A framework for rapid development of dynamic binary translators

(HS-IKI-EA-04-102)

David Holm (`dholm@gentoo.org`)
School of Humanities and Informatics
University of Skövde, Box 408
S-54128 Skövde, SWEDEN

Final Year Project in Computer Science, Spring 2004.
Supervisor: Henrik Grimm

A framework for rapid development of dynamic binary translators

Submitted by David Holm to Högskolan Skövde as a dissertation for the degree B.Sc., in the Department of Computer Science.

June 6, 2004

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.

Signed: ______________________________

A framework for rapid development of dynamic binary translators

David Holm (dholm@gentoo.org)

Abstract

Binary recompilation and translation play an important role in computer systems today. It is used by systems such as Java and .NET, and system emulators like VMWare and VirtualPC. A dynamic binary translator have several things in common with a regular compiler but as they usually have to translate code in real-time several constraints have to be made, especially when it comes to making code optimisations.

Designing a dynamic recompiler is a complex process that involves repetitive tasks. Translation tables have to be constructed for the source architecture which contains the data necessary to translate each instruction into binary code that can be executed on the target architecture. This report presents a method that allows a developer to specify how the source and target architectures work using a set of scripting languages. The purpose of these languages is to relocate the repetitive tasks to computer software, so that they do not have to be performed manually by programmers. At the end of the report a simple benchmark is used to evaluate the performance of a basic IA32 emulator running on a PowerPC target that have been implemented using the system described here. The results of the benchmark is compared to the results of running the same benchmark on other, existing, emulators in order to show that the system presented here can compete with the existing methods used today.

Several ongoing research projects are looking into ways of designing binary translators. Most of these projects focus on ways of optimising code in real-time and how to solve the problems related to binary translation, such as handling self-modifying code.

Keywords: Emulation, Binary Translation, Dynamic Recompilation, JIT-compiler

Acknowledgements

I would like to thank a number of people who has provided invaluable support throughout this project.

First of all I would like to extend my gratitude to my supervisor, Henrik Grimm, for providing invaluable feed-back and support throughout the project. Without his continuous dedication and interest in this project I would have overlooked numerous flaws in my design.

I would also like to thank Bill Buck and Raquel Velasco of Genesi S.á.r.l (**URL:** *http://www.genesi.lu*). They provided me with the idea of investigating how an efficient PC-emulator could be implemented on PowerPC. This idea then grew into the larger and more general project described here.

Contents

1 Introduction

The world of desktop computers is today dominated by one actor, Microsoft, and their platform Windows. Even though their expected market share is over 90% on the desktop, alternatives such as GNU/Linux and Apple's MacOS X are slowly starting to gain ground (Dalrymple 2003, Gulker 2003). The server market has already seen an explosion of GNU/Linux based servers as both pricing and availability of good software gives the competitors a tough challenge (Jaques 2003). As the alternative operating systems are becoming more popular the possibility of choosing an alternative hardware platform increases. Microsoft Windows exclusively runs on Intel hardware, except for a small excursion into the realm of some alternative hardware platforms made with Windows NT 4. MacOS X runs on PowerPC hardware and GNU/Linux is supported on numerous architectures, including the Intel IA32 and IA64 as well as the Motorola PowerPC. IA32 stands for "Intel Architecture 32" which has 32-bit long registers and IA64, also known as Itanium, has 64-bit long registers (*IA-32 Intel Architecture Software Developer's Manual* 2001*a*).

Since Microsoft Windows is the most widespread desktop operating system, it is only natural that it has the largest number of supported commercial applications. In order to ease user transition to other platforms, a number of applications are being developed that make it possible to run Windows applications on other platforms and architectures. Some open-source solutions include Darwine, which intend to run applications for Microsoft Windows on MacOS X (White 2004), and QEMU, which is a new project that emulates a number of different processors (Bellard 2004). There is also an open-source emulator targeted at system developers with a built-in debugger and support for instrumentation. This emulator is called Bochs and is currently the most compatible of the three (Lawton 2004). With the exception of Darwine all of the above mentioned applications have in common that they have been designed to run on a number of different architectures. Because of this, they include only a minimal set of architecture-specific optimisations and none of them are PowerPC-specific. This, of course, has a large negative impact on emulation speed since very few assumptions about the host system can be made.

There is also a number of commercial emulators in existence. Virtual PC is available for Apple MacOS and Microsoft Windows (*Virtual PC* 2004). Simics is an emulator similar to Bochs in functionality as it is also intended as a tool for system developers, with the difference being that it can emulate several architectures and not just IA32 (Magnusson 2004). There is also an emulator, known as VMWare, which only emulates the system hardware and not the CPU (*VMWare* 2004), i.e., instructions get passed to the real CPU in the computer instead of being emulated. This has several advantages, one being that there is no need to develop a CPU emulation core. Another advantage is that bugs and undocumented features will be handled correctly, not to forget the fact that instructions are executed at a one-to-one ratio. VMWare only runs on IA32 which is the biggest disadvantage of this kind of emulation. It exists for both Microsoft Windows and GNU/Linux. Two benchmarks were recently conducted comparing the performance of VMWare to Virtual PC running Microsoft Windows 98SE and Microsoft Windows XP (Pietro 2004*a*, Pietro 2004*b*). The test results acquired during the first benchmark show that integer and floating-point calculations run at about 90% of the native system's speed. As for memory addressing, VMWare is almost as fast as the host, since it is using the host memory directly, whereas Virtual PC is utilising about 60% of the host's capacity. The second benchmark performed by Pietro (2004*a*) shows a significant increase in memory addressing performance over

VMWare by Virtual PC when using Microsoft Windows XP. Recent versions of Virtual PC are bundled with Windows XP which seems to indicate that it has been optimised for running this operating system.

This report describe a set of scripting languages that are intended to aid the development of JIT-compiling emulators. *JIT* stands for just-in-time and refers to the method of deferring translation of binary code until it is about to be executed. A simple IA32 emulator for PowerPC is developed using these scripting languages, and the performance of this emulator is evaluated and compared to some of the existing IA32 emulators. Other emulators for other systems can be developed using these languages as well, but this is not evaluated in this report.

The *DAISY* project by IBM use a similar approach to constructing emulators as the one described in this report (Ebcioglu & Altman 1996). They have chosen to use VLIW as an intermediate representation. *VLIW* is an abbreviation for "Very Long Instruction Word", which is a low-level specification architecture, similar to the microcode system used internally by Intel's processors, that can be used to construct new machine-code instructions not present in the original design (Pountain 2003). DAISY supports run-time translation of VLIW into a native instruction set in case the target architecture does not support VLIW. One problem with the VLIW approach is that since the specification language is on a lower level than assembly language it has a higher complexity level. Another problem is that the run-time translation approach used by DAISY when not running on VLIW hardware adds overhead to the emulator which reduces performance. The primary focus of the DAISY project is to build a system that will run on top of a processor. All applications including the operating system will run on DAISY. This report will present a method better suited to run as a user-level application of which the focus is on the hardware specification language. Several problems coexist between DAISY and this project, such as how to handle self-modifying code.

LLVM (Low Level Virtual Machine) is a research project that has created a compiler framework which produces binaries that are executed inside a virtual machine, which is similar to emulation in the sense that the virtual machine is an "invented" hardware which the binaries are executed on (Lattner & Adve 2004). This project has engineered a virtual instruction-set architecture (see section 5.2) that is designed to make it easier to optimise during run-time. Although LLVM and this project has separate goals, some of the techniques presented by the LLVM project are used here. LLVM define a set of virtual instructions known as *LLVA* of which a subset have been adopted here. The main reason for choosing LLVA is that it has already been tested on several hardware architectures by the LLVM project, these include Intel IA32 and Sun Sparc which represent the two most common classes of processors, CISC and RISC (these concepts are described in section 5.1).

2 Background

The purpose of this chapter is to introduce the reader to a number of important concepts used throughout the report.

2.1 Emulation

The purpose of *emulation* is to enable someone to run applications on another platform than it was initially designed for. The use most commonly associated with emulation is to make it possible to play computer games from game consoles on a standard desktop computer. In an industrial setting, emulation might be used to test out an embedded system without running it on the actual hardware. This has the advantage of giving the developer the possibility to stop the emulation at any given time and inspect the current system state at a low level, right down to the processor status registers. Another well known example of emulation, even though it is usually not thought of as such, is the execution of Java or .NET byte-code. Java and .NET are both based on instruction sets that have been developed to be general enough that they can easily be emulated on any given system. This is what is known as *byte-code* as opposed to binary code which only runs on the specific hardware it was intended for. The runtime system emulates this invented instruction set when executing a Java or .NET byte-code file.

The difference between simulating a system and emulating it is defined by Fayzullin (2000) as follows:

> *"Emulation is an attempt to imitate the internal design of a device. Simulation is an attempt to imitate functions of a device. For example, a program imitating the Pacman arcade hardware and running real Pacman ROM on it is an emulator. A Pacman game written for your computer but using graphics similar to the real arcade is a simulator."*

ROM stands for Read-Only Memory, and, in the case of emulators, a ROM is usually a memory dump of code that has been burnt into an integrated circuit during manufacturing. Video games sold on cartridges are stored this way.

Four methods of emulation are identified by Sharp (2001). These are described here, divided into three groups.

2.1.1 Interpretive

An *interpretive emulator* reads one instruction at a time from the binary to be emulated and translates it into something that can be run natively. This is the most widely used technique because it is the most simple method to implement. However, it provides the slowest emulation speed. This is how Bochs works (Lawton 2004).

2.1.2 Threaded code

Instead of interpreting instructions one by one and calling procedures to handle each instruction separately, a method known as *threaded code* can be used. A block of instructions to be emulated is reconstructed in memory by a block of function pointers to procedures emulating each individual instruction. Each translated block may be kept in a cache, so that the second time it is executed it does not have to be translated over again. Sharp (2001) identifies one problem when using threaded code in a system emulator which is that there is no convenient way to handle self-modifying code. If this case is to be handled, an emulator based on threaded code would also need a fall-back to one of the other methods presented here. Threaded code is used by QEMU (Bellard 2004).

2.1.3 Recompiling

There are two forms of recompilation used, one is dynamic and the other is static. *Dynamic recompilation* is also referred to as just-in-time (JIT) compilation. It recompiles the instructions of the emulated system into instructions that can be executed by the native system in real-time. This makes it possible to apply low-level optimisations to blocks of code while the system is running. A JIT-compiler only compiles a block of code the first time it is executed and stores the compiled block in memory. Subsequent calls to an already compiled block of code use the compiled block from memory instead of recompiling it again (Austin & Pawlan 1999).

Static recompilation works like dynamic recompilation but instead of doing the compilation during runtime the binary to be emulated is converted only once, before being executed. Emulation of this kind is used by compilers that generate native binary code from Java byte-code for instance. This technique is less common among system emulators as it would be a very complex operation to statically compile an entire operating system and the applications to be run on it.

Java has a standard for how to open windows and draw graphics etc, as where an entire hardware system usually only provides a low-level interface to the graphics hardware and leaves the rest to the operating system. However, if the emulator is limited to running applications from just one operating system, such as Microsoft Windows, it would be possible to use static recompilation provided that calls to the Windows API are translated into calls for the native platform API.

