Functional Specification of Salto: A Retargetable System for Assembly Language Transformation and Optimization

`rev. 1.2pl2`

François Bodin   Zbigniew Chamski   Erven Rohou   André Seznec

October 16, 1997

Abstract

This document presents, in a user-oriented approach, the basic concepts and a specification of SALTO, a retargetable *System for Assembly Language Transformation and Optimization.* SALTO provides a framework for implementing complex manipulations of low-level codes, necessary when tuning performance-critical applications. This is achieved through the use of a detailed description of the target architecture, covering the instruction set, hardware configuration, and resource reservation tables of all instructions. An object-based user interface allows to easily implement complex transformations directly on the abstract representation of assembly programs, making it easy to cope with new hardware configurations, assembly language formats, and optimization techniques. The resulting software infrastructure lets the user concentrate on the implementation of actual optimizations and code instrumentation methods, suppressing the concern about implementing house-keeping tasks.

Contents

Chapter 1

Introduction

SALTO is a retargetable framework for developing a whole spectrum of tools that manipulate programs expressed in assembly language. The objective of such a system is to provide the user with a single environment that will allow him to implement tools needed for performance tuning on low-level codes, including assembly code schedulers, and profiling and tracing tools. The latter provide the user with information on where to focus optimizations and how efficient they can be, therefore allowing trade-off choices. Such a system is intended to address general computing as well as embedded systems, where optimizations are more critical and aggressive, but also where time-consuming techniques are more tolerable.

A large number of tools have been built to experiment with new optimizations or particular hardware mechanisms. This development phase is generally time-consuming and requires much investment. Utilities able to trace or profile programs exist, but they are often provided "as is": they are target-specific, and studying the behavior of another architecture or a different problem is likely to require a thorough rewriting of large pieces of code. As SALTO is retargetable wrt. the instruction set and the hardware details of the target hardware, it is likely to be a major help for such studies.

With SALTO we plan to address the field of software analysis and optimization for superscalar and VLIW architectures. The tool provides support for replicated resources and arbitrary levels of detail in hardware models. SALTO overcomes many limitations of previous solutions: it does not implement any algorithm by itself, and does not commit the user to a fixed set of techniques. Therefore, it should not be viewed as a compiler. At the same time, it does not operate on executable codes: the user of the tool can use human-readable information available in the assembly code to drive optimization of instrumentation strategies, whereas in executable code editors, much of this information is already lost.

To the user, SALTO provides an object-oriented interface designed to help manipulating assembly code. The classes provided by the interface allow to represent the complete description of the control-flow graph of the program (when available) and a model of the target architecture. They are easily accessible through the user interface and provide a comfortable way to implement algorithms without having to worry about supporting software.

1.1 System Overview

SALTO is composed of three parts: a kernel, a machine description file and an optimization or instrumentation module provided by the user. Figure 1.1 illustrates the organization of these three components.

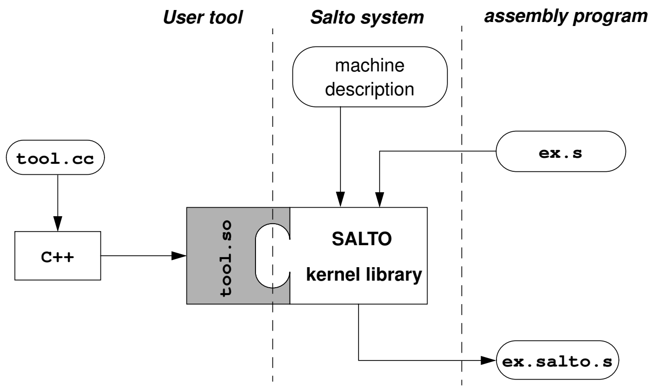

Figure 1.1: System Overview

- The kernel performs common house-keeping tasks the user doesn't want to worry about. The parsing of the machine description file and the assembly code, and the construction of the internal representation are done automatically. The internal representation is then available via the kernel's user interface.
- The machine description file provides a model of hardware configuration and the complete description of the instruction set, including per-instruction resource reservation tables.
- The optimization or instrumentation module is supplied by the user, and provides two entry points: the main function `Salto_hook`, and (optionally) the initialization function `Salto_init_hook`. If supplied, the initialization function is called immediately after parsing command line arguments. The system then reads the machine description file and the assembly code. Once the internal representation is successfully built, the control is passed to the user by calling `Salto_hook`.

1.2 Concepts and Features of Salto

Our system is built on three components: the data structures for representing the program, the machine description used to represent resource usage, and finally, on the user interface that enables the writing of instrumentation or scheduling algorithms.

1.2.1 Manipulated Objects

Data structures used in Salto are divided into two groups, depending on their role: on one hand, the representations of program control flow, and on the other, the descriptions of resource usage and data dependencies between instructions.

A program written in assembly language can be viewed as consisting of several procedures, each of which is a list of instructions. Within a procedure, instructions are grouped into basic blocks.

While parsing the code, Salto builds the list of the procedures it encounters. Each procedure has a list of basic blocks, and each block "knows" its list of instructions. Figure 1.2 illustrates this object structure. The internal representation of these objects is hidden by the user interface, as shown in section 1.2.2. Markers are added by the analyzer to give useful information about the code being processed: basic blocks frontiers (shaded instructions in figure 1.2), or procedures frontiers, name of current segment, etc.

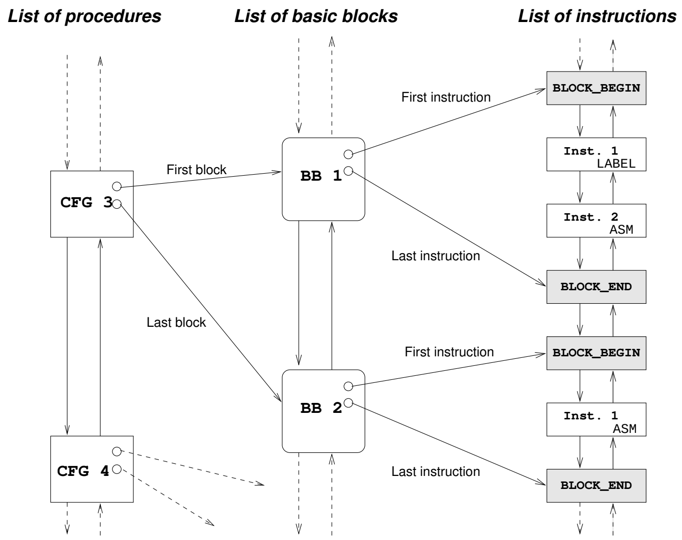

Figure 1.2: Organization of Control-Flow Structures

After parsing the code, a control flow graph (CFG) is built for each procedure. The vertices of a control flow graph are basic blocks and its edges denote the execution order of the blocks. Edges are labeled to indicate if they correspond to the taken or not-taken branch (see figure 1.3 for an example of the graph corresponding to a simple procedure written in

C language.) A known limitation of this scheme is its conservative nature, due to the static analysis of the assembly-code. If the target address of a branch instruction is a computed register value, then the SALTO parser leaves the determination of the actual branch target to a user-supplied tool.

```
/*
 * Computes the GCD of two integers
 * using Euclide's method
 */
int gcd(a,b)
      int a, b;
{
   int result;

   if (a<b) a∧ =b, b∧ =a, a∧ =b;
   if (a%b) result = gcd(a-b,b);
   else result=b;
   return result;
}
```

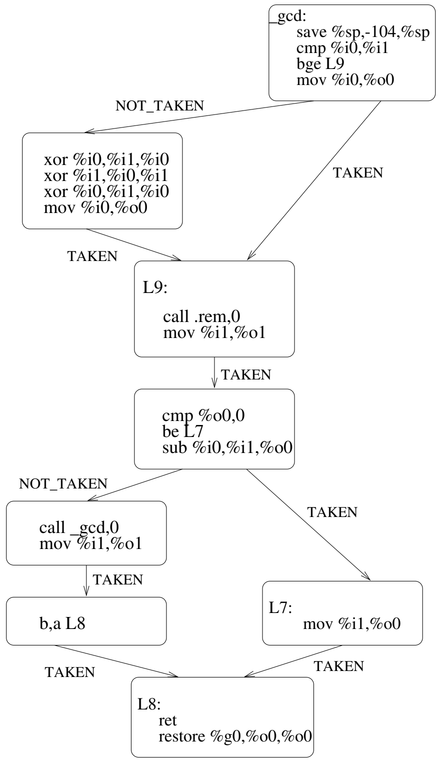

Figure 1.3: Control Flow Graph of a Simple Program

The second part of the data structures provided by SALTO gives information about the resources needed by an instruction to complete its execution. A resource is usually a register, a functional unit or the memory, but it could be any piece of hardware needed to describe the behavior of the machine (see section 1.2.3 for an explanation on how to define resources.) Each instruction needs a resource during a number of cycles with a particular access mode: either *read*, *write* or *use*.

Each instruction is described by a reservation table, which indicates the list of resources it needs and the mode and cycle a resource is accessed. This information is used when determining the type of data flow between two instructions: RAW (*read after write*), WAW (*write after write*), WAR (*write after read*).

The memory is seen as a unique resource and all memory accesses are considered to be to the same memory location. SALTO is conservative when checking data dependences and thus two memory accesses, one of which is a write, always lead to a dependence. However, the functions of the user interface make it possible for the user to write his own alias analysis algorithm to detect such situations and to implement a link to the data dependence analysis subsystem of a compiler.

1.2.2 User Interface

The object-oriented user interface provides a flexible way to deal with the internal data structures. Features of SALTO in this field include:

- access to the code at three different levels: procedure, basic block or instruction;
- modification of the code: insertion, deletion and modification of objects at each abstraction level ;
- unparsing;
- access to resource reservation tables;
- computation of dependences and delays between instructions caused by pipeline stalls;
- possibility of attaching arbitrary attributes, or annotations, to any kind of objects.

1.2.3 Target System Descriptions

SALTO is designed to be a retargetable tool. Thus, the target machine is described in a flexible way, allowing an accurate description at assembly and hardware level while retaining the ability to easily modify individual parameters.

This goal is achieved through the use of a Lisp-like language based on the reservation tables formalism. The description file is parsed by SALTO and an internal representation is built using RTL (Register Transfer Language) [29]. The target system description covers:

- the lexical and syntactical structure of the assembly language used — how do comments start, what are register names etc.;
- all the resources referenced by the instructions, needed for the computation of data dependences and access conflicts;
- the list of the instructions recognized by the assembler together with their various formats, and the associated reservation tables, including all predefined macro-instructions of the assembler;
- semantical information to warn SALTO about special features implemented in the architecture being described, such as *register by-pass* or *delayed branch* mechanisms.

1.2.4 Structure of the Manual

The remainder of this report is organized as follows. Section 2 provides the specification of target description files. Section 3 contains the detailed specification of the user interface, covering the types, classes, functions, and methods we intend to provide in SALTO. Finally, in section 4 we describe two applications developed using a limited prototype of the tool. These examples are further developed in the appendix. We conclude with an outline of intended development and experimentation work.

Chapter 2

SALTO Target Description Specifications

This chapter contains reference information on writing target system descriptions. These descriptions are the only source of information on the behavior of the target system, providing an effective separation of the target characteristics from the implementation of the transformation toolbox.

This chapter is divided into three major parts. The general notions underlying the description model are presented in section 2.1. The recommended organization of the actual description files is given in section 2.2. Sections 2.3 through 2.7 detail the components of target descriptions. For each concept represented in a target system description, these sections present the corresponding constructs, their syntax, and the associated semantical constraints when applicable.

2.1 Description Overview

The description of the target system contains the information on the architecture and its basic software environment, namely:

- the outline of assembler syntax and lexical structure;
- the list of hardware resources of the architecture in terms of registers, memories, functional units etc.;
- the description of the instruction set of the architecture, including predefined macros implemented by the assembler;
- the information on instruction semantics, such as branch delays;
- assembly directives (pseudo-instructions of the assembler).

The basic entities specified in the description of the assembly language are:

- comments, including comment start and end markers,
- register designations,
- operand designations,
- representations of addressing modes,
- mnemonics, macros and assembler directives.

Resources appearing in the target machine description are a model of *hardware components* potentially relevant to the optimization process. Therefore, any resource which can limit the exploitation of parallelism available in the source code should be represented, allowing its description to be used in the optimization process.

Resources which share the same set of properties (general registers, replicated functional units, etc.) are grouped into resource *classes*. Classes can be then used to designate "generic resources", such as "one of the general-purpose registers".

References to resources and resource classes appear in the description of *reservation tables* of individual instructions or operations. A reservation table indicates which resources are used by an instruction, at which cycles, and what use the instruction makes of each resource listed.

The reservation table concept provides a basis for the optimization process in presence of multiple resources, instruction pipelines, and instruction-specific scheduling constraints, by allowing the detection of conflicts and alternative resources. Reservation tables are attached to instructions when defining the target instruction set.

Instruction set descriptions contain the name (i.e., the mnemonic) of each instruction or operation, its format and operands, and a reservation table that includes all resources (or resource classes) required for its execution. The description of each macro-instruction contains the corresponding sequence of basic instructions or operations, and the rules for constructing their operands from the actual operands of the macro.

The instruction set description is completed by semantical information attached to control flow instructions. The three types of semantical constraints that are supported are *delayed load*, *delayed branch*, and *write by-pass*.

Finally, the description of assembly directives allows to analyze data declarations, data and code interleaving, alignment information etc.

The description of target systems uses a declarative language inspired by RTL, the "*Register Transfer Language*" used in machine descriptions of the GCC compiler suite. A description consists of a sequence of RTL definitions, in which any non-standard object must be defined before use. In addition, cpp directives can be used to avoid unnecessary expression repetition and to provide a hierarchy to the description files.

RTL's syntax is close to that of LISP. Comments start with a semi-colon (";") and continue to the end of line. Four types of RTL objects are supported in SALTO:

- integers: strings of digits with an optional leading minus sign,
- character strings, enclosed in double quotes (e.g., `"string"`),
- vectors: sequences of RTL objects separated by spaces and enclosed in square brackets (e.g., [obj_1 obj_2 ... obj_n]),
- expressions: sequences of RTL objects separated by spaces, preceded by a predefined expression code, and delimited by parentheses (e.g., ($expcode$ op_1 op_2 ... op_n)).

2.2 Description Organization

Before being read by SALTO, the description file is fed through the `cpp` preprocessor. Any `cpp`-supported directives can be used to help simplify and clarify the target description. Two `cpp` features are commonly used:

- `#define` macro-definitions, allowing to describe with a single identifier any multiply referenced complex expressions, such as reservation tables common to several instructions;
- `#include` file inclusion directives, which provide a hierarchy of description files.

Using the above `cpp` mechanisms, the description of a given target system consisting of a hardware architecture ***arch*** and a software environment ***target***, the description will be split into five files named ***arch***`.md`, ***arch***`-resource.def`, ***arch***`-macro.def`, ***arch***`-semantics.def`, and ***target***`.cc` (see figure 2.1.)

The main file ***arch***`.md` contains inclusion directives for the three remaining files, the description of the lexical structure of the assembly language and the definition of the target instruction set as manipulated by SALTO. File ***arch***`-macro.def` contains the description of expansion rules for the predefined macros of the assembler. File ***arch***`-resource.def` provides the definition of hardware resources, resource classes, and reservation tables. Finally, file ***arch***`-semantics.def` provides the definition of instruction-specific semantical constraints.

Given the great variety of directive semantics between supported targets, the description of assembly directives and support routines for their manipulation are provided in a C++ file named ***target***`.cc` which is compiled into a shared object file. This separates all target-dependent code from the SALTO kernel library. The base name of this file is purposely different from that of the hardware description file, since multiple ***target***`.cc` files may have to deal with the peculiarities of various compiler/assembler combinations for the same hardware architecture.

arch.md :

arch-resource.def :

<definition of hardware resources>
<definition of resource classes>
<definition of reservation tables>

arch-macro.def :

<description of predefined assembler macros>

arch-semantics.def :

<description of specific instruction semantics>

<description of lexical units of the assembly language>
<definition of the instruction set>

target.cc

<assembler-specific directives and actions>

Figure 2.1: Recommended layout of target system description files

In the following, the syntax of target system descriptions is presented using the BNF notation. We use the following typographical conventions:

- terminal symbols (keywords, litterals etc.) appear in `typewriter` font;
- non-terminals appear in *italics* and enclosed between chevrons ("<" and ">");
- alternatives are represented using the vertical bar ("|");
- optional expressions are followed by the $^{0|1}$ superscript;
- expressions that occur *at least once* are followed by the $^{+}$ superscript;
- expressions that occur *zero or more times* are followed by the * superscript;
- a fully parenthesized expression or string followed by any of the above superscripts matches a number of occurrences of that expression or string, as indicated by the superscript.

2.3 Target Identification

The target architecture is specified using the expression

(target "<*target_name*>")

where <*target_name*> is the name of the architecture being described. This definition is mandatory.

The target name provided in the description file can be later extracted by calling the function `getTargetName()`.

2.4 Hardware Resources

In SALTO, a resource represents an element of the target architecture that can influence its performance. As a rule of thumb, anything hardware element that can form a bottleneck in instruction scheduling should be represented as a resource.

Resources are classified according to three criteria:

functionality : resources are used either to store data, or to manipulate it; typical storage resources are registers and the main memory, and are the only resources that can be *read* and*written*; resources that manipulate data can only be *used* (i.e., busy, or reserved) and, for our purposes, are called "*functional units*";

complexity : a basic resource corresponds to a precise component of the target hardware; composite resources consisting of several adjacent individual resources are also supported, in order to represent aggregate registers, treated as a single operand by some instructions;

genericity : resources with similar characteristics may have names that obey a systematic naming scheme, e.g., a prefix followed by a numerical suffix; resources sharing the same characteristics can be grouped into classes whose members can be substituted to each other when needed, even if they are not systematically named.

2.4.1 Basic Resources

Resources must be named. The choice of names is left to the designer of the description of the architecture, *except* for the reserved resource name `"mem"` which always denotes the "main" memory (NB: multiple memory resources can be defined.)

A resource is characterized by four properties:

- its *name*, which is used to reference the resource in classes, reservation tables etc.;
- its *type* which identifies the type of operations that can be performed on that resource;
- the width of the data paths to and from the resource;
- (only for functional units) the replication level, i.e., the number of equivalent resources sharing exactly the same characteristics.

The definition of a resource is made using the `def_ress` expression with the following syntax:

```
<def_ress> ::=
        (def_ress
            <res_or_class_name>
            [
              (type "<type>")^{0|1}
              (width <width>)^{0|1}
              (limit <limit>)^{0|1}
            ])

<res_or_class_name> ::=
            (name "<name>") |
            (base_name "<genname>" <first> <last>)
```

where

- *<name>* and *<genname>* are alphanumeric strings beginning with a letter; *<genname>* corresponds to the *name prefix* for a *numbered* replicated resource;
- *<first>* and *<last>* are integers delimiting the range of suffixes of the names of a *numbered* replicated resource whose name prefix is *<genname>*;
- *<type>* can be either “`reg`”, “`functional_unit`” or “`memory`”; any other value is understood as “`unknown`”; if this field is omitted, the last value given in preceding resource definitions is used;
- *<width>* is an integer indicating the width of the resource in bits;
- (functional units only) *<limit>* is an integer indicating the number of identical, interchangeable instances of the resource; if this field is omitted, the last value given in preceding resource definitions is used; the default initial value is 1.

The reference to an already defined resource is made using the expression `ress`:

```
<resource_ref> ::=
        (ress
           <res_or_class_name>)
```

Example:

The definitions

```
; general registers r0 to r127
(def_ress (base_name "r" 0 127)
           [(type "reg") (width 32)])
; instruction decoders (issue units)
(def_ress "issue"
          [(type "functional_unit")
           (limit 5)])
; DSP ALU
(def_ress (name "dspalu")
          [(type "functional_unit")
           (limit 1)
   (width 32)])
```

correspond respectively to a set of 128 32-bit registers named `"r0"` through `"r127"`, a family of five instruction decoders which can be used interchangeably, and a unique DSP ALU with 32-bit data paths (N.B.: the replication limit of 1 must be explicit: the previous functional unit definition used a higher value, which would otherwise have been reused.)

