Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров

А.С. Камкин
kamkin@ispras.ru

Аннотация. В работе рассматриваются вопросы автоматизированного построения тестовых программ, предназначенных для функциональной верификации модулей обработки переходов микропроцессоров. Формулируются задачи, возникающие при создании таких программ, и предлагаются техники их автоматизированного решения. Статья фокусируется на общих вопросах тестирования механизмов обработки переходов и не затрагивает проблемы специфичные для конкретных микропроцессорных архитектур. Предложенные техники можно использовать в промышленных генераторах тестовых программ.

1. Введение

Во всех универсальных, вычислительно полных по Тьюрингу, компьютерах, начиная с электронного калькулятора ENIAC (1946 г.), в том или ином виде присутствуют *инструкции перехода* — инструкции, позволяющие управлять потоком выполнения программы: осуществлять переходы на заданные инструкции, организовывать ветвления и циклы. Устройство компьютера, отвечающее за обработку таких инструкций, обычно называют *модулем обработки переходов*[1].

С точки зрения программиста, возможности модуля обработки переходов типичного микропроцессора невелики и сводятся к условным и безусловным переходам, вызовам подпрограмм и возвратам из них. В то же время внутренняя, скрытая от посторонних глаз, логика работы модуля может быть очень сложной и запутанной за счет использования механизмов повышения производительности: предсказания переходов, суперскалярного выполнения и многих других.

[1] Не следует понимать этот термин буквально — физически отдельного модуля обработки переходов может и не существовать. В этом случае корректнее говорить о логике обработки переходов.

Не секрет, что сложность является основным источником ошибок, допускаемых при проектировании микропроцессоров. В случае же если сложность является скрытой, обнаружить ошибки бывает достаточно трудно из-за отсутствия средств, позволяющих явно управлять поведением тестируемой системы или хотя бы наблюдать его. Таким образом, несмотря на кажущуюся простоту, модули обработки переходов микропроцессоров могут содержать в себе нетривиальные, трудно обнаруживаемые ошибки.

От правильности обработки переходов зависит как производительность микропроцессора, так и его корректность — систематическая ошибка в предсказании переходов может снизить производительность компьютера, а, например, неправильное вычисление адреса перехода или некорректное восстановление состояния после спекулятивного выполнения неверно предсказанных инструкций являются серьезными функциональными ошибками.

Поскольку модуль обработки переходов тесно взаимодействует с другими компонентами микропроцессора, имеет смысл проверять его работу системно, в связке с окружением. Наиболее распространенным способом системной верификации микропроцессоров и других программируемых устройств является тестирование моделей уровня регистровых передач (RTL-моделей) с помощью тестовых программ. Предметом статьи является автоматизация построения таких программ. В работе рассматриваются вопросы генерации *структур переходов* (по сути, графов потока управления), *трасс выполнения* (маршрутов в графе потока управления) и *управляющего кода* (вспомогательных инструкций, обеспечивающих проход по заданному маршруту). Предлагаемые подходы основаны на комбинаторных техниках и могут использоваться для разных микропроцессорных архитектур.

Статья организована следующим образом. В разделе 2 кратко описываются возможности модулей обработки переходов микропроцессоров, и рассматриваются проблемы, возникающие при их верификации. Раздел 3 определяет основные понятия используемого подхода, а его детальное описание дается в разделе 4. В разделе 5 описывается инструментальная поддержка, и приводятся примеры тестовых программ. Раздел 6 является заключением.

2. Обработка переходов в микропроцессорах

Рассмотрим кратко, что представляют собой модули обработки переходов современных микропроцессоров, и какие проблемы возникают при их верификации.

2.1. Модули обработки переходов

Модулем обработки переходов (*branch unit*) называется подсистема микропроцессора, отвечающая за обработку *инструкций переходов* —

инструкций, осуществляющих изменение потока управления[2]. Простейшими инструкциями данного типа являются *инструкции безусловного перехода*. К более сложным относятся *инструкции условного перехода*, которые изменяют поток управления только при выполнении определенных условий на значения операндов (обычно условия переходов выражаются через сравнения вида =, ≠, <, >, ≥, ≤). Во многих микропроцессорных архитектурах также предусмотрены *инструкции вызова подпрограмм и возврата из них*. При вызове подпрограммы перед передачей управления на нее происходит сохранение адреса следующей инструкции (адреса возврата) в специальном регистре или стеке. При завершении подпрограммы этот адрес считывается и по нему делается передача управления.

Современные микропроцессоры имеют *конвейерную архитектуру*, что означает, что они параллельно обрабатывают сразу несколько инструкций (каждую на своей стадии) [1]. В зависимости от организации микропроцессора выполнение перехода может приводить к *приостановке конвейера* (когда после выборки инструкции перехода загрузка конвейера приостанавливается до тех пор, пока не будет принято решение о переходе) или *сбросу конвейера* (когда все инструкции, загруженные на конвейер после инструкции перехода, сбрасываются, если принимается решение о выполнении этого перехода).

В некоторых микропроцессорах для борьбы со сбросами конвейера имеются так называемые *слоты задержки переходов* (*branch delay slots*) [1,2]. Это означает, что перед обработкой инструкции перехода микропроцессор успевает выполнить (точнее, загрузить на конвейер) одну или несколько инструкций, следующих за ней, (для каждой архитектуры это число фиксировано — в архитектурах MIPS или SPARC имеется один слот задержки, в SHARC DSP их два [2]). Инструкции, расположенные в слотах задержки перехода, выполняются независимо от того, выполнен переход или нет. Заметим, что в слот задержки нельзя помещать инструкцию перехода.

Для постоянной загрузки конвейера (минимизации числа сбросов) также применяется прогнозирование переходов. Стратегии прогнозирования могут быть *статическими* и *динамическими*, кроме того, они могут различаться для разных типов инструкций перехода [3]. Статические методы предсказания переходов являются наиболее простыми. Их суть состоит в том, что все переходы одного типа либо всегда прогнозируются выполненными, либо всегда прогнозируются невыполненными. Например, в ранних микропроцессорах архитектуры MIPS и SPARC условные переходы «назад» предсказывались выполненными, а условные переходы «вперед» — нет. Статическое предсказание используется и в современных микропроцессорах, но только в качестве «подстраховки», когда невозможно применение динамических методов [3].

