Saint–Petersburg State University
Institute of Information Technologies

Processor Architecture Description Language (PADLA)

version 1.0

Dmitri Boulytchev

Saint–Petersburg
2002

Contents

1 Introduction

The paper describes language for macroarchitecture description (Processor Architecture Description LAnguage, PADLA) in informal, rather illustrative manner.

2 Denotations

Extended Backus–Naur notation is used to describe syntax. Nonterminals are denoted by *italic* shape, termanals are **bolded**. The $\mathbf{ascii}(n)$ terminal denotes symbol having ASCII–code n. If t_1 and t_2 are ASCII–symbols then t_1-t_2 denotes a set of ASCII–symbols having codes between t_1 and t_2 inclusively. Via *ASCII* we denote nonterminal that derives all ASCII–symbols separatedly.

Additionally the syntax description formalizm is enriched by the following constructions:

- $R\ \neg T$, where R — syntax expression, T — set of terminals, denotes set of terminals, derived by R except those in T
- if R — syntax expression, t — terminal, then following abbreviation will be used: $R \star t \rightleftharpoons R\ \{\ t\ R\ \}^*$

All grammar rules are numbered successively. Occurence of nonterminal in right–hand part of the rule can be supplied with cross–reference to rule defining it. For example occurence *identifier*(5:3) is supplied with reference to rule 5 on page 3.

An occurence of nonterminal can have underlined part. This part is considered as informal comment — so *<u>type</u>-identifier* means identifier of type.

Semantics and pragmatics are both described informally.

3 Lexical rules

The character set of PADLA is ASCII. There are seven lexem types available: whitespaces, strings, identifiers, keywords, constants, brackets, delimiters and operations:

(1) *token* ::
  whitespace(2:3)
  | *string*(3:3)
  | *identifier*(5:3)
  | *keyword*(10:3)
  | *constant*(11:4)
  | *bracket*(17:4)
  | *operation*(18:4)
  | *delimiter*(20:5)
;;

Whitespace is a sequence of blanks, tabulation characters, carriage returns or line feed symbols:

(2) *whitespace ::*
    { **ascii(32)** | **ascii(9)** | **ascii(10)** | **ascii(13)** }+
  ;;

String is a sequence of characters differ from carriage return delimited by double quotes; backslash character prevents succesive double quote symbol from considering as a string delimiter:

(3) *string ::* **"** *string-character*(4:3)* **"** *;;*

(4) *string-character ::* { *ASCII* ¬ { **"** | **ascii(10)** } } | **\"** *;;*

Identifier is a sequence of letters and digits started with letter; underline symbol is treated as letter. Lowercase and uppercase letters are not distinguished:

(5) *identifier :: letter*(6:3) { *letter* | *digit*(7:3) }* *;;*

(6) *letter ::* **A**-**Z** | **a**-**z** | **_** *;;*

(7) *digit :: octal-digit*(8:3) | **8** | **9** *;;*

(8) *octal-digit :: binary-digit*(9:3) | **2**-**7** *;;*

(9) *binary-digit ::* **0** | **1** *;;*

Keywords are reserved identifiers:

(10) *keyword ::*
  action
  | **asm**
  | **big**
  | **code**
  | **do**
  | **elif**
  | **else**
  | **ext**
  | **fi**
  | **format**
  | **if**
  | **in**
  | **instruction**
  | **ip**
  | **let**
  | **little**
  | **od**
  | **or**
  | **repr**
  | **resource**
  | **sext**
  | **size**
  | **strunc**

| **then**
| **trunc**
| **type**
| **where**
| **while**
| **with**
;;

There are four types of constants: decimal, octal, binary and hexadecimal:

(11) *constant ::*
decimal$_{(12:4)}$
| *binary*$_{(13:4)}$
| *octal*$_{(14:4)}$
| *hexadecimal*$_{(15:4)}$
;;

Decimal constant is a sequence of decimal digits:

(12) *decimal :: digit*$_{(7:3)}$+ *;;*

Binary constant is a sequence of binary digits immediately following the prefix **0b**:

(13) *binary ::* **0b** { *binary-digit*$_{(9:3)}$+ } *;;*

Octal constant is a sequence of octal digits immediately following the prefix **0o**:

(14) *octal ::* **0o** { *octal-digit*$_{(8:3)}$+ } *;;*

Hexadecimal constant is a sequence of hexadecimal digits immediately following the prefix **0x**:

(15) *hexadecimal ::* **0x** { *hexadecimal-digit*$_{(16:4)}$+ } *;;*

(16) *hexadecimal-digit :: digit*$_{(7:3)}$ | **a**-**f** | **A**-**F** *;;*

There are four types of brackets in PADLA — round, square, curved and angle:

(17) *bracket ::* **(** | **)** | **[** | **]** | **{** | **}**<> *;;*

Operation is any sequence of operation characters except '|', '||' и '**..**':

(18) *operation :: operation-character*$_{(19:4)}$+ ¬ { | | || | **..** } *;;*

(19) *operation-character ::*
+
| **-**
| *****

| /
| .
| $
| %
| &
| |
|
| @
| ^
| !
| \
| =
| <
| >
;;

Delimiters are listed below:

(20) *delimiter ::*
|
| ||
| ..
| ->
| <-
| ,
| :
| ;
;;

Lexem is a maximal sequence of characters satisfying rules given above. So

```
+..,..
```

shold be splitted to lexems as follows: operation '+..', delimiter ',', delimiter '..'.

