result is passed through the data alignment network again before it can be written correctly into the destination address. When the sources are aligned with each other but not to the destination, computation is carried out as normal, and the result is passed through the alignment network to align it with the destination

address.

The alignment correction of a source operand is done by performing offset moves of the unaligned vector to an aligned location pointed to by va0, an address register reserved specifically for alignment correction purposes. The base value of va0 can be set by the user via the VMSTC instruction like all address registers, but its lower $\log_2 MemWidth$ bits are updated with every alignment request to achieve the correct alignment. The user should reserve a section of scratchpad memory for reference by va0, because the data in this region may be overwritten at anytime.

Although the auto-correction could prevent mistakes in the computation due to misalignment of data, it erases any performance gain achievable with the new VIPERS II architecture. To help users fine-tune the software, counters are implemented for each of the three signals to keep track of how many misalignment cases there are in the vectorized code. The user can retrieve the counter values via control registers, and utilize this information to optimize the performance of the application.

3.6 Fracturable ALUs

The processing units in the original VIPERS can be compiled to have a fixed width of 8, 16, or 32 bits according to the *VPUW* parameter. The size of the vector elements in the register file is always *VPUW*, but the size in memory may be smaller; the vector load and store instructions perform sign-extension or clip the most significant bits when moving data to memory. In contrast, the VIPERS II architecture does not sign-extend on memory reads to match *VPUW*. The direct coupling between the scratchpad memory and the vector lanes gives a fixed input width of 32 bits for each lane. However, it still needs to have the capability of operating on halfword and byte sized elements. Therefore, each lane must have a processing unit that can execute on operands with varying widths as shown in Figure 3.13. Furthermore, each instruction must specify the operand size to the processing units.

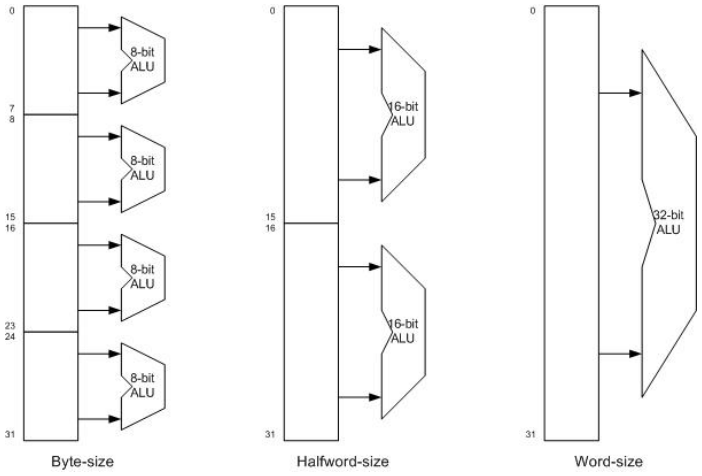


Figure 3.13: Processing Unit with Variable Width

In order to accomplish this, fracturable ALUs are introduced into the vector lanes and all arithmetic and logic vector instructions in VIPERS II carry an operand size indicator.

The fracturable ALUs perform arithmetic operations using a fracturable adder, and the multiply and shift operations are executed with a fracturable multiplier. The following subsections describe the implementation of the fracturable adder and multiplier.

3.6.1 Fracturable Adder

Figure 3.14 shows the fracturable adders as four 8-bit adders connected together via the carry-in and carry-out ports. The multiplexers on the carry-in ports are controlled by the operand size indicators in the instructions, setting the length of the adder chains. The implementation of the fracturable ALU overall is similar to that of VIPERS, which is shown in Figure 2.5, but with the two adders replaced by fracturable adders. As mentioned in Section 2.2.2, the two adders are required for the enhanced instructions such as minimum, maximum and absolute difference. If the particular application does not benefit from these instructions, VIPERS II can be configured so the second fracturable adder is removed to reduce resource usage.

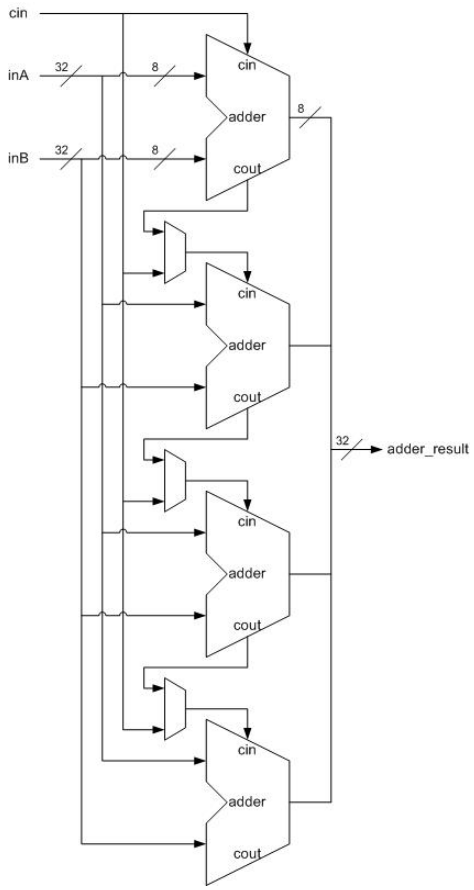


Figure 3.14: Fracturable Adder

3.6.2 Fracturable Multiplier

Design of the fracturable multipliers are not as straight forward as the fracturable ALUs. It is important to tailor the design so the FPGA resources are efficiently utilized. If the multiplier arrays are implemented by logic elements on the FPGA, it would be very flexible and the fracturable features can be added easily. However, the resource usage and performance might not be as good as using the hardware multipliers in Stratix III's DSP blocks.

The hardware multipliers, as black boxes, do not provide access to the internal signals necessary to make it fracturable. Therefore, the fracturable multipliers have to be built from smaller hardware multiplier blocks, whose results are combined using the partial product

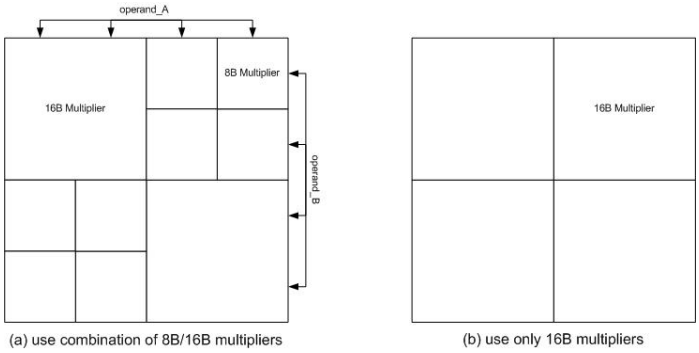


Figure 3.15: fracturable Multiplier Implementation

method to generate results for larger operand sizes. In VIPERS II, 16-bit multipliers are chosen as basic building blocks of the fracturable multipliers, because they are most suitable given the Stratix III's DSP block architecture. Each DSP block can be used to implement eight 8-bit multipliers, four 16-bit multipliers, or two 32-bit multipliers.

Figure 3.15 illustrates two possible implementations of the fracturable multiplier. The approach in Figure 3.15(a) has the advantage that the input operands can be fed directly into each multiplier block; however, it would cost one and a half DSP blocks and extra logic to compute a 16-bit multiply result. Figure 3.15(b) shows the current implementation, which consists of four 16-bit hardware multipliers, that fits nicely into a single DSP block. The only disadvantage is that the inputs need to be reorganized for computing 8-bit multiply results.

The inputs are arranged based on operand size so the four multipliers can be used to directly perform four byte-size or two halfword-size multiplies.

For word-size multiply, Figure 3.16 illustrates how the result can be obtained. The upper 16-bit multiply result is right shifted by 32 bits and the middle multipliers' results are shifted by 16 bits. The output of the shifters are then summed with the lower 16-bit results to form the final product.

The fracturable multipliers also serve as fracturable shifters for the shift operations. The multiplier output are arranged to generate the shift and rotate results. The logic and

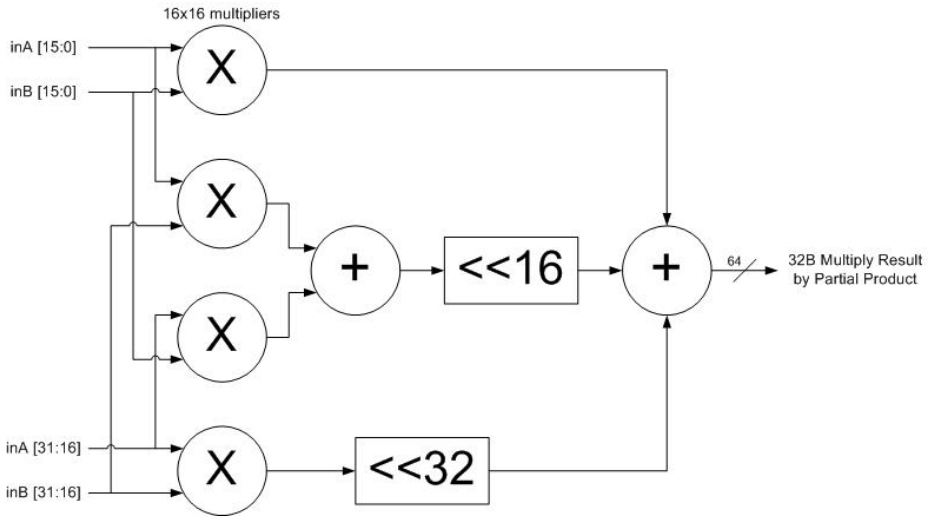


Figure 3.16: 32B Multiply Result by Partial Product Method

arithmetic shift is accomplished with unsigned and signed multiplies, respectively.

Obviously, the fracturable processing unit is implemented at the cost of FPGA resources, but this cost provides more gains than just being able to compute results for various operand sizes. Each vector lane in VIPERS II can generate 4 byte-size, 2 halfword-size, or 1 word-size results; making a 4 lane VIPERS II design equivalent to a 16 lane VIPERS when operating on byte-size vector elements. Furthermore, precious on-chip memory is not wasted by expanding byte-sized data to the $VPUW$-wide register file as done in VIPERS.

3.7 Summary

The VIPERS II microarchitecture employs a new pipeline structure to improve performance by providing the vector core with direct access to a vector scratchpad memory. This eliminates the load and store latencies which were a performance bottleneck of the original VIPERS. The pipeline structure is realized by four major architectural changes: address registers, vector scratchpad memory, data alignment crossbar network, and fracturable ALUs; each of them bringing different advantages to make the VIPERS II architecture more

favourable than VIPERS.

The address register and scratchpad memory combines to eliminate the load/store operations and provide better memory efficiency. The post-increment and pre-decrement features of the address register help lower loop overheads. The vector scratchpad memory also makes the VIPER II ISA more tolerant to scaling applications as the number of on-chip vectors can increase without changing the ISA. Operating the scratchpad at twice the clock frequency provides four read/write ports with the efficiency of just a single copy of data on-chip. The data alignment crossbar network is required to align vectors in memory, and it trades off performance for area by using the smaller switching networks instead of crossbars. The fracturable ALUs allow the vector lanes to process vector data of different sizes; on top of that, they also provide added computational power for the smaller operand sizes and more efficient usage of on-chip memory.

Chapter 4

Impact of Architectural Changes and Resource Usage

This chapter presents the impact of the architectural changes mentioned in Chapter 3 and resource usage results of the VIPERS II architecture. First, the impact of each of the four key architectural changes is examined in terms of resource usage and performance. Then, the overall resource usage of the different instances of VIPERS II is presented and discussed. Finally, the scalability of VIPERS II is discussed.

4.1 Address Registers

In the original VIPERS, benchmarks often need to be unrolled in order to optimize performance. By unrolling the loops in the benchmarks, the number of load and store operations are reduced by keeping the entire working set of data in the register file and reusing the data, resulting in an increase in performance. Figure 4.1 uses the inner loop of the median filter benchmark to illustrate how the loop unrolling improves performance. The inner loop of the median filter implements the bubble sort algorithm. In the unrolled version of the code, the working set of data is first loaded into 25 different vectors, which are then operated on by repeating the set of three instructions (VMAX, VMIN, and VADD) 222 times as shown in the bottom half of Figure 4.1(b). For the median filter benchmark with a 5×5 window, the inner loop at L14 in Figure 4.1(a) would be executed 222 times. By unrolling the loop, the number of load instructions was reduced from 222 down to just 25,

```
L14:
vld.b     v2, vbase2, vinc0
vmax      v31, v2, v4
vmin      v4, v2, v4
vst.b     v31, vbase2, vinc1
addi      r2, r2, 1
bge       r6, r2, .L14
```

(a)

```
;repeat 25 times for v1..v25
vld.b     v1, vbase1, vinc1
vld.b     v2, vbase1, vinc1

;repeat 222 times
vmax      v31, v1, v2
vmin      v1, v1, v2
vadd      v2, v0, v31
```

(b)

Figure 4.1: Median filter in VIPERS assembly: (a) rolled; (b) unrolled

```
L9:
vmovb     va2+, va1+
addi      r3, r3, 1
bge       r4, r3, .L9

.L14
vmstc     va1, r5           ;array[i]
vmstc     va3, r5           ;array[i]
vmstc     va2, r6           ;array[i+1]

.L15
vmaxb     va31, va1, va2
vminb     va3, va1, va2
vmovab    va2+, va31
addi      r2, r2, 1
bge       r3, r2, .L15

add       r5, r5, r4        ;i++
add       r6, r6, r4
addi      r3, r3, -1
addi      r7, r7, -1
bne       r7, zero, .L14
```

Figure 4.2: Median filter in VIPERS II assembly

thus improving performance.

With the direct scratchpad memory access of the new VIPERS II architecture, load and store operation are eliminated, so the performance of VIPERS II is expected to be similar to that of the unrolled VIPERS code without the instruction overhead of unrolling. Figure 4.2 shows the same median filter inner loop vector assembly written in the new VIPERS II instruction set architecture, which can be found in Appendix A.

The VIPERS II approach to the median filter benchmark is very similar to the unrolled version of VIPERS assembly. The first loop at L9 sets up the working set of data in aligned locations in memory, and the second loop performs the bubble sort on the vector data.

The vector instructions used in the second loop are similar to the three instructions used in the unrolled VIPERS code. This is evidence to show that VIPERS II can achieve similar performance to that of the VIPERS unrolled code without unrolling the loop. Taking the loop overheads into account, the VIPERS II version may be slightly lower. However, since the VIPERS II assembly does not require unrolling, the code can be written in 8 lines instead of the 247 lines that is needed for the unrolled code, providing the advantage of a more compact code that occupies less instruction memory space. Also, loop overhead can be reduced by unrolling a small amount, or by improving concurrent execution of the scalar and vector cores, or both.

