GLISS 1.1.6
Generator of Libraries for ISS
Reference Manual

March 2003
Authors : Marc Finet, Christine Rochange and Pascal Sainrat
# Contents

1 General Presentation of the Tool  ................................................................. 5
   1.0.1 Terminology  .................................................................................. 5
   1.0.2 What is GLISS?  ........................................................................... 5
   1.0.3 Approach used  ............................................................................ 5
   1.0.4 Aims of GLISS  ............................................................................ 6
   1.0.5 Interface with the simulator  ....................................................... 6
   1.0.6 Tools origin and enhancements  ................................................... 7

2 The Description Language of Instruction Sets ............................................. 8
   2.1 Base elements of Sim-nML .................................................................. 8
      2.1.1 Resources: registers, memory  ................................................. 8
      2.1.2 Addressing modes  ................................................................... 8
      2.1.3 The instructions  ...................................................................... 9
   2.2 Further on Sim-nML ........................................................................... 13
      2.2.1 Definitions and rules  ................................................................. 13
      2.2.2 Type of operators  .................................................................... 13
      2.2.3 Bit fields  .............................................................................. 14
      2.2.4 Control Structures  .................................................................. 14
      2.2.5 summary of the elements of the Sim-nML syntax ..................... 15
   2.3 About the program counter ................................................................. 15
   2.4 Definition of macros  ....................................................................... 16
   2.5 Including description files ................................................................. 16
   2.6 Interface with floating-point numbers (FPI) ..................................... 16
      2.6.1 Rounding modes  .................................................................... 17
      2.6.2 Exceptions  ............................................................................ 17
   2.7 Constants  ....................................................................................... 18
   2.8 Bit order  ....................................................................................... 18
   2.9 Byte order  ..................................................................................... 18
   2.10 Endianness and aliases  .................................................................. 19

3 The iss Library Generator ............................................................................ 20
   3.1 Installing and executing GLISS ....................................................... 20
      3.1.1 Installation  ............................................................................. 20
      3.1.2 Compilation  ............................................................................ 20
      3.1.3 Organization of files and directories of GLISS  ....................... 20
   3.2 Generating an instruction set library with GLISS .............................. 21
      3.2.1 General view  ......................................................................... 21
      3.2.2 Files and directories  .............................................................. 22
      3.2.3 Compilation of the library  ....................................................... 22
3.3 Interface with the simulator .................................................. 22
  3.3.1 The iss_init function .................................................. 23
  3.3.2 The function iss_fetch ............................................... 23
  3.3.3 The function iss_decode ............................................. 23
  3.3.4 The function iss_compute ........................................... 24
  3.3.5 The function iss_complete ......................................... 24
  3.3.6 The function iss_free ............................................... 24
  3.3.7 The function iss_halt .............................................. 25
  3.3.8 The function iss_disasm ........................................... 25
  3.3.9 The function iss_error ............................................. 25
3.4 External functions (C functions) ......................................... 25
  3.4.1 Memory access interface .......................................... 26
  3.4.2 External modules .................................................... 28
3.5 Dealing with exceptions .................................................... 30
3.6 Information tables .......................................................... 30
  3.6.1 Definition ............................................................ 30
  3.6.2 Additional fields .................................................... 30
  3.6.3 Access to additional fields ....................................... 31

4 Parameters of GLISS ........................................................... 33
  4.1 Parameters related to the processor description ..................... 33
    4.1.1 -i : name of the intermediate file ................................ 33
    4.1.2 -p : name of the next instruction pointer ....................... 33
    4.1.3 -P : name of the current instruction pointer ................... 33
    4.1.4 -m : name of the memory ......................................... 33
  4.2 Options related to files and directories ................................ 33
    4.2.1 -e : name of external modules having initialization functions 33
    4.2.2 -E : name of an external module without initialization function 34
    4.2.3 -o : name of the output directory ................................ 34
    4.2.4 -I : name of the directory containing the header files ........ 34
    4.2.5 -D : dump function name .......................................... 34
    4.2.6 -t : function names of protection/translation memory addresses 34
    4.2.7 -a : name of additional fields .................................... 35
  4.3 Various options ............................................................ 35
    4.3.1 -h/-help : prints the online help ................................ 35
    4.3.2 -v : prints the version ............................................ 35
    4.3.3 -d : debug level .................................................. 35
    4.3.4 -w : warnings masks ............................................. 35

5 Evolutions ................................................................. 37
  5.1 GLISS version 1.1.6 ...................................................... 37
  5.2 GLISS version 1.1.5 ...................................................... 37
    5.2.1 Memory ............................................................ 37
    5.2.2 Portability ........................................................ 38
    5.2.3 Interfaces ........................................................ 38

Bibliography ............................................................... 41

Annex : Sim-nML Grammar .................................................. 42
Foreword

This document describes the tool entitled GLISS which stands for “Generator of Libraries for Instruction Set Simulators”. Its former name was GEP, that’s why you may find this name in older documentation and in the source of the tools. GLISS is a toolchain and gep is one of the tools. We use sometimes, in this document and in the tool, emulator instead of instruction-set-simulator or more exactly instead of the library of functions, types and objects intended to be used by an instruction-set-simulator or any other simulator. This manual describes shortly the structure of GLISS and mainly its features and its use. After a general description of the tool, we present in chapter 2 the nMP formalism used to model instruction sets, which is largely based on Sim-nML. The features of the generator and its use are then described in the last part.

In this document, bytes are represented with the least significant bit on the right and the most significant bit on the left. Thus, number 71 or 0x47 in hexadecimal are represented by the binary word (0b)01000111.

In this manual, various policies are used to ease the comprehension. Their signification is given below:

- function
- program or tool
- file
- constant
- variable
- code
- nMP/Sim-nML syntax element
Chapter 1

General Presentation of the Tool

1.0.1 Terminology

architecture In this document, we call architecture or processor architecture the logical view of the processor, the one the programmer needs to know, that is, essentially, the view offered by the instruction set. The hardware details of the processor are called microarchitecture.

emulator or iss we call emulator or instruction-set simulator or iss a program which simulate the architecture of the processor and not the microarchitecture. An iss does not deal with time, the notion of cycle or nanosecond is not known; this why it is also called sometimes functional simulator.

logical state The logical state of an iss is made of the registers which are in the programmer's view (those that can be read or written by an instruction) and the memory. There can be several states.

instruction Generally, the word instruction indicates either

- an instruction from the program (also called a static instruction) i.e. an instruction that might be executed several times and with various values. In this document, we use instruction in this case.
- an instruction which is executed by the processor (also sometimes called a dynamical instruction). Such an instruction is executed only once by the processor. In this document, we call it the instance of an instruction.

host machine This term denotes the machine on which the simulator will be executed.

1.0.2 What is GLISS?

GLISS is a set of tools (nmp2nml.pl, gep, irg) which, when used sequentially, generates from the description of an instruction set and form some other information, a library which can be used to realize an instruction-set simulator.

1.0.3 Approach used

There are two kinds of simulator.

The fastest is generated by replacing each instruction of a program by its description in a high-level language (or directly in the assembler of the host machine). One compiled, the executable is a functional simulator of this and only this program. For each program to verify, it will be necessary to generate the corresponding iss. It is very difficult to obtain performance results
from such a simulator for a complex program. This kind of simulator is mainly for verifying the functionality of programs or studies on instruction sets.

The second kind of iss is slower. The iss is an instruction interpreter. It decodes instructions and makes the corresponding functionalities. It is compiled once and is able to execute several programs without recompiling. Recompiling is needed only if the instruction set changes. It reads programs compiled for the target instruction set. It might be coupled to a micro-architectural simulator (the iss is driven by the simulator) which will give performance results.

GLISS generates a library for the second kind of simulator.

1.0.4 Aims of GLISS

GLISS has been written to simplify the writing of a simulator. Describing the instruction set in a hierarchical formalism allows us to avoid to describe completely each instruction by factorizing the description of common patterns between several instructions.

Used by a very simple simulator, the library generated by GLISS can be used to verify the functionality of an executable.

Coupled to a micro-architectural simulator, the library generated by GLISS can be used to verify the functionality of the simulator, measure performance results, explore micro-architectural solutions, . . .

Modeling a new processor is made according to figure 1.1. From the programmer reference manual, the instruction and the logical resources are described and the library is automatically produced by GLISS. The model describes only the instruction set and the logical resources. However, in order to execute a program, other information is needed. They are described in external modules using the C language.

The micro-architectural simulator, written in systemC for example, is described according to the reference manual of the processor.

![Diagram of Micro-architectural/Instruction Set Simulators](image)

Figure 1.1: Micro-architectural/instruction set simulators

1.0.5 Interface with the simulator

GLISS generates a library which contains type definitions and functions. The instruction-set simulator itself should initialize a logical state (registers and memory) by, among other things, loading the program which must be executed on the simulator. Then the simulator should fetch,
decode, execute and complete the instructions by calling the corresponding functions of the library. In the case of a speculative processor, the simulator can create several states, one being the reference state i.e. the state corresponding to the execution of instructions sequentially and not speculatively.

1.0.6 Tools origin and enhancements

Most of the formalism used to describe instruction sets, internal formalisms and also some parts of the code implementing the tools come from the Sim-nML tool developed at the Indian Institute of Kanpur. The description has been extended to add some new functionalities and ease the description of an instruction set. A description using these enhancements can’t be used with Kanpur tools.

