Technische Universiteit
Eindhoven
University of Technology

Department of Electrical Engineering
Electronic Systems Group

SIMD Compiler Made Explicit in LLVM

Preparation Report

Guus Hendrikus Peter Leijsten

Supervisors:
Prof. dr. ir. Henk Corporaal
Dr. ir. Roel Jordans
Ir. Luc Waeijen

Draft version

Eindhoven, March 2017

Abstract

THIS IS MY ABSTRACT

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

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ALU	Arithmetic Logical Unit
AST	Abstract Syntax Tree
CP	Control Processor
DAG	Directed Acyclic Graph
DLP	Data Level Parallelism
ES Group	Electronic Systems Group
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
MIPS	Microprocessor without Interlocked Pipeline Stages
N/A	Not Applicable
PE	Processing Element
pJ	peta-Joule
RA	Register Allocation
RF	Register File
RP	Register Pressure
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
R-type	Register-type
SSA	Static Single Assignment
TTA	Transport Triggered Architecture
VLIW	Very Long Instruction Word
WB	Write Back

Chapter 1

Introduction

Nowadays, mobile phones have dedicated processors to support video processing. This embedded streaming processor consumes tens of pJ per operation (pJ/op) and the battery capacity is only sufficient for playing video applications for a few hours [1]. Furthermore, embedded systems like mobile devices have to run high performance applications like video encoding/decoding, wireless signal processing and 3D processing [2]. These kind of devices often have a limited energy source, and because it is a handheld device, heat produced by power dissipation is another concern. For these reasons, energy efficiency is becoming the bottleneck in the design of such embedded systems.

In general, it is important to improve both performance and energy efficiency. The Very Long Instruction Word (VLIW) architecture is one example architecture designed to improve performance by executing multiple instructions in parallel, exploiting a program's Instruction Level Parallelism (ILP). By exploiting parallelism, the processor requires less cycles to do the same amount of work, thereby improving performance.

The Transport Triggered Architecture (TTA) is similar to the VLIW architecture. However, instead of packing the operations in a single instruction, TTAs pack multiple transports in a single instruction [3]. For traditional VLIWs, each Register File (RF) is connected to every Function Unit (FU). Unlike VLIW, TTAs do not require that each FU has their own private connections to the RFs. Instead, an FU is connected to the RF by means of an interconnect. Another advantage that TTA has over VLIW is that it has explicit datapaths. With explicit datapaths, software bypassing is possible. The compiler can eliminate some RF accesses, which improves energy efficiency.

Image and video processing applications typically have a high amount of Data Level Parallelism (DLP). The advantage of the Single Instruction Multiple Data (SIMD) architecture is that it naturally exploits DLP by processing multiple operations in parallel instead of processing them sequentially. Therefore, the same performance can be achieved at a much lower clock frequency, thereby reducing energy consumption [2]. Also the instruction fetch and decode energy is shared amongst the processing units. This gives a higher energy efficiency with respect to an architecture where each processing unit has its own instruction fetch and decode, e.g. VLIW or TTAs. A common bottleneck to achieve even higher energy efficiency with SIMD are the register files which consume a large amount of energy every time they are accessed. This work focusses on improving the energy efficiency by reducing accesses to the register files.

The SIMD architecture also many processing units that each have their own register file. This sums up to many register files that consume around 34.6% of the total energy consumption [2]. To further reduce energy consumption, we add explicit bypassing on top of the SIMD in order to eliminate some RF accesses. Adding explicit bypassing on top of the SIMD architecture resembles the TTA. Namely SIMD with explicit bypassing has an explicit datapath, like TTA also has an explicit datapath.

In the past, a compiler for the proposed architecture has already been implemented. This compiler exploits explicit bypassing [2] and can compile a subset of OpenCL and C code [4]. However, this compiler has a custom backend, and in order to standardize and improve maintainability, we want to use a compiler framework that supports our needs. To this end we use the LLVM framework to design and build a compiler for the proposed SIMD architecture.

