Особенности задания значений регистров при программировании для RISC-V

Евгений Ерёмин

Всё чаще появляется информация о перспективах использования в разрабатываемых отечественных микропроцессорах архитектуры под названием RISC-V. К сожалению, её подробности, особенно на русском языке, описаны недостаточно. Между тем разработка программного обеспечения для RISC-V имеет определённую специфику. В частности, в данной статье подробно рассматривается проблема задания непосредственных операндов (констант) в инструкциях RISC-V. Показано, что эти широко распространённые действия в RISC-V имеют некоторые не слишком удобные для программиста особенности. На конкретных примерах описывается оптимальный выбор необходимых инструкций. Материалы статьи будут полезны при освоении технологии RISC-V, особенно тем будущим специалистам, которые не имеют предварительного опыта работы с программами на уровне языка микропроцессоров.

«... Очень трудно навскидку сказать, что измеряется такими величинами. Фактически 32-разрядное целое в большинстве случаев используется меньше, чем наполовину, – старшие биты оказываются незадействованными».

К. Касперски [1]

«Как правило, операнд является целым числом в дополнительном коде, и левый разряд поля операнда в команде является знаковым. При загрузке операнда в регистр знак расширяется влево для заполнения свободных разрядов регистра по обычному правилу».

В. Столлингс [2]

Введение

В последнее время большой интерес вызывает стандарт на систему команд процессоров под названием RISC-V [3, 4]. Всемирно известный Массачусетский технологический институт внёс RISC-V в список десяти прорывных технологий 2023 года [5]. Принципиальная особенность состоит в том, что любой желающий может пользоваться ею **бесплатно** вплоть до коммерческих реализаций непосредственно в кристалле. Кроме того, RISC-V – это **открытый** стандарт, т.е. предусмотрены возможности его расширения под конкретные задачи.

Разработку RISC-V начала в 2010 году команда учёных из университета Беркли под руководством профессоров Крсте Асановича и Дэвида Паттерсона. Последний широко известен не только как разработчик RISC-архитектуры SPARC, но и как автор классических книг по устройству компьютеров. Первоначально спроектированный для исследований компьютерной архитектуры и применения в образовании, RISC-V постепенно стал стандартом свободной и открытой архитектуры для промышленного применения. В 2015 году возник некоммерческий консорциум RISC-V Foundation, целью которого является создание открытого сообщества разработчиков программного и аппаратного обеспечения, основанного на RISC-V. В консорциум входят множество компаний, в том числе AMD, Google, Hewlett Packard Enterprise, IBM, Microsoft, NVIDIA, Oracle и другие. В 2022 году к ним присоединилась Intel. Большую роль в развитии RISC-V играют китайские производители и организации, ведущей среди которых является Китайская академия наук.

В 2022 году в России образован самостоятельный Альянс RISC-V, призванный развивать и популяризировать архитектуру RISC-V в нашей стране. Как написано на официальном сайте Альянса (https://riscv-alliance.ru), «основная цель российского Альянса – создание открытого сообщества разработчиков программного и аппаратного обеспечения, контролируемого участниками сообщества для дальнейшего развития архитектуры RISC-V в России».

В нашей стране налажено производство нескольких типов микроконтроллеров с архитектурой RISC-V. Наиболее известный из них под названием «Амур» (К1948ВК018) предназначен для широкого круга применений в устройствах автоматики. Другой микроконтроллер К1986ВК025 (и его модификации) позициониро-

ван более узко как устройство для приборов и систем учёта энергоресурсов. Несмотря на относительную скромность возможностей, эти чипы могут рассматриваться как первый успешный шаг на пути создания полноценных отечественных микропроцессоров. В то же время ограниченность ресурсов выпускаемых микросхем делает полезными понимание основных идей их программирования на низком уровне.

По нашему мнению, особую актуальность рассмотрению вопросов, связанных с RISC-V, придаёт недостаточное количество материалов по данной архитектуре, особенно на русском языке. Оригинальное английское изложение стандарта хотя и достаточно подробно для технических специалистов, но читается тяжело; часто многие очевидные авторам-разработчикам детали пропущены. Например, довольно трудно самостоятельно догадаться, зачем таким хитрым способом переставляются биты в смещении у команд переходов, а ведь там есть определённый смысл [6].

В качестве источников, помогающих разобраться в стандарте, прежде всего упомянем авторский Атлас [7] и диссертацию одного из разработчиков стандарта Э. Уотермана [8]. В широко известных книгах [9] и [10], переписанных недавно под RISC-V (последняя переведена на русский язык), про процессор рассказывается «попутно» – хорошо, понятно, но только те детали, которые связаны с изложением основной темы книги. Из удачных вводных материалов на русском языке хотелось бы также порекомендовать читателям обзор [11] и описание в Сети [12].

В данной статье будет детально рассмотрена простая на первый взгляд проблема программного занесения в регистр микропроцессора требуемого значения. Но, как оказывается при более подробном рассмотрении, при задании некоторых величин могут возникать трудности. Они специфичны для RISC-V, так что познания по другим процессорам не всегда помогают, о чём свидетельствуют встречающиеся на форумах вопросы.

Что такое RISC-V?

