# Transferbereich 14

---

## TDL
## A Hardware and Assembly Description Language

Number TDL 1.4

---

Daniel Kästner
kaestner@cs.uni-sb.de

February 2, 2000

# Contents

# Chapter 1

# Introduction

TDL (Target Description Language) is a descriptive language for the specification of the hardware resources and assembly language of a given target processor. Application areas of TDL are assembly-based postpass optimizations and analyses which have to consider target-specific properties in a machine-independent way. The properties of all hardware resources which can influence the analysis and optimization process must be known. Furthermore, the instructions of the processor must be recognized and the effects of each operation must be known. Each TDL specification is composed of several parts: a specifiation of the hardware resources, a description of the instruction set, a constraint section, and an assembly section.

In the resource section , each hardware component which can influence the analyses and optimizations is introduced. Behavior and assembly representation of the resources can be specified by dedicated attributes. By resource aliasing and the facilities of introducing user-defined, or virtual resources, different levels are supported.

The instruction set section contains a definition of the instruction set in the form of an attribute grammar. Each operation is represented by a rule of this grammar; repeatedly occurring instruction parts, as ,e.g., addressing modes, can be described by dedicated rules. A flexible attribute mechanism allows the identification of important operation properties which have to be considered for analyses and optimizations. This includes the assembly representation of each instruction, the set of source and destination operands, timing information, etc (see Sec.4). This way, important informations can be accessed directly without having to interprete the detailed specification of the operation semantics. The detailed semantics of the operations is specified by an extended register transfer language. The RTL-language used in TDL allows for a semantical analysis of the machine operations; e.g. an elementary type checking can be performed. Another important design consideration is the concept of abstraction: different abstraction levels are supported via functional abstraction. Thus the same description file can be used for applications requiring different levels of detail in the semantical specification.

The constraint section is composed of a set of constraints which must be respected to preserve correctness during code transformations. This way, also complex interactions between different operations can be described. An example is the restricted parallelism of

ALU- and multiplier operations of the ADSP-2106x. Parallel execution is only possible, if each operand is located in one dedicated register group. Otherwise, the operations are still feasible, but cannot be executed in parallel.

The assembly section deals with assembly macros, instruction delimiters, instruction delimiter assembly comments and directives .

Each TDL description is checked for semantical consistency . The properties of hardware resources and operations are specified by dedicated attributes. Each attribute is associated with a set of hardware resources and a dedicated domain. This holds for the predefined as well as for additional user-defined attributes . Assigning an incorrect value to an attribute or using an attribute to specifiy a hardware resource not associated with that attribute produces an error. Other semantical checks prevent, for example aliases to non-existing resources, exceeding register set sizes and so on.

Based on the machine description, several ANSI C files are generated. The generated functions initialize data structures containing all target dependent information specified in TDL. These files can be compiled and linked with the target application. Second, a parser for the assembly language is automatically generated. The parser transforms an input program into a generic control flow graph; it is implemented in PERL. This way, portability is ensured and rapid prototyping is supported, since it is easy to write simple parsers for subsets of the language by hand.

Throughout this documentation, the following lexical conventions are used: keywords are written in typewriter font, non-terminals in italics. The keywords are case-insensitive. However the spelling of all user-defined resources, attributes, and attribute values is case-sensitive.

# Chapter 2

# Related Work

Hardware description languages are used for a variety of different target applications: for architectural synthesis, hardware simulation, code generation and analyses. In consequence, a large number of hardware description languages has been developed. In the area of processor modelling and simulating, VHDL [11] and Verilog [14] are widely used; for code generation, well-known approaches are ISPS [2], MARIL [4], MIMOLA [12], or nML [6]. Recent developments are LISA [13] aiming at generating cycle-accurate simulators for architectures with complex pipelines and ISDL [8] which is intended to be used for generating a set of tools like assemblers, simulators and code generators. Another approach is the EXPRESSION [9] language. One important problem is to provide different levels of abstraction within the same specification; this is the reason for the great number of description languages adapted to different target applications.

However none of these languages can be used as a basis for assembly-based code analyses and transformations without major modifications. In this case, the target description must capture the properties of the hardware as well as the definition of the assembly language which is used. The only existing language in this area has been developed within the SALTO-framework [3].

In its basic structure, TDL is similar to MARIL and has been mainly influenced by SALTO [3] and ISDL [8]. The basic design philosophy of the SALTO system is to provide the user with an environment that allows him to implement tools for analyses and transformations of low-level code. From the description of the target machine and the assembly language a parser is generated which transforms the input program in a generic control flow graph. The user is offered an object oriented interface to access and manipulate the data structures representing this CFG. The SALTO-description language allows the user to introduce any kind of resources. However this flexibility is restricted by a small, predefined set of attributes which can be used to specify the properties of these resources whithin the machine specification. It is not possible to introduce additional user-defined attributes for the specification of hardware properties. Similarly, the instructions are specified by defining their assembly representation, their resource access sequence and the semantics. No additional attributes can be given to specify further properties which might be relevant for analyses or transformations. No detailed operation semantics can be specified; only a coarse classification of control flow operations and the specification of a jump target is possible. So the set of target applications built on top of SALTO is restricted; many analyses would need additional informations provided by the user externally to the machine description proper. Moreover, there is no mechanism to specify explicit constraints for modelling complex interactions among operations or between operations and allocation decisions (see Sec. 1, 5). Another important drawback is the lack of semantical checks for the machine description file. It is not possible during the parsing of the machine specification to inform the user of errors and inconsistencies within the description file. Thus it is possible to refer to resources which have not been declared, to define instructions which assign floating-point values to integer register, to exceed register file sizes, etc. The consequences of such erroneous specifications would only show up during the processing of assembly input programs. The instruction specification mechanism is not completely compositional. In order to shorten the description, `cpp` preprocessor directives can be used which are based on purely textual replacement without semantical information. If e.g. an instruction can be used with an optional guard, two different operation definitions would have to be provided if the differences in the operation behavior are to be expressed. Finally the LISP-like description syntax and exhaustive use of the `cpp` preprocessor directives can affect the readability of the specification for the human reader.

ISDL [8] is designed to support a wide range of tools, from code generators, disassemblers up to instruction set simulators. On one hand, the language is therefore more powerful than needed for assembly-based analyses and transformations. On the other hand, it cannot provide some flexibility which is essential for these applications. Since the information specified in ISDL is used to generate an assembler and disassembler, the binary coding of each instruction has to be declared in the instruction set description, which is not a prerequisite in the scope of TDL. The internal handling of assemblers and disassemblers allows ISDL to make simplifying assumptions on the structure of the assemble language. It is not possible to specify assembly directives or the syntax of assembly comments, instruction delimiters, etc in ISDL. The set of resource types supported by ISDL is restricted; there is no possibility for introducing additional user-defined resource types. In the current version of ISDL, there is no pipeline modelling; the focus is mainly on VLIW-like architectures without complex pipelines[1]. There is no attribute mechanism within ISDL, thus building analysers on top of an ISDL specification can be difficult. Some information can only be represented within the specification of the semantics of the operations so that analysers potentially have to interprete the complete semantical specification of each operation.

nML [6] is a machine description formalism which is used e.g. within the retargetable compiler CHESS [10]. The instruction set is modelled as an attribute grammar; each terminal of the grammar corresponds to a valid instruction. In addition to the instruction set grammar, an nML-description can contain declarations for storage objects, data types, constants and macros. Resource declarations are only possible for storage objects; declarations of functional units or caches are not supported and there is no possiblity of introducing

[1]In TDL, a simplified pipeline model is assumed, too. However there is ongoing work for a modelling of complex pipelines including out-of-order execution.

additional, user-defined resource types. However, if a description is to be used as a basis for program analyses, such informations have to be provided, e.g. for the static cache analysis of [7]. For the specification of the instruction set, in nML only the attributes `action` (semantics description), `syntax` (assembly representation) and `image` (binary representation) are predefined. In order to gather informations e.g. about source and destination operands, the semantics description has to be analysed; by introducing additional attributes for the operand description, redundancies would be introduced. Attributes are declared implicitly in nML without specifying a domain for the feasible attribute values. This way it is not possible to check the feasibility of attribute settings; there is no semantical analysis of the machine description itself. Also in nML no constraint mechanism to model complex dependences between operations is provided. Since nML has not been designed for postpass optimizations and analyses, a specification of the syntax of assembly comments, directives or of instruction delimiters is not supported.

The primary application area of TDL are assembly-based postpass optimizations and analyses, but also the generation of simulators should be supported. Another potential use is to derive code selector generator specifications from a TDL-description. The variety of application areas is primarily supported by the functional abstraction mechanism within the specification of operation semantics. Another important property of TDL is the modularity which is also at the base of the CSDL language families, developed at the University of Virginia[2]. A valid TDL description can consist only of the resource section; this way, it can be used as a pure hardware description language. Except from the resource section, all sections of a TDL description are optional; thus TDL can be flexibly used. The most serious difficulty when using a description formalism in an application it has not been originally designed for, is the interface to the application. In the worst case, the description language parser has to be completely reprogrammed. This interfacing problem is addressed by TDL in generating ANSI C-Code which represents all information specified by the TDL description. These C-routines are completely independent of the target application; thus integrating the TDL description and the TDL parser into another application is easily possible.

[2] `http://www.cs.virginia.edu/zephyr/csdl/overview.html`

# Chapter 3

# The Resource Section

The first part of the specification covers the description of the hardware resources of the target processor. The formalism provides pre-defined resource types and predefined attributes which can carry syntactical and semantical information. The user can add new values to the predefined attributes and can introduce new attributes with associated domains.

There are several correctness checks. For each assignment of a value to an attribute, it is checked whether this value is contained in the domain of the attribute. All user-defined attributes have to be explicitly specified. This specification includes the set of possible values and the resources which can be described by the given attribute.

The resource section is introduced by the keyword `Resources-Section`. In the following sections, all declarations which can be used in the resource section are described in detail.

## 3.1 Declaration of Register Sets

`Register` *genname asmname RangeSpec RegSpecList* ;

- *genname* can be any name starting with a letter followed by a sequence of alphanumeric characters. Each reference to the defined register set must use exact the same spelling and case as given in *genname*.

- *asmname* is a sequence of characters enclosed in double quotes ("). Its function is that of a format string similar to the format strings in C. If the string contains, e.g., a `%d`-directive, that means that the directive is to be replaced by an integer constant. This way, the naming of individual registers can be described implicitly. The assembly representation of each element register in the specified set is generated implicitly by substituing the `%d`-directive by the index of the register within its registerfile. Note however, that this default naming can be overridden by using the `SetProperties` directive by introducing a resource alias (see below).

- *RangeSpec*: $\epsilon$ | `[` *intval* `]` | `[` *intval* `:` *intval* `]`

If *RangeSpec* is empty, a registerset containing only one register is created. If only one integer is given in the brackets, then this *intval* is assumed to be the size of the register file. If *RangeSpec* = `[a:b]`, then a registerset containing `(b-a+1)` elements is created. Indexing always starts with 0.

- *RegSpecList*: ( *regspec* `,` ) * *regspec*
- `regspec`: `size=` *intval* | `type=` *typespec* | `usage=` *usagespec* | *key* `=` *attrib* | `renaming=` ( `on` | `off` ) | `value=` *hexval* | `ilpres=` `{` *resourcelist* `}`
  - The attribute `size` describes the bitsize of the register.
  - The attribute `type` describes the datatype of a register. Possible values of *typespec* are `signed`, `unsigned`, `float`, `fract`, and `storage`. The width of the type can optionally be specified by `<`*ival* `>` for `signed, unsigned, storage` and for `float` by `<`*ival1* `,` *ival2* `>` In case of floating point values, *ival1* denotes the size of the mantissa, *ival2* the size of the exponent. These types are identical to the types which are feasible in the RTL language for specifying semantics of operations. If no type is specified, by default the type `storage` is assumed; this denotes a storage location of unspecified type. If no size is explicitly given for the type, the size of the register is used as default width. Additional types can be defined via `ExtendValueSet` (see Sec. 3.7).
  - If a register is hardwired to a fixed value, this value is given by the attribute `value`. The register contents always have to be given in hexadecimal notation (*hexval*). The bit size of the constant (i.e. the number of bits used to represent this constant) must be identical to the bit size of the register.
  - The attribute `usage` describes the (semantical) use of the register introduced. Possible values of *usagespec* are `Index` (index register), `Offset` (offset register), `SP` (stack pointer), `FP` (frame pointer), `PC` (program counter), `ACCU` (accumulator register). Additional values can be associated to `usage` via `ExtendValueSet` (see Sec.3.7).
  - Sometimes register renaming should be performed during code optimizations. However there are also cases where no register renaming is allowed (`renaming = off`). This is the default case; if register renaming is to be performed, `renaming = on` has to be specified.
  - The attribute `ilpres` requires more detailed explanation. During code optimization in the PROPAN framework, integrated instruction scheduling and register assignment is performed. Consider a register set which can be subdivided into smaller register groups with a different behaviour e.g. with respect to parallelization properties. If an operation is read in from the input program, each operand which lies in a register of this register file must be mapped to the resource nodes of all register groups in that register file. This is required to model the consequences of assigning that operand to a register of a certain

register group. These subgroups can be declared as resource classes consisting of the appropriate element registers. Since the behavior of the different groups is different, using the complete register file containing the original operand is not sufficient for a correct modelling. *resourcelist* is a comma-separated list whose components have the form *genname RangeSpec*. The referenced resources may be register files, single registers, a subset of registers from a register file, or resource classes introduced for register resources. (*RangeSpec* is defined as above).

