Technische Universiteit
Eindhoven
University of Technology

Department of Electrical Engineering
Electronic Systems Group

Code Generation for SIMD with Explicit Datapaths

Master Thesis

Guus Hendrikus Peter Leijsten

Committee:
Prof. dr. Henk Corporaal
Dr. ir. Pieter Cuijpers
Dr. ir. Roel Jordans
Dr. ir. Lech Jozwiak
Ir. Luc Waeijen

Version 1.0

Eindhoven, August 2017

Abstract

This thesis describes the design of a code generator for a configurable programmable platform implementing an ultra-wide *Single Instruction Multiple Data* (SIMD) architecture with explicit datapaths. The distinguishing characteristic of this architecture is a wide array of *Processing Elements* (PEs) that exploit parallelism by processing many operations concurrently. Therefore, a high throughput can be achieved at a low clock frequency and thus low voltage, thereby with a high energy efficiency.

A LLVM-based compiler that targets this architecture exists, but lacks support for configurable explicit datapaths. This compiler is implemented completely within LLVM because it provides auto-vectorization that does support a configurable array size

This work has a focus on extending the compiler with explicit bypassing capabilities. Explicit bypassing consists of two compiler optimizations, e.g. operand forwarding and dead result elimination. With operand forwarding, a value or result of an operation is forwarded from a pipeline stage to the *Instruction Decode* stage, thereby, bypassing the *Register File* (RF). When all consumers of a variable obtain it using forwarding, it is never read from the RF. Therefore, that variable does not need to be stored in a register file, since it is not read from it anyway. Dead result elimination (which is possible with explicit bypassing) consists of avoiding such redundant store accesses. With implicit bypassing the hardware performs bypasses automatically, while it is the compilers responsibility to find and allocate bypasses with explicit bypassing.

Compiling with explicit datapaths has been an active topic of research and several architectures face similar challenges. The *Transport Triggered Architecture* (TTA) effectively faces the same challenge, except that compilation for TTAs is even more challenging because the datapaths is exposed in the instruction set. What TTAs have in common with code generation for SIMD with explicit datapaths is that explicit reads and writes are explicitly stated in the instruction set.

Several approaches to support explicit datapaths for the target SIMD architecture within LLVM have been ranked on implementation effort and expected results. Subsequently, one approach has been implemented and the efficiency of the generated code was accessed based on simulation results.

With explicit bypassing, around 40% of the accesses can be avoided. Since the RF consumers around 35% of the total energy consumption and around 40% of the communication with the RF can be avoided, an estimated improvement of around 15% follows.

Keywords: Compilers, SIMD, LLVM, Embedded Systems, Power Efficiency, Explicit Bypassing

Preface

Before starting, I would like to take this opportunity to express my gratitude. I would like to thank all the people that are close to me for their continuous support and motivation to study electronic and computer systems. I want to thank my mom who is always there for me and my brother who always helps me in many ways. I would like to thank the ES-group for the project and for the many meetings that were hold over the duration of this project.

Furthermore, I would also like to thank the leading expert on computer architectures at the TU/e, professor dr. Henk Corporaal for guidance and introducing me to the field. Many thanks also go to my supervisor from Radbound university in Nijmegen, dr. ir. Roel Jordans for guidance and help on the compiler, and for introducing me to compilers.

Also, many thanks go to ir. Luc Waeijen who assisted me in many ways on technical aspects of this master project and to dr. ir. Lech Jozwiak, who always helps a lot by providing extremely useful feedback.

More thanks go to dr. ir. Pieter Cuijpers for joining my committee, and I would like to thank Liu Zhenyuan for his initial implementation of the compiler and Boyan Liang for his view on the hardware side of this project.

Finally, I would like to conclude the thanks with a quote, “Tell me and I forget, Teach me and I remember. Involve me and I learn.” - by Benjamin Franklin.

Contents

List of Abbreviations

ALU Arithmetic Logical Unit
AST Abstract Syntax Tree
BB Basic Block
CFG Control Flow Graph
CNN Convolutional Neural Networks
CP Control Processor
DAG Directed Acyclic Graph
DDG Data Dependence Graph
DLP Data Level Parallelism
ES group Electronic Systems Group
FIR Finite Impulse Response
FU Functional Unit
ID Instruction Decode
IF Instruction Fetch
ILP Instruction Level Parallelism
IMEM Instruction Memory
IR Intermediate Representation
ISA Instruction Set Architecture
I-type Immediate-type
J-type Jump-type
LSU Load Store Unit
N/A Not Applicable
PE Processing Element
RA Register Allocator/Allocation
RaR Read-after-Read
RaW Read-after-Write
RF Register File
RP Register Pressure
RISC Reduced Instruction Set Computer
R-type Register-type
SIMD Single Instruction Multiple Data
SSA Static Single Assignment
TTA Transport Triggered Architecture
TU/e Eindhoven University of Technology
VLIW Very Long Instruction Word
WaW Write-after-Write
WB Write Back

Chapter 1

Introduction

Embedded systems are everywhere, over ninety percent of all microprocessors are manufactured as components of embedded systems. Never have we had such growth in the use of such embedded devices. Nowadays, most people carry a mobile phone that is more powerful than the computer I played my first game on in '98. Many embedded systems, like mobile phones, have to run high-performance applications, like wireless signal processing and 3D vision processing [**?**], which could be made possible by powerful processors (ARM64 / AArch64) that run at high clock speeds. However, these kinds of mobile devices often have a limited energy source, and because they are often handheld devices heat produced by power dissipation is also of concern. As embedded applications become more and more complex and are adopting more sophisticated algorithms, the issues of their computing performance and energy efficiency become more and more serious. To address their issues dedicated processors are developed for different kinds of embedded applications, like mobile computing, health care, collision avoidance and assisted driving [**?**].

The dissertation by Dongrui et. al. [**?**] aims to address these concerns, by investigating a low energy configurable programmable platform implementing an ultra-wide *Single Instruction Multiple Data* (SIMD) architecture. In general, a wide SIMD architecture consists of a *Control Processor* (CP) that runs in parallel with a wide array of *Processing Elements* (PEs). The CP is responsible for scalar operations and the control flow, while the PE array executes a single instruction on multiple data. Because the PEs execute the same instruction in each cycle, the instruction fetch and decode can be shared among the PEs. Therefore, the energy consumed by these parts is distributed over multiple PEs and becomes negligible. For certain kernels, it can exploit parallelism by processing multiple operations in parallel instead of processing them sequentially. Therefore, the required throughput can be achieved at a substantially lower clock frequency and thus lower voltage, thereby greatly reducing energy consumption and power dissipation [**?**].

To program these kind of architectures, an architecture specific compiler has to be build. Compilers are indispensable for high-level language to executable code translation, but have also a significant role in the design of computer architectures. During the design phase of an architecture, one may want to see how efficient such design is or what impact certain design decisions have. To analyze how efficient applications can execute on a design, we need a compiler to translate an application to machine code, which is then used to simulate the execution of such architecture. The combination of a processor architecture, a high-level application code, and compiler for this architecture decide the quality of a resulting hardware/software system. Therefore, architecture design and compiler development go hand in hand.

The *Electronic Systems* (ES) group at the *Eindhoven University of Technology* (TU/e) is doing research in a wide SIMD with low energy features in order to achieve a programmable platform configurable for specific applications for a high energy efficiency [**?**]. The current compiler for this platform is developed completely within the LLVM framework, but does not generate code for explicit datapaths. However, there is an older version of the compiler, which we refer to as the

legacy compiler that does generate code for explicit datapaths. It uses LLVM's front-end with a custom back-end that has limited maintainability and is stuck to an old version of LLVM's front-end, therefore, not benefiting from developments in the field of compilers.

1.1 Motivation

This master thesis aims at completing the transition to LLVM such that the developed compiler supports all design options including explicit datapaths. We would highly benefit from having a fully functional SIMD compiler in LLVM. There are compiler developers working around the clock on many different architectures, e.g. xCORE[1] (multicore microcontrollers), AArch64 (mobile phones) and x86 (modern computers). With an LLVM-based compiler, we benefit from developments on LLVM and greatly improve maintainability of the compiler.

We will measure the efficiency of the generated code for a very low energy ultra wide SIMD architecture in terms of code quality and energy efficiency. A practical LLVM-based compiler is compared to a legacy custom build compiler and efficiency is assessed using handwritten assembly code references. We want to know to what degree low power this architecture truly is and see if we can make a successful transition to LLVM. Many companies have standardized to LLVM already, and we compare our LLVM compiler to a legacy custom build compiler.

This work also focusses on improving the energy efficiency by implementing a specific optimization technique to reduce communication with the *register file* (RF). The RF is one of the most power-hungry, and often used components in a processor.

We analyze the gain in energy efficiency by exploiting explicit datapaths. The SIMD architecture has busses that contain time-dependent values of results of the functional units. Accessing one of these busses is cheaper than accessing a register file in terms of energy. Therefore, we use communication involving these buses to decrease the traffic involving the register file. Thereby, improving energy efficiency.

1.2 Problem Statement

For the architecture at hand, several features were selected to be considered. Each combination of values for these features results in a different hardware configuration:

- **Processor pipelining** is the technique to split the task of a processor up in multiple steps. Because the processor works on different steps at the same time, more instruction can be executed and thereby increasing throughput. A four-stage or five-stage pipeline can be chosen.
- **Bit-width of the data** that the processor operates on is configurable to 32 or 16 bits. With smaller 16-bit you may further improve energy efficiency, but requires knowledge of the programmer and consideration during application development. The 32-bit data width may be sufficient for embedded applications but is still smaller than 64-bit architectures.
- **Extension of the *Instruction Set Architecture* (ISA)** has also been considered. For example, it may be beneficial to have a *Functional Unit* (FU) that can do multiply-accumulate instructions for, e.g. signal processing filters, like *Finite Impulse Response* (FIR) filters, linear algebra, like matrix multiplication, and *Convolutional Neural Networks* (CNN).
- **Explicit or implicit bypassing** can be implemented in the processor. These features reduce accesses to register files by result forwarding (which is discussed in Chapter 2.3),

[1] www.xmos.com/products/silicon

and dead result elimination with explicit bypassing. With implicit, also called automatic bypassing, dedicate hardware detects and exploits these bypasses. With explicit bypassing, it is the responsibility of the compiler to exploit them.

In general, it is desired to have a compiler that satisfies some basic requirements, (i) it should be easy to maintain, (ii) it should be easy to add other features and (iii) it should produce high-quality code. Of course, it should always generate a correct code.

The legacy compiler has some input language limitations, maintenance problems and does not always generate a correct code.

- **Input Language Limitations:** One needs to implement an application in OpenCL to get vectorized code. Moreover, it only supports a subset of the OpenCL language. Altogether, this puts the responsibility on the programmer which is something that we want to avoid. Furthermore, the generated code is not vectorized when C code is used as input language.
- **Maintenance:** The back-end of the legacy compiler is too custom and, maintainability would benefit from standardization. Namely, because it is difficult to implement new features and not all considered features have been developed.

The practical LLVM-based compiler has drastically improved maintainability and uses LLVM's auto-vectorizer to generate vector instructions. Compared with the old compiler, the new compiler is more flexible and supports a large number of input languages. The optimization passes supplied by LLVM also improve the quality of the generated code. Furthermore, developments on LLVM make it easy to update the compiler and can, therefore, benefit from developments in compiler technologies. However, the new compiler does not support all features. Namely, it can only generate code for a target machine with four pipeline stages, implicit datapaths, and a bit-width of 32 bits. We will maintain and add features to the new compiler with a focus on the exploration of explicit datapaths.

This master thesis aims at completing the transition to LLVM such that the developed compiler supports all design options including explicit datapaths

The main problem is "How to generate efficient code for SIMD that exploits explicit datapaths?".

1.3 Thesis Overview

The next chapter describes background information that will provide key information to this thesis, including a basic introduction to LLVM, an overview of the SIMD architecture and related work.

Chapter 3 discusses LLVM-based code generation for our target architecture with a focus on explicit datapaths, followed by an evaluation in Chapter 4. Future work is presented before concluding in Chapter 5.

Chapter 2

Background

This chapter will aim to cover background knowledge about the LLVM Framework and features of a wide SIMD. After that, the related work follows, which discusses other exposed or explicit datapath architectures, the legacy compiler and developments on the LLVM-based compiler for the target SIMD architecture.

2.1 LLVM Compiler Framework

The LLVM project was started in 2000 by Chris Lattner, as a research project at the University of Illinois with the goal of providing a modern, *Static Single Assignment* (SSA)-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. It was first released in 2003 and the project has grown rapidly since then. It has become popular amongst major companies, e.g. Google, Apple, and Sony, for its powerful multi-stage compilation strategy and outstanding extendability. LLVM is a collection of modular and reusable compiler and toolchain technologies. Generally, LLVM follows a 3-phase design, which is divided between a frontend, a code independent optimizer and a backend, illustrated in Figure 2.1.

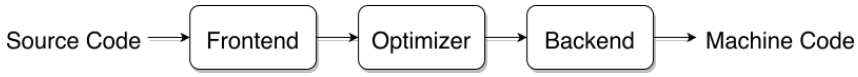


Figure 2.1: 3-phase design: frontend, optimizer and backend.

The frontend is responsible for translating code of an arbitrary programming language into LLVM's *Intermediate Representation* (IR) code. The LLVM instruction set represents a virtual architecture that captures the key operations of ordinary processors, but avoids machine specific constraints such as physical registers. Instead, it has an infinite amount of virtual registers in SSA form, which means that each virtual register is assigned only once and each use of a variable is dominated by that variable's definition. This simplifies the data flow optimizations because only a single definition can reach a particular use of a value, and to find that definition is trivial [**?**].

Figure 2.2 gives an overview of a frontend. The main task of the lexical analyzer is to read the input characters of the source program, group them into luxemes, and produce as output a sequence of tokens. These tokens are used by the parser for syntax analysis, where it is verified that the sequence of tokens can be reconstructed according to the syntax of the input language. The parser reports any syntax errors during this process and should be able to recover from the error in order to continue processing the rest of the program. The parser constructs a parse tree, and the semantic analyzer uses this parse tree to check for consistency with the language definition.

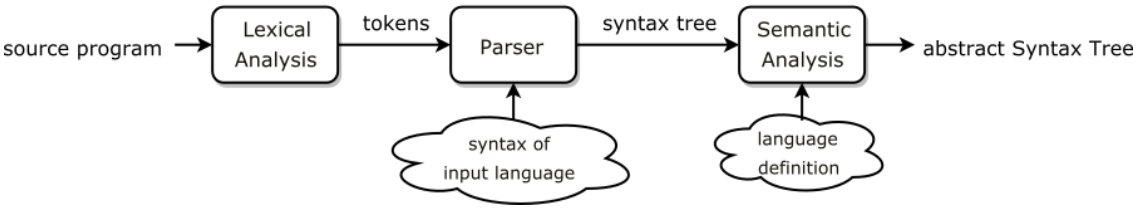


Figure 2.2: Overview of the components that the frontend compromises.

Type checking is also done during this stage, and the information is kept in a syntax tree. The result of these phases is an *Abstract Syntax Tree* (AST) of the program, which can be translated into three-address IR code.

Listing 2.1 shows an example C code where where two arguments are multiplied and added them together to a third argument. On the right-hand part of this listing one can see the output that is generated by the front-end. The notion of labels is already there which is similar to that of assembly code. The LLVM-IR code is in SSA form and consists of three-address operations. Here *nsw* indicates that the result is undefined in case of an overflow.

Listing 2.1: Fragment of C code with corresponding LLVM-IR.

```
int foo(int a, int b, int c)
{
  c += a*b;
  return c;
}
```

```
define i32 @foo(i32 %a, i32 %b,
                i32 %c) #0
entry:
  %mul = mul nsw i32 %b, %a
  %add = add nsw i32 %mul, %c
  ret i32 %add
}
```

The optimizer contains a collection of analysis and semantic-preserving transformations that can be used to optimize IR code. One of the advantages of LLVM is that when you build a new backend for any given processor architecture you immediately have access to all of these optimizations. Some optimization techniques are explained below which are explained in more detailed in other literature [**?**, Chapter 9].

- *Constant propagation:* computes for each point and each variable in the program, whether that variable has a unique constant value at that point. This can then be used to replace variable references with constant values.
- *Constant folding:* recognizes and evaluates constant expressions at compile time rather than runtime. For example, '*add* $1+2$' can be replaced by '3'. Statements like '*add* $1+2$' can be introduced by other optimizations, e.g. constant propagation.
- *Common sub-expression elimination:* recognizes that the same expression appears in more than one place, and that performance can be improved by transforming the code such that the expression appears in one only place.
- *Copy propagation:* replaces each target of a copy statement with that of the copied value. For example, when the copy statement $x = y$ is applied. Then the uses of x can be replaced by y. Some optimizations require that this optimization is performed afterward to clean up, e.g. common sub-expression elimination requires this pass to run afterward.
- *Dead code elimination:* removes code that does not affect the program's results. This avoids executing irrelevant operations and reduces the code size of a program.
- *Loop invariant code motion:* aims at moving code that is independent of the loop iteration out of the loop body. It does this by moving the loop independent statement above the loop,

saving it in a temporary variable, and use it in each iteration of the loop. Now the loop independent statement is computed only once instead of every iteration.

- *Function inlining:* verifies whether inlining functions in its callees gives a performance benefit. If doing this would give a performance benefit, it replaces the call to the function with the function body. This optimization often is useful for small functions because it reduces the overhead that is introduced when a function call is made, e.g. storing frame pointer, storing function parameters and jump to the code to where the function is defined.

