Example of the Use of the Style for Writing Theses at CS, SFU

by

Brown Burger

M.Sc., Even Better University, 1994

B.Sc. (Hons.), Even Better University, 1990

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

Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2013

APPROVAL

Name:	Brown Burger
Degree:	Doctor of Philosophy
Title of Thesis:	Example of the Use of the Style for Writing Theses at CS, SFU

Examining Committee: Dr. Brian Funt
Chair

Dr. Torsten Möller,
Professor, Senior Supervisor

Dr. Ghassan Hamarneh,
Associate Professor, Senior Supervisor

Dr. Mirza Faisal Beg,
Associate Professor, SFU Examiner

Dr. Christopher R. Johnson,
External Examiner, Distinguished Professor of
Computer Science, University of Utah

Date Approved: September 9th, 2013

Partial Copyright Licence

SFU

Abstract

Here you put the abstract of the thesis.

To whomever whoever reads this!

"Don't worry, Gromit. Everything's under control!"

— The Wrong Trousers, AARDMAN ANIMATIONS*, 1993*

Acknowledgments

Here go all the people you want to thank.

Contents

Approval ii

Partial Copyright License iii

Abstract iv

Dedication v

Quotation vi

Acknowledgments vii

Contents viii

List of Tables x

List of Figures xi

List of Programs xii

Preface xiii

1 Introduction 1

2 LLVM Background 2

2.1 LLVM Target-Independent Code Generator 2

2.1.1 Vector and Legalization 3

3 Design Objectives 5

3.1 Efficient Code Generation 7

4 Vector of $i2^k$ 8
4.1 Redefine Legality 9
4.2 In-place Lowering Strategy 10
4.2.1 Lowering for $vXi2$ 10
4.2.2 Inductive Doubling Principle 13
4.3 LLVM Vector Operation of $i2^k$ 17
4.4 Long Stream Addition 19

5 Implementation 22
5.1 Standard Method For Custom Lowering 24
5.1.1 Custom Lowering Strategies 24
5.1.2 DAG Combiner 24
5.2 Templated Implementation 26
5.2.1 Code Generation For $i2$ Vector 26
5.2.2 Test Code And IR Library Generation 27

6 Performance Evaluation 29
6.1 Vector of $i2^k$ Performance 29
6.1.1 Methodology 29
6.1.2 Performance Against IDISA 30
6.1.3 Performance Against LLVM 32
6.2 Parabix Critical Operations 32
6.2.1 Ideal 3-Stage Transposition on the Intel Haswell 33
6.2.2 Long Stream Addition 34

7 Conclusion 38

Bibliography 39

List of Tables

4.1 Supported operations and its semantics. 11
4.2 Legalize operations on $vXi1$ with iX equivalence. 11
4.3 Truth table of ADD on 2-bit integers and the minimized boolean functions for C. 12
4.4 Algorithm to lower $v32i4$ operations. 15
4.5 Rearranging index for BUILD VECTOR on $v64i2$ 18

6.1 Hardware Configuration 31
6.2 Software Configuration 31
6.3 Performance against LLVM native support for $i2^k$ vectors 32
6.4 Performance comparison of XML Validator (xmlwf) 32
6.5 Performance comparison of UTF-8 UTF-16 Transcoder 33
6.6 Ideal 3-Stage Transposition with PEXT 33
6.7 Micro benchmark for long stream addition against LLVM's original implementation. . 34

List of Figures

4.1 Type legalize process for $v32i1$ vector 9
4.2 Comparison between LLVM default legalize process and in-place lowering. 10
4.3 Addition of two $v32i4$ vectors. 14
4.4 LLVM default type legalization of $v8i4$ to $v8i8$. 14

5.1 System overview: modified instruction selection process 23
5.2 Test system overview. 27

6.1 Test Performance with XOR 30
6.2 Reciprocal instruction throughput against IDISA library 35
6.3 Total CPU cycles against IDISA library 36
6.4 Vector of $i2$ tested in a loop 37
6.5 Performance of icgrep with long stream addition 37

List of Programs

3.1 Shufflevector implementation of packh. 6
3.2 Shufflevector implementation of mergeh. 6
4.1 The function generated to lower ADD on $v64i2$. 13
4.2 Inductive doubling principle on $v16i8$ multiplication. 16
5.1 Implementation of `hsimd<2>::packh` with PEXT. 25
5.2 Templates for the IR Libray 28

Preface

Here go all the interesting reasons why you decided to write this thesis.

Chapter 1

Introduction

Chapter 2

LLVM Background

Introduce LLVM and IR.

2.1 LLVM Target-Independent Code Generator

Introduce a little bit. Need to talk about selectionDAG

LLVM as a modularized compiler tool chain, allows us to implement our optimization conveniently. As we discussed before, LLVM has a general code generation algorithm, the first stage is instruction selection, listed below[4]:

- Initial SelectionDAG Construction: generate SelectionDAG from LLVM IR.
- DAG Combine 1
- Legalize Types Phase
- Post Legalize Type DAG Combine
- Legalize Phase
- DAG Combine 2
- Instruction Select Phase
- Scheduling and Formation Phase

We can see there are DAG combine passes after the initial construction and each legalize phase[4]. DAG combine passes optimize 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 messy output.

The other advantage of DAG combiner is, you can choose the combine timing on your own. If you choose to combine before Legalize Types Phase, you can freely introduce illegal types into your combined results. This is different from legalizing phases. Generally speaking, you cannot introduce illegal types in Legalize Type Phase and cannot introduce illegal operations in Legalize Phase. This puts a limitation on machine-independent legalize strategies: $i8$ is the minimum integer type on X86 arch, programmer needs to extend every integer less than 8 bits to $i8$ before returning it to the DAG.

For each target, LLVM has a specific target lowering class, e.g. X86ISelLowering for X86 arch. We put our code generation logic here as a DAG combiner. For each shufflevector node, we check if the mask matches certain pattern, say consecutive odd numbers from 1 to 31, then we combine the node into a tree of operations, say X86ISD::PACKUS and logic/shifting.

2.1.1 Vector and Legalization

SIMD operations exploit data parallelism by performing the same operation on different data at the same time. Those data are grouped together as vectors. LLVM uses the notion `<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 $v4i32$.

