Compiled simulation of a vector VLIW processor

Visscher, L.

Award date:
2013

Thesis

Compiled Simulation of a Vector VLIW Processor

L. Visscher

August 16, 2013

Abstract

ST-Ericsson has developed a processor architecture, called EVP, which is currently being used in cellular modems. For the next generation of products, some changes to this architecture are necessary. The simulator that is currently being used to simulate the EVP makes architecture exploration a cumbersome process. In this thesis we present a new design and implementation for a simulation system for the EVP, based on the principle of compiled simulation. It facilitates rapid architecture exploration, due to the fact that it does not require an instruction encoding, assembler, and linker to be implemented. Yet it achieves the same accuracy as the current simulator and improves simulation speed by two orders of magnitude.

Graduation committee:
prof.dr.ir. C.H. van Berkel
prof.dr. H. Corporaal
dr. A. Turjan

Technische Universiteit Eindhoven University of Technology

ST ERICSSON

Contents

Chapter 1

Introduction

ST-Ericsson develops wireless products, which are used by other mobile device manufacturers as part of their products. The cellular modems developed at ST-Ericsson are implemented partly in hardware and partly in software. This design meets the performance and energy demands of a mobile communications device, but retains enough flexibility to allow the software implementation of a variety of communication protocols. The programmable core employed in those cellular modems is called the Embedded Vector Processor (EVP). It is a VLIW processor capable of launching multiple vector operations per cycle.

A simulator is available for the EVP. It can help a programmer with the debugging of an EVP application, by eliminating the need for a physical EVP core. But it can also assist in the verification of assembly code produced by the EVP compiler, which is developed in-house as well. For use in early architecture exploration it is less suited. The simulator takes EVP binaries as input, which means that an assembler and linker must also be available. Besides that, an encoding for the instruction set has to be developed, which can be a challenging task when dealing with a VLIW architecture. All of these have to change when a modification is made to the instruction set architecture, something that can happen quite frequently early in the design project. The cumbersome process that the existing simulator requires in order to perform architecture exploration has proven to be a major drawback.

1.1 Goals

The main goal of this project will be to create a simulator for the EVP that allows rapid architecture exploration, i.e. a simulator that requires only the definition of an instruction set, not its encoding, nor the implementation of an assembler and linker.

As a secondary goal we will aim to improve simulation speed. In this respect the existing simulator has been disappointing. To achieve this goal, the new simulator has to operate according to a method called compiled simulation, which will be explained in Section 2.1.3.

Furthermore, the simulator should be accurate, i.e. the cycle counts it reports should be within a small margin of what the current simulator reports. The use of the C standard library, and other libraries, should still be possible. Where possible, code should be reused from the existing simulator and compiler in order to minimize the implementation effort.

Chapter 2

Background

2.1 Simulation

Simulators are a vital tool in any processor design project [1, p. 254]. They are used for a variety of purposes, including architecture exploration and verification of designs. A simulator also allows hardware and software designers to work in parallel; without a simulator the development of hardware and software would be sequentially constrained. Not only does this co-development of hardware and software shorten the time-to-market, it can also potentially lead to a better overall system.

In order to enable such a broad scope of applications, simulators can be implemented on several levels of abstraction. Given the level of abstraction, there is still a choice of several techniques that can be used to implement a simulator. In general, a low-level simulation can be more accurate, but is slower than a high-level simulator, although performance also depends very much on the chosen simulation technique.

2.1.1 Abstraction levels

If we try to classify simulators based on their level of abstraction, we can distinguish four levels [1, pp. 257–258]:

- instruction-level simulation
- cycle-level simulation
- gate-level simulation
- physical simulation

The bottom two levels, gate-level and physical, represent very detailed simulations of hardware. Simulators that operate at these levels can be used by hardware designers in the design and verification of (parts of) a processor. They will not be used to simulate the execution of entire applications, simply because it would required too much time.

The top two levels provide enough abstraction from hardware details to allow the simulation of entire applications. Cycle-level simulation still models all stages of the pipeline, hence it is able to detect resource conflicts and could be used to verify whether a given instruction scheduling is feasible. Instruction-level simulators operate with a granularity of single machine instructions, which they basically treat as atomic actions. They can be employed very early in the design process for architecture exploration, assisting in the design of an efficient instruction set architecture (ISA).

2.1.2 Simulation techniques

The required level of abstraction partly determines the simulation technique to be used, but for each level of abstraction there are several possibilities, and some techniques are applicable to

multiple abstraction levels. Here we will limit ourselves to the discussion of simulation techniques that are applicable to instruction-level simulation. From now on we will often use the terms *host* and *target* to refer to the machine running the simulation, and the machine being simulated, respectively.

Interpretive simulation is arguably the most intuitive approach. An interpretive simulator is a piece of software that reads a target binary as input data, and then decodes and executes the target machine instructions one at a time. This approach is very flexible, but also quite slow.

Several techniques have been developed to improve simulation speed, but each of them introduce one or more limitations. A common idea they all share is to reduce the overhead of the decoding step. Both *compiled simulation* and *direct binary translation* take a two-step approach; first translate the entire program, which can then be executed natively on the host. These techniques will be discussed more extensively in the next section.

The last technique, called *source-level simulation*, is relatively new. It takes a three-step approach. The first is timing analysis, to determine for each basic block in the target binary the number of cycles it would take to execute it on the target machine. Next, each basic block is mapped onto a set of lines in the original source code. This step is non-trivial, since compiler optimizations can radically change the structure of the program, but a lot of progress has been made in this area and such a mapping seems to be feasible in most cases [6]. Finally, using this mapping, the timing information is annotated as instrumentation code into the original source code, which can then be compiled and run on the host machine.

2.1.3 Compiled simulation

Interpretive simulators spend a significant fraction of the time on decoding target machine instructions. Compiled simulation attempts to improve upon this by adopting a two-step approach; first the entire target program is translated into a program that can run on the host, which is then executed (natively) by the host machine. This means that instruction decoding at run-time concerns only instructions native to the host, hence the decoding hardware of the host can be utilized, which performs much better on this task than any software routine.

The translation process can be either directly from target machine code into host machine code, or via an intermediate representation in a high-level language. The former strategy is known as direct binary translation, the latter as compiled simulation. There are two advantages to using a high-level intermediate representation. Firstly, the translation process is independent from the host machine, since a regular compiler can be used to compile the intermediate representation into host machine code. Secondly, the host compiler can perform optimizations which will result in additional speedup, as compared to direct binary translation.

Traditional compiled simulation [2] takes binary executables as its input, although other approaches have also been tried, for example using object files [4]. We will take this approach even further, using the high-level source code of target applications as input. This circumvents some of the problems that are encountered when trying to translate machine code into a high-level programming language, which will be discussed shortly.

Terminology

The literature on compiled simulation does not seem to have agreed upon a standard terminology. Different terms are used for the same concept and, even worse, the term *compiled simulator* is used for completely different concepts. Therefore, before proceeding with the discussion on compiled simulation, some terminology has to be defined.

Simulation compiler The piece of software that performs the translation process. It takes its input from the target compiler toolchain; we do not care at which point, i.e. whether it is source code, binary executables, or any intermediate representation.

Simulator code The output generated by the simulation compiler — source code in a high-level language of choice.

Compiled simulator The end result of the compiled simulation toolchain, obtained by passing the simulator code through a compiler for the host platform. It is a host executable that simulates the execution of a specific application on a specific platform.

Program translation

The basic idea behind compiled simulation is to replace each machine instruction with a function call to a so-called *semantic function* that implements the semantics of the machine instruction. This idea can be applied straightforwardly to arithmetic and memory operations, but instructions that modify the control flow of the program introduce some complications.

Branches in machine code are not easily mapped onto high-level constructs such as loops and if-statements; in some cases it might even be impossible due to compiler optimizations [1, pp. 263–264]. An obvious solution is to use `goto` statements. In order to support computed branches, we could label each and every instruction with the address it would have had on the target machine. Such an excessive amount of code labels severely hampers the compilers ability to analyze control flow in the program, thereby reducing the amount of optimization that can be performed. Furthermore, if function calls are translated into `goto` statements as well, the entire program is translated into a single huge function. Since the running time of a lot of compiler optimizations depend super-linearly on function size [4], this approach could lead to impractically long compilation times. Still, compiled simulators are often implemented as one huge `switch` statement [2].

In order to simulate the execution of instructions, we must also simulate the memory and registers of the target machine. The simplest approach to simulating the data memory of a target machine is to create a single large array and use the target data memory addresses as indices in this array. However, this introduces additional address computations for each memory access. A direct mapping between simulated target memory and host memory could be established during the translation phase of compiled simulation, thereby increasing the performance at run-time. Note that this may make it much harder to track target data addresses during simulation, which might be a requirement for the simulator.

Registers can be simulated by means of a set of global variables. An optimizing compiler should be able to map the most frequently used of these variables onto host registers, enabling very efficient simulation. Some difficulties can arise when the target architecture contains aggregate registers, i.e. registers that can be addressed at different granularities; the content of such a register can either be stored in a single variable, or broken into its constituent parts and stored in multiple variables. The former approach requires bit masking when a single part is accessed, the latter requires multiple memory operations when the entire aggregate is accessed. Hence, it depends on the expected usage of such a register which of the approaches is most efficient.

Performance

Compiled simulators tend to require between one and 100 host instructions in order to simulate a single target instruction, which is up to three orders of magnitude faster than interpretive simulators [5]. Most of the speedup can be attributed to the reduction in instruction decoding overhead [3], but compiled simulators also make better use of instruction caching and prefetching capabilities of the host machine [1, p. 267]. The maximal speedup that can be achieved depends heavily on the similarity between the host and target instructions sets — on similar machines, a nearly one-to-one mapping of instructions might be possible.

Given the fact that the speedup of compiled simulation over interpretive simulation stems mostly from a reduction in instruction decoding overhead, the speedup that can be achieved when simulating a vector VLIW machine is expected to be more modest than for scalar architectures,

since the former can do quite a lot more work per instruction than the latter. On the other hand, the decoding of VLIW instructions is a complex task for which most general purpose processors are not well suited [1, p. 259], hence we still expect the speedup to be significant.

Limitations

The speedup of compiled simulation over interpretive simulation comes at a price; compiled simulation is much less flexible. The major restriction is that it requires static program code [3], i.e. no run-time dynamic program code and no self-modifying program code. The latter is rare [5], but the former is an important characteristic of operating systems [4], which are increasingly used in the embedded domain.

Compiled simulation is best suited for statically scheduled architectures, but it is not impossible to support dynamic scheduling [3]. This will, however, reduce the speedup over interpretive simulation, since scheduling must be performed at run-time. Unfortunately, nearly all general purpose computation systems are dynamically scheduled and this is also becoming increasingly common in microcontrollers.

2.2 Embedded Vector Processor

The Embedded Vector Processor (EVP) is an embedded VLIW processor that is used in the modems of mobile communication devices. Due to the large collection of communication standards that modern devices have to support, implementing such a modem fully in hardware is no longer an attractive option. The EVP is a more flexible solution that can execute the software implementation of many communication standards. These implementations often involve computationally intensive algorithms. The EVP provides the needed performance, and it does so within the energy budget of a mobile device.

2.2.1 Architecture

The core of the EVP contains four major components, as can be seen in Figure 2.1: a program control unit (PCU), an address computation unit (ACU), a scalar data computation unit (SDCU), and a vector data computation unit (VDCU). The SDCU and VDCU each contain several functional units (FUs) to support a variety of operations. Being a VLIW processor, each of these FUs can be operated in parallel, i.e. an instruction consists of up to 13 operations.

The vectors on which the EVP operates are short and of fixed size, not long and of variable length as is common for DSP processors. The advantage is that a complete vector operation can be executed in a single cycle. Since the basic word size on the EVP is 16 bits, and the vectors consists of 16 elements, a single vector is 256 bits wide.

The EVP has separate program and data memories, i.e. a Harvard architecture. The program memory is connected only to the PCU, but the data memory can be accessed by both the SDCU and the VDCU. To avoid a costly dual ported memory, only the VDCU has direct access to the memory; the SDCU is connected via a scalar cache.