[2] Под изменением потока управления понимается изменение *счетчика адреса* — регистра микропроцессора, содержащего адрес текущей инструкции.

Динамические методы подразумевают анализ истории переходов. Примером метода такого типа является двухуровневый алгоритм Йеха (Yeh), используемый в микропроцессорах Intel архитектуры P6 [4]. Первый уровень алгоритма представлен таблицей, хранящей для «каждой»[3] инструкции перехода историю четырех последних переходов. История хранится в виде четырехбитного паттерна и обновляется при каждом выполнении соответствующей инструкции (осуществляется сдвиг паттерна на один разряд влево, после чего в младший разряд записывается 1 или 0 в зависимости от того, выполнен переход или нет). Второй уровень состоит из шестнадцати двухбитных счетчиков с насыщением. Паттерн первого уровня используется в качестве индекса в таблице счетчиков. При осуществлении перехода счетчик инкрементируется, в противном случае декрементируется. Переход прогнозируется тогда и только тогда, когда счетчик находится в состоянии 2 или 3.

2.2. Особенности верификации модулей обработки переходов

Модули обработки переходов современных микропроцессоров имеют сложную организацию, и при их верификации, особенно если она осуществляется в связке со смежными компонентами, необходимо учитывать большое число факторов. Некоторые из них, типичные для многих микропроцессорных архитектур, перечислены ниже:

1. *тип инструкции перехода* (условный или безусловный переход, вызов подпрограммы или возврат из нее) — разные типы инструкций обрабатываются по-разному, поскольку у них просто напросто разная семантика;
2. *направление перехода* (вперед или назад) — направление перехода обычно влияет на прогноз, даваемый статическими методами;
3. *способ передачи адреса перехода* (непосредственно или через регистр) — если адрес перехода передается непосредственно, направление перехода можно определить уже на этапе декодирования инструкции;
4. *структура переходов* (граф потока управления) — некоторые архитектуры оптимизируют выполнение кода с определенной структурой переходов, например, «коротких» циклов;
5. *трасса выполнения* (история переходов) — история переходов влияет на результаты динамического прогнозирования, которые, в свою очередь, влияют на спекулятивную загрузку инструкций на конвейер;
6. *типы инструкций в слотах задержки переходов* (если архитектура предусматривает наличие слотов задержки) — в слотах задержки

[3] Понятно, что размер таблицы конечен. Идентификация инструкций осуществляется по младшим битам адреса.

могут находиться разные инструкции, в том числе, вызывающие исключения;

7. *типы спекулятивно выполняемых инструкций* (типы инструкций, следующих за переходом или находящихся по адресу перехода) — в микропроцессорах возможны ограничения на инструкции, которые можно выполнить спекулятивно;
8. *вызовы исключений* (исключения в слотах задержки и других инструкциях) — обработка исключений, вызванных инструкциями, находящихся в слотах задержки, обычно происходит особым образом, а инструкции, вызывающие исключения, не всегда можно выполнить спекулятивно;
9. *наличие зависимостей между инструкциями* (по регистрам и через память) — зависимости, например, между инструкциями, расположенными после перехода, и инструкциями до перехода (или инструкциями в слотах задержки) может приводить к блокировкам конвейера при спекулятивном выполнении;
10. *положение инструкции перехода в памяти* (относительно буфера предвыборки, кэш-памяти, страницы виртуальной памяти) — в зависимости от положения инструкции в памяти обработка перехода по ряду причин может происходить по-разному.

Как видно, факторов, влияющих на обработку переходов, достаточно много и ручная разработка полного комплекта тестовых программ вряд ли возможна. Огромное пространство состояний делают несостоятельной и чисто случайную генерацию. Целью настоящей работы является создание методологической поддержки для автоматизированной генерации тестовых программ. Поскольку задач, возникающих при тестировании модулей обработки переходов, достаточно много, мы решили ограничиться некоторыми из них, которые не связаны с техническими деталями работы таких устройств. Рассматриваемые вопросы затрагивают (в той или иной степени) только первые семь факторов из числа перечисленных (типы инструкций, структуры переходов и трассы выполнения).

3. Основные понятия предлагаемого подхода

В предлагаемом подходе построение тестовых программ осуществляется автоматически или полуавтоматически на основе формальной спецификации системы команд и описания тестовых ситуаций и зависимостей для инструкций микропроцессора. Построение тестовых программ осуществляется путем создания различных цепочек инструкций (последовательностей из инструкций разных типов с разной структурой переходов) и комбинирования для них тестовых ситуаций (событий, связанных с выполнимостью условий переходов, исключениями и т.п.) и зависимостей (взаимосвязей между инструкциями через регистры или

память) [5]. Рассматриваемый подход можно использовать и при создании тестовых программ на основе шаблонов, когда инженер-верификатор описывает набор ограничений (тестовый шаблон), а программа генерируется автоматически на основе заданных ограничений.

3.1. Структура тестовой программы

Тестовая программа представляет собой последовательность *тестовых вариантов*. Каждый тестовый вариант содержит *тестовое воздействие* — специально подготовленную цепочку инструкций, предназначенную для создания определенной ситуации в работе микропроцессора. Тестовое воздействие предваряется *инициализирующими инструкциями*, а после него может располагаться *тестовый оракул* — набор инструкций, проверяющих корректность состояния микропроцессора после выполнения тестового воздействия[4].

Таким образом, структуру тестовой программы можно описать с помощью формулы $Test = \{\langle Pre_i, Action_i, Post_i\rangle\}_{i=0,n-1}$, где Pre_i — это инициализирующие инструкции, $Action_i$ — тестовое воздействие, $Post_i$ — тестовый оракул. В простейшем случае каждая тестовая программа состоит из одного тестового варианта, то есть $Test = \langle Pre, Action, Post\rangle$.