The backend translates, according to a processor architecture, IR code to a target specific assembly language. It does this by going through a sequence of code generation stages, illustrated in Figure 2.3. The rectangular boxes indicate the data structure that is used by and produced by a given stage, and the name of each stage is denoted in a rectangular box with rounded corners. During this process, first, the IR code is lowered to a *Directed Acyclic Graph* (DAG) in which each node represents an instruction. However, for some architectures, not all data types and instructions are supported. For this reason, the DAG is legalized to something that is supported by the target architecture. Instruction selection maps each of the nodes onto machine nodes, by matching patterns. Then the DAG consists only of target specific machine instructions, in SSA form. Having naive machine instructions, the next step is to schedule them. The machine instructions are scheduled according to the resource information of the target processor and assign each instruction to a specific cycle. Now the instructions are represented in a list rather than a DAG, but still in SSA form. The *Register Allocator* (RA) then assigns physical registers to each of the virtual registers, now the list is not in SSA form.
The post-allocation pass can improve the schedule by taking physical registers and *Register Pressure* (RP), that is known at this point, into account. After that, some epilogue and prologue code may be inserted, for example, saving/restoring the caller/callee registers and reserving/destroying of the function's stack frame. Peephole optimizations are target specific improvements to the generated code. These optimizations deal with very specific optimizations that can only be done at the end of the process. Finally, the assembly printer prints the generated code to a file.

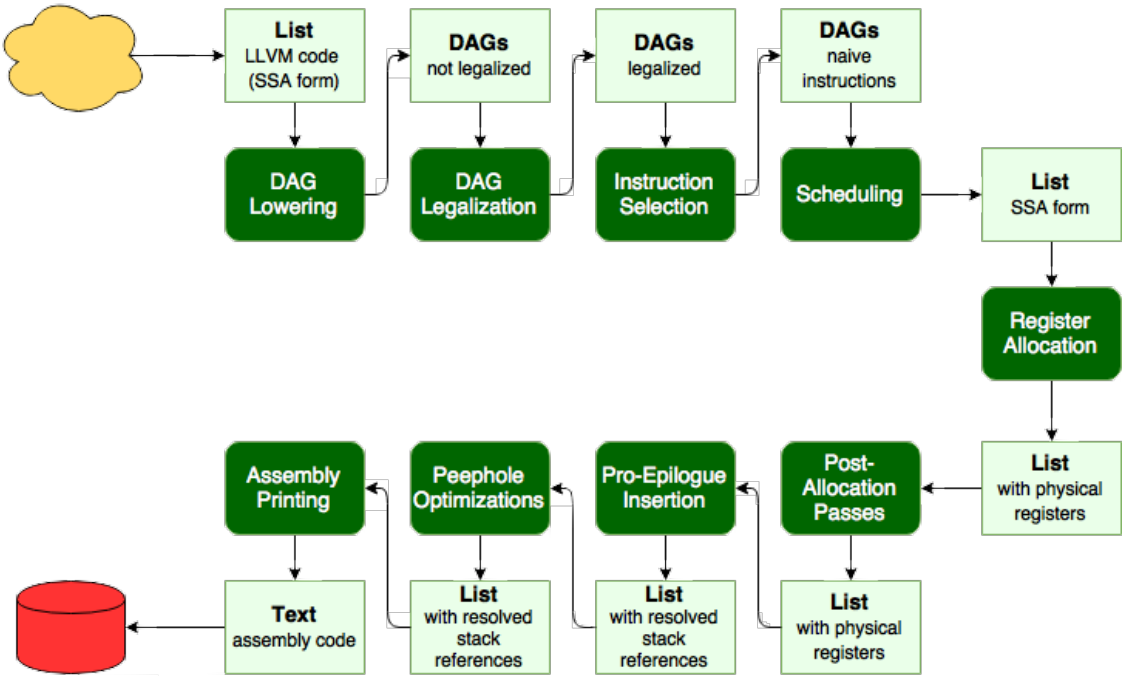


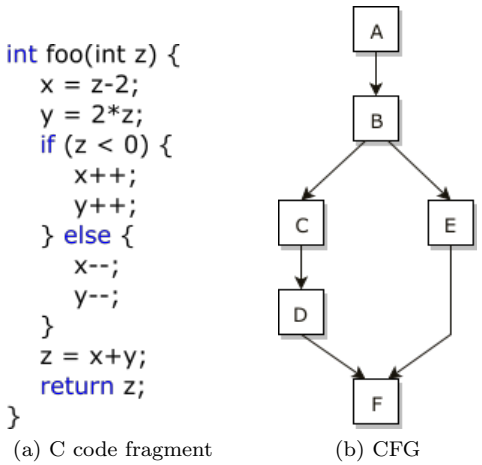
Figure 2.3: Code generation sequence, from LLVM code to assembly code.

2.2 Representation of Code

To analyze a code fragment, different representations can be explored. For instance, three-address IR code that is used by LLVM is conducive for further processing like optimizations and translations. Even further down the compilation line, there is assembly code which is basically object code, but in human readable form. However, to analyze a fragment of code, one does not always need this much detail. Moreover, having this much detail, sometimes makes analysis more difficult.

2.2.1 Control Flow

Control flow analysis is a code technique to analyze the control flow of a program. The control flow is expressed as a *Control Flow Graph* (CFG), which is a more abstract representation of code that uses a graph notation to show all paths that can be traversed through a program during its execution.



(a) C code fragment (b) CFG

Figure 2.4: Example CFG with C code.

Figure 2.4b shows a CFG for the code fragment in Figure 2.4a. In a CFG edges represent the control flow of the program and nodes represent basic blocks. A *Basic Block* (BB) is a sequence of instructions with no branches, except for entering and leaving the basic block.

2.2.2 Data Dependencies and Spilling

From a compilers perspective, certain operations address memory locations. A *store* operation accesses the memory to put the value of a register into memory at a certain location, addressable by its address. On the other hand, *load* operations load a value from memory at a certain location and puts that value in a register. These kind of operations are called memory operations. Other operations calculate a value and stores the result in a register. Operations that store the result in a register, or load a value from memory into a register, actually define that register. An example is given in the following code fragment:

```
lw   r2, r0, 4    # load from memory at location 0x4
lw   r3, r0, 5    # load from memory at location 0x5
mul  r3, r3, r2   # define r3 with r3 = r3 * r3
sw   r3, r0, 2    # store result at location 0x2
```

Before scheduling and register allocation, the sequence of instructions contain an unlimited number of virtual registers and the instructions are in a DAG that preserves all control flow and data dependencies. Data dependencies are ordering constraints that influence the order of execution. Typically, there are three kinds of data dependencies [**?**]:

1. There is a *Read-after-Write* (RaW) dependency, also called a *flow dependency* from operation a to operation b if a defines a register that may be used by b.
2. There is a *Write-after-Read* (WaR) dependency, also called an *anti-dependency* if for operations a and b when a uses a register that is redefined by b.
3. There is a *Write-after-Write* (WaW) dependency, or *output dependency* from operation a to operation b, if a defines a register that is redefined by b.

Flow dependencies are also known as *true dependencies* and anti and output dependencies as *false dependencies*, introduced by scheduling or register allocation. While in SSA form, each variable is defined exactly once, therefore, there are only true dependencies in that form. After scheduling and register allocation, where a physical register is assigned to each virtual register, multiple virtual registers may be assigned to one physical register. This is illustrated in Listing 2.2. This process introduces false dependencies and can often be resolved with renaming techniques [**?**, **?**].

Listing 2.2: Redefining physical registers by assigning them to multiple virtual registers.

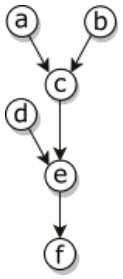
```
mul %v1, %src1, %src2
add %v2, %1,    %sum
mul %v3, %src3, %src4
add %v4, %v2,   v3
```

```
mul r1, r5, r6
add r2, r1, r21
mul r1, r7, r8   # redefines r1
add r2, r2, r1   # redefines r2
```

Figure 2.5b shows a data dependence graph corresponding to the code fragment in Figure 2.5a. In a data dependence graph nodes represent operations and the edges correspond to data dependencies.

```
(a) lw  r2, r5,  4
(b) lw  r3, r6,  6
(c) mul r3, r3,  r2
(d) lw  r4, r7,  0
(e) add r4, r3,  r4
(f) sw  r4, r11, 0
```

(a) Assembly fragment



(b) DDG

Figure 2.5: Example DDG with corresponding assembly code.

Sometimes there are not enough registers available to allocate physical rezgisters to all virtual registers because there are only a limited number of physical registers available. When the compiler runs out of registers to allocate, *spilling* may be necessary to free one or more registers by storing them on the stack. Consequently, it is required to retrieve them from the stack just before they are used.

2.3 Explicit Datapaths

Data goes through a particular path through the pipeline of a processor and typically, a pipeline is split up in multiple stages. With a pipelined processor, each of the stages are effectively working in parallel. Figure 2.6 illustrates hardware pipelining for a *Reduced Instruction Set Computer* (RISC) processor. The pipeline is split up in four or more stages, e.g. instruction fetch, instruction decode, execution and write-back stage. During the *instruction fetch* (IF stage) an instruction is loaded from *instruction memory* (IMEM), which is decoded in the *instruction decode* (ID) stage where the type of operation is determined by the opcode and operands are identified by their register addressing, and finally, the operation is executed and the result is written back to the register file.

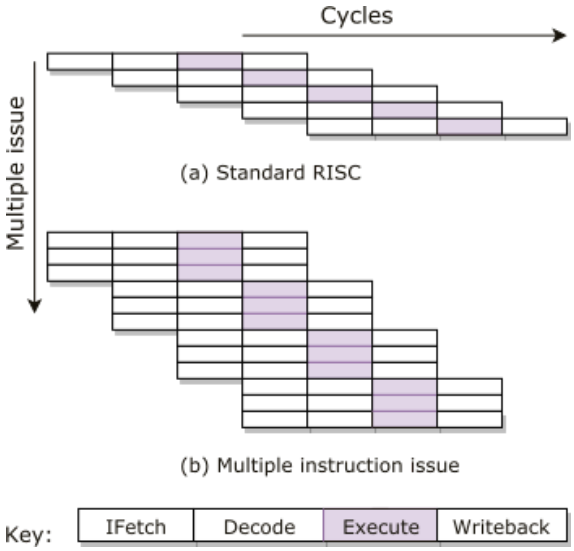


Figure 2.6: Pipelining and multiple instruction issue[**?**].

Operand forwarding: When bypassing is completely absent the result of an instruction can only be obtained after it is written back. On the other hand, with bypassing, a bypass network of wires and busses connects the execution and writeback stages back to the ID stage. These wires and busses may be used to forward a value to operands of another instruction. This way, the result of an instruction can be obtained before it has been written back to the register file. This behaviour is illustrated in Figure 2.7.

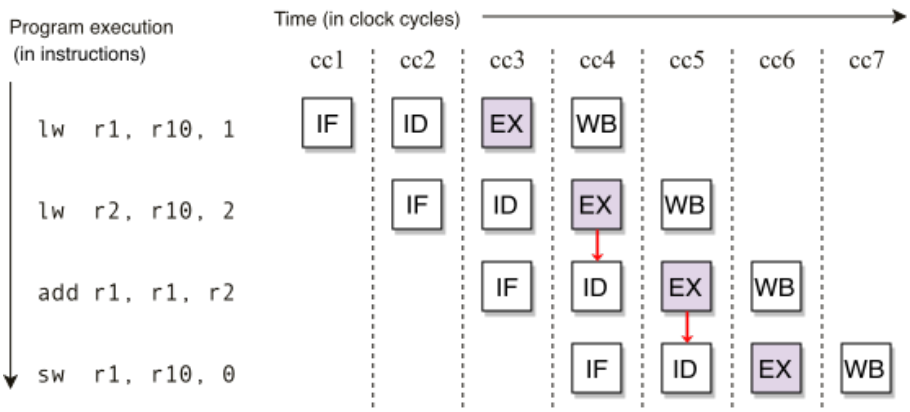


Figure 2.7: Instructions being executed using the single-cycle datapath, assuming pipelined execution.

Four consecutive instructions are executed of which the last two instructions have a RaW dependency with the instruction prior to it. As you can see in Figure 2.7 the instructions prior to said instructions are in the execution stage when their result is queried, namely, when said instructions are in the ID stage. So either the processor needs to be stalled for a cycle, such that the previous instruction is in the WB stage, or the result needs to be forwarded from the execution stage to the ID stage. Since insertion of stall cycles result in less efficient code, the result is forwarded using the bypass network (indicated with a vertical arrow from EX to ID).

Implicit Datapaths

With implicit bypassing (also called transparent bypassing), bypassing opportunities are detected and controlled by dedicated hardware on chip. However, doing this at run-time has some advantages and some disadvantages.

- **Advantage:** With bypassing, operand forwarding is possible which reduces the number of reads from the register file. This will reduce register file usage and may improve energy efficiency of the processor.
- **Disadvantage:** If you push the responsibility of identifying and exploiting forwarding to the hardware, it needs $2 \cdot d \cdot n$ comparators (where d is the number of stages between ID en WB and n the number of issue slots) to compare operands of an instruction to registers defined by previous instructions that are already in the pipeline. Altogether, this results in a more complex gate design and increased area.

It has been observed that variables in the register file are often transient, meaning that they last only for a short time. Reads from the register file of said variables can be avoided by forwarding which is shown in the previous paragraphs.

Now, we show that register writes of transient values can be avoided with explicit datapaths. The reason why this is only the case for explicit datapaths, and not with implicit bypassing, is that the hardware can only look at instructions that have already been executed, while the compiler can look at all instructions of a program.

Dead result elimination: If all consumers of a variable obtain it using forwarding, then it will never be read from the register file. Therefore, that variable does not need to be stored in a register file, since it is not read from it anyway. Hence, these obsolete stores may be avoided.

Explicit Datapaths

With explicit datapaths (sometimes referred to as exposed datapaths), bypass opportunities are detected and controlled by the compiler. Doing this at compile time rather than run-time has some advantages and some disadvantages.

- **Advantage:** With explicit bypassing, dead result eliminations is possible which reduces the number of writes to the register file. This will further reduce register file usage and may further improve energy efficiency of the processor.
- **Disadvantage:** The compiler needs to take datapaths and variables that are in the pipeline into consideration, because bypassing is explicit in the compiler. This results in a more complex compiler.

This project focusses on implementing these optimizations for a compiler that targets a wide SIMD architecture. The following section is devoted to explaining the target SIMD architecture.

2.4 SIMD Processor Architecture

The advantage of the SIMD architecture is that multiple operations are processed in parallel instead of sequentially. Therefore, the required throughput can be achieved at a substantially lower clock frequency and thus lower voltage, thereby greatly reducing voltage and thus energy consumption [**?**]. Furthermore, because each *Processing Element* (PE) executes the same instruction, the *Instruction Fetch* (IF) and *Instruction Decode* (ID) can be shared among the PEs. Therefore, the area and energy consumed by these parts is negligible.
A wide SIMD architecture [**?**] performs wide vector operations that exploit *Data Level Parallelism* (DLP) by executing the same instruction on multiple data simultaneously. Figure 2.8 shows a general overview of the SIMD processor. There is a *Control Processor* (CP) responsible for the control flow and scalar operations, and there is a wide array of PEs that execute in parallel with the CP, which are responsible for processing vector operations.

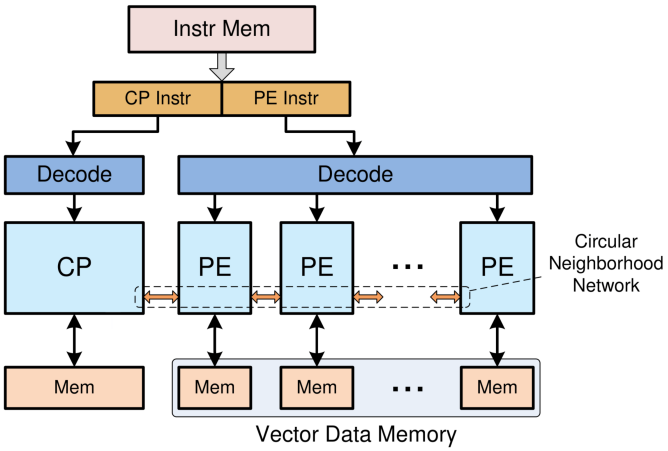


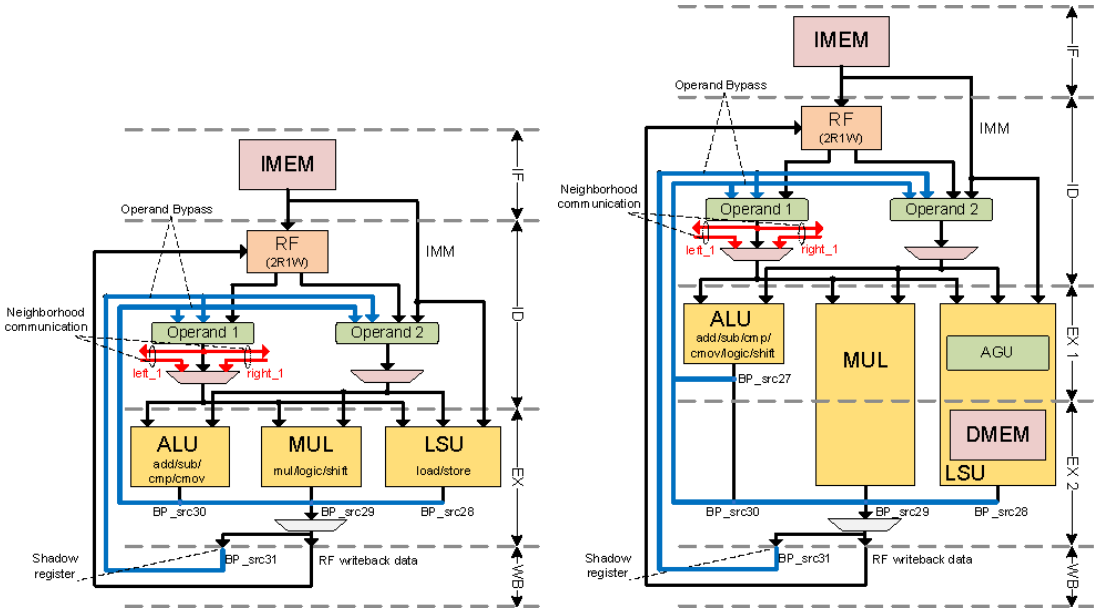
Figure 2.8: General overview of the wide SIMD architecture.

