1. Модели процессора на основе интерпретации

Понять — значит построить модель.

Уильям Томсон (лорд Кельвин)

«Интерпретатор» в общем значении слова — тот, кто занимается переводом текста с одного языка на другой. В контексте вычислительной техники этот термин противопоставляется трансляторам и компиляторам; последние два понятия описывают программы, преобразующие тексты на входном языке (машинном или высокого уровня) в новое представление, оперируя при этом достаточно большими его блоками — файлами, модулями, функциями и т.п. Интерпретатор же ограничивается работой над одной «строкой» (например, машинной инструкцией) входного языка. Следующая строка будет преобразована (*проинтерпретирована*) тогда, когда в этом возникнет необходимость.

1.1. Архитектурное состояние

Прежде чем перейти к описанию общей картины алгоритма, рассмотрим, какие структуры данных используются для представления состояния процессора в функциональной модели[1].

В любом классическом процессорном устройстве всегда присутствует регистр, хранящий адрес текущей исполняемой инструкции. Например, в архитектуре IA-32 [**intelmanual2a**] для этого используется семейство xIP: `IP`, `EIP`, `RIP`, в архитектуре ARM [20] он имеет название `pc`, в других системах он может называться по-другому, например, `IC` (*англ.* instruction counter). В дальнейшем для единообразия мы будем использовать обозначение `PC` (*англ.* program counter).

Кроме указателя инструкций, процессоры содержат множество других регистров, типы, назначение и параметры которых зависят от

[1]Более детально об архитектурном состоянии рассказывается в главе 8.

модели. В большинстве случаев присутствуют регистры общего назначения (*англ.* general purpose registers, GPR), используемые в арифметических операциях и при адресации памяти.

В языке Си (и C++) описание состояния может быть представлено структурой `state_t`, содержащей поля для всех регистров, а также ссылки на внешние устройства:

```
typedef uint32_t register_t; // ширина гостевых регистров
const int n_regs = 16; // число регистров
typedef struct {
    register_t pc; // счётчик инструкций
    register_t gpr[n_regs]; // регистры общего назначения
    uint8_t *memory; // указатель на ОЗУ
} state_t;
```

Заметим, что данное описание очень далеко от того, чтобы быть полным, однако оно даёт базовое представление того, с чем приходится иметь дело в начале разработки новой модели.

1.2. Стадии работы

Алгоритм работы в общих чертах напоминает стадии конвейера исполнения команд в настоящем процессоре[1] (рис. 2.1).

1. Извлечение (*англ.* fetch) кода инструкции из памяти по адресу, вычисляемому из значения `PC`. Конкретная формула зависит от деталей архитектуры и текущего режима процессора.
   В модели это действие идентично операции чтения из памяти и может вызывать соответствующие побочные эффекты.
2. Задача декодирования (*англ.* decode) состоит в том, чтобы по числу, полученному в предыдущей фазе, определить, какую операцию следует выполнить и какие аргументы в ней будут участвовать. Например, число `0x706a` в архитектуре IA-32 обозначает команду `PUSH 0x70` — поместить в стек число 0x70.
   Алгоритм и сложность декодирования сильно зависят от сложности самого языка инструкций целевой машины. Как правило,

[1] Отметим, что число стадий может быть различно у разных моделей и варьируется от трёх до двадцати и более.

Рис. 1.1. Рабочий цикл интерпретатора

процесс состоит из поиска и сопоставления битовых полей считанного машинного слова со значениями из заранее созданных таблиц. В силу многих факторов (например, переменной длины инструкций, различного смысла значений в различных режимах процессора, использования префиксов и т.п.) оно может занимать существенную часть времени работы интерпретатора. Мы рассмотрим декодирование подробнее в секции 2.4.

3. Исполнение (*англ.* execute) состоит из непосредственной симуляции функции только что декодированной инструкции. Как правило, это вычисление результата арифметической или логической операции, изменение режима модели процессора или передача контроля управления в другую секцию алгоритма.
   В модели каждому коду машинной операции (*опкоду*) должна соответствовать моделирующая процедура. Выбор нужной процедуры производится по опкоду:

```
switch (opcode) {
    case OPERATION1: ...
    case OPERATION2: ...
```

```
    ...
    default: ... // unknown command
}
```

4. Запись результата (*англ.* write back) операции в архитектурные регистры. Часть результатов также может быть расположена в оперативной памяти. Как и при её чтении (на этапе извлечения кода инструкции или получения входных операндов), при записи модель должна симулировать все побочные эффекты.
5. Продвижение указателя команд (*англ.* advance PC) на значение, соответствующее следующей инструкции. Т.к. большая часть существующих алгоритмов состоит из линейных участков, изредка прерываемых операциями ветвления, к нему прибавляется длина только что завершённой машинной команды.
   При моделировании необходимо учитывать ограниченность ширины регистра PC и возможность его переполнения.

```
const int instr_size = 2; // 16 bit CPU
const int addr_mask = 0xffff; // mask overflowed bits
state_t processor; // our CPU
...
processor.pc = (processor.pc + instr_size) & addr_mask;
```

1.3. Исключения и прерывания

Часто при обработке текущей инструкции возникает ситуация, когда нормальное её выполнение не может быть завершено, потому что были обнаружены недопустимые условия на входные операнды (например, целочисленное деление на ноль или недоступность памяти), или возникло какое-то внешнее условие, требующее немедленной обработки. При этом архитектурное состояние процессора изменяется определённым способом (как правило, управление передаётся на обработчик возникшей ситуации), в том числе регистр PC начинает указывать на новый участок кода.

