Systematic Support of Parallel Bit Streams in LLVM

by

Meng Lin

B.Eng., University of Science and Technology of China, 2012

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2014

APPROVAL

Name: Meng Lin

Degree: Master of Science

Title of Thesis: Systematic Support of Parallel Bit Streams in LLVM

Examining Committee:

Dr. Nick Sumner
Chair

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

Dr. Thomas C. Shermer,
Professor, Computing Science,
Supervisor

Dr. Fred Popowich,
Professor, Computing Science,
Supervisor

Dr. Arthur (Ted) Kirkpatrick,
Associate Director & Professor, Computing Science,
External Examiner

Date Approved: November 25th, 2014

Partial Copyright Licence

Abstract

Parabix (parallel bit stream) technology uses the single-instruction multiple-data (SIMD) capabilities of commodity processors for high-performance text processing. LLVM is a widely-used compiler-infrastructure that includes support for SIMD operations on vectors. This thesis investigates the feasibility of modifying LLVM to incorporate all the SIMD processing requirements of Parabix both to increase the portability of applications and to create additional opportunities to optimize those operations in the context of code generation. Our modifications include redefining type legality and lowering for vectors of small elements as well as insertion of logic to recognize and properly handle Parabix-critical operations. Experiments on the X86/SSE2 architecture show a speedup over the unmodified LLVM of about 300 times for some micro-benchmarks and demonstrate Parabix application performance substantially better than with the unmodified LLVM. We also demonstrate performance scaling in switching from X86/SSE2 to X86/AVX2 without any change in source code.

Key words: Parabix, LLVM, SIMD Optimization, Application Portability.

Acknowledgments

It is a great honor and pleasure for me to have my Master study in Simon Fraser University. I would like to give a big thanks to Dr. Robert Cameron for all of his earnest instructions. I learnt a lot from him during these two years.

I would also like to thank Dr. Nick Sumner for his deep knowledge on LLVM and his great patience in assisting my thesis work. I would like to thank Dr. Thomas Shermer and Dr. Fred Popowich for early feedback on my thesis draft. I would like to give another big thanks to Ken Herdy, who gave me a lot of support in those late nights in the lab. Nigel Medforth helped me with the experiments on icgrep as well as the Parabix tool chain. Linda Lin helped me on Parabix tools too.

I would like to thank all the committee members for their precious time spent on my thesis. I would like to thank Simon Fraser University and Dr. Cameron again for providing me this awesome chance of studying computing science in Canada as well as providing financial support. I would like to thank my parents and all my friends in Canada that encouraged me and supported me during this journey.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Nowadays Single Instruction Multiple Data (SIMD) instructions are broadly built in for most commodity processors. Compared with the traditional Single Instruction Single Data (SISD) instructions, SIMD provides an intra-register form of parallel computing by performing the same operation on many elements at the same time [17]. SIMD instructions are widely used for multimedia processing, digital signal processing or other compute-intensive applications [23, 26].

The recent method of parallel bit streams (Parabix) accelerates text processing using SIMD instructions, in applications such as UTF-8 to UTF-16 transcoding [11, 10], XML parsing [25, 13] and regular expression matching [28]. For these applications, byte streams of the input text characters are first transposed into 8 bit streams, one for each bit value of the character byte, and then loaded into SIMD registers so that 128 or 256 consecutive code units can be processed at once [14]. SIMD bitwise logic, shift operations, bit scans and other bit-based operations form the foundation of this programming model.

Although Parabix applications achieve substantial acceleration compared to sequential (SISD) equivalents, the Parabix tool chain needs to handle SIMD programming carefully. It is challenging for the following two major reasons:

1. SIMD instructions vary greatly among different instruction-set architectures (ISAs) which makes it hard to write portable SIMD programs. Some operations in Intel SSE may not exist in PowerPC AltiVec and vice versa. For example, integer comparison intrinsic $pcmpgtq$ in SSE4 does not have correspondence in PowerPC AltiVec.

2. Even within one specific SIMD instruction set, most operations are only available for some pre-chosen data sizes. This is referred to as ”sparse” instruction set in [9] and they gave a good example: in Intel SSE4, to shift-left a vector was implemented, but to shift-right was not. The 32-bit and 16-bit shift operations were available, but 64-bit shift was not [9].

The current Parabix tool chain uses the Inductive Doubling Instruction Set Architecture (IDISA) as an idealized computing model to overcome these two difficulties. Based on this model a library with the same name has been developed. It works well, but still has two shortcomings:

1. IDISA requires different header files for different architectures. Although it has a uniform API for portability, each header depends heavily on target-specific implementation details to provide the best implementation of each operation.

2. The IDISA generator [18] chooses the best implementation within the scope of a single function. This may be not the best when considering the context of this function. For example, when performing addition, we may know all the high bits of each field in a SIMD register is zero, making simplification possible.

This motivates us to find a better back end and currently, the most promising back end framework is LLVM. LLVM promises to enable out-sourcing of low-level and target-specific aspects of code generation [29, 21]. Switching to an LLVM back end benefits Parabix tool chain in the following ways:

1. LLVM provides a target-independent intermediate representation (IR) with a vector-of-integer type system that matches IDISA requirements closely. Parabix operations can be expressed with IR and hence can be ported, in principle, to any platform that LLVM supports, including X86, ARM, PowerPC, MIPS, SPARC and many more.

2. LLVM provides inter-procedural whole program analysis and optimization [22]. By using a built-in type system in the low level representation, LLVM keeps more static information to the back end and helps optimize Parabix operation with meaningful context.

3. LLVM provides just-in-time compilation which allows runtime source generation. This is critical to some applications such as regular expression matching, which generates sequences of Parabix operations on the fly according to the input regular expression.

However, the native back end of LLVM has a number of gaps in its support for parallel bit streams. SIMD support for 2-bit and 4-bit field width are not available; 1-bit field width is supported slowly. Packing high bits on 16-bit field width which is one of the four key elements in the IDISA model and critical for transposition does not produce proper machine code on X86. In this thesis, we extend LLVM to systematically support parallel bit streams and achieve high-performance code generation on the X86 target. We make the following contributions:

- We port the critical Parabix operations to the LLVM back end thus bringing all the benefits of LLVM discussed above into the Parabix technology.

- We redefine type legality in LLVM and extend the LLVM type system with the inductive doubling principle so that vectors of small element types are properly supported.
- We insert logic in the LLVM back end to recognize and handle key Parabix operations. This allows efficient code generation while keeping the whole source code in target-independent IR.
- We add a dedicated LLVM intrinsic for the long stream addition and enable high-performance chained addition on the unbounded integer model which can be applied in broader applications.
- We evaluate the new LLVM back end with both micro benchmarks on single Parabix operation and application level profiles. We get the same performance on X86 platform as the well-tuned IDISA library.

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of Parabix and LLVM. Chapter 3 shows the overall design goal; examples of the IR implementation are presented and compared with the IDISA library. Algorithms for machine code generation are discussed in Chapter 4 and the implementation details are in Chapter 5. Chapter 6 evaluates our work with both per-instruction benchmarks and application level profiles. Chapter 7 gives the conclusion.

Chapter 2

Background

2.1 SIMD and SWAR

SIMD is a parallel computing concept that performs the same instruction on different data to exploit data parallelism. Most of today's commodity processors supports SIMD within a register (SWAR). In this model, SIMD operations are applied within general-purpose or special registers that may be considered to be partitioned into fields. Operations on each field are independent from each other. This means for example, carry bits generated by addition could not pass to the next field.

The other important feature of the SWAR model is that the partition is not physical but rather a logical view of the register, so that different views are available on the same register. For a 128-bit SIMD register, a valid partition can be sixteen 8-bit fields as well as four 32-bit fields. There is no penalty from switching the logical view.

Some popular SIMD instructions sets are listed here:

- Intel MultiMedia eXtension (MMX). It defines eight 64-bit registers known as MM0 to MM7 which are aliases of the existing IA-32 Floating-Point Unit (FPU) stack registers. MMX only provides integer operations for early graphical applications thus is not a general purpose instruction set for SIMD programming [18].

- Intel Streaming SIMD Extensions (SSE) series. SSE extends the MMX instructions set and it introduces eight new independent 128-bit SIMD registers known as XMM0 to XMM7. Its successor SSE2 adds a rich set of integer instructions to the 128-bit XMM registers which makes it a useful SIMD programming model. AMD added support for SSE2 in its AMD64 architecture soon after the Intel released SSE2, making SSE2 broadly available across the desktop computers. Intel then released SSE3, SSSE3, SSE4 and AMD released SSE4a as the following SSE generations.

- Intel Advanced Vector Extensions (AVX). AVX extends the size of SIMD registers from 128 bits to 256 bits. It introduces 16 new registers YMM0 to YMM15 but still supports the 128-bit SSE instructions. More importantly, AVX shifts the two-operand operations towards the non-destructive three-operand form. Three-operand operation preserves the content in operand registers and could reduce the potential movement of data between registers. AVX supports a number of floating point operations on 256-bit registers, but does not support many of the integer operations that exist in SSE. Its successor AVX2 fills this gap and ensures the transition from SSE to AVX instructions with the same programming model. AVX2 is available on the Intel Haswell architecture. Its successor AVX512 has been announced to support 512-bit SIMD registers.

- ARM NEON. ARM as a popular mobile platform introduces its own SIMD extension named NEON in their Cortex-A series processors. It has thirty-two 64-bit registers (D0 to D31) as well as sixteen 128-bit registers (Q0 to Q15). In fact, $D_{2\times i}$ and $D_{2\times i+1}$ are mapped to the same physical location of the register Q_i. Some operations like multiplication on the 64-bit D registers can return result in the 128-bit Q register [18]. NEON supports the field width of 8 bits, 16 bits, 32 bits and 64 bits integer operations as well as 32-bit floating point operations.

In this thesis, we use SSE2 as main ISA target with 128-bit registers because SSE2 is broadly available on both Intel and AMD CPUs. We use AVX2 as main ISA target with 256-bit SIMD registers because AVX lacks support of integer SIMD operations.

2.2 Parabix Technology

Parabix technology is a programming framework for high-performance text processing that can utilize both SIMD and multi-core parallel processing facilities. It is built on top of the parallel bit streams concept. Byte-oriented input stream is first transposed into 8 bit streams with each stream corresponds to one bit location in the byte stream. For encodings that requires more than one byte, more bit streams can be introduced: one bit of the input code unit for one bit stream. Figure 2.1 gives an example of the transposition, B_0 to B_7 are the bit streams of the ASCII encoded input data. Zero bits are marked as periods (.) for clarity.

After the transposition, the character class bit streams are generated using bitwise logic, e.g. `[a]`, `[z9]` and `[0-9]` in the figure. With SIMD operations on the 128-bit register, 128 input code unit can be classified at the same time. Parabix defines a set of primitives on the arbitrary length bit stream, called the *Pablo Language*, which is usually applied on the character class bit streams to generate a number of *Marker Streams*. Marker Streams mark meaningful locations such as where a tag starts and ends in the XML document, matching positions of a partial regular expression. A

input data	a453z--b3z--az--a12949z--ca22z7--
B_7	....
B_6	1...1..1.1..11..1.....1..11..1...
B_5	11111111111111111111111111111111
B_4	.1111...11...1...111111....1111..
B_3	....111..111.111...1.1111....1.11
B_2	.11..11...11..11....1..11.....111
B_1	...11..111...1....1...1..1.1111..
B_0	1.11.11.1.111.1111.1.1.1111...111
[a]	1..........1...1.........1......
[z9]	....1....1...1.....1.11......1...
[0-9]	.111....1........11111.....11.1..

Figure 2.1: Basis and Character Class Streams, taken from [28].

simple counting or scanning through the marker streams are usually the final step in the Parabix technology.

Some useful Pablo primitives are listed below [12]:

1. Bitwise logic: AND, OR, XOR and NOT on arbitrary length bit streams.

2. Advance: shift forward the whole bit stream for 1 bit. In a little endian system, shift forward is to shift left because the bytes that comes first in the input stream reside in the lower memory address.

3. ScanThru: $s(M, C)$ denotes the operation of scanning from the marker stream M as the initial positions through the spans of ones in the stream C. Figure 2.2 shows an example of it.

$$s(M, C) = (M + C) \wedge \neg C$$

One example of ScanThru is in XML well-formedness checking, to check if a <tag> is written in correct syntax, M marks all the start positions of tags e.g. the next position after the opening angle bracket (<) and C is the marker stream for all legal tag content. $s(M, C)$ should mark all the positions of the closing angle bracket (>) which close tags. Say M_0 denotes the character class of >, then if $s(M, C) \wedge \neg M_0$ is not all zero, some tag is not closed properly with the > symbol. Note that all the tags in the stream are checked simultaneously in parallel in the unbounded bit streams model.

4. MatchStar: $m(M, C)$ returns all positions that can be reached by scanning from the initial positions marked in M along the spans of ones in the stream C for zero or more steps. MatchStar gets its name from the star operator (*) in the regular expression. It also has important application in long stream addition. Figure 2.3 shows an example of it.

input data	`----173942---654----1----49731----321--`
M_0	`....1........1......1......1...........`
$D =$ `[0-9]`	`....111111...111....1....11111....111..`
$M_0 + D$	`..........1.....1....1...11...1...111..`
$M_1 = (M_0 + D) \wedge \neg D$	`..........1.....1....1........1........`

Figure 2.2: ScanThru Using Bitstream Addition and Mask, taken from [12] and slightly modified. In the original figure, the digits are arranged in the natural order, which means the digits on the left have higher significance. To be consistent with other related figures, we reverse this arrangement in the modified figure.

input data	`a453z--b3z--az--a12949z--ca22z7--`
M_1	`.1..........1...1........1.....`
$C =$ `[0-9]`	`.111....1........11111.....11.1..`
$T_0 = M_1 \wedge C$	`.1..............1........1.....`
$T_1 = T_0 + C$	`....1...1............1......11..`
$T_2 = T_1 \oplus C$	`.1111...........111111....111...`
$M_2 = T_2 \vee M_1$	`.1111........1...111111....111...`

Figure 2.3: MatchStar primitive, where $M_2 =$ MatchStar(M_1, C), taken from [28].

The Pablo language is defined over unbounded bit streams which of course need to be translated into a block-at-a-time processing for real applications [28]. The Pablo compiler is used to support the translation, taking care of the carry bits across block boundaries with a carry queue. A block-at-a-time C++ code is generated as a result.

2.2.1 IDISA Library

To actually execute the C++ code, a runtime library is necessary. Cameron proposed the Inductive Doubling Instructions Set Architecture in [14]. It provides a simple, systematic model with uniform treatment of SIMD operations of all power-of-2 field widths. As he wrote, "inductive doubling refers to a general property of certain kinds of algorithm that systematically double the values of field widths or other data attributes with each iteration." [14]. There are four key elements of this architecture:

- A core set of binary functions on SIMD registers, for all field width equals to 2^k. To work with parallel bit streams, the operation ADD, SUB, SHL (shift left), SRL (logic shift right) and ROTL (rotate left) comprise the set.
- A set of *half-operand modifiers* that make possible the inductive processing of field width $2W$

in terms of combinations of field width W. These modifiers select either the lower half of the field or the higher half.

- Packing operations that compress two vectors of field width W into one vector of field width $W/2$. For example, collecting all the higher half bits of fields from two vectors into one.
- Merging operations that produce one vector of field width W with two vectors of field width $W/2$.

A C++ library is then developed after this model and it is called the IDISA library. To be clear, in the following sessions the abstract architecture is called the IDISA model to distinguish from the IDISA library. An interesting fact about the IDISA library is that it is actually generated automatically from a pool of strategies to avoid duplicated human work among different targets. When targeting a new platform, the natively supported SIMD instructions need to be mapped to proper operations in the IDISA model. This is sparse in the sense that many other operations defined in the model are still not available. The IDISA generator could fill the gaps with a pool of strategies which basically tells how to implement instruction C given instruction A and B are available. Multiple strategies for the same operation may exist and the generator chooses based on the least instruction count heuristic [18].

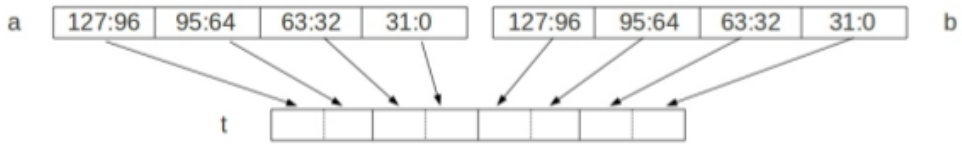

Figure 2.4: The logic of IDISA Horizontal Operations, cited from [18].

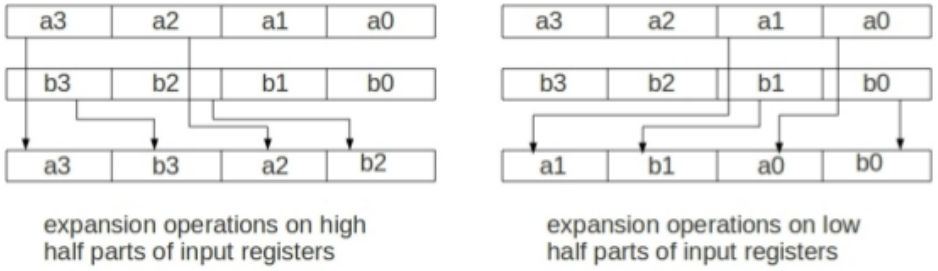

Figure 2.5: The logic of IDISA Expansion Operations, cited from [18].

The IDISA library divides SIMD operations into the following categories [2]:

- Vertical Operations (Template Class `simd<w>`): They are the most common SIMD operations between two registers where `w` is the field width. For example, `simd<8>::add(A, B)` aligns registers A and B vertically and adds up the aligned 8-bit fields. Different fields are independent of each other.
- Horizontal Operations (Template Class `hsimd<w>`): Operations like packing align the two operands horizontally, extract a portion of the bits in operands and concatenate into one full SIMD register. (Figure 2.4)
- Expansion Operations (Template Class `esimd<w>`): Operations that double the width of fields like merging which takes the higher 64 bits of the two operands (A and B), concatenates the first field from A and the first field from B to get a new field with the width doubled. Do the same to the following fields until a full SIMD register C is generated. (Figure 2.5)
- Field Movement Operations (Template Class `mvmd<w>`): Operations that copy and move the entire fields. The content of these fields should not change.
- Full Register Operations (Template Class `bitblock`): Operations that work with the contents of the whole SIMD registers. They are special vertical operations that has only one field.

The IDISA library claims to have better performance compared to the hand-written libraries. In the evaluation chapter, we will compare the performance of our LLVM back end with the IDISA library.

2.2.2 Critical Parabix Operations

There are at least five critical Parabix operations that can be the performance bottleneck and need special attention:

- Transposition. This is the first step of every Parabix application and it can be the primary overhead of some Parabix application. There are two major algorithms: the ideal three-stage implementation and the byte-pack implementation. The byte-pack implementation utilizes packing on 16-bit field width which is widely available on commodity processors. The ideal three-stage implementation uses the minimum number of packing instructions, but it requires packing on 2-bit, 4-bit and 8-bit field width which are not available on most of the commodity processors. Details of these two algorithms can be found in [14]. We implement both of the algorithm with the modified LLVM and compare their performance in Chapter 6.
- Inverse transposition. For some applications like the UTF-8 to UTF-16 transcoding, parallel bit streams need to be modified and translated back into byte streams, thus an inverse transposition is needed. As the inverse operation to the transposition, there are also two algorithms available which mirror the transposition algorithms. A detailed discussion can be found in [11].

- Long stream shift. Unbounded bit stream shift is translated into block-at-a-time shift. This operation shifts the whole SIMD register with the potential shift-in bits from the last block. We implement a peephole optimization in Chapter 5 that can improve the performance significantly.
- Long stream addition. The Pablo compiler deals with addition between unbounded bit streams using chained long stream additions, which adds two numbers as wide as the SIMD register with a carry-in bit and generates a carry-out bit. The naive approach chains 64-bit additions together to emulate 128-bit, 256-bit or 512-bit additions. Its time complexity grows linearly with the SIMD register size. A better algorithm is proposed in [28] which could add up to 4096 bits wide integers in constant time. We implement the constant-time algorithm in Chapter 4.
- Parallel deletion. This operation deletes bits from one bit stream at the positions identified by a deletion marker. Three different inductive doubling algorithms can be used for it. Refer to [11] for further detail.

These operations take most of the run time in the application level profile, so a slight improvement could benefit the application greatly. We study the LLVM support of the first four operations in this work and leave the parallel deletion to be future work.