This master project is part of a bigger project in which we work on the architecture and its compiler. There is another master's project going on about writing a compiler for the proposed SIMD architecture using the LLVM framework [5]. However, this compiler focusses on vector instructions and does not support explicit bypassing. Therefore, we will focus on adding explicit bypassing on top of the new compiler. Furthermore, the proposed SIMD architecture is designed to be configurable. Another master's project will investigate in how to generate hardware code for different combinations of parameters.

1.1 The SIMD Processor Architecture

The advantage of the SIMD architecture is that multiple operations are processed in parallel instead of processing them in a sequence. Therefore, the same performance can be achieved at a much lower clock frequency, thereby reducing energy consumption [2]. Furthermore, because each Processing Element (PE) executes the same instruction, the Instruction Fetch (IF) and Instruction Decode (ID) can be shared amongst the PEs, reducing energy consumption.

We propose a wide SIMD architecture [1] that performs wide vector operations that exploit DLP by executing the same instruction on multiple data simultaneously. Figure 1.1 shows a general overview of the SIMD processor. We have one Control Processor (CP) responsible for scalar operations and control flow e.g. jump/branch instruction. Furthermore there is a wide array of PEs responsible for vector operations. The CP executes in parallel with the PEs, exploiting ILP.

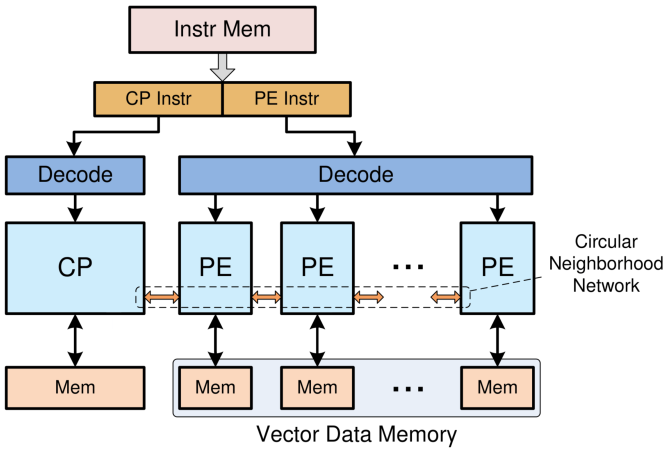


Figure 1.1: General overview of the wide SIMD architecture.

The proposed architecture has a Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA) that is divided up in three categories of instructions. In general, instructions have two operands and a destination register. Instructions that take two register files as operands are Register-type (R-type) instructions. Instructions that take a register file and an immediate as operands are called Immediate-type (I-type). The control flow can be controlled by using Jump-type (J-type) instructions, which can only be executed by the CP.

Furthermore, the architecture is designed to be configurable, e.g. width of the PE array, bit width of the wires and registers, the number of stages that the instruction pipeline consists of, and

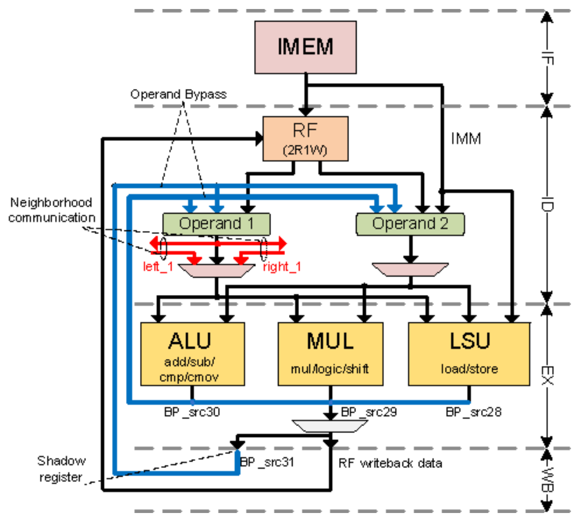


Figure 1.2: 4-stage pipeline processor overview.

whether it has implicit or explicit bypassing, can be configured. The datawidth of the wires and registers can be configured into 16-bits or 32-bits.