Ниже в качестве примера приведен фрагмент тестовой программы в системе команд MIPS [6], содержащий один тестовый вариант.

```
      /////////// Инициализирующие инструкции ///////////

      // Инициализация регистров инструкции 1:
      lui v0, 0xdead
      ori v0, v0, 0xbeaf

      lui a0, 0xdead
      ori a0, a0, 0xbeaf

      // Инициализация регистров инструкции 2:
      ori a1, zero, 0x0

      j test_action_begin
      nop // Слот задержки

      ////////////// Тестовое воздействие //////////////

backward_jump:
      j test_action_end
```

[4] Основным способом проверки правильности работы модуля обработки переходов является сравнение трасс, полученных при выполнении RTL-модели микропроцессора и его эталонного симулятора, а не встроенные в программы тестовые оракулы.

```
      addi v1, v1, 1             // Ситуация: Overflow=false
test_action_begin:
      beq  v0, a0, backward_jump // Ситуация: Condition=true
      addi v1, a1, 2009          // Ситуация: Overflow=false
test_action_end:

      ///////////////// Тестовый оракул /////////////////

      // Запись эталонного значения
      ori t1, zero, 2010

      // Ошибочное завершение при несоответствии результата
      bne v1, t1, test_fails
      nop // Слот задержки
```

Тестовое воздействие состоит из четырех инструкций: инструкции безусловного перехода `j`; инструкции `addi`, складывающей содержимое второго регистра-операнда с 16-битным непосредственным значением и сохраняющей результат в первом регистре; инструкции `beq`, осуществляющей переход в случае совпадения значений регистров; и еще одной инструкции `addi`. Точка входа в тестовое воздействие обозначена меткой `test_action_begin`. Напомним, что в архитектуре MIPS у инструкций перехода имеется один слот задержки, поэтому в данном примере при выполнении переходов (инструкций `j` и `beq`) также выполняются инструкции `addi`. Тестовый оракул сравнивает выходной регистр инструкций `addi` (он одинаковый у обеих инструкций) с эталонным значением.

Несколько слов о вспомогательных инструкциях, используемых в примере. Инструкция `lui` загружает 16-битное непосредственное значение в разряды [31:16] регистра-результата (биты [63:32] получаются путем знакового расширения). Инструкция `ori` осуществляет побитовое ИЛИ значения второго регистра с 16-битным значением, заданным в третьем операнде (результат записывается в первый регистр). Инструкция `bne` выполняет условный переход в случае неравенства значений регистров.

3.2. Тестовый шаблон

Важным понятием подхода, которое следует рассмотреть подробнее, является понятие *тестового шаблона*. Тестовым шаблоном называется абстрактная форма представления тестового воздействия или тестовой программы в целом, в котором вместо конкретных значений операндов инструкций указываются ограничения (тестовые ситуации и зависимости), которым они должны удовлетворять. По сути, каждый тестовый шаблон (если он не является противоречивым) задает некоторую цель тестирования. Разные тестовые программы, соответствующие одному и тому же тестовому шаблону, эквиваленты в том смысле, что каждая из них достигает эту цель.

Ниже приведен тестовый шаблон из рассмотренного выше примера. Шаблон состоит из инструкций `j`, `addi`, `beq` и еще одной инструкции `addi`. Для обеих инструкций `addi` задана одинаковая ситуация *Overflow=false* (отсутствие переполнения), а для инструкции `beq` ситуацией является *Condition=true* (выполнимость условия перехода). Кроме того, между двумя инструкциями `addi` имеется зависимость (первый и второй регистры первой инструкции должны совпадать с первым регистром второй инструкции).

```
backward_jump:
      j test_action_end
      addi R, R, ?             @ Overflow=false
test_action_begin:
      beq  ?, ?, backward_jump @ Condition=true
      addi R, ?, ?             @ Overflow=false
test_action_end:
```

Выполнение описываемого этим шаблоном тестового воздействия начинается с инструкции `beq`, расположенной по метке `test_action_begin`. Условие перехода выполнено (*Condition=true*), поэтому осуществляется переход на метку `backward_jump`. Выполняемая в слоте задержки инструкция `addi` не вызывает переполнения (*Overflow=false*). Далее осуществляется безусловный переход по метке `test_action_end` вместе с выполнением еще одной инструкции `addi`, которая также не вызывает переполнения (*Overflow=false*).

3.3. Структура переходов

Важным атрибутом тестового шаблона является *структура переходов*. В широком смысле, структурой переходов называется граф потока управления, задаваемый тестовым шаблоном. В более практическом понимании, структурой переходов является тестовый шаблон, который не содержит информации о тестовых ситуациях и зависимостях, но в котором все переходы снабжены метками. Ниже приведена структура переходов для рассматриваемого примера и соответствующий ей граф потока управления.

```
backward_jump:
    j test_action_end
    addi ?, ?, ?
test_action_begin:
    beq  ?, ?, backward_jump
    addi ?, ?, ?
test_action_end:
```

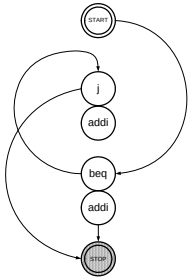

Рис. 1. Структура переходов и соответствующий граф потока управления.

Более подробно понятие структуры переходов рассматривается в разделе «*Построение структур переходов*».

3.4. Трасса выполнения

Помимо структуры переходов тестовый шаблон определяет *трассу выполнения*, задающую порядок обработки инструкций тестового воздействия (маршрут в графе потока управления). Для этого каждая инструкция условного перехода, входящая в тестовый шаблон, помечается последовательностью из значений истинности условий перехода (первое значение соответствует первому выполнению инструкции, второе — второму и т.д.; если инструкция перехода является недостижимой, она помечается пустой последовательностью). В рассматриваемом примере трасса выполнения полностью определяется истинностью условия перехода инструкции `beq` (*Condition=true*).

```
backward_jump:
    j test_action_end
    addi ?, ?, ?
test_action_begin:
    beq  ?, ?, backward_jump @ Condition=true
    addi ?, ?, ?
test_action_end:
```

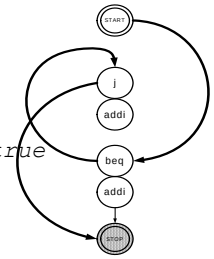

Рис. 2. Трасса выполнения и соответствующий маршрут в графе потока управления.

