Автоматизированная генерация декодеров машинных команд

Н.Ю. Фокина <nfokina@ispras.ru>
М.А. Соловьев <icee@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В работе предложен метод автоматизированной генерации декодеров машинных команд широкого класса процессорных архитектур с использованием транслятора языка ассемблера целевой архитектуры. Реализована программная система, использующая предложенный метод для генерации декодеров машинных команд различных архитектур. Система была протестирована на нескольких микроконтроллерах: PIC16F877A, AVR, Tricore, H8/300H.

Ключевые слова: декодер; микроконтроллер; бинарный код; автоматизированная генерация; система команд.

DOI: 10.15514/ISPRAS-2018-30(2)-4

Для цитирования: Фокина Н.Ю., Соловьев М.А. Автоматизированная генерация декодеров машинных команд. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 65-80. DOI: 10.15514/ISPRAS-2018-30(2)-4

1. Введение

Существует большое число различных архитектур процессоров. Большая часть из них приходится на микроконтроллеры. Для разработки машинно-зависимого системного программного обеспечения (компилятор, отладчик, компоновщик и т.д.), поддерживающего данные архитектуры, необходимо наличие декодера машинных команд данного процессора. Декодером называют инструмент, восстанавливающий текстовое представление машинной команды. Иногда декодер также извлекает другие свойства, например, зависимости по данным внутри инструкции, что необходимо для более глубокого анализа бинарного кода. В отличие от дизассемблера, производящего анализ всего исполняемого файла, декодер обрабатывает отдельные инструкции.

Таким образом, декодирование является подзадачей при дизассемблировании. Также декодер применяется как составная часть других инструментов: средств анализа бинарного кода, мониторов виртуальных машин, систем динамической двоичной трансляции и т.д.

Для подавляющего большинства архитектур существующие инструменты позволяют восстановить лишь текстовое представление инструкций. Это существенно усложняет разработку системного программного обеспечения, поскольку требуется добавлять поддержку каждой архитектуры отдельно. Необходимо обеспечить единообразный интерфейс для декодирования инструкций различных архитектур. Под единообразностью понимается возможность получения информации о мнемонике, числе и типе операндов инструкции, а также о каждом операнде (для операндов-регистров – имя регистра, для операндов-констант и смещений в памяти – соответствующее значение).

В некоторых случаях может отсутствовать, быть неполной или устаревшей документация, описывающая архитектуру системы команд (Instruction Set Architecture, ISA) процессора. При этом в набор системных программ для большинства архитектур входит ассемблер. С его помощью можно установить соответствие между инструкциями и задаваемыми ими кодировками. Эту информацию можно использовать при разработке декодера.

Поскольку написание декодера вручную является ресурсоемким, и полученный в результате инструмент может содержать ошибки, целесообразно генерировать декодер автоматически. Довольно широкое распространение получили специализированные языки описания системы команд, первым из которых был разработанный в 1997 году SLED/NJMC [1,2]. Их использование позволяет избежать написания системных программ (в частности, декодеров) вручную.

Однако можно добиться большей автоматизации, извлекая информацию о архитектуре системы команд из существующего транслятора языка ассемблера. Тогда для генерации декодера используется не низкоуровневое описание кодировок команд, а лишь описание синтаксиса языка ассемблера. Это позволяет существенно упростить работу аналитика и ускорить разработку декодеров. Это особенно актуально при разработке большого числа однотипных инструментов, поскольку при таком подходе аналитик может приступать к описанию очередной архитектуры, в то время как генератор автоматически извлекает информацию о кодировках команд.

В данной статье предложен метод, позволяющий автоматически на основе запусков транслятора языка ассемблера генерировать декодеры команд процессоров. Архитектура системы команд целевого процессора должна удовлетворять следующим требованиям:

- битовые поля, кодирующие операнды, не пересекаются (т.е. каждому операнду соответствует некоторое битовое поле);
- операнды-константы кодируются в дополнительном коде;
- значения констант лежат на непрерывном интервале и кратны некоторой степени 2 (например, последовательность .. -2, 0, 2, 4 ..

удовлетворяет данному требованию, а .. -4, 0, 2, 4 .. или 0, 1, 3, 5 .. – нет).

Разработанный метод лег в основу программной системы, описание которой также приводится в настоящей статье.

Дальнейшее изложение построено следующим образом. В разд. 2 приводится обзор работ, смежных с данной. В разд. 3 дается общее описание разработанной системы. В разд. 4 и 5 более подробно рассматриваются генерация описания кодировок команд и декодирование соответственно. В разд. 6 приводятся результаты тестирования разработанной системы. Разд. 7 содержит заключение.