Control flow changes can be costly on a VLIW machine. Since each instruction can consist of multiple operations, the pipeline can contain a large amount of work, all of which needs to be flushed when a branch is mispredicted. To mitigate this effect, the EVP supports several features that are commonly found on DSPs, but less often in general purpose machines: hardware loops, and predicated execution of most operations.

Closely related to predicated execution are masked vector operations. Most vector operations allow a vector mask to be supplied as one of the operands. In most cases the operation is then performed only on those elements of the vector that are selected by the mask. In some cases a vector mask is used to select between the elements of two input vectors.

In order to allow the efficient implementation of circular buffers, which are often used in signal processing, the EVP has a feature called modulo pointers. A modulo pointer is basically a triple register storing a *pointer*, a *base*, and a *size*. After initialization of a modulo pointer, the programmer can simply add offsets to this pointer. All checks that are required to keep the pointer within the bounds specified by *base* and *size* are implemented in hardware.

2.2.2 EVP-C

Programming a machine in assembly language is a cumbersome task in the case of a (super)scalar machine, but an extremely difficult task when dealing with a VLIW machine. This is mainly due to the fact that a VLIW machine is statically scheduled, i.e. the programmer has to determine which operations should be executed in parallel. A better approach is to move the burden of scheduling (and register allocation) to a compiler, and let the programmer write a sequential program in a high-level language. To this end, EVP-C was created.

EVP-C is an extension of the C programming language. A small example program is included in Appendix A.1. The added features can be categorized into three groups: EVP data types, intrinsics, and compiler directives. Intrinsics are functions that the compiler is aware of, i.e. the programmer can call such functions without providing an implementation for them, or linking to a library that contains them. It is through such intrinsics that the programmer can utilize the EVPs vector operations. The added compiler directives are used to provide the compiler with additional information that it can use to generate more efficient assembly, e.g. the programmer can specify that a loop will have a certain minimum number of iterations, which may enable its translation into a hardware loop.

2.3 GNU Compiler Collection

The GNU Compiler Collection (GCC) is a widely used compiler system; it is the standard compiler on most modern UNIX-based operating systems. Due to its open-source nature and modular architecture, it can be adapted for any new machine or programming language with limited effort. GCC consists of three major components: the front end, the middle end, and the back end [10].

The front end parses the original source files, hence for each programming language a different

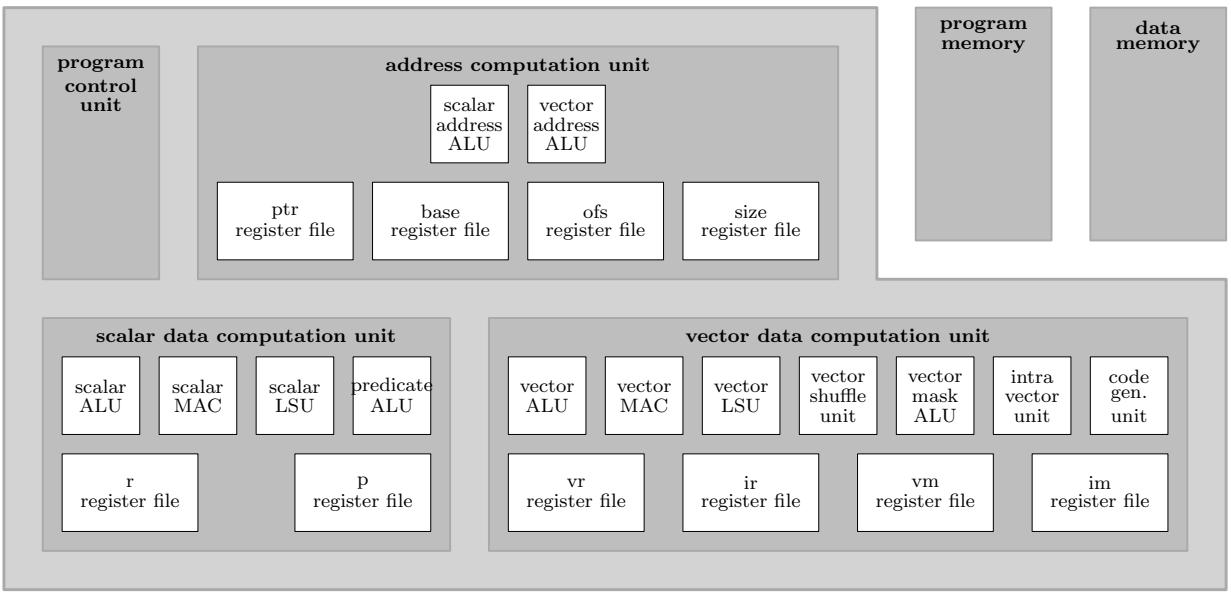

Figure 2.1: Components of the EVP core.

front end has to be created. The only job of the front end is to convert the input into a generic intermediate representation, which is then fed into the middle end.

The middle end performs most optimizations, which are aimed at increasing the performance of the final program or reduce its size. This part of the compiler is both language and target independent, which is a huge advantage when porting the compiler, since code optimization is complex. After performing all its optimization passes, the middle end converts the program into RTL format, which is another generic intermediate representation, but one that more closely resembles assembly language.

The back end performs some target dependent optimizations, but its most important tasks are register allocation and outputting the final assembly. It can be configured for a specific target by means of a machine description, which consists of two parts [9]. The first part is a file containing instruction patterns. Each instruction pattern contains an RTL fragment and a corresponding piece of assembly. These patterns are used in the final translation step — the conversion of RTL to assembly. The second part of the machine description consists of a C++ header file and a C++ source file. Together they contain all information about the target machine and the translation steps that cannot be expressed through the instruction patterns.

2.3.1 GCC port for the EVP

GCC has been ported to the EVP [12]. This required the creation of both a new front end and a new back end — a front end to process the EVP-C language, and a back end that performs optimizations specific to the EVP, does the instruction scheduling and register allocation, and finally emits EVP assembly.

Only the back end is of interest, within the scope of this thesis. And more specifically, only the final part that emits the assembly. The existing modifications will serve as a guide in creating the further modifications that make the compiler emit simulator code instead of assembly.

2.4 Newlib

An EVP application, when in its final form and running in a product setting, should perform its job quietly and without direct user interaction. However, during development of such applications the ability to output some debug messages can be a valuable tool. This is usually done through functions contained in the C standard library, which is available on many operating systems. For embedded systems that do not run an operating system, such as is the case with the EVP, an implementation of this library has to be provided.

Newlib is a lightweight implementation of the C standard library, intended for use on embedded systems. The only thing required to make it work on a particular architecture, is the implementation of a few low-level functions — a set of system calls for platforms with an operating system, or a board support package for platforms without.

Newlib has been ported to the EVP, in a way that makes it usable in conjunction with a debugger. The debugger does not run on the EVP itself, but on a general purpose machine that is connected to an EVP board. Every system call triggered by Newlib is caught by the debugger and transferred to the host machine. This allows a programmer to use a regular PC to interact with an application running on the EVP.

2.5 SuperTest

SuperTest is a commercial test and validation suite for compilers. It is used to test and validate the correctness of the VGCC compiler. The tests cover all basic language constructs as well as the C standard library. Additionally, all EVP intrinsics are thoroughly tested through a set of tests that is developed in-house.

Chapter 3

Design

The limitations of compiled simulation, which we discussed in Section 2.1.3, can largely be avoided when simulating the EVP. Instruction scheduling is static, which in the case of a VLIW processor means that the compiler checks the stream of operations for dependencies and independencies, and based on this analysis dispatches operations to FUs. This assignment does not change at run-time, which makes the EVP architecture a very suitable candidate for compiled simulation.

Furthermore, the EVP has an exposed pipeline, i.e. no interlocks, hence the compiler must explicitly insert `nop`s into the instruction stream to avoid pipeline hazards. These `nop`s will increase the cycle count during simulation run-time. Hence all static pipeline effects are accounted for, without the need for a pipeline model to be incorporated into the compiled simulators. This means that at run-time, the simulation can operate at instruction-level, which will benefit simulation speed, but its accuracy will be comparable to cycle-level simulation, due to the translation step being performed at cycle-level.

3.1 Simulation compiler

One of the first questions to address when constructing a simulation compiler, is which format to take as input. Most approaches described in the literature take their input from a late stage in the compiler chain. For various reasons that we will discuss in this section, our approach takes its input from very early in the compiler chain (see Figure 3.1). In general, using executable binaries as input has as an advantage that simulators can be generated even if the original source code is unavailable. However, this argument does not carry much weight in our case, since we are in the fortunate situation to have all source code available, including system libraries.

Not only do we have access to the source code of any EVP applications, but to the source code of the compiler as well. This leads to the interesting possibility of integrating the simulation compiler with the 'regular' compiler. However, care must be taken not to disturb the compilation process too much, since we want our compiled simulators to produce results that are identical to what they would have been when executed on an actual EVP. To this end, we modify only the GCC back end — instead of assembly we will make it emit the simulator code. This translation process will be described in the next few sections.

By using high-level source code as input for the simulation compiler, it is possible to construct compiled simulators that maintain the structure of the original program, i.e. a function in the original program code becomes a function in the simulator code. This not only speeds up the compilation process, as discussed in Section 2.1.3, but also makes the debugging of EVP applications easier, since the simulator code will be much more readable for a human programmer.

The integration of the simulator compiler within VGCC also means we have access to a lot of information about program structure and control flow, without the need for complicated analysis — this is already done for us by earlier stages of the compiler. More specifically, we can easily discern basic block boundaries, which will be of great help in the efficient handling of branches.

Since we are changing the compilation process at the latest possible stage, the translation of an application into simulator code can basically be seen as a mapping from richly annotated assembly code to C++ code. The different components of this mapping will be discussed in the next sections.

3.2 Registers

The EVPs registers will be modelled by a set of global variables. In compiled simulation of scalar architectures, it is very worthwhile to ensure that the variables representing target registers are mapped onto host registers as often as possible [1, p. 263], as this will significantly increase performance. However, given the size of the EVP vector registers, and the fact that the host processor will most likely be a scalar general purpose processor, such a mapping is impossible. Each simulated instruction will trigger memory operations on the host. But the most frequently used of these simulated vector registers will most likely remain in the lowest level of the hosts data cache, hence performance is expected not to deteriorate too much.

Most of the EVPs registers can be simulated straightforwardly with basic data types, but there are a few exceptions. The most obvious case concerns the 40-bit integer type that the EVP supports natively, but must be emulated on a general purpose host machine. Another case arises due to compound registers; the contents of some of these registers are necessarily stored non-contiguously in memory. The implementation of the different types of registers will be discussed in Section 4.3.2.

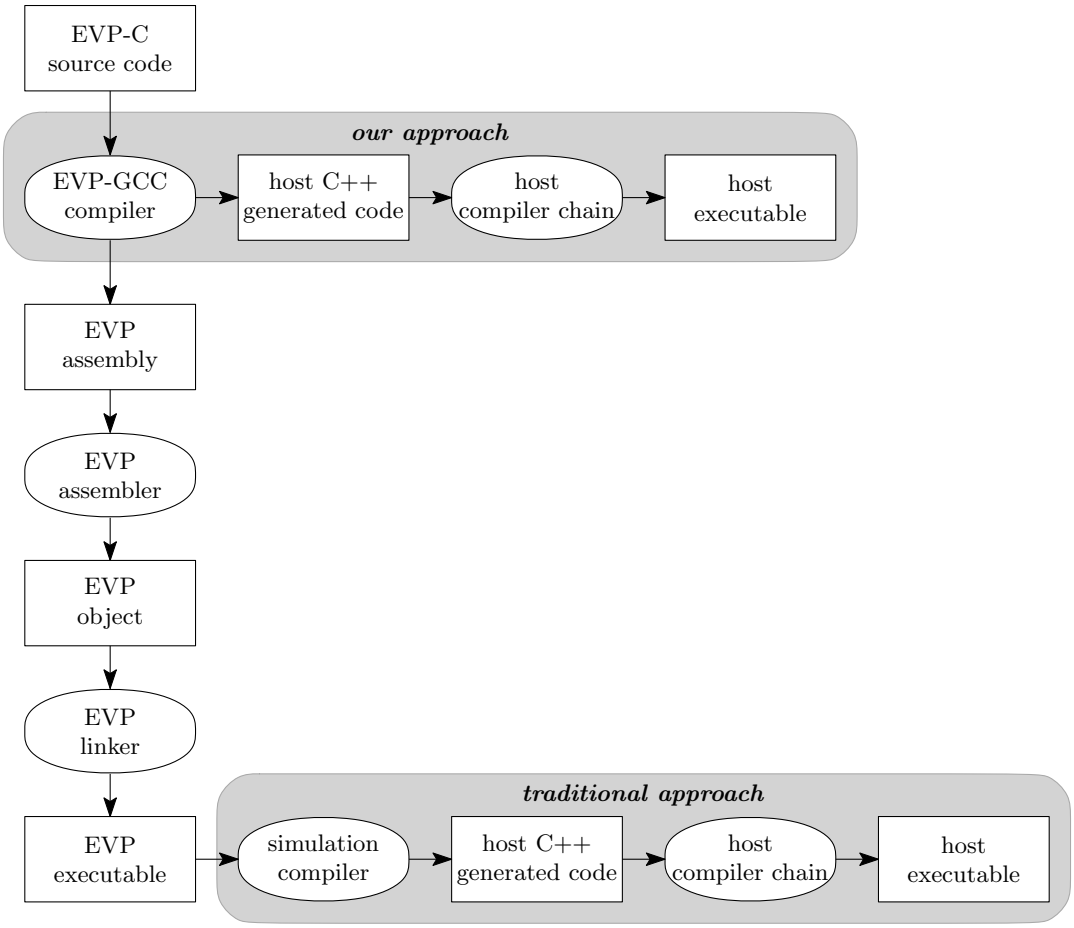

Figure 3.1: Workflow in traditional compiled simulation versus our approach.