Recompilation is the most complicated method of emulation to implement as it does not just require knowledge of the system to be emulated but also in-depth knowledge of the host system. The advantage is that it can run emulated binaries much faster than an interpretive or threaded code emulator would. Virtual PC is using dynamic recompilation (*Virtual PC* 2004).

2.2 Trampolines

Trampolines are needed when compiling a block of code which is meant to be executable. A small block of initialisation instructions, known as a *prologue*, and finalisation instructions, known as an *epilogue*, have to be added to the block. There are a couple of things that need to be handled by the trampoline and they vary between different platforms. There is however one thing they all have in common and that is setting up a local stack space. The main purpose of the stack in the dynamically compiled blocks of code is to act as a kind of swap-space for local registers, for instance, when using registers as temporary variables (Sanseri 2000, Aho, Sethi & Ullman 1986). The trampoline also controls passing data to and from generated code blocks by function arguments and return values.

The name *trampoline* comes from the fact that it is used as a kind of springboard by the emulator in order to jump to a block of recompiled code. This can be done in a number of ways. When implementing an emulator in C, one convenient way of handling it is to set up a function pointer and have it point to whatever block that is to be executed next. By using a function pointer the C-compiler, when compiling the emulator, will add the instructions necessary for storing and restoring all registers during a jump. If this is not done the emulator will most likely stop working when returning from the trampoline call unless the trampoline handles this as well (Sanseri 2000, Sharp 2001).

2.3 Address translation

In modern computer systems it is rare to find a system which only runs one process at a time. Usually the notion of several processes running simultaneously exist, commonly referred to as *multitasking*. When more than one process is running, the question of security arises. A buggy process might unintentionally try to access regions of memory which does not belong to it, or a malicious process might try to modify the behaviour of another process. To protect processes from each other the concept of virtual memory was invented. *Virtual memory* provides an abstraction for the processes executing on the CPU (Tanenbaum 2001).

Address translation is the term used to describe the method of providing the link between physical and virtual memory. These memory translations are carried out by the *memory management unit*, MMU, which is usually part of the processor or provided as a separate hardware circuit in the system.

2.4 Runtime optimisations

One method for doing runtime optimisations, which is also used by standard compilers, is peephole optimisation. *Peephole optimisation* provide simple low-level optimisations which are usually architecture-specific. It is implemented as a sliding window that analyses small blocks of assembler instructions looking for possible modifications to optimise the block.

Fischer & LeBlanc (1991) mentions elimination of multiplications by 1 and additions of 0, or replacing a sequence of instructions by a single instruction with the same effect as possible peephole optimisations. They state that a common implementation of a peephole optimiser is carried out by providing a hashed table of common patterns and their respective optimisations. Since the number of patterns is very large it is important that only common patterns are identified and stored to reduce the cost in time of the algorithm.

An interesting project is *LLVM* (2004), the Low Level Virtual Machine. LLVM optimises applications during compile-time, link-time, run-time, and in idle time between runs. A low-level RISC-like instruction set has been developed and applications that want to make use of LLVM must be compiled with a modified version of GCC (the GNU C-compiler) that can generate machine-code based on this instruction set. This means that applications must be specifically compiled to use LLVM and they cannot be run standalone. It is, however, possible to recompile LLVM-binaries into native binaries. Lattner & Adve (2004) outlines how this system works.

2.5 SIMD - Single Instruction, Multiple Data

SIMD is a method used to process multiple data entries in one operation. It is commonly used to carry out simple arithmetic or logical operations on vectors of data points; such as adding four floating point values to four other floating point values (*IA-32 Intel Architecture Software Developer's Manual* 2001*b*). Multimedia applications or scientific calculations provide the most frequent use of these kinds of instructions. As good vectorisation of data requires good planning and in-depth knowledge to be implemented into software, it is only used in a handful of applications, even though it can provide a considerable increase in performance.

SIMD instructions were first used in the 1970's in vector supercomputers. In 1994, Hewlett-Packard introduced SIMD to the general-purpose CPU market by implementing the MAX instruction set into its PA-RISC processor. Today it is found in processors such as the Motorola PowerPC, Intel IA32, and Sun Sparc to name a few (Espasa, Valero & Smith 1998, *Wikipedia* 2004).

3 Problem description

The importance of being able to efficiently execute binary code from other architectures is identified by Altman, Kaeli & Sheffer (2000). They state that an important factor when introducing a new platform is to aid existing software developers on other platforms to port their software to the new platform. Many developers find this task difficult and time consuming. By providing emulation, existing software can be run without modification. This eliminates the fear of loosing an investment for the developer if the platform fails as it will still be running on its initial target platform.

Another important aspect of having an efficient emulator is because of its use in byte-code based software platforms such as Java and .NET. Having a method that can easily be re-adapted to run different kinds of binary- or byte-code efficiently on different architectures will make it easier to write efficient emulators and virtual machines (i.e., for Java). This, in turn, is valuable to researchers experimenting with this kind of technology, for instance when working on new ways of optimising existing binary translators. The reason being that focus is shifted away from the actual implementation of the CPU-core as it is generated from the scripts, and therefore it is easier to modify the entire system by either changing the script or by modifying the parser so that it can generate the code that is needed, such as symbols for a debugger.

3.1 Aim and objectives

The aim of this paper is to evaluate a method for providing an efficient and portable emulator for binary code by using a set of scripting languages. In order to accomplish this task (i) the scripting languages that are used to define how the source architecture can be mapped to the target architecture are designed. This consists of a source definition language, called *Bricoleur*, a virtual instruction set, *Fanà*, and a target definition language. A basic emulator architecture is defined, which can be used together with the designed languages in order to produce a working CPU emulator. (ii) A simple IA32 emulator for PowerPC is developed using the scripting languages, and a small subset of the proposed emulator architecture. (iii) The performance of this emulator is measured and compared to other, existing, emulators using a benchmark in an effort to show that the system proposed in this report can provide competitive performance. Due to the limitations of the benchmark used the results might not reflect the performance of a complete emulator, as only the throughput of basic instruction is measured. Another limitation of this system is that it does not handle virtual memory, paging of memory, and memory protection. These features are required in order to emulate modern processors accurately.

3.2 Requirements

List of the two most important requirements along with compliance criteria.

1. CPU-core performance
   The CPU-core produced from the output of the scripts must perform better (execute code faster) than a CPU-core using only an interpretive- or threaded code-based core. It is however not required to outperform a CPU-core written totally from scratch that has been hard-coded for one specific source and target platform.

2. Expressiveness of scripting languages
   The scripting languages must be expressive enough so that all source and target architectures can be described by them. Yet, they must be simple enough so that the amount of work required to write a script for a source and target platform does not exceed the amount of work it would take to hard-code the solution from scratch for a specific source and target. This requirement is difficult to verify without a complete system implementation, and because of that, is not validated in this report.

   It must be possible to map every source instruction to a set of target instructions without losing any of the effects the source instruction would have if run on the native hardware. As is shown in section 6 this requirement is not met by the system proposed here.

4 Architecture

Proposed here is an architecture which intends to reduce the complexity of creating a JIT-compiler for emulators or virtual machines. One of the problems when creating a JIT-compiler is that they are difficult to port to other platforms since the implementation depends on being able to create binary code for the target platform. In order to reduce this problem, a virtual instruction set architecture is used to separate the source and target code in a way that is platform independent. When creating a dynamically recompiling emulator, large tables of data which describes how to decode and translate binary code has to be written. These tables make up the largest portion of what is known as the CPU translation core, or CPU-core for short. To avoid having to do repetitive work when designing these tables, scripting languages are defined which are used to provide enough information about the source and target architecture that a computer program can generate these tables automatically.

In order to achieve this, the system proposed by this report consists of a target definition language which is used to define how the target architecture behaves, as well as a source definition language with the purpose of defining the emulated system. The target is the system for which the CPU-core should generate binary code and the source system is the one that is to be emulated. The idea is to be able to combine the source definition of an architecture with a target definition of another architecture in order to produce the CPU-core for an emulator.

The process described is not intended to create a complete emulator, only a CPU-core which can then be used to build an emulator. Therefore the system as described here is not designed to run as fast as possible, but rather to present a method that can be adapted by someone who is familiar with the C programming language and assembler. Projects such as DAISY use complex definition languages that are not well documented publicly, and therefore require the user to study the project source code in order to be able to use it.

Sharp (2001) lists the following advantages of using an intermediate language (he refers to it as intermediate code). Advantages that do not apply to the design chosen for this project have been left out.

- The native code generator can be re-targeted to support other target platforms without affecting the implementation of the source architecture.
- Complex instructions that consists of multiple steps (or stages) can be decomposed into a set of simpler, virtual, instructions.
- It is easier to implement optimisations based on the intermediate code since it breaks down complex instructions into smaller units and therefore can provide more meta-data than the original instruction.

He also mentions that debugging is easier which also applies for this project, although in a different sense since he makes the assumption that the system will always use IA32 as the target architecture. Because of the intermediate language, a debugger that is portable across supported target architectures can be constructed since errors can be mapped to the V-ISA, and are therefore not dependent on the target architecture implementation.

4.1 A basic JIT-based emulator architecture

The design proposed here is based on the designs described by Sharp (2001) and Traut (1997).

A basic dynamically compiling emulator consists of four components. The first component in the pipeline is the dispatcher. The purpose of the *dispatcher* is to dispatch emulator execution to the appropriate handler. Handlers may include the instruction decoder, a trap or interrupt handler, or an emulator for a specific hardware in the emulated (source) system other than the CPU.

The second component is the *compiler* which is called by the dispatcher whenever it reaches an execution point which has not yet been compiled. The compiler parses the block of source binary code that is to be executed and compiles it into target binary code. The process involved in this step is further elaborated on in section 4.2.

A *translation cache* is used to store blocks after they have been compiled. Succeeding calls made to an already compiled block of code are simply rerouted to the translation cache, thereby bypassing the compiler. Each block carries a counter, which is keeping track of the number of calls made to that block. When the number of calls exceeds a predetermined threshold an optimiser is invoked on the code which will try to make it run faster. The reason why not all code that is compiled is optimised is because of the *principle of locality*, which states that only about 10% of the code is executed more than once (Tanenbaum 2001, Sharp 2001).

The downside to using a translation cache is that if the application being executed modifies its own code the emulator have to somehow detect the modification and invalidate that block in the cache. Detecting modifications of already compiled code is a complex problem not covered in this report.

The last component in this emulator architecture is the optimiser. The purpose of the *optimiser* is to make decisions based on predetermined rules that define how target instructions can be reordered or pruned in order to increase execution speed. For instance, the IA32 only require one instruction to load a 32-bit value into a register whereas the PowerPC requires two instructions. The compiler always tries to find the most exact mapping between the source and target in order to maintain emulation precision, and will therefore use 32-bit load instructions whenever the source architecture does. A simple peephole optimiser can be constructed that replaces the 32-bit load instructions on the PowerPC with their 16-bit equivalents.

Figure 1 is an activity diagram depicting the process beginning with looking up the next address in the program counter and ending in executing a block of code. Only the architecture-independent parts of the dispatchers work are shown in this diagram.

