Синтез ассемблера и дизассемблера из описания макроархитектуры

А.А.Симановский
asimanovsky@yahoo.com

Д.А.Шапоренков
dsha@acm.org

Санкт-Петербургский государственный университет
198504, Университетский пр., 28
Санкт-Петербург, Россия

Аннотация

В статье рассматриваются средства автоматической генерации ассемблера и дизассемблера по описанию макроархитектуры, реализованные в рамках проекта PADLA. Оба генератора принимают на вход формальное описание макроархитектуры целевого процессора. Генератор ассемблера порождает описания лексики и синтаксиса целевого ассемблера в формате входных файлов для утилит `ocamllex` и `ocamllyacc`, генератор дизассемблера — исходный текст дизассемблера на языке Objective Caml. Процесс получения ассемблера и дизассемблера не требует вмешательства пользователя, но организован таким образом, чтобы предоставить возможность тонкой настройки дизассемблера с целью увеличения скорости его работы, а также расширения синтаксиса ассемблера пользовательскими конструкциями. Порождаемые ассемблер и дизассемблер поддерживают широко распространенный формат .elf.

Введение

В данной статье рассматривается вопрос автоматической генерации ассемблера/дизассемблера в рамках методологии быстрого создания нового устройства, описанной в [1]. Эта методология предлагает следующие действия:

- извлечение формального описания макроархитектуры (подробно это понятие обсуждается ниже) устройства из программы на языке высокого уровня;

- генерация по формальному описанию макроархитектуры средств разработки (ассемблера, дизассемблера, симулятора, линкера, загрузчика и т.д.), генерация и уточнение описания микроархитектуры устройства (VHDL-модель).

В разделе 1 дано сравнение с другими работами на данную тему; в разделе 2 кратко освещается используемый язык описания макроархитектуры и особенно те его элементы, которые важны с точки зрения ассемблера и дизассемблера. В разделах 3 и 4 подробно описана генерация ассемблера и дизассемблера.

1 Обзор работ по данной тематике

Вопросы автоматической генерации ассемблера и дизассемблера рассматриваются во многих публикациях, посвященных задаче разработки программно-аппаратных систем. В обзоре [11] проводится краткий сравнительный анализ распространенных языков описания машинных архитектур (Architecture Description Language, ADL). Автор выделяет три группы языков: языки структурного уровня, позволяющие описывать компоненты архитектуры и связи между ними; языки поведенческого уровня, предназначенные для описания синтаксиса, машинного представления и семантики инструкций; смешанные языки, объединяющие возможности первых двух классов. Язык, рассматриваемый в настоящей статье, относится к третьей группе по этой классификации.

В работе [10] приводится описание средства New-Jersey Machine Code Toolkit, предназначенного для автоматизации процесса перенацеливания на новую платформу утилит, работающих с машинным кодом. Предлагается язык для спецификации машинного представления инструкций, режимов адресации и других архитектурно-зависимых деталей набора инструкций. Описываемая в статье система по входной спецификации на этом языке порождает набор процедур на языке C, выполняющих декодирование инструкций и настройку адресов. Процедуры используют примитивы входных и выходных бинарных потоков из библиотеки, также входящей в состав системы. Следует отметить, что система использовалась при реализации машинно-

независимого отладчика `ldb` и оптимизирующего компоновщика `mld`.

Наиболее близким к используемому нами языку PADLA является язык nML, описываемый в [5]. В работе [6] рассказывается об экспериментальной системе, генерирующей из спецификаций на этом языке ассемблер, дизассемблер и симулятор. Работа имела своей целью проверить пригодность средств автоматической генерации для DSP-процессоров. В статье обсуждаются проблемы, возникающие при попытке описания на языке nML сложных аспектов архитектуры процессоров (отложенные переходы/присваивания, прерывания, работа со счетчиком инструкций).

Дальнейшим развитием языка nML стал язык Sim-nML [9]. Работы [8], [7] посвящены генерации ассемблера и дизассемблера по описаниям на этом языке. Генератор ассемблера, предлагаемый авторами, использует `lex` и `yacc`. Настраиваемый дизассемблер принимает на вход объектный файл и описание процессора на Sim-nML. В обеих работах приводятся детальные алгоритмы, используемые авторами.

Стоит отметить, что существуют также средства формального описания, предазначенные для параметризации некоторой шаблонной архитектуры и не допускающие большой свободы в определении компонент архитектуры и набора инструкций. Примером такого средства является используемый в проекте Trimaran язык HMDES и его расширения для описания набора инструкций [3].

В заключение упомянем, что определенные возможности настройки на архитектуру процессора имеются в широко распространенном на многих платформах ассемблере `as` [4].

2 Описание макроархитектуры с точки зрения ассемблера и дизассемблера

Под *макроархитектурой* понимается представление целевого устройства с точки зрения разработчика компилятора. В частности, для данной статьи важно, что макроархитектура — это представление устройства, доступное непосредственно при программировании на языке ассемблера устройства. Макроархи-

тектура включает следующие элементы: ресурсы (память, регистры) и набор команд различных форматов. Используемое описание макроархитектуры представляет собой текстовый файл, разбитый на секции. Структура секций отражает приведенную выше классификацию.

