# Enhancing the Move framework

## Endianness port and Immediates handling

Ivo Janssen

Laboratory of Computer Engineering
Faculty of Informationtechnology and Systems
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft,
The Netherlands

May 11, 2001

Delft University of Technology
Faculty of Informationtechnology and Systems

| | | |
|---|---|---|
| Type | : | Master's Thesis |
| Number of pages | : | 104 |
| Date | : | May 11, 2001 |

| | | |
|---|---|---|
| Lab./Dept. | : | Laboratory of Computer Engineering |
| Code number | : | 1-68340-28(2001)-01 |
| Author | : | Ivo Janssen |
| Email | : | `janssen@cardit.et.tudelft.nl` |
| Title | : | **Enhancing the Move framework** |
| | | Endianness port and Immediates handling |
| Supervisor | : | Dr. H. Corporaal |
| Mentors | : | Ir. A. Cilio |
| | | Ir. H. Schot |

Vail's Second Axiom:

*The amount of work to be done increases in proportion to the amount of work already completed.*

# Enhancing the Move framework

## Endianness port and Immediates handling

# Abstract

At the laboratory of Computer Engineering of the Faculty of Informationtechnology and Systems of the Delft University of Technology, research has been done in automating the design process of application specific processors (ASPs). Within this so-called MOVE project a MOVE framework was developed which shortens the design time of ASPs. With this MOVE framework a MOVE processor can be designed. The MOVE processor architecture is a VLIW-like transport triggered architecture (TTA). The main advantages of this architecture are its flexibility and scalability.

The software framework includes a generic front-end compiler, GCC and its tools, and a back-end compiler. Research and implementation has been done on the whole framework to make the architecture both host-endianness independent and target-endianness independent. Also, work has been done on the back-end to make it possible to schedule long immediates (immediates which do not fit in the fixed-width instruction) into the VLIW-like instruction stream.

To make the framework endianness independent, the GNU front-end was altered to output either big-endian or little-endian code. The back-end, our in-house developed scheduler and simulator, was altered to compile and run correct on little-endian and big-endian hosts, and the back-end was altered to be able to read the different binaries made by the front-end and to be able to schedule and simulate the code correctly, independent of the host.

To schedule long immediates in the instruction stream, an algorithm to schedule these concurrently with the rest of the code has been constructed and data structures to hold the state of the immediates have been added to the scheduler. Where immediates used to be scheduled in dedicated immediate fields concatenated to the normal instruction word, now the immediates are scheduled in normal, otherwise unoccupied move slots. Care was taken that the routines and data structures do not interfere with already existing other algorithms in the scheduler. The algorithm increased the cycle count by several percents, but made dedicated immediate fields, that can take up 20% of the instruction word length, obsolete.

# Contents

# List of Figures

# List of Algorithms

# Part I

# Prologue

# 1 Introduction

This chapter gives a short introduction to the various topics covered in this thesis.

At the laboratory of Computer Engineering of the Department of Electrical Engineering, Delft University of Technology, research has been done in automating the design process of application specific processors (ASPs). ASPs represent a huge part of the microprocessor market, as they are used in increasingly popular embedded systems.

One of the largest part of the costs of an ASP is its design time. To shorten this design cycle, this laboratory has been developing an automated design framework based on the *Transport Triggered Architecture* paradigm. The concepts behind TTAs were developed in the same research group, and proved themselves to be especially suited for the ASP synthesis. For an in depth description of the move framework, please read chapter 2.

Several for-profit companies have been interested to take the Move framework principles and use them in their own products. One of them is NEC Computer and Communications Research Labs (CCRL) in Princeton, New Jersey, USA [1], later spun off into the independent company Eulix Networks. While developing a programmable communications processor, they needed a core that was both flexible in its interface and functionality while having a short design cycle. The MOVE framework was chosen to implement this core. The requirements as posed upon the MOVE framework by the communications processors specifications included, amongst others, a little-endian version of the MOVE core, support for long immediates, support for some special function units (SFUs) attached to the MOVE core, support for 64bit loads and stores, support for global registers, and support for an interface to the co-design simulator of the processor.

In September 1999, I was asked by my professor, dr. H. Corporaal, to join the development group in New Jersey to work primarily on two of these issues, namely the support for long im-

[1] see http://www.ccrl.nj.nec.com/

mediates and the support for endianness independence. I also did some work on other problems related to the integration of the MOVE core into the communications processor, but they will not be discussed here as they fall outside the scope of my master's thesis.

## 1.1 Endianness independence

The communications processor, as developed by Eulix Networks, deploys several on-core traffic control units as well as some busses, e.g. a standard PCI bus for external host-communications. Since these were all developed as little-endian modules, it was natural for the MOVE core to be little-endian, too. Traditionally, MOVE has been a big-endian target, developed and simulated on big-endian hosts, like HP's HPPA and Sun's Sparc architectures. Lately, development in the research laboratory has been shifted towards more common, cheaper, x86 platforms, running the Linux operating system. The x86 is a little-endian platform.

All these factors lead to the conclusion that the traditionally big-endian-host/big-endian-target architecture of MOVE needed to be extended to handle all four permutations of host and target endianness.

## 1.2 Long Immediates

Traditionally, MOVE has been primarily a research concept, with few actual realized hardware prototypes. As such, the limitation that the width of an immediate in bits needed to be shorter than the (fixed-width) instruction width of an instruction slot, could be easily overcome, since the simulator didn't need to work on the actual bits of the binary, but on a symbolic representation of the scheduled program in memory.

In the cases where a chip was actually realized, short term solutions were devised. One of them was to use a two-step stage, where an instruction containing an immediate was always followed by an instruction slot that did not contain an instruction but the value of the immediate. The program counter was incremented by two instead of one in this case.[2]. This was a good solution since the scheduling freedom was low anyway, due to the fact this particular MOVE instance had only 1 bus, but a bad one if the MOVE architecture would define multiple busses, since then it would be more advisable to schedule them into empty slots that are inherently present in VLIW scheduled instruction words.

Another solution was to add a dedicated immediate field at the end of the VLIW instruction word, that could never contain an instruction but only an immediate. Downside of this solution is that an instruction word would always contain one or more immediate fields, and that in case there was no instruction with an immediate present, bits would be wasted, which is a significant factor when it comes to low-cost embedded processors.

Above observations led to the conclusion that a new way to schedule immediates needed to be implemented. An implementation that would not waste bits but would try to schedule an immediate in unused instruction in the instruction word stream, thus minimizing the code size.

[2]This solution was implemented in the MicroMove [Jan97] processor by TNO-FEL, The Hague, Netherlands

## 1.3 Overview of the rest of the thesis

First, chapter 2 will describe the Move project, in order to have a good understanding of the principles of the Move framework.

The remaining of this thesis will further address the main two topics of this thesis.

Chapter 3 will address the ideas and difficulties behind endianness of both host-endianness dependencies and target-endianness dependencies. Chapter 4 will explain how these problems were addressed in making the Move framework both host-endianness as well as target-endianness independent.

Chapter 5 will explain the rationale behind long immediates in the Move framework. Chapter 6 will address the implementation of long immediate support in the Move framework. Chapter 7 will review these adaptations, and a quantitative and qualitative analysis will be given, as well as a comparison with a functionally similar approach to long immediates encoding and implementation in a different research group.

Finally, chapter 9 will draw conclusions on the results and will give some recommendations for future work on this subject.

# 2 The MOVE Framework

Due to the decreasing feature size of VLSI technology, the amount of hardware which can be integrated into a single chip increases. As a result, future processor chips may execute tens of operations concurrently. Many applications can profit from these huge amounts of hardware parallelism by designing an application specific instruction set processors (ASIP). Two problems emerge however: (1) the design space of ASIPs is very large; it is difficult to chose a satisfactory solution, and (2) the design complexity increases and therefore design cycle gets too long.

To alleviate these problems a design trajectory based on a *templated, transport triggered architecture* (TTA) has been developed. Using a restricted, but still very large, design space it is possible to automate the design trajectory based on a quantitative analysis of many design points. A key aspect of TTAs is the reduction of the on-chip data transport requirements; this may result in a better cost-performance ratio of the realized ASIPs. In this chapter we discuss an automated design process for ASIPs using the *MOVE framework*.

The chapter is structured as follows: Section 2.1 explains the MOVE framework, and discusses briefly how TTAs operate. Then, in section 2.2 several conclusions are drawn.

## 2.1 The MOVE framework

Designing ASICs based on templated application-specific instruction set processors (ASIPs) is an attractive solution that offers flexibility and a short design time while still retain part of the advantages of ASICs. The design process consists of finding the right architecture parameter values for the given application, such as the operation set, the amount of instruction level parallelism, and the sizes of the register files. Also, additions of special function units that can map a complex task into one single optimized instruction are possible. The quality of a solution depends on the offered performance and the implementation costs.

The synthesis framework presented in this section uses an architecture design space based on a *transport triggered architecture*, or TTA. This architecture is of the *instruction level parallel* type; it resembles the well known VLIW architectures. However, a key difference is that TTAs are programmed by specifying data transports instead of operations. This gives an finer level of control to the code generator, and allows for a more efficient use of hardware resources. Although we use a TTA template for designing ASIPs, the design space is still very large. Picking a proper solution (for a specific application) from this design space requires a quantitative analysis of many design points. This search process must largely be automated in order to reduce the design time. Therefore tools are needed, not only for making the quantitative analysis of hardware and software (generated code), but also for the automated search.

The MOVE framework consists of a set of tools for hardware and software synthesis. Within the synthesis process we use an *architecture template*, i.e. processors are built according to the pattern of a TTA. A specific TTA is defined by a set of architecture parameters, like the number and type of function units, the number of register files and registers, etc. At first sight this suggests that we restrict ourselves and therefore obtain inferior solutions. In practice however, several advantages emerge. Firstly, the template building blocks are pre-designed and can therefore be made very efficient, both in area and performance. Secondly, the architecture pipelining is worked out very carefully, alleviating many timing bottlenecks; prototype realizations learned where these bottlenecks exactly are. Finally, usage of a clearly defined design space allows the design of synthesis and evaluation tools, which not only generate a combined hardware and software solution, but also allow a quantitative analysis of the design space. Also, note that the template still covers a very large design space.

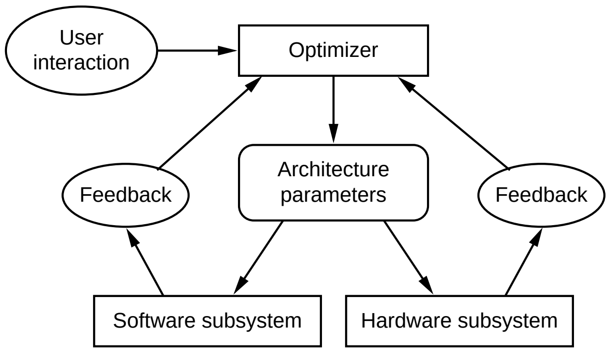


Figure 2.1: MOVE framework overview.

The synthesis of hardware and software for a given application is done using the MOVE framework; this framework produces both the layout of an ASIP and the corresponding object code to be executed on this ASIP. An overview of this framework is shown in figure 2.1. It consists of three main components:

1. **Optimizer** which is responsible for searching the design space and the interaction with the designer. It determines the configuration (i.e., the set of architecture parameters) to be evaluated next.

2. **Hardware subsystem** generating processor layout, and giving information on timing,

area, and power consumption.

3. **Software subsystem** generating instruction level parallel code, and giving statistical information on usage of hardware resources.

These components are detailed in following subsections. Before, we briefly describe how TTAs operate.

### 2.1.1 Transport triggered architectures

TTAs can be compared to VLIW architectures; their instructions are horizontally encoded; i.e. each instruction has a number of fields. Whereas fields for VLIWs specify RISC like operations, for TTAs they specify the required data transports. These transports may trigger operations as side effect. Programming transports adds an extra level of control to the code generator, and enables new optimizations; in particular, it allows us to get rid of many superfluous data transports to and from the register files and to reduce the on-chip connectivity[HC94].

A compiler views a TTA as a collection of function units (FUs), register files (RFs), *move buses*, and *sockets*; see figure 2.2. FUs perform operations, RFs provide temporary fast accessible storage, the network of move buses performs data transports between the FUs and RFs, and sockets interface FUs and RFs to move buses. Normally, each socket is connected to a different FU input/output or RF port.

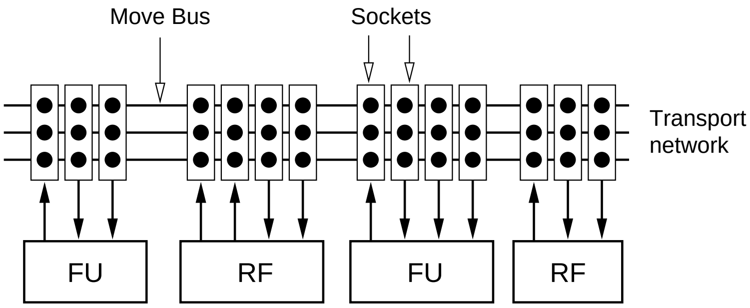


Figure 2.2: General structure of a TTA.

To illustrate TTA programming, consider the following three operations of an operation triggered machine, or OTA:

```
add  r1, r2, r3   /* r1 = r2 + r3 */
sub  r4, r2, r6   /* r4 = r2 - r6 */
st   r4, r1       /* store r4 at address r1 */
```

These operations can be translated into the following two TTA instructions:

```
r2->add_o,   r3->add_t,    r2->sub_o; r6->sub_t;
add_r->st_t, sub_r->st_o;
```

In the first instruction the four operands of the add and subtract operations are moved from the RF(s) to the FU inputs of the FUs that perform the two operations. In the second instruction the results of the add and subtract operations are moved from the FUs that performed them to the FU that performs the store operation. From this small example we already observe a few advantages of TTAs. The results of the add and subtract operations are not written back to the RF and the operands of the store operation are not read from the RF. The former saves RF write accesses and data transports, the latter saves RF read accesses. Since TTAs do not couple move buses and RF ports directly to FUs, as is the case for many VLIW and super-scalar architectures, the freed resources can be used for other operations. This makes that TTAs have a better hardware utilization, which implies less hardware for the same performance or more performance with the same hardware [HC94].

The interconnection network may be fully connected, as shown in figure 2.2, in which case every socket is connected to all move buses, or partially connected. A fully connected interconnection network simplifies the code generation task, but it likely results in a high bus load on the move buses which affects the achievable cycle time. Therefore, in practice the interconnection network will be partially connected and the compiler is responsible to use the available connections as well as possible.

Besides executing operations on data, TTAs need to provide immediate operands, conditional execution, and control flow changing operations. Details on these issues can be found in [Cor95b, CM91].

### 2.1.2 Optimizer

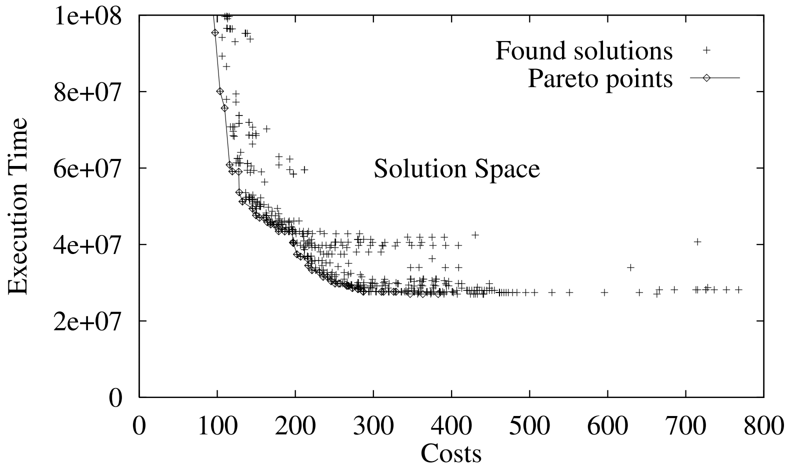


Figure 2.3: Possible solutions and Pareto points.

Two main design evaluation criteria are cost and performance, where performance is defined as the inverse of execution time. Costs may include the amount of chip area, number of pins, power dissipation, and code size[1]. Execution time is dependent on the number of executed operations, latencies, cache misses, and the clock cycle time. The *solution space* is given by all possible design points in the 2-dimensional cost-performance space. Figure 2.3 shows many generated solutions for a test application (described in [CH96]). As shown, the solution space is

[1]Currently are included area and pins only.

bounded by a curve connecting so called *Pareto points*.

The optimizer finds its way through this search space by iteratively trying different architecture solutions, and letting the software and the hardware subsystems produce relevant information about these solutions, like cycle time, costs and number of cycles needed to run the application. Based on this information a next design point is chosen by updating the parameters. The initial architecture parameter values can be chosen freely by the user. He can also specify an evaluation function (e.g. minimize the product of costs and execution time), and the stop criteria.

### 2.1.3 Hardware subsystem

The hardware subsystem of the MOVE framework is responsible for the realization of an application specific TTA in silicon. It accepts architecture parameter values, technology information and a cell library as input, and produces a VLSI layout (e.g. in CIF format) of the generated processor as output. Figure 2.4 shows its organization.

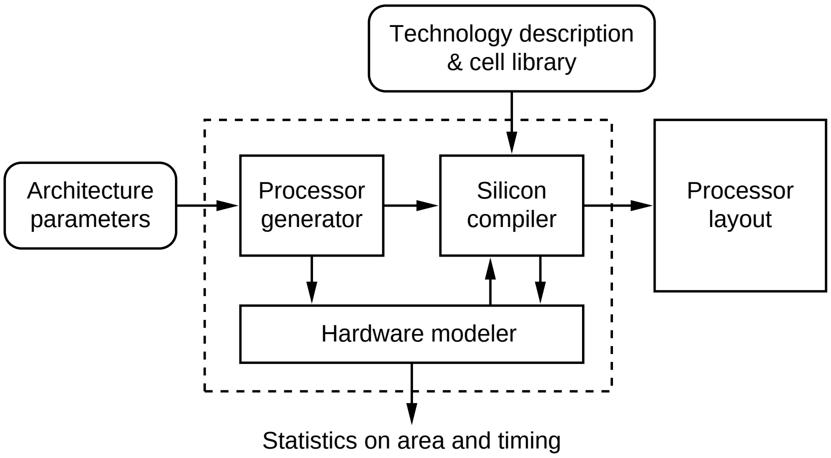


Figure 2.4: The hardware subsystem.

The design space explorer makes use of a *hardware model* to estimate the cost of design points. The costs of FUs are based on a 32-bit data path width[2], and relative to an integer FU. The minimum clock cycle time for a TTA realization is largely determined by the time needed to perform (and control) data transport.

### 2.1.4 Software subsystem

The software subsystem is detailed in figure 2.5. It provides the user with three main tools to develop code for TTAs. These tools are:

1. A compiler (referred to also as MOVE front-end) to translate HLL (high level language) code to sequential move code

[2]Although the hardware subsystem can generate processors for any data width, the software subsystem currently requires 32 bit integers.

