ForwardCom: открытый стандарт набора команд для высокопроизводительных микропроцессоров

Агнер Фог

17 августа 2016 г.

Оглавление

Глава 1. Введение

ForwardCom означает Forward Compatible Computer system.

Данный документ предлагает новую открытую архитектуру набора команд, спроектированную для оптимальной производительности, гибкости и масштабируемости. Проект ForwardCom включает в себя как новую архитектуру набора команд, так и соответствующую экосистему программных стандартов — прикладной двоичный интерфейс (ABI) , управление памятью, средства разработки, форматы библиотек и системные функции. Этот проект иллюстрирует улучшения, которых можно добиться посредством полного вертикального перепроектирования аппаратного и программного обеспечения, основываясь на открытом совместном процессе.

Настоящее руководство и весь связанный с ним код находятся по адресу https://github.com/ForwardCom.

1.1. Основные моменты

- Набор команд ForwardCom является компромиссом между принципами RISC и CISC, объединяя быстрый и оптимизированный дизайн декодирования и конвейера RISC–систем с компактностью и бóльшим числом работы, в пересчёте на одну команду, CISC–систем.
- Дизайн ForwardCom — масштабируется, для поддержки как малых встроенных систем, так и огромных суперкомпьютеров и векторных процессоров, без потери двоичной совместимости.
- Для обработки больших наборов данных предоставляются векторные регистры переменной длины.
- Циклы по массивам реализованы новым гибким способом, автоматически использующим максимальную длину вектора, поддерживаемую микропроцессором, во всех итерациях цикла, кроме последней. Последняя итерация автоматически использует такую длину вектора, в которую целиком помещается оставшееся число элементов. Для обработки с оставшихся данных и специальных случаев не нужно никакого дополнительного кода. Нет нужды компилировать код отдельно для разных микропроцессоров с разными длинами векторов.
- Когда становится доступным новый микропроцессор, с большей длины векторными регистрами, не нужно перекомпиляции или обновления программного обеспечения. Программное обеспечение гарантированно будет совместимым снизу вверх, и получит преимущество от более длинных векторов новых моделей микропроцессора.
- Сильные функции безопасности — фундаментальная часть аппаратного и программного дизайна.
- Управление памятью проще и эффективнее, чем в традиционных системах. Используются различные приёмы для устранения фрагментации памяти. Нет разбиения памяти на страницы и буфера ассоциативной трансляции (translation lookaside buffer, TLB). Вместо этого есть отображение памяти, с ограниченным количеством секций переменного размера.
- Нет динамически компонуемых библиотек (dynamic link libraries, DLLs) или разделяемых объектов. Вместо этого есть лишь один тип библиотек функций, который может использоваться и для статической, и для динамической компоновки. Загружается и компонуется лишь та часть библиотеки, которая действительно используется. Библиотечный код почти во всех случаях хранится непрерывно с кодом основной программы. В момент загрузки на основании аппаратной конфигурации, операционной системы, или окружения пользовательского интерфейса, можно автоматически выбирать разные версии функции или библиотеки.
- Предоставляется механизм для вычисления требуемого размера стека, что может в большинстве случаев предотвратить переполнение стека, не создавая при этом стек большего размера, чем необходимо.
- Предоставляется механизм оптимального распределения регистров по программным модулям и библиотекам функций. Это делает возможным сохранение большинства переменных в регистрах без сброса этих переменных в память. Векторные регистры могут быть сохранены эффективным способом, сохраняющим только ту часть регистра, которая действительно используется.

1.2. Основы

Архитектура набора команд — это стандартизированный набор машинных команд, который может выполнять компьютер. Имеется много используемых архитектур наборов команд.

Некоторые широко используемые наборы команд плохо спроектированы с самого начала. Эти системы много раз дополнялись расширениями и заплатками. Один из наихудших случаев — широко используемый набор команд x86 и его многочисленные расширения. Набор команд x86 — результат длительной истории недальновидных расширений и заплаток. Результат этой разработки — очень сложная архитектура, с тысячами разных кодов команд, очень сложная и дорогая для декодирования микропроцессором. Мы должны извлечь уроки из прошлых ошибок, чтобы сделать более хороший выбор при проектировании новой архитектуры набора команд и поддерживающего её программного обеспечения.

Дизайн должен быть основан на открытом процессе. Кырсте Асанович (Krste Asanović) и Дэвид Паттерсон (David Patterson) представили убедительные аргументы в пользу того, что следует предпочесть открытый набор команд. Открытость может быть решающей для успеха технического дизайна. Например, первоначальный IBM PC в начале 1980–х имел преимущество по сравнению с конкурирующими компьютерами, ибо открытая архитектура позволяла другим производителям аппаратного и программного обеспечения выпускать совместимое оборудование. IBM утратила своё доминирующее положение на рынке, когда она в 1987г. перешла к проприетарной Micro Channel Architecture. Также хорошо известен и не нуждается в дальнейшем обсуждении успех программного обеспечения с открытым исходным кодом. Единственно, что отсутствует в полной компьютерной экосистеме, основанной на открытых стандартах — открытая микропроцессорная архитектура, откроющая рынок также для меньших производителей микропроцессоров и для нишевых продуктов.

Настоящее руководство основано на обсуждении на различных интернет–форумах. Спецификации являются предварительными. Разработка нового стандарта извлекла бы пользу из длинной экспериментальной фазы, и было бы неразумно зафиксировать стандарт на данной, начальной, стадии.

1.3. Цели дизайна

Ранее опубликованные открытые наборы команд подходят для малых, дешёвых микропроцессоров, предназначенных для встроенных систем, однокристалльных систем, реализаций на основе ПЛИС для научных экспериментов, и т.п. Предлагаемая архитектура ForwardCom развивает эту идею, и нацелена на дизайн, который может превзойти существующие высокопроизводительные процессоры.

Набор команд ForwardCom основан на следующих приоритетах:

- Набор команд должен обладать простым и последовательным модульным дизайном.
- Набор команд должен представлять подходящий компромисс между принципами RISC, позволяющими быстрое декодирование, и принципами CISC, делающими возможным выполнение большей работы в расчёте на одну команду и более эффективное использование кэша кода.
- Дизайн должен быть расширяемым, чтобы новые команды и расширения можно было добавлять последовательным и предсказуемым образом.
- Дизайн должен быть масштабируемым, чтобы он подходил и для малых компьютеров с памятью на кристалле, и для огромных суперкомпьютеров с очень большими векторами.
- Дизайн должен быть конкурентоспособен по сравнению с имеющимися коммерческими, и сосредоточен на высокопроизводительных приложениях завтрашнего дня, а не низкопроизводительных приложениях дня вчерашнего.
- Поддержка векторов и других черт, которые, как давно доказано, существенны для высокой производительности, должна быть фундаментальной частью дизайна, а не неуклюжим отростком.
- Фундаментальной частью дизайна должна быть безопасность, а не специально добавляемые заплатки.
- Набор команд должен быть спроектирован в открытом процессе, с участием международного сообщества программистов и „железячников“, подобно работе по стандартизации в других технических областях.

- Весь вертикальный дизайн должен быть непроприетарным, и должен позволять всем создавать создавать совместимое программное обеспечение, аппаратное обеспечение, и оборудование, для тестирования, отладки, и эмуляции.
- Решения о командах и расширениях должны определяться не краткосрочными маркетинговыми соображениями монополистической микропроцессорной промышленности, а долгосрочными нуждами всего сообщества программистов, „железячников“, и организаций.
- Дизайн должен позволять построение совместимого снизу вверх программного обеспечения, которое будет работать без перекомпиляции на будущих процессорах с большего размера векторными регистрами.
- Дизайн должен позволять специфичные для приложения расширения.
- Базовые аспекты экосистемы — стандарт для ABI, ассемблер, компиляторы, библиотеки функций, системные функции, каркас пользовательского интерфейса, и т.п. — также, ради максимальной совместимости, должны быть стандартизированы.

Новому набору команд нелегко будет добиться успеха на коммерческом рынке, даже если этот набор лучше, нежели устаревшие системы, поскольку рынок предпочитает обратную совместимость с существующим программным и аппаратным обеспечением. Вряд ли набор команд ForwardCom станет успешным коммерческим продуктом, но обсуждение идеальных набора команд и программной экосистемы всё же могло бы быть полезным. Проект ForwardCom уже породил много важных идей, так что его стоит разрабатывать дальше, даже если мы не узнаем, когда он закончится. Настоящая работа может быть полезной, если по другим причинам должна возникнуть необходимость во введении нового набора команд. Работа будет особенно полезной для больших векторных процессоров; для приложений, в которых важна безопасность; для операционных систем реального времени; а равно и для проектов, в которых были бы препятствием патентные и лицензионные ограничения иных архитектур.

Предложения данного документа могут также быть полезны как источник вдохновения и для научных экспериментов. Многие идеи не зависят от деталей дизайна, и могут быть реализованы в существующих системах.

1.4. Сравнение с другими открытыми наборами команд

Было предложено несколько других открытых наборов команд, наиболее заметные — RISC-V и OpenRISC. Оба имеют чистый RISC–дизайн, с, преимущественно, фиксированным 32–разрядными командными словами. Эти наборы команд подходят для малых систем, где экономится место на кремнии, но они не спроектированы для высокопроизводительных суперскалярных процессоров, и не сосредоточены на деталях, критичных для достижения максимальной производительности в бóльших системах. Настоящее предложение рассматривается как следующий шаг к созданию открытого набора команд, который действительно эффективнее наилучших сегодняшних коммерческих наборов команд.

Типичный RISC–дизайн с размером команды, ограниченным 32 разрядами, оставляет ограниченное пространство для непосредственно заданных констант и адресов находящихся в памяти операндов. Программе среднего размера потребуются 32–разрядные относительные адреса находящихся в статической памяти операндов, во избежание переполнения во время процесса переразмещения адресов компоновщиком. Тридцатидвухразрядный относительный адрес требует нескольких команд в чистых RISC–дизайнах. Например, для прибавления находящегося в памяти операнда к регистру в чистом RISC–дизайне с только 32–разрядными командными словами вам потребуется пять команд: (1) загрузить младшую часть 32–разрядного адресного смещения; (2) прибавить старшую часть 32–разрядного адресного смещения; (3) прибавить точку отсчёта или указатель команд к этому значению; (4) прочесть операнд, находящийся по вычисленному адресу памяти; (5) выполнить желаемое сложение. Дизайн ForwardCom выполняет все указанные действия одной командой с удвоенным размером слова. Преимущество в скорости очевидно. Вычисление адреса, загрузка, и выполнение, чтобы достичь плавного прохода одной команды за такт в каждой ветви конвейера, осуществляются на своих стадиях конвейера,

Другое важное отличие состоят в том, что предыдущие RISC–дизайны имели ограниченную поддержку векторных операций. Дизайн ForwardCom вводит новую систему векторных регистров переменной длины, более эффективную и гибкую, чем наилучшие имеющиеся коммерческие дизайны. Эффективные векторные операции существенны для получения максимальной производительности, и были важным приоритетом в предлагаемом здесь дизайне ForwardCom.

1.5. Литература и ссылки

- Krste Asanović and David Patterson: "The Case for Open Instruction Sets. Open ISA Would Enable Free Competition in Processor Design". Microprocessor Report, August 18, 2014.
  www.linleygroup.com/mpr/article.php?id=11267
- RISC-V: The Free and Open RISC Instruction Set Architecture riscv.org
- OpenRISC: openrisc.io
- Open Cores: opencores.org
- Agner Fog: Proposal for an ideal extensible instruction set, 2015. A blog discussion thread that initiated the ForwardCom project.
  www.agner.org/optimize/blog/read.php?i=421
- Agner Fog: Stop the instruction set war, 2009. Blog post about the problems with the x86 instruction set.
  www.agner.org/optimize/blog/read.php?i=25
- Darek Mihocka: Standard Need To Be Forward Looking, 2007. Blog post criticizing the x86 instruction set standard.
  www.emulators.com/docs/nx02_standards.htm. See also the following pages.

Глава 2. Основы архитектуры

В данной главе приводится обзор наиболее важных черт архитектуры набора команд ForwardCom. Детали приводятся в последующих главах.

2.1. Полностью ортогональный набор команд

Набор команд ForwardCom — полностью ортогонален во всех отношениях. Одна и та же команда может использовать целочисленные операнды всех размеров и вещественные операнды всех точностей. Она может использовать регистровые операнды, операнды из памяти, или непосредственно заданные операнды. Она может также использовать много разных режимов адресации. Команды могут кодироваться в коротких формах с двумя операндами, в которых один и тот же регистр и как операнд–приёмник, и как операнд источник; или в более длинных формах с тремя операндами. Команда может работать со скалярами и векторами любого размера. Она может иметь предикаты или маски для условного выполнения для векторов на уровне элементов, и может иметь в качестве входных аргументов флаги — для определения режима округления, управления исключениями, и других деталей, там, где это нужно. Константные данные всех типов могут включаться в команды, и, для уменьшения размера команды, сжиматься различными способами.

Обоснование

Ортогональность реализуется посредством стандартизированного модульного дизайна, что упрощает реализацию аппаратуры, а также делает компиляцию более простой и гибкой, и облегчает компилятору преобразование линейного кода в векторный.

Поддержка непосредственно заданных констант всех типов является улучшением, по сравнению с имеющимися системами. Большинство имеющихся систем хранят вещественные константы в сегменте данных, и обращаются к ним через 32–разрядные адреса в коде команды. Это является пустой тратой места в кэше данных и вызывает много промахов кэша, так как данные разбросаны по разным секциям. Замена 32–разрядного адреса 32–разрядной непосредственно заданной константой делает код более эффективным и не увеличивает размер кода. Расширения, позволяющие 64–разрядные непосредственно заданные константы, возможны ценой наличия команд тройной длины. Однако эта возможность в базовом дизайне ForwardCom не требуется, поскольку, по объяснённым ниже причинам, приоритет состоял в минимизации количества разных размеров команд.

2.2. Размер команды

Набор команд ForwardCom для кода использует 32–разрядное слово. Команда может состоять из одного или двух 32–разрядных слов, с возможным расширением на три или более слова. Плотность кода можно увеличить, используя малые команды половинного размера, а равная 32 разрядам единица размера сохраняется соединением малых команд по две. Невозможно перейти во вторую малую команду из такой пары малых команд. В будущем можно добавить расширения с командами размером в три или более слова.

Обоснование

Архитектура CISC с многими разными размерами команд — неэффективна для суперскалярных процессоров, на которых мы хотим выполнять несколько команд за такт. Декодирующий препроцессор — часто узкое место. Вы должны определить длину первой команды, прежде чем узнаете начало новой команды. „Декодирование длин команд“ по своей сути — последовательный процесс, что делает сложным декодирование нескольких команд за такт. У некоторых микропроцессоров, чтобы обойти это узкое место, есть дополнительный „кэш микроопераций“, находящийся после декодера.

Желательно иметь как можно меньше разных длин команд, и чтобы облегчить определение длины каждой команды. Мы хотим малый размер команды для наиболее употребимых простых команд, но нам также

нужен больший размер команд, чтобы вместить вещи вроде большего набора регистров, команд с многими операндами, векторных операций с продвинутыми возможностями, 32–разрядных адресных смещений, и больших непосредственно заданных констант. Данное предложение — компромисс между компактностью кода, лёгким декодированием, и пространством для продвинутых возможностей.

2.3. Набор регистров

Имеется 32 регистра общего назначения (r0–r31) по 64 разряда в каждом, и 32 векторных регистра (v0–v31) переменной длины. Максимальная длина вектора — разная для разных реализаций аппаратуры. Регистры общего назначения можно использовать для целых чисел разрядности до 64 включительно и для указателей. Векторные регистры можно использовать для скаляров, либо для векторов, состоящих из целых и вещественных чисел.

Определяются следующие специальные регистры, видимые на уровне прикладной программы (все — 64–разрядные):

- указатель команд (Instruction pointer, IP);
- указатель секции данных (Data section pointer, DATAP);
- указатель блока окружения потока (Thread environment block pointer THREADP);
- указатель стека (Stack pointer, SP);
- численный управляющий регистр (Numeric control register, NUMCONTR).

Указатель стека идентичен r31. К другим специальным регистрам как к обычным регистрам обращаться нельзя.

Специального регистра флагов нет. Регистры r1–r7 и v1–v7 могут использоваться для масок, предикатов, и флагов для вещественных чисел, для управления такими атрибутами, как режим округления и управление исключениями.

Неиспользуемая часть регистра всегда устанавливается равной нулю. Это означает, что целочисленные операции с размером операнда, меньшим 64 разрядов, и векторные операции с размером вектора, меньшим максимального, всегда обнуляют неиспользуемые разряды регистра–приёмника.

Обоснование

Количество регистров — компромисс между плотностью кода и гибкостью. Цена сброса регистров в память обычно важна лишь в критичном наиболее глубоко вложенном цикле, которому вряд ли нужно более 32 регистров.

Мы можем избежать ложных зависимостей от предыдущего содержимого регистра, обнулив все неиспользуемые разряды регистра, вместо того, чтобы оставлять их неизменными. Аппаратура может сэкономить энергию, отключив неиспользуемые части исполнительных модулей и шин данных.

Специальный регистр флагов нежелателен для кода, планирующего вычисления как посредник между последними, и для векторного кода.

Причина обработки вещественных скаляров в векторных регистрах, а не в отдельных регистрах, заключается в облегчении для компилятора преобразования скалярного кода, включающего вызовы функций, в векторный код. Вещественный код часто содержит вызовы функций из математической библиотеки. Если библиотечная функция имеет векторы переменной длины и на входе, и на выходе, то та же функция может использоваться и для скаляров, и для векторов, а компилятор может легко векторизовать код, содержащий такие вызовы библиотечных функций.