In LLVM IR, programmer can write any kind of vectors, even $v1024i3$, and those vectors may not be supported by the target machine. LLVM has the notion of a "legal" vs. "illegal". A type is legal for a target only if it is supported by some operation. In selectionDAG, a DAG node is legal only if the target supports the operation and operands type. For example, $v16i8$ is legal on X86 SSE2 architecture, since the architecture supports ADD on 2 $v16i8$ vectors; but it does not support multiplication on 2 $v16i8$ vectors, so that the DAG node MUL on $v16i8$ is illegal. LLVM has Legalize Types and Legalize Operations Phases to turn illegal type or DAG into legal[4].

Legalize type 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 $v1iX$ as the edge case when LLVM is trying to split the incoming vector into sub vectors.
- **Vector Widening** adds dummy elements to make the vector fit the right register size. It will not change the type of the elements, e.g. $v4i8$ to $v16i8$.
- **Vector Element Promotion** preserves the number of elements, but promote the element type to a wider size, e.g. $v4i8$ to $v4i32$.

After type legalization, we may still have illegal DAG node, such as multiplication on $v16i8$ for

X86 SSE2 architecture; thus we need legalize operations phase. There are three strategies in this phase:

- **Expansion**: Use another sequence of operations to emulate the operation. Expansion strategy is often general.
- **Promotion**: Promote the operand type to a larger type that support the operation.
- **Custom**: Write a target-specific code to implement the legalization. Similar to Expansion, but with a specific target in mind.

When talk about Parabix background, talk about IDISA and ”bitcast”.

Chapter 3

Design Objectives

Need to explain Parabix transposition (byte-pack algorithm) and reverse transposition here. Also ideal transposition.

Shufflevector is a powerful LLVM instruction that can be used to manipulate vectors in a target-independent fashion. Its syntax is [1]:

```
<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 the other word, the element indexes are $0 \ldots n-1$ for `v1` and $n \ldots 2n-1$ for `v2`. The `mask` is an array of constant integer indexes, which indicates the elements we want to extract to form the `result`. Either `v1` or `v2` can be "undefined" to do shuffle within one vector.

With shufflevector, we can express IDISA functions like `hsimd::packh`, `hsimd::packl` in "pure" IR. "Pure" here means machine-independent. For an example, `hsimd<16>::packh(A, B)` extracts the high 8 bits of each field in A and B, concatenates them together to form the result vector. (Maybe a pic here). In traditional C++ library, we have to realize this operation for each platform: we use unsigned saturation $packuswb$ for X86 arch and use $vuzpq\_u8$ for NEON arch; both require some tweak on operands A, B. On the contrary, we can write `hsimd<16>::packh` for all the platforms as Program 3.1.

In this program, we first bitcast operands into $i8$ vectors. "Bitcast" is also an useful LLVM operation that converts between integer, vector and FP-values; it changes the data type without moving or modifying the data; so it requires the source and result type to have the same bit size. We then fill in the indexes of all the high bits in order, which is 1, 3, . . . , 31. (MB Pic). Target-specific logic is thus left to the LLVM backend and is no longer the burden of the programmer. We optimize LLVM backend for better code generation later in this thesis.

```
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
}
```

Program 3.1: Shufflevector implementation of packh, it is machine independent. `<4 x i32>` is a general vector type we use for all SIMD registers to simplify function interface.

```
define <4 x i32> @mergeh_8(<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 8,
        i32 24, i32 9, i32 25, i32 10, i32 26, i32 11, i32 27, i32 12,
        i32 28, i32 13, i32 29, i32 14, i32 30, i32 15, i32 31>

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

Program 3.2: Shufflevector implementation of mergeh, the function is self-explanatory and easy to understand.

Shufflevector can be used for a variety of operations, e.g. for `hsimd<16>::packl`, which packs all the low bits of each 16-bit field, we just change the mask to be 0, 2, 4, 6, ..., 30; for `packh` with different field width w, we can first bitcast the operands into vectors of $w/2$ element, and then shuffle with the similar increasing odd number mask. We also write the code of `esimd<8>::mergeh` in Program 3.2, where the function is self-explanatory and any programmer who understands shufflevector can understand it easily.

However, the shufflevector has one limitation: the `mask` can only contain constant integers, which prevents us from generating dynamic shuffle operation. Parabix deletion algorithm, for instance, takes two SIMD registers as input: A and DeletionMask. It will delete the bits from A marked by DeletionMask (delete the i_{th} bit of A if DeletionMask$_i = 1$) and shift the rest of A to the lower end. (MB Pic) This algorithm cannot be implemented in shufflevector because DeletionMask is not a constant. Given a DeletionMask, one can construct a shuffle mask to do both deletion and shifting, but LLVM does not support dynamic shuffle masks generated during the runtime.

3.1 Efficient Code Generation

(UNDER CONSTRUCTION)

(May need to be moved into implementation. For here, just give examples of custom lowering)

Shufflevector is a powerful instruction, but it's not supported directly by the hardware yet. So machine-specific code generation is necessary.

Chapter 4

Vector of $i2^k$

Parabix operation works on full range of vector types. For 128-bit SIMD register, Parabix supports $v128i1$, $v64i2$, $v32i4$, $\ldots$, $v1i128$, we call them as the vector of $i2^k$. Vector type $vXi8$, $vXi16$, $\ldots$, $vXi64$ is widely used for multimedia processing, digital signal processing and Parabix technology, they are well supported by the LLVM infrastructure, but the rest vector type with smaller element does not have perfect implementation. 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 current LLVM 3.4 backend for X86 architecture. After seeing a $v128i1$ vector, Type Legalize Phase would promote element type $i1$ to $i8$, and then split the vector to fit 128 bits register size; thus the incoming $v128i1$ turns into 8 $v16i8$ vectors. If we write AND on 2 $v128i1$ vectors, LLVM would produce 8 pairs of AND on $v16i8$ and also operations to truncate and concatenate back the $v128i1$ result; while we can simply bitcast $v128i1$ to any legal 128-bit vector like $v4i32$, do AND on them 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 would lead to huge selectionDAG generation and thus poor 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.