Рассмотрение начнем с названия. Аббревиатура RISC (*Reduced Instruction Set Computer* – компьютер с сокращённой системой команд) введена давно и описывает компьютер, у которого набор команд содержит только самые необходимые, зато очень быстро выполняемые операции (особенности сокращённого набора будут обсуждаться далее). Технология RISC зарекомендовала себя настолько хорошо, что даже создатели более ранней альтернативной ветви процессоров CISC (*Complex Instruction Set Computer* – компьютер с полной или, как иногда переводят, «сложной» системой команд), таких как Intel, сейчас используют внутри тщательно оптимизированное RISC-ядро.

Что касается последнего символа V в названии RISC-V, то это римская цифра пять, так что по-русски, видимо, правильно читать имя как «RISC пять» (или, возможно, «RISC пятый»). «Пятёрка» означает [3, 4] порядковый номер набора RISC-инструкций, разработанного в университете Беркли в США (предыдущие четыре: RISC-I, RISC-II, SOAR и SPUR). С другой стороны, авторы стандарта упоминают, что одновременно символ V ссылается на слова «Variations» и «Vectors», намекая на одну из целей разработки: исследование разнообразных компьютерных архитектур.

Как сокращают систему команд

«Областью, в которой “встречаются” программист и конструктор компьютеров, является набор машинных команд» [2]. В литературе часто используется английский эквивалент термина: ISA – *Instruction Set Architecture*.

С одной стороны, инженерам хочется, чтобы набор инструкций был как можно проще: тогда его легче реализовать и оптимизировать. Напротив, программисты желают видеть как можно больше всевозможных аппаратно реализованных инструкций, причём чем каждая из них будет более вариативной и мощной, тем удобнее: можно быстрее написать крупную программную систему и сделать её компактнее. В этом и заключается корень проблемы. По сути, речь идёт о **границе раздела** «полномочий» **аппаратной и программной** частей.

Первоначально программисты писали программы на языках, близких к машинному. Поэтому было удобно, когда система команд процессора хорошо развита и разнообразна. По сравнению с «ручными» вычислениями машины получали результат очень быстро, а значит, требования к оптимизации были тогда не особенно высокими. Но со временем ситуация существенно менялась. Росли возможности языков программирования высокого уровня, писать программы на них стало значительно легче и быстрее. Как следствие, всё большее количество программистов отдалялось от машинного языка, так что требования к удобству низкоуровневой системы команд компьютера отодвигались на задний план. Зато требования к производительности компьютера, наоборот, выросли, в том числе и потому, что по мере увеличения количества промежуточных программных слоёв происходит потеря эффективности результирующего кода.

Казалось бы, чем мощнее инструкции в системе команд процессора, тем легче получить эффективный код. Но, как показывает кропотливый статистический анализ кода транслированных программ, его основная часть состоит главным образом из простых команд [2]. Секрет, видимо, в том, что стремление к универсальности плохо сочетается со спецификой сложных и уникальных инструкций. Следовательно, и с этой точки зрения сильно усложнять систему команд тоже не требуется, напротив, вполне можно обойтись усечённым набором самых необходимых инструкций. Большой исторический опыт конструирования вычислительной техники де-факто сформировал и «отточил» компактный универсальный набор элементарных инструкций, на базе которого может быть реализована любая программа.

Таким образом, мы видим, что эволюция компьютеров естественным путём приводит к изменениям в построении набора машинных инструкций. Ради получения более высокой производительности от полной системы команд CISC происходит переход к упрощённой RISC. В частности, на основе анализа опубликованных данных в [13] сделан вывод о том, что при равной тактовой частоте производительность RISC по сравнению с CISC приблизительно удваивается.

Кроме того, есть ещё одно мощное положительное следствие из рассматриваемого RISC-подхода – упрощение электронной схемы процессора.

В типовой СБИС CISC-процессора примерно половина площади кристалла занята памятью микропрограмм для инструкций, а в микросхеме процессора RISC-I узел управления занимает только 6% площади микросхемы [2].

Как же следует строить набор инструкций RISC-процессоров, чтобы оставались только самые простые и быстрые инструкции? Здесь можно отметить следующие базовые идеи.

Известно, что наиболее медленные операции – это обмен данными с памятью, поскольку он требует согласования работы процессора с внешним для него ОЗУ. Отсюда во всех RISC-процессорах основная часть команд работает только с небольшой собственной внутренней памятью – *регистрами* (часто используется термин *регистровый файл*). Желательно, чтобы регистры были в достаточном количестве.

Для общения с ОЗУ тщательно разрабатывается несколько специализированных инструкций. В литературе их принято называть терминами *LOAD* и *STORE* – загрузить в регистр и сохранить из него. Попутно заметим, что рассматриваемые операции имеют довольно ограниченные возможности в адресации данных по сравнению с CISC-машинами, опять же ради повышения производительности.

Ещё одно важное свойство, помогающее ускорить выполнение операций в RISC, заключается в том, что все инструкции имеют стандартизированную кодировку: размер инструкций и их структура (*формат*), за редким исключением, одинаковы, что упрощает и ускоряет расшифровку.

Предположим, нам удалось разработать для RISC-процессора компактный набор инструкций. А как быть с теми командами, которые были «сокращены» при конструировании? Очень просто: их надо составлять из нескольких имеющихся. В частности, стержневая тема данной статьи как раз и состоит в изучении оптимальных комбинаций инструкций для одного из наиболее распространённых программных фрагментов.

