**Universidade de Aveiro** Departamento de
Electrónica, Telecomunicações e Informática,
**2014**

**João Carlos**
**Cordeiro Pires**

# Portar o GNU linker para o EVP DSP
# Porting of GNU linker for the EVP DSP

**Universidade de Aveiro** Departamento de
**2014** Electrónica, Telecomunicações e Informática,

**João Carlos**
**Cordeiro Pires**

# Portar o GNU linker para o EVP DSP
# Porting of GNU linker for the EVP DSP

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Paulo Bacelar Reis Pedreiras, Professor do Departamento de Electónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Orlando Miguel Pires dos Reis Moreira, Ericsson, Holanda.

**Agradecimentos**
**Acknowledgements**

I would like to use this space to thanks all the people that help me during the last few years. I probably will forget some of you, but things happen.

First of all I want to thanks to Orlando Moreira and Dmitry Cheresiz for giving me the opportunity to do my master thesis with them at Ericsson. Also I have to thanks to Paulo Pedreiras for being my advisor, even though my work was done far away.

I want to thank to all the people that work at Ericsson for all their help. Specially to Cupertino Miranda, for all his help and all the knowledge that he shared with me.

To my friends, thank you all. Thanks for all the good times that we have spend together, and thanks for put up with me in my worst times.

E como é obvio um grande obrigado para a minha família, que sempre me apoiou e sempre esteve disponível nos bons e maus momentos.

## Abstract

Nowadays, with the ubiquitous presence of mobile devices, modems play a relevant role. The EVP is a vector processor used by Ericsson on its 2G, 3G and 4G modems. Due to size, weight and power constraints, mobile devices have limited power sources, and because of that the hardware must be power efficient. The processors used in this type of devices have multiple power consumption states. If a processor is in a idle state i.e. without code to execute, it can switch off and consume less power. Moreover, if it is known beforehand that the processing takes less time than available, the processor can be set to a lower frequency (and lower voltage level), executing the same amount of work, but consuming considerable less energy. With this dynamic power consumption the autonomy of the devices is extended, because the processor can be in a reduced power consumption state most of the time.

One way to maximize the time during which the processor stays in a reduced power consumption state is through the execution of optimized code. This can be achieved through manual code optimization, or by letting the compiler apply optimizations on the code. Link Time Optimization (LTO) is a compilation technique that tries to expand the optimizations scope by joining the input files and compile them as a unique file. Using this technique the compiler has more information during the optimization phase and should be able to do better decisions about the optimizations to apply.

The team responsible for the development of Embedded Vector Processor (EVP) wanted to evaluate the benefits of LTO. To be able to implement and evaluate LTO it was necessary to migrate the GNU linker to support EVP and enable LTO compilation on the EVP backend of GNU Compiler Collection (GCC).

This dissertation reports this work, including the necessary concepts to understand the problem addressed, how was added EVP support in the GNU linker, and the necessary step to enable LTO compilation on GCC. Also it will be presented and analysed the performance improvements of using LTO.

The LTO compilation presents mixed results, some of the test cases have a small performance improvements, and other present a small performance degradation. This was kind os expected because of the organization of the test code. It is composed by several files that do not have dependencies between them, in this situation the LTO compilation do not present much benefits.

**Resumo**

Hoje em dia, com a ubiquidade dos dispositivos móveis, os modems desempenham um papel relevante. O EVP é um processador vectorial usado pela Ericsson nos seus modems 2G, 3G e 4G. Devido às condicionantes ao nível de tamanho, peso e energia, os dispositivos moveis tem fontes de energia limitadas, e por isso o hardware tem de ser energéticamente eficiente. Os processadores usados neste tipo de dispositivos, tem múltiplos estados de consumo. Se um processador estiver inactivo i.e. sem ter código a ser executado, o processador pode ser desligado passando a consumir menor energia. Mais ainda se for conhecido de antemào que o processamento demora menos tempo que o disponível, o processador pode reduzir a sua frequência(e a sua tensão), realizando a mesma quantidade de trabalho mas consumindo considerávelmente menos energia. Com o consumo energético dinâmico, a autonomia dos dispositivos é estendida, porque o processador pode estar grande parte do tempo num estado de baixo consumo energético.

Uma maneira de maximizar o tempo que o processador está num estado de baixo consumo energético é através do uso de código optimizado. Este pode ser obtido através de optimização manual, ou deixando o compilador aplicar as otimizações. LTO é uma técnica de compilação que tenta expandir o âmbito das optimizações através da junção dos ficheiros de entrada e compila-los como um só. O uso desta técnica permite que o compilador tenha mais informação durante a fase de optimização e deve conseguir tomar melhores decisões sobre as optimizações a aplicar.

A equipa responsável pelo desenvolvimento do EVP queria avaliar os benefícios do LTO. Para implementar e avaliar o LTO era preciso adicionar suporte para o EVP ao GNU linker e activar o LTO no GCC especifico para o EVP.

Esta dissertação reporta o trabalho realizado, incluindo os conceitos necessários para perceber o problema, como foi adicionado o suporte para o EVP no GNU linker e os passos necessários para activar a compilação LTO no GCC. Vão ser também apresentadas e analisadas as alterações de desempenho quando o LTO é usado.

A compilao LTO apresenta resultado ambíguos, em alguns dos casos de teste houve pequenos aumentos de desempenho e noutros casos pequenas diminuições de desempenho. Estes resultados eram mais ou menos esperados, devido a organizao do código de teste. Este é composto por diversos ficheiros que não tem dependências entre si, nestas situações a compilação LTO não traz praticamente nenhuns benefícios.

# Contents

# List of Figures

# List of Tables

# Acronyms

| | |
|---|---|
| **GNU** | GNU's Not Unix |
| **ISA** | Instruction Set Architecture |
| **ELF** | Executable and Linkable Format |
| **LSB** | Least Significant Bit |
| **MSB** | Most Significant Bit |
| **LTO** | Link Time Optimization |
| **IPO** | InterProcedural Optimization |
| **BFD** | Binary File Descriptor |
| **COFF** | Common Object File Format |
| **EVP** | Embedded Vector Processor |
| **VLIW** | Very Large Instruction Word |
| **PCU** | Program Control Unit |
| **SDCU** | The Scalar Data Computation Unit |
| **SALU** | Scalar Arithmetic Logic Unit |
| **PALU** | Predicate Arithmetic Logic Unit |
| **SMAC** | Scalar Multiply/Accumulate |
| **SLSU** | Scalar Load/Store Unit |
| **VDCU** | Vector Data Computation Unit |
| **VALU** | Vector Arithmetic Logic Unit |
| **VMALU** | Vector Mask Arithmetic Logic Unit |
| **VMAC** | Vector Multiply/Accumulate Unit |
| **IVU** | Intra Vector Unit |
| **VSHU** | Vector Shuffle Unit |
| **VLSU** | Vector Load/Store Unit |
| **ACU** | Address Calculation Unit |
| **DSP** | Digital Signal Processor |
| **AST** | Abstract Syntax Tree |
| **CFG** | Context Free Grammar |
| **DIE** | Debugging Information Entry |
| **GCC** | GNU Compiler Collection |
| **WCDMA** | Wideband Code Division Multiple Access |
| **SoCs** | Systems on Chip |
| **ILP** | Instruction-Level parallelism |
| **SDK** | Software Development Kit |
| **ALU** | Arithmetic Logic Unit |
| **SLU** | Store Load Unit |

# Chapter 1

# Introduction

In the modern world with the massification of mobile/embedded devices, their limitations in memory space, processing power and limited power source became more problematic. The creators of these type of devices are always trying to improve the autonomy without sacrificing its functionalities. The trend at the moment (at least in the smartphones) is to increase the number of functionalities in every iteration and continue to increase autonomy. There are three main areas that could bring autonomy improvements.

- **battery capacity:** if the battery can hold more power, the device autonomy will be higher.
- **improve hardware power consumption:** if the hardware is more power efficient, the power consumption will be smaller and therefore the autonomy is increased.
- **software performance:** if a task takes less time to complete, the hardware can return to a power save state sooner.

The first two improvements are only applicable for new devices, because they are only available in new hardware revisions.

But for existing models it is not so easy to take advantage from battery improvements and practically impossible to use new hardware. However, software improvements can always be applied to devices by means of firmware updates.

Let us focus on the software performance option. In this option the idea is to reduce the execution time of the tasks. This can be achieved in various ways:

- **use specialized hardware:** it is common in embedded devices to find Systems on Chip (SoCs) that contain a general purpose processor and a few specialized coprocessors. A vector processor is an example of one such specialized coprocessor. It allows, for instance, adding two vector registers in one single instruction. A vector register is a wider register that contain multiple elements. For instance, in the Embedded Vector Processor (EVP) the vector register is 256 bits wide and can store 16 elements of 16 bits. On a regular processor it would take various instructions.
- **manually replace inefficient algorithms:** it is common to have algorithms that implement the same functionality but have different execution times. Replacing one for the other will bring a performance improvement maintaining the same functionalities. However this replacement is done by a programmer and can be expensive the search for a better algorithm.
- **use automatic optimization tools:** apply transformations on the program reducing the execution time. These techniques always result in a longer compilation time. In this situation the optimizations are done by automatic tools, from the programmer perspective there are no cost associated with it, but it is necessary to buy, or develop the optimization tools.

The third option is the more interesting one, mainly because it doesn't require additional work. It is only necessary to inform the compiler about the optimization that should be applied and let it do the work. It can be though as a "free" performance improvement from the application developer point of view.

The type and number of optimizations present in a compiler is dependent of its version and usually recent versions have more and better optimizations. For instance, on version 4.5.0, GNU Compiler Collection (GCC) introduced the ability to do Link Time Optimization (LTO). LTO allows the compilation of multiple source files as if it was a single one. The fact that the compiler operates on a single file instead of multiple files has huge benefits when applying optimizations, because it expands the scope where the optimizations are applied. This will be discussed in more depth on subsection 2.1.4.

The EVP is developed by Ericsson, and is used within the development of thin modems, operating on 2G, 3G and 4G telecommunication networks. These modems will be used in embedded devices and therefore must comply with the design constrains of embedded devices, e.g. limited power. The Software Development Kit (SDK) team, responsible for the development of the compiler, linker, assembler and simulator, is searching for ways to improve the performance of the compiled code. To pursue this path the team migrated the old compiler to the GCC compiler. The migration brought some new optimization techniques, one of these techniques being LTO. When the compilation is done with LTO and the linker has some special support, this compilation brings more improvements that when only the compiler is used.

The current linker is a custom tool created more that 10 years ago. The tool was also created with another processor in mind and some characteristics of the processor are reflected in the linker design choices. For instance, the way the linker deals with memory pages, even though EVP does not have paged memory. More important the linker does not have support for LTO compilation. This support in the linker is important to obtain the best optimization possible. So two approaches appear: add LTO support to the current linker or port a linker with LTO support to EVP.

The decision was to port the linker present in the `binutils` package. The `binutils` is a software package develop by GNU's Not Unix (GNU) project, and it contains tools to handle binary files. It contains a linker, an assembler, and tools to inspect binary files.

It is expected that when the work is complete to have a linker that is able to replace the current one, test the LTO compilation and have some data regarding the performance changes that LTO brings.

Through the duration of this work, the GNU linker `ld` was ported to support EVP, the LTO support was enabled on GCC and check the performance improvements of LTO compilation.

The document is organized in 6 chapters, the first one explains the reasons behind the work, the second gives the reader the necessary detail to understand the follow chapters, the third one explains in a more detailed way the problem addressed in this dissertation, the fourth chapter explains how the work was done, the fifth presents the analysis of the LTO performance improvements and in the last one the conclusions and future work are presented.

# Chapter 2

# Theoretical Introduction.

In the early days of computers, programming a computer involved physically modifying the memory bits. To remove the need to manually change the bits, the programs began to be introduced in the machine via a keyboard. But programs were still written in machine code i.e. in its binary format. The programmer continued to change the memory but now it was not necessary to physical change it. The programmer would write the program first in a piece of paper and then type it into the machine. Both write the program and type it was a tedious and error prone task. But specially the typing part, because the computers did not have a display so it was impossible to catch typing errors until the program was executed. The example on Listing 2.1 is the MIPS machine representation of the code present on Listing 2.2, to make the representation shorter the vales are represented in hexadecimal instead of binary. In this example two values are loaded and then summed.

```
0x20010001
0x20020002
0x00221820
```

Listing 2.1: Example of a program in machine code.