In order to support a configurable number of PE elements, a neighbourhood network topology is chosen 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 places in a circular manner [4].

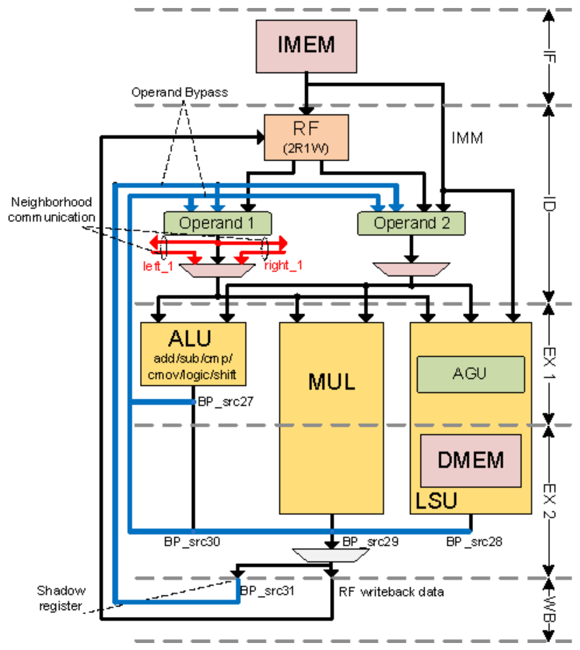


Figure 1.3: 5-stage pipeline with processor overview.

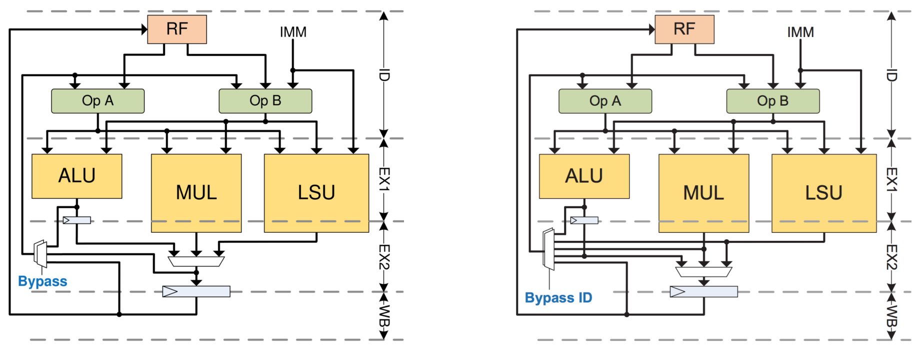
1.1.1 Processor Pipeline and Datapath

Generally, each processor (CP or PE) has its own registers and three functional units, i.e. ALU, MUL and LSU.

The instruction pipeline is divided up in four or five stages. Top down, we have an IF-stage, an ID-stage, one or more execution stages and a Write Back (WB) stage. The architecture shown in Figure 1.2 has four stages while the architecture shown in Figure 1.3 has five stages.

The neighbourhood communication network is implemented by overriding the output of *Operand* 1 in ID-stage. Depending on the decoded instruction, data is either selected from another (neighbouring) processor, or from itself. 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 [2]. The outputs can be used in the bypass network to bypass any of the operands in an instruction.

We can configure the SIMD to have either explicit, or implicit bypassing. With implicit bypassing, also called transparent bypassing it is the hardware's responsibility to handle bypassing. With explicit bypassing on the other hand, it is the compilers responsibility to handle bypassing.



(a) Datapath with implicit bypassing. (b) Datapath with explicit bypassing.

Figure 1.4: Bypassing network differences between implicit bypassing and explicit bypassing.

One of the advantages of explicit bypassing is that when possible a write to a register can be avoided. Thus reducing the total energy consumption of the register file. 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 [2]. Because of this, reducing the register file's energy consumption is of great importance. Furthermore, the explicit datapath shown in Figure 1.4b has two extra sources compared to the transparent datapath in Figure 1.4a. 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 we do not have to change the instruction format in order to specify that an operand of an instruction is bypassed from a previous instruction.