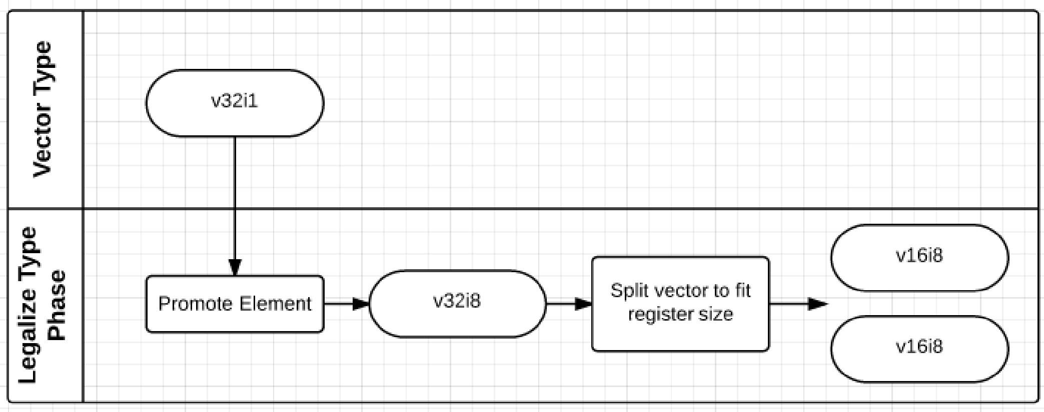

Figure 4.1: Type legalize process for $v32i1$ vector

4.1 Redefine Legality

In Chapter 2 we know that LLVM has three ways to legalize vector types: Scalarization, Vector Widening and Vector Element Promotion. None of these strategies would 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 would be the 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 intrinsics 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 will need to handle more illegal operations. LLVM has the facility to "expand" an illegal operation, so that we do not need to implement every operation on the type. For example $v32i1$, we did not lower its shufflevector but we can still write shufflevector on $v32i1$ in IR, and LLVM could expand it into sequence of extracting and inserting vector elements. Of course the performance is not good at all. So the new question arises, what is the necessary operations set to fully support a legal type, that every possible IR statement with this type can be compiled into a native machine code?

This question is hard to answer. In practice, we implemented (1) common binary functions listed in Table 4.2; (2) basic vector operations like INSERT VECTOR ELT, EXTRACT VECTOR ELT and BUILD VECTOR. All the meaningful test IR files we wrote work properly under this operations set.

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 would rather look at the same data with a different type. A trivial example would be 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 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 would require to shift different element with different offsets.

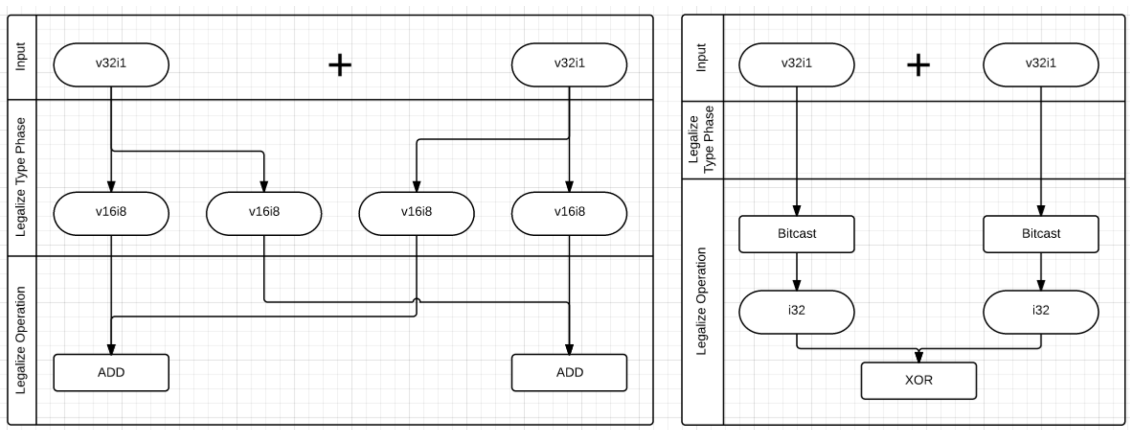

Figure 4.2: Comparison between LLVM default legalize process (left) and in-place lowering (right). The right marks $v32i1$ type legal and handles the operation ADD in the legalize operations phase. This will keep the data in the general registers without being promoted or expanded.

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[5] 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[10] and it is the principle overhead for some application like regular expression matching[14]. 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

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.

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.

of C separately with the logic combinations of a_0, a_1, b_0, b_1 and turn this into *Circuit Minimization Problem*: find minimized boolean functions for c_0 and c_1. We use Quine-McCluskey algorithm[12] to solve it; an example can be found in Table 4.3.

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.

Once we get the minimized boolean functions, we can apply it onto the whole $vXi2$ vector. Let us introduce one operation first, `IFH1`. `IFH1(Mask, A, B)` 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 to $(\texttt{Mask} \wedge \texttt{A}) \vee (\neg\texttt{Mask} \wedge \texttt{B})$.

Then 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 `IFH1` 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:

- 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.

