Parallel instruction decoding for DSP controllers with decoupled execution units

Andreas Pettersson
Master of Science Thesis in Electrical Engineering

Parallel instruction decoding for DSP controllers with decoupled execution units:

Andreas Pettersson

LiTH-ISY-EX--19/5218--SE

Supervisor: Oscar Gustafsson
ISY, Linköping University
Andréas Karlsson
MediaTek Sweden AB

Examiner: Kent Palmkvist
ISY, Linköping University

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2019 Andreas Pettersson
Abstract

Applications run on embedded processors are constantly evolving. They are for the most part growing more complex and the processors have to increase their performance to keep up. In this thesis, an embedded DSP SIMT processor with decoupled execution units is under investigation. A SIMT processor exploits the parallelism gained from issuing instructions to functional units or to decoupled execution units. In its basic form only a single instruction is issued per cycle. If the control of the decoupled execution units become too fine-grained or if the control burden of the master core becomes sufficiently high, the fetching and decoding of instructions can become a bottleneck of the system.

This thesis investigates how to parallelize the instruction fetch, decode and issue process. Traditional parallel fetch and decode methods in superscalar and VLIW architectures are investigated. Benefits and drawbacks of the two are presented and discussed. One superscalar design and one VLIW design are implemented in RTL, and their costs and performances are compared using a benchmark program and synthesis. It is found that both the superscalar and the VLIW designs outperform a baseline scalar processor as expected, with the VLIW design performing slightly better than the superscalar design. The VLIW design is found to be able to achieve a higher clock frequency, with an area comparable to the area of the superscalar design.

This thesis also investigates how instructions can be encoded to lower the decode complexity and increase the speed of issue to decoupled execution units. A number of possible encodings are proposed and discussed. Simulations show that the encodings have a possibility to considerably lower the time spent issuing to decoupled execution units.
Acknowledgments

I would like to express my sincere gratitude to my supervisor Andréas Karlsson at MediaTek for all the help, input and interesting discussions, both on- and off-topic. No question has been too small or too big. I would also like to thank everyone else at the Linköping site for their help and for their part in making this time very enjoyable.

This thesis concludes my studies at Linköping University and these five years have been wonderful. I have learned so much and am especially grateful for all the friends I have made. You are the reason these years have been so fantastic and I wish all of you the best. Finally, I would like to thank my family for your everlasting support.

Linköping, June 2019
Andreas Pettersson
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Notation</strong></td>
<td>ix</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Processor tasks</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2 The BBP2 processor</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Problem definition</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Thesis outline</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Theory</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Computer architectures</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Single Instruction stream-Single Data stream architecture</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Scalar processor</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3 Superscalar processor</td>
<td>6</td>
</tr>
<tr>
<td>2.1.4 Very Long Instruction Word processor</td>
<td>6</td>
</tr>
<tr>
<td>2.1.5 Single Instruction stream-Multiple Data streams</td>
<td>6</td>
</tr>
<tr>
<td>2.1.6 Vector processor</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Conflicts</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Data conflict</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Structural conflict</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 Control conflict</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Digital Signal Processor</td>
<td>8</td>
</tr>
<tr>
<td><strong>3 Master core architecture design</strong></td>
<td>9</td>
</tr>
<tr>
<td>3.1 Architecture design choices</td>
<td>9</td>
</tr>
<tr>
<td>3.1.1 In-order vs. out-of-order superscalar</td>
<td>9</td>
</tr>
<tr>
<td>3.1.2 Number of superscalar ways</td>
<td>12</td>
</tr>
<tr>
<td>3.1.3 Memory and code size</td>
<td>14</td>
</tr>
<tr>
<td>3.1.4 VLIW instruction encoding</td>
<td>14</td>
</tr>
<tr>
<td>3.1.5 Superscalar instruction encoding</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Superscalar vs. VLIW</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1 Issue and stalling of instructions with uncertain latencies</td>
<td>19</td>
</tr>
<tr>
<td>3.2.2 Instruction packaging around branch targets</td>
<td>21</td>
</tr>
</tbody>
</table>
4 Master core implementation results 25
  4.1 Implementation ........................................... 25
  4.2 Compilers .................................................... 26
  4.3 CoreMark benchmark results .............................. 27
  4.4 Synthesis results ........................................... 29

5 Issue to decoupled execution units 33
  5.1 Motivation .................................................... 33
  5.2 Instruction encoding for enhanced issue ................. 34
    5.2.1 Issue speedup with grouped instructions ............. 36
    5.2.2 Fetch bandwidth and instruction buffers ............... 39

6 Conclusions and future work 45
  6.1 Conclusions .................................................. 45
  6.2 Suggestions for future work .............................. 46

Bibliography 47
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASIP</td>
<td>Application Specific Instruction set Processor</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computer</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing or Digital Signal Processor</td>
</tr>
<tr>
<td>EU</td>
<td>Execution Unit</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit</td>
</tr>
<tr>
<td>GP-DSP</td>
<td>General Purpose Digital Signal Processor</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor</td>
</tr>
<tr>
<td>IM</td>
<td>Instruction Memory</td>
</tr>
<tr>
<td>LSU</td>
<td>Load-Store Unit</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiplier-Accumulator</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RTL</td>
<td>Register-Transfer Level</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction stream-Multiple Data streams</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction stream-Multiple Tasks</td>
</tr>
<tr>
<td>SISD</td>
<td>Single Instruction stream-Single Data stream</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
</tbody>
</table>
Applications run on embedded processors are growing more and more complex. The performance of the processors must be increased, while at the same time keeping the design and production costs low, and keeping the efficiency in terms of area and power consumption high. As the application code is evolving, the processors must do so too. This thesis aims to investigate how instruction level parallelism can be utilized to improve the performance of an embedded DSP SIMT processor with multiple decoupled execution units.

1.1 Background

The architecture under investigation in this thesis is an architecture that includes a master core which fetches instructions from a single instruction stream. The instructions are decoded and issued to functional units within the master core, or to external decoupled execution units, EUs. The architecture is shown in figure 1.1.

The decoupled execution units could be hardware accelerators, other cores, or a combination. To the master core the decoupled execution units are black boxes, which should be fed with certain instructions. Stall signals are however assumed to be connected to the master core from the decoupled execution units to indicate if the execution units, as a whole or individual ones, can accept more instructions, or if the master core needs to stall certain instructions.

Some of the instructions issued to the decoupled execution units require register arguments from the master core to be passed with the issued instruction. However, to not decrease the degree of decoupling, no writes to master core registers are assumed to be possible from the decoupled units. To return data from the decoupled units, the memory is instead intended to be used, which can be read and written by both the master core and the decoupled execution units.
Figure 1.1: The architecture under investigation. The master core fetches instructions from the instruction memory, IM, and issues them to functional units, FUs, or to decoupled execution units, EUs.

The decoupled execution units could have instruction queues to allow for more instructions to be issued to them before needing to stall further issues, but could also be fully occupied by a single issued instruction. Both cases will be handled in the same way by the master core, and the master core requires no information about the internal architecture or functionality of the decoupled execution units, other than what instructions to issue where.

The decoupled execution units would typically be introduced to solve some specialized task not possible to solve, or not possible to solve as efficiently, by the master core. They are also used to increase parallel execution.

1.1.1 Processor tasks
The processor under investigation is one of many processors in a mobile phone application. The tasks of the processor includes both digital signal processing, DSP, and control tasks. Much of the DSP tasks will be issued to decoupled execution units for faster handling, while all control tasks will be handled by the master core. Some smaller DSP tasks will still be handled by the master core, so DSP instruction support in the master core is required.

1.1.2 The BBP2 processor
One processor which is organized as described above, with a master core and multiple decoupled execution units, is the BBP2 processor [8], refined from the BBP1 processor [12]. The BBP2 processor has one controller core, acting as a
master core. The BBP2 also has two decoupled SIMD execution units: One 4-way complex MAC (CMAC) and one 4-way complex ALU (CALU). The controller core issues instructions to the CMAC and the CALU. Their network connections are also configured by the controller core. Further, the BBP2 has several accelerators designed to solve specific baseband processing tasks.

The BBP2 processor is classified by Nilsson [8] as a Single Instruction stream-Multiple Tasks, SIMT, architecture. RISC instructions executed by the controller core are mixed with vector instructions issued to and executed by the SIMD execution units. The vector instructions operate on large data sets for a number of cycles, thus providing the notion of the processor doing multiple tasks at the same time. This provides parallelism and a higher performance without having to issue multiple instructions each cycle, therein simplifying the control path.

1.2 Problem definition
The SIMT execution model describes how parallelism is exploited over the master core functional units and the decoupled execution units. In its basic form only a single instruction is issued every cycle, either to a master core functional unit or to a decoupled execution unit. However, if the control of the decoupled execution units becomes fine-grained, meaning that the decoupled execution units need frequent instructions to not be starved, or if the control burden of the master core becomes sufficiently high, the fetching and decoding of instructions can become the bottleneck of the system. In order to keep all functional units and decoupled execution units busy, one needs to consider methods for how to parallelize the instruction fetch and decode process.