Наконец, обсуждая вопрос о построении системы команд, нельзя не отметить, что в RISC-V заложена оригинальная стратегия: имеется небольшой обязательный набор инструкций, который можно расширять дополнительными стандартными модулями или даже разрабатывать свои. Например, в базовом наборе нет операций умножения и деления, но при необходимости можно добавить к нему расширение **M**, где они все сгруппированы. Аналогичным образом, если потребуется, в ISA включаются стандартные модули обработки вещественных чисел **F** (обычная точность) и **D** (двойная).

Использование констант в инструкциях

В программах часто используются некоторые неизменяющиеся значения (константы). Поэтому в инструкциях процессора полезно предусмотреть механизм для задания константы непосредственно в коде самой инструкции. Такой метод, когда обрабатываемое значение находится внутри команды и не требует дополнительного обращения к памяти, принято называть *непосредственной адресацией* данных. Саму константу обычно называют *непосредственным операндом* (англ. – *Immediate*). Достоинства очевидны: после считывания команды константа немедленно доступна для использования. Недостаток в непосредственном методе адресации тоже имеется: величина константы ограничена, поскольку она является одной из составных частей полной инструкции процессора. Как правило, в современных компьютерах длина поля под константу значительно меньше длины всего слова.

Хотя известны способы сделать размер константы произвольным, они не подходят для RISC-процессоров. Так, в 64-битных процессорах семейства Intel константа может иметь размер 1, 2, 4 или 8 байт [14]. Разумеется, длина команды при этом будет зависеть от размера константы, что противоречит принципу единства размера инструкций и их структуры, принятому в RISC-технологии. Другой способ справиться с ограничением и получить «полноразмерную» константу применялся в некогда распространённом семействе машин PDP. Там константа не входила в явном виде внутрь кода инструкции: собственно инструкция помещалась в первое слово, а константа – в следующее [15]. Такой способ для технологии RISC тоже не подходит, поскольку он нарушает однозначность чтения инструкций из памяти.

Для решения описанной проблемы в RISC любая константа обычно делится на две части: старшую и младшую, причём они не обязательно одинаковы по размеру. Каждая часть заносится в регистр с помощью отдельной инструкции. В итоге вместо одной команды в программе получаем две, но это как раз допускается RISC-идеологией.

Важно учитывать, что константы бывают как положительные, так и отрицательные. Поэтому в вычислительных устройствах широко применяется так называемое **правило расширения знака**: в случае отрицательных значений дополняемые слева разряды заполняются не нулями, а единицами. Часто говорят, что знаковый (по сложившейся традиции старший) бит «укороченной» константы расширяется до полного размера, иначе говоря, знак копируется во все добавленные разряды (см. второй эпиграф).

Расширение знака у непосредственного операнда часто осложняет работу с последним. Поэтому, например, в RISC-компьютере MIPS логические операции AND, OR и XOR специально объявлены беззнаковыми [16] и **на константы в них правило расширения знака не распространяется**. Работает это следующим образом.

Пример 1

Пусть требуется сформировать в регистре $s0 32-битное шестнадцатеричное значение 0x1234AA77. В MIPS полное значение делится на две половинки по 16 бит. В результате, используя исключительный характер логической операции OR, можно получить требуемую константу двумя инструкциями:

LUI $s0, 0x1234;
ORI $s0, 0xAA77.

Первая из них загружает в регистр $s0 старшие 16 бит, а вторая – младшие. Отметим, что команда **LUI** младшие биты в $s0 сбрасывает в ноль, так что в них в результате операции **ORI** всегда получится значение непосредственного операнда.

Хотя младшая половина константы в данном случае содержит в своём самом старшем бите 1, расширение знака операнда не происходит благодаря принятому для инструкции **ORI** исключению из общего правила. Забегая вперед, скажем, что в RISC-V «отключить» действие расширения

Рис. 1. Формирование константы из двух частей

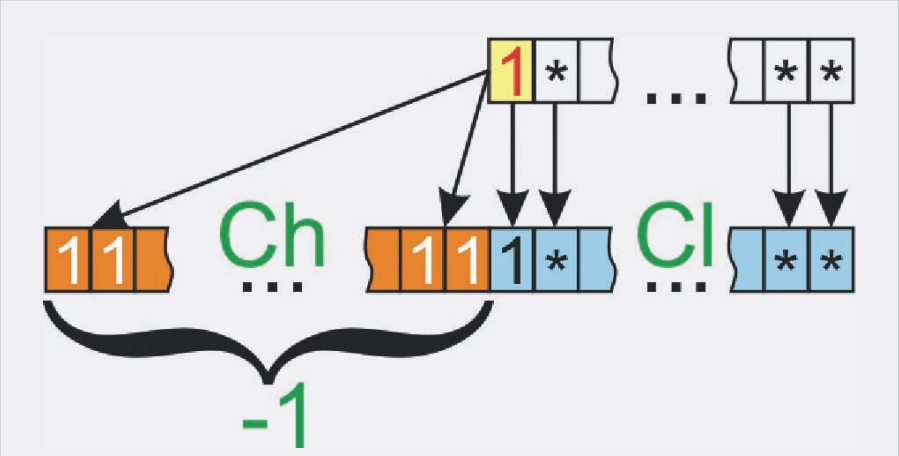

Рис. 2. Влияние расширения знака на формирование константы

знака в операциях над регистрами невозможно.

