

# Генерация тестовых данных для системного функционального тестирования FIFO-кэш-памяти микропроцессоров

Е.В. Корныхин

Московский государственный университет им. М.В. Ломоносова

факультет Вычислительной математики и кибернетики

Email: kornevgen@gmail.com

**Аннотация**—В статье рассматривается задача генерации начального состояния кэш-памяти для тестовых шаблонов, используемых в системном функциональном тестировании процессоров. В статье рассматривается генерация тестовых данных для базисных способов организации кэш-памяти: полностью ассоциативный кэш и кэш прямого отображения – а также для кэш-памяти общего вида, сочетающей в себе характеристики первых двух типов кэш-памяти. Генерация начального состояния кэш-памяти осуществляется путем решения ограничений, составленных для тестового шаблона.

## I. ВВЕДЕНИЕ

Микропроцессоры являются одной из важных частей вычислительных систем. Поэтому тестирование микропроцессоров, в том числе и на системном уровне, является актуальной задачей. Системное функциональное тестирование выполняется с использованием большого числа программ на языке ассемблера (тестовых программ). Такие программы загружаются в память машины, запускаются, процесс их исполнения протоколируется и затем анализируется. В результате выносится вердикт, в каких случаях микропроцессор себя ведет некорректно (некорректным считается поведение, отличное от того, что заявлено в спецификации). Однако для проведения системного тестирования для современных микропроцессоров требуется множество тестовых программ, что делает актуальной задачу автоматической генерации тестовых программ.

В статье [2] предложена технологическая цепочка построения тестовых программ на основе модели микропроцессора. В рамках этой цепочки сначала тестовые программы систематически строятся в абстрактном виде (в виде тестового шаблона) — без конкретных параметров инструкций. Тестовые шаблоны описывают последовательность инструкций, параметры инструкций с указанием происходящих событий (например, переполнение, промахи или попадания в кэш-памяти). В качестве параметров инструкции могут быть указаны регистры и константы. Последовательность инструкций чаще всего задана явно. Необходимость в тестовом шаблоне обычно возникает тогда, когда тестирование проводится нацеленным образом и эта цель выражена

последовательностью инструкций, каждая из которых должна быть исполнена заданным образом.

Чтобы получить тестовую программу по заданному тестовому шаблону, достаточно найти начальные значения регистров и той части кэш-памяти и других подсистем, с которыми работают инструкции шаблона. Добавляемые к шаблону данные называются тестовыми данными, а задача их построения – задачей генерации тестовых данных. По тестовым данным строится набор инструкций инициализации микропроцессора (загрузка значений в регистры, кэш и т.д.), который добавляется в начало тестового шаблона. Полученную таким образом тестовую программу можно исполнить и проверить, совпадает ли поведение каждой инструкции с тем, что было заявлено в тестовом шаблоне. Для сложных тестовых шаблонов (сложные зависимости, длинные последовательности инструкций) задача генерации тестовых данных становится отдельной проблемой, которой и посвящена данная статья.

## II. ОБЗОР РАБОТ ПО СИСТЕМНОМУ ФУНКЦИОНАЛЬНОМУ ТЕСТИРОВАНИЮ МИКРОПРОЦЕССОРОВ

В настоящее время в практике системного функционального тестирования микропроцессоров можно выделить следующие подходы к построению тестовых программ:

- *ручная разработка тестовых программ* хоть и практически неприменима для полного тестирования микропроцессора, всё же может применяться для тестирования особых, крайних случаев;
- *тестирование с использованием кросс-компиляции* применяется часто из-за невысокой сложности его проведения: после согласования спецификации микропроцессора можно начинать делать кросс-компилятор, а код, предназначенный для кросс-компиляции, уже готов. Однако гарантировать полноту такое тестирование не может;
- *случайная генерация тестовых программ* применяется так же часто в силу простоты автоматизации. Сгенерированные таким образом тестовые

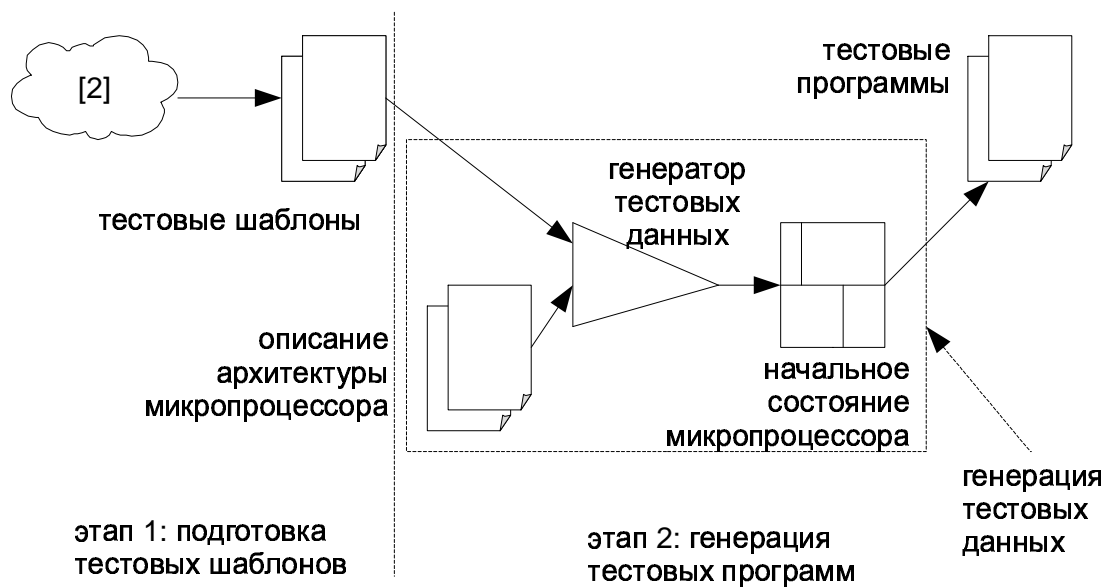


Рис. 1. Генерация тестовых программ на основе тестовых шаблонов

программы позволяют быстро обнаружить простые ошибки, однако не гарантируют полноты тестирования. Разрабатываются и более сложные варианты случайной генерации [4];

