

# Генерация тестовых данных для арифметической подсистемы центральных процессоров MIPS64

Е.Корныхин, ВМК МГУ

Москва, 2008

## Аннотация

Рассматривается задача генерации тестовых данных для тестирования арифметической подсистемы центральных процессоров MIPS64. Для решения этой задачи предлагается метод, использующий формальное описание поведения процессора. Формальное описание поведения процессора предлагается описывать на языке, допускающем автоматическое решение задачи (язык так же вводится в данной работе).

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.

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

## 1 Постановка задачи

Для правильной работы компьютерных систем необходимо, чтобы правильно работали все компоненты этой системы, в том числе и центральный процессор. 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 – некорректно (например, операция деления должна работать некорректно при подаче в качестве делителя числа ноль). Такое заданное поведение будем далее называть *тестовой ситуацией*. Примеры тестовых ситуаций: «происходит переполнение», «результат сформирован».

Нужно реализовать много различных тестовых

## 2 Существующие методы

Задача тестирования микросхем стала актуальна практически сразу, как появились сами микросхемы. Но сначала такие методы были чисто электротехнические. Настоящие программные методы появились тогда, когда мощность вычислительной техники достигла нужного уровня.

Одна и та же плата может быть представлена различными способами. От способа представления зависит и те алгоритмы, которые можно использовать для нахождения тестовых данных. Первые платы рисовались схемами. Первое упоминание о задаче для таких плат 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 зависимость 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 нужные 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 платы – 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 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 плате, 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 результат за приемлемое время.

## 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

1. найти формальное или полужормальное описание поведения операции
2. выделить аргументы операции
3. определить тестовые ситуации, возникающие при выполнении данной операции
4. для каждой тестовой ситуации определить пути 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

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 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)
- логические операции AND и OR
- оператор присваивания (например, `x := 5;`)
- оператор утверждения (например, `ASSERT x = 5;` – утверждение, что при выполнении данного оператора значение переменной `x` должно равняться 5)

Язык не включает условный оператор и операторы цикла, потому что для MIPS 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 MIPS 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 –  $rd$  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  $\leftarrow$  (GPR[rs]31 || GPR[rs]31..0) +
(GPR[rt]31 || GPR[rt]31..0)
    if temp32  $\neq$  temp31 then
        SignalException(IntegerOverflow)
    else
        GPR[rd]  $\leftarrow$  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 должна давать истинный (`true`) результат (иначе поведение неопределено, т.к. в псевдокоде написано `UNPREDICTABLE` – значения, при которых поведение неопределено, 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]  $\neq$  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 error: "+e);
            e.printStackTrace();
        }
    }
}

```

```

    }
    catch( EclipseException e ) {
        System.out.println("ECLIPSe error: "+e);
        e.printStackTrace();
    }
}
}

```

## 4 Продолжение работы

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

## 5 Заключение

В работе предложен метод систематической генерации тестовых данных для тестирования арифметической подсистемы центральных процессоров MIPS64. Приведен язык описания тестовых ситуаций и необходимый инструментарий для применения этого метода.

## 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.