2.4.2 Resource Aliases

Aliases (alternate names) of resources can be given using the `alt_name` expression. Aliases are most frequently used to reflect usage conventions of resources. The general form of the alias definition is

```
<alt_name>::=
        (alt_name
            "<name>" "<alias>")
```

where

- *<name>* is a string designating a resource already defined using `def_ress`;
- *<alias>* is a string giving the equivalent resource name.

Example:

The following are the aliases of the TM1000 registers

```
; register r2 (alias "rp") contains the return value of a function
(alt_name "r2" "rp")

; register r3 (alias "fp") is used as frame pointer by C/C++ programs
;  (see restriction in manual)
(alt_name "r3" "fp")

; register r4 (alias "sp") is the stack pointer (last word IN USE, growth
;  towards 0)
(alt_name "r4" "sp")

; register r5 (alias "rv") is the scalar return value register
(alt_name "r5" "rv")
```

2.4.3 Resource Classes

Resource classes provide a means of representing resources that are equivalent wrt. the renaming. Therefore, the classes should be defined in such a way that all resource of a class should be fully interchangeable.

A resource can belong to several classes, with different renaming and writability characteristics in each class. A resource class definition uses the following syntax:

```
<def_class> ::=
        (def_class
            "<class_name>" "<type>"
            [
              (ress | class
                  <res_or_class_name>
                  [
                    (norename)^0|1
                    (noallocate)^0|1
                  ]
              )+
            ])
```

where

- <*class_name*> is the alphanumeric string giving the name of the class;
- <*type*> is the type of the resources in the class, either "`reg`", "`functional_unit`" or "`memory`"; any other value is understood as "`unknown`";
- `(norename)`, when present, indicates that the resource cannot be renamed to another resource, i.e., cannot be substituted *by* another resource (example: a dedicated register — stack pointer, frame pointer etc.);

- `(noallocate)`, when present, indicates that the resource cannot be used as renaming target, i.e., cannot be substituted *to* another resource.

References to an already defined class are made using the `class` expression:

```
<class_ref> ::=
        (class
           <res_or_class_name> )
```

Example:

Below are the definitions of several register classes of the TM architecture:

```
; read-only registers
(def_class "RO_reg" "reg"
           [(ress (base_name "r" 0 1)   [(noallocate) (norename)])])

; writable registers
(def_class "DEST_reg" "reg"
           [
            (ress (base_name "r" 2 127) [])
           ])

; all registers
(def_class "GP_reg" "reg"
           [
            (class (name "RO_reg"))
            (class (name "DEST_reg"))
           ])

; C call parameters
(def_class "PARAM_reg" "reg"
           [
            (ress (base_name "r" 5 8) [(norename)])
           ])
```

Notes:

Resource names must be defined prior to the definition of resource classes, otherwise they will be ignored.

2.5 Reservation Tables

A reservation table lists all resources used by the corresponding instruction and, for each resource, describes the type and schedule of the accesses made to that resource, and any relevant renaming and scheduling constraints associated with that resource.

Resources can be designated by basic resource and class names, as specified positional operands of the instruction, or as sets of registers (multi-registers).

Reservation tables are *not* named. Therefore, to avoid replication of code in the descriptions of instruction sets, each reservation table should be defined as a `cpp` macro, and the resulting macro subsequently invoked in relevant instruction descriptions.

The syntax of reservation table definitions is

```
<reser_table>::=
        (reser_table
            [
              (<any_resource_ref>
                 [
                   (<usage>)
                   <schedule_info>
                   (class (name "<class_name>"))^{0|1}
                   (norename)^{0|1}

                 ]
              )^+
            ])

<any_resource_ref>::=
           <resource_ref> |
           <positional_param> |
           <multiregister>


<positional_param>::=
        (match_arg
           <num_arg>)

<multiregister>::=
        (multi_reg
           <positional_param> <index> <size>)

<schedule_info>::=
           (at_cycle <at>) |
           (from_cycle <from>) (to_cycle <to>)
```

where

- <*usage*> is either `read` or `write` (registers or memory), or `use` (functional units);
- <*class_name*> is the name of the class of resources which can be substituted to the

current resource during resource renaming; this class must already be defined;

- `(norename)`, if present, indicates that the resource cannot be substituted (renamed); this property takes precedence on the specification of a renaming class;
- <*positional_param*> matches the specified operand in the operand list of the instruction (or family thereof) to which the reservation table applies; <*num_arg*> is the position (counted from 0) of the operand;
- <*multiregister*> corresponds to a multi-register, i.e., an aggregate resource consisting of <*size*> consecutive hardware registers beginning <*index*> registers from the base register designated by `match_arg`; e.g., if the first operand of the instruction is `r32`, `(multireg (match_arg 0) 4 2)` corresponds to a multi-register set consisting of registers `r36` and `r37` — two registers starting four registers past `r32`; the offset <*index*> can be negative;
- <*from*> is the number of the *first* cycle at which the resource is reserved;
- <*to*> is the number of the *last* cycle at which the resource is reserved;
- <*at*> is the only cycle at which the resource is reserved: `(at_cycle` n`)` is equivalent to `(from_cycle` n`) (to_cycle` n`)`.

Example:

```
; resource common to all 'ftough' instructions
; instruction loading and decoding
#define ISSUE_FTOUGH (ress (name "issue2") [(use) (at_cycle 1)])

; general registers' class
#define   GP_REG_C   (class (name "GP_reg"))

; reservation table of 'ftough' instructions on the TM architecture
#define R_FTOUGH \
 (reser_table [ \
        ISSUE_FTOUGH \
        (ress (name "ftough") [(use)   (from_cycle 1) (to_cycle 16)]) \
(ress (match_arg 0)   [(read)  (at_cycle 1) GP_REG_C]) ; guard \
        (ress (match_arg 1)   [(read)  (at_cycle 1) GP_REG_C]) ; rsrc1 \
        (ress (match_arg 2)   [(read)  (at_cycle 1) GP_REG_C]) ; rsrc2 \
        (ress (match_arg 3)   [(write) (at_cycle 17)]) ; rdest \
        ])
```

2.6 Lexical Structure of the Assembly Language

The description of the lexical structure of the assembly language specifies the format of comments, operand separators and the generic operand-matching patterns used in identifying instruction operands during program parsing.

2.6.1 Comments

Three types of comments are supported:

- line comments,
- end-of-line comments,
- stream comments.

Line comments begin with a specific character or string in the first non-blank position of a line. End-of-line comments are the comments at the end of an otherwise non-empty line; they start with a specific character or string and are terminated by the end-of-line character. Stream comments are started and terminated by specific characters or strings.

Comment identification is controlled by the following definition:

- line comments on a line of their own:

  ```
  <line_comment_chars> ::=
          (line_comment_chars
              "<string>")
  ```

  where <*string*> is the character or the sequence of characters that start a line comment;

- end-of-line comments (comments following assembly language text):

  ```
  <comment_chars> ::=
          (comment_chars
              "<string>")
  ```

  where <*string*> is the character or the sequence of characters that start an end-of-line comment;

- start and termination of stream comments:

  ```
  <comment_start> ::=
          (comment_start
              "<start_string>")

  <comment_end> ::=
          (comment_end
              "<end_string>")
  ```

where

- <*start_string*> is the character or sequence of characters that begins a stream comment;
- <*end_string*> is the character or sequence of characters that terminates a stream comment;

Example:

```
(comment_chars "#")  ; end-of-line comment start on the MIPS
(comment_chars "!#") ; end-of-line comment start on the SPARC
(comment_start "(*") ; start of a stream comment on the TM family
(comment_end "*)")   ; end of a stream comment on the TM family
```

2.6.2 Non-Blank Separators

Single-character non-blank separators are listed in an aggregate definition of the form

```
<def_exact> ::=
        (def_exact
            "<string>" )
```

where <*string*> is a concatenation of the supported terminal symbols.

Multi-character separators are listed using a definition of the form

```
<def_separ> ::=
        (def_separ
            ["<string>"+])
```

where each <*string*> is the literal representation of a multi-character separator.

Notes:

1. The definition of instruction formats in the description of the instruction set can only use separators defined by means of a `def_exact` or `def_separ` expression.
2. If there are multiple `def_exact` or `def_separ` definitions, only the most recent one of each form is effective.

2.6.3 Pattern-Matching Tokens

The matching between actual operands in the assembly program and the symbolic operands in the description of the target instruction set relies on a set of meta-variables which must be recognized in appropriate positions when parsing the assembly program. In the target machine description, the meta-variables are designated using single-character tokens, and are associated with regular expression patterns describing the expected external representation of the corresponding actual operands.

A family of predefined input functions allows to read specific assembly language objects: identifiers, register names, arithmetic expressions, immediate integer or floating-point values:

- `regex` reads a sequence of characters matching a given regular expression <*reg_exp*>;
- `read_exp` reads an arithmetical expression;
- `read_imm` reads an integer value (signed or not);
- `read_flp` reads a floating-point value.

The full definition of a meta-variable follows the syntax

```
<meta_var_def> ::=
        (def_token
           "<char>"
           [
             (regex "<reg_exp>") |
             (read_exp) |
             (read_flp) |
             (read_imm <size> "<sign>")
           ] )
```

where

- <*char*> can be any non-blank alphabetic character not in the separator character sets (defined using `def_exact` and `def_separ`),
- <*reg_exp*> is the regular expression defining the pattern to be matched,
- <*size*> is the size of the immediate value in bits,
- <*sign*>, if equal to "`signed`", indicates that the number should be interpreted as signed. Any other value of <*sign*> forces the number to be treated as unsigned.

The syntax of admissible regular expressions is a subset of the classical regular expression syntax of `grep` and `Emacs`:

- a single dot ("`.`") matches any single character;
- a set of characters enclosed in square brackets ("`[`" and "`]`") matches any single character from that set; two characters separated by a dash ("`-`") in the set define an *interval* in terms of the ASCII code, from the character preceding the dash to the character following the dash, inclusive; a set starting with a caret ("`^`") matches any character *not* present in the set;

- a sequence of patterns matches the longest matching sequence of characters, starting from the leftmost pattern;
- an asterisk ("`*`") indicates *zero or more* occurrences of the preceding pattern;
- a sequence of patterns enclosed between a matching pair of strings "`\(`" and "`\)`" is treated as a single pattern;
- the string "`\|`" specifies the alternative between the largest single patterns adjacent to it;
- a backslash ("\") suppresses the special meaning of the immediately following character (backslash, dot, asterisk etc.).

Example:

```
; TM1000 tokens: registers or expressions

#define REG_TOKEN(CHAR)\
(def_token  \
  CHAR [(regex "r1[0-1][0-9]\\|r12[0-7]\\|\
                r[1-9][0-9]\\|r[0-9]")
       ])

#define EXP_TOKEN(CHAR)\
(def_token CHAR [ (read_exp) ])

REG_TOKEN("d")    ; destination register
REG_TOKEN("s")    ; first source register
REG_TOKEN("t")    ; second source register
REG_TOKEN("g")    ; guard register
EXP_TOKEN("m")    ; modifier
```

Notes:

1. It is good practice to order patterns in a regular expression so that the most specific patterns come first, and that the most general ones be at the end of the expression.
2. The choice of meta-variable names defined using `def_token` is purely arbitrary; the author of a machine description is free to choose a naming convention that is best suited for the design of the target system.

2.7 Instruction Sets

The description of the instruction set is made by providing individual descriptions of assembler-supported instruction/operation mnemonics and macros. Each instruction/operation description contains the name of the mnemonic, the meta-variable pattern identifying its operands, the reservation table information, and possibly, semantical information (presence of delays introduced by the instruction, etc.).

2.7.1 VLIW Instruction Width

The number of operations in a VLIW instruction (the maximum number of operations that can be issued in a single clock cycle) is defined using the expression

```
<inst_width>::=
        (inst_width
            <size>)
```

where *<size>* is an integer. If this definition is omitted, the default value of 1 is assumed.

2.7.2 Semantical Information

Semantical information supported by SALTO covers the specification of operands of control flow instructions, and the definitions of load, write, and branch delays (and the associated limitations on instruction scheduling).

In its most general form, the semantical information follows syntax

```
<semantical_info>::=
        (sem
           [<semantical_property>+])

<semantical_property>::=
           <control_target> |
           (return) |
           <delay_slot> |
           (noreorder)
```

Basic semantical information on branch, jump and call instructions uses a common format

```
<control_target>::=
        (<control_flow_insn>
           <num_arg>)
```

where

- *<control_flow_insn>* is either `branch`, `jump`, or `call`, depending on whether the instruction concerned is a conditional branch, an (unconditional) jump, or a subroutine call;

- <*num_arg*> is the position of the target address in the parameter list of the instruction, counted from zero; if the target is not known at compile time (i.e., indirect addressing is being used), the value of <*num_arg*> is `-1`.

The expression `(return)` indicates the return to an implicit address.

Delay slot information is provided using the `delay_slot` expression:

```
<delay_slot>::=
        (delay_slot
            <slot>)
```

where

- <*slot*> is the integer representing the branch delay, i.e., the number of instructions issued before the execution resumes at the new address.

The restrictions on instruction reordering introduced by the presence of delay slots are expressed using the `(noreorder)` property. If present, `(noreorder)` forbids the instructions *preceding* the current instruction to be rescheduled *after* it, and the instructions *following* it to be rescheduled *before* it. This restriction applies to instructions which terminate or are issued within the branch delay after the issue of the relevant control instruction.

Given the large variety and complexity of write-back by-pass definitions, by-pass facilities are expressed through a user-supplied hook function `updateDelay(...)` which produces updated instruction-to-instruction delay values on demand. In particular, this function (if available) is automatically called during the computation of minimum scheduling delay between instructions (see methods `int INST::getDelay()` and `int INST::getResDelay()` in section 3.3.5.)

2.7.3 Native Instructions

The general form of an instruction/operation definition is

```
<def_asm>::=
        (def_asm
            "<name>"
            [
              (input "<format>")
              <res_table>
              <semantical_info>^0|1
              (info "<any-text>")^0|1
            ])
```

where

- <*name*> is the mnemonic of the instruction/operation;

- <*format*> is a string representing the format of instruction operands; <*format*> consists only of terminal symbols defined using `def_exact` and `def_separ` expressions and of meta-variables defined using `def_token`;
- <*res_table*> is an expression describing the reservation table of the instruction or operation, either directly, or via a `cpp` macro-definition created using a `#define` directive;
- <*semantical_info*> describes the scheduling constraints introduced by the instruction/operation;
- the `info` field can contain any text. This field is used to attach an arbitrary textual information to an assembly instruction definition, and is not interpreted by SALTO. The text can be extracted at run time by calling method `INST::getAsmInfo()` on the instructions of the program (see section 3.3.5.)

Notes:

- The expression <*format*> (containing possibly an empty string) and a non-empty expression <*res_table*> are mandatory. However, the semantical information section is optional.
- If the target architecture uses an infix instruction format (e.g., "`opnd1 INSN opnd2 ... opndn`", a suitable preprocessing tool should transform instructions specified this format into a prefix one, in which the instruction precedes all its operands.

Example:

```
(def_asm  "fadd"            ; floating-point addition
   [(input  "g,m,s,t,d")    ; d <- s+t, plus guard (g) and modifier (m)
    R_FLOAT_ALU             ; reservation table of FP arithmetic
   ])
(def_asm  "jmpi"            ; jump to immediate address
   [(input  "g,m")          ; if (LSbit(g) == 1) then PC <- m
    R_BRANCH                ; reservation table of branching insns
    (sem  [ BRANCH(1)       ; "m" is the immediate address operand
            (delay_slot 3)  ; branch delay is 3
            (noreorder)     ; prevent insn reordering
          ])
   ])
```

2.7.4 Macro-Instructions

Macro-instructions of the assembly language are described in two steps:

- definition of the name and the operands of the macro;

- definition of the expansion of the macro-instruction into native instructions and their respective operands.

The first definition uses the `def_macro` expression. Its syntax is close to that of a native instruction description, except that the semantical information section is replaced with the expansion of the macro-instruction into native instructions:

```
<def_macro> ::=
        (def_macro
            "<name>"
            [
              (input "<format>")
              (info "<any-text>")0|1
              <macro_expansion>
            ])
```

where

- <*name*> is the mnemonic of the macro-instruction;
- <*format*> is a string representing the format of instruction operands; <*format*> consists only of terminal symbols defined using `def_exact` and of meta-variables defined using `def_token`; it is assumed that all operands follow the mnemonic and that only single-character separators are used between operands;
- <*macro_expansion*> is an expansion of the macro-instruction into native instructions, described using an `expand` construct (defined below);
- the `info` field can contain any text. This field is used to attach an arbitrary textual information to an assembly instruction definition, and is not interpreted by Salto. The text can be extracted at run time by calling method `INST::getAsmInfo()` on the instructions of the program (see section 3.3.5.)

Example:

```
; MIPS branch to immediate offset if greater or equal
(def_macro  "bge"
   [ (input  "s,t,i"  ) ; if (s>=t) then PC <- PC + i
     EXPAND_BGE_1       ; expansion: EXPAND_BGE_1
   ])
```

Notes:

Macros are expanded during the parsing of the input program.

The expansion of macro-instructions is described using `expand` expressions which define the expansion of the macro into native instructions. Each native instruction in the expansion is described by its name, its input format, and a vector of operand construction directives.

For each meta-variable (formal parameter) in the input format of each native instruction in the expansion of the macro, there is a matching element in the vector operand construction vector. Each such field describes the way the corresponding actual parameter is built from the actual parameters of the macro and from predefined elements.

The expressions that can appear in the operand construction vector are divided in two groups: three basic forms, and three compound expressions that use the basic forms:

- an expression `with_reg` indicating that the operand is the specified register;
- an expression `match_arg` indicating that the operand is one of the actual operands of the macro, located at the specified position (counted from zero);
- an expression `const_expr` indicating that the operand is the arithmetic expression given;
- an expression `add_expr` indicating that the operand is constructed by adding an integer expression specified using `const_expr` to the actual macro parameter specified by its position in `match_arg`;
- an expression `multi_reg` (see section 2.5 above) indicating that operand is a multi-register data set consisting of <*size*> consecutive hardware registers beginning <*index*> registers from the base register designated by `match_arg` or `with_reg`;
- an expression `imm_part` indicating that the operand is built using a specified part (lower- or upper-weight bits) of the actual parameter; the part selected is indicated by the non-terminal <*part*>; the size of the subset extracted is given by <*width*>; <*sign*> indicated the type of extension to be applied to the subset extracted (either signed or unsigned).
  NOTE: this expression is intended to be applied to immediate operands only.

The syntax of the definitions of macro-instruction expansions is

```
<macro_expansion> ::=
        (expand
           [
             (build_asm
                "<ins_name>" "<ins_format>"
                [
                  (with_reg "<reg>")*
                  <positional_param>*
                  (const_expr "<expr>")*
                  (multireg
                     <base_register> <index> <size>
                  )*
                  (add_expr
                     <positional_param>
                     (const_expr "<expr>")
                  )*
                  (imm_part
                     "<part>" "<sign>" <width>
                     <positional_param>
                  )*
                ]
             )+
           ])

<base_register> ::=
        <positional_param> |(with_reg "<reg>")
```

where

- *<ins_name>* is the name of the macro-instruction;
- *<ins_format>* is the string describing the format of the operands of the macro-instruction; the string `"*"` indicates that the appropriate format should be determined at run-time;
- the expression `(with_reg <reg>)` indicates that the register *reg* should explicitly be used when expanding this macro-instruction;
- *<expr>* is an arithmetical expression evaluating to a constant, such as `"7*4"`;
- *<index>* is a (possibly negative) integer used to compute the name of the first register of a multi-register (an aggregate register spanning several hardware registers and used as a single operand) from the register designated by *<base_register>*;
- *<size>* is an integer describing the width, in hardware registers, of the multi-register data set;
- *<part>* is either `upper` or `lower`;

- <*sign*> is either `signed` or `unsigned`; `signed` indicates that the value extracted should be sign extended to the width expected by the destination resource

- <*width*> is an integer indicating the size, in bits, of the subset to extract.