2. Обзор родственных работ

Решаемая задача является родственной по отношению к двум достаточно независимых группам работ. В первую группу входят работы по созданию инструментов, позволяющих восстанавливать соответствие между ассемблерными инструкциями и их кодировками; во вторую – разработка адаптируемых декодеров, использующих машинно-независимые алгоритмы для декодирования команд различных процессорных архитектур.

К первой группе относится инструмент Derive [3], который позволяет автоматически восстанавливать кодировки ассемблерных инструкций, а также генерировать описания данных инструкций. Работа инструмента основана на использовании ассемблера целевой архитектуры. Пользователем задается описание целевой архитектуры: перечень регистров, перечень мнемоник, описание синтаксиса языка ассемблера. Путем многократного запуска и анализа результатов работы ассемблера инструмент определяет, какие биты кодировки соответствуют мнемонике и операндам инструкции. Если отображение задается достаточно сложным образом, инструмент приостанавливает работу и запрашивает описание инструкции у пользователя.

Несколько другой подход был предложен в серии работ К. Коллберга [4, 5], посвященной разработке адаптируемого (self-retargetable) компилятора. Путем компиляции небольших программ, написанных на языке C, с помощью системных компилятора и ассемблера целевого процессора строится описание системы команд данной архитектуры, необходимое для машинно-зависимой генерации кода. Однако для многих микроконтроллеров компилятор C отсутствует, а некоторые не могут эмулировать абстрактную C-машину в силу ограниченности аппаратных ресурсов.

В ИСП РАН была разработана система MetaDSP [6], предназначенная для разработки кросс-инструментов для микроконтроллеров, включающих в себя ассемблер, дизассемблер, симулятор и профилировщик. Система ориентирована на интерактивную работу и имеет мощный графический интерфейс. Схема работы MetaDSP заключается в следующем: пользователь составляет файл описания архитектуры, который транслируется в файлы

спецификации на языке ISE. На основе этих описаний генерируются требуемые инструменты.

Авторами статьи [7] был разработан язык описания архитектуры системы команд Rosetta. Язык основан на регулярных выражениях, позволяющих с помощью одного описания задать группу инструкций, имеющих сходный тип битовых полей. Перед генерацией декодера производится предобработка, и каждое из описаний транслируется в группу задаваемых им кодировок. В процессе предобработки в заданные описания подставляются все допустимые значения параметров. Из полученного набора кодировок строится дерево. Кодировки в дереве группируются при наличии в них одинаковых битовых последовательностей; для каждой группы в дерево добавляется вершина. Таким образом, в листьях дерева хранятся конкретные кодировки. Полученное дерево сжимается, т.е. в нем выделяются общие поддеревья, и на его основе генерируется декодер (в виде C-кода).

Более сложные варианты деревьев разбора представлены в [8] и [9]. Построение дерева декодирования, описанное в работе [8], основывается на градиентном алгоритме, т.е. в каждом новом поколении ветвление производится по максимальному различающемуся числу значащих битов. В [9] развиваются идеи, предложенные в предыдущей работе: при построении дерева разбора учитываются не только значащие, но и незначащие биты. Последние определяют вероятность обнаружения задаваемой ими инструкции и, таким образом, определяют положение соответствующей вершины в дереве разбора.

Проведенный обзор работ позволяет сделать вывод о необходимости реализации нового инструмента, в котором бы отсутствовали недостатки рассмотренных работ. В частности, почти все рассмотренные инструменты требуют высокой квалификации от оператора. Форматы файлов спецификации архитектуры не только сложны для написания, но и предполагают наличие у оператора глубоких знаний специфики описываемой архитектуры. Интерактивность систем облегчает взаимодействие с пользователем, но, вместе с тем, принуждает его к постоянному контролю за системой.

Кроме того, все рассмотренные инструменты (за исключением MetaDSP) ориентированы на применение к крупным, широко распространенным архитектурам (Intel x86, ARM, SPARC, MIPS) и не учитывают специфики архитектуры большинства микроконтроллеров: простые способы кодирования команд (отсутствие нетривиальных преобразований операндов при кодировании), небольшое количество операндов в каждой из инструкций, и, зачастую, ограниченные наборы команд и регистров.

3. Архитектура системы

Для реализации системы требуется решение двух основные подзадач:

1) генерация описания кодировок команд целевой архитектуры;
2) машинно-независимое декодирование.