Аналогичным образом устроена система команд в RISC-архитектуре PowerPC [2]. А вот в компьютере SPARC, чтобы избежать учета влияния знака, принята другая хитрость [17]. Там константа для задания старшей («верхней») части регистра содержит 22 бита, а младшая – 13. Нетрудно видеть, что 22 + 13 > 32, т.е. имеет место запас, который позволяет при задании младшей части вообще не пользоваться знаковым битом.

Посмотрим теперь, как формируются константы в RISC-V.

Константа в RISC-V: постановка задачи

Будем рассматривать базовую архитектуру стандарта **RISC-V**, которая маркируется как RV32I: набор RISC-V инструкций для работы с 32-битными целочисленными данными. Регистры для таких данных, что очевидно, тоже будут иметь разрядность **32 бита**.

Система команд стандарта построена так, что многие инструкции имеют вариант, оперирующий с непосредственной константой. Возьмём для примера часто используемую инструкцию сложения с константой **ADDI** (**Add I**mmediate):

ADDI x7, x6, 8.

Она прибавляет к числу из регистра x6 константу 8 и результат помещает в регистр x7 (т.е. x7 = x6 + 8). Согласно стандарту под хранение непосредственного операнда в любой команде выделяется **12 бит**. Общее количество возможных значений, таким образом, 2^{12} = 4096. Из них ровно половина – отрицательные числа. Учитывая, что есть нулевое значение, окончательно имеем диапазон от –2048 до 2047 включительно.

Итак, в регистре 32 бита, а в константе – 12. Для заполнения 20 старших бит в системе команд RISC-V предусмотрена специальная инструкция **LUI**. Она загружает (**L**oad) в верхнюю (**U**pper) часть регистра константу, входящую непосредственно в код команды (**I**mmediate). Константа как раз имеет размер 20 бит. Младшие 12 бит при записи в регистр гарантированно обнуляются. Например, инструкция

LUI x31, 1

занесёт в регистр x31 число 1000_{16} = = 4096.

Объединяя **LUI** с **ADDI**, можно обеспечить доступ ко всем 20 + 12 = = 32 битам регистров. Сразу подчеркнём, что наличие отрицательных значений осложняет картину. Но об этом немного позже.

Для удобства обсуждения введём обозначения. Полную 32-битную константу, значение которой мы будем формировать в регистре, обозначим ***C***. Она образуется из двух неодинаковых по размеру частей. Старшие 20 бит (здесь и далее будем использовать полное обозначение $\mathbf{Ch}_{20}$, или просто **Ch**) задаются с помощью команды **LUI**, а младшие 12 ($\mathbf{Cl}_{12}$, или **Cl**) прибавляются посредством **ADDI**. Для большей наглядности процесс формирования константы в RISC-V изображён на рис. 1.

Будем пользоваться стандартной нумерацией бит справа налево, начиная с нуля. Тогда $\mathbf{Cl}_{12}$ состоит из бит с 11-го по 0-й, а $\mathbf{Ch}_{20}$ содержит биты константы с 31-го по 12-й. Формально можно считать, что знак $\mathbf{Cl}_{12}$ определяется 11-м битом константы. При расширении знака слагаемого значение этого бита заносится во все биты 31–12.

Далее ради наглядности будем писать **шестнадцатеричные** значения констант, подчёркивая основание системы счисления обозначением h после них. Например, 10h.

Примечание. Это не совсем по стандарту ассемблера, но попытки использовать общепринятую запись 0x10 делают инструкции RISC-V, где регистры **тоже маркируются буквой x**, плохо читаемыми: имена регистров трудно отличать от констант.

Варианты задания констант: примеры

Особенности формирования констант в RISC-V отчётливо видны на примерах.

Пример 2

Пусть требуется занести в регистр x6 положительное значение, не превышающее 7FFh, например, 20h. Очевидно, что для размещения данного небольшого числа в $\mathbf{Cl}_{12}$ вполне достаточно бит, поэтому хватает одной инструкции:

ADDI x6, x0, 20h.

В RISC-V содержимое x0 всегда гарантированно равно 0, так что результат получается 0+20 = 20h. Вычисленная константа будет записана в x6.

Мы знаем, что $\mathbf{Cl}_{12}$ определяет 12 младших бит числа. Старшие биты определяются по правилу расширения знака, которое было сформулировано выше. В нашем случае число положительно, поскольку его 11-й бит нулевой, следовательно, все 20 старших бит регистра x6 автоматически обнулятся и получится полно-

Таблица. Варианты формирования константы

Вариант	Пример	Значения *C*	Ch_{20}	Cl_{12}	Команды	Коррекция Ch+1
v1	2	0 ≤ C ≤ 7FF	= 0	≥ 0	ADDI R, x0, Cl	Нет, и нет LUI
v2	6	F...F800 ≤ C ≤ F...F	+1 = 0	< 0		Да, но нет LUI
v3	5	800 ≤ C ≤ FFF	= 0	< 0	LUI R, Ch+1 ADDI R, R, Cl	Да
	4	1000 < C < F...F800	≠ 0			
	3		≠ 0	> 0	LUI R, Ch ADDI R, R, Cl	Нет
v4	7	C = 1000, 2000, ...	≠ 0	= 0	LUI R, Ch	Нет

ценное 32-разрядное положительное число 20h.

