Generating Efficient Simulators from a Specification Language

1997-01-29

Fredrik Larsson

Computing Science Department
Uppsala University
Box 311, S-751 05 Uppsala, Sweden

This work has been carried out at
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden

and has been sponsored by
Ericsson Utvecklings AB
Box 1505, S-125 25 Älvsjö, Sweden

Abstract

A simulator is a powerful tool for hardware as well as software development. However, implementing an efficient simulator by hand is a very labour intensive and error-prone task. This paper describes a tool for automatic generation of efficient instruction set architecture (ISA) simulators. A specification file describing the ISA is used as input to the tool. Besides a simulator, the tool also generates an assembler and a disassembler for the architecture. We present a method where statistics is used to identify frequently used instructions. Special versions of these instructions are then created by the tool in order to speed up the simulator. With this technique we have generated a SPARC V8 simulator which is more efficient than our hand-coded and hand-optimized one.

Keywords: Instruction Set Simulator, Interpreter, Specification Language, Instruction Set Architecture, SPARC, Automatic Code Generation.
5 The Intermediate Format

5.1 Introduction ................................................. 20
5.2 Requirements .................................................. 20
5.3 Packing Parameters .......................................... 21
5.4 Storing the Intermediate Format .......................... 23
5.5 Optimizing Using Statistics ............................... 24
  5.5.1 Motivation .................................................. 24
  5.5.2 Specialization ............................................. 25
  5.5.3 Generalization ........................................... 27
  5.5.4 The Iteration Process .................................. 28

6 Generated Parts .................................................. 30

6.1 Introduction ................................................. 30
6.2 Main Include File ........................................... 30
6.3 The Decoder .................................................. 33
6.4 The Service Routines ....................................... 34
6.5 The Disassembler ............................................ 36
6.6 The Statistics Converter ................................. 36
6.7 The Assembler and Output Functions .................. 37

7 Performance .................................................... 38

8 Future Work ..................................................... 40

9 Related Work .................................................... 41

10 Conclusion ..................................................... 42

A An Example of a Specification - SPARC ................... 43

B An Example of a Decoder ...................................... 46

C An Example of Generated Service Routines ............... 48
1 Introduction

In this section we give a brief introduction to the field of simulators, why they are used and how they can be created. The purpose of this work will also be discussed.

1.1 Background

Ever since the early days of computer history, simulation techniques have been an important research area. When new computer systems are to be built it is essential that the behavior and correctness of such systems can be tested and verified in an efficient way. With a simulator it is, for example, possible for a hardware manufacturer to test new ideas and solutions for a component as well as measuring its performance even though it has not yet been built. Using simulators for hardware verification can thus reduce both the development cost and time significantly.

Besides hardware verification, simulators can be used as a tool in software development. A simulator can act as an ordinary profiler (counting instructions and locating commonly used routines etc.) but it has also the ability to collect information on a much lower level. For example, the cache performance of a program can be analyzed giving detailed information about which instructions cause cache misses. Pipeline throughput and functional unit usage can be examined. Also, if an entire operating system is run on top of a simulator the behavior and performance of the whole system can be analyzed. All this is important when optimizing for a specific target machine.

When debugging a non-deterministic system which depends very much on timing issues such as an operating system kernel or a server program (lots of timer events) a simulator can be a very helpful tool since if it is deterministic (a requirement) it can always reproduce the same state every time the system is run. In this way it is possible to isolate infrequent scheduling dependent errors.

Again, if a simulator exists for a new architecture which has not yet been built it is still possible to write software for it. Development of hardware and software can be done in parallel or in any order. The simulator g88 [1], for example, was used to debug a UNIX-kernel before hardware was available.

A third and perhaps most common reason for developing a simulator is that it makes it possible for one machine to run applications from another environment. A common word for this kind of simulator is emulator. The PCx Software Emulator [2] for the Amiga Computer which emulates a Pentium Pro is an example.

1.2 Levels of Abstraction

A simulator can simulate a processor at different levels of abstraction, from the analog transistor level to the instruction set architecture level as seen by an assembly-language programmer.
A list of different levels which could be identified is presented here [6]:

- **Instruction Set Architecture Level.** The level of the *instruction set architecture* (ISA) is the highest level of abstraction where only the result of each instruction is seen and not the mechanism behind it (an assembly language programmer’s view). For example no pipeline or functional units are modeled.

- **Organizational Level.** At the *organizational level* pipelines, instruction fetch and issue, caches, the MMU, the behavior of the functional units, etc. are simulated. This gives an almost clock-cycle true simulator in which the timing of the instructions and potential resource conflicts are considered.

- **Register Transfer Level.** The *register transfer level* describes the internal operation of the functional units, how storage elements, buses, and control signals are configured (bit level).

- **Logical Level.** The *logical level* simulates the logical equations that implements a given data-path.

The lower the simulation level is the more information need to be be processed which leads to longer simulation times. Of course different parts of the simulator can use different levels of abstraction. The user can concentrate herself on implementing the parts she wishes to examine on a lower level and thus reducing simulation time for other less interesting parts. It is even possible to switch level during simulation for example if more accurate information is needed from certain parts of a program while other parts are not so interesting besides keeping a correct abstract state. Such a technique has been used in SimOS [3] and is discussed in more details in [4].

**1.3 The Aim of This Thesis**

Implementing a simulator by hand could be a very labour-intensive and error-prone task. It could take several months to complete a simulator which means that a lot of work will be focused on making the simulator work instead of using it as a tool for making design decisions. It might also be difficult to verify the correctness of the simulator when it is ready for use.

It would be useful to have a tool which makes it possible to generate a simulator from some sort of specification (a meta-tool). Such tools already exists today but most of them are focused on abstraction levels below the ISA-level (see section 1.2) and are thus more suitable for hardware verification than for making instruction set simulators. An example of a *Hardware Definition Language* (HDL) is the VHDL [5]. For the ISA-level there does exists a tool [6, 7] but its solution for generating simulators – writing the specification of an ISA in an functional like language and then executing that specification when simulating – is not efficient enough for all needs.
Thus, the aim of this thesis was to construct a tool that could from a ISA-specification generate a simulator that should be as fast as or preferably faster than a hand-coded and hand-optimized one. Of course such a simulator will be considerably more efficient than simulators on lower abstraction levels.

1.4 Benefits of a Simulator Generation Tool

As indicated in the previous section there are several benefits of using a simulator generation tool (SGT). The user will make less errors if all information of an ISA is gathered into one (preferably compact) specification file instead of being scattered around over several source files. Changing or modifying instruction sets for a simulator is much easier done by means of a tool than by hand. The user can concentrate herself on central parts of the system, e.g. how the semantics of an instruction should be implemented in order to be efficiently executed. She can forget about routine work such as writing code for decoding instructions etc. which will be generated by the tool.

In our approach we use execution statistics of instructions to optimize the simulator. Such a task would be nearly impossible to do by hand or at least extremely laborious, especially if several simulators are to be optimized for different program types.

A SGT also has the ability of generating other utility tools such as assemblers, disassemblers and test programs for validating the correctness of the simulator which could be very helpful.

1.5 Organization of This Thesis

Section 2 describes some important simulation techniques used in the generated simulator. In section 3 we discuss our approach to a SGT. Section 4 describes our specification language. We use an intermediate format for more efficient interpretation and this is covered in section 5. The generated parts of the simulator is described in section 6. We present the performance of a generated simulator in section 7, related work in section 8, future work in section 9 and our conclusions in the last section.
2 Simulation Techniques

From previous work we are familiar with how to construct efficient simulators [1, 8, 9, 10, 11]. This knowledge we of course want to make use of when we design a simulator generation tool. Below we summarize some important techniques described in these referred papers which can be used to implement an efficient ISA-simulator.

2.1 Intermediate Format

A common way to implement a simulator is to use an interpretation loop that loops over the binary and interprets the instructions one at a time. The simulator decodes the instructions at run-time and then calls the corresponding service routine which performs the function of the instruction. If the instructions in the source code have complicated bit patterns (opcodes) which is very common for instruction sets, the decode phase could be very expensive to perform at run-time. Therefore, a better approach is to first translate the source code into an intermediate format which is much faster to interpret. Figure 1 shows this translation.

![Intermediate Format](image)

Figure 1: Mapping to intermediate format.

The major difference between the native format (the source code) and the intermediate format is that the latter is optimized for software interpretation, as opposed to hardware. In our scheme, it contains a pointer to the service routine for the instruction. With this representation we only need to decode the parameters of the instructions and perform a jump to the service routine. This makes the simulation much more efficient.

For performance reasons it is not always the best solution to have one service routine per instruction. Commonly used instructions could have special versions of service routines in
order to speed up the simulator.¹ Rarely used instructions could be brought together into one service routine where they could, for instance, share code.² This makes the mapping between instructions and service routines a rather difficult task when it comes to optimization. Our solution to this will be discussed in more details in section 5.

The instruction parameters such as register numbers, immediate values, branch offsets etc. must also be found in the intermediate format. But here we have the ability to store them in a different way which can make the simulation more efficient. We can for example pre-calculate certain transformations which otherwise must be performed at run-time.

With intermediate format we mean both what service routines there are (the mapping) and how their parameters are stored.

### 2.2 Threaded Code

With the technique explained above *threaded code* [8] can be used to make the simulator even more efficient. With threaded code no loop structure is used to make the interpretation. Instead, all necessary loop code is rolled out at the end of each service routine. This means that the last thing a service routine performs is a jump to the next one (which can be found in the intermediate format). In this way no subroutines need to be called and thus we can save a few assembler instructions.³ The simulator could be said to be executing a *thread* since it never returns from a call.

---

¹ For example if it is very common to add one, we could make a special service routine which only task is to perform that. In this way we do not need to extract the immediate value (1) from the instruction during run-time.

² This could lead to better cache utilization of the host machine running the simulator.

³ Typically we do not need to save the return address and to do the return from the subroutine. We also save the jump to the top of the interpretation loop since we do not have one.
3 The Simulator Generation Tool

In this section our concept of a simulator generating tool (ISA-level) will be presented. A more detailed description of the different parts will be given in the following sections.

3.1 Aims

The simulator generation tool should be able to:

- find a good intermediate format for fast interpretation
- generate efficient threaded C-code for the simulator
- generate assembler/disassembler
- make optimization by using statistics
- generate test suites for testing the correctness of the simulator

3.2 Design

Clearly, we need a specification of the architecture to make a simulator of. A complete ISA-specification must contain information about how instructions are coded (opcodes); the semantics of each instruction (what they should do); the syntax (for assembler/disassembler); register and memory structure. To simulate accurate timing, information about resources, e.g. caches and pipelines, needs to be added. The tool should then be able to generate all the different parts of a simulator from this specification.

This seemed to be a lot of work and not achievable within a six months thesis work. Therefore we had to focus on something at first. Since we already had a simulator, SimICS [13], which could run SPARC V8 [14] code, it was natural for us to use it as a basis and try to replace parts of it with generated ones. In this way we always had a version of the simulator which was runnable and we could always compare the generated parts with the old ones which helped us finding bugs (even in SimICS). We focused on the instruction set and on implementing SPARC V8 instructions from a specification using the simulator core with all register and condition codes as well as the memory simulation from SimICS. But all the time we had in mind that the tool should be able to generate simulators for other processors including CISC (Complex Instruction Set Computer) ones.