Example:

The following example shows the expansion of a branching macro "bge" ("branch if greater or equal") used by MIPS assemblers:

```
; replaces "bge s,t,i" by the sequence :
; $at <- (s < t) ? 1 : 0
; if ($at == 0) then PC <- PC + i
#define M_BGE_1\
  (expand [\
        (build_asm "slt" "d,s,t" \
                   [(with_reg "$at") \
                    (match_arg 0) \
                    (match_arg 1)])\
        (build_asm "beq" "s,t,i" \
                   [(with_reg "$at") \
                    (with_reg "$0") \
                    (match_arg 2)]) \
        ])
```

2.8 Target-Specific Functionalities

Target-specific preprocessing of assembly files is implemented in files named *arch*.cc for each supported target system *arch*. These files must provide the implementation of the following functions:

`void prog_db::setTargetDependentInfo(void):` the main interface function which sets up the appropriate pseudo-opcode table and performs any additional initialization tasks that might be required for the current target;

`void target_emit_hook(FILE *fg):` code to be executed before the *transformed* program is output to file **fg*.

`void target_label_hook(symbolS *xsymb):` function to be executed when a user-defined label (i.e., potentially a procedure entry point) is encountered. *xsymb* points to the symbol object associated with the label.

The pseudo-opcode table provides the list of supported assembler directives and the associated actions. Entries in the table contain the name of the directive with the leading dot ('.') stripped, and the name of the function to be executed when that directive is encountered.

A generic set of directives and their associated actions is implemented by the class `generic_pseudotab`. Additional directives and actions (including replacement actions for generic directive names) are provided in target-specific derived class `target_pseudotab`. The definition and the implementation of class `target_pseudotab` must be supplied by the user in file *`target`*`.cc`.

2.9 Putting It All Together

To illustrate the specification given above, let us take a closer look at several excerpts from a preliminary SALTO description of the Philips TriMedia TM1000 architecture. The full description is given in appendix A.

2.9.1 Defining the Resources

The minimal description of the resources available on the TM1000 is quite compact. It can be extended with special-purpose registers such as PCSW, but in the first place, it consists of the general registers and the memory:

```
;; SECTION I: REGISTERS
;; ====================
;;
;; * there are 128 general registers named r0 through r127
(def_ress
  (base_name "r" 0 127)
  [(type "reg") (width 32)])

;; memory: "memI is a reserved name, but the resource corresponding
;; to the name must be explicitly defined
(def_ress (name "mem")
  [(type "memory")
   (width 32)])
```

2.9.2 Defining the Functional Units

The definition of functional units contains directly the replication level of each unit, i.e., the number of distinct operations on that functional unit type that can be issued within a single cycle. Here, the number of issue units is thus limited to five, corresponding to the five issue slots available on the TM1000:

```
;; Subsection II.1: ISSUE UNITS
```

```
;; ---------------------------
;;
(def_ress (name "issue")
  [(type "functional_unit")
    (limit 5)])      ; five issue slots

#define ISSUE (ress (name "issue") USE_AT_1)

;; Subsection II.2: COMPUTATION AND MEMORY ACCESS UNITS
;; -------------------------------------------------

;; CPP macro - all functional unit declarations follow the same format
#define FU_DECL(fu_name,replication) \
  (def_ress (name fu_name) \
    [(type "functional_unit") \
     (width 32) \
     (limit replication) \
     ])

;; dummy memory access unit for exclusions between DMEM and DMEMSPEC FUs
;; They will BOTH have to use this resource at cycle 1, but while DMEM
;; operations take just one token, DMEMSPEC ops squat both :-)
FU_DECL("mem_dummy_fu", 2)

;; integer ALU units: basic integer operations; replication level: 5
FU_DECL("alu", 5)

;; BRANCH units: branch and jump operations; replication level: 2
FU_DECL("branch", 2)

;; FCOMP unit: floating-point comparisons, status info (including clock);
;; replication level: 1
FU_DECL("fcomp", 1)
```

The case of a limited number of write-back buses is handled by requiring the use of a "`writeback_bus`" resource by all instructions that perform a register write, and by limiting the replication level of that resource to the required value:

```
;; the number of write-back buses is limited to 5 (as of TM1000) while
;; instruction latencies are variable (1, 2, 3 or 17)
;; Therefore, we need a model of that limitation...
FU_DECL("writeback_bus", 5)

;; Each register write must be represented by the reservation of a
;; write-back bus at the last cycle of the instruction.
```

```
#define WRITEBACK_BUS_AT(N) (ress (name "writeback_bus") USE_AT_CYCLE(N))
```

2.9.3 Defining the Reservation Tables

In their simplest form, reservation tables on the TM1000 depend on the format of the instruction (number and kind of its operands) and on the functional unit type for that instruction.

Therefore, the reservation tables can be first defined as `cpp` macros, then instantiated with values appropriate for each category of instructions. The following are definitions of the two most frequently used reservation table forms; information on issue slot access, functional unit type and the latency of the instruction is provided when the macros are instantiated.

```
;; there''s a couple common defs for reservation tables...

;;   ...at cycle 1
#define USE_AT_1          [(use)   (at_cycle 1)]
#define READ_AT_1         [(read)  (at_cycle 1)]

;;   ...we also need reservations at cycle n
#define USE_AT_CYCLE(N)   [(use)   (at_cycle N)]
#define WRITE_AT_CYCLE(N) [(write) (at_cycle N)]

;; Default reservation table: guard, no modifier, three addresses, and use
;; everything at cycle 1 except result register and write-back bus
;; in other words,  "IF rguard OP rsrc1 rsrc2 -> rdest"
#define THREE_ADDR_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
     [ \
       ISSUE_INFO \
       (ress (name FU) USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 1) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       (ress (match_arg 3) WRITE_AT_CYCLE(N)) \
       WRITEBACK_BUS_AT(N) \
       ])

;; operations of the form  "IF rguard OP(modifier) rsrc1 -> rdest": guard,
;; modifier, one source and one destination register
#define MOD_SRC1_DEST_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
     [ \
       ISSUE_INFO \
```

```
        (ress (name FU) USE_AT_1) \
        (ress (match_arg 0) READ_AT_1) \
        (ress (match_arg 2) READ_AT_1) \
        (ress (match_arg 3) WRITE_AT_CYCLE(N)) \
        WRITEBACK_BUS_AT(N) \
        ])
```

2.9.4 Instruction Semantics

The modeling of instruction semantics for the TM1000 is currently restricted to the delay slots and the information on the branch target location, which is known for immediate jumps, but not for indirect ones:

```
;; Right now, there is only one type of constraints: delay slots, common to
;; all branch instructions.

;; conditional _branch_ information:
#define BRANCH_SEM_INFO \
(sem [ \
        (delay_slot 3) \
        (branch -1) \
        ])

;; unconditional _jump_ information: well, it is still a branch if the guard
;; is given
#define JUMP_SEM_INFO \
(sem [ \
        (delay_slot 3) \
        (jump 1) \
        ])

;; _return_ information - the instruction terminates a procedure/function
#define RETURN_SEM_INFO \
(sem [ \
        (delay_slot 3) \
        (noreorder) \
        (return) \
        ])
```

2.9.5 Instruction Definitions

Instruction definitions associate instruction names with input format information and reservation tables. Therefore, they rely on a previous definition of operand representations. Here's the definition of input tokens for an instruction format nearly matching the actual input format of the TM1000 assembler (except for the guard which has to be placed *after* the opcode together with instruction operands):

```
;; Section II.1: COMMENTS AND SEPARATORS
;; ====================================

(comment_chars "!")

;; parentheses, dash, "greater" sign (right chevron), 'I' and 'F' as
;; separators
(def_exact "()IF->")

;; Section II.2: META-VARIABLE TOKENS
;; =================================
;;
;; register tokens are quite nice:
#define REGISTER_REGEXP "r12[0-7]\\|r1[0-1][0-9]\\|r[1-9][0-9]\\|r[0-9]"
(def_token "g" [(regex REGISTER_REGEXP)]) ; guard register
(def_token "s" [(regex REGISTER_REGEXP)]) ; first source register
(def_token "t" [(regex REGISTER_REGEXP)]) ; second source register
(def_token "d" [(regex REGISTER_REGEXP)]) ; destination register

;; sometimes r0 must be explicitly given
(def_token "0" [(regex "r0")])

;; modifier tokens: integer expressions, possibly containing identifiers
(def_token "m" [(read_exp)])
```

Finally, the instructions are defined using all the above components:

```
;; Section III: INSTRUCTION SET SPECIFICATION
;; =========================================
;;
;; Section III.1: VLIW INSTRUCTION WIDTH
;; ====================================
;;
;; five issue slots per cycle
(inst_width 5)

;; Section III.2: NATIVE INSTRUCTIONS
;; =================================
;;
;; Instructions are assumed to complete at cycle following the latency cycle
```

```
;; - there''s an EXPLICIT decode/read cycle at t=1 (0 is insn fetch...)
;; Latencies given as parameters of reservation tables take this already
;; into account.
;;
;; BTW, spaces are not significant in the format string and ARE IGNORED
;; during unparsing - this needs post-processing :-)
#define INPUT_THREE_ADDR input "IF g s t -> d"
#define INPUT_DEST input "IF g -> d"
#define INPUT_MOD input "IF g (m)"
#define INPUT_MOD_DEST input "IF g (m) -> d"
#define INPUT_MOD_SRC1 input "IF g (m) s"
#define INPUT_MOD_SRC1_DEST input "IF g (m) s -> d"
#define INPUT_MOD_SRC1_SRC2 input "IF g (m) s t"
#define INPUT_SRC1 input "IF g s"
#define INPUT_SRC1_DEST input "IF g s -> d"
#define INPUT_SRC1_SRC2 input "IF g s t"
#define INPUT_NOP input "IF g"

;; Section III.2.1: ALU OPERATIONS
;; ===============================
;;
;; most "alu" operations are three-address ops completed in one cycle;
;; the write takes place at the next cycle (t=2)
#define ALU_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   THREE_ADDR_TABLE(ISSUE,"alu",2) \
   ])

ALU_THREE_ADDR_OP("iadd") ; signed integer addition
ALU_THREE_ADDR_OP("isub") ; signed integer subtraction
;; etc. etc.

;; "alu" ops that use an immediate value, source1 and dest register
#define ALU_MOD_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_MOD_SRC1_DEST) \
   MOD_SRC1_DEST_TABLE(ISSUE,"alu",2) \
   ])

ALU_MOD_SRC1_DEST_OP("ileqi") ; signed less or equal than imm
ALU_MOD_SRC1_DEST_OP("igtri") ; signed greater than immediate
;; etc. etc.
```

Chapter 3

SALTO User Interface Specification

This chapter describes the objects and functionalities directly available to the user of the system.

3.1 Classes and Objects

The assembly module, or *program*, processed by SALTO is seen as a list of *procedures*. Each procedure is represented by means of a *control flow graph (CFG)*. A control flow graph is a directed, possibly cyclic graph whose nodes are *basic blocks*, and whose edges correspond to control transfers (jumps) resulting from branch instructions.

Each basic block consists of a list of *instructions*, *labels*, and *assembler directives*. The information attached to an instruction consists of its source line number, opcode, a list of operands, and a reservation table of all resources involved in the execution of that instruction. Internally, SALTO uses two internal directives `BEGIN_BASIC_BLOCK` and `END_BASIC_BLOCK` to mark the beginning and the end of a basic block.

Instructions reference machine resources in two mutually exclusive ways: through uses and through accesses. The former are associated with functional units and characterize the fact that a given functional unit is required at a given stage of instruction execution. The latter are associated with storage resources, such as registers and memory locations, and describe data storage and extraction.

Resource requirements for a section of code can be represented using a user-modifiable *reservation table* which can then be used in optimizations involving instruction reordering.

Finally, any kind of additional information can be attached to a SALTO object using *attributes*, which can contain any type of data.

The object classes and types provided by the interface are the following:

`class CFG` is used to represent functions as *control flow graphs*. The vertices of this graph are objects of class `BB` (see below) and the edges are labeled depending on the result of the branch instruction (`TAKEN` or `NOT_TAKEN`);

`class BB` is used to represent *basic blocks*, i.e., linear sequences of instructions;

`class INST` implements assembly instructions, labels, and directives. An instruction is not necessarily a mnemonic of the processor's instruction set: it can be a macro-instruction, a pseudo-instruction (directive), a label, or a SALTO marker (e.g., begin/end of basic block, begin/end of loop etc.);

`class OperandInfo` models instruction operands, providing a simple means of manipulating operand contents: register names, constant symbols and values, etc.;

`class SaltoAttribute` provides the implementation of the *attribute* concept.

`class reserv_table1` implements reservation tables, used in scheduling algorithms;

Additional classes used in resource management are presented in section 3.6.

3.2 SALTO-Specific Types

In addition to the classes, five data types are provided in order to represent specific properties of objects. These are:

`enum cft_type:` enumerated type indicating the nature of the control flow associated with a conditional branch. The only values are `TAKEN` and `NOT_TAKEN`, corresponding respectively to the success (branch) and the failure (do not branch) of the condition.

`enum xNode_Type:` enumerated type indicating the category to which an instruction belongs. The complete list of instruction types is given below:

Value	Description	Example
`X_ASM_TYPE`	Assembler mnemonic	`ld [%o1+4],%o2`
`X_MACRO_TYPE`	Macro equivalent to one or more mnemonics	`nop`
`X_LABEL_TYPE`	Label	`L12:`
`X_EQUAL_TYPE`	Constant declaration	`MAX = 64`
`X_PSEUDO_TYPE`	Pseudo-instruction (directive)	`.global _main`
`X_INFO_TYPE`	SALTO-generated annotation	`BEGIN_BASIC_BLOCK`

`enum dependence:` enumerated type characterizing the nature of the data hazard between two instructions. The set of possible values is:

Value	Meaning
`NONE`	no common data
`RAW`	Read-after-write hazard (flow dependence)
`WAR`	Write-after-read (anti-dependence)
`WAW`	Write-after-write (output dependence)

`enum res_ref_type`: enumerated type indicating the type of *reference* made to a resource by an instruction. The set of possible values is

Value	Meaning	Example
RES_ID_REF	basic resource	r12
CLASS_ID_REF	resource class	any global register
MULTI_REF	block of registers	%f0 as 64-bit register
ARG_REF	instruction argument	operand #3

`enum res_type`: enumerated type indicating the type of the hardware resource being accessed. The set of possible values is

Value	Meaning	Example
REGISTER_RTYPE	register	"r12"
FUNCT_UNIT_RTYPE	functional unit	"alu"
MEMORY_RTYPE	memory reference	"an_identifier"

3.3 Salto Primitives

The primitives of Salto are divided into two groups: global functions, operating at top level in the target code, and class methods, which manipulate the contents and properties of specific Salto objects.

3.3.1 Programming Conventions Used in the Interface

All indices in lists (position of CFG in the program, of a basic block in a CFG, etc.) start from 0.

Failure of a function returning a pointer is indicated by returning *NULL*.

The first and last instruction in a basic block are special internal markers and can neither be moved nor extracted.

An instruction can belong to at most one basic block. To be moved from one block to another, an instruction must be first removed from its original block, then inserted into the destination one.

3.3.2 Global Primitives

Global functions provide the means of manipulating the list of procedures appearing in a program, extracting the name of the target architecture, and finding the position of an object (CFG, basic block, instruction) in its container (program, CFG, or basic block).

`void loadFile(char *fileName)` reads and parses the assembly file *fileName*. N.B.: this function should only be used if a *new* file has to be processed. In normal operation, an input file has already been read and parsed before the user application started executing.

`char *getTargetName(void)` returns the name of target architecture as specified in the target description being used, for example `"sparc"` or `"mips"`.

`unsigned int numberOfCFG(void)` returns the number of distinct control flow graphs in the program, that is, the number of procedures.

`CFG *getCFG(unsigned int` ***pos*** `)` returns the control flow graph of the ***pos***-th procedure in the current program. ***pos*** must be a value between 0 and `numberOfCFG() - 1`. Otherwise, an error message is generated and `NULL` is returned.

`unsigned int numberOfInstructions(void)` returns the number of instructions (including labels and directives) in the program seen as a flat list of instructions. N.B.: the expansion of macros was performed beforehand, when the input program was parsed.

`INST *getInstruction(unsigned int` ***pos*** `)` returns the ***pos***-th instruction in the program seen as a flat list of instructions. ***pos*** must be a value between `0` and`numberOfInstructions()`. Otherwise, an error message is generated and `NULL` is returned.

`void removeCFG(int` ***pos*** `)` suppresses the ***pos***-th procedure from the program. ***pos*** must be a value between 0 and `numberOfCFG() - 1`. Otherwise, an error message is generated and the call has no effect.

`unsigned int getPositionInPgm(CFG *`*cfg* `)` returns the position of the specified procedure in the current program. If the procedure has not been found in the abstract representation of the program, an error message is generated.

`unsigned int getPositionInCFG(BB *`*b* `)` returns the position of the specified basic block within its enclosing procedure. If the basic block has not been found in the abstract representation of the program, an error message is generated.

`unsigned int getPositionInBB(INST *st)` returns the position of the specified instruction within its enclosing basic block. If the instruction has not been found in the abstract representation of the program, an error message is generated.

`void produceCode(FILE *`***outFile*** `)` unparses the internal representation of the current assembly program to the specified file. ***outFile*** must already be open for writing. The default value of ***outFile*** is ***stdout***.

`void producePrologue(FILE *`***outFile*** `)` unparses the internal representation of the prologue of current assembly program to the specified file. The prologue consists of all instructions (directives, data labels etc.) from the beginning of the program up to (but not including) the first CFG of the program. ***outFile*** must already be open for writing. The default value of ***outFile*** is ***stdout***.

`void produceEpilogue(FILE *`*`outFile`*`)` unparses the internal representation of the epilogue of the current assembly program to the specified file. The epilogue consists of all instructions located past the end of the last CFG of the program. ***`outFile`*** must already be open for writing. The default value of ***`outFile`*** is ***`stdout`***.

3.3.3 Control Flow Graphs

The data in *`CFG`* objects is entirely privatized: all modifications of their values are performed through the methods listed below:

`char * CFG::getName(void)` returns the name of the procedure corresponding to this CFG, i.e., the first label of its first basic block. A *`NULL`* pointer is returned if the CFG is empty.

`unsigned int CFG::numberOfBB(void)` returns the number of basic blocks in the procedure.

`BB *CFG::getBB(unsigned int `*`pos`*`)` returns the *`pos`*-th basic block of the procedure. *`pos`* must be a value between 0 and `this->numberOfBB() - 1`. Otherwise, an error message is generated and `NULL` is returned.

`void CFG::deleteBB(unsigned int `*`pos`*`)` deletes the *`pos`*-th basic block and its instructions from the control flow graph and updates the edges of the graph. Prints an error message and does not modify the graph if *`pos`* is out of bounds.

`BB *CFG::createNewBB(void)` creates a new basic block with no instructions in it.

`BB *CFG::extractBB(unsigned int `*`pos`*`)` extracts a basic block from the procedure without destroying its contents or modifying the dependences (edges) of the graph. Allows the basic block to be inserted elsewhere. *`pos`* must be a value between 0 and `this->numberOfBB() - 1`. See also methods `CFG::linkBB()` and `CFG::unlinkBB()`.

`void CFG::insertBB(unsigned int `*`pos`*`, BB `*`*b`*`)` inserts a previously extracted or newly created basic block into the control flow graph at the position specified by *`pos`*. No edges are added to the graph. *`pos`* must be a value between 0 and `this->numberOfBB() - 1`. See also methods `CFG::linkBB()` and `CFG::unlinkBB()`.

`void CFG::linkBB(BB `*`*source`*`, BB `*`*sink`*`, enum cft_type `*`t`*`)` adds an edge between basic blocks *`source`* and *sink*. The parameter t indicates whether the edge corresponds to the branch being taken (test condition satisfied, t = `TAKEN`) or not (test condition failed, t = `NOT_TAKEN`).

`void CFG::unLinkBB(BB `*`*source`*`, BB `*`*sink`*`)` suppresses the edge between basic blocks *`source`* and *`sink`*.

`void CFG::producePrologue(FILE *outFile)` writes to file *outFile* the totality of the (pseudo-)code *preceding the first basic block* of the current procedure. This may include directives, data labels, comments etc.