2.4. Поддержка векторов

Векторные регистры могут содержать целые числа размером в 8, 16, 32, 64, и, возможно, 128 разрядов, или вещественные числа одинарной, двойной, и, возможно, четырёхкратной точности. Все элементы вектора обязаны иметь один и тот же тип. Элементы вектора обрабатываются параллельно. Например, векторное сложение даёт сумму двух векторов за одну операцию.

Векторные регистры имеют переменную длину. Каждый векторный регистр имеет дополнительные разряды для хранения длины вектора. Максимальная длина вектора зависит от аппаратуры. Например, если аппаратура поддерживает максимальную длину вектора, равную 64 байтам, а конкретному приложению нужно лишь 16 байт, то длина вектора устанавливается равной 16.

Некоторым командам необходимо явно указывать длину вектора, например, для чтения вектора из памяти. Эти команды используют регистр общего назначения для указания длины вектора. Длина обычно указывается как количество байтов, а не количество элементов вектора.

Сведения о максимальной длине вектора предоставляет специальный регистр. Максимальная длина, поддерживаемая процессором, должна быть степенью двойки. Используемая длина не обязана быть степенью двойки. Если используемая длина больше максимальной, то используется максимальная длина.

Содержимое векторного регистра может произвольно интерпретироваться как имеющее любые поддерживаемые типы и размеры. Например, аппаратура не предотвращает применение целочисленных команд к векторам, содержащим вещественные данные. То, что код имеет смысл — на совести программиста.

2.5. Циклы по векторам

Чтобы сделать циклы по векторам более компактными, предоставляется специальный режим адресации. Он использует базовый указатель P и отрицательный индекс J, и вычисляет адрес находящегося в памяти операнда как $P - J$, где P и J — регистры общего назначения. Это делает возожным выполнение цикла по массиву так, как это показано следующим псевдокодом:

```
P = адрес массива
J = размер массива (в байтах)
L = максимальная длина вектора (зависит от процессора)
X = векторный регистр
P += J; // указывает на конец массива
while (J > 0) {
    X = некая_операция(X),[P-J],(длина вектора, J)
    J -= L;
}
```

Данный цикл работает следующим образом: P указывает на конец массива, J представляет собой количество оставшихся элементов массива, уменьшающееся до тех пор, пока цикл не завершится. Цикл читает по одному вектору, находящемуся по адресу $[P - J]$, за раз. На всех итерациях цикла, кроме последней, J больше максимальной длины вектора, L, что заставляет процессор использовать максимальную длину вектора. Если размер массива не кратен максимальной длине вектора, то на последней итерации цикл будет использовать вектор меньшей длины, чтобы поместить оставшееся количество элементов. Очевидно, что цикл может содержать любое количество операций чтения векторов, записи векторов, и векторных арифметических команд, используя тот же принцип.

Данный цикл будет работать на разных процессорах, с разны максимальными длинами векторов, *не зная в момент компиляции максимальную длину вектора*. Таким образом, одна и та же часть программного обеспечения будет работать на разных микропроцессорах, с разными длинами векторов, без необходимости компилировать отдельно для каждого микропроцессора. Ещё одно преимущество состоит в том, что по завершении цикла не нужно никакого дополнительного кода для обработки оставшихся элементов, в случае, если размер массива не кратен длине вектора.

Обоснование

Большинство существующих систем имеют фиксированные длины векторов. Если разные процессоры имеют разные длины векторов, то вы должны компилировать код отдельно для каждой длины вектора. Каждый раз, когда на рынке появляется новый процессор с большей длиной векторов, вы должны откомпилировать новую версию кода, для новой длины вектора, используя вновь определённые расширения набора команд. Для нового программного обеспечения обычно требуется несколько лет для разработки и выхода на основной рынок. Для производителей программного обеспечения дорого разрабатывать и сопровождать разные версии их кода для каждой изредка появляющейся длины вектора.

Ещё одна проблема существующих систем заключается в том, что невозможно сохранить векторный регистр способом, который гарантированно совместим с последующими процессорами, имеющими более длинные вектора. Этой проблемы в дизайне ForwardCom нет, ибо длина вектора сохраняется в векторном регистре. Предоставляются команды для сохранения и восстановления векторов переменной длины и для сохранения лишь той части векторного регистра, которая действительно используется.

Дизайн ForwardCom делает возможным получение преимуществ нового процессора с более длинными векторными регистрами сразу же, без перекомпиляции кода. Метод выполнения цикла, описанный выше, делает это легко и очень эффективно. Вам не нужны разные версии кода для разных процессоров.

Можно получить тот же самый эффект и без специального режима отрицательной адресации, если обратить знак J и разрешить отрицательное значение в регистре, указывающем длину вектора, в то же время используя абсолютную величину в качестве длины вектора. Это решение менее элегантно и более запутанно, но его можно включить в дизайн ForwardCom, разрешив отрицательные значения при указании длины вектора.

Разворачивание цикла, как правило, не нужно: накладные расходы на цикл уже уменьшены до одной команды (вычитания и перехода, если положительно), и суперскалярный процессор выполнит много итераций параллельно, если цепочки зависимости не слишком длинны. Разворачивание цикла с многими аккумуляторами может быть полезно для сокрытия проходящей через весь цикл зависимости. В этом случае вы либо вставляете в развёрнутом коде после каждой секции команды управления циклом, либо вычисляете количество итераций цикла перед его началом.

У дизайна ForwardCom нет практического ограничения на длину вектора, которую может поддерживать микропроцессор. Большай микропроцессор с очень длинными векторами может быть полезен для вычислений с высокой степенью параллелизма по данным. Обсуждались и другие способы достижения высокой производительности при параллельной обработке данных, такие, как вращающиеся стеки регистров и программная конвейеризация, но был сделан вывод, что длинные векторы — метод, который может быть реализован эффективнее всего и в микропроцессоре, и в компиляторе.

2.6. Максимальная длина вектора

Максимальная длина векторных регистров будет разной у разных процессоров. Максимальная длина должна быть степенью двойки, и может быть столь большой, насколько желаемо, и должна быть не меньше 16 байт. Каждая команда может использовать меньшую длину, которой не нужно быть степенью двойки.

Максимальная длина может быть разной для элементов разных размеров. Например, максимальная длина для 32–разрядных целых чисел может быть равна 32 байтам, чтобы содержать восемь целых чисел, тогда как максимальная длина для 8–разрядных целых чисел могла бы быть равной 16 байтам, чтобы содержать 16 меньших чисел. Однако для разных типов с одним и тем же размером элемента максимальная длина должна быть одной и той же. Например, максимальная длина для вещественных чисел двойной точности должна быть такой же, что и для 64–разрядных целых чисел, поскольку циклы наверняка содержат оба типа, когда целочисленные вектора используются как маски для вещественных векторов. Максимальная длина для 32–разрядных элементов не может быть меньше, чем для элементов другого размера или типа операнда. Данное правило гарантирует возможность сохранения полного вектора при использовании 32–разрядного типа операнда.

Максимальная длина вектора, как правило, должна быть одна и та же для всех команд с одним и тем же типом данных. Однако могут быть исключения для команд, которые особенно дорого реализовать.

Несколько специальных регистров дают сведения о максимальной длине вектора, поддерживаемой аппаратурой, для каждого размера элемента вектора. Прикладная программа или операционная система может уменьшить максимальный размер вектора, что может быть полезным, если меньший размер вектора более подходит для конкретной цели.

Также можно уменьшить размер вектора для целей эмуляции. Виртуальные векторные регистры, которые больше, нежели поддерживает аппаратура, могут эмулироваться с помощью ловушек (синхронных прерываний), чтобы проверить функциональность программы на процессорах с большей максимальной длиной вектора, чем доступно в настоящий момент.

Когда команда указывает более длинный вектор, чем максимально возможно, то, если не активирована эмуляция больших векторов, используется максимальная длина. Это необходимо для эффективной реализации циклов по векторам, как описано выше, на с. 9.

2.7. Маски команд

Большинство команд может иметь регистр маски, который может использоваться для условного выполнения и для указания различных опций. Команды, работающие с регистрами общего назначения, в качестве регистра маски или предиката используют один из регистровs r1–r7. Бит 0 регистра маски указывает, исполняется ли команда, или нет; бит 1 — должен ли результат быть нулём или остаться неизменным, в случае, когда операция не выполняется.

Данный механизм может быть векторизован. Команды, работающие с векторными регистрами, в качестве регистров маски используют векторные регистры v1–v7. Вычисление над каждым элементом вектора обусловлено соответствующим элементом регистра маски.

Дополнительные разряды регистра маски используются для различных опций, перекрывая значения в численном управляющем регистре.

2.8. Режимы адресации

Вся адресация памяти — относительно некоторого базового указателя. Находящийся в памяти операнд может адресоваться с помощью одной из двух общих форм:

Address = BP + IX*SF
Address = BP + OS

Здесь BP — 64–разрядный базовый указатель, IX — 64–разрядный индексный регистр, SF — масштабирующий множитель, а OS — непосредственно заданное смещение. Базовый указатель присутствует всегда, остальные элементы — необязательны.

Базовый указатель, BP, может быть регистром общего назначения, указателем секции данных (DATAP), указателем команд (IP), или указателем стека (SP).

Индексный регистр, IX, может быть одним из регистров r0–r30. Значение, равное 31, означает отсутствие индексного регистра.

К индексному регистру можно применить лимит, в виде 16–разрядного беззнакового целого числа. Если индексный регистр больше (в смысле беззнакового сравнения), чем лимит, то возбуждается синхронное прерывание (trap).

Масштабирующий множитель, SF, равен размеру (в байтах) для скалярных операндов и для размножений значений. Для векторных операндов масштабирующий множитель равен 1. Как объяснено на с. 9, также доступен специальный режим адресации с SF $= -1$.

Смещение, OS, представляет собой расширенное знаком восьми–, шестнадцати–, или тридцатидвухразрядное число. Восьмиразрядные смещения умножаются на размер операнда. У шестнадцати– и тридцатидвухразрядных смещений множителя нет.

Поддержка режимов адресации и с индексом, и со смещением, — необязательна.

Переходы и вызовы указывают целевой адрес относительно указателя команд. Относительный адрес указывается со знаковым смещением размером в 8, 16, 24, или 32 разряда, умноженным на размер слова кода, равный 4 байтам. При 32–разрядном смещении охватывается адресный диапазон ± 8Гб.

Обоснование

Используется 64–разрядное адресное пространство. Относительная адресация используется для того, чтобы в коде команды избежать 64–разрядных адресов. В том редком случае, когда 64–разрядный абсолютный адрес необходим, он должен быть загружен в регистр, который затем используется как указатель.

Адресация с индексом, масштабируемым размером операнда, полезна для массивов. К индексу может быть применён лимит, так что границы массивов можно проверить без каких–либо дополнительных команд.

Адресация с отрицательным индексом полезна для эффективной реализации циклов по векторам, описанной на с. 9.

Указанные здесь режимы адресации охватят все распространённые применения, включая массивы, векторы, структуры, классы, и кадры стека.

Поддержка режима адресации с базовым указателем, индексом, и непосредственно заданным смещением, — необязательна, ибо это потребовало бы двух сумматоров на стадии конвейера, предназначенной для вычисления адреса, что могло бы ограничить максимальную тактовую частоту.

Глава 3. Форматы команд

3.1. Форматы и шаблоны

Все команды используют один из общих шаблонов форматов, показанных ниже (самые старшие разряды — слева). Базовая компоновка 32–разрядного слова кода показана в шаблоне A. Шаблоны B, C и D получаются из шаблона A заменой 8, 16, или 24 разрядов, соответственно, непосредственно заданными константами. Команды двойного и тройного размеров можно построить, добавив к одному из этих шаблонов одного или двух 32–разрядных слов. Например, шаблон A с дополнительным 32–разрядным словом, содержащим данные, называется A2. Шаблон E2 представляет собой расширение шабона A, в котором второе слово кода содержит дополнительное регистровое поле, дополнительные разряды кода операции, разряды опций, и данные.

Некоторые малые часто используемые команды можно закодировать в малом (tiny) формате, использующем половину слова кода. Две таких малых команды можно, используя шаблон T, упаковать в одно слово кода. Неспаренная малая команда, чтобы заполнить полное слово кода, должна комбинироваться с малого размера командой NOP.

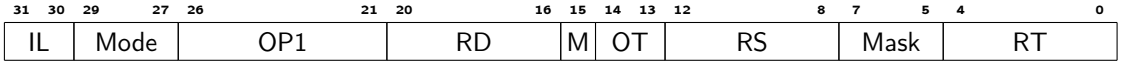

Рис. 3.1.1. Шаблон A. Имеется три регистровых операнда и регистр маски.

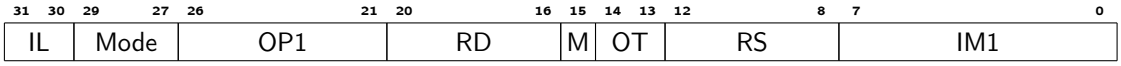

Рис. 3.1.2. Шаблон B. Имеется два регистровых операнда и 8–разрядная непосредственно заданная константа.

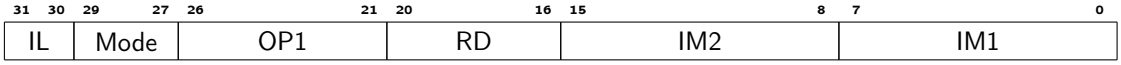

Рис. 3.1.3. Шаблон C. Имеется один регистровый операнд и две 8–разрядных непосредственно заданных константы.

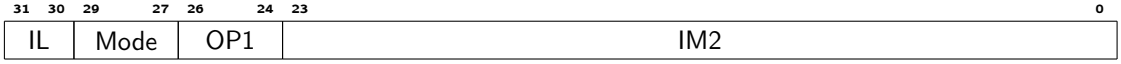

Рис. 3.1.4. Шаблон D. Нет регистрового операнда, но есть непосредственно заданная 24–разрядная константа.

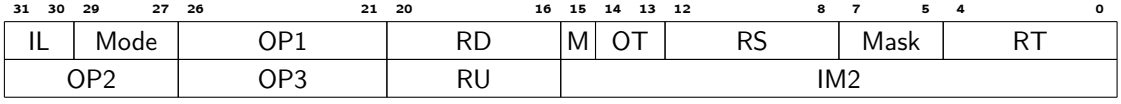

Рис. 3.1.5. Шаблон E2. Имеется 4 регистровых операнда, маска, 16–разрядная непосредственно заданная константа, и дополнительные разряды для кода операции или опций.

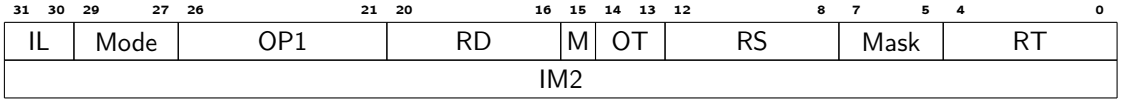

Рис. 3.1.6. Шаблон A2. Два слова. Как A, но с дополнительной 32–разрядной непосредственно заданной константой.

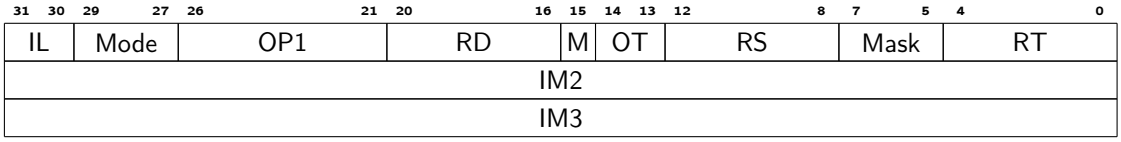

Рис. 3.1.7. Шаблон A3. Три слова. Как A, но с двумя дополнительными 32-разрядными непосредственно заданными константами.

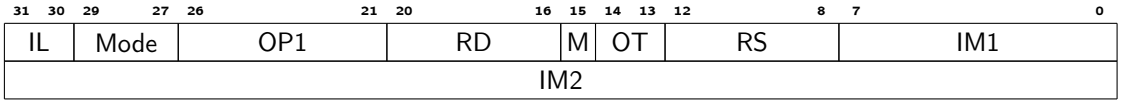

Рис. 3.1.8. Шаблон B2. Как B, но с дополнительной 32–разрядной непосредственно заданной константой.

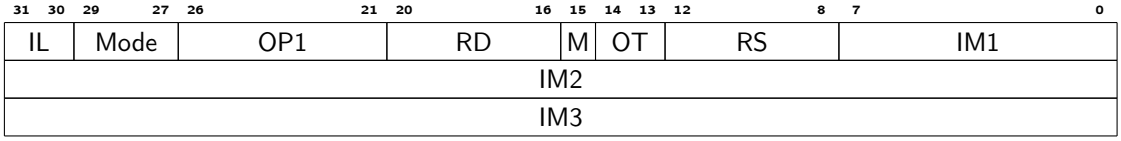

Рис. 3.1.9. Шаблон B3. Как B, но с двумя дополнительными 32-разрядными непосредственно заданными константами.

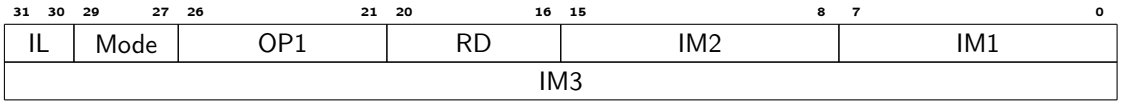

Рис. 3.1.10. Шаблон C2. Как C, но с дополнительной 32–разрядной непосредственно заданной константой.

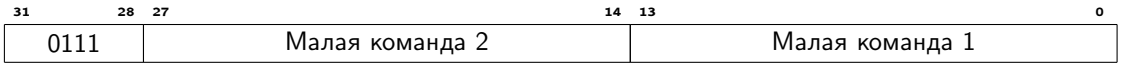

Рис. 3.1.11. Шаблон T. Одно слово, содержащее две малых команды.