Пример 3

А теперь занесём в этот же регистр большее положительное значение, например, 1020h. Здесь уже потребуются обе части: $\mathbf{Ch_{20}}$ = 1 и $\mathbf{Cl_{12}}$ = 20h. При разделении числа на составляющие учитываем, что 12 бит – это три последние **шестнадцатеричные** цифры в записи константы. В данном случае для формирования константы потребуются две инструкции.

LUI x6, 1;
ADDI x6, x6, 20h.

Первая определяет старшие биты $\mathbf{Ch_{20}}$, а вторая прибавляет к полученному в x6 значению $\mathbf{Cl_{12}}$. Полная запись сложения выглядит так: 00001 000 + + 00000 020 = 00001 020. Здесь **Ch** и **Cl** для наглядности разделены пробелом.

Но, к сожалению, не всегда дело обстоит так просто.

Пример 4

Попробуем занести в регистр x6 (можно было бы выбрать и другой) константу 1800h по схеме, предложенной в предыдущем примере. Получим инструкции LUI x6, 1 и ADDI x6, x6, 800h (специально не оформляем это как кодовый фрагмент, так как он будет работать не совсем правильно). Здесь **Ch** = 1, а **Cl** = 800h. Когда процессор RISC-V выполнит эти инструкции, то ответ получится не 1800h, как ожидалось, а всего лишь 800h. Подвох заключается в том, что по принятым правилам будут складываться числа 00001 000 + FFFFF 800 = 00000 800 (у второго слагаемого 800h автоматически расширяется знак – рис. 2).

Как это исправить? Обратим внимание, что в старших 20 битах после расширения знака **Cl** вместо нулей устанавливаются единицы. Фактически старшая часть операнда имеет шестнадцатеричное значение FFFFF, т.е. в общепринятом десятичном представлении –1. Следовательно, из **Ch** неявным образом вычитается 1. Если это заранее предусмотреть и скорректировать **Ch**, предварительно увеличив его на 1, то ответ станет правильным.

LUI x6, 2;
ADDI x6, x6, 800h.

Приведённый скорректированный фрагмент обеспечит занесение в регистр x6 требуемой константы 1800h.

Примечание. По нашему мнению, наиболее противоестественно здесь то, что значение исходной константы требуется **подменять** ради получения правильного итогового результата. Кстати говоря, это не только впечатление автора. Так, в [10] не без иронии сказано, что обязательное расширение знака непосредственного операнда, принятое в RISC-V, «выглядит довольно необычно». Перед нами типичный пример «встречи» программиста и конструктора машины, о которой говорилось выше, причём все выгоды тут явно получил конструктор.

Подчеркнём, что коррекция **Ch** требуется, только когда старший бит **Cl** (с номером 11) равен 1. В стандарте [4] можно найти красивую неявную рекомендацию: прибавлять к **Ch** старший бит **Cl** (см. описание псевдоопераций на с. 139). Действительно, когда старший бит равен 1, коррекция **Ch** будет, а когда он 0 – нет, поскольку прибавление нуля ничего не меняет. С теоретической точки зрения выглядит очень изящно, но потребуются дополнительные инструкции выделения бита и его сдвига. Сравните такой алгоритм с принятым в теоретической RISC-модели Д. Кнута [18], где загрузка каждой из частей константы никак **не влияла** на все остальные биты. Целесообразно также ещё раз вернуться к примеру 1 и убедиться, насколько в принятой там ISA задание констант устроено нагляднее и проще.

Пример 5

Рассмотрим ещё один выделяющийся вариант. Попробуем занести в регистр значение 800h = 2048. Формально следуя описанному в предыдущем примере алгоритму, немедленно получаем работоспособный фрагмент:

LUI x6, 1;
ADDI x6, x6, 800h.

Тем не менее присмотримся к ситуации повнимательнее. Сразу бросается в глаза, что для формирования 12-битной константы с формально нулевым **Ch** невозможно обойтись без предварительной операции **LUI**. Причём подчеркнём, что речь здесь идёт не об одном значении, а о целом диапазоне чисел от 800h до FFFh (т.е. от 2048 до 4095 в десятичном виде). Ещё более печально, что трудности возникают не только с числами. Как известно, для выделения значений отдельных бит при помощи логической операции AND используются константы-маски, содержащие единственную единицу в нужном бите. Это не какая-то экзотика: подобный приём часто требуется для анализа отдельного управляющего бита в регистре состояния внешнего устройства. Так вот, оказывается, что любую подобную маску, кроме 800h, удаётся получить одной инструкцией: биты от 0 до 10 можно установить посредством **ADDI**, а с 12 по 31 – с помощью **LUI**. И только для «несчастного» 11-го бита требуется нестандартный вариант из двух инструкций. А чем, собственно, отличается этот бит от всех остальных? Ничем, кроме заложенных в формат инструкций правил!

