Department of Computing Science

UNIVERSITY
of
GLASGOW

Individual Project Report
Level 4, 2004/2005

Porting the Vector Pascal Compiler to the Playstation 2

by

Peter Cooper
0102734
cooperp@dcs.gla.ac.uk

Class (SE4H)
Session 2004/2005

Department of Computing Science,
University of Glasgow
Lilybank Gardens
Glasgow G12 8QQ.

Abstract

Vector Pascal is an extension to the Pascal programming language which adds support for operating across arrays. The Vector Pascal compiler can take a Vector Pascal program and compile it to the appropriate target system, taking advantage of the vector instructions in the processor in the target system to produce programs that perform faster than those compiled without support for the vector instructions.

The Playstation 2 is a games console whose main processor has been coupled with vector processors which can very quickly perform operations common in games. Sony has released a distribution of Linux for the Playstation 2 which turns their console into a cheap personal computer. The aim of this project is to port the Vector Pascal Compiler to Linux on the Playstation 2 to produce programs that can take advantage of the vector processors in the Playstation 2.

Contents

1 Introduction 6

1.1 The Playstation 2 6

1.2 Vector Pascal 7

1.3 The Vector Pascal Compiler 8

2 Background 9

2.1 Vector Pascal 9

2.2 Vector Pascal Compiler 11

2.2.1 Compiler Structure 12

2.2.2 Porting to the Playstation 2 13

2.3 The Playstation 2 System 13

2.4 The Playstation 2 Processor (The Emotion Engine) 14

2.4.1 MIPS Processors 14

2.4.2 MIPS Instruction formats 15

R Type Instructions 15

I Type Instructions 16

J Type Instructions 16

2.4.3 MIPS Registers 17

2.4.4 Emotion Engine Extensions 19

Multimedia registers 19

Multimedia instruction formats 19

2.4.5 System control coprocessor 20

2.4.6 Floating point coprocessor 20

Floating point format 21

Fixed point format 21

Floating point registers 21

Floating point control registers 22

Floating point instructions 23

2.5 The Vector Processing Units 24

2.5.1 Operating Modes 24

2.5.2 The Vector Units 25

Floating point registers 26

Execution Units 26

Upper execution unit 27

Lower execution unit 28

Differences between the Vector Units 29

Controlling the vector units....29
The VIF....29
2.6 MIPS Memory Allocation....30
2.6.1 Position Independent Code....32
Global Offset Table....33
3 Design....36
3.1 The Stack....36
3.1.1 Entering a procedure....38
The Display....38
3.1.2 Procedure Call Mechanisms....45
Passing Parameters to Pascal Procedures....46
Passing Parameters to C Procedures....48
3.2 Double precision floating point operations....50
3.3 Multimedia instructions....52
3.4 The ILCG Emotion Engine Specification....54
3.4.1 Declaring Types....54
3.4.2 Declaring Registers....54
3.4.3 Declaring instructions....55
3.4.4 The Instruction Set....55
4 Implementation....58
4.1 The ILCG Machine Specification file....58
4.1.1 Implementing floating point math operations....60
4.1.2 Implementing double precision instructions....61
4.1.3 Implementing the multimedia instructions....62
4.1.4 Optimizing the ILCG machine specification....62
4.1.5 Multimedia optimisations....62
4.2 The EECG Walker subclass....63
4.2.1 Parallelism....66
4.3 Executing the compiler....66
4.3.1 Running the assembler....67
4.3.2 Running the C compiler....67
4.4 Changes made to the Vector Pascal Compiler....68
4.5 Optimizing operations....68
4.6 Comparison Operators....71
4.6.1 Less than (<)....72
4.6.2 Greater than (>)....73
4.6.3 Equal to (=)....73

4.6.4 Not equal to (<>) 74
4.6.5 Less than or equal to (<=) 74
4.6.6 Greater than or equal to (>=) 75
4.7 Floating point comparison operators 75
4.7.1 Less than, equal to, or less than equal to 76
4.7.2 Greater than, not equal to, or greater than and equal to 77
4.8 Double precision floating point comparisons 77
4.8.1 Less than (<) 78
4.8.2 Equal to (=) 79
4.8.3 Greater than (>) 79
4.8.4 Less than or equal to (<=) 80
4.8.5 Greater than or equal to (>=) 81
4.8.6 Not equal to (<>) 81
5 Testing 82
6 Current status 84
7 Future Improvements 85
7.1 Compiling units to different platforms 85
7.1.1 Defining an interface between different processors 86
7.1.2 Interfacing the Emotion Engine and Vector Units 86
7.1.3 Interfacing other platforms 89
7.2 Compiling to other MIPS processors 89
8 Future Platforms 90
8.1 The Cell Processor 90
8.2 Dual Core Processors 91
8.3 Using Graphics Cards as Vector Processors 92
8.4 Physics coprocessors 93
9 Conclusion 94
Appendix A 95
Appendix B1 102
Appendix B2 103
Appendix B3 104
References 105
Thanks to… 106

1 Introduction

Every few years, the number of transistors on a processor doubles. Coupled with this increase in the number of transistors, the complexity of the processor also increases. One problem facing processor designers is what to do with all the extra transistors they have available to them. In recent years, they have been using some of these extra transistors to incorporate vector operations into modern micro-processors, such as Intel's Pentium series [1], and the AMD Athlon [2]. These instructions allow operations to be applied to many pieces of data in parallel. For example, the Pentium 4 can add together two groups of four single precision floating point numbers simultaneously. Programmers who take advantage of these operations can greatly increase performance in their code, however, using these new features has tended to involve writing assembly code since typical programming languages and compilers do not have specific support for these features. Most of these new instructions will only be used in programs that strive for the greatest performance level possible, for example, in scientific applications, and in computer games. The instructions added to the processors lend themselves well to these applications, since, for example, the instructions to operate on four floating point numbers at a time translate directly into operations that are common in 3d applications, and especially games. It is this substantial increase in performance that led Sony to include vector operations in their Playstation 2 games console.

1.1 The Playstation 2

The Playstation 2 was released in Japan on the 4th or Match 2000. At the time of writing, this means it has been released for just over 5 years. When it was first released it was arguably the most powerful games console on the market, with a theoretical performance level of 6.2 billion floating point operations per second (6.2 GFLOPS) [3]. At the time, this led to news reports that Iraq had imported 4000 Playstation 2's to use as computer systems [4], since at the time the UN had sanctions stopping Iraq from purchasing most personal computers, but not games consoles.

Although the Playstation 2 was primarily designed for running games, Sony has released a version on Linux on the platform. This behaves much like Linux on a PC, and includes all of the necessary tools to write and compile C programs. It is possible to download web browsers and other common applications for the Playstation 2 and essentially use the Playstation 2 as if it were a desktop PC. The Playstation 2 even comes with USB ports

allowing common PC peripherals such as printers and scanners to be connected to it, although drivers for these may not be available for Linux.

The Playstation 2 has four main processors: the core processor; a floating point coprocessor; and two vector processors (which will be called the Vector Units from now on). It will be possible to write a Vector Pascal program that uses just the core processor and the floating point coprocessor, however, considerable performance benefits should be possible if the Vector Units are also used for processing. To give an example of the performance increases possible, a program that was compiled using Intel's 486 instruction set executed at a level of 80 million operations per second [5]. The same program compiled to the Pentium instruction set, taking advantage on the MMX vector instructions in the Pentium, executed 820 million operations per second. With the possibility of executing instructions on the main Playstation 2 processor, and simultaneously on both the Vector Units, the potential performance increase over just using the main Playstation 2 processor is considerable.

There has already been some research into using the Playstation 2 for scientific purposes. The Computation Chemistry Group at the University of Illinois used a single Playstation 2, operating on vectors using Vector Unit 0 to achieve a performance of 150 MFLOPS [6].The Computer Science Department, also at the University of Illinois and The National Center for Supercomputing Applications have used a network of 70 Playstation 2's all running Linux to construct a supercomputer [7]. This same team also researched methods for producing high performance matrix multiplying code on the Vector Units, and using just Vector Unit 1, they achieved 1 GFLOP of performance. This figure is still significantly lower than the 6.5 GFLOPS of theoretical performance available, but taking into account that they are doing all of this using only one vector unit, on a system that is also running Linux at the same time, the performance achieved is considerable.

1.2 Vector Pascal

Vector Pascal is an extension to the Pascal programming language that adds support for operating on arrays. It was designed by Dr Paul Cockshott, who continues to develop the language, and the associated Vector Pascal Compiler. It exploits the vector instructions in the platform it is operating on, while hiding the complexity of having to write assembly code from the programmer. There are other compilers that can optimize code to use vector instructions, Intel's C++ compiler [8], for example, but this has the disadvantage of only optimizing for Intel's platform, while the Vector Pascal compiler can be ported to new systems.

1.3 The Vector Pascal Compiler

The Vector Pascal Compiler includes all the code necessary to compile Vector Pascal programs to a number of different platforms, including the Pentium, Opteron, and PowerPC processors. Machine specifications are written in a language called ILCG (Intermediate Language for Code Generation) [9] and are passed through a code-generator-generator to create a Java class of the code-generator for the required target platform. The Vector Pascal Compiler included a partial machine specification for the Playstation 2, with many of the registers, instructions, and procedure call mechanisms already defined. This project aims to build on this partial specification to produce a more complete port of the Vector Pascal compiler, with the hope of getting basic programs executing in Linux on the Playstation 2, and possibly programs running using vector operations.

2 Background

2.1 Vector Pascal

Vector Pascal is an extension of the popular Pascal programming language that was introduced to allow programmers to make use of data parallel operations. These operate on arrays, so that where a programmer would have historically had to manually insert a loop into their code to iterate over the arrays in question, the language now allows the operation to be applied to the array in the same way as it would have been applied to the individual elements in the array. This is very intuitive because the language allows any operator that could have been applied to the elements of the array, to be applied to the whole array, for example, the Vector Pascal program given below adds each element in array v2 to that of the corresponding element of v3 and stores the resultant integer in that position of array v1.

```
PROGRAM vecadd;
VAR v1, v2, v3 : ARRAY[0..127] OF integer;
BEGIN
        v1 := v2 + v3;
END.
```

Figure 2-1

Compare this with the equivalent program written in C and it becomes easy to see advantages Vector Pascal has over other languages that do not support vector operations.

```
int main()
{
        int v1[128], v2[128], v3[128], i;
        for (i = 0; i < 128; i++)
                v1[i] = v2[i] + v3[i];
}
```

Figure 2-2

It should be noted that in C it is possible for the programmer to overload operators so that the '+' operator could then be used across arrays in C as it is in Vector Pascal. However, this would not benefit from any parallelism available on the target processor to speed up program execution and unless they are using a C compiler that can optimise with vector operations on their desired target platform, then it would be the job of the programmer to write the necessary assembly language code to make use of any parallelism the target platform has to offer. This would restrict the program (or at least the data parallel optimisations) to only

work on those platforms where the programmer has defined the suitable assembly code for vector operations, whereas Vector Pascal programs are generally platform independent.

These vector operations are not restricted to one dimensional arrays, but can be applied to matrices of two dimensions, or arrays or any suitable dimension. There are restrictions in place to prevent, for example, the assignment of an array of n dimensions to an array of m dimensions, where m < n. However, the reverse of this is possible, so program given in Figure 2-3 would be valid.

```
PROGRAM vecassign;
VAR
        dim1 : ARRAY[0..2] OF integer;
        dim2 : ARRAY[0..2,0..2] OF integer;
BEGIN
        dim2 := dim1;
END.
```

Figure 2-3

With this ability to work on arrays of single or multiple dimensions, the language lends itself well to working on classic mathematical vectors and matrices. For example, the dot product of two vectors is a scalar formed by taking the total of multiplying each element of the first vector by the corresponding element of the second vector. Figure 2-4 gives the Vector Pascal program to compute the dot product of the vectors v and w and stores the result in the scalar s.

```
PROGRAM vecdot;
VAR
        v, w : ARRAY[0..2] OF integer;
        s : integer;
BEGIN
        s := v.w;
END.
```

Figure 2-4

In this example, the dot operator returned the dot product of the two vectors. It actually represents the vector and matrix product operator, and as such can be used to multiply vectors by vectors, vectors to be multiplied by matrices, or matrices to be multiplied by matrices.

When working with arrays, there is sometimes the need to find the total of the array, or perhaps the product of all the entries in the array, or even the largest or smallest elements. Vector Pascal also provides the reduction operator to help developers do these tasks. The

reduction operator will work with most of the common operators in the Pascal language, for example, +, -, *, and also with the new MIN and MAX operators introduced by the Vector Pascal language, which will be discussed shortly. It has been given the name reduction operator, because it has the effect of reducing the rank – or number of dimensions – of an array by one. The operator is used by first writing a '\' which is the sign of the reduction operator, then the dyadic operator required, for example, +, -, *, and finally the array to perform the reduction on. If we take an array C containing the numbers (1, 2, 3, 4) then the total of this array can be found by writing \+C. This would generate a sequence of instructions by inserting the + operator between each element of C, i.e., (1 + 2 + 3 + 4), and would return the correct total of 10.

As previously mentioned, Vector Pascal extends the Pascal language with two new basic operators. The MIN and MAX operators behave exactly as one would think, and are simply shorthand to save the programmer the time of writing a conditional statement to make use of the greater or lesser of two variables. They are important because the specification of the Playstation 2 processor must include ways to execute all of the basic operators in Vector Pascal, and since MIN and MAX are now basic operators, they must be included in this specification.

Vector Pascal provides many more extensions than those given here, including support for input and output or arrays, polymorphic functions, and many other features. A complete outline of the language is given in the SIMD Programming Manual for Linux and Windows [5].

2.2 Vector Pascal Compiler

The Vector Pascal Compiler has been written in Java, and is used to compile programs written in the Vector Pascal programming language. In order to compile programs to machine code, the compiler relies on a number of tools. Some of these are part of the Vector Pascal Compiler, while others must be provided externally by the operating system environment the compiler is running on. The Vector Pascal Compiler provides the following:

1. The ilcg.pascal Java package which contains the compiler itself, as well as classes to support Pascal type generations. These translate a Vector Pascal program in an ILCG tree.
2. A code generator for each target machine the compiler should support. This translates the ILCG tree into assembly language for the target architecture. The compiler includes a number of these such as the Pentium and Opteron code generators. A new code generator will be required for the Playstation 2.

3. The ilcg.tree Java package which contains all the classes necessary to represent an ILCG tree.

Using the above tools provided by the Vector Pascal Compiler, it is possible to translate from a Vector Pascal program, down to the assembly language for the appropriate target architecture. However, further processing is required to translate from the assembly file to the machine code for the target machine. The external tools provided by the operating system must be able to satisfy this need. The external tools required are:

1. The Java run-time system. This is needed to run the Vector Pascal Compiler, not to build the assembly files into machine code.
2. An assembler for the target architecture, which translates from the assembly code output from the Vector Pascal Compiler, into object files on the target system, for example, the NASM assembler for x86 processors.
3. A C compiler and linker, which can compile the C run-time library and link it to the object file output from the assembler, thus creating the final executable file, for example, the GNU C compiler.

The above tools would be sufficient for running the Vector Pascal Compiler. However, in order to make changes to the compiler, or add a new code generator, additional tools are required. These are:

1. The Java SDK, for compiling any changes made to the classes in the ilcg.pacsal or ilcg.tree packages.
2. The m4 macro processor. Using this tool, it is possible to write the machine specification file with loops to, for example, define registers. These loops will be automatically expanded by the m4 macro processor, thus saving time when writing the machine specification.
3. The sable compiler generator, which is used to generate a Parser for machine specification files written in ILCG source code.
4. The ilcg Java Package which contains the classes necessary to call the above Parser, and generate an ILCG code-generator.

2.2.1 Compiler Structure

There are many different parts to the compiler, and when run, it will take the source file though a number of stages before finally outputting an assembly file which should correspond to the Pascal program that was input to the compiler. This assembly code will use vector

operations wherever they have been defined by the machine specification and suitable code for vectorisation has been included in the source program.

2.2.2 Porting to the Playstation 2

In order to port the compiler to a new platform, in this case the Playstation 2, two main files must be written: a file called EE.ilc which contains the description of the Playstation 2's Emotion Engine processor in ILCG; and EECG.java which is a subclass of the code generator for the Playstation 2. EECG.java will include, among other things, implementations of the abstract methods to handle procedure call and return; methods to assemble and link the object files, which will call the necessary external assembler and linker; and a method called getParallelism which defines the level of parallelism we should attempt to use for each type of vector. A more in depth look at the contents of the EE.ilc and EECG.java files will be given later in the Implementation section.

2.3 The Playstation 2 System

The Playstation 2 system is designed around a single main processor. There are a number of other processors including: a DMA controller which transfers data from memory to the different parts of the system; a graphics synthesizer (GS) which displays images on the television screen or monitor; an input/output processor (IOP) which has 2 megabytes of its own memory; a sound processor; and processors for compressing and decompressing data. The Playstation 2 also has a DVD drive and 32 megabytes of main memory, which are linked to the main core with a 128-bit data bus. Although 32 megabytes is sufficient for Playstation 2 games since they do not have to share the memory with an operating system, programs executed in Linux on the Playstation will have much less physical memory available to them. They will have virtual memory available because Linux runs from a hard disc drive, but using this as memory will severely degrade program performance. Figure 2.5 is adapted from one presented at SIGRAPH in February 2000 [3] and shows these components and the data paths linking them.

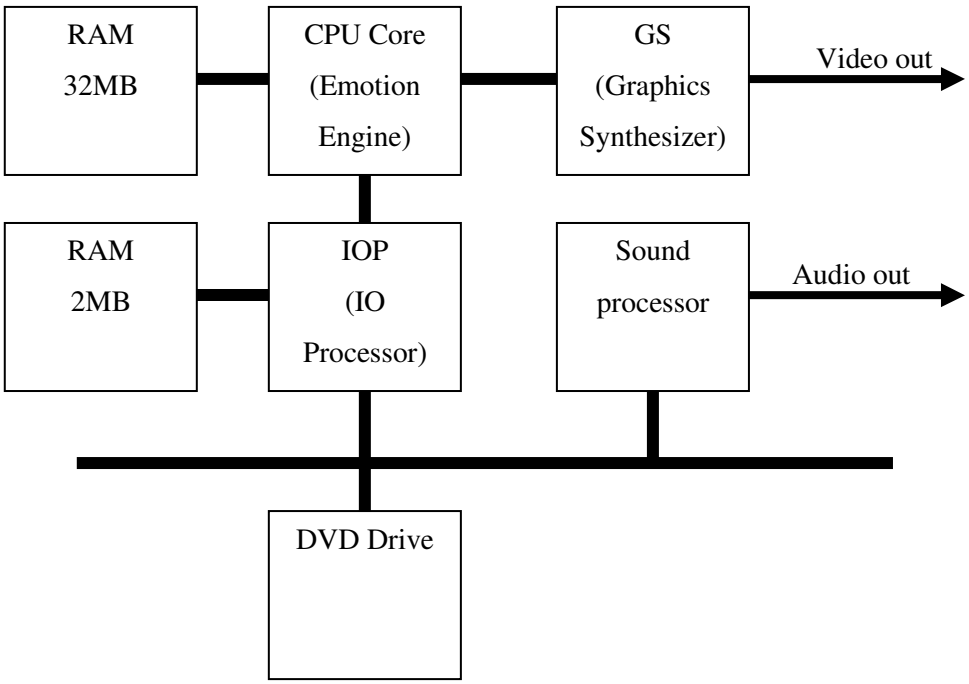


Figure 2-5

The most interesting point to note from Figure 2.5 is the link between the CPU Core and the graphics synthesizer. This will be discussed further when the vector units are covered.

2.4 The Playstation 2 Processor (The Emotion Engine)

The Playstation 2 contains not only the main processor, but also an input/output processor, a graphics processor, and numerous others. The focus of this project is on the main processor, which actually consists of 4 processors: the MIPS core; the floating point unit; and two vector processors. The most important one of these is the MIPS processor called the R5900, which implements most of the 64-bit MIPS III instructions, and some of the MIPS IV instructions. It also includes multimedia instructions which operate on 128-bit pieces of data as either: 2 64-bit values; 4 32-bit values; 8 16-bit values; or 16 8-bit values.

2.4.1 MIPS Processors

The MIPS architecture [10] was created in 1981 by John Hennessy. At the time, many processors were using CISC designs which were designed to implement as many complex operations in hardware as possible. This led to these processors being very expensive, and although they were very fast at executing the complex tasks they were designed for, the speed of the basic instructions was reduced by the added complexity from adding so many other instructions to the processor. The MIPS design aimed to use a deep instruction pipeline which is where it tries to have many parts of the processors working on the different parts of the execution of an instruction simultaneously. For example, while one instruction is being loaded from memory, another is being decoded, while another is having its operands fetched

from memory, while another is executing. The major setback with this design was that the processor would be limited to instructions that can execute in one cycle, since those that take longer would stall the pipeline, which makes the design far more complex. However, this was offset by the speed of the instructions that were included and the performance increase from using a deep pipeline.

The first MIPS design included 32 32-bit registers, and these registers still exist in the MIPS design to this day. It had a basic instruction set, including instructions to load and store values to and from registers and memory, and arithmetic and comparison instructions. The design also allowed for a number of coprocessors to be attached to the MIPS processor. Since the original design did not include floating point instructions, it was possible to add this functionality by using a floating point coprocessor. Instructions for the coprocessor can be sent directly to the main processor, and it will route them to the coprocessor. The Playstation 2 has 2 coprocessors in the form of a floating point unit; and one of the vector units.

2.4.2 MIPS Instruction formats

Instructions in MIPS are 32-bits wide. The designers of the MIPS core decided that all instructions should be the same length, and that instructions should generally fit into one of a number of groups. They also decided that, unlike the x86 instruction set of Intel's processor, that operations on data would require all data to reside in registers, i.e., it is not possible on MIPS to have one operand in a register and the other in memory as is permitted on x86 processors. These decisions were taken to simplify the design of the processor, and to simplify the instruction set since only one of each arithmetic instruction need be declared – that which operates on data in 2 registers – instead of that instruction, and a similar one for when one piece of data is in a register and the other is in memory. There are 3 main types of instruction format in the basic MIPS design: the R, I and J types. The use of only three main types of instructions simplifies the instruction decoder in the MIPS processor, and allows the compiler to synthesize more complex instructions from these three basic types.

R Type Instructions

opcode	rs	rt	rd	shamt	funct
31……26	25….21	20….16	15….11	10……..6	5…..1

Figure 2-6

Figure 2-6 shows the different fields that make up an R type instruction, and the bit positions that each field uses. R type instructions operate with three registers: two source registers and a destination register.

The purpose of the fields is as follows:

- opcode tells the processor what instruction to execute,
- rs and rt are the two source registers,
- rd is the destination register,
- shamt tells the processor how far to shift values in shift instructions,
- funct is an additional register to determine what instruction to execute.

It can be seen from the size of the fields that all the register fields are 5 bits long. This allows 2^5 (= 32) registers to be addressed by each field. This is sufficient since there are only 32 registers which need to be addressed. The opcode field is 6 bits long which allows 64 possible instructions to be used. However, this is a very small number, but by using the funct field as well, up to 2048 possible instructions could be used, far more than is needed.