A SIMD instruction word encodes two sub instructions, hence it resembles a 2-issue VLIW instruction word. Each sub instruction can generally be any instruction defined in the *Instruction Set Architecture* (ISA). The proposed architecture has a *Reduced Instruction Set Computer* (RISC) ISA and is divided up into three categories of instructions. The list of instructions can be found in Appendix B. In general, a instruction has three operands of which one destination register and two source operands. A Register-type (R-type) instruciton takes two registers as source operands. Instructions that take one register and a constant immediate as operands are called Immediate-type (I-type) instructions. Finally, the control flow can be manipulated by modifying the program counter using Jump-type (J-type) instructions, which can be executed only by the CP.

In order to support a configurable number of PE elements, a neighbourhood network topology is used for its scalability. With a circular neighbourhood network topology, the connection between the first and last PE does not introduce extra long wires, because the PEs can be placed in a circular manner [**?**].

2.4.1 Processor Pipeline and Datapath

Generally, each processing unit (CP or PE) has its own registers and three functional units, i.e. ALU, MUL and LSU.
The instruction pipeline is divided up into four or five stages. Top down, there is an IF-stage, an ID-stage, one or two execution stages and a Write Back (WB) stage, like traditional lockstep RISC processors. The architecture shown in Figure 2.9a has four stages while the architecture shown in Figure 2.9b has five stages.

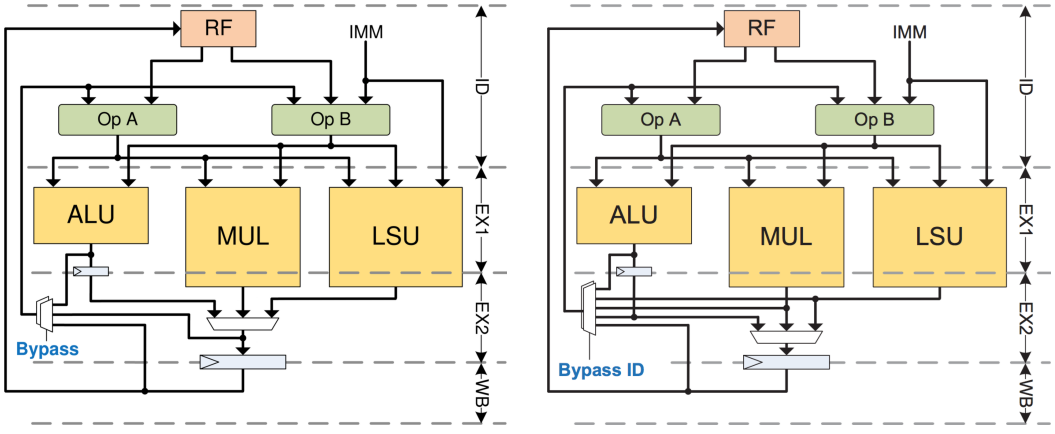


(a) 4-stage pipeline processor overview.

(b) 5-stage pipeline processor overview.

Figure 2.9: Bypassing network differences between the four-stage and the five-stage pipeline.

The neighbourhood communication network is implemented by overriding `operand 1` in the ID-stage. Depending on the decoded instruction, data is selected from another (neighbouring) processing unit, from the local RF or from the bypass network. Each FU has private input registers, which keep the result at the output of a compute unit valid as long as no new operation or input is assigned to it [**?**]. This *operand isolation* reduces toggling in the FUs, and creates extra opportunities for bypassing. The outputs can be forwarded using the bypass network to operands of another instruction.
The SIMD can be configured to have either implicit or explicit bypassing. With implicit bypassing, also called transparent bypassing it is the hardware's responsibility to handle bypassing. With



(a) Datapath with implicit bypassing.

(b) Datapath with explicit bypassing.

Figure 2.10: Bypassing network differences between implicit and explicit bypassing.

explicit bypassing, on the other hand, the bypasses are encoded in the instructions, and it is thus the compiler's responsibility to handle bypassing.
Explicit bypassing has as main advantage that certain writes to a register can be avoided. Namely, when the result of an instruction is bypassed and not used anywhere else, it does not need to be stored in a register (*dead result elimination*, Chapter 2.3). Avoiding writes to the RF reduces the total energy consumption. Since there are many register files in a wide SIMD, reducing the energy consumption of the register file has a large impact on the overall energy consumption [**?**]. Because of this, reducing the register file's energy consumption is of great importance. Furthermore, the explicit datapath shown in Figure 2.10b has two extra sources compared to the implicit datapath in Figure 2.10a. These additional bypass sources increase the chance that a result is being bypassed. In the explicit bypassing version, bypassing sources are directly accessible by the instruction. This is done by reserving part of the RF address space for the bypass sources. The disadvantage of this is that the register index space is reduced, however, the instruction format does not require changes in order to encode that an operand comes from the bypass network. (Chapter 3.4.1).
The total number of registers grows linearly with the number of PEs because each processor has 32 registers. Hence, a wide SIMD has in total many registers that consume a considerable amount of energy. Namely around 34.6% of the total energy consumption [**?**]. Hence, optimizing RF accesses is of great importance.

2.5 Related Work

There is related work for compilers that target SIMD architectures, in particular, a compiler has been developed, referred to as legacy compiler. Furthermore, compiling with explicit datapaths has been an active research topic for other architectures.

2.5.1 Other Exposed/Explicit Datapath Architectures

Compiling with explicit datapaths has been an active topic of research. Several architectures that face similar challenges have been investigated.

Transport Triggered Architectures

One of the main works that was investigated is the *Transport Triggered Architecture* (TTA) which has been developed in the MOVE project [**?**]. For TTA architectures, instructions consist of transports that specify the datapath. The difference between TTAs and traditional operation triggered architectures is that TTA are transport driven, hence its name.
Figure 2.11 illustrates how function units and register files are connected to an interconnect network which are programmed by instructions. Programming a TTA consists of moving operands to the input registers of a FU.

```
add r1, r2, r3   # define r1 with r1 = r2 + r3
sub r4, r2, 6    # define r4 with r4 = r2 - 6
sw  r4, r1, 0    # store result at address r1
```

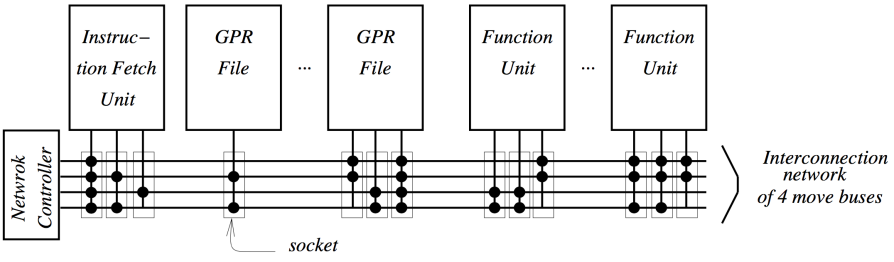


Figure 2.11: General structure of a TTA with interconnect network driven by data transports.

First step is to translate each n-operand m-result operation into $n+m$ moves (so called n *operand moves* and m *result moves*). `O1` and `O2` are input ports of the FU and `R` indicates the output (result) of a FU.

```
r2 -> O1add ; r3 -> O2add ; Radd -> r1
r2 -> O1sub ; r6 -> O2sub ; Rsub -> r4
r1 -> O1sw  ; r4 -> O2sw
```

Let us assume that there are two FUs named `alu1` and `alu2` for ALU operations, and one FU named `ls` for load-store operations. The suffixes '`alu1`', '`alu2`', and '`ls`' indicate the FU on which the operation is executed. If the above fragment would be scheduled such that the distance between the final operand and a corresponding result move should be at least the latency of the FU. Then the following (TTA assembly) code may be obtained:

```
r2 -> O1add.alu1 ; r3 -> O2add.alu1 ; r2 -> O1sub.alu2 ; r6 -> O2sub.alu2
Radd.alu1 -> r1  ; Rsub.alu2 -> r4
r1 -> O1sw.ls    ; r4 -> O2sw.ls
```

Bypassing: The outputs of the add and subtract operations can be directly moved to the load-store unit. This reduces the schedule by one cycle, however, the number of moves does not change.

```
r2 -> O1add.alu1; r3 -> O2add.alu1; r2 -> O1sub.alu2    ; r6 -> O2sub.alu2
Radd.alu1 -> r1 ; Rsub.alu2 -> r4 ; Radd.alu1 -> O1sw.ls; Rsub.alu2 -> O2sw.ls
```

Dead result move elimination: Next it may occur that the values in `r1` and `r2` are not live anymore because they are used only once. In that case corresponding moves can be skipped. This gives the following schedule:

```
r2 -> O1add.alu1 ; r3 -> O2add.alu1 ; r2 -> O1sub.alu2 ; r6 -> O2sub.alu2
Radd.alu1 -> O1st.ls ; Rsub.alu2 -> O2st.ls
```

The optimizations that is discussed here for TTA also apply on a wide SIMD architecture, which is discussed in Section 2.3. However, their approach can not be used because the SIMD architecture is an operation triggered architecture, data transports are a given. More information on explicit bypassing for TTAs can be found in the work of Hoogerbrugge et. al. [**?**, **?**].

Very Long Instruction Word Architectures

The *Very Long Instruction Word* (VLIW) architecture is designed to optimize *Instruction Level Parallelism* (ILP) by executing multiple instructions in parallel. Although RISC architectures take advantage of temporal parallelism by using hardware pipelining (explained in Chapter 2.3),

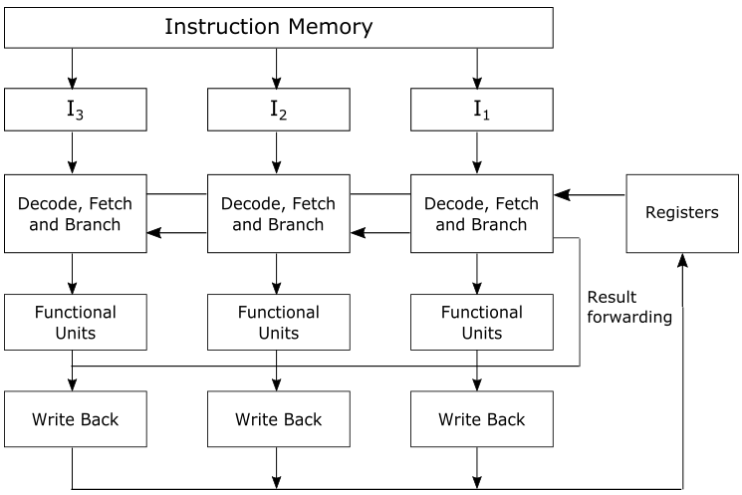


Figure 2.12: Block diagram of VLIW architecture with bypassing.

VLIW architectures take advantage of spacial parallelism by using multiple functional units to execute several operations concurrently.
Figure 2.12 shows a generic block diagram of a VLIW machine that has multiple instruction issues (three in this example), and each issue has its own decode stage and functional units. The figure also illustrates how the bypass network connects the outputs of the FUs back to the ID stage, allowing results to be forwarded. For VLIW architectures, the complexity of the bypass network grows linearly with the size of the instruction word. Furthermore, traditionally the register file requires n write and $2n$ read ports, where n is the length of the instruction word. This may require a more hungry register file as the power efficiency degrades when increasing the number of ports on the register file [**?**]. However, this requirement can be relaxed by clustering, where each cluster may have a dedicated register file, which is often done in modern VLIW architectures.

A reconfigurable VLIW architecture is developed in the ρ-VEX project at the University of Technology in Delft [**?**]. This architecture also considers explicit bypassing and other configurable design options, e.g. configurable issue-width, functional units and bit-width of the data.
However, their implementation can not be used because (i) they are using a gcc based compiler instead of a LLVM based compiler and (ii) not a lot of implementation details have been given in their paper.

2.5.2 Legacy compiler

S. Dongrui et al. proposed this processor architecture [**?**] while attaining his PhD [**?**]. The target wide SIMD architecture was design during that time and the compiler that was developed then has many issues. To start, the compiler has a LLVM frontend and a mostly in C++ implemented backend which is not according to LLVM standard. Therefore, it requires a frontend based on an old version of LLVM and since its front-end evolves over time, it is preferable to be able to update this to the newest version.

With C code applications as input language, the compiler can only compile to non-vectorized (CP only) instructions. To generate vectorized code an effort had been made to take OpenCL code as input language [**?**]. However, only a subset of OpenCL is supported. Moreover, compilation does often generate incorrect code, or none at all.

In his work he shows that the register file is one of the most frequently used, and most power-hungry components in a processor. Therefore, he introduced an explicit datapath to further improve energy efficiency on this extremely energy efficient processor architecture. Moreover, efficient code generation for such architecture is key to achieve high energy efficiency for the whole processor. For this reason, the efficiency of the generated code is improved by standardizing the compiler to a frequently used framework. Namely, by standardizing to LLVM. This way we may benefit from developments in the field of compilers and from improvements made to the LLVM framework.

2.5.3 Basic Compiler Design

This section explains an initial design of a SIMD back-end designed within LLVM developed by L. Zhenyuan [**?**]. He started to build a back-end in LLVM for the same purpose, but with a different goal, namely how to generate efficient vectorized code within LLVM. In order to support vector instructions, LLVM's auto-vectorizer has been used and intermediate code optimization passes have been implemented to generate SIMD specific intrinsics, that are, in turn, transformed to vector instructions and shuffle operations.
His work gives a basic design for this back-end that targets a wide SIMD architecture. The compiler that he developed is taken as a starting point, and is maintained, improved and extended which is discussed in later sections. When building a back-end in LLVM, first the instructions and registers have to be defined. Then illegal operations and types can be converted to legal ones. Then

during instruction selection, LLVM knows how to match DAG nodes to known instructions. The supported instructions and registers for this architecture are defined first. To support some special features of this architecture, custom passes are added to this back-end, which will be described in detail, see Chapter 3.1, where our contributions to this work are discussed. Furthermore, an assembler and a linker have been implemented as separate projects, which will briefly be discussed here as well.

Supported Instructions

All the instructions which have been defined in this compiler are listed in this part, with the corresponding brief explanations. The details of ISA can be found in Appendix **??**.

- **Arithmetic and Logic Instructions:** `add`, `addi`, `sub`, `muli`, `mulu`, `mului`, `or`, `ori`, `and`, `andi`, `xor`, `xori`, `sll`, `slli`, `sra`, `srai`, `srl` and `srli`.
  The instruction with suffix "I" is used to handle immediate value operand, which is referred to as I-type instructions. The suffix "U" means it is used for unsigned values. For others, the two input operands are both registers, which are commonly referred to as R-type instructions.

- **Flag Set Instructions:** `sfeq`, `sfne`, `sfles`, `sflts`, `sfges`, `sfgts`, `sfleu`, `sfltu`, `sfgeu` and `sfgtu`.
  The flag set instructions are used for comparison. If it is true, the flag register is set. The suffix "S" represents a signed value, while the suffix "U" represents an unsigned value.

- **Conditional Move:** `cmov`.
  This is usually used with flag set instructions. If the flag is set, the value in the input operand is moved to the output operand.

- **Immediate Extension:** `simm` and `zimm`.
  These two instructions are used to extend the immediate value from 8 bits to 26 bits. The maximum immediate value could be $2^{26} - 1$, instead of $2^8 - 1$. The details of these two instructions will be discussed in Section 3.2.4. In addition, a larger immediate value requires a sequence of instructions to be executed.

- **Conditional Branch:** `bf` and `bnf`.
  These two instructions also work with flag set instructions. By using the branch in- structions, the program can branch to the target address if the flag is set (with BF) or not set (with BNF).

- **Jump Instructions:** `j`, `jr`, `jal` and `jalr`.
  The difference with the conditional branch instructions is that the jump does not need to check the flag register, which is normally used during function call and return.

Register Configuration

There are two main register classes. Although each PE has its own register file in our architecture, it is not necessary to define a specific register class for every PE register file. One reason is the size of the PE array is configurable and the number of the vector register classes cannot be dynamic. Another reason is, the vector array actually is a single issue slot, which executes the same instruction for all PEs. It is sufficient to define one register class for the entire PE array. Each vector register defined in the back-end represents a line of registers in the PE array.
Table 2.1 shows the register configurations of both CP and PE-Array. Both `r0` and `v0` are connected to ground and contain constant value zero. `v1` is a special register, which contains the index of local PE. Note that, there are two stack pointers, `r11` and `v11`. As mentioned before, considering there are two separated data memory, a double frame stack is needed for the CP and PE-Array to access their memories directly and reduce the expensive data communication. Therefore, two stack pointers are defined to point to the top of scalar and vector frame stacks respectively. The

Table 2.1: Registers configuration.

