Designing hardware to interpret virtual machine instructions : concept and partial implementation for java byte code

Steinbusch, Otto

Award date:
1998

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
SECTION OF INFORMATION- AND COMMUNICATION SYSTEMS

PHILIPS

Designing Hardware to Interpret Virtual Machine Instructions

Concept and partial implementation for Java Byte Code

By Otto Steinbusch

Author	Otto Steinbusch	
Coach	Milton Ribeiro, MSEE	
Company	Handheld Computing Group, Philips Semiconductors	
Supervisor	Prof.Dr.-Ing. J.A.G. Jess	
Examiner	Dr.Ir. A.C. Verschueren	
Date	February 1998	ID 363006

Abstract

This Master's Degree thesis describes the project the author completed between June 1997 and February 1998 at Philips Semiconductors in Sunnyvale, California, USA. This project concludes the author's study of Information Technology at the Department of Electrical Engineering at the Eindhoven University of Technology. The thesis details the design and development of a modular hardware component called the Virtual Machine Interpreter (VMI). By using the VMI, programs written in Java Byte Code execute faster on a RISC platform. In this case the MIPS platform is targeted but any RISC platform could be targeted. The concept is applicable for any stack-oriented machine language but only implemented for Java Byte Code (JBC).

A functional model of the VMI was designed at the Philips Research Laboratories Eindhoven (PRLE) and used in simulations to prove the feasibility and estimate the performance of the VMI combined with a MIPS processor. These preliminary results are analyzed, verified and amended. Additional research brings new results. In cooperation with PRLE, a model of the VMI core was designed on register transfer level (RTL), using VHDL.

Conclusions

- At the evaluated level of system integration, the VMI speeds up the execution of JBC by 4.0 times, compared to a software interpreter.
- The VMI speeds up a typical large application for the embedded market by 2.6 times, because not all execution time is spent interpreting pure JBC.
- The structural VHDL model of the VMI core is an RTL description of the functional C model of the VMI core. It correctly handles the same bytecodes the functional model can handle. This is an important step towards a hardware implementation.
- The structural model of the VMI core will, with a few improvements, generate instructions fast enough not to endanger the performance gain as estimated in chapter three.

ACKNOWLEDGEMENTS

Working on this project has been an incredible experience for me. Silicon Valley has made a lasting impression on me. First and foremost, I'd like to thank my family for all their support. Thanks to Milton Ribeiro, Augusto de Oliveira and Cees Jan Koomen for making this project possible and helping me. Thanks to Jochen Jess and Ad Verschueren for guidance. Lots of thanks to Menno Lindwer for all his help and patience. Thanks to Hans Spanjaart and Padraig O'Mahony for their help with this report.

Otto Steinbusch

April 1998

TABLE OF CONTENTS:

1. INTRODUCTION

After a general introduction to the subject matter the assignment is detailed. To conclude this chapter a brief outline of the rest of the report follows.

1.1 SCOPE OF THE PROJECT

This project was carried out for three reasons. The first reason is the rise of Java, a programming language and a format for software distribution. Java technology has applications in a broad range of markets. Applications can range from interactive websites on the Internet to full-blown office suites for the desktop market. Other applications could be smart cards or smart telephones and TV set-top boxes. Basically any system, embedded or not, that could improve by adding one of Java's capabilities, such as dynamic and interactive update of software, forms a suitable target for the expanding Java industry. New opportunities are being discovered at a fast rate and time will only tell if Java becomes as truly ubiquitous as some people predict.

The second reason is caused by the needs of the Handheld Computing Group (HCG), based in Sunnyvale, California. HCG is a part of Philips Semiconductors. HCG produces chips and chipsets, which are specifically tailored to the needs in the embedded market with regards to performance, power consumption and cost. The people at HCG acknowledged two facts:

1. It is likely that more people will use Java in the future, both as producers and consumers of software.
2. Embedded systems cannot execute Java programs as fast as HCG wants, because embedded systems lack certain resources, especially memory.

For some of their products HCG wants a way to improve the speed of execution of Java programs.

The third reason is the research done by the Information Processing Architectures group, which is a part of the sector Information and Software Technology of the Philips Research Laboratories Eindhoven (PRLE, also known as Natlab). This group conducts experimental research and design in various areas. One particular project resulted in a functional design for a modular hardware component that can speed up the execution of Java programs. Prior to this project, an experiment was conducted at PRLE to verify whether this approach was viable. If additional research would sufficiently support the stated claims, it would be worthwhile to implement the design. For the remainder of this report, all references to 'PRLE' indicate the people that worked on the particular project above mentioned, unless otherwise stated.

For these three reasons, the author carried out the project described in the following subsection. The author remained focussed on HCGs wishes as listed below and he closely co-operated with PRLE during the project.

1.2 PROBLEM DESCRIPTION

HCG is interested to what extent the group can benefit from such a hardware component. Analysis showed this component to be sufficiently beneficial, so HCG wanted to have a structural model designed in a major hardware description language (HDL). This allows better simulations. Building a structural model is a natural step towards synthesis for FPGA or silicon implementations. HCG would also like to have an FPGA implementation.

1.3 COMPARING EXECUTION MECHANISMS

The performance of the hardware component is compared to the performance of a regular software interpreter, because this execution mechanism is available for most embedded systems. A regular software interpreter is a software program that interprets bytecodes one by one without caching the entire method. There are faster execution mechanisms available today.

- A just-in-time (JIT) compiler needs about 500 KB for program code and 2 MB for caching results to obtain a speed-up over a regular software interpreter of a factor of 4x to 5x. A JIT compiler compiles a method to native code prior to its execution. This data originates from a group at PRLE that researches compiler technology. JIT compilation requires more resources, especially memory, than many embedded systems can spare. An additional drawback with using a JIT compiler is that its unpredictability causes real time complications, which can be an issue in some embedded applications.
- Java-specific hardware, such as Sun's PicoJava, can execute JBC very fast, but it cannot execute legacy code. This drawback is not acceptable for Philips.
- A specialized processor that executes both MIPS code and JBC **natively** is an interesting option. At this time, it seems that such a solution yields considerable performance gain but implies a prohibitive amount of design effort as well. Research in this area has been done at PRLE as well.

Because of these drawbacks HCG is looking for an alternative execution mechanism. The hardware component described in this report offers an execution mechanism that does not have these drawbacks. At the start of this project, the estimated performance gain was not clear.

1.4 OUTLINE OF THE REPORT

Chapter 2 provides an elementary introduction to Java. The following concepts are treated: Java, Java Byte Code, the Java Virtual Machine and the task of a Java interpreter. Chapter 3 introduces the basic concepts of the Virtual Machine Interpreter (VMI) and contains a detailed analysis of the estimated performance gain. This concludes the first part of the assignment. The second part of the assignment, implementing the model, starts at chapter 4. Chapter 4 shows how the structural model of the VMI evolved. Chapter 5 treats the simulation of various models of the VMI.

A reference to another document is placed in square brackets, like this: [reference name].

2. JAVA

This chapter provides a brief overview of the basics of Java technology. In fact, only those aspects needed to understand this project will be treated. Some of the features of Java as a programming language will be mentioned but there is no need to be complete or to treat this subject in depth. To grasp the essentials of this project, some understanding of Java Byte Code is necessary, as are some of the internals of the Java Virtual Machine. If the reader is familiar with these concepts, he may skip this chapter.

2.1 INTRODUCTION

Since its introduction in 1992 by Sun Microsystems Java has had a tremendous impact. One of the confusing issues to people who are not proficient in the world of Java is the multitude of technologies that go by the same or a similar name. The most important ones will be treated in this section. In short, there are three key elements.

- **Java** is a general-purpose object-oriented programming language with C-like syntax. The object-oriented nomenclature is used, so 'procedures' are called '**methods**' and each data type and the methods that operate on it are grouped together in units called '**classes**'. An instance of a class is called an '**object**'.

  The Java programming language [JLS] has a large set of features. Since the language and the virtual platform on which programs are executed were developed together, they are tightly coupled. This means that some of the special features of the language entail special virtual machine instructions. Knowledge of the programming language is not essential to this project. For more information on object-oriented programming see [OOP].

- **Java Byte Code** (JBC) Usually, a Java program is not compiled directly to machine instructions for the user's platform. Instead, a program is compiled to an intermediate format called Java Byte Code. JBC is a stack-oriented machine language. JBC cannot be executed on a MIPS platform or any other RISC platform without a JBC Interpreter (see next bullet). 'Stack-oriented' means computations are done on an operand stack (see next bullet). A *stack* is a Last In First Out (LIFO) queue.

  Each compiled class resides in a separate file. At this point, these files can be transported across a file system or a network. These files must adhere to a very strict format in order to ensure platform-independence and enforce safety-restrictions. A program needs an implementation of the Java Virtual Machine (see next bullet) to execute. The different types of bytecodes and their characteristics play an important role in any effort to speed up the execution of Java programs. Java Virtual Machine instructions consist of one-byte opcodes followed by zero or more operand bytes. Hence the name **byte**code as opposed to binary code.

  The top of the operand stack implicitly acts as both a source and a destination address. This is fundamentally different from a RISC architecture like MIPS. A list of the different categories of bytecodes can be found in section 3.4 'Estimated performance gain' under subsection 'Dynamic bytecode distribution'.

- The **Java Virtual Machine** (JVM) is an abstract computing device with a stack architecture. Some understanding of the JVM is needed to understand

this project. The following items are important facts to know about the JVM and its internal memory structure, because the hardware component that is described in the following chapter needs to access different parts of the JVM and its internal memory structure.

1. The JVM contains a **JBC interpreter**, whether this is a software program or a specific hardware component. A JBC interpreter performs the basic *interpreter loop*: fetching, decoding and executing bytecodes, one by one. *Executing* JBC means translating it into machine code for the machine that is running the Java program and having the CPU execute those instructions. JBC only contains instructions from JVMs instruction set.
2. The memory space a Java program can use is called a *heap*.
3. A part of a program that can execute independently from other parts is called a *thread*. Execution can be divided among several threads. Threads are allocated on the heap.
4. The *method area,* which is allocated on the heap, contains per class structures such as a method's code. Another part of the *method area* is the *constant pool,* which contains information on *all* the classes of a program. The method area is shared among threads.
5. Objects are allocated on the heap and never explicitly deallocated. A background thread automatically deallocates memory that is no longer used. This is called garbage collection.
6. The JVM is stack-based because each thread has a *Java stack*, allocated on the *heap*. The Java stack holds a frame for each method. The frame for the current method is located at the top of the Java stack. Suppose the current method is called A. Suppose method A calls method B. A new frame for method B is pushed on top of the Java stack and this new frame becomes the current frame. Once execution of method B is finished, the results are placed in method A's frame. The frame for method B is discarded and the frame for method A is once again the current frame.
7. A *method frame* consists of three sections: a **local variable** section, an execution environment section (also known as frame state, which among others holds information to restore the state and a pointer to the constant pool) and an **operand stack** section. This stack is completely different from the *Java stack* that was mentioned earlier. In the operand stack section the operands for a bytecode are located. After the bytecode is executed, the result is also placed on the operand stack.

   Figure 1 depicts the internal JVM memory structure. The arrows indicate in which direction the stack grows. For more information the reader is referred to [JLS] and [JVM].

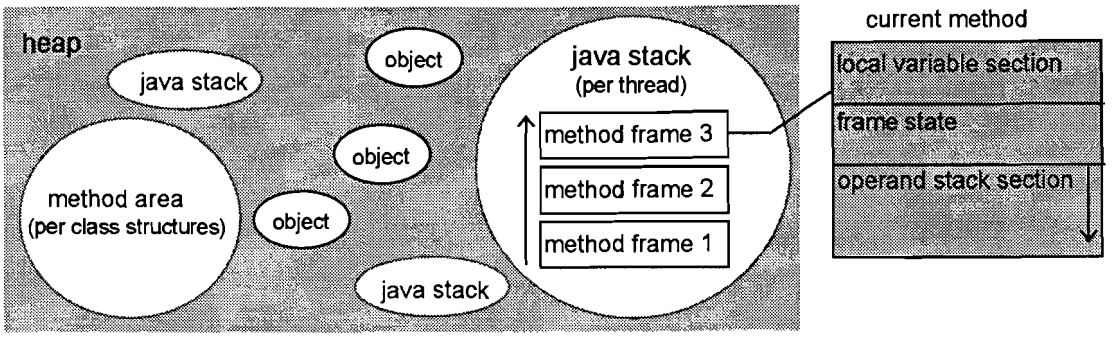

Figure 1: Internal JVM memory structure

3. THE VIRTUAL MACHINE INTERPRETER

This chapter provides both an introduction to the basic concepts of the VMI and a closer look at the acceleration it brings, compared to a regular software interpreter. The first part is essentially a compressed version of [MML]. Understanding the concepts is fundamental to this report. The latter part involves an elaborate study of the behaviour and characteristics of benchmark programs. The author has put together a benchmark suite for this purpose.

3.1 INTRODUCTION

PRLE set out to design a hardware module that can assist a CPU in executing Java programs. To be more precise, the module helps the CPU to perform the *interpreter loop*. After fetching and decoding a bytecode, the module translates it into MIPS instructions and then these instructions are fed to the CPU for execution. The part of the module that takes a bytecode and translates it is called the *VMI core*. The other parts that are needed to provide a way to interact with the CPU are jointly called the *support module*. The entire module, *VMI core* and *support module*, should work together with any general purpose CPU but not hamper the CPUs effectiveness for executing its own native code. This hardware module is called Virtual Machine Interpreter (VMI). For the remainder of this report, 'CPU' should be read as 'MIPS CPU', unless otherwise stated.

There are four system architecture options for the VMI. For these options, the VMI core remains the same but the support module is different. The key factor is the position of the VMI:

1. The CPU is connected to a general interconnect bus and the entire VMI is a peripheral on this bus (see following figure).
2. The VMI core is placed between the CPU and the memory from which the CPU fetches instructions. If the CPU communicates with the memory using a general bus, as is assumed here, the VMI can be placed in-between the CPU and the bus.
3. Similar to option 2, the VMI can be placed between the bus and the memory.
4. The VMI core is embedded in the CPU, between the instruction cache and the CPU core.

For reasons stated in [MML], the first option was chosen as depicted in the following figure.

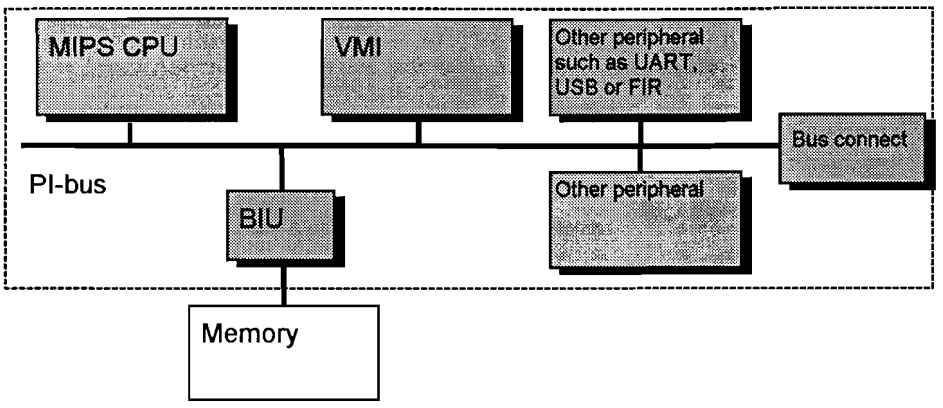

Figure 2: Chosen system architecture option

The PI-bus is a standard interconnect bus for this type of system. The BIU is the bus interface unit, which handles traffic to and from the memory. The dotted box indicates the chip boundary. The memory is off-chip.

