Санкт–Петербургский Государственный Университет
Институт Информационных Технологий

Processor Architecture Description Language (PADLA)

версия 1.0

Д.Ю.Булычев

Санкт–Петербург
2002

Содержание

1 Введение

Данный текст описывает язык описания макроархитектуры (Processor Architecture Description Language, PADLA). Это описание не является вполне формальным и выполняет скорее иллюстративную функцию.

2 Нотация

Для описания синтаксиса используется расширенная нотация Бэкуса–Наура. Нетерминалы обозначаются *наклонным* шрифтом, терминалы — **выделенным**. Терминал $\mathbf{ascii}(n)$ соответствует символу с ASCII–кодом n. Если t_1 и t_2 — ASCII–символы, то через t_1-t_2 обозначено множество ASCII–символов, попадающих (по своим кодам) в интервал между t_1 и t_2 включительно. Через *ASCII* обозначен нетерминал, выводящий все символы ASCII по отдельности.

Кроме того, форма описания синтаксиса расширена следующими дополнениями:

- выражение вида $R \neg T$, где R — синтаксическое выражение, T — множество терминалов, обозначает множество терминалов, порождаемых R, за исключением терминалов из T
- если R — выражение, t — терминал, то будет использоваться следующее сокращение: $R \star t \rightleftharpoons R \; \{ \; t \; R \; \}^*$

На всех правилах введена сквозная нумерация. Использующее вхождение нетерминала в правило может быть снабжено перекрестной ссылкой на правило, которое его определяет. Например, вхождение *identifier*(5:3) снабжено ссылкой на правило 5 на странице 3.

В некоторых случаях в правила будут входить нетерминалы, содержащие <u>подчеркнутую</u> часть. В таком случае эта подчеркнутая часть может быть рассмотрена как комментарий к нетерминалу. Например, *<u>type</u>-identifier* означает вхождение идентификатора типа.

Семантика и прагматика описаны на естественном языке.

3 Лексика

Набором символов PADLA является ASCII. Текст на PADLA состоит из лексем следующих типов: пустот, строк, идентификаторов, ключевых слов, констант, скобок, разделителей и знаков операций:

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

;;

Пустоты — это последовательности пробелов, символов табуляции, возврата каретки или перевода строки:

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

Строки — это последовательности символов отличных от перевода строки, ограниченные двойными кавычками, при этом символ обратной косой черты экранирует следующий за ним символ двойной кавычки:

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

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

Идентификаторы — это последовательности букв или цифр, начинающихся с буквы, при этом символ подчеркивания считается буквой. Прописные и строчные буквы различаются.

(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** *;;*

Ключевые слова — это предопределенные идентификаторы, которые могут быть использованы только строго определенным образом:

(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**
;;

Константы могут быть десятичными, восьмиричными, двоичными или шестнадцатиричными:

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

Десятичная константа — это последовательность десятичных цифр:

(12) *decimal* :: *digit*(7:3)+ ;;

Двоичная константа — это последовательность двоичных цифр, непосредственно перед которой указан префикс **0b**:

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

Восьмиричная константа — это последовательность восьмиричных цифр, непосредственно перед которой указан префикс**0o**:

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

Шестнадцатиричная константа — это последовательность шестнадцатиричных цифр, непосредственно перед которой указан префикс **0x**:

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

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

В PADLA используются скобки четырех видов — круглые, квадратные, фигурные и угловые:

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

Операция — это непрерывная последовательность символов операций, кроме '|', '||' и '**..**':

(18) *operation* :: *operation-character*(19:5)+ ¬ { | | || | .. } ;;

(19) *operation-character* ::

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

Разделители бывают следующие:

(20) *delimiter* ::

|
| ||
| ..
| ->
| <-
| ,
| :
| ;
;;

Лексемой считается максимальная последовательность символов, удовлетворяющая описанным выше правилам. Например, в строке

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

лексемы определяются следующим образом: знак операции '+..', разделитель ',', разделитель '..'.

4 Описание макроархитектуры

Текст на PADLA содержит последовательность описаний:

(21) *description* :: *description-item*(22:6)* ;;

Описаны могут быть ресурсы, типы, форматы, ассемблеры и инструкции:

(22) *description-item ::*
resource-declaration(23:6)
| *type-declaration*(33:8)
| *format-declaration*(47:14)
| *assembler-declaration*(51:15)
| *instruction-declaration*(54:16)
;;

4.1 Ресурсы

Ресурс — это место для хранения информации. В некотором смысле можно считать, что ресурсы являются аналогами переменных в обычных языках программирования. Для задания ресурсов необходимо определить размер хранимого значения (в битах), его структуру (разбиение на поля) и количество экземпляров данного ресурса:

(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*(70:19) *;;*

(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)**)** *;;*

Имена различных ресурсов должны различаться.