I Type Instructions

opcode	rs	rt	immediate
31……26	25….21	20….16	15….0

Figure 2-7

The noticeable differences between the I type and R type instructions are the lack of a third register, and the inclusion of a 16-it immediate value. These instructions are used to perform arithmetic on a source register and a 16-bit value. An example of this is the need to increment a number by one which, using only the R type addition instruction, would involve loading the 1 into a register and using that register in the addition. This can be shortened to just one instruction where the 1 can be passed in the immediate field. Not only does this reduce the number of instructions to execute, but it also saves the program having to use registers to store small values that could instead be passed in immediate instructions. The only shortfall of this instruction is the limit of only 16-bit immediate values. The MIPS designers decided against including a 32-bit immediate instruction since that would need an instruction of greater than 32-bits in length. They decided that it was better to keep all instructions at 32-bits in length, since that simplified the processor design, and 16-bit immediate values are usually large enough for most cases in programs.

J Type Instructions

opcode	address
31……26	25….0

Figure 2-8

J Type instructions are used to perform jumps in code, for example, jumping to a procedure, or jumping to the else branch of a conditional statement. The 26-bit address field allows 67 million different memory locations to be address. Given that memory locations have a size of 1 byte, this leads to a possible 67 megabytes of memory to be addressed. Since the Playstation 2 only has 32 megabytes of memory, it is therefore possible jump to any location in the physical memory with a J type instruction. There are three jump type instructions provided in the Emotion Engine: jump (J); jump and link (JAL); and jump and link from a register (JALR). The first of these simply causes the program counter to take the value of the address in the jump instruction. The second also changes the program counter, but before this, it saves the value of the program counter plus 4 to the return address register. This allows the called code to return to the instruction following the jump, and is used to call procedures. The final one of these performs the same function as jump and link, but instead of jumping to a memory address given in the address field, it jumps to the address contained in the register it is passed.

2.4.3 MIPS Registers

As previously seen, the MIPS core has 32 general purpose registers. These are given in Figure 2-9.

Register Name	Number	Symbol(s)
Zero register (always contains 0)	0	$zero
Assembler temporary (used only by assembler)	1	$at
Return registers	2-3	$v0-v1
Argument registers	4-7	$a0-a3
Temporary registers	8-15	$t0-t7
Saved temporary registers	16-23	$s0-s7
Temporary registers	24-25	$t8-t9
OS Kernel registers	26-27	$k0-k1
Global Pointer	28	$gp
Stack Pointer	29	$sp
Frame Pointer	30	$fp
Return Address register	31	$ra

Figure 2-9

All of the registers in Figure 2-9 were 32-bits wide on the original MIPS processors. They are 128-bits wide in the Emotion Engine, although the upper 64-bits are only used with the Multimedia (or Parallel) instructions, and not in normal program usage.

All of the registers have been designated a specific purpose, unlike the 8 general purpose registers on the Intel Pentium processors which can be used for any purpose, although normal practice dictates specific uses of some of those 8 registers. The zero register will always contain the value zero, and so any attempts to write a value to it will be ignored. It is useful because in calculations there is often the need for a zero value, so instead of the programmer having to specifically load a zero into a register, it is now always there if needed. The assembler temporary register is reserved for the assembler to use. This is necessary because there are pseudo instructions in MIPS assembly code that are broken down into a number of instructions by the assembler, and the assembler must have a register available to use during these instructions, or else it would have to use a register that the programmers code may already be using. The return registers are used to store the result from function calls. There are two of them because when they were originally 32-bits wide, two registers would be needed to return 64-bit values. The argument registers are used to pass up to four arguments to procedures, and if there are more than 4 arguments then it is up to the programmer to put the extra arguments onto the stack. This will be covered in more detail shortly.

The registers which are used in general calculations to store results and move values to and from memory are generally the saved and unsaved temporaries. They are called the saved and unsaved temporaries because this puts the responsibility for saving their values onto the called procedure and calling procedure respectively. What this means is that a procedure is free to change the values of the unsaved temporaries, however, if it calls a procedure, then the procedure it calls also has the same freedom. The result of this is that the calling procedure is responsible for saving the contents of any unsaved temporaries it will use before it calls another procedure, and restoring the values of these registers after the return from the called procedure. This can be contrasted with the saved temporaries where it is the called procedure, not the calling procedure that must save the registers it uses. This means that if a procedure needs to use any of the saved temporaries, it must save their values first, and restore them before it exits. Nevertheless, if it does use a saved temporary, it does not have to save and restore it before and after it calls a procedure, since if the procedure it calls made changes to that register it would have had to restore its value before exiting. This convention reduces registers spilling since a compiler can put values it needs after a procedure call in saved temporaries, as they are guaranteed to still be in those registers after the procedure returns, but can put values it does not need after the procedure call in the unsaved temporary registers.

Of the remaining registers, the OS Kernel registers serve a similar purpose to the assembler register and cannot be used by normal programs. The 9th saved temporary, register number

25, and the global pointer, have a special purpose when doing procedure calls, and accessing labels in assembly code, which will be covered later. The last register is used to store the return address for a procedure to exit to, while the stack and frame pointers are both used for working with the stack, which will be covered later.

In addition to these 32 registers, there is also the program counter (PC) which points to holds the address of the currently executing instruction and the HI and LO registers. The HI and LO registers store the upper and lower 64-bits of the results from multiplication and division instructions respectively. They have also been extended to 128-bits in size in the Emotion Engine, although, similarly to the general purpose registers, the upper 64-bits of each are only used by the multimedia instructions.

2.4.4 Emotion Engine Extensions

In addition to implementing most of the MIPS III registers and some MIPS IV registers, the R5900 processor includes a number of instructions to support multimedia applications. These instructions are similar to the MMX instructions on the Pentium processor. Like the Pentiums MMX instructions, when they are being executed, no other instructions can be executed by the processor. This is different from executing instructions in the Vector Units where it is possible for instructions to be executing in the Vector Unit and the main processor at the same time. Similar to MMX, these multimedia instructions have been designed to operate on a number of pieces of data at a time. This is an example of SIMD (Single Instruction Multiple Data) parallelism whereby many pieces of data are operated on at the same time by a single operator. It is these multimedia instructions that will provide the parallelism on scalar types in the Playstation 2 port of the Vector Pascal Compiler.

Multimedia registers

As mentioned in the section on MIPS registers, the Playstation 2 has 128-bit wide general purpose registers. The multimedia instructions use the general purpose registers, and it is only these instructions that use the whole 128-bits of the MIPS registers. In addition to providing instructions to do arithmetic using these registers, the multimedia extensions also include new instructions to load and store 128-bit values from memory for use in these extended general purpose registers.

Multimedia instruction formats

opcode = MMI					funct
31……26	25….21	20….16	15….11	10……..6	5…..1

Figure 2-10

The multimedia instruction format in Figure 2-10 is very similar to the R type format from the original MIPS design. The difference to the R type format is that the opcode is fixed to MMI, and it is the funct field that determines what instruction has to be executed. However, the funct field is only 5 bits wide and there are far more multimedia instructions than a 5 bit field can address. The solution to this is that there are 4 more classes of multimedia instructions within the MMI class given above. These are MMI0, MMI1, MMI2, and MMI3. For each multimedia instruction, one of these classes will occupy the funct field. Bits 6 to 10 are then used to represent the opcode of the instruction. The remaining fields can then be spit up into 1, 2, or 3 registers, depending on how many source registers an instruction requires. Note that if only one register field is used, then it must be the case that the source and destination registers are the same, or the destination register is being passed data from within the processors other registers, for example, the HI register. They may also include a shift amount field, which works in the same way as the same field in the R type format.

2.4.5 System control coprocessor

The system control coprocessor is attached to the main core as coprocessor 0. The main functions of this coprocessor are: cache control; memory control; exception handling; and providing breakpoint operations to the MIPS core. Instructions sent to the system control coprocessor will only be executed if the Emotion Engine core is in kernel mode, i.e., that the operating system kernel is accessing it, or bit 28 of the coprocessor status register is a 1, which indicates that coprocessor 0 is usable. It is not necessary for Vector Pascal programs to access the features provided by this coprocessor so its instruction set will currently not be included in the port of Vector Pascal to the Playstation 2.

2.4.6 Floating point coprocessor

The floating point unit (FPU) in the Playstation 2 is attached to the MIPS core as coprocessor 1. Floating point instructions can be written in the assembly code for the MIPS core, and if the MIPS receives these instructions it will simply pass them to the floating point unit for execution. Calls to the floating point unit cannot be executed in parallel with the MIPS core, so in one sense, the floating point coprocessor can really be seen as similar to FPUs in other processors, like the Pentium, and it is not really necessary for the programmer to know that the FPU is coprocessor 1. The only case where they must know this is when moving data from the MIPS core to the FPU since in this case, the instructions to move data are defined for each coprocessor.

Floating point format

The FPU only supports 32-bit floating point numbers. The format of the numbers supports the IEEE754 standard which means that it has 23-bit fraction field, an 8-bit exponent field, and a single sign bit.

sign	exponent	fraction
31..31	30…23	22….0

Figure 2-11

Although the FPU in the Playstation 2 uses the number format given in the IEEE754 standard, it does not comply entirely with this standard. For example, it does not support the Nan (Not a Number) or plus and minus infinity values defined in that standard.

Fixed point format

The FPU also supports the use of fixed point floating point numbers. These are just scalar numbers which are represented in the standard two's complement format. There are no arithmetic instructions in the FPU for fixed point numbers – these are in the MIPS core- but support has been included for these to allow for the CVT.S.W and CVT.W.S instructions which the FPU can use to convert to and from floating point to scalar numbers.

Floating point registers

There are 32 32-bit general purpose floating point registers in the FPU. They can each store a single floating point number. Unlike the MIPS registers, none of these registers have been reserved for special purposes like the assembler register, kernel registers, or stack and frame pointers. However, similarly to the MIPS core, there are floating point argument and return registers, as well as saved and unsaved temporary registers. The registers and their purposes are given in Figure 2-12 below.

Register Name	Number	Symbol(s)
Result registers	0 - 3	$f0 - $f3
Temporary registers	4 - 11	$f4 - $f11
Return registers	12 - 15	$f12 - $f15
Temporary registers	16 - 19	$f16 - $f19
Saved Temporary registers	20 - 31	$f20 - $f31

Figure 2-12

On 32-bit MIPS platforms, floating point registers are used in even/odd pairs. When storing a single precision floating point value in a register, the value will be placed in the even register, and the odd register will be left unused. If the hardware supports double precision numbers –

which the Emotion Engine does not – then the odd register would be used for the lower half of the double precision number, and the even register for the upper half. Since double precision numbers are not supported in the Emotion Engine, the odd registers are never used, and the compiler on the Playstation 2 will actually give an error if any attempts are made to use these registers. The temporary registers behave in the same way as the MIPS temporary registers, with respect to whether the calling or called procedure are responsible for saving the registers value. Usage of the result and argument registers will be covered later in the section on procedure call mechanisms.

In addition to the 32 general purpose registers, the FPU also has a single accumulator. This is a special register, in that it only has a latency of one cycle, while other floating point instructions have a latency of four cycles [11]. The latency of an instruction is the number of cycles the processor must wait before using the result of an instruction. This means that a sequence of say two floating point additions where the second addition needs the result of the first will cause the floating point unit to wait for four cycles after the first instruction before executing the second addition. The same sequence could be encoded using the accumulator instead, and due to its low latency, the floating point unit would not have to wait after the first instruction before executing the second [12]. It may be possible to include the accumulator in the Playstation 2 machine specification and so long as the Vector Pascal Compilers optimiser can keep the data in the accumulator after each operation, and not save it back to memory, as it would without any optimisations, significant speedups in the code execution could be possible.

Floating point control registers

The FPU can support up to 32 control registers, although only the first and last have been implemented on the Playstation 2. All of the control registers are 32-bits wide. The first control register is called the Implementation and Revision Register. This holds the implementation and revision numbers which can be queried to determine the capabilities of the FPU, for example, there will be a bit to tell the programmer whether double precision floating point numbers are supported or not. These are typically only used in diagnostic software and will not be used in this project.

The last control register is called the Control/Status register. It contains a number of status bits which can be used to determine the outcome of the last executed instruction. These bits are separated into the cause flags and the sticky flags. The cause flags are:

Flag	Name	Purpose
I	Invalid operation	Set to 1 if a 0/0 division, or square root of a negative number is attempted
D	Division by 0	Set to 1 if division by 0 is attempted
O	Overflow	Set to 1 if the exponent of the result of the previous instruction overflows
U	Underflow	Set to 1 if the exponent of the result of the previous instruction underflows

Figure 2-13

All of the above flags are set to 0 if their conditions do not hold. It is also possible for more than one of these to become 1 at the same time, for example, dividing 0 by 0 would lead to both the I and D flags being set to 1. The sticky flags are just copies of the cause flags, however, once they are se to 1, the processor will not clear them, and it is up to the programmer to specifically clear them if they wish to do so. In the FPU on the Playstation 2, the sticky flags are not used.

The final status bit which can change during program execution is the C bit. This occupies bit 23 in the control register and is set to 1 if the previous instruction was a comparison and the comparison is true or otherwise to 0. As will be seen later, it is possible to use this bit to get the result of floating point comparisons, as these results are not stored in normal floating point registers, but can be found according to this C bit.

There are a number of fields in this control register which are not used, or are set to always be 0 or 1. One notable case is the pair in bit positions 0 and 1 which define the rounding mode for the FPU. Together, these bits can return the values 0, 1, 2, or 3, which tell the FPU to round to either the nearest scalar, zero, plus infinity, or minus infinity respectively. The FPU in the Playstation 2 only supports rounding to zero so there bits are set to represent a 1 at all times. This inability to round to anything other than zero is another deviation from the IEEE754 standard.

Floating point instructions

The floating point instructions in the FPU can be grouped together into a number of types:

- Move instructions, which move data between floating point general purpose and control registers, and either the main MIPS core, or memory.

- Conversion instructions, mentioned earlier in the fixed point format section, which convert to and from integer and floating point representations of numbers.
- Arithmetic instructions which perform operations such as addition, multiplication, division, and rounding on floating point numbers.
- Comparison instructions which compare the value of two floating point registers and set the C bit in the control register if the condition is true.
- Branch instructions which can cause the processor to jump depending on the value in the C bit of the control register.

2.5 The Vector Processing Units

The vector processing units are self contained processors which include: a vector unit (VU); a small amount of data memory; and an interface to the DMA controller (VIF). The Playstation 2 has two Vector Processing units: VPU0 and VPU1, which are shown in Figure 2-14.

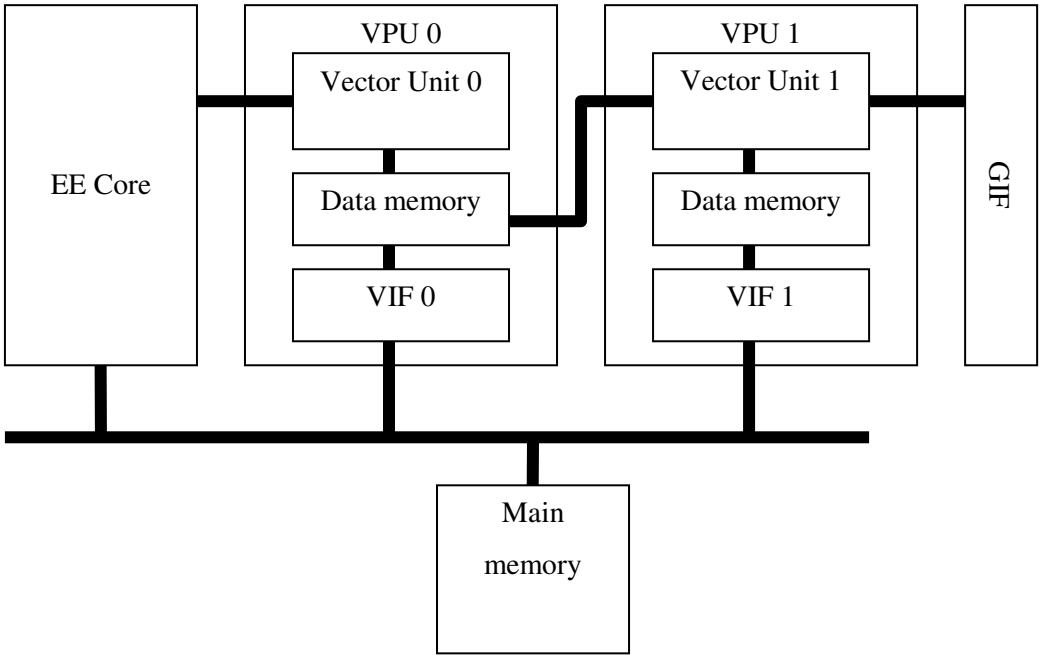


Figure 2-14

2.5.1 Operating Modes

Figure 2-14 shows a number of links between the core processor, memory, and the VPU's. Similar to the floating point unit, it is possible for vector unit 0 to operate as a coprocessor to the main core. When it operates in this mode, the main core will not be able to execute any of its own instructions while instructions it has issued to vector unit 0 are being executed. This operating mode is known as macro mode, and it is only possible for vector unit 0 and not

vector unit 1 to operate in this mode, as there is no connection from the core to vector unit 1. Since the VPU's have their own memory, a simple processor in the Vector Unit, and a VIF to transfer data to and from the VPU's memory and the main memory, it is possible for the VPU to run independently of the main core. This is known as micro mode because the memory inside the vector unit containing the instructions to execute is called the micro memory. When a vector unit is operating in micro mode, it is possible for the main core to continue working simultaneously. Vector Unit 1 always operates in micro mode, whereas vector unit 0 can operate in either micro mode or macro mode, but only in one mode at a time.

There is a link between the data memory of VPU0 and vector unit 1. This link is to show that the registers contained in vector unit 1 are memory mapped to the data memory of vector processing unit 0. This means that while VPU1 is not executing, data can be written to this area of memory, and once VPU1 starts execution, the data will be loaded from memory and placed in the corresponding vector unit 1 registers.

The GIF is an interface to the graphics synthesizer, which is a processor in the Playstation 2 which draws to the screen. It is possible for Vector Unit 1 to use special commands to send data it has processed to the GIF, which will then process it and put it on the screen. This feature is important in games which have complete control over the Playstation 2, but is not useful in simple command line programs compiled by the Vector Pascal Compiler.

2.5.2 The Vector Units

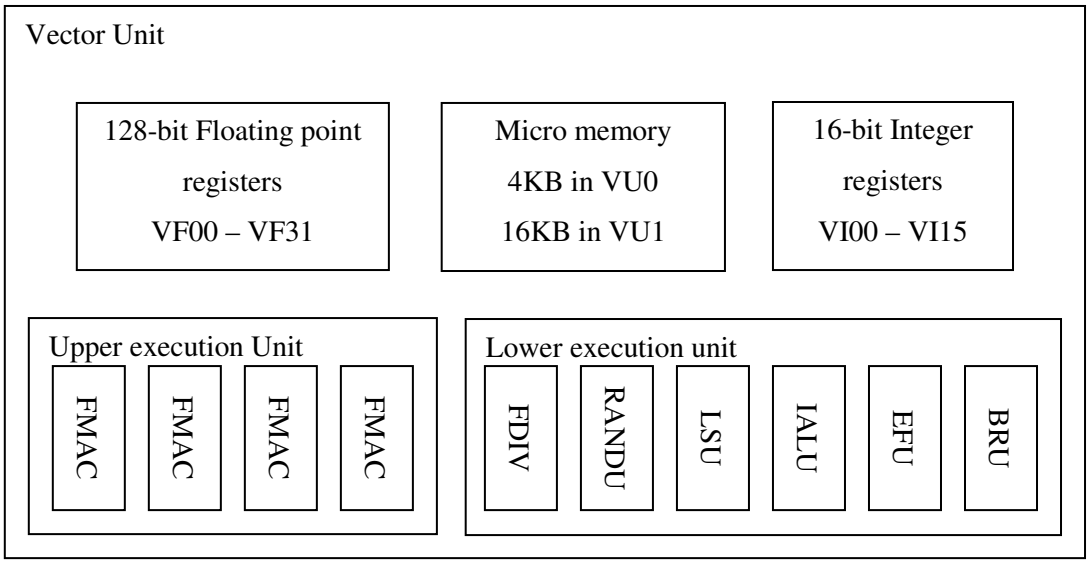


Figure 2-15

Figure 2-15 shows the different parts which make up a vector unit. There are 32 floating point registers which can each hold 4 single precision floating point numbers. There are 16

integer registers, which are only 16-bits in length, and so are typically only used for loops counters, and array indices. There are also a number of control registers, which have not been shown in Figure 2-15. These are similar to the floating point control registers and contain flags showing the status of execution, i.e., if any errors have taken place. In addition to these registers, there is a block of micro memory. This is in addition to the data memory in the vector processing unit, and contains the instructions to be executed when operating in micro mode. It is not used by vector unit 0 when operating in macro mode.

Floating point registers

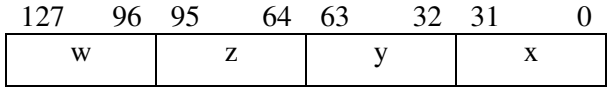


Figure 2-16

Figure 2-16 shows the layout of a 128-bit floating point register in the vector unit. There are four fields: w, x, y, and z, corresponding to the four fields in typical graphics vectors. The data memory in the vector units is also arranged in this way, i.e., in 128-bit blocks divided into the same fields as the registers. This allows instructions to be written to load and store a whole register to and from memory without having to take into account which field goes in which location in memory. Similarly to the floating point coprocessor, there is also an accumulator register, but this is now 128-bits wide to support the size of the floating point registers in the vector unit. None of the vector unit floating point registers have been reserved for parameter passing, stack manipulation, or other similar operations, with the exception of the first floating point register – VF00 – which, like the $0 register in a MIPS code, is a constant register. Unlike the MIPS constant register, this does not contain zero, but instead contains zeros in the x, y, and z fields, and a 1 in the w field.

Execution Units

In addition to the registers and internal memory, the Vector Unit contains two processing units: the upper execution unit; and the lower execution unit. Figure 2-15 shows these units and the units they contain which are used to implement their instruction sets. It is possible for the vector unit to execute an upper and lower unit instruction simultaneously, and so vector unit instructions are 64-bit in length: 32-bit for an upper instruction; and 32-it for a lower instruction. It should be noted that if an upper and lower instruction try to write to a register at the same time, priority will be given to the upper instruction and the lower instructions result will be discarded. Although Vector Unit 0 does contain a lower execution unit, it can only use it is micro mode, and not in macro mode. This is because the MIPS core only accepts 32-bit instructions, which can be sent to the upper execution unit, but 64-bit instructions are needed to give the lower execution commands at the same time. The lower

unit could have instead been chosen to work in macro mode, but as will be seen shortly, it tends to contain instructions to move data around, and operate on 16-bit integers, and has very few arithmetic instructions on floating point numbers, so the upper execution unit is likely to be more useful as a coprocessor.

Upper execution unit

The upper execution unit contains 4 FMAC's – Floating point Multiply Accumulators. These are arithmetic units which can each add, subtract, and multiply a single 32-bit floating point number, or even perform a multiplication and addition at the same time. By combining 4 of these, it is possible for the upper execution unit to process 4 floating point numbers in parallel. These are used to implement all of the true vector operations in the vector unit, i.e. operations that operate across a number of vales at the same time.

The upper instructions have a number of formats depending on their purpose, for example, there is a format for operations which use two floating point source registers and one source register, shown in

COP2	co = 1	dest	ft	fs	fd	OPCODE
31……26	25	24….21	20….16	15….11	10……..6	5…..1

Figure 2-17. There are also instructions known as broadcast instructions which can use a single field of a source register, and apply this to all the fields on another source register. An example of a broadcast instruction is VADDbc which adds the floating point value in the broadcast field to each of the four fields of the other source register.