To raise the abstraction level the assemblers were created. The assembler abstracts the computer. For instance, it is no longer necessary to know the binary representation of the instructions because the assembler translates the program written in assembly language into machine code. The assembly language is a low-level programming language that has a strong tie to hardware and it uses mnemonics, to identify the instructions. An example of a program in assembly language can be seen in Listing 2.2, this assembly code is equivalent to the C code present in Listing 2.3

```
addi $at, $zero, 1
addi $v0, $zero, 2
add  $v1, $at, $v0
```

Listing 2.2: Example of a program in assembly code.

Further down the line, the abstraction level was raised once again. This time from assembly language to high level programming languages. These programming languages introduce a new layer of abstraction from the computer. One of the features introduced by high level languages is the concept of variable. A variable is an abstraction used to represent something in memory. It has a name and a type, both giving it semantic value. But perhaps more important allows the program to be independent of the memory layout. For instance it is easier to deal with a variable named `age` and let the compiler deal with its memory position. In Listing 2.3 is an example of a program written in C language.

```
int a = 1;
int b = 2;
int c = a + b;
```

Listing 2.3: Example of a program in C language.

## 2.1 From source code to executable.

For most programmers compiling a program to an executable is simple and straightforward. It is only necessary to click in a button or execute a single command in a command line. In reality, there are a few steps between a source file and an executable. These steps are hidden from most people for the sake of simplicity.

The steps necessary to create an executable from some source code are depicted in Figure 2.1. I will explain them one by one in the following sections.

### 2.1.1 Compilation phase

As can be seen in Figure 2.1 the compilation is the first step to create an executable. In this step a compiler is used to translate code between two languages, the original files, usually written in a high level language, and the final files in assembly.

The structure of compiler evolved with time. In the beginning the compiler was a single program written in a monolithic way. With the increasing complexity of programming languages the compiler structure started to split to make development easier and faster, since each of the parts could be developed by different teams. Moreover this separation allowed modular testing. The most common division at this moment is:

- **the front-end:** is responsible for language analysis and translation of the source code into an intermediate representation.
- **the middle-end:** works on the intermediate representation and makes generic code optimizations.
- **the back-end:** translate the intermediate representation to machine code and applies optimizations that are specific to that machine.

Each of these different parts of the compiler are also divided in different phases. Each of these phases is independent of the others and each of them transforms the input and feeds it to the next phase.[2] These are the different phases:

- lexical analysis
- syntax analysis
- semantic analysis
- code optimization
- code generation

#### Lexical Analysis

Lexer is the name given to the program/function used in this phase. The purpose of the lexer is create tokens from a sequence of characters. A token is a sequence of characters that has meaning as a whole. The lexer will receive as input the content, of a source file, parse the input file, and outputs a list of pairs, token and token class.

If we think about almost all programming languages we can identify a few token classes that exist in the language. For instance, in the C language we typically have classes such as:

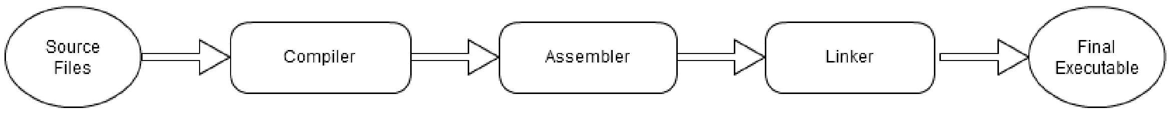

Figure 2.1: Passes from source code to executable

- **keywords:** words that are reserved for use by language constructs, e.g `for, if`
- **operators:** symbols that represent an operation. e.g `+, =`
- **identifiers:** usually a string of letter and number that starts with a letter. e.g. `foo`.
- **numbers:** a string compose only of digits. e.g `42`.
- **whitespace:** a sequence of spaces, tabs and newlines.

The lexer will go trough all the characters in the input file and tries to find the correspondent class. When this correspondence is met a pair of class and the sub-string is created and added to a list of tokens. Because of the simple nature of this step there are not many errors that can be caught here. They are basically malformed tokens i.e sub-strings that don't fit in any of the token classes.

An example of an input source and the corresponding lexer can be seen at Listings 2.4 and 2.5 respectively. On Listing 2.4 a token is something that is delimited by spaces. An example of a token is the keyword `if`. On Listing 2.5 we have a list of pairs. A pair is composed by a token and its class identifier. The list is built with all tokens present in Listing 2.4 and their correspondent classes.

```
if a == 0
  a = 1 ;
else
  a = 2 ;
```

Listing 2.4: Input of lexical step

```
<"if", keyword>, <"a", identifier>, <"==", operator>, <"0", number>, <"a"
    , identifier>, <"=", operator>, <"1", number>, <"else", keyword>, <"a
    ", identifier>, <"=", operator>, <"2", number>
```

Listing 2.5: Output of lexical step

## Syntax Analysis

This phase receives as an input a list composed by pairs token and token class, and outputs an Abstract Syntax Tree (AST). Parser is the name given to a function that creates an AST using a list of tokens, and grammar rules.

An AST is a data structure used to represent the structure of a program. An AST does not contain certain elements present in the source code because they are intrinsic to this structure. For instance, the parenthesis used in mathematical expressions to show the priority of the operands. In a AST the tree structure represents the order by which the expressions will be evaluated. Another important thing is the capability to add information to a node in an AST that will not change the execution result. This is important to store debug information.

A Context Free Grammar (CFG) is a grammar form used to represent a programming language. The reasons behind the use of CFG are: it is a simple representation, allowing an easy and efficient parsing.

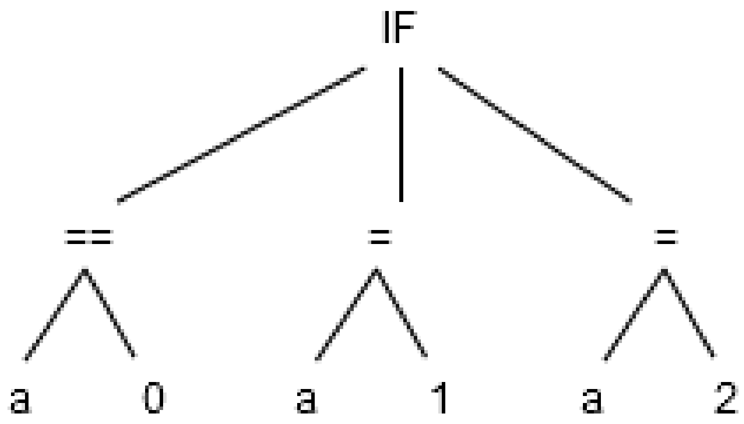

Figure 2.2: AST example.

During the construction of the AST all input tokens must be used. Because all tokens must match a rule in the CFG it is possible to catch all the situations where the input differs from the grammar rules. There are however some languages were the parser grammar is more abstract and the parser allows syntactical valid expressions that are not valid in the language. This is done to make the grammar simpler and these errors should be caught during semantic analysis.

On Figure 2.2 is an example of an AST that can be created from the code on Listing 2.4.

### Semantic Analysis

The semantic analysis receives an AST and outputs an AST with more information.

During this phase the AST is enhanced with more information about the program. With this new information invalid constructs that escaped the syntax analysis are caught. For example the type checking and verify if all the variables exist and are initialized before use are done in this phase.

### Code Optimization

The code optimization phase is usually the largest part of a compiler. It is composed by smaller parts and each of these parts does a specific type of optimization. Also it is the only phase that has not a well defined scope. This is a consequence of the fact that each optimization pass being independent from each other. Moreover which passes and the number of times that each pass will be executed is usually user-defined.

There are many optimizations that can be performed in this phase. To mention only a few:

- **dead code elimination:** in this pass, code that will never be executed is removed; this pass is discussed in more depth in section 2.1.4.
- **constant propagation:** there are situations where a variable is used to store a constant value. This is done because a variable has a name that gives it semantic value. Since it is a constant value the compiler could replace the variable for its value on all occurrences. Another thing done in this pass is the calculation of runtime constant expressions. An example of this can be seen in Listing 2.6.
- **loop transformations:** these are optimizations that operate on loops. Some examples, of optimizations that fit in this category.

- **loop invariant code:** in some situations some code inside a loop does not change its result between iterations. This code could be removed from inside the loop and the final result of the execution will be the same. See Listing 2.11 for an example.

- **loop unrolling.** In some situations exists the possibility that the code to manage the loop (variable incrementation, check for loop end) could be bigger than the code inside the loop. One way to reduce the loop overhead is to duplicate the code inside the loop and by this reduce the number of iterations needed in the original loop. See Listing 2.10 for an example.

## Code Generation

This is the last step in a compiler. The compiler will use a description of the target to transform the AST in assembly code. During the creation of the assembly code the compiler has to schedule the instructions in the correct order. It also has to define what variables will be stored in memory and what will be stored in registers.

It is also in this step that the debug information is derived from the extra information added to the AST.

```
double CirclePerimeter(Double radius)
{
  double pi = 3.1415; // constante value
  double perimeter = 2 * pi * radius;
  return perimeter;
}
```

Listing 2.6: Before constant propagation.

```
double CirclePerimeter(Double radius)
{
  double pi = 3.1415; // Unused value, can be removed.
  double perimeter = 6.283 * radius; // The two constant values are multiplied during the compilation.
  return perimeter;
}
```

Listing 2.7: After constant propagation.

```
int a, b, c;
c = 0;
b = 10;
for(i = 0; i < 10; i++)
{
  c +=i;
  a = b; // This line do always the same independent of the loop.
}
```

Listing 2.8: Before loop invariant extraction.

```
int a, b, c;
c = 0;
b = 10;
for(i = 0; i < 10; i++)
{
  c +=i;
}
  a = b; // So it can be removed from inside the loop.
```

Listing 2.9: After loop invariant extraction.

```c
int c;

for(i = 0; i < 10; i++)
{
  c = c + i;
}
```

Listing 2.10: Before loop unroll.

```c
int c;

for(i = 0; i < 10; i+=2) // The number of iterations is reduced to half.
{
  c = c + i;
  c = c + i + 1; // Duplicate the instructions inside the loop, to keep the same behavior.
}
```

Listing 2.11: After loop unroll.

### 2.1.2 Assembler

Just like the compiler, the assembler is also a translator, in this case not between two different languages but between two representations of the same language. Receives as the input the assembly code in a textual representation and outputs the same assembly code, this time in a binary representation.

#### Assembly language

The assembly language is a low level programming language that usually has a one to one relationship to each instruction in the Instruction Set Architecture (ISA) of the processor.

The basic block of the assembly language is the mnemonic that represents an operation and its parameters. The mnemonic tries to be related with the name of the operation. The number of parameters also depends on the type of operation. An example is shown on Listing 2.12. The `mov` instruction is used to move data from one place to another. So it must have two operands, the origin and destination. The `add` instruction adds two values and stores the result in a register. So it must have three operands, the two values to sum and the destination.

Some of the benefits of using assembly language instead of machine code are:

- **use of symbolic names:** in machine code all locations are described by a number, usually a memory address, but it could also be an offset that should be applied to the address of the current instruction. The use of symbolic names allows for more readable code and leaves the address resolution to the assembler. An example can be seen on Listings 2.13 and 2.14. On Listing 2.14 the instruction present on address `0x42` is associated with the `loop` label. When is necessary to reference this instruction the label can be used instead of its address. This allows the instructions to move freely across the code without the need to correct the addresses.

- **less machine dependant code:** if two processors implement the exact same ISA but the encoding is different, using two assemblers, each specific to each processor, allows the same code to be translated and executed in both processors. The use of symbolic name also helps sharing code between two processor architectures, because it allows memory abstraction.