4.2 The translation process

One of the tasks that is to be taken care of during translation is to extract instruction arguments from the source binary code. In the translation table depicted in figure 2, the type called “instruction_argument” specifies the kind of argument and how many bits it consists of. Arguments are dependent on the source architecture and are created when parsing a Bricoleur script, see section 5.4. One pre-defined argument class exists, that is the “ARG_IMMEDIATE” type, which indicates that an immediate data value follows.

When translating a block of binary code a decision has to be made about when to stop translating. The constructed block can not be too short since then the compiler will have to be called more often and emulation might be slow. The same thing will happen if the block is too long, since time also need to be spent emulating peripheral

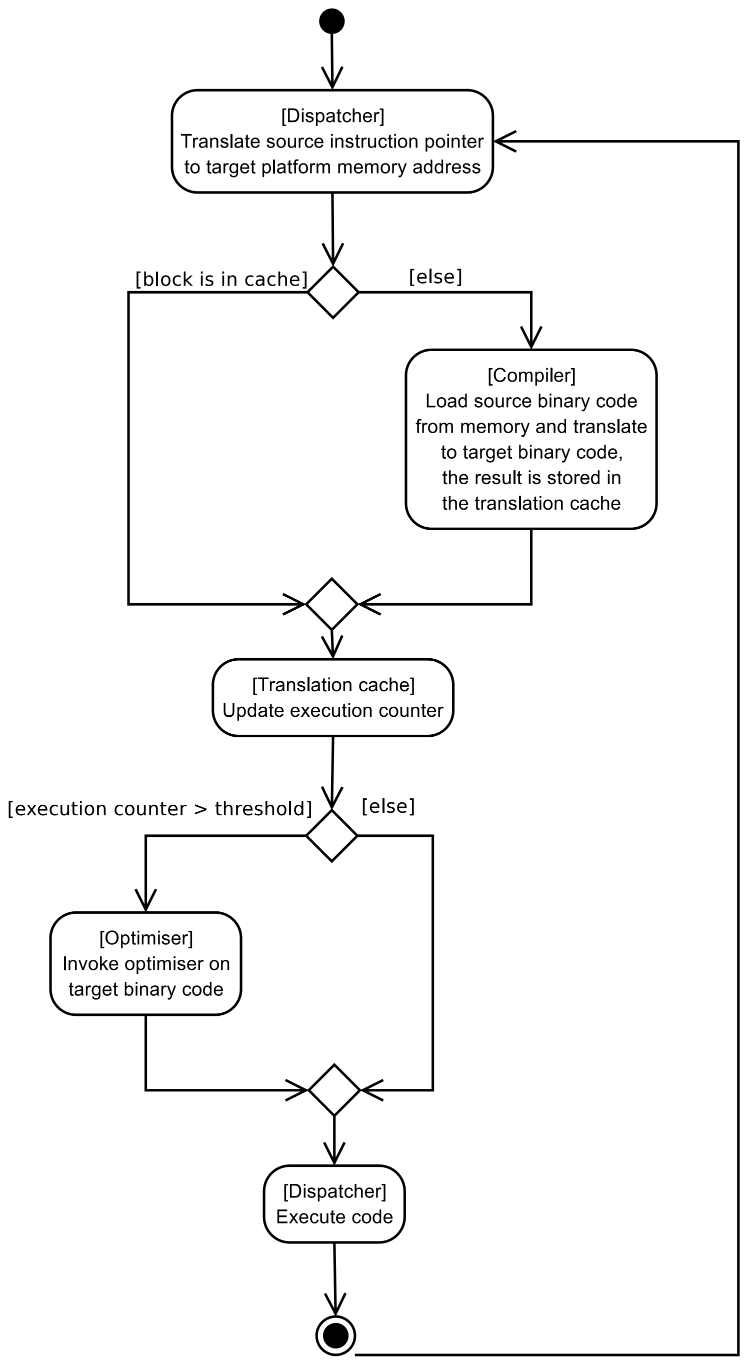


Figure 1: Activity diagram showing the emulation process related to the CPU-core. It starts with the dispatcher fetching the next address to be executed. If the block of code located at that address has not yet been compiled the compiler is invoked, otherwise the code is loaded from the translation cache, and executed. If the compiler was invoked it will translate the source instructions to native binary code and store it in the translation cache. If the code block has been executed from the cache a certain amount of times the optimiser is invoked on it. After the block of code is ready to be executed the dispatcher transfers control to it, and when execution is done the process is repeated for the next address.

hardware present in the source architecture, such as a graphics controller, network card etc. The emulated graphics hardware, for instance, has to be processed at least every frame in order to appear smooth.

Cmelik & Keppel (1994) identifies a number of breakpoints. These are instructions that transfer control outside the current block, software traps, and memory synchronisation instructions. Sharp (2001) mentions that a small forward branch outside the current block might be part of a loop and therefore should be included in the translated block to avoid performance loss. The break condition has to be stored and returned to the dispatcher so that it knows what to do next. If there is no obvious break-point in the binary block that is being translated, the size of the translation cache sets the upper bound of how many instructions to translate.

4.2.1 Translation table

An important component in an emulator is the CPU-core. The purpose of the *CPU-core* is to decompose a block of binary code in the source architecture's format, and to emulate their functionality as accurately as possible.

One way of implementing a CPU-core in a dynamically recompiling system is to provide a table indexed on the source opcodes, where each entry holds the information necessary in order to emulate that specific instruction on the target system, such as the binary translation to the target platform and how to read possible arguments. This is known as a translation table. An example of a translation table is depicted in figure 2, which is the table structure that have been devised for the emulator used to evaluate the method described in this report.

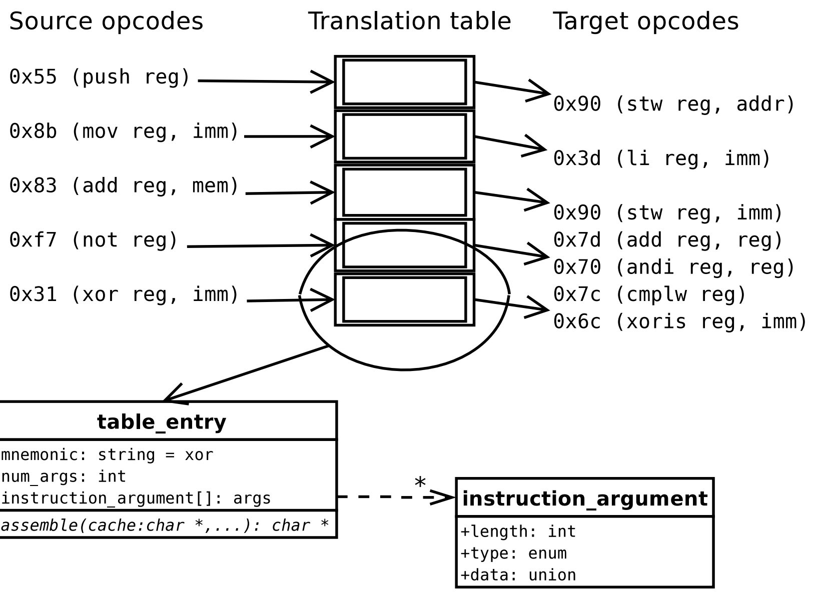


Figure 2: Example of a translation from source to target instruction set using a translation table. Source opcodes are used as the primary key of table entries. Each entry holds information about the opcodes arguments, and how to parse them, as well as a virtual method (assemble) which, when called, will produce the target opcodes.

Each entry in the table stores information about the instruction at that entry. This includes a string containing the name of the source instruction, which is useful for debugging purposes. The number of arguments and an array of "instruction_argument" type store meta-data on the instruction arguments, so that the emulator knows how to parse these from the binary code. Every entry has a virtual method called "assemble" which is called to emit the target platform instructions used to emulate the specific instructions. This method requires a pointer to the current position in the instruction cache to where it should write its data. Any parsed arguments must also be sent to assemble. An example of the assemble method for the IA32 instruction "add register, immediate" can be found in section 5.5. This process is detailed in section 4.2, and a sample implementation in C++ of the various data-types needed to construct the translation table is depicted in table 3.

```
/* Argument type */
typedef enum arg_e {
        ARG_NONE,   /* No argument */
        ARG_IMMEDIATE, /* Immediate value */
        ARG_EXTERN, /* External function call */
} arg_t;

/* Argument meta-data */
typedef struct inst_arg_s {
        uint8_t len;            /* Argument size in bits */
        arg_t   type;           /* Argument type */
        /* The Argument data */
        union {
                int32_t imm32;
                int16_t imm16;
                int8_t  imm8;
        };
        /* If type == ARG_EXTERN use the appropriate function */
        union {
                void    (*func8)(int8_t);
                void    (*func16)(int16_t);
                void    (*func32)(int32_t);
        };
} inst_arg_t;

/* Translation table entry type */
typedef struct entry_s {
        const char *mnemonic; /* Source instruction mnemonic */
        uint8_t     num_args; /* Number of arguments */
        inst_arg_t *args;     /* Array of pointers to argument data */
} entry_t;

/* Translation table type */
typedef map<uint32_t, entry_t *> ttable_t;
```

Figure 3: Example of the data-types required to construct a translation table in C++.

By placing the table into an array, source instructions can be identified in constant time. The intention is to be able to construct a block of instructions that can be executed on the target platform as quickly as possible.

5 CPU-core generation

In order to make it easier to create a portable CPU-core, a script language has been created. The language is divided into three parts. Bricoleur is the language whose purpose is to define how the source architecture is designed. Each instruction defined in Bricoleur contains a functional description made in the language called Fanà. The third part is the target mapping language which maps Fanà instructions to native instructions of the target architecture.

The CPU-core is constructed from a Bricoleur script and a target mapping by the process presented in figure 4.

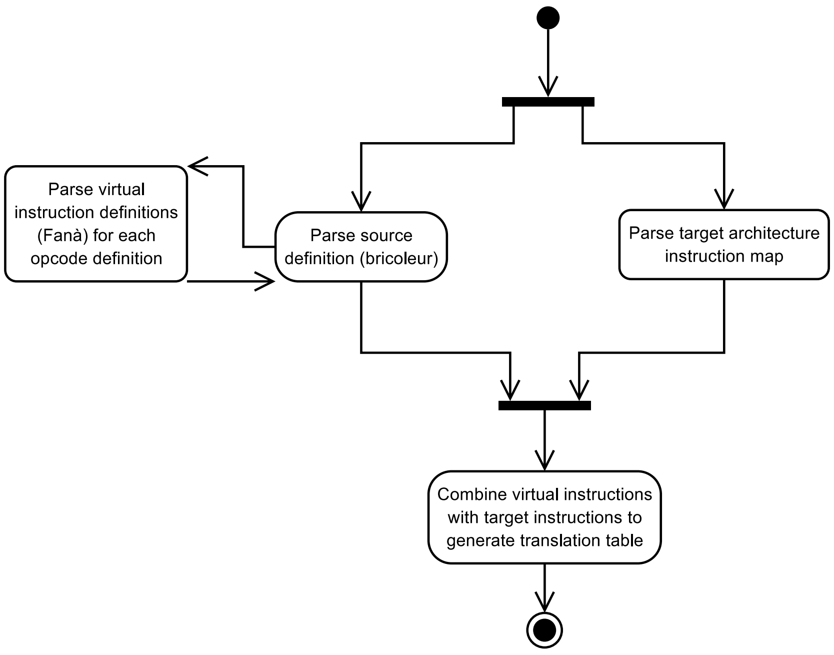

Parse virtual instruction definitions (Fanà) for each opcode definition
Parse source definition (bricoleur)
Parse target architecture instruction map
Combine virtual instructions with target instructions to generate translation table

Figure 4: Activity diagram of the process by which a CPU-core is generated from a Bricoleur script and a target architecture mapping script. Source and target scripts can be parsed independently of each other after which they are combined in order to produce a translation table, which can be used by an emulator.