На рис. 2.2 изображён цикл интерпретации, учитывающий тот факт, что практически в любой момент может произойти переход в состояние обработки исключительной ситуации, вносящее изменения

в архитектурное состояние, после чего цикл интерпретации начинается заново уже с новым PC.

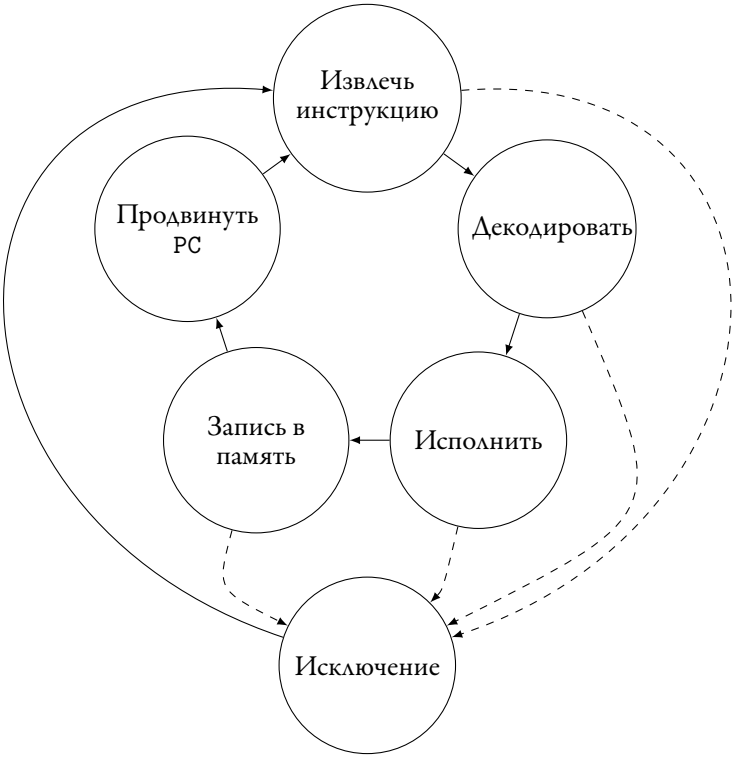

Рис. 1.2. Рабочий цикл интерпретатора. Показана возможность возникновения исключительной ситуации на любой стадии симуляции

1.3.1. Классификация

В документациях к различным процессорам [9, 20, 26] даны различные, зачастую внутренне противоречивые определения терминов, связанных с исключительными ситуациями. Тем не менее важно различать природу событий, их связь с текущим контекстом выполнения для того, чтобы корректно симулировать их эффекты.

Выделим основные группы исключительных событий по признакам наличия причинной связи между событиями, влиянием среды исполнения на возможность их возникновения, а также адресом возврата после их обработки. В скобках к терминам будут даны те имена, под какими они чаще всего встречаются в литературе; однако не следует полагаться на строгость данных соответствий. См. также замечания ниже.

– *Синхронные с повторением текущей инструкции* (промах, *англ.* fault) — событие, связанное причинно-следственно с выполнением текущей инструкции и обусловленное «неготовностью» среды исполнения к её успешному завершению. Примеры таких ситуаций: отсутствие физической страницы памяти с необходимыми данными; неготовность сопроцессора выполнять работу, т.к. он требует дополнительной инициализации. В этих случаях обработчик ситуации, находящийся в операционной системе, может модифицировать среду исполнения так, что завершение инструкции станет возможным, например, загрузить нужную страницу или включить сопроцессор. Дальнейшее возвращение на *тот же* PC с перезапуском инструкции позволит устранить проблему прозрачно для пользовательского приложения.
– *Синхронные без повторения текущей инструкции* (исключения, *англ.* exception). Как и в предыдущем случае, событие порождено текущей инструкцией. Однако его обработка не подразумевает её повтора, так как причина события неустранима и не связана со средой исполнения, но связана только с самой операцией и операндами. Примеры: инструкция целочисленного деления на регистр, содержащий ноль, всегда будет давать ошибку; инструкция, запрещённая к выполнению в текущем уровне привилегий, не может быть на нём исполнена никогда. Чаще всего (но не всегда!) исключения обозначают ошибку в программе. Управление после возврата из обработчика будет передано в место, не связанное с PC того кода, где произошло событие.
– *Ловушки* (*англ.* trap) также синхронны. При этом они обозначают явное «желание» программы быть прерванной и передать управление в определённую область кода — обработчик вызова.

Примером является инструкция SYSCALL, вызывающая системные функции операционной системы, такие как работа с файлами, создание новых процессов и т.п. Другой пример — команда, предназначенная устройству-сопроцессору, физически отсутствующему в системе, однако ОС умеет её эмулировать и таким образом способна вернуть правильный результат прозрачно для пользовательской задачи. С точки зрения прикладного ПО ловушка — это инструкция, семантика которой определяется не спецификацией ЦПУ, а используемой операционной системой и средой исполнения.

После обработки ловушки и возврата счётчик инструкций будет указывать на следующую команду в потоке исполнения, т.е. будет соответствовать нормальному потоку. Отметим, что разница между ловушками и промахами минимальна и их классификация зависит от верхнеуровневого смысла, который вкладывают в соответствующую подпрограмму-обработчик.