The Sim-nML grammar (see appendix page 42) is respected except a few restrictions due to one of the tools: i.e. These restrictions are:

- A concatenation can’t be followed by a bit selection: \((A:B)\langle x..y\rangle\).
- An alias can’t span two bytes.
- The result of the concatenation should be a multiple of 8 bits.
- The switch structure can be used only in GLISS versions \(> 1.1.3\).
Chapter 2

The Description Language of Instruction Sets

2.1 Base elements of Sim-nML

To start with the Sim-nML language, we show how a very simple processor can be modeled. Its definition is the following:

- R : A bank of 4 16-bit registers.
- PC : a 16-bit program counter
- M : a memory of \(2^{16}\) words (which can be indexed as a register)

2 addressing modes :

- register \(R_i\)
- indexed \((R_i)\)

- addition between 2 registers, result in a register
- subtraction between 2 registers, result in a register

4 instructions :

- branch relative, with an immediate value
- move between registers or indexed and register or indexed (4 possible cases)

2.1.1 Resources: registers, memory

Figure 2.1 on the next page shows how we define the registers and the memory. The keyword reg introduces the definition of a register or a register bank if the first parameter is different than 1. The second parameter is the type of the register which defines its size. The keyword mem introduces the definition of a memory. In this last case, the memory is characterized by the number of bits needed to address it and the size of an element.

2.1.2 Addressing modes

Addressing modes can be defined in nMP. Then, these addressing modes can be used when describing the instructions. The definition of addressing modes (cf. figure 2.2 on the following page) is based on the resources declared earlier. An addressing mode is introduced by the keyword mode and is given some attributes: syntax describes the format of the expression of the addressing mode in that assembly language and image describes the binary representation of the addressing mode in the executable code. The format of the attributes are inspired from I/O formats in C (for example, register \(R[1]\) will be designated by \(R1\) in the assemble language, and
// definitions of constants used in the description
let REGS=2
let MEM_SIZE=16

// definitions of types used in the description
type index = card( REGS )
type word = card( 16 )

// resources declaration
reg R[2*REGS,word]
reg PC[1,word]
mem M[MEM_ADDR_SIZE,word]

Figure 2.1: Registers and memory declaration

by 001 in the binary code). Concerning the assembly language, the Motorola syntax is adopted in this example: R1 is the value which is in register R1 while (R1) is the contents of the memory addressed by the value which is in R1.

Note that we can define groups of addressing modes in order to simplify the descriptions of instructions (example: move_mode is either mem_index or reg_move).

<table>
<thead>
<tr>
<th>mode move_mode = mem_index</th>
<th>reg_move</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode reg_move(i:index)=R[i]</td>
<td>syntax =format(&quot;R%d&quot;,i)</td>
</tr>
<tr>
<td></td>
<td>image =format(&quot;0%2b&quot;,i)</td>
</tr>
<tr>
<td>mode mem_index(i:index)=M[R[i]]</td>
<td>syntax =format(&quot;(R%d)&quot;,i)</td>
</tr>
<tr>
<td></td>
<td>image =format(&quot;1%2b&quot;,i)</td>
</tr>
<tr>
<td>mode register(i:index)=R[i]</td>
<td>syntax =format(&quot;R%d&quot;,i)</td>
</tr>
<tr>
<td></td>
<td>image =format(&quot;%2b&quot;,i)</td>
</tr>
</tbody>
</table>

Figure 2.2: Definition of addressing modes

Note: an image or a syntax has no parameter, the keyword format is omitted: the constant string is enough. Example: image ="01".

2.1.3 The instructions

Figure 2.3 on the next page shows how the instructions can be described. The definition of an instruction is introduced by the keyword op followed by an identifier and the parameters of the instruction. Then some attributes are given to this instruction: syntax et image have the same role as for addressing modes; action describes the actions to realize when executing the instruction. For example, for the instruction move_op, the action describes the affectation of the source operand to the source destination. Usually, the information concerning these three attributes are clearly indicated in the programming manual of a processor.
In Sim-nML it is possible to factorize the description of the semantics of an instruction (action), of the syntax (syntax) and of the binary code (image). Thus, groups of instructions can be defined, as for example the group alu_op which is composed of arithmetic and logic instructions, all having a binary code beginning by 0.

Note the presence of an operation noted “instruction” which is the root of all instructions. All instructions in the description must always be attached to this “generic instruction”. In the example of figure 2.3 (its tree is represented in figure 2.5 on page 12), this link is made by the definition of types type_inst and type_alu.

Note: In our example, any instruction increments the PC nut we will see, afterwards that describing this action is not useful since incrementing the PC is taken in charge by the library generated by our tool.

```
op instruction(t:type_inst)
    syntax = t.syntax
    image = t.image
    action = { PC = PC+1; }

op type_inst = move_op | alu_op | branch_op

op move_op(s:move_mode,d:move_mode)
    syntax = format("move %s,%s",s.syntax,d.syntax)
    image = format("10%ss",s.image,d.image)
    action = { d = s; }

op alu_op(t:type_alu)
    syntax = format("%s",t.syntax)
    image = format("0%ss",t.image)
    action = { t.action; }

op type_alu = add | sub

op add(s1:register,s2:register,d:register)
    syntax = format("add %s,%s,%s",s1.syntax,s2.syntax,d.syntax)
    image = format("1%ss%ss",s1.image,s2.image,d.image)
    action = { d=s1+s2; }

op sub(s1:register,s2:register,d:register)
    syntax = format("sub %s,%s,%s",s1.syntax,s2.syntax,d.syntax)
    image = format("0%ss%ss",s1.image,s2.image,d.image)
    action = { d=s1-s2; }

op branch_op(t:card(6))
    syntax = format("jmp $+\%d",t)
    image = format("11%6b",t)
    action = { PC = PC+t-1; }
```

Figure 2.3: Definition of the instructions

Note: Sometimes, there are several possible definition for an instruction. For example, only one instruction could have been defined for the addition and the subtraction as indicated in figure 2.4 on the following page. Both definitions are possible and afford the same result once it is converted by our tool.
```c
op alu_op(which:card(1),s1:register,s2:register,d:register)
    syntax = format("alu%d,%s,%s,%s",which,s1.syntax,s2.syntax,
        d.syntax)
    image = format("01%b%2b%2b%2b",which,s1.image,s2.image,
        d.image)
    action =
        if (which==1) then
            d = s1 + s2;
        else
            d = s1 - s2;
        endif;
```

Figure 2.4: Another definition

It is better to be as near as possible as the definitions given in the processor manual but both
definitions will give a correct result. This kind of choice happens when there are instructions
with several options. We can define it as a complex instruction with variants or as several simpler
instructions.

The instructions built from the description in figure 2.3 on the page before are the following:

- **add**
  - syntax="add R%d,R%d,R%d",s1,s2,d
  - image="01%b%2b%2b",s1,s2,d
  - action={ PC = PC + 1; d = s1 + s2; }

- **sub**
  - syntax="sub R%d,R%d,R%d",s1,s2,d
  - image="00%b%2b%2b",s1,s2,d
  - action={ PC = PC + 1; d = s1 - s2; }

- **branch**
  - syntax="jmp $+%d",t
  - image="11%6b",d
  - action={ PC = PC + 1; PC = PC + t -1; }

- **move1**
  - syntax="move R%d,R%d",s,d
  - image="101%2b1%2b",d
  - action={ PC = PC + 1; R[d]=R[s]; }

- **move2**
  - syntax="move R%d,(R%d)",s,d
  - image="101%2b0%2b",d
  - action={ PC = PC + 1; R[d]=M[R[s]]; }
Figure 2.5: The tree corresponding to the definition of the instructions
2.2 Further on Sim-nML

2.2.1 Definitions and rules

We saw that the description of an instruction set is a set of definitions (constant, types, resources) and elements of language (addressing modes, instructions, ...). Classically, one starts by expressing the definitions then the elements of the language. The order between the elements of the language is not imposed while the definitions of constants, types or resources must be expressed before their use.

A rule init might be defined in order to describe the actions to make during the initialization of the simulator. Essentially, it consists in initializing registers.

The action attribute of an instruction can contain calls to external functions (written in C). The name of the function must be written between " characters and the external module must be included during the compilation of the library (see paragraph 3.4.2 on page 28). For example:

```c
action={
    if (i==19) then
        "print" ("Error, overflow !\n");
    endif;
}
```

Finally, one can define temporary variables which are necessary to describe complex actions. These variables must be declared as registers but their name should start with TMP in order to distinguish them from the registers of the processor. For example:

```c
reg TMP64 [ 1 , int ( 64 ) ]
```

2.2.2 Type of operators

When describing instructions, care must be taken to correctly specify operands. Indeed, base operators (addition '+', multiplication '*' ...) are implemented using C operators. These operators take into account the type of the operands in entry before that of the result. Thus, if one wishes to carry out a calculation on 64 bits starting from operands 32 bits, it is necessary to start by converting the operands from 32 to 64 bits (if not, the operator C will make calculation on 32 bits and the result will be then converted on 64 bits).