Пример 6

Интересный и важный для практики частный случай имеет место при небольших отрицательных значениях. Рассмотрим, следуя [19], значение –1 = FFFFF FFFh. Здесь **Ch** = FFFFFh и **Cl** = FFFh. По общей схеме примера 4 получим инструкции LUI x6, 0 и ADDI x6, x6, FFFh. Первая из них явно бесполезна (что, кстати, в [19] не указывается). Несколько подправив вторую, получим оптимальное решение

ADDI x6, x0, FFFh

(напоминаем, что x0 всегда равно 0). Правильные (единичные) значения старших бит будут установлены за счёт расширения знака: здесь оно сыграет положительную роль.

Сравнивая данный вариант с примером 2, приходим к выводу, что в

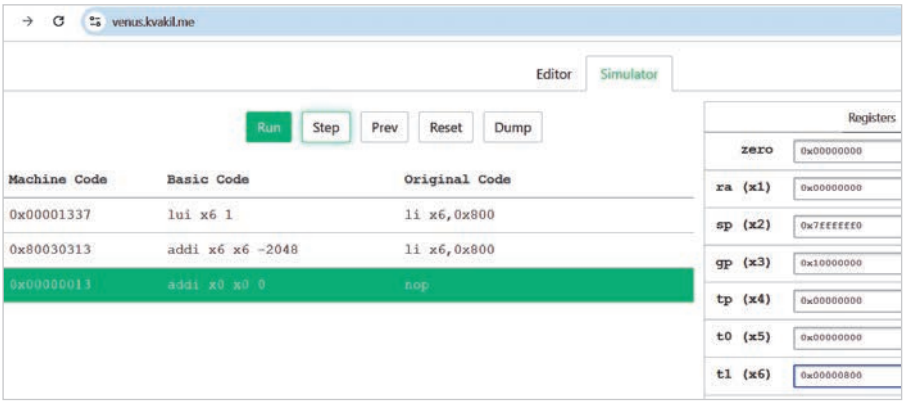

Рис. 3. Реализация примера 5 в симуляторе Venus

случае, когда константа **по модулю** (!) невелика, для её записи в регистр удаётся обойтись одной инструкцией **ADDI** независимо от знака. Это приятный вывод, поскольку чаще всего в программах требуются именно такие «небольшие» константы.

Пример 7

Рассмотрим ещё один вариант оптимизации. Существуют числа, содержащие в младшей части много нулей. Если в константе последние 12 бит нулевые, то **Cl** = 0, и прибавлять её нет смысла. Пусть, например, нам требуется занести в x6 константу 1000h. Тогда достаточно написать **LUI x6, 1.**

Данная оптимизация, в отличие от предыдущей, встречается довольно редко. Но зато если **Cl** = 0, то вторую инструкцию можно пропустить.

Подведём некоторые итоги по описанным выше примерам. Все возможные варианты формирования значения в регистре R (вместо R можно подставлять любой из регистров от x1 до x31) сведены в таблицу.

В вариантах v1 и v2 используется только команда **ADDI**, в v4 – только **LUI**, а в общем случае v3 требуются оба слагаемых. Обращаем ваше внимание на разницу в инструкции **ADDI** в вариантах v1 и v3.

Псевдоинструкция LI

Для удобства программирования в языке ассемблер часто вводятся так называемые псевдоинструкции. *Псевдоинструкция* – это запись некоторой формальной (не существующей в системе команд процессора) инструкции, которая заменяется одной или, реже, несколькими реальными инструкциями. Пусть, например, требуется скопировать содержимое x6 в x7. Инструкция копирования в явном виде отсутствует в RISC-V, но указанное действие может быть легко выполнено как

ADDI x7, x6, 0.

Записанную специфическую операцию сложения легко понять: к x6 прибавляется константа 0 (т.е. фактически значение не меняется!), и результат помещается в x7. Ради удобства программирования и для повышения наглядности можно договориться вместо такой операции ввести псевдоинструкцию **mv**, которая в нашем случае примет вид

mv x7, x6.

Встречая в тексте псевдоинструкцию **mv**, транслятор легко преобразует её запись в текст реально существующей команды **ADDI** (подобное преобразование исходного текста программы принято называть *препроцессинг*, т.е. обработка **перед** процессом трансляции).

Примечание. Интуитивное представление, что сложение будет выполняться дольше, чем простое копирование данных, как ни странно, не соответствует действительности. Даже в довольно старой модели процессора Intel 80386 команды **ADD** и **MOV** уже выполняются за одинаковое число тактов [20].

К теме нашего обсуждения имеет непосредственное отношение псевдоинструкция **li** (**L**oad **I**mmediate). Вот конкретный пример:

li x6, 9999h.

Эту псевдоинструкцию принципиально невозможно заменить на какую-либо одну инструкцию RISC-V, потому что константа «не помещается» в отведённые форматом 12 бит. Поэтому транслятор вынужден подобрать эквивалентную последовательность команд, которая выполнит требуемое действие.

Вместо раскрытия псевдооперации **li** в документации стандарта [3, 4] записана «обтекаемая» фраза «Myriad Sequences», что можно перевести как «множество последовательностей». Очевидно, что речь идёт об отсутствии единого универсального решения для произвольного аргумента. В более подробном Атласе [7] имеется намёк на устройство таких последовательностей: они используют «как можно меньше инструкций» и расширяются для RV32I как «**lui and/or addi**» (про RV64I речь пойдёт в следующем разделе). По сути, это такое «архивированное» описание приведённой выше таблицы, вполне понятное посвящённым.