Рис. 3.1.12. Формат каждой малой команды.

Смысл каждого поля описан в следующей таблице.

Таблица 3.1.1. Поля в шаблоне команды

Имя поля	Смысл	Значения
IL	Длина команды	0 или 1: 1 слово = 32 разряда 2: 2 слова = 64 разряда 3: 3 или более слов
Mode	Формат	Определяет формат шаблона и использование каждого поля. Когда необходимо, расширяется разрядами M. Детали см. ниже.

OP1	Код операции	Определяет операцию, например, сложение или пересылку.
OT	Тип и размер (OS) операнда	0: 8–разрядное целое, OS = 1 байт 1: 16–разрядное целое, OS = 2 байта 2: 32–разрядное целое, OS = 4 байта 3: 64–разрядное целое, OS = 8 байт 4: 128–разрядное целое, OS = 16 байт (необязательно) 5: вещественное одинарной точности, OS = 4 байта 6: вещественное двойной точности, OS = 8 байт 7: вещественное четырёхкратной точности, OS = 16 байт (необязательно) Когда необходимо, поле OT расширяется разрядами M.
RD	Регистр–приёмник	r0–r31 или v0–v31. Также используется для первого операнда–источника, если формат команды не указывает достаточное количество операндов.
RS	Регистр–источник	r0–r31 или v0–v31. Регистр–источник, указатель, индекс, или регистр длины вектора.
RT	Регистр–источник	r0–r31 или v0–v31. Регистр–источник или указатель.
RU	Регистр–источник	r0–r31 или v0–v31. Регистр–источник.
Mask	Регистр маски	0 означает отсутствие маски. 1–7 означает, что для маски и битов опций используется регистр общего назначения или векторный регистр.
M	Тип операнда или режим.	Расширяет поле Mode, когда разряды 1 и 2 этого поля оба равны нулю (регистры общего назначения). В противном случае расширяет поле OT (векторные регистры).
OP2	Код операции	Расширение кода операции.
IM1, IM2, IM3	Непосредственно заданные константы	8–, 16–, 32–, или 64–разрядный непосредственно заданный операнд, или адресное смещение, или разряды опций. Соседние поля IM могут быть слиты вместе.
OP3	Опции	Разряды опций, разряды режима, или непосредственно заданные константы.

Согласно приводимой ниже таблице, у команд есть несколько различных форматов, определяемых полем IL и разрядами режима.

Таблица 3.1.2. Список форматов команд

Имя формата	IL	Mode	Шаблон	Использование
0.0	0	0	A	Три операнда (RD, RS, RT), являющихся регистрами общего назначения.
0.1	0	1	B	Два регистра общего назначения (RD, RS) и 8–разрядный непосредственный операнд (IM1).
0.2	0	2	A	Три операнда (RD, RS, RT), являющихся векторными регистрами.
0.3	0	3	B	Два векторных регистра (RD, RS) и размножаемый 8–разрядный непосредственный операнд (IM1).
0.4	0	4	A	Один векторный регистр (RD), находящийся в памяти операнд с указателем (RT) и длина вектора, указанная в регистре общего назначения (RS).
0.5	0	5	A	Один векторный регистр (RD) и находящийся в памяти операнд с базовым указателем (RT). Отрицательный индекс и длина вектора указаны в RS. Используется для циклов по векторам, как объяснено на с. 9.
0.6	0	6	A	Один векторный регистр (RD) и находящийся в памяти скалярный операнд, имеющий базовый указатель (RT) и индекс (RS), умножаемый на размер операнда.

0.7	0	7	B	Один векторный регистр (RD) и находящийся в памяти скалярный операнд, имеющий базовый указатель (RS) и 8–разрядное смещение.
0.8	0	0 M=1	A	Один регистр общего назначения (RD) и находящийся в памяти операнд, имеющий базовый указатель (RT) и индекс (RS), умножаемый на размер операнда.
0.9	0	1 M=1	B	Один регистр общего назначения (RD) и находящийся в памяти операнд, имеющий базовый указатель (RT) и 8–разрядное смещение.
1.0	1	0	A	Одноформатные команды. Три регистра общего назначения в качестве операндов.
1.1	1	1	C	Одноформатные команды. Один регистр общего назначения и 16–разрядный непосредственный операнд.
1.2	1	2	A	Одноформатные команды. Три векторных регистра в качестве операндов.
1.3	1	3	B, C	Одноформатные команды. Два векторных регистра и размножаемый 8–разрядный непосредственный операнд, либо один векторный регистр и размножаемый 16–разрядный операнд.
1.4	1	4	B	Команды перехода с двумя регистровыми операндами и 8–разрядным смещением.
1.5	1	5	C, D	Команды перехода с одним регистровым операндом, 8–разрядной константой (IM2), и 8–разрядным смещением (IM1); либо нет регистрового операнда, но есть 24–разрядное смещение.
1.8	1	0 M=1	B	Одноформатные команды. Два регистра общего назначения и 8–разрядный непосредственный операнд.
T	1	6-7	T	Две малых команды.
2.0	2	0	A2	Два регистра общего назначения (RD, RS) и находящийся в памяти операнд с базовым указателем (RT) и 32–разрядным смещением (IM2).
2.1	2	1	A2	Три регистра общего назначения и 32–разрядный непосредственный операнд IM2.
2.2	2	2	A2	Один векторный регистр (RD) и находящийся в памяти операнд с базовым указателем (RT) и 32–разрядным смещением (IM2). Длина вектора указывается регистром общего назначения RS.
2.3	2	3	A2	Три вектоных регистра и размножаемый 32–разрядный непосредственный операнд IM2.
2.4.0	2	4	E2	OP3=00xxxx. Два векторных регистра (RD, RU) и находящийся в памяти скалярный операнд с базовым регистром (RT) и 16–разрядным смещением (IM2), расширяемым до длины (RS).
2.4.1	2	4	E2	OP3=01xxxx. Два векторных регистра (RD, RU) и находящийся в памяти скалярный операнд с базовым регистром (RT), 16–разрядным смещением (IM2), длиной (RS).
2.4.2	2	4	E2	OP3=10xxxx. Два векторных регистра (RD, RU) и находящийся в памяти операнд с базовым регистром (RT), отрицательным индексом (RS), and length (RS). Optional support for offset IM2 $\neq$ 0, otherwise IM2 = 0.
2.4.3	2	4	E2	OP3=11xxxx. Два векторных регистра (RD, RU) и находящийся в памяти скалярный операнд с базовым регистром (RT), масштабируемый индекс (RS), и лимит RS $\leq$ IM2 (беззнаково).

2.5	2	5	E2	Три вектоных регистра (RD, RS, RT) и размножаемое 16–разрядное непосредственно заданное целое число IM2. IM2 сдвигается влево на значение, указанное 6–разрядной беззнаковой величиной OP3, если OP3 не используется для других целей. RU обычно не используется.
2.6	2	6	A2	Одноформатные команды. Три регистра общего назначения и 32–разрядный непосредственный операнд.
2.7	2	7	A2, B2, C2	Команды перехода (OP1 $<$ 16). Одноформатные команды. Три векторных регистра и 32–разрядный непосредственный операнд.
2.8.0	2	0 M=1	E2	OP3=00xxxx. Три регистра общего назначения (RD, RS, RU) и находящийся в памяти операнд с базовым регистром (RT) и 16–разрядным смещением (IM2).
2.8.1	2	0 M=1	E2	OP3=01xxxx. Два регистра общего назначения (RD, RU) и находящийся в памяти операнд с базовым регистром (RT), индексом (RS), и без масштабирования. Необязательная поддержка для смещения IM2 $\neq$ 0, иначе IM2 = 0.
2.8.2	2	0 M=1	E2	OP3=10xxxx. Два регистра общего назначения (RD, RU) и находящийся в памяти операнд с базовым регистром (RT) и масштабируемым индексом (RS). Необязательная поддержка для смещения IM2 $\neq$ 0, иначе IM2 = 0.
2.8.3	2	0 M=1	E2	OP3=11xxxx. Два регистра общего назначения (RD, RU) и находящийся в памяти операнд с базовым регистром (RT), масштабируемым индексом (RS), и лимит RS $\leq$ IM2 (беззнаково).
2.9	2	1 M=1	E2	Три регистра общего назначения (RD, RS, RT) и 16–разрядное непосредственно заданное целое число IM2. IM2 сдвигается влево на значение, указанное 6–разрядной беззнаковой величиной OP3, если OP3 не используется для других целей. RU обычно не используется.
3.0	3	0	A3, B3	Команды перехода. Одноформатные команды с регистрами общего назначения в качестве операндов. Необязательно.
3.1	3	1	A3	Три регистра общего назначения и 64–разрядный непосредственно заданный операнд. Необязательно.
3.2	3	2	A3	Одноформатные векторные команды. Необязательно.
3.3	3	3	A3	Три векторных регистра и размножаемый 64–разрядный непосредственно заданный операнд. Необязательно.
3.8	3	0 M=1		В настоящее время не используется.
4.x	3	4-7		Зарезервировано для последующих команд, размером в 4 и более слова.

3.2. Кодирование операндов

Тип операнда

Тип и размер операндов определяется полем OT, как указано выше. Если поля OT нет, то, по умолчанию, тип операнда — 64–разрядное целое (OS = 8).

Тип регистра

Команды могут использовать либо регистры общего назначения, либо векторные регистры. Регистры общего назначения используются и для операндов–источников, и для операндов–приёмников, и для маск, если режим равен 0 или 1 (с M = 0 или 1). Векторные регистры используются и для операндов–источников,

и для операндов–приёмников, и для маск, если режим равен значению между 2 и 7. Равное нулю значение поля маски означает, что маски нет, и операция является безусловной.

Регистры–указатели

Команды с находящимся в памяти операндом всегда используют адреса относительно базового указателя. Базовый указатель может быть регистром общего назначения, указателем секции данных, или указателем команд. Базовый указатель определяется полем RS или RT. Это поле интерпретируется следующим образом.

Команды с форматами без смещения или с 8–разрядным смещением (0.4-0.9) могут использовать любой из регистров r0–r31 в качестве базового указателя. Регистр r31 является указателем стека.

Команды с форматами, имеющими 16–разрядное или 32–разрядное смещение (2.0, 2.2, 2.4, 2.8), могут использовать те же регистры, кроме r29, заменяемого указателем секции данных (DATAP), и r30, заменяемого указателем команд (IP). Это применимо также и к форматам с неиспользуемым 16–разрядным смещением (форматы 2.4.2 и 2.4.3).

Малые команды, имеющие находящийся в памяти операнд, в качестве указателя в 4–разрядном RS поле могут использовать r0–r14 или указатель стека (r31) — равное 15 значение поля RS обозначает указатель стека.

Индексные регистры

Команды с форматами, имеющимм индекс, в качестве индекса могут использовать r0–r30. Значение в поле индекса (RS), равное 31, означает отсутствие индекса. Знаковый индекс умножается на размер операнда (OS) для форматов 0.6, 0.8, 2.4.3, 2.8.2, 2.8.3; на 1 для формата 2.8.1; или -1 для форматов 0.5 и 2.4.2. Результат складывается со значением базового указателя.

Смещения

Смещения могут быть восьми–, шестнадцати–, или тридцатидвухразрядными. Значение смещения расширяется знаком до 64 разрядов. Восьмиразрядное смещение умножается на размер операнда (OS), определяемый полем OT. Шестнадцатиразрядное или тридцатидвухразрядное смещение не масштабируется. Результат складывается со значением базового указателя.

Поддержка режимов адресации и с индексом, и со смещением (форматы 2.4.2, 2.8.1, 2.8.3) — необязательна. Если этот тип адресации, требующий двух сложений, не поддерживается, то смещение в IM2 должно быть нулём.

Лимит для индекса

В форматах 2.4.3 и 2.8.3 имеется 16–разрядный лимит для индексного регистра, что полезно для проверки границ массивов. Если значение индексного регистра, рассматриваемое как беззнаковое целое число, больше беззнакового лимита, то порождается ловушка (trap, синхронное прерывание).

Длина вектора

Длина находящегося в памяти вектора указывается (для форматов 0.4, 0.5, 2.2, 2.4) в поле RS, регистрами r0–r30. Значение поля RS, равное 31, используется для обозначения скаляра той же длины, что и размер операнда (OS).

Значение длины находящегося в регистре вектора задаёт длину векторного операнда, находящегося в памяти, в байтах, а не в количестве элементов. Если это значение больше максимально возможной длины вектора, то используется максимальная длина вектора. Длина вектора может быть равна нулю. Поведение для отрицательных значений длины зависит от реализации: либо данное значение рассматривается как беззнаковое, либо используется абсолютная величина.

Длина вектора должна быть кратна размеру операнда (OS), указываемому полем OT. Если длина вектора не кратна размеру операнда, то поведение для частично определённого элемента вектора зависит от реализации.

Длина вектора для находящихся в векторных регистрах операндов–источников сохраняется в регистре.

Комбинирование векторов с различными длинами

Длина вектора, находящегося в приёмнике, будет такой же, что длина вектора, находящегося в первом из операндов–источников, даже если первый операнд–источник использует поле RD.

Как следствие, при комбинировании векторов с разными длинами длина результата определяется порядком операндов.

Если операнды–источники имеют разные длины, то длины будут настроены так, как показано ниже. Если векторный операнд–источник слишком длинен, то лишние элементы игнорируются. Если векторный операнд–источник слишком короток, то отсутствующие элементы будут равны нулю.

Находящийся в памяти скалярный операнд (форматы 0.6 и 0.7) не размножается, а рассматривается как короткий вектор, и дополняется нулями до длины вектора–приёмника.

Размножаемый операнд, находящийся в памяти (формат 2.4.1) будет использовать длину вектора, заданную регистром, указанным в поле RS.

Размножаемый непосредственно заданный операнд будет использовать ту же длину вектора, что и операнд–приёмник.

Непосредственно заданные константы

Непосредственно заданные константы могут иметь разрядность, равную 4, 8, 16, 32, и, необязательно, 64. Непосредственно заданные поля, как правило, выровнены на естественные адреса, и интерпретируются следующим образом.

Если поле OT указывает целочисленный тип, то поле (непосредственно заданное) рассматривается как целое число. Если поле меньше размера операнда, то оно расширяется знаком до подходящего размера. Если поле больше размера операнда, то излишние разряды игнорируются. Усечение слишком большого непосредственно заданного операнда не взведёт никакого условия переполнения.

Если поле OT указывает вещественный тип, то поле рассматривается так. Непосредственно заданные поля разрядности, меньшей 32, интерпретируются как знаковые целые числа, и преобразуются в вещественные числа желаемой точности. Тридцатидвухразрядное поле рассматривается как вещественное число одинарной точности, и, если необходимо, преобразуется к желаемой точности. Шестидесятичетырёхразрядное поле (если таковое поддерживается) интерпретируется как вещественное число двойной точности. Шестидесятичетырёхразрядное поле для типа операнда, являющегося числом одинарной точности, не допускается. Несколько необязательных команд формата 1.3C имеют в качестве операндов вещественные непосредственно заданные константы половинной точности, которые преобразуются в скаляр одинарной или двойной точности.

Шестнадцатиразрядные константы в форматах 2.5 и 2.9 могут сдвигаться влево на значение, указанное 6–разрядной беззнаковой величиной OP3, чтобы получить 64–разрядное знаковое значение. Всё, что окажется за пределами 64 разрядов, игнорируется. Сдвиг выполняется до какого–либо преобразования в вещественные числа. Никакого сдвига не выполняется, если OP3 используется в других целях.

Команду можно сделать компактной, если использовать наименьший размер поля непосредственно заданного операнда, в который помещается действительное значение константы.

Регистры масок

Трёхразрядное поле маски указывает регистр маски. Если приёмником является регистр общего назначения, то используются регистры r1–r7; а если приёмник — векторный регистр, то используются регистры v1–v7. Значение поля маски, равное нулю, означает отсутствие маски и безусловное выполнение, с использованием опций, указанных в численном управляющем регистре.

Если маска является векторным регистром, то она рассматривается как вектор, имеющий тот же размер элемента, что и указанный полем OT. Каждый элемент регистра– маски применяется к соответствующим компонентам результата.

Смысл регистров флагов описан в следующем разделе.

3.3. Кодирование масок

Регистр маски может быть регистром общего назначения r1–r7 или векторным регистром v1–v7. Равное нулю значение поля маски означает отсутствие маски.

Разряды регистра маски кодируются так.

Таблица 3.3.1. Разряды регистра маски и численного управляющего регистра

Номер разряда	Смысл
0	Предикат или маска. Операция выполняется только в том случае, если этот разряд равен единице. Если этот разряд равен нулю, то операция не выполняется, и подавляются все условия проверки арифметических ошибок.
1	Зануление. Этот разряд определяет результат, когда разряд №0 равен 0. Равенство нулю этого разряда зануляет результат, а равенство единице оставляет значение без изменения, т.е. результат — такой же, как и значение в первом операнде–источнике на входе. Разряд №1 не оказывает влияния, когда разряд №0 равен единице.
2	Обнаруживать беззнаковое целочисленное переполнение.
3	Обнаруживать знаковое целочисленное переполнение.
6	Распространять ошибочные разряды, обнаруженные разрядами №№2 и 3. Это экспериментальная возможность, см. с. 55.
7	Возбудить прерывание, если обнаружено переполнение, указанное разрядами №2 или 3.
18-19	Режим округления для вещественных чисел: 00 = к ближайшему или чётному 01 = в меньшую сторону 10 = в большую сторону 11 = к нулю
20	Поддерживать денормализованные числа. Денормализованные вещественные числа рассматриваются как нуль (это, как правило, быстрее), когда данный разряд равен 0.
22	Более хорошее распространение нечисел (NAN). Если равен нулю этот разряд, то строго придерживаться стандарта IEEE 754-2008 (или более позднего) для значений NAN. Равное единице значение разряда №22 улучшает распространение NAN и использует значения NAN для отслеживания ошибок вычислений с плавающей запятой. Детали описаны на с. 55.
26	Разрешить возбуждение прерывания при вещественном переполнении и при делении на нуль.
27	Разрешить возбуждение прерывания при выполнении недопустимой операции с плавающей запятой.
28	Разрешить возбуждение прерывания при вещественном антипереполнении и потере точности.
29	Разрешить возбуждение прерывания при NAN в качестве входных аргументов команд сравнения и команд преобразования вещественных чисел в целые.

