Язык описания макроархитектуры для технологии совместной программно-аппаратной разработки

Д.Ю.Булычев
db@tepkom.ru
Санкт-Петербургский государственный университет
198504, Университетский пр., 28
Санкт-Петербург, Россия

Аннотация

В статье рассматривается язык описания макроархитектуры программируемых вычислительных устройств, развитый в рамках проекта построения комплекса инструментальных средств для поддержки процесса совместной разработки программно-аппаратных систем. Данный язык представляет собой формализм промежуточного уровня, который позволяет описать целевой процессор с точки зрения его системы команд. Степень детализации такого описания делает возможным как автоматический синтез модели микроархитектуры (например в виде VHDL-описания), так и порождение технологических средств для разработки программ (ассемблер, симулятор и т.д.), настроенных на данную архитектуру.

Введение

Многообещающее и активно развиваемое научным сообществом направление совместной программно-аппаратной разработки встроенных систем (hardware-software codesign) основано на идее о взаимозависимом проектировании целевого программного обеспечения и его аппаратной платформы с целью получения системы, оптимальной относительно набора определенных критериев. В силу сложности и взаимовлияния этих критериев использование данного подхода требует, как правило, мно-

гократного прототипирования целевой системы в разных обстановках, в том числе и на различных аппаратных платформах.

Последнее означает, что стадия настройки (retargeting) средств программирования — компилятора, ассемблера, линкера, симулятора и т.д. — становится частой, базовой операцией в цикле разработки. Таким образом, традиционные средства "раскрутки", обеспечивающие полуавтоматическое решение данной задачи, перестают удовлетворять требованиям предметной области и задача автоматизации настройки средств программирования на систему команд целевого процессора оказывается чрезвычайно актуальной. С другой стороны, использование специфических черт реализации алгоритмов целевого программного обеспечения подразумевает специализацию системы команд процессора для его нужд. Поскольку задача извлечения системы команд из поведенческого описания микроархитектуры, вообще говоря, представляется технически неразрешимой, для решения обеих поставленных задач необходим формализм, описывающий вычислительное устройство с точки зрения его системы команд. Общее описание технологии, основанной на таком принципе, содержится в [2].

Данная статья представляет язык описания макроархитектуры PADLA (Processor Architecture Description Language), позволяющий охарактеризовать целевое устройство в такой степени общности, которая достаточна для автоматической генерации средств разработки программного обеспечения, и на таком уровне детализации, который делает возможным синтез спецификации микроархитектуры этого устройства.

1 Аналогичные разработки

Существует большое количество формализмов, описывающих модель вычислительного устройства на различных уровнях, в том числе и на уровне системы команд. Однако большинство из них изначально были ориентированы на более узкое использование и поэтому здесь не рассматриваются. Подробнее с ними можно ознакомиться в [1].

Ближайшими прямыми аналогами языка описания макроархитектуры являются формализмы nML и λ-RTL.

Язык описания архитектуры nML [4, 5] создан в Техническом

университете г. Берлина в конце 1980-х — начале 1990-х гг. nML является ядром технологии CHESS [6], которая позиционируется ее разработчиками как полное и законченное решение в данной области.

Текст на nML содержит описание всех основных компонентов машинной архитектуры с точки зрения системы команд (памяти, регистров, режимов адресации). Язык позволяет выразить правила порождения двоичного кода и ассемблерного представления инструкций, а также специфицировать семантику инструкций в императивных терминах.

Интересной особеностью nML является то, что для описания системы команд использован механизм атрибутных грамматик. Машинная команда при описании на nML рассмаривается как результат вывода из некоторого нетерминала в грамматике специального вида. Правило такой грамматики семантически соответствует группе частично определенных команд с общими свойствами. Нетерминалы в грамматике nML-описания снабжены атрибутами, что предоставляет способ связать с ними некоторую дополнительную информацию — например правила кодирования, представление в виде ассемблерного текста и т.д. Каждое правило может требовать наличия каких-либо атрибутов у входящих в его состав нетерминалов и описывать процедуру их преобразования для вычисления аналогичного или другого атрибута у выводимого нетерминала, что обеспечивает возможность определения различных характеристик команд путем распространения атрибутных значений “снизу вверх” по дереву вывода. В частности, описание семантики команды является одним из атрибутов нетерминала, из которого эта команды выводится. Такая организация nML позволяет описывать систему команд в сжатой форме, хотя и вводит дополнительные зависимости между описаниями разных команд.