– *Асинхронные прерывания* (*англ.* interrupt). В отличие от всех ранее рассмотренных событий, они вызваны причинами, внешними по отношению к текущему контексту исполнения, и означают некоторое состояние внешней среды, требующее внеочередной обработки. Примеры: жёсткий диск готов передать новую порцию данных, ранее запрошенную независимым процессом; температурный датчик сигнализирует о превышении измеряемой температуры порогового значения; таймер сообщил о прошествии запрограммированного в него интервала. Прерывание никак не связано с опкодом, адресом или аргументами инструкций — оно могло произойти чуть раньше или позже, а могло и вовсе не произойти. Однако игнорировать его в общем случае нельзя. Часто недопустимо откладывать вызов обработчика во времени дольше, чем на некоторый краткий период времени.

  Будет ли после возвращения из обработчика перезапущена инструкция, на которой возникло прерывание, зависит от конкретной архитектуры процессора, однако в любом случае её исполнение должно пройти таким образом, чтобы сам факт обработки был скрыт от прерванного приложения.

Следует также отметить следующие особенности существующих центральных процессоров.

1. Программное прерывание (*англ.* software interrupt) — событие, вызываемое специальной инструкцией (например, в IA-32 это `INT`), обработка которого напоминает вызов процедуры. Т.е., несмотря на название, оно соответствует ловушке, а не прерыванию.
2. В некоторых архитектурах, например SPARC [26], подпрограмма-обработчик синхронного события может сама выбрать, следует ли перезапускать текущую инструкцию. Для возвращения из подпрограммы-обработчика могут использоваться две различные инструкции — `RETRY` для перезапуска (в случае обработки промаха) и `DONE` для исполнения следующей команды за текущей (для выхода из обработки ловушек). Для поддержки такой возможности в архитектуру введён регистр `nPC`, в любой момент указывающий на следующую за текущей инструкцию.

1.3.2. Обработка исключительных ситуаций при симуляции

Существование исключений и прерываний (а они отсутствуют разве что только в узкоспециализированных микроконтроллерах) существенно усложняет логику как аппаратной системы, так и моделирующей среды. Непредсказуемость их возникновения создаёт множество веток исполнения в структурированном коде, усложняя его структуру, а также негативно влияя на скорость исполнения.

Обычный структурированный код процедурных и объектно-ориентированных языков высокого уровня состоит из вложенных вызовов процедур (методов), каждая из которых по окончании работы возвращает управление в вызвавшую процедуру по адресу, сохранённому на стеке. Однако моделирование исключительных ситуаций подразумевает возможность их возникновения в множестве мест — индивидуальных блоках эмуляции инструкций. При этом после изменения архитектурного состояния управление должно быть передано на начало следующего цикла интерпретации.

Забегая вперёд, заметим, что особенно остро эта проблема передачи управления встаёт не в интерпретаторах, а в двоичных трансляторах, часть кода которых создаётся динамически. При этом часто при исполнении этого кода заранее нельзя сказать, какова будет структура стека в момент обнаружения исключительной ситуации, и его развёртывание до необходимого уровня вложенности с помощью серии обычных возвратов из процедур будет достаточно дорогостоящей операцией, нивелирующей преимущества быстрого исполнения.

Естественным способом передачи управления в такой ситуации является нелокальный «прыжок» — переход, использующий пару функций `setjmp()` и `longjump()`, описанных в стандарте библиотеки Си.

Функция `setjmp` сохраняет контекст в переменной `env` и возвращает 0, если выход из неё был после её прямого вызова. Если произошёл возврат из `longjmp`, то функция возвращает ненулевое значение.

Функция `longjmp` возвращает выполнение в точку вызова `setjmp` со значением `val`. При этом все объекты с неавтоматическим выделением памяти сохраняют своё значение.

Пример использования `setjmp()` и `longjmp()`[1]:

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;
void second(void) {
  printf(''second\n'');/* печать на экран */
  longjmp(buf,1);/* переходит по метке buf и возвращает код 1*/
}

void first(void) {
  second();
  printf(''first\n'');/* этой печати не произойдёт*/
}

int main() {
  if ( ! setjmp(buf) ) {
    first(); /* при исполнении вернёт код 0*/
  } else { /* по возвращении из longjmp вернёт 1*/
```

[1]Пример взят из Википедии: `http://en.wikipedia.org/wiki/Setjmp.h`.

