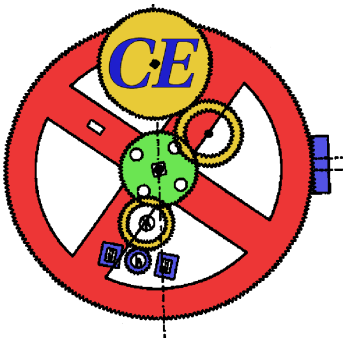

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2007

MSc THESIS

Two tools to aid new developments with SimpleScalar and CellSim

C.M. van der Hoeven

CE-MS-2007-18

Abstract

This thesis presents two tools that will aid researchers to evaluate new architectural ideas. The first tool is integrated in a toolchain that has been developed around the SimpleScalar processor simulator. SimpleScalar is a simulation toolset that is often used in computer architecture research. SimpleScalar provides a mechanism to synthesize new instructions without modifying the assembler, by annotating existing instructions in the assembly files. To facilitate the extension of the SimpleScalar simulator, two tools have already been designed by the Computer Engineering Laboratory of Delft University of Technology. To facilitate extending SimpleScalar even further this thesis presents the SimpleScalar Macro Tool (SSMT), a tool to enable the developer to write ISA extensions in the C programming language instead of in assembly which is required thus far. This makes extending the SimpleScalar simulation toolset with new instructions much easier because the developer can now concentrate on writing optimal code without having to bother about register allocation and instruction ordering of the existing ISA. Experiments conducted on several multimedia kernels show that SSMT does not introduce a significant performance overhead and in some cases even increases the performance by up to 6% compared to hand-written assembly code.
The second tool is an extension of the the Cell Broadband Engine Simulator (CellSim) that has been developed by the Barcelona Supercomputing Center. This simulator is designed to cycle-accurately simulate the Cell Broadband Engine that was recently designed by a joint operation of Sony, Toshiba and IBM. The Cell BE processor implements a PowerPC processor core which controls eight synergistic processors that are optimized for vector operations. Everything is connected by a circular bus that is optimized for large data transfers. CellSim is developed in the rather new simulator environment UNISIM to enable easy modular design. The tool presented in this work is designed to augment CellSim implements a profiler that allows the programmer to selectively log the events of his or her choice and view the results with cycle-level accuracy. This aids the developer in finding bottlenecks in either hardware or software.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Two tools to aid new developments with SimpleScalar and CellSim

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

C.M. van der Hoeven
born in Delft, Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Two tools to aid new developments with SimpleScalar and CellSim

by C.M. van der Hoeven

Abstract

This thesis presents two tools that will aid researchers to evaluate new architectural ideas. The first tool is integrated in a toolchain that has been developed around the SimpleScalar processor simulator. SimpleScalar is a simulation toolset that is often used in computer architecture research. SimpleScalar provides a mechanism to synthesize new instructions without modifying the assembler, by annotating existing instructions in the assembly files. To facilitate the extension of the SimpleScalar simulator, two tools have already been designed by the Computer Engineering Laboratory of Delft University of Technology. To facilitate extending SimpleScalar even further this thesis presents the SimpleScalar Macro Tool (SSMT), a tool to enable the developer to write ISA extensions in the C programming language instead of in assembly which is required thus far. This makes extending the SimpleScalar simulation toolset with new instructions much easier because the developer can now concentrate on writing optimal code without having to bother about register allocation and instruction ordering of the existing ISA. Experiments conducted on several multimedia kernels show that SSMT does not introduce a significant performance overhead and in some cases even increases the performance by up to 6% compared to hand-written assembly code.
The second tool is an extension of the the Cell Broadband Engine Simulator (CellSim) that has been developed by the Barcelona Supercomputing Center. This simulator is designed to cycle-accurately simulate the Cell Broadband Engine that was recently designed by a joint operation of Sony, Toshiba and IBM. The Cell BE processor implements a PowerPC processor core which controls eight synergistic processors that are optimized for vector operations. Everything is connected by a circular bus that is optimized for large data transfers. CellSim is developed in the rather new simulator environment UNISIM to enable easy modular design. The tool presented in this work is designed to augment CellSim implements a profiler that allows the programmer to selectively log the events of his or her choice and view the results with cycle-level accuracy. This aids the developer in finding bottlenecks in either hardware or software.

Laboratory : Computer Engineering
Codenumber : CE-MS-2007-18

Committee Members :

Advisor: B.H.H. (Ben) Juurlink

Advisor: D. Borodin

Advisor: C.H. Meenderinck

Chairperson:	K. Goossens
Member:	B.H.H. (Ben) Juurlink
Member:	D. Borodin
Member:	C.H. Meenderinck
Member:	J.S.S.M. Wong
Member:	N.P. van der Meijs

Contents

List of Figures

Acknowledgements

I would like to thank Ben Juurlink and the people of the Barcelona Supercomputing Center, because without them, the two projects described in this thesis would not have been possible. I would like to thank Ben Juurlink also for the help he gave me while writing this thesis by correcting mistakes and the English language and giving me advice about the layout of the chapters.
I would also like to thank Demid Borodin and Cor Meenderinck for helping me with writing this thesis as well as with all the questions I had during the development of the two tools. For helping me out with any questions I had about CellSim I also would like to thank David R'odenas-Pic'o, Felipe Carbacas and Augusto Vega from the Barcelona Supercomputing Center who are part of the team that develops CellSim and provided me with the bits of information I needed every now and then.
Further I would like to thank all my family and friends who supported me during the time I spent working on the projects and kept listening to stories they only partially understood.

C.M. van der Hoeven
Delft, The Netherlands
November 20, 2007

1 Introduction

Simulation is often used to explore new ideas or optimizations in existing solutions. This thesis describes two tools that will aid the developer with either one of these problems and extra work we have performed between the development of these tools.
First a tool is described that works within a toolchain that has been developed around the SimpleScalar Toolset [14, 16] by the Computer Engineering Laboratory of Delft University of Technology. SimpleScalar is a processor simulator toolset that provides a developer with the possibility to introduce ISA extensions, functional units and architectural elements to an existing processor. The current toolchain, that consists of two tools, eases the implementation of these new elements which is quite error prone, especially to a novice at SimpleScalar. The first tool is called the SimpleScalar Instruction Tool (SSIT) [19, 11]. It facilitates the developer with the possibility to introduce new functional units and instructions to the existing processor. It also enables the developer to write readable assembly code to optimize a program with new instructions. The second tool is called the SimpleScalar Architecture Tool (SSAT) [19, 11]. This tool enables the developer to create registers with different sizes and to extend or alias existing registers of the existing processor. It also enables the developer to use the new, aliased or extended registers in readable assembly code. The new tool that is introduced into the toolchain is called the SimpleScalar Macro Tool (SSMT) and introduces a macro-like programming interface to transfer the programming with ISA extensions from the assembly programming level, where SSIT and SSAT operate, to the C programming level. The use of this macro-like programming interface results, surprisingly, in most to a performance that is comparable to hand-written assembly and in some cases even to a performance that is higher than hand-written assembly code. The SimpleScalar Macro Tool is described in detail in Chapter 2.
After completing the SimpleScalar Macro Tool it became clear that some members of the Computer Engineering research group were focusing on a different simulator: the Cell Broadband Engine [7] Simulator (CellSim) that is developed by the Barcelona Supercomputing Center. This simulator cycle-accurately simulates the Cell Broadband Engine which is developed jointly by Sony, Toshiba and IBM and is known in the consumer market by its appearance in the Playstation 3. The simulator, however, is still being developed. Therefore we have contributed to this project by implementing and testing a total of 53 instructions for the Synergistic Processor Unit (see Section 3.1 and [7]) of the simulator. This is described in Chapter 4.
The second tool that is described in this thesis is developed for the Cell simulator mentioned above. It is a tool that enables a developer to log arbitrary events and make a detailed profile of the use of the processor during execution. This tool is still under development, however, and some of the implementations that are mentioned in Chapter 5 are still subject to changes.

2 SimpleScalar Macro Tool

2.1 Introduction to the SimpleScalar Macro Tool

The Computer Engineering Laboratory of Delft University of Technology is working with the SimpleScalar Processor Simulator Toolset [16, 14, 10]. This toolset enables the developers to effectively implement new instructions into the existing SimpleScalar Processor and simulate them to evaluate new ideas. In order to make this easier for the developer there are already two tools available to work with the SimpleScalar toolset. The first of these tools is the SimpleScalar Instruction Tool (SSIT) [19, 11]. This tool was developed to facilitate the implementation of new instructions into the SimpleScalar simulator and to write readable assembly code as well as to add new functional units to the processor. The second tool is called the SimpleScalar Architecture Tool (SSAT) [19, 11]. This tool was built to facilitate the extend the SimpleScalar architecture with registers of different sizes and to give the developer the ability to alias or extend existing registers of the SimpleScalar simulator as well as to write readable assembly code.
Although SSIT and SSAT substantially simplify the work of a developer, he or she still needs to program the ISA extensions in assembly language, giving the developer all the problems that come with this task. For that reason a third tool has been developed. The new tool, that is discussed below, is called SSMT or SimpleScalar Macro Tool. This tool is created to facilitate the use of new instructions by means of a macro-like programming interface to write ISA extensions directly in the C code that is to be optimized for the new instruction set.
This chapter is organized as follows: Section 2.2 briefly discusses work related to SSMT. In Section 2.3 the externals and internals of the tool itself are discussed. After that the use of the tool is illustrated by means of a small example. To test and evaluate SSMT, some handwritten kernels were rewritten in C code with the use of the original files and the new macros and are compared to their hand written counterpart. The changes that were made to the original files to use the new macros are discussed in Section 2.4. The performance results of these rewritten kernels are discussed next in Section 2.5 followed by an explanation of these results.

2.2 Related Work

This section discusses some of the work that is related to the SimpleScalar Macro Tool and thereby related to SimpleScalar itself. To start with the latter, SimpleScalar has been used to evaluate many new architectural designs such as novel branch predictors [18, 26], cache organizations [22, 28] , instruction set extensions [17, 25] and fault tolerance schemes [24, 15]. Although SSMT is not used in recent research projects due to its recent development, the predecessors of SSMT, SSIT and SSAT [19, 11], have been used

in research projects such as the implementation of Intel's Streaming SIMD Extension (SSE) [21] and a comparison between a media extension proposal named MMMX and MMX in [25]. Both tools are also used by faculties of the University of Antwerp and Yale University.
Macro-like programming interfaces are not uncommon in the area of Computer Engineering. The Altivec ISA extensions [20] and the TriMedia CPU64 [27] ISA extensions which were both designed to speed up processing by using vector instructions both use macros to make programming possible. Even though in the case of the Altivec extensions this was realized using GCC intrinsics and the macros for the TriMedia were used to make automated changes to both the simulator and the compiler [23]. GCC intrinsics [5] is another means of providing a programmer with a macro programming interface with the difference that they need the compiler to be partially reprogrammed and the fact that type checking and possibly automated register allocation is performed. Programming intrinsics, however, is more difficult and involves more programming effort and requires a thorough understanding of the inner workings of the GNU GCC compiler without giving much benefits over macros or in-line assembly.

2.3 Tool Description

The SimpleScalar Macro Tool (SSMT) is a tool designed to work within the tool chain of which the SSIT and SSAT tools are already part. This toolchain is depicted in Figure 2.1. Unlike SSIT and SSAT, which work on assembly code, SSMT is used before compiling C code to assembly. SSMT takes in the SSIT configuration file and generates a header file with C macros that can be used throughout any C code that needs uses the new instructions these macros refer to. This section first describes the usage of SSMT, followed by the internals of the tool. Next some design choices are discussed and an example is given.

2.3.1 Usage

SSMT is a command line tool with the following usage description:

```
--== SimpleScalar Macro Tool  ==--

This tool is intended for creating a header file
to make new instructions available as macros.
Register allocation for new registers needs to
be done manually.

Options are as follows:

-c Specify the SSIT configuration file, default is 'ssit_config.cfg'
-o Specify the output file, default is 'ssmt.h'
-h Print this help
```

The tool can be used like in the following examples:

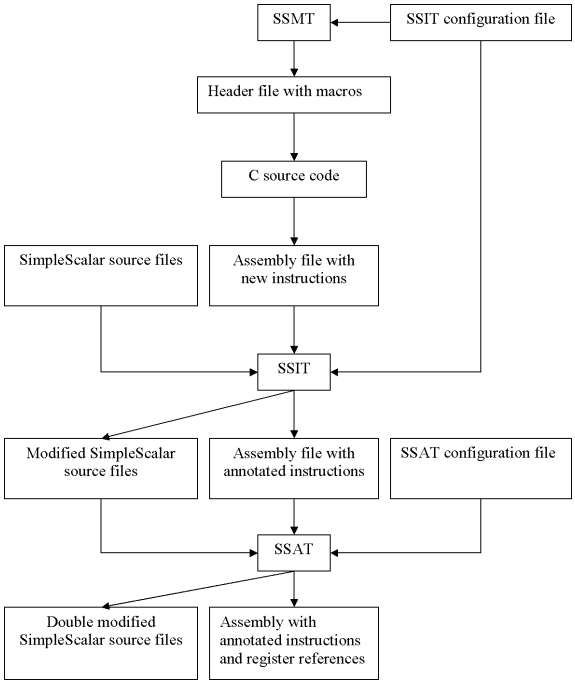

Figure 2.1: Toolchain surrounding SimpleScalar and involving SSMT, SSIT and SSAT

- ssmt
  Produces 'ssmt.h' from 'ssit_config.cfg'

- ssmt -c other_ssit.cfg
  Produces 'ssmt.h' from 'other_ssit.cfg'

- ssmt -o other_out.h
  Produces 'other_out.h' from 'ssit_config.cfg'

- ssmt -c other_ssit.cfg -o other_out.h
  Produces 'other_out.h' from 'other_ssit.cfg'

- ssmt -h
  Prints the usage description without doing anything more

- ssmt -h -c other_ssit.cfg
  Prints the usage description and produces 'ssmt.h' from 'other_ssit.cfg'

2.3.2 Internals

This section discusses the internals of the SimpleScalar Macro Tool. First the SSIT configuration file is described which is needed to use SSMT and specifies what information for creating the macros is used. Then the general form of the macro is discussed. Finally the translation of the macros during compile time is described.

2.3.2.1 SSIT Configuration File

In order to run SSIT and SSMT properly, a configuration file needs to be provided. The contents and the format of this file have been defined by the developers of SSIT, but is very suitable to provide SSMT with the appropriate information. A SSIT configuration file consists of a number of sections with either one of two types: (1) FUNCTIONAL_UNITS sections in which new functional units can be defined or (2) MACHINE.DEF sections that allow a developer to define new instructions. Since the FUNCTIONAL_UNITS sections are not used by SSMT they are not discussed further. The MACHINE.DEF sections are constructed as follows:

```
#define INSTRUCTION_NAME_IMPL {                 \
<implementation of the new instruction in C>    \
}
DEFINST( "<instruction.name>",   <operands>,
         "<annotated.instruction.name>",
         <FU_class>,             <iflags>,
         <odep1>, <odep2>,       <idep1>, <idep2>, <idep3>
)
```

where

<**instruction.name**> is the readable mnemonic of the new instruction with dots instead of spaces, e.g. "vadd" for "vector add".

<**operands**> is a string that specifies the instruction operand fields (the instruction format), e.g. "d,s,t" for destination and two source registers.

<**annotated.instruction.name**> is the name of an existing SimpleScalar instruction which will be annotated to pass the new instruction through the SimpleScalar assembler, e.g. "add" to replace "vadd" by "add/15:0(*)" where * is a number.

<**FU_class**> is the resource class required by this new instruction, e.g. "NewFU" to indicate a user defined functional unit described in the functional unit section.

<**iflags**> are instruction flags, used by SimpleScalar for fast decoding, e.g. "F_ICOMP" to indicate that this instruction only performs calculations.

<**odep1**>, <**odep2**> are two output dependency designators. They specify which registers are modified by the instruction and are used to determine data dependencies, e.g. "GPR(RD)" or "DSSAT_XX(SSAT_XX)" to indicate an existing general purpose register or a register defined with SSAT respectively.

<**idep1**>, <**idep2**>, <**idep3**> are input dependency designators. They specify which registers are read by the instruction, e.g. "GPR(RD)" or "DSSAT_XX(SSAT_XX)" to indicate an existing general purpose register or a register defined with SSAT respectively.

For more information on the SSIT configuration file, please refer to the SSIT manual [11].
From the SSIT configuration file, everything except <FU class> and <iflags> is used to construct the macros that are described below.