В состав основного выражения (*ground-expression*) могут входить только следующие операции (перечисленные в порядке возрастания приоритета):

- бинарные '+' (сложение) и '–' (вычитание), при этом операция '–' считается левоассоциативной;
- бинарные '*****' (умножение), '/' (частное) и '**%**' (остаток), при этом операции '/' и '**%**' считаются левоассоциативными;
- бинарная '^' (возведение в степень);
- унарные '+' и '–';

Операндом этих операций в *ground-expression* может быть либо *ground-expression*, либо константа (*constant*). Значение *ground-expression* вычисляется при компиляции описания. Диапазон поддерживаемых значений определяется реализацией.

Мощность ресурса (*resource-cardinality*) описывает количество его экземпляров. Если *resource-cardinality* опущена, то количество экземпляров принимается равным 1. Если же указан интервал (*integer-interval*), то он трактуется как диапазон адресов, при этом нижняя граница обязана не превосходить верхнюю (допускаются отрицательные границы). Если l и u — соответственно нижняя и верхняя границы диапазона адресов, то ширина адреса в битах вычисляется в зависимости от выбранного представления чисел.

Тип ресурса (*resource-type*) задает размер хранимого значения и (возможно) его разбиение на поля. В простейшем случае после символа '**:**' следует десятичная константа, задающая битовый размер ресурса. Если же ресурс содержит поля, то их имена должны быть перечислены в фигурных скобках с возможным указанием длины. Если длина поля не указана, то она принимается равной 1. Все поля данного ресурса должны иметь различные имена. При наличии полей битовая длина ресурса вычисляется как сумма длин полей, самое левое поле соответствует старшим битам. Максимальный размер битовой длины определяется реализацией.

Дополнительные свойства ресурса *resource-details* определяют правило агрегирования экземпляров ресурса (*endianness*) и позволяют задать указатель на текущую инструкцию.

Агрегирование определяет правило группировки экземпляров ресурса при доступе к значениям большей длины. При указании **big** (*big–endian*) старшим частям значения соответствуют младшие адреса, при указании **little** (*little–endian*) — младшие. По умолчанию ресурсы считаются big–endian.

В описании макроархитектуры должен быть указан ровно один ресурс мощности 1, содержащий адрес текущей исполняемой инструкции. Для этого используется свойство **ip**, у которого в скобках указан тот ресурс, в котором хранится программа. Мощность этого ресурса должна быть больше 1, опережающее его использование не допускается.

Например, описание

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

определяет четыре ресурса: ресурс `mem` мощностью 65536 экземпляров ширины 8 бит без разбиения на поля, 16 экземпляров 16–ти разрядного ресурса `R`, один экземпляр 16–ти битового ресурса `PC` и один экземпляр 8-ми битового ресурса `flags`, разбитого на поля `C` и `O` по одному биту и поля `Aux` и `Rest` по три бита. Ресурсы `mem` и `R` являются little–endian, `PC` содержит адрес текущей инструкции в ресурсе `mem`.

4.2 Типы

Тип служит для формальной группировки параметров инструкций. Его аналогом является режим адресации в системах команд процессоров.

Описание типа выглядит следующим образом:

(33) *type-declaration* :: **type** *type-declaration-item*(34:8)+ ;;

(34) *type-declaration-item* :: *simple-type*(35:8) | *compound-type*(36:8) ;;

(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*(37:8) ⋆ **,** **)**
;
;;

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

Простой тип (*simple-type*) позволяет снабдить именем тип битовой шкалы, длина которой задается либо явным образом (десятичная константа в квадратных скобках), либо вычисляется для интервала в зависимости от формата представления чисел. Интервал в описании простого типа должен удовлетворять в точности тем же условиям, что и интервал в описании ресурса.

Составной тип (*compound-type*) позволяет сгруппировать вместе битовые шкалы, заданные либо с помощью интервала, либо с использованием имени простого типа, при этом этот простой тип уже должен быть описан.

Например, описание

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

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

описывает четыре типа: простой тип `Regnum` шириной 4 бита, простой тип `Const` ширины, достаточной для хранения чисел в диапазоне от 0 до 65535, составной тип `Indir` с параметром типа `Regnum`, составной тип `IndirOffset` с параметрами типов `Regnum` и `Const`.

Для ресурсов, имеющих мощность более единицы, автоматически порождается составной тип с тем же именем и одним параметром, имеющим такую же ширину, что и адрес этого ресурса. Иными словами, для описания ресурсов, приведенных в предыдущем примере, будут автоматически порождены следующие типы:

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

Соответственно, типы с такими именами уже не могут быть описаны.

4.3 Кодовые выражения

Кодовые выражения позволяют описать различные действия над битовыми шкалами и строками, которые необходимо произвести для порождения объектного кода и ассемблерного текста. Кодовые выражения бывают трех типов: битовые шкалы, строки и n–ки, при этом результатом выражения над строками является некоторое регулярное выражениею

4.3.1 Синтаксис

Синтаксис кодовых выражений таков:

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

Два идущих подряд кодовых выражения трактуются как их конкатенация, а связанных операцией ',' — как *конкатенация с возможным разделителем.* Конкатенируемые выражения должны быть одновременно либо битовыми шкалами, либо строками. Если операнды — битовые шкалы, то обе эти конструкции обозначают конкатенацию. Значение этих операций над строками описано в разделе 4.5 на странице 15.