Traditional parallel fetch and decode methods such as superscalar and VLIW are possible methods to use to improve instruction throughput and it is of high interest to evaluate the benefits and drawbacks of these methods for the SIMT architecture. Also, it is of interest to investigate what additional improvements can be done considering that the fetch and decode pipeline in the master core can avoid deep inspection of instructions for decoupled execution units, as long as it can decide the destination unit.

The thesis will try to answer the following two questions:

- What are the benefits and drawbacks of a superscalar architecture and a VLIW architecture for a processor with a combination of DSP and control tasks?
- How can consecutive instructions meant for a decoupled execution unit be packaged and issued to increase the performance of the processor?

1.3 Thesis outline
Some classifications and concepts central to the thesis is described in chapter 2. Chapter 3 presents and discusses important aspects of the master core architecture design. The choice of master core architectures, their RTL implementations
and the synthesis and benchmark results are presented in chapter 4. Chapter 5 presents and discusses different aspects relating to issuing to decoupled execution units. Conclusions and possible future work is given in chapter 6.
This chapter will describe some concepts central to the work done. Some important computer architectures will be described, and the concept of conflicts is defined. A general description of a digital signal processor is also given.

2.1 Computer architectures
This section will describe some computer architectures and organizations central to digital signal processors and the work carried out.

2.1.1 Single Instruction stream-Single Data stream architecture
A very common architecture is the Single Instruction stream-Single Data stream, SISD, architecture, a classification described in Flynn’s Taxonomy [4]. A processor with a SISD architecture operates on a single instruction stream, fetching the instructions in order. All instructions operate on a single piece of data, and even though this could for example be multiple registers as different operands, no vector operations are performed.

There are many variants of the SISD architecture, both throughout history and today.

2.1.2 Scalar processor
A scalar processor is the simplest variant of the SISD architecture. In a scalar processor, no instruction level parallelism is exploited, only fetching one instruction at a time, decoding it, and then executing it. Most scalar processors do however have a pipeline to increase the throughput of instructions.

The advantages of a scalar processor is that it is quite easy to design and compile code for. However, as it exploits no parallelism, most programs operate much
slower on a scalar processor than on another type of processor.

### 2.1.3 Superscalar processor
A superscalar processor is a more complex type of processor than a scalar processor, but is still classified as a variant of the SISD architecture. A superscalar processor tries to dynamically schedule instructions to be executed in parallel. It still operates on a single instruction stream, but fetches multiple instructions each clock cycle and then evaluates in the decode stage if they can be executed at the same time. A number of possible conflicts, described in section 2.2, can prevent instructions from being executed at the same time.

### 2.1.4 Very Long Instruction Word processor
Another type of processor which utilizes parallelism is a Very Long Instruction Word, VLIW, processor. A VLIW processor relies on a compiler to statically determine what instructions can be executed in parallel. A VLIW processor still operates on a single instruction stream, but the stream consists of larger instruction packets packaged by the compiler. This means that no conflict checks between instructions within a packet are needed at runtime, but instead demands more of the compiler, which needs to account for the same conflicts as the superscalar processor.

### 2.1.5 Single Instruction stream-Multiple Data streams
Another architecture classification described in Flynn's Taxonomy [4] is the Single Instruction stream-Multiple Data streams, SIMD, architecture. A processor with a SIMD architecture operates on a single instruction stream, fetching instructions in order, but then using each instruction for multiple data streams. This means that every operation can operate on multiple pieces of data, allowing for fast processing of large amounts of data without increasing the code size.

### 2.1.6 Vector processor
A common kind of processor with a SIMD architecture is the vector processor. A vector processor typically have vector registers, often in combination with scalar registers. Vector registers can hold multiple pieces of data. Instructions can fetch data to vector registers quickly, and can generally operate on all pieces of data in a vector register at the same time.

Vector processors are very common in DSP applications due to the fact that they can process large amounts of data quickly, and most DSP operations should be performed on a fast stream of data.

### 2.2 Conflicts
A conflict is in this thesis defined as a conflict between instructions, where for some reason an instruction cannot be executed due to the execution of another instruction.

There are three kinds of conflicts that can arise [1], [10], namely data conflicts, structural conflicts and control conflicts. All three are described below.
2.2 Conflicts

ADD R0, R1, R2 ; R0 + R1 -> R2
SUB R2, 0x1, R3 ; R2 - 1 -> R3
AND R4, R1, R2 ; R4 & R1 -> R2

Listing 2.1: Instructions with data dependencies.

2.2.1 Data conflict
A data conflict can occur when there are data dependencies between instructions. Data dependencies can be of three different kinds – true dependency, anti-dependency and output dependency. A true dependency (or read after write, RAW) occurs when an instruction requires data produced by an earlier instruction. An anti-dependency (or write after read, WAR) occurs when an instruction overwrites a variable read by an earlier instruction. An output dependency (or write after write, WAW) occurs when an instruction overwrites a variable written by an earlier instruction.

Listing 2.1 show three instructions which exhibit all three different data dependencies. Instruction 2 has a true data dependency on instruction 1, as it requires the result produced by instruction 1 as input. Instruction 2 has an anti-dependency on instruction 3, as instruction 3 will overwrite R2 which is read by instruction 2. Instruction 3 has an output dependency on instruction 1, as both of them write to R2.

A true data dependency cannot be removed, but can in some cases be mitigated or avoided by reordering the instructions to allow for enough time between two instructions with a true data dependency for the first instruction to finish before the second instruction is started.

Output and anti-dependencies can be removed by using more registers, where the second occurrence of a destination register is changed to another, unused register. However, all following instructions that read that register also need to be changed to instead read the new register. This can be done by the compiler, or by the processor at run time. When done by the processor it is generally referred to as register renaming. As an example, in listing 2.1 above, R2 in instruction 3 could for instance be renamed to another register, thus removing the anti-dependency and the output dependency.

2.2.2 Structural conflict
A structural conflict can occur when multiple instructions want to use the same resource. To decrease the risk of a structural conflict, extra copies of resources can be added. Resources can here be for example ALUs, but also connections like register file ports. By adding extra copies of resources, instructions that would normally cause a structural conflict can be executed without a conflict, by using different instances of the resource.

2.2.3 Control conflict
A control conflict occurs when branching and the target address is unknown until the branch is executed. A control conflict also occurs when a branch is condi-
tional, i.e. when it is unknown beforehand whether or not the branch should be taken. When a control conflict occurs it is uncertain which instructions should be executed, and thus the processor will need to wait until the conflict is resolved.

2.3 Digital Signal Processor

A Digital Signal Processor, DSP, is a processor specifically designed to solve digital signal processing tasks. The degree of specialization can vary, and DSPs can be divided into two main groups – Application Specific Instruction set Processor DSPs, ASIP DSPs, and General Purpose DSPs, GP-DSPs. ASIP DSPs are more specialized than GP-DSPs, but both are more specialized than a General Purpose Processor, GPP. The specialization comes from the knowledge that DSPs will only be used for certain applications, and the architecture can be optimized accordingly. The more specific the application, the more optimizations can be done.

The nature of digital signal processing tasks is very much kept in mind when designing a DSP. For example, a common operation in DSP applications is the MAC (multiply-accumulate) operation, and much can be gained from adding special MAC units and optimizing them. Other special instructions and accelerators can also be added to a DSP.

DSPs can have varied architectures but are generally RISC-based with CISC enhancements [7]. This means that most instructions are typically single-cycle instructions which operate solely on the register file, or between the memory and the register file (load/store). The CISC enhancements can for example be special instructions with extra hardware for convolution, division or multiplication. In addition, the architecture is typically some variant of SISD or SIMD.

Throughout this thesis the acronym DSP will be used interchangeably as meaning Digital Signal Processor or Digital Signal Processing.
This chapter presents and discusses some architecture design choices which can impact the performance and design- and production costs of the master core.

Given the overall architecture and the tasks described in section 1.1, the master core is essential to the performance of the complete system. The master core handles all control instructions and issues all instructions to the decoupled execution units. This means that the master core could end up limiting the performance if it is not performing well enough. Therefore it is of interest to investigate how the performance of the master core is impacted by the choice of architecture.

### 3.1 Architecture design choices

A performance that is better than what a scalar processor generally can provide is wanted. There are multiple different architectures that could provide this. If the majority of the tasks are DSP related, a SIMD architecture might be preferable, to quickly process large amounts of data. However, as described in section 1.1.1, the tasks run on the master core is of both DSP and control nature. With more control tasks present, a SIMD architecture might be hard to fully utilize. Instead a SISD architecture might be better. This is one reason why superscalar and VLIW architectures are of most interest. However, both of these can look quite different, and thus superscalar and VLIW design choices will be investigated.