2.3 LLVM Basics

The *Low Level Virtual Machine (LLVM)* is an open-source, well developed compiler tool that is dedicated to the compiler writers. It originates in 2000 with Lattner's Master thesis [21] and has been gaining popularity ever since. Today it is developed into a high-performance static compiler back end with just-in-time compilers and life-long program analysis and optimization, which means program analysis and optimization in compile time, link time and run time [29, 22]. It supports a variety of targets from Intel X86, PowerPC to the ARM mobile platform and hides the low-level target-specific issues for the compiler writers.

LLVM defines an intermediate representation (IR) as its virtual instruction set and IR is used as not only the input code to the LLVM tool chain but also the internal representation for analysis and optimization passes. This enables the programmer to use LLVM as a pipeline and to inspect output from each step. For example, with the C/C++ compiler of LLVM, Clang, source code in C/C++ is compiled into IR with different level of front end optimization. IR files are then linked together with link-time optimizations. The resulting IR is put through a good number of LLVM passes. LLVM optimizations are organized into passes. There are three types of passes [5]: analysis passes that collect information for other passes, transform passes that change the program and utility passes that provide general helper functionality for other passes. Each of the passes consumes LLVM IR

as input and generates IR output. The improvements they make to the file can be easily checked by running these passes alone.

Although the IR is low-level, it preserves high-level static information through the strong type system and static single assignment (SSA) form. SSA form guarantees only one assignment to every variable. SSA helps calculate the high-level data flow [15]. The main design goal of the IR is to be low-level enough so that most programming languages can target to it while maintaining the most high-level information to make aggressive back end optimization possible [29].

LLVM IR defines instructions and intrinsic functions. There are terminator instructions which produce control flow (return, branch, etc.), binary instructions (add, subtract, etc.), bitwise binary instructions (shift left/right, logic operations, etc.), memory instructions (load, store, etc.) and other instructions. Intrinsic functions are extension of IR instructions. Their names must all start with a ”llvm.” prefix [1]. Example intrinsic functions are standard C library intrinsics (memcpy, sqrt, sin, floor, etc.), bit manipulation intrinsics (population count, byte swap, etc.), debugger intrinsics, exception handling intrinsics and so on. They can be general operations for all the platforms as well as target-specific; e.g. `llvm.x86.sse2.psrli.w` corresponds to SSE2 native instruction $psrlw$. To achieve portability, we use none of these target-specific intrinsics in our work.

After optimization passes, the IR code is processed through the target-independent code generator and the machine code (MC) layer to become the native machine code. We describe the code generation process in detail in the next section as it is the major piece of logic we extend for parallel bit streams.

2.4 LLVM Target-Independent Code Generator

Although the name of the LLVM code generator contains “target-independent”, it is not. There are genetic strategies that can be applied across different targets, but target specific information is heavily required throughout the code generation process. We want to clarify this point before we start to describe the LLVM code generator.

The first stage for code generation is Instruction Selection, which translates LLVM code into the target-specific machine instructions. After that, there are machine level optimizations like live-interval analysis and register allocation. Instruction Selection is done by the following steps [4] (we describe each step in the following text):

- Initial SelectionDAG Construction: generate SelectionDAG from LLVM IR.
- DAG Combine 1
- Type Legalization Phase
- Post Type Legalization DAG Combine

- Operation Legalization Phase
- DAG Combine 2
- Instruction Select Phase
- Scheduling and Formation Phase

LLVM internally constructs a graph view of the input code called SelectionDAG where DAG is short for directed acyclic graph. Each node in the DAG represents an operation with an opcode, a number of operands and a number of return values. If the DAG node A uses the return value of another DAG node B, there will be an edge from B to A. The SelectionDAG enables a large variety of very-low-level optimization. And it also benefits the instruction scheduling process by recording the instruction dependency in the graph.

There are DAG combine passes after the initial construction and each legalization phase [4], discussed shortly. DAG combine passes clean up SelectionDAG with both general and machine-dependent strategies, making the work easier for initial constructor and legalizers: they can focus on generating accurate SelectionDAG, good and legal operations with no worries of the messy output.

Instruction Select Phase is the bulk of target-specific logic that translates a legal SelectionDAG into a new DAG of target code with pattern matching facility. For example, a node of floating point addition followed by a floating point multiplication could be merged into one FMADDS node on the target that supports floating point multiply-and-add (FMA) operations [4].

The Scheduling and Formation Phase assigns an order to each target instruction following the target's constraints. After that, a list of machine instructions are generated and the SelectionDAG is no longer needed.

Now we look at how LLVM deals with SIMD instructions. SIMD data are grouped into vectors and LLVM uses the notation `<N x iX>` to represent a vector of N elements, where each of the element is an integer of X bits [1, 9]. `<N x iX>` is also denoted as $vNiX$ as $vNiX$ is the internal type name used in the LLVM source code; e.g. `<4 x i32>` is the same with $v4i32$. Various operations can be applied on vectors and the LLVM back end knows how to lower them into proper machine instructions.

In LLVM IR, programmers can write any kind of vectors, even $v1024i3$. Those vectors may not be supported by the target machine. LLVM has the notion of "legal" vs. "illegal" types. Legality is a target-specific concept. A DAG node is legal if it only uses the supported operation on the supported types. Unsupported types are illegal types for the target. For example, $i1$ is not supported in X86, it is illegal together with all the operations that take $i1$ operands or return $i1$ values. Addition on $v16i8$ is legal for X86 SSE2 but multiplication on $v16i8$ is not since there is no native support of it. The type $v16i8$ is considered to be legal in this case. LLVM code generator has all the target details. It uses type legalization and operation legalization phases to turn illegal type or DAG into legal[4].

Type legalization phase has three ways to legalize vector types[9]: *Scalarization*, *Vector Widening* and *Vector Element Promotion*.

- **Scalarization** splits the vector into multiple scalars. It is often used for the transformation from $v1iX$ to iX.
- **Vector Widening** adds dummy elements to make the vector fit the right register size. It does not change the type of the elements, e.g. $v4i8$ to $v16i8$.
- **Vector Element Promotion** preserves the number of elements, but promotes the element type to a wider size, e.g. $v4i8$ to $v4i32$.

After type legalization, we may still have an illegal DAG node. This is because some operation on the legal type is not supported by the target. We need the operation legalization phase. There are three strategies in this phase:

- **Expansion**: Use another sequence of operations to emulate the operation. Expansion strategy is often general in the sense that it may use slow operations such as memory load and store, but it generates native code with correct outcomes.
- **Promotion**: Promote the operand type to a larger type that the operation supports.
- **Custom**: Write target-specific code to implement the legalization. This is similar to Expansion, but with a specific target in mind.

No illegal type should be introduced in the operation legalization phase which puts a limitation on the machine-independent legalize strategies: $i8$ is the minimum integer type on X86 and programmer needs to extend every integer less than 8 bits to $i8$ before returning it to the DAG. On the other hand DAG combine is different, you can choose the combine timing on your own. If you choose to combine before type legalization phase, you can freely introduce illegal types into your combined results.

The current legalization mechanism of LLVM is not sufficient to handle Parabix code efficiently. We propose new strategies and redefine legality in Chapter 4.

2.5 Summary

In this chapter we reviewed the Parabix technology which is a parallel text processing model and IDISA library which is a C++ library for SIMD programming. We also provided LLVM basics and described its target-independent code generator. The IDISA library has to maintain target-specific header files and is hard to further optimize outside the function scope. So we propose to replace the IDISA library with a LLVM back end in the next chapter.

Chapter 3

Design Objectives

In this chapter we discuss our overall goal for using LLVM as a new Parabix back end. First, we show that the IDISA library could be replaced by a pure target-independent IR library.

To start, let us look at one IDISA vertical operation: `simd<8>::add`. IDISA library implements this function with the compiler intrinsic that directly translates into the assembly code, so different header files have to be maintained for different instruction sets such as Program 3.1 and Program 3.2. However, with the LLVM IR, we can implement it as Program 3.3; no low level detail is specified here.

Most of the IDISA vertical operations can be expressed with a few lines of IR code. A bit more examples are listed here:

- Vector addition, subtraction, multiplication and shifting. There are IR instructions that correspond one-to-one with them.
- Integer comparison such as equality, greater than and unsigned less than. In IR, there is one instruction called 'icmp' which does the comparison. The only difference is that for vector type `<N x iX>`, the comparison result of 'icmp' is in type `<N x i1>` while IDISA requires it to be in type `<N x iX>` (All ones in an element means true and all zeros means false). We need to perform a sign extension by coping the sign bit of the $i1$ result until it reaches the size of iX (Program 3.4).

```
template <> bitblock128_t simd<8>::add(bitblock128_t arg1, bitblock128_t arg2)
{
  return _mm_add_epi8(arg1, arg2);
}
```

Program 3.1: Implementation of `simd<8>::add` for X86 SSE2

```
template <> bitblock128_t simd<8>::add(bitblock128_t arg1, bitblock128_t arg2)
{
  return (bitblock128_t)vaddq_u8((uint8x16_t)(arg1), (uint8x16_t)(arg2));
}
```

Program 3.2: Implementation of `simd<8>::add` for ARM NEON

```
define <16 x i8> @simd_add_8(<16 x i8> %arg1, <16 x i8> %arg2) {
entry:
  %r = add <16 x i8> %arg1, %arg2
  ret <16 x i8> %r
}
```

Program 3.3: Implementation of `simd<8>::add` with LLVM IR

```
define <16 x i8> @simd_eq_8(<16 x i8> %arg1, <16 x i8> %arg2) {
entry:
  %r1 = icmp eq <16 x i8> %arg1, %arg2
  %r2 = sext <16 x i1> %r1 to <16 x i8>
  ret <16 x i8> %r2
}
```

Program 3.4: Implementation of `simd<8>::eq` with LLVM IR. `Sext` is the instruction for sign extension.

```
define <16 x i8> @simd_max_8(<16 x i8> %a, <16 x i8> %b) {
entry:
  %m = icmp sgt <16 x i8> %a, %b
  %r = select <16 x i1> %m, <16 x i8> %a, <16 x i8> %b
  ret <16 x i8> %r
}
```

Program 3.5: Implementation of `simd<8>::max` with LLVM IR. `Select` selects elements according to the first operand: $\mathtt{r}_i = \begin{cases} \mathtt{a}_i & \text{if } \mathtt{m}_i = 1 \\ \mathtt{b}_i & \text{otherwise} \end{cases}$.

- Operations that have no IR correspondence such as `simd::min` and `simd::max`. They can be emulated with a sequence of IR, e.g. `simd<8>::max` in Program 3.5.

For horizontal operations, IDISA also needs to maintain target-specific logic. For example, to implement `hsimd<16>::packh`, it uses unsigned saturation $packuswb$ for X86 SSE2 and uses $vuzpq\_u8$ for NEON; for X86 SSE series after SSSE3, it uses the instruction $pshufb$. The author of the IDISA library needs to know these instruction sets very well. On the other hand, LLVM IR introduces a powerful instruction which can express most of the horizontal and expansion operations. It is the *shufflevector*.

```
<result> = shufflevector <n x <ty>> <v1>, <n x <ty>> <v2>, <m x i32> <mask>
; yields <m x <ty>>
```

The first two operands are vectors of the same type and their elements are numbered from left to right across the boundary. In other words, the element indices are $0 \ldots n-1$ for `v1` and $n \ldots 2n-1$ for `v2`. The `mask` is an array of constant integer indices, which indicates the elements we want to extract to form the `result`. Shufflevector is often used together with the *bitcast* operation. Bitcast converts between integer, vector and FP-values and changes the data type without moving or modifying the data, thus requiring the source and result type to have the same size in bits. With shufflevector and bitcast, we could write `hsimd<32>::packh` in Program 3.6. Figure 3.1 explains the indices used in the shuffle mask.

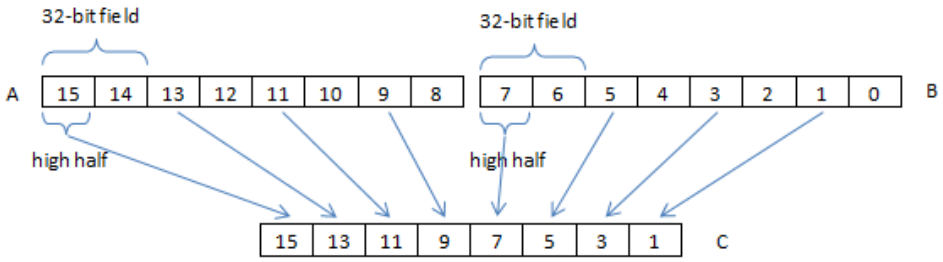

Figure 3.1: Shufflevector and `hsimd<32>::packh`. The vectors are bitcasted into $v8i16$ and the indices for the shuffle mask are drawn in the cell.

Program 3.6 can be easily generalized for packing high on any power-of-two field width. For other horizontal operations:

- Packing low: the same bitcast needs to be done but shufflevector with a different mask. For example, `hsimd<32>::packl` can be implemented with the mask `0, 2, 4, 6, 8, 10, 12, 14`.

```
define <8 x i16> @hsimd_packh_32(<4 x i32> %a, <4 x i32> %b) {
entry:
  %aa = bitcast <4 x i32> %a to <8 x i16>
  %bb = bitcast <4 x i32> %b to <8 x i16>
  %rr = shufflevector <8 x i16> %bb, <8 x i16> %aa, <8 x i32> <i32 1, i32 3,
        i32 5, i32 7, i32 9, i32 11, i32 13, i32 15>

  ret <8 x i16> %rr
}
```

Program 3.6: Shufflevector and `hsimd<32>::packh` in LLVM IR. Horizontal operations half the width of fields and that effect is reflected in the return value type.

- Packing sign mask: it packs together all the sign bits from each field of the operand. This can be implemented with the less than comparison. For example, `hsimd<32>::signmask(a)` is equivalent to `icmp slt <4 x i32> %a, <4 x i32> <i32 0, i32 0, i32 0, i32 0>` which returns a `<4 x i1>` sign mask vector.
- Other operations require coding a sequence of IR like `hsimd<32>::add_hl(a, b)`. They are less frequently used in the Parabix application.

Shufflevector and bitcast could also cover IDISA expansion operations. We list the IR code for `esimd<16>::mergeh` in Program 3.7 and explain the indices in Figure 3.2. This program divides the high-half of each SIMD register into 16-bit fields and merges them together. IR program is self-explanatory; programmers who understand shufflevector can understand its behaviour.

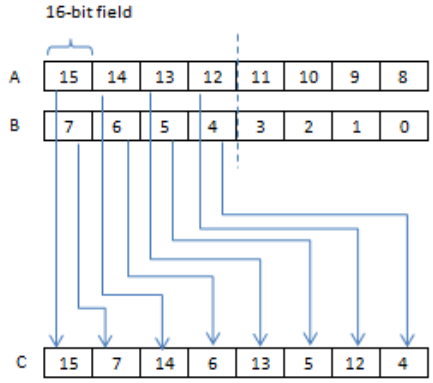

Figure 3.2: Shufflevector and `esimd<16>::mergeh`. The indices for the shuffle mask are drawn in the cell.

The rest of the expansion operations can be implemented as the following:

```
define <4 x i32> @esimd_mergeh_16(<8 x i16> %a, <8 x i16> %b) {
entry:
  %rr = shufflevector <8 x i16> %b, <8 x i16> %a, <8 x i32> <i32 4, i32 12,
        i32 5, i32 13, i32 6, i32 14, i32 7, i32 15>

  %rr1 = bitcast <8 x i16> %rr to <4 x i32>
  ret <4 x i32> %rr1
}
```

Program 3.7: Shufflevector and `esimd<16>::mergeh` in LLVM IR. Expansion operations double the width of fields.

- Merge low: similar to merge high with a different mask. For example, `esimd<16>::mergel` uses the mask `0, 8, 1, 9, 2, 10, 3, 11`.
- Unary operations like sign extension and zero extension: LLVM has built-in instructions with the same function.

For field movement operations:

- Field extract or insert: IR offers two vector instructions `insertelement` and `extractelement` for them.
- Constant fill: it fills each field with an integer constant. In IR this can be coded with vector constants such as `<4 x i32> <i32 1, i32 10, i32 30, i32 99>`.
- Unary and binary movement: those operations move fields within one register or among two registers and they can be implemented with shufflevector.

The full register operations could be coded in large-size integers like $i128$ and $i256$. You can add / multiply / shift it as a normal integer. In fact, all the integer instructions LLVM support can be applied to them thus enabling more complexed operations that IDISA does not support.

To sum up, we are able to replace the IDISA library with pure IR implementation. However, there is still one question to answer: since LLVM has its own C++ compiler called Clang and we could compile the C++ IDISA library into IR, what is the difference between the Clang-generated IR and our hand-written IR library? We show a tool chain diagram in Figure 3.3 which explains what is a Clang-generated IR and how the IR library is compiled against the Parabix applications. There are at least three major differences:

1. Clang could not remove all the target-dependency from the C++ source. Not every IR instruction is target-independent. For example, IDISA function `hsimd<8>::packh` compiles to Program 3.8 and all the functions that start with `@llvm.x86.sse2` are only available on ISAs that support X86 SSE2. This is inherent to the use of direct compiler intrinsic in IDISA.

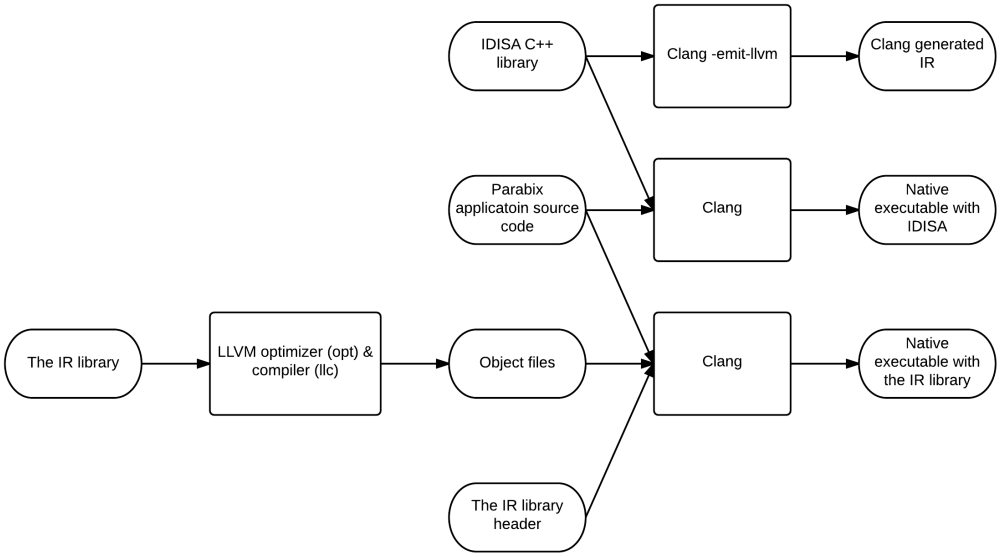

Figure 3.3: Tool chain diagrams for compiling and incorporating the IR library with Parabix. The IR library header is used to declare external functions.

```
define <2 x i64> @hsimd_packh_8(<2 x i64> %a, <2 x i64> %b) #4 {
entry:
  %0 = bitcast <2 x i64> %a to <8 x i16>
  %1 = tail call <8 x i16> @llvm.x86.sse2.psrli.w(<8 x i16> %0, i32 8) #1
  %2 = bitcast <2 x i64> %b to <8 x i16>
  %3 = tail call <8 x i16> @llvm.x86.sse2.psrli.w(<8 x i16> %2, i32 8) #1
  %4 = tail call <16 x i8> @llvm.x86.sse2.packuswb.128(<8 x i16> %3,
                                                       <8 x i16> %1) #1
  %5 = bitcast <16 x i8> %4 to <2 x i64>
  ret <2 x i64> %5
}
```

Program 3.8: Clang-generated IR for `hsimd<8>::packh` from compiling the IDISA function

2. Illegal operations are handled in different levels. Compare the following IR implementation for `simd<4>::add`:

```
add <32 x i4> %a, %b
```

   With the Clang-generated IR from the IDISA SSE2 file:

```
%and.i.i.i = and <2 x i64> %b, %m0
%0 = bitcast <2 x i64> %a to <16 x i8>
%1 = bitcast <2 x i64> %and.i.i.i to <16 x i8>
%add.i.i10.i = add <16 x i8> %0, %1
%2 = bitcast <16 x i8> %add.i.i10.i to <2 x i64>
%3 = bitcast <2 x i64> %b to <16 x i8>
%add.i.i.i = add <16 x i8> %0, %3
%4 = bitcast <16 x i8> %add.i.i.i to <2 x i64>
%and.i.i.i.i = and <2 x i64> %2, %m0
%and.i.i7.i.i = and <2 x i64> %4, %m1
%or.i.i.i.i = or <2 x i64> %and.i.i.i.i, %and.i.i7.i.i
ret <2 x i64> %or.i.i.i.i
```

   Given that addition on $v32i4$ is not supported by this target, the latter implements it with $v16i8$ addition and a few logic operations in the source code level. Target information is required. And even if the latter code is migrated to a target that supports $v32i4$ addition natively, it could not use that ability unless some optimization could recognize the intention behind these 12 lines.