The value of `ilpres` will have no effect, if no register renaming is allowed for the current register set.

The attributes can be given in any order. Apart from the attributes `type` and `size`, all other attributes are optional. In addition to the predefined attributes, the user can use additional attributes to describe the registerset; these attributes are given in the form of key/value-pairs. All attributes have to be declared before they are used by `DefineAttribute` (see below).
Example: `Register indreg "i%d" [0:15] size=40, type=signed, usage=Index;`

The attributes of a register file, parts of a registerfile or of individual registers in a register file can be changed explicitly after the definition by `Register`. For this, the `SetProperties` declaration is used.

`SetProperties` *genname RangeSpec RegSpecList* ;

- *genname* is the generic name of the register file which is affected by the change.
- *RangeSpec* is syntactically defined as above. However the meaning is slightly different. If no range delimiter is given at all, then the attributes given in *RegSpecList* will be transferred to the complete register file. If *RangeSpec* is of the form `[a]`, then the attributes are changed only for the `a`th component register of the register file. If *RangeSpec* is of the form `[a:b]`, then all element registers whose indices are in this range are affected by the attribute definition.
- *RegSpecList* is the same as above.

## 3.2 Declaration of Functional Units

`FuncUnit` *genname asmname* $\underline{?}$ *repspec* $\underline{(}$ *key* `=` *value* `,` $\underline{)}$ $\underline{*}$ $\underline{(}$ *key* `=` *value* $\underline{)}$ $\underline{?}$ ;

- Here, the assembler template *asmname* can be omitted, if the functional resource is never named in the assembler language.
- *repspec*: *intval* $\underline{|}$ $\epsilon$
  *repspec* indicates how many instances of that functional unit exist. If no replication level is specified, it is assumed that there is exactly one instance.

Examples:

- `FuncUnit ALU 3;`
- `FuncUnit Shifter;`

## 3.3 Declaration of Memories

`Memory` *genname asmname RangeSpec MemSpecList* `;`

- *RangeSpec*: $\epsilon$ | `[` *intval* `]` | `[` *intval* `:` *intval* `]`

  *RangeSpec* is used to specify the range of memory cells contained in the declared memory. If no range is given, there is no information about the size of the memory. Giving only one value is feasible; however this would cause a memory consisting of only one cell to be declared. By a complete range specification as, e.g., by `[0:1023]` a 1024-Bit Memory would be declared. The flexible syntax supports memories embedded in a larger address space.

- *MemSpecList*: ( *MemSpec* `,` ) * *MemSpec*
- *MemSpec*: `type =` *MemTypeQual* | `align =` *intval* | `access =` *intval* | *key* `=` *value*
  - `type` describes the type of the memory being declared. Possible values for *MemTypeQual* are `Data` denoting a data memory, `Instr` denoting a program memory, and `Mixed` which declares a memory which can be used to store both instructions and data. Additional types can be defined via `ExtendValueSet` (see below).
  - `align` describes the alignment of the memory in bits.
  - `access` describes the access width of the memory in bits.

Apart from the predefined attributes (`type`, `align` and `access`), the user can define additional attributes; these attributes are given in the form of key/value-pairs. All attributes have to be declared by `DefineAttribute` (see below) before they are actually used. The attributes can be given in any order; `type`, `align` and `access` have always to be specified.

Example: `Memory PM "pm" type=mixed, align=32, access=32;`

## 3.4 Declaration of Caches

`Cache` *genname CacheSpecList* `;`

- *CacheSpecList*: ( *CacheSpec* `,` ) * *CacheSpec*

- *CacheSpec*: `type =` *MemTypeQual* | `size =` *intval* | `assoc =` *intval* | `linesize =` *intval* | *key* `=` *value*
  - `Type` is the same as in memory declarations; so is the meaning of *MemTypeQual*.
  - `Size` gives the number of cache lines. From this and the line size of the cache, the absolute size of the cache can be calculated.
  - `Assoc` gives the associativity of the cache.
  - `Linesize` specifies the linesize of the cache in bytes.

Again, additional attributes can be specified by the user in form of key/value pairs. All user-defined attributes have to be declared by `DefineAttribute` before they are actually used. The attributes can be given in any order; `type`, `size`, `assoc` and `linesize` have always to be defined.

Example: `Cache InstrCache assoc=2, size=192, linesize=48, type=instr;`

## 3.5 Declaration of Generic (User-Defined) Resources

`DefineResource` *genname asmname GenSpecList* `;`

- *genname*: as usual.
- *asmname*: as usual.
- *GenSpecList*: ( ( *key* `=` *attrib* `,`) * *key* `=` *attrib* ) ?

  A sequence of key/value pairs can be given to describe the generic resource. All attributes have to be declared via `DefineAttribute` before they are actually used.

Example: `DefineResource Myresource "res" key1=value1, key2=value2;`

## 3.6 Declaration of Attributes

`DefineAttribute` *genname* `{` *valuelist* `}` `associated to` *restypelist* `;`

- *genname* is the name of that attribute which is used in the declaration of resources.
- *Valuelist*: ( *Valuelist* `,` *Value* ) * *Value*
- *Value*: `Int` | `Float` | *intval* | *fltval* | *strval* | `[` *intval* `:` *intval* `]`
  - `Int` denotes an integer type.
  - `Float` denotes a floating-point type.

- *intval* is a concrete integer value.
- *fltval* is a concrete real value.
- *strval* is a string enclosed in `"`'s.
- `[` *intval* `:` *intval* `]` is an interval of real numbers.

- *restypelist*: ( *restypelist* `,` *restype* ) * *restype*
  - *restype* can be `Register`, `Memory`, `FuncUnit`, `Cache`, `Operation` and *genname*.
  - Here, *genname* refers to a user-defined resource type.
  - Since attributes can also be defined for `Operation` (appears in the instruction set section), a dedicated resource type has been introduced for operations (instructions): `Operation`.
  - Since user annotations have to be given in the form of attribute/value pairs, the resource type `Annotation` is used to declare attributes used in user annotations.

Example: `DefineAttribute Replacement { "LRU", "FILO" } associated to Cache;`

## 3.7 Extensions of the Domains of Predefined Attributes

- `ExtendValueSet type for Register by {` *Valuelist* `}`
- `ExtendValueSet usage for Register by {` *Valuelist* `}`
- `ExtendValueSet type for Memory by {` *Valuelist* `}`
- `ExtendValueSet mode for Operation by {` *Valuelist* `}`
- `ExtendValueSet guard for Operation by {` *Valuelist* `}`

*Valuelist* is the same as shown in the declaration of attributes.

## 3.8 Definition of Aliases for Registers and Register Sets

The general idea is to introduce an alternative name for certain register files or parts of register files. This way, different logical views of the same physical register set can be specified. The aliases can differ from the original resource with respect to their attributes. Attributes which differ from the original resource have to be explicitly given in the alias-definition, all attributes which are not given are taken from the original resource. An alias definition must always occur after the definition of the aliased resource. If a register file $A$ is declared as an alias for a previously declared register file $B$, the alias relationship is registered in $A$ and $B$ itself, and in each element register. Depending on the type of the alias mapping, each element register of $A$ corresponds to one or more element registers of $B$, or to some part of an element register of $B$.

`RegisterAlias` *AliasName AliasAsmName OrigName RangeSpec RegMap OptRegSpecList* ;

- *AliasName* is the *genname* of the introduced alias.
- *AliasAsmName* is the assembly representation of the alias.
- *OrigName* is the genname of the resource which is aliased.
- *RangeSpec* is the syntactically the same as for the declaration of register sets. However, if it is of the form `[a]`, then the alias refers exclusively to that element register of register file *genname* which is addressed by index `a`. If a complete range is specified, then the alias refers to all element registers within that range; if no range is given, then the alias is an alias for the whole register file.
- *RegMap*: `Mapping = [` *intval* : *intval* `]`

  Sometimes an alias of a register introduces registers which are composed of several registers of the original registerfile or which are parts of the original registers. In these cases, the mapping indicates how many registers of the old resource correspond to one register of the new resource (the alias). If the mapping is `a:1`, then one register of the alias corresponds to `a` successive registers of the original resource. If the mapping is `1:b`, then `b` successive registers of the alias correspond to one register of the original resource. One of the *intval*'s has to be equal to one.
- *OptRegSpecList* mostly corresponds to *RegSpecList*. However in contrast to Sec.3.1, it can be empty and an additional attribute `numbering` can be given. Feasible attribute values of `numbering` are `{auto, oinc, odec}`; the detailed meaning is explained below.

A frequent application of register aliases is the modelling of register files whose element registers are composed from several element registers of another, previously declared register file. An example are double registers which are composed from two integer registers. The assembler representation of the alias register is generated automatically; by default, the element registers are numbered in increasing order. Consider the following example:

```
RegisterAlias Doublereg "d%d" ireg mapping=[2:1], type=double;
```

Then the double register `d0` corresponds to the integer registers `r0,r1`, the double register `d1` corresponds to the integer registers `r2,r3`, etc.

However the naming scheme of the individual registers can be more complicated. An example is the Texas Instruments TI C6x. In this processor two general purpose registers can be grouped to form one long register. The assembly representation of the long registers corresponds to the concatenation of the assembly representation of the comprised general purpose registers. The consequence is that in the assembly representation specification of each individual register more than one placeholder `%d` is used. In such cases, the correct numbering can be enforced with the help of an additional attribute `numbering` which can take the values `oinc`, `odec` and `auto`. The value `oinc` indicates that the numbers in the assembly representation of each register appear in increasing order. Similarly `odec` is used to specify decreasing order. The default value `auto` covers the cases when only one placeholder is needed. The register declarations of the long registers of the TI C6x reads as follows:

```
RegisterAlias Longreg "A%d:A%d" ireg mapping=[2:1], type=double, numbering=odec;
```

The assembly representation of the element registers of the register file `Longreg` then is `A1:A0` for the long register composed of the general purpose registers `A0` and `A1`, `A3:A2`, etc.

## 3.9 Definition of Aliases for other Resources

- `BitAlias` *BitName BitAsmRep OrigName RangeSpec* < *BitNumber* >
- `FuncUnitAlias` *AliasName OrigName GenSpecList*
- `MemoryAlias` *AliasName AliasAsmRep OrigName RangeSpec MemSpecList*
- `CacheAlias` *AliasName OrigName CacheSpecList*
- `GenericAlias` *AliasName AliasAsmRep OrigName GenSpecList*

A special case of register aliasing is the introduction of dedicated names for individual bits of a register; additionally alias relationships can be declared for functional units, memories, caches, and to all user-introduced resources. The replication level of functional units cannot be modified in an alias definition. If an alias has to be introduced for exactly one instance of one functional, an index has to be given to identify the correct instance using the *RangeSpec* syntax. Similar to the register alias mechanism the alias relationships are registered and in the aliases and in the original resources.

Examples:

- `FuncUnitAlias meineALU ALU;`
- `MemoryAlias meinDM "mdm" DM;`
- `CacheAlias meinCache InstrCache size=4999;`

## 3.10 Preprocessing

Before the machine description is parsed, it is fed to the `cpp`-preprocessor. Thus, in order to shorten the machine description, the `#define` mechanism known from C can be used. Similarly the `#include` directive can be used to distribute the machine description over several files.

## 3.11 Definition of Resource Classes

Often, some resources have to be grouped, e.g., in the description of the instruction set of the target processor. These resource groups are no aliases, since they are just a kind of abbreviation for a set of different resources.
`ResourceClass` *genname* `{` *gennamelist* `}`

- *genname* denotes the generic name which is used to refer to the resource class.
- *gennamelist* denotes the set of resources which are elements of the defined group. All resources contained in that set must have been defined before the `ResourceClass` definition.
- Example:

  ```
  ResourceClass ureg {ireg,freg,indreg,mreg,breg,lreg};
  ```

# Chapter 4

# The Instruction Set Section

In this section the instruction set of the target processor is described together with its assembly representation. The instruction set is given in the form of an attribute grammar; each operation definition can be considered a rule of this grammar. Operation parts as, e.g., addressing modes or operands, can be described by individual rules. The operations of the input file are then parsed according to the given rules; the properties of the operations are derived from the attribute definitions associated with each rule. There are several separated attribute sets; the first set is used to identify the operands of the instruction and some important general informations about the operation execution. Examples are the `dst` attribute which is used to describe the primary destination of an operation, the `base` attribute used to describe the base value for indirect memory references, etc. This attributes play also an important role for the specification of the semantics of an operation (see Subsec. 4.1). The next attribute set is used to describe the assignment to ILP resources in the integer linear programming model and the reservation table information needed for pipeline modelling. The third attribute set deals with the timing information; the last attribute set can be considered as a single attribute explicitly specifying the detailed semantical behavior of the operations by means of a register transfer language (4.1).

The following subsection gives a short summary of the basic concepts of the semantics specification; then the syntax of the instruction set definition is given in detail. The last subsection introduces the register transfer language of TDL.

## 4.1 The Specification of the Semantics

When describing operation semantics the description formalism depends on the intended use. If e.g. the purpose of a semantical specification is only to generate hardware simulators a very powerful method is to give the semantics in `C` [5]. However this way, the semantical specification cannot be used as a base for code analyses and optimizations.