`void CFG::produceEpilogue(FILE *outFile)` writes to file *outFile* the (pseudo-)code *following the last basic block* of the current procedure.

`void CFG::produceCode(FILE *outFile)` writes to file *outFile* the complete code of the current procedure, including its prologue and epilogue.

3.3.4 Basic Blocks

The objects of class *BB* represent the basic blocks of the target code, that is, lists of instructions containing neither branches nor jumps, except at the end. As for class *CFG*, there is no direct access to the data of class *BB*. All accesses are made through the methods listed below:

`unsigned int BB::numberOfInstructions(void)` returns the number of instructions in the current basic block. NOTE: as macro expansion is performed beforehand, the count returned will be that corresponding to the *expanded code*.

`unsigned int BB::numberOfAsm(void)` returns the number of actual assembler mnemonics in the current basic block. NOTE: as macro expansion is performed beforehand, the count returned will be that corresponding to the *expanded code*.

`INST *BB::getInstruction(unsigned int pos)` returns the *pos*-th instruction in the current basic block. *pos* must be a value between `0` and `this -> numberOfInstructions() - 1`. If *pos* is out of bounds, `NULL` is returned and an error message is generated.

`INST *BB::getAsm(unsigned int pos)` returns the *pos*-th assembler mnemonic of the current basic block. *pos* must be a value between `0` and `this -> numberOfAsm() - 1`, otherwise `NULL` is returned and an error message is generated.

`void BB::extractInstruction(unsigned int pos)` suppresses the *pos*-th instruction from the current basic block. *pos* must be a value between `1` and `this -> numberOfInstructions() - 2`, otherwise an error message is generated and the call has no effect. *Reminder:* the first and the last instruction of the basic block are SALTO markers and cannot be removed.

`void BB::extractInstruction(INST *st)` suppresses the specified instruction from the current basic block. Instruction *st* must belong to the current basic block, otherwise an error message is generated and the call has no effect. *Reminder:* the first and the last instruction of the basic block are SALTO markers and cannot be removed.

`void BB::extractAsm(unsigned int pos)` suppresses the *pos*-th *assembly* instruction from the current basic block. *pos* must be a value between `0` and `this -> numberOfAsm() - 1`, otherwise an error mesage is issued and the call has no effect.

`void BB::insertInstruction(unsigned int pos, INST *st)` inserts a new instruction before the instruction at position *pos* in the current basic block. *pos* must be a value between `1` and `this -> numberOfInstructions() - 1`, otherwise an error message is generated and the call has no effect. The position given is that at which the inserted instruction should appear *after* the call. N.B.: an instruction can only belong to one basic block: if instructions are moved between blocks, they must first be extracted from the original block, then inserted into the destination one.

`void BB::insertAsm(unsigned int pos, INST *st)` inserts a new assembly instruction before the assembly instruction at position *pos* in the current basic block. *pos* must be a value between `0` and `this -> numberOfAsm()`, otherwise an error message is generated and the call has no effect. The position given is that at which the inserted instruction should appear in the assembly instruction list *after* the call. If the position given is `0` and the block contains no assembly instructions, or if the position given is `this -> numberOfAsm()`, the assembly instruction *st* is inserted as the last instruction of the block. N.B.: an instruction can only belong to one basic block: if instructions are moved between blocks, they must first be extracted from the original block, then inserted into the destination one.

`void BB::swapInstruction(unsigned int pos1, unsigned int pos2)` exchange the instructions located at positions *pos1* and *pos2* in the current basic block. *pos1* and *pos2* must be comprised between `1` and `this -> numberOfInstructions() - 2`.

`void BB::orderAccordingToCycles(void)` reorders the instructions of the basic block according to the schedule attributed beforehand to each instruction using calls to `INST::setCycle()`.

`void BB::addNecessaryNops(void)` insert all necessary NOP pseudo-instructions corresponding to the cycles for which no instructions are scheduled. See also `orderAccordingToCycles()` and `INST::setCycle()`.

`unsigned int BB::numberOfSuc(void)` returns the number of successors of the current basic block in its enclosing control flow graph.

`unsigned int BB::numberOfPred(void)` returns the number of predecessors of the current basic block in its enclosing control flow graph.

`BB *BB::getSuc(unsigned int pos)` returns the *pos*-th successor of the current basic block in the enclosing control flow graph. *pos* must be between `0` and `this -> numberOfSuc() - 1`, otherwise `NULL` is returned and an error message is generated.

`BB *BB::getPred(unsigned int pos):` returns the *pos*-th predecessor of the current basic block in the enclosing control flow graph. *pos* must be between `0` and `this -> numberOfPred() - 1`, otherwise `NULL` is returned and an error message is generated.

`enum cft_type BB::getSucType(unsigned int pos)` returns the type of the *pos*-th successor of the current basic block in the enclosing control flow graph. *pos* must be between `0` and `this -> numberOfSuc() - 1`, otherwise the call returns `NOT_TAKEN` and an error message is generated.

`enum cft_type BB::getPredType(unsigned int pos):` returns the type of the *pos*-th predecessor of the current basic block in the enclosing control flow graph. *pos* must be between `0` and `this -> numberOfPred() - 1`, otherwise the call returns `NOT_TAKEN` and an error message is generated.

`void BB::addSuc(BB *b, enum cft_type t)` adds a successor of the current basic block and updates the predecessor list of the basic block being added. The parameter *t* indicates whether the edge corresponds to the branch being taken (test condition satisfied, *t == TAKEN*) or not (test condition failed, *t == NOT_TAKEN*).

`void BB::addPred(BB *b, enum cft_type t)` adds a predecessor of the current basic block and updates the successor list of the basic block being added. The parameter *t* indicates whether the edge corresponds to the branch being taken (test condition satisfied, *t == TAKEN*) or not (test condition failed, *t == NOT_TAKEN*).

`void BB::notPredAnymore(unsigned int pos)` suppresses the edge between the current basic block and its *pos*-th predecessor. *pos* must be a value between `0` and `this -> numberOfPred() - 1`, otherwise an error message is generated.

`void BB::notSucAnymore(int pos)` suppresses the edge between the current basic block and its *pos*-th successor. *pos* must be a value between `0` and `this -> numberOfSuc() - 1`, otherwise an error message is generated.

`unsigned int BB::contains(INST *st)` checks whether or not instruction *st* belongs to the current basic block. Returns `0` if the instruction was not found or was a marker pseudo-instruction. A non-zero return value is the position of the instruction in the basic block.

`void BB::produceCode(FILE *fg)` writes the external representation of the current basic block to the file *fg*.

`INST *BB::firstInstruction(void)` returns the first instruction of the current basic block. It is necessarily a marker pseudo-instruction `BEGIN_BASIC_BLOCK` (type `X_INFO_TYPE`).

`INST *BB::lastInstruction(void)` returns the last instruction of the current basic block. It is necessarily a marker pseudo-instruction `END_BASIC_BLOCK` (type `X_INFO_TYPE`).

3.3.5 Instructions

The class *`INST`* implements a representation of target code instructions. As for the classes *`CFG`* and *`BB`*, all data of class `INST` objects are private and can only be manipulated using the methods listed below.

Type and property predicates

The type and several semantical propreties of an instruction can be checked by calling the following predicates:

`xNode_Type INST::getType(void)` returns the type of the current instruction (see section 3.2 above.)

`bool INST::isLabel(void)` returns `true` if the current instruction is a label.

`bool INST::isPseudo(void)` returns `true` if the current instruction is a "pseudo-instruction", i.e., an assembler directive.

`bool INST::isAsm(void)` returns `true` if the current instruction is an actual assembler instruction.

`bool INST::isBranch(void)` returns `true` if the current instruction is a conditional branch. NOTE: applies only to actual assembler instructions; otherwise, returns `false` and generates an error message.

`bool INST::isJump(void)` returns `true` if the current instruction is an unconditional jump. NOTE: applies only to actual assembler instructions; otherwise, fails with an error message.

`bool INST::isCall(void)` returns `true` if the current instruction is a subroutine call. NOTE: applies only to actual assembler instructions; otherwise, returns `false` and generates an error message.

`bool INST::isReturn(void)` returns `true` if the current instruction is a return from subroutine. NOTE: applies only to actual assembler instructions; otherwise, returns `false` and generates an error message.

`bool INST::isNop(void)` returns `true` if the current instruction is a NOP. NOTE: applies only to actual assembler instructions and macros; otherwise, returns `false` and generates an error message.

`bool INST::isCTI(void)` returns `true` if the current instruction is a control transfer instruction (branch, jump, call or return). NOTE: applies only to actual assembler instructions and macros; otherwise, returns `false` and generates an error message.

Construction and duplication of instructions

New instructions can be created using the following set of functions:

`INST *newAsm(char *`*`opcode`*`, unsigned int `*`numOps`*` = 0, ...)` returns a new assembler instruction with mnemonic *`opcode`* and *`numOps`* operands, built using the first instruction format specified for that mnemonic in the machine description file. Instruction operands are passed in the optional argument part as C++ references to class `OperandInfo` objects. A reservation table matching the operands of the instruction is also created and attached to the instruction. NOTE: each operand is passed as a separate argument.

`INST *newAsm(char *`*`opcode`*`, char *`*`format`*`, unsigned int `*`numOps`*` = 0, ...)` returns a new assembler instruction with mnemonic *`opcode`* and *`numOps`* operands, built using the specified instruction format. A matching instruction declaration must exist in the machine description file. Operands are passed in the optional argument part as C++ references to class `OperandInfo` objects. A reservation table matching the operands of the instruction is also created and attached to the instruction. NOTE: each operand is passed as a separate argument.

`INST *newLabel(char *`*`name`*`)` returns a new label with the specified name. NOTE: *`name`* should not contain the trailing colon character (‘`:`’).

`INST *newPseudo(char *`*`text`*`)` returns a new pseudo-instruction whose textual representation (including the leading dot) is *`text`*.

The duplication of an instruction is implemented through the method ‘`copy()`’:

`INST *INST::copy(void)` returns a copy (a clone) of the current instruction.

Issue cycle manipulation

The cycle at which the instruction is to be issued can be directly manipulated through the following two methods:

`int INST::getCycle(void)` extracts the cycle at which the instruction will be issued. By convention, a negative value indicates that the instruction has not been scheduled yet.

`void INST::setCycle(int `*`c`*`)` sets the cycle at which the instruction will be issued.

Name, informative annotations, and textual representation

The *`INST`* interface provides access to the textual representation of instructions and to :

`char *INST::getName(void)` called on an assembly instruction or a macro, returns the mnemonic without the arguments. On a label, returns the textual representation of its symbol, without the trailing colon. On a pseudo-instruction (assembler directive), returns the name of the directive, including the leading dot ('.').

`char *INST::getAsmInfo(void)` returns the contents of the textual information field attached to the assembler instruction or macro-operation definition matching the current instruction. Note: this method should only be called on assembler instructions and macros.

`char *INST::unparse(void)` returns the unparsed (external) representation of the current instruction irrespective of attributes attached to that instruction. The memory space for the string is allocated through a call to `new char[]` and should be released after use.

`char *INST::unparse(char *`*st*`)` stores in *st* the unparsed (external) representation of the current instruction, irrespective of attributes attached to that instruction. The size of the memory area pointed to by *st* must be sufficient to hold the unparsed text.

`void INST::produceCode(FILE *`*outFile*`)` writes the unparsed representation of the current instruction to the file *outFile*. If an attribute of type `UNPARSE_ATT` containing a pointer to a character string is attached to the instruction, only the that string is written, instead of the textual representation of the instruction. Comments specified through attributes of type `COMMENT_ATT` attached to the instruction are printed in their order of attachment after the textual representation of the instruction.

Operands

The following methods provide the means of extracting and replacing instruction operands. They should only be called on actual assembly instructions. Operand abstractions (class *OperandInfo*) are further discussed in section 3.4.)

`unsigned int INST::numberOfOperands(void)` returns the number of operands attached to the instruction.

`operand *INST::getRawOperand(unsigned int `*pos*`)` returns the low-level representation of the *pos*-th operand of the instruction. *pos* must be in the range 0..`this -> numberOfOperands() - 1`, otherwise an error message is generated and the call returns `NULL`.

`void INST::setRawOperand(unsigned int `*pos*`, operand *`*op*`)` sets the *pos*-th low-level operand of the instruction to *op*. *pos* must be in the range 0..`this -> numberOfOperands() - 1`, otherwise an error message is generated and the call has no effect.

`OperandInfo &INST::getOperand(unsigned int `*pos*`)` returns the *abstraction* of the *pos*-th operand of the instruction. *pos* must be in the range 0..`this -> numberOfOperands() - 1`, otherwise an error message is generated and the call returns a reference to an operand of type `unknownOpdT`.

`void INST::setOperand(unsigned int pos, OperandInfo &op)` sets the *pos*-th operand of the instruction from the operand abstraction *op*. *pos* must be in the range 0..`this -> numberOfOpe-rands() - 1`, otherwise an error message is generated and the call has no effect.

Resource accesses

`int INST::numberOfInput(void)` returns the number of resources *read* by the current instruction. NOTE: applies only to actual assembler instructions.

`int INST::numberOfOutput(void)` returns the number of resources *written* by the current instruction. NOTE: applies only to actual assembler instructions.

`int INST::numberOfUse(void)` returns the number of resources *used* by the current instruction. NOTE: applies only to actual assembler instructions.

`res_ref *INST::getInput(int pos)` returns the *pos*-th resource read by the current instruction; *pos* must be in the range 0..`numberOfInput() - 1`. NOTES: 1) the order of resources returned by `getInput()` does not necessarily match the chronological order in which they are accessed by the instruction; 2) this method applies only to actual assembler instructions.

`res_ref *INST::getOutput(int pos)` returns the *pos*-th resource written by the current instruction; *pos* must be in the range 0..`numberOfOutput() - 1`. NOTES: 1) the order of resources returned by `getOutput()` does not necessarily match the chronological order in which they are accessed by the instruction; 2) this method applies only to actual assembler instructions.

`res_ref *INST::getUse(int pos)` returns the *pos*-th resource used by the current instruction; *pos* must be in the range 0..`numberOfUse() - 1`. NOTE: the order of resources returned by `getUSe()` does not necessarily match the chronological order in which they are used by the instruction; 2) this method applies only to actual assembler instructions.

`void INST::setInput(int pos, res_ref *r)` updates the description of the *pos*-th resource *read* by the instruction; *pos* must be in the range 0..`numberOfInput() - 1`. NOTE: applies only to actual assembler instructions.

`void INST::setOutput(int pos, res_ref *r)` updates the description of the *pos*-th resource *written* by the instruction; *pos* must be in the range 0..`numberOfOutput() - 1`. NOTE: applies only to actual assembler instructions.

`void INST::setUse(int pos, res_ref *r)` updates the description of the *pos*-th resource *used* by the instruction; *pos* must be in the range 0..`numberOfUse() - 1`. NOTE: applies only to actual assembler instructions.

`void INST::getResUsageMode(res_ref *r, int *tab, int len)` fills the integer array *tab* of size *len* with the markers indicating the nature of references made by the current instruction to the resource *r* at each cycle of its execution. Non-zero entries in the array correspond to the cycles at which the resource is referenced by the instruction. NOTE: applies only to actual assembler instructions; otherwise, fails with an error message.

`void INST::setResUsageMode(res_ref *r, int *tab, int len)` sets the *use* information of resource **r* from the integer array **tab* of size *len* containing the markers indicating the nature of references made by the current instruction to the resource **r* at each cycle of its execution. NOTE: applies only to actual assembler instructions; otherwise, fails with an error message.

`int INST::noReorder(void)` returns `TRUE` (non-zero) if the instruction cannot be moved, e.g., if it lies in a *delay slot*. NOTE: applies only to actual assembler instructions; otherwise, fails with an error message.

`enum dependence INST::dependsOn(INST *ii, bool noCtrlFlow, bool noMem)` returns the type of the data dependence between instruction *ii* and current instruction, assuming that *ii* is executed *before* the current instruction. By default, both *noCtrlFlow* and *noMem* are *not* set (value `false`). The value returned is one of `NONE`, `RAW`, `WAW` and `WAR` (see section 3.2 above.) If the flag *noCtrlFlow* is set, `INST::dependsOn(...)` does not check whether instruction *ii* follows current instruction in the control flow (otherwise, it returns `NONE`.) If the flag *noMem* is set, no tests are made for memory dependences. N.B.: both instructions (*this* and *ii*) should belong to the same basic block.

`int INST::getDelay(INST *ii)` determines the minimum delay between current instruction and instruction **ii* that will solve all data aliasing conflicts. If the function `int updateDelay(int delay, INST *first, INST *last, enum dependence dep)` is defined, it is called with *first == this* and *last == ii* to account for the write-back by-pass, if any.

`int INST::getResDelay(INST *ii)` determines the minimum delay between current instruction and instruction **ii* that will solve all resource conflicts, assuming that there is *exactly one* instance of every functional unit. If the function `int updateDelay(int delay, INST *first, INST *last, enum dependence dep)` is defined either in user's tool, or in the target-specific module of SALTO (searched in that order), it is called with *first == this* and *last == ii* to account for the write-back bypass, if any.

3.4 Operand Abstraction

The contents of instruction operands should only be manipulated through the operand abstraction class `OperandInfo`. This class provides primitives allowing to extract and modify operand values, including register renaming and arbitrary expression substitutions.

The abstract operand contains information on the low-level Salto object representing the actual operand and, for resource operands (registers and memory), gives the type and schedule of accesses made to the operand by the instruction it is attached to. Access schedules are given wrt. the issue cycle of the instruction (0 by convention). Multi-cycle accesses must span an interval of consecutive cycles.

NOTE: expression operands are supposed to evaluate to the form <symbol> + [<symbol>] + [<signed_constant>].

3.4.1 Constructors

A family of six constructors is provided to allow for a fast and easy construction of operand abstractions:

`OperandInfo(operand *`*`op`*`, accessT `*`acc`*`=notAccessed, uint `*`from`*`=0, uint `*`to`*`=0)`
builds an operand abstraction from the low-level operand *`op`*. If present, the remaining fields (*`acc`*, *`fromCycle`*, *`toCycle`*) define the type and the schedule of the access.

`OperandInfo(char *`*`name`*`, accessT `*`acc`*`=notAccessed, uint `*`from`*`=0, uint `*`to`*`=0)`
builds the abstraction of an access to the resource *`*rsrcName`*. If present, the remaining fields (*`acc`*, *`fromCycle`*, *`toCycle`*) define the type and the schedule of the access.

`OperandInfo(SymbolS *`*`symb`*`)` builds the abstraction of a reference to a symbol whose low-level Salto representation is *`*symb`*. There is no access information, since the operand is not a resource.

`OperandInfo(int `*`cst`*`)` builds the abstraction of a reference to an integer constant *`cst`*.

`OperandInfo(double `*`fpcst`*`)` builds the abstraction of a reference to a floating-point constant *`fpcst`*.

`OperandInfo(void)` builds an empty operand abstraction, to be completed through further initializations.

3.4.2 Type Manipulation

The type of an operand abstraction can be checked/modified by calling the following predicates and property methods:

`operandT OperandInfo::getType(void)` returns the type of the operand. The type can be `unknownOpdT`, `resIdentOpdT`, `multiresIdentOpdT`, `FPconstOpdT`, `exprOpdT`, `addExprOpdT`, `upOrLowPartOpdT`, `resPlaceholderOpdT`, or `multiresPlaceholderOpdT`.

`void OperandInfo::setType(operandT `*`type`*`)` sets the type of the operand abstraction to *`type`*.

`bool OperandInfo::isResIdent(void)` returns `true` if the operand is a reference to a single resource (register or memory).

`bool OperandInfo::isMultiresIdent(void)` returns `true` if the operand is a reference to an aggregate resource (such as SPARC's double-precision pseudo-registers consisting of two adjacent `%f` registers).

`bool OperandInfo::isFPconst(void)` returns `true` if the operand is a floating-point constant.

`bool OperandInfo::isExpr(void)` returns `true` if the operand is a constant integer expression as defined above.

`bool OperandInfo::isAddExpr(void)` returns `true` if the operand is an integer expression containing an addition of symbol values or a metavariable term (such as a macro parameter).

