Метод генерации тестовых данных для тестирования арифметических операций процессоров MIPS64

Е.Корныхин

Аннотация

Рассматривается задача генерации тестовых данных для тестирования арифметической подсистемы центральных процессоров MIPS64. Для решения этой задачи предлагается метод, использующий формальное описание поведения процессора. Это описание предлагается делать на специализированном языке, допускающем автоматическое решение поставленной задачи.

1 Постановка задачи

Производители компьютерных систем стараются по возможности избавить все компоненты этих систем от ошибок в работе. Одним из таких компонент являются центральные процессоры. Одним из широко используемых методов проверки правильности работы центрального процессора является тестирование. В данной работе рассматривается тестирование операций центрального процессора архитектуры MIPS64 [1]. Стандарт этой архитектуры описывает поведение каждой операции центрального процессора. При некоторых параметрах операция должна работать корректно и формировать результат, при других – должна создавать исключительную ситуацию (например, операция деления должна создавать исключение при подаче в качестве делителя числа ноль). Оба варианта поведения соответствуют своей *тестовой ситуации*. Примеры тестовых ситуаций: «происходит переполнение», «результат сформирован».

Нужно реализовать много различных тестовых ситуаций. Стандарт архитектуры MIPS64 включает 246 инструкций. Из них 35 инструкций являются арифметическими. Они порождают много тестовых ситуаций. Из-за большого количества тестовых ситуаций их построение необходимо автоматизировать.

Таким образом, требуется построить программу, которая для заданной операции центрального процессора выдавала такие значения ее аргументов, чтобы при исполнении этой операции с построенными аргументами реализовалась определенная тестовая ситуация.

2 Существующие методы

Как и для задачи генерации тестовых данных для программного обеспечения, методы генерации тестовых данных для аппаратного обеспечения можно разделить на две группы: динамические методы и статические методы. Динамические методы предполагают запуск тестируемой системы на исполнение и анализ ее работы. Статические методы используют модель тестируемой системы и алгоритмы работы с этими моделями. Задача тестирования микросхем стала актуальна практически сразу, как появились сами микросхемы. Исторически первыми были предложены динамические методы. Настоящие программные методы появились тогда, когда мощность вычислительной техники достигла нужного уровня.

Одна и та же микросхема может быть представлена различными способами. От способа представления зависят и алгоритмы, которые можно использовать для нахождения тестовых данных. Первым представлением были структурные схемы (графическое изображение в виде совокупности частей, на которые ее можно разделить по определенным признакам, и связей между частями с указанием направления передачи воздействий). Первое упоминание о задаче тестирования таких микросхем было проведено в 1959 году Ричардом Элдредом [4]. Задача была названа ATPG (Automatic Test Pattern Generation). Все методы решения этой задачи можно поделить на функциональные и структурные. В функциональных методах решения схема рассматривается как черный ящик или, математически, лишь как функция значений на выходных портах от значений на входных портах. Структурные методы наоборот рассматривают схему как белый ящик. В них пытаются создать тесты так, чтобы задействовать все составляющие схемы («провода»). Исторически первым алгоритмом ATPG был D-алгоритм, в котором строились тестовые последовательности для нахождения «константных ошибок» (т.е. случаев, когда в некотором месте схемы вместо правильного всегда генерировалось константное значение - ноль или единица). Ошибка распространялась к входам и к выходам, постепенно формируя тестовые значения на входе и выходе. Другие методы решения задачи ATPG используют SAT-инструменты (SATisfiability) и алгоритмы работы с BDD (Binary Decision Diagram). Задача тестирования плат оказала влияние даже на их проектирование: был разработан набор требований к структуре платы – DFT (Design For Test), призванный облегчить ее тестирование.

Другое представление микросхемы – RTL-модель (Register Transfer Language). Первое упоминание о таких моделях относится к 1980 году [5]. Использование этих моделей обусловлено возможностью синтеза схем на их основе. Таким образом можно выполнять все тестирование на RTL-модели и в результате сгенерировать схему, не содержащую ошибок, не потеряв при этом нужных свойств модели и не изменив функциональности. Группа исследователей из Университета Корсики [6] предложила использовать для поиска тестовых данных выделение путей в графе потока управления и применения к ним логических систем программирования с ограничениями

(CLP [3]). Этот метод применялся и к программам на языках высокого уровня, но большие выразительные возможности таких программ становились препятствиями на пути эффективного применения этих идей. Исследователи из Англии предложили использовать SAT для решения задачи генерации тестовых данных прямо на основе RTL-модели [7]. Группа итальянских исследователей предложила для этой же задачи использовать генетические алгоритмы [8].

И, наконец, третье представление – программная модель. Она не имеет прямого отношения к микросхеме, но описывает на языке достаточно высокого уровня ее функциональность. Именно такое представление используется в данной работе. Это направление развивается не так сильно. Можно отметить работу немецких исследователей, которые попытались применить целочисленное линейное программирование к задаче верификации [9]. Применить этот метод в нашем случае не удастся, потому что он не предполагает операции умножения над числами (умножение не является линейной операцией) и не гарантирует получение результата за приемлемое время.