2.3.2.2 The Macros

This chapter discusses the general form of the macros. The general form of the macros is the same for all macros:

```
SSMT_INSTRUCTION_NAME_XXXXX(arguments)
```

with

- **SSMT**
  To make the macros distinguishable from other (user defined) macros and to avoid potential name conflicts, each macro created with SSMT starts with these letters.

- **INSTRUCTION_NAME**
  The name of the instruction in capitals and underscores in the place of dots to indicate which instruction the macro produces.

- **XXXXX**
  String of at most five character to indicate whether the arguments that are passed on are either a string containing the name of a register or a variable name containing a value. Each character can be one of four possibilities:

  - *Nothing*
    If the number of arguments is less than five because the instruction does not need five registers to operate, some of the letters are omitted and do not appear in the macro.
  - *V*
    If the argument type is supposed to be a variable name, the letter is a 'V'.
  - *R*
    If the argument type is supposed to indicate a register, the letter is an 'R'. The register name should be indicated by a string for correct results. More about this comes later.
  - *I*
    For the instructions that need an immediate operand, there is the letter 'I' to indicate that the variable should be an immediate value.

  To determine which character is needed, SSMT takes the <operands> section of the MACHINE.DEF section to determine whether a character should be V/R/nothing or I. When it is I it is printed. When it should be 'V', 'R' or 'nothing', the input and output arguments sections are considered. When a register slot contains "DNA", nothing is needed and the character is omitted. When a register slot contains "SSAT" a register created with SSAT is meant to be used, the character becomes 'R'. When the register slot contains "DGPR" it means that either a normal variable name can be used, but also an aliased register. This splits the macro into two and produces one macro containing a 'V' and one containing an 'R' for the argument that is considered at the moment.

- **(arguments)**
  The arguments of an instruction should be stated in the same order as in the output and input argument sections of the MACHINE.DEF section of the SSIT configuration file, separated from each other with a comma. Whether these arguments are variable names, strings of characters or immediate values depend on what is needed by the variation of the macro that is used. When wrong arguments are used, the macro might still produce a result in the corresponding assembly file, but it will most likely produce an error during compilation because the compiler might try to parse a string as a variable or vice versa for example.

Due to some minor differences in the implementation of different types of instructions, there are two types of macros that, although they comply with the above definition, have a layout where only the name is variable. They are described below.

Comment Macros
To give the developer the possibility to comment on the written assembly code in the resulting assembly file, this macro is in every header file. It enables the user to put a comment into the assembly via C code. The only thing that needs to be passed on is a string with only the comment itself. This macro has 'COMMENT' in the place for the instruction name, nothing in the place of the X's and only one argument of type 'string'.

Load and Store Macros
Because instructions that work on memory have a specific form that is defined by the developers of SimpleScalar, and the assembly signature of these instructions is standard as well, we have decided that memory instructions should have a standard representation. Once a memory instruction is detected, a different branch in the SSMT program is taken to create these instruction macros. Memory instructions (either load or store) have the following two prototypes:

```
SSMT_INSTRUCTION_NAME_RV(register name, variable address)
SSMT_INSTRUCTION_NAME_RVI(register name, variable address,
  immediate offset value)
```

Memory instructions are assumed to be defined with SSIT because of new or aliased registers that are defined by SSAT. That means that the developer would have to give the target or destination register in assembly, something that translates to the 'R' in the macro, which means that a string containing a register name must be provided. The 'V' means that a variable name must be provided. In this case, the address of a variable must be provided which is placed in a general purpose register. With the second macro, the developer can give a small memory offset to a base address. This macro was introduced for optimization reasons. Whenever an address is given to the macro that is not already placed in a register because of previous usage, this address is calculated and placed in a register. When frequent small deviations from a base address are required in an application, this results in a lot of additions in assembly code. To remove these additions from assembly code, a small memory offset can be given to an address to preserve the base address and avoid the need for these extra instructions. More about this below.

2.3.2.3 Behind the Macros

Since the macros that are defined by SSMT are regular C macros, they are replaced during the pre-processing of a C file. All macros are replaced by extended in-line assembly because this is a means for the programmer to put assembly code into an assembly file through C code. The benefit of extended in-line assembly over normal in-line assembly is the possibility to use variable names in the in-line assembly code. This saves the programmer from finding the location of local variables in the stack in order to be able to use them or from doing register allocation by hand. The general form of the extended in-line assembly for the instructions is as follows [4]:

```
asm("instruction_name  X, X, X, X, X": outputs: inputs);
```

where

- **instruction_name**
  First comes the instruction in question.

- **X, X, X, X, X**
  Next come the outputs and inputs of the instruction when necessary. The X's here represent either an enumeration of variables which are further defined by the outputs and inputs or the strings that are given as arguments of the macro, e.g. when a macro contains three variable names the X's become "%0, %1, %2", when the macro contains three register names the X's become "*register_name*, *register_name*, *register_name*".

- **: outputs: inputs**
  When the arguments of the macro are variable names, they are represented in this list in correct order. When examining the header file, one will notice the '"=r"' or '"r"' in front of the variable name. The "=r" means that a variable is an output variable and is supposed to be placed in a register. The "r" means that the variable is an input variable that is supposed to be placed in a register.

The two special cases for the comment macro and macros for memory instructions comply with the above but have a standard form.

Comment Macro
The comment macro is replaced by:

```
asm("#"comment)
```

Where "comment" is the string that is put in the macro. It is not necessary to include the assembly comment sign ('#' in this case) in the comment line itself but it will not generate an error when it is.

Load and Store Macros Because memory instructions always have the same kind of macro, the in-line assembly is also generally the same. The two macros that are given above are replaced by one of the following two lines respectively:

```
asm("instruction_name   " register_name " 0(%0) ":: "r" (variable_address))
asm("instruction_name   " register_name " %0(%1)":: "i" (immediate_offset_value),
       "r" (variable_address))
```

Where *register_name* is a C string containing the name of a register, *variable_address* is the address of a variable and *immediate_offset_value* is an immediate value. The percent signs in front of the 0 and 1 are to tell the compiler that that is the destination of the first or second value in the list that follows. Since there are no output variables that need to be translated because they are given by their register names, there are two colons behind each other between the assembly and the variable.

2.3.3 Design Choices

During the design of the SimpleScalar Macro Tool, some choices (mostly regarding the macros themselves) needed to be made. These choices are stated below:

- **C macros**
  The programming interface is implemented using C macros. An alternative to achieve approximately the same results would have been in-line functions. The benefit of in-line functions is that type checking is done at compile time. C macros were chosen in the end because it would make no difference to the source code that uses the macros, the end result would be that the macro is replaced by in-line assembly and even with the use of macros, errors would indicate that the programmer made a mistake in the arguments he is using. Even though the error is a parse error in case of the macros as opposed to a type checking error in case of in-line functions.

- **SSMT**
  Every macro starts with 'SSMT'. This is done because it would make the macros unique enough to avoid collisions with macros that are defined by the user or by another program.

- **Argument type identifier**
  In SimpleScalar, each instruction has at most two output registers and three input registers it depends on, therefore all macros have at most five final letters to indicate the types of the arguments they take. This is done because macros in C are simply replaced by something else, and therefore lack type checking. Since no type checking is done, these letters make the programmer more aware of what he is writing and what arguments he needs to pass on and decreases the chance of programmer errors.
  Another reason for adding the letters to the macros is the fact that with SSAT it is possible to extend the number of any register type that is already available. On top of that, it is possible to make aliases for existing registers to make programming easier. Therefore, for each variable that can be filled in as an argument of a macro, the programmer should also be able to fill in the name of an extended or aliased register.

- **Extended in-line assembly**
  Since the output of the macros needs to be assembly language and these macros should simplify the work of a programmer, extended in-line assembly was chosen because it directly generates assembly into the output assembly file and is also able to translate variable names used in the macro into a load or store operation where this is needed because a variable is not already placed into a register. The alternative to achieve the same result would have been GCC intrinsics [5]. Extended in-line assembly was chosen because it would produce the same results with less programming effort.

2.3.4 Example

This chapter shows and discusses an example of the workflow when using SSMT. The example consists of snippets taken from five different files. The first piece of code is the SSAT configuration file:

```
NEW_REGTYPE xx, 7, 8
```

This configuration file defines eight new registers with a width of seven bytes. The second snippet is taken from the SSIT configuration file.

```
#define AVG_IMPL { \
// This instruction calculates the average of its two inputs \
// and stores the result in its output register \
                                                \
  sword_t result = (GPR(RS)+GPR(RT)) >> 1;      \
  SET_GPR(RD, result);                          \
}

DEFINST( "avg", "d,s,t", "add",
         avg_FU, F_ICOMP,
         DGPR(RD), DNA, DGPR(RS), DGPR(RT), DNA
       )
#define VADD_IMPL { \
// This instruction takes in an array of 7 bytes and performs \
// a vector addition of the inputs                          \
                                                            \
  char result[7] = {0,0,0,0,0,0,0};                         \
  int i;                                                    \
                                                           \
  for (i = 0; i < 7; i++)                                   \
  {                                                         \
    result[i] = SSAT_DEM(RS)[i] + SSAT_DEM(RT)[i];          \
  }                                                         \
  SET_SSAT_DEM(RD, result);                                 \
}

DEFINST( "vadd", "d, s, t", "add",
           add_FU, F_ICOMP,
```

```
          DSSAT_XX(RD), DNA, DSSAT_XX(RS), DSSAT_XX(RT), DNA
      )

#define LDXX_IMPL { \
// This instruction loads a vector of 7 bytes from memory \
                                                          \
  const int vec_size = 7;                  \
    byte_t result[vec_size];                \
    enum md_fault_type fault;                \
    int i;                                  \
                                           \
    for( i=0; i< vec_size; i++ )             \
    {                                       \
        result[i] = READ_BYTE(GPR(BS) + OFS + i, fault);  \
        if( fault != md_fault_none )         \
            DECLARE_FAULT(fault);             \
    }                                       \
                                           \
    SET_SSAT_DEM( RT, result );              \
}

DEFINST( "ldxx",   "t,o(b)", "lb",
          ld_FU, F_MEM|F_LOAD|F_DISP,
          DSSAT_XX(RT),DNA, DNA,DGPR(BS),DNA
      )

#define STXX_IMPL { \
// This instruction stores an array of 7 bytes to memory \
    const int reg_size = 7;                                          \
    const byte_t* source = SSAT_DEM(RT);                             \
    enum md_fault_type fault;                                        \
    int i;                                                           \
                                                                     \
    for( i=0; i<reg_size; i++ )                                     \
    {                                                                \
        WRITE_BYTE( source[i], GPR(BS) + OFS +  i, fault );          \
        if( fault != md_fault_none )                                 \
            DECLARE_FAULT(fault);                                    \
    }                                                                \
}
  DEFINST( "stxx", "t,o(b)", "sb",
          st_FU, F_MEM|F_STORE|F_DISP,
          DNA, DNA, DSSAT_XX(RT), DGPR(BS), DNA
        )
```

Here, four different instructions are defined. The first is an instruction that calculates the average of its two input registers. Second an instruction that calculates an addition of two vectors. These vectors can be placed in the registers that are defined with SSAT. To be able to load or store a value to or from the new vector registers, a load and a store instruction are defined as well.

Next comes the entire header file that was generated by SSMT.

```
SSMT_COMMENT(comment) asm("#"comment)

SSMT_AVG_VVV(out0,in0,in1) asm("avg %0,%1,%2":"=r" (out0):"r" (in0),"r" (in1))
SSMT_AVG_VVR(out0,in0,in1) asm("avg %0,%1," in1 "":"=r" (out0):"r" (in0))
SSMT_AVG_VRV(out0,in0,in1) asm("avg %0," in0 ",%1":"=r" (out0):"r" (in1))
SSMT_AVG_VRR(out0,in0,in1) asm("avg %0," in0 "," in1 "":"=r" (out0):)
SSMT_AVG_RVV(out0,in0,in1) asm("avg " out0 ",%0,%1"::"r" (in0),"r" (in1))
SSMT_AVG_RVR(out0,in0,in1) asm("avg " out0 ",%0," in1 ""::"r" (in0))
SSMT_AVG_RRV(out0,in0,in1) asm("avg " out0 "," in0 ",%0"::"r" (in1))
SSMT_AVG_RRR(out0,in0,in1) asm("avg " out0 "," in0 "," in1 ""::)

SSMT_VADD_RRR(out0,in0,in1) asm("vadd " out0 "," in0 "," in1 ""::))

SSMT_LDXX_RV(out0,in0) asm("ldxx " out0 ", 0(%0)"::"r" (in1))
SSMT_LDXX_RVI(out0,in0,in1) asm("ldxx " out0 ", %0(%1)"::"i" (in1),"r" (in0))

SSMT_STXX_RV(out0,in0) asm("stxx " out0 ", 0(%0)"::"r" (in1))
SSMT_STXX_RVI(out0,in0,in1) asm("stxx " out0 ", %0(%1)"::"i" (in1),"r" (in0))
```

Every header file starts with a 'comment macro', followed by the macros generated according to the SSIT configuration file. Note that since the instruction calculating the average of two values (avg) can use either normal or aliased registers, this instruction ended up with eight different possibilities. The vector addition instruction can only operate on new vector registers because it operates only on registers that are defined with SSAT, therefore only one possibility exists. The load and store instructions have their pre-defined layout.

The following states a piece of executable C-code that demonstrates the use of several macros.

```
#include "ssmt.h"
#include <string.h>

char d[8] = {1,2,3,4,5,6,7,8};  // Global for illustrative purposes

int main(int argc, char ** argv)
{
  short a = 6, b = 4; // For the avg instruction
  char c[7] = {7,6,5,4,3,2,1}; // For the vector addition
  char result[7] = {0,0,0,0,0,0,0};  // Two arrays for the result
  char result2[9] = {0,0,0,0,0,0,0,0,0}; // of the vector addition

  SSMT_AVG_VVV(a, a, b);
  // Now 'a' contains the average of 'a' and 'b'

  SSMT_LDXX_RV("xx1", &c);
  SSMT_LDXX_RVI("xx2", &d, 1);
  // variable 'c' is loaded entirely into register 'xx1' directly
  // variable 'd' is loaded into register 'xx2' starting with the
```

```
  // second element for illustrative purposes.

  SSMT_VADD_RRR("xx3", "xx2", "xx1");
  // vectors 'c' and 'd' are added.

  SSMT_STXX_RV("xx3", &result);
  SSMT_STXX_RVI("xx3", &result2, 2);
// the result of the vector addition is stored directly into variable
// 'result' and with an offset of two elements into 'result2' for
// illustrative purposes.

  return 0;
}
```

Variables 'a' and 'b' are meant to demonstrate the 'avg' instruction. The average of their values replaces the original value of variable 'a'. The first load instruction loads the value of the array of seven characters into register 'xx1'. Newly defined registers are translated to arrays of characters (bytes) by SSAT. To obtain correct results, the address that is given to the macro should address at least the number of bytes that is needed to fill the defined register, in this case an array of seven bytes, however a 'long' or 'double', both consisting of eight bytes would also have been sufficient. The second load instruction also loads an array of seven characters, however it offsets the address by one byte. Stating **SSMT_LDXX_RR("xx2", &d[1])** would have given the same end result, however, it would have resulted in more elaborate, thus less efficient, assembly code. The store instructions operate the same way. Here, if the second store is replaced by **SSMT_STXX_RR("xx3", &result2[2])**, the end result would be the same, but the assembly code would be less efficient.

The last part of the example is a small piece of the assembly code that is generated by GCC with optimization level 2 enabled:

```
  li    $5, 0x00000006  # initialize variable 'a'
..
  li    $2, 0x00000004  # initialize variable 'b'
#APP
  avg   $5,$5,$2   # result of SSMT_AVG_VVV();
#NO_APP
  addu  $2,$sp,16  # calculate the address of variable 'c'
#APP
  ldxx  $xx1,0($2) # result of SSMT_LDXX_RV();
#NO_APP
  la    $6,d  # load the address of variable 'd'. Because
              # 'd' is global, its name is used.
#APP
  ldxx  $xx2, 1($6)    # result of SSMT_LDXX_RVI();
  vadd  $xx3,$xx2,$xx1 # result of SSMT_VADD_RRR();
#NO_APP
  addu  $3,$sp,24   # calculate the address of 'result'
#APP
  stxx  $xx3, 0($3) # result of SSMT_STXX_RV();
```

```
#NO_APP
  addu  $2,$sp,32   # calculate the address of 'result2'
#APP
  stxx  $xx3, 2($2) # result of SSMT_STXX_RVI();
#NO_APP
  move  $16,$3
..
```

As can be seen in the assembly code, every time a macro is used, it is automatically placed between *#APP* and *#NO_APP*. These markers indicate the start and end positions of assembly code that is inserted by an in-line assembly statement in C code.