Table 1.1: Special purpose registers.

Register	Purpose
r0	Constant value zero.
r1	PE_ID (PE only).
r3 and *r4*	Return value registers.
r5 through *r8*	Argument passing.
r9	Link register (CP only).
r10	Frame pointer.
r11	Stack pointer.

The total number of registers grows linearly with the number of PEs because each processor has 32 registers. With a wide SIMD, we therefore have many registers that in total consume a considerable amount of energy, namely 34.6% of the total energy consumption [2]. Some of these registers have a special purpose, e.g. register R_0 is connected to ground is has a static value of 0, these are shown in Table 1.1.

1.2 The LLVM Infrastructure

The LLVM project 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, although 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 extendibility. LLVM is a collection of modular and reusable compiler and toolchain technologies. Generally, LLVM follows a 3-phase design, which is divided up in a frontend, a code independent optimizer and a backend, illustrated in Figure 1.5.

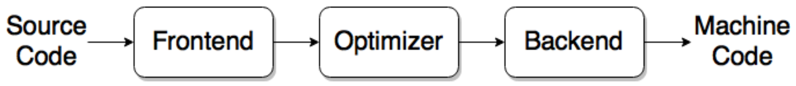


Figure 1.5: 3-phase design: frontend, optimizer and backend.

The frontend is responsible for translating code of an arbitrary 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 dataflow optimizations because only a single definition can reach a particular use of a value, and finding that definition is trivial [6]. As the first phase of a compiler, 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 we verify that the sequence of tokens can be reconstructed according to the syntax of the input language. The parser should report any syntax errors during this process and should be able to recover 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. Type checking is also done during this stage, and the information is kept in the syntax tree. The result of these phases is an Abstract Synctax Tree (AST) of the program, which can be translated into three-address IR code.

The optimizer contains a collections 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. Below we give some of these optimizations that are explained more detailed in literature [7, 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 expressions 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, if we have a copy statement, $x = y$. Then the uses of x can be replaced by y. Some optimizations require that this optimizations is performed afterwards to cleanup, e.g. common sub-expression elimination requires this pass to run afterwards.
- *Dead code elimination* removes code that do not affect the programs 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 performance benefit, it replaces the call of 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 in 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 1.6. 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 one 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 into machine nodes, by matching patterns. Then we have a DAG consisting of only target specific machine instructions, in SSA form. Having naive machine instruction, the next step is to schedule them. We schedule the machine instructions 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, taking the physical registers and register pressure, that is known at this point, into account. After that, some epilogue and prologue code might need to be inserted, e.g. saving/restoring the caller/callee registers and reserving/destroying of the function's stack frame. The peephole optimizations are target specific improvements to the schedule that has been constructed. These optimizations deal with very specific optimizations that can only be done at the end of the process. At last, the assembly printer, prints the assembly code.

1.2.1 Scheduling and RA

TODO: exlain phase ordering between scheduling and RA, from ref [7, Chapter 10.2.4]. Introduce terms spilling, register pressure, and use this reference [8].