Scalar Register	Vector Register	Purpose
r0	v0	Constant value zero
N/A	v1	Constant PE index
r3∼r4	v3∼v4	Return registers
r5∼*r8*	v5∼v8	Argument passing registers
r9	N/A	Link register
r10	N/A	Frame pointer
r11	v11	Stack pointer
r1, r2 and r12∼r31	v2, v9∼v10 and v12∼v31	General purpose registers

details of separate frame stacks is not described here, but can be found in L. Zhenyuans thesis [**?**, Chapter 4].

Vectorizer

Support for LLVM's Auto-Vectorizer has been developed by L. Zhenyuan. He added support for this to our back-end during his time [**?**, Chapter 5]. He defined a couple of patterns to match and IR level transformations that transform loops to vector instructions and shuffle operations. Furthermore, he added a cost function to decide whether to vectorize a given loop automatically. However, sometimes the auto-vectorizer fails to vectorize a simple loop. In those cases, pragmas are manually inserted directly before a loop in the C code. Clang uses these pragmas for making decisions on whether or not to vectorize a given loop. In the end, this may give better results, as will be shown in Chapter 4.
Supported vector types are: `v1i32`, `v2i32`, `v4i32`, `v8i32`, `v16i32`, `v32i32`, `v64i32`, `v128i32`, `v256i32`, `v512i32`, `v1024i32` and `v2048i32`.
The actual legal vector type should be equal to or smaller than the `PENum`. `PENum` is a variable defined in the back-end, which can be configured using `pe-num` flag. The default value is 8, in which case the legal vector type can only be `v1i32`, `v2i32`, `v4i32` and `v8i32`.

Linker and Assembler

An assembler has been implemented that can parse assembly code and translate it into binary code. Furthermore, a linker has been developed which takes one or more binary files and combines them into a single executable file. These have been developed as separate projects and within the LLVM framework. Unfortunately, the linker still has some problems that need to be resolved before it can be used. Therefore, a custom linker is chosen instead of the standard linker supplied by LLVM. We would benefit from using LLVM's linker because the custom linker can work only on a single file. Unfortunately, this linker can not be used yet because it is a work in progress.

Chapter 3

LLVM-based Compiler for SIMD

This chapter focusses on each of the code generation phases and what our contributions are to this compiler. This chapter describes how a compiler without support for explicit datapaths (which is described in Section 2.5.3) is maintained and extended to a compiler with support for explicit datapaths.

3.1 Back-end Code Generation

This section briefly discusses each of the code generation stages, which includes custom passes and standard passes supplied by the LLVM framework. Before generating code, it uses LLVM's front-end to translate a high-level language to an intermediate representation, called LLVM-IR which can be further optimized and used as input for our back-end. The code generation passes in the back-end specific compiler can be categorized in three major categories.
Firstly, the top row in Figure 3.1 shows passes that work on creating a schedule. The second row illustrates a sequence of compilation passes that do back-end specific transformations. Here special features that our architecture has are taken care of. At last, with an LLVM internal representation of back-end specific code for the target architecture, it emits assembly or ELF object code (depicted in the rightmost part of the image) that the processor can understand.

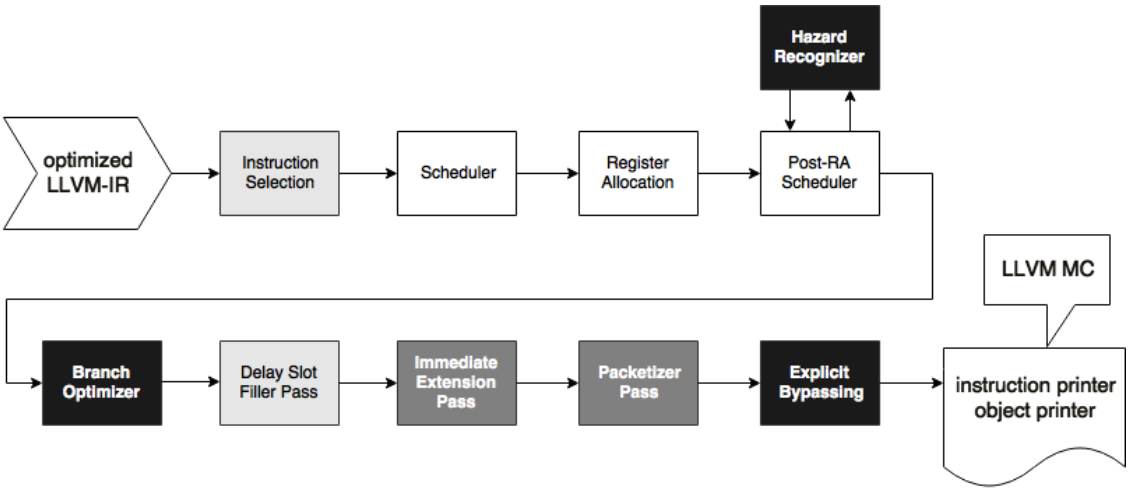


Figure 3.1: Overview of the phases that the back-end is comprised of.

Note that the passes are marked from light to dark, where a black background means that the pass was created during this project and are the main contributions of this work. Dark grey indicates

that a pass is extended or improved, light grey indicates the pass has hardly changed and a white background means that the pass is supplied by LLVM.

- **Instruction Selection:** Uses a DAG, called `SelectionDAGs`, an abstraction for code representation suitable for phases ranging from instruction selection, legalization to lowering. Instruction selection is implemented in `SimdISelDAGToDAG`, which is derived from `SelectionDAGISel` and consists of a bunch of transformations to transform specific instructions into instruction that are supported by our architecture. For example, transforming operations on immediate values with a value higher than one byte is partially implemented here. Lowering nodes of a DAG is done in `SimdISelLowering`, which is derived from `TargetLowering`. There the SIMD intrinsics and ISD instructions are lowered to *Machine Instructions* (MI) and sequences of MIs.

- **Instruction Scheduler:** MIScheduler supplied by LLVM
  MIScheduler is an instruction scheduler which supports VLIW scheduling. Considering there are two issue slots in this architecture, a VLIW scheduler would meet our requirements very well. One and two schedulers are defined for four-stage pipeline and five-stage pipeline respectively. The four-stage pipeline scheduler is the default one while the five-stage pipeline has the default one, and an additional post-register allocation scheduler.

- **Register Allocation:** Greedy Allocator supplied by LLVM
  The Greedy allocator is a default allocator in LLVM. Since there is no specific requirement to register allocation, for now, the default allocator is suitable. Apart from the default register allocator, there are more register allocators to choose from and, it is possible to implement a custom register allocator.

- **Post-Register Allocation Scheduler:** A second scheduling pass which is only needed when generating code for a five-stage pipeline configuration, and is disabled otherwise. The post-RA scheduler uses a hazard recognizer which decides whether to prefer certain instruction over other instructions to detect and resolve RaW hazards.

- **Hazard Recognizer:** Consecutive instructions may have hazards with the five-stage pipeline configured. That is when an instruction uses an operand defined in the previous cycle but has a latency of two cycles. For this thesis, we have implemented LLVM's `ScoreboardHazardRecognizer` to detect and resolve these hazards.

- **Branch Optimizer:** An inefficiency in the generated code was observed where two jump operations control the work flow, but one jump is not necessary when it jumps to the successor that follows immediately.

- **Delay Slot Filler:** For this architecture, jump and branch instructions modify the program counter (PC) during the instruction decode stage. At that point, the instruction that follows the jump has already been fetched. The slot that follows a jump or branch instruction is called a delay slot. This pass aims to utilize delay slots by filling them with useful instructions.

- **Immediate Extension:** In principle, immediate operations have a one-byte immediate operand. However, sometimes you may want to use a larger immediate. This pass allows us to use a larger immediate operand using `zimm` and `simm` instruction. These instructions have an 18-bit operand that allows for larger constants to be used.

- **Packetizer:** This pass creates VLIW bundles that consist of a scalar and a vector instruction, by consulting whether an issue slot is available on which the operation can execute. This pass implements VLIW packetizer supplied by the LLVM framework.

- **Explicit Bypassing:** This pass exploits explicit datapaths in the generated code by using the bypass network. This is developed as a post-processing pass, but can be replaced by any of the approaches discussed in Chapter 3.4.

Table 3.1: Relations between architecture design features and code generation phases.

Feature	Code Gen. Pass	Explanation
Hardware Pipeline	Delay Slot Filler	This pass utilizes the delay slots (which is a product lockstep pipelining).
	Hazard Recognizer	This pass is necessary for a five-stage pipeline configuration where not all instructions have a single cycle latency.
Bit-width	Immediate Extension	Lowering immediate operands is done differently for different data bit-width.
ISA extension	Instruction Selection & Instruction Lowering	New instructions need to be described in the back-end.
Explicit datapaths	Explicit Bypassing	This optional pass is developed to exploit explicit datapaths. It can be enabled using compiler flag `-explicit`.

- **Instruction Printer:** MC is a sub-project of LLVM, which uses `MCInst` to represent an instruction which is different from the code generators notion of a `MachineInstr`. MC is used during the last code generation stage when printing the instructions to a given output format. Printing for different output formats is further divided in binary format and SIMD assembly language in XAS-format.

The following sections give a more elaborate discussion on each custom pass and describe their relation to each other. However, before that have a look at Table 3.1 which gives an overview of the passes responsible for each of the features from Chapter 1.2.

3.2 Custom Passes

This section describes the custom passes in more detail. Dependencies and relations to other passes are described. Furthermore, it will discuss the entire toolchain, which includes an assembler, a linker, and a simulator.

3.2.1 Hazard Recognizer

When generating code for a four-stage pipeline configuration, all instructions have a latency of one cycle. In that case, hazard recognition is not necessary. In order to support code generation for a five-stage pipeline, a hazard recognizer has been developed. The hazard recognizer is used by the post-RA scheduler to determine whether two consecutive instructions can be scheduled after each other.

```
addi r13, r0, 14
addi r12, r0, 10
mul  r2,  r5, r13   # latency=2
mul  r3,  r6, r12   # latency=2
add  r2,  r2, r3    # RaW dependency
```

It detects hazards by considering whether an operation has a RaW dependency with instruction that came prior to it. A true hazard is when the instruction that came prior to it has a RaW dependency and a latency of more than one cycle. The *post-register allocation* (Post-RA) scheduler does a linear scan through the list of operation and queries the hazard recognizer whether an instruction can be scheduled. When it detects a hazard, it will consider other instructions that are ready to be scheduled, and if there are none available without a hazard, it will insert a no-op and the processor will be stalled for a cycle.

3.2.2 Branch Optimizer

There were many double branch instructions in loop structures. Preliminary benchmarks already showed that a double branch instruction does not always help. For example, consider the assembly code fragments in Listing 3.1 and Listing 3.2. Note that at this point, the delay slots that always follow after a jump or branch instruction are still absent because the delay slot filler pass has not run yet.

Listing 3.1: Fragment of assembly code to illustrate the first and second cases that are considered by the branch optimization pass.

```
$BB1:
    addi   r1, r1, 1
    sfltsi r1, 64
    bf     $BB1
    j      $BB2
$BB2:      ...
```

```
$BB1:
    addi   r1, r1, 1
    sfltsi r1, 64
    bf     $BB1
$BB2:      ...
           ...
```

1. The first case is where a block ends with a jump instruction to the block that immediately follows. The example shows a loop in which a counter is incremented until it reaches sixty-four. As long as the counter is less than that, the flag will be true and the branch is executed. When the counter increases, at some point the condition breaks and the flag becomes false. The program counter then points to the first instruction after the branch instruction, which is a jump instruction. However, the jump goes to the successor that immediately follows. When the jump is removed, the first instruction after the branch instruction is still the successor block that immediately follows. Hence the behaviour without that jump is identical and the superfluous jump can be removed.

2. The second case has a branch instruction to the block that immediately follows and a jump to somewhere else. The first step then is to reverse the branch condition. This can be achieved by changing a `bf` (branch flag) instruction into a `bnf` (branch not flag) and vice versa. After the branch is reversed, the situation becomes a jump instruction to the block that immediately follows and a branch instruction to somewhere else.

Listing 3.2: Fragment of assembly code to illustrate the third case that is covered by the branch optimization pass.

```
$BB2: ...
    sf condition  # set-flag
    bf $BB3
    j  $BB2
$BB3: ...
```

```
$BB2: ...
    sf   condition  # set-flag
    bnf  $BB2
$BB3: ...
      ...
```

When a (double) jump intruction(s) jumps to a successor that immediately follows, the branch optimizer can always remove one jump with these cases.

3.2.3 Delay Slot Filler

During the execution of a conditional branch or jump instruction the *program counter* (PC) is modified while the instruction is in the *instruction decode* (ID) stage. A side product of lockstep pipelining which is introduced in Chapter 2.3, is that when a jump instruction is being decoded in the ID stage, the next instruction with $PC = PC + 4$ has already been fetched from IMEM before the jump is executed. Therefore, the instruction that is followed by the jump instruction is executed presumptuously, which is referred to as a delay slot.
In order assure correct behaviour this pass intentionally places a no-op after each jump or branch instruction. However, sometimes it can do better. Namely, it can use an instruction from before

the jump instead of a no-op. This pass performs a backwards search to look at the two instructions before the jump, and the instruction that comes after the jump, which are referred to as $prev_1$, $prev_2$, and $next$ respectively. When the backwards search does not fill the delay slot it intentionally insert a no-op, and if there is a vector operation that comes after the delay slot, it also inserts a vector-nop, so that the packetizer does not bundle it with the delay slot later on.

Listing 3.3: Fragment of assembly code to illustrate behaviour of the delay slot filler for case one.

```
$BB1:
    v.add r3, r6, r14
    v.add r4, r5, r12
    j $BB1
    nop
```

```
$BB1:
    v.add r3, r6, r14
    j $BB1
    nop
    v.add r4, r5, r12
```

1. The first case is when the previous two instructions are both vector instructions. In that case, the vector instruction prior to the jump instruction can be moved to after it. Now the vector instruction that remains before the jump ($prev_1$) gets bundled with the jump instruction. If there was originally a scalar instruction after the jump instruction, it could get bundled with the vector instruction that is moved by this pass, later on by the packetizer. Therefore, it inserts a no-op to ensure the delay slot.

2. When the instruction before the jump ($prev_1$) is a scalar instruction with no dependencies to the jump itself, it can moved to the delay slot. If a bundled instruction was moved, then it is done. Otherwise, if there are two vector operations prior to the jump instruction, it can move one of them to after the jump, thereby, fully utilizing the delay slot.

Listing 3.4: Fragment of assembly code to illustrate behaviour of the delay slot filler for case two.

```
$BB1:
    v.add r3, r6, r14
    v.add r4, r5, r12
    add   r3, r3, r1
    j $BB1
    nop
```

```
$BB0_1:
    v.add r3, r6, r14
    j $BB1
    add   r3, r3, r1
    v.add r4, r5, r12
```

The two cases covered by this pass are illustrated in Listing 3.3 and 3.4. However, many delay slots are still not being utilized. Extending this pass such that more delay slots may be utilized will be added to future work.

3.2.4 Immediate Extention

Most operations in Appendix B.2 have as last operand, a one byte immediate. However, sometimes one may need larger numbers. Therefore, constants are lowered during instruction selection. The following three cases are given:

1. When the immediate can be expressed with one byte it is trivial. It does not need to change anything.

2. When the immediate value is larger than one byte, but can be expressed with 26 bits, it adds a `zimm` or `simm` operation in front of it. These operations have a 18 bit immediate that represent the upper 18 bits for the operation that follows.

```
simm 3           # 3 << 8 = 768
addi r3, r5, 12  # r3 = r5 + 768 + 12
```

3. If the immediate requires more than 26 bits, it requires a couple of instructions to be added in order to put the immediate value in a register. Firstly, the upper 6 bits go to a register and are shifted all the way to the left. Subsequently, the lower 26 bits are added to it using the previous cases.

```
add  r3, r0, 2     # upper 6 bits of the immediate
slli r3, r3, 26    # 2 << 26
zimm 3             # 3 << 8, upper 18 bits
addi r3, r3, 12    # lower 8 bit
```

`ImmExtension` is a class that is derived from `MachineFunctionPass` and adds a `zimm` or `simm` when necessary. Pseudo code for this algorithm can be found in L. Zhenyuan his thesis [**?**, Appendix B].

The contribution of this work is that `SimdISelDAGToDAG` (instruction selection) is extended such that a larger range of immediate values is supported. Namely, from 26 to 32 bits. Furthermore, a bug that was found and resolved in the part that inserts `simm` operations. One may use even operations with more than 32 bits since carry-using operations can be selected. These operations have three operands: The first two are the normal LHS and RHS, and the third is the input carry flag. The operations can then be chained together for adding and subtracting arbitrarily large values.

3.2.5 Packetizer

Using a packetizer transforms a sequential list of mixed scalar and vector operations into VLIW instructions that contain one scalar and one vector instruction. It does this by using *VLIWPacketizerList* from the LLVM framework. It searches for packets by going in a top-down approach through the list of operations until the end of the machine function is reached. It aims at filling all operation slots of an instruction, in our case a scalar and a vector operation. If an operation is encountered of a slot which is already full, it ends the packetized instruction and it proceeds to the next packet.