- *случайная генерация тестовых программ на основе тестовых шаблонов* предполагает разделение процесса генерации тестовой программы на два этапа (см. рис. 1): на первом подготавливаются тестовые шаблоны – абстрактные представления тестовых программ (в тестовых шаблонах для параметров инструкций вместо значений указываются ограничения на значения) – а на втором этапе по тестовым шаблонам генерируются тестовые программы. Второй этап включает в себя *генерацию тестовых данных*, т.е. генерацию параметров инструкций (параметров-констант) и начальных значений регистров, ячеек кэш-памяти, строк TLB и т.д. Иногда выбор регистров для инструкций задается в тестовом шаблоне, а иногда выбор регистров может сделать генератор тестовых данных.

Тестовые шаблоны описывают последовательность инструкций, параметры инструкций с указанием происходящих событий (например, переполнение, промахи или попадания в кэш-памяти). В качестве параметров инструкции могут быть как явно указаны регистры и константы, так и предоставлена возможность инструменту самому подобрать регистры. Последовательность инструкций чаще всего задана явно. Необходимость в тестовом шаблоне обычно возникает тогда, когда тестирование проводится нацеленным образом и эта цель выражена последовательностью инструкций, каждая из которых должна быть исполнена заданным образом. Пример тестового шаблона для модельной системы ко-манд:

```
REGISTER reg1:32;
REGISTER reg2:32;
REGISTER reg3:32;
ADD reg1, reg1, reg2 @ overflow
LOAD reg3, reg2, 0 @ hit
MUL reg1, reg2, reg3 @ normal
```

В этом тестовом шаблоне 3 инструкции – ADD, LOAD и MUL. У каждой инструкции указаны параметры – или 3 регистра, или 2 регистра и константа – и информация о том, как должна быть исполнена инструкция: с переполнением (overflow), с кэш-попаданием (hit) или без исключений (normal). Чтобы получить тестовую программу по этому шаблону, достаточно задать начальные значения регистров reg1, reg2 и reg3 и той части кэш-памяти, с которой работает вторая инструкция (LOAD) (это и будут тестовые данные для данного шаблона). Это можно сделать, добавив в начало тестового шаблона инструкции инициализации состояния микропроцессора. Полученную тестовую программу можно исполнить и проверить, совпадает ли поведение каждой инструкции с тем, что было заявлено в тестовом шаблоне.

В данной статье нас будут интересовать лишь инструкции работы с памятью, коих две:

- «LOAD reg, address» осуществляет загрузку значения в переменную reg из памяти по адресу в переменной address;
- «STORE reg, address» осуществляет сохранение значения из переменной reg в памяти по адресу в переменной address.

Обратимся к задаче генерации тестовых данных. Среди известных работ можно выделить следующие методы ее решения:

- комбинаторные техники;

- решение задачи ATPG;
- разрешение ограничений.

*Комбинаторные техники* применимы в случае простых тестовых шаблонов. Такие тестовые шаблоны включают лишь простые ограничения, а именно указание области значений переменной. Причем все значения этой области в тестовой программе равноправны. Техника хоть и простая, но довольно ограниченная в применении, поскольку не всегда получается привести ограничения на переменные к такому простому виду. В работе исследователей из Fujitsu Lab. [5] предлагается описать тестовые программы в виде выражений (Test Specification Expressions, TSE), а инструкции микропроцессора – на языке ISDL. Специальный генератор строит тестовые программы, удовлетворяющие TSE. Kohno и Matsumoto [6] рассматривают задачу верификации конвейерных микропроцессоров, используя для этого генерацию тестовых программ с помощью тестовых шаблонов. Области значений переменных в таких шаблонах складываются из регистров и числовых констант.

Исследователи из Politecnico di Milano [7] предложили генерировать тестовые данные с использованием *техник решения задачи ATPG* (Automatic Test Pattern Generation). ATPG – задача поиска значений входных сигналов («векторов») схемы с целью поиска ее некорректного поведения. ATPG чаще применяется для модульного тестирования, если известна RTL-модель микропроцессора. Задача ATPG известна давно и для ее решения существуют (в том числе коммерческие) инструменты. Для применения ATPG при генерации тестовых программ необходимо, чтобы RTL-модель микропроцессора была готова к моменту генерации тестовых данных. Кроме того, использование такой методики именно для функционального тестирования ограничено, поскольку тесты на функционирование микропроцессора приходится строить с учетом модели спроектированного микропроцессора, которая сама же при этом будет и тестироваться.