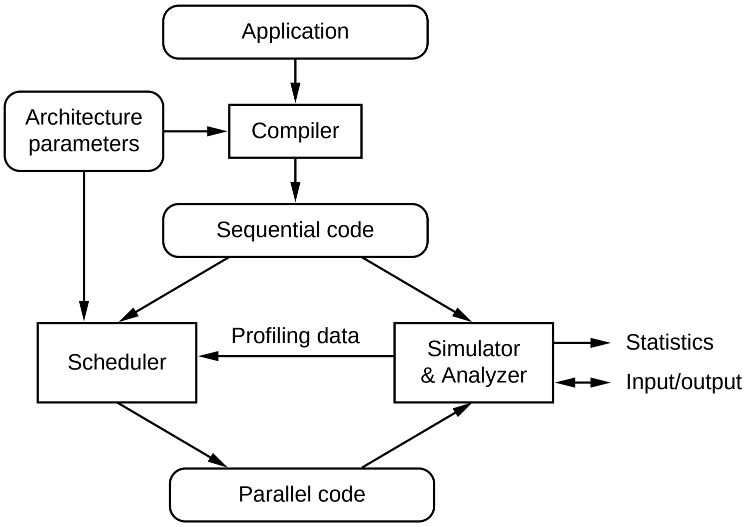


Figure 2.5: The software subsystem.

2. A scheduler (or MOVE back-end) to schedule the sequential code and produce parallel code for a target TTA
3. A simulator and analyzer to verify and evaluate both the sequential and parallel code.

The software subsystem accepts any application coded in C or C++ and translates it into text representation of MOVE parallel code for a specific TTA. The components of the software subsystem are described in detail below.

### Compiler

The MOVE front-end is a combination of three tools: proper compiler, assembler and linker. Their relationship is depicted in figure 2.6. In order to be assured of good code quality, good HLL compatibility, support for new HLLs and an extensively debugged compiler, a port of GNU C compiler (*gcc*), assembler, and linker was made. These software packages are ported to produce binary sequential MOVE code for a *MOVE generic machine*. This code is sequentially ordered by instructions; each move referring to the same operation is grouped in a single instruction, resembling OTA instructions. Sequential code is used as intermediate representation of the program and is read by the scheduler.

### Scheduler

The scheduler is the most important part of the software subsystem. Its main function is to schedule moves of sequential code, i.e. to assign FUs to operations and to assign cycles, sockets and buses to moves. The scheduler has to generate instruction level parallel code, while exploiting all the available hardware resources. To this purpose, the scheduler uses profiling data (like execution frequencies) from the simulator. Several preliminary optimizations on sequential code are also applied. The scheduler uses advanced techniques like extended basic block scheduling, software pipelining, and speculative execution, in order to enhance code motions and consequently

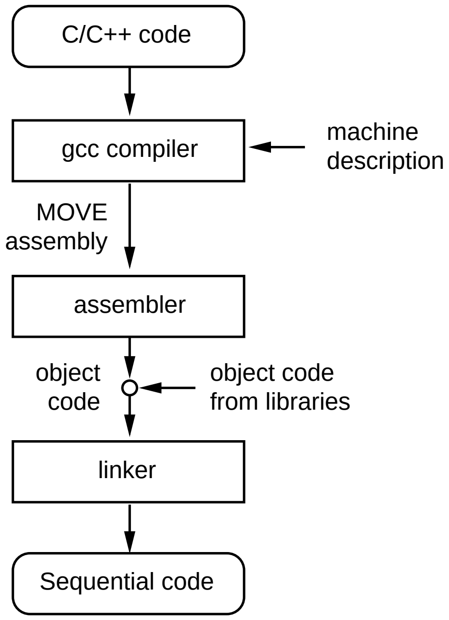


Figure 2.6: Relations between the ported GNU compiler, the assembler and the linker.

inter basic block parallelism [Hoo96]. All specific optimizations of TTAs (result bypassing, dead moves elimination, operand sharing) are performed during the scheduling process. The parallel code is fully parameterized on the template configuration, which is specified in a machine description file. In this file processor resources, like supported FUs, amount of registers and interconnection network are described. Profiling information is not strictly necessary, but helps the scheduler to work more efficiently.

### Simulator

The simulator accepts either sequential MOVE code or parallel MOVE code. Its output consists of profiling information, application output and execution statistics. The simulator has three purposes in the MOVE framework:

1. To verify the compiler and the scheduler. It is virtually impossible to port a compiler and write a scheduler without simulating the produced code.

2. To evaluate architecture parameters. The results of the evaluation are cycle counts and various statistics about resource utilization and compilation events (e.g. the number of operand swaps and the number of loop scheduled using software pipelining).

3. To provide profiling data to the scheduler. Profiling data consist of execution counts for each basic block and each control flow edge between basic blocks in the program. With this information the scheduler can decide which code motion between basic blocks is most profitable.

## 2.2 Conclusions

In this chapter we showed an automated design trajectory for ASIPs based on transport triggered architectures. This trajectory has two fundamental capabilities:

1. It maps arbitrary applications, written in C/C++, into a combination of hardware and software.

2. It offers the possibility to do a quantitative analysis of large parts of the design space.

The search process to find a proper solution consists of resource and connectivity optimization. Resource optimization attempts to find the cost effective set of resources. Connectivity optimization reduces the connectivity in order to reduce bus load and cycle time. As side effect it has been demonstrated that the synthesized TTAs require far less connectivity and fewer register ports than more traditional instruction level parallel architectures.

# Part II

# Endianness

# 3 Overview on endianness

The MOVE framework historically ran on big-endian platforms like HPPA, Sparc and MIPS. With the increased popularity of Linux running on relatively cheap little-endian platforms, like the x86 platform, the need for a port of the MOVE framework to a little-endian host platform arose. At the same time, the "PcomP" implementation of the MOVE architecture was decided to be little-endian. This and the next chapter will deal with changing the MOVE framework to be running independent of the host platform's endianness, as well as changing the MOVE framework to be able to generate and simulate code for both little and big-endian targets.

This chapter will give an overview on endianness dependence itself, and how that affects implementation of tools, with the emphasis on emulation tools. Chapter 4 will discuss how the principles of this chapter are used in the port of the MOVE framework.

## 3.1 Endianness in general

Endianness, deals with the ordering of fields within an item. Usually it means byte ordering within a halfword, word and double word. However it can also mean bit ordering within bytes. Byte ordering is the most visual, since most memory systems are byte addressable, IO works on bytes, strings are packed as bytes within words, etc. Bit ordering comes into play when structures are accessed with bitfields in a byte. The latter becomes architecturally visible when a program accesses bitfields within a byte.

The historical name "endianness" refers to the book "Gulliver's travel" by Jonathan Swift. The Lilliputians liked to break their eggs on the small end and the Blefuscudians on the big end. According to the book, *...It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large*

*Volumes have been published upon this Controversy*. The analogy is taken very well, since there is no real "correct" implementation. Both have their advantages and disadvantages.

The official definition of big-endian and little-endian is the following:

**big endian** ordering means storing the least significant byte at the most significant address.

**little endian** ordering means storing the least significant byte at the least significant address.

Consider the following 16 bytes of data in table 3.1

| **address** | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **contents** | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |

Table 3.1: Memory as an array of bytes

If instead of considering the memory as an array of bytes, we consider the same memory contents as four 32-bit words. Then it will be shown that a little or a big endian machine have a different view on the memory. Table 3.2 will show a little-endian memory and table 3.3 will show a big-endian memory.

| **contents** | | | | **word addr** |
|---|---|---|---|---|
| 03 | 02 | 01 | 00 | **00** |
| 07 | 06 | 05 | 04 | **04** |
| 11 | 10 | 09 | 08 | **08** |
| 15 | 14 | 13 | 12 | **12** |

Table 3.2: Little endian memory as an array of words

| **word addr** | **contents** | | | |
|---|---|---|---|---|
| **00** | 00 | 01 | 02 | 03 |
| **04** | 04 | 05 | 06 | 07 |
| **08** | 08 | 09 | 10 | 11 |
| **12** | 12 | 13 | 14 | 15 |

Table 3.3: Big endian memory as an array of words

In the two tables, the bytes are grouped into four byte words, which are shown in the normal Arabic form, with the most significant byte on the left. In table 3.2 the word address column was put on the right ("little end") because the computer uses the address of the least significant byte, the byte on the right, to address the word. In table 3.3, the address column is on the left ("big-end"), showing that the computer addresses the most significant byte in words operations. As a result, a little endian processor loading the 32-bit work at word address `0x00` would obtain the value `0x03020100`, while a big endian processor would obtain the value `0x00010203`.

An important observation that has to be made is that if a certain machine is unable to access bits in a byte, the endianness with respect to bit-ordering means nothing. This is because the program has no way to access data smaller than a byte anyway, and as long as the machine stores and retrieves the data in a consistent way, it doesn't matter how exactly this is done. The same goes for a machine which is word-addressable, because words will always be fetched as a whole from memory, and it doesn't matter how those words are stored in memory.

Examples of little-endian machines are the Intel x86 family and various architectures from DEC, like the VAX, the PDP-11 and the Alpha. Examples of big-endian machines are the Sun Sparc, HPPA and the m68k architecture. Still other architectures, like the PowerPC, the MIPS and the Intel IA-64 architecture, are capable of operating in either big or little endian mode. Usually the operating system dictates the endianness that the processor is going to use during that boot.

Regarding endianness, we can divide the problem in two parts. First is the most known one, the so-called "host endianness". This kind of endianness concerns the problem of begin able to use data between platforms of different endianness. What one wants to do is, for instance, write a binary file on a big-endian platform, and read it on a little-endian platform. This is what section 3.3 discusses. Section 3.4 discusses another problem regarding endianness, the so-called target-endianness. This kind of endianness concerns the fact that a certain host should be able to process data in a certain endianness, without having that data being related to the host itself. This is a common scenario when dealing with foreign binaries. MOVE specific, this means that a certain host has to be able to process binaries, and simulate binaries, from any endianness. First we will present some solutions to handle endianness, both on the software level as on the hardware level.

## 3.2 Solutions

### 3.2.1 Software solutions

There are different ways to agree on an interface:

1. **Form an endianness-independent transport layer.** Certain graphics applications, like Framemaker, have an option to write the data to an endianness independent file. With Framemaker, this is the `.mif`-format, or Maker Independent Format. This file can be read on any platform, but is two to three times larger than the normal `.doc` format that Framemaker uses. Sun has its XDR (eXternal Data Representation) format [Zuk98], which, apart from endianness independence, also claims independence from various floating point implementations.

2. **Let the data be stored/transmitted in the native endianness of the sender.** This is done along with tag or header that indicates endianness. This method is deployed by the TIFF graphics format, which can either be "IBM ordered" or "Macintosh ordered". A header specifies whether the data is big ("Macintosh ordered") or little ("IBM ordered) endian. Another scenario where this principle is applicable is when dealing with binaries from a different architecture. If a binary is in a certain endianness, and it is emulated or simulated on a host machine, this host machine has to check for endianness in that 'target' architecture and handle accordingly.

3. **Let the data always be a certain kind of endianness.** This is the most common way to handle endianness, since it is unambiguous what the data's endianness is. This way a program only has to take into account its own endianness. This is also the way the Internet works. Data transmitted over the internet is always big-endian, and Unix systems provide the system calls `ntohl(3)` and `htonl(3)`, for respectively "network-to-host" swapping and "host-to-network" swapping.

### 3.2.2 Changing and detecting endianness

To "encode" an endianness independent layer, like in option 1, every designer is free to choose his own implementation. Option 2 and 3 only need a so-called "byte-swap" or "byte-reordering". For this "byte-swap", a very simple piece of code can be used:

```
#define BSWAP32(x) \
  x = ((((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >>  8) | \
      (((x) & 0x0000ff00) <<  8) | (((x) & 0x000000ff) << 24))
```

What we do here is just swapping bytes 1 and 4, and bytes 2 and 3 in a word. This will indeed swap the data, and will convert tables 3.2 and 3.3 into each other. This macro is exactly the code used in the `ntohl(3)` and `htonl(3)` calls on little-endian platforms. On big-endian platforms, where there is no swap needed, since host ordering and network ordering are the same, these two calls are null-macros, a very efficient way to implement a no-operation in C or C++.

A good way to detect the endianness of a platform is the following standard piece of code:

```
long i = 0x44332211;
unsigned char* a = (unsigned char*) &i;
end = (*a != 0x11);
printf("The endianess is %s!\n", ((end==1)?"big":"little"));
```

This piece of code will fetch one byte out of the word `i`, by getting the byte addressed by the pointer to `i`. If the platform's endianness is little, the byte `0x11` is fetched, since the *least* significant byte is stored at the most significant address, which is the pointer's address. If the platform's endianness is big, the byte `0x44` is fetched, since the *most* significant byte is stored at the most significant address. And the most significant address is the address to which the pointers `i` and `a` point.

### 3.2.3 Hardware solutions

Although not in the scope of this thesis, a brief overview on hardware solutions to the endianness problem are discussed. If there is no way to have software do byte-swapping, the hardware has to do this. A common principle to do this is to give the system multiple views on the address space where the endianness-dependent devices, like memory or graphics processor, reside. These various views are called *apertures*. For example, a PCI based graphics adapter that internally uses a little endian processor can provide two apertures (ranges of addresses) for its frame buffer. Accesses to the little endian aperture store the data as presented on the bus directly into the frame buffer; accesses to the big endian aperture swaps the data bytes before storing them. Thus, an application running on a big endian processor can simply access the big endian aperture and store its big endian data just as if it were running on a little endian processor. The device takes care of swapping the data in hardware as necessary.

## 3.3 Host endianness

Host endianness concerns the way data can be kept portable across platforms. This section will concentrate mainly on files, although its principles can easily be extended to other forms of interoperatibility, like shared memory between a graphics processor and a CPU, or inter-architecture buses (PCI, SCSI). Regarding Move, this means that solution 3 from section 3.2.1 is used: Let the data on disk always have the same endianness. This way the only check a program has to do is its own endianness.

### 3.3.1 Example

As an example of host-endianness, we will explain the profiling code in the scheduler. The profiles, including frequency counts and memory dependencies, are target-independent data. Therefore, they are stored on disk in a pre-defined endianness. In this case this is big-endian.

The functions `Prog::SaveProfile` and `Prog::LoadProfile` are responsible for writing and reading the profiles. Therefore the following code can be found:

```
for(p = proc; p; p++)
{
  for(b = p->blck; b; b++)
  {
#if HOST_LITTLE_ENDIAN
    f = SwapEndianess(b->freq);
#else
    f = b->freq;
#endif
    file.write((char *) &f, sizeof(double));
  }
}
```

For each procedure, and each block in that procedure, an integer `b->freq` exists, indicating the frequency of that basic block during execution. Because agreed was that all data on disk was big endian, a byte swap is performed, governed by the predefine `HOST_LITTLE_ENDIAN`. The function `SwapEndianess` performs the actual byte swap in the word `b->freq`.

## 3.4 Target endianness

Target endianness concerns the portability of foreign code on a platform of random endianness. This is especially relevant here, since any platform, either big or little, should be able to omit, schedule and simulate binaries from any MOVE architecture. Now it is, contrary to host-endianness, important to keep the endianness of a file intact while reading and writing. This means we take on option 2 as presented in the list of subsection 3.2.1. This option says that the endianness of a certain file can depend. This means that the program has to know up front what kind of file it is dealing with. Regarding Move, this means that the tools have to know what kind of endianness the target-binary has. Since the host-endianness still has to be taken into account, the rules for swapping get more complicated. Table 3.4 shows all cross-combinations possible. For swapping, the macro as presented in subsection 3.2.2 can be used.

| | **target big** | **target little** |
|---|---|---|
| **host big** | no swap | swap |
| **host little** | swap | no swap |

Table 3.4: Endianness swap depending on host and target

### 3.4.1 Example

As an example of target endianness, we take the reading of the text segment of the serial binary. The text segment is "target-dependent", that means that the serial binary can contain either Move code for a little-endian Move architecture or a big-endian Move architecture. The scheduler is compiled with an option that indicates what kind of target-endianness is should be capable of handling. This is defined in the `config.h` file of the scheduler:

```
#define __pcomp__

#ifdef __pcomp__
  #define TARGET_LITTLE_ENDIAN              1
  #define TARGET_BIT_FIELDS_LEFT_TO_RIGHT   0
#else
  #define TARGET_LITTLE_ENDIAN              0
  #define TARGET_BIT_FIELDS_LEFT_TO_RIGHT   1
#endif
```

By setting the `__pcomp__` define, the target-endianness is known throughout the whole scheduler. Then we add some extra defines that indicate whether the target-endianness is different from the host-endianness (remember that the host-endianness is already stated in the `HOST_LITTLE_ENDIAN` defines, derived from the presence of the compiler-required defines `__386` and `__sun`.

```
#if    HOST_LITTLE_ENDIAN && TARGET_LITTLE_ENDIAN
  #define SWAP_ENDIANESS 0
#elif  HOST_LITTLE_ENDIAN && !TARGET_LITTLE_ENDIAN
  #define SWAP_ENDIANESS 1
#elif !HOST_LITTLE_ENDIAN && TARGET_LITTLE_ENDIAN
  #define SWAP_ENDIANESS 1
#elif !HOST_LITTLE_ENDIAN && !TARGET_LITTLE_ENDIAN
  #define SWAP_ENDIANESS 0
#endif
```

Now we can proceed to the file `exec.h`, which implements the `AOut` class that reads in the serial binary. Now we can read in the whole text segment. class `AoutMove` represents a move instruction in the binary.

```
AoutMove* text = (AoutMove *) &image[N_TXTOFF(*exec)];
#if SWAP_ENDIANESS
  SwapEndianess(text, exec->a_text);
#endif
```

We simply map the whole binary (represented by `image`) to the `text` array. Now the only thing left to change is the ordering of the bytes in a word. Since this ordering is dependent on both the host ordering as the target ordering, the `SWAP_ENDIANESS` define governs whether a swap is needed, pursuant to table 3.4.

# 4 Endianness implementation

This chapter will explain how the MOVE framework was made endianness-independent. The previous chapter explained the issues concerning endianness, and this chapter will deploy the guidelines from that chapter on the MOVE framework, to make it host-platform independent as well as move-target independent.

## 4.1 The MOVE framework

The MOVE framework consists of roughly two parts: the front-end and the back-end. This can be visualized in figure 2.5. The front-end consists of a standard freely available compiler, the GNU C Compiler, including its tools like assembler, linker and auxiliary tools. This compiler can be easily ported to other architectures by means of writing a new architecture plug-in. The tools related to the compiler, the so-called binary tools, include an assembler, linker and various other tools that operate on the binary. These tools were ported to the MOVE architecture by rewriting the "m68k" assembly format to fit the MOVE specification. Lastly, a system library, also called the C-library, is needed. This library does not need any MOVE specific changes, but it does have some endianness dependencies that need to be resolved. The adaptations needed to make the front-end endianness independent can be found in section 4.2

The back-end consists of the scheduler tools, including a simulator, scheduler and various auxiliary tools that help the simulator and the scheduler, like a design-space explorer, a call-graph visualizer and a tool to view the assembly code in a human readable form. The adaptation needed to make the back-end endianness independent can be found in section 4.3

The approach taken was to make the whole framework first host-endianness independent. This alone would allow us to deploy the MOVE framework on Linux platforms instead of Sparc platforms, even if there was no problem of target endianness (i.e. the Move architecture has only one endianness, e.g. big, and only the endianness of the host has to be taken into account). This port was done in such a way, that further work on making the framework target-endianness independent, would be straightforward and trivial. Another requirement was that the host-endianness dependencies would be compile-time invisible, so that there would be no need to set compile-time switches to help the framework determine what kind of host platform is was compiled on. This was achieved through the use of the predefines that the compiler sets: e.g. `__i386` is set during building and installation on an Intel platform.