Listing 3.5: Illustration of how a list of mixed scalar and vector operations are transformed into 2-issue instructions.

```
v.addi r2,  r0,  a
add    r11, r10, r0
v.addi r3,  r0,  b
v.lw   r2,  r2,  0
v.lw   r3,  r3,  0
v.addi r11, r11, 4
v.mul  r2,  r3,  r2
v.addi r3,  r0,  c
v.sw   r3,  r2,  0
lw     r10, r11, 0
jr r9
addi   r11, r11, 4
```

```
add  r11, r10, r0 || v.addi r2,  r0,  a
                  || v.addi r3,  r0,  b
                  || v.lw   r2,  r2,  0
                  || v.lw   r3,  r3,  0
                  || v.addi r11, r11, 4
                  || v.mul  r2,  r3,  r2
                  || v.addi r3,  r0,  c
lw   r10, r11, 0  || v.sw   r3,  r2,  0
jr r9             || v.nop
addi r11, r11, 4  || v.nop
```

Listing 3.5 illustrates the transformation performed by the packetizer. The first two operations get put together in a packet by filling both the scalar and the vector slot. Then the vector operations are put in their own packet because the vector slot is already full. The load word operation is put with the last vector operation, thereby, fully utilizing the packet and the last two scalar operations get their own packet as well.
No contributions have been made to this pass, however, the packetizer is actually used to resolve an issue with the assembler. Without these modifications, each packet may have either one or two operations. However, this is modified to always fill a packet with a no-ops or vector no-ops when it is not full. The assembler translates the operations to binary code and when the VLIW

instructions are not full, it will insert only a sub-instruction, which makes it difficult to determine where the next instruction starts.

3.2.6 Explicit Bypassing

This pass exploits the bypass network in an explicit manner. Result forwarding and dead result elimination are performed on a generated code. Currently, it does this as a post-processing step, but it may be moved to somewhere else in the compilation chain. In general, the information of which operations reside in the pipeline at a point in time is needed to decide which results can be forwarded. Therefore, the behaviour of the bypass network is modelled at compile-time. The model is then used to decide whether a certain operand of an instruction may be bypassed. When an operand is bypassed, the liveliness information of the register that is bypassed is used to decide whether a certain write access to the RF is still needed. Effectively, if the variable that was bypassed is dead after a use (denoted by a register *kill*) it does not need to be stored anymore, since it will not be used later on.

Listing 3.6: Fragment of assembly code to illustrate operand forwarding and dead result elimination. Appendix A shows pseudo code for this pass.

```
lw  r1, r10, 1
lw  r2, r10, 2
mul r1, r1,  r2
sw  r1, r10, 0
```

```
lw  r1,  r10, 1
lw  --,  r10, 2
mul --,  r1,  LSU
sw  MUL, r10, 0
```

The assembly code in the above example starts with loading two values from memory. Subsequently, the values are multiplied and the result is stored back to memory. The second load produces a result that is immediately used. Therefore, it can be forwarded using operand forwarding (which is discussed in Chapter 2.3). This is encoded with `LSU`, because loads are executed by the *load store unit* (LSU). Similarly, the result of the multiplication is immediately used and forwarded. In this case it is encoded with `MUL`, denoting the functional unit that executes multiplications.
The example in Listing 3.6 is a self-contained assembly code fragment, so the result of these instructions are not used outside of what you can see. Hence, each result has exactly one use, and is dead after that use. Therefore, the variables that are bypassed will not be read from the RF, because they are forwarded from the bypass network instead. According to dead result elimination (introduced in Chapter 2.3) these obsolete stores can be removed, which is encoded using '`--`'. When an instruction has this as destination, the write enable is put to low when that instruction is in the writeback stage, and the result of that instruction is not written back to the RF. Reducing communication with the RF leads to an energy efficienty improvement, however, the variable is only available for as long as it resides in the pipeline.

3.3 Source-level Linker

Figure 3.2 shows a process to do compilation and simulate the output of the compiler. The resulting assembly code is simulated in order to verify the correctness of a benchmark (C file). The simulation generates a directory with the memory dumps after running the program. It also produces statistics that indicates how often a certain line is executed, and we can deduce from the statistics file how many memory reads and writes, how many register read/writes, and how many bypassed reads and writes there are.

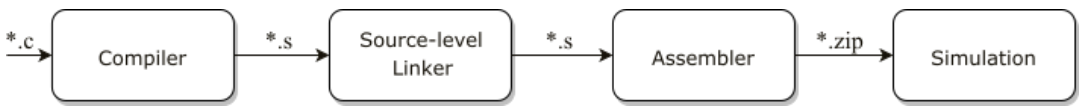


Figure 3.2: Workflow of simulation.

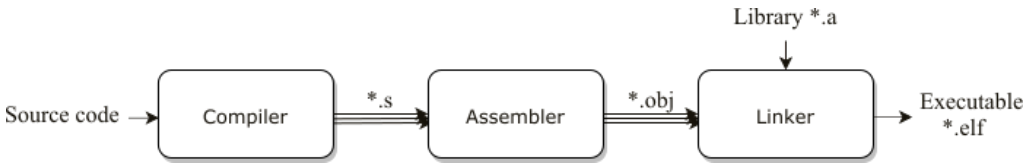


Figure 3.3: Standard linking process.

Doing the compilation and preprocessing steps from the compiler results in unlinked assembly code. Here symbols are not resolved yet, and it does not automatically generate a *_start* function. The source-level linker resolves the symbols and inserts a *_start* function in which the stack is initialized. However, the source-level linker works only for a single input file, therefore, all benchmarks are implemented in a single C file. The assembler and the simulator from the legacy toolchain are used and the source-level linker is necessary for the assembler to work.
Figure 3.3 illustrates the standard linking process when using the LLVM tools to do everything up to simulation. Other students have implemented an assembler and a linker within the LLVM framework. The assembler mainly consists of a parser that parses assembly code, and it uses LLVM-MC to form instructions to print them and the LLVM linker is developed within LLD. However, the new linker and assembler can not be used yet because it is still a work in progress.

3.4 Exploiting Explicit Datapaths

This section describes the implementation and design decisions of the explicit datapath implementation on the LLVM-based compiler.

3.4.1 Design Decisions

First, let us list the design decisions. The first design decision is how to encode explicit datapaths such that the hardware knows about bypasses. Then there is the manner of when in the compilation chain does it actually encode these bypasses.

Encoding bypasses

The first design decision is how to encode that an operand is forwarded. There are two ways to encode that a source operand comes from the bypass network:

i. Add bits to each instruction to specify that a certain operand comes from the bypass network. It requires three bits per source operands because there are five different bypass sources with a five-stage pipeline. Each instruction may have up to two source operands that can be bypassed. Therefore, this approach requires six additional bits to be added to each instruction.

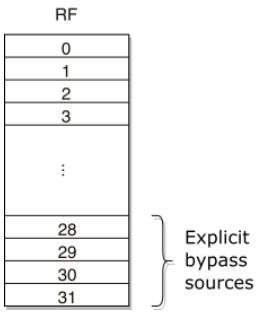


Figure 3.4: Illustration of reserving a portion of the register index space to encode explicit datapaths.

ii. Use register index space to indicate that an operand comes from the bypass network. This requires at most five registers to be reserved because there are at most five bypass sources.

The approach in which the register index space is used to encode that an operand is bypassed was chosen because the hardware description of the architecture corresponds to that, and because this does not require any modification to the instruction format.
The other approach would be beneficial since it does not reduce the register file index space such that limitations to high register pressure can be avoided.

When and how to allocate explicit bypasses

When to actually do this can be categorized in three approaches. Trivially, it is not possible to do explicit bypass allocation before scheduling, because one needs a schedule to allocate explicit bypasses. (i) Ideally it would be done before register allocation (Figure 3.5a) where there are still virtual registers. The number of virtual registers reduces with each store that is avoided. Thereby, reducing register pressure and relaxing the job for the register allocator.
However, when the register allocator runs out of registers and inserts spill code between two instructions that were already bypassed, this bypass may not be valid anymore and may even require an additional register to undo that bypass.
(ii) Alternatively it can be done after register allocation. However, the physical registers that are freed by dead result elimination are not exploited when doing this after RA. Because, if spill code was necessary it has already been inserted during register allocation. Therefore, after the explicit bypasses have been allocated, the resulting assembly code may have redundant spill code and does not gain from reduced register pressure. With this approach it does not matter whether to do scheduling or RA first, which is illustrated in Figure 3.5b.
(iii) Finally, it is also possible to do scheduling and register allocation in one go with constraint programming. However, this problem is NP-complete and may take an unbounded amount of time.

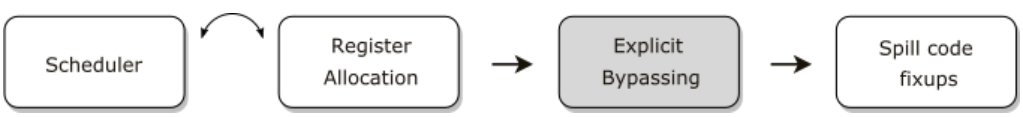
To summarize, ideally the explicit datapaths are exploited before register allocation to gain from reduced register pressure, but it is extremely difficult to insert spill code. So alternatively, it can be done after register allocation which may potentially have redundant spill code. For the later approach, an effort to clean up redundant spill code may be desired for more efficient code.

How to exploit explicit register allocation is a very broad question, in fact this is the main goal of this assignment. Let us split up in two categories of approaches:

i. One way to exploit explicit datapaths is to group instructions close to their use. Moreover, if an instruction is strictly adjacent to its use, it may immediately be bypassed.

ii. Another way to exploit explicit datapaths is by going through a basic block (a code sequence with no branches in except to the entry and no branches out except at the exit) and allocate

(a) Explicit bypassing before RA.



(b) Explicit bypassing after RA.

Figure 3.5: When to apply explicit bypasses.

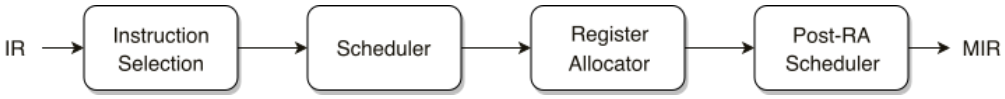


Figure 3.6: Phase ordering problem.

explicit bypasses as a post-processing step, in the sense that this is done when the schedule does not change anymore.

The approach in the first category where instructions are grouped together has a problem. When a bypass is done early on, it needs to be verified if it is still valid after a change to the schedule has been made. In the compilation chain (Figure 3.6) the order in which instructions are executed is first decided by the general scheduler. The register allocator may insert spill code at any place in the schedule. After that, the order may change again by the post-RA scheduler or by a custom pass.

Now let us discuss the second approach that works on a basic block. The assumption in this approach is that the order in which instructions are executed is set and does not change anymore. Therefore, each bypass that is applied stays valid. The problem that remains is when there are multiple branches to the basic block. With multiple branches in, the state of the pipeline and which bypasses may be allocated can be different depending on which branch is taken. This problem is referred to as the join problem which is illustrated by example in Listing 3.7. Depending on which branch is taken the value of register `r7` may be forwarded from `ALU` when coming from `%if`, or from `MUL` when coming from `%else` which leads to an ambiguous pipeline state. Thus, the use of `r7` in `%end` is forwarded from the bypass network, but is ambiguous and this together with being bypassed is not allowed for explicit datapaths. Therefore, such ambiguous behaviour should either be avoided or fixed, such that it is absent in the generated code.

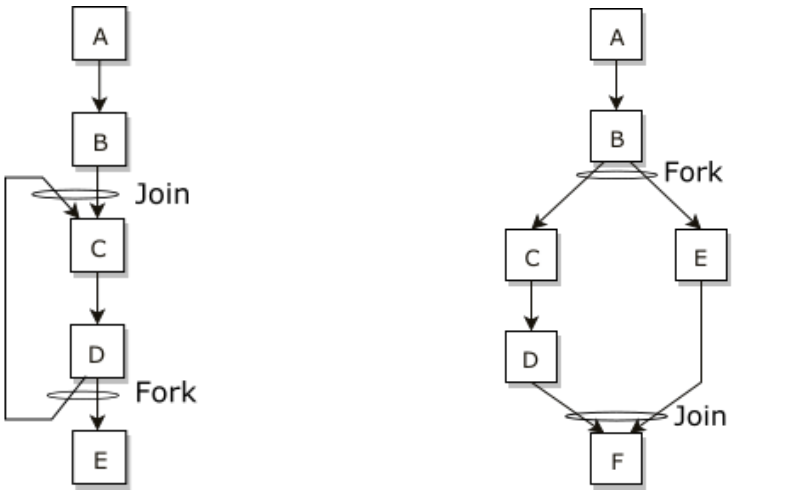
Listing 3.7: Fragment of C code with corresponding assembly, to illustrate the join problem.

```
int foo(int a, int b, int c)
{
  if (a > b)
    c += 5;
  else
    c = c * 3;
  c = c - a;
  return c;
}
```

```
$foo:             # a = r5, b = r6 and c = r7
  sfles r5, r6
  bf    $else
  nop
$if:
  j     $end
  add   r7, r7, 5    # r7 in ALU
$else:
  mul   r7, r7, 3    # r7 in MUL
$end:
  sub   r3, r7, r5   # r7 from MUL or ALU?
```

Bypassing with a limited scope of only a basic block requires multiple no-ops to be inserted on the end of each block, such that the pipeline state is flushed and no ambiguous behaviour occurs. Going from intra- to an inter basic block scope the ambiguous behaviour may be fixed using a technique which is discussed in the next section. With a function-level scope the pipeline state needs to be flushed only on a function call, because many instructions will be executed before returning. No-ops are not required because the prologue and epilogue insertions guarantee unambiguous behaviour between function calls. Going to a module-level scope would even allow bypasses from one function to another. However, because of time limitations, and because this only increases the bypasses with just a little bit, a module-level scope has not been implemented.

To conclude, the first approach discussed above may improve utilization of the bypass network, but can not be used as a stand-alone approach, since it does not solve the join problem. While the second approach works on a single basic block, it can easily be extended to a larger scope (for example, a function-level scope or a module-level scope). The following sections will show how the join problem can be tackled when a larger scope is considered.



(a) Fork-join in a loop.

(b) Fork-join in an if-else statement.

Figure 3.7: Fork-join illustration in CFGs.

3.4.2 Join Problem

The fork-join model typically branches off (fork) execution at a designated point in the program, and joins (merge) at a subsequent point to resume execution. When a value is defined in between the fork and join points, and is forwarded to after the join point, it may be defined in one branch, but not or differently in the other branch/branches. When a bypass is encoded in an instruction, it is statically defined, in the sense that whatever branch is taken the value that is forwarded should always come from the bypass source that is encoded in the instruction. For example, take the following assembly code fragment, whatever path is taken to reach this point, the first source operand is always taken from the `ALU`, and the second source operand is taken from `MUL` unit.

```
$join-point:
   add r1, ALU, MUL
```

Figure 3.7 shows two example CFGs with a join point. The following assembly code listings will illustrate how ambiguous bypasses may alter behaviour of the program. After that, a way to resolve such ambiguous behaviour is discussed.

```
$A:
   ⋮
$B:
   nop              || v.addi r9,  r0,  32
   nop              || v.lw   r2,  r9,  0
   nop              || v.lw   r3,  r9,  1
   addi r3, r0, 16  || v.lw   r4,  r9,  2
   ori  r4, r0, 0   || v.addi r10, r0,  64
$C:
   lw   r6, r3, 3   || v.add  r12, CP,  r0
   lw   r6, r3, 2   || v.add  r13, CP,  r0
   lw   r6, r3, 1   || v.add  r14, CP,  r0
$D:
   nop              || v.mul  r12, r4,  r12
   nop              || v.mul  r13, r3,  r13
   addi r4, r4, 0   || v.add  r9,  CP,  r10
   addi r4, r4, 4   || v.add  r12, r12, r13
   sfne r4, 12      || v.mul  r14, r2,  r14
   bf   $C          || v.add  r14, r14, r12
   addi r3, r3, 32  || v.sw   r14, r9,  0
$E: ⋮
```

The assembly code fragment above for the CFG in Figure 3.7a shows a 3-by-3 matrix multiplication benchmark. In basic block `B`, each load operations loads a row from the vector memory. Then, each PE has an entire column stored in the RF. In each loop iteration (`CD`), one row of values is loaded from CP memory and communicated to the PE elements by basic block `C`. Subsequently, in block `D` each element of a row from scalar memory is multiplied with a value from a column. The results of the multiplications are added together, thereby, forming a row of the resulting matrix, which is then stored back to vector memory.
When using the bypass network to forward a value from one basic block to another basic block, it is required that said value comes from the same bypass source regardless of the predecessor that is executed before it. For the matrix multiplication example, when bypassing the occurrences of `r3`, the value may be forwarded from `ALU` when predecessor `D` is executed, or from bypass source `WB` when predecessor `B` is executed. Note that this gives a conflict because `r3` does not come from the same bypass source in each of the predecessors. Therefore, a fix is required that puts the value in the desired bypass source.
Considering that there can be any nummer of predecessors, the heuristic developed during this project takes the most frequently executed predecessor block as reference, and fixes all other predecessor to match. In this example, the loop body (`CD`) is executed more often than the loop header (`B`). Therefore, block `B` is modified such that it matches block `D`. To achieve this it can either swap the last two scalar operations in `B`. However, a more generic approach would be to insert an instruction at the end of `B` to put the value in the correct bypass source.

```
$A:
   ⋮
$B:
   sfles r5, r6
   bf    $E
$C:
   ⋮
$D:
   j     $F
   addi  r7, r7, 5
$E:
   sub   r7, r7, r8
   mul   r7, r7, 3
$F:
   add   r3, r7, r5
```

The assembly code fragment above corresponds to the CFG in Figure 3.7b. It illustrates another example where a conflict may occur when forwarding a value over the join point to block `F`. In predecessor block `D`, the value in `r7` can be obtained by forwarding the result produced by the `ALU`. However, when predecessor block `E` is executed the value of `r7` may be forwarded from `MUL` instead. This conflict can be resolved by the same generic approach, namely, by inserting an operation at the end of either predecessor of `F` such that the value of `r7` may be forwarded using an identical bypass source, regardless of which predecessor is executed.