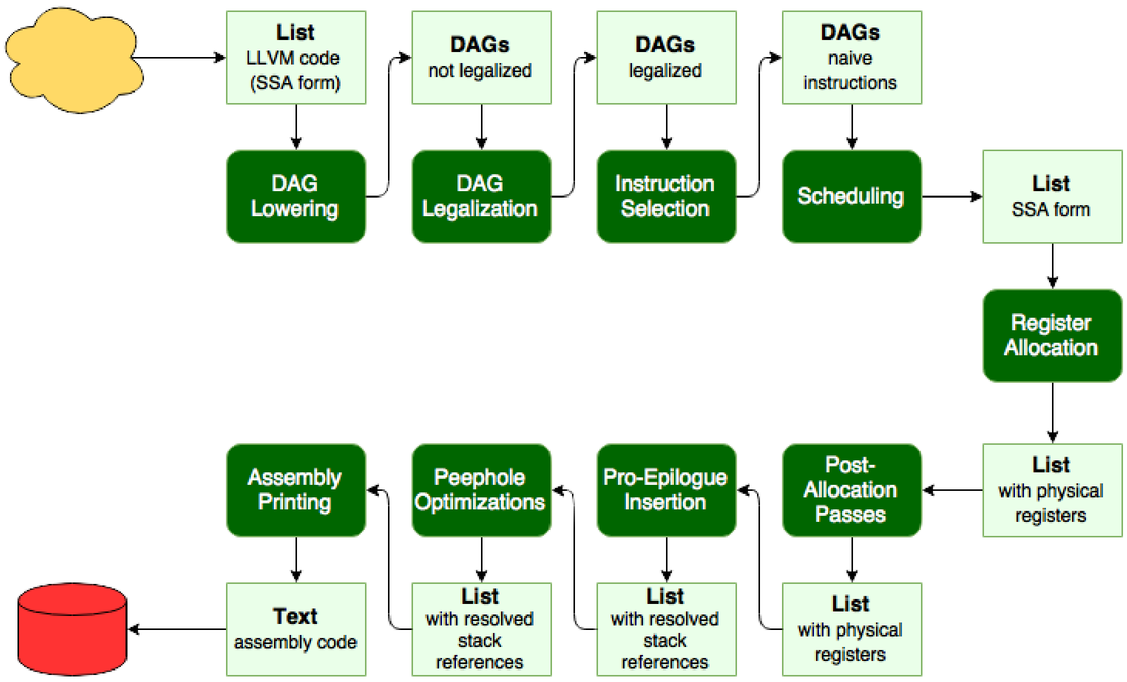


Figure 1.6: Code generation sequence, from LLVM code to assembly code.

1.3 Problem statement

An SIMD architecture has been designed and the compiler that has already been build in the past is too custom and difficult to maintain. Many optimizations still have to be implemented. One of the advantages of LLVM is that a large community works on it. Many optimizations are already in place and can immediately be used for our compiler. Standardizing and having proper tool support improves maintainability of the compiler. For this reason, a new compiler is being implemented in LLVM.

The compiler that is being implemented in LLVM supports vector operations and can compile any IR code into SIMD specific assembly. However, the old compiler applies implicit bypassing, that we have explained in Chapter 3.

The goal of this project is therefore to generate efficient code to support explicit bypassing.

1.3.1 Main Questions

1. How do we implement explicit bypassing in LLVM.
2. Which steps are necessary to do said implementation.
3. What alternative approaches are there to implement it.
4. How do we know which approach is better.
5. How is this implementation of explicit bypassing relevant for other architectures.

1.4 Overview

In the remainder of this report we will discuss related works in Chapter 2. We will discuss bypassing, and show the difference between implicit and explicit bypassing in Chapter 3. Proposed solutions for how to achieve our goal will be discussed in Chapter 4. We will provide a planning for this master thesis in Chapter 5 and early conclusions are given in Section 6.

Chapter 2

Related Work

Chapter 3

Explicit Bypassing

Figure 3.1 shows the basic principle of bypassing. We have scheduled two instructions and the result of the first instruction is used in the second instruction. During the first cycle, the first instruction is fetched from Instruction Memory (IMEM). In the second cycle, the first instruction selects the operands from the RF, and the second instruction is fetched from IMEM. In the third cycle, the first instruction is executed, while the second instruction loads the operands from the RF. During the fourth cycle, the result of the first instruction is written back to the RF, while the second instruction is executed.

Normally, the result of the first instruction is not available before it has been written back to the register file. However, as indicated by the arrow that goes from the first to the second instruction, we bypass this value directly to the second instruction. Now the result of the first instruction is available one cycle earlier, and because of this, the second instruction can be scheduled one cycle earlier.

