IDISA+: A PORTABLE MODEL FOR HIGH PERFORMANCE SIMD PROGRAMMING

by

Hua Huang

B.Eng., Beijing University of Posts and Telecommunications, 2009

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the
School of Computing Science
Faculty of Applied Science

SIMON FRASER UNIVERSITY
Fall 2011

APPROVAL

Name: Hua Huang

Degree: Master of Science

Title of Thesis: IDISA+: A Portable Model for High Performance SIMD Programming

Examining Committee:

Dr. Kay C. Wiese
Associate Professor, Computing Science
Simon Fraser University
Chair

Dr. Robert D. Cameron
Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Thomas C. Shermer
Professor, Computing Science
Simon Fraser University
Supervisor

Dr. Arrvindh Shriraman
Assistant Professor, Computing Science
Simon Fraser University
SFU Examiner

Date Approved: 7 December 2011

Declaration of
Partial Copyright Licence

Abstract

Most of today's commodity processors have single-instruction multiple-data(SIMD) instructions built in and provide SIMD within a register. However, different processor vendors tend to have different SIMD instruction sets which poses significant challenges to cross-platform SIMD programming. This thesis proposes a model called IDISA+ to overcome the compatibility issues and enable portable SIMD programming. There are more than 60 well-selected SIMD operations defined in the model, which are believed to support a broad range of applications. We have implemented the model as a toolkit with two components, a code generator for producing portable libraries and a test suite for both correctness and performance analysis on the libraries. For performance concerns, our model uses a least instruction count mechanism to select the best among implementation alternatives of library routines. The experimental results demonstrate the effectiveness of the generator and show that generated libraries in our model perform better than hand-tuned libraries.

Acknowledgments

It is a great honor and pleasure for me to have had my Master study at School of Computing Science, Simon Fraser University. I would like to thank all the people who helped and supported me during my graduate study, without whom completing this thesis would be impossible.

Foremost, I am truly indebted and thankful to my senior supervisor Dr. Robert D. Cameron for his invaluable guidance and support. Dr. Cameron helped me to complete this research with patience and numerous efforts, and he also provided me a free academic environment which makes me enjoy doing research with him. This thesis would not have been possible without him.

I would like to express my gratefulness to Dr. Thomas C. Shermer for being my supervisor and spending plenty of time to discuss with me on my research. I would also like to thank my thesis examiner Dr. Arrvindh Shriraman for reviewing this thesis and providing in-depth comments about it. I thank Dr. Kay C. Wiese for serving the chair of my defense.

I am obliged to many of my colleagues in Dr. Cameron's lab for their help and encouragement. They are Ken Herdy, Dan Lin, Vera Lukman, Nigel Medforth, Rui Yang and Shiyang Yang. I specially thank Ken Herdy, Dan Lin and Nigel Medforth for their insightful ideas and suggestions regarding my research.

Last but not least, I thank my parents and family for their love, care and encouragement. This thesis is dedicated to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In traditional scalar processing, operands are fetched in serial fashion into the processing unit before execution. At each step, a single instruction is applied on one piece of data to produce a single result. For example, an arithmetic add instruction would add together only two operands to get an answer at a time. This sequential computing system is often referred to as Single Instruction Single Data (SISD) in Flynn's taxonomy [24], which exploits neither instruction-level parallelism nor data-level parallelism.

Desktop computers, especially personal computers (PCs), have gone through a rapid development in past decades. As a result, compute-intensive applications, like multimedia processing and digital signal processing, became more and more popular on PCs. Many improvements and expansions have been integrated into architectures to meet the strong demand for this particular type of computing and improve the performance over traditional SISD systems of which one of the most important features is Single Instruction Multiple Data (SIMD).

SIMD is a parallel processing technology which is able to perform the same instruction on multiple data simultaneously. Compared to SISD, SIMD mainly has two advantages. First of all, when the data is stored in blocks, SIMD can load one block of data at a time in only a single instruction [48] instead of a series of instructions fetching the data one by one in SISD. Another advantage is that instructions in SIMD can be applied to all the data of a block in one operation. More specifically, for the arithmetic add operation, SIMD would perform the addition on eight pairs of values to produce eight sums in a register. Thus, from the computer architecture's point of view, SIMD systems exploit data-level parallelism, and are being used to improve the performance of today's many software applications.

Nowadays, commodity processor manufacturers including Intel, AMD, ARM and IBM have expanded their instruction set architectures with SIMD extensions to accelerate algorithms used in gaming and multimedia related processing. Some of the SIMD extensions have even evolved for many generations, such as the Streaming SIMD Extensions (SSE) series on Intel platform [56]. However, each processor vendor develops and maintains his own versions of SIMD instruction sets due to the lack of a unified commercial standard. That being said, although most commodity processors support SIMD techniques natively, challenges still exist for creating SIMD applications over different platforms. Detailed reasons are listed as follows.

- Because of the high diversity of design and associated algorithms in architectures, the instructions, field widths and the size of registers differ substantially between different SIMD instruction sets. An application written in a source-level language is often implemented based on the current hardware details, such as the available SIMD operations and memory access restrictions. Such implementations are unlikely to work properly if the underlying platform is changed. For example, there is an instruction called *pcmpgtq* in Intel SSE4 instruction set [30] which is to compare two pairs of packed quadword (64-bit) data for "greater than" simultaneously, however, no such instruction existed in PowerPC AltiVec instruction set [47]. Hence, an application built on Intel platform with SSE4 using the *pcmpgtq* instruction is not likely to work on the PowerPC platform. This makes it difficult and time consuming to write programs which achieve good performance over various architectures, even though these architectures are fairly similar to each other in most important ways.

- A SIMD instruction set is usually designed and implemented as an expansion within an existing architecture. Due to the redesign and modification cost of the architecture, the instruction set would have a set of limited functionality and algorithms that is believed to be the most economical and effective implementation on its host platform. Thus, it is common that even within the same SIMD instruction set, the implementation of an instruction is only available on some pre-chosen data sizes. Take the Intel SSE2 instruction set [32] as an example. The instruction *pmaxsw* in SSE2 computes the maximum of packed signed word (16-bit) integers. But this instruction only works on packed 16-bit numbers, and does not provide the functionality of maximum for other data sizes. This makes it tough to use SSE2 if we have an application that

heavily relies on 32-bit integer maximum operations. Also, for those instructions under some specific data sizes and not yet implemented as built-ins, the alternative implementations should be found if the cross-platform programming is a concern.

- Often, programs developed based on an early version of a SIMD instruction set continue to be used on the later versions. But, new instructions that may improve performance are usually checked in as the instruction set evolves within an architecture. It is quite difficult for programmers to tune the performance of the programs over multiple generations.

As SIMD has been incorporated into many commercial processors and is still been actively maintained and promoted by the chip vendors, it is believed that SIMD instruction sets will continue to evolve in many aspects such as new instructions, larger register size and so forth. On the other hand, developers have also used many kinds of SIMD instruction sets to build fast applications on various platforms over years. As a result, programmers might transfer their code from one platform to another with a hope of no incompatibility issues, and they would also like to make use of the capabilities of a newly released instruction set without doing too much work.

Towards addressing the cross-platform issues of SIMD programming and filling the gap between developers and underlying SIMD instruction sets, a portable and high performance SIMD programming model is presented in this thesis.

The model is based on the inductive doubling principle [19] for making in-register SIMD operation sets. Details about inductive doubling principle are provided in the Chapter 3. For implementing the model itself, the thesis has focused on building a toolkit to provide portable and high performance SIMD library support targeting kinds of commodity platforms. There are three main contributions in the thesis:

- The model defines a set of carefully selected operations as library routines which provide a uniform and clean interface for higher level development over various platforms. The operation set is well defined and is believed to capture the most important features of SIMD integer programming so that a wide range of applications could be built upon it.

- The model is built in a way that the implementations of SIMD operation are automatically generated according to the current architecture information and compiler flags.

It is also designed to be very flexible that it allows users to define new operations and add support for new architectures. In fact, the process of defining a new operation or adding support of a entire new architecture into the model is easy and fast. More details about this will be explained in later chapters.

- Performance concerns are considered in the model as well. Normally speaking, a model with more portability maybe expected to sacrifice performance. However, this is not true in our model. In the experiments of the thesis, it shows that the implementations generated by the model achieve slightly better performance than the hand-tuned implementations.

The remainder of this thesis is organized as follows. Chapter 2 reviews the background of SIMD implementations on various architectures and the related work on portable SIMD programming. The detailed definition of the model is given in Chapter 3. In Chapter 4, the algorithm for implementing the model is presented. Chapter 5 collects all the evaluation results and gives some related analysis. At last, Chapter 6 concludes the thesis with a summary of results and directions for future work.

Chapter 2

Background

2.1 SIMD Basics

SIMD is a parallel computing concept which describes computers with processing units that perform the same operation on multiple data elements simultaneously. The first use of SIMD instructions was in vector supercomputers of the early 1970s, which could operate on a vector of data with a single instruction [7]. In contrast to scalar computing, SIMD computing is able to apply instructions to each of the vector's elements independently or cumulatively. Unlike other parallel computing systems such as multi-core computing or distributed computing, the development of SIMD is a relatively cheap way of exploiting parallelism with emphasis on data-level parallelism. Generally speaking, SIMD systems can be divided into two types, vector-based and array-based systems.

In the vector-based system, it usually has one processor with a set of vector registers. When executing an instruction, data is loaded into one register which can store some fixed number of elements, and then the instruction is performed on some or all of the elements simultaneously. With the help of vector processing, it allows the processor to achieve better performance than traditional scalar processing. If the SIMD register size is 128-bit, it could get 4 times faster compared to the non-parallel processing when doing 4 pairs of 32-bit integer addition. Figure 2.1 shows the general idea about SIMD vector processing.

However, the vector-based SIMD processors are not well suitable for solving problems involving two or more dimensions. In the array-based system, the SIMD processors are used to deal with the data in large multi-dimensional arrays. Such a system would have a single control unit and a bunch of multiple processing elements (PEs) which are connected

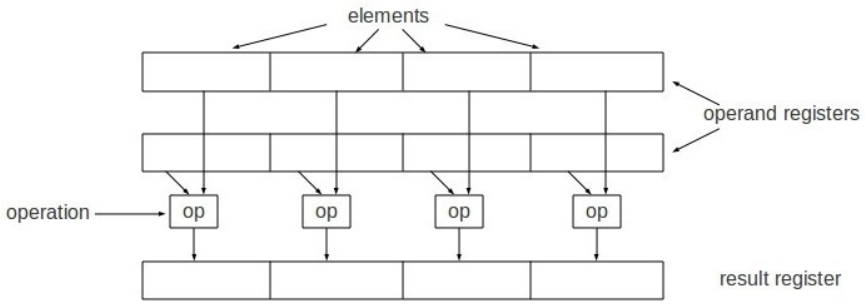

Figure 2.1: An Example of SIMD Vector Processing

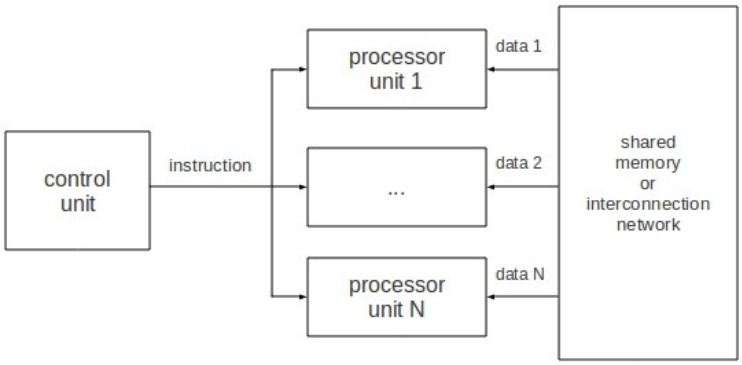

Figure 2.2: An Example of SIMD Array Processing

in shapes of multi-dimension. The instructions are distributed by the control unit to the PEs, and each PE receives the instructions and applies them on its own data stream. An example of SIMD array system is shown in Figure 2.2.

Historically, the early SIMD machines were array-based systems and mostly characterized by massively parallel processing-style supercomputers. The most important reason is that supercomputers are normally used to process and analyze a very large amount of data as in meteorological applications and physics simulations, hence, the SIMD on array-based processors with capability of processing multi-dimensional data efficiently were the best choice to accomplish the jobs. As the inexpensive Multiple Instruction Multiple Data (MIMD) approaches became more powerful later, the interest in SIMD array-based systems waned [7]. However, since desktop processors have become powerful enough in terms of high CPU clock rates, large register complements and advanced system bus designs to support real-time multimedia applications, the SIMD approach is now widely used in machines from

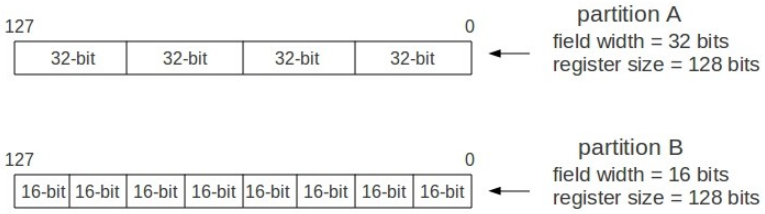

Figure 2.3: Examples of Two Different Partitions on a 128-bit Register

the desktop-computer market in a form of vector-based processing.

2.2 SIMD Within A Register

In this thesis, we target our portable SIMD programming model on the systems which support SIMD within a register (SWAR). Almost all of today's PCs and workstations are the SWAR-capable systems, in which SIMD instructions are executed across sections of a register.

The SWAR model has a very unique property that every register in it can be partitioned into fields and each field is independent from any other field. However, the partition is not a real physical partition but a logical view of partition on registers. For example, a register with 128 bits could be viewed as four 32-bit fields (in Figure 2.3) or eight 16-bit fields (in Figure 2.3) or other possible partitions. Each partition is independent and a SIMD operation in SWAR requires corresponding operand fields to be manipulated without interfering with adjacent fields.

As you see, this property of partition matches perfectly with the SIMD processing idea that an operation is applied on many data simultaneously, hence, the SWAR model is very suitable for supporting SIMD operations with a minimal requirement of hardware enhancements. Since SWAR doesn't restrict the size of each field in a register, we can explicitly set the precision for the field widths as desired when applying operations. More importantly, SWAR systems are designed in a way that there is no penalty for data crossing the logical boundaries inside a register, which provides room for inductive algorithms on emulating SIMD operations.

Unfortunately, the implementations of SWAR systems strongly depend on the architectures and the target services of the systems. Available operations are determined according

to the needs of applications especially the multimedia related programs, so the instruction sets are different between architectures, and even within the same architecture some operations are not supported or supported on just a few field widths.

In summary, the SWAR model with the feature of logical partition on registers and its diverse implementations over platforms, is the target for which our high performance and portable SIMD programming model is built.

2.3 Commercial SIMD Instruction Sets

Having a comprehensive understanding about current mainstream SWAR families in IT industry is very important for developing a good SIMD programming model over different kinds of platforms. In the past, the SWAR instruction sets were made to improve the performance of multimedia programs and were usually integrated into the existing architectures as new instruction sets using the SIMD processing paradigm.

Each instruction set was customized by individual microprocessor vendor for the sake of better supporting compute-intensive algorithms and applications on its host platform. Given the variety of microprocessors, each SWAR extension has a unique set of instructions and the supported SIMD operations vary widely. But, this doesn't mean SWAR instruction sets are totally different from each other. In fact, the underlying algorithms of many SWAR operations are equivalent for different platforms although the operations might differ in available field widths or performance. So it turns out that every SWAR instruction set has implementations which are similar or even identical to those of others.

Early SWAR instruction sets on a platform were normally limited to a few of instructions which only serve the most frequently used operations and applications for this particular architecture. Later extensions were often created to address some issues in previous versions and provide a wider range of SWAR related instructions. Thus, one SWAR instruction set might evolve for many generations and ultimately become more complete and powerful.

In this section, several typical SWAR instruction sets in current commodity processors are reviewed to show their capabilities and limitations and also to provide the guidance for designing and implementing our portable SIMD programming model.

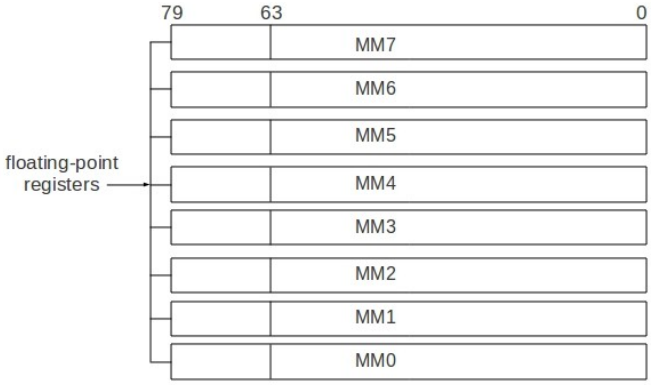

Figure 2.4: The View of MMX Registers

2.3.1 Intel MMX

MultiMedia eXtension (MMX) is a SWAR instruction set designed by Intel in 1996 to improve performance of multimedia and communication algorithms. It was the first major addition to the Intel Architecture-32 (IA-32) since the Intel 386^{TM}architecture [29]. The definition of MMX technology evolved from the Intel i860^{TM}architecture which was the industry's first general purpose processor for graphics rendering with the functionality of parallel computing on multiple adjacent data operands.

MMX adapts to SIMD approach by defining the packed data format for data representation which allows the input data to be processed simultaneously in small data fields such as 8-bit or 16-bit. There are eight 64-bit registers defined in MMX, known as MM0 through MM7. To maintain the compatibility with the IA-32 architecture of that time, those registers were not defined as a new set of registers but aliases of the existing IA-32 Floating-Point Unit (FPU) stack registers. MMX only uses the low 64 bits of each FPU register and sets the unused bits to be all ones to reduce confusion between a MMX data value and a valid floating-point value. Figure 2.4 shows the overview of MMX registers in the FPU registers. However, because the dual usage of the FPU registers doesn't allow the concurrent execution of both MMX code and floating-point code (meaning that the MMX code and floating-point code must be in separate code sequences), there is a mode switch cost if an application uses both codes.

However, MMX only provides integer operations to meet the requirement of integer math used in early graphical applications. The set of operations provides a relatively wide range of

support for the SIMD programming, which includes arithmetic logic operations, saturating arithmetic, fixed-point arithmetic and repositioning of data elements within a register. The design of MMX had limited it to contain new instructions specifically designed for audio, graphics and other multimedia applications. Although there were 57 new instructions added in MMX [45], the implementations are not available over consistent field widths and also lack full support for many types of operations, such as comparison operations. Moreover, many of the instructions have little application outside the multimedia domain. Thus, MMX is not a general purpose instruction set for a high level SIMD programming model.

For further information about MMX, readers can refer to [50] which provides an overview of the MMX instruction sets and also [32] which has detailed information about the MMX instructions including intrinsics and programming convention.

2.3.2 SSE Series

A few years after MMX was released, Intel introduced Streaming SIMD Extensions (SSE) in their Pentium III series processors in 1999. SSE instruction sets were subsequently extended in a series of versions including SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4a and SSE4.2. Intel was the main contributor designing all SSE generations except SSE4a which was created by AMD.

SSE

In contrast to the MMX extension, SSE added 70 new instructions, primarily dedicated to support single-precision floating-point data [56]. It also added a few integer instructions such as minimum or maximum instructions that work on MMX registers. In addition, SSE added a set of eight new independent registers named XMM0 through XMM7 which are 128-bit SIMD floating-point registers. In short, MMX instructions are still available in SSE using the 80-bit floating-point stack registers to process integer data while SSE operations process the XMM registers with four 32-bit single-precision floating-point values as its data representation.

SSE2

In 2001, SSE2 was released by Intel along with the initial version of Pentium 4 to extend the earlier SSE instruction set [32]. It migrated all the MMX instructions to operate on XMM registers allowing an application to work on both SIMD and scalar floating-point data

without the switch cost required between MMX instructions and floating-point operations. More importantly, SSE2 added a rich set of integer instructions which extends MMX functionality to 128-bit XMM registers. This gives SSE2 a significant advantage that it could execute instructions twice as fast as MMX in theory due to the doubled register size.

AMD soon added support for SSE2 in its Athlon 64 processors based on the AMD64 architecture. In AMD's implementation, it doubled the number of XMM registers to sixteen as XMM0 through XMM15. In 2004, Intel adopted these additional registers as part of their SSE2 implementation for the IA-64 architecture [2].

SSE3

The third iteration of SSE instruction sets of Intel is the SSE3, which was introduced with the Prescott revision of the Pentium 4 processors in 2004. In SSE3, the major improvement is that some instructions working horizontally in a register with floating-point values were added to simplify certain DSP and 3D graphics related algorithms [9]. A new instruction which does misaligned integer vector load was also included to achieve better performance on loading data across cache-line boundaries.

SSSE3

SSSE3, the fourth generation of the SSE series, was included with Core micro-architecture based Intel processors in 2006. It contains 16 new instructions compared to its precursor, most of which are signed magnitude arithmetical instructions and instructions working horizontally on XMM registers with packed integer data.

SSE4

As the last iteration in SSE family, SSE4 was officially announced in 2006, and became available in hardware in early 2007 for both Intel and AMD processors. SSE4 now has three variation, SSE4.1, SSE4a and SSE4.2 which contribute more than 50 instructions all together.

SSE4.1 is the major extension of SSE4 with 47 instructions added including some instructions that are not specific to the multimedia domain such as conditional copying or shuffle of elements from one location to another based on the bits in an immediate operand or a XMM register. Some other instructions are mainly about arithmetic operations on packed integer data and memory related operations.

SSE4.2 completed the SSE4 instruction set by adding 7 new instructions, most of which are comparison operations on packed explicit length strings.

SSE4a, introduced by AMD on the AMD K10 micro-architecture in 2007 [10], implements part of SSE4.1 instructions from Intel and adds its own 6 instructions for bit manipulation as well.

Evolving for years and reaching five generations in total, the series of SSE instruction sets have formed a powerful SWAR extension in the microprocessor industry. However, SIMD operations supported in SSE series are still limited to a few field widths, and there are only a few instructions which are outside multimedia applications. Although the SSE series is fully upward compatible with MMX, there are two different sets of programming intrinsics for MMX and SSE. This poses a challenge for developers to migrate MMX based programs to SSE based programs, which might require them to make significant revision to source code in order to adopt the SSE programming convention.

The initial goal and design of SSE instruction sets is given in [56]. Full descriptions of SSE instructions and programming model can be found in the IA-64 and IA-32 architecture software developer's manual [32]. Some detailed information about SSE4a was posted at the online AMD developer central [10].

2.3.3 ARM NEON

An advanced SIMD extension named NEON was introduced by ARM in their Cortex-A series processors to improve the performance of multimedia and signal processing algorithms such as 3D graphics, gaming and audio/speech processing. It has a comprehensive set of instructions and some of the instructions are shared with the ARM Vector Floating Point (VFP) extension. NEON supports 8-bit, 16-bit, 32-bit and 64-bit signed and unsigned integers and also 32-bit single precision floating point values naturally.

There are 32 featured 64-bit registers associated with NEON implementation, which can be accessed by both NEON and VFP processing units. Interestingly, NEON can view its register set in two different ways, one way is thirty-two 64-bit doubleword registers (D0-D31), the other is sixteen 128-bit quadword registers (Q0-Q15). In fact, the doubleworld registers and the quadword registers alias each other with the 64-bit registers D_{2*i} and D_{2*i+1} mapping against the same physical location of the register Q_i.