Предлагаемый метод применим, когда RTL-модель либо еще не существует (а стандарт уже разработан!), либо когда она недоступна или неизвестна. Наличие стандарта (по сути, документа), в котором для каждой операции описана ее функциональность, предопределило его использование в качестве основы метода генерации тестовых данных. Использование стандарта является новизной метода. Зачастую описание функциональности в стандарте не содержит лишней информации по сравнению с RTL-моделью. Это обуславливает большую эффективность генерации тестовых методов из такого описания, нежели из RTL-модели.

3 Текущие результаты

1. предложен новый метод для построения тестовых данных процессора
2. разработан язык для описания тестовых ситуаций
3. построен прототип генератора тестовых данных

3.1 Метод построения тестовых данных

С помощью предлагаемого метода по формальному или полуформальному описанию поведения операции можно построить код, использующий тестовые данные. При этом будет задействована логическое программирование с ограничениями (constraint logical programming).

1. найти формальное или полуформальное описание поведения операции
2. выделить аргументы операции
3. определить тестовые ситуации, возникающие при выполнении данной операции

4. для каждой тестовой ситуации определить способы ее достижения

5. для каждой тестовой ситуации составить описание на предлагаемом автором данного исследования языке (см. п. 3.2)

6. запустить генератор тестовых ситуаций (см. п. 3); он создаст файл с промежуточным представлением

7. написать программный код на одном из поддерживаемых языков программирования (C, C++, Java), который обрабатывает файл с промежуточным представлением; в результате получится программа (на C, C++ или Java), в которой сначала отыскиваются тестовые данные, а затем используются в других вычислениях

Заметьте, что файл с описанием ситуации достаточно создать один раз (и переписывать его только при смене спецификации тестовой ситуации), запускать генератор тестовых ситуаций для одной тестовой ситуации тоже один раз. Однако для каждого получения новых тестовых данных необходимо запускать логический интерпретатор.

3.2 Язык описания тестовых ситуаций

Язык описания тестовых ситуаций представляет из себя простой императивный язык с единственным типом данных - целым числом, состоящим из заданного числа бит (никаких явных ограничений на число бит не делается), операторами присваивания и утверждения (см. ниже). А также язык включает все операции псевдокода, на котором описаны операции центрального процессора в стандарте [1]:

- получение бита числа с заданным номером (например, `x[7]` – 7й бит числа `x`)
- получение диапазона бит числа с заданными номерами границ этого диапазона (например, `x[8..5]` – диапазон бит с 8го по 5й, включая оба граничных бита)
- конкатенация чисел (например, `x.y` – число, двоичная запись которого сначала состоит из двоичной записи числа `x`, а за ним - из двоичной записи числа `y`)
- битовая степень числа – конкатенация числа с самим собой нужное количество раз (например, `x^5` – битовая степень числа `x`)
- привычные арифметические операции (сложение, вычитание, умножение)
- операции сравнения чисел (на больше, меньше)
- операция знакового увеличения размера числа (например, `x[16->64]` – это 64-х битное число, равное и по модулю, и по знаку 16-ти битному числу, равному `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)

Язык не включает условный оператор и операторы цикла, потому что для `MIPS64` все тестовые ситуации удалось описать без них. Описание ситуации на таком языке представляет собой последовательность операторов, при выполнении которых должна произойти тестовая ситуация.

Язык не включает логическую операцию `NOT`. Это связано с ограничением применяемого логического интерпретатора, работа которого основана на методе резолюций. Тем более, что для `MIPS64` все тестовые ситуации удалось описать без применения `NOT`. Для этого нужно использовать версии вспомогательных функций, используемых в псевдокоде, операторов сравнения, логических операторов, в которые уже внесён оператор `NOT` (например, вместо `NOT(NotWordValue(x))` использовать `WordValue(x)`).

3.3 Генератор тестовых данных

Генератор на входе получает файл с описанием тестовой ситуации, транслирует его в промежуточное представление, исполняет промежуточное представление и, наконец, анализируя результат этого исполнения, формирует значения аргументов операции центрального процессора. В качестве промежуточного представления используется логическая программа, а ее исполнение проводится с помощью логического интерпретатора с открытым кодом ECLiPSe [2]. Выбор именно этого логического интерпретатора обусловлен тем, что он поддерживает технологию CLP [3] – логического программирования с ограничениями. Инструменты, поддерживающие эту технологию, позволяют составлять и находить значения переменных для набора логических выражений (*ограничений*), на которых каждое логическое выражение было бы истинно. Каждый оператор описания тестовой ситуации может быть сведен к набору ограничений. Поиск значений переменных для этого набора ограничений дает как раз нужные значения аргументов операции центрального процессора. Отсутствие в языке описания тестовых ситуаций операторов цикла гарантирует завершение работы инструмента на любом описании тестовой ситуации.