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

```
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 $v64i2$.

4.2.2 Inductive Doubling Principle

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`, `hsimd<8>::packl` and `esimd<4>::mergeh`, which are required by Ideal Three-Stage Transposition / Inverse Transposition; $vXi4$ is also the critical part of the IDISA model. But unfortunately, the strategies discussed above cannot be applied to $vXi4$ efficiently.

Circuit Minimization Problem is NP-hard[7, 13]. For $vXi4$, we would have 4 boolean functions of 8 variables: $c_i = f_i(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3), i \in \{0, 1, 2, 3\}$, and it is known that most boolean functions on n variables have circuit complexity at least $2^n/n$[13] 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 *Inductive Doubling Principle* [10] and we will show that this general principle can be applied for $vXi4$ and even wider vector element type, e.g. multiplication on $v16i8$, to get better performance.

We use $v32i4$ as an example to illustrate Inductive Doubling Principle. To legalize $v32i4$, LLVM would promote this type into $v32i8$, widen every element to $i8$ and shift every element except the first one. Figure 4.4 shows an example of widening $v8i4$ into $v8i8$, we can see unnecessary movement of vector element during widening. On a platform with 128 bits SIMD register, $v32i8$ will further

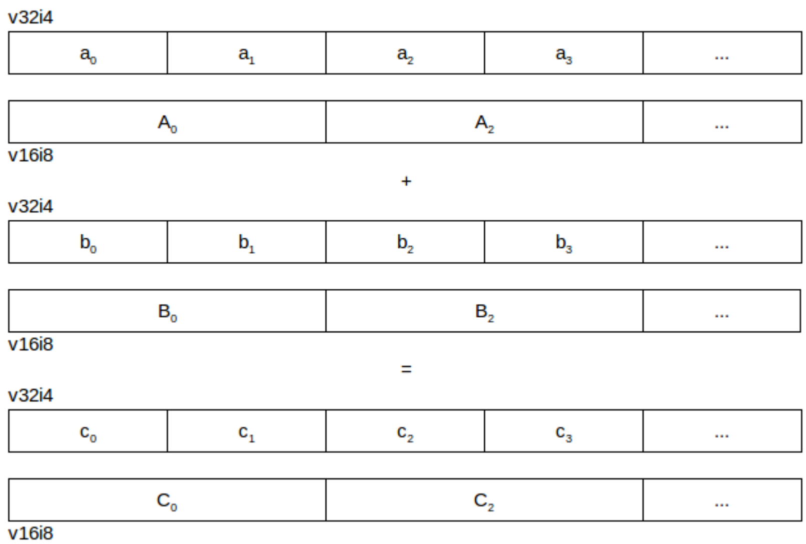

Figure 4.3: To add 2 $v32i4$ vectors, a and b, we bitcast them into $v16i8$ vectors. The lower 4 bits of $A_0 + B_0$ gives us c_1. We then mask out a_1 and b_1 (set them to zero), do add again, and the higher 4 bits of the sum is c_0.

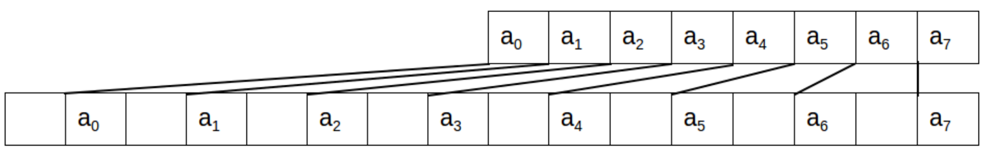

Figure 4.4: LLVM default type legalization of $v8i4$ to $v8i8$. a_0 to a_7 are $i4$ elements and they are shifted with different offsets during the element type promotion.

be divided into two $v16i8$ and take 2 registers to hold, while the original type $v32i4$ has 128 bits in size and should be able to reside in only 1 register. Inductive Doubling Principle could achieve the latter for us. It would bitcast the vector in-place, view the same register as $v16i8$ type and emulate $i4$ 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:

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

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

Generally, as Dr Cameron 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.”[10]. He described four key elements of this architecture:

- A core set of binary functions on iX vectors, for all $X = 2^k$. To work with parallel bit streams, the operation ADD, SUB, SHL, SRL and ROTL (rotate left) comprise the set.
- A set of *half-operand modifiers* that make possible the inductive processing of $i2X$ in terms of combinations of iX. These modifiers select either the lower X bits of each $i2X$ element or the higher X bits.
- Packing operations that compress two `<N x iX>` vectors into one `<2N x i(X/2)>` vector. Like `hsimd<8>::packh` we mentioned in Chapter 3.
- Merging operations that produce one `<N x iX>` vector from two `<2N x i(X/2)>` vectors. Like `esimd<8>::mergeh`, it is the inverse function of packing.

For this section, we will only use the fact that 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 needs explain 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.

$$C = IFH1(HiMask_8, HiBits, LowBits)$$

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 and 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 generates 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. Look back to Figure 4.3, we