An important design goal of the semantics description formalism of TDL was to achieve a high degree of flexibility. When the bit-level behavior has not to be known for the target application it should be feasible to only describe that part of the operation semantics which are really needed. On the other hand, if bit-level exactness is required an exhaustive description must be possible. It is common practice to specify the semantics of an operation by using a register transfer language (RTL). In TDL an extended RTL language is used whose syntax has been adapted to the C-language. Similar RTL languages have been used within nML [6] and ISDL [8]. Restricting the language to pure register transfer operations would lead to very long semantical specifications, thus higher order operations as, e.g., loops or function calls have been introduced.

The semantics of an operation is defined by a sequence of statements which are the basic entities of the TDL-version of RTL. Possible RTL statements are assignments, conditional expressions, loops and function calls. Each RTL statement describes a change to the state of the target architecture; source and destination operands are either constants or refer to storage resources of the processor. Therefore, the type of all operands referring to storage resources is a bit sequence of the appropriate length. Each resource used in the RTL specification has to be declared in the resource section. RTL expressions include logical and arithmetic operations, accesses to individual registers of register files and to individual memory cells. It is also possible to access single bits of a storage resource. In order to allow for a concise semantics specification it is often useful to use temporary storage resources which do not exist in the real hardware but simplify the description of the semantics. Thus there is a mechanism to declare virtual storage resources; these consist of a sequence of bits of a specified length. The feasible constants are integer values in decimal, hexadecimal and binary format as well as floating point numbers.

The attribute mechanism introduced above plays an important role in distinguishing different levels of semantical detail. When dealing with data dependences it must be known if an operation can change the value of a condition register. But how these bits are changed is not necessarily interesting on that level. On the other hand if an analysis is performed that deals with conditional instructions, the exact behavior is essential. The standard passes to calculate the data dependences and construct the control flow graph refer only to that components of the machine state which are described by the explicitly given attributes of the operations. User-defined sophisticated analyses requiring more detail have to access the complete semantical specification in RTL language and interprete the given behavior. Of course, if only standard analyses have to be performed, the detailed description is not needed and therefore can be omitted. In ISDL a similar distinction is made by differentiating between the semantics proper and the so-called side effects of an operation.

Often instructions have a quite complex behavior; if this behavior would have to be specified only by means of RTL expressions this would lengthen the description considerably. Thus a function call mechanism is required to modularize the specification of the semantics of the operations. The concept of functions used within RTL is slightly different from the functions as used in programming languages. Since the behavior of hardware resources has to be modelled, there is no need for scoping and advanced parameter passing mechanisms. All arguments passed to functions can be thought of bit sequences; these can be evaluated or modified within the functions. Yet the function calling mechanism plays an important role in generating RTL descriptions: apart from its role in modularizing the description it allows to interprete one sequence of RTL statements with different levels of abstraction. The primary application area of TDL-specifications are postpass analyses and optimizations, but in the future also the automatic generation of simulators or high-level code generators should be supported. Each of these applications differs from the others by the required level of detail in the RTL description. If the bit-level behaviour of a function body is not of interest for the target application, it is possible to consider the function as a black box. When generating a simulator for the described architecture, the function body would have to be considered in full detail. There is a set of predefined functions as e.g. `_sext` (sign extension) which can be considered as atomic switch functions . It is also possible to introduce user-defined "canonical" functions. In the current version of TDL, the canonical functions are treated as black boxes, i.e. there is no function body. However in order to allow for an automatic consistency check of the RTL description, each canonical function must be explicitly declared. This is done by providing a prototype specifying number and type of arguments as well as the type of the return value. Thus, the semantics is implicit in the function call; the target applications are responsible for the correct interpretation. Of course, if, e.g., the program execution is to be simulated using the RTL description then the function body must be defined explicitly. This will be included in a future version of TDL.

## 4.2 Declaration of an Assembly Operation

The description of the instruction set begins with the keyword `Instructionset-Section`. The syntax of an instruction declaration is as follows:

`DefineOp` *genname asmrep* `{` *attriblist* `} , {` *restable* `} , {` *semantics* `} ;`

- *genname*: This is the generic name of the instruction to be defined. By this name, the instruction can be referenced in the following sections of the hardware description. The name must consist of a letter followed by a possibly empty list of alphanumeric characters.

- *asmrep*: `"` ( *symbol* | *formatsymbol* ) * `"` This is a format string which defines the assembly representation of the instruction.

  - *symbol* can be any symbol except `"` and `%`. If a `%` is to be treated as a character, it has to be expressed by the format symbol `%%`.
  - *formatsymbol*: `%d`: integer in decimal notation, `%x`: integer in hexadecimal notation, `%f`: double, `%s`: string. The directive `%!(ip)` refers to a non-terminal named `ip`. Each non-terminal can be viewed as a function which sets some attribute values and returns the assembly representation of the matched partial string.

  Example usage: Consider the format string `"dm(%s,%s)=%6d"`. With the help of the attributes, the user can specify that the two `%s` directives are to be substituted by the assembly representation of some hardware resources. If the attribute specification `dst="$1" in {ireg}` is given, and `ireg` has been declared as a registerfile with assembly representation `r%d`, then this is interpreted as follows: the first `%`-directive (`$1`) corresponds to the assembly representation of a resource of type `ireg`. Then a match would occur, e.g., for `r3`. Moreover it is known that this resource is used as destination for that operation. If the attribute specification reads only `dst="$1"`, then any string following `"dm("` would match. The format string `"%6d"` describes that any 6 bit integer is expected in this instruction position as an immediate value.

- *attriblist*: ( *attrspec* , ) * *attrspec*

  In *attriblist* all attributes related to the syntactical appearance of the operation are specified. The attributes defined here represent the semantical information used in the standard passes to construct the control flow graph and the data and control dependence graphs.

  - *attrspec*: *iattr* = *iattrvalue* | `noreorder`
  - *iattr*: `dst` | `base` | `offset` | `mode` | `guard` | `src1` | `src2` | `src3` | `target` | `elsetarget` | `termcond` | `counter` | *genname*

    For each attribute, there can be at most one definition.

    `noreorder` : explicitly prohibits any code movement across the actual operation.

    `dst1` : primary destination of the operation.

    `dst2` : secondary destination of the operation.

    `base` : is used as base address in address calculations.

    `offset` : is used as offset in address calculations.

    `mode` : an operation mode; predefined values are `post` denoting a post-modify memory access and `pre` denoting a pre-modify memory access. In both modes the base register is assumed to be modified. Additional values can be defined by the user.

    `guard` : this attributes indicates if the execution of the operation depends on special conditions which are determined dynamically by the actual state of the machine. Possible values are `true` and `false`.

    `src1, src2, src3, src4, src5` : the resources which are read by that operation.

    `target, elsetarget` denote jump targets.

    `counter` : this attribute can be associated to counter-based loop instructions; it refers to the location of a loop counter register.

    ***genname*** refers to a user-defined attribute which must have been introduced via `DefineAttribute` and associated to `Operation`.

- *iattrvalue*: *genname* | *genname* `[` *intval* `]` | *predefvalue* | `"$`*digit* `"` ( `in` *resourcelist* ) ? *OptDType*

  *genname* represents a reference to a resource which has to be defined in the Resources Section. It is also possible to specify a dedicated element register of a register set as operand of the instruction. Then the index of that register within the enclosing registerfile has to be given in brackets. The set of predefined values contains, e.g. the value `post` for the attribute `mode`. The user can specify additional values using the `ExtendValueSet` directive. The sequence `"$1"` means that the corresponding value depends on the actual instruction instance (when reading in the instruction from an assembly file) and corresponds to the first placeholder in the assembly representation. By `in` *resourcelist*, it is specified in which resource types the corresponding argument has to lie. The extension `in` *gennamelist* can also be omitted. If the `%`-directive refers to a non-terminal the argument type is the return type of that non-terminal (see below). Otherwise the argument defaults to the data type which corresponds to the format string, i.e. if the corresponding argument appears as `%s`, then it is a string, a `%d` denotes an integer, a `%f` a float and so on.

- *OptDType*: `as` *datatype* | $\epsilon$: Some instructions interprete their arguments always as containing dedicated data types. This can be specified by using the keyword `as`; the feasible types are summarized in sec 4.3.

- *restable*: *ExecutionAlternatives* ; *WritebackAlternatives*

  The execution model of TDL differentiates between two resource types, an operation can be assigned to: first the resource which actually executes the operation, and second the resource reponsible for the write back of the result. By specifying the execution resource, it can be modelled that the number of available instances of each resource type is not exceeded. Similarly, by specifying a write-back resource, it can be assured that the maximum number of results which can be written to the result bus per cycle is not exceeded. In *ExecutionAlternatives* all resource types which can be used to execute the current operation are given; *WritebackAlterantives* enumerates the alternative resource types for the write-back resource of the current operation. Since the specification of a write-back bus is not necessary for all architectures; *WritebackAlternatives* can be empty.

  - *ExecutionAlternatives*: ( *ExecutionAlternatives* `|` ) * *ReservationSpecification*
    *ReservationSpecification*: *genname* *OptTimingSpec*
    *OptTimingSpec*: `(` *TimingList* `)` | $\epsilon$
    *Timing*: `Latency =` *digit* `, Slots =` *digit* `, ExecutionTime =` *digit*

    Since the timing behaviour of an operation may depend on the resource by which it is executed, the timing information can be specified separatedly for each resource. The order does not matter; if no timing informations are specified, a default value of 0 is assumed.

  - *WritebackAlternatives*: $\epsilon$ | *ExecutionAlternatives*
    Here, the resources which are resonsible for coordinating the write back of the results can be enumerated. Typically, a result bus will be declared here. Since in some cases, bus contention of result write backs are already prevented by the modelling of the executing resources (e.g. for the ADSP-2106x), this entry can be empty.

- *semantics*: see subsections 4.1 and 4.3

### 4.2.1 Example

In this example it is assumed that a non-terminal `optguard` has already been defined which sets the guard attribute and introduces a temporary storage location `cond`. The last part of the operation definition is the specification of the semantics of the operation (see 4.3).

```
DefineOp ComplexInstr "%!(optguard)%s = %s + %s" {
    dst1="$1" in {ireg}, src1="$2" in {ireg}, src2="$3" in {ireg}},
    {ALU1(latency=1, slots=0, exectime=1)|ALU2(latency=1, slots=0,
     exectime=2); WbBus(latency=1)},
    {if (cond==1) {dst1 = src1 + src2;} };
```

## 4.3 The Register Transfer Language

The RTL description of the semantics consists of a sequence of statements. There are 8 different types of statements: assignments, conditional statements, for- and while-loops, switch statements, declarations of virtual temporary storage location and function calls.

An assignment is used to change the state of the machine; the value of the expression on the right hand side is assigned to the storage resource denoted by the left hand side. The left hand side can be a reference to a complete memory or register file as well as to an individual memory cell or an individual register of a register file. It is also possible to access any bit of these storage resources separately. In order to cope with individual operation instances, also the operand attributes (`dst, base, offset, src1` ...) can be given here.

The basic data type used in RTL expressions are bit sequences of a given length. For this the keyword `storage` is used; it denotes a storage location which (in the actual context) is not associated to a concrete arithmetic data type. However, if the contents of a storage location have to be evaluated in the context of the RTL specification, the resulting value depends on the numeric representation. In many cases the numeric representation which is used to evaluate a storage location depends on special conditions or mode settings. E.g. the contents of a fixed-point register might be interpreted as signed integer in two's complement representation, as unsigned integer or as fractional. Floating point formats differ in the number of bits dedicated to the mantissa and exponent. The required format always must be known when such an evaluation has to be performed; otherwise the semantics would not be well-defined. Each TDL-description is checked for semantical consistency. It is checked that the arguments of all RTL-operations have the correct data type and the correct bit width. This does not mean that RTL is strongly typed, since this often would represent an obstacle for describing the machine semantics. If no arithmetic data type is required, the type `storage` is used which simply represents a sequence of bit values.

The following data types are feasible in TDL:

- `signed<n>` denotes an $n$-bit signed integer in two's complement representation.
- `unsigned<n>` denotes an $n$-bit unsigned integer.
- `float<m,n>` denotes an IEEE floating point value with an $m$-bit mantissa and an $n$-bit exponent.
- `fract<m,n>` denotes fractional values.
- `storage<n>` denotes a storage location of length $n$ with unspecified type.

For each of these data types (except from `storage`), dedicated evaluation functions are predefined which ensure that the argument is evaluated in the correct format. They differ from the "normal" functions which are used to describe the functionality of hardware components (see below). The evaluation functions are used to make explicit in the semantical description how the contents of a storage location or a constant are evaluated but they are not associated to a dedicated physical hardware component.

In the following, the evalation functions and the set of predefined functions are explained in detail. For each function, its prototype is given, similarly as it is required within a TDL-specification when introducing canonical functions. However there is a difference: in contrast to the prototype specification mechanism of TDL, all arguments of functions are named within this chapter. If the type of an argument does not matter, the keyword `anytype` is used. This subsumes all the data types mentioned above (`signed, unsigned, float, fract, storage`). The meaning of `anytype` is very similar to the meaning of `storage`. However while the keyword `storage` is used to describe a storage location with unknown or unspecified data type, `anytype` denotes the set of all possible data types including `storage`. The effect on the evaluation is the same: in both cases, the contents of the storage location can be interpreted as element of any arithmetic data type. Some functions expect their arguments to be of a specified type; then this type is explicitly given in the function prototype. When introducing a formal parameter of type `storage`, `signed` or `unsigned`, no size has to be specified. In this case, the size of the argument does not matter. Since all arguments are bit sequences which are passed to the functions, their length is always known, so that a correct evaluation is possible. Function calls are typechecked with respect to the number of arguments, their data types and their bit widths.

