Генерация тестовых данных для тестирования арифметических операций центральных процессоров

Е.В. Корныхин
kornevgen@ispras.ru

Аннотация. Рассматривается задача генерации тестовых данных для тестирования арифметической подсистемы центральных процессоров. Для ее решения предлагается использовать метод, позволяющий строить тестовые данные систематически на основе формального описания поведения отдельных команд микропроцессора. Предложенный метод апробирован на командах арифметической подсистемы микропроцессоров MIPS64.

1. Постановка задачи

Одним из способов тестирования микропроцессоров является проверка корректности их работы на некотором наборе программ (такие программы называют *тестовыми программами*). Программы компилируются, загружаются в память и выполняются микропроцессором. Результаты этого выполнения протоколируются и используются для анализа правильности работы микропроцессора. Задача построения тестовой программы может быть разбита на две подзадачи. Первая подзадача — сгенерировать последовательность вызовов операций процессора, в которой на месте операндов отсутствуют конкретные данные, но присутствует информация о том, как должна исполняться каждая команда, например, должно ли происходить переполнение при выполнении команды сложения. Такие последовательности называют *тестовыми шаблонами* (test template). Вторая подзадача — дополнить тестовые шаблоны конкретными числами, чтобы в результате получить готовую тестовую программу. Настоящая работа посвящена второй подзадаче. Алгоритм решения первой подзадачи можно найти, например, в работе [1].

Предлагаемая в данной работе методика построения тестовых данных для арифметической подсистемы процессоров не работает напрямую с микропроцессором — она использует описания поведения отдельных операций процессора. Поэтому ее можно отнести к классу модельно-ориентированных методик (model-based). Многие исследователи для тестирования микропроцессоров пользуются методами, не основанными на

моделях. Одним из часто используемых подходов является встраивание в сам дизайн процессора дополнительной схемы проверки его работы (BIST) [2]. Генераторы тестовых программ разрабатываются с учетом этой дополнительной схемы. При своем выполнении на микропроцессоре с внедренной BIST такие тестовые программы генерируют, помимо основных результатов, протокол своей работы, на основе которого делается заключение об успешности тестирования. Однако не во всех случаях можно дополнить дизайн процессора дополнительной схемой. В таких случаях следует пользоваться другими методами тестирования — к таким методам относится и предлагаемый метод генерации тестовых программ.

Проводимое тестирование микропроцессора, как и любое функциональное тестирование, должно проверять соответствие требованиям, предъявляемым к операциям микропроцессора, и быть «достаточно представительным» (обеспечивать высокий уровень «покрытия») [3]. Для некоторых микропроцессоров функциональные требования операций сформулированы в открытом стандарте архитектуры [4]. Например, так выглядит описание требований к арифметической операции ADD в стандарте архитектуры MIPS64:

Purpose:
To add 32-bit integers. If an overflow occurs, then trap.

Description: `rd ← rs + rt`
The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is signed-extended and placed into GPR *rd*.

Restrictions:
If either GPR rt or GPR rs does not contain signed-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Такие описания обычно формулируют все функциональные требования к операции, не вдаваясь в детали реализации.

Предлагаемый метод построения тестов нацелен на достижение полного покрытия всех возможных комбинаций ситуаций, описанных в функциональных требованиях, – *ветвей функциональности* – в рамках небольших, содержащих две-четыре команды программ. В приведенном выше описании выделены две ветви функциональности – сложение с переполнением («If the addition results in ... overflow…») и нормальное сложение («If the addition does not overflow …»). Будем называть *тестовым вариантом* набор значений аргументов одной команды. Ветви

функциональности разбивают пространство тестовых вариантов на классы эквивалентности: в один класс эквивалентности попадают тестовые варианты, при выполнении которых исполняется одна ветвь функциональности. Классы эквивалентности такого разбиения пространства тестовых вариантов будем называть *тестовыми ситуациями*. Кроме ветвей функциональности, классы эквивалентности в пространстве тестовых вариантов могут быть выделены на основании дополнительных эвристик. Источниками дополнительных эвристик могут служить общие знания об ошибках (например, ошибки часто происходят на значениях, близких к границе области допустимых значений параметра команды) или дополнительные знания разработчиков-схемотехников (узкие места реализации). В этих случаях могут быть выделены дополнительные классы эквивалентности в пространстве тестовых вариантов.