```
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)
```

```
# 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
```

Program 4.2: Inductive doubling principle on $v16i8$ multiplication. LLVM 3.4 generate poor machine code, which will $pextrb$ every $i8$ field and multiply them with $mulb$. We simplify it through 2 $pmullw$, which is the multiplication on $v8i16$.

need to mask out a_1, b_1 and do add again, because $a_1 + b_1$ may produce carry bit to the high 4 bits. If we mask out only the high bit of a_1 and b_1, still we will not produce 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:

$$m = (10001000\dots1000)_2 \tag{4.3}$$

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

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

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

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

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 1-add algorithm while keeping the rest $v32i4$ operations under general in-place promotion strategy. We will 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 will discuss how to custom lower these nodes on $i2^k$ vectors.

BUILD VECTOR takes an array of scalars as input and output a vector with these scalars as elements. Take $v64i2$ vector on X86 SSE2 architecture for an example; ideally, the input would provide an array of 64 $i2$ scalars and BUILD VECTOR assembles them into a $v64i2$ vector. More specifically, since $i2$ is illegal on all X86 architecture, the legal input is actually 64 $i8$ scalars. The naive approach would be creating an ”empty” $v64i2$ vector, truncating every $i8$ into $i2$ and inserting it 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 is all $i8$. We rearrange them according to Table 4.5 and build 4 $v16i8$ vectors V_1, V_2, V_3, V_4. The final build result is:

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

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 $v32i4$.

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

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

EXTRACT VECTOR ELT takes 2 operands, a vector V and an index i. It returns the i_{th} element of V. The semantics would not allow much parallelism in the implementation. On X86 architecture, there are built-in intrinsics to extract vector element, such as $pextrb$ ($i8$), $pextrw$ ($i16$), $pextrd$ ($i32$) and $pextrq$ ($i64$); for smaller element type, 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 type 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 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 type, 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. In stead, 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 and of course this needs to be translated into an block-at-a-time implementation[14]. 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`. Dr Cameron developed a general model using SIMD methods for efficient long-stream addition up to 4096 bits in [14].

In this section, we will replace the internal logic of wide integer addition ($i128$, $i256$ etc.) of LLVM with the Parabix long-stream addition. Same with Dr Cameron's work in [14], we assume the following SIMD operations on $i64$ vectors 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 [14], which would distribute the `N` bits of the mask, one bit each to the lower end of the `N` $i64$ elements.

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

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 will 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 implement 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 would be introduced, like 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 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 will describe our realization of Parabix technology inside 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 principle 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 the separate module.
3. Put our code in auto-generated, thorough test.

Most of our code sits in LLVM Target-Independent Code Generator[4]. From Chapter 4, we know that current type legalization process of LLVM have big performance penalty for small element vectors, so our approach would mark $i1$, $i2$ and $i4$ vector legal type first, and then handle them in Legalize Operation Phase. For convenience, we name this set of vector types *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 would 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 XMM7 registers if we have SSE2 or to 16-byte stack slots otherwise.

Now Legalize Type Phase would recognize our $i1$, $i2$, $i4$ vectors as legal and pass them onto Legalize Operation Phase. We have two major methods to handle $i2^k$ vectors here: *Custom Lowering* and *DAG Combining*.

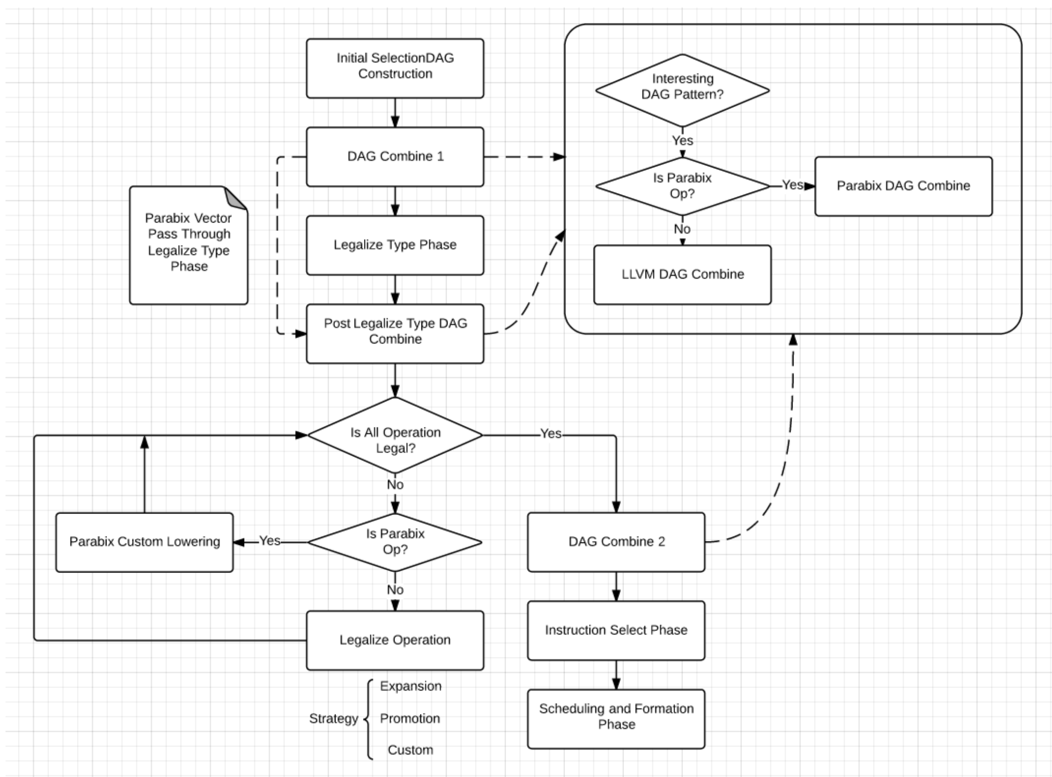

Figure 5.1: Overview of the modified instruction selection process. Logic for Parabix vectors are hooked into two places: the Legalize Operation Phase and DAG Combine Phases. Parabix Custom Lowering and Parabix DAG Combine are both modularized and separated.

Figure 5.1 gives an overview of our implementation. Custom Lowering resides in the Legalize Operation Phase and LLVM does it through iteration, which allows the legalizer to introduce new illegal operations within each iteration. Every time the legalizer finds an illegal operation, it will check if that is a Parabix operation and if so redirect to the Parabix Custom Lowering module. DAG Combining is similar but we have multiple combine timings available and it is designed mainly for cleaning up the messy output of the legalizers.

In the following sections, we will 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 will 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 Legalize Type 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 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 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 custom lowering facility and 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.

The first example is shufflevector for packh(”pack high”) and packl(”pack low”). LLVM 3.4 does not generate the best assembly code for $v16i8$ packing, it generates a sequence of $pextrw$ and $pinsrw$. So we create the following DAG Combiner:

- **Pattern**: shufflevector on $v16i8$ with mask = 0, 2, 4, . . . , 30 (packl) or mask = 1, 3, 5, . . . , 31 (packh).
- **Combine Result**: one PACKUS node, which would unsigned saturate two $v8i16$ into $v8i8$ vectors and concatenate them into one $v16i8$.

Furthermore, since $v128i1$, $v64i2$ and $v32i4$ are legal vectors now, we have the chance to optimize shufflevector on them too. Still use packh/packl as an example, we can utilize the PEXT node introduced by the Intel Haswell Architecture, BMI2. PEXT is an useful instruction for bit manipulation on $i32$ and $i64$. Given the $i8$ variable A = abcdefgh, Mask = $(10101010)_2$, PEXT(A, Mask)

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

would return R = aceg (a,b,c,d,e,f,g are single bits). So PEXT would extract bits from A at the corresponding bit locations specified by Mask. With this in mind, we can implement `hsimd<2>::packh` as Program 5.1; for readability, it is in pseudo IR.

According to Program 5.1, we create the following DAG Combiner:

- **Pattern**: shufflevector on $v128i1$, $v64i2$ or $v32i4$ with mask = 0, 2, 4, $\ldots$, $NumElt \times 2 - 2$ or mask = 1, 3, 5, $\ldots$, $NumElt \times 2 - 1$. $NumElt$ 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 short cut for the programmer to do ad hoc optimizations and 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 with the same amount and we can have efficient realization for $v32i4$ with $v4i32$ shifts, while we apply In-place Promotion strategy for $v32i4$ arbitrary shifting in the Parabix custom lowering.

Apart from this, DAG Combiner can also optimize operations with illegal type in the phase DAG Combine 1, which is not possible in the custom lowering. 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 Legalize passes [4]. Second, it cannot utilize the legalization iteration; in custom lowering, general strategies may introduce new illegal operations and they are hard to avoid since ”illegal” is a target-specific concept; these illegal operations will be lowered in the next iteration and so on. The DAG Combiner, on the other hand, 1) Should not generate illegal operations in the phases after the Legalize Operation 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 many duplicated code, especially in the test cases; they are against software design principles and they are hard to maintain, sometimes even hard to write: a thorough test file for $i2^k$ vector could contain more than two thousand lines of code, most of which are in the same pattern. To keep DRY (don't repeat yourself) and save programmer time, we introduced Jinja template engine [3]. According to [6], Jinja belongs to the Engines Mixing Logic into Templates, it allows to embed 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, and the reader must read the driver code as well as the template file to understand the output. Examples can be found in Program 5.2.
- 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), and how to present these data is not its responsibility. In the other word, 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 formula which reside in one compact data script and we wrote template files to generate 11 C++ functions for them. This approach allows us to:

- Collect 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 plan to 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.

5.2.2 Test Code And IR Library Generation

To test the vector of $i2^k$, we need a pure IR library and a test driver. We could compile the IR library into one object file and link it with the driver, so that the driver could generate random test data, pass them to both the IR library and the reference library IDISA+ [11], and compare their results. The test system overview can be found in Figure 5.2. We use templates for both the IR library and the driver, some sample templates can be found in Program 5.2.

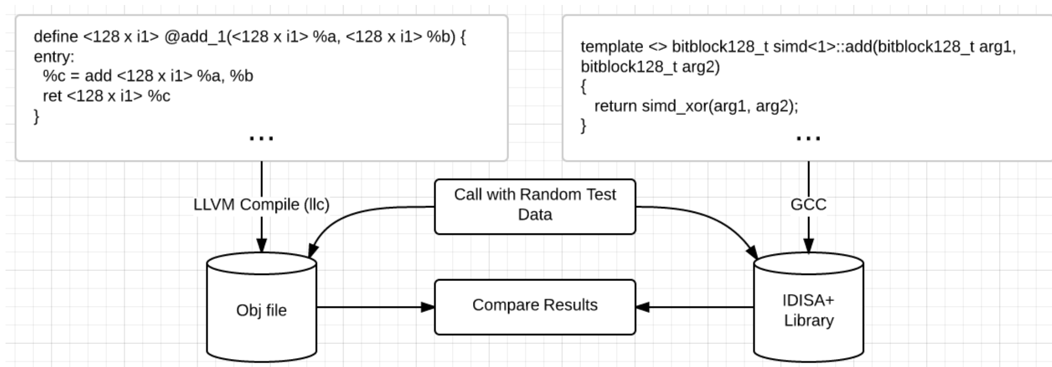

Figure 5.2: Test system overview. The pure IR library is first compiled into the native object file and then linked with the driver. The driver call functions from the both side to check correctness.