2.4 Experimental Evaluation

To determine the impact on the performance of the use of the macros, some multimedia kernels that were optimized for the use of MMX technology are used. Originally, these kernels were compiled and, from GCC-compiled assembly code, rewritten and optimized by hand using the MMX instructions and registers that were added to SimpleScalar to comply with the workflow that is required to use SSIT and SSAT. Now, the original C code of the multimedia kernels is rewritten in C code to use the macros that are generated with SSMT and resemble the hand written assembly code as much as possible.
This section first discusses the general changes that are made to the original C code to use the macros of SSMT. After that the different kernels that were used for testing and evaluating SSMT are named and explained. Finally, the performance results of the kernels is discussed with a brief explanation of the differences between hand written and macro using code.

2.4.1 General Changes to Multimedia Kernels

For testing and evaluating SSMT, eight multimedia kernel files are analyzed and modified. The original C-code of these kernels showed a common layout that is described below:

```
global variables for input and output

kernel function

initialization function

main function
  call the initialization function

  call the kernel function

  print results on screen
```

The use of global variables for both input and output was striking since it is considered esthetically undesired by many programmers. It does however make programming

in assembly a lot easier since the variables can now (in assembly) be accessed by name because the assembler will turn these into the correct memory locations in due time.

To change the existing C code of the kernels, the assembly from the hand written kernel files was taken and transformed into macros that were to be put in the C code. From the hand written assembly files, everything that was not part of any MMX instruction was taken out and translated back to C code. This means that for example for-loops, if-else-statements, loop variables and pointers were identified and translated to C code. This is needed to keep certain hand written optimizations such as loop unrolling. These optimizations often make sense in MMX enabled code, but not in C code because MMX enables a single instruction on multiple data elements. A small example is given below. The following piece of code is the actual assembly code of one of the kernels. First some special instructions are called to display some statistics, then the address of the source and destination memory is loaded followed by the instructions that perform the actual calculations. Finally the same statistics as in the beginning are called to check the value with which they incremented.

```
getextinstr  # print the number of executed instructions
getcominstr  # print the number of committed instructions
getcycle     # print the cycle count so far

  la    $8, ImageX   # load array addresses
  la    $9, ImageY
  li    $10, 8       # number of rows (N), load row loop counter
  pxor  $mm7, $mm7, $mm7       # create a register containing the value '0'

$loop_col:
  li    $12, 8       # number of columns (M), load column loop counter

$loop_row:
  ld32  $mm1, 0($8)  # load half the first row of both input arrays
  ld32  $mm2, 0($9)  # into registers
  punpcklbw  $mm1, $mm1, $mm7  # unpack the four bytes that were loaded
  punpcklbw  $mm2, $mm2, $mm7  # to eight bytes
  paddw $mm1, $mm1, $mm2       # add the elements
  packuswb   $mm1, $mm1, $mm7  # pack the result to four bytes,
                               # rounding is done here
  st32  $mm1, 0($8)  # store the result in 'ImageX'

  addu  $8, $8, 4    # increment both array addresses by 4 bytes since
  addu  $9, $9, 4    # the previous four bytes have been processed
  subu  $12, $12, 4  # decrement the column loop counter by 4
                     # because 4 bytes have been calculated now
  bne   $0, $12, $loop_row  # continue with the rest of the row
                            # if not done yet.
$loop_row_finished:
  subu  $10, $10, 1         # decrement the row loop counter
  bne   $0, $10, $loop_col  # continue with the next row until the
                            # entire array is done
$loop_col_finished
```

```
getcycle     # print the cycle count thus far
getexinstr   # print the number of executed instructions
getcominstr  # print the number of committed instructions
```

This assembly code is optimized for the use of MMX instructions. The original C code of this kernel is shown below. Here, only the global variables and the actual kernel function are given:

```
const int N=8, M=8;

  unsigned char ImageX[8][8] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                                0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
                                0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
                                0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
                                0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
                                0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
                                0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
                                0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77,
                               };

  unsigned char ImageY[8][8] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                                0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
                                0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
                                0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
                                0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
                                0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
                                0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
                                0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77,
                               };

void add_images(void)
{
  int i, j, sum;
  for(i=0; i<N; i++)
  {
    for (j=0; j<M; j++)
    {
      sum = ImageX[i][j] + ImageY[i][j];
      if (sum>255) ImageX[i][j] = 255;
      else ImageX[i][j] = sum;
    }
  }
}
```

When this C code is translated into C code that uses the MMX instruction macros, the code becomes as is stated below. The macros and loop variables are written to resemble the hand written assembly code as close as possible. Only the contents of the function is shown, the global variables are the same.

```
int i, j;
```

```
char * IX, * IY;

SSMT_GETEXINSTR();
SSMT_GETCOMINSTR();
SSMT_GETCYCLE();

IX = &ImageX[0][0];
IY = &ImageY[0][0];

SSMT_PXOR_RRR("$mm7", "$mm7", "$mm7");

for (i = 0; i < N; i++)
{
  for (j = 0; j < M; j++)
  {
    SSMT_LD32_RV("$mm1", IX);
    SSMT_LD32_RV("$mm2", IY);
    SSMT_PUNPCKLBW_RRR("$mm1", "$mm1", "$mm7");
    SSMT_PUNPCKLBW_RRR("$mm2", "$mm2", "$mm7");
    SSMT_PADDW_RRR("$mm1", "$mm1", "$mm2");
    SSMT_PACKUSWB_RRR("$mm1", "$mm1", "$mm7");
    SSMT_ST32_RV("$mm1", IX);
    IX += 4;
    IY += 4;
  }
}

SSMT_GETCYCLE();
SSMT_GETEXINSTR();
SSMT_GETCOMINSTR();
```

As can be seen in the above example, every MMX-like instruction is replaced by a macro in C-code and all surrounding assembly is translated into the statements they represent. This is done to ensure that performance is not affected because of optimizations that are done in assembly itself.

2.4.2 Multimedia Kernels

In order to make a proper comparison a total of eight multimedia kernels were modified to use the macros from the header file that was created with SSMT. All kernels are optimized for the use of MMX-like instructions and thus work on vectors of four or eight bytes at a time. Three of the kernels actually have two forms that are basically the same. One of the two is a small version that works on matrices of eight by eight or eight by sixteen bytes. The other one is a large version that works on vectors and matrices with much larger dimensions to more accurately simulate the use in real life where different dimensions can make a difference in cache behaviour for example. Besides that, the larger dimensions reduce the impact of overhead which makes the relative performance different for both kernels. The kernels that were used are described below.

- **Add Image**
  This kernel adds two images of the same size and stores the result in one of the input images. The images are matrices of unsigned bytes thus limiting the values between 0 and 255. Saturation is done in the MMX instructions.
  The small version of the 'Add image' kernel adds two images of 8 x 8 bytes.
  The large version of the 'Add image' kernel adds two images of 704 x 576 bytes. These images are converted from Windows Bitmap files and are loaded by the initialization part of the kernel.

- **DCT**
  The DCT kernel computes the Discrete Cosine Transformation. In this particular case the algorithm works on a matrix of 704 x 576 bytes filled with pseudo-random values and stores the result in a different one.

- **IDCT**
  This kernel calculates the Inverse Discrete Cosine Transformation, which is the inverse of the Discrete Cosine Transformation. This kernel operates on a matrix of 704 x 576 bytes filled with pseudo-random values. The result is stored in a different matrix than the original one.

- **Matrix Transpose**
  The transpose of a matrix is calculated with this kernel. Like the kernel that adds two images this kernel has two versions:
  The smaller version transposes an 8 x 8 matrix, the larger version transposes a matrix of 704 x 576. The values in the small matrix are hard-coded, the large matrix is filled at run-time with pseudo-random values.

- **Matrix Vector Multiplication**
  This kernel has a version that works on a matrix of 8 x 16 bytes and a vector of 16 bytes, the small version. The large version works on a matrix of 704 x 576 bytes and a vector of 576 bytes. The values of the small matrix and vector are hard-coded, the values of the large matrix and vector are filled with pseudo-random values at run-time.

2.5 Experimental Results

This section discusses the results of the comparison between hand written code and C code using macros. The total comparison is between (1) the original C code, (2) C code optimized with GCC optimization level 2, (3) hand written assembly, (4) C code using macros, (5) C code using macros and optimized with GCC optimization level 2 and (6) C code using memory macros with offsets and optimized with GCC optimization level 2.

The comparison came in three different stages. In the first stage only the original C code, compiled without optimizations, the hand written assembly and the C code with macros, again compiled without optimizations, were compared to each other. After that, optimization level 2 of GCC was enabled and again the results were compared. Finally

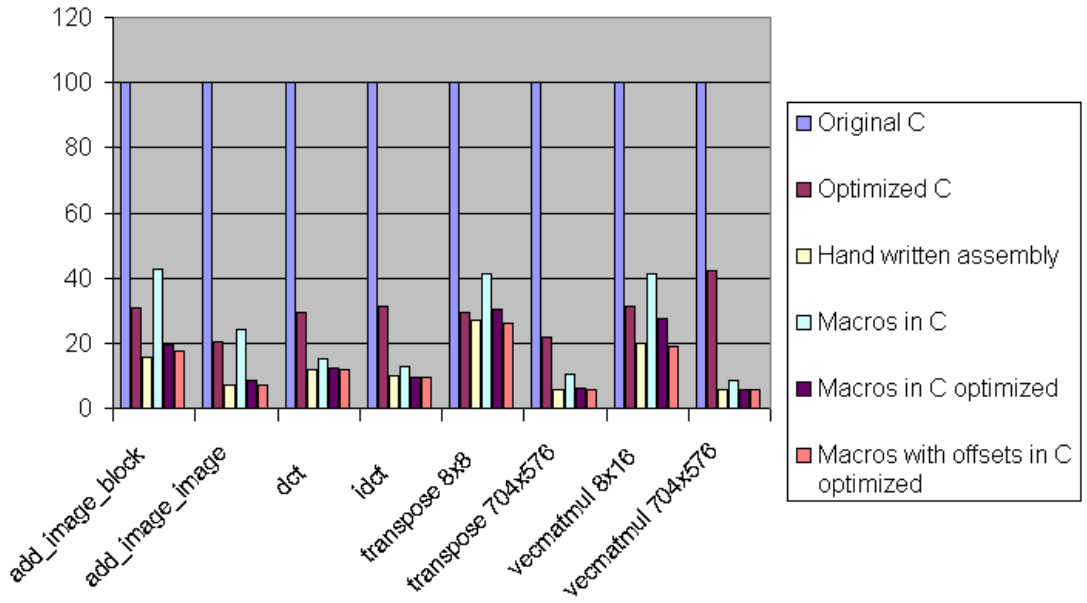

Figure 2.2: Cyclecount needed to execute different multimedia kernels, normalized to the number of cycles needed to execute the original unoptimized C code

after comparing the generated assembly with the hand written assembly, the load and store macros with offsets were added to SSMT and one last time a comparison was made between the generated and hand written assembly. The results of the performance comparison are presented after that. Last results of the comparison between hand written assembly and generated assembly are discussed.

2.5.1 Differences in Performance

This section discusses the results of the performance comparison between the six versions of the same kernel, all of them mentioned above. As a measurement, the total number of cycles that were needed to complete the task were taken and compared to each other. The results are normalized to the number of cycles that were needed to run the original, unoptimized C-code and shown in Figure 2.2.

It is clear from the figure that hand written assembly with the use of the MMX-like instructions has a significant impact on the performance of the kernels. Also clear is that the use of the macros in C-code means a significant loss in performance compared to hand written code when the code is not optimized with GCC. The reason of this performance loss is discussed in Section 2.5.3. However, optimizing the macro using C-code with GCC gives performance that is fairly comparable to hand written assembly. With the exception of one case where the performance dropped to 88% of the performance of hand written assembly, C code using the macros has a performance that varies from 99% up

to 106% of the performance of hand written assembly where 100% is the performance level of hand written assembly. An explanation for this can be found below.

2.5.2 Differences in Assembly

This section discusses several reasons that cause a loss or increase in performance when the macros are used in C code. These reasons can all be found in differences between hand written assembly and assembly code that is generated by GCC. Therefore this section is shows many assembly code examples.

2.5.2.1 Array Indexing

All arrays, no matter how many dimensions they have, are one dimensional in memory. Array indexing in C is a nice way to let the compiler figure out how to arrive at the correct memory address. Performance however is lost during this process, in normal C-code as well as in the C-code that uses the macros. Shown below are two pieces of plain C-code that both access an array. The first piece uses automatic array indexing which is available in C. The second piece takes the address of the array and does manual array indexing.

```
for (i = 0; i < N; i++)
{
  for (j = 0; j < M; j++)
  {
    array[i][j] = 5;
  }
}
```

And

```
array_ptr = &array[0][0];  // The index is unnecessary but clearer
for (i = 0; i < N; i++)
{
  for (j = 0; j < M; j++)
  {
    *array_ptr = 5;
    array_ptr++;
  }
}
```

Both pieces of code will have the same result when executed (even the order in which the array is accessed is the same), but different number of execution cycles. This is because the first piece of C-code is translated into the following piece of assembly code (the array is assumed to be a global variable and an array of bytes):

```
  li     $10, 5
  li     $2, N
  move   $4, $0      # outer loop variable
  blez   $2, $in_between
```

```
  la     $6, array
  li     $3, M

$outer_loop:

  move   $5, $0     # inner loop variable
  blez   $3, $end
  multi  $7, $4, N

$inner_loop:

  addu   $8, $6, $7 # calculate the address
  addu   $8, $5, $8

  sw     $10, 0($8) # store the 5

  addu   $5, $5, 1
  slt    $9, $5, $3
  bne    $9, $0, $inner_loop

$in_between:

  addu   $4, $4, 1
  slt    $9, $4, $2
  bne    $9, $0, $outer_loop

$end:
```

The second piece of code is translated into the following (again, the array is assumed to be global and to be an array of bytes):

```
li     $10, 5
  li     $2, N
  move   $4, $0      # outer loop variable
  blez   $2, $in_between

  la     $6, array
  li     $3, M

$outer_loop:

  move   $5, $0     # inner loop variable
  blez   $3, $end

$inner_loop:

  sw     $10, 0($6) # store the 5
  addu   $6, $6, 1  # increase pointer

  addu   $5, $5, 1  # increase inner loop variable
```

```
  slt    $9, $5, $3
  bne    $9, $0, $inner_loop

$in_between:

  addu   $4, $4, 1  # increase outer loop variable
  slt    $9, $4, $2
  bne    $9, $0, $outer_loop

$end:
```

The first example uses two additions (two lines beneath *$inner_loop*) for each element and a multiplication (line above *$inner_loop*) extra for each row to calculate the address of the variable on which the operation takes place. The second example on the other hand only uses one addition (second line beneath $inner_loop) to calculate the address that is needed. This difference does not seem to be much but every instruction counts and for large arrays, this can count up to quite some loss in performance.

2.5.3 Memory Operations

To illustrate the huge performance loss in the first stage of the development of SSMT and the small differences between the second and third state of the development, this section describes the resulting assembly code that is generated when the macros from the different stages are used. First, the macros that do not give offsets and are compiled without optimizations are discussed, followed by the same macros but with GCC optimization level 2 enabled. Last, the macros with offsets that are compiled with GCC optimization level 2 enabled are discussed. The following paragraphs focus on the following piece of hand written assembly which is assumed to be part of a loop and translated into macro using C code (the dots indicate the possibility of code in between):

```
...
  la    $1, ImageX
  la    $2, ImageY
  la    $3, ImageZ
...
  ld64  $mm1, 0($1)
  ld64  $mm2, 8($1)

  ld64  $mm3, 0($2)
  ld64  $mm4, 8($2)

  ld64  $mm5, 0($3)
  ld64  $mm6, 8($3)
...
  ld64  $mm7, 0($1)
  ld64  $mm8, 8($1)
...
```

In this case the arrays 'ImageX', 'ImageY', 'ImageZ' are global two-dimensional arrays, the 'ld64' instruction is an MMX-like instruction and the '$mm*' registers are 8

byte wide. While reading the following paragraphs it needs to be kept in mind that the pattern above is meant to be repetitive.