Target endianness was implemented in such a way that a single switch in the two Makefiles is sufficient, one for the front-end and one for the back-end. This would require a recompile for each target. This was considered not a problem, since the MOVE framework's directory layout also included a directory for pre-compiled libraries, which would be target-endianness dependent anyway. A new tree for either little or big-endian targets, including compiler-binaries and libraries for each target, is then created.

First, some defines are set to indicate the host and target endianness:

- If the host platform is big-endian (check `__sun` flag, a preprocessor predefine that is always set on Sparc machines), then the define `HOST_BIG_ENDIAN` is set to `1` and the define `HOST_LITTLE_ENDIAN` is set to `0`. Else, the host platform is little endian, and both defines get their inverse value.

- If the target architecture is little-endian (check setting in `config.h` for the back-end, a Makefile directive in the front-end) then the define `TARGET_LITTLE_ENDIAN` is set to `1` and the define `TARGET_BIG_ENDIAN` is set to `0`. Else, the target architecture is big endian, and both defines get their inverse value.

Note that we explicitly set all preprocessor defines to a value, instead of just defining or undefining them. Now that these 4 defines are set throughout the whole framework, in a uniform way, we can easily check for these values whenever we encounter a dependency.

A couple of extra defines are derived from the four mentioned above. In many cases, e.g. cases were we have both target- and host-endianness dependencies, we want to have a way to check whether the two endiannesses are the same or different. Therefore we add one more preprocessor define, `SWAP_ENDIANESS`. This define is set to `1` when the endiannesses are different, and set to `0` when they are the same. Since converting from little to big endianness is the same as converting from big endianness to little, this define comes in handy whenever we have encounter a structure that is both dependent on the host and the target platform's endianness.

Endianness in files on disk can be divided in two variants. One variant is a file that contains target-dependent information. This includes the serial binary and the parallel binary. The other variant is a file that does not contain target-dependent information, like the profiling files of the back-end. These files are always stored in a big-endian way. To process structures in files that have a target-endianness dependency, and of course implicitly also a host-endianness dependency, we now can use the `SWAP_ENDIANESS` define to read and write these files correctly. To process files that only have a host-endianness dependency, we can use the `HOST_BIG_ENDIAN` or its inverse, the `HOST_LITTLE_ENDIAN` define.

More problematic are files that contain both target-dependent data (apply target-endianness) and target-independent (apply host-endianness) data. The serial binary, for instance, also contains various structures that have nothing to do with the program, but are necessary for a correct binary. Care must be taken that the routines that operate on these files can recognize the various structures and process them correctly. The approach taken is to have all host-endianness dependencies written out in big-endian on disk. This corresponds with the format on disk that the legacy tools used. The target-endianness is stored either little or big endian, and a tag (in this case the binary-header) is used to differentiate between the two forms of target-endianness.

To visualize the cross platform requirements on the front-end, please look at figure 4.1. This will show that every step in the process should be able to read either output from a platform with the same endianness, as well as output from a platform with a different endianness. This figure shows the trajectory for one target-endianness. To list all possible cross-relationships, one should duplicate this figure for another target-endianness. These two figures, each representing a target-endianness, then would be completely unrelated. For example, a linker compiled to link big-endian move code cannot read little-endian move code.

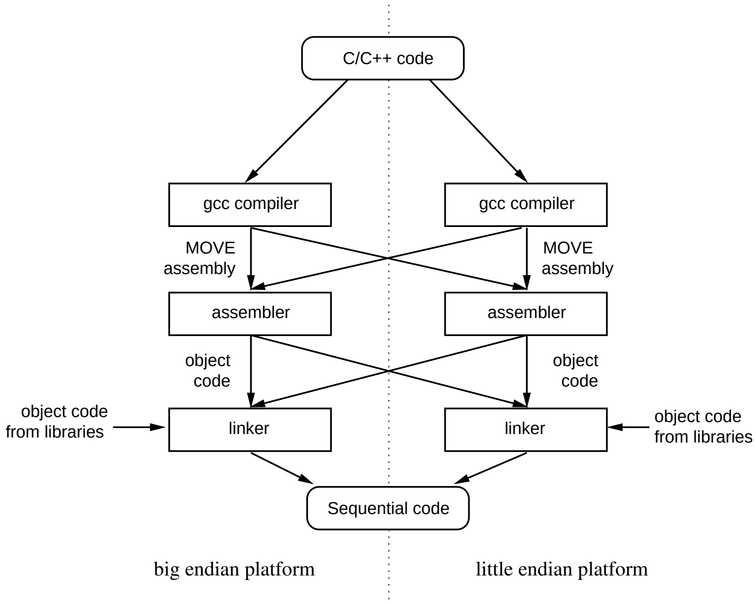


Figure 4.1: The front-end on both endianness platforms

## 4.2 The MOVE front-end

The front-end consists of the GCC compiler, version 2.7.0, and the `bintools` distribution, version 1.38, together with a standard BSD C-library. All three components can be made endianness independent on their own, as long as the interface format is clearly defined. We have three interface formats that we have to deal with:

1. plain text assembly code (`.s` files)
2. non-linked object files (`.o` files)
3. linked binaries (no suffix)

Regarding host-endianness dependencies, we impose the requirement on these two latter formats that they are independent of the host-endianness. This results in the property that a file can be created on one platform, and read back by another platform, without any problems.

The first format, the plain text assembly, has no host-endianness dependencies, since it is plain text. Plain text is stored sequentially on disk, and has no endianness at all. The other two do have dependencies, because the object format deploys both bit-fields as well as byte-ordering within a word.

Regarding target-endianness dependencies, we impose the requirement on these three formats that the binary formats all use the same encodings, so that there is no format difference between the data in an object and the data in a binary. This also makes sure that object files archived in a library can be linked against other object files without worrying whether a certain routine came from a stand-alone object file or a library.

Target-endianness can be divided up in two parts: The actual instructions and the helper information, such as the symbol table and the relocation table. The choices made for the various types of endianness for these various kinds of data inside an object file, will be further explained in subsection 4.2.2, which deals with the port of the assembler, linker and binary tools.

### 4.2.1 GCC

Since GCC version 2.7.0 is already host-endianness independent, no code changes were needed. Target-endianness is controlled by some directives in the so called "target-description macros", as defined the the `machine.h` file of GCC. The relevant macros are `BITS_BIG_ENDIAN`, `BYTES_BIG_ENDIAN` and `WORDS_BIG_ENDIAN` [Sta94]. For little-endian targets, all three defines are set to zero, and for big-endian targets all three defines are set to one.

Since GCC outputs plain text `.s`-files, the assembly doesn't need to care about most endianness problems. A word will be represented as its decimal equivalent in ASCII, and it's the assembler's job to encode this in 4 bytes. What does matter is bitfields within a byte, which will be implemented by the compiler by shifting the byte several bits, as well as accessing bytes within a word explicitly in C. This is also handled by the compiler by outputting code that shifts a word several bytes in order to be able to access the right byte in a word. All the compiler needs is the three above-mentioned switches to take care of these cases. This concludes the port of both the target-endianness as well as the host-endianness dependencies of the GCC compiler.

### 4.2.2 Assembler, linker and auxiliary binary tools

The version of the `bintools` [1] used in the MOVE framework is version 1.38. This is unfortunately a very old version, and completely obsolete. There are two major drawbacks with this version: The architecture format is not easily changed, instead, for each port, a complete implementation of the tools exist, instead of a configurable plugin, like GCC-2.7.0 or the newer versions of `bintools`. The second drawback is that host-endianness independence is not implied, like it is in GCC-2.7.0 or the newer versions of `bintools`.

These considerations led to a feasibility study of deploying a newer version of the tools. A new version of the `bintools` distribution was inspected, but porting to the existing MOVE binary format, while preserving all intermediate formats and the actual serial binary that the

[1] the common name for the group of assembler, linker and other tools

back-end can read, would be a lot of work. On the other hand, some work on making these tools host-endianness aware was already in progress. Conclusion was that a complete new port to the MOVE binary format would require more work than changing the current code base. Therefore it was decided the current version 1.38 was to be changed.

The binary tools consist of the following programs:

- `as`, the assembler
- `ld`, the linker/loader
- `ar`, the library archiver
- `size`, a utility to print segment sizes
- `nm`, a utility to print out symbols
- `objdump`, a utility to dump various segments
- `ranlib`, a utility to index a library archive
- `dem`, a utility to demangle C++ symbols
- `c++-filt`, another utility to demangle C++ symbols
- `strip`, a utility to strip a binary from its symbols

These tools can be divided in three parts, the assembler, the linker and the rest of the tools.

#### 4.2.2.1 The assembler

The assembler was very straightforward to port. The function `md_number_to_chars` is responsible for placing a value representing a byte, halfword or word in a file stream. It does this by putting each byte in the correct position in the 4-byte word. For this it uses the C++ >> operator, which is already host-endianness safe (the compiler will recognize that operator and rearrange bytes and words in order to let the >> operator be endianness-independent). So the only consideration is the target endianness. By adding a check for `TARGET_LITTLE_ENDIAN` it was trivial to put the bytes on the right spot in the word, according to the rules of endianness.

The function `md_number_to_chars` is, amongst others, called from the routine that emits the text segment. Since the move binary instruction field encoded in the binary file contains bitfields, the calls to `md_number_to_chars` are re-ordered in case of a little endian target.

#### 4.2.2.2 The linker

The linker needs to actually read all various segments of an object, in order to be able to read the symbol and relocation tables, alter them and write them back into the final binary. Since this old version of `ld` processes all sections with direct calls to the Unix system call to `read(2)` and `write(2)`, it was not directly possible to swap various bitfields and bytes. These functions read a whole block at once into a buffer, without the option to swap while reading. Therefore for every kind of segment, new routines called read_<segment> and write_<segment>

are created to be put in place of the actual `read(2)` and `write(2)` calls. These hooks then read the corresponding segment in a buffer, process that buffer for endianness according to the `HOST_LITTLE_ENDIAN` and `TARGET_LITTLE_ENDIAN` predefines, as explained in chapter 3.1, and return the buffer to the calling routine.

In this fashion, the following routines are defined to serve as hook for the real `read(2)` and `write(2)` calls:

- `read_header`, for the binary header
- `read_integer`, for a simple number
- `read_symroot`, for symbol table indexes
- `read_symbols`, for symbols
- `read_symdef`, for symbol definitions
- `read_reloc`, for relocation information
- `read_arhdr`, for the archive header
- `read_text`, for the text segment
- `read_strings`, for simple strings of text

Also, their `write_` counterparts are defined.

These functions read or write the corresponding segment, with knowledge on where bitfields and other boundaries (such as 16 bit data that only needs to be swapped on half-word boundaries) on segments occur, so they can apply the endianess switches correctly.

#### 4.2.2.3 Bintools

The bintools, coming from the same distribution as the linker in the previous section, suffer from the same drawback that they deploy direct calls to `read(2)` and `write(2)`. In addition to this, they also use direct calls to `fread(3)` and `fwrite(3)`. This means that the list presented in the previous section needs to be duplicated to also be able to hook all calls to `fread(3)` and `fwrite(3)`. For the rest, the port of these bintools is straightforward and implies, just as with the linker, replacing all occurrences to `read(2)`, `write(2)`, `fread(3)` and `fwrite(3)` with calls to their corresponding hooks.

### 4.2.3 System libraries

The GNU C Library is endianness independent, except at one point. There are different ways of encoding floating point numbers. They are:

1. **vax** for the VAX D_floating format
2. **tahoe** for the TAHOE double format
3. **national** for IEEE machines whose floating point implementation has similar byte ordering as the NATIONAL 32016 with 32081

4. **ieee** for other IEEE machines

A comparison with floating point implementations on other architectures learned that big-endian IEEE machines use the "ieee" format, while little-endian IEEE machines use the "national" format. The `makelibs` script that generates the `libm` library, the math part of the C-library, was changed so that when the libraries were compiled for a little-endian MOVE target, the "national" encoding would be used.

## 4.3 The MOVE back-end

The back-end communicates with the front-end through only one thing, namely the binary. As explained in the previous section, the host-endianness of the platform where the framework runs on does not matter, only the target-endianness of the MOVE architecture will influence the contents of the binary.

Also, it was noted that tools could be run from any host platform. That means files written to disk between runs of various parts of the back-end should be host-endianness independent, too. The files we are dealing here with are:

1. *the serial binary* that is produced by the front-end
2. *profiling* data that is written to disk
3. *the parallel binary* that is produced by the scheduler
4. *the parallel assembly* that is produced by the scheduler

Subsection 4.3.1 will deal with the reading of the serial binary, subsection 4.3.2 with the scheduling of the MOVE code, subsection 4.3.3 will deal with the profiling files and subsection 4.3.4 will deal with the write of the parallel binary. The parallel assembly file is just plain text and has no endianness dependencies.

### 4.3.1 Binary reader

The binary reader is, endianness-wise, the trickiest part of the back-end. It has the task of reading the binary generated by the GNU front-end and convert it into internal data structures. With respect to endianness, this means two things:

1. The binary must be read independent from the **host endianness**. That means that in parts where there is no target-endianness dependency, the define `HOST_LITTLE_ENDIAN` must be checked, and a byte swap must occur if the host endianness is different from the endianness of the structure on disk (which is always big endian, like it was in the legacy framework).
2. The binary must be read independent from the **target endianness**. That means that in parts where there is a target-endianness dependency, the define `SWAP_ENDIANNESS` must be checked, and a byte swap must occur if the host endianness is different from the endianness of the structure on disk (which can be either way, depending on the target architecture).

As mentioned in subsection 4.2.2, the serial binary contains various different sections. The class AOut takes care of reading in the binary, section by section. The following structures needed adaptation on endianness

- the header, which contains bitfields. The header does not contain any target-dependent code, so only HOST_LITTLE_ENDIAN is checked for the bitfields. However, the assembler output routines swap everything. As discussed during the front-end discussion, the md_numbers_to_chars is called for every segment, even the segments that contain no target-dependent information. This is because of the old version of the assembler. Therefore we still need a SWAP_ENDIANESS check on the whole header, also on the parts that do not contain any target-endianness dependencies.

- the text and data parts are both host and target endianness dependent. Therefore these are read in using the SWAP_ENDIANESS directive. The moves themselves are stored as bitfields in a word, so we need to guard them with a HOST_LITTLE_ENDIAN check and swap accordingly.

- the relocation data is stored on disk, independent of the target, always in big-endian format., therefore only HOST_LITTLE_ENDIAN is checked. Inside the relocation data, we also have to deal with bitfields, so also on those bitfields a HOST_LITTLE_ENDIAN check is required.

- the symbols data are stored as strings on disk. These strings are endianess independent. The only caveat here is that the symbol also contains a word with bitfields. This word needs a SWAP_ENDIANESS check. (not a HOST_LITTLE_ENDIAN check like other bitfields, since the whole data is not swapped on byte ordering during read, as the other are)

Concluding we can say that in general, only target-dependent data like the text and data segments, are both checked against host and target dependencies. Therefore we have to guard these reads with a SWAP_ENDIANESS. Other non-target related data need only to be guarded with HOST_LITTLE_ENDIAN. Bitfields, however, must be also checked against host endianness at all times. This check is implemented by changing the declaration of the structure at compile-time to match the bitfield-ordering of that specific endianness.

### 4.3.2 Scheduler

The scheduler is responsible for taking the internal data-structure representing the serial binary, and converting it into another data-structure that represents the parallel binary. This involves various steps but none of these steps work on the actual data in the binary. Values like addresses are already put in the data structures in a correct way by the binary reader, and the binary writer is responsible for converting the structures back into a binary. Therefore, the scheduling algorithms do no need any adaptations with respect to endianness dependencies.

### 4.3.3 Simulator

Simulation generates profiling information, like frequency count and memory dependencies. This profiling data can be written to disk to be used in subsequent runs of the scheduler or simulator. Therefore care must be taken when writing and reading these files. The routines `Prog::SaveProfile`, `Prog::ReadProfile`, `Prog::LoadMDeps`, `Prog::SaveMDeps` and `Prog::MDepsExists` are therefore guarded with checks on `HOST_LITTLE_ENDIAN` to make sure the profiling data is written to disk in a host independent fashion (namely big-endian), so profiling data can be used across platforms.

The simulator itself does not operate on files. It, however, has other dependencies on endianness. The simulator has to make sure it can offer the program under simulation an environment that is a correct image of the actual target platform. This means that parts of the simulator that represent a feature of the target platform, like the memory, the register files, the buses, have a target-endianness dependency. Most parts, however, have only one way to access them, e.g. register files and buses can only be written to by whole words at once, without the possibility to access smaller parts of these words. If one wants a smaller part of a register, this would already have been addressed in the actual move assembly, that the GCC front-end has generated correctly already. In other words, these components of the target architecture are word-addressable and nothing else.

The memory is the big exception here. Memory, although usually written to in whole words, is byte-addressable. Also, the MOVE architecture provides operations to access half-words and bytes in memory. This means every write or read to or from memory needs to be split up in bytes, which will then be written to the memory in a fashion depending on the target-endianness of the architecture.

Specifically, the memory class `SimMem` has member functions like `SimMem::WriteW`, `SimMem::WriteH`, `SimMem::WriteB`, `SimMem::WriteS` and `SimMem::WriteD` for respectively writing words, halfwords, bytes, single-precision floats and double-precision floats. Also, their read-counterparts are present. These functions all work by first getting a whole 32-bit quantity from memory, then byte swap depending on the `SWAP_ENDIANESS` predefine, and either read or write the correct (part of a) word. The implementation of the `SimMem` class can be found in appendix A.

### 4.3.4 Binary writer

The binary writer's job is to convert the internal data structure of the program into two files: a readable assembly output, usually called `b.txt`, and a parallel binary, suitable for feeding to an actual chip, called `b.out`. The data structures itself have no endianness dependencies, and the assembly output is plain text. The focus here lies on the parallel binary.

We assume that by setting the target endianness, we also specify the bit-order of the instruction stream. This means that if we write out a little-endian binary, both the byte-ordering in the instruction word (which can be fairly large, e.g. 128 bits for PcomP), and the bit-ordering per byte is little endian.

For the implementation this means, that we have to make the routine that outputs the so called `BitArray` endianness aware. The `BitArray` represents the actual move instruction word bits. This routine is the `OutputBinary(ostream &os, Insn *insn)` routine. This routine is responsible for allocating the `BitArray`. Class `BitArray` is already overloaded with the << operator. The change is that this overload function is altered to make it target-endianness aware, so that it can bit-swap the entire instruction word, if necessary.

## 4.4 Conclusions

The port of the MOVE framework to be both host and target-endianness independent is completed with success. The tools compile without any compile time options given on both big and little-endian hosts. The distribution is altered so that one compile-time switch will make both the front-end as the back-end target-endianness aware. This distribution then can be installed in a parallel directory tree on the same machine and a simple change to the shell's default search path can let the tools switch between the little and big-endian targets.