5.1 Microprocessor architecture types

While developing these scripting languages both CISC and RISC architectures were evaluated. Each instruction on CISC architectures consists of one or several byte blocks that identify the instruction, followed by none or several byte blocks for each instruction argument (*SY6500/MCS6500 Microcomputer Family Programming Manual* 1976, *IA-32 Intel Architecture Software Developer's Manual* 2001*c*, *Wikipedia* 2004). The binary value that makes up the instruction is known as an opcode. RISC architectures on the other hand have a constant opcode size regardless of the number of arguments, normally four bytes in length. The instruction is identified by a constant number of bits starting from the most significant bit of the instruction. Arguments are placed in bit-fields that are encoded into the four byte opcode. For this reason it is not possible to load immediate values that are 32-bits or larger using only one instruction, which is possible on CISC architectures (*MIPS32 Architecture For Programmers* 2003, *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* 2001, *Wikipedia* 2004).

5.2 Virtual instruction set architecture

A *virtual instruction set architecture*, or *V-ISA*, defines a set of low-level instructions that are not used by any existing hardware design. These instructions are typically similar to the assembler instructions found in todays microprocessors. Both Java and Microsoft's .NET define their own V-ISAs and when an application is compiled for one of these systems it is translated into the machine code that the respective V-ISA has defined (Adve, Lattner, Brukman, Shukla & Gaeke 2003).

The low level virtual machine project, *LLVM*, has defined a V-ISA called *LLVA* (Lattner 2004). The purpose of *LLVM* is to provide an architecture specifically designed for optimisation. *LLVM* is a compiler framework that compile high-level languages into LLVA instructions. LLVA is designed with the intention of being executable on any modern architecture, and it has been tested on Intel IA32 and Sun Sparc. The project has also taken into account the difficulties of handling exceptions and self-modifying code and has drawn from the experience accumulated by other projects conducted in the area (Lattner & Adve 2004, Adve et al. 2003).

Here we use a subset of the instruction set defined by *LLVA* as the glue between the source and target definitions. This works as an abstraction layer in order to increase portability by reducing the number of mappings that have to be written. If a new target platform is to be added, the developer would only have to create mappings between the target platform's instruction set and the virtual instruction set, otherwise every instruction of every source definition would have to modified to accommodate the new platform. This reduces the number of required mappings from $n*(n-1)$ to $2n$ (where n equals the number of target platforms).

The downside to using an architecturally impartial virtual instruction set is that it does not accommodate for making optimisations to the target code based on the source architecture. Finding a close mapping from the source to target architecture instruction set for complex instructions become more difficult. For instance, when SIMD instructions are broken into virtual instructions that do not contain vector instructions it is difficult to map these back into vector instructions on the target side.

5.3 Fanà - a virtual instruction set

The Sufis call this state fanà, the annihilation of the individual selfhood. In fanà the characteristics of the little self dissolve so that the big Self can show through.
- Nachmanovitch (1990, p. 52)

Fanà is a a virtual instruction set created as part of this project. It acts as an intermediate code to make it easier to add new target architectures to the system by specifying a common interface between the source and target definitions. Fanà is a stripped down version of LLVA (see section 5.2). Since LLVA is designed to be generated by compiling high-level languages like C and C++ it contains some constructs that are not useful when generating code from binary machine language. One example is the “vaarg” instruction which is used for handling variadic procedure arguments. LLVA is designed to be easy to optimise and therefore makes a good candidate as a V-ISA for this project. Since LLVA have all the necessary functions, have been tested on both CISC and RISC architectures, and is similar to IA32 instructions which is the most widely used desktop architecture at this time, and is therefore easy to learn for the majority of computer programmers, other virtual instruction sets have not been evaluated for this system.

Fanà is a RISC-like language based on a small set of instructions. The context-free grammar for Fanà can be found in section A.1. Another option would have been to use a CISC-like language based on a large number of instructions. The advantage of having a large number of instructions is that complex instructions, such as the FCOS (cosine) instruction on IA32, are easier to map between the source and target if they both have a compatible implementation. On the negative side is the fact that it is difficult to predefine what instructions to include in the language. One solution would be to allow arbitrary introduction of new instructions but this method would make it difficult to construct a good optimiser. Leone & Lee (1994) is using instruction block templates to speed up code generation in their project. Templates could be introduced into Fanà as well to define complex instructions, and at the same time allow the target definition to provide a direct mapping from a template to a single native instruction that produce the same state, if one exists.

5.3.1 Description

The type specifiers available in Fanà are defined by the Bricoleur script to which the current script belong. There are however three predefined types that are always available. They are, float, which is a 32-bit floating point type, double, which is a 64-bit floating point type, and label which is used to set a label at the destination of a branch instruction.

Instructions are carried out from right to left, which means that when an instruction has a target argument it is always the one furthest to the left. For instance, "store uint32 * register reg3, reg5" would store the contents of reg5 at the memory location stored in reg3. The first argument of the store instruction must be of pointer type. "sub uint32 reg0, reg1, reg2" should be read as $reg0 = reg1 - reg2$.

"shl" and "shr" shifts the second argument the number of bits specified by the third argument either left or right, zeros are placed in the new positions. "rol" and "ror" works like shl and shr except that the bits are wrapped and no zeros are added.

Branch operations are either relative or absolute. An absolute branch is of pointer type and cannot be negative, unlike the relative branch. "goto" is an unconditional branch which is always carried out, but there are also a number of conditional branches. A conditional branch is dependent on the conditional flags set automatically by a previously executed instruction or by implicitly using "cmp" before the branch instruction. "gotolt" branches if the first argument of the comparison is less than the second, "gotolte" on less than or equal, "gotoeq" on equal, "gotogte" on greater than or equal, and "gotogt" on greater than.

It is possible to declare temporary variables in a block of Fanà if that is necessary to express the source architecture instruction. Temporary variables should be declared as late as possible to make it easier for a dynamic register allocator to decide when to swap a register to the stack (if it has to). Temporaries are translated into calls to the register allocation system and therefore they are not mapped by the target mapping language to any target specific declarations.

Example of a loop that iterates 100 times, written in Fanà:

```
/* Store the value 100 in r1 */
mov sint32 gpu.r1, 100;
/* Set a branch label */
label loop;
/* Subtract 1 from r1 */
sub sint32 gpu.r1, gpu.r1, 1;
/* Compare r1 to 0 */
cmp gpu.r1, 0;
/* If comparison resulted in r1 being greater than 0,
* branch to label loop */
gotogt label loop;
```

5.4 Bricoleur - the source definition language

There is a French word, bricolage, which means making do with the material at hand: a bricoleur is a kind of jack-of-all-trades or handyman who can fix anything.
- Nachmanovitch (1990, p. 86).

Bricoleur is the source architecture definition language. The purpose of Bricoleur is to describe the source CPU-architecture so that a CPU-core can be automatically generated in C-code by combining the Bricoleur definition with a target definition. The only things the developer has to manually add are handlers for software interrupts and memory management, in case the system requires more than linear memory access. The context-free grammar can be found in section A.2.

5.4.1 Details

A Bricoleur script can consist of up to four different sections. The first section is the "flags" section where CPU-flags are defined. After that follows the "types" section where new type-specifiers can be defined. All registers available in the source architecture are defined in the "registers" section. The fourth and last section is the "opcodes" section where instructions are defined. The contents of each section are enclosed by braces like in the C programming language ("{", "}").

The "flags" section contains a list of identifiers which represent a 1-bit flag, such as an arithmetic overflow. Each flag is separated by a semicolon. Flags can be used by instructions to register certain conditions or to carry out conditional operations. The flag section has to be defined first in a bricoleur script.

The "types" section has to come before the opcode definitions so that type-names can be evaluated by the parser. A new type is defined by first giving it an identifier followed by the size in bits and an optional "signed" keyword, which indicates that the type is signed instead of unsigned (the default). Every new type declaration must end with a semicolon. Types that are declared in this section can only be used to describe integer values. For that reason the predefined types "float" and "double" exist, float is a 32-bit floating point value, and double is a 64-bit floating point value (which corresponds to the float and double in Fanà). In order to be able to handle complex argument types that cannot be defined using bricoleur, such as the IA32 ModR/M argument (see section C.1.1), it is also possible to call an external C-function. These kinds of argument types are defined by using the keyword "extern" followed by the name of the external function and argument type. The argument is extracted from the binary data and passed to the specified function as a single argument.

After the type specifier the registers available on the source platform are defined in the "registers" section. First of all a group-name has to be defined after which a block enclosed by braces contains the register definitions. A register is defined by entering its name followed by a type specifier. If the register can be divided into sub registers it is possible to start a new block with braces and define any sub registers, an example of this is available in section B.1. Every register definition has to be ended by a semicolon. Registers are referenced either by their group name or by "<group>.<register>".

The final and required section is the "opcodes" section where the source architecture instructions are defined. A new instruction is defined by simply entering the name of the instruction followed by a new section. Within this section each new opcode is defined by entering the keyword "op" followed by either a constant or a ranged constant. A ranged constant is on the form "(value, bitsize)" where value is the actual opcode value and bitsize is the number of bits it occupies of the opcode field. If the opcode has arguments, the definition is followed by yet another section where each argument type is defined using either a type specifier or another ranged constant. If an argument is not parsed from the binary code it is prepended by the keyword "implied". An example of an implied argument can be found in section B.1. Each opcode definition inside an instruction section must have the same number of arguments. If different versions of the same opcode exist that have a different number of arguments a new instruction block must be created, therefore it is possible to have multiple instruction blocks that are identified by the same instruction name. The opcode definitions are followed by an optional flags definition which is a list of flags which are modified by the operation. An appropriate method to properly handle condition-flags have not been found, and consequently, it is currently not possible to emulate code which contains instructions which depend on condition-flags. The final section is the "func"-block which is a list of Fanà instructions that define how the instruction is to be emulated. Arguments are accessible as "%<number>", where <number> is the position of the argument in the definition list, starting with 1.

The language permits defining the same instruction multiple times. The reason behind this design is that there exists architectures which allow the same instruction to have a different number of arguments depending on what the programmer want to achieve. The Fanà definition depends on the fact that every opcode definition within that instruction-block have the same number of arguments. For this reason it would not be possible to define all versions of an instruction which allows a varying number of arguments inside one instruction definition.