The following evaluation functions can be used:

- `_signed(anytype s, signed n)`: The contents of $s$ are interpreted as two's complement value of length $n$. If the bit width of storage resource $s$ is larger than $n$, only the least significant $n$ bits are considered. The second parameter can be omitted; then the entire storage resource is considered in full length.
- `_unsigned(anytype s, signed n)`: The contents of $s$ are interpreted as unsigned value of length $n$. If the bit width of storage resource $s$ is larger than $n$, only the least significant $n$ bits are considered. The second parameter can be omitted; then the entire storage resource is considered in full length.
- `_float(anytype s, signed m, signed n)`: The contents of $s$ are interpreted as floating point value with an $m$-bit mantissa and an $n$-bit exponent.
- `_fract(anytype s, signed m, signed n)`: The contents of $s$ are interpreted as fractional (fixed point) values; however the concrete representation has not been determined.

In the following example, the contents of the integer register $i4$ is evaluated as a two's complement number if a bit in a status register `ASTAT<3>` is set to one, otherwise, it is interpreted as an unsigned value.

```
if(ASTAT<3>==1) {if (_signed(i4)>1) {...}
                 else {if (_unsigned(i4)>1) {...})};
```

The control structures `if, for, while, switch` behave in the same way as the corresponding C operations. Temporary virtual storage resources can be introduced; they are invariably associated with the indicated data type. The bit sizes are enclosed in brackets ('`<>`'); e.g. the statement `signed<32> iterator;` introduces a temporary storage location with name `iterator` that is used to hold 32-bit two's complement values. Due to the structure of the instruction set section (attribute grammar), the complete RTL description of an operation instance can depend on some parts of the instruction described by non-terminals. Then the semantical description of that operation part is not contained in the RTL sequence of the main part of the operation. If the semantical description of a non-terminal contains a declaration of a temporary storage location which is used in the main part of the specification, it is necessary for this temporary storage location to be declared in the main part of the RTL description via the keyword `external`. There is a set of predefined functions; the naming convention is such that these functions always are named with a leading '`_`'. Additional user defined functions can be declared and used (see Subsec. 4.3.1).

The RTL expressions are defined as follows:

- Constants. These include
  - integer constants given in decimal notation. These constants represent 32-bit two's complement values. By using the special evaluation functions (e.g. `_unsigned()`) a representation in other formats can be forced as well.

- binary and hexadecimal values. If an evaluation of such constants is needed, by default they are interpreted as unsigned values. By using the evaluation functions they can also be interpreted as numbers of different format. If the constant `0xFF` has to be interpreted as two's complement representation of $-1$, this can be achieved by writing `_signed(0xFF,8)`. A more detailed description of these evaluation functions can be found later in this chapter.
- floating point values, e.g. `1.235`. By default, they are interpreted as 32-bit IEEE floating point numbers, but by using the evaluation function `_float`, any floating-point representation can be forced.
- Some predefined constants which represent unary operators in the RTL language. There are four different two-bit constants to denote the floating-point rounding mode :
  * `round_down` ($0b00$): IEEE 754 mode for rounding down.
  * `round_up` ($0b01$): IEEE 754 mode for rounding up.
  * `round_nearest` ($0b10$: IEEE 754 mode for rounding to nearest.
  * `round_zero` ($0b11$): IEEE 743 mode for rounding toward zero.

• Storage references. These include references to complete memories or register files as well as to individual memory cells or individual registers of a register file. It is also possible to access any bit of these storage resources separately. This is done by appending the position of the desired bit enclosed in `<>`. If a storage resource appears within an expression the contents of this location are evaluated; then the type of the data contained in it must be known. If it appears on the left hand side of an assignment, the storage area itself is denoted.

• Field names. Feasible values are the names of all attributes which can appear in the first attribute set of an operation description. (`dst, src1, src2, src3, src4, src5, base, offset, counter, node, type, guard, target`, as well as additional user-defined attributes)

• Attribute names and values, if they are associated with the resource type `Operation` (see Sec. 4). Since the user can introduce additional attributes and define a set of feasible values for each attribute, it must be possible to refer to these attributes in the semantical description. However, the set of operations which can be performed with user-defined attributes and their values is restricted: only comparison operations are allowed. The reason for this is that these attributes are not associated to storage resources; therefore there is no need to perform assignments to such attributes within the RTL description. However, the operation semantics may depend on the value of such attributes. This restriction also holds for user-provided extensions of the predefined attributes `guard`, `mode`, `type`.

• Binary arithmetic expressions; the following operators are feasible:

- The logical conjunction (`&&`) and disjunction (`||`) operators. The operands must be 1-bit unsigned values and the return type is a 1-bit unsigned value, too.
- Bitwise and, or and xor (`&`, `|`, `^`). The bit width of the operands must match; the return type results from synthesising the types of the operands (see Subsec. 4.3.2).
- Equality and inequality test operators (`==`, `!=`) These operators can be used with operands of any type; the operands are compared bitwise. The result type is `unsigned<1>`. When using these operators for floating-point numbers, no floating-point exceptions are registered. If exceptions have to be modelled, the dedicated floating-point comparison operators (`feq, fneq,` ...) have to be used.
- Arithmetic comparison operators (`<`, `>`, `<=`, `>=`). The arguments of these functions must have numerical type, i.e. they must be of type `signed, unsigned` or `float`. If the types of the operand differ, an automatic type conversion is performed. The result is a 1-bit unsigned value. When using these operators for floating-point numbers, no floating-point exceptions are registered. If exceptions have to be modelled, the dedicated floating-point comparison operators (`feq, fneq,` ...) have to be used.
- Right and left shift (`<<`, `>>`). The right operand, which must be an unsigned integer value, gives the number of bit positions the left operand has to be shifted. The type of the return value is the type of the first operand.
- The usual arithmetic operations (`+`, `-`, `*`, `/`, `%`). The operands must be numerical values; any combination of numerical types is feasible. So, these operators are overloaded, however when using them it must always be clear which format the operands have. If the format of the operands depends on the instruction itself, it can be specified by using the `as` extension to the attribute declaration (see 4.2), e.g., `src1="$1" in {ireg} as signed<32>`. If the format of the operands depends on runtime conditions as e.g. special mode settings, the operation format must be made explicit by using the evaluation or conversion functions (see above). The operators can be used with operands of different types. The following automatic conversions take place:
  - $signed, unsigned \longrightarrow signed$
  - $signed, float \longrightarrow float$

  For fixed point operands, the bit width of the result is the maximum of the bit widths of the operands. For floating point values, both the sizes of mantissa and exponent of the return value correspond to the maximum of mantissa resp. operand of both operands. If the bit width available for the destination resource is too short for the exact result, it is truncated by default. If this is not desirable, the modelling has to be explicitly specified by RTL code or by a dedicated RTL-function.

- Unary arithmetic expressions:
  - Logical complement (`!`). The operand has to be a 1-bit unsigned value; so is the return type.
  - Bitwise complement (`~`). The operand type is used as return type; any type is allowed.
  - Unary minus (`-`). Here the same conditions hold as for the binary arithmetic operators.

- Calls to predefined as well as user-defined functions. The difference consists in the function naming: the names of the user-defined functions are given as strings i.e. they are enclosed in `"`. E.g., a predefined function is the sign extension function `_sext`; an example of a user-defined function name is `"user_defined_fun"`. The parameter list is enclosed in round brackets `()`; the local parameters are separated by commas. In the following, the same notation is used as above, however the type of the return value precedes the function name. In most cases a (virtual) storage resource is returned whose contents can be assigned to other resources or which can be manipulated itself (i.e. it can be used either on the left or on the right hand of assignments). Some functions do not produce results; this is denoted by the keyword `void`, taken from `C`. If an actual parameter of a function is omitted, the keyword `nil` can be given. This is helpful when using the predefined functions if some flag settings are not necessary for the target architecture.
  - `void _include(n)` function which takes the generic name of a non-terminal as single argument. Within the instruction set definition, non-terminals can be used which may carry partial semantical information in the form of RTL statements. If the RTL sequence for the complete operation is constructed the RTL statements of a non-terminal $n$ are introduced in that place within the RTL sequence of the whole operation where the statement `_include`($n$) is located. By default, the RTL statements of non-terminals are inserted at the beginning of the operation's RTL-sequence in the order of their occurrence.
  - `void _exec(storage)`: The argument must always be the program counter; it executes the instruction whose address is contained in the PC. The PC is not incremented automatically to allow for explicit assignments to the program counter, e.g., by jumps.
  - `storage _msb(anytype s, signed n)` extracts the $n$ most significant bits of storage resource $s$ (big endian notation) and returns this bit sequence [1].
  - `storage _lsb(anytype s, signed n)` extracts the $n$ least significant bits of storage resource $s$ (big endian notation) and returns this bit sequence [1].

[1]Subsequent changes to the returned bit sequence also change the state of the storage resource passed as first argument.

- `storage _bext(anytype s, signed m, signed n)` extracts the bit sequence from the $n$th to the $m$th bit of $s$ and returns it [1].
- `storage _conc(anytype s1, anytype s2)` concatenates the bit sequences contained in the arguments $s1$ and $s2$. The result is a bit sequence whose length is the sum of the lengths of both arguments.
- `signed _sext(signed s, signed n)` performs a sign extension of the value $s$ to $n$ bits. The return value is always a storage resource containing a two's complement number [1].
- `signed _zext(signed s, signed n)` performs a zero extension of the value $s$ to $n$ bits. The return value is always a storage resource containing a two's complement number [1].
- $T$ `_iadd` ($T_1$ $s_1, T_2$ $s_2$, `unsigned<1>` $t$, `unsigned<1>` $m$, `storage<1>` $o$)

  This function adds the contents of $s_1$ and $s_2$ interpreted as two's complement numbers if $t == 1$ and as unsigned values if $t == 0$. The feasible data types for the operands are $T_1, T_2 \in$ { `storage, signed, unsigned` }. Here also the unspecified type `storage` is feasible, since the other parameters of the function determine uniquely how the arguments of the function are to be interpreted. However an error is reported when the storage location passed as argument has previously been specified as location of a floating point value. The result type $T$ is determined according to table 4.3.2. If $m == 1$, saturation is performed; then in the case of positive (negative) overflow the largest positive (smallest negative) number of the given bit width is returned. The bit width of the result is the maximum of the bit widths of $s_1$ and $s_2$. If an overflow occurs, the bit $o$ is set to 1.
- $T$ `_isub` ($T_1$ $s_1, T_2$ $s_2$, `unsigned<1>` $t$, `unsigned<1>` $m$, `storage<1>` $o$)

  This function subtracts the contents of $s_2$ from $s_1$ interpreted as two's complement numbers if $t == 1$ and as unsigned values if $t == 0$. If $m == 1$, saturation is performed; then in the case of positive (negative) overflow the largest positive (smallest negative) number of the given bit width is returned. The result is always a two's complement number; its bit width is the maximum of the bit widths of $s1$ and $s2$. If an overflow occurs, the bit $o$ is set to 1. The feasible data types for the operands are $T_1, T_2 \in$ { `storage, signed, unsigned` }. The result type $T$ is determined according to table 4.3.2.
- $T$ `_iaddc` ($T_1$ $s_1, T_2$ $s_2$, `unsigned<1>` $c$, `unsigned<1>` $t$, `unsigned<1>` $m$,
  `storage<1>` $o$)

  This function adds the contents of $s_1$, $s_2$ and a carry bit $c$. $s_1$ and $s_2$ are interpreted as two's complement numbers if $t == 1$ and as unsigned values if $t == 0$. If $m == 1$, saturation is performed; then in the case of positive (negative) overflow the largest positive (smallest negative) number of the given bit width is returned. The bit width of the result is the maximum of the bit widths of $s_1$ and $s_2$. If an overflow occurs, the bit $o$ is set to 1. The feasible data types for the operands are $T_1, T_2 \in \{$ `storage, signed, unsigned` $\}$. The result type $T$ is determined according to table 4.3.2.

- $T$ `_isubc` ($T_1$ $s_1, T_2$ $s_2$, `unsigned<1>` $c$, `unsigned<1>` $t$, `unsigned<1>` $m$, `storage<1>` $o$)

  This function subtracts the contents of $s_2$ and a borrow bit $c$ from $s_1$. $s_1$ and $s_2$ are interpreted as two's complement numbers if $t == 1$ and as unsigned values if $t == 0$. If $m == 1$, saturation is performed; then in the case of positive (negative) overflow the largest positive (smallest negative) number of the given bit width is returned. The bit width of the result is the maximum of the bit widths of $s_1$ and $s_2$. If an overflow occurs, the bit $o$ is set to 1. The feasible data types for the operands are $T_1, T_2 \in \{$ `storage, signed, unsigned` $\}$. The result type $T$ is determined according to table 4.3.2.

- `signed _iabs(signed s, unsigned<1> m, unsigned<1> o)`

  calculates the absolute value of the two's complement number $s$. If $m == 1$, saturation is performed; then in the case of positive (negative) overflow the largest positive (smallest negative) number of the given bit width is returned. The result has the same bit width as $s1$. If an overflow occurs, the bit $o$ is set to 1.