Разряды 8–9, 16–17, 24–25, и т.д. в векторном регистре маски могут использоваться подобно разрядам 0–1 для 8–разрядных и 16–разрядных операндов. Все прочие разряды зарезервированы для использования в будущем.

Векторные команды трактуют регистр маски как вектор с тем же размером элемента (OS), что и у операндов. У каждого элемента вектора–маски имеются битовые коды, перечисленные выше. У разных элементов вектора могут быть разные разряды маски.

Когда поле маски равно нулю или отсутствует, в качестве маски используется численный управляющий регистр (NUMCONTR). Когда у команды нет регистра маски, регистр NUMCONTR размножается для всех элементов вектора, используя столько разрядов регистра NUMCONTR, сколько указано размером операнда. В этом случае ко всем элементам вектора применяется одна и та же маска. Разряд №0 регистра NUMCONTR обязан быть равным 1.

3.4. Формат команд перехода, вызова, и ветвления

Большинство ветвлений в обычном коде основаны на результате арифметической или логической команды (АЛУ). Дизайн ForwardCom комбинирует команду АЛУ и условный переход в одну команду. Например, цикл можно реализовать одной командой, которая уменьшает счётчик и выполняет переход, если счётчик не

достиг нуля, либо увеличивает счётчик с отрицательного значения, и выполняет переход, если это значение не достигло нуля.

Переходы, вызовы, ветвления, и многопутёвые ветвления используют следующие форматы.

Таблица 3.4.1. Список форматов для команд передачи управления

Формат	IL	Mode	OP1	Шаблон	Описание
1.4	1	4	OPJ	B	Короткая версия, с двумя регистровыми операндами (RD, RS) и 8–разрядным смещением (IM1).
1.5 C	1	5	OPJ	C	Короткая версия, с одним регистровым операндом (RD), и либо с 8–разрядной непосредственно заданной константой (IM2) и 8–разрядным смещением (IM1), либо с 16–разряднымым смещением (склеены IM2 и IM1).
1.5 D	1	5	0-7	D	Переход или вызов с 24–разрядным смещением.
2.7.0	2	7	0	B2	Версия двойного размера, с двумя регистровыми операндами и с 32–разрядным смещением (IM2). IM1 = OPJ.
2.7.1	2	7	1	B2	Версия двойного размера, с регистровым операндом–приёмником, регистровым операндом–источником, 16–разрядным смещением (младшая половина IM1) и 16–разрядным непосредственно заданным операндом (старшая половина IM2).
2.7.2	2	7	2	C2	Версия двойного размера, с одним регистровым операндом (RD), одной 8–разрядной непосредственно заданной константой (IM2) и 32–разрядным смещением (IM3).
2.7.3	2	7	3	C2	Версия двойного размера, с одним регистровым операндом (RD), 8–разрядным смещением (IM2) и 32–разрядной непосредственно заданной константой (IM3).
2.7.4	2	7	4	C2	Двойного размера системный вызов, без OPJ, с 16–разрядной константой (IM1,IM2), и с 32–разрядной константой (IM3).
3.0.0	3	0	0	C2	Нет операции (NOP).
3.0.1	3	0	1	B3	Версия тройного размера, с регистровым операндом–приёмником, регистровым операндом–источником, 32–разрядным непосредственно заданным операндом (IM2), и 32–разрядным смещением (IM3). Необязательно.

Команды перехода, вызова, и ветвления имеют знаковые смещения размером в 8, 16, 24, или 32 разряда, относительно указателя команд. Или, точнее, относительно конца команды. Данное смещение умножается на размер слова команды (равный 4 байтам), чтобы охватить диапазон в плюс–минус полкилобайта для коротких условных переходов с 8–разрядным смещением, в плюс–минус 32 мегабайта для 24–разрядных смещений, и плюс–минус 8 гигабайт для 32–разрядных смещений. Необязательный формат тройного размера включает безусловный переход и вызов с 64–разрядным абсолютным адресом.

Версия с шаблонами C и C2 не имеет поля OT. Когда поля OT нет, тип операнда — 64–разрядное целое. С вещественными типами использовать шаблоны C и C2 невозможно. Когда есть поле OT и M=1, команды будут использовать векторные регистры (только первый элемент). Иными словами, команды АЛУ–перехода будут использовать векторные регистры только когда указан вещественный тип (или, если таковой поддерживается, 128–разрядный целочисленный тип). Во всех иных случаях используются регистры общего назначения. Можно использовать поразрядные логические команды с векторными командами, указав вещественный тип.

Поле OPJ определяет операцию и условие перехода. Данное поле — 6–разрядное в версии одинарного размера, и 8–разрядное — в более длинных версиях. Два дополнительных разряда в более длинных версиях используются так: бит 6 зарезервирован для использования в будущем, и обязан быть равным нулю; а бит 7 может использоваться для указания поведения цикла, как подсказка для выбора оптимальной ветви

алгоритмом предсказания ветвлений.

Младшие 6 разрядов поля OPJ содержат следующие коды:

Таблица 3.4.2. Список команд передачи управления: переходы, вызовы, возвраты

OPJ	Бит 0 поля OPJ	Функция	Комментарий
0-7	часть смещения	Безусловный переход с 24–разрядным смещением.	Формат 1.5 D.
8-15	часть смещения	Безусловный вызов с 24–разрядным смещением.	Формат 1.5 D.
0-1	инвертирован	Знаково вычесть, и перейти, если отрицательно (sub_sign_jmpneg).	Форматы 1.4 и 2.7.0. Не для плавающей запятой.
2-3	инвертирован	Знаково вычесть, и перейти, если положительно (sub_sign_jmppos).	Форматы 1.4 и 2.7.0. Не для плавающей запятой.
4-5	инвертирован	Беззнаково вычесть, и перейти, если заём (sub_unsign_jmpborrow).	Форматы 1.4 и 2.7.0. Не для плавающей запятой.
6-7	инвертирован	Беззнаково вычесть, и перейти, если не нуль либо заём (sub_unsign_jmpnzc).	Форматы 1.4 и 2.7.0. Не для плавающей запятой.
8-9	инвертирован	Вычесть, и перейти, если не ноль (sub_jmpnzero).	Форматы 1.4 и 2.7.0. Не для плавающей запятой.
10-11	инвертирован	Знаково вычесть, и перейти, если переполнение (sub_sign_jmpovfl).	Форматы 1.4 и 2.7.0. Не для плавающей запятой.
12-15		Зарезервировано для последующего использования.	Форматы 1.4 и 2.7.0.
16-17	инвертирован	Знаково сложить, и перейти, если отрицательно (add_sign_jmpneg).	Не для плавающей запятой.
18-19	инвертирован	Знаково сложить, и перейти, если положительно (add_sign_jmppos).	Не для плавающей запятой.
20-21	инвертирован	Беззнаково сложить, и перейти, если перенос (add_unsign_jmpcarry).	Не для плавающей запятой.
20-21	инвертирован	Перейти, если один из операндов равен $\pm\infty$ или является NAN (cmp_float_jmpinfnan).	Для вещественных операндов.
22-23	инвертирован	Беззнаково сложить, и перейти, если не нуль, либо перенос (add_unsign_jmpnzc).	Не для плавающей запятой.
22-23	инвертирован	Перейти, если один из операндов — денормализован (cmp_float_jmpsubnorm).	Для вещественных операндов.
24-25	инвертирован	Сложить, и перейти, если не нуль (add_jmpnzero).	Не для плавающей запятой.
26-27	инвертирован	Знаково сложить, и перейти, если переполнение (add_sign_jmpovfl).	Не для плавающей запятой.
28-29	инвертирован	Сдвинуть влево на n разрядов, и перейти, если не нуль (shift_jmpnzero).	Беззнаково сдвинуть вправо, если n отрицательно.
30-31	инвертирован	Сдвинуть влево на n разрядов, и перейти, если перенос (shift_jmpcarry).	Беззнаково сдвинуть вправо, если n отрицательно.
32-33	инвертирован	Знаково сравнить, и перейти, если ниже (cmp_sign_jmpbelow).	

34-35	инвертирован	Знаково сравнить, и перейти, если выше (cmp_sign_jmpabove).	
36-37	инвертирован	Беззнаково сравнить, и перейти, если ниже (cmp_unsign_jmpbelow).	Целочисленные операнды.
36-37	инвертирован	Перейти, если один из операндов — NAN (cmp_float_jmpunordered).	Вещественные операнды.
38-39	инвертирован	Беззнаково сравнить, и перейти, если выше (cmp_unsign_jmpabove).	Целочисленные операнды.
38-39	инвертирован	Перейти, если один из операндов равен $\pm\infty$ (cmp_float_jmpinf).	Вещественные операнды.
40-41	инвертирован	Сравнить, и перейти, если не равно (cmp_jmpneq).	
42-43	инвертирован	Поразрядное И без записи результата, и перейти, если не ноль (test_jmpnzero).	
44-45	инвертирован	Поразрядное И, и перейти, если не ноль (and_jmpnzero).	
46-47	инвертирован	Поразрядное или, и перейти, если не ноль (or_jmpnzero).	
48-49	инвертирован	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ, и перейти, если не ноль (xor_jmpnzero).	
50-51	инвертирован	Проверить один разряд, и перейти, если не ноль (testbit_jmpnzero).	
52-53	инвертирован	Проверить один разряд векторного регистра, и перейти, если не ноль (testbit_jmpnzero).	
54-57		Зарезервировано для последующего использования.	
58-59	0 переход 1 вызов	Косвенно, с адресом указателя в регистре, указанном в RS, и смещением указателя в IM1 или IM2 (jump/call).	Форматы 1.4 и 2.7.0.
58-59	0 переход 1 вызов	Безусловный косвенный переход/вызов с 16–разрядным или 32–разрядным смещением, либо с 64–разрядным абсолютным адресом (jump/call).	Форматы 1.5 C, 2.7.2, и 3.0.1.
60-61	0 переход 1 вызов	Использовать таблицу адресов относительно регистра, указанного в RD. RT = базовый адрес таблицы, RS = индекс*OS (jump/call).	Формат 1.4, шаблон A.
60-61	0 переход 1 вызов	Безусловный переход или вызов по адресу в регистре, указанном в RS (jump/call).	Формат 1.5.
62	0	Возврат из функции (return).	Формат 1.4.
62	0	Возврат из системной функции (sys_return).	Формат 1.5.
63	1	Системный вызов. ID — в регистре, указанном в RT, блок разделяемой памяти — в RD, длина — в RS. Маски нет (sys_call).	Формат 1.4, шаблон A.
63	1	Системный вызов. ID — в константах, блок разделяемой памяти — в RD, длина — в RS. Маски нет (sys_call).	Форматы 2.7.1, 2.7.4 и 3.0.1.
63	1	Безусловное прерывание (trap). Номер прерывания — в IM1 (trap).	Формат 1.5.
63	1	Заполнитель для неиспользуемой памяти кода. Все поля равны 1 (filler).	Формат 1.5.
63	1	Прерывание, если (беззнаково) RD > IM3. IM2 = 38. Номер прерывания фиксирован (cmp_unsign_trapabove).	Формат 2.7.3.

Знаковые целочисленные сравнения корректируются при переполнении, а знаковые сложения и вычитания — нет. Например, если A — большое положительное целое число, а B — большое отрицательное целое

число, то sub_sign_jmpneg выполнит переход, если вычисление A-B из–за переполнения даст отрицательный результат, а cmp_sign_jmpbelow не выполнит переход, ибо A — больше B.

Комбинирование команд АЛУ и команд условного перехода можно закодировать в форматах 1.4, 1.5 C, 2.7.0, 2.7.1, 2.7.2, 2.7.3, и 3.0.1, за исключением вычитания, которое не может быть закодировано в формате 1.5 C. Вычитание с непосредственно заданной константой в качестве операнда можно заменить сложением с отрицательной константой. Место в коде, которое в формате 1.5 C использовалось бы вычитанием, вместо этого используется для кодирования команды прямого перехода и вызова с 24–разрядным смещением, используя формат 1.5 D, в котором младшие три разряда поля OP1 используются как часть 24–разрядного смещения.

Операции сложения и вычитания для вещественных операндов обычно не поддерживаются, поскольку бóльшие задержки этих вещественных операций усложнят дизайн конвейера. Вещественные сравнения поддерживаются, поскольку можно выполнить операцию вещественного сравнения за один такт, используя беззнаковое целочисленное сравнение совместно экспоненты и мантиссы, со специальной обработкой знакового разряда и значений NAN.

Команда проверки разряда (testbit_jmpnzero) проверяет в первом операнде разряд с номером n, где n — значение во втором операнде (RS или IM2). Она полезна для проверки битовых полей, знаковых разрядов, и результата команд сравнения. Независимо от типа операнда, второй операнд интерпретируется как беззнаковое целое число.

Команды сдвига влево сдвигают первый операнд на количество разрядов, указанное вторым операндом, влево, если тот положителен, и вправо, с расширением нулём, если второй операнд отрицателен. Переносом является последний выдвинутый разряд. Независимо от типа операнда, операнды трактуются как целые, но в случае, если указан вещественный тип операнда (M = 1), — используются векторные регистры.

Безусловные и косвенные переходы и вызовы используют указанные выше форматы, в которых неиспользуемые разряды должны быть нулями. Разряд №0 поля OPJ равен нулю для команд перехода, и единице для команд вызова.

Команды табличного косвенного перехода/вызова предназначены для облегчения реализации многопутёвых ветвлений (операторы switch/case), таблиц функций в интерпретаторах кода, и таблиц виртуальных функций в объектно–ориентированных языках с полиморфизмом. Таблица адресов перехода или вызова хранится в виде знакового смещения относительно произвольной точки отсчёта, которая может быть адресом таблицы, базовым адресом кода, или любой другой точкой отсчёта. Тип операнда указывает размер элемента таблицы. Обязательно должны поддерживаться 16–разрядные и 32–разрядные смещения, прочие размеры — необязательно. Использование относительных адресов делает таблицу компактнее, нежели при использовании 64–разрядных абсолютных адресов. Команда работает следующим образом: вычисляется адрес элемента таблицы, как сумма базового указателя (RT) и индекса (RS), умноженного на размер операнда; по данному адресу читается знаковое значение, которое умножается на 4; то, что получилось, расширяется знаком до 64 разрядов и складывается с указанной точкой отсчёта (RD); по вычисленному адресу совершается переход или вызов. Индекс массива (RS) умножается на размер операнда, в то время как элементы таблицы умножаются на размер слова команды (4). Поддержка маски — необязательна.

Таблицу, используемую командами табличного перехода/вызова, можно разместить в секции константных данных (CONST), что делает возможным использование в качестве точки отсчёта базового адреса таблицы, и улучшает безопасность, давая доступ к таблице только для чтения.

Когда используются соглашения вызова, сформулированные на с. 73, командам возврата смещение в стеке не нужно.

Системные вызовы для идентификации системных функций используют числа, ID, а не адреса. Этот ID является комбинацией ID модуля, идентифицирующего конкретный модуль системы или драйвер устройства, и ID функции, идентифицирующего конкретную функцию в этом модуле. Как ID модуля, так и ID функции — либо оба — 16–разрядные, либо оба — 32–разрядные, так что суммарный размер ID системного вызова может достигать 64 разрядов. Командае sys_call имеет следующие варианты:

Таблица 3.4.3. Варианты команды системного вызова

Формат	Тип операнда	ID функции	ID модуля
1.4	32 разряда	разряды 0-15 регистра RT	разряды 16-31 регистра RT
1.4	64 разряда	разряды 0-31 регистра RT	разряды 32-63 регистра RT
2.7.1	32 разряда	разряды 0-15 поля IM2	разряды 16-31 поля IM2
2.7.4	64 разряда	разряды 0-15 поля IM21	разряды 0-31 поля IM3
3.0.1	64 разряда	разряды 0-31 поля IM2	разряды 0-31 поля IM3

Команде sys_call можно указать блок памяти, разделяемый с системной функцией. Адрес блока памяти задаётся в регистре, указываемом полем RD, а длина — в регистре, указываемом полем RS. Этот блок памяти, к которому у вызывающего должны быть права доступа, разделяется с системной функцией. Системные функции получат те же права на этот блок, что и вызывающий поток, т.е. те же права на чтение и/или запись. Это полезно для быстрой передачи данных между вызывающим и системной функцией. У вызывающей и вызываемой функций нет никакого совместного доступа ни к какой другой области памяти. Если и поле RD, и поле RS равны нулю (т.е. указан регистр r0), то никакой блок памяти не разделяется. Команда sys_call формата 2.7.4 не может иметь никакого блока разделяемой памяти.

Параметры системных функций передаются в регистрах, следуя тем же соглашениям вызова, что обычные функции. Регистры, используемые для передачи параметров, обычно отличаются от регистров, указанных в полях RD, RS и RT. Параметры функций, не помещающиеся в регистры, должны находиться в блоке разделяемой памяти.