Example of a simple CPU-architecture consisting of two 8-bit registers and an add instruction:

```
types {
    uint8   8; /* unsigned 8-bit type */
}

registers {
    gpu {
        /* registers A and B are of uint8 type */
        A uint8;
        B uint8;
    }
}

opcodes {
    add {
    /* Opcode 1 means that the add is carried out on the A register
     * Opcode 2 means the B register */
        op 0x1 { implied register gpu.A, uint8 };
        op 0x2 { implied register gpu.B, uint8 };
        func {
               /* Add the first argument to the second and
                * store the result in the first argument */
               add uint8 %1, %1, %2;
        };
    }
}
```

5.5 Target mapping

The target mapping from Fanà is constructed using a definition language that resembles Bricoleur but which is much simpler in design. Section A.3 holds the context-sensitive grammar of the target mapping language. Types and registers are defined the same way as in Bricoleur (see section 5.4), but there are no global "flags" or "opcodes" sections. The target mapping contains a new section called "translators" where Fanà instructions are mapped to small sections of code that translate them into native binary code.

5.5.1 Usage

In the "translators" section instructions are defined by starting a new section identified by the Fanà instruction name. The instruction section contains a nested list of instruction arguments followed by a code block. The code-block should contain C-code which generate the necessary target binary code to emulate the instruction. The instruction pointer is accessed as "p" and arguments are accessed the same way as in Fanà, using %<number>. Each block is then translated into C-functions which are called by the emulator's compiler to generate the target binary code to emulate each instruction. An example of such a C-function can be found in figure 5. Each code block can contain an optional flags section which define which flags are modified. These flags must conform to the naming scheme defined by the source architecture bricoleur script.

Example of what the translation mapping for the result presented in figure 5 looks like (GENOP3 and EMIT32 are specific to PowerPC code generation and are explained in section B.2):

```
types {
    sint32 32 signed; /* 32-bit signed type */
}

translators {
    add {
        register gpu {
            sint32 {
                EMIT32(p, GENOP3(ADDIS, %1, %1, (int16_t) (%2 >> 16)));
                EMIT32(p, GENOP3(ADDI, %1, %1, (int16_t) (imm & 0xffff)));
            }
        }
    }
}
```

emit_add_r_i32

```
inline uint8_t *emit_add_r_i32(uint8_t *p, ppc_gpr_t reg, int32_t imm) {
    /* add the higher 16 bits of the immediate value to reg */
    EMIT32(p, GENOP3(ADDIS, reg, reg, (int16_t) (imm >> 16)));
    /* add the lower 16 bits of the immediate value to reg */
    EMIT32(p, GENOP3(ADDI, reg, reg, (int16_t) (imm & 0xffff)));

    /**
     * Set the AF, SF and PF flags, the remaining flags can be
     * mapped to existing PowerPC flags set by addis/addi.
     */
    p = emit_mov_r_i32(p, gpr_r11, imm);
    p = emit_set_af(p, reg, gpr_r11);
    p = emit_set_sf(p, reg, 32);
    p = emit_set_pf(p, reg);

    /* Return the next available address in the code block */
    return p;
}
```

Figure 5: The assemble method of the IA32 instruction “add register, immediate” for PowerPC. The 32-bit immediate value has to be broken into two 16-bit parts since RISC architectures do not support 32-bit immediate values. The handling of flags have been added manually after code generation.

6 Results & analysis

A benchmark is created in order to analyse emulation performance. The benchmark consists of a function that calculates prime numbers. The reason this benchmark have been chosen is that it will only measure pure instruction throughput as it is a mathematical function that does not require to store or retrieve information from an external source, and therefore will not use functionality that is currently unsupported such as memory management. This section will also show why the second requirement is not met by the system published in this report.

6.1 Performance benchmark

This benchmark consists of a very simple algorithm for finding all prime numbers between 3 and n (in this case $n = 10,000$). This algorithm is simple to implement and requires enough processing power to make it measurable. There is also no need for memory access because it is a brute force algorithm which does not use information about already found primes.

C-code of algorithm:

```
for (i = 3; i < 10000; i++) {
    prime = 1;
    for (j = 2; j < i; j++) {
        if (!(i % j)) {
            prime = 0;
        }
    }
}
```

When compiled with GCC, optimised for minimum size (to make it easier to implement the emulator as it does not require support for as many instructions), it translates into the instructions described in table 1. Code related to condition flags is listed in table 2. On the PowerPC, register r20 is statically mapped to EAX on the IA32, r22 to EDX, r24 to ESP, and r25 to EBP. Branches in the PowerPC code are relative, but in the IA32 code the absolute address has been used in this example to make it easier to see what the destination of the jump is.

Table 1: IA32 to Fanà to PowerPC instruction mapping for the benchmark

Address	**IA32**	**Fanà**	**PowerPC**
0x1	mov edx, 0x3	mov sint32 register gpu.edx, 0x3	lis r22, 0x0 ori r22, r22, 0x3
0x6	mov ebp, esp	mov sint32 register gpu.ebp, register gpu.esp	or r25, r24, r24
0x8	mov eax, 0x2	mov sint32 register gpu.eax, 0x2	lis r20, 0x0 ori r20, r20, 0x2
0xd	cmp eax, edx	cmp gpu.eax, gpu.edx	subfco. r22, r20 *Test and set parity flag* *Test and set adjust flag* *Test and set sign flag*
0xf	jge 0x16	gotogte 0x16	bge 12
0x11	inc eax	add sint32 gpu.eax, gpu.eax, 1	li r11, 1 addco. r20, r20, r11 *Test and set parity flag* *Test and set adjust flag* *Test and set sign flag*
0x12	cmp eax, edx	cmp gpu.eax, gpu.edx	subfco. r22, r20 *Test and set parity flag* *Test and set adjust flag* *Test and set sign flag*
0x14	jl 0x11	gotolt 0x11	blt -12
0x16	inc edx	add sint32 gpu.edx, gpu.edx, 1	li r11, 1 addco. r22, r22, r11 *Test and set parity flag* *Test and set adjust flag* *Test and set sign flag*
0x17	cmp edx, 0x270f	cmp edx, 0x270f	lis r11, 0x0 ori r11, r11, 0x270f subfco. r11, r22 *Test and set parity flag* *Test and set adjust flag* *Test and set sign flag*
0x1d	jle 0x8	gotolte 0x8	ble -48

Table 2: PowerPC functions that test and set various bits in the IA32 EFLAGS register, the result is stored in r19. x86_pf_table is an external table of constants.

Function	**PowerPC Instructions**
Test and set parity flag	li r11, 0xff
	and r12, <destination register>, r11
	li r11, <x86_pf_table>
	lbz r11, <destination register>, r11
	cror cr1, cr0, cr0
	cmpw r11, 0
	beq 12
	ori r19, r19, 0x4
	b 12
	lis r11, 0xfffffffb
	and r19, r19, r11
	cror cr0, cr1, cr1
Test and set adjust flag	xor r11, <destination register>, <source register>
	li r12, 0x10
	cror cr1, cr0, cr0
	cmpw r12, 0
	beq 12
	ori r19, r19, 0x10
	b 12
	lis r11, 0xffffffef
	and r19, r19, r11
	cror cr0, cr1, cr1
Test and set sign flag	li r11, 8
	subfic r11, r11, <bit length>
	neg r11, r11
	sraw r11, <destination register>, r11
	li r12, 0x80
	cror cr1, cr0, cr0
	cmpw r12, 0
	beq 12
	ori r19, r19, 0x80
	b 12
	lis r11, 0xffffff7f
	and r19, r19, r11
	cror cr0, cr1, cr1

The algorithm has been placed inside a loop on the IA32 which executes it 100 times. On a Pentium 3/933MHz the algorithm executes in about 28.0 seconds. In the emulator the native code is constructed by parsing the source (IA32) binary code after which it is executed once. This process is repeated 90 times, and every tenth time (plus the first run) it is executed twice instead of once, achieving a total of 100 runs. This design is used to compensate for the principle of locality (see section 4.1). The result from running the test on a 1GHz PowerPC 7447 (also known as a G4) is about 14.5 seconds of total duration of execution. The generated PowerPC-instructions are not optimised in any way.

Applying a simple peephole-optimisation to the code which replaces 32-bit register load operations with 16-bit loads resulted in a four second reduction in execution speed on the PowerPC. When further optimising the code by removing unnecessary CPU-flags it runs more than twice as fast. Both these optimisations were made manually since an automatic optimiser have not been implemented.

Using QEMU, which is an emulator using threaded code (see section 2.1.2), on the same PowerPC as mentioned above, the duration of execution is approximately 162.0 seconds. While on Bochs, which is an emulator using an interpretive CPU-core (see section 2.1.1), the result is 2880.0 seconds.

The results presented in table 3 show that the system presented in this report can compete with existing open source emulators. A more complex benchmark containing more complicated instructions would probably result in better test accuracy. The amount of code required to emulate each instruction should be relative among the different emulators since they should all present the same result after execution, so it is likely that the result of such a benchmark will not be all that different. If the benchmark was carried out on a complete system emulator that is able to handle memory protection and SIMD instructions the results would likely be different, but those kind of functions are not handled by the system presented in this report.

The second result presented in table 3, labelled "Bricoleur and Fanà", is a modified version of the auto-generated code that contains instructions for testing and updating source architecture condition flags. The current implementation does not fully support handling condition flags but the benchmark would not compare well to existing systems that handle these flags. Since an optimiser would be able to remove these flags, as they are not used by the code in the benchmark, the results of running the system without the flags is also presented.

It is shown by the results that pure instruction execution is several times faster compared to the existing, benchmarked, software solutions in this simple case. Further testing of a more complete emulator, using a more complex algorithm as a benchmark, is necessary in order to establish that this system performs better than other emulators in the general case.

Table 3: Results of execution timing by running the benchmark in different environments. In all cases, except the native run, the emulator was executed on a 1GHz PowerPC 7447. The Unix command "time" was used to measure the duration of execution.

Environment	**Duration (s)**
Pentium 3/933MHz (Native)	21.6
PowerPC 7447/1GHz (Native)	15.1
Our emulator	32.9
Our emulator (no flags)	14.5
QEMU 0.5.5	162.0
Bochs 2.1.1	2880.0

6.2 Expressiveness of scripting languages

As stated in section 3.2 the scripting languages must be expressive enough so that all source and target architectures can be described. Several instructions have been tested throughout the duration of this project and one IA32 instruction have been identified which can not be implemented by using Fanà, and therefore this requirement is not met by the system proposed here. The instruction that breaks the requirement is “rep” which is used to repeat a specified instruction a specific amount of times. Using the implementation of Bricoleur and Fanà as described here does not allow branching between different instruction definitions, only inside a Fanà block, and it is therefore not possible to properly define these kind of instructions.

7 Related work

Two projects that use similar methods to the one described in this report have been identified during this undertaking. The first is Syn68k by ARDI which is a commercial project. Not much is known about the internals of Syn68k except the details published by Hostetter (1995). The project uses a Lisp-like definition language and supports generating either a dynamically recompiling CPU-core for IA32 or an interpretive core for other architectures. Bricoleur is using a C-like syntax since C is more popular than Lisp among programming languages, and the intention is to make this system accessible to as many users as possible (BV 2004). Syn68k have been used in the creation of their Macintosh emulator, Executor.

The second project is DAISY by IBM (Ebcioglu & Altman 1996). The main purpose of DAISY is to develop a binary translator that can run code on VLIW-capable processors. It is also possible to attach a second back-end that translates VLIW-instructions to native code for non-VLIW systems. When running on a VLIW system this emulator will probably perform better since it is optimised for translation directly to VLIW, but if using a translator back-end it is likely to run slower than the system presented in this report since the translation from VLIW to native code is performed real-time, whereas translation from Fanà to native code is performed off-line.