NEON can utilize both register views to process data, which means that data from

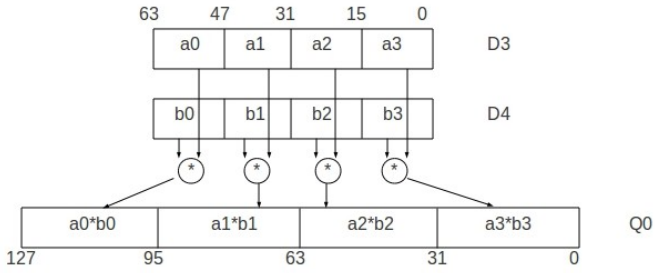

Figure 2.5: An Example of Multiplication in NEON

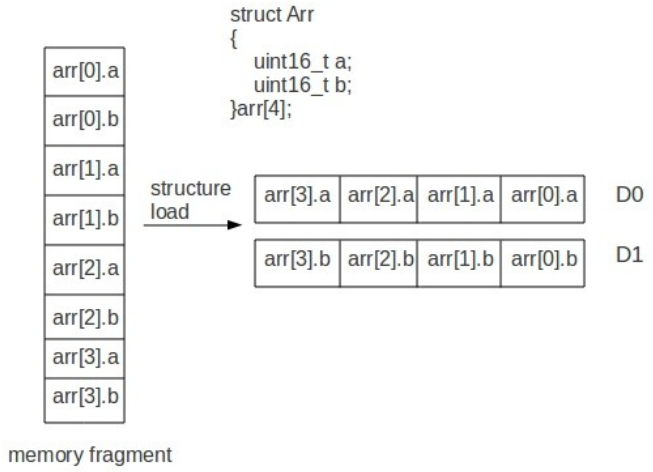

Figure 2.6: An Example of Structure Load in NEON

registers in different views could be accessed in the same instruction. With this particular property, instructions in NEON could have different size input and output registers and be able to promote or demote data elements in an operation. For example, in Figure 2.5, "$VMULL.S16\ Q0, D3, D4$" multiplies four pairs of 16-bit values from doubleworld registers D3 and D4 at a time and produces four 32-bit products in the 128-bit quadworld register Q0.

Another notable property of NEON is that, it not only has instructions which can load and store multiple data from or to a SIMD register, but also includes some instructions that can transfer complete data structures between several SIMD registers and memory location with interleaving and de-interleaving options. An example is shown in Figure 2.6.

There are many references about ARM NEON available. Some introduction on NEON

can be found in [16, 51]. The white paper titled "ARM NEON support in the ARM compiler" [15] discusses the compiler support for SIMD from the automatic recognition approach and also the perspective of the use of intrinsic functions. The assembler guide [14] describes all the instructions and the underlying assembly programming of NEON in detail.

2.3.4 AltiVec

AltiVec is a SIMD extension to the PowerPC architecture [6], developed by Motorola, Apple and IBM in late 90s. Apple was the main consumer of AltiVec until they switched to Intel x86 based processors in 2006 while Motorola had been the main manufacturer for providing AltiVec chips. IBM declined to be involved in AltiVec, instead, they have made the VMX which is similar to AltiVec out of their Power processors. AltiVec is now a trademark owned solely by Freescale, the former semiconductor products division of Motorola.

AltiVec expands the PowerPC architecture through the addition of a vector processing unit with thirty-two 128-bit registers built in. It supports 8-bit, 16-bit and 32-bit integers and also 32-bit floating point data. There are 164 new instructions added in AltiVec, providing a general set of instructions including intra and inter-element arithmetic instructions, intra and inter-element conditional instructions and some powerful repositioning instructions. Unlike the SSE series instructions that store results back into a source operand register, each instruction in AltiVec is a non-destructive instruction which can preserve the content of source operand after the instruction is executed. Furthermore, AltiVec has a powerful and flexible vector permute instruction which can extract the data elements from either of two operand registers and reposition them in the resulting register according to the parameters in yet another register. This allows for sophisticated manipulations on ordering data in a single instruction.

An introduction on AltiVec and the ways to vectorize code using AltiVec are shown in [26]. The programming environment manual of AltiVec technology [46] contains guide for assembler programmers and has detailed information about instructions. High-level programmers should refer to [47] which provides C/C++ programming interface for using the AltiVec instruction set.

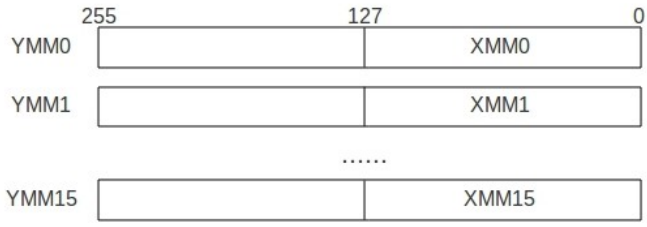

Figure 2.7: The Overview of AVX Registers

2.3.5 Intel AVX

In 2008, a new SIMD extension to Intel x86 architecture named Advanced Vector Extensions (AVX) was announced. AVX extends the previous Intel SIMD instruction sets such as MMX and SSE series by adding many new features to provide even better performance on multimedia applications and vector processing. The actual implementation came with Intel Sandy Bridge processors [3] in early 2011.

Compared to SSE series, AVX increases the size of its SIMD registers from 128 bits to 256 bits. The AVX registers named YMM0 through YMM15 are aliased over the initial 128-bit XMM registers of SSE series with the XMM registers as the lower 128-bit of the corresponding YMM registers, shown in Figure 2.7. Hence, AVX fully supports legacy SSE instructions including the SIMD data type used in SSE as well. AVX instructions operate on 8-bit, 16-bit, 32-bit, 64-bit and 128-bit integers as well as 32-bit and 64-bit floating point values.

One of the most important improvements made in AVX is that instructions are non-destructive, mostly with two source operands and a separate destination operand. Thus, the AVX instructions enable the preservation of the content in source operand after execution so that potential movements between registers as in the SSE series could be reduced. Besides, there are a few of four-operand instructions added to allow smaller and faster code written in AVX.

As another significant change made in AVX, a new extension coding scheme (VEX) has been designed to encode instructions and make future additions easier. The VEX is a prefix encoding with space of two or three bytes, which was designed to simplify the current and future Intel x86 based instruction encoding and allow more instructions to be included and encoded. It extends the older SSE instructions by adding a VEX prefix for accessing the

new 256-bit registers and three-operand forms. Unfortunately, the current implementation of AVX only allows the integer instructions to operate on the lower half of YMM registers with options to zero or retain the higher half parts. However, there are some instructions which can move or extract data of either the high or low 128 bits of the YMM registers, making it possible to achieve parallel processing on 256 bits.

According to Intel's documentation [31], it is believed that AVX is designed to support registers with 512 or 1024 bits in the future. AVX2, the next immediate generation of AVX, will ensure every legacy SSE instruction has a VEX form to operate on the entire 256 bits of the YMM registers naturally. In [43], it introduces the overall design and goal of AVX. The Intel AVX programming reference [31] contains specific information about instructions and programming model in AVX and AVX2. Besides, a AVX C/C++ intrinsics emulation provided by Intel [1] enables developers to program with Intel AVX intrinsics on the machines which do not support AVX.

2.3.6 Others

Because of their wide usage and comprehensive support of instructions, several representative SIMD extensions on various commodity microprocessors are reviewed in above sections. Nevertheless, there are also other extensions which play important roles in the parallel processing area, which either contribute as the basis of the recent SIMD extensions or provide functionality for some specific applications beside multimedia.

Intel i860, also known as 80860, was the first microprocessor having vector processing instructions, introduced by Intel in 1989. It has a graphics unit using FPU registers as SIMD registers to process instructions for 3D graphical applications. The development of i860 was the groundwork for Intel x86 based system to support SIMD instructions, and it influenced the MMX functionality heavily in Intel Pentium processors later [50].

In 1994, HP developed its initial version of Multimedia Acceleration Extension named MAX-1 on 32-bit PA-RISC 1.1 architecture PA-7100LC processors [42, 40]. It uses a very small set of SIMD instructions and enables real-time video decompression without the need of special hardware enhancements [41]. Two years later, with the 64-bit PA-RISC 2.0 architecture introduced, HP extended the earlier MAX-1 instruction set to create a new set of SIMD extension called MAX-2 which could operate on the 64-bit registers with more instructions available.

Sun Microsystems developed a SIMD extension called Visual Instruction Set (VIS) for

SPARC V9 microprocessors, and the first implementation of VIS was released along with UltraSPARC microprocessor in 1995 [55, 54]. The second generation, VIS-2 was implemented in the UltraSPARC III later as an enhancement. VIS uses the 64-bit floating point registers to hold data which is similar to the Intel MMX design. The instructions of VIS are primarily for visual and graphical applications such as format conversions between pixel data and 16/32-bit values and some arithmetic operations on data with small bits.

Since Intel MMX doesn't support SIMD instructions on floating point values, AMD added a new extension named 3DNow! on AMD K6-2 processors in 1998 to enable arithmetic operations on single precision floating point values. In the first generation of Athlon processors, AMD introduced the enhanced 3DNow! as the extension to the older 3DNow! by adding many new 3DNow! and MMX instructions [12]. Couple years after enhanced 3DNow! was released, starting from the AMD Athlon XP, 3DNow! Professional has been used as the name for a combination of 3DNow! technology and SSE instructions sets.

2.4 Related Work

With the rapid growth of SIMD instruction sets on commodity microprocessors, quite a bit of research has been done on these SIMD extensions, expanding from building a portable SIMD programming model over platforms to using SIMD instructions to auto-vectorize or optimize the sequential programs [11, 49]. To my knowledge, the research can be divided into two categories, the compiler technology for SIMD instructions and the library support for SIMD programming.

Compiler Technology for SIMD Instructions

Fisher's work on compiling for SIMD within a register [22] proposed a general-purpose SIMD programming model with a C-like module language and an associated compiler named Scc. Based on the C language, the module language adopted many base data types from C and also added a vector type to support SIMD programming. Thus, it allows programmers to describe the SWAR data types and algorithms in a portable manner. The associated compiler Scc supports parsing the code written in the module language and implementing the specified algorithms into C code based on the capabilities of the target architecture. According to Fisher's PhD thesis [23], this model supports code generation on AltiVec, 3DNow!, MMX and SSE.

While Fisher's work was the groundwork for portable SIMD programming, the implementations generated by his model did not achieve good performance in some circumstances, particularly when the application relied on many non-built-in operations. As shown in his thesis [23], the code generated by Scc targeting MMX did not gain any speedup but slowed down the performance at a ratio between 0.4 and 0.8. Although there were many compiler techniques added into the model to optimize the generated implementations for non-built-in operations, the study on emulation techniques for non-built-in operations was still insufficient which limits the further use of the model for a general-purpose SIMD programming.

Bocchino and Adve presented a virtual vector instruction set called Vector LLVA for media processing [34]. The Vector LLVA supports arbitrary-length vectors for streaming processors and also fixed-length vectors for processors equipped with multimedia/SWAR extensions like AlitVec and SSE. It defines a relatively rich set of instructions covering general arithmetic operations, vector-memory operations and data movement operations. This vector programming model was also attached with translators which are able to translate the Vector LLVA code into C code that use the intrinsics and programming conversions of three target architectures (Motorola RSVP, AltiVec and Intel SSE2) individually.

However, Vector LLVA model requires developers to write and tune applications at an instruction level which is challenging work for source-language level programmers. And the performance of implementations generated by Vector LLVA was not well balanced, sometimes slightly better than hand-tuned implementations while sometimes worse. Besides, the Vector LLVA has a fixed set of instructions and its translators only support those built-in instructions, hence, it is not likely to allow people to add new instructions or operations for their own purpose.

Liquid SIMD [20] is another compiler technology for supporting programming and applications on multiple SIMD extensions. It has both compiler support and dynamic translation to decouple the instruction set architecture from the implementation of a SIMD accelerator. The way it achieves portability relies on two stages. First, it identifies SIMD instructions and compiles them into a virtualized SIMD schema using the scalar instruction set of a processor. Then, during program execution, it uses a light-weight dynamic translation engine to convert these scalar instructions back into SIMD instructions that can be executed on an arbitrary SIMD accelerator. However, more experiments on other platforms are needed to be done as Liquid SIMD was only demonstrated on the ARM platform.

In addition to the compiler technology for portable SIMD programming, there are some

other techniques which could parallelize parts of a sequential program automatically. The idea is to identify the potential parallelism spots in source codes, especially in the loops and basic blocks, and try to replace the traditional instructions by SIMD instructions so that certain speedups could be gained. In [36], Larsen and Amarasinghe presented a novel way of viewing parallelism in multimedia applications called Superword Level Parallelism (SLP). They also developed a compiler for detecting SLP targeting basic blocks rather than loop nests. In Larsen's PhD thesis [35], he worked further on SLP technology and made two major improvements, one is the effective management of memory alignment for compiling short-vector instructions, the other is a selective vectorization technique for balancing computation across scalar and vector resources in a processor based on software pipelining. Shin also proposed a compiler framework based on SLP to exploit parallelism automatically in sequential programs [53]. Shin's compiler has a number of optimizations compared to Larsen's, it extends SLP in the presence of control flow constructs to increase the applicability of SLP and treats the SIMD register file as compiler-controlled cache to avoid unnecessary memory access.

Library Support for SIMD Programming

In his PhD thesis [52], Rojas implemented a set of multimedia macros for portable optimized programs. Each macro has an individual implementation for each supported architecture, and all macros together provide a unique interface for high-level programming. The author developed the implementation for each macro on a certain architecture by manually trying different algorithms and picking the best one with the criteria in terms of instruction count and execution time in CPU cycles. The set of macros supports 8-bit, 16-bit and 32-bit integers and 32-bit floating point values on AltiVec, Intel MMX/SSE/SSE2 and TriMedia TM 1300. To gain the portability for SIMD programming, Rojas's macros actually slow down the performance as the experimental results in his work shown that the performance based on the portable macros lost around 12% for a certain algorithm compared to a specific hand-tuned implementation.

To encourage the development of applications in compute-intensive areas on Mac OS X or later computers, Apple has created an accelerate framework [13, 28] for high performance numerical computations. The framework's implementations are highly optimized for AltiVec and Intel SSE platforms by Apple to fully take advantage of those platforms'

capabilities especially the SIMD processing units. There are two sub-frameworks in the accelerate framework, vecLib and vImage. General numerical operations including addition, multiplication and so forth for both scalar and vector data are included in vecLib, while vImage provides a set of image processing routines. This accelerate framework provides developers a set of abstract programming interfaces in C to allow them to use the vector processing resources of the target platforms without worrying about the low level difference between platforms.

libSIMD [4] is a open-source mathematical library using SIMD processing capabilities of 3DNow! and Intel SSE processors to accelerate some commonly used algorithms. The library provides function interfaces in C for programmers and only supports 32-bit and 64-bit precision floating point values. Most implementations in the libSIMD were written in assembly codes to get as much performance as possible. Similarly, SIMDx86 [8] is another optimized math library mostly written in assembly codes for graphics applications especially the 3D games engines and 3D visualizations and so on. SIMDx86 provides C function routines that can work on 3DNow!/Enhanced 3DNow!, Intel MMX and SSE/SSE2 platforms.

Instead of building a C library for high-level SIMD programming, some work has been done to provide C++ libraries using template metaprogramming mechanism. EVE [21], an object oriented SIMD library designed for AltiVec processors, which was built upon a template metaprogramming engine to support a STL-like programming interface for developers to write efficient applications compared to the traditional C libraries. Provided by pixelglow software, a C++ library called macstl [5] was distributed for generic SIMD programming to Macintosh and Windows platforms. In macstl, there is a class named vec for manipulating vector data, which supports a standard vector initialization syntax in a C++ template manner and includes a common programming interface for developers to write fast SIMD applications that work with AltiVec and Intel MMX/SSE/SSE2/SSE3 instruction sets. Due to the strong demand of fast numerical applications, Boost just announced in early 2011 that they are going to add a SIMD library [27] to support vector processing on various architecture families including Intel x86, PowerPC and ARM.

Chapter 3

Model Definition

3.1 Objectives

As shown in the related work of Chapter 2, many compilers made for SIMD instructions have a problem that the performance usually drops off compared to a specific hand-tuned implementation. With these compilers, programmers are normally required to play with the low-level instructions directly when writing applications. In this thesis work, instead of building a virtual SIMD instruction set and an associated compiler, we decided to create a library support for the portable SIMD programming.

The goal of this library support is to enable portable programming among diverse SWAR architectures, provide a clean and uniform interface for ease of programming, and more importantly, ship the same and even better performance as the specific architecture-dependent implementations.

Portability

Supporting programming over platforms is a challenging task. First of all, there are many operations that have to be implemented as for a SIMD library, and even worse, an operation usually has several versions depending on the operating field width. Thus, it is impractical and very time-consuming for human-beings to develop each library for every architecture by hand. Secondly, even if hand-written libraries are acceptable, finding the best implementations of them involves lots of human efforts and expertise knowledge in computing. Besides, a library is often released as a set of fixed interfaces so that adding a new operation or function which incorporates well with the library is difficult.

The way we achieve portability mainly relies on two aspects. First and foremost, we do not want the library in our model to be hand-written because that requires huge efforts and is not flexible for further optimization. Therefore, we have developed a framework which could generate the objective libraries for various architectures automatically in C++. In the next place, the framework we have built provides an interface that allows users to add new operations easily and improve the library's implementations with a few effort.

Performance

Performance is a critical concern when designing libraries. In particular, a portable library should not trade too much performance out in order to support portability. However, there are many factors that affect the performance, such as cache algorithm, instruction pipeline interactions and memory access latency and so forth. Nevertheless, an optimized library must take full advantage of the target's capabilities, especially its instruction set, to keep the number of instructions of an application as less as possible. To accomplish that, our model has a mechanism to optimize the implementation of each operation in terms of minimizing instruction count. Generally speaking, it is expected our model out-performs the hand-optimized libraries.

3.2 Inductive Doubling Principle

Before designing what SWAR operations our model wants to support and showing how we support them, it's better to first look at some properties of SWAR operations in general.

- Any of SWAR operations applies the same operation on partitioned fields of a register simultaneously with each field has the same width.
- The size of fields is normally 16 bits, 32 bits and other power-of-2 bit widths; these work well with the frequently used data types like *char* and *int* in most programming languages.
- In current available SWAR instruction sets on commodity processors, an operation might not have full support for all power-of-2 field widths, i.e., the operation lacks implementations for some field widths.

When it comes to implement a SWAR operation for a missing field width, it is very natural to use the implementation of the same operation on other close field widths to do the

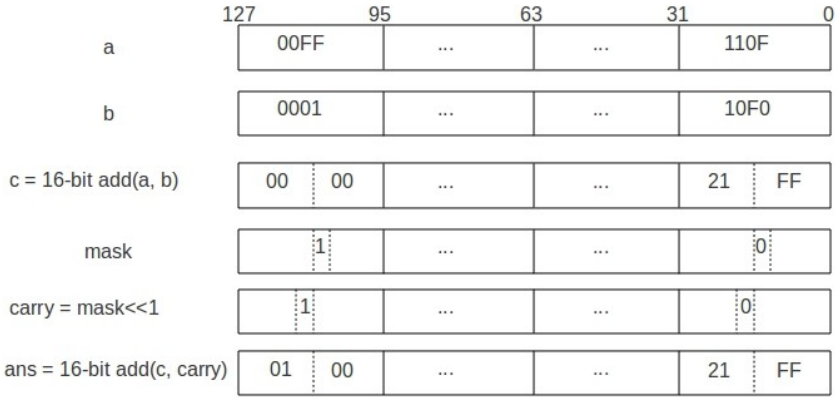

Figure 3.1: An Example of Using 16-bit Addition to Emulate 32-bit Addition

job. In [19], Dr. Cameron has proposed an inductive doubling instruction set architecture (IDISA) which nicely captures the idea of simulating SWAR operations for missing field widths based on the observation that implementing a SWAR operation on a certain data size could use the algorithm for the operation on halved data size and vice versa.

For example, we could use an addition operation on 8 pairs of 16-bit numbers simultaneously along with some shifting and combination work to get an operation of capability to perform addition on 4 pairs of 32-bit numbers at a time. Figure 3.1 gives such an example.

Conversely, the addition on 4 pairs of 32-bit numbers can also be used to simulate the 8 pairs of 16-bit numbers addition. The idea is shown in Figure 3.2.

Overall, inductive doubling principle provides a general approach for emulating a SWAR operation for missing field widths within the same operation family. However, people could always find implementations for a SWAR operation in a totally different manner instead of the inductive doubling fashion. An extreme example would be using bit-wise xor to accomplish the 1-bit addition between two SIMD registers. That being said, inductive doubling principle is not the entire methodology of our model but a basic foundation which provides guidance for making the library support in our model.

3.3 Integer Operations or Floating Point Operations

To best serve the higher level applications using our library support, we have defined a set of well-chosen operations in our model. The operations are carefully selected in order to

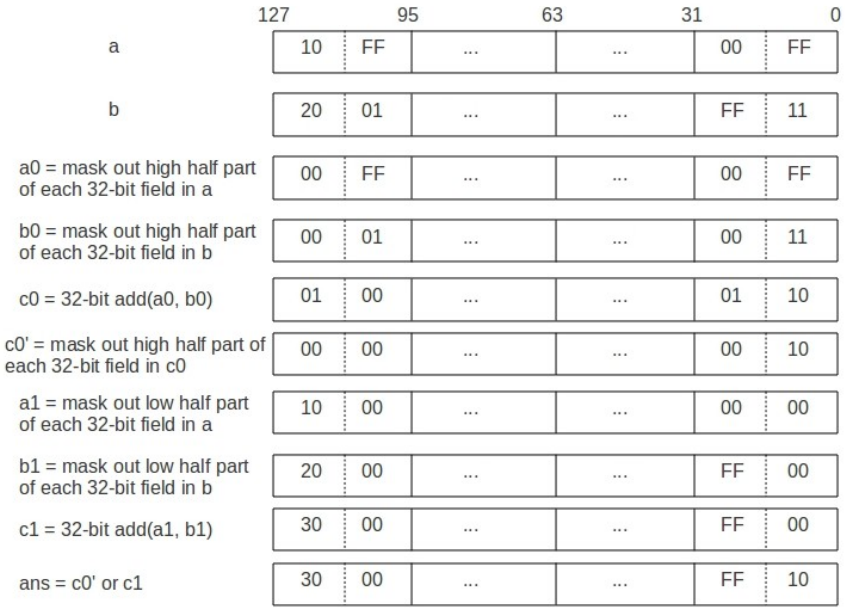

Figure 3.2: An Example of Using 32-bit Addition to Emulate 16-bit Addition

fully exploit the features of vector parallelism and supply SIMD functionalities as well as possible.

Unlike many other portable SIMD programming models which primarily serve the fast mathematic applications and support operations on vectors of floating point values, the current operations defined in our model are intended for working on vectors of integer values. The reasons why we decided to support integer operations not floating point operations are as follows.

- A floating point data is often either single precision (32-bit) or double precision (64-bit) in terms of binary representation. Usually, operations on the floating point data can not be simulated in the inductive doubling manner. For example, it is very difficult to get the result with high enough precision by using two 32-bit floating point values to act like a 64-bit floating point value when applying SIMD operations. However, that is not a problem for integer numbers as we could easily use two 32-bit integers to hold a 64-bit integer without losing any precision.

- If we want to define operations for floating point values, there would be two independent versions for each operation with one works for 32-bit floating point values while

the other works for 64-bit floating point values. This essentially requires two different SIMD data types and makes our model quite complex.