4 Macroarchitecture Description

PADLA macroarchitecture definition consists of sequence of descriptions:

(21) *description :: description-item*$_{(22:5)}$* *;;*

Items to describe are resources, types, formats, assemblers and instructions:

(22) *description-item ::*
resource-declaration$_{(23:6)}$
| *type-declaration*$_{(33:7)}$
| *format-declaration*$_{(47:13)}$
| *assembler-declaration*$_{(51:14)}$
| *instruction-declaration*$_{(54:15)}$
;;

4.1 Resources

Resource is an abstraction for value holder. Informally speaking resources are analogues of variables in classic programming languages. To define resource it's bit width, intrinsic structure (splitting into fields) and number of instances should be supplied:

(23) *resource-declaration* :: **resource** *resource-description*(24:6)+ ;;

(24) *resource-description* ::
resource-identifier(5:3)
resource-cardinality(25:6) **:** *resource-type*(27:6)
resource-details(30:6) **;**
;;

(25) *resource-cardinality* :: [**(** *integer-interval*(26:6) **)**] ;;

(26) *integer-interval* :: *ground-expression* **..** *ground-expression*(70:18) ;;

(27) *resource-type* :: *decimal*(12:4) | **{***named-scale*(28:6)**}** ;;

(28) *named-scale* :: *named-scale-item*(29:6) ⋆ **,** ;;

(29) *named-scale-item* :: *field-identifier*(5:3) [**:** *decimal*(12:4)] ;;

(30) *resource-details* :: [*endianness*(31:6)] [*counter*(32:6)] ;;

(31) *endianness* :: **big** | **little** ;;

(32) *counter* :: **ip** **(***resource-identifier*(5:3)**)** ;;

Different resources must have different names.

The only operations can be used in *ground-expression* are listed below in increasing priority order:

- binary '+' (addition) and '–' (subtraction) where '–' is left–associative;
- binary '*' (multiplication), '/' (quotient) and '%' (reminder) where '/' and '%' are left–associative;
- binary '^' (power);
- unary '+' and '–' (negation);

Only *ground-expression* or *constant* can be supplied as operand of these operations in *ground-expression*. The value of *ground-expression* is calculated at compile–time; the range of possible values is implementation–dependent.

resource-cardinality describes number of resource instances. If it is omitted then resource is treated as having one instance. If *integer-interval* specified then it is treated as a range of addressing values to access instances of this resource so lower bound of range must be less or equal to upper (negative bounds are allowed). If l and u — lower and upper bound of addressing range correspondingly then bit width of address is calculated dependently on the binary numbers representation format.

resource-type defines bit width of value to hold and (possible) its decomposition into a list of fields. In a simplest case decimal constant defining bit width is specified after the ':'. If resource contains fields then their names and bit widths must be listed in curved brackets. If bit width of field is omitted then it is assumed 1. All fields of the same resource must have different names. Bit width of resource decomposed into fields is calculated as the sum of their bit width. The maximal value of bit width is implementation–dependent.

resource-details describe rule to aggregate contiguous instances of the resource and allow to define pointer to current instruction.

Aggregate rule describes order in which instances of resource are grouped to access values of size bigger than one instance. Aggregate rule **big** (*big–endian*) defines the order when higher parts of value correpond to smaller addresses, **little** (*little–endian*) — to bigger ones. All resources assumed big–endian on default.

In macroarchitecture description one resource with one instance holding address of current instruction (instruction pointer) must be declared. The property **ip** is used for this purpose. Name enclosed in brackets points to resource in which program is stored. This resource in turn must have more then one instance, forwarded declaration is not allowed.

For example

```
resource
  mem (0..2^16-1) :  8 little;
  R (0..15)       : 16 little;
  PC              : 16 ip (mem);
  flags           : {C, O, Aux:3, Rest: 3};
```

describes four resources: resource `mem` having 65536 instances of 8 bit width each without decomposition into fields, 16 instances of 16–bit resource `R`, one instance of 16–bit resource `PC` and one instance of 8–bir resource `flags`, decomposed into one–bit fields `C` and `O` and 3–bit fields `Aux` and `Rest`. Resources `mem` and `R` are little–endian, `PC` holds address of current instruction in resource `mem`.

4.2 Types

Type is a way to formally group instruction's parameters so it is analogue of addressing mode in real instruction sets.

The rules for type definition are shown below:

(33) *type-declaration* :: **type** *type-declaration-item*(34:7)+ ;;

(34) *type-declaration-item* :: *simple-type*(35:7) | *compound-type*(36:7) ;;

(35) *simple-type* ::
simple-type-identifier(5:3) **:** *decimal*(12:4) | *integer-interval*(26:6) **;**
;;

(36) *compound-type* ::
compound-type-identifier(5:3)
(*compound-type-parameter*(37:7) ⋆ **,** **)**
;
;;

(37) *compound-type-parameter* :: *simple-type-identifier*(5:3) ;;

simple-type allows to name type of bit scale having width described either explicitly (as decimal constant enclosed in square brackets) or calculated to hold values from given range dependently on the binary representation format. In latter case the range must satisfy exactly the same restrictions as ones for resource declaration.

compound-type allows to group bit scales described either by value range or simple type name under assumption the type with this name has already declared.

For example, declaration

```
type
  Regnum : 4;
  Const  : 0..2^16-1;

  Indir (Regnum);
  IndirOffset (Regnum, Const);
```

describes four types: simple type `Regnum` of width 4, simple type `Const` of width enought to hold values from 0 to 65535, compound type `Indir` with parameter of type `Regnum`, compound type `IndirOffset` with parameters of type `Regnum` and `Const`.