В первоначальном своем варианте язык nML не предназначался для генерации описания микроархитектуры. В последнее время, однако, предпринимаются попытки расширения языка с этой целью [9], в том числе путем введения в nML возможности указания деталей микроархитектурной реализации.

Язык описания семантики машинных команд λ-RTL [7] разрабатывается в рамках проекта Zephyr в унивеситете Вирджинии, США. Данный проект входит в состав проекта

NCI (National Compiler Infrastructure), целью которого является создание универсальной переносимой многоязыковой и многоплатформенной среды компиляции.

В основе λ-RTL лежит понятие "список регистровых передач" (RTL, Register Transfer List), который можно интерпретировать как список элементарных операций. Сам λ-RTL можно рассматривать как метаязык, позволяющий в сравнительно компактной форме описывать сложные комбинации RTL-выражений. При этом осуществляется контроль типов на основе их вывода в специальной типовой системе, что в случае успеха гарантирует формальную корректность.

Несмотря на концептуальную целостность, λ-RTL не предоставляет средств для исчерпывающего описания макроархитектуры. В частности, отсутствует возможность описания ассемблерного синтаксиса и правил кодирования машинных команд. Это объясняется тем, что сам λ-RTL является частью семейства языков CSDL (Computer Systems Description Languages), которое включает в себя отдельные формализмы для описания различных свойств макроархитектуры [8]. Однако интерфейс между формализмами CSDL на сегодняшний день не разработан.

2 Язык описания макроархитектуры

Предлагаемый нами формализм во многом заимствует черты как λ-RTL, так и nML. Например, принцип ортогонального описания инструкций и явной спецификации всех побочных эффектов, полиморфная типовая система и вывод типов напоминают λ-RTL; в то же время, соединение описания семантики, правил порождения ассемблерного текста и двоичного кода, процедурное описание инструкций, описание ресурсов и их свойств аналогичны таким же средствам языка nML.

Отсутствие в λ-RTL средств описания правил двоичного кодирования и текстового представления инструкций составляет настолько существенное его отличие от предлагаемого языка, что анализ более мелких расхождений неуместен, поэтому мы сконцентрируемся на отличиях между описываемым языком и nML.

Принципиальные отличия таковы:

1. описание на nML может содержать неформальные элементы (обращения к сторонним функциям). Это упрощает задачу построения симулятора (для чего nML первоначально и создавался), однако порождает трудности для синтеза микроархитектуры и построения кодогенератора. Предлагаемый же нами формализм позволяет создавать только полностью формальные, функционально замкнутые описания;

2. в отличие от nML никакие детали микроархитектуры (т.е. свойства, невыразимые на уровне системы команд) не могут быть специфицированы средствами описываемого языка. С одной стороны, это осложняет построение симулятора, поскольку не позволяет учесть параллелизм на уровне инструкций, с другой стороны, оставляет большую свободу для синтеза микроархитектуры. Заметим также, что параллелизм на уровне инструкций и конвейеризация — это свойства именно микроархитектурной реализации, в значительной степени не имеющие отношения к системе команд. Поэтому все попытки увязать вместе два уровня в рамках единого формализма ведут к компромиссам, усложняя описание макроархитектуры, сужая возможности порождения эффективной низкоуровневой модели и все равно не позволяя добиться точного поведения симулятора.

Помимо указанных отличий существует ряд более мелких. Например, среди конструкций, предназначенных для описании семантики на nML, отсутствует параллельное исполнение, невозможно описать ограничения на значения параметров, фиксирован формат представления чисел и т.д.

Спецификация макроархитектуры в предлагаемом нами языке состоит из описания ресурсов, типов, инструкций и правил порождения текстового представления и двоичного кода. Более подробное определение языка содержится в [3].

2.1 Ресурсы

Ресурс — это место для хранения значений. В некотором смысле ресурсы аналогичны переменным в языках программирования.

С точки зрения архитектуры процессоров ресурсы позволяют описывать банки памяти, регистровые файлы и т.д.

Ресурсы могут иметь следующие свойства:

- мощность, т.е. число экземпляров;
- тип хранимого значения;
- принцип агрегирования.

Типом хранимого значения может быть либо битовая шкала, либо набор полей заданной ширины. Например, описание

```
resource
   mem (0..2^16-1) : 8;
```

задает ресурс `mem`, имеющий 65536 экземпляров, каждый из которых хранит 8-битовое значение. Описывающий мощность диапазон должен быть задан с помощью статических выражений, при этом он трактуется как диапазон возможных адресов. Если ресурс существует в единственном экземпляре, то для него диапазон адресов не указывается.