   On the other hand, the former one-line code is not extended until the legalization phase. The target-specific details are thus left to the LLVM code generator.

3. Since the illegal operation is not extended until the legalization phase, more optimizations are available. The high-level intention of the IR instruction is better preserved. For `simd<4>::add`, if one of the operand is all zero, the IR optimizer could remove the single line of `add <32 x i4>` more easily than removing the 12 lines of code in the Clang-generated IR. It is useful for constant combination as well as other peephole optimizations because it simplifies the pattern recognition. We give an example of peephole optimization on the long integer shifting in Chapter 5.

This comparison explains our design goal: to replace the IDISA library with a high-level target-independent IR library. It is different from the IDISA library fundamentally in the way that it tries not to instruct the compiler how to implement this operation, but rather tell what to implement.

Could LLVM compile the IR library to efficient machine code? Experiments on X86 suggests no. Simple functions that directly correspond to native instructions like `simd_add_8` in Program 3.3 can be compiled correctly, but for some more complex instructions like shufflevector, where no target so far has native support for it, poor machine code might be generated (e.g. pack high for 16-bit fields). Furthermore, LLVM does not have good support for vectors of small elements, simple code like `add <128 x i1> %a, %b` would generate a large amount of memory operations and eight additions on SIMD registers. To achieve a better code generation, we bring many of the strategies from IDISA to the LLVM back end. The next two chapters describe this in detail.

Chapter 4

Vector of $i2^k$

Parabix operations work on full range of vector types. For the 128-bit SIMD register, Parabix supports $v128i1$, $v64i2$, $v32i4$, ..., $v1i128$, which we refer to as the vector of $i2^k$. Vector types $vXi8$, $vXi16$, ..., $vXi64$ are widely used for multimedia processing, digital signal processing and Parabix technology. They are well supported by the LLVM infrastructure, but the remaining vector types with smaller elements are not well-supported. For instance, $vXi1$ is a natural view of many processor operations, like AND, OR, XOR; they are bitwise operations. However, $v32i1$, $v64i1$ and $v128i1$ are all illegal on the current LLVM 3.5 back end for X86 architecture. The default type legalization process could not handle them efficiently.

For example, after seeing a $v128i1$ vector, the type legalization phase decides $i1$ is not natively supported, so it promotes the element type $i1$ to $i8$. Now it gets $v128i8$ which is 1024-bit in size and too large to fit in any register. The type legalizer then splits the vector into 8 $v16i8$ vectors to fit the 128 bits register size. If we think about doing bitwise-and on 2 $v128i1$ vectors, we find that LLVM produces 8 pairs of bitwise-and on $v16i8$. On the other hand, we can simply bitcast $v128i1$ to any legal 128-bit vector like $v4i32$, perform the bitwise-and and bitcast the result back to $v128i1$. The performance penalty of type legalization is high in this example. Another type legalization example of $v32i1$ can be found in Figure 4.1.

LLVM applies the same promote element strategy to vectors of $i2$ and $i4$, which leads to huge SelectionDAG generation and thus poorly performing machine code. On the other hand, $i1$, $i2$ and $i4$ vectors are important to Parabix performance-critical operations, such as transposition and deletion; Parabix applications, such as DNA sequence (ATCG pairs) matching which can be encoded into $i2$ vectors most efficiently, requires a better support of small element vectors. The inductive doubling instruction set architecture (IDISA) which is the ideal model for Parabix needs a core set of functions on the $i2^k$ vectors as the first key element. All these reasons motivate us to find better implementation of $i1$, $i2$ and $i4$ vectors.

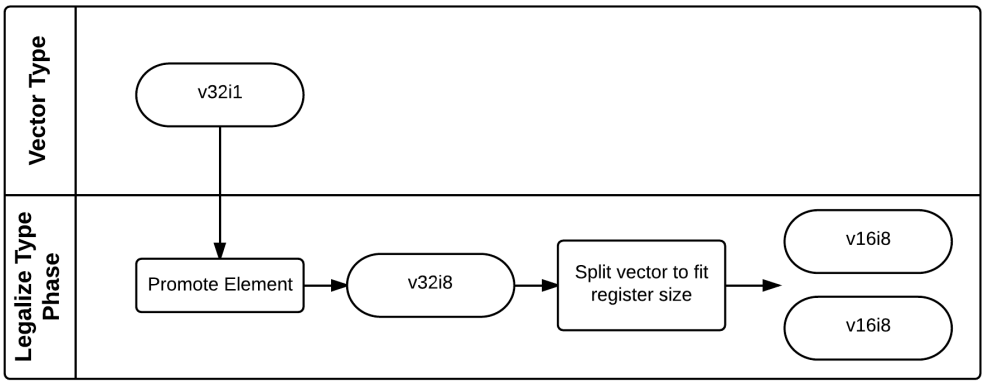

Figure 4.1: Type legalize process for $v32i1$ vector

4.1 Redefine Legality

From Chapter 2 we know that LLVM has three ways to legalize vector types: Scalarization, Vector Widening and Vector Element Promotion. None of these strategies could legalize small element vectors properly. Think about $v32i1$: it fits in the general 32-bit registers, and we can not benefit from extending or splitting the vector in wider or more registers, not to mention scalarizing it. It is best to store $v32i1$ vectors just in the general 32-bit register and properly handle the operations on them.

So we want to redefine the type legality inside LLVM. Instead of having direct hardware intrinsic on it, we define a vector type which has the same size in bits with one of the target's registers to be a legal vector type. The definition of the illegal operation remains the same. Under this definition, $v32i1$ is legal on any 32-bit platform, $v64i1$ is legal on any 64-bit platform and $v64i2$ is legal on any platform with 128-bit SIMD registers. However, as more types are legal, we need to handle more illegal operations that are not directly supported by the processor.

We implement every IR instruction that works on small $i2^k$ vectors except: (1) floating point instructions, because Parabix use integer operations exclusively, (2) division and remainder, because no Parabix operation require it, (3) shufflevector for vectors of small elements, because a general high-performance implementation of them is hard and unnecessary; instead we optimize some special cases that are used in the Parabix operations. To sum up, we implement the following instructions for the $i2^k$ vectors: (1) common binary functions listed in Table 4.1; (2) basic vector operations like INSERT VECTOR ELT, EXTRACT VECTOR ELT and BUILD VECTOR.

LLVM has the facility to "expand" an illegal operation, so that some operations we did not implement are still available. Consider $v32i1$, we did not lower its shufflevector but we can still write

Operation	Semantics
ADD	$c_i = a_i + b_i$
SUB	$c_i = a_i - b_i$
MUL	$c_i = a_i * b_i$
AND, OR, XOR	Common logic operations.
NE	Integer comparison between vectors. $c_i = 1$ if a_i is not equal to b_i.
EQ	$c_i = 1$ if a_i is equal to b_i
LT	$c_i = 1$ if $a_i < b_i$. a_i and b_i is viewed as signed integer
GT	$c_i = 1$ if $a_i > b_i$. a_i and b_i is viewed as signed integer
ULT	Same with LT, but numbers are viewed as unsigned integer
UGT	Same with GT, but numbers are viewed as unsigned integer
SHL	$c_i = a_i << b_i$. Element wise shift left
SRL	$c_i = a_i >> b_i$. Element wise logic shift right
SRA	$c_i = a_i >> b_i$. Element wise arithmetic shift right

Table 4.1: Supported operations and its semantics. A, B is the operands, C is the result. a_i, b_i, c_i is the i_{th} element.

shufflevector on $v32i1$ in IR. LLVM could expand it into a sequence of extracting and inserting vector elements. Unfortunately, the performance is not good. Efficient shufflevector support is possible with the new instructions PEXT and PDEP introduced in the Intel Haswell BMI2. According to [24], arbitrary n-bit permutation can be done using no more than $\lg n$ GRP instructions. It means for arbitrary $v32i1$ shufflevector, no more than 10 PEXTs, 5 ORs and 5 SHIFTs are needed.

4.2 In-place Lowering Strategy

With the redefined legality, we provide the fourth way to legalize vector type: In-place Lowering. It is called "in-place" because we do not rearrange the bit value of the vector data, we look at the same data with a different type. A trivial example is the logical operations on `<32 x i1>`; we can simply bitcast `<32 x i1>` to $i32$ and perform the same operation. Almost all the operations on $vXi1$ can be simulated with a few logic operations on iX (except the basic insert and extract vector operations) as listed in Table 4.2. Figure 4.2 shows the overall process of lowering $v32i1$ addition.

In-place Lowering allows us to copy the vector between registers or shift the vector within the register boundary. But it is different from vector element promotion. Refer to the Figure 4.4, vector element promotion requires shifting different element with different offsets.

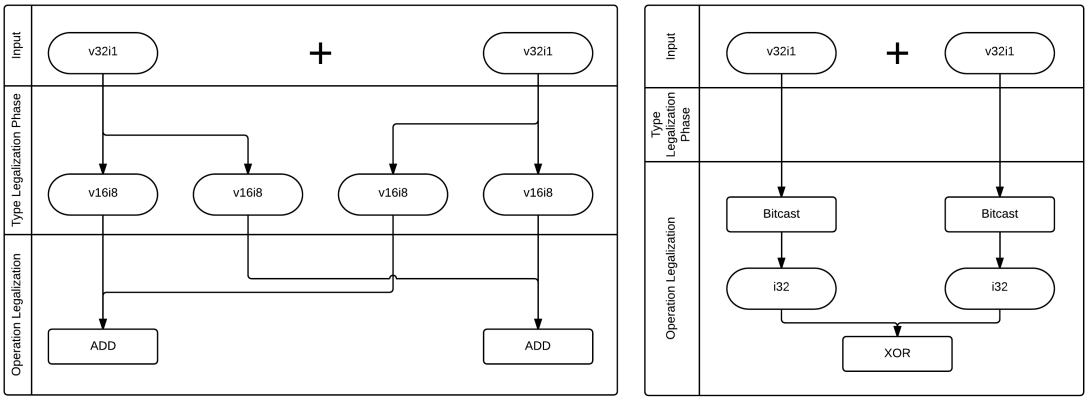

Figure 4.2: Comparison between LLVM default legalize process (left) and in-place lowering (right). The right marks $v32i1$ type legal and handles the operation ADD in the operation legalization phase. This keeps the data in the general registers without being promoted or expanded.

4.2.1 Lowering for $vXi2$

Vector type $vXi2$ has important role in the IDISA model and Parabix transposition and inverse transposition. Ideal Three-Stage Parallel Transposition [6] requires `hsimd<4>::packh` and `hsimd<4>::packl`, which can be implemented with shufflevectors on $v64i2$. Shufflevectors of $v64i2$ are also required by Ideal Inverse Transposition, for `esimd<2>::mergeh` and `esimd<2>::mergel`. Transposition is the first step of every parabix application[14]. It is also a significant overhead for some application like regular expression matching[28]. So good code generation for $vXi2$ is important.

Lowering $vXi2$ is harder than $vXi1$, so we propose a systematic framework using logic and 1-bit shifting operations. Consider A, B as two $i2$ integers, $A = a_0a_1$ and $B = b_0b_1$, we can construct a truth table for every operation $C = OP(A, B)$. We then calculate the first bit and the second bit of C separately with the logic combinations of a_0, a_1, b_0, b_1 and turn this into the *Circuit Minimization Problem*: find minimized Boolean functions for c_0 and c_1. We use the Quine-McCluskey algorithm[19] to solve it; an example can be found in Table 4.3.

Once we get the minimized Boolean functions, we can apply it onto the whole $vXi2$ vector. To do this, we need `IFH1(Mask, A, B)` which selects bits from vector `A` and `B` according to the `Mask`. If the i_{th} bit of `Mask` is 1, $\texttt{A}_i$ is selected, otherwise $\texttt{B}_i$ is selected. `IFH1(Mask, A, B)` simply equals $(\texttt{Mask} \wedge \texttt{A}) \vee (\neg\texttt{Mask} \wedge \texttt{B})$. With `IFH1`, if we have calculated the all high bits (c_0 for all the element) and low bits (c_1 for all the element), we can combine them with special `HiMask`, which equals to $101010\ldots10$, 128 bits long in binary. To calculate all the high bits of each $i2$ element, we bitcast A, B into full register type (e.g. $v32i1$ to $i32$, $v64i2$ to $i128$ or $v2i64$) and then do the following substitution on the minimized Boolean functions:

Operation on $vXi1$	iX equivalence
ADD(A, B)	XOR(A', B')
SUB(A, B)	XOR(A', B')
MUL(A, B)	AND(A', B')
AND(A, B)	AND(A', B')
OR(A, B)	OR(A', B')
XOR(A, B)	XOR(A', B')
NE(A, B)	XOR(A', B')
EQ(A, B)	NOT(XOR(A', B'))
LT(A, B), UGT(A, B)	AND(A', NOT(B'))
GT(A, B), ULT(A, B)	AND(B', NOT(A'))
SHL(A, B), SRL(A, B)	AND(A', NOT(B'))
SRA(A, B)	A'

Table 4.2: Legalize operations on $vXi1$ with iX equivalence. A, B are $vXi1$ vectors, A', B' are iX bitcasted from $vXi1$. For $v128i1$, we use $v2i64$ instead of $i128$ since LLVM supports the former better.

A	B	C
00	00	00
00	01	01
00	10	10
	...	
11	11	10

$$c_0 = (a_0 \oplus b_0) \oplus (a_1 \wedge b_1)$$
$$c_1 = a_1 \oplus b_1$$

Table 4.3: Truth-table of ADD on 2-bit integers and the minimized Boolean functions for C.

- For a_0 and b_0, replace it with A and B.
- For a_1 and b_1, replace it with $A << 1$ and $B << 1$.
- Keep all the logic operations.

So $c_0 = (a_0 \oplus b_0) \oplus (a_1 \wedge b_1)$ becomes $(A \oplus B) \oplus ((A << 1) \wedge (B << 1))$, which simplifies to $(A \oplus B) \oplus ((A \wedge B) << 1)$. We use shifting to move every a_1 and b_1 in place. For all the lower bits of each $i2$ element, the rules are similar:

- For a_1 and b_1, replace it with A and B.
- For a_0 and b_0, replace it with $A >> 1$ and $B >> 1$.
- Keep all the logic operations.

```
static SDValue GENLowerADD(SDValue Op, SelectionDAG &DAG) {
  MVT VT = Op.getSimpleValueType();
  MVT FullVT = getFullRegisterType(VT);
  SDNodeTreeBuilder b(Op, &DAG);

  if (VT == MVT::v64i2) {
    SDValue A = b.BITCAST(Op.getOperand(0), FullVT);
    SDValue B = b.BITCAST(Op.getOperand(1), FullVT);

    return b.IFH1(/* 10101010...10, totally 128 bits */
                  b.HiMask(128, 2),
                  /* C0 = (A0 ^ B0) ^ (A1 & B1) */
                  b.XOR(b.XOR(A, B), b.SHL<1>(b.AND(A, B))),
                  /* C1 = (A1 ^ B1)*/
                  b.XOR(A, B));
  }

  llvm_unreachable("GENLower of add is misused.");
  return SDValue();
}
```

Program 4.1: The function generated to lower ADD on $v64i2$.

Program 4.1 is the actual custom code to lower $v64i2$ addition. One thing to mention here is that we employ a template system to automatically generate custom lowering code and the corresponding testing code. We describe the template system later in Chapter 5.

4.2.2 Inductive Doubling Principle For $i4$ Vector

Now we have better code generation for $vXi1$ and $vXi2$, $vXi4$ vectors are our next optimization target. Shufflevectors of $vXi4$ are used in `hsimd<8>::packh`, `packl` and `esimd<4>::mergeh`, which are required by Ideal Three-Stage Transposition / Inverse Transposition; $vXi4$ is also a critical part of the IDISA model. But unfortunately, the strategies discussed above cannot be applied to $vXi4$ efficiently.

Circuit Minimization Problem is NP-hard[27, 20]. For $vXi4$, 4 Boolean functions of 8 variables are necessary: $c_i = f_i(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3), i \in \{0, 1, 2, 3\}$. It is known that most Boolean functions on n variables have circuit complexity at least $2^n/n$ [20] and we need 1-bit, 2-bit, 3-bit shifting on A, B. So the framework on $vXi2$ could not generate efficient code for us at this time. Instead, we introduce the *Inductive Doubling Principle* [14] and we show that this general principle can be applied for $vXi4$ and even wider vector element types, e.g. multiplication on $v16i8$, to get better performance.

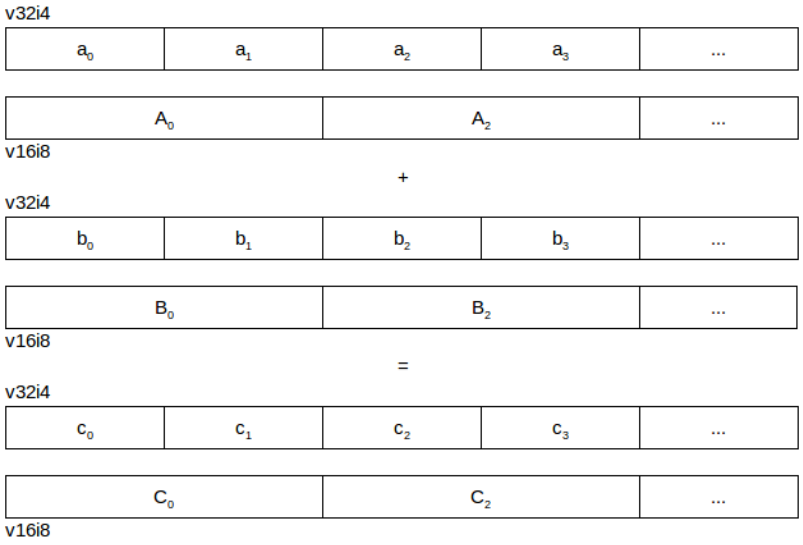

Figure 4.3: To add 2 $v32i4$ vectors, a and b, we bitcast them into $v16i8$ vectors. The lower 4 bits of $A_0 + B_0$ gives us c_1. We then mask out a_1 and b_1 (set them to zero), do add again, and the higher 4 bits of the sum is c_0.

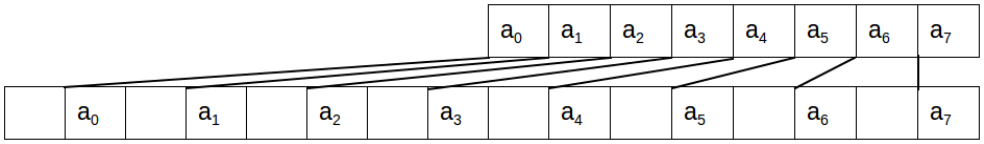

Figure 4.4: LLVM default type legalization of $v8i4$ to $v8i8$. a_0 to a_7 are $i4$ elements and they are shifted with different offsets during the element type promotion.

We use $v32i4$ as an example to illustrate the Inductive Doubling Principle. To legalize $v32i4$, LLVM promotes this type into $v32i8$, widens every element to $i8$ then shifts every element except the first one. Figure 4.4 shows an example of widening $v8i4$ into $v8i8$, we can see unnecessary movement of vector elements during widening. On a platform with a 128 bits SIMD register, $v32i8$ is further divided into two $v16i8$ and requires 2 registers to hold, while the original type $v32i4$ has 128 bits in size and should be able to reside in only 1 register. The Inductive Doubling Principle could achieve the latter for us. It bitcasts the vector in-place, views the same register as $v16i8$ type and emulates $i4$ operations with $i8$; e.g. in Figure 4.3, to get `add <32 x i4> %a, %b`, we calculate $c_0, c_2, \ldots, c_{30}$ (high 4 bits in each $i8$ element) and $c_1, c_3, \ldots, c_{31}$ (low 4 bits in each $i8$ element) separately with 2 $v16i8$ additions:

$$C = \texttt{IFH1}(HiMask_8, A \wedge HiMask_8 + B \wedge HiMask_8, A + B) \tag{4.1}$$

$$HiMask_8 = (1111000011110000 \ldots 11110000)_2 \tag{4.2}$$

So we can emulate SIMD operations on iX vectors with $iX/2$ or $i2X$ vector operations. We implemented all the operations on $vXi4$ with this principle and the algorithm is listed in Table 4.4. One thing that needs to be explained is SETCC, which is the internal representation of integer comparison in LLVM. It has a third operand to determine comparison type, such as SETEQ (equal), SETLT (signed less than), and SETUGE (unsigned greater or equal to). The third operand preserves in our algorithm.

$$C = \texttt{IFH1}(HiMask_8, HiBits, LowBits)$$

Operation $v32i4$	$HiBits$	$LowBits$
	All operation is on $v16i8$	
MUL	$MUL(A >> 4, B >> 4) << 4$	Default
SHL	$SHL(A \wedge HiMask_8, B >> 4)$	$SHL(A, B \wedge LowMask_8)$
SRL	$SRL(A, B >> 4)$	$SRL(A \wedge LowMask_8, B \wedge LowMask_8)$
SRA	$SRA(A, B >> 4)$	$SRA(A << 4, (B \wedge LowMask_8)) >> 4$
SETCC	Default	$SETCC(A << 4, B << 4)$
Default OP	$OP(A \wedge HiMask_8, B \wedge HiMask_8)$	$OP(A, B)$

In the table:

$A >> 4$: logic shift right every $i8$ element by 4 bits
$A << 4$: shift left of every $i8$ element by 4 bits
$HiMask_8 = (11110000 \ldots 11110000)_2$
$LowMask_8 = (00001111 \ldots 00001111)_2$

Table 4.4: Algorithm to lower $v32i4$ operations. The legalization input is $c = OP(a, b)$, where a, b, c are $v32i4$ vectors. A, B, C is the bitcasted results from a, b, c. They are all $v16i8$ type.