GCC Output Without Optimization The content of this paragraph will be about the output of GCC without optimizations enabled as well as the way the optimized assembly is translated into macros. Firstly the piece of hand written assembly above will be translated into C code with the use of the macros that are generated by SSMT:

```
...
  SSMT_LD64_RV("$mm1", ImageX);
  SSMT_LD64_RV("$mm2", ImageX + 8);

  SSMT_LD64_RV("$mm3", ImageY);
  SSMT_LD64_RV("$mm4", ImageY + 8);

  SSMT_LD64_RV("$mm5", ImageZ);
  SSMT_LD64_RV("$mm6", ImageZ + 8);
...
  SSMT_LD64_RV("$mm7", ImageX);
  SSMT_LD64_RV("$mm8", ImageX + 8);
...
```

Which in turn is compiled by GCC into:

```
...
  la    $2, ImageX
  ld64  $mm1, 0($2)
  la    $2, ImageX
  addu  $2, $2, 8
  ld64  $mm1, 0($2)
...
  la    $2, ImageY
  ld64  $mm1, 0($2)
  la    $2, ImageY
  addu  $2, $2, 8
  ld64  $mm1, 0($2)
...
  la    $2, ImageZ
  ld64  $mm1, 0($2)
  la    $2, ImageZ
  addu  $2, $2, 8
  ld64  $mm1, 0($2)
...
  la    $2, ImageX
  ld64  $mm1, 0($2)
  la    $2, ImageX
  addu  $2, $2, 8
  ld64  $mm1, 0($2)
...
```

Since every memory instruction is translated into two or even three separate instructions every single time, the performance of the resulting program is, depending on the number of memory instructions, much less than the hand written assembly. This can also be seen in Figure 2.2.

GCC Output With Optimizations Enabled To optimize a piece of C-code, with or without macros in it, it is unnecessary to change the code itself. Therefore, the C-code that is used to optimize is the same as the C-code that was used for the unoptimized version of the kernel. This time however the resulting assembly looks like the following:

```
...
  la     $2, ImageX
  la     $3, ImageY
  la     $4, ImageZ
...
  addu   $5, $2, 8
  addu   $6, $3, 8
  addu   $7, $4, 8
...
  ld64  $mm1, 0($2)
  ld64  $mm2, 0($5)
...
  ld64  $mm3, 0($3)
  ld64  $mm4, 0($6)
...
  ld64  $mm5, 0($4)
  ld64  $mm6, 0($7)
...
  ld64  $mm7, 0($2)
  ld64  $mm8, 0($5)
```

The piece of code above is still not as efficient as the hand written assembly, but instead of every memory instruction being translated into two or three separate instructions, now every memory instruction that operates on the same memory address shares the same load address and addition instruction.

GCC Output With Special Memory Macros After discovering that extended in-line assembly does not produce offsets along with addresses in registers, the final addition to SSMT was made. This final adjustment resulted in one extra macro for each instruction that operates on memory (load and store instructions). This last change made it necessary to change the piece of code to be rewritten to:

```
...
  SSMT_LD64_RV("$mm1", ImageX);
  SSMT_LD64_RVI("$mm2", ImageX, 8);
...
  SSMT_LD64_RV("$mm3", ImageY);
  SSMT_LD64_RVI("$mm4", ImageY, 8);
...
```

```
  SSMT_LD64_RV("$mm5", ImageZ);
  SSMT_LD64_RVI("$mm6", ImageZ, 8);
...
  SSMT_LD64_RV("$mm7", ImageX);
  SSMT_LD64_RVI("$mm8", ImageX, 8);
...
```

The resulting assembly code will be very much like the hand written assembly code (the code is the same in this example). However, there will still be some differences in the resulting assembly in general that will be discussed in the next paragraph.

Other Code Segments Not all differences in assembly (and thus differences in performance) can be explained by the above. Although quite some effort is put into generating the same assembly code, if only for the (in this case) MMX-like part of the assembly, there are still differences between the hand written assembly code and the generated code. These differences are beforehand not determinable and depend on the way GCC interprets the written C-code. It is however possible to give some probable reasons to indicate the nature of these differences.

- **FOR Loops**
  FOR loops are statements that are widely used by programmers. The way a FOR loop is translated into assembly is in general dependent on the way it is written in C. An initialization or loop condition with a certain variable for example will give a different result than the same thing with values that are known at compile time. GCC makes these kind of decisions based on what is known at compile time, without regards to the context of the loop (something that the compiler can not recognize because it lacks the intelligence to do so). A human programmer however, can take the context into account and therefore write a more efficient FOR loop.

- **Registers**
  When it comes to registers, the compiler is most often better than human programmers. Where GCC has no problem at all taking all available registers into account, a human programmer has far more difficulties in doing so. For that reason a human programmer will most likely settle with a less optimal solution and take a decrease in performance for granted to be able to write code that is more understandable (and thus easier to write and debug), where GCC writes code that is less readable but more efficient when it comes to registers.

- **Other Statements**
  Unlike the FOR loop that in most cases will give less performance when automatically generated, or the register allocation that will give a better performance when done automatically, there are also things done automatically that in general give a better performance, but not always. Examples of this kind of behavior are unfortunately not available.

2.6 Merger Between SSMT and SSIT: SSMIT

Upon completion of the SimpleScalar Macro Tool it became clear that the time the toolchain of SSIT, SSAT and SSMT used could fairly easily be decreased by merging SSIT and SSMT. This merged version is called the SimpleScalar Macro and Instruction Tool (SSMIT) and is capable of changing the source files of SimpleScalar to include the newly designed instructions and functional units, producing a header file with macros and producing annotated instructions in generated assembly. SSMIT requires a SSIT configuration file to work and needs to be executed before compiling C code into assembly. Its location in the toolchain is depicted in Figure 2.3.
A similar fusion of two of the tools was already implemented by merging SSIT and SSAT into the SimpleScalar Instruction and Architecture Tool (SSIAT). Merging SSMT with SSAT has not been researched yet, however, we believe that the merger of SSMT with SSAT will be substantially more difficult than the merger between SSMT and SSIT. This because SSAT works with the allocation of registers throughout a program, which is information that is only available after compile time.
This section first discusses the usage of SSMIT followed by the inner workings. This chapter is concluded with a small discussion on the impact of SSMIT on the compile time of a program.

2.6.1 Usage of SSMIT

As are SSIT and SSMT, SSMIT is a command line tool and has the following usage description:

```
--== SimpleScalar Macro and Instruction Tool  ==--

USAGE: ssmit [options]
Options:
-c <filename> or --config <filename>
  Specify the configuration file name    Default:  "ssit_config.cfg"
-o <filename> or --outfile <filename>
  Specify the output file name           Default:  "ssmt.h"
-u <true/false> or --update <true/false>
  Update the SimpleScalar source files   Default:  "false"
-s <dirname> or --ssdir <dirname>
  Specify the SimpleScalar source directory Default: "simplesim-3.0"
-h  Print  usage instructions
```

This usage description is actually the usage description of both SSIT and SSMT in one with the exception of the option in SSIT that allows the user to change the input and output assembly file names.

2.6.2 Workings of SSMIT

The inner workings of SSMIT are virtually the combined workings of SSIT and SSMT. Its source code is a partial copy of the source code of SSIT, a full copy of SSMT and a small addition to fulfill the data format needs of SSMT. SSMIT is, like SSMT, used

before compiling C code for the use with SimpleScalar. As mentioned before, SSMIT can perform three tasks: (1) Change the source files of SimpleScalar, (2) Produce a header file containing macros and (3) Produce assembly code containing annotated instructions.
Changing the source files of SimpleScalar is done in the same way as it is done by SSIT. SSMIT takes in the SSIT configuration file and inserts code into key areas of the SimpleScalar source files. After changing the source files SimpleScalar needs to be recompiled. Producing a header file containing macros is done almost the same way as it is done in SSMT. SSMIT takes in the SSIT configuration file and according to the information that is within the configuration file, several macros are defined. However, instead of inserting the readable instruction name into the macro, the annotated instruction is inserted. Therefore a macro that is defined with SSMT as

```
SSMT_AVG_VVV(out0,in0,in1)  asm("avg %0,%1,%2":"=r"(out0):"r"(in0),"r"(in1))
```

becomes, defined with SSMIT

```
SSMT_AVG_VVV(out0,in0,in1)  asm("add/15:0(1) %0,%1,%2":"=r"(out0):"r"(in0),"r"(in1))
```

Thereby producing annotated instructions in generated assembly when the C code that is using the macros are compiled. This is not very user friendly though and SSMIT should not be used to produce the header file whenever the resulting assembly code needs to be inspected later.

2.6.3 Impact of SSMIT

The use of SSMIT in the toolchain around SimpleScalar is shown in Figure 2.3. Unfortunately, the impact of SSMIT on the total compile time of a program was not tested. This because it is fairly impossible to give an accurate number for this impact. The time it takes SSIT to change all the entries in an assembly file depends on a combination of the size of a file, the number of total possible instructions for which each instruction in the file needs to be tested and the total number of actual instructions that need to be replaced.
Although a precise number for the impact of SSMIT cannot be given one main advantage of SSMIT however is that whenever SSAT does not need to be used, the compilation of any program can be done in one single step instead of in three steps (compile to assembly, run through SSIT, SSAT or SSIAT, assemble). For small programs, this is not that much of a difference, but for large programs it significantly simplifies the compilation process.

2.7 Concluding Remarks

In this chapter, the SimpleScalar Macro Tool (SSMT) is discussed. This tool is designed and developed to cooperate with the two already available tools called SimpleScalar Instruction Tool (SSIT) and SimpleScalar Architecture Tool (SSAT) that were designed to

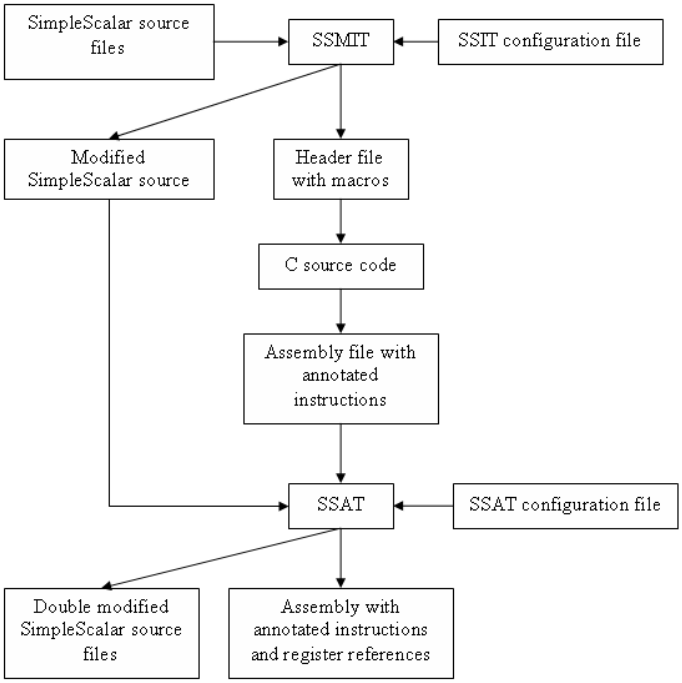

Figure 2.3: Toolchain surrounding SimpleScalar, involving SSMIT and SSAT

simplify the use of SimpleScalar. These tools ease the implementation of new functional units, instructions and architectural elements into the SimpleScalar processor simulator and make the use of the new instructions and architectural elements in assembly language manageable by enabling a developer to write readable instructions in assembly code instead of the annotated instructions that are used by SimpleScalar. SSMT relieves a developer from programming large parts of applications in assembly, something that is necessary at this moment whenever one or more new instructions are introduced into the SimpleScalar processor, by providing him with a macro programming interface that enables him to program new assembly instructions in C programming language instead of in assembly language. The performance overhead of this programming interface was tested and found to be minimal. In some cases the performance of a program even increased compared to hand written assembly by at most 6%.
In addition to SSMT, a fusion of SSIT and SSMT was implemented to speed up development even further. This fusion is called the SimpleScalar Macro and Instruction Tool (SSMIT). This tool can perform all actions that SSIT and SSMT combined would perform except from the ones that are no longer needed because of the merger. Where SSMT would insert readable instructions into assembly code for SSIT to replace them by annotated instructions, SSMIT inserts the annotated instructions immediately, making a pass through SSIT unnecessary. This has advantages and disadvantages: When compiling (large) applications which do not need a pass through SSAT, an intermediate step involving assembly code is unnecessary thereby speeding up the compilation pro-

cess. When debugging however, the resulting assembly code becomes almost unreadable, forcing the developer back to using SSIT and SSMT separately.
In retrospect, the development of SSMT could have gone faster if I had not made the mistake of starting from scratch and reused the part of SSIT that reads the configuration file and converts it into a usable data format. Also, getting the macro programming interface to perform as well as it does, involved the comparison of a lot of hand written assembly code with generated assembly code to discover the source of performance differences and find ways to make these differences disappear. In the end however, I am confident that SSMT and SSMIT can be used to speed up development processes and optimizations of applications that require the use of new instructions.

Background on Cell and CellSim

3

After the completion of the SimpleScalar Macro Tool, we focused on a different project. The main goal of this project was to develop a profiling capability for a simulator that is developed by the Barcelona Supercomputing Center and is called CellSim. This simulator is being developed to cycle accurately simulate the Cell Broadband Engine (Cell BE) that has jointly been developed by Sony, Toshiba and IBM and is known in the consumer market by its appearance in the Playstation 3. Because the chapters that describe the work that has been done during this project contain several abbreviations and require some knowledge about both Cell BE and CellSim, this chapter first describes the Cell BE and then CellSim.

3.1 Cell Processor

The Cell processor mainly consists of five elements (see Figure 3.1):

- Power Processing Element (PPE)
- Synergistic Processing Element (SPE)
- Element Interconnect Bus (EIB)
- I/O interface
- Memory subsystem

The PPE is the main core, controlling the workflow and the eight SPEs as well as running the operating system of the programmers choice. The SPEs are processors that are designed with vector operations and optimized for processing large amounts of data. Although the PPE is considered the core of Cell and runs the operating system, the SPEs do most of the work. All elements of the Cell BE are interconnected by the EIB which has been optimized to support the high bandwidth that is needed to process large amounts of data. The I/O interface is the interface for the processor to the outside world and the Memory subsystem is the controller for memory operations. Below the PPE, SPE and the EIB are described in more detail.

3.1.1 PPE

The Power Processing Element is a 64-bit Power-Architecture-Compliant core with a 23-stage pipeline and a dual-issue design that does not dynamically reorder instructions at issue time. The PPE is the controller for the eight SPEs which do most of the computational work.

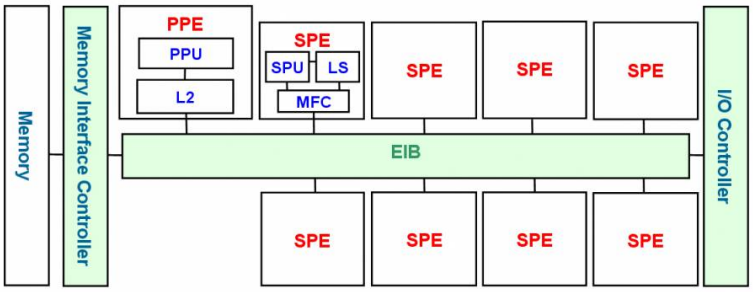

Figure 3.1: Overview of the Cell Processor, taken from [3]

The PPE contains 32 KB Level 1 instruction and data caches and a 512 KB Level 2 cache. The processor provides two simultaneous threads of execution within the processor and can be viewed as a two-way multiprocessor with shared dataflow. This gives software the effective appearance of two independent processing units. The processor is composed of an instruction unit (IU), a fixed-point execution unit (XU) and a vector scalar unit (VSU).
The IU is responsible for the instruction fetch, decode, branch, issue and completion. It fetches four instructions per cycle per thread into an instruction buffer from which the instructions are dispatched. After decoding and dependency checking, instructions are dual-issued to an execution unit. The outcome of a branch is predicted by a 4-KB by 2-bit branch history table with 6 bits of global history per thread.
The XU does the fixed-point instructions and memory operating instructions. It consists of a 32-bit by 64-bit general purpose register file per thread, a fixed-point execution unit and a load/store unit. The load/store unit consists of the Level 1 data-cache, a translation cache, an eighty-entry miss queue and a 16-entry store queue. The load/store unit supports a non-blocking L1 D-cache which allows cache hits under misses.
The VSU takes care of all vector and floating point instructions. Its floating-point execution unit consists of a 32- by 64-bit register file per thread, as well as a ten-stage double-precision pipeline. The VSU vector execution units are organized around a 128-bit dataflow. The vector unit contains four subunits: simple, complex, permute, and single-precision floating point. All instructions are 128-bit SIMD with varying element width (2×64-bit, 4×32-bit, 8×16-bit, 16×8-bit, and 128×1-bit) [7].