3.2 Basic Concepts

The following sections will highlight the basic concepts of the VMI.

Generating instructions using tables

The VMI looks up each bytecode in a table and replaces it with its translation, which consists of MIPS instructions. The VMI uses more tables with bytecode dependent information, e.g. the number of arguments a bytecode uses, to translate a bytecode. A more detailed explanation of the tables VMI uses can be found in section 4.2. 'Unpipelined model of the VMI core'.

The register stack

JBC is a stack-oriented machine language and RISC machine language is register-oriented (i.e. uses a load-store architecture and general purpose registers). The operands for JBC instructions reside on the operand stack and the results are placed on the operand stack. Implementations of a stack-oriented machine language on a processor with a RISC architecture must perform extra instructions to move data to and from the stack. To reduce accesses to memory, the top positions of the operand stack are mapped onto CPU registers. Hence, popping arguments from the stack and pushing results on the stack only involves operations on registers. The resulting MIPS instructions depend on the state of the stack, e.g. the exact position of the top of the stack (TOS). A VMI register, the *Register Stack Pointer* (RSP), holds the address of the current top of the stack.

The translation table contains MIPS instructions for which the register fields and immediate fields are encoded. These instructions are called *instruction skeletons* because the register fields and the immediate fields must be decoded.

For example, often a register field must be the current value of the TOS. Since there are 5 bits in a register field, there are 32 different values that an encoded

register field can have. Encoding parts of an instruction in smaller steps is often called *micro coding*. Using micro code for generating stack-state dependent MIPS instructions is an elegant way to make this mapping efficient. The reader can find a more detailed explanation of *instruction skeletons* and the encoding of register fields and immediate fields in section 4.7 'Internals of logic block IG explained'.

The register stack has only so many positions. Therefore it is necessary to either flush stack values to memory, if the stack is too full to push the bytecode's resulting values, or restore stack values from memory, if not all the bytecode's operands are available in the register stack.

Using micro code to direct instruction generation

Bytecodes can be parameterized by one or more operand bytes. Rather than providing VMI with different instruction sequences for all possible parameters, these operands should influence the resulting MIPS instruction sequence. Two types of micro code accompany the instruction skeletons in the Translation Table.

The first type of micro code controls how MIPS instructions are generated by combining instruction skeletons with

- bytecode parameters,
- immediate values,
- stack-state-dependent values such as the registers that act like source, target or destination in a MIPS instruction.

This micro code is placed in the register fields and immediate fields of the instruction skeletons.

The second type of micro code consists of control signals that direct the process of translating bytecodes. For example, at the end of a sequence of MIPS instructions, one particular bit in the additional micro code, called *last_instr*, will be set to signal that the translation for this bytecode is finished.

These control signals form a bitvector that accompanies every instruction skeleton. Every entry in the Translation Table consists of one instruction skeleton with encoded fields and one bitvector of micro code for controlling the VMI itself, as depicted in the following figure. TT is actually a list of sequences of TT entries, one sequence for every bytecode. Some bytecodes have a TT sequence with only one entry (i.e. the bytecode translates into only one MIPS instruction) while other bytecodes can have as much as 16 entries in their TT sequence.

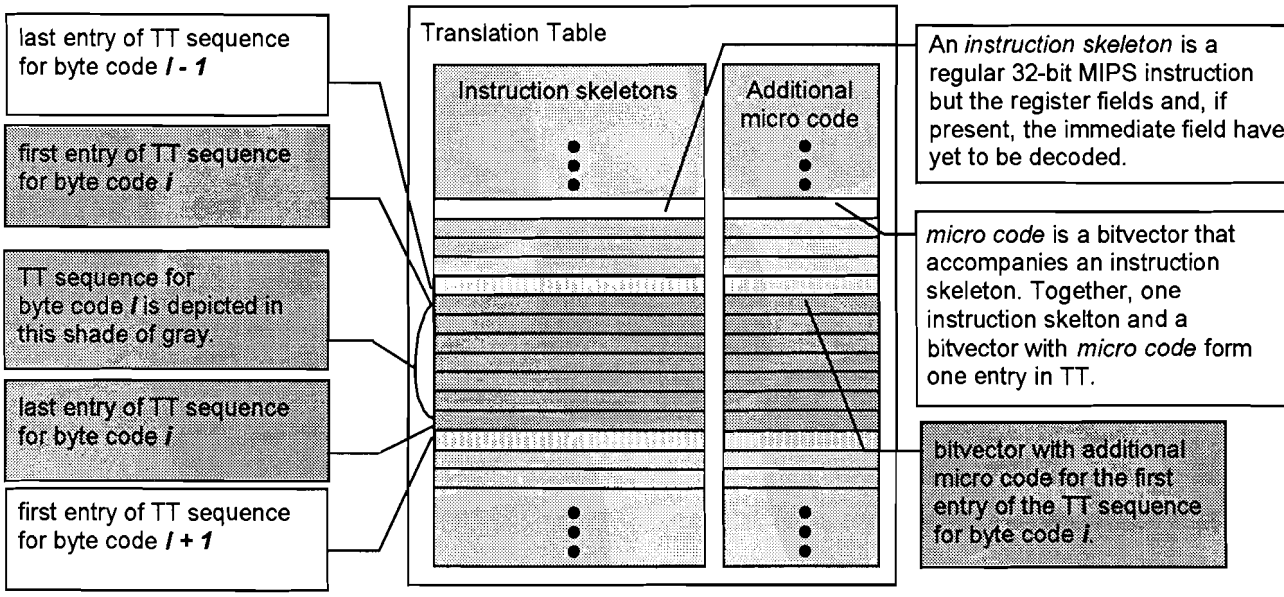

Figure 3: An entry in TT consists of one instruction skeleton and one bitvector with additional micro cod

The VMI range principle

A part of the physical address map is assigned to the VMI. This part is called the *VMI range*. Whenever the CPU tries to read an instruction from within this range, the VMI responds and supplies an instruction. In order to start JBC execution, the startup code does two important things:

1. It makes sure the VMI knows at which address the JBC starts. The VMI fetches bytecodes, starting from this address, and translates them into sequences of CPU instructions.
2. It makes the CPU jump to the first address in the VMI range. This will cause the VMI to detect a read operation in its range. The VMI will place a generated instruction on the bus. While the CPU executes this instruction, VMI generates the next CPU instruction.

When the VMI detects a fetch near the last address in its range, a jump to the beginning of the VMI range is inserted.

The following figure depicts the VMI range as a part of the memory.

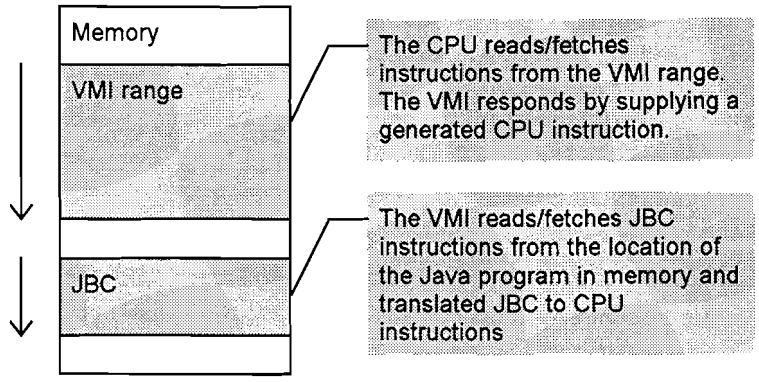

Figure 4: The VMI range memory is a part of the memory

3.3 FUNCTIONAL MODEL OF THE VMI

For the remainder of this report, whenever a model of the VMI is discussed, it is assumed to be the bus peripheral version as chosen in this chapter.

Essentially, the VMI core translates sequences of bytecodes into CPU instructions by nesting two 'switch' statements. A 'switch' statement is a multi-way decision that tests whether an expression matches one of a number of constant values and branches accordingly. The outer switch selects a bytecode. The inner switches are directed by a state variable. This state machine keeps track of the progress of the translation of a bytecode. On a higher level, the *control state machine* directs the progress of the entire VMI.

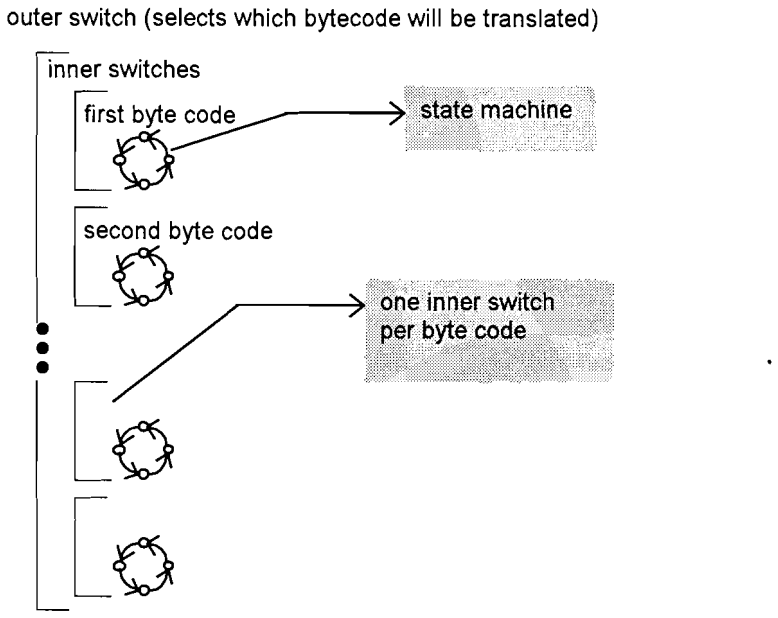

Figure 5: VMI uses nested switches

The functional model of the VMI is a monolithic unit, programmed in C. There are several parts that make the VMI core interact with the rest of the system, i.e. the CPU and the main memory. These parts are jointly called the *support module* and include the following:

- The PI-bus interface, which has a master/slave functionality.
- The *bytecode cache,* which supplies the VMI core with bytecodes and operands. It interacts with the main memory via the PI-bus as a bus master.
- The *register file*, which holds all the registers of the VMI.
- The *instruction FIFO*, which holds the CPU instructions that were generated by the VMI core. The instruction FIFO is a part of the register file. The instructions are sent to the CPU by the *instruction issue module*, which acts as a bus slave. The way the VMI uses the instruction FIFO is patented.
- The *register access module*, which allows the CPU to write results to one of VMIs registers. In this case the VMI acts as a bus slave. The VMI core can generate MIPS instructions that cause the CPU to write directly to one of the registers of the VMI. This feature is used for example in case of a conditional branch. The result of a comparison between CPU registers is written to a VMI register, so the VMI can determine whether it should take the branch or not. This feature is also used to restore the state of the VMI after a context switch.

The following figure shows the functional model of the VMI.

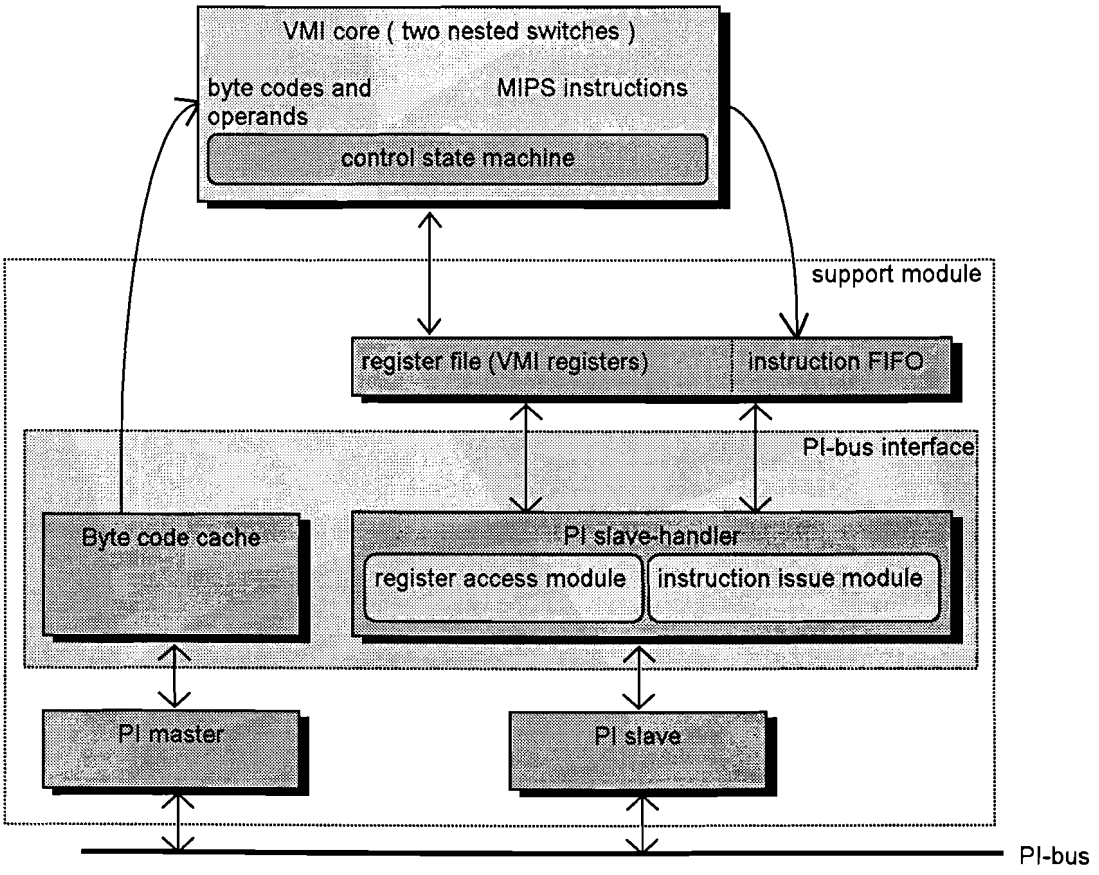

Figure 6: Functional model of the VMI

All the elements of the support module are connected to the control state machine, although this is not drawn.

3.4 ESTIMATED PERFORMANCE GAIN

The performance of the VMI as a bus peripheral is compared to the performance of a regular software interpreter. The subject of this section is the version of the VMI that is introduced in [MML], since initially this version is going to be implemented.

The estimated performance gain over a software interpreter depends on a number of variables. Different programs will be accelerated at different rates. As mentioned before, HCGs interest is the performance of typical Java programs on embedded systems. This means an entire office suite is much too big. On the other hand, cute little dancing Java applets, nowadays used to make web-sites look more attractive, are not expected to be the typical Java programs in years to come.

3.4.1 Sun's dynamic distribution of bytecodes

The research on the estimated performance gain starts with the results from [MML]. The approximation of the performance gain in that report uses the dynamic distribution of bytecodes as given by Sun Microsystems in [PicoJava]. The *dynamic distribution* states how many times each bytecode is executed when the program runs, usually as a percentage. *Static distribution* counts the bytecodes when the program is stored, also usually as a percentage. Sun obtained the dynamic distribution by measuring two programs:

- *javac*, the Java compiler has 422 KB of code, which is a lot for most embedded systems. More importantly, *javac* has no resemblance to any interesting functionality for a user on an embedded system. This program hardly interacts with the user or the rest of the system.
- *RayTracer* generates an image of a 1400-triangle dinosaur standing on a glossy table. The code size is 36 KB but still the program is considered not to be a good benchmark, mainly due to lack of user interactivity.

This section continues to use the numbers from Sun. For subsequent sections other programs are used to determine the dynamic distribution because the two programs that Sun uses do not reflect a typical functionality. The following table lists Sun's numbers for dynamic bytecode distribution.