Furthermore, this method is applicable to vectors of wider element type. Multiplication on $v16i8$, for example, generates poor code on LLVM 3.4 (Program 4.2): the vectors are finally scalarized and 16 multiplications on $i8$ elements are generated. With in-place promotion, we bitcast the operands into $v8i16$ and generate 2 SIMD multiplications ($pmullw$) instead.

However, the algorithm in Table 4.4 cannot guarantee the best performance. Addition on $v32i4$ requires 2 $v16i8$ additions, but we can actually implement it with one. Looking back to Figure 4.3, we need to mask out a_1, b_1 and do add again, because $a_1 + b_1$ may produce a carry bit to the high 4 bits. If we mask out only the high bit of a_1 and b_1, still we do not produce a carry and we can calculate c_0 and c_1 together in one $v16i8$ addition. All we need to solve is how to put the high bit back. The following equations describe the 1-add algorithm:

```
define <16 x i8> @mult_8(<16 x i8> %a, <16 x i8> %b) {
entry:
  %c = mul <16 x i8> %a, %b
  ret <16 x i8> %c
}
```

```
# LLVM 3.4 default:
  pextrb $1, %xmm0, %eax
  pextrb $1, %xmm1, %ecx

  mulb   %cl
  movzbl %al, %ecx
  pextrb $0, %xmm0, %eax
  pextrb $0, %xmm1, %edx

  mulb   %dl
  movzbl %al, %eax
  movd   %eax, %xmm2
  pinsrb $1, %ecx, %xmm2
  pextrb $2, %xmm0, %eax
  pextrb $2, %xmm1, %ecx

  mulb   %cl
  movzbl %al, %eax
  pinsrb $2, %eax, %xmm2
  pextrb $3, %xmm0, %eax
  pextrb $3, %xmm1, %ecx

  mulb   %cl
  movzbl %al, %eax
  pinsrb $3, %eax, %xmm2
  pextrb $4, %xmm0, %eax
  pextrb $4, %xmm1, %ecx
  ...
  ...
  (16 mulb blocks in total)

            (a)
```

```
# Inductive doubling result:
  movdqa %xmm0, %xmm2
  pmullw %xmm1, %xmm2
  movdqa .LCPI0_0(%rip), %xmm3
  movdqa %xmm3, %xmm4
  pandn  %xmm2, %xmm4
  psrlw  $8, %xmm1
  psrlw  $8, %xmm0
  pmullw %xmm1, %xmm0
  psllw  $8, %xmm0
  pand   %xmm3, %xmm0
  por    %xmm4, %xmm0
  retq

            (b)
```

Program 4.2: LLVM 3.4 generates poor machine code for $v16i8$ multiplication. (a) is the generated code from LLVM 3.4. It $pextrb$ every $i8$ field and multiply them with $mulb$. The overall process in (a) is sequential. (b) is the generated code from the inductive doubling principle. In (b), two $pmullw$ which is the multiplication on $v8i16$ are used to improve performance.

$$m = (10001000 \dots 1000)_2 \tag{4.3}$$

$$A_h = m \wedge A \tag{4.4}$$

$$B_h = m \wedge B \tag{4.5}$$

$$z = (A \wedge \neg A_h) + (B \wedge \neg B_h) \tag{4.6}$$

$$r = z \oplus A_h \oplus B_h \tag{4.7}$$

Equation (4.6) uses only one $v16i8$ addition and equation (4.7) put the high bit back. Our vector legalization framework is flexible enough that we can choose to legalize $v32i4$ addition with the 1-add algorithm while keeping the remaining $v32i4$ operations under the general in-place promotion strategy. We discuss our framework implementation in Chapter 5.

4.3 LLVM Vector Operation of $i2^k$

In addition to the binary operations listed in Table 4.1, LLVM provides convenient vector operations like *insertelement*, *extractelement* and *shufflevector*, internally, they are DAG node INSERT VECTOR ELT, EXTRACT VECTOR ELT and VECTOR SHUFFLE. Another important internal node is BUILD VECTOR. In this section, we discuss how to custom lower these nodes on $i2^k$ vectors.

BUILD VECTOR takes an array of scalars as input and returns a vector with these scalars as elements. Take the $v64i2$ vector on the X86 SSE2 architecture for an example; ideally, the input provides an array of 64 $i2$ scalars and they are then assembled into a $v64i2$ vector. More specifically, since $i2$ is illegal on all X86 architectures, the legal input is actually 64 $i8$ scalars. The naive approach is to create an "empty" $v64i2$ vector, truncate every $i8$ into $i2$ and insert them into the proper location of the "empty" vector. We propose a better approach by rearranging the index.

Let us denote the input array as $a_0, a_1, \dots, a_{63}$, a_i are all $i8$. We rearrange them with $v16i8$ BUILD VECTOR according to Table 4.5 and build 4 $v16i8$ vectors V_1, V_2, V_3, V_4. The final build result is:

$$V = V_1 \vee (V_2 << 2) \vee (V_3 << 4) \vee (V_4 << 6) \tag{4.8}$$

a_{60}	$\dots$	a_{12}	a_8	a_4	a_0	V_1
a_{61}	$\dots$	a_{13}	a_9	a_5	a_1	V_2
a_{62}	$\dots$	a_{14}	a_{10}	a_6	a_2	V_3
a_{63}	$\dots$	a_{15}	a_{11}	a_7	a_3	V_4

Table 4.5: Rearranging index for BUILD VECTOR on $v64i2$

SIMD OR and SHL are used in this formula, thus improving the performance by parallel computing. Rearranging index approach can be easily generalized to fit BUILD VECTOR of $v128i1$ and

$v32i4$.

EXTRACT VECTOR ELT takes 2 operands, a vector V and an index i. It returns the i_{th} element of V. On the X86 architecture, there are built-in intrinsics to extract vector elements, such as $pextrb$ ($i8$), $pextrw$ ($i16$), $pextrd$ ($i32$) and $pextrq$ ($i64$); for smaller element types, we could extract the wider integer that contains it, shift the small element to the lowest bits and truncate. Following algorithm gives an example of extracting the i_{th} element from the $v64i2$ vector `V`.

- Bitcast `V` to $v4i32$ `V'` and extract the proper $i32$ `E`. Since every $i32$ contains 16 $i2$ elements, the index of `E` is $\lfloor i/16 \rfloor$.

$$\texttt{V'} = \texttt{bitcast <64 x i2> V to <4 x i32>}$$

$$\texttt{E} = \texttt{extract element V'},\lfloor i/16 \rfloor$$

- Shift right `E`, to put the element we want in the lowest bits.

$$\texttt{E'} = \texttt{E >>}(2 \times (i \bmod 16))$$

- Truncate the high bits of `E'` to get the result.

$$\texttt{R} = \texttt{truncate i32 E' to i2}$$

The choice of $v4i32$ does not make a difference, we can use any of the wider element vector types mentioned above. On the X86 architecture, the support of extraction on $v8i16$ starts at SSE2, while others start at SSE4.1, so we choose $v8i16$ extraction in our code to target a broader range of machines.

INSERT VECTOR ELT is similar, it takes 3 operands, a vector V, an index i and an element e. It inserts e into the i_{th} element of V and returns the new vector. Same as EXTRACT VECTOR ELT, X86 SSE2 supports $v8i16$ insertion ($pinsrw$), SSE4.1 supports $v16i8$ ($pinsrb$), $v4i32$ ($pinsrd$) and $v2i64$ ($pinsrq$); for smaller element types, we could extract the wider integer that contains the element, modify the integer and insert it back. Following algorithm gives an example of inserting `e` into the i_{th} element of the $v64i2$ vector `V`.

- Bitcast `V` to $v4i32$ `V'` and extract the proper $i32$ `E`.

$$\texttt{V'} = \texttt{bitcast <64 x i2> V to <4 x i32>}$$

$$\texttt{E} = \texttt{extract element V'},\lfloor i/16 \rfloor$$

- Truncate `e` and shift it to the correct position.

$$\texttt{e'} = \texttt{zero extend (e} \wedge (11)_2\texttt{) to i32}$$

$$\texttt{f} = \texttt{e' <<}(2 \times (i \bmod 16))$$

- Mask out old content in `E`, put in the new element.

$$\texttt{m} = (11)_2 << (2 \times (i \bmod 16))$$

$$\texttt{E'} = (\texttt{E} \wedge \neg \texttt{m}) \vee \texttt{f}$$

- Insert back `E'` to generate the new vector `R`.

$$\texttt{R} = \texttt{insert element V', E',} \lfloor i/16 \rfloor$$

We have discussed VECTOR SHUFFLE in Chapter 3. We did not develop a general lowering strategy for the small element VECTOR SHUFFLE. An efficient general lowering requires better machine instructions such as PEXT. Instead, we focused more on special cases that matter to Parabix critical operations, we optimized those cases to match performance of the hand-written library.

4.4 Long Stream Addition

Parabix technology has the concept of adding 2 unbounded streams, which must be translated into a block-at-a-time implementation[28]. One important operation is unsigned addition of 2 SIMD registers with carry-in and carry-out bit e.g. `add i128 %a, %b` or `add i256 %a, %b` with $i1$ carry-in bit `c_in` and generates $i1$ carry-out bit `c_out`. Cameron et al.[28] developed a general model using SIMD methods for efficient long-stream addition up to 4096 bits.

In this section, we replace the internal logic of wide integer addition ($i128$, $i256$ etc.) of LLVM with the Parabix long-stream addition. Following Cameron et al.[28], we assume the following SIMD operations legal on the target:

- `add <N x i64> X, Y`, where `N` $= RegisterSize/64$. SIMD addition on each corresponding element of the $i64$ vectors, no carry bits could cross the element boundary.
- `icmp eq <N x i64> X, -1`: compare each element of `X` with the all-one constant, returning an `<N x i1>` result.
- `signmask <N x i64> X`: collect all the sign bit of $i64$ elements into a compressed `<N x i1>` vector. From the LLVM speculation, this operation is equivalent to `icmp lt <N x i64> X, 0`, which is the signed less-than comparison of each $i64$ element with 0. In the real implementation we use target-specific operations for speed, e.g. $movmsk\_pd$ for SSE2 and $movmsk\_pd\_256$ for AVX.
- Normal bitwise logic operations on `<N x i1>` vectors. For small `N`, native support may not exist, so we bitcast `<N x i1>` to iN and then zero extend it to $i32$. This conversion could also help with the 1-bit shift we use later.

- `zext <N x i1> m to <N x i64>`: this corresponds to `simd<64>::spread(X)` in [28]. The internal logic of LLVM zero extension for $vNi1$ is not good enough, so we implement our own. An example of $v2i1$ is in Table 4.6.

Spread 2-bit value $(\texttt{ab})_2$ to $v2i64$	
M_0	ab
$M_1 = M_0 \times$ 0x8001	..............ab.............ab
$M_2 = M_1 \wedge$ 0x10001	..............a.............b
$M_3 =$ bitcast M_2 to $v2i16$	<2 x i16> <i16 a, i16 b>
$M_4 =$ zext M_3 to $v2i64$	<2 x i64> <i64 a, i64 b>

Table 4.6: Example of spreading $v2i1$ to $v2i64$

We then present the long stream addition of 2 $N \times 64$ bit values `X` and `Y` with these operations as follows:

1. Get the vector sums of `X` and `Y`.

$$\texttt{R} = \texttt{add <N x i64> X, Y}$$

2. Get sign masks of `X`, `Y` and `R`.

$$\texttt{x} = \texttt{signmask <N x i64> X}$$

$$\texttt{y} = \texttt{signmask <N x i64> Y}$$

$$\texttt{r} = \texttt{signmask <N x i64> R}$$

3. Compute the carry mask `c`, bubble mask `b` and the increment mask `i`.

$$\texttt{c} = (\texttt{x} \wedge \texttt{y}) \vee ((\texttt{x} \vee \texttt{y}) \wedge \neg \texttt{r})$$

$$\texttt{b} = \texttt{icmp eq <N x i64> R, -1}$$

$$\texttt{i} = \texttt{MatchStar(c*2+c\_in, b)}$$

MatchStar is a key Parabix operation which is developed for regular expression matching:

$$\texttt{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C) | M$$

4. Compute the final result `Z` and carry-out bit `c_out`.

$$\texttt{S} = \texttt{zext <N x i1> i to <N x i64>}$$

$$\texttt{Z} = \texttt{add <N x i64> R, S}$$

$$\texttt{c\_out} = \texttt{i >> N}$$

One note here for the mask type: `c` and `i` are literally all `<N x i1>` vectors, but we actually bitcast and zero extend them into $i32$. This is useful in the formula `c*2+c_in`, `MatchStart` and `i >> N`; in fact, after we shift left `c` by `c*2`, we already have an $N + 1$ bit integer which does not fit in `<N x i1>` vector. The same is true for `i`; so when we write `zext <N x i1> i to <N x i64>`, there is an implicit truncating to get the lower `N` bits of `i`, but when we shift right `i` by `i >> N`, we do not do such truncation.

LLVM internally implements long integer addition with a sequence of ADDC and ADDE, which is just chained 64-bit additions (or 32-bit additions on 32-bit target). We replace that with the long stream addition model thus improving the performance by parallel computing. As the hardware evolves, wider SIMD registers will be introduced, like the 512-bit register in Intel AVX512, our general implementation could easily adopt this change in hardware and add two $i512$ in constant time.

During our implementation, we found there was no intrinsic in IR for addition with carry-in and carry-out bit, there was only one intrinsic `uadd.with.overflow` for addition with carry-out bit. To realize unbounded stream addition, the ability to take the carry-in bit is necessary, otherwise we would end up with two `uadd.with.overflow` to include the carry-in bit. So we introduced a new intrinsic `uadd.with.overflow.carryin` and backed it with the long stream addition algorithm.

Chapter 5

Implementation

In this Chapter, we describe our realization of Parabix technology inside the LLVM facility. LLVM is a well-structured open source compiler tool chain which is under rapid development. So during our implementation, we tried our best to follow its design principles while keeping our code modularized and isolated to be able to easily integrate with new versions of LLVM. Our goal of code design is to:

1. Use general strategies across different types and operations to reduce repeated logic.
2. Minimize code injection in the existing source and put Parabix logic in a separate module.
3. Check correctness of every operation we implement. Since most of the test code follows the same pattern, they should be generated automatically to reduce repeated human work.

Most of our code reside in LLVM Target-Independent Code Generator[4]. From Chapter 4, we know that the current type legalization process of LLVM has a big performance penalty for small element vectors. So our approach marks $i1$, $i2$ and $i4$ vector legal type first, and then handles them in the operation legalization phase. For convenience, we name this set of vector types the *Parabix Vector*.

We walk through the following steps to mark a type legal on a certain target:

- **Add new register class in target description file.** LLVM uses TableGen (.td files) to describe target information which allows the use of domain-specific abstractions to reduce repetition [4]. Registers are grouped into register classes which further tie to a set of types. We introduced GR32X for 32-bit general register like EAX EBX for $v32i1$, GR64X for 64-bit general register like RAX RBX for $v64i1$, VR128PX for 128-bit vector register like XMM0 to XMM15 for $v128i1$, $v64i2$, $v32i4$ Types within the same register class can be bitcasted from one to the other, since they can actually reside in the same register.

- **Set calling convention.** They are two kinds of calling convention to set: return value and argument calling convention. For example, we instruct LLVM to assign $v64i2$ type return value to XMM0 to XMM3 registers, assign $v64i2$ argument type to XMM0 to XMM3 registers if SSE2 is available or to 16-byte stack slots otherwise.

Now the type legalization phase recognizes our $i1$, $i2$, $i4$ vectors as legal and passes them onto the operation legalization phase. We have two major methods to handle $i2^k$ vectors: *Custom Lowering* and *DAG Combining*.

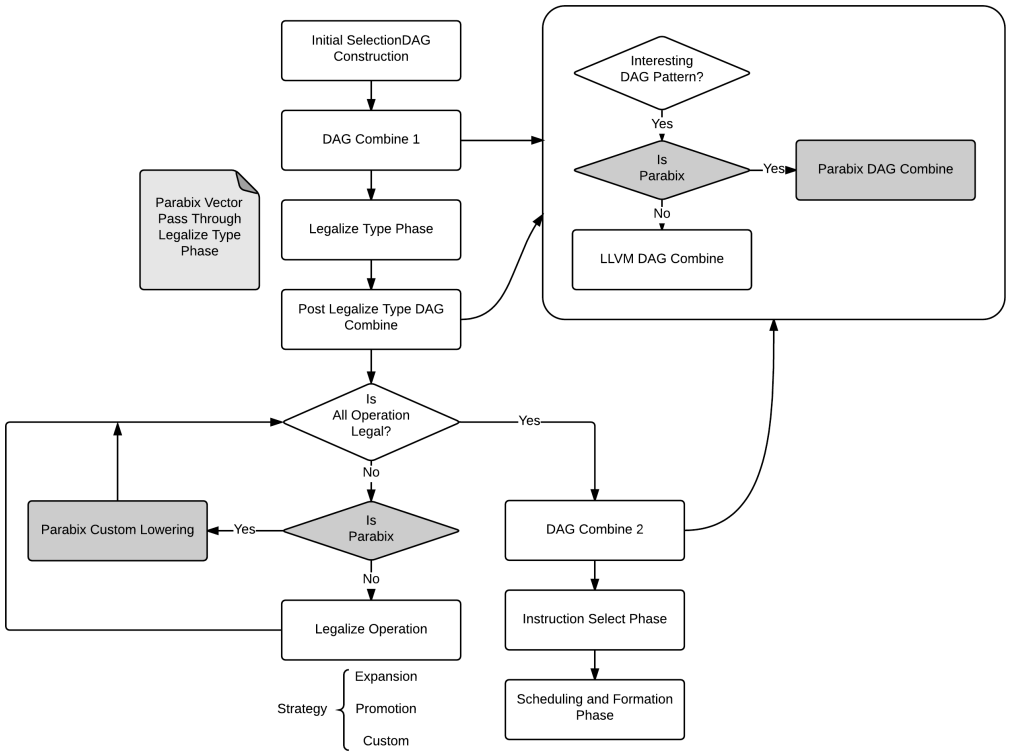

Figure 5.1: Overview of the modified instruction selection process. Logic for Parabix vectors are hooked into two places: the operation legalization phase and DAG Combine Phases. They are coloured with grey background.

Figure 5.1 gives an overview of our implementation. LLVM constructs a SelectionDAG with the input IR source code and then puts it through DAG Combine 1 for cleaning up and optimization. The optimized DAG is fed into the type legalization phase. We mark the Parabix Vectors type-legal so that they could pass through this phase without being changed. There is another DAG combining phase after the type legalization. Then, the resulting DAG is fed into the operation legalization phase which is done by iteration. Custom Lowering for Parabix resides in this phase. In each iteration,

the legalizer checks every DAG node for its legality. If the node is illegal and performs operation on some Parabix vector, the Parabix Custom Lowering code will be called to legalize this node; otherwise the default LLVM logic will handle this illegal node. The iteration terminates when every DAG node is legal. Through iteration, the legalizer could introduce new illegal nodes into the graph.

