DOI: 10.15514/ISPRAS-2020-32(2)-7

Исследование технологии RISC-V

[1,2] *В.А. Фролов, ORCID: 0000-0001-8829-9884 <vova@frolov.pp.ru>*
[1] *В.А. Галактионов, ORCID: 0000-0001-6460-7539 <vlgal@gin.keldysh.ru>*
[2] *В.В. Санжаров, ORCID: 0000-0001-6455-6444 <vadim.sanzharov@graphics.cs.msu.ru>*
[1] *Институт прикладной математики им. М.В. Келдыша РАН, 125047, Москва, Миусская пл., д.4*
[2] *Московский государственный университет им. М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*

Аннотация. Система команд – это стержень, вокруг которого строится весь остальной процессор. Ошибки или негибкость в решениях, однажды заложенные в систему команд, остаются с этим поколением процессоров навсегда. Поэтому одна из ключевых причин, по которой рост производительности современных CPU замедлился, заключается в том, что исходный код процессоров «испортился» в прямом и переносном смысле этого слова: процессоры внутри становятся сложными, из-за чего их дальнейшее развитие затрудняется. Разработка современных ЭВМ (CPU, GPU или специализированных систем) – это крайне дорогостоящий процесс, состоящий из большого количества затратных статей. Поэтому вопрос цены, или, скорее, целесообразности разработки процессора является ключевым. В данной работе мы провели исследование существующих популярных систем команд процессора и сделали выводы о перспективности в настоящее время направления RISC-V и других открытых систем команд CPU. Мы постарались ответить на следующие вопросы: почему система команд процессора – это действительно важно? Почему именно RISC-V, чем он лучше остальных? Какие возможности RISC-V открывает для российских разработчиков и какие у него есть аналоги?

Ключевые слова: RISC-V; система команд

Для цитирования: Фролов В.А., Галактионов В.А., Санжаров В.В. Исследование технологии RISC-V. Труды ИСП РАН, том 32, вып. 2, 2020 г., стр. 81-98. DOI: 10.15514/ISPRAS–2020–32(2)–7

Investigation of the RISC-V

[1,2] *V.A. Frolov, ORCID: 0000-0001-8829-9884 <vova@frolov.pp.ru>*
[1] *V.A. Galaktionov, ORCID: 0000-0001-6460-7539 <vlgal@gin.keldysh.ru>*
[2] *V.V. Sanzharov, ORCID: 0000-0001-6455-6444 <vadim.sanzharov@graphics.cs.msu.ru>*
[1] *Keldysh Institute of Applied Mathematics RAS, Miusskaya sq., 4, Moscow, 125047, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskiye Gory, Moscow, 119991, Russia*

Abstract. An Instruction Set Architecture (ISA) is the core around which the rest of the CPU is built. Errors or inflexibility in decisions once embedded in a system of instructions remain with this generation of processors forever. Therefore, one of the key reasons why the performance growth of modern CPUs has slowed down is that the source code of the processors “got corrupted” in literal and figurative sense of the word: the processors

inside become complex which makes their further development difficult. The development of modern computers (CPU, GPU or specialized ones) is in any case an extremely expensive process consisting of a large number of costly articles. Therefore, the issue of cost of developing a processor is the corestone. In this work we conducted a study of existing popular processor command systems and made conclusions about the prospects of the RISC-V and other open source instruction set architectures. We tried to answer the following questions: why the processor instruction set architecture is really important? Why RISC-V, why is it better than the others? Which opportunities does RISC-V open for developers around the world and what analogues does it have?

Keywords: RISC-V, Instruction Set Architecture

For citation: Frolov V.A., Galaktionov V.A., Sanzharov V.V. Investigation of the RISC-V. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 2, 2020. pp. 81-98 (in Russian). DOI: 10.15514/ISPRAS–2020–32(2)–7

1. Введение

Одна из наших основных целей как разработчиков – уметь писать высокоэффективные приложения. Проблема в том, что в современном мире это не так просто сделать. Среднестатистический код на C++ использует едва ли десятую долю производительности современных CPU [1]. Такая ситуация во многом сложилась от отсутствия прозрачного интерфейса между программистом и аппаратурой.

- Высокоуровневое алгоритмическое описание на C++ (или другом языке) транслируется в ассемблер с применением самых разных техник оптимизации вплоть до автоматической векторизации. На этом этапе компилятор активно использует машинно-зависимые оптимизации (то есть оптимизирует код для конкретного семейства процессоров).
- Ассемблер в действительности не позволяет получить представление о том, насколько данный код оптимизирован, даже несмотря на то, что он напрямую транслируется в машинный код. Вот здесь есть фундаментальная проблема.

Дело в том, что помимо архитектурного уровня, то есть непосредственно системы команд, видимой компилятору и программисту, существует ещё микроархитектурный – то, как именно эти команды выполняются внутри процессора. Таким образом, система команд – это не что иное, как интерфейс, а конкретное поколение процессоров (например, AMD Zen [2] или Intel Cascade Lake) – это реализация интерфейса. Когда интерфейс устаревает (перестаёт удовлетворять актуальным требованиям по какой-либо причине), программист его меняет. Однако существуют ситуации, когда интерфейс нельзя изменить – если его старой версией пользуются другие разработчики. Для программных систем это не является большой проблемой, поскольку старый интерфейс, как правило, может быть реализован через новый и, таким образом, можно двигаться вперёд, сохраняя поддержку для пользователей старого интерфейса какое-то время. Однако для аппаратуры это не так.

Проблема #1: Любая лишняя функциональность в системе команд будет тратить ресурсы кристалла (транзисторы, частоту, тепловыделение) и увеличивать сложность решения в целом. Дополнительные стадии конвейера, нацеленные, например, на сохранение частоты будут причиной роста латентности (непосредственное время выполнения) инструкций, что может повлиять на эффективность остальной системы. Поскольку обратная совместимость для CPU важна, с каждым новым поколением процессора, расширяющим базовый интерфейс, проблемы растут как снежный ком. Мы рассмотрим недостатки существующих систем команд в обзоре. Однако прежде чем мы к ним перейдём, необходимо упомянуть ещё несколько проблем.

Проблема #2: **Отсутствие кроссплатформенности.** Высокопроизводительные программные компоненты (библиотеки) на сегодняшний день очень сильно зависят от конкретной аппаратуры. Производители процессоров намеренно выпускают “открытый” софт, напичканный как можно большим числом своих инструкций. Например, Intel предоставляет бесплатную библиотеку трассировки лучей Embree (и многие другие

бесплатные библиотеки), сверху до низу прошитую аппаратно-зависимыми «инстринсиками». Излишне упоминать о стоимости процессоров Xeon, под которые эта библиотека оптимизируется. Перенести подобную библиотеку на процессор с другой системой команд – задача, эквивалентная по стоимости написанию своего аналога с нуля. Хуже всего, что перенос кода «в лоб» приведёт к низкой производительности, и, следовательно, не будет иметь смысла.