The auto increment and decrement features of the address register helps to reduce loop overhead and make the code more compact. In the VMOVAB instruction, the contents of address register va2 are incremented to point to the head of the next vector in preparation for the VMAXB and VMINB instructions in the following loop. Without the post-increment feature, a VMSTC instruction would be needed at the start of L14 to set up the address register for each loop. This demonstrates how the auto-increment/decrement feature can help reduce the amount of loop overhead. Another key advantage of the post-increment/pre-decrement feature is that it requires fewer address registers to be implemented. For the median filter benchmark, although 25 vectors are set up in the vector memory, only 4 address registers are required to execute the benchmark because va2 is incremented in each loop to subsequently reference each of the 25 memory locations. This can be used to reduce the resource usage of VIPERS II.

4.2 Vector Scratchpad Memory

In the original VIPERS, the memory usage was inefficient because each vector resides in 3 different on-chip locations: 2 copies in the register file for dual-port reads, plus a third possible copy in the main memory if it resides on-chip. By replacing the register file, the

Architecture	VIPERS				VIPERS II
	Rolled		Unrolled		
VPUW	8-bit	32-bit	8-bit	32-bit	
# of vectors used	3	3	26	26	26
# of data bits used	3072	12288	26624	106496	13312
# of addr. reg. used					4
Total # of addr + data bits					13632

Table 4.1: Memory Usage Comparison of Median Filter Benchmark

new VIPERS II architecture requires only one copy of the vector in the on-chip memory. To quantify the memory usage, a median filter benchmark with byte-size elements and vector length of 64 is used as an example to compare the memory usage of the two generations of VIPERS. Table 4.1 shows the number of bits used in each of the three versions of median filter code shown above. To provide a fair comparison, these numbers assume that the VIPERS uses an off-chip main memory just like the VIPERS II.

The number of vectors used is determined from the assembly code. The unrolled version of VIPERS uses 25 vector registers to store the entire data set, and 1 extra vector to store the temporary maximum result. For VIPERS II, although it uses only four address registers to access the data, it also stores 26 vectors in the scratchpad memory.

The number of bits used for VIPERS can be calculated using the equation: $NBits = 2 \times NVectors \times VL \times VPUW$, where the VL and $VPUW$ stands for vector length and vector processor width. The factor of 2 in the equation accounts for the two copies of register file in the original VIPERS. The $VPUW$ parameter defines the VIPERS architecture at compile time. VIPERS with $VPUW = 8$ is implemented with 8-bit wide register file and ALUs and can only operated on byte-size data elements, which is sufficient for the byte-size median filter inputs. However, a VIPERS design with $VPUW = 32$ would be able to execute vector instructions on all of the supported data sizes, making it much more flexible. The number of bits used for VIPERS II is given by the equation:$NBits = NVectors \times VL \times OpSize$, where $OpSize$ represents the operand size. Operand size refers to the size of the data elements use

Architecture	VIPERS				VIPERS II
	Rolled		Unrolled		
VPUW	8-bit	32-bit	8-bit	32-bit	
Window Size	any	any	7×7	7×7	22×22
Vector Length	1024	256	$\frac{4096}{W^2+1}$	$\frac{1024}{W^2+1}$	$\frac{(65536 \div OpSize)}{W^2+1}$

Table 4.2: Largest Median Filter Benchmark with 64kB Budget

at this point in the program and can be changed at run time. The number of bits used in the address registers are also included. Accounting for the storage required to implement the auto-increment and circular buffer features, each address register in VIPERS II takes up 80 bits.

The unrolled version of the VIPERS code uses 23 more vectors than the looped version, so the performance gain comes at the cost of memory usage. VIPERS II uses only half as many bits as the 8-bit VIPERS, which is entirely due to the elimination of data duplication in the register file. However, since the same VIPERS II design can also operate on all supported operand sizes, we must also compare it to a 32-bit implementation of VIPERS, which requires about 8 times as many bits to implement. This advantage comes from the use of fracturable ALUs in the VIPERS II architecture. With more efficient memory usage, the VIPERS II architecture is able to achieve performance similar to the unrolled code, while keeping the memory usage comparable to the rolled version of the code.

Looking at the memory efficiency from another perspective, Table 4.2 shows the largest median filter that can be built given a 64kb budget for the on-chip memory (register file in VIPERS and scratchpad memory in VIPERS II). First, we look at the largest window size that can be supported by each architecture. The rolled version of VIPERS loads the vector from main memory in every loop, so the on-chip memory budget does not limit the window size. As mentioned in Section 3.4, the unrolled version of the VIPERS, limited to the 64 vector registers defined in the ISA, can only support a window size up to 7×7. In contrast, the VIPERS II can support a maximum 22×22 window with a vector length of 16.

Architecture	VIPERS - Crossbars			VIPERS II - DACN			VIPERS II / VIPERS
MemWidth	128	256	512	128	256	512	
V4	1177	-	-	768	-	-	$\frac{768}{1177} = 0.653$
V8	-	5372	-	-	1718	-	$\frac{1718}{5372} = 0.315$
V16	-	-	18730	-	-	2942	$\frac{2942}{18730} = 0.160$

Table 4.3: Resource Usage Comparison between Data Alignment Crossbar Network and Crossbar

The second indicator used is the maximum vector length that can be supported. This can be calculated from dividing the available memory by the number of vectors needed for the implementation. The rolled version of VIPERS requires a total of three vector registers to implement, so vector registers would support up to a vector length of 1024 for the 8-bit implementation, and 256 for the 32-bit implementation. For the unrolled version of VIPERS and VIPERS II, the number of registers used depends on the window size, so the maximum vector length supported is shown as a function of window size, W. For VIPERS, the available memory in the numerator can be calculated by $\frac{32768}{VPUW}$; due to the duplicate register file, 32kb is used instead of 64kb. For a window size of 7×7 bytes, the maximum vector length for VIPERS is 81 and for VIPERS II is 163.

4.3 Data Alignment Crossbar Network

In VIPERS II, the rearrangement of data is achieved using the data alignment crossbar network instead of the read and write crossbars. This decision is made in an attempt to trade off some performance for resource usage. Table 4.3 shows the resource usage in number of ALMs for the read/write crossbars and the data alignment network. The resource usage is shown for different combinations of memory width and number of vector lanes. These results confirm that the data alignment network consumes less resources than the crossbars.

The read crossbar in the original VIPERS distributes data into the vector lanes, so the memory width doesn't have to match the width of vector lanes. Table 4.4 shows that by keeping the memory width fixed at 128-bits, the VIPERS can keep the size of its crossbars

Architecture	VIPERS		
NLanes	4	8	16
# of ALMs used	1177	1131	1189

Table 4.4: VIPERS Crossbar at MemWidth = 128

Architecture	VIPERS(V4)-Unrolled	VIPERS II(V4)		
Operand Size	ALL	32-bit	16-bit	8-bit
Total # cycles	3024	4480	3296	2815
# cycles/pixel	189	280	206	176

Table 4.5: Cycle Count Comparison of Median Filter Benchmark

relatively constant regardless of the number of vector lanes in the design. So the VIPERS design still has an area advantage when the memory width is narrow and fixed, but it would also be slower because of the limited bandwidth.

4.4 Fracturable ALUs

As mentioned in Section 3.6, the fracturable processing units of VIPERS II is not only capable of computing operands of different width, but also capable of computing on more elements when using byte-size or halfword-size data. Table 4.5 shows the cycle count result from ModelSim simulations of the median filter benchmark with a vector length of 16. These results demonstrate the improvement in performance as the operand size decreases.

The VIPERS result shown here is the result for the unrolled version of the benchmark. As mentioned in Section 3.3, the VIPERS II is expected to provide performance similar to the unrolled versions of VIPERS, but in reality, the word-size result of VIPERS II is actually slightly slower due to the loop overheads. Despite the overhead, with the added computational power, the fracturable ALUs are able to provide better performance in terms of cycle counts when byte-size operands are used. It is also worth noting that the numbers shown in the table assume the worst-case scenario where no concurrency exists between scalar and vector operations. The concurrency is not captured by the simulation results

Architecture	VIPERS			VIPERS II		
Configuration	V4	V8	V16	V4	V8	V16
Fmax (MHz)	113.05	115.49	108.99	47.70	48.92	44.88
M9Ks	15	23	39	9	17	64
ALMs	6239	8190	12214	10520	15548	24737
DSP Blocks*	4	6	11	4	8	16
36-bit Mult.	4	8	16	-	-	-
18-bit Mult.	6	10	18	16	32	64
12-bit Mult.	-	1	1	-	-	-
9-bit Mult.	1	-	-	-	-	-

* One Stratix III DSP block fits 2×36b multipliers, 4×18b multipliers, 6×12b multipliers, or 8×9b multipliers.

Table 4.6: Resource usage of VIPERS II with different number of lanes

because the setup does not include the Nios II processor. The concurrent execution of scalar and vector instructions will actually help provide a slight improvement on these results.

The vector length of 16 is chosen for comparison between the two designs. However, since the fracturable processing units can operate on more elements, the VIPERS II pipeline is underutilized for byte and halfword operands. If the vector length is increased to 64 to fully utilize the pipeline, VIPERS II takes as little as 70 cycles per pixel to execute the median filter benchmark.

4.5 Resource Usage

The different configurations of the new VIPERS II architecture are compiled using Quartus II to measure their resource usage and compare them against the equivalent VIPERS designs. Both generations of VIPERS are compiled targeting the Stratix III EP3SL150F1152C3ES device that is found on the Altera DE3 development board, which will be used to implement the system in hardware.

Table 4.6 shows the resource usage of fully featured VIPERS II processors with a different number of vector lanes and how they compare to the original VIPERS. The resource usage results for VIPERS is slightly different from the results shown in [30] because the target

device for compilation is different. For fair comparison, VIPERS was configured with all features except the local memory, which VIPERS II does not offer. Furthermore, the processing width of the VIPERS is set to 32 bits in order to be capable of computing vectors at all three operand sizes.

As expected, VIPERS II consumes much more logic than the original VIPERS; it was pointed out in Section 3.2 that the address registers and the fracturable processing units are responsible for most of the increase in ALM and DSP block consumption. The fracturable multipliers in VIPERS II are implemented using four 18-bit multipliers, which take up an entire DSP block; on the other hand, the original VIPERS uses one 36-bit multiplier per lane, which consumes only half a DSP block. The VIPERS also used DSP blocks for its memory interface and multiply-accumulate (MAC) units, so the number of DSP blocks used is not exactly half of VIPERS II's usage.

To further examine the resource usage of VIPERS II, Table 4.7 breaks down the resource usage in VIPERS II to four major components: vector lanes, address generation, data alignment network, and vector memory. Note that the resource usage of the data alignment network shown here is higher than the values in Table 4.3 because it includes the tag generation logic and selectable delay network [5]. The difference of one in the number of M9Ks shown in Tables 4.6 and 4.7 comes from control register storage. With the exception of the vector length (VL) register, which is read on every execution, all other control registers are read only when required; therefore, the control registers can be implemented more efficiently with memory blocks instead of flip-flops.

The address generation logic implemented using flip-flops takes up 56% of the logic elements used in the V4 design. However, since the number of address registers do not change with the number of lanes, the resources consumed by address generation become less of a factor for large designs, accounting for only 25% of the overall usage of the V16 device.

On the other hand, as the table shows, the vector lanes and data alignment crossbar

Configuration	Component	ALMs	DSP Blocks	M9Ks
V4	Vector Lane	3549	4	0
	Address Gen.	5740	0	0
	Align Network	1019	0	0
	Vector Memory	0	0	8
V8	Vector Lane	7132	8	0
	Address Gen.	5897	0	0
	Align Network	2358	0	0
	Vector Memory	0	0	16
V16	Vector Lane	14205	16	0
	Address Gen.	6264	0	0
	Align Network	4260	0	31
	Vector Memory	0	0	32

Table 4.7: VIPERS II Resource Usage Breakdown

Architecture	VIPERS II - no MAC			VIPERS II - no MAC, no Multipliers		
Configuration	V4	V8	V16	V4	V8	V16
ALMs	10219	14972	24648	8355	11161	16920
DSP Blocks	4	8	16	0	0	0
M9Ks	9	17	33	9	17	33

Table 4.8: Resource usage of VIPERS II, without MAC/multipliers

network will grow proportionally as the number of vector lanes increase. The fracturable processing units account for most of the added ALM usage in the vector lanes, especially with the implementation of fracturable multipliers. The partial product method described in Section 3.6 requires multiplexers for input and output selection and adders to sum up the partial products, resulting in added resources. Another source of the increased ALM usage is the sum-reduction tree, which also involves adders, in the MAC unit. Table 4.8 quantifies the amount of resource savings achievable by removing the MAC unit and multipliers. The results reveal an average cost of 120 ALMs/lane for the MAC units, and 330 ALMs/lane for the multipliers.

Although the resource usage numbers are larger than anticipated, there are a few modifications that can be made to reduce resource consumption. One way is to reduce the number of address registers. As it was pointed out in Section 4.1, the auto-increment/decrement

feature allows the same benchmark to be implemented using fewer address registers. From our experience in developing benchmarks with the VIPERS II assembly, the number of address registers required to implement the benchmarks is no more than eight. Reducing the number of address registers from thirty-two down to sixteen would result in approximately 3000 less ALMs in each of the VIPERS II designs, making the smaller four-lane designs of VIPERS and VIPERS II comparable in area.

Another key resource to examine is the amount and the efficiency of memory usage. Table 4.6 shows that VIPERS II uses 6 fewer M9Ks than VIPERS in the V4 and V8 configurations, this difference is due to the UTIIe scalar processor and the VIPERS memory interface. In the V16 configuration, VIPERS II consumes more M9Ks than VIPERS, because 31 extra M9Ks are used in the alignment network as shift registers. If only the vector registers and vector memory are concerned, the number of M9Ks consumed are exactly the same for both architectures, but the register file in VIPERS has less capacity due to data duplication. Table 4.9 shows the efficiency of the memory in each design using the four lane configuration and VIPERS architecture with a $VPUW$ of 32 bits.