After the operation legalization phase, there is another DAG combining to clean up the messy output. The last two steps for machine code generation remain the same. There is Parabix DAG Combine logic in all the three DAG combining phases. Each DAG combining phase maintains a work list. It traverses the DAG graph for specific operations. If the operation works with Parabix vectors and it fits a certain pattern, the Parabix DAG Combine logic will replace the node with a new node or a new sub-tree of nodes. Examples of DAG patterns can be found in Section 5.1.2. Part(or all) of the new nodes can be appended to the work list so that the legalizer knows to combine it further later. There is no built-in iteration for DAG combining, but the iteration can be emulated by keeping appending the processed nodes to the work list if necessary.

In the following sections, we discuss some custom lowering strategies and how they are organized to fit our design goal. Then we give some examples of the Parabix DAG Combiner which are usually special cases for a certain operation. Finally we show how we use templates to generate code and test cases for the sake of DRY (don't repeat yourself).

5.1 Standard Method For Custom Lowering

5.1.1 Custom Lowering Strategies

After the type legalization phase, one shall not generate illegal types again. This means all the phases after type legalization are target-specific. But in practice, almost all the targets support $i8$, $i32$ and $i64$, so there are still general strategies we can apply across targets. For different types like $v32i1$ and $v128i1$, general strategies also exist to lower both of them. We define three legalize actions as the following:

1. Bitcast to full register and replace the operation code. This is useful for all $i1$ vectors, we need to specify the new operation code when defining the action, e.g. XOR for ADD on $v32i1$.

2. In-place promotion. Automatically apply $i2X$ vector operations on an iX vector following the Inductive Doubling Principle.

3. Custom. Same concept with LLVM Custom Lowering, manually replace an illegal DAG node with a sequence of new DAG nodes. They can be illegal nodes, but they cannot introduce illegal types. All $i2$ vectors are lowered here, also the 1-add version of the $v32i4$ addition.

5.1.2 DAG Combiner

DAG Combiner is the supplement to the custom lowering facility. It often focuses on special cases e.g. one operation and a subset of possible operands. We give a few examples of Parabix DAG Combiner here.

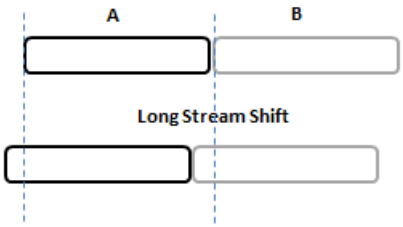

Figure 5.2: Long stream shifting

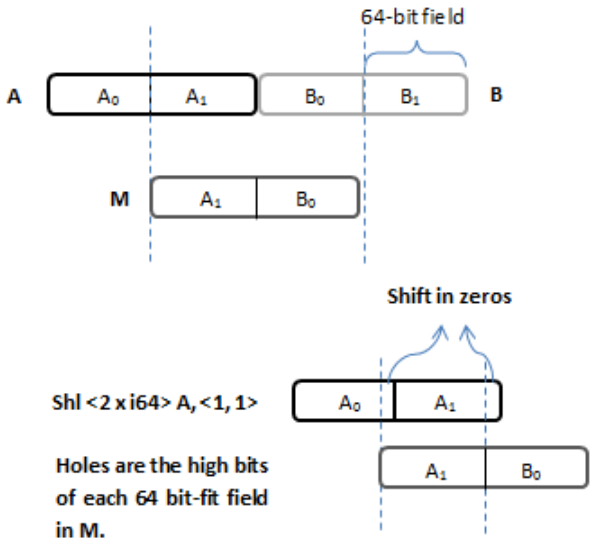

Figure 5.3: A better algorithm for long stream shift.

We first show a peephole optimization that greatly improves the performance of long stream shifting. Long stream shifting shifts left one whole SIMD register with potential shift-in bits from the other SIMD register. Refer to Figure 5.2, we want to shift left `A` by n bits and shift-in the highest n bits from B. For simplicity, we assume $n = 1$ and the SIMD register is 128-bits wide. The most straight-forward implementation is listed below:

```
A1 = shl i128 A, 1

B1 = lshr i128 B, 127
```

```
R = or i128 A1, B1
```

This algorithm is called double shift. It describes clearly what we want to implement so it is the preferred IR code in the library. Since shift on $i128$ is not natively supported, double shifts of $v2i64$ and one shufflevector are needed to implement the $i128$ shift. Thus, we need 4 $v2i64$ shifts, 2 shufflevectors and 3 logic or operations. These shufflevectors are further lowered into one or two instructions which are machine dependent.

A better algorithm that uses 2 $v2i64$ shifts, 1 shufflevector and 1 logic or is implemented. Refer to Figure 5.3, the algorithm is listed below:

```
M = shufflevector <2 x i64> A, B, <2 x i32> <i32 1, i32 2>

D = shl <2 x i64> A, <2 x i64> <i32 1, i32 1>

E = lshr <2 x i64> M, <2 x i64> <i32 63, i32 63>

R = or <2 x i64> D, E
```

`M` is the key to this algorithm. When we shift left `A` as $v2i64$, we create holes (1-bit shift-in zeros) in the lower end of each field. The correct fill-in data for these holes are just the highest bit in each field of `M`. So we shift right `M` as $v2i64$ to align the bits into correct positions and use logic or operation to fill the holes. This algorithm can be generalized to long stream shift of an arbitrary amount.

Once the back end recognizes the double-shift code, it re-implements with the latter algorithm. The DAG combining process is described in Figure 5.4.

The next example is shufflevector for `hsimd<16>::packh`. With the similar code described in Chapter 3, LLVM 3.5 does not generate the best assembly code. It generates a sequence of $pextrw$ and $pinsrw$. Even the newest LLVM trunk generates 27 lines of assembly. We create the following DAG Combiner and get the 5 lines of equivalent assembly code in Program 5.1.

- **Pattern**: shufflevector on $v16i8$ with mask = 1, 3, 5, . . . , 31. The target supports SSE2.
- **Combine Result**: one PACKUS node, which unsigned saturates two $v8i16$ into $v8i8$ vectors and concatenates them into one $v16i8$.

```
psrlw   $8, %xmm0
psrlw   $8, %xmm1
packuswb    %xmm0, %xmm1
movdqa  %xmm1, %xmm0
retq
```

Program 5.1: The optimized assembly code for `hsimd<16>::packh` on SSE2

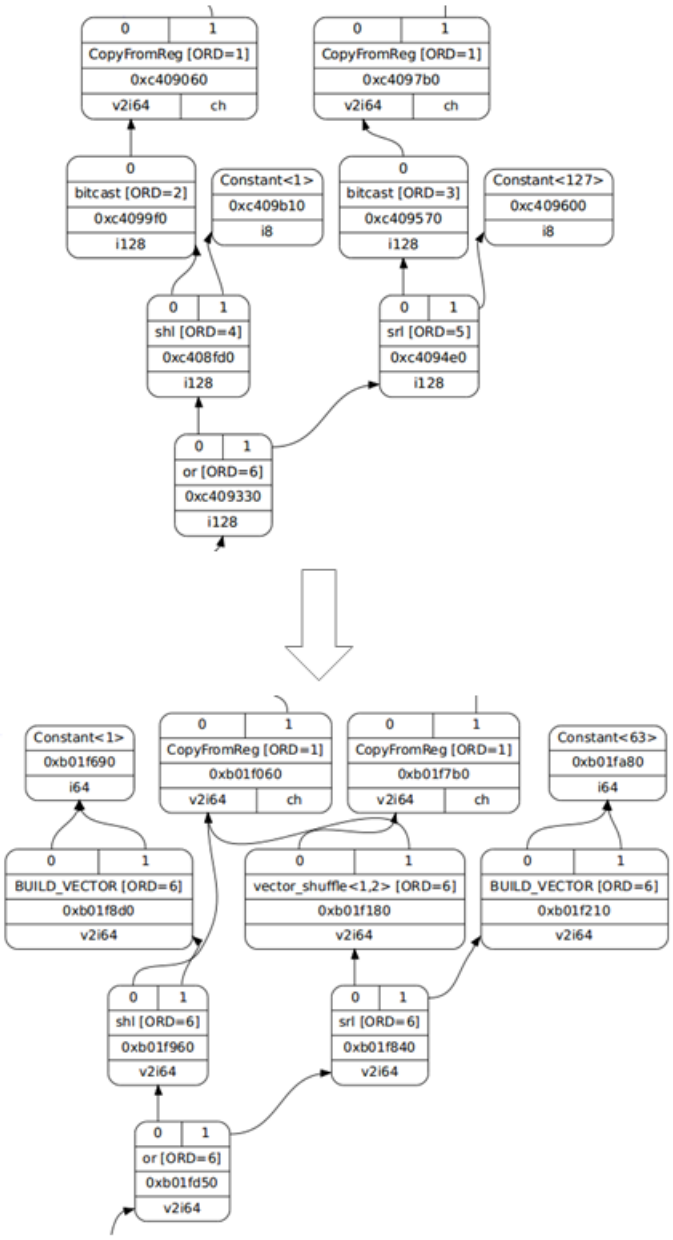

Figure 5.4: The DAG combiner for long stream shift. When the above pattern is recognized, it gets replaced with the better implementation below.

The efficient implementation of `packh` on vectors of small elements are possible with the PEXT instruction introduced by the Intel Haswell BMI2. PEXT is a useful instruction for bit manipulation on $i32$ and $i64$. Given the $i8$ variable A = $(\text{abcdefgh})_2$, Mask = $(10101010)_2$, PEXT(A, Mask) returns R = $(\text{aceg})_2$. PEXT extracts bits from A at the corresponding bit locations specified by the mask. With this in mind, we can implement `hsimd<2>::packh` as Program 5.2 (in pseudo IR for readability).

```
define <2 x i64> @packh_2(<2 x i64> A, <2 x i64> B) {
entry:
  ; extract lower 64 bits (A0) and higher 64 bits (A1)
  A0 = extractelement <2 x i64> A, i32 0
  A1 = extractelement <2 x i64> A, i32 1

  Mask = 0xAAAAAAAAAAAAAAAA ; 1010...1010 in binary
  P0 = PEXT(A0, Mask) | (PEXT(A1, Mask) << 32)

  ; same for B
  B0 = extractelement <2 x i64> B, i32 0
  B1 = extractelement <2 x i64> B, i32 1
  P1 = PEXT(B0, Mask) | (PEXT(B1, Mask) << 32)

  ret <2 x i64> <i64 P0, i64 P1>
}
```

Program 5.2: Implementation of `hsimd<2>::packh` with PEXT.

According to Program 5.2, we create the following DAG Combiner:

- **Pattern**: shufflevector on $v128i1$, $v64i2$ or $v32i4$ with mask = 0, 2, 4, ..., $NumElt \times 2 - 2$ or mask = 1, 3, 5, ..., $NumElt \times 2 - 1$. $NumElt$ is the number of elements for each type e.g. $NumElt = 32$ for $v32i4$.
- **Combine Result**: four PEXT nodes combined with OR and SHL.

To summarize, this kind of DAG Combiner provides a shortcut for the programmer to do peephole optimization. It can co-exist with a full custom lowering, like the relationship between immediate shifting and arbitrary shifting. Immediate shifting shifts all the vector elements by the same amount, allowing efficient realization for $v32i4$ with $v4i32$ shifts, while we apply the In-place Promotion strategy for $v32i4$ arbitrary shifting in the Parabix custom lowering.

DAG Combiner can optimize operations with illegal types in the phase DAG Combine 1, while for the custom lowering all the types must be legal. But we cannot simply put all the Parabix Custom Lowering logic inside the DAG Combiner. First, it is against LLVM design; DAG Combiner is designed for cleaning up, either the initial code or the messy code generated by the legalization passes [4]. Second, DAG Combiner cannot utilize the legalization iteration; in custom lowering,

general strategies may introduce new illegal operations which are hard to avoid since “illegal” is a target-specific concept. The DAG Combiner, on the other hand, 1) should not generate illegal operations in the phases after the operation legalization phase, 2) although it can also work in iteration, most of the lowering logic for common operations are not programmed in this module, we would end up with illegal non-Parabix operations.

5.2 Templated Implementation

During our implementation, we encountered a lot of duplicated code, especially in the test cases; Such duplication is against software design principles and is hard to maintain, sometimes even hard to write; a thorough test file for $i2^k$ vector contains more than two thousand lines of code, most of which are in the same pattern. To keep DRY and save programmer time, we introduced the Jinja template engine [3]. According to [7], Jinja belongs to the Engines Mixing Logic into Templates, it allows embedding logic or control-flow into template files. We use Jinja because:

- We can write all pieces of the content in one file, so it is easier to understand. Where in the Engines using Value Substitution, the driver code usually contains many tiny pieces of content. The reader must read the driver code as well as the template file to understand the output. One template example can be found in Program 5.5.
- Like the standard Model-View-Controller structure in the web design, our driver code needs only provide abstract data (like operation names in IR and the corresponding C++ library calls). How to present these data is not its responsibility. In the other words, we can have significant changes in the template without changing the driver.
- Jinja uses python and python is easy and quick to use.

5.2.1 Code Generation For $i2$ Vector

In Chapter 4, we legalized $i2$ vector operations with boolean functions. In our implementation, with the Quine-McCluskey solver, we got 11 sets of formulas which reside in one compact data script. We wrote template files to generate 11 C++ functions for them. This approach has the following benefits:

- It collects all the critical formula together so that possible future updates are easy to deploy.
- Implementation details only reside in the template file, so we are able to change the code structure easily. For now we create one function for each formula, but it is possible that we could create one big switch statement and generate one case for each formula instead. This can be done with only a few lines of change in the template.

Example formula of $v64i2$ addition as well as the function template is listed in the Program 5.3 and Program 5.4.

```
"add_2": r'''
    tmp = simd_xor(arg1, arg2)
    return simd_ifh1(simd_himask(fw),
                     simd_xor(tmp, simd_slli(1, simd_and(arg1, arg2))), tmp)
'''
```

Program 5.3: Minimum boolean function for $v64i2$ addition.

```
{% for name in FunctionNames %}
/* Generated function */
static SDValue GENLower{{ name.op }}(SDValue Op, SelectionDAG &DAG) {
  MVT VT = Op.getSimpleValueType();
  MVT FullVT = getFullRegisterType(VT);
  SDNodeTreeBuilder b(Op, &DAG);

  if (VT == MVT::v64i2) {
    SDValue A1 = b.BITCAST(Op.getOperand(0), FullVT);
    SDValue A2 = b.BITCAST(Op.getOperand(1), FullVT);

{% for line in Implement[name.c] %}
    {{ line | trim }};
{% endfor %}
  }

  llvm_unreachable("GENLower of {{ name.op }} is misused.");
  return SDValue();
}

{% endfor %}
```

Program 5.4: Custom lowering function template for $v64i2$. This template file generates one function for each operation.

5.2.2 Test Code And IR Library Generation

The IDISA library is mature and well tested. It is thoroughly tested against the IDISA+ Tester [18] and its correctness is also proved by many of the Parabix applications. So we test correctness of our modified LLVM by comparing results with the IDISA library. For example we extend LLVM to support $i1$ vectors, but does the extension work? We answer this question by first constructing a IR

library of all the functions defined on $i1$ such as the `add_1` listed in the top left corner in Figure 5.5. We then write a driver to generate random test data, put them through the IR functions as well as the corresponding IDISA functions, and check if the results are the same.

There is one problem in this approach. The driver and IDISA are both written in C++ but the IR library is in low-level representation. How to mix them together? We solve the problem by compiling the IR code into object files. A separate header file of IR-function signatures is maintained so that the driver code can call IR functions as external functions. The overview of the test system can be found in Figure 5.5. We use templates for both the IR library and the driver, some sample templates can be found in Program 5.5.

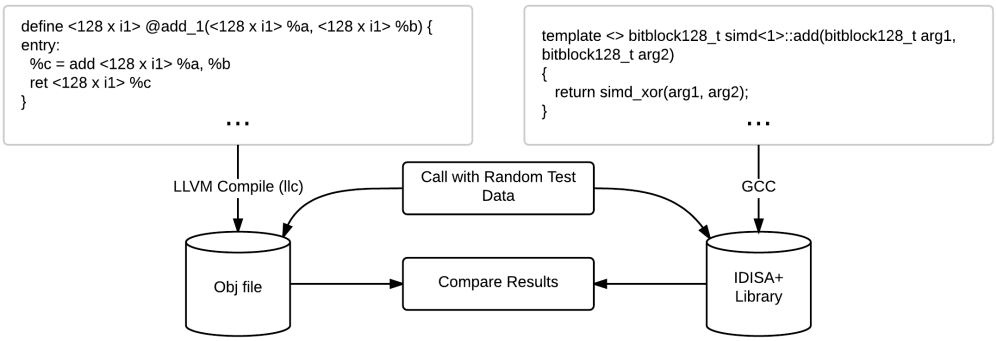

Figure 5.5: Test system overview. The pure IR library is first compiled into the native object file and then linked with the driver. The driver call functions from the both side to check correctness.

```
{% for name in FunctionNamesI4 %}
define <32 x i4> @{{name.c}}(<32 x i4> %a,
                             <32 x i4> %b)
{
entry:
  %c = {{ name.op }} <32 x i4> %a, %b
  {% if "icmp" in name.op %}
  %d = sext <32 x i1> %c to <32 x i4>
  ret <32 x i4> %d
  {% else %}
  ret <32 x i4> %c
  {% endif %}
}
{% endfor %}
```

```
define <32 x i4> @add_4(<32 x i4> %a,
                        <32 x i4> %b)
{
entry:
  %c = add <32 x i4> %a, %b
  ret <32 x i4> %c
}
```

```
define <32 x i4> @eq_4(<32 x i4> %a,
                       <32 x i4> %b)
{
entry:
  %c = icmp eq <32 x i4> %a, %b
  %d = sext <32 x i1> %c to <32 x i4>
  ret <32 x i4> %d
}
```

Program 5.5: Templates for the IR Library. On the top is the template, and two different output are listed below. We use embedded for loop and if statements.

Chapter 6

Performance Evaluation

In this chapter, we focus on the performance evaluation to assess whether our LLVM back end matches the performance of the hand-written library and whether back end optimizations provide performance advantages. We first validate our vector of $i2^k$ approaches, and then present the performance of some critical Parabix operations via an application-level profile.

6.1 Vector of $i2^k$ Performance

In Chapter 4, we presented different approaches to lower $i1$, $i2$, $i4$ and some $i8$ operations within one SIMD register. In this section, we validate our approaches by showing the improved run-time performance.

6.1.1 Methodology

Testing small pieces of critical code can be tricky, since the testing overhead can easily overwhelm the critical code and make the result meaningless. Agner Fog provides a test program which uses the Time Stamp Counter for clock cycles and Performance Monitor Counters for instruction count and other related events [8]. We measure the reciprocal throughput, the CPU cycles and the instructions count. The reciprocal throughput is measured with a sequence of same instructions where subsequent instructions are independent of the previous ones. In Fog's instruction table, he noted that a typical length of the sequence is 100 instructions of the same type and this sequence should be repeated in a loop if a larger number of instructions is desired.

The reciprocal throughput is an important attribute of instructions, but it is not directly related to the run time. So we also measure the CPU cycles and the instructions count. The less CPU cycles means less program run time. We found the CPU cycles and the instructions count behave

consistently, that less CPU cycles usually leads to less instructions count. The only difference is that the instructions count are more stable. We show the improvement of performance by showing improved CPU cycles or instructions count.

We did one simple experiment with SIMD XOR ($xorps$) to validate this program. In Figure 6.1, we show the measured performance of executing different number of XOR instructions; they are organized into one for loop. We have checked the assembly code to make sure the XOR operations are not optimized away.

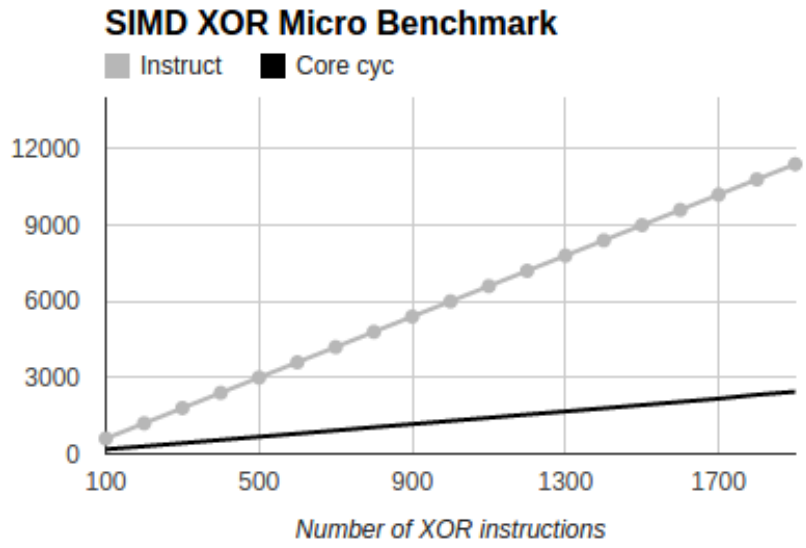

Figure 6.1: Test performance with XOR. The dotted line is instruction count and the other line is core CPU cycles.