- **usage of higher level constructs:** one example is the use of virtual instructions. These instructions are expanded by the assembler into two or more instructions. An example of this can be seen on Listings 2.15 and 2.16. For instance, in the MIPS architecture the instruction la(Load Address) that loads a 32 bits value into a register is translated into a lui(Load Upper Immediate) and a ori(Or Immediate), since the MIPS ISA does not have any instruction that have a 32 bits immediate field.

```
mov r1, r2
addi $2, $1, 1
```

Listing 2.12: MIPS Mnemonics

```
0x42  add r1, 3, 2
0x44  jump 0x42
```

Listing 2.13: Use raw addresses

```
0x42  loop: add r1, 3, 2
0x44        jump loop
```

Listing 2.14: Label use

```
la r1, 0x42
```

Listing 2.15: Example of virtual instruction

```
lui r1, 0x0
ori r1, 0x42
```

Listing 2.16: Expansion of virtual instruction.

### The assembler

The work done by the assembler is to convert a textual representation into binary encoding. It reads an instruction, converts it and repeats untils the end of the file. There is however a complication; the addition of symbolic names creates the need to know the value that each symbol represents. This can be done in a one pass or two pass schemes. [3]

In a one-pass assembler the symbol table is created at the time that the output is generated. This makes the assembler more complex because of the need to deal with symbols that are not yet defined. When the assembler finds a symbol there are three possible situations: the symbol exists in the symbol table and it is already defined, it exists in the symbol table but it is not defined or the symbol does not exist in the symbol table. The first situation is the easiest one, since the assembler only has to replace the symbol by its value and encode the instruction. The second and third cases are similar, the only difference is that the third one has to create a new entry in the symbol table. In these cases the instruction will be encoded with the field where the symbol will be stored set to zero and the assembler must keep track of the position where the undefined symbol appeared. After the end of the file was found the assembler needs to substitute all undefined symbols with the correct values.

In a two-pass assembler the input file is parsed two times, in the first pass the symbol table is created and all the symbols values are calculated. In the second pass, when a symbol is found, its value is retrieved from the symbol table and the instruction in encoded using that value.

## 2.1.3 Linker

As previously said, for each source file an object file will be generated. Usually the program is composed by more than one file, and at some point it is necessary to join them in order to make the final executable. This is the job of a linker.

When the assembler creates an object file, it sets the beginning of the file at the address zero. When a linker is linking more that one file it will have to adjust the addresses of the object files so that there are no address conflicts.

A linker will receive object files as input, and for each of these files will join the sections according the instructions on the linker script.[4] For more information about linker scripts see section 2.1.3. When the sections are joined their start addresses change to reflect their new positions, and that raises a problem on instructions that are position dependent. A position dependent instruction is an instruction that references explicitly a memory location. When these types of instructions appear is necessary to set the memory location to the correct address. This operation is called a relocation. It is responsibility of the assembler to decide when a relocation is needed, and the linker is responsible to resolve them.

### Relocations

A relocation is the way the assembler informs the linker about instructions that are position dependent and the linker will need to do some work on them.[4]

A relocation is an indication to an instruction that needs to be changed. A simple example of a relocation is when an instruction references an address inside the same section. When the section start address changes, all instructions inside this section will be shifted by the start address. In this situation the linker goes through all the relocations and adds the initial position of the section to the value in the instructions.

However a program is usually composed by more that one section and tends to have references between sections. In this situation it is necessary more information to resolve the relocation

One example of the information needed to resolve all types of relocation can be seen at Listing 2.17. This example shows the format that Executable and Linkable Format (ELF) files use to store relocations.[5] The ELF format has two ways to represent relocations, the difference between them is in the `r_addend` field. The `r_offset` gives the location where the relocation must be applied. The `r_info` stores two pieces of information. In the 8 Least Significant Bits (LSBs) are stored the type of relocation, the type is specific to each processor, and in the other 24 bits are stored the index in the symbol table of the symbol used to calculate the relocation. The `r_addend` is one of the values used to calculate the relocation value. This value can be stored in two places, in the structure, the case of `Elf32_Rela` or on the object file in the position where the relocation will be applied.

```
typedef struct {
  Elf32_Addr r_offset;
  Elf32_Word r_info;
} Elf32_Rel;

typedef struct {
  Elf32_Addr r_offset;
  Elf32_Word r_info;
  Elf32_Sword r_addend;
} Elf32_Rela;
```

Listing 2.17: Type of ELF relocations

The way that relocations are calculated has to do with their type and since the type is specific to each processor, the process can be really different between processors. The reasons behind this difference has to do with instruction encoding, and memory addressing modes.

To calculate the value of a relocation two values are needed, the `r_addend` and the value of the symbol given by the field `r_info`. The address is calculated by summing these two values. The calculation part is pretty much the same for all relocations, the big difference is how the value is stored in the instruction. An example of this difference is shown on Listings 2.18 and 2.19.

```
symbolValue = 0x00002014
addend = 0x0
relocation = 0x00002014 //symbolValue + addend
jmp 0x00002014
```

Listing 2.18: EVP relocation example

```
symbolValue = 0x00002014
addend = 0x0
relocation = 0x00002014 //symbolValue + addend
relocation = 0x805  //relocation >> 2
//relocation also drop the 4 MSB.
j 0x805
```

Listing 2.19: MIPS relocation example

On Listing 2.18 is shown a relocation where the complete address is stored in the instruction. Now on Listing 2.19 the size of the stored value is 26 bits, the four Most Significant Bit (MSB) are dropped and the value is shifted right two times.

## Linker Scripts

During the linking process the linker needs to know how to aggregate the different sections, and where each of these aggregations will be loaded into memory. The linker scripts are used to inform the linker on how to organize the executable.

The simple linker script present on Listing 2.20 tells the linker to put all input sections that have a name that ends with `.text` in an output section with the name `.text`.[6] The start address of this section will be 0x1000. The other sections(`.data, .bss`) will have the same behaviour except for the load address, which in the `.data` section, the load address will be 0x8000 and the `.bss` section will be loaded at the end of `.data` section.

```
SECTIONS
{
    .text 0x1000 : { *(.text) }
    .data 0x8000 : { *(.data) }
    .bss : { *(.bss) }
}
```

Listing 2.20: A simple linker script.

The linker scripts are much more powerful. We only saw a simple example that shows how to aggregate sections and define where they should be loaded. A few more things that are possible to do with linker scripts are:

- create new symbols.
- create memory bocks that have a start address, a size and access flags(i.e. read, write, executable).
- create unusual memory layouts, e.g more that one memory space, overlays.

## Overlays

The use of overlays is a method that allows the execution of programs bigger than the physical memory available. This feature was relevant in the beginning of the computer age were the available memory was small, and there was the need to run programs that were bigger than the memory available. Today even if it is not a necessary feature in the computers, continues to be a relevant feature in the embedded world, were the available memory continues to be scarce.[4]

As it can be deduced from its name, overlays refer to some sections of a program that have the same memory address, i.e. that their memory addresses overlap. The linker will create sections that have the same memory address and then in runtime the program load/swap the overlays according to the needed functionalities.

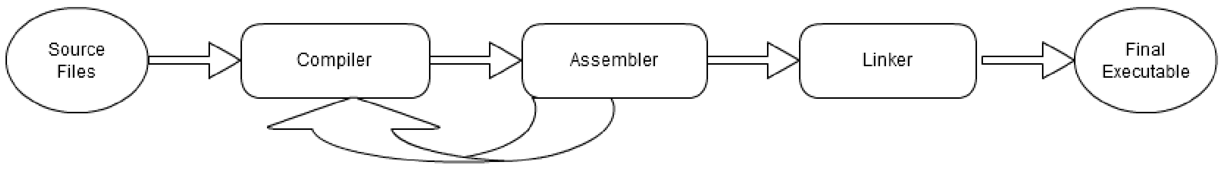

Figure 2.3: Passes from source code to executable in LTO

One way to create a simple overlay is shown on Listing 2.21. If we compare the linker script on Listing 2.20 with the one on Listing 2.21 it is easy to see that the last one has two more section, `.ov1` and `.ov2`. Each of these sections is a overlay. These two section have the same load address, and the code will be prepared to run in that memory location. Also since the sections have the same start address they can not be loaded into memory at the same time.

The example shown in Listing 2.21 is a simple example of overlays from the point of view of the linker. All overlay management must be done by the program which needs to know how the functions are spread across the different overlays, how to load, and when to load them. However there are also implementations that change the burden of managing the overlays from the programmer to the linker.

```
SECTIONS
{
    .text 0x1000 : { *(.text) }
    .ov1  0x3000 : { *(.ov1)  }
    .ov2  0x3000 : { *(.ov2)  }
    .data 0x8000 : { *(.data) }
    .bss     : { *(.bss)  }
}
```

Listing 2.21: A simple overlay linker script.

### 2.1.4 Link Time Optimization

As it was said before, during the compilation phase each source file is compiled independent of the others. This can be seen as a beneficial thing since the process could be executed in parallel, but also has the problem that no information can be shared between the different compilation processes. Also for various reasons the code tend to be divided in functions and these functions spread across different files, this means that the information about the dependencies of a function may not be present in the optimization phase.

InterProcedural Optimization (IPO) is the name given to optimization techniques that expand the scope of optimizations across functions.[2] When applying IPO the more information about the whole program the compiler has, the better.

LTO is one of the methodologies used to increase the information present in the compiler before applying the IPO.

The compilation flow when LTO is active is shown in Figure 2.3. As it can be seen the files created by the assembler are feed as input to the compiler and the compilation process is repeated. Another difference is in the files generated by the assembler. These files have additional information that is sufficient for the compiler to recreate its internal structures and continue with the compilation process.[7]

### First Compilation

The first compilation is almost identical between a normal compilation and a LTO compilation. The big difference is that when LTO is active the compiler will output the necessary information to latter continue with the compilation process. This information is emitted in special sections on the assembly file and they contain the intermediate representation, a symbol table and all the definitions for the compilation unit, i.e. the input source code. The object files outputted by a LTO compiler will be called “fat” object files. However the “fat” object files have the side effects of doubling the compilation time and increse their sizes.[7]

### Second Compilation

In the second invocation, the compiler will receive a group of “fat” object files instead of source files. The compiler will read the intermediate representation from each of the files and it reconstructs the compilation unit for each file. After all files are read the compiler will merge all the compilation units into one.

The compiler will then run the optimization phase again. The compilation unit at this moment can be big and that means bigger compilation times. To improve the compilation times the GCC has two ways of running the optimization phase. [7]

- **single compilation unit:** the compiler works on a single compilation unit. Since it has more information the global optimization can be better, but the compilation time will also increase.
- **multiple compilation units:** the compiler will create smaller compilation units and apply the optimizations in each of these smaller units. The partition should try to minimize dependencies between the compilation units. Since the compiler has multiple compilation units the optimization process can be run in parallel.

At the end of the optimization phase the compiler will generate assembly files that will be assembled, and linked like in a normal compilation.

### Linker plugin

This feature allow the exchange of information between the linker and the compiler.

The “fat” object files contain both the assembly code and the information needed to resume the compilation process. These files can be used to build libraries that contain executable code and the information needed in the second pass of a LTO compilation. Let’s call these libraries “fat” libraries.

Since a “fat” library contains the information needed on the second compilation, why not use these information when compiling programs that use the library.

This information can be used if the linker has support for it, through the use of the linker plugins. The idea is, after the first compilation the linker will be invoked and it will resolve the external dependencies. The linker will then invoke the second compilation with the object files generated in the first compilation and all information gathered in the linking process. Using this compilation process, the amount of data that is given to the second compilation is bigger and because of that there are more room to optimizations.

### IPO Benefits

IPO is a set of techniques that uses the relationship between caller and callee in order to better optimized them.[2] The main optimization techniques used are:

- **constant propation:** it was already discussed in section 2.1.1.
- **procedure cloning:** is used to duplicate functions that one of the parameters is constant. It can be seen at an extreme case of constant propagation. For instance, assume the follow function `f(int i)`, if this function is always called with `i` equals five, the function can be cloned and create the

function `f_5()`. This one does not have parameters, and in all cases where the `i` was used it was replaced by the value five.