Если значение, которое можно хранить в экземпляре ресурса, допускает доступ к своим частям, тип ресурса может быть описан как набор полей:

```
resource
   AX : {AH : 8, AL : 8};
```

В данном случае ресурс `AX` обладает единственным экземпляром, который разбит на два поля — `AH` и `AL`. При обращении к полям они должны быть указаны привычным образом: `AX.AH` или `AX.AL`. В то же время, ресурс доступен и целиком через идентификатор `AX`.

Принцип агрегирования описывает правило, по которому группируются последовательные экземпляры ресурса. Такая группировка возникает при использовании конструкции "`[]`". Например, выражение

```
mem (x) [2]
```

обозначает, что два экземпляра ресурса `mem` с адресами `x` и `x+1` группируются, образовывая значение шириной 16 бит. При этом возможно два подхода:

- младшему адресу соответствует *старшая* часть значения (big-endian);
- младшему адресу соответствует *младшая* часть значения (little-endian).

Правило агрегирования задается в описании ресурса после типа значения. По умолчанию ресурс считается big-endian (`big`), little-endian ресурсы должны быть явно описаны с применением ключевого слова `little`, например

```
resource
   cache (0..126*1024-1) : 8 little;
```

Среди всех ресурсов в описании макроархитектуры обязательно должен присутствовать указатель на текущую инструкцию. Таковым считается ресурс, имеющий 1 экземпляр и описанный со специальным атрибутом `ip`:

```
resource
   PC : 16 ip (mem);
```

Идентификатор в скобках должен указывать на ресурс, который хранит исполняемую программу. Этот ресурс не может иметь мощность 1.

2.2 Типы

Тип является аналогом режима адресации. Поскольку сама организация доступа к ресурсам при различных режимах адресации определяется на уровне микроархитектуры, с точки зрения макроархитектуры режим адресации может быть рассмотрен как способ группировки параметров инструкции.

Описание типов не является необходимой частью в описании макроархитектуры, однако при спецификации уже существующих процессоров их введение облегчает работу.

С помощью языка описания макроархитектуры может быть введен тип битовой шкалы или составной тип. Например, описание

```
type
  addr : 16;
  imm  : 16;
  IndirOffs (addr, imm);
```

определяет три типа: битовые шкалы `addr` и `imm` шириной 16 бит и составной тип `IndirOffset` с параметрами типов `addr` и `imm`. Фактически составной тип служит для именования сгруппированных простых типов.

Для удобства возможно описание типа с помощью задания диапазона значений. При этом необходимое количество бит вычисляется автоматически в соответствии с указанной схемой представления чисел:

```
type
  const : 0..15;
```

Для ресурсов мощности больше 1 автоматически определяется одноименный составной тип с одним параметром, имеющим ту же ширину, что и адрес для данного ресурса. Например, описание

```
resource
   R (0..15) : 16;
```

автоматически вводит составной тип `R`, имеющий один параметр ширины 4.

2.3 Выражения

Выражения могут быть построены из экземпляров ресурсов, констант и символических имен с помощью обычных арифметических операций, которые полиморфны относительно битовой ширины своих операндов.

Заметим, что все эти операции семантически рассматриваются как функции над битовыми строками. Таким образом, средствами данного формализма могут быть описаны архитектуры, использующие практически произвольные схемы представления

чисел. Для настройки на конкретную схему реализация должна предоставить лишь небольшой набор зависимых от этой схемы конструкций (преобразования десятичного числа в битовую строку, вычисления количества бит, необходимых для представления данного числа и т.д.).

Обозначим через $[n]$ тип битовой шкалы ширины n, а через $([n],[m]) \to [k]$ — тип бинарной операции, получающей операнды типов $[n]$ и $[m]$ соответственно и возвращающей значение типа $[k]$ (тип для унарной операции имеет аналогичное обозначение $[n] \to [m]$). 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. Количество позиций для сдвига не может превышать ширину левого (битового) операнда. Операция '/' — левоассоциативная.

5. Унарные операции:
   - $'\sim' : ([n]) \to [n]$ — двоичное дополнение;
   - $'-' : ([n]) \to [n]$ — арифметическое дополнение;
   - $'!' : ([1]) \to [1]$ — логическое отрицание.

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

- $'\texttt{ext}' : ([n], *) \to [m]$ — беззнаковое расширение;
- $'\texttt{sext}' : ([n], *) \to [m]$ — знаковое расширение;
- $'\texttt{trunc}' : ([n], *) \to [m]$ — беззнаковое усечение;
- $'\texttt{strunc}' : ([n], *) \to [m]$ — знаковое усечение.