Table 1: Dynamic distribution, as percentages, of bytecodes according to Sun

load	field load	store	compute	branch	call / return	Push constant	misc. stack	new object	others
34.5	20.0	11.0	9.2	7.9	7.3	6.8	2.1	0.4	0.8

The following table uses this distribution. It is taken from [MML] and shows the performance gain of the VMI over a general software interpreter. For each category the additional data in the first column details which bytecodes are a part of this category, unless it is obvious. These details will be used to estimate the performance gain more accurately. The following categories are examples that will be used in the following tables.

- For example, all bytecodes **load** are a part of the category **Load** where **load** includes bytecodes *iload*, *lload*, *fload*, *dload*, *aload* but also *iload_0*, *iload_1*, *iload_2*, *iload_3* and so on.
- *ldc** includes bytecodes like *ldc* and *ldc2_w* that push items from the constant pool onto the stack.
- *get** includes bytecodes *getstatic* and *getfield* and is a part of **Field load**.
- *i&l* indicates computational bytecodes that operate on integers and longs.
- *f&d* indicates computational bytecodes that operate on floats and doubles.

Table 2: Theoretical model from [MML] with detailed bytecode categories.

Byte code	Dynamic freq.	VMI cycles (VC)	time_VMI	SW cycles	time_SW
Load **load** *ldc**	34.5	2	69.0	53	1829
Field load **aload get**	20.0	21	420.0	66	1320
Store **astore *store* put**	11.0	2	22.0	53	583
Compute *i&l f&d convert iinc*	9.2	4	36.8	61	561
Branch	7.9	20	158.0	65	514
Call / return *invoke* *return*	7.3	50	365.0	95	694
Push const **const**	6.8	2	13.6	53	360
Misc.stack	2.1	4	8.4	61	128
New Object	0.4	75	30.0	120	48
Others	0.8	50	40.0	95	76
Total	100.0		1162.8		6113

Sun's dynamic bytecode distribution is listed in column two. The column *VMI cycles* estimates how many cycles it would take to execute a particular type of bytecode with the use of the VMI. The column *time_VMI* lists how much time it would take to execute all the application's bytecodes of one type. It is the result of multiplying the dynamic frequency with the number of VMI cycles this type of bytecode takes. The column *SW cycles* lists the amount of cycles a software interpreter is expected to need to execute a particular type of bytecode. This number is a function of the number of VMI cycles, based on empirical research at PRLE on software interpreters. The last column lists the amount of time a software interpreter is expected to take for one type of bytecode. The theoretical speed-up is obtained by dividing the total time it would take an average application to complete on a software interpreter with the same number for the VMI. The result is a speed-up of (6113 / 1162.8) is **5.3** times.

3.4.2 The benchmark suite

As stated before, Sun's numbers for the dynamic distribution of bytecodes were obtained by profiling non-typical programs. In this section the six programs that are selected to form the benchmark suite are profiled.

After careful consideration the following six applets were selected to use for extracting profiling data. These programs rank high according to the world's leading independent Java applet repositories such as JARS.com, Gamelan and JavaWorld. These benchmark programs form the benchmark suite that will be used throughout the report. Three applets are of considerable size and functionality. The other three applets are smaller and less useful but still good examples of proper object-oriented-coding. The smaller applets are capable of the type of behaviour often found to enhance the appearance of web-sites. The applets differ in the extent to which they use a graphical user interface, interactivity, processing power for various means and typical object-oriented-coding techniques.

- *DigSim.* This applet allows the user to build simple electronic circuits and analyze and simulate them. The code size is 180 KB.
- *Jess.* The Java Expert System Shell is a clone, entirely written in Java, of the CLIPS Expert System Shell by NASA. Rule-based programs can be used to solve Artificial Intelligence problems by use of heuristics. The code size is 150 KB.
- *Logik.* This applet allows the user to enter logic equations, performs all sorts of operations of them and display them in different manners. The code size is 204 KB.
- *Asternoids.* Nifty little shoot-'m-up game of 18 KB.
- *Hanoi.* Small applet featuring the ancient problem of moving different size rings from one pole to the next, while never placing a larger ring on a smaller ring, on any of the three poles. The code size is 14 KB.
- *AlexWarp.* This applet lets the user distort GIF-pictures in a predictable way by clicking and dragging the mouse on the picture. The code size is 9 KB.

Dynamic distribution for the benchmark suite

The six selected applets were run with a special command line parameter that causes information about all executed bytecodes to be sent to standard output. This output was filtered on the fly by a Perl-script that processed this data. Using the same categories that were used in the previous table, the profiling results are as follows:

Table 3: Dynamic distributions for the six benchmark programs. Numbers are rounded.

	DigSim	Jess	Logik	Asternoids	Hanoi	AlexWarp
Load	36.7	38.4	37.7	35.4	37.2	39.1
load	36.5	37.4	37.3	32.9	36.9	37.5
ldc*	0.2	1.0	0.4	2.5	0.3	1.6
Field load	22.4	18.5	15.1	21.1	18.6	18.1
*aload	6.0	4.7	3.1	5.0	1.4	6.4
get*	16.4	13.8	12.0	16.1	17.2	11.7
Store	5.9	6.9	8.8	6.7	7.7	9.5
*astore	0.3	0.3	0.4	1.8	0.2	1.7
store	4.4	5.7	6.3	2.8	4.5	7.3
put*	1.2	0.9	2.1	2.1	3.0	0.5
Compute	7.6	3.9	7.5	9.2	3.9	8.9
i&l	4.0	1.5	5.1	3.9	3.2	4.4
f&d	0.1	0.0	0.0	2.6	0.0	0.0
convert	0.4	0.1	0.2	0.9	0.0	0.0
iinc	3.1	2.3	2.2	1.8	0.7	4.5
Branch	11.6	13.7	10.6	10.0	8.6	10.5
Call / return	7.5	10.2	12.0	6.0	16.1	6.6
invoke*	4.3	5.1	6.5	3.6	9.2	3.8
*return	3.2	5.1	5.5	2.4	6.9	2.8
Push const	6.2	5.5	5.0	9.0	4.3	4.6
const						
Misc.stack	0.5	0.6	1.1	1.4	1.4	0.2
New Object	0.3	0.2	0.5	0.2	0.5	0.1
Others	1.3	2.1	1.8	0.9	1.7	2.5
Total	100.0	100.0	100.0	100.0	100.0	100.0

If these numbers are compared to the column *Dynamic frequency* in table 2 it seems, at first glance, that these numbers match rather well. Note that the categories **Branch** and **Call/return** post higher numbers for the benchmark programs. This is important because these bytecodes take quite a few cycles. Before this new data is used to refine table 2, the following diagram shows the difference between Sun's numbers and the new data. The dynamic frequency for every bytecode category is simply the average of the six benchmark programs.

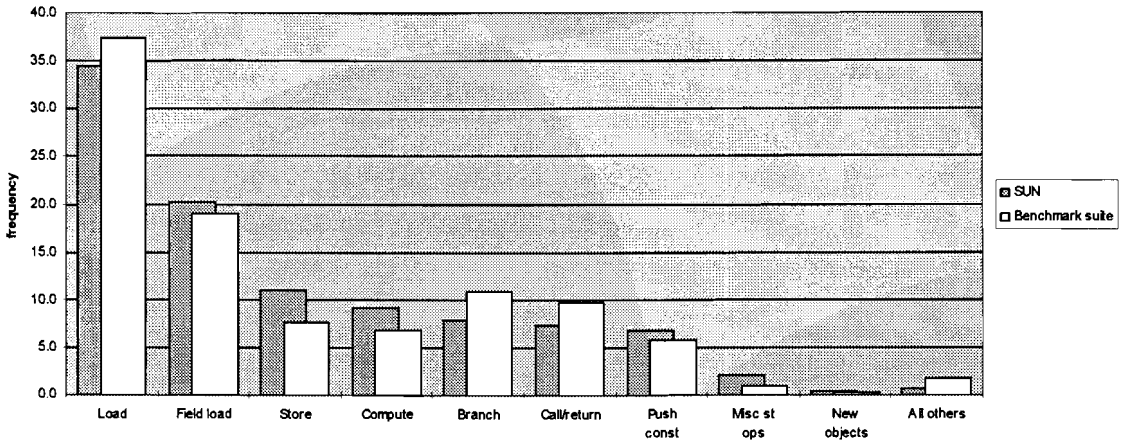

Figure 7: Dynamic distribution for the benchmark suite and Sun's test programs

The refined version of table two is as follows:

Table 4: Refined version of table 2.

Byte code	Dynamic freq.	VMI cycles (VC)	time_VMI	SW cycles	time_SW
Load **load** *ldc**	37.4	2	74.9	53	1984
Field load **aload* *get**	19.0	21	398.0	66	1251
Store **astore* **store** *put**	7.6	2	15.1	53	401
Compute *i&l* *f&d* *convert* *iinc*	6.8	4	27.4	61	417
Branch	10.8	20	216.9	65	705
Call / return *invoke** **return*	9.8	50	487.7	95	927
Push const **const**	5.7	2	11.5	53	305
Misc.stack	0.9	4	3.4	61	52
New Object	0.3	75	22.0	120	35
Others	1.7	50	85.8	95	163
Total	100.0		1342.7		6240

The resulting speedup is a factor of (6240 / 1342.76) is **4.6** times. Although Sun's numbers for the dynamic distribution seem to match well, the estimated performance gain decreases from 5.3 times to 4.6 times.

The final improvement of this table uses more detailed bytecode categories. The number of VMI cycles is specified for every part of a category. The estimates for the number of VMI cycles each type of bytecode takes are slightly changed on

account of improved insight into VMI-internals. Some of the estimates from [MML] turned out (or seem at this point) to be too optimistic. In addition to the categories introduced so far, some categories are sub-divided in *simple* and *complex*. This is done because some of these bytecodes need much more cycles to complete than others. For several categories, the numbers in this table are still estimated.

Table 5: Refinement of table 4.

Byte code	Dynamic frequency	VMI cycles (VC)	time_VMI	SW cycles	time_SW
Load	37.3		202.1		2493
load	36.3	5	182.2	65	2368
*ldc**	1.0	20	19.9	125	125
Field load	18.9		312.5		1165
**aload*	4.4	5	22.2	50	222
*get**	14.5	20	290.3	65	943
Store	7.6		62.2		492
**astore*	0.8	5	3.9	65	50
store	5.2	5	25.8	65	336
*put**	1.6	20	32.5	65	106
Compute	6.8		159.4		485
i&l(simple)	3.2	2	6.4	53	169
i&l(complex)	0.5	100	50.5	145	73
f&d(simple)	0.3	100	28.2	145	41
f&d(complex)	0.2	150	27.0	195	35
convert	0.3	5	1.3	65	17
iinc	2.3	20	46.0	65	150
Branch	10.8	20	214.7	65	698
Call / return	9.8		609.4		1048
*invoke**	5.4	100	544.8	145	790
**return*	4.4	15	64.6	60	258
Push const	5.8		11.5		305
const	5.8	2	11.5	53	305
Misc.stack	0.9	4	3.4	49	42
New Object	0.4	100	42.0	145	61
Others	1.7		106.2		186
(simple)	0.3	4	1.2	61	18
(complex)	1.4	75	105.0	120	168
Total	100.0		1723.8		6975

The new estimated speed-up is obtained by dividing the total time it would take a benchmark program to complete on a software interpreter with the same number for the VMI. The result is a speed-up of (6975 / 1723.8) is **4.0** times. However, this is a conservative estimate. Keep in mind that the numbers presented in this section are applicable to the version of the VMI as introduced in [MML]. This means the VMI is a **bus peripheral** with a very simple cache and very simple register stack management (i.e. flushing or restoring half the register stack). Several optimizations, like choosing a different architecture option, can further improve the speed-up. Preliminary research indicates that the speed-up could be improved to **7** times.

3.4.3 Amdahl's Law applied to the VMI

During the execution of a program, not all time is spent interpreting JBC. If the program invokes a native method, which is usually optimized platform-specific code for directly accessing the hardware, the JBC interpreter is halted until the native method finishes. Spending time on other code than JBC affects the speed-up according to the following formula, also known as Amdahl's Law:

$$T = \frac{1}{\left(1 - a + \frac{a}{s}\right)}, \{a \in [0,1]\}$$ Formula (1)

T is the resulting speed-up, *a* is the part of the program that is affected and *s* is the speed-up factor. The following figure shows the resulting speed-up for different values of *a* and *s*.

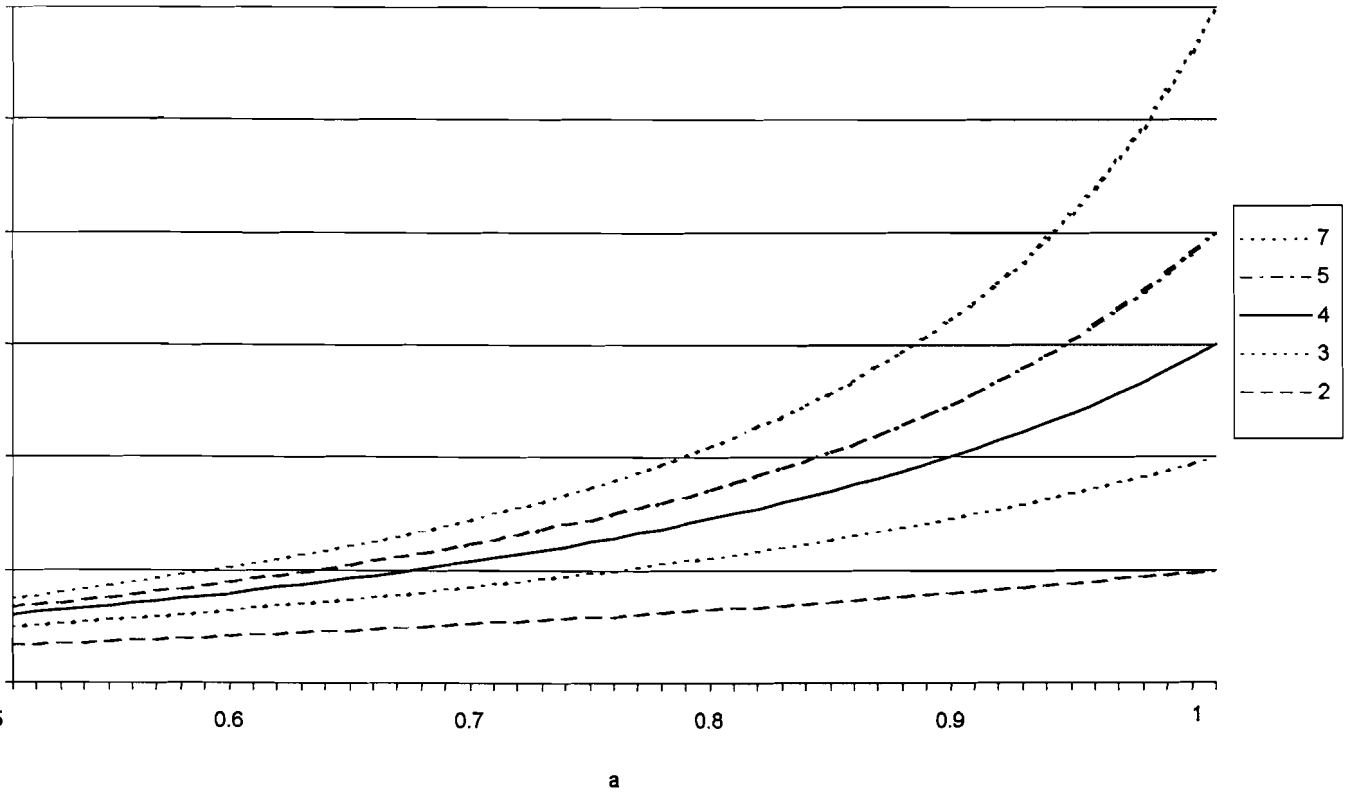

Figure 8: Speed-up T for different values of a *and* s

Before applying Amdahl's Law to the VMI and the benchmark suite, there is one more issue to take a look at. One of the reasons Java programs can be smaller than other programs is because the user is expected to have a set of libraries installed on the client-side. Java programs consist of three parts:

1. JBC coded by the application programmer.
2. JBC in the class libraries, originally coded by SUN.

It is possible to compile (part of) the class libraries for your platform and store this compiled source code instead of the Java code. This will be called precompilation throughout this report. Precompilation would compromise the platform independence but in some embedded markets this is less of an issue. The following two examples illustrate which considerations have to be taken into account:

1. A handheld PC. Sun updates the class libraries every couple of months. New functionality is added and old classes are replaced. Precompilation is not very viable because the user wants to (download and) use the latest version of Java. However, perhaps a part of the class libraries will be 'stable' enough to be considered as a target for precompilation.
2. A set-top box. The functionality is built-in and not likely to change a lot in the future. Flexibility, portability and platform-independence are less of an issue, so it is advantageous from a performance point of view to precompile as many of the class libraries as possible. It would be interesting to know which part of the class libraries can be precompiled, but that information is not available.