	VIPERS	VIPERS II		
Operand Size	ALL	32-bit	16-bit	8-bit
# M9Ks	8	8		
# Elem. in memory	1024	2048	4096	8192
# Elem./M9K	128	256	512	1024

Table 4.9: Memory Efficiency

The register files in VIPERS have a fixed width equal to $VPUW$; therefore, their capacity is also fixed regardless of operand size. With word-size operands, the register files are fully utilized, so the factor-of-two difference shown in the capacity is the result of data duplication in the original VIPERS. Having the advantage of storing data at their natural length, VIPERS II can provide up to 8 times more capacity than VIPERS when using byte-size data. These results demonstrate that by replacing physical register files with address registers, the VIPERS II architecture can utilize the on-chip memory resources much more

efficiently.

4.6 Scalability

This section discusses the scaling of VIPERS II and scaling-induced stress on FPGA resources.

As the VIPERS II design scales up to more lanes, the first resource bottleneck is the number of MLABs available on a given Stratix III device. This is due to the underutilization of the MLABs, which can be improved by adding some surrounding logic as mentioned in Section 3.3.1. For the purpose of scaling, the flag register depth is reduced to two to relieve the MLAB bottleneck.

The next resource bottleneck is the DSP blocks. The Stratix III family of devices offers up to 112 DSP blocks in a single FPGA, which can be used to implement a 64-lane instance of the VIPERS II.

The DSP blocks are used to implement the fracturable multipliers in the VIPERS II. To bypass the DSP bottleneck, VIPERS II can be configured to have no multipliers. In this mode, the amount of M9K memory becomes the bottleneck. Based on estimates, a 256-lane design should fit in the largest Stratix III chip, but due to lengthy compile times results are not yet available.

4.7 Design Verification

The top priority in the design of VIPERS II is its functionality, so extensive testing was performed at various stages of the design to verify the correctness of the design.

The major components of VIPERS II, such as the vector address registers, data alignment crossbar network, and the fracturable ALUs, have all been tested individually before being integrated into the final design. For the 4, 8, and 16-lane instances of VIPERS II, the DACN was verified using various combinations of offset and operand sizes, for vector

lengths up to 128 elements. The fracturable adders and multipliers were exhaustively tested for byte-size and halfword-size operands. For word-size operands, 100,000 random input test cases were verified for each of the signed and unsigned operations.

The fully integrated VIPERS II design was tested for each of the supported vector instructions at all three operand sizes. The design was also verified against a scalar Nios II processor in hardware using the three benchmark applications, which will be introduced in Chapter 5.

Chapter 5

Benchmark Results

This chapter presents the benchmark performance results of VIPERS II. First, the benchmark preparation is described for each of the three benchmarks. Next, both the simulated performance results from ModelSim and the hardware performance results as implemented on the Altera DE3 development board are presented.

5.1 Benchmark

Three benchmarks are chosen to compare the performance of VIPERS II and VIPERS:

- 16-tap FIR filter
- Image median filter
- Block matching motion estimation

Both the median filter and motion estimation were part of the original VIPERS benchmark suite. The AES encryption benchmark was replaced by the 16-tap FIR filter because the AES benchmark does not involve many load/store operations and is not expected to benefit much from the changes to the ISA. In the following sections, the operation and preparation of each benchmark is presented.

5.1.1 16-Tap Finite Impulse Response Filter

The 8-tap FIR filter was introduced earlier as an example to demonstrate the vector processing programming model. Once again, the operation of the FIR filter is described by the

equation:

$$y[n] = \sum_{k=0}^{7} x[n-k]h[k].$$

A scalar implementation of the 8-tap FIR filter would consists of an inner loop that performs eight multiply-add operations on the input data and filter coefficients, and an outer loop to iterate over the entire sample size. The inner loop can be vectorized to exploit data-level parallelism. Figure 2.1 shows the same FIR filter implemented with the vector assembly of the original VIPERS. The code uses the VMAC instruction to multiply the vector of filter coefficients with data samples, and the VCCZACC instruction sums up the multiplication results. To prepare for the next operation, the VUPSHIFT instruction is used to shift the data samples by one element.

Vector assembly of the FIR filter for VIPERS II uses the same VMAC/VCCZACC instruction pair to compute the results. However, since the vectors are stored in memory, an up-shifting of the vector would be the same as moving the vector to a new location and will take longer to execute. Therefore, a different approach must be taken to iterate over the entire range of data samples.

Figure 5.1 shows a 16-tap FIR filter benchmark implementation using the VIPERS II architecture. To maximize performance, the data samples and filter coefficients must be aligned in the vector memory. This is accomplished by having multiple copies of the filter coefficients inside the memory, one at each aligned location as shown in Figure 5.2. Once the coefficients are in place, the address registers will be incremented to step through all of the data samples as illustrated by the curved arrows in Figure 5.2.

5.1.2 Block Matching Motion Estimation

The block matching motion estimation calculates the displacement of an image block using the sum of absolute differences (SAD) metric:

```
; 16-tap FIR Benchmark
; VL = NTAPS = 16
; MemWidth = (n * NTAPS) * opSize, n is a postive integer

movi      r12, N            ;N = sample size
movi      r5, NTAPS
movi      r6, (n*NTAPS)
movi      r4, 1
vmstc     vinc1, r4         ;set vinc1 to 1
vmstc     vinc2, r6+1       ;set vinc2 to MemWidth+1
vmstc     VL, r5            ;VL = NTAPS

vmstc     va1, r1           ;sample data location
vmstc     va2, r2           ;filter coefficient location (aligned to va1)
vmstc     va3, r3           ;initial location of filter coefficients
vmstc     va5, r8
vmstc     vindex, zero      ;set vindex for vext.vs

mov       r4, r6
.L13:                       ;setup n*NTAPS copies of filter coefficients,
                            ;each offset by 1 element
vmovb     va2+, va3
addi      r4, r4, -1
bne       r4, zero, .L13

.L14:                       ;outer loop over all samples
vmstc     va2, r2           ;reset va2
mov       r4, r6

.L15:                       ;inner loop over MemWidth of elements
vmacb     va1+, va2+
vcczaccw  va5
vextw.vs  r7, va5
stw       r7, 0(r10)
addi      r10, r10, 4
addi      r4, r4, -1
bne       r4, zero, .L15

sub       r12, r12, r5
bne       r12, zero, .L14
```

Figure 5.1: FIR Filter Benchmark in VIPERS II Assembly

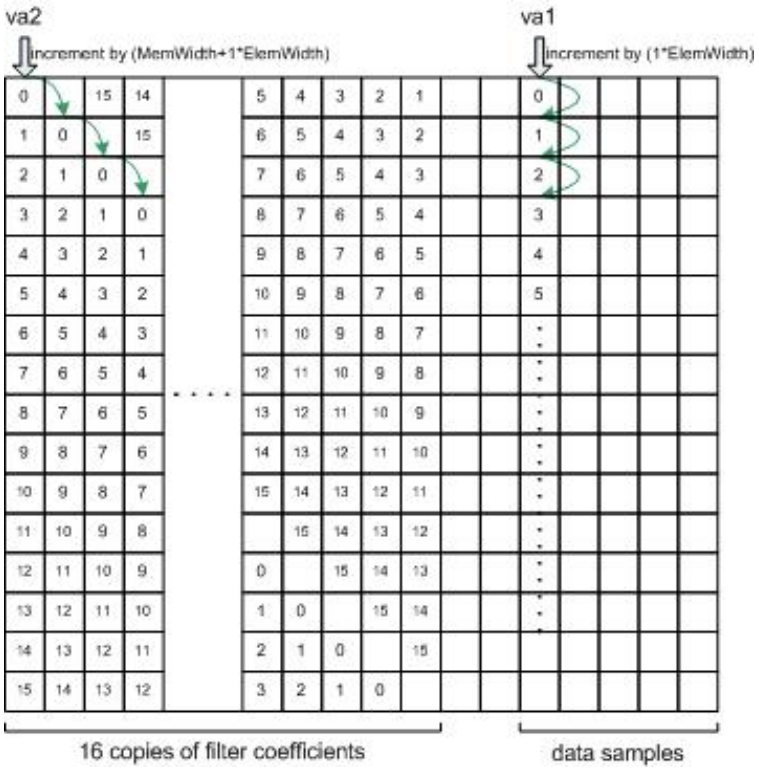


Figure 5.2: Vector Scratchpad Memory setup for 16-tap FIR Filter

```
for(n=−16; n<16; n++)
        for(m=−16; m<16; m++) {
                sad = 0;
                for(j=0; j<16; j++)
                        for(i=0; i<16; i++) {
                                temp = c[j][i] − s[n+j][m+i];
                                sad += ((temp >> 31) == 1)? (−temp) : temp;
                        }
                result[n][m] = sad;
        }
```

Figure 5.3: Motion estimation C code (Source: [29])

$$SAD[m,n] = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |c[i][j] - s[i+m][j+n]|.$$

Figure 5.3 shows the C implementation of the full search block matching algorithm (FSBMA), which computes the SADs within a 32×32 search window for a 16×16 reference image block. The two nested inner loops calculate the absolute difference of each pixel between two 16×16 images. The two outer loops iterate over the entire search window.

This algorithm can be vectorized by replacing the innermost loop with a sixteen-element vector. Figure 5.4 illustrates the FSBMA implemented with VIPERS assembly. The inner loop of the code computes the absolute difference of a row of pixels in the image block using the VABSDIFF instruction and iterates through all 16 rows. At the end of each loop, the VMAC/VCCZACC instructions are used to sum up all elements in the resulting vector. This is referred to as the one-window version of the motion estimation benchmark.

For VIPERS II, the motion estimation benchmark can be implemented using the same set of instructions, as shown in Figure 5.5. However, with the new architecture, the two load instructions at the beginning of the loop (L15) are eliminated, providing significant improvement in performance. This saves programmers the extra effort to unroll the loops for optimal performance.

```
        vmstc       va1, r1                 ; load va1

        vmstc       vbase2, r6              ; load y base
        vadd        v5, v0, v0              ; zero sum
.L15:                                       ; innermost loop over rows                5
        vld.b       v2, vbase1, vinc1       ; macroblock pixels, vinc1 = 16
        vld.b       v3, vbase2, vinc2       ; frame pixels, vinc2 = IMGWIDTH
        vabsdiff    v4, v2, v3
        vadd        v5, v5, v4              ; accumulate to sum
        addi        r2, r2, -1              ; j++                                     10
        bne         r2, zero, .L15          ; loop again if (j<15)

        vfld        vfmask1, vbase4, vinc4  ; load flag mask
        vmac.1      v6, v5                  ; accumulate across sum
        vcczacc     v6                      ; copy from accumulator                   15
        vmstc       VL, r9                  ; reset VL after vcczacc
        vext.vs     r3, v6                  ; extract result to scalar core
```

Figure 5.4: Motion Estimation Benchmark in VIPERS Assembly (Source: [29])

In the one-window version, the vector length is limited to 16 because of the size of the reference image. However, it is possible to increase the vector length to 32 by matching two copies of the reference block as illustrated by Figure 5.6. In order to support a vector length of 32, the two-window version of motion estimation must be implemented on VIPERS with at least 8 vector lanes. The same approach can be applied to improve performance in the VIPERS II architecture. However, since the maximum vector length for VIPERS II is more relaxed than for VIPERS, the two-window version can be executed with any number of lanes. Note that it is possible to implement the two-window motion estimation benchmark with a 4-lane instance of VIPERS by increasing the number of data elements in the register file from 4 to 8. However, this would double the number of M9Ks consumed by the design.

5.1.3 Image Median Filter

The median filter is commonly used in image processing to filter out unwanted noise in an image. It works by replacing each pixel with the median value of surrounding pixels. Figure 5.7 shows the C implementation of a median filter that searches for the median within a

```
; Motion Estimation Benchmark
; MemWidth = BLK_WIDTH

vmstc     vinc1, r3        ;set vinc1 = IMG_WIDTH
vmstc     vinc2, r4        ;set vinc2 = BLK_WIDTH
vmstc     VL, r4
vmstc     va1, r1          ;reference block location in image
vmstc     va2, r2          ;reference block location in memory
vmstc     va3, r1          ;absolute difference results
vmstc     va4, r1
vmstc     va5, r1          ;accumulated results
vmstc     va6, r1
vmstc     va7, r1
vmstc     vindex, zero
movi      r5, 16

.L9                        ;move reference block data
vmovb     va2+, va1+
addi      r5, r5, -1
bne       r5, zero, .L9

movi      r8, 16               ;loop variable for 16x16 block
movi      r7, IMG_LENGTH-15    ;loop variable for column

.L14                       ;outer loop over a column in image
vmstc     va1, r6          ;set va1 to point to starting location in image
vmstc     va2, r2          ;reset va2 to reference block
vxor      va5, va4, va4    ;reset va5 to 0, note that va4 and va5 points
                           ;to the same vector

.L15:                      ;inner loop over rows of block
vabsdiffb va3, va1+, va2+
vaddb     va5, va4, va3
addi      r8, r8, -1
bne       r8, zero, .L15

vmacb     va5, va6
vcczaccw  va7
vextw.vs  r10, va7
stw       r10, 0(r9)
addi      r9, r9, 4
add       r6, r6, r3       ;increment to next block in column
addi      r7, r7, -1
bne       r7, zero, .L14
```

Figure 5.5: Motion Estimation Benchmark in VIPERS II Assembly

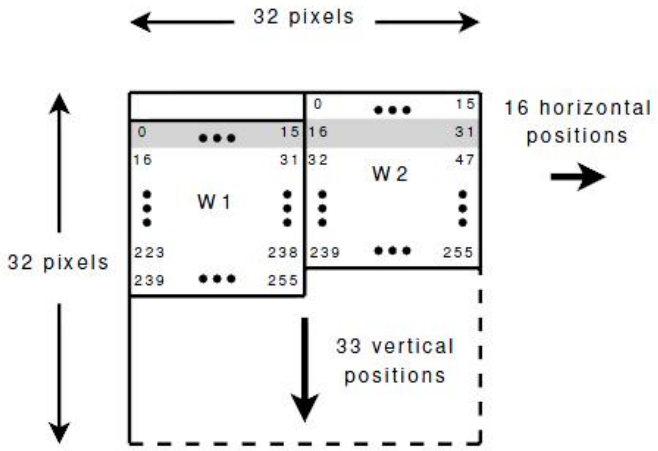


Figure 5.6: Two Window Motion Estimation (Source: [29])

```
for (i=0; i<=12; i++) {
        min = array[i];
        for (j=i; j<=24; j++) {
                if (array[j] < min) {
                        temp = min;
                        min = array[j];
                        array[j] = temp;
                }
        }
}
```

Figure 5.7: 5×5 Median Filter C code (Source: [29])

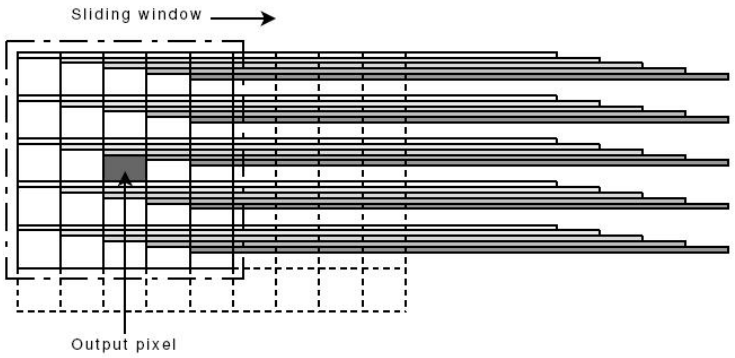


Figure 5.8: Vectorizing Median Filter Benchmark (Source: [29])

5×5 window using bubble sort.