`bool OperandInfo::isUpOrLowPart(void)` returns `true` if the operand is an MSB or LSB part of a resource's contents.

`bool OperandInfo::isResPlaceholder(void)` returns `true` if the operand is a meta-variable (such as a macro parameter) instantiated with a reference to a single resource.

`bool OperandInfo::isMultiresPlaceholder(void)` returns `true` if the operand is a meta-variable (such as a macro parameter) instantiated with a reference to an aggregate resource.

Operand contents manipulation

The contents of operand abstractions should only be manipulated using the methods given below.

`res_ref OperandInfo::getRawResource(void)` returns the pointer to the low-level resource reference associated with the operand.

`void OperandInfo::setRawResource(res_ref *rsrc)` sets the pointer to the low-level resource reference associated with the operand.

`char *OperandInfo::getName(void)` returns the name of the resource associated with the operand.

`char *OperandInfo::rename(char *newName)` renames the resource associated with the operand to *newName*. The subject must be a resource reference abstraction. The new resource must belong to the same class as the original one. On success, the value returned is the name of the original resource. On failure, the method returns `NULL`.

`unsigned int OperandInfo::substitute(char *`*`oldText`*`, char *`*`newText`*`)` substitutes *`newText`* for every non-overlapping occurrence of *`oldText`* in the (unparsed) representation of the expression corresponding to the abstraction. The textual representation produced by substituting *`newText`* for *`oldText`* is checked for syntaxical correctness, then parsed to rebuild the expression. The method fails if the operand is not an abstraction of a constant expression. The value returned is the number of substitutions performed. CATCH: make sure that the value of *`oldText`* matches the actual external representation of the expression you want to substitute.

`operand *OperandInfo::getValue(void)` returns the low-level operand representation corresponding to the abstraction.

`void OperandInfo::setValue(char *`*`resName`*`)` sets the abstraction to be a reference to resource *`resName`*.

`void OperandInfo::setValue(char *`*`multiresName`*`, unsigned int `*`offset`*`, `*`size`*`)` sets the abstraction to be a reference to an aggregate resource of base name *`multiresName`*, consisting of *`size`* elements starting at position *`offset`* in the resource vector.

`void OperandInfo::setValue(double `*`fpValue`*`)` sets the abstraction to be the floating-point constant *`fpValue`*.

The following group of methods is intended for users familiar with the internal structures of SALTO. Each '`set...`' method updates all internal fields of the abstraction, leaving it in a consistent state.

NOTE: all methods return and take C++ *references*.

`ident &OperandInfo::getResIdent(void)` returns the low-level single resource reference description associated with the abstraction.

`void OperandInfo::setValue(ident &`*`id`*`)` sets the abstraction to match low-level single resource reference *`id`*.

`multi_ident &OperandInfo::getMultiresIdent(void)` returns the low-level aggregate resource reference description associated with the abstraction.

`void OperandInfo::setValue(multi_ident &`*`multiId`*`)` sets the abstraction to match low-level aggregate resource reference *`multiId`*.

`flp_const &OperandInfo::getFPconst(void)` returns the low-level floating-point constant representation associated with the abstraction.

`void OperandInfo::setValue(flp_const &`*`flp`*`)` sets the abstraction to match low-level floating-point constant representation *`flp`*.

`exprn &OperandInfo::getExpr(void)` returns the low-level constant expression representation associated with the abstraction.

`void OperandInfo::setValue(exprn &xp)` sets the abstraction to match low-level constant expression representation ***xp***.

`add_expr &OperandInfo::getAddExpr(void)` returns the low-level additive expression representation associated with the abstraction.

`void OperandInfo::setValue(add_expr &addxp)` sets the abstraction to match low-level additive expression representation ***addxp***.

`up_or_low_part &OperandInfo::getUpOrLowPart(void)` returns the low-level MSB/LSB selector representation associated with the abstraction.

`void OperandInfo::setValue(up_or_low_part &uplow)` sets the abstraction to match low-level MSB/LSB selector representation ***uplow***.

`placeholder &OperandInfo::getResPlaceholder(void)` returns the low-level single resource meta-variable representation associated with the abstraction.

`void OperandInfo::setValue(placeholder &ph)` sets the abstraction to match low-level single resource meta-variable representation ***ph***.

`multi_placeholder &OperandInfo::getMultiresPlaceholder(void)` returns the low-level aggregate resource meta-variable representation associated with the abstraction.

`void OperandInfo::setValue(multi_placeholder&mph)` sets the abstraction to match low-level aggregate resource meta-variable representation ***mph***.

Resource access information

`accessT OperandInfo::getAccessType(void)` returns the type of the access. The type can be `notAccessed`, `readAccess`, `writeAccess`, or `useAccess`.

`void OperandInfo::setAccessType(accessT access)` sets the access type to ***access***.

`unsigned int OperandInfo::getFirstCycle(void)` returns the first cycle at which the operand resource is accessed, counted from instruction issue. The value returned is not meaningful if the access type is `notAccessed`.

`unsigned int OperandInfo::getLastCycle(void)` returns the last cycle at which the operand resource is accessed, counted from instruction issue. The value returned is not meaningful if the access type is `notAccessed`.

`void OperandInfo::setFirstCycle(unsigned int first)` sets the first cycle at which the operand resource is accessed.

`void OperandInfo::setLastCycle(unsigned int last)` sets the last cycle at which the operand resource is accessed.

3.5 Attributes

Attributes provide a means of annotating instructions and basic blocks with arbitrary information. Attributes attached to an object are structured into a list. An attribute has three properties:

- a ***value***, which is an opaque string of bytes;
- a ***size***, which is an integer representing the length of *value* in bytes,
- a ***type***, which is an integer representing the nature of the information stored in the value field; for system-defined attributes, ***type*** is a negative number.

3.5.1 Predefined Attributes

There are five predefined attribute types. The user can define additional types; user-defined types must correspond to strictly positive type codes.

`NO_ATT` is the attribute type returned by the SALTO attribute manipulation calls on failure;

`UNPARSE_ATT` modifies the behavior of `produceCode()` methods of classes `INST` and `BB`; if this attribute is present, the text produced by the unparser will be the value of the attribute, and not the external representation of actual basic block or instruction.

`CYCLE_ATT` is used by schedule management functions `INST::setCycle()`, and `INST::getCycle()` and `BB::orderAccordingToCycles()` when setting scheduling information of instructions in a basic block and reordering the instructions of a basic block according to their cycle numbers.

`COMMENT_ATT` allows to attach comments to a basic block or an instruction. When unparsing an instruction, an end-of-line comment is added. When unparsing a basic block, a number of line comments sufficient to represent the value of the attribute is placed immediately before the first instruction of the basic block.

`EXTENDED_BB_ATT`, if attached to a basic block, indicates that this block contains multiple branch instructions and/or backward jumps to the beginning of the block, but still has a single entry point (the first instruction of the block).

`INST_IL_ID_ATT` allows to attach to an instruction its ID number in an external tool attached to SALTO.

3.5.2 Attribute Management

`SaltoAttribute::SaltoAttribute(int` ***t*** `, void *`***pt*** `, int` ***size***`)` default constructor; builds an attribute of type ***t***, containing the data of size ***size*** pointed to by ***pt***;

`int SaltoAttribute::getAttributeType(void)` returns the type of the attribute;

`void SaltoAttribute::setAttributeType(int t)` sets the type of the attribute;

`void *SaltoAttribute::getAttributeData(void)` returns a pointer to the data field of the attribute;

`void SaltoAttribute::setAttributeData(void *d)` sets the data pointer of the attribute to *d*;

`int SaltoAttribute::getAttributeSize()` returns the size of the data field of the attribute;

`void SaltoAttribute::setAttributeSize(int s)` sets the size of the data field of the attribute to *s*;

`int SaltoAttribute::getTypeNode()` returns the type of the object to which the attribute is attached; possible return values are `CFGNODE`, `BBNODE` and `INSTNODE`.

`void *SaltoAttribute::getPtToSalto()` returns a pointer to the SALTO object to which the attribute is attached; this pointer must be re-cast according to the type of the attribute.

`void SaltoAttribute::setPtToSalto(void *sa)` sets the pointer to the SALTO object to which the attribute is attached;

`void SaltoAttribute::resetPtToSalto()` sets the pointer to SALTO object to `NULL`.

`void SaltoAttribute::setPtToSalto(BB &st)` sets the pointer to SALTO object to the address of basic block *&st*;

`void SaltoAttribute::setPtToSalto(INST &st)` sets the pointer to SALTO object to the address of instruction *&st*;

`void SaltoAttribute::getCFG(void)` returns the pointer to the CFG attribute is attached to; returns NULL if the attribute is attached to a SALTO object of another type.

`BB *SaltoAttribute::getBB(void)` returns the pointer to the basic block the attribute is attached to; returns NULL if the attribute is attached to a SALTO object of another type.

`INST *SaltoAttribute::getINST(void)` returns the pointer to the instruction the attribute is attached to; returns NULL if the attribute is attached to a SALTO object of another type.

`SaltoAttribute *copy(void)` copy an attribute; the data field is *not* copied.

3.5.3 Attribute Usage

The following are methods for attribute manipulation, available on objects of classes *CFG*, *BB*, and *INST*.

N.B: since these methods are applicable to objects of several classes, no class prefix is given in the prototypes listed.

`int numberOfAttributes(void)` returns the number of attributes currently attached to the object;

`int numberOfAttributes(int` ***type*** `)` returns the number of attributes of type ***type*** already attached to the object;

`SaltoAttribute *getAttribute(int` ***pos*** `)` returns the ***pos***-th attribute of the current object;

`SaltoAttribute *getAttribute(int` ***pos*** `, int` ***type*** `)` returns the ***pos***-th attribute of type
type attached to the current object;

`void setAttribute(int` ***pos*** `, SaltoAttribute *`***att*** `)` sets the ***pos***-th attribute of the current object to ***att***; prints an error message and returns without side effects if no such attribute exists;

`void setAttribute(int` ***pos*** `, int` ***type*** `, SaltoAttribute *`***att*** `)` sets the ***pos***-th attribute of type ***type*** attached to the object to ***att***; prints an error message and returns without side effects if no such attribute exists;

`int attributeType(int` ***pos*** `)` returns the type of the ***pos***-th attribute attached to the object; returns `NO_ATT` if no such attribute exists;

`void *attributeValue(int` ***pos*** `)` returns the pointer to the value of ***pos***-th attribute of the object; returns NULL if no such attribute exists;

`void *attributeValue(int` ***pos*** `, int` ***type*** `)` returns the value of the ***pos***-th attribute of type ***type*** attached to the object; returns NULL if no such attribute exists;

`void addAttribute(int` ***type*** `, void *`***a*** `, int` ***size*** `)` adds an attribute of type ***type*** and size ***size***, and containing the data pointed to by ***a***, at end of attribute list of the current object; no copy of data is made;

`void addAttribute(int` ***type*** `)` adds an attribute of type ***type*** and containing no data;

`void addAttribute(void *`***a*** `, int` ***size*** `)` adds a type-less attribute of size ***size*** containing the data pointed to by ***a***, at end of attribute list of the current object; no copy of data is made;

`void addAttribute(Attribute *att)` adds a complete attribute pointed to by *att* at the end of the attribute list of the object;

`void *deleteAttribute(uint pos)` removes the *pos*-th attribute of the object.

3.6 Reservation Table Management

Resource information accessible to the user covers only resource references: the user cannot change the properties of the hardware resource itself; only resource *references* made by instructions and represented in reservation tables can be accessed and modified.

The accesses to resources attached to operands can be modified using the operand abstraction. However, to modify resources which do not appear explicitly in the assembly code, it is necessary to manipulate the contents of reservation tables.

In the following, we present the *current state* of the resource interface, which is undergoing deep restructuring aiming at a more systematic and orthogonal organization.

3.6.1 Resource Descriptions

Resource descriptions in SALTO are maintained as a global resource database, searchable by resource names and ID numbers. Each resource has a unique ID number, a type, a name, and at most two aliases. The global resource database provides translations from resource names to IDs and database entries.

Resource type can be any of `UNKNOWN_RTYPE`, `REGISTER_RTYPE`, `FUNCT_UNIT_RTYPE`, or `MEMORY_RTYPE`.

Resource descriptions are implemented through the class `rdb_res_entry` providing the following methods:

`char *rdb_res_entry::name(int which = 0)` returns the name of the resource. When non-default values of *which* (1 and 2) are given, returns respectively the first and the second alias of the resource.

`int rdb_res_entry::getType(void)` returns the type of the resource (see above.)

`int rdb_res_entry::get_res_limit(void)` returns the replication level of the resource.

To access the contents of the global resource database, it must first be extracted from SALTO's database server by the following statement:

```
ResourceDataBase &rdb = xxx_server -> GetResT() ;
```

Once the resource database is extracted, individual resource entries can be searched by name and by resource ID numbers:

`ResId_T ResourceDataBase::get_res_id(char *name)` returns the ID of the resource *name*.

`char *ResourceDataBase::get_res_name(ResId_T id)` returns the name of the resource bearing ID number *id*.

`rdb_res_entry *ResourceDataBase::get_res(ResId_T id)` returns the resource entry matching the *id* given.

`int ResourceDataBase::get_res_limit(char *name)` returns the replication level of resource *name*.

3.6.2 Resource References

Resource references contain information on resource accesses made during the execution of an instruction. A resource reference represents *exactly one* resource access: if the same resource is accessed several times during the execution of an instruction, each access will be represented by a separate resource reference object.

Resource references are implemented in class `res_ref`, further specialized into `res_ref_id` (basic resource references) and `multi_ref` (multi-register references). The methods available to the user are:

`enum res_ref_type res_ref::get_ref_type(void)` returns the type of the reference.

`ResId_T res_ref::get_res_id(void)` returns the ID of the resource accessed by current reference.

`bool res_ref::same_res_as(res_ref *otherRef)` returns `true` if current reference and *otherRef* correspond to accesses to the same resource.

`int res_ref::get_limit(void)` returns the replication level of the resource being accessed.

`bool res_ref::norename(void)` returns true if the resource cannot be renamed.

`bool multi_ref::same_multires_as(res_ref *otherRef)` returns true if the current reference accesses the same multi-register as *otherRef*.

`res_ref *multi_ref::get_base_res(void)` returns the base register of the current multi-resource (i.e., the resource used as reference point in determining the composition of the multi-register, see section 2.5).

`int multi_ref::get_size(void)` returns the number of elements constituting the multi-register being referenced.

`int multi_ref::get_index(void)` returns the offset of the first element of a multi-register wrt. its base register (see above.)

3.6.3 Reservation Table Entries

A reservation table entry indicates the nature and the schedule of a resource access: the information it contains specifies what operation is performed on the resource (read, write, use) and when (first and last cycle of the access). Resources which are accessed several times when executing an instruction give raise to as many reservation table entries as there are distinct accesses being made to the resource.

A reservation table entry provides the following functionalities:

`res_ref *reserv_entry::get_res(void)` returns the resource reference it describes

`void reserv_entry::set_res(res_ref *`*`newRef`*`)` sets the resource reference to *`newRef`*.

`enum access_mode reserv_entry::get_access_mode(void)` returns the mode of access made to the resource referenced by the current entry.

`void reserv_entry::set_access_mode(enum access_mode `*`newMode`*`)` changes the access mode of the current entry to *`newMode`*.

`int &reserv_entry::from_cycle(void)` returns a reference to the integer representing the first cycle of the current access.

`int &reserv_entry::to_cycle(void)` returns a reference to the integer representing the last cycle of the current access.

3.6.4 Reservation Tables

Reservation tables are use to represent resource accesses within a common time frame, allowing to check for resource access conflicts over that time frame. A reservation table is attached to each assembly instruction, but the user can also set up reservation tables spanning several instructions. Reservation tables are used in instruction scheduling to ensure the feasibility of a given instruction schedule.

Reservation tables are implemented in the class `reserv_table1` as lists of reservation table entries and can be manipulated using the following methods of that class:

`int get_size(void)` returns the number of entries in the reservation table.

`void insert_res(res_ref *`*`ref`*`, enum access_type `*`acc`*`, int `*`fr`*`, int `*`to`*`)` adds an entry which corresponds to the reference *`ref`* made in mode *`acc`* between cycles *`fr`* and *`to`*, inclusive.

`void delete_res(int `*`pos`*`)` removes the *`pos`*-th entry in the current reservation table.

`reserv_entry* get_entry(int `*`pos`*`)` returns the *`pos`*-th entry of the current reservation table.

reserv_entry* get_entry(res_ref *ref , enum access_mode acc) returns the entry of the current reservation table in which the resource referenced by ***ref*** is accessed in mode ***acc***.

reserv_entry* get_entry(ResId_T id , enum access_mode access) returns the entry of the current reservation table in which the resource with ID ***id*** is accessed in mode ***access***.

3.6.5 Usage Examples

To illustrate the operation of the current resource interface, let us have a closer look at two short examples: low-level dependence checking, and counting of accesses performed by an instruction.

Flow dependence checking

The following is an excerpt from the implementation of class `INST`. The method `INST::isRAW(INST *source)` is used internally to check if the current instruction may depend on instruction ***source***. For read access made by the current instruction (***this***), the method searches for a write access to the same resource performed in the instruction supplied as argument.

```
/*
 * Does instruction -this- have a RAW hazard with -ii- ?
 * We check if -this- reads a resource written by -ii-.
 */
bool
INST::isRAW(INST *ii, bool ignoreMem)
{
  int i, j, rRid, rWid;
  int nOfInput, nOfOutput;
  res_ref *rread, *rwrite;

  nOfInput = numberOfInput();
  for(i=0; i < nOfInput; i++) {
    rread = getInput(i); // current insn reads rread
    // if 'rread' is a memory reference and we ignore the mem, let's
    // just skip it
    if (ignoreMem
          &&
          (rdb . get_res(rread → get_res_id()) → getType() == MEMORY_RTYPE))
       continue;
    rRid = rread → get_res_id();
```

```
    nOfOutput = ii → numberOfOutput();
    for(j=0; j < nOfOutput; j++) { // is it written in instruction 'ii'?
      rwrite = ii → getOutput(j);
      rWid = rwrite → get_res_id();
      if (rWid == rRid)
        return true; // if so, there's indeed a RAW hazard between 'ii' and 'this'
    }
  }
  return false;
}
```

Number of resource accesses

The code below is again taken from the implementation of class `INST`. The method `INST::numberOfx(...)` implements the counting of resource accesses made by an instruction in the access mode specified as argument.

```
/*
 * Return the number of resources read, written, and used by the
 * instruction. The same resource will be counted as many times as it is
 * accessed by the instruction.
 */
int
INST::numberOfx(enum access_mode acces)
{
  reserv_table1 *reserv;
  reserv_entry *entry;
  int i, k;

  if (!isAsm()) {
    saltoWarn("*** INST::numberOfx():  insn %#lx is not an ASM operation!\n",
              (unsigned long) this);
    return -1;
  }
  k = 0;
  reserv = ((xAsm *)this) → get_reser();
  for(i = 0; i < reserv → get_size(); i++) {
    entry = reserv → get_entry(i);
    if (entry → get_access_mode() == acces) k++;
  }
  return k;
}
```

Chapter 4

Application Examples

To illustrate SALTO usage, this chapter presents two programs: a basic block execution count and local scheduler. These examples give an idea of the range of tools that can be implemented using SALTO, but are not representative of state-of-the-art profiling or scheduling techniques.

4.1 Instrumentation

The following is a small example of assembly code instrumentation. The goal of this piece of code is to instrument basic blocks to obtain a basic block invocation count. SALTO adds a comment before each basic block using the `COMMENT_ATT` attribute and adds instrumentation code at the beginning of each block. The code added saves some register values before calling the function `bbcount` which performs the counting proper. This function may be written in a high-level language. In our example, it is written in C. The function is linked with the generated assembly code to produce an executable.