При решении первой подзадачи каждой ассемблерной инструкции целевой архитектуры ставится в соответствие ее двоичная кодировка, строится некоторое отображение, позволяющее в дальнейшем декодировать команды данного процессора. Вторая подзадача – декодирование поступающих команд с использованием информации, полученной в результате решения первой подзадачи.

Система состоит из трех модулей:

- модуль генерации описания кодировок команд;
- конвертер, осуществляющий преобразование файла описания кодировок команд из двоичного формата в текстовый формат и обратно;
- библиотека декодирования.

Разработанные модули могут работать независимо, однако предполагается следующий сценарий использования системы:

1) генерация описания кодировок команд;
2) анализ и модификация файла описания кодировок команд;
3) декодирование инструкций целевой архитектуры.

Диаграмма потока работ приведена на рис. 1. Подчеркнутыми надписями на схеме обозначены компоненты разработанной программной системы. Числа над стрелками обозначают порядок выполнения; числа в квадратных скобках обозначают необязательные этапы работы.

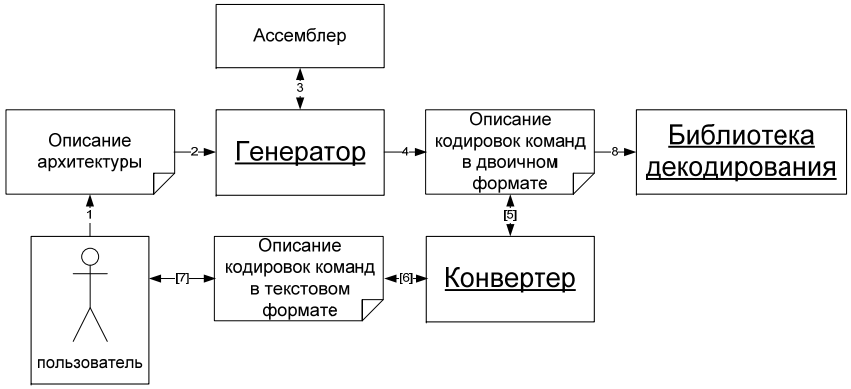

Рис. 1. Диаграмма потока работ
Fig. 1. Workflow diagram

Сначала пользователь на основе документации подготавливает файл описания архитектуры. На основе этого файла производится генерация входных данных для транслятора языка ассемблера. Полученные с его помощью кодировки

обрабатываются, и в результате генерируется файл описания кодировок команд (в двоичном формате). Данный файл можно либо непосредственно использовать для декодирования, либо с помощью конвертера преобразовать в текстовый формат и вручную изменить. Затем полученный текстовый файл нужно преобразовать обратно в двоичный формат и также использовать для декодирования.

В дальнейших разделах приводится развернутое описание разработанных компонентов. При этом используется следующая терминология:

- *команда* – управляющая конструкция целевого процессора;
- *инструкция* – ассемблерное представление корректной команды;
- *кодировка* – двоичное представление команды.

4. Генерация описания кодировок команд

В результате работы генератора по заданному пользователем описанию процессорной архитектуры с использованием ассемблера целевой архитектуры строится описание кодировок команд процессора.

4.1 Файл описания архитектуры

Описание синтаксиса языка ассемблера задается в виде шаблонов инструкций, где в соответствующие позиции вместо переменных подставляются имена регистров и целочисленные константы, а также мнемоники, задающие команду. Группировка шаблонов по синтаксическим признакам позволяет существенно упростить написание входного файла. Более того, можно задавать как неполное (возможно, с ущербом для последующего декодирования), так и избыточное описание архитектуры. Это позволяет не описывать в точности, операнды какого типа необходимы, а перебрать все возможные комбинации типов операндов и выяснить, какие из них используются в инструкции.