Более подробно трассы выполнения и метод их генерации рассматриваются в разделе «*Построение трасс выполнения*».

3.5. Управляющий код

Для того чтобы обеспечить заданную трассу выполнения, в тестовый шаблон встраивается *управляющий код*. Для рассматриваемого примера это можно и не делать, поскольку единственная инструкция условного перехода, содержащаяся в тестовом шаблоне, выполняется ровно один раз, следовательно, всю необходимую подготовку можно выполнить в инициализирующих инструкциях до выполнения тестового воздействия. В общем случае инструкции условного перехода могут выполняться по нескольку раз, причем значения истинности условий при этом могут меняться.

Рассмотрим структуру переходов, отличающуюся от рассмотренной ранее тем, что в ней отсутствует инструкция `j`, которая располагалась по метке `backward_jump`. Для этой структуры переходов рассмотрим трассу

выполнения, соответствующую последовательности {*true*, *false*} значений истинности условий инструкции `beq`.

```
backward_jump:
    addi ?, ?, ?
test_action_begin:
    beq  ?, ?, backward_jump @ Condition={true, false}
    addi ?, ?, ?
test_action_end:
```

Обозначим регистры инструкции `beq` как `R1` и `R2`. Для того чтобы обеспечить истинность условия при первом выполнении `beq`, инициализирующие инструкции заносят в `R1` и `R2` одинаковые значения. Еще раз подчеркнем, что инициализация не является частью управляющего кода, поскольку она выполняется вне тестового шаблона. Управляющий код должен изменить значение истинности условия с *true* на *false* после первого выполнения инструкции. Это можно сделать следующим образом.

```
backward_jump:
    addi R1, R1, 1 // управляющий код
    addi ?,  ?,  ?
test_action_begin:
    beq  R1, R2, backward_jump @ Condition={true, false}
    addi ?,  ?,  ?
test_action_end:
```

Заметим, что в данном примере изменение значения регистра `R1` (или `R2`) можно осуществить и в одной из инструкций `addi`, находящихся в слотах задержки переходов. Более подробно вопросы генерации управляющего кода рассматриваются в разделе «*Построение управляющего кода*».

4. Предлагаемый подход

В основе предлагаемого подхода к автоматизации генерации тестовых программ для модулей обработки переходов микропроцессоров лежат техники построения структур переходов, трасс выполнения и управляющего кода. Входными параметрами генератора являются классы эквивалентности инструкций переходов, *базовых блоков* (цепочек инструкций, не содержащих переходов) и инструкций в слотах задержки, а также ограничения на число переходов в тестовых шаблонах и на длину трасс выполнения. При описании предлагаемых техник будем, не ограничивая общности рассуждений, считать, что в микропроцессоре имеется один слот задержки перехода.

4.1. Построение структур переходов

Пусть инструкции переходов разбиты на классы эквивалентности, то есть задано фактор-множество $\Sigma_{jump} = \Sigma_{if} \cup \Sigma_{goto}$, где Σ_{if} — фактор-множество

условных переходов, а Σ_{goto} — безусловных[5]. Пусть, кроме того, заданы классы эквивалентности слотов задержки Σ_{delay} и базовых блоков Σ_{block}. Каждый элемент Σ_{if} имеет вид `if` *C* `goto` *i*, где *C* — это *условие*, которое может принимать значения *true* или *false*, а *i* — *метка перехода*, принимающая целое неотрицательное значение. Элементы множества Σ_{goto} имеют вид `goto` *i*. Для генерации структур переходов устройство слотов задержки и базовых блоков не важно.

Обозначим через Σ объединение $\Sigma_{jump} \cup \Sigma_{delay} \cup \Sigma_{block}$. Структурой переходов называется конечная последовательность S из элементов множества Σ, в которой после каждого перехода (элемента множества Σ_{jump}) следует слот задержки (элемент множества Σ_{delay})[6], а все метки переходов удовлетворяют ограничению $0 \le i < |S|$. Пример структуры переходов приведен ниже.

$$\{B_0, \texttt{if}\ C_1\ \texttt{goto}\ 6, D_2, B_3, \texttt{goto}\ 1, D_5, B_6, \texttt{if}\ C_7\ \texttt{goto}\ 0, D_8\}.$$

Здесь и далее мы будем обозначать слоты задержки буквой D с некоторым индексом, а базовые блоки — буквой B. Заметим, что в структурах переходов допускаются переходы на любые инструкции, в том числе на самих себя. Структура переходов, в которой нет ни одного перехода, называется *вырожденной*.

При построении структур переходов задаются такие параметры, как размер тестового шаблона и число инструкций переходов. Можно наложить и некоторые дополнительные ограничения, чтобы уменьшить число тестов, например, можно запретить подряд идущие базовые блоки, если нет переходов во внутренние блоки. Схема комбинаторной генерации структур переходов достаточна простая, поэтому мы не будем останавливаться на ней подробно. Сначала перебираются различные цепочки инструкций, включая инструкции переходов, затем — метки переходов. При использовании дополнительных ограничений осуществляется отсечение некоторых решений.

4.2. Построение трасс выполнения

Трассой выполнения структуры переходов $S = \{S[i]\}$ называется последовательность $T = \{T_j\}$ из индексов элементов S, которая удовлетворяет следующим свойствам:

[5] Инструкции вызова подпрограмм и возврата из них для простоты изложения не рассматриваются.

[6] Напомним, что мы рассматриваем случай, когда у инструкции перехода имеется ровно один слот задержки. Если слота задержки нет или их несколько, ограничения очевидным образом меняются.

- $T_0 = 0$[7];
- если $S[T_i] \in \Sigma_{if}$ и $S[T_i] =$ `if` C_i `goto` L_i, тогда
  - $|T| > i+1$, $T_{i+1} = T_i+1$[8] и выполняется одно из свойств:
    - если $T_{i+1} = |S|-1$, $|T| = i+2$ ($C_i = false$);
    - если $T_{i+1} < |S|-1$, $|T| > i+2$ и $T_{i+2} = T_i+2$ ($C_i = false$);
    - $|T| > i+2$ и $T_{i+2} = L_i$ ($C_i = true$);
- если $S[T_i] \in \Sigma_{goto}$ и $S[T_i] =$ `goto` L_i, тогда
  - $|T| > i+2$, $T_{i+1} = T_i+1$ и $T_{i+2} = L_i$;
- если $S[T_i] \in \Sigma_{block}$, тогда
  - если $T_i = |S|-1$, $|T| = i+1$;
  - если $T_i < |S|-1$, $|T| > i+1$ и $T_{i+1} = T_i+1$.