Приведём простой пример. В x64 есть инструкция shuffle, которая позволяет произвольным образом перераспределить содержимое векторного регистра. С помощью такой инструкции, например, можно реализовать векторное произведение для двух трёхмерных векторов, хранящихся в регистрах:

```
__m128  shuffle_yzxw(__m128 a_src)
{ return _mm_shuffle_ps(a_src, a_src, _MM_SHUFFLE(3, 0, 2, 1)); }

__m128 cross(const __m128 a, const __m128 b)
 {
    const __m128 a_yzx = shuffle_yzxw(a);
    const __m128 b_yzx = shuffle_yzxw(b);
    const __m128 c =
           _mm_sub_ps(_mm_mul_ps(a, b_yzx), _mm_mul_ps(a_yzx, b));
    return shuffle_yzxw(c);
 }
```

Однако оказывается, что эффективно реализовать на ARM аналог shuffle именно с таким порядком компонент (yzxw), как и в общем случае, невозможно. Но это не значит, тем не менее, что код, использующий векторные произведения, не может быть эффективно реализован на ARM. Просто вычисления и данные изначально должны быть организованы по-другому (например, полноценная векторизация кода с обработкой по 4 элементам, которая в общем случае даже на x64 работает лучше). Конечно, опытный читатель может возразить, что, ограничиваясь лишь конструкциями языка программирования (например, C++ или Ada), он может избежать проблем с переносимостью. Однако, даже если не брать в расчёт аппаратно-зависимые библиотеки, сохраняются нюансы (см. ниже).

Проблема #3: Компилятор. Разработчики процессоров (например, AMD) активно вносят свой вклад, особенно в gcc [2] и ядро Linux [3], потому что они в этом непосредственно заинтересованы. Компилятор и ОС – это такой же зависимый от производителя аппаратуры софт. Нельзя ожидать, что один и тот же код оптимизируется под разные процессоры одинаково хорошо, даже если они имеют практически идентичную функциональность (хотя LLVM и амортизирует эту проблему, но лишь отчасти).

Совершенно отдельный разговор – **безопасность системы** (***проблема #4***). Здесь всё очень плохо, потому что вам её, попросту говоря, никто не гарантирует (см. уязвимости «Spectre» и «Meltdown»). Хотите быть защищены – делайте целиком свой компьютер плюс ОС, компилятор и драйверы с нуля.

И, наконец, (***проблема #5***) – **эффективное взаимодействие потоков** в многопоточной программе – вопрос, не решённый в большинстве систем команд на должном уровне. Сейчас это особенно важно, так как процессорных ядер становится всё больше.

Таким образом, производители процессоров тянут одеяло на себя, зачастую намеренно добавляя «удобные» инструкции в свои процессоры. Разработчики программных систем начинают привыкать к этому «удобству», не понимая, к чему это приведёт в дальнейшем при переносе кода на процессор с другой системой команд. Мы не хотим сказать, что конкретно инструкция shuffle – это плохая идея. Мы хотим сказать, что этот вопрос не должен решаться одной компанией, если кроссплатформенность, обратная совместимость и безопасность для нас важны. Должен быть устойчивый стандарт, на который вы, как разработчик, можете

опираться, чтобы ваша программа в будущем могла бы быть перенесена на новые, лучшие, более быстрые, энергоэффективные или безопасные вычислительные системы без существенных потерь в производительности.

Замечание. Интересно упомянуть в этой связи графические процессоры. Поскольку обратная совместимость для них никогда не являлась целью, GPU прогрессировали существенно быстрее и достигли известных успехов. Нельзя сказать, что там всё просто, но проблемы переносимости ПО для них решаются на уровне графических и вычислительных API, что само по себе намного более гибко, чем система команд. В настоящее время производители GPU смогли договориться о едином открытом стандарте, называемом Vulkan, который поддерживают почти все современные десктопные и мобильные GPU. Как и RISC-V, Vulkan – хороший и грамотно спроектированный стандарт. Мы надеемся, что расскажем о том, почему это так в нашей следующей статье. А пока мы можем резюмировать, что, как бы странно это ни звучало, в современном мире программные системы и алгоритмы, интенсивно использующие GPU, в целом гораздо более кроссплатформенны, чем их CPU аналоги.

2. Обзор существующих систем команд

Прежде всего, необходимо разобраться в накопленном мировом опыте.

2.1 X86/X64

Intel и AMD добились успехов в распространении процессоров этой архитектуры благодаря обратной совместимости, агрессивным оптимизациям и лидирующим технологиям производства [4]. Дизайн набора инструкций – не их сильная сторона. Об этом говорит хотя бы то, что внутри этих процессоров x86/x64 инструкции давно уже транслируются в некоторое более простое, RISC-подобное представление. Всё это очень напоминает какую-нибудь старую программу с legacy функциональностью, которая до сих пор жива только потому, что все к ней уже привыкли. Но если отбросить совместимость и посмотреть объективно, в x86/x64 «просто нет смысла» ([4], с.12). А ведь это было сказано ещё в далёком 1994 г. [5]. Основная проблема x86/x64 – это исключительно раздутая (больше 2000) и плохо структурированная система команд. Многие команды уже давно никем не используются и не поддерживаются, но продолжают оставаться в системе команд в силу обратной совместимости. Вот несколько основных проблем.

Кодирование команд в x86/x64. В x86/x64 команды могут занимать разное число байт: от 1 до 15. При этом более короткие операции со временем стали использоваться реже. Изначальная идея была в том, чтобы кодировать наиболее частые операции меньшим числом байт (см. коды Хаффмана). Но со временем она повернулась обратной стороной. Например, целых 6 8-битных инструкций должны обрабатывать числа в десятеричном представлении, что уже давно не поддерживается в gcc ([4], с.12) и не реализуется современными процессорами, но по-прежнему занимает место в системе кодирования команд. Аналогично можно сказать и про весь сопроцессор x87, который хоть и используется в старых 32 битных программах, в 2020 году является атавизмом.

Регистры. В x86/x64 регистров очень много (рис. 1). Но в то же время, как ни странно, их очень мало. Много – из-за обратной совместимости x64 с x86 и его сегментными регистрами, x87 стеком FPU и прочими уже давно никем не используемыми вещами. Мало, потому что полезных регистров лишь 16 (а в x86 их всего 8). При этом, новые регистры всё время нарастают поверх старых (рис. 1), потому что 32 битные процессоры должны были уметь выполнять существующий бинарный 16 битный код, а 64 разрядные – 32 битный.

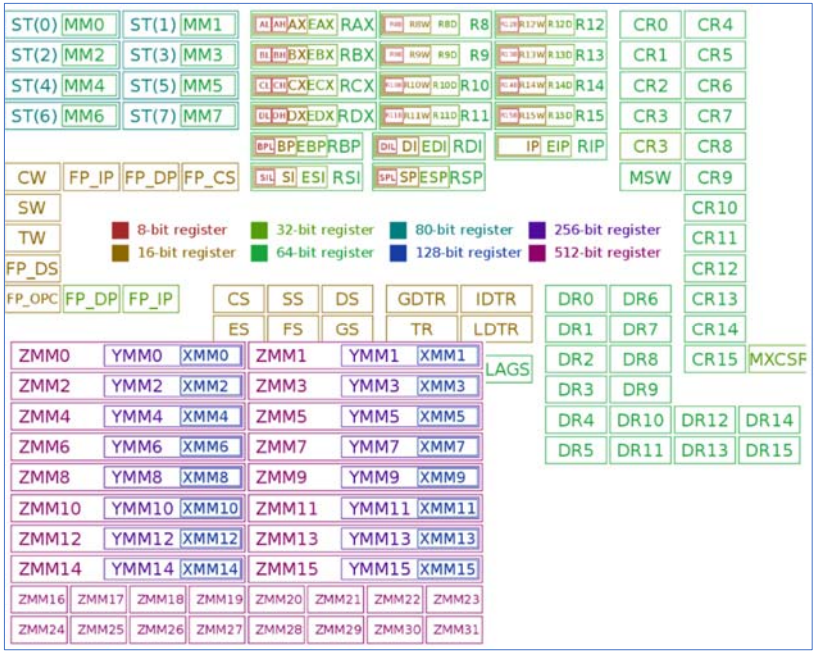

Рис. 1. Регистры x64
Fig. 1. x64 registers

Может показаться, что для современных чипов это число не так велико, но это совершенно не так: (1) Регистровая память в современных CPU может иметь большое количество портов и в действительности быть очень дорогой; (2) механизм переименования регистров, безусловно применяемый во всех современных CPU с внеочередным выполнением команд, использует в N раз больше физических регистров, чем в системе команд доступно логических (где N обычно от 2 до 4). Такое большое число неиспользуемых регистров граничит с преступлением. Но это ещё не всё и современный x64 обладает рядом других проблем.