3.2.1 First Approach

Figure 2 shows what the tool is expected to do. The intermediate format is first created from the specification and then all the components. The decoder needs both the native
format (described in the specification) and the intermediate format since it should do the translation between the two. This translation is performed before the instruction is executed the first time. The intermediate code is then saved so that the decoder does not have to be called on subsequent invocations of the instruction. The service routines are created from the intermediate format and the assembler and the disassembler is generated directly from the specification.

Optimizations was not considered in this first approach instead we concentrated our work on integrating the generated parts with SimICS. Our specification language\textsuperscript{4} was simple but expressive; the whole SPARC V8 instruction set could be described but we only implemented a few test instructions which worked well together with SimICS. The intermediate format used a 1:1 mapping between instructions and service routines since this was the easiest to begin with.

The decoder, the disassembler and a primitive assembler worked fine as well.

### 3.2.2 Improvements

With the first approach we had something that could run so we now focused on improvements. The intermediate format defines a mapping between opcodes and service routines. This format can be changed by making specializations or generalizations of service routines. A specialization defines new service routines that handles special cases faster than the original routine. For instance a specialized version of an add instructions could be a increment instruction which always adds with one. The increment instruction can be implemented more efficiently than the add instruction since we do not need to extract the immediate value (in this case \texttt{1}) from the intermediate format. A generalization on the other hand is the opposite of a specialization. Here the same service routine is used for several instructions. This way they can share code and thus we can make better use of the instruction cache of the host machine running simulator.

When finding an efficient intermediate format, execution statistics can be used to give hints of which service routines that need to be created (which specializations and generalization we should make). Figure 3 describes our scheme of how to do this.

\textsuperscript{4}See section 4.4.1 for information about how the specification language looked like in this first approach.
Here a statistics converter is first created from the specification. The purpose of this module is to convert raw opcode statistics to instruction statistics. Opcode statistics contains information of how frequent the execution of every unique opcode (bit pattern) is, i.e., how common the instruction `add \%g2, 10, \%g7` is for example. This information could be output from a simulator or some other tool. In instruction statistics on the other hand every bit pattern matching a service routine are grouped together. A bit pattern is parsed into a set of fields with assigned values. The frequency of each field set is kept so that if a service routine is specialized into two new, the statistics is also split between them. The reason for doing this translation is that the specification gives us information about the instructions on the instruction level (the `add` instruction) rather than on the opcode level (all variants of `add`).

When the statistics converter is generated (a stand alone tool\(^6\)) the generation of the intermediate format is performed. This process could be viewed as an iteration where better and better intermediate formats are produced. For each format a simulator is created and by measuring its performance we can see if it was more efficient than the previous version. The statistics are used to see which service routine to create. For a more detailed description about this process see section 5.

As before the decoder needs information about the native format as well as the intermediate. But this decoder is a little more complicated than the previous one. Since no 1:1 mapping is used between instructions and service routines this one must be able to recognize special instances of the instructions and which instructions that should use the same service routine.

The assembler could be created directly from the specification but the disassembler uses some

---

\(^5\) SPARC V8 instruction. Adds 10 to the contents of the global register \%g2 and stores the result in global register \%g7.

\(^6\) This means that in order to create the simulator the SGT must first create another tool. The SGT has now become a meta-meta-tool.
information from the intermediate format. In this way it is able to show which special service routine a certain instruction uses. This is a nice feature when debugging and optimizing the simulator.

3.2.3 Core Interface

With the scheme described in the previous section the architecture of a generated simulator can be viewed in figure 4.

Here the shaded parts of the simulator are not generated by the tool and must thus be supplied from some other source. In our case we use SimICS. Devices include memory hierarchy, disks or other storage units, bit-mapped screens and other peripheral devices we wish to simulate. The simulator core contains all start up code for the simulator, generic event queues to control the simulator, command line interface, debugging facilities, architectural aspects such as the structure of the register file and if delay slots are used etc. The simulator core should be so generic that it could support different architecture specifications.

When an instruction needs to refer to the core part of the system – it could be a memory instruction for instance reading from memory – it uses special access primitives. These primitives are used directly in the specification and thus we make the specification independent of the user core and device parts (except the primitives of course). This means that it is very easy to replace these parts without changing the specification, e.g. adding a new timer device or changing the cache size.

The simulator needs the disassembler since it should be possible to step through a program instruction by instruction and also to disassemble parts of a program.

The helper tools consists of the same disassembler as in the simulator, an assembler and the statistics converter. The shaded part of the helper tool just controls it and does not need to be changed for different architectures.
3.2.4 Test Suites

The process of generating test suites is rather complicated since it requires deeper knowledge about the semantics of the instructions than what is necessary for generating a simulator. We must for example know that a branch instruction is a branch instruction in order to test it properly. So, generation of test suites was not considered during the time of the thesis work. A SPARC V8 suite developed earlier was used to verify correctness [12].

3.3 Implementation

The SGT, called SimGen, was implemented in C-code for maximum performance. Handling large amounts of statistics require a fast tool. However, the development time could have been shorter if we had used a language with better support for symbols and dynamic data-types. Flex and Yacc was used for generating parsers for the specification language.

3.4 Discussion

In our approach we separate the instructions which are specified in the description language from the devices and the simulator core which must be implemented by hand. This solution was not our intention from the beginning. Instead, we wanted to be able to generate the whole architecture from a single specification. Lack of time forced us to concentrate on the instruction set. However, specifying a entire system including register structure, memory hierarchy, delayed slots etc. could be a very tricky task. Especially if we want the system to be as efficient as possible. Therefore, implementing those parts by hand is not such a bad idea after all.
4 The Description Language

In this section we will talk about our specification language, how it was developed and how the final version looks.

4.1 Requirements

The specification language should

- describe the processor on the ISA-level
- be able to express RISC as well as CISC architectures
- include syntax descriptions for the instructions
- be expressive but compact
- make it possible to generate efficient simulator code
- be easy to use

We thus exclude description of

- register architecture and condition codes
- memory hierarchy
- other devices

which must be implemented by hand by the user but it should be possible to access their functionality within the specification. This could be done by using macros, functions or global variables. The service routines which will contain these references later on will be compiled and linked with the user written modules.

4.2 What Needs to be Expressed?

What must be expressed in the specification language are first, the bit patterns (opcodes) that codes the instructions (see next section for an example). This information is needed by the decoder, the disassembler and the statistics converter since they must be able to identify the different instructions in the native format. The assembler also needs it when writing the instructions to a file. Second, the semantics of instructions is of course needed and can in an abstract way be viewed as functions transforming the state of the simulated processor from one state to another. Third, the syntax of the instructions which is used by the disassembler and assembler is also needed. Instruction statistics (describing how frequently different bit patterns are used) for optimization must also be specified somewhere, but this information is better stored in a separate place since it is created by a tool rather than handwritten.
4.3 Example of Instruction Coding

Typical machine code instructions are built up by different fields which forms formats. Those formats could be of varying length which is common in CISC instruction sets or have a fixed length as in RISC instructions (typically 32 bits on a modern µ-processor). Below two instructions from the SPARC V8 architecture are shown.

Add with register (add rs1, rs2, rd)

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i</th>
<th>—</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>XXXX</td>
<td>000000</td>
<td>YYYY</td>
<td>0</td>
<td>00000000</td>
<td>ZZZZ</td>
</tr>
</tbody>
</table>

Add with signed immediate (add rs1, simm13, rd)

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i</th>
<th>simm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>XXXX</td>
<td>000000</td>
<td>YYYY</td>
<td>1</td>
<td>W W W W W W W W W W W</td>
</tr>
</tbody>
</table>

Each column in the tables represent one field. In the first row the name of the fields are given and in the second row their values (in binary). Those fields with numbers here (op, op3, i) identify the instruction. The only difference in this case between the instructions is the value of the i-field which tells whether the instruction is an “add with register” or an “add with immediate”. The fields containing the letter X, Y, Z or W have variable values and are parameters to the instruction (pointing out different registers or being immediate values). The first add instruction has a field which is not used but should be set to zero.

The semantics of the first instruction is to add the contents of register rs1 with the contents of register rs2 and stores the result in register rd. The widths of the register fields are 5 bits since the SPARC V8 architecture has 32 different registers. The second add instruction uses an immediate value as an operand instead of a register. This value is stored in simm13 which can hold values from -4096 to 4095 (twos complement). Since all SPARC V8 registers are 32 bits, this field must be sign-extended before use.

4.4 Design

4.4.1 First Approach

Our first approach to the specification language can be viewed in figure 5 where the two add instructions from the previous section are described.

The specification begins with a description of different formats used. The first is called f3a and the second f3b. Both have a width of 32 bits. Between the square brackets all fields of the format are listed together with their widths. The sum must be equal to the format width.

Then follows an instruction specification which states that the add-instruction should use format f3a and interpret rs1, rs2 and rd as parameters. The pattern declaration constrains the values of the fields op, op3 and i, i.e. they identify the add-instruction. The syntax
Figure 5: Example of a specification of the SPARC instructions “add with register” and “add with immediate”.

declaration is just C-code used for the disassembler. This code could use the parameter fields as ordinary variables. The semantics parts is also built up by C-code and is merely copied to the instruction’s service routine. Again, the parameter fields can be used as variables. The REG() construct is a macro defined in the simulator core which expands to a register reference.

The addi-instruction (add immediate) is described in the same manner. The sign_extend() is also a macro which (surprise!) sign-extends the simm13 value to 32 bits.

Drawbacks

The specification approach described here is rather straight forward. We define the format of the instructions and then the parameters, syntax and semantics. A benefit of this is that it is rather simple but it has some drawbacks too. When specifying a whole architecture this way the specification file tends to become rather large, and thus hard to read, since every single instruction must be specified although some of them only differ slightly. Take the two add instructions for example. They have two different addressing modes but their common semantics is to add the first operand to the second and then store the result in the destination. We want to have a specification on a higher abstraction level which should separate the description of instructions and addressing modes.\footnote{The Motorola 680x0 architecture, for example, has over ten different addressing modes. This could therefore be very useful.}

In the following section we will present a solution for this as well as other constructs to make it possible to generate an efficient intermediate format from the specification.
4.4.2 Improvements

Field Declaration

We have replaced the format declaration used in the first approach with field declarations. This was done mainly because we want to be able to refer to fields globally without a particular format.

```
fields <32>
  op<31:30> rd<29:25> op3<24:19> rs1<18:14> i<13:13> -simm13<12:0> rs2<4:0>
```

Here some of the SPARC V8 fields are specified. The number 32 after the `fields`-keyword is an offset and means that these fields are placed within the first 32 bits of an opcode (op uses the first 2 bits and rs2 the last 5). Several declarations with different offsets can be specified. This is useful when we want to define fields for a CISC architecture which has different format widths. The minus character before `simm13` specifies that this field should be sign extended before usage and that the tool now has this responsibility.