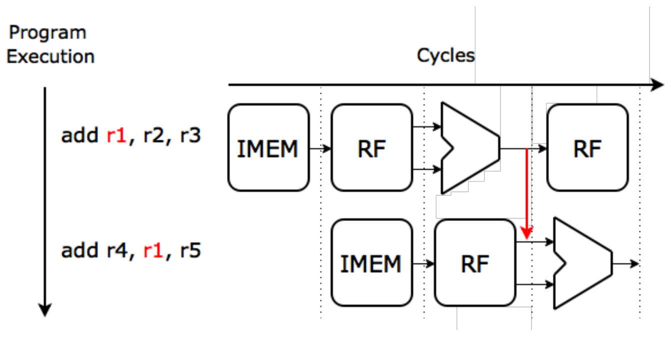


Figure 3.1: Illustration of bypassing and software pipelines in general.

With implicit bypassing we have wires that connect the outputs from EX stage to the ID stage, as illustrated in Figure 3.2. Furthermore, we also have wires that go from the WB stage to the ID stage, however this is not shown in this example. To detect bypasses, the HW matches whether the operands of the currently issued instruction to the destination address of previously issued instructions. If we have a match, and the result is still available in the pipeline, we can then bypass it. We have a mux that controls which inputs are used, i.e. a value from a register, or a value from a bypass. The bypass detection hardware is in control which of these values is selected.

For explicit bypassing, we have the same wires that connect outputs of EX stage to the ID stage as we have in implicit bypassing. However, with explicit bypassing the compiler is responsible for detecting when we can bypass the result of a instruction. In Figure 3.3 we show that the control signal to the mux, that selects the input from a register or from a bypass, is now controlled by the

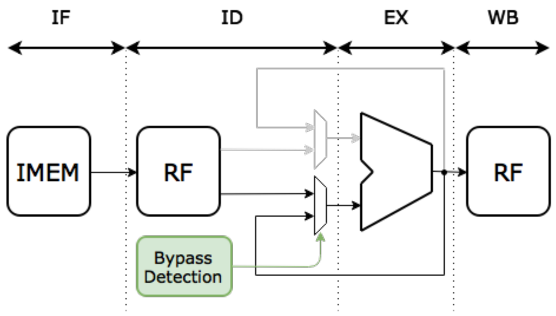


Figure 3.2: Illustration that shows the basic principles of implicit bypassing.

compiler. The compiler encodes this information in an instruction. This way, during instruction decoding, the control signal is immediately available. Furthermore, we have a read-enable flag on the register file. If we then want to take the data from a bypass, we can set this value to zero. This way, we can avoid speculative reads accesses from the RF. We use the same signal from the ID stage, to control the read-enable on the register file and the signal that controls the mux.

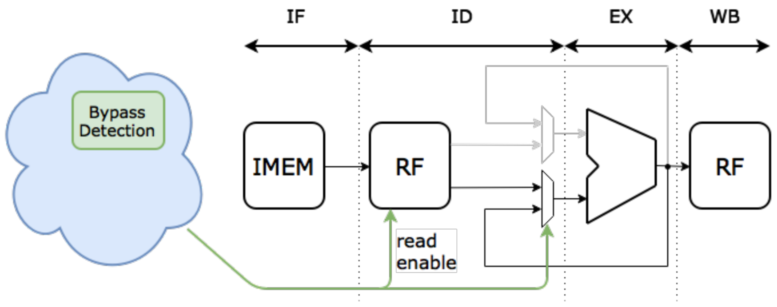


Figure 3.3: Illustration that shows the basic principles of explicit bypassing, read enabled port on RFs to avoid unnecessary reads.

We can use the liveliness information during compilation to determine whether a variable is live after it is used. If the variable is not live after it is used often indicated by a kill of a variable, we can disable the write on the register, since it will not be needed anymore. Figure 3.4 illustrates this by adding a write-enable flag on top of the read-enable that was already present from Figure 3.3. Adding a write-enable flag allows us to avoid speculative write accesses to the RF.