The function called for each block is

```
#include "salto.h"

void add_code(BB *bb, int cpt) {
  INST *nop;
  char instcode[STR_MAX];

  nop = bb → newNOP();
  sprintf(ch, "\t save register values \n"
          "\t mov %d,%%o0\n"
          "\t call _bbcount \n"
          "\t restore values \n", cpt);
  nop →addAttribute(UNPARSE_ATT, strdup(ch), strlen(ch));
  bb → insertAsm(0, nop);
}
```

```
void Salto_hook() {
   CFG *proc;
   BB *bb;
   int i, j, ncfg, nbb, cpt=0;
   char ch[20];

   ncfg = numberOfCFG();
   for(i=0; i < ncfg; i++) { // for each procedure
      proc = getCFG(i);
      nbb = proc → numberOfBB();
      for(j=0; j < nbb; j++) { // for each basic block
         bb = proc → getBB(j);
         sprintf(ch, "bb %d", ++cpt); // comment to add to a BB
         bb → addAttribute(COMMENT_ATT, strdup(ch), strlen(ch));
         add_code(bb, cpt);
      }
   }
   produceCode(stdout); // Finished, now dumps instrumented code on stdout
}
```

4.2 Local Reordering

As an example of local reordering we have implemented a list scheduling algorithm using SALTO. The main function is `reorder`, which builds the dependence matrix for the instructions of the current block: `dep[i][j]` equals to 1 if instruction number i depends on instruction number j. The main loop computes the scheduling cycle for each instruction until a branch is seen: `verify_predecessors` checks if all instructions that have a data dependence have already been scheduled. `earliest_cycle` then computes the delays before all previous results are obtained. `IsConflict` is used to detect resource conflicts. The branch instructions, if they exist, and the delay slot are processed afterwards. The blocks are effectively reordered by `orderAccordingToCycles` and NOPs are added if necessary by `addNecessaryNops`.

```
#include "salto.h"

int verify_predecessors(int verif, int **dep, INST **inst) {
   for (int i=0; i<verif; i++)
      if ( (dep[verif][i]) && (inst[i]→getCycle() < 0) ) return 0;
   return 1;
}

int earliest_cycle(int s, int **dep, INST **inst) {
   int i, z, max = 0;
   for (i=0; i<s; i++)
```

```
      if (dep[s][i]) {
         z = inst[i] → getCycle() + inst[s] → getDelay(inst[i]);
         if (z>max) max = z;
      }
   return max;
}

void build_dep_matrix(int **dep, INST **inst, int n) {
   for (int i=0; i<n; i++) {
      dep[i][i]=0;
      for (int j=0; j<i; j++)
         dep[i][j] = ( inst[i]→IsDep(inst[j]) ≠ 0 );
   }
}

INST **instr;
int **dep;

void reorder(BB *bb) {
   int i, to_be_scheduled, cycle_min, nasm, offset, branch_seen,brindex,last_cycle;
   TRES *res_table = new TRES; // need a reservation table

   nasm = bb → numberOfAsm();
   instr = new (INST *)[nasm]; // to avoid calling getAsm each time we need it
   for (i=0; i<nasm; i++) instr[i] = bb → getAsm(i+1);

   build_dep_matrix(dep, instr, nasm); // build the dependence matrix
   branch_seen = last_cycle = 0;

   to_be_scheduled = nasm; // number of instructions to be scheduled
   while (to_be_scheduled && !branch_seen) { // before a branch instruction
      for (i=0; i < nasm; i++) {
         if (instr[i] → isCTI()) {
            brindex = i;
            branch_seen = 1;
            break;
         }
         // Is this instruction already scheduled ?
         if ( (instr[i]→getCycle()) < 0 ) {
            // Are all the predecessors scheduled ?
            if (verify_predecessors(i, dep, instr)) {
               // Wait for data dependencies to be resolved
               cycle_min = earliest_cycle(i, dep, instr);
               // Now wait for resources to be available...
               offset = 0;
               while (res_table→IsConflict(instr[i],cycle_min+offset)) offset++;
               // Mark resources occupancy into reservation table
```

```
          res_table → markRes(instr[i], cycle_min + offset);
          if (cycle_min + offset > last_cycle) last_cycle = cycle_min+offset;
          // Specify the cycle
          instr[i] → setCycle(cycle_min + offset);
          to_be_scheduled--;
          break;
        }
      }
    }
  }
  // The case of the branch instruction
  if (branch_seen) {
    instr[brindex] → setCycle(last_cycle + 1);
    to_be_scheduled--;
  }
  // The delay slot instruction, if any
  if (to_be_scheduled) instr[nasm-1] → setCycle(last_cycle + 2);

  // Reorder according to values specified by setCycle()
  bb → orderAccordingToCycles();
  bb → addNecessaryNops();
  delete res_table;
  delete instr;
}
```

4.3 Local Label Renaming

Label renaming is typically needed when replicating code. The modifications must be applied at two distinct types of locations: in label declarations, and in the references to the labels in instructions.

To rename label references, we have to modify *expressions* appearing in instruction arguments. This is performed using the expression substitution primitive on operand abstractions. Labels themselves are replaced with new labels, built from scratch using the new name.

The application code in the example below performs a brute-force, substring-based renaming of all labels and symbols appearing in the code sections of the assembly file. However, it is presented for illustrative purposes and does not handle the directives and labels which are located in sections other than `"text"`, thus potentially introducing inconsistencies between declarations and uses of data labels.

```
#include "salto.h"

// replace all non-overlapping occurrences of oldTxt with newTxt in
// src, leaving hte result in dst. NO CHECKS FOR OVERFLOWS ARE MADE.
```

```
// NOTE: the argument order follows the conventions of str...() functions.

unsigned int
strSubstitute(char *dst, char *src, char *newTxt, char
                                    *oldTxt)
{
  unsigned int res = 0 ;
  char *occurrence ;

  while ((occurrence = strstr(src, oldTxt))) {
    strncpy(dst, src, occurrence - src) ; // leading common text
    dst += occurrence - src ;
    strcpy(dst, newTxt) ; // new text where old one should be
    dst += strlen(newTxt) ;
    src = occurrence + strlen(oldTxt) ; // new begin of common txt
    res++ ;
  }
  strcpy(dst, src) ; // final step: txt past last occurrence

  return res ; // number of substitutions performed
}

char *oldText, *newText;


void usage(void)
{
  fprintf(stderr,
          "Simple renaming module.  Module-specific parameters (after '--'):"
          "\n\n"
          " -h print this help message and exit\n"
          " -from <word> label (sub-)string to be replaced\n"
          " -to <word> label (sub-)string to use instead\n\n"
          "Both '-from' and '-to' arguments must be given.\n") ;
}


void Salto_init_hook(int argc, char **argv)
{
  int i;
  oldText = NULL ;
  newText = NULL ;

  for (i = 0; i < argc - 1; i++) { // we leave out the last arg
                                    // (should be a value, not an option)
    if (!strcmp(argv[i], "-from")) // (sub-)string to be replaced
      oldText = argv[i + 1] ; // it necessarily exists...
```

```
    if (!strcmp(argv[i], "-to")) // new value to be used in renaming
      newText = argv[i + 1] ; // it necessarily exists...

    if (!strcmp(argv[i], "-h")) {
      usage() ; // print help message and exit
      exit (0) ; // successfully
    }
  }

  if (!strcmp(argv[i], "-h")) { // help asked for ==> exit successfully
    usage() ;
    exit (0) ;
  }

  if ((!oldText)
      ||
      (!newText)) { // help badly needed ==> fail
    usage() ;
    exit (1) ;
  }

  // here, things go ahead normally
}

// user application entry point... Either linked dynamically
// (default), or statically (through "make static")

void Salto_hook(void)
{
  BB *bb;
  CFG *cfg;
  INST *insn, *label, *directive;
  unsigned int ncfg, nbb, ninsts, nopds, i, j, k, l;
  int substitutions;

  ncfg = numberOfCFG() ; // number of CFG in program
  if (!ncfg) return ; // nothing to do if no procedures found

  for (i = 0; i < ncfg; i++) { // FOR ALL PROCEDURES
    cfg = getCFG(i) ; // get current CFG

    nbb = cfg -> numberOfBB() ; // number of basic blocks

    for (j = 0; j < nbb; j++) { // FOR ALL BASIC BLOCKS
      bb = cfg -> getBB(j) ; // get current BB
```

```
ninsts = bb → numberOfInstructions(); // number of insns in BB

for (k = 1; k < ninsts - 1; k++) { // FOR ALL INSTRUCTIONS...
                                    // NOTE: first and last insn are
                                    // reserved oned and shouldn't be
                                    // scanned, hence the range 1..n-2

  insn = bb → getInstruction(k) ; // get current instruction

  if (insn → isAsm()) {

    // for ASM insns, replace name substring in operands by
    // substituting all non-overlapping occurrences of oldText with
    // newText. The replacement is performed on operand abstractions.

    nopds = insn → numberOfOperands() ;

    // scan all operands of the instruction
    for (l = 0; l < nopds; l++) {
      OperandInfo &op = insn → getOperand(l) ;

      if (op . isExpr()) { // only for expressions
        // substitute the strings; retvalue is number of substits
        substitutions = op . substitute(oldText, newText) ;
        if (substitutions) {
          insn → setOperand(l, op) ; // make it effective
          fprintf(stderr,
                  "Operand substitution succeeded @"
                  " CFG/BB/INST/OPD=%d/%d/%d/%d...\n",
                  i,j,k,l) ;
        }
      }
    }
  }
  else if (insn → isLabel()) {

    // for labels, replace the label by a new one, whose name
    // results from the substitution of oldText with newText...

    char *oldLabelName, newLabelName[80];

    oldLabelName = insn → getName() ; // get name of label

    // replace all non-overlapping occurrences of oldText with
    // newText. Note the the str...() argument ordering.

    substitutions =
```

```
            strSubstitute(newLabelName, oldLabelName, newText, oldText) ;

        // if at least one substitution succeeded
        if (substitutions) {
            label = newLabel(newLabelName) ; // make a new label
            bb → insertInstruction(k, label) ; // insert in place
                                        // of original insn at pos k (= old label).
                                        // The old label gets shifted by 1 position
            bb → removeInstruction(k + 1) ; // remove the old label
            fprintf(stderr,
                    "Label replacement succeeded @"
                    " CFG/BB/INST = %d/%d/%d...\n",
                    i, j, k) ;
        } // if (substitutions)
      } // if (insn → isLabel())

      else if (insn → isPseudo()) {

        // for directives, we have to go a little bit into Salto
        // internals (note the type cast :-)

        char *oldDirective, newDirective[256] ;

        // get the full text of the directive
        oldDirective = ((xPseudo *) insn) → getText() ;

        // substitute strings
        substitutions =
            strSubstitute(newDirective, oldDirective, newText, oldText) ;

        // process if successful
        if (substitutions) {
            directive = newPseudo(newDirective) ; // make the new one
            bb → insertInstruction(k, directive) ; // insert in BB
            bb → removeInstruction(k + 1) ; // remove the old label
            fprintf(stderr,
                    "Directive replacement succeeded @"
                    " CFG/BB/INST = %d/%d/%d...\n",
                    i, j, k) ;
        } // if (substitutions)
      } // if (insn → isPseudo())
    } // for all insns
  } // for all basic blocks
} // for all procedures

// generate the modified code...
produceCode(stdout) ;
```

}

Bibliography

[1] A. Aiken and A. Nicolau. Perfect pipelining : a new loop parallelization technique. In *Lecture Note In Computer Science*, pages 221–235, 1988.

[2] Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. Software pipelining. *ACM Computing Surveys*, 27:367–432, September 1995.

[3] Thomas Ball and James R Larus. Optimally profiling and tracing programs. In *ACM Transactions on Programming Languages and Systems*, volume 16, pages 1319–1360, July 1994.

[4] F. Bodin, F. Charot, and C. Wagner. Overview of a high-performance programmable pipeline architecture. In *ACM Supercomputing*, 1988.

[5] François Bodin. *Optimisation de microcode pour une architecture horizontale et synchrone, Étude et mise en œuvre d'un compilateur*. PhD thesis, Université de Rennes 1, June 1989.

[6] François Bodin, Gwendal Le Fol, and Frédéric Raimbault. OCO : manuel de l'utilisateur (version préliminaire). Technical Report 930, Irisa, May 1995.

[7] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. The MARION system for retargetable instruction scheduling. In *Programming Languages and Systems*, 1991.

[8] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. Integrating register allocation and instruction scheduling for RISCs. In 4^{th} *International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[9] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994.

[10] Lucile Cognard. *Modélisation du comportement dynamique des processeurs pour la génération automatique de réordonnanceurs de code*. PhD thesis, Université d'Orléans, 1995.

[11] LSI Logic Corporation. *SPARC Architecture Manual (Version 7)*, 1990.

[12] DEC. *ATOM User Manual*, March 1994.

[13] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Languages and Compilers for Parallel Comuting*, pages 213–229, 1989.

[14] Christine Eisenbeis, William Jalby, and Alain Lichnewski. Compiler techniques for optimizing memory and register usage on the Cray 2. In *International Journal of High Speed Computing*, June 1990.

[15] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1985.

[16] Bodin F., Beckman P., Gannon D., and Srinivas J.G.S. Sage++: A class library for building fortran and c++ restructuring tools. In *Object-Oriented Numerics Conference*, April 1994.

[17] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. In *IEEE Transactions on Computers*, pages 478–490, July 1981.

[18] Franco Gasperoni. *Scheduling for horizontal systems : the VLIW paradigm in perspective*. PhD thesis, New-York University, 1991.

[19] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In 27^{th} *Annual International Symposium on Microarchitecture*, pages 85–94, 1994.

[20] Stanford Compiler Group. SUIF compiler system. Web pages available from `http://suif.stanford.edu`.

[21] H.Emmelmann, F-W.Schroeer, and R.Landwehr. Beg - a generator for efficient back ends. In *SIGPLAN Conference on Programming Language Design and Implementation*, volume 24, July 1989.

[22] Gordon Irlam. Spa package. Electronically available at `ftp://chook.cs.adelaide.edu.au/pub/sparc/spa-1.0.tar.gz`, 1991.

[23] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328. ACM SIGPLAN, June 1988.

[24] Philips Electronics. TriMedia TM1000 Preliminary Data Book. Philips Electronics North America Corp., Sunnyvale, CA, USA, March 1997. Electronically available from `http://www.trimedia.philips.com`

[25] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In 27^{th} *ternational Symposium on Microarchitecture*, pages 63–74, December 1994.

[26] J. Ruttenberg, G. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Sigplan Conference on Programming Language Design and Implementation*, May 1996.

[27] Michael D. Smith. Tracing with `pixie`. Technical report, Harvard university, 1991.

[28] Amitabh Srivastava and Alan Eustace. `Atom`: A system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[29] R. M. Stallman. *Using and porting GNU CC.* Free Software Foundation, Jun 1993.

[30] Richard Uhlig and Trevor Mudge. Trace-driven memory simulation : A survey. Technical report, University of Michigan, 1995.

[31] Daniel Windheiser. *Optimisation de la localité de données et du parallélisme à grain fin.* PhD thesis, Université de Rennes 1, May 1992.

