VECTOR THREAD PROCESSOR

A Thesis

submitted by

SANGMESHWAR MANGRULE

for the award of the degree

of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
June 2016

THESIS CERTIFICATE

This is to certify that the thesis titled **Vector Thread Processor,** submitted by **Sangmeshwar Mangrule,** to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology,** is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V. Kamakoti
Project Guide
Professor
Dept. of Computer Science and Engineering
IIT Madras

Place: Chennai, 600 036
Date: 10th June, 2016

Acknowledgement

I would like to express my sincere gratitude to my guide, **Dr. V.Kamakoti** for his valuable guidance, encouragement and advice. His immense motivation helped me in making firm commitment towards my project work.

My special thanks to **Mr. G.S. Madhusudan** for his encouragement and motivation throughout the project. His valuable suggestions and constructive feedback were very helpful in moving ahead with my project work.

I would like to thank my co-guide **Dr. Nitin Chandrachoodan** and faculty advisor **Dr. Deelip Nair** who have patiently listened, evaluated, and guided us throughout the program.

My special thanks to my project team members Neel Gala, Arjun Menon, Rahul and Vikas Chauhan for their help and support.

Sangmeshwar Mangrule
(EE14M060)

Abstract

The vector-thread (VT) architecture unifies the vector and threaded compute models. The VT abstraction provides the programmer a control processor with a vector of virtual processors (VPs). The control processor uses vector-fetch commands to broadcast instructions to all the VPs or each VP can use thread-fetch commands to direct its own control flow. A seamless intermixing of the vector and multithreaded control mechanisms allows a VT architecture to compactly and flexibly encode the applications like parallelism and locality and a VT design exploits these to improve performance and efficiency. We present a VT architecture which is considerably simpler to implement and easier to program. Using an extensive design-space exploration of full VLSI implementations of many accelerator design patterns, we evaluate the varying tradeoffs between the programmability and implementation efficiency among the MIMD, vector-SIMD, and VT patterns.

Contents

List of Figures

Tables

Abbreviations

VT	Vector Thread
CP	Control Processor
VP	Virtual Processor
MIMD	Multiple Instruction, Multiple Data
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
VPV	Virtual Processor Vector
CMD	Command Management Unit
EC	Execution Cluster

Chapter 1

Introduction

For the productive use of increasing transistor counts while coping with the wire delay and power dissipation, Parallelism and locality are the key application characteristics exploited by computer architects. Conventional sequential architectural ISAs provide minimal support for encoding parallelism or locality, so high-performance implementations are needed to devote considerable area and power to on-chip structures that extract parallelism or that support arbitrary global communication. The large area and power overheads are studied by the demand for even small improvements in performance for popular ISAs. Many important applications have huge parallelism, however, with graph dependencies and communication patterns that can be statically determined. ISAs that expose more parallelism mitigate the need for area and power intensive structures. Similarly, ISAs that expose locality reduce the need for long range communication and complex interconnect. The challenge is to develop an efficient encoding architecture which provides more parallelism and locality to reduce the area and power consumption of the microarchitecture.

VT architecture allows large amounts of structured parallelism to be compactly encoded in a form that allows a simple microarchitecture to attain high performance at low power and small area by avoiding complex control and data path structures and by reducing activity on long wires. Implementation of the VT architecture also exploits instruction-level parallelism within AIBs.

Thus the VT architecture supports all forms of application parallelism and this flexibility provides new ways to parallelize code which is difficult to vectorize or that gives excessive synchronization costs when threaded. Instruction Locality is improved by allowing common code to be factor out and executed once on the control processor and by executing the same AIB many times on each VP in turn. Data locality is improved as most operand communication is isolated within an individual VP.

We will first introduce a set of five architectural design patterns for DLP cores, comparing their programmability and efficiency. The MIMD pattern flexibly supports a mapping of data-parallel tasks to a collection of simple scalar and multithreaded cores, but lacks the efficient execution of regular DLP. The vector-SIMD and subword-SIMD patterns can significantly

reduce the energy on regular DLP, but require complicated programming for irregular DLP. The single-instruction multiple-thread (SIMT) and vector thread (VT) patterns are hybrid patterns that attempt to offer alternative tradeoffs between programmability and efficiency.

There is a large design space to explore while reducing these patterns to an efficient VLSI design. There are a set of parameterized synthesizable micro-architectural components and how these components can be combined to form various complete RTL designs for the different architectural design patterns so that it reduces total design effort and allowing a fairer comparison across different patterns. Another important thing is to use the same RISC ISA for both vector and scalar code which greatly reduces the effort required to develop an efficient VT compiler. Multi-lane implementations are usually more efficient than multi-core single-lane implementations and are easier to program as they require less partitioning and load balancing.

.

Chapter 2

Architectural design patterns

By considering the data parallel mechanisms in efficient computing environment, The data parallel applications can be categorized in two dimensions depending on the memory access and control flow access. The regularity with which data memory is accessed and the regularity with which control flow changes. The two categories are Regular data parallelism and Irregular data parallelism.

Regular data-level parallelism (DLP) has well structured data accesses where the data addresses can be compactly encoded and are known in advance of when the data is ready. Regular DLP also has well structured control flow where the control decisions are either known statically or well in advance of when the control flow actually occurs. Irregular DLP may have less structured data accesses where the addresses are more dynamic and difficult to predict and might also have less structured control flow. Irregular DLP also include some small number of inter-task dependencies that force a portion of each task to wait for previous tasks to finish.

Figure1 illustrate the spectrum from regular to irregular DLP. There are several studies which demonstrate that full DLP applications contain a mix of both regular and irregular DLP. First of all it is possible to improve performance and energy-efficiency even on irregular DLP. Finally, a consistent way of mapping both regular and irregular DLP simplifies the programming methodology. The next section presents five architectural design patterns and describes how each pattern handles both regular and irregular DLP.

```
for (i = 0; i < n ;  i++)
  C[i] = x * a [i] + B[2*i];
```

(a) Regular DA & Regular CA

```
for ( i=0 ; i < n ; i++ )
  E[C[i]] = D[A[i]] + B[i];
```

(b) Irregular DA & Regular CA

```
for ( i = 0; i < n; i++ )
  C[i] = false; j = 0;
while ( !C[i] & (j < m) )
  if ( A[i] == B[j++] )
    C[i] = true;
```

(e) Irregular DA & Irregular CF

```
for (i=0; i < n ; i++)
  x = ( A[i] > 0 ) ? y : z ;
  C[i] = x * A[i] + B[i] ;
```

(c)Regular DA & Irregular CF

```
for (i=0 ; i < n ; i++)
  if ( A[i] > 0 )
    C[i] = x * A[i] + B[i];
```

(d) Irregular DA & Irregular CF

Figure 1: Types of Data-Level Parallelism – Examples expressed in a C-like pseudo code and are ordered from regular DLP (i.e., regular (DA) and (CF)) to irregular DLP (i.e. irregular DA and irregular (CF)).

Different Architectural Patterns for Data-Parallel Accelerators

1.1 MIMD Architectural Design Pattern

The multiple-instruction multiple-data (MIMD) pattern is the simplest one to build a data-parallel accelerator. A large number of scalar cores are connected across a single chip. Programmers can map each data-parallel task to a separate core but without proper dedicated DLP mechanisms, it is difficult to get energy-efficiency advantages when executing DLP applications. These scalar cores can be extended to support multithreading per core which helps to improve performance by hiding various latencies. Figure1 (a) shows the programmer's logical view and Figure1 (b) an example implementation for the multithreaded MIMD pattern. All of the design patterns include a host thread (HT) as part of the programmer's logical view. The HT runs on the general-purpose processor and is responsible for configuration, application startup, interaction with the operating system and managing the data-parallel accelerator. We refer to the threads that run on the data parallel accelerator as micro-threads (μTs), as they are lighter weight than the threads which run on the GPP. The primary advantage of the MIMD pattern is the easier and flexible programming model, and since every core can execute an independent task, there will be little difficulty in mapping both the regular and irregular DLP applications. This can simplify parallel programming comparing to the other design patterns, but the primary disadvantage is that this pattern does little to improve the energy efficiency of DLP applications.