#### 3.1.1 In-order vs. out-of-order superscalar

When designing a superscalar processor one must decide if instructions always should be executed in-order, or if one can allow for out-of-order execution. Out-of-order execution means that a number of instructions are examined and possibly reordered to get less stalls due to conflicts. This is done in hardware at runtime.
Out-of-order execution can lead to more optimized code execution, with fewer idle execution units and less stalls due to conflicts. However, implementing out-of-order execution adds complexity to the hardware, both in terms of extra area and power needed for analysis, and in terms of development and design. Implementing out-of-order execution is not trivial, and introduces new problems.

One problem regards interrupts. When saving the state of the processor before running the interrupt one usually saves the address of the instruction to return to when the interrupt is done. When executing instructions out-of-order that is not enough, as more information about which instructions that have been executed already is needed, to not miss any instructions and to not execute any instruction twice. A similar argument can be made for exceptions.

The window size of an out-of-order processor determines how much the performance is increased. The window size is the number of instructions that are analyzed at the same time before reordering. This does not have to be the same as how many instructions can be issued at the same time, the window size is likely larger. The larger the window size, the better the chances are for finding independent instructions that can be moved. But as the window size goes up, the complexity of the hardware increases rapidly, as the number of dependency checks grows exponentially with the window size.

An in-order processor will never reorder instructions, but preserve the order of instructions in the program. If a conflict is detected, the instruction with the conflict and all following instructions are stalled until the conflict is resolved. This means that there will be more stalls overall, but less logic will be required.

The performance of an in-order processor is heavily dependent on the compiler. If the compiler takes into account how wide the processor is, what functional units are available, and how much latency each instruction has, it can reorder the instructions at compile time, just as the out-of-order processor would do at runtime. This will increase the performance significantly, and give similar performance as an out-of-order processor without the added hardware, but it requires an advanced compiler.

As an example for a 2-way, in-order, superscalar processor one can examine the code in listing 3.1. If we assume that all instructions only have a latency of 1 cycle, meaning the result can be used in the following cycle, we can see that the ordering is poor, due to the fact that instruction 3 is dependent on instruction 2, which is dependent on instruction 1. The same goes for instruction 6, 5 and 4. This means that the processor will have to execute the instructions as in table 3.1, where only one cycle executes two instructions in parallel. This can be improved by reordering the instructions as in listing 3.2. This new instruction order spaces out the dependencies, allowing for an execution as in table 3.2, where all cycles execute two instructions in parallel.

The reordering example above could likely be solved by an out-of-order processor during runtime, and could for an in-order processor be done at compile time. However, if the latencies of the instructions were longer it would be harder. More
Listing 3.1: Poorly ordered instructions.

Table 3.1: Execution of poorly ordered instructions in listing 3.1 in a 2-way, in-order superscalar processor.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Execute slot 0</th>
<th>Execute slot 1</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LDI 0x9, R0</td>
<td>–</td>
<td>R0 data conflict in slot 1</td>
</tr>
<tr>
<td>2</td>
<td>ADD R0, R1, R2</td>
<td>–</td>
<td>R2 data conflict in slot 1</td>
</tr>
<tr>
<td>3</td>
<td>SUB R2, 0x1, R3</td>
<td>LDI 0x5, R4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADD R4, R1, R5</td>
<td>–</td>
<td>R5 data conflict in slot 1</td>
</tr>
<tr>
<td>5</td>
<td>SUB R5, 0x1, R6</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>

Listing 3.2: Well ordered instructions.

Table 3.2: Execution of well ordered instructions in listing 3.2 in a 2-way, in-order superscalar processor.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Execute slot 0</th>
<th>Execute slot 1</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LDI 0x9, R0</td>
<td>LDI 0x5, R4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADD R0, R1, R2</td>
<td>ADD R4, R1, R5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SUB R2, 0x1, R3</td>
<td>SUB R5, 0x1, R6</td>
<td></td>
</tr>
</tbody>
</table>
instructions would have to be moved to space out the dependencies further. A formulae for the number of instructions needed between two dependent instructions (true dependency) given the latency of the first instruction and the number of ways in the processor can be formulated.

**Theorem 3.1.** The number of instructions required between two instructions, where the second instruction have a true data dependency on the first instruction, to guarantee no stalls are generated due to data dependencies is

\[ C_{\text{latency}} P_{\text{ways}} - 1 \]

where \( C_{\text{latency}} \) is the latency of the first instruction, i.e. the number of cycles before the result can be used (counting the starting cycle), and \( P_{\text{ways}} \) is the number of ways in the processor.

A compiler for an in-order superscalar processor should ideally try to use this when reordering the instructions to produce optimal code. It should also be used by an out-of-order processor when reordering the instructions at runtime. However, it should be noted that by spacing out dependencies, the number of registers required might increase. For example, looking back at the code in listing 3.1 one could reuse some of the registers, reducing the number of registers required without increasing the execution time. The same could not be done in listing 3.2.

So, the choice between in-order and out-of-order comes down to four things:

- What performance is required?
- What production cost is acceptable?
- What design cost is acceptable?
- Is an advanced compiler available or possible to design?

These questions need to be answered when making the decision. As every application has different properties the questions have no universal answer, and therefore need to be examined in each case separately.

### 3.1.2 Number of superscalar ways

Another choice to make when designing a superscalar processor is how many ways the processor should have. A way is here defined as a slot in which an instruction can be issued. A 4-way processor can for example issue a maximum of 4 instructions per cycle, but often issues less than 4 instructions due to conflicts.

The ways can be homogenous or heterogenous in terms of what instructions can be issued in each way, most likely due to differences in what functional units are available in each way. However, the ways do not have to be defined as separate slots or paths entirely, one could instead see the number of ways as how many instructions that can be decoded and issued to functional units each cycle. In this case the functional units can be seen as in a pool which the ways can pick
from. This gives more freedom when assigning instructions to ways, but requires extra muxing of signals when choosing the inputs to the functional units instead.

Most likely there will be more ways than functional units of each type. For example, a 4-way processor might have 4 arithmetic units, but only 2 multipliers, meaning that the heterogeneity comes from the fact that 4 arithmetic instructions could possibly be issued at the same time, but not 4 multiplications. For a homogenous 4-way processor there would be no such restrictions, meaning that the processor would need 4 units of each function.

As the possible performance increases with the number of ways one might think that more is always better. But extra ways come at a cost. For each added way, more dependency checks are required and the amount grows exponentially with the number of ways. The number of functional units might also have to be increased to actually get better performance. Both the extra checks and functional units will increase the area and power consumption, as well as increasing the design and verification cost. Therefore one needs to think about if the possible extra performance is required and/or can be afforded.

As more ways and dependency checks are added, more and more timing constraints are introduced. This is in some ways a worse and more complex problem. The extra constraints can reduce the possible clock frequency of the design, thus lowering the number of instructions that can be executed per second. The lower clock frequency may still be worth it, if the overall performance is increased thanks to the extra ways. If not, no extra ways should be added, unless other improvements also are made. To decrease the timing constraints one can introduce extra pipeline steps. This can allow for a much higher clock frequency, but will introduce other problems. Extra pipeline steps will make the design more complex, resulting in a higher design and verification cost. Extra steps will also introduce a higher pipeline latency, meaning it will take longer to fill the pipeline, which for example needs to be done if a branching instruction has caused the pipeline to be flushed.

Another thing to consider is that more ways do not always result in an increase in performance. A superscalar processor relies on the instruction level parallelism of the program. If there are not enough instructions in the program that can be run in parallel, extra ways might be idle most of the time. This means that the nature of the application and specific program which is intended to be run determines whether or not extra ways will increase performance.

So the choice of the number of ways and if they should be homogenous or heterogenous comes down to similar questions as in section 3.1.1, namely:

- What performance is required?
- What production cost is acceptable?
- What design cost is acceptable?
- How parallel is the application?
Just as in section 3.1.1, these questions are answered differently for each application, and need to be reflected on separately in each case.

### 3.1.3 Memory and code size

In many modern embedded processors, on-chip memories account for more than half of the core chip area. For example, the BBP2 processor has a memory area of about 55% of the core area [8]. Furthermore, memory access often account for more than half [5], or even as much as 70% [3], of the total power consumption of the chip. These numbers indicate that reducing the size of memories can greatly reduce the total chip area as well as reducing the power consumption.

Memories can be shared by both instructions and data, but can also be split into data memories and program memories. If the code size of the programs running on the processor is reduced, the size of the program memory, or the combined memory, can also be reduced. One way to reduce the code size can be to optimize the instruction encoding. Another way is to alter the application code and the compiler to aim for a smaller code size.

### 3.1.4 VLIW instruction encoding

When designing a VLIW processor the instruction encoding can greatly impact the code size and also somewhat the decoding complexity, due to the fact that the instruction parallelism needs to be explicitly defined in the VLIW instruction. Therefore one should carefully consider the instruction encoding.