Intermediate Form

A new construct is the intermediate form declaration. Here it is possible to specify certain transformations that should be applied to the fields during the translation to the intermediate format. For example when accessing a register within a service routine we must multiply the register number with the register width in order to get the correct offset from the beginning of the simulated register file.\(^8\) If we can calculate this during the translation to the intermediate format rather than during run-time we can save a shift instruction for every register access. Of course more complicated transformations can be used if necessary.

```
intermediate form
  rd<9><2>  #=> `REG_OFFSET_DST(rd) << 2`#
  rs1<9><2> #=> `REG_OFFSET_SRC(rs1) << 2`#
  rs2<9><2> #=> `REG_OFFSET_SRC(rs2) << 2`#
```

Above some transformations for the SPARC register fields are shown. The macros used here are defined in the simulator core and are used to calculate the correct position within the register file.\(^9\) The numbers after the field names states that the transformed field needs a total number of 11 bits \((9 + 2)\) of which the lowest 2 always will be zero (since we multiply with 4). The tool can make use of this information when packing the fields into the intermediate format. If all fields do not fit for example it can strip some zero-bits without losing information.\(^10\)

---

\(^8\) Some simulator host architectures may have an assembler instruction for this but not all.

\(^9\) The structure of the SPARC register file (register structure) is rather complicated since it uses register windows. However some smart transformations can be used to speed up register accesses but these are not explained here. See [12].

\(^10\) It then of course needs to shift the field at run-time reproducing the zero-bits. If it was a complicated transformation however we could still gain execution time.
Combinative Context-Sensitive Macros

By using *combinative context-sensitive macros* (CCS-macros) we can specify instruction semantics on a higher abstraction level and thus get a much more readable specification. Figure 6 below shows how this could be done with our add instructions considered earlier.

```
define OP1
  fields [ rs1 ] syntax "%r{ld:rs1}" semantics # { REG(rs1) #}

define OP2
  case (i == 0) ->
    fields [ rs2 ] syntax "%r{ld:rs2}" semantics # { REG(rs2) #}
  case (i == 1) ->
    fields [ simm13 ] syntax "{ld:simm13}" semantics # { simm13 #}

define DST
  fields [ rd ] syntax "%r{ld:rd}" semantics # { REG(rd) #}

instruction add({OP1}, {OP2}, {DST})
  pattern
    (op == %10 & op3 == %000000)
  syntax
    "add {OP1}, {OP2}, {DST}"
  semantics
    # { {DST} = {OP1} + {OP2}; #}
```

Figure 6: Here three different macros are defined, one for each operand of the add instruction. The OP2-macro expands differently depending on the value of the i-field.

A CCS-macro has a case-list which assigns different meanings to the macro depending on which boolean expression evaluates to true. A case expression is built up by constraints on fields that must be satisfied in order for the corresponding case-branch to be used. Exactly one of these expressions must be true for every possible valuation of the fields. If only one meaning of the macro is requested the case expression can be omitted, e.g. OP1 and DST. Each branch has three different parts which corresponds to the context sensitivity of the macro. If it is used in the parameter list of an instruction definition the fields-declaration will replace the macro, i.e. the field list between the square brackets. If the macro is used in the syntax area the syntax-definition replaces the macro and if it is used in the semantics part of an instruction the semantics-definition replaces the macro.

The effect of using such macro is shown in figure 7 where the instruction definition for the add-instruction is expanded into two different versions.
Add With Register

<table>
<thead>
<tr>
<th>instruction add_i_0(rs1, rs2, rd)</th>
<th>pattern (op == %10 &amp;&amp; op3 == %000000 &amp;&amp; i == 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax &quot;add %r(ld:rs1), %r(ld:rs2), %r(ld:rd)&quot;</td>
<td>semantics</td>
</tr>
</tbody>
</table>

Add With Immediate

<table>
<thead>
<tr>
<th>instruction add_i_1(rs1, simm13, rd)</th>
<th>pattern (op == %10 &amp;&amp; op3 == %000000 &amp;&amp; i == 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax &quot;add %r(ld:rs1), (%ld:simm13), %r(ld:rd)&quot;</td>
<td>semantics</td>
</tr>
</tbody>
</table>

Figure 7: The add instruction definition is expanded to two different versions.

What has happened is that since the OP2-macro had two cases we got an instruction for each case. Note that the pattern part now has an extra condition on i which comes from the macro. If more than one multi-case macro is used then an instruction definition for every combination (hence the name combinatorial) of cases will be generated. In figure 8 some arithmetic and logical instructions of the SPARC V8 architecture are defined. After the macro expansion here we get a total of 16 instruction definitions (8 instructions with 2 addressing modes each). The uses keyword is used to declare the use of the ARITH-macro. Macros used as parameters are implicitly declared since this is so common.

```define ARITH
  case (op3 == %000000) -> syntax "add" semantics %{ + %#}
  case (op3 == %000100) -> syntax "sub" semantics %{ - %#}
  case (op3 == %000001) -> syntax "and" semantics %{ & %#}
  case (op3 == %000101) -> syntax "andn" semantics %{ & ~%#}
  case (op3 == %000010) -> syntax "or" semantics %{ | %#}
  case (op3 == %000110) -> syntax "orn" semantics %{ | ~%#}
  case (op3 == %000011) -> syntax "xor" semantics %{ ^ %#}
  case (op3 == %000111) -> syntax "xnor" semantics %{ ^ ~%#}

instruction arith([OP1], [OP2], [DST]) uses ARITH
  pattern (op == %10)
  syntax "{ARITH} {OP1}, {OP2}, {DST}"
  semantics %{ {DST} = {OP1}{ARITH}{OP2}; %}"
```

Figure 8: Some SPARC arithmetic and logical instructions in one definition assuming the OP1, OP2 and DST are already defined. The field-declaration part of the case branches can be skipped since no fields are used. We switch on the op3-field since it identifies the instructions.
Syntax Strings

The syntax definition used in the description language needs an explanation. A definition of the form "A{T:E}B" corresponds to the output from the C function call `printf("A%TB", E)`. E is an arbitrary C-expression with the type T. The A and B parts are ordinary text including the '%' character since an expression is placed between the curly brackets. {} is written \{\}. For example "Sum: {1d:1+2}" corresponds to `printf("Sum: %ld", 1+2)`. This representation was used because it matched well with the macros since we only have to replace the macros with the syntax strings without having to bother about C-style argument lists. We also believe it is easier to parse these syntax strings when making a full assembler but we did not have time to do within this thesis work.

Virtual Fields

Sometimes it is possible to have a more efficient but not complete representation of some part of an architecture. The states that is not representable are so unusual that we really want to make use of the more efficient technique. Since we still want to be correct we can let the simulator execute in different modes, one correct but slow and one incomplete but fast. If we detect a state that is not representable we can switch over to the slower mode and then switch back as soon as possible. The simulation of condition codes in SimICS uses this technique. Note that this is only used for simulator efficiency, i.e. representation of simulator state and must not change the behavior of the simulated processor.

Since we use different representations we need different service routines, one per mode, and thus we must be able to express this in the specification language. Our solution for this is to introduce something we call virtual fields which can be used as usual fields in case expressions or in the pattern part of an instruction definition. Instead of being a part of an opcode they represent internal states of the simulator. For every virtual field the user need to supply a function (in the simulator core for example) which defines the meaning of the field, i.e. which different values it can have and when. See appendix B for an example of the use of such function. The virtual fields are declared among the other fields as follows:

```plaintext
fields <32>
   op<31:30> rd<29:25> mode<virtual> onpage<virtual>
```

and can be used like ordinary fields:

```plaintext
instruction f007({Q}, {M}, {JH})
   pattern
      (bar == 42 && mode == 0 && onpage == 1)
   ...
```

We use a mode-field in our SPARC V8 implementation to specify whether an instruction should execute in our optimized mode for condition codes or not. If the decoder detects, by
means of a user defined function, that we are in the optimized mode a faster service routine is invoked for the current instruction which uses the more efficient representation. If a service routine finds out that the state is not representable it switches over to the slower mode and re-executes the instruction. But this time the service routine for the slower mode is used. In the same way if a slow mode service routine finds out that it is possible to execute in optimized mode it can switch back to that mode. Since some instructions now have two different service routines we need to store both service routine pointers in the intermediate format. Actually we use two formats, one for each mode.

An on-page field is used to separate branch instructions which jumps to the same memory page from those who fall off. This is done because we can implement on-page branches more efficiently than off-page branches. An on-page branch does not need to calculate the simulated physical address we should jump to by using its virtual address, i.e. going through the TLB (Translation Look-aside Buffer). Instead, we can just add a proper offset to the physical address. If the decoder detects an on-page branch a service routine implementing the more efficient target calculation is used. For more information of this see [12].

4.5 Discussion

Our main goal was to be able to generate a simulator from our specification language that should be as fast as or preferably faster than a hand-coded and hand-optimized simulator. To be as good as a human designer the tool must know what optimizations and transformations that could be applied to certain constructs of the simulated system, e.g. better representation of condition codes and how to perform on page branches more efficiently. To program a tool with this information is very tricky, if not impossible. It is not very flexible either since different optimizations could be applied to different architectures. A better approach is to let the specification language contain constructs for optimization purposes such as our intermediate form-declaration and the possibility to use virtual fields. These constructs were at first developed in order to express optimizations used in SimICS for our SPARC V8 implementation but we believe they could be as useful for other architectures.

To be able to describe the instruction semantics on a higher abstraction level as well as still being able to generate efficient code we invented the CCS-macros. From the beginning they were only intended to express different addressing modes but as figure 8 shows they can be used to express other things as well. An alternative way to express different addressing modes in a compact way could be to evaluate which addressing mode at run-time instead, e.g. using the i-field as a parameter and switch on it. In this way we also only need a single instruction definition for the two different add instructions, but we will increase execution time. The CCS-macros gives us both speed and a compact specification which is easy to read.

All text between the # and the # markers are pure C-code (besides the use of CCS- macros) and just copied to the source code of the generated simulator. This means that errors will first show up when compiling the simulator and not when the tool parses the specification. This is of course a drawback but has the advantage of being easy to write since we do not need to invent a new language for the user to express the instructions’ semantics in.
5 The Intermediate Format

Earlier we have talked about the intermediate format in very schematic way. In this section we will give a more detailed description of this topic and how it is used by the SGT.

5.1 Introduction

From section 2 we are familiar with the basic concept of an intermediate format. We make a translation to a format that is more efficient to interpret than the native because it contains pointers to the service routines implementing the instructions as well as efficiently stored parameters. But what do we mean by an instruction? The instructions used by a SPARC-processor for example have a width of 32 bits. This gives a total number of $2^{32}$ possible patterns. Should all the legal ones be viewed as different service routines (i.e. no parameters at all) or should just one service routine be used (with the 32 bits opcode as the only parameter) with a very complicated semantic? Clearly, the truth must be somewhere between these extreme cases. We cannot use $\sim 2^{32}$ service routines and by using one we have not gained so much since then the decoder has to be a part of the service routine.

When looking in an architecture manual certain fields divide the bit patterns into different instructions. It could be “add with register” or “add with immediate” for example. This division is a very intuitive one when choosing an intermediate format, but is it the best?

Suppose that it is very common to add with the immediate value 1. In this case we could have a special service routine which just does that. In this way we do not have to extract the intermediate parameter from the intermediate format during run-time since we already know the parameter equals 1. We can gain some instructions in the service routine and it also gives the compiler that compiles the service routine a chance of making further optimizations.