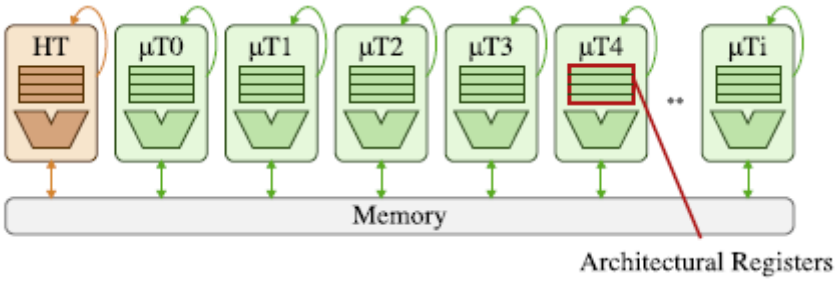

Fig1: (a) Programmer's logical view – MIMD [2 & 13]

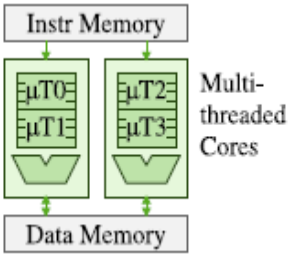

Fig1: (b) Typical core Microarchitecture – MIMD [2 & 13]

1.2 Vector-SIMD Architectural Design Pattern

In the vector single-instruction multiple-data (Vector-SIMD) pattern as shown in figure2 (a) the control thread (CT) uses vector memory instructions to move data between main memory and vector registers whereas vector arithmetic instructions to operate on vectors of elements at once. One way to think of this pattern is as each CT manages an array of μTs that execute in lock-step; each μT is responsible for one element of the vector and the hardware vector length is the number of μTs (e.g., four in Figure2 (b)). In this context, μTs are sometimes referred to as virtual processors (VP). Unlike in the MIMD pattern, the HT in the Vector-SIMD only interacts with the CTs and does not directly manage the μTs. Even the HT and CTs must still allocate work at a coarse-grain among themselves via software, this configuration overhead is amortized by the hardware vector length. The CT distributes work to the μTs with vector instructions enabling very efficient execution of fine-grain DLP. In a typical vector-SIMD core, the CT is mapped to a control processor and the μTs are mapped across one or more vector lanes in the vector unit. The

vector memory unit (VMU) handles executing vector memory instructions and the vector issue unit (VIU) handles the dependency checking and dispatch of vector arithmetic instructions.

The vector memory commands are divided into two parts: the address portion goes to the VMU which issue the request to MEMORY while the register write/read portion goes to the VIU. For vector loads, the register writeback (wb) waits until the data returns from memory and then controls writing the vector register file two elements per cycle over two cycles. Note that the VMU/VIU are decoupled from the vector lanes to allow the implementation to overlap processing multiple vector loads. The vector arithmetic operations are also processed two elements per cycle over two cycles, some μTs are inactive because the corresponding vector flag is false. The temporal mapping of μTs to the same lane is an important aspect of the vector-SIMD pattern. We can imagine that using a larger vector register file to support longer vector lengths that would keep the vector unit busy for tens of cycles. The fact that one vector command can keep the vector unit busy from any cycles but decreases instruction issue bandwidth pressure. So as in the MIMD pattern we can exploit instruction-level parallelism by adding support for executing multiple instructions per μT per cycle, but unlike the MIMD pattern in vector-SIMD it may not be necessary to increase the issue bandwidth since one vector instruction occupies a vector functional unit for many cycles and almost all vector-SIMD accelerators will take advantage of multiple functional units and also support bypassing (also called vector chaining) between these units. A final point to note is that how the control processor (CP) decoupling and multi-cycle vector execution enables the control thread to continue executing while the vector unit is still processing older vector instructions. This decoupling means the control thread can quickly work through the loop overhead instructions so that it can start issuing the next iteration of the loop as soon as possible.

Finally, vector-SIMD pattern can improve energy efficiency in three different ways:

1. Some instructions are executed once by the CT instead for each μT like MIMD pattern
2. For operations that the μTs do execute, the CP and VIU can amortize various overheads such as instruction fetch, decode, and dependency checking etc.
3. For memory accesses which the μTs still execute the VMU can efficiently move the data in large blocks

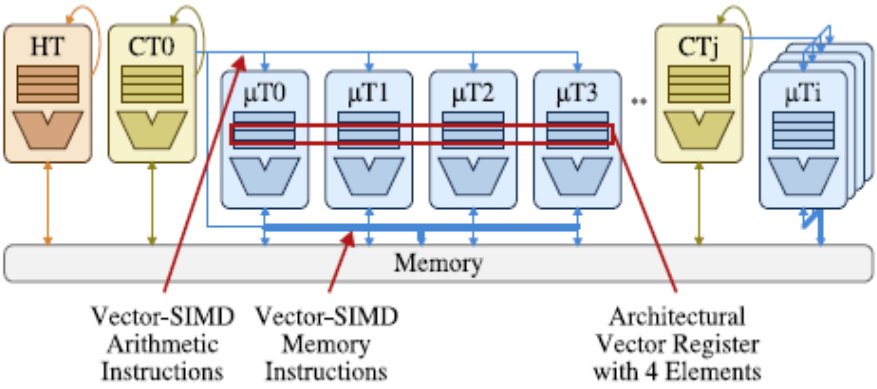

Fig2: (a) Programmer's logical view – Vector-SIMD [2 & 13]

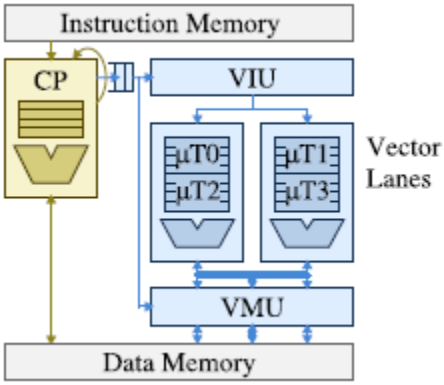

Fig2: (b) Typical core microarchitecture - Vector-SIMD [2 & 13]

1.3 Subword-SIMD Architectural Design Pattern

The subword single-instruction multiple-data (subword-SIMD) architectural pattern shown in Figure3 (a) captures some important differences from the vector-SIMD pattern. In this pattern, A full-word scalar datapath is vector like unit with standard scalar registers often corresponding to a double-precision floating-point unit. The pattern leverages these existing scalar datapaths and registers to execute multiple operations in a single cycle. Some variants support bitwidths larger than the widest scalar datatype, in which case the datapath can only be fully utilized with subword-SIMD instructions and the other variants unify the CT and SIMD unit such that the same datapath is used for control, scalar arithmetic, and subword-SIMD instructions. Subword-

SIMD has short vector lengths that are exposed to software as fixed width datapaths whereas vector-SIMD has longer vector lengths that are exposed to software as a true vector of elements. In vector-SIMD, the vector length is exposed in such a way that the same binary can run on many different implementations with varying hardware resources whereas code for one subword-SIMD implementation is usually less portable to other implementations with varying h/w resources. Vector-SIMD has more flexible data-movement operations which alleviates the need for software data shuffling, while Subword-SIMD often requires shuffling elements via special permute instructions, and this leads to a large amount of cross-element communication

The vector-SIMD pattern is better suited to exploiting large amounts of data-parallelism as opposed to a more general-purpose workload with smaller amounts of data-parallelism so we do not focus more on subword-SIMD.

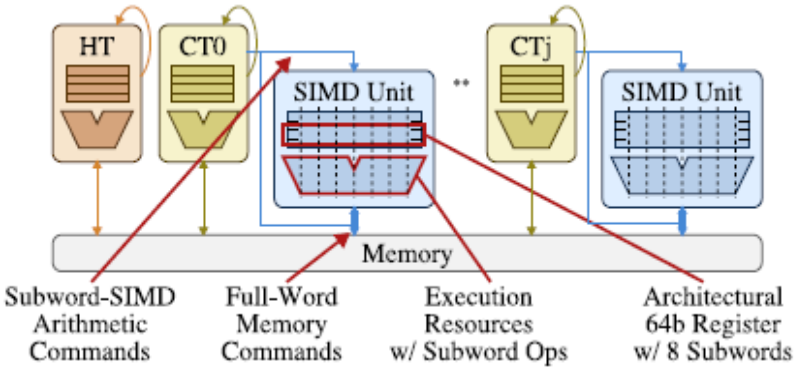

Fig3: (a) Programmer's logical view – Subword-SIMD [2 & 13]

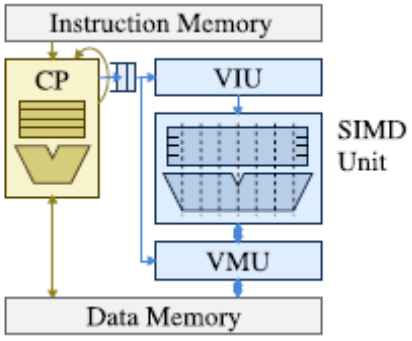

Fig3: (b) Typical core microarchitecture - Subword-SIMD [2 & 13]

2.4 SIMT Architectural Design Pattern

The single instruction multiple-thread (SIMT) pattern is a hybrid with a programmer's logical view shown is similar to the MIMD pattern but an implementation similar to the vector-SIMD pattern. As shown in Figure4 (a) below the SIMT pattern supports a large number of μTs but no CTs; the HT is responsible for directly managing the μTs and a μT block is mapped to a SIMT core which contains vector lanes similar to those in the vector-SIMD pattern. However, since there is no CT, the VIU is responsible for amortizing overheads and executing the μT scalar instructions in lock-step when they are coherent. The VIU also manages the situation when the μTs execute a scalar branch possibly causing them to diverge. μTs can sometimes re-converge through static hints in the scalar instruction stream or dynamic hardware mechanisms. SIMT has only scalar loads and stores, but the VMU can include a memory coalescing unit to dynamically detect when these scalar accesses can be converted into vector memory operations. The SIMT pattern usually exposes the concept of a μT block to the programmer that the barriers are sometimes provided for intra-block synchronization and application performance depends heavily on the coherence and coalescing opportunities within a μT block.