Пример описания синтаксиса языка ассемблера представлен на рис. 2. Любая строка, содержащая ключевые слова (*opcode*, *operand*), является шаблоном. Все символы шаблона, не входящие в состав ключевых слов, при генерации входных файлов для транслятора языка подставляются непосредственно. За каждым из шаблонов следует список относящихся к нему мнемоник. Вместо ключевого слова *opcode* генератором будут подставлены все следующие за ним мнемоники, вместо слова *operand* – операнды (регистры из списка регистров и константы). Список имен регистров процессора задается пользователем.

```
1  opcode operand, @(operand, operand)
2  MOV.B MOV.W MOV.L
3  opcode #operand, operand
4  BAND BIAND BIST BIXOR BNOT BOR BSET BTST BXOR CMP.B
5  CMP.W CMP.L DEC.W DEC.L INC.W INC.L LDC.B MOV.B
6  MOV.W MOV.L OR.B OR.W OR.L ORCSUB.W SUB.L SUBS SUBX
7  XOR.B XOR.W XOR.L XOR.C
8  opcode operand, @operand
9  BNOT BSET BTST MOVTPE
10 Opcode
11 EEPMOV RTE RTS SLEEP
```

Рис. 2. Фрагмент описания синтаксиса языка ассемблера (H8/300H)
Fig. 2. Fragment of the assembly language syntax description (H8/300H)

4.2 Алгоритм генерации описания кодировок команд

Основным этапом генерации декодера является генерация описания кодировок команд заданной архитектуры. Для получения этого описания используется ассемблер целевой архитектуры.

Алгоритм генерации описания можно разделить на 4 этапа:

1) разбор входных файлов (описание синтаксиса языка ассемблера, описание набора регистров);
2) генерация возможных перестановок операндов, сохранение необработанных кодировок;
3) обработка: определение опкодов, масок и типов операндов;
4) сохранение результатов в выходной файл.

Обычно операнды инструкции разделяют на три класса:

- константа;
- регистр или прямо указанный элемент памяти;
- элемент памяти, адресуемый косвенно.

Для получения текстового представления инструкции важна лишь синтаксическая запись операнда. Таким образом, целесообразно выделять только два типа операндов: (1) регистры; (2) константы.

Поскольку синтаксически регистры обозначаются именами, операндом-регистром является такой операнд, который может принимать значение из определенного множества текстовых строк. Таким образом, регистрами считаются любые именованные значения (в том числе, именованные константы); константами – целые шестнадцатеричные числа.

Производится перебор всех возможных комбинаций операндов; множество регистров перебирается полностью, множество констант – побитово, т.е. подставляются только такие константы, в которых единичным является только

1 бит (степени 2), а также 0. Данный подход позволяет существенно повысить скорость анализа. Поскольку известно, что все операнды инструкции лежат на непрерывных интервалах, каждый из битов, кодирующих операнд, существенно влияет на его значение. Это позволяет перебирать не все комбинации битов, а только отдельные их позиции, в результате чего сложность перебора сокращается с экспоненциальной до линейной. Ноль также необходим, поскольку при его отсутствии невозможно определить операнды, кодируемые одним битом (например, флаги).

Кодировки одной инструкции, имеющие разную битовую длину, рассматриваются раздельно. Это также позволяет упростить и ускорить анализ, поскольку не требуется в явном виде хранить информацию о значимости каждого бита кодировки и анализировать незначащие биты.

Кодировки сохраняются в таблицу, отдельную для каждой инструкции (см. рис. 3). Каждая новая корректная кодировка добавляется в таблицу, в столбец, соответствующий порядковому номеру операнда, в строку, соответствующую его значению. Ячейки таблицы заполняются парами (опкод; маска). В данном случае опкодом называется часть кодировки, задающая операцию (т.е. код операции, КОП) и операнд, соответствующий данной ячейке.

Если ячейка пуста, кодировка сохраняется в ней; корректными считаются все ее биты, т.е. все биты маски единичные. Если ячейка не пуста, то нужно изменить маску кодировки, т.е. корректными битами должны оставаться только те, которые постоянны для данной ячейки.

После того, как таблица построена, данные из нее используются для определения кода операции и масок операндов.

Определение кода операции производится следующим образом. Вначале производится перебор всех значений каждого операнда. *instr* – некоторый столбец построенной таблицы, т.е. позиция операнда фиксирована.

Инициализация производится при обнаружении первого ненулевого значения:

```
mask = instr[i].mask;
opcode = instr[i].opcode;
```

Для каждого последующего ненулевого значения (`j != i`):

```
mask &= ~((opcode & mask) ^ (instr[j].opcode & instr[j].mask));
opcode &= mask;
```

КОП/маска инструкции определяется как *побитовое и* КОП/масок для каждого из ее операндов. Полученные таким образом КОП и маска не зависят ни от одного из операндов инструкции и определяют закодированную с их помощью команду.