**Fixed instruction length**

In some sense the simplest approach is to have a VLIW instruction word that have one fixed sized slot for every functional unit. This could look like in figure 3.1a, if the processor has 2 ALUs, 1 LSU (load/store unit) and 1 branch unit. If a functional unit cannot be used, due to lack of a certain type of instruction or due to conflicts, a NOP is inserted in that slot instead. This encoding is very fast to decode, as no muxing between the slots to different functional units is required. However, in a more complex processor there will likely be many more functional units such as multipliers, dividers and additional ALUs and LSUs. In this case the VLIW instruction would become extremely long, and most slots would be empty (with NOPs inserted) which would make the code size very large. A grouping of functional units could then be done, with one slot for each group. This would still be quite efficient as a functional unit will always get its inputs from the same slot, but this would require more of the compiler when placing instructions into the slots, unless all functional unit types are available for each slot.

All functional units can also be shared by all slots. In that case the compiler do not have to place instructions into any specific slot, resulting in unspecific, general slots. For total freedom all types of functional units would have as many instances as there are slots, but that is often not preferred as it can be expensive and would not necessarily increase performance considerably. Instead one can have fewer functional units and mux between the different slots depending on the instructions in them. This could then look as in figure 3.1b for a 4-way VLIW processor. In this case the slots still have a fixed size. The fixed size allows for
3.1 Architecture design choices

(a) VLIW encoding with fixed slot size, fixed number of slots and specific slots.

(b) VLIW encoding with fixed slot size and fixed number of slots.

(c) VLIW encoding with fixed slot size, variable number of slots, and a field for number of slots.

Figure 3.1: Three possible VLIW encodings with fixed slot size.

simpler encoding and decoding as all slots can be decoded at the same time.

The encoding in figure 3.1b will still generally have quite bad code size due to added NOPs. To get rid of the NOPs one could instead have a variable number of slots, and instead of adding a NOP when not being able to populate a slot, the slot can be removed. But the information about which instructions that can be run in parallel still needs to be kept somehow. One way to solve it would be as in figure 3.1c, where an extra field is added as a header to the VLIW instruction. The header holds how many slots are in the VLIW instruction. This introduces some extra checks when decoding but can decrease the code size drastically. An upper limit on the number of slots would be needed, as the processor will still have an upper limit on how many instructions it can decode at a time. A lower limit can be imposed (most likely 1), but is not necessary. Depending on the upper limit, a lower limit can possibly save a bit in the header as a length of 0 for example do not have to be possible. If no lower limit is imposed, the length of 0 can instead be used to make the processor skip one cycle. This could be the same as inserting NOPs in all slots. Depending on how often this is used it could save more overall code size than the possibly saved bit in the header.

Variable instruction length

All encodings described so far have a fixed slot size. A fixed slot size can produce a faster and smaller decoder thanks to the fact that the start position of each instruction is known, as it means that all slots can be decoded in parallel and that no large amount of muxing is required. However, having all instructions being the same length produces an unnecessarily large code size. If one allows for instructions of different lengths the code size can be decreased.

Given an instruction set, an application program and the goal to minimize the overall code size, more common instructions will get shorter encodings, and more uncommon will get longer ones. Further optimizations can be done by altering the instruction set, by adding special variations of some instructions and giving
(a) VLIW encoding with variable slot size with size information in slots, and fixed number of slots.

(b) VLIW encoding with variable slot size, fixed number of slots, and a field for start position of each slot.

(c) VLIW encoding with variable slot size with size information in slots, variable number of slots, and a field for number of slots.

(d) VLIW encoding with variable slot size, variable number of slots, a field for number of slots, and a field for start positions of each slot.

Figure 3.2: Four possible VLIW encodings with variable slot size.
them their own encoding. For example, a very common instruction in most programs is to push a register to the stack. This is in reality a store instruction with the address specified in the stack pointer register. So, a push instruction could be added where only the source register needs to be specified, and where the stack pointer is always used as the address pointer. Such common, specialized instructions can decrease code size further, as the new instruction can get a shorter encoding. Some restrictions can be imposed when optimizing the instruction lengths. A common restriction is to have the lengths be a variable number of full bytes, and possibly disallowing certain number of bytes. This will of course decrease the potential gain of the optimization but can make fetch and decode easier.

A possible encoding with variable instruction length is shown in figure 3.2a. Here the number of slots is fixed but the slots have variable length, as long as the instructions they hold. The size of each instruction is located as a header for each instruction. The drawback of variable-length instructions becomes apparent here. All four slots should be decoded and run in parallel but that is not possible as the start positions of the slots are unknown. To find the second instruction one must first determine the size of the first. For the third instruction, the size of both the first and the second instruction must be extracted before the start of the third instruction can be determined. This problem grows with the number of slots.

A way to decrease the impact of the unknown start positions is to encode the VLIW instruction as in figure 3.2b, which has a header with the start positions of all slots. If the slot start position fields have a fixed size, the start positions can be read in parallel, decreasing the time needed to determine the location of every instruction. The instructions can then also be read in parallel. Information about the size of each instruction can still be placed in the instruction encoding as in figure 3.2a, as this can simplify the encoding.

The encodings in figure 3.2a and figure 3.2b both suffer from the same problems as the fixed-length instruction encodings with a fixed number of slots, that NOPs often needs to be inserted. However, the impact will not be as large as for the fixed-length instructions due to the fact that NOP can have a short encoding. Even so, the NOPs can be removed, resulting in an encoding as in figure 3.2c where the number of slots is variable. Like for the fixed-length instructions, a field with information about the number of slots present is added. The problem with the slot start positions arises again. Therefore an encoding like in figure 3.2d can be produced, where once again the slot start positions are specified in a separate field. The number of slots present can dictate the length of the slot start positions field, or not if one wants to keep the logic simple, as the start position then would have to once again be calculated, but can at least be done so in parallel for the different slots.

Variations
Variations to the encodings above, as well as completely different ones, can of course be used. The ones mentioned are in no way the only ones, nor are they
necessarily the best ones, merely a natural evolution described.

### 3.1.5 Superscalar instruction encoding

When designing a superscalar processor it is important to consider the instruction encoding, just like for a VLIW processor described in section 3.1.4. The difference between superscalar code and VLIW code is that there is no explicit parallelism in the superscalar code, the possibility to run instructions in parallel is up to the processor to examine. Therefore superscalar code only consists of individual instructions.

A simple instruction encoding is shown in figure 3.3a. Here fixed-length instructions are used. A superscalar processor wants to fetch and examine multiple instructions at once. A fixed-length instruction encoding allows for easy fetch and fast decode thanks to the known start position of each instruction, and multiple instructions can be decoded in parallel. Compared to fixed-length VLIW instruction encodings, superscalar code is typically much smaller, especially for VLIW encodings where NOPs need to be inserted. However, superscalar code size can be reduced as well. This can for example be done by allowing variable-length instructions.

Figure 3.3b shows a variable-length instruction encoding with the size of each instruction specified at the start of the instruction. Other possibilities to determine the size are of course possible, such as having the last bit of every byte determining if the next byte is a new instruction or is a part of the previous instruction. In any case, the variable size can decrease the overall code size, just like for VLIW, where common instructions get shorter encodings. Specialized common instructions can further decrease the code size. However, just like for VLIW, the unknown instruction length is problematic when trying to decode multiple instructions at once. The size of the first instruction needs to be determined before the position of the second instruction can be known. This introduces extra timing constraints. Some things can however be done to mitigate this. One possibility is to decode all possible start positions and then choosing the correct ones once known. This is effective but can be very expensive in terms of area and power cost.

One variable-length instruction encoding developed specifically for parallel fetch and decode is the Heads and Tails, HAT, encoding [9]. HAT-encoding splits all instructions into a fixed-length head and a variable-length tail. The instructions are then packaged into bundles. If a cache is used, the width of the bundles should be the same as the width of the cache lines. Two example bundles can be seen in figure 3.4. First in the bundle is a field specifying the number of instructions in the bundle minus one, then comes the fixed-length heads followed by potentially an empty region and then the tails in reverse order, with the first tail starting at the last bit of the bundle. Instructions are bundled as many as can fit or until the number of instructions field is saturated. Restrictions on the length of the tails can be set, and one can allow some instructions without tails, or not. Due to the variable length of the instructions it is not always possible to perfectly fit them in a bundle, instead a region between the heads and the
3.2 Superscalar vs. VLIW

Superscalar and VLIW processors have both generally better performance than a scalar processor, but costs more to design and produce. They do not however always perform better, and they have some particularities which makes them perform differently in certain cases. These particularities should be kept in mind when choosing between a superscalar and VLIW processor. Some of the particularities are described below.

3.2.1 Issue and stalling of instructions with uncertain latencies

Data dependency is one of the main reasons for stalling an instruction. One very common situation in all applications are loads from memory followed by operations on the loaded data. The operation instructions have a data dependency on the load instructions and might therefore result in stalls.
Listing 3.3: Instruction sequence with high potential for stalling due to uncertain load latencies.