The most important difference between implicit and explicit bypassing is that we can avoid specu-

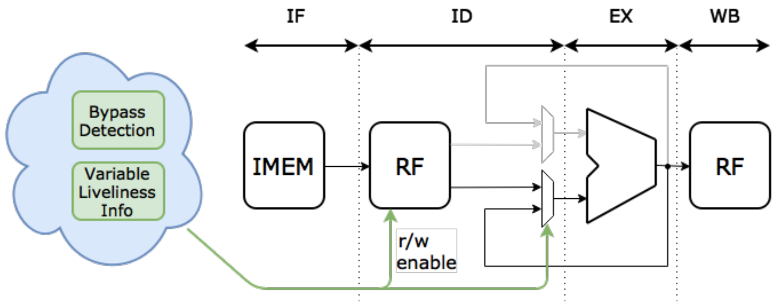


Figure 3.4: Illustration that shows the basic principles of explicit bypassing, read/write enabled port on RFs to avoid unnecessary writes.

lative writes accesses with explicit bypassing. This is not possible with implicit bypassing, because we would have to look at operations that will be scheduled in the future, in order to see whether the result of an operation will be used later on, which is obviously not possible.

We will now show with an example that avoiding speculative write accesses can reduce the Register Pressure (RP).

Listing 3.1: Example code fragment where no bypassing is specified.

```
mul r1, r2, r3
add r4, r1, r5
```

Listing 3.1 shows a code fragment with a multiplication and an addition. We have a flow dependent dependency. Namely, the result of the multiplication is used by the addition.

Listing 3.2: Example code fragment avoiding a read access.

```
mul r1, r2, r3
add r4, MUL, r5
```

In Listing 3.2 we have replaced the use of $r1$ with MUL. This indicates that we do not take the first operand from the RF, but from a bypass instead. Since we obtain the result of the multiplication from the bypass network, we do not require a read access to $r1$ anymore.

Listing 3.3: Example code fragment avoiding a read and a write access.

```
mul --, r2, r3
add r4, MUL, r5
```

When the result of $r1$ is not needed anymore, for example, the live range of $r1$ spans no further than this addition, the write access can be avoided. By specifying $--$ as destination, the result will not be written back, as illustrated in Listing 3.3. Register $r1$ is now completely removed from the example, effectively freeing that register. We have now reduced RP by one, since we require one less register. The freed register can be used for other calculations, which may lead to a higher performance.

3.1

Introduceer bypassing over loop iterations, example, conclusion below example.

Listing 3.4: Example code fragment bypassing through loop iteration.

```
      lw r6, r10, 5
loop: add r7, r6, %src1
        ⋮
      sflts r6, 0
      bf loop
      addi r6, r6, -1
        ⋮
```

Here we want to apply bypassing to bypass the result of the last add instruction in the loop before we branch to the first instruction in the loop. However, when we go in the loop for the first time, $r6$ comes from the LSU instead of the ALU. Therefore, sometimes it is necessary to insert an instruction before we enter a loop to bypass over loop iterations.

Chapter 4

Proposed Solutions

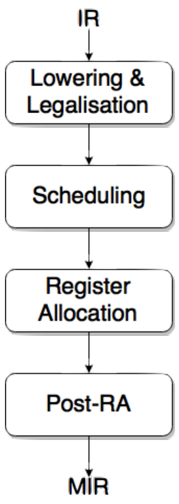


Figure 4.1: General code generation phases.

Post RA pass to allocate bypass sources by demoting operands to bypass registers where possible. When a bypass is allocated, it replaces the register that it used to take as operand, therefore effectively freeing a register. However in the post-RA stage, spill code has already been inserted, this will not benefit in terms of freeing registers, unless we move spill code explicitly. This will be be the easiest approach, because even without moving the spill code, this method will still work, but with too early inserted spill code as a result.

Bypass-aware register allocator that allocates bypass sources as much as possible, and then allocates any virtual registers not bypass as any normal register allocator. In this approach the freed registers can be utilized which should give overall better results.