3.1.2 SPE

The Synergistic Processing Element is the main processing element of the Cell Processor and consists mainly of three components: A local store, the Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). The local store is the largest component in the SPE and is used to store data as well as instructions. To minimize area it is implemented by a single-port SRAM cell of 256 KB.
The SPU implements a new instruction set architecture that is optimized for power and performance on computing-intensive and media applications. All execution units of the SPU are organized around a 128-bit dataflow and contain a large register file with

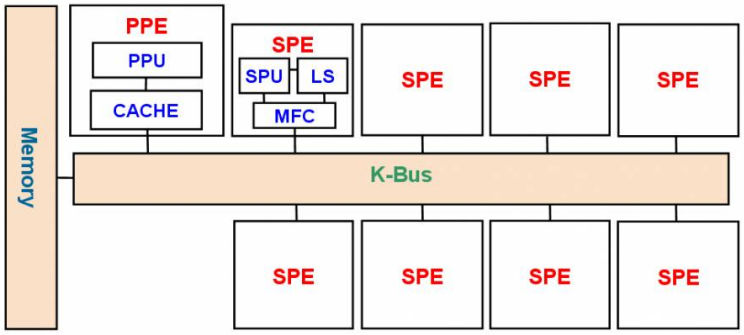

Figure 3.2: Overview of the Cell Simulator components, taken from [3]

128 entries. All instructions are 128-bit SIMD with varying element width (2×64-bit, 4×32-bit, 8×16-bit, 16×8-bit, and 128×1-bit) and fully pipelined with the exception of two-way SIMD double-precision floating point instructions. Data and instructions are tranfered between local memory and system memory by asynchronous coherent DMA commands that are executed by the MFC [7].

3.1.3 EIB

The Element Interconnection Bus allows for communication between all elements in the Cell processor. It consists of one address bus and four 16-byte wide data rings of which two run clockwise and two run counter-clockwise. Each ring can sustain up to three concurrent data transfers as long as their paths do not overlap and the total number of transfers do not exceed eight. The EIB operates at half the system clock speed which means that the theoretical maximum bandwidth of the EIB at 3.2 GHz is 204.8 GB/s.

3.2 Cell Simulator

With the purpose of having a heterogeneous chip multiprocessor research infrastructure, a cycle-accurate simulator has been developed by the Barcelona Supercomputing Center that simulates the Cell BE processor. This simulator is called CellSim and it is based upon the UNISIM environment [13]. UNISIM proposes a common modular simulation approach and defines several libraries and a common communication protocol. UNISIM follows from the trend to design processors by breaking them down into modules that can be designed separately, something that can now be reflected in a simulator as well. For most of the modules of the Cell BE, a module in CellSim is created using the UNISIM environment. Figure 3.2 shows that the PPU and SPUs each have their own module which contain the same elements as the Cell BE. The EIB of the Cell BE is implemented as a K-bus in CellSim which makes it more flexible and scalable. The I/O controller of the Cell BE is omitted in CellSim as well as the MIC. This means that the memory, which also has a module of its own, is directly connected to the EIB.

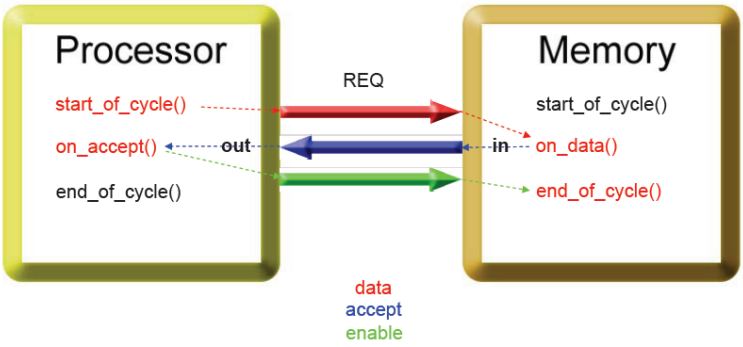

Figure 3.3: Communication protocol between the modules of CellSim, taken from [3]

In addition to the UNISIM communication protocol, the developers of CellSim have defined a communications interface which is used in every inter-modular communication. This protocol is shown in Figure 3.2. Because all modules are designed to communicate by the defined protocol, interchanging modules with optimized versions or for research purposes becomes much simpler and thus less time consuming.

3.3 Conclusion

In this chapter we have described the Cell BE processor and CellSim to provide the background that is needed to fully understand the used abbreviations and descriptions in the following chapters.

4 Instructions for CellSim

4.1 Introduction

Although CellSim is operative at the time of writing, it is still under development and therefore only partially functional. Several instructions from the SPU instruction set for example, were still not implemented. This chapter describes the implementation of the missing instructions, which consisted mainly of writing a C++ class for each instruction. In this chapter, Section 4.2 gives an overview of the file organization and background information that is relevant to writing missing or new instruction proposals for CellSim. Section 4.4 gives a list of the implemented instructions and Section 4.3 describes the method that was used to implement the instructions and gives a small example. More information about CellSim can be found in Section 3.2 or at [3].

4.2 Information for Writing SPU Instructions

To be able to write new instructions, it is required to have information about the directory structure and files that require attention when writing new instructions for the CellSim SPU. Also important is knowledge about the instruction format of Cell and the use of its registers. The directory structure and important files are described in Subsection 4.2.1. The instruction format and the use of the registers of Cell is described in Subsection 4.2.2.

4.2.1 File and Directory Structure of CellSim

To be able to write new instructions, it is required to know something about the class organization of CellSim and the directory structure that is used to order the source files. The directory structure and files that are important when implementing instructions are shown below:

```
cell-sim-unstable/
|-library/        Contains the source files of the Processor and Cache
| |...            modules. After compilation the files corresponding to
|                 the configuration are placed in /modules/processor/.
|
|-native/         Contains all files related to code native to cell.
| |...            Source code of the libspe that is needed to compile
|                 programs for CellSim and some examples of Cell programs.
|
|-modules/        Contains all source files from which CellSim is compiled.
  |-src/
    |-processor/    Contains the PPE, EIB, and Memory files
```

```
|-accelerator/  Contains the SPE files
  |-commands/       Contains MFC command classes
  |-instructions/   Contains the instruction classes
    |-RI8/          Contains the instructions of format RI8
    |-RI10/         Contains the instructions of format RI10
    |-RI16/         Contains the instructions of format RI16
    |-RI18/         Contains the instructions of format RI18
    |-RR/           Contains the instructions of format RR
    |-RRR/          Contains the instructions of format RRR
```

As can be seen in the above, all instructions for the SPU are gathered in a single directory and ordered by the format of the instruction. More about the instruction format in the next subsection.
Besides the files that contain the instructions themselves, several other files are important to be able to successfully implement new instructions. These files with an explanation of their importance is stated below:

- **BitHandler.hxx**
  The Cell BE uses bigEndian when representing data in registers. CellSim is written for littleEndian machines. Therefore, when implementing instructions the endian must be changed. The BitHandler class implements methods to change the endian for all configurations of a register as well as a method to extract bits from a 32-bit word.

- **RegisterFile.cxx/hxx**
  The RegisterFile class implements the register file of CellSim and is used to obtain the contents of a register.

- **Register.hxx**
  A class template that implements a single register in all possible configurations.

- **/instructions/Instruction.hxx**
  A class that contains all basic members and method for all instructions. Every instruction inherits directly or indirectly from this class.

- **/instructions/BranchInstruction.hxx**
  Class that contains methods that are specific to branch instructions.

- **/instructions/ChannelInstruction.hxx**
  Class that contains methods that are specific to channel instructions.

- **/instructions/HaltStopInstruction.hxx**
  Class that contains methods that are specific to halt/stop instructions.

- **/instructions/MemoryInstruction.hxx**
  Class that contains methods that are specific to memory instructions.

- **/instructions/InstructionInfo.hxx**
  Macros and type definitions that are related to instructions, opcode and format.

- **/instructions/InstructionContainer.cxx/hxx**
  Class containing an array with all instructions, indexed by their opcode.

- **/instructions/Instructions.h**
  Method to put all instructions into the InstructionContainer class.

- **/instructions/Instructions_inc.h**
  Listing of includes of all instruction class files.

These files are important when implementing instructions for which intrinsics are already implemented in the compiler that is provided within the development kit. To implement entirely new instructions, more files need to be used or modified. This, however, goes beyond the scope of this text.

4.2.2 Instruction and Register Format

All instructions of the Cell BE SPU are 32-bit wide and have either one of the formats that are shown in Figure 4.1. In this figure, RA, RB and RC are the addresses of the input registers of an instruction. RT is the address of the output register. Immediate values are incorporated in the instruction and are, depending on the format of the instruction, 7, 10, 16 or 18 bits wide.
As can be seen in Figure 4.2.2, the register address fields are 7 bits wide. This is necessary because each SPU is equipped with 128 registers that are all 128-bit wide. The format of the registers is not static and can be used in either one of the following configurations: 1×128-bit (quadword), 2×64-bit (2 doublewords), 4×32-bit (4 words), 8×16-bit (8 halfwords) or 16×8-bit (16 bytes) depending on the used instruction.
Sometimes, like in the case of conditional branch instructions, an instruction does not operate on all elements of a register. Instead it uses all, 16 or 8 bits of the 32-bit wide leftmost 'word' of a register which is called the "preferred slot" [2].

4.3 Implementing Instructions

To be able to successfully implement each instruction, it is required to have certain information about an instruction. After obtaining the required information, the actual implementation can be done. The information that is required along with its use is described in Subsection 4.3.1. The procedure that can be followed to write an instruction and insert it into CellSim along with an example is given in Subsection 4.3.2.

4.3.1 Required Information Before Implementing Instructions

Before implementing an instruction, four pieces of information that define the instruction need to be present:

- name/mnemonic

- opcode

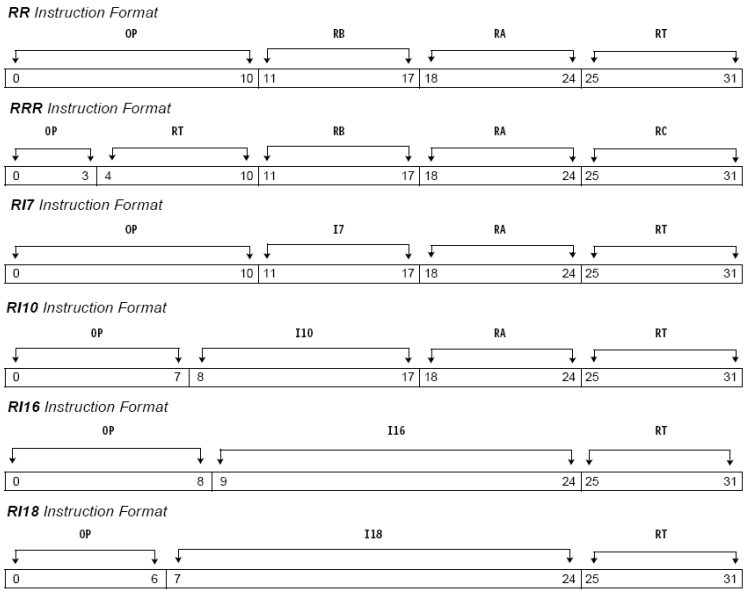

Figure 4.1: The seven different instruction formats, taken from [2]. 'OP' is the opcode or identification number of an instruction. 'I*' is an immediate field containing a value that can be used in the instruction itself. RA, RB, RC contain the addresses of the input registers. RT contains the address the output register

- format
- functionality

The opcode and mnemonic of an instruction are required to insert the instruction into the array in the InstructionContainer class. The mnemonic of an instruction is needed several times in the class file of the instruction and to name this file in the first place. It is also needed to include the correct file in **/instructions/Instruction_inc.h**. The format of an instruction is required to obtain the correct information in the configure stage of an instruction and to place the class file into the correct directory. To be able to write the execution part of an instruction, the functionality of the instruction is required.

4.3.2 Implementation Procedure

After inspection of the already implemented instructions it became clear that all instructions could be mapped onto a single template. This template is as follows:

```
01 #ifndef __[Mnemonic]__
02 #define __[Mnemonic]__
03
04 #include "spu_defs.h"
05 #include <string>
```

```
06 #include <assert.h>
07
08 #include "RegisterFile.hxx"
09 #include "Instruction.hxx"
10 #include "BitHandler.hxx"
11
12 // if necessary
13 #include "[ParentClass].hxx"
14
15 class [Mnemonic]:public [ParentClass]
16 {
17 public:
18   [Mnemonic](std::string name):[ParentClass](name)
19   {
20   }
21
22   virtual ~ [Mnemonic]()
23   {
24   }
25
26   void configure(spu_word_t iword, spu_word_t pc)
27   {
28     [ParentClass]::configure(iword, pc);
29     //read all operands into appropriate member of class Instruction; e.g.:
30     _rt_index = BitHandler::getBits(iword, 25, 31);
31   }
32
33   virtual void execute(RegisterFile * rf)
34   {
35     [ParentClass]::execute(rf);
36
37     //declare destination value variable if necessary; e.g.:
38     RegisterFile::spu_GPR_t rt_value;
39     //read operands from registers if necessary; e.g.
40     RegisterFile::spu_GPR_t ra_value = rf->readGPR(_ra_index);
41
42     //perform the functionality of the operation; e.g. A:
43     for (int i = 0; i < 4; i++) {
44       BitHandler::chEndianW(&(ra_value.words[i]));
45       BitHandler::chEndianW(&(rb_value.words[i]));
46       rt_value.words[i] = ra_value.words[i] + rb_value.words[i];
47       BitHandler::chEndianW(&(rt_value.words[i]));
48     }
49
50     //write result back into register if necessary; e.g.:
51     rf->writeGPR(_rt_index, rt_value);
52   }
53 };
54
55 #endif //__[Mnemonic]__
```

Here [Mnemonic] is the mnemonic that is used to identify the instruction and [ParentClass] is either one of the classes that are defined because some instructions need more functionality than is given by the general Instruction class. In this template lines 1, 2 and 55 are part of a C++ definition to make sure that no instruction is accidentally defined twice. Lines 3-14 include necessary files for correct inheritance and use of helper classes. Lines 12 and 13 are only required when the instruction that is described is a special instruction like a branch for example. Lines 15-53 contain the class itself with a constructor and destructor function at the beginning.
The "configure" method at line 26 configures the available variables to contain the correct register addresses or immediate values. In this method the indexes of the registers that used are determined from the instruction. Immediate values are determined and initialized here as well.
The "execute" method is where the actual action takes place. In this method, the actual value that is contained within a register is collected and, if necessary, its endian is changed. After that the operation an instruction is supposed to execute is executed, usually in a for-loop. Finally the result is given to the output register.
Because different instructions use different register configurations, the Register class contains references to each possibility. When, for example, an instruction operates on halfwords, the value in a register can be accessed as an array of eight halfwords. When an instruction operates on bytes, however, the value in a register can be accessed as an array of 16 bytes. All possible ways to access the values in a register are:

- Array of 2 doublewords of type 'double'.
- Array of 4 words of type 'float'.
- Array of 2 doublewords of type 'long'.
- Array of 4 words of type 'int'.
- Array of 8 halfwords of type 'short'.
- Array of 16 bytes of type 'char'.