Assume a fraction *a3* of the execution time is spent in native methods and a fraction *a2* is spent in class libraries. Furthermore, assume the VMI can accelerate all JBC, be it class library code or programmer's code, by a factor *s1* and further assume that precompiling class libraries accelerates their execution time by a factor *s2*. The following formula gives the resulting speed-up in that case:

$$T = \frac{1}{\left(1-\left((1-a3-a2)+a2\right)+\frac{(1-a3-a2)}{s1}+\frac{a2}{s2}\right)}, \{a3, a2 \in [0,1]\} \qquad \text{Formula (2)}$$

This formula can be simplified by introducing a factor *a1* for JBC coded by the application programmer. The following formulas hold:

$$((a1, a2, a3) \in [0,1]) \wedge (a1 + a2 + a3 = 1) \qquad \text{Formula (3)}$$

$$T = \frac{1}{\left(a3+\frac{a1}{s1}+\frac{a2}{s2}\right)} \qquad \text{Formula (4)}$$

Formula (4) is used if the class libraries are precompiled.

$$T = \frac{1}{\left(a3+\frac{(a1+a2)}{s1}\right)} \qquad \text{Formula(5)}$$

Formula (5) is used if the class libraries are not precompiled.

The dynamic behaviour of benchmark programs w.r.t. these types of code is shown in the following figure.

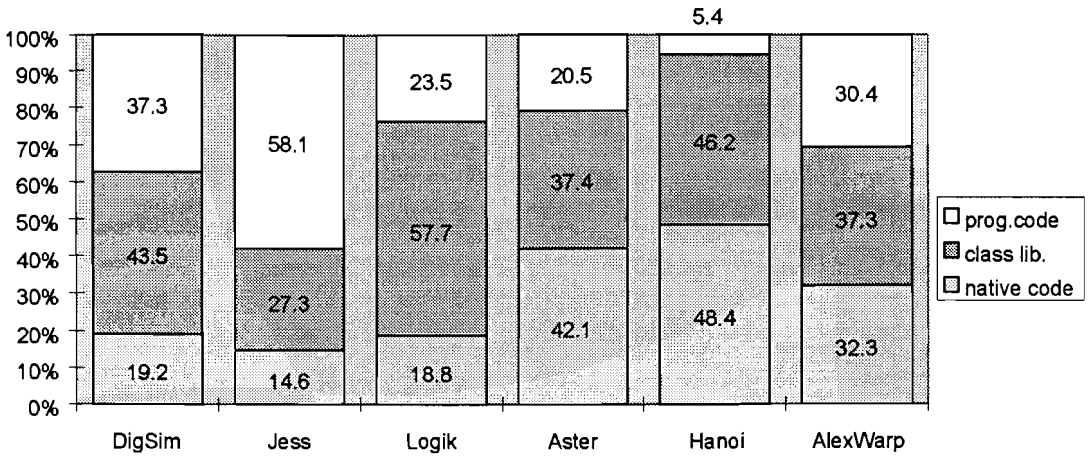

Figure 9: Percentages of execution time spent on programmer's code, class libraries and native code.

Figure 9 shows that the dynamic behaviour of programs can vary a lot. Generally, larger applications execute less native code than smaller applications. In the following two pie charts, the six applications are subdivided on account of their code size.

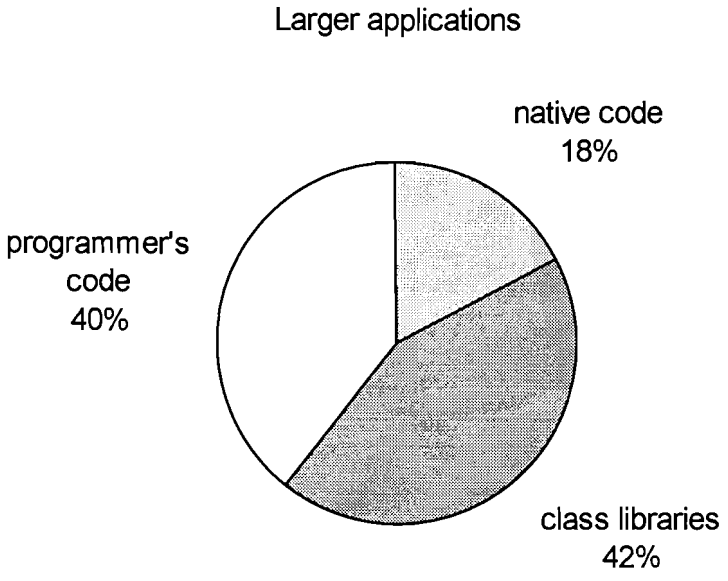

Figure 10: Average dynamic behaviour of benchmark programs DigSim, Jess and Logik

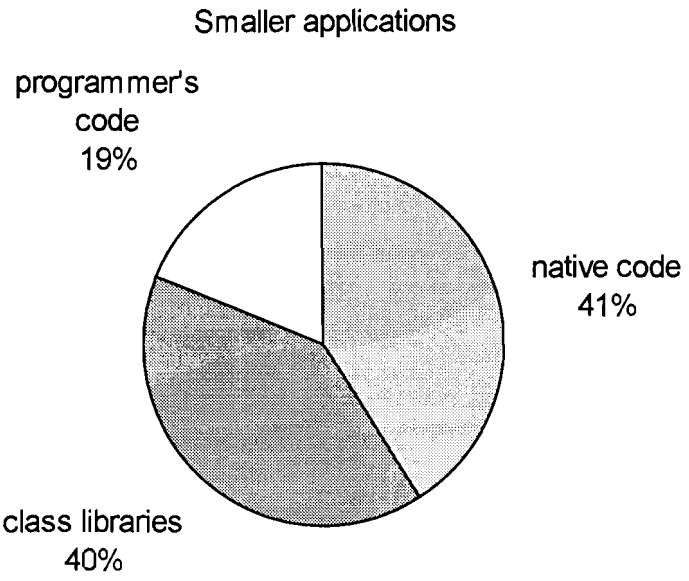

Figure 11: Average dynamic behaviour of benchmark programs Asternoids, AlexWarp and Hanoi

Suppose the programmer code is accelerated by a factor 4.0, as determined earlier in this section, and suppose precompilation accelerates the class libraries twenty times, as determined by previous research done at PRLE. The following figure gives the resulting speed-up for three different options:

1. Not using VMI but only precompiling the class libraries
2. Using VMI for all JBC but not using precompilation
3. Using both VMI and precompilation

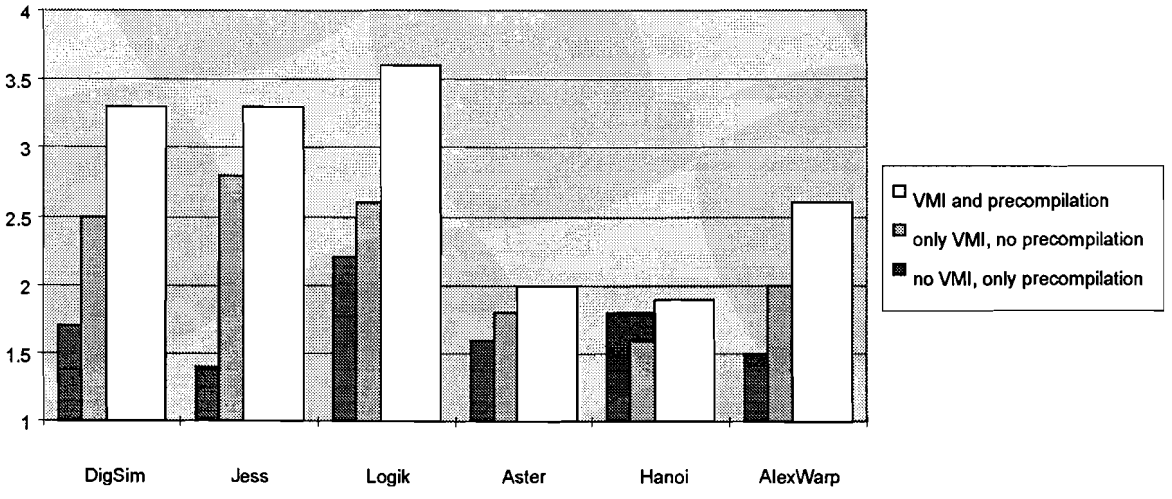

Figure 12: Estimated performance gain for different options

It is interesting to notice that the larger applications benefit more from the VMI because a larger part of the execution time is spent in the programmer's code.

The results are grouped in the following table. The first three rows list the speed-up compared to a software interpreter. The last row states the ratio of rows 1 and 3 which means: assume precompilation is a given, this number is the speed-up that results from using the VMI. The numbers in bold font are repeated in the following section as part of the conclusions.

Table 6: Speed-up factors for the use of VMI (as a bus peripheral) and/or precompilation

	Larger applications	*Smaller applications*
No VMI, only precompilation	1.7	1.6
Only VMI, no precompilation	**2.6**	1.8
VMI and precompilation	**3.4**	2.1
Ratio: with/without VMI	**2.0**	1.3

3.5 Discussion and Conclusions

It is important to acknowledge that not all the parts of the time spent executing a program can be speeded up in the same way. It is possible to accelerate pure JBC by using the VMI. It is possible to precompile class libraries, although this has disadvantages as well. Furthermore, it is possible that dedicated hardware for a particular embedded system can improve the performance, compared to a desktop system, in one particular area such as class loading, graphic processing or garbage collection. If one of these particular areas is improved, a larger *percentage* of the execution time is spent on pure JBC. This in turn means the VMI effectively speeds up an application more. The conclusions are as follows:

- The estimated speed-up for the current model (i.e. VMI as a bus peripheral), as introduced in [MML], appears to be **4.0** instead of **5.3** for pure JBC for two reasons:
  1. Typical programs that interest HCG have slightly different characteristics compared to previous test programs.
  2. The numbers from [MML] are less specific and slightly optimistic in some cases. Also, improved insight into VMI-internals has led to more detailed information about the number of cycles different bytecodes need to execute.
- The speed-up of a complete program differs from the speed-up for pure JBC as stated by Amdahl's Law. This was previously not taken into account. The VMI speeds up the total executing time of an average larger application by a factor of **2.6** times, if no part of the class libraries is compiled prior to execution.
- If the class libraries are compiled prior to execution, the total executing time of an average larger application speeds up by a factor of **3.4** times compared to a regular software interpreter. Without the VMI, this would be **2.0** times slower.

4. STRUCTURAL MODEL OF THE VMI CORE

This chapter provides a look at how the structural model of the VMI evolved. Features of JBC necessitate additions to the model, as do exception processing and VMI specific features. The chapter starts with a look at the unpipelined model of the VMI core and an introduction to pipeline design. Gradually the model will be completed throughout this chapter. Some features are not discussed, or not in depth, because they are based on proprietary information. A detailed description can be found in [VMI].

4.1 INTRODUCTION

In chapter 3 the functional model of the VMI is introduced. A functional model is not suitable for developing hardware. HCG wants to have a structural model designed of the VMI in a hardware description language. Ultimately, HCGs goal with regard to this project is to have the VMI accelerate Java applications that are running on a handheld computer that is powered by HCGs flagship 100 MHz system-on-a-chip: *Poseidon*. Having a structural model of the VMI is a step in that direction. The differences between a functional model and a structural model are far from trivial. This chapter discusses the development of a structural model of the VMI core.

The following section explains how the VMI core works. Because of the performance requirements for the VMI, the core will be designed using a technique called *pipelining*. Pipelining is a technique to improve the throughput of an architecture. Designing a pipeline starts with understanding the unpipelined model of the architecture at hand.

> *The goal of this chapter is to come to a structural model of the VMI core that has the same capabilities as the functional model.*

4.2 UNPIPELINED MODEL OF THE VMI CORE

For the VMI core, executing a bytecode means translating it into MIPS instructions and placing these instructions in a FIFO. The following tables drive the translation of JBC to MIPS instructions:

- The Translation Table (TT) contains a sequence of instruction skeletons and micro code for each bytecode.
- The Code Index Table (CIT) contains the index in TT where a sequence of instruction skeletons start. Because the MIPS instruction sequences for different bytecodes have different lengths it is not possible to deduce the starting point in TT only from the bytecode.
- The Pop Table (Pop) contains the number of items each bytecode pops from the operand stack.
- The Push Table (Push) contains the number of items each bytecode pushes on to the operand stack. This table and the previous table are used to determine whether the register stack will overflow or underflow when the bytecode is executed with the current stack state. If this is the case, stack maintenance must take care of this problem. Although this could theoretically be done in many ways, the VMI simply flushes or restores half of the register stack prior to translating the bytecode. The reason to select this strategy can be found in [MML].

- The Argument Table (Arg) contains the number of argument bytes that follow the opcode byte for each bytecode.

The following figure depicts an initial version of the unpipelined VMI core. This version is not fully functional, it is only meant to give the reader a general feel for how the VMI works. A functional model that complies with the specifications in [MML] is available. The following paragraph explains how the initial model works.

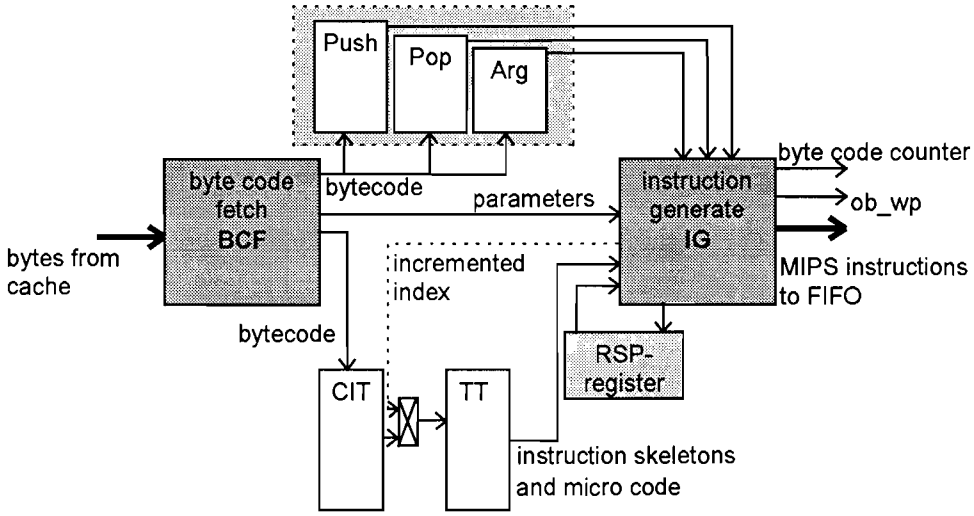

Figure 13: Initial version of unpipelined VMI core

The white rectangles are tables. The larger two dark gray rectangles are combinatorial logic. The small light gray rectangle labeled 'RSP-register' is the register that holds the value for the register stack pointer. The arrows indicate the flow of data.