From the figure, we can see the instruction count and CPU cycles grows linearly with the number of XOR instructions. So we can conclude that Fog's test program can be used to compare two pieces of critical code: the one with more measured CPU cycles is more complex and has more instructions. Note that from the figure, it seems the throughput of $xorps$ is 4, which is different from Intel's document (3.0 in document). We found this may be related to the compiler optimization on the loop; when we flattened the loop we got the throughput around 2.7. In order to eliminate this undesired effect, we flatten all the test code in the following sections.

In the following sections, we write micro benchmarks with Agner Fog's test program and compare performance between different implementation. Our test machine is X86 64-bit Ubuntu with Intel Haswell, and the detailed configuration can be found in Table 6.1. In order to inline pure IR functions (instead of a function call into one object file), we compile all the test code into LLVM bit code (binary form of LLVM IR) and then link / optimize them together. The default compile flag is to use Intel SSE2 instruction set on the 64-bit OS.

CPU Name	Intel(R) Core(TM) i5-4570 CPU
CPU MHz	3200
FPU	Yes
CPU(s) enabled	4 cores
L1 Cache	32 KB D + 32KB I
L2 Cache	256 KB
L3 Cache	6 MB
Memory	8GB

Table 6.1: Hardware Configuration

Operating System	Ubuntu (Linux X86-64)
Compiler	Clang 3.5-1ubuntu1, GCC 4.8.2
LLVM	LLVM 3.5
File System	Ext4

Table 6.2: Software Configuration

6.1.2 Performance Against IDISA

We compare our lowering on pure IR functions with the IDISA Library [18] which is written in C++. To test each operation, we generate a sequence of 500 such operations where none them has to wait for the previous one. 100 operations which are suggested by Agner seems too short for a stable result. This test sequence is generated by a template file.

For completeness, we choose all the operations listed in Table 4.1 except: NE (not equal), because IDISA does not support this operation; arbitrary shifts (SRL, SRA, SHL) because IDISA only supports immediate shifts; bitwise-logic (AND, OR, XOR), because the underline implementation are exactly the same and as simple as one line of machine code; we remove them to simplify the test code. Finally, we get eight operations for the micro-benchmark: ADD, SUB, MULT, EQ, LT, GT, ULT and UGT.

The performance comparison is listed in Figure 6.2 and Figure 6.3. From Figure 6.2, we can see for $i1$ and $i4$ vectors, the IR library has the similar CPU cycles with IDISA but it performs better with $i2$ vectors, especially on integer comparison.

The underlying logic for both libraries is the same, but it is implemented in different levels. For the IDISA library, `simd<2>::ugt` is inline-extended immediately by the compiler front end and its semantics of integer comparison lost ever after, while in the IR library, for the whole life cycle before the instruction selection, `ugt_2` keeps its semantics. The extension of `ugt_2` is delayed until the instruction selection phase, right before machine code generation. And the delayed extension may help the compiler optimize as we discussed in Chapter 3. We checked that the IDISA function `simd<2>::ugt` and IR function `ugt_2` (whose underlying code is just `icmp ugt <64 x i2> %a, %b`) generated different assembly code.

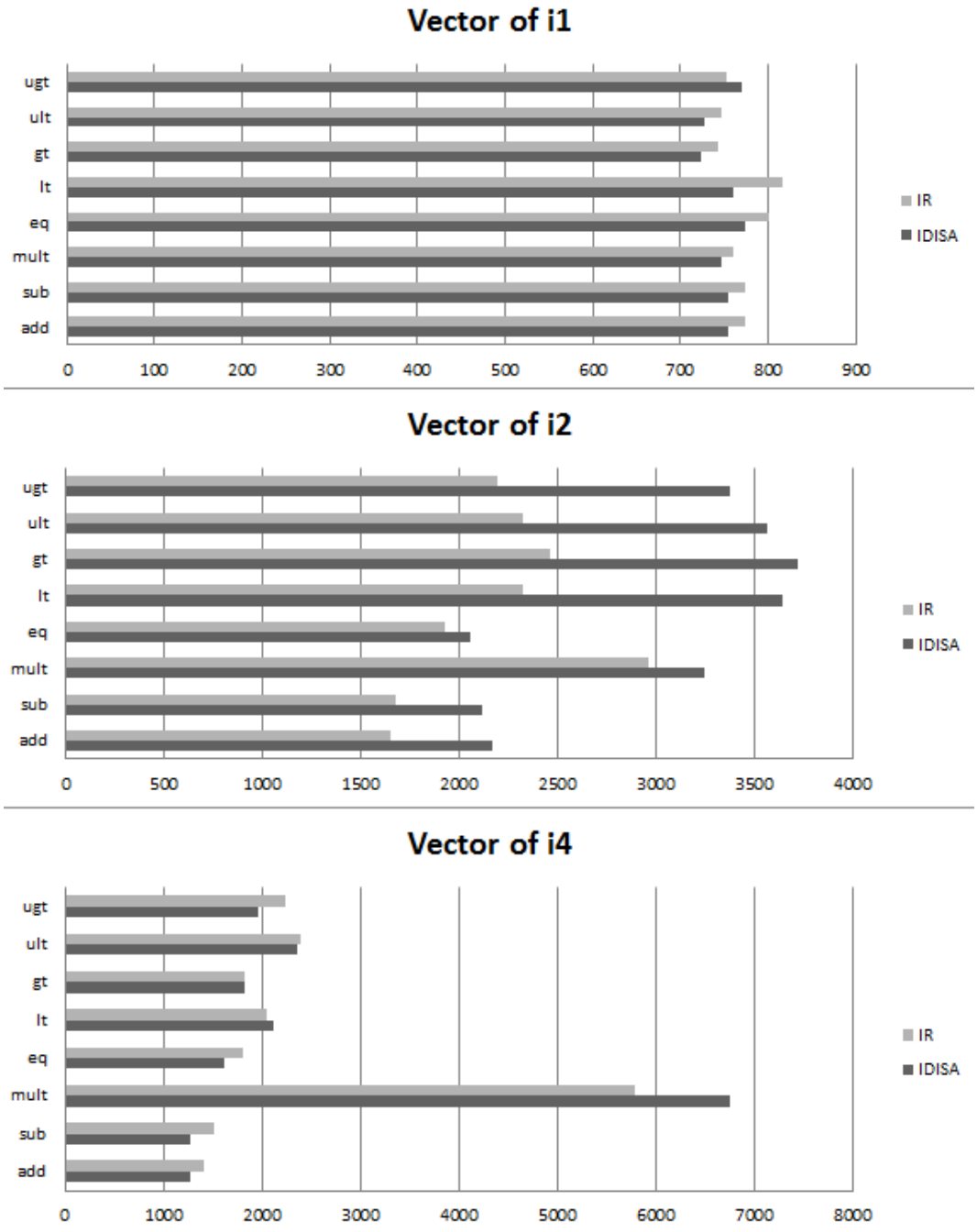

Figure 6.2: Total CPU cycles against IDISA library; for $i1$ and $i4$ vectors, IR library has the similar performance with IDISA but it performs better with $i2$ vectors, especially on integer comparison.

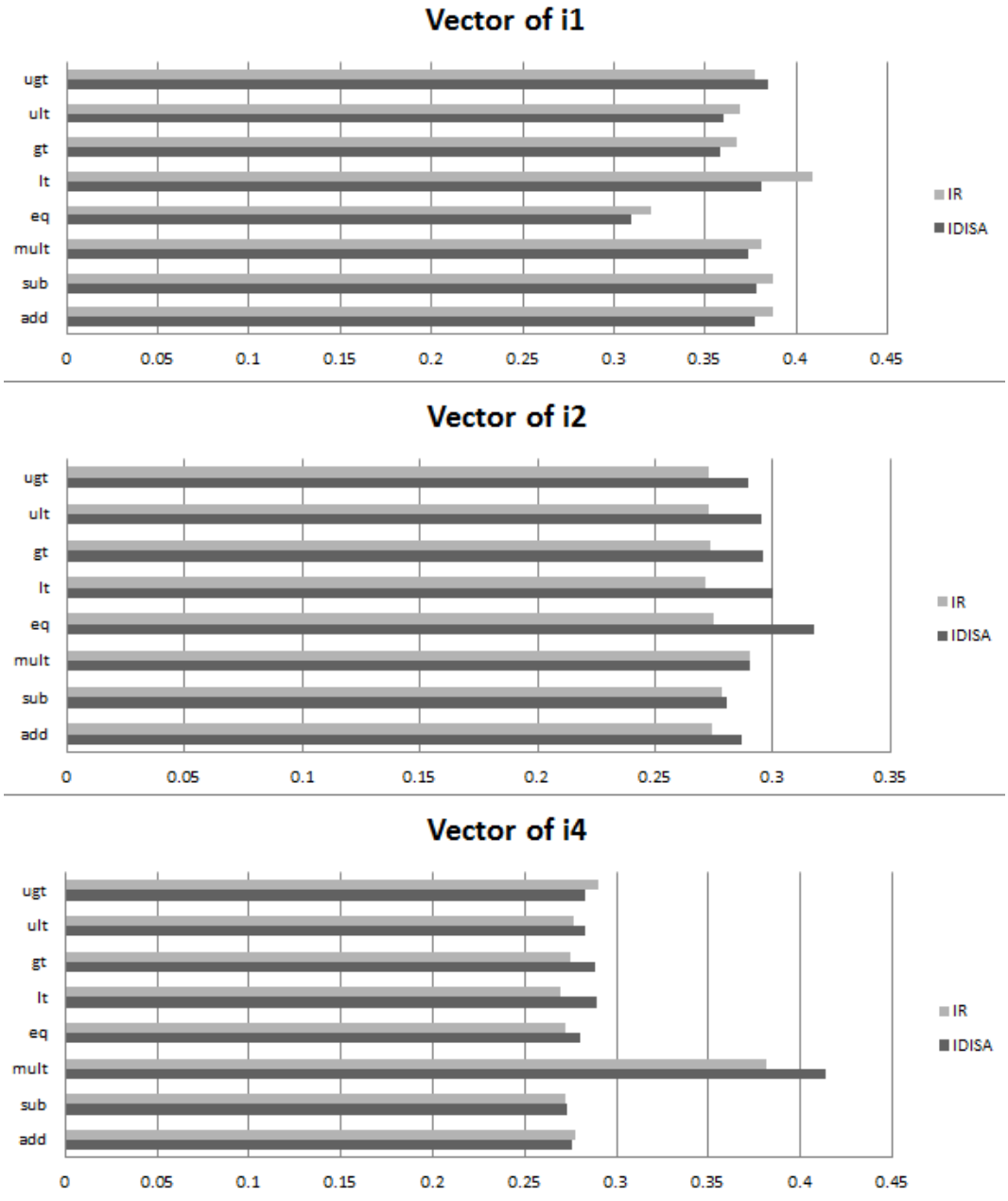

Figure 6.3: Reciprocal instruction throughput against IDISA library. IR and IDISA share almost identical throughput.

However, the delay in expansion is not always good. Take multiplication on the $i2$ vector for an example, we can see our IR library has slightly bettered total CPU cycles, but if we write our instructions sequence with a loop, IDISA library wins (Figure 6.4). Loop optimization is responsible for this difference; we did observe lines of assembly code hoisted outside the loop. Because all the operations tested here take two operands and the same constant value is used for all the second operand for simplicity, there is duplicated logic in each loop iteration that can be hoisted and shared. Hoisting was not done with the IR library. So early expansion in IDISA provides some optimization opportunity to the compiler.

From the reciprocal throughput comparison (Figure 6.3), the IR library loses on $i1$ vectors but wins most of the cases in $i2$ and $i4$; it relates to a better instruction selection. IDISA library is generated from a strategy pool based on the number of machine instructions which are treated equally with cost 1. But machine instructions actually have different throughput in the real hardware, and the LLVM back end has more knowledge of that, thus selecting better instructions.

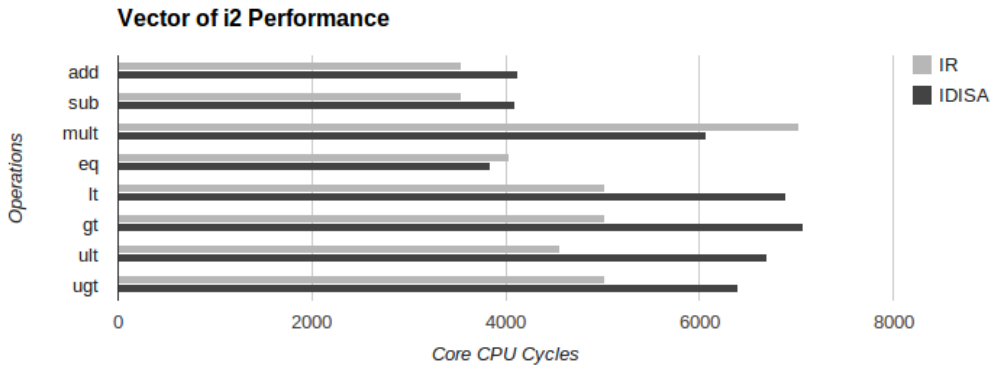

Figure 6.4: The same benchmark for $i2$ vectors with the instruction in a loop. Code in Figure 6.2 can be seen as the flattened version of this figure. We find IDISA here wins in the multiplication on $i2$, while IR wins it in Figure 6.2. Loop optimization should be responsible for it.

6.1.3 Performance Against LLVM

We compare our lowering with native LLVM. LLVM could not handle $i2$, $i4$ vectors but could handle $i1$ vectors slowly. Detailed performance data can be found in Table 6.3. We can see that our approach fills the gap of the LLVM type system.

	$i1$	$i2$	$i4$	$i8$
add	302	X	X	1
sub	310	X	X	1
mult	X	X	X	10
eq	273	X	X	1
lt	X	X	X	1
gt	X	X	X	1
ult	349	X	X	1
ugt	290	X	X	1

Table 6.3: Performance against LLVM native support of $i2^k$ vectors. 'X' means compile error or compile too slowly (longer than 30s), the remaining number means the ratio of CPU cycles speed up: add takes 302 times of cycles that our lowering needs. For $i8$, we apply the inductive doubling strategy on the multiplication, which explains the 10 times speed up.

6.2 Parabix Critical Operations

In this section, we evaluate our work by replacing Parabix critical operations with the IR library. We first choose transposition and inverse transposition as two representative operations and then measure performance in two Parabix applications: XML validator and UTF-8 to UTF-16 transcoder. Note that we did not rewrite the whole application with an IR library, part of the application is still IDISA but some critical operations are replaced. The default compile flag is to use the Intel SSE2 instruction set on a 64-bit OS.

To compare performance, we use the same data files used in [13]. The description of these files can be found in Table 6.4.

File Name	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
File Type	document	document	data	data	data
File Size (MB)	66	7	11	76	3
Markup Item Count	406k	74k	280k	4634k	18k
Attribute Count	18k	3k	160k	463k	30k
Avg. Attribute Size	8	8	6	5	9
Markup Density	0.07	0.13	0.57	0.76	0.87

Table 6.4: XML Document Characteristics. Taken from [13].

Table 6.5 shows the performance of the XML Validator. The only difference of xmlwf0 and xmlwf1 is their transposition code. The one in xmlwf1 is written in pure IR with the byte-pack algorithm (the source code can be found in Appendix A.1). We can see xmlwf0 and xmlwf1 share almost identical performance. LLVM 3.5 cannot handle packing on 16-bit field width very well so we custom lower the shufflevector and generate PACKUS instructions for X86 to get this good performance.

Another interesting observation is, when we re-compiled the same code on the Intel Haswell platform, we got almost no improvement for xmlwf0, since the IDISA library linked in is written with

	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
xmlwf0	3.93	4.36	4.55	4.89	5.18
xmlwf0 on Haswell	3.92	4.36	4.55	4.87	5.17
xmlwf1	3.92	4.37	4.56	4.86	5.18
xmlwf1 on Haswell	3.56	3.97	4.16	4.45	4.78

Table 6.5: Performance comparison of XML validator (xmlwf), in a thousand CPU cycles per thousand byte. In the table, xmlwf0 is implemented with full IDISA library and xmlwf1 is a copy of xmlwf0 with the transposition replaced.

	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
U8u16_0	281.46	37.11	40.06	244.94	10.20
U8u16_0 Haswell	272.68	34.21	39.84	242.56	10.11
U8u16_1	284.17	36.71	41.65	255.57	10.60
U8u16_1 Haswell	267.14	34.64	38.53	237.66	9.98

Table 6.6: Performance comparison of UTF-8 UTF-16 transcoder, in a million CPU cycles. U8u16_0 is written in IDISA, U8u16_1 has the transposition and inverse transposition part replaced.

direct SSE2 intrinsic so only SSE2 instructions can be generated. But we got a slightly better performance for xmlwf1, because the IR library is target-independent. LLVM back end knows AVX2 is available so it generates VEX prefixed operations with three-operand form.

Similar performance data on the UTF-8 to UTF-16 transcoder is listed in Table 6.6. U8u16_0 is written in IDISA and U8u16_1 has both the transposition and inverse transposition part replaced. Since our modified LLVM could not generate machine code for inverse transposition as good as IDISA, there are performance drops from U8u16_0 to U8u16_1. We also tried to compile them on the full Haswell, which gave us similar performance benefit from using AVX2 VEX operations. The feature of being target-independent helps Parabix to enjoy the improvement of hardware without changing its source code.

6.2.1 Ideal 3-Stage Transposition on the Intel Haswell

Intel Haswell architecture introduces the PEXT operation which can be used for the ideal 3-stage transposition (source code in Appendix A.2). We evaluated its performance in Table 6.7. The performance dropped with PEXT, but the major reason is that PEXT can only work on $i32$ or $i64$ integers for the current architecture. As the hardware evolves, we may have PEXT on SIMD registers directly. At that time, we can expect a better performance in xmlwf2, may be better than both xmlwf0 and xmlwf1 since 3-stage transposition is proved to be optimal under the IDISA model [14]. Our approach provides a new chance to exploit future hardware improvements.

	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
xmlwf0 on Haswell	3.92	4.36	4.55	4.87	5.17
xmlwf1 on Haswell	3.56	3.98	4.16	4.45	4.78
xmlwf2 on Haswell	4.11	4.49	4.69	4.97	5.30

Table 6.7: Performance of the ideal 3-stage transposition in a thousand CPU cycles per thousand byte. Xmlwf2 uses the ideal 3-stage transposition algorithm. Xmlwf1 uses byte-pack algorithm in IR, xmlwf0 uses the same algorithm in IDISA.

6.2.2 Long Stream Addition And Shift

We replaced the internal logic of big integer addition in Chapter 4 and introduced a new intrinsic: `uadd.with.overflow.carryin`. We evaluate them in this section by first comparing the long-stream addition algorithm with LLVM's original implementation and then some application level profiles for the new intrinsic.

We wrote micro benchmarks with Fog's test program. We put 200 independent additions on $i128$ and $i256$. We choose 200 because 200 is a small number that can give us stable performance results. It was tricky to make the test program right; we generated random data for the operands and carefully inserted the carry-out bit back to the return value so that LLVM knows to use the long-stream-addition logic. In order to be consistent throughout the comparison, we used the same compiler flag for all the runs (`-mavx2` for gcc and `-mattr=+avx2,+bmi2` for LLVM tool chain). The result is listed in Table 6.8.

	Core CPU Cycles	Instructions
LLVM on $i128$	1455	4199
Long stream addition on $i128$	2416	6552
LLVM on $i256$	4234	9798
Long stream addition on $i256$	2656	6959

Table 6.8: Micro benchmarks for long stream addition against LLVM's original implementation.

Long stream addition does not perform well on $i128$. Since there are only two sequential additions involved (1 $addq$ and 1 $adcq$), parallel computing does not save much but introduces new complexity. However, on $i256$ long stream addition has much better performance than the sequential one which generates 1 $addq$ and 3 $adcq$. As the width of the operand doubles, the CPU cycles from LLVM increases to the rate of 2.91, while in the long stream addition, the rate is only 1.10. This is because the time complexity of our algorithm is independent of the operand size while the sequential one has the time complexity linear to the operand size. Our algorithm scales better when the width of SIMD registers grows. We can confidently predict that on the Intel AVX512, long stream addition on $i512$ would out-perform the sequential one significantly.