3.3 Instructions

The basic idea of compiled simulation, as was discussed in Section 2.1.3, is to replace each target machine instruction with a call to a semantic function that can be executed on a host machine. Since we are simulating a VLIW architecture, each instruction is actually a bundle of operations. It makes sense to implement semantic functions for the individual operations rather than for all possible VLIW instructions. The simulation of a single VLIW instruction is therefore implemented by several semantic functions, executed sequentially.

Care must be taken when the individual operations are executed sequentially. Although the operations within a single VLIW instruction are independent, when executing them sequentially we might introduce dependencies. For example, if one operation reads from register `r5` and another writes to `r5` (in the same cycle), then the first operation expects to consume the old value of `r5`. Obviously, the read must be placed before the write. Luckily, the compiler had already been adapted to emit the operations in an order that allows sequential execution. This was needed for the debugger, to enable 'sub-stepping' — placing breakpoints within an instruction, not only between instructions. Hence no further reordering of the operations within instructions is required to ensure correct sequential execution.

3.3.1 Control flow changes

Most semantic functions simply perform a computation and modify the contents of registers, which is easily dealt with by passing the correct parameters. However, those semantic functions that modify the control flow of the program require more careful consideration. Even more so due to the fact that the EVP utilizes delay slots, i.e. the first few instructions right after the branching instruction are executed even if the branch is taken. This means that we have to develop some sort of mechanism that delays the execution of control flow changes.

The VGCC compiler produces assembly code where the end of delay slots is marked by a piece of comment (see Appendix A.2). We can adapt this feature to emit a macro instead. This macro executes the actual control flow change in the compiled simulator, while the semantic function corresponding to the EVPs control flow change operation only records the type and target of the branch. The implementation of this macro is discussed in Section 4.3.3. Recording the type of a branch is straightforward, but recording its target is a bit more involved in a high-level programming language.

The target of a direct branch is known at compile time; VGCC always puts a label at the target instruction and outputs the argument of the branch instruction in symbolic form, i.e. a code label. When translating a labelled instruction, we can simply keep the code label, as this is a feature that is supported in C++ as well. This means that the argument that the semantic function receives is a valid code label. However, the C++ language only allows code labels to be used in conjunction with `goto` statements — they are not values that can be assigned to variables. Luckily, GCC has an extension that allows the conversion of code labels into void pointers, which makes it possible to store the target of the direct branch in a variable.

The target of an indirect branch can in principle be any instruction and be unknown at compile time, which would seem to require every instruction to be labelled, as was discussed in Section 2.1.3, leading to reduced performance. However, the compiler produces indirect branches only as a result of switch statements and indirect calls only as a result of function pointers. In both of these cases the target is a labelled instruction and hence an approach very similar to the one for direct branches can be employed. The only difference is that the branch is executed in two steps: first a move instruction that puts the branch target into a simulated register, and then the indirect branch instruction.

Although the usage of the `goto` statement has been debated in the past [7], the arguments are geared towards the production and maintenance of program code by a human programmer. Since in this case we are dealing with generated code, most of the arguments against the usage of the

`goto` statement do not apply. A compiler is perfectly capable of making sense of a `goto`-ridden program.

Besides branches, function calls and returns are also executed with a delay. Since we are preserving the functional structure of the program, both of these cases are easily dealt with. The semantic function implementing the call instruction stores a function pointer as the branch target — there is no need to rely on GCC extensions this time. When the control flow changing macro is executed, the host performs a function call to this address, not a `goto`. This makes the semantic function for the return instruction even simpler; the branch target is already on the host stack, so we only need to record the correct branch type, not the branch target, and the macro can simply perform a return statement on the host.

A further concern when dealing with functions calls is the passing of parameters and return values. These values will be placed in the simulated registers, i.e. global variables, as part of the simulation. Hence function calls in the compiled simulator only need to execute the control flow change and therefore all functions in the simulator code have zero parameters and return `void`.

3.3.2 Hardware loops

A special type of control flow change comes in the form of hardware loops. A hardware loop is initiated by an instruction taking three arguments — two code labels, indicating the start and end of the loop body, and the number of iterations. Just like other control flow changes, the loop instruction is followed by a few delay slots. When the instructions in the delays slots have been executed, the program jumps to the start of the loop body and executes the loop the proper number of times. After the last iteration, the program continues with the instructions immediately after the loop body (instead of continuing with those after the loop instruction).

Producing simulator code for hardware loops would seem to be done most easily and efficiently using `while`-loops, but some complications arise when this is done. The piece of the program that is executed as a hardware loop, is not necessarily exclusive to this loop. For example, when control flows normally into it, it should be executed just once, not iterated. However, although such constructions are allowed by the architecture, the compiler will never produce such code — control flow can only reach the loop body through the hardware loop operation. A more pressing issue is the fact that the hardware loop operation and the loop body can be arbitrarily far apart. Translating them into a single `while`-loop would require a reordering of the input program, and hence a much larger modification of the compiler back end than for any other instruction — other instructions allow a one-to-one mapping from assembly code to C++ simulator code.

Instead of using while-loops, our solution is `goto` based. It allows for easy translation from assembly to simulator code; the hardware loop operation and the loop body can be translated separately. When the loop body is being translated, there is in fact no need to know that it will be used as such. Any sequence of basic blocks can be used as loop body, just like when the code is executed natively. The approach is very similar to the one used for delaying branches. The semantic function only records the proper information (start and end of the loop body, and number of iterations), but does not change the control flow. This is done by macros; one at the end of the delay slots, that causes the jump to the start of the loop, and one at the end of the loop body that causes the iterating behaviour. Note that this could mean that we have to place a macro in a piece of code that has already been translated — the loop body can in principle be placed before the hardware loop instruction. To avoid this, a loop end macro is emitted right before each code label. This might seem redundant and inefficient, but it can be implemented in such a way that the unnecessary macros will be optimized away completely when the simulator code is compiled. This will be discussed in Section 4.3.3.

3.4 Memory

The EVP has separate data and program memories. It has no instructions that operate on the program memory; only the program control unit (PCU) has access to this memory. Hence, there is no need to explicitly simulate the program memory.

The data memory could be simulated by means of a single large array, with data addresses used as indices, but as discussed in Section 2.1.3 a more direct mapping would be more efficient. Since we use high-level source code as a starting point, this is quite easy to achieve. For each global variable and constant that appears in the source program we simply generate an equivalent global variable or constant in the simulator code. These variables and constants can then be used directly, without additional addressing overhead.

Note that this means that the compiled simulation of an application can, and probably will, have a different memory layout than when the same application is executed natively on the EVP. This might be problematic if, at some point in the future, there is the need to incorporate a data cache simulator in the compiled simulations [6]. This problem cannot be solved by employing a different technique for simulating data memory; it is inherent to the implementation of the simulation compiler as a modified GCC back end, since the addresses of global variables and constants are not known until the linker phase.

Having a different memory layout in simulations raises the question of what to do with pointers. Should they store target addresses or host addresses? The former requires an address conversion to be computed each time the data memory is accessed. Such a conversion is non-trivial and induces a performance penalty. The latter has several complications, each of which will be discussed in Chapter 4, but all of them can be solved without degrading performance, hence it is the approach we will use.

Remember that each EVP register is modelled by a global variable. This means we now have two sources of global variables in the simulator code. This can lead to name clashes, and hence simulator code that does not compile or, even worse, code that does compile but produces erroneous results. To prevent this, all global variables, constants, and functions receive a prefix.

Besides global variables and constants, we also need to simulate the stack — it contains local variables, and function parameters in those cases when they cannot be passed via registers. Simulation of the stack is achieved by reserving a block of host memory and properly initializing the stack pointer. The stack is accessed only through the stack pointer — the stack pointer is a special purpose register (simulation of registers is discussed in the next section). As discussed earlier, this simulated register will contain a host address, not a target address, hence accessing the stack introduces no additional overhead.

This leaves only dynamically allocated memory. EVP applications intended for use in a final product do not make use of dynamic memory. For debugging purposes the application can be linked with a ported implementation of the standard C library (see Section 3.6), which requires the presence of a heap. A block of host memory is reserved in which the heap can be created. Initialization and further management of the heap is taken care of by the standard C library itself.

3.5 Components of a compiled simulator

So far we have mostly discussed the simulation compiler and how it produces simulator code. This simulator code by itself does not compile into a fully functioning executable, it requires some additional components that are the same for every simulation. Figure 3.2 shows the objects that are linked to form a compiled simulator, and the sources from which these objects are compiled and assembled.

Besides the generated simulator code, we need an implementation for the semantic functions. We can, with some effort, reuse their implementations from the existing interpretive simulator, as will be discussed in Section 4.2. The semantic functions could be compiled into a library and then

linked with every simulator, but instead we keep them in a set of header files that is included in each simulator source file. This increases the compilation time, but allows inlining of the semantic functions which will result in a much faster execution of the compiled simulator. The semantic functions operate on simulated register, i.e. global variables, the types of which are defined in an additional header file, as are the macros that are used throughout the generated simulator code. All these header files combined with a simulator source file compile into a simulator object. The final executable (the compiled simulator) can be composed of several such objects; each source file of the original EVP application will compile into a single simulator object.

The simulator objects need to be linked with an auxiliary object in order to form a fully functional executable. This auxiliary object provides the simulation environment, i.e. the global variables that simulate the EVPs registers and an array that can hold the simulated stack. It also contains the `main` function, which first initializes the registers and then calls the EVP startup routine (it is at this moment that compiled simulation actually starts). Furthermore it contains some helper functions, e.g. to produce output once the simulation has finished, and to handle system calls while the simulation is running.

Finally, the compiled simulator can be linked with libraries, as will be discussed in Section 3.6. The objects contained in these libraries are created in the same way as the simulator objects we discussed earlier in this section. Library functions are executed as part of the simulation, and hence add to the cycle count.