# Part III

# Immediates

# 5 Immediates overview

A large part of this thesis is devoted to the issue of encoding immediates in the MOVE processor. In this chapter, an introduction on immediates in MOVE is given. In section 5.1 and 5.2 and general overview on immediates and their implementation in other architectures is given, and in section 5.3 the current state of immediates in MOVE is given. The next two chapters will deal with the implementation of the new immediate framework and a review on the implemented code.

## 5.1 What are immediates

A processor usually has different ways to supply operands to its operations. Usual ways include register reference mode (`add r3, r1, r2`) and immediate mode (`add r3, r1, #234`).

An immediate is a way to pass a constant value directly from the instruction stream to an operation in the processor. To encode an immediate in the instruction stream, caution has to be taken. An immediate can take up relatively many bits of the instruction word, e.g. you only need 5 bits to encode 32 registers, but you need already 10 bits to be able to specify, e.g. constant 911. If you want to do an add of two constant 32bit values into a register, you need to encode, apart from the result register, 64 additional bits. To handle the encoding of large immediates into the instruction stream, different architectures have come up with different solutions.

Encoding immediates in the instruction stream poses several problems:

- As already mentioned, the code size will increase, since constants take a lot of bits, especially compared to (efficient) encodings for the address space for registers.
- If the immediate bits are separated from the operation that uses them, scheduling becomes

more difficult.

- We can distinguish between signed and unsigned immediates. Care has to be taken when sign-extending immediates to fit a certain immediate field.

## 5.2 Immediates in other architectures

Before we discuss the implementation of immediates, we will describe how other common architectures have solved the problem of immediates.

### 5.2.1 CISC

As an example on how a typical CISC machine has solved the immediate problem, we'll take the x86 instruction set as an example. The actual name CISC ("Complex Instruction Set Computer") indicates that its instruction set can be very specific and large. A common feature of CISC is that is has many different addressing modes. We'll take the example of the x86 "ADD" [Int97] instruction here, and list all possibilities of using this "ADD" instruction with an immediate operand.

| **Opcode** | **Instruction** | **Description** |
|---|---|---|
| 04 ib | ADD AL, imm8 | Add imm8 to AL |
| 05 iw | ADD AX, imm16 | Add imm16 to AX |
| 05 id | ADD EAX, imm32 | Add imm32 to EAX |

Table 5.1: Partial X86 ADD instruction reference

As you can see, this typical example of CISC deploys multiple immediate lengths per instruction, some of them even with their own opcode. Also noteworthy is the fact that the length of the instruction word depends on the size of the immediate used (which is by the way not uncommon for CISC machines, since they deploy this mechanism already to be able to encode various addressing modes)

### 5.2.2 RISC

As an example for immediates used in a typical RISC ("Reduced Instruction Set Computer") machine, we take the PA-RISC2.0 architecture from HP [PH97]. The following table lists some instruction formats of the PA-RISC2.0:

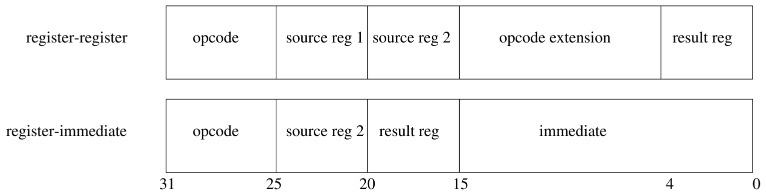


Figure 5.1: PA-RISC2.0 instruction format

Interesting features to point out here are:

1. All PA-RISC2.0 instructions are 32 bits wide. This already constrains the ways to encode an immediate in an instruction.

2. All instructions of a certain class have the same structure, with fields that specify source and destination registers, immediate constants and opcodes in the same bits of the instruction word.

3. Constants in register-immediate operations are always 16 bits long. This implies that the PA-RISC2.0 architecture cannot do direct arithmetic operations on constants larger than 16 bits. If it needs larger constants, then these have to be either constructed from smaller values, or stored in memory and loaded in a register, so that register-register type operations can be used.

### 5.2.3 VLIW

As in introduction to immediates in the MOVE architecture, a look at how typical VLIW ("Very Large Instruction Word") machines handle immediates is very useful, as the actual instruction word format of the MOVE resembles a VLIW very much. VLIWs are by design bound to the same principle of RISC: All instructions in an instruction word have to be the same size. Actually, a VLIW is nothing more (from an instruction word encoding point of view) than a series of RISC instructions (sometimes called 'atoms' or 'slots') concatenated. Examples of VLIWs are:

- The Intel IA-64 architecture (3 issue). This is not a genuine VLIW, but has interesting properties, also in relationship to MOVE.

- The Philips Trimedia (5 issue). The Trimedia is a DSP chip, MOVE is also targeted towards DSP applications.

- Texas Instruments C6x series (8 issue). The TI C6x series is also a DSP architecture.

We will discuss the IA-64 architecture in greater detail now:

The Intel IA-64 architecture, albeit not a genuine VLIW, is a VLIW based architecture, called the EPIC [1] architecture, and has a special type of operation called 'Extended', in which two slots together form a 60 bit immediate suitable for e.g. address displacements in a relative branch. A so called "template" of 5 bits at the beginning of the instruction word contains, amongst other information, information on what kind of instructions the various slots contain [Int00]. Note that you will see that the solution for immediates in the MOVE framework resembles the solution that the developers of the IA-64 architecture came up with.

Figure 5.2 shows a sample of IA-64 instruction format. The template "t" specifies what kind of instructions each slot contains. The special "instruction" `L+X` indicates that this slot encodes long immediate bits. The other instructions `A`, `B` and
M represent other type of instructions, like ALU instructions, branch instructions or memory instructions.

[1] Explicit Parallel Instruction Computing

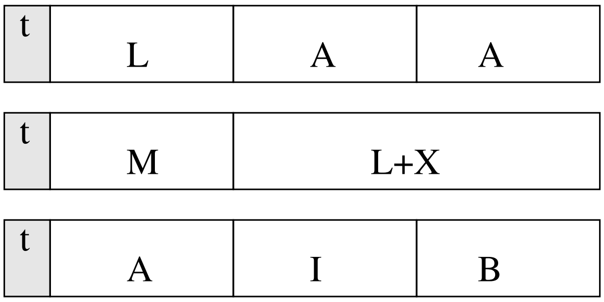


Figure 5.2: Sample IA-64 instruction stream; 128 bits wide

## 5.3 Immediates in MOVE

This section explains what the existing state of the implementation of immediates in the MOVE architecture was until now. Also, some key problems of the current state are highlighted, so a good motivation for a new implementation can be formed, before a new way of handling immediates is presented in chapter 6

### 5.3.1 Existing implementation

The MOVE architecture is from a instruction encoding point of view a VLIW. That means several fixed-width operations are concatenated to form a wide instruction word, with one slot per data transport (as opposed to one slot per functional unit in regular VLIWs). So we have the same problems as current RISCs and VLIWs have, as they were presented in the previous section. To highlight a few:

1. Fixed width slots. If an immediate is larger than the instruction slot width, one cannot encode the whole immediate in one slot. And this is without taking into account the space needed for the opcode, a destination register, et cetera.
2. Fixed instruction formats. Since MOVE has a standard instruction format to make the job of the Instruction Decode unit as simple as possible, it's not possible to assign arbitrary fields of a slot to immediate bits.

Some other considerations that make MOVE instructions different from other architectures:

1. Since a move consists of only one data transport, the only part of the instruction that can contain an immediate is the source of the move. This is different from an OTA based RISC architecture, where we usually have three operands: Two source operands, which both can contain an immediate, and a result operand, which cannot contain an immediate. Some instructions don't even have a result operand (e.g. the jump instruction).
2. There is no opcode field for the whole operation in a move. This makes it harder to actually let the processor know we have an immediate in a move instead of a reference to a functional unit socket.

The current solution offered for immediates in MOVE is the following:

1. Immediates shorter than the space reserved for the source socket can be encoded in the source field of the move itself. This is done by reserving part of the opcode space of the source field for constants. We call this type of immediate the “short immediate”.

2. Immediates larger than the short immediate are placed in one or more reserved fields that are concatenated at the end of the instruction word. This is encoded by added sockets to the instruction set, one for each immediate field you add to the instruction word. Each cycle the immediate fields are read (even if the actual information in that field is void in that cycle) and placed in an “immediate register”, which resides in the Instruction Fetch Unit. A source field of a move then can address the socket connected to that immediate register to use the immediate value.

This solution is visualized in figure 5.3.

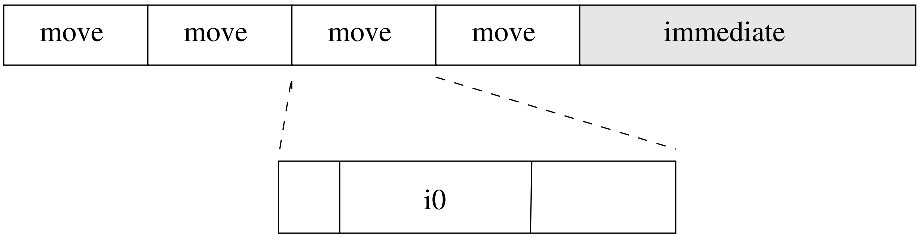


Figure 5.3: Dedicated immediate slot in instruction word

Algorithm 1 is a pseudo algorithm of the current long immediate scheduling, as performed in the function `FindImmMoveBus`. First, it is checked whether an immediate is able to fit in the source field of a move. If so, the standard movebus-allocation code is called. This allocation code includes adding a candidate mask for this move to the cycle and check via the Bipartite Matching Algorithm if a correct schedule is possible. If the immediate fits and the move can be scheduled correctly, a success is returned. If the move doesn’t fit in the source fields, we have a long immediate. All immediate registers are then iterated. For each immediate register, a check is done if the immediate register can hold the immediate under schedule. If this is also true, all that remains is check the movebus allocation with the candidate mask that relates to the chosen immediate register. By claiming the move and the immediate register, the dedicated field that contains the actual immediate bits is claimed implicitly.

The approach that is currently used for implementing immediates in MOVE suffers from various drawbacks that limit the optimal use of resources offered by the MOVE architecture. These drawbacks are:

1. Since every move instruction word contains a fixed immediate field, in cases where there are no immediates used, this field stays empty and bits in the instruction stream are wasted. Especially in embedded processors, program memory space is costly.

2. Since one move instruction word can contain only a limited number of (long) immediates (usually one), moves might have to be scheduled in another cycle, only because the long immediate value can not be put in that cycle, while all other resource requirements are met.

### 5.3.2 Possible solutions

Because of above mentioned problems, a new scheme to implement immediates in MOVE had to be devised, one which would address above mentioned limitations of the current implementation.

Initial thought on a new implementation can be read in [LC95]. This document describes basic options for implementing long immediates, without caring too much about practical implementation.

The following basic options for encoding long immediates were considered:

1. **Larger source.** Extend the size of the whole move slot to accommodate for the whole immediate to fit in the source field of the move.

2. Add a **dedicated immediate field** to the move instruction format. This is actually the implementation used historically, but has many problems, as seen in the previous section.

3. Use **multiple instruction formats**. This could be either replacing some move slots in the instruction word with immediate bits, or replacing a whole instruction word with immediates bits. The latter is the solution chosen by the "MicroMove" implementation of the MOVE architecture.

4. **Immediate construction.** Various move source fields (which can contain only a limited number of bits) can together make up one large immediate. This can be done either **sequential** or in **parallel**.

For a more in-depth discussion on these solutions, please seec̃itelimmoptions. This document also contains a preliminary overview of properties of above mentioned solutions, which will be repeated for clarity in table 5.2.

| property | Source fields | Dedicated imm. fields | Multiple instr. formats | Sequential imm. construction | Parallel imm. construction |
|---|---|---|---|---|---|
| area | + | + | + | − | + |
| instr. bandwidth | − | − | ± | + | ± |
| move bus utilization | + | + | − | − | − |
| latency | + | + | + | − | + |

Table 5.2: Properties of long immediate encoding options, qualitative indication

As concluded by the document, a practical solution would probably be a combination of the basic ideas of this table. So far, two implementations have been made. The first, dubbed the "resource variant", which is the one that this report is about, had a set of additional requirements, see section 5.3.3. The other implementation, dubbed the "pseudo-move variant", has been implemented by TNO-FEL. This implementation, and its performance in relation to the "resource variant", can be found in section 8. Both variants use a combination of the "multiple instruction format" and the "parallel immediate construction" ideas. The rest of this chapter will discuss the road to the "resource variant", and the next chapter will discuss its implementation.

### 5.3.3 Requirements of a new implementation

A list of requirements of the new immediate implementation was made:

### The instruction stream

1. An immediate that does not need more bits than the source field of the move should not take any space outside that move. This is what we call a “short immediate”. The other requirements listed deal with the concept of a “long immediate”, i.e. an immediate that does not fit in the move itself.

2. If an immediate does not fit in the source field of a move itself, extra bits of the instruction stream are needed. Preferably, these extra bits should be drawn from unused bits in the instruction stream.

3. A reserved slot just for immediates is optional, as these would be wasted in case there is no immediate to be scheduled.

4. A decoupling between immediate value and its move introduces extra state, which needs to address two things: A dependency between the immediate value and its move, and a bookkeeping mechanism that keeps track where the immediate value is placed in the instruction word and the instruction stream.

### The scheduling algorithms

The newly implemented algorithms should interfere as less as possible with the also existing scheduling algorithms. The current source has a lot of implicit assumptions that could cause unwanted effects if the new algorithms aren’t made as stand-alone as possible.

### The background of the assignment

Although the MOVE project needed a recode of the long immediate implementation because of reasons mentioned in the previous paragraphs, the problem became more urgent when an actual hardware implementation of the MOVE framework was to be deployed at NEC CCRL. The machine description of the NEC variant of the MOVE processor was already finalized. Although the implementation would be incorporated in the standard MOVE scheduler, rudimentary performance decisions were primarily made based upon a machine description of the NEC processor core.

After careful review of all options, a combination of the “multiple instruction format” and the “parallel immediate construction”, as explained in [LC95] has been chosen. The proposal for this new implementation, called the “resource variant” can be found in section 6.2.

**Algorithm 1** FindImmMoveBus

```
// check to see if the immediate fits in the source field
for all destination sockets do
   for all movebus do
      if not immediate fits in source field then
         continue
      end if
      check other resources
      if all resource demands met then
         add movebus to candidate mask
      end if
   end for
   try movebus allocation
   if movebuses are allocatable then
      add possible scheduling
   end if
end for
if possible scheduling was found then
   choose best possible scheduling
   assign resources
   return SUCCESS
else
   // immediate didn't fit in src field
   for all immediate registers do
      if not immediate fits in immediate register then
         continue
      end if
      for all destination sockets do
         for all movebus do
            check other resources
            if all resource demands met then
               add movebus to candidate mask
            end if
         end for
         try movebus allocation
         if movebuses are allocatable then
            add possible scheduling
         end if
      end for
   end for
end if
if possible scheduling was found then
   choose best possible scheduling
   assign resources
   // by claiming the ireg, the dedicated move slot is automatically claimed too
   return SUCCESS
else
   return FAILED
end if
```

# 6 The resource variant

This chapter will explain the implementation of the new long immediate encoding, the “resource variant”, in the MOVE scheduler. During the discussion of the implementation, various design decisions will be explained and motivated.

First, in section 6.1 some basic information on the MOVE scheduler will be discussed, so the context of the implementation will be clear. In section 6.2 the actual proposal of new data structures and algorithms for the “resource variant” will be discussed.

For an extensive reference to the MOVE scheduler’s algorithms and data structures, please read [Joh96].

## 6.1 Internal workings of the MOVE scheduler

Before we can start a discussion on the long immediate implementation, an understanding of the compiler internals is needed. The whole package “MOVE compiler” consists of several parts. Please refer to section 2.1.4 and in particular figure 2.5 to get a broad understanding of the compiler trajectory. In the next paragraphs we will discuss each part of the compiler and how immediates fit in.

### 6.1.1 GCC front-end

The front-end of the compiler trajectory is based upon GCC, version 2.7.0. It has been ported to a generic MOVE target. This target is representing intermediate move assembly code, packed in a binary format loosely based on the m68k a.out binary format. It features a serial stream of move instructions, working on a virtual machines that has enough registers to avoid spilling. The source field in the binary format is 32 bits wide, and as such the problem of "long immediates" is void here, as all immediates fit in the source field of the move. Here we assume the MOVE architecture has a maximum width of 32 bits, an assumption that holds throughout the whole MOVE framework at this point.

The front-end does not know anything about the immediate sizes of the target machine, and as such, can not distinguish between "short" and "long" immediates. This is not needed, since the front-end has no idea on how the instruction stream will be after scheduling. The problems stated in section 5.3.3 do not apply here.

It is not needed to explain the inner workings any further, it suffices to note that the emitted binary contains the immediate in the source field of the move, annotated with a flag that indicates whether the source field contains an immediate or a socket.

The a.out instruction format is the following:

```
#if HOST_LITTLE_ENDIAN
        short dst;
        char imm; // the immediate flag
        char grd;
#else
        char grd;
        char imm; // the immediate flag
        short dst;
#endif
        int src;
```

### 6.1.2 Scheduler

For a more in-depth working of the scheduler, please read [Cil00]. What will be discussed here is an overview on the scheduler's workings focused on the immediate support.

For a good understanding on how the Long Immediate algorithms hook in the Scheduler, algorithm 2 will briefly give an overview on the main loop of the scheduler. The algorithm iterates through each dependency-free move in each basic block of each procedure of the program, and will try to schedule that move. (A move is a node in the scheduler's Data Dependency Graph). To schedule, cycles are searched for enough resources, including units, sockets, and buses. If all requirements are met, a schedule is successful and a next node will be scheduled.

This is of course a gross simplification of the algorithm, but it shows in what steps a move is scheduled. For ease of understanding, issues like register allocation, backtracking or importing are left out, since they're not related to the long immediate implementation.

### 6.1.3 Simulator

The simulator can be split in two parts: A so-called "serial" simulator and a "parallel" simulator.

The serial simulator simulates the serial code as it was read in by the binary reader. This is nothing more than the exact binary code that the front-end emits. As we already discussed above, all immediates are considered equal and "short". Since the serial simulator only works on the code emitted by the front-end, unaltered by the scheduler, the immediate problem does not apply here, just as it didn't apply to the front-end of the MOVE compiler. All back-end code

**Algorithm 2** Scheduler algorithm

```
for all procedure in program do
  for all basic block in procedure do
    switch (scheduling scope)
    for all node = GetReadyOperation do
      switch (type of operation)
      for all moves belonging to node → operation do
        compute min_cycle and max_cycle of move
        for cycle = max_cycle downto min_cycle do
          for all units that can handle operation do
            for all sockets that are connected to unit do
              check various resources (FindMoveBus())
              for all movebuses connected to socket do
                tentatively assign operation to this movebus
                if movebuses are allocatable (AssignMBusses()) then
                  return SUCCESS
                end if
              end for
            end for
          end for
        end for
      end for
    end for
  end for
end for
```

that works with serial MOVE code, uses the MOVE code as provided by the GCC front-end, i.e. immediates always fit in the source field.

The parallel simulator simulates the code after it has been scheduled. It performs a cycle-accurate simulation of the scheduled "VLIW"-like code. Algorithm 3 shows the current parallel simulator algorithm. Note that the `while(not quit)` loop will be broken when an exit statement in the code is detected and the `quit` flag is set (in a different function).