COP2	co = 1	dest	ft	fs	fd	OPCODE
31……26	25	24….21	20….16	15….11	10……..6	5…..1

Figure 2-17

Figure 2-17 shows the macro mode instruction format called "MacroOP Field type 1" in the Playstation 2 Vector Unit manual [13]. The fields have the following purposes:

- COP2 tells the MIPS core that this is an instruction for coprocessor 2, i.e., VU0
- co is a bit permanently set to 1 signifying a coprocessor instruction
- dest is 4 bit destination field and will be covered shortly
- ft, fs, and fd and the two source registers and a destination register respectively.
- OPCODE contains the instruction to be executed by vector unit 0.

The dest field contains 4 bits: one for each of the w, x, y, and z fields in the floating point registers. It is possible to tell the instruction to operate on any combination of the w, x, y, and

z fields instead of having to operate on all four. This would be useful if, for example, it was necessary to scale the x, y, and z fields, but not the w field. To do this, a multiply instruction could be applied but with only the x, y, and z fields set to 1, and not the w field. The dest field is present in all of the macro mode instructions, and the corresponding micro mode upper unit instructions. This allows only a few fields to be operated on, as well as making it possible to only write to certain memory locations, or load from certain memory locations.

Another advantage of using the destination fields is that it allows the vector unit to operate on floating point vectors whose length is not a multiple of four. In Vector Pascal, it will use vector instructions for as much of a vector operation as possible, and then use corresponding normal operations on the remaining elements. In the vector unit, there are no so called normal instructions so the burden of completing the operation on the final few elements would rest on the MIPS core, or specifically the floating point unit. By using the destination fields, the remaining 1, 2, or 3 elements of the floating point vector can be operated on without fear of overwriting the memory of the data after the vector in memory.

Lower execution unit

Unlike the upper execution unit, this unit does not contain duplicates of any of its subunits, but instead has a number of subunits, each of which provides a different piece of functionality. These are:

- FDIV – used to execute floating point division and square root calculations, which are stored in a dedicated Q register, not a normal floating point register.
- RANDU - generates random numbers for those instructions which require this functionality.
- LSU – the load and store unit which moves data between registers and the vector processing units' data memory.
- IALU which is a simple arithmetic unit to add and subtract 16-bit integers.
- EFU – the elementary function unit. This unit is only present on Vector Unit 1, and is used to implement instructions such as exponentiation, logarithms, and trigonometric functions. Some EFU functions take vector inputs, while others take scalar inputs, but the results are all scalars and are stored in the dedicated P register.
- BRU – this deals with branching instructions, i.e., jumps, and causes the vector unit to move its program counter to a new position in micro memory.

Although the lower execution unit can perform a number of useful operations, including square roots, and trigonometric functions, these are not suitable for use on a single number, as

the time taken to transfer the number to the vector unit, start the vector unit, do the calculation, and transfer the result would likely be far longer than the time taken to execute the given operation in the MIPS core, or floating point unit. However, given a vector of floating point numbers, and an operation such as square root, it would be efficient to transfer the whole vector to the vector unit, perform a square root on each element of the vector, and transfer the result back, so long as the vector is of a reasonable size. Generating sequences of instructions to perform this type of operation should be possible in ILCG.

Differences between the Vector Units

A number of differences have been highlighted between VU0 and VU1. These are:

1. VU0 can operate in macro mode as a MIPS coprocessor, VU1 cannot.
2. VU0 has only 4KB of data memory and 4KB of micro memory, VU1 has 16KB for each.
3. VU1 has an additional unit in its lower execution unit called the EFU which performs, among other things, trigonometric functions.
4. VU1 can send data to the graphics synthesizer to be displayed on screen, VU0 cannot.

Controlling the vector units

For a vector unit to operate in micro mode, its memory must be filled with the data and instructions needed, and then it must receive a signal to start execution. Starting a micro code routine in Vector Unit 0 is very simple because there is a macroinstruction called VCALLMS which tells VU0 to start execution at a given address in its micro memory. Since there is no direct link between VU1 and the MIPS code, a corresponding instruction for VU1 is not possible. However, VU1 watches a VU0 control register called CMSAR1, and if a value is passed to this register using the macro instruction CTC2, which transfers data from the MIPS code to a given control register, then VU1 will begin execution at the address in its micro memory which is stored in the CMSAR1 register. If VU1 is executing, then any change to this register will be ignored. It is also possible to use the VIF to start execution in either vector unit by means of a VIFcode instruction called MSCAL containing the start address of execution.

The VIF

When the DMA memory controller in the Playstation 2 is instructed to pass data to the vector units, it first compresses the data to increase the amount of data that can be sent using the limited memory bandwidth. The VIF must then decompress this data and place it in the micro memory of data memory for its parent vector processing unit. The VIF also has its own instruction set, and it is therefore possible to write commands which will be executed by the

VIF. These instructions do not provide the same functionality of typical processors, i.e., arithmetic operations, but are instead used to notify the VIF to transfer data from the main memory to the data or micro memory of a vector unit. VIF instructions cannot access the data of the vector units while the vector unit is executing, or another VIF instruction is accessing the VU memory, and will stall in either of these cases.

This is one of the possible ways to get data into the vector units. The other is by use of the memcpy C function, which can be instructed to copy some data from one location in memory to another. However, to use memcpy, both the source and target addresses must be in the same physical memory, i.e., both in the main memory on the Playstation 2. This would mean that using memcpy would not be possible for transfer to the vector units' memories, however, the vector units memories have been mapped to the physical memory in the Emotion Engine. This means that it is possible to set the data and instructions for each processor by simply transferring to the appropriate position in memory. It is also possible to set the VU1 registers since they have been mapped to the VU0 data memory, which is itself mapped to main memory. The VU0 registers do not have to be mapped to memory because they are accessible via the macroinstructions when operating as a coprocessor. Although it is possible to use memcpy to set vector unit memory, and VU1 registers, it is preferable to use to the VIF instructions instead, as they can execute while the main processor continues execution of other instructions, but memcpy would have to use the main processor to execute.

2.6 MIPS Memory Allocation

When a program is compiled, it is impossible to know where is will reside in the physical memory of the system it is executing on. For this reason, programs are compiled into their own address space, ranging from 0 to $2^{31} - 1$ and which gives the program the impression that it has exclusive access to the complete memory of the system. When the program executes, these virtual addresses within the program are translated into the physical addresses of the locations they really occupy. Within the virtual address space a program believes it has access to, there are a number of sections, each of which has a fixed start address on MIPS processors.

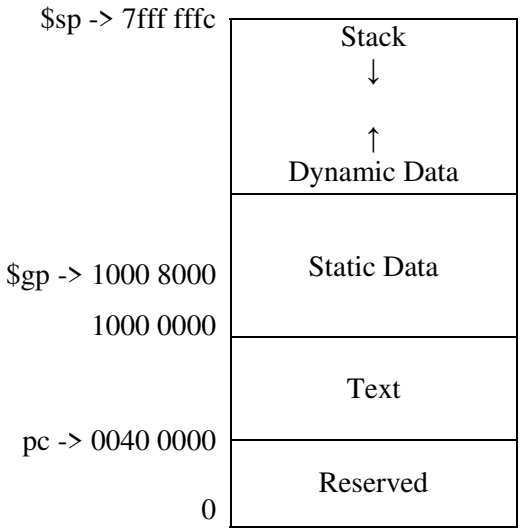


Figure 2-18

Figure 2-18 shows the different segments in a MIPS program, and the addresses they may occupy. The addresses are given as hex values. The text section, which starts at 40 0000 and finishes at 1000 0000, contains the instructions for the program. The first instruction will be in memory location 40 0000 and so this is the location the program counter points at when the program begins execution. Above this is the static data segment which stores constants, and other static variables. The global pointer is set to point to 1000 8000, which is an address within the static data section. Given that the load and store word instructions use a 16-bit immediate value as an offset, it is possible to address any memory location from 1000 0000 to 1000 ffff using the global pointer as a base. If the global pointer did not point to the data segment, then accessing static data would need two instructions: one to load the start address of the static data section into memory, and another to load the specific variable required:

```
la $s0, 1000 0000
lw $s0 4000($s0)
```

Figure 2-19

This can be contrasted with Figure 2-20 which uses the global pointer to access the item at 1000 4000 and takes only one instruction.

```
lw $s0 -4000($gp)
```

Figure 2-20

The final section of memory is empty when the program starts, but will be used by the stack and the heap during program execution. The stack grows downwards from the top of this section of memory, while the heap grows upwards from the end of the static data section.

2.6.1 Position Independent Code

On the Playstation 2, the C Compiler compiles all code to be position independent code (PIC). This means that all references within the code are relative, i.e., are offsets from the program counter, or the global, stack or frame pointers. This allows the program to reside at any location in memory when it is called, and not be restricted to having to occupy specific virtual memory addresses in the current processes virtual memory, i.e., the code will execute correctly no matter where it is positioned in memory. This can be contrasted with absolute addressing whereby instructions can contain absolute memory addresses. This means that for the code to execute correctly, the absolute addresses must coincide with the virtual addresses for the process, or else the data referenced in the instruction will not be in the location the instruction thought it was in.

In absolute addressing, all labels in the assembly code and pieces of data in the data section can be given specific addresses. Any references to these labels within the assembly instructions, for example, a load instruction, can be replaced by the address of the label, and similarly for data items. However, the Playstation 2 uses position independent code, which means that absolute addressing is no longer allowed, and all references must be offsets.

There are two main techniques for satisfying the requirement that there are no absolute addresses in the assembly code:

1. Jumps to labels for branches or functions are replaced by jumps relative to the current program counter.
2. References to absolute addresses are replaced by a piece of code which can be used to calculate the address of the required piece of data.

The first one of these can be satisfied by the compiler, which works out at what value the program counter would be at for each label to jump to. Any jumps to labels will then be replaced by jumps to the specific values the program counter takes at each label. For example, if the original code to jump to DEST was:

Program counter	Instruction
0	main:
4	j LABEL
…..	…..
64	DEST:

Figure 2-21

Then the PIC equivalent would be:

Program counter	Instruction
0	main:
4	j 64
…..	…..
64	DEST:

Figure 2-22

Figure 2-22 shows that the label DEST has been left unchanged, but the jump instruction which used to reference it directly, now jumps to the program counter value the label occupies. This will all be done automatically by the compiler since it can pass over all the code finding the value the program takes at each label, and then pass back over the code replacing all references to the labels by the program counter values.

The second technique is more complicated than the first and involves the creation of a global offset table.

Global Offset Table

Although absolute addresses can no longer be used in the assembly code, the data for the program will exist at absolute addresses during program execution. Each object file contains its own global offset table, which holds the addresses of all the functions and data values in the object file [15]. References to absolute addresses are replaced by lookups to the corresponding entry in the table, which at run-time will contain the absolute address of the item required. The global offset table is held in the data segment of the object file, and is referenced through the global pointer. Since each object file has its own global offset table, when a function is called, it must set the global pointer to point to its own global offset table, as the global pointer would have been pointing to the global offset table for the calling procedure, which may have been in a different object file. Functions called from other functions in the same object file do not have to reload the global pointer as it will already be

set to the global offset table for that object file. Some compilers provide an alternative entry point to procedures in the same object file which skips the loading of the global pointer [14] as it is unnecessary, and this reduces the overhead of calling position independent code.

The MIPS assembler provides two pseudo instructions which programmers can use to setup the global pointer to use the global offset table. These are cprestore and cpload. The pseudo instructions to interact with the global offset table have been prefixed with cp which stands for context pointer and is another name for the global pointer.

```
.set noreorder
.cpload $25
.set reorder
```

Figure 2-23

Figure 2-23 gives the assembly pseudo instructions to set the global pointer to point to the global offset table for the current object file. The cpload pseudo instruction is expanded into the two instructions in Figure 2-24. The set noreorder/reorder pseudo instructions tell the assembler to not reorder the instructions that cpload expands to, since the cpload instruction must be the first instruction to be executed in a function, and reordering could move it elsewhere. If the cpload instruction was not the first to be executed in the function, then the global pointer is not guaranteed to have been set correctly and then any references which followed it may not contain the correct pointer to the global offset table. The _gp_disp variable in Figure 2-24 is the offset from the address of the start of the function, and the global offset table. This variable will be replaced by the actual number it represents by the linker, which will know how far the function is from the global offset table. By adding this to the virtual memory address of the function, the global pointer will then contain the run-time virtual memory address of the global offset table. Given that the code is position independent, the virtual memory address of the function will not be known until run-time. The MIPS Application Binary Interface (API) [16] specifies that the “virtual address of a called function is passed to the function in register $25”. The second line of assembly cpload expands to add this virtual memory address to the offset to give the final value the global pointer should contain to point to the global offset table.

```
la $gp, _gp_disp
addu $gp, $gp, $25
```

Figure 2-24

Although the assembly code in Figure 2-24 will set the global pointer correctly, it has now added the requirement that the virtual address of the current procedure is contained in register

$25 when the procedure is called. The assembler satisfies this requirement by finding all of the jump and link instructions and replacing them with jump and link register instructions, where the register containing the address to jump to is register $25. Of course, prior to the jump and link register instruction, the address of the procedure to call must be loaded into register $25. This would typically have been done using a load address instruction, but that would rely on an absolute address, and so will instead be replaced by a lookup into the global offset table.

Once the global pointer has been set correctly, it is then necessary to save it to the stack. This is because when the current procedure calls another, the value of the global pointer will be set to the called functions global offset table, and so once it returns, any references to the global offset table in the current function would be invalid. The assembler provides the cprestore pseudo instruction to save the global pointer to a position in the stack, and reload from this position after each jump and link instruction, which correspond to procedure calls. The cprestore instruction is simply passed a number which represents the offset from the stack pointer in which space to save the global pointer has been reserved. For example, “cprestore 16” would emit “sw 16($sp)” at the position in the assembly where the cprestore was written. The assembler also has to reload the global pointer from this position on the stack after each jump and link instruction. It does this by searching for all the jump and link instructions and, in this case, emits a “lw 16($fp)” instruction at the position where execution would continue from once the called procedure returns. It can be seen from the generated instructions that the assembler uses the stack pointer as the base from which to save the global pointer, but uses the frame pointer as the base for restoring the global pointer. This is because the MIPS ABI [16] states that the frame pointer should point to the same location as the stack pointer once the procedure starts executing its body of code. The stack pointer is then free to change value to allow new data to be pushed onto the stack. However, as will be seen in the section on the stack, this assumption made by the MIPS ABI, which has been implemented in the Playstation 2 assembler, conflicts with the assumptions present in the Vector Pascal Compiler with respect to the layout of the stack, and the stack and frame pointer.

3 Design

3.1 The Stack

When a program is executed, there are typically two areas of memory it has access to: the heap; and the stack. The heap is an area of memory used to store data which may be shared by many parts of the program, and tends to live longer than the procedure that created it. This can be contrasted with the stack which is used by each procedure to store data local to that procedure, and that will no longer exist once the procedure exits. A procedure is given a stack frame when it is entered which has enough space for any registers that must be saved while the procedure is running, and also enough space for any local variables the procedure uses. In some circumstances, for example where the procedure runs out of registers to use in a calculation, it may also spill registers onto the stack, which means that the registers value is stored on the stack, to be reloaded later, possibly to be used to complete the calculation it was being used in.

The stack is accessed via registers called the stack pointer (SP) and frame pointer (FP). The frame pointer contains the address of the start of the stack frame of the current procedure whereas the stack pointer contains the address of the end of the current procedures stack frame. This conflicts with the Playstation 2 assembler which assigns both the stack and the frame pointer to the end of the stack. The stack on most platforms grows downwards, meaning that the end of the stack is at a lower address than the top. This also means that to save a variable to the stack – pushing it onto the stack – the space for the variable must be subtracted from the stack pointer.

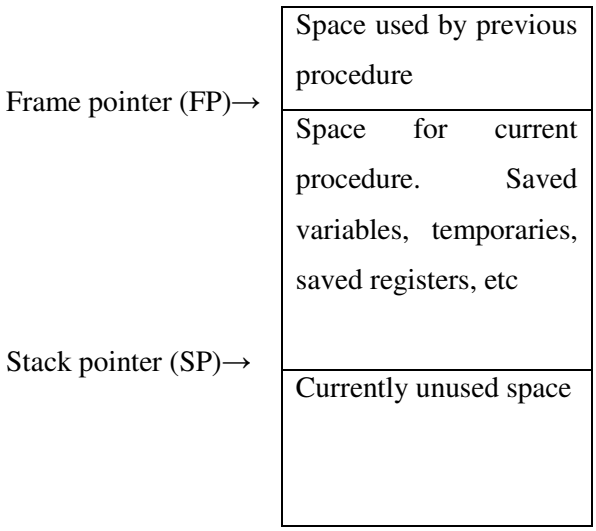


Figure 3-1

When a local variable needs to be accessed, its address is given as an offset from the frame pointer. The offset could have been given from the stack pointer, but since the offset is a fixed number, changes to the stack pointer would invalidate all the offsets, and normally compilers cannot guarantee that the stack pointer will not change during execution of a procedure. As will be seen later, the implementation of the port to the Playstation 2 could make this guarantee, but the assumptions in the compiler relating to the stack and frame pointer would make it impossible to use just the stack pointer, and not the frame pointer.

When a procedure is called, there are a number of values which have to be saved onto the stack. These values include the frame pointer, which will be moved to point to the top of the new procedures stack frame, and so must have its value saved so that it can be restored when the procedure exits. The return address must also be saved. This is the address of the next instruction to execute after the line of code where the procedure was called. When the procedure exits, it should return to this address so that execution of the program continues from the point at which the procedure call occurred. The return address is stored in a register, but must be saved since this register is overwritten with each procedure call, and so will be lost if a called procedure then calls another, i.e., the first called procedure will not know where to return to when it exits because the return address was overwritten when it called a procedure.

There are a number of assumptions made by the compiler with respect to the positions of variables on the stack. The first of these is that the parameters to the current procedure are a fixed offset up from the frame pointer. This means that the frame pointer must retain the same value throughout execution of the procedure to ensure that the offsets to the parameters remain valid. The second assumption also concerns the frame pointer, but this time relates to the display which is a list of the frame pointers of all the procedures the current procedure is nested within, and by indexing into this list, the current procedure can use any value seen by parent procedures. Similar to the arguments, the offsets from the frame pointer to the display values are also assumed within the compiler, and so must be adhered to when creating the display and setting the value of the frame pointer. Both these assumptions have highlighted the need for the frame pointer to remain the same throughout execution of the procedure, but given that the arguments and display are on the stack, it is also important to make sure that no other variables try to occupy these positions on the stack, since they would then corrupt the memory of whatever variable was in the memory they used.

3.1.1 Entering a procedure

A procedure is entered when program execution jumps to the label that marks the beginning of the procedure. There are a number of steps which must take place when entering a procedure. These are:

1. The return address register must be saved since if this procedure calls another, the value in this register will be overwritten.
2. The frame pointer must be saved, so that when the procedure exits, the frame pointer can be restored to point at the address it was pointing at before calling the procedure.
3. Any of the saved registers which are used during execution of the procedure must be saved. This is to comply with the MIPS standard which puts the responsibility on saving these registers on the called procedure, and not the calling procedure.
4. In the Playstation 2, the global pointer must be set up for this procedure.
5. A list of frame pointers from parent procedures – called the display – must be created.

The Display

In a language such as Pascal which allows for procedures to be nested within each other, a mechanism must be provided which allows an inner procedure to find the memory locations where variables from the outer procedures reside. The implementation used by Vector Pascal is the display. This is a list of the frame pointers for each of the parent procedures of the current procedure. Since all the variables for procedures are given as given offsets from their frame pointer, it is possible to find any variable from outer procedures if their frame pointer and the required offset are known. For variables in Vector Pascal, the frame pointers in the display are all at fixed offsets from the current procedures frame pointer, and the offsets for any variable are stored within the variable object in the language and so can be output directly in the assembly code, i.e., no calculations are needed to calculate the offset at runtime.

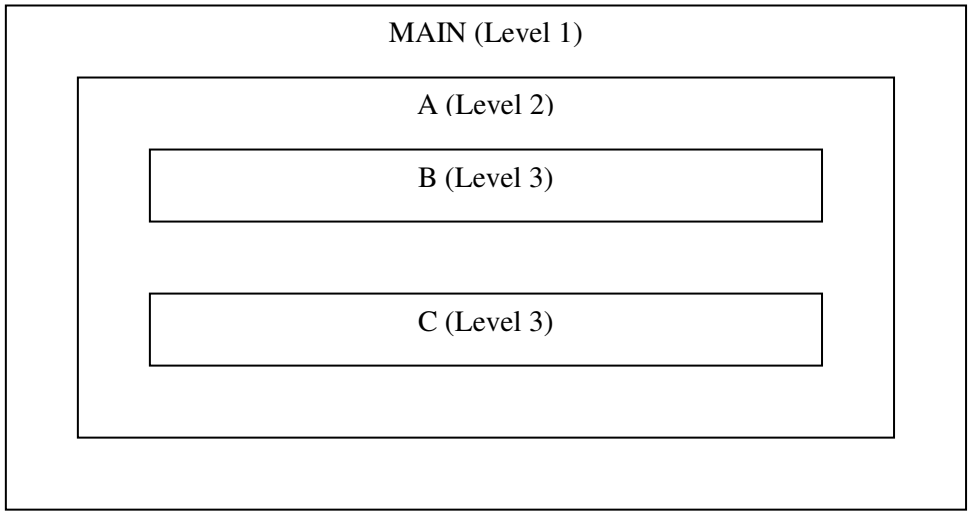


Figure 3-2

Figure 3-2 – adapted from the Intel Architecture Software Developers manual [1] - shows procedures B and C, both nested within procedure A, which is itself nested within the MAIN procedure. On the Intel platform, a nested procedure is assigned a level, which gives a measure of how many frame pointers are contained in its display, and so a higher level means that a procedure is more deeply nested than a lower level procedure. A procedure at level N will have a display which contains N frame pointers, the last of which will be its on frame pointer. Intel assigns the outermost procedure the level 1, whereas in the Vector Pascal Compiler, the outermost level is assigned a level of 0. This makes little difference in the implementation, since the variables in the main procedure in Vector Pascal are not stored on the stack, but are instead held within the data section of the compiled program. References to these variables, at any nesting level, are not given as offsets from the frame pointer, but instead from a fixed label in memory. Therefore, in both the Vector Pascal Compiler, and Intel's implementation of the display, the level 1 procedures are the first to have a frame pointer from which variables are offset. The x86 compatible specifications in the Vector Pascal Compiler use the ENTER instruction, which is built into the processor, and which automatically creates the display for the given level of procedure. The ENTER instruction puts the frame pointers in the display at fixed offsets from the frame pointer for the current procedure and the Vector Pascal Compiler assumes that these frame pointers in the display are at the offsets the ENTER instruction saved them at. On the Playstation 2, there is no ENTER instruction, but Intel has documented its behaviour, so the same functionality can be synthesized from more basic MIPS instructions. The ENTER instruction can also create space for the stack frame of the current procedure by subtracting a storage amount from the stack pointer, although this is not needed on the Playstation 2, since the stack pointer is subtracted at a different time. The pseudo code for the ENTER instruction is given in Figure 3-3.

```
PUSH frame pointer (FP)
FRAME_PTR = stack pointer (SP)
IF level > 0 THEN
        DO (level -1) TIMES
                FP = FP – 4
                PUSH pointer(FP) //push data pointed to by FP
        OD;
        PUSH FRAME_PTR;
FI;
FP = FRAME_PTR;
SP = SP –STORAGE;
```

Figure 3-3

The above code saves the current frame pointer to the stack. It then stores the current stack pointer in a temporary variable called the FRAME_PTR, which will just be a register. Next, it loops a given number of times, and at each iteration subtracts from the frame pointer, and then copies the data the frame pointer is pointing to. This has the effect of copying the display from the outer procedure, since the display for the outer procedure occupies the locations in memory that are being copied into this procedures display. Once the display from the calling procedure has been saved, the current procedures frame pointer is pushed onto the end of the display. This is not used by the current procedure, or any other procedures at the same level as the current one, but is instead saved so that procedures at a lower level will have access to this procedures stack frame because they will copy this frame pointer from the end of the display when they set up their own display.