For resources with more than one instance corresponding compound type with the same name is declared implicitly. This compound type has one parameter of the same width as address of the resource. As for example given above the following types are declared:

```
type
  R (0..16);
  mem (0..2^16-1);
```

So the types with the same names cannot be declared explicitly.

4.3 Code Expressions

Code expressions allow to describe miscellaneous actions performed over bit scales and strings in order to produce binary code and assembler text. There are three types of code expressions: bit scales, strings and tuples.

4.3.1 Syntax

The syntax rules for code expressions are as follow:

(38) *code-expression ::*
code-expression-I (39:9)
| *code-expression-I code-expression*
| *code-expression-I* , *code-expression*
;;

Two successive code expressions are considered as concatenation of their values; two code expressions connected with the , are considered as *concatenation with delimiter*. Both expressions must simultaneously be either bit scales or strings. If both operands are bit scales then both these constructions denote concatenation. The meaning of concatenations for string operands is described in section 4.5 on page 14.

(39) *code-expression-I* ::
    code-expression-II (42:9)
  | *code-expression-II* *trimmer* (40:9)
;;

(40) *trimmer* :: **[** *trimmer-type* (41:9) **]** ;;

(41) *trimmer-type* ::
    : *decimal* (12:4)
  | *decimal* **:**
  | *decimal* **:** *decimal*
;;

trimmer denotes slicing of subscale from a bit scale. Left n–bits (**[**n**:]**), right n–bits (**[:**n**]**) (n must be greater than 0 in either case) and bits from n–th to m–th inclusively (**[**n**:**m**]**) could be sliced. In latter case bits are numbered left–to–right starting from 0, n must not exceed m.

(42) *code-expression-II* ::
    code-expression-II **.** *decimal* (12:4)
  | *code-primary* (43:9)
;;

Expression e**.**k denotes selection of k–th member from tuple, denoted by expression e.

(43) *code-primary* ::
    string (3:3)
  | *identifier* (5:3)
  | *code-invocation* (44:9)
  | *literal* (45:9)
  | *constant* (11:4)
  | *code-compound* (46:9)
  | **(** *code-expression* (38:8) **)**
;;

(44) *code-invocation* ::
    identifier (5:3) $<$ *code-expression* (38:8) $\star$ **;** $>$
;;

(45) *literal* :: *constant* (11:4) **:** *decimal* (12:4) ;;

(46) *code-compound* :: **{** *code-expression* (38:8) $\star$ **,** **}** ;;

In code expression *string* denotes string operand, *identifier* — code parameter (see section 4.6, page 15) or code name. Application of function over code expressions corresponds to construction *code-invocation* (see sections 4.4, page 13 and 4.5, page 14). *literal* is a constant with explicitly defined bit length.

This length cannot be smaller than implicitly defined length of the constant. Implicitly defined length of constant c is calculated dependently on the binary numbers representation format. Finally *code-compound* allows to construct tuple from separate code expressions.

Next section contains more rigor description of code expression types and typing rules.

4.3.2 Types and type analysis

The basic types of code expressions are as follows:

- *string* — string;
- $[n]$ — bit scale of length n;
- $[> n]$ — bit scale of length not less than n;
- $\{t_1, t_2, \ldots t_n\}$ — tuple of length k, where t_i are types;
- $\{t_1, t_2, \ldots t_n, \ldots\}$ — tuple of length not smaller than k, where t_i are types;

So code expressions are *polymorphic* on length of bit scale or tuple.

To describe typing rules traditional notation is used. We associate with any code expression type deduced from types of it's subexpressions. To distinguish syntax constructions from types we further enclose them in apostrophs. Denotation $'e' : t$ states for claim "expression e has type t". Typing rule is written as fraction where numerator describes assumptions on subexpression types and denominator describes deduced type of expression. If no assumptions are given then numerator consists of $\emptyset$.

Typing rules for concatenation and concatenation with delimiter are listed below:

$$\frac{'e_1' : [> n], 'e_2' : [> m]}{'e_1 e_2' : [> n + m]} \tag{1}$$

$$\frac{'e_1' : [> n], 'e_2' : [m]}{'e_1 e_2' : [> n + m]} \tag{2}$$

$$\frac{'e_1' : [n], 'e_2' : [> m]}{'e_1 e_2' : [> n + m]} \tag{3}$$

$$\frac{'e_1' : [n], 'e_2' : [m]}{'e_1 e_2' : [n + m]} \tag{4}$$

$$\frac{'e_1' : [> n], 'e_2' : [> m]}{'e_1, e_2' : [> n + m]} \tag{5}$$

$$\frac{'e_1' : [> n], 'e_2' : [m]}{'e_1, e_2' : [> n + m]} \tag{6}$$

$$\frac{'e_1' : [n],' e_2' : [> m]}{'e_1,e_2' : [> n+m]} \tag{7}$$

$$\frac{'e_1' : [n],' e_2' : [m]}{'e_1,e_2' : [n+m]} \tag{8}$$

$$\frac{'e_1' : string,' e_2' : string}{'e_1 e_2' : string} \tag{9}$$

$$\frac{'e_1' : string,' e_2' : string}{'e_1,e_2' : string} \tag{10}$$

The next are typing rules for tuple member selection are subscale slicing:

$$\frac{'e' : \{t_1, t_2, \ldots, t_k\}, i \leq k}{'e.i' : t_i} \tag{11}$$

$$\frac{'e' : \{t_1, t_2, \ldots, t_k, \ldots\}, i \leq k}{'e.i' : t_i} \tag{12}$$