- $T$ `_imul` ($T_1$ $s_1, T_2$ $s_2$, `unsigned<1>` $t_1$, `unsigned<1>` $t_2$, `signed` $wo$, `storage<1>` $o$)

  This function multiplies the contents of $s_1$ and $s_2$. $t_1$ and $t_2$ indicate whether $s_1$ and $s_2$ have to be interpreted as unsigned or two's complement numbers; a $t$-value of 1 causes two's complement evaluation. If an overflow occurs, the bit $o$ is set to 1. The feasible data types for the operands are $T_1, T_2 \in \{$ `storage, signed, unsigned` $\}$. The result type $T$ is determined according to table 4.3.2. The value of $wo$ gives the bit width of the result.

- $T$ `_idiv` ($T_1$ $s_1, T_2$ $s_2$, `unsigned<1>` $t_1$, `unsigned<1>` $t_2$, `signed` $wo$, `storage<1>` $o$)

  This function divides the contents of $s_1$ by $s_2$. $t_1$ and $t_2$ indicate whether $s_1$ and $s_2$ have to be interpreted as unsigned or two's complement numbers; a $t$-value of 1 causes two's complement evaluation. If an overflow occurs, the bit $o$ is set to 1. The feasible data types for the operands are $T_1, T_2 \in \{$ `storage, signed, unsigned` $\}$. The result type $T$ is determined according to table 4.3.2. The value of $wo$ gives the bit width of the result.

- $T$ `_imod` ($T_1$ $s_1, T_2$ $s_2$, `unsigned<1>` $t_1$, `unsigned<1>` $t_2$, `signed` $wo$, `storage<1>` $o$)

  This function calculates the remainder of the integer division of $s_1$ by $s_2$. $t_1$ and $t_2$ indicate whether $s_1$ and $s_2$ have to be interpreted as unsigned or two's complement numbers; a $t$-value of 1 causes two's complement evaluation. If an overflow occurs, the bit $o$ is set to 1. The feasible data types for the operands are $T_1, T_2 \in \{$ `storage, signed, unsigned` $\}$. The result type $T$ is determined according to table 4.3.2. The value of $wo$ gives the bit width of the result.

– $T$ `_fadd` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  This function adds $s_1$ to $s_2$. The feasible data types for the operands are $T_1, T_2 \in$ $\{$`storage, float<x,y>` $\}$. Both storage resources are interpreted as floating point values with an $ms$-bit mantissa and an $me$-bit exponent. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The result will also be in the $float(ms, me)$ format. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  * $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
  * $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
  * $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
  * $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
  * $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
  * $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
  * $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

  The result type is determined according to table 4.3.2.

– $T$ `_fsub` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  This function subtracts $s_1$ from $s_2$. The feasible data types for the operands are $T_1, T_2 \in$ $\{$`storage, float<x,y>` $\}$. Both storage resources are interpreted as floating point values with an $ms$-bit mantissa and an $me$-bit exponent. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The result will also be in the $float(ms, me)$ format. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  * $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
  * $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
  * $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
  * $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
  * $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
  * $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
  * $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

  The result type is determined according to table 4.3.2.

– $T$ `_fabs` ($T_1$ $s_1$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

This function calculates the absolute value of storage resource $s_1$. The feasible data types for the operand are $T_1 \in \{$`storage, float<x,y>` $\}$. The storage resources is interpreted as floating point value with an $ms$-bit mantissa and an $me$-bit exponent. If $s_1$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The result will also be in the $float(ms, me)$ format. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

* $ieeeexp[0] : DBZ$ Standard IEEE divide-by-zero flag.
* $ieeeexp[1] : INX$ Standard IEEE inexact flag.
* $ieeeexp[2] : UNF$ Standard IEEE underflow flag.
* $ieeeexp[3] : OVF$ Standard IEEE overflow flag.
* $ieeeexp[4] : INV$ Standard IEEE invalid flag.
* $ieeeexp[5] : IFZ$ "Input flushed to zero".
* $ieeeexp[6] : OFZ$ "Output flushed to zero".

The result type is determined according to table 4.3.2.

– $T$ `_fmul` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

This function multiplies $s_1$ with $s_2$. The feasible data types for the operands are $T_1, T_2 \in \{$`storage, float<x,y>` $\}$. Both storage resources are interpreted as floating point values with an $ms$-bit mantissa and an $me$-bit exponent. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The result will also be in the $float(ms, me)$ format. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

* $ieeeexp[0] : DBZ$ Standard IEEE divide-by-zero flag.
* $ieeeexp[1] : INX$ Standard IEEE inexact flag.
* $ieeeexp[2] : UNF$ Standard IEEE underflow flag.
* $ieeeexp[3] : OVF$ Standard IEEE overflow flag.
* $ieeeexp[4] : INV$ Standard IEEE invalid flag.
* $ieeeexp[5] : IFZ$ "Input flushed to zero".
* $ieeeexp[6] : OFZ$ "Output flushed to zero".

The result type is determined according to table 4.3.2.

– $T$ `_fdiv` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

This function divides $s_1$ by $s_2$. The feasible data types for the operands are $T_1, T_2 \in \{$`storage, float<x,y>` $\}$. Both storage resources are interpreted as floating point values with an $ms$-bit mantissa and an $me$-bit exponent. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page

29 can be used. The result will also be in the $float(ms, me)$ format. The bit sequence *ieeeexp* is used to register IEEE floating point exceptions; they are used as follows:

* $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
* $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
* $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
* $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
* $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
* $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
* $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

The result type is determined according to table 4.3.2.

– $T$ `_frnd` ($T_1$ $s_1$, `signed` $ms_1$, `signed` $me_1$, `signed` $ms_2$, `signed` $me_2$, `storage<2>` $r$, `storage<1>` *ieeeexp*)

This function rounds the floating point value contained in $s_1$ with an $ms_1$-bit mantissa and an $me_1$-bit exponent to a floating point value with an $ms_2$-bit mantissa and an $me_2$-bit exponent. The feasible data type for the operand is $T_1 \in \{$`storage, float<x,y>`$\}$. If $s_1$ is of type `float<x,y>` where $x \neq ms_1$ or $y \neq me_1$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The bit sequence *ieeeexp* is used to register IEEE floating point exceptions; they are used as follows:

* $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
* $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
* $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
* $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
* $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
* $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
* $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

The result type is determined according to table 4.3.2.

– $T$ `_fgt` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` *ieeeexp*)

This function checks if $s_1$ is greater than $s_2$. Both arguments have to be floating point values with an $ms$-bit mantissa and an $me$-bit exponent. The result type is `unsigned<1>`. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The bit sequence *ieeeexp* is used to register IEEE floating point exceptions; they are used as follows:

* $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
* $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
* $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
* $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.

  * $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
  * $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
  * $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

- $T$ `_flt` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  This function checks if $s_1$ is smaller than $s_2$. Both arguments have to be floating point values with an $ms$-bit mantissa and an $me$-bit exponent. The result type is `unsigned<1>`. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  * $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
  * $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
  * $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
  * $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
  * $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
  * $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
  * $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

- $T$ `_feq` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  This function checks if $s_1$ is equal to $s_2$. Both arguments have to be floating point values with an $ms$-bit mantissa and an $me$-bit exponent. The result type is `unsigned<1>`. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  * $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
  * $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
  * $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
  * $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
  * $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
  * $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
  * $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

- $T$ `_fneq` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  This function checks if $s_1$ is not equal to $s_2$. Both arguments have to be floating point values with an $ms$-bit mantissa and an $me$-bit exponent. The result type is `unsigned<1>`. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  - $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
  - $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
  - $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
  - $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
  - $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
  - $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
  - $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

- $T$ `_fgeq` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  This function checks if $s_1$ is greater or equal than $s_2$. Both arguments have to be floating point values with an $ms$-bit mantissa and an $me$-bit exponent. The result type is `unsigned<1>`. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  - $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
  - $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
  - $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
  - $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
  - $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
  - $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
  - $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

- $T$ `_fleq` ($T_1$ $s_1$, $T_2$ $s_2$, `signed` $ms$, `signed` $me$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  This function checks if $s_1$ is smaller or equal than $s_2$. Both arguments have to be floating point values with an $ms$-bit mantissa and an $me$-bit exponent. The result type is `unsigned<1>`. If $s_1$ or $s_2$ is of type `float<x,y>` where $x \neq ms$ or $y \neq me$, an error is reported. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  - $ieeeexp[0]$ : $DBZ$ Standard IEEE divide-by-zero flag.
  - $ieeeexp[1]$ : $INX$ Standard IEEE inexact flag.
  - $ieeeexp[2]$ : $UNF$ Standard IEEE underflow flag.
  - $ieeeexp[3]$ : $OVF$ Standard IEEE overflow flag.
  - $ieeeexp[4]$ : $INV$ Standard IEEE invalid flag.
  - $ieeeexp[5]$ : $IFZ$ "Input flushed to zero".
  - $ieeeexp[6]$ : $OFZ$ "Output flushed to zero".

- `storage _lsh(anytype s, signed n)` performs a logical shift of the value in $s$ by $n$ positions. If n is greater than 0, a right shift is performed, otherwise a left shift.

- `signed _ash(signed s, signed n)` performs an arithmetic shift of the value in $s$ by $n$ positions; the value is interpreted as a two's complement number. If n is greater than 0 a right shift is performed, otherwise a left shift.
- `storage _rot(anytype s, signed n)` rotates the contents of storage $s$ by $n$ positions. If $n$ is greater than 0, a rotate right is performed, otherwise a left rotate.
- `signed cvu2s(unsigned s, signed n)` converts an unsigned $s$ to an $n$-bit two's complement value.
- $T$ `_i2f` ($T_1$ $s_1$, `signed` $ms$, `signed` $me$, `unsiged<1>` $t$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  The function converts $s_1$ in a floating point number with an $ms$-bit mantissa and an $me$-bit exponent. $s_1$ is treated as a two's complement value if $t == 1$ and as an unsigned value if $t == 0$. The parameter $r$ denotes the rounding mode; here the rounding modes of page 29 can be used. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  * $ieeeexp[0] : DBZ$ Standard IEEE divide-by-zero flag.
  * $ieeeexp[1] : INX$ Standard IEEE inexact flag.
  * $ieeeexp[2] : UNF$ Standard IEEE underflow flag.
  * $ieeeexp[3] : OVF$ Standard IEEE overflow flag.
  * $ieeeexp[4] : INV$ Standard IEEE invalid flag.
  * $ieeeexp[5] : IFZ$ "Input flushed to zero".
  * $ieeeexp[6] : OFZ$ "Output flushed to zero".
- $T$ `_f2i` ($T_1$ $s_1$, `signed`$rs$, `signed` $ms$, `signed` $me$, `unsiged<1>` $t$, `storage<2>` $r$, `storage<7>` $ieeeexp$)

  The function converts a floating point number $s_1$ with an $ms$-bit mantissa and an $me$-bit exponent into a two's complement value if $t == 1$ and into an unsigned value if $t == 0$. The parameter $rs$ denotes the bit width of the result type, the parameter $r$ denotes the rounding mode. Here the rounding modes of page 29 can be used. The bit sequence $ieeeexp$ is used to register IEEE floating point exceptions; they are used as follows:

  * $ieeeexp[0] : DBZ$ Standard IEEE divide-by-zero flag.
  * $ieeeexp[1] : INX$ Standard IEEE inexact flag.
  * $ieeeexp[2] : UNF$ Standard IEEE underflow flag.
  * $ieeeexp[3] : OVF$ Standard IEEE overflow flag.
  * $ieeeexp[4] : INV$ Standard IEEE invalid flag.
  * $ieeeexp[5] : IFZ$ "Input flushed to zero".
  * $ieeeexp[6] : OFZ$ "Output flushed to zero".

### 4.3.1 Canonical Functions

User-defined canonical functions have to be introduced by a function prototype; this prototype is used for type checking within the specification of the semantics of the operations. The syntax is defined as follows:

`RTLFunction` *ReturnType Name* `(` ( *ArgType* `,` ) * | *void* `);`

- *ReturnType* resp. *ArgType* are the types of the result, resp. the formal parameters. The feasible types are those introduced above.
- *Name* is a string enclosed in `"` denoting the name of the canonical function.
- Example: `RTLFunction float<24,8> "Sqrts" (float<24,8>);`

The precedence of the operators is exactly the same as defined for the `C` language.

### 4.3.2 Type Inference

When the operands of operations and functions have different types, the type of the resulting value is determined according to Table 4.3.2. The following type variables are used: $t_1 \in \{\texttt{signed, unsigned, float}\}$, $t_2 \in \{\texttt{signed, unsigned}\}$, $t \in \{\texttt{signed, unsigned, float, storage}\}$ , $c$: any numerical constant. Special attention has to be paid when dealing with constants. If a constant is used with a storage location of a given type, the constant is automatically converted to this type which is also propagated to the destination location as result type.

| Operand 1 | Operand 2 | Result |
|---|---|---|
| $\texttt{storage<}m\texttt{>}$ | $t_1\texttt{<}n\texttt{>}$ | $\texttt{storage<}max(m,n)\texttt{>}$ |
| $\texttt{float<}m\texttt{,}n\texttt{>}$ | $t_2\texttt{<}u\texttt{>}$ | $\texttt{float<}m,n\texttt{>}$ |
| $\texttt{signed<}m\texttt{>}$ | $\texttt{unsigned<}n\texttt{>}$ | $\texttt{signed<}max(m,n)\texttt{>}$ |
| $\texttt{float<}m_1\texttt{,}n_1\texttt{>}$ | $\texttt{float<}m_2\texttt{,}n_2\texttt{>}$ | $\texttt{float<}max(m_1,m_2),max(n_1,n_2)\texttt{>}$ |
| $t_1$ | $c$ | $t_1$ |
| $t$ | $t$ | $t$ |

Table 4.1: TypeCombinations.