(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*
;;

Триммер (*trimmer*) обозначает выборку части битовой шкалы. Возможна выборка левых n бит (**[**n**:]**), правых n бит (**[:**n**]**) (в обоих случаях n должно быть отлично от 0) и бит с n-го по m-го включительно (**[**n**:**m**]**). В последнем случае биты нумеруются справа–налево, начиная с 0, n должно не превосходить m.

(42) *code-expression-II ::*
code-expression-II **.** *decimal*(12:4)
| *code-primary*(43:10)
;;

Выражение e**.**k означает выборку k–го компонента из n–ки, заданной выражением e.

(43) *code-primary ::*
$\quad$ *string*(3:3)
$\quad$ | *identifier*(5:3)
$\quad$ | *code-invocation*(44:10)
$\quad$ | *literal*(45:10)
$\quad$ | *constant*(11:4)
$\quad$ | *code-compound*(46:10)
$\quad$ | **(** *code-expression*(38:9) **)**
;;

(44) *code-invocation ::*
$\quad$ *identifier*(5:3) $<$ *code-expression*(38:9) $\star$ **;** $>$
;;

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

(46) *code-compound ::* **{** *code-expression*(38:9) $\star$ **,** **}** *;;*

Строка (*string*) в кодовом выражении обозначает строковый операнд. Идентификатор (*identifier*) — кодовый параметр (см. раздел 4.6, стр. 16) или имя. Применению функции над кодовыми выражениями соответствует конструкция *code-invocation* (подробнее о таких функциях см. разделы 4.4, стр. 14 и 4.5, стр. 15). Литерал (*literal*) — это константа, снабженная явным указанием ее битовой длины. Длина, необходимая для хранения константы c, определяется исходя из формата представления чисел. Указанная в литерале длина не должна быть меньше длины, вычисленной этим способом. Наконец, составное выражение (*code-compound*) позволяет создать n–ку из значений кодовых выражений.

В следующем разделе будут более подробно описаны правила типового контроля для кодовых выражений.

4.3.2 Типы и их анализ

Базовыми типами кодовых выражений являются следующие:

- *string* — строкового;
- $[n]$ — битовая строка длины n;
- $[> n]$ — битовая строка длины не меньше, чем n;
- $\{t_1, t_2, \dots t_n\}$ — тип n–ки длины k, где t_i — типы;
- $\{t_1, t_2, \dots t_n, \dots\}$ — тип n–ки длины не меньше, чем k, где t_i — типы;

Таким образом, кодовые выражения могут быть *полиморфны* относительно длины битовой шкалы или n–ки.

Далее для описания правил соответствия типов будет использована традиционная нотация, в которой каждому кодовому выражению приписывается тип в зависимости от типов его подвыражений. Для определенности синтаксические конструкции далее будут окружены апострофами, а утверждение «выражение e имеет тип t» будет обозначаться как $'e' : t$. Правила типовой корректности записываются в виде дроби, числитель которой содержит предположения относительно типов подвыражений, а знаменатель

— вывод типа для выражения, составленного из них. Если никаких предположений не требуется, то в числителе будет стоять $\emptyset$.

Правила для конкатенации и конкатенации с разделителем выглядят так:

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

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

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

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

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

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

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

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

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

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

Далее сформулируем правила типовой корректности для выборки компоненты n–ки и триммера:

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

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

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

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

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

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

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

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

Правила для констант ('c'), литералов ('$c : w$') и строк ('s') выглядят так:

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

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

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

Здесь $width(c)$ обозначает битовую длину константы, вычисленную в зависимости от двоичного формата представления чисел.

Наконец, правило для составного кодового выражения выглядит очевидным образом:

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

Для того, чтобы описать правила типовой корректности для параметров и применения функции над кодовыми выражениями, введем дополнительно набор *типовых переменных*, которые будем далее обозначать строчными буквами греческого алфавита (α, β, γ и т.д.) и бинарное отношение «*быть подтипом*» на множестве всех типов (обозначим $\preceq$). Отношение $\preceq$ определяется следующим образом:

1. $t \preceq t$, где t — произвольный тип;

2. $t \preceq \alpha$, где α — произвольная типовая переменная, t — произвольный тип;

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. $p_1 \preceq q_1, \ldots, p_k \preceq q_k \Rightarrow \{p_1, p_2, \ldots, p_k\} \preceq \{q_1, q_2, \ldots, q_k, t_1, t_2, \ldots, t_n\}$, где t_i — произвольные типы;

Тип функции над кодовыми выражениями обозначим через $(t_1, t_2, \ldots, t_k) \to t$, где $t_1, t_2, \ldots, t_k$ — типы параметров, t — тип результата. Тогда правило типовой корректности для применения функции выглядит следующим образом:

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

Теперь опишем правила контроля типов для параметра функции над кодовыми выражениями. Для этого введем следующее обозначение: для выражения e и входящего в него подвыражения p обозначим через $t_e(p)$ тип, который должно иметь p, чтобы выполнялись правила типовой корректности. Этот тип определяется следующим образом:

1. $t_{'p\ \ e'}(p) = [> 0]$, если $'e' : [> k]'$ или $'e' : [k]$;
2. $t_{'p\ \ e'}(p) = string$, если $'e' : string$;
3. $t_{'p,e'}(p) = [> 0]$, если $'e' : [> k]'$ или $'e' : [k]$;
4. $t_{'p,e'}(p) = string$, если $'e' : string$;
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. $t_{'f(e_1, e_2, \ldots, e_k = p, \ldots, e_n)', 'f' : (t_1, t_2, \ldots, t_k, \ldots, t_n) \to t}(p) = t_k$;

Данные правила являются неоднозначными — они позволяют приписывать операнду конкатенаций разные типы. Эта неоднозначность разрешается тем, что, как мы увидим ниже, в одно и то же кодовое выражение могут входить либо строки, либо константы и литералы.

Типы параметров функции над кодовыми выражениями определяются исходя из разновидности этой функции и ее описания. Подробнее об этом см. разделы 4.4, стр. 14 и 4.5, стр. 4.5.

Наконец, следующее правило обеспечивает обобщение типа выражения p в разных контекстах:

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

Таким образом, тип кодового выражения полагается равным наименьшему в смысле отношения $\preceq$ типу среди всех типов, которые требуются от этого выражения во всех контекстах.

4.4 Форматы

Форматы — это способ описать различные параметризованные действия над битовыми шкалами, которые затем можно использовать для порождения двоичного кода. Описание формата вводит в рассмотрение его имя, набор и типы параметров (если они есть) и его тело:

(47) *format-declaration* :: **format** *format-declaration-item*$_{(48:14)}$+ ;;

(48) *format-declaration-item* ::
format-identifier$_{(5:3)}$ [*format-parameters*$_{(49:14)}$] =
format-code-expression$_{(38:9)}$ **;**
;;

(49) *format-parameters* :: $<$ *format-parameter*$_{(50:14)}$ $\star$ **;** $>$

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

Таким образом, формат определяет функцию над кодовыми выражениями.

Имена форматов, а также имена параметров для данного формата должны быть различны. Допускается опережающее использование идентификаторов форматов при условии отсутствия рекурсии. Параметр формата перекрывает формат с тем же именем. Для параметров формата могут быть заданы десятичные значения их битовой ширины. Если битовая ширина параметра задана равной w, то его типом считается $[w]$, иначе тип выводится в соответствии с правилами, описанными выше, в первоначальном предположении, что он описывается новой типовой переменной. Диапазон значений битовой ширины зависит от реализации.

Телом формата является кодовое выражение, которое не должно содержать строк, операции конкатенации и конкатенации с разделителем не различаются. Конструкция применения в теле формата интерпретируется как применение другого формата (рекурсивные применения не допускаются).

Примеры:

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

Здесь тип `binop` — $([4],[2],[2]) \to [8]$.

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

Тип этого формата — $(\{\alpha,\beta,\gamma,\ldots\}) \to \gamma$.

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

Тип `g` — $[3]$.

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

Тип `h` — $\{[3],[4]\}$.

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

Тип `trim` — $([> 3]) \to [3]$.

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

Тип `join` — $([> 0], [> 0]) \to [> 0]$.

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

Это выражение некорректно, поскольку тип подвыражения `a|b` полагается $[> 0]$.

4.5 Ассемблеры

Ассемблеры — это способ описать различные параметризованные действия над строками, которые затем можно использовать для порождения текстового представления на языке целевого ассемблера. Описание ассемблера вводит в рассмотрение его имя, набор параметров (если они есть) и его тело:

(51) *assembler-declaration* :: **asm** *assembler-declaration-item*(52:15)+ ;;

(52) *assembler-declaration-item* ::
assembler-identifier(5:3) [*assembler-parameters*(53:15)] =
assembler-code-expression(38:9) ;
;;

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

Таким образом, ассемблер определяет функцию над кодовыми выражениями.

Имена ассемблеров, а также имена параметров для данного ассемблера должны быть различны. Допускается опережающее использование идентификаторов ассемблеров при условии отсутствия рекурсии. Параметр ассемблера перекрывает ассемблер с тем же именем.

Телом ассемблера является кодовое выражение, которое не должно содержать литералов, констант и триммеров. Результат применения этой функции интерпретируется как регулярное выражение над константными строками и параметрами, при этом обычная конкатенация обозначает конкатенацию регулярных выражений, а конкатенация с разделителем — такую конкатенацию, в которой между операндами допускается произвольное число пробелов. Таким образом,

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

обозначает регулярное выражение `"ab"`, а

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

— регулярное выражение `"a"{" "+}"b"`.

Конструкция применения в теле ассемблера интерпретируется как применение другого ассемблера (рекурсивные применения не допускаются). Всем параметрам ассемблера приписывается тип *string*.

Примеры:

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

Тип этого ассемблера — $(\mathit{string}, \mathit{string}, \mathit{string}) \to \mathit{string}$.

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

Тип этого формата — $(\{\alpha, \beta, \mathit{string}, \ldots\}) \to \mathit{string}$.

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

Тип `g` — *string*.