These arrays are contained within a union of arrays and are accessed by defining an instance of the Registerclass and give it a value with a call to the register file (line 40). After that all arrays can be accessed as a public variable of the Register instance.
After writing an instruction class, it needs to be inserted into the simulator. To do this, only two lines have to be added to two different files:

- To **/instructions/Instructions.h** the following needs to be added:
  `_instrs[[opcode]] = new [Mnemonic]("[Mnemonic]");`
  The opcode is entered as a three digit hexadecimal number. To get the twelve bits that are required for this the opcode that is given with the instruction needs to be extended to the right with a single zero and to the left with as many zeros as is needed to obtain twelve bits.
- To **/instructions/Instructions_inc.h** the following needs to be added:
  `#include "[Mnemonic].hxx"`

4.3.3 Example

To illustrate the procedure for adding instructions to the SPU, this example is given. It contains the actual implementation of one of the instructions that is described in Section 4.4. The mnemonic of the instruction that was chosen for this example is **FSMH**, its full name is **Form Select Mask Halfwords**. The functionality of this instruction is to form a mask of halfwords by reading eight bits of register RA. The opcode of this instruction (see [2]) is **00110110101 (0x1B5)** and its format is **RR**. Now that all required information is present we create a file called **Fsmh.hxx** in directory **/instructions/RR** with the following contents:

```
#ifndef __Fsmh__
#define __Fsmh__

#include "spu_defs.h"
#include <string>
#include <assert.h>

#include "RegisterFile.hxx"
#include "Instruction.hxx"
#include "BitHandler.hxx"

class Fsmh:public Instruction
{
public:
  Fsmh(std::string name):Instruction(name)
  {
  }

  virtual ~ Fsmh()
  {
  }

  void configure(spu_word_t iword, spu_word_t pc)
  {
    Instruction::configure(iword, pc);
    _rt_index = BitHandler::getBits(iword, 25, 31);
    _ra_index = BitHandler::getBits(iword, 18, 24);
  }

  virtual void execute(RegisterFile * rf)
  {
    Instruction::execute(rf);

    RegisterFile::spu_GPR_t rt_value;
    RegisterFile::spu_GPR_t ra_value = rf->readGPR(_ra_index);

    BitHandler::chEndianW(&(ra_value.words[0]));

    for (int i = 0; i < 8; i++)
    {
```

```
      if (BitHandler::getBits(ra_value.words[0], 24 + i, 24 + i) == 0)
      rt_value.halfwords[i] = 0x0;
      else
        rt_value.halfwords[i] = 0xFFFF;
    }

    rf->writeGPR(_rt_index, rt_value);
  }
};

#endif //__Fsmh__
```

As is shown, all appearances of [Mnemonic] are replaced with "Fsmh" and, because this instruction is not a special instruction, all appearances of [ParentClass] are replaced with "Instruction". In the configure method, the indexes of the used registers (registers RA and RT) are determined by obtaining the correct bits of the instruction. The execute method of this class first defines two register values (ra_value and rt_value) and puts the value of register RA in the former one. Then the endian of the first word is changed because the bytes that determine the mask are the rightmost eight bits of the preferred slot of register RA. Next, all eight halfwords of register RT are given all zeros or all ones depending on the values of the respective bits of register RA. No endian change of the result is necessary because the results do not change when the endian is changed. Finally the result is written to the output register in the register file.
In order to make this class work within the simulator the following two lines must be added to **/instructions/Instructions.h** and **/instructions/Instructions_inc.h** respectively:
`_instrs[0x1B5] = new Fsmh("Fsmh");`
`#include "Fsmh.hxx"`
This makes sure that an instance of this instruction class is put into the array containing all instructions and that the corresponding file is included while compiling the simulator.

4.4 Implemented Instructions

As can be seen in Section 4.2.1, the instructions are ordered by the format they have. The directory structure of the source of the simulator therefore contains a folder for each of the formats with the exception of 'RR' and 'RI7' instructions which are both in the 'RR' directory due to their similarity. The instructions listed below are the instructions that were implemented by us and are sorted first by instruction format and then by mnemonic.

- RI8 instructions

  - [CFLTI] Convert Floating Point to Signed Integer
    Convert four single precision floating point numbers to rounded, or clipped, signed integer numbers.

  - [CFLTU] Convert Floating Point to Unsigned Integer
    Convert four single precision floating point numbers to rounded, or clipped, unsigned integer numbers.

- RI10 instructions

  - [CGTBI] Compare Greater Than Byte Immediate
    Compare the sixteen bytes of register RA with the immediate value and set per byte if they are greater than the immediate value.
  - [HGTI] Halt if Greater Than Immediate
    Use signed comparison between the immediate value and the preferred slot of Ra and halt execution if Ra is greater than the immediate value.
  - [HLGTI] Halt if Logically Greater Than Immediate
    Use unsigned comparison between the immediate value and the preferred slot of Ra and halt execution if Ra is greater than the immediate value.
  - [MPYI] Multiply Immediate
    Multiply the four words of register Ra with the sign extended immediate value.
  - [ORBI] OR Byte Immediate
    Bitwise 'OR' of the 16 bytes of register Ra with the immediate value.
  - [SFHI] Subtract From Halfword Immediate
    Subtract the eight halfwords from the sign extended immediate value.
  - [XORBI] XOR Byte Immediate
    Bitwise 'XOR' between the sixteen bytes and the immediate value.
  - [XORHI] XOR Halfword Immediate
    Bitwise 'XOR' between the eight halfwords and the sign extended immediate value.

- RI16 instructions

  - [BRA] Branch Absolute
    Branch to the address that is in the immediate value.
  - [BRASL] Branch Absolute and Set Link
    Branch to the address that is in the immediate value and leave a link to the current address in register RA.

- RR and RI7 instructions

  - [BG] Borrow Generate
    Compare the four words of register RA to register RB and set the result to '1' if the value of RB is the largest.
  - [BGX] Borrow Generate Extended
    If the least significant bit of register RT is '0' add '1' to register RB, then do the same as in "Borrow Generate".

- [BIHNZ] Branch Indirect if Halfword Non-Zero
  If the rightmost halfword of the preferred slot of register RT is non-zero, branch to the address in the preferred slot of register RA.
- [BIHZ] Branch Indirect if Halfword Zero
  If the rightmost halfword of the preferred slot of register RT is zero, branch to the address in the preferred slot of register RA.
- [CGX] Carry Generate Extended
  Add the four words of registers RA, RB and the least significant bit of register RT and place the carry in RT
- [DFNMA] Double Float Negative Multiply and Add
  Multiply the double precision floating point numbers in registers RA and RB and add the result to the floating point number of register RT. If the result is not "Not A Number", negate it.
- [DFNMS] Double Float Negative Multiply and Subtract
  Multiply the double precision floating point numbers in registers RA and RB and subtract the value in register RT. If the result is not "Not A Number", negate it.
- [DSYNC] Data Synchronize
  Synchronize dataflow to ensure consistency if observed from outside.
- [EQV] Equivalent
  Bitwise 'XNOR' with registers RA and RB.
- [FESD] Floating point Extend Single to Double
  Extend two of the single precision floating point numbers in register RA to double precision floating point numbers.
- [FRDS] Floating point Round Double to Single
  Round the two double precision floating point numbers in register RA to single precision floating point numbers
- [FRSQEST] Floating point Reciprocal Square Root Estimate
  Estimate the floating point reciprocal of the square root of register RA.
- [FSMB] Form Select Mask for Bytes
  Form a byte mask by reading 16 bits of register RA.
- [FSMH] Form Select Mask for Halfwords
  Form a halfword mask by reading eight bits of register RA.
- [GBB] Gather Bits from Bytes
  Count the number of bits in each byte in register RA.
- [GBH] Gather Bits from Halfwords
  Count the number of bits in each halfword of register RA.
- [HEQ] Halt if Equal
  If the values in registers RA and RB are equal, halt execution.
- [HGT] Halt if Greater Than
  If the signed value in register RA is larger than the signed value in register RB, halt execution.

- [HLGT] Halt if Logically Greater Than
  If the unsigned value in register RA is larger than the unsigned value in register RB, halt execution.
- [MFSPR] Move From Special Purpose Register
  Move the value in the Special Purpose Register to register RT.
- [MPY] Multiply
  Multiply the odd halfwords of register RA with the odd halfwords of register RB.
- [MPYHHA] Multiply High High and Add
  Multiply the even halfwords of register RA and register RB and add the value in the words of RT.
- [MPYHHAU] Multiply High High and Add Unsigned
  Unsigned multiplication of the even halfwords of register RA and register RB and add the value in the words of RT.
- [NAND] NAND
  Bitwise 'NAND' of registers RA and RB.
- [ORX] Or Across
  Bitwise 'OR' of the words of register RA with themselves.
- [ROT] Rotate
  Rotate the words of register RA by the amount given in the words of register RB.
- [ROTH] Rotate Halfword
  Rotate the halfwords of register RA by the amount given in the halfwords of register RB.
- [ROTHI] Rotate Halfword Immediate
  Rotate the halfwords of register RA by the amount given in the immediate field of the instruction.
- [ROTHM] Rotate Halfword and Mask
  Right shift the halfwords of register RA by the amount given in register RB. Zero fill at the left.
- [ROTMA] Rotate and Mask Algebraic Word
  Right shift the words of register RA by the amount given in register RB. Replicate the leftmost bit.
- [ROTQBI] Rotate Quadword by Bits
  Rotate the entire contents of register RA by the number of bits given in register RB.
- [ROTQBII] Rotate Quadword by Bits Immediate
  Rotate the entire contents of register RA by the number of bits given in the immediate field.
- [ROTQBYBI] Rotate Quadword by Byte from Bit Shift Count
  Rotate the entire contents of register RA by the number of bytes given by register RB.

- [SFX] Subtract from Extended
  If the least significant bit of register RT is '0', add '1' to the value in register RA. Then subtract from register RB.
- [SHLH] Shift Left Halfword
  Shift the halfwords of register RA to the left by the number of bits given in register RB. Zero fill at the right.
- [SHLHI] Shift Left Halfword Immediate
  Shift the halfwords of register RA to the left by the number of bits given in the immediate field. Zero fill at the right.
- [SHLQBI] Shift Left Quadword by Bits
  Shift the entire contents of register RA left by the number of bits given in register RB. Zero fill at the right.
- [SUMB] Sum Bytes
  Sum the bytes of every word of register RA and register RB.

To test the instructions that were implemented, the test program 'intrinsics' that is provided with the Cell SDK, was used. This test program compared the output of 100 executions of each instruction with random input values to a value that is computed natively. When all 100 executions compare equal to the reference value, the instruction is considered correct. After running the test program, nearly all instructions were considered correct. The only instructions that did not pass the test were FRDS (Floating Point Round Double to Single) and two instructions that were already implemented: Multiply and Add and Multiply and Subtract, both at the floating point part. This is probably because rounding is done differently in the test program because when printing the results of both the result of the instruction and the natively calculated value, no difference could be observed.
The last six instructions of the above listing are optional in the original design of the Cell BE processor and therefore not included in the test program. We decided not to test or include these instructions in the final simulator although they should work correctly.

4.5 Conclusion

In this chapter we have described the file organization and directory structure of that part of the simulator that is relevant to writing new instructions. Also described is the different instruction formats and the register configurations that are used in the Cell BE processor. Next a detailed description of the implementation procedure is given to describe the template that can be used for each instruction class. Following are the lines that need to be added to some files in order to have the instruction function properly in the simulator. Finally an example of one instruction is given to illustrate the procedure in more detail followed by a list of the instructions that were implemented to complete the SPU instruction set.

5 CellSim Profiler Tool

5.1 Introduction to the CellSim Profiler Tool

The Cell Broadband Engine Architecture (Cell BE) is a microprocessor that is jointly developed by Sony, Toshiba and IBM. Its main purpose for development was to bridge the gap between desktop processors and specialized high-performance processors. To achieve this, Cell combines a general purpose Power Architecture core with eight coprocessing elements interconnected through a specialized high-bandwidth data bus [7]. To simulate the Cell BE, IBM has implemented a simulator (Systemsim) with three precision levels and an incorporated analysis tool to get an impression of the performance of a program [6]. This simulator, however, along with its profiling abilities is not open source and can therefore not be modified or extended. This makes it impossible to use Systemsim in computer architectural research programs. To have a heterogeneous chip multiprocessor research infrastructure, the Barcelona Supercomputing Center is developing a Cell BE Simulator (CellSim). CellSim is being developed in the UNISIM environment to be able to provide a modular design and thereby the exchange of different modules easy. Although CellSim is operational, its usefulness is limited as no profiling functionality has been implemented. This chapter describes the CellSim Profiler Tool, a framework that enables the developer to log events of choice and check for bottlenecks in a design. The CellSim Profiler Tool is designed to give the developer the ability to insert profiling capabilities into the simulator at the source level and control these from within the simulator or from simulated software. In collaboration with the developers at the Barcelona Supercomputing Center the general structure of the tool was defined. The main goal of this work is implementation and optimization of the tool, including changes to the general structure if necessary.
This chapter first discusses some of the work that is related to CellSim and the profiler tool in Section 5.2. Next the proposal for the implementation of the profiler is discussed in Section 5.3 followed by a description of the implementation of the profiler in Section 5.4. Section 5.6 discusses some of the problems that were met during the course of this project as well as changes that were made in the original design.

5.2 Related Work

CellSim is developed using the UNISIM environment [13]. This environment was chosen because of the modular design of the Cell BE processor that maps easily to the modular design possibilities of the UNISIM environment. Other simulator environments that could have been used, but do not support the modular design as much as UNISIM, are SystemC [12] and the predecessor of UNISIM, MicroLib [8], two simulator environments that are based on the C++ programming language and provide libraries to facilitate the

implementation of architectural components.
Considering simulators, the number of possibilities is as large as the number of processors available today if not larger. One simulator that fits very well in the context of this thesis is the SimpleScalar simulator [14, 16, 10]. Along with the possibility to extend the existing processor, it also features four different modes of execution including a profiling implementation, the so called profile-sim version. Another simulator with profiling abilities is Systemsim, a Cell BE simulator that is provided with the Cell BE SDK [1]. This profiler enables the developer to view statistics and even detailed information up to the machine state of each processor at run-time. Although very detailed, Systemsim cannot be used for research in computer architectures because it is not open source en can therefor not be modified to fit the needs of a researcher.

5.3 Implementation Proposal

To build the framework that enables the developer to implement his or her own profiler into CellSim, a UML description along with an implementation proposal has been designed in collaboration with the Barcelona Supercomputing Center. This Section first discusses the UML description followed by the way the profiler tool can be inserted into CellSim. Then the communication between the profiler and simulated software is discussed and finally the usage of the profiler from simulated software is given.

5.3.1 UML Description

For this tool, a UML description was developed in collaboration with the Barcelona Supercomputing Center. This section describes the UML description as it is at the end of this writing. Some minor changes have been applied to the original description. The UML description is depicted in Figure 5.1.

An explanation of Figure 5.1 is given below.

Service is the overall class from which Logger and Directory inherit most of their methods. Methodss that are available from this class are: **clear** to clear the counters, **dump** to dump the current value and metric to the screen, and **trace** to enable or disable tracing of events.

Logger is the actual logging instance. In this instance a record is kept to count the number of events that have happened since the beginning or the last 'clear' command. A call to **event** will increment the counter when 'value' is not equal to 0. If tracing is enabled, a call to the processor's **trace_event** will follow. A call to **clear** will clear the current counting value, **dump** will print the current value and metric to the screen and **trace** enables or disables the tracing. Each logger has its own unique 'event_number' which is automatically generated and compatible with 'Paraver[1] event numbers'.

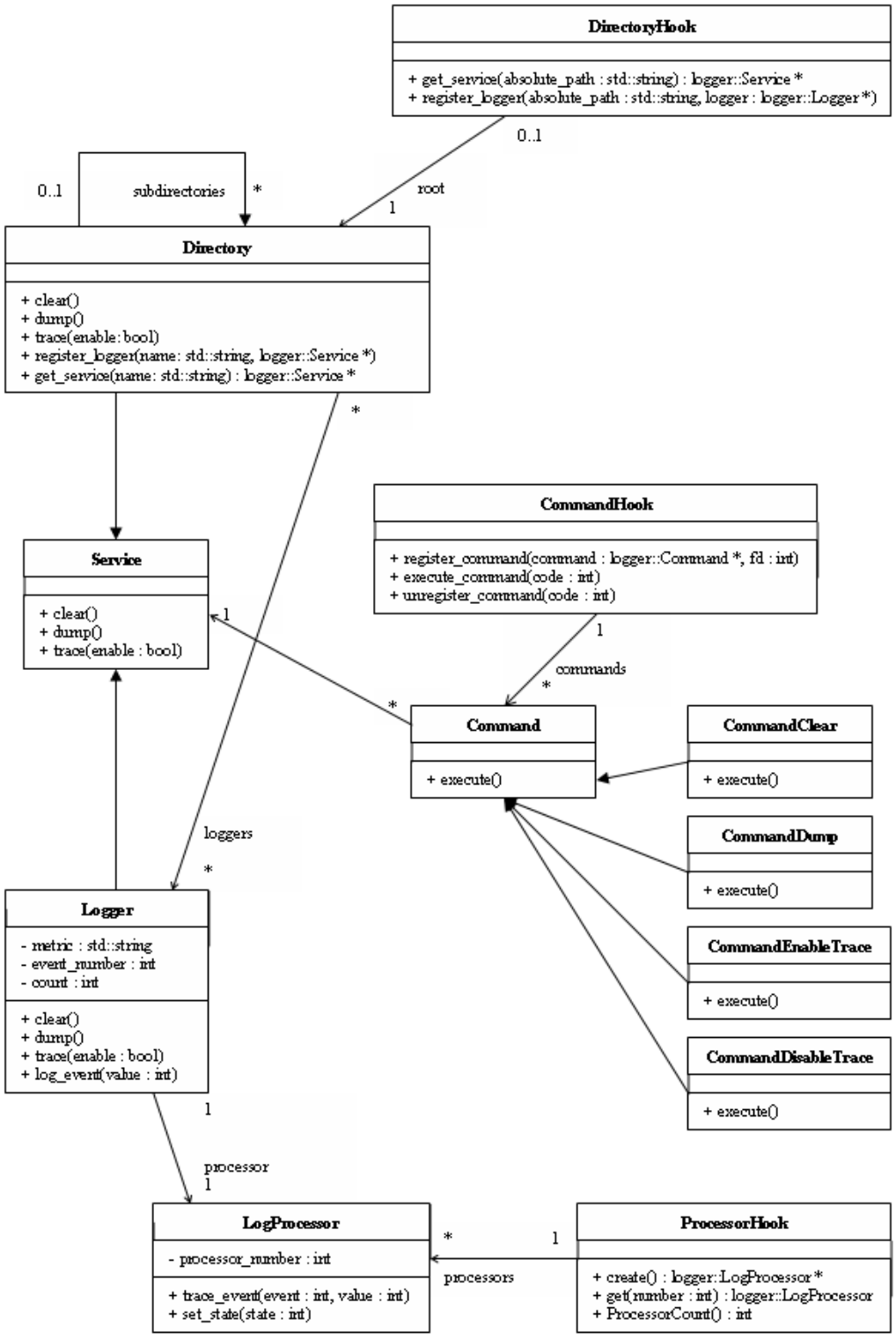

Figure 5.1: UML description of the CellSim Profiler Tool

Processor is the actual logging interface. A processor is accessed whenever a logging-event happens. Each processor instance has its own unique number starting with 1.**trace_event** retrieves the current cycle and uses it to generate a new 'Paraver[1]-event'. **set_state** sets the state of the processor for the 'Paraver[1]-trace'.