```
{% 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 5.2: 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 show that our LLVM backend could not only match performance with the hand-written library, but also provide a better chance to optimize according to the specific target. We would first validate our vector of $i2^k$ approaches, and then present the performance of some critical Parabix operations via application-level profile.

6.1 Vector of $i2^k$ Performance

In Chapter 4, we present different approaches to lower $i1$, $i2$, $i4$ and some $i8$ operations within one SIMD register. In this section, we would validate our approaches by showing the improved runtime 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. Dr. 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 pick the reciprocal throughput as our measurement and it is measured with a sequence of same instructions where subsequent instructions are independent of the previous ones. In Dr. 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.

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

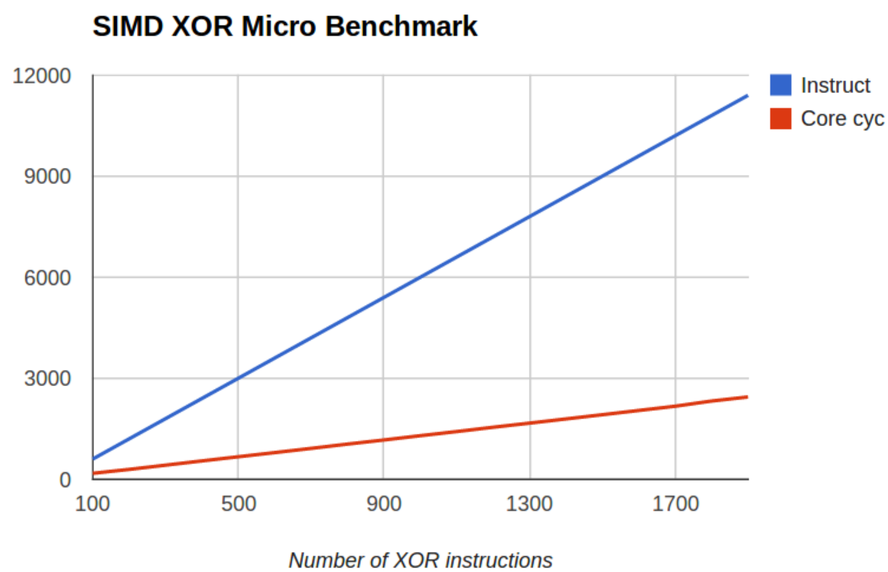

Figure 6.1: Test performance with XOR.

From the figure, we can see the instructions count and CPU cycles grows linearly with the number of XOR instructions. So we can conclude that Dr. 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 have more instructions. Note that from the figure, it seems the throughput of $xorps$ is 4, which is different from Intel's document (3 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 to 3. In order to eliminate this undesired effect, we would flatten the test code in the following sections.

In the following sections, we would write micro benchmarks with Agner Fog's test program and compare reciprocal throughput 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 bitcode (binary form of LLVM IR) and then link / optimize them together. The default compile flag is to use Intel SSE2 instruction set and 64 bit.

6.1.2 Performance Against IDISA

We compare our lowering on pure IR function with IDISA+ Library [11] which is written in C++. To test each operation, we generate a sequence of 500 such operations where none them has to wait

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 3.4
File System	Ext4

Table 6.2: Software Configuration

for the previous one. 100 operations seems not long enough for a stable result. The performance comparison is listed in Figure 6.2 and Figure 6.3. We can see for $i1$ and $i4$ vectors, IR library has the similar performance 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 level. For IDISA library, `simd<2>::ugt` got 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 and this may help the compiler to optimize. The expansion of `ugt_2` is delayed until the instruction selection phase, right before machine code generation. We checked that 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.

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 better total CPU cycles, but if we write our instructions sequence with a loop, IDISA library would win (Figure 6.4). Loop optimization should be responsible for this difference and we did observe some kind of hoisting in the assembly code. Early expansion in IDISA also provides more optimization opportunity to the compiler front end.

Further more, from the reciprocal throughput comparison (Figure 6.2), IR libary loses a bit on $i1$ vectors but wins most of the cases in $i2$ and $i4$; it may relate to a better instruction selection. IDISA library is generated from a strategy pool based on the number of basic instructions which are treated equally as cost 1. But basic instructions actually have different throughput in the real hardware, and LLVM backend are aware of that, thus selecting better instructions.