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

Тип `h` — $\{\mathit{string}, \mathit{string}\}$.

4.6 Инструкции

Далее будем считать, что описание макроархитектуры содержит следующую общую часть:

```
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);
```

Инструкция — это именованная совокупность параметров, семантических действий над ресурсами и правил порождения текстового представления и двоичного кода:

(54) *instruction-declaration ::* **instruction** *instruction-description*(55:16) *;;*

(55) *instruction-description ::* *instruction-identifier*(5:3) **:** *match-list*(56:16) *;;*

(56) *match-list ::* *match-item*(57:16)+ *;;*

(57) *match-item ::*
[*instruction-parameters*(58:16)] **->** *instruction-body*(62:17) *;;*

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

(59) *constraint ::* **where** *constraint-expression*(70:19) *;;*

(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*(63:17)]
  [*code-description*(64:17)]
  [*repr-description*(65:17)]
;;

(63) *action-description* :: **action** { *action-specification*(77:24) } ;;

(64) *code-description* :: **repr** { *code-specification*(66:17) } ;;

(65) *repr-description* :: **repr** { *repr-specification*(67:17) } ;;

(66) *code-specification* :: *format-code-expression*(38:9) ;;

(67) *repr-specification* :: *assembler-code-expression*(38:9) ;;

Имена инструкций должны различаться. Список сопоставления (*match-list*) позволяет перечислить совокупности возможных операндов для данной инструкции и соответствующие этому списку семантические действия и правила порождения текстового представления и двоичного кода. Каждый список сопоставления в свою очередь состоит из параметров (*instruction-parameters*) и тела (*instruction-body*).

4.6.1 Параметры инструкции

Описание параметров инструкции задает список образцов (*compound-pattern*) и ограничение (*constraint*), которое позволяет задать некоторые дополнительные свойства параметров инструкции. В качестве ограничения может быть использовано выражение специального вида (см. стр. **??**).

Каждый образец списка образцов представляет из себя указание типа параметра. Рассмотрим примеры.

В описании

```
instruction nop:
  ->
```

список образцов пуст и ограничение отсутствует.

В описании

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

список образцов состоит из двух элементов — образцов `R(i)` и `R(j)`. Каждый из них представляет собой указание типа операнда инструкции, снабженного параметрами (соответственно `i` и `j`), при этом количество параметров должно соответствовать числу параметров составного типа (*compound-type*). Ограничение в данном примере позволяет отразить тот факт, что данная инструкция допускает в качестве параметров только регистры с разными номерами.

В описании

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

в качестве второго элемента списка образцов использован простой тип (*simple-type*), поэтому имя параметра отделено от имени типа двоеточием. Так же можно поступить, если образец указывает на тип, который одновременно является ресурсом:

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

Если элемент списка образцов указывает на тип, который не является ресурсом, то так написать нельзя.

Таким образом, список образцов позволяет описать типы операндов инструкции и задать множество ее параметров. Эти параметры делятся на два класса — *кодовые* параметры и *семантические* параметры.

Кодовые параметры — это параметры, которые введены использованием в качестве индексов составного типа или именованием вхождения простого типа (в примерах выше — `i`, `j` и `x`). Их атрибутом является битовая ширина, которая принимается равной битовой ширине соответствующего параметра составного типа (в примерах выше — 4, 4 и 8 соответственно). Кодовые параметры могут использоваться как в описании семантики инструкции, так и в описании правил порождения двоичного кода и ассемблерного представления.

Семантические параметры — это параметры, введенные именованием вхождения типов, являющихся ресурсами (в примере выше — `r`). Они могут быть использованы только в описании семантики инструкции, где они считаются синонимами того экземпляра ресурсов, который они именуют (для примера выше `r` считается синонимом `R(i)`).

Имена всех параметров, как кодовых, так и семантических, должны различаться.

Дальнейшие примеры:

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

Параметры `i` и `addr` — кодовые длины 4 и 16 бит соответственно, параметры `dst` и `src` — семантические, связанные с экземплярами `R(i)` и `mem(addr)` соответственно.

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

Параметр `j` — кодовый длины 4.

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

Параметр `offs` — кодовый длины 8.

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

Ошибочное описание, поскольку тип `IndirIndex` не является ресурсом.

4.6.2 Выражения и термы

Выражения, используемые в описании семантики инструкций, состоят из знаков бинарных и унарных операций, применений функций, операций конкатенации и триммеров, операции доступа к полям ресурса, констант, литералов и термов:

(68) *term ::*
term-primary(69:19) [**[** *decimal*(12:4)**]**]
| *term-primary* **.** *field-identifier*(5:3)
;;

(69) *term-primary ::*
resource-identifier(5:3) [**(** *expression-list*(86:25) **)**]
;;

(70) *expression ::*
expression-I(71:19)
| *expression-I* **|** *expression*
;;

(71) *expression-I ::*
expression-II(72:19)
| *expression-II* *binary-operation*(18:5) *expression-I*
;;

(72) *expression-II ::*
unary-operation(18:5) *expression*(70:19)
| *function-keyword*(10:3) **(** *expression-list*(86:25) **)**
| *expression* *trimmer*(40:9)
| *primary*(73:19)
;;