8 Conclusion

In this section the conclusions drawn from the experience gained throughout the duration of this project will be presented.

8.1 Summary

System emulation have several uses both in commercial and non-commercial settings. A system emulator can be used to be able to run older software when making a transition to a newer system that isn't backwards compatible. Another use is to be able to run software that only exists for another platform than the one currently being used, for example, Windows applications on GNU/Linux. However, building a fast emulator can be a daunting process that involves many repetitive tasks which can result in a time consuming, and costly, venture to take on.

In order to remedy this problem a system using three scripting languages, that can be parsed by a computer program in order to generate larger portions of code that otherwise would have to be manually created, are presented and evaluated in this report. These are

1. *Bricoleur*, which is used to describe the source architecture. Bricoleur have been developed from scratch during this project.
2. *Fanà*, which is a virtual instruction-set architecture that describe how to translate source instructions. Fanà is based on the LLVA V-ISA presented by Adve et al. (2003), but with some modifications to adapt it to this system.
3. A *target mapping* language that maps Fanà instructions to code generators for the target platform. It resembles Bricoleur and have been developed from scratch as well.

Parts of the system have been implemented in order to test it using a simple benchmark. The implementation is not presented in this report as it is not relevant to the system that is described. The scripting languages only provide a partial solution to the problem as some parts such as condition flags and memory management is not fully covered by the solution. The benchmark has been modified to take these limitations into account in order to maintain comparability to other emulators.

As shown by the benchmark this system provide better results when emulating parts of an IA32 microprocessor on a PowerPC compared to existing open source emulators. However, in order to show that it performs better in all contexts a system that provides a more complete emulator will have to be implemented so that an algorithm of a higher complexity can be evaluated. Under the conditions used here the system fulfils the first requirement listed in section 3.2.

The second requirement that have been defined is that it should be possible to define all kinds of source and target architectures using the scripting languages presented here. Section 6 show that this requirement is not met since source instructions that have to make branches outside their local Fanà definition can not be defined using the scripting languages defined in this report.

8.2 Future work

Topics that provide interesting research areas related to the work presented in this report is described in this section.

8.2.1 Flags

Condition flags are not handled at all by the current implementation. A possible implementation would be to add blocks of Fanà to the flags-section of Bricoleur in order to define how each flag is set.

One way it could be handled is to in-line the code to update the flag registers in the Fanà definition for each source instruction. As shown in section 6 a lot can be gained by having an optimiser remove unnecessary flags and therefore it must be possible to give hints to the optimiser so that it knows how to remove the in-lined code for each flag when possible.

8.2.2 Hot-spot optimiser

Since there is no point in optimising a block if it will take longer time to optimise and execute it than to run it unoptimised the hot-spot optimiser can decide what level of optimisation to apply based on the number of times it has been accessed by the emulated system.

8.2.3 Persistent translation cache

Blocks of code that are accessed often can use its access counter as a sort of priority measurement in order to keep a small cache of translated and optimised code on stable storage. That way blocks that are often accessed and have been highly optimised survive between emulation runs.

8.2.4 Support for defining memory management

Since memory management is part of many CPUs (see section 2.3) it would be beneficial if it could be defined in the scripting language as well since it might need access to certain CPU registers and state changes.

8.2.5 Catching common system calls

Some common system calls, such as the ones found in the standard C library (i.e., memcpy, strcpy etc), could be caught by the emulator and executed using native code. As some of these operations tend to be heavily optimised in modern operating systems it would run faster using the native code rather than using translated code.

8.2.6 Self-modifying code

The system proposed in this paper does not handle self-modifying code (SMC). Klaiber (2000) talks about the system used in the Crusoe microprocessor to handle this. The Crusoe, however, is using a hardware-based solution which would not work as a software-only implementation. LLVM supports a constrained form of SMC done using LLVA and only applies to future executions of the code block, not to the current execution. Neither of these solutions will work in an emulator.

8.2.7 Fanà templates

Introduce support for templates into Fanà as described in section 5.3. Templates will make it easier to support a closer mapping of complex instructions, such as vector instructions, between the source and target system.

References

Adve, V., Lattner, C., Brukman, M., Shukla, A. & Gaeke, B. (2003), LLVA: A Low-level Virtual Instruction Set Architecture, *in* 'Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)', San Diego, California. [Accessed 04.02.25].
URL: *http://llvm.cs.uiuc.edu/pubs/2003-10-01-LLVA.ps*

Aho, A. V., Sethi, R. & Ullman, J. D. (1986), *Compilers - Principles, Techniques, and Tools*, Addison-Wesley Publishing Company.

Altman, E., Kaeli, D. & Sheffer, Y. (2000), 'Welcome to the opportunities of binary translation', *Computer* .
URL: *citeseer.ist.psu.edu/altman00welcome.html*

Austin, C. & Pawlan, M. (1999), *Advanced Programming for the Java 2 Platform*, Addison-Wesley Publishing Company. [Accessed 04.04.20].
URL: *http://java.sun.com/developer/onlineTraining/Programming/JDCBook/*

Bellard, F. (2004), 'QEMU', (Version: 0.5.4).[Software]. [Accessed 04.04.30].
URL: *http://fabrice.bellard.free.fr/qemu*

BV, T. S. (2004), 'TIOBE programming community index for june 2004'. [Accessed 04.06.05].
URL: *http://www.tiobe.com/tpci.htm*

Cmelik, R. & Keppel, D. (1994), 'Shade: A fast instruction-set simulator for execution profiling', *ACM SIGMETRICS Performance Evaluation Review* **22**(1), 128–137. [Accessed 04.04.07].
URL: *http://citeseer.ist.psu.edu/cmelik93shade.html*

Dalrymple, J. (2003), 'Apple market share rises slightly', *Newsforge* . [Accessed 04.02.10].
URL: *http://www.newsforge.com/business/03/08/13/1424212.shtml*

Ebcioglu, K. & Altman, E. (1996), 'DAISY: Dynamic compilation for 100% architectural compatibility'. [Accessed 04.05.17].
URL: *http://domino.watson.ibm.com/library/cyberdig.nsf/a3807c5b4823c53f85256561006324be/fd183b4dacc05b7585256593007209e7?OpenDocument*

Espasa, R., Valero, M. & Smith, J. E. (1998), Vector architectures: Past, present and future, *in* 'International Conference on Supercomputing', pp. 425–432.
URL: *http://citeseer.ist.psu.edu/espasa98vector.html*

Fayzullin, M. (2000), 'How to write a computer emulator'. [Accessed 04.02.16].
URL: *http://fms.komkon.org/EMUL8/HOWTO.html*

Fischer, C. N. & LeBlanc, R. J. (1991), *Crafting A Compiler With C*, The Benjamin/Cummings Publishing Company, Inc.

Gulker, C. (2003), 'Global IT firm predicts Linux will have 20% desktop market share by 2008', *Newsforge* . [Accessed 04.02.10].
URL: *http://www.newsforge.com/business/03/08/13/1424212.shtml*

Hostetter, M. (1995), 'Syn68k - ARDI's dynamically compiling 68lc040 emulator'. [Accessed 04.05.24].
URL: *http://www.ardi.com/SynPaper/index.html*

IA-32 Intel Architecture Software Developer's Manual (2001*a*), Vol. 3: System Programming Guide, Intel Corporation.

IA-32 Intel Architecture Software Developer's Manual (2001*b*), Vol. 1: Basic Architecture, Intel Corporation.

IA-32 Intel Architecture Software Developer's Manual (2001*c*), Vol. 2: Instruction Set Reference, Intel Corporation.

Jaques, R. (2003), 'Linux brightest star in flat server market', *Vnunet* . [Accessed 04.02.10].
URL: *http://www.vnunet.com/News/1141301*

Klaiber, A. (2000), 'The technology behind Crusoe processors'. [Accessed 04.02.08].
URL: *http://www.transmeta.com/pdfs/paper_aklaiber_19jan00.pdf*

Lattner, C. (2004), 'LLVM language reference manual'. [Accessed 04.04.23].
URL: *http://llvm.cs.uiuc.edu/docs/LangRef.html*

Lattner, C. & Adve, V. (2004), LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *in* 'Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)', Palo Alto, California. [Accessed 04.02.25].
URL: *http://llvm.cs.uiuc.edu/pubs/2004-01-30-CGO-LLVM.ps*

Lawton, K. (2004), 'Bochs', (Version: 2.1).[Software]. [Accessed 04.01.25].
URL: *http://bochs.sourceforge.net/*

Leone, M. & Lee, P. (1994), Lightweight run-time code generation, *in* 'Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation', University of Melbourne, Department of Computer Science, pp. 97–106. [Accessed 04.04.07].
URL: *http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mleone/papers/lw-rtcg.ps*

LLVM (2004), (Version: 1.1).[Software]. [Accessed 04.02.25].
URL: *http://llvm.cs.uiuc.edu/*

Magnusson, P. (2004), 'Simics', (Version: 1.6.6).[Software]. [Accessed 04.02.23].
URL: *http://www.virtutech.se/products/simics.html*

MIPS32 Architecture For Programmers (2003), Vol. 2: The MIPS32 Instruction Set, MIPS Technologies Inc.

Nachmanovitch, S. (1990), *Free Play, Improvisation in Life and Art*, Penguin Putnam Inc.

Pietro, H. D. (2004*a*), 'VirtualPC 2004 vs. VMWare 4: Part ii'. [Accessed 04.02.23].
URL: *http://usuarios.lycos.es/hernandp/articles/vpcvsII.html*

Pietro, H. D. (2004*b*), 'VirtualPC 2004 vs. VMWare 4 performance review'. [Accessed 04.02.23].
URL: *http://usuarios.lycos.es/hernandp/articles/vpcvs.html*

Pountain, D. (2003), ‘The word on VLIW’. [Accessed 04.05.17].
URL: *http://www.byte.com/art/9604/sec8/art3.htm*

Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture (2001), Motorola Inc.

Sanseri, S. K. (2000), ‘Porting Kaffe to a new architecture - chapter 2: Trampolines in the Kaffe JIT compiler’. [Accessed 04.04.06].
URL: *http://www.cs.pdx.edu/ sanseri/kaffe/k2.html*

Sharp, D. (2001), A dynamically recompiling ARM emulator, Master’s thesis, University of Warwick. [Accessed 04.03.23].
URL: *http://llvm.cs.uiuc.edu/pubs/2002-12-LattnerMSThesis.ps*

SY6500/MCS6500 Microcomputer Family Programming Manual (1976), MOS Technology Inc. [Accessed 04.04.29].
URL: *http://www.6502.org/datasheets/synertek_progman.pdf*

Tanenbaum, A. S. (2001), *Modern Operating Systems*, second edn, Prentice-Hall, Inc.

