Для цитирования: Родзевич С. А. Аппаратная поддержка двоичной трансляции x86 вещественной арифметики в процессорах «Эльбрус» // Вопросы радиоэлектроники. 2018. № 2. С. 59–64.
УДК 004.4'2

С. А. Родзевич[1]

[1] АО «МЦСТ»

АППАРАТНАЯ ПОДДЕРЖКА ДВОИЧНОЙ ТРАНСЛЯЦИИ X86 ВЕЩЕСТВЕННОЙ АРИФМЕТИКИ В ПРОЦЕССОРАХ «ЭЛЬБРУС»

Двоичная трансляция – исполнение кода, скомпилированного под одну архитектуру, на процессорах другой архитектуры. В целях расширения области применения процессоров «Эльбрус» применяется двоичная трансляция с архитектуры x86, так как она является наиболее распространенной и под нее написано огромное количество ПО. Вещественная арифметика является отдельным подмножеством системы команд x86 и имеет некоторые особенности, затрудняющие ее эффективную двоичную трансляцию на процессорах «Эльбрус». В статье рассматриваются эти особенности и проблемы, к которым они приводят в контексте двоичной трансляции. Приводятся основные особенности архитектуры «Эльбрус», описывается многоуровневая схема двоичного транслятора. Рассматриваются последовательные этапы развития аппаратной поддержки двоичной трансляции вещественной арифметики и то, как они повлияли на эффективность генерируемого кода третьего уровня двоичного транслятора. В результате была разработана схема, которая позволила избавиться от большей части проблем и сделала эффективность трансляции вещественной арифметики сопоставимой с таковой для целочисленной.

Ключевые слова: *«Эльбрус», двоичная трансляция.*

Введение

Технология двоичной трансляции применяется в тех случаях, когда необходимо исполнять код, скомпилированный под одну архитектуру, на процессорах другой архитектуры. Так как ПО под процессоры «Эльбрус» практически отсутствует (за исключением того, что можно собрать из исходных кодов), а под процессоры x86 написано огромное его количество, то для запуска этого ПО (в том числе операционных систем, например Windows) было решено использовать двоичную трансляцию.

Ввиду того что архитектуры «Эльбрус» и x86 существенно отличаются, необходимо тщательно учитывать и при необходимости эмулировать все особенности последней, чтобы поведение программы на процессорах «Эльбрус» в точности соответствовало таковому на процессорах x86. Это относится, в частности, к вещественной арифметике, реализация которой в архитектуре «Эльбрус» довольно сильно отличается от x86.

С целью более эффективной трансляции кода вещественной арифметики в процессорах «Эльбрус» была реализована ее поддержка со стороны аппаратуры, которая со временем совершенствовалась. В данной статье рассмотрены особенности реализации вещественной арифметики в архитектурах x86 и «Эльбрус», а также этапы улучшения поддержки двоичной трансляции x86 вещественной арифметики в процессорах «Эльбрус».

Механизм работы вещественной арифметики в архитектуре x86

Изначально в первых процессорах Intel для поддержки вещественной (FP – floating point) арифметики использовался отдельный математический сопроцессор, реализующий соответствующее подмножество системы команд x86, которое стали называть x87 (как и в случае с x86, по названию первого чипа с этим сопроцессором, 8087). Позднее, начиная с процессора 80486, данный набор инструкций стали реализовывать непосредственно внутри процессора (тем не менее название «x87» все еще используют для обозначения этого подмножества x86). Однако в целях обратной совместимости механизм работы с вещественной арифметикой остается неизменным до сих пор. В этом есть свои минусы, так как, в отличие от многих других реализаций, регистры в x87 представлены не в виде массива, а в виде стека, и семантика вещественных операций здесь соответствующая.

Рассмотрим подробнее, как устроена система команд x87. Вещественные регистры представляют собой стек из восьми элементов и адресуются

с ST(0) по ST(7). Номер регистра считается относительно вершины стека, т.е. если вершина стека равна 2, то ST(3) будет адресовать 5-й регистр. Вершина стека хранится в отдельном регистре, для простоты назовем его TOP (в реальности он является частью статусного слова FPU, но в рамках данной статьи это несущественно).

У большинства операций одним из аргументов (или результатом) неявно является ST(0), т.е. производится работа с вершиной стека. Присутствуют также инструкции, которые читают/записывают значение в оперативной памяти, при этом возможно преобразование значения в целое и обратно. Собственно стековая модель проявляется в том, что типичный x87 код устроен таким образом: чтение аргументов из памяти (PUSH), выполнение над ними операции, запись результата в память (POP). Промежуточные значения можно хранить на оставшихся регистрах и использовать их при необходимости.