3.6 Libraries

One of the limitations of compiled simulation, which we discussed in Section 2.1.3, is the lack of support for dynamic linking. This would seem to exclude the possibility to use libraries. However, due to the fact that we preserve the functional structure of programs, dynamic linking is possible with our approach. The only remaining difficulty in the usage of libraries, is the passing

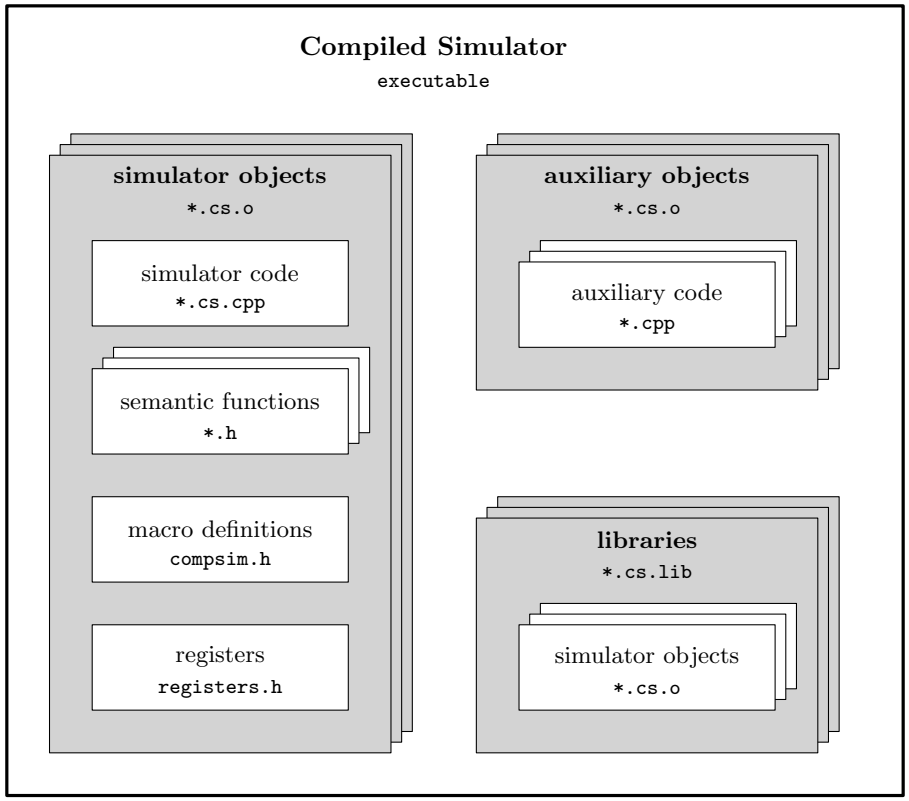

Figure 3.2: The various parts of which a compiled simulator consists.

of parameters and return values. This happens through the simulated registers and simulated stack, but an arbitrary library would expect these values on the host stack. We either need to provide a mechanism that translates between the two calling conventions, or we could produce a compiled simulation version of the library. The latter is only possible if the sources of the library are available.

EVP applications in general do not make use of dynamically linked libraries, but during debugging the C standard library is often used. To enable this, Newlib has been ported to the EVP, as was also mentioned in Section 2.4. The creation of a compiled simulation version of this library requires a modification to its build process, but also the handling of system calls needs to be changed. Instead of triggering the debugger, we need to invoke a routine that passes the system call to the host. Such a system call handler can be implemented as part of the auxiliary code.

3.7 Instrumentation code

The construction of a simulator is not of much use if we cannot extract useful information from it. To this end, instrumentation code will be inserted by the simulation compiler. For now we will limit ourselves to the counting of cycles, both globally and per function. The simulation compiler will emit a macro at the start of each function and before each bundle of semantic functions to facilitate this.

The cycle counters will be incremented for each simulated EVP instruction. One might argue that it is more efficient to aggregate these increments into a single addition per basic block, which is possible since the EVP is statically scheduled. However, the instrumentation code is part of the simulator code, which is passed through an optimizing compiler to produce a compiled simulator. Hence the compiler will take care of this aggregation of cycle counter increments, thereby minimizing the overhead induced by the instrumentation code.

Chapter 4

Implementation

4.1 Compiler modifications

The design of our compiled simulation system is such that we do not have to make very invasive modifications to the compiler in order to make it generate simulator code instead of assembly. It is at the latest possible stage that we intervene; wherever the compiler emits a piece of assembly, we modify the output statement to emit a corresponding piece of simulator code instead. To be more precise, we include a conditional that selects between the two possible outputs, based on a new command-line option that we added to VGCC. The addition of the new command-line option required only the addition of a single entry to the option definition file of GCC [9]. The addition of the command-line option means that the same compiler that is used for regular EVP compilation jobs can be used for compiled simulation, simply by adding a single flag to the command-line. The remaining modifications can be categorized into two groups: modifications to the machine description files, and modifications to the C code of the compiler back end. We will discuss both categories in the next two sections.

4.1.1 Machine description

The mapping from RTL, the final internal representation used by the compiler, to assembly code is governed in a large part by the instruction patterns defined in so-called machine description files. Such instruction patterns consist of four or five parts:

1. An optional name.
2. An RTL template. The compiler will try to match this template with parts of the program that is being compiled. If a match is found, the instruction pattern may be applicable.
3. A condition, in the form of a small piece of C code. When the RTL template matches, this condition has to be passed as well in order for the instruction pattern to be applied.
4. An output template. Either in the form of string or a small piece of C code, this determines what should be emitted to the assembly file.
5. An optional attribute vector, which is basically used to contain any additional information that is needed in the compilation process.

Only the output template is of interest to us. These used to be simple strings, e.g. `"vadd32 %0, %1, %2, %3 %#"` in the complete example below, but those were replaced by small pieces of C code selecting between two strings. One of the strings being the original assembly output, the other the semantic function call. The naming of semantic functions is slighly different from the assembly instructions, as will be explained in more detail in Section 4.2. Predication is handled differently in compiled simulation; instead of passing a predicate argument to the semantic function, an `if`-statement is put in front of the function call. This can lead to a more efficient

simulator, since in cases where the predicate is false a function call is eliminated, but it also simplifies the implementation of the semantic functions.

```
(define_insn "INSN_BUILTIN___po_vadd_32"
[
   (set
      (match_operand:EV1 0 "vr_register_operand" "=REG_VR_")
      (unspec:EV1
      [
         (match_operand:EV1 1 "vr_register_operand" "REG_VR_")
         (match_operand:EV1 2 "vr_register_operand" "REG_VR_")
         (match_operand:BI 3 "predicate_register_operand" "REG_P__")
         (match_operand:EV1 4 "vr_register_operand" "0")
      ] UNSPEC___BUILTIN___po_vadd_32)
   )
]
  ""
  "*{ return (evp_flag_comp_sim ? \"if (%3) vadd32_vrD1_vrA1_vrB1( %0, %1, %2 ); %#\"
                                : \"vadd32 %0, %1, %2, %3 %#\"); }"
  [
   (set_attr_alternative "corebits"     [(const_int corebits__vadd32_vrD1_vrA1_vrB1)])
   (set_attr_alternative "ioshape"      [(const_int ioshape__vadd32_vrD1_vrA1_vrB1)])
   (set_attr_alternative "mpvar"        [(const_int mpvar_P)])
   (set_attr             "opcodesize"   "3")
   (set_attr             "predicable"   "no")
   (set_attr_alternative "shape"        [(const_int shape__vadd32_vrD1_vrA1_vrB1)])
  ]
)
```

There are a great number of these instruction patterns, but luckily not many of them are written by hand. Most of them are generated from a database by means of a Perl script. This script was modified to produce output as shown in the example above.

4.1.2 Back end code

The outputting of all assembly constructs besides machine instructions, e.g. data sections or labels, is dealt with by GCC via a set of macros and hooks. Many of them were changed, some were added, but all of it comes down to producing conditional output very similar to what we did with the instruction patterns. The only change required to mainline GCC concerned the assembling of the contents of variables and constants, i.e. initialization. For some reason there were no macros or hooks available to modify the behaviour of this procedure.

The modifications discussed so far cover the entire mapping from assembly to simulator code, but there remains one final modification to be discussed.

Secondary output file

In the previous chapter we discussed the translation of EVP-C programs into simulator code in terms of a mapping from EVP assembly to equivalent C++ code. This mapping has been designed in such a way that we did not have to reorder or duplicate instructions, which worked perfectly fine when translating the program text. However, assembly code can also contain data sections, which have to be translated into global variables in the simulator code. In C++, variables (and functions) have to be declared before they are used, but in assembly code one can refer to a label that appears later in the code.

An initial solution was to modify some parts of mainline GCC, and the call graph module [11] in particular, in order to emit all variables and constants before functions and to disable the reordering of functions. The idea behind this was to completely eliminate the need for forward declarations. However, problems could still arise when the initializer of a variable referred to a variable coming after it. And at a later stage it also turned out that GCC emits `import` directives, used to import external symbols when the program consists of more than one compilation unit, at the very end of the assembly file and that this was very difficult to change. Hence this initial solution was discarded in favor of a more elegant one.

A limited, but sufficient, form of reordering can be achieved through modification of the back end code alone. Instead of writing output to a single file, the simulation compiler produces two files; one source file that contains function definitions and variable initializations, and a header file that contains declarations for all of them. This approach introduced a new problem, caused by the fact that forward declarations of static variables are not possible in C++.[1] This problem was solved using anonymous namespaces, as will be discussed in Section 4.3.1. Using this approach it is again possible to produce simulator code that closely matches the assembly code, and without requiring modifications to mainline GCC.

4.2 Reuse of semantic functions

One of the major components of a compiled simulator, besides the generated simulator code, are the semantic functions. We need about 1500 of them; starting from scratch and implement all of them is not feasible within the time constraints of this project. The pre-existing interpretive simulator obviously contains an implementation of all the semantics required to simulate the EVP. Reusing those has two benefits: it can save us a lot of time, and we know this implementation to be correct. Unfortunately, it has some downsides as well. The interpretive simulator contains a reduced set of more generic semantic functions. We will need to provide a mapping between all possible EVP operations and this set of semantic functions. Furthermore, we must ensure that we simulate the EVPs registers in a way that is compatible with how it is done in the interpretive simulator. Even so, reusing the semantic functions is preferable to implementing them from scratch.

The reduction in the number of required semantic functions was achieved in two ways. Firstly, many vector operations can be described as the application of scalar semantics to each element of the vector, e.g. vector addition. Semantic functions for such vector operations are left out. Secondly, some operations are the composition of two or more simpler operations, e.g. load/store operations that also update the address operand. Such operations also do not need to be explicitly present in the set of semantic functions. But then we need a mapping from EVP operations to semantic functions that can express the iteration or composition of these basic functions. Such a mapping was already available, in the form of a set of wrapper functions. These wrappers were originally intended for verification purposes, they allow the semantic functions to be tested in isolation. Unfortunately, they were not created with efficiency in mind, and even worse, they were untested. Hence they proved to be a rich source of bugs. Debugging these wrappers and improving their efficiency was, in all likelihood, less time-consuming than implementing the semantics from scratch.

4.3 Auxiliary code

So far we have discussed the modifications made to the VGCC compiler to make it produce simulator code, and how we provided implementations for the semantic functions. What remains to be discussed is the implementation of the auxiliary code, i.e. the code that implements the simulation environment and the macros that were discussed in Chapter 3.

4.3.1 Declarations and scope

In C++ we have functions, variables, and constants. In assembly we only have labels — the type of section in which the label occurs determines what kind of object we are dealing with. In C++ the scope of a variable is known from the moment it is declared. In assembly a label can be exported by a later export directive. These labels can be used as operands of instructions, and their values

[1] A forward declaration is only possible by using the `extern` keyword, implying that the forward declared object has external linkage. The `static` keyword implies internal linkage. Combining the two leads to a contradiction.

can be stored in registers, since each label represents an address, either in instruction memory or data memory. Their counterparts in C++, however, are of different types, i.e. straightforward assignment would produce compiler errors. All in all there are a few difficulties in the translation process that need to be overcome.

The homogeneity of assembly labels is emulated by introducing an additional variable along with each function, variable, or constant. This companion variable is a plain integer and stores a typecast pointer to its associated object. It is this integer companion that will be used as operand of instructions, i.e. as parameter of semantic functions.

A label in assembly that is not exported, is local to the compilation unit. The most obvious way to model this in C++ is to make the variable or function `static`. However, a forward declaration is not possible for `static` variables, and we needed forward declarations to avoid the need for reordering variables and functions. The problem is solved by putting every function, variable, and constant in an anonymous namespace, which has the same effect as making them `static`,[2] but allows forward declarations. We can later change their scope, i.e. make them externally visible, by introducing an additional variable outside the anonymous namespace that is a copy of the integer companion. They assembly import directive is now also easy to implement by means of a macro that creates an integer companion inside the anonymous namespace and initializes it with the value of the externally visible copy. Note that the underlying variables are always local and are only accessed through pointers. This form of indirect memory access might seem inefficient, but the pointers used to access them never change after initialization, hence the compiler should be able to optimize them away in many cases.