### 4.3.3 Detailed Grammar of Register Transfer Language

```
IntConst        := [0-9]|[1-9][0-9]+
String          := \"[^\n"]*\"
BinConst        := "0b"[0-1]+
HexConst        := "0x"[0-9A-Fa-f]+
FloatConst      := [0-9]+"."[0-9]+
VarName         := [A-Fa-f][A-Fa-f0-9]*
```

```
SemanticSpec   := StatList | /* empty */;
StatList       := StatList Stat | Stat;
Stat           := Asgn | IfStat | ForStat | WhileStat | SwitchStat |
                VStorageDecl | PreFunc | CanFunc;
Asgn           := AsgnLHS ':=' Expr ';';
IfStat         := 'if' '(' Expr ')' '{' StatList '}' |
                'if' '(' Expr ')' '{' StatList '}' 'else'
                '{' StatList '}';
ForStat        := 'for' '(' Stat Expr ';' Stat ')' '{' StatList'}';
WhileStat      := 'while' '(' Expr ')' '{' StatList '}';
SwitchStat     := 'switch' '(' Expr ')' '{' SwitchCaseList
                  SwitchOptDef '}';
VStorageDecl   := 'external' DType Varname | DType Varname;
PreFunc        := PreFuncName '(' ParaList ')';
CanFunc        := String '(' ParaList ')';
AsgnLHS        := StorageRef | StorageRef '<' IntConst '>' |
                  PreFunc | CanFunc;
Expr           := Expr '||' Expr | Expr '&&' Expr | Expr '|' Expr |
                  Expr '^' Expr | Expr '&' Expr | Expr '==' Expr |
                  Expr '!=' Expr | Expr '<' Expr | Expr '>' Expr |
                  Expr '<=' Expr | Expr '>=' Expr | Expr '<<' Expr |
                  Expr '>>' Expr | Expr '+' Expr | Expr '-' Expr |
                  Expr '*' Expr | Expr '/' Expr | Expr '%' Expr |
                  '-' Expr | '~' Expr | '!' Expr | '(' Expr ')' |
                  PreFunc | CanFunc | IntConst | HexConst | BinConst |
                  FloatConst | StorageRef | StorageRef '<' IntConst '>' |
                  AttribRef | AttribRef '<' IntConst '>' |
                  ExtAttribVal | 'nil';
SwitchCaseList := SwitchCaseList SwitchCase | SwitchCase;
SwitchCase     := 'case' IntConst ':' '{' StatList '}';
SwitchOptDef   := 'default' ':' '{' StatList '}';
StorageRef     := Identifier | Identifier '[' Expr ']';
AttribRef      := 'dst1' | 'dst2' | 'base' | 'offset' | 'counter' |
                  'src1' | 'src2' | 'src3' | 'src4' | 'src5' |
                  'target' | Identifier;
ExtAttribVal   := 'round_up' | 'round_down' | 'round_nearest' |
                  'round_zero' | 'true' | 'false' | Identifier;
ParaList       := ParaList ',' Expr | Expr | /* empty */;
DType          := 'signed' OptSize1 | 'unsigned' OptSize1 |
                  'float' Optsize2 | 'storage' OptSize1 |
                  'fract' Optsize2;
OptSize1       := '<' IntConst '>' | /* empty */;
OptSize2       := '<' IntConst ',' IntConst '>'
```

```
PreFuncName    := '_include' | '_exec' | '_ahs' | '_bext' | '_cvu2s' |
                  '_fabs'| '_fadd'| '_fdiv'| '_float'| '_fmul'| '_fract' |
                  '_frnd'| '_fsub'| '_fgt' | '_flt' | '_feq' | '_fneq' |
                  '_fsub'| '_fgeq' | '_fleq' | '_i2f' | '_f2i' | '_iabs'|
                  '_iaddc'| '_iadd'| '_idiv' | '_imod'| '_imul'| '_isub'|
                  '_isubc'| '_lsb'| '_lsh' | '_msb' | '_rot' | '_sext'|
                  '_signed' | '_unsigned' | '_conc';
```

### 4.3.4 Example

- `dst:=DM[base+offset];`
- `dst:=src1+src2; if (dst<0) {ASTAT<0>:=1;}`
- `int<1> cond; cond:=src1>src2;`
  `if (cond>0) {ASTAT<0>:=0;} else {ASTAT<0>:=1;}`

## 4.4 Defining Operation Classes and Instruction Formats

TDL provides means to declare special operation classes containing operations which share some common properties and should be grouped. This is useful e.g. when declaring constraints which hold for a certain group of operations. The syntax of an operation class declaration reads as follows: `OperationClass` *genname* `{` *gennamelist* `}`. The *gennamelist* must be a list of operations already defined via `DefineOp`.

The execution model underlying the TDL specification is that each instruction of the target processor can be composed of several microoperations. However for analysis purposes, the bit size of an instruction must be known. Moreover in some assembly languages the order of the contained operations is important. Therefore the concept of instruction formats is introduced. An instruction format describes which operations can be grouped together to form a VLIW-like instruction. Each operation that can appear in a certain position in an instruction format is mapped to an appropriate operation class and this class appears in the appropriate order of the instruction format declaration. An instruction format can be declared as follows:

`InstructionFormat` *genname* `width =` *intval*
$\underline{(}$ *OrderedOpClassList* $\underline{|}$ *UnorderedOpClassList* $\underline{)}$ ;

- `width` describes the bit size of the instruction format defined.
- *UnorderedOpClassList* = `{` *OpClassList* `}` is a list of operation classes describing the operations which can be assigned together in an instruction. The order of the operations does not matter.

- *OrderedOpClassList* = `[` *OpClassList* `]` is a list of operation classes describing the operations which can be assigned together in an instruction. The order of the operations is important; first the operations belonging to the first operation class may appear then those of the second class, etc.

- Examples:

  – `InstructionFormat ifo1 {opclass1, opclass2};` says that an instruction of format `ifo1` can be consists of an operation of `opclass1` and one operation of `opclass2` which can be given in any order.

  – `InstructionFormat ifo2 [opclass3, opclass4];` says that an instruction of format `ifo2` consists of an operation of `opclass3` and one operation of `opclass4` where the operation of `opclass3` must be given first.

## 4.5 Definition of Non-Terminals

`OpNT` *genname NTDefinition* `;`

Non-terminals can be used in the instruction set section. Often different operations share some common components; the non-terminals provide a kind of abstraction so that these components have not to be explicitly declared for each instruction. In the terminology of attribute grammars, these non-terminals represent rules which are used to specify operation parts. The syntax of a non-terminal definition is defined as follows.

- *NTDefinition*:
  ( *SynRep OptAction* `|` ) $\ast$ *SynRep OptAction* The definition of a non-terminal consists of a sequence of alternatives; each alternative can be associated to a optional action.

- *SynRep*: Is a combination of literal characters and `%`-directives enclosed in double quotes `"`. These directives can be used to refer to resources, other non-terminals as well as strings and numbers. Some characters have a special meaning. As indicated above, `|` is used to describe syntactical alternatives; by `{` *text* `}*`, a possibly empty sequence of *text* is specified. If `|,{,},*,\` are to be used as pure characters, they must be preceded by a `\`.

- *OptAction*: `{` *OpAttrSpecList* `}`, `{` *OptTimingSpecList* `}`, `{` *SemanticSpec* `}` If no action is specified, the non-terminal describes only the syntactic appearance without relevant consequences on the semantics of the operation or the timing behavior. But all properties of operations which can be defined by `DefineOp` can also be specified in a non-terminal definition (for the meaning of *OpAttrSpecList* `}`, and *TimingSpecList*, see Sec. 4.2). *OptTimingSpecList* denotes a possibly empty list of timing specifications. *SemanticSpec* describes parts of the operation semantics which depend of the operation part associated to the non terminal. They are inserted into the RTL-description of the superior operation; the position in this RTL sequence can be specified by the `_include`-function.

Example:

```
OpNT flag "eq" {guard=true},{},{unsigned<1> cond; cond := ASTAT<0>;};
OpNT optguard "if %!(flag)" | "" {guard = false}, {}, {};
```

# Chapter 5

# The Constraints Section

This section allows for the specification of constraints required to correctly model parallelization restrictions between operations and interdependences between scheduling and allocation decisions. A similar constraint mechanism exists in ISDL [8]; however there the constraints are formulated over the lexical structure of assembly operations. In TDL, the constraints always refer to explicitly declared operation properties or resources. This way, the information of previous sections is reused in the constraint part and elementary semantical checks can help prevent errors in the constraint specification itself.

Each constraint consists of a premise and a rule. The premise has to be a condition which can be statically evaluated; it specifies the condition under which the rule must hold. The TDL-runtime environment generates a function table; for each constraint, an individual function is generated as an element of this table. These functions first check the precondition and, if it is satisfied, write out integer linear programming constraints which are equivalent to the rule-part of the specified constraint. In future versions, subroutines will additionally be generated which can be incorporated in a graph-based scheduling algorithm, e. g. list scheduling.

The constraint section is introduced by the (optional) keyword `Constraints-Section`, which has to be followed by at least one constraint specification. If no constraints are necessary for the target architecture, the keyword `Constraints-Section` is omitted. Each individual constraint declaration must have the following form:

*ConstraintDef*: *ConstrExpr* : *ConstrExpr* ;.

The first occurrence of *ConstrExpr* denotes the premise-part of the constraint, the second one the rule-part; both constraint parts underly the same syntax. Semantical checks assure that the premise part of the constraint can be statically evaluated; if this is not the case, an error is reported.

The constraints are used to prevent illegal combinations of operations during instruction scheduling or to prevent illegal combinations of resource allocation decisions. In the premise part of each constraint, local variables for operations are declared which are bound to the appropriate operation classes. These variables represent symbolic names which have to be used in the rule-part of the constraint to specify the desired restrictions.

In the ADSP-2106x e. g. [1], there are two memory banks which can be used as program resp. data memory. Accesses to program memory and data memory can be executed simulaneously for certain operation combinations, but some combinations are illegal so for some operations, no parallel execution is possible. Program and data memory are declared as parallel resources and for the illegal operation combinations constraints are generated which prohibit parallel execution. This restriction can be formulated in TDL as follows:

```
op1 in {NonParaDM} & op2 in {NonParaPM}: !(op1 && op2);
```

In the following, the detailed grammar of the constraint expressions is given.

- *ConstrExpr*:
  - *ConstrExpr* `|` *ConstrExpr*
    Disjunction of two constraint expressions. This operation has the lowest priority.
  - *ConstrTerm*
- *ConstrTerm*:
  - *ConstrTerm* `&` *ConstrTerm*
    Conjunction of two constraint expressions. The priority of `&` is higher than the priority of `|`.
  - *ConstrFactor*
- *ConstrFactor*:
  - *Identifier* `.` *Fieldref* `in {` *ResRefList* `}` |
    *Fieldref* denotes the name of an attribute which is associated with `Operation`. *Identifier* refers to a symbolic name of an operation variable which is implicitly declared if this expression is used in the premise part of the constraint and there is no previous occurrence of *Identifier* in the current constraint. If it is used in the rule-part, *Identifier* must have been bound to an operation class by the premise of the same constraint. If this is not the case, an error is reported. *ResRefList* is a list of generic names of resources which must have been introduced in the resource section.
  - *Identifier* `in {` *OpOrOpclassList* `}` |
    This expression can only be used in the premise part of a constraint since the mapping of operations to operation classes is always statically known. It means that a precondition of the current constraint is that *Identifier* refers to an operation contained in *OpOrOpclassList*. This list contains generic names of operations or operation classes. Moreover, if it is the first occurrence of *Identifier* in the premise-part of the current constraint, it is introduced as the symbolic name of an operation variable which will be used in the rule-part of the constraint.

- *Identifier1* `->` `(` *intval* `)` *Identifier2* |

  The operation *Identifier2* must be scheduled exactly *intval* cycles after *Identifier2*. This expression can only be used in the rule part of a constraint, since the execution time of the operations is always a dynamic property.

- *Identifier1* `->>` `(` *intval* `)` *Identifier2* |

  The operation *Identifier2* must be scheduled at least *intval* cycles after *Identifier2*. This expression can only be used in the rule part of a constraint, since the execution time of the operations is always a dynamic property.

- *ConstrOpnd* `=` *ConstrOpnd* | *ConstrOpnd* `!=` *ConstrOpnd* | *ConstrOpnd* `<` *ConstrOpnd* | *ConstrOpnd* `<` *ConstrOpnd* | *ConstrOpnd* `>=` *ConstrOpnd* | *ConstrOpnd* `!<=` *ConstrOpnd* |

  These expressions denote the usual tests for equality, inequality as well as arithmetic comparisons. *ConstrOpnd* can be a reference to a declared resource, a numerical constant, and the generic name of any predefined or user-defined attribute resp. attribute value. However the arithmetic comparison operators (`<`, `>`, `<=`, `>=`) can only be used with numerical operands. If this is not the case, an error is reported.