Different from the FIR filter, the median filter benchmark can be vectorized to exploit outer-loop parallelism. Figure 5.8 illustrates how the vectorizing is achieved. A total of 25 vectors with length VL are created, each containing a row of pixel data in the image. The starting element of each vector is offset by one pixel, such that the first element of the 25 vectors would present a 5×5 image window.

The VIPERS II vector assembly of median filter is shown in Figure 5.9. The inner bubble sort algorithm is the same as the scalar implementation; however, every instruction computes VL number of results. Note that the median filter benchmark can use vector lengths up to the width of the image, making it a good match for the relaxed MVL available in VIPERS II.

```
; Median Filter - Bubble Sort Benchmark
; MemWidth = VL * opSize

movi      r4, MemWidth
movi      r20, 25

;move target data to desired location
vmstc     VL, r16
vmstc     va1, r11          ;set va1 to sample data location
vmstc     va2, r5           ;set va2 to image location
vmstc     va7, r7           ;set va7 to maximum vector location
vmstc     vinc1, r4         ;set vinc1 = MemWidth
vmstc     vinc2, r13        ;set vinc2 = ImgWidth

.L9:                        ;setup 25 vectors in memory
vmovb     va1+, va2+
addi      r20, r20, -1
bne       r20, zero, .L9

;sort 25 vectors to find median
movi      r7, 12
movi      r3, 25
addi      r6, r5, r4
vmstc     vinc2, r4         ;set vinc2 = MemWidth

.L14                        ;12 outer loops to get to array[12]
vmstc     va1, r5           ;set va1 to array[i]
vmstc     va3, r5           ;set va3 to array[i]
vmstc     va2, r6           ;set va2 to array[i+1]
movi      r2, 1

.L15                        ;inner loop comparing array[i] to array[i+n]
vmaxb     va7, va1, va2     ;va7, va1 are aligned
vminb     va3, va1, va2     ;imitate vminb va1, va1, va2
vmovab    va2+, va7
addi      r2, r2, 1
bge       r3, r2, .L15

add       r5, r5, r4        ;i++
add       r6, r6, r4
addi      r3, r3, -1
addi      r7, r7, -1
bne       r7, zero, .L14

vmstc     va1, r5           ;set va1 to array[12]
vmstc     va3, r5           ;set va3 to array[12]
vmstc     va2, r6           ;set va2 to array[13]
movi      r2, 12

.L16:                       ;last 12 comparisons to get the median
vminb     va3, va1, va2+
vminb     va1, va3, va2+    ;induce stall to ensure correctness
addi      r2, r2, -1
bne       r2, zero, .L16
```

Figure 5.9: Median Filter Benchmark in VIPERS II Assembly

5.2 Performance

This section presents the performance of the new VIPERS II architecture on the three benchmarks: FIR filter, motion estimation, and median filter. First, the performance result from ModelSim simulations are presented and compared against that of the original VIPERS. Then, the performance of the VIPERS II system in hardware is compared to the Nios II scalar processor.

5.2.1 Simulated Performance

Table 5.1 lists the simulated performance results for the three benchmarks in terms of instruction count and cycle count. Each benchmark was simulated for V4, V8, and V16 configurations and all simulation runs use byte-size operands.

The table shows results for several different versions of the benchmarks. For the motion estimation benchmark, two versions of the code were used: the one-window version using a vector length of 16 and the two-window version using a vector length of 32. Motion estimation in the original VIPERS used the one-window version for V4, because it has a vector length limit of 16, and the two-window version was used for V8 and V16; therefore, the two-window version must be introduced for fair comparison. The VIPERS II vector assembly codes do not have unrolled versions, because they are expected to achieve performance that are similar to the unrolled VIPERS code without unrolling. The table is structured to show the corresponding results in the two architectures side-by-side.

Table 5.1 shows the speedup of VIPERS II over VIPERS based on the simulated cycle counts. As expected, the new VIPERS II architecture demonstrates improved performance over the rolled versions of the benchmarks. When comparing the unrolled versions, the added computational power from the fracturable ALUs help overcome the loop overheads, so the VIPERS II actually delivered slightly better results than that of the unrolled code.

When factoring in the operating frequency, the speedup is not as good because the

Architecture	VIPERS			VIPERS II		
Configuration	V4	V8	V16	V4	V8	V16
Fmax (MHz)	113.05	115.49	108.99	47.70	48.92	44.88
Dynamic Instruction Count						
FIR-16	-	1315	1300	1048	996	988
Median Filter	93	47	23	84	42	21
Median Filter unrolled	44	22	11			
Motion Est. 1 Win	111840	-	-	77420	77420	77420
Motion Est. 1 Win unrolled	37568	-	-			
Motion Est. 2 Win	-	56232	56232	38716	38716	38716
Motion Est. 2 Win unrolled	-	37568	37568			
Simulated Cycle Count						
FIR-16	-	6128	5964	1392	1496	1612
Median Filter	496	256	141	176	90	46
Median Filter unrolled	189	95	48			
Motion Est. 1 Win	717120	-	-	152173	155245	159341
Motion Est. 1 Win unrolled	157792	-	-			
Motion Est. 2 Win	-	346203	269214	96581	81221	84293
Motion Est. 2 Win unrolled	-	88288	55328			
Speedup (cycle counts only)						
FIR-16				-	4.096	3.700
Median Filter				2.818	2.844	3.065
Median Filter unrolled				1.074	1.056	1.043
Motion Est. 1 Win				4.713	-	-
Motion Est. 1 Win unrolled				1.037	-	-
Motion Est. 2 Win				7.425	4.262	3.194
Motion Est. 2 Win unrolled				1.634	1.087	0.656
Speedup (including Fmax)						
FIR-16				-	1.735	1.523
Median Filter				1.189	1.205	1.262
Median Filter unrolled				0.453	0.447	0.430
Motion Est. 1 Win				1.988	-	-
Motion Est. 1 Win unrolled				0.438	-	-
Motion Est. 2 Win				3.133	1.806	1.315
Motion Est. 2 Win unrolled				0.689	0.460	0.270

Table 5.1: VIPERS II Simulated Performance Results

VIPERS II can only operate at about half the clock speed of the original VIPERS. This initial design of the VIPERS II focuses more on the functionality of the vector processor, so not much effort was put into operating at a high frequency. Based on the static timing analysis of Quartus, the critical path was found to be within the address generation logic. Since the vector scratchpad memory is operating at twice the frequency of the vector core, the address generation logic failed to keep up with demands. Introducing a deeper pipeline would increase the operating frequency of VIPERS II, but cannot be included in this thesis due to time constraints.

Comparing the different configurations of VIPERS II, the results show that the performance actually drops as the number of lanes increases for the FIR filter and motion estimation benchmarks. This unexpected behaviour is the result of the benchmarks' limitation on vector length. The 16-tap FIR filter only has a natural vector length of 16. The motion estimation benchmark has a natural vector length of 32 when scanning two windows at a time; however, that is as high as it can go because the search range is only a 32×32 block. As a result, although VIPERS II offers four times as much computational power at byte-size granularity, the vectors in these two benchmarks are not long enough to fill the pipeline and benefit from the large number vector lanes. Especially in the FIR filter benchmark, the extra overhead required to setup the vectors in a wider scratchpad memory actually degrades performance as the number of vector lanes increases.

5.2.2 Hardware Performance

The VIPERS II system was implemented in hardware using the Altera DE3 development board. The vector processor, the Nios II, and the DMA engine was implemented in the Stratix III EP3SL150F1152C3ES device, and the system-level connections between these components are accomplished with the Avalon fabric. An external DDR2 memory is connected as the main memory via the provided interface on the DE3 board. On the system level, the clock network generates an extra 7ns of delay, lowering the operating frequency

Benchmark	FIR-16		Motion Estimation		Median Filter	
	Time(ms)	Speedup	Time(ms)	Speedup	Time(ms)	Speedup
Nios II/f	1.39	1.00	5.23	1.00	94.43	1.00
V4+Nios II/f	0.24	5.76	1.31	4.00	7.88	11.98
V8+Nios II/f	0.24	5.83	1.11	4.70	3.98	23.76
V16+Nios II/f	0.25	5.49	1.91	2.75	2.01	47.07

Table 5.2: VIPERS II Performance in hardware

of VIPERS II down to 37.5 MHz.

Table 5.2 illustrates the hardware performance of the VIPERS II comparing against the performance of the fastest Nios II processor, operating at 166 MHz, on the DE3 board. The benchmarks are slightly different from what was used for the simulations in the previous section. The FIR filter uses 16 taps with 1024 data elements. The median filter benchmark times are for the computation of a 512×16 image. Only the one-window motion estimation benchmark is used, and the execution times are recorded for a 32×32 search range. Moreover, in order to ensure the correctness of the design, the VIPERS II results were verified using the Nios II outputs. Although the benchmarks were written for byte-size data, the operand size is set to half-word for the hardware runs to avoid any overflow issues and ensure correctness versus the Nios II output.

Figure 5.10 plots the speedup of VIPERS II over the Nios II. For the median filter benchmark, as expected, the speedup increases as the number of vector lanes increases. However, for the FIR filter and motion estimation benchmark, the limitation on vector length and setup overhead degrades the amount of achievable speedup as the number of vector lanes increases.

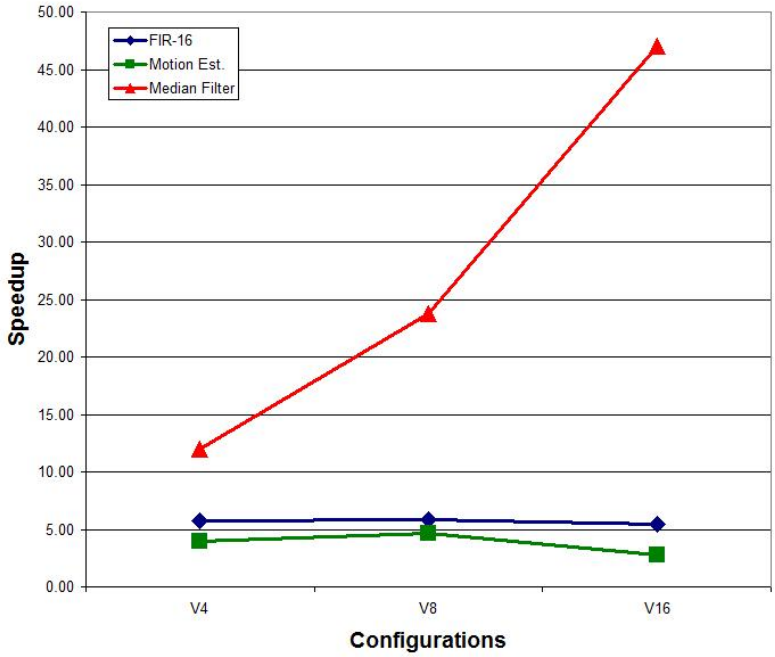


Figure 5.10: Speedup: VIPERS II vs. Nios II

Chapter 6

Conclusion

In [30, 31], the VIPERS soft vector processor demonstrated the potential of vector processing to improve performance of data-parallel embedded applications. The VIPERS architecture provides scalable performance and area without hardware design effort. However, it has a few shortcomings:

- load and store latencies
- data duplication
- high instruction count for unrolled code

This thesis proposed the new VIPERS II architecture, which aims to overcome these shortcomings of the original VIPERS.

The VIPERS II architecture takes advantage of the fast on-chip memory available in modern FPGAs to allow the vector processor to access data directly from memory. This is made possible by replacing the vector register file with address registers that point to locations in memory where vector data are stored. The removal of the vector register file eliminates the need for vector load/store operations, and it also reduces the number of copies of vector data in the on-chip memory down to one. With the load and store latencies removed, there is no need to spend the extra design effort in unrolling the applications, and the vector instruction code remains compact.

With the direct coupling of vector memory to the vector datapath, fracturable processing units are introduced to allow execution on different operand sizes. These fracturable processing units also provides twice or four times as much computational power with operand

sizes of 16 bits and 8 bits, respectively. By implementing the VIPERS II instruction set as an extended custom instruction of Nios II, the soft vector processor can utilize Nios II as its scalar core, providing a fully pipelined scalar unit with debug capability.

The VIPERS II is not yet tuned for high operating frequency, so it is only operating at about half the frequency of the original VIPERS. Assuming we can bring the operating frequency of VIPERS II to be similar to the original VIPERS, the new architecture can achieve 3x speedup over the rolled benchmarks of VIPERS at a cost of 2x the number of ALMs used. Compared against the unrolled code, the VIPERS II would provide similar performance without actually unrolling the loop. Moreover, the VIPERS II soft vector processor is able to provide up to 40x speedup over the scalar Nios II processor.

6.1 Future Work

The VIPERS II architecture presented in this thesis demonstrated its advantage over the original VIPERS architecture. However, there is still much fine tuning that can be done to further improve the design.

Operating Frequency

As mentioned earlier, the VIPERS II has a much lower operating frequency than the original VIPERS. This may be improved by a deeper, more balanced pipeline. The current critical path is in the address generation logic. Because the vector memory is operating at double the frequency of the vector core, the source address generation logic only has half a cycle to output the read address to vector memory. It would greatly enhance the performance of VIPERS II if the address generation logic can be pipelined to improve the operating frequency of the vector processor.

Data Alignment Crossbar Network

The control and tag generation for strided and indexed vector moves need to be added to the VIPERS II implementation to provide all the features of the original VIPERS. Implementing the DACN using the Omega network and its time-folded version, and comparing the area and performance against the current Benes network implementation may save resources. Moreover, if a generic control algorithm can be developed, the DACN can be utilized to support vector permutation instructions for some common permutations, such as perfect shuffle, unshuffle, and vector reversal.