The figures below show the contents of the stack frame after calls are made to the procedures MAIN, A, B, and C assuming the nesting given earlier. The first shows the stack frame of MAIN once it has been entered. The second shows the stack frame for A after it has been called from MAIN. Similarly, the third shows the effect of calling B from A, while the fourth shows a call from B to C, which demonstrates what happens when a procedure calls another at the same nesting level as its own, and not a child procedure.

After entering MAIN

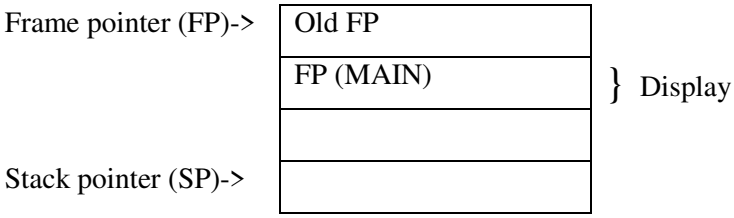


Figure 3-4

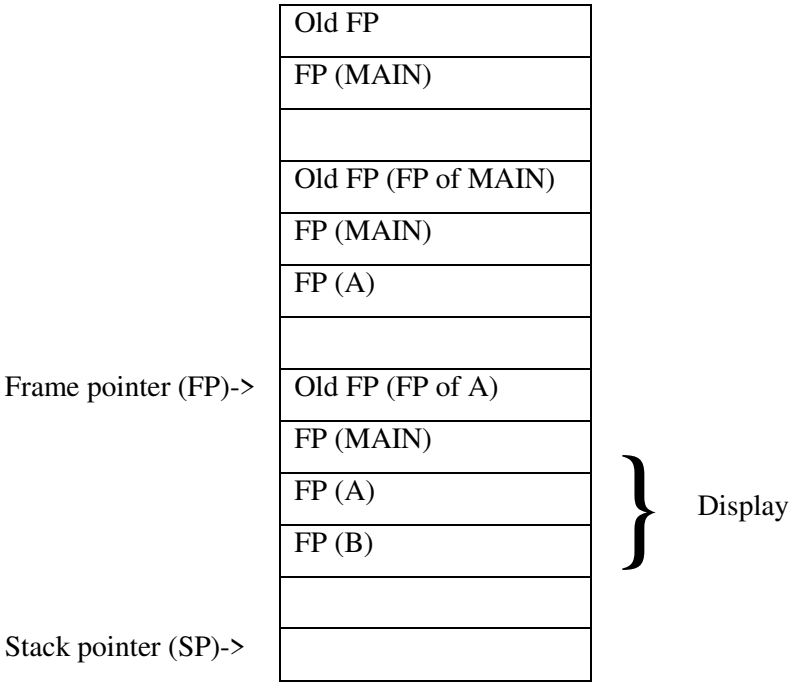
The stack frame is very simple for MAIN. It saves the old frame pointer so that it can restore it once it exits. It is at level 1 and so is not nested within any outer procedures. This means that it does not have to copy any frame pointers from outer procedures into its display, so its display simply contains its own frame pointer for child procedures to copy into their own displays. Some space has been given after the display, which would contain saved registers and local variables. The stack pointer points to the end of this allocated space.

After entering A from MAIN

Figure 3-5

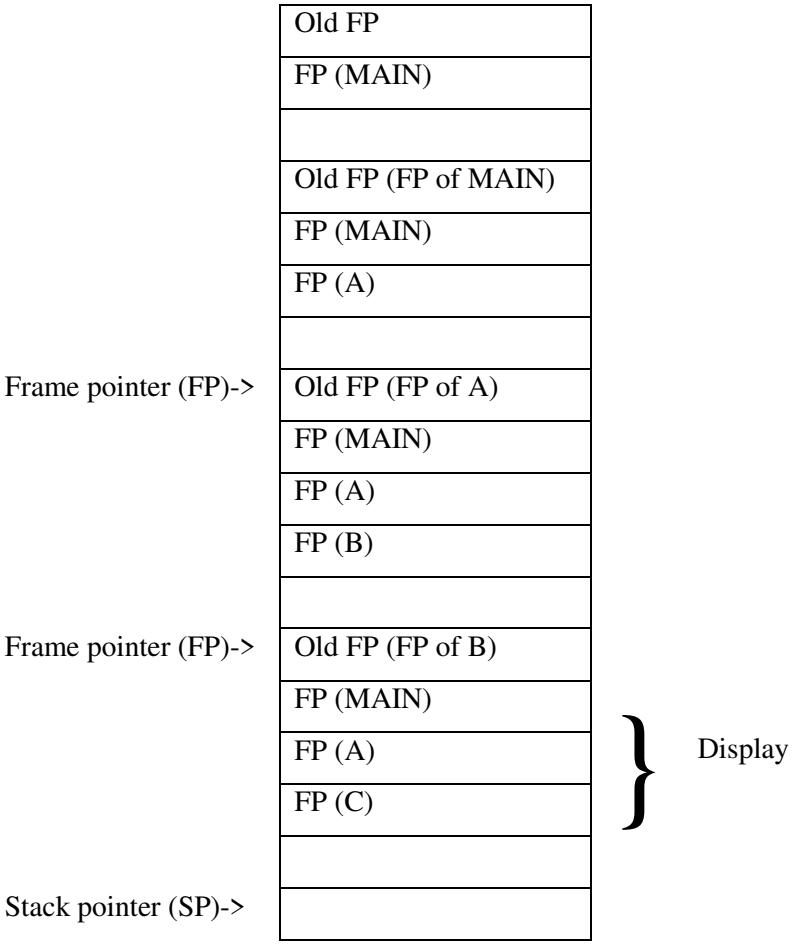
When A is entered, the old frame pointer – in this case the frame pointer for MAIN - is saved to the location where the stack pointer currently addresses. This procedure is at level 2 so copies 1 frame pointer from the display of MAIN, which is the frame pointer for MAIN, which it saved for nested procedures such as this own to copy. The frame pointer for A is then saved to the end of the display. Finally, the frame pointer for A is moved to point at the top of the stack frame for A, which is the bottom of the stack frame from MAIN, i.e., the location the stack pointer addressed before A was called.

After entering B from A



Similar to MAIN calling A, when B is called from A, it saves the old frame pointer – which is now the frame pointer for A – to the stack, but this time copies two frame pointers from the display A created. As with the other cases, it then copies its own frame pointer to the end of the display, and moves the frame pointer to point to the top of its stack frame.

After entering C from B (Note, B and C are at the same level)



When C is called from B, it saves B's frame pointer, and then has to copy the display into its own stack frame. However, unlike the other cases, C in not nested within B, but is instead at the same level of B. This means that it should not have access to B's stack frame, and thus B's variables. The loop to copy the display deals with this because C will copy 2 frame pointers from the display into its own display. These two frame pointers are both copied from B's display, but are pointers to MAIN and A respectively. This allows C to access the variables for both MAIN and A, which is correct since C is at the same nesting level as B, and they are both nested with A and MAIN. The frame pointer to B's stack frame is not copied because it is in the third position of B's display, and since C is at level 3, it only copies the first 2 positions. As before, C then copies its own frame pointer to the end if its display, and moves its frame pointer to the top of its stack frame.

This example dealt with a procedure calling another at the same nesting level, however, the ENTER instruction also handles a child procedure calling a parent procedure correctly. For example, if C was to call A, then A should copy the frame pointer to MAIN from C's display.

This will be the case using ENTER because A is at level 2 so will copy one frame pointer from the calling procedures display – the frame pointer to MAIN. The pointer to MAIN is guaranteed to be at the top of C's display since C is nested within MAIN, and MAIN is at level 1. If MAIN's frame pointer was not at the top of C's display then C would not be nested within MAIN, and so would not be able to call A.

As mentioned in the section on the stack, one of the assumptions built into the Vector Pascal Compiler is the offsets from the current procedures frame pointer to the frame pointers of all parent procedures in the display. These examples clearly show that the top of the display is addressed by taking the frame pointer and subtracting 4 from it. This gives the address of a level 1 frame pointer. Simply subtracting another 4 will give a level 2 pointer, and so on. This assumption also prevents the stack and frame pointer from pointing to the same location in memory, since that would force the display to be beyond the end of the stack, i.e., within the space that would next be used by the stack. This causes problems with restoring the global pointer, as, seen in the section on the Global Offset Table, since the assembler assumes that the frame pointer points to the end of the stack frame, not the start of it. If the frame pointer points to the top of the stack frame, then the global pointer will be saved to an address below the frame pointer, but restored from an address above the frame pointer, which will not contain the saved global pointer, but instead the arguments to the current procedure. There are a number of possible solutions to this problem:

1. The global pointer is in a memory location lower than the frame pointer, so a negative offset could be passed to the cprestore pseudo instruction.
2. An offset of 0 could be passed to cprestore; this would allow the global pointer to be restored from the frame pointer and would not interfere with the procedures arguments.
3. There is no requirement that the Pascal code use the $fp register as a frame pointer, so the $fp register could be set to equal the $sp register as the assembler assumes, and another register could be used as the frame pointer to access variables, arguments, and the display.

The first solution is not possible because the assembler does not allow a negative offset to be given to the cprestore instruction. By looking at the instructions generated by the cprestore method, there is no reason to restrict a negative offset based on the generated instructions alone. However, it is likely that the assembler writers did this to prevent saving to an address lower than the frame pointer because they assume the frame pointer points to the end of the stack frame, and so addresses lower than the frame pointer would be invalid.

The second solution would be possible, but the compiler tends to save the old frame pointer in the address pointed to by the current frame pointer, and the return address in the memory location above that. These are the only two locations which could have been used to save the global pointer, as higher addresses are used for arguments. Although it would have been possible to use one of these locations to save the global pointer, it was thought best to keep the layout of the stack as close to other platforms to save confusion.

The third method was implemented in the port to the Playstation 2. This involved replacing all occurrences of the frame pointer with the register which would be the new frame pointer. The $fp register could then be moved to point to the same piece of memory as the stack pointer, and so any attempts to reload the global pointer will be successful. Both the stack pointer and frame pointer are guaranteed to be unchanged after a procedure call, however, if the frame pointer is not chosen wisely, then it may not have the same value after a procedure call as it had before the call. The solution to this is to use one of the saved temporary registers as the frame pointer, since if a called procedure changes its value, it must restore the value before exiting.

3.1.2 Procedure Call Mechanisms

In order to call a procedure is called, there are a number of steps the compiler must take:

1. Any caller saved registers, i.e. the unsaved temporaries, with values that are needed after the procedure call are saved
2. The arguments to the called procedure are placed in registers and/or on the stack so that the called procedure can access them
3. The return address is loaded into the return address register
4. A jump instruction is executed to cause the program counter to begin execution at the start of the called procedures body.

The third and fourth points in the above list are both satisfied the using the jump and link instruction which is covered earlier in the Global Offsets Table section. The first point involves working out which unsaved temporary registers hold values needed after the procedure call and pushing their values onto the stack. These values can then be restored after the procedure call. However, this relies on the Vector Pascal Compiler knowing the location of all procedure calls, which is impossible on the Playstation 2. This is because, although for standard procedure calls, a method will be called to load the arguments correctly and perform the jump to the procedure, there are procedure calls in the specification which are essentially hidden to the compiler. These are the calls to the double precision floating point operations, which will be covered shortly, and also calls to Math routines for trigonometric

results. Note that it would be possible to search the assembly output from the compiler and determine where all the function calls are, and then establish which registers need to be saved before each function call. However, this would be non trivial and has not currently been implemented in the port of Vector Pascal to the Playstation 2. It would also slow the compiler since it would have to make at least one more pass over the code to work out what registers to save.

The final point to cover is the method in which parameters are passed to procedures. Both the Vector Pascal Compiler and the MIPS GNU C compiler have assumptions over where the arguments will be located. The Playstation 2 port must put the arguments to procedures in exactly the correct registers and memory locations or else it will be impossible to call either C functions, or other Pascal procedures.

Passing Parameters to Pascal Procedures

In Vector Pascal, all arguments are pushed onto the end of the stack. None of the argument registers are used. Arguments smaller than 4 bytes, for example, 16-bit integers, or characters, are aligned on 4 byte boundaries, i.e., they are given 4 bytes of space, even though they need less. Arguments of larger than 4 bytes are simply put straight onto the stack. When a Pascal procedure is called, it assumes that the last argument to be passed is at the top of the stack – the location pointed to by the stack pointer. Once the procedure has been entered, this argument will be at the location given by adding 8 to the frame pointer. If all arguments were 4 bytes in length, then the second would be 12 up from the frame pointer, and the third would by 16 up and so on. However, if for example the first argument is 8 bytes in length, then the second will be at an offset of 16 from the frame pointer and not 12. The Vector Pascal Compiler is aware of when arguments larger than 4 bytes are passed and can change the offsets it uses in code accordingly. This does not change the calling code to a great extent since all that is needed is a check for the size of the argument and a suitable subtraction of that amount from the stack pointer to make enough space.

Arguments can be passed in either right to left or left to right order. In a left to right order, the leftmost argument will be pushed onto the stack first, which means that it will be at the highest memory address of any of the arguments, and the stack pointer will point to the last argument. In right to left order, the rightmost argument will be pushed first, and so the first argument will be pointed to by the stack pointer. In Vector Pascal, procedures can use either ordering. The procedure call method, which pushes the arguments onto the stack, can access the class for the procedure it is calling, and retrieve the ordering variable. It can then push the arguments in left to right or right to left order, as expected by the procedure being called.

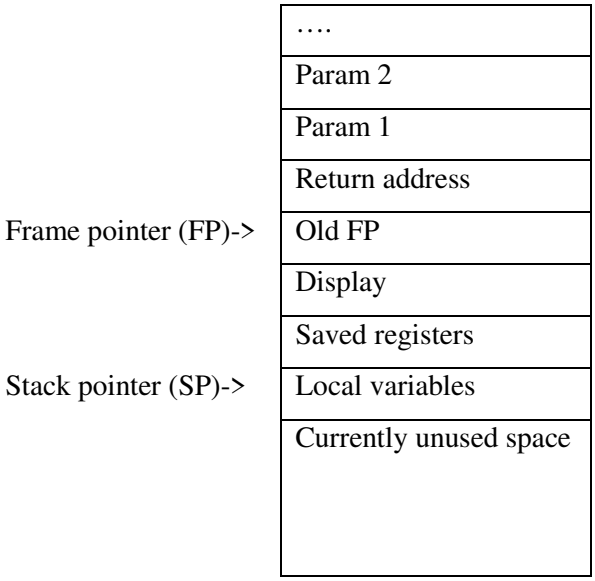


Figure 3-8

Figure 3-8 shows the layout of the stack frame for a procedure in Vector Pascal on the Playstation 2 which was called using right to left parameter passing. The stack frame contains the saved registers, local variables, and display which was all covered earlier. The frame pointer is pointing to the memory location containing the frame pointer from the calling procedure, and the return address is contained in the memory location above this. The offset to the arguments from the frame pointer can be seen to be 8 bytes, since an offset of 0 gives the old frame pointer and an offset of 4 gives the return address. Since the parameters were passed in right to left order, the last parameter was pushed onto the stack first, and so the first parameter is at the smallest offset from the frame pointer. The can be contrasted with Figure 3-9 which shows a call to a procedure with N arguments passed in left to right order.

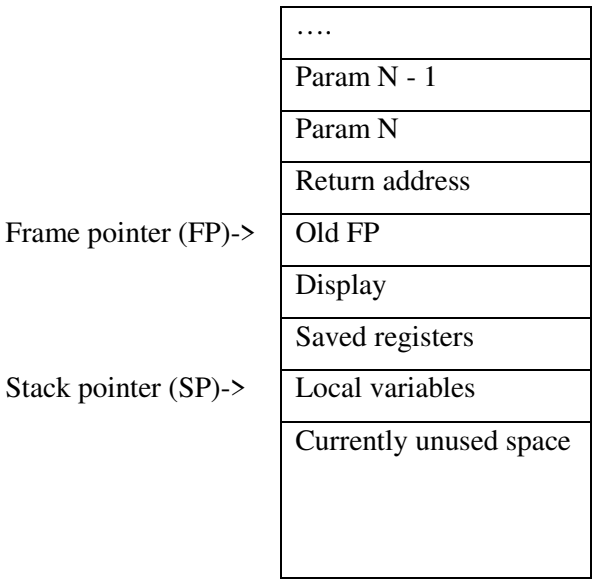


Figure 3-9

Passing Parameters to C Procedures

Whereas the Vector Pascal calling convention puts all the arguments on the stack, the C calling convention is to put as many arguments in registers, and then the remaining arguments onto the stack, although the compiler assumes that space has been saved on the stack for those parameters passed in registers as well, so that the called procedures can saved the parameters to this space if necessary. In C, all parameters are passed in left to right order, that is, generally the first parameter will be in register $a0, with the second in $a1, and so on. Parameters which must be passed on the stack are also passed in this order, so the last parameter will be at the highest memory address. As with Pascal, arguments of less than 4 bytes are promoted to use 4 bytes, either on the stack, or in an argument register. Parameters can be passed in the 4 MIPS argument registers - $a0 - $a3 - and the 4 floating point argument registers, although the floating point registers must be used in pairs, so only register $f12 and $f14 are really available for parameter passing. Both 64-bit integers and double precision floating point numbers, which take up 8 bytes, are passed in the MIPS argument registers, or on the stack, as required. However, even though the MIPS registers are 128-bits in size, 64-bit arguments are instead passed in the lower 32-bits of a pair of registers. In addition to this, they can only be passed in an even/odd pair of registers, not an odd/even pair, i.e., in $a0 and $a1, or $a2 and $a3, but not in $a1 and $a2. Although double precision numbers are floating point, they are not passed in the floating point argument registers.

The floating point registers can store only two arguments. However, they do not store the first two floating point arguments, but instead the first two arguments if they are floating point numbers. This means that if the first argument is a floating point number, but the second is an integer, then the first will be stored in $f12, but the second will be stored in $5. The restrictions on the $f14 are greater in that an argument will only be store in it if the argument is a floating point number, and the second is a floating point number, i.e., $f14 will can only be used if $f12 has been used. If $f12 was not used, the floating point argument will be passed in the $5 register, in a similar way to the third and fourth floating point arguments being passed in registers $6 and $7 as there are then no more floating point registers left to pass parameters in.

Since 64-bit arguments are passed in a pair of registers, it is possible for some of the MIPS registers to not be used, even if there are no arguments in the corresponding floating point register. This can happen if the first argument fits in either the first floating point or MIPS register. If the second argument is a 64-bit one, then the next usable register is $1, but the 64-bit arguments can only use even/odd pairs of registers, so $a1 would not be used, and $2 and

$a3 would store the argument. A similar case is possible if the next register to use is $a3, since this would involve the argument being passed in a register and on the stack. It will instead pass the whole argument on the stack, and so $a3 will be unused. If a 64-bit argument is passed on the stack, it must be passed in memory addresses on 8 bit alignment. This means that if the first argument passed onto the stack is 4 bytes long, then if the second is 8 bytes long, it will not occupy the next available space on the stack, since that is only aligned to 4 bytes, but will instead leave that space unused, and use the 8 bytes after it.

Arguments						**MIPS registers**				**FP regs.**		**First 6 Stack locations (offset from sp)**					
1	2	3	4	5	6	a0	a1	a2	a3	f12	f14	0	4	8	12	16	20
i	i	f	f	i	i	1	2	3	4			5	6				
f	i	f	f	f	f		2	3	4	1		5	6				
d	i	d	i	i	i	1	1	2				3	3	4	5	6	
i	d	f	i	f	f	1		2	2			3	4	5	6		
i	f	i	d	i	d	1	2	3				4	4	5		6	6

Figure 3-10

Figure 3-10 shows a number of possible combinations of arguments and the locations they would be passed either in MIPS or floating point registers, or on the stack. The first block shows the types of the first 6 arguments, given by an i, f, or d, to represent integer, single precision floating point, or double precision floating point. 64-bit integers would behave in the same way as double precision floating point. The second block shows the argument number that is contained in each MIPS argument register, and similarly for the third block and the floating point argument registers. The fourth block gives the first 24 bytes of the stack, and shows the space taken on the stack for the arguments that use it. These examples demonstrate the following:

- The first row shows that even though the floating point registers are not used, then third and forth arguments use the third and fourth MIPS registers instead.
- The second row shows that the first floating point argument will be passed in the first floating point register, and that floating point numbers can be passed in the third and fourth MIPS registers.
- The third row shows that double precision numbers use pairs of registers, and will not place part of a double precision number in a register and part of the stack.
- The fourth row shows that double precision numbers will not use a MIPS register if it is odd, but will instead use the even/odd pair of registers that follows it.

- The final row shows that double precision numbers will skip parts of the stack space if it would put them on a 4 byte alignment, instead of an 8 byte alignment.

3.2 Double precision floating point operations

Due to the lack of a double precision floating point unit in the Playstation 2, arithmetic using double precision numbers must be emulated. The designers of Playstation 2 Linux were aware of this need to perform double precision arithmetic and so provided functions which can be called directly from within the assembly code. There is a C header file called floatemu.h which contains all the functions that have been written for developers to use. There are a large number of these functions available, including all the common operations, such as addition, subtraction, multiplication, and division. The main functions used are given in Figure 3-11.

Function	Description
dpadd(a, b)	Returns the addition of a and b
dpsub(a, b)	Returns the subtraction of b from a
dpmul(a, b)	Returns the multiplication of a and b
dpdiv(a, b)	Returns the division of a by b
dpcmp(a, b)	Returns -1, 0, 1 if a is <, =, > than b
dptofp(a)	Returns the floating point number given by a
fptodp(a)	Returns the double precision floating point number given by a
dptoli(a)	Returns the integer representation of the double a
li.d	Loads a double precision constant into the destination register

Figure 3-11

All the above procedures are called by loading the inputs into the argument registers ($4-7) and retrieving the result from the return registers ($2-3). The exceptions are the fptodp and dptofp which pass the floats in using the floating point argument registers ($f10-f12) and get the float results in the $f0 register. Also, li.d is not a function in the floatemu.h header, but instead a pseudo instruction implemented by the assembler which allows double precision constants to be loaded into registers.

The lack of support for double precision numbers also raises a problem with calling functions with double precision parameters. Despite the fact that the general purpose registers in the Playstation 2 can store 64-bit numbers, double precision floating point numbers are instead split into two 32-bit parts and each is stored in its own general purpose register. This causes a problem with the code generator in the Vector Pascal Compiler because there is no way to

reserve a pair of registers to store a single piece of data. One possibility would be to reserve all of the odd registers in the machine specification, and then only use the even registers. The assembler can then be instructed to use an even register for a double precision number, but will use both the even register and the odd one that follows it. However, this would half the number of usable registers in the MIPS core, and since only the saved temporaries are currently used, there are already a limited number of registers to use. It is likely that general instructions will be executed more often that double precision ones, and so it would be better to maintain a high level of performance for the general instructions, even if that results in slightly lower double precision performance.