Помимо вершины стека (TOP) имеется отдельный регистр для хранения маски валидности регистров стека – VAL (он также является частью статусного слова, но, как и в случае с TOP, в данном контексте это несущественно). Смысл маски валидности заключается в следующем. При записи в какой-либо регистр стека (в частности, при PUSH) он становится валидным (соответствующий бит в маске выставляется в 1). Регистр помечается как невалидный, когда производится его POP (соответствующий бит в маске выставляется в 0). При выполнении PUSH в уже валидный регистр будет выработано исключение stack overflow (т.е. попытка сделать PUSH в полностью заполненный стек). При чтении невалидного регистра (в частности, при выполнении POP) будет выработано исключение stack underflow (т.е. попытка прочитать значение, которого нет, например при пустом стеке). Стоит отметить, что маска валидности является обычным байтом (по биту на каждый из восьми регистров), и при доступе к ней (со стороны программы) вершина стека никак не используется.

Указатель вершины стека (собственно сама стековая модель) и маска валидности являются основными особенностями x87, с которыми возникают трудности при двоичной трансляции.

Помимо этого, MMX-инструкции в x86 используют те же регистры, что и FP-инструкции. При этом при исполнении MMX-инструкций TOP становится равным 0, а все регистры помечаются как валидные. Для возврата к возможности исполнения FP-инструкций предусмотрена специальная инструкция EMMS, которая также обнуляет TOP, но помечает все регистры как невалидные (для возможности загрузки в них значений вещественными операциями) [1].

Особенности архитектуры «Эльбрус»

Архитектура «Эльбрус» относится к классу VLIW и представляет собой суперскалярную архитектуру с длиной машинной команды до 512 бит (от 1 до 8 двойных слов, размер слова – 32 бита). Каждая машинная команда состоит из так называемых слогов, каждый из которых представляет собой команду для некоторого устройства процессора. Размер слога равен одному слову. В одной команде могут быть, например, несколько арифметических операций (включают в себя чтение/запись памяти), операция передачи управления, а также несколько служебных операций.

Рассмотрим основные отличия архитектуры «Эльбрус» от архитектуры x86, которые являются существенными в контексте данной статьи.

Исполнение команд

В архитектуре x86 инструкции выполняются последовательно одна за другой, и следующая инструкция не начнет выполняться, пока не будут готовы результаты текущей. При этом современные процессоры, конечно, считывают не одну инструкцию, а сразу несколько, и выполняют некоторые оптимизации на аппаратном уровне, переводя последовательность команд x86 (CISC) в последовательность команд внутреннего представления процессора (RISC). Но семантика последовательного выполнения команд при этом сохраняется.

В архитектуре «Эльбрус» широкая команда считывается за один такт, и если все операнды для всех слогов готовы, то немедленно начинается их исполнение, и в следующем такте считывается уже следующая команда. Если же один из операндов не готов, процессор простаивает, ожидая готовности всех операндов. Таким образом, управление логикой процессора переносится на компилятор и возникает задача планирования, т.е. наиболее эффективного размещения инструкций внутри длинных слов, чтобы процессор простаивал как можно меньше.

Регистры

В архитектуре x86 программисту доступно всего 8 регистров общего назначения (16 в архитектуре x86–64). При этом почти всегда возникает серьезная нехватка регистров, и интенсивно обрабатываемые данные приходится хранить в оперативной памяти, что компенсируется выполнением ряда оптимизаций на аппаратном уровне.

В архитектуре «Эльбрус» программисту доступно 256 регистров общего назначения размером двойного слова (64 бита). Это в большинстве случаев избавляет от проблемы нехватки регистров, но появляется задача распределения переменных по регистрам, т.к. процессор, в отличие от x86, не будет выполнять каких-либо оптимизаций аппаратно.

Предикатное исполнение

В архитектуре x86 либо инструкции выполняются безусловно, либо исполнение зависит от значения флагов (например, команды перехода JZ, JBE и т.д.).

В архитектуре «Эльбрус» любая арифметическая операция может быть поставлена под предикат. В процессоре имеются 32 предикатных регистра и набор операций над ними. Предикатный регистр может хранить 1 или 0. Если значение предикатного регистра, управляющего выполнением данной операции, равно 1, операция будет исполнена. В противном случае исполнения не произойдет, как не произойдет и ожидания готовности операндов. Таким же образом производится условная передача управления [2].