Примером трассы выполнения для рассмотренной в предыдущем разделе структуры переходов $\{B_0, \texttt{if}\ C_1\ \texttt{goto}\ 6, D_2, B_3, \texttt{goto}\ 1, D_5, B_6, \texttt{if}\ C_7\ \texttt{goto}\ 0, D_8\}$ является последовательность:

$$\{0, 1, 2, 6, 7, 8\}.$$

Ей соответствует следующий *поток выполнения* (последовательность элементов структуры переходов):

$$\{B_0, \texttt{if}\ C_1\ \texttt{goto}\ 6, D_2, B_6, \texttt{if}\ C_7\ \texttt{goto}\ 0, D_8\}.$$

Заметим, что в большинстве случаев по трассе можно восстановить значения истинности условий выполняемых переходов (исключения составляют ситуации, когда метка перехода указывает на инструкцию, следующую за слотом задержки). В дальнейшем будем считать, что каждое вхождение условного перехода в трассу выполнения (или поток выполнения) помимо индекса (соответственно, элемента структуры переходов) содержит также значение истинности условия. Таким образом, трасса и соответствующий ей поток выполнения для рассматриваемого примера имеют вид:

$$\{0, (1, true), 2, 6, (7, false), 8\} \text{ и}$$
$$\{B_0, \texttt{if}\ C_{1|true}\ \texttt{goto}\ 6, D_2, B_6, \texttt{if}\ C_{7|false}\ \texttt{goto}\ 0, D_8\}.$$

Трасса выполнения полностью определяется своей подпоследовательностью, содержащей только вхождения условных переходов. Такая подпоследовательность называется *редуцированной трассой*. Для рассматриваемого примера редуцированная трасса имеет вид $\{(1, true), (7, false)\}$.

[7] Для простоты считается, что выполнение начинается с первой инструкции.

[8] Выполнение слота задержки перехода.

Рассмотрим более сложную трассу выполнения той же самой структуры переходов:

{0, (1, *true*), 2, 6, (7, *true*), 8, 0, (1, *false*), 2, 3, 4, 5, (1, *true*), 2, 6, (7, *false*), 8}.

Ее редуцированное представление имеет вид:

$$\{(1, true), (7, true), (1, false), (1, true), (7, false)\}.$$

Или еще короче в форме отображения:

$$\{(1 \to \{true, false, true\}), (7 \to \{true, false\})\}.$$

Последовательность, состоящая из значений истинности условий перехода, называется *трассой выполнения инструкции перехода*. Пусть T — трасса выполнения структуры переходов, тогда через $T(i)$ будем обозначать трассу выполнения инструкции перехода с индексом i.

При построении трасс выполнения заданной структуры переходов указываются такие параметры, как максимальная длина трассы выполнения, максимальная длина редуцированной трассы или максимальная длина трассы одного перехода. Для перечисления трасс, удовлетворяющих такого рода ограничениям, можно использовать поиск в глубину. Ниже приводится алгоритм построения трасс выполнения, в котором используется представление редуцированных трасс в форме отображений. Кроме того, помимо трасс условных переходов в алгоритме используются трассы безусловных переходов (последовательности, состоящие только из *true*).

вход: структура переходов S;

вход: максимальная длина трассы перехода M[9];

выход: множество трасс выполнения R;

1. результат $R \leftarrow \varnothing$;
2. стек возвратов $B \leftarrow \varnothing$;
3. признак завершенности трассы $C \leftarrow false$;
4. индекс текущей инструкции $i \leftarrow 0$;
5. текущая редуцированная трасса $T \leftarrow \varnothing$;
6. **цикл**: ищем первую инструкцию перехода в S, начиная с инструкции i:
   a. если инструкция перехода не найдена, то
      i. $R \leftarrow R \cup \{T\}$;
      ii. если $B = \varnothing$, то алгоритм завершает работу;
      iii. $C \leftarrow true$;
      iv. $i \leftarrow top(B)$;
      v. перейти на **цикл**;
   b. если инструкция перехода найдена и j — ее индекс, то

[9] Для наглядности здесь используется ограничение на длины трасс выполнения отдельных переходов.

   i. $i \leftarrow j$;
   ii. если $C = false$ и $|T(i)| < M$, то
      1. $T \leftarrow (T \setminus \{i\}) \cup \{i \to T(i) \cdot \{k\}\}$, где
         a. $k = false$, если $S[i] \in \Sigma_{if}$;
         b. $k = true$, если $S[i] \in \Sigma_{goto}$;
      2. $push(B, i)$;
      3. $i \leftarrow l$, где
         a. $l = i+1$, если $S[i] \in \Sigma_{if}$;
         b. $l = L_i$, если $S[i] \in \Sigma_{goto}$ и $S[i] =$ `goto` L_i;
      4. перейти на **цикл**;
   iii. если $C = true$ и $T(i)_{|T(i)|-1} = false$, то
      1. $T \leftarrow (T \setminus \{i\}) \cup \{i \to T(i)_{0, \ldots, |T(i)|-2} \cdot \{true\}\}$;
      2. $i \leftarrow L_i$, где $S[i] =$ `if` C_i `goto` L_i;
      3. $C \leftarrow false$;
      4. перейти на **цикл**;
   iv. пока $B \neq \varnothing$ делать
      1. $T \leftarrow (T \setminus \{i\}) \cup \{i \to T(i)_{0, \ldots, |T(i)|-2}\}$;
      2. $i \leftarrow pop(B)$;
      3. если $T(i) \neq \varnothing$ и $T(i)_{|T(i)|-1} = false$, то
         a. $T \leftarrow (T \setminus \{i\}) \cup \{i \to T(i)_{0, \ldots, |T(i)|-2} \cdot \{true\}\}$;
         b. $i \leftarrow L_i$, где $S[i] =$ `if` C_i `goto` L_i
         c. $C \leftarrow false$;
         d. перейти на **цикл**;
   v. если $B = \varnothing$, то алгоритм завершает работу.