However, it is possible to work around this by storing values in a single register and only transferring them to two registers when passing the value as an argument in a procedure call. The Emotion Engine implements the load and store double instructions to load and store 64-bit values to and from memory. If the double precision number can be loaded into a single register, it should then be possible to call the double precision functions in Figure 3-11 by copying the lower 32-bits of the register to an even argument register, and the upper 32-bits to the odd argument register following it. Two possibilities for doing this are given in Figure 3-12 and Figure 3-13, which take the double precision number in register A, and assign the lower ands upper 32-bits to registers B and C respectively.

```
MOVE B, A      // moves the lower 32-bits of A to the lower 32-bits of B
DSRA32 C, A, 0 // shifts the upper 32-bits of A into the lower 32-bits of C
```

Figure 3-12

```
MOVE B, A      // moves the lower 32-bits of A to the lower 32-bits of B
PROT3W C. A. 0 // copies bits 32..63 in A into bits 0..31 in C
```

Figure 3-13

Both these examples should make it possible to get from a single register to two registers for double precision operators. Similar instructions make it possible to get from a result in two registers back to a single register. However, these rely on the load and store double instructions which have the restriction that they must access memory locations aligned on 64-bit boundaries. There is no way to guarantee that this will be the case, and as the Pascal Parameter Passing section covered, if the second argument to a procedure is 64-bits long, but the first argument was less than 64-bits long, then the second definitely be aligned on a word boundary and not double word. The MIPS designers have taken this into account with the load and store left and right instructions. Instead of storing a double to a memory location,

store left and store right can be called with the same memory location. If the location is aligned on a 64-bit boundary, then only one of these instructions will write to memory, but if it is not aligned properly, then part of the register will be saved to memory by the store left instruction, and the other part will be saved by store right. The load equivalents behave similarly.

One other alternative to these methods is to define all the double precision instructions in the machine specification to operate on memory address and not registers. This would mean that the instruction would have to contain all the necessary instructions to load double precision numbers from memory into the argument registers, call the appropriate function, and then save the function result from the result registers back to memory. An example of the sequence necessary to add two double precision numbers is given in Figure 3-14. It assumes that the values to add are in locations with offsets 0 and 8 from the frame pointer, and the result should be written to the memory location with an offset of 16 from the frame pointer.

```
LW $4, 0($fp)
LW $5, 4($fp)
LW $6, 8($fp)
LW $7, 12($fp)
JAL dpadd
SW $2, 16($fp)
SW $3, 20($fp)
```

Figure 3-14

The code in Figure 3-14 uses load word instructions to load the upper and lower bits of the double precision number into registers. Since the Vector Pascal Compiler will align arguments and data in memory to word boundaries, there is no issue with alignment when using these instructions. It is possible to force the assembler to generate a pair of load and store word instructions in place of corresponding load or store double instruction. This can be done by wrapping the instruction in a ".set mips2" and ".set mips0" pair of pseudo instructions. The first one of these will force the assembler to use MIPS II instructions which don't include store and load double as these are part of the MIPS III instruction set, and so the assembler will synthesize these using the equivalent word instructions. The second pseudo instruction will tell the assembler to return to its default setting for future instructions.

3.3 Multimedia instructions

The multimedia instructions are used in the Playstation 2 to operate on vectors of integers in parallel. They are called in a very similar way to the ordinary instructions, and even use a very similar instruction format to the normal instructions. The difficulty with using these instructions lies with loading and saving the registers with 128-bit values. This is because the load quad and store quad instructions both require 128-bit alignment, which cannot be guaranteed in the compiler. However, as with the double precision instructions, it is possible to synthesize these from instructions which don't have this alignment restriction. To generate 128-bit loads and stores, 2 64-bit loads or stores are generated using the load and store left and right instructions previously covered. This will make it possible to load the upper and lower 64-bits into two different registers. All that is required after that is to join the two 64-bit values together to give the 128-bit value the multimedia instructions operate on. To combine two 64-bit values, the parallel copy lower doubleword (PCPYLD) instruction can be used. The example in Figure 3-15 shows this instruction combining the values in registers $8 and $9 to give the value in register $10.

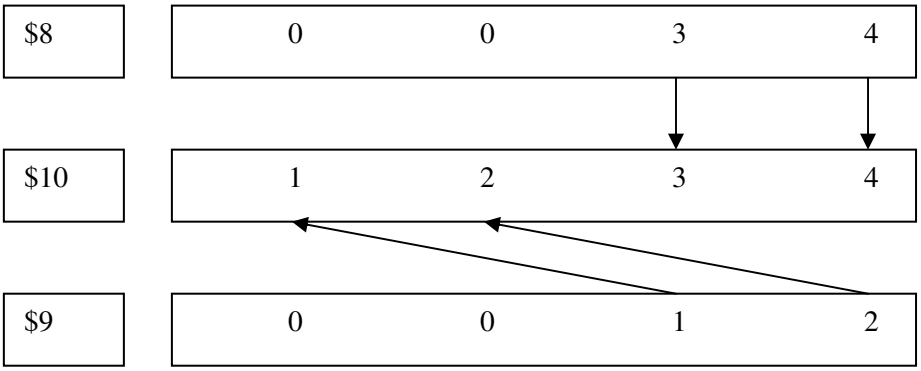


Figure 3-15

The reverse is possible when saving the 128-bit values is required. The instruction only has to move the upper 64-bits back down to the lower 64-bits of a register, and does not need to do anything with the lower 64-bits of the register. This is because they can be saved directly to memory from the position in the register they are already in.

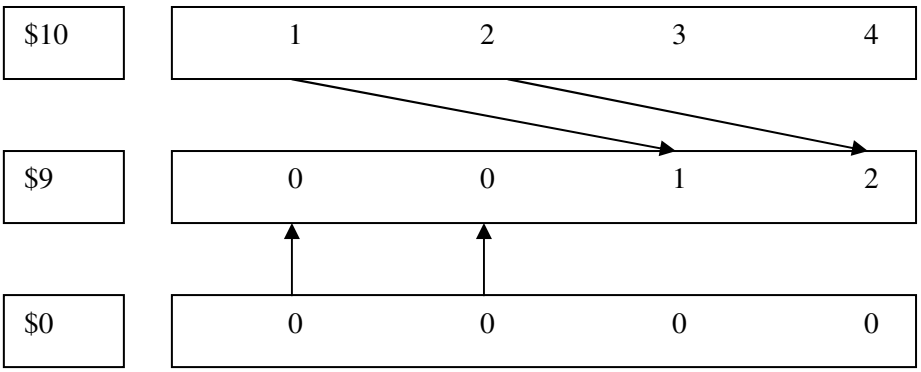


Figure 3-16

3.4 The ILCG Emotion Engine Specification

This file contains all of the instructions that are to be implemented for the port of the compiler to the Playstation 2. The file can logically be divided into a number of sections: the top of the file which has declarations of all the required types and the registers; the middle sections which contains a pattern for each instruction to implement; and an instruction set at the end of the file which lists all of the instructions in the middle section in the order in which they should be checked when matching them to the ILCG file of the Vector Pascal source file. The finished ILCG machine specification will be passed though a code-generator-generator to create a Java class of the code-generator for the Playstation 2. The class created will be a subclass of a prewritten class called Walker, and so the Playstation 2 code-generator can be known as a walker.

3.4.1 Declaring Types

ILCG supports addressing memory in chunks from 1 byte up to 16 bytes. These are declared as terminal symbols in ILCG with 1,2,4,8, and 16 byte chunks called octet, halfword, word, doubleword, and quadword respectively. The integer types have also been defined, and it is possible to use the following terminal symbols for integers: int8, uint8, int16, uint16, int32, unit32, int64, and uint64. In the Playstation 2 machine specification these have then been grouped in to signed and unsigned types for operations which work on any size, but specify and signed or unsigned constraint. Floating point types have been declared as float or double although it is possible to use ieee32 or ieee64 if a specific representation is required. It is also possible to define reference types which are pointers to data of the above types. The registers in ILCG are declared as ref types since they essentially point to a location containing a piece of data of a specific type, for example, a single precision floating point register would have type ref ieee32.

3.4.2 Declaring Registers

It is possible to define a new register in ILCG, or for a register to be defined as an alias for an existing register, for example, in the Pentium, the BP register is a portion of the EBP register. It is also possible to define sets of registers using a pattern as seen in Figure 3-17 below, which associates the name freg with the registers F0 to F5. These are useful because instructions in hardware are not declared for a specific register but instead for a set of registers.

```
pattern freg means [F0|F1|F2|F3|F4|F5];
```

Figure 3-17

3.4.3 Declaring instructions

An instruction in ILCG is just a pattern which accepts a number of arguments and specifies how those arguments can be combined. The code-generator for the target machine will take the source program in ILCG form and use these instruction patterns to match against patterns in the source program. It is possible for the instruction to accept non-terminal symbols, i.e., ones which have been defined in the machine specification. This is important as it allows different addressing modes to be defined, e.g., register to register or register to memory. The instruction can then accept a pattern that will match against either of these addressing modes, allowing a greater deal of abstraction in the ILCG code, since the single instruction can match a number of possibilities. Figure 3-18 shows the ILCG instruction for addition of two floating point numbers using the register set in Figure 3-17.

```
instruction pattern ADDS(freg r1, freg r2, freg r3)
      means[r1:= +( ^( r2), ^( r3))]
         assembles[‘add.s 'r1',' r2 ',' r3];
```

Figure 3-18

3.4.4 The Instruction Set

The instruction set is simply a list of all of the instructions that should be used to match against the ILCG source code, and the order in which to check the instructions. The order is important because there will be several possible ways to match a given ILCG sequence, but typically only one optimal way of doing so. By giving certain instructions before others, it is possible to control the assembly output of given ILCG code, although it can be difficult to ensure that optimal code is generated for more complicated cases.

A simple example of the importance of the ordering of instructions can be seen with the following example.

```
mem(a) := mem(b)
```

This example copies the piece of data from memory location b into memory location a. The optimal code for such an operation is

```
lw      reg, 0(b)
sw      reg, 0(a)
```

This code loads the 32-bit value from the memory location b into a register reg, and then stores it to the memory location a. This code would be generated if the instruction set comprised of the following

```
instructionset[ SW|LW];
```

This is because the walker will look at the SW pattern and notice that it is of the form "mem(a) := some register". This is similar to the ILCG code in that both of them have a memory location then an ":=". The SW pattern does then have a register whereas the ILCG has another memory location, but this is not a problem because the walker will then try to load the memory location into a register to make it compatible with the SW pattern. This it will try to match the pattern "some register := mem(a)". It will search from the start of the instruction set again to match this, and will fail to match this on the SW pattern because the SW pattern has a memory location on the left of the ":=" whereas the pattern to match has a register. However, it will then try to match this pattern to LW and will find it is an exact match. This will cause the LW to return that it has matched this pattern to the SW pattern which will also return since there are no more remaining unmatched pieces of code. Thus, the sequence generated will be the LW instruction then the SW instruction.

However, if just one more instruction is inserted, the MOV instruction, which moves a piece of data from one register to another, then the above code may now take three instructions even though it only needs two.

```
instructionset[ SW|MOV|LW];
```

If the instruction set is now defined as above, then the following steps will be taken when matching the same ILCG code.

1. The SW pattern will again match "mem(a) := some register" and instruct the walker to try to match "some register := mem(b)"
2. The new pattern will again fail to match SW, but will now match MOV because both start with "some register :=". Similarly to SW, it has a register on the right, and not a memory address, but will instruct the code generator to try to match "some register := mem(b)"
3. The pattern from MOV will again fail to match SW. It will match MOV again, but the code generator will notice that this match has already taken place and avoid doing it again to stop an infinite match from taking place. The pattern will then match LW.
4. The ILCG code will therefore generate a LW followed by MOV, and finally a SW.

It can be seen that the problem with the above instruction set is that the LW instruction should be matched before the MOV instruction to stop these unnecessary moves from taking place. By simply putting the MOV pattern after the LW pattern, this will be resolved and optimal code will be generated for the above example.

```
instructionset[ SW |LW| MOV];
```

What can be seen from the above example is that even a relatively trivial case can cause unnecessary code to be generated. More complicated interactions, such as those between integer and floating point registers took place in the development of the machine specification. For example, if a MOV instruction was needed by the code generator, but MOV was preceded by moves to and from the floating point registers, then instead of doing a MOV between integer registers with one instruction, a move would first be done from the source integer register to the floating point register, and then from the floating point register back to the destination integer register.

4 Implementation

4.1 The ILCG Machine Specification file

Some examples of the possible code that can appear in an ILCG machine specification were given in the Design section. These displayed the simple case of trying to move a piece of data in memory. The ILCG specification for the Playstation 2 contains instructions to perform all of the basic operations in the ILCG specification, including: addition, subtraction, multiplication, division, remainder, min, max, shifts left and right, assignments, bitwise Boolean operators, and Boolean comparisons. It also contains instructions to plant labels in assembly code for branch statements to jump to, and for implementing condition jumps, i.e., IF statements which are used for assembling IF statements in Pascal, and for testing the condition of loops to break from a loop when it has finished. In addition to these operations, the following floating point instructions have been written: addition, subtraction, multiplication, division, Boolean comparisons, assignment, conversion to and from integers, and a number of trigonometric functions.

Before all of these instructions could be defined, the registers and corresponding register sets were defined. First the 32 128-bit MIPS registers were defined. All of the other integer registers were then defined as subtypes of these registers. A scalar register set was created for all of the types given in the ILCG Design section. In addition to creating a set for each type of register, some sets were combined to create sets for instructions which could handle any integer sized argument, for example, the Boolean operator AND operates over the lower 64-bits of a MIPS register regardless of the size of the data in the registers. This does not cause any problems, because the code-generator will generate instructions to only extract the part of the register needed once the instruction has completed, for example, it will use SW if it required on the lower 32-bits of a register. Some patterns were also declared which were up of a number of types.

The 32 floating point registers were then defined, as well as the floating point control register, although it is not used in any instructions. Aliases of these floating point registers were needed which can store 32-bit integers. This is necessary because the floating point to integer convert instruction saves its result to a floating point register, but the code generator would be unable to move this result from the floating point register as it would believe that the register was only floating point. By defining the integer alias registers, the type of a floating point register can then be an integer, and the move instruction will be successful. This is somewhat

dangerous because the code generator may then start to load integers into floating point registers and perform floating point arithmetic on them, in the belief that the arithmetic is actually for integers. Careful use of type casts in the floating point and integer instructions has removed this possibility.

Once all of the registers were defined, the addressing modes of the Emotion Engine had to be defined. These addressing modes are using by the code generator to allow instructions to accept data from registers as well as memory, and the code generator can insert loads and saves to registers from memory where arguments must be loaded or saved to and from memory. The MIPS design does not allow operands for instructions to be present in memory, i.e., both must be in registers, or one can be an immediate value, and so the addressing modes are very simple of the Emotion Engine. They are:

- Offsets from labels
- Offsets either up or down from registers
- The contents of a register
- Offsets from the global pointer

With all of the register, types, and memory addressing modes decaled, it was now possible to declare instruction patterns. Typically, a pattern is needed for each individual instruction, but there are cases where many instructions can be generated from a single pattern. One of these cases is the dyadic floating point operations, e.g., addition, subtraction, multiplication, division. It is possible to declare an operation in ILCG which associates a label in ILCG with a given ILCG operation, and generate a given symbol when that label is placed in an instruction patterns assembly output. An operation was declared in the machine specification for each of the common ILGG operations. Patterns were then generated for those which have something in common, for example, the dyadics mentioned previously, or the MIN and MAX instructions. An instruction, for example, RRRieee32 in Figure 4-1, can then take as one of its arguments an operations pattern. In this example, the code generator could then match any instruction of the form “register: = register ‘op’ register”, as long as the registers are floating point. The equivalent integer instruction looks very similar but uses integer registers instead of the floating point registers.

```
pattern arithoperator means[add|sub|divide|mul];
instruction pattern RRRieee32( arithoperator op, freg r1, freg r2, freg r3)
      means[r1:= op( ^( r2), ^( r3))]
         assembles[op '.s 'r1',' r2 ',' r3];
```

Figure 4-1

4.1.1 Implementing floating point math operations

There are a number of mathematical operations that Vector Pascal allows on floating point numbers. There are: absolute value, cosine, sine, tangent, natural logarithm, and square root. There are instructions for both absolute value and square root, so these can be translated directly into a single assembly language instruction. However, there is a C library in PS2 Linux for executing all of these operations. The C file in Figure 4-2 was compiled to see how the C compiler implements the required operations. The assembler generated revealed that the compiler first converted a floating point number to a double precision floating point number, then called the function to accept a double precision number, and then convert the result back to a floating point number. The assembly generated for the sine operation is given in Figure 4-5. This code can almost be copied directly from the assembly output to the ILCG machine specification.

```
int main()
{
  float a,b;
  b = 0.5f;
  a = sin(b);
  a = cos(b);
  a = tan(b);
  a = log(b);
  return 0;
}
```

Figure 4-2

```
l.s    $f12,36($fp)
jal    fptodp
move   $4,$2
move   $5,$3
jal    sin
move   $4,$2
move   $5,$3
jal    dptofp
s.s    $f0,32($fp)
```

Figure 4-3

4.1.2 Implementing double precision instructions

The double precision instructions that have been implemented are the basic operations found in Figure 3-11 as well as the floating point math functions. As there are no instructions for double precision numbers, the square root and absolute operations also had to be implemented as function calls, whereas for single precision numbers they could use the appropriate instructions. The double precision operations were therefore defined in a very similar way to the sine function in Figure 4-5, but instead of translating from a single to double precision number and back again, instructions were needed before the function call to translate from using one register to represent a double precision number in the machine specification, to using two register in C procedure calls.

Figure 4-4 shows the assembly generated for the instruction to set register $16 to be the sum of registers $17 and $18.

```
l.s    $f12,36($fp)
jal    fptodp
move   $4,$2
move   $5,$3
jal    sin
move   $4,$2
move   $5,$3
jal    dptofp
s.s    $f0,32($fp)
```

Figure 4-4

4.1.3 Implementing the multimedia instructions

The difficult part of using the multimedia instructions on the Playstation 2 is in designing how to load and store 128-bits which are not guaranteed to be aligned properly in memory. This issue was covered in the design section. Assuming the sequence to generate for the load and store instructions is known, using the multimedia instructions is not much harder than using the general MIPS instructions. They are declared in much the same way, but use vector typed registers instead of the normal typed registers, and they must cast their arguments to their type to prevent interference with other vector instructions, or the normal MIPS functions.

4.1.4 Optimizing the ILCG machine specification

Throughout development, many test programs have been written and the generated assembly code observed by eye. It is relatively easy to see instructions that are unnecessary, or simply the wrong way of performing a certain operation. The example given earlier in the ILCG design section is one, where the instructions were given in the wrong order, and so a MOV instruction was inserted into the assembly code when it was not necessary. Figure 4-5 shows another example which copies the value of register $2 to $3, but does so via the floating point registers. The optimal code for this sequence is simply “move $3, $2”.

```
MTC1   $2, $f20        // moves $2 into floating point register $f20
MFC1   $3, $f20        // moves from $f20 into $3
```

Figure 4-5

The example given above would be generated if the move to coprocessor 1 (MTC1) instruction was ahead of the MOV instruction in the instruction ordering. It has the disadvantage of not only using two instructions, but also takes up a floating point register unnecessarily. If all of the floating point registers were in use when the sequence was generated then one of the floating point registers would have had to be saved to the stack, and reloaded after the move instructions had completed. This would add a load and store to the instructions executed, and they would also take time to access memory.

4.1.5 Multimedia optimisations

The multimedia instructions in the Emotion Engine have been designed to not only operate efficiently on vectors of data, but also to move them around in ways necessary to multimedia applications. For example, there are instructions to move fields around within the register in a way which could, for example, keep the W field in a graphics vector in the same position, while swapping the other fields from X, Y, Z, to Z, Y, X ordering. A more useful application in terms of the Vector Pascal language is to be able to load a single value into a register, and have that value copied to all of the fields of a vector. This is a way to vectorise the operation

where a value if copied to all of the positions of an array in Vector Pascal. This would currently generate a loop to assign the value to each position in the array, but it is possible to use the multimedia operators to allow vectors instructions to reduce the number of loop iterations required. This can be seen in Figure 4-7 where an integer variable is loaded from memory, copied into the vector, and then the vector can be saved back to memory if necessary, but with all of its fields now assigned to the variables value, which is assumed to be 1 here.

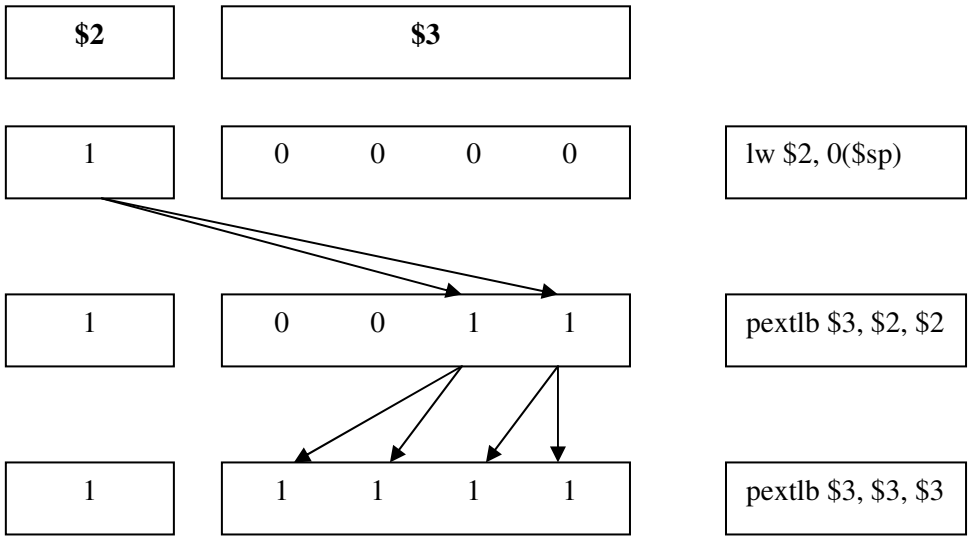


Figure 4-6

This example demonstrates how it is possible to further avoid loops, and increase vectorisation of code, even if one of the pieces of data is a scalar variable. This example may not be significantly faster than a loop once, but an equivalent using 16-bit or even 8-bit numbers could be much faster than the normal approach. It should be noted that this is currently not possible in Vector Pascal, as the code generator cannot vectorise for code with a vector on one side and a scalar on the other, but such a feature may be implemented in a future release.

4.2 The EECG Walker subclass

The EECG.java file contains the Java class for the Walker for the Playstation 2. The important tasks defined in this file are:

- Implementing procedure call mechanisms from both C and Pascal
- Overriding getParallelism to define the parallelism available on the Emotion Engine
- Implementing the assembly and linker methods

The details of the procedure call mechanisms have been covered earlier, where it was mentioned that it would be possible for this compiler to guarantee that the stack is never moved during execution of a procedures body of code. When compiling a Vector Pascal program, the compiler will call the EECG method for procedure call, which will output some code that is standard to all procedures. It will then instruct the code generator to generate the code for the body of the procedure, before outputting some more standard code to restore registers, clean up the stack, and exit the procedure. After the body of code has been generated, one of the standard codes placed in the output is an assembly constant telling the compiler how much space is needed by the procedure. A piece of assembly code generated at the start of procedure call can use this constant to make enough space for the procedure on the stack. Whenever a procedure is called from within the current procedure, variables will be pushed onto the stack, thus moving the stack frame. However, in the Emotion Engine, due to the 128-bit register width, the stack frame must remain a multiple of 16 at all times due to alignment restrictions. This is not an issue when calling other Vector Pascal procedures, since they assume an alignment of only 4, but it must be ensured when calling C procedures as it is impossible to know whether they will execute correctly if they assume the stack pointer is a multiple of 16 and it is not.