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

Заметим, что результаты всех арифметических операций вычисляются без потери точности. Выражение `R(0)+R(1)+R(2)`, строго говоря, не является корректным с точки зрения типов. Действительно, результат второго сложения имеет тип [17] и не может быть непосредственно использован как агрумент первого сложения, поскольку его левый операнд (`R(0)`) имеет тип [16]. Таким образом, правильной является запись

`sext(R(0),17)+R(1)+R(2)`, при которой расширение операнда до нужной длины описано явно. К счастью, это расширение обычно генерируется автоматически по эвристически определенным правилам (см. [3]), однако не всегда это означает то, что имелось в виду. Поэтому при написании выражений необходимо помнить об этой тонкости и явно выписывать приведения операндов, если они отличаются от принятых по умолчанию.

В состав выражения могут входить числовые константы и экземпляры ресурсов. Константы, в свою очередь, могут быть десятичными (`256`), двоичными (`0b10000000`), восьмиричными (`0o400`) и шестнадцатеричными (`0x100`). Кроме того, для константы может быть явно указана битовая длина (например, `10:16` — десятичная константа 10, занимающая 16 бит). Десятичная константа преобразуется в битовую шкалу в соответствии с принятой схемой кодирования чисел. Что же касается констант по остальным основаниям, то они представляют собой сокращенную запись битовой шкалы, в которой каждая цифра обозначает соответствующее число бит.

Экземпляры ресурсов могут быть использованы в выражениях привычным образом, например `AX`, `R(1)`, `mem(R(0))` и т.д. При этом для ресурсов мощности больше 1 должно обязательно быть указано выражение, определяющее адрес экземпляра. Экземпляры ресурса могут быть сгруппированы как было описано выше.

Помимо числовых констант и ресурсов в состав выражений могут входить *триммеры* и символические константы.

Триммеры предоставляют возможность работы с битовыми шкалами как со строками. Например, выражение `R(0)[:3]` обозначает выборку 3-х правых бит из `R(0)`. Допустимы также выборка левых n-бит (например `R(0)[10:]`) и нескольких средних бит (например `R(0)[3:5]`). Кроме того, значения выражений могут быть сконкатенированы применением операции "`|`" (например `R(0) | R(1)`).

Символические константы — это идентификаторы, которые имеют некоторое специальное статически определяемое значение. Например, идентификатор `size` обозначает длину бинарного представления текущей команды в экземплярах ресурса, который хранит программу.

2.4 Императивные конструкции

Императивная часть языка описания макроархитектуры представлена следующими конструкциями:

- присваиванием;
- последовательным и параллельным исполнением;
- условной и циклической конструкциями;
- конструкцией связывания;
- конструкцией останова.

Присваивание имеет вид `получатель <- источник`. В качестве получателя может выступать экземпляр ресурса или поле этого экземпляра, в качестве источника — выражение. Если источник в присваивании имеет иной тип, нежели получатель, то либо он автоматически знаково расширяется, либо у него отбрасывается старшая часть. Таким образом,

```
PC <- PC+size
```

эквивалентно

```
PC <- strunc (PC+size, 16)
```

а

```
AX <- 1:2
```

эквивалентно

```
AX <- sext (1:2, 16)
```

Разделителем при параллельном исполнении является символ “||”, а при последовательном — “;”.

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

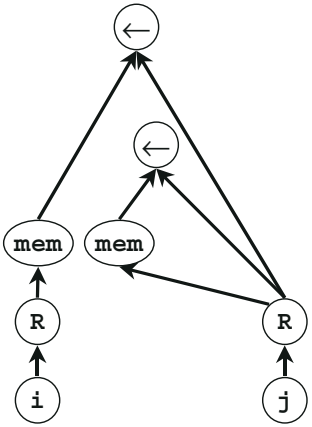

Рис. 1: Пример графа потока данных при параллельном исполнении

```
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 осуществляется 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 рисунке 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 можно проверить не всегда. В некоторых случаях положение спасет введение ограничений на параметры инструкций (см. ниже). Так, параллельная конструкция

```
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).