Address Registers

Although the flip-flop implementation of address register provides the flexibility needed, the cost in resources is much higher than anticipated. It would be beneficial to devise a way to implement the address registers more efficiently, perhaps using on-chip memory. This might be difficult for the *base* address register because of all the surrounding logic. However, the *incr* registers and the *window* registers are both read-only for most vector operations. It should be relatively easy to implement these supplementary registers with on-chip memory. Finally, the current VIPERS II implements all 32 vector address registers; since applications rarely use more than 8 address registers, reducing the number of address registers in the architecture can also provide area savings.

MLAB Usage

As mentioned in Section 3.3.1, the MLABs are not fully utilized when used as 16×1 memory blocks, surrounding logic needs to be implemented to allow much longer vector flag registers. The MLABs in the Stratix III FPGA can be configured, forming memory blocks of size 16×8 up to 16×20. The wider memory can be read via a multiplexer to emulate a deeper memory block.

Design Verification

Although the individual building blocks of VIPERS II, such as the fracturable ALUs and the data alignment crossbar network, have been tested extensively, the fully integrated design has only been verified for the three benchmark applications. More extensive, full-scale testing have to be carried out with other benchmarks to ensure the functionality and validity of the VIPERS II architecture.

References

[1] H. Ahmadi and W. E. Denzel. A survey of modern high-performance switching techniques. *IEEE Journal on Selected Areas in Communications*, 7(7):1091–1103, 1989.

[2] Altera. Nios II. `http://www.altera.com/products/ip/processors/nios2/ni2-index.html`.

[3] Altera. Nios II custom instruction user guide. `http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf`.

[4] Altera. Stratix III device handbook, volume 1. `http://www.altera.com/literature/hb/stx3/stx3_siii5v1.pdf`, July 2009.

[5] Krste Asanovic. *Vector microprocessors*. PhD thesis, University of California, Berkeley, 1998.

[6] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 1(1):6–26, 2002.

[7] R. Banakar, S. Steinke, B. S Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78, 2002.

[8] V. E Benes. Optimal rearrangeable multistage connecting networks. *Bell Systems Technical Journal*, 43(7):1641–1656, 1964.

[9] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[10] S. H. Dhong, O. Takahashi, M. White, T. Asano, T. Nakazato, J. Silberman, A. Kawasumi, H. Yoshihara, and A. IBM. A 4.8 GHz fully pipelined embedded SRAM in the streaming processor of a CELL processor. In *2005 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 486–487, 2005.

[11] R. Ernst. Codesign of embedded systems: Status and trends. In *Readings in Hardware/Software Co-Design*, page 54, 2001.

[12] B. Flachs, S. Asano, S. H. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, et al. A streaming processing unit for a CELL processor. In *2005 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 134–135, 2005.

[13] Blair Fort, Davor Capalija, Zvonko G. Vranesic, and Stephen D. Brown. A multithreaded soft processor for SoPC area reduction. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 131–142. IEEE Computer Society, 2006.

[14] M. Kandemir, J. Ramanujam, M. J Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A Compiler-Based approach for dynamically managing Scratch-Pad memories in embedded systems. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):243–260, 2004.

[15] Christoforos Kozyrakis. *Scalable Vector Media Processors for Embedded Systems*. PhD thesis, University of California at Berkeley, May 2002. Technical Report UCB-CSD-02-1183.

[16] Christoforos E. Kozyrakis and David A. Patterson. Scalable vector processors for embedded systems. *IEEE Micro*, 23(6):36–45, 2003.

[17] K. Y. Lee. A new benes network control algorithm. *IEEE Trans. Comput.*, 36(6):768–772, 1987.

[18] Y. Lin, H. Lee, Y. Harel, M. Woh, S. Mahlke, T. Mudge, and K. Flautner. A system solution for High-Performance, low power SDR. In *SDR Technical Conference*, 2005.

[19] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. SODA: a low-power architecture for software radio. *SIGARCH Comput. Archit. News*, 34(2):89–101, 2006.

[20] Binu Mathew and Al Davis. An energy efficient high performance scratch-pad memory system. *Proceeding of 2004 International Design Automation Conference (DAC'04)*, 2004.

[21] P. R Panda, N. D Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):704, 2000.

[22] JongSoo Park, Sung-Boem Park, James D. Balfour, David Black-Schaffer, Christos Kozyrakis, and William J. Dally. Register pointer architecture for efficient embedded processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 600–605, Nice, France, 2007. EDA Consortium.

[23] D. Pham, E. Behnen, M. Bolliger, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, B. Le, Y. Masubuchi, et al. The design methodology and implementation of a first-generation CELL processor: a multi-core SoC. In *Proceedings of the IEEE 2005 Custom Integrated Circuits Conference*, pages 45–49, 2005.

[24] Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

[25] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, page 286, 2003.

[26] P. Yiannacouras, J. G Steffan, and J. Rose. Data parallel FPGA workloads: Software versus hardware. In *Proceedings of the 2009 International Conference on Field Programmable Logic and Applications.*, pages 51–58, Progue, Czech Republic, 2009.

[27] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 61–70, Atlanta, GA, USA, 2008. ACM.

[28] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Fine-grain performance scaling of soft vector processors. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 97–106, Grenoble, France, 2009. ACM.

[29] Jason Yu. Vector processing as a soft-cpu accelerator. Master's thesis, University of British Columbia, 2008.

[30] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. Vector processing as a soft processor accelerator. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):1–34, 2009.

[31] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core CPU accelerator. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 222–232, Monterey, California, USA, 2008. ACM.

Appendix A

VIPERS II Instruction Set Architecture (ISA)

A.1 Introduction

VIPERS II is a single-instruction-multiple-data (SIMD) array of processors organized into parallel datapaths called *vector lanes*. The number of lanes is the based on the vector scratchpad memory width (MemWidth). All lanes execute the same operation specified by a single vector instruction. Each vector lane hosts a complete copy of all functional units, and connects directly to its own 32-bit section of the vector scratchpad memory.

The VIPERS II ISA supports all of the original VIPERS instructions, except for the vupshift, which is replaced by vector moves in the new architecture. The differences of this ISA from the VIPERS ISA are:

- uses vector address registers instead of vector registers,
- auto post-increment/pre-decrement and circular buffer features,
- different instruction encoding,
- fracturable processing units,
- vector move instructions for data alignment,
- no VUPSHIFT instruction,
- no instruction modifies VL as a side effect.

Table A.1: List of configurable processor parameters

Parameter	Description	Typical
MemWidth	Scratchpad memory width (bits)	128, 256, 512
MACL	MAC chain length in words (0 is no MAC)	0,2,4,8
Vmult	Vector lane hardware multiplier	On/Off
Vmanip	Vector manipulation instructions (vector insert/extract)	On/Off

A.1.1 Configurable Architecture

This ISA specifies a set of features for an entire family of soft vector processors with varying performance and resource utilization. The ISA is intended to be implemented by an instance of the processor generated by a CPU generator using a few user-selectable configuration parameters. An implementation or instance of the architecture is not required to support all features of the specifcation. Table A.1 lists the configurable parameters and their descriptions, as well as typical values. These parameters will be referred to throughout the specification.

MemWidth is the the primary determinant of performance of the processor. It controls the number of parallel vector lanes and functional units that are available in the processor. The remaining parameters are used to enable or disable optional features of the processor.

A.2 Vector Register Set

The following sections define the registers in the VIPERS II architecture. Control registers and distributed accumulators will also be described.

A.2.1 Vector Address Registers

The architecture defines thirty-two 20-bit vector address registers directly addressable from the instruction opcode. The vector address registers store scratchpad memory locations of vectors to be accessed by the instruction. The size of 20 bits is sufficient to address a

scratchpad memory implemented using all of the on-chip memory blocks available in the EP3SL150 device that we are using. Auto post-increment, pre-decrement, and circular buffer features are implemented to lower loop overhead.

Vector address register `va0` is reserved for misalignment correction. It must be initialized to point to a free region within the scratchpad memory. This free region must be sufficiently large to hold an entire vector plus MemWidth extra bits.

A.2.2 Vector Scalar Registers

Scalar registers are read for vector-scalar instructions which require both a vector and a scalar operand. Scalar registers of the vector processor are located in the scalar core, which are implemented within Altera's Nios II soft processor. The Nios II ISA defines thirty-two 32-bit scalar registers. Scalar register values can also be transferred to and from vector registers or vector control registers using the `vins.vs`, `vext.vs`, `vmstc`, `vmcts` instructions, respectively.

A.2.3 Vector Flag Registers

The architecture defines 8 vector flag registers. The flag registers are written to by comparison instructions and by flag logical instructions. Most instructions in the instruction set support conditional execution using one of two vector masks, determined by a mask bit in the instruction opcodes. The vector masks are stored in the first two vector flag registers. A value of 1 in a vector lane's mask register will disable the lane processor from executing the instruction. Table A.2 shows a complete list of flag registers.

A.2.4 Vector Control Registers

Table A.3 lists the vector control registers in the soft vector processor.

The `vindex` control register holds the vector element index that controls the operation of vector insert and extract instructions. The register is writeable. For vector-scalar in-

Table A.2: List of vector flag registers

Hardware Name	Software Name	Contents
$vf0	vfmask0	Primary mask; 1 disables lane operation
$vf1	vfmask1	Secondary mask; 1 to disables lane operation
$vf2	vfgr0	General purpose
...	...	...
$vf6	vfgr4	General purpose
$vf7		Integer overflow

Table A.3: List of control registers

Hardware Name	Software Name	Description
$vc0	VL	Vector length
$vc1	vindex	Element index for insert (vins) and extract (vext)
...	...	...
$vc13	vmissrc	Num. of occurrences for misaligned sources
$vc14	vmisdsta	Num. of occurrences for misaligned src A to dest.
$vc15	vmisdstb	Num. of occurrences for misaligned src B to dest.
$vc16	vstride0	Stride register 0
...	...	...
$vc31	vstride15	Stride register 15

sert/extract, `vindex` specifies which data element within the vector register will be written to/read from by the scalar core. For vector-vector insert/extract, `vindex` specifies the index of the starting data element for the vector insert/extract operation.

The `vmissrc`, `vmisdsta`, and `vmisdstb` control registers count the number of occurences for each type of misalignment. These are intended to aid the user when performance tuning software.

A.2.5 Vector Address Increment/Decrement Registers

The architecture defines 32 20-bit vector address increment/decrement registers (vi0–vi31). These registers hold the auto post-increment or pre-decrement amount of their corresponding vector address registers.

A.2.6 Vector Window Registers

The architecture defines 32 20-bit vector window registers (vw0–vw31). These registers hold the window size of their corresponding vector address registers, when used in circular buffer mode. A vector window register with value of zero indicates the corresponding address register is not in circular buffer mode. Setting a bit to one in the window register prohibits the corresponding address bit from being updated by a post-increment or pre-decrement operation.

A.2.7 Vector Sum Reduction

The fracturable multiplier described in the previous section uses the DSP blocks in individual multiplier mode, so the accumulator and sum reduction hardware can not be utilized for the multiply-accumulate feature. Figure A.1 shows the sum-accumulate unit, which consists of a sum-reduction tree and an accumulator, designed for each of the vector lanes. For byte-size data, the sum-reduction starts at the first level of adders; for halfword-size operands, the multiplier results enters the tree at the second level. At the output of the

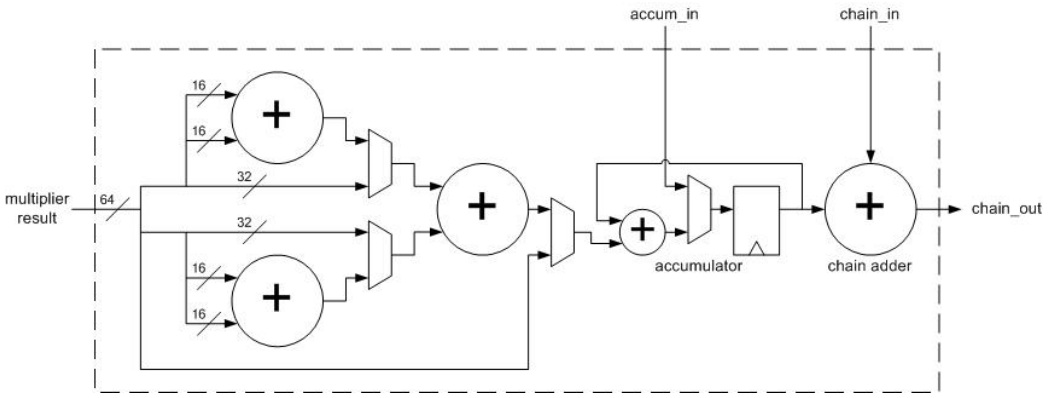


Figure A.1: Sum-Reduction Tree and Accumulator

adder tree, a 32-bit accumulator is used to store the sum-reduced results. The chain adder in the sum-accumulate unit is used to sum reduce the results across vector lanes. Unlike the original VIPERS, the VIPERS II's implementation of the multiply-accumulate operation reuses the multipliers, making it more efficient in terms of resource usage.

The compress copy operation (VCCACC/VCCZACC) is accomplished by summing up all the accumulator results by connecting multiple sum-accumulate units together. In the original VIPERS, the compress copy operation may require multiple iterations of the VMAC/VCCACC instructions to produce a single result. Since the sum-accumulate units of VIPERS II do not use the fixed DSP blocks, internal signals are fully accessible and are utilized to implement a simpler compress copy operation.