Ловушки работают подобно прерываниям. Безусловные ловушки имеют 8–разрядный номер прерывания в IM1, являющийся индексом в таблице векторов прерываний, первоначально находящейся по равному нулю абсолютному адресу. Команда безусловной ловушки для дополнительной информации может использовать IM2. Условная ловушка предназначена для проверки границ массивов. Номер прерывания фиксирован (решение о значении ещё не принято). Условная ловушка может (необязательно) поддерживать в IM2 иные коды условия, использующие те же самые коды, что и OPJ в табл. 3.4.2.

Команда ловушки с единицами во всех разрядах всех полей (код операции — 0x6FFFFFFF) может использоваться в качестве заполнителя неиспользуемых частей памяти кода.

3.5. Назначение кодов операций

Коды операций и форматы могут быть назначены новым командам в соответствии со следующими правилами.

- Многоформатные команды. Часто используемые команды, которым нужно поддерживать много различных типов операндов, режимов адресации, и форматов, используют большую часть (или все) из следующих форматов: 0.0-0.9, 2.0-2.5, 2.8-2.9, и (необязательно) 3.1 и 3.3 (если поддерживаются команды тройного размера). Во всех этих форматах используется одно и то же значение поля OP1. Поле OP2 обязано быть равно 0. Команды с немногими операндами–источниками идут первыми.

- Малые команды. В малой версии доступны лишь некоторые из наиболее употребимых команд, поскольку место есть только для 32 малых команд. Как показано в табл. 4.2.1 на с. 31, команды упорядочиваются по количеству и типам операндов.

- Команды передачи управления, т.е. переходы, ветвления, вызовы, и возвраты, могут быть закодированы как короткие команды с IL = 1, Mode = 4 или 5, и OP1 от 0 до 63, либо как команды двойного размера, с IL = 2, Mode = 7, OP1 от 0 до 15, и (необязательно) как команды тройного размера, с IL = 3, Mode = 0, OP1 от 0 до 15. См. с. 20.

- Короткие одноформатные команды с регистрами общего назначения. Используйте форматы 1.0, 1.1, и 1.8, с любым значением поля OP1.

- Короткие одноформатные команды с векторными регистрами. Используйте форматы 1.2 и 1.3, с любым значением поля OP1.

- Одноформатные команды двойного размера с регистрами общего назначения могут использовать форматы 2.8 и 2.9, с любым значением полей OP1 и OP2 ≥ 8 (для одного и того же OP1 предоставляйте сходные команды), и формат 2.6 с любым значением поля OP1.

- Одноформатные команды двойного размера с векторными регистрами могут использовать форматы 2.4 и 2.5, с любым значением полей OP1 и OP2 ≥ 8 (для одного и того же OP1 предоставляйте сходные команды), и формат 2.7 с OP1 в диапазоне от 16 до 63.

- Одноформатные команды тройного размера с регистрами общего назначения могут использовать формат 3.0 с OP1 в диапазоне от 16 до 63.

- Одноформатные команды тройного размера с векторными регистрами могут использовать формат 3.2 с любым значением поля OP1.

- Последующие команды, с длиной, большей трёх 32–разрядных слов, кодируются с IL = 3, Mode = от 4 до 7.
- Новые опции или другие модификации существующих команд могут использовать разряды OP3 или разряды регистра маски.
- Новые режимы адресации могут быть реализованы как одноформатные команды чтения и записи. Новые режимы адресации или иные модификации, которые применимы ко всем мультиформатным командам, могут использоваь для разрядов опций поле OP3. Если разрядов поля OP3 недостаточно, то возможно, в качестве последнего средства, использование значений поля OP2 из диапазона от 1 до 7.

Все неиспользуемые поля должны быть равны нулю. Для команд с наименьшим количеством входных операндов следует предпочесть наименьшие значения кодов в OP1.

Операнды назначаются следующим образом. Операнд–приёмник представляет собой регистр, указанный в поле RD. Операнды–источники используют регистровые поля RS, RT и RU, если только эти поля не задействованы для других целей (например, базового указателя, индекса, длины вектора). Если имеется находящийся в памяти операнд, либо непосредственно заданный операнд, то он должен быть последним из операндов–источников. Если выбранный формат имеет меньше операндов–источников, нежели необходимо для команды, то поле RD используется и как операнд–приёмник, и как первый из операндов–источников. Если всё ещё недостаточно операндов, то формат для конкретной команды использоваться не может. Если в формате имеется больше операндов, чем необходимо, то любой находящийся в памяти операнд или непосредственно заданный операнд будет последним операндом–источником, имея приоритет перед любым регистровым операндом. Неиспользуемые поля операндов должны быть равны нулю.

Глава 4. Списки команд

Команды ForwardCom перечислены в файле с разделителями в виде запятых, название которого — instruction_list.csv. Этот файл предназначен для использования ассемблерами, дизассемблерами, отладчиками, и эмуляторами. Данный список предварителен, и, возможно, изменится. Пожалуйста, помните, что списки в этом документе синхронизированы со списком в instruction_list.cvs.

Этот список команд имеет следующие поля:

Таблица 4.0.1. Поля в файле со списком команд

Поле	**Смысл**
Name	Имя команды, используемое ассемблером.
Category	1: одноформатная команда, 2: малая команда, 3: многоформатная команда, 4: команда перехода.
Formats	См. приводимую ниже табл. 4.0.2.
Template	Шестнадцатиричное число: 0xA–0xE для шаблонов A–E, 0x1 для малого шаблона, 0x0 для множественных шаблонов.
Source operands	Количество операндов–источников, включая регистровые операнды, находящиеся в памяти операнды, и непосредственно заданные операнды, но не включая маску, разряды опций, длину вектора, и индекс.
OP1	Код операции, OP1.
OP2	Дополнительный код операции, OP2. Ноль, если нет.
OP3 bits used	Количество разрядов в поле OP3, используемых для опций. Поле OP3 для счётчика сдвига в форматах 2.5 и 2.9 только если указанное здесь значение равно нулю.
Operand types general purpose registers	Шестнадцатиричное число, указывающее требуемую и необязательную поддержку каждого из типов операндов для регистров общего назначения. По поводу смысла каждого разряда см. приводимую ниже табл. 4.0.3.
Operand types scalar	Шестнадцатиричное число, указывающее требуемую и необязательную поддержку каждого из типов операндов для скалярных операций в векторных регистрах. По поводу смысла каждого разряда см. приводимую ниже табл. 4.0.3.
Operand types vector	Шестнадцатиричное число, указывающее требуемую и необязательную поддержку каждого из типов операндов для векторных операций. По поводу смысла каждого разряда см. приводимую ниже табл. 4.0.3.
Immediate operand type	Тип непосредственно заданного операнда для одноформатных команд. См. приводимую ниже табл. 4.0.4.
Description	Описания команды и комментарии.

Таблица 4.0.2. Смысл полей формата в файле со списком команд

Категория	**Интерпретация полей форматов**
1. Одноформатная команда.	Число из трёх шестнадцатиричных цифр. Крайняя левая цифра — значение поля IL (0-3). Средняя цифра — значение поля Mode, или комбинированного поля M+Mode (0-9). Крайняя правая цифра — подрежим, определяемый полем OP3 в форматах 2.4.x и 2.8.x, или OP1 в формате 2.7.x. Иначе нуль. Например, 0x283 означает формат 2.8.3.
2. Малая команда	0 Нет операндов. 1 RD = операнд–приёмник, являющийся регистром общего назначения, RS = непосредственно заданный операнд. 2 RD = операнд–приёмник, являющийся РОН[1], RS = регистр источник, являющийся РОН. 4 RD = операнд–приёмник, являющийся РОН, RS = указатель на находящийся в памяти операнд–источник. 5 RD = операнд–источник, являющийся РОН, RS = указатель на находящийся в памяти операнд–приёмник. 8 RD = операнд–приёмник, являющийся векторным регистром, RS — неиспользуется. 9 RD = операнд–приёмник, являющийся векторным регистром, RS — непосредственно заданный операнд. 10 RD = операнд–приёмник, являющийся векторным регистром, RS — являющий векторным регистром источник. 11 RD = операнд–источник, являющийся вектоным регистром, RS — являющийся приёмником РОН r0-r14,r31. 12 RD = операнд–приёмник, являющийся векторным регистром, RS = указатель на находящийся в памяти операнд–источник. 13 RD = операнд–источник, являющийся вектоным регистром, RS = указатель на находящийся в памяти операнд–приёмник.
3. Многоформатная команда	Шестнадцатиричное число, составленное из разрядов, каждый из которых отвечает за поддержку своего формата: 0x0000001 Формат 0.0: три регистра общего назначения. 0x0000002 Формат 0.1: два регистра общего назначения и 8–разрядное непосредственно заданное значение. 0x0000004 Формат 0.2: три векторных регистра. 0x0000008 Формат 0.3: два векторных регистра и 8–разрядное непосредственно заданное значение. 0x0000010 Формат 0.4: один вектор и находящийся в памяти операнд. 0x0000020 Формат 0.5: один вектор и находящийся в памяти операнд с отрицательным индексом. 0x0000040 Формат 0.6: один вектор и находящийся в памяти скалярный операнд с индексом. 0x0000080 Формат 0.7: один вектор и находящийся в памяти скалярный операнд с 8–разрядным смещением. 0x0000100 Формат 0.8: один РОН и находящийся в памяти операнд с индексом. 0x0000200 Формат 0.9: один РОН и находящийся в памяти операнд с 8–разрядным смещением. 0x0000400 Формат 2.0: два РОН и находящийся в памяти операнд с 32–разрядным смещением. 0x0000800 Формат 2.1: три РОН и 32–разрядное непосредственно заданное значение. 0x0001000 Формат 2.2: один векторный регистр и находящийся в памяти операнд с 32–разрядным смещением. 0x0002000 Формат 2.3: три векторных регистра и 32–разрядное непосредственно заданное значение.

[1]Регистр общего назначения

	0x0004000	Формат 2.4.0: два векторных регистра и находящийся в памяти скалярный операнд с 16–разрядным смещением.
	0x0008000	Формат 2.4.1: два векторных регистра и находящийся в памяти операнд с 16–разрядным смещением.
	0x0010000	Формат 2.4.2: два векторных регистра и находящийся в памяти операнд с отрицательным индексом.
	0x0020000	Формат 2.4.3: два векторных регистра и находящийся в памяти скалярный операнд с индексом и лимитом.
	0x0040000	Формат 2.5: три векторных регистра и сдвинутое 16–разрядное непосредственно заданное значение.
	0x0080000	Формат 2.8.0: три РОН и находящийся в памяти операнд с 16–разрядным смещением.
	0x0100000	Формат 2.8.1: два РОН и находящийся в памяти операнд с немасштабируемым индексом.
	0x0200000	Формат 2.8.2: два РОН и находящийся в памяти операнд с масштабируемым индексом.
	0x0400000	Формат 2.8.3: два РОН и находящийся в памяти скалярный операнд с индексом и лимитом.
	0x0800000	Формат 2.9: три РОН и сдвинутое 16–разрядное непосредственно заданное значение.
	0x1000000	Формат 3.1: три РОН и 64–разрядное непосредственно заданное значение (необязательно).
	0x2000000	Формат 3.3: три векторных регистра и 64–разрядное непосредственно заданное значение (необязательно).
4. Команда перехода	Шестнадцатиричное число, составленное из разрядов, каждый из которых отвечает за поддержку своего формата:	
	0x001	Формат 1.4: два регистра и 8–разрядное смещение.
	0x002	Формат 1.5 C: один регистр, 8–разрядное непосредственно заданное значение, и 8–разрядное смещение.
	0x004	Формат 1.5 C: 16–разрядное смещение.
	0x008	Формат 1.5 D: нет регистра, но есть 24–разрядное смещение.
	0x010	Формат 2.7.0: два регистра и 32–разрядное смещение.
	0x020	Формат 2.7.1: два регистра, 16–разрядное непосредственно заданное значение, и 16–разрядное смещение.
	0x040	Формат 2.7.2: один регистр, 8–разрядное непосредственно заданное значение, и 32–разрядное смещение.
	0x080	Формат 2.7.3: один регистр, 32–разрядное непосредственно заданное значение, и 8–разрядное смещение.
	0x100	Формат 2.7.4: системный вызов, 16–разрядный ID фунции и 32–разрядный ID модуля.
	0x200	Формат 3.0.1: два регистра, 32–разрядное непосредственно заданное значение, и 32–разрядное смещение.
	0x400	Формат 3.0.1: 64–разрядный абсолютный адрес.

Таблица 4.0.3. Указание типов операндов, поддерживаемых регистрами общего назначения, скалярами в векторных регистрах, или векторами. Данное значение представляет собой шестнадцатиричное число, составленное из разрядов, каждый из которых отвечает за поддержку своего формата

0x0001	поддержка 8–разрядных целых
0x0002	поддержка 16–разрядных целых
0x0004	поддержка 32–разрядных целых
0x0008	поддерка 64–разрядных целых
0x0010	поддержка 128–разрядных целых
0x0020	поддержка вещественных чисел одинарной точности

0x0040	поддержка вещественных чисел двойной точности
0x0080	поддержка вещественных чисел четырёхкратной точности
0x0100	необязательная поддержка 8–разрядных целых
0x0200	необязательная поддержка 16–разрядных целых
0x0400	необязательная поддержка 32–разрядных целых
0x0800	необязательная поддержка 64–разрядных целых
0x1000	необязательная поддержка 128–разрядных целых
0x2000	необязательная поддержка вещественных чисел одинарной точности
0x4000	необязательная поддержка вещественных чисел двойной точности
0x8000	необязательная поддержка вещественных чисел четырёхкратной точности

Таблица 4.0.4. Тип непосредственно заданного заданного операнда для однoформатных команд

0	нет аргументов или многоформатная команда
1	4–разрядное знаковое целое число
2	8–разрядное знаковое целое число
3	16–разрядное знаковое целое число
4	32–разрядное знаковое целое число
5	64–разрядное знаковое целое число
6	8–разрядное знаковое целое число, сдвигаемое на указанное количество разрядов
7	16–разрядное знаковое целое число, сдвигаемое на указанное количество разрядов
8	16–разрядное знаковое целое число, сдвигаемое на 16 разрядов
9	32–разрядное знаковое целое число, сдвигаемое на 32 разрядов
17	4–разрядное беззнаковое целое число
18	8–разрядное беззнаковое целое число
19	16–разрядное беззнаковое целое число
20	32–разрядное беззнаковое целое число
21	64–разрядное беззнаковое целое число
33	4–разрядное знаковое целое число, преобразуемое в вещественное число
34	8–разрядное знаковое целое число, преобразуемое в вещественное число
35	16–разрядное знаковое целое число, преобразуемое в вещественное число
39	16–разрядное знаковое целое число, сдвигаемое на указанное количество разрядов, и преобразуемое затем в вещественное число
64	вещественное число половинной точности
65	вещественное число одинарной точности
66	вещественное число двойной точности

Команды перехода перечислены на с. 21. Все прочие категории команд перечислены в последующих таблицах.

4.1. Список многоформатных команд

Следующий список команд охватывает общепринятые команды, которые можно закодировать в большинстве форматов или во всех форматах, предназначенных для многоформатных команд.

Таблица 4.1.1. Список многоформатных команд

Команда	**OP1**	**Операндов–источников**	**Описание**
nop	0	0	Нет операции.
move	1	1	Копирование значения.
store	2	1	Сохранение значения в памяти.

prefetch	3	1	Предвыборка из памяти.
sign_extend	4	1	Расширить меньшее целое знаком до 64 разрядов.
add	8	2	src1 + src2.
sub	9	2	src1 - src2.
sub_r	10	2	src2 - src1.
compare	11	2	Сравнение. Использует коды условий, см. с. 41.
mul	12	2	src1 · src2.
mul_hi_s	13	2	(src1 · src2) ≫ OS, знаково (только для целых чисел).
mul_hi_u	14	2	(src1 · src2) ≫ OS, беззнаково (только для целых чисел).
mul_ex_s	15	2	Перемножить элементы с чётными номерами векторов, состоящих из знаковых целых чисел, для получения результатов двойного размера.
mul_ex_u	16	2	Перемножить элементы с чётными номерами векторов, состоящих из беззнаковых целых чисел, для получения результатов двойного размера.
div	17	2	src1 / src2 (необязательно для целочисленных векторов).
rem	18	2	Остаток от деления (необязательно для целочисленных векторов).
min	20	2	Знаковый минимум.
max	21	2	Знаковый максимум.
min_u	22	2	Беззнаковый минимум для целых чисел, минимум из абсолютных величин для вещественных.
max_u	23	2	Беззнаковый максимум для целых чисел, максимум из абсолютных величин для вещественных.
and	32	2	src1 & src2.
and_not	33	2	src1 & (~src2).
or	34	2	src1 | src2.
xor	35	2	src1 ^ src2.
shift_left	36	2	src1 ≪ src2.
shift_rightu	37	2	src1 ≫ src2, с расширением нулём.
shift_rights	38	2	src1 ≫ src2, с расширением знаком.
rotate	39	2	Вращать влево, если src2 положительно, и вправо — если отрицательно.
extract_bit	40	2	Выделить разряд. (src1 ≫ src2) & 1.
set_bit	41	2	Установить разряд. src1 | (1 ≪ src2).
clear_bit	42	2	Сбросить разряд. src1 & ~ (1 ≪ src2).
toggle_bit	43	2	Инвертировать разряд. src1 ^ (1 ≪ src2).
mul_add	46	3	±src1 ± src2 · src3 (необязательно).
add_add	47	3	±src1 ± src2 ± src3 (необязательно).
userdef55 – userdef62	55-62	2	Зарезервировано для определяемых пользователем команд.
undef	63	2	Неопределённый код. Гарантированно возбуждает прерывание (trap) во всех будущих реализациях.