The loop in Figure4 (b) maps to the SIMT pattern in a similar way as in the MIMD pattern except that each μT is only responsible for a single element as opposed to a number of elements. Since there are no control threads (CT) and thus no similarity to the vector-SIMD pattern, a combination of dedicated hardware and software is required to manage the stripmining. The host thread (HT) tells the hardware how many μT blocks are required for the computation and the hardware manages the case when the number of requested μT blocks is greater than what is available in the actual hardware. In the common case, where the application vector length is not statically guaranteed to be evenly divisible by the μT block size then each μT must use a scalar branch to verify that the computation for the corresponding element is actually necessary.

There are some issues that can prevent the SIMT pattern from achieving vector-like energy efficiencies on regular DLP that the μTs must redundantly execute instructions that would otherwise be amortized onto the CT. Regular data accesses are encoded as multiple scalar accesses which must be dynamically transformed into vector-like memory operations. On the other hand, the lack of a control thread (CT) necessitates per μT stripmining calculations and prevents access-execute decoupling which can efficiently tolerate memory latencies. Even though the ability to achieve vector-like efficiencies on coherent μT instructions helps to improve energy-efficiency compared to the MIMD pattern. However, The real strength of the

SIMT pattern is that it provides a simple way to map complex data-dependent control flow with μT scalar branches.

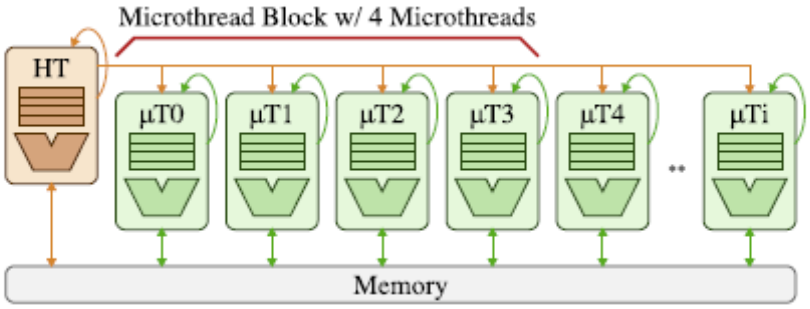

Fig4: (a) Programmer's logical view – SIMT [2 & 13]

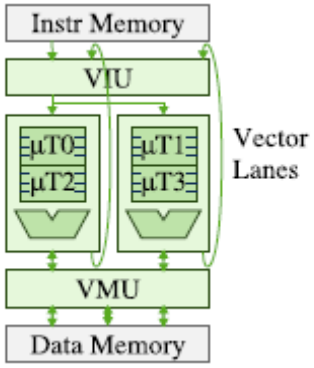

Fig4: (b) Typical core microarchitecture - SIMT [2 & 13]

2.5 VT Architectural Design Pattern

The vector-thread (VT) pattern is also a hybrid pattern like SIMT but it takes a very different approach. As shown in Figure5 (a) the HT manages a collection of CTs and each CT manages an array of μTs. Similar to the vector-SIMD pattern, this allows various overheads to be amortized onto the CT and CTs can execute vector memory commands to efficiently handle regular data accesses. In this pattern, the CT does not execute vector arithmetic instructions like vector-SIMD but instead uses a vector fetch instruction to indicate the start of a scalar instruction stream that should be executed by the μTs. The VIU allows μTs to execute coherently as in the SIMT pattern but they can also diverge after executing scalar branches.

In VT pattern loop control, regular data accesses and stripmining are handled just as in the vector-SIMD pattern. Instead of vector arithmetic instructions (VAI), we use a vector fetch instruction (VFI) with one argument which indicates the instruction address at which all μTs should immediately start executing. All μTs execute the scalar instructions till the stop instruction. An important aspect of the VT pattern is that the interaction between vector registers as accessed by the control thread and scalar registers as accessed by each μT. Each μT's scalar register an implicitly refers to that μT's element of the vector register. In other words, the vector register as seen by the control thread and the scalar register as seen by the μTs are two views of the same register. The μTs cannot access the control thread's scalar registers as this would significantly complicate control processor (CP) decoupling. Shared accesses are thus communicated with a scalar load by the control thread and a scalar-vector move instruction which copies the given scalar register value into each element of the given vector register. A scalar branch used to encode data-dependent control flow.

An explicit scalar-vector move instruction writes the scalar value into each element of the vector register with two elements per cycle over the two cycles. The unit-stride vector load instruction is executed as in the vector-SIMD pattern. The control processor (CP) then sends the vector fetch instruction to the VIU. The VIU fetches the branch instruction and issues them across μTs. The VIU waits until all μTs resolve the scalar branch as similar to the SIMT pattern. If all μTs either they take the branch or do not take, then the VIU can start fetching from the appropriate address. If some μTs take the branch while others do not, then the μT diverge and the VIU needs to keep track of which μTs are executing which side of the branch.

VT achieves vector-like energy-efficiency while maintaining the ability to flexibly map regular and irregular DLP. Control instructions are executed once by the control thread per-loop. A scalar branch provides a convenient way to map complex data-dependent control flow and the VIU is still able to amortize instruction fetch, decode and dependency checking for vector arithmetic instructions (VAI). VT uses the same vector memory instructions to move blocks of data efficiently between memory and vector registers. However there are some overheads including the extra scalar-vector move instruction, vector fetch instruction, and μT stop instruction.

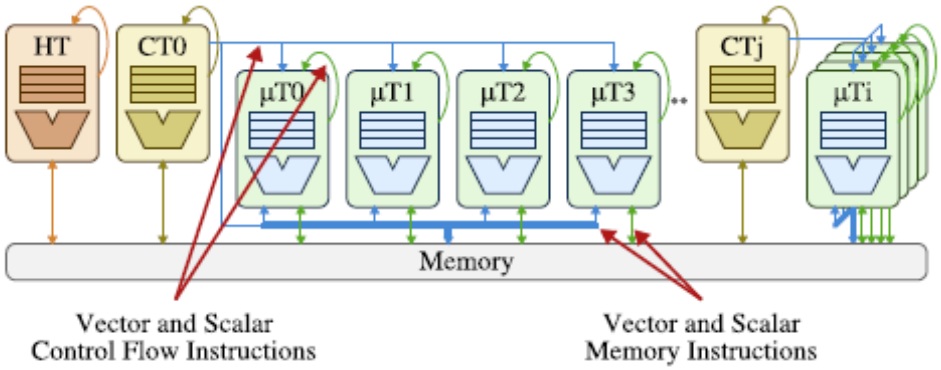

Fig5: (a) Programmer's logical view – VT [2 & 13]

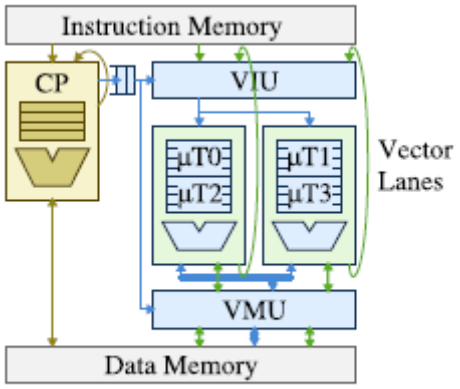

Fig5: (b) Typical core microarchitecture - VT [2 & 13]

Chapter 3

Vector Thread Processor

This section first describes the abstraction view of a VT architecture provides to a programmer then gives an overview of the physical model for a VT machine.

1.4 VT Abstract Model

The vector-thread architecture unifies the vector and multithreaded models. A conventional control processor (CP) interacts with a vector of virtual processor (VPV), as shown in Figure6 (a). The programming model consists of two instruction sets, one for the control processor (CP) and one for the VPs. Applications can be mapped to the VT architecture in a variety of ways but it is especially well suited to executing loops. Each VP executes a single iteration of the loop and the control processor is responsible for managing the execution.