Секция `resource` содержит описание ресурсов, которые включают в себя регистры и различные виды памяти. Для каждого ресурса указана разрядность и другие семантические атрибуты. Ресурс, в котором располагается программа, помечен специальным признаком.

Секция `type` содержит описание возможных типов параметров инструкций. Типы различаются разрядностью.

Секция `format` описывает форматы инструкций. Данная секция, например, может содержать стандартные форматы для наборов однотипных команд (таких, как сложение, вычитание, различные виды сдвигов и т.д.).

Секция `asm` позволяет пользователю осуществлять группировку элементов описания синтаксиса инструкций. В частности, целесообразно устанавливать единый синтаксис для режимов адресации и других управляемых общей семантикой элементов синтаксиса ассемблера.

Последняя секция содержит описания машинных инструкций. Для каждой инструкции и формата может быть указана семантика инструкции, ее синтаксис и бинарное представление.

Описание синтаксиса инструкции в описании макроархитектуры позволяет увязать семантику и синтаксис инструкций, например, проверить наличие в синтаксисе языка ассемблера отражения таких деталей целевой архитектуры, как число операндов команд. Хотя описание синтаксиса инструкций не является обязательной частью описания макроархитектуры, оно, как и описание бинарного представления инструкции, необходимо для генерации ассемблера и дизассемблера. На рисунке 1 приведен пример описания “игрушечной” макроархитектуры.

Архитектура, описанная в примере, имеет следующие компоненты: память, адресация которой производится побайтово (старшие байты хранятся по меньшим адресам, так называемая big-endian архитектура); указатель следующей команды в памяти; шестнадцать 16-битных регистров; четыре инструкции: останов (`stop`), обмен содержимого регистров (`swap`), пересыл-

```
resource
  mem(0..1023):8 big;
  r (0..15)   :16;
  pc          :16 ip(mem);
type
  im : -2^15..2^15-1;
  lab: 0..1023;

instruction stop: ->
   action { halt }
   code   { 0x01 }
   repr   { "halt" }

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 mov:
   r(i), x:imm ->
   action {r(i) <- x || pc <- pc + size}
   code { 0x02 0x0 i x}
   repr {"ld", "r"i, ",", x}

instruction jmp:
   x: lab ->
    action {pc <- x}
    code { 0x0e 0x0:6 x }
    repr {"jmp", x}
```

Рис. 1: Пример описания архитектуры

ку (`mov`) непосредственного операнда в регистр и безусловный переход (`jmp`).

Инструкции ассемблера данной архитектуры будут иметь вид

```
jmp   2
ld   r1, 7
swap r2, r1
halt
```

Более подробные комментарии к языку описания макроархитектуры можно найти в статье [2].

3 Генерация ассемблера

Генерируемый ассемблер представляет собой утилиту, принимающую на вход текст программы на языке ассемблера и порождающую в качестве выхода файл формата .elf, содержащий программу в кодах машинных инструкций архитектуры.

3.1 Структура ассемблера

Структура генерируемого ассемблера отражает основные требования к подобному инструменту разработки, каковыми являются

- простота внесения изменений и понятность сгенерированного исходного кода ассемблера;
- эффективность разбора ассемблером входной программы и генерации кода.

Исходный код ассемблера генерируется в виде входных файлов для утилит `ocamllex` и `ocamlyacc`[1]; логика, генерирующая код целевой машины, располагается в семантиках правил утилиты `ocamlyacc`. Такая схема позволяет проверить возможность наличия конфликтов в описании синтаксиса языка ассемблера. На рисунке 2 приведен ассемблер, порожденный для описанной в разделе 2 макроархитектуры.

[1] Утилита `ocamlyacc` — генератор табличных синтаксических LALR(1)-анализаторов на языке OCaml, функционально аналогичен широкораспространенной утилите `yacc`; утилита `ocamllex` — функциональный аналог утилиты `lex`.

```
==== scanner.mll: ====
  rule yylex = parse
       | ".." letter letter* ":"
             { Identifier0(get_declared_symbol(lexeme lexbuf)) }
       | "." letter letter*
             { Identifier1(get_used_symbol(lexeme lexbuf)) }
==== ... ====
       | "jmp"  {Token16}
==== ... ====
       | ['0'-'9']+ {Unbound(num_of_string (lexeme lexbuf))}
==== ... ====
==== parser.mly =====
  user_var_decl : Identifier0 {($1,4)};
  user_var_use : Identifier1 {($1)};
==== ... ====
  o0_6_1 : Token8 { state1 := true};
  n0 : o0_6_1 n6 {};
==== ... ====
  n6 : Token14 o6_2 o6_3 Token14 unbound New_line {
       state.(0) <-$1; state.(1) <- $5;
       word_out 16;
       put_out (val_of_string "10") 8;
       put_out state.(0) 4;
       put_out state.(1) 4;
       stop_w(); reset_state() };
==== ... ====
```

Рис. 2: Фрагменты порожденого ассемблера для приведенного примера архитектуры. Показаны: обработка меток, строковых и числовых литералов в лексическом анализаторе; обработка меток в синтаксическом анализаторе, пример порождающих правил для инструкции `swap`.