Помимо простого присваивания в языке описания макроархитектуры существует форма множественного присваивания, которая имеет вид

$$r_1, r_2, \ldots, r_k \quad \texttt{<-} \quad e_1, e_2, \ldots, e_k$$

где r_i — экземпляры ресурсов, e_i — выражения. Такая форма присваивания является сокращением для параллельной конструкции

$$r_1\texttt{<-}e_1 \quad || \quad r_2\texttt{<-}e_2 \quad ||\ldots|| \quad r_k\texttt{<-}e_k$$

Условная и циклическая конструкции аналогичны таковым в обычных языках программирования. В качестве условия в них может выступать любое выражение, имеющее тип [1]. Например, допустимы следующие конструкции:

```
if AX == 0 then PC <- label else PC <- PC+size fi
```

```
while R(i) <> 0 do
  mem (to + R(i)) <- mem (from + R(i));
  R(i) <- R(i)+1
od
```

Связывание позволяет ввести символические имена, являющиеся синонимами для выражений. Эти имена текстуально эквивалентны тем выражениям, с которыми они связаны. Например,

```
let x, y = R(i), R(j) in
x <- y || y <- x
```

эквивалентно

```
let x, y = R(i), R(j) in
R(i) <- R(j) || R(j) <- R(i)
```

Наконец, конструкция останова имеет вид

```
halt
```

и предназначается для спецификации останова процессора.

2.5 Инструкции

С точки зрения языка описания макроархитектуры инструкции — это именованные совокупности действий над ресурсами. Для описания инструкции необходимо задать ее имя, список параметров и тело. Обычно несколько разных инструкций могут разделять одну и ту же мнемонику, поэтому допускается описывать инструкцию, которая принимает различные списки параметров:

```
instruction mov:
   R(i), R(j) ->
     action {R(i) <- R(j) || PC <- PC+size}

   R(i), mem(addr) ->
     action {R(i) <- mem(addr)[2] || PC <- PC+size}
```

В данном случае описана инструкция `mov`, которая может принимать в качестве параметров либо два номера регистров, либо номер регистра и адрес в памяти, и осуществляет пересылку значения. Заметим, что поскольку битовая ширина регистра превосходит ширину ячейки памяти, для пересылки использовано агрегирование.

Аргументы инструкции задаются в виде списка типов, при этом в качестве параметров составного типа должны выступать идентификаторы, которые и рассматриваются как битовые параметры инструкции. В примере выше этими параметрами являются `i` и `j` ширины 4 в первом случае и `i` и `addr` ширины 4 и 16 соответственно — во втором.

Если среди параметров упомянут простой тип, то значение параметра связывается с идентификатором, который отделяется от имени типа двоеточием:

```
instruction mov:
   R(i), x:imm ->
     action {R(i) <- x || PC <- PC+size}
```

Здесь параметрами инструкции являются `i` ширины 4 и `x` ширины 16.

Помимо собственно перечисления аргументов инструкции может быть задано ограничение на ее параметры:

```
instruction add:
   R(i), R(j) where i<>j ->
     action {R(i) <- R(i)+R(j) || PC <- PC+size}
```

В данном случае ограничение констатирует, что инструкция `add` получает в качестве аргументов два различных регистра.

Наконец, список аргументов может быть пуст:

```
instruction nop:
   -> action {PC <- PC+size}
```

Тело инструкции в общем случае содержит описание ее семантики и правил построения ассемблерного текста и машинного кода. Семантика инструкции описывается в секции `action` и представляет собой императивную конструкцию. Средства задания правил построения машинного кода и ассемблерного представления будут описаны позже.

Дальнейшие примеры:

```
instruction jmp:
   label: imm -> action {PC <- label}

instruction jz:
   R(i), label: imm ->
      action
      {
        if R(i) == 0
          then PC <- label
          else PC <- PC+size
        fi
      }
```

```
instruction rjmp:
   offs: imm -> action {PC <- PC+offs}

instruction div:
   R(i), R(j) where j<>i & j<>i+1 & i<15 ->
      action {
        let pair = R(i) / R(j) in

        PC     <- PC+size    ||
        R(i)   <- pair [16:] ||
        R(i+1) <- pair [:16]
      }
```

2.6 Кодовые выражения

Кодовые выражения являются той частью языка описания макроархитектуры, которая предназначена для спецификации правил построения двоичного кода и текстового ассемблерного представления инструкций. Фактически кодовые выражения позволяют описать некоторую синтаксическую структуру, которая трактуется как регулярное выражение, порождающее двоичные шкалы в случае двоичного кода и строки — в случае текстового представления.