Итак, каждой команде процессора соответствует свой набор тестовых ситуаций. В тестовой программе может использоваться несколько команд, аргументы которых могут зависеть друг от друга. В результате могут возникать различные комбинации тестовых ситуаций, а количество тестовых программ, необходимых для покрытия всех возможных комбинаций, возрастает очень быстро. В таком случае вычисление тестовых данных становится крайне трудоемкой задачей, требующей автоматизации.

2. Существующие методы

Если рассматривать тестовую ситуацию в отдельности, то можно увидеть, что она представима набором условий, которые должны быть выполнены одновременно. Это наводит на мысль использовать для построения подходящих тестовых данных какую-нибудь технику решения систем условий. К таким техникам можно отнести CSP (constraint satisfaction problem) [15], SAT (SATisfiabitily problem) [5] и линейное программирование [13].

SAT – задача проверки выполнимости: для данной формулы логики предикатов требуется построить значения свободных переменных, при которых формула примет истинное значение. Существующие SAT-инструменты базируются либо на вероятностных алгоритмах, либо на алгоритме DPLL (Davis-Putnam-Logemann-Loveland algorithm), основанном на переборе с возвратом по глубине формулы конъюнктивного вида. Эффективность работы таких инструментов определяется размером формулы, истинность которой они выясняют, включающим как количество переменных в ней, так и размеры областей значений отдельных переменных.

В арифметической подсистеме современных процессоров приходится иметь дело с 64-битными переменными, представляющими натуральные числа. Размер формулы, зависящей лишь от одной такой переменной, равен 2^{64}, а на практике переменных может быть много. SAT-инструментов, способных эффективно справляться с формулами такого размера, сейчас не существует.

Поэтому применение SAT-инструментов к этой задаче достаточно ограничено. Видимо, этим объясняется более распространенное применение SAT-инструментов к задаче ATPG [8,10] — генерации входных сигналов схемы. Кроме того, отмеченные особенности SAT-инструментов не исключают возможности использовать их в качестве составных частей других инструментов. В таком случае SAT-инструменту будет передана лишь подзадача, которая будет уже лежать в области применимости этого инструмента. К примерам такого использования SAT-инструментов можно отнести SAT-based (Bounded) Model Checking [9], SAT-based Planning [12], SAT-based Formal Verification [11].

Задача линейного программирования известна очень давно [13]. Она имеет огромное количество полезных приложений (задача о максимальном паросочетании, транспортная задача и многие другие). Для описания арифметических команд процессоров применяются нелинейные операции, которые не дают использовать линейное программирование. Однако показано [14], что некоторые битовые операции над натуральными числами выражаются в виде задач целочисленного линейного программирования. Это дает надежду на использование целочисленного линейного программирования для решения частных задач в рамках более общих инструментов, хотя такая возможность пока не реализована.

Инструменты на основе CSP представляют серьезную альтернативу остальным технологиям. Constraint – «ограничение» – булевское выражение над произвольными переменными с ограниченными областями значений. Задача остается всё той же – найти значения переменных, удовлетворяющих всем заданным ограничениям. CSP отличается от других подходов алгоритмами нахождения нужных значений переменных. В основе большинства CSP-инструментов лежит семейство алгоритмов MAC (Maintaining Arc Consistency) [15]. В них семейство ограничений представляется гиперграфом. Вершины этого гиперграфа – переменные, гипердуги – ограничения (гипердуга – подмножество вершин, размер которого не обязательно равен двум). Алгоритм пытается постепенным сужением области значений переменных достичь выполнения всех ограничений, сначала выбирается одна гипердуга, сужаются области значений ее вершин-переменных, это затрагивает другие гипердуги – к ним алгоритм применяется рекурсивно. CSP-подход используется в хорошо известном инструменте генерации тестовых данных для тестовых шаблонов Genesys-Pro от IBM [16]. Это инструмент наиболее хорошо подходит для решения задаче, поставленной в данной статье. Однако закрытость инструмента Genesys-Pro не дает возможности провести более глубокий анализ используемых в нем технологий. Известно, что разработчики IBM решили написать свою версию алгоритма семейства MAC, адаптированную под определенные задачи. Кроме того, Genesys-Pro работает с моделью, в которой гиперграф ограничений уже описан. Заметим, что создание модели не входит в задачи Genesys-Pro и должно быть сделано вручную. Кроме Genesys-Pro, существуют и другие