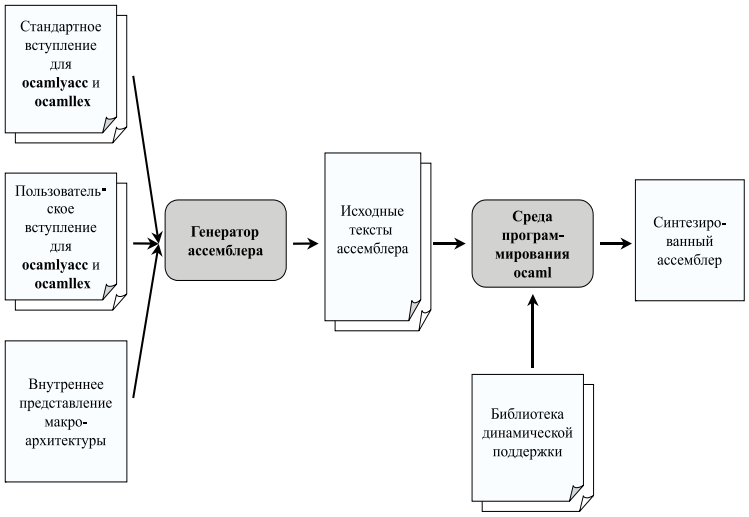

Рис. 3: Схема генерации ассемблера

На рисунке 3 представлена схема генерации ассемблера, показывающая, как интегрируются описание макроархитектуры и расширения. В схеме присутствует поддержка возможности синтаксических расширений. Подобные расширения могут включать в себя как стандартные конструкции, характерные для многих языков ассемблера (переменные, метки и т.д.), так и конструкции, специфичные для данного языка ассемблера.

Стандартное (общее) вступление — это входной файл построителя анализаторов, содержащий описание синтаксиса и семантики стандартных расширений языка ассемблера. Пользовательское (специфичное) вступление — файл, содержащий описание синтаксиса специфичных расширений языка ассемблера. Генератор ассемблера порождает правила, описывающие синтаксис машинных инструкций, и объединяет их с правилами из вступлений. Аналогичная схема используется для генерации входного файла построителя лексических анализаторов, содержащего описание лексического анализатора языка ассемблера.

Приведем пример работы данной схемы для описания меток. Для встраивания меток в ассемблер предусмотрены два нетерминала: `user_label_decl` и `user_label_use`. Правила вывода

этих нетерминалов, позволяющие использовать метки, описываются в пользовательском вступлении для построителя синтаксических анализаторов и могут быть такими:

```
%token Label_name(string)
%token Colon
%token Dot
$$$#1
user_label_decl : Dot Label_name Colon {$ 2}
user_label_use  : Label_name {$ 1}
```

Теперь описывается лексема `Label_name`:

```
letter = [‘a’- ’z’] | [’A’-’Z’] | [’0’-’9’] | ’_’
$$$#2
| ":" {Colon}
| "." {Dot}
| "." letter+ {Label_name (lexeme lexbuf)}
```

Приведенных выше строк достаточно, чтобы использовать метки в программах на языке сгенерированного ассемблера. Если добавить эти строки к тексту ассемблера, сгенерированного для примера на рисунке 1, то можно писать так:

```
..01:
      jmp .01
```

Данный подход имеет явные преимущества перед внесением ручных изменений в сгенерированный исходный код ассемблера. Предлагаемая поддержка стандартных расширений синтаксиса не сложнее, чем внесение подобных расширений во вручную написанный ассемблер.

Основным недостатком предлагаемой структуры является потенцитальный конфликт между лексемами и правилами расширяемого синтаксиса и генерируемыми лексемами и правилами. Например, не рекомендуется использовать в расширениях обычную лексему идентификатора, начинающуюся с символа, так как это может привести к конфликтам с именами инструкций, которые не будут разрешены при генерации ассемблера. Предпочтительным вариантом является снабжение подобных лексем специфическими префиксами, например “.”, как это было сделано в приведенном выше примере. Более правильным решением явился бы анализ правил общего и пользовательского вступления совместно с генерацией правил для разбора синтаксиса инструкций.

3.2 Определение грамматики генерируемого ассемблера

Ниже подробно обсуждается генерация правил для построителя синтаксических анализаторов по описанию макроархитектуры. Структура описания синтаксиса машинных команд, используемая в описании макроархитектуры, непригодна для генерации ассемблера, поскольку может задаваться пользователем из соображений, не имеющих отношения к синтаксису (например, инструкции могут объединяться в группы с общей семантикой).