Resolving Conflicts

When an operand can be bypassed, but has an ambiguous pipeline state, a simple correction is usually sufficient to make it unambiguous such that it can be bypassed. Ambiguous behaviour in the pipeline may occur when bypassing over a join point (a basic block that has multiple predecessors). The bypasses that are allocated when assuming the pipeline state of the most frequently executed predecessor is taken as reference, and when a different bypass would be allocated when assuming the pipeline state of another predecessor block, the other predecessor block is modified such that it matches the bypass allocation given by the reference block. Four cases are considered:

1. If the reference forwards a value using `ALU`, but is not in `ALU` when another predecessor is executed. Then an operation is inserted to the end of these other predecessors which adds zero to the value that is bypassed.

2. If the reference forwards a value using `MUL`, but is not in `MUL` when another predecessor is executed. Then an operation is inserted to the end of these other predecessors which multiplies the forwarded value with one.

3. If the reference forwards a value using `LSU`, but comes from another bypass source by other predecessors. Then the load is copied, and inserted at the end of these other predecessors.

4. When it is not possible to bypass the value from the reference block, but the other block or blocks do require a value to be bypassed. Then a nop is added to the end of the other predecessor(s) such that the value will be written back before it is used.

For a five-stage pipeline configuration an additional no-op is added when a multiplication or a load was inserted at the end of a block, so that the result is ready when it is used in the first operation of a basic block. This concludes all cases, however, the `WB` has not been considered. When the reference block forwards a value from `WB` over a join point, it requires an instruction to be inserted at a non-trivial position in the basic block, depending on where the uses occur.
For this reason, the current implementation does not allow forwarding using `WB` over a join point. It achieves this by performing a check when the WB source is considered in a block that has multiple predecessors. It checks whether the instruction that defines the value and the instruction that it is forwarded to are in the same basic block, and whether the defining instruction dominates its use. If both conditions hold, then we are still good, since it is just being forwarded within a basic block and not over a join point.
This concludes the implementation of resolving ambiguous pipeline states. The next chapter is devoted to assessing the quality of the generated code by analyzing the simulation results.

Chapter 4

Evaluation

In order to evaluate the performance of the compiler, a set of benchmarks is used. The assembly code generated by the two compilers will be compared on 'code quality' and on the number of register reads and writes it can avoid by exploiting explicit datapaths. To verify correctness of the generated code, the simulation output of the code generated by the LLVM compiler and of the code produced by the legacy compiler are compared to a reference. The references are generated using GCC[1] and the result of GCC is an executable that prints the reference output upon execution. This reference output is compared to the simulation results and if they match, the compiler can generate a correct code.
To compare the two compilers, scalar-only code is considered such that the comparison is fair. Then the auto-vectorizer is enabled for further analysis which results in more efficient code. This in turn is compared to handwritten assembly code to see how efficient the generated code truly is.

4.1 Benchmarks

The collection of benchmarks is presented in Table 4.1. Originally there are more benchmarks apart from these, but they are left out because they are part of, or too similar to another benchmark. Furthermore, there is not a handwritten reference for each of these benchmarks. Namely, only *binarization*, *color conversion* and *convolution* have vectorized assembly code references. From these references *binarization* and *convolution* have been handwritten by L. Waeijen, and *color conversion* is vectorized by the legacy compiler using OpenCL code.

Table 4.1: List of benchmarks.

Benchmark	**Comments**	**Complexity**
Addition	Sums the individual elements of matrix A and B.	– –
Binarization	Converts a pixel image to a binary image.	–
Convolution	Adds each pixel to its local neighbors, weighted by a kernel.	++
DES	A symmetric-key algorithm for encryption of data.	++
Histogram	Plots the frequency distribution of a data set.	+
Matrix Multiplication	Multiplies matrix A and B to form matrix C.	+
Matrix Transpose	Calculates A^T by reordering each row.	+/–
YUV2RGB	Color conversion to transform a YUV image to a RGB image.	+

[1] gcc.gnu.org

Table 4.2: Summarizes which of the benchmarks have been vectorized and provides an overview of how many cycles were additionally executed because of instructions inserted by Section 3.4.2 for both scalar- and vector-version, and how many cycles were additional executed with explicit bypassing for the legacy compiler.

Benchmark	**Vectorized**	**Additional cycles exec.**		
		(legacy)	**(scalar)**	**(vector)**
		4st./5st.	4st./5st.	4st./5st.
Addition	Yes	1/2	0/0	0/0
Binarization	Yes	0/0	1/2	1/2
Convolution	No	33/32	4000/15432	-
DES	No	351/367	1/770	-
Histogram	Yes*	0/2	0/0	257/130
Matrix Multiplication	Yes	0/0	8/2	1/4
Matrix Transpose	Yes*	0/0	1/0	1/2
YUV2RGB	Yes	0/48	0/0	1/4

- N/A.
* vectorized, but less efficient than scalar version.

4.2 Legacy Compiler Results

This chapter uses the geometric means for averages over normalized statistics (RF accesses and cycle counts). The overall gain by using explicit bypassing are shown in Figure 4.1, Figure 4.3 and Figure 4.6 for the legacy compiler, scalar version and vector version of the new compiler respectively. In these figures, the RF accesses performed in the code generated for explicit datapaths are normalized to traditional (implicit) bypassing.

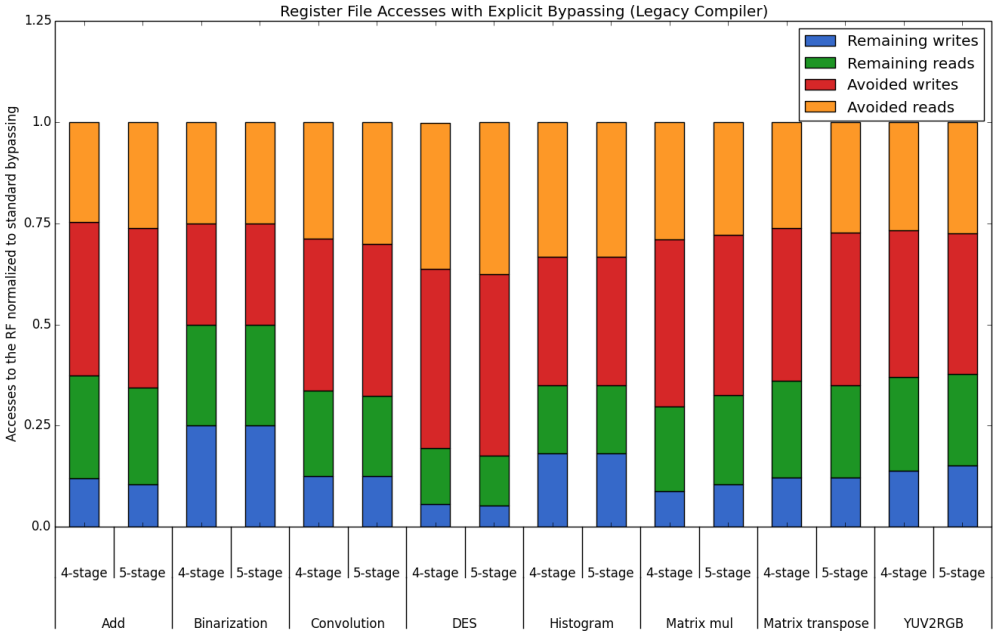


Figure 4.1: Register file accesses with explicit bypassing on the legacy compiler (accesses are normalized to the total number of accesses with automatic bypassing) with an overall reduction of at least 50%, up to 90% and on average 72% of write- and 58% of the read accesses.

The normalized RF accesses that are performed by the code generated by the legacy compiler is shown in Figure 4.1. You may observe that there are many bypass opportunities where the most bypass opportunities are found in the *DES* benchmark where it can save roughly 75% read- and 90% write accesses, on an absolute of 100k write accesses and 150k read accesses in 100k cycles. Since the RF consumes around 35% of the total energy, this would lead to an overall energy improvement of an estimated 30% for that specific benchmark.

4.3 Scalar Version Comparison

This section compares the legacy compiler to the current compiler. It does this by comparing the code generated by the legacy compiler with scalar-only code generated by the current compiler. This gives the following results shown in Figure 4.2. For some benchmarks (*matrix multiplication*) the new compiler has effectively a two times speedup. However, for some benchmarks (*YUV2RGB*) there is actually a decrease in performance. This is caused due to inefficiencies caused by the source-level linker which always adds a `simm` instruction before using a global variable (even if that is not necessary). Furthermore, the code that the legacy compiler generates for *YUV2RGB* has only three basic block, while the code generated by the new compiler has eighteen basic blocks. The new compiler has a decrease in performance for this benchmark because of this structural overhead. However, the new compiler performs better on average.

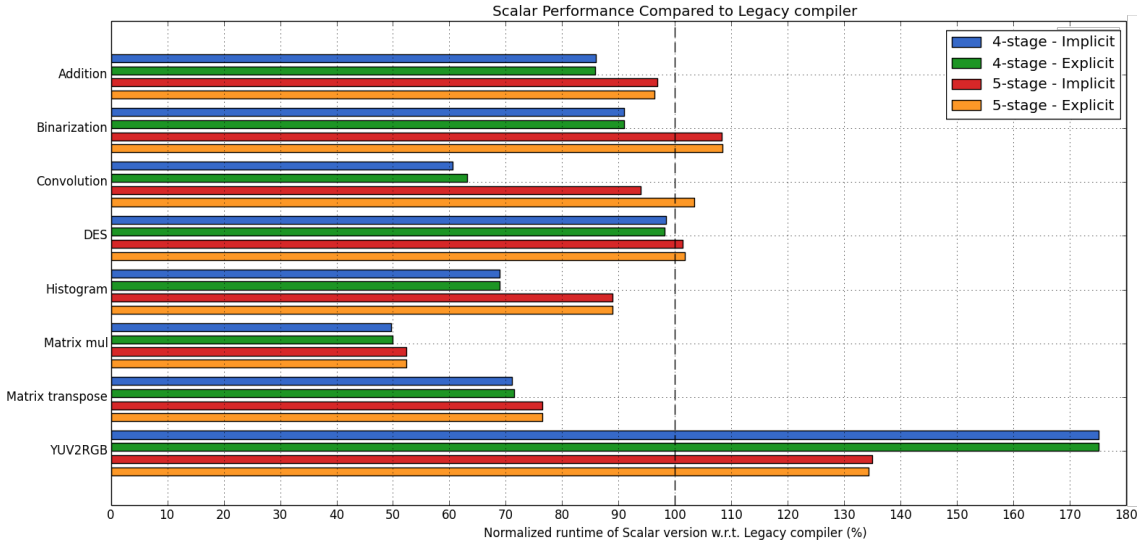


Figure 4.2: Scalar version performance comparision (where cycle counts are normalized to the legacy compiler) where less than 100% indicates an improvement and more than 100% indicates a slowdown compared with the legacy compiler. The average improvement is a reduction of 13.8% in cycles.

Note the code generated for a five-stage pipeline is always worse than the code generated for a four-stage pipeline, because it is effectively the same code with possibly additional stall cycles inserted when a hazard was encountered by hazard recognizer described in Chapter 3.2.1.
To conclude, the new compiler is overall faster with a reduced execution time of 13.8% on average (geometric mean) and since power consumption is directly related to energy and execution time, this already gives significant improvements.
Figure 4.3 shows a bar chart with how many register file accesses can be avoided with explicit bypassing in the new compiler (accesses are normalized to automatic bypassing). The optimizations for these measurements include optimization level `-O3` and a scheduler optimization flag, `-misched-nolimit=1` to limit the scheduling scope and not schedule too far. The instructions are

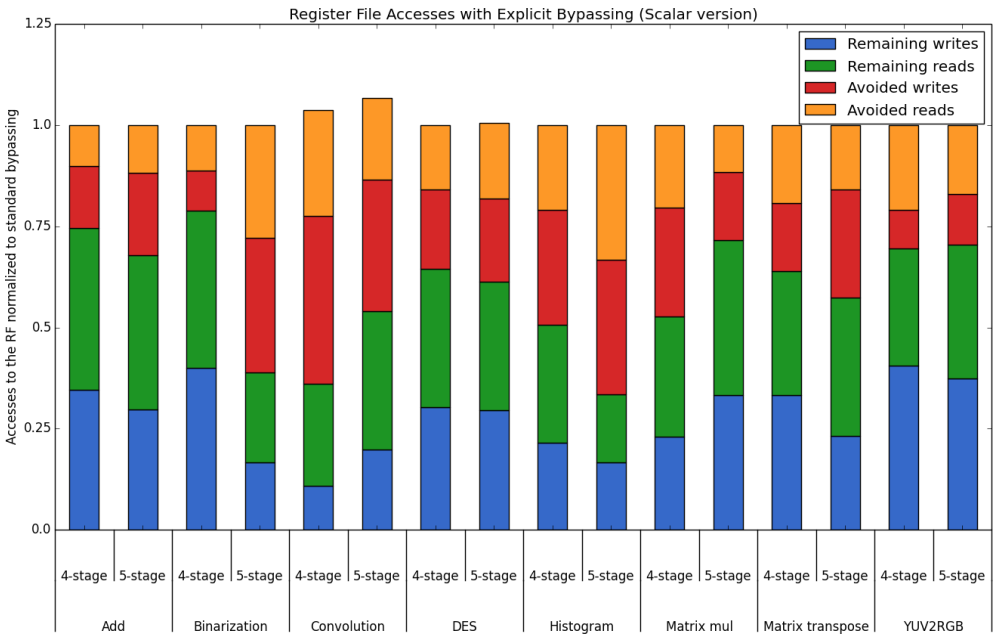


Figure 4.3: Scalar version register accesses (RF accesses are normalized to implicit bypassing), with an average reduction of over 42% on write- and 36% on read accesses.

scheduled closer to their use by not buffering instructions during scheduling. The difference in the total number of cycles does not differ compared to `-O3` with no other optimization flags, but does give an overall improvement in bypass opportunities. There are at least 20% of the accesses that can be avoided, and up to even 83% of the writes can be avoided (*convolution*). On average

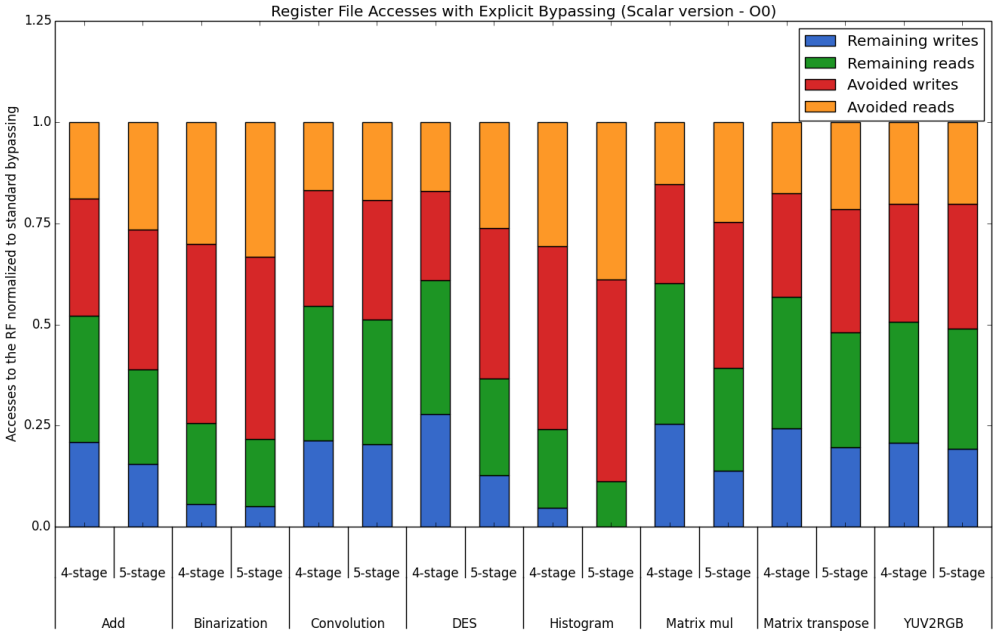


Figure 4.4: Scalar version register accesses with opt-level `-O0` (RF accesses are normalized to implicit bypassing) with an average reduction of 66% on write- and 45% on read accesses, and even up to 99.9% of the write accesses can be avoided.

Table 4.3: Absolute cycles and accesses for *binarization* and *histogram* with an explicit datapath and five-stage pipeline.

Benchmark	**Scalar (O3)**			**Scalar (O0)**			**Legacy**		
	Cycles	Reads	Writes	Cycles	Reads	Writes	Cycles	Reads	Writes
Binarization	20816	6405	3208	32022	8007	**1607**	19211	8001	4805
Histogram	8206	2053	1544	12310	2055	**7**	9228	2561	2052

42% of the write and 36% of the read accesses are avoided. Note that convolution jumps over one, which is because that benchmark contains a lot of spill code in the explicit bypass version and the cleanup pass to remove redundant spill code is not implemented yet. Therefore, the explicit version has a significant longer runtime and many more instructions than with standard bypassing and has, therefore, more RF accesses than standard (implicit) bypassing. This can also be seen in Table 4.2 where it notes that for *convolution* an additional 4000 and 15432 cycles are executed with explicit datapath version of the four-stage and five-stage pipelines respectively.