ProcessorHook is responsible for giving every processor a unique number, therefore making identification possible. It also collects all processors.

Directory is a means to keep track of all the loggers as well as to interact with them. It has two main components: a list of subdirectories and a list of loggers. **get_subdirectory** is responsible to get the next level subdirectory and create it if it does not exist. If any of the inherited methods of Service are called (**clear**, **dump** or **trace**) it is automatically applied to all subdirectories and loggers registered to this directory.

DirectoryHook is responsible for creating and maintaining a directory-like structure that is used for the interaction with, and maintenance of the loggers. It creates and contains the root directory.

Command is the responsible class for performing one operation on one selected service. Commands that are given from simulated running software relate to instances of the subclasses of command. Each subclass implements a different form of the **execute** method.

CommandHook is responsible for collecting all the registered commands and assign them unique numbers for identification purposes. Each command needs to be registered. Once registered, a command can be executed as many times as needed until it is unregistered.

5.3.2 Adding a Logger Instance to the Simulator

To be able to use the profiling tool, the simulator needs to be changed. Since a developer knows what needs to be profiled, he also knows where in the architecture the profiling needs to be done. In the area of the processor where the profiling needs to take place, a developer needs to declare the number of loggers he needs and register them to one or more directories. After that he can use each logger as many times as he needs. The following piece of code can be inserted at any part of the simulator:

```
// Include the correct header files
#include "Logger.hpp"
#include "DirectoryHook.hpp"

// Instantiate the loggers that are needed
logger::Logger myLogger;
```

[1]Paraver [9] is a tracing tool which is developed by the Barcelona Supercomputing Center and the Universitat Politècnica de Catalunya. It takes in raw trace data and can display it in several graphical or textual ways.

```
// Register the logger(s) to different directories and set its name
myLogger.set_metric("MyLogger");
logger::DirectoryHook::register_logger("/some/path/", &myLogger);

// Start normal execution code
..
myLogger.log_event(1);
..
myLogger.log_event(0);
..
```

In the above example, one logger is instantiated, named and registered to a directory. Although it is not shown in this small example, one logger can be registered to many directories and one directory can have many loggers registered to it. This is done because it enables the developer to perform one action, like a dump, to all loggers that are related to each other with only a single command. These loggers are assumed to be placed within the same directory. After the logger instance is registered it is available for use, this is shown in the last five lines. Here an event call is performed twice. The first call will produce an increase in the counter of the logger instance and, if tracing is enabled, an event call is added to the list of event calls. The second time, no increase in the logger counter is produced, however, an event call is added to the list of event calls.

5.3.3 Interaction With Simulated Software

In order to able to interact with the profiler from simulated running software one of the instructions of the CellSim processor is adjusted to function as an interface. The 'AND' instruction is chosen and changed in such a way that when executed with three times the same register for its arguments it executes the command that is indicated with the value in this register. The choice for the 'AND' instruction is because of the nature of its behaviour. When executed normally with three times the same register for its arguments, this register will afterward contain the exact same value as it did before the execution (the 'OR' instruction could have been chosen as well). This means that any program that has been altered to make profiling possible, doesn't need to be changed back and recompiled in order to be able to run it on another Cell simulator or actual Cell processor, although in order to run it efficiently it should. The use of an existing instruction instead of a specially implemented instruction is because the use of an existing instruction means that the compiler and assembler can be left unchanged.
To make it possible to register and unregister commands, the system calls 'open' and 'close' are modified. The system call 'open' already has the ability to handle filenames with "/processor" and "/accelerator" in a special way. Now "/logger" is added to these special cases. Whenever a file is opened using "/logger" as the root of the absolute path it will create a new command which is automatically registered. The command type must be indicated by the "flags" argument that is part of the system call, e.g. *open("/logger/some/path/service, CMD_CLEAR)* will create a new instance of the class CommandClear and register it to the logger or directory named 'service' inside the directory '/some/path'. Whenever this is done, the command identification number

(command id) is given instead of the file descriptor that is normally returned by 'open'. 'CMD_CLEAR' is defined in a header file that should be included in the code of the simulated software.
The system call 'close' is used to recognize the command ids and unregister the commands that were used throughout the program. It is not necessary to unregister commands before ending a simulated program, however due to the fact that only 256 commands can be registered it is wise to unregister a command when it is no longer needed. The files and parts of the code that is involved to realize the above are discussed in Section 5.4.1.

5.3.4 Usage From The Simulated Application

For the use in a simulated program, a header file is created with the following contents:

```
enum commands {CMD_CLEAR, CMD_DUMP, CMD_ENABLE_TRACE, CMD_DISABLE_TRACE};
#define logger_create_cmd(target, dir, command) (target = open(dir, command))
#define logger_execute(command) asm("and, %0, %0, %0"::"r" command)
#define logger_destroy_cmd(command) close(command)
```

First, all the possible commands are enumerated and thereby given a number. Second, a macro to create command identifiers is defined. It requires the target variable, a directory name and a command. The commands available are the ones that are enumerated in the first line. Third, for the execution of a command at the desired time is done by placing an 'AND' instruction with three the same registers in the assembly code of the program, note that 'command' is the number that is returned by 'logger_create_cmd'. Last, the possibility to destroy or unregister a command when it is no longer used. This is a one on one translation to the systemcall 'close'. 'command' is the number that is returned by 'logger_create_cmd'.

5.4 Implementation of the Profiler

In order to implement the profiler framework, some changes needed to be made to three of the original source files of CellSim. These changes are discussed first in this section, followed by the files that were added to CellSim in order to get the functionality that was intended.

5.4.1 Modified Files

To locate the files that were modified to obtain the desired functionality the relevant directory tree is shown below:

```
cell-sim-unstable/
|-library/
| |-accellerator/
| |-processor/
|   |-src/
|     |-sc/             Contains the system calls 'open' and 'close'
```

```
|
|-native/
| |...
|
|-modules/
  |-src/
    |-processor/
    |-accelerator/
      |-commands/
      |-instructions/
        |-RR/           Contains the 'AND' instruction
        |...
```

The files that were modified as well as the modifications that were made are listed below:

- */library/processor/is/sc/open.cpp*
  This file contains the implementation of the system call 'open'. It was changed to include the file **CommandHook.hpp**, **CommandClear.hpp**, **CommandDump.hpp**, **CommandEnableTrace.hpp** and **CommandDisableTrace.hpp** to be able to create and register new commands. The 'logger' option is added to the special cases that are checked. Last, in the listing of the special cases, the logger case is added. After entering the possibility, an output stream pointer with a number that is closest to zero is requested like in the other special cases. After that the correct command is chosen by means of the 'flags' argument of the system call and created. Last, the new command is registered to the directory that is indicated by the given string and the output stream pointer is returned to the calling program.

- */library/processor/is/sc/close.cpp*
  This is the file containing the system call 'close'. Because the index number for the commands is obtained the same way as when the case would have been *"/accelerator"* or *"/processor"*, the 'close' system call is modified only to unregister a command when the already existing code could not be executed.

- */modules/src/accellerator/instructions/RR/And.hxx*
  This file contains the implementation of the 'AND' instruction. The entire execute method is turned into an if-else-statement. Whenever the three registers of this instruction are the same, the value of these registers is taken and used as an index to get the proper command. This command is then executed after which the instruction is finished. When the three registers are different, the instruction is executed as if nothing special is going on.

5.4.2 Added Files

To implement the UML description that is given in Section 5.3.1, new classes and files needed to be created. The files that were created are listed in this section. Each file description includes a short description of the function of each method it contains. These

files are located in the directory: `/modules/src/logger`. This directory is added to CellSim.

Service.cpp/hpp Implements the parent class for the Logger class and the Directory class. The methods in this class are empty.

Directory.cpp/hpp Implements the Directory class. It inherits from the Service class. The following methods are part of it:

- **Directory()** Constructor to instantiate the root of all directories. Its name is set to "/logger".
- **Directory(string)** Constructor to instantiate any subdirectory. The name given to the constructor is given to the directory.
- **clear()** Method to clear the counters of all loggers registered to this directory and all subdirectories.
- **dump()** Method to dump the metric and value of the counters of all loggers registered to this directory and all subdirectories.
- **trace(boolean)** Method to turn on or off tracing for the loggers registered to this directory and all subdirectories.
- **register_logger(string, Logger*)** Register a logger to this directory or one of its subdirectories. The string is broken down into tokens by the Tokenizer class. When a directory with the name of the token exists, registration continues in this directory. When a directory does not exist, the directory is created and then registration continues in the new directory. When the initial string is completely broken down, the Logger is registered to the directory that is the current directory.
- **get_service(string)** To get a subdirectory or logger instance. The string is broken down into tokens. When the string is not completely broken down, search only takes place in the subdirectories. At the end of the string, subdirectories and logger instances are searched. Whenever a subdirectory or logger can not be found, 'NULL' is returned.

DirectoryHook.cpp/hpp Implements the DirectoryHook class. This class is used to create and maintain the directory structure that is used to identify and communicate with loggers and directories. The methods in this class are static and implement the following:

- **get_service(string)** Method to get a directory or logger from the directory structure. The string contains the full path of the service that is required. If "/logger" is at the base of this string, it is removed. After that the get_service method of the root directory is called with the remaining string.
- **register_logger(string, Logger*)** To register a logger to the directory structure a string with the full pathname must be provided along with the logger. This method creates the "/logger" directory in the home directory of the user on the hard drive. After that the register_service method of the root directory is called.

Logger.cpp/hpp These files implement the Logger class. Loggers are used to identify events and determine whether or not a counter must be increased or trace elements must be created. Each logger is provided with an instance of the LogProcessor class. The methods are as follows:

- **Logger()** Constructor to initialize all variables.
- **clear()** Clear the counter.
- **dump()** Print the name of the logger and the value of the counter to the screen.
- **trace(boolean)** Enable or disable tracing.
- **log_event(int)** If the provided integer is not equal to zero, increment the counter. If the provided integer is not equal to zero and tracing is enabled, call the processors method trace_event.
- **set_metric(string)** Set the name of this logger.
- **get_metric(string)** Get the name of this logger.
- **set_event_number(int)** Change the event number or identification number of this logger.

Processor.cpp/hpp Implement the LogProcessor class. Its main purpose is to collect trace data. Due to the possibility that a different user wants different data, this class is subject to changes. Its functions are the following:

- **LogProcessor(int)** Constructor that initializes the trace vectors, identification number and state trace elements that record the state of the processor.
- **trace_event(int, int)** Method to record events. The arguments contain an event number and a value.
- **set_state(int)** Set the state of the LogProcessor instance.
- **print_trace()** Print the trace data to file. The implementation of this method varies for every trace file configuration that is used, currently it has none.

ProcessorHook.cpp/hpp The ProcessorHook class is used to assign each processor a unique identification number. It is used by the Logger class to get the LogProcessor instance it requires and it holds a vector containing all the LogProcessor instances that are instantiated. Its methods are the following:

- **create()** Method to create a new LogProcessor instance and return it to the caller.
- **get(int)** Get the LogProcessor instance that is indicated by the argument, if it is not found, return 'NULL'.
- **ProcessorCount()** Return the number of LogProcessors currently available.
- **PrintAll()** Print all log data to file. Currently there is no working implementation available.

Command.cpp/hpp Class from which every Command class inherits. It only contains the method 'execute' which is empty for this parent class. Commands can be accessed from simulated software to communicate with the profiler tool. All children only have a constructor that gets the Service instance (Directory or Logger) they must apply to and an execute method to perform their specific task on the Service they contain.

CommandHook.cpp/hpp The CommandHook class maintains a list of all Commands that are created and is called to (un)register, execute or get Commands that are created. This class is used by simulated software to influence the profiler. The methods of this class are:

- **register_command(Command*, int)** Inserts a command into the array of commands contained within this class. The integer argument is the index at which the command pointer is supposed to be inserted. This index is obtained with the system call 'open'.
- **execute_command(int)** The argument of this method is an index number. The command is executed when it is found in the array. When it is not found, an error message is given.
- **unregister_command(int)** Deletes a command from the array of commands.
- **get_command(int)** Get the command that is at the provided index in the array.

CommandClear.cpp/hpp CommandClear inherits from Command. It executes the 'clear' method of its Service.

CommandDump.cpp/hpp CommandDump inherits from Command. It executes the 'dump' method of its Service.

CommandEnableTrace.cpp/hpp CommandEnableTrace inherits from Command. It enables tracing for its Service. If the Service is a directory this means that all subdirectories and underlying Loggers are affected.

CommandDisableTrace.cpp/hpp CommandDisableTrace inherits from Command. It disables tracing for its Service. If the Service is a directory this means that all subdirectories and underlying Loggers are affected.

Tokenizer.cpp/hpp Class that can split up a full directory path into separate directories. The tokens that are returned are the directory names including the '/' at the beginning. If called with an empty string, the string argument is assumed to be filled with a string and the next '/' character index is returned. If called with a non-empty string, the string argument is initialized.

Tracer.cpp/hpp These files are meant to contain methods to print a complete trace file containing the information that is required to perform analysis of a programs running. This is, however, not completed into a fully working system and still in development.

profiler.h Header file containing the proposals mentioned before to include in a program that is intended to make use of the profiler.

5.5 Example of the Profiler use in Simulated Software

To illustrate the use of the profiler tool, a small example is given in this section. This example only concerns the use in a simulated program and therefore assumes that the changes described in Section 5.3.2 are applied to some part of CellSim. For this example it is not important where these changes have been inserted, because it does not matter to simulated software what changes have been made to the simulator. The following C code is a program that initializes two commands, clears the counter of the logger, then only performs a simple addition and afterward dumps the counter value.

```
int main(unsigned long long id, unsigned long long argp)
{
  int a = 6, b = 4;
  int logger_clear, logger_dump;

  logger_create_cmd(logger_clear, "/some/path/MyLogger", CMD_CLEAR);
  logger_create_cmd(logger_dump, "/some/path/MyLogger", CMD_DUMP);
  logger_execute(logger_clear);

  a = a + b;

  logger_execute(logger_dump);
  return 0;
}
```

This code can be compiled to be run on a SPU and run as such.

5.6 Difficulties During the Development of the CellSim Profiler Tool

During the development of the CellSim Profiler Tool and the implementation of the missing instructions for the SPU for the simulator, several problems needed to be solved. This section first lists these problems and gives a description of their solution. After that, some of the changes in the original design that were made during the implementation of the profiler tool are discussed.

5.6.1 Problems That Needed to be Overcome

During the development of almost anything, problems that are caused by misunderstandings or gaps in the knowledge of a developer are the cause of delays in the development as well as the cause for a developer to increase his knowledge of the subject at hand. During this project, the writer came across several problems that were (partially) solved to overcome them.

- C++ Programming Language
  To program the instructions that were needed for the SPU part of the simulator, pieces of C++ code needed to be read and written. Although writing the instructions became filling in a template after a while, programming the profiler tool was done from scratch. Therefore we needed to learn the C++ programming language by means of an instruction book and examples from the code that was at hand.