работы по применению CSP к рассматриваемой в статье задаче. Например, для работы инструмента MA^{2}TG [17] необходима модель архитектуры микропроцессора на некотором языке с добавленными в модель ограничениями (assert), что также приходится делать вручную, а сами авторы не дают каких-либо систематических способов выделения ограничений. В подходе, предлагаемом в данной статье, вопрос выделения ограничений описан более четко.

3. Предлагаемый метод решения

Предлагается использовать специальную нотацию для записи отдельных тестовых ситуаций команд, а их комбинирование и вычисление тестовых данных будет производиться автоматически. В методе используется наличие описания поведения команд процессора в открытом стандарте архитектуры. Такие стандарты архитектуры доступны для MIPS64 [4], PowerPC [18], UltraSPARC [19] и многих других архитектур микропроцессоров. Построение тестовых данных по предлагаемому методу для данной команды выполняется следующим образом:

1. В открытом стандарте архитектуры найти формальное или полуформальное описание поведения данной команды.
2. Выделить тестовые ситуации, возникающие при выполнении данной команды; эти тестовые ситуации обычно соответствуют ветвям функциональности команды (они извлекаются из найденного описания поведения команды). Иногда на основе дополнительных эвристик (часто встречающихся ошибок, знаний разработчиков схемы) определяются дополнительные ситуации.
3. Описать выделенные ситуации на предлагаемом языке (TeSLa, см. ниже). Если такая ситуация соответствует ветви функциональности, ее описание задается условиями некоторой ветви потока управления псевдокода. Если в тестовую ситуацию включены дополнительные к ветви функциональности условия, их тоже следует описать на предлагаемом языке. Предлагаемый язык является уточнением псевдокода, на котором обычно описывается поведение команд процессора. В результате этого шага получится файл – описание тестовой ситуации.
4. Запустить специальный генератор, передав ему в качестве параметра описание тестовой ситуации, полученное на шаге 3. Генератор имеет API, позволяющий встроить вызов в другие программы тестирования процессора. В результате работы генератора будут получены тестовые данные для данной команды, отвечающие тестовой ситуации, переданной генератору.

3.1. Язык описания тестовых ситуаций TeSLa

Язык описания тестовых ситуаций TeSLa [22] (Test Situation Language) представляет собой простой императивный язык с единственным типом

данных – целыми числами, состоящими из заданного числа бит (никаких явных ограничений на число бит не предполагается), операторами присваивания и утверждения (см. ниже). Язык включает все операции псевдокода, которые обычно используются для описания поведения команд процессора:

- получение бита числа с заданным номером (например, «x[7]» – 7-й бит числа x; биты нумеруются от младших к старшим; младший бит имеет номер 0; биты располагаются в числе по убыванию номеров);
- получение диапазона бит с заданными номерами границ этого диапазона (например, «x[8. .5]» – диапазон бит числа x с 8-го по 5-й, включая оба граничных бита);
- конкатенация чисел (например, «x.y» – число, двоичная запись которого состоит из двоичной записи числа x, за которой следует двоичная запись числа y);
- битовая степень числа – конкатенация числа с самим собой нужное количество раз (например, «x^5» – 5 раз повторенная запись числа x);
- привычные арифметические операции (сложение, вычитание, умножение);
- операции сравнения чисел на «больше-меньше»;
- операции знакового увеличения размера числа (например, «(64)x» – это 64-х битное число, равное и по модулю и по знаку числу x);
- логические операции AND и OR;
- оператор присваивания (например, «x := 5;»);
- оператор утверждения (например, «ASSERT x = 5;» – утверждение, что при исполнении данного оператора значение переменной *x* должно равняться 5).