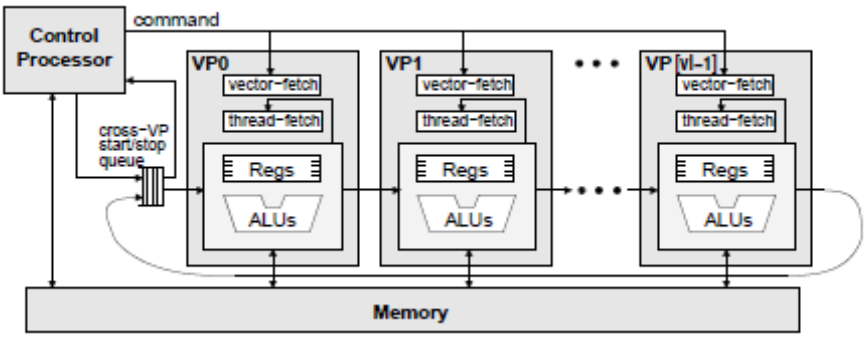

Figure 6: (a) Abstract model of a vector-thread architecture [1]

A VP contains a set of registers and has the ability to execute RISC-like instructions with virtual register specifiers. VP instructions are grouped into atomic instruction blocks (AIBs) and the unit of work issued to a VP at one time. There is no automatic program counter (PC) or implicit instruction fetch mechanism for VPs; all instruction blocks must be explicitly requested by either the control processor or the VP itself.

The control processor (CP) can direct VPs' execution using a *vector-fetch* command to issue an AIB to all the VPs in parallel, or a *Thread-fetch* to target an individual VP. Vector-fetch commands provide a programming model similar to conventional vector machines but a large block of instructions can be issued at once. As a simple example, below Figure6 (b) shows the mapping for a data parallel vector-vector add loop. The AIB for one iteration of the loop contains

two loads, an add, and a store instructions. A vector-fetch command sends this AIB to all the VPs in parallel and thus initiates vl loop iterations, where vl is the length of the virtual processor vector (VPV) i.e., the vector length. Every VP executes the same instructions but operates on different data elements determined by its index number. Though a more efficient alternative to the individual VP loads and stores shown in the example, a VT architecture also provides vector memory commands issued by the control processor which move a vector of elements between memory and a register in each VP.

The VT abstract model connect VPs in a unidirectional ring topology and allows a sending instruction on VP(n) to transfer data directly to a receiving instruction on VP(n+1). This type of *cross-VP* data transfer is dynamically scheduled and resolved when the data becomes available. Cross-VP data transfers allow loops with cross-iteration dependencies to be efficiently mapped to the vector thread architecture, as shown in the figure6 (c). A single vector-fetch command introduces a chain of prevVP receives and nextVP sends that spans the VPV. The control processor can push an initial value into the *cross-VP start/stop queue* before executing the vector-fetch command. After the chain executes, the final cross-VP data value from the last VP turns around and is written into the same queue. It can then be popped by the control processor or consumed by a subsequent prevVP VP0 during stripmined loop execution.

The ability to freely intermix vector-fetches and thread-fetches allows a VT architecture to combine the best attributes of both vector and multithreaded execution paradigms. As shown in figure6 (d), the control processor can broadcast a vector-fetch command to launch a vector of VP threads and each of which continues to execute as long as it issues thread-fetches. These thread-fetches break the rigid control flow of traditional vector machines by enabling the VP threads to follow independent control paths. Thread-fetches are broadening the range of loops which can be mapped efficiently to VT, allowing the VPs to execute data-parallel loop iterations with conditionals or inner-loops. Beyond these loops, the VPs can also be used as *free-running threads*, where they operate independently from the control processor and retrieve tasks from a shared work queue.

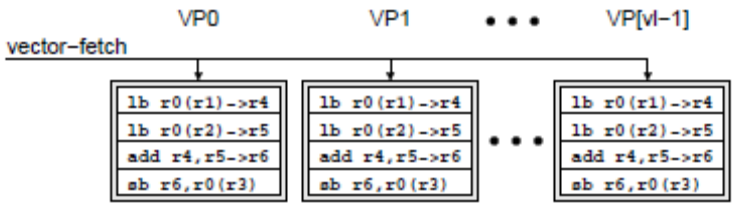

Figure 6: (b) Vector-fetch commands [1 & 9]

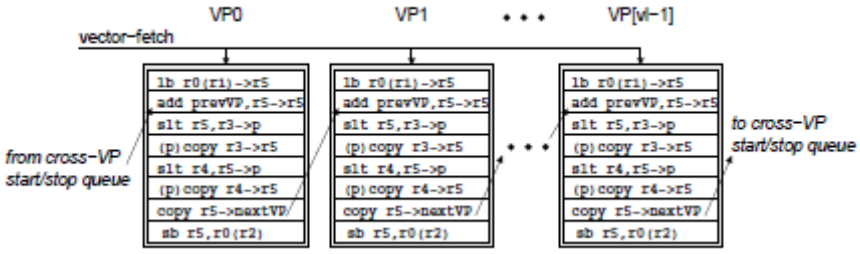

Figure 6: (c) Cross-VP data transfers [1 & 9]

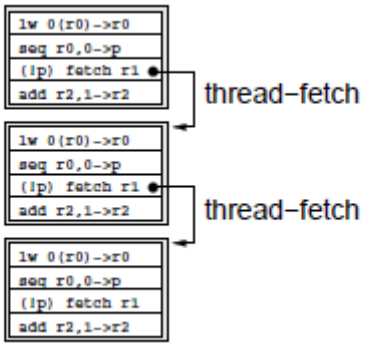

Figure 6: (d) VP threads [1 & 9]

The VT architecture allows software to efficiently expose structured parallelism and locality. Compared to a conventional threaded architecture, the VT model allows common book-keeping code to be factored out and executed once on the control processor rather than in each

thread. AIBs enable a VT machine to efficiently amortize instruction fetch overhead and provide a framework for easily handling temporary state. Vector-fetch commands explicitly encode instruction locality and parallelism, allowing a VT machine to attain high performance while amortizing control overhead. Vector-memory commands avoid separate load and store requests for each element and can be used to exploit memory data-parallelism even in loops with non-data-parallel. For loops with cross-iteration dependencies, cross-VP data transfers explicitly encode synchronization and communication, avoiding heavyweight inter-thread memory coherence and synchronization primitives.

1.5 VT Physical Model

A physical model is the expected structure for efficient implementations of the abstract model. The VT physical model contains a conventional scalar control processor together with a *vector thread unit* (VTU) that executes the VP code. To exploit the parallelism exposed by the VT abstract model, the VTU contains an array of processing *lanes* parallel as shown in Figure7 (a). VPs are mapped into the lanes and the VPV is striped across the lane array. Each lane contains functional units, which are time-multiplexed across the VPs and physical registers, which hold the state of VPs mapped to the lanes. Unlike to the traditional vector machines, the lanes in a VT machine execute decoupled from each other. Figure7 (b) shows an abstract view of how VP execution is time multiplexed on the lanes for both vector-fetched and thread-fetched AIBs. This fine-grain interleaving helps VT machines hide memory, functional unit and thread-fetch latencies.

As shown in figure7 (a)**,** each lane contains a *command management unit* (CMU) and an *execution cluster*. An execution cluster consists of a functional unit, register file and a small AIB cache. The CMU buffers commands from the control processor in a queue (cmd-Q) and holds pending thread-fetch addresses for the corresponding lane's VPs. The CMU also holds the tags for the lane's AIB cache in cache tag. The AIB cache holds one or more AIBs and must be at least large enough to hold an AIB of the maximum size defined in the VT architecture.