- Двухадресные инструкции. Такие инструкции всегда перезаписывают один из операндов, что препятствует нормальным оптимизациям и вынуждает компилятор вставлять дополнительные операции «mov». Эта проблема даже породила своеобразный мем в программистской среде: «i like to mov it mov it».
- Некоторые предикатные инструкции призваны повысить производительность. Тем не менее, зачастую эта цель в x64 не достигается, т. к. их семантика плохо продумана. Например, в x64 есть команда условной загрузки. Однако, если для безусловной загрузки исключение при обращении по некорректному адресу строго обязательно, то для условной загрузки это не специфицируется. Из-за этого компилятор очень редко может использовать эту инструкцию для выполнения оптимизации, когда необходимо гарантировать корректность выполнения кода. Кстати говоря, именно эти инструкции активно используются в широко известных уязвимостях Spectre и Meltdown (которые, впрочем, затрагивают большинство современных архитектур, использующих спекулятивное исполнение команд). После исправлений критических уязвимостей в драйверах ОС производительность некоторых приложений наоборот упала на 30%.

- Некоторые регистры общего назначения в действительности такими не являются. Например, результат деления всегда будет находиться в паре DX/AX, а результат сдвига всегда поступает через DX. ESI/EDI также имеют специальную семантику. В целом такой шаблон проектирования приводит к неэффективному перемещению данных между регистрами и стеком [4].

2.2 ARMv7

На сегодняшний день x86/x64 аккумулирует в себе существенно больше недостатков, чем все другие системы. Неудивительно, что в области встроенных систем, где важна энергоэффективность, x86/x64 проиграла архитектуре ARM. Если сравнивать ARM и x86/x64, то ARM безусловно выглядит лучше. Однако и здесь проблем достаточно.

- Отсутствие поддержки 64-битного адресного пространства в ARMv7.
- ARMv7 – это, на самом деле не одна, а целых 3 системы команд: обычный режим и два сжатых: Thumb и Thumb 2. Из-за этого, в итоге, декодеры команд должны понимать три системы команд, что увеличивает энергопотребление, задержку и стоимость проектирования.
- ARMv7 по факту не является классическим RISC набором. Например, программный счётчик – это один из адресуемых регистров. А это означает, что почти любая инструкция может изменить поток управления (т. е. выполнить переход). Хуже всего то, что последний значащий бит счетчика программы отражает, какой именно набор команд в данный момент выполняется (ARM или Thumb), то есть обычная инструкция сложения может изменить то, какой именно набор команд в данный момент выполняется на процессоре!
- Использование кодов условий для переходов и предикации по факту лишь усложняет высокопроизводительные реализации с внеочередным выполнением команд, хотя может быть плюсом для процессоров с очередным выполнением или простым внеочередным выполнением на основе scoreboard.
- Совершенно отдельная история – комплексные команды для вызова функций вроде «LDMIAEQ SP, R4-R7, PC», которая «выполняет 6 загрузок, увеличивает счётчик программ и записывает 7 регистров в стек, включая адрес возврата … и, кроме того, это условная команда, то есть на самом деле это ещё одновременно и условный переход» [4].
- Наконец, ARMv7 – это очень большая система команд, включающая в себя более 600 инструкций даже без плавающей точки и SIMD.

Конечно, вопрос о вреде или пользе комплексных команд, как и вопрос о раздутости самой системы команд, не очевиден: много команд – это хорошо или плохо? Сложные инструкции – это хорошо или плохо? Попробуем объяснить нашу позицию, используя аналогию. Когда вы разрабатываете программный интерфейс, он должен максимально простым и очевидным способом выражать то, что происходит в вашей системе и отражать те цели, которые вы преследуете. Он не должен быть слишком высокоуровневым или, наоборот, слишком низкоуровневым для ваших задач. Он не должен содержать лишних функций, так как это усложнит поддержку в будущем. Система команд процессора – это квинтэссенция интерфейса между программой и аппаратурой, – и эта квинтэссенция должна содержать только то, что действительно будет реализовано на аппаратном уровне. Поэтому для ответа на этот вопрос следует обратить внимание на существующий опыт и для каждого подобного случая принять взвешенное решение. Если среди целей мы видим переносимость, простоту, обратную совместимость и безопасность, то, вероятно, от комплексных команд лучше отказаться.

2.3 ARMv8

В 2011 году, через год после начала проекта RISC-V, ARM анонсировала полностью переработанную систему команд ARMv8 с 64-битными адресами и расширенным набором целочисленных регистров. В новой архитектуре инженеры ARM убрали функции ARMv7, которые усложняли реализацию: счетчик команд больше не является частью набора целочисленных регистров; предикатных инструкций больше нет; комплексные инструкции с множественной загрузкой и сохранением были удалены, а кодирование команд в целом упрощено. Однако не все проблемы были решены, и, кроме того, добавились новые.

- Условные коды по-прежнему используются в операциях перемещения cmov, что создаёт определённые проблемы для переименования регистров: если условие не выполняется, команда все равно должна скопировать старое значение в новый физический регистр. Это фактически делает условное перемещение единственной инструкцией, читающей три исходных операнда вместо двух.
- ARMv8 не является модульной. Например, SIMD инструкции, предполагающие наличие 32 «жирных» векторных регистров строго обязательны к реализации. Это не подходит для многих встроенных решений, где важны низкие себестоимость и энергопотребление.

В целом ARMv8 является очень большой и напичканной самой разной функциональностью коммерческой системой команд: более 1000 инструкций в 53 форматах, описанные примерно на 6000 страниц документации. При этом, некоторые важные вещи всё же упущены: например, отсутствует какой-либо аналог Thumb (что важно для многих встроенных систем в силу ограниченной памяти для кода), нет объединённых инструкций сравнения + перехода. Кроме того, такая функциональность как размещение констант непосредственно в коде инструкции (так называемый immediate формат инструкции) в обеих версиях ARM ограничена: нет возможности загрузить произвольную 32 битную константу непосредственно из кода программы за 1 или, хотя бы, за 2 инструкции.