What we have discussed so far holds true for global variables, i.e. objects that will be allocated on the heap. The translation of variables local to a function, i.e. the ones that will be put on the stack, is a different story. The translation of them is the easiest of all, since we have to do absolutely nothing at all in this case. They are represented by a numeric offset with respect to the stack pointer, not by a symbolic label. As long as the stack pointer is properly initialized to the base of the array that holds the simulated stack, all address computations on the stack will yield valid host addresses.

4.3.2 Register types

Scalars

The variables that simulate the EVPs registers will be used frequently, hence the efficiency of their implementation is an important factor in achieving a fast simulator. For example, the predicate register file can modelled using the basic data type `bool`. Combined with the inlining of semantic functions, this allows most operations on predicates to be translated into a single host machine instruction.

Similarly, we would like to represent the scalar register file with `int`s. However, already with this simple case there are a few complications. The scalar register file consists of 32 registers, each 16 bits wide, but it can also hold 32-bit values, stored in a pair of adjacent registers with the constraint that the first register is an even numbered register. This means that we can simulate this register file efficiently by means of a `union` construct.

```
union reg_file_scalar {
    uint16_t hi[32];
    uint32_t si[16];
    uint64_t di[8];
};

extern reg_file_scalar r;
```

[2]Before C++11, names in an anonymous namespace actually had external linkage, but were given a unique prefix at compile time, achieving the same effect. Since C++11 names in an anonymous have internal linkage by default.

```
static uint16_t& r0 = r.hi[0];
...
static uint32_t& r1r0 = r.si[0];
...
static uint40_t r2r1r0(r.di[0]);
...
```

The scalar register file can also store 40-bit values in three adjacent registers, where the number of the first register must be a multiple of four. So as long as we take care not to modify the upper bits, the `union` approach can still be used by adding an array of 64-bit integers. The protection of the upper bits is guaranteed by means of a small wrapper class, that stores a reference to the 64-bit integer and overloads the assignment operator and typecast operator.

Vectors

EVP vector registers have a fixed width of 256 bits, but they can be used as a vector of 32 elements of 8 bits, 16 elements of 16 bits, or 8 elements of 32 bits. This behaviour can be simulated in the exact same way as we did with the scalar register file. A more complicated situation arises when we consider compound vectors — the combination of two or three adjacent vector registers, much like in the scalar register file. The layout of for example a double vector, that stores 16 elements of 32 bits each, is such that the lower bits of each element are stored in the first vector and the upper bits in the second vector. Hence, the value of a single element of the vector is not contiguously stored in memory, but is split into two parts, that are exactly one vector length apart.

An obvious way to simulate the memory layout of vectors would be to implement them as a class with getter and setter methods for the vector elements. However, due to the reuse of the semantic functions, our implementation of the vector registers must be compatible with the one used in the interpretive simulator. This means that the vector class is expected to have a getter method for vector elements that returns an object. This object should behave as if it is a regular scalar value. Hence we need implementations of the compound versions of each integer type, i.e. 16-bit, 32-bit, 40-bit, and 64-bit. The classes that implement this are very similar to the one implementing the 40-bit integer that was needed for the scalar register file. Each class stores two pointers and overloads the assignment operator and the typecast operator.

```
struct uint16c_t // compound 16 bit unsigned integer
{
    uint8_t *lsw, *msw;

    uint16c_t(uint8_t* lo, uint8_t* hi): lsw(lo), msw(hi) {}

    void operator=(const uint16_t& val) const {
        *lsw = uint8_t(val & 0xFF);
        *msw = uint8_t(val >> 8 & 0xFF);
    }
    operator uint16_t () const {
        return uint16_t(*lsw) | uint16_t(*msw) << 8;
    }
};
```

4.3.3 Delayed control flow changes

As was discussed in Section 3.3.1, we need a macro that can execute control flow changes. Two global variables are introduced to store the type and target of a control flow change; these variables are set by the semantic functions. The macro can then be implemented with a `switch` statement, each case executing a different type of control flow change. In most cases the value of `cfc_type` can be determined at compile time, due to the fact that the semantic functions are inlined. This means that the compiler can optimize away the `switch` statement, leaving only a single control flow changing instruction. All but one of these instructions are very simple; only the case of

function calls looks complicated, but it is just the variable `cfc_target` being cast to a function pointer and then immediately called.

```
static enum {JUMP, RETURN, CALL, CONTINUE} cfc_type = CONTINUE;
static void* cfc_target;

#define DELAY_SLOT_END                                                    \
  switch (cfc_type)                                                       \
  {                                                                       \
      case JUMP:    cfc_type = CONTINUE; goto *cfc_target;                \
      case RETURN:  cfc_type = CONTINUE; return;                          \
      case CALL:    cfc_type = CONTINUE; (*((void(*)())cfc_target))();    \
      default:      /*CONTINUE*/;                                        \
  }
```

Note that this mechanism for delayed control flow changes is only correct if branching instructions do not occur within the delay slots of a previous branching instruction; the macro belonging to the first branch would execute the control flow change specified by the second branch. The instruction set architecture allows a branch to be placed in a delay slot of a previous branch if and only if the two branches are predicated and those predicates are mutually exclusive [8, p. 52], but it is possible nonetheless. However, the VGCC compiler currently never produces such code, hence support for this was not (yet) needed. If future modifications of the compiler make the generation of such code possible, then the situation could be remedied by not keeping the cfc type and target as single variables, but as queues instead.

Hardware loops

When we discussed hardware loops in Section 3.3.2 we came to the conclusion that we needed two macros, besides the semantic functions of course. The first macro to jump to the start of the loop body, and a second macro at the end of the loop body to control the iterating behaviour. The jump to the start of the loop body can be implemented by reusing the mechanism created for regular branches. Hence, we only need to implement the second macro.

When we are translating a piece of code, we do not know whether or not it will be used as a loop body. An easy way to deal with this is by treating the end of every basic block as a possible loop end, i.e. placing the loop macro before every label in the program. This sounds a lot more inefficient than it actually is; when implemented correctly, the compiler optimizations can remove most of the unnecessary checks.

As was the case with branches and function calls, the semantic functions for hardware loops just store some information in global variables. In this case they store number of iterations and the begin and end of the loop body, but also update a variable that keeps track of the loop depth, which is needed since nested hardware loops are possible. By making these variables `static`, i.e. contain there scope to a single compilation unit, the compiler should be able to deduce the value of `loop_depth` at compile time, since it only depends on program structure, not on any input values. This implies that, at labels that are not part of a loop, the outermost `if`-statement of the loop check macro can be statically resolved to False, leading to the entire macro being optimized away.

```
#define MAX_LOOP_DEPTH 3

static int      loop_depth = -1;
static uint32_t loop_counter[MAX_LOOP_DEPTH];
static void*    loop_begin[MAX_LOOP_DEPTH];
static void*    loop_end[MAX_LOOP_DEPTH];

#define LOOP_CHECK(LABEL)                                          \
  if (loop_depth >= 0 && loop_end[loop_depth] == &&LABEL)          \
  {                                                                \
    if (--loop_counter[loop_depth] != 0)                           \
      goto *((void*)loop_begin[loop_depth]);                       \
    else                                                           \
```

```
    --loop_depth;                                          \
  }

#define LABEL(NAME)  \
  LOOP_CHECK(NAME);  \
  NAME:              \
```

Having the loop status variables as `static` (local) variables, means that we can exceed the specified maximum loop depth by nesting loops that are in different compilation units, e.g. create a loop around a function call to a function that is defined in a different C++ file and also contains a loop nest. Since the compiled simulator will most likely be used for architecture exploration purposes, not verification, this is not seen as much of a problem. However, if needed the situation could easily be remedied by, for example, introducing an additional loop depth counter that is truly global.

Jump tables

A `switch` statement is often compiled into assembly code that utilizes a so called jump table — a small read-only data section that contains the instruction address of each case label. There are several difficulties when translating this construct into simulator code. First of all, the variable that represents the jump table must necessarily be a local variable inside a function, since its initializer will contain code labels, which are not valid outside the scope of the function. C++ does not support forward declarations of local variables, but in the assembly code the jump table is placed *after* the instruction that uses it, so it would seem that some reordering is unavoidable in this case.

Instead a solution was found that eliminates the need for a variable representing the jump table; it can be passed as a literal to the semantic function. The jump table itself is output to the header file as a macro definition, making it irrelevant if it is placed before or after the instruction using it. For example, a jump table with label L9 generates the following entry in the header file:

```
#define L9 JUMP_TABLE { &&L3, &&L4, &&L5, &&L6, &&L7, &&L8, }
```

In the simulator source file it can then be used like this:

```
move_slsu_ptrD_DMADDR32( ptr0, L9 );
```

The little helper macro `JUMP_TABLE` is introduced for readability purposes. It ensures that the list of labels that follows it is interpreted as an 'array of void pointers'-literal. The pointer to this construct is then cast to a plain integer, to avoid warnings when storing this value in one of the simulated registers.

```
#define JUMP_TABLE (int)(const void*const[])
```

Chapter 5

Evaluation

5.1 Verification

Before a simulator can be used to verify the correctness of programs, it must first be proven correct itself. Correctness of a simulator can be decomposed into two parts: functional correctness and simulation correctness. Obviously, the simulated program should produce identical output for any given input as the native execution of the program would have produced. On top of that the simulator should keep track of (some of) the internals of the simulated architecture, and these values must be correct as well.

5.1.1 Functional correctness

Having SuperTest available greatly simplified the verification of the compiled simulators produced by our simulation compiler, but it also made the testing very thorough. Configuration of the test suite involved not much more than adapting the build process of the test programs. Once this was accomplished, the entire batch of tests could be started with a single command, producing an overview of the results a while later.

Our compiled simulators successfully passed all the tests in the entire C suite, and most of the EVP intrinsics tests. The tests that failed either had their validation depend on the produced assembly code, in which case the reason for failure is obvious, or their source code contained assembly code, which is not supported by the simulation compiler, as was discussed earlier. Although even a test suite as comprehensive as SuperTest cannot guarantee with absolute certainty that our simulation compiler produces a functionally correct result in each and every case, the fact that we passed all the tests still inspires a high degree of confidence.

5.1.2 Simulation correctness

A simulator should correctly keep track of the internals of the simulated architecture, which in our case culminates in an accurate cycle count being reported. Verification of the cycle count was done by comparing the reports of the compiled simulator with that of the interpretive simulator, which is known to be accurate. The interpretive simulator was used with its data cache simulation turned off, since compiled simulation does not (yet) support this feature. In general the two simulators seem to agree, but there are two exceptions. First, there always seems to be a difference of six cycles, no matter how large or small the total cycle count. A first hypothesis to explain this was that maybe one simulator counts the delay slots after the last `return` and the other one does not. Later we found that all Newlib functions introduce deviations in the cycle count, which is most likely due to different optimization flags being used during the build of Newlib. This then might also explain the off-by-six error, since the `exit` routine, which is part of Newlib, is always called after the return of the `main` function. So, if we rebuild the compiled simulation version of Newlib

with the exact same optimization flags as were used for the native EVP version, both simulators will agree on the cycle count, down to the last cycle.

5.2 Performance

One of our goals was to improve simulation speed. It is time to see how well we did in this area. The most obvious way to evaluate this is by comparing compiled simulation with its direct competitor, the pre-existing interpretive simulator. The latter has shown some disappointing performance and due to this a revised version has been created, called `fast-sim`, that sacrifices most validation checks and introduces cached instruction decoding in order to achieve a more reasonable simulation speed. We will determine the speedup of compiled simulation with respect to this quicker version of the interpretive simulator. All reported simulation times are the medians of five runs of the simulator, on an Intel® Xeon® X5260 processor at 3.33 GHz.

5.2.1 Compiled simulation versus interpretive simulation

The EVP can execute several vector operations in parallel, but it can also run scalar-only programs, and anything in between. Scalar EVP instructions can often be mapped onto a single host instruction, and like we argued in Section 2.1.3, this yields a large performance benefit over interpretive simulation. However, as the amount of work per instruction increases, the performance of compiled simulation is expected to converge towards that of interpretive simulation. In order to test this hypothesis, two small benchmark programs were created. One purely scalar, and one consisting mostly of parallel vector operations.