Рассмотрим пример построения множества трасс выполнения для структуры переходов $S = \{B_0,$ `if` C_1 `goto` $0,\ D_2\}$ при ограничении на длину трасс переходов $M = 2$.

В начале работы алгоритма $R \leftarrow \varnothing$, $B \leftarrow \varnothing$, $C \leftarrow false$, $i \leftarrow 0$ и $T \leftarrow \varnothing$. На первой итерации цикла ищется первая инструкция перехода в структуре S. Такая инструкция расположена по индексу $j = 1$ (это единственный переход в S). Делается присваивание $i \leftarrow j = 1$. Поскольку $C = false$, $|T(i)| < M$ ($|T(1)| = 0$) и $S[i] \in \Sigma_{if}$ ($S[1] =$ `if` C_1 `goto` 0), в трассу $T(i)$ добавляется $k = false$ ($T(1) = \{false\}$), в стек возвратов B добавляется индекс 1 ($B = \{1\}$), индекс i увеличивается на 1 ($i = 2$). После этого осуществляется поиск перехода, начиная с индекса $i = 2$. Переход не найден, поэтому в результат добавляется трасса $T = \{(1 \to \{false\})\}$, делается присваивание $C \leftarrow true$, индекс i получает значение верхушки стека B ($B = \{1\}$ и $i = 1$).

На следующей итерации ищется переход, начиная с $i = 1$. Находится инструкция, расположенная по индексу $j = i = 1$. Поскольку $C = true$ и в трассе $T(1)$ последний элемент равен *false*, он меняется на *true* ($T(1) = \{true\}$), после чего делаются присваивания $i \leftarrow L_1 = 0$ и $C \leftarrow false$. Снова ищется переход — на этот раз, начиная с $i = 0$. Индекс найденной инструкции $j = 1$. Делается присваивание $i \leftarrow j = 1$. Поскольку $C = false$, $|T(i)| < M$ ($|T(1)| = 1$) и $S[i] \in \Sigma_{if}$, в трассу $T(i)$ добавляется $k = false$ ($T(1) = \{true, false\}$), в стек B добавляется индекс 1 ($B = \{1, 1\}$), индекс i увеличивается на 1 ($i = 2$). Далее ищется переход, начиная с $i = 2$. Такой инструкции нет. В результат добавляется трасса $T = \{(1 \rightarrow \{true, false\})\}$, после чего делаются присваивания $C \leftarrow true$ и $i \leftarrow top(B)$ ($B = \{1, 1\}$ и $i = 1$).

На очередной итерации ищется переход, начиная с $i = 1$. Снова $j = i = 1$. Поскольку $C = true$ и в трассе $T(1)$ последний элемент равен *false*, осуществляется его замена на *true* ($T(1) = \{true, true\}$). После этого выполняются присваивания $i \leftarrow L_1 = 0$ и $C \leftarrow false$. Снова ищется переход, и находится по индексу $j = 1$. Делается присваивание $i \leftarrow j = 1$. Поскольку $|T(i)| = M$ и $B \neq \varnothing$, из трассы $T(i)$ удаляется последний элемент ($T(1) = \{true\}$), из стека извлекается индекс 1 и присваивается i ($B = \{1\}$ и $i = 1$). Последний элемент трассы $T(i)$ не равен *false*. Трасса укорачивается ($T(1) = \varnothing$), из стека извлекается индекс и присваивается i ($B = \varnothing$ и $i = 1$). Стек пуст, поэтому алгоритм завершает свою работу. Его результатом являются две трассы:

- $T = \{(1 \rightarrow \{false\})\}$;
- $T = \{(1 \rightarrow \{true, false\})\}$.

4.3. Построение управляющего кода

Для построения управляющего кода, то есть кода отвечающего за проход по требуемой трассе, используется следующий подход. Инструкциям перехода сопоставляются уникальные регистры, называемые *управляющими*, которые не используются в других инструкциях тестового шаблона[10]. В определенные позиции тестового шаблона вставляются инструкции, называемые *управляющими*, которые изменяют значения управляющих регистров так, чтобы обеспечить требуемую трассу выполнения. Совокупность управляющих инструкций образуют *управляющий код*.

Чтобы обеспечить заданные значения истинности условий, для каждой инструкции условного перехода в тестовую программу добавляется массив значений и осуществляется загрузка одного или обоих управляющих регистров последовательными значениями этого массива. Возникает задача

[10] При наличии зависимостей по регистрам между инструкциями переходов и другими инструкциями тестового шаблона задача построения управляющего кода существенно усложняется и не всегда разрешима.

нахождения позиций в тестовом шаблоне, в которых нужно делать такую загрузку. Для наглядности рассмотрим эту задачу при условии, что управляющие инструкции можно добавлять только в базовые блоки.

Пусть дана невырожденная структура переходов S и ее трасса T. Пусть i — индекс некоторой инструкции перехода в структуре S. Будем говорить, что трасса $T(i)$ является *пустой*, если i не содержится в T. Непустые трассы, содержащие равные между собой значения истинности условий, будем называть *тривиальными*. Таким образом,

- $\{\}$ — пустая трасса;
- $\{true, true, \ldots, true\}$ — тривиальная трасса;
- $\{false, false, \ldots, false\}$ — тривиальная трасса;

Напомним, что начальные значения истинности условий переходов обеспечиваются инициализирующими инструкциями тестового шаблона, поэтому управляющий код имеет смысл только для тех инструкций условных переходов, трассы которых являются непустыми и нетривиальными.

Сегментом $T_{i,j}$ трассы выполнения T называется ее фрагмент $\{T_i, T_{i+1}, \ldots, T_j\}$, заключенный между двумя соседними вхождениями одной и той же инструкции перехода. Пусть $T_i = (k, C_1)$, а $T_j = (k, C_2)$. В этом случае говорят, что $T_{i,j}$ является сегментом инструкции k. Если к тому же $C_1 = C_2$, сегмент называется *тривиальным*.