Figure A.2 illustrates how the sum-accumulate units are connected in a 4-lane instance of VIPERS II, with MAC chain length (MACL) of 2. The MACL defines how many sum-accumulate units are connected in series via the chain adder. With a MACL of 2, the VCCACC instruction can sum-reduce the accumulator results to a single 32-bit result in 2 cycles. In the first cycle, the accumulator results of sum-accumulate units 0 and 1 is summed up by the chain adder and stored in the accumulator of sum_accum0, and the accumulator results of sum-accumulate units 2 and 3 is summed and stored in sum_accum1. In the second cycle, the results generated in the previous cycle is summed up and written

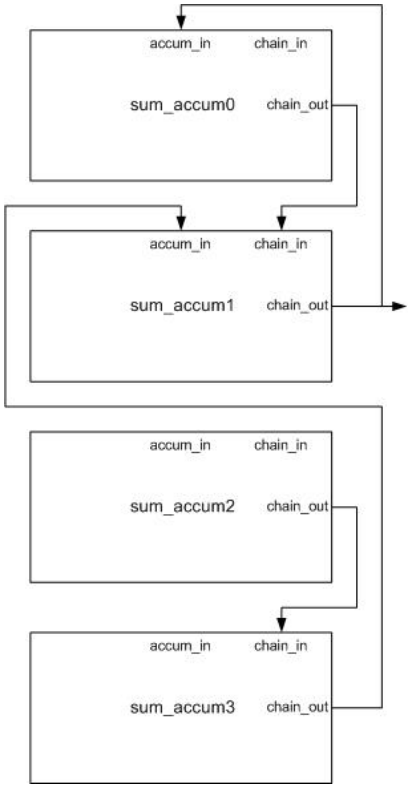


Figure A.2: MAC Chain

back in to the vector scratchpad memory. When the number of lanes is less than or equal to MACL, the sum-reduction completes in one cycle. If NLane is greater than MACL, the sum-reduction will require $\log_{MACL} NLane$ (round up) cycles to finish.

A.3 Instruction Set

The following sections describe in detail the instruction set of VIPERS II soft vector processor, and different variations of the vector instructions.

A.3.1 Data Types

The VIPERS II processor supports data widths of 32-bit words, 16-bit halfwords, and 8-bit bytes, and both signed and unsigned data types. The data width of an instruction is defined

by the opcode.

A.3.2 Data Alignment

Since vector lanes are connected directly to the vector memory, data to be operated on must be aligned. When misalignment occurs, vector move instructions are used to move the data into alignment.

The instruction set supports three types of vector move:

1. Offset move
2. Strided move (not yet implemented)
3. Indexed move (not yet implemented)

A.3.3 Flag Register Use

Almost all instructions can specify one of two vector mask registers in the opcode to use as an execution mask. By default, `vfmask0` is used as the vector mask. Writing a value of 1 into the mask register will cause that lane to be disabled for operations that use the mask. Some instructions, such as flag logical operations, are not masked.

A.3.4 Instructions

The instruction set includes the following categories of instructions:

1. Vector Integer Arithmetic Instructions
2. Vector Logical Instructions
3. Vector Flag Processing Instructions
4. Vector Manipulation Intructions
5. Vector Move Instructions

A.4 Instruction Set Reference

The following sections list the complete instruction set for each of the instruction type. Table A.4 describes the possible qualifiers in the assembly mnemonic of each instruction.

Table A.4: Instruction qualifiers

Qualifier	Meaning	Notes
op.vv *op*.vs *op*.sv	Vector-vector Vector-scalar Scalar-vector	Vector arithmetic and logical instructions may take one source operand from a scalar register. A vector-vector operation takes two vector source operands; a vector-scalar operation takes its second operand from the scalar register file; a scalar-vector operation takes its first operand from the scalar register file. The .sv instruction type is provided to support non-commutative operations.
op.1	Use vfmask1 as the mask	By default, the vector mask is taken from vfmask0. This qualifier selects vfmask1 as the vector mask.

In the following tables, instructions in italics are not yet implemented. The term 'VP' stands for 'virtual processor', which represents the set of functional units for a lane. Note that the width of the VP depends upon the operand size, either byte, halfword, or word.

A.4.1 Integer Instructions

Name	*Mnemonic*	*Syntax*	*Summary*

Absolute Value	`vabsb`
	`vabsh`
	`vabsw`
		`.vv[.1] vD, vA`	Each unmasked VP writes into vD the absolute value of vA.

Absolute Difference	`vabsdiffb`
	`vabsdiffh`
	`vabsdiffw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, vB`
			Each unmasked VP writes into vD the absolute difference of vA and vB/rS.

Add	`vaddb`
	`vaddh`
	`vaddw`
	`vaddub`
	`vadduh`
	`vadduw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP writes into vD the signed/unsigned integer sum of vA and vB/rS.

Subtract	`vsubb`
	`vsubh`
	`vsubw`
	`vsubub`
	`vsubuh`
	`vsubuw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the signed/unsigned integer result of vA/rS minus vB/rS.

Multiply High	`vmulhib`
	`vmulhih`
	`vmulhiw`
	`vmulhiub`
	`vmulhiuh`
	`vmulhiuw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP multiplies vA and vB/rS and stores the upper half of the signed/unsigned product into vD.

Multiply Low	`vmullob`
	`vmulloh`
	`vmullow`
	`vmulloub`
	`vmullouh`
	`vmullow`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP multiplies vA and vB/rS and stores the lower half of the signed/unsigned product into vD.

Integer Divide	*vdiv*
	vdivu
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the signed/unsigned result of vA/rS divided by vB/rS, where at least one source is a vector.

Shift Right Arithmetic	`vsrab`
	`vsrah`
	`vsraw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the result of arithmetic right shifting vB/rS by the number of bits specified in vA/rS, where at least one source is a vector.

Name	*Mnemonic*	*Syntax*	*Summary*

Minimum	`vminb`
	`vminh`
	`vminw`
	`vminub`
	`vminuh`
	`vminuw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP writes into vD the minimum of vA and vB/rS.

Maximum	`vmaxb`
	`vmaxh`
	`vmaxw`
	`vmaxub`
	`vmaxuh`
	`vmaxuw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP writes into vD the maximum of vA and vB/rS.

Compare Equal, Compare Not Equal	`vcmpeb`
	`vcmpeh`
	`vcmpew`
	`vcmpneb`
	`vcmpneh`
	`vcmpnew`
		`.vv[.1] vF, vA, vB`
		`.vs[.1] vF, vA, rS`
			Each unmasked VP writes into vF the boolean result of comparing vA and vB/rS.

Compare Less Than	`vcmpltb`
	`vcmplth`
	`vcmpltw`
	`vcmpltub`
	`vcmpltuh`
	`vcmpltuw`
		`.vv[.1] vF, vA, vB`
		`.vs[.1] vF, vA, rS`
		`.sv[.1] vF, rS, vB`
			Each unmasked VP writes into vF the boolean result of whether vA/rS is less than vB/rS, where at least one source is a vector.

Compare Less Than or Equal	`vcmpleb`
	`vcmpleh`
	`vcmplew`
	`vcmpleub`
	`vcmpleuh`
	`vcmpleuw`
		`.vv[.1] vF, vA, vB`
		`.vs[.1] vF, vA, rS`
		`.sv[.1] vF, rS, vB`
			Each unmasked VP writes into vF the boolean result of whether vA/rS is less than or equal to vB/rS, where at least one source is a vector.

Multiply Accumulate	`vmacb`
	`vmach`
	`vmacw`
	`vmacub`
	`vmacuh`
	`vmacuw`
		`.vv[.1] vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP calculates the product of vA and vB/rS. The products of vector elements are summed, and the summation results are added to the distributed accumulators.

Name	*Mnemonic*	*Syntax*	*Summary*
Compress Copy from Accumulator	`vccaccw`	`vD`	The contents of the distributed accumulators are reduced by a chain adder, and the result written into vD. To avoid result overflow, the instruction only has word-size version, and only the bottom 32 bits of the result are written. This instruction is not masked and remaining elements of vD are not modified.
Compress Copy and Zero Accumulator	`vczaccw`	`vD`	The operation is identical to `vccacc`, except the distributed accumulators are zeroed as a side effect.

A.4.2 Logical Instructions

Name	Mnemonic	Syntax	Summary

And	`vandb`
	`vandh`
	`vandw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP writes into vD the logical AND of vA and vB/rS.

Or	`vorb`
	`vorh`
	`vorw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP writes into vD the logical OR of vA and vB/rS.

Xor	`vxorb`
	`vxorh`
	`vxorw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
			Each unmasked VP writes into vD the logical XOR of vA and vB/rS.

Shift Left Logical	`vsllb`
	`vsllh`
	`vsllw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the result of logical left shifting vB/rS by the number of bits specified in vA/rS, where at least one source is a vector.

Shift Right Logical	`vsrlb`
	`vsrlh`
	`vsrlw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the result of logical right shifting vB/rS by the number of bits specified in vA/rS, where at least one source is a vector.

Rotate Right	`vrotb`
	`vroth`
	`vrotw`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each unmasked VP writes into vD the result of rotating vA/rS right by the number of bits specified in vB/rS, where at least one source is a vector.

A.4.3 Vector Move Instructions

Name	*Mnemonic*	*Syntax*	*Summary*

Offset Move	`vmovb`
	`vmovh`
	`vmovw`
		`[.1] vD, vA`	The VPs perform a contiguous vector move from offset locations vA to vD. This operation moves unaligned data into alignment via the DACN. In cases where vA is aligned to vD, the DACN is bypassed to avoid adding unecessary delay to the instruction. The size of each element in memory is given by the opcode.

Constant Stride Move	*vsmovb* *vsmovh* *vsmovw*	`[.1] vD, vA, vstride`	The VPs perform a strided vector move from memory location in vA to vD contiguously. The signed stride is given by vstride control register (default is vstride0). The stride is in terms of elements, not in terms of bytes. The size of each element in memory is given by the opcode.
Constant Stride Scatter	*vssctb* *vsscth* *vssctw*	`[.1] vD, vA, vstride`	The VPs perform a contiguous vector move from location vA to strided locations starting at vD. The signed stride is given by the vstride control register (default is vstride0). The stride is in terms of elements, not in terms of bytes. The width of each element in memory is given by the opcode.

Name	*Mnemonic*	*Syntax*	*Summary*
Indexed Move	*vxmovb* *vxmovh* *vxmovw*	`[.1] vD, vA, vB`	The VPs perform an indexed-vector move from memory location vA to vD contiguously. The signed offsets are given by vB and are in units of bytes, not in units of elements. The effective addresses must be aligned to the width of the data in memory. The size of each element in memory is given by the opcode.
Indexed Scatter	*vxsctb* *vxscth* *vxsctw*	`[.1] vD, vA, vB`	The VPs perform an indexed-vector scatter of vA. The base address is given by vbase (default vbase0). The signed offsets are given by vB. The offsets are in units of bytes, not in units of elements. The effective addresses must be aligned to the width of the data being accessed.

A.4.4 Vector Manipulation Instructions

Name	Mnemonic	Syntax	Summary

Merge	`vmergeb`
	`vmergeh`
	`vmergew`
		`.vv[.1] vD, vA, vB`
		`.vs[.1] vD, vA, rS`
		`.sv[.1] vD, rS, vB`
			Each VP copies into vD either vA/rS if the mask is 0, or vB/rS if the mask is 1. At least one source is a vector. Scalar sources are truncated to the VP width.

Vector Insert	`vinsb`
	`vinsh`
	`vinsw`
		`.vv vD, vA`	The leading portion of vA is inserted into vD. vD must be different from vA. Leading and trailing entries of vD are not touched. The lower $\log_2$(MemWidthByte) bits of vector control register vc_{vindex} specifies the starting position in vD. The vector length specifies the number of elements to transfer. This instruction is not masked.

Vector Extract	`vextb`
	`vexth`
	`vextw`
		`.vv vD, vA`	A portion of vA is extracted to the front of vD. vD must be different from vA. Trailing entries of vD are not touched. The lower $\log_2$(MemWidthByte) bits of vector control register vc_{vindex} specifies the starting position in vD. The vector length specifies the number of elements to transfer. This instruction is not masked.

Scalar Insert	`vinsb`
	`vinsh`
	`vinsw`
		`.vs vD, rS`	The contents of rS are written into vD at position vc_{vindex}. The lower $\log_2$(MemWidthByte) bits of vc_{vindex} are used. This instruction is not masked and does not use vector length.

Scalar Extract	`vextb`
	`vexth`
	`vextw`
	`vextub`
	`vextuh`
	`vextuw`
		`.vs rS, vA`	Element vc_{vindex} of vA is written into rS. The lower $\log_2$(MemWidthByte) bits of vc_{index} are used to determine the element in vA to be extracted. The value is sign/zero-extended. This instruction is not masked and does not use vector length.

Compress	*vcomp*	`[.1] vD, vA`	All unmasked elements of vA are concatenated to form a vector whose length is the population count of the ask (subject to vector length). The result is placed at the front of vD, leaving trailing elements untouched. vD must be different from vA.
Expand	*vexpand*	`[.1] vD, vA`	The first n elements of vA are written into the unmasked positions of vD, where n is the population count of the mask (subject to vector length). Masked positions in vD are not touched. vD must be different from vA.

A.4.5 Vector Flag Processing Instructions

Name	Mnemonic	Syntax	Summary
Scalar Flag Insert	`vfins`	`.vs vF, rS`	The boolean value of rS is written into vF at position vc_{vindex}. The lower vc_{logmvl} bits of vc_{vindex} are used. This instruction is not masked and does not use vector length.
And	`vfand`	`.vv vFD, vFA, vFB` `.vs vFD, vFA, rS`	Each VP writes into vFD the logical AND of vFA and vFB/rS. This instruction is not masked, but is subject to vector length.
Or	`vfor`	`.vv vFD, vFA, vFB` `.vs vFD, vFA, rS`	Each VP writes into vFD the logical OR of vFA and vFB/rS. This instruction is not masked, but is subject to vector length.
Xor	`vfxor`	`.vv vFD, vFA, vFB` `.vs vFD, vFA, rS`	Each VP writes into vFD the logical XOR of vFA and vFB/rS. This instruction is not masked, but is subject to vector length.
Nor	`vfnor`	`.vv vFD, vFA, vFB` `.vs vFD, vFA, rS`	Each VP writes into vFD the logical NOR of vFA and vFB/rS. This instruction is not masked, but is subject to vector length.
Clear	`vfclr`	`vFD`	Each VP writes zero into vFD. This instruction is not masked, but is subject to vector length.
Set	`vfset`	`vFD`	Each VP writes one into vFD. This instruction is not masked, but is subject to vector length.
Flag Load	`vfld`	`vF, vA`	The VPs perform a contiguous vector flag load from memory location in vA into vF. The LSB of each byte-size data is loaded into vF. The address in vA must be aligned to *MemWidth*. This instruction is not masked.