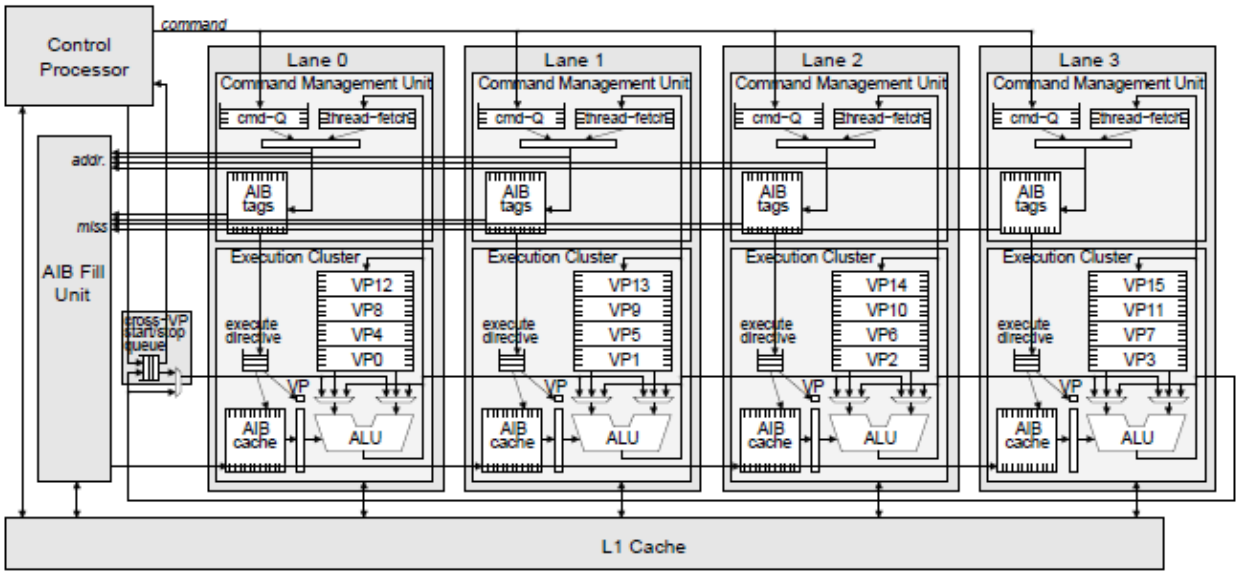

Figure 7: (a) Physical model of a VT machine [2]

The implementation has four parallel lanes in the vector-thread unit (VTU) and VPs are mapped across the lane array with the low-order bits of a VP index indicating the lane to which it is stripped. The configuration shown uses VPs with five virtual registers, and with twenty physical registers each lane is able to support four VPs. Each lane is divided into a command management unit (CMU) and an execution cluster. The execution cluster has an associated cross-VP start-stop queue.

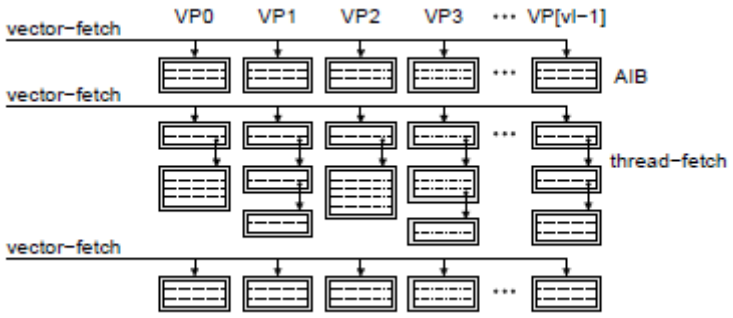

Figure 7: (b) The control processor can use a vector-fetch command to send an AIB to all the VPs, after which each VP can uses thread-fetch to fetch its own AIBs.

. The CMU chooses a vector-fetch, VP-fetch or thread-fetch command to process. The fetch command contains an address which is looked up in the AIB tags and if there is a miss, a request is sent to the fill unit which retrieves the requested AIB from the primary cache. The fill unit handles one lane's AIB miss at a time except if lanes are executing vector-fetch commands when refill overhead is amortized by broadcasting the AIB to all lanes at the same time.

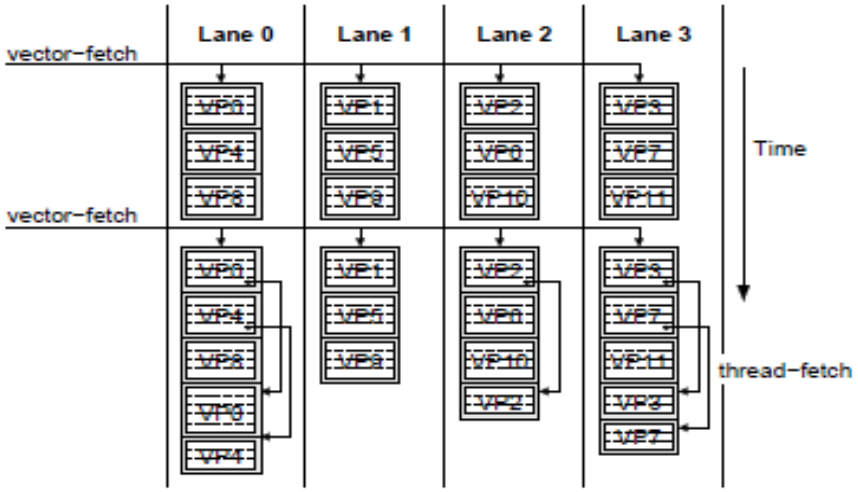

Figure 7: (c) Lane Time-Multiplexing. Both vector-fetch and thread fetch AIBs are time-multiplexed on the physical lanes.

After a miss refill has been processed or after a fetch command hits in the AIB cache, the CMU generates an *execute directive* which contains an index into the AIB cache. For a vector-fetch command the execute directive provides that the AIB should be executed by all VPs mapped to the lane whereas for a VP-fetch or thread-fetch command it identifies a single VP to execute the AIB. The execute directive is sent to a queue in the execution cluster, leaving the CMU free to begin processing the new commands. The CMU is able to overlap the AIB cache refill for new commands with the execution of previous ones but must track which AIBs have outstanding execute directives to avoid overwriting their entries in the AIB cache. The CMU must also ensure that the VP threads execution has completed before initiating a subsequent vector-fetch.

In processing a execute directive, the cluster takes VP instructions one by one from the AIB cache and executes them for the appropriate VP. While processing an execute-directive

from a vector-fetch command, all of the instructions in the AIB are executed once for one VP before moving on to the next. The virtual register indices in the VP instructions are combined with an active VP number to create an index into the physical register file. To execute a fetch instruction, the cluster sends the requested AIB address to the CMU where the VP's associated pending address of that thread-fetch register is updated.

The lanes in a VTU are connected with an unidirectional ring network to implement the cross-VP data transfers. When a cluster encounters an instruction with a prevVP receive, it stalls until the data is available from its previous lane. When the VT architecture allows multiple cross-VP instructions in a single AIB with some sends preceding some receives, the hardware implementation must provide sufficient buffering of send data to allow all the receivers in an AIB to execute. By induction, deadlock is avoided if each lane ensures that its predecessor can never be blocked to send its cross-VP data.

Chapter 4
SCALE Vector Thread Architecture

SCALE is an instance of the VT architectural paradigm designed for embedded systems applications. The SCALE architecture has a MIPS-based scalar control processor extended with a VTU. The SCALE VTU aims to provide high performance at low power for a wide range of applications while using a small area. In this section we will describe the SCALE VT architecture, a simple code example implemented on SCALE, and gives an overview of the SCALE microarchitecture.

4.1 Clusters

To improve the performance while reducing energy, area and circuit delay, SCALE extends the single-cluster VT model (shown in Figure1) by partitioning VPs into multiple execution clusters with independent register sets. VP instructions process an individual cluster and perform RISC-like operations. Source operands must be local to the cluster but results can be written to any cluster in the VP and its result from an instruction can be written to multiple destinations. Each cluster within a VP has a separate predicate register (pr), and instructions can be positively or negatively predicated.

SCALE clusters are heterogeneous, but all clusters support basic integer computations. Additionally, Cluster0 supports memory access instructions, cluster1 supports fetch instructions, and cluster3 supports integer multiply and divide. Though we do not consider, also the SCALE architecture allows clusters to be enhanced with layers of additional functionality (e.g., floating-point operations, fixed-point operations, and sub-word SIMD operations), or new clusters to be added to perform specialized operations.

4.2 Registers and VP Configuration

The general registers in each cluster of a VP are categorized as either private registers (pr's) and shared registers (sr's). Both private and shared registers can be written and read by VP

instructions and by commands from the control processor. The main difference between them is that the private registers preserve their values between AIBs, while shared registers may be overwritten by a different VP. Shared registers can be used as temporary within an AIB to increase the number of VPs that can be supported by a fixed number of physical registers. The control processor can also use vector-write the shared registers to broadcast scalar values and constants used by all VPs.

In addition to the general registers, each cluster also has *chain registers* (cr0 and cr1) associated with the two ALU input operands and these can be used as sources and destinations to avoid reading and writing the register files. Like shared registers, chain registers may be overwritten between AIBs and they can also be implicitly overwritten when a VP instruction uses their associated operand position. Cluster0 has a special chain register called the store-data (sd) register through which all data for VP stores must pass.

In the SCALE architecture, the control processor configures the VPs by indicating how many shared and private registers are needed in each cluster. The length of the virtual processor vector (VPV) changes with each re-configuration to reflect the maximum number of VPs that can be supported. This operation is done once outside each loop and state in the VPs is undefined across reconfigurations. Within a processing lane, the VTU maps shared VP registers to shared physical registers. Control processor vector-write to a shared register are broadcast to each lane, but individual VP writes to a shared register are not coherent across lanes.