$$\frac{'e' : [n], k \leq n}{'e[: k]' : [k]} \tag{13}$$

$$\frac{'e' : [> n], k \leq n}{'e[: k]' : [k]} \tag{14}$$

$$\frac{'e' : [n], k \leq n}{'e[k :]' : [k]} \tag{15}$$

$$\frac{'e' : [> n], k \leq n}{'e[k :]' : [k]} \tag{16}$$

$$\frac{'e' : [n], k \leq m \leq n}{'e[k : m]' : [m - k + 1]} \tag{17}$$

$$\frac{'e' : [> n], k \leq m \leq n}{'e[k : m]' : [m - k + 1]} \tag{18}$$

Rules for constant ('c'), literal ('$c : w$') and string ('s') are as follow:

$$\frac{\emptyset}{'c' : width(c)} \tag{19}$$

$$\frac{'c : w', w \geq width(c)}{'c : w' : [w]} \tag{20}$$

$$\frac{\emptyset}{'s' : string} \tag{21}$$

Here $width(c)$ denotes bit width calculated dependently on the binary numbers representation format.

Finally the rule for type of tuple is

$$\frac{'e_1' : t_1,' e_2' : t_2, \ldots ,' e_k' : t_k}{'\{e_1, e_2, \ldots , e_k\}' : \{t_1, t_2, \ldots , t_k\}} \tag{22}$$

To describe typing rules for parameters and applications of functions over code expressions we consider additionally the set of *type variables*, denoted by small greek letters (α, β, γ etc.) and $\preceq$ binary *subtyping* relation over set of all types defined as follows:

1. $t \preceq t$, where t is arbitrary type;
2. $t \preceq \alpha$, where α — arbitrary type variable, t — arbitrary type;
3. $n \leq m \Rightarrow [> m] \preceq [> n]$
4. $p_1 \preceq q_1, \ldots , p_k \preceq q_k \Rightarrow \{p_1, p_2, \ldots , p_k\} \preceq \{q_1, q_2, \ldots , q_k\}$
5. $p_1 \preceq q_1, \ldots , p_k \preceq q_k \Rightarrow \{p_1, p_2, \ldots , p_k\} \preceq \{q_1, q_2, \ldots , q_k, ...\}$
6. $p_1 \preceq q_1, \ldots , p_k \preceq q_k \Rightarrow \{p_1, p_2, \ldots , p_k\} \preceq \{q_1, q_2, \ldots , q_k, t_1, t_2, \ldots , t_n\}$, where t_i — arbitrary types;

We denote $(t_1, t_2, \ldots , t_k) \to t$ the type of function taking parameters of types $t_1, t_2, \ldots , t_k$ and returning result of type t. The typing rule for function application is

$$\frac{'f' : (t_1, t_2, \ldots , t_k) \to t,' e_1' : p_1,' e_2' : p_2, \ldots ,' e_k' : p_k, p_i \preceq t_i}{'f < e_1; e_2; \ldots ; e_k >' : t} \tag{23}$$

To define typing rules for parameters of functions over code expressions denote $t_e(p)$ for code expression e and it's subexpression p the type p must have to satisfy typing rules. This type is defined as follows:

1. $t_{'p\ \ e'}(p) = [> 0]$, if $'e' : [> k]'$ or $'e' : [k]$;
2. $t_{'p\ \ e'}(p) = string$, if $'e' : string$;
3. $t_{'p,e'}(p) = [> 0]$, if $'e' : [> k]'$ or $'e' : [k]$;
4. $t_{'p,e'}(p) = string$, if $'e' : string$;
5. $t_{'p[:n]'}(p) = [> n]$;
6. $t_{'p[:n]'}(p) = [> n]$;
7. $t_{'p[n:m]'}(p) = [> (m - n + 1)]$;

8. $t_{'p.k'}(p) = \{\alpha_1, \alpha_2, \ldots, \alpha_k, \ldots\}$;

9. $t_{'f(e_1,e_2,\ldots,e_k=p,\ldots,e_n)','f':(t_1,t_2,\ldots,t_k,\ldots,t_n)\to t}(p) = t_k$;

Given rules are ambigious because they allow to associate different types with operands of concatenations. This ambiguity can be easily resolved since actually code expression cannot contain strings and bit scales simultaneously. The types of function parameters are defined taking into account function definition (see sections 4.4, page 13 and 4.5, page 14 for details).

Finally the rule given below allows to generalize type of expression p over all occuring contexts:

$$\frac{'p' : t, t_e(p) \preceq t}{'p' : t_e(p)} \tag{24}$$

So the type of code expression assumed to be least (in the sense of $\preceq$ relation) common type over all types allowed for this expression in all occuring contexts.

4.4 Formats

Formats serv to define various parameterized actions over bit scales that can further be used to generate binary machine code. Format description introduces it's name, the set of named parameters (if any) and the body:

(47) *format-declaration* :: **format** *format-declaration-item*(48:13)+ ;;

(48) *format-declaration-item* ::
  format-identifier(5:3) [*format-parameters*(49:13)] =
    format-code-expression(38:8) ;
;;

(49) *format-parameters* :: < *format-parameter*(50:13) ⋆ ; >

(50) *format-parameter* ::
  format-parameter-identifier(5:3) [: *decimal*(12:4)]
;;

So format defines a function over code expressions.

Format names as well as parameter names for the same format must be different. Forwarded declarations are allowed assumed no recursion occurs. Format parameter hides format with the same name. Decimal value w of bit width can be specified for format parameter and so the type of the parameter assumed $[w]$. If no bit width specified then the type is deduced in according to rules given above initially assuming it to be described by the new type valiable. The range of bit width values is implementation–dependent.

The format body is a code expression containing no strings. Operations '|' and '||' are not distinguished, function applications are treated as invocations of other formats (no recursive applications allowed).