Flag Store	`vfst`	`vD, vF`	The VPs perform a contiguous vector flag store of vF to memory location in vD. The flag data in vF are written into the LSB of each byte in vD. The address in vD must be aligned to *MemWidth*. This instruction is not masked.
Population Count	*vfpop*	`rS, vF`	The population count of vF is placed in rS. This instruction is not masked.
Find First One	*vfff1*	`rS, vF`	The location of the first set bit of vF is placed in rS. This instruction is not masked. If there is no set bit in vF, then the vector length is placed in rS.
Find Last One	*vffl1*	`rS, vF`	The location of the last set bit of vF is placed in rS. The instruction is not masked. If there is no set bit in vF, then the vector length is placed in rS.
Set Before First One	*vfsetbf*	`vFD, vFA`	Register vFD is filled with ones up to and not including the first set bit in vFA. Remaining positions in vF are cleared. If vFA contains no set bits, vFD is set to all ones. This instruction is not masked.
Set Including First One	*vfsetif*	`vFD, vFA`	Register vFD is filled with ones up to and including the first set bit in vFA. Remaining positions in vF are cleared. If vFA contains no set bits, vFD is set to all ones. This instruction is not masked.
Set Only First One	*vfsetof*	`vFD, vFA`	Register vFD is filled with zeros except for the position of the first set bit in vFA. If vFA contains no set bits, vFD is set to all zeros. This instruction is not masked.

A.4.6 Miscellaneous Instructions

Name	Mnemonic	Syntax	Summary
Move Scalar to Control	`vmstc`	`vc, rS`	Register rS is copied to vc. Writing vc_{vpw} changes vc_{mvl}, vc_{logmvl} as a side effect.
Move Control to Scalar	`vmcts`	`rS, vc`	Register vc is copied to rS.

A.5 Special Cases

This section describes the behaviour of some special cases of the VIPERS II instruction. The general VIPERS II syntax is:

$$vop \qquad (-)vaD(+),\ (-)vaA(+),\ (-)vaB(+)$$

where vop is the vector instruction, vaA and vaB are the address registers pointing to the source vectors, and vaD is the address register pointing to the destination vector.

A.5.1 Three Different Locations

The typical case where all three vectors are in different locations in the memory, there is no restriction on the auto-increment/decrement combinations that may be applied to the operands.

$$vaddw.vv \qquad -va1,\ -va2,\ va3+$$

A.5.2 Destination Overwrite

In cases where the destination location is the same as one or both of the source locations:

$$vaddw.vv \qquad va1+,\ -va1,\ va2+ \quad (1)$$

$$vaddw.vv \qquad va1+,\ va1,\ va1 \quad (2)$$

The overwriting of data in the original source location is allowed, but the user must assign a different address register pointing to the same location as the destination address register. As explained in Section 3.3.2, the base register value changes throughout the execution of an instruction, so another destination register must be used to ensure data is written to the correct location in the scratchpad memory. With a new destination address register assigned, any combination of auto-increment/decrement can be realized. For correct operation, VIPERS II instructions (1) and (2), must be rewrote as (3) and (4), respectively.

$$vaddw.vv \qquad va3+,\ -va1,\ va2+ \quad (3)$$

$$vaddw.vv \qquad va3+,\ va1,\ va1 \quad (4)$$

A.5.3 Source Reused

For cases where the two source locations are the same:

$$vaddw.vv \qquad va2,\ va1+,\ va1+ \quad (5)$$

$$vaddw.vv \qquad va2,\ va1,\ -va1 \quad (6)$$

$$vaddw.vv \qquad va2,\ -va1,\ va1+ \quad (7)$$

These instructions can be carried out and the reading of input vectors will be correct; however, only the auto-increment/decrement on source A is performed. Therefore, if the auto-increment/decrement designators are the same for both sources as in (5), the instruction can be executed correctly as is. If the sources have different increment/decrement requests, a different address register must be used for referencing the source to achieve the desired increment/decrement sequence. For example, VIPERS II instructions (6) and (7), must be rewrote as (8) and (9), respectively, for correct increment/decrement.

$$vaddw.vv \qquad va2,\ va1,\ -va3 \quad (8)$$

$$vaddw.vv \qquad va2,\ -va1,\ va3+ \quad (9)$$

A.6 Nios II Custom Instruction Formats

The VIPERS II instructions are processed as custom instructions of Nios II. The format of the Nios II custom instruction is shown below.

The 6-bit opcode field contains the Nios II custom instruction opcode, 0x32. The 8-bit custom instruction field is used to define up to 256 different operations. Currently,

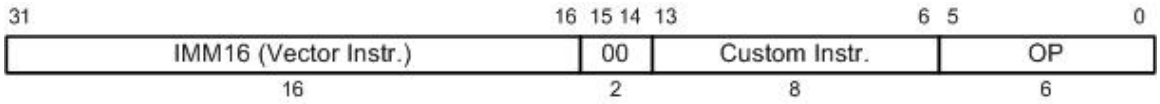


Table A.11: Nios II Custom Instructions

Custom Instr. Field	Description
0x00	Read from DMA control registers
0x01	Write the least significant 16-bit of VIPERS II instruction
0x02	Write the most significant 16-bit of VIPERS II instruction
0x03	Write to DMA control registers
0x04	Read from vector to scalar queue
0x05	Write to scalar to vector queue

custom instructions are defined for vector instruction transfer and DMA control register read/write as listed in Table A.11. The 16-bit immediate field is used to dispatch the vector instruction to the vector core, each vector instruction is transferred using two Nios II custom instructions, with the least significant 16-bit sent first.

A.7 VIPERS II Instruction Formats

A 32-bit VIPERS II instruction is formed from the execution of two Nios II custom instructions. The encoding of these VIPERS II instructions is presented in this section.

VIPERS II instructions can be divided in to three classes: vector-vector instructions, vector-scalar instructions, and vector flag instructions. The vector-vector and vector-scalar instructions are defined for each of the three operand sizes. The vector instruction class is defined by using 6-bit opcodes from the unused/reserved Nios II opcode space – this maintains compatibility with the Nios II encoding should the need arises to directly mix instruction streams. Table A.12 lists the Nios II opcode values used by the soft vector processor instructions.

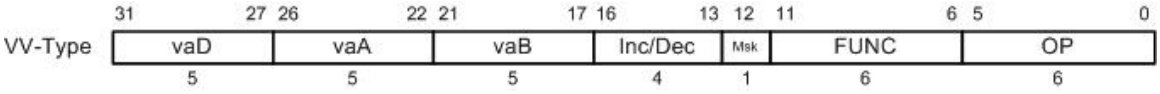
Table A.12: Nios II Opcode Usage

Nios II Opcode	Vector Instruction Type
0x19	Vector-Vector instructions (Byte)
0x1D	Vector-Vector instructions (Halfword)
0x1F	Vector-Vector instructions (Word)
0x39	Vector-Scalar instructions (Byte)
0x3D	Vector-Scalar instructions (Halfword)
0x3F	Vector-Scalar instructions (Word)
0x3E	Vector flag, misc instructions

A.7.1 Vector-Vector and Vector-Scalar Instructions

The vector-vector format (VV-type) covers most vector arithmetic, logical, and vector processing instructions. It specifies three vector address registers, a 4-bit increment/decrement indicator field, 1-bit mask select, a 6-bit vector opcode FUNC, and the 6-bit Nios II opcode shown in Table A.12. Instructions that take only one source operand use the vaA field.

Six bits are required to define the auot-increment/decrement for all three operands in the instruction, which makes the VIPERS II instruction 34-bit. To keep the instruction at 32-bit, the common auto-increment/decrement combinations are encoded using 4 bits as shown in Table A.13.

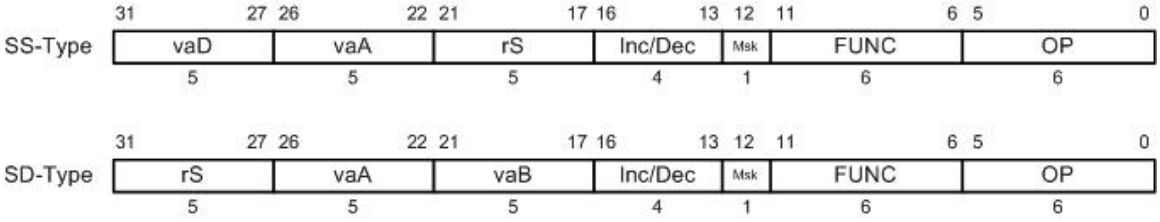


Vector-scalar instructions that take one scalar register operand have two formats, depending on whether the scalar register is the source (SS-Type) or destination (SD-Type) of the operation.

Table A.14 lists which instructions use a scalar register as a source and as a destination.

Table A.13: Increment/Decrement Encoding

Inc/Dec Field	Destination	Source A	Source B
0x0	-	-	-
0x1	-	-	inc
0x2	-	inc	-
0x3	-	inc	inc
0x4	inc	-	-
0x5	inc	-	inc
0x6	inc	inc	-
0x7	inc	inc	inc
0x8	-	-	dec
0x9	-	dec	-
0xA	-	dec	dec
0xB	dec	-	-
0xC	-	inc	dec
0xD	-	dec	inc
0xE	inc	dec	inc
0xF	dec	inc	inc



A.7.2 Vector Move Instructions

The vector move instruction has the same format as the vector-vector instruction with a single input operand. The strided and indexed vector moves have their own instruction format: VMS and VMX-type, respectively.

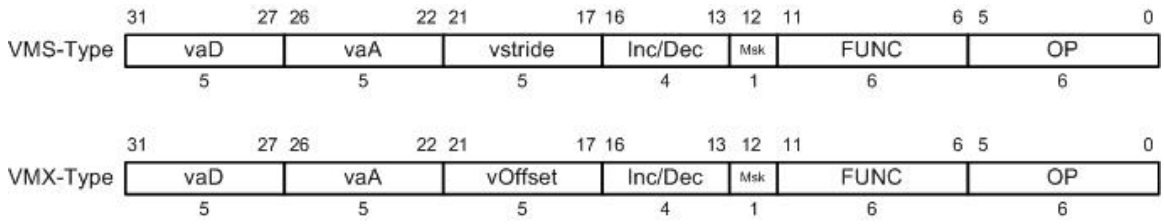


Table A.14: Scalar register usage as source or destination register

Instruction	Scalar register usage
op.vs	Source
op.sv	Source
vins.vs	Source
vext.vs	Destination
vmstc	Source
vmcts	Destination

Strided vector move instructions use the vaB field to indicate which control register (vstride0–15) contains the stride information. Indexed move instructions use the vaB field to indicate which vector holds the indexes.

A.7.3 Control Register Instructions

The address registers, increment/decrement registers, and window registers are accessed as control registers via the VMSTC/VMCTS instructions using the Nios II opcode 0x3E. Therefore, a 7-bit field, vControl, is required in the VMSTC/VMCTS instruction format to access the 128 control registers. The encoding of vControl is shown in Table A.15.

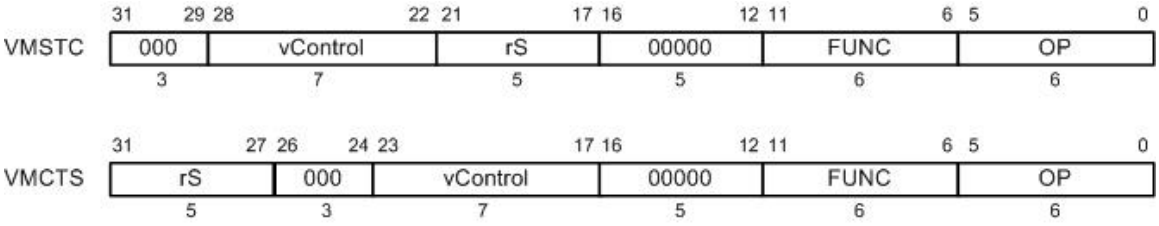


Table A.15: vControl Field Encoding

vControl	Register Accessed
0x00	Control Register 0 (VL)
...	...
0x1F	Control Register 31 (vstride15)
0x20	Address Register 0
...	...
0x3F	Address Register 31
0x40	Increment/Decrement Register 0
...	...
0x5F	Increment/Decrement Register 31
0x60	Window Register 0
...	...
0x7F	Window Register 31

A.7.4 Instruction Encoding

Arithmetic/Logic/Move Instructions

Table A.16 lists the function field encodings for vector register instructions. Table A.17 lists the function field encodings for vector-scalar and scalar-vector operations. These instructions use the vector-scalar instruction format. The same instruction mapping applies to all operand sizes.

Flag and Miscellaneous Instructions

Table A.18 lists the function field encoding for vector flag logic and miscellaneous instructions.

Table A.16: Vector register instruction function field encoding

	[2:0] Function bit encoding for .vv							
[5:3]	000	001	010	011	100	101	110	111
000	vadd	vsub		vmac	vand		vor	vxor
001	vaddu	vsubu		vmacu		vabsdiff		
010	vsra	vcmpeq				vcmplt	*vdiv*	vcmple
011	vmerge	vcmpneq	vsll	vsrl	vrot	vcmpltu	*vdivu*	vcmpleu
100		vmax	vext	vins		vmin	vmulf	vmulh
101		vmaxu				vminu	vmulfu	vmulhu
110	vccacc		*vcomp*	*vexpand*	vmov	vabs	vsmov	vxmov
111	vczacc						vssct	vxsct

Table A.17: Scalar-vector instruction function field encoding

	[2:0] Function bit encoding for .vs							
[5:3]	000	001	010	011	100	101	110	111
000	vadd	vsub		vmac	vand		vor	vxor
001	vaddu	vsubu		vmacu		vabsdiff		
010	vsra	vcmpeq				vcmplt	*vdiv*	vcmple
011	vmerge	vcmpneq	vsll	vsrl	vrot	vcmpltu	*vdivu*	vcmpleu
100		vmax	vext	vins		vmin	vmulhi	vmullo
101		vmaxu	vextu			vminu	vmulhiu	vmullou
	[2:0] Function bit encoding for .sv							
[5:3]	000	001	010	011	100	101	110	111
110	vsra	vsub				vcmplt	*vdiv*	vcmple
111	vmerge	vsubu	vsll	vsrl	vrot	vcmpltu	*vdivu*	vcmpleu

Table A.18: Flag and miscellaneous instruction function field encoding

	[2:0] Function bit encoding for flag/misc instructions							
[5:3]	000	001	010	011	100	101	110	111
000	vfclr	vfset			vfand	vfnor	vfor	vfxor
001	*vffl1*	*vffl1*						
010	*vfsetof*	*vfsetbf*	*vfsetif*					
011			vfins.vs		vfand.vs	vfnor.vs	vfor.vs	vfxor.vs
100								
101	vmstc	vmcts						
110			vfld	vfst				
111								

Appendix B

Data Alignment Crossbar Network Background

B.1 Multistage Networks

Multistage switching/interconnect networks can be found in a wide variety of communication systems, from the early telecommunication systems to today's parallel computing systems. This section presents a few examples of multistage switching networks as background on the development of the data alignment crossbar network.