6.1.3 Performance Against LLVM

We compare our lowering with native LLVM. LLVM could not handle $i2$, $i4$ vectors and handle $i1$ vectors slowly. Detailed performance data can be found in Table 6.3. We can see that our approach fills the gap of 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 rest number means the ratio of CPU cycles speed up: add takes 302 times of cycles that our lowering needs. For $i8$, we apply 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 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 operation is replaced. The default compile flag is to use Intel SSE2 instruction set and 64 bit.

	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
xmlwf0	3.93	4.364	4.553	4.891	5.18
xmlwf0 on Haswell	3.929	4.363	4.554	4.876	5.178
xmlwf1	3.929	4.371	4.566	4.861	5.186
xmlwf1 on Haswell	3.566	3.978	4.163	4.451	4.787

Table 6.4: Performance comparison of XML validator (xmlwf), in 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.

Table 6.4 shows the performance of the XML Validator. The only difference of xmlwf0 and xmlwf1 is their transposition code, and the one in xmlwf1 is written in pure IR with the byte-pack algorithm. We can see xmlwf0 and xmlwf1 share almost identical performance and it is not for free. LLVM 3.4

	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
U8u16_0	281.46	37.11	40.06	244.94	10.2
U8u16_0 Haswell	272.68	34.21	39.84	242.56	10.11
U8u16_1	284.17	36.71	41.65	255.57	10.6
U8u16_1 Haswell	267.14	34.64	38.53	237.66	9.98

Table 6.5: Performance comparison of UTF-8 UTF-16 transcoder, in million CPU cycles. U8u16_0 is written in IDISA, U8u16_1 has the transposition and inverse transposition part replaced.

cannot handle packing on 16-bit field width very well and we custom lower the shufflevector and generate PACKUS instruction for X86.

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 SSE2 intrinsics and only SSE2 instructions can be generated; but we got a slightly better performance for xmlwf1, because the IR library is target-independent and LLVM backend knows other instruction sets like SSE3, SSE4 is available on this platform, it generates better code with them.

Similar performance data on UTF-8 to UTF-16 transcoder is listed in Table 6.5. U8u16_0 is written in IDISA and U8u16_1 has both the transposition and inverse transposition part replaced. We also tried to compile them on the full Haswell, which gave us similar performance benefit.

6.2.1 Ideal 3-Stage Transposition on the Intel Haswell

Intel Haswell architecture introduces PEXT operation which can be used for the ideal 3-stage transposition. We evaluated its performance in Table 6.6. We can see the performance drops with PEXT, but the major reason is that PEXT can only work on $i32$ or $i64$ integer for the current architecture, not the algorithm. As the hardware evolves, we may have PEXT on SIMD registers directly and 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 [10]. Our approach provides a new chance to exploit future hardware benefit without changing the source code.

	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
xmlwf0 on Haswell	3.929	4.363	4.554	4.876	5.178
xmlwf1 on Haswell	3.566	3.978	4.163	4.451	4.787
xmlwf2 on Haswell	4.11	4.49	4.69	4.978	5.308

Table 6.6: Ideal 3-stage transposition on xmlwf2. Xmlwf1 uses byte-pack algorithm in IR, xmlwf0 uses the same algorithm in IDISA.

6.2.2 Long Stream Addition

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

We wrote micro benchmark with Dr. Fog's test program and we put 200 independent additions on $i128$ and $i256$. It was tricky to make the test program right, we generated random data for the operands and we carefully inserted the carry-out bit back to the return value so that our long stream addition logic would not be optimized away. 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 6.7.

	Core CPU Cycles	Instructions
Long stream addition on $i128$	3043	7952
LLVM on $i128$	1455	4199
Long stream addition on $i256$	4103	10152
LLVM on $i256$	4234	9798

Table 6.7: Micro benchmark for long stream addition against LLVM's original implementation.

Long stream addition does not perform well on $i128$, since there is only two sequential additions involved (1 $addq$ and 1 $adcq$) and parallel computing would not save much but introduce complexity. However, we can see on $i256$ long stream addition has almost identical performance with the sequential implementation, 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 1.35, under 2; thus we could confidently predict that on the Intel AVX512, long stream addition on $i512$ would perform better than the sequential addition.

We then show application level profile of 'icgrep' which is an tool for regular expression matching with bitwise data parallelism. We replaced the internal "add with carry" logic with one single intrinsic and plotted the performance in Figure 6.5. We can see the performance drops because that version of icgrep works with 128-bit SIMD registers and long stream addition does not work well on $i128$. We could expect better performance on wider SIMD registers.