As you can see the simulator is not completely cycle-accurate because immediates are read from the source field of the move, instead of making a distinction between short and long immediates. Short immediates are indeed read from the source field. Long immediates however are read from the immediate slot that was concatenated at the end of the instruction. Even without a new implementation, a better way to handle this would be decoupling the use of the immediate and the process of reading the immediate value from the instruction stream and storing it in a so called "immediate register". This register is then read when the immediate is used (which is for now, always in the same cycle).

### 6.1.4 Binary writer

The binary writing was also developed at NEC, concurrent with the initial work on the long immediates. Various hierarchical `OutputBinary()` functions iterate through the whole pro-

**Algorithm 3** SimulatePar(Proc*, int offset)

```
instruction = SkipEntryInstructions(proc,offset)
block = instruction-¿blck
while not quit do
  for all move in instruction do
    get src value from either register, immediate or functional unit
    put value on movebus
  end for
  for all move in instruction do
    get value from movebus
    if register type == operand/trigger register then
      feed value to operand/trigger register
    end if
    if register type == jump/call/trap then
      set jump latency counters
    end if
  end for
  advance functional unit pipelines with one cycle
  update jump latency counters
  if one of the jump counters == 0 then
    jump accordingly to address
  else
    instruction++
  end if
  if end of basic block reached then
    get default successor block and make it current
    instruction = first instruction from new block
  end if
end while
```

gram.

`OutputBinary(ostream &, Insn *)` is the function that is responsible for outputting instruction words. It is called for every instruction in a basic block. This function has always included some support for the new long immediate implementation, because it was developed when the initial data structures were already specified. The original algorithm of `OutputBinary(ostream &, Insn *)` can be read in algorithm 4

## 6.2 The resource variant

In this section, the new implementation of immediates in the MOVE framework will be discussed. Certain design decisions imposed by the list of requirements in the previous chapter will be explained. Basically, the following implementation method is chosen: (figure 6.1 illustrates this implementation)

**Algorithm 4** OutputBinary(ostream &, Insn *)

allocate a `BitArray` to hold the bits for this instruction word
**for all** moves in this instruction **do**
  call `OutputBinary(ostream &, Move*)`
  add this slot to the list of occupied slots
**end for**
**for all** slots not in list of occupied slots **do**
  encode a `NOP` to this field
**end for**

1. Just as in the existing method, placing the immediate in the instruction stream and using the value from the immediate register are decoupled. We call the former action the "IReg Write" (immediate register write) or "Immediate Define" and the latter action the "IReg Read" or "Immediate Use".
2. The dedicated immediate slot at the end of an instruction word can be replaced by reserved slots for immediates in the instruction stream. Certain move slots in the instruction word now can either be a move, an immediate or a part of an immediate.
3. To bookkeep this, every instruction word gets a tag, usually of the length of a couple of bits. The tag is called Long Immediate Control Tag, or "LIT" when abbreviated. This tag specifies which move slots contain immediates and to which Immediate Register they are written
4. The new Long Immediate Control Tag logically contains a list of so called "Long Immediate Micro Operations", which basically specifies a bit mask that indicates which move slots are occupied by immediate bits in that cycle, and to what immediate registers these slots are written. One LIT tag can contain multiple micro operations, and this way a LIT can write more than one Immediate Register at a time.
5. When the Instruction Fetch Unit reads the tag, and immediates are detected, the Unit fills the appropriate Immediate Register with the value from the instruction word. One immediate can be constructed out of multiple move slots. In this case, the move slots making up that one immediate must be scheduled in the same cycle.
6. There is no need to use the immediate in the same cycle. The Immediate Registers are part of the state of the machine.
7. When the immediate is used, the source socket will be that of the appropriate Immediate Register, and its value will be placed on the move busses.
8. Although a long immediate occupies a move slot, it is not represented in the move list of a cycle. (the move list is the data structure that holds a list of moves per cycle in the scheduler). A long immediate is only defined by the presence of a LIT tag at that cycle.

### 6.2.1 GCC front-end

As explained in the previous section, the front-end needs no adaptations for the new long immediate format.

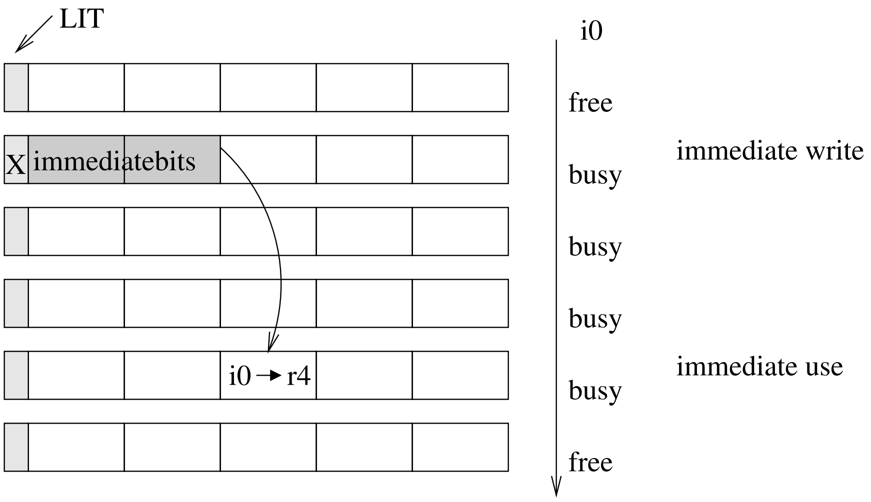


Figure 6.1: Scheduling of long immediates

### 6.2.2 Binary reader

The binary reader needs no adaptations either, since immediates are processed and converted only by the scheduler. As said, all serial MOVE code in the back-end has its immediates in the source field of the move.

### 6.2.3 Mach file

The MOVE framework uses various auxiliary files during scheduling. One of the most important ones is the `mach` file, the machine description file. This file specifies how many movebuses, how many sockets, which functional units, etcetera, a specific instance of the target architecture has.

This file also specifies how many Immediate Registers the specific architecture has. This used to be a very simple specification, because it only had to specify the number of Immediate Registers and their size and socket. The place in the instruction stream where the immediate bits would end up was already determined, in a dedicated immediate field next to the instruction word.

Now the `mach` file has to specify:

1. How many immediate registers there are, together with their sockets, size and whether they are signed or unsigned (this is identical to the old situation)
2. The specification of the LIT tag, which indicates how Immediate Registers are written, by means of the so called "micro operations".

An example of the old format and the new format follow, after which a formal specification of the new format will be presented. This example reflects the PcomP specification.

The old format specifies three immediate registers, and implies three dedicated fields:

```
ImmediateUnits
{
        i1              32, signed, ir_1;
```

```
        i2              20, signed, ir_2;
        i3              20, signed, ir_3;
}
```

The new format specifies the same three immediate registers, but now the “Control” field specifies how these three immediate registers can be written to, by means of an encoding in the LIT tag:

```
LongImmediate
{
 Registers:
    i0  20, signed, ir_0;
    i1  20, signed, ir_1;
    i2  32, signed, ir_2;
 Control:
    {};
    i0 20 : { 4 };
    i1 20 : { 5 };
    i0 20 : { 4 }, i1 20: { 5 }, i2 32: {4,5};
}
```

The long immediate specification is specified in the `LongImmediate` block. Inside, the following formal specification is used:

The first part, which is similar to the existing format, is introduced by '`Registers:`' and contains lines of the format:

```
<ir_name><immediate_length>,<immediate_signed>,<sock_name>;
```

the second part is introduced by '`Control:`' and contains lines of the format:

```
[<ir_name>:]{[<slot_num>[,...]]}[,...];
```

the labels have the following meanings:

| | |
|---|---|
| **ir_name** | A string which gives the name of the immediate register. |
| **immediate_length** | An integer which indicates the length of the immediate register. |
| **immediate_signed** | A member of the set {signed,unsigned} which indicates whether the immediate bits should be sign- or zero-extended to *immediate_length*. |
| **sock_name** | The name of the socket to which the immediate register is connected. |
| **slot_num** | An integer which indicates the instruction field (move slot or immediate reserved field) from which the immediate bits are read. A comma separated list of field numbers specifies that the immediate is constructed by concatenating the field contents. This integer is counted from zero. |

Each line of the control part can contain one or more comma-separated elements. An element, called *micro-operation*, specifies three things: (1) the destination immediate register, (2) a number before ':', which specifies the number of significant bits contained in the instruction field(s), (3) the list of fields from which the immediate bits are read. Bits from these fields are concatenated to form one long immediate.

An important special case is the empty line, in which no immediate register and no field list is present. This line represents a "no-operation", whereby no immediate register is written (reserved fields are wasted, but move slots are all available for transport programming). An empty line *must* be present if there is more than one line in the control part. This specifies that no immediates are written in that cycle.

### 6.2.4 Data structures

The MOVE scheduler uses four main data structures to hold the state of the machine, the program and the scheduling. They will be introduced here and the changes needed by the new long immediate implementation are mentioned.

For a complete reference to the changed data structures, see appendix B.

#### 6.2.4.1 Mach

The `Mach` class contains the machine description. The class is defined in file `mach.h`. It completely describes the target machine.

**LImmControl** The `Mach` class is augmented with a structure called `immctrl` of type `LImmControl`. This is the class that implements the tag in each instruction that gives information on what move slots in the novelist represent immediates or parts thereof, and whether they are signed or not. Its data members are:

- **`int slots`**, which defines the bit mask of written slots. The `int slots` member is nothing more than an OR'd mask of individual bit masks of each Micro Operation.
- **`LImmMOpList mops`**, which contains a list of Micro Operations.

**LImmMOp** Class **`LImmMOp`** is the atomic "Long Immediate Micro Operation" that writes to an Ireg from a certain bit mask. Its main members:

- **`nbits`**, which specifies how many bits this operation can write.
- **`slots`**, which contains the bit mask that specifies which move slots contain immediates bits.

**IReg** Class **`IReg`** is the class that implements the Immediate Register. It is amended with a data members and three member functions to read that data. The data member is:

- **`LImmMOpList mops`**, which lists all Micro Operations that can write to this IReg.

The member functions are:

- **`IsImmediateFits(Move*)`** and **`IsNotImmediateFits(Move*)`**, which take a `Move` and return a boolean indicating if the immediate in that Move fits in the IReg
- **`LImmMOpIter PossibleEncoding(int size)`**, which returns a pointer to the first micro operation that can write this IReg and encodes at least `size` bits.

#### 6.2.4.2 Prog

The `Prog` data structure holds the scheduled information of the program. Its substructures are `Proc`, `Blck`, `Insn` and `Move`, for respectively the procedures, the basic blocks, the instruction words and the individual moves.

- We now amend the `Insn` with a pointer to one **`LImmControl`** instance, to implement the presence of the LIT tag. This way, a scheduled instruction has the necessary information to know which move slots are occupied by immediates by means of the `slots` member of the LImmControl class.
- We also amend the `Insn` with a list of data flow edges **`DFlow`** of immediate uses, so we have an easy way to find the use (i.e. `IReg` read) or uses of an immediate.
- The `Move` class is augmented with a back-pointer to the **`Insn`** where the immediate was defined.

#### 6.2.4.3 RTabEntry

The `RTabEntry` data structure holds all resource information on each cycle during scheduling. This includes state like occupied sockets, move busses, immediate registers and state of the function units in that cycle. It also provides functions to claim or release resources.

- The `RTabEntry` structure is also amended by a **`LImmControl`** pointer. This way, a cycle under scheduling has all the necessary information on the use of immediates in that cycle.
- Also, the `RTabEntry` structure's functions that operate on **`IRegs`** are extended so that the array that holds the occupation of `IRegs` now is implemented with a reference counter instead of a boolean. This eases the amount of work needed if immediate-sharing is ever implemented. Immediate-sharing means that a long immediate is only defined once in the instruction stream and only once written to the `IReg`, but used multiple times by subsequent reads from this `IReg`. More information on this in section 7.2.2.

#### 6.2.4.4 DDG

The `DDG` is the Data Dependency Graph and holds all interrelations between moves and basic blocks during scheduling. A move is represented by the class `Node` and a basic block is represented by the class `SNode`. Since we have implemented the long immediates as nothing more than a presence of a LIT tag, there is no need to change the `Node` or `SNode` classes.

### 6.2.5 Scheduler algorithms

Now that we have all extensions to the data structures explained, we will explain the modifications made to the existing algorithms and we will discuss all new algorithms. As stated in the list of requirements, our goal was to interfere as less as possible with the existing algorithms. This is done mainly by two design decisions:

1. All scheduling of long immediates is done after the long immediate use is scheduled. The last step in scheduling a move is finding a movebus for it. So in `FindImmMoveBus` we add the hook that will schedule the IReg Write.

2. All operations performed on the list of moves in a cycle do not know about the presence of a long immediate, since it is not in the move list at all. It turns out the only place where allocation of moves into slots is done is during the `Marriage` function (which is called from `RTabEntry::AssignMBusses()`). This function is called with a testing-flag, which is true when this function is called during scheduling and false when this function is called after scheduling, during placement. By replacing the real move list (implemented by the `cand` mask and `nmoves` index) with a pseudo list that includes bit masks for long immediates, we have a atomic and stand-alone way to check for resource requirements of movebuses.

The rest of this section will discuss the algorithms used during scheduling and unscheduling.

#### 6.2.5.1 Scheduling

The main algorithm associated with the long immediate implementation is the immediate write scheduling algorithm. We will present all algorithms and helper functions here.

The name "resource variant" already indicates that immediates are not represented by (pseudo-)moves but only by a resource claim in the `RTabEntry` table. An immediate in the serial code (e.g. `#456 -> add_o`) is split in half: `i?? -> add_o` (the immediate use) and `#456 -> i??` (the immediate write) [1].

This split will be made in the inner-most loop of the scheduling algorithm, which is `FindImmMoveBus()`

The overall algorithm is presented first in algorithm 5, after which various parts will be explained in detail. This algorithm can be compared to algorithm 1 and differs from it in the following ways: After the resources on the immediate register are checked, and a possible move bus allocation (with the candidate mask relating to the chosen immediate register), there is not commit yet, but a tentative assignment. From that point, a cycle from the current cycle down to cycle zero is searched in which the immediate bits can be written. For each cycle on in that write-read (def-use) chain, the resources on the immediate registers are checked, and a check for a suitable encoding is checked. The latter is done by tentatively assigning a new LIT tag and see if the movebuses in that cycle are still allocatable, as well as a check if all existing immediate writes in that cycle are still preserved. If all these checks succeed, the immediate write and immediate use are declared "final", and the function will return "success" on the schedule of that immediate.

The implementation for this algorithm has been coded in 5 functions:

1. `FindImmMoveBus`, taking care of the search for an Immediate Register and claiming all resources for the immediate use.
2. `ScheduleLImm`, taking care of iterating through the cycles and checking on ireg resources in cycles between the read and write of the immediate.
3. `FindIRegWriteBus`, taking care of tentatively assigning a LIT tag and claiming the Immediate Registers in the path between the read and write of the immediate.

[1] `#456` here represents an arbitrary immediate larger than the maximum size for a short immediate, and `i??` represents an arbitrary immediate register

**Algorithm 5** Overall scheduling of long immediates

```
FindImmMoveBus()
try to fit immediate in src field as short immediate
if not fits then
  for all iregs do
    assign socket of ireg to source socket
    check resources on ireg
    if possible allocation for immediate use found then
      tentatively claim resources for immediate use
      for this cycle downto mincycle do
        check if ireg is still free
        check if encoding is available
        tentatively assign LIT tag
        if movebuses allocatable then
          commit schedule of immediate use and write
          return SUCCESS
        end if
      end for
    end if
  end for
end if
return FAILURE
```

4. `IsLImmControlValidSubset` which can determine if a tentatively assigned LIT tag indeed encodes all bits needed for the immediates currently under scheduling *and* still encodes immediates that were already scheduled in that cycle
5. `RTabEntry::AssignMBusses`, a function now extended with functionality that makes sure that long immediate fields also are taking into considerations when a move list is mapped on move buses.

We will now go into detail on each of these 5 functions.

**FindImmMoveBus** In `FindImmMoveBus()` an immediate register is chosen and `ScheduleLImm` is called. If this returns with a success return code, the immediate register is claimed and scheduling is successful. The pseudo code is can be read in algorithm 6.

**ScheduleLImm** The `ScheduleLImm` function is responsible for iterating through all cycles in order to find a suitable cycle for the immediate write, i.e. a cycle where the immediate bits can be encoded. To check if a certain cycle is suitable, it first checks if the Immediate Register that was chosen is still available and then checks if the resources for the immediate write by calling `FindIRegWriteBus`. The pseudo code for this algorithm can be found in algorithm 7.

**FindIRegWriteBus** The function `FindIRegWriteBus` takes care of choosing a suitable LIT tag. It can do this by upgrading an already existing LIT to one that encodes the mi-

**Algorithm 6** FindImmMoveBus(move, cycle)

```
try to fit immediate in src field as short immediate
if not fits // try to schedule a long immediate then
   for all ireg do
      replace src_sock with socket of ireg
      check resources on ireg
      if possible allocation for immediate use found then
         add ireg to list of possible iregs
      end if
   end for
   for all iregs in possible-iregs-list do
      tentatively claim resources for immediate use
      call ScheduleLImm
      if ScheduleLImm succeeds then
         make iregs resource claim permanent
         return SUCCESS
      end if
   end for
end if
```

**Algorithm 7** ScheduleLImm(read_node, ireg)

```
for cycle = read_node-¿cycle downto zero do
   if ireg in this cycle is busy then
      return FAILURE
   end if
   if FindIRegWriteBus succeeds then
      return SUCCESS
   end if
end for
return FAILURE
```

cro operations as specified by the existing LIT as well as the micro operation that can write to the immediate register currently under scheduling. It does this by means of the function `IsLImmControlValidSubset`. If these tests all succeed, it tentatively assigns that LIT tag to the cycle and checks if all the movebuses are still allocatable. If all this succeeds, this function returns a success return value. The pseudo code for `FindIRegWriteBus` can be found in algorithm 8.

**IsLImmControlValidSubset** The function `IsLImmControlValidSubset` is perhaps one of the trickier algorithms discussed here. It takes three arguments, two LIT tags and an immediate register. It has two tasks. One is to check if a certain LIT tag can encode all immediate registers that another LIT tag can encode, plus the immediate register passed to the function. The other one is its dual, namely to check if a certain LIT tag can encode all immediate registers that another LIT tag can encode, *except* the immediate register passed to the function. Both tasks

**Algorithm 8** FindIRegWriteBus(write_cycle, read_cycle, ireg)

```
if current LIT already writes this ireg (as a side effect) then
  return FAILURE
end if
for all micro operations that write this ireg and can encode enough bits for this immediate do
  fetch the accompanying LIT tag
  if not IsLImmControlValidSubset then
    continue with next micro operation
  else
    tentatively assign new LIT tag
    if move buses are allocatable // call AssignMBusses then
      for cycle = write_cycle downto read_cycle do
        claim RTabEntry-¿ireg
      end for
      return SUCCESS
    else
      revert to old LIT tag
    end if
  end if
end for
return FAILURE
```

can be done by the same function, and you can get the dual function by swapping the two LIT tags on the argument list. Pseudo code for `IsLImmControlValidSubset` can be found in algorithm 9.