Examples:

```
format binop <op: 4; a : 2; b : 2> = op a b;
```

Here the type of `binop` is $([4],[2],[2]) \to [8]$.

```
format third <x> = x.3;
```

The type of this format is $(\{\alpha, \beta, \gamma, \ldots\}) \to \gamma$.

```
format g = third <{1:1, 2:2, 3:3}>;
```

The type of `g` is $[3]$.

```
format h = third <{1:1, 2:2, {3:3, 4:3}}>;
```

The type of `h` is $\{[3],[4]\}$.

```
format trim <a> = a [:3];
```

The type of `trim` is $([>3]) \to [3]$.

```
format join <a; b> = a b;
```

The type of `join` is $([>0],[>0]) \to [>0]$.

```
format error <a; b> = (a b) [3:];
```

This expresssion has no type since the type of subexpression `a|b` assumed $[>0]$.

4.5 Assemblers

Assemblers serv to define various parameterized actions over strings that can further be used to generate assembler program textual representation. Assembler description introduces it's name, the set of named parameters (if any) and the body:

(51) *assembler-declaration* :: **asm** *assembler-declaration-item*(52:14)+ ;;

(52) *assembler-declaration-item* ::
assembler-identifier(5:3) [*assembler-parameters*(53:14)] = *assembler-code-expression*(38:8) ;
;;

(53) *assembler-parameters* :: $<$ *assembler-parameter-identifier*(5:3) ⋆ ; $>$
;;

So assembler defines a function over code expressions.

Assembler names as well as parameter names for the same assembler must be different. Forward declarations are allowed assumed no recursion occurs. Assembler parameter hides assembler with the same name.

The assembler body is code expression containing no literals, constants and trimmers. The result of this function application is treated as regular expression over constant strings, parameters and other code expressions. Concatenation here means concatenation of regular expression; as for concatenation with delimiter it denotes the concatenation of regular expressions which allows arbitrary number of whitespaces between its operands. So

```
"a" "b"
```

denotes regular expression `"ab"`, but

```
"a", "b"
```

— regular expression `"a"{" "+}"b"`.

In assembler body function applications are treated as invocations of other assemblers. Type of assembler parameter is deduced in according to the rules given above initially assuming it to be described by the new type variable.

Examples:

```
asm binop <op; a; b> = op a b;
```

The type of this assembler is $(string, string, string) \to string$.

```
asm third <x> = x.3;
```

The type of this assembler is $(\{\alpha, \beta, \gamma, \ldots\}) \to \gamma$.

```
asm g = third <{"a", "b", "c"}>;
```

The type of `g` is *string*.

```
asm h = third <{"a", "b", {"c", "d"}}>;
```

The type of `h` is $\{string, string\}$.

4.6 Instructions

For simplicity assume all macroarchitecture descriptions below contain the same common part:

```
resource
  mem (0..2^16-1):  8;
  PC              : 16 ip (mem);
  R   (0..15)     : 16;

type
  Const  : 0..2^8-1;
  Regnum : 0..15;
  Indir      (Regnum);
  IndirIndex (Regnum, Const);
```

Instruction is named aggregate of parameters, semantic actions over resources and rules for binary code and assembler text generation:

(54) *instruction-declaration* :: **instruction** *instruction-description*(55:15) ;;

(55) *instruction-description* :: *instruction-identifier*(5:3) **:** *match-list*(56:15) ;;

(56) *match-list* :: *match-item*(57:15)+ ;;

(57) *match-item* ::

[*instruction-parameters*(58:16)] **->** *instruction-body*(62:16) *;;*

(58) *instruction-parameters ::*
compound-pattern(60:16) [*constraint*(59:16)] *;;*

(59) *constraint ::* **where** *logic-expression*(70:18) *;;*

(60) *compound-pattern :: operand-pattern*(61:16) ⋆ **,** *;;*

(61) *operand-pattern ::*
type-identifier(5:3)
| *parameter-identifier* **:** *operand-pattern*
| *type-identifier* **(** *parameter-identifier* ⋆ **,)**
;;

(62) *instruction-body ::*
[*action-description*(63:16)]
[*code-description*(64:16)]
[*repr-description*(65:16)]
;;

(63) *action-description ::* **action** **{** *action-specification*(77:23) **}** *;;*

(64) *code-description ::* **repr** **{** *code-specification*(66:16) **}** *;;*

(65) *repr-description ::* **repr** **{** *repr-specification*(67:16) **}** *;;*

(66) *code-specification :: format-code-expression*(38:8) *;;*

(67) *repr-specification :: assembler-code-expression*(38:8) *;;*

Different instructions must have different names. *match-list* allows to enumerate all desirable lists of instruction parameters and to associate corresponding semantic actions and rules for binary code and assembler text generation with each of them. Each match list in turn consists of parameters (*instruction-parameters*) and body (*instruction-body*).

4.6.1 Instruction parameters

Instruction parameters allow to list the patterns (*compound-pattern*) and to specify additional parameters' properties via *constraint* (an expression of a special kind, see p. **??**).

Each pattern in the list represents type of the parameter. Consider the examples:

```
instruction nop:
  ->
```

The list of patters is empty and no constraint specified.

```
instruction mov:
  R(i), R(j) where i <> j ->
```

The list of patterns consists of two elements `R(i)` and `R(j)`. Each of them points to *compound-type* `R` and defines sertain occurence of corresponding resource via specifying parameter its parameter (`i` or `j` corresondingly). The number of parameters in such a case must coincide with number of parmeters of *compound-type*. As for constraint in this example it allows to reflect the fact that given instruction takes different registers as it's operands.

```
instruction mov:
  R(i), x:Const ->
```

Here *simple-type* is pointed as a second parameter so colon is used to separate parameter name from pattern. The same is permitted for pattern pointing to resource:

```
instruction mov:
  r:R(i), x:Const ->
```

So the pattern list allows to define types of instruction operands and to specify the set of it's parameters. These parameters can be divided into two classes: *code* parameters and *semantic* parameters.