Наиболее впечатляющих результатов достигают инструменты, использующие для генерации тестовых данных *разрешение ограничений*. Ограничение с логической точки зрения то же, что и предикат, а задача разрешения ограничений – то же, что и задача выполнимости системы предикатов, но для решения этой задачи применяются специальные алгоритмы [1]. В работе [9] исследователей из Китайского Национального Университета технологий безопасности описывается инструмент МААТГ. Тестовый шаблон для него может содержать лишь ограничения равенства или неравенства значений и указание области значений переменной. Для задания архитектуры микропроцессора используется описание на языке EXPRESSION. Другой инструмент – Genesys-Pro [8] – позиционируется компанией IBM как разработка, впитавшая лучшее из разработок последних 20 лет. Тестовые шаблоны позволяют задавать тестовые программы переменной длины. Для любой инструкции

в тестовом шаблоне может быть указана эвристика для выбора значений параметров [3]. Среди возможных эвристик есть и эвристики на события в кэш-памяти. Однако в известных работах не раскрывается содержание таких эвристик, что не дает возможности понять эффективность генерации программ, нацеленных на тестирование памяти. Система команд микропроцессора должна быть описана в виде ограничений (constraint net) на операнды, код операции, что не является естественным описанием поведения инструкции, особенно если в рамках нее выполняется несколько последовательных вычислений на основе параметров инструкции. Для генерации параметров очередной инструкции Genesys-Pro использует уже построенную тестовую программу и состояние микропроцессора, которое известно полностью. Этот подход обеспечил масштабируемость на большие тестовые шаблоны, но и привел к необходимости использования механизма возврата (backtracking), если выбрать параметры для очередной инструкции.

В данной работе при решении задачи генерации тестовых данных также используется разрешение ограничений. В отличие от МААТГ тестовые шаблоны могут содержать не просто ограничения равенства или неравенства, а более сложные ограничения, например, кэш-промах. А по сравнению с Genesys-Pro в данной статье делается попытка транслировать тестовый шаблон в ограничения целиком <sup>1</sup>. При этом отпадает необходимость в механизме возврата <sup>2</sup>. Особенностью тестовых шаблонов, получаемых в рамках [2], является фиксация для каждой инструкции регистров-параметров. Для таких шаблонов (особенно если в них много зависимостей) Genesys-Pro будет работать крайне неэффективно, поскольку теряется возможность с помощью выбора параметров «подогнать» исполнение очередной инструкции под заданные в тестовом шаблоне для нее события. На тестовых шаблонах из [2] Genesys-Pro будет работать следующим образом: выберет некоторое начальное состояние микропроцессора, начнет исполнять тестовый шаблон (поскольку начальное состояние ему известно), но как только дойдет до инструкции, которая будет исполнена не так, как требуется в шаблоне, Genesys-Pro сделает возврат в самое начало, а именно ему придется выбрать другое начальное состояние микропроцессора и весь процесс запустить заново. Такой процесс генерации тестовых данных слишком неэффективен. Кроме того попытка наивного переноса идей из представленных в

<sup>1</sup>Известно, что задача разрешения ограничений (т.е. задача выполнимости) NP-полна. Это означает, что для больших тестовых шаблонов предлагаемый в данной статье метод может быть не столь эффективным. Однако действительно длинные тестовые шаблоны в практике тестирования микропроцессоров применяются редко.

<sup>2</sup>Из-за этого качественно меняется разрешаемая система ограничений. Genesys-Pro сводит общую задачу к множеству задач, на порядок меньшей сложности. Кроме того, в данной статье предлагается более технологичный метод построения тестовых данных: описание архитектуры микропроцессора может быть получено из стандарта архитектуры микропроцессора и представляет собой понятное для человека императивное задание.

обзоре инструментов (кодирование изменений состояния каждого регистра и зависимостей между ними в виде ограничений) для инструкций работы с памятью приводит к очень сложным ограничениям<sup>3</sup>, которые не удается разрешить за приемлемое время.

### III. ГЕНЕРАЦИЯ ТЕСТОВЫХ ДАННЫХ ДЛЯ ПОЛНОСТЬЮ АССОЦИАТИВНОЙ КЭШ-ПАМЯТИ

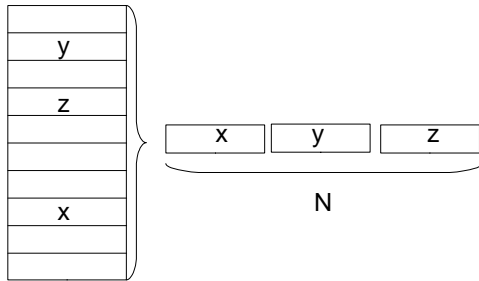


Рис. 2. Полностью N-ассоциативный кэш

Полностью ассоциативный кэш состоит из заданного количества ячеек (их количество называется *ассоциативностью кэша*). В каждой ячейке кэша могут храниться данные любых ячеек памяти. Все ячейки кэша соответствуют разным ячейкам памяти. Все ячейки кэш-памяти равноправны. При обращении к памяти первым делом данные ищутся в кэше во всех ячейках кэш-памяти *параллельно*. Ситуация *кэш-попадания* означает, что одна из ячеек кэш-памяти соответствует требуемому адресу. Ситуация *кэш-промаха* означает, что ни одна из ячеек кэш-памяти не соответствует требуемому адресу. В случае кэш-промаха одна из ячеек кэш-памяти *вытесняется* и заменяется на данные по требуемому адресу. Согласно стратегии вытеснения FIFO (First-In First-Out) будут вытеснены данные, которые находятся в кэше дольше всего. Везде далее под фразой «вытесняемый адрес  $x$ » будут пониматься вытесняемые данные по адресу  $x$ .