**AssignMBusses** The member function `RTabEntry::AssignMBusses` is responsible for checking if a certain set of moves, together with their constraints on connectivity of their sockets to move buses, can be mapped on the move buses. For this, it uses the "Marriage" algorithm, also known as the "Bi-partite Graph Matching Algorithm".

Each move in the move list has a mask that indicates which move buses it can be scheduled on. (this is because normally, a MOVE processor has no full connectivity between sockets and move buses.)

In the original situation, one would call the marriage function with the graph of edges between move buses and moves, and see if one can find a bi-partite mapping of this graph.

The problem we now have is that long immediate writes, although not part of the move list, do occupy slots in the instruction word, or equivalently, buses on the transportation network. For this, the `AssignMBusses` function is extended with a small routine that temporarily adds a mask for each long immediate write, thus creating a "pseudo move", before calling the marriage function.

Since the mask for this slot only contains the slot itself, this will be the only mapping the "Marriage" function can find. The result is that the Marriage function cannot map any other move on this bus anymore, which is exactly the desired behavior.

After the marriage function is called, these "pseudo moves" are removed, in order to restore

**Algorithm 9** IsLImmControlValidSubset(LIT super, LIT sub, ireg cur)

```
for all iregs do
  if this ireg is not written by sub then
    if this ireg is written by super then
      if this ireg is the "cur" ireg and this ireg is busy then
        return FAILURE // super LIT defines too much
      else
        continue // this is the ireg that was supposed to be added
      end if
    else
      continue // this ireg is not involved at all
    end if
  else
    // this ireg is written by sub
    if this ireg is not written by super then
      return FAILURE // super LIT is not a superset
    else
      // current ireg is written by both LITs
      if the micro operations from both the super and the sub LIT can encode the same
      number of bits then
        continue // this ireg has passed the compliance test
      end if
    end if
  end if
end for
// at this point we have checked all iregs for compliance
// and they all passed
return SUCCESS
```

the original move list and masks.

The new pseudo code for this function can be found in algorithm 10.

**Algorithm 10** AssignMBusses(RTabEntry)

```
for all move buses do
  if move bus is in the slots list of the LIT tag then
    add mask with this bus to the set of mask
    increment the number of moves in this list
  end if
end for
perform the marriage function
restore the old move list
```

#### 6.2.5.2 Unscheduling

During scheduling, it is possible that the scheduler needs to come back on previous decisions to schedule. During bypassing, for instance, the bypassed node is declared "dead", or "killed", but if the bypass doesn't succeed in the end, that node needs to be declared "unkilled".

**KillNode** The process of unscheduling nodes is done by the function `KillNode()` Normally, unscheduling a node involved releasing all resources occupied by that node. Note that although resources in the `RTabEntry` table are released, but all information on these resources is still preserved in the data members of the `Move`. These data members include the `cand` mask for the movebuses, the `src_sock` and the `dst_sock` and finally the `ireg` which indicates what Immediate Register was used. The process of unscheduling nodes is done by the function `KillNode()`

The extension made to `KillNode()` is that, instead of just releasing the Immediate Register in the current cycle, we also look up where the immediate write was performed and we release all resources (LIT tag, immediate register claims) related to this immediate read. For this, we define a helper function `LookupIRegWrite`.

**LookupIRegWrite** The function `LookupIRegWrite` will find a defining cycle for a given ireg, and will optionally release all resources associated with this immediate read. It can be views as the dual of `FindIRegWriteBus`.

It will do this by iterating through the cycles from the read node downwards. We release (if asked) all immediate registers found, until we find a cycle that has a LIT tag that writes to the immediate register in question. At that point, we (if asked) downgrade the LIT tag to another tag that still writes to all immediate registers that the original did, except for the immediate register that is being released. Here we can profit from the dual functionality of `IsLImmControlValidSubset`.

The pseudo code for `LookupIRegWrite` can be found in algorithm 11. The algorithm takes as in-arguments the ireg that needs to be released, the read-node and a boolean that indicates whether we are only looking up the write or that we are actually releasing all resources associated with this ireg. The function returns the `snode` (basic block) and the cycle where the LIT tag was found.

**UnkillNode** Because killed moves are simple marked "dead", but not actually removed from the schedule, it's very easy to reschedule deleted nodes. The function `UnkillNode()` just claims back all resources used by the `Move` to be unkilled in the corresponding `RTabEntry()`.

A relevant problem of the new long immediate encoding is that it makes immediate registers a multiple-cycle resource, like general purpose registers, and multiple-cycle resources are more difficult to manage. When the def-use chain of the immediate write and use are killed, a release of the resources implicates that all information on where the immediate write was scheduled is lost.

That's why `UnkillNode()` has to completely reschedule the long immediate write by calling `ScheduleLImm()`. The scheduler requires that any node that has been marked "dead" can be successfully unkilled, this means that this call to `ScheduleLImm()` can not fail. For-

**Algorithm 11** LookupIRegWrite(read-node, bool release, cycle, snode)

```
write-cycle = cycle
for cycle = write-cycle downto zero do
  if release then
    release ireg in this cycle
  end if
  for all micro ops from LIT tag in this cycle do
    if this micro op writes to the ireg then
      we found the defining cycle
      for all LIT tags do
        if IsLImmControlValidSubset(current LIT, new LIT, ireg) then
          assign new LIT
          return SUCCESS
        end if
      end for
      if no suitable LIT tag to downgrade found then
        abort // this cannot happen
      end if
    end if
  end for
end for
if no defining cycle found yet then
  abort // this cannot happen
end if
```

tunately, this is impossible, since it the state of resources between a `KillNode()` and an `UnkillNode()` will not change.

Scheduling will be aborted when `ScheduleLImm()` fails, but this has never been observed in any benchmark, so it is assumed that the assumption in the previous paragraph holds.

### 6.2.6 Simulator algorithms

During the discussion of the working of the simulator it became clear that only the parallel simulator needs adaptation. The parallel simulator function `SimulatePar()` was extended with a global array `ireg[]` that represents the state of the Immediate Registers. For each cycle this array contains the values of the various Immediate Registers. The algorithm is also changed:

1. At the beginning of each cycle, a check for a LIT is done, and if found, the `ireg[]` array is filled with the appropriate values. This concludes the "immediate write" stage. Actual values of immediates are found by following the `DFlw iruses` field of the `Insn`. Since this is the exact same way that we build the binary in the binary writer stage, this method represents a valid way of looking up actual immediate values.
2. During the processing of each move, when an immediate source is detected, the original action was to put the value of the src-field of that move on the movebus. This behavior is still valid for long immediates, since the internal representation stores the value of the

immediate in the source field, even if the socket related to that src-field now is an immediate register socket. We now split this case in two separate cases: If the immediate is a short immediate (no presence of a `move->ireg` pointer), we just copy the value of the src-field to the movebus; but if the immediate was a long immediate, the appropriate value from the `ireg[]` array is read and put on the movebus. This concludes the "immediate read" stage.

Because the only architectural visible change to the system was the presence of Immediate Registers, the adaptation to the parallel simulator was very straightforward.

The changes to the algorithm are explained in algorithm 12, please refer to algorithm 3 for the original algorithm.

### 6.2.7 Binary writer

The binary writer's classes need to be extended with functionality for writing the LIT tag at the beginning of the move instruction word, and with functionality for filling move slots with actual bits if this move slot contains immediate bits.

The `mapgen` utility checks the `mach`-file for Long Immediates and extends the size of an instruction word to accommodate for dedicated immediate fields, i.e. immediate fields that are not shared with the normal move slots.

The initial version of `OutputBinary(ostream &, Insn *)` first allocates a `BitArray` that represents the instruction word and uses `OutputBinary(ostream &, Move *, BitArray &)` to fill all move slots. After that, it fetches the `LImmControl` tag from the current `Insn`. The functionality is extended by an algorithm that will compute which move slots actually contain immediate bits and it will fill those fields with the value from `insn->iruses` (the list of pointers to the moves that this instruction encodes immediate bits for). The new `OutputBinary(ostream &, Insn *)` will be shown in algorithm 13. Please compare to the original implementation as shown in algorithm 4.

**Algorithm 12** SimulatePar(Proc*, int offset)

```
instruction = SkipEntryInstructions(proc,offset)
block = instruction-¿blck
while not quit do
  write immediate bits from long immediate fields in ireg[]
  for all move in instruction do
    if move is a long immediate read then
      get src value from ireg[]
    else
      get src value from either register or functional unit
    end if
    put value on movebus
  end for
  for all move in instruction do
    get value from movebus
    if register type == operand/trigger register then
      feed value to operand/trigger register
    end if
    if register type == jump/call/trap then
      set jump latency counters
    end if
  end for
  pass "clock pulse" to functional units
  update jump latency counters
  if one of the jump counters == 0 then
    jump accordingly to address
  else
    instruction++
  end if
  if end of basic block reached then
    get default successor block and make it current
    instruction = first instruction from new block
  end if
end while
```

**Algorithm 13** OutputBinary(ostream &, Insn *)

```
allocate a BitArray to hold the bits for this instruction word
// we first encode all long immediate bits in the BitArray
for all iruses pointers in this instruction do
    look up value to be encoded from src field in iruses-¿move
    look up slots that write this ireg in this cycle
    add slots to the list of occupied slots
    encode bits into instruction stream.
end for
for all moves in this instruction do
    call OutputBinary(ostream &, Move*)
    add this slot to the list of occupied slots
end for
for all slots not in list of occupied slots do
    encode a NOP to this field
end for
```

# 7 Long immediates review

This chapter will evaluate the implementation of long immediates as described in chapter 6. This will be done by means of a quantitative benchmark which will evaluate instruction count and code size . Also, this chapter will present a comparison with an equivalent, independent, implementation of long immediates. Lastly, motivations and guidelines for future work on the long immediates support are given.

## 7.1 Performance review

This section will give a quantitative review of the “resource variant”. First, an overview on the used benchmark suite and the machine descriptions are given. Then, the actual data is presented in several graphs. Subsection 7.1.3 will draw some conclusions from the gathered results.

### 7.1.1 The benchmark suite

The implementation was tested with four different machine configurations. The relevant parts of those machine descriptions can be found in appendix C. A quick overview on the specifics of these mach files:

1. `mach.pcomp`, the PcomP architecture for which this implementation was written originally. It features 6 buses, and 3 immediate registers. Two 20-bit immediate registers can be written in the same cycle via slot 4 and 5, or one 32-bit immediate register can be written via a concatenated slot 4 and 5. (note that slots are counted from zero, so slot 4 and 5 are the two most significant buses).

2. `mach.one`, the PcomP architecture with only 1 32-bit immediate register, which is written from slot 5
3. `mach.small`, a small architecture with 3 buses and 1 32-bit immediate register.
4. `mach.big`, an architecture with 8 buses and 2 independent 32-bit immediate registers.

These 4 machine descriptions were benchmarked against the new "resource variant" and against the old implementation with dedicated move slots. For the old implementation, the same number of dedicated move slots were added to the machine description as the number of immediate registers in the new variant. The benchmark suite consists of the following benchmarks:

- arfreq
- g722main
- music
- radproc
- edge
- expand
- flatten
- smooth
- cjpeg
- djpeg
- go
- compress
- m88ksim

The tests were conducted to measure the code-size, measured in number of instructions. This means that in the new implementation, the number of instructions will increase, since normally a dedicated immediate slot was used, where now the immediates are scheduled in the instruction stream itself. But since the dedicated slots are not needed anymore, the instruction-word size will decrease. Therefore also the multiplication of number of instructions and the instruction-word size is presented. This metric gives a real indication of the achieved improvement in terms of program memory savings.

The exact metrics of the 4 machine descriptions are presented in table 7.1.

| **mach file** | **number of move slots** | **move slot width** | **total width** | **dedicated fields** | **total width incl. ded. fields** |
|---|---|---|---|---|---|
| `mach.pcomp` | 6 | 20 | 120 | 32 | 152 |
| `mach.one` | 6 | 20 | 120 | 32 | 152 |
| `mach.small` | 3 | 32 | 96 | 32 | 128 |
| `mach.big` | 8 | 32 | 256 | 64 | 320 |

Table 7.1: Metrics of machine descriptions

### 7.1.2 The results

The four tables 7.2 to 7.5 will show the instruction count increase, and the total code-size when the instruction count is multiplied by the instruction word length in bits. Both sets of measurements are also presented relative to the old implementation. Each table shows the results for one machine description. Note that not all benchmarks are represented in each machine description. Since the MOVE framework is a project still in development, not all benchmarks could complete correctly. In that case that benchmark is removed from the suite for that machine description.

| benchmark | old instr.count | old codesize | new instr.count | new codesize | relative instr.count | relative codesize |
|---|---|---|---|---|---|---|
| arfreq | 988 | 150176 | 1005 | 120600 | 101.72 | 80.30 |
| g722main | 4057 | 616664 | 4214 | 505680 | 103.87 | 82.00 |
| music | 4085 | 620920 | 4163 | 499560 | 101.91 | 80.45 |
| radproc | 2613 | 397176 | 2675 | 321000 | 102.37 | 80.82 |
| edge | 4192 | 637184 | 4332 | 519840 | 103.33 | 81.58 |
| expand | 3926 | 596752 | 4029 | 483480 | 102.62 | 81.01 |
| flatten | 3269 | 496888 | 3373 | 404760 | 103.18 | 81.45 |
| smooth | 2950 | 448400 | 3067 | 368040 | 103.96 | 82.07 |
| cjpeg | 8409 | 1278168 | 8556 | 1026720 | 101.75 | 80.32 |
| djpeg | 9824 | 1493248 | 9996 | 1199520 | 101.75 | 80.32 |
| compress | 3571 | 542792 | 3696 | 443520 | 103.50 | 81.71 |
| **averages** | | | | | 102.72% | 81.10% |

Table 7.2: `mach.pcomp` benchmark results

| benchmark | old instr.count | old codesize | new instr.count | new codesize | relative instr.count | relative codesize |
|---|---|---|---|---|---|---|
| arfreq | 984 | 149568 | 997 | 119640 | 101,32 | 79,99 |
| radproc | 2617 | 397784 | 2658 | 318960 | 101,56 | 80,18 |
| edge | 4255 | 646760 | 4354 | 522480 | 102,32 | 80,78 |
| expand | 4004 | 608608 | 4023 | 482760 | 100,47 | 79,32 |
| flatten | 3330 | 506160 | 3360 | 403200 | 100,90 | 79,65 |
| smooth | 3001 | 456152 | 3040 | 364800 | 101,29 | 79,97 |
| cjpeg | 8425 | 1280600 | 8511 | 1021320 | 101,02 | 79,75 |
| djpeg | 9855 | 1497960 | 9979 | 1197480 | 101,25 | 79,94 |
| go | 41035 | 6237320 | 41618 | 4994160 | 101,42 | 80,06 |
| compress | 3602 | 547504 | 3661 | 439320 | 101,63 | 80,24 |
| m88ksim | 12060 | 1833120 | 12155 | 1458600 | 100,78 | 79,56 |
| **averages** | | | | | 101.27% | 79.95% |

Table 7.3: `mach.one` benchmark results

| benchmark | old instr.count | old codesize | new instr.count | new codesize | relative instr.count | relative codesize |
|---|---|---|---|---|---|---|
| arfreq | 1160 | 148480 | 1237 | 118752 | 106,64 | 79,98 |
| expand | 4947 | 633216 | 5250 | 504000 | 106,12 | 79,59 |
| flatten | 4209 | 538752 | 4600 | 441600 | 109,29 | 81,97 |
| smooth | 3943 | 504704 | 4270 | 409920 | 108,29 | 81,22 |
| cjpeg | 11042 | 1413376 | 11517 | 1105632 | 104,30 | 78,23 |
| compress | 4357 | 557696 | 4638 | 445248 | 106,45 | 79,84 |
| **averages** | | | | | 106.85% | 80.14% |

Table 7.4: `mach.small` benchmark results

| benchmark | old instr.count | old codesize | new instr.count | new codesize | relative instr.count | relative codesize |
|---|---|---|---|---|---|---|
| arfreq | 1078 | 344960 | 1082 | 276992 | 100,37 | 80,30 |
| g722main | 4503 | 1440960 | 4550 | 1164800 | 101,04 | 80,83 |
| music | 4676 | 1496320 | 4693 | 1201408 | 100,36 | 80,29 |
| radproc | 2852 | 912640 | 2844 | 728064 | 99,72 | 79,78 |
| edge | 5190 | 1660800 | 5245 | 1342720 | 101,06 | 80,85 |
| expand | 4689 | 1500480 | 4810 | 1231360 | 102,58 | 82,06 |
| flatten | 4041 | 1293120 | 4162 | 1065472 | 102,99 | 82,40 |
| smooth | 3745 | 1198400 | 3823 | 978688 | 102,08 | 81,67 |
| cjpeg | 9112 | 2915840 | 9199 | 2354944 | 100,95 | 80,76 |
| go | 46043 | 14733760 | 46339 | 11862784 | 100,64 | 80,51 |
| compress | 4103 | 1312960 | 4115 | 1053440 | 100,29 | 80,23 |
| **averages** | | | | | 101.10% | 80.88% |

Table 7.5: `mach.big` benchmark results

### 7.1.3 Conclusions

From the benchmarks we can observe the following:

- The instruction count increases in the new implementation. This is expected behavior, since now the immediates have to take up space normally occupied by moves.
- The average increase is about 1 to 3% for fairly large architectures. This number is relatively low, since the average share of immediate-moves in the instruction stream is much higher than that. This means that most immediates were encoded in unoccupied move slots. This is also backed by the fact that in fairly large machines, with e.g. 6 buses, the last 2 or 3 buses only achieve a utilization of about 20%.
- The `mach.small` architecture, with only 3 buses, has to take a 6% increase in cycle count. The move bus utilization is much higher or efficient in small architectures. Therefore immediates cannot always be scheduled in unoccupied slots and extra instructions have to be added.
- Since the immediates now are scheduled in the move slots, the dedicated move fields can go away. This results in an average of a 20% shorter instruction word. This can also be observed from table 7.1. Since the average instruction count increase is relatively low, the effective gain of this implementation is the removal of the dedicated move fields.

We can finally conclude that the "resource variant" implementation of the long immediates in the MOVE framework has resulted in an implementation where there is a clean hook for scheduling immediates in the code, without cluttering the code base with extra support for immediates in various parts of the code. The main hook where the implementation comes into play is in `FindImmMoveBus`. The quantitative results indicate that the average increase in cycle count is relatively very low compared to the number of moves that have an immediate in its source field. This increase in cycle count is completely overshadowed by the reduction in code size since we do not need a dedicated move field anymore. The real efficiency gained thus is a 1% increase in execution time against a 20% (average) decrease in code size. Especially in environments where MOVE is deployed, e.g. embedded systems, the emphasis lies on the code size and not so much on the execution time.

## 7.2 Future work

The design space of Long Immediates in MOVE is not completely exhausted yet. Also, other tools, like `explore` can benefit from a long immediate implementation. These issues will be discussed in this section.

### 7.2.1 Exploration

`explore` is a tool that evaluates different machine configurations in order to come up with an optimal (cost/performance wise) MOVE configuration for a certain application. A short introduction into optimization of a move configuration through exploration is given in chapter 2 and especially section 2.1.2.