2.4 MIPS

Основная проблема MIPS – недостаточная гибкость системы команд. Если x64 и ARM зачастую содержат слишком высокоуровневые инструкции, то в MIPS перекос идёт в другую сторону: команды опускаются до уровня микроархитектуры, предполагая строго определённую реализацию в аппаратуре. Например, результат умножения и деления сохраняется в специальном внутреннем 64 битном регистре. То есть результат нужно получать отдельной командой даже если вам нужны только 32 младших бита.

```
mult  R2, R3
mfhi  R4
mflo  R5
```

По факту старшие 32 бита почти всегда отбрасываются (как вы думаете, какой тип будет иметь результат умножения двух 32 битных чисел в gcc с настройками по умолчанию?), а используется только младшая часть. Здесь мы видим задержку, искусственно введённую в систему команд (ещё одним примером такой задержки в MIPS является так называемая Branch Delay Slot – команда «nop», которая вставляется сразу за командой перехода). Проблему разной длины конвейера для разных команд можно решить, реализовав простой вариант внеочередного выполнения команд (scoreboard), но оставаясь при этом в рамках полного детерминизма и простоты. С другой стороны, наличие этого дополнительного внутреннего состояния усложняет суперскалярную реализацию с внеочередным выполнением команд ([4], стр. 4).

С плавающей точкой в MIPS также не всё хорошо. Обратите внимание, что в MIPS ассемблере инструкции, работающие с двойной точностью, используют только чётные

номера регистров. Так происходит, поскольку числа с двойной точностью занимают 2 смежных регистра – например, пару (f2, f3) при обращении к f2.

```
add.s    $f1, $f0, $f1      # single precision add
sub.s    $f0, $f0, $f2      # single precision sub
add.d    $f2, $f4, $f6      # double precision add
sub.d    $f2, $f4, $f6      # double precision sub
```

Такое решение имеет свои плюсы. Поскольку двойная и одинарная точность обычно не смешиваются в C++ коде, можно более эффективно использовать один и тот же набор физических регистров: либо для того, либо для того. Это контрастирует, например, с x64 и ARMv8, в которых для двойной точности используется половина 128 битного регистра, а для одинарной – одна четверть. Кроме того, преобразования из одинарной точности в двойную могут быть реализованы относительно безболезненно (правда для x64 и ARMv8 это тоже верно). Однако есть и минус. Использование смежных пар регистров усложняет суперскалярную реализацию с внеочередным выполнение команд при реализации переименования регистров: логически смежные после переименования регистры перестают быть таковыми в физическом регистровом файле ([4], стр.6, п.3).

2.5 SPARC

Отличительная черта SPARC это регистровые окна. На первый взгляд идея не такая уж плохая. Вы расширяете фактическое количество регистров при помощи приёма «база + смещение». Пишете R1, а фактически подразумеваете SP+R1. Удобно. Кусочек стека сохраняется на регистрах. При вызове функций не нужно ничего никуда перемещать, да и компилятор становится, вроде как, немного проще.

Однако регистровые окна дороги в плане реализации. Огромное количество регистров, находящихся «в стеке», по факту не используются (напомним, что регистры всё-таки дороже чем L1 кэш-память из-за необходимости многопортовой памяти; например, в Эльбрусе, который основан на SPARC, регистровый файл содержит 20 портов, что на наш взгляд очень много ([6], с.126). Кроме того, когда регистровая память всё-таки заканчивается, наступает апокалипсис в операционной системе ([4], с.6).

Справедливости ради следует сказать, что можно было бы реализовать стек регистров через кольцевой буфер и выполнять параллельно откачку/подкачку неиспользуемой части регистров в память. Но для этого нужно делать ещё больше портов в регистровой памяти! Что-то похожее наблюдается в уже упомянутом Эльбрусе ([6], с.139), но в методичке [6] эта тема на наш взгляд раскрыта недостаточно полно. В итоге современные исследователи сходятся во мнении: если у вас есть лишние транзисторы, лучше просто сделать больше регистров и предоставить компилятору развлекаться со встраиванием функций. Не стоит перекладывать эту задачу на уровень аппаратуры.

Однако есть ещё одна ложка дёгтя. В SPARC существует отдельный набор регистров для плавающей точки (организован аналогично MIPS). Передача между целочисленными регистрами и регистрами с плавающей точкой осуществляется только через память. А ещё есть 8 глобальных регистров. Если процедура использует арифметику с плавающей точкой или меняет глобальные регистры, их нужно сохранять в стеке. Но в каком именно стеке? То есть у нас их теперь два, один в памяти для плавающей точки и глобальных регистров и один на регистрах для целых чисел. Если всё-таки стек только один, то мы должны сначала сохранить данные во временные ячейки памяти только для того, чтобы потом поместить их в наш регистровый стек, где они, кстати говоря, никак не будут использоваться. В результате идея регистровых окон для плавающей точки в SPARC не работает.

2.6 Power

Система команд Power появилась в результате эволюции и развития систем команд, разрабатывавшихся в первую очередь компанией IBM. Ход развития выглядит так:

POWER −> PowerPC −> Power ISA v2.x −> Power ISA v3.x.

С одной стороны, Power – это довольно сложная система команд, которая разрабатывалась главным образом компанией IBM, которая не славится открытостью. С другой стороны, с недавних пор эта архитектура стала полностью открытой на весьма привлекательных условиях [7]. Хотя систему команд Power нельзя назвать простой – чуть меньше 1000 инструкций в 25 форматах (включая векторные инструкции), все же она не столь сложна как ARM. Кроме того, особенностью Power является модульность – каждый регистр относится к функциональному классу, а большинство инструкций внутри класса используют только принадлежащие к этому классу регистры. Только небольшое количество инструкций передают данные между разными функциональными классами. К функциональным классам относятся: условный, с фиксированной точкой, с плавающей точкой и векторный. Таким образом, конкретная реализация может не поддерживать, например, операции с плавающей точкой (как некоторые встроенные решения на архитектуре Power). И «отключить» поддержку оказывается относительно просто. Также такое разделение функциональности в процессоре дает информацию о зависимости между регистрами компилятору. Однако у такого подхода неизбежно появляются и недостатки – взаимодействие между разными функциональными классами может приводить к задержкам, зависимым от реализации. Забегая вперёд, скажем, что этот подход очень напоминает RISC-V и, возможно, именно поэтому разработчики RISC-V ***умалчивают о системе команд Power*** в работе [4].

Отдельно следует упомянуть условные регистры. Также как в ARM и SPARC, в Power есть специальные регистры, в которых выставляются 4 бита состояния после выполнения операций сравнения. Но в Power присутствуют 8 их копий. Каждая из этих копий может быть результатом инструкции сравнения и каждая копия может быть источником ветвления. Такая избыточность применяется, чтобы инструкции могли использовать разные условные состояния без конфликта (когда в разных ветвях кода получаются разные состояния для последующих ветвлений). Кроме того, предоставляется возможность совершать логические операции между этими 4-х битными регистрами, что позволяет одной ветви проверять более сложные условия.

Ещё одна особенность Power – два специальных регистра Link и Count. Регистр Link используется вместо регистра общего назначения (как во многих других архитектурах) для хранения адреса возврата при вызове процедуры. Регистр Count используется как счетчик в циклах с фиксированным числом итераций. Используя этот специальный регистр, аппаратное обеспечение ветвления может быстро определить вероятное ветвление, так как значение регистра известно в начале цикла выполнения. Кроме того, оба регистра могут содержать адрес условной ветви и имеются специальные инструкции, позволяющие получить этот адрес. Подобное решение с одной стороны дает определенные преимущества в скорости при обработке ветвлений, а с другой вносит дополнительную сложность в реализацию.