Code parameters are those used either as parameters of *compound-type* or as synonimouses of *simple-type* (`i`, `j` and `x` is examples above). Their main property is bit width that assumed to be equal to bit width of corresponding parameter of *compound-type* or *simple-type* (4, 4 and 8 correspondingly in examples above). Code parameters can further be used in instruction sematic description, in rules for generating machine code and assembler text.

Semantic parameters are those that name the occurences of types that are resources (`r` in example above). They further can be used in semantic action description as synonimous for the resource instance they name (so `r` is synonimous for `R(i)` is example above).

The names of both code and semantic parameters must be different, the parameter can simultaneously be both code and semantic.

Further examples:

```
instruction mov:
  dst:R(i), src:mem(addr) ->
```

Parameters `i` and `addr` are code with lengths 4 and 16 bits correspondingly, parameters `dst` and `src` are semantic associated with instances `R(i)` and `mem(addr)` correspondingly.

```
instruction mov:
  dst:R(i), Indir(j) ->
```

Parameter `j` is code with length 4.

```
instruction mov:
  dst:R(i), IndirIndex(j, offs) ->
```

Parameter `offs` is code with length 8.

```
instruction mov:
  dst:R(i), src:IndirIndex(j, offs) ->
```

Invalid definition since `IndirIndex` is not resource.

4.6.2 Expressions and terms

Expressions used in instruction semantic description consist of binary and unary arithmetic operations, function applications, operations of concatenation, slicing and resource field access, constants, literals and terms:

(68) *term* ::
    term-primary(69:18) [**[** *decimal*(12:4) **]**]
  | *term-primary* **.** *field-identifier*(5:3)
;;

(69) *term-primary* ::
    resource-identifier(5:3) [**(** *expression-list*(86:23) **)**]
;;

(70) *expression* ::
    expression-I(71:18)
  | *expression-I* | *expression*
;;

(71) *expression-I* ::
    expression-II(72:18)
  | *expression-II* *binary-operation*(18:4) *expression-I*
;;

(72) *expression-II* ::
    unary-operation(18:4) *expression*(70:18)
  | *function-keyword*(10:3) **(** *expression-list*(86:23) **)**
  | *expression* *trimmer*(40:9)
  | *primary*(73:18)
;;

(73) *primary* ::
    size
  | *constant*(11:4)
  | *literal*(45:9)
  | *substruct*(74:18)
  | *term*(68:18)
  | **(** *expression* **)**
;;

(74) *substruct* :: *term-primary*(69:18) **with** **{** *substruct-fields*(75:18) **}** ;;

(75) *substruct-fields* :: *substruct-field*(76:18) ⋆ **,** ;;

(76) *substruct-field* :: *field-identifier*(5:3) = *expression*(70:18) ;;

Operation of resource field access (*substruct*) can be used to modify some fields of resource. For example given resource definition

```
resource
  Scale : {A:3; B:3; C:3};
```

`Scale with {A=0:3, C=5:3}` is shorthand form of `0:3 | Scale.B | 5:3`.

Term is:

- instruction parameter identifier;
- resource identifier;
- resource instance selecton '`t(e)`', where '`t`' — term corresponding to resource, '`e`' — addressing expression;
- resource field selection '`t.F`', where '`t`' — term corresponding to resource,'`F`' — field name;
- aggregate '`t[n]`', where '`t`' — term, corresponding to resource, '`n`' — number of instances to aggregate;

Expressions can contain reserved identifier **size** denoting a static constant equals to length of binary representation of current instruction measured in instances of resource holding the program.

Aggregation allows to access to several contiguous instances of resource. These instances are grouped in according to aggregation rule defined in that resource definition.

4.6.3 Expression types

All expressions in instruction semantic description must have type $[n]$. An exostive list of all unary and binary operations with their types is given below. All operations are split into groups by increasing priority, operations inside the same group have the same priorities:

1. Logical operations:
   - '`&`' : $([n], [n]) \rightarrow [n]$ — bitwise 'and';
   - '`?`' : $([n], [n]) \rightarrow [n]$ — bitwise 'or';
   - '`@`' : $([n], [n]) \rightarrow [n]$ — bitwise 'xor';
2. Comparison operations:
   - '`<`' : $([n], [n]) \rightarrow [1]$ — less;
   - '`>`' : $([n], [n]) \rightarrow [1]$ — more;
   - '`<=`' : $([n], [n]) \rightarrow [1]$ — less or equal;
   - '`>=`' : $([n], [n]) \rightarrow [1]$ — more or equal;
   - '`==`' : $([n], [n]) \rightarrow [1]$ — equal;
   - '`<>`' : $([n], [n]) \rightarrow [1]$ — not equal;
3. Addition and subtraction:
   - '`+`' : $([n], [n]) \rightarrow [n+1]$ — addition;
   - '`-`' : $([n], [n]) \rightarrow [n+1]$ — subtraction;

   Operation '-' is left–associative.
4. Multiplication, division and shifts:
   - '`/`' : $([n], [n]) \rightarrow [2n]$ — division and reminder. Left n bits of result hols quotent, rigth n — reminder;

- `'*'` : $([n],[n]) \to [2n]$ — multiplication;
- `'<<'` : $([n],*) \to [n]$ — logical shift to left. Vacant position on right is filled by zero.
- `'<*'` : $([n],*) \to [n]$ — arithmetic shift to left. Vacant position on right is filled by zero.
- `'>>'` : $([n],*) \to [n]$ — logic shift to right. Vacant position on left is filled by zero.
- `'*>'` : $([n],*) \to [n]$ — arithmetic shift to right. Vacant position on left is filled by sign bit content.

