Optimization of a compiler for a custom microprocessor

Degree project in compiler technology

2nd level, 30 HEC

Valence, France 2010/2011

Xavier LEGROS

04/03/2011

legros@kth.se

KTH Information and Communication Technology

Supervisor: Philippe GROSSI, Design Authority, THALES Avionics

Examiner: Ass. Prof. Christian SCHULTE, Department of Software and Computer Systems, School of Information and Communication Technology, KTH

Release agreement

Autorisation de diffusion

	First Name	Family Name	Position	Date	Signature
Written by	Xavier	LEGROS	Intern	03/03/2011	
Approved by	Philippe	GROSSI	Supervisor	03/03/2011	
	Lionel	FRAICHARD	Head of the Department	03/03/2011	
Authorized by	THALES Gervais VAZE Officier de Sécurité Suppléant Division Avionique		Security Officer	03/03/2011	

Abstract

New systems have higher and higher requirements in terms of reliability, safety, power consumption and performance. To meet those requirements, a custom processor can be a solution. That is why Thales is developing a processor that includes many features that make its architecture different from common architectures of general-purpose processors. To become widely usable, a fully optimizing compiler is the cornerstone that leads to its success.

While most of the existing compilers manage to deal with code optimization for general purpose processors, their efficiency can considerably decrease when optimizing code for a custom processor whose architecture differs from general purpose processors.

A new compiler had to be developed to translate C source code into highly optimized assembly code that would be able to deal with this processor's abilities, such as DSP-like instructions, in the best way possible, while introducing key features necessary to meet the requirements of Thales' line of products.

Code optimizations have been introduced and enhanced, using pattern detection, control flow analysis or peephole optimization. At the same time, new key features have been added to the compiler, such as a compilation report, built-in functions support, and also fixed-point representation support. Many of these resources were lacking in existing compilers, which brings a noticeable added value to this compiler.

As the processor is still under development, tests were carried out using a VHDL simulation model of the processor designed specifically to test the entire platform. The set of tests focused both on general purpose applications, and also on data processing algorithms. The optimization of this compiler led to an 18.3% shorter execution time for the set of tests, while reducing the total program size by 13.1% at least. Moreover, important features that existing compilers are lacking, have been added to the compiler. It helped improving the user-friendliness of the pair compiler/processor and it has proven to be of great interest for programmers.

Acknowledgments

I want to express my gratitude to all of those who made this thesis possible.

First, I would like to thank my supervisor at Thales Avionics, **Philippe Grossi**. Throughout my work, he has been helping me with critical reviews, providing detailed background knowledge, expertise, guidance and encouragement. His personal involvement in the project made it possible to write a successful thesis and I am immensely grateful for this.

I would like to show my heartiest gratitude to **Christian Schulte** for being my compiler teacher and examiner, for his directions and guidance throughout all the formalities of report writing, but also for his advices that helped me to achieve this thesis.

I'm also thankful for the assistance the **Micro-electronic Department** gave me at all levels of the project, for sharing their knowledge, for providing me with priceless feedback and for patiently answering all my questions.

Lastly, I express my gratitude to my family and friends, without whose patience, constructive critiques and support it would be far more difficult to complete this project.

List of Abbreviations

ALU:	**A**rithmetic and **L**ogic **U**nit
ASIC:	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
CFG:	**C**ontrol **F**low **G**raph
CPU:	**C**entral **P**rocessing **U**nit
DSP:	**D**igital **S**ignal **P**rocessor
FPU:	**F**loating **P**oint **U**nit
HLL:	**H**igh **L**evel **L**anguage
ISA:	**I**nstruction **S**et **A**rchitecture
IR:	**I**ntermediate **R**epresentation
LSB:	**L**east **S**ignificant **B**it
MMU:	**M**emory **M**anagement **U**nit
MSB:	**M**ost **S**ignificant **B**it
OS:	**O**perating **S**ystem
RISC:	**R**educed **I**nstruction **S**et **C**omputer
VHDL:	**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage

Table of Content

1 Objectives ... 3

2 Introduction ... 5

2.1 Preface ... 5

2.2 What is a compiler? ... 5

2.3 The binary generation tool-chain ... 5

2.3.1 Preprocessor ... 6

2.3.2 Assembler ... 6

2.3.3 Linker ... 7

2.4 Benefits of a compiler ... 7

2.4.1 Portability ... 7

2.4.2 Maintenance ... 7

2.4.3 Reusability ... 7

2.4.4 Quality of the code ... 7

2.5 Internals of a compiler ... 8

2.5.1 The frontend ... 8

2.5.2 The core ... 9

2.5.3 The backend ... 11

2.6 Retargetable compilers ... 11

2.7 Problems ... 12

2.8 Thesis Overview ... 13

3 The Compiler ... 15

3.1 Test methodology ... 15

3.1.1 Description of the optimization levels ... 15

3.1.2 Software tests ... 16

3.1.3 Hardware tests ... 16

3.2 Test process ... 17

4 Compiler optimizations ... 19

4.1 Mulneg pattern detection ... 19

4.1.1 Principle ... 19

4.1.2 Results ... 20

4.2 Mulacc pattern detection ... 20

4.2.1 Principle ... 20

4.2.2 Constraints and Limitations ... 22

4.2.3 Results ... 22

4.3 Advanced operator strength reduction ... 24

4.3.1 Principle ... 24

4.3.2 Results ... 26

5 Hardware and compiler optimizations ... 27

5.1 Support of new addressing mode for arithmetic and bitwise instructions ... 27

5.1.1 Problem ... 27

5.1.2 Principle ... 28

5.1.3 Results ... 29

5.2 Support of the hardware out-of-order algorithm for the FPU ... 30

5.2.1 Principle ... 30

5.2.2 Results 30

6 New compiler features 33

6.1 Compilation report 33

6.2 Built-in function support 33

6.2.1 Principle 33

6.2.2 Consequences on existing tests 35

6.3 Porting the C standard library 37

6.4 Fixed-point representation support 38

6.4.1 Context 38

6.4.2 Compiler modifications 38

6.5 Debug support 40

6.5.1 What is a debugger 40

6.5.2 The need for debugging support 40

7 Overall results and discussion 43

7.1 Execution time 43

7.2 Program size 44

8 Conclusion 47

9 Future work 49

9.1 Instruction scheduling 49

9.2 Add new functions to standard libraries 49

9.3 Create a complete source debugging tool-chain 49

10 References 51

11 List of figures 53

12 List of Tables 53

13 Appendices 55

1 Objectives

This master thesis is part of a larger project aimed at developing a custom multi-purpose microprocessor to integrate it into a large variety of systems. To make this custom microprocessor easily programmable and use its full potential, the development of a powerful compiler is a key factor. A stable C compiler fully optimized for a specific target, would offer the developers a good level of services to become as user-friendly as a widespread compiler for common microprocessors.

This processor is a RISC (Reduced Instruction Set Computer) processor oriented for digital signal-processing and data-stream applications, also offering a complete instruction-set for multi-purpose applications. Its architecture, which is between the one of a general-purpose processor and the one of a Digital Signal Processor (DSP), presents several challenges when optimizing a compiler for such an architecture.

While most of the existing compilers manage to deal with code optimization for general-purpose processors, their efficiency can considerably decrease when optimizing code for a custom processor whose architecture differs from general-purpose processors.

The main objective of this thesis is to optimize a C compiler to increase performance while adding new features to it, such as development of standard libraries or built-in function support. In order to improve it further, the compiler was tested extensively to find and eliminate existing errors.

2 Introduction

2.1 Preface

As this thesis deals with compiler's optimization, it is crucial to have general knowledge about compilers, their internals and how to optimize such a system. Compiler technology is a complex subject, and requires a lot of understanding of different areas of computer science. All the information is gathered in various books, papers and discussions.

This chapter gives a state of the art of what a compiler does and how it does it, and reveals the problematic of optimizing a compiler. More information can be found in [1], [2], [3] and [4].

2.2 What is a compiler?

Programming languages are notations used by programmers to describe and express computations that can be performed by a computer. However, before a program can be interpreted by a machine, the source code must first be translated into a more basic representation. The software systems that translate source code into another computer language are called compilers.

Most compilers translate High Level languages (HLL) into assembly language for a target processor but translating from an HLL to another HLL (Java to C for example) is also possible.

A compiler is basically a translator. Yet, this task can be very challenging and raises a lot of issues.

Also, translating a source code into another language does not give an executable binary program that can run on a microprocessor. That's why a compiler is a part of bigger tool-chain.

2.3 The binary generation tool-chain

A compiler itself is rarely used alone. It is a part of a much bigger tool-chain that transforms a HLL source code into an executable program that is adapted to the target machine. From now on, the term source language refers to the C programming language. The organization and the different parts of it are shown in **Figure 1**

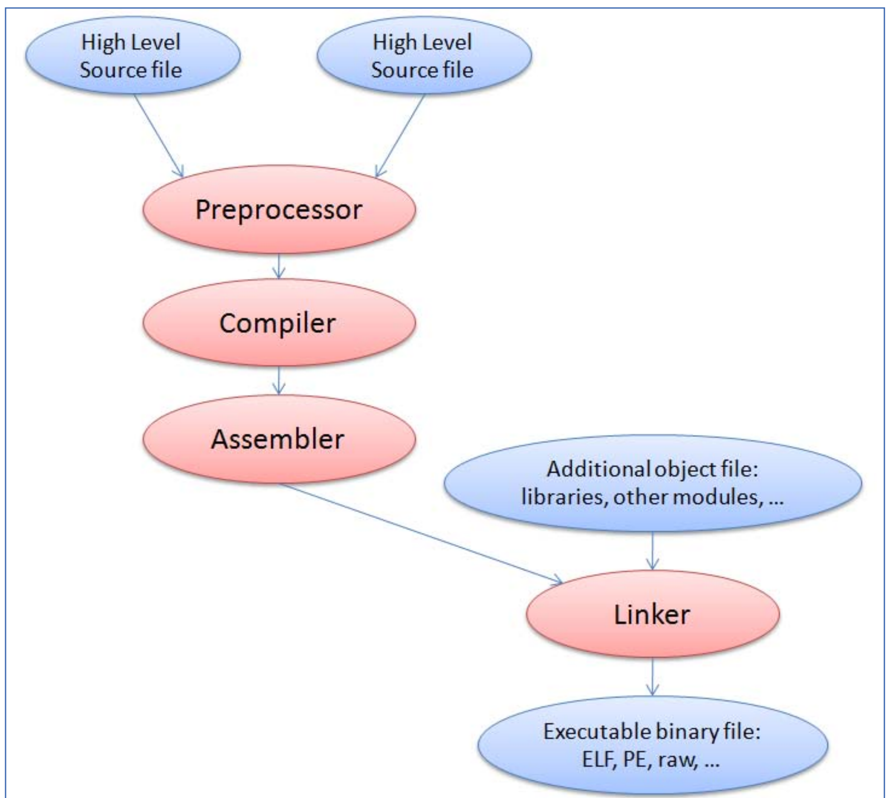

Figure 1 : The binary generation tool-chain

2.3.1 Preprocessor

The preprocessor is in charge of expanding macros (`#define`), including files (`#include`), following conditional statements (`#if`, `#ifdef`, …), and removing comments, which are useless for the compiler.

It basically deals with directives such as:

```
#include <stdio.h>
#define DEBUG 1
#ifdef
...
```

The most common preprocessor for the C language on UNIX and Linux machines is CPP (C PreProcessor) [5].

2.3.2 Assembler

The assembler works mainly as an encoder: it reads assembly language instructions, and encodes them in binary, following a scheme described in the Instruction Set Architecture (ISA) of the target processor.

Ultimately, the assembler produces an object file, which is a binary file, but not yet executable in most cases. This object file, among program and data, also holds relocation information, used in the linking stage.

Object files come in a wide variety of format, and they can differ depending on the architecture and the Operating System (OS) used. The most well-known formats are PE under Windows and ELF under UNIX systems.

2.3.3 Linker

The linker takes several object files and combines them together to produce the final executable binary. In the process, it also has to compute relocation information that will be used by the Memory Management Unit (MMU) of the processor when loading the program in memory.

2.4 Benefits of a compiler

Typically, each target machine has its own assembly code. Even if writing programs in HLLs is now common, one may wonder why is it good to add a new layer (the HLL) to raise the level of abstraction, making the process more complicated so that a new tool (a compiler) would be needed.

First, programming directly in assembly requires knowledge about the processor. In order to write efficient assembly programs, it is of great importance to be very familiar with the target processor's internals, which can be really difficult. This becomes more and more problematic today, when modern processors are more and more complex, providing powerful facilities directly in their hardware. Without good knowledge about modern processors in general, it is rather problematic to write good code for that given processor.

Compilers were not only designed to avoid coding directly in assembly language. In fact, much more important reasons lead to the use of compilers. These reasons are:

2.4.1 Portability

Assembly code is specific to a target machine whereas HLL source code is not. In order to get a target specific assembly code, it is necessary to pass an HLL source code though a compiler. This idea can be well illustrated by the following slogan: "Code once, compile anywhere!"

2.4.2 Maintenance

As software becomes more and more complex, a good maintenance level needs to be achieved. HLLs significantly facilitate this task as they are much more "user-friendly" than assembly code. Moreover, assembly language is the lowest level, which makes it hard to debug errors.