В системе команд Power есть ряд и других специфичных инструкций – множественной загрузки и сохранения (до 32 регистров), генерации случайного числа в регистре, набор из 48 инструкций для поддержки десятичной плавающей точки и др. В плане векторного расширения можно отметить поддержку 128-битных чисел с фиксированной точкой, а также расширения, добавляющего векторно-скалярные инструкции. Таким образом, Power обладает рядом довольно специфичных особенностей, которые могут быть как плюсом (производительность), так и минусом (сложность). Эта специфичность нашла свое отражение и в существующих применениях и реализациях этой системы команд - суперкомпьютеры (первые две строчки в рейтинге TOP500 занимают компьютеры на базе Power9) и

микроконтроллеры (главным образом в автомобильной и авиационной отраслях). Среди других примеров реализации системы команд Power – одна из компонент процессора Cell Broadband Engine игровой консоли Playstation 3, а также компьютеры Apple Power Mac, выпускавшиеся до 2006 года.

Разнообразие применений Power привело к тому, что одна из основных проблем Power заключается не в системе команд как таковой, а в экосистеме, которая создается пользователями – разработчиками. Узкоспециализированные применения не дают сформироваться широкому сообществу разработчиков, способных разрабатывать эффективные приложения под эту систему команд и способствующих её поддержке и развитию.

Однако в последние годы эта ситуация начала меняться. Появились доступные обычному пользователю персональные компьютеры, использующие процессор Power9 [8], отличающиеся открытостью «прошивок» (firmware) загрузчика и других «сопутствующих» компонентов. Стала открытой сама система команд, появились реализации в FPGA с открытым исходным кодом [9]. А проект по разработке открытого GPU, исходно называвшийся Libre-RISCV [10], и, как следует из названия, ориентировавшийся на RISC-V, переключился на архитектуру Power (причины, правда, скорее политические [11]). Развивается проект по созданию открытого ноутбука на базе PowerPC [12]. Но все же пока что все реализации архитектуры Power в кремнии происходили при непосредственном участии IBM. Получится ли у Power, ставшей открытой, догнать RISC-V или время все же упущено, и IBM, однажды потеряв рынок персональных компьютеров, вновь повторила свою ошибку – будущее покажет. На наш взгляд Power – достойный конкурент RISC-V.

3. Исследование системы команд RISC-V

Система команд RISC-V разрабатывается в университете Berkley. Их основная концепция звучит следующим образом: RISC-V должна стать единой системой команд для всех типов ЭВМ от микроконтроллеров до высокопроизводительных. В действительности, за этим перфекционистским лозунгом стоит ряд практических и очень конкретных целей.

1. **Повышение энергоэффективности и производительности**, уменьшение числа необходимых транзисторов, удешевление разработки и поддержки процессорных ядер за счёт грамотно созданного интерфейса: разработчики процессорных ядер в минимальном варианте обязаны реализовывать лишь относительно небольшой набор инструкций (по сравнению с остальными открытыми и закрытыми системами команд) для получения полноценно работающей системы. После чего они могут сосредоточиться на интересующих именно их областях.
2. **Модульность и расширяемость.** Минимальных наборов команд на сегодняшний день существует достаточно много и RISC-V далеко не самый минимальный из них. Основная проблема существующих систем команд в том, что небольшой набор команд не универсален. Он бывает ориентирован, например, для высокопроизводительных вычислений или областей, требующих специфической аппаратной функциональности (например, многопоточность, графика, криптографические системы). RISC-V призван решить эту проблему благодаря модульной структуре системы команд. Разработчики ЭВМ могут реализовывать существующие расширения (например, RVF32 для работы с плавающей точкой или RVA32 для атомарных операций), либо добавлять свои собственные – система команд спроектирована с прицелом на расширение. То есть предполагается, что вы, как разработчик конкретной системы, будете её расширять. Это в корне отличает RISC-V, например, от ARM, где не только расширяться уже очень проблематично, но и сама компания ARM прямо запрещает подобные расширения по условиям лицензии.

3. **Удешевление разработки** различных электронных систем за счёт создания единой открытой экосистемы: компиляторы, операционные системы, драйверы, периферия. Наличие открытых репозиториев, в которых многое уже реализовано и за счёт многолетней поддержки сообщества доведено до высокого качества, не нуждается в дополнительных комментариях.
4. **Обратная совместимость** программного обеспечения и работа на будущее – программы, разработанные для старых процессоров (вернее, процессоров с поддержкой меньшим количеством расширений), должны работать на новых (вернее, процессорах той же системы команд, но с большим количеством реализованных расширений) без перекомпиляции. Плюс совместимость библиотек на бинарном уровне. То есть, скомпилированная статическая библиотека для одной ОС и одного процессора может быть использована как есть на другой ОС с другим процессором благодаря совместимости на уровне компилятора и собственно процессора. Здесь, конечно, следует сделать одну оговорку. Если в программе вы используете некоторое существующее расширение (например, плавающую точку) или, тем более, какое-то своё специальное расширение, то как есть эта библиотека на другой системе, возможно, не заработает. Тем не менее, благодаря наличию спецификации расширений ситуация поправима! Компилятор может перекомпилировать вашу библиотеку, заменив, например, команды плавающей точки или команды любого другого расширения на вызовы соответствующих функций с программной реализацией. В случае зоопарка коммерческих систем команд вроде x64 или ARM это сделать существенно труднее.
5. **Безопасность #1 (Security).** Если в некотором RISC-V процессоре обнаружена уязвимость, вы можете легко заменить его на другой RISC-V процессор, пусть и не такой эффективный, но зато без закладок потенциального противника. При этом вы сможете запустить на нём всё существующее ПО с минимальными затратами усилий и времени.
6. **Безопасность #2 (Safety).** Программное обеспечение, от которого зависит жизнь людей, проходит специальную процедуру сертификации и иногда формальной верификации, где корректность программы доказывается математически. Стоимость этих процедур может многократно превышать затраты на непосредственную разработку ПО. Однако, даже если вы, например, доказали корректность вашего ядра операционной системы, процедура сертификации – это отдельная проблема и зачастую сменить аппаратуру при этом не представляется возможным. По этой причина система команд Power на сегодняшний день так прочно обосновалась в области гражданской авиации. Стандарт RISC-V призван удешевить процедуру сертификации поскольку система команд и софт по большей части остаются неизменными (Имеется ввиду, что какая-то часть вашей системы будет меняться так или иначе в силу требований того или иного заказчика. Важно, чтобы эта часть оставалась небольшой, тогда затраты на сертификацию будут низкими).

3.1 Базовый набор инструкций RISC-V

Система команд RISC-V имеет следующие отличительные черты:

1. **Отсутствие неявных внутренних состояний.** Результат любой операции (кроме команд перехода) всегда помещается в регистр общего назначения. То есть в RISC-V нет, например, флагов состояний, которые устанавливает инструкция cmp в x86. Вместо этого результат команды сравнения помещается в один из регистров общего назначения. Такой подход существенно упрощает суперскалярную реализацию с внеочередным выполнением команд, но при этом по сравнению с флагами или другими состояниями он не вносит существенных накладных расходов для простых реализаций.