Only decimal constant (*decimal*) is allowed as right operand of shift operation. The number of positions to shift cannot exceed width of the left operand. Operation `/` is left–associative.

5. Unary operations:

   - `'~'` : $([n]) \to [n]$ — binary complement;
   - `'-'` : $([n]) \to [n]$ — arithmetic complemet $(0-)$;
   - `'!'` : $([1]) \to [1]$ — logic negation;

Beyond these operations four additional constructions can be used in expressions:

- `'ext'` : $([n],*) \to [m]$ — unsigned extension;
- `'sext'` : $([n],*) \to [m]$ — signed extension;

Both these functions must take decimal constant (*decimal*) as their second parameter. The value of the constant cannot be less than bit width of first parameter.

- `'trunc'` : $([n],*) \to [m]$ — unsigned truncate;
- `'strunc'` : $([n],*) \to [m]$ — signed truncate;

Both these functions must take decimal constant (*decimal*) as their second parameter. The value of the constant cannot be greater than bit width of first parameter.

Typing rules for slicing, concatenation, constants and literals are analoguous to those for code expressions (see section 4.3.2, page 10).

Bit width of terms is defined as follows:

1. bit width of resource instance is defined according to resource definition;
2. bit width of resourcse field is defined according to resource field definition;
3. bit width of resource aggregate is defined as multiplication of resource instance bit width to number of instances in aggregate;

Bit width of expression used to address instance of resource must be equal to bit width of resource's address. Number of instances in aggregate cannot exceed number of all instances of the resource aggregated. The bit width of code parameters is defined according to section 4.6.1, page 16.

Bit width of expression **size** assumed to be that of resource declared using attribute **ip**.

Examples:

```
1:2+2:2
```

This expression has type [3].

```
1:8+2:4*4:4
```

This expression has type [9].

```
(1:8+2:4*4:4)[:8]
```

This expression has type [9].

```
mem(512:16) = 0:8
```

This expression has type [1].

```
mem(512:16)[4]
```

This expression has type [32].

If expression `e` has type $[n]$ in context requiring type $[m]$ and $m > n$ then `e` can be cast on default to needed type using construction `sext(e, m)`. For example, expression

```
mem(512)
```

will automatically be converted to

```
mem(sext(512, 16))
```

When default cast is not appropriate implicit conversion should be used.

4.6.4 Constraints

Constraints are expressions of a special kind. The only place they are allowed is instruction parameters definition.

From syntax point of view constraints are essential expression(70:18)s while their type structure, operands and allowed operations are different.

Each subexpression of constraint can have either *Numeric* (universal numeric) or *Boolean* type provided that constraint at whole has type *Boolean*.

Only code parameters (see p. 17) or decimal constants (decimal(12:4)) are allowed operands of constraint. All operands are treated as *Numeric*.

Operations, allowed in constraints, are listed below. They are grouped by increasing priority, operations inside the same group have the same priorities:

1. Logic operations:

- $'\&'$: $(Boolean, Boolean) \to Boolean$ — logic 'and';
- $'?'$: $(Boolean, Boolean) \to Boolean$ — logic 'or';
- $'@'$: $(Boolean, Boolean) \to Boolean$ — logic exclusive 'or';

2. Comparison operations:

   - $'<'$: $(Numeric, Numeric) \to Boolean$ — less;
   - $'>'$: $(Numeric, Numeric) \to Boolean$ — greater;
   - $'<='$: $(Numeric, Numeric) \to Boolean$ — less or equal;
   - $'>='$: $(Numeric, Numeric) \to Boolean$ — greater or equal;
   - $'=='$: $(Numeric, Numeric) \to Boolean$ — equal;
   - $'<>'$: $(Numeric, Numeric) \to Boolean$ — not equal;

3. Addition and subtraction:

   - $'+'$: $(Numeric, Numeric) \to Numeric$ — addition;
   - $'-'$: $(Numeric, Numeric) \to Numeric$ — subtraction;

   Operation '-' is left-associative.

4. Multiplication and division:

   - $'/'$: $(Numeric, Numeric) \to Numeric$ — divisor.
   - $'\%'$: $(Numeric, Numeric) \to Numeric$ — reminder.
   - $'*'$: $(Numeric, Numeric) \to Numeric$ — multiplication;

5. Unary operations:

   - $'-'$: $Numeric \to Numeric$ — arithmetic complement (unary minus);
   - $'!'$: $Boolean \to Boolean$ — logic negation;

4.6.5 Instruction body

Instruction body consists of semantic description (*action-specification*), rule for binary code (*code-specification*) and rule for assembler text representation (*repr-specification*).

Rules for binary code generation and assembler text representation are conventional format and assembler code expression correspondingly. These expressions can have code parameters occurencies. These parameters are treated as bitscales of derived width while occure in format expression and as strings containing string representation of parameters — in assembler expressions. In the latter case number representation is the same as conventional PADLA number representation format.

Example:

```
instruction mov:
  dst:R(i), src:mem(addr) ->
    code {0x01:4 i addr}
    repr {"mov", "R"i, ",", "[", addr, "]"}
```

The type of format code expression in instruction description must be $[n]$, assmbler code expression — *string*. The value of assembler code expression is treated as rule for textual representation of given instruction.