Table 3.3: Execution of instructions in listing 3.3 in an in-order scalar processor.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Execute slot</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD [R0], R1; [R0] -&gt; R1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADD R2, 0x4, R2; R2 + 4 -&gt; R2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LOAD [R2], R3; [R2] -&gt; R3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADD R1, R4, R5; R1 + R4 -&gt; R5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CMP 0, R3; 0 - R3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>stall</td>
<td>R1 data conflict</td>
</tr>
<tr>
<td>21</td>
<td>ADD R1, R4, R5</td>
<td>Load to R1 done</td>
</tr>
<tr>
<td>22</td>
<td>stall</td>
<td>R3 data conflict</td>
</tr>
<tr>
<td>23</td>
<td>CMP 0, R3</td>
<td>Load to R3 done</td>
</tr>
</tbody>
</table>

Table 3.4: Execution of instructions in listing 3.3 in a 2-way, in-order superscalar processor.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Execute slot 0</th>
<th>Execute slot 1</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD [R0], R1</td>
<td>ADD R2, 0x4, R2</td>
<td>R1 data conflict in slot 1</td>
</tr>
<tr>
<td>2</td>
<td>LOAD [R2], R3</td>
<td></td>
<td>R1 data conflict</td>
</tr>
<tr>
<td>3</td>
<td>stall</td>
<td></td>
<td>R1 data conflict</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>stall</td>
<td></td>
<td>R1 data conflict</td>
</tr>
<tr>
<td>21</td>
<td>ADD R1, R4, R5</td>
<td></td>
<td>Load to R1 done, but not to R3</td>
</tr>
<tr>
<td>22</td>
<td>CMP 0, R3</td>
<td></td>
<td>Load to R3 done</td>
</tr>
</tbody>
</table>
In order to mitigate the effect of load latencies, the instructions can if possible be reordered to allow for independent instructions to be executed while waiting for the load to finish. If the load latency is known at compile time this can be quite effective. However, if the load latency is unknown, or uncertain, the compiler will not know how to schedule the instructions optimally. An unknown load latency can arise in a processor with a memory hierarchy, which are present in most processors, where for example a cache or multiple memories at different latencies are used. In the unknown case, the compiler must fall back to a default latency.

Another common situation in most applications is that there is a load instruction with the address calculated by an earlier arithmetic instruction. This in itself is not too bad, but in combination with the previously mentioned load situation things can get problematic.

Listing 3.3 is an example of the situation described above. Instruction 4 is dependent on the load in instruction 1, and instruction 3 needs the result from instruction 2 as an address. Finally, instruction 5 needs the result from the load in instruction 3. During scheduling, the compiler in this example assumed that the latency of both loads is 1 cycle, but will in reality be 20 cycles. This discrepancy could come from the compiler assuming that the values are cached, where a load from the cache might take 1 cycle, but where a cache miss might take 20 cycles. Arithmetic operations have a latency of 1 cycle.

When executing the instructions on an in-order scalar processor they will execute as in table 3.3. Due to the longer than anticipated load latencies, multiple stall cycles will be introduced. In this example processor, multiple loads can be active at the same time. The total execution time will be 23 cycles.

The same program executed on a 2-way, in-order superscalar processor will execute as in table 3.4. This will look mostly like the scalar processor, only moving one addition, resulting in a total execution time of 22 cycles.

If the instructions in listing 3.3 are recompiled for a 2-way VLIW processor, using the same scheduling rules, the instruction sequence in listing 3.4 might be produced. When executing this on a VLIW processor which also have the 20 cycle load latency it will look as in table 3.5. Due to the processor not being able to separate the slots in the VLIW instructions, the second load cannot be executed until the data dependency in the addition is resolved. This results in no overlap of the load latencies, which in turn results in a total execution time of 42 cycles.

If the load latencies would have been 1 cycle as assumed by the compiler, there would be no big difference between the superscalar and VLIW processors. But when the load latencies are unknown or uncertain, problems like these can be introduced, and the scheduling becomes more difficult.

### 3.2.2 Instruction packaging around branch targets

Branching can in many situations prove to be problematic. One such problem is that a branch instruction will affect the packaging of VLIW instructions around
Listing 3.4: VLIW instructions with high potential for stalling due to uncertain load latencies.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Execute slot 0</th>
<th>Execute slot 1</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD [R0], R1</td>
<td>ADD R2, 0x4, R2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>stall</td>
<td>–</td>
<td>R1 data conflict</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>stall</td>
<td>–</td>
<td>R1 data conflict</td>
</tr>
<tr>
<td>21</td>
<td>LOAD [R2], R3</td>
<td>ADD R1, R4, R5</td>
<td>Load to R1 done</td>
</tr>
<tr>
<td>22</td>
<td>stall</td>
<td>–</td>
<td>R3 data conflict</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>stall</td>
<td>–</td>
<td>R3 data conflict</td>
</tr>
<tr>
<td>42</td>
<td>CMP 0, R3</td>
<td>–</td>
<td>Load to R3 done</td>
</tr>
</tbody>
</table>

Table 3.5: Execution of instructions in listing 3.4 in a 2-way VLIW processor.

the target address. VLIW instructions are statically defined by the compiler and express explicit parallelism. If the execution of an instruction is uncertain it must often be separated from other instructions.

An example of when this behavior exhibits itself can be seen in listing 3.5. In this code there are two conditional branch instructions with target addresses 13 and 16 which both lies in a region with independent instructions, which ideally should be run as much in parallel as possible. When run on a superscalar processor this will not be an issue since the instructions are scheduled dynamically. But there will be an issue for a VLIW processor. The VLIW compiler needs to take the target addresses into account when packaging the instructions, because of the note that a branch cannot end up in the middle of a VLIW instruction, only at the start. For the examined instruction sequence, the equivalent VLIW code would be as in listing 3.6. One can see that some instructions remain by themselves, even though no dependencies are present, as the target addresses needs to be at the start of an VLIW instruction.

The instruction packaging around branch targets is a problem both during compilation and when executing the code. A VLIW compiler needs to adjust VLIW instructions to fit branch targets, making the compiler slightly more complex. When executing the code, the separation of VLIW instructions will cause fewer instructions to be run in parallel, thus increasing the execution time.

A way to relieve the need to separate instructions around branch targets would be to specify in the branch instruction what slots of the target VLIW instruction
3.2 Superscalar vs. VLIW

should be executed. This means that the compiler would alter the branch instruction depending on the target location instead of separating the instruction at the target. This in turn means that the code might execute faster, as more instructions can be executed in parallel when only passing by a branch target. The complexity of the compiler will be roughly the same, but the complexity of each branch instruction will be increased, which the hardware must account for. This could prove to be costly.

Listing 3.5: Instruction sequence with branch targets.

Listing 3.6: VLIW instruction sequence with branch targets.
The benefits and drawbacks of two general architectures – superscalar and VLIW – were discussed in the previous chapter. In order to further compare these two architectures, they were implemented. The goal of the implementations was to measure the actual performance and logic area of superscalar and VLIW processors. As discussed in the previous chapter, there are many variations of both architectures. As the time was limited and the design space is very large, only one representative design of each architecture was implemented and examined.

4.1 Implementation

Given the overall architecture, the tasks and the architecture design choices described in the previous chapters, two interesting architectures were chosen to be implemented.

The first architecture is a 2-way in-order superscalar architecture, with variable-length instructions varying up to 8 bytes.

The second architecture is a 2-way VLIW architecture, with variable-length instructions up to 8 bytes long, and with up to 2 slots. The reason for this choice of architecture was that a similar architecture was available from a previous project. Only some changes had to be applied to it.

An in-order scalar architecture was used as a baseline for comparison.

The architectures were chosen quite similar in order to be able to compare them more effectively. For instance, the instruction set is very much the same. Also, if either architecture would have more ways than the other it would be harder to compare the performance and possible throughput. In contrast to this, there are some earlier studies comparing superscalar and VLIW processors, and also
comparing them to SIMD vector processors [6], [11]. These studies show large performance differences between the processors, where the VLIW processors outperform the superscalar processors. However, neither study accounts for the fact that the processors under examination have different number of ways, where the VLIW processors have considerably more ways than the superscalar processors. Therefore it is of interest to instead compare two architectures with the same number of ways. However, the cost and performance might not scale the same with the number of ways for both architectures, so there are still arguments to be made for comparing architectures with varying number of ways.

The two architectures under investigation were implemented in RTL. During the course of the implementation a number of tests were performed to verify the functionality. These were however not exhaustive and the implementations are not confirmed to work in all cases. Shortcuts were also taken in some cases in the terms of disallowing certain instruction combinations to avoid problematic behavior that would be possible to sort out but would take more time than what was available.

When the implementations were done, the CoreMark [2] benchmark was run on both processors, as well as the scalar processor. The RTL implementations were also synthesized to ASIC to get the area and timings of the implementations. From the timings the synthesis tool also reports the maximum possible frequency that could be used.

4.2 Compilers

In order to run code on the implemented processors, two compilers were needed, one for each processor. For the VLIW processor, a compiler from a previous project could be used.

For the superscalar processor no compiler was available. To save time however, the VLIW compiler was modified in such a way that it would only output VLIW instructions with one slot, and with header information removed, leaving only the actual instruction. As these instructions are individual instructions they can be run in the superscalar processor. This same reworked compiler could be used for the scalar processor as well.