С целью осуществления семантического контроля для кодовых выражений введена система типов. Базовыми типами являются строки и битовые шкалы, а производными — n-ки и функции. Функции позволяют выразить систему форматов машинных инструкций, которая, как правило, присутствует в любой системе команд. Чтобы отличать функции над строками от функций над битовыми шкалами, они описываются раздельно и называются соответственно *ассемблерами* и *форматами*. Например,

```
asm binop <mnemo, first, second> =
  mnemo, first, ",", second
```

описывает способ ассемблерного представления бинарной операции. Здесь `binop` — имя ассемблера, `mnemo`, `first` и `second` — имена его параметров. Операция “,” обозначает конкатенацию через пробелы. В данном случае типы параметров ассемблера полагаются равными *string*, а тип `binop` —

$(string, string, string) \to string$. Если через `WS` обозначить регулярное выражение, которое порождает символы, считающиеся в языке целевого ассемблера пробелами, то значением применения `binop<M, F, S>` является регулярное выражение

```
M WS* F WS* "," WS* S
```

Для группировки значений с целью получения n-ки в кодовых выражениях служит конструкция "`{}`". Например, выражение

```
{"R", "0", "none"}
```

группирует три строковых значения (`"R"`, `"0"` и `"none"`) в тройку. Тип этой тройки можно обозначить через $\{string, string, string\}$.

Для доступа к элементу n-ки нужно через точку указать его номер: `x.3`. Поскольку это осуществимо для всех n-ок, содержащих не менее трех элементов, `x` можно приписать тип $\{\alpha, \beta, \gamma, \ldots\}$ (где α, β, γ — любые типы), который содержательно обозначает тип n-ки, включающей не менее 3-х элементов. Типом же `x.3` оказывается γ. Таким образом, кодовые выражения полиморфными относительно длины n-ки.

Например, если есть описание

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

то выражение

```
third <{"a", "b", {"c", "d"}, "e"}>
```

будет иметь тип $\{string, string\}$, а выражение

```
third <{"a", "b", "c"}>
```

— тип $string$.

В кодовые выражения, которые оперируют со строками, могут входить только операции конкатенации, конструирования и выборки элемента из n-ки, а также конструкция применения функции над кодовыми выражениями. Выражения, оперирующие с битовыми шкалами, могут кроме этого включать в себя триммеры. Поскольку выборка нескольких бит из битовой шкалы возможна для всех шкал, ширина которых больше некоторого фиксированного значения, битовые кодовые выражения

оказываются дополнительно полиморфными относительно длины битовой шкалы. Если через $[> n]$ обозначить тип битовой шкалы, содержащей не менее n бит, то типом триммера `x[:3]`, например, оказывается $[3]$, а типом `x` — $[> 3]$. Дальнейшие примеры:

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

Здесь слово `format` указывает на то, что `trim` — функция над битовыми шкалами. Тип `trim` — $[> 3] \to [3]$.

```
format join <x, y> = x y;
```

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

```
format error <x, y> = (x y) [:3];
```

Функция `error` не имеет типа (т.е. считается некорректной), поскольку выражение `x y` в данной ситуации имеет тип $[> 0]$.

Для удобства записи и дополнительного контроля в функциях над битовыми кодовыми выражениями допускается явная спецификация ширины параметров. Например, функция

```
format ok <x:3, y:2> = (x y) [:3];
```

имеет тип $([3], [2]) \to [3]$.

Наконец, операция конкатенации через пробел в битовых кодовых выражениях обозначает обычную конкатенацию.

3 Пример описания

В данном разделе мы рассмотрим короткий пример полностью описанной машинной архитектуры. Эта архитектура относится к типу RISC (Reduced Instruction Set Computer), ее описание демонстрирует взаимодействие компонент рассматриваемого языка.

Ресурсы данного процессора включают в себя один килобайт памяти (`mem`) с возможностью адресации по 8 бит, шестнадцать шестнадцатиразрядных регистров (`r`), указатель на текущую инструкцию (`pc`) и указатель вершины стека (`sp`):

```
resource
  mem (0..1023) : 8;
  r(0..15) : 16;
  pc       : 16 ip (mem);
  sp       : 10;
```

Кроме ресурсов в качестве операндов инструкций могут быть использованы шестнадцатибитовый непосредственный операнд (`imm`) и десятибитовые метки (адреса памяти, `lab`):

```
type
  imm : 16;
  lab : 10;
```

Группа инструкций передачи управления содержит инструкцию останова (`halt`), инструкцию условного перехода по единице в регистре (`j1`), инструкцию вызова (`call`) и инструкцию возврата (`ret`):

```
instruction stop:
  ->
   action {halt}
   code   {0x0f}
   repr   {"halt"}

instruction j1:
  r(i), x:lab ->
   action {
     if r(i) == 1 then pc <- x else pc <- pc+size fi
   }
   code   {2:2 i x}
   repr   {"j1", "r"i, ",", x}

instruction call:
  x:lab ->
   action {
     mem (sp) [2], sp <- pc+size, sp - 2 || pc <- x
   }
   code {31:6 x}
   repr {"call", x}
```