4.3 Vector Memory Commands

In addition to the VP load and store instructions, SCALE defines vector-memory commands issued by the control processor for efficient execution of structured memory accesses. Like vector-fetch commands, these operate across the virtual processor vector (VPV); a vector-load writes the load data to a private register in each VP whereas a vector store reads the store data from a private register in each VP. SCALE also supports vector-load commands which target the shared registers to retrieve values used by all VPs. In addition to the unit stride and strided vector-memory access patterns, SCALE also provides vector segment accesses where each VP loads or stores several contiguous memory elements to support “array-of-structures” data layouts efficiently.

4.4 SCALE Code Example

The SCALE code to implement the decoder function from the c code presented is shown below. The code is divided into two sections: one with MIPS control processor code in the .text section and SCALE VP code in the .sisa (SCALE ISA) section. The SCALE VP code implements one iteration of the loop with a single AIB. cluster0 accesses memory, cluster1 accumulates index, cluster2 accumulates valpred, and cluster3 does the multiply.

```
void decode_ex(int len, u_int8_t* in, int16_t* out)
{
 int i;
 int index = 0;
 int valpred = 0;
     for(i = 0; i < len; i++)
          {
          u_int8_t delta = in[i];
          index += indexTable[delta];
                    index = index < IX_MIN ? IX_MIN : index;
                    index = IX_MAX < index ? IX_MAX : index;
                    valpred += stepsizeTable[index] * delta;
                    valpred = valpred < VALP_MIN ? VALP_MIN : valpred;
                    valpred = VALP_MAX < valpred ? VALP_MAX : valpred;
                    out[i] = valpred;
                    }
}
```

Figure 8: C code for decoder example.

.text # control processor code

```
decode_ex: # a0=len, a1=in, a2=out
# configure VPs: c0:p,s c1:p,s c2:p,s c3:p,s
vcfgvl t1, a0, 1,2, 0,3, 1,3, 0,0          # (vl,t1) = min(a0,vlmax)
sll    t1, t1, 1                           # output stride
la     t0, indexTable
vwrsh  t0, c0/sr0                          # indexTable addr.
la     t0, stepsizeTable
vwrsh  t0, c0/sr1                          # stepsizeTable addr.
vwrsh  IX_MIN, c1/sr0                      # index min
vwrsh  IX_MAX, c1/sr1                      # index max
vwrsh  VALP_MIN, c2/sr0                    # valpred min
vwrsh  VALP_MAX, c2/sr1                    # valpred max
xvppush  $0, c1                            # push initial index = 0
xvppush  $0, c2                            # push initial valpred = 0
stripmineloop:
setvl  t2, a0                              # (vl,t2) = min(a0,vlmax)
vlbuai  a1, t2, c0/pr0                     # vector-load input, inc ptr
vf vtu_decode_ex                           # vector-fetch AIB
vshai  a2, t1, c2/pr0                      # vector-store output, inc ptr
subu   a0, t2                              # decrement count
bnez   a0, stripmineloop  # loop until done
xvppop  $0, c1                             # pop final index, discard
xvppop  $0, c2                             # pop final valpred, discard
vsync                                      # wait until VPs are done
jr ra                                      # return
```

.sisa # SCALE VP code

```
vtu_decode_ex:
.aib begin
c0 sll  pr0, 2 -> cr1                      # word offset
c0 lw   cr1(sr0) -> c1/cr0                 # load index
c0 copy  pr0 -> c3/cr0                     # copy delta
c1 addu  cr0, prevVP -> cr0                # accum. index
c1 slt  cr0, sr0 -> p                      # index min
c1 psel  cr0, sr0 -> sr2                   # index min
```

```
c1 slt   sr1, sr2 -> p                          # index max
c1 psel  sr2, sr1 -> c0/cr0, nextVP             # index max
c0 sll   cr0, 2 -> cr1                          # word offset
c0 lw    cr1(sr1) -> c3/cr1                     # load step
c3 mult.lo cr0, cr1 -> c2/cr0                   # step*delta
c2 addu  cr0, prevVP -> cr0                     # accum. valpred
c2 slt   cr0, sr0 -> p                          # valpred min
c2 psel  cr0, sr0 -> sr2                        # valpred min
c2 slt   sr1, sr2 -> p                          # valpred max
c2 psel  sr2, sr1 -> pr0, nextVP                # valpred max
.aib end
```

Figure : SCALE code implementing decoder example

Chapter 5

SCALE Microarchitecture

The SCALE microarchitecture is an extension of the general VT architecture model shown in figure7 (a). In each lane has a single CMU and one physical execution cluster per VP cluster. Each cluster has a dedicated output bus which broadcasts data to the other clusters in the lane and it also connects to its sibling clusters in neighboring lanes to support cross-VP data transfers. An overview of the SCALE lane microarchitecture is as shown in figure8 (a).

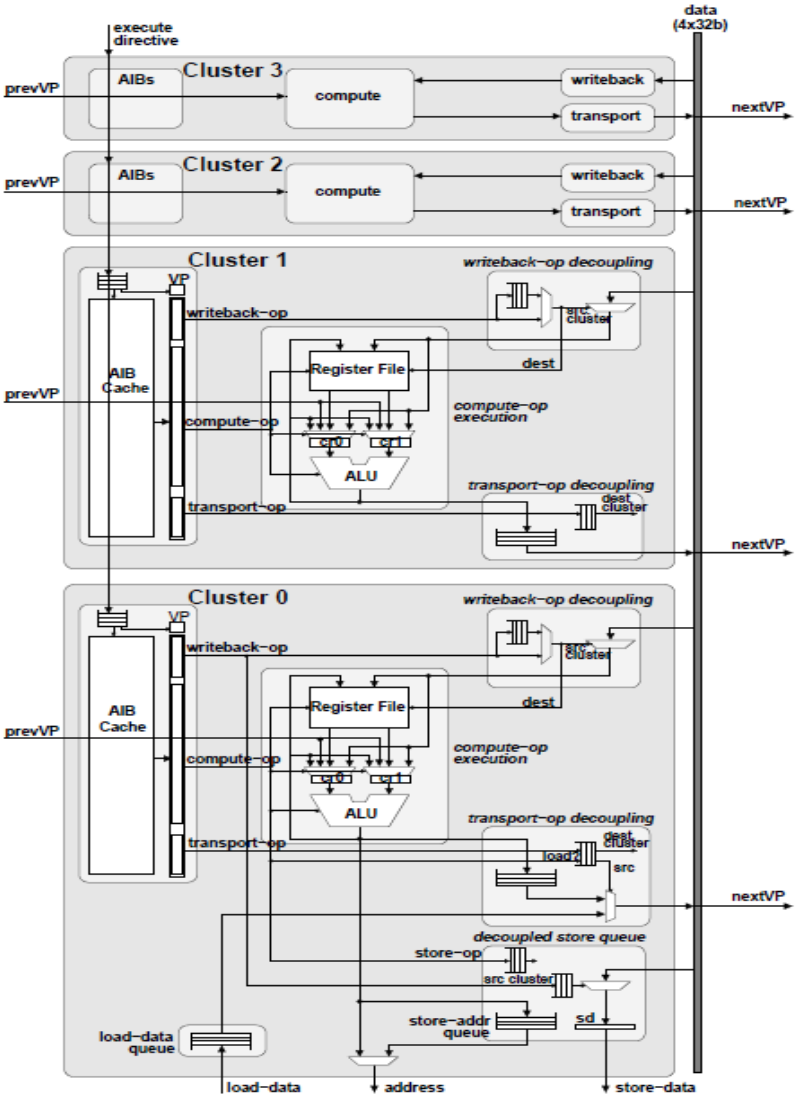

Figure 8: (a) SCALE Lane Microarchitecture [1]

In the above Scale Lane Microarchitecture The AIB caches hold micro-op bundles. The compute-op is a local RISC operation on the cluster, the transport-op sends data to external clusters and the writeback-op receives data from external clusters. Clusters 1, Cluster 2 and Cluster 3 are basic cluster designs with writeback-op and transport-op decoupling resources (cluster 1 is shown in detail, clusters 2 and 3 are shown in abstract). Cluster 0 connects to memory and includes memory access decoupling resources.

5.1 Micro-Ops and Cluster Decoupling

The SCALE ISA is portable across multiple SCALE implementations, but is designed to be easy to translate into implementation-specific micro-operations or *micro-ops*. The assembler translates the SCALE software ISA into the native hard-ware ISA at compilation time. There are three categories of hardware micro-ops. 1. A *compute-op* performs the main RISC-like operation of a VP instruction 2. A *transport-op* sends data to another cluster and 3. A *writeback-op* receives data sent from an external cluster. An assembler reorganizes micro-ops derived from an AIB into *micro-op bundles* which target a single cluster and do not access other clusters' registers. Figure8 (b) how the SCALE VP instructions from the above decoder example are translated into micro-op bundles. All inter-cluster data dependencies are encoded by the transport-ops and writeback-ops which are added to the sending and receiving cluster respectively. This allows the micro-op bundles to be packed together independently for each cluster from the micro-op bundles for other clusters.