Таким образом, первым действием генератора ассемблера является преобразование синтаксического и бинарного представления инструкций с их форматами в форму, лишенную заданной пользователем структуры. Полученные описания инструкций задают *язык генерируемого ассемблера* целевого устройства. Этот язык определен над алфавитом, состоящим из обычных символов, объединенных с множеством пронумерованных операндов инструкций:

$$A = ASCII \cup \{Unbound_i\}_{i=1}^{n},$$

где n — максимальное число операндов инструкции в данной архитектуре, $ASCII$ — обычные 8-битовые символы кодировки ASCII, а $Unbound_i$ — металексема i-го операнда инструкции. В случае, когда значение символа i не принципиально, будет использоваться упрощенное обозначение $Unbound$. Явное выделение операндов инструкции позволяет сохранить более полную информацию о ее формате.

Целью дальнейшей работы генератора ассемблера является построение эффективного анализатора, принимающего язык описания машинных инструкций и генерирующего их бинарное представление. Очевидно, что в приведенной постановке задача имеет тривиальное неэффективное решение в предположении, что

$$Unbound \cap ASCII* = \emptyset$$

— входной поток следует поочередно сопоставлять с представлением каждой из инструкций до получения совпадения. На самом деле, описанная выше ситуация не выполняется, так как металексемы $Unbound_i$ реального ассемблера представляют собой так же некоторые множества слов над алфавитом ASCII,

например, они могут быть множествами численных литералов. Далее обсуждается, как перейти к ситуации, когда *Unbound* являются множествами слов над алфавитом ASCII, и построить более эффективный анализатор.

Правила для утилиты `ocamlyacc` будут генерироваться на основе конечного автомата, распознающего язык генерируемого ассемблера. Предварительно для данного языка построено множество лексем, которое заменит ASCII-алфавит и позволит использовать множества слов над ASCII алфавитом в качестве *Unbound*. Отметим, что алгоритм порождает грамматику, не различающую $Unbound_i$ между собой; деятельность, связанная с их различением, располагается в семантиках генерируемых правил.

Ниже приведено описание алгоритма выделения лексем. Главной идеей алгоритма является построение такого набора лексем, который обеспечит однозначное определение следующей лексемы во входном потоке во время лексического анализа однократным просмотром. Это достигается благодаря тому, что для построенного алгоритмом множества лексем выполняется условие, что ни одна из них не является префиксом другой. Если данное требование распространить на слова над ASCII алфавитом, которые могут выступать в качестве *Unbound* (т.е. никакая лексема не является префиксом *Unbound*, и *Unbound* не является префиксом лексемы), то можно построить бесконфликтные (однозначно определяющие текущую лексему) правила для лексического анализатора без возвратов и заглядываний вперед. Это расширенное требование не выражается в простой форме относительно входных данных алгоритма генерации ассемблера. Существуют два метода обеспечения расширенного требования, которые может применить пользователь:

- использовать заведомо уникальные символьные префиксы для описания *Unbound*;
- произвести генерацию с умолчательным видом *Unbound* (в настоящий момент это целые десятичные числа) и затем, исследуя определенные утилитами `ocamlyacc` и `ocamllex` конфликты, установить, какое ограничение на *Unbound* необходимо.

Ниже приведен текст упрощенного варианта (опущена работа

с типами, рассмотрены лишь некоторые элементы внутреннего представления, неоптимизирован процесс разбиения на лексемы) обсуждаемого алгоритма построения множества лексем.

Алгоритм 1. Выделение лексем.

```
splitted : map of lexemes (* История расщепления лексем *)
lex_tab : set of lexemes (* Текущий набор лекем *)

(* Добавление новой лексемы
   (с возможной перестройкой конфликтующих с ней) *)
function add_lexeme (l: lexeme)
  begin
    if l in lex_tab then return [l];
    else if (существует l1 : l - префикс l1) then do
        splitted += (l1,l,постфикс l1);
        lex_tab  -= l1;
        add_lexeme (постфикс l1);
        return add_lexeme l;
    od
    else if (существует l1 : l1 - префикс l) then do
        splitted += (l,l1,постфикс l);
        return l1 + add_lexeme (постфикс l);
    od
    lex_tab += l;
    return [l];
  end

(* Функция, разбивающая на лексемы структурное
   представление инструкции. Возвращает список. *)
function unwrap (text: asm)
begin
  if text is
    Literal(v) (* литерал *)
      do return [add_lexeme Const(v)] od
    Join(t1,t2)(* катенация *)
      do return unwrap(t1) + unwrap(t2) od
    ... (* действия для других элементов описания
           макроархитектуры *)
end

(* Строим набор лексем и записываем при помощи них
   инструкции *)
instrs: array of instruction
Step1:
   forall i in instrs do unwrap i; od
```

```
(* обновить возможно устаревшие после изменений
   списки *)
forall i in instrs do unwrap i; od
```

Параллельно с построением множества лексем перестраивается представление языка инструкций, который после разбиения оказывается записанным при помощи новых лексем.

Итак, данный алгоритм позволяет:

- построить бесконфликтный лексический анализатор при помощи генератора лексических анализаторов `ocamllex` или указать невозможность построения такого анализатора;
- оставить достаточную свободу для выбора вида *Unbound* и сформулировать формальный критерий корректности этого выбора;
- в большинстве случаев уменьшить порождаемый на следующем шаге конечный автомат, распознающий язык ассемблера.