The model works as follows. Based on the value of the signal labeled *bytecode counter* (bcc), the bytecode cache supplies a bytecode to the gray block labeled 'bytecode fetch' (BCF). This block passes the bytecode on to CIT and to the three tables up in figure 13. CIT gives the index in TT where the instruction skeletons and micro code for this bytecode start. TT passes only one instruction skeleton and one bitvector with micro code at a time. Logic block 'instruction generate' (IG) must increment the index for TT until all the instruction skeletons and their micro code for the current bytecode are passed to IG. The dotted arrow indicates the incremented index. The logic that helps TT choose between the indexes from CIT and IG is drawn as a small rectangle in between CIT and TT.

Using the information from the *RSP*-register, Push and Pop, the block labeled 'instruction generate' (IG) determines if there is a stack overflow or underflow if this bytecode is translated with the current stack state. If this is the case, IG must first generate instructions to deal with the stack overflow or underflow. One of the possibilities to do so is to flush or restore half of the stack. The reader is referred to [MML] for information on different possibilities to handle stack maintenance.

Using the Argument Table, IG determines how many operands the bytecode needs. Sequentially, *bcc* is increased and BCF passes the parameters to IG. If the translation is finished, *RSP* is set to the proper value, as is *bcc*. Finally, throughout the translation, the *output buffer write pointer* (*ob_wp*, the index into

the instruction FIFO) must be updated along with every instruction that is generated.

The following section explains what pipelining is and how to apply this technique to a design.

4.3 PIPELINE DESIGN

The concept of pipelining is explained very well in [H&P]. The information in this paragraph can be found there in more detail.

Pipelining the execution of instructions is an implementation technique whereby multiple instructions are **overlapped** in execution. Each step in the pipeline completes a part of the execution. Different steps are completing different parts of different instructions in parallel. Each of these steps is called a *pipe stage*. The pipe stages are connected one to the next to form a pipe. Instructions enter at one end, progress through the stages and exit at the other end. Because all stages proceed at the same time, the slowest stage determines the speed at which the pipeline can operate. The designer's goal is to balance the length of each pipeline stage. With traditional pipelining for the CPU come three types of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the program counter.

In the case of the VMI, things are somewhat different but all three types of hazard could occur. Much of the design effort concentrates on these hazards.

4.3.1 Pipe stages and stage boundaries

Once it is clear how the unpipelined model works it is possible to define the pipe stages. The line that divides two pipe stages is called a *pipe stage boundary*. Registers are logically placed on all stage boundaries to transfer data from one stage to another. In VMIs (simulated) reality, all the registers, as well as the instruction FIFO are combined in a *register set* or *register file*, a part of the support module of the VMI. In the figures in this chapter, registers are drawn on their logical position, rather than as a part of the register file.

The designer should keep the data path and the control path separated. The original *control state machine* is divided into several smaller control state machines for each pipe stage. In general, a pipe stage could look like the following figure: (each vertical dotted line indicates a pipe stage boundary)

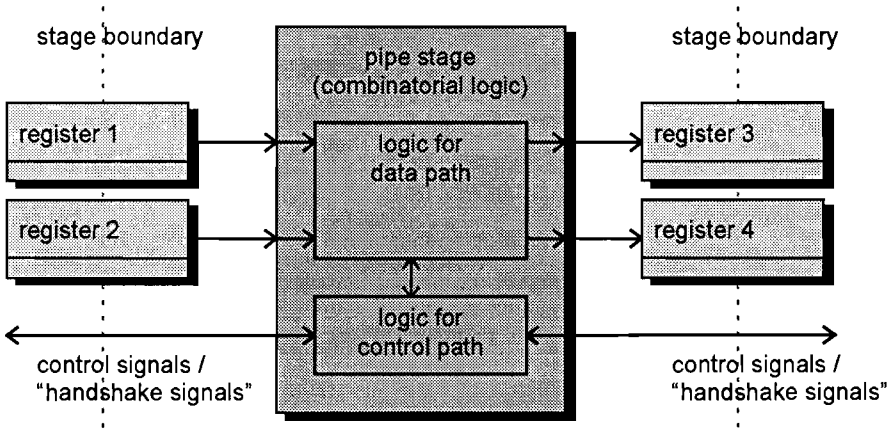

Figure 14: Generic model for a pipe stage

Not all data comes from registers. Some data comes from a table that is indexed by the value in a register. Because interfacing with a table is usually difficult, it is good practice not to access more than one table serially in one pipe stage. Indexing a table with a value in a register synchronizes the flow of data because all registers are triggered by the clock signal. With the addition of tables, a pipeline could look like this:

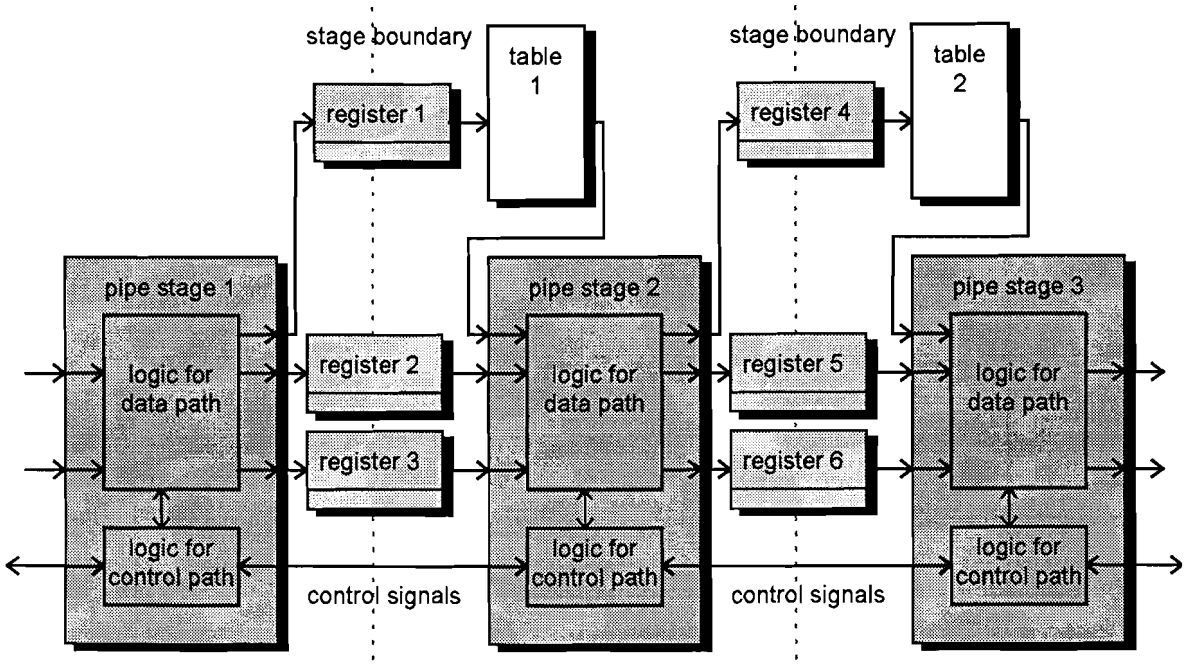

Figure 15: Generic model for a pipeline that uses tables

4.3.2 Pull mechanism

The dataflow in a pipeline can be controlled in two ways.

1. A *push mechanism* means data is pushed through a pipeline from left to right.
2. A *pull mechanism* means data is pulled through the pipeline from right to left.

In the case of the VMI, the rightmost pipe stage is actually the instruction issue module that sends instructions to the CPU when requested. So, the pull mechanism is chosen for the pipelined VMI core.

An illustrative way of looking at the pull mechanism is as follows. The control signals flowing to the right are called *cookie* and the ones flowing from right to left are called *hungry*. A pipe stage can only generate new output (i.e. a *cookie*) if the next stage is *hungry* (i.e. ate the previous *cookie*). Now, if the CPU does not want to receive any instructions, the rightmost *hungry* signal will become inactive. The control logic for the rightmost pipe stage will propagate this signal from right to left, until all *hungries* are inactive. Conversely, if a pipe stage cannot process the data on its input (i.e. eat the *cookie*) it will signal to suspend the data flow (i.e. tell the previous stage it is not *hungry*).

4.3.3 Initial version of a pipelined VMI core

The following figure is a refinement of figure 13. In this model the design rules from the previous sections are applied. The following paragraph explains how the model works. This model is not complete but forms an interim step towards a complete model.

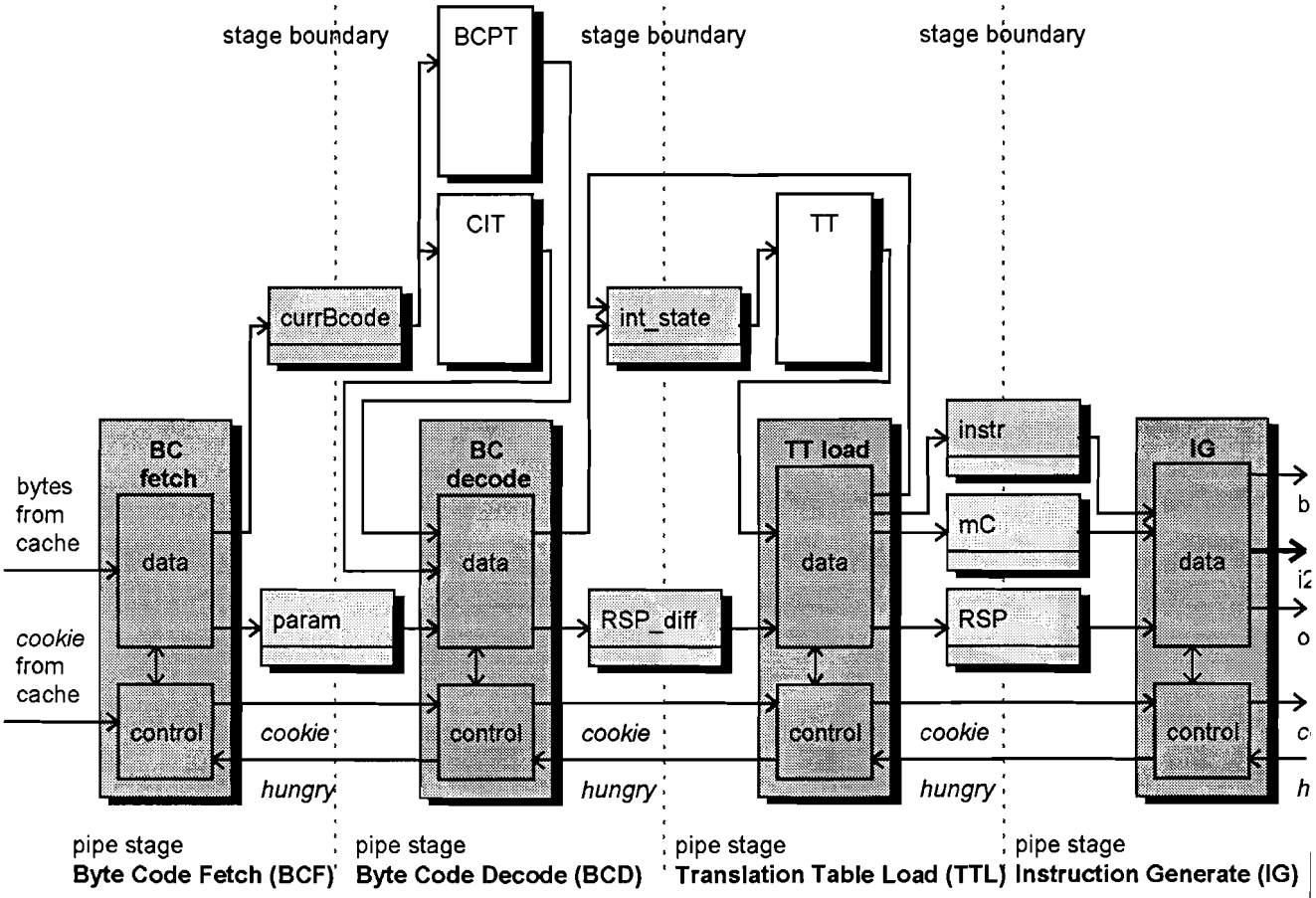

Figure 16: Initial version of a pipelined VMI core

The tall white rectangles are tables. The larger four dark gray rectangles are blocks with combinatorial logic with the same name as the pipe stage they are located in. The five smaller light gray rectangles are the registers VMI uses. The arrows indicate the direction of the flow of data or control signals. Not all connections are drawn. Inputs enter a logic block or register on the left side and outputs exit on the right side.

Based on the value of *bcc*, the bytecode cache supplies a bytecode to the leftmost pipe stage called *BC fetch*. The signal labeled '*cookie* from cache' is set, so the pipestage knows it should put the byte it just received in register *currBcode*, which means *current bytecode*.

The three tables that are bounded by a light gray box in figure 13 are combined in one new table: Byte Code Properties Table (BCPT), which is indexed by the *currBcode* register. Each of the four types of information that is stored in BCPT can only assume a limited number of different values. If this number is n, this type of information can be encoded using $\lceil^2 \log n\rceil$ bits. For every entry, the bits that signify each type of data are simply concatenated in a set order. The BCPT feeds bytecode dependant data, such as the number of arguments, pops and pushes to the second pipe stage which is called *BC decode*. Based on this information and the value of the *RSP* register this stage can determine if the translation of this bytecode would cause a stack overflow or underflow. The connection between the output of the *RSP* register and *BC decode* is not drawn.

The difference between the number of pops and pushes is stored in the *RSP_diff* register, which is subsequently used in stage *TT load* to update the value of the *Register Stack Pointer* register. The connection between the output of the *RSP* register and *TT load*, needed to update the *RSP* register, is not drawn. The connection between the output of the *mC* register and *TT load* is not drawn.

BC decode passes the index for the Translation Table, which it receives from CIT, to the *int_state* register. This name is chosen because this value is in fact the **state** machine variable that controls the inner switch, as discussed before. Each entry in TT consists of a **pair** of one instruction skeleton and one bitvector of micro code. The output of TT is passed to two registers by *TT load*. The instruction skeleton is stored in the *instr* register. The micro code is stored in the *mC* register. *TT load* increments the value in *int_state* and indexes the next entry of TT. The logic that helps the *int_state* register choose between the two input values is not drawn. In fact, it is probably better to pass the incremented value of *int_state* to *BC decode*. Then the control logic in *BC decode* can decide which value to store in the *int_state* register and the *int_state* register has only one input. All registers, as well as all tables, should preferably have only one input.

From this figure it is not clear what should happen if *BC decode* detects a stack overflow or underflow. *IG* must first generate a sequence of MIPS instructions that deal with the stack overflow or underflow and then proceed to translate the current bytecode. One of the following sections will discuss a possible solution for this problem.

IG uses the data from BCPT, the *param* register, the *RSP* register, *instr* register and *mC* register to decode the register fields and possibly the immediate field in the instruction skeleton to produce a MIPS instruction. This instruction is placed on the output labeled *'i2FIFO'*, together with a proper value for *ob_wp*. The connections between BCPT or the *param* register and *IG* are not drawn.