Описание ситуации на таком языке состоит из последовательности операторов присваивания и утверждения. Заканчиваться последовательность должна «оператором ситуации», который семантически идентичен оператору утверждения с меткой-идентификатором предиката этого оператора. После выполнения последовательности операторов из описания тестовой ситуации процессор должен находиться в той тестовой ситуации, описание которой и составлялось.

Поскольку язык является пока только прототипом, он не включает условный оператор и операторы цикла. Однако для описания всех тестовых ситуаций арифметических команд процессора MIPS64, приводящих к исключениям (exception), условный оператор и оператор цикла не требуются, и средств даже прототипа языка хватает. Язык не включает логическую операцию NOT. Это связано с техникой работы генератора. Однако опять же, например, для описания всех тестовых ситуаций арифметических команд процессора MIPS64, приводящих к исключениям, операция NOT не требуется. В некоторых случаях приходится использовать вспомогательных версии предикатов, используемых в псевдокоде, операторов сравнения, логических

операторов, в которые уже внесён оператор NOT (например, вместо NOT(NotWordValue (x)) использовать WordValue(x)).

3.2. Генератор тестовых данных

Генератор [22] на входе получает файл с описанием тестовой ситуации, транслирует его в промежуточное представление и исполняет промежуточное представление, формируя значения аргументов данной команды. Генератор основан на CLP (Constraint Logic Programming) – логическом программировании с ограничениями [21]. Такой генератор позволяет, используя механизмы логического программирования (унификацию и перебор с возвратом), собирать по ходу выполнения логической программы наборы ограничений и тут же проверять их на совместность (здесь работает как раз CSP-часть логического интерпретатора). Несовместность набора ограничений на текущем шаге – достаточный повод сделать возврат. Выполнение логической программы может соответствовать тексту описания тестовой ситуации (поскольку описание тестовой ситуации есть *последовательность* операторов). А ограничения, добавляемые на очередном шаге выполнения логической программы, будут следовать из текущего оператора в описании тестовой ситуации. В качестве логического интерпретатора в генераторе используется инструмент с открытым кодом ECLiPSe [20]. Этот логический интерпретатор является достаточно мощным и обладает удобными средствами для трансляции в него описаний тестовых ситуаций. Отсутствие в языке описания тестовых ситуаций операторов цикла гарантирует завершение работы инструмента на любом описании тестовой ситуации.

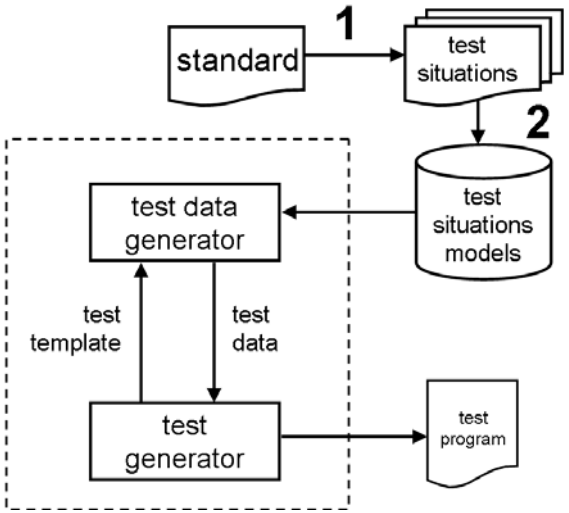

Рис. 1. Структура системы генерации тестов

На Рис. 1 представлена структура всей системы генерации тестов. Подсистема, о которой идет речь в данной статье, называется test data generator. Она получает на вход тестовый шаблон от test generator и возвращает ему тестовые данные (test data), соответствующие этому тестовому шаблону. Для своей работы test data generator использует модели тестовых ситуаций – описания их на языке TeSLa, которые составляются из открытого стандарта. Цифрами помечены шаги ручной работы: 1 – выделение тестовых ситуаций, 2 – составление описаний тестовых ситуаций на языке TeSLa.