Вещественная арифметика в архитектуре «Эльбрус»

В архитектуре «Эльбрус» вещественные инструкции изначально были частью системы команд процессора, поэтому недостатки, присущие x86, здесь отсутствуют. Вещественные операции работают с теми же регистрами, что и все прочие операции (а для адресации регистров используется классическая схема массива), поэтому какая-либо дополнительная семантика для них отсутствует.

Однако при двоичной трансляции x86 кода возникает необходимость поддерживать семантику x86, и без аппаратной поддержки со стороны процессора это было бы очень ресурсоемко, т.к. нужно хранить вершину стека и маску валидности, при исполнении каждой операции рассчитывать адреса операндов, проверять валидность и выполнять модификацию вершины и маски валидности по завершении операции. В итоге для выполнения большей части этих действий было решено сделать в процессоре аппаратную поддержку.

Первая версия аппаратной поддержки x86 вещественной арифметики

Впервые аппаратная поддержка x86 вещественной арифметики была реализована в процессорах «Эльбрус-3М». В области глобальных регистров (на которых хранятся значения x86 регистров) было выделено окно шириной в 8 регистров, при доступе в которые применялась семантика «вращения»; для простоты будем считать, что регистры этого окна нумеруются с 0, и назовем их G(0)–G(7). Для этого существовал отдельный регистр BGR, в котором хранился указатель вершины стека (top), а также маска валидности (val). При обращении к регистру из данного окна к номеру регистра прибавлялось значение указателя вершины по модулю 8, т.е. регистру G(x) соответствовал регистр G((x+BGR.top) mod 8). В такой схеме – если хранить в BGR.top текущее значение x86 указателя вершины, то x86 регистру ST(x) соответствует эльбрусовский G(x).

Проблема адресации регистров решена, теперь рассмотрим маску валидности. Как было упомянуто выше, маска валидности хранилась в BGR.val. Для проверки валидности читаемого регистра приходилось извлекать эту маску, циклически сдвигать ее на BGR.top (т.к. нам известен только относительный номер проверяемого регистра ST(x)), после чего проверять в ней соответствующий бит и эмулировать выдачу исключения в случае нарушения валидности. Все это делалось программно, что было не особо эффективно.

По завершении операции для модификации значений BGR.top и BGR.val имелась специальная операция ABG, которая по сути выполняла семантику PUSH или POP (только одного за раз), т.е. уменьшала/увеличивала значение BGR.top и взводила/сбрасывала соответствующий бит в маске валидности (в данном случае уже с учетом сдвига на BGR.top).

Особенности поддержки вещественной арифметики в третьем уровне двоичного транслятора

Двоичный транслятор x86→«Эльбрус» имеет четыре уровня оптимизации. Каждый следующий уровень используется при превышении количества исполнения некоторой части x86-кода определенного порога [3]:

1. Интерпретатор. x86-инструкции считываются и исполняются одна за другой.
2. Шаблонный транслятор. Производится набор линейного участка (последовательности x86-инструкций без передачи управления), и генерируется нативный код (код архитектуры «Эльбрус»), соответствующий этому участку.
3. Быстрый регионный компилятор. Производится набор так называемого региона, который состоит из линейных участков и дуг между ними (дуги соответствуют передачам управления), после чего каждый линейный участок практически независимо от других транслируется в нативный код (в процессе производятся некоторые несложные оптимизации).
4. Оптимизирующий регионный компилятор. То же самое, что третий уровень, но оптимизация выполняется по максимуму (более 100 фаз оптимизации).

В данной статье рассмотрим только третий уровень двоичного транслятора.

Очевидно, что генерировать все описанные выше действия, необходимые для выполнения x86-семантики, для каждой вещественной инструкции

неэффективно. Поэтому применяется следующая техника. В линейном участке выделяется так называемый FP-фрагмент, который начинается с первой FP-инструкции и заканчивается на инструкции передачи управления (конец линейного участка) или на MMX-инструкции. Внутри FP-фрагмента известны все изменения указателя вершины и маски валидности (относительно начала фрагмента), поэтому оптимизация производится в следующем порядке (рассматривается вариант для первой версии аппаратной поддержки):

1. Генерируется основная семантика всех FP-инструкций без учета маски валидности. При этом запоминается суммарное изменение указателя вершины, а также формируются две маски: регистров, которые должны быть валидными, и регистров, которые должны быть невалидными на начало фрагмента (т.е. статически известно, что при нарушении этих условий где-то в FP-фрагменте будет выработано исключение).
2. После завершения генерации основной семантики для всего фрагмента в его начале генерируется код, проверяющий валидность регистров (по маскам, полученным на предыдущем шаге) и выдающий исключение в случае ошибки.
3. В конце фрагмента генерируется сведение контекста, которое включает в себя запись значений в измененные глобальные регистры и генерацию необходимого количества операций ABG для смещения указателя вершины на полученное на первом шаге значение.