- In supporting only the SWAR integer operations, there is still an extensive application space. Not only are the traditional media processing applications based on integer operations, but also are many other applications such as high-speed XML parsing using parallel bit stream technology [17, 18].

At present, we define integer operations in the model as built-in operations because we mainly target our model on some specific applications where integer operations are the dominant operations. However, our portable model is designed as an improvable model so that users can define their own operations even like floating point operations in the model easily if they really want to.

3.4 Defined Operations

Every operation in our model is defined to work on a SIMD register with a number of fw-bit fields, and the width of all fields is a power-of-2, i.e., $fw = 2^k$ for $0 \leq k \leq K$, where the size of the SIMD register is $sz = 2^K$ bits. Among the commercial SWAR instruction sets, the value of K is 6 for Intel MMX with 64-bit registers, 7 for Intel SSE series and ARM NEON with 128-bit registers, and 8 for the Intel AVX with 256-bit registers.

If a certain field width fw is specified, a register r can be viewed as sz/fw fields with fields indexed r_0 through $r_{sz/fw-1}$, in which $r_{sz/fw-1}$ is the highest field and r_0 is the lowest one. In Figure 3.3, it shows the view of numbering. For an operation op working on two register a, b with fw-bit fields, we denote it as $op_{fw}(a, b)$.

So far, there have been more than 60 operations defined in the model with all of them can be grouped into six categories: logic operations, vertical operations, horizontal operations, expansion operations, field movement operations and bitblock operations. Before going into the detailed description of operations, we first present several virtual functions in Table 3.1 to better explain the functionalities of defined operations.

3.4.1 Logic Operations

A logic operation performs bit-wise operation on one or more registers and is in the form of $r = simd\_op(a)$ or $r = simd\_op(a, b)$, where a and b are input registers. There are six logic

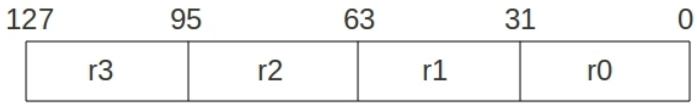

Figure 3.3: The View of Field Numbering on a 128-bit Register

Function Name	Description
$Signed(r_i)$	return the i-th field of register r as a signed integer
$Unsigned(r_i)$	return the i-th field of register r as an unsigned integer
$High_{fw}(r_i)$	select the high half part of the i-th field of register r, i.e., $r_i >> (fw/2)$
$Low_{fw}(r_i)$	select the low half part of the i-th field of register r, i.e., $r_i \& ((1 << (fw/2)) - 1)$
$NumberOfField_{fw}(r)$	return the number of fields in r, i.e., sz/fw
$UnsignedSaturation_{fw}(r_i)$	if $r_i < 0$, return 0 else if $r_i \leq 2^{fw/2} - 1$, return r_i else, return $2^{fw/2} - 1$
$SignedSaturation_{fw}(r_i)$	if $r_i < -2^{fw/2-1}$, return $-2^{fw/2-1}$ else if $r_i \leq 2^{fw/2-1} - 1$, return r_i else, return $2^{fw/2-1} - 1$
$Index_{fieldNum}(mask, i)$	if $fieldNum = 2$, return $1\&(mask >> i)$ else if $fieldNum = 4$, return $3\&(mask >> (i * 2))$ else if $fieldNum = 8$, return $7\&(mask >> (i * 3))$ else if $fieldNum = 16$, return $15\&(mask >> (i * 4))$

Table 3.1: Virtual Functions Used to Better Explain the Operations' Functionalities

Operation	Meaning	Description
simd_nor	perform bit-wise *or* on a, b and then perform bit-wise *not* on the result	$t = \neg(a \mid b)$
simd_not	perform bit-wise *not* on a	$t = \neg a$
simd_andc	perform bit-wise *and* on a, $\neg b$	$t = a \wedge (\neg b)$
simd_or	perform bit-wise *or* on a, b	$t = a \vee b$
simd_and	perform bit-wise *and* on a, b	$t = a \wedge b$
simd_xor	perform bit-wise *xor* on a, b	$t = a \oplus b$

Table 3.2: The Logic Operations

operations provided in the model, In Table 3.2, it describes all these logic operations.

3.4.2 Vertical Operations

The vertical operations refer to the operations which perform on vertically aligned fields between two different registers. They operate on operand registers with fw-bit fields and produce a register with fw-bit fields as result. Many common operations fall into this category such as addition, subtraction and so on. The general computing logic of vertical operations is shown in Figure 2.1.

Depending on the number of arguments, vertical operations can be further divided into four different sub-categories, the vertical bitmask constants, the vertical unary operations, the vertical binary operations and the vertical ternary operations.

The Vertical Bitmask Constants

Only two bitmask operations are defined in the vertical operations, they are *himask* and *lomask* operations. Both of them do not have any argument in addition to the field width information, and they can be denoted in the form of $t = vertical\_op_{fw}()$, where t is a register with a certain bit pattern. Table 3.3 shows the two operations.

The Vertical Unary Operations

These operations are in the form of $t = vertical\_op_{fw}(a)$, where fw is the field width, a is the only argument which can be either a register or an immediate value, and t is the result register. Table 3.4 lists all the vertical unary operations with detailed description.

Operation	Meaning	Description
himask	return a register with the high half part of each field all set to be 1 while the low half part of each field all set to be 0	$t_i = ((1 << (fw/2)) - 1) << (fw/2)$
lomask	return a register with the high half part of each field all set to be 0 while the low half part of each field all set to be 1	$t_i = (1 << (fw/2)) - 1$

Table 3.3: The Vertical Bitmask Operations

Operation	Meaning	Description
abs	calculate the absolute value of each field	$t_i = -a_i$ if $Signed(a_i) < 0$ else a_i
neg	negate each field	$t_i = -Signed(a_i)$
add_hl	add the high and low half parts of each field	$t_i = High_{fw}(a_i) + Low_{fw}(a_i)$
xor_hl	bitwise xor the high and low half parts of each field	$t_i = High_{fw}(a_i)$ xor $Low_{fw}(a_i)$
popcount	bit counting in each field	$t_i =$ the number of 1 bits in a_i
ctz	count trailing zeros in each field	$t_i =$ the number of consecutive 0 bits in a_i counting from the right
constant	return a register with every field set to be the same value specified in a, where a is an immediate value	$t_i = a$

Table 3.4: The Vertical Unary Operations

Operation	Meaning	Description
add	addition on the corresponding fields of a and b	$t_i = a_i + b_i$
sub	subtraction on the corresponding fields of a and b	$t_i = a_i$ - b_i
mul	multiplication on the corresponding fields of a and b	$t_i = a_i$ * b_i
eq	check equality on the corresponding fields of a and b	$t_i = (1 << fw) - 1$ if $a_i == b_i$ else 0
gt	check signed greater than on the corresponding fields of a and b	$t_i = (1 << fw) - 1$ if $Signed(a_i) > Signed(b_i)$ else 0
ugt	check unsigned greater than on the corresponding fields of a and b	$t_i = (1 << fw) - 1$ if $Unsigned(a_i) > Unsigned(b_i)$ else 0
lt	check signed less than on the corresponding fields of a and b	$t_i = (1 << fw) - 1$ if $Signed(a_i) < Signed(b_i)$ else 0
ult	check unsigned less than on the corresponding fields of a and b	$t_i = (1 << fw) - 1$ if $Unsigned(a_i) < Unsigned(b_i)$ else 0
max	get maximum values from the corresponding fields of a and b	$t_i = a_i$ if $Signed(a_i) > Signed(b_i)$ else b_i

Table 3.5: The Vertical Binary Operations Part I

The Vertical Binary Operations

Vertical binary operations are in the form of $t = vertical\_op_{fw}(a, b)$, where fw is the field width, a is the operand register and b can be either an operand register or an immediate value, and t is the result register. The vertical binary operations are organized in Table 3.5 and Table 3.6.

The Vertical Ternary Operations

There is only one vertical ternary operation defined in the model, the ifh operation. We denote it as $t = vertical\_ifh_{fw}(a, b, c)$, where fw is the field width, a, b and c are the operand registers, and t is the result register. Table 3.7 explains this operation in details.

Operation	Meaning	Description
umax	get unsigned maximum values from the corresponding fields of a and b	$t_i = a_i$ if $Unsigned(a_i) > Unsigned(b_i)$ else b_i
min	get minimum values from the corresponding fields of a and b	$t_i = a_i$ if $Signed(a_i) < Signed(b_i)$ else b_i
umin	get unsigned minimum values from the corresponding fields of a and b	$t_i = a_i$ if $Unsigned(a_i) < Unsigned(b_i)$ else b_i
sll	shift each field in a_i left logical by the number of bits specified in b_i	$t_i = a_i << b_i$
srl	shift each field in a_i right logical by the number of bits specified in b_i	$t_i = Unsigned(a_i) >> b_i$
sra	shift each field in a_i right arithmetic by the number of bits specified in b_i	$t_i = Signed(a_i) >> b_i$
slli	shift each field in a_i left logical by the number of bits specified in b, where b is an immediate value	$t_i = a_i << b$
srli	shift each field in a_i right logical by the number of bits specified in b, where b is an immediate value	$t_i = Unsigned(a_i) >> b$
srai	shift each field in a_i right arithmetic by the number of bits specified in b, where b is an immediate value	$t_i = Signed(a_i) >> b$

Table 3.6: The Vertical Binary Operations Part II

Operation	Meaning	Description
ifh	select fields from either b or c based on the highest bit of the corresponding fields in a	$t_i = b_i$ if the highest bit of a_i is set else c_i

Table 3.7: The Vertical Ternary Operations

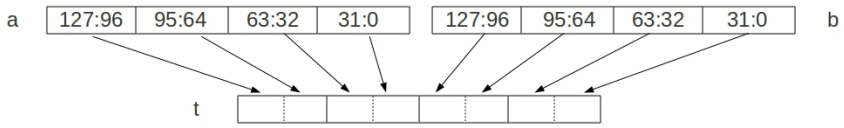

Figure 3.4: The Computing Logic of Horizontal Operations on 32-bit fields

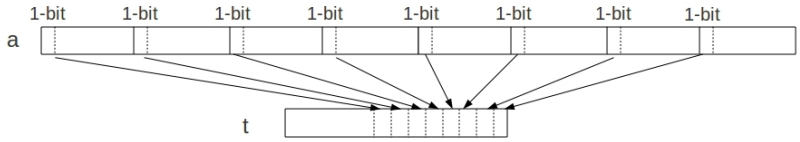

Figure 3.5: The Computing Logic of Horizontal Signmask Operation

3.4.3 Horizontal Operations

The horizontal operations accept one or two input registers and pack the fields of two registers into a single result register or an ordinary integer value. The process of packing a fw-bit field might pack one or more bits of the field, the high half or the low half of the field or even the entire field under certain mechanisms such as saturation. The computing logic of horizontal operations on 32-bit fields is shown in Figure 3.4.

Depending on the number of arguments, horizontal operations can be further divided into two different sub-categories, the horizontal unary operations and the horizontal binary operations.

The Horizontal Unary Operations

Only one horizontal unary operation has been defined in the model, which is called *signmask* operation. We denote it as $t = horizontal\_signmask_{fw}(a)$, where a is an operand register and t is an integer number. The Table 3.8 and Figure 3.5 below show the idea of *signmask*.

Operation	Meaning	Description
signmask	pack together the highest bit of each field in a and return the result as an integer	see Figure 3.5

Table 3.8: The Horizontal Unary Operations

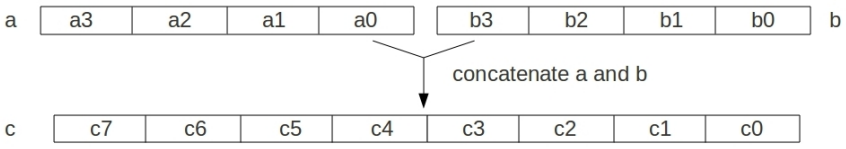

Figure 3.6: The Concatenation of Two Operand Registers

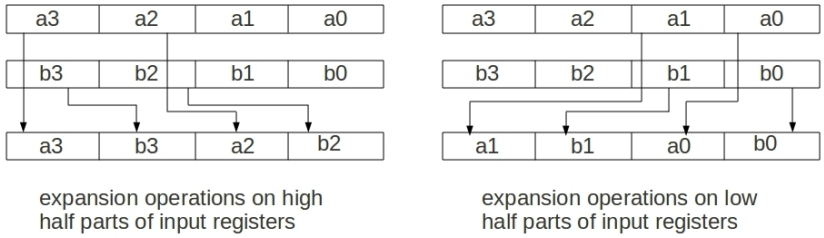

Figure 3.7: The Computing Logic of Expansion Operations

The Horizontal Binary Operations

Horizontal binary operations are in the form of $t = horizontal\_op_{fw}(a, b)$, where fw is the field width, a and b are input registers, and t is the result register. We use c as the concatenation of a and b for better illustration of horizontal binary operations, see Figure 3.6. The horizontal binary operations are shown in Table 3.9 and Table 3.10.

3.4.4 Expansion Operations

The expansion operations take one or two operand registers and use only half number of the fields from each input register to get a single result register with the same width as the operand registers. Essentially, these operations double the size of data fields and that's also why they are called expansion operations. The general computing logic of expansion operations is shown in Figure 3.7.

Depending on the number of arguments, expansion operations can be further divided into two different sub-categories, the expansion unary operations and the expansion binary operations.

Operation	Meaning	Description
add_hl	pack the sums between the high halves and the low halves of each field of c into t	$t_i[fw-1:fw/2] = High_{fw}(c_{2*i+1}) + Low_{fw}(c_{2*i+1})$ $t_i[fw/2-1:0] = High_{fw}(c_{2*i}) + Low_{fw}(c_{2*i})$
min_hl	pack the minimum between the high halves and the low halves of each field of c into t	$t_i[fw-1:fw/2] = High_{fw}(c_{2*i+1})$ if $Signed(High_{fw}(c_{2*i+1})) < Signed(Low_{fw}(c_{2*i+1}))$ else $Low_{fw}(c_{2*i+1})$ $t_i[fw/2-1:0] = High_{fw}(c_{2*i})$ if $Signed(High_{fw}(c_{2*i})) < Signed(Low_{fw}(c_{2*i}))$ else $Low_{fw}(c_{2*i})$
umin_hl	pack the unsigned minimum between the high halves and the low halves of each field of c into t	$t_i[fw-1:fw/2] = High_{fw}(c_{2*i+1})$ if $Unsigned(High_{fw}(c_{2*i+1})) < Unsigned(Low_{fw}(c_{2*i+1}))$ else $Low_{fw}(c_{2*i+1})$ $t_i[fw/2-1:0] = High_{fw}(c_{2*i})$ if $Unsigned(High_{fw}(c_{2*i})) < Unsigned(Low_{fw}(c_{2*i}))$ else $Low_{fw}(c_{2*i})$

Table 3.9: The Horizontal Binary Operations Part I

Operation	Meaning	Description
packl	pack the low halves of each field of c into t	$t_i[fw-1:fw/2] = Low_{fw}(c_{2*i+1})$ $t_i[fw/2-1:0] = Low_{fw}(c_{2*i})$
packh	pack the high halves of each field of c into t	$t_i[fw-1:fw/2] = High_{fw}(c_{2*i+1})$ $t_i[fw/2-1:0] = High_{fw}(c_{2*i})$
packus	pack the entire each field of c with unsigned saturation into t	$t_i[fw-1:fw/2] = UnsignedSaturation_{fw}(c_{2*i+1})$ $t_i[fw/2-1:0] = UnsignedSaturation_{fw}(c_{2*i})$
packss	pack the entire each field of c with signed saturation into t	$t_i[fw-1:fw/2] = SignedSaturation_{fw}(c_{2*i+1})$ $t_i[fw/2-1:0] = SignedSaturation_{fw}(c_{2*i})$

Table 3.10: The Horizontal Binary Operations Part II

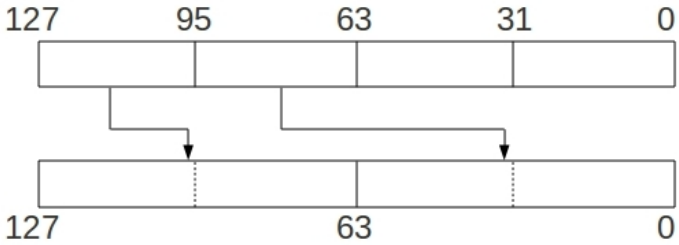

Figure 3.8: The Computing Logic of Expansion Unary Operations

Operation	Meaning	Description
signextendh	sign-extend each of the high half fields of a to get t	see Figure 3.8
signextendl	sign-extend each of the low half fields of a to get t	see Figure 3.8
zeroextendh	zero-extend each of the high half fields of a to get t	see Figure 3.8
zeroextendl	zero-extend each of the low half fields of a to get t	see Figure 3.8

Table 3.11: The Expansion Unary Operations

The Expansion Unary Operations

Expansion unary operations are in the form of $t = expansion\_op_{fw}(a)$, where fw is the field width, a is the only input register and t is the result register. Table 3.11 lists all the operations. For better understanding the expansion unary operations, Figure 3.8 gives a explicit demonstration on how they works.

The Expansion Binary Operations

Expansion binary operations are in the form of $t = expansion\_op_{fw}(a, b)$, where fw is the field width, a and b are input registers, and t is the result register. In Table 3.12, it describes all the expansion binary operations.

3.4.5 Field Movement Operations

The field movement operations are the operations which could move, extract or rearrange the fields of operand registers while leaving the content of fields unchanged. They can be

Operation	Meaning	Description
mergeh	merge the high half fields of a and b to get t	see Figure 3.7
mergel	merge the low half fields of a and b to get t	see Figure 3.7
multh	multiply the high half fields of a and b to get t	see Figure 3.7
multl	multiply the low half fields of a and b to get t	see Figure 3.7

Table 3.12: The Expansion Binary Operations

further divided into three sub-categories, field movement binary operations, field movement ternary operations and field movement pattern fill operations.

The Field Movement Binary Operations

Field movement binary operations are in the form of $t = field\_movement\_op_{fw}(a, b)$, where fw is the field width, a is an input register and b is either an register or an immediate value, and t is the result which could be either an register or an integer number depending on the operations. Table 3.13 shows all these operations.

The Field Movement Ternary Operations

There are two field movement ternary operations available in the model, which are *dsrli* and *dslli* operations. They are in the form of $t = field\_movement\_op_{fw}(a, b, c)$, the first two arguments are input registers, while the last one is an immediate value. Both of them apply shifting on the concatenation of two operand registers and return a single register with the width as the operand registers. Figure 3.9 shows the idea and Table 3.14 describes the operations.

The Field Movement Pattern Fill Operations

These pattern fill operations are mainly used to fill the fields of the result register with some values. They are in the form of $t = field\_movement\_op_{fw}(v_1, ..., v_i, ...)$, where v_i is an integer. Table 3.15 lists all these operations.

Operation	Meaning	Description
splat	broadcast the b-th field of a into each field of t, where b is an immediate value	$t_i = a_b$
slli	shift fields of a left logical by the number of fields specified in b, where b is an immediate value	$t_i = a_{i-b}$ if $i - b \geq 0$ else 0
srli	shift fields of a right logical by the number of fields specified in b, where b is an immediate value	$t_i = a_{i+b}$ if $i+b < fieldNum$ else 0, where $fieldNum = NumberOfField_{fw}(a)$
shufflei	shuffle the fields of a based on the mask b, where b is an immediate value	$t_i = a_j$ where $j = Index_{NumberOfField_{fw}(t_i)}(b, i)$
shuffle	shuffle the fields of a based on the masks specified in b, where b is an operand register	$t_i = a_{b_i}$ if $Signed(b_i) \geq 0$ else 0
extract	extract the b-th field of a and return it as an unsigned integer	$t = Unsigned(a_b)$

Table 3.13: The Field Movement Binary Operations

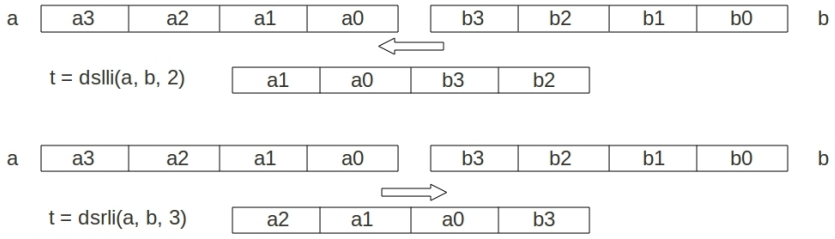

Figure 3.9: The Computing Logic of Fill Movement Ternary Operations

Operation	Meaning	Description
dslli	shift fields of the concatenation of a and b left logical by the number of fields specified in c	$t_i = a_{i-c}$ if $i - c \geq 0$ else $b_{fieldNum-c+i}$, where $fieldNum = NumberOfField_{fw}(a)$
dsrli	shift fields of the concatenation of a and b right logical by the number of fields specified in c	$t_i = b_{i+c}$ if $i + c < fieldNum$ else $a_{i+c-fieldNum}$, where $fieldNum = NumberOfField_{fw}(b)$

Table 3.14: The Field Movement Ternary Operations

Operation	Meaning	Description
fill	fill each field of t with v_1	$t_0 = v_1$; $t_1 = v_1$; ...
*fill*2	fill alternating fields of t with v_1 and v_2	$t_0 = v_1$; $t_1 = v_2$; ...
*fill*4	fill each set of 4 fields of t with v_1 through v_4	$t_0 = v_1$; $t_1 = v_2$; $t_2 = v_3$; $t_3 = v_4$; ...
*fill*8	fill each set of 8 fields of t with v_1 through v_8	$t_0 = v_1$; $t_1 = v_2$; $t_2 = v_3$; $t_3 = v_4$; $t_4 = v_5$; $t_5 = v_6$; $t_6 = v_7$; $t_7 = v_8$; ...
*fill*16	fill each set of 16 fields of t with v_1 through v_{16}	$t_0 = v_1$; $t_1 = v_2$; $t_2 = v_3$; $t_3 = v_4$; $t_4 = v_5$; $t_5 = v_6$; $t_6 = v_7$; $t_7 = v_8$; $t_8 = v_9$; $t_9 = v_{10}$; $t_{10} = v_{11}$; $t_{11} = v_{12}$; $t_{12} = v_{13}$; $t_{13} = v_{14}$; $t_{14} = v_{15}$; $t_{15} = v_{16}$; ...

Table 3.15: The Field Movement Pattern Fill Operations

Operation	Meaning	Description
any	check if there is any non-zero bit in a	$t = true$ if a contains at least one non-zero bit, otherwise, $t = false$
all	check if there is no zero bit in a	$t = true$ if a contains no zero bit, otherwise, $t = false$

Table 3.16: The Bitblock Unary Operations

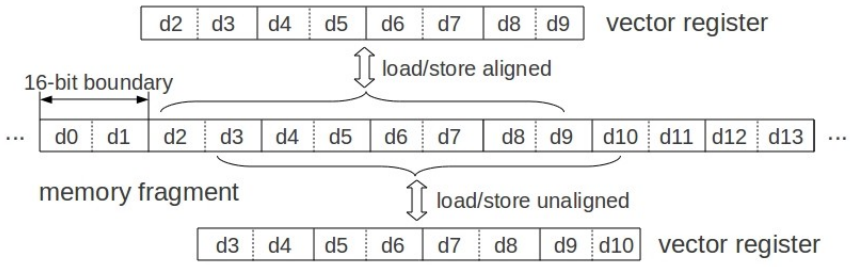

Figure 3.10: Examples of Bitblock Load/Store Operations

3.4.6 Bitblock Operations

The bitblock operations treat the operand registers as unpartitioned registers and apply operations on the entire body of the registers. Depending on the features of operations, they can be divided into two sub-categories, bitblock unary operations and bitblock memory operations.