Наконец, следует отметить недостаток алгоритма, который заключается в относительно большой сложности. Можно показать, что сложность алгоритма составляет $O(n^3)$, где n — число лексем, строимых алгоритмом[2].

3.3 Построение синтаксического анализатора языка ассемблера

Язык описания инструкций является конечным языком и может быть разобран конечным автоматом. Существует стандартный алгоритм построения конечного автомата по такому языку — эмуляция разбора конечным автоматом входного потока с разветвлением в ситуациях, когда входной язык допускает не единственный принимаемый остаток входного потока. Для построения конечного автомата, принимающего язык генерируемого ассемблера, используется модификация этого алгоритма.

[2] В текущей реализации, например, выделение лексем для архитектуры PDP-11 на Intel PIII-800 в ОС Linux занимает порядка двух минут, что составляет почти 50% времени работы генератора ассемблера и в 10 раз дольше, чем время генерации оптимизированного дизассемблера.

Алгоритм 2. Построение конечного автомата.

```
(* Выполняет один шаг конечного автомата,
   разветвляясь, если необходимо *)
function step (s: state, prefix: list of tokens,
               chains: list of lists of tokens )
  cur_ts : map of lists of tokens
  is_final : boolean
  cur_inputs : set of tokens

  (* Строит остатки входных цепочек языка,
     остающиеся после очередного шага *)
  function gen_chains (rest: list of tokens)
  begin
    if rest != [] then do
      cur_inputs = {первые лексемы из элементов rest};
      if {вторые лексемы из элементов rest} = [] then do
        if (не помечали state как финальное) then do
            добавить state к финальным состояниям автомата
        od
        is_final := True
      od else do
        cur_ts +=
          (cur, {первые лексемы из элементов rest})
        return
          (current,
            {списки остальных лексем из элементов rest}
          )
      od
    od
    return (current, []);
  end

begin
  is_final := False
  new_chains := gen_chains(chains)

  if |cur_inputs| is
      0 then do
          добавить к автомату состояние state;
        od
      1 then
        if not is_final then do
          step (state,
                prefix + [элемент cur_inputs],
```

```
                new_chains);
          od else do
            добавить к автомату состояние state;

            for_all input in cur_inputs do
              if state - не финальное then do
                new_state: state;
                добавить к автомату
                  переход (state, new_state, prefix, input);

                step (
                      new_state,
                      [],
                      trans.filter_inputs(new_chains,
                                          cur_ts,
                                          new_chains)
                     )
              od
            od
 end

step2:
 step (initial_state,[], input_language)
```

Стоит отметить, что данный алгоритм не выполняет минимизацию полученного конечного автомата. Нами не проводилась практическая оценка эффекта минимизации. К достоинствам минимизации относится уменьшение автомата и, как следствие, уменьшение количества порождаемых правил. К недостаткам следует отнести усложнение семантик — ассемблер должен будет сохранять информацию о текущей инструкции и ее формате. Если же не проводить минимизацию, эта информация определяется текущим правилом, которому соответствует семантика. Упрощенный текст алгоритма порождения правил по конечному автомату поясняет последнее утверждение.

Алгоритм 3. Порождение правил для `ocamlyacc`.

```
(* Генерируем правила для ocamlyacc.
   semantics - фунция, определяющая семантику правила *)
 trans : map of transitions (* карта переходов автомата *)
 finals : set of final_states (* множество финальных состояний *)
 tokens: list of tokens (* список лексем *)
```

```
Step3:
    forall t in trans do
      tokens := t.prefix + t.symbol
      print t.to_state, t.tokens, t.from_state;
      напечатать семантику для t.tokens
    od
    for all f in finals do
      print f, f.prefix;
      (* Так как завершающему правилу соответствует инструкция,
         однозначно определяем семантику *)
      напечатать семантику для бинарного представления состояния f
    od
```

В отличие от стандартного алгоритма построения автомата, добавляются завершающие правила, в семантиках которых производится генерация бинарного представления инструкции с использованием запомненных ранее значений *Unbound*. Значения *Unbound* запоминаются в семантиках как завершающих, так и незавершающих правил.

3.4 Выходные файлы ассемблера

Сгенерированный ассемблер в процессе работы записывает бинарное представление программы в файл формата .elf. Программа записывается в отдельную секцию .elf-файла как последовательность битов, формат которой определяется в соответствии с описанием ресурса памяти. Адреса отсчитываются от начала секции .text в .elf-файле в байтах целевой машины. Формат файла соответствует спецификации TIS ELF версии 1.2.

4 Генерация дизассемблера

Основной функцией дизассемблера является декодирование машинных инструкций и представление их в мнемоническом виде, пригодном для чтения пользователем. Дизассемблер принимает на вход код в виде потока байт, обычно получаемого из объектного модуля или исполняемого файла, и превращает его в текст. Кроме того, весьма полезна возможность распознавания символьной информации — имен переменных и меток, если такая информация была сохранена при ассемблировании исходной

программы. В этом случае адреса в коде, которым соответствуют символические имена, могут быть представлены в результирующем тексте в виде этих имен.