	операнд 1 [reg]	операнд 2 [imm]	
r0	(0xf800; 0xfff8)	(0xf800; 0xfe0f)	0
r1	(0xf810; 0xfff8)	(0xf801; 0xfe0f)	2^0
r2	(0xf820; 0xfff8)	(0xf802; 0xfe0f)	2^1
r3	(0xf830; 0xfff8)	(0xf804; 0xfe0f)	2^2
r4	(0xf840; 0xfff8)	NULL	2^3
r5	(0xf850; 0xfff8)	NULL	2^4
r6	(0xf860; 0xfff8)	NULL	2^5
r7	(0xf870; 0xfff8)	NULL	2^6
r8	(0xf880; 0xfff8)	NULL	2^7
r9	(0xf890; 0xfff8)	NULL	2^8
r10	(0xf8a0; 0xfff8)	NULL	2^9
	.....	.....	

Рис. 3. Пример таблицы кодировок для инструкции `BLD reg, imm` *(AVR)*
Fig. 3. Example of an encoding table for the instruction `BLD reg, imm` *(AVR)*

Определение масок операндов принципиально не отличается от определения опкода. При обнаружении первого ненулевого значения операнда:

```
op_mask = 0;
temp = instr[i].opcode & instr[i].mask;
```

Для всех последующих ненулевых значений (`j != i`):

```
op_mask |= temp ^(instr[j].opcode & instr[j].mask);
temp &= ~op_mask;
```

Поскольку, в соответствии с налагаемыми ограничениями, битовые поля кодировок не пересекаются, маска операнда содержит все биты, кодирующие операнд.

4.3 Особенности трансляторов языка ассемблера

Ассемблеры для некоторых архитектур имеют особенности, некритичные для их использования при трансляции, но влияющие на генерацию файла описания. Одной из таких особенностей является дублирование кодировок. Ассемблер не считает ошибочным задание в качестве операнда слишком большого значения константы, а генерирует кодировку, где данный операнд закодирован как какая-либо меньшая допустимая константа (как правило, 0). Поскольку данная инструкция с меньшей константой также корректна, возникает дублирование, и при декодировании такая кодировка может быть разобрана как правильно, так и неправильно.

Например, для архитектуры PIC16F877A (ассемблер gpasm), некорректная из-за переполнения второго операнда инструкция INCF 0x0, 0x80 кодируется как 0xA0, что на самом деле соответствует инструкции INCF 0x0, 0x0. В таких случаях учитывается только первое (с меньшей константой) вхождение данной кодировки.

Еще одной особенностью является совпадение кодировок регистров. В некоторых случаях синтаксисом языка ассемблера допускается задание имени регистра как константы-номера в регистровом файле. В таком случае, если определенный операнд может задаваться и регистром, и константой, и типы остальных операндов попарно совпадают, необходимо выяснить, не является ли избыточной кодировка, в которой данный операнд является непосредственно заданной константой.

Для обнаружения такой ситуации используется следующий алгоритм: если для инструкции найдутся такие отображения, что для заданного операнда множество допустимых кодировок непосредственно заданных констант является подмножеством допустимых кодировок регистров, и при этом размеры кодировок равны, а также кодировки всех остальных операндов совпадают и по типу, и по значениям, необходимо пометить то отображение, где данный операнд имеет константный тип. После проверки всех отображений, помеченные отображения необходимо удалить как избыточные.

Например, инструкция архитектуры AVR (ассемблер avr-as) ADD $r2, $r3 также может быть задана как ADD 0x2, 0x3. Из-за этого полученный файл описания кодировок команд содержит дублирующиеся инструкции (см. рис. 4). После преобразования будет сгенерирован файл, согласно которому операндами инструкции ADD могут являться только регистры (что соответствует действительности).

Однако, если операнды-регистры всегда кодируются числами (как, например, в архитектуре PIC16F877A), предложенное преобразование может не работать. В инструкциях CALL и GOTO адреса перехода кодируются целыми 11-битными числами, в то время как регистровый файл содержит 512 регистров (т.е. регистр кодируется не более чем 9 битами). Таким образом, из-за того, что число регистров недостаточно, чтобы покрыть все множество значений данного операнда, в любом случае будет сгенерировано два описания каждой из данных инструкций.

```
1  add reg_1, reg_1
2  000011baaaaabbbb
3
4  add reg_1, imm
5  000011baaaaabbbb
6
7  add imm, reg_1
8  000011baaaaabbbb
9
10 add imm, imm
11 000011baaaaabbbb
```

```
1  add reg_1, reg_1
2  000011baaaaabbbb
```

Рис. 4. Фрагмент сгенерированного описания до и после преобразования (AVR)
Fig. 4. Fragment of the generated description before and after the transformation (AVR)

5. Декодирование

Процесс декодирования представляет собой обработку последовательных запросов к библиотеке декодирования. Правила декодирования задаются с помощью файла описания кодировок команд, сгенерированного ранее.