Say that several instructions are very seldom used. Then we can use a single service routine for them which make it possible for them to share code and thus we can make better use of the instruction cache on the host machine running the simulator.

These two cases show that we may get a more efficient simulator if we use an alternative way to map bit patterns to service routines. The use of execution statistics will help us finding such a mapping.

5.2 Requirements

- The service routine parameters should be packed in an efficient way.

- The intermediate format including service routines should be printable so it is possible for the user to see the decisions made by the tool.

- The tool should optimize the set of service routines by using statistics.
5.3 Packing Parameters

When running a service routine some fields are implicitly set to specific values, i.e. the fields used by the decoder to identify which service routine to store a pointer to in the intermediate format. These fields we call static fields since their values are never changed for a specific service routine. This means that we do not need to store them in the intermediate format. What remains are the parameter fields which holds register numbers, intermediate values, branch offsets etc. This information we of course want to store in an efficient way.

From previous work [13] we know that it is better to pack all parameters in one machine word than using one for each parameter. This has to do with slow memory loads. If we store the parameters in a single word we have to extract them by shifting and masking but we still gain execution time since these operations will be made on registers instead of memory cells. In SimICS the intermediate format is 64 bits wide, 32 bits for the service routine pointer and 32 bits for storing parameters. Since we started by implementing the same architecture as SimICS simulates (SPARC V8) the same size for the intermediate format was used. When implementing a CISC architecture for example we perhaps need to use a wider format. However, the algorithm packing the parameters is flexible enough to use any width.

What we want to minimize is of course the number of assembler instructions which extract the parameters. When we do this we must also consider aspects such as sign extension of parameters before usage, the need to use zero-bits or not, and the intermediate transformation of parameters which the user can specify. See section 4.4.2.

Our algorithm is described below and it must be pointed out that it is designed for generating extraction code to be run on the SPARC architecture host. Thus, some other packing of parameters is perhaps better for another host machine with another instruction set.

Packing Algorithm

1. The algorithm first checks to see if all parameters fit into the intermediate format when all intermediate transformations are performed before the packing, i.e. the intermediate field widths are used.

2. If the parameters do not fit then zero-bits are removed for some or all fields. The user can specify that some of the least significant bits in a field always will be zero (see section 4.4.2) and thus we only need to store the most significant bits.

3. If still the parameters do not fit some or all of the intermediate transformations must be performed at run-time instead. Of course, if we have complicated transformations we could gain execution time by storing them in separated machine words and thus be able to pre-calculate the transformation anyway. But since this require deeper knowledge of the transformations we have not considered this.
4. If the parameters cannot be packed here the algorithm gives up. However, for an architecture with fix instruction width this case is very unlikely to occur since if we use the same width for the intermediate format as for the native (excluding the service routine pointer) the parameters must fit.

5. Now, the parameters must be placed in the right order depending on their type. We use the following heuristic to decide how to order them:

- A parameter which need to be sign extended (S) should be placed to the left in the format. We thus only need to perform one arithmetic shift to sign-extend and extract the parameter. If other parameters must be sign-extended they must first be shifted up and then arithmetic-shifted down again to produce the sign.
- Zero-bits (Z) users should be placed in the middle where only a shift and a mask are enough for the extraction and reproducing of zero-bits.
- Normal parameters which do not need any transformation (P) should be placed to the right or to the left where only a mask or a shift is enough for the extraction. Of course if they must be placed in the middle, we need both a shift and a mask.
- An index (I) parameter which identifies instructions within a generalized service routine (see section 5.5.3) must be placed on the same location for all instructions sharing a service routine. We have chosen to place them in the least significant bits (on the right).
- If the service routine only has one parameter no shifting and masking is of course necessary and signed parameters could be pre-sign-extended.

This figure summarizes the parameter placements:

```
SSSSSSSSSSSSSSPPPPPPPPPPPPZZZZZZZZZZZPPPPPPPPPPPII
```

In order to extract all these parameters we need a total of 8 SPARC instructions (which are faster then 5 loads). One shift for the S-field, shift+mask for the high P-field, shift+mask for the Z-field, shift+mask for the low P-field and a mask for the index field.

Our parameter placement heuristic is not optimal. There are situations where it is possible to save one or a few instructions if we reorder the parameters in an intelligent way. For example an optimization that could be done is trying to place Z-fields at the bit-position equal to the number of zero-bits. In this way only one mask is necessary for extracting and zero-bit-extending such parameter (we mask away the bits around the field).

We could use an algorithm which generates all possible permutations of the fields and then calculates which one is the most efficient. Such method should find the optimal packing but could be rather slow.

---

11 The parameters should be stored in a memory structure in this case but this has not been implemented yet.
12 Unless we have introduced a too wide index field when generalizing. See section 5.5.3.
5.4 Storing the Intermediate Format

One of the requirements we had on the intermediate format was that it should be printable in some readable form. We did not think it was enough to look at the produced service routine to find out the actions performed by the tool and since we wanted to find a good solution by iteration over the intermediate format (see section 3.2.2) we needed to be able to read it as well. The solution to this was to store it in a text-format, both human readable and computer readable/writable. Below in figure 9 the add instruction in its intermediate form is viewed.

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>op   0 1 + 2 0 #(#)</td>
</tr>
<tr>
<td>rd   2 6 + 9 2 #(#_OFFSET_DST(rd) &lt;&lt; 2 #)</td>
</tr>
<tr>
<td>op3  7 12 + 6 0 #(#)</td>
</tr>
<tr>
<td>rs1  13 17 + 9 2 #(#_OFFSET_SRC(rs1) &lt;&lt; 2 #)</td>
</tr>
<tr>
<td>i    16 18 + 1 0 #(#)</td>
</tr>
<tr>
<td>rs2  27 31 + 9 2 #(#_OFFSET_SRC(rs2) &lt;&lt; 2 #)</td>
</tr>
<tr>
<td>simm 19 31 - 13 0 #(#)</td>
</tr>
</tbody>
</table>

Service Routine add_i_0
{
 Instruction add_i_0
 {
  Fields
   rs1 = parameter
   rs2 = parameter
   rd = parameter
   op = 2
   op3 = 0
   i = 0
  Intermediate Format
   A) rd 0 10 + 0
   B) rs2 12 20 + 2
   C) rs1 21 31 + 0
   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
   CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
  Syntax
   "add %{s:usrGetRegStr(rs1), %s:usrGetRegStr(rs2), %s:usrGetRegStr(rd)}"
  Semantics
   #
   REG(rd) = REG(rs1) + REG(rs2) ;
   #
  Statistics 0
 }
}

Figure 9: The “add with register” instruction represented in the intermediate format.

First all fields are declared; their positions in the native format, start-bit to stop-bit\(^\text{13}\), if they should be sign-extended (\(-\)) or not (\(+\)) and then what was specified in the intermediate form declaration in the specification – the intermediate width of the field, number of zero-bits and the transformation.

\(^{13}\)In the SPARC specification the bits are numbered the other way around, e.g. 31 to 25 instead of 0 - 6, as this is also used in the SPARC architecture manual [14].
After all field declarations a list of service routine definitions follows. Each service routine has in turn a list of instructions that share this service routine. In the figure the service routine called \texttt{add.i.0} only contains the instruction \texttt{add.i.0} (same name).

An instruction definition first declares which fields to use and the constraints bound to them. Next we can see how our tool has packed the intermediate parameters for the service routine. The packing must be made with respect to the intermediate form declaration in the specification, i.e. new field widths and whether zero-bits can be removed. For the add instruction the sum of the field widths is 33 bits so the removal of zero-bits is necessary here. Two bits has been removed from \texttt{rs2}.

Next comes the syntax declaration and then the semantics.

A new part compared to the specification is the statistics which is used when optimizing the simulator. Here the user can add statistics about the instructions (by means of the tool). These statistics contains information of how frequent different parameters are for the instructions. We will see how this is used in the next section.

5.5 Optimizing Using Statistics

5.5.1 Motivation

The typical action performed by a service routine implementing a triadic\textsuperscript{14} instruction (such as SPARC-add) is viewed in the following pseudo assembler code:

```plaintext
;; r0 contains the packed parameters, r10 contains the start of the
;; simulated register file

service_routine_add: /* (add gX, gY, gZ) */

and r0, MASK_P1, r1 ; Store first operand parameter in register r1.

srl r0, SHIFT_P2, r2 ; Store second operand in r2 by first shifting
and r2, MASK_P2, r2 ; and then masking.

srl r0, SHIFT_P3, r3 ; Store destination register number in r3.

ld [r10 + r1], r1 ; Load r1 with emulated register value (gX).
ld [r10 + r2], r2 ; Load r2 with emulated register value (gY).

add r1, r2, r4 ; Perform the actual instruction (add).

st r4, [r10 + r3] ; Store the result in the destination register (gZ).

epilogue(); ; The unrolled interpretation loop code. Contains 5
; assembler instructions including the loading of
; r0 for the next service routine and a jump there.
```

A total of 13 instructions. Now suppose that the \texttt{add gX, g1, gX} is very common (add the\textsuperscript{14} An instruction which uses three parameters is called triadic.

25
contents of register g1 to gX) so we make a special version of it:

```plaintext
; r0 contains now only one parameter gX (since we know we will use
; the emulated machine register g1) and can thus be used directly

service_routine_add: /* add gX, g1, gX */

ld [r10 + r0], r1    ; Load r1 with the emulated register value of gX.
ld [r10 + IM_G1], r2 ; Load r2 with the emulated register value of g1.
                        ; The offset to g1 is stored in IM_G1 (a constant now).
add r1, r2, r4        ; Perform the actual instruction (add).
st r4, [r10 + r0]     ; Store the result in the destination register.
epilogue()
```

This service routine just contains 9 assembler instructions and is thus significantly faster. If we make special versions of the most common instructions this way we will hopefully get a more efficient simulator. The ideal is to have special case of every possible instruction (∼2^{32}) but this solution is of course not realistic because the memory on our host machine will probably run out long before we get there and we will certainly trash the instruction cache somewhere along the way. Therefore we want exactly so many service routines that fit into the hosts instruction cache. Special versions of common instructions and sharing routines among unusual ones (to bring down instruction cache usage).

In the following subsections we will show how to do this by using statistics.

### 5.5.2 Specialization

The process of making special versions of commonly used service routines we refer to as **specialization**. Our SGT understands two specialization techniques and instruction statistics is used for both of them. When using the first specialization technique, the tool searches for the service routine with the highest accumulated frequency count when matching a parameter field to a fixed value. A new service routine is added where this field has been removed from the parameters and added to the static fields with the constraint that it should be equal to the value. The old service routine is kept to handle all cases where the condition on the field does not hold.

What we bypass in the special version is the extraction of the parameter which can save us two assembler instructions (a shift and a mask). We also get less parameters in the intermediate format and this could lead to better packing of the remaining fields. Perhaps no zero-bits need to be aborted for example.