4.2. Список малых команд

В одном 32–разрядном слове кода помещается две малых команды. Если малая команда не может спариваться ни с чем иным, то она спаривается с малой nop.

Если не оговорено иное, то операнды малых команд — 64–разрядные. RD — регистр–приёмник, и, в большинстве случаев, также и первый из регистров–источников. RS может быть одним из регистров r0-r15, v0-v15, либо непосредственно заданной расширенной знаком 4–разрядной константой. Команды с указателем в RS используют в качестве указателя r0-r14, когда RS — от 0 до 14, и указатель стека (r31), когда RS равно 15.

Поскольку адрес команды должен быть кратен четырём, то невозможно перейти на вторую команду из пары малых команд. Если прерывание или ловушка возникли при выполнении малой команды, то обработчик

прерывания должен запомнить, какая из двух малых команд была прервана.

Таблица 4.2.1. Список малых команд, работающих с регистрами общего назначения

Команда	**OP1**	**Описание**
nop	0	Нет операции.
move	1	RD = расширенная знаком константа из поля RS.
add	2	RD += расширенная знаком константа из поля RS.
sub	3	RD −= расширенная знаком константа из поля RS.
shift_left	4	RD ≪= беззнаковая константа из поля RS.
shift_rightu	5	RD ≫= беззнаковая константа из поля RS (расширение нулём).
move	8	RD = регистровый операнд RS.
add	9	RD += регистровый операнд RS.
sub	10	RD −= регистровый операнд RS.
and	11	RD &= регистровый операнд RS.
or	12	RD |= регистровый операнд RS.
xor	13	RD ^= регистровый операнд RS.
move	14	Прочесть RD из находящегося в памяти операнда, имеющего заданный полем RS (RS = r0-r14, r31) указатель.
store	15	Записать RD в находящийся в памяти операнд, имеющий заданный полем RS (RS = r0-r14, r31) указатель.

Таблица 4.2.2. Список малых команд, работающих с векторными регистрами

Команда	**OP1**	**Описание**
clear	16	Очистить регистр RD, установив его длину равной нулю.
move	17	RD = знаковое 4–разрядное целое из RS, преобразованное в скаляр одинарной точности.
move	18	RD = знаковое 4–разрядное целое из RS, преобразованное в скаляр двойной точности.
move	19	RD = RS. Копировать вектор любого типа.
add	20	RD += RS, вектор из вещественных чисел одинарной точности.
add	21	RD += RS, вектор из вещественных чисел двойной точности.
sub	22	RD −= RS, вектор из вещественных чисел одинарной точности.
sub	23	RD −= RS, вектор из вещественных чисел двойной точности.
mul	24	RD *= RS,вектор из вещественных чисел одинарной точности.
mul	25	RD *= RS, вектор из вещественных чисел двойной точности.
add_cps	28	Получить размер сжатого образа для RD и прибавить его к регистру общего назначения RS.
sub_cps	29	Получить размер сжатого образа для RD и вычесть его из регистра общего назначения RS.
restore_cp	30	Восстановить векторный регистр RD из сжатого образа, на который указывает RS.
save_cp	31	Сохранить векторный регистр RD в сжатый образ, на который указывает RS.

4.3. Список одноформатных команд

Эти команды имеются преимущественно в одном или нескольких форматах.

Таблица 4.3.1. Список одноформатных команд, работающих с регистрами общего назначения

Команда	**Формат**	**OP1**	**Описание**
bitscan_f	1.0	1	Сканировать разряды вперёд. Найти номер самого младшего разряда RS, равного единице (необязательна).

bitscan_r	1.0	2	Сканировать разряды назад. Найти номер самого старшего разряда RS, равного единице (необязательна).
round_d2	1.0	3	Округлить RS к ближайшей меньшей степени числа 2.
round_u2	1.0	4	Округлить RS к ближайшей большей степени числа 2.
move	1.1	0	Переслать 16–разрядную расширенную знаком константу в регистр общего назначения RD.
move_u	1.1	1	Переслать 16–разрядную расширенную нулём константу в регистр общего назначения RD (может использоваться в качестве первого шага загрузки 32–разрядной константы, если не поддерживаются команды двойного размера).
add	1.1	2	Прибавить 16–разрядную расширенную знаком константу к RD.
sub	1.1	3	Вычесть 16–разрядную расширенную знаком константу из RD.
subr	1.1	4	Вычесть RD из 16–разрядной расширенной знаком константы.
mul	1.1	5	Умножить RD на 16–разрядную расширенную знаком константу.
div	1.1	6	Поделить RD на 16–разрядную расширенную знаком константу.
add	1.1	7	Сдвинуть 16–разрядную расширенную знаком константу влево на 16 разрядов и прибавить к RD.
move	1.1	16	RD = IM2 $\ll$ IM1. Знаково расширить IM2 до 64 разрядов и сдвинуть результат влево, на количество разрядов, указанное беззнаковым значением в IM1.
add	1.1	17	RD += IM2 $\ll$ IM1. Знаково расширить IM2 до 64 разрядов, сдвинуть результат влево, на количество разрядов, указанное беззнаковым значением в IM1, и прибавить то, что получилось, к RD.
and	1.1	18	RD &= IM2 $\ll$ IM1. Знаково расширить IM2 до 64 разрядов, сдвинуть результат влево, на количество разрядов, указанное беззнаковым значением в IM1, и выполнить поразрядное И с RD.
or	1.1	19	RD |= IM2 $\ll$ IM1. Знаково расширить IM2 до 64 разрядов, сдвинуть результат влево, на количество разрядов, указанное беззнаковым значением в IM1, и выполнить поразрядное ИЛИ с RD.
xor	1.1	20	RD ^= IM2 $\ll$. Знаково расширить IM2 до 64 разрядов, сдвинуть результат влево, на количество разрядов, указанное беззнаковым значением в IM1, и выполнить поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ с RD.
abs	1.8	0	Абсолютная величина целого числа. Использовать насыщение, если IM1 = 1.
shift_add	1.8	1	Сдвинуть и сложить. RD += RS $\ll$ IM1 (сдвинуть вправо, расширив нулём, если IM1 отрицательно).
read_spe	1.8	32	Прочитать содержимое специального регистра RS в РОН RD.
write_spe	1.8	33	Записать содержимое РОН RS в специальный регистр RD.
read_cpb	1.8	34	Прочитать содержимое регистра возможностей RS в РОН RD.
write_cpb	1.8	35	Записать содержимое РОН RS в регистр возможностей RD.
read_perf	1.8	36	Прочесть счётчик производительности.
read_perfs	1.8	37	Прочесть счётчик производительности, используя сериализацию.
read_sys	1.8	38	Прочесть содержимое системного регистра RS в РОН RD.
write_sys	1.8	39	Записать содержимое РОН RS в системный регистр RD.
load_hi	2.6	0	Загрузить 32–разрядную константу в старшую часть регистра общего назначения. Младшая часть зануляется. RD = IM2 $\ll$ 32.
insert_hi	2.6	1	Вставить 32–разрядную константу в старшую часть регистра общего назначения, оставляя младшую часть без изменения. RD = (RS & 0xFFFFFFFF) | (IM2 $\ll$ 32).

add_unsigned	2.6	2	Прибавить расширенную нулём 32–разрядную константу к регистру общего назначения.
sub_unsigned	2.6	3	Вычесть расширенную нулём 32–разрядную константу из регистра общего назначения.
add_hi	2.6	4	Прибавить 32–разрядную константу к старшей части регистра общего назначения. RD = RS + (IM2 $\ll$ 32).
and_hi	2.6	5	Поразрядное И старшей части регистра общего назначения с 32–разрядной константой. RD = RS & (IM2 $\ll$ 32).
or_hi	2.6	6	Поразрядное ИЛИ старшей части регистра общего назначения с 32–разрядной константой. RD = RS | (IM2 $\ll$ 32).
xor_hi	2.6	7	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ старшей части регистра общего назначения с 32–разрядной константой. RD = RS ^ (IM2 $\ll$ 32).
address	2.6	32	RD = RS + IM2, RS может быть THREADP (28), DATAP (29), или IP (30).

Таблица 4.3.2. Список одноформатных команд, работающих с векторными регистрами и регистрами разного типа

Команда	**Формат**	**OP1, OP2**	**Описание**
set_len	1.2	0	RD = векторный регистр RT со значением длины, указанным в RS.
get_len	1.2	1	Записать длину векторного регистра RS в регистр общего назначения RD.
set_num	1.2	2	Заменить длину векторного регистра на RS·OS.
get_num	1.2	3	Получить длину векторного регистра, делённую на размер операнда.
compress	1.2	4	Сжать вектор RT длины RS до вектора половинной длины и половинного размера элемента: двойная точность $\rightarrow$ одинарная точность, 64–разрядное целое $\rightarrow$ 32–разрядное целое, и т.п.
compress_ss	1.2	5	Сжать целочисленный вектор RT длины RS до вектора половинной длины и половинного размера элемента, знаково и с насыщением (необязательна).
compress_us	1.2	6	Сжать целочисленный вектор RT длины RS до вектора половинной длины и половинного размера элемента, беззнаково и с насыщением (необязательна).
expand	1.2	7	Расширить вектор RT длины RS/2 и половинного размера элемента до вектора длины RS с полного размера элемента: половинная точность $\rightarrow$ одинарная точность, 32–разрядное целое $\rightarrow$ 64–разрядное целое (с расширением знаком), и т.п.
expand_us	1.2	8	Расширить вектор RT длины RS/2 и половинного размера элемента до вектора длины RS с полного размера элемента: 32–разрядное целое $\rightarrow$ 64–разрядное целое (с расширением нулём), и т.п.
compress_sparse	1.2	9	Сжать разреженный вектор с элементами, указанными разрядами маски, в непрерывный вектор. RS = длина входного вектора (необязательна).
expand_sparse	1.2	10	Расширить непрерывный вектор до разреженного вектора, позиции в котором указаны разрядами маски. RS = длина результирующего вектора (необязательна).
extract	1.2	11	Выделить один элемент из вектора RT, начинающийся со смещения RS·OS и имеющий размер OS, в скаляр, находящийся в векторном регистре RD.
insert	1.2	12	Заменить один элемент вектора RD, начинающийся со смещения RS·OS, скаляром из RT.

broadcast	1.2	13	Разослать первый элемент вектора RT во все элементы вектора RD, имеющего длину, указанную в RS.
bits2bool	1.2	14	Младшие n разрядов из RT распаковываются в булев вектор RD длины RS, с одним разрядом в каждом элементе, где n = RS / OS.
bool2bits	1.2	15	Булев вектор RT длины RS упаковывается в младшие n разрядов RD, беря разряд с номером 0 в каждом элементе, n = RS / OS. Длина RD должна быть достаточна для хранения, по меньшей мере, n разрядов.
bool_reduce	1.2	16	Логический вектор RT длины RS сворачивается, посредством комбинирования разрядов номер 0 всех элементов. Результатом является скалярное целое, в котором разряд №0 является поразрядным И всех разрядов с номером 0, а разряд №1 — поразрядными ИЛИ. Остальные разряды зарезервированы для последующего использования.
shift_expand	1.2	18	Сдвинуть вектор RT влево на RS байтов и увеличить длину вектора на RS. Младшие RS байтов в RD будут нулями.
shift_reduce	1.2	19	Сдвинуть вектор RT вправо на RS байтов и уменьшить длину вектора на RS. Младшие RS байт вектора RT теряются.
shift_up	1.2	20	Сдвинуть вектор RT влево на RS элементов. Младшие RS элементов вектора RD будут нулями, а старшие RS элементов из RT — теряются.
shift_dn	1.2	21	Сдвинуть вектор RT вправо на RS элементов. Старшие RS элементов вектора RD будут нулями, а младшие RS элементов из RT — теряются.
div_ex_s	1.2	24	Разделить вектор RS, состоящий из целых чисел двойного размера, на знаковые целые числа из RT. Размер элемента RS равен 2·OS. Деление выполняется на элементы вектора RT (размер элемента которого равен OS), имеющие чётные номера. Частные сохраняются в элементах вектора RD, имеющих чётные номера, а остатки — в элементах с нечётными номерами (для векторов — необязательно).
div_ex_u	1.2	25	То же, но для беззнаковых целых (для векторов — необязательно).
sqrt	1.2	26	Квадратный корень (для вещественных чисел, необязательно).
add_c	1.2	28	Сложить с переносом. Вектор имеет два элемента. Старший элемент используется как перенос и на входе, и на выходе (необязательна).
sub_b	1.2	29	Вычесть с заёмом. Вектор имеет два элемента. Старший элемент используется как заём и на входе, и на выходе (необязательна).
add_ss	1.2	30	Сложить целочисленные векторы, со знаковым насыщением (необязательна).
add_us	1.2	31	Сложить целочисленные векторы, с беззнаковым насыщением (необязательна).
sub_ss	1.2	32	Вычесть целочисленные векторы, со знаковым насыщением (необязательна).
sub_us	1.2	33	Вычесть целочисленные векторы, с беззнаковым насыщением (необязательна).
mul_ss	1.2	34	Умножить целочисленные векторы, со знаковым насыщением (необязательна).
mul_us	1.2	35	Умножить целочисленные векторы, с беззнаковым насыщением (необязательна).
shl_ss	1.2	36	Сдвинуть влево целочисленные векторы, со знаковым насыщением (необязательна).

shl_us	1.2	37	Сдвинуть влево целочисленные векторы, с беззнаковым насыщением (необязательна).
add_oc	1.2	38	Сложить с проверкой переполнения (необязательна).
sub_oc	1.2	39	Вычесть с проверкой переполнения (необязательна).
subr_oc	1.2	40	Обратное вычитание с проверкой переполнения (необязательна).
mul_oc	1.2	41	Умножить с проверкой переполнения (необязательна).
div_oc	1.2	42	Деление с проверкой переполнения (необязательна).
input	1.2	48	Прочесть из порта ввода. RD = векторный регистр, RT = адрес порта, RS = длина вектора (привилегированная команда).
output	1.2	49	Записать в порт вывода. RD = являющийся векторным регистром операнд–источник, RT = адрес порта, RS = длина вектора (привилегированная команда).
gp2vec	1.3 B	0	Переслать значение из регистра общего назначения RS в находящийся в векторном регистре RD скаляр.
set_bits_x	1.3 B	1	Установить все разряды, кроме одного. RD = RS | ~ (1 ≪ IM1).
clear_bits_x	1.3 B	2	Сбросить все разряды, кроме одного. RD = RS & (1 ≪ IM1).
make_sequence	1.3 B	3	Создать вектор, состоящий из RS последовательных чисел. Первое значение равно IM1.
mask_length	1.3 B	4	Создать маску с true в первых RS байтах. Разряды опций — в IM1.
vec2gp	1.3 B	8	Переслать значение первого элемента векторного регистра RS в регистр общего назначения RD.
bitscan_f	1.3 B	9	Сканировать разряды вперёд. Найти индекс самого младшего разряда регистра RS, равного единице (для векторов необязательна).
bitscan_r	1.3 B	10	Сканировать разряды назад. Найти индекс самого старшего разряда регистра RS, равного единице (для векторов необязательна).
float2int	1.3 B	12	Преобразование вещественного значения в целочисленное значение, имеющее тот же размер операнда. Режим округления указывается в IM1.
int2float	1.3 B	13	Преобразование целочисленного значения в вещественное значение, имеющее тот же размер операнда.
round	1.3 B	14	Округлить вещественное число к целочисленному значению в вещественном представлении. Режим округления указывается в IM1.
round2n	1.3 B	15	Округлить к ближайшему кратному числа 2^n. RD = $2^n \cdot$ round($2^{-n} \cdot$ RS). n — знаковая целочисленная константа, находящаяся в IM1 (необязательна).
abs	1.3 B	16	Абсолютная величина целого числа. Если IM1 = 1, то используется насыщение.
popcount	1.3 B	17	Подсчитать количество разрядов регистра RS, равных 1.
broadcast	1.3 B	18	Разослать 8–разрядную константу во все элементы регистра RD, имеющего длину, указанную в регистре RS (31 в поле RS даёт скалярный результат).
fp_category	1.3 B	19	Проверить, принадлежат ли вещественные числа категориям, указанным константой.
byte_reverse	1.3 B	20	Обратить порядок байтов в каждом элементе вектора.
bit_reverse	1.3 B	21	Обратить порядок разрядов в каждом элементе вектора (необязательна).
truth_tab2	1.3 B	24	Булева функция от двух переменных, заданная таблицей истинности.
read_spev	1.3 B	30	Прочесть специальный регистр RT в векторный регистр RD, имеющий длину RS.