2. **Отсутствие предикатных инструкций.** Вернее, их нет в базовом наборе, но они есть в векторном расширении. Предикатные инструкции совершенно необходимы для VLIW процессоров (Эльбрус) и для векторной обработки данных. Однако, выигрыш в скалярном коде от них обычно не очень большой. Для повышения эффективности ветвлений разработчики RISC-V предлагают добавлять простую реализацию предсказателя ветвлений (т. н. Branch Target Buffer в виде небольшой таблицы адресов перехода), которая дополнительно позволяет избавиться и от Branch Delay Slot. С другой стороны, в отсутствие таких инструкций она упрощает реализацию суперскалярного процессора с внеочередным выполнением команд.
3. **Компактный базовый набор инструкций.** В базовом наборе (рис. 2) всего 11 базовых арифметических инструкций (большинство из них могут встречаться в 2-х формах, давая в сумме 21 инструкцию), 10 инструкций для обращения в память и 8 инструкций для переходов. Итого – 39 инструкций.
4. **Загрузка произвольной 32 битной константы** из памяти инструкций за 2 команды (liu + следующая команда в I-формате). При этом большая часть констант длиной в 12 бит или меньше загружаются из кода программы за 1 инструкцию (собственно I-формат).
5. **32 регистра общего назначения**, что в 2 раза больше чем в большинстве RISC аналогов.
6. **Наличие сжатого 16-битного представления** команды (RV32C) аналогично ARM Thumb для микроконтроллеров.
7. **Поддержка так называемой ослабленной модели памяти** (Relaxed Memory Model) в базовом наборе инструкций (рис. 3). Это не настоящие атомарные операции, для них есть специальное расширение RV32A. Ослабленная модель памяти – это более явное (чем принято традиционно) выражение синхронизации данных между потоками в многопоточной программе. В C++ для этого существует специальная часть стандартной библиотеки (см. std::memory_order). Дело в том, что передача данных между потоками – это в действительности очень непростая и зачастую дорогостоящая операция с учётом многоуровневого кэша в современных процессорах. Традиционно принятая сильная модель памяти, когда любое изменение в памяти, сделанное в одном потоке должно быть немедленно увидено в другом потоке, является одним из краеугольных камней, ограничивающих рост производительности при росте количества ядер. Ослабленная модель памяти помогает программисту более явно выразить его намерения и, например, не выполнять дорогостоящую операцию синхронизации данных через L2/L3 кэш при записи в ячейку памяти, если она не нужна программисту в данный момент.
8. **Наличие спаренных (fused)** операций сравнения + переход, уменьшающие размер программного кода.
9. Инструкции, **ускоряющие вызов функций** (jal), вызов виртуальных функций и выполнение операторов switch (jalr) также присутствуют.

Instruction	Format	Meaning
`add rd, rs1, rs2`	R	Add registers
`sub rd, rs1, rs2`	R	Subtract registers
`sll rd, rs1, rs2`	R	Shift left logical by register
`srl rd, rs1, rs2`	R	Shift right logical by register
`sra rd, rs1, rs2`	R	Shift right arithmetic by register
`and rd, rs1, rs2`	R	Bitwise AND with register
`or rd, rs1, rs2`	R	Bitwise OR with register
`xor rd, rs1, rs2`	R	Bitwise XOR with register
`slt rd, rs1, rs2`	R	Set if less than register, 2's complement
`sltu rd, rs1, rs2`	R	Set if less than register, unsigned
`addi rd, rs1, imm[11:0]`	I	Add immediate
`slli rd, rs1, shamt[4:0]`	I	Shift left logical by immediate
`srli rd, rs1, shamt[4:0]`	I	Shift right logical by immediate
`srai rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate
`andi rd, rs1, imm[11:0]`	I	Bitwise AND with immediate
`ori rd, rs1, imm[11:0]`	I	Bitwise OR with immediate
`xori rd, rs1, imm[11:0]`	I	Bitwise XOR with immediate
`slti rd, rs1, imm[11:0]`	I	Set if less than immediate, 2's complement
`sltiu rd, rs1, imm[11:0]`	I	Set if less than immediate, unsigned
`lui rd, imm[31:12]`	U	Load upper immediate
`auipc rd, imm[31:12]`	U	Add upper immediate to `pc`

Рис. 2. Вычислительные инструкции RISC-V. Целочисленные инструкции из базового набора RISC-V. В действительности различных инструкций всего 11, поскольку большая часть инструкций просто встречается в 2-х форматах – R-формат (стандартный) и I-формат (когда второй операнд считывается прямо из самой инструкции). Следует сказать, что хранить константы в коде – это отличная идея, повышающая производительность. Поскольку инструкции так или иначе уже находятся в чипе, прочитать константу из памяти инструкции ничего не стоит

Fig. 2. Computational instructions of RISC-V. Integer instructions from the RISC-V core set. In reality, there are only 11 different instructions, since most of the instructions are just found in 2 formats – R-format (standard) and I-format (when the second operand is read directly from the instruction itself). It should be said that storing constants in code is a great idea that improves performance. Since instructions are already on the chip one way or another, reading a constant from the instruction memory is worthless

Instruction	Format	Meaning
`lb rd, imm[11:0](rs1)`	I	Load byte, signed
`lbu rd, imm[11:0](rs1)`	I	Load byte, unsigned
`lh rd, imm[11:0](rs1)`	I	Load half-word, signed
`lhu rd, imm[11:0](rs1)`	I	Load half-word, unsigned
`lw rd, imm[11:0](rs1)`	I	Load word
`sb rs2, imm[11:0](rs1)`	S	Store byte
`sh rs2, imm[11:0](rs1)`	S	Store half-word
`sw rs2, imm[11:0](rs1)`	S	Store word
`fence pred, succ`	I	Memory ordering fence
`fence.i`	I	Instruction memory ordering fence

Рис. 3. Инструкции RISC-V для работы с памятью, включая специальные инструкции fence для синхронизации данных в многопоточных программах

Fig. 3. RISC-V instructions for working with memory, including special fence instructions for synchronizing data in multi-threaded programs

Таким образом, исключительно компактный набор менее чем в 40 инструкций содержит внушительный объем полезной функциональности.

3.2 Плавающая точка в RISC-V

Расширение RV32F, добавляющее плавающую точку, добавляет 32 новых регистра. Сразу обращает на себя внимание наличие инструкций преобразования в целые числа и обратно и, кроме того, инструкции непосредственного перемещения данных из целочисленного регистра в регистр с плавающей точкой, что является проблемой для многих существующих систем команд, когда данные приходится перемещать через память. Для двойной точности в RV32D операций перемещения нет, поскольку регистры двойной точности занимают в 2 раза больше бит (тем не менее, они есть в RV64D), но операции преобразования плавающей точки в целочисленную и обратно в RV32D сохраняются. Ещё обращает на себя внимание наличие инструкций FMA (спаренная операция умножения и сложения/вычитания), существенно сокращающих размер кода и повышающих производительность.

3.3 Векторное расширение RISC-V

Система команд RISC-V содержит ещё много интересных расширений. Мы хотим обратить внимание читателя на расширение RV32V, предназначенное для реализации векторных операций. Оно имеет следующие ключевые особенности.

1. **Длина вектора не фиксируется** в наборе инструкций, а задаётся в программе специальной инструкцией. Это в корне отличается от существующих подходов и позволяет вам выполнять на процессорах с небольшой длиной вектора программы, скомпилированные под более широкий вектор. Пусть и с меньшей производительностью. В современных x64 процессорах, например, это не так. Если вы собрали вашу программу, например, с AVX512, то эта программа сможет выполняться только на дорогих моделях Xeon и последних процессорах 109*0X, содержащих широкие векторные регистры zmm.
2. **Предикатное выполнение**, то есть наличие масок во всех векторных операциях. Такой подход одинаково хорош как для обычных CPU программ, так и для реализации массивно-параллельных систем (OpenCL и аналоги).
3. **Векторные регистры** существуют отдельно от остальных регистров, а перемещение данных между скаляром и вектором также поддерживается аппаратно. Благодаря этому в RISC-V отсутствуют неожиданные нюансы в смешанном скалярно-векторном коде, как это есть, например, в x64 (и особенно в x86), где зачастую лучше явно использовать векторный тип __m128 вместо float или int, если вы хотите быть уверены в том, что данные в память из векторных регистров лишний раз не выгружаются.
4. Другие необходимые в векторном коде операции вроде загрузки данных из памяти с определённым шагом, **gather и scatter**, а также поддержка матриц.