An important Parabix application is regular expression matching. A grep-like tool was written recently with bitwise data parallelism called 'icgrep' [16]. Icgrep uses LLVM just-in-time compiling

facility to generate IR code on the fly according to the input regular expression. We use icgrep to evaluate our new intrinsic for long stream addition. Its signature is in Program 6.1 and it is used for the ”add with carry” logic in icgrep.

```
{i128, i1} @llvm.uadd.with.overflow.carryin.i128(i128 %a, i128 %b, i1 %carryin)
; return a pair of sum and carry-out bit
```

Program 6.1: Signature of `uadd.with.overflow.carryin`.

To compare with the unmodified LLVM, we need to emulate this new intrinsic. LLVM supports `uadd.with.overflow` which does not take the carry-in bit into account. The pseudo code for ”add with carry” is listed in Program 6.2. Then we can compare our back end with the unmodified LLVM. The same regular expressions and data files are used in [28]. Since we only care about the improvement between back ends, the details of the regular expressions are not important. We show the relative instructions count in Figure 6.5 and we get around 20 percent improvement. The experiment was done with 128-bit SIMD registers. Most of the improvement comes from the fact that our back end only requires one addition. We also measured the relative CPU cycles. There are around 10 percent reduction in CPU cycles with our new intrinsic, which implies that the sequential addition has higher throughput (instructions per cycle).

```
declare {i128, i1} @llvm.uadd.with.overflow.i128(i128 %a, i128 %b)
;return a pair of sum and carry-out bit

{i128, i1} @add_with_carry(i128 %a, i128 %b, i1 %carryin) {
entry:
  cin = zext %carryin to i128

  {s1, c1} = @llvm.uadd.with.overflow.i128(%a, %b)
  {sum, c2} = @llvm.uadd.with.overflow.i128(s1, cin)

  cout = or i1 c1, c2
  ret {sum, cout}
}
```

Program 6.2: Pseudo code for ”add with carry” logic in with unmodified LLVM.

We then compare the long stream shifting algorithms. We discussed two algorithms in Section 5.1.2; we implemented a DAG combiner in our back end. IR code of double shift can be compiled on the unmodified LLVM, so it is convenient that no new source code needs to be written. We used exactly the same code on both of the back ends. To avoid a difference in ”add with carry”, we used a logic that generates the same machine code on both back ends. The comparison of long

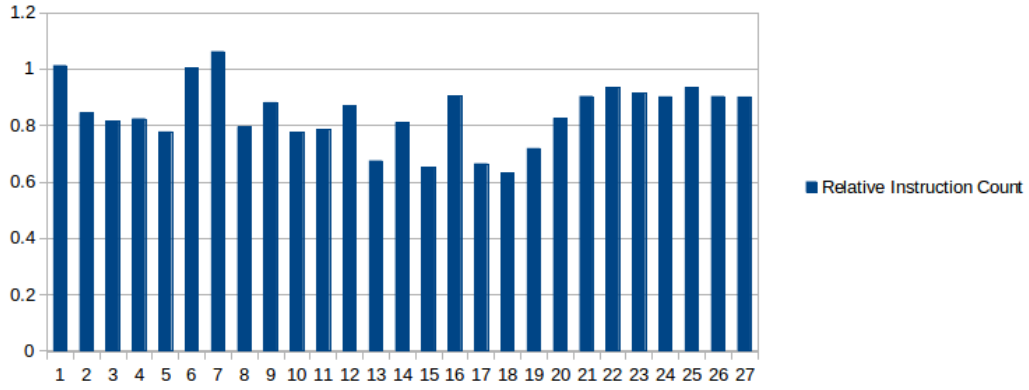

Figure 6.5: Improved instruction count with long stream addition. The number in the figure is the ratio of the instruction count in our back end to the count in unmodified LLVM. 27 pairs of regular expressions and data files are evaluated.

stream shifting with 128-bit SIMD registers is listed in Figure 6.6. We can see a reduction of 10 to 20 percent in instructions. For CPU cycles, a reduction of 20 percent is achieved. The reduction in CPU cycles equals to, if not greater than, the reduction in instructions count. It means the long stream shifting uses less instructions as well as maintaining a good throughput.

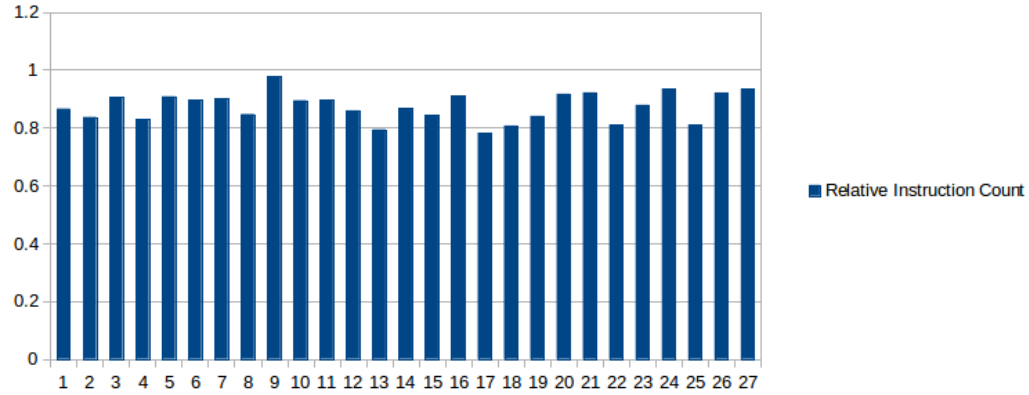

Figure 6.6: Improvement with long stream shifting in instruction count. The number in the figure is the ratio of the instruction count in our back end to the count in unmodified LLVM.

Next, we study the behaviour of long stream addition and shifting with 256-bit SIMD registers. We turned on both of the optimizations for icgrep. The relative instruction count is listed in Figure 6.7. From the micro-benchmark we already know that long stream addition works better than the sequential addition on this wider SIMD register. Together with long stream shifting, we achieved

a substantial 30 to 40 percent instruction reduction against the unmodified LLVM. We also achieved around 40 percent reduction in CPU cycles.

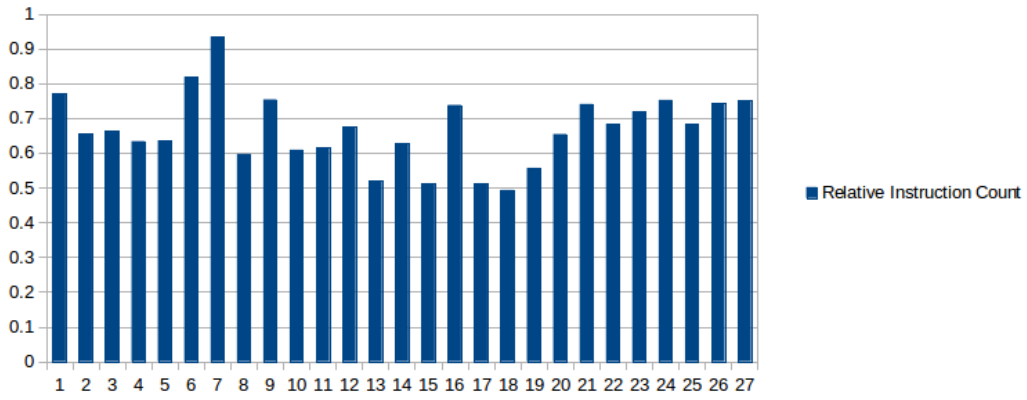

Figure 6.7: Improvement of icgrep on machines with 256-bit SIMD registers. Both long stream shifting and addition are used. The instruction count is compared with the unmodified LLVM with 256-bit SIMD registers.

Finally, we study the scalability of the modified LLVM. We want to check the improvement we can get by switching from 128-bit SIMD to 256-bit SIMD programming. We show the relative instructions count in Figure 6.8. Extended LLVM can always benefit around 30 percent from using the wider SIMD registers, but the original LLVM sometimes even has performance drops. Figure 6.9 shows the same comparison on a different metric: CPU cycles. Extended LLVM benefit around 20 percent from the wider SIMD registers. So we conclude that our LLVM back end has better scalability.

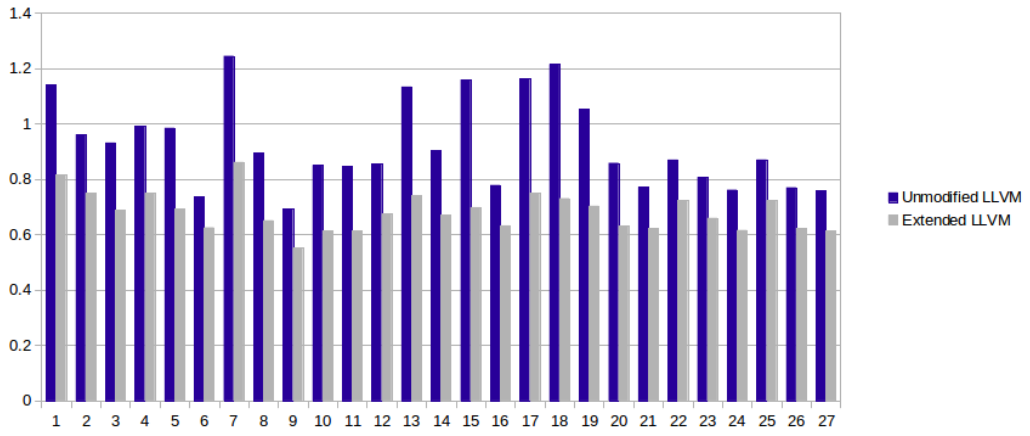

Figure 6.8: Improved scalability of icgrep. With the modified LLVM, icgrep gets more instructions count improvement by switching from 128-bit SIMD to 256-bit SIMD registers. Both long stream addition and shifting are used.

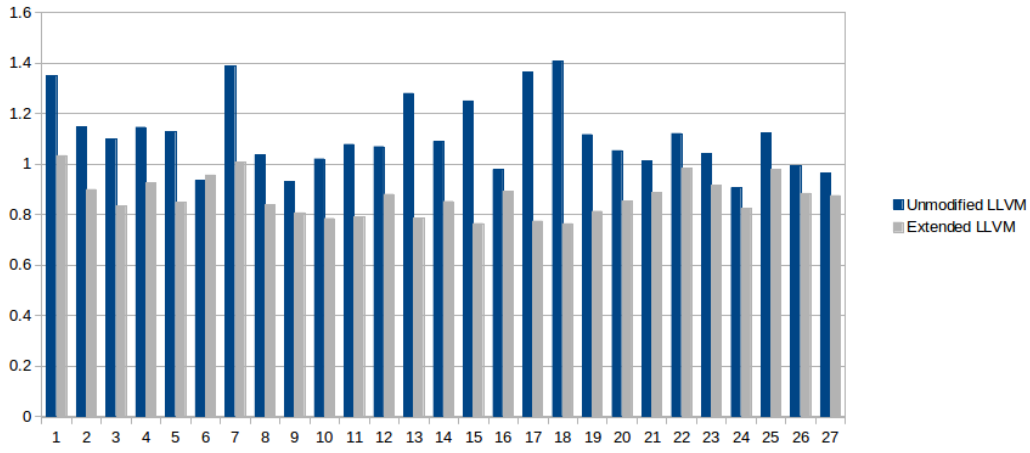

Figure 6.9: Improved scalability of icgrep in CPU cycles.

Chapter 7

Conclusion

In this thesis, LLVM as a new back end is introduced to the Parabix technology. A target-independent IR library of critical Parabix operations is also developed. LLVM brings in mature inter-procedure optimization, just-in-time compilers and outsources the machine-level code generation from the Parabix framework.

A systematic support for the vector of $i2^k$ is developed to extend the LLVM code generator with the IDISA model. A new LLVM intrinsic is added to enable chained additions on unbounded bit streams, which can be used for a broad category of applications. Long stream addition as well as shifting algorithms are built into the LLVM back end. In one specific target, Intel X86, efficient native code has been generated and the performance is as good as the well-tuned IDISA library. In some micro-benchmarks, it even achieves 300 times speed up over the unmodified LLVM. Performance improvement over different sub-targets (e.g. X86 SSE2 and AVX2) has been witnessed without any change in the IR library.

Although we tried hard to keep our extension to LLVM modularized and separated, our code is not able to merge with the newest LLVM trunk. One of the major reasons is that we redefined legality which involves small code changes in many places. We need to further track these changes in the future.

For more future work, new optimization passes for the Parabix can be developed. As one of the major reasons for its high performance, Parabix uses long sequence of bitwise logic and shift operations without any branch or loop statement. Specific optimization like a new register allocation algorithm may benefit this style of programming very much. More peephole optimizers may be added to LLVM to combine sequences of operations into compact SIMD intrinsics.

Parabix with LLVM has better chances to target different platforms efficiently such as the SPARC servers from Sun and the ARM mobile platform. Further extension of the LLVM code generator can be done in the future for these platforms.

Bibliography

[1] LLVM Language Reference Manual . http://llvm.org/docs/LangRef.html. 11, 12

[2] IDISA toolkit project. http://parabix.costar.sfu.ca/wiki/IDISAproject. 8

[3] Jinja documentation. http://jinja.pocoo.org/. 43

[4] The LLVM target-independent code generator. http://llvm.org/docs/CodeGenerator.html. 11, 12, 36, 42

[5] LLVM's Analysis and Transform Passes. http://llvm.org/docs/Passes.html. 10

[6] The parabix transposition. http://parabix.costar.sfu.ca/wiki/ParabixTransform. 25

[7] Python templating. https://wiki.python.org/moin/Templating. 43

[8] Test programs for measuring clock cycles and performance monitoring. http://www.agner.org/optimize/. 47

[9] Yosi Ben Asher and Nadav Rotem. Hybrid type legalization for a sparse SIMD instruction set. *ACM Trans. Archit. Code Optim.*, 10(3):11:1–11:14, September 2008. 1, 12, 13

[10] Robert D Cameron. u8u16–a high-speed utf-8 to utf-16 transcoder using parallel bit streams. Technical report, Technical Report TR 2007-18, Simon Fraser University, Burnaby, BC, Canada, 2007. 1

[11] Robert D Cameron. A case study in simd text processing with parallel bit streams: Utf-8 to utf-16 transcoding. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 91–98. ACM, 2008. 1, 9, 10

[12] Robert D Cameron, Ehsan Amiri, Kenneth S Herdy, Dan Lin, Thomas C Shermer, and Fred P Popowich. Parallel scanning with bitstream addition: An xml case study. In *Euro-Par 2011 Parallel Processing*, pages 2–13. Springer, 2011. 6, 7

[13] 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 center for advanced studies on collaborative research: meeting of minds*, page 17. ACM, 2008. viii, 1, 53

[14] Robert D. Cameron and Dan Lin. Architectural Support for SWAR Text Processing with Parallel Bit Streams: The Inductive Doubling Principle. *SIGPLAN Not.*, 44(3):337–348, March 2009. 1, 7, 9, 25, 27, 54

[15] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991. 11

[16] Dale Denis. High-Performance Regular Expression Matching with Parabix and LLVM. Master's thesis, Simon Fraser University, November 2014. 55

[17] Randall James Fisher. *General-purpose Simd Within a Register: Parallel Processing on Consumer Microprocessors*. PhD thesis, West Lafayette, IN, USA, 2003. AAI3108343. 1

[18] Hua Huang. IDISA+: A portable model for high performance SIMD programming. Master's thesis, Simon Fraser University, December 2011. 2, 4, 5, 8, 44, 49

[19] E.D. Johnson. *Quine-McCluskey: A Computerized Approach to Boolean Algebraic Optimization*. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1981. 25

[20] Valentine Kabanets and Jin-Yi Cai. Circuit minimization problem. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 73–79. ACM, 2000. 27

[21] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`. 2, 10

[22] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. 2, 10

[23] Ruby B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, 1995. 1

[24] Ruby B Lee, Zhijie Shi, and Xiao Yang. Efficient permutation instructions for fast software cryptography. *Micro, IEEE*, 21(6):56–69, 2001. 24

[25] Nigel Woodland Medforth. icxml: Accelerating xerces-c 3.1. 1 using the parabix framework. Master's thesis, Simon Fraser University, 2013. 1

[26] Huy Nguyen and Lizy Kurian John. Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 11–20, New York, NY, USA, 1999. ACM. 1

[27] Willard V Quine. The problem of simplifying truth functions. *American mathematical monthly*, pages 521–531, 1952. 27

[28] Arrvindh Shriraman Kenneth S. Herdy Dan Lin Benjamin R. Hull Meng Lin Robert D. Cameron, Thomas C. Shermer. Bitwise data parallelism in regular expression matching. 1, 6, 7, 10, 25, 33, 34, 56

[29] David A Terei and Manuel MT Chakravarty. An llvm backend for ghc. In *ACM Sigplan Notices*, volume 45, pages 109–120. ACM, 2010. 2, 10, 11