Описание макроархитектуры содержит синтаксис машинных инструкций и их бинарное представление, так что задача генератора дизассемблера сводится к построению программы, которая превращает второе в первое. В настоящей реализации эта программа состоит из драйвера, определяющего алгоритм работы дизассемблера и не зависящего от описания макроархитектуры, и нескольких определений данных, получаемых из описания макроархитектуры. Содержание драйвера и определений будет рассмотрено ниже. На каждом шаге работа дизассемблера может быть представлена как сопоставление некоторой части входного потока с образцом, который является набором битов, и выбор следующего действия в зависимости от результатов сопоставления. Тем самым дизассемблер работает по принципу конечного автомата.

4.1 Получение масок инструкций из описания макроархитектуры

Назовем *маской* длины n отображение

$$m : \{0, ..., n-1\} \to \{0, 1, Undefined\}$$

Маска фиксирует значения некоторых битов от 0 до $n-1$, оставляя другие биты неопределенными. Набор из n битов $b_0...b_{n-1}$ *удовлетворяет маске* m, если

$$\forall i \in \{0, ..., n-1\} \qquad m(i) \neq Undefined \Rightarrow m(i) = b_i$$

Бинарное представление инструкции задает маску, при этом позиции неопределенных битов соответствуют операндам инструкции. Заметим, однако, что неопределенные биты в бинарном представлении инструкций не всегда могут принимать произвольные значения — описание макроархитектуры позволяет накладывать ограничения на операнды инструкций (см. [2]). *Пересечением* масок m_1, m_2 назовем маску m, определяемую как

$$\forall i \in \{0, ..., n-1\} \qquad m(i) = \begin{cases} m_1(i), & m_1(i) = m_2(i) \\ Undefined, & \text{иначе} \end{cases}$$

Генератор дизассемблера работает следующим образом. Вначале выполняется обход внутреннего представления описания макроархитектуры, которое порождается компилятором описаний в процессе синтаксического анализа. Во время обхода собирается бинарное представление инструкций, т. е. маски, и устанавливается соответствие между масками и текстовым представлением инструкций. В этом соответствии содержится информация, достаточная для восстановления операндов инструкции из набора бит, удовлетворяющего маске инструкции.

Среди действий, выполняемых на данном этапе, представляет определенный интерес обработка триммеров. Триммеры, допустимые в кодовых выражениях (см. [2]), позволяют выбирать часть набора битов, ограничиваемую слева и справа либо концами самого набора, либо значениями триммера. Внутри триммера может находиться и операнд инструкции. Во избежание излишнего усложнения и снижения эффективности порождаемого дизассемблера триммеры обрабатываются в генераторе. А именно, все триммерные выражения приводятся к виду

$$identifier\ \ trimmer$$

(здесь используются обозначения из [2]), что достигается за счет раскрытия вложенных триммеров и выделения константных составляющих.

4.2 Общий алгоритм дизассемблирования

Результатом первого этапа является список масок, упорядоченный по возрастанию длин, и соответствие, позволяющее по маске восстановить синтаксическое представление инструкции. Такая информация достаточна для построения декодера инструкций, который и составляет основу порождаемой программы-дизассемблера (драйвера). Предположим, что имеется процедура `syntax (m:mask, b:bitscale)`, возвращающая синтаксическое представление инструкции с маской `m`, которой удовлетворяет набор битов `b`. Схематическое описание драйвера приведено ниже.

Алгоритм 4. Дизассемблирование входного потока.

```
(* Сопоставляет биты b[0]..b[n-1] с заданной маской *)
function try_mask (m: mask, b[0]..b[n-1]:bitscale)
 if b[0]..b[n-1] удовлетворяет m then do  (* (1) *)
   print syntax (m,b[0]..b[n-1])
   return True
 od
 return False
end

(* Пытается дизассемблировать очередную инструкцию из
   входного потока байт *)
function try_disasm_instr (s:input stream)
begin
  n: integer;      (* текущая длина масок *)
  bits: bitscale;  (* первые биты входного потока *)

  forall n in MaskLengths do             (* (2) *)
    прочитать n бит из входного потока s
       в переменную bits;
    forall m in Masks (n) do
      if (try_mask (m,b[0]..b[n-1])) return True;
    od
    вернуть прочитанные биты во входной поток;
  od
  return False
end

(* Пытается дизассемблировать поток байт *)
function disasm (s:insput stream)
begin

  while (not end_of_stream (s)) do
    if (not try_disasm_instr) then do
      сообщить об ошибке дизассемблирования;
      return False
    od
  od
  return True
end
```

Здесь `MaskLengths` — возможные длины масок, а `Masks(n)` — список масок длины `n`. Эти два параметра определяются описанием макроархитектуры.