B.1.1 Clos Network

One of the earlier multistage switching networks is the Clos network, first formalized by Charles Clos in 1953 [9]. Designed for telephone switching systems, the Clos network provides a way to realize large crossbar switching systems by multiple stages of smaller, more feasible crossbar switches. The Clos network consists of three stages: input stage, middle stage, and output stage; each stage is made up of a number of crossbars, as illustrated in Figure B.1. Note that the middle stage can be broken down further to form a 3-stage Clos network of its own. By repeating this, the Clos network can contain any odd number of stages. An example of this is the Benes network, which will be discussed in the next section.

The Clos networks are defined by three integers m, n, and r. The input stage is made up of r crossbars, each with n number of input and m number of outputs. The middle stage consists of m crossbars, each with r number of inputs and outputs. Finally, the output

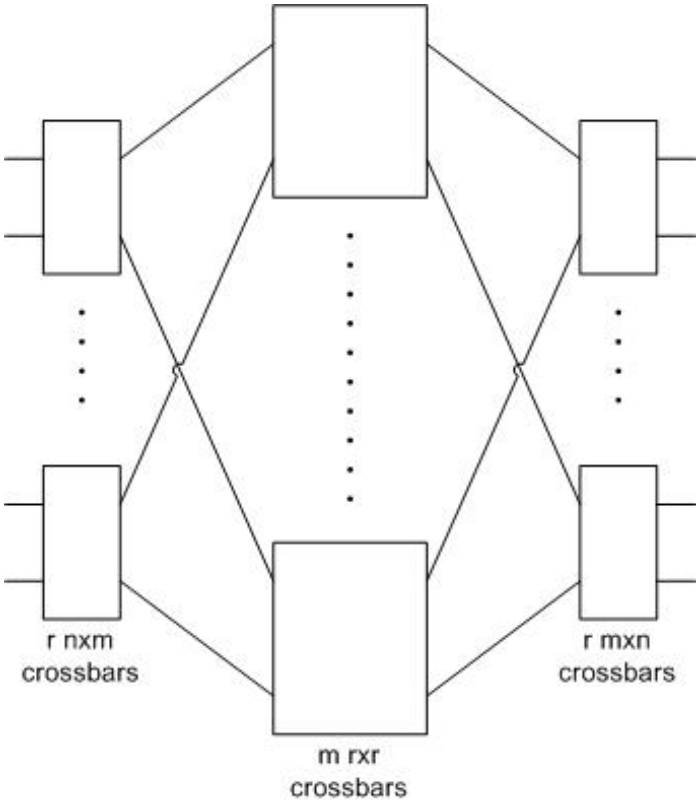


Figure B.1: Clos Network

stage consists of r crossbars, each with m number of inputs and n number of outputs. The blocking characteristic of Clos networks are defined by the relationship between m and n. Blocking refers to when two or more signals routing through the switching network contends for a physical link in the network causing the routing to fail. In [9], Clos stated that if $m \geq 2n - 1$ the network is strict-sense nonblocking, meaning that an unused input on the input stage crossbars can always connect to an unused output of the output stage crossbar without having to rearrange already established connections. A Clos network with $m \geq n$ is rearrangeably nonblocking, meaning that an unused input can always connect to an unused output, but existing connections may have to be rerouted through a different middle stage. With $m < n$, blocking could occur and not all input/output pairing can be realized.

B.1.2 Benes Network

The Benes network is a special case of the Clos family of networks, with $m = n = 2$, and is rearrangeably nonblocking as defined above. A $N \times N$ Benes network has $2 \log N - 1$ stages, each stage made up of $\frac{N}{2}$ 2×2 crossbars. Figure B.2 shows an 8×8 Benes network. The Benes network can be constructed recursively, moving from the outside in as shown by the dash lines in Figure B.2. First, the input and output stages are built with 2×2 switching elements, leaving the two middle stages each with $\frac{N}{2}$ inputs and outputs. Then repeat this step for each of the two middle crossbars, until the middle stage is a single 2×2 switching element. This recursive construction will help provide some insight to how the control algorithm for VIPERS II's permutation network is derived.

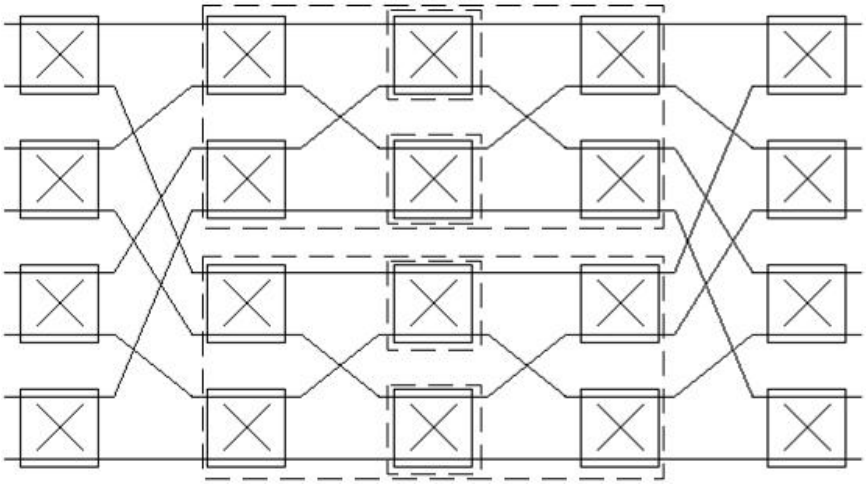


Figure B.2: 8x8 Benes Network

B.2 Control Algorithm

All switching networks require some form of control algorithm to help guide the inputs to their corresponding outputs. The simplest control scheme is bit control, where the inputs of a $N \times N$ network are assigned a destination tag showing its destination as a binary number. Each bit of the destination tag is then used to control the switching element at

certain stages of the network. Bit control is the natural control scheme for the Omega and Banyan networks; however, both the Omega and Banyan are blocking networks, so they are not capable of realizing all of VIPERS II's permutation requirements.

In order to meet these requirements, the bigger, rearrangeably non-blocking Benes network is chosen to implement the $N \times N$ data alignment network in the VIPERS II architecture. Since the network will be implemented in hardware, an efficient and simple control algorithm must be developed.

In [17], a control algorithm for the Benes network was proposed. Lee viewed the Benes network as two subnetworks: the first $\log N - 1$ stages is a distribution network controlled by a Complete Residue Partition Tree (CRPT), and the remaining $\log N$ stages is bit controlled. The paper demonstrated that if the destination tags of groups of 2^j inputs at the latter $\log N$ stages form a Complete Residue System (CRS), it would be passable using bit control. For example, in a 8×8 network with destination tag $b_2b_1b_0$, each set of two inputs at the switching element must have a (0, 1) pair for b_2 and the group of four inputs must have (00, 01, 10, 11) quadruples for b_2b_1.

Using the results from [17], a simple control algorithm for VIPERS II's data alignment network is devised. The control algorithm targets only the first $\log N - 1$ stages of the network, setting up the inputs so the latter $\log N$ stages can be bit controlled.

For offset moves, the first $\log N - 1$ stages are set to pass the data straight through as shown in Figure B.3. The contiguous nature of vector elements and the reverse shuffle interconnect pattern through the first 2 stages ensure that the inputs at the third stage has CRS form as shown by the circles. The reverse shuffle at the input stage makes sure that each pair of inputs will not end up in the same half network; one will be routed to the top half and the other to the lower half. With the contiguous destination tags, this means successive tags such as 000 and 001 will not be routed through the same half network. Applying the same analogy to each of the half network repeatedly, it can be seen that the inputs at the middle stage will have CRS form and can be routed through the final $\log N$

stages by bit control.

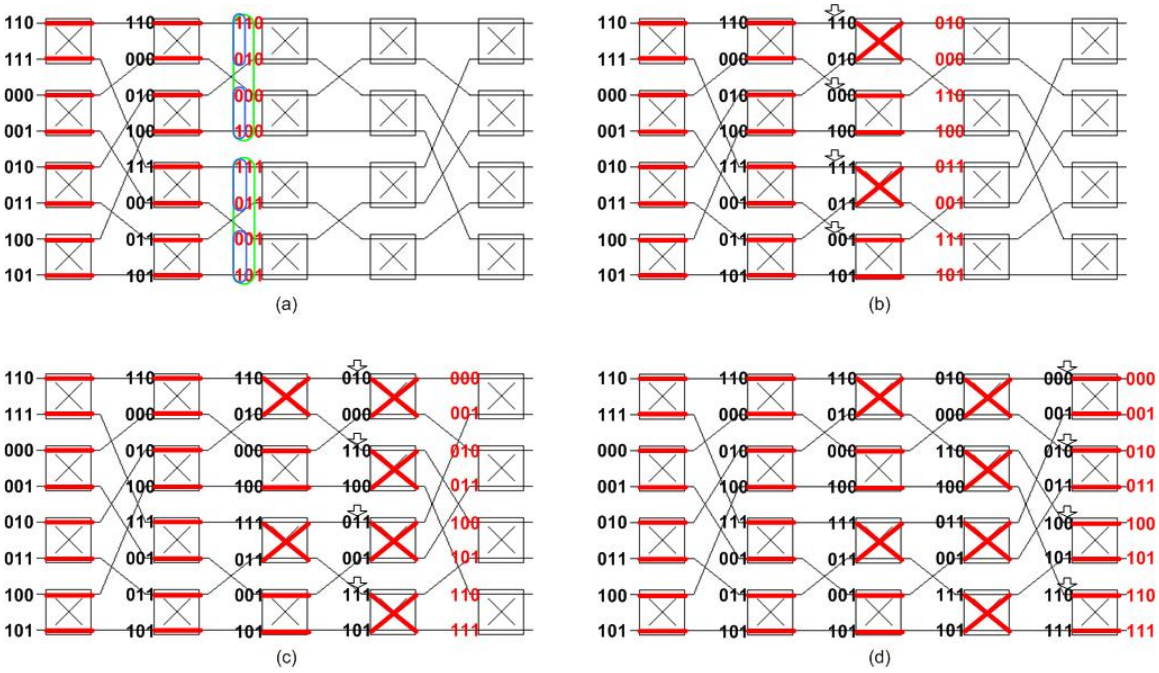


Figure B.3: Offset Move in 8×8 Benes Network

For strided moves, the first $\log N - 1$ stages are set as follows:

- if stride is odd, all switches are set to pass the data straight through,
- if stride is even, the switch settings would alternate every $\frac{Stride}{2}$ switches.

Figure B.4 shows the example of a strided move with stride = 2, where the switch setting alternates for every switch. The alternating switch setting is required to make sure that consecutive inputs do not get routed through the same half network, and thus ensuring CRS forms at the middle stage.

Due to time constraints, no formal proof was derived for this control scheme for the Benes network. However, it was verified by software for networks up to 128×128 in size, tested for all possible offset and stride input combinations.

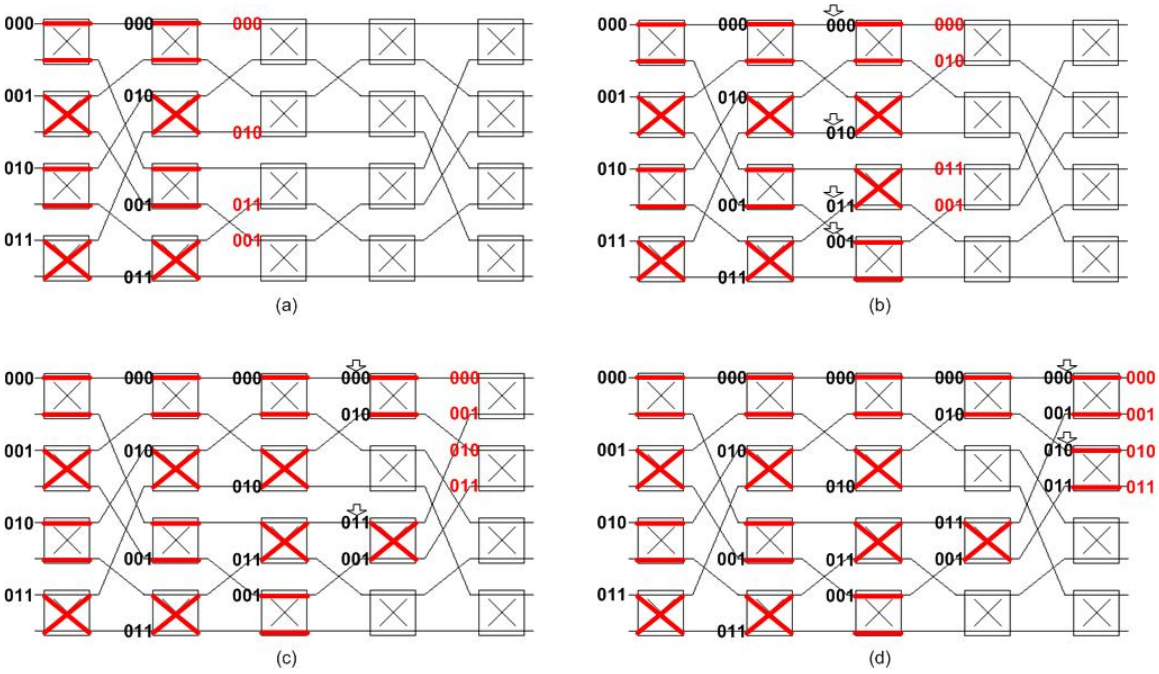


Figure B.4: Strided Move in 8×8 Benes Network

B.3 Tag Generation

Self-routing of data through the latter $\log N$ stages of the Benes network requires a destination tag for each valid input. The tag generation unit is used to compute these destination tags, which declares the target output terminal for each input data. The destination tags are calculated based on the source and destination offsets and the stride of the requested move operations. For offset moves, the destination tags can be computed using the following equation:

$$DestTag = [(ByteLocation - SrcOffset) + DestOffset]\%MemWidthByte$$

The *ByteLocation* stands for the input data's location in the memory slice being moved, starting from 0 for the byte that is aligned to the memory width. The *SrcOffset* and *DestOffset* are the source and destination byte offsets respectively, and *MemWidthByte* is the memory width in number of bytes. This equation assigns a tag value between 0 to (*MemWidthByte*-1) to every input byte in the memory slice that is being moved. As

mentioned previously, the tag generation for strided and indexed moves are not implemented at this moment.