Предлагаемый алгоритм построения ограничений для тестового шаблона в случае полностью ассоциативной кэш-памяти основывается на следующих свойствах вытесняемых адресов:

- 1) вытесняемый адрес был добавлен ранее одной из инструкций тестового шаблона (или находился среди адресов начального состояния кэш-памяти);

<sup>3</sup> Для кодирования состояния микропроцессора можно использовать формулу длиной порядка размера памяти ( $mem_0 = var_0 \wedge mem_1 = var_1 \wedge \dots$ ); каждое изменение производится по неизвестному индексу, поэтому при записи нового состояния микропроцессора приходится перебирать все возможные варианты ( $mem[i] := x \wedge mem_1 = var_1 \wedge \dots \vee (i = 1 \wedge mem_0 = var_0 \wedge mem_1 = x \wedge \dots) \vee \dots$ ), а если таких изменений несколько, то приходится рассматривать все возможные варианты значений индексов. Получающаяся формула имеет размер порядка  $|L| \cdot 2^n$ , где  $|L|$  – размер памяти, а  $n$  – количество изменений памяти. В данной работе предложен метод кодирования изменений, приводящий к формуле размера порядка  $|L| + n$ .

- 2) между вытеснением адреса и кэш-промахом этого адреса происходят кэш-промахи всех остальных адресов, содержащихся в кэше на момент вытеснения.

Алгоритм строит ограничения на следующие переменные:

- 1)  $\alpha_1, \alpha_2, \alpha_3, \dots$  – адреса начального состояния кэш-памяти (их количество равно ассоциативности кэш-памяти);
- 2) адреса, при обращении к которым происходят кэш-попадания (их количество равно количеству инструкций, при обращении к которым происходят кэш-попадания);
- 3) адреса, при обращении к которым происходят кэш-промахи (их количество равно количеству инструкций, при обращении к которым происходят кэш-промахи);
- 4)  $L_0, L_1, \dots$  – переменные-состояния кэш-памяти (их количество – на единицу больше количества инструкций, при обращении к которым происходят кэш-промахи).

Итак, каждая инструкция, при обращении к которой происходит кэш-попадание, порождает 1 новую переменную; каждая инструкция, при обращении к которой происходит кэш-промах порождает 1 переменную-состояние кэш-памяти и 1 переменную-вытесняющий адрес. Алгоритм формирует ограничения для каждой очередной инструкции следующим образом ( $N$  – ассоциативность кэш-памяти):

- 1) «начальные ограничения» генерируются для любого шаблона один раз:  $L_0 = \{\alpha_1, \alpha_2, \dots, \alpha_N\}$ ,  $|L_0| = N$  (или, по-другому, все числа  $\alpha_1, \alpha_2, \dots, \alpha_N$  разные);
- 2) «ограничения кэш-попадания» генерируются для каждой инструкции, при обращении к которой происходит кэш-попадание:  $x \in L$ , где  $x$  – адрес в инструкции,  $L$  – текущая переменная-состояние кэш-памяти;
- 3) «ограничения кэш-промаха» генерируются для каждой инструкции, при обращении к которой происходит кэш-промах ( $x$  – вытесняющий адрес,  $L$  – текущая переменная-состояние кэш-памяти):  $x \notin L, L' = L \cup \{x\} \setminus \{y\}$ , где  $y$  – вытесняемый адрес – это адрес, к которому происходит кэш-промах в инструкции, находящейся перед данной инструкцией на  $N$  инструкций (в этом выражается стратегия вытеснения). Если перед данной инструкцией меньше  $N$  инструкций, то в порядке помещения адресов в начальное состояние выбирается соответствующий адрес начального состояния.  $L'$  становится текущей переменной-состояния кэш-памяти для следующей инструкции.

В качестве «кэш-попадания адреса  $x$ » рассматриваются также «кэш-промахи адреса  $x$ », т.к. в результате кэш-промаха данные по этому адресу подгружаются в

кэш, и адреса данных, лежащих перед исполнением тестового шаблона в кэше (в порядке «последнего к ним обращения»).

Рассмотрим пример тестового шаблона и построение для него тестовых данных. Будем рассматривать его для 3-х ассоциативного кэша.

LOAD x, y @ Hit  
STORE u, z @ Miss  
LOAD z, y @ Hit

Первым делом определимся с именами переменных так, чтобы они не меняли свое значение (введем «версии» переменных). Учтем, что LOAD дает новую версию переменной, идущей первым аргументом (т.к. в него загружается значение из памяти):

LOAD  $x_1, y_0$  @ Hit  
STORE  $u_0, z_0$  @ Miss  
LOAD  $z_1, y_0$  @ Hit

Введем переменные начального состояния кэш-памяти:  $\{\alpha, \beta, \gamma\}$  (их количество равно ассоциативности кэш-памяти).

Наша задача состоит в поиске значений  $x_0, y_0, z_0, u_0, \alpha, \beta, \gamma$ , на которых инструкции тестового шаблона будут исполнены в заданных тестовых ситуациях. Логично предположить, что решение не будет единственным. Нам в данном случае достаточно найти какое-либо одно решение.

Первые ограничения связаны с описанием попаданий и промахов, как принадлежности текущей переменной-состоянию кэш-памяти:

$y_0 \in \{\alpha, \beta, \gamma\}$ ,  
 $z_0 \notin \{\alpha, \beta, \gamma\}$ ,  
 $y_0 \in \{\alpha, \beta, \gamma\} \setminus \{\alpha\} \cup \{z_0\}$ ,  
 $\alpha, \beta, \gamma$  – все разные

Упрощаем полученную систему ограничений:

$y_0 \in (\{\alpha, \beta, \gamma\} \cap \{\gamma, \beta, z_0\})$   
 $z_0 \notin \{\alpha, \beta, \gamma\}$ ,  
 $\alpha, \beta, \gamma$  – все разные

Что упрощается до такой системы ограничений:

$y_0 \in \{\gamma, \beta\}$   
 $z_0 \notin \{\alpha, \beta, \gamma\}$ ,  
 $\alpha, \beta, \gamma$  – все разные

Заметьте, что  $x_0$  и  $u_0$  нигде не принимают участие – их значение может быть произвольным.