The cast operation is made using the unary operator coerce. For example, to make a calculation on 64 bits with operands declared as 32-bit operands, we must write:

```c
    TMP64 = coerce(int(64),TMP32) * coerce(int(64),TMP32);
```
2.2.3 Bit fields

To access to fields of bits, functions 'range' (<a..b>) are provided. See also section Bit order page 18. For example, to swap two 16-bit words, we must write:

```
TMP16 = TMP32<32..16>;
TMP32<32..16> = TMP32<15..0>;
TMP32<15..0> = TMP16;
```

2.2.4 Control Structures

There are no loop structure like while and for in Sim-nML. However, it is possible to build such loops with the base elements of Sim-nML. For example, one can make a for structure like this:

```
action = { i = 0;
  loop; }
loop = { if (i<4) then
     R[i]=0;
     i=i+1;
     loop;
   endif; }
```
2.2.5 summary of the elements of the Sim-nML syntax

<table>
<thead>
<tr>
<th>keyword</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>let</td>
<td>constant (of different types: string, binary number, hexadecimal...)</td>
</tr>
<tr>
<td>let x=3</td>
<td></td>
</tr>
<tr>
<td>let err=&quot;iss error&quot;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>definition of types (boolean, integer ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>type byte = card(8) // 8 bits unsigned</td>
<td></td>
</tr>
<tr>
<td>type word = int(32) // 32 bits signed</td>
<td></td>
</tr>
<tr>
<td>type float32 = float(9,23) // 32 bits floating-point</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mem/reg</th>
<th>definition of memory/registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg AX[1,card(16)]</td>
<td></td>
</tr>
<tr>
<td>reg R[12,int(32)]</td>
<td></td>
</tr>
<tr>
<td>mem M[16,byte]</td>
<td></td>
</tr>
<tr>
<td>reg AL[1,card(8)]</td>
<td></td>
</tr>
<tr>
<td>alias = AX[0]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mode</th>
<th>addressing modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode reg_ind_zero(r:index)</td>
<td></td>
</tr>
<tr>
<td>if r == 0 then</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>R[r]</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>syntax = format(&quot;r%d&quot;,r)</td>
<td></td>
</tr>
<tr>
<td>image = format(&quot;0%0d&quot;,r)</td>
<td></td>
</tr>
<tr>
<td>mode reg_or_imm = reg_ind_zero</td>
<td>imm</td>
</tr>
<tr>
<td>mode imm(i : immediat) = i</td>
<td></td>
</tr>
<tr>
<td>image = format(&quot;1%d&quot;,i)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>op instr_int = instr_add</td>
<td>instr_sub //'or' rule</td>
</tr>
<tr>
<td>op instr_sub(rs: reg_ind_zero, rd : reg_or_imm) //'and' rule</td>
<td></td>
</tr>
<tr>
<td>image = format(&quot;add %s %s &quot;,rd,rs)</td>
<td></td>
</tr>
<tr>
<td>syntax = format(&quot;00%0s&quot;,reg_ind_zero,reg_or_imm)</td>
<td></td>
</tr>
</tbody>
</table>

| >> | right and left shifts |
| ---------------------------------- |
| TMP_BYTE = TMP_BYTE >> 7; // keep only one bit | |

| >>> | right and left rotations |
| ---------------------------------- |
| TMP_BYTE = TMP_BYTE >>> 1; // 1-bit right rotation | |

| <> | field of bits |
| ---------------------------------- |
| TMP_BYTE = TMP_BYTE2<5,.0>; //keep the six lowest bits | |
| TMP_BYTE2<5,.0> = TMP_BYTE | |

| coerce | cast |
| ---------------------------------- |
| TMP64 = coerce(INT64, TMP32); | |

2.3 About the program counter

The example given previously corresponded to a very simple processor. In the current processors, one needs two instruction counters: one points on the instruction in progress (the CIA), the other
on the next instruction (NIA). Thus, a branch instruction uses the first pointer to calculate the second. Incrementing of the PC of the next instruction (NIA) is carried out automatically when fetching an instruction according to the size of the instruction. So it should not be specified in the description of the instructions.

2.4 Definition of macros

To avoid recopying portions of code, one can define macros. The definitions of macros start with the keyword macro, possibly followed by a parameter list between brackets. The character ’=’ precedes the body (the definition) of the macro. The syntax of macros is the same as the syntax of macros in C, and thus long lines can be cut out in several lines thanks to the character ’\’ (backslash), placed at the end of the line (not followed by spaces). The comments are authorized but only if they are alone on a line. The macro is then implicitly continued on the following line (even if the comment does not end with a ’\’). Also, a macro cannot be followed of a comment.

Example:

```c
macro FP_DIV = FPCR<22..22>
macro isabnormal(x) = (x>=0) & & \n    // check if normal underflow
    (x<2) \n    // or overflow
    || (x>>5)
```

2.5 Including description files

For a better organization of the description of an instruction set, it can be divided in various parts (and thus several nMP files). To include these files in the main file of the description (the file which contain the root instruction), one should write:

```c
include "nom_fichier.nmp"
```

By default, the file will be included at the place where the include command is. It is possible to define an included file in the beginning of a file even when the description must be included at the end (because it uses resources defined in the file where it will be included). The keyword op must precede the include command.

```c
include op "nom_fichier.nmp"
```

Note: the included file must be an nMP file. Several levels of inclusion is possible but there are no re-entrance test. It’s up to you to avoid to include a first file which include a second file which include the first file!

2.6 Interface with floating-point numbers (FPI)

An IEEE 754 interface is provided between the nMP description and the floating-point unit of the host machine. It includes the rounding modes and the exceptions. The FPI takes, at the level of the nMP description, the form of constants and functions.
2.6.1 Rounding modes

Rounding modes of the host machine are specified using some constants:

<table>
<thead>
<tr>
<th>FPI_TONEAREST</th>
<th>rounding to the nearest</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPI_UPWARD</td>
<td>rounding upward</td>
</tr>
<tr>
<td>FPI_DOWNWARD</td>
<td>rounding downward</td>
</tr>
<tr>
<td>FPI_TOWARDZERO</td>
<td>rounding towards zero</td>
</tr>
</tbody>
</table>

In order to access to the current rounding mode of the host machine and to modify it, two functions are provided:

- `fpi_setround(mode)` modify the rounding mode
- `fpi_getround()` returns the rounding mode

2.6.2 Exceptions

Six constants define the IEEE floating-point exceptions:

<table>
<thead>
<tr>
<th>FPI_INEXACT</th>
<th>inexact result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPI_DIVBYZERO</td>
<td>division by zero</td>
</tr>
<tr>
<td>FPI_UNDERFLOW</td>
<td>number too small to be represented</td>
</tr>
<tr>
<td>FPI_OVERFLOW</td>
<td>number too high to be represented</td>
</tr>
<tr>
<td>FPI_INVALID</td>
<td>invalid operand</td>
</tr>
<tr>
<td>FPI_ALLEXCEPT</td>
<td>all the above exceptions</td>
</tr>
</tbody>
</table>

**Note:** To combine several exceptions the binary 'or' operator ('|') can be used. Constant `FPI_ALLEXCEPT` is the binary 'or' of all exceptions.

To set exceptions, three functions are provided:

- `fpi_clearexcept(exceptions)` clear all exceptions
- `fpi_raiseexcept(exceptions)` sets all exceptions
- `fpi_testexcept(exceptions)` returns the exceptions which are set

Of course, exception flags are set only by floating-point operations.

Example of use of the FPI interface:

```
// clear all flags
fpi_clearexcept(FPI_ALLEXCEPT);
// compute
FPR[0]=FPR[0]/FPR[1];
// check result
if (fpi_testexcept(FPI_DIVBYZERO)) then
   // div by zero happened
   exit;
endif;
```
2.7 Constants

Constant definitions(let x=...) can’t have a size higher than 32 bits because they are implemented as 32-bit integers. To use larger constants, macros should be used because they are kept as a textual form.

Example:

LET x=23 // no problem
LET x64=0xfffffffffffffff // possible error if the host machine is a 32-bit one
MACRO x64=0xfffffffffffffff // ok!

2.8 Bit order

In a nMP description, fields of bits can be accessed using (VAR<a..b>). The result is an unsigned integer. To set a field of bits, the value which must be written should be an unsigned integer.

The nMP variable called bit_order sets the direction of the interval. This variable has been introduced in order to let the user choose the direction he likes. If bit_order is equal to uppermost, the first value (a) corresponds to the bit having the most significant and the second (b) to the least significant. If bit_order is equal to lowermost, it is the contrary.

Example:

TMP = 0x3A; // 0b00111010
   // bit0 : 0, bit1: 1, bit2 : 0, bit3: 1,
   // bit4 : 1, bit5 : 1, bit6 : 0, bit7: 0
   // uppermost
TMP<5..2>; //=15 (0b1110)
   // lowermost
TMP<2..5>; //=15 (0b1110)

   Usually, in the uppermost mode, the first operand is higher than the second one and reciprocally in lowermost mode.
   Example:

TMP = 0x3A; // 0b00111010
   // uppermost
TMP<2..5>; //=7 (0b0111)
   // lowermost
TMP<5..2>; //=7 (0b0111)

   The only case where the user must specify the bits in opposite order is the case of an inversion of the order of the bits of a variable (mirror).
   TMP<7..0>=TMP<0..7>; // mirroring:

   If the bit_order is not set, GLISS assumes a lowermost mode.

2.9 Byte order

For each memory defined in a description, a variable should be defined to indicate if the memory is in big or little endian mode. The name of this variable is <memory_name>_is_little. One can assign to this variable either a constant (1 or 0) or a string representing C code. This code
can use any symbol used in the nMP description except macros, that is any register or constant. When specifying C code, you can’t use operators specific to nMP such as bit fields operators or shifts. If such operations are needed, one can use the functions in the file `operators.h` in the `gliss/templates` directory.

This variable defines the access mode to the memory in the nMP description. It also parameterizes memory accesses in external modules. See section 3.4.1 page 26.

Examples:

```c
// Endianness of memories M and N
let M_is_little = 1     // M is always little
let N_is_little = "STAT & 0x020" // little endian if the 6th bit of STAT register is set
```

**Note:** In fact, this variable is not useful in the nMP description since, up to now, only bytes are accessed in an nMP description.