Traut, E. (1997), ‘Building the Virtual PC’, *BYTE* . [Accessed 04.01.25].
URL: *http://www.byte.com/art/9711/sec4/art4.htm*

Virtual PC (2004), (Version 6.1).[Software]. [Accessed 04.01.25].
URL: *http://www.microsoft.com/mac/products/virtualpc/ virtualpc.aspx?pid=virtualpc*

VMWare (2004), (Version: 4.0.5.6030).[Software]. [Accessed 04.02.23].
URL: *http://www.vmware.com/*

White, J. (2004), ‘Darwine’, (Version: DP1.06).[Software]. [Accessed 04.02.11].
URL: *http://darwine.sourceforge.net/*

Wikipedia (2004), Wikimedia Foundation. [Accessed 04.02.18].
URL: *http://en.wikipedia.org/*

A Context-free grammars in Backus-Naur form

A.1 Fanà

Fanà is a virtual instruction-set architecture.

⟨*fana*⟩→ **‘{’** ⟨*instruction list*⟩ **‘}’**

⟨*instruction list*⟩→ ⟨*instruction*⟩ **‘;’**
| ⟨*instruction list*⟩ ⟨*instruction*⟩ **‘;’**

⟨*instruction*⟩→ ⟨*declaration*⟩
| ⟨*nop*⟩
| ⟨*set clr*⟩
| ⟨*double*⟩
| ⟨*triplet*⟩
| ⟨*compare*⟩
| ⟨*branch*⟩

⟨*argument*⟩→ ⟨*register*⟩
| ⟨*flag*⟩
| ⟨*type specifier*⟩ ⟨*variable*⟩
| ⟨*constant*⟩

⟨*nop*⟩→ **‘nop’**

⟨*set clr*⟩→ **‘set’** ⟨*flag*⟩
| **‘clr’** ⟨*flag*⟩

⟨*double*⟩→ ⟨*double ops*⟩ ⟨*type specifier*⟩ ⟨*argument*⟩ **‘,’** ⟨*argument*⟩

⟨*triplet*⟩→ ⟨*triplet ops*⟩ ⟨*type specifier*⟩ ⟨*argument*⟩ **‘,’** ⟨*argument*⟩ **‘,’** ⟨*argument*⟩

⟨*double ops*⟩→ **‘load’**
| **‘store’**
| **‘mov’**

⟨*triplet ops*⟩→ ⟨*arithmetic*⟩
| ⟨*bitwise*⟩

⟨*arithmetic*⟩→ **‘add’**
| **‘sub’**
| **‘mul’**
| **‘div’**
| **‘rem’**

⟨*bitwise*⟩→ **‘and’**
| **‘or’**
| **‘xor’**
| **‘shl’**
| **‘shr’**
| **‘rol’**
| **‘ror’**

⟨*compare*⟩→ **‘cmp’** ⟨*argument*⟩ **‘,’** ⟨*argument*⟩

⟨*branch*⟩→ ⟨*branch op*⟩ ⟨*type specifier*⟩ ⟨*register*⟩
| ⟨*branch op*⟩ ⟨*type specifier*⟩ ⟨*variable*⟩

| ⟨*branch op*⟩ **‘label’** ⟨*identifier*⟩

⟨*branch op*⟩→ **‘goto’**
| **‘gotolt’**
| **‘gotolte’**
| **‘gotoeq’**
| **‘gotogte’**
| **‘gotogt’**

⟨*type specifier*⟩→ ⟨*identifier*⟩ [⟨*pointer*⟩]

⟨*declaration*⟩→ ⟨*identifier*⟩ ⟨*identifier*⟩

⟨*variable*⟩→ **‘%’**

⟨*register*⟩→ **‘register’** ⟨*identifier*⟩

⟨*flag*⟩→ **‘flag’** ⟨*identifier*⟩

⟨*pointer*⟩→ **‘$*$’**

⟨*constant*⟩→ ⟨*int*⟩
| ⟨*hex*⟩
| ⟨*oct*⟩

⟨*identifier*⟩→ ⟨*letter*⟩ | ⟨*letter*⟩ ⟨*digit*⟩

A.2 Bricoleur

Bricoleur is the source architecture definition language.

⟨*bricoleur*⟩→ [⟨*type definitions*⟩] ⟨*opcode definitions*⟩

⟨*type definitions*⟩→ **‘types’** ‘{’ [⟨*type definition list*⟩] ‘}’

⟨*opcode definitions*⟩→ **‘opcodes’** ‘{’ ⟨*opcodes list*⟩ ‘}’

⟨*func block*⟩→ **‘func’** ⟨*fanà*⟩ **‘;’**

⟨*instruction block*⟩→ ⟨*op list*⟩ [⟨*flags list*⟩] ⟨*func block*⟩

⟨*op block*⟩→ ‘{’ [⟨*argument list*⟩] ‘}’

⟨*argument list*⟩→ ⟨*argument*⟩
| ⟨*argument list*⟩ **‘,’** ⟨*argument*⟩

⟨*flags list*⟩→ **‘flags’** ‘{’ [⟨*identifier list*⟩] ‘}’ **‘;’**

⟨*identifier list*⟩→ ⟨*identifier*⟩
| ⟨*identifier list*⟩ **‘,’** ⟨*identifier*⟩

⟨*op list*⟩→ ⟨*op definition*⟩
| ⟨*op list*⟩ ⟨*op definition*⟩

⟨*opcodes list*⟩→ ⟨*instruction definition*⟩
| ⟨*opcodes list*⟩ ⟨*instruction definition*⟩

⟨*type definition list*⟩→ ⟨*type definition*⟩
| ⟨*type definition list*⟩ ⟨*type definition*⟩

⟨*type definition*⟩→ ⟨*identifier*⟩ ⟨*constant*⟩ [**‘signed’**] **‘;’**

⟨*op definition*⟩→ **‘op’** ⟨*constant*⟩ [⟨*op block*⟩] **‘;’**

⟨*instruction definition*⟩→ ⟨*identifier*⟩ **‘{’** ⟨*instruction block*⟩ **‘}’**

⟨*argument*⟩→ [**‘implied’**] ⟨*argument type*⟩

⟨*argument type*⟩→ ⟨*register*⟩
  | ⟨*flag*⟩
  | ⟨*type specifier*⟩
  | ⟨*constant*⟩

⟨*type specifier*⟩→ ⟨*identifier*⟩ [pointer]

⟨*register*⟩→ **‘register’** ⟨*identifier*⟩

⟨*flag*⟩→ **‘flag’** ⟨*identifier*⟩

⟨*pointer*⟩→ **‘∗’**

⟨*constant*⟩→ ⟨*int*⟩
  | ⟨*hex*⟩
  | ⟨*oct*⟩
  | **‘(’** ⟨*constant*⟩ **‘,’** ⟨*constant*⟩ **‘)’**

⟨*identifier*⟩→ ⟨*letter*⟩ | ⟨*letter*⟩ ⟨*digit*⟩

A.3 Target mapping

The target mapping is used to define how Fanà instructions map to instructions that can be executed natively on the target architecture.

⟨*variable*⟩→ **‘%’** ⟨*digit*⟩

⟨*pointer*⟩→ **‘∗’**

⟨*type specifier*⟩→ (**‘float’** | **‘double’** | ⟨*identifier*⟩) [⟨*pointer*⟩]

⟨*argument*⟩→ **‘register’** ⟨*identifier*⟩ [**‘.’** ⟨*identifier*⟩]
  | **‘flag’** [⟨*identifier*⟩]
  | ⟨*type specifier*⟩

⟨*code definition*⟩→ [⟨*flags definition*⟩] ⟨*C source code with in-lined* ⟨*variable*⟩*s*⟩

⟨*flags definition*⟩→ **‘flags’** **‘{’** ⟨*identifier list*⟩ **‘}’** **‘;’**

⟨*type definition list*⟩→ ⟨*identifier*⟩ ⟨*constant int*⟩ [**‘signed’**] **‘;’**
  | [⟨*type definition list*⟩]

⟨*register definition list*⟩→ ⟨*identifier*⟩ ⟨*type specifier*⟩ [**‘{’** [⟨*register definition list*⟩] **‘}’**] **‘;’**
  | [⟨*register definition list*⟩]

⟨*instruction definition list*⟩→ ⟨*instruction definition*⟩
  | ⟨*instruction definition list*⟩

⟨*register definition group*⟩→ ⟨*identifier*⟩ **‘{’** ⟨*register definition list*⟩ **‘}’**
  | [⟨*register definition group*⟩]

⟨*type definitions*⟩→ **‘types’** **‘{’** [⟨*type definition list*⟩] **‘}’**

⟨*register definitions*⟩→ **‘registers’** **‘{’** [⟨*register definition group*⟩] **‘}’**

⟨*instruction definition*⟩→ ⟨*argument*⟩ **‘{’** (⟨*instruction definition list*⟩ | ⟨*code definition*⟩) **‘}’**

$\langle$*instruction*$\rangle \rightarrow \langle$*identifier*$\rangle$ **‘**{**’** $\langle$*instruction definition*$\rangle$ **‘**}**’**

$\langle$*translator list*$\rangle \rightarrow \langle$*instruction*$\rangle$
| $\langle$*translator list*$\rangle$

$\langle$*translators*$\rangle \rightarrow$ **‘types’** **‘**{**’** [$\langle$*type definition list*$\rangle$] **‘**}**’**
| **‘registers’** **‘**{**’** [$\langle$*register definition group*$\rangle$] **‘**}**’** | **‘translators’** **‘**{**’**
$\langle$*translator list*$\rangle$ **‘**}**’**

B Scripts

This section contains several Bricoleur and Fanà sample scripts.

B.1 Example of Bricoleur scripts

Example of Bricoleur code emulating parts of a Motorola 6502 microprocessor:

```
/* These are the available flags for M6502 */
flags {
    C; /* Carry             */
    Z; /* Zero              */
    I; /* Interrupt Disable */
    D; /* Decimal Mode      */
    B; /* Break Command     */
    V; /* Overflow          */
    N; /* Negative          */
}

/* These are types we will be using */
types {
    uint8   8; /* unsigned 8-bit type */
    uint16 16; /* unsigned 16-bit type */
}

/* The registers available in the M6502 */
registers {
    /* General purpose registers (integer) */
    gpu {
        A uint8; /* register A is of uint8 type */
        X uint8; /* register X is of uint8 type */
        Y uint8; /* register Y us if uint8 type */
    }
}

/* ADC (add with carry) opcode definition for M6502 */
opcodes {
    /* Add to accumulator with carry using pointer fetched from a table */
    adc {
        op 0x61 { uint8, implied register gpu.X };
        op 0x71 { uint8, implied register gpu.Y };
        flags { C, Z, V, N };
        func {
               uint16 * addr;
               add uint16 addr, %1, %2;
               load register gpu.A, addr;
        };
    }
}
```