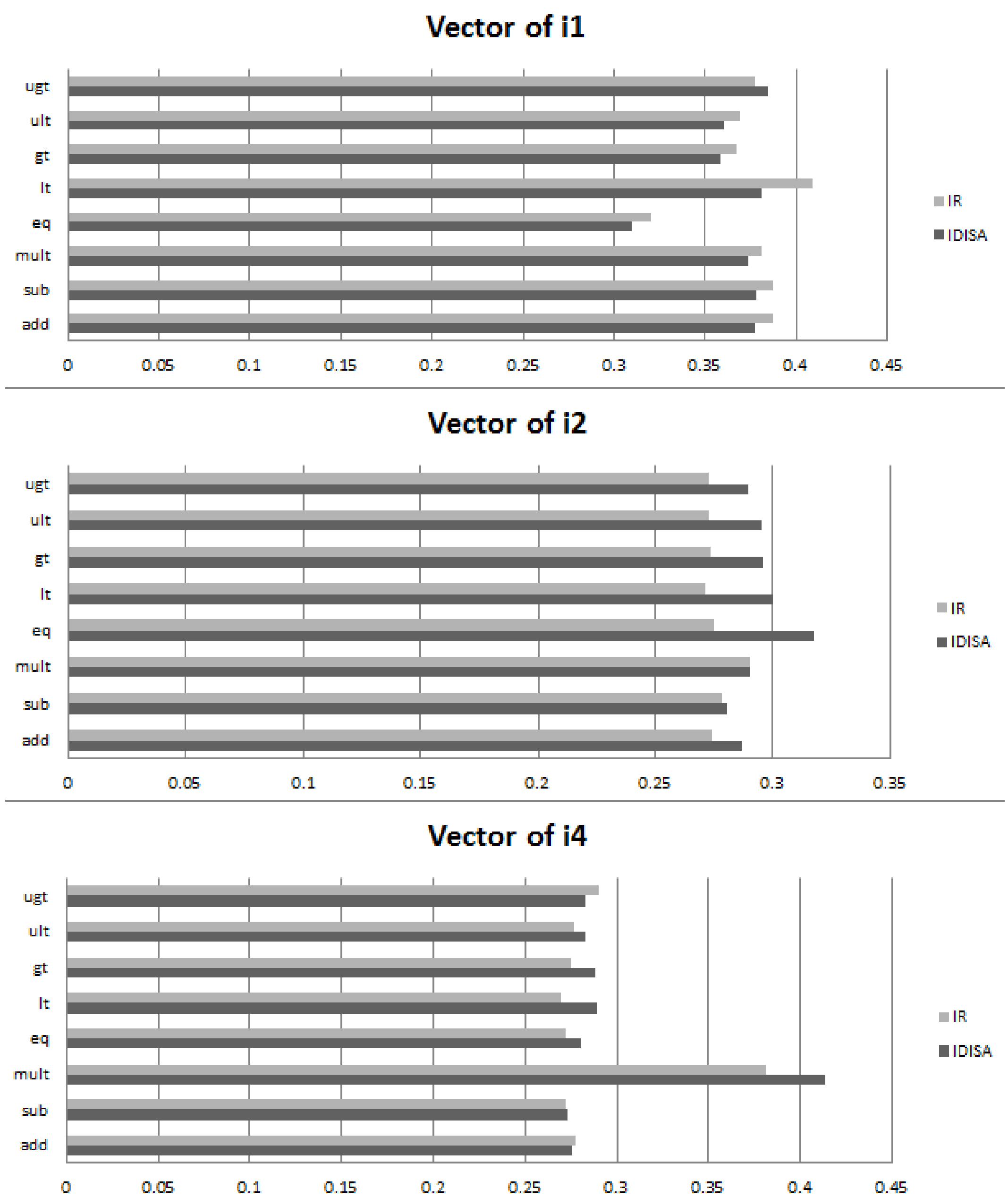

Figure 6.2: Reciprocal instruction throughput against IDISA library. IR and IDISA share almost identical throughput.

Figure 6.3: 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.

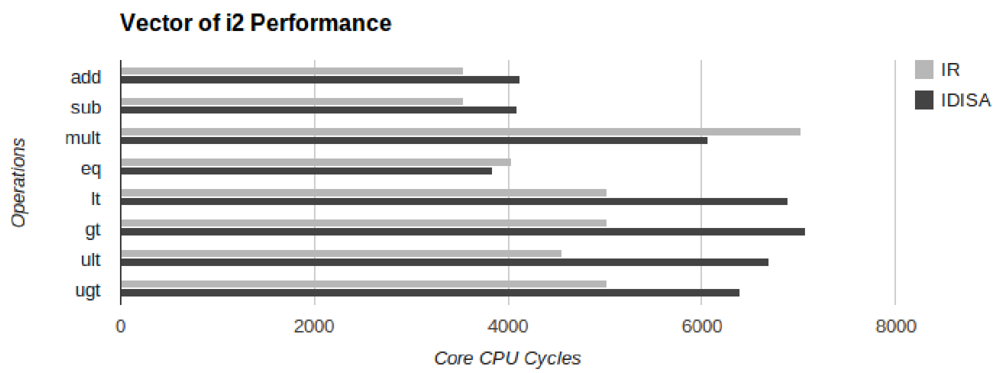

Figure 6.4: The same benchmark for $i2$ vectors with the instruction in a loop. Code in Figure 6.3 can be seen as the flattened version of this figure. We find IDISA here wins in the multiplication on $i2$, while IR wins it in Figure 6.3. Loop optimization should be responsible for it.

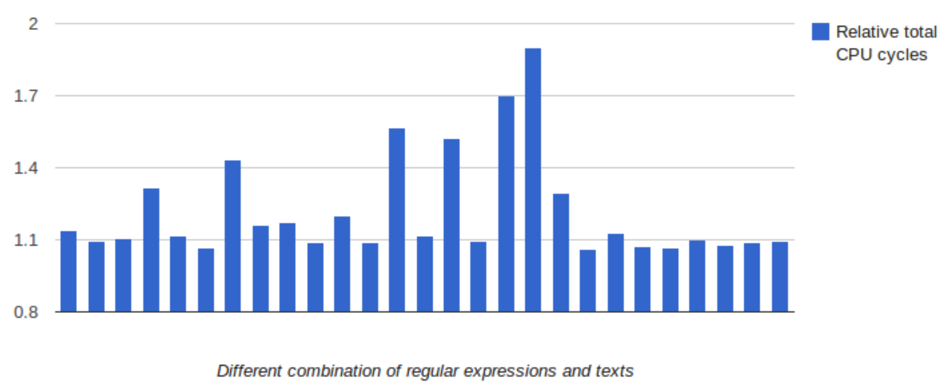

Figure 6.5: Performance of icgrep with long stream addition. This version of icgrep is based on $i128$ so long stream addition actually slows down the performance; but for wider SIMD register, it would get the same performance or even better. For different regular expressions, the portion of ”add with carry” code can be different, which explains the difference of the relative cycles across the x axis.

Chapter 7

Conclusion

In this thesis, we demonstrated that it is possible to extend LLVM type system to support Parabix technology. We have shown systematic support of the vector of $i2^k$ and support of critical Parabix operations in the target-independent IR library. We have also shown in one specific target: Intel X86, we can generate efficient native code

Bibliography

[1] LLVM Language Reference Manual . http://llvm.org/docs/LangRef.html. 3, 5

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

[3] Jinja documentation. http://jinja.pocoo.org/. 26

[4] The LLVM target-independent code generator. http://llvm.org/docs/CodeGenerator.html. 2, 3, 22, 25

[5] The parabix transposition. http://parabix.costar.sfu.ca/wiki/ParabixTransform. 10

[6] Python templating. https://wiki.python.org/moin/Templating. 26

[7] Quine McCluskey algorithm wikipedia. 13

[8] Test programs for measuring clock cycles and performance monitoring. 29

[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. 3

[10] 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. 10, 13, 14, 33

[11] Hua Huang. IDISA+: A portable model for high performance SIMD programming. Master's thesis, Simon Fraser University, December 2011. 27, 30

[12] E.D. Johnson. *Quine-McCluskey: A Computerized Approach to Boolean Algebraic Optimization*. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1981. 12

[13] 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. 13

[14] 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. 10, 19, 20