Необходимость второго шага нуждается в пояснении. Дело в том, что в третьем уровне двоичного транслятора большинство операций выполняется в спекулятивном режиме. В этом режиме операция в качестве результата вместо исключения выдает специальное значение (называемое диагностическим), которое при попадании в неспекулятивную операцию приводит к исключению. Это позволяет гораздо эффективнее размешивать операции линейного участка, т.к. вычисления можно произвести как можно раньше, а исключения проявить уже потом. В случае FP-фрагмента приходится проверять валидность регистров в самом начале. Дело в том, что, хотя для результатов FP-операций в любом случае генерируются специальные неспекулятивные потребители (для проявления арифметических исключений, например, округление или деление на 0), с их помощью не получится проявить исключения overflow/underflow, т.к. операции ABG в целях оптимизации генерируются только в конце фрагмента, и в процессе его исполнения BGR.top и BGR.val сохраняют свои значения такими, какими они были на начало фрагмента. В любом случае с точки зрения производительности выгоднее проявить исключения как можно скорее, т.к. в случае исключения будет произведен выход из региона и весь линейный участок (с самого начала) будет исполнен в интерпретаторе (обработка вообще всех исключений происходит только в интерпретаторе).

Для MMX-фрагментов используется более простая схема. Все, что требуется, – это обнулить указатель вершины и пометить все регистры как валидные (как этого требует семантика x86). MMX-фрагмент, в свою очередь, заканчивается на инструкции передачи управления или на FP-инструкции. Таким образом, в пределах одного линейного участка может быть несколько FP/MMX-фрагментов.

Данная схема гораздо эффективнее генерации полной семантики для каждой инструкции, однако имеет ряд недостатков:

1. Код для проверки маски валидности, генерируемый в начале фрагмента, получается достаточно громоздким (в случае короткого фрагмента может быть длиннее, чем основная семантика фрагмента).
2. Для выдачи исключения приходится генерировать дополнительную конструкцию для выхода из региона, т.к. аппаратно исключение не генерируется.
3. Так как операция ABG одновременно меняет и указатель вершины, и соответствующий бит маски валидности (т.е. выполняет семантику PUSH/POP), то невозможно в полной мере поддержать запись в произвольный регистр стека (т.к. его нельзя будет сделать валидным). Для этого при проверке валидности в начале фрагмента консервативно проверяются регистры, в которые производится запись, – они должны быть валидными.
4. При большом суммарном смещении указателя вершины приходится генерировать достаточно много операций ABG, причем в коде они могут размещаться только по одной в широкую команду. При этом операции ABG нельзя поставить под предикат, что является проблемой для проведения некоторых оптимизаций.
5. Задержка от записи в BGR до его использования (т.е. до чтения регистров из вещественного окна) составляет целых 6 тактов (в то время как большинство других задержек, кроме обращений в память, составляют 1–3 такта). Это приводит к потере производительности в случае, когда за FP-фрагментом следует MMX (нельзя начать выполнять MMX-операции до сброса BGR, а сброс BGR цепляется за операции ABG в конце FP-фрагмента).

Рассмотрим, как эти проблемы были решены в последующих версиях системы команд «Эльбрус».

Дальнейшее развитие аппаратной поддержки

Третья версия системы команд

В третьей версии системы команд (процессор «Эльбрус-2S») была решена первая проблема из перечисленных выше. Для этого была введена специальная операция VFBG, заменившая собой весь тот код, который раньше приходилось генерировать для проверки маски валидности. Аргументами данной операции являлись две маски (валидных и невалидных регистров). Чтение BGR.val, сдвиг на BGR.top и проверка масок выполнялись аппаратно. Результатом операции являлся предикат, под которым и производился выход из региона в случае несоответствия.

Третья версия системы команд имела множество улучшений, которые косвенно повлияли и на производительность FP/MMX, но основные проблемы никуда не делись. В результате в шестой версии системы команд (процессоры пока находятся в стадии разработки) была полностью переработана аппаратная поддержка двоичной трансляции FP/MMX. Далее речь пойдет о ней.