Некоторые замечания к приведенным процедурам. Во-первых, как упоминалось выше, описание макроархитектуры позволяет задавать ограничения на значения операндов инструкций, например, можно указать интервал, в который должно попадать значение операнда. Некоторое усложнение условия в строке (1) позволяет проводить сопоставление с маской с учетом этих ограничений. Во-вторых, в строке (2) перебор возможных длин масок производится в порядке возрастания. Теоретически возможны ситуации, когда маска длины n_1 является префиксом некоторой другой маски длины $n_2 \geq n_1$ (формально это означает, что имеется последовательность бит $b_0 ... b_{n_2}$, удовлетворяющая обеим маскам). Такие ситуации рассматриваются как некорректные и должны приводить к выдаче генератором сообщения об ошибке в описании макроархитектуры. Наконец, строка (3) означает, что в случае неудачной попытки дизассемблировать начало входного потока (т.е. если не нашлось ни одной маски, которой удовлетворяет начало потока) дизассемблерование останавливается. Вместо этого можно было бы попытаться пропустить некоторое количество бит входного потока и возобновить дизассемблирование с новой позиции. Представляется разумным пропускать количество бит, равное размеру байта (который известен из описания макроархитектуры).

4.3 Оптимизация порождаемого дизассемблера с помощью группировки масок

Таким образом, для построения дизассемблера достаточно уже описанных действий на первом этапе работы генератора. Однако порождаемый дизассемблер неэффективен, поскольку он будет вынужден многократно повторять одни и те же сравнения битов при последовательном переборе масок и сопоставлении входного потока с каждой из них. В любом наборе инструкций многие маски имеют непустые пересечения, и знания о том, что последовательность битов не удовлетворяет этому пересечению, достаточно, чтобы сразу отбросить пересекающиеся маски. Поэтому второй этап работы генератора заключается в группировке пересекающихся масок с тем, чтобы при дизассемблировании можно было эффективно отсекать маски целыми группами.

Далее будут использоваться термины “простая маска” для

обозначения масок инструкций и “групповая маска” для обозначения маски, являющейся пересечением группы масок (в частности, групповая маска может быть пересечением других групповых масок). Группировка масок требует незначительного изменения алгоритма работы дизассемблера. Изменения касаются только процедуры `try_mask`. Заметим, что успешное сопоставление входного потока с пересечением группы масок означает, что может быть продолжен поиск подходящей маски среди масок группы. Таким образом, перебор масок представляет собой поиск по лесу (назовем его “лес масок”), листьями деревьев которого являются простые маски. С учетом этого замечания процедуру `try_mask` можно переписать следующим образом:

```
function try_mask (m: mask, b[0]..b[n-1]:bitscale)
begin
  if m - групповая маска then do
    forall m' in MaskGroup (m) do
      if (try_mask (m', b[0]..b[n-1])) return True;
    od
    return False
  od else do (* m - простая маска *)
    print syntax (m,b[0]..b[n-1]);
    return True
  od
end
```

Здесь `MaskGroup(m)` — это группа масок, пересечением которых является групповая маска `m`.

Группировка масок выполняется с помощью приведенного ниже алгоритма. Алгоритм параметризуется величиной `MinGroupSize`, обозначающей минимальное число масок в группе (и, тем самым, ограничивающей минимальную степень внутреннего узла в дереве леса масок). Его основой является функция `intersect_masks`, которая пытается выделить группу масок и групповую маску из заданного набора масок (одинаковой длины). Функция начинается с оптимистического предположения о том, что все маски заданного набора входят в группу. В процессе работы группа сужается. На каждом шаге рассматривается очередной бит в каждой из масок группы и подсчитывается число масок, имеющих в данной позиции биты ’1’ и ’0’ (`ones` и `zeros`). Если количество масок в одном из наборов `ones` и `zeros` превосходит их количество в другом на-

боре (и не меньше минимального размера группы), то в группе оставляются только маски этого набора и значение групповой маски полагается равным соответствующему значению бита. В противном случае значение групповой маски на данном бите остается неопределенным.

Алгоритм 5. Группировка масок.

```
(* Возвращает новую маску и два списка, в сумме составляющих
   исходный: список масок исходного списка, для которых новая
   маска является пересечением, и остальные маски исходного
   списка
 *)
function intersect_masks (n:integer, masks: list of mask)
  active: list of mask
  group_mask: mask

  (* active - рабочий список тех масок, которые еще остаются
     в группе *)
  active := masks;

  for i:=0 to n-1 do   (* перебираем все биты *)
    ones,zeros: list of masks

    if (|active| < MinGroupSize) goto fail;

    ones  := { m из active : m (i) = 1 };
    zeros := { m из active : m (i) = 0 };

    if (|ones| > |zeros| and |ones| >= MinGroupSize)
    then do
      group_mask (i) := 1;
      active := ones
    od else if (|zeros| >= MinGroupSize) do
      group_mask (i) := 0;
      active := zeros
    od else do
      group_mask (i) := Undefined
    od
  od

  if (|active| >= MinGroupSize)
    return (group_mask,active,masks - active);
fail:
  return (group_mask, [], masks)
```

```
end

(* Выполняет группировку масок фиксированной длины n *)
function group_masks (n: integer)
  worklist, newlist: list of mask
  changed: boolean

  worklist := Masks (n);
  changed := True;

  while (changed and |worklist| >= MinGroupSize) do
    group_mask: mask
    grouped, rest: list of mask

    (group_mask, grouped, rest) :=
       intersect_masks (worklist);

    newlist += group_mask;
    MaskGroup (group_mask) := grouped;
    if worklist != rest then do
      worklist := rest;
      changed := True
    od else do
      changed := False
    od
  od
  return newlist
end

(* Оптимизирует маски *)
function transform_masks ()
  forall n in MaskLengths do
    Masks (n) := group_masks (n)
  od
end
```