```
    printf(''main\n'');/* печать на экран*/
  }
  return 0;
}
```

Нельзя отрицать, что использование как нелокальных[1] переходов с помощью `longjump`, так и локальных[1] переходов по метке с помощью оператора `goto` языка Си нарушает модульность кода и лёгкость его чтения, а также может быть источником алгоритмических ошибок. Однако на это приходится идти ради увеличения скорости работы приложения.

1.4. Реализация декодера

Теория вопросов разбора и лексического анализа выражений хорошо разработана для языков высокого уровня и описывается во всех книгах, посвящённых задаче построения компиляторов [**eac2011**]. Считаемая классической «Книга дракона» [**dragonbook**] также подробно рассматривает вопрос разбора выражений.

1.4.1. Особенности разбора машинных языков

Машинное представление инструкций некоторой системы — всего лишь один из языков, и вся теория разбора выражений к нему применима. Однако, имеется ряд особенностей, позволяющих в ряде практически важных случаев строить более простые декодеры для машинного кода.

Переменная или постоянная длина инструкций. Многие RISC-процессоры имеют фиксированную длину инструкций, например, 16 или 32 бита. При этом адрес всех инструкций всегда выровнен. Для однозначного декодирования достаточно считать из памяти одно машинное слово.

[1] Т.е. пересекающих границу отдельной процедуры.
[1] Внутри одной процедуры.

С другой стороны, более древние CISC-системы чаще всего используют переменную длину инструкций. Так, в архитектуре IA-32 длина инструкций может составлять от 1 до 15 байт.

Префиксный код. В подавляющем числе случаев набор инструкций определяется *префиксным кодом* — никакая последовательность бит, определяющая разрешённую инструкцию, не является точным префиксом для другой инструкции. Это свойство означает отсутствие неоднозначности при декодировании инструкций с переменной длиной.

Влияние режима процессора на смысл. В значительной части архитектур процессор может находится в нескольких режимах работы, определяющих его функциональность. Например, процессоры IA-32 могут иметь следующие режимы работы[2]: 16-битный реальный, 16-битный «нереальный», 16-битный защищённый, 32-битный защищённый, 64-битный защищённый. Процессор ARM может быть в режиме 32-битных команд, 16-битных Thumb-команд, а также в недавно появившемся 64-битном режиме, доступным для некоторых моделей. При этом кодировка команд разных режимов может быть несовместима. Так, в архитектуре IA-32 в 64-битном режиме однобайтные последовательности из диапазона 0x40–0x4f не являются полными инструкциями, а представляют собой части более длинных команд. Во всех остальных режимах им соответствуют варианты инструкции `DEC`. Поэтому при декодировании необходимо учитывать режим.

Странности Иногда ISA может иметь совершенно неожиданные особенности. Например, в Intel Itanium ширина группы из трёх инструкций, составляющих связку(*англ.* bundle), почти всегда равна 128 битам. При этом ширина каждой отдельной инструкции равна 41 биту, а пять оставшихся бит несут общую информацию о группе в целом (т.н. шаблон). *Для некоторых* шаблонов ширина одной из инструкций удваивается до 82 бит, таким образом, в связке остаётся лишь две инструкции!

[2]Для краткости описания здесь опущены такие режимы, как System Management Mode и VMX root/non-root.

1.4.2. Ввод и вывод процедуры декодера

В реальном процессоре за задачу декодирования отвечает отдельный блок логических элементов микросхемы. В симуляторе ему соответствует некторая процедура, написанная на языке программирования. Рассмотрим, что подаётся на её вход и какие результаты она должна выдавать.

Как должно быть понятно из описанного выше, на вход декодера подаётся массив байт известной длины, полученный на фазе Fetch. Кроме того, ему может быть известен текущий режим процессора и адрес начала массива в памяти гостя.

В результате работы декодер должен вернуть код ошибки и результаты анализа последовательности в виде списка полей результата (мы вернёмся к ним чуть позже). При этом возможны следующие значения для кода ошибки.

1. Декодирование успешно (код 0). Массив байт был распознан как допустимая инструкция, и список полей содержит информацию о коде операции и её аргументах.
2. Декодирование неуспешно (код 1). Ни одна инструкция, определённая в архитектуре, не соответствует входному массиву байт. При этом содержимое полей результата не несёт смысла. Что происходит в этой ситуации дальше на этапе исполнения? Это зависит от архитектуры. Чаще всего невозможность декодировать ведёт к генерации исключения[1]. В некоторых случаях некорректная инструкция может быть воспринята как `NOP` — отсутствие операции.
3. Для ISA с переменной длиной инструкций возможна третья ситуация — входных данных недостаточно для принятия однозначного решения (код -1). Другими словами, на вход декодера передали только часть инструкции, и он, не имея информации о том, какие байты идут в памяти дальше, сообщает об этом.

[1]Подчеркнём, что эта ситуация не является внутренней ошибкой самого симулятора — поведение процессора на неизвестных инструкциях должно быть описано в документации и является штатной ситуацией в его работе.

На рис. 2.3 приведён пример алгоритма, сочетающего в себе итерации фаз Fetch и Decode и позволяющего провести декодирование для инструкций с переменной длиной.

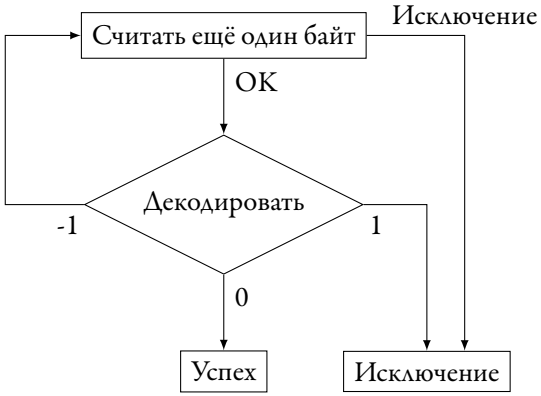

Рис. 1.3. Блок-схема декодирования, учитывающая переменную длину инструкции

У наблюдательного читателя может появиться вопрос: зачем использовать этот достаточно сложный и наверняка неэффективный алгоритм? Поскольку размер самой длинной инструкции всегда известен, а используемый код префиксный, то можно сделать Fetch последовательности, достаточной для вмещения как минимум одной инструкции. Затем декодировать её первый префикс, а оставшиеся «лишние» байты проигнорировать.

К сожалению, этот метод может генерировать исключения, отсутствующие в реальной системе, при попытке декодирования инструкций, находящихся близко к концу страницы или сегмента симулируемой памяти. Это связано с тем, что в системах, использующих механизмы страничной адресации или сегментации, разные диапазоны памяти имеют разные свойства. Если при чтении массива для декодирования «с запасом» пересекается граница между двумя такими диапазонами, и второй из них при чтении вызывает исключение, то и всё декодирование будет давать ложное исключение, тогда как на самом деле текущая инструкция корректна, не пересекает границ и должна

быть успешно распознана (рис. 2.4). Для того, чтобы избежать подобной ситуации, в алгоритме рис. 2.3 на каждой итерации читается и добавляется только один байт.

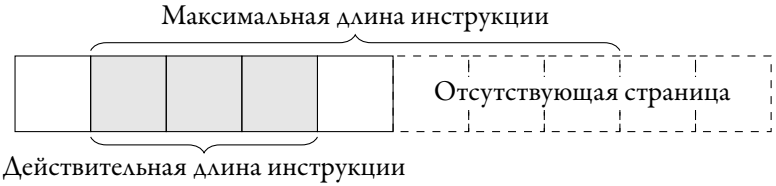

Рис. 1.4. Пересечение границы страниц при декодировании, вызывающее ложное исключение

1.4.3. Поля результата

Какую информацию должны содержать поля результата при успешном декодировании?

- Код операции (опкод), определяющий функцию, выполняемую инструкцией.
- Длину инструкции для ISA, в которых она является переменной.
- Информацию о каждом операнде. Она может включать в себя: порядковый номер регистра, его ширину, тип; для операций обращения к памяти — адрес и его ширина.
- Дополнительная информация, влияющая на исполнение. К ней может относиться наличие префиксов, модифицирующих операцию или размеры операндов и т.п.

Если сохранять результат в виде структуры языка Си, то она будет иметь следующий тип:

```
typedef struct decode_result {
    int length; // длина инструкции
    opcode_t opcode; // код операции
    int num_operands; // число операндов инструкции
    struct {
```

```
        operand_type_t type; // тип операнда
        union {
            int32    i32;
            int16    i16;
            int8     i8;
            float    f32;
            offset_t off;
        } value; // варианты хранимого значения
    } operands[MAX_OPERANDS]; // массив с операндами
} decode_result_t;
```

Для каждой конкретной архитектуры поля данной структуры будут свои собственные, отражающие особенности её формата инструкций.

1.4.4. Декодирование как распознавание шаблонов

В документации на центральные процессоры формат инструкций чаще всего описывается в виде таблиц, определяющих, какие биты машинного представления инструкции определяют логические поля, такие как опкод и операнды (см. рис. 2.5). Группы инструкций могут описываться одним и тем же форматом. Верно и обратное — несколько форматов могут описывать варианты одной и той же инструкции. При этом полное число различных форматов зависит от самой ISA и может быть достаточно велико.

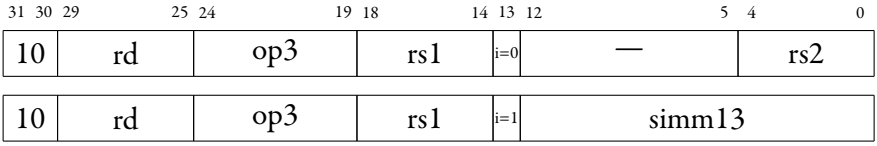

Рис. 1.5. Описание битовых полей инструкции. Пример взят из описания архитектуры SPARC [26], инструкция ADD и её варианты

Задача декодера состоит в сопоставлении входной строки данному набору шаблонов, нахождении совпадения, вычислении значений отдельных битовых полей и формировании значений полей логических. Для корректного декодирования любой входной последовательности должен соответствовать максимум один шаблон. Если же совпадений больше, то либо в кодировке инструкций, либо в реализации

декодера присутствует ошибка.

Связь битовых и логических полей инструкции

Каждое логическое поле инструкции, такое как номер регистра, может зависеть от нескольких битовых полей, а также режима процессора. Например, в последних поколениях IA-32 номер одного из векторных регистров ZMM (шириной 5 бит) в 64-битном режиме процессора определяется как комбинация следующих битовых полей : 3 бита `ModRM.Reg`, 1 бит `REX.R` и 1 бит `EVEX.`$\bar{\texttt{R}}$, причём последний из них следует инвертировать.

1.4.5. Оптимизация процесса декодирования

Число шаблонов, с которыми необходимо сравнивать декодируемый массив, может быть огромно. TODO

На рис. **??**

TODO Дерево

Рис. 1.6. Дерево декодирования

1.5. Увеличение скорости работы интерпретатора

Главным преимуществом рассмотренной схемы является её простота в реализации, модификации и отладке. Практически всегда в проектах по созданию программной модели нового процессора первым этапом является разработка интерпретационной модели, которая затем используется как эталон для тестирования последующих улучшений модели, оптимизирующих эффективность исполнения.

Основным недостатком интерпретатора является низкая скорость. Были разработаны многочисленные приёмы увеличения скорости интерпретации. Рассмотрим базовые идеи, используемые в них.

1.5.1. Сцепленная интерпретация

Одной из причин низкой скорости работы является неэффективное использование различных аппаратных ресурсов хозяйской системы, призванных уменьшить влияние явлений, разрушительных для конвейерной обработки. Так, из-за использования единого `switch` в теле цикла, из которого передача управления может быть осуществлена во множество мест, предсказатель переходов процессора не может каждый раз правильно предугадать адрес инструкции перехода, что вызывает сброс конвейера и задержку в несколько тактов. Этот негативный эффект проявляется в начале обработки каждой новой гостевой инструкции. Вместо концентрации условного перехода в одном месте желательно «размазать» его по многим местам в коде, уменьшив в каждом из них число вариантов адреса (в идеале — до одного). Этого можно достичь, если вызывать обработчик следующей инструкции сразу после конца работы текущей инструкции, без возвращения в общий цикл. Такой алгоритм интерпретации называется *сцепленным* (*англ.* threaded), см. рис. 2.6.

TODO Диаграмма switch-схемы

Рис. 1.7. Сравнение методов переключаемой и сцепленной интерпретаций

Пример реализации сцепленной интерпретации в псевдокоде дан ниже. Предполагается, что этап декодирования уже проведён, и в памяти содержится информация о том, какой будет следующая инструкция.