Appendix A

Prototype description of the Philips TriMedia TM1000 architecture

A.1 Definition of resources

```
;;-----------------------------------------------------------------------------
;;  Philips TriMedia-1000 processor description (restricted to the DSPCPU)
;;
;;  RESOURCES section: declaration of hardware resources
;;
;;  Zbigniew CHAMSKI <{zchamski,manche}@irisa.fr>
;;-----------------------------------------------------------------------------

;; SECTION I: REGISTERS
;; ====================
;;
;; * there are 128 general registers named r0 through r127
(def_ress
  (base_name "r" 0 127)
  [(type "reg") (width 32)])

;; memory: the name is reserved, but the resource corresponging to the name
;; is NOT defined
(def_ress (name "mem")
  [(type "memory")
   (width 32)])

(def_ress (name "PC")
  [(type "reg") (width 32)])

;; SECTION II: FUNCTIONAL UNITS
;; ===========================
```

```
;;
;; * there are twelve (12!) computation/load/store functional units
;;
;; * there are five (5) issue slots (units), restricted wrt. FU''s that can be
;;    handled by each issue slot (see issue slot section below)
;;
;; * the FU list with issue constraints is given below (N.B.: latencies
;;   given already include the bypass of one cycle on READ-after-WRITE.)
;;
;; -----------------------------------------------------------------------
;; |  FU name   | pipe depth    |    latency / delay    | issue slots   |
;; =======================================================================
;; |  alu       |       1       |       latency =  1    |   1 2 3 4 5   |
;; |  branch    |       4 (N/A) |       delay   =  3    |     2 3 4     |
;; |  const     |       1       |       latency =  1    |   1 2 3 4 5   |
;; |  dmem      |       3       |       latency =  3    |         4 5   |
;; |  dmemspec  |       1       |       latency =  3    |           5   |
;; |  dspalu    |       2       |       latency =  2    |   1   3       |
;; |  dspmul    |       3       |       latency =  3    |     2 3       |
;; |  falu      |       3       |       latency =  3    |   1     4     |
;; |  fcomp     |       1       |       latency =  1    |       3       |
;; |  ftough    |       2 (16+1)|       latency = 17    |     2         |
;; |  ifmul     |       3       |       latency =  3    |     2 3       |
;; |  shifter   |       1       |       latency =  1    |   1 2         |
;; -----------------------------------------------------------------------

;; there''s a couple common defs for reservations...

;;   ...at cycle 1
#define USE_AT_1          [(use)   (at_cycle 1)]
#define READ_AT_1         [(read)  (at_cycle 1)]

;;   ...we also need reservations at cycle n
#define USE_AT_CYCLE(N)   [(use)   (at_cycle N)]
#define WRITE_AT_CYCLE(N) [(write) (at_cycle N)]

;; Subsection II.1: ISSUE UNITS
;; ----------------------------
;;
(def_ress (name "issue")
  [(type "functional_unit")
   (limit 5)])      ; five issue slots

#define ISSUE (ress (name "issue") USE_AT_1)

;; Subsection II.2: COMPUTATION AND MEMORY ACCESS UNITS
;; ----------------------------------------------------
```

```
;; CPP macro - all declarations but one ("mem_dummy_fu") follow the same
;; format
#define FU_DECL(fu_name,replication) \
  (def_ress (name fu_name) \
    [(type "functional_unit") \
     (width 32) \
     (limit replication) \
     ])

;; dummy memory access unit for exclusions between DMEM and DMEMSPEC FUs
;; They will BOTH have to use this resource at cycle 1, but while DMEM
;; operations take just one token, DMEMSPEC ops squat both :-)
FU_DECL("mem_dummy_fu", 2)

;; integer ALU units: basic integer operations; replication level: 5
FU_DECL("alu", 5)

;; BRANCH units: branch and jump operations; replication level: 3
FU_DECL("branch", 3)

;; CONST units: loading immediate values into a register; replication level:
;; 5
FU_DECL("const", 5)

;; DMEM units: memory accesses; replication level: 2
FU_DECL("dmem", 2)

;; DMEMSPEC unit: cache management; replication level: 1
FU_DECL("dmemspec", 1)

;; DSPALU units: quick and efficient arithmetical ops on bytes and
;; half-words; replication level: 2
FU_DECL("dspalu", 2)

;; DSPMUL unit: dot-products, parallel multiplies etc. on bytes and
;; half-words; replication level: 2
FU_DECL("dspmul", 2)

;; FALU unit: floating-point adds/subs/valabses, conversions etc.;
;; replication level: 2
FU_DECL("falu", 2)

;; FCOMP unit: floating-point comparisons, status info (including clock);
;; replication level: 1
FU_DECL("fcomp", 1)
```

```
;; FTOUGH unit: "hard" computations - floating-point divisions and SQRTs;
;; replication level:
FU_DECL("ftough", 1)

;; IFMUL unit: full multiplier - floating-point and integer multiplies;
;; replication level: 2
FU_DECL("ifmul", 2)

;; SHIFTER unit: rotations and shifts; replication level: 2
FU_DECL("shifter", 2)

;; the number of write-back buses is limited to 5 (as of TM1000) while
;; instruction latencies are variable (1,2,3 or 17)
;; Therefore, we need a model of that limitation...
FU_DECL("writeback_bus", 5)

;; Each register write must be represented by the reservation of a
;; write-back bus at the last cycle of the instruction.
#define WRITEBACK_BUS_AT(N) (ress (name "writeback_bus") USE_AT_CYCLE(N))

;; Section III: RESERVATION TABLES
;; ===============================

;; Reservation tables on the TM1000 come in four durations: 1, 2, 3 or 17
;; cycles and with four types of pipelining: none, 2 stages, 3 stages, and
;; FTOUGH (16 cycles non-pipelined computation followed by 1 cycle
;; write-back).  In practice, there is one table per "non-issue" functional
;; unit.  At the time being we assume that the instructions are represented
;; in canonical prefix format, with guard, modifier, source1, source2 and
;; dest as operands 0 through 4 - someone has to do the corresponding
;; preprocessing job.

;; default reservation table: guard, no modifier, three addresses, and use
;; everything at cycle 1 except result register and write-back bus
;; in other words,  "IF rguard OP rsrc1 rsrc2 -> rdest"
;; #define THREE_ADDR_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 1) READ_AT_1) ; 1st opd (modifier skipped)
;;         (ress (match_arg 2) READ_AT_1) ; 2nd operand
;;         (ress (match_arg 3) WRITE_AT_CYCLE(N)) ; destination register
;;         WRITEBACK_BUS_AT(N) ; there''s a write...
;;         ])
#define THREE_ADDR_TABLE(ISSUE_INFO,FU,N) \
```

```
(reser_table \
     [ \
       ISSUE_INFO \
       (ress (name FU) USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 1) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       (ress (match_arg 3) WRITE_AT_CYCLE(N)) \
       WRITEBACK_BUS_AT(N) \
       ])

;; operations of the form  "IF rguard OP rsrc1 -> dest": guard, one source and
;; one destination address, no modifier
;; #define SRC1_DEST_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 1) READ_AT_1) ; 1st opd (modifier skipped)
;;         (ress (match_arg 2) WRITE_AT_CYCLE(N)) ; destination register
;;         WRITEBACK_BUS_AT(N) ; there''s a write...
;;         ])
#define SRC1_DEST_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
     [ \
       ISSUE_INFO \
       (ress (name FU) USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 1) READ_AT_1) \
       (ress (match_arg 2) WRITE_AT_CYCLE(N)) \
       WRITEBACK_BUS_AT(N) \
       ])

;; operations of the form  "IF rguard OP(modifier) rsrc1 -> rdest": guard,
;; modifier, one source and one destination register
;; #define MOD_SRC1_DEST_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 2) READ_AT_1) ; first operand
;;         (ress (match_arg 3) WRITE_AT_CYCLE(N)) ; destination register
;;         WRITEBACK_BUS_AT(N) ; there''s a write...
;;         ])
#define MOD_SRC1_DEST_TABLE(ISSUE_INFO,FU,N) \
```

```
(reser_table \
     [ \
       ISSUE_INFO \
       (ress (name FU) USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       (ress (match_arg 3) WRITE_AT_CYCLE(N)) \
       WRITEBACK_BUS_AT(N) \
       ])

;; operations of the form  "IF rguard OP rsrc1 rsrc2": guard and two source
;; registers, no modifier, no destination register
;; #define SRC1_SRC2_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 1) READ_AT_1) ; 1st opd (modifier skipped)
;;         (ress (match arg 2) READ_AT_1) ; 2nd operand
;;         ])
#define SRC1_SRC2_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
     [ \
       ISSUE_INFO \
       (ress (name FU) USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 1) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       ])

;; indirect jumps pf the form  "IF rguard OP rsrc1 rsrc2": guard and two
;; source registers, no modifier, no destination register
;; #define BRANCH_SRC1_SRC2_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 1) READ_AT_1) ; 1st opd (modifier skipped)
;;         (ress (match arg 2) READ_AT_1) ; 2nd operand
;;             (ress (name "PC") WRITE_AT_CYCLE(1)) ; write PC immediately
;;         ])
#define BRANCH_SRC1_SRC2_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
     [ \
       ISSUE_INFO \
```

```
        (ress (name FU) USE_AT_1) \
        (ress (match_arg 0) READ_AT_1) \
        (ress (match_arg 1) READ_AT_1) \
        (ress (match_arg 2) READ_AT_1) \
                (ress (name "PC") WRITE_AT_CYCLE(1)) \
        ])

;; operations of the form  "IF rguard OP rdest": guard and destination
;; register, no modifier, no source registers
;; #define DEST_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 1) WRITE_AT_CYCLE(N)) ; destination register
;;         WRITEBACK_BUS_AT(N) ; there''s a write...
;;         ])
#define DEST_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
      [ \
        ISSUE_INFO \
        (ress (name FU) USE_AT_1) \
        (ress (match_arg 0) READ_AT_1) \
        (ress (match_arg 1) WRITE_AT_CYCLE(N)) \
        WRITEBACK_BUS_AT(N) \
        ])

;; operations of the form "IF rguard OP rsrc1": guard and one source
;; register, no modifier, no destination register
;; #define SRC1_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 1) READ_AT_1) ; 1st opd (modifier skipped)
;;         ])
#define SRC1_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
      [ \
        ISSUE_INFO \
        (ress (name FU) USE_AT_1) \
        (ress (match_arg 0) READ_AT_1) \
        (ress (match_arg 1) READ_AT_1) \
        ])
```

```
;; operations of the form  "IF rguard OP(modifier) rdest": guard, modifier,
;; destination, no source register
;; #define MOD_DEST_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 2) WRITE_AT_CYCLE(N)) ; destination register
;;         WRITEBACK_BUS_AT(N) ; there''s a write...
;;         ])
#define MOD_DEST_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
     [ \
       ISSUE_INFO \
       (ress (name FU) USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 2) WRITE_AT_CYCLE(N)) \
       WRITEBACK_BUS_AT(N) \
       ])

;; operations of the form  "IF rguard OP(modifier)": guard, modifier, no
;; source nor destination registers;  NB: the number of cycles is unused
;; #define BRANCH_MOD_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;             (ress (name "PC") WRITE_AT_CYCLE(1)) ; write PC immediately
;;         ])
#define BRANCH_MOD_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
     [ \
       ISSUE_INFO \
       (ress (name FU) USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (name "PC") WRITE_AT_CYCLE(1)) \
       ])

;; immediate jumps of the form "IF rguard OP(modifier)": guard, modifier, no
;; source nor destination registers;  NB: the number of cycles is unused
;; #define MOD_TABLE(ISSUE_INFO,FU,N)
;; (reser_table
;;       [
;;         ISSUE_INFO
;;         (ress (name FU) USE_AT_1) ; 1st stage of func.unit
```

```
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         ])
#define MOD_TABLE(ISSUE_INFO,FU,N) \
(reser_table \
     [ \
       ISSUE_INFO \
       (ress (name FU) USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       ])

;; FTOUGH operations with two source operands, 16 cycles arithmetic, 1 cycle
;; write-back...
;; #define FTOUGH_THREE_ADDR_TABLE
;; (reser_table
;;       [
;;         ISSUE
;;         (ress (name "ftough") [(use) (from_cycle 2) (to_cycle 17)])
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 1) READ_AT_1) ; 1st opd (modifier skipped)
;;         (ress (match arg 2) READ_AT_1) ; second operand
;;         (ress (match_arg 3) WRITE_AT_CYCLE(18)) ; destination register
;;         WRITEBACK_BUS_AT(18) ; there''s a write...
;;         ])
#define FTOUGH_THREE_ADDR_TABLE \
(reser_table \
     [ \
       ISSUE \
       (ress (name "ftough") [(use) (from_cycle 2) (to_cycle 17)]) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 1) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       (ress (match_arg 3) WRITE_AT_CYCLE(18)) \
       WRITEBACK_BUS_AT(18) \
       ])

;; FTOUGH operations with one source operand, 16 cycles arithmetic, 1 cycle
;; write-back...
;; #define FTOUGH_SRC1_DEST_TABLE
;; (reser_table
;;       [
;;         ISSUE
;;         (ress (name "ftough") [(use) (from_cycle 2) (to_cycle 17)])
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 1) READ_AT_1) ; 1st opd (modifier skipped)
;;         (ress (match_arg 2) WRITE_AT_CYCLE(18)) ; destination register
;;         WRITEBACK_BUS_AT(18) ; there''s a write...
;;         ])
```

```
#define FTOUGH_SRC1_DEST_TABLE \
(reser_table \
     [ \
       ISSUE \
       (ress (name "ftough") [(use) (from_cycle 2) (to_cycle 17)]) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 1) READ_AT_1) \
       (ress (match_arg 2) WRITE_AT_CYCLE(18)) \
       WRITEBACK_BUS_AT(18) \
       ])

;; DMEMSPEC operations without a destination register: guard, modifier, and
;; one source register; this operation squats both dummy memory access units
;; so that no DMEM operation can be issued in a cycle in which a DMEMSPEC
;; operation is.
;; #define DMEMSPEC_MOD_SRC1_TABLE(N)
;; (reser_table
;;       [
;;         ISSUE
;;         (ress (name "dmemspec") USE_AT_1)
;;        (ress (name "mem_dummy_fu") USE_AT_1) ; 1st token
;;             (ress (name "mem_dummy_fu") USE_AT_1) ; 2nd token
;;         (ress (match_arg 0) READ_AT_1) ; guard
;;         (ress (match_arg 2) READ_AT_1) ; first operand
;;         ])
#define DMEMSPEC_MOD_SRC1_TABLE(N) \
(reser_table \
     [ \
       ISSUE \
       (ress (name "dmemspec") USE_AT_1) \
       (ress (name "mem_dummy_fu") USE_AT_1) \
               (ress (name "mem_dummy_fu") USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       ])

;; DMEMSPEC operations with a destination register: guard, modifier, one
;; source register, and a destination register;  this operation squats both
;; dummy memory access units so that no DMEM operation can be issued in a
;; cycle in which a DMEMSPEC operation is.
;; #define DMEMSPEC_MOD_SRC1_DEST_TABLE(N)
;; (reser_table
;;       [
;;         ISSUE
;;         (ress (name "dmemspec") USE_AT_1)
;;         (ress (name "mem_dummy_fu") USE_AT_1) ; 1st token
;;             (ress (name "mem_dummy_fu") USE_AT_1) ; 2nd token
```

```
;;          (ress (match_arg 0) READ_AT_1) ; guard
;;          (ress (match_arg 2) READ_AT_1) ; first operand
;;          (ress (match_arg 3) WRITE_AT_CYCLE(N)) ; destination register
;;          WRITEBACK_BUS_AT(N) ; there''s a write...
;;          ])
#define DMEMSPEC_MOD_SRC1_DEST_TABLE(N) \
(reser_table \
     [ \
       ISSUE \
       (ress (name "dmemspec") USE_AT_1) \
       (ress (name "mem_dummy_fu") USE_AT_1) \
               (ress (name "mem_dummy_fu") USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       (ress (match_arg 3) WRITE_AT_CYCLE(N)) \
       WRITEBACK_BUS_AT(N) \
       ])

;; DMEM operations with a modifier and one source register: guard, modifier,
;; one source register and one destination register
;; NB1: we assume that the memory read takes place at the first cycle
;; NB2: two DMEM ops can be issued in a cycle, but NONE when a DMEMSPEC op is
;; issued - this is why we use one "mem_dummy_fu" unit per DMEM op...
;; #define DMEM_MOD_SRC1_DEST_TABLE(N)
;; (reser_table
;;        [
;;          ISSUE
;;          (ress (name "dmem") USE_AT_1)
;;          (ress (name "mem_dummy_fu") USE_AT_1) ; just one token
;;          (ress (name "mem") READ_AT_1) ; memory access
;;          (ress (match_arg 0) READ_AT_1) ; guard
;;          (ress (match_arg 2) READ_AT_1) ; first operand
;;          (ress (match_arg 3) WRITE_AT_CYCLE(N)) ; destination register
;;          WRITEBACK_BUS_AT(N) ; there''s  a write...
;;          ])
#define DMEM_MOD_SRC1_DEST_TABLE(N) \
(reser_table  \
     [ \
       ISSUE \
       (ress (name "dmem") USE_AT_1) \
       (ress (name "mem_dummy_fu") USE_AT_1) \
       (ress (name "mem") READ_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       (ress (match_arg 3) WRITE_AT_CYCLE(N)) \
       WRITEBACK_BUS_AT(N) \
       ])
```

```
;; DMEM operations with two source registers and one destination register:
;; no modifier;
;; NB1: we assume that the memory read takes place at the first cycle
;; NB2: two DMEM ops can be issued in a cycle, but NONE when a DMEMSPEC op is
;; issued - this is why we use one "mem_dummy_fu" unit per DMEM op...
;; #define DMEM_THREE_ADDR_TABLE(N)
;; (reser_table
;;      [
;;        ISSUE
;;        (ress (name "dmem") USE_AT_1)
;;        (ress (name "mem_dummy_fu") USE_AT_1) ; just one token
;;        (ress (name "mem") READ_AT_1) ; memory access
;;        (ress (match_arg 0) READ_AT_1) ; guard
;;        (ress (match_arg 1) READ_AT_1) ; 1st opd (modifier skipped)
;;        (ress (match_arg 2) READ_AT_1) ; second operand
;;        (ress (match_arg 3) WRITE_AT_CYCLE(N)) ; destination register
;;        WRITEBACK_BUS_AT(N) ; there''s a write...
;;        ])
#define DMEM_THREE_ADDR_TABLE(N) \
(reser_table \
     [ \
       ISSUE \
       (ress (name "dmem") USE_AT_1) \
       (ress (name "mem_dummy_fu") USE_AT_1) \
       (ress (name "mem") READ_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 1) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       (ress (match_arg 3) WRITE_AT_CYCLE(N)) \
       WRITEBACK_BUS_AT(N) \
       ])

;; DMEM store operations with modifier and two source registers;
;; NB1: we assume that the memory write takes place at the last cycle, and
;; that register reads take place at the first cycle.
;; NB2: two DMEM ops can be issued in a cycle, but NONE when a DMEMSPEC op is
;; issued - this is why we use one "mem_dummy_fu" unit per DMEM op...
;; #define DMEM_MOD_SRC1_SRC2_TABLE(N)
;; (reser_table
;;      [
;;        ISSUE
;;        (ress (name "dmem") USE_AT_1)
;;        (ress (name "mem_dummy_fu") USE_AT_1) ; just one token
;;        (ress (match_arg 0) READ_AT_1) ; guard
;;        (ress (match_arg 2) READ_AT_1) ; 1st operand
;;        (ress (match_arg 3) READ_AT_1) ; second operand
```

```
;;         (ress (name "mem") WRITE_AT_CYCLE(N)) ; write into memory
;;         ])
#define DMEM_MOD_SRC1_SRC2_TABLE(N) \
(reser_table \
     [ \
       ISSUE \
       (ress (name "dmem") USE_AT_1) \
       (ress (name "mem_dummy_fu") USE_AT_1) \
       (ress (match_arg 0) READ_AT_1) \
       (ress (match_arg 2) READ_AT_1) \
       (ress (match_arg 3) READ_AT_1) \
       (ress (name "mem") WRITE_AT_CYCLE(N)) \
       ])
```

A.2 Semantical constraints

```
;;-----------------------------------------------------------------------------
;;  Philips TriMedia-1000 processor description (restricted to the DSPCPU)
;;
;;  SEMANTICS section: semantical constraints in instructions
;;
;;  Zbigniew CHAMSKI <{zchamski,manche}@irisa.fr>
;;-----------------------------------------------------------------------------

;; there is only one type of constraints: delay slots, common to all branch
;; instructions.

;; conditional _branch_ information:
;; #define BRANCH_SEM_INFO \
;; (sem [ \
;;        (delay_slot 3) ; there are three delay slots \
;;        (noreorder)    ; forbid reordering of insns \
;;        (branch -1)    ; indirect addressing \
;;        ])
#define BRANCH_SEM_INFO \
(sem [ \
       (delay_slot 3) \
       (noreorder) \
       (branch -1) \
       ])

;; unconditional _jump_ information: well, it is still a branch if the guard
;; is given
;; #define JUMP_SEM_INFO \
```

```
;; (sem [ \
;;       (delay_slot 3) ; there are three delay slots \
;;       (noreorder)    ; forbid reordering of insns \
;;       (jump 1)       ; target address in modifier \
;;       ])
#define JUMP_SEM_INFO \
(sem [ \
       (delay_slot 3) \
       (noreorder) \
       (jump 1) \
       ])
```