Наличие псевдоинструкции **li** освобождает программиста от необходимости следить за особенностями кода констант. Так, в недавно вышедшей книге по ассемблеру RISC-V [21] расширение знака в константах вообще не рассматривается. Тем не менее особенности, связанные с расширением знака в процессоре, от этого не исчезают. Поэтому, по мнению автора, утверждение «введём псевдоинструкцию **li** и забудем о деталях задания констант» некорректно: помнить стоит хотя бы потому, что в других операциях, где псевдоинструкции не предусмотрены, программисту тоже будет необходимо грамотно задать непосредственный операнд (при этом, возможно, потребуется модификация по правилам, описанным выше). В частности, именно такая ситуация возникает в логических инструкциях, когда в маске в 11-м бите находится 1.

И ещё одно замечание. Препроцессинг, по своей сути, есть обработка текста. Псевдоинструкция **li** заметно сложнее: надо выделить в строке текстовую запись константы, перевести её в двоичный код, разделить его на части и по получившимся значениям **выбрать способ** и сформировать подходящий к конкретному случаю фрагмент **из одной или двух строк**. Между прочим, перед этим потребуется перевести старшую и младшую части двоичного числа обратно в текстовую форму. Таким образом, действия для псевдоинструкции **li** выходят далеко за рамки задачи преобразования одного текста в другой.

Чтобы убедиться в правильности приведённого выше описания механизмов формирования констант, можно взять какое-то профессионально написанное программное обеспе-

чение и убедиться, что для загрузки константы в регистр действительно может потребоваться несколько команд. Автору кажется самым простым, быстрым и универсальным путем обратиться к онлайн-симулятору **Venus** (https://venus.kvakil.me/). Работа с ним не потребует никаких дополнительных знаний и предварительной подготовки.

Воспользуемся тем, что Venus поддерживает псевдооперацию **li**. Наберём на странице редактора программы (*Editor*) любой из примеров 2…7 в виде строки

li x6, const

(где const – это требуемое итоговое значение десятичной константы; напомним, что перед шестнадцатеричным значением необходимо добавлять символы *0x*). Изучим показанную на странице выполнения (*Simulator*) результирующую программу и убедимся в том, что Venus работает в **полном соответствии** с построенной выше таблицей.

Результаты пошагового выполнения (режим *Step*) для примера 5 приведены на рис. 3. Итоговую константу можно посмотреть в правой части окна в регистре x6.

Константы 64 бита в RISC-V

Стандарт RISC-V предусматривает не только 32-битную, но и 64-битную и даже 128-битную архитектуры. Расширение, как подчёркивается в стандарте, делается в основном ради увеличения адресного пространства ОЗУ, что позволит наращивать его объём. Касательно размера данных 2^{32} = = 4 294 967 296 представляет собой вполне достаточную для практики величину (вспомните также первый эпиграф к статье). Если в отдельных задачах потребуется бóльшая разрядность, то программисты умеют с этим справляться. Например, на основе 32-битных данных реализована неограниченная целочисленная арифметика в языке Python [22].

Кратко рассмотрим формирование 64-битных констант в архитектуре RV64I. Она сохраняет все инструкции набора RV32I и добавляет к нему 15 дополнительных инструкций. Подчеркнём, что инструкции в RV64I по-прежнему **имеют размер 32 бита**.

Разрядность регистров процессора увеличивается до 64 бит, а значит, нужно иметь механизм загрузки констант такого размера. Как уже упоминалось ранее, последовательность инструкций при формировании констант «пунктирно намечена» в Атласе RISC-V [7]. Для 64-битной константы там рекомендована серия из 8 инструкций. Первые две повторяют задание 32-битной константы (**LUI** и **ADDI**), что подробно обсуждалось выше. Далее предлагается трижды повторить пару инструкций **SLLI** (сдвиг влево) и **ADDI**. Такой алгоритм позволяет постепенно (за три шага) заполнить оставшиеся 32 бита. Приятным обстоятельством служит тот факт, что прибавлять за один раз достаточно 10–11 бит, так что знаковый разряд можно не трогать. Следовательно, формирование 64-битной константы не приводит к появлению новых вариантов, связанных с расширением знака. В то же время для некоторых значений констант, особенно при наличии в них большого числа нулей, количество инструкций можно сократить. Кроме того, в зависимости от конструкции машины, вместо выполнения 8 (!) предложенных инструкций может оказаться быстрее прочитать одной командой готовую константу из ОЗУ.

Заключение

На основе проведённого детального рассмотрения вариантов формирования константы в регистрах процессоров стандарта RISC-V можно сформулировать следующие выводы.

Любая константа заносится в 32-битный регистр по частям; старшая **Ch** содержит 20 бит, а младшая **Cl** – 12.

Занесение младшей части константы **Cl** в регистр выполняется с помощью инструкции **ADDI**. Операнд для неё всегда получается «механическим» выделением младших 12 бит (3 последние шестнадцатеричные цифры в константе).

Для занесения старшей части константы **Ch** в регистр используется инструкция **LUI**, В качестве начального значения её операнда берутся старшие 20 бит (5 первых шестнадцатеричных цифр из 8 в полной записи 32-битного числа). Конечное значение зависит от состояния старшего (11-го) бита в **Cl**. Если в этом бите 1, то **Ch** приходится корректировать, прибавляя к нему 1. В противоположном случае коррекция не требуется.