3.3. Пример

В качестве примера применения метода рассмотрим построение тестовых данных для операции ADD. В стандарте архитектуры MIPS64 описание этой операции располагается на странице 35. Описание состоит из нескольких частей:

- **Format** фиксирует порядок операндов в команде (сначала возвращаемое значение, а потом аргументы);
- **Purpose** одним предложением описывает семантику команды;
- **Description** содержит более полное описание функциональности команды (из этого поля получаются ветви функциональности команды ADD);
- **Restrictions** содержит предусловие выполнения команды;
- **Operation** состоит из псевдокода, описывающего поведение команды ADD;
- **Exceptions** фиксирует те исключение, которые может порождать команда ADD;
- **Programming Notes** содержит дополнительный текстовый комментарий, не относящийся напрямую к остальным частям описания.

Приведенный формат описания арифметической команды является стандартным для описания архитектуры MIPS64. Выделяем ветви функциональности, читая **Description**:

Description: `rd ← rs + rt`
The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is signed-extended and placed into GPR *rd*.

Поскольку описание функциональности разбивается на два случая («If the addition results … overflow» и «If the addition does not overflow…»), команда ADD имеет две ветви функциональности – каждая соответствует своему случаю. Первая ветвь соответствует возникновению переполнения, вторая ветвь – нормальному исполнению команды. Выделим тестовые ситуации на

основе найденных ветвей функциональности. Первой тестовой ситуацией будет «возникновение переполнения», второй – «нормальное исполнение команды». Дополнительных тестовых ситуаций выделять не будем. Далее каждую тестовую ситуацию опишем на TeSLa. Для этого задействуем **Operation**:

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← sign_extend(temp31..0)
endif
```

Вначале идет предусловие: если верно, что «NotWordValue(GPR[rs]) or NoWordValue(GPR[rt])», то поведение операции непредсказуемо (UNPREDICTABLE). Во всех остальных случаях сначала идет присваивание в переменную temp (обратите внимание, ее размерность – 33 бита), а затем ветвление, как раз и обозначающее две ветви функциональности (две тестовые ситуации). Чтобы произошла тестовая ситуация «возникновение переполнения», достаточно, чтобы $temp_{32} \neq temp_{31}$. Чтобы произошла тестовая ситуация «нормальное исполнение команды», наоборот, достаточно, чтобы $temp_{32} = temp_{31}$. Тогда на TeSLa код первой тестовой ситуации будут выглядеть следующим образом:

```
VAR rs : 32;
VAR rt : 32;
ASSERT WordValue(rs) AND WordValue(rt);
temp := rs[31].rs[31..0] + rt[31].rt[31..0];
SITUATION temp[32] # temp[31] IS IntegerOverflow .
```

а второй тестовой ситуации – следующим образом:

```
VAR rs : 32;
VAR rt : 32;
ASSERT WordValue(rs) AND WordValue(rt);
temp := rs[31].rs[31..0] + rt[31].rt[31..0];
SITUATION temp[32] = temp[31] IS NormalAddition .
```


По сравнению с **Operation**, добавилась информация о размерах переменных. Она получена из чтения **Description**.

Осталось запустить генератор, который построит значения для переменных, перечисленных в начале описания тестовой ситуации с ключевым словом VAR. Например, для первой тестовой ситуации могут быть сгенерированы такие значения: rs = 2147483650, rt = 2147493620, а для второй – rs = 147483650, rt = 47493620. Генератору предоставляется самостоятельный выбор конкретных значений в тех случаях, когда подходящий тестовый вариант не единственный. Это дает возможность при разных запусках генератора строить разные значения и получать на выходе разные тестовые программы, что может способствовать более качественному тестированию процессора.

4. Заключение

В данной работе представлен метод генерации тестовых данных для тестирования арифметической подсистемы центральных процессоров, нацеленный на достижение различных комбинаций тестовых ситуаций в последовательностях обращений к арифметическим командам. Метод позволяет строить тестовые данные систематически на основе формального описания поведения отдельных команд микропроцессора. Предложенный метод апробирован на командах арифметической подсистемы микропроцессоров MIPS64, сейчас он начинает использоваться в промышленном проекте по тестированию одного из микропроцессоров этого семейства.

В данный момент созданы прототипы языка описания тестовых ситуаций и генератора данных по этим описаниям, но значимые практические результаты пока не получены. На следующем этапе язык описания тестовых ситуаций будет расширен, чтобы в нем можно было определить все тестовые ситуации арифметических операций процессора MIPS64 (это требуется для практического применения метода). В дальнейшем предполагается доработать предложенный метод генерации тестовых данных, чтобы он стал применим для команд микропроцессора, не имеющих такого подробного формального или полуформального описания (к таким командам относятся, например, операции с памятью).

Литература

[1] А. С. Камкин. Генерация тестовых программ для микропроцессоров. Труды ИСП РАН т. 14, ч. 2, стр. 23-64.

[2] http://en.wikipedia.org/wiki/Built-in_self-test.

[3] В. В. Кулямин, А. К. Петренко. Тестирование на основе моделей: теория, инструменты, применения — Ломоносовские чтения 2004.

[4] MIPS64® Architecture for Programmers Volume II: The MIPS64® Instruction Set (http://www.cs.ucr.edu/~junyang/teach/F04_203A/MIPS64manual.pdf)

[5] J. Gu, P. W. Purdom, J. Franco, B. W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. DIMACS series in Discrete Mathematics and Theoretical Computer Science, vol. 35, pp. 19-152, American Mathematical Society, Providence, RI, 1997.
[6] http://en.wikipedia.org/wiki/DPLL_algorithm.
[7] http://en.wikipedia.org/wiki/Automatic_test_pattern_generation.
[8] Z. Zeng, K. R. Talupuru, M. Ciesielski. Functional test generation based on word-level SAT. Journal of Systems Architecture: the EUROMICRO Journal, 51(8):488-511, 2005.
[9] Heon-Mo Koo, P. Mishra. Test generation using SAT-based bounded model checking for validation of pipelined processors. Great Lakes Symposium on VLSI, Proceedings of the 16-th ACM Great Lakes symposium on VLSI, pp. 362-365, Philadelphia, PA, USA. 2006.
[10] G. Fey, S. Junhao, R. Drechsler. Efficiency of Multi-Valued Encoding in SAT-based ATPG. ISMVL, pp. 25, 2006.
[11] M. Ganai, A. Gupta. SAT-Based Scalable Formal Verification Solutions. Springer. 2007.
[12] Ji-Ae Shin, E. Davis. Processes and continuous change in a SAT-based planner. Artificial Intelligence, 166(1-2):194-253, 2005.
[13] http://ru.wikipedia.org/wiki/Линейное_программирование.
[14] R. Brinkmann, R. Drechsler. RTL-datapath verification using integer linear programming. IEEE VLSI Design'01 & Asia and South Pacific Design Automation Conference, Bangalore, pp. 741-746, 2002.
[15] А. Л. Семенов. Методы распространения ограничений: основные концепции. Труды Совещания по интервальной математике и методам распространения ограничений ИМРО'03, Новосибирск, 2003.
[16] E. Bin, R. Emek, G. Shurek, A. Ziv. Using a constraint satisfaction formulation and solution techniques for random test program generation. IBM Systems Journal Artificial Intelligence. Volume 41, no. 3. 2002.
[17] Li Tun Li, Zhu Dan, Guo Yang, Liu GongJie, Li SiKun. MA2TG: a functional test program generator for microprocessor verification. Proceedings of 8-th Euromicro Conference on Digital System Design, pp. 176-183, 2005.
[18] Enhanced PowerPC Architecture http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600682CC7.
[19] UltraSPARC Architecture 2005 Specification (Hyperprivileged Edition) http://opensparc-t1.sunsource.net/specs/UA2005-current-draft-HP-EXT.pdf.
[20] K. Apt, M. Wallace. Constraint Logic Programming using Eclipse. Cambridge University Press, 2007.
[21] K. Marriott, Peter J. Stuckey. Programming with Constraints. MIT Press, 1998.
[22] TeSLa-project http://tesla-project.googlecode.com.