The other way to specialize is to look for service routines where two or more parameters frequently have the same value. Then we only need to store one of them and thus we will
again save parameter extraction instructions. When looking at statistics this seems to be rather common. The SPARC add instruction for example very often uses the same register as source and destination.

```
Service Routine add_i,1:
{
  Instruction add_i,1:
  {
    Fields
    rs1 --- parameter simm13 --- parameter
    rd --- parameter sp --- 2
    sp --- 0 i --- 1
    Intermediate Format
      a) simm13 0 12 - 0
      b) rs1 14 22 + 2
      c) rd 23 31 + 2

    Syntax
    %#
    REG(rd) = REG(rs1) + simm13
    #}

    Statistics 26500
    # freq rs1 simm13 rd ; freq rs1 simm13 rd :
      500 -> 8 8 8 ; 5000 -> 9 1 9 :
      1000 -> 28 10 9 ; 1500 -> 30 -40 10 :
      500 -> 30 40 12 ; 5000 -> 13 1 13 :
      500 -> 13 4 13 ; 5000 -> 24 1 18 :
      1000 -> 16 8 21 ; 3500 -> 28 1 28 :
      2500 -> 25 1 25 ; 5000 -> 25 4 25 :
      500 -> 18 666 26 ; 5000 -> 26 5 26 :
      500 -> 26 15 26 ; 1000 -> 26 -13 26 :
      5000 -> 26 -11 26 ; 5000 -> 26 -7 26 :
      500 -> 26 -5 26 ; 1000 -> 26 -1 26 :
      2500 -> 28 1 28 ; 4500 -> 29 1 29 :
      1000 -> 29 12 29 :
  }
}
```

Figure 10: The original “add immediate” instruction with statistics.

Figure 10 shows the “add immediate” instruction together with instruction execution statistics. One row in the statistics table shows how frequent certain parameter values are. The instruction `add %r28, 10, %r9` occurs 1000 times, for instance.

Our tool uses a simple algorithm when deciding what specialization to perform. For each service routine it checks which parameter value occurs most frequent in the statistics. In figure 10 this is `simm13 = 1` which occurs 14500 times. The algorithm also counts how frequent parameters with the same value are. In the same figure we find `rs1 = rd` 21500 times. The constraint with highest value wins and the corresponding specialization method is used for it.

In figure 11 (right) a new service routine has been created where `rs1` is equal to `rd`. All statistics data where this condition holds has also been moved from the original service routine (left) to the new. We see that the new routine only needs two parameters, `rd` and `simm13`, and thus we save extraction instructions. The parameters are also better stored for the new routine since we do not need to remove any zero-bits here.
Further specializations can now be made either from the original routine or from the new one. The statistics show that adding with 1 seems to be very common, so a possible choice could be to make a special instance of the new routine where \( s \) is equal to \( rd \). If this is done, we almost get the same code as shown in the optimized service routine of section 5.5.1. The only difference is that since we use an immediate value, we do not need to load the value from a register which saves us another instruction.

**Compiler Optimizations**

When we specialize a parameter, we replace its occurrence in the semantics with the value or parameter it was equal to, thus replacing variables with constants or other variables. This gives the compiler that compiles the generated service routine an opportunity to make further optimizations such as constant propagation and common subexpression elimination.

We have not measured how much it is possible to gain by compiler optimizations, but we think it can make a difference especially for instructions with long and complicated semantics.

**5.5.3 Generalization**

The process of grouping uncommon service routines together we call *generalization*. The purpose of this is to minimize the instruction cache usage of the host machine running the
simulator. Doing the optimal selection is a very hard task which involves deep analysis of instruction semantics. This is beyond the scope of this work so we use an approximative method in which we identify the most infrequent used service routine by using the statistics. We then bring them together where they at least can share the epilogue code\textsuperscript{15} which is identical for most of the service routines.

A generalization is almost the reverse to a specialization. However, generalizations does not move existing static fields to parameter fields. Instead it creates a new parameter field which the service routine switches on in order to execute the correct instruction semantics. The new parameter should be placed at the same location in the intermediate format for all instructions sharing the routine, otherwise it is impossible to recognize them. See figure 15 for an example of how a generalized service routine looks.

An optimization that can be done in order to increase code sharing is to gather instructions which have their parameters packed in the same way, thus making it possible for them to share parameter extraction code. If this is used we get a trade off between instruction usage (statistics) and parameter packing similarities which we do not know how to handle. Thus, our tool does not consider this.

In the following subsection we will present a way to control specializations and generalizations.

\subsection{The Iteration Process}

In figure 12 our scheme of finding an efficient simulator is shown. From the specification the SGT (SimGen) creates the first intermediate format (intermediate.0) where all CCS-macros are expanded. We thus have one service routine per instruction. This version of the intermediate format corresponds to the native format of the architecture and it thus forms the basis for our iteration process. This is also where we create the statistics converter since it should use the native format to convert opcode frequencies to instruction frequencies.

The intermediate format is stored in the text format explained in section 5.4 and contains all information from the original specification file including the syntax definitions. Thus, we do not need the specifi-

\textsuperscript{15}The unrolled interpretation loop code.
cation file any more.\textsuperscript{16}

Instruction statistics is then added to the intermediate format by the tool. Here (intermediate.e) an instructions appears like the add-instructions in figure 10.

Now when we have statistics our iteration process can begin. The user has basically two possible choices for each iteration step:

- \textit{the number of new service routines to create by specializations}
- \textit{how many instructions to bring together by generalizations}

If the user specifies for example 10 new service routines the tool will search through the statistics for the 10 best specializations creating a new service routine for each case. The new intermediate format will then contain this new configuration. If fewer service routines are requested on the other hand the tool will make as many generalizations as needed.

For every new intermediate format a corresponding simulator can be created. Thus, the user can test the performance of the simulator at each iteration step and by making more specializations and generalizations its possible to tune the simulator for the host machine which should run it. In this way it could be possible to find the best instruction cache usage.

As we save every intermediate format on the way we can easily undo bad decisions. This process can of course be made automatic but we have not figured out a good way yet.

\textsuperscript{16}This is somewhat different from our view of the system in section 3.2.2 where the decoder used both the specification and the intermediate format for example. But in practice it is the same thing since all source information needed by the tool is just copied into the intermediate format text file. It was practical to store all information in the same place since we should find an efficient intermediate format by iteration.
6 Generated Parts

In this section we will present the different parts of the simulator, how they are generated and what their source code looks like.

6.1 Introduction

Since SimICS is written in C (with some GNU-extensions) it was natural for us to use the same language for our generated parts. We could have chosen assembler of course to get maximum control of the code but with a little experience it is rather easy to predict what kind of machine code the compiler (GCC) will produce, especially since we do not use any complicated constructs. C has also the benefit of being more readable than assembler and easier to port to other platforms.

C-code should be created for the following parts:

- Decoder
- Service Routines
- Disassembler
- Statistics Converter
- Assembler and Opcode Output Functions

As said before the remaining parts must be implemented by hand.

6.2 Main Include File

Since we wanted the source code to be readable while still being efficient we decided to generate a main include file which contains usual operations needed by the different parts. The main contents of this file are:

- extraction macros for the native format fields
- intermediate format packing macros for each instruction
- extraction macros for the intermediate format fields
Native Format Extraction Macros

The purpose of these macros is to extract fields from the native format. They are used by the decoder, disassembler and the statistics converter which all need to examine these fields to achieve their tasks. Each macro takes a variable in which the extracted field will be stored and a char-pointer to the beginning of an op code. This way it is easy to use variable length op code which are used in CISC instructions. The following two macros shows the extraction of the SPARC V8 rd (A) and simm13 (B) fields.

```c
/* .AAAAA. ........ ....BBBBB BBBBBBBB */
#define EXTRACT_NATIVE_rd(rd, code) \ 
  { \ 
    rd = ((code[0] >> 1) & 0x1f); \ 
  }
#define EXTRACT_NATIVE_simm13(simm13, code) \ 
  { \ 
    simm13 = (code[2] >> 0) & 0x1f); \ 
    simm13 = (simm13 << 0) | ((code[3] >> 0) & 0xff); \ 
    simm13 = sign_extend_int32(simm13, 13);  \ 
  }
```

The first macro has to shift the first op code-byte one bit to the right and then mask off the most significant bits in order to extract the rd-field. The second macro is a little more complicated since the simm13-field covers more than one byte. It must also be sign-extended which is done by the macro

```c
#define sign_extend_int32(v, w) \ ((long)((v) << (32-(w)))) >> (32-(w)))
```

which first shifts the field v up and then arithmetic-shifts it down again to produce the right sign.

Since these macros are automatically generated by a rather general algorithm (which is not presented here) a lot of unnecessary code is laid out such as shifting fields zero bits and masking a byte with 0xff. We have not spent any time for removing this code; we leave this to the compiler, i.e. RISC.³⁷

Intermediate Format Packing Macros

Intermediate format packing macros are used, as indicated by their name, to pack parameter fields into the intermediate format. The decoder uses them for this task. Below we show the macros used for the add immediate instruction which we have considered earlier.

³⁷ Relocating Interesting Stuff to the Compiler.
The macro first performs the intermediate transformations for `rs1` and `rd` as specified by the user in the specification. In order for all fields to fit $(13 + 11 + 11 > 32)$ two zero-bits are used for them; thus they are shifted down 2 bits each.\(^\text{18}\) All fields are then packed into the intermediate format.

We will get a macro like this for all different instructions which results in a rather big include file (~ 4000 lines for SPARC V8 without specializations). This could be cut down by letting instructions with the same intermediate format share packing macros. This has however not been done.

### Intermediate Format Extraction Macros

These macros are the most time critical ones since they are used within the service routines to extract parameters. Below the extraction macros are shown for the same add instruction which was packed above:

```c
/* simm3  rs1  rd */
/* AAAAAAAAA AAA A/.B BB BBB BBB CCC CCC CC C*/
#define EXTRACT_INTERMEDIATE_rs1_B(w, code) w = (code >> 7) & 0x7fc;
#define EXTRACT_INTERMEDIATE_rd_A(w, code) w = (code << 2) & 0x7fc;
#define EXTRACT_INTERMEDIATE_simm3_A(w, code) w = ((long)code >> 19));
```

The first macro extracts the `rs1`-field by first shifting it 7 bits to the right and then masking of the bits around it, thus producing zero-bits as well. The second macro works in the same way except the shift which goes in the other direction. `simm3` is extracted and sign-extended with only one shift.

Unlike the packing macros the intermediate extraction macros have been designed to be shared by several instructions. But since a field can be packed differently for different service routines several macros can exist for the same field, therefore a letter (here A, B which has nothing to do with format showed in the comment) is appended to the macro names to distinguish them from each other.

\(^\text{18}\)It looks pretty stupid to first shift the fields up two bits in the transformation and then back again, but since we do not analyze the user written transformation things like this could occur. However, the decoder is not time critical since it is only run once per instruction.
The include file also contains other things needed such as constant definitions, prototypes, inclusions of user written headers etc.

6.3 The Decoder

The decoder’s task is to identify the different instructions in the native format and then construct the corresponding intermediate format for them. The naïve way of doing this is to loop through all instructions for an incoming opcode and test the static fields until a match occurs. Then we extract the parameter fields so we can do intermediate translation. Since many instructions uses the same fields a lot of tests will be duplicated this way. For example the two add-instructions we have looked at several times only differs in the i-fields, thus it would be a waste of time to check the op and op3-fields for both of them. To avoid this we first build a decode tree in which a branch corresponds to an instruction with a certain format and certain static values. In this way we will prevent duplicated tests.