Пусть адреса занимают 8 бит. Тогда все переменные принимают значения из области от 0 до 255. Разрешая полученные ограничения, можно получить такие значения для переменных (замечу снова, что набор значений не является единственным):

$\gamma = y_0 = x_0 = u_0 = 0$   
 $\beta = 1$   
 $\alpha = 2$   
 $z_0 = 3$

Проверим исполнение тестового шаблона с такими значениями переменных:

начальные значения кэша: [0, 1, 2]  
LOAD x, 0 - Hit, т.к.  $0 \in \{0, 1, 2\}$ ;

STORE 0, 3 - Miss, т.к.  $3 \notin \{0, 1, 2\}$ ; согласно FIFO 3 попадает в кэш-память, 2 из кэша вытесняется, новое состояние кэш-памяти равно [3, 0, 1]

LOAD z, 0 - Hit, т.к.  $0 \in \{3, 0, 1\}$

Все инструкции тестового шаблона были исполнены согласно указанным тестовым ситуациям.

#### IV. ГЕНЕРАЦИЯ ТЕСТОВЫХ ДАННЫХ ДЛЯ КЭШ-ПАМЯТИ ПРЯМОГО ОТОБРАЖЕНИЯ

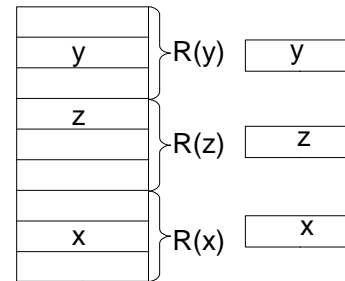


Рис. 3. Кэш прямого отображения

Вся память поделена на непересекающиеся части (будем называть их *регионами*). Каждому региону соответствуют 1 ячейка кэша. Других ячеек в кэше нет. В каждой ячейке кэша могут храниться данные только своего региона. При обращении к памяти по адресу первым делом данные ищутся в (единственной) ячейке кэша своего региона. Ситуация *кэш-попадания* означает, что в этой ячейке кэш-памяти находятся данные именно по требуемому адресу. Ситуация *кэш-промаха* означает, что в ячейке кэш-памяти требуемого региона содержатся данные по адресу отличному от требуемого. В случае кэш-промаха данные из ячейки кэш-памяти *вытесняются* и заменяются на данные по требуемому адресу. Поскольку вытесняемые данные определены однозначно, о стратегиях вытеснения для кэш-памяти прямого отображения не говорят.

Предлагаемый алгоритм построения ограничений для тестового шаблона в случае кэш-памяти прямого отображения основывается на следующих свойствах вытесняемых адресов:

- 1) любой (в том числе и вытесняемый) адрес был добавлен ранее одной из инструкций тестового шаблона (или находился среди адресов начального состояния кэш-памяти);
- 2) между вытеснением адреса и кэш-промахом этого адреса происходят кэш-промахи всех остальных адресов, содержащихся в кэше на момент вытеснения.

Алгоритм строит ограничения на следующие переменные:

- 1)  $\alpha_1, \alpha_2, \alpha_3, \dots$  – адреса начального состояния кэш-памяти (их количество равно количеству регионов);

- 2) адреса, при обращении к которым происходят кэш-попадания (их количество равно количеству инструкций, при обращении к которым происходят кэш-попадания);
- 3) адреса, при обращении к которым происходят кэш-промахи (их количество равно количеству инструкций, при обращении к которым происходят кэш-промахи);
- 4) вытесняемые адреса (их количество равно количеству инструкций, при обращении к которым происходят кэш-промахи);
- 5)  $L_0, L_1, \dots$  – переменные-состояния кэш-памяти (их количество – на единицу больше количества инструкций, при обращении к которым происходят кэш-промахи).

Введем функциональный символ  $R(y)$ , который для адреса  $y$  возвращает множество всех ячеек того же региона, что и  $y$ . Для этого функционального символа справедливы следующие свойства:

$$\begin{aligned} \forall x (x \in R(x)) \\ \forall x \forall y (x = y \rightarrow R(x) = R(y)) \\ \forall x \forall y (R(x) = R(y) \leftrightarrow x \in R(y)) \\ \forall x \forall y (R(x) = R(y) \leftrightarrow y \in R(x)) \\ \forall x \forall y (x \notin R(y) \rightarrow x \neq y) \end{aligned}$$

Алгоритм формирует ограничения для каждой очередной инструкции следующим образом ( $N$  – количество регионов):

- 1) «начальные ограничения» генерируются для любого шаблона один раз:  $|\{\alpha_1, \alpha_2, \dots, \alpha_N\}| = N$  (или, по-другому, все числа  $\alpha_1, \alpha_2, \dots, \alpha_N$  разные),  $|\{R(\alpha_1), R(\alpha_2), \dots, R(\alpha_N)\}| = N$  (или, по-другому, все множества  $R(\alpha_1), R(\alpha_2), \dots, R(\alpha_N)$  разные);
- 2) «ограничения кэш-попадания» генерируются для каждой инструкции, при обращении к которой происходит кэш-попадание:  $x \in L$ , где  $x$  – адрес в инструкции,  $L$  – текущая переменная-состояние кэш-памяти;
- 3) «ограничения кэш-промаха» генерируются для каждой инструкции, при обращении к которой происходит кэш-промах ( $x$  – вытесняющий адрес,  $y$  – вытесняемый адрес,  $L$  – текущая переменная-состояние кэш-памяти):  $y \in L, x \notin L, L' = L \cup \{x\} \setminus \{y\}, R(y) = R(x)$ ,  $L'$  становится текущей переменной-состояния кэш-памяти для следующей инструкции.