If we allocate bypasses after scheduling, but before register allocation, we can change a virtual register into a bypass register when possible. However, during register allocation or any other passes that follow, code might be inserted that break a previously allocated bypass. For this reason, we will need another pass that will identify these broken bypasses, and change them back into virtual registers so that they are dealt with by the register allocator.

Before scheduling, we can identify instruction pairs that we can bypass. Then we change the virtual register that is bypassed into physical (bypass) registers and glue the two instructions together. This way the scheduler will try to keep them together, and if it fails to do so, we might need to undo this bypass allocation in order to ensure that the program still works correctly.

Implement a bypass-aware combined scheduler and register allocator.

In Chapter 1.1.1 we have briefly discussed the bypassing sources of the proposed architecture. For convenience we have given them again in Table 4.1.

Table 4.1: Alias for each bypassing source, BP_src in Figure 1.2 and Figure 1.3.

Register:	**Bypass source:**	**Alias:**	
		4 stages	**5 stages**
r27	BP_src27	N/A	ALU1
r28	BP_src28	LSU	LSU
r29	BP_src29	MUL	MUL
r30	BP_src30	ALU	ALU2
r31	BP_src31	WB	WB

Proposed solutions: phase ordering problem, discuss each possible implementation here, ordered by difficulty. Indicate that in the available timespan, we can not implement all of these solutions.

Chapter 5

Planning

Course-grain and fine-grain planning for the graduation phase. The graduation phase will take six months, so we distribute that time span over each of the phases in Table 5.1.

Table 5.1: Time table for the graduation phase.

Phase	**Tasks**	**Duration**
		2 weeks
		2 weeks
		2 weeks
Thesis Phase	Write report and collect benchmark data.	6 weeks
	Total time	24 weeks (6 months)

Chapter 6

Conclusions

Write your conclusions here.

Bibliography

[1] Y. He, Y. Pu, Z. Ye, S. Londono, R. Kleihorst, A. Abbo, and H. Corporaal, "Xetal-Pro: An Ultra Low Energy and High Throughput SIMD Processor," *in Design Automation Conference (DAC), 47th ACM/IEEE*, pp. 543–548, 2010. 1, 2

[2] L. Waeijen, D. She, H. Corporaal, and Y. He, "SIMD Made Explicit," *in Proceedings of the 13th International Conference on Embedded Computer Systems (SAMOS- XIII)*, pp. 330–337, 2013. 1, 2, 4, 5

[3] J. Janssen, "Compiler Strategies for Transport Triggered Architectures," Master's thesis, Delft University of Technology, 2001. 1

[4] D. She, Y. He, L. Waeijen, and H. Corporaal, "OpenCL Code Generation for Low Energy Wide SIMD Architectures With Explicit Datapath," *in Proceedings of the 13th International Conference on Embedded Computer Systems (SAMOS- XIII)*, pp. 322–329, 2013. 1, 3

[5] L. Zhenyuan, "Code Generation of SIMD Architecture With Automatic Bypassing," 2016. 2

[6] C. Lattner and V. Adve, "The LLVM Instruction Set and Compilation Strategy," 2002. 5

[7] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, & Tools*. Pearson Education, 2 ed., 2006. 5, 7

[8] S. Hack, D. Grund, and G. Goos, "Register allocation for programs in SSA-form," 2006. 7

Appendix A

Supported Operations

This appendix contains descriptions for each of the instructions that are available on the proposed architecture.

Table A.1: List of R-type instructions shared by the CP and PEs.

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	MUL	Shift left logical.
21	SRA	MUL	Shift right arithmetic.
22	SRL	MUL	Shift right logical.
23	ROR	MUL	Rotate right register.

Table A.2: List of I-type instructions, shared by the CP and PEs.

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	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	SLLI	MUL	Shift left logical.
21	SRAI	MUL	Shift right arithmetic.
22	SRLI	MUL	Shift right logical.
23	RORI	MUL	Rotate right register.
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 A.3: List of J-type instruction that can only the CP can execute.

Opcode	**Operation**	**Description**
0	NOP	Do nothing
1	SysCall	System call
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