In order to fulfil this alignment requirement, all of the places in the code where the stack would have been manipulated were replaced with code to add to counters for how much the stack pointer should be offset by. After the procedure has called the code generator on its body of code, the maximum amount of stack space required to call the procedure with the most arguments will be known, and this can be added onto the stack space already needed by the procedure. A calculation can then be performed to round the stack up to the next multiple of 16. Since GCC will compile the runtime library, and will give the stack in main a value which is a multiple of 16, the stack will have a multiple of 16 before any Pascal code is called. By ensuring that the stack is always subtracted by multiples of 16, the Vector Pascal compiler can guarantee that it will always be a multiple of 16 when calling other procedures.

It was the necessary to find all the locations in the code where a variable would have pushed to the stack. This would typically entail subtracting 4 from the stack, and then saving the variable to “0($sp)”, i.e., an offset of zero from the stack pointer. However, the stack pointer cannot be manipulated in this way. There are two solutions to this problem. The first is that there are some variables pushed before the stack pointer is changed and so they will know their offsets from the old stack pointer. For example, Figure 4-7 shows the code generated n

the original code by the first two pushes onto the stack, which occurs before the stack would have been changed to subtract the space necessary for the procedure.

```
addi $sp, $sp, -4
sw $31, 0($sp)
addi $sp, $sp, -4
sw $fp, 0($fp)
```

Figure 4-7

Since it is known that these are always the first two variables to be saved to the stack, their offsets from the old stack pointer will always be -4 and -8. It is therefore possible to replace this with the code from Figure 4-8. Note that another advantage of getting the compiler to calculate all of the offsets is that the stack will now only be manipulated once for each procedure call, and not once for each variable pushed in each procedure call. This will result is far fewer instructions being executed when calling some procedures.

```
sw $31, -4($sp)
sw $fp, -8($fp)
```

Figure 4-8

The other method to save variables is to generate new instances of the ILCG Variable class, which will automatically be allocated space by the current procedures local store allocator and will also automatically be given an offset from the frame pointer. A small ILCG tree can then be generated which creates a statement to assign the register which needs to be saved to memory. The code generator will then generate the necessary instructions to save the variable. A reverse of this statement can be used at a later point to restore the variable from the same location in memory back to its register. This is the method used to save registers, whereby an array of all the registers to save can be created. A loop can then cycle through the registers to save, create a variable for each of them, and attach a statement to the tree to save and restore the register at the appropriate points. Since all the statements to save the registers can be part of a larger statement, it is possible only call the code generator once, and not once for each register.

This mechanism for saving registers can also be used to reserve space for the global pointer. This must be saved using the cprestore pseudo instruction as explained in the section on the Global Offset Table. This will not be saved by an explicitly declared store instruction as are the other registers, as the cprestore instruction will save it to the offset it is given. However, it is necessary to calculate the offset for the memory location global pointer will occupy. A

variable must be declared for the global pointer even though it will not be used by the code generator to save to the stack. However, creating a variable for the global pointer will have the effect of causing the procedure to allocate space for the global pointer. The offset from the frame pointer to this space can then be found by accessing the offset field of the variable instance. A was seen previously, this offset cannot be negative, which it is from the frame pointer. However, at the point in the code where the cprestore is generated, the stack will have been subtracted by the amount necessary for the procedure. By simply adding the offset to the reserved space, the positive offset value from the stack pointer will be generated. This value can then be passed to cprestore to cause it to save and restore the global pointer from the correct location in memory.

4.2.1 Parallelism

The parallelism for the Playstation 2 can be found from the instruction set, for example, there are instructions to operate on 16 8-bit numbers at a time. The parallelism returned from the getParallelism method is given in Figure 4-9. Types not in the list have a parallelism of 1 returned.

Length of Type	Parallelism
8-bit integer	18
16-bit integer	8
32-bit integer	4
32-bit float	4

Figure 4-9

4.3 Executing the compiler

The Vector Pascal compiler cannot be executed on the Playstation 2. This is because the Playstation 2 does not have a Java run-time environment, which is an essential piece of software the platform must provide for the compiler to run. This can be solved by creating a cross compiler environment in which the Vector Pascal compiler in run on a machine with Java installed, in this case a Linux machine, and the compiler output can be transferred to the Playstation 2 and compiled and run there. It would be possible to generate the assembly file for a Vector Pascal program and pass this to the Playstation 2 to be assembled. However, the include statements in the file would contain addresses of folders on the Linux machine, and not on the Playstation 2. The solution to this is to assemble the file on the Linux machine. It is possible to get the source code for the binary utilities, including the assembler, from Disc 2 of the Linux distribution for the Playstation 2. This source can then be compiled on the Linux

PC, thus allowing MIPS assembly files to be assembled on a machine other than the Playstation 2.

Once the object file has been created, it only has to be linked with the C runtime library, and it will then be executable on the Playstation 2. However, it was not possible to compile the GCC compiler on the Linux PC, as it required the PS2 Linux standard C libraries which were only on the Playstation 2 hardware. These could have been transferred to the Linux PC, but due to the large amount of disc space they require, a different solution as necessary. Had the compiler been available on the Linux PC, the executable fie would still have had to be transferred to the Playstation 2 to be executed. It is possible to instead transfer the object file output from the assembler. Although this has not been linked with the run-time library, the run-time library C file can simply be transferred to the Playstation 2, and then both it and the assembly output can be linked natively. It is then possible to execute the linker output on the Playstation 2. Appendix A includes a Perl script used to execute the conformance tests. This script shows the different parts of the build process and the commands necessary to automate the compiling process.

4.3.1 Running the assembler

The assembler for the Playstation 2 is distributed with a number of patches to make it work with the Playstation 2 instruction set. However, the default options for the assembler do not build for the Playstation 2, but instead for standard MIPS processors. This causes the assembler to fail, and a number of errors messages displayed saying that instructions which are known to be present in the hardware are not supported by the assembler. This can be overcome by passing the "-m5900" argument to the assembler. This tells the assembler to use the Playstation 2 instruction set, as the Playstation 2's processor is known as the R5900.

4.3.2 Running the C compiler

The GCC compiler on the Playstation 2 is used to compile the run time library, and link it with the object file generated by the assembler. The GCC program builds for the R5900 by default and so the argument passed to the compiler is not necessary here. However, both the floating point math operations, and all of the double precision operations use the standard C math library. The compiler will generate error messages if this is not linked with the object file, as there will be undefined references to math functions implemented in the library. By passing the "-lm" argument to the GCC program, the math library will be included by the linker.

The other argument which must be passed to the compiler is "-DEECG" which defines the symbol "EECG" in the run time library C file. This is necessary because the run-time library contains stringcompare and stringassign methods which are by default commented out. These methods are instead implemented in a macro file containing a hand coded assembly implementation. However, the macro file is written for the Pentium, and so is not compatible with the Playstation 2. The solution to this was to uncomment the implementations of these procedures in the run time library, but surround them with "#ifdef EECG" codes. This will cause them to only be compiled into the run-time library when EECG in not defined, i.e., on all the platforms other than the Playstation 2.

4.4 Changes made to the Vector Pascal Compiler

During development there were a number of issues raised when writing the machine specification. One of these was the limit of 64 registers in a specification. This limit existed because the compiler used a 64-bit integer in Java to store a bit for every register in use. This was used to store the registers in use before sequences of instructions were generated. The compiler could then check future sequences and if the sequence was the same and the registers in use were the same, then the compiler would not have to call the code generator for the sequence, and could just output the from the saved sequence.

This limit of 64 registers was easily exceeded by the Playstation 2 specification as there were 32 integer registers and 32 floating point registers. Once the HI and LO, control, and parallel registers were added as well, the specification had close to 100 registers. The 64-bit number in the compiler was replaced with an instance of the Java BitSet class, which, as before, allowed for a bit to be set for every register in use. When the compiler needed to know what registers were in use, the BitSet was turned into a string, and returned to the compiler.

4.5 Optimizing operations

The following sections discuss some of the possible ways of implementing various operations in the MIPS hardware, while trying to use the fewest instructions possible. These include ways of implementing all of the comparison operators for all the types supported by the Vector Pascal language. The following example shows the usage of the MIN operator in Vector Pascal and presents a number of ways of implementing this instruction in hardware.

```
PROGRAM minimum;
VAR
        a, b, c : integer;
BEGIN
        a := b min c;
END.
```

Figure 4-10

On many processors, Figure 4-10 would have to be translated into a conditional statement which will include jump statements when written in assembly language. Jump statements cause processor execution to halt while it loads the new location to start processing from, and since a processor cannot do any useful work while halted, it is an expensive operation. Figure 4-11 presents a number of different ways to encode the MIN operator using the Playstation 2 instruction set. Assume that the identifiers A, B, and C are the registers containing the respective variables.

Conditional with jumps	Conditional move	Built in MIN instruction

slt temp, B, C
beq $0, temp, else_label
move A, B
j end_label
else_label:
move A, C
end_label:
	slt temp, B, C
	movn A, B, temp
	movz A, C, temp
		PMINW A, B, C

Figure 4-11

The conditional jumps algorithm above implements the minimum operator by ensuring that only one of the move instructions can ever be executed each time that code is run. It does this by first comparing the B and C values and setting a temporary register to be one if B is less than C, or zero otherwise. The beq instruction then branches to the else code if the temporary values was zero, i.e., C was smaller than B, and the else code sets A to be C. If the temporary value was one, then A is set to B, and a jump is made to the end of the conditional. The jump is necessary because the processor executes instructions in the order they appear in the code, and if the jump to the end is not made, then the else branch would be executed, thus setting A to be C no matter what the values of B and C are.

The conditional move code relies on two instructions which will only move a value if their condition is set. The code starts the same as the conditional jump code, by first setting a temporary value depending on whether B or C is smaller. The two remaining lines of assembly are functionally the same as the conditional jump ode, but execute without the need for jump instructions. The jumps are not necessary because only one of the instructions can meet the condition, so only one move will ever take place. The movn instruction will only move B into A if the temporary value is not zero, which is correct since the temporary value will be one if B is less than C and should be stored in A. The movz instruction is the opposite in that if moves C into A if the temporary value is zero.

The final code uses a MIN instruction which is built into the processor, and should therefore be the fastest of all the methods available for calculating the minimum value. It is possible to use the values for the throughput and latency of each of the instructions in the three versions of code to work out which is fastest, and by what amount.

The latency of an instruction is the amount of time it takes for that instruction to execute (Comp Design and Org). It can be thought of as a good measure for the theoretical performance of a piece of code, but run-time factors like memory access times could make the code execute much slower. The latency for all of the instructions used in the above examples is 1. This means that the fastest code will be the one that executes the fewest instructions, i.e., the code using the built in PMINW instruction. The latency for each version of code is therefore 4, 3, and 1 processor cycles. However, the code using conditional jumps will cause the processor pipeline to halt, and since the pipeline is 6 cycles long it incurs another 6 cycles of execution time, thus making it take at least 10 cycles. It can therefore be seen that the conditional jumps version is around three times slower than the conditional moves version, which is itself 3 times slower than using the built in instruction. The built in instruction also has the advantage of being able to do four MIN operations at once, and so its theoretical speed advantage if used across vectors of integers would be another four times greater than it already is in this example.

This is a somewhat fabricated example of a possible speed up given the addition of a built in minimum instruction to the Playstation 2 processor. A more important point it displays is the advantage of trying to avoid using jumps as much as possible. It is generally not possible to avoid using jumps in normal code, but as can be seen from the conditional moves way of implementing the minimum operator, it is sometimes possible to avoid jumps when writing assembly for specific operations.

4.6 Comparison Operators

Symbol	Description
A <> B	True if A is not equal to B
A = B	True if A is equal to B
A > B	True if A is greater than B
A < B	True if A is less than B
A >= B	True if A is greater than or equal to B
A <= B	True if A is less than or equal to B

Figure 4-12

Figure 4-12 lists all of the Boolean comparison operators. The Playstation 2 contains an instruction to compare the values in two registers called "Set on Less Than" (SLT). The SLT instruction compares scalars of 64-bits in size so it is possible to implement all of the above comparison operators for numbers up to 64-bits with the SLT instruction. However, the instruction returns a 1 if a comparison is true, and a 0 if it is false, but the compiler uses the values -1 (all 1's in binary) for true, and 0 for false. This means that a true represented by a 1 must be mapped onto a true represented by a -1. This would be possible using a conditional statement, similar to the example of implementing MIN given earlier. However, as with MIN, it would be preferable to do this without using a conditional since that would cause the processor to jump, and thus result in an expensive stall of the pipeline. The difficulty in implementing these operations without a conditional is that a 1 must be mapped to a -1, but without changing the 0, since false is represented by a 0 in both the processor and the compiler. This means that a simple operation like taking the result and subtracting 2 from it, which would be a valid way to turn 1 to -1, will not work, since that same operation would change the value of false from 0. This is further complicated because the only instruction that can do a comparison on the Playstation 2 can only check if the value in one register is less than the value in another register, but comparisons are also needed for equality, inequality, and values being greater than one another.

4.6.1 Less than (<)

This is the simplest of all the operators to translate into a sequence of assembly language instructions since all that has to be done is to take the result of the SLT instruction and map the true value of 1 to a -1. It was necessary to find a way to map a 1 to a -1, while leaving the value of 0 unchanged. Instead of looking at this as leaving the 0 unchanged, it can instead be seen as having 2 mappings, which both must be achieved using the same function.

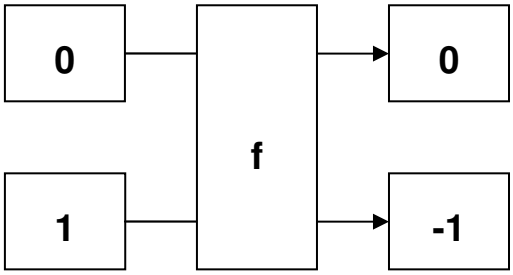


Figure 4-13

In this case the mapping is quite simple, since simply negating the left hand side will give the right hand side. In fact, it will be seen in the other examples that this mapping applies to all cases where we want to map the true and false values in the Playstation 2 directly to those in the code. There are cases where we need to map false to true, and true to false, which will be given later. Now that the mapping had been found, all that remains is to implement it is assembly code. Unfortunately there is no negation instruction in the processor, however, the first register ($0) always contains 0 and subtracting from 0 is the same as negation. Assuming that we want a register B to be the result of X < Y, the assembly necessary to achieve this is given below.

```
SLT B, X, Y
SUB B, $0, B
```

Given this simple code, it is possible to formally assess its correctness, since either X is less than Y before the comparison, or it is equal to or greater than Y. Figure 4-14 shows that given any values in the X and Y registers, the value in the B register will be correct after the end of the above sequence of instructions.

X < Y	Value in B after SLT	Value after SUB	Result
True	0001	1111	True
False	0000	0000	False

Figure 4-14

4.6.2 Greater than (>)

The greater than operation is simply a rearrangement of the less than operation, and so if we wanted the result of X > Y we can simply use the less than operation but instead of checking for X < Y, we check for Y < X.

4.6.3 Equal to (=)

Since there is no instruction for comparing equality, we have to look at the possible values for X and Y, and find a way to implement equality using only the SLT instruction.

X < Y	X > Y	X = Y
False	True	False
True	False	False
False	False	True

Figure 4-15

As can be seen from Figure 4-15, there are only three possible cases for the results of X < Y, or X > Y to take. They can either individually each be true, or both can be false. It is mathematically impossible for both to be true simultaneously. If X is not greater than Y then it must be less than or equal to Y, and similarly if Y is not greater than X then it must be less than or equal to X. If both of these conditions hold at the same time then it must be the case that X and Y are equal. In this case, if we take the results of X < Y and X > Y and logically OR them together, we will get false values where we need true, and true results where we need false. It is then simply a case of finding a new mapping which can map false to true and true to false.

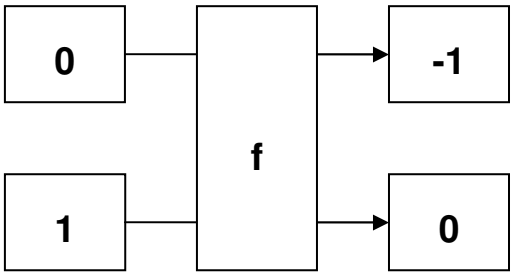


Figure 4-16

This mapping can be implemented by subtracting 1 from the left hand side to get the required result on the right hand side.

Returning to the equality example, we can now get to a state where we have a Boolean value which is the logical OR of two less than comparisons. This can be implemented directly in

assembly language, and when appended with the above mapping, will result in the Boolean value containing the correct result. Note, there is no subtract immediate instruction in the processor, but an add immediate with a negative value achieves the same result. The assembly code to implement B = (X = Y) is given below. The temp1 and temp2 values are used to show that temporary registers are needed here to store parts of the result. It would be possible to use the B register instead of one of the temporary registers.

```
SLT temp1, X, Y
SLT temp2, Y, X
OR B, temp1, temp2
ADDI B, B, -1
```

Figure 4-17

4.6.4 Not equal to (<>)

As the name suggests, this is just the negation of the equality operator. It would be possible to take result of the equality operator, negate it, and have a valid result. However, this is inefficient since the equality operator used the second mapping to map true to false and false to true. Negating the result of the equality operator is logically the same as using the first mapping instead of the second since second does essentially negate the result whereas the first does not. The assembly in Figure 4-18 below implements B = (X <> Y).

```
SLT temp1, X, Y
SLT temp2, Y, X
OR B, temp1, temp2
SUB B, $0, B
```

Figure 4-18

4.6.5 Less than or equal to (<=)

Similar to the inequality operator, this operator can be implemented using the negation of one of the other operators, in this case, the greater than operator. Since the greater than operator uses the first mapping to achieve its result, we can implement less than or equal to using the same code, but with the second mapping instead of the first.

```
SLT B, X, Y
ADDI B, B, -1
```

Figure 4-19

4.6.6 Greater than or equal to (>=)

Similar to less than or equal to, this is just the negation of less than and is given by the following code.

```
SLT B, X, Y
ADDI B, B, -1
```

Figure 4-20

4.7 Floating point comparison operators

As with scalar numbers, we also need to perform the same comparisons on floating point numbers. Instead of having a single instruction available for comparisons, which was all that was available for scalars, we now have three instructions available to use:

Instruction	Description
C.EQ.S X, Y	1 if X = Y, 0 otherwise
C.LE.S X, Y	1 if X <= Y, 0 otherwise
C.LT.S X, Y	1 if X < Y, 0 otherwise

Figure 4-21

The difficulty of performing floating point comparisons no longer lies with having to find ways of doing all the operators with only the one instruction, but instead with accessing the result of the instruction without branching. This is because the floating point comparison instructions do not set a general purpose register with their result, but instead set a single bit – the C bit - in a floating point control register to either one or zero depending on the result of the comparison. There are instructions which can branch depending on whether the value in that control bit is a zero or a one, but it would be better to avoid using these.

Bits 31 - 24	C	Bits 22-0		
xxxxxxxx	1 or 0	xxxxxxx	xxxxxxxx	xxxxxxxx

Figure 4-22

As Figure 4-22 shows, the C bit is the 23rd bit in the control register (PS2 manual). The purpose of the other bits is not important at this time. There is an instruction – CFC1 - which can transfer the whole control register to a general purpose register. With the whole control register now in a general purpose register, we would now like to have just a one or a zero in the least significant bit of the register, and zeros in all the other bits, i.e., we want the register to represent the scalar values of zero or one. This can be achieved by first shifting the register right by 23 places, thus moving the C bit to the least significant bit position, and then masking out the other bits to zero so that only the value of the least significant bit remains. This masking can be achieved by using the logical AND instruction with a one in each bit position

where we wish the bit in the source register to remain in the destination register, which in this case is just the least significant bit position, thus making the mask the scalar value one. Once the AND instruction has completed, we will have a one or a zero in the general purpose register, and can simply use the first mask from the scalar comparisons to map the scalar 1 to -1, and 0 to 0.

4.7.1 Less than, equal to, or less than equal to

These there operations all have corresponding instructions to do the comparison directly. When the comparison instruction and instructions to extract the C bit have executed, we will have a one or a zero, representing true and false respectively. These instructions then all use the first map to get the resultant truth value the compiler expects. The assembly code for the operations B = (X * Y) is given below, where * can be replaced by EQ, LT, or LE for equality, less than or equal to, or less than comparisons respectively. Note that the $31 is the assembly code for the floating point control register.

```
C.*.S X, Y
CFC1 B, $31
SRL B, B, 23
ANDI B, B, 1
SUB B, $0, B
```

Figure 4-23

Although this sequence of instructions appears quite long considering it must be generated for every floating point comparison, it would still execute quicker than the corresponding code using the branch instructions since they would need 6 cycles just for the branch, plus a cycle for each additional instruction. Figure 4-24 shows the contents of the B register after each instruction assuming that the C bit is one, i.e., the comparison is true.

Instruction	Value in B register
CFC1	xxxxxxxx 1xxxxxxx xxxxxxxx xxxxxxxx
SRL	00000000 00000000 0000000x xxxxxxx1
ANDI	00000000 00000000 00000000 00000001
SUB	11111111 11111111 11111111 11111111

Figure 4-24

This demonstrates that if the comparison is true, then the result will be a register full of 1's. The corresponding table for a false comparison is given in Figure 4-25.

Instruction	Value in B register
CFC1	xxxxxxxx 0xxxxxxx xxxxxxxx xxxxxxxx
SRL	00000000 00000000 0000000x xxxxxxx0

ANDI	00000000 00000000 00000000 00000000
SUB	00000000 00000000 00000000 00000000

Figure 4-25

Although this table is mostly full of zero's and appears to show little about what the compiler does for floating point comparisons, it is important because it demonstrates that the compiler can generate correct true and false values using exactly the same sequence of instructions.

4.7.2 Greater than, not equal to, or greater than and equal to

These three operators are all negations of the other three operators, for example, (X >= Y) is the same as NOT (X < Y). This means that the assembly code from above can be used, but with just one modification, which is that we must now negate the result. This is because a result of true for (X < Y) would correspond to a value of false for (X >= Y). Looking back to the scalar comparisons, we had a similar situation there, and it was solved by using the second mapping for truth values, instead of the first. The assembly code for the remaining three comparisons is given in Figure 4-26 below, where * can again be replaced by EQ, LT, or LE, but now for inequality, greater than or equal to, and greater than respectively.

```
C.*.S X, Y
CFC1 B, $31
SRL B, B, 23
ANDI B, B, 1
ADDI B, B, -1
```

Figure 4-26

4.8 Double precision floating point comparisons

In the design section, we saw that the Playstation 2 has no double precision floating point unit, and so all operations on these numbers must be emulated. One of the functions that have been written supports comparing doubles and is called dpcmp. It is passed two double precision numbers and will return either a -1, 0 or 1 depending on if the first number is respectively less than, equal to, or greater than the second. This function can therefore be used to implement all of the comparison operators on double precision numbers.

The first stage in using this function was to realise that it can always return one of three results. The results must be grouped together with two of the results in one group, and the remaining result in another group. Each group corresponds to either the true, or false values for the operator that is being implemented, and we must find a sequence of instructions that will eventually give us the required true or false values given these possible outputs.

4.8.1 Less than (<)

To implement X < Y, the dpcmp function is called with the arguments X and Y and will return one of the three following outcomes:

1. if X < Y, then a -1 will be returned,
2. if X = Y then a 0 will be returned and,
3. if X > Y then a 1 will be returned.