Рассмотрим тот же пример тестового шаблона и построение для него тестовых данных. Пусть память поделена на 3 региона в зависимости от остатка от деления на 3 адреса (т.е.  $R(x) = R(y) \Leftrightarrow 3|(x - y)$ ).

LOAD  $x, y$  @ Hit  
STORE  $u, z$  @ Miss  
LOAD  $z, y$  @ Hit

Первым делом определимся с именами переменных так, чтобы они не меняли свое значение (введем «версии» переменных). Учтем, что LOAD дает новую версию

переменной, идущей первым аргументом (т.к. в него загружается значение из памяти). Для описания вытесняемого адреса введем фиктивную переменную  $z'_0$  (в ответ ее значение не попадет):

LOAD  $x_1, y_0$  @ Hit  
STORE  $u_0, z_0$  @ Miss  $\rightarrow z'_0$   
LOAD  $z_1, y_0$  @ Hit

Введем переменные начального состояния кэш-памяти:  $\{\alpha, \beta, \gamma\}$  (по одной для каждого региона).

Наша задача состоит в поиске значений  $x_0, y_0, z_0, u_0, \alpha, \beta, \gamma$ , на которых инструкции тестового шаблона будут исполнены в заданных тестовых ситуациях. Логично предположить, что решение не будет единственным. Нам в данном случае достаточно найти какое-либо одно решение.

Согласно алгоритму для кэш-попаданий и кэш-промахов будут выделены следующие ограничения:

$$\begin{aligned} y_0 \in \{\alpha, \beta, \gamma\}, \\ z_0 \notin \{\alpha, \beta, \gamma\}, \\ z'_0 \in \{\alpha, \beta, \gamma\}, \\ y_0 \in \{\alpha, \beta, \gamma\} \setminus \{z'_0\} \cup \{z_0\}, \\ R(z_0) = R(z'_0), \\ \alpha, \beta, \gamma - \text{все разные} \\ R(\alpha), R(\beta), R(\gamma) - \text{все разные} \end{aligned}$$

Упростим полученную систему ограничений:

$$\begin{aligned} z'_0 \in \{\alpha, \beta, \gamma\}, \\ y_0 \in \{\alpha, \beta, \gamma\} \setminus \{z'_0\}, \\ z_0 \notin \{\alpha, \beta, \gamma\}, \\ 3|(z_0 - z'_0), \\ \alpha, \beta, \gamma - \text{все разные} \\ R(\alpha), R(\beta), R(\gamma) - \text{все разные} \end{aligned}$$

Заметьте, что  $x_0$  и  $u_0$  нигде не принимают участие – их значение может быть произвольным.

Пусть адреса занимают 8 бит. Тогда все переменные принимают значения из области от 0 до 255. Разрешая полученные ограничения, можно получить такие значения для переменных (замечу снова, что набор значений не является единственным):

$$\begin{aligned} \alpha &= x_0 = u_0 = 0 \\ \beta &= y_0 = 1 \\ \gamma &= 2 \\ z_0 &= 3 \end{aligned}$$

Проверим исполнение тестового шаблона с такими значениями переменных:

начальные значения кэша:  $L = [(R = 0) \mapsto 0, (R = 1) \mapsto 1, (R = 2) \mapsto 2]$

LOAD  $x, 1$  - Hit, т.к.  $R(1) = L[R = (1 \bmod 3)]$

STORE  $0, 3$  - Miss, т.к.  $R(3) \neq L[R = (3 \bmod 3)]$ , 1 из кэша вытесняется, новое ее состояние равно  $L = [(R = 0) \mapsto 0, (R = 1) \mapsto 3, (R = 2) \mapsto 2]$

LOAD  $z, 0$  - Hit, т.к.  $R(0) = L[R = (0 \bmod 3)]$

Все инструкции тестового шаблона были исполнены согласно указанным тестовым ситуациям.

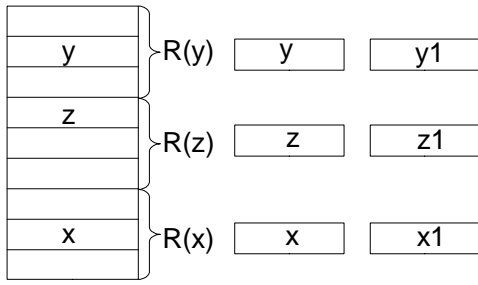


Рис. 4. Кэш общего вида

## V. ОБЩИЙ СЛУЧАЙ

Предлагаемый алгоритм построения ограничений для тестового шаблона основывается на следующих свойствах вытесняемых адресов:

- 1) вытесняемый адрес был добавлен ранее одной из инструкций тестового шаблона (или находился среди адресов начального состояния кэш-памяти);
- 2) между вытеснением адреса и кэш-промахом к нему происходят кэш-попадания ко всем остальным адресам кэша в данном регионе (в этом смысле FIFO немного напоминает другую стратегию вытеснения – LRU, Least Recently Used).

Алгоритм строит ограничения на следующие переменные:

- 1)  $\alpha_1, \alpha_2, \alpha_3, \dots$  – адреса начального состояния кэш-памяти (их количество равно ассоциативности кэш-памяти);
- 2) адреса, при обращении к которым происходят кэш-попадания (их количество равно количеству инструкций, при обращении к которым происходят кэш-попадания);
- 3) адреса, при обращении к которым происходят кэш-промахи (их количество равно количеству инструкций, при обращении к которым происходят кэш-промахи);
- 4)  $L_0, L_1, \dots$  – переменные-состояния кэш-памяти (их количество – на единицу больше количества инструкций, при обращении к которым происходят кэш-промахи).