3.4 Пример

Рассмотрим метод на примере операции ADD. В стандарте [1] описание этой операции расположено на странице 36. Сначала выделяем аргументы этой операции. Для этого читаем: «Description: $rd \leftarrow rs + rt$». Значит, аргументы этой операции – rt и rs. Определяемся с тестовыми ситуациями. Читаем псевдокод операции из описания операции в стандарте:

```
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
```

В стандарте [1] возникающее исключение помечается вызовом функции `SignalException`. Одной из тестовых ситуаций будет создание исключения `IntegerOverflow`, т.е. переполнение при сложении. Как должен выполняться этот псевдокод, чтобы произошло переполнение? (т.е. вызвалась функция `SignalException(IntegerOverflow)`) Сначала вычисление выражения «NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])» не должно давать истинный (`true`) результат (иначе поведение не определено, т.к. в псевдокоде написано `UNPREDICTABLE` – значения, при которых поведение не определено, надо исключать). Затем должно выполниться присваивание в *temp* и, наконец, значения 32го и 31го битов этой переменной не должны совпасть. Только при таком исполнении псевдокода вызовется `SignalException(IntegerOverflow)`. Записываем это исполнение на языке, описанном в п. 3.2:

```
VAR INT rs;
VAR INT rt;

ASSERT WordValue(rs) AND WordValue(rt);

LONG temp :=
   rs[31].rs[31..0] + rt[31].rt[31..0];

SITUATION IntegerOverflow WHEN
   temp[32] # temp[31].
```

Как видите, `NOT( NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) )` был заменен на `WordValue(GPR[rs]) or WordValue(GPR[rt])` и далее для сокращения убраны обращения к `GPR`. В итоге получилось `WordValue(rs) AND WordValue(rt)`.

Запускаем генератор (см. п. 3).

Наконец, последний шаг – написание программного кода. Осталось встроить поиск значений переменных rs и rt в другой программный код. Например, на `Java` это можно сделать так:

```
import java.io.IOException;
import java.util.List;
import ru.LesdenSolver;
import ru.LogicalVariable;
```

```
import ru.NoSolution;
import ru.EclipseException;

public class Main {
  /** compile situation file to
        intermediate */
  public List<LogicalVariable> compile(
     String situation, String interm ) {
      return LesdenSolver.compile(
         situation, logicprog );
  }

  /** run generator to get test data
        and use it */
  public void run( String interm,
    List<LogicalVariable> params ) {
      try {
       List<LogicalVariable> parameters =
          LesdenSolver.solve(path, params);
       for(LogicalVariable var: parameters)
       {
          ...
          // name = var.getCanonicalName()
          // value = var.getValue()
       }
      }
      catch( NoSolution t ) {
        ... //no solutions
      }
      catch( IOException e ) {
        System.out.println("I/O: "+e);
        e.printStackTrace();
      }
      catch( EclipseException e ) {
        System.out.println("ECLiPSe: "+e);
        e.printStackTrace();
      }
  }
}
```

4 Продолжение работы

В работе предложен метод систематической генерации тестовых данных для тестирования арифметической подсистемы центральных процессоров MIPS64. Приведен язык описания тестовых ситуаций и необходимый ин-

струментарий для применения этого метода. Построенный метод достаточно технологичен – для его применения достаточно понимать стандарт [1] и уметь переводить его в предложенный язык.

В дальнейшем планируется проанализировать другие стандарты архитектур центральных процессоров для выработки языка описаний тестовых ситуаций, применимых к более широкому классу архитектур центральных процессоров. Под новый язык планируется доработать прототип генератора. Планируется провести более широкую апробацию построенного инструмента.

Список литературы

[1] MIPS64 Architecture For Programmers VolumeII: The MIPS64 Instruction Set. Document Number: MD00087. Revision 2.00. June 9, 2003.

[2] K.Apt, M.Wallace. Constraint Logic Programming using Eclipse. Cambridge University Press, 2007.

[3] K.Marriott, Peter.J.Stuckey. Programming with Constraints. MIT Press, 1998.

[4] Richard D. Eldred: Test Routines Based on Symbolic Logical Statements. J. ACM 6(1): 33-37 (1959)

[5] Davidson and Fraser; The Design and Application of a Retargetable Peephole Optimizer; ToPLaS v2(2) 191-202 (April 1980)

[6] C.Paoli, M.-L. Nivet, J.-F.Santucci, A.Campana. Electronic Design, Test and Applications, 2002. Proceedings. The First IEEE International Workshop on Volume , Issue, 2002 Page(s):382 - 386

[7] F.Fallah, S.Devadas, K.Keutzer. Functional Vector Generation For HDL Models Using Linear Programming and 3-Satisfiability // in Proceedings of the Design Automation Conference, pp. 528-533, June 1998.

[8] F.Corno, A.Manzone, A.Pincetti, M.Sonza Reorda, G.Squillero. Automatic Test Bench Generation for Validation of RT-level Descriptions: an Industrial Experience // DATE2000: Design, Automation and Test in Europe, Paris (F), March 2000, pp. 385-389.

[9] R.Brinkmann, R.Drechsler. RTL-datapath verification using integer linear programming // In IEEE VLSI Design'01 & Asia and South Pacific Design Automation Conference, Bangalore, pages 741–746, 2002.