The Bitblock Unary Operations

All the unary operations defined in the model are boolean functions and are in the form of $t = bitblock\_op(a)$, where a is the only input register and t is a boolean value. In Table 3.16, it shows these bitblock unary operations.

The Bitblock Memory Operations

The memory operations defined here include four memory load or store operations, which are load aligned, load unaligned, store aligned and store unaligned. It is suggested that using aligned load or store should give the better performance gain in the higher-level applications. Table 3.17 describes all these memory operations.

Operation	Meaning	Description
load_aligned	load and return a bitblock value from an aligned location p, where p is a bitblock type pointer	see Figure 3.10
load_unaligned	load and return a bitblock value from an unaligned location p, where p is a bitblock type pointer	see Figure 3.10
store_aligned	store a bitblock value to an aligned location p, where p is a bitblock type pointer	see Figure 3.10
store_unaligned	store a bitblock value to an unaligned location p, where p is a bitblock type pointer	see Figure 3.10

Table 3.17: The Bitblock Memory Operations

3.5 Chapter Summary

In this chapter, we have described many important aspects of our portable model including the design goal, the inductive doubling principle as the foundation for implementing the model and also the core of our model, a set of more than 60 well-defined operations for supporting a rich set of SIMD functionalities. Compared to the programming mode called IDISA in Dr. Cameron's work [19], our portable model refactors and extends the IDISA with many enhancements including more operations defined, an generator for auto-generating library codes and so forth. To distinguish from IDISA, we use IDISA+ as the name for the model.

Chapter 4

Model Implementation

4.1 Programming Interfaces

The libraries produced by IDISA+ offer developers a set of programming interfaces for using the operations defined in IDISA+ without touching the native SIMD instructions. Since the libraries are provided in C++, we will use these C++ programming interfaces for illustration.

4.1.1 Class Declaration

As there are six different categories for IDISA+ operations, it is intuitive to use the C++ class to present each category except the logic one. We map the vertical operations to class *simd*, the horizontal operations to class *hsimd*, the expansion operations to class *esimd*, the field movement operations to class *mvmd* and the bitblock operations to class *bitblock*. For logic operations, we do not map them to any class, instead, we make them as static functions in the libraries with prefix of *simd_* in their function names. And for the vertical, horizontal, expansion and field movement operations, each of them should have the field width as an argument, thus, we adopt the C++ template syntax and make the field width as the template arguments of the classes for these operations. Table 4.1 shows the general class declaration for each category of operations.

However, some architectures may have multiple SIMD instruction sets available, such as Intel Sandy Bridge processors which support AVX as well as SSE series. For such architectures, we want the users to be able to switch between different instruction sets with

Operation Category	Class Name	Class Declaration

vertical operations	simd	*template* < *uint32_t fw* >
		class simd
		{
		...
		};

horizontal operations	hsimd	*template* < *uint32_t fw* >
		class hsimd
		{
		...
		};

expansion operations	esimd	*template* < *uint32_t fw* >
		class esimd
		{
		...
		};

field movement operations	mvmd	*template* < *uint32_t fw* >
		class mvmd
		{
		...
		};

bitblock operations	bitblock	*class bitblock*
		{
		...
		};

Table 4.1: The General C++ Class Declaration for IDISA+ Operations

Class Name	Class Declaration for 128-bit Instruction Sets	Class Declaration for 256-bit Instruction Sets

simd	*template* < *uint*32_*t* *fw* >
	*class simd*128
	{
	...
	};
		template < *uint*32_*t* *fw* >
		*class simd*256
		{
		...
		};

hsimd	*template* < *uint*32_*t* *fw* >
	*class hsimd*128
	{
	...
	};
		template < *uint*32_*t* *fw* >
		*class hsimd*256
		{
		...
		};

esimd	*template* < *uint*32_*t* *fw* >
	*class esimd*128
	{
	...
	};
		template < *uint*32_*t* *fw* >
		*class esimd*256
		{
		...
		};

mvmd	*template* < *uint*32_*t* *fw* >
	*class mvmd*128
	{
	...
	};
		template < *uint*32_*t* *fw* >
		*class mvmd*256
		{
		...
		};

bitblock	*class bitblock*128
	{
	...
	};
		*class bitblock*256
		{
		...
		};

Table 4.2: The Specific C++ Class Declaration for IDISA+ Operations Depending on Register Size

our libraries. So we add two specific sets of class declaration for 128-bit instruction sets and 256-bit instruction sets in addition to the general one, in which, one set is *simd*128, *hsimd*128, *esimd*128, *mvmd*128 and *bitblock*128, and the other set is *simd*256, *hsimd*256, *esimd*256, *mvmd*256 and *bitblock*256. Table 4.2 shows the two sets of class declaration.

4.1.2 Data Type

To enhance the portability of our libraries, we use exact width integer types from C99 standards [44] for defining field widths, integer arguments, integer return types and immediate values. Except when the type of field widths is *uint*32_*t*, all other integers involved in the

Instruction Set	Built-in SIMD Type	Libraries' SIMD Type	Type for Field Width	Type for Other Integers
SSE2 through SSE4.2	__m128i	bitblock128_t	uint32_t	uint64_t
NEON	uint64x2_t	bitblock128_t	uint32_t	uint64_t
AVX	__m256	bitblock256_t	uint32_t	uint64_t

Table 4.3: Data Types Used in the Libraries

libraries are defined as 64-bit unsigned integers, i.e., *uint*64_*t*. For example, the *signmask* operation in *hsimd* class returns an unsigned integer with *uint*64_*t* type and the immediate value for *constant* operation of *simd* class is also *uint*64_*t* and so on.

Each SIMD instruction set usually has two data types for representing a SIMD register, the integer one and the floating point one. Some might even have another type for the register with double precision floating point values. In order to make our libraries simple and clean, we only use one type to represent SIMD data or registers for a certain instruction set, which are *bitblock*128_*t* for 128-bit SIMD instruction sets and *bitblock*256_*t* for 256-bit SIMD instruction sets. Table 4.3 gives a summary on the data types.

4.1.3 Function Declaration

In the generated libraries of IDISA+, each function implements an operation defined in Chapter 3 on fields of a certain size. For the operations which have immediate values as arguments, the corresponding functions are defined as template member functions in their classes with the immediate values being the template arguments. For example, the *srli* operation in *simd*128 class would have the following declaration.

```
template <uint32_t fw>
class simd128
{
  ...
  template <uint64_t sh>
  static IDISA_ALWAYS_INLINE bitblock128_t srli(bitblock128_t arg1);
  ...
};
```

The *sh* is the immediate value associated with *srli* operation and is defined as the template argument of *srli* in *simd*128 class. This allow compilers to explicitly know that *sh* is a compile-time constant and should be given or calculated before the run-time. The $IDISA\_ALWAYS\_INLINE$ is a micro defined by us which is to tell the compiler that the function decorated with it must be inlined. Both tricks explained here should help improve the performance to some extend.

4.1.4 Library Layout and Usage

In Figure 4.1, it shows an overview on the structure of libraries generated by IDISA+. All these ten files in the picture are produced by the IDISA+ generator automatically, in which, *idisa.hpp* is the highest level library that gives developers access to both 128-bit and 256-bit SIMD instruction sets. Similarly, *idisa*128.*hpp* defines the 128-bit instruction sets, such as Intel SSE2 through SSE4.2 and ARM NEON, while *idisa*256.*hpp* supports the 256-bit instruction sets, now only Intel AVX on current commodity processors.

In general, users only have to include the *idisa.hpp* file in order to make use of the IDISA+ operations. The default setting of *idisa.hpp* is to provide operations on SSE2 instruction set with $blocksize = 128$. Users can define the *blocksize* to 256 if they want to switch to operations on AVX instruction set. With *idisa.hpp*, the way of calling IDISA+ operations is to use the general C++ class member functions, such as *simd*<32> :: *add*, *mvmd*<16> :: $fill$ or *bitblock* :: *any*.

If users are working on AVX instruction set but want to use the operations on SSE2, they can include *idisa*128.*hpp* and define a macro USE_SSE2 to get access to the SSE2 operations. In this case, the way of calling SSE2 operations is to use the specific C++ class declarations, such as *simd*128<32> :: *sub*, *mvmd*128<16> :: $fill2$ or *bitblock*128 :: *all*. The separated libraries for 128-bit and 256-bit instruction sets enable users to freely use these two versions of IDISA+ operations in the same program.

4.2 Generator of IDISA+

For implementing a portable SIMD programming model, there are usually two ways to go. One is doing hand-tuned implementations for each platform separately, the other is writing a compiler or translator to translate the instruction written code into real programming language code or assembly code. No matter which way people choose to create a portable

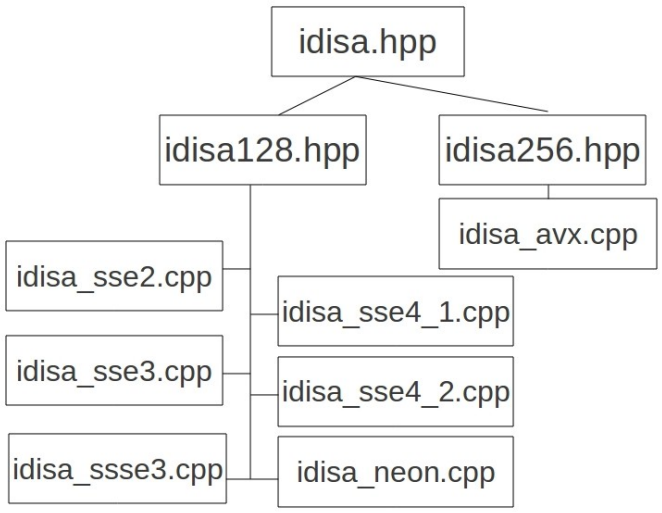

Figure 4.1: The Layout of Libraries Generated by IDISA+

SIMD programming model, the implementations generated by the model should be as good as possible in terms of speed, system resource consumption and so forth. Furthermore, a fairly nice model would automatically generate implementations to avoid as many human efforts involved in writing or developing as possible.

The following content in this section presents our approach of having a library generator to create a portable model as well as the most important modules of generator in details.

4.2.1 System Architecture

Figure 4.2 shows the overall structure of the generator. The working process of the generator is given as follows.

- At starting stage, the generator accepts the instruction set information specified by the user, and loads all the corresponding built-in intrinsics of that instruction set and then pushes this data to the translator module.

- When receiving the intrinsics, the translator starts to load all the defined operations and parse each available strategy to get the temporary implementations for operations. Then it sends these temporary implementations of operations to the analyzer module.

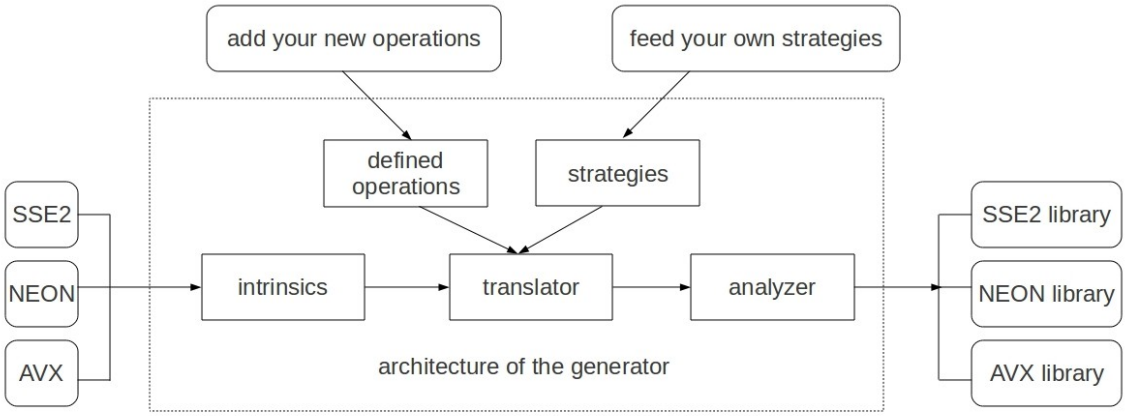

Figure 4.2: The Architecture of IDISA+ Generator

- After having the temporary implementations of operations, analyzer scans and analyzes each of them to find the best one for every defined operation according to some criteria. At last, it produces the library containing the final implementations of operations for the input instruction set.

Outside the main body of the generator, users are allowed to create new operations as a defined operation and also feed their own strategies so that the generator could produce implementation for a new operation or improve the existing implementations as well.

4.2.2 Description of Modules

Intrinsics

In general, an intrinsic is more like a built-in function in a certain programming language for which the developers do not need to write the implementation. The compilers of this particular language are able to recognize the intrinsic and implement it in an inline manner as one or more instructions.

Almost of all today's SIMD instruction sets provide intrinsics as the wrapper of the supported SIMD instructions in C/C++ languages for higher level programming. For example, Table 4.4 shows the SSE2, NEON and AVX intrinsics for integer addition on 32-bit fields.

The intrinsics module defines dictionaries for mapping the operations to the corresponding intrinsics in different instruction sets. Besides, the dictionaries also include some intrinsics that are not matched to any operation. Those are the intrinsics used for emulating

Instruction Set	Operation	Intrinsic
SSE2	$simd128\text{<}32\text{>} :: add$	$\_m128i\ \_mm\_add\_epi32(\_m128i\ a,\ \_m128i\ b)$
NEON	$simd128\text{<}32\text{>} :: add$	$uint32x4\_t\ vaddq\_u32(uint32x4\_t\ a,\ uint32x4\_t\ b)$
AVX	$simd256\text{<}32\text{>} :: add$	$\_m256i\ \_mm256\_add\_epi32(\_m256i\ a,\ \_m256i\ b)$

Table 4.4: An Example on Intrinsics

the operations which are not natively supported. With the intrinsics module, the generator would be able to tell which operation has an intrinsic so that it knows to directly use the intrinsic as the real implementation for that operation. For those operations which do not have intrinsics, the generator can switch the task to translator and let the translator figure out implementations for them based on the available strategies.

IDISA+ Operations

The operations defined in Chapter 3 are imported into the generator as IDISA+ operations. Each IDISA+ operation defines three fields *signature*, *args_type* and *return_type*, which are helpful for facilitating the generator's work. Table 4.5 shows some IDISA+ operations.

We have included four reserved types to help define the IDISA+ operations, they are SIMD_type, $range(0, x)$, $unsigned\_int(x)$, int. The meanings of those reserved types are shown in Table 4.6. As stated in earlier sections, integers involved in the generated libraries are defined as either $uint32\_t$ or $uint64\_t$ integers for simplicity on data type. However, the IDISA+ testing framework has to know the exact data type or range for each integer argument so that it could create test data in the right range for testing operations of the libraries. That is the main reason we make $range(0, x)$ and $unsigned\_int(x)$ as two of the reserved types.

Strategies

As discussed before, not every IDISA+ operation is supported naturally for a specific instruction set. So far, there is little research conducted on effectively making and optimizing the implementations for those non-built-in operations. A common way used in many SIMD programming models is to have certain fixed rules to connect non-built-in operations with built-in ones. But the rules are usually implemented in an ad-hoc way and are independent among different SIMD programming models. And, optimizations based on the rules are still

Operation	IDISA+ Operation
or	*signature*: SIMD_type *simd_or*(SIMD_type $arg1$, SIMD_type $arg2$) *args_type*: {$arg1$: SIMD_type, $arg2$: SIMD_type} *return_type*: SIMD_type
simd_add	*signature*: SIMD_type *simd*<fw> :: *add*(SIMD_type $arg1$, SIMD_type $arg2$) *args_type*: {$arg1$: SIMD_type, $arg2$: SIMD_type} *return_type*: SIMD_type
simd_srli	*signature*: SIMD_type *simd*<fw> :: *srli*<sh>(SIMD_type $arg1$) *args_type*: {sh : range(0, fw), $arg1$: SIMD_type} *return_type*: SIMD_type
hsimd_signmask	*signature*: *int hsimd*<fw> :: *signmask*(SIMD_type $arg1$) *args_type*: {$arg1$: SIMD_type} *return_type*: *unsigned_int*(64)
mvmd_fill	*signature*: SIMD_type *mvmd*<fw> :: *fill*(*int* $val1$) *args_type*: {$val1$: *unsigned_int*(fw)} *return_type*: SIMD_type

Table 4.5: Examples of IDISA+ Operations

Type Name	Description
SIMD_type	It stands for the general type for a SIMD register and will be replaced by either *bitblock*128_*t* or *bitblock*256_*t* in the generated libraries
range$(0, x)$	It stands for an integer which is in range of [0, x-1] and will be replaced by *uint*64_*t* in the generated libraries.
unsigned_int(x)	It stands for a x-bit unsigned integer and will be replaced by *uint*64_*t* in the generated libraries.
int	It stands for a general-purpose integer type and is not identical to the *int* type in any programming language. It is only used in signatures of idisa operations to imply the data types are integer. It will be replaced by *uint*64_*t* in the generated libraries.

Table 4.6: The Reserved Types for Defining IDISA+ Operations

insufficient in these models.

In our IDISA+ model, we present a strategy based approach which groups together a set of strategies of connecting non-built-in and built-in operations and is able to analyze these rules and find the best rules for implementing the non-built-in operations. This approach is so flexible that it allows users to create their own strategies for improving performance or supporting new operations.

A strategy essentially provides a general algorithm for emulating one or more non-built-in operations based on built-in ones. Shown below is a strategy that is applicable for emulating addition or subtraction operations on n-bit fields based on addition or subtraction operations on $2n$-bit fields.

```
strategy_1 =
{
  "body":r'''
hiMask = simd_himask(2*fw)
return simd_ifh(1, hiMask, simd_op(2*fw, arg1, simd_and(hiMask, arg2)),
                simd_op(2*fw, arg1, arg2))
''',
  "ops":["simd_add", "simd_sub"],
  "fws":[-1],
  "platforms":[configure.ALL],
},
```

In the IDISA+ model, all strategies are written in Python language syntax as a dictionary of four elements. The first element is the body of the strategy describing its algorithms, the second element stores a list of operations that are suitable to use this strategy, the third element is a list of field widths that are applicable for the operations to operate on and the last element contains the platforms for which the strategy applies. For better understanding of the concepts of a strategy, some important features of strategies are listed below.

- In the strategy body, functions with *simd*/*hsimd*/*esimd*/*mvmd*/*bitblock* as prefix are the IDISA+ operations. For example, *simd_ifh* is the ifh operation in the *simd* class while *hsimd_packh* is the *packh* operation in the *hsimd* class. When writing a function in the strategy body, the first argument is sometimes reserved for indicating

the field width if the operation is in *simd*, *hsimd*, *esimd* or *mvmd* class, such as $simd\_add(2 * fw, arg1, arg2)$, and the second argument is reserved for the immediate value if the operation has a template argument, e.g., $simd\_slli(fw, 1, arg1)$.

- We have also included three abstract functions called *simd_op*, *simd_uop* and *simd_sop*, which are used to simplify the strategy writing. In *strategy_1*, the *simd_op* will be replaced by either *simd_add* or *simd_sub* in the strategy translation phrase. Similarly, the *simd_uop* or *simd_sop* would be replaced by the unsigned or signed version of an operation, such as *simd_ugt* when $op = gt$ or *simd_gt* when $op = ugt$.

- As some strategies might only work for several field widths, so there is a requirement on field width for each strategy. The fws element in a strategy is defined as an array which could be a list of one or more values. $[-1]$ in *strategy_1* means the strategy could use every possible field width, and $[2, 4, 8]$ infers that the strategy only works on 2-bit, 4-bit or 8-bit fields.

- Strategy bodies may contain some intrinsics directly. Such intrinsics are those intrinsics which can not be mapped to any IDISA+ operation directly but could be used for simulating some IDISA+ operations. For example, _mm_slli_si128 is an intrinsic in Intel SSE2 for shifting the entire 128-bit field left by certain amount of bytes while shifting in zeros, and it can be used to implement *simd_slli* on 128-bit fields. Hence, _mm_slli_si128 can be written directly into the strategy body, and the generator will automatically check it with the dictionary in intrinsics module to make sure it is an intrinsic and emit it as it is in the generated libraries.

In practice, there are many strategies existing for implementing a certain operation. Below shows another strategy for addition on n-bit fields based on addition on $n/2$-bit fields.

```
strategy_2 =
{
  "body":r'''
partial = simd_add(fw/2, arg1, arg2)
carryMask = simd_or(simd_and(arg1, arg2), simd_andc(simd_xor(arg1, arg2),
                    partial))
```

Operation	Algorithm
$simd<1> :: add(arg1, arg2)$	$simd\_xor(arg1, arg2)$
$simd<1> :: sub(arg1, arg2)$	$simd\_xor(arg1, arg2)$
$simd<1> :: mult(arg1, arg2)$	$simd\_and(arg1, arg2)$
$simd<1> :: eq(arg1, arg2)$	$simd\_not(simd\_xor(arg1, arg2))$
$simd<1> :: gt(arg1, arg2)$	$simd\_andc(arg2, arg1)$
$simd<1> :: ugt(arg1, arg2)$	$simd\_andc(arg1, arg2)$
$simd<1> :: lt(arg1, arg2)$	$simd\_andc(arg1, arg2)$
$simd<1> :: ult(arg1, arg2)$	$simd\_andc(arg2, arg1)$
$simd<1> :: max(arg1, arg2)$	$simd\_and(arg1, arg2)$
$simd<1> :: umax(arg1, arg2)$	$simd\_or(arg1, arg2)$
$simd<1> :: min(arg1, arg2)$	$simd\_or(arg1, arg2)$
$simd<1> :: umin(arg1, arg2)$	$simd\_and(arg1, arg2)$
$simd<1> :: ctz(arg1)$	$simd\_not(arg1)$

Table 4.7: The Algorithms for Vertical Binary Operations on 1-bit Fields

```
carry = simd_slli(fw, fw/2, simd_srli(fw/2, fw/2-1, carryMask))
return simd_add(fw/2, partial, carry)
''',
  "ops":["simd_add"],
  "fws":range(2, currentRegSize+1),
  "platforms":[configure.ALL],
}
```

For some operations on small fields, we could create very concise but efficient strategies only based on logic and a few shifting operations. A quick example would be using *simd_xor* operation to emulate *add* and *sub* operations on 1-bit fields. Table 4.7 summarizes these kinds of computing algorithms for some vertical operations on 1-bit fields. Similarly, we could use shifting operations in addition to the logic operations to emulate those operations on 2-bit fields. The reason shifting operations are involved is because the low bit of each 2-bit field in input registers contributes to the high bit of every 2-bit field in the result registers.