Example of Bricoleur code emulating parts of a PowerPC microprocessor:

```
flags {
    LT; /* Less Than        */
    GT; /* Greater Than     */
    EQ; /* Equal            */
    SO; /* Summary Overflow */
    CA; /* Carry            */
    OV; /* Overflow         */
}

types {
    /* Name of type followed by size in bits */
    reg     5; /* 5-bit unsigned type */
    uint32 32; /* 32-bit unsigned type */
}

registers {
    gpu {
       r0  uint32;
       r1  uint32;
       r2  uint32;
       r3  uint32;
       r4  uint32;
       r5  uint32;
       r6  uint32;
       r7  uint32;
       r8  uint32;
       r9  uint32;
       r10 uint32;
       r11 uint32;
       r12 uint32;
       r13 uint32;
       r14 uint32;
       r15 uint32;
       r16 uint32;
       r17 uint32;
       r18 uint32;
       r19 uint32;
       r20 uint32;
       r21 uint32;
       r22 uint32;
       r23 uint32;
       r24 uint32;
       r25 uint32;
       r26 uint32;
       r27 uint32;
       r28 uint32;
       r29 uint32;
       r30 uint32;
       r31 uint32;
    }
}

opcodes {
    /* Add to accumulator with carry updating CR0 register */
    addc. {
        /**
         * Opcode is identified by the integer value 31 stored in the first
         * 6 bits. Then there is the rD, rA, and rB arguments which are five
         * bits in length each. Followed by a single 0-bit, the value 10
         * stored by nine bits, and finally 1.
         * 6 + 5 + 5 + 5 + 1 + 9 + 1 = 32 bits
         */
        op (31, 6) { (register gpu, 5), (register gpu, 5), (register gpu, 5),
                     (0, 1), (10, 9), (1, 1) };
        flags { CA, LT, GT, EQ, SO };
        func { add uint32 %1, %2, %3; };
    }
}
```

The 32-bit general purpose registers of the IA32 can be broken down into one 16-bit registers which can be further broken down into two 8-bit registers. This is what the definition looks like.

```
types {
    /* Name of type followed by size in bits */
    uint32 32; /* 32-bit unsigned type */
    uint16 16; /* 16-bit unsigned type */
    uint8   8; /* 8-bit unsigned type */
}

registers {
    gpu {
        eax uint32 {
            ax uint16 {
                ah uint8;
                al uint8;
            }
        }
        ecx uint32 {
            cx uint16 {
                ch uint8;
                cl uint8;
            }
        }
        edx uint32 {
            dx uint16 {
                dh uint8;
                dl uint8;
            }
        }
        ebx uint32 {
            bx uint16 {
                bh uint8;
                bl uint8;
            }
        }
    }
}
```

B.2 Example of translation mapping

Example mapping of Fanà instruction “mov” to PowerPC binary code generators.

```
/**
 * These macros are not part of the translation mapping. They are used to
 * construct the binary code for PowerPC instructions and are provided here
 * to make it easier to understand how the mapping works.
 */

/* Store the 32-bit value x in pointer p and increase the pointer */
#define EMIT32(p, x) \
    *((uint32_t *) (p)) = (uint32_t) (x); \
    (p) += 4

/* Mask and shift the specified operand */
#define GEN_OPER(oper, val) \
    (((val) & ppc_operands[(oper)].mask) << ppc_operands[(oper)].shift)

/* Generate the opcode for operation op using two arguments */
#define GENOP2(op, oper1, oper2) \
    (ppc_opcodes[(op)].opcode | \
    GEN_OPER(ppc_opcodes[(op)].operands[0], (oper1)) | \
    GEN_OPER(ppc_opcodes[(op)].operands[1], (oper2)))

/* Generate the opcode for operation op using three arguments */
#define GENOP3(op, oper1, oper2, oper3) \
    (ppc_opcodes[(op)].opcode | \
    GEN_OPER(ppc_opcodes[(op)].operands[0], (oper1)) | \
    GEN_OPER(ppc_opcodes[(op)].operands[1], (oper2)) | \
    GEN_OPER(ppc_opcodes[(op)].operands[2], (oper3)))


types {
    uint32 32; /* 32-bit unsigned type */
}

registers {
    gpu {
       r0  uint32;
       r1  uint32;
       r2  uint32;
```

```
        r3  uint32;
        r4  uint32;
        r5  uint32;
        r6  uint32;
        r7  uint32;
        r8  uint32;
        r9  uint32;
        r10 uint32;
        r11 uint32;
        r12 uint32;
        r13 uint32;
        r14 uint32;
        r15 uint32;
        r16 uint32;
        r17 uint32;
        r18 uint32;
        r19 uint32;
        r20 uint32;
        r21 uint32;
        r22 uint32;
        r23 uint32;
        r24 uint32;
        r25 uint32;
        r26 uint32;
        r27 uint32;
        r28 uint32;
        r29 uint32;
        r30 uint32;
        r31 uint32;
    }
}

/* This is the actual translation mapping */
translators {
    mov {
        register gpu {
            sint32 {
                EMIT32(p, GENOP3(LIS, %1, (%2 >> 16)));
                EMIT32(p, GENOP3(ORI, %1, %1, (%2 & 0xff)));
            }

            sint16 {
                EMIT32(p, GENOP2(LI, %1, %2));
            }

            register {
                EMIT32(p, GENOP3(ORA, %1, %2, %2));
            }
        }
    }
```

```
}
```

EMIT32 writes a new 32-bit instruction to the instruction pointer and then updates it to the next empty instruction slot. GENOPx are macros that create binary code for the PowerPC based on a table of instruction encodings, which has also been hand-coded and is not part of the project design.

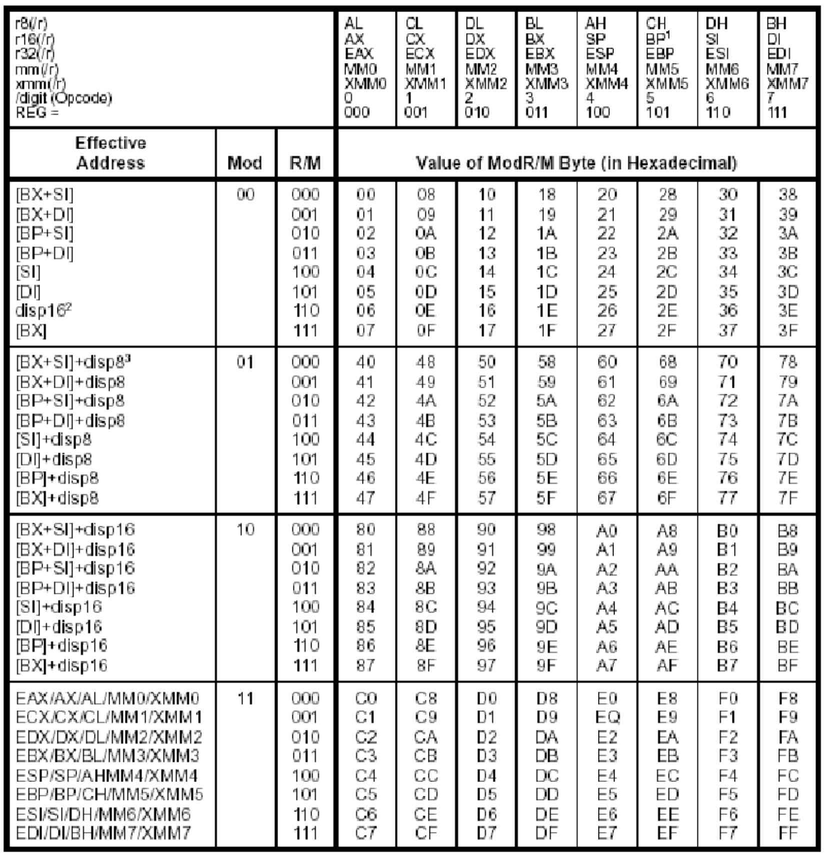
C Architecture-specific information

C.1 IA32

C.1.1 ModR/M

Some IA32 instructions use the ModR/M argument style which is a look-up table that identifies different addressing modes and registers. There is one table for 16-bit arguments and another for 32-bit arguments.

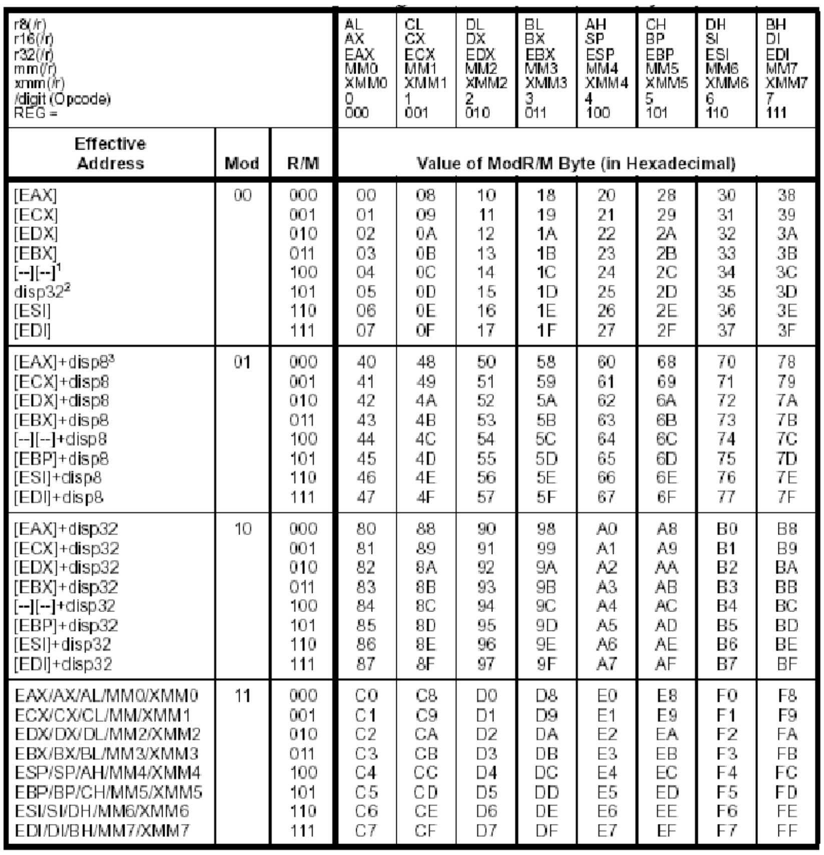
Figures 6 and 7 are taken from *IA-32 Intel Architecture Software Developer's Manual* (2001*c*, pp. 2.5–2.7).



r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
/digit (Opcode)
REG =
			AL
			AX
			EAX
			MM0
			XMM0
			0
			000
				CL
				CX
				ECX
				MM1
				XMM1
				1
				001
					DL
					DX
					EDX
					MM2
					XMM2
					2
					010
						BL
						BX
						EBX
						MM3
						XMM3
						3
						011
							AH
							SP
							ESP
							MM4
							XMM4
							4
							100
								CH
								BP[1]
								EBP
								MM5
								XMM5
								5
								101
									DH
									SI
									ESI
									MM6
									XMM6
									6
									110
										BH
										DI
										EDI
										MM7
										XMM7
										7
										111

Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16[2]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	EQ	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AHMM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Figure 6: 16-bit ModR/M table



r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
/digit (Opcode)
REG =
			AL
			AX
			EAX
			MM0
			XMM0
			0
			000
				CL
				CX
				ECX
				MM1
				XMM1
				1
				001
					DL
					DX
					EDX
					MM2
					XMM2
					2
					010
						BL
						BX
						EBX
						MM3
						XMM3
						3
						011
							AH
							SP
							ESP
							MM4
							XMM4
							4
							100
								CH
								BP
								EBP
								MM5
								XMM5
								5
								101
									DH
									SI
									ESI
									MM6
									XMM6
									6
									110
										BH
										DI
										EDI
										MM7
										XMM7
										7
										111

Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Figure 7: 32-bit ModR/M table