Finally, after the translation is complete, *bcc* must be incremented to point to the next bytecode. Astute readers will have noticed that the leftmost pipe stage only performs any actual work at the beginning of the translation of a bytecode to fetch the opcode bytes and any operand bytes. Arguably this pipe stage is not really a part of the pipeline. Additions to the model, made in subsequent sections, will further change the role of the leftmost stage and all other stages too. There are limits to the extent to which it is possible to start translating the next bytecode before the current bytecode is fully translated.

Not all control signals are drawn. The *cookies* and *hungries* behave like described before.

4.4 JBC SPECIFIC FEATURES

The initial model of the VMI core is introduced in the previous section. This model does not have the same capabilities as the functional model. In this section and the next two sections, more features will gradually be added. The features in this section are needed because of characteristics of JBC.

- The first characteristic is that all bytecodes use four or less operand bytes. This means it is possible to supply the core with both the opcode byte and all the operand bytes it needs to translate a bytecode at the same time. The VMI will not need to increment *bcc* many times to separately fetch every operand byte with the addition proposed in section 4.4.1.
- The second problem comes from a peculiarity of the JBC machine language. The bytecode *wide* changes the behaviour the JBC instruction that follows. Section 4.4.2 explains how the structural model deals with this problem.

4.4.1 Fetching parameters

Two bytecodes, *tableswitch* and *lookupswitch*, have a variable number of argument bytes. All other bytecodes use zero to four operand bytes. *Argument* bytes, *operand* bytes and *parameter* bytes are all used for the same thing. In the initial model, parameters were fetched separately when needed. In the enhanced version of the model, the bytecode cache fetches one opcode byte and four operand bytes at a time. Since *tableswitch* and *lookupswitch* are executed in a proprietary manner, all other bytecodes can now be executed by the VMI without separately fetching parameters. *BC fetch* will have separate inputs for opcode bytes and operand bytes as indicated in the following figure. If a bytecode uses zero operand bytes, the *param* register is simply not used.

The bytecode cache sends a control signal called '*cookie* from cache' to the block *BC fetch*. If the cache's outputs for the opcode byte and the parameter bytes are consistent with the cache's input, which is *bcc*, then the control signal is active. Otherwise it is inactive. The following figure depicts the new *BC fetch* pipe stage.

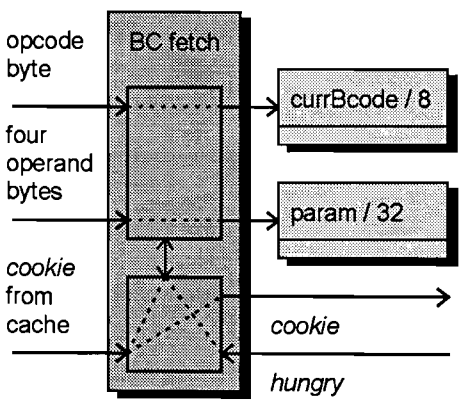

Figure 17: BC fetch pipe stage

4.4.2 Control for *wide* bytecode

The *wide* bytecode modifies the behaviour of another bytecode. It takes one of two formats, depending on the bytecode being modified. The first form of *wide* modifies one of the bytecodes *iload*, *fload*, *aload*, *lload*, *dload*, *istore*, *fstore*, *astore*, *lstore*, *dstore* or *ret*. The second form applies only to the *iinc* bytecode. In either form, *wide* is followed by the bytecode it modifies and two operand bytes that form a 16-bit index to a local variable in the current frame. In the second form, the next two operand bytes form a signed 16-bit constant. The modified bytecodes operate as normal, except for the use of the wider index and, in case of *iinc*, the larger increment range.

The entries for *iinc* in TT are changed to accommodate both the regular and the wide format. The VMI must remember whether the previous bytecode was *wide*, so that pipe stage *IG* knows it should use more parameters than the number of arguments as given by BCPT. This is accomplished by adding a one bit register on the pipe boundary between *BC decode* and *TT load*, called *wide_bit.* The only action of a *wide* bytecode is to set the *wide_bit*. Every other bytecode resets it.

The number of arguments from BCPT is stored in a register called *no_args* on the boundary between *BC decode* and *TT load*. Next, *param* (which is an output of logic block *BC fetch)*, *no_args* and *wide_bit* are used to form a new *param* register that replaces the old version. The proper number of bytes are placed in the rightmost position within this 32-bit register. This means pipe stage *IG* no longer needs to check the number of arguments from BCPT while generating MIPS instructions or updating *bcc*. Instead, the entire new *param* register can be used, since unused bits will be set to zero. The following figure depicts how this

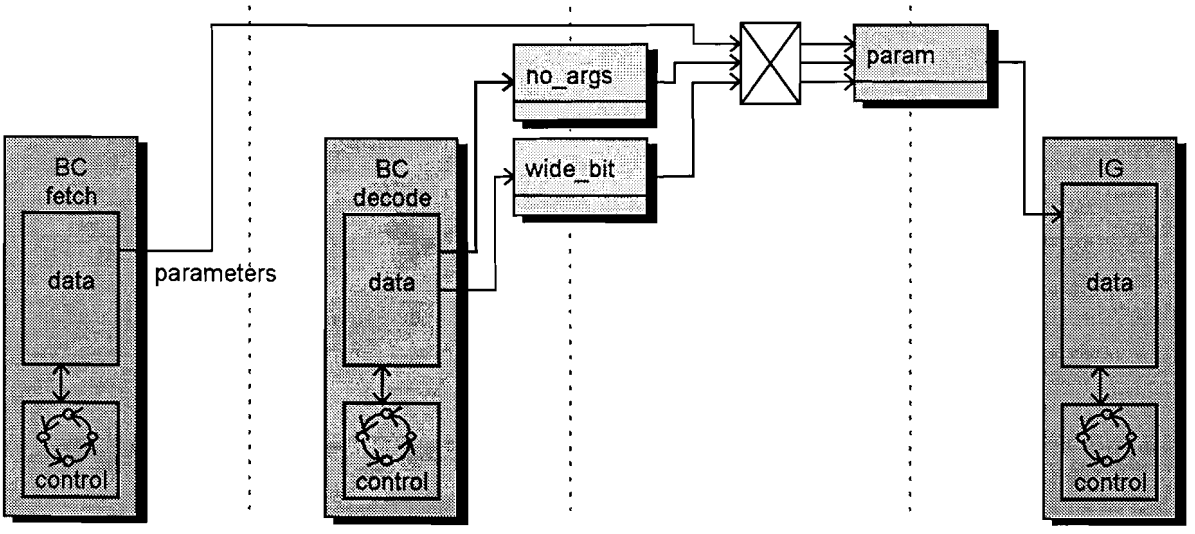

Figure 18: New param register and control for wide bytecode

changes the model.

4.5 CONTROL FLOW

There are three issues introduced in this section. The solutions for the first two issues are needed for most implementations of a JBC interpreter in hardware, not just the VMI. The third issue is specific for the VMI **as a bus peripheral** and deals with the communication between the VMI and the CPU.

- The first issue is directing the inner switch. So far, it is unclear how TT is subsequently indexed with new values in order to feed one instruction skeleton after another to logic block *IG*. This is explained in section 4.5.1.
- The second issue that is essential to the VMI core is asking the support module for the right instruction at the right time. Due to the fact that the VMI actually performs conditional and unconditional **jumps**, with aid of the CPU, it is an important issue when and how to update the value of *bcc*. This will be explained in section 4.5.2.
- The third issue is an intrinsic feature of the VMI core. The VMI can communicate with the CPU thanks to a special mechanism that is explained in section 4.5.3.

4.5.1 Directing the inner switch

At the start of the translation of a bytecode, *BC decode* stores the index in the Translation Table, which it receives from CIT, in a register called *int_state register.* After the first instruction skeleton has been used to generate the first of a sequence of MIPS instructions, the *int_state* register should be incremented on every clock cycle until the sequence is finished. If the sequence is finished, a bit in the micro code called *last instruction* is set.

To increment *int_state*, the output of the *int_state* register is by default incremented by one and connected to a new input of *BC decode*. *BC decode* multiplexes these this new input and the index from CIT with the aid of micro code *last instruction* (*last_instr*).

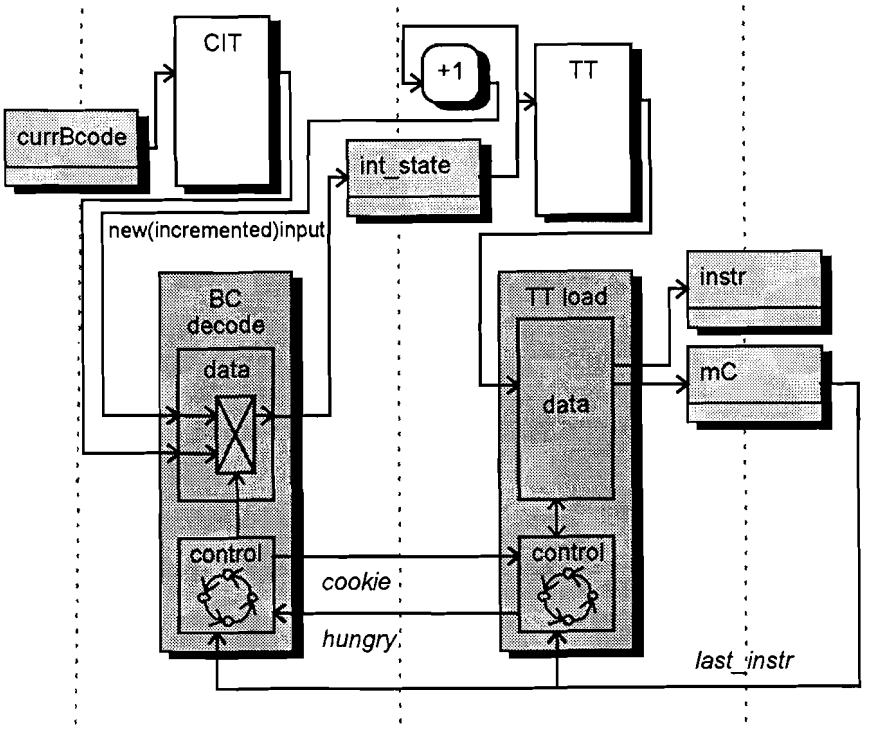

Figure 19: Directing the inner switch using int_state. BC decode *can select an incremented value of the* int_state *value and pass it to the* int_state *register.*

4.5.2 Updating bcc

In the current model, the pipe stage *IG* has two separate main functions that will now be placed in separate logic blocks. Logic block *instruction generation* (*IG*) generates the MIPS instructions and logic block *address adder* (*AA*) updates *bcc*. This subsection takes a closer look at *AA*. Updating *bcc* involves two matters. For most bytecodes *bcc* must be incremented according to the number of operand bytes a bytecode has. Secondly, the VMI **performs** conditional jumps and unconditional jumps.

1. The first feature of *AA* can be implemented in several ways. The obvious solution would be to connect the *mC* register to *AA* and increment *bcc* as soon as control signal *last_instr* is active. However, sometimes *last_instr* is active but *bcc* must not yet be incremented. This happens for example with a register stack overflow. This will be explained in section 4.6.1 'Stack maintenance'. Therefore the control state machine of *BC decode* is extended (with a signal called *bcc increment* or *bcc_inc*) to increment *bcc*. The following figure depicts these additions.

2. The second feature of *AA*, performing jumps, was mentioned in section 3.3 'Functional model of the VMI'. The first bytecode involved is *goto*, an **unconditional jump**. Because executing *goto* only changes *bcc*, no MIPS instructions are generated. The Java Virtual Machine Specification, [JVM], states that *goto* has two operand bytes which are concatenated to form a signed 16-bit offset. Execution should proceed at this offset from the address of the *goto* instruction. Since the two operand bytes are already available in the right order, all that remains to be done is to add these 16 bits to *bcc*.

   To achieve this, one bit in the micro code, *add_par2bcc*, is active for the *goto* bytecode. The part of the pipe stage *IG* that updates *bcc* normally increases *bcc*

after the last instruction of the translation has been generated. However, if *add_par2bcc* is also active at the same time, the offset is added to *bcc* instead.

Theoretically, this bytecode needs no instruction skeletons. In the current implementation however, *int_state* indexes **pairs** of one instruction skeleton and one bitvector of micro code. A bit is added to the micro code bitvector to indicate if a MIPS instruction should actually be generated from the accompanying instruction skeleton. This bit is called *gen_instr*. Apart from the *goto* bytecode, there are more bytecodes that require this feature.

This completes the additions needed to accommodate *goto*. Because the pipe stage *IG* has just been split into two blocks, this part of the model now looks like this:

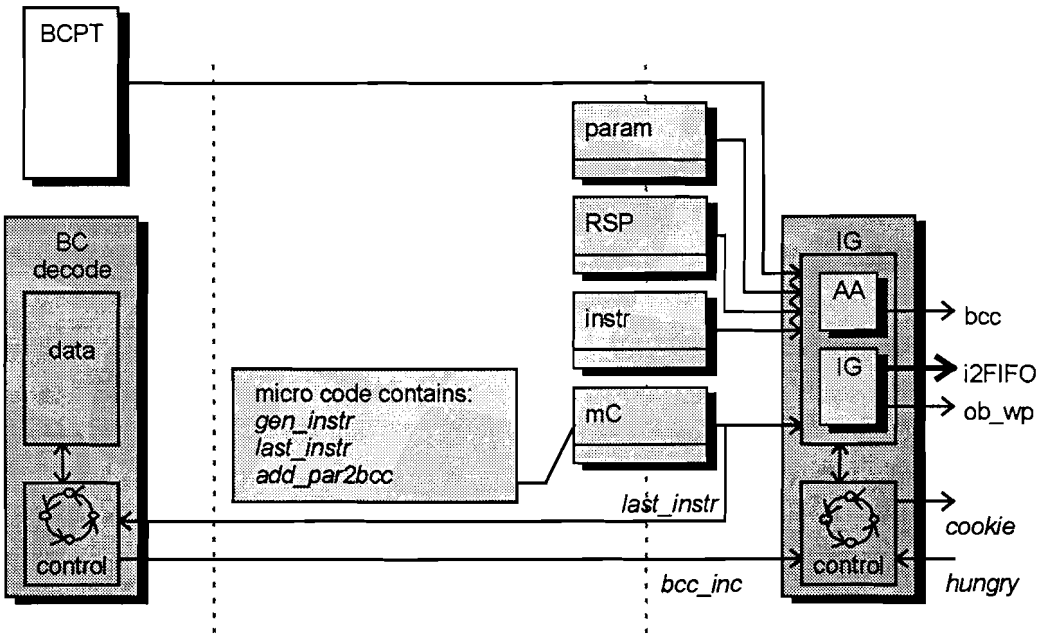

Figure 20: Additions to pipe stage IG for updating bcc

Conditional jumps are more complicated than unconditional jumps. Jumps are also called branches. With conditional jumps, one or two values on the operand stack are compared against zero or each other. If the comparison succeeds, the two operand bytes are combined to form an offset for *bcc*. As discussed before, the VMI can perform branches by generating instructions that cause the CPU to first check a condition and then write the result to one of VMIs registers. The VMI has to wait until it detects that the CPU has written a value to one of VMIs registers. How the mechanism to wait for an action by the CPU works will be treated in one of the following subsections. Depending on the resulting value, the branch is taken or not. A taken branch implies a jump in TT. If a branch is not taken, the index for TT is by default incremented by one. Only when the VMI is waiting for the CPU to write back a value, *int_state* can be incremented by a variable amount.