Очевидно, что управляющий код должен вызываться, по крайней мере, внутри нетривиальных сегментов. Нетрудно показать, что при достаточном числе регистров (когда каждая инструкция перехода имеет по крайней мере один управляющий регистр отличный от других) справедливо следующее утверждение.

Утверждение. Для заданной трассы выполнения структуры переходов построение управляющего кода возможно тогда и только тогда, когда внутри каждого нетривиального сегмента трассы есть хотя бы один базовый блок.

Рассмотрим структуру переходов S и ее трассу выполнения T. Пусть i — индекс некоторой инструкции условного перехода в структуре S, а I — подмножество индексов базовых блоков. Будем говорить, что множество I является *покрытием* инструкции i базовыми блоками, если для любого нетривиального сегмента T' инструкции i существует индекс $j \in I$, лежащий в T'. Базовые блоки, входящие в покрытие, будем называть *покрывающими*.

Рассмотрим следующую задачу. Для заданной структуры переходов и ее трассы выполнения требуется найти минимальное покрытие базовыми блоками каждой инструкции условного перехода. В некотором смысле, это

задача минимизации объема управляющего кода[11]. Задачи такого типа известны как *задачи о минимальном покрытии*. Для их эффективного решения используются приближенные алгоритмы, среди которых наиболее известен «жадный» алгоритм. Это простой метод, идея которого, применительно к нашей задаче, состоит в следующем. Для каждой инструкции условного перехода сначала выбирается базовый блок, который покрывает максимальное число нетривиальных сегментов рассматриваемой инструкции, затем из числа оставшихся берется блок, покрывающий максимальное число еще не покрытых сегментов и т.д. Алгоритм завершается, когда множество выбранных блоков становится покрытием.

В качестве примера рассмотрим структуру переходов

$$\{B_0, \texttt{if}\ C_1\ \texttt{goto}\ 6, D_2, B_3, \texttt{goto}\ 1, D_5, B_6, \texttt{if}\ C_7\ \texttt{goto}\ 0, D_8\}$$

и ее трассу выполнения

{0, (1, *true*), 2, 6, (7, *true*), 8, 0, (1, *false*), 2, 3, 4, 5, (1, *true*), 2, 6, (7, *false*), 8}.

Для инструкции условного перехода, расположенной по индексу 1, имеется два нетривиальных сегмента:

- {(1, *true*), 2, **6**, (7, *true*), 8, **0**, (1, *false*)}, содержащий два базовых блока 0 и 6;
- {(1, *false*), 2, **3**, 4, 5, (1, *true*)}, содержащий один базовый блок 3.

Каждый базовый блок из множества {0, 3, 6} покрывает только один из двух сегментов. В качестве покрытия можно использовать {0, 3} или {3, 6}.

Пусть для каждой инструкции условного перехода построено покрытие базовыми блоками. В каждый покрывающий блок вставляются управляющие инструкции. В частности, это могут быть инструкции вида:

```
lw   R1,(P1)
addi P1, P1, 4
```

Первая инструкция загружает в один из управляющих регистров (регистр `R1`) значение из заранее подготовленного массива. Вторая инструкция увеличивает текущую позицию в массиве (регистр `P1`). При таком подходе значение второго управляющего регистра считается фиксированным.

[11] Теоретически, управляющий код может оказывать влияние на ситуацию, создаваемую тестовым шаблоном, поэтому, чтобы снизить это влияние, желательно строить управляющий код как можно меньшего объема. Для большинства практических задач влиянием управляющего кода небольшого объема можно пренебречь.

Заполнение массивов осуществляется инициализирующими инструкциями. Для каждой инструкции условного перехода берется первое значение истинности условия. Последовательно просматриваются элементы совокупной трассы выполнения. Если встречается базовый блок, принадлежащий покрытию, в массив заносится произвольное значение управляющего регистра, обеспечивающее текущее значение истинности условия. Если встречается рассматриваемая инструкция перехода, смотрится, это ее последнее вхождение или нет. Если вхождение не является последним, текущее значение истинности меняется на следующее, иначе, инициализация массива завершается.

Для трасс с «простой» структурой возможно построение компактного управляющего кода, состоящего из одной управляющей инструкции на одну инструкцию условного перехода. Это удобно, поскольку управляющую инструкцию в этом случае можно разместить в слоте задержки перехода.

Рассмотрим инструкцию, в которой в качестве условия перехода используется сравнение `R1 > R2`. Для удобства будем считать, что значение регистра `R2` равно 0. Если трасса выполнения перехода имеет вид {*true*, …, *true*, *false*, …, *false*}, можно в управляющий регистр `R1` записать положительное число (величина которого зависит от количества элементов *true* в трассе и числа выполнений покрывающих блоков), а в качестве управляющей инструкции использовать декремент `R1`. Если же трасса имеет вид {*false*, …, *false*, *true*, …, *true*}, можно записать в `R1` неположительное значение и использовать в качестве управляющей инструкции инкремент `R1`.

Сравнения вида >, ≥ и ≤ обрабатываются аналогичным образом. При сравнении на равенство или неравенство возможно аналогичное упрощение управляющего кода для трасс выполнения перехода, содержащих одно вхождение *true* или *false* соответственно, то есть трасс вида {*false*, …, *false*, *true*, *false*, …, *false*} для равенства и {*true,* …, *true*, *false*, *true*, …, *true*} для неравенства.

5. Инструментальная поддержка

В данном разделе дается краткое описание инструментальной поддержки предложенных методов, и приводятся примеры сгенерированных тестовых программ.

5.1. Генератор MicroTESK

Рассмотренные техники построения тестовых программ для модулей обработки переходов микропроцессоров интегрированы в генератор MicroTESK [7], разработанный в Институте системного программирования РАН. Генератор строит тестовые программы путем комбинирования цепочек инструкций ограниченной длины. Для каждой цепочки перебираются множества зависимостей и тестовых ситуаций (для инструкций перехода в

качестве ситуаций используются трассы выполнения). Для сокращения перебора используются различные эвристики, например, факторизация инструкций, ограничение глубины зависимостей и ограничение длины трасс выполнения.