Для построения ограничений кэш-памяти общего вида так же будет использоваться функциональный символ  $R(x)$ , причем ровно в том же смысле, какой он имел для кэш-памяти прямого отображения.

Итак, каждая инструкция, при обращении к которой происходит кэш-попадание, порождает 1 новую переменную; каждая инструкция, при обращении к которой происходит кэш-промах порождает 1 переменную-вытесняющий адрес и 1 переменную-вытесняемый адрес. Алгоритм формирует ограничения для каждой очередной инструкции следующим образом ( $N$  – ассоциативность кэш-памяти):

- 1) «начальные ограничения» генерируются для любого шаблона один раз:  $L_0 = \{\alpha_1, \alpha_2, \dots, \alpha_N\}$ ,

$|L_0| = N$  (или, по-другому, все числа  $\alpha_1, \alpha_2, \dots, \alpha_N$  разные);

- 2) «ограничения кэш-попадания» генерируются для каждой инструкции, при обращении к которой происходит кэш-попадание:  $x \in L$ , где  $x$  – адрес в инструкции,  $L$  – текущая переменная-состояние кэш-памяти;
- 3) «ограничения кэш-промаха» генерируются для каждой инструкции, при обращении к которой происходит кэш-промах ( $x$  – вытесняющий адрес,  $y$  – вытесняемый адрес,  $L$  – текущая переменная-состояние кэш-памяти):  $y \in L, x \notin L, L' = L \cup \{x\} \setminus \{y\}, R(x) = R(y), \text{fifo}(y)$ ,  $L'$  становится текущей переменной-состояния кэш-памяти для следующей инструкции.

Ограничение  $\text{fifo}(y)$  описывает свойство, что  $y$  является вытесненным адресом.

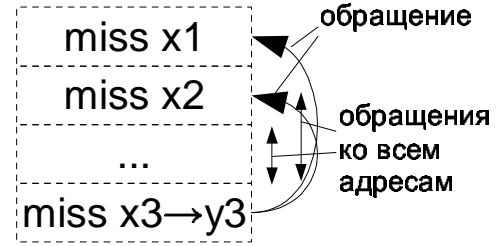


Рис. 5.  $\text{fifo}(y3)$

Ограничение  $\text{fifo}(y)$  представляется дизъюнкцией ограничений, соответствующих всевозможным местам кэш-промаха с вытесняющим адресом  $y$ . Каждый дизъюнкт к кэш-промаху адреса  $x$  одной из предыдущих инструкций (или к адресу из начального состояния кэша, если все предыдущие инструкции не подходят) фиксирует конъюнкцию двух свойств:

- 1)  $x = y$
- 2)  $(\{x_1, x_2, \dots, x_n\} \cap R(y) = (L \setminus \{y\}) \cap R(y)$ , где  $x_1, x_2, \dots, x_n$  – все адреса, к которым происходят обращения с кэш-промахами между обращением к  $x$  и вытеснением  $y$ .

Последнее ограничение можно упростить, используя следующую лемму:

**Лемма 1:** Для любых конечных множеств  $X, Y$  и  $Z$  если существует  $y$  такой, что  $y \in Y \cap Z$  и  $y \notin X$ , то  $X \cap Y = (Z \setminus \{y\}) \cap Y \Leftrightarrow Y \cap (Z \setminus X) = \{y\}$ .

**Доказательство:**  $X \cap Y = (Z \setminus \{y\}) \cap Y \Leftrightarrow X \cap Y = Z \cap Y \cap \{y\}$ .

Обозначим  $A = Z \cap Y, B = X \cap Y$ . По условию  $y \notin B$  и  $y \in A$ . Кроме того  $B = A \setminus \{y\}$ . Значит,  $A = B \sqcup \{y\}$ , откуда  $A \setminus B = \{y\}$ . Осталось показать, что  $A \setminus B = (Z \setminus X) \cap Y$ :  $A \setminus B = A \cap \bar{B} = Z \cap Y \cap \bar{X} \cap \bar{Y} = Z \cap Y \cap (\bar{X} \cup \bar{Y}) = (Z \cap Y \cap \bar{X}) \cup (Z \cap Y \cap \bar{Y}) = Z \cap \bar{X} \cap Y = (Z \setminus X) \cap Y$ . ■

Таким образом, ограничение  $\text{fifo}(y)$  представляется дизъюнкцией по предыдущим вытесняющим адресам и адресам начального состояния кэша  $x$  конъюнкций вида:

$$x = y$$

$\wedge$

$R(y) \cap (L \setminus \{x_1, x_2, \dots, x_n\}) = \{y\}$ , где  $x_1, x_2, \dots, x_n$  – все адреса, к которым происходят обращения с кэш-промахами между обращением к  $x$  и вытеснением  $y$ .