![Decode Tree Diagram]

Figure 13: A decode tree for 9 different instructions.

Figure 13 shows how such tree could look for the following instructions: (A) specialized version of add where rs1 = rd, (B) specialized version of add immediate where rs1 = rd and simm13 = 1, (C) specialized version of add immediate where rs1 = rd, (D) add, (E) add immediate, (F) subcc\(^\text{19}\) in mode 0, (G) subcc in mode 1, (H) subcc immediate in mode 0, (I) subcc immediate in mode 1.

Here we also see the usage of virtual fields. The subcc instructions have two different versions, one for each mode which corresponds to different representation of condition codes.

\(^{19}\) Subtracts but also sets condition codes.
The circle marks the root node of the tree. A rounded rectangle represents a new field in the format and corresponds to the extraction of the field in the code to generate. The first one is called op and occurs in all SPARC V8 instructions. None of these nodes has any siblings here but this could be the case if several different fields start at the same bit-position. The rectangle formed nodes are children of the field nodes and represent different values the fields have in different instructions. These nodes correspond to tests on the fields in the code to generate. For example the op could have values from 0 to 3 and simm13 could be equal to 1 (in the specialized instruction B). A “P” means that a field is used as a parameter. Each leaf in the tree represents a unique instruction and the action to perform when we have found it. That is here to pack the parameters and set the service routine pointer for the intermediate format.

When the decode tree has been built it is rather easy to generate the decoder from it. We only have to traverse the tree in in-order and lay out the proper code. In appendix B the corresponding code to figure 13 is shown together with some comments.

It could be possible to rearrange the tree and test the different fields in some other order. For example the most used fields first. This way we could split the search space faster and thus get a more efficient decoder. But we have not implemented this since the decoder is not the most time critical in the system (it is only used once per instruction).

### 6.4 The Service Routines

```c
void service_routines()
{
    entry_points[SIM_ENTRY_INDEX_add_i_1_rs1_rd_simm13_1] = &SIM_ENTRY_POINT_add_i_1_rs1_rd_simm13_1;
    entry_points[SIM_ENTRY_INDEX_add_i_1_rs1_rd] = &SIM_ENTRY_POINT_add_i_1_rs1_rd;
    entry_points[SIM_ENTRY_INDEX_add_i_1] = &SIM_ENTRY_POINT_add_i_1;
    return;
    ///* Service Routines */
}
```

Figure 14: Building jump table for the service routines. The &&-operator means the address of the label (GNU-extensions to C).

From the intermediate format text file it is rather straightforward to create the service routines. Basically we put all of them in one C-function (service routine function) with labels pointing out their positions. Since the GNU-extension of C let us take the address of a label it is easy to build up a jump table for the service routines. This is done by letting the initialization phase of the simulator call the service routines function which then stores the addresses in a table (array). This procedure is shown in figure 14 and is necessary because labels cannot be referred to globally. The decoder uses the jump table to set the service routine pointers in the intermediate code.

---

20 Actually, this could be any constraint on the field such as op > 7, op != 3 etc. But this is not necessary for describing the SPARC-architecture.
In figure 15 we show some generated service routines. To the left we have three different versions of the add immediate instruction. The first is the general one which can handle all cases. It begins with extracting the three parameters \texttt{simm3}, \texttt{rs1} and \texttt{rd} from the variable \texttt{rOP} which holds the parameter part of the intermediate format. The instruction’s semantics and the epilogue are then executed. The next routine is used whenever the condition \texttt{rs1 = rd} holds. Note that \texttt{rs1} has been replaced by \texttt{rd} in the semantics. The last service routine is specialized a step further with \texttt{simm3} equal to \texttt{1}.

The service routine on the right side in the figure is an example of a generalized one. It is composed by the three SPARC V8 instructions \texttt{and}, \texttt{or} and \texttt{xor}. An extra index field, \texttt{INDEX}, needs to be introduced which we can switch on to reach the correct semantics. In principle we have replaced the old static fields by a new one in order to distinguish between the instructions. The new field is however smaller and easier to decode than the old ones.

The purpose of generalizing is, as mentioned before, to bring down instruction cache usage by

\footnote{The epilogue checks for interrupt events, handles them if necessary, and branches to the next instruction after copying its parameters into \texttt{rOP}.}
sharing code. In this case we were lucky since all parameter extraction code could be shared. If the parameters are packed differently for each instruction they have to be extracted separately within the corresponding case block. The index field has of course the same location.

The intermediate format of these instructions are shown below:

<table>
<thead>
<tr>
<th>Intermediate Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>A) rs2</td>
</tr>
<tr>
<td>B) rs1</td>
</tr>
<tr>
<td>C) rd</td>
</tr>
<tr>
<td>D) INDEX</td>
</tr>
</tbody>
</table>

Here the two middle fields need two zero-bits each. The index field is two bits wide since we must be able recognize three different instructions.

6.5 The Disassembler

The disassembler is generated in the same way as the decoder but instead of building the intermediate format it writes the instruction’s syntax (our syntax strings are easily converted to printf statements). It also ignores the virtual fields since these are not a part of the native format.

The following figure shows how output from the assembler could look:

| <0x00000078> | ld | %o6 + 64], %10 |
| <0x0000007c> | add | %o6, %6, %11 |
| <0x00000080> | sub | %o6, %32, %o6 |
| <0x00000084> | orcc | %g0, %g1, %g0 |
| <0x00000088> | be | 0x10098 |
| <0x0000008c> | or | %g0, %g1, %e0 |
| <0x00000090> | call | 0x102cc |
| <0x00000094> | nop | |
| <0x00000098> | sethi | %hi(0x21400), %e0 |

6.6 The Statistics Converter

The statistics converter is also generated by using a decode tree but here we must take care of the virtual fields. This has to do with the fact that we do not have any statistics which tells us how frequent different virtual field values are. Remember that virtual fields represent internal states of the simulator which cannot be derived from the native format. Typically a virtual field corresponds to different modes in the simulator. The instructions which can execute in those different modes have one service routine per mode (since we have different semantics for each mode), and thus it is necessary to spread the instruction’s statistics among them according to their relative usage. Our generalization and specialization-algorithms will not work if this is not done.

The user must measure these frequencies in some way or estimate them. Then they are given to the statistics converter which processes the statistics accordingly.
For example if the simulator has two modes and runs in mode one 95% of the time, we supply this information to the statistics converter. If we have one instruction `subcc X,Y,Z` with frequency 1000, we will get `subcc_mode_1 X,Y,Z` with frequency 950 and `subcc_mode_2 X,Y,Z` with frequency 50 after the conversion.

This is an approximation since all instructions for a certain mode will get the same relative division. For example, if the simulator could give us a better information about the usage of certain instructions in different modes we could do better. This has however not been done yet.

6.7 The Assembler and Output Functions

We generate a very simple assembler which understands instructions in the format:

```
name parnum1 parnum2 ... parnumN
```

where `name` is the name of the instruction after the expansion of the combinative macros. For example `add i 0`. The parnum are the corresponding parameter values in decimal.

When the assembler has identified an instruction it calls the instruction's output function which writes the opcode to a file. An example of such a function is the following:

```c
void out_add_i_0(FILE *out, u_long rs1, u_long rs2, u_long rd)
{
  u_long w;

  w = 0;
  w = (w << 2) | 0x2;
  w = (w << 5) | ((rd >> 0) & 0x1f);
  w = (w << 6) | 0x0;
  w = (w << 5) | ((rs1 >> 0) & 0x1f);
  w = (w << 1) | 0x0;
  w = (w << 8) | 0x0;
  w = (w << 5) | ((rs2 >> 0) & 0x1f);

  fwrite((char *)&w, sizeof(u_long), 1, out);
}
```

The assembler was mainly created to test the decoder and disassembler. But the output functions could be useful if we want to generate test suites in the future.
7 Performance

In order to measure the performance of a generated simulator and test how much it is possible to gain by specializations we have run the GCC benchmark from SPEC 95 on a generated simulator. This benchmark is the optimization part of the GNU C-Compiler GCC. To keep down the execution time, we only use peep-hole optimization and thus we use a modified version of the benchmark.

We first collected execution statistics of instructions from the benchmark and then we generated different versions of the simulator using different numbers of specializations. This way we managed to get the simulator 2 % better than the hand-optimized version (classic) and 3 % better than itself when no specializations was used. The simulator only executed in the service routines 31 % of the total time; the rest was spent in other parts of the simulator. Thus, we get a more fair picture if we only consider the time spent in the service routines, i.e. the part of the simulator which we can affect with specializations and generalizations. The following diagram shows this time for different version of the simulator when run on a Super-SPARC (40 MHz).

![Figure 16: The performance of different specialized simulators on the SPEC 95 benchmark GCC.](image)

We see that with no specializations the generated simulator is slower than the classic one, but with only 10 specializations it has become faster. We get the most efficient simulator by 25 specializations. This one has 10 % faster service routines compared to the one with no specializations and 7 % faster than the service routines in the classic simulator. With 30 specializations something happens. The most likely reason for this performance degradation is that we trash the instruction cache of the host machine. We have not examined if gener-

\[22\] A SPARC V8 user level simulator. Floating point and co-processor instructions were not implemented.
alizations could change the performance here to the better. However, we have learned that the use of the instruction cache is extremely important for the performance. If the instruction cache is badly used the effect of specializations is reduced. Therefore, we first sort the service routines by the execution statistics so that frequently used service routines are placed consecutively in memory. This optimization increases the speed of the service routines by a few per cent.

We have also optimized simulators for some other small applications and the behavior seems to be the same—the simulator gets faster when we specialize. However, more programs should be tested to evaluate the importance of specializations and generalizations.

Figures 17: This diagram shows how many service routines that correspond to 50, 75, 90, 95 per cent of the total service routine calls for different specialized simulators. From the GCC benchmark.

The diagram in figure 17 shows that the instruction usage of the GCC benchmark is very skewed, i.e. a very small amount of instructions stand for large usage. For example only 7 service routines (= instructions when no specializations are made) out of 325 stands for 50% of the total service routine calls and only 41 for 95%. This motivates specializations since they will be applied to the far most used instructions. We can also see in the diagram that when we specialize a service routine (split it into two parts) the more general one seems to be quite common too. With 100 specializations, i.e. 100 new service routines, almost all of the more general ones stayed among the 95% most called service routines and half of them stayed among the 75% most called. However, only around 20 stayed among the 50% most called.
8 Future Work

A complete simulator generation tool is a rather complex system and since this work have been carried out as a six month thesis much work still needs to be done. In the following list we summarize some of the most important things that could be improved or added to the system in the future.

- Augment the specification language with constructions for specifying register structure and memory hierarchies as well as the structure of the TLB and paging system; if the processor is a little endian or big endian; include instruction timing information with the ability to specify resources such as functional units and pipelines; adding constructs for specifying delay slots and instruction issue etc. Clearly, a lot of work can be done here but we must not forget that the tool need to able to generate efficient code for the new part as well.