Шестая версия системы команд

В шестой версии системы команд было принято решение существенно переделать механизм поддержки трансляции FP/MMX с учетом полученного на тот момент опыта. Базовые принципы остались прежними, однако подход для работы с указателем вершины и маской валидности полностью изменился и позволил решить все имеющиеся проблемы:

1. Операция ABG теперь сдвигает BGR.top на произвольное значение (за одну операцию), при этом не изменяя маску валидности. Также появилась возможность ставить ее под предикат. Это решило проблемы 3 (частично) и 4.
2. Операция VFBG вместо выдачи предиката стала генерировать исключение в случае ошибки. Это решило проблему 2.
3. Для модификации маски валидности была введена операция MODBGV, которая в качестве аргументов принимает маски битов, которые нужно выставить в 1 и 0. Это также решило проблему 3, т.к. теперь стало возможным сделать валидным произвольный регистр, в который была запись.
4. Проблема 5 была решена за счет введения операции RBG, которая сбрасывает значение BGR.top в 0. При этом соответствующее изменение масок (для MMX-инструкции и EMMS) производится отдельно операцией MODBGV. Так как эти две операции могут находиться в одной широкой команде, а задержка от них до использования BGR составляла один такт, то проблемы больше не возникает.

Заключение

В результате, пусть на практике эти изменения в среднем и не привели к более-менее значительным улучшениям производительности, текущая система команд позволяет более гибко работать с вещественным стеком. В частности, ожидается получить достаточно хороший прирост производительности в третьем уровне после полноценной реализации оптимизации if-conversion (склеивание нескольких узлов в один с постановкой операций под предикаты), т.к. теперь операции ABG можно ставить под предикат. Сейчас в третьем уровне if-conversion склеивает только узлы по наиболее вероятному пути (из-за особенностей архитектуры компилятора третьего уровня), поэтому в полной мере эффект от всех улучшений увидеть не удается.

СПИСОК ЛИТЕРАТУРЫ

1. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, Jan. 2015.
2. Ким А.К., Перекатов В.И., Ермаков С.Г. Микропроцессоры и вычислительные комплексы семейства «Эльбрус». СПб.: Питер, 2013. 272 с.
3. Система динамической двоичной трансляции x86→«Эльбрус» / Н.В. Воронов, В.Д. Гимпельсон, М.В. Маслов, А.А. Рыбаков, Н.С. Сюсюкалов // Вопросы радиоэлектроники. 2012. № 3. Т. 4. С. 89–108.

ИНФОРМАЦИЯ ОБ АВТОРЕ

Родзевич Сергей Аркадьевич, инженер-программист, АО «МЦСТ», 119334, Москва, ул. Вавилова, д.24, тел.: 8 (916) 227-08-65, e-mail: sergey.a.rodzevich@mcst.ru.

For citation: Rodzevich S.A. Hardware support for binary translation of x86 floating point arithmetic in Elbrus processors. Voprosy radioelektroniki, 2018, no. 2, pp. 59–64.

S.A. Rodzevich

HARDWARE SUPPORT FOR BINARY TRANSLATION OF X86 FLOATING POINT ARITHMETIC IN ELBRUS PROCESSORS

Binary translation is the execution of code compiled for one architecture on processors of a different architecture. In order to expand the scope of the Elbrus processors, binary translation from the x86 architecture is used, because it is the most common,

and under it is written a huge amount of software. Real arithmetic is a separate subset of the x86 command system and has some features that make it difficult to efficiently binary broadcast on Elbrus processors. The article considers these features and problems, which they lead to in the context of binary translation. The main features of the Elbrus architecture are described, and a multilevel scheme of the binary translator is described. Sequential stages of the development of hardware support for the binary translation of real arithmetic and how they affected the efficiency of the generated code of the third level of the binary translator are considered. As a result, a scheme was developed that made it possible to get rid of most of the problems and made the efficiency of translating real arithmetic comparable to that of an integer one.

Keywords: FP, Elbrus, binary translation.

REFERENCES

1. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, Jan. 2015.
2. Kim A.K., Perekatov V.I., Ermakov S.G. *Mikroprocessory i vychislitelnye kompleksy semejstva «Ehlbrus»* [Elbrus microprocessors and computing systems]. Saint Petersburg, Piter Publ., 2013, 272 p. (In Russian).
3. Voronov N.V., Gimpelson V.D., Maslov M.V., Rybakov A.A., Syusyukalov N.S. Dynamic binary translation system x86→Elbrus. *Voprosy radioelektroniki,* 2012, no. 3б Vol. 4, pp. 89–108 (In Russian).

AUTHOR

Rodzevich Sergey, engineer-programmer, JSC MCST, 24, ulitsa Vavilova, Moscow, 119334, Russian Federation, tel.: +7 (916) 227-08-65, e-mail: sergey.a.rodzevich@mcst.ru.