Translator

Since all the strategies are written in Python syntax, there must be a module which could parse the strategies and produce the corresponding C++ codes. In our model, we developed

Strategy Name	Function Calls
strategy_1	[*simd_himask_16*, *simd_ifh_1*, *simd_add_16*, *simd_and*, *simd_add_16*]
strategy_2	[*simd_add_4*, *simd_or*, *simd_and*, *simd_andc*, *simd_xor*, *simd_slli_8*, *simd_srli_4*, *simd_add_4*]

Table 4.8: The Function Calls in Strategies

Strategy Name	C++ Implementations

strategy_1	*template*<> IDISA_ALWAYS_INLINE *bitblock*128_*t*
	simd<8> :: *add*(*bitblock*128_*t arg*1, *bitblock*128_*t arg*2)
	{
	*bitblock*128_*t hiMask* = *simd*<16> :: *himask*();
	return simd<1> :: *ifh*(*hiMask*, *simd*<16> :: *add*(*arg*1,
	simd_and(*hiMask*, *arg*2)), *simd*<16> :: *add*(*arg*1, *arg*2));
	}

strategy_2	*template*<> IDISA_ALWAYS_INLINE *bitblock*128_*t*
	simd<8> :: *add*(*bitblock*128_*t arg*1, *bitblock*128_*t arg*2)
	{
	*bitblock*128_*t partial* = *simd*<4> :: *add*(*arg*1, *arg*2);
	*bitblock*128_*t carryMask* = *simd_or*(*simd_and*(*arg*1, *arg*2),
	simd_andc(*simd_xor*(*arg*1, *arg*2), *partial*));
	*bitblock*128_*t carry* = *simd*<8> :: *slli*<4>(
	simd<4> :: *srli*<3>(*carryMask*));
	return simd<4> :: *add*(*partial*, *carry*);
	}

Table 4.9: The C++ Implementations of Strategies

such module called translator that takes the intrinsics, the IDISA+ operations and all available strategies together to create a table containing candidate lists for every operation on some field widths. The candidate list for an operation on a certain field width not only has the real C++ implementation for it but also extracts a list of all the function calls in that C++ implementation for further analysis.

For example, when the translator is working on *simd*<8> :: *add*, i.e., *op* = *simd_add* and $fw = 8$, and the optional strategies are *strategy_1* and *strategy_2*. It would generate the following candidate list for *simd*<8> :: *add*, in which the function calls are shown in Table 4.8 and the associated C++ implementations are shown in Table 4.9.

Analyzer

Given the candidate lists produced by the translator, we must have a method to effectively select the best implementation for an operation with a specific field width. In the analyzer module, it uses the following mechanism to evaluate each temporary implementation.

Least Instruction Count

The implementation which has the least instruction count is the best one. Meanwhile, the least instruction count mechanism makes an assumption that every intrinsic provided by an instruction set has cost of only one instruction. However, it is notable that there are some intrinsics which are composite intrinsics with each of them containing more than one instructions. But those intrinsics are few in a given instruction set, so the assumption still holds in most cases such that the compiler produces single instruction for an intrinsic.

Based on the least instruction count mechanism, an iterative algorithm has been developed to automatically find the best implementation for every IDISA+ operation. The pseudo-code summarized in Algorithm 1 illustrates the idea of this iterative algorithm.

- Initially, every built-in operation has cost of 1 and its C++ implementation is set directly to be the corresponding intrinsic. Any other non-built-in operation has cost of a very large value and its C++ implementation is set to be null at this moment.
- At each iteration, the algorithm scans the candidate lists and gets a list of function calls and temporary implementations for each operation op_fw. Then it exams each function calls in the list to calculate an estimated cost $tmpCost$, and compares the $tmpCost$ with op_fw's current best cost $opCost[op\_fw]$. If $tmpCost < opCost[op\_fw]$, the algorithm will set $opCost[op\_fw]$ to be $tmpCost$ and the C++ implementation of op_fw is updated to be the temporary implementation whose cost is $tmpCost$.
- If there are no updates occurring within an iteration, the algorithm terminates and returns the implementations for all operations. Otherwise, it continues to the next iteration.

Time Complexity of the Iterative Algorithm

In each iteration, the algorithm mainly does two things. First, it scans every strategy to try

Algorithm 1 IterativeAlgorithm(Intrinsics, CandidateLists)

```
1: opCost ← {}, cppImp ← {}
2: for op_fw in CandidateLists do
3:    opCost[op_fw] ← a very big positive number
4: end for
5: for op_fw in Intrinsics do
6:    #Each intrinsic only has one instruction
7:    opCost[op_fw] ← 1
8:    cppImp[op_fw] ← Intrinsics[op_fw]
9: end for
10: while True do
11:    changed = False
12:    for op_fw in CandidateLists do
13:       for (funcCalls, tmpImp) in CandidateLists[op_fw] do
14:          tmpCost ← CalculateCost(funcCalls)
15:          if tmpCost < opCost[op_fw] then
16:             opCost[op_fw] ← tmpCost
17:             cppImp[op_fw] ← tmpImp
18:             changed ← True
19:          end if
20:       end for
21:    end for
22:    if changed == False then
23:       break
24:    end if
25: end while
26: return cppImp
```

to update the costs and implementations for operations. Secondly, there is at least one operation gets updated in each iteration, otherwise, the algorithm would terminate. Suppose we have N operations and M strategies, in the worst case, the algorithm is going to run N iterations with M strategies being scanned in each, so the total time complexity is $O(N*M)$. In practice, the algorithm works pretty fast and normally produces the results in one second.

Set Different Cost for Intrinsics

As stated before, we assume each intrinsic has a cost of one instruction and therefore such an intrinsic takes one CPU cycle in terms of execution time. However, in reality, one instruction usually takes less than a CPU cycle to be finished due to the instruction-level parallelism in the modern processors with superscalar architectures. Some instructions like multiplication might take more than one CPU cycle depending on the hardware implementation for that instruction.

Thus, people normally use the reciprocal throughput for better estimating the cost of an instruction. Reciprocal throughput is measured in cycles per instruction, which reflects the time of executing an instruction. In the instruction tables provided by Agner Fog [25], reciprocal throughput of most instructions for Intel, AMD and VIA CPUs are obtained based on the author's own experiments on these machines. We could evaluate strategies more accurately by setting cost for intrinsics in IDISA+ model according to the reciprocal throughput values from Fog's paper.

4.3 Tester of IDISA+

In Figure 4.3, it shows the overall architecture of IDISA+ tester. There are two major components, one is the correctness testing module for testing correctness for each individual function in the generated libraries, and the other one is the performance analysis module for getting the actual assembly instruction count of each function. This section reviews the general idea of these two modules.

4.3.1 Correctness Testing

Correctness testing is mainly composed of the following three phases.

- After the module receives the generated library and the defined IDISA+ operations

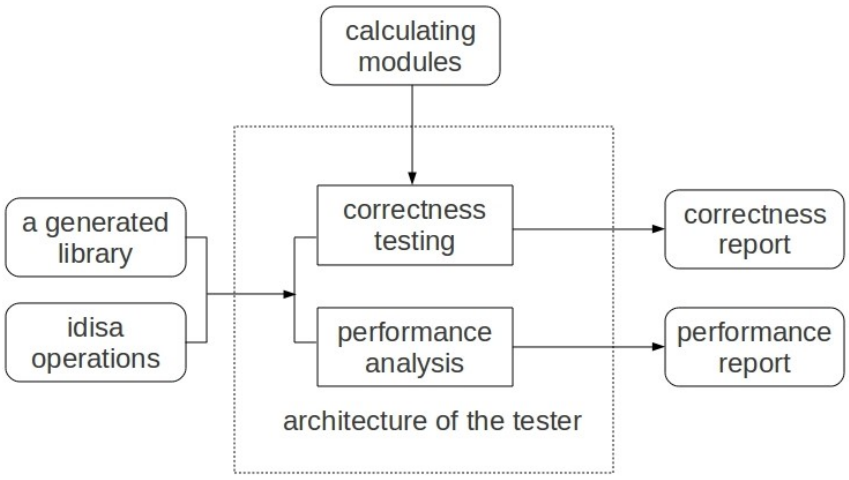

Figure 4.3: The Architecture of IDISA+ Tester

from the generator, it then parses the library as well as the IDISA+ operations in order to know what operations have been implemented and also to determine the range of input data for them.

- Based on the information collected in the first phase, it starts to generate a set of test data for each function, normally one hundred cases per function. Meanwhile, it produces a C++ file containing all the testing procedures including routines which call the functions in the generated library with the auto-generated input data and write the results on the disk. Then the module compiles and runs this C++ file with the generated library and another library called *utility.h* containing some I/O routines and other auxiliary programs.

- The module now loads the simulation programs for IDISA+ operations from the calculating modules. Those programs can take the input data in the second phase and produce results as standard answers for correctness checking. In the end, the module compares the answers produced by the simulation programs with the results produced in the second phase to check if there is any function could not pass the correctness testing.

One advantage is that the calculating modules are designed to be independent from the correctness module so that users are able to do the correctness testing on their new

operations by simply adding the corresponding simulation programs. Another advantage is, this module provides unit testing in an automatic manner to avoid potential bugs and also helps verify newly created strategies.

4.3.2 Performance Analysis

The working process of performance analysis is quite similar to that of correctness testing. At first, it parses the generated library and the IDISA+ operations to know which operations have been implemented in the library. According to the information about the operations, it will produce a C++ file for measuring the performance in terms of instruction count for each function in the generated library. Such a C++ file only has a main function in which a specific function is called once. Then for each C++ file, the performance analysis module compiles it into assembly codes and uses a parser to get the number of assembly instructions for the corresponding function.

4.4 Chapter Summary

In this chapter, we have presented the underlying design and algorithms of the IDISA+ model in details. The model has been implemented as a toolkit which includes two components, an improvable generator for producing the portable libraries and a comprehensive test framework for both correctness and performance analysis on the libraries. The modular design of the model has shown many advantages in terms of the capabilities to add new operations and architectures, the ease of maintenance, and an automatic way of getting portable SIMD libraries. Based on the least instruction count mechanism, it is believed that the libraries generated by the model would achieve promising results on performance. In the next chapter, we will conduct some evaluation work to explore the performance of the libraries.

Chapter 5

Evaluation

In this chapter, we focus on presenting the evaluation work of our IDISA+ model to demonstrate that the model can not only successfully generate C++ implementations for the defined operations but can also optimize the implementations by selecting those with the least number of instructions.

5.1 Overview

Before starting to evaluate the IDISA+ model's performance, we first show some important numbers about the model on applicable strategies, used intrinsics, IDISA+ operations and IDISA+ functions. Table 5.1 summarizes these statistics.

Applicable Strategies

We have created 242 strategies in total to support simulating the non-built-in operations with some power-of-2 field widths. A strategy might be a universal strategy which is applicable for certain operations in all kinds of instruction sets. *strategy_1* and *strategy_2* in Chapter 4 are two such strategies. There are also some other strategies which are only suitable for operations in a specific instruction set. For example, a strategy using intrinsics of an instruction set is only applicable to operations in that instruction set. In the current implementation of IDISA+ model, there are 187 strategies applicable to SSE2, 171 strategies applicable to NEON and 168 strategies applicable to AVX.

In Appendix B, there are tables which describe the number of applicable strategies for each IDISA+ function in SSE2, NEON and AVX.

InstructionSet / *Category*	SSE2	NEON	AVX
Applicable Strategies	187	171	168
Intrinsics	71	88	83
IDISA+ Operations	61	58	58
IDISA+ Functions	376	366	411

Table 5.1: Some Statistics on SSE2, NEON and AVX in IDISA+

Used Intrinsics

An intrinsic in an instruction set only corresponds to an operation with a specific field width. As we have more than 60 operations defined in IDISA+ model and each of them normally has 7 or 8 different instances or functions depending on the field width, it is common that not many IDISA+ functions have intrinsics supported in a given instruction set. Hence, there are only 71 intrinsics used for SSE2, 88 intrinsics used for NEON and 83 intrinsics used for AVX in our model to directly implement some IDISA+ functions.

Available IDISA+ Operations and Functions

With the current available strategies and intrinsics, we are now able to generate 376 IDISA+ functions for SSE2, 366 IDISA+ functions for NEON and 411 IDISA+ functions for AVX. Although AVX has less number of intrinsics and applicable strategies, AVX have more IDISA+ functions compared to SSE2 or NEON due to its larger SIMD register size.

In Appendix A, there are tables showing all the available IDISA+ functions in SSE2, NEON and AVX.

5.2 Evaluation on IDISA+ Implementations

Given the implementations produced by the IDISA+ generator, we want to know whether the cost estimation in the generator is reasonable and whether these implementations are the best possible implementations based on the available strategies and intrinsics. Hence, we designed two experiments to evaluate these two aspects, in which one experiment is to study the difference between the estimated instruction count and the real instruction count for each IDISA+ function in order to gain some understanding about the effectiveness of the generator's cost evaluation mechanism, the other experiment is to compare the best

implementation with the second best implementation for each IDISA+ function based on the generator's cost estimation mechanism to see how the generator does on distinguishing the best strategy from other strategies.

5.2.1 Estimated Instruction Count vs Real Instruction Count

We have chosen five operations as examples to help present the comparison between estimated instruction count and real instruction count. In Table 5.3, it lists the related instruction count for these operations. The numbers are divided into three groups, the group for the estimated instruction count, the group for real non-movement instruction count and the group for real instruction count. The meanings of these groups are given below.

- The estimated instruction count is predicted by the IDISA+ generator according to its least instruction count mechanism.
- The real non-movement instruction count is the number of non-movement instructions for a IDISA+ function and is obtained through G++ 4.4.5 compiler under highest (-O3) optimization level. The reason we report this kind of instruction count is that the generator can not predict the movement instructions but the normal SIMD instructions. It is more accurate to compare the difference between the estimated instruction count with the non-movement instruction count.
- The real instruction count is the number of all instructions including the normal SIMD instructions as well as movement instructions for a IDISA+ function and is obtained through G++ 4.4.5 compiler under highest (-O3) optimization level.

From Table 5.3, we can see that the number of estimated instruction count is very close to the real non-movement instruction count. The main reason is that the generator evaluates the instruction count of a IDISA+ function based on the number of intrinsics this function would use, and an intrinsic is often substituted by single instruction through compilation. However, there might be some extra movement instructions carried out by the compiler to implement a function due to the limited number of registers in real computer systems. Unfortunately, the current generator could not predict the movement instructions so that the real instruction count is always greater than the estimated instruction count in practice.

In Table 5.2, we show the C++ implementation and assembly code for *simd*<4>::add. The estimated instruction count is also given as 6 in the C++ implementation, while we

The C++ Implementation of *simd*<4>::add

//The Estimated Instruction Count is 6
template<> IDISA_ALWAYS_INLINE *bitblock*128_*t*
simd<4> :: *add*(*bitblock*128_*t* *arg*1, *bitblock*128_*t* *arg*2)
{
return *simd*<1> :: *ifh*(*simd*<8> :: *himask*(),
simd<8> :: *add*(*arg*1, *simd_and*(*simd*<8> :: *himask*(), *arg*2)),
simd<8> :: *add*(*arg*1, *arg*2));
}

The Assembly Code of *simd*<4>::add
movdqa .LC15(%rip), %xmm0 movdqa %xmm1, %xmm3 paddb %xmm2, %xmm3 pand %xmm0, %xmm2 paddb %xmm2, %xmm1 movdqa %xmm0, %xmm2 pand %xmm1, %xmm0 pandn %xmm3, %xmm2 movdqa %xmm2, %xmm3 por %xmm0, %xmm3 movdqa %xmm3, 16(%rsp)

Table 5.2: The C++ Implementation and Assembly Code for *simd*<4>::add

can easily tell from the assembly code that there are exactly 6 non-movement instructions (2**paddb*, 2**pand*, 1**pandn*, 1**por*) and 11 instructions in total. In Appendix C, we collect the estimated and real instruction count information for each SSE2 individual function.

In addition to the above five operations, we also report a chart in Figure 5.1 which gives an overall picture about estimated and real instruction count for all IDISA+ functions in SSE2. An inflection point of a curve in the chart represents the average instruction count over all IDISA+ operations with a certain field width (e.g., 2 or 4). Thus, a curve in the chart shows the trend of a specific type of instruction count of IDISA+ functions over power-of-2 fields. In Figure 5.1, the curve on the top is the trend for real instruction count, which is always above either the curve for estimated instruction count or the curve for real non-movement instruction count. The black continuous curve in the middle is for estimated instruction count while the curve for real non-movement instruction count is at the bottom, and these two curve are very close to each other with the one for non-movement instruction count being lower.

Operation \ *fw*	1	2	4	8	16	32	64	128
The Estimated Instruction Count								
bitblock_all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2
esimd_mergeh	31	21	11	1	1	1	1	N/A
hsimd_packh	N/A	47	33	19	3	21	7	7
mvmd_splat	16	13	9	5	2	1	5	13
simd_add	1	10	6	1	1	1	1	11
The Real Non-movement Instruction Count								
bitblock_all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2
esimd_mergeh	31	21	11	1	1	1	1	N/A
hsimd_packh	N/A	47	33	19	3	15	7	4
mvmd_splat	11	15	11	7	4	1	5	13
simd_add	1	10	6	1	1	1	1	8
The Real Instruction Count								
bitblock_all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2
esimd_mergeh	44	30	16	2	2	2	2	N/A
hsimd_packh	N/A	70	49	28	4	23	10	8
mvmd_splat	14	22	17	12	8	2	9	21
simd_add	2	17	11	2	2	2	2	12

Table 5.3: The Estimated and Real Instruction Count of Several IDISA+ Operations in SSE2

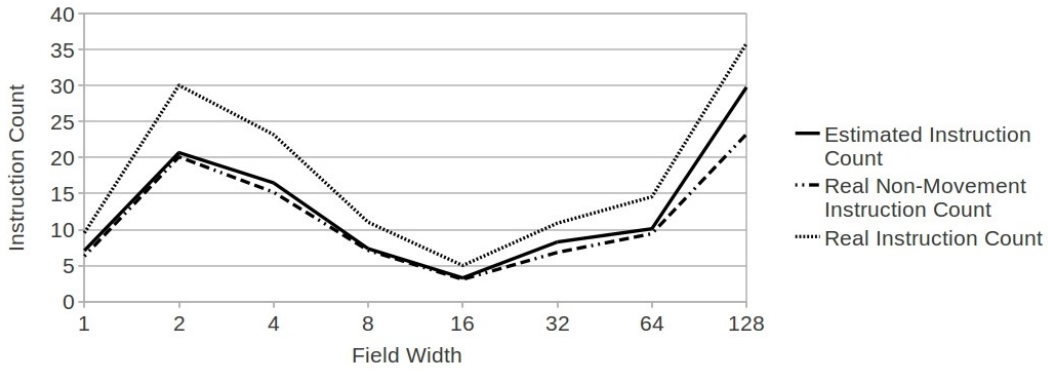

Figure 5.1: Comparison Between Estimated and Real Instruction Count for SSE2

Although the estimated instruction count is quite close to the real non-movement instruction count, the real non-movement instruction count appears to be less than the estimated one overall. We think there are two main reasons. First, there are implementations with ternary conditional statements (i.e., $cond$? $stateA$: $stateB$) involved, and the generator is not able to identify the truth of the condition in a ternary conditional statement and it always sets the cost of this ternary conditional statement to be the higher estimated cost between the two result statements. Obviously, there is an overhead when dealing with the ternary conditional implementations which might make the estimated instruction count greater than the real non-movement instruction count. Secondly, the G++ compiler often does some very strong optimizations as we compile our IDISA+ implementations with the highest optimization level, which could also reduce the final instruction count.

As shown in the above tables and the figure, the generator has a good performance in terms of modeling the number of non-movement instructions for the real assembly code. And also, a function with less non-movement instructions would normally have less movement instructions which ends up with less total instructions. So, we believe that the non-movement instruction count is a strong factor for weighting a function. In the next section, we show how our model does on evaluating two different implementations of a function.

5.2.2 Best Implementation vs Second Best Implementation

A certain function in the IDISA+ libraries would have many different kinds of implementations as we have created hundreds of strategies to help implement certain operations on some fields. It would be interesting if we can compare the estimated best implementation

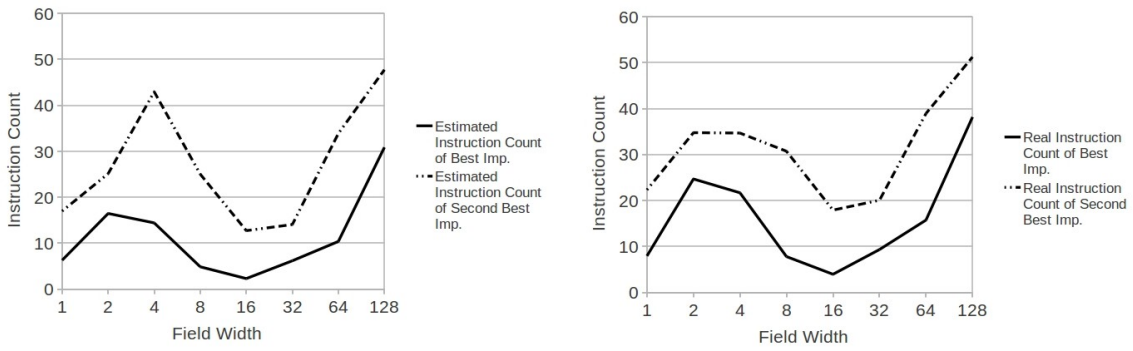

Figure 5.2: Comparison Between Best and Second Best Implementations for SSE2

with the estimated second best implementation for each function to verify whether the least instruction count mechanism actually works.

In Table 5.4, we list the various instruction count including the estimated instruction count on the best and second best implementations, and the real non-movement and total instruction count on the best and second best implementations for three functions, $simd<4> :: max$, $hsimd<16> :: packh$ and $esimd<8> :: signextendh$. The second best implementation of each function tends to have more assembly instructions in reality than the best implementation as suggested in the table, which matches well to the estimation of the generator. Appendix C has listed the related instruction count for all IDISA+ functions in SSE2.

In Table 5.5, we present the C++ code of both best and second best implementation for $simd<4> :: max$. The assembly code generated by G++ compiler for these two implementations are given in Table 5.6, in which we can easily tell that the estimated best implementation is indeed better than the estimated second best implementation with 13 instructions versus 21 instructions.