Заметим, что высота всех деревьев в лесу масок не превышает 1, т.е. эти деревья состоят из корня — групповой маски и листьев — простых масок. В качестве альтернативы можно было бы продолжать группировку масок, получая деревья большей высоты. Вопрос об автоматическом определении оптимальной высоты деревьев в лесу масок и выбор параметра `MinGroupSize` для данного описания макроархитектуры является пока открытым. Для архитектуры IBM-360, использовавшейся нами в качестве примера, параметр `MinGroupSize` был подобран из эмпирических

соображений с помощью сравнения времен работы нескольких вариантов сгенерированного дизассемблера на достаточно большой программе.

4.4 Оценка сложности генератора и порождаемого дизассемблера

Оценим сложность декодирования инструкции. В худшем случае сложность операции сопоставления маски с набором битов в терминах операций сравнения битов равна количеству n битов в наборе. Сопоставление набора битов с простыми масками, объединенными групповой маской, обходится в худшем случае в g (где g — средний размер группы масок) операций сопоставлений с маской, если набор битов удовлетворяет групповой маске (в противном случае сопоставление с простыми масками не требуется). Линейный поиск групповой маски, которой удовлетворяет данный набор битов, в худшем случае занимает M операций сопоставления маски, если M — количество групповых масок. Таким образом, в худшем случае сложность декодирования инструкции составляет $(M + g) * n$ операций сравнения битов.

Рассмотрим вопрос о сложности алгоритма генерации дизассемблера. Интерес представляет сложность процедуры группировки масок `transform_masks`. Пусть L_n — количество масок длины n, μ =`MinGroupSize`. В худшем случае список `active` в процедуре `intersect_masks` остается постоянным (т.е. не уменьшается) на каждой итерации цикла по номерам битов. При этом общее количество проверок битов составляет $L_n * n$. Количество вызовов `intersect_masks` из `group_masks`, очевидно, не превосходит L_n/μ. Отсюда, общее количество проверок битов, выполнямых в процессе работы процедуры `group_masks`, не превосходит $(L_n^2 * n)/\mu$. Тем самым, сложность процедуры `transform_masks` в терминах количества проверок битов не превосходит

$$(\sum_{n-\text{длина маски}} L_n^2 * n)/\mu.$$

Заключение

В данной статье были рассмотрены алгоритмы генерации ассемблера и дизассемблера по описанию макроархитектуры, ре-

ализованные в наборе программных средств в рамках проекта PADLA. Среди дальнейших направлений развития генераторов ассемблера и дизассемблера можно отметить:

- поддержку восстановления после ошибок в ассемблере и дизассемблере. В особенности это касается дизассемблера, поскольку при дизассемблировании программы, написанной на языке ассемблера, где код может перемежаться с данными, ошибки являются обычным делом;
- повышение эффективности генератора ассемблера;
- улучшение диагностики неоднозначностей в бинарном представлении инструкций из описания макроархитектуры в генераторе дизассемблера.

В настоящее время мы работаем над этими проблемами. Кроме того, ведется разработка генераторов других утилит, необходимых для программирования в машинных кодах — загрузчика и компоновщика объектных модулей.

Список литературы

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

[2] Булычев Д.Ю. Язык описания макроархитектуры для технологии совместной программно-аппаратной разработки // Наст. сборник. — С. 23-48.

[3] Aditya S., Mahlke S., Rau R. Code Size Minimization and Retargetable Assembly for Custom EPIC and VLIW Instruction Formats // ACM Transactions on Design Automation of Electronic Systems. — 2000. — P. 752-773.

[4] Elsner D., Fenlason J. Using as: the GNU assembler. — Free Software Foundation, 1993.

[5] Fauth A., van Praet J., Freericks M. Describing Instruction Set Processors Using nML // Proceedings of European Design and Test Conference. — 1995. — P. 503-507.

[6] Hartoog M. e.a. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign // Design and Automation Conference. — 1997. — P. 303-306.

[7] Jain N.C. Disassembler using High Level Processor Models: Master Thesis. — Kanpur: 1999.

[8] Kumari S. Generation of Assemblers using High Level Processor Models: Master Thesis. — Kanpur: 2000.

[9] Rajesh V. A Generic Approach To Performance Modeling and its Aplication to Simulator Generator: Master Thesis. — Kanpur: 1998.

[10] Ramsey N., Fernandez M.L. The New Jersey Machine-Code Toolkit // USENIX Technical Conference. — 1995. — P. 289-302.

[11] Quin W. A Survey of Architecture Description Languages. http://campuscgi.princeton.edu/~znhuang/mescal/ppt/27.pdf.