- **dead code elimination**
- **function inlining**

### Inlining

The inlining technique is based on the possibility to replace a function call by the function content. This is beneficial because saves the cost of a function call, this is more notorious if the function that was inlined is small, because in this case most of the execution time is spend on the call. However the inlining functionality has the side effect of code size increase. When a function is inlined, the calls to that function are being replaced by its content, that means that the code is duplicated through all the executable. Usually the compiler takes the code increase as a metric to decided if the function should be inlined.

Since the function call will be replaced by the function code, it must be known on the compilation unit where the function is called. For instance if the code on Listing 2.22 is on the same compilation unit the function call of `circlePerimeter` on line 2 can be replaced by the content of print function, obtaining the code shown on Listing 2.23.

As it was said before for the inlining to work the caller and callee must be on the same compilation unit, which is not always true. The use of LTO has the benefit of join all the compilation units into a single one. Now in this bigger compilation unit the optimization process could do a more aggressive inlining.

Another interesting thing is that the inline optimization can produce dead code. The `circlePerimeter` function on Listing 2.23 is never executed and it can be safely removed.

```
1  ...
2    double perimeter =
         circlePerimeter(radius);
3  ...
4
5  double circlePerimeter(double
       radius)
6  {
7    double perimeter = 6.283 *
         radius;
8    return perimeter;
9  }
```

Listing 2.22: Before inline function.

```
1  ...
2    double perimeter = 6.283 *
         radius;
3  ...
4
5  double circlePerimeter(double
       radius)
6  {
7    double perimeter = 6.283 *
         radius;
8    return perimeter;
9  }
```

Listing 2.23: After inlined function.

### Dead Code Elimination

The dead code elimination is a technique that finds and removes code that will never be executed. This optimization usually has more impact in the code size than in the execution speed.

An example of dead code elimination can be seen at Listing 2.24. In this example if both functions `main` and `deadFunction` are in different compilation units, then it is impossible to know the message that will be printed.

However when using LTO the function `deadFunction` on line 6 will be the target of constant propagation and becomes obvious that the code present on the else block will always be executed, in this situation the line 3. Since the executed code is always the same, the rest of the code can be removed. After removing the unneeded code from the `main`, it became like the one shown on Listing 2.25.

```
int main()
{
  if(deadFunction())
  {
    printf("Never executed");
  }
  else
  {
    printf("Always executed");
  }
}

int deadFunction()
{
  return 0;
}
```

Listing 2.24: Before dead code elimination.

```
int main()
{
  printf("Always executed");
}

int deadFunction()
{
  return 0;
}
```

Listing 2.25: After dead code elimination.

## 2.2 Object files

An object file is a file that contains code and data generated by the assembler in a binary form.[4]
Besides code and data, a list of relocations, a list of symbols and debug information can also be present in an object file. The information present depends upon the purpose of the object file. For instance a final version of a program has no need for debug information.
An object file has a combination of these three functionalities:

- **linkable:** is a file that will be used as input for a linker. Besides code and data this type also has a symbol table and a relocation table.
- **executable:** its purpose is to be loaded into memory and executed. Not that much information is needed in this type of file besides code and data, but the code and data must be arranged in a way that allow execution, i.e. all relocation resolved, code and data starting address must comply with memory constraints.
- **loadable:** it is able to be loaded into memory as a library. Have almost the same configuration as a executable, the big difference is that a loadable file could have some relocation information that will be resolved when is loaded into memory.

### 2.2.1 ELF Files

There are a lot of different types object files. It goes from simple ones that only have data to the more complex ones. The ELF format belongs to the last category. It is a fairly complex format because supports linkable, executable and loadable files. This format was created by AT&T to replace the Common Object File Format (COFF) format as the default object file format in Unix System V.[5]
The ELF format has two different interpretation, one made by the assembler and linker and the other by the loader. On the first interpretation the basic unit is the section. The assembler creates sections based on information given by the compiler, the linker goes through all sections, joins them and creates program headers. A program header is composed by multiple sections and defines the memory layout.

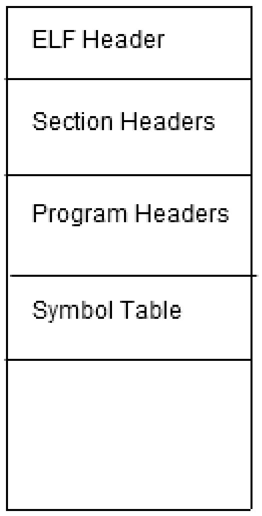

Figure 2.4: ELF file Structure

The other view is the interpretation of the loader, that will look at the program headers and loads them at the correct memory addresses.

A possible structure of an ELF file is show on Figure 2.4. Note that only the ELF header has a fixed position.

- **ELF header:** It is always at the beginning of the file, and contains information about the file. e.g. file type, size, endiness, entry point, the location and size for the other structures.
- **section headers:** Is an array of section header structures. Each section header describes a section. It contains the name, size, type, flags, location and memory load address of a section.
- **program headers:** An array of program headers. Each program header describes a memory segment. It contains the type, location, flags, and memory load address of a segment.
- **symbol table:** Contains all the symbols and respective values.
- **string table:** Contains the strings that are used on the names of sections and symbols. A string is a sequence of null terminated characters.
- **relocations:** A list with the relocations to apply. The format used to store the relocations is shown at Listing 2.17.

## 2.3 Binutils

Binutils is a open source project. Its purpose is to provide free tools to create and manage binary files. The main focus is the development of a linker and an assembler, even tough a few more small tools are present. Other important functionality is the capability to manipulate multiple types of object files, support various processor architectures and support for diverse operating systems.

In order to achieve the multiple object files support and multiple processor architectures, all file manipulations are abstracted thought the use of `libbfd`.

### 2.3.1 Binary File Descriptor

With the intention to handle multiple object files and support multiple processor architectures, Binary File Descriptor (BFD) was proposed and implemented in the `libbfd` library. The purpose of `libbfd` is to handle the myriad of different combinations of object files and processor architectures in a simple and equal way. One of the benefits of this approach is that it makes the addition of a new target simpler, because only the code of `libbfd` needs to be changed.

`libbfd` has two different parts, lets call them front-end and back-end. The front-end is the interaction point for the clients. It is composed by functions to manipulate BFD structures. The back-end is specific to each processor architecture and object file type, and it is responsible for the translation between object files and its BFD representation and vice versa.

A simple interaction could be, a client opens a file, the back-end will translate it to a BFD structure, the client manipulate the BFD and in the end writes the result to a file. Once again the back-end will be used to do the translation but this time from BFD to object file. Note that the origin and destination files may have different types.

### 2.3.2 ld

`ld` is one of the two linkers available in binutils package. Its purpose is to have a linker that allows the linkage of different object file types on multiple computer architectures.

`ld` achieves the multiple object files support with the use of `libbfd`. Using `libbfd` allows to handle in a uniform way all the different object file types. The linker could receive as input multiple object files in different formats that will be linked together.

### 2.3.3 gas

`gas` is the assembler in binutils package. It is an assembler that supports various object file types trough the use of `libbfd`. Is has support for multiple processor architectures and respective assembly languages.

### 2.3.4 gold

`gold` is a linker that only handles ELF files and a few processor architectures. It was created to have better performance that `ld`, specially when linking C++ code. It is one of the few tools in Binutils package that do not make use of `libbfd`.

### 2.3.5 Other tools

Binutils has a few more tools. None of them as big and complex as `ld`, `gas` or `gold`.

- **objdump:** is used to inspect object files. It allows to see the list of section, symbols, relocations and a few more relevant information. Makes use of `libbfd` to read files.
- **readelf:** has the same functionalities as `objdump` but only for ELF files. It was created as a way to debug ELF backend of `libbfd`.

## 2.4 Debug information

Debug information is extra information present in an executable. This information is not necessary for the correct execution of the program, but it is used by a debugger to help debug the program. The use of debug information allows for source-level debugging. As the name implies the source-level debugging allows the debugger to show the original source code, and the debug operations are applied into the original source code instead of assembly code, i.e. allows to step to the next line of code, inspect the value of variables, break the execution when a function or line of code is reached.

### 2.4.1 DWARF

The DWARF format was created to be used by the C compiler and debugger present in the Unix System V Release 4 during the 1980s.[8] The format continue to evolve with time. The more notable changes from the first version are, reduce the generated data, support for new languages, new processors and extend the support for current supported languages.

The information is stored in a tree format, where each node is a Debugging Information Entry (DIE). A DIE is composed by a Tag field that represents the DIE type, and data fields that are specific to the DIE type. The names given to the tags field are prefixed by DW_TAG and the name of fields are prefixed by DW_AT. In Listing 2.26 it is shown an example of a DIE. In this case it is shown an example of an integer type with 4 bytes.[8]

```
DW_TAG_base_type
DW_AT_name = int
DW_AT_byte_size = 4
DW_AT_encoding = signed
```

Listing 2.26: A DIE that represent and integer type.

## 2.5 EVP

The EVP is a embedded vector Digital Signal Processor (DSP) developed by Ericsson and used in their 2G, 3G and 4G modems.

The following characteristics are used to reduce the complexity of the hardware and make it more power efficient.

- **Exposed pipeline:** this means that the processor does not have hardware to stall, insert bubbles or flush the pipeline. When these capabilities don't exist the compiler becomes responsible for the correct execution of the pipeline. So the compiler must guaranteed that the generated code does not create resource conflicts and the delay slots are properly filled.
- **Very Large Instruction Word (VLIW):** it is a method that creates bundles of instructions that don't have dependencies between them and therefore can be executed in parallel.

### 2.5.1 Architecture

A block diagram of the architecture of EVP can be seen in Figure 2.5

The more relevant block are:

- **Program Control Unit (PCU):** handles the instruction decoding, branching and hardware loops.
- **The Scalar Data Computation Unit (SDCU):** executes the scalar instructions. This block contains two register files, the scalar one with 32 registers that are 16 bits wide and the predicate one with 8 register that are one bit wide. The scalar registers are named from `r0` to `r31` and each of them can be used as a 16 bits operand. It is also possible to join two register together in order to archive 32 bits operand. In this situation the two register must be contiguous and the lower one must be an even number e.g `r2r3`. This block also contains a Scalar Arithmetic Logic Unit (SALU), a Predicate Arithmetic Logic Unit (PALU), a Scalar Multiply/Accumulate (SMAC) and a Scalar Load/Store Unit (SLSU).