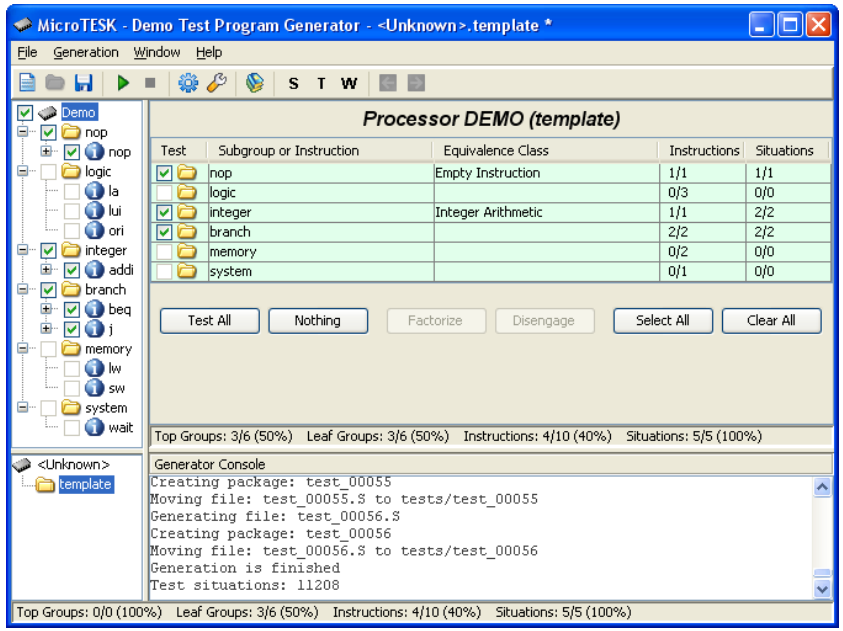

Рис. 3. Графический интерфейс демо-версии генератора MicroTESK.

Более подробная информация по методу, используемому в MicroTESK, доступна в статье [5]. Демо-версию генератора можно скачать на сайте [8].

5.2. Примеры тестовых программ

Ниже приведены фрагменты тестовых программ в системе команд MIPS, построенных автоматически генератором MicroTESK. При построении программ использовалась одна инструкция условного перехода (инструкция `beq`) и одна инструкция безусловного перехода (инструкция `j`). Для базовых блоков и слотов задержки использовались инструкции `nop` и `addi`. Рассматриваемые программы предполагают следующую обработку исключений. Если исключение возникает при выполнении инструкции базового блока, управление передается на следующую за ней инструкцию. Если исключение происходит в слоте задержки перехода, инструкция, вызвавшая исключение, заменяется на `nop`, после чего управление повторно передается на инструкцию перехода.

Пример 1

```
// Инициализация инструкции beq[3]
ori $15, $0, 0x0

// Инициализация инструкции addi[5]
lui $24, 0x8000
ori $24, $24, 0x2f56

// Тестовое воздействие
test_action_5016:

j 1f // Метка: 1, Трасса: {true}, Блоки: {}
1:
nop
beq $15, $0, 4f // Метка: 4, Трасса: {true}, Блоки: {}
4:
addi $2, $24, 0xcb01 // Переполнение
```

Пример 2

```
// Инициализация инструкции beq[1]
ori $2, $0, 0x0

// Инициализация инструкции addi[7]
lui $4, 0x7fff
ori $4, $4, 0x8f01

// Инициализация инструкции beq[8]
la   $25, beq_trace_array_1
ori  $7,  $0,  0x0
sw   $7,  0($25)
addi $25, $25, 0x4
ori  $7,  $0,  0x0
sw   $7,  0($25)
addi $25, $25, 0x4
lui  $7,  0xb566
ori  $7,  $7,  0xa7e8
sw   $7,  0($25)
addi $25, $25, 0x4
la   $25, beq_trace_array_1

// Инициализация инструкции addi[9]
lui $12, 0x7fff
ori $12, $12, 0xf916
```

```
// Тестовое воздействие
test_action_5197:

0:
beq $2, $0, 2f // Метка: 2, Трасса: {true, false,
 false}, Блоки: {2}
nop
2:
addi $2, $2, 0xffff // Управляющий код для beq[1]
lw $9, 0($25)       // Управляющий код для beq[8]
addi $25, $25, 0x4  // Управляющий код для beq[8]
addi $29, $4, 0x73a9 // Переполнение
beq $9, $0, 0b // Метка: 0, Трасса: {true, true, false},
Блоки: {2}
addi $15, $12, 0x5835 // Переполнение
```

6. Заключение

В статье рассмотрены общие задачи, возникающие при построении тестовых программ для модулей обработки переходов микропроцессоров, и методы автоматизации их решения. Предложенные методы можно использовать в инструментах генерации тестовых программ для микропроцессоров, в частности, они реализованы в генераторе MicroTESK. Описанный в статье подход не зависит от микроархитектуры микропроцессора. С одной стороны, это является достоинством, поскольку подход позволяет быстро создавать тесты на самых ранних стадиях проектирования. С другой стороны, для детального тестирования могут потребоваться более узконаправленные тесты, учитывающие специфику механизмов обработки переходов конкретной микропроцессорной архитектуры. Методы автоматизации разработки таких тестов являются темой наших дальнейших исследований.

Литература

[1] D. Patterson, J. Henessy. *Computer Organization and Design*. 3rd Edition, Morgan Kaufmann, 2005.

[2] Википедия (http://en.wikipedia.org), статья *Branch delay slot*.

[3] Википедия (http://en.wikipedia.org), статья *Branch predictor*.

[4] T.-Y. Yeh, Y.N. Patt. *Two-Level Adaptive Training Branch Prediction*. Proceedings of International Symposium on Microarchitecture, 1991.

[5] А.С. Камкин. *Генерация тестовых программ для микропроцессоров*. Труды ИСП РАН, т. 14, ч. 2. М., 2008. С. 23–63.

[6] MIPS64™ Architecture For Programmers. Revision 2.0. MIPS Technologies Inc., June 9, 2003.

[7] A. Kamkin. *MicroTESK: Automation of Test Program Generation for Microprocessors*. Proceedings of East-West Design & Test Symposium, 2009.

[8] http://hardware.ispras.ru.