It is only the case where -1 is returned that the result should be true and so this shall make up the first group. The second group is made up of the cases where 0 or 1 is returned. A sequence of instructions must now be found which will eventually give us a -1 for all numbers in the first group, and a 0 for all numbers in the second group. This can again be seen as a mapping from a set of inputs, to their corresponding outputs. The mapping for this operator is given below. The mappings for all the other operators are very similar and shall not be given.

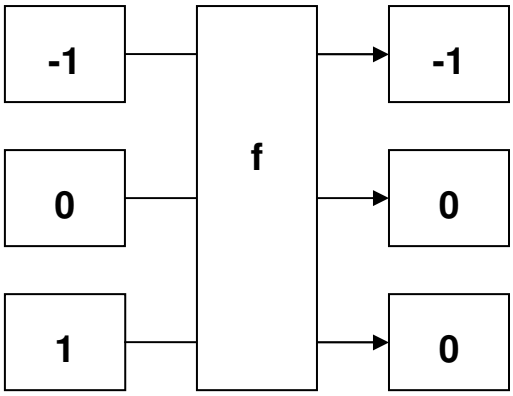


Figure 4-27

It is possible to implement this mapping using a single instruction – shift right arithmetic (SRA). This shifts the lower 32 bits of the register right by a given number of places, inserting the sign bit (bit 31) into the rightmost positions of the resultant number. This will suitably implement the above mapping because the -1 is a register full of 1's which will remain so after the shift, and similarly a register full of zeros will also be full of zeros after the shift. Only a register containing the integer value one would be affected since the one would be lost when shifted right, and the sign bit is zero, so only zeros would fill the rightmost positions of the resultant register. This would cause the one to be turned into a zero as needed. This can be seen in the following table which demonstrates this mapping on 4 bit numbers.

Value	Bit representation	Sign bit	After shift right by 1	Result
-1	1111	1	1111	-1
0	0000	0	0000	0
1	0001	0	0000	0

Figure 4-28

This mapping has successfully grouped the two false values together, while leaving the single true value in its own group. It has also generated the final truth values which were needed and so no further instructions are necessary here to get those final values. It will be seen later that some of the other operators require more instructions to get the final truth values from their groups.

4.8.2 Equal to (=)

For the equality operator, only a result of zero from dpcmp should generate a true result. The results of -1 and 1 should both give false. Both 1 and -1 can be distinguished from 0 by their value of 1 in the least significant bit position which can be used to group them together. By using the logical AND of the result and 1, the other bit positions are masked out and only the least significant bit position remains. This causes both -1 and 1 to become 1, while 0 remains 0. The final truth values are then found by subtracting 1 from all the results.

dpcmp result	Bit representation	After AND 1	After SUB of 1	Result
-1	1111	0001	0000	0
0	0000	0000	1111	-1
1	0001	0001	0000	0

Figure 4-29

4.8.3 Greater than (>)

The dpcmp function will return a 1 if the first argument is greater than the second. It is this case that should generate a true result, whereas -1 and 0 should generate false. Unlike the "less than" case, there is no simple similarity between the -1 and 0 results. However, if we subtract 1 from all the results, then the -1 becomes -2, 0 becomes -1 and 1 becomes 0. There is now a similarity between the two entries to be grouped together since they both contains 1's. Shifting right by one then causes both the false results to be -1, and the true result to be 0. These are the exact opposite of what is required, but using the NOT operator, the correct results are then given.

dpcmp	Bit representation	After SUB of 1	Shift right by 1	NOT	Result
-1	1111	1110	1111	0000	0
0	0000	1111	1111	0000	0
1	0001	0000	0000	1111	-1

Figure 4-30

4.8.4 Less than or equal to (<=)

In the last case, when dealing with the greater than operator, the results had to be inverted because the true values were false, and the false values were really true. The less than or equal to operator can instead be written as the NOT of the greater than operator. This means that by taking the sequence of operations to get the greater than result and adding another NOT to the end of it, the correct results for less than or equal to will be generated. However, there is already a NOT at the end of the sequence and applying the NOT operator twice has the same effect as not applying it at all. Therefore, the sequence for this operator is simply the same sequence as for the previous operator, but without the application of NOT at the end.

dpcmp	Bit representation	After SUB of 1	After Shift right by 1	Result
-1	1111	1110	1111	0
0	0000	1111	1111	0
1	0001	0000	0000	-1

Figure 4-31

4.8.5 Greater than or equal to (>=)

Similar to the last case, this operator can be written as the NOT of the less than operator. In the less than case, to group the results together, a simple shift right by 1 was used which gave the correct results. The sequence for this operator can be generated by taking the shift right by 1, and applying the NOT operator afterwards.

Value	Bit representation	After shift right by 1	NOT	Result
-1	1111	1111	0000	0
0	0000	0000	1111	-1
1	0001	0000	1111	-1

Figure 4-32

4.8.6 Not equal to (<>)

It is also possible to encode this operator by applying a NOT to the result of the equality operators sequence of instructions. However, this would be inefficient as it is possible to encode inequality with the same number of instructions as equality. The first stage of equality used a mask to zero out all but the least significant bits of each result. This successfully grouped the results together, into those which should be true and false, with values 0 and 1 respectively. These were not the correct truth values, but the application of one of the mappings from the scalar comparisons resulted in the correct truth values. In the section earlier on scalar comparisons, it was noted that to get the opposite truth values, the other mapping could be applied instead. In this case, the first mapping was the subtraction of 1, however, if instead the results are subtracted from 0, the correct truth values will be output. It is this substitution of mappings that makes it possible to perform inequality on double precision numbers in the same number of instructions as equality.

dpcmp result	Bit representation	After AND 1	After SUB from 0	Result
-1	1111	0001	1111	-1
0	0000	0000	0000	0
1	0001	0001	1111	-1

Figure 4-33

5 Testing

There were numerous tests run during the course of this project, including:

- Writing small C programs to see the structure of the assembly generated. This revealed the need for the cpload and cprestore instructions which save and restore the global pointer.
- Writing C program to see how parameters are passed.
- Writing Vector Pascal programs and manually looking at the assembly output. This was done to ensure that the assembly was valid, and was correct. It also illustrated cases where valid, but useless instructions were bring generated, which led to instructions being changed and reordered to remove as many of the useless instructions as possible.
- Testing running programs on the Playstation 2, and if the program generated a segmentation fault, using the gdb debugger to analyse the code and fix the problem it encountered.
- Compiling all 221 Pascal conformance tests using the Playstation 2 specification in Vector Pascal.
- Altering the machine specification if any programs failed to compile for the Emotion Engine, but compiled successfully for the Pentium.
- Running all of the conformance tests which compiled.

The most significant results from testing are those generated by the ISO conformance tests. Since there are 221 tests, it is infeasible to run them all manually so a Perl script was created which would compile each test, run it, and pipe the output to a results file for future inspection. Appendix A contains the Perl script used, and a table of the results. The same tests were performed on the Pentium platform as well. The main results are:

- Of the 221 tests, 73 failed to compile
- Of the 148 remaining tests, 80 passed the conformance tests
- 23 conformance tests outputted a FAIL
- 40 tests failed to output to the screen
- 1 test was passed on the Playstation 2 which failed on the Pentium 4
- 12 tests were failed on the Playstation 2 which passed on the Pentium 4
- 31 tests which generated no output on the Playstation 2 generated output on the Pentium 4

These tests demonstrate the successful implementation of many of the issues involved in porting the compiler, but most importantly, since output is being generated, the procedure call mechanism is working correctly. In the cases where no output was being generated, it is possible that the assembly was incorrect, or there may be mistakes in the machine specification which cause assembly to not be generated, but still allow the compiler to exit successfully.

Tests were also generated throughout development to check if basic operations were working, for example, Figure 5-1 shows a Vector Pascal program to test the basic double precision operators. The output from this program is given in Figure 5-2.

```
program dpops;
var e,f : double;
begin
      e := 20000.0005;
      writeln(e);
      f := e + 1.5;
      writeln(f);
      f := e * 2.0;
      writeln(f);
      f := e - 3.0;
      writeln(f);
      f := e / 3.0;
      writeln(f);}
end.
```

Figure 5-1

```
20000.000499999998
20001.500499999998
40000.000999999997
19997.000499999998
6666.666833333333
```

Figure 5-2

In addition to testing double precision operations, programs small programs were written to test the basic multimedia instructions. These were declared to perform addition, subtraction, multiplication, and assignments of 128-bit vectors. Appendices B1, B2, and B3 list the source programs and results for unsigned 8-bit, signed 16-bit, and signed 32-bit vectors respectively.

6 Current status

Currently, the port of the Vector Pascal compiler to the Playstation 2 has been shown to function correctly for a large number of the conformance tests, and has produced working output for floating point and double precision instructions, as well as for the results of some integer vector operations. The compiler and its source code is being released to the open source community to allow them to develop it further if they wish to do so. The complete compiler with source code and this project report will be compressed and stored on the Vector Pascal website for download by anyone who wishes to use it. The following message was posted on the announcements board on the Playstation 2 Linux site.

“

To the Playstation 2 Linux community,

I have just completed my final year computing project at the University of Glasgow and would like to announce its release to the open source community. My project involved porting an existing compiler for a language called Vector Pascal to the Playstation 2. This language is an extension of Pascal which automatically generates parallel instructions for operations given on arrays. It currently supports most of the MIPS core instruction set, the floating point unit, and some of the multimedia instructions. In the future, the compiler may even support the vector units in micro mode.

The site for the compiler is http://www.dcs.gla.ac.uk/~wpc/reports/compilers/compilerindex/x25.html where a link will is available for you to download it.

Regards,
Peter Cooper
“

7 Future Improvements

The first improvement which should be made to this project is to generate test programs to fully test all of the instructions in the Emotion Engine machine specification. An analysis of the programs which failed the conformance tests should be done to make sure the failures are not due to the Emotion Engine specification. More importantly, the programs which ran the conformance tests but failed to output should be debugged to find out why no output was generated. The most obvious future improvement in terms of performance gained is the addition of code to execute instructions on the vector units. The macro mode registers and some of the common instructions have been defined, an generate what appears to be correct code, but which fails during execution. Fixing this code to make macro mode work would give a substantial increase in performance in code executing on the Playtation 2. Defining code to work on the either vector unit in micro mode is far mode difficult as they have their own data and instruction memories which must be given the appropriate information. The ILCG language currently does not support have more than one physical memory, so at present, interacting with the vector units cannot be done at the machine specification level. However, an extension proposed by Dr Paul Cockshott to allow units in Pascal to be compiled to different processors will allow an implementation of the vector units to be produced.

7.1 Compiling units to different platforms

In the current version of Vector Pascal, there is no way to compile code to the vector units, as they have different physical memory from the main core. Also, it is impractical to allow parts of a source file to be compiled to one processor, and other parts to another processor, since there would be global and nested variables visible to both machines which would cause problems with synchronisation, i.e., ensuring that both processors can see the same value for these variables. In addition to this, the vector units on the Playstation 2 have a very limited amount of memory, and if they had to store all variables, even if they did not need them, or even could not operate on them, this would quickly fill up their data memory.

The proposed extension would make use of the ability for Pascal programs to include units which are just like a library in C and contain functions the program needs to access. If a unit was written to only use types and instructions available to the vector unit, then it would be possible to compile that unit in Vector Pascal – assuming a suitable machine description has been written of course – and then the unit could be loaded into the instruction memory on the Playstation 2. The unit could define functions and procedures which are visible to the

program which includes it. In a typical unit, calls to these functions would be executed in the same way as a call to a function in the same file, i.e., parameters are pushed onto the stack, the function is called, and the result is retrieved from the result register. When calling a function in a different physical memory from the caller, the caller would have to transfer the parameters to the memory of the other processor, and then send a signal to the processor to start execution of the called function, and finally copy the result back from the memory in the other processor where the result was saved to. One possible way to do this would be to define an interface between processors.

7.1.1 Defining an interface between different processors

When a unit is included in a Vector Pascal program, the assembly output of the program will contain a number of include statements – one for each unit included, plus the inclusion of the default system unit, and a macro file if one is necessary. If the unit has been compiled to a different target platform than the program, then the assembly for the program and the nit will be incompatible and so will raise assembler errors. It is therefore undesirable to have the program include the assembly for any units not on the same platform. It would be possible to write interface code which can search for these include statements for different platforms, and substitute them for an include to an interface file. This interface file would have to have functions defined which use the same arguments and labels as those defined in the unit, but would be written in the language of the programs target platform and so could be included successfully by the assembler. The functions in the interface file would then contain code to transfer data to and from the units' functions, and call the units functions.

7.1.2 Interfacing the Emotion Engine and Vector Units

```
program ee;
uses floatops:apu[0];
var
        A, B, C: array [0..3] OF real;
begin
        a := fpadd(b,c);
end;
```

Figure 7-1

Figure 7-1 gives an example of a program which defines three arrays of 4 floats. It will be assumed that this file will be compiled to the Emotion Engine (EE) and that the floatops unit will be compiled to Vector Unit 0 (VU0). The important line is the second which declares this program to use the unit called floatops, and which wants this unit to be compiled to the platform with apu 0. The apu is an argument that will be passed to the compiler, for example, "apu0=VU0" would tell the compiler that any units with apu equal to 0 should be compiled to vector unit 0. At this stage, the compiler could check to ensure that a suitable interface has been defined between the Emotion Engine platform the program will be compiled to and the Vector Unit 0 platform the floatops unit will be compiled to.

The first step the compiler would take to compile the above code is to compile floatops to the VU0 target. When a program or unit is compiled, unique labels are inserted into the assembler code to identify each function. A call to a function will not use the name of the function in assembler, but instead the unique label for that function. When a unit is compiled, an mpu file is created for that unit. This is a serialized version of the ILCG tree representing that unit and contains all of the function labels, and the function arguments for that unit. It also contains any declared types or constants that programs including the unit may wish to use. Once the units for Figure 7-1 have been compiled, the program itself will be compiled. All code will be compiled as normal and the code generator does not have to take any special steps to distinguish code intended to call different platforms. The call to fpadd above would be generated as normal, with the parameters B and C passed on the stack, and a JAL instruction inserted to jump to the label for fpadd as defined in the mpu file for the floatops unit.

Once the assembler output has been generated for each of the units and the above program, the interface code could be called. This would generate as assembler file written for the Emotion Engine and would contain a function for each of the functions in the floatops unit. The functions in the interface would be given the same labels as those in the mpu file for floatops as this will then cause the assembler to jump to the interface versions of the functions when the main program tries to call these functions. This may raise some confusion, as there would then be a function present in the unit, and in the assembly code interface with the same label. One possible solution to this is to have all functions append their platform to the end of labels in the same way that their platform is appended to the name of the assembly output file. Each function in the interface would take its parameters and transfer them to Vector Unit 0. This can be done using the malloc procedure defined in C, or as covered in the VIF section, VIF codes can be used to instruct the DMA controller to transfer data from the location in memory where the parameters have been passed, to a location in VU0 data memory for that

parameter. Since it is only possible to transfer to a vector unit when it is not running, the interface to the unit can place the arguments in any locations in VU0 data memory, so long as it does not put parameters in the same memory locations, and defines a suitable place for the vector unit to place its results.

It is possible that as well as transferring the data to the vector unit, the instructions may also have to be passed. A function could be written to pass the instructions when the initialisation code for the floatops unit is called, but the instruction memory in the Vector Units is small, and it may not be possible to hold all the instructions for a unit in the micro memory at the same time.

Once the data has been loaded and the instructions are contained in micro memory, the vector unit must be instructed to start execution. There are a number of ways to do this, which have been covered earlier in the section on the Vector Units. The only way which can start either Vector Unit is the use of a VIF instruction called MSCAL. This can be passed the address of the first instruction to execute in micro memory, and will cause the vector unit to start execution at that address. For this to be possible, the addresses of the vector unit functions must be known. It may be possible to calculate these before execution, or the function to be executed could just always be loaded into address 0.

Once the interface assembly code has been generated, the interface generator can search the assembly file for the main program for the include statement for the unit and platform it has been designed to interface with. It can substitute this include line for one to include its own interface assembly code that it has generated. When the main program is then assembled, it will include the interface and will have code to suitably call the unit it uses. The only issue remaining is how to get the code from the object file for the unit into memory for the main program. There are two ways of doing this on the Playstation 2:

- Sony submitted a library of code as part of their Vector Unit competition which included methods to load a compiled vector unit object file and put it in memory [24]
- It is possible to strip the vector unit object file of any header code to get just the instructions. These can then be attached to the object file generated for the main program and suitable instructions given in the main program can extract this data from the object file and load them into memory.

It would be preferable to be able to execute instructions in the vector units while other instructions are being executed in the MIPS core. One way to achieve this, without adding any new syntax to the Vector Pascal language is to define a standard that if a function is

called on another platform, then the program will stall until the function has returned its result, whereas if a procedure has been called then it will execute without the main program stalling. This does place some requirements on the programmer in that they must ensure that it is possible for the procedure to execute in parallel with other code without causing undesirable results. The programmer may also wish for the main program to wait until a procedure has finished executing. A simple way to do this would be to define a wait function which could take a parameter for the apu to wait for. The interface could then define this function, and it could check the status of the control registers for the vector units to see when the vector unit has finished execution. It would be possible to simply define this as a busy loop which checks the status of the vector unit every cycle and breaks once execution has finished. Once the loop breaks, the function would exit and normal program code could continue.

7.1.3 Interfacing other platforms

In the case of the Emotion Engine, the interface code was used to transfer between processors in the same physical machine. However, the interface code can contain any valid code and so there is no reason why it could not call other computers via a sockets library. This would allow programs to be executed in parallel on a potentially very large number of machines at the same time. In this case the interface code would be far more complex, but other than this added complexity, there is no reason why a cluster of computers could not be used in parallel.

7.2 Compiling to other MIPS processors

It has been seen that the Playstation 2 R5900 processor implements instruction sets from MIPS I, II, II, and IV designs. It would be possible to separate these out into separate ILCG machine specifications for each of these platforms. A machine specification for the MIPS II design, for example, would be able to include the MIPS I design in its ILCG specification by way of the ILCG include tag. The MIPS II design would then only need to declare the MIPS II instructions, and not the MIPS I. It would also be possible to only define the registers, types, and memory addressing modes in the MIPS I design, and these too could be used by MIPS II. Using this feature, the Emotion Engine specification would only have to define the multimedia instructions, as it would get all of its other instructions from the MIPS specifications it includes. This would allow the Vector Pascal Compiler to compile to a number of extra targets will very little work, as most of the MIPS features are already defined in the Emotion Engine specification, and could simply be moved to the appropriate new machine specifications.

8 Future Platforms

8.1 The Cell Processor

In February 2005 at the International Solid-State Circuits Conference (ISSCC), Sony, IBM, and Toshiba gave the public their first insight into what they have called the Cell processor. The Cell processor has been designed to be a very flexible architecture as IBM does not believe that "the one-size-fits-all model of the PC does not apply in the embedded space" [25]. The Cell processor features a core CPU based on the IBM POWER architecture, and then up to 8 Synergistic Processing Elements (SPE). These SPE units are very similar in function to the Vector Units in the Playstation 2 in that they have their own vector processor and their own memory [26].

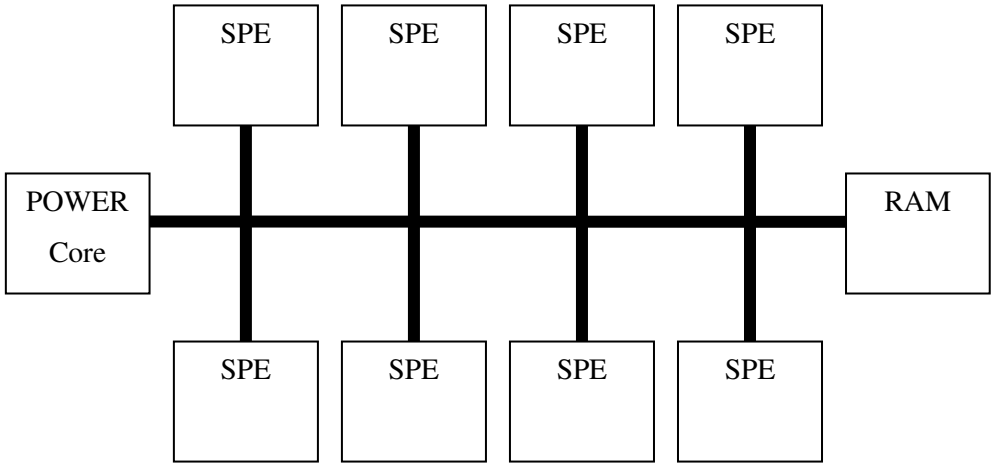


Figure 8-1

The structure of the Cell processor can be seen in Figure 8-1. It shows the POWER core, and the SPE units. Due to the speed of the elements in Cell, there is a need for very high memory bandwidth. This has been provided in Cell by Rambus, who manufactured the memory for the original Pentium 4, and for the Playstation 2. The memory in Cell has a bandwidth of over 70GB/s [27], far higher than the 6.4GB/s available to desktop PC's using DDR400 memory, or the DDR3 memory found in top of the range PC graphics cards. In addition to this high performance memory, the SPE's have also each been given 256KB of cache memory. This is far more than the 16KB found in Vector Unit 1 in the Playstation 2 and so the SPE should be able to process a lot of data before having to load more data from the memory. All this memory bandwidth will be needed if Cell is to get close to the 256GFLOPS it can theoretically perform at, assuming all 8 SPE's are used, and the processor is running at the 4GHZ speed demonstrated at the ISSCC conference.

The POWER core in Cell has been designed to coordinate all of the SPE units, as well as functioning as 64-bit processor which is fully compliant with the POWERPC instruction set. It will have a 32KB level 1 cache, and 512KB of level 2 cache. The core features a floating point unit similar to the Emotion Engine one, in that it still does not comply fully to IEEE754 single precision standards when rounding numbers. However, it will feature a fully IEEE754 compliant double precision floating point unit, which will be far quicker than the emulated instructions on the Emotion Engine. Since Cell using a POWERPC core, and not a MIPS one, there is no possibility of using the Playstation 2 port of Vector Pascal to execute code on the Cell architecture. Also, Sony have not yet announced whether they will release a Linux Kit for the Playstation 3, although IBM do plan to use Cell as the core for new workstations [25], and as IBM manufacture the Apple G5 processor, there is a chance that Apple may adopt Cell as the processor for their future PC's. However, any work that is done to get the Vector Units working in parallel with the MIPS core on the Playstation 2 will be applicable to performing the same task on the Cell, but with a far larger amount of parallelism.

8.2 Dual Core Processors

An alternative approach to that taken in Cell is to not have a single main processor, and many child processors, but instead put two fully capable processors on the same die. Intel recently demonstrated a dual core version of their Pentium 4 processors called the Pentium D [17]. Their main competitor, AMD, displayed a dual core version of their Opteron processor in 2004 [18]. Both companies have outlined plans to introduce dual core versions of their whole CPU ranges in the coming years. Intel has also added that dual core is just the first step, and that eventually CPU's with many cores will be introduced [17].

Dual core processors do, however, have one significant downfall in that they significantly increase the complexity of software designed to use it. One industry which sees will be the first to utilise dual core processing in their software in the computer games industry. With the Playstation 3 using a Cell processor, rumours suggesting the Xbox 2 will have multiple CPU's, and future PC's having dual cores, future games will have to utilise all of this power available to them to achieve their desired performance level. However, one programmer in the games industry has highlighted the two to three times greater time to develop code for these systems [19].