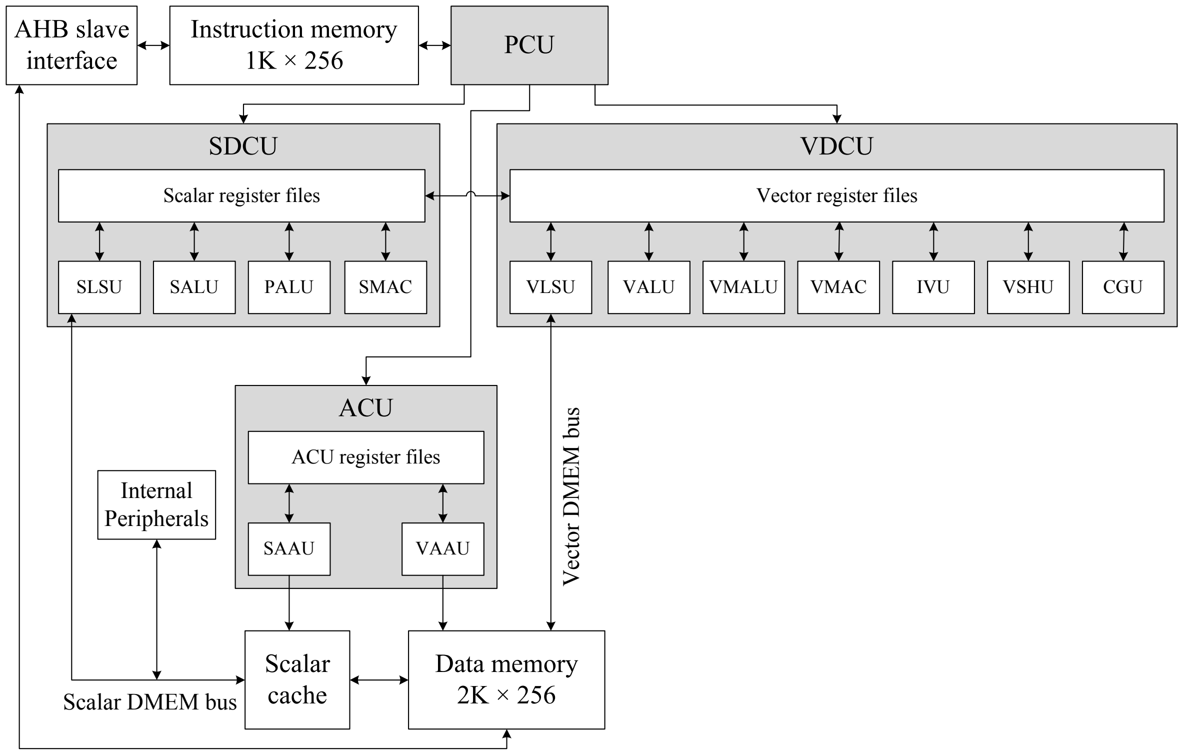

Figure 2.5: EVP core block diagram.[1]

- **Vector Data Computation Unit (VDCU):** takes care of vector instructions. In this block are present five register files. The vector register file (`vr`) has 16 registers 256 bits wide. The register can store 32 elements of 8 bits, 16 elements of 16 bits and 8 elements of 32 bits. Two or three registers can be joined in order to make a 512 bits or 640 bits register which can hold 16 elements of 32 bits or 16 elements of 40 bits. The mask register file (`vm`) contains 8 elements of 32 bits and is used to make the storage of the calculated values optional. i.e. depending if a bit is one or zero the corresponding vector element will be saved or not. The `ir` and `im` register files are used to store the result of inter-vector operations, on `vr` and `vm` register respectively. The `vsp` are used in the operations on Vector Shuffle Unit (VSHU). Other unit present in this block are, Vector Arithmetic Logic Unit (VALU), a Vector Mask Arithmetic Logic Unit (VMALU), a Vector Multiply/Accumulate Unit (VMAC), a Intra Vector Unit (IVU), a VSHU and a VDCU.
- **Address Calculation Unit (ACU):** performs address calculation using the pointer and offset register that will be used by Vector Load/Store Unit (VLSU) and SLSU.

### 2.5.2 Exposed pipeline

The pipeline technique is used in processors to enhance performance.[9] The idea behind pipeline is that instructions processing can be divided in small steps and these steps can be executed in parallel. This allows that multiple instructions are in execution, each of them will be in a different step and this way increase the throughput.

One of the assumptions when writing a program is that when an instruction starts to execute the previous one has already finish its execution. However this is false in a pipelined processor. This is problematic when two instructions that have a data dependency between them are executed, because it could lead to unexpected results. In order to guarantee that the execution result is the same between a pipelined processor and a non pipelined it is necessary to take care of the data dependencies between instructions. For this is necessary to maintain a list of instructions in execution and their dependencies. If at some point one of the instructions does not have a dependency satisfied because another instruction did not finish its execution, that instruction and all that are behind must be stalled until the dependency is satisfied and the execution can continue.

Another situation where the pipeline content must be changed is in the case of branches. When a branch is started to execute the next instruction is not yet known. Now there are two options.

- The next instructions slots are fill with nops (the number depends on how many cycles the branch takes to execute) and after the branch is resolved starts to execute the correct instruction. This is call delay slots.
- Chose one of the two possible instructions. After the branch is resolved if the choice was correct, continue with the processing. Otherwise clean the pipeline and start processing the correct instruction.

A processor is said to have a exposed pipeline when the pipeline control is done by the compiler, i.e. the compiler will schedule the instructions to avoid data dependencies and it must fill the correct number of delay slots. The content of delay slots may be nops or instructions that have to be executed and have no effect in the branch result.

### 2.5.3 VLIW

Modern processors are composed by multiple functional units. Each of these units implements a set of related operations, e.g. the Arithmetic Logic Unit (ALU) implements the arithmetic and logic operations, and a Store Load Unit (SLU) implements the operations of load and store information on external memory. They also have multiple units that implement the same functionality and can be executed in parallel. With the existence of multiple units, there is the possibility of executing multiple instruction in parallel.

The VLIW is one architecture design technique used to maximise the processor usage. The idea is to execute instructions in parallel that do not have dependencies between them. In an ideal scenario all functional units are being used. There are a few techniques to archive this. The VLIW, superscalar, and out-of-order execution. These techniques could be separated into two groups. In one group the instruction schedule is done on the fly by the processor. In this group is the superscalar and the out-of-order executions. Because the scheduling is done by the hardware, this one becomes much more complex. On the other hand the compiler has no additional complexity. On the other group the instruction scheduling is done on compile time, which is the case for VLIW. In this case the compiler will be more complex but the hardware will be simpler.

As the name suggests the VLIW uses large instructions.[9] Each instruction can be seen as a bundle of smaller instructions, where the small instructions control a functional unit. Obviously the number of small instructions present in these bundles has some constraints.

- **number of functional units:** the number of units present on the processor defines the maximum number of small instructions per bundle.
- **size:** there could be cases where the size of the bundle is not big enough to have instructions to all functional units.
- **resource conflicts:** the main source of conflicts is when a instruction wants to use a value that is calculated by the previous instruction. These two instructions can not be executed in parallel, this is called a data dependency. Another type os conflicts can be called architectural conflicts. These

type of conflicts are intrinsic to the hardware design. For instance if two units could not be used at the same time.

It was already said that VLIW architecture puts more complexity in the compiler that any other architecture. The main reason for this has to do with the instruction schedule. For a correct schedule the compiler must have a very detail knowledge about the ISA. For each instruction it must know the needed resources, the latency and possible conflicts. Another important thing is the capability to find instructions that can be bundled together that do not change the output of the program.

### 2.5.4 Intrinsics

Intrinsics are a kind of function that instead of being in a library is declared inside the compiler. Also they do not behave as a normal function, since they are usually inlined. The main purpose of intrinsics is related with exposing characteristics of the processor.

The EVP has a great number of intrinsics, most of them related with vectors. A few example are shown on Listing 2.27.

```
v3 = evp_vadd_16(v1, v2);
vadd16 vrD, vrA, vrB
```

Listing 2.27: Example of EVP intrinsics.

# Chapter 3

# Problem Description and Approach

## 3.1 Problem description

After the reader has been presented with the main concepts that will be important for him to understand the problem, it is time to describe the problem in a more detailed way.

It was mentioned before that the EVP is used in embedded devices and because of that, must abide to some of the design constrains inherent to embedded devices. One of the constrains that the EVP must comply is the power consumption. It is necessary to be as power efficient as possible. One way of improving power efficiency is through optimized code, allowing the processor to return to a power saving state more quicker.

The team that developed the EVP wanted to check the benefits that LTO compilation can bring, and check the possibility to adopt a newer linker.

### 3.1.1 Replace the linker

The main reason behind the migration of the linker has to do with the usage of LTO. Even though is not necessary a new linker, it is always a great asset to have a linker that supports linker plugins. Through this functionality the linker can send information that is only available during link time to the compiler. This information, can be the LTO information present on libraries, or information about the visibility of symbols.

Also, the current linker was created with another processor in mind and some of the design choices appear to be made to make the implementation for that processor easier. One example is the way the linker deals with memory pages, when EVP does not have paged memory.

### 3.1.2 LTO compilation

After a linker was chosen and ported, the compiler must be changed to allow for LTO compilation.

After all the pieces are in the correct places it is time to measure the performance changes introduced by the LTO compilation. To measure these changes a test suite called BlissTest will be used. This test suite uses certain Wideband Code Division Multiple Access (WCDMA) use cases. It is an important test suite because allow to check the EVP performance using code similar to the one executed in the real world. The metrics used in this comparison will be the number of cycles spent on execution and the executable size. The EVP simulator will be used to get the number of cycles used to execute the program.

## 3.2 Approach

### 3.2.1 Linker

The new linker would need to support LTO, a feature that at this moment it is only included on the two linkers present on the binutils package.

The `gold` linker was created to have better performance when linking ELF files and C++ applications. The advantages of `gold` is the more recent code base, it is written in C++, and has a better performance that `ld`, however it is still considered in beta stage.

The `ld` is the default linker in the binutils package. It is a mature tool and relies on `libbfd` to manipulate the object files. The fact that it is based on `libbfd` means that after the EVP support has been added, all tool except the linker and assembler, became available with little or even no additional work.

After some consideration `ld` was the chosen linker. The reasons to chose ld instead of gold, were the code maturity, and the extra tools available after `libbfd` support EVP.

To add EVP support in `ld`, it is necessary to add the EVP support to `libbfd` and then change `ld` to make use of the new `libbfd` target. A more detail explanation on how to do these two steps can be found on section 4.1.

### 3.2.2 LTO enabling

The compiler has a new part that is responsible for emitting the LTO information, that is target independent. However the information that is emitted can contain target specific information and this one must be emitted by the target backend. More specifically the intrinsics must be emitted by the backend.

Another necessary change has to do with the compilation process. In Figure 2.3 it is shown that the LTO compilation has more steps. When the GCC driver is used all the extra steps are hidden, and from the user perspective the compilation process remains the same. Unfortunately, the GCC driver is not used. Instead, it is used one developed in house which is not aware of necessary steps to compile with LTO. So it can not be used to achieve a complete LTO compilation. For that the last three steps must manually executed, i.e. second compilation, assembly, and linking.

### 3.2.3 Performance Measures

Measuring the performance differences between a normal compilation and a LTO one is important. To measure these changes it was chosen the BLISS test platform, that executes certain WCDMA use cases. This code is a great test case because it has strong resemblance with the code that is executed in EVP. Two executables are create one with a normal compiler the other with a LTO compiler. Both executables were then executed in a EVP simulator that gives profiling information. With this profiling information the number of cycles elapsed during execution can be calculated and this value is one of the metrics used to check performance. Another performance metric is the executable size.

# Chapter 4

# Implementation

This chapter describes the implementation of the `libbfd` backend, the `ld` support for EVP, and the enabling of LTO in GCC in detail. The chapter follows the order which the work was done. First it will explain how the backend for `libbfd` was made. Then it deails the work done in `ld` to adapt it to use the backend previously created and finally reports on the work done to enable LTO compilation for EVP.

## 4.1 Creation of a libbfd backend

A backend in `libbfd` is responsible for the translation between the file format to the internal structures used by `libbfd`. Moreover the backend also handles the relocations. To do its job the backend needs information about the target. This information is stored in various data structures and each of them keeps a specific information, e.g relocation and architecture information. The data structures and their meanings will be presented in this section. Note that not all data structures that can be used in a backend will be presented, namely the one used for functionalities not need in EVP.

### 4.1.1 Architecture description

The first step is to define the processor architecture. This is done through the use of the structure present in Listing 4.1. In particular lines 6–9 are used by `libbfd` to identify the processor, `arch` represents the architecture of the processor and `mach` is the processor model. In this case VD32042 is the model of a processor that implements the EVP architecture. The fields on lines 12 and 13 are used to internally to check the compatibility between models and to find an architecture by name. The remaining fields store generic information about the processor architecture, such as the word size, the address space, the minimum information unit or byte and the alignment used by the sections (lines 3, 4, 5 and 10).

For each processor model one `bfd_arch_info_type` should be declared and filled with the correct values for that model. In the case of EVP only the fields related with the model change, but other architectures can have more change between models. One example of this is the AVR architecture where models have different memory spaces. Each of these declarations must be inside a bfd directory in a file called cpu-arch.c, where arch is the name of the architecture.