The scalar program, when simulated with both simulators, showed a speedup of roughly 100 times in favor of compiled simulation, as can be seen in Table 5.1. This two orders of magnitude improvement is what is generally expected from compiled simulation. The second program, performing two vector additions per instruction, showed a speedup of only 5.6 over the interpretive simulator. This supports our earlier hypothesis.

The workload per instruction can be increased even further by means of a small adaptation to the parallel vector addition benchmark: switching from integers to the EVPs native 16-bit floating point format. This floating point format is not supported by the host machine, hence it has to be emulated. This reduced the speedup even further to only 2.4 times, which is the worst performance that has been observed.

Given this two orders of magnitude difference in speedup, depending on the nature of the program being simulated, one might wonder how well compiled simulation performs when offered an application that is a more realistic representation of the kinds of workloads that the EVP will actually have to deal with, instead of small artificial benchmarks that show the extreme cases. To this end, another two benchmark applications were simulated. The first a collection of FFT kernels, and the second a complete W-CDMA stack.

The FFT benchmark consists largely of vector operations with a fair amount of parallel

			Simulation time		**Simulation speed**		
Benchmark	**Size**	**Cycles**	*fast-sim*	*compsim*	*fast-sim*	*compsim*	**Speedup**
scalar sorting	19 kB	$8.8 \cdot 10^7$	9.572 s	0.094 s	9 MHz	931 MHz	101.8×
parallel vector addition	10 kB	$8.1 \cdot 10^7$	24.957 s	4.305 s	3 MHz	19 MHz	5.8×
float16 vectors	10 kB	$8.1 \cdot 10^7$	24.076 s	10.065 s	3 MHz	8 MHz	2.4×
FFT	244 kB	$1.2 \cdot 10^8$	22.609 s	4.878 s	5 MHz	24 MHz	4.6×
W-CDMA	589 kB	$9.8 \cdot 10^7$	10.144 s	0.085 s	10 MHz	1149 MHz	119.3×

Table 5.1: A comparison of the simulation speed of compiled simulation versus the quickest version of the interpretive simulator.

operations. Hence it is computationally intensive, but at a more realistic level than our worst-case artificial benchmark, which did little else other than adding floating point vectors. When simulating the FFT kernel, compiled simulation showed a speedup of nearly five times over the interpretive simulator. Remember that we are comparing an early prototype of compiled simulation with a heavily optimized interpretive simulator. It is not unreasonable to expect that the speedup can reach a full order of magnitude with some improvements to the compiled simulator (more on this in Chapter 6) — and that for an application that, for all intends and purposes, can be considered worst-case.

The benchmark applications that we simulated so far were all very small. The speedup of compiled simulation is most interesting when simulating much larger applications, e.g. complete protocol stacks. To this end we ran simulations of the W-CDMA stack. Surprisingly, it showed a speedup of 119 times — more than for the scalar-only benchmark. The W-CDMA benchmark does contain some scalar parts, but also plenty of parallelism and vector operations. So how can this larger speedup be explained? The answer lies mostly within the interpretive simulator, not compiled simulation. The very small scalar benchmark offers a nearly best-case scenario for the instruction decoding cache in `fast-sim`. A very limited number of instructions have to be decoded and stored in the cache, after which all decoding is reduced to cache access. This in contrast with the W-CDMA stack, which is a fairly large body of code. This hypothesis can be tested by simulating the W-CDMA stack on the original version of the interpretive simulator, which does not have an instruction decoding cache. When comparing these results with compiled simulation, we see a speedup of about 1800 times for the scalar-only benchmark and about 1400 times for the W-CDMA stack, which seems to support our hypothesis.

From this it also follows that we have not yet seen the best-case speedup of compiled simulation. Such behaviour should be elicited by a large, scalar-only application. However, the EVP was never intended for such applications, hence the best-case scenario could only arise under very artificial circumstances. All in all, a speedup of around two orders of magnitude can be expected, when simulating realistic workloads.

5.2.2 Compiled simulation versus native execution

In the previous section we judged the performance of compiled simulation by means of a comparison with another simulator. This clearly showed the benefits of compiled simulation over the current approach, but it tells us nothing about the remaining potential for improvement. We compared the speed of compiled simulation with that of native execution on the host. This should give us an idea of the overhead introduced by compiled simulation. All benchmarks with vector operations are of course excluded, since these cannot be executed natively on the host machine, leaving only the scalar sorting benchmark. Two new benchmarks were created as well, for reasons that will be explained later.

The scalar sorting benchmark, when compiled directly for the host, ran only slightly faster than when executed as compiled simulation. This spells good news for the efficiency of compiled simulation, or so it seems. With a little thought one can conclude that a sorting algorithm is constrained by memory bandwidth, since it consists of little else other than loads, stores, comparisons, and loop control. The added computational overhead introduced by compiled simulation

	Execution time		
Benchmark	*native*	*compsim*	**Slowdown**
scalar sorting	0.082 s	0.094 s	1.1 ×
binomial coefficients	0.388 s	0.658 s	1.7 ×
Lucas-Lehmer	0.068 s	0.585 s	8.6 ×

Table 5.2: Scalar benchmarks compiled directly for the host, and as compiled simulations. The differences in execution times are an indication of the overhead introduced by compiled simulation.

is hidden by the time it takes to perform all the memory operations. What we need is a computationally intensive benchmark.

To this end we implemented a Mersenne prime tester, the Lucas-Lehmer algorithm. This showed a slowdown of nearly an order of magnitude. Inspection of the code quickly led to the conclusion that the 32-bit integer division contained in the algorithm is not supported natively on the EVP, instead a software routine is invoked. This software division will thus also be part of compiled simulation, whilst the direct compilation allowed usage of the hardware divider in the host. Here we can clearly see that differences in the instruction sets of host and target machine can lead to inefficient simulation (we already discussed this in Section 2.1.3).

Finally we created a benchmark that computes a great number of binomial coefficients, using 16-bit integers. This required only instructions available on both architectures, but also contained both function calls and loops in its main workload — such constructs are a potential source of slowdown within compiled simulation, as was discussed in Section 3.3.1. With this benchmark we saw that the execution time of the compiled simulator was about 1.7 times longer than that of the native binary, i.e. compiled simulation overhead accounted for about 41% of the total execution time of the compiled simulator.

Chapter 6

Conclusions and future work

We have created a compiled simulation system for the EVP and verified the simulators it produces. Besides the implicit requirement of correctness, we had set ourselves two goals. The first goal, quick and easy adaptation to changes in the architecture, was met by the design of the simulation system. A change in the instruction set or the hardware architecture no longer requires the modification of the instruction encoding, the assembler, and the linker. Only the compiler will have to be modified, and possibly some new semantic functions have to be created. This makes it much easier to experiment with, for example, changing the size of a register file, or the addition or removal of a functional unit.

The second goal, better simulation performance, has been under investigation in Section 5.2. We observed a speedup of two orders of magnitude over a highly optimized interpretive simulator, which is what can generally be expected according to the literature. Furthermore, we evaluated the efficiency of our approach by comparing the execution times of a compiled simulator with a native execution of the original program, and saw that the overhead of compiled simulation comprised less than half of the total execution time. It seems safe to conclude that both goals were achieved.

Of course we might seek to further improve the performance of compiled simulation. Trying to improve the efficiency of the auxiliary code, i.e. all the macros, instrumentation, and register types, will meet with limited success. The performance gain for scalar programs will be less than a factor of two, from what we saw in Section 5.2.2. For vector programs the gain is suspected to be even less, since simulation overhead comprises a smaller fraction of the workload in those cases. Hence we could better seek improvement in other areas.

The implementation of the semantic functions, and in particular the wrappers, could be improved, especially in the case of vector operations. The code on which our wrappers are based, was created for verification purposes, to test semantic functions in isolation; it was not written with performance in mind, and worst of all it was untested and was a rich source of bugs. Although we have come a long way in debugging and improving the wrappers, some work still remains to be done.

The underlying semantic functions assume that all operands are distinct, i.e. if we provide the same register as destination operand and as one of the input operands, then in some cases the semantic function will not execute the semantics of its associated EVP operation correctly. This is the case for vector operations that do not produce elements of the output vector in the same order as they are consumed from the input vectors. The issue is resolved in the wrappers, by introducing temporary variables for all the operands. Since each vector is 256 bits wide, this easily adds up to a kilobyte of data transfer for each call to a semantic function, i.e. several kilobytes per simulated instruction. This might make memory bandwidth the bottleneck in simulator performance (this effect may be visible in Table 5.1; the switch from integer vectors to 16-bit float vectors caused no additional delay with the interpretive simulator). We improved the wrappers by removing all but the temporary variables for the destination vectors. Even those are superfluous in most cases, but those cases need to be identified. Once they have been identified, it is a simple matter

to remove the redundant ones, thereby increasing simulation speed.

Another inefficiency has to do with unmasked vector operations. The semantic functions and wrappers treat every operation as being masked, a default value of all ones being used in cases where no mask is supplied explicitly. In practice, a significant fraction of vector operations will be executed without a vector mask. Having a separate implementation for the unmasked version of a vector operation can be of benefit in many cases. Many vector semantics are implemented as a loop containing a conditional call to a scalar semantic function. There is no need for this conditional in an unmasked version of a vector semantic function. If the loop is unrolled as well, the semantic function would become completely branch-free, which tends to enhance performance.

A completely different approach to improving simulation speed would be to exploit the parallelism present in EVP applications. There seems to be plenty of it; not only can each instruction consist of up to 13 independent operations, but also each vector operation in itself has a similar degree of parallelism. It would seem that another two orders of magnitude of speedup are possible, by introducing multi-threading in the compiled simulators. However, exploiting the available parallelism in a multi-threaded fashion on a general purpose host would require very fine-grained synchronization. The parallelism in a VLIW machine is implicitly synchronized every clock cycle — no FU can run faster than any of the others. This synchronization has to be made explicit in a multi-threaded model. The added overhead of thread synchronization could diminish the performance gains of parallel execution, or even make the simulator run slower than the single-threaded version. In fact, some early attempts with OpenMP showed just this. Possibly we can reduce the granularity of synchronization by means of a data flow analysis; two successive instructions that are independent do not require any synchronization between them. Another way to tackle the problem could be to choose a platform where thread synchronization is cheaper. For example GPUs have a lightweight barrier sync mechanism that is ideal for our situation, i.e. a Cuda or OpenCL version of compiled simulation may be a viable option. In any case, parallelizing compiled simulation is a topic that needs much further investigation, but it is the direction in which the largest possible gains in simulation speed lie.