This solution have a problem. The VLIW compiler is optimized and will produce near optimal instruction sequences for the VLIW processor, whereas the superscalar compiler will not. As the superscalar compiler believes that it is compiling for a VLIW processor with a single slot, it does not take into account that multiple instructions can be run in parallel. In other words, it does not produce optimal or near optimal instruction sequences as described in section 3.1.1 and theorem 3.1. This will most likely cause more instructions to be stalled and fewer instructions to be run in parallel. The same issue arises with loop unrolling, the compiler will not unroll optimally for the superscalar processor, or most likely not unroll at all.

Engineering effort has been spent to optimize the instruction encoding for the VLIW processor. This was not possible for the superscalar processor due to time
Table 4.1: CoreMark results for the superscalar and VLIW processors in relation to a scalar processor.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Scalar</th>
<th>Superscalar</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total cycles</td>
<td>1</td>
<td>0.900</td>
<td>0.825</td>
</tr>
<tr>
<td>Stall cycles</td>
<td>0.170</td>
<td>0.191</td>
<td>0.189</td>
</tr>
<tr>
<td>1-way issue cycles</td>
<td>0.830</td>
<td>0.588</td>
<td>0.431</td>
</tr>
<tr>
<td>2-way issue cycles</td>
<td>0</td>
<td>0.121</td>
<td>0.205</td>
</tr>
<tr>
<td>Code size</td>
<td>1</td>
<td>1</td>
<td>1.016</td>
</tr>
</tbody>
</table>

limitations. The VLIW processor instruction encodings were, mainly, also used in the superscalar processor. This is not ideal, a new optimization should have been done to produce more efficient encodings for use in the superscalar processor.

4.3 CoreMark benchmark results

To compare the two architectures the CoreMark [2] benchmark was used. CoreMark is typically used to measure the performance of processors in embedded systems. It includes operations for list processing, matrix manipulation, state machines, and CRC (cyclic redundancy check).

CoreMark was compiled using the compilers for the two processors and then run. CoreMark was also compiled for and run on the scalar processor. As the same compiler was used for both the scalar and superscalar processor, the two processors execute exactly the same code.

Table 4.1 shows the results for the superscalar and VLIW processors relative to the results for the scalar processor. A 2-way issue cycle is for the superscalar processor a cycle where two instructions are found to be without conflicts and are issued at the same time. A 1-way issue cycle for the superscalar processor is instead a cycle where only one instruction could be found to be without conflicts and therefore only one instruction can be issued. Also, for the superscalar processor, a stall cycle is a cycle where no instruction can be issued due to conflicts. For the VLIW processor, a 2-way issue cycle is a cycle where the processor issues a VLIW instruction with two slots, and a 1-way issue cycle is instead a cycle where the processor issues a VLIW instruction with only one slot. A stall cycle, for the VLIW processor, is a cycle where the current VLIW instruction cannot be issued due to conflicts. For the scalar processor, a 1-way issue cycle is a cycle where an instruction is without conflicts and can be issued, and a stall cycle is a cycle where an instruction cannot be issued due to conflicts. A 2-way issue cycle is not possible for a scalar processor.

One can see that the total number of cycles is lower for both the superscalar processor and the VLIW processor than for the scalar processor. The number of stall cycles are however higher for both the superscalar and the VLIW processor. The decrease in overall run time instead comes from the fact that instructions
can be run in parallel, which can be seen in the decrease of 1-way issue cycles, and that 2-way issue cycles are present. One can also clearly see that the VLIW processor runs in fewer cycles, partly because of fewer stall cycles, but primarily due to the higher rate of 2-way issue.

The relation of 2-way issue cycles indicates that the superscalar processor cannot dynamically schedule as many instructions in parallel as the VLIW compiler have explicitly specified in the VLIW code. There are a couple of things that will contribute to this. One thing is the shortcuts taken during implementation of the superscalar processor that disallows certain instruction combinations to be run in parallel, forcing them to be run sequentially.

Another reason for the lower rate of 2-way issue is the compiler used for the superscalar processor, described in section 4.2. This is probably the main reason, as the instruction sequences produced are suboptimal, especially since the superscalar implementation is in-order.

The number of stall cycles is higher for both the superscalar and the VLIW processor. This is probably due to the fact that the dynamic scheduler for the superscalar processor and the VLIW compiler will schedule as many instructions as possible in parallel, even though that may incur stalling in a following cycle. The scalar processor instead executes one instruction at a time and may therefore avoid some stall cycles. An example of this can be seen in listing 4.1. If multiplications are assumed to have a latency of 2 cycles and additions have a latency of 1 cycle, a superscalar processor will schedule instruction 1 and 2 to be run in parallel. This will then result in a one cycle stall due to the data dependency between instruction 3 and instruction 1. A scalar processor would instead run one instruction at a time, removing the need to stall before instruction 3. The total number of cycles might still be same for the scalar as for the superscalar, but the number of stall cycles will be higher for the superscalar. The same thing might happen for the VLIW processor depending on the scheduling rules of the compiler, as the scheduling is done at compile time.

Even though both the superscalar and the VLIW processor have more stall cycles than the scalar processor, the number is higher for the superscalar processor. One explanation could once again be the compiler producing suboptimal instruction sequences, not spacing out dependencies enough. However, one could argue that the number of stall cycles should be lower than for the VLIW processor due to better handling of long latency instructions like loads, as discussed in section 3.2.1. But this would require such a situation to be present in the benchmark code,

**Listing 4.1:** Instruction sequence that might cause a stall in a superscalar processor, but not in a scalar processor, when multiplications have a 2-cycle latency and additions have a 1-cycle latency.

```plaintext
1  MUL R0, R1, R1 ; R0 * R1 -> R1
2  ADD R2, 0x4, R2 ; R2 + 4 -> R2
3  ADD R1, R4, R5 ; R1 + R4 -> R5
```
which is not certain. A more thorough profiling of the code would be required to answer this question with more certainty.

One aspect where the scalar and the superscalar processors get a better value than the VLIW processor is code size, which is approximately 1.6% larger for the VLIW processor. One explanation for this is that the VLIW code have more unrolled loops, which increases the code size but also increases the number of instructions that can be run in parallel. A better superscalar compiler would unroll more loops, but would hopefully also have closer to optimal encodings. More unrolled loops would probably also make the superscalar code size larger than the scalar, since there is not as large a need in the scalar processor for unrolls as in the superscalar processor.

### 4.4 Synthesis results

To compare the cost and further compare the performance of the superscalar architecture with the VLIW architecture the two implementations were synthesized to ASICs.

Prior to starting the synthesis a frequency goal is set. The synthesis tool will try to reach this goal. If the goal is set high, multiple optimizations needs to be done, like duplicating logic to run things in parallel. The higher the goal the more optimizations are tried, and the synthesis time will increase. If the goal is set too high, the synthesis tool might not be able to produce a solution with the specified frequency, but instead gives the best one it found. This might not be the best one possible, but is simply the best one the synthesis tool found.

The area of the synthesized logic will heavily depend on the frequency goal. The area will generally increase with the frequency goal, as logic is duplicated. Therefore, to compare the area of two designs a frequency goal that is achievable by both designs should be set.

The synthesis tool used is not completely reliable in the sense that very minor changes in the code can affect the resulting area or possible frequency. The changes might be as minor as moving a line of code, even without changing it. This presents a problem when trying to compare two designs. It is unclear if it is the nature of the designs or the particularities of the synthesis tool that is the reason for differences in the synthesis results. This should be kept closely in mind when examining synthesis reports.

Figure 4.1 shows the relative areas produced by the synthesis tool for the VLIW and the superscalar processors, normalized with the first area of the superscalar design. All frequency goals were achieved by both designs. The highest frequency goal is close to the possible frequency limit of the superscalar processor.

One can see that the area is mostly somewhat larger for the superscalar processor. The differences are however so small that it is difficult to discern if they are due to the differences in the designs or due to synthesis tool particularities. The fact that the superscalar design approaches its possible maximum frequency is likely
Figure 4.1: Relative ASIC area for the VLIW and the superscalar processors. All frequencies were achieved by both designs.

Table 4.2: Synthesis results for the superscalar and VLIW processors in relation to each other.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Superscalar</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible frequency, with unachievable frequency goal</td>
<td>1</td>
<td>1.196</td>
</tr>
</tbody>
</table>

the reason why the area increases faster for the superscalar design than for the VLIW design.

Interestingly, for the lowest frequency goal the superscalar design gets a smaller area than the VLIW design. This is most likely due to synthesis tool particularities. The area of the VLIW design also dips for the second frequency goal, which further suggests that the first two values do not give an accurate representation of the relationship between the two designs.