On the EVP case the file is called cpu-evp.c. A partial content of this file is shown on Listing 4.2.

```
typedef struct bfd_arch_info
{
  int bits_per_word;
  int bits_per_address;
  int bits_per_byte;
  enum bfd_architecture arch;
  unsigned long mach;
  const char *arch_name;
  const char *printable_name;
  unsigned int section_align_power;
  bfd_boolean the_default;
  const struct bfd_arch_info * (*compatible) (const struct bfd_arch_info
      *a, const struct bfd_arch_info *b);
  bfd_boolean (*scan) (const struct bfd_arch_info *, const char *);
  void *(*fill) (bfd_size_type count, bfd_boolean is_bigendian,
      bfd_boolean code);
  const struct bfd_arch_info *next;
}
bfd_arch_info_type;
```

Listing 4.1: bfd_arch_info definition.

```
const bfd_arch_info_type bfd_evp042_arch =
{
  16,                       /* bits in a word */
  32,                       /* bits in an address */
  8,                        /* bits in a byte */
  bfd_arch_evp,             /* architecture enum*/
  bfd_march_vd32042,        /* machine architecture */
  "Evp",                    /* architecture name */
  "Evp:VD32042",            /* architecture printable name */
  0,                        /* section aliment */
  FALSE,                    /* is the default arch? */
  bfd_default_compatible,   /* function that returns a compatible arch */
  bfd_default_scan,         /**/
  bfd_arch_default_fill,    /**/
  NULL                      /* next arch */
};
```

Listing 4.2: Architecture definition for EVP.

### 4.1.2 Relocations

In this step was created the definitions of how the relocations are calculated. This information is stored in the structure named `reloc_howto_stuct` and its definition is shown at Listing 4.3. The information present in this structure and their meaning is:

- **type:** is an internal value that is used to identify the relocation type.
- **rightshift:** stores the value of how many LSB should be drop. In some architectures the instructions must be in address multiple of $n$ in this situation the $n$ LSB will always be zero and can be dropped.
- **size:** the size to read and write from the object file, the values available are zero to read a byte, one to read two bytes and two that read four bytes.

- **bitsize:** the size in bits of the item to relocate. It is used to check for overflow.
- **pc_relative:** true if the relocation is relative to the instruction address, false otherwise.
- **bitpos:** used to shift the value to the correct position. For instance if the value is 8 bits and it should be in the MSB.
- **complain_on_overflow:** the type of overflow to check after the relocation is calculated.
- **(*special_function):** if non NULL, the relocation will be calculated by this function. Otherwise the generic function will be used.
- **name:** name of the relocation.
- **partial_inplace:** indicate where the *addend* value is stored, if in the object file or in the relocation structure.
- **src_mask:** is used to extract the relocation value from the data read from the object file.
- **dst_mask:** indicate which data to replace with the relocation value.
- **pcrel_offset:** if the instruction address is present in the data.

```
struct reloc_howto_struct
{
  unsigned int type;
  unsigned int rightshift;
  int size;
  unsigned int bitsize;
  bfd_boolean pc_relative;
  unsigned int bitpos;
  enum complain_overflow complain_on_overflow;
  bfd_reloc_status_type (*special_function) (bfd *, arelent *, struct
      bfd_symbol *, void *, asection *, bfd *, char **);
  char *name;
  bfd_boolean partial_inplace;
  bfd_vma src_mask;
  bfd_vma dst_mask;
  bfd_boolean pcrel_offset;
 };
```

Listing 4.3: reloc_howto_struct Definition.

A `reloc_howto_stuct` structure must be defined for each of the supported relocations. One example is shown on Listing 4.4. The more relevant values of the example on Listing 4.4 are:

- **rightshift:** on line 4, the value zero indicates that all the bits of the address are preserved.
- **size:** on line 5, the value two indicates that four bytes will be read from the object file.
- **complain_overflow:** on line 9, will verify if the relocation value overflows. The value is considered as a signed number one bit larger than bitfield.
- **src_mask:** on line 13, all the information read from the object file is discarded.
- **dst_mask:** on line 14, all bits of the relocation value will be written to the object file.

```
struct reloc_howto_struct =
{
  R_EVP_LSB32,
  0,
  2,
  32,
  FALSE,
  0,
  complain_overflow_bitfield,
  bfd_elf_evp_ov_reloc,
  "R_EVP_LSB32",
  TRUE,
  0,
  0xFFFFFFFF,
  FALSE
};
```

Listing 4.4: EVP reloc howto example

### 4.1.3 How a relocation is calculated

`libbfd` defines `bfd_perform_relocation` as a generic function used to resolve relocations. Inside this function if `special_function` is not `null` the `special_function` will be invoked. The `special_function` can be used to deal with specific cases that are not covered by the `bfd_perform_relocation`. After `special_function` returns and depending on the value the function `bfd_perform_relocation` will continue its execution or also return. This means that `special_function` does not necessary need to resolve the relocation, it can only change some of the values involved in the relocation calculation and let the `bfd_perform_relocation` do the rest of the work.

The `bfd_perform_relocation` could resolve the EVP relocations. However the usage of overlays on EVP brings the need to do additional work on relocation. The `bfd_perform_relocation` will use the load address of a section instead of the virtual address, this presents no problem when the values are the same. However this is not the case for overlays, an overlay section must have a different load address and virtual address. To resolve a relocation on a overlay section the `special_function` is used, the function will replace the load address by the virtual address and returns the execution to `bfd_perform_relocation` that will conclude the calculation.

If the `special_function` is ignored the algorithm to resolve a relocation is always similar. On Listing 4.5 is a simplified version of it.

The algorithm has two parts that may look difficult to understand, one on line 6 and the other on line 12. I will explain them in detail and show why they are important.

The first part on line 6 is responsible for drop unwanted bits and put the remaining one in the correct position. The drop of unwanted bits can occur in architectures that constrain memory access to addresses multiple of $2^n$.

In this situation the $n$ LSB will be zero. If the $n$ LSB are always the same they can be dropped, reducing the number of bits in the instruction necessary to store a memory address.

```
 1 //Calculate the correct value
 2 relocation = symbol.value;
 3 relocation += symbol.section.lma
 4 relocation += addend
 5
 6 //Put the value in the right position
 7 relocation >> right_shift
 8 relocation << bitpos
 9
10 //read and write the value
11 aux = read_from_object_file()
12 // save bits that not belong to the relocation
13 aux1 = aux & ~dst_mask
14 // gets the bit that belong to the relocation
15 aux = aux & src_mask
16 aux += relocation
17 // drop unwanted bits
18 aux = aux & dst_mask;
19 aux = aux | aux1
20 write_to_object_file(aux);
```

Listing 4.5: Pseudo code for generic function

Imagine a byte where R represents the position where the relocation value must be stored and another byte where V represents the relocation value, like the ones present on Listing 4.6 in line 1. Before apply the or operation to the two bytes V must be align with R, so the relocation value must be shifted left one time, to became the one on line 5. After apply the or operation it becomes the one on line 9.

```
 1 //Original values
 2 xxxRRRRx // byte read
 3 xxxxVVVV // relocation value
 4
 5 //Align bits
 6 xxxRRRRx // byte read
 7 xxxVVVVx // relocation value
 8
 9 //After the or operation
10 xxxVVVVx // result
```

Listing 4.6: Instrucion encoding

The other important step is on Lines 12–19 of Listing 4.5, replace only the correct bits and leave the other. Also the bits that will be replaced must be zero. An example of this operation can be seen on Listing 4.7. The `x` represent any value, and `V` represents the relocation value. The `dst_mask` indicates the bits where the relocation value will be stored and the `src_mask` indicates the bits where the *addend* is stored, if the *addend* is keep in the object file. The step on Lines 4–7 at Listing 4.7 is used to keep the bits that will not be replaced and zeroing the other bits. On the step on Lines 9–12 at Listing 4.7 the addend value is read. This step is only relevant in the situations where the addend value is not present in the relocation structure, for instance in the case of `Elf32_Rel`. On the situation where the addend value is stored in the object file the `src_mask` should have the bit pattern to retrieve only the correct bits, and in the case where it is stored in the structure the value should be zero. The step on Lines 14–17 at Listing 4.7 is used to remove unneeded bits from the relocation value. In the last step the value present in the object file and the relocation value are joined to create the value that will be written.

```
00001111 // dst_mask
00000000 // src_mask

// Save bits
xxxxxxxx // byte read
11110000 //~dst_mask
xxxx0000 // bits to keep

// Get addend value
xxxxxxxx // byte read
00000000 // src_mask
00000000 // addend value

//Drop unwanted bits on the relocation value
xxxxVVVV // relocation value
00001111 // dst_mask
0000VVVV // striped relocation value

//Align bits
0000VVVV // striped relocation value
00001111 // bits to keep
xxxxVVVV // value to store
```

Listing 4.7: Steps to obtain the value to write.

### 4.1.4 File Operations

To handle the multiple object file formats, `libbfd` has a structure similar to the one presented on Figure 4.1. It uses a layer scheme where in each layer the process is specialized. The object file manipulation layer is responsible for translate the object file format to the internal `libbfd` structures. On the target specific layer is responsible for doing the target specific things, for instance decode the file header, or doing the relocations.

The target specific layer is implemented through the use of function pointers. The idea is that some functions of `libbfd` have code similar to the one presented on Listing 4.8. The code will run the `elf_info_to_howto` function if it is defined. On Listing 4.9 can be seen the way to create the functions and to define them.

In this work the target will manipulate ELF files. `libbfd` already has a ELF manipulation layer, so it only needs the target specific code.

The target code for `libbfd` is in the bfd directory in files with the name format-target.c, where format is the object file format name and the target is the processor name e.g. elf-evp.c contains the specific code for EVP manipulate ELF files. The minimum information contained in this file is:

- a list of the relocations, using the `struct reloc_howto_struct` shown on Listing 4.3
- a function that translate between BFD relocation type and processor relocation type. This is done using the `elf_info_to_howto` function pointer.

After adding this information the `libbfd` is ready to be used by the linker. It is able to do relocations and handle the generic information in the ELF files. However it does not know the meaning of the flags in the ELF header and how to define some EVP specific debug information. These functionalities will be added using the function pointers used to extend the object file manipulation.

A complete list of the function pointers available in a ELF backend can be seen in the `struct elf_backend_data` on the elf-bfd.h file.

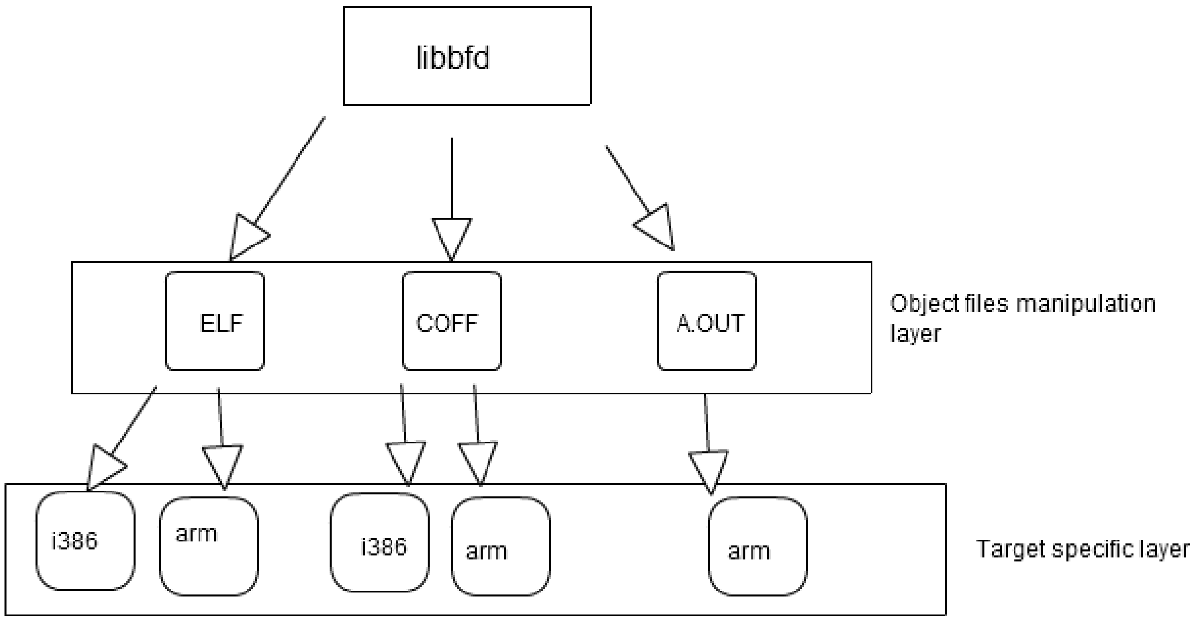

Figure 4.1: Architecture of the libbfd backend.