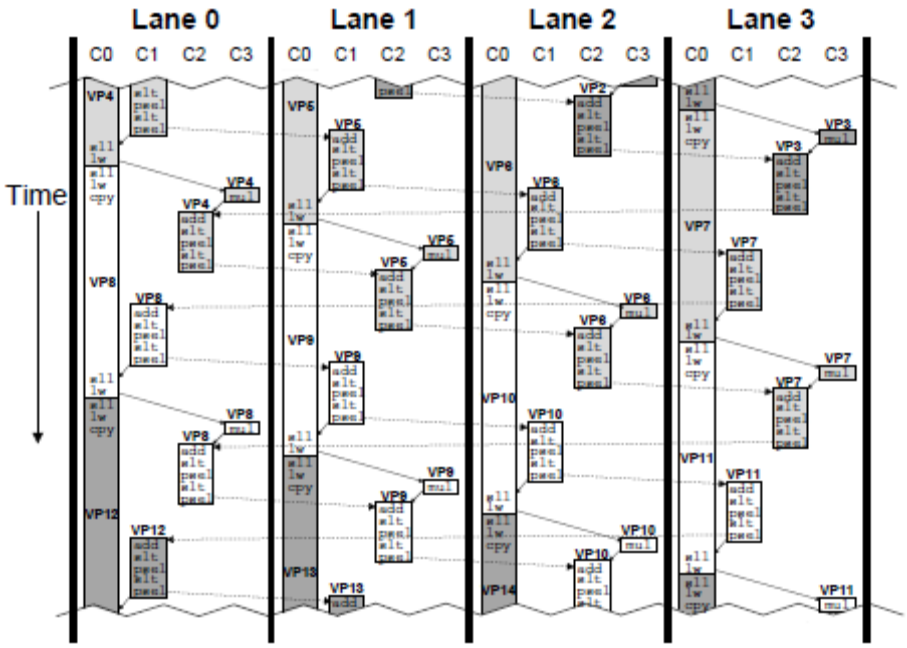

Figure 8: (b) Execution of decoder example on SCALE architecture. Each cluster executes in-order, but cluster and lane decoupling allows the execution to automatically adapt to the software critical path. Arrows represent critical dependencies (solid for inter-cluster within a lane, dotted for cross-VP) [1]

Inter-cluster data transfers are partitioned into transport and writeback operations enables decoupled execution between clusters. In SCALE, a cluster's AIB cache contains micro-op bundles and each cluster has a local execute directive queue and local control. Each cluster processes its transport-ops in order and broadcasts result values onto its dedicated output data bus and each cluster processes its writeback-ops in order, writing the values from external clusters to its local registers. The synchronization of the inter-cluster data dependencies with handshake signals which extend between the clusters and a transaction completes only when both the sender and the receiver are ready. Although compute-ops execute in order, each cluster contains a transport queue to allow execution to proceed without waiting for external destination clusters to receive the data, and a writeback queue to allow execution to proceed without waiting for data from external clusters until it is needed by a compute-op. Thus, queues make inter-cluster synchronization more flexible and thereby enhance cluster decoupling.

A schematic diagram of the example decoder loop executing on SCALE is shown in Figure8 (b). Each cluster executes the vector-fetched AIB for each VP mapped to its lane and decoupling allows each cluster to target to the next VP independently. Execution adapts to the software critical path as each cluster's local data dependencies resolve. In the above example, the accumulations of index and valpred must execute serially but all of the other instructions are not on the software critical path. Further, the two accumulations can execute in parallel, so the cross-iteration serialization penalty is paid only once. Each VP loop iteration executes over a period of 30 cycles, but the combination of multiple lanes and cluster decoupling within each lane leads to as many as six loop iterations executing simultaneously.

5.2 Memory Access Decoupling

All VP loads and stores execute on cluster0 (c0) and it is specially designed to enable *access-execute decoupling*. Typically, c0 loads data values from memory and sends them to other clusters then computation is performed on the data and finally results are returned to c0 and stored to the memory. With this type of basic cluster decoupling, c0 can continue execution after a load without waiting for the other clusters to receive the data. Further, Cluster0 is enhanced to hide memory latencies by continuing execution after a load misses in the cache and therefore it might retrieve load data from the cache out of order. Even though like other instructions, load operations on cluster0 use transport-ops to deliver data to other clusters in order and uses a *load data queue* to buffer the data and preserve the correct ordering.

Interestingly, when cluster0 encounters a store, it does not have to wait for the data to be ready. Instead it buffers the store operation, including the store address and in the *decoupled store queue* until the store data is available. When a SCALE VP instruction targets the store data (sd)register, the resulting transport-op sends data to the store unit rather than to c0. Thus, the store unit acts as a primary destination for inter-cluster transport operations and it handles the writeback-ops for sd. Store decoupling allows a lane's load stream to slip ahead of its store stream but loads for a given VP are not allowed to bypass previous stores to the same address by the same VP.

5.3 Vector Memory Accesses

As discussed above vector-memory commands are sent to the clusters as special execute directives which generate micro-ops instead of reading them from the AIB cache. For a vector-load, writeback-ops receive the load data on the destination cluster and for a vector-store, compute-ops and transport-ops on the source cluster read and send the store data. To allow overlapped execution of vector fetched AIBs and vector-memory operations, chaining is provided.

The vector-memory commands are also sent to the *vector memory unit (VMU)* which performs the necessary cache accesses. The vector-memory unit can only send one address to the cache in each cycle but it takes advantage of the structured access patterns to load or store multiple elements with each access. The vector-memory unit uses load and store data to and from cluster 0 in each lane to reuse the buffering already provided for the decoupled VP loads and stores.

Chapter 6

Scale VT Instruction Set Architecture

The Scale architecture is an instance of the vector-thread architectural paradigm. For the Hardware /software interface, Scale is particularly targeted at embedded systems—the goal is to provide high performance with low power dissipation for a wide range of applications while using only a small area. The Instruction set is provided as follows:-

configure VPs and set vector length	vcfgvl rdst, rlen, nc0p, nc0s, nc1p, nc1s, nc2p, nc2s, nc3p, nc3s	Configure the VPs with n_{c0p} private and n_{c0s} shared registers in cluster 0, n_{c1p} private and n_{c1s} shared registers in cluster 1, etc. The n_{c0p} parameter is also used as the number of private store-data registers in cluster 0. State in the VPs becomes undefined if the new configuration is not the same as the existing configuration. The configuration determines the maximum vector length which the VTU hardware can support, and this value is written to vlmax. The new vector length is then computed as the minimum of r_{len} and vlmax, and this value is written to vl and r_{dst}.
set vector length	setvl r_{dst}, r_{len}	The new vector length length is computed as the minimum of r_{len} and vlmax, and this value is written to vl and r_{dst}.
vector-fetch	vf label	Send the AIB located at the label to every active VP in the VPV.
VP fetch	vpf[.nb] r_{vp}, label	Send the AIB located at the label to the VP specified by r_{vp}. The *.nb* version is non-blocking.
vector sync	Vsync	Stall until every VP in the VPV is idle and has no outstanding memory operations
vector fence	Vfence	Complete all memory operations from previous vector commands (vector-fetch, vector-load, and vector-store) before those from any subsequent vector commands.
VP sync	vpsync r_{vp}	Stall until the VP specified by r_{vp} is idle and has no outstanding memory operations.
VTU kill	Vkill	Kill any previously issued VTU commands at an arbitrary point of partial execution and reset the VTU into an idle state.
VP reg-read	vprd[.nb] r_{vp}, r_{dst}, c_{src}/r_{src}	Copy c_{src}/r_{src} in the VP specified by r_{vp} to r_{dst}. The *.nb* version is non-blocking.