- Implement simulators for other architectures to evaluate the generality, usability and utility of the specification language.

- Generate test suites to verify the correctness of the simulator. This could be done by generating programs which contains all implemented instructions with random values as parameters as well as critical values such as MIN_INT, -1, 0, 1, MAX_INT. Of course, we need to be careful with branches and memory addresses. The test programs should write their state at given times to a file. They can then be run both on the simulator and on a native system. The written states can be compared to verify at least that the test programs runs correctly.

- Make the generation of a full assembler possible. Could be useful when test suites should be created.

- Assign a cost function for each parameter which corresponds to how much we can gain if we specialize it. This cost function should depend on the statistics, of course, but also on parameter type and placement within the intermediate code. For example, we gain more if we specialize a parameter which has to perform its intermediate transformation at run-time (due to compiler optimizations that can be performed) than for one which do not have such or has it pre-calculated.

- Better analysis of how specializations and generalizations affect the performance of the generated simulator. Collect execution statistics from a large set of programs and then measure the performance of the simulator running other programs. Investigate how important the placement of service routines is for the instruction cache usage of the host machine.

- Make our iteration process for optimizing the simulator automatic.
9 Related Work

As mentioned in the introduction there has not been much work on simulator generation tools for the instruction set architecture level. There exists a few but their main approach has not been focused on how to generate an efficient simulator which was the main goal with this thesis work.

Tood A. Cook and Ed Harcourt [6, 7] has developed a functional programming language, LISAS, which is used as a specification language for an instruction set architecture. The language contains constructs for describing instruction formats, declaring memory sizes with word lengths and simple register files. The semantics of an instruction is implemented by a function in the language which uses other helper functions to determine different addressing modes etc. When simulating, the specification file (program) is simply executed. We have not seen any measurements of how efficient this approach is and if it is possible to compile this language to efficient code. The specification language lacks syntax description for the instructions.

Another tool is the Visualization-based Microarchitecture Workbench (VMW) by Trung A. Diep [15]. It uses templates to specify how instructions are coded and which fields to use as parameters. The templates contains hexa-decimal mask values which is used to identify different fields. An API (Applications Programming Interface) is defined for C++ which is used to help the programmer implement the simulator. The API includes functions for getting information about the next instruction to execute, the contents of memory addresses and caches, if some functional units are occupied etc. Thus, no description language is used to specify the architecture, instead the user programs the simulator in C++ using this API. A nice feature of the tool is that it has a graphical user interface for controlling the simulator. We have not found any performance measurement of an implemented simulator using the VMW.

Some people have addressed subsets of the problem, such as the New Jersey Machine-Code Toolkit [16] (NJMCTK). The NJMCTK is used for helping programmers writing applications which process machine code, i.e. assemblers, disassemblers and debuggers. The user can specify how instructions are coded in a description language and then the tool is able to generate code for an instruction encoder as well as a decoder. At first we considered to use the tool for decoding native instruction and producing our intermediate format but it turned out to be too inflexible. However, we have borrowed some ideas and terminology from the NJMCTK.

Another work that could be mentioned here is Automatic Generation of Assemblers by John D. Wick [17].
10 Conclusion

We now have a tool which will be very helpful during the development of efficient simulators. The specification language provides a compact but expressive way to describe different instruction set architectures and our experience so far indicate significantly shorter development times with much less errors.

Furthermore, our test results show that a generated simulator can be faster than a hand-coded and hand-optimized one. Thus, we do not lose in performance here, which usually is the main dilemma when using utilities of this kind.
A An Example of a Specification - SPARC

The following text is an example of a description file which describes a subset of the SPARC V8 instruction set architecture.

// Declaration of the positions and names of the different fields.
// The disp2on and disp2off fields are actually the same but since
// different intermediate transforming for on-page and
// off-page purposes (see below) will be used the field is coded as two
// fields</3>
// op/2</2> rd/</2>/</2> op/2</2> rs/</2>/</2> i/</2>/</2> -simm/</2>/</2> rs/2</2>/</2> a/</2>/</2> cond/</2>/</2> op/2</2> /-disp/2>/</2>/</2> /-disp/2>/</2>/</2> imm/</2>/</2> shcnt</2>/</2> disp/2/on/</2>/</2> disp/2/off/</2>/</2> trap</2>/</2> mode/virtual/<virtual/>
// Some intermediate transformations to speed up the simulator */
intermediate form
r0<2><2> # ( REG_OFFSET_DST(rd) << 2 #)
r1<2><2> # ( REG_OFFSET_SRC(rs1) << 2 #)
r2<2><2> # ( REG_OFFSET_SRC(rs2) << 2 #)
r0<9>/</2>/</2> # ( imm<2>2 /</2>/</2>/</2>/</2>
// Transform for an on-page branch.
disp2off<2><2> # ( disp2off << 2 #) // Transform for an on-page branch.
// Definition of some useful CCS-macros */
define OP1
fields [ rs1 ] syntax "'(s:in/(rs1,0,7) /'g:in(rs1,8,15) /'s:in(rs1,16,23) /'i:in(rs1,24,31))'(1d:rs1 % 8)'
semantics # ( REG(rs1) /#/)
define OP2
  case i -- 0 ->
    fields [ rs2 ]
syntax "'(s:in/(rs2,0,7) /'g:in(rs2,8,15) /'s:in(rs2,16,23) /'i:in(rs2,24,31))'(1d:rs2 % 8)'
semantics # ( REG(rs2) /#/)
case i -- 1 ->
  fields [ simm3 ]
syntax "'(id:simm3)'
semantics # ( simm3 /#/)
define DST
fields [ rd ] syntax "'(s:in(rd,0,7) /'g:in(rd,8,15) /'s:in(rd,16,23) /'i:in(rd,24,31))'(1d:rd % 8)'
semantics # ( REG(rd) /#/)
define BDD
  case mode -- 0 -> semantics #(0#)
case mode -- 1 -> semantics #($#)
/* Used to select the correct primitive when branching on-page or off-page */
define OR_ON_OFF_PAGE
  case onpage -- 0 -> semantics #(OFF#)
case onpage -- 1 -> semantics #(ON#)
/* Used to code if a branch is annulled or not */
define ANNUL
  case a -- 0 -> syntax "
case a -- 1 -> syntax ".a" semantics #(annull_epilogue/() /#)
/* Different fields are used depending on on-page or off-page branches */

define DISP/2/2
  case onpage /=/= 0 -> fields [disp/2/2 syntax '0
  case onpage /=/= 1 -> fields [disp/2/2 off] syntax '1

/* The names of all conditional branch instructions */

define BRANCH_NAMES
  case cond /=/= %1/0/0/1 -> syntax 'bne
  case cond /=/= %0/0/0/1 -> syntax 'be
  case cond /=/= %1/0/1/0 -> syntax 'bg
  case cond /=/= %0/0/1/0 -> syntax 'ble
  case cond /=/= %1/0/1/1 -> syntax 'bge
  case cond /=/= %0/0/1/1 -> syntax 'bl
  case cond /=/= %1/1/0/0 -> syntax 'bgu
  case cond /=/= %0/1/0/0 -> syntax 'bleu
  case cond /=/= %1/1/0/1 -> syntax 'bcc
  case cond /=/= %0/1/0/1 -> syntax 'bcs
  case cond /=/= %1/1/1/0 -> syntax 'bpos
  case cond /=/= %0/1/1/0 -> syntax 'bneg
  case cond /=/= %1/1/1/1 -> syntax 'bvc
  case cond /=/= %0/1/1/1 -> syntax 'bvs

/* The condition that must hold in order for a branch to be taken. The first 14 are used in our optimized mode for condition codes and the last 16 are used in the other mode. */

define COND
  case mode /=/= 0 & & cond /=/= %1/0/0/1 -> semantics #/ if (/((CMP_VALUE_A != CMP_VALUE_B))
  case mode /=/= 0 & & cond /=/= %0/0/0/1 -> semantics #/ if (/((CMP_VALUE_A == CMP_VALUE_B))
  case mode /=/= 0 & & cond /=/= %1/0/1/0 -> semantics #/ ((int)CMP_VALUE_A > (int)CMP_VALUE_B)
  case mode /=/= 0 & & cond /=/= %0/0/1/0 -> semantics #/ ((int)CMP_VALUE_A < (int)CMP_VALUE_B)
  case mode /=/= 0 & & cond /=/= %1/0/1/1 -> semantics #/ ((int)CMP_VALUE_A >/= (int)CMP_VALUE_B)
  case mode /=/= 0 & & cond /=/= %0/0/1/1 -> semantics #/ ((int)CMP_VALUE_A </= (int)CMP_VALUE_B)
  case mode /=/= 0 & & cond /=/= %1/1/0/0 -> semantics #/ (CMP_VALUE_A > CMP_VALUE_B)
  case mode /=/= 0 & & cond /=/= %0/1/0/0 -> semantics #/ (CMP_VALUE_A < CMP_VALUE_B)
  case mode /=/= 0 & & cond /=/= %1/1/0/1 -> semantics #/ ( CMP_VALUE_A >/= CMP_VALUE_B)
  case mode /=/= 0 & & cond /=/= %0/1/0/1 -> semantics #/ ( CMP_VALUE_A </= CMP_VALUE_B)
  case mode /=/= 0 & & cond /=/= %1/1/1/0 -> semantics #/ /((rCMP_VALUE_A - rCMP_VALUE_B) >/> 31) /=/= 0
  case mode /=/= 0 & & cond /=/= %0/1/1/0 -> semantics #/ /((rCMP_VALUE_A - rCMP_VALUE_B) >/> 31) /=/= 1
  case mode /=/= 0 & & cond /=/= %1/1/1/1 -> semantics #/ /((rCMP_VALUE_A /- rCMP_VALUE_B) ^/ rCMP_VALUE_A) /=/= rCMP_VALUE_B
  case mode /=/= 0 & & cond /=/= %0/1/1/1 -> semantics #/ /((rCMP_VALUE_A /- rCMP_VALUE_B) ^/ rCMP_VALUE_A) < rCMP_VALUE_B
  case mode /=/= 1 & & cond /=/= %1/0/0/1 -> semantics #/ !ccode_z
  case mode /=/= 1 & & cond /=/= %0/0/0/1 -> semantics #/ ccode_z
  case mode /=/= 1 & & cond /=/= %1/0/1/0 -> semantics #/ !(ccode_z | | (ccode_n != ccode_v))
  case mode /=/= 1 & & cond /=/= %0/0/1/0 -> semantics #/ (ccode_z | | (ccode_n != ccode_v))
  case mode /=/= 1 & & cond /=/= %1/0/1/1 -> semantics #/ ccode_n == ccode_v
  case mode /=/= 1 & & cond /=/= %0/0/1/1 -> semantics #/ ccode_n != ccode_v
  case mode /=/= 1 & & cond /=/= %1/1/0/0 -> semantics #/ ![ccode_c & ccode_z]
  case mode /=/= 1 & & cond /=/= %0/1/0/0 -> semantics #/ ccode_c & ccode_z
  case mode /=/= 1 & & cond /=/= %1/1/0/1 -> semantics #/ ![ccode_c]
  case mode /=/= 1 & & cond /=/= %0/1/0/1 -> semantics #/ ccode_c
  case mode /=/= 1 & & cond /=/= %1/1/1/0 -> semantics #/ ![ccode_n]
  case mode /=/= 1 & & cond /=/= %0/1/1/0 -> semantics #/ ccode_n
  case mode /=/= 1 & & cond /=/= %1/1/1/1 -> semantics #/ ![ccode_v]
  case mode /=/= 1 & & cond /=/= %0/1/1/1 -> semantics #/ ccode_v