Since long immediates are part of the real MOVE configuration now, instead of the old implementation of fixed fields alongside the instruction word, exploration should evaluate different long immediate combinations.

Since the design of long immediates in MOVE is so extremely flexible, an implementation of long immediates in `explore` should be designed very carefully. The design space is quite large:

1. Different move slots can be used as immediate bit field
2. Move slots can be concatenated to form larger immediates
3. Different immediate registers can write to different move slots
4. A certain cycle can combine various writes to immediate registers

Because of all these degrees of freedom, an enormous design space can be evaluated. During an implementation, a designer should carefully choose with degrees of freedom are useful for exploration. Quick-and-dirty tests can probably reveal which changes in the machine configuration have a big impact and, as such, will be suitable for exploration.

### 7.2.2 Immediate sharing

Immediate sharing is the concept of writing to an immediate register once, and reading from that immediate register many times. Figure 7.1 tries to visualize this.

Currently, the algorithms are designed in such a way, that an immediate write and read are coupled together. The scheduling algorithm schedules the read (a `i0 -> dst` move), after which a suitable slot, where the immediate will be written to an immediate register, will be found. The check whether an immediate register resource is available is done through the functions`RTab::IsBusy(IReg*)` and `RTab::IsFree(IReg*)`. Currently, these functions will return a notion of “busy” if the immediate register resource is occupied.

Already implemented, however, is the function `RTab::IsBusy(IReg*, int val)`, which will return a notion of “free”, even if the immediate resource is busy, but the `val` value matches the value already in the immediate register. This way, resource checking by means of `RTab::IsBusy(IReg*, int val)` provides a way to share immediates in immediate registers.

The functions `RTab::Claim(IReg*)` and `RTab::Release(IReg*)` both have provisions for immediate sharing. Instead of a boolean value that indicates “busy”, a reference counter is used.

As can be seen, the data structures and functions are already suited for a long immediate sharing implementation. What is needed is adaptation of the main algorithms to take advantage of this.

To get an idea how effective sharing of long immediates is, the following test was conducted: Every time a long immediate is tried for a schedule, a call to `RTab::IsBusy(Ireg*, int val)` is done. A counter was kept how many time this call returned “false”, while the actual value in the immediate register was the same as the immediate value under schedule. This counter effectively counts the number of times that immediate sharing was possible. Another counter tallied the total number of tries for immediate sharing. These two counters give an idea of the frequency of possible immediate sharings. The result for an extensive benchmark consisting of 30 tests resulted in an overall frequency of about 0.5% to 1.0%.

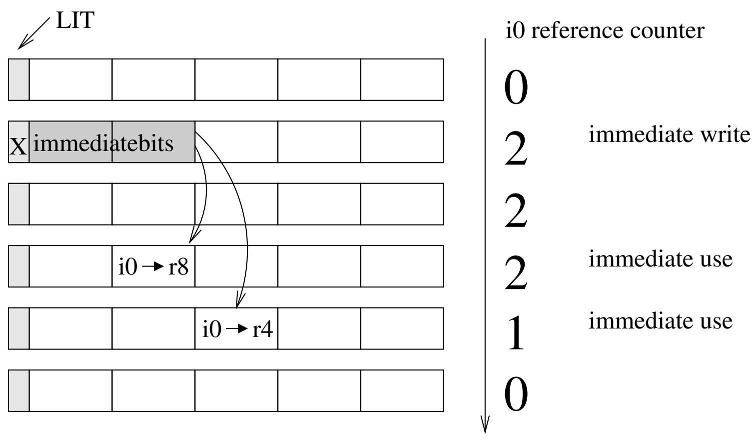


Figure 7.1: Sharing of long immediates

### 7.2.3 Region scheduling of immediates

Region scheduling is the concept of moving moves to predecessor blocks, in order to take advantage of possible empty slots in those predecessors. For more information on region scheduling, also known as interbasicblock scheduling, see [Cor95a] and [Hoo96].

Region scheduling normally applies automatically to moves. Immediate writes are not real moves, however, but nothing more than a resource occupied, and accounted for by the Long Immediate Tag. This makes it impossible for the scheduler to transparently "import" [1] immediate writes in predecessor blocks. Figure 7.2 visualizes the concept of importing long immediates. However, an algorithm analogous to the already existing import routines can be developed for the immediate writes.

The function `ScheduleLImm()` is responsible for finding a suitable cycle. Normally a loop from a certain `max_cycle` down to cycle zero tries to find a suitable cycle. If we reach cycle zero, and no suitable cycle is found, `ScheduleLImm()` returns `false`. Instead of returning false, code analogous to `SchedulOp()` and `SchedulOp2()` can be constructed: If no suitable cycle is found, try to import the immediate write in all predecessors of the current block.

To get an idea how effective importing could be, the following test was conducted: Every time a long immediate is tried for schedule, a counter is incremented every time the main loop in `ScheduleLImm` (see algorithm 7) fails. This means the loop hit the ceiling of a basic block, after which importing to predecessors could be deployed. Another counter keeps the total number of tries for immediate sharing. These two counters give an idea of the frequency of importing possibilities. The result for an extensive benchmark consisting of 30 tests, done with various machine descriptions and various benchmarks, resulted in an overall frequency of 5% to 10%. This means that in 5 to 10 percent of the cases a schedule in that basic block with that specific set of sockets, immediate registers and move buses failed. Most probably another combination of those sockets, immediate registers and move buses could be scheduled correctly, without the need to resort to importing. This observation is backed by the fact that usually in over 75% of the immediate schedules, the write is scheduled in the same cycle as the read, keeping

[1] the concept of placing moves in predecessor blocks

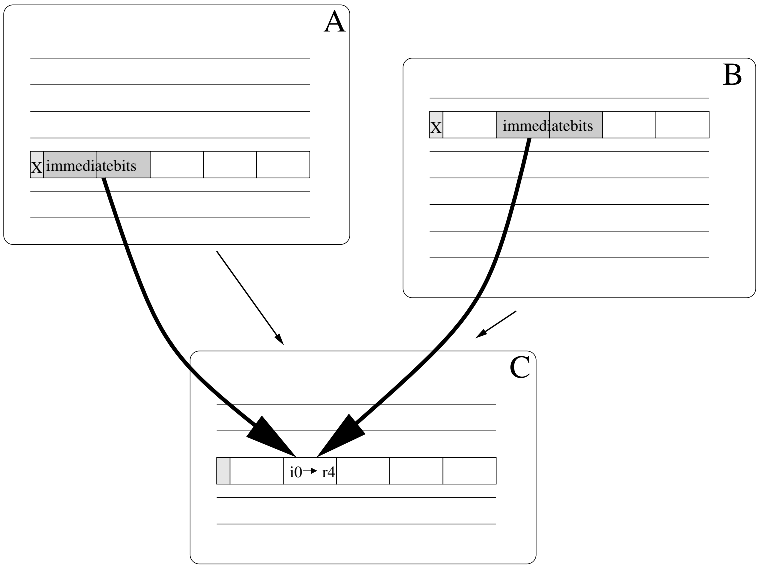


Figure 7.2: Importing of long immediates

the immediate register only busy for that one cycle.

### 7.2.4 Conclusions

Although the immediate implementation could still be extended, tests show that the extra efficiency achieved by implementing “importing” and “sharing” of immediate writes seem low compared to the already achieved benefits obtained by removing the dedicated move slots. The implementation of long-immediate exploration in the `explore` tools however will be a valuable addition to the framework, since this will give the designer the ability to make an architecture description that is as optimal as possible. For instance, an extra immediate register might boost performance by enabling the scheduling of more than one immediate in one cycle, but one has to consider the fact that in hardware this register might be expensive.

# 8 The pseudo-move variant

The long immediate implementation as discussed in the previous chapters (the "resource variant") is not the only implementation done on the MOVE framework. Parallel to this implementation, primarily driven by the PcomP architecture (see chapter 1), TNO-FEL had a need for its own implementation. This section will discuss the motivation behind that implementation, and a brief overview of it. This implementation, dubbed the "pseudo-move" variant will be compared to the "resource variant" implementation, both qualitative, by comparing the algorithms and drawing conclusions, and quantitatively, by benchmarking both implementations.

## 8.1 Implementation

The implementation discussed in this report has an important paradigm: "Try to make the implementation as stand-alone as possible, so existing code won't have to be changed a lot." This decision was made because the current source of the scheduler was considered to be quite difficult to understand in the short amount of time that was available. Also, because of many non-documented inter-dependencies of data structures in the source, at first sight obvious changes to the scheduler might result in bugs in completely different parts of the scheduler. That this assumption didn't hold in the end, can be read in chapter 9.

The implementation of TNO-FEL used another paradigm: "Implement as obvious as possible, in order to get the implementation done quick, even if that means making changes to various fundamental data structures".

Basically, the "pseudo-move" implementation is the following:

1. Before scheduling, transform a move containing an immediate in a separate operation that writes to a virtual register, and a move that reads this virtual register instead of the

immediate directly. A data dependency is then added to link the two operations. Note that while the "immediate read" move still is a real move, the "immediate write" move is a pseudo move, since it doesn't represent a real transport during execution, but merely an encoding of bits.

2. A new operation `IMMEDIATE` is defined, as well as the flags `IMMEDIATE` and `LONG_IMMEDIATE` for the `Node` and `Move` classes.
3. During scheduling, the operation `IMMEDIATE` is handled on the same level as other classes of operations like `COPY`, `OPERATION`, `JUMP` and `CALL`. The moves that make up this operation; at least one immediate write and exactly one immediate read; are all treated as real moves, appearing in move lists and eligible for optimizations like importing.
4. The last step is performed after scheduling and will distribute the bits of the immediate over all "immediate write" moves, so the binary writer knows what bits to write to what fields. Interesting detail is that also the "immediate write" move can store a couple of bits in the unused opcode field of the "immediate register" address.

Figure 8.1 will visualize the transformation of the moves. `i0` here represent any immediate register, since this will change during scheduling anyway. r̃33 here represent any virtual register. Later on, this virtual register will be assigned to a real register by the register allocator.

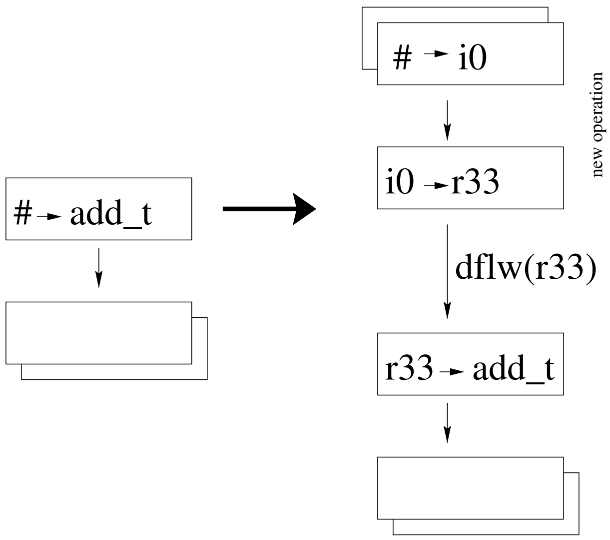


Figure 8.1: Transformation to immediate operation

This is all implemented via a couple of routines in the scheduler source and a new machfile format. The machfile format is altered to incorporate the movebus where the immediate bits of that particular immediate register are read from, and it is augmented with a field that specify how many immediate bits can be stored in the source field of the "immediate use" move. Although that source field contains the encoding for an immediate register, e.g. "i0", this immediate register has no further opcode specifications, so some bits remain available for immediate bits. An example of an old and a new section of `ImmediateUnits` is the following:

Old format:

```
ImmediateUnits
{
  i0  16, signed, i0_r;
}
```

New format:

```
ImmediateUnits
{
  i0  16, signed, {m2,m3}, 3, i0_r;
}
```

As you can see, the new format specifies in what move slot the immediate bits are written (in this case from slots that server movebus `m2` and `m3`), and an integer specifying how many immediate bits can be stored in the opcode field of the source field of the immediate use (in this case 3 bits).

Algorithm 14 will show how a move containing immediate bits is converted into a new operation, the original move and a data dependency between them. The `BuildLongImmediates` is called for every move during the computation of the the data flow information in routine `ComputeDFlowInfo`.

During scheduling, the routines `ScheduleInterBasicBlock` and `ScheduleIntraBasicBlock` are responsible for dispatching to schedule functions based on the operation type. We know have a new operation, the immediate operation. Every operation is scheduled from its "hook", for a normal operation the trigger move, and in this case the "immediate write" move. If the two dispatch routines encounter such a move, they will dispatch to a new routine, `ScheduleLongImmediate`. The `ScheduleLongImmediate` routine and its helper routines are based on the `ScheduleOperation` routines. The analog of the trigger move is the "immediate read" move and the analog of the operand moves are the pseudo moves that represents the "immediate write". Now normal scheduling of the new operation can proceed.

After scheduling, a routine called `AssignLongImmediates` is called. This routine is responsible for assigning parts of the bitfields of an immediate to the various "immediate write" pseudo moves *and* to the empty opcode-field bits of the "immediate register" source field of the "immediate read" move. The "immediate" moves have a special field `imm_val` for this purpose, which is used during binary write when the actual bits are written into the binary stream. Algorithm 15 will give an overview of this algorithm.

## 8.2 Qualitative comparison

The advantages of this implementation over the "resource variant" are:

1. Since the immediate writes are real moves, they can be treated as such by the scheduler.
2. Importing of immediate writes to other basic blocks is possible.

The disadvantages of the "pseudo move" implementation are:

**Algorithm 14** BuildLongImmediates (*Move)

```
// compute the number of slots needed for this immediate
for all move slot do
    if move slot can contain immediate bits then
        increment nr_of_slots_needed
        decrement immediate with width of move slot
    end if
    if immediate width reaches zero then
        break
    end if
end for
// make the new operation
make new r_move and r_insn
set r_move to org_guard:i0->free_virtual_register
set r_move to "IMMEDIATE RESULT" type
add r_move to r_insn
insert r_insn to current block
for all nr_of_slots_needed do
    make new i_move and i_insn
    set i_move to org_guard:imm_val->i0
    set i_move to "LONG IMMEDIATE" type
    add i_move to i_insn
    insert i_insn to current block
end for
// link both operations together
if original move was a "COPY" move then
    bypass now, delete original move completely
else
    clear "IMMEDIATE" flag from original move
    set source field of original move to the free virtual register
    add data flow dependency between r_move and the original move via the virtual register
end if
```

1. This solution adds one more move to the list of move than the “resource variant”. This disadvantage can be canceled out if the move from the original operation is bypassed by the immediate write in the “immediate operation”.
2. The “resource variant” has more flexibility in the writing of immediate registers, since the `mach` file format has been altered a lot to accommodate for a wide range of possible encoding formats, specified by the LIT tag.
3. Sharing of immediate values is not possible in the “pseudo move” variant.

The next subsection will evaluate both implementations, showing whether above mentioned advantages and disadvantages are indeed proven right.

**Algorithm 15** AssignLongImmediates

```
for all procedures in program do
  for all long immediate operations in procedure do
    for all moves in this operation do
      if move is an immediate read move then
        assign bits to ireg opcode field (write value of bits in the imm_val field)
        adjust remaining immediate value by shifting
      end if
      if move is an immediate write move then
        calculate maximum number of bits that this move slot can contain
        assign bits to this move slot (write value of bits in the imm_val field)
        adjust remaining immediate value by shifting
      end if
    end for
  end for
end for
```

## 8.3 Quantitative comparison

The two implementations "resource variant" and "pseudo-move variant" are both benchmarked against the same suite. This suite is a subset of the suite used for the evaluation of the implementation. Only three mach-files were taken into consideration and the set of benchmarks was not as extensive as in the "resource variant" review. Only instruction counts were taken into account, since the codesize reduction due to the dedicated immediate fields is the same for both implementations. Tables 8.1 to 8.3 will give the results between both implementations and a relative figure that indicates which implementation achieves a lower instruction-count increase. Since the two code bases are otherwise very different, it is unwise to try to compare the raw instruction counts. For instance, the compiler used to implement the "resource variant" has much more bug fixes and is developed further than the compiler used to implement the "pseudo-move variant". Other than that, the compiler parameters were kept the same as much as possible. Used options include "interbasicblock scheduling", "early" register allocation, and a machine format with at least 32 general purpose registers and a fully connected transportation network. For the machfiles, see appendix C.

The tables consist of the following columns:

1. The instruction counts of the "resource variant", both without (old) and with (new) long immediate implementation
2. The instruction counts of the "pseudo-move variant", both without (old) and with (new) long immediate implementation
3. Relative increase in instruction count for both the "resource variant" and the "pseudo-move" variant.

| benchmark | old resource instr.count | new resource instr.count | old ps.-move instr.count | new ps.-move instr.count | resource relative instr.count | ps.-move relative instr.count |
|---|---|---|---|---|---|---|
| arfreq | 991 | 1014 | 1166 | 1168 | 102.32 | 100.17 |
| edge | 4338 | 4453 | 3852 | 3971 | 102.65 | 103.09 |
| flatten | 3554 | 3616 | 3695 | 3751 | 101.74 | 101.52 |
| smooth | 3216 | 3273 | 3138 | 3212 | 101.77 | 102.36 |
| cjpeg | 8571 | 8749 | 9192 | 9318 | 102.07 | 101.37 |
| **averages** | | | | | 102.11% | 101.70% |

Table 8.1: `mach.pcomp` comparison

| benchmark | old resource instr.count | new resource instr.count | old ps.-move instr.count | new ps.-move instr.count | resource relative instr.count | ps.-move relative instr.count |
|---|---|---|---|---|---|---|
| arfreq | 993 | 1014 | 1166 | 1166 | 102.11 | 100.00 |
| edge | 4390 | 4432 | 3870 | 3939 | 100.95 | 101.78 |
| flatten | 3593 | 3606 | 3670 | 3695 | 100.36 | 100.68 |
| smooth | 3222 | 3239 | 3111 | 3151 | 100.52 | 101.29 |
| cjpeg | 8590 | 8678 | 9187 | 9271 | 101.04 | 100.91 |
| **averages** | | | | | 101.37% | 100.93% |

Table 8.2: `mach.one` comparison

| benchmark | old resource instr.count | new resource instr.count | old ps.-move instr.count | new ps.-move instr.count | resource relative instr.count | ps.-move relative instr.count |
|---|---|---|---|---|---|---|
| arfreq | 1160 | 1249 | 1313 | 1346 | 107.67 | 102.51 |
| edge | 5677 | 6086 | 5136 | 5384 | 107.20 | 104.82 |
| flatten | 4500 | 4804 | 4589 | 4795 | 106.75 | 104.49 |
| smooth | 4215 | 4492 | 4041 | 4242 | 106.57 | 104.97 |
| cjpeg | 11146 | 111696 | 11924 | 12447 | 104.93 | 104.39 |
| **averages** | | | | | 106.63% | 104.24% |

Table 8.3: `mach.small` comparison

## 8.4 Conclusions

Several conclusions can be drawn from the previous two subsections.

- Both implementation yield about the same increase in cycle count. Although the “pseudo move variant” uses one more move in its conversion, this move can be bypassed in a lot of cases. Also, we already saw in the conclusions for the “resource variant” that for fairly large machines, the bus utilization is low enough to be able to store the immediate write (and even an extra move in the “pseudo move variant”) without having to add a new instruction.