Table 4.2 shows the relative timing results for the superscalar and the VLIW processors, normalized with the results for the superscalar processor. One can see that the VLIW processor can reach a higher frequency than the superscalar processor, when synthesizing with a high unachievable frequency goal, the same for both designs. This difference is larger than the area difference and can therefore with more certainty be believed to be due to the difference in architecture. One reason for the lower possible frequency can be the extra dependency checks in the superscalar processor which are not present nor needed in the VLIW processor. If this is the limiting factor it will grow more limiting with the number of
ways, and will do so exponentially as the number of checks grows exponentially with the number of ways.

The VLIW processor has had some timing improvements made but no such improvements have been made for the superscalar processor. This is another likely reason why the possible frequency is higher for the VLIW processor. Such improvements should ideally be investigated and introduced if not too expensive, if a higher clock frequency is required. One such improvement could be to increase the number of pipeline stages, thus making the timing constraints less limiting. However, adding pipeline stages will have other side effects. For example, a longer pipeline means that more time is spent filling it, and the performance loss is greater each time the pipeline has to be flushed.
This chapter will investigate how instructions can be encoded to allow for wide issue from a master core to decoupled execution units. A motivation why this is interesting is first provided, and several possible encodings are then presented and discussed.

5.1 Motivation

For the modem stage of a baseband processor, Nilsson [8, p. 36-38] identified four interesting properties:

- Complex computing: Most computations are performed on complex valued data.
- Vector property: A large portion of the computations is performed on long vectors of data.
- Control flow: The control flow is predictable to a large extent.
- Streaming data: The processor operates on streaming data.

These properties still hold to some extent. Complex computations are still the main part of all computations and the data is still streaming. However, in some situations all of the above mentioned properties do not hold. For examples, the length of the vectors of data may be shorter and the control flow may be less predictable.

The SIMT architecture and the BBP2 processor got its good performance and parallelism from the execution units operating on long vectors of data, and could therefore afford to issue only a single instruction per cycle to the SIMD execution units. If the vectors shrink and the control portion grows, the issue pace should
increase to keep the same parallelism in a processor with decoupled execution units. It is therefore of interest to investigate how to encode instructions to allow for a wider issue.

### 5.2 Instruction encoding for enhanced issue

The first step when decoding instructions in a processor with one or multiple decoupled execution units is to determine if the instruction should be executed in the master core or, if not, in which of the decoupled execution units. The instruction encoding can make this easier or harder.

Figure 5.1 shows a common instruction encoding in a scalar or superscalar processor, or as a subinstruction in a VLIW instruction package. It specifies the operation which should be performed, possibly with some immediates in the operation field, and maybe some register arguments defined. The length can be fixed or variable. In a single executor environment this is a relatively good encoding, as all information is used in the same unit as where it is decoded. However, in a multiple executor environment this is not always the case. If the instruction is meant for another execution unit, the complete operation still needs to be decoded to determine where it should be issued. This is unnecessary, and makes the decoding logic more complex in the master core, while probably still requiring decoding logic in the decoupled execution units.

To relieve the need to decode the complete instruction in the master core some information is needed earlier. As proposed by e.g. Tell [12, p. 69], a number of bits at the start of the instruction can indicate which execution unit it is meant for. If the unit is the master core, the instruction should be decoded further, otherwise it should be issued to the correct execution unit. With the added unit field, the instruction can look like in figure 5.2. The fact that the operation field is greyed out means that the master core does not need to decode this field if the instruction is meant for another execution unit.

The encoding in figure 5.2 can be expanded into a variable-length instruction encoding as in figure 5.3. The operation field can be longer, or shorter, and the length of it is specified in a size field. The size field is required as no other information about the length is available until the operation field has been decoded. If the instruction is meant for the master core, no size field is necessary as the master core can read the instruction stream. The unit field can then be followed
Figure 5.3: Instruction encoding with a unit field and a variable size, specified in a size field.

Figure 5.4: Instruction encoding with a unit field and master core register arguments specified.

Figure 5.5: Instruction encoding with a unit field and a variable number of master core register arguments specified, with the number of registers specified.

Figure 5.6: Instruction encoding with a unit field, master core register arguments specified, and a variable size specified in a size field.

directly by the operation, which can be encoded in a number of different ways.

A problem that arises with the encodings in both figure 5.2 and figure 5.3 is if the decoupled execution unit that the instruction is meant for requires some registers from the master core as input arguments. To not again require the complete instruction to be decoded to determine the register arguments a new field can be introduced. Figure 5.4 shows an encoding where the unit field is followed by a master core register field, specifying which registers should be sent to the execution unit when issuing the instruction. The operation field can still hold information about arguments which are local to the execution unit the instruction is meant for. As the number of master core registers are not specified, a fix number of registers needs to be specified. One possibility is to instead have a few bits determining if a register specified actually should be sent with the issue or not. Another solution is to do as in figure 5.5, where the number of registers is specified in another field. Just like for the size field, the register fields can be left out, or in reality moved, when the instruction is meant for the master core.

To allow for a variable length operation field, a size field can be added to the encodings with the register fields, resulting in encodings as in figures 5.6 and 5.7.

All of the above encodings can be used in superscalar processors as any other instruction. They can also be used in VLIW processors, inserted into slots in VLIW instructions. Another possibility for VLIW processors is that the VLIW
instructions as a whole are meant for the master core or a single execution unit, in other words having the unit, register and size fields be a header for the VLIW instructions. The operation field can in itself also hold multiple operations that can be separated by the decoupled execution units. Especially the encodings with a size field can allow for multiple operations to be grouped into a single instruction. In this way, consecutive instructions with only one operation, which all should be executed by an execution unit, can be grouped into a single, longer instruction with multiple operations.

Some limits on the size and the number of registers will have to be imposed in a real system, to have the number of bits of some fields be known. Other properties of the system will also limit the sizes, such as the number of read ports on the master core register file and the issue bandwidth from the master core to the decoupled execution units.

The limitations based on the number of read ports and issue bandwidth will limit the issue speed, but may necessarily not actually limit the number of register arguments or the length of the instruction, if one allows for a multi-cycle issue. The register file can be read over many cycles and the read data and the instruction data can be issued over many cycles. This can lead to very compact code as the number of header fields are minimized. They are however required to be wider which can increase the code size if the instructions meant for decoupled execution units cannot be grouped and issued together in most cases.

Another issue arising when issuing at a very high speed can be that the fetch bandwidth is too narrow. This is heavily dependent on the processor architecture. For example, an issue bandwidth wider than a fetch bandwidth may be impossible to utilize. However, if buffers are used, a wide issue bandwidth may be useful, as most cycles probably will not consume all of the fetch bandwidth, thus allowing for some cycles at a time to issue more than the fetch bandwidth without problems.

If the fetch bandwidth is wide but the issue bandwidth is limited, a multi-cycle issue may be done in parallel with master core instructions following the instruction being issued, as long as no dependencies are present.

### 5.2.1 Issue speedup with grouped instructions

As discussed above, consecutive single-operation instructions meant for a decoupled execution unit can be grouped into longer instructions. A grouped instruction could look like in figure 5.7, with all operations specified in the operation field. By grouping operations, the issue speed can possibly be increased. Fewer
**Figure 5.8:** Comparison of the number of EU issue cycles in a typical program for varying upper limit of grouping EU instructions. $B$ is the issue bandwidth in bytes per cycle and $p$ is the number of register arguments that can be read and issued per cycle. Registers can be issued in parallel with instruction data, and fetch bandwidth is assumed to be larger than the issue bandwidth. The average number of register arguments per instruction is assumed to be 0.5 and the average size of the instructions are assumed to be 4 bytes.

headers need to be decoded, while still issuing the same operations to the execution units.

To calculate the possible speedup gained from grouping instructions a number of simulations were run. Code from a typical program was analyzed and it was found that the number of consecutive single-operation instructions issued to a decoupled execution unit range from 1 instruction up to 23 instructions, with the frequency dropping quickly with a growing number of consecutive instructions. These statistics were used in the simulations.

Other factors that impact the number of issue cycles to decoupled execution units are given above. Some assumptions were made regarding these to simplify the simulations. The average size of an instruction meant for a decoupled execution unit was assumed to be 4 bytes, and the average number of register arguments
per instruction was assumed to be 0.5. The fetch bandwidth was assumed to be larger than the issue bandwidth, and would therefore not affect the simulations. Register arguments were assumed to be able to be issued at the same time as instruction data. Multi-cycle issues were allowed.

To investigate the impact of the issue bandwidth and the register argument read and issue limit, five different bandwidths and register argument limits were used. The issue bandwidth was always set to 8 times the register argument limit. This meant that neither would be more limiting than the other given that the average instruction size to average number of register arguments relation was 4 to 0.5, also a factor of 8 difference.

The instruction encodings in the simulations were assumed to be similar to the encoding in figure 5.7, to allow for a variable size. The upper limit of single-operation instructions that could be grouped and encoded in the same instruction was varied from 1 up to 23.