During this work only a few of them were used, and some examples and their purpose are:

- `elf_backend_begin_write_processing`. It is executed before any calculation for the final file is done, i.e. the the section address will be calculated. It can be used to change things that have any influence on final address, for instance add new sections.

- `elf_backend_modify_segment_map`. It is used to make the final changes on segment maps. Segment maps is the equivalent of program headers on ELF.

- `elf_backend_final_write_processing`. It is done right before writing the object file. The only changes that can be done are in the file header, e.g. the flags field.

- `elf_info_to_howto`. Convert between EVP relocation to BFD relocations based on relocation type.

```
if(elf_info_to_howto != NULL)
{
  (*elf_info_to_howto)(...)
}
```

Listing 4.8: Test for target specific functions.

```
#define elf_info_to_howto evp_info_to_howto

void evp_info_to_howto(bfd *abfd , arelent *cache_ptr, Elf_Internal_Rela
    *dst)
{
  unsigned r_type;
  r_type = ELF32_R_TYPE(dst->r_info);
  cache_ptr->howto = evp_reloc_lookup(r_type);
}
```

Listing 4.9: How to define a target specific function.

### 4.1.5 Debug Sections

The linker is also used to create a section that contains debug information to be used by the simulator. The content of this section is related with overlays, containing the name, load position and run position for each overlay.

The information on these sections is necessary for the simulator. The EVP itself do not handle the overlays. That is done by the program running in the general purpose processor where the EVP is attached. Since the simulator only simulates the EVP, it must have a mechanism to handle the overlays. So the debug section is used to inform the simulator about the name, load and run addresses of the overlays.

#### .debug_info_groups

The debug information is stored using the DWARF 2 format, which has been extended to support memory spaces, address range and overlays. The content of these extensions is shown in Listings 4.10, 4.11 and 4.14.

```
size      DW_FORM_DATA4
tag       DW_FORM_DATA2
name      DW_FORM_STRING
```

Listing 4.10: Memory space.

```
size      DW_FORM_DATA4
tag       DW_FORM_DATA2
parent    DW_FORM_REF
begin     DW_FORM_ADDR
end       DW_FORM_ADDR
```

Listing 4.11: Address range.

```
size      DW_FORM_DATA4
tag       DW_FORM_DATA2
parent    DW_FORM_REF
name      DW_FORM_STRING
source    DW_FORM_REF
dest      DW_FORM_REF
```

Listing 4.12: Overlay range.

Every DIE in the DWARF format has the `size` and `tag` in common. These field are used to identify the DIE and its contents. The `size` is a 4 bytes field that contains the size of the DIE, excluding the 4 bytes of the `size` field. The `tag` is a 2 bytes field that identifies the DIE type.

A program segment is represented using a memory space and an address range. The memory space contains the program segment name and the address range is used to store the begin and end addresses for that memory.

On Listing 4.13 can be seen an example of a program segment. The program segment has the name `PRG` and extends though all the memory.

```
Memory Space
size  6
tag   TAG_GRP_MEM
name  PRG

Address Range
size
tag     TAG_GRP_ARANGE
parent  Memory space PRG
begin   0x00000000
end     0xFFFFFFFF
```

Listing 4.13: Example of a program segment.

An overlay is represented using a overlay range, a memory space and a address range. The overlay range contains the name, the source and destination locations, the memory space which stores in what memory the overlay is located, and the address range, used to store the memory address of the overlay. On the example shown on Listing 4.14 the overlay `ov1` will be executed in the `PRG` memory on the addresses between 0x21000 and 0x2FFFF, and it is located in the `EXT` memory between 0x1000 and 0xFFFF addresses.

```
Memory Space PRG
name  PRG

Memory Space EXT
name  EXT

Overlay Range
parent  Memory Space PRG
name    ov1
source  Address range origin
dest    Address range destin

Address Range origin
parent  Memory space EXT
begin   0x00001000
end     0x0000FFFF

Address Range destin
parent  Memory space PRG
begin   0x00021000
end     0x0002FFFF
```

Listing 4.14: Example of a overlay.

## 4.2 Changes in ld

After implement the EVP support in `libbfd` it is necessary to inform the `ld` about the existence of a new backend, moreover to maintain compatibility with the current build scripts, a new command line option will be added.

Part of the behavior of `ld` is configurable through the use of configuration files present in the directory `emultempl` inside the `ld` directory. When compiling `ld`, one of these files will be used to generate some of the `ld` code. Some of the behaviors that are possible to do with this method are, add new command line options, do some processing after all input files where opened.

The changes done in EVP where, add command line options to maintain the compatibility with vlink and verify if all the input files have the same version.

The EVP has a few different models that have architectural differences between them that make them incompatible. The incompatibilities have to do with the resources present in some version that are not available in others, i.e. some sequences of instructions are valid in some version but are invalid in others. Currently vlink has two methods to decide against which version the linking is done. First it can receive as a command line option the version. Second derives, it from the input files.

To ensure that the same behaviour is maintained there is the need to add a new command line option and a version check for all the input files.

### 4.2.1 New command option

The code necessary to add new command line options is shown at Listing 4.15.

```
PARSE_AND_LIST_PROLOGUE='#define␣OPTION_TARGET␣301'

PARSE_AND_LIST_LONGOPTS='{"tgt",␣required_argument,␣NULL,␣OPTION_TARGET},'

PARSE_AND_LIST_OPTIONS='fprintf␣(file,␣_("␣␣-tgt␣Used␣for␣specify␣the␣EVP␣machine.␣Possible␣value:␣vd32040,␣vd32041,␣vd32042,␣vd32043\n"));'

PARSE_AND_LIST_ARGS_CASES='
␣␣␣␣case␣OPTION_TARGET:
␣␣␣␣␣␣target_name␣=␣optarg;
␣␣␣␣␣␣break;'
```

Listing 4.15: Example to add a command line option.

The code is composed of macros that will generate the necessary code. An explanation of the macros used in this situation is:

- **PARSE_AND_LIST_PROLOGUE** will be inserted outside a function. It can be used to create defines or global variables. In this case it is used to create the macro used to identify the target option.
- **PARSE_AND_LIST_LONGOPT** is used to define the parameters list. The values needed in this macro are, the name of the parameter, if have arguments, and a numeric identifier.
- **PARSE_AND_LIST_OPTIONS** will be printed at the end of the help message. It should explain the usage for the added parameter.
- **PARSE_AND_LIST_ARGS_CASES** must contain a case to each of the new command options. Each case must contain the code to handle that option.

### 4.2.2 Version Verification

To guarantee that all files are in the same format two approaches were followed. If the version was set using a command line option all the file must have an equal version, otherwise test if all the files have the same version.

The validation code can be seen at Listing 4.16.

```
evp_after_open(void)
{
  gld${EMULATION_NAME}_after_open();
  //Check each input file for the same mach value.
  bfd *abfd = link_info.input_bfds;
  bfd_boolean fail = FALSE;
  for(;abfd != NULL; abfd = abfd->link_next)
  {
    //get elf flags
    unsigned long e_flags = (elf_elfheader(abfd)->e_flags & 0xFF000000);
    if(e_flags != mach)
    {
      /* Different formats, abort link. */
      einfo("%X%P:␣Invalid␣file␣formats:␣file␣%B␣is␣not␣in␣%s␣format\n",
         abfd, target_name);
      fail = TRUE;
    }
  }
```

Listing 4.16: Code to validate the version of the input files.

The `evp_after_open` is invoked after all input files were open. This function will iterate through all files. If a file has a different version than the previously defined, an error message will be printed, this procedure is repeated for each file. After all the files were tested if one or more errors were found the linking process is aborted.

## 4.3 Small changes in Binutils package

It was already said that almost all tools in `binutils` package depend on `libbfd`, and for these tools no change is necessary to support the EVP target. However the `readelf` does not depend on `libbfd` and is a useful tool for inspect ELF files.

The `readelf` is a program that was created to inspect only ELF files, and because of that can show details about the format that are extremely helpful to debug `libbfd`.

Since the ELF file does not have many differences across different processors, the necessary modification were quite small. The changes were mainly related with target information, e.g. the target name, the meaning of the flags. Through the code there are some functions that are responsible for print the information based on the input. One example is the function `get_machine_name` that receives as an input the machine code present in the ELF header, and prints the machine name. This function is a huge switch case, were each case corresponds to one of the known architectures.

## 4.4 LTO enablement in gcc

The LTO compilation is mainly target independent, however there are a couple of functions that the targets must implement.

The first one is the macro ASM_OUTPUT_COMMON that is used to create a common label that has a name and a size. This is used to create a symbol in the assembly files, which is then used to mark the files that have LTO information.[10]

The second one is the ability to emit the intermediate representation of the intrinsics by the target backend. The LTO implementation defines a hook named TARGET_BUILTIN_DECL that must be defined if the target has any intrinsic. The function should return the intrinsic tree code. The prototype of the function can be seen at Listing 4.17.[11]

```
#define TARGET_BUILTIN_DECL evp_builtin_decl

tree evp_builtin_decl(unsigned code, bool initialize_p);
```

Listing 4.17: Intrinsic Hook.

GCC is a software package that is composed by multiple executables. Each of these executables has a well defined scope. For instance the executabe cc1 is the C compiler. When GCC is built with LTO, a new executable is created named lto1, that is responsible for the LTO step. During the link phase of lto1 a couple of duplicate references appear. The affected code is related with some of the pragma definitions and since it is not essential code the approach was disable it for the LTO compiler. This approach was the quickest one, but can have performance penalties.

# Chapter 5

# Discussion of results.

In this chapter the performance improvements from using `ld`, and LTO will be presented and discussed. In the beginning it will be present how the result were obtained. After that the performance result will be discussed, the first one only with the `ld` and after the ones obtained using LTO.

## 5.1 Linker Validation

To validate the tool chain it was used a compiler test suit called SuperTest. SuperTest is a commercial suite for compiler validation that has tests to validated the C language correctness and also has tests to validate all EVP intrinsics. The last ones were created by the EVP development team.

In order to think about replace the linker, the new one must be able to correctly link all the test present in SuperTest. After that was achieved, the linker has considered to be on the same state as the current one and ready to start some benchmarks.

## 5.2 How the data was collected

The EVP SDK has a simulator, that allow the execution and debugging of EVP code. It is a cycle-accurate simulator, that means that the simulator tries to replicate the processor execution on a cycle based. This type of simulator is used to accurately benchmark a processor.

The simulator gives profiling information. The important information needed for this document is the number of cycles spend by function. With this information is possible to measure the performance changes introduced by the different tool chains. Moreover since there is also information for each function, it is easier to find the source of the performance changes.

Some more of the information available is, the function size, Instruction-Level parallelism (ILP), the instructions executed in each functional unit, and information about overlays.

### 5.2.1 BLISS test

BLISS test is a test platform that execute certain WCDMA use cases. The test platform is composed of a unique file. This file has all the functions necessary to read the input values, run the test, output the values and comparing the output values with the reference ones.

This test platform was chosen because it is partially a code that will run on EVP in the real world. So it can represent more accurately the performance changes that the use of LTO presents. Also it was an important test for the linker. This was the biggest executable created by ld.

## 5.3 Linker performance changes

After linking the same source code using both linkers, the executable size was measured and it was executed to validate if the link process had created a valid executable.