```
// Массив labels содержит адреса переходов для всех обработчиков
```

```
labels = [INSTR_A, INSTRb, ... INSTR_X, ... INSTR_Y ...];

INSTR_X: // Текущая инструкция X
    X_handler(operands, PC); // обработчик инструкции
    PC++;
    goto label[PC]; // Сразу к обработчику новой инструкции
```

Для реализации сцепленной схемы используемый для написания модели язык должен поддерживать указатели на метки в коде. Стандарт ANSI Си не позволяет этого делать. Но, например, в GNU C доступно соответсвующее расширение языка.

1.5.2. Интерпретация с кэшированием

Промежуточным звеном между интерпретатором и транслятором является кэширующий интерпретатор. В нём вместо достаточно медленной отдельной фазы генерации кода используется только кэш (промежуточное хранилище с быстрым доступом) декодированных инструкций (рис. 2.7). Если он реализован эффективно, то решение будет сбалансировано: при исполнении зацикленного кода модель ЦПУ будет достаточно быстрой, а при исполнении линейного кода будет незначительно проигрывать простому интерпретатору, рассмотренному ранее.

1.6. Модификация интерпретатора — добавление новых инструкций

Часто возникает задача расширения функциональности некоторой модели для представления функциональности нового процессора, отличающегося от старого наличием новых инструкций и дополнительных регистров процессора. Например, начиная с Intel Pentium IV в 2001 году были введены команды семейства SSE2, работающие с регистрами `XMM0`–`XMM7`.

Для того чтобы минимально модифицировать старый, хорошо отлаженный код модели, но при этом и поддержать новые системы, можно воспользоваться тем обстоятельством, что оригинальная модель не распознаёт новые инструкции как допустимые и должна вы-

звать обработку исключения `#UD` (*англ.* undefined opcode). Однако, мы даём модели «второй шанс», вызывая второй декодер новых инструкций. Если он подтверждает, что может декодировать переданный ему машинный код, вызывается новая часть интерпретатора, ответственная за новый набор инструкций (рис. 2.8).

Очевидно, что данную схему можно расширить для каскадного включения большего числа новых наборов инструкций. Её достоинство — гибкость подключения новой функциональности к уже существующей модели; дополнительные декодеры и симуляторы инструкций могут быть взяты из независимых источников и сравнительно легко адаптированы для использования. Недостаток тоже очевиден: последовательный вызов декодеров менее быстр, чем реализация, объединяющая их все в единую сущность.

1.7. Простой пример

Приведем код (на языке Си) интерпретационной модели некоторого упрощённого для целей данного примера процессора со следующей архитектурой.

1.7.1. Регистры

В рассматриваемом примере три регистра для арифметических операций и один указатель текущей инструкции.

- `R0` — регистр общего назначения
- `R1` — регистр общего назначения
- `R2` — регистр общего назначения
- `IP` — указатель команд

1.7.2. Команды

Набор инструкций включает в себя только две арифметические операции, а также работу с памятью.

- `ADD` — сложение, можно прибавлять к регистру регистр или число

– SUB — вычитание, можно вычитать из регистра число
– LOAD — загрузка ячейки памяти в регистр
– STORE — сохранение регистра в памяти

1.7.3. Код модели

```
struct DecodedInstr {
    enum Operation Op;
    enum Argument Arg1;
    enum Argument Arg2;
};
```

```
int R0, R1, R2, IP;  // Модель регистров
class Memory Mem;    // Модель внешней памяти
```

```
for (;;) { // бесконечный цикл
    int Instr = FetchInstr();
    struct DecodedInstr DecInstr = Decode(Instr);
    Execute(DecInstr);
}

int FetchInstr() {
    return Mem.Load32Bits(IP); // Загружаем 4 байта из памяти
    по адресу PC
}

struct DecodedInstr Decode(int Instr) {
    switch (Instr) { // Перебираем все реализованные инструкции
    case 0: // ADD R0, R0
        return {.Op = OP_ADD, .Arg1 = ARG_R0, .Arg2 = ARG_R0
    };
    case 1: // ADD R0,R1
        return {.Op = OP_ADD, .Arg1 = ARG_R0, .Arg2 = ARG_R1
    };
    // ...
    }
}
```