The result of the simulations can be seen in figure 5.8. The graph shows that an issue speedup is clearly gained from grouping instructions. However, it also shows that the issue bandwidth and register argument limit clearly limits the speedup. The best speedup is gained when the number of grouped instructions have a size and number of register arguments that are just at the limit of how much can be issued per cycle, or when they are an integer multiple of the issue limit. For example, for the setup with a bandwidth of 8 bytes and a register argument limit of 1, the best possible speedup is gained at a maximum of 2 grouped instructions, but also at a maximum of 4, 6, 8 and so on.

An interesting feature of the speedup is that the speedup gets worse after hitting the first optimum. This is due to the simulations always grouping the maximum number of single-operation instructions as far as possible. This means that consecutive single-operation instructions might not be optimally grouped. Again for example, for the bandwidth of 8 bytes, if the number of consecutive single-operation instructions is 4 and the grouping limit is 3 single-operation instructions, the first three single-operation instructions will be grouped, leaving only one single-operation instruction in the next group. This will be issued in three cycles, two for the first group and one for the second. A better way of grouping the single-operation instructions in this case would be two in the first group and two in the second group. This would be issued in a total of two cycles, one for each group.

With a wide enough issue bandwidth and high enough register argument limit, the number of issue cycles when allowing grouping can go as low as below 30 % of the cycles required without grouping. This speedup is very large but has to be put in relation to the cost of it, which the simulations have not taken into account. With a higher upper limit to the number of grouped instructions, the code size gets bigger due to increased size fields. A wider issue bandwidth can also be very costly in terms of area and power consumption. However, one can see that the speedup quickly slows down even for the highest bandwidth and
5.2 Instruction encoding for enhanced issue register issue limit. This is due to the nature of the instructions that are issued. As the number of consecutive instructions meant for a decoupled execution unit is typically quite low, a very high grouping limit only affects a few instruction sequences. Therefore, the cost of a high issue bandwidth, register argument limit and grouping limit will probably outweigh the gain, and some middle ground should probably be chosen.

5.2.2 Fetch bandwidth and instruction buffers
The simulations in the previous section assumed that the fetch bandwidth was not limiting. However, as discussed previously, a narrow fetch bandwidth might very well limit the issue speed. To show this, another set of simulations were run. Their results can be seen in figure 5.9. In these simulations, the fetch bandwidth was assumed to be 16 bytes. All other assumptions and variations stayed the same.

It is clear that the fetch bandwidth is limiting. An issue bandwidth that is greater than the fetch bandwidth provides no further speedup, meaning that the issue bandwidth should at most be the same fetch bandwidth.

However, the statement above might need to be retracted if some alterations to the system are made. If an instruction buffer is introduced, an issue bandwidth wider than the fetch bandwidth can be useful. Given a not too narrow fetch bandwidth and variable-length instructions, most instructions not intended for a decoupled unit will not need the complete fetch bandwidth, thus instead filling the buffer with the unused data. When then arriving at a block of instructions meant for a decoupled unit, the buffer can be used to issue instructions faster than what the fetch bandwidth would normally allow.

The simulations were again altered to include a buffer. Figure 5.10 shows how the issue speedup depends on the size of the buffer. To test the buffers to their full extent, the issue bandwidth and register argument limit were set sufficiently wide to never be limiting. The fetch bandwidth was still kept at 16 bytes, and was therefore the limiting factor. The buffer was assumed to be full when arriving at the instruction blocks.

The results show that a larger buffer decreases the number of issue cycles. However, they also show that the gain is not linear, as it decreases with an increasing buffer size. The reason is program dependent, and how all sizes relate to each other.

To show the impact of the program characteristics, yet another couple of simulations were run. In addition to the typical program used in the previous simulations, three new programs were examined. The first program only have blocks of instructions meant for issue to a decoupled unit which consists of 1 to 4 consecutive single-operation instructions, the second only contain blocks of 5 to 8 consecutive single-operation instructions, and the third only contains blocks of 20 to 23 consecutive single-operation instructions. The fetch bandwidth was kept at 16 bytes, and the issue bandwidth and register argument limit were still not limiting.
Figure 5.9: Comparison of the number of EU issue cycles in a typical program. The fetch bandwidth is 16 bytes per cycle. $B$ is the issue bandwidth in bytes per cycle and $p$ is the number of register arguments that can be read and issued per cycle. Registers can be issued in parallel with instruction data, and fetch bandwidth is assumed to be larger than the issue bandwidth. The average number of register arguments per instruction is assumed to be 0.5 and the average size of the instructions are assumed to be 4 bytes.
5.2 Instruction encoding for enhanced issue

Figure 5.10: Comparison of the number of EU issue cycles in a typical program for varying buffer sizes. The fetch bandwidth is 16 bytes per cycle, and the issue bandwidth and register argument limit are sufficiently high to not be limiting. Registers can be issued in parallel with instruction data. The average size of the instructions are assumed to be 4 bytes.
Figure 5.11: Difference in the number of EU issue cycles for four different programs run on a system without a buffer and on a system with a buffer of 32 bytes. The fetch bandwidth is 16 bytes per cycle, and the issue bandwidth and register argument limit are sufficiently high to not be limiting. Registers can be issued in parallel with instruction data. The average size of the instructions are assumed to be 4 bytes.

All four programs were run in a system without a buffer and then again in a system with a buffer of 32 bytes. The differences can be seen in figure 5.11, where a positive value denotes a higher number of cycles required for the system without a buffer. For the program with only 1 to 4 consecutive instructions, no speedup is gained when using a buffer. This is because all blocks is at maximum 16 bytes, meaning that the fetch bandwidth is not limiting.

The largest speedup when using a buffer is gained for the program with 5 to 8 consecutive instructions. This is because they have a size between 20 and 32 bytes, meaning that they are larger than the fetch bandwidth but can still fit in the buffer. A lower speedup is gained for the program with 20 to 23 consecutive instructions. The buffer will allow for faster issue of the first 32 bytes, but the following bytes will be limited by the fetch bandwidth, thus reducing the impact of the buffer. The typical program is somewhere in between as it consists of all type of blocks.
The simulations all show that a buffer is generally a good idea. However, the simulations have not taken into account the cost of introducing the buffer. Further, the assumption that the buffer is full when arriving at the blocks of interest may be quite unlikely, and is heavily dependent on instruction encodings and how common branch instructions are, as these might flush the buffer, since the use of a buffer can generally be seen as a prefetching scheme.
Conclusions and future work

This chapter will present some conclusions from the work done and discuss some possible future work.

6.1 Conclusions

This thesis has investigated the benefits and drawbacks of superscalar and VLIW designs for parallel decode in an embedded DSP SIMT processor with a master core and decoupled execution units.

Multiple aspects of superscalar and VLIW designs, as well as different instruction encodings for the two have been discussed. A 2-way in-order superscalar design and a 2-way VLIW design were implemented in RTL. A benchmark was used to test the performance of the two designs. To compare the costs, the two designs were also synthesized.

It was found that the VLIW design performed better than the superscalar design when running the benchmark. Several reasons for this were presented, where the fact that a better compiler was used for the VLIW code than for the superscalar code was likely to be the biggest reason. Both the superscalar design and the VLIW design performed better than an in-order scalar design, which was expected.

The VLIW design was also slightly smaller and could achieve a higher clock frequency than the superscalar design when synthesized. The area difference was however so slight that it could not be determined if it was a result of the difference in architecture, or if it could instead be a result of particularities in the synthesis tool used. The clock frequency difference was larger and was most likely due to the dependency checks present in the superscalar design, and that no timing
improvements were made to the superscalar design.

The thesis also investigated how consecutive instructions meant for a decoupled execution unit could be packaged and issued.

Several different encodings were presented and the potential benefits and drawbacks were discussed. Simulations showed that by grouping single-operation instructions meant for a decoupled execution unit, the number of cycles required for issue could be decreased considerably, given that the issue bandwidth and register argument limits allowed for it.

It was also found by simulation that the fetch bandwidth limits the issue speed. However, it was found that this could be somewhat mitigated by using a prefetch buffer, given that the issue bandwidth and register argument limit allowed for a greater speed than the fetch bandwidth would. The impact of a prefetch buffer was also found to be dependent on the nature of the program run, where all programs will have their own optimal buffer size.

### 6.2 Suggestions for future work

A number of areas have been identified which could be investigated further.

In order to better compare the superscalar and VLIW designs’ performances, a better superscalar compiler should be developed. As identified in the thesis, an in-order processor is heavily dependent on the compiler.

Instead of, or in addition to, developing a better compiler, an out-of-order superscalar design could be investigated. It would be of interest to find how costly the extra logic of an out-of-order processor would be and what performance gain one could get.

In this thesis, only 2-way processors were implemented. Designs with more ways could be implemented in order to examine the effect of the exponential complexity growth for the superscalar dependency checks, put in relation to the VLIW code size. This could show how well these designs would scale with the number of ways.

None of the encodings presented for enhanced issue to decoupled execution units were implemented. To correctly assess the efficiency of the encodings, they should be implemented and the results should be examined. It would be of interest to see what gains would be achieved in an implementation, to determine their real world application.
Bibliography


for Embedded Systems, CASES ’01, pages 168–175, New York, NY, USA, 2001. ACM.