При декодировании инструкции необходимо получить результат следующего вида:

- информацию о корректности кодировки;
- если кодировка корректна, необходимо получить ее ассемблерное представление.

В свою очередь, обработка каждого запроса состоит из следующих этапов:

1) определение мнемоники;
2) определение операндов.

При этом, поскольку несколько инструкций, имеющих одинаковый опкод и маску, могут различаться в зависимости от операндов, при декодировании операндов может понадобиться повторный поиск и декодирование мнемоники. Также частным случаем является ситуация, когда одному опкоду соответствует несколько мнемоник. Это чаще всего происходит, если архитектура системы команд предусматривает наличие более высокоуровневых команд для облегчения разработки прикладных программ.

Например, в архитектуре AVR существует команда CLR (очистить бит в регистре), кодировка которой совпадает с кодировкой команды ANDI (*побитовое и* с константой), поскольку данные команды семантически эквивалентны. При возникновении подобных коллизий выбирается первая обнаруженная подходящая мнемоника.

По опкоду и маске определяется соответствующая им мнемоника. Для этого среди всех известных пар опкод/маска находится такая, которая соответствовала бы переданной кодировке, и длина которой совпадала бы с длиной переданного буфера.

После того, как определена мнемоника, декодируются операнды. Операнды-регистры и операнды-константы обрабатываются по-разному. Для констант выбираются требуемые биты переданной инструкции и сжимаются до непрерывной последовательности (т.е. незначащие биты опкода удаляются). Если же операнд имеет регистровый тип, после определения его индекса описанным выше способом осуществляется поиск в соответствующей таблице, где индексом является полученное число, значением – текстовая строка, содержащая имя регистра.

Ключевым фактором, определяющим скорость декодирования, является структура данных, используемая для хранения описания кодировок команд. Кроме того, на общее время работы декодера влияет, хотя и намного менее существенно, время, требуемое на десериализацию файла описания кодировок команд. Таким образом, подходят следующие структуры данных:

1. **Список.** Звенья представляют собой структуры, содержащие информацию о кодировках команд (опкод, маска, кодировки операндов) и инструкциях (мнемоника и типы операндов). Поиск осуществляется с помощью линейного прохода по всем кодировкам. Достоинствами данной структуры являются простота ее реализации и низкие накладные расходы, затрачиваемые на десериализацию описания инструкций. Тем не менее, из-за линейной временной сложности операции поиска требуемой инструкции, эта структура данных практически применима лишь для архитектур с небольшим набором инструкций (не более 30-50).
2. **Список деревьев**. Для реализации алгоритма поиска инструкции, имеющего менее чем линейную временную сложность, на рассматриваемом множестве должно быть задано отношение порядка. В связи с тем, что поиск осуществляется одновременно по опкоду и маске, требуется, чтобы заданная операция отношения существенно учитывала обе данных зависимости. В общем случае это невозможно. Поскольку количество масок существенно меньше, чем количество опкодов, целесообразно сгруппировать все инструкции с одинаковыми масками в двоичные деревья поиска, обеспечив таким образом логарифмическое время поиска инструкции внутри каждого дерева, т.е. при фиксированной маске. Таким образом, вершины, в которых совпадают маски, объединяются в сбалансированные деревья. Корни деревьев произвольным образом связываются в список. Поиск осуществляется по списку линейно, далее для каждого из деревьев осуществляется бинарный поиск по опкоду. Сложность десериализации при таком подходе увеличивается незначительно – с

$O(1)$ для списка до $O(m * log(k))$, где m – число различных масок, k – число инструкций с данной маской. Такая временная сложность приемлема с учетом того, что данная операция осуществляется однократно, и величины m и k сравнительно невелики.

В реализации поддерживаются обе предложенных структуры данных, но по умолчанию используется список деревьев, что обусловлено его большей эффективностью при больших размерах набора инструкций.

6. Результаты тестирования

Реализованная программная система была протестирована на архитектурах PIC16F877A, AVR, Tricore, H8/300H.

В табл. 1 приведено время генерации декодеров для различных архитектур. Временем генерации считается суммарное время, затраченное оператором на написание входных файлов (описание синтаксиса инструкций, описание набора регистров и набора инструкций), а также время работы непосредственно генератора.