Причиной появления указанной неоднозначности служит обязательность расширения знака непосредственного операнда во всех операциях, в том числе в **ADDI**. Из-за этого старшие биты могут стать ненулевыми и изменить итоговую сумму. Здесь трудности программирования возникают вследствие упрощения аппаратной конструкции.

В общем случае (вариант v3 в таблице) константа заносится в регистр двумя инструкциями: сначала **LUI**, а затем **ADDI**. Когда значение константы по модулю мало (варианты v1 или v2), то **Ch** = 0 и команда **LUI** не нужна (операнд rs1 в **ADDI** при этом меняется на x0). Это очень важный и часто встречающийся на практике случай. Есть ещё довольно редкий вариант v4, в котором **Cl** = 0; тогда исключается команда **ADDI**.

Подчеркнём, что 11-й бит в младшей части константы выделен исключительно форматом инструкций. Все разговоры о знаке **Cl** носят формальный характер. Более того, бит этот не следует путать с «настоящим» знаком итогового значения константы в 31-м бите.

Существует псевдоинструкция **li**, которая «скрывает» все перечисленные выше варианты от программиста. Она позволяет написать любую константу и быть уверенным, что компилятор подберёт инструкции, обеспечивающие правильный результат. По мнению автора, понимать существующие в RISC-V особенности формирования констант всё равно требуется, хотя бы для того, чтобы не ошибиться там, где нет псевдоинструкций.

Литература

1. *Касперски К.* ПК: решение проблем. СПб.: БХВ-Петербург, 2003. 560 с.
2. *Столлингс В.* Структурная организация и архитектура компьютерных систем. М.: Издательский дом «Вильямс», 2002. 896 с.
3. *Waterman A., Lee Y., Patterson D.A., Asanović K.* The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Technical Report No. UCB/EECS-2016-118. May 31, 2016. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf.
4. The RISC-V Instruction Set Manual. Volume I: Unprivileged Architecture. Version 20250508. URL: https://docs.riscv.org/

reference/isa/_attachments/riscv-unprivileged.pdf.
5. A chip design that changes everything: 10 Breakthrough Technologies 2023. URL: https://www.technologyreview.com/2023/01/09/1064876/riscv-computer-chips-10-breakthough-technologies-2023/.
6. Регистры и модель памяти. Виды адресации. URL: https://uneex.ru/LecturesCMC/ArchitectureAssembler2022/02_MemoryRegisters.
7. *Patterson D.A., Waterman A.S.* The RISC-V Reader: An Open Architecture Atlas. Strawberry Canyon, 2017. 200 p. URL: https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/books/rvbook.pdf.
8. *Waterman A.S.* Design of the RISC-V Instruction Set Architecture. PhD dissertation, Dept. Elect. Eng. and Comp. Sci. Univ. of California, Berkeley, CA, USA, 2016. URL: https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf.
9. *Patterson D.A., Hennessy J.L.* The Hardware/Software Interface: RISC-V Edition. Cambridge, MA: Morgan Kaufmann Publishers, 2018. URL: https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/books/HandP_RISCV.pdf.
10. *Харрис Д.М., Харрис С.Л.* Цифровая схемотехника и архитектура компьютера: RISC-V. М.: ДМК Пресс, 2021. 810 с.
11. *Фролов В.А., Галактионов В.А., Санжаров В.В.* Исследование технологии RISC-V // Труды ИСП РАН. 2020. Том 32, вып. 2. С. 81–98.
12. Создание процессора со свободной архитектурой RISC-V. URL: https://v2020e.ru/blog/sozdanie-protsessora-so-svobodnoj-arkhitekturoj-risc-v-chast-1.
13. *Крейгон Х.* Архитектура компьютеров и ее реализация. М.: Мир, 2004. 416 с.
14. Intel 64 and IA-32 Architectures Software Developer's Manual. 2023. 5066 p.
15. *Лин В.* PDP-11 и VAX-11. Архитектура ЭВМ и программирование на языке ассемблера. М.: Радио и связь, 1989. 320 с.
16. *Харрис Д.М., Харрис С.Л.* Цифровая схемотехника и архитектура компьютера. М.: ДМК Пресс, 2017. 792 с. (Бесплатный официальный перевод книги доступен по ссылке https://is.ifmo.ru/books/2016/digital-design-and-computer-architecture-russian-translation_July16_2016.pdf).
17. *Таненбаум Э.* Архитектура компьютера. СПб.: Питер, 2003. 704 с.
18. *Кнут Д.Э.* Искусство программирования. Том 1, вып. 1. MMIX – RISC-компьютер нового тысячелетия. М.: ООО «И.Д. Вильямс», 2007. 160 с.
19. *Ледин Дж.* Современная архитектура и устройство компьютеров. СПб.: БХВ-Петербург, 2024. 656 с.
20. *Смит Б.Э., Джонсон М.Т.* Архитектура и программирование микропроцессора INTEL 80386. М.: Конкорд, 1992. 334 с.
21. *Смит С.* Программирование на языке ассемблера RISC-V. М.: ДМК Пресс, 2025. 276 с.
22. *Golubin A.* Python Internals: Arbitrary-precision Integer Implementation. URL: https://rushter.com/blog/python-integer-implementation/.