Appendix A

Appendices: Example Code From The IR Library

A.1 Transposition With Byte-Pack Algorithm

```
define <4 x i32> @packh_16(<4 x i32> %a, <4 x i32> %b) alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <16 x i8>
  %bb = bitcast <4 x i32> %b to <16 x i8>
  %rr = shufflevector <16 x i8> %bb, <16 x i8> %aa, <16 x i32> <i32 1,
        i32 3, i32 5, i32 7, i32 9, i32 11, i32 13, i32 15, i32 17,
        i32 19, i32 21, i32 23, i32 25, i32 27, i32 29, i32 31>

  %rr1 = bitcast <16 x i8> %rr to <4 x i32>
  ret <4 x i32> %rr1
}

define <4 x i32> @packl_16(<4 x i32> %a, <4 x i32> %b) alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <16 x i8>
  %bb = bitcast <4 x i32> %b to <16 x i8>
  %rr = shufflevector <16 x i8> %bb, <16 x i8> %aa, <16 x i32> <i32 0,
        i32 2, i32 4, i32 6, i32 8, i32 10, i32 12, i32 14, i32 16,
        i32 18, i32 20, i32 22, i32 24, i32 26, i32 28, i32 30>

  %rr1 = bitcast <16 x i8> %rr to <4 x i32>
  ret <4 x i32> %rr1
}

define <4 x i32> @ifh_1(<4 x i32> %cond, <4 x i32> %b, <4 x i32> %c)
alwaysinline {
entry:
  %not_cond = xor <4 x i32> %cond, <i32 -1, i32 -1, i32 -1, i32 -1>
```

```
  %t0 = and <4 x i32> %cond, %b
  %t1 = and <4 x i32> %not_cond, %c
  %r = or <4 x i32> %t0, %t1

  ret <4 x i32> %r
}

define <4 x i32> @srli_16(<4 x i32> %a, <8 x i16> %shift_mask)
alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <8 x i16>
  %r0 = lshr <8 x i16> %aa, %shift_mask
  %rr = bitcast <8 x i16> %r0 to <4 x i32>
  ret <4 x i32> %rr
}

define <4 x i32> @slli_16(<4 x i32> %a, <8 x i16> %shift_mask)
alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <8 x i16>
  %r0 = shl <8 x i16> %aa, %shift_mask
  %rr = bitcast <8 x i16> %r0 to <4 x i32>
  ret <4 x i32> %rr
}

define void @s2p_step_ir(<4 x i32> %s0, <4 x i32> %s1,
  <4 x i32> %hi_mask, <8 x i16> %shift_mask, <4 x i32>* %p0,
  <4 x i32>* %p1) alwaysinline {
entry:
  %t0 = call <4 x i32> @packh_16(<4 x i32> %s0, <4 x i32> %s1)
  %t1 = call <4 x i32> @packl_16(<4 x i32> %s0, <4 x i32> %s1)

  %t2 = call <4 x i32> @srli_16(<4 x i32> %t1, <8 x i16> %shift_mask)
  %q0 = call <4 x i32> @ifh_1(<4 x i32> %hi_mask, <4 x i32> %t0,
                              <4 x i32> %t2)
  %t3 = call <4 x i32> @slli_16(<4 x i32> %t0, <8 x i16> %shift_mask)
  %q1 = call <4 x i32> @ifh_1(<4 x i32> %hi_mask, <4 x i32> %t3,
                              <4 x i32> %t1)

  store <4 x i32> %q0, <4 x i32>* %p0
  store <4 x i32> %q1, <4 x i32>* %p1

  ret void
}

define <8 x i16> @const16_1() alwaysinline {
entry:
  ret <8 x i16> <i16 1, i16 1, i16 1, i16 1, i16 1, i16 1, i16 1, i16 1>
}

define <8 x i16> @const16_2() alwaysinline {
entry:
  ret <8 x i16> <i16 2, i16 2, i16 2, i16 2, i16 2, i16 2, i16 2, i16 2>
}
```

```
85  define <8 x i16> @const16_4() alwaysinline {
86  entry:
87    ret <8 x i16> <i16 4, i16 4, i16 4, i16 4, i16 4, i16 4, i16 4, i16 4>
88  }

90  define <4 x i32> @himask_2() alwaysinline {
91  entry:
92    ret <4 x i32> <i32 -1431655766, i32 -1431655766,
93                   i32 -1431655766, i32 -1431655766>
94  }

96  define <4 x i32> @himask_4() alwaysinline {
97  entry:
98    ret <4 x i32> <i32 -858993460, i32 -858993460,
99                   i32 -858993460, i32 -858993460>
100 }

102 define <4 x i32> @himask_8() alwaysinline {
103 entry:
104   ret <4 x i32> <i32 -252645136, i32 -252645136,
105                  i32 -252645136, i32 -252645136>
106 }

108 define void @s2p_bytepack_ir(<4 x i32> %s0, <4 x i32> %s1, <4 x i32> %s2,
109   <4 x i32> %s3, <4 x i32> %s4, <4 x i32> %s5, <4 x i32> %s6,
110   <4 x i32> %s7, <4 x i32>* %p0, <4 x i32>* %p1, <4 x i32>* %p2,
111   <4 x i32>* %p3, <4 x i32>* %p4, <4 x i32>* %p5, <4 x i32>* %p6,
112   <4 x i32>* %p7) {
113 entry:
114   %bit00224466_0 = alloca <4 x i32>, align 16
115   %bit00224466_1 = alloca <4 x i32>, align 16
116   %bit00224466_2 = alloca <4 x i32>, align 16
117   %bit00224466_3 = alloca <4 x i32>, align 16
118   %bit11335577_0 = alloca <4 x i32>, align 16
119   %bit11335577_1 = alloca <4 x i32>, align 16
120   %bit11335577_2 = alloca <4 x i32>, align 16
121   %bit11335577_3 = alloca <4 x i32>, align 16
122   %bit00004444_0 = alloca <4 x i32>, align 16
123   %bit22226666_0 = alloca <4 x i32>, align 16
124   %bit00004444_1 = alloca <4 x i32>, align 16
125   %bit22226666_1 = alloca <4 x i32>, align 16
126   %bit11115555_0 = alloca <4 x i32>, align 16
127   %bit33337777_0 = alloca <4 x i32>, align 16
128   %bit11115555_1 = alloca <4 x i32>, align 16
129   %bit33337777_1 = alloca <4 x i32>, align 16

131   %call10 = call <4 x i32> @himask_2()
132   %call11 = call <8 x i16> @const16_1()
133   call void @s2p_step_ir(<4 x i32> %s0, <4 x i32> %s1,
134   <4 x i32> %call10,
135   <8 x i16> %call11, <4 x i32>* %bit00224466_0,
136   <4 x i32>* %bit11335577_0)
137   %call14 = call <4 x i32> @himask_2()
138   %call15 = call <8 x i16> @const16_1()
139   call void @s2p_step_ir(<4 x i32> %s2, <4 x i32> %s3,
```

```
  <4 x i32> %call14,
  <8 x i16> %call15, <4 x i32>* %bit00224466_1,
  <4 x i32>* %bit11335577_1)
  %call18 = call <4 x i32> @himask_2()
  %call19 = call <8 x i16> @const16_1()
  call void @s2p_step_ir(<4 x i32> %s4, <4 x i32> %s5,
  <4 x i32> %call18,
  <8 x i16> %call19, <4 x i32>* %bit00224466_2,
  <4 x i32>* %bit11335577_2)
  %call22 = call <4 x i32> @himask_2()
  %call23 = call <8 x i16> @const16_1()
  call void @s2p_step_ir(<4 x i32> %s6, <4 x i32> %s7,
  <4 x i32> %call22,
  <8 x i16> %call23, <4 x i32>* %bit00224466_3,
  <4 x i32>* %bit11335577_3)
  %p23 = load <4 x i32>* %bit00224466_0, align 16
  %p24 = load <4 x i32>* %bit00224466_1, align 16
  %call24 = call <4 x i32> @himask_4()
  %call25 = call <8 x i16> @const16_2()
  call void @s2p_step_ir(<4 x i32> %p23, <4 x i32> %p24,
  <4 x i32> %call24,
  <8 x i16> %call25, <4 x i32>* %bit00004444_0,
  <4 x i32>* %bit22226666_0)
  %p25 = load <4 x i32>* %bit00224466_2, align 16
  %p26 = load <4 x i32>* %bit00224466_3, align 16
  %call26 = call <4 x i32> @himask_4()
  %call27 = call <8 x i16> @const16_2()
  call void @s2p_step_ir(<4 x i32> %p25, <4 x i32> %p26,
  <4 x i32> %call26,
  <8 x i16> %call27, <4 x i32>* %bit00004444_1,
  <4 x i32>* %bit22226666_1)
  %p27 = load <4 x i32>* %bit11335577_0, align 16
  %p28 = load <4 x i32>* %bit11335577_1, align 16
  %call28 = call <4 x i32> @himask_4()
  %call29 = call <8 x i16> @const16_2()
  call void @s2p_step_ir(<4 x i32> %p27, <4 x i32> %p28,
  <4 x i32> %call28,
  <8 x i16> %call29, <4 x i32>* %bit11115555_0,
  <4 x i32>* %bit33337777_0)
  %p29 = load <4 x i32>* %bit11335577_2, align 16
  %p30 = load <4 x i32>* %bit11335577_3, align 16
  %call30 = call <4 x i32> @himask_4()
  %call31 = call <8 x i16> @const16_2()
  call void @s2p_step_ir(<4 x i32> %p29, <4 x i32> %p30,
  <4 x i32> %call30,
  <8 x i16> %call31, <4 x i32>* %bit11115555_1,
  <4 x i32>* %bit33337777_1)

  %p31 = load <4 x i32>* %bit00004444_0, align 16
  %p32 = load <4 x i32>* %bit00004444_1, align 16
  %call32 = call <4 x i32> @himask_8()
  %call33 = call <8 x i16> @const16_4()
  call void @s2p_step_ir(<4 x i32> %p31, <4 x i32> %p32,
  <4 x i32> %call32,
  <8 x i16> %call33, <4 x i32>* %p0, <4 x i32>* %p4)
```

```
  %p33 = load <4 x i32>* %bit11115555_0, align 16
  %p34 = load <4 x i32>* %bit11115555_1, align 16
  %call36 = call <4 x i32> @himask_8()
  %call37 = call <8 x i16> @const16_4()
  call void @s2p_step_ir(<4 x i32> %p33, <4 x i32> %p34,
  <4 x i32> %call36,
  <8 x i16> %call37, <4 x i32>* %p1, <4 x i32>* %p5)
  %p35 = load <4 x i32>* %bit22226666_0, align 16
  %p36 = load <4 x i32>* %bit22226666_1, align 16
  %call40 = call <4 x i32> @himask_8()
  %call41 = call <8 x i16> @const16_4()
  call void @s2p_step_ir(<4 x i32> %p35, <4 x i32> %p36,
  <4 x i32> %call40,
  <8 x i16> %call41, <4 x i32>* %p2, <4 x i32>* %p6)
  %p37 = load <4 x i32>* %bit33337777_0, align 16
  %p38 = load <4 x i32>* %bit33337777_1, align 16
  %call44 = call <4 x i32> @himask_8()
  %call45 = call <8 x i16> @const16_4()
  call void @s2p_step_ir(<4 x i32> %p37, <4 x i32> %p38,
  <4 x i32> %call44,
  <8 x i16> %call45, <4 x i32>* %p3, <4 x i32>* %p7)

  ret void
}
```

A.2 Transposition With Ideal 3-Stage Algorithm

```
define <4 x i32> @packh_8(<4 x i32> %a, <4 x i32> %b) alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <32 x i4>
  %bb = bitcast <4 x i32> %b to <32 x i4>
  %rr = shufflevector <32 x i4> %bb, <32 x i4> %aa, <32 x i32>
   <i32 1, i32 3, i32 5, i32 7,  i32 9,  i32 11,  i32 13,  i32 15,
   i32 17,  i32 19,  i32 21,  i32 23,  i32 25,  i32 27,  i32 29,
   i32 31,  i32 33,  i32 35,  i32 37,  i32 39,  i32 41,  i32 43,
   i32 45,  i32 47,  i32 49,  i32 51,  i32 53,  i32 55,  i32 57,
   i32 59,  i32 61,  i32 63>

  %rr1 = bitcast <32 x i4> %rr to <4 x i32>
  ret <4 x i32> %rr1
}

define <4 x i32> @packl_8(<4 x i32> %a, <4 x i32> %b) alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <32 x i4>
  %bb = bitcast <4 x i32> %b to <32 x i4>
  %rr = shufflevector <32 x i4> %bb, <32 x i4> %aa, <32 x i32>
   <i32 0,  i32 2,  i32 4,  i32 6,  i32 8,  i32 10,  i32 12,  i32 14,
   i32 16,  i32 18,  i32 20,  i32 22,  i32 24,  i32 26,  i32 28,
   i32 30,  i32 32,  i32 34,  i32 36,  i32 38,  i32 40,  i32 42,
   i32 44,  i32 46,  i32 48,  i32 50,  i32 52,  i32 54,  i32 56,
   i32 58,  i32 60,  i32 62>
```

```
  %rr1 = bitcast <32 x i4> %rr to <4 x i32>
  ret <4 x i32> %rr1
}

define <4 x i32> @packh_4(<4 x i32> %a, <4 x i32> %b) alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <64 x i2>
  %bb = bitcast <4 x i32> %b to <64 x i2>
  %rr = shufflevector <64 x i2> %bb, <64 x i2> %aa, <64 x i32> <i32 1,
   i32 3,  i32 5,  i32 7,  i32 9,  i32 11,  i32 13,  i32 15,  i32 17,
   i32 19,  i32 21,  i32 23,  i32 25,  i32 27,  i32 29,  i32 31,
   i32 33,  i32 35,  i32 37,  i32 39,  i32 41,  i32 43,  i32 45,
   i32 47,  i32 49,  i32 51,  i32 53,  i32 55,  i32 57,  i32 59,
   i32 61,  i32 63,  i32 65,  i32 67,  i32 69,  i32 71,  i32 73,
   i32 75,  i32 77,  i32 79,  i32 81,  i32 83,  i32 85,  i32 87,
   i32 89,  i32 91,  i32 93,  i32 95,  i32 97,  i32 99,  i32 101,
   i32 103,  i32 105,  i32 107,  i32 109,  i32 111,  i32 113,
   i32 115,  i32 117,  i32 119,  i32 121,  i32 123,  i32 125,  i32 127>

  %rr1 = bitcast <64 x i2> %rr to <4 x i32>
  ret <4 x i32> %rr1
}

define <4 x i32> @packl_4(<4 x i32> %a, <4 x i32> %b) alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <64 x i2>
  %bb = bitcast <4 x i32> %b to <64 x i2>
  %rr = shufflevector <64 x i2> %bb, <64 x i2> %aa, <64 x i32>
   <i32 0,  i32 2,  i32 4,  i32 6,  i32 8,  i32 10,  i32 12,  i32 14,
   i32 16,  i32 18,  i32 20,  i32 22,  i32 24,  i32 26,  i32 28,
   i32 30,  i32 32,  i32 34,  i32 36,  i32 38,  i32 40,  i32 42,
   i32 44,  i32 46,  i32 48,  i32 50,  i32 52,  i32 54,  i32 56,
   i32 58,  i32 60,  i32 62,  i32 64,  i32 66,  i32 68,  i32 70,
   i32 72,  i32 74,  i32 76,  i32 78,  i32 80,  i32 82,  i32 84,
   i32 86,  i32 88,  i32 90,  i32 92,  i32 94,  i32 96,  i32 98,
   i32 100,  i32 102,  i32 104,  i32 106,  i32 108,  i32 110,
   i32 112,  i32 114,  i32 116,  i32 118,  i32 120,  i32 122,
   i32 124,  i32 126>

  %rr1 = bitcast <64 x i2> %rr to <4 x i32>
  ret <4 x i32> %rr1
}

define <4 x i32> @packh_2(<4 x i32> %a, <4 x i32> %b) alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <128 x i1>
  %bb = bitcast <4 x i32> %b to <128 x i1>
  %rr = shufflevector <128 x i1> %bb, <128 x i1> %aa, <128 x i32>
   <i32 1,  i32 3,  i32 5,  i32 7,  i32 9,  i32 11,  i32 13,  i32 15,
   i32 17,  i32 19,  i32 21,  i32 23,  i32 25,  i32 27,  i32 29,  i32
   31,  i32 33,  i32 35,  i32 37,  i32 39,  i32 41,  i32 43,  i32 45,
   i32 47,  i32 49,  i32 51,  i32 53,  i32 55,  i32 57,  i32 59,  i32
   61,  i32 63,  i32 65,  i32 67,  i32 69,  i32 71,  i32 73,  i32 75,
   i32 77,  i32 79,  i32 81,  i32 83,  i32 85,  i32 87,  i32 89,  i32
```

```
   91,  i32 93,  i32 95,  i32 97,  i32 99,  i32 101,  i32 103,  i32
   105,  i32 107,  i32 109,  i32 111,  i32 113,  i32 115,  i32 117,
   i32 119,  i32 121,  i32 123,  i32 125,  i32 127,  i32 129,  i32
   131,  i32 133,  i32 135,  i32 137,  i32 139,  i32 141,  i32 143,
   i32 145,  i32 147,  i32 149,  i32 151,  i32 153,  i32 155,  i32
   157,  i32 159,  i32 161,  i32 163,  i32 165,  i32 167,  i32 169,
   i32 171,  i32 173,  i32 175,  i32 177,  i32 179,  i32 181,  i32
   183,  i32 185,  i32 187,  i32 189,  i32 191,  i32 193,  i32 195,
   i32 197,  i32 199,  i32 201,  i32 203,  i32 205,  i32 207,  i32
   209,  i32 211,  i32 213,  i32 215,  i32 217,  i32 219,  i32 221,
   i32 223,  i32 225,  i32 227,  i32 229,  i32 231,  i32 233,  i32
   235,  i32 237,  i32 239,  i32 241,  i32 243,  i32 245,  i32 247,
   i32 249,  i32 251,  i32 253,  i32 255>

  %rr1 = bitcast <128 x i1> %rr to <4 x i32>
  ret <4 x i32> %rr1
}

define <4 x i32> @packl_2(<4 x i32> %a, <4 x i32> %b) alwaysinline {
entry:
  %aa = bitcast <4 x i32> %a to <128 x i1>
  %bb = bitcast <4 x i32> %b to <128 x i1>
  %rr = shufflevector <128 x i1> %bb, <128 x i1> %aa, <128 x i32>
        <i32 0,  i32 2,  i32 4,  i32 6,  i32 8,  i32 10,  i32 12,  i32
        14,  i32 16,  i32 18,  i32 20,  i32 22,  i32 24,  i32 26,  i32
        28,  i32 30,  i32 32,  i32 34,  i32 36,  i32 38,  i32 40,  i32
        42,  i32 44,  i32 46,  i32 48,  i32 50,  i32 52,  i32 54,  i32
        56,  i32 58,  i32 60,  i32 62,  i32 64,  i32 66,  i32 68,  i32
        70,  i32 72,  i32 74,  i32 76,  i32 78,  i32 80,  i32 82,  i32
        84,  i32 86,  i32 88,  i32 90,  i32 92,  i32 94,  i32 96,  i32
        98,  i32 100,  i32 102,  i32 104,  i32 106,  i32 108,  i32
        110,  i32 112,  i32 114,  i32 116,  i32 118,  i32 120,  i32
        122,  i32 124,  i32 126,  i32 128,  i32 130,  i32 132,  i32
        134,  i32 136,  i32 138,  i32 140,  i32 142,  i32 144,  i32
        146,  i32 148,  i32 150,  i32 152,  i32 154,  i32 156,  i32
        158,  i32 160,  i32 162,  i32 164,  i32 166,  i32 168,  i32
        170,  i32 172,  i32 174,  i32 176,  i32 178,  i32 180,  i32
        182,  i32 184,  i32 186,  i32 188,  i32 190,  i32 192,  i32
        194,  i32 196,  i32 198,  i32 200,  i32 202,  i32 204,  i32
        206,  i32 208,  i32 210,  i32 212,  i32 214,  i32 216,  i32
        218,  i32 220,  i32 222,  i32 224,  i32 226,  i32 228,  i32
        230,  i32 232,  i32 234,  i32 236,  i32 238,  i32 240,  i32
        242,  i32 244,  i32 246,  i32 248,  i32 250,  i32 252,  i32
        254>

  %rr1 = bitcast <128 x i1> %rr to <4 x i32>
  ret <4 x i32> %rr1
}

define void @s2p_ideal_ir(<4 x i32> %s0, <4 x i32> %s1, <4 x i32> %s2,
                          <4 x i32> %s3,
                          <4 x i32> %s4, <4 x i32> %s5, <4 x i32> %s6,
                          <4 x i32> %s7,
                          <4 x i32>* %p0, <4 x i32>* %p1,
                          <4 x i32>* %p2, <4 x i32>* %p3,
```

```
                          <4 x i32>* %p4, <4 x i32>* %p5,
                          <4 x i32>* %p6, <4 x i32>* %p7) {
entry:

  %bit0123_0 = call <4 x i32> @packh_8(<4 x i32> %s0, <4 x i32> %s1)
  %bit0123_1 = call <4 x i32> @packh_8(<4 x i32> %s2, <4 x i32> %s3)
  %bit0123_2 = call <4 x i32> @packh_8(<4 x i32> %s4, <4 x i32> %s5)
  %bit0123_3 = call <4 x i32> @packh_8(<4 x i32> %s6, <4 x i32> %s7)
  %bit4567_0 = call <4 x i32> @packl_8(<4 x i32> %s0, <4 x i32> %s1)
  %bit4567_1 = call <4 x i32> @packl_8(<4 x i32> %s2, <4 x i32> %s3)
  %bit4567_2 = call <4 x i32> @packl_8(<4 x i32> %s4, <4 x i32> %s5)
  %bit4567_3 = call <4 x i32> @packl_8(<4 x i32> %s6, <4 x i32> %s7)

  %bit01_0 = call <4 x i32> @packh_4(<4 x i32> %bit0123_0,
  <4 x i32> %bit0123_1)
  %bit01_1 = call <4 x i32> @packh_4(<4 x i32> %bit0123_2, <4 x i32>
  %bit0123_3)
  %bit23_0 = call <4 x i32> @packl_4(<4 x i32> %bit0123_0, <4 x i32>
  %bit0123_1)
  %bit23_1 = call <4 x i32> @packl_4(<4 x i32> %bit0123_2, <4 x i32>
  %bit0123_3)
  %bit45_0 = call <4 x i32> @packh_4(<4 x i32> %bit4567_0, <4 x i32>
  %bit4567_1)
  %bit45_1 = call <4 x i32> @packh_4(<4 x i32> %bit4567_2, <4 x i32>
  %bit4567_3)
  %bit67_0 = call <4 x i32> @packl_4(<4 x i32> %bit4567_0, <4 x i32>
  %bit4567_1)
  %bit67_1 = call <4 x i32> @packl_4(<4 x i32> %bit4567_2, <4 x i32>
  %bit4567_3)

  %pp0 = call <4 x i32> @packh_2(<4 x i32> %bit01_0, <4 x i32> %bit01_1)
  %pp1 = call <4 x i32> @packl_2(<4 x i32> %bit01_0, <4 x i32> %bit01_1)
  %pp2 = call <4 x i32> @packh_2(<4 x i32> %bit23_0, <4 x i32> %bit23_1)
  %pp3 = call <4 x i32> @packl_2(<4 x i32> %bit23_0, <4 x i32> %bit23_1)
  %pp4 = call <4 x i32> @packh_2(<4 x i32> %bit45_0, <4 x i32> %bit45_1)
  %pp5 = call <4 x i32> @packl_2(<4 x i32> %bit45_0, <4 x i32> %bit45_1)
  %pp6 = call <4 x i32> @packh_2(<4 x i32> %bit67_0, <4 x i32> %bit67_1)
  %pp7 = call <4 x i32> @packl_2(<4 x i32> %bit67_0, <4 x i32> %bit67_1)

  store <4 x i32> %pp0, <4 x i32>* %p0
  store <4 x i32> %pp1, <4 x i32>* %p1
  store <4 x i32> %pp2, <4 x i32>* %p2
  store <4 x i32> %pp3, <4 x i32>* %p3
  store <4 x i32> %pp4, <4 x i32>* %p4
  store <4 x i32> %pp5, <4 x i32>* %p5
  store <4 x i32> %pp6, <4 x i32>* %p6
  store <4 x i32> %pp7, <4 x i32>* %p7

  ret void
}
```