```
void Execute(struct DecodedInstr DecInstr) {
    int *Arg1, *Arg2;  // Указатели на аргументы операции
    // Какой первый аргумент операции?
    switch (DecInstr.Arg1) {
        case ARG_R0: Arg1 = &R0; break;
        case ARG_R1: Arg1 = &R1; break;
        case ARG_R2: Arg1 = &R2; break;
    }
    // Какой второй аргумент операции?
    switch (DecInstr.Arg2) {
        case ARG_R0: Arg2 = &R0; break;
        case ARG_R1: Arg2 = &R1; break;
        case ARG_R2: Arg2 = &R2; break;
    }
    // Выполнить операцию
    switch (DecInstr.Op) {
    case OP_ADD:
        *Arg1 += *Arg2;
        IP += 4; // Продвинуть указатель команд на следующую
    инструкцию
        break;
    case OP_SUB:
        *Arg1 -= *Arg2;
        IP += 4;
        break;
    case OP_LOAD:
        *Arg1= Mem.Load32Bits(*Arg2);
        IP += 4;
        break;
    // ...
    }
}
```

1.8. Заключительные замечания

Проект Bochs [12] является хорошим примером зрелого интерпретатора, содержащего сложную модель процессора для существующей архитектуры IA-32. В технических заметках к программе [**bochs-under-hood**] её авторы описывают множество полезных

приёмов, применимых как к организации модели-интерпретатора для процессора любой архитектуры, так и специфичных для архитектуры IA-32, являющейся одной из сложнейших в реализации.

1.9. Вопросы к главе 2

Вариант 1

1. Какие из указанных ниже компонентов обязательны для реализации интерпретатора:
   a) декодер,
   b) дизассемблер,
   c) кодировщик (енкодер),
   d) блоки реализации семантики отдельных инструкций,
   e) кэш декодированных инструкций.
2. Опишите, что происходит на стадии Fetch работы процессора.
3. Опишите, что происходит на стадии **Writeback** работы процессора. Для каких инструкций эта стадия будет опущена?
4. Какой вид программ обычно исполняется в привилегированном режиме процессора?
5. Какие эффекты могут наблюдаться при невыровненном (unaligned) чтении из памяти в существующих архитектурах:
   a) возникновение исключения,
   b) замедление операции по сравнению с аналогичной выровненной,
   c) данные будут считаны лишь частично,
   d) возможны все перечисленные выше ситуации?
6. Какая из следующих типов ситуаций при исполнении процессора является асинхронной по отношению к работе текущей инструкции?

a) прерывание (interrupt),
b) ловушка (trap),
c) исключение (exception),
d) промах (fault)?

7. Выберите правильный вариант окончания фразы: Сцепленный интерпретатор работает быстрее переключаемого (switched), так как

   a) удачно использует предсказатель переходов хозяйского процессора,
   b) кэширует недавно исполненные инструкции,
   c) транслирует код в промежуточное представление,
   d) не требует обработки исключений.

Вариант 2

1. Какой из типов регистров всегда присутствует во всех классических архитектурах:

   a) указатель стека,
   b) аккумулятор,
   c) указатель текущей инструкции,
   d) регистр флагов,
   e) индексный регистр.

2. Опишите, что происходит на стадии **Decode** работы процессора.
3. Опишите, что происходит на стадии **Advance PC** работы процессора. Для каких инструкций эта стадия будет опущена?
4. Какой вид программ обычно исполняется в непривилегированном режиме процессора?
5. Почему самый простой вид декодера машинных инструкций — однотабличный — не пользуется большой популярностью?

6. Выберите правильные варианты окончания фразы: Наличие единственного `switch` для всех гостевых инструкций в коде интерпретатора
    a) увеличивает его скорость по сравнению со схемой сцепленной интерпретации,
    b) упрощает его алгоритмическую структуру по сравнению со схемой сцепленной интерпретации,
    c) уменьшает его скорость по сравнению со схемой сцепленной интерпретации,
    d) не влияет на скорость работы интерпретатора.
7. Почему редко представляется возможным при симуляции процессора разместить все гостевые регистры на физических регистрах?

Рис. 1.8. Схема работы кэширующего интерпретатора

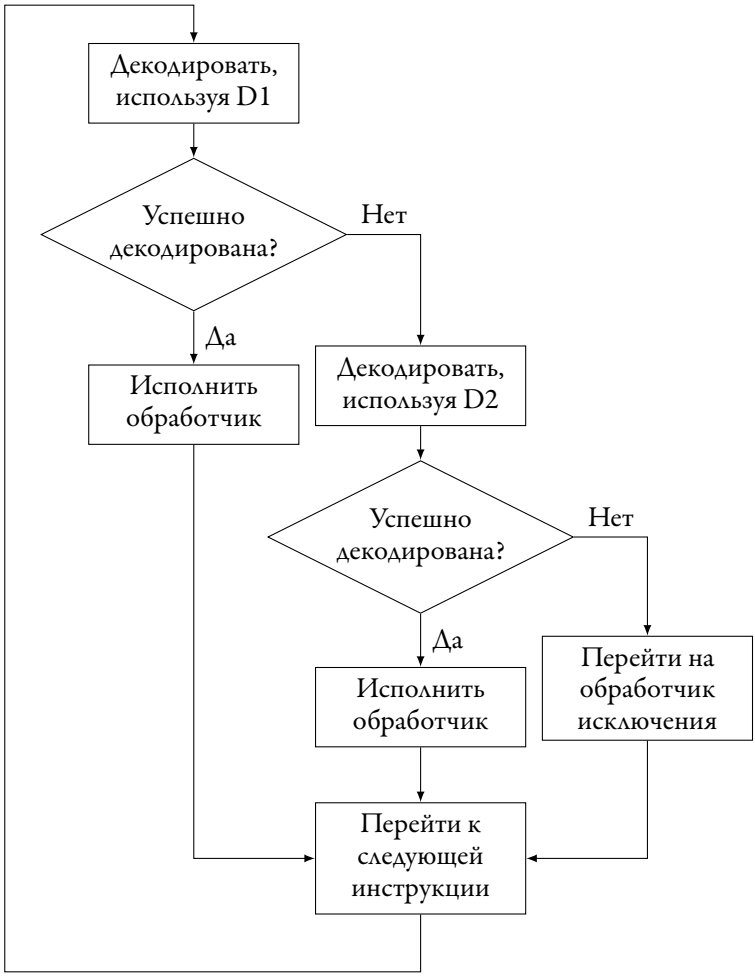

Рис. 1.9. Ступенчатая схема вызова декодеров при обнаружении инструкции, не поддерживаемой оригинальной моделью. При обнаружении в потоке инструкций машинного кода, не распознаваемого D1, управление передаётся на D2

Литература

1. Alpha Architecture Book. — 1998. — URL: http://openwatcom.mirror.fr/devel/docs/alphaarchitecturehandbook.pdf (дата обр. 20.10.2012) ; http://lib.mipt.ru/book/282937/.