move	1.3 C	32	Переслать 16–разрядную константу в 16–разрядный скаляр (необязательна).
add	1.3 C	33	Сложить размноженную 16–разрядную константу с 16–разрядными элементами вектора (необязательна).
and	1.3 C	34	Выполнить поразрядное И размноженной 16–разрядной константы с 16–разрядными элементами вектора (необязательна).
or	1.3 C	35	Выполнить поразрядное ИЛИ размноженной 16–разрядной константы с 16–разрядными элементами вектора (необязательна).
xor	1.3 C	36	Выполнить поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ размноженной 16–разрядной константы с 16–разрядными элементами вектора (необязательна).
move	1.3 C	38	RD = IM2 $\ll$ IM1. Расширить IM2 знаком до 32 разрядов и сдвинуть результат влево на беззнаковое значение IM1, чтобы получить 32–разрядный скаляр (необязательна).
move	1.3 C	39	RD = IM2 $\ll$ IM1. Расширить IM2 знаком до 64 разрядов и сдвинуть результат влево на беззнаковое значение IM1, чтобы получить 64–разрядный скаляр (необязательна).
add	1.3 C	40	RD += IM2 $\ll$ IM1. Сложить размноженную сдвинутую знаковую константу с 32–разрядными элементами вектора (необязательна).
add	1.3 C	41	RD += IM2 $\ll$ IM1. Сложить размноженную сдвинутую знаковую константу с 64–разрядными элементами вектора (необязательна).
and	1.3 C	42	RD &= IM2 $\ll$ IM1. Выполнить поразрядное И размноженной сдвинутой знаковой константы с 32–разрядными элементами вектора (необязательна).
and	1.3 C	43	RD &= IM2 $\ll$ IM1. Выполнить поразрядное И размноженной сдвинутой знаковой константы с 64–разрядными элементами вектора (необязательна).
or	1.3 C	44	RD |= IM2 $\ll$ IM1. Выполнить поразрядное ИЛИ размноженной сдвинутой знаковой константы с 32–разрядными элементами вектора (необязательна).
or	1.3 C	45	RD |= IM2 $\ll$ IM1. Выполнить поразрядное ИЛИ размноженной сдвинутой знаковой константы с 64–разрядными элементами вектора (необязательна).
xor	1.3 C	46	RD ^= IM2 $\ll$ IM1. Выполнить поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ размноженной сдвинутой знаковой константы с 32–разрядными элементами вектора (необязательна).
xor	1.3 C	47	RD ^= IM2 $\ll$ IM1. Выполнить поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ размноженной сдвинутой знаковой константы с 64–разрядными элементами вектора (необязательна).
add	1.3 C	48	RD += IM21 $\ll$ 16. Сложить размноженную знаковую сдвинутую влево на 16 разрядов 16–разрядную константу с 32–разрядными элементами вектора (необязательна).
add	1.3 C	49	RD += IM21 $\ll$ 16. Сложить размноженную знаковую сдвинутую влево на 16 разрядов 16–разрядную константу с 64–разрядными элементами вектора (необязательна).
mov	1.3 C	56	Переслать преобразованную вещественную константу половинной точности в скаляр одинарной точности (необязательна).
mov	1.3 C	57	Переслать преобразованную вещественную константу половинной точности в скаляр двойной точности (необязательна).
add	1.3 C	58	Сложить размноженную вещественную константу половинной точности с каждым элементом вектора, состоящего из чисел одинарной точности (необязательна).

add	1.3 C	59	Сложить размноженную вещественную константу половинной точности с каждым элементом вектора, состоящего из чисел двойной точности (необязательна).
mul	1.3 C	60	Перемножить размноженную вещественную константу половинной точности с каждым элементом вектора, состоящего из чисел одинарной точности (необязательна).
mul	1.3 C	61	Перемножить размноженную вещественную константу половинной точности с каждым элементом вектора, состоящего из чисел двойной точности (необязательна).
permute	2.5	2, 8	Элементы вектора RT переставляются в пределах каждого блока размером RS байтов, индексы указаны в регистре RU. Каждый индекс отсчитывается относительно начала блока. Индекс вне диапазона порождают нули. Максимальный размер блока зависит от реализации.
concatenate	2.5	2, 9	Вектор RT длины RS и вектор RU длины RS склеиваются в вектор RD длины 2·RS.
truth_tab3	2.5	3, 8	Булева функция от трёх переменных, заданная таблицей истинности (необязательна).
truth_tab4	2.5	4, 8	Булева функция от четырёх переменных, заданная таблицей истинности (необязательна).
mul_add	2.5	3, 9	RD = ± RS ± RT · RU (необязательна, но рекомендуется).
add_add	2.5	3, 10	RD = ± RS ± RT ± RU (необязательна).
add_add_add	2.5	3, 11	RD = ± RS ± RT ± RU ± IM2 Сложить три векторных регистровых операнда и 16–разрядную константу IM2 (необязательна).
add_add_add	2.5	4, 11	RD = ± RD ± RS ± RT ± RU Сложить четыре векторных регистровых операнда (необязательна).
load_hi	2.7	16	Сделать вектор из двух элементов. dest[0] = 0, dest[1] = IM2.
insert_hi	2.7	17	Сделать вектор из двух элементов. dest[0] = src1[0], dest[1] = IM2.
make_mask	2.7	18	Создать вектор, в котором разряд №0 каждого элемента идёт из разрядов поля IM2, а оставшиеся разряды — из RS.
replace	2.7	19	Заменить элементы в RS константой IM2.
replace_even	2.7	20	Заменить элементы в RS, имеющие чётные номера, константой IM2.
replace_odd	2.7	21	Заменить элементы в RS, имеющие нечётные номера, константой IM2.
broadcast	2.7	22	Разослать 32–разрядную константу во все элементы регистра RD, имеющего длину, указанную в регистре RS (31 в поле RS даёт скалярный результат).
permute	2.7	33	Элементы вектора RT переставляются в пределах каждого блока размером RS байтов. 4·n разрядов поля IM2 используется в качестве индекса, с 4 разрядами на каждый элемент в блоке размера n. Тот же шаблон используется в каждом блоке. Количество элементов в каждом блоке: n = RS / OS $\leq$ 8.

Таблица 4.3.3. Список одноформатных команд с операндами, находящимися в памяти.

Команда	**Формат**	**OP1, OP2**	**Описание**
store	2.7 B	48	Сохранить 32–разрядную константу IM2 в находящемся в памяти операнде, имеющем базу RT и 8–разрядное смещение IM1 (необязательна).
fence	2.4.x	0, 8	Барьер в памяти, по чтению, по записи, или полный (указывается OP3).
cmp_swap	2.8.x	1, 8	Атомарное сравнение и обмен.

read_insert	2.4.0 2.4.3	2, 8	Заменить в векторе RD один элемент, начинающийся со смещения RS·OS, скалярным операндом, находящимся в памяти (необязательна).
move_store	2.4.x	3, 8	Условная пересылка и сохранение. Разряды маски = 01 или 11: сохранить RU. Разряды маски = 10: сохранить zero. Разряды маски = 11: сохранить RD. (необязательна).
extract_store	2.4.0	3, 9	Выделить один элемент из вектора RD, начинающийся со смещения RS·OS и имеющий размер OS, в находящийся в памяти операнд с базой RT и смещением IM2 (необязательна).
extract_store	2.4.3	3, 9	Выделить один элемент из вектора RD, начинающийся со смещения RS·OS и имеющий размер OS, в находящийся в памяти операнд с базой RT, масштабируемым индексом RU и беззнаковым лимитом RU $\leq$ IM2 (необязательна).
compress_store	2.4.1	3, 10	Сжать вектор RD длины RS в вектор половинной длины и половинного размера элемента. Двойная точность $\rightarrow$ одинарная точность, 64–разрядное целое $\rightarrow$ 32–разрядное целое, и т.п. Сохранить в памяти с базой RT, смещением IM2, длиной RS/2 (необязательна).
add_store	2.4.x	4, 8	Сложить RD и RU, сохранить результат в находящемся в памяти операнде (необязательна).
sub_store	2.4.x	4, 9	Вычесть RU из RD, сохранить результат в находящемся в памяти операнде (необязательна).
mul_store	2.4.x	4, 10	Перемножить RD и RU, сохранить результат в находящемся в памяти операнде (необязательна).
read_memory_map	2.4.2	48, 8	Прочесть карту памяти. RD = элемент карты, RT = указатель памяти, RS = длина вектора и отрицательный индекс как для источника, так и для приёмника (привилегированная).
write_memory_map	2.4.2	48, 9	Записать карту памяти. RD = элемент карты, RT = указатель памяти, RS = длина вектора и отрицательный индекс как для источника, так и для приёмника (привилегированная).

4.4. Описание команд

В данном разделе описываются команды, для которых нужны специальные объяснения.

Многоформатные команды

nop

Рекомендуется кодировать команды NOP в виде 32–разрядного слова, все разряды которого равны нулю. Процессору разрешается пропускать такого рода команды NOP на ранней стадии конвейера, и настолько быстро, насколько он может. Пара малых команд, в которой вторая команд — NOP, может рассматриваться как одна команда.

Эти команды NOP могут использоваться только как заполнители, но не для задержек.

move

Копирование значения из находящегося в памяти операнда, регистра, или непосредственно заданной константы в регистр. Если приёмник — векторный регистр, а источник — непосредственно заданная константа, то результат будет скаляром. Значение размножено не будет, поскольку нет никакого иного входного операнда, который бы указывал длину вектора. Если желателен вектор, то используйте команду broadcast.

Команда move с непосредственно заданным операндом — предпочтительный метод обнуления регистра.

Команда move имеет несколько дополнительных малых и одноформатных вариантов. Ассемблер, как правило, выберет самый короткий вариант, в который помещаются указанные операнды.

store

Операнды–источники и операнды–приёмники меняются ролями, так что значение из RD записывается в находящийся в памяти операнд. Допускаются только форматы, которые указывают находящийся в памяти операнд (скалярный, либо векторный без размножения).

Размер находящегося в памяти операнда определяется полем размера операнда (OS), когда указан находящийся в памяти скалярный операнд, или длиной векторного регистра в RS, когда указан находящийся в памяти векторный операнд.

Аппаратура должна быть в состоянии обрабатывать размеры находящихся в памяти операндов, не являющиеся степенями двойки, не используя при этом дополнительной памяти (чтение и перезапись за пределами находящегося в памяти операнда — не допускаются, разве что обращение из других потоков блокировано во время выполнения операции, и подавляются исключения отсутствия доступа). Разрешается записывать операнд по кусочкам.

Замаскированная операция с разрядами маски №№ 0 и 1, равными нулю одновременно, запишет в память ноль.

Замаскированная операции с разрядом №0, равным 0, и разрядом №1, равным единице, для векторных регистров может поддерживаться, а может и не поддерживаться. Если поддерживается, то эта комбинация оставит ячейку памяти без изменения, и не может быть реализована как чтение, совмещённое с записью, поскольку это не было бы потокобезопасно.

prefetch

Предвыборка находящегося в памяти операнда в кэш. Разные варианты могут указываться разрядами №№0–3 поля OP3 форматов 2.4 и 2.8.

sign_extend

Входным операндом может быть 8–разрядное, 16–разрядное, или 32–разрядное целое число. Это целое число расширяется знаком, чтобы получить 64–разрядный результат в регистре общего назначения или скаляр в векторном регистре. Если входной операнд — вектор, то в каждом 64–разрядном блоке используется только первый элемент. Вещественные типы использоваться не могут.

min and max

min(src1,src2) = src1 < src2 ? src1 : src2

max(src1,src2) = src1 < src2 ? src1 : src2

Операнды рассматриваются как знаковые. Есть также версия для беззнаковых целых:

min_u(src1,src2) = src1 < src2 ? src1 : src2

max_u(src1,src2) = src1 < src2 ? src1 : src2

Когда беззнаковая версия применяется к вещественным операндам, она берёт абсолютные величины операндов, и имя команды изменяется:

min_abs(src1, src2) = min(abs(src1), abs(src2))

max_abs(src1, src2) = max(abs(src1), abs(src2))

Обработка вещественных операндов, являющихся нечислами (NAN), определяется разрядом №22 регистра маски или вещественного управляющего регистра. Если разряд №22 равен нулю, то, в соответствии с IEEE Standard 754-2008, результатом является не являющийся нечислом операнд. Если разряд №22 равен единице, то входной операнд, являющийся NAN, распространяется дальше.

Являющийся NAN операнд, который не является распространяемым, породит ловушку, если установлен флаговый разряд №29.

Поразрядные логические команды

Это следующие команды: and, and_not, or, xor. Вещественные операнды обрабатываются точно так же, как и целочисленные.

Команды манипуляции разрядами

Для манипуляции разрядами предоставляются следующие команды:

extract_b: выделить разряд с номером src2 из src1

set_b: заменить в src1 разряд с номером src2 на 1

clear_b: заменить в src1 разряд с номером src2 на 0

toggle_b: инвертировать в src1 разряд с номером src2

Вещественный операнд в src1 рассматривается как целое число такого же размера. Индекс разряда в src2, независимо от типа операнда, интерпретируется как 8–разрядное беззнаковое целое число.

Эти команды могут быть реализованы с 8–разрядной непосредственно заданной константой для src2, вместо большей константы, которая потребовалась бы, если бы мы использовали команды AND, OR, XOR для манипулирования одиночными разрядами. Эти команды могут также использоваться с вещественными числами, в основном для манипулирования знаковым разрядом.

mul_add

Совмещённые умножение и сложение.

dest = ± src1 ± (src2 · src3)

Команда совмещённого умножения и сложения часто может значительно улучшить производительность вещественного кода.

Поддерживаются только форматы, допускающие три операнда.

Знаки операндов могут быть изменены, что указывается в разрядах №№0–3 поля OP3 в форматах, использующих шаблон E2, включая дополнительный формат 2.5, при этом

разряд 0: изменить знак у элементов вектора src1 с чётными номерами

разряд 1: изменить знак у элементов вектора src1 с нечётными номерами

разряд 2: изменить знак у элементов вектора src2·src3 с чётными номерами

разряд 3: изменить знак у элементов вектора src2·src3 с нечётными номерами

Команда делает возможным выполнение умножения–и–сложения, умножения–и–вычитания, умножения–и обратного–вычитания, и т.п. Можно также выполнять умножение с чередующимися сложением и вычитанием, что полезно при вычислениях с комплексными числами. В других форматах, в которых поле OP3 отсутствует, смены знака нет. Поддерживается дополнительная одноформатная версия mul_add, с четырьмя регистровыми операндами и полем OP3.

Поле OP3 не используется как счётчик сдвига в форматах 2.5 и 2.9.

Поддержка целочисленных операндов — необязательна. Поддержка вещественных операндов — необязательна, но желательна.

add_add

Два сложения в одной команде.

dest = ± src1 ± src2 ± src3

Поддерживаются только форматы, допускающие три операнда.

Знаки операндов могут быть изменены, что указывается в разрядах №№0–2 поля OP3 в форматах, использующих шаблон E2, включая дополнительный формат 2.5, при этом

bit 0: изменить знак src1

bit 1: изменить знак src2

bit 2: изменить знак src3

В других форматах, в которых поле OP3 отсутствует, смены знака нет. Поддерживается дополнительная одноформатная версия add_add, с четырьмя регистровыми операндами и полем OP3.

Поле OP3 не используется как счётчик сдвига в форматах 2.5 и 2.9.

Желательна точность вещественных операндов, меньшая единицы самого младшего разряда численно самого большого операнда, но промежуточный результат не вычисляется с неограниченной точностью. Аппаратная реализация может подстроить экспоненты для всех операндов за первый такт, и использовать сеть сумматора цепи умножения.

Данную команду следует поддерживать, только если её можно реализовать так, чтобы она была быстрее двух последовательных команд сложения. Она может поддерживаться для целочисленных операндов, вещественных операндов, или и для тех, и для других. См. также команду add_add_add (с. 49).

Команды сравнения

Команда сравнения сравнивает два операнда–источника и сохраняет результат в разряде №0 приёмника. Когда используются форматы 0.0-0.3 или 2.0-2.3, условие определяется добавочным кодом, сохранённым в третьем операнде–источнике. Форматы, использующие шаблон E2 (2.4, 2.5, 2.8, 2.9) кодируются иначе: код условия содержится в поле OP3. Шестнадцатиразрядное поле IM2 форматов 2.5 и 2.9 используется как второй операнд–источник, который не сдвигается на OP3.

Оставшиеся разряды результата копируются из регистра маски, или из численного управляющего слова, если маска не используется. Это подходит, когда результат используется как маска.

Коды условий определены в следующей таблице:

Таблица 4.4.1. Коды условий для команд сравнения

Разряд	Смысл
0	Инвертирует условие.
1-2	Определяет условие: 0 = меньше, 1 = равно, 2 = больше, 3 = неупорядочено.
3	Для целочисленных операндов: 0 = знаковые операнды, 1 = беззнаковые операнды. Для вещественных операндов: данный разряд указывает результат, если один операнд, либо оба, являются нечислами (NAN).

Команды сравнения могут быть замаскированы. Разряд №0 результата равен разряду №1 регистра маски, если разряд №0 регистра маски равен нулю.

Команды малого формата

clear

Данная команда устанавливает длину векторного регистра равной нулю. Всё содержимое теряется. Регистр может затем рассматриваться как неиспользуемый.

Операции push и pop

Команд push и pop нет. Регистр общего назначения R можно втолкнуть в стек с помощью следующей пары малых команд:

```
add sp,-8
store [sp],R
```

Регистр общего назначения R можно вытолкнуть из стека с помощью следующей пары малых команд:

```
move R,[sp]
sub sp,-8
```

Обратите внимание, что константа -8 может содержаться в 4–разрядном знаковом поле RS, а константа 8 — нет. Именно по этой причине мы прибавляем и вычитаем -8, а не выполняем обратные операции с +8.

Для этих последовательностей ассемблер может поддерживать макросы с именами push и pop.

Сохранение и восстановление векторных регистров

При сохранении векторного регистра переменной длины мы не хотим сохранять максимальную длину, когда используется только часть регистра. Как следствие, вы имеем команды save_cp и restore_cp, предназначенные для сохранения и восстановления векторных регистров без использования большего объёма памяти, нежели необходимо.

Обратите внимание, что формат сохранённого образа зависит от реализации. Как правило, команда save_cp сохранит длину вектора, за которой следует столько байтов, сколько указано длиной, а команда restore_cp прочтёт длину, а затем прочтёт столько байтов, сколько указано длиной.

Микропроцессору разрешается сжимать данный любым способом, который можно обработать достаточно быстро. Например, булев вектор, использующий только один разряд в каждом элементе, может, очевидно, быть сжат до гораздо меньшего размера. Образ неиспользуемого векторного регистра будет, как правило, содержать несколько нулевых байтов для длины.

Программному обеспечению никогда не следует использовать сохранённый образ ни для чего иного, нежели восстановление векторного регистра на той же модели микропроцессора, что сохранила его, поскольку формат образа несовместим у разных моделей.