Стоит отметить, что в силу экспоненциальной сложности алгоритма наибольшее влияние на время работы оказывает количество операндов в каждой из инструкций. На практике максимально допустимое число операндов – 3, что делает систему применимой для большого числа современных микроконтроллеров.

Менее существенно на время работы влияет размер набора регистров и размер набора инструкций. Большим размером набора регистров обусловлено, в частности, увеличение времени работы генератора при генерации описания PIC16F877A (512 регистров) по сравнению с AVR (32 регистра). Тем не менее, для всех рассмотренных архитектур время генерации декодера не превышает суток и занимает существенно меньше времени, чем написание соответствующего инструмента вручную.

Табл. 1. Время генерации файла описания кодировок команд
Table 1. Time of generation of an encoding description file

Архитектура	Время работы, чч:мм		
	Пользователь	Генератор	Общее
PIC16F877A	00:30	02:18	02:48
AVR	00:30	00:25	00:55
Tricore	03:00	18:39	21:39
H8/300H	01:30	03:14	04:44

Табл. 2 и рис. 5 отражают скорость работы полученного декодера по сравнению с существующими инструментами для соответствующих архитектур.

Разработанный инструмент позволяет получать декодеры, превосходящие написанные вручную по скорости в несколько раз.

Табл. 2. Сравнительная характеристика декодеров
Table 2. Comparative characteristics of decoders

Архитектура	Скорость декодирования, инстр/с		Соотношение скоростей
	Разработанный инструмент	Эталонный декодер	
PIC16F877A	26 222.79	3 886.15 (gpdasm)	6.75
AVR	1 762.71	865.22 (avr-objdump)	2.04
Tricore	753.99	368.78 (objdump 2.13)	2.07
H8/300H	1 845.34	496.06 (h8300-hms-objdump)	3.72

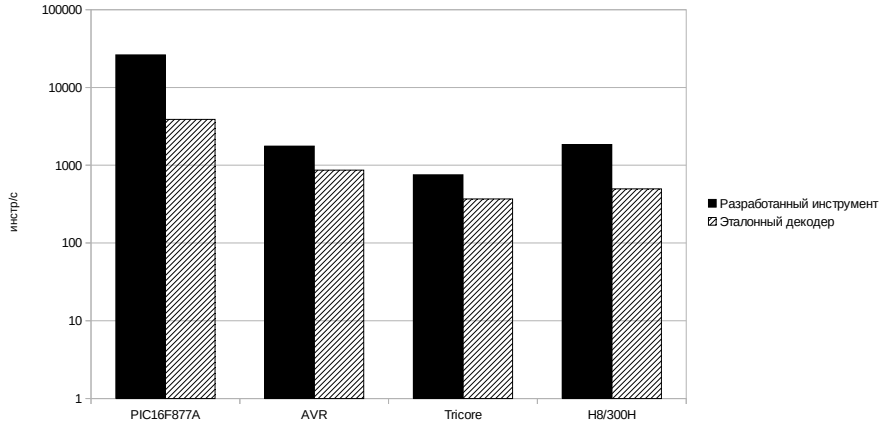

Рис. 5. Сравнительная характеристика декодеров
Fig. 5. Comparative characteristics of decoders

Тестирование производилось на машине Intel Xeon E3-1240 v2 3.40 ГГц с 8 Гб ОЗУ, платформа Ubuntu Linux 14.04 x86_64.

7. Заключение

В настоящей работе предложен метод автоматизированной генерации декодеров машинных команд; метод был реализован в виде программной системы, включающей в себя генератор описания кодировок команд, библиотеку декодирования, а также конвертер, позволяющий преобразовывать полученное описание кодировок команд из двоичного формата в текстовый и обратно, что делает систему более гибкой.

Разработанные инструменты позволяют существенно упростить поддержку большого числа различных процессорных архитектур, поскольку предложенный формат файлов не требует высокой квалификации оператора. Вместе с тем, автоматизированная генерация декодеров производится существенно быстрее, чем при ручном или полуавтоматическом (описание кодировок команд на некотором языке вручную) написании соответствующего инструмента.

Программная система была протестирована на нескольких целевых архитектурах. Скорость декодирования полученных библиотек во всех случаях выше, чем у стандартных инструментов, находящихся в открытом доступе.

Список литературы

[1]. Ramsey N., Fernandez M.F. The New Jersey Machine-code Toolkit. Proceedings of the USENIX Technical Conference, 1995. pp. 289-302.