/* This is an interesting definition which defines all SPARC V8 conditional branches. After the expansion of the CCS-macro, we get a total of 112 instruction definitions. The COND has 28 different entries, ANNUL and ON_OR_OFF_PAGE has 2 each, and 112. JUMP_REL_ON_PAGE and JUMP_REL_OFF_PAGE are primitives defined in the simulator core */

instruction branches (DISP/2) uses JUMP_REL_ON_PAGE, JUMP_REL_OFF_PAGE, ANNUL, CORD

pattern "(BRANCH_RACES)($ANNUL)(DISP/2)"
  semantics "$
  if ($CORD)
    $JUMP_REL_ON_OFF_PAGE_PAGE $((DISP/2));
    ($ANNUL)
  $
  $}
instruction sethi(imm2, DST)
  pattern
    (op -- 00 00 sp2 -- 0000)
  syntax
    "sethi %hi(0x{imm2 << 10}), (DST)"
  semantics
    $(DST) = imm2; #)

/* MEMORY_LOAD_g is a core primitive to load an address from memory */

instruction ld(OP1, OP2, DST)
  pattern
    (op -- 01 00 sp3 -- 00000000)
  syntax
    "ld [OP1] + (OP2), (DST)"
  semantics
    #(MEMORY_LOAD_g(DST), (OP1) + (OP2), u_long, ld, Xreg); #)

instruction add(OP1, OP2, DST)
  pattern
    (op -- 00 00 sp3 -- 00000000)
  syntax
    "add (OP1), (OP2), (DST)"
  semantics
    #((DST) = (OP1) + (OP2); #)

/* Subcc has different semantics depending the mode used. */

instruction subcc(OP1, OP2, DST)
  pattern
    (op -- 01 00 sp3 -- 000100 00 mode -- 0)
  syntax
    "subcc (OP1), (OP2), (DST)"
  semantics
    #
    rCMP_VALUE_A = (OP1);
    rCMP_VALUE_B = (OP2);
    (DST) = rCMP_VALUE_A - rCMP_VALUE_B;
    #)

/* If we are in nopt-mode, just switch back to opt-mode and re-execute the instruction since the result of an subcc always can be represented in opt-mode. */

instruction subcc()
  pattern
    (op -- 01 00 sp3 -- 000100 00 mode -- 1)
  semantics
    #
    switch_to_opt();
    #)
B An Example of a Decoder

This is an example of a decoder for some variants of the add and subcc instructions (defined in appendix A). Different degrees of specialisations have been applied to the add instructions. The subcc uses virtual fields since it has different semantics for different modes.

```c
long sgDecode(INTERMEDIATE_CODE p, u_char *code, CPU_STATE __state)
{
  u_long
  sp, r6, r3, r1, i, r2, mode;
  long
  simm3;

  EXTRACT_BVATE_sp(sp, code);
  if (sp == 2)
  {
    EXTRACT_BVATE_sp3(sp3, code);
    if (sp3 == 0)
    {
      EXTRACT_BVATE_rsi(rsi, code);
      if (REG_OFFSET_SRC(rsi) <= 2) -- (REG_OFFSET_DST(rdi) <= 2)
    }
  }

  EXTRACT_BVATE_i(i, code);
  if (i == 0)
  {
    EXTRACT_BVATE_rsi2(rsi2, code);
    PACK_INTERMEDIATE_add_i_rsi_rsi2(code-(parameters, rsi2, code));
    return 0;
  }

  if (i == 1)
  {
    EXTRACT_BVATE_simm3(simm3, code);
    if (simm3 == 1)
    {
      PACK_INTERMEDIATE_add_i_rsi_rsi2_simm3_i(code-(parameters, rsi2, simm3, code));
      return 0;
    }
  }

  PACK_INTERMEDIATE_add_i_rsi(rsi, code-(parameters, rsi, i, r2, code));
  p->entry_ptr = v9_entry_print[SEG_ENTRY_INDEX_add_i_rsi_rdi];
  return 0;
}

EXTRACT_BVATE_i(i, code);
if (i == 0)
{
  EXTRACT_BVATE_rsi2(rsi2, code);
  PACK_INTERMEDIATE_add_i_rsi2(code-(parameters, rsi2, code));
  p->entry_ptr = v9_entry_print[SEG_ENTRY_INDEX_add_i_rsi2];
  return 0;
}

if (i == 1)
{
  EXTRACT_BVATE_simm3(simm3, code);
  PACK_INTERMEDIATE_add_i_rsi_simm3_i(code-(parameters, simm3, rsi, code));
  p->entry_ptr = v9_entry_print[SEG_ENTRY_INDEX_add_i_rsi2];
  return 0;
}
}

if (sp == 20)
{
  EXTRACT_BVATE_rsi(rsi, code);
  EXTRACT_BVATE_i(i, code);
  if (i == 0)
  {
    EXTRACT_BVATE_rsi2(rsi2, code);
    mode = mode VirtualFieldMode(__state, "subcc_i_0", "r6", "r6", "r1", "r2", 0);
    PACK_INTERMEDIATE_subcc_i_0_mode_0(code-(parameters, rsi2, code));
    p->entry_ptr = v9_entry_print[SEG_ENTRY_INDEX_subcc_i_0_mode_0];
    return 0;
  }
}
```

Above the corresponding decoder of figure 13 is shown. It extracts the native fields and compare them to different values in order to identify the incoming instruction. When a field is used as a parameter field it is just extracted since no comparison is necessary. However, since specializations are used, the same field can act as both static and as a parameter for different service routines. For example on line 29 the `simm13` is checked against 1 to see if the specialized service routine should be used or not.

On line 16 two fields (after the application of the user specified intermediate transformations) are compared to see if they are equal. This test is performed to see if a specialized service routine which uses equal parameter values should be used.

The use of a virtual field is shown on line 63 and below. The variable `mode` (virtual field) is set to the value returned by the user defined function `usrGetVirtualField`. Such a function gets the state of the CPU as a parameter as well as the name of the instruction and its parameters with corresponding values. The function can now return an appropriate value based on this data. In this case the mode of the simulator is returned (see the virtual field part of section 4.4.2). We can now test if a service routine for mode 0 should be used or one for mode 1.

`v9_entry_points` is an array which holds all service routine pointers. When the right service routine is found for an instruction the corresponding pointer is stored in the intermediate format together with the packed service routine parameters.
C  An Example of Generated Service Routines

This example shows the corresponding service routines for those instructions decoded in appendix B.

```c
#include "v3/include.h"

void v3_service_routines_t()
{
    v3_entry_points[SIM_ENTRY_INDEX_add_i_0_rsi_rdi] = &SIM_ENTRY_POINT_add_i_0_rsi_rdi;
    v3_entry_points[SIM_ENTRY_INDEX_add_i_0] = &SIM_ENTRY_POINT_add_i_0;
    v3_entry_points[SIM_ENTRY_INDEX_add_i_1_rsi_rdi_simm13_1] = &SIM_ENTRY_POINT_add_i_1_rsi_rdi_simm13_1;
    v3_entry_points[SIM_ENTRY_INDEX_add_i_1_rsi_rdi] = &SIM_ENTRY_POINT_add_i_1_rsi_rdi;
    v3_entry_points[SIM_ENTRY_INDEX_add_i_1] = &SIM_ENTRY_POINT_add_i_1;
    v3_entry_points[SIM_ENTRY_INDEX_subc_i_0_mode_0] = &SIM_ENTRY_POINT_subc_i_0_mode_0;
    v3_entry_points[SIM_ENTRY_INDEX_subc_i_0_mode_1] = &SIM_ENTRY_POINT_subc_i_0_mode_1;
    v3_entry_points[SIM_ENTRY_INDEX_subc_i_1_mode_0] = &SIM_ENTRY_POINT_subc_i_1_mode_0;
    v3_entry_points[SIM_ENTRY_INDEX_subc_i_1_mode_1] = &SIM_ENTRY_POINT_subc_i_1_mode_1;
    return;

    /* --- Service Routines --- */

    SIM_ENTRY_POINT_add_i_0_rsi_rdi:
    {
        u_long
        rs2, rd;
        EXTRACT_INTERMEDIATE_A(rs2, mOP);
        EXTRACT_INTERMEDIATE_A(rd, mOP);
        REG(rd) = REG(rd) + REG(rs2);
        epilogue();
    }

    SIM_ENTRY_POINT_add_i_0:
    {
        u_long
        rs2, rd;
        EXTRACT_INTERMEDIATE_A(rs2, mOP);
        EXTRACT_INTERMEDIATE_A(rd, mOP);
        REG(rd) = REG(rd) + REG(rs2);
        epilogue();
    }

    SIM_ENTRY_POINT_add_i_1_rsi_rdi_simm13_1:
    {
        u_long
        rs2;
        EXTRACT_INTERMEDIATE_A(rs2, mOP);
        REG(rd) = REG(rs2) + simm13;
        epilogue();
    }

    SIM_ENTRY_POINT_add_i_1_rsi_rdi:
    {
        u_long
        rd;
        long
        simm3;
        EXTRACT_INTERMEDIATE_A(simm3, mOP);
        REG(rd) = REG(rd) + simm3;
        epilogue();
    }

    SIM_ENTRY_POINT_add_i_1:
    {
        u_long
        rs2, rd;
        long
        simm3;
        EXTRACT_INTERMEDIATE_A(simm3, mOP);
        REG(rd) = REG(rs2) + simm3;
        epilogue();
    }

    SIM_ENTRY_POINT_subc_i_0_mode_0:
    {
        u_long
        rs2, rd;
        EXTRACT_INTERMEDIATE_A(rs2, mOP);
    }
}
```

49
```
EXTRACT_INTERMEDIATE(rs1, rs1, rd, dop);

rCMP_VALUE_A = REG(rs1);

rCMP_VALUE_B = REG(rs2);
REG(rd) = rCMP_VALUE_A - rCMP_VALUE_B;
epilogue();
}

SIM_ENTRY_POINT_sub cc_i_mode_0:
{
    u_long rs1, rd;
    long simm13;
    EXTRACT_INTERMEDIATE(simm13, A, simm13, dop);
    EXTRACT_INTERMEDIATE(rs1, B, rs1, dop);
    rCMP_VALUE_A = REG(rs1);
    rCMP_VALUE_B = simm13;
    REG(rd) = rCMP_VALUE_A - rCMP_VALUE_B;
    epilogue();
}

SIM_ENTRY_POINT_sub cc_i_mode_1:
{
    u_long INDEX_1;
    switch(INDEX_1()){
        case 0: /* sub cc_i_mode_1 */
            switch_to_opt();
            break;
        case 1: /* sub cc_i_mode_1 */
            switch_to_opt();
            break;
    }
    epilogue();
}
```
References