VP reg-write	vpwr[.nb] r_{vp}, r_{src}, c_{dst}/r_{dst}	Copy r_{src} to c_{dst}/r_{dst} in the VP specified by r_{vp}. The *.nb* version is non-blocking.
vector reg-write shared	vwrsh r_{src}, c_{dst}/r_{dst}	Copy r_{src} to c_{dst}/r_{dst} in every VP in the VPV. r_{dst} must be a shared register
cross-VP push	xvppush r_{src}, c_{dst}	Push a copy of r_{src} to the cross-VP start/stop queue for cluster c_{dst}.
cross-VP pop	xvppop r_{dst}, c_{src}	Pop from the cross-VP start/stop queue for cluster c_{src} and store the value into r_{dst}
cross-VP drop	xvpdrop c_{src}	Pop from the cross-VP start/stop queue for cluster c_{src} and discard the value

Basic VTU commands

Operation	***Assembly format***	***Summary***
unit-stride vector load	vL r_{base}, c_{dst}/r_{dst} vLai r_{base}, r_{inc}, c_{dst}/r_{dst}	Each active VP in the VPV loads the element with address: r_{base} + width · VPindex (where width is the number of bytes determined by the opcode, and VPindex is the VP's index number) to c_{dst}/r_{dst}. The load-data is zero-extended for the u versions of the opcodes, or sign-extended otherwise. r_{dst} must be a private register. For the *ai* versions of the opcode, r_{base} is automatically incremented by r_{inc}.
segment-strided vector load	vLsegst n, r_{base}, r_{str}, c_{dst}/r_{dst} vLseg n, r_{base}, c_{dst}/r_{dst} vLst r_{base}, r_{str}, c_{dst}/r_{dst}	Similar to *unit-stride vector load*, except each VP loads n elements with addresses: $r_{base} + r_{str} \cdot VPindex + width \cdot 0$ $r_{base} + r_{str} \cdot VPindex + width \cdot 1$ . . . $r_{base} + r_{str} \cdot VPindex + width \cdot (n - 1)$ to $c_{dst}/(r_{dst}+0)$, $c_{dst}/(r_{dst}+1)$, . . ., $c_{dst}/(r_{dst}+(n-1))$. For the simplified *seg* versions of the opcode, the stride (r_{str}) is equal to the segment width (width · n). For the simplified *st* versions of the opcode, the segment size (n) is 1.
shared vector load	vLsh r_{base}, c_{dst}/r_{dst}	Every VP in the VPV loads the element with address r_{base} to c_{dst}/r_{dst}. The load-data is zero-extended for the u versions of the opcodes, or sign-extended otherwise. r_{dst} must be a shared or chain register.
unit-stride vector store	vS r_{base}, $c0/sd_{src}$ vS r_{base}, $c0/sd_{src}$,P vSai r_{base}, r_{inc}, $c0/sd_{src}$ vSai r_{base}, r_{inc}, $c0/sd_{src}$,P	Each active VP in the VPV stores the element in $c0/sd_{src}$ to the address: r_{base} + width · VPindex (where width is the number of bytes determined by the opcode, and VPindex is the VP's index number). For the predicated versions, the store only occurs if the store-data predicate register in cluster 0 is set to one with the (c0/sdp) argument or zero with the (!c0/sdp) argument. sd_{src} must be a private store-data register. For the *ai* versions of the opcode, r_{base} is automatically incremented by r_{inc}.
segment-strided vector store	vSsegst n, r_{base}, r_{str}, $c0/sd_{src}$ vSsegst n, r_{base}, r_{str}, $c0/sd_{src}$,P vSseg n, r_{base}, $c0/sd_{src}$ vSseg n, r_{base}, $c0/sd_{src}$,P vSst r_{base}, r_{str}, $c0/sd_{src}$ vSst r_{base}, r_{str}, $c0/sd_{src}$,P	Similar to *unit-stride vector store*, except each VP stores the n elements in $c0/(sd_{src}+0)$, $c0/(sd_{src}+1)$, . . ., $c0/(sd_{src}+(n-1))$ to the addresses: $r_{base} + r_{str} \cdot VPindex + width \cdot 0$ $r_{base} + r_{str} \cdot VPindex + width \cdot 1$ . . .

		$r_{base} + r_{str} \cdot VPindex + width \cdot (n - 1)$ For the simplified *seg* versions of the opcode, the stride (r_{str}) is equal to the segment width (width · n). For the simplified *st* versions of the opcode, the segment size (n) is 1.

L = {lb, lbu, lh, lhu, lw}

S = {sb, sh, sw}

P = {(c0/sdp), (!c0/sdp) **Vector Load and Store commands**

Below table : **VP instruction opcodes**

Arithmetic and Logical Instructions (all clusters)

Memory (cluster 0)

Mnemonic	***Operation***
addu	addition
La*	load address
subu	subtraction
and	logical and
or	logical or
xor	logical exclusive or
nor	inverse logical or
sll	shift left logical
srl	shift right logical
sra	shift right arithmetic
seq	set if equal
sne	set if not equal
slt	set if less than
sltu	set if less than unsigned
psel	select based on predicate reg.
Copy*	copy
li?	load immediate

Mnemonic	***Operation***
Lb	load byte (sign-extend)
Lbu	load byte unsigned (zero-extend)
Lh	load halfword (sign-extend)
Lhu	load halfword unsigned (zero-extend)
Lw	load word
Sb	store byte
Sh	store halfword
Sw	store word
lw.atomic.add*	atomic load-add-store word
lw.atomic.and*	atomic load-and-store word
lw.atomic.or*	atomic load-or-store word

Fetch (cluster 1)

Fetch*	fetch AIB at address
psel.fetch	select address based on predicate reg. and fetch AIB
addu.fetch	compute address with addition and fetch AIB

Multiplication / Division (Cluster 3)

mulh	16-bit multiply (signed×signed)
mulhu	16-bit multiply (unsigned×unsigned)
mulhus	16-bit multiply (unsigned×signed)
multu.lo	32-bit multiply (unsigned×unsigned) producing low-order bits
multu.hi	32-bit multiply (unsigned×unsigned) producing high-order bits
mult.lo	32-bit multiply (signed×signed) producing low-order bits
mult.hi	32-bit multiply (signed×signed) producing high-order bits
divu.q	32-bit divide (unsigned/unsigned) producing quotient
div.q	32-bit divide (signed/signed) producing quotient
divu.r	32-bit divide (unsigned÷unsigned) producing remainder
div.r	32-bit divide (signed÷signed) producing remainder

Chapter 7
Conclusion and Future Work

Conclusion

The vector-thread architectural paradigm allows software to more efficiently encode the parallelism and locality present in many applications, while the structure provided in the hardware / software interface enables high-performance implementations that are efficient in area and power. The VT architecture support for all types of parallelism and this flexibility enables new ways of parallelizing codes. For example, by allowing vector-memory operations to feed directly into threaded code. VT exploits parallelism and locality more effectively than traditional superscalar, VLIW, or multithreaded architectures. The Scale VT architecture demonstrates that the VT is well-suited to all-purpose embedded computing, letting a single compact design provide competitive performance across a range of applications.

VT abstraction introduces a small set of primitives to allow software to succinctly encode parallelism and locality and seamlessly inter-mingle DLP, TLP, and ILP. For example, Virtual processors, AIBs, vector-fetch and vector memory commands, thread-fetches, cross VP data transfer.

Future Work

This thesis provides good start for future work on VT-based data-parallel accelerators. It gives specific directions for future work with respect to the instruction set, microarchitecture and programming methodology.

Improving Execution of Irregular DLP – This architecture indicates that the vector fragment mechanism alone is not sufficient for efficient execution of highly irregular DLP. We have introduced vector fragment merging, interleaving, and compression as techniques that can potentially improve the performance and energy efficiency on such codes. The next step would be to implement these techniques and measure their impact for our benchmarks using our evaluation methodology.

This thesis has focused on a single data-parallel core, but there are many interesting design issues with respect to which how these cores can be integrated together.

References

[1] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovi´c *MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139*

 ronny, cbatten, krste_@csail.mit.edu

[2] C. Batten. Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators. PhD Thesis, MIT, 2010.

[3] C. Batten, R. Krashinsky, S. Gerding, and K. Asanovi´c. Cache Refill/Access Decoupling for VectorMachines. Int'l Symp. on Microarchitecture (MICRO), Dec 2004.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs:Stream Computing on Graphics Hardware. ACM Transactions on Graphics (TOG), 23(3):777–786,Aug 2004.

[5] R. Espasa and M. Valero. Decoupled Vector Architectures. Int'l Symp. on High-Performance ComputerArchitecture (HPCA), Feb 1996.

[6] W.W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. DynamicWarp Formation: Efficient MIMD ControlFlow on SIMD Graphics Hardware. ACM Transactions on Architecture and Code Optimization (TACO), 6(2):1–35, Jun 2009.

[7] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. IEEE Micro, 26(2):10–24, Mar 2006.

[8] M. Hampton and K. Asanovi´c. Compiling for Vector-Thread Architectures. Int'l Symp. on Code Generationand Optimization (CGO), Apr 2008

[9] T.-C. Chiueh. Multi-threaded vectorization. In *ISCA-18*, May 1991.

[10] C. R. Jesshope. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. *Australia Computer Science Communications*, 23(4):80–88, 2001.

[11] C. Kozyrakis. *Scalable vector media-processors for embedded systems*. PhD thesis, University of California at Berkeley, May 2002.

[12] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *ISCA-30*, June 2003.

[13] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanovi´c. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators. Int'l Symp. on Computer Architecture (ISCA), Jun 2011