A.3 Main file: assembler structure and the instruction set

```
;;-----------------------------------------------------------------------------
;;  Philips TriMedia-1000 processor description (restricted to the DSPCPU)
;;
;;  MAIN FILE: includes, assembler structure and instruction set definition
;;
;;  Created by
;;  Zbigniew CHAMSKI <{zchamski,manche}@irisa.fr>
;;-----------------------------------------------------------------------------

;; Section I: INCLUDE DIRECTIVES
;; =============================

#include "tm1-res.def"
; resource declarations,
; including reservation tables
;; #include "tm1-macro.def"           ; no macros as of yet
#include "tm1-sem.def"
; semantical information


;; Section II: LEXICAL STRUCTURE OF THE ASSEMBLER
;; ==============================================

;; Section II.1: COMMENTS AND SEPARATORS
;; =====================================

(line_comment_chars "!")
(comment_chars "#")

;; parentheses, dash,  and "greater" sign (right chevron) as
```

```
;; separators...
(def_exact "(),->IF")


;; Section II.2: META-VARIABLE TOKENS
;; ==================================
;;
;; register tokens are quite nice:
#define REGISTER_REGEXP \
  "r12[0-7]\\|r1[0-1][0-9]\\|r[1-9][0-9]\\|r[0-9]"
(def_token "g" [(regex REGISTER_REGEXP)]) ; guard register
(def_token "s" [(regex REGISTER_REGEXP)]) ; first source register
(def_token "t" [(regex REGISTER_REGEXP)]) ; second source register
(def_token "d" [(regex REGISTER_REGEXP)]) ; destination register

;; sometimes r0 must be explicitly given
(def_token "0" [(regex "r0")])

;; (def_token "m" [(regex REGISTER_REGEXP)]) ; modificateur bidon
;; modifier tokens: integer expressions, possibly containing identifiers
(def_token "m" [(read_exp)])

;; Section III: INSTRUCTION SET SPECIFICATION
;; ==========================================
;;
;; Section III.1: VLIW INSTRUCTION WIDTH
;; =====================================
;;
;; five issue slots per cycle
(inst_width 5)

;; Section III.2: NATIVE INSTRUCTIONS
;; ==================================
;;
;; Instructions are assumed to complete at cycle following the latency cycle
;; - there''s an EXPLICIT decode/read cycle at t=1 (0 is insn fetch...)
;; Latencies given as parameters of reservation tables take this already
;; into account.

;; BTW, spaces are not significant in the format string and ARE IGNORED
;; during unparsing.
#define INPUT_THREE_ADDR input "IF g s t -> d"
#define INPUT_DEST input "IF g -> d"
#define INPUT_MOD input "IF g (m)"
#define INPUT_MOD_DEST input "IF g (m) -> d"
#define INPUT_MOD_SRC1 input "IF g (m) s"
#define INPUT_MOD_SRC1_DEST input "IF g (m) s -> d"
```

```
#define INPUT_MOD_SRC1_SRC2 input "IF g (m) s t"
#define INPUT_SRC1 input "IF g s"
#define INPUT_SRC1_DEST input "IF g s -> d"
#define INPUT_SRC1_SRC2 input "IF g s t"

;; Section III.2.0: The NOP
;; ========================
;;
;; A 'nop' is a particular case: it can be issued anywhere, anytime, has no
;; operands and simply takes one issue unit...

(def_asm "nop"
  [(input "IF g" )
   (reser_table
    [ ISSUE ])
   ])

;; Section III.2.1: ALU OPERATIONS
;; ===============================
;;
;; most "alu" operations are three-address ops completed in one cycle
;; the write takes place at the next cycle (t=2)
#define ALU_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   THREE_ADDR_TABLE(ISSUE,"alu",2) \
   ])

ALU_THREE_ADDR_OP("iadd") ; signed integer addition
ALU_THREE_ADDR_OP("isub") ; signed integer subtraction
ALU_THREE_ADDR_OP("igtr") ; signed greater
ALU_THREE_ADDR_OP("igeq") ; signed greater or equal
ALU_THREE_ADDR_OP("ieql") ; signed equal
ALU_THREE_ADDR_OP("ineq") ; signed not equal
ALU_THREE_ADDR_OP("ugtr") ; unsigned greater
ALU_THREE_ADDR_OP("ugeq") ; unsigned greater or equal
ALU_THREE_ADDR_OP("bitand") ; bitwise logical and
ALU_THREE_ADDR_OP("bitor") ; bitwise logical or
ALU_THREE_ADDR_OP("bitxor") ; bitwise logical exclusive or
ALU_THREE_ADDR_OP("bitandinv") ; bitwise logical "and not"
ALU_THREE_ADDR_OP("carry") ; carry bit from unsigned add
ALU_THREE_ADDR_OP("izero") ; if zero select zero
ALU_THREE_ADDR_OP("inonzero") ; if nonzero select zero
ALU_THREE_ADDR_OP("packbytes") ; pack least significant bytes
ALU_THREE_ADDR_OP("mergelsb") ; merge least significant bytes
ALU_THREE_ADDR_OP("mergemsb") ; merge most significant bytes
ALU_THREE_ADDR_OP("pack16lsb") ; pack least significant half-words
```

```
ALU_THREE_ADDR_OP("pack16msb") ; pack most significant half-words
ALU_THREE_ADDR_OP("ibytesel") ; signed select byte
ALU_THREE_ADDR_OP("ubytesel") ; unsigned select byte

;; "alu" ops that use an immediate value, source1 and dest register
#define ALU_MOD_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_MOD_SRC1_DEST) \
   MOD_SRC1_DEST_TABLE(ISSUE,"alu",2) \
   ])

ALU_MOD_SRC1_DEST_OP("ileqi") ; signed less or equal than imm
ALU_MOD_SRC1_DEST_OP("igtri") ; signed greater than immediate
ALU_MOD_SRC1_DEST_OP("igeqi") ; signed greater or equal than imm
ALU_MOD_SRC1_DEST_OP("ilesi") ; signed less than immediate
ALU_MOD_SRC1_DEST_OP("ieqli") ; signed equal to immediate
ALU_MOD_SRC1_DEST_OP("ineqi") ; signed not equal to immediate
ALU_MOD_SRC1_DEST_OP("uleqi") ; unsigned less or equal than imm
ALU_MOD_SRC1_DEST_OP("ugtri") ; unsigned greater than immediate
ALU_MOD_SRC1_DEST_OP("ugeqi") ; unsigned greater or equal than imm
ALU_MOD_SRC1_DEST_OP("ulesi") ; unsigned less than immediate
ALU_MOD_SRC1_DEST_OP("ueqli") ; unsigned equal to immediate
ALU_MOD_SRC1_DEST_OP("uneqi") ; unsigned not equal to immediate
ALU_MOD_SRC1_DEST_OP("iaddi") ; signed add immediate
ALU_MOD_SRC1_DEST_OP("isubi") ; signed subtract immediate

;; "alu" ops that use one source register and a destination reg
#define ALU_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_SRC1_DEST) \
   SRC1_DEST_TABLE(ISSUE,"alu",2) \
   ])

ALU_SRC1_DEST_OP("bitinv") ; bitwise logical not
ALU_SRC1_DEST_OP("sex8") ; sign extend LSbyte
ALU_SRC1_DEST_OP("sex16") ; sign extend LShalf-word

;; h_iabs uses explicitly r0 to compute the absolute values - we need a
;; dedicated reservation table...

(def_asm "h_iabs"
  [(input "g 0 t -> d") ; guard r0 source -> dest
   (reser_table
    [ISSUE
     (ress (name "alu") USE_AT_1)
     (ress (name "r0") READ_AT_1)
     (ress (match_arg 0) READ_AT_1) ; guard register
```

```
    (ress (match_arg 2) READ_AT_1) ; value to be abs''ed
    (ress (match_arg 3) WRITE_AT_CYCLE(2)) ; result
    ])
  ])

;; Section III.2.2: BRANCH OPERATIONS
;; ==================================
;;
;; All branch operations take three delay slots and recover in one cycle
#define BRANCH_IMMEDIATE(op) \
(def_asm op \
  [(INPUT_MOD) \
   BRANCH_MOD_TABLE(ISSUE,"branch",1) \
   JUMP_SEM_INFO \
   ])

#define BRANCH_INDIRECT(op) \
(def_asm op \
  [(INPUT_SRC1_SRC2) \
   BRANCH_SRC1_SRC2_TABLE(ISSUE,"branch",1) \
   BRANCH_SEM_INFO \
   ])

BRANCH_IMMEDIATE("ijmpi") ; interruptible jump to imm
BRANCH_IMMEDIATE("jmpi") ; non-interruptible jump to imm

BRANCH_INDIRECT("ijmpf") ; interruptible jump on false
BRANCH_INDIRECT("ijmpt") ; interruptible jump on true
BRANCH_INDIRECT("jmpf") ; jump on false
BRANCH_INDIRECT("jmpt") ; jump on true

;; Section III.2.3: CONST OPERATIONS
;; =================================
;;
;; Just two ops... loading immediate values into registers

(def_asm "iimm" ; load signed immediate
  [(INPUT_MOD_DEST)
   MOD_DEST_TABLE(ISSUE,"const",2)
   ])

(def_asm "uimm" ; load unsigned immediate
  [(INPUT_MOD_DEST)
   MOD_DEST_TABLE(ISSUE,"const",2)
   ])

;; Section III.2.4: DMEM OPERATIONS
```

```
;; ================================
;;
;; There are three operand patterns:
;;   op(m) src1 -> dest, op src1 src2 -> dest, and op(m) src1 src2

;; latency is three cycles
#define DMEM_MOD_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_MOD_SRC1_DEST) \
   DMEM_MOD_SRC1_DEST_TABLE(4) \
   ])

;; latency is three cycles
#define DMEM_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   DMEM_THREE_ADDR_TABLE(4) \
   ])

;; latency is three cycles
#define DMEM_MOD_SRC1_SRC2_OP(op) \
(def_asm op \
  [(INPUT_MOD_SRC1_SRC2) \
   DMEM_MOD_SRC1_SRC2_TABLE(4) \
   ])

DMEM_MOD_SRC1_DEST_OP("ild8d") ; 8-bit signed load w/offset
DMEM_MOD_SRC1_DEST_OP("uld8d") ; 8-bit unsigned load w/offset
DMEM_MOD_SRC1_DEST_OP("ild16d") ; 16-bit signed load w/offset
DMEM_MOD_SRC1_DEST_OP("uld16d") ; 16-bit unsigned load w/offset
DMEM_MOD_SRC1_DEST_OP("ld32d") ; 32-bit load w/offset

DMEM_THREE_ADDR_OP("ild8r") ; signed 8-bit load w/index
DMEM_THREE_ADDR_OP("uld8r") ; unsigned 8-bit load w/index
DMEM_THREE_ADDR_OP("ild16r") ; signed 16-bit load w/index
DMEM_THREE_ADDR_OP("uld16r") ; unsigned 16-bit load w/index
DMEM_THREE_ADDR_OP("ld32r") ; 32-bit load with index
DMEM_THREE_ADDR_OP("ild16x") ; signed 16-bit load w/scaled index
DMEM_THREE_ADDR_OP("uld16x") ; unsigned 16-bit load w/scaled index
DMEM_THREE_ADDR_OP("ld32x") ; 32-bit load with scaled index

DMEM_MOD_SRC1_SRC2_OP("h_st8d") ; 8-bit store with offset
DMEM_MOD_SRC1_SRC2_OP("h_st16d") ; 16-bit store with offset
DMEM_MOD_SRC1_SRC2_OP("h_st32d") ; 32-bit store with offset

;; Section III.2.5: DMEMSPEC OPERATIONS
;; ====================================
```

```
;;

;; latency is three cycles
#define DMEMSPEC_MOD_SRC1_OP(op) \
(def_asm op \
  [(INPUT_MOD_SRC1) \
   DMEMSPEC_MOD_SRC1_TABLE(4) \
   ])

;; latency is three cycles
#define DMEMSPEC_MOD_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_MOD_SRC1_DEST) \
   DMEMSPEC_MOD_SRC1_DEST_TABLE(4) \
   ])

DMEMSPEC_MOD_SRC1_OP("dcb") ; copy back data cache block
DMEMSPEC_MOD_SRC1_OP("dinvalid") ; invalidate data cache block

DMEMSPEC_MOD_SRC1_DEST_OP("rdstatus") ; read data cache status bits
DMEMSPEC_MOD_SRC1_DEST_OP("rdtag") ; read data cache address tag

;; Section III.2.6: DSPALU OPERATIONS
;; ==================================
;;

#define DSPALU_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   THREE_ADDR_TABLE(ISSUE,"dspalu",3) \
   ])

;; DSP valabses: these OPs use explicitly r0 as first operand - let''s enforce
;; it
;; #define DSPALU_r0_SRC2_DEST_OP(op) \
;;   [(input "g 0 t -> d") \ ; guard r0 src2 -> dest
;;    (reser_table \
;;     [ISSUE \
;;      (ress (name "dspalu") USE_AT_1) \
;;      (ress (name "r0") READ_AT_1) ; must be explicitly given \
;;      (ress (match_arg 0) READ_AT_1)  ; guard register \
;;      (ress (match_arg 2) READ_AT_1) ; value to be abs''ed \
;;      (ress (match_arg 3) WRITE_AT_CYCLE(3)) ; result: latency is two \
;;  ; cycles when using the bypass \
;;      ])
#define DSPALU_r0_SRC2_DEST_OP(op) \
(def_asm op \
```

```
  [(input "g 0 t -> d") \
   (reser_table \
    [ISSUE \
     (ress (name "dspalu") USE_AT_1) \
     (ress (name "r0") READ_AT_1) \
     (ress (match_arg 0) READ_AT_1) \
     (ress (match_arg 2) READ_AT_1) \
     (ress (match_arg 3) WRITE_AT_CYCLE(3)) \
     ]) \
  ])

DSPALU_THREE_ADDR_OP("ume8ii") ; sum of valabses of unsigned diffs
DSPALU_THREE_ADDR_OP("ume8uu") ; sum of valabses of signed diffs
DSPALU_THREE_ADDR_OP("dspiadd") ; clipped signed add
DSPALU_THREE_ADDR_OP("dspisub") ; clipped signed sub
DSPALU_THREE_ADDR_OP("dspuadd") ; clipped unsigned add
DSPALU_THREE_ADDR_OP("dspusub") ; clipped unsigned sub
DSPALU_THREE_ADDR_OP("dspidualadd") ; dual clipped signed add, 1/2 words
DSPALU_THREE_ADDR_OP("dspidualsub") ; dual clipped signed sub, 1/2 words
DSPALU_THREE_ADDR_OP("iavgonep") ; quad unsinged/signed add, bytes
DSPALU_THREE_ADDR_OP("iflip") ; negate src2 if src1 is zero
DSPALU_THREE_ADDR_OP("iclipi") ; clip signed to signed
DSPALU_THREE_ADDR_OP("uclipi") ; clip signed to unsigned
DSPALU_THREE_ADDR_OP("uclipu") ; clip unsigned to unsigned
DSPALU_THREE_ADDR_OP("quadavg") ; unsigned bytewise quad average
DSPALU_THREE_ADDR_OP("dspuquadaddui") ; quad clipped add of unsigned/signed
DSPALU_THREE_ADDR_OP("imax") ; signed maximum
DSPALU_THREE_ADDR_OP("imin") ; signed minimum

DSPALU_r0_SRC2_DEST_OP("h_dspiabs") ; clipped signed absolute value
DSPALU_r0_SRC2_DEST_OP("h_dspidualabs") ; dual clipped signed absval hfwords


;; Section III.2.7: DSPMUL OPERATIONS
;; ==================================

#define DSPMUL_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   THREE_ADDR_TABLE(ISSUE,"dspalu",3) \
])

DSPMUL_THREE_ADDR_OP("ifir16") ; signed dot-product of 1/2words
DSPMUL_THREE_ADDR_OP("ufir16") ; unsigned dot-product of 1/2words
DSPMUL_THREE_ADDR_OP("ifir8ii") ; signed dot-prod of signed bytes
DSPMUL_THREE_ADDR_OP("ifir8ui") ; signed dot-prod of s''d/uns''d bytes
DSPMUL_THREE_ADDR_OP("ufir8uu") ; unsigned dot-prod of unsigned bytes
```

```
DSPMUL_THREE_ADDR_OP("dspidualmul") ; dual clipped mult''y of signed 1/2wd
DSPMUL_THREE_ADDR_OP("quadumulmsb") ; quad clipped mult''y of unsd MSbytes

;; Section III.2.7: FALU OPERATIONS
;; ================================

;; source-to-dest operations: conversions
#define FALU_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_SRC1_DEST) \
   SRC1_DEST_TABLE(ISSUE,"falu",4) \
   ])

#define FALU_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   THREE_ADDR_TABLE(ISSUE,"falu",4) \
   ])

FALU_SRC1_DEST_OP("fabsval") ; absolute value of float
FALU_SRC1_DEST_OP("fabsvalflags") ; flags from absolute value of float
FALU_SRC1_DEST_OP("ifixieee") ; convert float to signed, crnt mode
FALU_SRC1_DEST_OP("ifixieeeflags") ; flags from "ifixieee"
FALU_SRC1_DEST_OP("ifixrz") ; convert float to signed, rnd -> 0
FALU_SRC1_DEST_OP("ifixrzflags") ; flags from "ifixrz"
FALU_SRC1_DEST_OP("ifloat") ; convert signed to float
FALU_SRC1_DEST_OP("ifloatflags") ; flags from "ifloat"
FALU_SRC1_DEST_OP("ifloatrz") ; convert signed to float, rnd -> 0
FALU_SRC1_DEST_OP("ifloatrzflags") ; flags from "ifloatrz"

FALU_SRC1_DEST_OP("ufixieee") ; convert float to unsignd, crnt mode
FALU_SRC1_DEST_OP("ufixieeeflags") ; flags from "ufixieee"
FALU_SRC1_DEST_OP("ufixrz") ; convert float to unsignd, rnd -> 0
FALU_SRC1_DEST_OP("ufixrzflags") ; flags from "ufixrz"
FALU_SRC1_DEST_OP("ufloat") ; convert unsignd to float, crnt mode
FALU_SRC1_DEST_OP("ufloatflags") ; flags from "ufloat"
FALU_SRC1_DEST_OP("ufloatrz") ; convert unsignd to float, rnd -> 0
FALU_SRC1_DEST_OP("ufloatrzflags") ; flags from "ufloatrz"

FALU_THREE_ADDR_OP("fadd") ; floating-point addition
FALU_THREE_ADDR_OP("faddflags") ; flags from floating-point addition
FALU_THREE_ADDR_OP("fsub") ; floating-point subtraction
FALU_THREE_ADDR_OP("fsubflags") ; flags from floating-point subtract

;; Section III.2.9: FCOMP OPERATIONS
;; =================================
;;
```

```
;; all ops complete in a single cycle and write the result at t=2

#define FCOMP_DEST_OP(op) \
(def_asm op \
  [(INPUT_DEST) \
   DEST_TABLE(ISSUE,"fcomp",2) \
   ])

#define FCOMP_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   THREE_ADDR_TABLE(ISSUE,"fcomp",2) \
   ])

#define FCOMP_SRC1_SRC2_OP(op) \
(def_asm op \
  [(INPUT_SRC1_SRC2) \
   SRC1_SRC2_TABLE(ISSUE,"fcomp",2) \
   ])

#define FCOMP_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_SRC1_DEST) \
   SRC1_DEST_TABLE(ISSUE,"fcomp",2) \
   ])

#define FCOMP_SRC1_OP(op) \
(def_asm op \
  [(INPUT_SRC1) \
   SRC1_TABLE(ISSUE,"fcomp",2) \
   ])

FCOMP_DEST_OP("cycles") ; clock reading, LSword
FCOMP_DEST_OP("hicycles") ; clock reading, MSword
FCOMP_DEST_OP("readdpc") ; destination program counter value
FCOMP_DEST_OP("readpcsw") ; read program cntrl and status word

FCOMP_SRC1_OP("writedpc") ; write destination program counter
FCOMP_SRC1_OP("writespc") ; write source program counter

FCOMP_SRC1_SRC2_OP("writepcsw") ; write program cntrl and status word

FCOMP_THREE_ADDR_OP("feql") ; FP "equal" test
FCOMP_THREE_ADDR_OP("feqlflags") ; FP "equal" test flags
FCOMP_THREE_ADDR_OP("fgeq") ; FP "greater or equal" test
FCOMP_THREE_ADDR_OP("fgeqflags") ; FP "greater or equal" test flags
FCOMP_THREE_ADDR_OP("fgtr") ; FP "greater"
```

```
FCOMP_THREE_ADDR_OP("fgtrflags") ; FP "greater" test flags
FCOMP_THREE_ADDR_OP("fneq") ; FP "not equal" test
FCOMP_THREE_ADDR_OP("fneqflags") ; FP "not equal" test flags

FCOMP_SRC1_DEST_OP("fsign") ; FP sign test
FCOMP_SRC1_DEST_OP("fsignflags") ; FP sign test flags

;; Section III.2.10: FTOUGH OPERATIONS
;; ===================================

#define FTOUGH_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   FTOUGH_THREE_ADDR_TABLE \
   ])

#define FTOUGH_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_SRC1_DEST) \
   FTOUGH_SRC1_DEST_TABLE \
   ])

FTOUGH_THREE_ADDR_OP("fdiv") ; FP division
FTOUGH_THREE_ADDR_OP("fdivflags") ; flags from FP division

FTOUGH_SRC1_DEST_OP("fsqrt") ; FP square root
FTOUGH_SRC1_DEST_OP("fsqrtflags") ; flags from FP square root


;; Section III.2.11: IFMUL OPERATIONS
;; ==================================
;;
;; all ops use a three-stage pipeline and write the result at cycle 4

#define IFMUL_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   THREE_ADDR_TABLE(ISSUE,"ifmul",4) \
   ])

IFMUL_THREE_ADDR_OP("dspimul") ; clipped signed multiply
IFMUL_THREE_ADDR_OP("dspumul") ; clipped unsigned multiply
IFMUL_THREE_ADDR_OP("fmul") ; FP multiply
IFMUL_THREE_ADDR_OP("fmulflags") ; flags from FP multiply
IFMUL_THREE_ADDR_OP("imul") ; LSword of signed multiply
IFMUL_THREE_ADDR_OP("imulm") ; MSword of signed multiply
IFMUL_THREE_ADDR_OP("umul") ; LSword of unsigned multiply
```

```
IFMUL_THREE_ADDR_OP("umulm") ; MSword of unsigned multiply


;; Section III.2.12: SHIFTER OPERATIONS
;; ===================================
;;
;; all ops complete in one cycle and write the result at t=2

#define SHIFTER_THREE_ADDR_OP(op) \
(def_asm op \
  [(INPUT_THREE_ADDR) \
   THREE_ADDR_TABLE(ISSUE,"shifter",2) \
   ])

#define SHIFTER_MOD_SRC1_DEST_OP(op) \
(def_asm op \
  [(INPUT_MOD_SRC1_DEST) \
   MOD_SRC1_DEST_TABLE(ISSUE,"shifter",2) \
   ])

SHIFTER_THREE_ADDR_OP("asl") ; arithmetic shift left
SHIFTER_THREE_ADDR_OP("asr") ; arithmetic shift right
SHIFTER_THREE_ADDR_OP("funshift1") ; funnel-shift (rotate) left 1 byte
SHIFTER_THREE_ADDR_OP("funshift2") ; funnel-shift (rotate) left 2 bytes
SHIFTER_THREE_ADDR_OP("funshift3") ; funnel-shift (rotate) left 3 bytes
SHIFTER_THREE_ADDR_OP("lsr") ; logical shift right
SHIFTER_THREE_ADDR_OP("rol") ; rotate left

SHIFTER_MOD_SRC1_DEST_OP("asli") ; arithmetic shift left by immediate
SHIFTER_MOD_SRC1_DEST_OP("asri") ; arithmetic shift right by immediate
SHIFTER_MOD_SRC1_DEST_OP("lsri") ; logical shift right by immediate
SHIFTER_MOD_SRC1_DEST_OP("roli") ; rotate left by immediate
```