The left chart In Figure 5.2 shows the average estimated instruction count for the best and second best implementations over the IDISA+ functions. The right chart in the same figure shows the average real instruction count for the best and second best implementations over the IDISA+ functions. As both charts suggest, the best implementation thought by the generator is always better than the second best implementation for any function, which indicates that the least instruction count mechanism in our generator works quite well in terms of selecting the best strategies to get the promising implementations.

	simd<4> :: *max*	*hsimd*<16> :: *packh*	*esimd*<8> :: *signextendh*
Est. IC of the Best Imp.	9	3	4
Est. IC of the Second Best Imp.	13	5	8
Real Non-mov. IC of the Best Imp.	9	3	4
Real Non-mov. IC of the Second Best Imp.	13	5	5
Real IC of the Best Imp.	13	4	6
Real IC of the Second Best Imp.	21	7	7

Table 5.4: The Estimated and Real Instruction Count (IC) of Best and Second Best implementations of Several IDISA+ Functions in SSE2

The Best C++ Implementation of *simd*<4>::max

```
//The Estimated Instruction Count is 9
template<> IDISA_ALWAYS_INLINE bitblock128_t
simd<4> :: max(bitblock128_t arg1, bitblock128_t arg2)
{
    bitblock128_t high_bit = simd<4> :: constant<8>();
    return simd_xor(simd<4> :: umax(simd_xor(arg1, high_bit),
            simd_xor(arg2, high_bit)), high_bit);
}
```

The Second Best C++ Implementation of *simd*<4>::max

```
//The Estimated Instruction Count is 13
template<> IDISA_ALWAYS_INLINE bitblock128_t
simd<4> :: max(bitblock128_t arg1, bitblock128_t arg2)
{
    return simd<1> :: ifh(simd<4> :: gt(arg1, arg2), arg1, arg2);
}
```

Table 5.5: The Best and Second Best C++ Implementation for *simd*<4>::max in SSE2

The Assembly Code of the Best Imp. of *simd*<4>::max	The Assembly Code of the Second Best Imp. of *simd*<4>::max
	movdqa .LC15(%rip), %xmm1
	movdqa %xmm3, %xmm0
	movdqa %xmm2, %xmm4
	movdqa %xmm1, %xmm5
movdqa .LC15(%rip), %xmm0	pslld $4, %xmm0
movdqa .LC16(%rip), %xmm3	pslld $4, %xmm4
pxor %xmm0, %xmm2	pand %xmm1, %xmm0
pxor %xmm0, %xmm1	pand %xmm1, %xmm4
movdqa %xmm3, %xmm4	pcmpgtb %xmm0, %xmm4
pand %xmm2, %xmm3	movdqa %xmm1, %xmm0
pand %xmm1, %xmm4	pand %xmm2, %xmm0
pmaxub %xmm2, %xmm1	pandn %xmm4, %xmm5
pand .LC17(%rip), %xmm1	pcmpgtb %xmm3, %xmm0
pmaxub %xmm4, %xmm3	pand %xmm1, %xmm0
por %xmm3, %xmm1	por %xmm5, %xmm0
pxor %xmm1, %xmm0	movdqa %xmm0, %xmm1
movdqa %xmm0, 16(%rsp)	pand %xmm0, %xmm2
	pandn %xmm3, %xmm1
	movdqa %xmm1, %xmm3
	por %xmm2, %xmm3
	movdqa %xmm3, 16(%rsp)

Table 5.6: The Assembly Code of the Best and Second Best C++ Implementation for *simd*<4>::max in SSE2

	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
Hand-written Libraries	5.858	6.309	7.408	7.74	8.581
IDISA+ Libraries	5.859	6.305	7.406	7.675	8.389

Table 5.7: The Performance of Xmlwf on the Hand-written Libraries and the IDISA+ Libraries (cycle per byte)

5.3 The Generated IDISA+ Libraries for Higher Level Applications

In this section, we want to show how the generated IDISA+ libraries behave for higher level applications. Thus, we choose two applications which are all developed in our lab and both have two versions with one version using the hand-written libraries and the other version using the IDISA+ libraries. The hand-written libraries are highly tuned and optimized as they were hand written initially and have been maintained and updated through a long period of time.

Xmlwf

Xmlwf stands for a kind of applications which are to determine if a XML file is well-formed. In our lab, we developed Xmlwf based on the parallel bit stream technology using SIMD instructions. The performance of Xmlwf on both the hand-written libraries and the IDISA+ libraries is shown in Table 5.7. The IDISA+ libraries perform as fast as the hand-written libraries do with an exception of 0.2 cycle faster on the last test file *soap.xml*. The not significant difference between these two versions is because Xmlwf heavily relies on logic, shifting and packing operations and these operations have already been highly optimized in the old libraries so that the room for further optimization is just little.

Symbol Table

The Symbol Table application here is to use SIMD techniques to support fast validation on names have the correct XML name syntax if they have appeared previously in the document, which avoids the traditional byte-by-byte checking for the names. The Symbol Table program is currently built upon the top of the Xmlwf and it receives the data from Xmlwf and then processes the validation with the help from symbol table look-up.

In Table 5.8, the performance of Symbol Table based on both hand-written Libraries and

	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
Hand-written Libraries	7.141	7.754	9.331	10.219	10.965
IDISA+ Libraries	6.974	7.541	9.225	10.164	10.815

Table 5.8: The Performance of Symbol Table on the Hand-written Libraries and the IDISA+ Libraries (cycle per byte)

IDISA+ libraries are shown. The version using IDISA+ libraries performs about 0.2 cycle faster than the version using hand-written libraries. The performance gain here is mainly due to the full support of SIMD operations as well as optimizations for every single function in IDISA+ libraries. In the hand-written Libraries, some operations are only supported with a few of field widths while some operations are not even supported at all. When using hand-written libraries to implement the Symbol Table application, we have to manually simulate some functionalities in a combination of functions from the libraries. In contrast, many of those functionalities are directly supported and high optimized in the IDISA+ libraries. Hence, a slightly better performance can be achieved by using our auto-generated IDISA+ libraries.

Chapter 6

Conclusion

6.1 Thesis Summary

In this thesis, we reviewed the concept of SIMD within a register and also several commercial SIMD instruction sets on commodity microprocessors. Many compute-intensive applications have adopted those commercial SIMD instruction sets to improve performance. They also have encouraged more and more research work on issues of using these SIMD instructions, especially the portable SIMD programming over platforms.

To enable portable SIMD programming, we presented a model called IDISA+ in the thesis. The goal of this model is to automatically produce C++ libraries for supporting portable SIMD programming on various architectures. We defined more than 60 operations in our model to fully exploit the advantages of SIMD computing. The defined operations are well-organized and carefully-selected, which cover not only the traditional arithmetic operations but also a good number of field packing and re-arranging operations. These operations are grouped into six different categories based on the the shape or structure of field manipulation by the operations.

To implement the IDISA+ model, a toolkit was constructed including a generator for producing C++ libraries and a test framework for testing the libraries. The generator is developed as an improvable module which allows users to add new operations and strategies as long as they follow the definition rules in the generator. We have created an algorithm based on least instruction count mechanism to discover the best strategy from all the available strategies for implementing a certain operation. This algorithm works reasonably fast in practice with time complexity of $O(M * N)$, where M is the number of operations and

N is the number of strategies. The test framework provides correctness and performance testing for each individual function in the generated libraries. Thus, our IDISA+ model is guaranteed to offer bug-free libraries with this test framework.

In sum, there are four major advantages in our model. First, it supports auto-generating the libraries with little human effort while the hand-written libraries usually require considerable effort. Secondly, with the correctness testing module in the tester, we are able to provide bug-free libraries while errors are usually a big concern with hand-written libraries. Thirdly, as shown in the previous chapter, the libraries generated by the model achieve better performance compared to the hand-written libraries. Finally, our model gives users more opportunities to interact with it, such as improve the performance of generated libraries by feeding some better strategies, or get more operations supported by simply adding the operations through the designated modules.

6.2 Future Work

The model introduced in this thesis has few optimizations from compiler technologies, it would be very nice to incorporate more compiler technologies into the model. One of such candidates is the LLVM compiler infrastructure [37, 38, 39]. LLVM is a modern and SSA-based compiler platform with the capability of supporting both static and dynamic compilation for arbitrary programming languages. It has a very useful target-independent code generator which supports SIMD code generation over multiple modern architectures. One possible improvement is to let LLVM understand the strategies defined in IDISA+ model and integrate the analyzer module into LLVM so that it could use the optimization modules in LLVM to optimize the emitted codes with the knowledge from the strategies. In such approach, a reduction in cycles could be expected with the help of LLVM's register and instruction scheduling algorithms. In addition, we might also enable some other optimizations by adding a peephole optimizer to LLVM. There are situations where a sequence of IDISA+ operations could be replaced by single composite SIMD intrinsic available on a particular architecture. Recognizing such instruction patterns and replacing them by the intrinsic would have substantial benefit.

The other future work on the model is to improve the accuracy of cost estimation. One such work is to make the generator be able to identify the truth of the conditions in ternary conditional statements of a strategy. Fortunately, almost all conditions are compile-time

conditions which makes it possible for the generator to evaluate the cost of a strategy containing ternary conditional statements more accurately at the stage where the candidate lists are produced (i.e., the translation stage). If the condition of a ternary conditional statement is not a compile-time condition, we could set the cost of the statement to be the average cost of all branches. Another possible work is to try to assign different cost for intrinsics, for example, logic operations are usually made as intrinsics which have very low cost in terms of reciprocal throughput (e.g. less than 0.5). It might be more reasonable to set the cost of intrinsics of logic operations to 0.5 instead of 1. We could also adopt the numbers from Fog's paper [25] to guide the cost assignment and see what the performance of the generated libraries would be.

Although the next extension of AVX will be capable to support integer operations on the entire 256-bit register (AVX2 is planned to be released in 2013), it is still an interesting work to improve the implementation for current AVX instruction set. Since the current version of AVX does not support integer operations on the entire 256-bit register but only on the register's low 128 bits, we have to extract the content from an AVX register twice (one for the high 128 bits and the other for the low 128 bits) every time before we actually start the processing of the content. What is worse is that we then have to combine these two content into an AVX register after finishing processing them in order to return a result with AVX vector type. In a single function, there might be a case that the content in an AVX register was just combined couple instructions ahead where it has to be extracted again. This gives non-negligible overhead which we want to get rid of. One possible way to overcome this problem is to add a optimization module in the generator which can analyze the implementation of a single function and try to optimize out the combination operation if there is an extraction on the same content after it.

IDISA+ model now only supports SSE series, NEON and AVX instruction sets, so another future work is to add new architecture support in the model, such as Altivec. And also, we might define and add some new operations in the model to support other specific applications in the future.

Appendix A

IDISA+ Functions

Operation \ *fw*	1	2	4	8	16	32	64	128
bitblock :: all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: any	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: load_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: load_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: store_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: store_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
esimd<fw> :: mergeh	√	√	√	√	√	√	√	N/A
esimd<fw> :: mergel	√	√	√	√	√	√	√	N/A
esimd<fw> :: multh	×	×	×	×	×	×	×	N/A
esimd<fw> :: multl	×	×	×	×	×	×	×	N/A
esimd<fw> :: signextendh	√	√	√	√	√	√	√	N/A
esimd<fw> :: signextendl	√	√	√	√	√	√	√	N/A
esimd<fw> :: zeroextendh	√	√	√	√	√	√	√	N/A
esimd<fw> :: zeroextendl	√	√	√	√	√	√	√	N/A
hsimd<fw> :: add_hl	N/A	√	√	√	√	√	√	√
hsimd<fw> :: min_hl	N/A	√	√	√	√	√	√	√
hsimd<fw> :: packh	N/A	√	√	√	√	√	√	√
hsimd<fw> :: packl	N/A	√	√	√	√	√	√	√

hsimd<fw> :: packss	N/A	√	√	√	√	√	√	√
hsimd<fw> :: packus	N/A	√	√	√	√	√	√	√
hsimd<fw> :: signmask	N/A	×	√	√	√	√	√	√
hsimd<fw> :: umin_hl	N/A	√	√	√	√	√	√	√
mvmd<fw> :: dslli	N/A	√	√	√	√	√	√	√
mvmd<fw> :: dsrli	N/A	√	√	√	√	√	√	√
mvmd<fw> :: extract	√	√	√	√	√	√	√	N/A
mvmd<fw> :: fill	√	√	√	√	√	√	√	√
mvmd<fw> :: fill2	√	√	√	√	√	√	√	N/A
mvmd<fw> :: fill4	√	√	√	√	√	√	N/A	N/A
mvmd<fw> :: fill8	√	√	√	√	√	N/A	N/A	N/A
mvmd<fw> :: fill16	√	√	√	√	N/A	N/A	N/A	N/A
mvmd<fw> :: shuffle	×	×	×	×	×	×	×	×
mvmd<fw> :: shufflei	×	×	×	×	√	√	√	×
mvmd<fw> :: slli	N/A	√	√	√	√	√	√	√
mvmd<fw> :: splat	√	√	√	√	√	√	√	√
mvmd<fw> :: srli	N/A	√	√	√	√	√	√	√
simd<fw> :: abs	N/A	√	√	√	√	√	√	√
simd<fw> :: add	√	√	√	√	√	√	√	√
simd<fw> :: add_hl	N/A	√	√	√	√	√	√	√
simd_and	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: constant	√	√	√	√	√	√	√	√
simd<fw> :: ctz	√	√	√	√	√	√	√	√
simd<fw> :: eq	√	√	√	√	√	√	√	√
simd<fw> :: gt	√	√	√	√	√	√	√	√
simd<fw> :: himask	N/A	√	√	√	√	√	√	√
simd<fw> :: ifh	√	√	√	√	√	√	√	√
simd<fw> :: lomask	N/A	√	√	√	√	√	√	√
simd<fw> :: lt	√	√	√	√	√	√	√	√
simd<fw> :: max	√	√	√	√	√	√	√	√

simd<fw> :: min	√	√	√	√	√	√	√	√
simd<fw> :: mult	√	√	√	√	√	√	√	√
simd<fw> :: neg	N/A	√	√	√	√	√	√	√
simd_nor	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_or	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: popcount	√	√	√	√	√	√	√	√
simd<fw> :: sll	×	×	×	×	×	×	√	√
simd<fw> :: slli	N/A	√	√	√	√	√	√	√
simd<fw> :: sra	×	×	×	×	×	×	×	×
simd<fw> :: srai	N/A	√	√	√	√	√	√	√
simd<fw> :: srl	×	×	×	×	×	×	√	√
simd<fw> :: srli	N/A	√	√	√	√	√	√	√
simd<fw> :: sub	√	√	√	√	√	√	√	√
simd<fw> :: ugt	√	√	√	√	√	√	√	√
simd<fw> :: ult	√	√	√	√	√	√	√	√
simd<fw> :: umax	√	√	√	√	√	√	√	√
simd<fw> :: umin	√	√	√	√	√	√	√	√
simd_xor	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: xor_hl	N/A	√	√	√	√	√	√	√

Table A.1: All the SSE2 Functions in IDISA+

Operation \ *fw*	1	2	4	8	16	32	64	128
bitblock :: all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: any	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: load_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: load_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: store_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: store_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√

esimd<fw> :: mergeh	√	√	√	√	√	√	√	N/A
esimd<fw> :: mergel	√	√	√	√	√	√	√	N/A
esimd<fw> :: multh	×	×	×	×	×	×	×	N/A
esimd<fw> :: multl	×	×	×	×	×	×	×	N/A
esimd<fw> :: signextendh	√	√	√	√	√	√	√	N/A
esimd<fw> :: signextendl	√	√	√	√	√	√	√	N/A
esimd<fw> :: zeroextendh	√	√	√	√	√	√	√	N/A
esimd<fw> :: zeroextendl	√	√	√	√	√	√	√	N/A
hsimd<fw> :: add_hl	N/A	√	√	√	√	√	√	√
hsimd<fw> :: min_hl	N/A	√	√	√	√	√	√	√
hsimd<fw> :: packh	N/A	√	√	√	√	√	√	√
hsimd<fw> :: packl	N/A	√	√	√	√	√	√	√
hsimd<fw> :: packss	N/A	√	√	√	√	√	√	√
hsimd<fw> :: packus	N/A	√	√	√	√	√	√	√
hsimd<fw> :: signmask	N/A	×	√	√	√	√	√	√
hsimd<fw> :: umin_hl	N/A	√	√	√	√	√	√	√
mvmd<fw> :: dslli	N/A	√	√	√	√	√	√	√
mvmd<fw> :: dsrli	N/A	√	√	√	√	√	√	√
mvmd<fw> :: extract	√	√	√	√	√	√	√	N/A
mvmd<fw> :: fill	√	√	√	√	√	√	√	×
mvmd<fw> :: fill2	√	√	√	√	√	√	√	N/A
mvmd<fw> :: fill4	√	√	√	√	√	√	N/A	N/A
mvmd<fw> :: fill8	√	√	√	√	√	N/A	N/A	N/A
mvmd<fw> :: fill16	√	√	√	√	N/A	N/A	N/A	N/A
mvmd<fw> :: shuffle	×	×	×	×	×	×	×	×
mvmd<fw> :: shufflei	×	×	×	×	×	×	×	×
mvmd<fw> :: slli	N/A	√	√	√	√	√	√	√
mvmd<fw> :: splat	√	√	√	√	√	√	√	√
mvmd<fw> :: srli	N/A	√	√	√	√	√	√	√
simd<fw> :: abs	N/A	√	√	√	√	√	√	√
simd<fw> :: add	√	√	√	√	√	√	√	√

simd<fw> :: add_hl	N/A	√	√	√	√	√	√	√
simd_and	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: constant	√	√	√	√	√	√	√	√
simd<fw> :: ctz	√	√	√	√	√	√	√	√
simd<fw> :: eq	√	√	√	√	√	√	√	√
simd<fw> :: gt	√	√	√	√	√	√	√	√
simd<fw> :: himask	N/A	√	√	√	√	√	√	√
simd<fw> :: ifh	√	√	√	√	√	√	√	√
simd<fw> :: lomask	N/A	√	√	√	√	√	√	√
simd<fw> :: lt	√	√	√	√	√	√	√	√
simd<fw> :: max	√	√	√	√	√	√	√	√
simd<fw> :: min	√	√	√	√	√	√	√	√
simd<fw> :: mult	√	√	√	√	√	√	×	×
simd<fw> :: neg	N/A	√	√	√	√	√	√	√
simd_nor	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_or	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: popcount	√	√	√	√	√	√	√	√
simd<fw> :: sll	×	×	×	×	×	×	×	×
simd<fw> :: slli	N/A	√	√	√	√	√	√	√
simd<fw> :: sra	×	×	×	×	×	×	×	×
simd<fw> :: srai	N/A	√	√	√	√	√	√	√
simd<fw> :: srl	×	×	×	×	×	×	×	×
simd<fw> :: srli	N/A	√	√	√	√	√	√	√
simd<fw> :: sub	√	√	√	√	√	√	√	√
simd<fw> :: ugt	√	√	√	√	√	√	√	√
simd<fw> :: ult	√	√	√	√	√	√	√	√
simd<fw> :: umax	√	√	√	√	√	√	√	√
simd<fw> :: umin	√	√	√	√	√	√	√	√
simd_xor	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A

simd<fw> :: xor_hl	N/A	√	√	√	√	√	√	√

Table A.2: All the NEON Functions in IDISA+

Operation \ *fw*	1	2	4	8	16	32	64	128	256
bitblock :: all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: any	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: load_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: load_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: store_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
bitblock :: store_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	√
esimd<fw> :: mergeh	√	√	√	√	√	√	√	√	N/A
esimd<fw> :: mergel	√	√	√	√	√	√	√	√	N/A
esimd<fw> :: multh	×	×	×	×	×	×	×	×	N/A
esimd<fw> :: multl	×	×	×	×	×	×	×	×	N/A
esimd<fw> :: signextendh	√	√	√	√	√	√	√	√	N/A
esimd<fw> :: signextendl	√	√	√	√	√	√	√	√	N/A
esimd<fw> :: zeroextendh	√	√	√	√	√	√	√	√	N/A
esimd<fw> :: zeroextendl	√	√	√	√	√	√	√	√	N/A
hsimd<fw> :: add_hl	N/A	√	√	√	√	√	√	√	√
hsimd<fw> :: min_hl	N/A	√	√	√	√	√	√	√	√
hsimd<fw> :: packh	N/A	√	√	√	√	√	√	√	√
hsimd<fw> :: packl	N/A	√	√	√	√	√	√	√	√
hsimd<fw> :: packss	N/A	√	√	√	√	√	√	√	√
hsimd<fw> :: packus	N/A	√	√	√	√	√	√	√	√
hsimd<fw> :: signmask	N/A	N/A	×	√	√	√	√	√	√
hsimd<fw> :: umin_hl	N/A	√	√	√	√	√	√	√	√
mvmd<fw> :: dslli	N/A	√	√	√	√	√	√	√	√
mvmd<fw> :: dsrli	N/A	√	√	√	√	√	√	√	√
mvmd<fw> :: extract	√	√	√	√	√	√	√	N/A	N/A

mvmd<fw> :: *fill*	√	√	√	√	√	√	×	×	×
mvmd<fw> :: *fill*2	√	√	√	√	√	√	×	×	N/A
mvmd<fw> :: *fill*4	√	√	√	√	√	√	×	N/A	N/A
mvmd<fw> :: *fill*8	√	√	√	√	√	√	N/A	N/A	N/A
mvmd<fw> :: *fill*16	√	√	√	√	√	N/A	N/A	N/A	N/A
mvmd<fw> :: *shuffle*	×	×	×	×	×	×	×	×	×
mvmd<fw> :: *shufflei*	×	×	×	×	×	×	×	×	×
mvmd<fw> :: *slli*	N/A	√	√	√	√	√	√	√	√
mvmd<fw> :: *splat*	√	√	√	√	√	√	√	√	√
mvmd<fw> :: *srli*	N/A	√	√	√	√	√	√	√	√
simd<fw> :: *abs*	N/A	√	√	√	√	√	√	√	√
simd<fw> :: *add*	√	√	√	√	√	√	√	√	√
simd<fw> :: *add_hl*	N/A	√	√	√	√	√	√	√	√
simd_and	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: *constant*	√	√	√	√	√	√	√	√	√
simd<fw> :: *ctz*	√	√	√	√	√	√	√	√	√
simd<fw> :: *eq*	√	√	√	√	√	√	√	√	√
simd<fw> :: *gt*	√	√	√	√	√	√	√	√	√
simd<fw> :: *himask*	N/A	√	√	√	√	√	√	√	√
simd<fw> :: *ifh*	√	√	√	√	√	√	√	√	√
simd<fw> :: *lomask*	N/A	√	√	√	√	√	√	√	√
simd<fw> :: *lt*	√	√	√	√	√	√	√	√	√
simd<fw> :: *max*	√	√	√	√	√	√	√	√	√
simd<fw> :: *min*	√	√	√	√	√	√	√	√	√
simd<fw> :: *mult*	√	√	√	√	√	√	√	√	√
simd<fw> :: *neg*	N/A	√	√	√	√	√	√	√	√
simd_nor	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_or	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: *popcount*	√	√	√	√	√	√	√	√	√

simd<fw> :: sll	×	×	×	×	×	×	×	×	×
simd<fw> :: slli	N/A	√	√	√	√	√	√	√	√
simd<fw> :: sra	×	×	×	×	×	×	×	×	×
simd<fw> :: srai	N/A	√	√	√	√	√	√	√	√
simd<fw> :: srl	×	×	×	×	×	×	×	×	×
simd<fw> :: srli	N/A	√	√	√	√	√	√	√	√
simd<fw> :: sub	√	√	√	√	√	√	√	√	√
simd<fw> :: ugt	√	√	√	√	√	√	√	√	√
simd<fw> :: ult	√	√	√	√	√	√	√	√	√
simd<fw> :: umax	√	√	√	√	√	√	√	√	√
simd<fw> :: umin	√	√	√	√	√	√	√	√	√
simd_xor	√	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: xor_hl	N/A	√	√	√	√	√	√	√	√