2.4.3 Reusability

HLLs ease code reusability between projects: They allow the programmer to manipulate objects, structures, data types, that have been defined with a level of abstraction high enough to program generic modules.

2.4.4 Quality of the code

Depending on the needs, compilers can optimize code according to different criteria, such as speed, code size, power consumption, etc.

More importantly, a compiler does not get unfocused halfway through the optimization process: whatever the size of the code is, the compiler will do its best to optimize it, which would be more time and energy consuming for a person.

However, even though a good compiler can produce code of good quality, the output of the compiler might not be good enough in time critical applications. Hence, time critical sections need to be coded manually in assembly.

2.5 Internals of a compiler

A compiler is generally organized in different parts that are depicted in **Figure 2.**

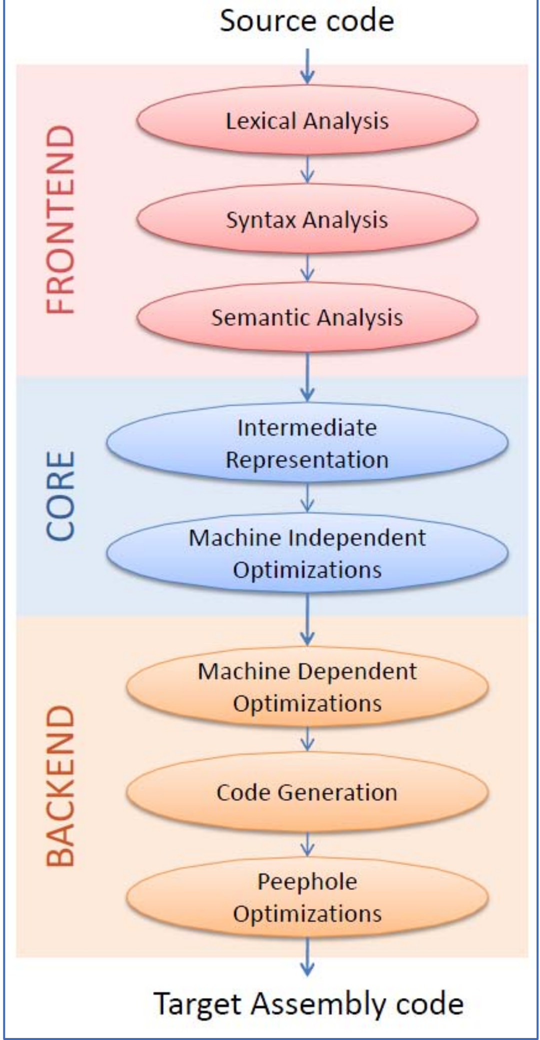

Figure 2 : Internals of a compiler

2.5.1 The frontend

The frontend analyses the input source code and deduces information from it for the subsequent parts of the compiler to work on. It checks whether the program is correctly written in terms of programming language's syntax and semantics. Here legal and illegal programs are recognized. Errors are reported, if any.

- Lexical analysis: Read the input source code and group the characters into meaningful tokens representing variables, language keywords, etc.
- Syntax analysis: This pass uses the tokens produced by the lexical analysis and creates a tree-like representation that depicts the grammatical structure of the source code.
- Semantic analysis: This pass uses the syntax tree to check the source code for consistency with the HLL definition. Type checking is also performed during this pass.

2.5.2 The core

The core generates the Intermediate Representation (IR) for the following backend. Many optimization efforts are focused on this part.

- Intermediate representation: An explicit low level intermediate representation of the source code is done. This is the representation that the backend will use to produce the machine dependent code. Representations can be graph-based or can follow a 3-addresses code form.
- Machine independent optimizations: The IR is optimized. The optimization often aims at increasing code speed, but other goals can also be achieved, depending on the desired objectives: short code size, reduced power consumption, etc.

  Common optimizations include:

  - Common Sub-expression Elimination:

    ```
    void f(int x, int y) {
     q(x * y, x * y);
    }
    ```

    becomes

    ```
    void f(int x, int y) {
     int tmp;
     tmp = x * y;
     q(tmp, tmp);
    }
    ```

    The expression `x*y` is used twice so the optimization consists in computing it only once and put the result into a temporary variable.

  - Constant propagation:

    ```
    int f() {
     int x;
     x = 1;
     return x;
    }
    ```

    becomes

    ```
    int f() {
     return 1;
    }
    ```

    Since the variable `x` is not modified between its assignation and the return statement, the optimization consists in deleting this unnecessary variable and returning the constant 1.

  - Dead code elimination:

    ```
    int f(int y) {
     int x;
     x = 3;
     y = 2 + 4*x;
     return x;
     x = 2;
     y = 3;
    }
    ```

    becomes

```
int f(int y) {
 int x;
 x = 3;
 y = 2 + 4*x;
 return x;
}
```

Since the code following the return statement is not reachable, this code is deleted.

– Loop unrolling:

```
for (i=0 ; i<4 ; i++) {
 f(i);
}
```

becomes

```
f(0);
f(1);
f(2);
f(3);
```

The boundaries of the `for` loop are known at compile-time, so the optimization consists in getting rid of the loop code and copy the code inside the loop as many times as it is executed. This optimization results in a shorter execution time but increases the program size.

– Function inlining:

```
int f(int y) {
 return ((5*x) + 7);
}

int main() {
 int var = 4;
 int res;

 res = f(var);
 return res;
}
```

becomes

```
int f(int y) {
 return ((5*x) + 7);
}

int main() {
 int var = 4;
 int res;

 res = ((5*var) + 7);
 return res;
}
```

The optimization consists in performing the code of the function without executing this function, which is way faster since no jump to the function is performed, and hence, no register saving is needed.

– Loop hoisting:

```
for (i=0 ; i<10 ; i++) {
 b = 7;
 a = 4*i + b;
}
```

becomes

```
b = 7;
for (i=0 ; i<10 ; i++) {
 a = 4*i + b;
}
```

Since the statement `b = 7` is independent of the loop, it can be moved out of the loop.

- Register allocation: this is the process of assigning a large number of program variables onto a small number of CPU registers. Register allocation can happen over a basic block (local register allocation), over a whole function (global register allocation), or in-between functions as a calling convention (interprocedural register allocation). It avoids the loading and unloading of operands at each instruction to make full use of available registers. As a consequence, memory utilization becomes lower, resulting in higher speed and lower power consumption. A basic block is a straight-line sequence of code with only one entry point and only one exit.

Many of those optimizations are based on data-flow analysis. Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. This way, it is possible to make powerful analysis and optimizations. A CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

2.5.3 The backend

The backend is responsible for translating the IR into the target assembly code. The target instructions are chosen from one or several nodes of the IR.

- Machine-dependent optimizations: This phase is mostly about combining several IR nodes into one single machine-dependent instruction. The backend also performs instruction scheduling: Reorder instructions to make better use of the different pipelines and processing elements a CPU may offer. (Avoid pipeline stalls, etc.)
- Peephole optimizations [6]: Final obvious local optimizations, like redundant `moves`/`load` removal or unnecessary `nop` removal.

2.6 Retargetable compilers

Modern compilers are designed to be retargetable, which means they can be relatively easily enhanced to support new target machines.

Since a great amount of passes are machine-independent, a careful design allows the compiler to keep all its code in a common, target-independent core, while providing functionalities to later plug a machine-dependent code generator into it. That way, development time for a new processor is shortened, and more importantly, the target-independent code is factorized, so that it only needs to be developed and tested once.

The core holds all the target and HLL independent code, and provides interfaces to plug frontends and backends into it as depicted in **Figure 3**.

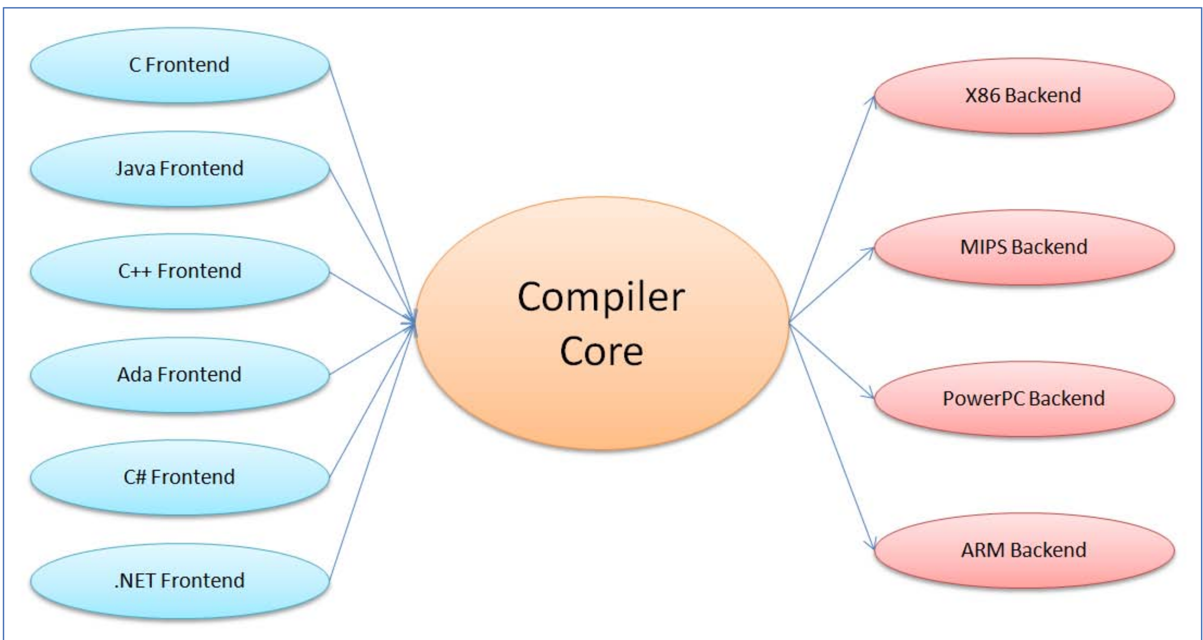

Figure 3 : A modular compiler

Most compilers only provide retargetability at the backend level, and are focused on only one HLL. Some bigger compilers are retargeted at both frontend and backend level. GNU Compiler Collection [7], also known as GCC and Low Level Virtual Machine (LLVM) [8] are such compilers.

2.7 Problems

These retargetable architectures are really great for compiler development. However, since the core is completely factorized in a machine-independent fashion, it is necessary to completely rely on its interfaces to write a backend for a new machine. Problems arise when the target machine is very different from "traditional" architectures, and thus breaks assumptions the core developers made.

As a result, even parts of the compiler that are said to be machine independent can make assumptions on the target machine. The vast majority of machines have no problems with that, but when trying to write a backend for an unusual machine architecture, especially custom designed CPUs, this can lead to a great deal of problems.

Those problems can be difficult to tackle when optimizing a compiler. That is why it might be necessary to modify the core of the compiler that needs to be optimized, even though, according to the modularity principle, only the backend should be modified.

Also, even on retargetable compilers, extensive knowledge on how the core uses the backend is necessary to program the exact desired behavior. Otherwise, small changes can have huge consequences on the output.

2.8 Thesis Overview

Section 3 of this report describes the compiler on which the thesis was based and the test methodology used to ensure its correct behavior. A brief description of the different optimization levels of the compiler is also provided in this section

Section 4 describes the compiler optimizations that have been performed during the thesis. The principle of each optimization and its results are also provided.

Section 5 describes optimizations of the global system composed of the hardware and the compiler. It highlights the importance of hardware/software co-design and its benefits. In this section, specific optimizations involving hardware and software modifications will be described.

Section 6 describes the new features that have been added to the compiler, their impact and benefits for the end user.

Finally, this thesis ends with **Section 7**, **Section 8** and **Section 9** which summarize results obtained from the previous sections and contain some proposals for future work and extensions of this master thesis project.

3 The Compiler

For this project, the decision had been made to use a retargetable C compiler other than GCC. This choice is a compromise between the complexity and the maturity of the compiler.

The chosen compiler implements every common optimization described previously and respects all the notions that have been introduced in the previous sections.

3.1 Test methodology

To ensure the proper functioning of the compiler and evaluate the progress made by the different optimizations, it is necessary to set up a large variety of tests. These tests have to check that the compiler can issue valid assembly code, whatever the source program is, and make sure that the compilation did not alter the meaning of the source program.

Moreover, these tests are also a way to evaluate the performance of the compiler concerning several aspects such as the execution time or the program size.

For this project, two types of tests have been implemented. Each type is based on a set of source programs written in C that the compiler must compile.

3.1.1 Description of the optimization levels

In the following sections, results will be analyzed depending on the optimization levels that have been used to compile the different tests. The compiler provides 5 levels of optimization that are set using a compilation option that goes from `-O0` to `-O4`. Those optimization levels correspond to certain optimization flags that enable or not different kind of optimizations.

In optimization level `-O0`, no optimization is performed. The compiler simply does register allocation and translate the source program into assembly code.

In optimization level `-O1`, the following optimizations are triggered:

- Register allocation
- Removal of unreachable code and unused labels (based on control-flow graph analysis).
- Common sub-expression elimination, copy propagation and constant propagation only within basic blocks.
- Loop optimization (loop-invariant code motion).
- Unused object elimination
- DSP-like instruction optimization such as `mulacc`/`mulneg` detection
- Strength reduction

Optimization level `-O2` performs the same optimizations than `-O1` but the common sub-expression elimination, copy propagation and constant propagation throughout the whole program instead of being limited to basic blocks.