2. AMD64 Architecture Programmer's Manual Volume 1: Application Programming. — Advanced Micro Devices. 2012. — URL: http://support.amd.com/us/Processor_TechDocs/24592_APM_v1.pdf (дата обр. 29.12.2012).
3. *Bellard Fabrice* QEMU, a Fast and Portable Dynamic Translator // FREENIX Track: 2005 USENIX Annual Technical Conference. — 2005. — URL: http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf (дата обр. 15.02.2012).
4. *Blaauw G.A.* The structure of SYSTEM/360, Part V: Multisystem organization // IBM Systems Journal. — 1964. — Т. 3. — С. 181. — URL: http://www.research.ibm.com/journal/sj/032/blaauw.pdf.
5. *Doran Mark*, *Zimmer Vincent*, *Rothman Michael* Beyond BIOS: Exploring the Many Dimensions of the Unified Extensible Firmware Interface // Intel Technology Journal. — 2011. — Окт. — Т. 15, вып. 1. — С. 8—21. — ISSN: 1535-864X. — URL: http://www.intel.com/technology/itj/2011/v15i1/index.htm (дата обр. 02.07.2012).
6. *Dulong Carole*, *Shrivastav Priti*, *Refah Azita* The Making of a Compiler for the Intel® Itanium™ Processor // Intel Technology Journal. — 2001. — Авг. — URL: http://download.intel.com/technology/itj/q32001/pdf/art_4.pdf.
7. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems / Leonid Baraz [и др.] // In 36th International Symposium on Microarchitecture. — 2003. — 191–201.

8. IEEE Standard for Floating-Point Arithmetic. — IEEE Computer Society, авг. 2008. — DOI: 10.1109/IEEESTD.2008.4610935. — URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4610933; IEEE Std 754-2008.
9. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volumes 1–3. — Intel Corporation. 2012. — URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (дата обр. 25.06.2012).
10. KVM wiki. — URL: http://www.linux-kvm.org/page/Main_Page.
11. *M. Russinovich*, *D. Solomon*, *A. Ionescu* Windows Internals, 6th Edition, Part 1. — Microsoft Press, 2012. — ISBN: 978-0-7356-4873-9.
12. *Mihoka Darek*, *Shwartsman Stanislav* Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure // ISCA-35 Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation. — URL: http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf (дата обр. 05.05.2012).
13. MPI: A Message-Passing Interface Standard. Version 2.2. — Message Passing Interface Forum. Сент. 2009. — URL: http://www.mpi-forum.org/docs/docs.html.
14. NetBSD/amd64. — 2007. — URL: http://www.netbsd.org/ports/amd64/ (дата обр. 09.09.2012).
15. OpenMP Application Program Interface version 3.0. — 2008. — URL: http://www.openmp.org/mp-documents/spec30.pdf.
16. *Pratt Ian* Xen and the art of virtualization. — 2006. — URL: http://www.cl.cam.ac.uk/netos/papers/2006-xen-ols.pdf.
17. Rosetta. — Apple Computer Inc. — URL: http://www.apple.com/asia/rosetta/ (дата обр. 09.09.2012).

18. Running Nonnative Applications in Windows 2000 Professional. — Microsoft Corporation. — URL: http://technet.microsoft.com/ru-ru/library/cc939094(en-us).aspx (дата обр. 09.09.2012).

19. *Singh Amit* Mac OS X Internals: A Systems Approach. — Addison Wesley. — ISBN: 978-0-321-27854-8. — URL: http://osxbook.com/ (дата обр. 09.09.2012).

20. *Sloss Andrew N.*, *Symes Dominic*, *Wright Chris* ARM System Developer's Guide. Designing and Optimizing System Software. — Morgan Kaufmann, 2004. — ISBN: 1-55860-874-5.

21. *Smith Tony* ARMv8 Tools – Everything You Need To Develop for AArch64. — Окт. 2012. — URL: http://blogs.arm.com/software-enablement/827-armv8-tools-everything-you-need-to-develop-for-aarch64/ (дата обр. 28.12.2012).

22. System V Application Binary Interface. AMD64 Architecture Processor Supplement. — AMD Corporation. — URL: http://www.x86-64.org/documentation/abi.pdf (дата обр. 14.02.2012).

23. The 68LC040 Emulator. — Apple Computer Inc., 1996. — URL: http://developer.apple.com/legacy/mac/library/documentation/mac/PPCSoftware/PPCSoftware-13.html (дата обр. 09.09.2012).

24. VirtualBox architecture. — Oracle Corporation. — URL: http://www.virtualbox.org/wiki/VirtualBox_architecture (дата обр. 25.09.2010).

25. VMware ESXi info page. — VMWare. — URL: http://www.vmware.com/products/vsphere/esxi-and-esx/index.html.

26. *Weaver D.L.*, *Germond T.*, *International SPARC* The SPARC architecture manual: version 9. — PTR Prentice Hall, 1994. — ISBN: 9780130992277. — URL: http://books.google.ru/books?id=JNVQAAAAMAAJ.

27. Режим Windows XP. — Microsoft Corporation. — URL: http://windows.microsoft.com/ru-RU/windows7/products/features/windows-xp-mode (дата обр. 09.09.2012).