(73) *primary ::*
size
| *constant*(11:4)
| *literal*(45:10)
| *substruct*(74:19)
| *term*(68:19)
| **(** *expression* **)**
;;

(74) *substruct :: term-primary*(69:19) **with** **{** *substruct-fields*(75:19) **}** *;;*

(75) *substruct-fields :: substruct-field*(76:19) ⋆ **,** *;;*

(76) *substruct-field :: field-identifier*(5:3) = *expression*(70:19) *;;*

Операция доступа к полям ресурса (*substruct*) используется для того, чтобы изменить сразу несколько полей в значении ресурса с полями. Например, если дано описание ресурса

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

то терм `Scale with {A=0:3, C=5:3}` является сокращенной формой записи выражения `0:3 | Scale.B | 5:3`.

В качестве терма могут выступать:

- идентификатор параметров инструкции;
- идентификатор ресурсов;
- выборка экземпляра ресурса в виде '`t(e)`', где '`t`' — терм, соответствующий ресурсу, '`e`' — выражение для адреса;
- выборка поля для ресурса с полями в виде '`t.F`', где '`t`' — терм, соответствующий ресурсу, '`F`' — имя поля;
- агрегат в виде '`t[n]`', где '`t`' — терм, соответствующий ресурсу, '`n`' — количество экземпляров для агрегирования;

В состав выражений может входить специальный символ **size**, который обозначает статическую константу, равную длине машинного представления текущей инструкции в экземплярах ресурса, хранящего программу.

Агрегирование позволяет получить доступ к нескольким последовательным экземплярам ресурса. При этом эти экземпляры группируются в соответствии с правилом группировки, задаваемым в описании данного ресурса.

4.6.3 Типы выражений

Все выражения в описании семантики инструкции должны иметь тип $[n]$. Ниже приводится список унарных и бинарных операций с указанием их типов. Операции разбиты на группы в соответствии с приоритетом, внутри группы приоритеты одинаковы, группы отсортированы по возрастанию приоритетов:

1. Логические операции:
   - `'&'` : $([n],[n]) \to [n]$ — побитовое 'и';
   - `'?'` : $([n],[n]) \to [n]$ — побитовое 'или';
   - `'@'` : $([n],[n]) \to [n]$ — побитовое исключающее 'или';
2. Операции сравнения:
   - `'<'` : $([n],[n]) \to [1]$ — сравнение на меньше;
   - `'>'` : $([n],[n]) \to [1]$ — сравнение на больше;
   - `'<='` : $([n],[n]) \to [1]$ — сравнение на меньше или равно;
   - `'>='` : $([n],[n]) \to [1]$ — сравнение на больше или равно;
   - `'=='` : $([n],[n]) \to [1]$ — сравнение на равно;
   - `'<>'` : $([n],[n]) \to [1]$ — сравнение на не равно;
3. Сложение и вычитание:
   - `'+'` : $([n],[n]) \to [n+1]$ — сложение;
   - `'-'` : $([n],[n]) \to [n+1]$ — вычитание;

Операция '-' — левоассоциативная.

4. Умножение, деление и сдвиги:

   - '/' : $([n], [n]) \to [2n]$ — деление с остатком. Левые n бит результата хранят частное, правые n — остаток;
   - '*' : $([n], [n]) \to [2n]$ — умножение;
   - '<<' : $([n], *) \to [n]$ — логический сдвиг влево. Свободная позиция справа заполняется нулем;
   - '<*' : $([n], *) \to [n]$ — арифметический сдвиг влево. Свободная позиция справа заполняется нулем;
   - '>>' : $([n], *) \to [n]$ — логический сдвиг вправо. Свободная позиция слева заполняется нулем;
   - '*>' : $([n], *) \to [n]$ — арифметический сдвиг вправо. Свободная позиция слева заполняется содержимым знакового бита;

   Правым операндом операций сдвига может быть только десятичная константа (*decimal*). Количество позиций для сдвига не может превышать ширину левого (битового) операнда. Операция / — левоассоциативная.

5. Унарные операции:

   - '~' : $([n]) \to [n]$ — двоичное дополнение;
   - '-' : $([n]) \to [n]$ — арифметическое дополнение;
   - '!' : $([1]) \to [1]$ — логическое отрицание;

Кроме данных операций, в состав выражения могут входить следующие конструкции:

- 'ext' : $([n], *) \to [m]$ — беззнаковое расширение;
- 'sext' : $([n], *) \to [m]$ — знаковое расширение;

Обе эти функции должны получать в качестве второго аргумента десятичную константу (*decimal*). Значение этой константы не может быть меньше, чем длина первого (битового) операнда.

- 'trunc' : $([n], *) \to [m]$ — беззнаковое усечение;
- 'strunc' : $([n], *) \to [m]$ — знаковое усечение;

Обе эти функции должны получать в качестве второго аргумента десятичную константу (*decimal*). Значение этой константы не может быть больше, чем длина первого (битового) операнда.

Типовые правила для триммеров, конкатенации, констант и литералов аналогичны таковым в кодовых выражениях (см. раздел 4.3.2, стр. 10).