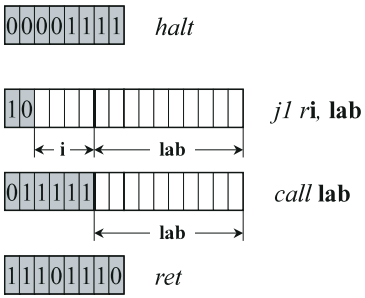

Рис. 2: Группа инструкций передачи управления

```
instruction ret:
  ->
   action {sp <- sp+2; pc <- mem (sp) [2]}
   code   {0xee}
   repr   {"ret"}
```

Форматы двоичного кодирования данных инструкций и их ассемблерное представление показаны на рисунке 2.

Группа инструкций пересылки содержит инструкцию загрузки константы в регистр (`ld`), инструкцию пересылки между регистрами (`mov`), инструкцию обмена содержимым регистров (`swap`) и инструкцию установки вершины стека (`st`):

```
instruction mov:
  r(i), x:imm ->
   action {r(i) <- x || pc <- pc+size}
   code   {1:4 i x}
   repr   {"ld", "r"i, ",", x}

  r(i), r(j) ->
   action {r(i) <- r(j) || pc <- pc+size}
   code   {2:8 i j}
   repr   {"mov", "r"i, ",", "r"j}
```

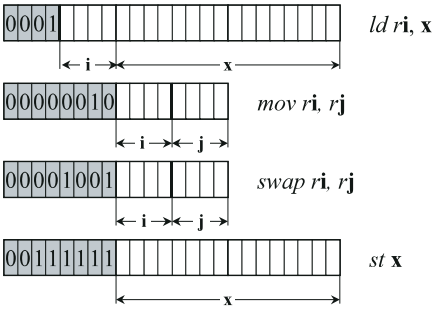

Рис. 3: Группа инструкций пересылки

```
instruction swap:
  r(i), r(j) ->
   action {
     r (i) <- r(j) || r(j) <- r(i) || pc <- pc+size
   }
   code   {0x0a i j}
   repr   {"swap", "r"i, ",", "r"j}

instruction st:
  x:imm ->
   action {sp <- x || pc <- pc+size}
   code   {0x3f x}
   repr   {"st" x}
```

Форматы двоичного кодирования данных инструкций и их ассемблерное представление показаны на рисунке 3.

Наконец, группа арифметических инструкций содержит инструкцию декремента (`dec`) и инструкцию умножения (`mul`):

```
instruction dec:
  r(i) ->
   action {r (i) <- r(i)-1 || pc <- pc+size}
   code   {0x0e i 0:4}
   repr   {"dec", "r"i}
```

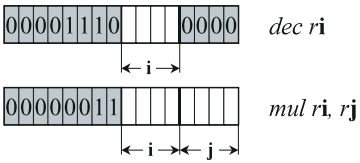

Рис. 4: Группа арифметических инструкций

```
instruction mul:
  r(i), r(j) ->
   action {
     r(i), pc <- trunc (r(i)*r(j), 16), pc + size
   }
   code   {3:8 i j}
   repr   {"mul", "r"i, ",", "r"j}
```

Форматы двоичного кодирования данных инструкций и их ассемблерное представление показаны на рисунке 4.

Видно, что в данной "игрушечной" архитектуре невозможно использование памяти иначе как для хранения значений на стеке; кроме того, отсутствуют инструкции работы со стеком (кроме `call` и `ret`), почти вся арифметика, работа с флагами и т.д. Очевидно также, что эти возможности легко могут быть добавлены введением новых инструкций с соответствующими семантиками.

Обсуждение

В заключение обсудим особенности использования данного языка для описания систем команд реальных процессоров.

Очевидно, что произвольную систему команд теоретически можно описать в рамках данного формализма просто введя для каждой команды и каждых значений ее операндов одну инструкцию без параметров. Однако такой способ описания является громоздким, а в ряде случаев и неприемлемым, поскольку спецификация команды, имеющей k операндов ширины $w_1, w_2, \ldots, w_k$ соответственно, может потребовать описания $2^{w_1} \times 2^{w_1} \times \ldots \times 2^{w_k}$ отдельных инструкций. В силу этого вопрос

о практической полноте предложенного формализма остается существенным.