Instruction semantic description defines action performed by instruction over the resources:

(77) *action-specification* :: *sequential*(78:23) *;;*

(78) *sequential* ::
  parallel(79:23)
  | *parallel* **;** *sequential*
;;

(79) *parallel* :: *ground-construction*(80:23) $\star$ **||** *;;*

(80) *ground-construction* ::
  let-binding(81:23)
  | *assignment*(84:23)
  | *conditional*(87:23)
  | *loop*(89:23)
  | *halt*(90:23)
;;

(81) *let-binding* :: **let** *bind-list*(83:23) **in** *binding-area*(82:23) *;;*

(82) *binding-area* :: *sequential*(78:23) *;;*

(83) *bind-list* :: *identifier*(5:3) $\star$ **,** **=** *expression-list*(86:23) *;;*

(84) *assignment* :: *term-list*(85:23) **<-** *expression-list*(86:23) *;;*

(85) *term-list* :: *term*(68:18) $\star$ **,** *;;*

(86) *expression-list* :: *expression*(70:18) $\star$ **,** *;;*

(87) *conditional* :: **if** *logic-expression*(70:18) *conditional-tail*(88:23) *;;*

(88) *conditional-tail* ::
  then *sequential*(78:23) **fi**
  | **then** *sequential* **else** *sequential* **fi**
  | **then** *sequential* **elif** *logic-expression*(70:18) *conditional-tail*
;;

(89) *loop* ::
  while *logic-expression*(70:18) **do** *sequential*(78:23) **od**
;;

(90) *halt* :: **halt** *;;*

In instruction semantic dscription *ground-construction*s are grouped into *sequential* and *parallel* caluses. '**;**' serves for sequential clause delimiting, '||' — for parallel.

Parallel construction $s_1||s_2$ is considered correct if the destinations of assignments in s_1 are all different from destinations of assignments in s_2. For exmple parallel construction

```
mem (R(i)) <- R(j) || R(j) <- mem (R(j))
```

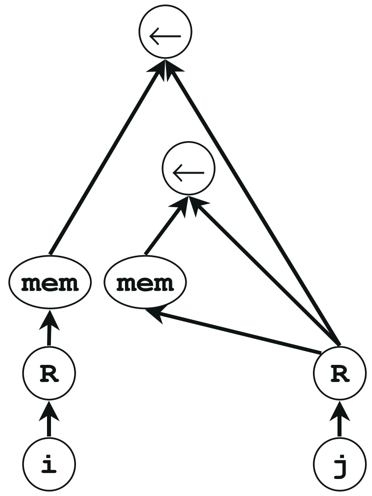

Figure 1: An example of data-flow graph on parallel execution

is correct. In the same time construction

```
R(i) <- R(i)+R(j) || R(i) <- 0
```

is incorrect since both its parts have assignments to resource `R(i)`.

Informally speaking the execution of a parallel construction is split into the tree stages. At first all arguments of all expressions are read from resorces. Then all operations are executed. Finally all calculated values are stored into the resources. So actually parallel construction allows to describe data-flow graph of a calculation. Data-flow graph for correct parallel construction given above is shown at the figure 1.

Sequential construction is conversed to parallel form if possible i.e. if the correctness of result could be proven. For example sequential construction

```
R(i)<-0; mem(addr)<-R(i)
```

is conversed to parallel

```
R(i)<-0 || mem(addr)<-0
```

since destinations of both assignments (`R(i)` and `mem(addr)`) are different.

The correctness of parallel construction could not be proven for each case. Sometimes constraint(59:16) on instruction parameters may help. For example, parallel construction

```
R(i)<-R(j) || R(j)<-R(k)
```

is correct only when `i<>j`. When it is unable to verify correctness of the parallel construction sequential form have to be used instead (in this case it could not be conversed to parallel form).

let-binding allows to introduce a synonimous for expression in such a way that all occurencies of synonimous name inside *binding-area* are replaced with synonimee expression. So

```
let a, b = R(i), mem(x)[2]+mem(y)[2] in
a <- b [:16];
```

is equivalent to

```
R(i) <- (mem(x)[2]+mem(y)[2]) [:16]
```

In assignment term list of destinations is located left to operator '<-' while expression list of sources is located right to it. The number of elements in both lists must be the same, types of corresponding elements must coincide. The source of assignment can be cast to needed type on default either by signed extension or by unsigned truncation. For example assignments

```
PC <- PC+size;
mem(R(i)) <- 1;
```

will automatically be converted to

```
PC <- trunc (PC+size, 16);
mem(R(i)) <- sext (1, 8);
```

conditional(87:23) construction allows to implement branch in dependence of condition expression value. This expression must have type [1]. Example:

```
if R(i) = 0:16 then
   flags.Z <- 1:1
elif R(i) < 0:16 then
   flags.Z <- 0:1;
   flags.N <- 0:1
else
   flags.Z <- 0:1;
   flags.N <- 1:1
fi
```

Similarly loop(89:23) construction allows to implement repeating execution of group of actions provided conditional expression is true. Condition expression must have type [1]. Example:

```
while R(i) <> 0:16 do
  mem (R(j)) <- mem (R(k));

  R(j) <- (R(j) + 1:16) [:16];
  R(k) <- (R(k) + 1:16) [:16];
  R(i) <- (R(i) - 1:16) [:16]
od
```

halt(90:23) construction serves to express processor halt. For example, construction

```
if pc == 0:16 then halt else pc <- trunc (pc+size:16, 16) fi
```

halts processor on attempt to execute instruction from address 0.

Index