With the lowest optimization level (`-O0`), the improvement on RF accesses go up to 66% write- and 45% avoided read accesses on average. With such low optimization level, the total number of cycles increases rapidly with an increase of up to 6-7 times (convolution). However, for some benchmarks, e.g. histogram, it still performs quite well with only a 50% increase in cycles, while the increase in utilization of the bypass network is significant (99.9% write accesses are avoided). Figure 4.4 shows RF accesses with explicit bypassing, the lowest optimization level and scalar-only code. A large performance lost due to such low optimization level was found for all benchmarks, except for *histogram* and *binarization*. For these two benchmarks which execute in around 50% more cycles, do have a significant result by avoiding up to almost all write accesses. Table 4.3 shows the absolute accesses and cycle counts for these benchmarks. Note that even though a higher latency results from such low optimization flag, the absolute write accesses drastically decreases.

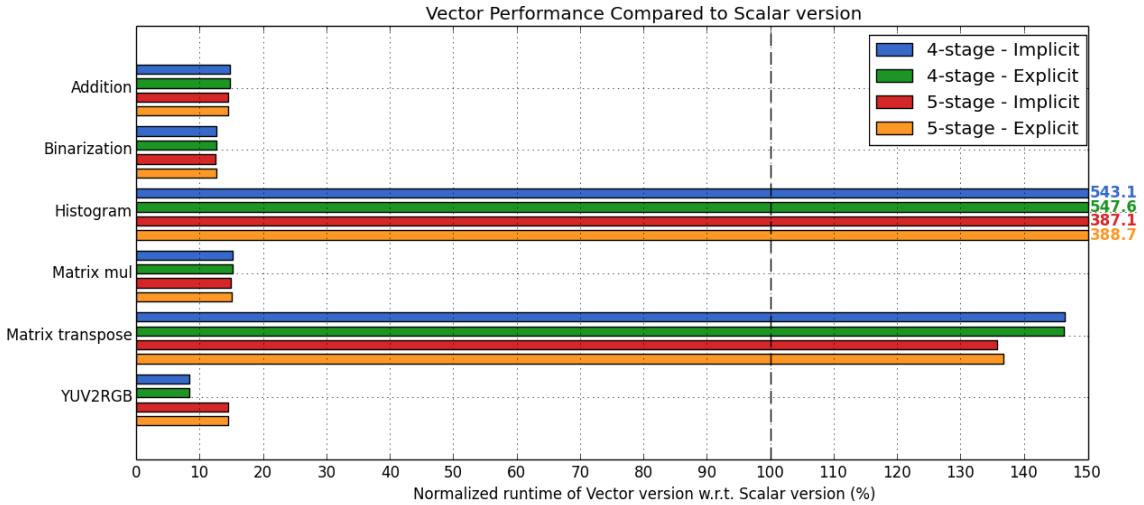


Figure 4.5: Vector version performance gain (cycle counts are normalized to that of the scalar-version), with an average improvement of 65 percent.

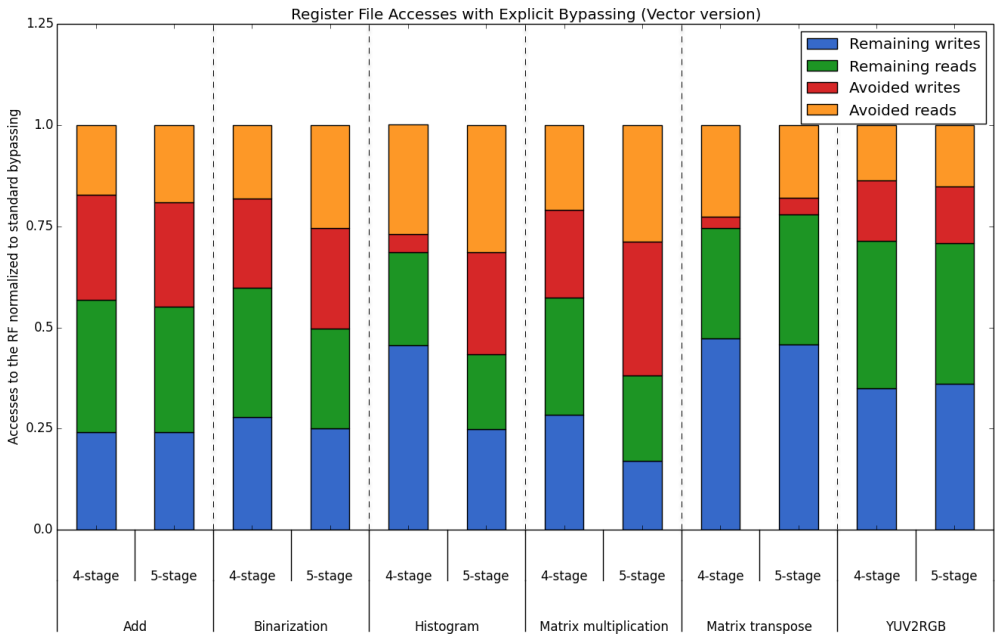


Figure 4.6: RF accesses for the vector version, accesses are normalized to standard bypassing.

4.4 Vector Version Comparison

The *Data Encryption Standard* (DES) benchmark has not been vectorized because of irregular memory accesses. For other benchmarks e.g. *histogram* and *matrix transpose*, vectorization results in less efficient code. This is often caused by neighborhood network communication which is expensive for the target architecture.
Convolution has not been vectorized because of an open issue with vectorized kernels that use reduction. The *binarization* and *YUV2RGV* benchmarks had some difficulties during vectorization because of a compilation problem when multiple set-flag and conditional move operations are present in a generated code. Problems like these and others are described in the future work section in Chapter 5. In overall, the vectorized versions perform better with a maximal obtained speedup of eleven times faster (*YUV2RGB*). However, as already indicated, sometimes, vectorization results in less efficient code, and can cause a four-times slowdown in execution time (histogram). Figure 4.5 provides an bar chart on which one can see how efficient a benchmark can be vectorized and Table 4.2 provides an overview with which of the benchmarks are vectorized.
Finally, Figure 4.6 shows the RF accesses with explicit bypassing for the vector versions. With an average around 30% on write- and over 40% of read accesses can be avoided. With up to 66% avoided write accesses (*matrix multiplication*).

Chapter 5

Conclusions

The previous chapters have described a compiler based on the LLVM infrastructure that is used to compile code for the target SIMD architecture. This chapters summarizes the results and lists limitations of the compiler before presenting future work.

Table 5.1: Summary of RF access avoided with the techniques implemented in this work.

RF Accesses	Scalar version	Vector version
Writes	42%	30%
Reads	36%	40%

With register file optimizations that are developed for the new compiler, around 40% of the accesses can be avoided. Since the RF consumers around 35% of the total energy consumption and around 40% of the communication with the RF can be avoided, an estimated improvement of around 15% follows. Table 5.1 summarized the percentages of RF communication that can be avoided with explicit bypassing.

5.1 Limitations

This sections aims at compiling a list of limitations currently in the compiler or architecture.

- **Floating-point operations** are not supported, because of the lack of a *Floating Point Unit* (FPU), just like most low-end embedded systems do not have a FPU. The functionality of a FPU can be emulated with a soft-floating point library, however a LLD linker is required to link a soft-floating point library together with compiler-rt[1] for software floating-point emulation.
- **Divisions** and **modulo operations** are not supported. Similarly to the previous point, this functionality can be emulated with library calls. However, the actual linking of such libraries requires LLD to properly work.
- **No built-in functions** are supported because there is no C-runtime library nor a finished linker. Because of this, includes to stdlib (standard library) that allow functions like malloc and stdio (input/output library) for file operations are not possible and result in many errors.
- **No inline assembly** support. Some compilers allow low-level code (assembly) to be written embedded within a program. LLVM does have support for this, but this is not supported/implemented for the SIMD backend.

[1] compiler-rt.llvm.org

- **Interrupts** are not supported in the current SIMD architecture design.
- There is **no status register** that contains information about the state of the processor. Common status register flags include a *zero flag* that indicates that the result of the previous operation was zero, a *sign flag* indicates that the result of an operation was negative. There are flag registers that do the same, but require an additional instruction to set. Moreover, status register flags also include an *overflow flag* indicating that a result does not fit in the register or an *interrupt flag* to enable/disable interrupts. The later two flags do have an impact. Namely, the programmer is responsible for avoiding overflows, since they are not detected by the hardware.

5.2 Future Work

Future work can be split up in a couple of categories. First of all, there is future work consisting of bug fixes and other improvements to the compiler. Furthermore, there is also future work that consists of adding functionality to the compiler.

- There is future work from L. Zhenyuan, which includes:
  - The scalar and vector processors share the same frame index. Whenever something is pushed to the stack, the shared frame pointer is updated. This results in gaps in the stacks which can be solved with two frame indices, one for the scalar processor and one for the vector array.
  - Shuffle pattern recognition during instruction lowering. Shuffle pattern recognition and replacement is currently done before DAG lowering. Because of this, the opportunity is lost to generate new vector shuffle operations in the instruction combination during the instruction lowering stage.
- A cleanup pass to remove obsolete spill-code in the generated code after explicit bypass allocation has not been developed yet. Since there is a benchmark where this would be beneficial (convolution) it is added to future work.
- Solving all open issues described in Section 5.3.
- The LLD linker (standard LLVM linker) has to be tested and any issues have to be resolved.
- Implement a disassembler to translate binary code back to assembly code. This would also make debugging the linker a lot easier, therefore, a skeleton implementation is already provided in the source directory. However, it has not been implemented yet because it falls outside the scope of this assignment.

5.3 Open Issues

This section describes the issues in the backend that are also described on the issue tracker in GitLab[2].

- As discussed at the end of Chapter 3.4.1, a module-level scope has not been implemented. Instead, a function-level scope is implemented, and the pipeline state model is reset on a function call, such that it does not have false bypasses over a function call. The epilogue and prologue insertion ensures correct behaviour between function calls, namely the first and last few instructions of a function update the stack pointers. Therefore, no no-ops are required, however, currently the pipeline state is reset on a function call, while it should effectively be reset on the delayslot after a function call.

[2] git.ics.ele.tue.nl/SIMD/LLVM/issues

- Other open issues about explicit bypassing can be found under issue #13, which include a multipass checkConflict approach (where checkConflict is called again upon fixing a conflict), and support for conflicts on the PE array.
- During an internship where another student investigated in how to generate code for *Convolutional Neural Networks* (CNN) and during vectorization of the *convolution* benchmark, a problem was found in which intermediate results calculated by the PE array are not communicated back to the CP. A minimal code example in which this problem occurs can be found on GitLab (issue #11), which helps with debugging.
- Conditional moves operations have a set-flag operations in front of it. The compiler sees these this as two separate operations, and for some reason starts with a couple of set-flag operations. However, the conditional move instruction uses only a single flag register (`P0`) to determine which value it should move. Therefore, it is not possible to start with a couple of set-flag operations. See the example below, where this problem is illustrated:

  ```
  %flagA = v.sflts %vreg19, %vreg20
  %flagB = v.sfgts %vreg19, %vreg13
  %resultA = v.cmov %flagA, %vreg16, %vreg19, %vreg20
  %resultB = v.cmov %flagB, %vreg16, %vreg16, %vreg13
  ```

  The assembly code above is a fragment of code that is produced by the compiler. During register allocation, this gives an error, because it can not find two flag registers, namely, it can only choose flag register `P0`. This bug occurs with the vectorized versions of *binarization* and *YUV2RGB*. For *binarization* the problem is avoided by not unrolling, so that it performs only one conditional move operation per loop iteration. For *YUV2RGB*, manual corrections to the generated code were required to ensure correct simulation behaviour. This issue is also described on GitLab (issue #10).

To conclude, this work has intrigued me in energy efficient embedded systems. The search for low energy systems will definitely continue and the next project may reveal other ways to reduce energy consumption for general purpose processors.

Appendices

Appendix A

Pseudo Code

This appendix provides pseudo code of the pass that exploits explicit datapaths from Chapter 3.4. First of all, the children of each node in the dominator tree are sorted on reverse post-order tree traversel. Then each node and the functions `EnterScope` and `ProcessBlock` are called.

```
Data: PerformEBA
Result: Explicit bypass allocation is performed
Sort dominator tree on reverse post order traversal;
Node = depthFirst(DominatorTree.root());
while node do
    block = Node.block();
    EnterScope(block);
    ProcessBlock(block);
    Node = next(Node);
end
```

Algorithm 1: Outer function called by `RunOnMachineFunction`.

`EnterScope` sets up the pipeline state by taking the bypass state from the most-frequently executed predecessor block. Subsequently, it modifies other predecessor blocks to match their resulting pipeline state when it is required by a bypass.

```
Data: EnterScope
Result: Pipeline state is setup and conflicts on predecessors has been resolved
if isLoop then
    for block in loop do
        AnalyzeBlock(state, block);
    end
    setState(state);
else
    if preds.size() > 1 then
        setState(MostFrequentlyExecutedPred.getState());
        checkConflicts(preds);
    else
        if preds.size == 1 then
            setState(pred.getState());
        else
            initState();
        end
    end
end
```

Algorithm 2: First function called by `PerformEBA`.

Where `AnalyzeBlock` goes through a basic block, and updates the pipeline state model accordingly. The procedure calls to `setState` and `getState` set the pipeline state model to a given state, or queries the pipeline state model of a basic block. The following pseudo code shows functionality of `ProcessBlock`, which is similar to `AnalyzeBlock`, but also allocates explicit bypasses that it finds.

```
Data: ProcessBlock
Result: Explicit bypass allocation is performed on a basic block
for instruction in block do
    allocateBypasses(instruction);
    insertIntoPipeline(instruction);
    register instruction in pipeline state model;
    if end cycle then
        propagate pipeline state;
    end
end
```

Algorithm 3: Second function called by `PerformEBA`.

```
Data: allocateBypasses
Result: Explicit bypass allocation is performed on an instruction
for operand in instruction do
    match = matchOperandInPipeline();
    if match then
        bypass operand;
    end
    if match & isKill then
        avoid store;
    end
end
```

Algorithm 4: Inner function called by `ProcessBlock`.

Appendix B

Instruction Set Architecture

This appendix contains descriptions of the instructions that are supported on the proposed SIMD architecture. We have split the instructions into three classes, Register-type instructions that take only registers as operand. Immediate-type instructions have an immediate value as last operand. Finally, Jump-type instructions are instructions that modify the program counter.

B.1 Register-type Instructions

Opcode	Operation	FU	Description
0	N/A	N/A	Not used.
1	N/A	N/A	Not used.
2	add	ALU	Signed addition.
3	sub	ALU	Signed subtraction.
4	mul	MUL	Signed multiplication.
5	mulu	MUL	Unsigned multiplication.
6	or	ALU	Zero extended bitwise or.
7	and	ALU	Zero extended bitwise and.
8	xor	ALU	Zero extended bitwise xor.
9	cmov	ALU	Conditional move.
10	sfeq	ALU	Set flag if equal.
11	sfne	ALU	Set flag if not equal.
12	sfles	ALU	Set flag if less or equal, signed.
13	sflts	ALU	Set flag if less, signed.
14	sfges	ALU	Set flag if greater or equal, signed.
15	sfgts	ALU	Set flag if greater, signed.
16	sfleu	ALU	Set flag if less or equal, unsigned.
17	sfltu	ALU	Set flag if less, unsigned.
18	sfgeu	ALU	Set flag if greater or equal, unsigned.
19	sfgtu	ALU	Set flag if greater, unsigned.
20	sll	ALU	Shift left logical.
21	sra	ALU	Shift right arithmetic.
22	srl	ALU	Shift right logical.
23	ror	ALU	Rotate right register (not used).

Table B.1: List of R-type instructions, both for CP and PEs.

B.2 Immediate-type Instructions

Opcode	Operation	FU	Description
0	simm	ALU	Signed upper 18-bit for next immediate instruction.
1	zimm	ALU	zero extended upper 18-bit for next immediate instruction.
2	addi	ALU	Signed addition.
3	N/A	N/A	Not used.
4	muli	MUL	Signed multiplication.
5	mului	MUL	Unsigned multiplication.
6	ori	ALU	Zero extended bitwise or.
7	andi	ALU	Zero extended bitwise and.
8	xori	ALU	Zero extended bitwise xor.
9	cmovi	ALU	Conditional move.
10	sfeqi	ALU	Set flag if equal.
11	sfnei	ALU	Set flag if not equal.
12	sflesi	ALU	Set flag if less or equal, signed.
13	sfltsi	ALU	Set flag if less, signed.
14	sfgesi	ALU	Set flag if greater or equal, signed.
15	sfgtsi	ALU	Set flag if greater, signed.
16	sfleui	ALU	Set flag if less or equal, unsigned.
17	sfltu	ALU	Set flag if less, unsigned.
18	sfgeui	ALU	Set flag if greater or equal, unsigned.
19	sfgtui	ALU	Set flag if greater, unsigned.
20	slli	MUL	Shift left logical.
21	srai	MUL	Shift right arithmetic.
22	srli	MUL	Shift right logical.
23	rori	MUL	Rotate right register (not used).
26	lb	LSU	Load byte.
27	sb	LSU	Store byte.
28	lh	LSU	Load half word.
29	sh	LSU	Store half word.
30	lw	LSU	Load word.
31	sw	LSU	Store word.

Table B.2: List of I-type instructions, both for CP and PEs.

B.3 Jump-type Instructions

Opcode	Operation	Description
0	nop	Do nothing
1	SysCall	System call (not used)
2	bf	Branch if flag is set
3	bnf	Branch if flag is not set
4	j	Jump
5	jal	Jump and link
6	jr	Jump register
7	jalr	Jump and link register

Table B.3: List of J-type instruction that only the CP can execute.