В целом можно сказать, что векторное расширение спроектировано с учётом опыта многих существующих архитектур и, как и остальная часть системы команд, выполнено грамотно. Интересно в этой связи отметить такой проект как библиотеку enoki, которая уже сегодня пытается эмулировать похожую функциональность программно на x64 и ARM, а в gcc варьируемая длина вектора в настоящий момент реализована на уровне компилятора (правда в gcc она должна быть кратна 4) [13].

4. RISC-V сегодня

В настоящий момент в сообщество RISC-V входит большое число известных компаний (рис 4). Из наиболее интересных мы бы хотели упомянуть компанию Nvidia, использующую RISC-V и язык Ada для создания беспилотных автомобилей, и компанию Alibaba, которая представила летом 2019 г. первый IP-блок для искусственного интеллекта. Кроме того, RISC-V уже получает популярность на уровне крупных государственных структур: Индия объявила RISC-V национальным стандартом, US DARPA требует RISC-V в качестве обязательного компонента по ряду программ, в том числе – по всему направлению HW

security research, Israel Innovation Authority создает общую платформу GenPro на базе RISC-V, в Китае объявлена объемная программа гос. субсидирования решений на базе RISC-V (2018), а Европейский Союз обсуждает крупную программу высокопроизводительных вычислений на основе RISC-V. Кроме того, RISC-V становится основой учебных программ по Computer Science и Electronic Engeneering направлений во многих университетах. Наконец, RISC-V проник и в Россию [14].

Рис. 4. Некоторые из компаний, участвующих в разработке экосистемы RISC-V и использующих эту экосистему в своих решениях

Fig. 4. Some of the companies involved in the development of the RISC-V ecosystem and using this ecosystem in their decisions

Мы уверены, что можно найти ещё примеры, доказывающие зрелость стандарта. Важно не это, а то, что на сегодняшний момент RISC-V уже имеет сформированную экосистему:

- Open Source ПО: GCC, Linux, BSD, LLVM, QEMU, FreeRTOS, ZephyrOS, LiteOS, SylixOS;
- коммерческое ПО: Lauterbach, Segger, Micrium, ExpressLogic и др.;
- открытые процессорные ядра: Rocket, BOOM, RI5CY, Ariane, PicoRV32, Piccolo, SCR1 (Российское ядро от Syntacore), Hummingbird и др.;
- коммерческие процессорные ядра: Codasip, Cortus, C-Sky, Nuclei, SiFive, Syntacore (Российская) и др.;
- использование RISC-V внутри компании: Nvidia, Western Digital, Qualcom, CloudBear (Российская) и др.

4.1 Перспективы для отечественных разработчиков

Нельзя не упомянуть наиболее известные отечественные процессоры – Эльбрус от МЦСТ и немного менее известные NMC4 (нейроматрикс) от НТЦ Модуль, и «мультикор» от Элвис. Как Эльбрус, так и NMC4 являются процессорами VLIW со своими компиляторами и целиком своей инфраструктурой. С технической точки зрения, на наш взгляд, это очень хорошие решения, особенно когда необходимо добиваться высокой производительности. Например, современный Эльбрус является абсолютным рекордсменом по числу команд в такт (до 50) и на многих задачах побеждает последние новинки от Intel и AMD. Но эти решения дорогие и по большей части закрытые.

Проблема большинства отечественных компаний в том, что их решения не следуют открытым стандартам в достаточной степени, из-за чего даже внутри нашей страны их можно использовать лишь ограниченно. Пересекаясь с отечественными разработками в области ПО для гражданской авиации, мы встречаем нежелание разработчиков связываться с закрытой и специфической экосистемой Эльбруса, как и с любой другой закрытой и малопонятной системой в принципе. При этом необходимо помнить, что для авиации, как и для многих других систем необходимы процедуры сертификации, где закрытые решения недопустимы. Следует сказать и о том, что экспортные системы вооружения зачастую также поставляются с иностранной электроникой, потому что нашей (как и любой другой закрытой системе) заказчики не доверяют.

При всех её преимуществах, к сожалению, VLIW-технология (используемая Эльбрусом и NMC4) никак не состыкуется с идеологией RISC-V, поскольку противоречие заложено в корне: если RISC-V подход *предоставляет интерфейс,* разделяя архитектурный и микро-архитектурный уровни, то VLIW-подход прямо противоположен: система команд формируется непосредственно под микроархитектурный уровень конкретного процессора с конкретным числом блоков определённого типа. То есть, например, если у двух разных VLIW-процессоров разное количество блоков умножения с плавающей точкой (и больше они ничем не отличаются), то компилятор генерирует для них разные программы. С другой стороны, это не значит, что для VLIW-подхода стандартизация и открытость невозможна в принципе. Например, это может быть OpenCL или Vulkan: если рассмотреть выполнение рабочей группы потоков (например, из 256 или 512 элементов) в массивно-параллельной модели как цикл, тогда такой цикл, очевидно, можно конвейеризовать программно. Поскольку в большинстве случаев потоки OpenCL работают над данными независимо, механизм программной конвейеризации циклов должен показать хорошие результаты.

4.2 Недостатки RISC-V

Несмотря на ряд очевидных преимуществ RISC-V по сравнению с аналогами, есть ряд нюансов:

1. Любая организация по стандартизации подобная консорциуму RISC-V (или, например, Khronos) имеет две стороны медали. Одна сторона – это открытость, совместимость и лёгкость вхождения в область для небольших компаний. Другая сторона состоит в ограничении гибкости – предложить своё расширение системы команд позволяется лишь очень крупным спонсорам и даже для них этот процесс не простой. Как мы уже упоминали, проект по разработке открытого GPU, называющийся Libre-RISCV (так и не сменивший название), переключился на архитектуру Power, ссылаясь на невозможность получить необходимую документацию от сообщества. Причины конфликта понятны: разработчики Libre-RISCV предлагали векторное расширение (названное ими Simple-V), которое, по-видимому, было не интересно сообществу RISC-V из-за готовности собственного векторного аналога.
2. Когда речь идёт о высокой производительности, практически во всех материалах по RISC-V видно очень сильное смещение в сторону суперскалярных ядер с внеочередным выполнением команд. С одной стороны, это ожидаемо, поскольку такой способ повышения производительности является логическим развитием идеологии RISC в принципе. С другой стороны, это выглядит как лоббирование определённой технологии с целью продвижения собственных разработок.
3. RISC-V всё-таки не является панацеей, поскольку в текущем виде не может быть основой, например, для VLIW процессоров (и VLIW подход в принципе нигде не упоминается). Кроме того, сообщество RISC-V в подавляющем большинстве случаев умалчивает о графических процессорах, обходясь в презентациях туманными отговорками.

4. По сравнению, например, с Power экосистема RISC-V выглядят менее зрело (хоть и более целостно), поскольку реализаций в кремнии в настоящее время у RISC-V всё-таки существенно меньше, а Power уже давно используется в авиации.