The following example is bytecode *ifne*, which first pops an integer off the operand stack. *ifne*'s two operand bytes are concatenated to form an offset value. If the integer is not equal to zero, execution branches to the offset.

Example

	Instruction skeletons	Micro code
1	SLTU(rsp-1,0,rsp-1)	
2	SW(rsp-1,*const4*,*vmi*)	
3	NOP	wait_CPU wait_ctrlVal(0) inc_int_state(2)
4	NOP	add_par2bcc no_instr last_instr
5	NOP	no_instr last_instr

A special program assembles and encodes these instruction skeletons and micro codes into bitvectors for TT. The first entry contains an instruction skeleton of a *Set On Less Than Unsigned* MIPS instruction. By default, *gen_instr* is set. Micro code *no_instr* is the inverse of *gen_instr*. The second entry contains an instruction skeleton of a MIPS instruction that stores the result of the first instruction at the memory location specified by base *vmi* and offset *const4*, which is a new register of the VMI called *ctrl* register. It usually takes some cycles before the CPU completes this instruction.

The third entry contains micro code *wait_CPU*. It stalls the VMI until it has received the value that was just stored. This value is compared with the *wait_ctrlVal*, zero in this case. If these values are the same, register *int_state* is incremented by the value stored in *inc_int_state*, in this case two. TT entry 4 would then be skipped and the next one would be number 5, which is only a *last_instr*. If the two values are not the same, *int_state* is by default incremented by one. If that happens, entry number 4 is used. Micro code *add_par2bcc* is set and logic block AA adds the offset in the *param* register to *bcc*.

End Example

The capability to make jumps within TT is useful for more bytecodes.

The following subsection discusses what additions to the model are needed to make it possible to wait for the CPU to write a result to one of VMIs registers.

4.5.3 Waiting for the CPU

The problem of enabling the VMI to wait for an action by the CPU is different from most problems with the VMI micro architecture. This is because the obvious approach to the problem meets with unexpected difficulties. Recall the logic block labeled '+1' in figure 19. Suppose this block is connected to the output of the *mC* register. Next, suppose this block would only increment *int_state* by one if micro code *wait_CPU* is inactive. By the time the logic block notices that *wait_CPU* is active, *BC decode* has already stored the new *int_state* value in the *int_state* register. Since this register connects to *TT load*, it is too late for changes in *int_state*. For similar reasons, it would not work to make changes to the state machine in the control logic block of *BC decode*. The solution is to place a new logic block, called *TTX*, between the *int_state* register and the *TT load* data logic block. This new block is mainly controlled by *wait_CPU*. The following figure shows the position of *TTX*.

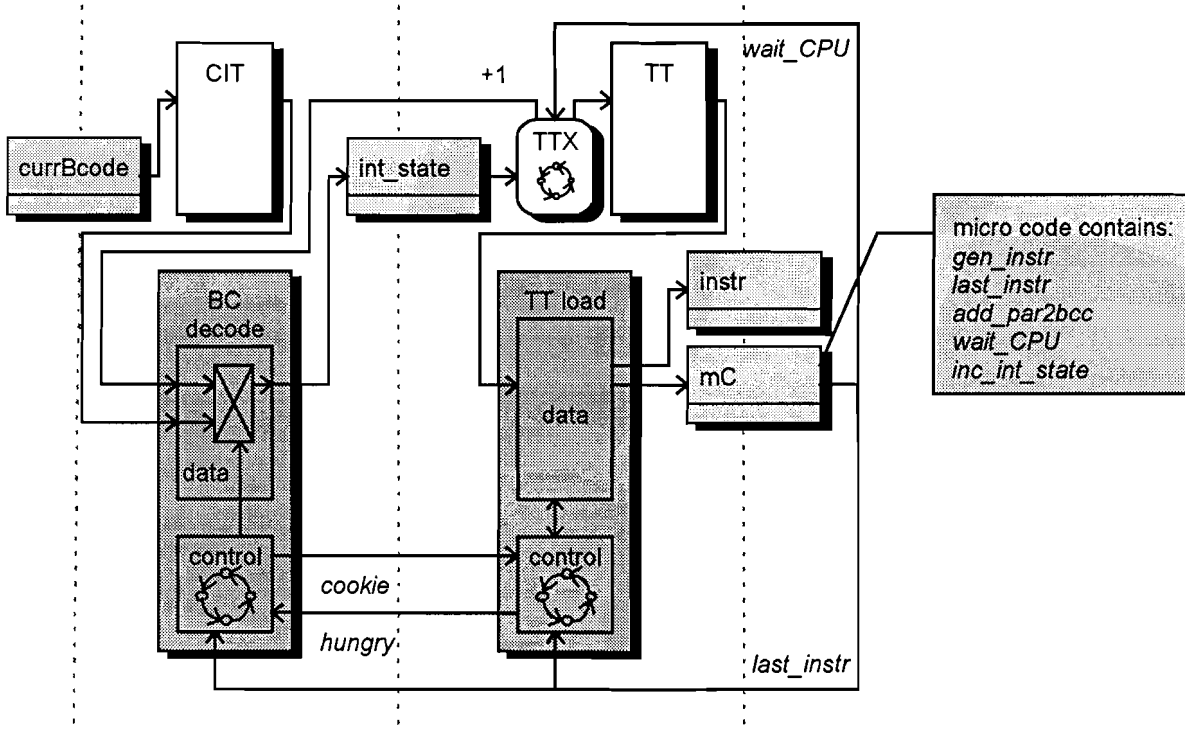

Figure 21: New logic block enables waiting for the CPU

4.6 EXCEPTION PROCESSING

- The VMI maps the top of the operand stack on CPU registers. Before the VMI begins the translation of a bytecode, it checks whether this would cause a register stack overflow or underflow. Section 4.6.1 suggests a possible way to extend the current model to facilitate stack maintenance.
- The VMI can act as a bus master and cause a deadlock. Section 4.6.2 explains briefly how the VMI can break the deadlock.

4.6.1 Stack maintenance

Logic block *BC decode* uses the number of pops and pushes of a bytecode (as found in BCPT), the value of the *RSP* register and also the size of the register stack to determine if translating the current bytecode **would** cause a stack overflow or underflow. If this would be the case, half of the register stack is flushed or restored to or from memory **prior** to translating the bytecode. After that, the translation proceeds as usual. This way, the translation of a bytecode can safely complete without a stack exception. Other implementations of a JBC interpreter may choose a different solution for stack maintenance.

TT contains sequences for flushing or restoring half of the register stack. These sequences can be indexed by an entry from CIT, because the JVM specification lists less than 256 bytecodes and some bytecodes are garanteed by the JVM specification to remain implementation dependent.

From now on, the bytecode for stack overflow is 254 and the bytecode for stack underflow is 253. Byte codes from actual JBC never contain these values, they are unused. This addition is made solely for the purpose of controlling the VMI. If, for example, a stack overflow occurs, *BC decode* stores the value 254 in the *currBcode* register. The process of generating MIPS instructions from these two new special bytecodes remains the same. After the last MIPS instruction for flushing half of the register stack has been generated, the VMI must continue with the bytecode it originally wanted to translate. This time, this bytecode will not cause a stack overflow.

This subsection introduces a new logic block called *BC mux*. It is placed just before the *currBcode* register and just after the data logic block of *BC fetch*. *BC mux* monitors a signal called *mux_select*. This signal is controlled by *BC decode*. If there is no stack overflow or underflow, *mux_select* is inactive. In this case *BC mux* passes a bytecode from *BC fetch* to the *currBcode* register. The following paragraph will explain what happens if *mux_select* is active.

The *RSP* register is not drawn. If *BC decode* detects a stack overflow or underflow, two things happen. First, signal *mux_select* is set to a value that indicates whether there is a stack overflow or underflow. And second, the control state machine of *BC decode* goes to a state that is similar to its default state, except that *bcc* is not incremented when *last_instr* is active. The reason for this is VMI must **continue** with the bytecode that caused the stack overflow or underflow. Recall that in section '4.5.2 Updating *bcc*' a special control signal *bcc_inc* was introduced to let *BC decode* determine when to increment *bcc*. *BC mux* monitors *mux_select* and stores the value 254 in the *currBcode* register in case of a stack overflow, or 253 in case of a stack underflow. Once the register stack is flushed or restored, *mux_select* becomes inactive and the VMI can proceed.

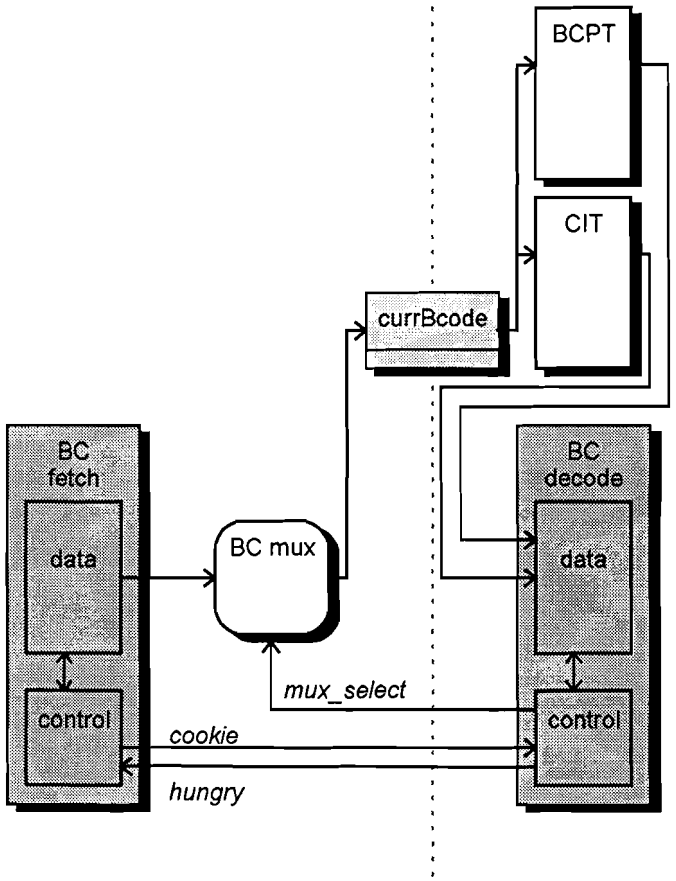

Figure 2: Adding BC mux *to the model*

4.6.2 Preventing deadlock

If the VMI is a bus peripheral, a deadlock could occur. Suppose the CPU requests an instruction from the VMI but at the same time the VMI needs to fetch a bytecode in order to be able to generate the requested instruction. The PI-bus is locked by the CPU, so the VMI cannot fetch a bytecode. Since no new instruction can be generated, the CPU will not release the bus. This is called **deadlock**.

The logic in the support module detects this situation and makes sure an instruction is sent to the CPU to break the deadlock. This is done by activating a new control signal which connects to *IG*. If pipe stage *IG* does not have a

generated instruction available, it must generate a **NOP**. Just as with other generated instructions, *IG* also determines the position in the FIFO of a new instruction.

4.7 INTERNALS OF LOGIC BLOCK *IG* EXPLAINED

With the additions described previously in this section most bytecodes can be handled. This subsection focuses on the internals of logic block *IG*, a crucial part in the process of translating JBC to MIPS instructions. It has been mentioned previously that the register fields and possibly the immediate field, of an instruction skeleton is encoded. There are basically three formats for MIPS instructions: I-type, J-type and R-type. The J-type is not used for the instruction skeletons. Instructions of the I-type format are composed of a 6-bit operation code, a 5-bit source register specifier, a 5-bit target register specifier and a 16-bit immediate, branch displacement or address displacement. Instructions of the R-type format are composed of a 6-bit operation code, a 5-bit source register specifier, a 5-bit target specifier, a 5-bit destination specifier, a 5-bit shift amount and a 6-bit function field. This table shows these two formats.

Table 7: Two instruction formats for 32-bit MIPS instructions.

bits	31-26	25-21	20-16	15-11	10-6	5-0
I-type	opcode	rs	rt	immediate		
R-type	opcode	rs	rt	rd	sa	funct

First, the logic block IG determines whether the instruction skeleton has format I-type or R-type by checking the operation code. In the latter case, *sa* and *funct* are passed unchanged. For *rs*, *rt* and *rd* the decoding works as follows: there are 32 different values a 5-bit register specifier can have. Some of those are reserved for specifying the CPU registers that hold pointers to the various memory areas of the JVM, like the constant pool. Others are reserved for specifying the CPU registers that the CPU itself uses to pass parameters to functions (usually registers 4 through 7) or store results of functions (usually registers 2 and 3). These numbers coincide with the conventional use of these registers. The register mapping is a part of the VMI specification.

Most values however, are used to specify entries in the register stack. Not just the register stack **top** but other entries as well. Specifying any register is done by specifying the offset from the first CPU register (called the *lower bound*) that is used either for the register stack, passing parameters or storing results. Apart from the lower bounds, the register mapping details the size of the register stack, the size of the parameter passing section (usually 4) and the size of the result storing section (usually 2). New registers (jointly called *regMap0*) that reside on the pipe boundary between *TT load* and *IG* pass the register mapping to *IG*. Other new registers (jointly called *regMap1*) on this boundary pass the numbers that indicate which CPU registers hold pointers to various memory areas of the JVM to *IG*.

In case the format of an instruction skeleton is I-type, decoding *rs* and *rt* is the same as for R-type. The 16-bit immediate value is decoded as follows: all the different values a 16-bit immediate can have, are split in mutually exclusive ranges. A target substitution type is associated with each range. For example, the first target substitution type is constants. If the immediate field has a value in

the first range, a constant is substituted for the immediate value. So, VMI can only issue a small set of immediate constants but it can also use other target substitution types, like the value in the *param* register. If the immediate field has a value in the second range, the contents of the *param* register are substituted for the immediate value.

4.8 FUTURE IMPROVEMENTS

With the current model, there is a delay of several cycles between sequences of generated MIPS instructions. In a true pipeline, pipe stage *BC fetch* would start fetching a new byte code as soon as the current bytecode was stored in the *currBcode* register. The VMI does not do this yet. The delay is present because *bcc* is only incremented after *last_instr* is set. It takes time to access the bytecode cache and, when there is a cache hit, to store the new bytecode in *currBcode* (and the new operand bytes in the *param* register). If there is a cache miss, the delay is even larger. Optimizing the pipeline has not been a high priority yet. It will be in the future.

The new bytecode and its operand bytes can be retrieved by the bytecode cache earlier, possibly hiding the cache miss latency, with further improvements to the control of the model. It is possible to load the *currBcode* register with the next bytecode **before** the translation of the current bytecode is finished. Suppose that *BC mux* was placed **after** the *currBcode* register and before *BC decode*, BCPT and CIT. With changes to the state machines involved, the next bytecode could be stored in *currBcode* but not yet passed on until the right time. *BC mux* would have to be extended to multiplex the contents for the *param* register as well. With these changes, the delay between two bytecodes could be one cycle, possibly none.

If the pipeline would function optimally, all data flowing across pipe stage boundaries would need to be temporarily stored in registers. Right now, the figures in this report resemble the structure of the VHDL-code which has no timing problems.

The issue of how many MIPS instructions the VMI needs to generate will be addressed in the following chapter.

5. SIMULATION

This chapter takes a closer look at three ways of simulating a hardware design: functionally, structurally and, using both, with co-simulation.

5.1 INTRODUCTION

At a certain point during any design, the time comes to go beyond the scope of paper and pencil. Ideas are put to test using simulations. Hardware design is no exception. The following sections will discuss three types of simulation for the VMI: functional simulation, structural simulation and co-simulation, which uses the first two.