Appendix A

Example code

A.1 EVP-C source code

```
#include <evp_vprint.h>

extern v_t vec1;
extern int op;

v_t vec2 = evp_v_init_16(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
int n    = 100;

int main()
{
  __LoopIterMin(10)
  for (int i = 0; i < n; i++)
  {
    op += i;
    op %= 5;
  }

  switch (op)
  {
    case 0: vec1 = vec2; break;
    case 1: vec1 = evp_vadd_16(vec1, vec2); break;
    case 2: vec2 = evp_vadd_16(vec1, vec1); break;
    case 3: evp_v_printf_16(" %5d", vec1); break;
    case 4: evp_v_printf_16(" %5d", vec2); break;
    default: return -1;
  }

  return 0;
}
```

A.2 EVP assembly

```
//
// EVP Compiler r42d1: vcc1 v4.2 rev15 (gcc 4.5.0) - vd32042 - May 29 2013 14:48:08
//

.data "readonly"
.LC0:
    .byte    " %5d\000"

.text "program"

//////////////////////////
// function main
//
// Local Frame (32 bytes):
//  ptr15[  28] = regsave `ra`  (4)
//
// Parameters:
```

```
//////////////////////

.prg_main:
    .export @main
    .entry @main;
main:
// [1]
    load r1, @n                          // - IMM+SLSU  example.c:12 insn_id:116
||  vptr_update ptr15, -32               // f VLSU      example.c:10 insn_id:139
||  move r2, 0                           // - SALU      example.c:12 insn_id:9
// [2]
    load r3, @op                         // - IMM+SLSU  example.c:12 insn_id:21
// [3]
    store_ofs ptr15, 28, ra              // f IMM+SLSU  example.c:10 insn_id:140
// [4]
    do r1, .L3, .L15-1                   // - IMM+PCU   example.c:? insn_id:583
    //DELAY SLOT END
.L3:
// [1]
    move r7, 5                           // - SALU      example.c:15 insn_id:26
// [2]
    add r3, r2, r3                       // - SALU      example.c:14 insn_id:25
// [3]
    clb r4, r7                           // - SALU      example.c:15 insn_id:27
||  move_smac r6, r3                     // - SMAC      example.c:15 insn_id:134
// [4]
    asrm r5, r7, r4                      // - SALU      example.c:15 insn_id:28
// [5]
    div r6, r5, r4, 16                   // - SALU      example.c:15 insn_id:29
// [6]
    nop
// [7]
    add r2, r2, 1                        // - SALU      example.c:12 insn_id:32
// [8]
    nop
// [9]
    nop
// [10]
    nop
// [11]
    nop
// [12]
    nop
// [13]
    nop
// [14]
    nop
// [15]
    macni r3, r6, 5                      // - SMAC      example.c:15 insn_id:31
.L15:
// [1]
    move r0, -1                          // - SALU      example.c:25 insn_id:11
// [2]
    cmpgtui p1, r3, 4                    // - SALU      example.c:18 insn_id:41
||  store @op, r3                        // - IMM+SLSU  example.c:12 insn_id:39
// [3]
    br .L4, p1                           // - IMM+PCU   example.c:18 insn_id:237
// [4]
    nop
// [5]
    nop
// [6]
    nop
// [7]
    nop
// [8]
    nop
// [9]
    nop
// [10]
    nop
    //DELAY SLOT END
// [1]
    move r1r0, 0                         // - SALU      example.c:18 insn_id:136
```

```
||   move_slsu ptr0, .L10                    // - IMM+SLSU  example.c:18 insn_id:132
// [2]
     move r0, r3                             // - SALU      example.c:18 insn_id:137
// [3]
     move ptr8, r1r0                         // - SALU      example.c:18 insn_id:44
// [4]
     nop
// [5]
     move r9r8, ptr8                         // - SALU      example.c:18 insn_id:46
// [6]
     asrm r3r2, r9r8, -2                     // - SALU      example.c:18 insn_id:47
// [7]
     move ofs0, r3r2                         // - SALU      example.c:18 insn_id:48
// [8]
     nop
// [9]
     load_ofs r1r0, ptr0, ofs0               // - SLSU      example.c:18 insn_id:50
// [10]
     nop
// [11]
     nop
// [12]
     br r1r0                                 // - PCU       example.c:18 insn_id:238
// [13]
     nop
// [14]
     nop
// [15]
     nop
// [16]
     nop
// [17]
     nop
// [18]
     nop
// [19]
     nop
     //DELAY SLOT END

.data "readonly"
     .align  4
     .align 4
.STATIC10:
.L10:
      .byte4 .L5
      .byte4 .L6
      .byte4 .L7
      .byte4 .L8
      .byte4 .L9

.text "program"
.L5:
// [1]
     vload vr0, @vec2                        // - IMM+VLSU  example.c:20 insn_id:131
||   move r0, 0                              // - SALU      example.c:28 insn_id:13
// [2]
     nop
// [3]
     nop
// [4]
     vstore @vec1, vr0                       // - IMM+VLSU  example.c:20 insn_id:59
     //DELAY SLOT END
.L4:
// [1]
     load_ofs ra, ptr15, 28                  // f IMM+SLSU  example.c:29 insn_id:144
// [2]
     ptr_update ptr15, 32                    // f SLSU      example.c:29 insn_id:145
// [3]
     nop
// [4]
     nop
// [5]
     br ra                                   // - PCU       example.c:29 insn_id:239
// [6]
```

```
    nop
// [7]
    nop
// [8]
    nop
// [9]
    nop
// [10]
    nop
// [11]
    nop
// [12]
    nop
    //DELAY SLOT END
.L8:
// [1]
    callau @evp_v_fprintf_16                // - IMM+PCU    example.c:23 insn_id:574
// [2]
    load ptr0,  @_impure_data+8             // - IMM+SLSU   example.c:23 insn_id:82
// [3]
    vload vr0, @vec1                        // - IMM+VLSU   example.c:23 insn_id:84
// [4]
    move_slsu ptr8, .LC0                    // - IMM+SLSU   example.c:23 insn_id:83
// [5]
    nop
// [6]
    nop
    //DELAY SLOT END
// [7]
    move r0, 0                              // - SALU       example.c:28 insn_id:15
||  brau .L4                                // - IMM+PCU    example.c:23 insn_id:241
// [8]
    nop
// [9]
    nop
// [10]
    nop
// [11]
    nop
// [12]
    nop
    //DELAY SLOT END
.L9:
// [1]
    callau @evp_v_fprintf_16                // - IMM+PCU    example.c:24 insn_id:578
// [2]
    load ptr0,  @_impure_data+8             // - IMM+SLSU   example.c:24 insn_id:92
// [3]
    vload vr0, @vec2                        // - IMM+VLSU   example.c:24 insn_id:94
// [4]
    move_slsu ptr8, .LC0                    // - IMM+SLSU   example.c:24 insn_id:93
// [5]
    nop
// [6]
    nop
    //DELAY SLOT END
// [7]
    move r0, 0                              // - SALU       example.c:28 insn_id:14
||  brau .L4                                // - IMM+PCU    example.c:24 insn_id:243
// [8]
    nop
// [9]
    nop
// [10]
    nop
// [11]
    nop
// [12]
    nop
    //DELAY SLOT END
.L6:
// [1]
    vload vr2, @vec1                        // - IMM+VLSU   example.c:21 insn_id:130
||  move r0, 0                              // - SALU       example.c:28 insn_id:12
```

```
// [2]
    brau .L4                                    // - IMM+PCU    example.c:21 insn_id:244
// [3]
    vload vr3, @vec2                            // - IMM+VLSU   example.c:21 insn_id:65
// [4]
    nop
// [5]
    nop
// [6]
    vadd16 vr1, vr2, vr3                        // - VALU       example.c:21 insn_id:66
// [7]
    vstore @vec1, vr1                           // - IMM+VLSU   example.c:21 insn_id:67
    //DELAY SLOT END
.L7:
// [1]
    move r0, 0                                  // - SALU       example.c:28 insn_id:16
||  brau .L4                                    // - IMM+PCU    example.c:22 insn_id:245
// [2]
    vload vr5, @vec1                            // - IMM+VLSU   example.c:22 insn_id:72
// [3]
    nop
// [4]
    nop
// [5]
    vadd16 vr4, vr5, vr5                        // - VALU       example.c:22 insn_id:74
// [6]
    vstore @vec2, vr4                           // - IMM+VLSU   example.c:22 insn_id:75
    //DELAY SLOT END
.LPROCEND_main:
    .size   @main, .LPROCEND_main-@main
.section ".debug_info", "r"
.PRGINFO0:

.section ".prginfo", "r"
.PRGINFO1:
    .byte4  .PRGINFO2-.PRGINFO1 // size
    .byte2  0x1 // TAG_PRGINFO_UNIT
    .byte4  .PRGINFO0   // debug_info_offset
    .byte   0x1 // major version
    .byte   0x0 // minor version
.PRGINFO3:
    .byte4  .PRGINFO4-.PRGINFO3 // size
    .byte2  0x2 // TAG_PRGINFO_FUNC
    .byte4  __dbgindex__(@main)
    .byte4  @main   // begin label
    .byte4  .LPROCEND_main  // end label
.PRGINFO5:
    .byte4  .PRGINFO6-.PRGINFO5 // size
    .byte2  0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[0]     // start addr
    .byte4 .prg_main[4]     // end addr
    .byte4  0x3f800000  // weight (1.000)
.PRGINFO6:
.PRGINFO7:
    .byte4  .PRGINFO8-.PRGINFO7 // size
    .byte2  0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[19]    // start addr
    .byte4 .prg_main[29]    // end addr
    .byte4  0x3f800000  // weight (1.000)
.PRGINFO8:
.PRGINFO9:
    .byte4  .PRGINFO10-.PRGINFO9    // size
    .byte2  0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[29]    // start addr
    .byte4 .prg_main[48]    // end addr
    .byte4  0x3f000000  // weight (0.500)
.PRGINFO10:
.PRGINFO11:
    .byte4  .PRGINFO12-.PRGINFO11   // size
    .byte2  0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[48]    // start addr
    .byte4 .prg_main[52]    // end addr
    .byte4  0x3e2aaaab  // weight (0.167)
.PRGINFO12:
```

```
.PRGINFO13:
    .byte4   .PRGINFO14-.PRGINFO13    // size
    .byte2   0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[52]      // start addr
    .byte4 .prg_main[64]      // end addr
    .byte4   0x3f800000   // weight (1.000)
.PRGINFO14:
.PRGINFO15:
    .byte4   .PRGINFO16-.PRGINFO15    // size
    .byte2   0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[64]      // start addr
    .byte4 .prg_main[70]      // end addr
    .byte4   0x3e2aaaab   // weight (0.167)
.PRGINFO16:
.PRGINFO17:
    .byte4   .PRGINFO18-.PRGINFO17    // size
    .byte2   0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[70]      // start addr
    .byte4 .prg_main[76]      // end addr
    .byte4   0x3e2aaaab   // weight (0.167)
.PRGINFO18:
.PRGINFO19:
    .byte4   .PRGINFO20-.PRGINFO19    // size
    .byte2   0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[76]      // start addr
    .byte4 .prg_main[82]      // end addr
    .byte4   0x3e2aaaab   // weight (0.167)
.PRGINFO20:
.PRGINFO21:
    .byte4   .PRGINFO22-.PRGINFO21    // size
    .byte2   0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[82]      // start addr
    .byte4 .prg_main[88]      // end addr
    .byte4   0x3e2aaaab   // weight (0.167)
.PRGINFO22:
.PRGINFO23:
    .byte4   .PRGINFO24-.PRGINFO23    // size
    .byte2   0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[88]      // start addr
    .byte4 .prg_main[95]      // end addr
    .byte4   0x3e2aaaab   // weight (0.167)
.PRGINFO24:
.PRGINFO25:
    .byte4   .PRGINFO26-.PRGINFO25    // size
    .byte2   0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[95]      // start addr
    .byte4 .prg_main[101]     // end addr
    .byte4   0x3e2aaaab   // weight (0.167)
.PRGINFO26:
.PRGINFO27:
    .byte4   .PRGINFO28-.PRGINFO27    // size
    .byte2   0x3 // TAG_PRGINFO_LOOP
    .byte4 .prg_main[4]       // entry addr
    .byte4 .prg_main[19]      // exit addr
    .byte4   0x3f800000   // weight (dummy)
.PRGINFO29:
    .byte4   .PRGINFO30-.PRGINFO29    // size
    .byte2   0x4 // TAG_PRGINFO_BB
    .byte4 .prg_main[4]       // start addr
    .byte4 .prg_main[19]      // end addr
    .byte4   0x4121c71c   // weight (10.111)
.PRGINFO30:
.PRGINFO28:
.PRGINFO4:

.text "program"
    .export @vec2

.data "readwrite"
    .align  32
vec2:
    .size   @vec2, 32
    .byte   0
    .byte   0
```

```
    .byte   1
    .byte   0
    .byte   2
    .byte   0
    .byte   3
    .byte   0
    .byte   4
    .byte   0
    .byte   5
    .byte   0
    .byte   6
    .byte   0
    .byte   7
    .byte   0
    .byte   8
    .byte   0
    .byte   9
    .byte   0
    .byte   10
    .byte   0
    .byte   11
    .byte   0
    .byte   12
    .byte   0
    .byte   13
    .byte   0
    .byte   14
    .byte   0
    .byte   15
    .byte   0
    .export @n
    .align  2
n:
    .size   @n, 2
    .byte2  100
    .import @_impure_data
    .import @evp_v_fprintf_16
    .import @vec1
    .import @op
.section ".prginfo", "r"
.PRGINFO2:
```