- CellSim
  The Cell BE Processor is a very complex piece of hardware. Therefore, CellSim is a complex piece of software. Understanding CellSim to a level at which it was possible to extend the existing simulator to make the necessary interaction between profiler and simulated software possible took quite some time. Next to that, the organization of the source files and the way they are compiled into a fully working set of binary files and libraries combined with the lack of experience of the writer in the use of so called 'Makefiles' caused a whole lot of trouble in developing the profiler tool. Eventually, we gained a moderate understanding of the 'Makefiles' and a solution that worked was found.

- Translation from idea to implementation
  At the beginning of the project, a UML description and a description of the implementation were established in collaboration with the Barcelona Supercomputing Center. However, communications about both these items were difficult and it took a lot of time before mutual understanding was obtained.

- Communication
  Communication between ourselves and the designers of CellSim who are located in Barcelona, was a nuisance in itself. Finding the right person for a problem at hand appeared more difficult than originally thought and although the support came with good information, communications in general were not fully satisfactory.

5.6.2 Changes From the Original Idea and Suggestions

As is custom with any project, during the process of development, some aspects of the original idea are changed to make the final implementation better or easier for the developer. Below, changes that were made to the original idea are discussed with the reason for which they were changed.

- Communication Between Simulated Software and Profiler
  To make communication between a simulated application and the profiler possible, the implementation of the system call 'open' was changed to act differently whenever a special pattern was entered for the pathname of the file that was to be opened. As is explained in Chapter 5.3.3, the final implementation uses the special pattern to distinguish the need for a new 'Command' and the rest of the pathname to identify the 'Logger' or 'Directory' that should be associated with the 'Command'. The type of command however is given by the second variable of the 'open' systemcall, where it originally was intended to be part of the pathname that is supplied with the systemcall. Hence instead of

the actual implementation which requires *open("/logger/some/path/loggername", CMD_CLEAR)*, to initialize a 'CommandClear', the original idea required *open("/logger/some/path/loggername/CommandClear")* to achieve the same result.
The reason for this change in implementation is the fact that the user of the profiler is already protected from making mistakes by the use of a macro for this implementation. Added to that, this implementation probably consumes less time to execute. The macro that is used to initialize a 'Command' consists of the string of characters containing the name of the 'Logger' instance and a variable containing the identifying value for the type of 'Command' that must be generated. The original idea would require a translation from 'identifying value' to 'string' and a concatenation of two strings in simulated software followed by a separation of these concatenated strings during the decoding in the implementation of the system call. The actual implementation now consists of a simple replacement of a macro by the system call followed by a 'switch statement' in the implementation of the system call, something that is much faster than the original idea would have been.

- 'register_logger' Method
  The 'register_logger' function in the 'Directory' and 'DirectoryHook' classes which is used to register a 'Logger' instance to a certain or several 'Directories' was originally intended to be a 'register_service' function with which it would be possible to register certain 'Directory' instances to another 'Directory'. This was changed because we decided that it would be easier to just register a 'Logger' instance and create the 'Directory' structure whenever it was needed automatically instead of by hand. This was done for the convenience of the developer and not for speed since both implementations are estimated to be equal in the use of processor time.

- Use of the 'LogProcessor' Class
  In the original idea as well as in the actual implementation, the classes 'LogProcessor' and 'ProcessorHook' exist to log events and provide unique processor identification numbers respectively. In actual use however each 'Logger' instance contains exactly one 'LogProcessor' instance which should both have the same identification number. Further, the 'LogProcessor' class is only used to store log data and ultimately should provide a file producing method or class with the appropriate information. All this information however could also be stored within the 'Logger' class itself, making both 'LogProcessor' and 'ProcessorHook' superfluous. They are both implemented for their possibilities to make the implementation of different tracefile generators easier as well as for the clarity they provide.

The above changes of the original plan are just the most important ones. The implementation of several helper functions and use of variables that are not in the UML description is not discussed here.

5.7 Concluding Remarks

At the Barcelona Supercomputing Center, a Cell Broadband Engine Simulator (CellSim) is being developed. CellSim is designed to cycle-accurate simulate the Cell BE processor, a processor with one core processor which is augmented with and controls eight synergistic processing units that are optimized for processing large amounts of data in the form of vectors. To be able to make detailed profiles of the execution of the simulator, a profiler is being developed and described in this chapter. This profiler lets a developer introduce profiling capabilities to parts of choice into the simulator and make detailed profiles of the use of different parts of the processor. Contrary to profilers that make a profile of an entire processor, this profiler needs to be inserted in the elements a developer wants to profile and then only profiles these elements. This saves a lot of time when executing the simulator, because no unnecessary parts of the simulator and profiler are executed. The downside however, is that a developer needs to insert the profiler himself.
For this project, a general idea and UML description were established in collaboration with the Barcelona Supercomputing Center and the main goal was to implement this idea. During implementation however, some changes were made to the original idea of implementation to speed up the execution of the profiler or to make its use easier for the developer that has to use the profiler elements. Although the profiler itself is fully operational, including an interface between simulated software and profiler, there still is no output file which represents the profiled data. This process was found to be much more problematic than anticipated.
Several problems during the development of the profiler significantly slowed down the process. The main problems that were met were communications with the Barcelona Supercomputing Center about the profiler itself, and errors during compilation of the simulator and our inexperience in using "make" and "automake". Nonetheless a working framework has been set up that can be easily implemented and extended to produce proper output.

6 Conclusion and Future work

This chapter first gives a brief summary of this thesis followed by a reflection on the work and decisions that were taken during the project and some recommendations for future work.

6.1 Summary

This thesis first describes the SimpleScalar Macro Tool, a tool that is designed to work within the toolchain that has been developed around the SimpleScalar processor simulator. SimpleScalar is a processor simulator that provides a developer with the possibility to extend the ISA of the processor with instructions that need evaluation or verification, functional units and registers. Along with the simulator, a version of the GNU C Compiler is provided to ensure easy development. Around SimpleScalar, two other tools have already been developed by the Computer Engineering Laboratory of Delft University of Technology. These two tools, the SimpleScalar Instruction Tool and the SimpleScalar Architecture Tool, ensure easy implementation of ISA extensions which is quite error prone, even for experts at SimpleScalar. All these tools ease the implementation of ISA extensions however, they still require a developer to program these extensions in assembly programming language, leaving the developer with all problems that come with programming at this low programming level such as register allocation and instruction ordering even when the compiler can do this at least for the native ISA. The new tool, SimpleScalar Macro Tool, provides the developer with a macro programming interface giving him the opportunity to program ISA extensions in a high level programming language and relieve him from many of the worries of assembly programming.
Evaluation of the SimpleScalar Macro Tool concluded that the use of the macros, when properly compiled, resulted in virtually no performance loss compared to hand written assembly code, something that is generally not associated with the use of a macro programming interface. In some cases, the macro coded C program even performed up to 6% better than hand written assembly code mainly due to better register allocation.
Also described in this thesis is a simulator for the Cell Broadband Engine, a processor consisting of one processor core controlling eight vector optimized co processors resulting in an incredible increase in computational power compared to conventional processors. This simulator, the Cell Simulator or CellSim, is developed by the Barcelona Supercomputer Center in Barcelona and is designed to perform cycle accurate simulations of the Cell BE processor. It is designed using the UNISIM simulator environment. At the beginning of the project the simulators co processor units ISA was not entirely implemented. I implemented and tested the missing instructions and thereby completed this part of the simulator.
For CellSim as well, I implemented a profiler tool that enables designers to make de-

tailed profiles of only those parts of the processor that need profiling. Unfortunately, at the time of writing this tool is not entirely completed and therefore lacks the ability to present the collected data in the desired file format. The profiler can, however, be inserted in desired elements of the simulator and interact with simulated software that is compiled for this simulator. Information about the performance impact is very limited because this depends on the number of elements that need profiling.

6.2 Conclusion

This section discusses and reflects on some of the actions that were taken during the two projects that were done for this thesis project.
The SimpleScalar Macro Tool took quite some time to test and evaluate. This was mainly due to the fact that in order to make the macros that are created with SSMT operate at the performance level that they are operating on now, all the kernel files that were used needed to be compared by hand to their hand written counterparts to find out the cause of performance decreases and then find a solution that effectively counteracted this decrease. This caused me to optimize the kernel files with GCC and eventually create the special duo of macros for the memory instructions. Something that also took a lot of time was the translation from hand written assembly code back to correct C-code because even though the original C-files were provided, some of the applied optimizations in the hand written assembly were done in the calculations of memory addresses and therefore difficult to spot and rewrite as valid C-code. However in the end we believe that we have delivered a tool that is worth using and will prove to be a help in the development of new software and the use of ISA extensions in SimpleScalar.
In retrospect, some decisions that were made could have been made differently to speed up the process of development like the decision to start from scratch with the writing of code for SSMT. It would have been better to look into the source code of SSIT and reuse the part that takes in the configuration file and breaks it down to pieces of useful information, or at least take this source as a basis for ideas for implementation. SSMIT does reuse the functions of SSIT that translate the configuration file into data with a usable format.
Most of the problems during the development of the CellSim Profiler Tool were caused by our inexperience in the use of C++ and Makefiles and difficulties in the communication with the Barcelona Supercomputing Center. We also insisted on finding out the use of Makefiles for ourself as a part of the learning curve where it would have been much faster when we would just have asked an experienced user what needed to be done. Although when we finally did ask questions, the answers were only partially relevant due to the communications problem consisting of bad english and the inability to put the problem into words.
The few decisions that were taken during the development of the CellSim were, in my opinion, for the better of the tool itself to speed up the execution or to ease the use of the profiler.

6.3 Future Work

The toolset around SimpleScalar containing SSIT, SSAT, SSIAT, SSMT and SSMIT has, in our opinion, reached the limit on operations that can be performed. Since all these tools were designed to keep the compiler intact, all actions taken must take place before or after compile time which gives each of the programs a disadvantage that cannot be overcome. Because of this disadvantage the number of things a developer needs to take into account, even with all the available tools, is still considerable with register allocation for the architectural extensions as just an example. These problems, however, can be addressed at compile time because that is the time when all the information is present and decisions are made. To conquer them would mean that the compiler must be changed in order to be able to address all the problems at hand. Future work would at least address the problem of register allocation, something a human is just no good at. The compiler could be changed to be able to do automatic register allocation for the extensions that were made to the SimpleScalar processor. The problem of instruction ordering is on a whole different level and would mean very significant changes to increase the compilers intelligence and knowledge of the instructions that can be used. Since the SimpleScalar processor can be changed in an infinite number of ways, finding a recipe to make this an automated process could prove to be impossible.
Slightly off the subject of SSMT, SSIT and SSAT could be augmented with an option to repair SimpleScalar to its original state or to replace the changes that were made by an earlier design. Especially in the beginning of the implementation stage, this could speed up the development process whenever the functional implementation of a new instruction is not entirely correct yet.
The CellSim Profiler Tool is just in its beginning stages of development and can be significantly extended to make several things possible. For starters, there is no means of presenting the gathered data in any file whatsoever. It is however, possible to change this to any output format that is required fairly easily. The number of commands that can be executed from simulated software is now limited to four (clear, dump, enable trace and disable trace). Commands to set flags in a tracefile for example could prove to be very useful to indicate a change in certain variables or the execution of certain functions.

Bibliography

[1] *Cell SDK*, http://www.alphaworks.ibm.com/tech/cellsw.

[2] *Cell Synergistic Processor Unit, Instruction Set Architecture*, http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44.

[3] *CellSim-Modular Simulator for Heterogeneous Multiprocessor Architectures*, http://pcsostres.ac.upc.edu/cellsim/doku.php.

[4] *GCC Extended In-line Assembly Documentation*, http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm.

[5] *GCC Internal Documentation*, http://gcc.gnu.org/onlinedocs/gccint.

[6] *IBM Full-System Simulator for the Cell Broadband Engine Processor*, http://www.alphaworks.ibm.com/tech/cellsystemsim.

[7] *Introduction to the Cell multiprocessor*, http://www.research.ibm.com/journal/rd/494/kahle.html.

[8] *MicroLib Homepage*, http://microlib.org.

[9] *Paraver Tracing Tool*, http://www.cepba.upc.edu/paraver/.

[10] *SimpleScalar LLC webpage*, http://www.simplescalar.com/.

[11] *SSIT, SSAT and SSIAT webpage*, http://ce.et.tudelft.nl/~demid/SSIAT/.

[12] *SystemC Homepage*, http://www.systemc.org.

[13] *UNISIM Homepage*, http://unisim.org.

[14] Todd Austin, Eric Larson, and Dan Ernst, *SimpleScalar: An Infrastructure for Computer System Modeling*, Computer **35** (2002), no. 2, 59–67.

[15] D. Borodin, B.H.H. Juurlink, and S. Vassiliadis, *Instruction-Level Fault Tolerance Configurability*, IC-SAMOS VII: Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation, July 2007, pp. 110–117.

[16] Doug Burger and Todd M. Austin, *The SimpleScalar Tool Set, Version 2.0*, SIGARCH Comput. Archit. News **25** (1997), no. 3, 13–25.

[17] D. Cheresiz, B.H.H. Juurlink, S. Vassiliadis, and H.A.G. Wijshoff, *The CSI Multimedia Architecture*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2005), 1–13.

[18] Daniel A. Jimenez, *Piecewise Linear Branch Prediction*, Proc. 32nd Annual Int. Symp. on Computer Architecture (ISCA-05) (Washington, DC, USA), IEEE Computer Society, 2005, pp. 382–393.

[19] B.H.H. (Ben) Juurlink, Demid Borodin, Roel J. Meeuws, Gerard Th. Aalbers, and Hugo Leisink, *The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT)*, unpublished manuscript, available at http://ce.et.tudelft.nl/ demid/SSIAT/.

[20] R. Hochsprung H. Scales K. Diefendorff, P.K. Dubey, *Altivec extensions to powerpc accelerates media processing*, IEEE Micro (2000).

[21] Dusty Lefevre, *Performance Analysis of Extended Subwords and the Matrix Register File*, Master's thesis, Universiteit Antwerpen, 2006.

[22] Serkan Ozdemir, Debjit Sinha, Gokhan Memik, Jonathan Adams, and Hai Zhou, *Yield-aware cache architectures*, Proc. 39th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-39) (Washington, DC, USA), IEEE Computer Society, 2006, pp. 15–25.

[23] Evert-Jan D. Pol, Bas Aarts, Jos T. J. van Eijndhoven, P. Struik, Pieter van der Wolf, Frans Sijstermans, M. J. A. Tromp, and Jan-Willem van de Waerdt, *Trimedia cpu64 application development environment*, ICCD, 1999, pp. 593–598.

[24] Eric Rotenberg, *AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors.*, FTCS-29 (Madison, Wisconsin, USA), IEEE Computer Society Press, Jun 1999, pp. 84–91.

[25] Asadollah Shahbahrami, Ben Juurlink, Demid Borodin, and Stamatis Vassiliadis, *Avoiding Conversion and Rearrangement Overhead in SIMD Architectures*, International Journal of Parallel Programming (2006), 237–260.

[26] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark, *Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches From a Large Global History*, Proc. 30th Annual Int. Symp. on Computer Architecture (ISCA-05) (New York, NY, USA), ACM Press, 2003, pp. 314–323.

[27] Jos T. J. van Eijndhoven, Kees A. Vissers, Evert-Jan D. Pol, P. Struik, R. H. J. Bloks, Pieter van der Wolf, Harald P. E. Vranken, Frans Sijstermans, M. J. A. Tromp, and Andy D. Pimentel, *Trimedia cpu64 architecture*, ICCD, 1999, pp. 586–592.

[28] Chuanjun Zhang, *Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches*, Proc. 33rd Annual Int. Symp. on Computer Architecture (ISCA-06) (Washington, DC, USA), IEEE Computer Society, 2006, pp. 155–166.

Curriculum Vitae

Born in Delft on the 12th of December in 1982. Attended primary school until the age of 12 in the year 1995.
Attended secondary school/high school at one of the highest possible levels and graduated in 2002. Continued at the University of Technology in Delft at the faculty of Electrical Engineering Mathematics and Computer Science to obtain a bachelors degree in Electrical Engineering.
After receiving his bachelor degree in 2005, he went for a Master degree in Computer Engineering landing him by Ben Juurlink for his thesis. Carsten is a quiet person who can listen to and understand problems quite well. He has a general interest in pretty much anything technical with a preference for electronics and control systems. He enjoys working out, running, good tv shows and designing and creating electrical circuits as well as (interactive/embedded) software. During his time at the Univesity of Technology in Delft, Carsten has helped his fellow students by working as a student assistant during practical exercises of the faculties for Electrical Engineering, Aerospace Engineering and Industrial Design. **C.M. van der Hoeven**