Битовая длина термов, входящих в состав выражений, вычисляется следующим образом:

1. длина экземпляра ресурса определяется исходя из его описания;

2. длина поля экземпляра ресурса определяется исходя из описания этого поля;

3. длина агрегата ресурса определяется как произведение длины экземпляра этого ресурса на число экземпляров в агрегате;

Битовая длина выражения, используемого в качестве адреса для выборки экземпляра ресурса, должна быть равна длине адреса для этого ресурса. Адрес для выборки должен всегда присутствовать при обращению к ресурсу, если его мощность превосходит 1, и отсутствовать в противном случае. Количество агрегируемых экземпляров ресурса не может превосходить его мощности.

Битовая длина кодовых параметров инструкции определяется так, как это описано в разделе 4.6.1 (стр. 17).

Битовая длина выражения **size** полагается равной длине ресурса, описанного с атрибутом **ip**.

Примеры:

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

Выражение имеет тип [3].

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

Выражение имеет тип [9].

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

Выражение имеет тип [9].

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

Выражение имеет тип [1].

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

Выражение имеет тип [32].

Если выражение `e` имеет тип $[n]$ в контексте, требующем тип $[m]$ и $m > n$, то `e` по умолчанию будет приведено к нужному типу с помощью конструкции `sext(e, m)`. Например, выражение

```
mem(512)
```

автоматически будет приведено к виду

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

В случаях, когда умолчательные приведения не подходят, следует пользоваться явными преобразованиями.

4.6.4 Ограничения

Ограничения — это выражения специального вида, которые могут употребляться только в описании параметров инструкции.

Синтаксически ограничения определяются как обычные выражения (expression$_{(70:19)}$), однако их типовая структура, разрешенные операнды и операции отличаются от таковых в выражениях, описывающих семантику инструкции.

Подвыражения ограничения могут быть одного из двух типов — *Numeric* (универсальный числовой) или *Boolean* (логический). При этом само ограничение обязано иметь тип *Boolean*.

Операндами в ограничениях могут быть либо кодовые параметры (см. стр. 18), либо десятичные константы (decimal$_{(12:4)}$). Операндам приписывается тип *Numeric*.

Операции, разрешенные в ограничениях, перечислены ниже. Они разбиты на группы в соответствии с приоритетом, внутри группы приоритеты одинаковы, группы отсортированы по возрастанию приоритетов:

1. Логические операции:
   - `'&'` : $(Boolean, Boolean) \to Boolean$ — логическое 'и';
   - `'?'` : $(Boolean, Boolean) \to Boolean$ — логическое 'или';
   - `'@'` : $(Boolean, Boolean) \to Boolean$ — логическое исключающее 'или';
2. Операции сравнения:
   - `'<'` : $(Numeric, Numeric) \to Boolean$ — сравнение на меньше;
   - `'>'` : $(Numeric, Numeric) \to Boolean$ — сравнение на больше;
   - `'<='` : $(Numeric, Numeric) \to Boolean$ — сравнение на меньше или равно;
   - `'>='` : $(Numeric, Numeric) \to Boolean$ — сравнение на больше или равно;
   - `'=='` : $(Numeric, Numeric) \to Boolean$ — сравнение на равно;
   - `'<>'` : $(Numeric, Numeric) \to Boolean$ — сравнение на не равно;
3. Сложение и вычитание:
   - `'+'` : $(Numeric, Numeric) \to Numeric$ — сложение;
   - `'-'` : $(Numeric, Numeric) \to Numeric$ — вычитание;

   Операция '-' — левоассоциативная.
4. Умножение и деление:
   - `'/'` : $(Numeric, Numeric) \to Numeric$ — неполное частное при делении с остатком.
   - `'%'` : $(Numeric, Numeric) \to Numeric$ — остаток при делении с остатком.
   - `'*'` : $(Numeric, Numeric) \to Numeric$ — умножение;

5. Унарные операции:

   - '-' : *Numeric* → *Numeric* — арифметическое дополнение;
   - '!' : *Boolean* → *Boolean* — логическое отрицание;

4.6.5 Тело инструкции

Тело инструкции состоит из описания семантики (*action-specification*), правила порождения двоичного кода (*code-specification*) и правила порождения текстового представления (*repr-specification*).

Правила пророждения кода и текстового представления являются соответственно форматными и ассемблерными кодовыми выражениями, в которые могут входить кодовые параметры. При этом в форматном выражении они трактуются как битовые шкалы соответствующей длины, а в ассемблерном — как строки, содержащие строковое представление параметров в том же формате, что и константы PADLA.

Пример:

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

Тип значения форматного кодового выражения в опсании инструкции должен быть $[n]$, ассемблерного — *string*. При этом регулярное выражение, являющее значением ассемблерного выражения, трактуется как правило текстового представления данной инструкции.