Optimization level `-O3` performs the same optimizations than `-O2` plus additional optimizations:

- Pointer analysis, alias analysis
- More aggressive loop optimization (loop unrolling, induction variable elimination)
- Function inlining
- Inter-procedural analysis and cross-module optimizations

Optimization level `-O4` performs the same optimizations than `-O3` but also activates the target instruction scheduler. As there is no instruction scheduler developed yet, the resulting programs for optimization level 3 and 4 are the same.

3.1.2 Software tests

The aim of those tests is to check whether or not the compiler compiles without making mistakes, and if the generated assembly program is correct at the syntactic level. No checks are made on the proper functioning of the generated program, but it can very quickly test the compiler on a substantial set of source programs.

To respect the principle of independency between the developer and the tester, those software test programs come from test suite of other compilers such as GCC or LLVM which have not been developed by Thales. Moreover, using complex test suite such has the one of GCC (which has been under development for many years) is a way to use the maturity of GCC, and hence, the maturity of its test suite, and evaluate with accuracy the correctness of a compiler under development.

3.1.3 Hardware tests

The purpose of these tests is to check the proper functioning of the program after compilation and evaluate the performance of the compiler. Only this kind of test makes sure that the compilation has not altered the meaning of the source program nor the expected results.

Part of these tests focuses on very specific checks (on floating-point numbers, arithmetic operations, or loop constructions, for example). They are called basic tests. Those are made to test as extensively as possible every class of C constructions to ensure that they are correctly treated by the compiler. Basic tests include as many combinations of variables and operation as possible using different types (`int`, `short`, `long`, `float`, etc.), different storage-classes (`static`, `extern`, local, global, etc.) to be as exhaustive as possible.

Those tests are also used to evaluate the impact of an optimization on one specific type of C construction. For example, the following code is extracted from one existing test case that evaluates performance of different types of operation on floating-point numbers. The idea is to make sure that results are correct according to the standard (IEEE 754 [9] is the standard for floating-point number representation) and also evaluate the performance of such a program.

```c
float x;
double d1 = 2.5e45;
double d2 = -2.5e45;
double D[5] = {1.0e51, 0.0, -8.3e6, 3.9e4, -
5.1236e39};

int i = 0x50201;
long int l = 0x80030010005;

int main () {
 double y;

 float a = 2.3e24;
 float b = -8456.216e1;

 x = a + b;         // Basic test
 y = D[4] + d1;     // Test on array
 y = 5.0e2 * d2;    // Test with immediate value
 x = (float) d2;    // Conversion test
 y = (double) a;    // Conversion test
 i = (int) D[2];    // Conversion test
 a = 2.0e19;
 l = (long int) a;  // Test on overflow
 f = -0.0;
 f = 0.0 / f;       // Test for NaN values
 f = (-0.0) * f;    // Test for NaN values
}
```

The second category of tests focuses on realistic tests that mix different C constructions. Those tests are useful when evaluating the performance of the compiler on meaningful and realistic tests. Indeed, even though basic tests are useful when evaluating the performance of one particular class of C constructions, the main focus of those tests is to ensure the accuracy of the resulting program. They are not necessarily relevant when evaluating the performance of a "real" program that actually has a purpose other than checking accuracy of constructions as tricky as possible.

That is why the second category of tests focuses on meaningful or/and heavy algorithms (such as matrix inversion algorithm, FIR-filter algorithm, basic cryptography algorithm, etc.), in the field of data processing, which is the main field of application of the processor. Moreover, basic tests do not involve a lot of variables, temporaries or typical C constructions. Hence, it is hardly possible to test the limits of the compiler or the limits of the target machine without this second category of test. Compiling one of the basic tests does not necessarily require many registers of the microprocessor for example, which means that the mechanism where every register is in use cannot be tested using basic tests.

A description of all the hardware tests can be found in **Appendix 1**.

3.2 Test process

The verification process consists in comparing reference results with results given by the system composed by the compiler and the microprocessor of Thales. Hardware analysis is conducted through VHDL simulations (called virtual verifications) using the C executable generated by the compiler. Each test case is compiled both through Thales' compiler and

GCC. The output is processed by a VHDL simulation model of the processor and results are compared with results of GCC executed on a host processor (considered as the golden reference, which means that those results will be considered as the results to achieve). The comparison is printed in a result file that can be seen in **Appendix 2**.

The test case is passed if no difference between the golden reference and the simulation results is found.

Figure 4 illustrates the different steps of a hardware test:

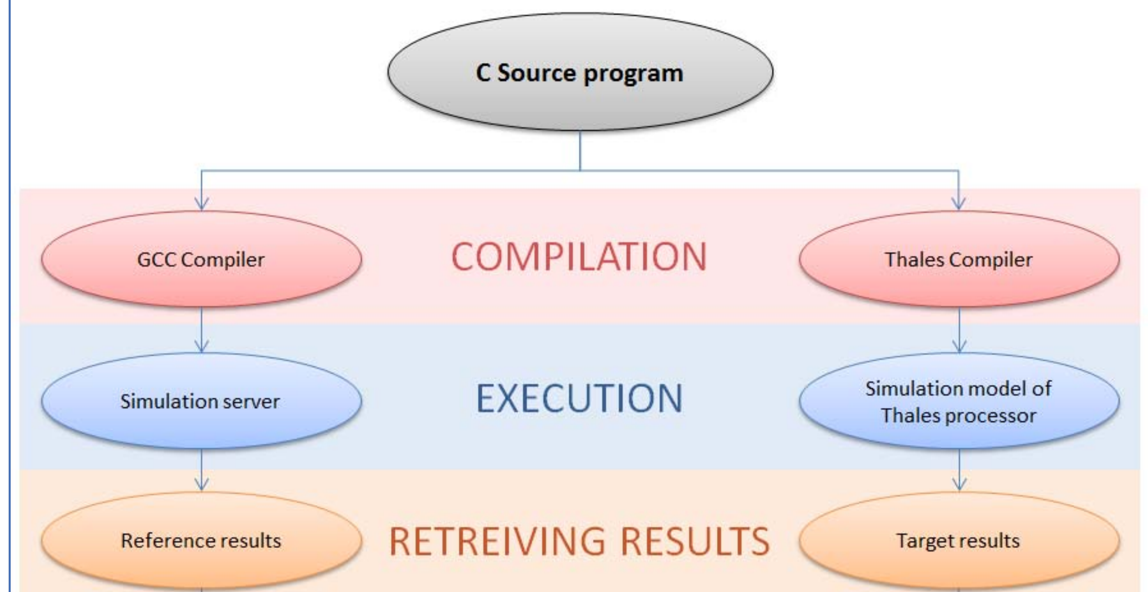

Figure 4 : Overview of the different steps of a hardware test

These tests can be automatically performed with various optimizations. Several scripts are used to synthesize all information of the result files into a table summarizing useful information. An example of a test summary can be seen in **Appendix 3** and **Appendix 4**.

Performing all tests takes a long time (more than an hour), but it is an effective way to check the proper functioning of the compiler, and ensure that no regression has been introduced after a change or the addition of a feature in the compiler.

GCC is used as a reference compiler. Indeed, even though it is not free of bugs, this compiler is the most extensively used in the world. It has been developed for many years and is known as the most stable and complete compiler. Being able to develop a compiler that provides the same results as GCC is ambitious considering its complexity.

4 Compiler optimizations

This section introduces the major optimizations that have been performed on the compiler to improve the resulting assembly code. Many other small optimizations have been also performed. However, they will not be described since they consist in a lot of scattered modifications all over the code of the compiler.

4.1 Mulneg pattern detection

4.1.1 Principle

The `mulneg` instruction is a DSP-like instruction provided by the microprocessor, which allow making a multiplication and a negation in one single instruction. Each of the following pieces of C code can be transformed into one single `mulneg`:

```
a = (-b) * c;
```

```
a = b * (-c);
```

```
a = -(b * c);
```

```
a = b * c;
a = -a;
```

```
b = -b;
a = b * c;
```

This can be done using pattern detection in the backend. When a multiply followed or preceded by a subtraction is detected, it is a potential candidate for `mulneg` optimization. However, something special needs to be taken care of. The last example illustrates a case that might not be transformed into a `mulneg`. If `b` is used after this statement, it cannot be optimized into a `mulneg`. The following example shows this special case where it cannot be optimized:

```
b = 5;
b = -b;            // Now, value of b is -5
a = b * c;
d = 2 * b;         // Value of d is -10
```

If this example is optimized using `mulneg`, the result would be wrong:

```
b = 5;
a = mulneg(b,c);
d = 2 * b;         // Value of d is 10 -> FALSE
```

Hence, the IR needs to be scanned to check the use of the variable holding the negation result, until this variable is overwritten. If it is not used in between, the optimization can be performed. If this variable is used in between, the optimization cannot be performed. This check is performed using control-flow graph analysis.

4.1.2 Results

Table 1 and **Table 2** show the result of this optimization on tests that present possible `mulneg` patterns, since they are the only one with which improvements can be seen. As described in **Section 3.1.1**, DSP-like optimizations are only activated from optimization level `O1`, which is why the following tables do not show results for the optimization level `O0`.

Table 1 : Results of the mulneg optimization in terms of execution time

Optimization level	Without `mulneg` optimization	With `mulneg` optimization	Speed-up
O1	146 204 clock cycles	138 382 clock cycles	5,4%
O2	116 963 clock cycles	111 701 clock cycles	4,5%
O3	108 254 clock cycles	106 534 clock cycles	1,6%
O4	108 254 clock cycles	106 534 clock cycles	1,6%

Results are slightly better for the execution time than the program size. This is explained by the fact that a `mulneg` optimization performed in a loop only reduces the code size by one, but the execution time is reduced by the number of time the loop is executed.

Table 2 : Results of the mulneg optimization in terms of program size

Optimization level	Without `mulneg` optimization	With `mulneg` optimization	Evolution
O1	2 143 words	2 119 words	-1,1%
O2	2 187 words	2 152 words	-1,6%
O3	2 823 words	2 791 words	-1,1%
O3	2 823 words	2 791 words	-1,1%

4.2 Mulacc pattern detection

4.2.1 Principle

The `mulacc` instruction (respectively `mulsub`) is also a DSP-like instruction provided by the processor, which does a multiplication and accumulation in the same instruction (`a = a + (b * c)`) (respectively `a = a - (b * c)`). A multiply-and-accumulate operation is very common in digital-signal-processing. Optimizing it can increase the performance drastically.

This optimization is also a matter of pattern detection in the first place, but the conditions to be able to perform this optimization are totally different.

First, such an optimization is only valuable while being in a loop. If not in a loop, doing a simple multiplication followed by an addition is strictly equivalent to doing a multiply and

accumulate and store the result into the destination variable (both in terms of execution time and program size). This is why this optimization should only be performed within a loop. This case will be illustrated by a proper and simple example. Considering the following piece of code:

```
i = 0;
while (i<100) {
 dest = dest + (a * b);
 i++;
}
```

This piece of code would be translated into a corresponding non-optimized representation in the IR that would look like this:

```
i = 0;
while (i<100) {
 temp = (a * b);          // 4 clock cycles
 dest = dest + temp;      // 1 clock cycle
 i++;                     // 1 clock cycle
}
```

A quick estimation of the execution time shows that the code inside the loop would take at least 6 clock cycles to execute (4 clock cycles to execute a multiplication (3 for the multiplication in itself and 1 to store the result into variable `temp`), 1 for an addition, and 1 for an increment). This piece of code is inside a loop which is executed 100 times. Hence, the time to compute this inside code is equal to 600 clock cycles. To make it simpler, the code generated to jump inside the loop and the code to test variable `i` has been ignored since it does not matter in this case.

Optimizing this code using a `mulacc` will save a lot of time and the optimized code would be the following:

```
i = 0;
accumulator = 0;    // Initialized accumulator to 0
while (i<100) {
 mulacc(a * b);     // 3 clock cycles
 i++;               // 1 clock cycle
}
dest = accumulator;       // Store the result into
                          // destination variable
```

It should be noted that in this piece of code, `accumulator` is not a variable but the special register used by the MULAC unit to compute a multiplication. The resulting execution time to execute the code inside the loop is 4 clock cycles. Indeed, the time to perform a multiplication and an accumulation is still 3 clock cycles, but we do not have to store the result into a temporary variable. The resulting time to compute this inside code is equal to 400 clock cycles. 2 extra cycles should be added to be accurate (1 to initialize the accumulator to 0 and 1 to store the result into the destination variable).

Performing this optimization resulted in decreasing the execution time by 198 cycles, or 33%! It should be specified that the result of this optimization depends on how many times the loop is executed. The more times the loop is executed, the more powerful the `mulacc` optimization is.

The principle of this `mulacc` optimization is exactly the same for the `mulsub` instruction.

4.2.2 Constraints and Limitations

This last example points out the constraints that go with a `mulacc` optimization.

The first thing that can be noticed is the need to detect loops, and the ability to hoist the initialization of the accumulator and the storage of the result out of the loop. This requires a loop analysis based on a control-flow graph representation.