Table A.3: All the AVX Functions in IDISA+

Appendix B

Strategy Count

Operation \ *fw*	1	2	4	8	16	32	64	128
bitblock :: *all*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: *any*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: *load_aligned*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: *load_unaligned*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: *store_aligned*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: *store_unaligned*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
esimd<*fw*> :: *mergeh*	2	2	2	3	3	3	3	N/A
esimd<*fw*> :: *mergel*	2	2	2	3	3	3	3	N/A
esimd<*fw*> :: *multh*	0	0	0	0	0	0	0	N/A
esimd<*fw*> :: *multl*	0	0	0	0	0	0	0	N/A
esimd<*fw*> :: *signextendh*	2	2	2	2	2	2	2	N/A
esimd<*fw*> :: *signextendl*	1	1	1	1	1	1	1	N/A
esimd<*fw*> :: *zeroextendh*	2	2	2	2	2	2	2	N/A
esimd<*fw*> :: *zeroextendl*	1	1	1	1	1	1	1	N/A
hsimd<*fw*> :: *add_hl*	N/A	1	1	1	1	1	1	1
hsimd<*fw*> :: *min_hl*	N/A	1	1	1	1	1	1	1
hsimd<*fw*> :: *packh*	N/A	2	2	2	2	2	2	2
hsimd<*fw*> :: *packl*	N/A	2	2	2	2	2	3	4

hsimd<fw> :: packss	N/A	1	1	1	2	2	1	1
hsimd<fw> :: packus	N/A	2	2	2	3	2	2	2
hsimd<fw> :: signmask	N/A	1	2	3	2	2	2	2
hsimd<fw> :: umin_hl	N/A	1	1	1	1	1	1	2
mvmd<fw> :: dslli	N/A	1	1	1	1	1	1	1
mvmd<fw> :: dsrli	N/A	1	1	1	1	1	1	1
mvmd<fw> :: extract	1	2	2	2	3	2	1	N/A
mvmd<fw> :: fill	1	1	1	2	2	1	1	1
mvmd<fw> :: fill2	1	1	1	1	1	1	1	N/A
mvmd<fw> :: fill4	2	2	2	2	2	2	N/A	N/A
mvmd<fw> :: fill8	2	2	2	2	2	N/A	N/A	N/A
mvmd<fw> :: fill16	2	2	2	2	N/A	N/A	N/A	N/A
mvmd<fw> :: shuffle	0	1	1	1	1	1	1	0
mvmd<fw> :: shufflei	0	0	0	0	1	1	1	0
mvmd<fw> :: slli	N/A	2	2	3	2	2	2	2
mvmd<fw> :: splat	2	2	2	2	3	3	2	1
mvmd<fw> :: srli	N/A	2	2	3	2	2	2	2
simd<fw> :: abs	N/A	3	2	2	2	2	2	2
simd<fw> :: add	3	4	3	4	4	4	4	3
simd<fw> :: add_hl	N/A	5	3	3	2	2	2	2
simd_and	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: constant	1	1	1	2	2	1	1	1
simd<fw> :: ctz	2	2	1	1	1	1	1	1
simd<fw> :: eq	3	3	2	3	3	3	2	2
simd<fw> :: gt	5	6	5	6	6	6	4	3
simd<fw> :: himask	N/A	1	1	1	1	1	1	1
simd<fw> :: ifh	3	2	2	2	2	2	2	2
simd<fw> :: lomask	N/A	1	1	1	1	1	1	1
simd<fw> :: lt	6	7	6	6	6	6	5	4
simd<fw> :: max	6	7	6	6	7	6	5	4

simd<fw> :: min	6	7	6	6	7	6	5	4
simd<fw> :: mult	3	3	2	2	3	2	2	2
simd<fw> :: neg	N/A	2	1	1	1	1	1	1
simd_nor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_or	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: popcount	2	1	1	1	1	1	2	2
simd<fw> :: sll	0	0	0	0	0	0	1	1
simd<fw> :: slli	N/A	1	1	1	1	1	1	1
simd<fw> :: sra	0	0	0	0	0	0	0	0
simd<fw> :: srai	N/A	5	3	3	4	4	2	2
simd<fw> :: srl	0	0	0	0	0	0	1	1
simd<fw> :: srli	N/A	1	1	1	1	1	1	1
simd<fw> :: sub	3	4	3	4	4	4	4	3
simd<fw> :: ugt	7	7	6	6	6	6	6	5
simd<fw> :: ult	7	7	6	6	6	6	6	5
simd<fw> :: umax	5	5	4	5	4	4	4	3
simd<fw> :: umin	5	5	4	5	4	4	4	3
simd_xor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: xor_hl	N/A	2	1	1	1	1	1	1

Table B.1: The Number of Applicable Strategies for each SSE2 Function in IDISA+

Operation \ *fw*	1	2	4	8	16	32	64	128
bitblock :: all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: any	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: load_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: load_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: store_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1

bitblock :: *store_unaligned*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
esimd<*fw*> :: *mergeh*	2	2	2	3	3	3	3	N/A
esimd<*fw*> :: *mergel*	2	2	2	3	3	3	3	N/A
esimd<*fw*> :: *multh*	0	0	0	0	0	0	0	N/A
esimd<*fw*> :: *multl*	0	0	0	0	0	0	0	N/A
esimd<*fw*> :: *signextendh*	2	2	2	2	2	2	2	N/A
esimd<*fw*> :: *signextendl*	1	1	1	1	1	1	1	N/A
esimd<*fw*> :: *zeroextendh*	2	2	2	2	2	2	2	N/A
esimd<*fw*> :: *zeroextendl*	1	1	1	1	1	1	1	N/A
hsimd<*fw*> :: *add_hl*	N/A	1	1	1	1	1	1	1
hsimd<*fw*> :: *min_hl*	N/A	1	1	1	1	1	1	2
hsimd<*fw*> :: *packh*	N/A	2	2	2	3	3	4	2
hsimd<*fw*> :: *packl*	N/A	2	2	2	3	3	4	4
hsimd<*fw*> :: *packss*	N/A	1	1	1	1	1	1	1
hsimd<*fw*> :: *packus*	N/A	2	2	2	3	2	2	2
hsimd<*fw*> :: *signmask*	N/A	1	2	2	3	3	3	2
hsimd<*fw*> :: *umin_hl*	N/A	1	1	1	1	1	1	2
mvmd<*fw*> :: *dslli*	N/A	1	1	1	1	1	1	1
mvmd<*fw*> :: *dsrli*	N/A	1	1	1	1	1	1	1
mvmd<*fw*> :: *extract*	1	2	2	3	3	3	2	N/A
mvmd<*fw*> :: *fill*	1	1	1	2	2	1	1	0
mvmd<*fw*> :: *fill*2	1	1	1	1	1	1	1	N/A
mvmd<*fw*> :: *fill*4	2	2	2	2	2	2	N/A	N/A
mvmd<*fw*> :: *fill*8	2	2	2	2	1	N/A	N/A	N/A
mvmd<*fw*> :: *fill*16	2	2	2	1	N/A	N/A	N/A	N/A
mvmd<*fw*> :: *shuffle*	0	0	0	0	0	0	0	0
mvmd<*fw*> :: *shufflei*	0	0	0	0	0	0	0	0
mvmd<*fw*> :: *slli*	N/A	2	2	2	2	2	2	2
mvmd<*fw*> :: *splat*	3	3	3	3	3	3	3	2
mvmd<*fw*> :: *srli*	N/A	2	2	2	2	2	2	2
simd<*fw*> :: *abs*	N/A	3	2	3	3	3	2	2

simd<fw> :: add	3	4	3	4	4	4	4	3
simd<fw> :: add_hl	N/A	5	3	3	2	2	2	2
simd_and	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: constant	1	1	1	2	2	1	1	1
simd<fw> :: ctz	2	2	1	1	1	1	1	1
simd<fw> :: eq	3	3	2	3	3	3	2	2
simd<fw> :: gt	5	6	5	6	6	6	4	3
simd<fw> :: himask	N/A	1	1	1	1	1	1	1
simd<fw> :: ifh	4	2	2	2	2	2	2	2
simd<fw> :: lomask	N/A	1	1	1	1	1	1	1
simd<fw> :: lt	6	7	6	7	7	7	5	4
simd<fw> :: max	6	7	6	7	7	7	5	4
simd<fw> :: min	6	7	6	7	7	7	5	4
simd<fw> :: mult	3	3	2	3	3	3	2	2
simd<fw> :: neg	N/A	2	1	2	2	2	1	1
simd_nor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_or	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: popcount	2	1	1	2	1	1	1	2
simd<fw> :: sll	0	0	0	0	0	0	0	0
simd<fw> :: slli	N/A	1	1	2	1	1	1	1
simd<fw> :: sra	0	0	0	0	0	0	0	0
simd<fw> :: srai	N/A	5	3	4	4	4	3	2
simd<fw> :: srl	0	0	0	0	0	0	0	0
simd<fw> :: srli	N/A	1	1	2	1	1	1	1
simd<fw> :: sub	3	4	3	4	4	4	4	3
simd<fw> :: ugt	7	7	6	6	6	6	6	5
simd<fw> :: ult	7	7	6	6	6	6	6	5
simd<fw> :: umax	5	5	4	4	4	4	4	3
simd<fw> :: umin	5	5	4	4	4	4	4	3

simd_xor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: xor_hl	N/A	2	1	1	1	1	1	1

Table B.2: The Number of Applicable Strategies for each NEON Function in IDISA+

Operation \ *fw*	1	2	4	8	16	32	64	128	256
bitblock :: all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: any	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: load_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: load_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: store_aligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
bitblock :: store_unaligned	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1
esimd<fw> :: mergeh	2	2	2	3	3	3	3	2	N/A
esimd<fw> :: mergel	2	2	2	3	3	3	3	2	N/A
esimd<fw> :: multh	0	0	0	0	0	0	0	0	N/A
esimd<fw> :: multl	0	0	0	0	0	0	0	0	N/A
esimd<fw> :: signextendh	2	2	2	2	2	2	2	2	N/A
esimd<fw> :: signextendl	1	1	1	1	1	1	1	1	N/A
esimd<fw> :: zeroextendh	2	2	2	2	2	2	2	2	N/A
esimd<fw> :: zeroextendl	1	1	1	1	1	1	1	1	N/A
hsimd<fw> :: add_hl	N/A	1	1	1	1	2	2	1	1
hsimd<fw> :: min_hl	N/A	1	1	1	1	1	1	1	2
hsimd<fw> :: packh	N/A	2	2	2	2	2	2	1	2
hsimd<fw> :: packl	N/A	2	2	2	2	2	2	3	3
hsimd<fw> :: packss	N/A	1	1	1	2	2	1	1	1
hsimd<fw> :: packus	N/A	2	2	2	3	3	2	2	2
hsimd<fw> :: signmask	N/A	N/A	1	3	2	2	2	2	2
hsimd<fw> :: umin_hl	N/A	1	1	1	1	1	1	1	2
mvmd<fw> :: dslli	N/A	1	1	1	1	1	1	1	1

mvmd<fw> :: *dsrli*	N/A	1	1	1	1	1	1	1	1
mvmd<fw> :: *extract*	1	2	2	3	3	3	1	N/A	N/A
mvmd<fw> :: *fill*	1	1	1	2	2	1	0	0	0
mvmd<fw> :: *fill*2	1	1	1	1	1	1	1	1	N/A
mvmd<fw> :: *fill*4	2	2	2	2	2	3	1	N/A	N/A
mvmd<fw> :: *fill*8	2	2	2	2	3	1	N/A	N/A	N/A
mvmd<fw> :: *fill*16	2	2	2	3	1	N/A	N/A	N/A	N/A
mvmd<fw> :: *shuffle*	0	0	0	0	0	0	0	0	0
mvmd<fw> :: *shufflei*	0	0	0	0	0	0	0	0	0
mvmd<fw> :: *slli*	N/A	2	2	2	2	2	2	2	2
mvmd<fw> :: *splat*	2	2	2	3	3	3	2	2	1
mvmd<fw> :: *srli*	N/A	2	2	2	2	2	2	2	2
simd<fw> :: *abs*	N/A	3	2	3	3	3	2	2	2
simd<fw> :: *add*	3	4	3	4	4	4	4	3	3
simd<fw> :: *add_hl*	N/A	5	3	3	2	2	2	2	2
simd_and	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: *constant*	1	1	1	2	2	1	1	1	1
simd<fw> :: *ctz*	2	2	1	1	1	1	1	1	1
simd<fw> :: *eq*	3	3	2	3	3	3	3	2	2
simd<fw> :: *gt*	5	6	5	6	6	6	5	3	3
simd<fw> :: *himask*	N/A	1	1	1	1	1	1	1	1
simd<fw> :: *ifh*	3	2	2	2	2	2	3	2	3
simd<fw> :: *lomask*	N/A	1	1	1	1	1	1	1	1
simd<fw> :: *lt*	6	7	6	6	6	6	5	4	4
simd<fw> :: *max*	6	7	6	7	7	7	5	4	4
simd<fw> :: *min*	6	7	6	7	7	7	5	4	4
simd<fw> :: *mult*	3	3	2	2	3	3	2	2	2
simd<fw> :: *neg*	N/A	2	1	1	1	1	1	1	1
simd_nor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

simd_or	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: popcount	2	1	1	1	1	1	2	1	2
simd<fw> :: sll	0	0	0	0	0	0	0	0	0
simd<fw> :: slli	N/A	1	1	1	1	1	1	1	1
simd<fw> :: sra	0	0	0	0	0	0	0	0	0
simd<fw> :: srai	N/A	5	3	3	4	4	2	2	2
simd<fw> :: srl	0	0	0	0	0	0	0	0	0
simd<fw> :: srli	N/A	1	1	1	1	1	1	1	1
simd<fw> :: sub	3	4	3	4	4	4	4	3	3
simd<fw> :: ugt	7	7	6	6	6	6	6	5	5
simd<fw> :: ult	7	7	6	6	6	6	6	5	5
simd<fw> :: umax	5	5	4	5	5	5	4	3	3
simd<fw> :: umin	5	5	4	5	5	5	4	3	3
simd_xor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: xor_hl	N/A	2	1	1	1	1	1	1	1

Table B.3: The Number of Applicable Strategies for each AVX Function in IDISA+

Appendix C

Instruction Count

Operation \ *fw*	1	2	4	8	16	32	64	128
bitblock :: all	×	×	×	×	×	×	×	2
bitblock :: any	×	×	×	×	×	×	×	2
bitblock :: load_aligned	×	×	×	×	×	×	×	1
bitblock :: load_unaligned	×	×	×	×	×	×	×	1
bitblock :: store_aligned	×	×	×	×	×	×	×	1
bitblock :: store_unaligned	×	×	×	×	×	×	×	1
esimd<fw> :: mergeh	31	21	11	1	1	1	1	×
esimd<fw> :: mergel	31	21	11	1	1	1	1	×
esimd<fw> :: multh	×	×	×	×	×	×	×	×
esimd<fw> :: multl	×	×	×	×	×	×	×	×
esimd<fw> :: signextendh	31	33	13	4	4	12	21	×
esimd<fw> :: signextendl	31	33	13	4	4	12	25	×
esimd<fw> :: zeroextendh	24	14	4	3	3	3	4	×
esimd<fw> :: zeroextendl	24	14	4	3	3	3	1	×
hsimd<fw> :: add_hl	×	93	74	42	7	41	13	15
hsimd<fw> :: min_hl	×	93	82	45	10	41	16	32
hsimd<fw> :: packh	×	47	33	19	3	21	7	7
hsimd<fw> :: packl	×	45	31	17	3	19	5	7
hsimd<fw> :: packss	×	120	86	36	1	1	79	288

hsimd<fw> :: packus	×	85	83	35	1	37	18	38
hsimd<fw> :: signmask	×	×	24	1	4	25	32	39
hsimd<fw> :: umin_hl	×	93	80	42	7	44	19	34
mvmd<fw> :: dslli	×	9	9	3	3	3	3	3
mvmd<fw> :: dsrli	×	9	9	3	3	3	3	3
mvmd<fw> :: extract	1	1	1	1	1	2	4	×
mvmd<fw> :: fill	1	1	1	1	1	1	1	1
mvmd<fw> :: fill16	15	7	3	1	×	×	×	×
mvmd<fw> :: fill2	1	1	1	1	1	5	5	×
mvmd<fw> :: fill4	5	5	5	5	3	1	×	×
mvmd<fw> :: fill8	13	13	7	3	1	×	×	×
mvmd<fw> :: shuffle	×	×	×	×	×	×	×	×
mvmd<fw> :: shufflei	×	×	×	×	17	1	1	×
mvmd<fw> :: slli	×	4	4	1	1	1	1	1
mvmd<fw> :: splat	16	13	9	5	2	1	5	13
mvmd<fw> :: srli	×	4	4	1	1	1	1	1
simd<fw> :: abs	×	9	19	5	5	5	17	49
simd<fw> :: add	1	10	6	1	1	1	1	11
simd<fw> :: add_hl	×	3	4	4	3	3	3	16
simd_and	1	×	×	×	×	×	×	×
simd_andc	1	×	×	×	×	×	×	×
simd<fw> :: constant	0	0	0	0	0	0	0	0
simd<fw> :: ctz	1	14	14	13	16	19	14	30
simd<fw> :: eq	2	8	9	1	1	1	5	15
simd<fw> :: gt	1	15	10	1	1	1	15	66
simd<fw> :: himask	×	0	0	0	0	0	0	0
simd<fw> :: ifh	3	8	13	4	4	4	8	15
simd<fw> :: lomask	×	0	0	0	0	0	0	0
simd<fw> :: lt	1	15	18	5	5	5	20	75
simd<fw> :: max	1	18	9	4	1	4	18	65
simd<fw> :: min	1	18	9	4	1	4	18	65

$simd<fw>::mult$	1	23	31	10	1	30	11	205
$simd<fw>::neg$	×	8	6	1	1	1	1	11
$simd\_nor$	2	×	×	×	×	×	×	×
$simd\_not$	1	×	×	×	×	×	×	×
$simd\_or$	1	×	×	×	×	×	×	×
$simd<fw>::popcount$	0	3	7	11	14	17	12	18
$simd<fw>::sll$	×	×	×	×	×	×	7	11
$simd<fw>::slli$	×	2	2	2	1	1	1	4
$simd<fw>::sra$	×	×	×	×	×	×	×	×
$simd<fw>::srai$	×	4	10	5	1	1	5	21
$simd<fw>::srl$	×	×	×	×	×	×	7	11
$simd<fw>::srli$	×	2	2	2	1	1	1	4
$simd<fw>::sub$	1	11	6	1	1	1	1	11
$simd<fw>::ugt$	1	14	12	3	3	3	14	51
$simd<fw>::ult$	1	14	20	7	7	7	18	55
$simd<fw>::umax$	1	16	6	1	4	7	20	47
$simd<fw>::umin$	1	16	6	1	4	7	20	47
$simd\_xor$	1	×	×	×	×	×	×	×
$simd<fw>::xor\_hl$	×	4	4	4	3	3	3	6

Table C.1: The Estimated Number of Instructions for Each Best SSE2 Function in IDISA+

Operation \ fw	1	2	4	8	16	32	64	128
$bitblock::all$	×	×	×	×	×	×	×	2
$bitblock::any$	×	×	×	×	×	×	×	2
$bitblock::load\_aligned$	×	×	×	×	×	×	×	1
$bitblock::load\_unaligned$	×	×	×	×	×	×	×	1
$bitblock::store\_aligned$	×	×	×	×	×	×	×	1
$bitblock::store\_unaligned$	×	×	×	×	×	×	×	1

$esimd<fw> :: mergeh$	31	21	11	9	9	9	9	×
$esimd<fw> :: mergel$	31	21	11	9	9	9	9	×
$esimd<fw> :: multh$	×	×	×	×	×	×	×	×
$esimd<fw> :: multl$	×	×	×	×	×	×	×	×
$esimd<fw> :: signextendh$	35	37	17	8	8	16	29	×
$esimd<fw> :: signextendl$	31	33	13	4	4	12	25	×
$esimd<fw> :: zeroextendh$	28	18	8	7	7	7	5	×
$esimd<fw> :: zeroextendl$	24	14	4	3	3	3	1	×
$hsimd<fw> :: add\_hl$	×	93	74	42	7	41	13	15
$hsimd<fw> :: min\_hl$	×	93	82	45	10	41	16	38
$hsimd<fw> :: packh$	×	89	87	39	5	39	41	83
$hsimd<fw> :: packl$	×	45	31	17	33	19	21	7
$hsimd<fw> :: packss$	×	120	86	36	21	37	79	288
$hsimd<fw> :: packus$	×	97	88	60	51	46	39	75
$hsimd<fw> :: signmask$	×	×	24	1	4	25	32	39
$hsimd<fw> :: umin\_hl$	×	93	80	42	7	44	19	51
$mvmd<fw> :: dslli$	×	9	9	3	3	3	3	3
$mvmd<fw> :: dsrli$	×	9	9	3	3	3	3	3
$mvmd<fw> :: extract$	1	1	1	1	1	2	4	×
$mvmd<fw> :: fill$	1	1	1	1	1	1	1	1
$mvmd<fw> :: fill16$	29	29	17	9	×	×	×	×
$mvmd<fw> :: fill2$	1	1	1	1	1	5	5	×
$mvmd<fw> :: fill4$	11	11	11	7	5	13	×	×
$mvmd<fw> :: fill8$	27	15	13	13	9	×	×	×
$mvmd<fw> :: shuffle$	×	×	×	×	×	×	×	×
$mvmd<fw> :: shufflei$	×	×	×	×	17	1	1	×
$mvmd<fw> :: slli$	×	4	4	4	4	4	4	4
$mvmd<fw> :: splat$	17	35	73	149	4	7	5	13
$mvmd<fw> :: srli$	×	4	4	4	4	4	4	4
$simd<fw> :: abs$	×	29	30	27	13	13	19	80
$simd<fw> :: add$	24	13	28	6	6	6	8	15

simd<fw> :: add_hl	×	7	9	4	3	3	13	16
simd_and	1	×	×	×	×	×	×	×
simd_andc	1	×	×	×	×	×	×	×
simd<fw> :: constant	0	0	0	0	0	0	0	0
simd<fw> :: ctz	1	15	14	13	16	19	14	30
simd<fw> :: eq	37	15	14	9	5	5	5	15
simd<fw> :: gt	3	16	13	8	8	11	16	66
simd<fw> :: himask	×	0	0	0	0	0	0	0
simd<fw> :: ifh	4	18	13	18	8	8	18	69
simd<fw> :: lomask	×	0	0	0	0	0	0	0
simd<fw> :: lt	3	16	21	16	13	13	23	84
simd<fw> :: max	4	19	13	8	4	8	23	69
simd<fw> :: min	4	19	13	8	4	8	23	69
simd<fw> :: mult	57	73	439	204	68	38	427	205
simd<fw> :: neg	×	11	6	1	1	1	1	11
simd_nor	2	×	×	×	×	×	×	×
simd_not	1	×	×	×	×	×	×	×
simd_or	1	×	×	×	×	×	×	×
simd<fw> :: popcount	0	3	7	11	14	17	20	28
simd<fw> :: sll	×	×	×	×	×	×	7	11
simd<fw> :: slli	×	2	2	2	1	1	1	4
simd<fw> :: sra	×	×	×	×	×	×	×	×
simd<fw> :: srai	×	8	12	7	4	4	10	24
simd<fw> :: srl	×	×	×	×	×	×	7	11
simd<fw> :: srli	×	2	2	2	1	1	1	4
simd<fw> :: sub	26	13	30	6	6	6	8	15
simd<fw> :: ugt	3	17	13	12	10	10	17	56
simd<fw> :: ult	3	17	21	15	11	11	19	55
simd<fw> :: umax	4	18	9	7	14	17	21	47
simd<fw> :: umin	4	18	9	7	14	17	21	47
simd_xor	1	×	×	×	×	×	×	×

$simd<fw> :: xor\_hl$	×	6	4	4	3	3	3	6

Table C.2: The Estimated Number of Instructions for Each Second Best SSE2 Function in IDISA+