5.2 SIMULATING THE FUNCTIONAL MODEL

The functional model of the VMI core was programmed in such a way that it can be used with a simulator called 'Tool for System Simulation'(TSS). The support module is an integrated part of the VMI model.

5.2.1 Tool for System Simulation

TSS is a C-simulation framework, which allows for a high level description of complex systems in the C programming language. Describing the system in C has the advantage of high speed simulation. TSS comes with a set of models that describe hardware modules like a MIPS R3000, UART, PI-bus controller, interrupt controller and memory. Modules are instantiated and connected in a netlist file. An instance contains a number of ports, a number of processes and a state. Processes are used to describe the actual functionality of an instance.

A TSS process is similar to a VHDL process in the sense that both are associated with a list of ports, called the *sensitivity list*, which triggers the execution of the process. There are of course differences between TSS processes and VHDL processes. One of the more subtle differences is that assigning the current value to a port again also triggers TSS processes. In VHDL, all assignments are called *transactions* and processes are not triggered by *transactions* but by *events*, which is when a port *changes* its value.

All the elements of the chosen architecture option as depicted in figure 2 are available as TSS models. The functional model of the VMI was tested and simulated together with models of a MIPS R3000, PI-bus controller, memory and other peripherals. Refer to [TSS] for details on TSS and its use. Refer to [MML] section 2.6 'Implementation Aspects of the Simulator' for more details on simulating VMI using TSS. The limits and restrictions on the programs that this simulator can execute are listed in [MML].

The only two results of the experiments using this simulation that need to be repeated here are as follows:

1. The performance of the VMI using the TSS simulator was limited by the traffic on the PI-bus. The speed with which this model of the VMI could generate instructions was adequate to continuously send MIPS instructions to the CPU as it requested them. It is not necessary to generate one MIPS instruction per cycle, because the overhead introduced by the PI-bus limits the number of instructions the VMI can send to the CPU per cycle. If any of

the other architecture options is chosen and implemented, a delay in the pipeline of the VMI core could affect the performance.

2. The simulations support the numbers used in column 3 of table 5 'Theoretical speed-up and dynamic distribution for benchmark programs'.

5.3 STAND ALONE SIMULATION

The environment used for simulating VHDL is called Leapfrog. The term 'stand alone' can be used in many ways. As soon as the initial model of the VMI core was described in VHDL, much like depicted in figure 16, tiny programs could be 'simulated' with Leapfrog. The model had no connections to a PI-bus or a CPU or to memory. Hence the name 'stand alone' is used for this level of simulation. *IG* output *bcc* was connected to a new register, the *bcc* register, which in turn indexed the bytecode cache. The cache was simply an array in which the program was loaded prior to simulation. The execution was considered successful if the correct MIPS instructions were put on *IG* output *i2FIFO*, one after another. These simulations are structural, because the VHDL code described an RTL-model of the VMI core.

Although this is not a high level simulation, it allowed the user to test and add features to the initial model. Sections 4.4.1 'Fetching parameters', 4.5.1 'Directing the inner switch', 4.5.2 'Updating bcc' and 4.7 'Internals of logic block *IG* explained' discuss features that were added to the initial model using stand alone simulation. It should come as no surprise to the reader that a feature like the one described in section 4.5.3 'Waiting for the CPU' could not be tested with stand alone simulation. The mechanism for stack maintenance, i.e. the use of *BC mux*, was also tested with this simulation.

The stand alone simulation is a useful platform for improving a model and perfecting it before it is tested in a more complex environment. However, the model is not really thoroughly tested until the environment it is supposed to work in is simulated as well.

5.4 CO-SIMULATION

Theoretically, it is possible to test the VHDL model together with VHDL models of a PI-bus, a MIPS CPU and memory. However, these models are not available in VHDL and such a simulation would be very slow. Luckily, TSS can co-simulate with Leapfrog, using the Leapfrog Foreign Model Interface (FMI). This is a way to make all TSS models available in Leapfrog. PRLE went to great lengths to structurally split the monolithic VMI unit into three parts: the VMI core, the register file and the rest of the support module. All the registers that are a part of the VMI, including the FIFO, are combined in the register file. The TSS model of the VMI core replaces the VHDL model of the VMI core as depicted in the following figure:

The reader can see the similarities between this figure and figure 6. The TSS channels are VHDL models and all parts in the TSS simulator are C models.

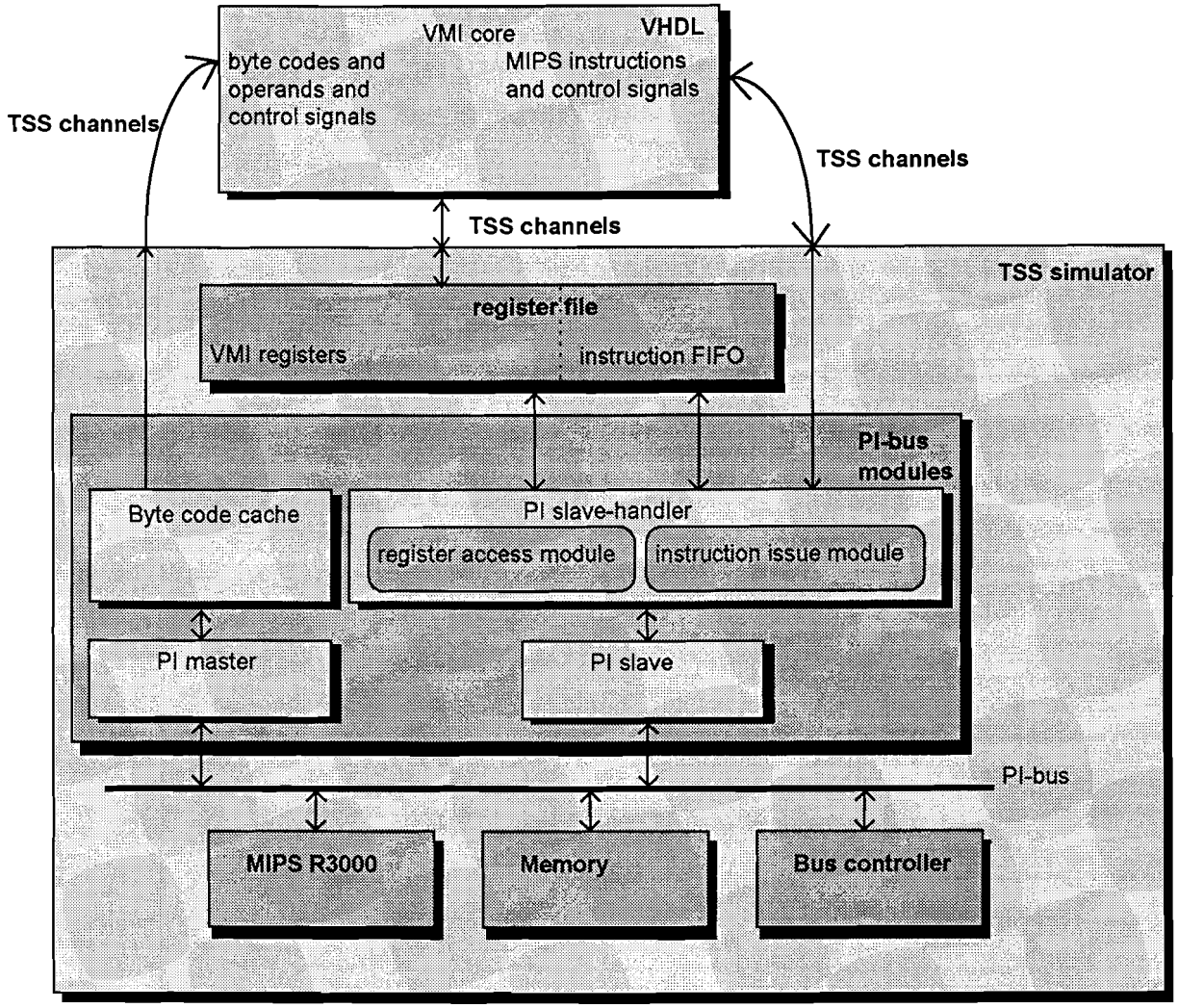

Figure 24: Co-simulation model. The VMI core, the TSS channels and TSS kernel are accessible through Leapfrog

The result of these simulations show that the structural model is not as fast as the functional model. The structural model takes more cycles to translate the same JBC instructions in MIPS instructions. However, with the improvements listed in section 4.8 'Future improvements' it is possible to generate instructions as fast or almost as fast as the functional model. Performance measurements of the functional model have shown that the VMI generates instructions faster than the CPU can 'consume' or execute them because of the protocol overhead introduced by the PI-bus. Therefore, the conclusions of chapter three are valid and applicable as long as the structural model can **also** generate instructions **faster** than the CPU can execute them, which will be the case once a few improvements to the model are made. If the VMI would be placed at another position in the system architecture, the speed of generating instructions could become an issue.

5.5 CONCLUSIONS

These conclusions are the result of the simulations.

- Testing a VHDL model that interacts with components like a MIPS CPU is not practical with stand alone simulations.
- TSS is a powerful tool for co-simulation, especially because of the models that come with the package.
- The structural VHDL model of the VMI core is an RTL description of the functional C model of the VMI core. It can correctly handle all the bytecodes in hardware that the functional model handles in hardware. This is an important step towards a hardware implementation.
- The structural model of the VMI core will, with a few improvements, generate instructions fast enough not to endanger the performance gain as estimated in chapter three.
- With several improvements, the structural model can generate instructions as fast as the functional model.

6. CONCLUSIONS AND RECOMMENDATIONS

The conclusions are:

- The VMI speeds up the execution of pure JBC by 4.0 times, compared to a software interpreter.
- The VMI speeds up a typical large application for the embedded market by 2.6 times, because not all execution time is spent interpreting pure JBC.
- The structural VHDL model of the VMI core is an RTL description of the functional C model of the VMI core. It can correctly handle all the bytecodes that the functional model handles. This is an important step towards a hardware implementation.
- The structural model of the VMI core will, with a few improvements, generate instructions fast enough not to endanger the performance gain as estimated in chapter three.

The recommendations are:

- It would be interesting to learn from additional research which part of the class libraries can be compiled prior to execution.
- The support module has to be ported to VHDL to allow synthesis.
- Once the entire VMI can be synthesized, the first objective of this project should be an FPGA implementation.
- A full set of class libraries and native libraries is needed to provide a Java compliant execution platform.

7. APPENDIX A: REFERENCES

MML

M.M. Lindwer
Hardware Acceleration for Virtual Machine Instructions, Illustrated for the Java Virtual Machine
Nat.Lab. Technical Note 173/97
Philips Electronics, 1997

JVM

Tim Lindholm, Frank Yellin
The Java Virtual Machine Specification
Addison-Wesley Longman Inc, 1997, Reading, Massachusetts, USA
ISBN 0-201-63452-X

JLS

James Gosling, Bill Joy, Guy Steele
The Java Language Specification
Addison-Wesley Longman Inc, 1996, Reading, Massachusetts, USA
ISBN 0-201-63451-1

OOP

NEXTSTEP
Object-Oriented Programming and the Objective C Language
Addison Wesley Publishing Company, 1993, Reading, Massachusetts, USA

VMI

M.M. Lindwer, Ir. O.L. Steinbusch
VMI Micro Architecture
Philips Research Laboratories Eindhoven, 1997, Eindhoven, The Netherlands
Memo: RWB-516-me-97054

H&P

John L. Hennessy, David A. Patterson
Computer Architecture: A Quantative Approach, Second Edition
Morgan Kaufmann Publishers Inc, 1995, San Francisco, California, USA

K&R

Brian W. Kernighan, Dennis M. Ritchie
The C programming language
2nd Edition
Prentice Hall Software Series, 1988, Englewood Cliffs, New Jersey, USA

W&S

Larry Wall, Randal L. Schwartz
Programming Perl
O'Reilly and Associates Inc, 1991, Sebastopol, California

ASH

Peter J. Ashenden
The designer's guide to VHDL
Morgan Kaufmann Publishers Inc, 1996, San Francisco, California, USA

K&H

Gerry Kane, Joe Heinrich
MIPS RISC Architecture
Prentice Hall PTR, 1992, Englewood Cliffs, New Jersey, USA

3910

MIPS PR3910 Microprocessor Risc Core, User Manual
Version 0.1
Philips Semiconductors, 1995

TSS

J.C.Koot
TSS Manual,
version 0.3,
PS-ASIC Service Group Eindhoven, 1996, The Netherlands

8. APPENDIX B: LIST OF ABBREVIATIONS

BIU	Bus Interface Unit
CPU	Central Processing Unit
FIFO	First In First Out, a type of queue
FIR	Fast Infra Red
FMI	(Leapfrog) Foreign Model Interface
FPGA	Field Programmable Gate Array
HCG	Handheld Computing Group
JBC	Java Byte Code
LIFO	Last In First Out, a type of queue
MIPS	MIPS is a RISC architecture
OO	Object Oriented
PRLE	Philips Research Laboratories Eindhoven
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
TSS	Tool for System Simulation
UART	
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VMI	Virtual Machine Interpreter

9. APPENDIX C: DISTRIBUTION

Author O.L. Steinbusch

Title Designing Hardware to Interpret Virtual Machine Instructions

Distribution

EUT	Eindhoven University of Technology
PS	Philips Semiconductors
PRLE	Philips Research Laboratories Eindhoven
PRLR	Philips Research Laboratories Redhill, UK
CP&T	PRLE-WAH
BE	Business Electronics
PCC	Philips Consumer Communications
LEP	Philips-Limeil-Brévannes, France

Full report

J.A.G Jess	EUT-EH	Eindhoven
A.C. Verschueren	EUT-EH	Eindhoven
M. Hershenson	PS-HCG	Mountainview, USA
J.A. de Oliveira	PS-ASG/ESTC	Sunnyvale, USA
M.M. Lindwer	PRLE	WL-1
J.H.A. Gelissen	PRLE	WL-p
W.J.H. Lippmann	PRLE	WL-1
A.J. Bink	PRLE	WL-1
J.-Y. Brunel	PRLE	WL-1
R. Bruzzone	PCC	Le Mans, France
M. Emons	PS-Nijmegen	FB-3
H. Frydlender	LEP	Limeil-Brévannes, France
P. Gorissen	PRLE	WL-1
L. de Haas	CP&T	WAH
J. Hoekstra	CP&T	WAH
R.J. Houldsworth	PRLR	Redhill, UK
C.M. Huizer	BE-DVS	SX-2
B. Kajiyama	PS-HCG	Mountainview, USA
A. Klap	PS	Nijmegen
P. Klapproth	PS-ASG/ESTC	WAY-4
F. Klok	PRLR	Redhill, UK
J.C. Koot	PS-ASG	WAY-p061
W. Kruytzer	PRLE	WL-1
P. Lippens	PRLE	WAY-4
S. van Loo	PRLE	WL-1
A.C.M. Oerlemans	PS-CIC	BAE-2
D.E. Penna	PRLR	Redhill, UK
J.R. Piesing	PRLR	Redhill, UK
J. Rapeli	PCC	Le Mans, France
F.W. Sijstermans	PRLE	WL-1
G.A. Slavenburg	PS-Trimedia	Sunnyvale, USA
P. Stravers	PS-ASG/ESTC	Sunnyvale, USA
S. Tickell	PRLR	Redhill, UK
J. Verhoeks	PS-Nijmegen	FM-1
M. Treffers	PRLE	WL-1
J.A. Trescher	PRLE	WL-1
M. Vlot	PS-C&M	BAE-2
H.W.J. van de Wiel	PRLE	WL-1
F. Zandveld	PRLE	WL-1