Описание семантики инструкции задает ее действия над ресурсами:

(77) *action-specification* :: *sequential*(78:24) ;;

(78) *sequential* ::
    parallel(79:24)
  | *parallel* **;** *sequential*
;;

(79) *parallel* :: *ground-construction*(80:24) ⋆ **||** ;;

(80) *ground-construction* ::
    let-binding(81:24)
  | *assignment*(84:24)
  | *conditional*(87:25)
  | *loop*(89:25)
  | *halt*(90:25)
;;

(81) *let-binding* :: **let** *bind-list*(83:24) **in** *binding-area*(82:24) ;;

(82) *binding-area* :: *sequential*(78:24) ;;

(83) *bind-list* :: *identifier*(5:3) ⋆ **,** **=** *expression-list*(86:25) ;;

(84) *assignment* :: *term-list*(85:24) **<-** *expression-list*(86:25) ;;

(85) *term-list* :: *term*(68:19) ⋆ **,** ;;

(86) *expression-list* :: *expression*$_{(70:19)}$ $\star$ **,** ;;

(87) *conditional* :: **if** *logic-expression*$_{(70:19)}$ *conditional-tail*$_{(88:25)}$;;

(88) *conditional-tail* ::
  then *sequential*$_{(78:24)}$ **fi**
  | **then** *sequential* **else** *sequential* **fi**
  | **then** *sequential* **elif** *logic-expression*$_{(70:19)}$ *conditional-tail*
;;

(89) *loop* ::
  while *logic-expression*$_{(70:19)}$ **do** *sequential*$_{(78:24)}$ **od**
;;

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

В описании семантики простые конструкции (*ground-construction*) сгруппированы в последовательное (*sequential*) и параллельное (*parallel*) исполнения. Разделителем последовательного исполнения является '**;**', параллельного — '||'.

Параллельное исполнение $s_1||s_2$ считается корректным в том случае, когда получатели присваиваний, входящих в s_1, отличны от получателей присваиваний, входящих в s_2. Например, следующая параллельная конструкция корректна:

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

В то же время параллельная конструкция

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

некорректна, поскольку в обоих ее членах имеется присваивание ресурсу `R(i)`.

Неформально говоря, исполнение параллельной конструкции происходит в три этапа. Сначала читаются значения всех аргументов, затем происходит вычисление значений всех выражений, после чего вычисленные значения присваиваются получателям присваиваний. Таким образом семантически параллельная конструкция позволяет описать граф потока данных. Граф потока данных для расмотренного выше примера корректной параллельной конструкции изображен на рис. 1.

Последовательная конструкция по возможности приводится к параллельной. Эта возможность возникает тогда, когда удается проверить корректность получаемой параллельной конструкции. Например, последовательное исполнение

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

приводится к параллельной форме

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

поскольку очевидно, что получатели в обоих присваиваниях (`R(i)` и `mem(addr)`) различны.

Корректность параллельной конструкции может быть проверена не всегда. В некоторых случаях положение может спасти введение ограничений на параметры инструкций (constraint$_{(59:16)}$). Так, параллельная конструкция

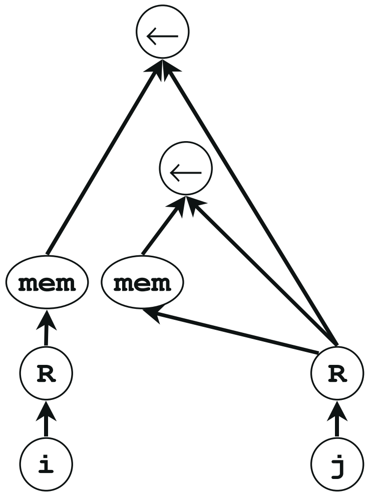

Рис. 1: Пример графа потока данных при параллельном исполнении

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

корректна, если i<>j. В общем же случае в ситуации, когда проверить корректность невозможно, необходимо пользоваться последовательной конструкцией (которая, очевидно, в этом случае не будет приведена к параллельной форме).

Простыми конструкциями являются связывание (*let-binding*), присваивание (*assignment*), условная (*conditional*) и циклическая (*loop*) конструкции.

Связывание позволяет ввести синонимы для выражений. При этом вхождения связанных идентификаторов в выражения внутри области связывания (*binding-area*) заменяются на те выражения, с которыми они связаны. Таким образом, конструкция

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

полностью эквивалентна конструкции

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

В присваивании слева от знака '<-' находится список термов (получателей), справа — список выражений (источников). Количество элементов в обоих списках должно быть одинаково, типы соответствующих элементов в обоих списках должны совпадать. При необходимости источник присваивания будет по умолчанию приведен к нужному типу (либо знаково расширен, либо беззнаково усечен). Таким образом, присваивания

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

будут автоматически преобразованы к виду

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

Условная конструкция позволяет осуществить ветвление в зависимости от значения выражения, имеющего тип [1]. Например:

```
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
```

Аналогичным образом циклическая конструкция позволяет осуществить повторное исполнение группы действий в зависимости от истинности некоторого условия. Выражение, использующееся в качестве условия, должно иметь тип [1]. Пример:

```
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
```

Конструкция halt(90:25) предназначена для описания ситуации остановки процессора. Например, выражение

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

останавливает процессор при попытке исполнения инструкции по адресу 0.

Предметный указатель