The values for the executable size can be seen at Table 5.1. The executable created by ld is smaller than the one created by the vlink. This difference was unexpected, since both files were created with the same object files and both files have successful executions, so it was expected that the file sizes were similar. After some investigation it became apparent that the difference was in the number os symbols present in each executable. For instance in the executable used in this test the number of symbols was 1010 in `vlink`, and 448 in `ld`. After looking for the missing symbols, it became noticeable that the executable created by `vlink` had duplicated symbols, i.e. symbols that have the same name, value and are in the same section, that ld remove. A symbol in an ELF is represented by the structure show at Listing 5.1, the size of this structure is 16 bytes. The executable from `vlink` has more 562 symbols, and each symbol has 16 bytes size, this represents a total 8992 bytes extra in the executable created by `vlink`. The remove of duplicate symbols is the biggest contributor to the differences in sizes. Another small one is related with the program segments, the `vlink` always creates 7 program segments even if they are not used and are independent of the ones defined in the linker scripts. In the `ld` case the program segments are created according the linker script instructions.

```
typedef struct {
  Elf32_Word    st_name;
  Elf32_Addr    st_value;
  Elf32_Word    st_size;
  unsigned char st_info;
  unsigned char st_other;
  Elf32_Half    st_shndx;
} Elf32_Sym;
```

Listing 5.1: Symbol representation on a ELF file.

| | vlink | ld |
|---|---|---|
| Size | 573Kb | 564Kb |

Table 5.1: Size for WCDMA executables

When the executables were run the number of cycles from the executable linked with `ld` was a surprise. In every test case it takes less 13 cycles to execute that the `vlink` executable. This was odd, since the linker does not change code and therefore no differences are expected in this area. From looking at the number of cycles that are spent in each function, the `memmove` function is the one that has different execution cycles. However the number of times the function is invoked is different for almost all test cases, but there is a difference of 13 cycles in every test case. Unfortunately the source of this improvement was not found. The best guess is that it must be something at the start of execution, and is only executed once. This guess is based on the fact that the improvement is constant no matter the number of invocations of the function.

## 5.4 LTO performance

To test the LTO performance changes, three executables were created in order to measure the size and execution time changes. The non LTO executable was built using the normal tool chain. Two LTO executables were built, the code was compiled with the compiler with LTO support, and were linked with `vlink` and `ld`.

On Table 5.2 can be seen the size of the executables. Looking at the Table 5.2 two things come up: the LTO improves the executable size in 9 Kb when `vlink` is used and 13 Kb when is linked with `ld`. The difference between the two linkers is expected by the reasons described in section 5.3. However in this case they are smaller, since the number of symbols present in the `vlink` executable has 667 symbols while in the ld has 448. It was expected that the LTO executable should have a smaller size than the non LTO but it still is an impressive improvement.

| | vlink | lto vlink | lto ld |
|---|---|---|---|
| Size | 573Kb | 564Kb | 560Kb |

Table 5.2: Size for WCDMA executables

It is time to look at the performance changes that the LTO executables have. On Table 5.3 can be seen the number of cycles spent in two tests. In this table the execution values from the LTO executable linked with `vlink` were omitted, because they are similar to the values from the `ld` executable. Only two test cases are presented. The ones presented are the more interesting ones, the one that had the biggest performance improvement and the one that had the biggest performance penalty. On column `vlink` contains the values for the non LTO executable, and on the lto column are the values for the LTO executable, $\delta$ cycles is the difference between the cycles count of `vlink` and `ld` a negative number represents a decrease in performance and a positive number a performance increase, and in the last one is the performance difference in a percentage value, a value less that a 100 represents a decrease in performance and a value above represents an increase.

| Test Number | vlink | lto | $\delta cycle$ | |
|---:|---:|---:|---:|---:|
| 102008 | 85175985 | 85178977 | $-2992$ | 99.996% |
| 104000 | 1225665 | 1225110 | 555 | 100.045% |

Table 5.3: Cycles for WCDMA executables.

Now that the information about the execution times was collected it is time to try to understand the reasons behind this performance changes. Unfortunately it is not a trivial task to find the source of these performance changes. The best way to do it is inspecting the GIMPLE produced by each of the GCC steps.

When the flag `-fdump-tree-all` is passed to GCC, this will output the GIMPLE at various stages of the compilation.[12] To find the source of the performance changes, it is necessary to have the GIMPLE output of the two compilation processes, the reference one and the one with performance changes. The output of each step must be compared until they have differences. This allows to identify the step where the changes began to appear, and the starting point to investigate the changes. However the point where the outputs start to diverge do not mean that it is the responsible for the performance changes, that could be introduced by the following steps.

The 102008 test case uses the memory qualifiers pragma and this pragma belongs to the group of pragmas that were disabled in order to enable LTO.

The memory qualifiers are a way to tell the compiler that pointers that have different memory qualifiers, will never point to the same memory address. This is important because it tells the compiler that these pointers will never have a read-after-write dependency.

On Listings 5.3 and 5.4 are shown the GIMPLE outputs generated by the normal compiler and by the LTO compiler respectively and the original code is show on Listing 5.2. In the input code a pointer to vector `v1` is created and is associated with the memory qualifier `M1`, and the pointer is incremented. In the incorrect output a new variable is created to store the value of `v1` and will be used in the input of `evp_add_ptr`. The creation of this new variable creates one more attribution every time a pointer associated with a memory qualifier is used.

In this test case the function that has the biggest performance penalty is `Alg_Rce`. The function contains two loops. The outer loop iterates at least one time and has a maximum of five iterations, and the inner loop a minimum of one iteration and a maximum of three iteration. The inner loop has four situations where memory qualifiers are used and that gives more four cycles per iteration. On the outer loop the use of memory qualifiers creates an additional 6 cycles per iteration. In the best case, both loops have only one iteration and the function spends 10 more cycles in execution. In the worst case, where both loops take the maximum number of iterations, the function spends 102 more cycles executing. This function is called 60 times and needs more 2340 cycle to execute on the LTO executable, that gives a mean of 39 cycles per invocation. Since the mean value of cycles per invocation is inside the minimum and maximum values for the two loops and no other significant difference was detected on the final code for this function, it is safe to say that the memory qualifiers are the responsible for the performance penalty in this function.

```
v_t __M1 *v1;
v1 = (__M1 v_t*)evp_add_ptr(v1, 1);
```

Listing 5.2: Input code

```
struct v1_t * v1;
struct v1_t * D.19285;

D.19285_3 = _evp_add_ptr (v1, 1);
v1_4 = (struct v1_t *) D.19285_3;
```

Listing 5.3: Correct output GIMPLE

```
struct v1_t * v1;
struct v1_t * D.19284;
struct v1_t * D.19285;

D.19284_2 = (const void *) v1;
D.19285_3 = _evp_add_ptr (D.19284_2, 1);
v1_4 = (struct v1_t *) D.19285_3;
```

Listing 5.4: Incorrect output GIMPLE

The test case 104000 is the other interesting one, mainly because it is the one with the biggest performance improvement. This test case has an improvement of 555 cycles, and the function that has the better improvement goes from 177 cycles to 84 cycles per invocation.

The reason behind this big improvement in this function is the use of hardware loops, and an optimization that goes wrong. The hardware loops is a feature where the processor controls the loop execution. The processor must know the begin and end address of the loop and the number of iterations. The advantage of the hardware loops is the removal of the control code for the loop, and in the case of EVP the delay slots present after the jump instruction. For instance, in the test case 104000, by converting the loop to hardware loops, two jump instruction and receptive delay slots are removed.

To enable the hardware loops it is necessary to know certain characteristics of the loop, namely to know the first and last instruction of the loop but more important it is necessary to know the number of iterations. For instance, the for loop on Listing 5.6 is a candidate to became a hardware loop, the number of iterations is known before the cycle begins and it is constant during its execution. However the while loop also present on Listing 5.6 is not a candidate to a hardware loop, because the number of iterations is not known.

The original code is similar to the one present on Listing 5.5. It has two `for` loops were the number of iterations is calculated before the loops begin. Also both `i` and `j` are never used outside the loop control statements. During the ivopts optimization pass the non LTO compiler will transform the code present on Listing 5.5 on the code present on Listing 5.6 and by doing this transformation the number of iterations is lost and the hardware loops can not be used. More important, the loop control code generated is awful, it is constant during the loop iterations, but it is recalculated in every iteration.

```
int *array;
for(i = 0; i < outerLoopIter; i++)
{
  ...
  for(j = 0; j < innerLoopIter; j++)
  {
    ...
    array++;
  }
  ...
}
```

Listing 5.5: Original loops

```
int *array;

int *originalArray = array;
for(i = 0; i < outerLoopIter; i++)
{
  ...
  int *stop = innerLoopIter * sizeof(int) + originalArray;
  while(array != stop)
  {
    ...
    array++;
  }
  ...
}
```

Listing 5.6: After loop transformation

# Chapter 6

# Conclusions

The main goals defined for this dissertation were achieved. The `ld` linker present in binutils package was ported to EVP target. The new liker was able to produce correct executables in all the test cases. It was also used during the LTO tests also with correct results. It is my believe that the linker is ready to replace the current linker. However, it is not possible to say that its behaviour is completely identical to the current linker. The current linker has a well tested code base and it is possible that in some corner cases its behaviour is different than the one in `ld`. To ensure that these cases are caught, for the first times that `ld` is used, the executable, should be well tested to ensure that the linker maintains the correct behaviour of the executable.

The LTO performance evaluation results show that some performance improvements can be expected from this technique. However it needs more work. The code base used to test LTO is not one of the best ones, since most of the files are self contained, which means that they do not have external dependencies that can be fulfilled by the LTO compilation. Nonetheless LTO was able to show some performance improvements and it is expected that these could be bigger in a code base that has more dependencies between files.

Another important thing to have in mind, regardless LTO, is the fact that the compiler used was GCC 4.5.0, which contains the first implementation of LTO in GCC and it is a 4 years old version. Since it was the first version it contains bugs in the LTO compilation. In some situations the bugs present in this version make the compilation impossible.[13]

## 6.1 Future Work

In order o improve the benefits of the LTO compilation the following points should be addressed. These points should bring considerable performance improvements and make the use of LTO more simpler.

- **enable the disable code:** to enable LTO some code needed to be commented. In order to not suffer from performance penalties, the commented code must be activated again.
- **upgrade the compiler:** it was already mentioned that the GCC version used is 4 years old, and it was in this version that the LTO support was added. In the last 4 years there was much improvements in the LTO. Namely the code receive optimizations that make it faster and reduced the memory footage. It also received new features, such as the ability to emit only the LTO information and the ability to compiler larger program in a parallel way. Outside LTO was also a lot of improvements, new optimization techniques and improvements to the existing ones.
- **integrate LTO step in `vcom`:** `vcom` is the compilation driver used. A compilation driver is a program responsible to invoke the correct programs to compile some program. `vcom` is develop in

house, and so far it doesn't support LTO compilation. Adding such support would make the process automatic, making it more accessible to other developers.

- **create fat libraries:** a fat library contains both assembly code and the LTO information. Using the linker plugins the information present in this libraries can be used in the compilation, creating a bigger scope where the optimizations can be applied.

# Bibliography

[1] Mirshahab Vahedi. Iterative instruction scheduling for a vliw processor. Master's thesis, Delft University of Technology, Delft, Netherlands, 2013.

[2] Steven S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[3] D. Salomon. *Assemblers and loaders.* Computers and their applications. Ellis Horwood, 1992.

[4] J.R. Levine. *Linkers and Loaders.* Operating Systems Series. Morgan Kaufmann, 2000.

[5] T. I. S. Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2, 1995.

[6] ld linker scripts. `https://sourceware.org/binutils/docs/ld/Scripts.html`. Accessed: 2014-06-01.

[7] Design overview. `https://gcc.gnu.org/onlinedocs/gccint/LTO-Overview.html`. Accessed: 2014-06-01.

[8] Michael J. Eager. Introduction to the dwarf debugging format, 2007.

[9] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2002.

[10] Output of uninitialized variables. `http://gcc.gnu.org/onlinedocs/gccint/Uninitialized-Data.html`. Accessed: 2014-06-01.

[11] Miscellaneous parameters. `http://gcc.gnu.org/onlinedocs/gccint/Misc.html`. Accessed: 2014-06-01.

[12] Options for debugging your program or gcc. `http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html`. Accessed: 2014-06-01.

[13] Taras Glek and Jan Hubicka. Optimizing real world applications with gcc link time optimization. *arXiv preprint arXiv:1010.2196*, 2010.