Appendix C

Installation Guide

C.1 Overview

Welcome to the installation guide. In order to get started, you first need to know some basic information. First, this project comes in four pieces. The first piece is the LLVM suite. This contains tools, libraries and the implementation of our compiler. It also contains a suite of programs with a testing harness that can be used to further test LLVM's functionality and performance as well as testing our own compilers functionality.
The second piece is the Clang frontend, which compiles C, C++, objective C and objective C++ to LLVM bitcode. Once compiled into LLVM bitcode it can be manipulated with the tools from the LLVM suite.
There is a third, optional piece called LLD, which is a linker from the LLVM project. That is a drop-in replacement for system linkers. More information about LLD can be found on the LLD section of the LLVM website, https://lld.llvm.org.
The fourth piece is called the Solver Toolchain, which contains a Verilog implementation of the target SIMD architecture, as well as tools for debugging and simulation. This Verilog implementation will at some point be replaced by a new templated Verilog implementation. Furthermore, RTL sources and its installation instructions can be found on ES-group's Gitlab under the project called 'Hardware'.

C.2 Prerequisites

- Have Git installed.
- CMake version 3.4.3 or above.
- GNU Make 3.79 or above.
- GCC version 4.8.0 or above.
- Python version 2.7 or above (if you want to run the test suite).

C.3 Installation Guide

For Windows, you may need to connect to one of the TU/e servers over SSH. For Linux and Mac OS X, this guide can be followed by executing the commands (followed by $) in the terminal.

1. **Check out the git repository:**

```
$ cd where-you-want-this-project-to-live
$ git clone git@git.ics.ele.tue.nl:SIMD/LLVM.git -b explicit llvm
```

2. **Checkout Clang:**

```
$ cd where-you-want-this-project-to-live
$ cd llvm/tools
$ git clone https://github.com/llvm-mirror/clang.git -b release_40
```

   Then, follow steps for adding the SIMD target to clang, which is described in Section C.4.

3. (optional) **Check out LLD linker:**

```
$ cd where-you-want-this-project-to-live
$ cd llvm/tools
$ git clone git@git.ics.ele.tue.nl:SIMD/lld.git
```

4. **Configure and build LLVM and Clang:**

```
$ cd where-this-project-lives
$ mkdir build
$ cd build
$ cmake ../ -DLLVM_TARGETS_TO_BUILD="Simd"
    -DLLVM_DEFAULT_TARGET_TRIPLE="simd-unknown-unknown"
$ make -j4
```

Optionally, a generator can be provided with cmake by adding '`-G generator`' to cmake command, for example:

```
$ cmake -G Ninja ../ -DLLVM_TARGETS_TO_BUILD="Simd"
    -DLLVM_DEFAULT_TARGET_TRIPLE="simd-unknown-unknown"
```

Some common generators are:

- **Ninja:** for generating Ninja build files. Most llvm developers use Ninja for its focus on speed.
- **Visual Studio:** for generating Visual Studio projects and solutions.
- **Xcode:** for generating Xcode projects.

C.4 Configuring Clang

To register our target to Clang, we need to modify `Targets.cpp` which can be found in the clang libraries, `path_to_where_llvm_lives/tools/clang/lib/Basic/Targets.cpp`.

1. Each target has one or more classes describing it. Add the following class declaration to describe the SIMD target in `Targets.cpp` before the function `AllocateTarget`, e.g. line 6321:

```
class SimdTargetInfo : public TargetInfo {
public:
    SimdTargetInfo(const llvm::Triple &Triple,
                   const TargetOptions &Opts)
    : TargetInfo(Triple) {
        BigEndian = false;
        NoAsmVariants = true;
        IntWidth = 32;
        IntAlign = 32;
        LongWidth = 32;
        LongLongWidth = 64;
        LongLongAlign = 64;
        SuitableAlign = 32;
        SizeType = UnsignedInt;
```

```
            IntMaxType = SignedLongLong;
            IntPtrType = SignedInt;
            PtrDiffType = SignedInt;
            SigAtomicType = SignedLong;
            WCharType = UnsignedChar;
            WIntType = UnsignedInt;
            resetDataLayout("e-p:32:32-i8:8:32-i16:16:32-i32:32-"\
                            "i64:64-v64:64-v128:128-v256:256-v512"\
                            ":512-v1024:1024-v2048:2048-n32-S64");
        }
        void getTargetDefines(const LangOptions &Opts,
                              MacroBuilder &Builder) const override {
            Builder.defineMacro("__SIMD__");
        }
        ArrayRef<Builtin::Info> getTargetBuiltins() const override{
            return None;
        }
        BuiltinVaListKind getBuiltinVaListKind() const override {
            return TargetInfo::VoidPtrBuiltinVaList;
        }
        const char *getClobbers() const override {
            return "";
        }
        ArrayRef<const char *> getGCCRegNames() const override {
            static const char * const GCCRegNames[] = {
                "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7",
                "r8", "r9", "r10", "r11", "r12", "r13", "r14",
                "r15", "r16", "r17", "r18", "r19", "r20", "r21",
                "r22", "r23", "r24", "r25", "r26", "r27", "r28",
                "r29", "r30", "r31"
            };
            return llvm::makeArrayRef(GCCRegNames);
        }
        ArrayRef<TargetInfo::GCCRegAlias>
                getGCCRegAliases() const override {
            return None;
        }
        bool validateAsmConstraint(const char *&Name,
                                   TargetInfo::ConstraintInfo &Info)
                                   const override {
            return false;
        }
        int getEHDataRegisterNumber(unsigned RegNo) const override {
            // R5=ExceptionPointerRegister R6=ExceptionSelectorRegister
            if(RegNo == 0) return 5;
            if(RegNo == 1) return 6;
            return -1;
        }
    };
```

2. Each target triple is covered in a switch statement in `AllocateTarget` function. Add the following case to that switch:

```
case llvm::Triple::simd:
```

```
        return new SimdTargetInfo(Triple, Opts);
```

Now that we have added our target to Clang, you can proceed with the installation guide or start using the compiler.

Appendix D

File Structure

The file structure of the Simd compiler project consists of the following:

- `llvm` — LLVM root directory
  - `lib` — Library files
    - `Target` — Target specific files
      - `Simd` — SIMD target files
        - `AsmParser`
          - `SimdAsmParser.cpp` — Parse SIMD assembly to MCInst instructions
        - `Disassembler`
          - `SimdDisassembler.cpp` — A skeleton for a SIMD disassembler
        - `InstPrinter`
          - `SimdInstPrinter.cpp` — Prints MCInst to assembly file
          - `SimdInstPrinter.h` — MCInst printer header file
        - `MCTargetDesc`
          - `SimdABIInfo.cpp` — Information about SIMD ABI's
          - `SimdABIInfo.h` — ABI information header file
          - `SimdAsmBackend.cpp` — Generic interface to SIMD specific assembler
          - `SimdAsmBackend.h` — Header file for assembler interface
          - `SimdBaseInfo.h` — Helper functions and enum defs for MC and SIMD
          - `SimdELFObjectWriter.cpp` — Prints object in ELF format
          - `SimdELFStreamer.h` — Empty file, should be removed
          - `SimdFixupKinds.h` — Type of fixups that may be encountered
          - `SimdMCAsmInfo.cpp` — Contains some basic section info for SIMD
          - `SimdMCAsmInfo.h` — Section directive header file
          - `SimdMCCodeEmitter.cpp` — Encode SIMD machine instructions
          - `SimdMCCodeEmitter.h` — Generic instruction encoding interface header file
          - `SimdMCExpr.cpp` — Assembler expressions which are needed for parsing
          - `SimdMCExpr.h` — Assembler expressions header file
          - `SimdMCInstrInfo.cpp` — Utility functions for SIMD specific MCInst queries
          - `SimdMCInstrInfo.h` — MCInst header file
          - `SimdMCTargetDesc.cpp` — SIMD specific target descriptions
          - `SimdMCTargetDesc.h` — SIMD specific target descriptions header file
        - `SimdAsmPrinter.cpp` — Prints machine instrs to assembly file
        - `SimdAsmPrinter.h` — Machine Instr printer header file
        - `SimdBranchSimplify.cpp` — Removes obsolete jump instructions
        - `SimdCallingConv.td` — Describes SIMD calling conventions
        - `SimdDelaySlotFiller.cpp` — Custom pass to utilize delay slots
        - `SimdExplicitBypassRegs.cpp` — Custom pass for explicit datapaths
        - `SimdExplicitBypassRegs.h` — Explicit datapath pass header file
        - `SimdFavorNonGenericAddrSpaces.cpp` — Optimization pass for memory instrs

- `SimdFrameLowering.cpp` ..... Lowering stack and frame Information
- `SimdFrameLowering.h` ..... Frame lowering header file
- `SimdGlobalArrayToVector.cpp` ..... Changes global arrays to vector addr space
- `SimdHazardRecognizer.cpp` ..... Hazard detection for 5-stage pipeline
- `SimdHazardRecognizer.h` ..... Hazard detection header file
- `SimdInsertImmExtend.cpp` ..... Pass to insert `zimm` and `simm` instrs
- `SimdInstrFormats.td` ..... Defines type of instructions
- `SimdInstrInfo.cpp` ..... Contains helper functions for manipulating instrs
- `SimdInstrInfo.h` ..... Instruction info header file
- `SimdInstrInfo.td` ..... Defines scalar Instructions
- `SimdInstrInfoVec.td` ..... Defines vector Instructions
- `SimdInstrInfoVecFlag.td` ..... Defines flagged vector Instructions
- `SimdISelDAGToDAG.cpp` ..... Instruction selector for SIMD target
- `SimdISelLowering.cpp` ..... Lowers LLVM code into a selection DAG
- `SimdISelLowering.h` ..... Lowering header file
- `SimdLowerAlloca.cpp` ..... Lowers vector alloca to local addr
- `SimdMachineFunctionInfo.cpp` .... Contains machine function helper functions
- `SimdMachineFunctionInfo.h` ..... Machine function Info header file
- `SimdMCInstLower.cpp` ..... Lowers machine instrs to their MCInst records
- `SimdMCInstLower.h` ..... Lower to MCInst header file
- `SimdPromoteConstant.cpp` ..... Promotes constants to global variables
- `SimdRegisterInfo.cpp` ..... Implements register specific helper functions
- `SimdRegisterInfo.h` ..... Register info header file
- `SimdRegisterInfo.td` .... Defines SIMD specific registers and their dwarf index
- `SimdSched4ST.td` ..... Defines two issue slots and single cycle latencies
- `SimdSched5ST.td` .... Defines two issue slot and latencies for every function unit
- `SimdSchedule.td` ..... Defines InstrItinClass for every function unit
- `SimdSelectionDAGInfo.cpp` ..... Empty selection DAG implementation
- `SimdSelectionDAGInfo.h` ..... Selection DAG header file
- `SimdShuffleModel.cpp` ..... Contains function pass to insert shuffle operations
- `SimdSubtarget.cpp` ..... Declares the SIMD specific subclass
- `SimdSubtarget.h` ..... SIMD subtarget header file
- `SimdTargetMachine.cpp` ..... Configures SIMD custom passes
- `SimdTargetMachine.h` ..... Target machine header file
- `SimdTargetObjectFile.cpp` ..... Contains helper functions for sections
- `SimdTargetObjectFile.h` ..... SIMD object sections header file
- `SimdTargetTransformInfo.cpp` ..... Cost estimation of SIMD transformations
- `SimdTargetTransformInfo.h` ..... SIMD cost estimation header file
- `SimdVLIWPacketizer.cpp` ..... SIMD custom packetizer Pass
- `SimdVLIWPacketizer.h` ..... SIMD packetizer header file
- `TargetInfo`
  - `SimdTargetInfo.cpp` ..... SIMD target implementation

Appendix E

Class Diagrams

There is a class called `BypassState` which should be inherited by each pipeline that we support, e.g. four-stage and five-stage pipeline. It models the values on busses in the bypass network.
We give a class diagram for `BypassState` in Figure E.1. The classes `Bypass4State` and `Bypass5State` represent the derived classes for the four-stage and five-stage pipeline respectively. Here we use `insertBypassToken` to pass the instructions in a basic block as tokens to `BypassState` one at a time and propagate the tokens each cycle using `propegateBypassTokens`. There is a variable in the derived classes, called `pipe` which represents the instructions that are

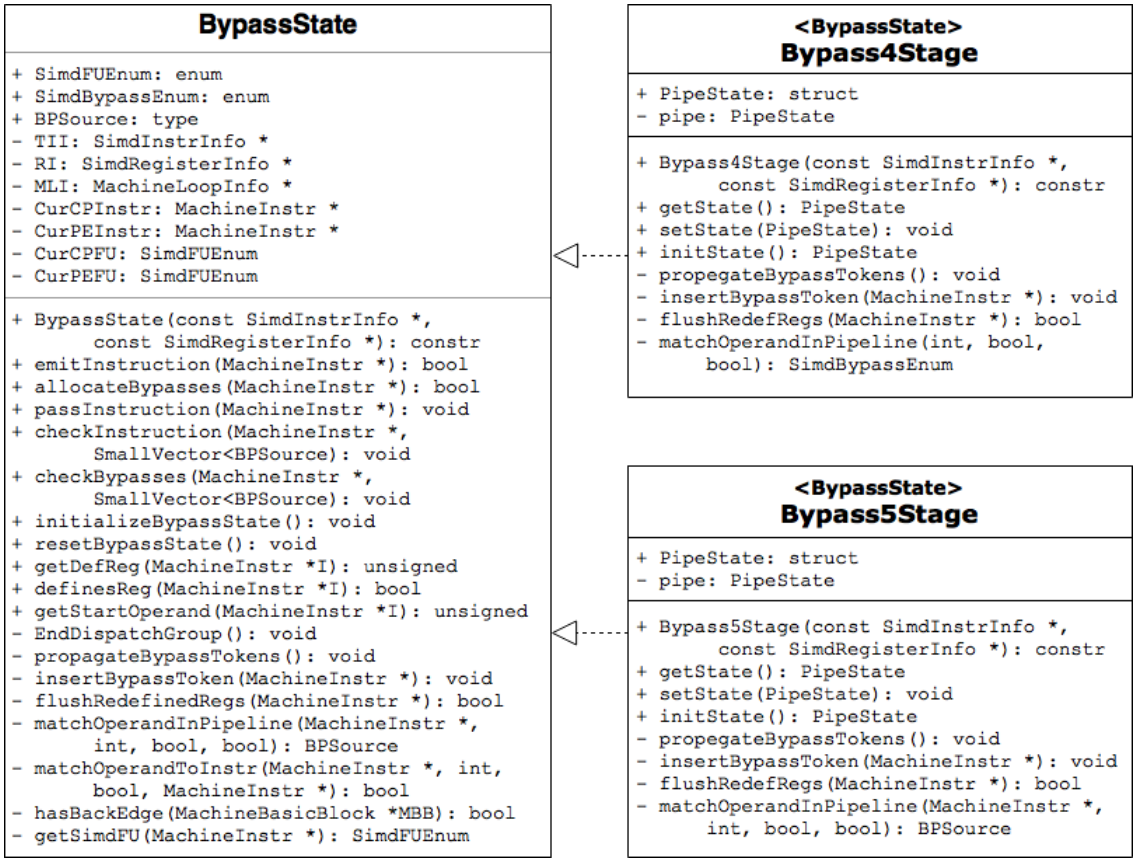


Figure E.1: Class diagram for BypassState, and the inherited classes for the four-stage and five-stage pipelines.

currently in our pipeline. The pipeline state can be queried at any giving moment using function `getState`. This function can be used to acquire the state just before a jump or at the end of a basic block. The functions `initState` and `setState` may be used to initialize the state to a empty state, or a given state which is typically done at the beginning of a basic block. We use a function called `matchOperandInPipeline` to see what bypass can be allocated on a particular use operand of an instruction. It checks whether the operand uses a register defined by any of the instructions the pipeline (by calling `matchOperandToInstr` and `getDefReg` for each instruction can be forwarded). In general, it needs to find the newest definition of the register under consideration, therefore, when inserting a bypass token in the pipeline state we remove all occurrences using `flushRedefRegs`. This way we never have more than one instruction in the pipeline that define the same register, and thus always find the newest definition, if any.
Now lets continue with functions that handle instructions which are used to emit, pass or check an instruction (`emitInstruction`, `passInstruction` and `checkInstruction`). Emitting an instruction consists of the process of specifying which operations are in the current cycle and bypassing their operands according to the current pipeline state. Then pass instruction does the same, but the operands are not bypassed and check instruction does also not actually bypass the operands, but does keep the bypasses that it would allocate in a list. The functions `allocateBypasses` and `checkBypasses` do the actual bypassing work by calling `matchOperandInPipeline` which compares each operand of an instruction to that can be bypassed. However, there are also flag operands that we do not consider.

```
%loop
    sfgts r2, -64     || v.sfltu   P1, r1, 4
    bf    %loop       || v.slli.P1 r2, r1, 2
    add   r2, r2, -1  || v.add.P1  r7, r6, r2
%end:
                          ⋮
```

Flag operands occur before register source operands. The code fragment above shows an example of predicate instructions that uses flag operands to either do or not do a certain operation. In this case, we do a shift and add it with something if PE index is smaller than four ($P1$ is true). The function `getStartOperand` is used to skip flag operands. Alternatively we could also just ignore an operand if it is a flag operand.

After each cycle, a call is made to `EndDispatchGroup` which calls `insertBypassTokens` for each instruction in the current dispatch (can be two operations, a scalar and a vector op) and propagate the bypass tokens. So to summarize, operands are bypassed when they come across, and at the end of each cycle the pipeline is updated.