В целях сокращения объема спецификации группа команд с одинаковыми семантиками и правилами кодирования может быть записана с помощью одной инструкции. Заметим, что такой принцип группировки с точки зрения процессора содержательно соответствует понятию “инструкция”, поскольку различные экземпляры такой инструкции для процессора неразличимы (ведь при декодировании и исполнении их он поступает совершенно одинаковым образом). Таким образом, с точки зрения размера описания все упирается в то, насколько компактно множество команд процессора разобъется на классы инструкций в соответствии с принципом единства семантики и правил декодирования.

Легко видеть, что средства описания семантики образуют алгоритмически полный набор, поэтому с их помощью можно описать инструкции, имеющие произвольную семантику. С другой стороны, язык описания двоичного кодирования и ассемблерного представления весьма небогат, что и накладывает основные ограничения на множество описываемых систем команд. Можно вообразить себе систему команд, в которой машинный код для инструкции получается путем процедуры, невыразимой через операции конкатенации и вырезки, применяемые над двоичным представлением операндов. В этом смысле данный формализм существенно неполон. Заметим, однако, что описываемая ситуация является, скорее, патологией, чем нормальным явлением, поскольку усложнение алгоритма декодирования влечет усложнение процессора. Аналогично, средства описания ассемблерного представления не позволяют специфицировать произвольный конкретный синтаксис языка ассемблера (это ограничение очевидно не является существенным).

Все эти замечания говорят о том, что мы имеем дело с низкоуровневым языком описания *внутреннего представления* системы команд, который не всегда удобен для непосредственного использования. В то же время, подобный формализм необходим, поскольку для достижения приемлемого качества системы в целом могут применяться преобразования, приводящие к системам команд, которые невыразимы (или трудновыразимы) с помощью формализмов более высокого уровня. Недостаточный

же уровень абстракции может быть компенсирован созданием надъязыков путем использования препроцессоров, систем расширения синтаксиса и др.

Отсутствие средств спецификации параллелизма на уровне инструкций (*instruction-level parallelism, ILP*) является не недостатком (или недоработкой), а логическим следствием того, что такой параллелизм есть свойство микроархитектурной реализации, слабо проявляющее себя на уровне системы команд.

Существенным недостатком рассматриваемого языка следует считать невозможность описания декомпозиции устройства на отдельные вычислительные блоки, реализующие самостоятельные системы команд и взаимодействущие путем передачи данных. Несмотря на то что такая декомпозиция кажется ортогональной по отношению к предлагаемому формализму, именно отсутствие средств спецификации внешних событий не позволяет в настоящий момент адекватно выразить такие существенные аспекты макроархитектуры, как обмен с внешними устройствами, прерывания и т.д. В то же время, введение таких черт неизбежно влечет за собой необходимость проработки вопросов, связанных с совместимостью интерфейсов этих блоков, правилами видимости их внутренности и т.д. Работа в данном направлении ведется.

Список литературы

[1] Булычев Д.Ю., Вигдорчик Е.Н., Ломов Д.С., Смирнов М.Н. Современные средства описания машинных архитектур: Технический отчет. — СПб.: Институт информационных технологий СПбГУ, 2001. — 35 с.

[2] Булычев Д.Ю. Разработка программно-аппаратных систем на основе описания макроархитектуры // Наст. сборник. — С. 7-22.

[3] Булычев Д.Ю. Язык описания макроархитектуры процессоров: Технический отчет. — СПб.: Институт информационных технологий СПбГУ, 2002. — 22 с.

[4] Fauth A. Beyond Tool-Specific Machine Description // Code Generation for Embedded Processors. — Boston/London/Dordrecht: Kluwer Academic Publishers, 1995. — P. 138-152.

[5] Freericks M. The nML Machine Description Formalism. Version 2.0. — http://www.techfak.uni-bielefeld.de/ags/ti/publications.html.

[6] Lanneer D., van Praet J. e.a. CHESS: Retargetable Code Generation for Embedded DSP Processors // Code Generation for Embedded Processors. — Boston/London/Dordrecht: Kluwer Academic Publishers, 1995. — P. 85-102.

[7] Ramsey N., Davidson J.W. Specifying Instructions' Semantics Using λ-RTL (Interim Report). — http://www.cs.virginia.edu/zephyr/csdl/lrtlindex.html.

[8] Ramsey N., Fernandez M.F. Specifying Representations of Machine Instructions // ACM Transactions on Programming Languages and Systems. — 1997. — Vol. 19. № 3. — P. 492-524.

[9] Sim-nML: Sim-nML Grammar Specification. — http://www.cse.iitk.ac.in/sim-nml/doc.html.