- *Identifier1* `&&` *Identifier2* |

  Operation *Identifier1* must be scheduled at the same cycle as operation *Identifier2*.

- `!` `(` *ConstrExpr* `)` |

  The constraint expression *ConstrExpr* is negated.

- `(` *ConstrExpr* `)`

  This provides for nested constraint expressions.

As a more elaborate example, the complete constraint-section of the ADSP-2106x is given below:

```
Constraints-Section

/* Parallelization Restrictions */
op1 in {ShiftClass} & op2 in {PrgFlowClass, MulClass, MoveOrModifyClass}:
       !(op1 && op2);
op1 in {MiscClass} & op2 in {PrgFlowClass, MulClass, MoveOrModifyClass}:
       !(op1 && op2);
op1 in {NonParaDM} & op2 in {NonParaPM}: !(op1 && op2);
op1 in {PrgFlowClass} & op2 in {DMClass}: !(op1 && op2);
op1 in {NonParaPrgFlow} & op2 in {AluFixedClass, AluFloatClass, MulClass}:
       !(op1 && op2);
```

```
/* Constraints for Multifunctional Operations */
op1 in {MultiAluFixed} & op2 in {MultiMulFixed}:
       !(op1 && op2) | op1.src1 in {iregC} & op1.src2 in {iregD}
       & op2.src1 in {iregA} & op2.src2 in {iregB};
op1 in {MultiAluFloat} & op2 in {MultiMulFloat}:
       !(op1 && op2) | op1.src1 in {iregC} & op1.src2 in {iregD}
       & op2.src1 in {iregA} & op2.src2 in {iregB};
```

The constraints for multifunctional operations have the following meaning: if an operation of the class `MultiAluFixed` is to executed in the same cycle than an operation of the class `MultiMulFixed`, then all operands have to reside in special register groups. Otherwise, the operands are not restricted to certain register groups.

# Chapter 6

# The Assembly Section

In the assembly section, the characteristics of the assembly language which are not directly related to the instruction set of the target processor have to be specified. This includes the notation for assembly comments, of directives, operation and instruction delimiters (e.g. a comma can be used to separate operations of one instruction, whereas semicolons separate different instructions). Moreover, it is possible to introduce delimiters which enclose user annotations of the source file.

- `LineComment` StringList ;

  *StringList* is a comma-separated non-empty list of strings. Each string represents the character sequence denoting the start of a line comment.

- `Comment` *StringPairList* ;

  - *StringPairList*: <u>(</u> `(` *String* `,` *String* `)` `,` <u>)</u> <u>*</u> `(` *String* `,` *String* `)`
    Each *StringPair* gives the character sequence denoting the start of a general comment which can comprise several lines and the corresponding character sequence denoting the end of it.

- `OperationDelimiter` *String* ;

  declares a character sequence which is used to separate several orthogonal operations contained in the same instruction.

- `InstructionDelimiter` *String* ;

  declares a character sequence which separates different instructions.

- `AnnotationDelimiter` *String* ;

  declares a character sequence which encloses each user annotation. User annotations consist of a comma-separated list of attribute/value pairs; the attributes which can be used here have to be especially introduced via the `DefineAttribute` statement.

- Here are some examples:

- `LineComment "!";`
- `Comment ("/*", "*/"), ("(*", "*)");`
- `OperationDelimiter ",";`
- `InstructionDelimiter ";";`
- `AnnotationDelimiter "@";`

## 6.1 Declaration of Assembly Directives

The declaration of assembly directives is similar to the declaration of the instruction set of the target processor. A special type of non-terminals can be used to construct complex directives, similar to the non-terminals used in the instruction set section. Some directives can be expected to be used in most assembly languages, so there are some predefined directive types and some predefined attributes. However, due to the great diversity of assembly languages the focus will be on user-defined directives and user-defined attributes to specify those directives.

`DefineDirective` *genname asmrep* `{` ( *attrib* `=` *value* `,` ) * ( *attrib* `=` *value* `}` `;`

- Predefined attributes are
  - `type` In addition to the following predefined values, the user can declare additional values via `ExtendValueSet`.
    - `SegStart` denotes a directive declaring the beginning of a memory segment.
    - `SegEnd` denotes a directive declaring the end of a memory segment.
    - `GlobalVar` denotes a directive declaring a global variable.
    - `ExternVar` denotes a directive declaring an external variable.
    - `VarInit` denotes a variable initialization.
    - `ArrayInit` denotes a directive specifying an array initialization.
    - `Literal` can be used to indicate that the text of the directive can be written out without modification; no semantical information must be extracted. An example are line directives which lose their meaning anyway after reordering.
  - `name` is an attribute used to specify segment or variable names; it can take arbitrary string values.
  - `memory` is used in segment directives to specify the memory (which must be a resource declared in the resource section) containing the segment.
  - `size` is used in array initializations to indicate the size of the array; its value must be an integer.

- `init` specifies the concrete value which is given in variable initializations. For arrays the complete initialization sequence is passed as a string. Feasible values again are integers.
- `opnd1 - opnd5` Sometimes it is necessary to refer to previously declared resources in the definition of an assembly directive. Therefore those attributes have been introduced whose interpretation is left to the user; feasible values are any numeric or alphanumeric constant as well as references to previously declared resources.
- Moreover additional attributes can be set in the form of key/value pairs.

## 6.2 Declaration of Non-Terminals for Directives

The declaration of the non-terminals used in the specification of directives differs from the declaration of operation non-terminals. The reason is that non-terminals are used to specify attribute setting of operations or directives which depend on some part of the assembly representation of the operation or the directive. Since the attributes associated to operations are different from the attributes associated to directives, an own mechanism is introduced.

`DirNT` *genname DirNTDefinition* `;`

Similar to the operation non-terminals, the non-terminals used in the assembly section provide an abstraction mechanism to facture out common directive parts, so that these components have not to be explicitly declared for each directive. In the terminology of attribute grammars, these non-terminals represent rules which are used to specify directive parts. The syntax of a non-terminal definition is defined as follows.

- *DirNTDefinition*:
  ( *SynRep OptAction* `|` ) * *SynRep OptAction* The definition of a non-terminal consists of a sequence of alternatives; each alternative can be associated to a optional action.
- *SynRep*: Is a combination of literal characters and `%`-directives enclosed in double quotes `"`. These directives can be used to refer to resources, other non-terminals as well as strings and numbers. Some characters have a special meaning. As indicated above, `|` is used to describe syntactical alternatives; by `{` *text* `}*`, a possibly empty sequence of *text* is specified. If `|`,`{`,`}`,`*`,`\` are to be used as pure characters, they must be preceded by a `\`. Additionally, some escape sequences are introduced. This includes first the placeholders `%s`, `%d`, `%x`, `%f`, `%s`, etc. Non-terminals can also be nested; a reference to another non-terminal is introduced by `!%(`, followed by the generic name of the non-terminal and a

closing brace `)`. When dealing with assembly directives, in some cases the contents can be ignored. This is supported by the escape sequence `^()`; then any character except for those given within the brackets will be consumed.

– *OptAction*: `{` *DirAttrSpecList* `}` The attribute list *DirAttrSpecList* can contain any of the attributes introduced in Sec.6.1.

- Examples:

  – `DefineDirective DirSegStart ".segment /%s %s;"`
    `{type=SegStart,  name="$2"};`