## 2.10 Endianness and aliases

Variables longer than one byte can be stored in memory in two ways: least significant first (little endian), or most significant first (big endian).

```
<table>
<thead>
<tr>
<th>Memory</th>
<th>Little Endian</th>
<th>Big Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>msh</td>
<td>lab</td>
</tr>
<tr>
<td>Little Endian</td>
<td>lab</td>
<td>msh</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Figure 2.6: Storing data in memory: little endian/big endian

The place of the least and most significant bytes is a problem because the host machine has an endian (x86, little endian, and SPARC, big endian) and processors which must be simulated has also its own endian (little or big endian).

To simplify the problem in GLISS, the simulator must be generated on the host machine, that is the machine on which it will be compiled and executed. Thus, when generating the simulator, GLISS checks the endian of the host machine. Furthermore, the simulator should be executed on the same machine.

To define properly access types to variable parts, one can use Sim-nML aliases as in the example (cf. figure 2.7). A 16-bit register `ax` and 2 8-bit registers are defined: `al` which corresponds to the least significant byte `ax` (it starts with bit 0), and `ah` which corresponds to the most significant byte `ax` (starting from bit 8). This raises the ambiguity of accesses to the parts of `ax`.

```
reg ax[1,card(16)]
reg al[1,card(8)] // LSB of ax
    alias = ax[0]
reg ah[1,card(8)] // MSB of ax
    alias = ax[8]
```

Figure 2.7: Defining an alias in Sim-nML
Chapter 3

The iss Library Generator

Producing the simulator library of a new instruction set is made in four phases:

- writing of the nMP description of the instruction set and of the external modules,
- installation and compilation of the generator (if not yet done for another processor),
- generation and compilation of the library,
- writing and compilation of the simulator which will use the functions of the library.

3.1 Installing and executing GLISS

3.1.1 Installation

GLISS is given as a tgz archive. Extracting the files is done in the following way:

- under Linux: `tar -zxvf gliss.tgz`
- under Solaris: `gunzip --stdout gliss.tgz | tar xvf -`

The directory `gliss` is created and contains all the extracted files and directories. In particular, a `Makefile` is extracted.

3.1.2 Compilation

As explicated in the paragraph 2.10, the generation of a library must be done on the machine on which it will be executed. GLISS must then be compiled on this machine. Supported machines are x86/linux and SPARC/Solaris.

All the GLISS tool-chain can be compiled by just typing `make` in the directory `gliss`. Some variable names can be given as parameters. These are defined in the file `src/gep/gep_common.h`. These variables may be useful to define particular types common to the library and the simulator.

3.1.3 Organization of files and directories of GLISS

Only executables and directories are in the root directory `gliss`.

```
gliss  --------+-- src --------+-- gep  --------+-- gep.c  
          |            +-- gep_common.h (gep source)
          |            +-- ... (gep sources)
          |            +-- iraction.c (gep source)
```
3.2 Generating an instruction set library with GLISS

3.2.1 General view

As shown in figure 3.1, GLISS builds from the instruction set description in an intermediate format a set of files containing the code of the functions of the library. These files are the compiled using gcc which generates the functions library (.a) and the corresponding header files (.h). These header files must be included in the simulator code using the include directive of the C language to be able to access to the functions of the library. The list of all the header files is itself in the file iss extern.h. Thus, only this file has to be included in the simulator. An example of instruction set description (the PowerPC) is provided on our web site and it also contains a minimal simulator.

![Diagram of the library generation process]

Figure 3.1: Organization of iss library generator
3.2.2 Files and directories

Files which are generated

The following table shows the files generated when creating an instruction-set library.

<table>
<thead>
<tr>
<th>file</th>
<th>program</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppcm.mmp</td>
<td>nmp2nml.pl</td>
<td>nMP description</td>
</tr>
<tr>
<td>ppcm.nml</td>
<td></td>
<td>sim-nML description</td>
</tr>
<tr>
<td>ppcm.ir</td>
<td>irg</td>
<td>intermediate representation generator</td>
</tr>
<tr>
<td></td>
<td>gep</td>
<td>intermediate format (a set of tables)</td>
</tr>
<tr>
<td>iss_code.c</td>
<td></td>
<td>library source</td>
</tr>
<tr>
<td>...</td>
<td>gcc</td>
<td>c compiler, exists on most *nix</td>
</tr>
<tr>
<td>iss_types.c</td>
<td></td>
<td>library source</td>
</tr>
<tr>
<td></td>
<td>gcc</td>
<td>functions library</td>
</tr>
<tr>
<td>emul.a</td>
<td></td>
<td>header files</td>
</tr>
<tr>
<td>emul.h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Organization of files and directories

When creating an instruction-set library, gliss creates a directory for this library. the name of this directory can be specified using an option as mentioned in section 4). In this directory, gliss creates three subdirectories (src, lib and include) which contains respectively the sources of the library, the functions library (compiled from the source files) and the corresponding headers. If older files exist, they are erased.

```
emul-directory --+-src---------+-- iss_code.c (emul source)  
|                  +-- iss_decode.c (emul source)   
|                  +-- ...                        
|                  +-- iss_types.c (emul source)   
|                  +--lib---------+-- emul.a  (library)  
|                  +--include-----+-- emul.h (include)  
|                  +-- ...                        
|                  +-- iss_types.h (include)  
+ Makefile
```

3.2.3 Compilation of the library

In order to easily compile the library, gliss generates a Makefile. It is highly recommended not to modify the compilation flags for the iss_code.c file. Especially, the -fno-strict-aliasing flag must be absolutely used with optimization levels 2 or 3 (-O2, -O3) because the library contains structure pointers. Without this flag, the code generated by gcc will not be correct. See the PowerPC simulator manual [1] to obtain more details on how to compile the simulator.

3.3 Interface with the simulator

The interface with the simulator is made of the following functions:
• iss_fetch
• iss_decode
• iss_compute
• iss_complete
• iss_init (to initialize the state of the simulator)
• iss_free (to deallocate a decoded instruction)
• iss_dump (to show the values of the registers)
• iss_halt (to stop the simulation and frees the memory)
• iss_disasm (to disassemble an instruction)
• iss_error (to print a message and quit)

The prototypes of these functions are in the file emul.h which is in the include directory which is created when generating the library.

3.3.1 The iss_init function

This function initializes a logical state (registers and memory) and various external modules whose list is given when generating the library (cf paragraph 3.4.2). It returns a pointer towards the state which has been created.

3.3.2 The function iss_fetch

This function receives an address and returns the instruction code (a suite of bytes) stored at this address in the memory. Usually, this code is transmitted to the iss_decode function and can also be used to verify the memory hierarchy of the simulator when modeled.

3.3.3 The function iss_decode

This function receives an instruction code (a suite of bytes) and returns an instance of the corresponding instruction. Its type (instruction_t) is composed of several fields:

<table>
<thead>
<tr>
<th>field name</th>
<th>type</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>ident</td>
<td>enum</td>
<td>identifier of the instruction (number)</td>
</tr>
<tr>
<td>instrinput</td>
<td>ii_t *</td>
<td>instruction input interface</td>
</tr>
<tr>
<td>instroutput</td>
<td>ii_t *</td>
<td>instruction output interface</td>
</tr>
</tbody>
</table>

The field ident is unique for each type of instruction. It is made form the name of the instruction prefixed by ID_. For example, for the add instruction, the identifier is ID_ADD. The list of all identifiers is given in the emul.h file. The fields instrinput and instroutput are the instruction interface. They are element tables, each element being of type ii_t, which define the inputs and the outputs of an instruction. The type ii_t is composed of the following fields:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>type_param_t</td>
<td>type of the input/output (parameter or register)</td>
</tr>
<tr>
<td>val</td>
<td>value_t</td>
<td>value</td>
</tr>
</tbody>
</table>
The type field defines the type of the input/output. The type \texttt{param_t} type is an enum defined in the \texttt{iss\_include.h} file. If the input (output) is a parameter, that is a value coded in the instruction code, then the type is \texttt{PARAM\_T}. Otherwise it is the name of the register followed by \texttt{T}. For example (\texttt{GPR\_T} for register GPR). The type \texttt{VOID\_T} is the end of the table.

The \texttt{val} field gives, if it is a parameter, the value found when decoding the instruction. If it a register file then val contains the value of the index in this file to access to the register specified in the code of the instruction. For example, if an instruction reads GPR[5] then the element (type=\texttt{GPT\_T},val=5) will be in the instrinput table. If the value is -1, it means that the register number is dynamic (and thus could not be found when generating the emulator).

Example: for instruction add r2,r4,r3 (as defined in figure 2.3, page 10), the instrinput table will be (\texttt{PARAM\_T},2) (\texttt{PARAM\_T},4) (\texttt{PARAM\_T},3) (\texttt{R\_T},4) (\texttt{R\_T},3) (\texttt{VOID\_T},XXX), and the instroutput table: (\texttt{R\_T},2) (\texttt{VOID\_T},XXX).

### 3.3.4 The function \texttt{iss\_compute}

This function corresponds to the operation which must be realized by the instruction.

The role of the function \texttt{iss\_compute} is to avoid to write the code in the simulator. This function modifies the state which is transmitted to it. Thus, a simulator modeling a processor with speculative execution can transmit a speculative state different from the non speculative state.

Example of use:

```c
instr=iss\_decode(buffer);
if (instr->ident == ID\_ADD) {
    // for instr ID\_ADD, instr->instrinput[2].type == GPR\_T
    GPR(state)[instr->instrinput[2].val.uint8]=3;
    // or GPR\_PO(state)=3; // GPR\_PO is a macro defined by GLISS
    // for instr ID\_ADD, instr->instrinput[3].type == GPR\_T
    GPR(state)[instr->instrinput[3].val.uint8]=4;
    // or GPR\_P1(state)=4;
    iss\_compute(instr,state);
    printf("3+4=%d\n",GPR(state)[instr->instroutput[0].val.uint8]);
}
```

**Note:** in this example, one can use macros to easily access to the variables. These macros are defined in the \texttt{include\_iss\_include.h} file which is created when generating the library.

### 3.3.5 The function \texttt{iss\_complete}

As well as the \texttt{iss\_compute} function, this function receives an instruction and a state. It updates the state it receives (usually the non speculative state, i.e. the one provided by \texttt{iss\_init}). Contrarily to the previous function, this function updates the memory and, if external functions are called they receive in their first parameter \texttt{COMPLETE} and not \texttt{COMPUTE}. Thus, one can define external functions with different outcomes according to the speculative state of the instruction.

### 3.3.6 The function \texttt{iss\_free}

This function receives an instruction and is in charge of freeing the space allocated to this instruction by the \texttt{iss\_decode} function. Each call to the decode function must then be followed by a call to \texttt{iss\_free}, otherwise the memory will quickly become saturated.
3.3.7 The function iss_halt

This function stops the simulator and frees the memory allocated during the initialization or the execution. This function also calls halt functions of external modules if needed. See the part on external modules in section 3.4.2 on page 28.

3.3.8 The function iss_disasm

This function receives an instruction and a buffer. It stores in the buffer (which must previously have been allocated) by the string resulting from disassembling the instruction. Disassembling is made using the syntax field in the nMP description. To use this function, the library must be compiled with the variable ISS_DISASM, this way:

make all OPT=-DEMUL_DISASM

3.3.9 The function iss_error

This function receives a string. It prints this string, indicates where the error happened and quits the simulator.

3.4 External functions (C functions)

A nMP description can contain calls to C functions which are described in external files i.e. files not generated by GLISS because nMP can’t model anything.

In a nMP description, the name of a function must be written between double quotes ("). External functions must be described in one of the external files in the extern directory.

The first argument of these functions must be: complete whose type is int. complete must have one of the two values: COMPUTE or COMPLETE, which tells if the function is called speculatively or not i.e. by function iss_compute or iss_complete. Thus according to the nature of the execution, a function can have two outcomes. However, it is not necessary to specify this argument in the nMP description.

An external function can access to registers thanks to an argument whose type is state_t and must be called state in order to access to these parameters using the macros defined in the file iss_macro.h because the name state is directly coded in the macro definition. To access to all macro and functions which might be necessary in an external function, the file issExtern.h and eventually the file operators.h must be included in the external file.

Accessing to the memory in an external file is described in section 3.4.1.

Example:

```c
<mp description>
op add(x : index, imm : card(32))
    action = { R[x] = R[x] + imm;
                "print"(R[x],imm,"state",x);
}

<external file>
#include "issExtern.h"

void print(int complete, int32_t r_x, uint32_t imm,
```
state_t *state, uint5_t x){
    if(!complete)
        printf("Be careful, speculative values\n");
    printf("R[x]=%08x, immediat =%08x\n",r_x,imm);
    printf("M(32bits)[0]=%08x\n", READ32_MEM_LITTLE_DATA(complete,M,mem->read_32(0));
    /* using state */
    printf("registre PC = %08x\n", NIA); //NIA is a macro coded as state->nia
    printf("R[x]=%08x (x=%d)\n", R[x.val], x.val); //R is also a macro
    return 0;
}

Note: Two types of macros exist to access to the registers. The macros used above and defined
in the file iss_macros.h are to be used in external functions in which state must absolutely be
defined.

In a simulator, one can used the macros defined in the file iss_include.h which contains a
lot of generic macros. These macros access to the registers of the state they receive as argument.
Usually, the src directory contains objects to be used in external functions while functions for
the simulator are in the directory include.

3.4.1 Memory access interface

Access functions

In a nMP/Sim-nML description, memory accesses are only one-byte wide. However, an interface
is provided to be used in C external functions. It provides reads and writes on 8, 16, 32, 64 bits
and via a buffer.

The interface consists in macros which access to the memory. This interface takes into account
translation and access rights checking when it exists. The following macros are provided:

- direct access:

  - READ8_MEM_LITTLE_DIRECT (int, memory_t *, address_t, instruction_t *)
  - READ16_MEM_LITTLE_DIRECT (int, memory_t *, address_t, instruction_t *)
  - READ32_MEM_LITTLE_DIRECT (int, memory_t *, address_t, instruction_t *)
  - READ64_MEM_LITTLE_DIRECT (int, memory_t *, address_t, instruction_t *)
  - READ8_MEM_BIG_DIRECT (int, memory_t *, address_t, instruction_t *)
  - READ16_MEM_BIG_DIRECT (int, memory_t *, address_t, instruction_t *)
  - READ32_MEM_BIG_DIRECT (int, memory_t *, address_t, instruction_t *)
  - READ64_MEM_BIG_DIRECT (int, memory_t *, address_t, instruction_t *)
  - READ_BUF_MEM_DIRECT (int, memory_t *, address_t, void *buf, int size, instruction_t *)

  - WRITE8_MEM_LITTLE_DIRECT (int, memory_t *, address_t, uint8_t, instruction_t *)
  - WRITE16_MEM_LITTLE_DIRECT (int, memory_t *, address_t, uint16_t, instruction_t *)
  - WRITE32_MEM_LITTLE_DIRECT (int, memory_t *, address_t, uint32_t, instruction_t *)
  - WRITE64_MEM_LITTLE_DIRECT (int, memory_t *, address_t, uint64_t, instruction_t *)
  - WRITE8_MEM_BIG_DIRECT (int, memory_t *, address_t, uint8_t, instruction_t *)
  - WRITE16_MEM_BIG_DIRECT (int, memory_t *, address_t, uint16_t, instruction_t *)
  - WRITE32_MEM_BIG_DIRECT (int, memory_t *, address_t, uint32_t, instruction_t *)
  - WRITE64_MEM_BIG_DIRECT (int, memory_t *, address_t, uint64_t, instruction_t *)
- `WRITE_BUF_MEM_DIRECT (int, memory_t *, address_t, void *buf, int size, instruction_t *)`

- code access:
  - `READ8_MEM_LITTLE_CODE (int, memory_t *, address_t, instruction_t *)`
  - `READ16_MEM_LITTLE_CODE (int, memory_t *, address_t, instruction_t *)`
  
  [... ]
  - `WRITE32_MEM_BIG_CODE (int, memory_t *, address_t, uint32_t, instruction_t *)`
  - `WRITE64_MEM_BIG_CODE (int, memory_t *, address_t, uint64_t, instruction_t *)`
  - `WRITE_BUF_MEM_CODE (int, memory_t *, address_t, void *buf, int size, instruction_t *)`

- data access:
  - `READ8_MEM_LITTLE_DATA (int, memory_t *, address_t, instruction_t *)`
  - `READ16_MEM_LITTLE_DATA (int, memory_t *, address_t, instruction_t *)`
  
  [... ]
  - `WRITE32_MEM_BIG_DATA (int, memory_t *, address_t, uint32_t, instruction_t *)`
  - `WRITE64_MEM_BIG_DATA (int, memory_t *, address_t, uint64_t, instruction_t *)`
  - `WRITE_BUF_MEM_DATA (int, memory_t *, address_t, void *buf, int size, instruction_t *)`

- data access (according to the endian given in argument):
  - `READ8_MEM_DATA (int, int, memory_t *, address_t, instruction_t *)`
  - `READ16_MEM_DATA (int, int, memory_t *, address_t, instruction_t *)`
  
  - `READ32_MEM_DATA (int, int, memory_t *, address_t, instruction_t *)`
  - `READ64_MEM_DATA (int, int, memory_t *, address_t, instruction_t *)`
  - `READ_BUF_MEM_DATA (int, int, memory_t *, address_t, void *buf, int size, instruction_t *)`
  - `WRITE8_MEM_DATA (int, int, memory_t *, address_t, uint8_t, instruction_t *)`
  - `WRITE16_MEM_DATA (int, int, memory_t *, address_t, uint16_t, instruction_t *)`
  
  - `WRITE32_MEM_DATA (int, int, memory_t *, address_t, uint32_t, instruction_t *)`
  - `WRITE64_MEM_DATA (int, int, memory_t *, address_t, uint64_t, instruction_t *)`
  - `WRITE_BUF_MEM_DATA (int, int, memory_t *, address_t, void *buf, int size, instruction_t *)`

The first argument, whose type is `int` corresponds to the variable `complete`, which is the first argument of external functions. The second argument, whose type is `memory_t *`, is the memory which will be accessed. The last argument, whose type is `instruction_t *`, must be a pointer towards the instruction which wants to access the memory. This might be useful when a mmu is modeled. An example of their use is given in the preceding section.

When accesses are parameterized by the endian, the second argument is the endian.

The file `iss_mem_interface.h` defines these functions and is reference in `issExtern.h` and `issInclude.h`.
Interfaces

As accesses to the memory can be regulated by some external functions (when modeling a memory management unit as in the PowerPC example), several modes exist: direct, code and data. In the external file describing the memory management unit, functions can be defined for the three modes: "code", "data" et "direct".

The "code" mode considers the memory as being instructions (binary code). In the "data" mode, accesses are supposed to access to data. Finally, in the "direct" mode, accesses are made directly, thus not through any external functions.

Defining functions for translating addresses or checking access rights

Translation functions must have the following prototype:
address_t trans_code(int complete, memory_t *mem, address_t a, instruction_t *i);

A pointer towards the instruction is passed to the functions in order to give to the MMU more dynamic and static information (see the definition of the instruction_t type, paragraph 3.3.3 on page 23, and the tables, section 3.6 on page 30). The complete argument can be used to check access rights or not.

A "direct" mode function should not modify the address but can be used to check access rights.

A "code" mode function will receive a null pointer instead of an instruction pointer as it is called by the iss_fetch function and not by the execution of an instruction.

The prefix of the name of the functions must be given on the GLISS command line. To each prefix, two functions, suffixed _read and _write must be defined. See option `-t', in the section parameters of gliss, section 4 on page 33.

3.4.2 External modules

External modules are made of external functions as described in the above section and of various modules which are needed to adapt the simulator to its environment (format of the executable file to be loaded, operating system of the host machine).

External functions are usually grouped in one or several .c files with their header files (.h) which contain the prototypes of the functions called by the nMP description.

Both files (.c and .h) are called an external module.

An external module can have an initialization function and a halt function, either both functions or none. If one is needed and not the other, this last function can be empty. When it exists, the name of the initialization function must be <module_name>_init (where <module_name> is the name of the module, that is <module_name>.c is the name of the file containing the functions.
The name of the halt function must be <module_name>_halt.

The function iss_init which is automatically generated by GLISS calls the initialization function of each external module which is specified using the `-e' option on the command line of GLISS. Halt functions are called by the function iss_halt for the same modules. To define a module which have no initialization and no halt function, the module should be declare on the command line of GLISS using the `-E' option instead of the `-e' command.

The simulator can pass arguments to initialization function of a module through arguments of the iss_init. These arguments must be list of pointers (void **). If several modules must be initialized and their initialization functions have arguments, the arguments must be given to the iss_init function in the order the modules are specified on the GLISS command line. The logical state of the processor (its registers) state_t *state is the first argument of the initialization functions so they can access to these registers. A pointer to a memory can also be
passed to an initialization function. The name of the memory must be specified after the name of
the external module followed by the ’+’ character on the command line of GLISS. Halt functions
have no arguments since they are provided for freeing the memory allocated by the initialization
function.

Example:
<exempl.nmp>
if (div_by_zero) then
   "treat_except"( 1 ); // external function, 1=div by zero
endif;

... [user@host]>gep -e except+M -e foo ... ...

< emul.c file> /* Generated by gliss */
iss_init(..., void *arg_except[], void *arg_foo[]){
    state_t *state; /* real state */
    init_mem(...);
    ...
    except_init(state, M, arg_except);
    foo_init(state, arg_foo);
    return;
}

void iss_halt(
    except_halt();
    decode_halt();
    ...
    return;
}

<except.c file>
memory_t *memory;

void except_init(state_t *state,memory_t *mem,void *arg[]){
    int i=0;
    memory=mem;
    while (arg[i]!="null"){
        if (i==0) /* init GPR[0] */
            GPR[0]=* (int *)arg[i];
        if (i==1) /* init M[0]=arg[i] */
            WRITE32_MEM_LITTLE DIRECT(COMplete,memory,* (uint32_t *)arg[i],0);
        i++;
    }
    return;
}

void iss_halt(void){
    /* nop : no allocation */
    return;
}
3.5 Dealing with exceptions

Exceptions due to instructions should be defined in the nMP description. Depending on the type of applications, it might call an external module or not.

If an exception is raised when calling iss_compute, it should be ignored since it might be a speculative exception.

If an exception is raised by the host machine (divide by zero, segmentation fault, etc), it is probably due to an error in the nMP description or in an external file.

Example:

Even if the processor which is modeled doesn’t raise an exception when a division by zero occurs, the test must be done in the nMP description and appropriate action should be taken in order to avoid that the host machine makes the division by zero and raises an exception.

// Bad description
op_divise_sans_exception(ra: registre, rb: registre) \naction = { ra = ra / rb;}
// Good description
op_divise_sans_exception(ra: registre, rb: registre) \naction = { if (rb!=0) ra = ra / rb;}

3.6 Information tables

3.6.1 Definition

GLISS generates tables containing static information on instructions. They describe the arguments of instructions (but not the values except for immediate mode) and their outputs. The name of the main table is iss_table and its type is defined in the file emul.h.

3.6.2 Additional fields

One can add fields in a nMP description in addition to the mandatory fields i.e. syntax, image et action. Thus, one can add specific information to each instruction in the description tables generated by GLISS.
For such a field to be taken into account by GLISS, a '-a' option should be added followed by the name of the field and its type on the GLISS command line.

The type can be a base one: string, float, [u]int[8]/16/32/64], a type defined by the user (between double quotes). The type might also be a table just adding '[' after the type name. For tables, the separator to use in the description might be specified between '[' and ']'. The default separator is ',. Finally, one can specify the size of the table between the characters '[' and ']' (after the separator if specified). Otherwise, the table will have different sizes according to each instruction.

If the type is a user type, the field in the description is considered as a C expression. This code is executed when created instructions, that is when calling the iss_init function. So, such a function must use only static information. A file might be included to add variables. The name of the file must be specified just after the name of the type (without any space) and rounded by double quotes. The directory where this file is must be included, so it must be specified using the '-I' option (see the following chapter on the arguments of GLISS).

Examples:
- a foo string -> adds the foo field whose type is string;
- a foo2 int8 -> adds the foo2 field whose type is int8;
- a foo3 int16[ ] -> adds the foo3 field whose type is a table of int16, elements being separated (in the nMP description) by spaces ' ';
- a foo4 string[/4] -> adds the foo4 field whose type is a table of 4 strings, elements being separated by the '/' character.
- a foo5 "function_t""my_fun.h" -I/home/gliss/inc -> adds the foo5 field whose type is function_t, and includes the <my_fun.h> file which is in the /home/gliss/inc directory.

In the nMP description, fields are added below the mandatory ones.
Example:

```c
op add(...)
  syntax="...
  image="add ...
  action={...

  foo = "addition"
  foo2 = "-3"
  foo3 = "1234 23 23 32"
  foo4 = "ex/am/pl/e"
  foo5 = "FnAdd | FnMul | FnInt" // defined in my_fun.h
```

**Note:** Note that even if a field has an int type, the value must be given between double quotes (see foo2).

### 3.6.3 Access to additional fields

As mentioned earlier, GLISS generates structure tables which describe the instructions. Each element of the iss_table corresponds to an instruction and each element has various fields including the additional fields.

Example: iss_table[instr->ident].foo

In the case of tables, the size of the table is defined in a field whose name is the name of the field followed by _size.

Example: iss_table[instr->ident].foo4_size
To afford a better performance, strings are managed as identifiers. To each string found in the nMP description, an identifier is added. The name of this identifier is the name of the additional field followed by the name of the variable. In the above example, at least 4 identifiers will be generated: foo4_ex, foo4_am, foo4_pl, foo4_e. Accessing to the 3rd string (“pl”) is made like this (iss_table_foo4[foo4_pl]).

**Note:** '['; ']'" and '!' must be protected by ' ' characters in the command line of GLISS to avoid the interpretation by the shell or by prefixing these characters with \ (anti-slash).
Chapter 4

Parameters of GLISS

4.1 Parameters related to the processor description

4.1.1 -i: name of the intermediate file

'i' specifies the name of the intermediate file used when generated the library.

Mnemonic: 'i' stands for intermediate.

4.1.2 -p: name of the next instruction pointer

It specifies the name of the next instruction pointer used in the nMP description.
Mnemonic: 'p' like pc (program counter).

4.1.3 -P: name of the current instruction pointer

It specifies the name of the current instruction pointer used in the nMP description.
Mnemonic: 'P' like pc (program counter). Not the same as 'p'.

4.1.4 -m: name of the memory

It specifies the name of the memory where instruction code will be loaded. If only one memory exists in the description, it might be omitted.
Mnemonic: 'm' like memory.

4.2 Options related to files and directories

4.2.1 -e: name of external modules having initialization functions

It specifies the name of an external module which should be included when compiling the library. Of course, several modules might be declared but each after a '-e' switch.

The order modules are specified should correspond to the order of the arguments of the iss_init function call. (see section 3.4.2 on page 28 for more details).
Mnemonic: 'e' like external.
4.2.2 -E : name of an external module without initialization function

It specifies the name of an external module which contains functions called by the simulator. Several modules can be declared but each after a `-E' switch.

Mnémonic: 'E' like external.

4.2.3 -o : name of the output directory

It specifies the directory where the sources, the library and the headers will be generated. See the section Organization of files and directories page 22. This option might be omitted. In this case, the name of the directory is the name contained in the variable proc defined in the nMP description (let proc = "ppc" for example). If this variable is not defined in the nMP description, the name of the directory is emul.

Mnémonic: 'o' like output.

4.2.4 -I: name of the directory containing the header files

It specifies the name of the directory where header files of external modules or header files defining types used by option '-a'. The first directory to declare must be the directory of external modules because it will be used to create the dependencies in the Makefile of the library.

Note: the name might be relative to the source files or, better, absolute.

Mnémonic: 'I' like include. Not the same as 'i'.

4.2.5 -D : dump function name

It specifies the name of the dump function which prints the values in the registers of the processor. This function is declared as:

```c
void dump(FILE *fd, state_t *state);
```

It must be defined in one of the external modules. If the option is omitted, a default function will be generated which prints the values of all the registers and linearly. Thus, it is preferable to define this function if it is needed.

Note: This function is called when the simulator tries to complete an illegal instruction before printing an error message.

Mnémonic: 'D' like dump. Not the same as 'd'.

4.2.6 -t: function names of protection/translation memory addresses

It defines prefixes for the functions in charge of the protection/translation of memory addresses. This option has 2 arguments: the type of access ("code", "data" or "direct") and the prefix of the function names. For each prefix, two functions must exist: <prefix>_read and <prefix>_write, whose prototype is the following:

```c
address_t trans_code(int complete, memory_t *mem, address_t a, instruction_t *i);
```

Example: using the command line gliss ... -t code foo_code -t direct foo_direct, external modules must define four functions: foo_code_read, foo_code_write, foo_direct_read, foo_direct_write. See section 3.4.1 on page 28 for more details.

Mnémonic: 't' like translation.
4.2.7 -a : name of additional fields
It defines the name of an additional field of the nMP description which must be used. This option has 2 arguments: the name of the field and its type. See section 3.6.2 for more details.
Mnémonic: 'a' like adding a field.

4.3 Various options

4.3.1 -h/-help : prints the online help
It prints the list of the options of GLISS.
Mnémonic: 'h' like help.

4.3.2 -v : prints the version
It prints the version of gliss.
Mnémonic: 'v' like version.

4.3.3 -d : debug level
It specifies the GLISS level debug i.e. the number of information printed by GLISS when generating a library. A level is a set of 32 flags. To each flag corresponds information that must or must not be printed.

<table>
<thead>
<tr>
<th>level</th>
<th>flag</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0x00000000)</td>
<td>base information</td>
</tr>
<tr>
<td>1</td>
<td>(0x00000001)</td>
<td>steps for instructions</td>
</tr>
<tr>
<td>2</td>
<td>(0x00000002)</td>
<td>generation of instructions images</td>
</tr>
<tr>
<td>3</td>
<td>(0x00000004)</td>
<td>summary of instructions images</td>
</tr>
<tr>
<td>4</td>
<td>(0x00000008)</td>
<td>generation of identifiers</td>
</tr>
<tr>
<td>5</td>
<td>(0x00000010)</td>
<td>generation of decoding tables</td>
</tr>
<tr>
<td>6</td>
<td>(0x00000020)</td>
<td>generation of symbol tables</td>
</tr>
<tr>
<td>7</td>
<td>(0x00000040)</td>
<td>generation of parameters</td>
</tr>
<tr>
<td>9</td>
<td>(0x00000100)</td>
<td>generation of arguments decoding</td>
</tr>
<tr>
<td>10</td>
<td>(0x00000200)</td>
<td>size of variables</td>
</tr>
<tr>
<td>11</td>
<td>(0x00000400)</td>
<td>generation of tables (short)</td>
</tr>
<tr>
<td>12</td>
<td>(0x00000800)</td>
<td>generation of tables (detail)</td>
</tr>
</tbody>
</table>

Example of use:
-d 0x00000809 activates levels 12, 4, et 1.

Note: Option ‘-d’ followed by “help” prints the list of levels as above.
Mnémonic: 'd' like debug.

4.3.4 -w : warnings masks
It defines the warnings which should not be printed. As for option ‘-d’, the argument is a set of flags and to each flag corresponds warnings which must or must not be printed. Whatever this value, all warnings will be recorded in the file gep.warnings.
<table>
<thead>
<tr>
<th>level</th>
<th>flag</th>
<th>information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00000000</td>
<td>warnings which can’t be hidden</td>
</tr>
<tr>
<td>1</td>
<td>0x00000001</td>
<td>warnings concerning the command line (twice the same switch for example)</td>
</tr>
<tr>
<td>2</td>
<td>0x00000002</td>
<td>variables, memories or registers which are described but not used</td>
</tr>
<tr>
<td>3</td>
<td>0x00000004</td>
<td>string constants not used</td>
</tr>
<tr>
<td>4</td>
<td>0x00000008</td>
<td>warnings concerning memory definitions</td>
</tr>
<tr>
<td>5</td>
<td>0x00000010</td>
<td>reverse bits order in &lt;..&gt;</td>
</tr>
<tr>
<td>6</td>
<td>0x00000020</td>
<td>arguments which are modified</td>
</tr>
<tr>
<td>7</td>
<td>0x00000040</td>
<td>faulty additional fields</td>
</tr>
</tbody>
</table>

**Note:** Option `-w` followed by “help” prints the table above.

**Mnémonic:** ‘w’ like warning.
Chapter 5

Evolutions

5.1 GLISS version 1.1.6

All names starting by `emul_` are replaced by the same names but starting by `iss_`. Example: `emul_init` becomes `iss_init`.

**Impact:** external files, simulator: change the names.

5.2 GLISS version 1.1.5

5.2.1 Memory

Management of translation/protection of memory addresses (since v1.1.5)

There are three types of memory accesses for each memory described in the nMP file. The prototype of a translation/protection function is:

```c
address_t trans_code(int complete, memory_t *mem, address_t a, instruction_t *i);
```

Since GLISS 1.1.5, functions are different for reads and writes and direct mode functions might be used to just check access rights and not translate addresses. See section 3.4.1, page 26 for more details on using these functions.

**Impact:** external modules, simulator: the good types and functions must be used.

Taking in charge several memories (since v1.1.3)

As several memories can be defined, it must be specified which memory will contain the code of the application to execute on the simulator. A option (`-m`) has been added.

**Impact:** nMP description: keywords `reg` and `mem` have now their own meanings. Thus the keyword `mem` must be used only for memories; Be careful: 1.0 descriptions might not be correct in 1.1.3 and higher.

**Impact:** external modules: Access functions to the memory have changed. See section 3.4.1 on page 26, page 26.

Memory interface (since v1.1.5)

The new version is defined in section 3.4.1 on page 26.

**Impact:** external modules: name of functions have changed.

Endianness of memories (since v1.1.5)

Now, the endian is defined for each memory. It might be a constant or a C expression. The constant `<memory_name>_is_little` must be defined to tell if the memory is little or big.
The endian for the loader is not useful anymore and, thus the '-f' option is not valid.

**Impact:** nMP description: The constant `byte_order` is no more used. It must be removed from the description. See section 2.9 on page 18;

**Impact:** external modules: The new memory access functions must be used. See 3.4.1 on page 26;

**Impact:** library Makefile: the '-f' option must be removed. It is no more valid.

### 64 bits and higher memories (since v1.1.4)

Memories may now be specified with and address bus of more than 32 bits.

**Impact:** nMP description: The argument of a memory definition is now the number of bits of addresses and no more the number of bytes of the memory.

Example: `mem M[32,byte].`

Note: nMP descriptions written for GLISS 1.0 must be modified.

### 5.2.2 Portability

**Irg available on Sun (since v1.1.3)**

All the tools are now running under Solaris on SPARC machines.

**Impact:** Processor Makefile: specification of architecture suppressed.

### New macro processor (since v1.1.4)

A new macro-processor `mp2m1.pl` has replaced `m4g.pl` and `m4`. Macro calls with a wrong number of arguments are now considered as errors while they were ignored before. Moreover, warnings are printed if arguments of a macro are not used in the macro.

**Impact:** Processor Makefile: dependencies `%nml : %nml.m4` et `%nml.m4 : %nmp` should be replaced by dependency `%nml : %nmp` in old Makefiles.

### 5.2.3 Interfaces

**Additional fields (since v1.1.4)**

On can add its own fields in a nMP description. See section 3.6.2 page 30 and section 4.2.7 page 35 for more details.

**Impact:** nMP description: fields can be added and they are taken into account provided they are specified on the GLISS command line.

**Impact:** simulator: Access to the additional fields can be implemented

**Impact:** Makefile: '-a' options are not recognized by GLISS versions older than 1.1.4.

### New types (since v1.1.4)

The interface between the simulator and the external modules has been modified. Now, a pointer to a logical state must be passed to an external function. The “real” state of the processor is given by the `iss_init` function. To know which registers/memory are used/modified by an instruction, the fields instrinput/instroutput must be used. See 3.3.3 page 23 for more details.

**Impact:** external modules: Old functions having `ii_t *ii` as argument must be changed to have `state_t state`.

**Impact:** simulator: the `iss_init` function returns a pointer towards the real state of the
processor which is usually transmitted to the \texttt{iss\_compute} function. The macros which provide access to the registers must have a \texttt{state\_t} * argument (and no more a \texttt{ii\_t} * argument).

Include files (since v1.1.2)

\texttt{include} files have been simplified.

\textbf{Impact}: external modules: \texttt{iss\_extern.h} and eventually \texttt{operators.h} are the only files to include.

\textbf{Impact}: simulator: \texttt{iss\_include.h} and eventually \texttt{operators.h} are the only files provided by GLISS to include in the simulator.

Definition of external modules without initialization function (since v1.1.1)

Now, external modules may not have a systematically-called initialization function. Use the '\texttt{E}' option instead of the '\texttt{e}' one on the \texttt{gliss} command line. This decreases the number of parameters passed to the \texttt{iss\_init} function.

\textbf{Impact}: Processor Makefile: Option '\texttt{E}' instead of '\texttt{e}' for modules which do not need initialization.

\textbf{Impact}: external modules: if initialization is not necessary, \texttt{(void <module\_name>\_init(state\_t *state, void *arg\_list[])}} should be removed.

\textbf{Impact}: simulator: \texttt{NULL} pointers corresponding to modules without initialization must be removed form the \texttt{iss\_init} function call.

\texttt{iss\_halt} function added (since v1.1.3)

The aim of this function is to free the memory previously allocated. External modules having an initialization function must have a \texttt{halt} function even if it empty because the module does not allocate memory. See 3.3.7 on page 25 for more details.

\textbf{Impact}: external modules: if there is an initialization, the \texttt{void <module\_name>\_halt(void)} function must be added. If a module allocates memory, it must have an initialization function and a halt function which frees the memory. The initialization function might be empty but should be there.

\textbf{Impact}: Processor Makefile: for modules without initialization, '\texttt{E}' should be changed in '\texttt{e}'.

\textbf{disassembling information} (since v1.1.4)

Syntax can now be included in the library and a function is provided to get this syntax (see page 25).

\textbf{Impact}: library Makefile call: \texttt{OPT=-DISS\_DISASM} should be added to the library Makefile command line in order to use this function.

Example: \texttt{make all OPT=-DISS\_DISASM}.

Decoding optimization (since v1.1.2)

By default, a cache of decoded instructions is used in the library. This cache should be deactivated if self-modifying code is executed on the simulator. The cache is deactivated with the \texttt{ISS\_NO\_CACHE\_DECODE} option. Furthermore, the prototype of the \texttt{iss\_decode} function has changed: an additional argument is the instruction address (which is needed to look in the cache). The cache can be configured. To see the possible configuration options, type \texttt{make info} in the \texttt{src} directory created by GLISS.

\textbf{Impact}: simulator: the \texttt{iss\_decode} function call must be modified, even if the cache is not
used.

**Impact:** library Makefile: the -DISS_NO_CACHE_DECODE flag should be added if the decode cache is not wanted. Example: `make all OPT=-DISS_NO_CACHE_DECODE`. 
Bibliography

Annex : Sim-nML Grammar

MachineSpec :
  | MachineSpec LetDef
  | MachineSpec TypeSpec
  | MachineSpec MemorySpec
  | MachineSpec RegisterSpec
  | MachineSpec VarSpec
  | MachineSpec ModeSpec
  | MachineSpec OpSpec
  | MachineSpec ResourceSpec
  | MachineSpec ExceptionSpec
  | MachineSpec error

LetDef: LET ID
  = LetExpr

ResourceSpec: RESOURCE

ResourceList

ResourceList:
  ID
  | ID '[: CARD_CONST ]'
  | ResourceList ',' ID
  | ResourceList ',' ID '[: CARD CONST ]'

ExceptionSpec: EXCEPTION

IdentifierList

IdentifierList:
  ID
    | IdentifierList ',' ID

TypeSpec: TYPE ID
  = TypeExpr

TypeExpr: BOOL
  | INT '(' LetExpr ')' 
  | CARD '(' LetExpr ')' 
  | FIX '(' LetExpr ',' LetExpr ')' 
  | FLOAT '(' LetExpr ',' LetExpr ')' 

; OpSpec: OP ID
OpRulePart
;
OpRulePart: AndRule AttrDefList
| OrRule
;
OrRule: ’=
Identifier_Or_List
;
Identifier_Or_List:
ID
| Identifier_Or_List ’|’ ID
;
AndRule: ’( ParamList ’)
;
ParamList:
| ParamListPart
| ParamList ’,’ ParamListPart
;
ParamListPart:
ID
 ’:’ ParaType
;
ParaType: TypeExpr
| ID
;
AttrDefList:
| AttrDeflist AttrDef
;
AttrDef:
ID ’=’ AttrDefPart
| SYNTAX ’=’ AttrExpr
| IMAGE ’=’ AttrExpr
| ACTION ’=’ ’{’ Sequence ’}’
| USES ’=’ UsesDef
;
AttrDefPart:
Expr
| ’{’ Sequence ’}’
;
AttrExpr:
ID ’.’ SYNTAX
| ID ’.’ Image
| STRING_CONST
| FORMAT ’(’ STRING_CONST ’,’ FormatIdlist ’)’
;
FormatIdlist:
FormatId
| FormatIdlist ’,’ FormatId
FormatId:
ID
| ID ',' IMAGE OptBitSelect
| ID '.' SYNTAX
| DOLLAR '+' ID
;
OptBitSelect:
| BIT_LEFT CARD_CONST DOUBLE_DOT CARD_CONST BIT_RIGHT
;
Sequence:
| statementList ','
;
StatementList:
Statement
| StatementList ',' Statement
;
Statement:
| ACTION
| ID
| ID '.' ACTION
| ID '.' ID
| Location '=' Expr
| ConditionalStatement
| STRING_CONST '(' ArgList ')
| ERROR '(' STRING_CONST ')
;
ArgList:
| Expr
| Arg1List ',' Expr
;
Opt_Bit_Optr:
| BIT_LEFT Bit_Expr DOUBLE_DOT Bit_Expr BIT_RIGHT
;
Location:
ID Opt_Bit_Optr
| ID '[' Expr ']' Opt_Bit_Optr
| Location DOUBLE_COLON Location
;
ConditionalStatement:
IF Expr THEN Sequence OptionalElse ENDIF
| SWITCH '(' Expr ')' '{' CaseList '}
;
OptionalElse:
| ELSE Sequence
;
CaseList:
CaseStat
| CaseList CaseStat
;
CaseStat:
CaseOption ' : ' Sequence
;
CaseOption:
CASE Expr
| DEFAULT
;
Expr:
COERCES (' Type ',' Expr)'
| FORMAT (' STRING_CONST ',' ArgList ')'
| STRING_CONST (' ArgList ')
| ID '.' SYNTAX
| ID '.' IMAGE
| ID '.' ID
| Expr DOUBLE_COLON Expr
| ID '[' Expr ']' Opt_Bit_Optr
| ID Opt_Bit_Optr
| Expr '+' Expr
| Expr '-' Expr
| Expr '*' Expr
| Expr '/' Expr
| Expr '%' Expr
| Expr DOUBLE_STAR Expr
| Expr LEFT_SHIFT Expr
| Expr RIGHT_SHIFT Expr
| Expr ROTATE_LEFT Expr
| Expr ROTATE_RIGHT Expr
| Expr '<' Expr
| Expr '>' Expr
| Expr LEQ Expr
| Expr GEQ Expr
| Expr EQ Expr
| Expr NEQ Expr
| Expr '&&' Expr
| Expr '!' Expr
| '|' Expr
| '!' Expr
| '|' Expr
| '++' Expr %prec '++'
| '--' Expr %prec'--'
| Expr AND Expr
| Expr OR Expr
| '(' Expr ')''
| FIXED_CONST
| CARD_CONST
| STRING_CONST
| DOLLAR
| BINARY_CONST
| HEX_CONST
| IF Expr THEN Expr OptionalElseExpr ENDIF
<p>| SWITCH '{' Expr '}' '{' CaseExprList '}' |
| BitExpr: |
|     ID |
|     BitExpr '<em>' BitExpr |
|     BitExpr '-' BitExpr |
|     BitExpr '</em>' BitExpr |
|     BitExpr '/' BitExpr |
|     BitExpr '%' BitExpr |
|     BitExpr DOUBLE_STAR BitExpr |
|     '(' BitExpr ')' |
|     FIXED_CONST |
|     CARD_CONST |
|     STRING_CONST |
|     BINARY_CONST |
|     HEX_CONST |
| CaseExprList: |
| CaseExprStat |
|     CaseExprList CaseExprStat |
| CaseExprStat: |
| CaseOption ':' Expr |
| OptionalElseExpr: |
|     ELSE Expr |
| UsesDef: |
| UsesOrSequence |
|     UsesDef ',' UsesOrSequence |
| UsesOrSequence: |
| UsesIfAtom |
|     UsesOrSequence '|' UsesIfAtom |
| UsesIfAtom: |
| UsesIndirectAtom |
|     IF Expr THEN UsesIfAtom OptionalElseAtom ENDIF |
| OptionalElseAtom: |
|     ELSE UsesIfAtom |
| UsesIndirectAtom: |
| UsesCondAtom |
|     ID '.' Uses |
|     '(' UsesDef ')' |
| UsesLocationList AND ID '.' Uses |
| UsesLocationList AND '(' UsesDef ')' |
| UsesCondAtom: |
| UsesAndAtom |</p>
<table>
<thead>
<tr>
<th><code>{ Expr '}</code></th>
<th>UsesAndAtom</th>
</tr>
</thead>
</table>
| `|` | UsesAndAtom:
| UsesLocationList | UsesActionList |
| `|` | UsesActionList:
| | ActionTypeList OptionalAction |
| | TimeActionList OptionalTime |
| `|` | ActionTypeList:
| | `#` `{ Expr '}`
| | ActionTypeList `:` UsesActionAttr `#` `{ Expr '}` |
| `|` | TimeActionList:
| | `:` UsesActionAttr |
| | TimeActionList `#` `{ Expr '}` `:` UsesActionAttr |
| `|` | OptionalAction:
| | `:` UsesActionAttr |
| `|` | OptionalTime:
| | `#` `{ expr '}` |
| `|` | UsesActionAttr:
| | ID |
| `|` | ACTION |
| `|` | UsesLocationList:
| | UsesLocation |
| `|` | UsesLocationList `&` UsesLocation |
| `|` | UsesLocation:
| | ID Opt_Bit_Optr |
| | ID `[` Expr `]` Opt_SecDim Opt_Bit_Optr |
| `|` | Opt_SecDim:
| | `[` `]` |