A.3 Generated simulator code

A.3.1 Header file

```
namespace {

DECLARE_CONSTANT(LC0);
DECLARE_FUNCTION(main);
#define L10 JUMP_TABLE { &&L5, &&L6, &&L7, &&L8, &&L9, }
DECLARE_OBJECT(vec2,32);
DECLARE_OBJECT(n,2);
IMPORT(_impure_data);
IMPORT(evp_v_fprintf_16);
IMPORT(vec1);
IMPORT(op);

}
```

A.3.2 Source file

```
//
// EVP Compiler r42d1: vcc1 v4.2 rev15 (gcc 4.5.0) - vd32042 - May 29 2013 14:48:08
//

#include "compsim.h"
#include "example.cs.h"

namespace {


CONSTANT(LC0) { ' ', '%', '5', 'd', '\0',  };


//////////////////////////
// function main
//
// Local Frame (32 bytes):
//  ptr15[  28] = regsave `ra`  (4)
//
// Parameters:
//////////////////////////

EXPORT(main)
FUNCTION(main)
CYCLE   load_rD1_DMADDR32( r1, evp_n );                 // - IMM+SLSU  example.c:12 insn_id:116
        vptr_update_ptrA_VLSIMMN( ptr15, -32 );         // f VLSU      example.c:10 insn_id:139
        move_rD1_IMM5( r2, 0 );                         // - SALU      example.c:12 insn_id:9
CYCLE   load_rD1_DMADDR32( r3, evp_op );                // - IMM+SLSU  example.c:12 insn_id:21
CYCLE   store_ofs_ptrA_DMOFS16_gr1XC( ptr15, 28, ra );// f IMM+SLSU  example.c:10 insn_id:140
CYCLE   do_rR1_loop_start_loop_end( r1, &&L3, &&L15 );// - IMM+PCU   example.c:? insn_id:583
        DELAY_SLOT_END

LABEL(L3)

CYCLE   move_rD1_IMM5( r7, 5 );                         // - SALU      example.c:15 insn_id:26
CYCLE   add_rD1_rA1_rB1( r3, r2, r3 );                  // - SALU      example.c:14 insn_id:25
CYCLE   clb_rD1_rA1( r4, r7 );                          // - SALU      example.c:15 insn_id:27
        move_smac_rD1_rA1( r6, r3 );                    // - SMAC      example.c:15 insn_id:134
CYCLE   asrm_rD1_rA1_rB1( r5, r7, r4 );                 // - SALU      example.c:15 insn_id:28
CYCLE   div_rD1_rA1_rB1_DIVPRECMAX16( r6, r5, r4, 16 );// - SALU      example.c:15 insn_id:29
CYCLE   // nop
CYCLE   add_rD1_rA1_IMM5( r2, r2, 1 );                  // - SALU      example.c:12 insn_id:32
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   macni_rD1_rA1_IMM4( r3, r6, 5 );                // - SMAC      example.c:15 insn_id:31

LABEL(L15)

CYCLE   move_rD1_IMM5( r0, -1 );                        // - SALU      example.c:25 insn_id:11
CYCLE   cmpgtui_pD_rA1_UIMM5( p1, r3, 4 );              // - SALU      example.c:18 insn_id:41
        store_DMADDR32_rB1( evp_op, r3 );               // - IMM+SLSU  example.c:12 insn_id:39
CYCLE   br_PMADDR_delay3_7_5_1( &&L4, p1 );             // - IMM+PCU   example.c:18 insn_id:237
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
        DELAY_SLOT_END
CYCLE   move_rD2_IMM5( r1r0, 0 );                       // - SALU      example.c:18 insn_id:136
        move_slsu_ptrD_DMADDR32( ptr0, L10 );           // - IMM+SLSU  example.c:18 insn_id:132
CYCLE   move_rD1_rA1( r0, r3 );                         // - SALU      example.c:18 insn_id:137
CYCLE   move_gr2D_rA2( ptr8, r1r0 );                    // - SALU      example.c:18 insn_id:44
CYCLE   // nop
CYCLE   move_rD2_gr2A( r9r8, ptr8 );                    // - SALU      example.c:18 insn_id:46
```

```
CYCLE   asrm_rD2_rA2_IMM5( r3r2, r9r8, -2 );          // - SALU       example.c:18 insn_id:47
CYCLE   move_gr2D_rA2( ofs0, r3r2 );                  // - SALU       example.c:18 insn_id:48
CYCLE   // nop
CYCLE   load_ofs_rD2_ptrA_ofsB( r1r0, ptr0, ofs0 );// - SLSU       example.c:18 insn_id:50
CYCLE   // nop
CYCLE   // nop
CYCLE   br_rR2_delay3_7_5_1( r1r0 );                  // - PCU        example.c:18 insn_id:238
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
        DELAY_SLOT_END

        // jump table L10

LABEL(L5)

CYCLE   vload_vrD1_DMADDR32( vr0, evp_vec2 );         // - IMM+VLSU   example.c:20 insn_id:131
        move_rD1_IMM5( r0, 0 );                       // - SALU       example.c:28 insn_id:13
CYCLE   // nop
CYCLE   // nop
CYCLE   vstore_DMADDR32_vrB1( evp_vec1, vr0 );        // - IMM+VLSU   example.c:20 insn_id:59
        DELAY_SLOT_END

LABEL(L4)

CYCLE   load_ofs_gr1XD_ptrA_DMOFS16( ra, ptr15, 28 );// f IMM+SLSU   example.c:29 insn_id:144
CYCLE   ptr_update_ptrA_LSIMMP( ptr15, 32 );          // f SLSU       example.c:29 insn_id:145
CYCLE   // nop
CYCLE   // nop
CYCLE   br_ra_delay3_7_5_1( ra );                     // - PCU        example.c:29 insn_id:239
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
        DELAY_SLOT_END

LABEL(L8)

CYCLE   callau_PMADDR_delay7_5_1_1( evp_evp_v_fprintf_16 );// - IMM+PCU    example.c:23 insn_id:574
CYCLE   load_gr1D_DMADDR32( ptr0, evp__impure_data+8 );// - IMM+SLSU   example.c:23 insn_id:82
CYCLE   vload_vrD1_DMADDR32( vr0, evp_vec1 );         // - IMM+VLSU   example.c:23 insn_id:84
CYCLE   move_slsu_ptrD_DMADDR32( ptr8, evp_LC0 );     // - IMM+SLSU   example.c:23 insn_id:83
CYCLE   // nop
CYCLE   // nop
        DELAY_SLOT_END
CYCLE   move_rD1_IMM5( r0, 0 );                       // - SALU       example.c:28 insn_id:15
        brau_PMADDR_delay3_5_5_1( &&L4 );             // - IMM+PCU    example.c:23 insn_id:241
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
CYCLE   // nop
        DELAY_SLOT_END

LABEL(L9)

CYCLE   callau_PMADDR_delay7_5_1_1( evp_evp_v_fprintf_16 );// - IMM+PCU    example.c:24 insn_id:578
CYCLE   load_gr1D_DMADDR32( ptr0, evp__impure_data+8 );// - IMM+SLSU   example.c:24 insn_id:92
CYCLE   vload_vrD1_DMADDR32( vr0, evp_vec2 );         // - IMM+VLSU   example.c:24 insn_id:94
CYCLE   move_slsu_ptrD_DMADDR32( ptr8, evp_LC0 );     // - IMM+SLSU   example.c:24 insn_id:93
CYCLE   // nop
CYCLE   // nop
        DELAY_SLOT_END
CYCLE   move_rD1_IMM5( r0, 0 );                       // - SALU       example.c:28 insn_id:14
        brau_PMADDR_delay3_5_5_1( &&L4 );             // - IMM+PCU    example.c:24 insn_id:243
CYCLE   // nop
CYCLE   // nop
```

```
CYCLE    // nop
CYCLE    // nop
CYCLE    // nop
         DELAY_SLOT_END

LABEL(L6)

CYCLE    vload_vrD1_DMADDR32( vr2, evp_vec1 );       // - IMM+VLSU   example.c:21 insn_id:130
         move_rD1_IMM5( r0, 0 );                     // - SALU       example.c:28 insn_id:12
CYCLE    brau_PMADDR_delay3_5_5_1( &&L4 );           // - IMM+PCU    example.c:21 insn_id:244
CYCLE    vload_vrD1_DMADDR32( vr3, evp_vec2 );       // - IMM+VLSU   example.c:21 insn_id:65
CYCLE    // nop
CYCLE    // nop
CYCLE    vadd16_vrD1_vrA1_vrB1( vr1, vr2, vr3 );     // - VALU       example.c:21 insn_id:66
CYCLE    vstore_DMADDR32_vrB1( evp_vec1, vr1 );      // - IMM+VLSU   example.c:21 insn_id:67
         DELAY_SLOT_END

LABEL(L7)

CYCLE    move_rD1_IMM5( r0, 0 );                     // - SALU       example.c:28 insn_id:16
         brau_PMADDR_delay3_5_5_1( &&L4 );           // - IMM+PCU    example.c:22 insn_id:245
CYCLE    vload_vrD1_DMADDR32( vr5, evp_vec1 );       // - IMM+VLSU   example.c:22 insn_id:72
CYCLE    // nop
CYCLE    // nop
CYCLE    vadd16_vrD1_vrA1_vrB1( vr4, vr5, vr5 );     // - VALU       example.c:22 insn_id:74
CYCLE    vstore_DMADDR32_vrB1( evp_vec2, vr4 );      // - IMM+VLSU   example.c:22 insn_id:75
         DELAY_SLOT_END
END_FUNCTION(main)

EXPORT(vec2)

INITIALIZE(vec2,32) { BYTE(0), BYTE(0), BYTE(1), BYTE(0), BYTE(2), BYTE(0), BYTE(3), BYTE(0),
                      BYTE(4), BYTE(0), BYTE(5), BYTE(0), BYTE(6), BYTE(0), BYTE(7), BYTE(0),
                      BYTE(8), BYTE(0), BYTE(9), BYTE(0), BYTE(10), BYTE(0), BYTE(11), BYTE(0),
                      BYTE(12), BYTE(0), BYTE(13), BYTE(0), BYTE(14), BYTE(0), BYTE(15), BYTE(0),
};

EXPORT(n)
INITIALIZE(n,2) { BYTE2(100),  };

}
```

Bibliography

[1] Joseph A. Fisher, Paolo Faraboschi, Cliff Young, *Embedded computing: A VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[2] Christopher Mills, Stanley C. Ahalt, Jim Fowler, *Compiled Instruction Set Simulation*. Software — Practice and Experience, vol. 21 (8), pp. 877–889, 1991.

[3] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, Andreas Hoffmann, *A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation*. Proceedings of the 39th annual Design Automation Conference, pp. 22–27, ACM, 2002.

[4] Moo-Kyoung Chung, Chong-Min Kyung, *Improvement of Compiled Instruction Set Simulator by Increasing Flexibility and Reducing Compile Time*. Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping, 2004.

[5] Zivojnovi, Vojin, Steven Tjiang, Heinrich Meyr, *Compiled simulation of programmable DSP architectures*. Journal of VLSI signal processing systems for signal, image and video technology 16.1, pp. 73–80, 1997.

[6] Zhonglei Wang, Jörg Henkel, *HyCoS: Hybrid Compiled Simulation of Embedded Software with Target Dependent Code*. Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 133–142, 2012.

[7] Edsger W. Dijkstra, *Go To Statement Considered Harmful*. Communications of the ACM, Vol. 11, No. 3, pp. 147–148, 1968.

[8] *Instruction Set Manual — Instruction set for VD32042*. ST-Ericsson, 2011.

[9] Richard M. Stallman, *GNU compiler collection internals: for GCC version 4.5.0*. Free Software Foundation (2010).

[10] Diego Novillo, *From source to binary: The inner workings of GCC*. Red Hat Magazine (`www.redhat.com/magazine/002dec04/features/gcc`), 2004.

[11] Jan Hubička, *The GCC call graph module: a framework for inter-procedural optimization*. Proceedings of the 2004 GCC Developers' Summit, pp. 65–78, 2004.

[12] Alex Turjan, Dmitry Cheresiz, Claudiu Zissulescu, Wim Kloosterhuis, *VGCC: the GCC port for the EVP architecture*. ST-Ericsson, 2012.