Also, like the `mulneg` optimization, the variable `temp` should only be used for this multiply-and-accumulate operation, and not elsewhere. Otherwise, it is not possible to perform the optimization.

The last thing to take care of is to allocate an available accumulator throughout the whole loop so that it cannot be accessed or modified by another multiply operation that could be inside the loop. If there is no available accumulator anymore, the optimization cannot be performed. For information, 8 accumulators are available in the MULAC unit, which means that 8 overlapping `mulacc` optimizations can be performed. It is more than enough for a large majority of applications.

4.2.3 Results

`Mulacc` optimization is focused on one particular type of C construction and performance will be improved in particular cases described earlier in this section. The hardware test `algo/matrix.c` is well suited to evaluate the results of a `mulacc` optimization. This test performs multiplication of several matrices (integer and floating-point matrices), which uses such a construct.

The formula to multiply two matrices A and B into a resulting matrix R is the following:

$$[R]_{i,j} = [A]_{i,k} \times [B]_{k,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$

for each coefficient (i, j) of the matrix. This can be translated into the following C code where the three matrices$[A]$, $[B]$ and $[R]$ are of size five:

```
for (i=0; i < 5; i++) {
  for (j=0; j < 5; j++) {
    for (k=0; k < 5 ; k++) {
      R[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Matrix multiplication is very common in data and signal processing. Moreover, the above code shows that the multiply-and-accumulate operation is in this case nested inside 3 loops. The efficiency of the `mulacc` optimization will be even higher.

The following table shows the result of the optimization applied to the previous code section.

Table 3 : Results of the mulacc optimization on matrices multiplication of dimension 5

Optimization level	Without `mulacc` optimization	With `mulacc` optimization	Speed-up
O1	9208 clock cycles	7239 clock cycles	21.4%
O2	5741 clock cycles	3716 clock cycles	35.3%
O3	5618 clock cycles	3593 clock cycles	36.0%
O4	5618 clock cycles	3593 clock cycles	36.0%

Note that results for optimization level O0 are not provided because optimization level O0 disables every possible optimization. Hence, `mulacc` optimization is only available in O1, O2, O3 and O4.

This optimization, which is not usually provided by general purpose compilers, is very efficient. Also, as mentioned before, the more often the loop is executed, the larger the performance imporvement. The following table shows the result of the same optimization, on the same piece of code but with matrices of dimension 32. Speed-up due to the `mulacc` optimization is much higher.

Table 4 : Results of the mulacc optimization on matrices multiplication of dimension 32

Optimization level	Without `mulacc` optimization	With `mulacc` optimization	Speed-up
O1	1 991 012 clock cycles	1 438 052 clock cycles	27.8%
O2	1 306 409 clock cycles	753 449 clock cycles	42.3%
O3	1 273 642 clock cycles	720 682 clock cycles	43.4%
O4	1 273 642 clock cycles	720 682 clock cycles	43.4%

Data processing, such as digital image processing, often works with very large matrices. For example, 1280x1024 pixels is a very common picture format. Yet, it has not been possible to build a test using such large matrices. The amount of data grows very fast with the dimension of the matrix. When the above tests were done, the processor only provided small memory capabilities. Hence, realistic tests using common picture format were not possible.

However, tests that have been performed validate the optimization and are still an accurate benchmark for performance.

4.3 Advanced operator strength reduction

4.3.1 Principle

As mentioned before, the core of the compiler does many optimizations including strength reduction, that is to say replacing expensive operations by equivalent but less expensive operations (i.e. quicker operations) [10], [11] and [12]. The most common strength reduction is replacing a multiplication where one of the operands is an integer constant power of 2, by a shift to the left of the binary representation of the other operand. The number of shift is equal to $\log_2$(constant). For example, this piece of code:

```
b = a * 2;
d = 8 * c;
```

can be optimized using strength reduction. The resulting code would be:

```
b = a << 1;
d = c << 3;
```

As a matter of fact, those instructions are strictly equivalent. However, a multiplication takes at least 3 clock cycles, while a shift only needs 1 clock cycle to be executed. The optimized code is 66% faster and needs only 2 instructions while the non-optimized code needs 4 assembly instructions. The following is an example of the non-optimized assembly code:

```
mul a,2
mov $ACC,b
mul 8,c
mov $ACC,d
```

while the optimized assembly code would look like this:

```
lshift a,1,b
lshift c,3,d
```

For negative integers, the sign extension needs to be taken care of. This is why an arithmetic shift is performed for negative integer, while it is a logical shift for positive integers.

This basic optimization is already performed by the core of the compiler, but advanced strength reduction can still be performed to raise the optimization level for the target.

For example, the core of the compiler does not perform strength reduction on unsigned division by a power of 2. Indeed, a division by a power of 2 is equivalent to a right shift of the binary representation of the second operand. Another common strength reduction is not performed by the core of the compiler: unsigned modulo by x can be replaced by a logical `AND` with x-1 when x is a power of 2. The modulo operation finds the remainder of an integer division of one number by another.

The following example shows some instructions that can be optimized:

```
b = a / 2;
d = c / 8;
f = e mod 16;
```

can be optimized by:

```
b = a >> 1;
d = c >> 3;
f = e and 15;
```

The resulting code is much faster! Indeed, an integer division or a modulo requires as much time to execute as the length of its operands. That is to say that a division of 2 `int` will take 32 clock cycles, 16 clock cycles for a `short`, and 8 for a `char`. Hence the optimized code is 96% faster for 32-bits integer, and requires three times less instructions compared to the non-optimized version.

To reduce even more the load of the MULAC unit, strength reduction on multiply instructions can still be improved, since strength reduction can be extended to multiplication by x where x is not only a constant power of 2, but also a constant power of 2±1 or a constant power of 2±2. Indeed, a multiplication by 5 is equivalent to a multiplication by 4 followed by and addition (i.e. `a*5 = a*4 + a`). Here is an example of such an optimization:

```
b = a * 9;
d = 15 * c;
```

can be optimized in:

```
temp = a * 8;
b = temp + a;
temp2 = c * 16;
d = temp2 - c;
```

And the final result of this optimization is:

```
temp = a << 3;
b = temp + a;
temp2 = c << 4;
d = temp2 - c;
```

Here, the speedup is 33% and the code-size remains unchanged since a multiplication is done using two instructions (one to compute the multiplication, and one to store the result in the destination variable). Also, this code does not require the MULAC unit, but only the ALU. Hence, this optimization makes more `mulacc` optimizations possible as the load of the MULAC unit is reduced.

When optimizing multiplication by a constant power of 2±2, the execution time remains the same and the program size is slightly higher as it can be seen on the following example:

```
b = a * 10;
d = 14 * c;
```

would be optimized in:

```
temp = a << 8;
temp = temp + a;
b = temp + a;
temp2 = c << 16;
temp2 = temp2 - c;
d = temp2 - c;
```

However, this optimization will still be performed as it decreases the load of the MULAC unit, which makes more `mulacc` optimization possible as more accumulators can be available.

4.3.2 Results

Table 5 shows the result of this optimization in terms of execution time. We can see that the results are better in lower level of optimization. This is due to the fact that the core of the compiler already performs some strength reduction which depends on the optimization level. The higher the optimization is, the more aggressive the strength reduction of the core will be.

Those results show that backend strength reduction and core strength reduction are complementary. Using both allows for better execution time.

Table 5 : Results of the advanced strength reduction in terms of execution time

Optimization level	Without advanced strength reduction	With advanced strength reduction	Speed-up
O0	1 022 630 clock cycles	892 486 clock cycles	12.7%
O1	823 773 clock cycles	767 854 clock cycles	6.8%
O2	644 871 clock cycles	632 126 clock cycles	2.0%
O3	632 232 clock cycles	621 417 clock cycles	1.7%
O4	632 232 clock cycles	621 417 clock cycles	1.7%

However, **Table 6** shows that the program size is slightly higher in optimization level `O3` and `O4`. This overhead is not due to the strength reduction itself, but to additional instructions that need to be inserted to load and store the results. However, this overhead is very low and can be disregarded.

Table 6 : Results of the advanced strength reduction in terms of program size

Optimization level	Without advanced strength reduction	With advanced strength reduction	Evolution
O0	12 828 words	12 793 words	-0.3%
O1	13 496 words	13 480 words	-0.1%
O2	13 050 words	13 052 words	+0.0%
O3	14 430 words	14 458 words	+0.2%
O4	14 430 words	14 458 words	+0.2%

5 Hardware and compiler optimizations

As this part of the project is still under development, dialog between the developer team (both hardware and software developers) has been really important. Thanks to this, it has been possible to reach a global level of performance that would probably not have been possible otherwise.

Nowadays, dialog between microprocessor developers and compiler developers is almost inexistent. Because of this, several microprocessor functionalities are not handled properly by compilers and vice versa. Sharing experience between each other makes it possible to improve the global performance of such strongly bonded systems.

The fact that both the microprocessor and the compiler are under development at the same time makes this dialog possible and very important. While developing the compiler, the lack of a particular functionality can be pointed out, and modifications can be done on the microprocessor to tackle this issue.

The fact that the processor is a synthesizable microprocessor makes it easy to modify and improve, which would be impossible for an ASIC.

The creation of a new addressing mode for arithmetic and bitwise instructions illustrates very well this possibility.

5.1 Support of new addressing mode for arithmetic and bitwise instructions

5.1.1 Problem

The memory architecture of the processor is different from a classic load/store architecture where every operand needs to be loaded into a register before the computation. Indeed, it allows to access operands directly from memory, and the result can directly be stored in memory as well. Having a register-only addressing mode can be seen as not as useful as for the traditional processors. That is why the original instruction set of the processor did not provide arithmetic instruction using only registers as operands and destination, i.e.:

```
add $R1,$R2,$R3          # $R3 = $R2 + $R1
```

Nevertheless, this limitation revealed itself quite problematic for the compiler when emitting code for tightly coupled instructions such as:

```
x = a + b;
z = y - x;
```

With the original addressing mode, the code above could have been translated as followed:

```
add d(a),d(b),d(x)
add d(y),d(x),d(z)
```

which means that every operand are accessed via the data memory, and that the results are stored in data memory as well. The process of storing the result of an operation in data memory is called a write-back. This write-back takes more than 1 clock cycle, which

is problematic when the result of an instruction is directly used in the next instruction. In this case, the processor will stall until the write-back is completed and resume execution afterwards, which takes at least 3 clock cycles. When instructions are interdependent, the processor spends more time waiting for the completion of the write-back than computing results. This is why a new addressing mode has been created for arithmetic and bitwise instructions to answer this specific need.

5.1.2 Principle

As mentioned before, a register-only addressing mode has been created. It uses 2 registers as operands, and also a register as destination, like the instructions of a classic load/store processor.

To support this new addressing mode, the main modification of the compiler concerned the register allocation that needed to be adjusted to allow register-only instructions.

However, it was not the only required modification since this addressing mode presents one more specificity.

The way of accessing right and left operands is not symmetric on Thales' processor. This means that for some instructions, the amount of possibilities offered to access the right operand differs from the one to access the left operand. This development choice has been made in order to limit the processor's hardware complexity, and decrease its area and power consumption, which are both key factors in embedded system.

This particular specificity has an impact on the register-only addressing mode. Indeed, the first register of a register-only instruction needs to be the last register used as destination of a previous instruction. Otherwise, the processor will need to stall the execution in order to refresh its value. For the sake of performance, it is important that the compiler does the refreshing of the last value explicitly. Refreshing the value of the last destination register takes only 1 clock cycle. If not done explicitly, the processor will detect it and stall for at least 3 clock cycles (because of a pipeline flush). The following example of strength reduction highlights this issue.

When compiling the following C code:

```
b = a * 7;
```

the compiler will perform strength reduction and transform it into the following code:

```
temp = a << 3;
b = temp - a;
```

which can be translated into the following assembly code:

```
# assuming that $R1 holds a, $R2 holds temp and
# $R3 holds b
rshift 3,$R1,$R2
sub $R1,$R2,$R3     # Performs $R3=$R2-$R1
```

When the processor executes this piece of assembly code, it will stall before executing the subtraction. Indeed, the most recently used register before the subtraction is $R2

whereas the first operand of the `sub` instruction is `$R1`. Refreshing the value contained in `$R1` is needed. Doing it explicitly will save 2 clock cycles:

```
# $CON is a special constant register in which
# every bit is set to 1
rshift 3,$R1,$R2
and $CON,$R1,$R1    # Only takes 1 clock cycle
sub $R1,$R2,$R3
```

This technique is used for instructions in which operands cannot be swapped. If the instruction would have been an addition instead of a subtraction, swapping `$R1` and `$R2` would have been possible as a + b = b + a. In those cases, operand swapping is performed, which does not introduce an overhead.

5.1.3 Results

Table 7 and Table 8 show the performance induced by this new addressing mode.

Table 7 : Results of the register-only addressing mode in terms of execution time

Optimization level	Without register-only addressing mode	With register-only addressing mode	Speed-up
O0	894 271 clock cycles	880 385 clock cycles	1,6%
O1	769 338 clock cycles	760 002 clock cycles	1,2%
O2	629 848 clock cycles	624 403 clock cycles	0,9%
O3	616 702 clock cycles	611 851 clock cycles	0,8%
O4	616 702 clock cycles	611 851 clock cycles	0,8%

Table 8 : Results of the register-only addressing mode in terms of program size

Optimization level	Without register-only addressing mode	With register-only addressing mode	Evolution
O0	12 718 words	12 690 words	-0.2%
O1	13 412 words	13 416 words	+0.0%
O2	12 999 words	12 974 words	-0.2%
O3	14 369 words	14 361 words	-0.1%
O4	14 369 words	14 361 words	-0.1%

Previous tables show that the results of this optimization are rather limited. Results might have been better if the register allocation would have been done differently. Indeed, this new addressing mode points out a limitation that might be the cause of some issues in the future. For now, register allocation is done in the core of the compiler and there is no relation between the way how two adjacent IR nodes are treated. Register allocation is done by calling some backend support functions that evaluate the cost of allocating a variable to a certain register. But this call is done independently for each IR node even

though the most efficient register allocation can only be reached by analyzing dependencies between instructions and by allocating registers according to each instruction capabilities.

In this case, the compiler might not use this new addressing mode in the best way possible because once the backend emits the code, it might be forced to reload operands, or pass by some intermediary instruction as a workaround. This is one of the problems that arise when trying to apply techniques for general purpose processors, where the addressing mode of each instruction is very similar (mostly register access), to this specific processor.

5.2 Support of the hardware out-of-order algorithm for the FPU

5.2.1 Principle

The out-of-order algorithm, also known as the Tomasulo algorithm [13], is an algorithm that improves performance and utilization ratio of a computation unit by reordering instructions. In other words, it allows sequential instructions that would normally be stalled due to certain dependencies to execute non-sequentially (out-of-order execution).

In early versions of the processor, the FPU was devoid of such an algorithm. Thus, it was not possible to use the same register twice in the same floating-point instruction. For example, the following instruction was not possible because the floating-point register `$FP1` is both used as operand and destination:

```
fadd $FP1,$FP2,$FP1        #FP1 = FP1+FP2
```

As a consequence, the register allocation and the backend were developed to handle such situations. In this case, the code produced by the compiler induced a large overhead, both in terms of execution time and program size, because it was necessary to use more temporary registers.

This particular situation did not meet the project's requirements so it has been decided to enhance the FPU so that it would be possible to deal with this issue in hardware.

When the enhanced version of the FPU was released, the compiler had to be modified to get rid of the unnecessary intermediary code.

The modifications' main impact was on register allocation. However, several modifications needed to be made in the entire backend.

5.2.2 Results

As explained before, this optimization specifically focuses on floating-point instructions so improvements can only be seen on tests involving floating-point operations. The following tables show results for tests that involve floating-point instructions only.

Table 9 (respectively **Table 10**) shows the evolution of the execution time (respectively program size).

Table 9 : Results of support of hardware out-of-order algorithm in terms of execution time

Optimization level	Without out-of-order	With out-of-order	Speed-up
O0	925 295 clock cycles	861 084 clock cycles	6,9%
O1	758 227 clock cycles	740 058 clock cycles	2,4%
O2	683 345 clock cycles	606 308 clock cycles	11,3%
O3	674 236 clock cycles	597 408 clock cycles	11,4%
O4	674 236 clock cycles	597 408 clock cycles	11,4%

Table 10 : Results of support of hardware out-of-order algorithm in terms of program size

Optimization level	Without out-of-order	With out-of-order	Evolution
O0	7 696 words	7 613 words	-1.1%
O1	8 217 words	8 118 words	-1.2%
O2	7 960 words	7 874 words	-1.1%
O3	9 199 words	9 082 words	-1.3%
O4	9 199 words	9 082 words	-1.3%

These tables show that the main impact of this optimization is on execution time, even though the program size decreased noticeably. This can be explained by the very nature of this optimization. This enhancement deals with floating-point instructions which are much slower than integer instruction. For example, a floating-point addition needs 10 clock cycles to execute, while the same operation involving integers can be computed in 1 clock cycle. When removing one single floating-point instruction, the execution time is at least reduced by 10 (and much more if this instruction is located in a loop), whereas the program size is only reduced by one.

6 New compiler features

From the start, the compiler was considered functional, but some important features were missing for it to meet the company's requirements in term of usability, performance, and transparency.

This section introduces new features that have been implemented to meet those requirements.

6.1 Compilation report

The compilation report has been developed to provide a good level of information about the system, and what kinds of operations have been performed on the source code. It logs significant information about the compilation process.

Moreover, it gives information to developers on how did the compilation go, what optimizations have been performed and hints that can trigger new optimizations are also suggested. Notes are sorted into a set of categories to be quickly and easily understandable. Available categories are:

- Messages
- Warnings
- Errors
- Hints: Suggested hints that could trigger a new optimization by doing some minor changes in the source code
- Instruction optimizations
- DSP optimizations: This category gather all notifications about detected and optimized DSP-like instructions such as `mulacc` or `mulneg`
- Peephole optimizations

This feedback from the compiler is very important and helpful for a developer when optimizing a piece of software. This compilation report is created for each compiled file. An example of a classic compilation report can be seen in **Appendix 5**.

6.2 Built-in function support

6.2.1 Principle

All "classic" target machine instruction are handled by the compiler. However, if the target machine provides specific instructions that differ from the "classic" instruction-set of a RISC microprocessor, they might not be handled properly. Those instructions are generally input/output and system instructions. In this case, what could correspond to one single target machine instruction might be transformed into several nodes in the IR, which makes it almost impossible to translate it back to the corresponding target instruction while emitting code in the backend. In some cases, it cannot even be handled at all and generate a compilation error. That is why it is needed to find a way to tell the compiler about those instructions.

One solution to achieve this is to write such instructions directly in assembly and include it into the HLL program. This method requires the programmer to know the

corresponding assembly instruction and its syntax, which is what a compiler tries to avoid in the first place. Another drawback with this technique is that no compiler optimization can be performed with this code. Indeed, assembly code is treated by the compiler as a black box, and hence skipped during optimization.

A better way to treat special target instructions is to implement built-in function support in the compiler. A built-in function (generally called built-in) is a function available for use in a given language whose implementation is handled specially by the compiler. Typically, it substitutes a sequence of automatically-generated instructions for the original function call, similar to an inline function. Unlike an inline function though, the compiler has an intimate knowledge of the built-in function and can therefore integrate it better and optimize it for the situation.

After analyzing in details the compiler's internal mechanisms, it has been decided to implement built-in functions using function attributes (because this feature was already supported by the compiler). Function attributes are additional information that is provided to the compiler that takes specific decisions accordingly. A new function attribute has been created: the attribute `__builtin_`.

Even though function attributes were a mechanism originally provided by the compiler, the handling of the `__builtin_` attribute differs from the handling of usual attributes. A significant amount of time has been dedicated to understand how a built-in function should be treated in each step of the compilation (mostly core and backend), and how this treatment differs from other function handling. Declaring a function as a built-in is a way to tell the compiler that this function is not a real function, but a specific instruction of the target, that should be handled in the backend (also known as intrinsic function).

Knowing this, the first decision that could be made would be to bypass the core and all its optimizations so that the built-in would only be handled in the backend. This decision would not be the smartest one. Indeed, doing this is strictly equivalent to the use of inline assembly code, which is what built-in functions try to avoid.

However, if a built-in function is in fact a single target instruction, one may wonder what can be optimized in a built-in function. The answer is: a built-in function in itself cannot be optimized, as it is a single target instruction, but the way its parameters are accessed can be.

The core of the compiler spends most of its time doing optimization such as constant propagation, and more importantly, register allocation. When a built-in function is declared, the operands of the target machine instructions are perceived as the arguments of the corresponding built-in function. For example, the target instruction `out` is a system instruction used to output data on the processor's interface. To be used in a C program, it has been declared as a built-in function.

To output variable x (located in data memory) at channel 1 of the interface, the following assembly instruction is used:

```
out d(x),c(1)
```

The prototype of the corresponding built-in function is the following:

```
__builtin_ extern void out(int channel, int src);
```

This function can be used in a C program. The corresponding C code of the previous assembly instruction would be:

```
out(1,x);
```

Using the built-in function instead of the assembly code is more efficient because register allocation can still be performed through the built-in function.

Also, the built-in function is much more easy to use. For example, the assembly code to output a global variable is different from the code to output a local variable. When using the built-in, the programmer does not have to worry about it anymore: this difference in the resulting assembly code will be handled by the compiler itself.

6.2.2 Consequences on existing tests

The built-in `out` was one of the most important for the sake of the existing tests. Indeed, the only way to make sure that a computation went correctly was to display the result of this computation using the instruction `out`. Before the built-in was implemented, the instruction `out` was used via inline assembly code. This was a very inconvenient limitation which is illustrated by the following example.

Assuming that the addition instruction needs to be tested, the following code should be written:

```
c = a + b;                    // Perform the addition
inline("out d(_c),c(1)");        // output the result
```

Using inline code is a huge limitation in this case. The only way a programmer can know what to write as an inline code is to use global variables. This way, the programmer knows that the way to access global variable `c` is by writing `d(_c)` as a global variable is directly accessed via data memory with the label "`_nameofthevariable`". On the opposite, the position of a local variable and the way to access it is unknown by the programmer since local variables are stored on the stack. This means that in this example, `c` should be declared as a global variable in order to be able to use it. This limitation was a real problem when testing because only global variables were used to display results. The first consequence was that tests were not as broad as they should have been, because possibilities of testing every different addressing mode were limited by the compiler itself. This was very critical. The second consequence concerned optimizations. Indeed, the use of a global variable is problematic when optimizing as it can be accessed by any function of the program at any time, which is why they are usually a barrier when optimizing. Moreover, those tests are not only used to evaluate performance, but also to check the correctness of the compiler, which means the instruction `out` is used heavily.

Now that this built-in is available, every inlined `out` has been replaced by the corresponding built-in, and corresponding global variables has been changed to local variables. The performance of the tests was then much better. The following is a comparison of the global execution time between tests using inlined `out` and tests using built-in `out`.

Table 11 : Results of the use of built-in out with local variables on the total execution time

Optimization level	Using inlined out with global variables	Using built-in out with local variables	Speed-up
O0	892 486 clock cycles	891 980 clock cycles	0.06%
O1	767 854 clock cycles	742 196 clock cycles	3.3%
O2	632 126 clock cycles	600 571 clock cycles	5.0%
O3	621 417 clock cycles	590 420 clock cycles	5.0%
O4	621 417 clock cycles	590 420 clock cycles	5.0%

The simple fact of using the built-in `out` and local variables is 5% faster. Even though it is not a real optimization of the compiler in term of code generation, this new service provided by the compiler takes down some constraints which allow the programmer to write faster code.

This new feature also impact the resulting program size as it is shown in the following table.

Table 12 : Results of the use of built-in out with local variables on the total program size

Optimization level	Using inlined out with global variables	Using built-in out with local variables	Evolution
O0	12 793 words	12 935 words	+1.1%
O1	13 480 words	12 134 words	-10.0%
O2	13 052 words	11 131 words	-14.7%
O3	14 458 words	12 766 words	-11.7%
O4	14 458 words	12 766 words	-11.7%

Again, results are very conclusive in terms of program size. However, when no optimization is activated, the resulting program is bigger, and for some tests slower. This can be explained by the difference of addressing mode. Global variables are accessed directly via the data memory (using `d(var)`), when local variables are stored onto the stack. Hence, additional stack pointer computation needs to be performed, which is why performance and program size are slightly worse in O0.

Nevertheless, results are much better in higher optimization levels. Indeed, using local variables makes possible many of the usual optimizations that would not be possible

when using global variables. For example, constant propagation, sub-expression elimination, invariant code motion are more efficient with local variables, which is why performance are better when optimizations are activated.

For now, 17 built-in functions have been implemented in the compiler, which enhance significantly the possibilities of the compiler. For example, a specific built-in has been created to handle functions that have a variable number of arguments, such as the `printf` function. Also, many mathematic functions, such as the sine, cosine, exponential and square root functions have been implemented as built-in functions.

As previously mentioned, built-in functions are mostly input/output and system instructions and software functions already implemented in hardware (faster). Most of those built-in functions are not meant to be used directly by the programmer. However, they are essential when porting C standard library for this custom processor.

6.3 Porting the C standard library

Making a custom compiler is the first step towards a successful and user-friendly microprocessor. The second one is porting the C standard libraries [14] for this specific processor.

The C standard libraries consist of a collection of functions and library routines used to implement common operations, such as input/output, string and memory handling, and many more, in the C programming language. Those libraries provide an interface between the machine and the developer, so that the developer does not need to know the specificities of the target machine, which is needed for low level functions. A set of complete and optimized libraries helps to raise the level of abstraction so that the code used by the programmer is still machine-independent. Hence, the machine-dependent code is handled in the lower levels of those libraries. This explains why libraries need to be customized and optimized for each target.

Part of the project was to initiate such libraries so that basic functions would be provided to the end user. However, the purpose was not to build a set of functions as comprehensive as the one provided by the C standard libraries of GCC for an x86 architecture [15].

Also, it is sometimes hard to evaluate boundaries between services provided by libraries and services provided by an OS which is on an even lower abstraction level, compared to libraries. As standard libraries make use of target built-ins, they also use services provided by the OS. For example, file management and memory management are generally services provided by the OS. Hence, those services should not be developed inside the standard libraries.

As a result, a reduced set of functions has been developed and provided to the end user. Those libraries include for example the well-known `printf` function as well as similar functions `fprintf` and `sprintf`. Also some functions provided in math.h have been implemented such as the `exp` function.

The purpose of this was not only to provide libraries to the end user, but also to make sure that it was possible to build custom libraries using this tool-chain (compiler, assembler and linker). It turned out that the tools chain was lacking some features (for example, specific built-in functions to access the arguments of a function that can have a variable number of arguments), which had to be implemented to be able to develop such libraries.

Moreover, it should be pointed out that developing new libraries is also a way to broaden the spectrum of the tests. Writing code for libraries is different from writing code for a general program. It uses much more complex constructions. Hence, writing and testing those libraries is also a way to complete the existing test collection.

6.4 Fixed-point representation support

6.4.1 Context

Very few computer languages include built-in support for fixed-point values [16], since most applications use floating-point representations that are standardized. Floating-point representations are easier to use than fixed-point representations, because they can handle a wider dynamic range and do not require programmers to specify the number of digits after the radix point. On the other hand, floating-point representation requires either an FPU, which introduce rather large silicon area and power consumption on the chip, or software-emulated floating-point operations. This last solution introduces very large software overhead, which is translated into much larger code and much longer execution time.

This is why the floating-point representation might not be suited for embedded systems.

Providing basic support for fixed-point computation would be of great interest, both in terms of performance, and in terms of user-friendliness of the compiler. Some instructions, such as the move-and-shift instruction, are well suited for fixed-point computation. Nevertheless, the compiler was not able to use this kind of instructions when using software-emulated fixed-point representation.

This is why it has been decided to enhance the compiler's possibilities to handle built-in support for fixed-point representation. Existing compilers very rarely provide this feature. Providing built-in fixed-point support would bring a real added value, especially for embedded system programmers. It represents a breakthrough as far as efficiency, user-friendliness and capacities are concerned.

6.4.2 Compiler modifications

New data types have been created to handle fixed-point representation, using new target variable-attributes. Those attributes have the advantage of being ignored by the core of the compiler, so that special treatment of those attributes is handled in the backend. This way, the machine-independence principle of the frontend and the core is still respected.

Those variable-attributes represent the number of bits of the fractional part in the type. They are named `__fracX`, where `X` is the number of bits of the fractional part. This

attribute will be interpreted by the backend, which will perform special operation for those new types.

Using C standard data types in combination with fixed-point variable attributes makes many combinations possible. For example, an `unsigned int __frac15` will be interpreted by the backend as the following format:

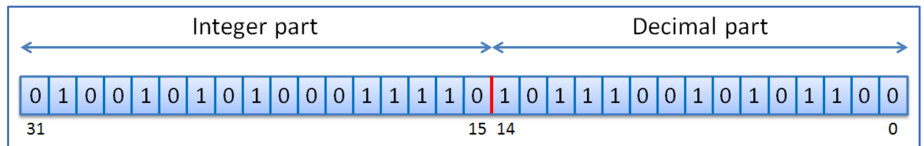

Figure 5 : Binary representation of an unsigned integer with 15 fractional bits

Whereas a `signed short __frac3` will be interpreted by the backend as the following format

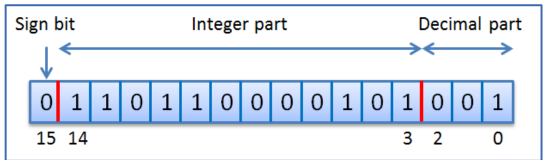

Figure 6 : Binary representation of a signed short with 3 fractional bits

This particular fixed-point representation is of great benefit for programmers. Without such built-in representation, it is impossible to perform a multiplication of two 32-bits integers inside a third one. Indeed, in fixed point representation, a multiply operation of two 32-bits integers produces a 64-bits integer, which cannot fit inside another 32-bits integer which forces programmers to use 64-bits integers. This is not optimized at all, and it forces the programmer to use twice as much data memory as needed since a `long int` must be used to store the temporary result.

The combination of built-in fixed-point representation and specific instructions makes it possible to perform the above-mentioned operation without any workaround. Indeed, the processor provides some instructions that are very interesting in fixed-point representation. For example, the MULAC computation unit (the unit that performs multiply-and-accumulate instructions) has an instruction that allows the programmer to retrieve the most significant bits (MSBs) of the accumulator to store them in memory, while discarding the least significant bits (LSBs). This instruction is perfect for fixed-point integer multiplication, and it would be translated into the following assembly code:

```
mul d(_a),d(_b)
mov $ACC>>32,d(_c)          # Performs c = a * b
```

The above shift-and-move instruction was provided by the processor, but any general purpose compiler would not be able to use this instruction since none of the C language structures would yield to the appropriate IR nodes that could be interpreted by the backend as a shift-and-move instruction. Using built-in fixed-point representation inside the compiler makes it possible.

6.5 Debug support

6.5.1 What is a debugger

When programming, it is very important to have powerful debugging tools. Debuggers such as GDB [17] are very famous for their capabilities that help many programmers. The capacity to automatically debug a program is the ability to go backward in the compilation process. The basic feature of a debugger is to be able to display the C source code from the executable file. To do so, additional information, called debug information, is added by the compiler to the executable file. This information will then be processed by an external debugging tool which is capable of retrieving the original source code, tracing and altering the execution of computer programs. Also, some debuggers allow the user to monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior. Yet those are very advanced debugging tools which require a lot of time to develop.

6.5.2 The need for debugging support

Because the compiler is under development, no debugging information was provided to the programmer. The only way to debug a program was to look at the assembly code produced by the compiler. When this is acceptable when debugging very small programs, it becomes very insufficient when debugging a larger program that uses libraries for example. That is why basic debugging information has been provided.

The first part was to include information as commentary inside the assembly code produced by the compiler. This helps relate the assembly code back to the source program by providing the operation performed (in the IR form), the line and the file where this code comes from. The following is an example of such information:

```
mov $R10,d($C1--)
#Put $C1 at the top of the stack
loadsubi 67,$C1
and $C4,$CON,$R3
#Instr ASSIGN, line: 46 of file printf.c
xor $CON,$CON,$R3
mov $R3,d($C1+18)
#Instr CALL, line: 67 of file printf.c
bra $C3,_atoi
```

Looking at this, it is possible to see how the compiler transformed a C code line into the target assembly code. Some bugs can be pointed out using this feature.

However, as described in section 2.3, the assembly code can only be accessed before the linking process. This means that it is only possible to look inside one only source file. If several files are needed to produce a final executable (as it is often the case), it is not easy to see how those files interact with each other.

That is why a disassembler has been developed. A disassembler is a very common computer program that translates machine language into assembly language, the inverse operation to that of an assembler. It is used to produce a string that is human-readable out of the instruction code and is very useful for programmers. This string can be

displayed inside the simulation program to be more interactive. Coupled to the existing simulation tools, it is of great help to see a human-readable trace of the execution of a program. Here is an example of the simulation tool coupled to the disassembler:

Figure 7 : Disassembly code inside the simulation tool

As depicted on **Figure 7** the disassembled instruction is much easier to read than the instruction code.

Nevertheless, additional information is needed to reach the level of abstraction provided by C language when debugging. Debug information which enables source-level debugging of executable programs is an important feature that improves greatly the user-friendliness of a compiler. Depending on the object format and debugger used, the format and capabilities of debug information can vary widely. Therefore, it is the responsibility of each backend to generate debug information. However, for common debug standards there are modules which can be used by the backend and do most of the work. Such a module was already available in the compiler, but it has never been used. This existing module uses the DWARF debug standard [18] and more precisely its second version, called DWARF2. DWARF is a widely used, standardized debugging data format. Work has been done in the backend to use the functions of DWARF module to insert debug information inside the assembly code.

However, the assembler and the linker are not able the process this information yet. Modifying those tools and develop a debugger is part of the project and is part of the future work.

7 Overall results and discussion

The evaluation of performance is done by collecting information (such as program size and execution time) using the test methodology described earlier. The objectives were to decrease the execution time of the set of test cases while also reducing the size of the resulting program. The main focus was on optimization of signal and data processing programs, which is the main field of application of this project.

During the thesis, many optimizations have been performed, each of them rising the global level of performance of the compiler. At the same time, new features have been developed to complete the set of tools of the compiler and its possibilities. Usually, adding new features is counterproductive because it makes the compiler and the generated code more complicated, but it has been managed to be avoided. Hence, the compiler is now more optimized and provides a better level of service to the developer.

7.1 Execution time

The most important criteria when evaluating the performance of a compiler is the execution time. The lower the execution time is, the better the compiler is optimized. **Figure 8** shows the evolution of the execution time of all the hardware tests in the different optimization levels available:

Figure 8 : Evolution of the overall execution time

The x-axis represents the different versions of the compiler that has been developed. Each of them corresponds to one or several optimizations that have been done and/or news features that have been implemented. A description of each version can be seen in **Appendix 6**.

This figure shows a global overview of the progress that has been performed: in optimization level `O0`, the execution time has been reduced by 22.0% (from 1 121 314 to 874 607 cycles) and by 18.3% (from 745 840 to 609 454 cycles) in optimization level `O4`.

This graph shows several significant steps, such as the one from version 6 to version 7, which is mainly due to a heavy work on the entire optimizer that was improved to delete unnecessary loop tests when the boundaries are constant. It was also at this point that it has been decided to activate the peephole for every optimization level. Hence, many unnecessary instructions got removed. The step between version 13 and version 14 can be explained by the optimization on the out-of-order algorithm as explained in **Section 5.2**.

The results on data-processing oriented algorithm are even more conclusive. One of the test cases is specifically focusing on data processing of matrices (matrix inversion algorithm, called the MT4 algorithm). For this algorithm, execution time has been reduced by 23.1% in `O0` optimization level and by 21.0% in `O4` optimization level. **Figure 9** shows the evolution of the execution time for the different versions of the compiler.

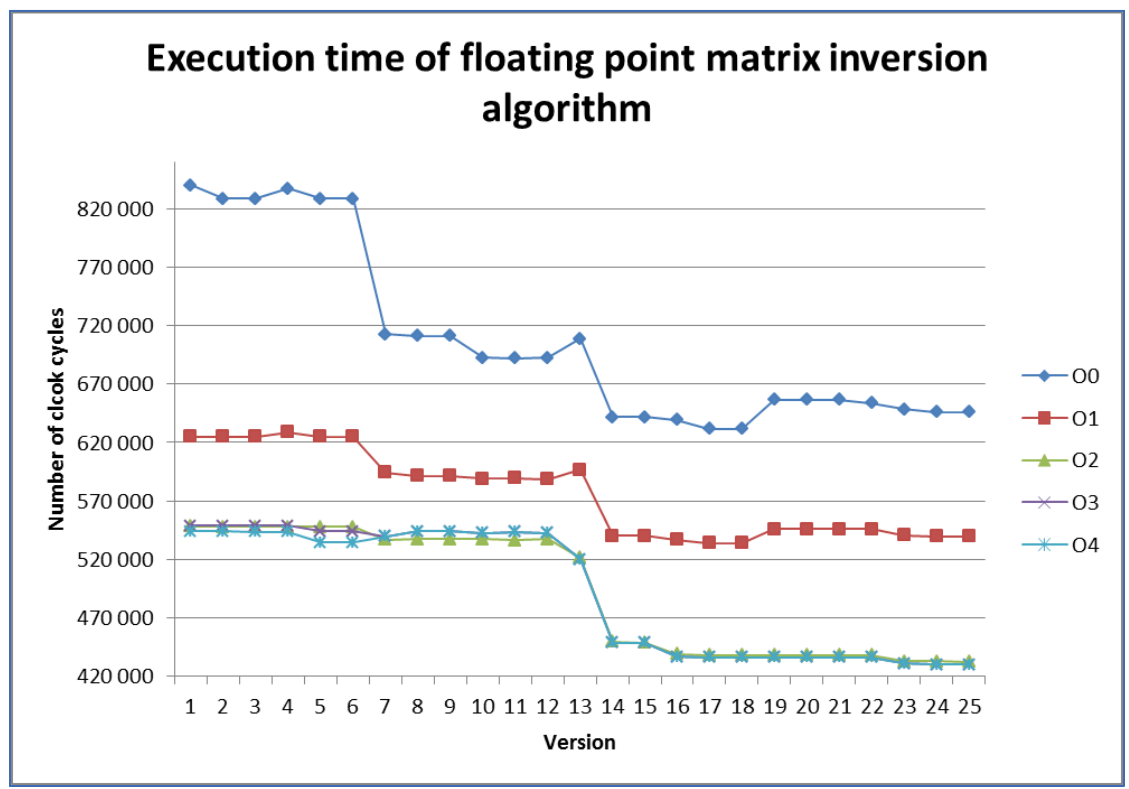

Figure 9 : Evolution of the execution time of the matrix inversion algorithm

7.2 Program size

The program size is the second most important criteria, especially in embedded system where the storage capacities are reduced. **Figure 10** shows the evolution of the program size of all the hardware tests in the different optimization levels available:

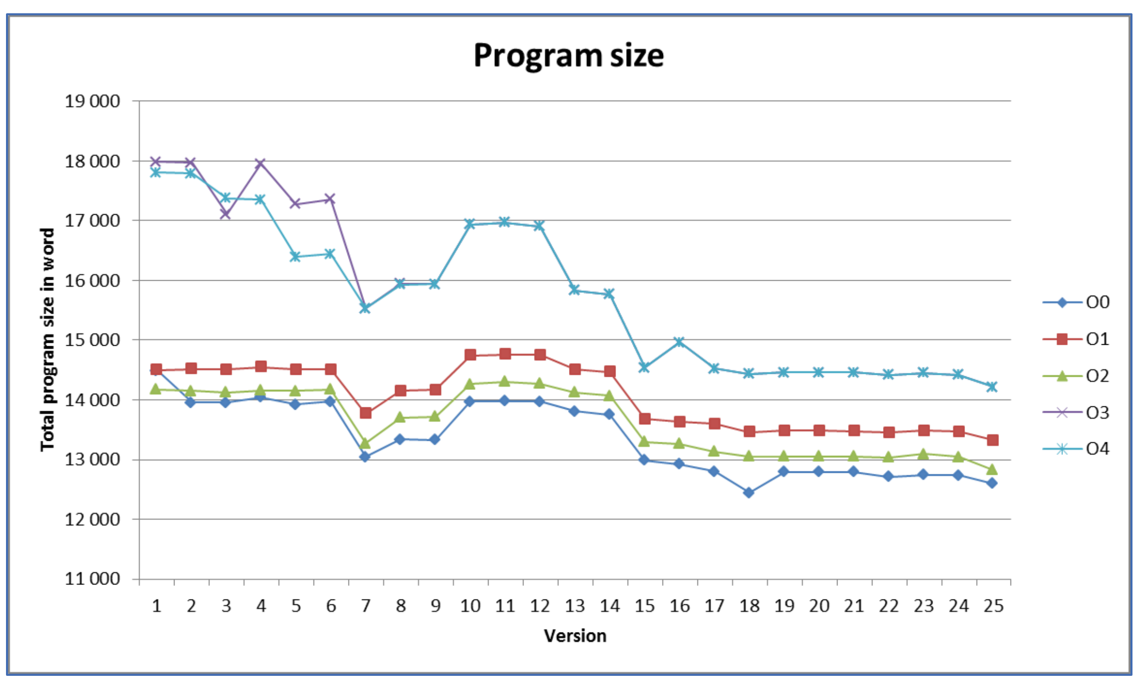

Figure 10 : Evolution of the overall program size

This figure shows a global overview of the progress that has been done in term of program size: in optimization level `O0`, it has been reduced by 13.1% (from 14 491 to 12 599 words) and by 20.2% (from 17 802 to 14 214 words) in optimization level `O4`.

Once again, test cases focusing on data processing present even better results. For the MT4 algorithm, program size has been reduced by 29.7% in `O0` optimization level and by 55.9% in `O4` optimization level, as it can be seen on the following figure:

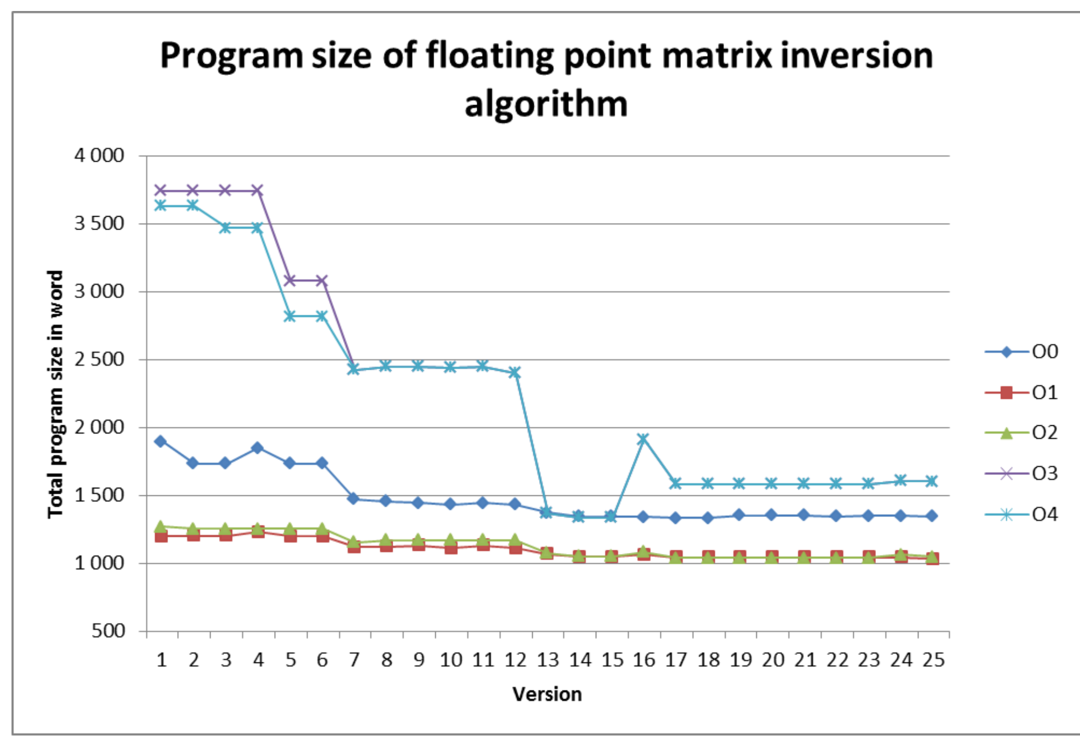

Figure 11: Evolution of the program size of the matrix inversion algorithm

8 Conclusion

To summarize this project, following conclusions have been made.

Developing and optimizing a compiler is something that takes time and that needs to be carried out very carefully, keeping in mind what the underlying hardware is. As a result it was possible to enhance both the performance and the services provided by the compiler so that it can now be spread over several departments of the company for more intensive use and feedback, which will make the development go faster.

Working on a "simple" compiler (compared to GCC) made heavy modifications of the backend (and the core) possible. This would have been much more complicated using GCC and would have taken a lot more time. GCC is very complicated which tend to be an issue when tweaking the compiler for an unusual target machine.

On the other hand, GCC is a much more stable and mature compiler. It has been developed for years by a large community of developers, and for a large set of target machines. Also, GCC comes with many additional tools from the GNU collection. As a result, built-in support, standard libraries, or debugging support is already taken care of, whereas it had to be developed entirely in the project's compiler.

This compiler's internal mechanisms were easy enough to be able to modify the compiler rather easily, which made all those optimizations possible. This is one of the most important criteria when developing an application during a limited period of time.

9 Future work

9.1 Instruction scheduling

A compiler is a translator. It reads every source line and transforms it into assembly code. And even though the core performs some basic block reordering, the compiler itself cannot reorder assembly instructions to make full use of the target module parallelism. This task is dedicated to the instruction scheduler.

As the processor has several modules that can run simultaneously (the ALU, the FPU, the MULAC unit, …), reordering assembly instruction could noticeably rise up performance. A floating-point division can take more than 30 clock cycles to execute. Without reordering the instructions, it is very likely that the processor will wait most of this time for this instruction to complete. Reordering helps to fill waiting time with other module computation and thus makes the computation ratio rise. A basic instruction scheduler has already been implemented, but it needs to be developed further to make full use of the parallelism offered by the processor.

9.2 Add new functions to standard libraries

C standard libraries are very complex, and demand a huge amount of work to implement each function that is usually provided by standard libraries. So far, only a subset of these libraries has been implemented. This is why their development should continue to keep providing more and more services to the final user, keeping in mind where the boundary between standard libraries and the OS is.

9.3 Create a complete source debugging tool-chain

The existing compilation tool-chain lacks of source debugging support. While it is not an important issue when debugging small programs, it becomes crucial when debugging large programs. Basic mechanisms for source debugging are in place but an automated debugging tool-chain does not exist yet. It is planned to develop such a debugger in the next six months.

10 References

[1] **Rainer Leupers, Peter Marwedel**. Retargetable Compiler Technology for Embedded Systems: Tools and applications. Springer Netherlands, 2001. ISBN: 0-7923-7578-5.

[2] **Andrew W. Appel**. Modern Compiler Implementation in C. Cambridge University Press, 2004. ISBN: 0-521-60765-5.

[3] **Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman**. Compilers: principles, techniques and tools, Second edition. Addison-Wesley, 2006. ISBN: 0-321-49169-6.

[4] **Fred C. Chow, Mark I. Himelstein, Earl Killian, Larry Weber**. Engineering a RISC compiler system. COMPCON 1986: 132-137.

[5] The C Preprocessor. [URL] http://gcc.gnu.org/onlinedocs/cpp/

[6] **William M. McKeeman**. Peephole optimization. Commun. ACM 8(7): 443-444, 1965.

[7] GCC, the GNU Compiler Collection. [URL] http://gcc.gnu.org/

[8] The LLVM Compiler Infrastructure. [URL] http://llvm.org/

[9] IEEE 754: Standard for Binary Floating-Point Arithmetic. [URL] http://grouper.ieee.org/groups/754/

[10] **John Cocke, Ken Kennedy**. An Algorithm for Reduction of Operator Strength. Commun. ACM 20(11): 850-856, 1977.

[11] **Keith D. Cooper, L. Taylor Simpson, Christopher A. Vick**. Operator Strength Reduction, ACM Trans. Program. Lang. Syst. 23(5): 603-625, 2001.

[12] **Jeffrey Sheldon, Walter Lee, Ben Greenwald, Saman Amarasinghe**. Strength Reduction of Integer Division and Modulo Operations. ACM ISBN:3-540-04029-3, 2001.

[13] **Tomasulo, Robert**. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM Journal of Research and Development. January 1967. 11(1):25-33.

[14] **Samuel P. Harbison, Guy L. Steele**. C: A Reference Manual (5th Edition). Prentice Hall, 2002. ISBN 0-13-089592X

[15] GNU C Library. [URL] http://www.gnu.org/s/libc/

[16] **Erick L. Oberstar**. Fixed-point representation and fractional Math. Revision 1.2, 2007.

[17] GDB: The GNU Project Debugger. [URL] http://www.gnu.org/software/gdb/

[18] DWARF official page. [URL] http://dwarfstd.org/

11 List of figures

Figure 1 : The binary generation tool-chain 6
Figure 2 : Internals of a compiler 8
Figure 3 : A modular compiler 12
Figure 4 : Overview of the different steps of a hardware test 18
Figure 5 : Binary representation of an unsigned integer with 15 fractional bits 39
Figure 6 : Binary representation of a signed short with 3 fractional bits 39
Figure 7 : Disassembly code inside the simulation tool 41
Figure 8 : Evolution of the overall execution time 43
Figure 9 : Evolution of the execution time of the matrix inversion algorithm 44
Figure 10 : Evolution of the overall program size 45
Figure 11: Evolution of the program size of the matrix inversion algorithm 45

12 List of Tables

Table 1 : Results of the mulneg optimization in terms of execution time 20
Table 2 : Results of the mulneg optimization in terms of program size 20
Table 3 : Results of the mulacc optimization on matrices multiplication of dimension 5 23
Table 4 : Results of the mulacc optimization on matrices multiplication of dimension 32 23
Table 5 : Results of the advanced strength reduction in terms of execution time 26
Table 6 : Results of the advanced strength reduction in terms of program size 26
Table 7 : Results of the register-only addressing mode in terms of execution time 29
Table 8 : Results of the register-only addressing mode in terms of program size 29
Table 9 : Results of support of hardware out-of-order algorithm in terms of execution time 31
Table 10 : Results of support of hardware out-of-order algorithm in terms of program size 31
Table 11 : Results of the use of built-in out with local variables on the total execution time 36
Table 12 : Results of the use of built-in out with local variables on the total program size 36

13 Appendices

Appendix 1. Description of each hardware tests

The first part of the hardware tests focus on very particular categories of C constructions and data types. Those tests contain a large variety of combinations in one particular area and focus on cases that might be tricky. Also, they are useful when evaluating performance on specific optimizations. Here is a brief description of those tests:

- Arithmetic_64.c: checks arithmetic on 64-bits integers which is emulated by software as it is a 32-bits processor. Currently, this file tests:
    - addition, subtraction, multiplication and shift on variables, constants and arrays
    - bitwise operations (and, or, not, xor) on variables, constants, arrays and pointers on pointers on long long
    - addition on pointers on long long and pointers on pointers on long long
- Arithmetic.c: checks arithmetic operations on integers. Currently, this file tests:
    - addition, subtraction, multiplication, division and shift on variables, constants, arrays and pointers
    - subtraction between pointers
- Arithmetic_extra.c: checks arithmetic operations which have reverse order of operands in assembler. More tests are performed than in arithmetic.c.
- Bitwise.c: check variables initialization and bitwise operations on integers. Currently, this file tests:
    - global variables initialization
    - variables assignment
    - and, or, not, xor
    - operators priority and parentheses
- Call_functions.c: checks functions: calls, parameters and return values. The verification is done on the return value after the function call. Currently, this file tests:
    - basic functions of type int, int*, void
    - functions with multiples parameters (up to 8)
    - function waiting for parameters of type int, long, int*, long*
    - that parameters are not altered by function when they should not have been
    - function call in a function
    - recursivity
    - that variables are actually locals (there are locals variables with same name in different functions which should not change after a function call)
    - static variables
    - function pointers

- data_type.c: checks conversions on every types others than floating-point and assignment from a variable to another variable. Currently, this file tests:
    - conversion from char to (char / unsigned char / short int / unsigned short int / int / unsigned int / long long int / unsigned long long int)
    - conversion from char to (char ... unsigned long long int) after an addition between two char
    - conversion to (char ... unsigned long long int) after an addition between int and char
    - conversion to (char ... unsigned long long int) after an addition between char and unsigned short int
    - conversion to (char ... unsigned long long int) after a subtraction between unsigned short int and unsigned int
    - conversion to (char ... unsigned long long int) after an addition between unsigned long long int and int
    - storage in memory and reload from it
- div_mode.c: checks extensively every possible division mode and type conversion between operands. Currently, this file tests:
    - division mode (8,16,32 or 64bits)
    - signed and unsigned division
    - cross type use and type conversion (example (unsigned float) f = (signed char) b / (unsigned long) c)
    - tricky values such as 0.0/(-0.0), use of NaNs, …
- Float.c: checks arithmetic on floating-point variables and conversions from floating-point types to integral types (and vice-versa). Currently, this file tests:
    - assignment
    - addition, subtraction, multiplication and division on variables, constants and arrays.
    - conversions from float to (double, int, long), double to (float, int, long), int to (float, double) and long to (float, double)
    - conversion int / long to float / double (and vice-versa) which could result in incorrect value (negative number, overflow, null number, underflow)
- for_loop1.c: checks for loops and addressing mode in it. Currently, this file tests:
    - basic for loops (growing and decreasing)
    - assignment on induction variable with constants, variables and arrays
    - stop with different tests using variables, constants and arrays
- struct.c: checks structure use. Currently, this file tests:
    - structure with int and long long fields
    - copy of field in a variable
    - assignment of a structure in another
    - addition of fields of different structures
    - function waiting for structure in parameter, and returning a structure

- while_loop1.c: checks while loop with different exit tests. Currently, this file tests:
  - basic while loops with <, >, <= and >=
  - tests on type int, unsigned int and long int with global variables, local variables, constants and arrays
  - if between values for which the comparison (subtraction) exceeds limits

- while_loop2.c: checks while loop with using floating-point variables. Currently, this file tests:
  - tests with float
  - if on double limits (positive infinity and negative infinity)

The second part of those tests is real and meaningful algorithms that focuses on evaluating performance in "real" conditions. Here is a brief description of those tests:

- MT4-invmat.c: a complex algorithm which computes a floating-point vector from a matrix. Currently, this file tests:
  - matrix initialization (for loop, modulo, division) in a function (matrix address is given as a parameter)
  - multiple computations on matrix (multiplication, division, addition, subtraction) performed in for loops
- Bubblesort.c: a well-known algorithm which sorts a list of integer.
- Luhn.c: an algorithm which performs a simple checksum.
- Matrix.c: checks operations on matrices. Currently, this file tests:
  - matrix initialization (for loop, modulo, division) in a function (matrix address is given as a parameter
  - matrix multiplication on int*int->int, long*long->long, double*double->double, int*long->long and double*int->long
- Pgcd.c: an algorithm which finds the Greatest common divisor (GCD) of two integers.
- Rot13.c: an encryption algorithm.
- Fir.c: an integer 51 taps Finite impulse response (FIR) filter

Appendix 2. Example of result file