- For small architectures this utilization becomes more of a problem. You will see that the “pseudo move” variant gets a slightly better efficiency in scheduling, since the immediate write is a normal move under scheduling, a task for which many optimizations are written.
- Because the compiler used for the “pseudo-move variant” is not as optimized, more “holes” in the instruction stream will appear, which is beneficial for efficient scheduling of long immediates, which can take advantage of a less dense utilized transportation network

We can now re-evaluate the advantages and disadvantages as presented in subsection 8.2:

- The main advantages of the “resource variant” are: There are relatively less adaptations to the existing framework necessary. Adding a new type of “move” to the scheduler involves adaptations of the code base in several parts of the scheduler. The other advantage is that the design space of choosing which move slots write to which immediate register and when, is much more flexible in the “resource variant”. The advantage of the “resource variant” that it doesn’t need the extra move proved to be invalid for reasons stated in the previous bullet points.
- The main advantages of the “pseudo-move” is that since the immediate write is a real move, scheduling of the immediate write can be more efficient, since much work has already be done on this field.
- It will be shown in the next section, section 7.2, that the advantages of possible importing and sharing have very few effect on the achieved cycle counts. Both the “pseudo move variant”, which was able to import, and the “resource variant”, which was able to share, could benefit from these two optimizations, but both optimizations will probably never be implemented.

# Part IV

# Epilogue

# 9 Conclusions and recommendation

This chapter will present the conclusions on this thesis in section 9.1. Then section 9.2 will present the recommendations for this thesis.

## 9.1 Conclusions

This section will give conclusions on all topics covered in this thesis. First, subsection 9.1.1 will present conclusions on the endianness port. Subsection 9.1.2 will present conclusions on the long immediates implementations. Finally, subsection 9.1.3 will give some general conclusions on the whole graduation process.

### 9.1.1 Endianness

The original problem of the endianness port of the MOVE framework was twofold. On one side it was desired to run the tools on (cheaper) little-endian machines, instead of on big-endian Suns and HPs, as done so far. On the other side a certain realization of the MOVE framework at NEC was to be embedded in a little-endian chip. Therefore is was desired that the MOVE framework was to be made target-endianness aware. Both problems have a lot in common with each other, so it was natural that both problems were solved at the same time.

Both problems were solved for all parts of the MOVE framework: the front-end consisting of the GCC compiler, the GNU assembler, linker and bintools, and the C system libraries, and the back-end, consisting of the `sched` source that is used to build the scheduler, simulator, and various helper tools.

The result is a framework that can be either compiled for little-endian or big-endian target, by setting a Makefile switch at compile time. This framework can then be installed in parallel to a framework of different target-endianness on one machine. The intermediate files are host-endianness independent. One can take any intermediate file, such as objects, binaries and profiling information, and process them further on a machine that has a different host-endianness than the machine where that file was generated.

### 9.1.2 Long Immediates

The original problem of the long immediates was to find a better way to schedule long immediates. The old framework used dedicated immediate fields alongside the instruction word. The goal was to get rid of these dedicated fields and use ordinary move slots for storing immediates.

An implementation, dubbed the "resource variant" was developed. It is called the "resource variant" since the only way to track the immediates in the move slots is by checking various resource tables such as the immediate register's occupation and the contents of a special tag in each instruction word. The method followed is to first completely schedule the use of the immediate (e.g. the `i0 -> r3` move), after which a special routine tries to schedule the immediate definition (e.g. the `# -> i0` move). Also taken into account is the backtracking of scheduling, the so-called "killing" and "unkilling" of nodes.

This resulted in an implementation that adds between 1 and 5 percent to the instruction count. This increase is low compared to the number of moves actually containing an immediate, which can be between 10 and 30 percent. It turns out that most immediate writes are scheduled in otherwise empty move slots. Also, take into consideration that since the dedicated immediate fields are not needed anymore, the total code size (instruction count multiplied by the instruction word length) decreases, up to 20 percent for some configurations.

A comparison with another implementation of Long Immediates in the MOVE framework was conducted. This other implementation, dubbed the "pseudo move variant" uses pseudo-moves to represent the immediate writes like `# -> i0`. This gives this implementation a better flexibility to schedule these immediate writes, but it is more limited in flexible immediate register configurations, a feature of the "resource variant". Both implementation achieve about the same performance in terms of instruction count. Regarding implementation costs, it was expected that the "resource variant" would be easier to implement, since all code was hooked from one point in the scheduler. In the end this expectation did not come true, since several unexpected bugs turned up in various parts of the code. The amendments made to the data structures with the expected, documented behavior in mind, triggered several bugs that expected a different behavior. Closer look at the documented API revealed that the API was indeed ambiguous at some points. Fixing the triggered bugs to make the whole framework consistent again broke the paradigm of a clean code interface. Concluding we can say that one of the planned main advantages of the "resource variant", this clean code interface, did not hold.

### 9.1.3 General

This subsection will give some generic conclusions on the work conducted and the environment in which is was conducted. When I started this task, it was only supposed to be a small part of my complete graduation. Due to various factors, of which the two most important the difficulty of the scheduler source and the environment in which I conducted part of my thesis, my large task list gradually shrank to the two main topics as discussed in this thesis. Apart from these two projects, also various other small projects related to the MOVE project and the PcomP processor at NEC C&CRL were undertaken, but they fall outside the scope of this thesis

The scheduler source is a fairly large code base, contributed to by various developers, who use their own coding-style and even language. Also, the documentation on the source is not complete and in some cases even ambiguous. This results in the fact that a minor adaptation in one part of the source can result in revealing a bug or just misunderstanding of the API in a completely different part of the source. This makes bug hunting very cumbersome. Also, because of the obsolete versions of the GNU front-end, the front-end is very hard to adapt. This all results in the fact that the estimated time on bug-hunting is about 50% of the time spent on the project.

## 9.2 Recommendations

This section will give recommendations on future work on the projects discussed in this thesis. Subsection 9.2.1 will give some recommendations on the endianness work and subsection 9.2.2 will present some recommendations on the long immediates implementation.

### 9.2.1 Endianness

A few observations with respect to the endianness port can be made:

- The front-end tools are quite obsolete. The used version of the assembler, linker and bintools is version 1.38, which was current in 1993. The GNU project has developed much better, and most important, much better portable tools since then. The newest versions use a common library, `libbfd`, that takes care of all binary-format specific tasks. This means that the assembler, linker and bintools itself need very little changing. `libbfd` is a very easily portable library, constructed with the easy port to a new architecture in mind. Also changing the endianness of a target would have been much easier with this library. It is recommended that in the future a newer version of the GNU tools are used, in order to be better prepared for future modifications. The reason that this was not undertaken in this project was that due to time constraints, a quicker, albeit less long-term, solution was chosen.
- All intermediate forms of the front-end are binary. Since the back-end needs to parse this binary back again into an internal format, it might be advisable to look into the possibilities of a textual representation of all intermediate steps. This eliminates a lot of endianness problems, and simplifies the reader of the back-end. Also intermediate steps can be visually inspected much easier this way. Drawbacks might include a larger size and possible parsing complexity, but I do recommend that this option is taken into consideration.

### 9.2.2 Long Immediates

The work done on the long immediates represent a complete implementation for scheduling long immediates in the MOVE framework. However, there is always room for more improvements. Various ideas for future work will be presented, together with a judgment on the feasibility and usefulness of those ideas. These ideas are already explained in detail in section 7.2, together will outlines on how to implement these.

**Exploration** Exploration means trying different machine descriptions until an optimal architecture is found. Optimal is here defined as a balance between costs (die area) and performance (execution time). Long Immediate Encoding could be another parameter that is automatically exploited.. Since the design of long immediates in MOVE is so extremely flexible, an implementation of long immediates in `explore` should be designed very carefully. It is recommended that the `explore` tool is adapted to take Long Immediates into account.

**Immediate sharing** Immediate sharing means re-using the immediate write. Tests have shown that the possibilities for immediate sharing are very limited in normal applications. The framework as implemented is however almost ready to deploy immediate sharing, all data structures and helper functions are in place. However, the complexity of the scheduler makes it unlikely that the scheduler will produce correct code immediately. A decision should be made whether this immediate sharing is desired, based on a review of the performance gain and a study how hard it will be to enable immediate sharing correctly.

**Region scheduling of immediates** Region scheduling, or importing of immediates means moving and copying an immediate write to predecessor blocks if an immediate write cannot be scheduled in its home basic block. The same logic as in the previous paragraph goes here too: A fair part of the code and algorithms needed to implement this in the scheduler is already present. However, tests show that possibilities for importing are relatively low compared to the work that will be needed to achieve a correct implementation. History has learned that the scheduler source is full of undocumented side effects, and care should be taken to not underestimate the time needed to get a bug-free implementation.

# A Endianness related data structures

This appendix contains excerpts of some scheduler data classes, with emphasis on how they deal with endianness.

## A.1 SimMem

```
class SimMem
{
public:
    virtual ~SimMem() { };
    virtual void WriteW(int, s32) = 0;
    virtual void WriteH(int, s16) = 0;
    virtual void WriteB(int, s8 ) = 0;
    virtual void WriteS(int, f32) = 0;
    virtual void WriteD(int, f64) = 0;
    virtual s32 ReadW(int) = 0;
    virtual s16 ReadH(int) = 0;
    virtual s8  ReadB(int) = 0;
    virtual f32 ReadS(int) = 0;
    virtual f64 ReadD(int) = 0;
...
protected:
#if SWAP_ENDIANESS
    s32 S(s32 data) { return SwapEndianess(data); }
    s16 S(s16 data) { return SwapEndianess(data); }
    f32 S(f32 data) { return SwapEndianess(data); }
```

```
    f64 S(f64 data) { return SwapEndianess(data); }
#else
    s32 S(s32 mem_data) { return mem_data; }
    s16 S(s16 mem_data) { return mem_data; }
    f32 S(f32 mem_data) { return mem_data; }
    f64 S(f64 mem_data) { return mem_data; }
#endif
};

// ########################################################################
//  Class SimMem_internal
// ########################################################################
//  Internal memory model used by the simulator.
// ########################################################################
class SimMem_internal : public SimMem
{
public:
    void WriteW(int addr, s32 mem_data)
{ *(s32 *) Phys(addr) = S(mem_data); }
    void WriteH(int addr, s16 mem_data)
{ *(s16 *) Phys(addr) = S(mem_data); }
    void WriteB(int addr, s8 mem_data)
{ *(s8 *) Phys(addr) = mem_data; }
    void WriteS(int addr, f32 mem_data)
{ *(f32 *) Phys(addr) = S(mem_data); }
    void WriteD(int addr, f64 mem_data)
{
    mem_data = S(mem_data);

    ((int *) Phys(addr))[0] = ((int *) &mem_data)[0];
    ((int *) Phys(addr))[1] = ((int *) &mem_data)[1];
}
    s32 ReadW(int addr)
{ return S(*(s32 *) Phys(addr)); }
    s16 ReadH(int addr)
{ return S(*(s16 *) Phys(addr)); }
    s8 ReadB(int addr)
{ return *(s8 *) Phys(addr); }
    f32 ReadS(int addr)
{ return S(*(f32 *) Phys(addr)); }
    f64 ReadD(int addr)
    {
f64 mem_data;

((int *) &mem_data)[0] = ((int *) Phys(addr))[0];
((int *) &mem_data)[1] = ((int *) Phys(addr))[1];

return S(mem_data);
    }
...
}
```

# B Long immediate related data structures

This appendix contains excerpts of some scheduler data classes, with emphasis on the new long immediate format.

**LImmMOp**

```
class LImmMOp : public ListItem {
    friend Mach;

public:
    LImmMOp(const LImmMOp&);
    LImmMOp(unsigned, int, int);
    unsigned Slots()          const { return slots; };
    void SetSlots(unsigned s)       { slots = s; };
    int Bits()                const { return nbits; };
    int Encoding()            const { return enc; };
private:
    const int nbits;
    unsigned slots;
    const int enc;
};
```

**LImmControl**

```
class LImmControl : public ListItem {
    friend int yyparse();
    friend Mach;
```

```
public:
    unsigned  Slots()                     const { return slots; };
    void SetSlots(unsigned s)                   { slots = s; };
    const LImmMOpList& GetMOpList()       const { return microops; };
    void CheckContents();

private:
    LImmControl(LImmMOpList& list);
    LImmMOpList microops;
    unsigned slots;
};
```

**Mach**

```
class Mach {
...
    const LImmControl* Mach::GetDefaultEncoding();
    const LImmControl& GetLImmControl(int index);
    int NumLImmOperations() { return icontrol.Count(); };
...
}
```

**IReg**

```
class IReg: public ListItem, public Mark
{
...
public:
    LImmMOpIter PossibleEncoding(int size);
    LImmMOpIter GetMOp()        { return mops; };
    LImmMOpList mops;
...
}
```

**Insn**

```
class Insn: public ListItem, public FlagSet, public Mark
{
...
public:
    const LImmControl* ImmControlOp() { return immctrl; };
    void SetImmControlOp(const LImmControl* newimm) { immctrl=newimm;};
private:
    const LImmControl* immctrl;
...
}
```

**RTabEntry**

```
class RTabEntry: public Mark
{
public:
  int IsFree(IReg *ireg, int val)
  {
    int idx = ireg->Index();
    return ireg_busy[idx] == 0 || ireg_val[idx] == val;
  }
  int IsBusy(IReg *ireg)
  {
    int idx = ireg->Index();
    return ireg_busy[idx];
  }
  int IsBusy(IReg *ireg, int val)
  {
    return !IsFree(ireg, val);
  }
  void Claim(IReg *ireg, int val)
  {
    int idx = ireg->Index();

    ireg_busy[idx]++;
    ireg_val[idx] = val;
  }
  void Release(IReg *ireg)
  {
    ireg_busy[ireg->Index()]--;
  }
  long IRegVal(IReg* ireg) const
  {
      return ireg_val[ireg->Index()];
  }
private:
  char ireg_busy[MAX_N_IMMEDIATE_REGISTERS];
  long ireg_val[MAX_N_IMMEDIATE_REGISTERS];

private:
  const LImmControl* immctrl;
public:
  const LImmControl* ImmControlOp() { return immctrl; };
  void SetImmControlOp(const LImmControl* newimm) { immctrl=newimm; };
```

# C Machine description files

In this appendixes, the relevant parts of the used mach files are presented. All mach files have a fully-connected transport network. Presented for each machine will be the used mach file used for the “resource variant”, then the one used for the “pseudo-move variant” and lastly the mach file used for no immediates at all.

## C.1 mach.small

### “resource variant”

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
}

LongImmediate
{
Registers:
    i0 32, signed, ir_0;
Control:
```

```
    {};
    i0 32 : { 2 };
}
```

### “pseudo-move variant”

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
}

ImmediateUnits
{
    i1 32, signed, {m3}, 0, ir_1, ;
}
```

### old long immediates support

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
}

ImmediateUnits
{
    i1 32, signed, ir_1;
}
```

## C.2 mach.pcomp

### “resource variant”

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
```

```
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
    m4 64, 8, signed;
    m5 64, 8, signed;
    m6 64, 8, signed;
}

LongImmediate
{
Registers:
    i0 20, signed, ir_0;
    i1 20, signed, ir_1;
    i2 32, signed, ir_2;
Control:
    {};
    i0 20 : { 4 };
    i1 20 : { 5 };
    i0 20 : { 4 }, i1 20: { 5 }, i2 32: {4,5};
}
```

### “pseudo-move variant”

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
    m4 64, 8, signed;
    m5 64, 8, signed;
    m6 64, 8, signed;
}

ImmediateUnits
{
    i1 32, signed, {m5},    0, ir_1;
    i2 20, signed, {m6},    0, ir_2;
    i3 20, signed, {m5,m6}, 0, ir_3;
}
```

### old long immediates support

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4
```

```
MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
    m4 64, 8, signed;
    m5 64, 8, signed;
    m6 64, 8, signed;
}

ImmediateUnits
{
    i1 32, signed, ir_1;
    i2 20, signed, ir_2;
    i3 20, signed, ir_3;
}
```

## C.3 mach.one

### "resource variant"

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
    m4 64, 8, signed;
    m5 64, 8, signed;
    m6 64, 8, signed;
}

LongImmediate
{
Registers:
    i0 32, signed, ir_0;
Control:
    {};
    i0 32 : { 5 };
}
```

### "pseudo-move variant"

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4
```

```
MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
    m4 64, 8, signed;
    m5 64, 8, signed;
    m6 64, 8, signed;
}

ImmediateUnits
{
    i0 32, signed, {m6}, 0,  ir_0;
}
```

### old long immediate support

```
#define N_IREGS         32
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
    m4 64, 8, signed;
    m5 64, 8, signed;
    m6 64, 8, signed;
}

ImmediateUnits
{
    i0 32, signed, ir_0;
}
```

## C.4 mach.big

### "resource variant"

```
#define N_IREGS         64
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
```

```
    m3 64, 8, signed;
    m4 64, 8, signed;
    m5 64, 8, signed;
    m6 64, 8, signed;
    m7 64, 8, signed;
    m8 64, 8, signed;
}

LongImmediate
{
Registers:
    i1 32, signed, ir_1;
    i2 32, signed, ir_2;
Control:
    {};
    i1 32: {6};
    i2 32: {7};
    i1 32: {6}, i2 32: {7};
}
```

### old long immediate support

```
#define N_IREGS         64
#define N_FREGS         48
#define N_BREGS         4

MoveBusses
{
    m1 64, 8, signed;
    m2 64, 8, signed;
    m3 64, 8, signed;
    m4 64, 8, signed;
    m5 64, 8, signed;
    m6 64, 8, signed;
    m7 64, 8, signed;
    m8 64, 8, signed;
}

ImmediateUnits
{
    i1 32, signed, ir_1;
    i2 32, signed, ir_2;
}
```

# Bibliography

[CH96] Henk Corporaal and Jan Hoogerbrugge. Cosynthesis with the MOVE framework. In *CESA 96*, 1996.

[Cil00] Andrea Cilio. Documentation of the current scheduling algorithm. Technical report, Delft University of Technology, 2000.

[CM91] Henk Corporaal and Hans Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[Cor95a] Hendrik Corporaal. *Transport Triggered Architectures – Design and Evaluation*. PhD thesis, Technical University of Delft, August 1995.

[Cor95b] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

[HC94] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of transport triggered architectures. In *MICRO-27*, Santa Clara, December 1994.

[Hoo96] Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Technical University of Delft, February 1996.

[Int97] Intel Corporation. *Intel Architecture Software Developer's Manual*, 1997. Volume 2, Instruction set reference.

[Int00] Intel Corporation. *The IA-64 Architecture Software Developer's Manual*, July 2000. Volume 3, rev 1.1, Instruction set reference.

[Jan97] Ivo Janssen. Software tools umove processor. Technical Report FEL-97-S277, October 1997.

[Joh96] Johan Janssen, Andrea Cilio. The move software framework. Technical report, Delft University of Technology, 1996.

[LC95] Reinoud Lamberts and Henk Corporaal. Options for long immediates in the move framework. Technical report, Delft University of Technology, 1995.

[PH97] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Second edition, 1997. Web extension I: Survey of RISC architectures.

[Sta94] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA, September 1994. version 2.6.

[Zuk98] Steven Zuker. Endianness in solaris. February 1998.