A novel approach will also have to be implemented in Vector Pascal if it is to support these processors. Unlike the Emotion Engine, there is no point in one core assigning a task to another core then waiting for the result, as the first core could simply have done the task

itself. In the Emotion Engine the vector units have different functionality to the main core, so this is acceptable since the vector unit will, in most cases, be far quicker at performing the task than the main core. One possible approach is to introduce threads to the Vector Pascal language in a similar way they were introduced to languages like Java. However, this puts the burden of thread synchronisation, and ensuring data is not corrupted by threads on the programmer. One potential solution could be to assign different parts of the same calculation to the different processors, i.e., assign one half of a vector operation to one core, and the other half of the vector operation to the other core. This would allow up to twice as may vector operations to be performed at the same time, although the second processor would then be unused for ordinary operations. This is the approach taken by Intel in their multi-threading enabled Fortran Compiler [8].

8.3 Using Graphics Cards as Vector Processors

There has been considerable research recently into using the graphics cards found in most modern PC's to perform vector operations. This is taking place because a top of the range graphics card can achieve a theoretical performance of 40 GFLOPS, while a 3GHz Pentium 4 can only achieve 6 GFLOPS [20]. Graphics cards do currently suffer from a limited instruction set, and very few registers, and also can currently only operate on streams of data, and not reference individual memory address. However, they are able to operate across very large pieces of data at a time, and have very high memory bandwidth. With the introduction of the PCI Express interface, there is now a far higher bandwidth between the graphics card and the main CPU which makes it less costly to transfer data to be processed by the graphics card particularly if the operation the graphics card is performing is a time consuming one.

Some notable uses of graphics card include:

- Image processing in medical applications, which currently takes a very long time on general purpose machines [20]
- Processing audio streams [21]
- Creating a cluster of GPU's to model a flow simulation of contaminant air particles in Times Square in New York [22]

Although it may be difficult to program the Graphics Processing Unit as a general purpose computer, using it to provide say an image processing library unit in Pascal could provide significant speedups over executing only on a single CPU.

8.4 Physics coprocessors

At the recent Game Developers Conference, a company called AGEIA introduced a physics card for PC's called PhysX [23]. This has been designed to take the burden of physics processing away from the CPU in the same way that the graphics card did this with 3D image processing. Few details are currently available for the card, but it is known that it will be released for PCI and PCI Express interfaces. The developers of the PhysX system have also released a software development kit to allow software developers to write software to use the system, in the same way that Open GL and Direct X allow developers to give graphics cards instructions. It may be possible to use this physics unit as a vector unit which would allow simultaneous processing in the main processor and in this unit.

9 Conclusion

Porting Vector Pascal to a new platform is not a trivial task. Writing the machine specification is not difficult once the importance of using instruction ordering, type casts and patterns is realised. The most difficult part of porting the compiler is understanding the assumptions built into the compiler, and implementing ways to support these given the limitations of the Playstation 2, for example, the issue of alignment when loading and saving data larger than 4 bytes in size.

The compiler has been designed in a very modular way, and it will be possible to port it to a wide variety of new platforms, and possibly some of those given in the Future Platforms Section. It should be possible in the future to add many of the features discussed earlier, including the more advanced vectorisation possibilities. With the increased demands on programmers to use vector instructions in general purpose computers, the simplicity of using these vector operations in Vector Pascal should make it easier for programmers to write very fast programs, taking full advantage of the parallelism available on the target platform, with minimal effort from the programmers perspective.

Appendix A

The script below compiles each conformance test on the Linux machine Java was running from. The standard output generated from the Vector Pascal Compiler is appended to a file called output.txt to make it possible to see at a later time what programs compiled properly, and the reasons why some failed to compile. It then assembles the output, automatically including the system.pas unit, and sends the generated object file to the Playstation 2. It relies on the ability for SSH to generate Kerberos keys to allow remote access to the Playstation 2 without the need for entering passwords. The file is then linked with the run-time library on the Playstation 2 to generate an executable file. The program is then executed on the Playstation 2, and the output appended to the results.txt file on the Linux machine running the Perl script.

```
#! /usr/bin/perl -w


system("ls -1 ../CONFORM/CONF*.PAS >files");
open FILES,"files";
$mmpcdir="../..";
while(<FILES>){
        $file = $_;
        chomp($file);
        system "echo Source ".$file."\n";
        system "rm EECG.vwu\n";
        system  "cp ".$file." ./test.pas\n" ;
        system "echo Source ".$file." >> output.txt";
   system " java -Xmx200m  -Dmmpcdir=${mmpcdir} -jar ${mmpcdir}/mmpc.jar test -S
-cpuEE -felf -Ap.s >> output.txt";
        system "mipsEEel-linux-as -a=asm.lst -m5900 -o p.o ./p.s\n ";
        system "scp p.o angel19:\n";
        system  "ssh  angel19  \"mipsEEel-linux-gcc  p.o  mmpc/rtl.c  -o  a.out  -lm  -
DEECG\n\"";
        system "ssh angel19 \"./a.out\n\" >> results.txt";
}
```

The table below lists: the conformance tests, the outputs from the Vector Pascal compiler for those which compiled and successfully printed something to screen, and whether the Pentium passed the same tests.

Test	Compiled?	EE Output	P4 Output
CONF001.PAS	Y	PASS...6.1.1-1	PASS...6.1.1-1
CONF002.PAS	Y	PASS...6.1.1-2	PASS...6.1.1-2
CONF003.PAS			
CONF004.PAS	Y	FAIL...6.1.2-1	FAIL...6.1.2-1
CONF005.PAS			
CONF006.PAS	Y	PASS...6.1.2-3	PASS...6.1.2-3
CONF007.PAS	Y	PASS...6.1.3-1	PASS...6.1.3-1
CONF008.PAS	Y	PASS...6.1.3-2	PASS...6.1.3-2
CONF009.PAS	Y	PASS...6.1.5-1	PASS...6.1.5-1
CONF010.PAS	Y	PASS...6.1.5-2	PASS...6.1.5-2
CONF011.PAS	Y	PASS...6.1.6-1	FAIL...6.1.6-1
CONF012.PAS	Y	PASS...6.1.6-2	PASS...6.1.6-2
CONF013.PAS	Y	PASS...6.1.6-3	PASS...6.1.6-3
CONF014.PAS	Y	PASS...6.1.7-1	PASS...6.1.7-1
CONF015.PAS	Y	FAIL...6.1.7-2	PASS...6.1.7-2
CONF016.PAS	Y	FAIL...6.1.7-3	PASS...6.1.7-3
CONF017.PAS	Y	PASS...6.1.8-1	PASS...6.1.8-1
CONF018.PAS	Y	PASS...6.1.8-2	PASS...6.1.8-2
CONF019.PAS	Y	FAIL...6.1.9-1	FAIL...6.1.9-1
CONF020.PAS	Y	PASS...6.1.9-2	PASS...6.1.9-2
CONF021.PAS	Y	PASS...6.1.9-3	PASS...6.1.9-3
CONF022.PAS	Y	FAIL...6.2.1-1	PASS...6.2.1-1
CONF023.PAS	Y	FAIL...6.2.1-2	PASS...6.2.1-2
CONF024.PAS	Y	None in code	None in code
CONF025.PAS	Y	PASS...6.2.2-1	PASS...6.2.2-1
CONF026.PAS	Y	PASS...6.2.2-2	PASS...6.2.2-2
CONF027.PAS	Y	PASS...6.2.2-3	PASS...6.2.2-3
CONF028.PAS	Y	PASS...6.2.2-4	PASS...6.2.2-4
CONF029.PAS	Y	PASS...6.2.2-5	PASS...6.2.2-5
CONF030.PAS	Y	FAIL...6.2.2-6	PASS...6.2.2-6
CONF031.PAS	Y	PASS...6.2.2-7	PASS...6.2.2-7
CONF032.PAS	Y	FAIL...6.3-1	FAIL...6.3-1
CONF033.PAS	Y	FAIL...6.3-10	PASS...6.3-10
CONF034.PAS	Y	PASS...6.4.1-1	PASS...6.4.1-1
CONF035.PAS	Y	PASS...6.4.2.2-1	PASS...6.4.2.2-1
CONF036.PAS	Y	PASS...6.4.2.2-2	PASS...6.4.2.2-2
CONF037.PAS	Y	FAIL...6.4.2.2-3	FAIL...6.4.2.2-3

CONF038.PAS	Y	PASS...6.4.2.2-4	PASS...6.4.2.2-4
CONF039.PAS	Y	PASS...6.4.2.2-5	PASS...6.4.2.2-5
CONF040.PAS	Y	PASS...6.4.2.2-6	PASS...6.4.2.2-6
CONF041.PAS	Y	PASS...6.4.2.2-6	PASS...6.4.2.2-6
CONF042.PAS	Y	PASS...6.4.2.2-8	PASS...6.4.2.2-8
CONF043.PAS	Y	PASS...6.4.2.3-1	PASS...6.4.2.3-1
CONF044.PAS	Y	PASS...6.4.2.3-2	PASS...6.4.2.3-2
CONF045.PAS	Y	PASS...6.4.2.3-3	PASS...6.4.2.3-3
CONF046.PAS	Y	PASS...6.4.2.3-4	PASS...6.4.2.3-4
CONF047.PAS	Y	PASS...6.4.2.4-1	PASS...6.4.2.4-1
CONF048.PAS	Y	PASS...6.4.2.4-2	PASS...6.4.2.4-2
CONF049.PAS	Y	PASS...6.4.3.1-1	PASS...6.4.3.1-1
CONF050.PAS	Y	PASS...6.4.3.1-2	PASS...6.4.3.1-2
CONF051.PAS	Y	PASS...6.4.3.2-1	PASS...6.4.3.2-1
CONF052.PAS	Y	No output	No output
CONF053.PAS	Y	PASS...6.4.3.2-3	PASS...6.4.3.2-3
CONF054.PAS			
CONF055.PAS	Y	PASS...6.4.3.3-1	PASS...6.4.3.3-1
CONF056.PAS	Y	PASS...6.4.3.3-2	PASS...6.4.3.3-2
CONF057.PAS	Y	PASS...6.4.3.3-3	PASS...6.4.3.3-3
CONF058.PAS	Y	PASS...6.4.3.3-4	PASS...6.4.3.3-4
CONF059.PAS	Y	PASS...6.4.3.3-5	PASS...6.4.3.3-5
CONF060.PAS	Y	PASS...6.4.3.3-6	PASS...6.4.3.3-6
CONF061.PAS	Y	PASS...6.4.3.3-7	PASS...6.4.3.3-7
CONF062.PAS	Y	PASS...6.4.3.3-17	PASS...6.4.3.3-17
CONF063.PAS			
CONF064.PAS	Y	No output	
CONF065.PAS	Y	No output	PASS...6.4.3.4-11
CONF066.PAS	Y	PASS...6.4.3.5-1	PASS...6.4.3.5-1
CONF067.PAS			
CONF068.PAS			
CONF069.PAS			
CONF070.PAS			
CONF071.PAS			
CONF072.PAS			
CONF073.PAS			
CONF074.PAS			
CONF075.PAS			

CONF076.PAS			
CONF077.PAS	Y	No output	PASS...6.4.4-1
CONF078.PAS			
CONF079.PAS	Y	PASS...6.4.5-1	PASS...6.4.5-1
CONF080.PAS	Y	PASS...6.4.5-2	PASS...6.4.5-2
CONF081.PAS	Y	PASS...6.4.5-3	PASS...6.4.5-3
CONF082.PAS			
CONF083.PAS			
CONF084.PAS	Y	PASS...6.4.5-6	PASS...6.4.5-6
CONF085.PAS	Y	No output	PASS...6.4.6-1
CONF086.PAS	Y	No output	PASS...6.4.6-2
CONF087.PAS	Y	PASS...6.4.6-3	PASS...6.4.6-3
CONF088.PAS			
CONF089.PAS			
CONF090.PAS			
CONF091.PAS			
CONF092.PAS	Y	PASS...6.6.1-1	PASS...6.6.1-1
CONF093.PAS			
CONF094.PAS	Y	No output	PASS...6.6.2-1
CONF095.PAS			
CONF096.PAS	Y	No output	PASS...6.6.2-3
CONF097.PAS	Y	FAIL...6.6.2-4	PASS...6.6.2-4
CONF098.PAS	Y	FAIL...6.6.2-11	PASS...6.6.2-11
CONF099.PAS	Y	No output	PASS...6.6.2-12
CONF100.PAS	Y	FAIL...6.6.3.1-1	PASS...6.6.3.1-1
CONF101.PAS			
CONF102.PAS	Y	No output	PASS...6.6.3.1-3
CONF103.PAS			
CONF104.PAS	Y	PASS...6.6.3.1-7	PASS...6.6.3.1-7
CONF105.PAS			
CONF106.PAS			
CONF107.PAS	Y	No output	PASS...6.6.3.2-3
CONF108.PAS	Y	FAIL...6.6.3.3-1	PASS...6.6.3.3-1
CONF109.PAS	Y	No output	PASS...6.6.3.3-2
CONF110.PAS	Y	FAIL...6.6.3.3-3	PASS...6.6.3.3-3
CONF111.PAS			
CONF112.PAS			
CONF113.PAS			

CONF114.PAS			
CONF115.PAS			
CONF116.PAS			
CONF117.PAS			
CONF118.PAS			
CONF119.PAS	Y	No output	No output
CONF120.PAS			
CONF121.PAS			
CONF122.PAS			
CONF123.PAS	Y	PASS...6.6.5.3-2	PASS...6.6.5.3-2
CONF124.PAS			
CONF125.PAS			
CONF126.PAS			
CONF127.PAS	Y	PASS...6.6.5.3-20	PASS...6.6.5.3-20
CONF128.PAS			
CONF129.PAS			
CONF130.PAS	Y	No output	FAIL...6.6.5.3-27
CONF131.PAS			
CONF132.PAS			
CONF133.PAS	Y	FAIL...6.6.6.2-1	FAIL...6.6.6.2-1
CONF134.PAS	Y	No output	No output
CONF135.PAS			FAIL...6.6.6.2-3
CONF136.PAS	Y	No output	
CONF137.PAS	Y	FAIL...6.6.6.4-1	FAIL...6.6.6.4-1
CONF138.PAS	Y	No output	PASS...6.6.6.4-2
CONF139.PAS			
CONF140.PAS	Y	PASS...6.6.6.4-10	PASS...6.6.6.4-10
CONF141.PAS			
CONF142.PAS	Y	PASS...6.6.6.5-2	PASS...6.6.6.5-2
CONF143.PAS			
CONF144.PAS			
CONF145.PAS			
CONF146.PAS			
CONF147.PAS			
CONF148.PAS			
CONF149.PAS	Y	No output	PASS...6.7.1-9
CONF150.PAS			
CONF151.PAS	Y	No output	PASS...6.7.2.2-1

CONF152.PAS	Y	FAIL...6.7.2.2-2	FAIL...6.7.2.2-2
CONF153.PAS	Y	FAIL...6.7.2.2-3	FAIL...6.7.2.2-3
CONF154.PAS			
CONF155.PAS	Y	PASS...6.7.2.3-1	PASS...6.7.2.3-1
CONF156.PAS	Y	No output	PASS...6.7.2.4-1
CONF157.PAS	Y	No output	PASS...6.7.2.4-2
CONF158.PAS			
CONF159.PAS	Y	No output	No output
CONF160.PAS			
CONF161.PAS	Y	No output	PASS...6.7.2.5-2
CONF162.PAS			
CONF163.PAS			
CONF164.PAS	Y	No output	No output
CONF165.PAS			
CONF166.PAS	Y	No output	FAIL...6.8.1-8
CONF167.PAS			
CONF168.PAS	Y	No output	FAIL...6.8.2.4-1
CONF169.PAS	Y	No output	No output
CONF170.PAS			
CONF171.PAS			
CONF172.PAS	Y	PASS...6.8.3.7-1	PASS...6.8.3.7-1
CONF173.PAS	Y	PASS...6.8.3.7-2	PASS...6.8.3.7-2
CONF174.PAS	Y	PASS...6.8.3.7-3	PASS...6.8.3.7-3
CONF175.PAS	Y	PASS...6.8.3.8-1	PASS...6.8.3.8-1
CONF176.PAS	Y	PASS...6.8.3.8-2	PASS...6.8.3.8-2
CONF177.PAS			
CONF178.PAS	Y	No output	No output
CONF179.PAS	Y	PASS...6.8.3.9-3	PASS...6.8.3.9-3
CONF180.PAS	Y	FAIL...6.8.3.9-4	PASS...6.8.3.9-4
CONF181.PAS	Y	PASS...6.8.3.9-23	PASS...6.8.3.9-23
CONF182.PAS	Y	PASS...6.8.3.9-25	PASS...6.8.3.9-25
CONF183.PAS	Y	PASS...6.8.3.9-26	PASS...6.8.3.9-26
CONF184.PAS	Y	FAIL...6.8.3.9-28	FAIL...6.8.3.9-28
CONF185.PAS	Y	PASS...6.8.3.10-1	PASS...6.8.3.10-1
CONF186.PAS	Y	PASS...6.8.3.10-2	PASS...6.8.3.10-2
CONF187.PAS	Y	PASS...6.8.3.10-3	PASS...6.8.3.10-3
CONF188.PAS	Y	FAIL...6.8.3.10-4	FAIL...6.8.3.10-4
CONF189.PAS	Y	FAIL...6.8.3.10-5	FAIL...6.8.3.10-5

CONF190.PAS	Y	PASS...6.8.3.10-6	PASS...6.8.3.10-6
CONF191.PAS	Y	PASS...6.8.3.10-8	PASS...6.8.3.10-8
CONF192.PAS	Y	No output	Output, but no PASS
CONF193.PAS	Y	No output	Output, but no PASS
CONF194.PAS	Y	No output	Output, but no PASS
CONF195.PAS			
CONF196.PAS	Y	No output	Output, but no PASS
CONF197.PAS			
CONF198.PAS	Y	No output	Output, but no PASS
CONF199.PAS	Y	No output	Output, but no PASS
CONF200.PAS			
CONF201.PAS			
CONF202.PAS			
CONF203.PAS			
CONF204.PAS			
CONF205.PAS	Y	No output	Output, but no PASS
CONF206.PAS			
CONF207.PAS			
CONF208.PAS	Y	PASS...6.10-2	PASS...6.10-2
CONF209.PAS	Y	PASS...6.10-3	PASS...6.10-3
CONF210.PAS	Y	PASS...6.10-5	PASS...6.10-5
CONF211.PAS	Y	PASS...6.10-6	PASS...6.10-6
CONF212.PAS	Y	No output	Output, but no PASS
CONF213.PAS	Y	No output	Output, but no PASS
CONF214.PAS	Y	PASS...6.8.3.5-23	Output, but no PASS
CONF215.PAS			
CONF216.PAS			
CONF217.PAS	Y	No output	PASS...6.4.3.3-25
CONF218.PAS	Y	PASS...6.1.5-14	PASS...6.1.5-14
CONF219.PAS	Y	No output	Output, but no PASS
CONF220.PAS	Y	No output	Output, but no PASS
CONF221.PAS	Y	No output	Output, but no PASS

Appendix B1

Test program for unsigned 8-bit integer vectors.

```
program bytevec;
type int8 = -128..127;
type intarray = array[0..15] of int8;
var
      a,b,c : intarray;
begin
      a[0] := 1;
      a[1] := 2;
      a[2] := 3;
      a[3] := 4;
      a[4] := 5;
      a[5] := 6;
      a[6] := 7;
      a[7] := 8;
      a[8] := 9;
      a[9] := 10;
      a[10] := 11;
      a[11] := 12;
      a[12] := 13;
      a[13] := 14;
      a[14] := 15;
      a[15] := 16;
      writeln(a);
      b := a + a;
      writeln(b);
      c := b;
      b := a - c;
      writeln(b);
end.
```

Output:

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14   15   16

2  4  6  8  10  12  14  16  18  20  22  24  26  28  30  32

-1  -2  -3  -4  -5  -6  -7  -8  -9  -10  -11  -12  -13 -14 -15 -16
```

Appendix B2

Test program for signed 16-bit integer vectors.

```
program hwvec;
type intarray = array[0..7] of word;
var
      a,b,c : intarray;
begin
      a[0] := 1;
      a[1] := 2;
      a[2] := 3;
      a[3] := 4;
      a[4] := 5;
      a[5] := 6;
      a[6] := 7;
      a[7] := 8;
      writeln(a);
      b := a + a;
      writeln(b);
      c := b;
      b := a - c;
      writeln(b);
      b := a * a;
      writeln(b);
end.
```

Output:

```
1  2  3  4  5  6  7  8

2  4  6  8  10  12  14  16

-1  -2  -3  -4  -5  -6   -7   -8

1   4   9   16   25   36   49   64
```

Appendix B3

Test program for signed 32-bit integer vectors.

```
program swvec;
type intarray = array[0..3] of integer;
var
      a,b,c : intarray;
begin
      a[0] := 1;
      a[1] := 2;
      a[2] := 3;
      a[3] := 4;
      writeln(a);
      b := a + a;
      writeln(b);
      c := b;
      b := a - c;
      writeln(b);
      b := a * a;
      writeln(b);
end.
```

Output:

```
1         2         3         4

2         4         6         8

-1        -2        -3        -4

1         4         9         16
```

References

1. Intel, Intel Architecture Software Developers Manual 1, 1999
2. AMD, 3DNow! Technology Manual, 1999
3. Sony, Introducing Playstation 2, SIGGRAPH 2000
4. WorldNet Daily, Why Iraq's Buying Up Playstation 2's, 2000
5. Paul Cockshott, Kenneth Renfrew, SIMD Programming Manual for Linux and Windows, 2004
6. Craig Steffen, University of Illinois Computing Science Department, Other Playstation 2 Projects, 2003
7. New York Times, From Playstation to supercomputer for $50,000, 2003
8. Intel, Intel OpenMP C++/Fortran Compiler for Hyper-threading Technology: Implementation and Performance, Intel Technology Journal Q1, 2002
9. Paul Cockshott, Direct Compilation of High Level Languages for Multi-media Instruction-sets, Department of Computer Science, University of Glasgow, Nov 2000
10. David A. Patterson, John L. Hennessy, Computer Organisation and Design, Third Edition, 2005
11. Sony, EE Core Instruction Set Manual, 2001
12. Dr Henry S Fortuna, University of Abertay, PS2 Linux Programming
13. Sony, VU User's Manual, 2001
14. SGI, SGI Tech Pages, IRIX 5.3 Man Pages, 1993
15. HP, HP-UX Linker and Libraries User's Guide, 1997
16. MIPS, System V Application Binary Interface, 1996
17. Intel, Dual Core Architecture, Intel Developer Forum, Spring 2005
18. AMD, AMD Demonstrates Worlds First x86 Dual-core processor, 2004
19. Anandtech, The Quest for More Processing Power, Part Two, 2005
20. Martin Botnen, Harald Ueland, The GPU as a Computational Resource in Medical Image Processing, 2004
21. BionicFX, BionicFX announces first Audio Processing on Nvidia GPU, Press Release, 2004
22. Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover, GPU Cluster for High Performance Computing, 2004
23. Anandtech, A Closer look at PhysX : Our look at the PPU, 2005
24. Playstation 2 Linux, SCEA Devcon 2003, VU Demo Coding Contest, 2003
25. IBM, IBM Venture Capital Group, STI Cell Processor, 2005
26. Ars Technica, Introducing the IBM/Sony/Toshiba Cell processor, Parts I and II, 2005
27. David T. Wang, Real World Technologies, ISSCC 2005 : The Cell Processor, 2005

Thanks to…

Paul Cockshott for his essential guidance throughout the project

Gary Gray for all his help getting the Playstation 2 set up

The communities at Playstation-linux.org and ps2dev.org for all of their useful tutorials