```
#Results file
#  with stimulus file <tmp.stim>
#  with program <basic/arithmetic.bin>
#  with expected results <basic/arithmetic.c.gcc.ref>
#Executing program N°0 at 1.0225 us
#============================================================
#  Time      Page  Addr  Hex Data     pass  good data (hex & decimal)
 1.5225 us     0    22  0000012C     OK  0000012C           300
 1.5525 us     0     0  00000005     OK  00000005             5
 1.5725 us     0     1  FFFFFFF3     OK  FFFFFFF3           -13
 1.5975 us     0     2  00000018     OK  00000018            24
 3.0475 us     0    42  FFFFFFFF     OK  FFFFFFFF            -1
 3.0975 us     0    43  00000014     OK  00000014            20
#============================================================
#Program ends successfully at 3.1375 us

#Runtime duration : 2115 ns
#Runtime cycles : 423
#Runtime active : 387 (1935 ns) *** ratio=91% (Stall losses)
#Runtime stall count : 35 (36 cycles)

#Flag0 asserted : 15 cycles (72 ns)
#Flag1 asserted : 0 cycles (0 ns)
```

Appendix 3. Example of Hardware test report

call_functions	O0	O1	O2	O3	O4
Status	PASS	PASS	PASS	PASS	PASS
Cycles	3051	2896 (-155)	2966 (-85)	2137 (-914)	2137 (-914)
Max Expected	3346	3236	3104	2449	2449
Active Cycles	2952	2804 (-148)	2874 (-78)	2078 (-874)	2078 (-874)
Stall Count	55	49 (-6)	49 (-6)	32 (-23)	32 (-23)
Cycles in Stall	99	92 (-7)	92 (-7)	59 (-40)	59 (-40)
Ratio	96%	96% (+0%)	96% (+0%)	97% (+1%)	97% (+1%)
Program Size (in word)	898	842 (-56)	846 (-52)	970 (+72)	970 (+72)
Test Date	16/11/10 15:56:33	16/11/10 16:36:19	16/11/10 16:51:28	16/11/10 17:05:33	16/11/10 17:17:59
Notes	-	-	-	- More code emited	- More code emited

Appendix 4. Hardware tests summary

Overall Virtual Analysis Status: **WARNING**

Test Case	O0	O1	O2	O3	O4
arithmetic_64	**WARN**	PASS	PASS	PASS	PASS
arithmetic	PASS	PASS	PASS	PASS	PASS
struct	PASS	PASS	PASS	PASS	PASS
while-loop1	**WARN**	**WARN**	**WARN**	PASS	PASS
bubblesort	PASS	PASS	PASS	PASS	PASS
luhn	PASS	PASS	PASS	**WARN**	**WARN**
matrix	PASS	PASS	PASS	PASS	PASS
pgcd	PASS	**WARN**	**WARN**	**WARN**	**WARN**
rot13	PASS	PASS	PASS	PASS	PASS

Appendix 5. Example of compilation report

```
################################################################
##                                                            ##
## Compilation report                                         ##
## Thales Compiler Version 1.10                               ##
##                                                            ##
## Date : 16-02-2011 - 10:33:06                               ##
##                                                            ##
################################################################


/--------------------------------------------------------------/
/--------------  SUMMARY OF COMPILATION REPORT  --------------/
/--------------------------------------------------------------/

Compilation of file algo/matrix.c with options:
      - optimization level = 65537 (default optimization level -O1)
      - crossmodule optimization activated
      - inline peephole activated
      - fpu support activated (--fpu)
      - barrel shifter support activated (-barrelshift)

Memory used:
      - Data memory: 19 words
      - Program memory: 181 words   (174 instructions, 7 constants)

Number of different messages:
      - 2 NOTES
      - 1 WARNING
      - 1 HINT
      - 2 INTRUCTION OPTIMIZATIONS
      - 4 PEEPHOLE OPTIMIZATIONS

/--------------------------------------------------------------/
/--------------          END OF SUMMARY         --------------/
/--------------------------------------------------------------/


/**************************  NOTES  **************************/
sync instruction was not emitted as previous div mode is equal to the current one  (line 67 of
file matrix.c)
removed unnecessary start test for a FOR loop with constant boundaries             (line 167
of file matrix.c)

/*************************  WARNING  *************************/
Implicit conversion of operand 1 from SIGNED to UNSIGNED: make sure values contained by the
operand are possible UNSIGNED values  (line 76 of file matrix.c)

/***************************  HINT  **************************/
MULACC transformation line 173 was not possible due to modification of $ACC inside the loop.
Avoid use of other MULT to trigger optimization  (line 173 of file matrix.c)

/*****************  INTRUCTION OPTIMIZATIONS  ****************/
Optimized MULT by CONST=(power of 2) into LSHIFT by log2(CONST)  (line 86 of file matrix.c)
Optimized MULT by CONST into 1 LSHIFT and 1 ADD                  (line 173 of file matrix.c)

/******************  PEEPHOLE OPTIMIZATIONS  *****************/
nop instruction deleted                                          (line 565 of file matrix.asm)
loadsubi instructions deleted (loadaddi value has been changed)  (line 565 of file matrix.asm)
loadsubi instructions deleted (loadaddi value has been changed)  (line 707 of file matrix.asm)
nop instruction deleted                                          (line 720 of file matrix.asm)
loadaddi and loadsubi instructions deleted                       (line 817 of file matrix.asm)


################################################################
##                                                            ##
##                 End of compilation report                  ##
##                                                            ##
################################################################
```

Appendix 6. Description of the different versions of the compiler

Versions:	Description:
Version 1	• Removed `sync` instructions (not necessary anymore) • Bugfix while transforming `-f` into `-1*f` in floating-point numbers • Added basic debug information into the .asm file
Version 2	• Assembly peephole optimization
Version 3	• IR peephole optimization
Version 4	• Strength reduction on multiply by a power of 2 (optimized into a shift to the left)
Version 5	• Bugfix on `must_convert()` function, on the assembly peephole and on the shifting operation handler • Optimized `transform_trivial_operation()` function
Version 6	• Added support for division mode provided by the microprocessor (division on `char`, `short`, or `int`) • Added `mulneg` detection optimization • Strength reduction on division by a power of 2 (optimized into a shift to the right) and on unsigned modulo by a power of 2 (optimized into an and with value -1)
Version 7	• Bugfix on assembly peephole • Activation of the assembly peephole on every optimization level • Improvement of the `mulneg` detection • Suppression of unnecessary test for loops with constant boundaries
Version 8	• Added built-in function support to the whole compiler • Changed inline assembly for instruction `out` into a built-in function • Added `mulacc` detection optimization
Version 9	• Bugfix on `mulacc` and `mulneg` optimization • Optimized `ADD 1,X,Y` into `SUB -1,X,Y` for integer addition (uses `$CON` register)
Version 10	• Optimized strength reduction on multiply by a (power of 2)±1 or ±2 (optimized into a shift to the left and some `ADD/SUB`) • Improvement of the `mulacc` detection
Version 11	• Optimized operand access by swapping operands (depending on the different addressing mode of the different instructions)
Version 12	• Optimization of built-in function `out` into 2 separated built-in functions: a built-in `setpage` and a built-in `out`. Allows built-in function `setpage` to be hoisted out of the loop if possible
Version 13	• Added support for a new addressing mode for arithmetic instructions.
Version 14	• Added new built-in functions `flagset`, `flagclr`, `flagpulse`, `flagsetall`, `flagclrall`, `flagpulseall`, `sin`, and `cos`. • Added compiler support for multi-accumulator architecture (not activated yet) • Bugfix on instructions `mulclr.u` et `mulclr.us`, `fconvsp`, `mulacc` and `out` • Activated support for new FPU architecture using out of order algorithm. Allows using the same floating-point register as operand and destination. • Optimized parameters for loop optimization.
Version 15	• Many bugfixes • Changed support of instruction `div` to handle new format for the instruction

	(including 64bits integer division) using new instruction `div8`, `div16`, `div32`, `div64`, `divload`, and `divload64`.
Version 16	• Correct a bug on conversion algorithm from float to integer • Correct a bug on `fconvsp` • Correct a bug on addressing mode of instructions `mulclr.u` and `mulclr.us` • Correct a bug on built-in handling • Changed conversion algorithm in the core of the compiler for DIV operation
Version 17	• New peephole optimization to delete unnecessary move to tmp_Rreg and tmp_Creg • New peephole optimization to delete unnecessary reload of last_Rreg • Add support for new single precision FPU instructions. Now instructions such as `faddsp`, `fmulsp`, or `fsubsp` are available (is faster to compute than double precision instructions)
Version 18	• Add support for new instruction mov64 • Add new built-in sqrt and sqrtsp to be able to use those built-ins directly in the C code (very important for libraries) • New peephole optimization to deleted unnecessary move to tmp_FPreg and tmp_FPreg2
Version 19	• New peephole optimization to handle post-increment and post-decrement capabilities. • New peephole optimization to combine a `loadaddi` followed by a `loadsubi` by 1 single instruction `loadaddi` or `loadsubi`
Version 20	• Add new built-in __builtin_va_start to be able to handle variable arguments function (variadic functions) correctly
Version 21	• Optimized bitwise and arithmetic instructions when using indexed addressing mode for operand 1 done with use of `Cx+Imm` and operand swap • Correct a bug on built-in out • Correct a bug on `setreturn` for floating-point type • Correct a bug on wrong overlapping internal flag values
Version 22	• Peephole optimization to use post incrementation and post decrementation capabilities • Core optimization to delete unnecessary type conversion
Version 23	• Developed new built-in `in` • Correct a bug on multiply-to-shift conversion • Optimized crt0.s using `djnzd` instruction instead of `djnz`
Version 24	• Optimized `loadoperand()` function to take advantage of the new register-only addressing mode
Version 25	• Peephole to remove unnecessary `nop` instructions