[2]. Ramsey N., Fernandez M.F. Specifying Representations of Machine Instructions. ACM Transactions on Programming Languages and Systems, 19(3), 1997. pp. 492-524.

[3]. Hsieh W.C., Engler D.R., Back G. Reverse-Engineering Instruction Encodings. Proceedings of the General Track: 2002 USENIX Annual Technical Conference, 2001. pp. 133-145.

[4]. Collberg C.S. Reverse Interpretation + Mutation Analysis = Automatic Retargeting. Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, 1997. pp. 57-70. DOI: 10.1145/258916.258922.

[5]. Collberg C.S. Automatic Derivation of Compiler Machine Descriptions. ACM Transactions on Programming Languages and Systems, 24(4), 2002. pp. 369-408. DOI: 10.1145/567097.567100.

[6]. Рубанов В.В., Михеев А.С. Интегрированная среда описания системы команд встраиваемых процессоров. Труды ИСП РАН, том 9, 2006 г., стр. 143-158.

[7]. Krishna R., Austin T. Efficient Software Decoder Design. IEEE Computer Society Technical Committee on Computer Architecture Newsletter, 2001.

[8]. Theiling H. Generating Decision Trees for Decoding Binaries. Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, 2001. pp. 112-120. DOI: 10.1145/384197.384213.

[9]. Qin W., Malik S. Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits. Proceedings of the 40th Annual Design Automation Conference, 2003. pp. 764-769. DOI: 10.1109/DAC.2003.1219122.

Automated generation of machine instruction decoders

N.Yu. Fokina <nfokina@ispras.ru>
M.A. Solovev <icee@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. This paper proposes a method of automated generation of machine instruction decoders for various processor architectures, mainly microcontrollers. Only minimal, high-level input from user is required: a set of assembly instruction templates and a list of register names. The method utilises the target architecture assembler to reveal the mapping of assembly-level instructions onto their binary encodings by mutating variables in the templates. The recovered mapping is then used as the central part of the architecture-independent decoder. The developed tools allow to significantly simplify the support of a large number of different processor architectures, since the proposed file format does not require high skill of the operator. At the same time, automated generation of decoders is performed much faster than manual or semi-automatic (description of the command character encodings in a certain language manually) development of a corresponding tool. A system based on the proposed method has been implemented and tested over a set of four microcontroller architectures: PIC16F877A, AVR, Tricore, H8/300H. The speed of decoding of our system is in all cases higher than that of standard tools that are in the public domain

Keywords: decoder; microcontroller; binary code; automated generation; instruction set.

DOI: 10.15514/ISPRAS-2018-30(2)-4

For citation: Fokina N.Yu., Solovev M.A. Automated generation of machine instruction decoders. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue. 2, 2018, pp. 65-80 (in Russian). DOI: 10.15514/ISPRAS-2018-30(2)-4

References

[1]. Ramsey N., Fernandez M.F. The New Jersey Machine-code Toolkit. Proceedings of the USENIX Technical Conference, 1995. pp. 289-302.

[2]. Ramsey N., Fernandez M.F. Specifying Representations of Machine Instructions. ACM Transactions on Programming Languages and Systems, 19(3), 1997. pp. 492-524.

[3]. Hsieh W.C., Engler D.R., Back G. Reverse-Engineering Instruction Encodings. Proceedings of the General Track: 2002 USENIX Annual Technical Conference, 2001. pp. 133-145.

[4]. Collberg C.S. Reverse Interpretation + Mutation Analysis = Automatic Retargeting. Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, 1997. pp. 57-70. DOI: 10.1145/258916.258922.

[5]. Collberg C.S. Automatic Derivation of Compiler Machine Descriptions. ACM Transactions on Programming Languages and Systems, 24(4), 2002. pp. 369-408. DOI: 10.1145/567097.567100.

[6]. Rubanov V.V., Mikheev A.S. Integrated Environment for Embedded Processors Instruction Set Description. Trudy ISP RAN/Proc. ISP RAS, 2006, vol. 9, pp. 143-158 (in Russian).

[7]. Krishna R., Austin T. Efficient Software Decoder Design. IEEE Computer Society Technical Committee on Computer Architecture Newsletter, 2001.

[8]. Theiling H. Generating Decision Trees for Decoding Binaries. Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, 2001. pp. 112-120. DOI: 10.1145/384197.384213.

[9]. Qin W., Malik S. Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits. Proceedings of the 40th Annual Design Automation Conference, 2003. pp. 764-769. DOI: 10.1109/DAC.2003.1219122.