  – `DefineDirective DirSegEnd ".endseg" {type=SegEnd};`

  – `DefineDirective DirGcc ".gcc_compiled;" {type=Literal};`

  – `DefineDirective DirGlobVar ".global %s;" {type=GlobalDecl};`

  – `DefineDirective DirExtVar ".extern %s;" {type=ExternDecl};`

# Appendix A

# TDL-Specification of the ADSP-2106x

In this chapter a TDL specifictaion of the Analog Device ADSP-2106x SHARC is given. For reasons of space, some parts have been omitted, but all constructs which are used are represented by at least one example. For this reason, some resources or operations appear in this exerpt without definition, which of course is not feasible in a true application.

```
Resources-Section

FuncUnit ALU 1;

// Integer Registers
Register ireg   "r%d" [0:15] size=40, type=signed<32>, renaming=true;
ResourceClass iregA   {ireg[0:3]};
ResourceClass iregB   {ireg[4:7]};
ResourceClass iregC   {ireg[8:11]};
ResourceClass iregD   {ireg[12:15]};
SetProperties ireg ilpres={iregA, iregB, iregC, iregD};

// Index Registers
Register indreg "i%d" [0:15] size=40, type=signed<32>, usage=Index;
SetProperties indreg[6] usage=FP;
SetProperties indreg[7] usage=SP;

// Modify Registers
Register mreg   "m%d" [0:15] size=40, type=signed<32>, usage=Offset;

// Length Registers
Register lreg   "l%d" [0:15] size=40, type=signed<32>;
```

```
// Base Registers
Register breg   "b%d" [0:15] size=40, type=signed<32>;

// Floating point registers are aliases of the integer registers
RegisterAlias freg "f%d" ireg mapping=[1:1], type=float<24,8>;

// MR registers, used as accumulators
Register MRFreg "MR%dF" [0:2] size=32, type = signed, usage=accu;
SetProperties MRFreg[2] size=16;

Register MRBreg "MR%dB" [0:2] size=32, type=signed, usage=accu;
SetProperties MRBreg[2] size=16;

RegisterAlias MRF "MRF" MRFreg mapping=[3:1], size=80;
RegisterAlias MRB "MRB" MRBreg mapping=[3:1], size=80;

// Now the control flow registers
Register FADDR          "faddr"          size=24, type=signed;
Register DADDR          "daddr"          size=24, type=signed;
Register sPC            "pc"  usage=PC, size=24, type=signed, ;
Register PCSTK          "pcstk"          size=24, type=signed;
Register PCSTKP         "pcstkp"         size=5,  type=signed;
Register LSTKP          "lstkp"          size=5,  type=signed;
Register LADDR          "laddr"          size=32, type=signed;
Register CURLCNTR       "curlcntr"       size=32, type=signed;
Register LCNTR          "lcntr"          size=32, type=signed;
Register MODE1          "mode1"          size=32, type=signed;
Register MODE2          "mode2"          size=32, type=signed;
Register IRPTL          "irptl"          size=32, type=signed;
Register IMASK          "imask"          size=32, type=signed;
Register IMASKP         "imaskp"         size=32, type=signed;
Register ASTAT          "astat"          size=32, type=signed;
Register STKY           "stky"           size=32, type=signed;
Register USTAT1         "ustat1"         size=32, type=signed;
Register USTAT2         "ustat2"         size=32, type=signed;

// Data Memory
Memory DM "dm" type=data,  align=16, access=32;
Memory PM "pm" type=mixed, align=16, access=32;

Cache InstrCache assoc=2, size=192, linesize=48, type=instr;
```

```
// Standard-Resource, used for ILP-Modelling
DefineResource Standard "std" 1;
DefineResource Ctrl "ctrl" 1;

ResourceClass ureg    {ireg,freg,indreg,mreg,breg,lreg};
ResourceClass indregA {indreg[0:7]};
ResourceClass indregC {indreg[8:15]};
ResourceClass mregB   {mreg[0:7]};
ResourceClass mregD   {mreg[8:15]};
ResourceClass sreg    {MODE1, MODE2, IRPTL, IMASK,
                       IMASKP, ASTAT, STKY, USTAT1, USTAT2};

DefineAttribute exec_count {int} associated to Annotation;

InstructionSet-Section

ExtendValueSet mode for Operation by {"ssi", "sui", "usi",
        "uui", "ssf", "suf", "usf", "uuf", "ssfr", "sufr",
        "usfr", "uufr",  "si", "ui", "sf", "uf", "signext",
        "extexp"};

OpNT flag "eq" {guard=true},{;},{unsigned<1> cond;
                cond := ASTAT<0>;}
        | "lt" {guard=true},{;},{
                unsigned<1> cond;
                if ((ASTAT<0>==0b0)
                    &&((ASTAT<2>==0b1)^((ASTAT<1>==0b1)
                    &&(MODE1<13>==0b0))) ||
                    ((ASTAT<10>==0b1)&&(ASTAT<2>==0b1)
                    &&(ASTAT<0>==0b0)))
                   {cond := 0b1; }  else {cond:= 0b0;} };

OpNT optguard "if %!(flag)" | "" {guard = false}, {;}, {};

OpNT jumpmod "(DB)" {},{Ctrl(slots=2);)},{} | "(LA)" | "(CI)"
        | "(DB,LA)" {},{Ctrl(slots=2);)},{}
        | "(DB,CI)" {},{Ctrl(slots=2);)},{} |"";

OpNT dbmod "(DB)" {},{Ctrl(slots=2);},{} |"";

OpNT retmod "(DB)" {},{Ctrl(slots=2);},{} | "(LR)" | "(DB,LR)" {},
                   {Ctrl(slots=2);},{};
```

```
OpNT imm24_t "%24d" {target="$1" as signed<24>},{;},{}
        | "%s" {target="$1" as signed<24>},{;},{};
OpNT imm24_b "%24d" {base="$1" as signed<24>},{;},{}
        | "%s" {base="$1" as signed<24>},{;},{};
OpNT imm24_o "%24d" {offset="$1" as signed<24>},{;},{}
        | "%s" {offset="$1" as signed<24>},{;},{};
OpNT imm6_t "%6d" {target="$1" as signed<6>},{;},{}
        | "%s" {target="$1" as signed<6>},{;},{};
OpNT imm32_b  "%32d" {base="$1" as signed<32>},{;},{}
        | "%s" {base="$1" as signed<32>},{;},{} ;
OpNT imm32_o  "%32d" {offset="$1" as signed<32>},{;},{}
        | "%s" {offset="$1" as signed<32>},{;},{} ;
OpNT imm32_s1 "%32d" {src1="$1" as signed<32>},{;},{}
        | "%s" {src1="$1" as signed<32>},{;},{} ;
OpNT imm32_s2 "%32d" {src2="$1" as signed<32>},{;},{}
        | "%s" {src2="$1" as signed<32>},{;},{} ;
OpNT imm8_s2 "%6d" {src2 = "$1" as signed<8>},{;},{}
        | "%s" {src2 = "$1" as signed<8>},{;},{};
OpNT imm8_s3 "%6d" {src3 = "$1" as signed<8>},{;},{}
        | "%s" {src3 = "$1" as signed<8>},{;},{};

OpNT semod "(SE)" {mode=signext},{;},{} |"";

OpNT exmod "(EX)" {mode=extexp},{;},{} |"";

OpNT mod1 "(SI)" {mode=si},{;},{} |"(UI)" {mode=ui},{;},{} |
          "(SF)" {mode=sf},{;}, {} |"(UF)" {mode=uf},{;},{};

OpNT mod2 "(SSI)" {mode=ssi},{;},{} |"(SUI)" {mode=sui},{;},{}
        |"(USI)" {mode=usi},{;},{} |"(UUI)" {mode=uui},{;},{}
        |"(SSF)" {mode=ssf},{;},{} |"(SUF)" {mode=suf},{;},{}
        |"(USF)" {mode=usf},{;},{} |"(UUF)" {mode=uuf},{;},{}
        |"(SSFR)" {mode=ssfr},{;},{} |"(SUFR)" {mode=sufr},{;},{}
        |"(USFR)" {mode=usfr},{;},{} |"(UUFR)" {mode=uufr},{;},{};

RTLFunction void "CLoop" (signed, storage);
RTLFunction void "Repeat" (unsigned<1>, storage);

DefineOp DMwriteIM "%!(optguard)dm(%s,%s)=%s" {
        dst1=DM, base="$2" in {indregA} as signed<32>,
        offset = "$3" in {mregB} as signed<32>,
        src1 = "$4" in {ureg}, mode =post},
        {DM(latency=1, slots=0, exectime=1);},
```

```
        {external unsigned<1> cond;
         if (cond==1) {base:=base+offset; DM[base]:=src1;}};

DefineOp DMreadIM "%!(optguard)%s=dm(%s,%s)" {
        dst1="$2" in {ureg},
        base="$3" in {indregA} as signed<32>,
        offset = "$4" in {mregB} as signed<32>, src1 = DM,
        mode = post}, {DM(latency=1, slots=0, exectime=1);},
        {external unsigned<1> cond;
         if (cond==1) {base:=base+offset; dst1:=DM[base];}};

DefineOp RegAsgn "%!(optguard)%s = %s" {
        dst1="$2" in {ureg}, src1="$3" in {ureg}},
        {DM(latency=1, slots=0, exectime=1);},
        {external unsigned<1> cond;
         if (cond==1) {dst1:=src1;}};

DefineOp Modify "%!(optguard)modify(%s,%s)" {
        dst1="$2" in {indreg} as signed<32>,
        src1="$2" in {indreg} as signed<32>,
        src2="$3" in {mreg} as signed<32>},
        {DM(latency=1, slots=0, exectime=1);},
        {external unsigned<1> cond;
         if (cond==1) {dst1:=src1+src2;}};

OperationClass MoveOrModifyClass {DMwriteIM, DMreadIM,
        PMwriteIM, PMreadIM, DMwriteMI, DMreadMI, PMwriteMI,
        PMreadMI, DMwriteIcon, PMreadIcon, PMwriteIcon,
        PMreadIcon, DMwriteconI, DMreadconI, PMwriteconI,
        PMreadconI, RegAsgn, Modify};

DefineOp LJumpAbs "%!(optguard)jump%!(imm24_t)%!(jumpmod)"
        {mode="$3", type=Jmp}, {Ctrl(latency=1, exectime=1);},
        {external unsigned<1> cond;
         if (cond == 1) {sPC:=_signed(target,24);}};

DefineOp LJumpRel "%!(optguard)jump(PC, %!(imm24_t))%!(jumpmod)"
        {mode="$3", type=Jmp}, {Ctrl(latency=1, exectime=1);},
        {external unsigned<1> cond;
         if (cond == 1) {sPC:=sPC+_signed(target,24);}};

DefineOp CallMI "%!(optguard)call(%s,%s)%!(dbmod)"
        {base = "$3" in {indreg}, offset = "$2" in {mreg},
```

```
        mode="$4", type=Call}, {Ctrl(latency=1, exectime=1);},
        {_include(optguard); external unsigned<1> cond;
         if (cond == 1) {PCSTKP:=PCSTKP+1; PCSTK:=sPC;
         sPC:=_sext(base+offset,24);}};

DefineOp Rts "%!(optguard)rts%!(retmod)" {mode="$2", noreorder,
         type=Ret}, {Ctrl(latency=1, exectime=1);}, {sPC:=PCSTK;
         PCSTKP:=PCSTKP-1;};

DefineOp CLoopIAbs "lcntr = %16d, do %!(imm24_t) until lce"
        {Counter = "$1" as signed<16>, type=Loop},
        {Ctrl(latency=1, exectime=1, slots=0);},
        {"CLoop"(Counter, target);}

OperationClass PrgFlowClass {LJumpAbs, LJumpRel, LCallAbs,
        LCallRel, JumpMI, NJumpRel, CallMI, NCallRel, CallMIElse,
        NCallRelElse, BranchMI, NBranchRel, Rts, RtsElse, Rti,
        RtiElse, CLoopIAbs, CLoopIRel, CLoopRAbs, CLoopRRel,
        DoUntilAbs, DoUntilRel};

DefineOp ImmDMwrite "dm(%!(imm32_b))=%s"
        {dst1=DM, src1 = "$2" in {ureg}},
        {DM(latency=1, exectime=1, slots=0);},
        {DM[_signed(base,32)]:=src1;};

DefineOp ImmDMread "%s=dm(%!(imm32_b))"
        {dst1="$1" in {ureg}, src1=DM},
        {DM(latency=1, exectime=1, slots=0);},
        {dst1:=DM[_signed(base,32)];};

DefineOp AluFixed1 "%s=%s+%s" {dst1="$1" in {ireg},
        src1="$2" in {ireg}, src2="$3" in {ireg}},
        {Standard(latency=1, slots=0, exectime=1);},
        {dst1:=_iadd(src1, src2, 0b1, MODE1<13>, ASTAT<1>);
         if(dst1==0) {ASTAT<0>:=0b1;} if(dst1<0) {ASTAT<2>:=0b1;}};

DefineOp AluFixed2 "%s=%s-%s" {dst1="$1" in {ireg},
        src1="$2" in {ireg},src2="$3" in {ireg}},
        {Standard(latency=1, slots=0, exectime=1);},
        {dst1:=_isub(src1, src2, 0b1, MODE1<13>, ASTAT<1>);
         if(dst1==0) {ASTAT<0>:=0b1;}
         if(dst1<0) {ASTAT<2>:=0b1;}};
```

```
DefineOp AluFixed14 "%s=%s and %s" {dst1="$1" in {ireg},
        src1="$2" in {ireg}, src2="$3" in {ireg}},
        {Standard(latency=1, slots=0, exectime=1);},
        {dst1:=src1&src2; if(dst1==0) {ASTAT<0>:=0b1;}
         if(_signed(dst1,32)<0) {ASTAT<2>:=0b1;}};

DefineOp AluFixed18 "%s=min(%s,%s)" {dst1="$1" in {ireg},
        src1="$2" in {ireg}, src2="$3" in {ireg}},
        {Standard(latency=1, slots=0, exectime=1);},
        {if (src1<=src2) {dst1:=src1;} else {dst1:=src2;}
         if(dst1==0) {ASTAT<0>:=0b1;}
         if(dst1<0) {ASTAT<2>:=0b1;}};

DefineOp MulFixed1 "%s=%s*%s %!(mod2)" {dst1="$1" in {ireg},
        src1="$2" in {ireg}, src2="$3" in {ireg}},
        {MUL(latency=1, slots=0, exectime=1);},
        {unsigned<1> s1; unsigned<1> s2;
         if (mode==ssi) {s1:=0b1;s2:=0b1;}
         if (mode==sui) {s1:=0b1;s2:=0b0;}
         if (mode==usi) {s1:=0b0;s2:=0b1;}
         if (mode==uui) {s1:=0b0;s2:=0b0;}
         dst1:=_imul(src1, src2, s1, s2, 32, ASTAT<1>);};

DefineOp MulFixed12 "%s=sat %s %!(mod1)" {dst1="$1" in {ireg},
        src1="$2" in {MRF}}, {MUL(latency=1, slots=0, exectime=1);},
        {dst1:="sat"(src1, 40);};

DefineOp Shift1 "%s=lshift %s by %s" {dst1="$1" in {ireg},
        src1="$2" in {ireg}, src2="$3" in {ireg}},
        {Standard(latency=1, slots=0, exectime=1);},
        {dst1:=_lsh(src1, src2); if (dst1==0) {ASTAT<12>:=0b1;}
         if (src2>0) {ASTAT<11>:=0b1;}};

DefineOp Shift5 "%s=ashift %s by %s" {dst1="$1" in {ireg},
        src1="$2" in {ireg}, src2="$3" in {ireg}},
        {Standard(latency=1, slots=0, exectime=1);},
        {dst1:=_ash(src1, src2); if (dst1==0) {ASTAT<12>:=0b1;}
         if (src2>0) {ASTAT<11>:=0b1;}};

OperationClass AluClass {AluFixedClass, AluFloatClass, ShiftClass};
OperationClass MulClass {MulFixedClass, MulFloatClass};
OperationClass ComputeClass {AluClass, MulClass};
```

```
InstructionFormat FormatComputeMoM width=48
        {[AluClass],[MulClass], [MoveOrModifyClass],
         [MoveOrModifyClass]};
InstructionFormat FormatPrgFlow width=48
        {PrgFlowClass,[ComputeClass]};
InstructionFormat FormatImmediate width=48 {ImmediateClass};
InstructionFormat FormatMisc width=48 {MiscClass};

Assembly-Section

OperationDelimiter ",";
InstructionDelimiter ";";
AnnotationDelimiter "@";
Comment ("/*", "/*"), ("{", "}");

DefineDirective DirSegStart ".segment /%s %s;"
        {type=SegStart, name="$2"};

DefineDirective DirSegEnd ".endseg" {type=SegEnd};

DefineDirective DirGcc ".gcc_compiled;" {type=Literal};

DefineDirective DirGlobVar ".global %s;" {type=GlobalDecl};

DefineDirective DirExtVar ".extern %s;" {type=ExternDecl};

OpNT value "%d|%f|%d|%s";
OpNT vallist "{%!(value),}*%!(value)";

DefineDirective DirVarInit ".var %s = %!(value)"
       {type=VarInit, name="$1", init="$2"};

DefineDirective DirArInit ".var %s[%d] = %!(vallist)"
        {type=ArrayInit, name="$1", size="$2", init="$3"};
```

# Bibliography


[1] Analog Devices. *ADSP-2106x SHARC User's Manual*, 1995.

[2] Mario R. Barbacci. Instruction Set Processor Specifications (ISPS): The Notation and Its Applications. *IEEE Transactions on Computers*, C-30(1):24 – 40, January 1981.

[3] Francois Bodin, Zbigniew Chamski, Erven Rohou, and André Seznec. *Functional Specification of SALTO: A Retargetable System for Assembly Language Transformation and Optimization. rev. 1.00 beta.* INRIA, June 1997.

[4] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion System for Retargetable Instruction Scheduling. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 229–240, June 1991.

[5] Trung A. Diep, Christopher Nelson, and John Paul Shen. Performance evaluation of the power pc 620 microarchitecture. *ACM Transactions*, 1995.

[6] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proceedings of the European Design and Test Conference*, pages 503 – 507. IEEE, 1995.

[7] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, September 1997.

[8] George Hadjiyiannis. ISDL: Instruction Set Description Language Version 1.0. Technical report, MIT RLE, April 1998.

[9] A. Halambi, Grun P., V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: A Language for Architecture Exploration through Compiler/Simulator Retargetability. *Proceedings DATE*, March 1999.

[10] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS:Retargetable Code Generation For Embedded DSP Processors. In Peter Marwedel and Gert Goossens, editors, *Code Generation for Embedded Processors*, chapter 5, pages 85–102. Kluwer, Boston; London; Dortrecht, 1995.

[11] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 12th edition, 1993.

[12] L. Nowak. Graph Based Retargetable Microcode Compilation in the MIMOLA Design System. *20th Annual Workshop on Microprogramming*, pages 126 – 132, 1987.

[13] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. *Proceedings of the DAC*, 1999. To be published.

[14] Donald Thomas and Philip Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 2nd edition, 1995.

# Index