Operation \ fw	1	2	4	8	16	32	64	128
$bitblock :: all$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2
$bitblock :: any$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3
$esimd<fw> :: mergeh$	31	21	11	1	1	1	1	N/A
$esimd<fw> :: mergel$	31	21	11	1	1	1	1	N/A
$esimd<fw> :: signextendh$	31	34	14	4	4	12	16	N/A
$esimd<fw> :: signextendl$	31	34	14	4	4	12	17	N/A
$esimd<fw> :: zeroextendh$	24	14	4	3	3	3	1	N/A
$esimd<fw> :: zeroextendl$	24	14	4	3	3	3	1	N/A
$hsimd<fw> :: add\_hl$	N/A	91	72	40	7	29	13	9
$hsimd<fw> :: min\_hl$	N/A	91	80	43	10	29	16	27
$hsimd<fw> :: packh$	N/A	47	33	19	3	15	7	4
$hsimd<fw> :: packl$	N/A	45	31	17	3	13	5	4
$hsimd<fw> :: packss$	N/A	120	76	36	1	1	73	175
$hsimd<fw> :: packus$	N/A	82	76	34	1	30	18	31
$hsimd<fw> :: signmask$	N/A	N/A	10	1	4	13	17	20
$hsimd<fw> :: umin\_hl$	N/A	91	78	40	7	32	19	28
$mvmd<fw> :: dslli$	N/A	7	3	3	2	3	2	2
$mvmd<fw> :: dsrli$	N/A	7	2	3	3	2	2	2
$mvmd<fw> :: extract$	1	1	1	1	1	2	4	N/A
$mvmd<fw> :: fill$	0	0	0	0	0	0	0	0
$mvmd<fw> :: fill16$	8	3	0	0	N/A	N/A	N/A	N/A
$mvmd<fw> :: fill2$	0	0	0	0	0	3	3	N/A
$mvmd<fw> :: fill4$	2	3	3	3	0	0	N/A	N/A
$mvmd<fw> :: fill8$	3	7	3	0	0	N/A	N/A	N/A

mvmd<fw> :: shufflei	N/A	N/A	N/A	N/A	10	1	1	N/A
mvmd<fw> :: slli	N/A	2	4	1	1	1	1	1
mvmd<fw> :: splat	11	15	11	7	4	1	5	13
mvmd<fw> :: srli	N/A	1	1	1	0	1	0	0
simd<fw> :: abs	N/A	9	17	6	6	6	18	41
simd<fw> :: add	1	10	6	1	1	1	1	8
simd<fw> :: add_hl	N/A	3	4	4	3	3	3	10
simd_and	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: constant	0	0	0	0	0	0	0	0
simd<fw> :: ctz	1	14	14	13	16	19	15	25
simd<fw> :: eq	2	8	9	1	1	1	5	9
simd<fw> :: gt	1	15	10	1	1	1	16	44
simd<fw> :: himask	N/A	0	0	0	0	0	0	0
simd<fw> :: ifh	3	8	11	5	5	5	9	13
simd<fw> :: lomask	N/A	0	0	0	0	0	0	0
simd<fw> :: lt	1	15	18	5	5	5	21	61
simd<fw> :: max	1	18	9	4	1	4	19	57
simd<fw> :: min	1	18	9	4	1	4	19	57
simd<fw> :: mult	1	23	31	10	1	28	10	125
simd<fw> :: neg	N/A	8	7	2	2	2	2	9
simd_nor	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_or	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: popcount	0	3	7	11	14	17	13	16
simd<fw> :: sll	N/A	N/A	N/A	N/A	N/A	N/A	6	9
simd<fw> :: slli	N/A	2	0	2	1	1	1	1
simd<fw> :: srai	N/A	4	11	6	1	1	6	13
simd<fw> :: srl	N/A	N/A	N/A	N/A	N/A	N/A	6	9
simd<fw> :: srli	N/A	2	2	0	1	1	1	4
simd<fw> :: sub	1	11	6	1	1	1	1	8

simd<fw> :: ugt	1	14	12	3	3	3	15	38
simd<fw> :: ult	1	14	20	7	7	7	19	37
simd<fw> :: umax	1	16	6	1	4	7	20	41
simd<fw> :: umin	1	16	6	1	4	7	20	41
simd_xor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: xor_hl	N/A	3	3	3	3	3	3	3

Table C.3: The Real Number of Non-Movement Instructions for Each Best SSE2 Function in IDISA+

Operation \ *fw*	1	2	4	8	16	32	64	128
bitblock :: all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2
bitblock :: any	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3
esimd<fw> :: mergeh	44	30	16	2	2	2	2	N/A
esimd<fw> :: mergel	44	30	16	2	2	2	2	N/A
esimd<fw> :: signextendh	45	50	20	6	6	21	22	N/A
esimd<fw> :: signextendl	45	50	20	6	6	21	23	N/A
esimd<fw> :: zeroextendh	35	21	8	5	5	5	2	N/A
esimd<fw> :: zeroextendl	35	21	8	5	5	5	2	N/A
hsimd<fw> :: add_hl	N/A	133	107	59	10	43	18	15
hsimd<fw> :: min_hl	N/A	133	118	64	16	43	24	40
hsimd<fw> :: packh	N/A	70	49	28	4	23	10	8
hsimd<fw> :: packl	N/A	68	47	26	5	21	8	8
hsimd<fw> :: packss	N/A	172	115	55	2	2	113	275
hsimd<fw> :: packus	N/A	122	117	50	2	45	28	50
hsimd<fw> :: signmask	N/A	N/A	13	1	4	17	22	26
hsimd<fw> :: umin_hl	N/A	133	114	59	10	48	28	45
mvmd<fw> :: dslli	N/A	9	4	4	3	4	3	3
mvmd<fw> :: dsrli	N/A	9	3	4	4	3	3	3
mvmd<fw> :: extract	1	1	1	1	1	2	4	N/A

mvmd<fw> :: fill	0	0	0	0	0	0	0	0
mvmd<fw> :: fill16	12	7	0	0	N/A	N/A	N/A	N/A
mvmd<fw> :: fill2	0	0	0	0	0	6	6	N/A
mvmd<fw> :: fill4	5	6	6	6	0	0	N/A	N/A
mvmd<fw> :: fill8	6	15	7	0	0	N/A	N/A	N/A
mvmd<fw> :: shufflei	N/A	N/A	N/A	N/A	15	2	2	N/A
mvmd<fw> :: slli	N/A	3	6	2	2	2	2	2
mvmd<fw> :: splat	14	22	17	12	8	2	9	21
mvmd<fw> :: srli	N/A	2	2	2	1	2	1	1
simd<fw> :: abs	N/A	16	29	12	12	12	29	62
simd<fw> :: add	2	17	11	2	2	2	2	12
simd<fw> :: add_hl	N/A	5	7	7	5	5	5	15
simd_and	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: constant	0	0	0	0	0	0	0	0
simd<fw> :: ctz	2	23	23	21	26	31	23	39
simd<fw> :: eq	4	11	14	2	2	2	8	14
simd<fw> :: gt	2	22	16	2	2	2	22	68
simd<fw> :: himask	N/A	0	0	0	0	0	0	0
simd<fw> :: ifh	6	13	17	8	8	8	14	21
simd<fw> :: lomask	N/A	0	0	0	0	0	0	0
simd<fw> :: lt	2	22	27	8	8	8	30	91
simd<fw> :: max	2	28	13	8	2	8	28	86
simd<fw> :: min	2	28	13	8	2	8	28	86
simd<fw> :: mult	2	35	48	16	2	43	16	190
simd<fw> :: neg	N/A	15	12	3	3	3	3	12
simd_nor	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_or	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: popcount	0	5	11	18	23	28	20	25
simd<fw> :: sll	N/A	N/A	N/A	N/A	N/A	N/A	17	21

simd<fw> :: slli	N/A	3	0	3	2	2	2	2
simd<fw> :: srai	N/A	6	17	8	2	2	8	17
simd<fw> :: srl	N/A	N/A	N/A	N/A	N/A	N/A	17	21
simd<fw> :: srli	N/A	3	3	0	2	2	2	6
simd<fw> :: sub	2	18	11	2	2	2	2	11
simd<fw> :: ugt	2	20	19	5	5	5	20	54
simd<fw> :: ult	2	20	32	11	11	11	27	52
simd<fw> :: umax	2	25	9	2	6	12	33	65
simd<fw> :: umin	2	25	9	2	6	12	33	65
simd_xor	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: xor_hl	N/A	5	5	5	5	5	5	5

Table C.4: The Real Number of All Instructions for Each Best SSE2 Function in IDISA+

Operation \ *fw*	1	2	4	8	16	32	64	128
bitblock :: all	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2
bitblock :: any	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3
esimd<fw> :: mergeh	31	21	11	9	9	9	9	N/A
esimd<fw> :: mergel	31	21	11	9	9	9	9	N/A
esimd<fw> :: signextendh	32	35	15	5	5	13	18	N/A
esimd<fw> :: signextendl	31	34	14	4	4	12	17	N/A
esimd<fw> :: zeroextendh	25	15	5	4	4	4	2	N/A
esimd<fw> :: zeroextendl	24	14	4	3	3	3	1	N/A
hsimd<fw> :: add_hl	N/A	91	72	40	7	29	13	9
hsimd<fw> :: min_hl	N/A	91	80	43	10	29	16	27
hsimd<fw> :: packh	N/A	86	80	38	5	32	40	55
hsimd<fw> :: packl	N/A	45	31	17	21	13	12	4
hsimd<fw> :: packss	N/A	120	76	36	21	31	73	175
hsimd<fw> :: packus	N/A	94	81	51	38	34	38	53

hsimd<fw> :: signmask	N/A	N/A	10	1	4	13	17	20
hsimd<fw> :: umin_hl	N/A	91	78	40	7	32	19	45
mvmd<fw> :: dslli	N/A	7	3	3	2	3	2	2
mvmd<fw> :: dsrli	N/A	7	2	3	3	2	2	2
mvmd<fw> :: extract	1	1	1	1	1	2	4	N/A
mvmd<fw> :: fill	0	0	0	0	0	0	0	0
*mvmd<fw> :: fill*16	3	18	9	5	N/A	N/A	N/A	N/A
*mvmd<fw> :: fill*2	0	0	0	0	0	3	3	N/A
*mvmd<fw> :: fill*4	7	7	7	3	3	9	N/A	N/A
*mvmd<fw> :: fill*8	7	8	9	9	5	N/A	N/A	N/A
mvmd<fw> :: shufflei	N/A	N/A	N/A	N/A	10	1	1	N/A
mvmd<fw> :: slli	N/A	2	4	1	1	1	0	1
mvmd<fw> :: splat	19	25	51	103	4	11	5	13
mvmd<fw> :: srli	N/A	1	1	1	1	1	0	1
simd<fw> :: abs	N/A	25	29	25	14	14	19	54
simd<fw> :: add	24	11	26	6	6	6	8	15
simd<fw> :: add_hl	N/A	7	9	4	3	3	10	10
simd_and	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: constant	0	0	0	0	0	0	0	0
simd<fw> :: ctz	1	15	14	13	16	19	15	25
simd<fw> :: eq	35	15	14	9	5	5	5	9
simd<fw> :: gt	3	16	13	8	8	11	17	44
simd<fw> :: himask	N/A	0	0	0	0	0	0	0
simd<fw> :: ifh	×	10	11	16	9	9	18	46
simd<fw> :: lomask	N/A	0	0	0	0	0	0	0
simd<fw> :: lt	3	16	21	16	13	13	23	50
simd<fw> :: max	3	19	13	8	4	8	24	47
simd<fw> :: min	4	19	13	8	4	8	24	47
simd<fw> :: mult	57	73	IF	162	64	31	258	125
simd<fw> :: neg	N/A	12	7	2	2	2	2	9

simd_nor	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_not	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_or	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<*fw*> :: *popcount*	0	3	7	11	14	17	20	23
simd<*fw*> :: *sll*	N/A	N/A	N/A	N/A	N/A	N/A	6	9
simd<*fw*> :: *slli*	N/A	2	0	2	1	1	1	1
simd<*fw*> :: *srai*	N/A	9	13	8	5	5	7	18
simd<*fw*> :: *srl*	N/A	N/A	N/A	N/A	N/A	N/A	6	9
simd<*fw*> :: *srli*	N/A	2	2	0	1	1	1	4
simd<*fw*> :: *sub*	25	11	29	6	6	6	8	15
simd<*fw*> :: *ugt*	3	16	13	12	10	10	18	38
simd<*fw*> :: *ult*	3	17	21	16	11	11	20	37
simd<*fw*> :: *umax*	4	18	9	7	14	17	22	41
simd<*fw*> :: *umin*	4	18	9	7	14	17	22	41
simd_xor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<*fw*> :: *xor_hl*	N/A	6	3	3	3	3	3	3

Table C.5: The Real Number of Non-Movement Instructions for Each Second Best SSE2 Function in IDISA+

Operation \ *fw*	1	2	4	8	16	32	64	128
bitblock :: *all*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2
bitblock :: *any*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3
esimd<*fw*> :: *mergeh*	44	30	16	14	14	14	14	N/A
esimd<*fw*> :: *mergel*	44	30	16	14	14	14	14	N/A
esimd<*fw*> :: *signextendh*	46	51	21	7	7	22	24	N/A
esimd<*fw*> :: *signextendl*	45	50	20	6	6	21	23	N/A
esimd<*fw*> :: *zeroextendh*	36	22	9	6	6	6	3	N/A
esimd<*fw*> :: *zeroextendl*	35	21	8	5	5	5	2	N/A
hsimd<*fw*> :: *add_hl*	N/A	133	107	59	10	43	18	15

hsimd<fw> :: min_hl	N/A	133	118	64	16	43	24	40
hsimd<fw> :: packh	N/A	125	121	54	7	47	64	88
hsimd<fw> :: packl	N/A	68	47	26	34	21	21	8
hsimd<fw> :: packss	N/A	172	115	55	33	49	113	275
hsimd<fw> :: packus	N/A	136	118	75	58	53	62	86
hsimd<fw> :: signmask	N/A	N/A	13	1	4	17	22	26
hsimd<fw> :: umin_hl	N/A	133	114	59	10	48	28	74
mvmd<fw> :: dslli	N/A	9	4	4	3	4	3	3
mvmd<fw> :: dsrli	N/A	9	3	4	4	3	3	3
mvmd<fw> :: extract	1	1	1	1	1	2	4	N/A
mvmd<fw> :: fill	0	0	0	0	0	0	0	0
*mvmd<fw> :: fill*16	7	31	19	12	N/A	N/A	N/A	N/A
*mvmd<fw> :: fill*2	0	0	0	0	0	6	6	N/A
*mvmd<fw> :: fill*4	13	13	13	7	6	18	N/A	N/A
*mvmd<fw> :: fill*8	12	12	18	18	12	N/A	N/A	N/A
mvmd<fw> :: shufflei	N/A	N/A	N/A	N/A	15	2	2	N/A
mvmd<fw> :: slli	N/A	3	6	2	2	2	1	2
mvmd<fw> :: splat	27	37	78	158	6	21	9	21
mvmd<fw> :: srli	N/A	2	2	2	2	2	1	2
simd<fw> :: abs	N/A	36	46	40	25	25	29	79
simd<fw> :: add	38	16	39	11	11	11	12	23
simd<fw> :: add_hl	N/A	13	16	7	5	5	15	15
simd_and	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd_andc	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
simd<fw> :: constant	0	0	0	0	0	0	0	0
simd<fw> :: ctz	2	23	23	21	26	31	23	39
simd<fw> :: eq	52	24	19	14	8	8	8	14
simd<fw> :: gt	6	22	19	14	14	21	23	68
simd<fw> :: himask	N/A	0	0	0	0	0	0	0
simd<fw> :: ifh	×	14	17	25	14	14	25	68
simd<fw> :: lomask	N/A	0	0	0	0	0	0	0

$simd<fw> :: lt$	5	22	30	26	20	20	32	75
$simd<fw> :: max$	4	32	21	14	8	14	36	71
$simd<fw> :: min$	8	32	21	14	8	14	36	71
$simd<fw> :: mult$	85	109	IF	217	98	42	391	190
$simd<fw> :: neg$	N/A	19	12	3	3	3	3	12
$simd\_nor$	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$simd\_not$	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$simd\_or$	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$simd<fw> :: popcount$	0	5	11	18	23	28	32	35
$simd<fw> :: sll$	N/A	N/A	N/A	N/A	N/A	N/A	17	21
$simd<fw> :: slli$	N/A	3	0	3	2	2	2	2
$simd<fw> :: srai$	N/A	14	19	10	7	7	11	28
$simd<fw> :: srl$	N/A	N/A	N/A	N/A	N/A	N/A	17	21
$simd<fw> :: srli$	N/A	3	3	0	2	2	2	6
$simd<fw> :: sub$	40	16	43	11	11	11	11	24
$simd<fw> :: ugt$	5	25	19	19	14	14	25	54
$simd<fw> :: ult$	6	24	33	22	17	17	28	52
$simd<fw> :: umax$	6	31	14	12	23	28	32	65
$simd<fw> :: umin$	6	31	14	12	23	28	32	65
$simd\_xor$	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$simd<fw> :: xor\_hl$	N/A	9	5	5	5	5	5	5

Table C.6: The Real Number of All Instructions for Each Second Best SSE2 Function in IDISA+

Bibliography

[1] Intel AVX C/C++ Intrinsics Emulation. http://software.intel.com/en-us/articles/avx-emulation-header-file/.

[2] Intel IA-64 Wikipedia Page. http://en.wikipedia.org/wiki/Itanium.

[3] Intel Sandy Bridge Wikipedia Page. http://en.wikipedia.org/wiki/Sandy_Bridge.

[4] libSIMD Homepage. http://libsimd.sourceforge.net/.

[5] macstl Homepage. http://www.pixelglow.com/macstl/.

[6] PowerPC Wikipedia Page. http://en.wikipedia.org/wiki/PowerPC.

[7] SIMD Wikipedia Page. http://en.wikipedia.org/wiki/SIMD.

[8] SIMDx86 Homepage. http://simdx86.sourceforge.net/.

[9] SSE3 Wikipedia Page. http://en.wikipedia.org/wiki/SSE3.

[10] SSE4a at AMD Developer Central. http://blogs.amd.com/developer/tag/sse4a/.

[11] James Abel, Kumar Balasubramanian, Mike Bargeron, Tom Craver, and Mike Phlipot. Applications tuning for streaming SIMD extensions. Intel Technology Journal, (Q2):13, May 1999.

[12] Advanced Micro Devices, Inc. 3DNow! Technology Manual, 2000. Order number 21928G/0.

[13] Apple Inc. Accelerate Framework Reference, 2011.

[14] ARM Corporation. ARM RealView Compilation Tools Assembler Guide. ARM Corporation, 2002-2010.

[15] ARM Corporation. ARM NEON support in the ARM compiler. White Paper, ARM Corporation, 2008.

[16] ARM Corporation. Introducing NEON Development Article. ARM Corporation, 2009.

[17] Robert D. Cameron, Ehsan Amiri, Kenneth S. Herdy, Dan Lin, Thomas C. Shermer, and Fred Popowich. Parallel scanning with bitstream addition: An XML case study. Technical Report 2010-11, School of Computing Science, Simon Fraser University, October 2010.

[18] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. High performance XML parsing using parallel bit stream technology. In Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, page 17. IBM, 2008.

[19] Robert D. Cameron and Dan Lin. Architectural support for SWAR text processing with parallel bit streams: the inductive doubling principle. ACM SIGPLAN Notices, 44(3):337–348, March 2009.

[20] Nathan Clark, Amir Hormati, Sami Yehia, Scott A. Mahlke, and Krisztián Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In HPCA, pages 216–227. IEEE Computer Society, 2007.

[21] Joel Falcou and Jocelyn Serot. E.V.E., an object oriented SIMD library. Scalable Computing: Practice and Experience, 6(4):31–42, December 2005.

[22] Randall J. Fisher and Henry G. Dietz. Compiling for SIMD within a register. In LCPC, volume 1656 of Lecture Notes in Computer Science, pages 290–304. Springer, 1998.

[23] Randall James Fisher. General-purpose SIMD within a register: Parallel processing on consumer microprocessors. PhD thesis, Purdue University, January 01 2003.

[24] Michael Flynn. Some computer organizations and their effectiveness. IEEETC:JOURNAL, 21(9):948–960, 1972.

[25] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. 2011.

[26] Sam Fuller. Motorola's AltiVec technology. Technical report, Motorola Corporation, 1998.

[27] Mathias Gaunard, Joel Falcou, and Jean-Thierry Lapreste. Practical simd acceleration with boost.simd. Boostcon, 2011.

[28] Apple Inc. Taking advantage of the accelerate framework. Technical report, 2011.

[29] Intel. Intel IA-32 software developer's manual, 2002.

[30] Intel Corporation. Intel SSE4 Programming Reference, 2007.

[31] Intel Corporation. Intel Advanced Vector Extensions Programming Reference, 2011.

[32] Intel Corporation. Intel IA-64 Architecture Software Developer's Manual: Combined Volumes: 1, 2A, 2B, 3A and 3B. Intel Corporation, May 2011.

[33] Ellsworth David Johnson. Quine-mccluskey: a computerized approach to boolean algebraic optimization. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1982.

[34] Robert L. Bocchino Jr. and Vikram S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In Proceedings of the 2nd International Conference on Virtual Execution Environments, pages 46–56. ACM, 2006.

[35] Samuel Larsen. Compilation techniques for short-vector instructions. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2006.

[36] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, pages 145–156, Vancouver, British Columbia, June 18–21, 2000.

[37] Chris Lattner and Vikram Adve. The LLVM instruction set and compilation strategy, October 2002.

[38] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation, January 2004.

[39] Chris Lattner and Vikram S. Adve. The LLVM compiler framework and infrastructure tutorial. In LCPC, volume 3602 of Lecture Notes in Computer Science. Springer, 2004.

[40] Ruby Lee. Effectiveness of the MAX-2 multimedia extensions for PA-RISC 2.0 processors. IEEE Computer Society Press, 1997.

[41] Ruby B. Lee. Accelerating multimedia with enhanced microprocessors. IEEE Micro, 15(2):22–32, April 1995. Presented at Hot Chips VI, Stanford University, CA, August 14–16, 1994.

[42] Ruby B. Lee and Jerome C. Huck. 64-bit and multimedia extensions in the PA-RISC 2.0 architecture. In COMPCON, pages 152–160, 1996.

[43] Chris Lomont. Introduction to intel advanced vector extensions. Intel Software Network, 2011.

[44] Randy Meyers. The new C: Introducing C99. C/C++ Users Journal, 18(10), October 2000.

[45] Millind Mittal, Alex Peleg, and Uri Weiser. MMX technology architecture overview. Intel Technology Journal, (Q3):12, 1997.

[46] Motorola Corporation. AltiVec Technology Programming Environments Manual, November 1998.

[47] Motorola Corporation. AltiVec Technology Programming Interface Manual, June 1999.

[48] David A. Patterson and John L. Hennessy. Computer organization and design - the hardware / software interface (3. ed.). Morgan Kaufmann, 2007.

[49] Ivan Pryanishnikov, Andreas Krall, and R. Nigel Horspool. Compiler optimizations for processors with SIMD instructions. Softw, Pract. Exper, 37(1):93–113, 2007.

[50] Ariel Ortiz Ramirez. An overview of Intel's MMX technology. Linux Journal, May 1999.

[51] Venu Gopal Reddy. Neon technology introduction. ARM Corporation, 2008.

[52] Juan C. Rojas and Miriam Leeser. Multimedia Macros for Portable Optimized Programs. PhD thesis, Northeastern University, August 2003.

[53] Jaewook Shin. Compiler Optimizations for Architectures Supporting Superword-level Parallelism. PhD thesis, University of Southern California, Dept. of Computer Science, 2005.

[54] Sun Microsystems. VIS Instruction Set Users Manual, 1997.

[55] Sun Microsystems. The VIS instruction set, version 1.0. White paper, Sun Microsystems, Network Circle Santa Clara, CA 95054, USA, June 2002.

[56] Shreekant Thakkar and Tom Huff. The Internet Streaming SIMD Extensions. Intel Technology Journal, (Q2):8, May 1999.