Рассмотрим тот же тестовый шаблон для памяти из 3-х регионов ( $R(x) = R(y) \leftrightarrow 3|(x - y)$ ) и 2-х ассоциативной кэш-памяти:

LOAD  $x, y$  @ Hit  
STORE  $u, z$  @ Miss  
LOAD  $z, y$  @ Hit

Вводим аналогично двум предыдущим случаям версии переменных и фиктивную переменную  $z'_0$ :

LOAD  $x_1, y_0$  @ Hit  
STORE  $u_0, z_0$  @ Miss  $\rightarrow z'_0$   
LOAD  $z_1, y_0$  @ Hit

Введем переменные для начального состояния кэша:  $\alpha_1, \alpha_2$  для первого региона,  $\beta_1, \beta_2$  для второго региона,  $\gamma_1, \gamma_2$  для третьего. Сама система ограничений будет иметь следующий вид:

$y_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\}$ ,  
 $z'_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\}$ ,  
 $z_0 \notin \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \cap R(z'_0)$ ,  
 $y_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \cup \{z_0\} \setminus \{z'_0\}$ ,  
 $R(z_0) = R(z'_0)$ ,  
 $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2$  – все разные,  
 $R(\alpha_1) = R(\alpha_2)$ ,  
 $R(\beta_1) = R(\beta_2)$ ,  
 $R(\gamma_1) = R(\gamma_2)$ ,  
 $R(\alpha_1), R(\beta_1), R(\gamma_1)$  – все разные

И дизъюнкция, описывающая  $fifo(z'_0)$  (достаточно одного дизъюнкта для получения решения):

$$z'_0 = \gamma_1 \wedge R(z'_0) \cap (\{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \setminus \{\gamma_2\}) = \{z'_0\}$$

$\vee$

...

Упрощаем полученную систему ограничений:

$y_0 \in \{\alpha_1, \dots, \gamma_2\}$ ,  
 $z'_0 \in \{\alpha_1, \dots, \gamma_2\}$ ,  
 $z_0 \notin \{\alpha_1, \dots, \gamma_2\} \cap R(z'_0)$ ,  
 $y_0 \in \{\alpha_1, \dots, \gamma_2, z_0\} \setminus \{z'_0\}$ ,  
 $R(z_0) = R(z'_0)$ ,  
 $z'_0 = \gamma_1$ ,  
 $R(\gamma_1) \cap \{\gamma_1\} = \{\gamma_1\}$

$\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2$  – все разные,  
 $R(\alpha_1) = R(\alpha_2)$ ,  
 $R(\beta_1) = R(\beta_2)$ ,  
 $R(\gamma_1) = R(\gamma_2)$ ,  
 $R(\alpha_1), R(\beta_1), R(\gamma_1)$  – все разные

И окончательное упрощение:

$z'_0 = \gamma_1$ ,  
 $y_0 = \gamma_2$ ,  
 $z_0 \notin \{\gamma_1, \gamma_2\}$ ,  
 $R(z_0) = R(\gamma_2)$ ,  
 $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2$  – все разные,  
 $R(\alpha_1) = R(\alpha_2)$ ,  
 $R(\beta_1) = R(\beta_2)$ ,

$$R(\gamma_1) = R(\gamma_2),$$

$R(\alpha_1), R(\beta_1), R(\gamma_1)$  – все разные

Получено решение с минимальным количеством задействованных регионов (задействован только регион, в котором находятся адреса  $\gamma_1$  и  $\gamma_2$ ).

Разрешая эти ограничения для 8-битных адресов, можно получить такое решение:

$$\begin{aligned} \alpha_1 &= 0, \alpha_2 = 3, \\ \beta_1 &= 1, \beta_2 = 4, \\ \gamma_1 &= 2, \gamma_2 = 5, \\ x_0 &= 0, y_0 = 5, z_0 = 8, u_0 = 0. \end{aligned}$$

В общем случае для символьного решения подобного рода систем ограничений (с операциями над множествами) могут применяться специальные алгоритмы разрешения на основе того, что все множества конечные (достаточно рассматривать  $R(x)$  как множество лишь тех адресов, к которым обращаются инструкции тестового шаблона).

## VI. ЗАКЛЮЧЕНИЕ

В статье рассматривалась задача генерации начального состояния кэш-памяти для тестовых шаблонов. В статье предложен алгоритм, строящий систему ограничений на конечные множества адресов. Результатом решения такой системы являются начальные значения регистров и ячеек кэш-памяти. В статье рассмотрено построение ограничений для полностью ассоциативной кэш-памяти, кэш-памяти прямого отображения и кэш-памяти общего вида.

## СПИСОК ЛИТЕРАТУРЫ

- [1] Семенов А.Л. Методы распространения ограничений: основные концепции. *PSI'03/ИМРО — Интервальная математика и методы распространения ограничений*, 2003.
- [2] Камкин А.С. Генерация тестовых программ для микропроцессоров. *Труды ИСП РАН*, 14(2):23–64, 2008.
- [3] L.Fournier E.Marcus M.Rimon M.Vinov A.Ziv A.Adir, E.Almog. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design and Test of Computers*, 21(2):84–93, Mar/Apr 2004.
- [4] M. Sonza Reorda G. Squillero F. Corno, E. Sanchez. Automatic test program generation – a case study. *IEEE Design & Test, Special issue on Functional Verification and Testbench Generation*, 21(2):102–109, MArch-April 2001.
- [5] K.Takayama F.Fallah. A new functional test program generation methodology. *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 76–81, 2001.
- [6] N.Matsumoto K.Kohno. A new verification methodology for complex pipeline behavior. *Proceedings of the 38st Design Automation Conference (DAC'01)*, 2001.
- [7] F.Ferrandi D.Sciuto M.Beardo, F.Bruschi. An approach to functional testing of vliw architectures. *Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, pages 29–33, 2000.
- [8] Y.Lichtenstein M.Rimon M.Vinov M.Behm, J.Ludden. Industrial experience with test generation languages for processor verification. *Proceedings of the 41st Design Automation Conference (DAC'04)*, 2004.
- [9] Y.Guo G.Liu S.Li T.Li, D.Zhu. Maatg: A functional test program generator for microprocessor verification. *Proceedings of the 2005 8th Euromicro conference on Digital System Design (DSD'05)*, 2005.