5. Заключение

Современные электронные системы стали настолько сложны, что следование открытым стандартам критически важно для жизнеспособности проекта и его экономической целесообразности. Даже при острой необходимости в наличии специфической аппаратной функциональности нет ни одной причины для того, чтобы не рассматривать открытые стандарты как базу для разрабатываемой технологии. RISC-V – хороший и грамотно спроектированный стандарт, позволяющий решать при разработке программно-аппаратных систем такие проблемы как совместимость (в том числе обратная совместимость в долгосрочной перспективе), безопасность, сертификация, энергопотребление, эффективная реализация многопоточности и удешевление разработки. Это важно прежде всего при разработке центральных процессоров и их окружения (интерфейс памяти, кэш и др.), операционных систем, компиляторов, драйверов и высокопроизводительных библиотек. Если вы начинаете новый проект в одной из этих областей – стоит рассмотреть RISC-V.

Среди аналогов достойным конкурентом является система команд Power (и проект OpenPOWER), которая на деле доказала возможность применения единой системы команд в различных приложениях: от встроенных систем до суперкомпьютеров. На него стоит обратить внимание, если для вас важна зрелость технологии, но, возможно, менее важны затраты на стоимость разработки самого чипа или имеется хорошее готовое решение. MIPS недавно также стала открытой благодаря развитию RISC-V, но её недостатки ограничивают расширение. SPARC – очень устаревшая система команд, не оправдавшая себя сразу в нескольких решениях (прежде всего регистровые окна). Кроме того, есть проблемы с преобразованием плавающей точки в целые числа и обратно, поэтому SPARC – в настоящее время это один из самых плохих вариантов.

Остальные рассмотренные нами популярные системы команд – закрытая интеллектуальная собственность коммерческих компаний, только лишь лицензирование которых стоит от 1 миллиона долларов. Доминирующая на сегодняшний день ARM, имея огромный фактический тираж своих чипов (и выигрывая, в основном, дешевизной отдельных экземпляров) до последнего будет сопротивляться переходу мирового сообщества на открытые технологии, но на наш взгляд это лишь вопрос времени. Кроме того, с развитием открытого программного обеспечения необходимость в поддержке x86/x64 в 2020 году исчезающе мала и велика вероятность того, что со временем персональные компьютеры также перейдут на RISC-V, потому что в x86/x64 на сегодняшний день «просто нет смысла» [5].

Список литературы / References

[1] Tony Albrecht. Pitfalls of Object-Oriented Programming, 2009. Available at: http://harmful.cat-v.org/software/OO_programming/_pdf/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf, accessed 01.04.2020.

[2] Venkataramanan Kumar. [patch][x86_64]: AMD znver2 enablement., 2018. Available at: https://gcc.gnu.org/legacy-ml/gcc-patches/2018-10/msg01982.html?print=anzwix, accessed 01.04.2020.

[3] Michael Larabel. AMD vs. Intel Contributions To The Linux Kernel Over The Past Decade, 2020. Available at: https://www.phoronix.com/scan.php?page=news_item&px=AMD-Intel-2010s-Kernel-Contrib, accessed 01.04.2020.

[4] Waterman A.S. Design of the RISC-V instruction set architecture. Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2016-1, 2016. Available at: https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf, accessed 31.03.2020.

[5]. Michael Slater. AMD's K5 Designed to Outrun Pentium. Microprocessor Report, 1994. Available at: http://cgi.di.uoa.gr/~halatsis/Advanced_Comp_Arch/Papers/k5, accessed 31.03.2020.

[6]. Ким А.К., Перекатов В.И., Ермаков С.Г. Микропроцессоры и вычислительные комплексы семейства "Эльбрус". СПб., Питер, 2013, 272 стр. / Kim A.K., Perekatov V.I., Ermakov S.G. Microprocessors and computing systems of the Elbrus family. St. Petersburg, Peter, 2013, 272 p.

[5] Hugh Blemings. Final Draft of the Power ISA EULA Released, 2020. Available at: https://openpowerfoundation.org/final-draft-of-the-power-isa-eula-released/, accessed 31.03.2020.

[6] Raptor Computing Systems. Talos II – the world's first computing system to support the new PCIe 4.0 standard – also boasts substantial DDR4 memory, dual POWER9 CPUs., 2019. Available at: https://www.raptorcs.com/TALOSII/, accessed 31.03.2020.

[7] Anton Blanchard, Paul Mackerras. Microwatt project on GitHub. A tiny Open POWER ISA softcore written in VHDL 2008, 2020. Available at: https://github.com/antonblanchard/microwatt, accessed 31.03.2020.

[8] Luke Kenneth Casson Leighton, Yehowshua Immanuel, Jacob Lifshay and others. Libre-RISCV GPU project., 2019. Available at: https://libre-riscv.org/3d_gpu/, accessed 31.03.2020.

[9] Luke Kenneth Casson Leighton, [libre-riscv-dev] power pc, 2019. Available at: http://lists.libre-riscv.org/pipermail/libre-riscv-dev/2019-October/003035.html, accessed 31.03.2020.

[10] Open Hardware GNU/Linux PowerPC notebooks, 2020, Available at: https://www.powerpc-notebook.org/en/, accessed 31.03.2020.

[11] GCC Manual. , 6.49. Using Vector Instructions through Built-in Functions, 2019, Available at: https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Vector-Extensions.html, accessed 01.04.2020.

[12] Крсте Асанович, Александр Редькин и др. Технический симпозиум RISC-V Moscow, 2019 / Krste Asanovic, Alexander Redkin and others. Technical Symposium RISC-V Moscow, 2019. Available at: https://riscv.expert/, accessed 01.04.2020.

Информация об авторах / Information about authors

Владимир Александрович ФРОЛОВ – кандидат физико-математических наук, старший научный сотрудник Института прикладной математики им. М.В. Келдыша РАН, научный сотрудник факультета Вычислительной математики и кибернетики Московского университета. Сфера научных интересов: реалистичная компьютерная графика, моделирование освещённости, разработка программных систем оптического моделирования, параллельные и распределённые вычисления.

Vladimir FROLOV – PhD in computer graphics, senior researcher in the Keldysh Institute of Applied Mathematics of Russian Academy of Sciences and researcher in computer graphics of Moscow State University. Research interests: realistic computer graphics, light transport simulation, elaboration of optical simulation software systems, GPU computing.

Владимир Александрович ГАЛАКТИОНОВ – доктор физико-математических наук, профессор, заведующий отделом компьютерной графики и вычислительной оптики Института прикладной математики им. М. В. Келдыша РАН с 2003 г. Сфера научных интересов: компьютерная графика, оптическое моделирование, создание программных систем оптического моделирования.

Vladimir GALAKTIONOV – Doctor of Science in physics and mathematics, Professor, Head of Computer graphics department in the Keldysh Institute of Applied Math of RAS since 2003. Research interests: computer graphics, optical simulation, elaboration of optical simulation software.

Вадим Владимирович САНЖАРОВ – младший научный сотрудник факультета Вычислительной математики и кибернетики Московского университета. Сфера научных интересов: компьютерная графика, системы фотореалистичного синтеза изображений, параллельное и распределённое программирование.

Vadim SANGAROV – junior researcher in computer graphics at Moscow State University. Research interests: realistic computer graphics, elaboration of optical simulation software systems, concurrent and distributed computing.