Размер сохранённого образа можно прибавить к указателю командой add_cps или вычесть из указателя командой sub_cps. Поле RS задаёт указатель, которым может быть r0–14 или r31 (указатель стека).

Векторный регистр V можно сохранить в стеке (втолкнуть в стек) с помощью следующей пары малых команд:

```
sub_cps sp,V
save_cp [sp],V
```

Векторный регистр V можно восстановить из стека (вытолкнуть) с помощью следующей пары малых команд:

```
restore_cp V,[sp]
add_cps sp,V
```

Те же команды можно использовать для сохранения регистров при переключении задач. При сохранении таким способом, неиспользуемые векторные регистры будут использовать очень мало места.

Размер сжатого образа, указываемого командами add_cps и sub_cps, при использовании указателя стека должен быть кратным 8, чтобы сохранить должное выравнивание стека.

При переключении задач, когда, как правило, используется другой указатель команд, разрешается использовать меньший размер, не кратный 8. В этом случае для управления форматом сохранённого образа должен быть предоставлен управляющий регистр.

Команде restore_cp разрешается читать больше байтов, чем необходимо, вплоть до значения, равного максимальной длине вектора плюс 8 байт, и отбрасывать впоследствии, когда станет известна действительная длина, любые избыточные байты.

Одноформатные команды, использующие регистры общего назначения и специальные регистры

read_spe, write_spe

Чтение или запись специального регистра. Ниже указаны определённые на данный момент специальные регистры. Разрядность каждого равна 64. Эти регистры инициализируются их принятыми по умолчанию значениями в момент запуска программы.

Таблица 4.4.2. Список специальных регистров

Номер специального регистра	Смысл
0	Численный управляющий регистр (NUMCONTR).
1	Идентификатор (ID) марки микропроцессора
2	Номер версии микропроцессора.
28	Указатель блока окружения потока (THREADP).
29	Указатель секции данных (DATAP).

read_cpb, write_cpb

Чтение или запись регистра возможностей процессора. Эти регистры используются для указания возможностей процессора, таких, как поддержка необязательных команд и ограничения на длину вектора. Разрядность такого регистра равна 64. В момент запуска программы эти регистры инициализируются принятыми для них по умолчанию значениями.

Находящаяся в IM1 непосредственно заданная константа определяет детали операции:

Таблица 4.4.3. Смысл непосредственно заданной константы в командах read_cpb и write_cpb

Номер разряда	**Смысл**
0	0: чтение/запись возможностей для типа операнда, указанного в разрядах №№5-7. 1: чтение типичных возможностей для всех типов операндов / запись возможностей для всех соответствующих типов операндов.
1	0: чтение текущего значения регистра, которое, возможно, изменено. 1: чтение реальных возможностей аппаратуры (записывать нельзя).
5-7	Тип операнда для возможностей.

Таблица 4.4.4. Список регистров возможностей

Номер регистра возможностей	**Смысл**
0	Максимальная длина вектора для команд общего назначения.
1	Максимальная длина вектора для команд перестановки.
2	Максимальная длина блока для команд перестановки.
3	Максимальная длина вектора для compress_sparse и expand_sparse.
8	Поддержка необязательных команд, работающих с регистрами общего назначения. Каждый разряд отвечает за свою команду.
9	Поддержка необязательных команд, работающих со скалярами в векторных регистрах. Каждый разряд отвечает за свою команду.
10	Поддержка необязательных команд, работающих с векторами. Каждый разряд отвечает за свою команду.

Изменение значений максимальной длины вектора оказывает следующее влияние. Если максимальная длина уменьшается до величины, меньшей физически возможной, то любая попытка сделать более длинный вектор даст вектор с уменьшенной длиной. Поведение векторных регистров, у которых до уменьшения максимальной длины уже была большая длина, зависит от реализации. Если максимальная длина устанавливается равной значению, которое больше физически возможного, то любая попытка сделать вектор размера, большего физически возможного, вызовет ловушку, для облегчения эмуляции. Регистры возможностей с номерами 0–3 для целей эмуляции могут быть увеличены. Значение регистров возможностей с номерами 0–3 должно быть степенью двойки.

Регистры возможностей с номерами 8–9 могут модифицироваться, для тестовых целей или для того, чтобы сообщить программному обеспечению: „не используй конкретные команды“. То же значение будет возвращаться при чтении регистра. Попытка выполнить неподдерживаемую команду вызовет ловушку, независимо от значения регистра возможностей.

read_sys, write_sys

Эти команды предназначены для обращения к различным регистрам, доступным только для чтения.

read_perf

Прочесть внутренний счётчик тактов, количество выполненных команд, или иные связанные с производительностью счётчики.

read_perfs

То же, что и read_perf. Данная команда сериализуется. Это означает, что она не может выполняться неупорядоченно.

popcount

Команда popcount считает количество единичных разрядов в целом числе. Она может также использоваться для генерирования чётности.

bitscan_f

Сканировать разряды вперёд.
Найти индекс самого младшего единичного разряда, т.е. наибольшее X, для которого
(((1 ≪ X) - 1) & src1)) == 0.

bitscan_r

Сканировать разряды назад.
Найти индекс самого старшего единичного разряда, т.е. наибольшее X, для которого (1 ≪ X) ≤ src1.

round_d2

Округлить к ближайшей степени двойки, т.е. 1 ≪ bit_scan_reverse(src1).

round_u2

Округлить к ближайшей большей степени двойки, т.е.
(S & (S-1)) == 0 ? S : 1 ≪ (bit_scan_reverse(S) + 1),
где S = src1.

shift_add

Сдвинуть и сложить, dest = src1 + (src2 ≪ src3); src1 использует тот же регистр, что dest; src3 — 8–разрядная знаковая непосредственно заданная константа.

Если src3 — отрицательно, то будет выполняться сдвиг вправо.

address

Вычислить адрес относительно указателя, сложив 32–разрядную расширенную знаком константу с регистром общего назначения или специальным регистром. Регистром–указателем может быть r0-r27, THREADP (28), DATAP (29), IP (30) или SP(31).

cmp_swap

Команда атомарного сравнения и обмена, используемая для синхронизации и безблокировочного разделения данных между потоками; src1 и src2 — регистровые операнды; src3 — находящийся в памяти операнд, который должен быть выровнен по естественному адресу. Все операнды рассматриваются как целочисленные, независимо от указанного типа операнда. Операция выполняется так:

```
temp = src3;
if (temp == src1) src3 = src2;
return temp;
```

Если необходимо, могут быть реализованы и другие атомарные команды (в формате 2.8 с OP1 = 1 и увеличивающимися значениями в OP2).

Одноформатные команды с РОН в качестве входного операнда и векторным регистром в качестве выходного, или наоборот

gp2vec

Значение регистра общего назначения копируется в находящийся в векторном регистре скаляр. Длина будет равна размеру операнда. Никаких преобразований типа не выполняется.

vec2gp

Первый элемент векторного регистра копируется в регистр общего назначения. Если указан целочисленный тип, имеющий разрядность, меньшую 64, то значение знаком расширяется до 64 разрядов. Если указан тип „вещественное число одинарной точности“, то значение расширяется нулём до 64 разрядов. Никаких иных преобразований типов не выполняется.

set_len

Устанавливает длину векторного регистра равной количеству байт, указанному в регистре общего назначения. Если указанная длина больше длины, максимально допустимой для указанного типа операнда, то будет использоваться максимальная длина.

Если выходной вектор длиннее входного, то дополнительные элементы будут нулями. Если выходной вектор — короче входного, то лишние элементы будут отброшены.

get_len

Получает длину векторного регистра (в байтах). Результат сохраняется в регистре общего назначения.

set_num

То же, что и set_len, но длина умножается на размер операнда.

get_num

То же, что и get_len, но длина делится на размер операнда.

mask_length

Сделать булев вектор, для того, чтобы замаскировать первые n элементов вектора, где n = RS / (размер операнда). Выходной вектор RD будет иметь ту же длину, что и входной вектор RD. RS указывает длину части, которая разрешается маской. IM1 содержит следующие разряды опций:
разряд №0 = 0: разряд №0 будет равен 1 в первых n элементах результата, и 0 — в остальных;
разряд №0 = 1: разряд №0 будет равен 0 в первых n элементах результата, и 1— в остальных;
разряд №1 = 1: разряд №1 всех элементов результата равен 1;
разряд №2 = 1: копировать разряд №1 каждого элемента из входного вектора RD;
разряд №3 = 1: копировать разряд №1 каждого элемента из численного управляющего регистра;
разряд №4 = 1: копировать оставшиеся разряды из входного вектора RD;
разряд №5 = 1: копировать оставшиеся разряды из численного управляющего регистра.
Разряды результата, не установленные никакой из этих опций, будут нулями.

make_sequence

Сделать вектор, длина которого равна RS байтов. Количество элементов равно RS/(размер операнда). Первый элемент равен IM1, следующий элемент — IM1+1, и т.д. Поддержка вещественных чисел — необязательна.

Другие одноформатные команды, которые могут изменять длину вектора

bits2bool

Расширить смежные разряды векторного регистра до булева вектора, с одним разрядом в каждом элементе

bool2bits

Преобразовать булев вектор из n элементов в n непрерывных разрядов векторного регистра. Длина результирующего вектора будет равна степени двойки, достаточной для хранения n разрядов.

shift_expand

Длина вектора увеличивается на указанное количество байтов, посредством сдвига всех байтов вправо и добавления нулевых младших байтов. Если получившаяся длина больше максимальной длины вектора для указанного типа операнда, то старшие байты теряются.

shift_reduce

Длина вектора уменьшается на указанное количество байтов, посредством сдвига всех байтов вправо. Если получившаяся длина меньше нуля, то результатом будет вектор нулевой длины. Указываемый тип операнда — игнорируется.

compress

Элементы вектора преобразуются к элементам половинного размера. Длина результирующего вектора будет равна половине длины входного вектора. Поле OT указывает тип операнда входного вектора. Вещественные числа двойной точности преобразуются в числа одинарной точности. Целочисленные элементы преобразуются к половинному размеру отбрасыванием старших разрядов. Поддержка следующих преобразований необязательна: одинарная точность в половинную, четырёхкратная точность в двойную, 8–разрядное целое в 4–разрядное.

Если длина входного вектора отличается от длины, указанной в RS, то перед сжатием длина преобразуется к длине в RS.

compress_ss

То же, что и compress. Целые числа трактуются как знаковые, и сжимаются с насыщением. Вещественные операнды использоваться не могут. Данная команда необязательна.

compress_us

То же, что и compress. Целые числа трактуются как беззнаковые, и сжимаются с насыщением. Вещественные операнды использоваться не могут. Данная команда необязательна.

expand

Это операция, обратная к compress. Результирующий вектор имеет указанную длину, а входной вектор — половинную длину. Поле OT указывает тип операнда выходного вектора. Вещественные числа одинарной точности преобразуются в числа двойной точности. Целые числа преобразуются в числа двойного размера расширением знаком. Поддержка следующих преобразований необязательна: половинная точность в одинарную, двойная точность в четырёхкратную, 4–разрядное целое в 8–разрядное.

Если длина входного вектора отличается от RS/2, то перед расширением она преобразуется. Если получившаяся длина превышает максимальную длину вектора для указанного типа операнда, то лишние элементы теряются.

expand_us

То же, что expand. Целые числа расширяются знаком. Вещественные операнды использоваться не могут.

Одноформатные команды, которые могут перемещать данные горизонтально из одного вектора в другой

Для очень длинных векторов задержка этих команд может зависеть от расстояния перемещения (указанного в RS).

extract

Выделить один элемент вектора в скаляр, находящийся в векторном регистре. Находящийся вне диапазона индекс порождает нулевой результат. Может использоваться размер операнда, равный 16 байтам, даже если этот размер никак иначе не поддерживается.

insert

Заменить один элемент вектора, вставив скаляр в позицию, указанную индексом. Находящийся вне диапазона индекс оставит вектор без изменения. Может использоваться размер операнда, равный 16 байтам, даже если этот размер никак иначе не поддерживается.

shift_up

Сдвинуть элементы вектора влево на количество элементов, указанное в RS. Младшие RS элементов приёмника становятся нулями, а старшие RS элементов источника — теряются.

Данная команда отличается от shift_expand тем, что величина сдвига указывается как количество элементов, а не количество байтов, и тем, что длина вектора не изменяется.

shift_dn

Сдвинуть элементы вектора вправо на количество элементов, указанное в RS. Старшие RS элементов приёмника становятся нулями, а младшие RS элементов источника — теряются.

Данная команда отличается от shift_reduce тем, что величина сдвига указывается как количество элементов, а не количество байтов, и тем, что длина вектора не изменяется.

permute

Эта команда переставляет элементы вектора. Вектор делится на блоки, размером по RS байтов каждый. Размер блока должен быть степенью двойки и кратен размеру операнда. Элементы могут произвольно меняться местами в каждом блоке, но не между блоками. Каждый элемент результирующего вектора является копией элемента входного вектора, выбранного соответствующим индексом в индексном векторе. Индексы отсчитываются от начала того блока, которому принадлежат, так что индекс, равный нулю, выберет первый элемент блока входного вектора, и вставит его в соответствующую позицию результирующего вектора. Один и тот же элемент входного вектора может быть скопирован во много элементов результирующего вектора. Индекс, находящийся вне диапазона, в качестве результата даст нуль. Индексы интерпретируются как целые числа, независимо от типа операнда.

У команды permute есть две версии. Первая версия указывает индексы в векторе той же длины и того же размера элемента, что и входной вектор.

Вторая версия указывает индексы как 32–разрядную непосредственно заданную константу, с 4 разрядами на элемент. Эта константа разбивается на самое большее 8 элементов, по 4 разряда в каждом. Если блоки имеют более 8 элементов, то, для заполнения блока, последовательность из 8 элементов повторяется. В этой версии команды permute один и тот же шаблон индексов будет применён ко всем блокам.

Максимальный размер блока для команды permute зависит от реализации и задаётся специальным регистром. Причина этого ограничения на размер блока состоит в том, что сложность аппаратуры растёт квадратично с ростом размера блока. Полная перестановка возможна, если длина вектора не превышает максимального размера блока. Если RS больше максимального размера блока, то порождается ловушка.

Есть два способа комбинирования результатов многократных команд перестановки. Один метод заключается в использовании находящихся вне диапазона индексов, чтобы порождать нули для неиспользуемых элементов, а затем применять операцию поразрядного ИЛИ результатов. Другой метод состоит в использовании масок для комбинирования результатов.

Команды перестановки полезны для переупорядочивания данных, транспонирования матриц, и т.п.

Когда размер блока достаточно велик, чтобы содержать всю таблицу, команды перестановки могут также использоваться для параллельного просмотра таблицы.

Наконец, команды перестановки могут использоваться для сбора и рассеивания данных в пределах области, не большей длины вектора или размера блока.

broadcast

Копирует первый элемент входного вектора во все элементы выходного вектора. Если максимальная длина вектора более 16 байт, то поддерживается размер элемента, равный 16 байтам (128 разрядов), даже если этот размер никак иначе не поддерживается.

Иные одноформатные векторные команды

Арифметика с насыщением

Сюда относятся команды add_ss, add_us, sub_ss, sub_us, mul_ss, mul_us, shl_ss, shl_us.

Эти команды используются для арифметических операций с насыщением. Выход за верхнюю границу даст максимальное значение для заданного размера операнда. Выход за нижнюю — минимальное значение

Поддержка данных команд необязательна.

Сложение с переносом и вычитание с заёмом

Это команды add_c, sub_b. В них dest и src1 являются векторами из двух целых чисел, а src2 — вектор из целых чисел, в котором используется только первый элемент.

add_c:

```
sum = src1[0] + src2[0] + (src1[1] & 1)
dest[0] = bit 0-63 of sum
dest[1] = bit 64 of sum
```

sub_b:

```
sum = src1[0] - src2[0] - (src1[1] & 1)
dest[0] = bit 0-63 of sum
dest[1] = bit 64 of sum
```

Поддержка этих команд необязательна. Более длинные векторы — не поддерживаются. См. с. 51 по поводу иных вариантов для более длинных векторов.

Арифметические команды с проверкой переполнения

Это команды add_oc, sub_oc, subr_oc, mul_oc, div_oc.

Данные команды для арифметических вычислений используют элементы векторов, имеющие чётные номера. Каждый следующий элемент вектора, имеющий нечётный номер, используется для обнаружения переполнения. Если первый операнд–источник является скаляром, то результирующий операнд будет вектором из двух элементов.

Условия переполнения указываются следующими разрядами:

разряд №0. беззнаковое целочисленное переполнение(перенос)

разряд №1. знаковое целочисленное переполнение

разряд №2. вещественное переполнение

разряд №3. недопустимая вещественная операция

Значения распространяются так, чтобы к результат переполнения операции и обоим входным операндам применяется поразрядное ИЛИ.

Данные команды необязательны.

Расширенное деление

Это команды div_ex_s, div_ex_u.

Они необязательны, и могут поддерживаться и для скаляров, и для векторов; только для скаляров; или не поддерживаться вовсе.

byte_reverse

Данная команда обращает порядок байтов в целом числе. Она может использоваться при чтении и записи двоичных файлов данных с обратным порядком байтов.

read_spev

Значение поля RT указывает специальный регистр, который нужно прочитать. Приёмник является векторным регистром, длина которого указана в RS.

На данный момент определены следующие специальные регистры:

Таблица 4.4.5. Специальные регистры, которые можно прочесть в векторы

Номер специального регистра	**Смысл**
0	Численный управляющий регистр (NUMCONTR). Это значение рассылается во все элементы регистра–приёмника с указанными размером операнда и длиной.
1	Имя процессора. Результат является заканчивающейся нулём строкой в кодировке UTF-8, содержащей торговую марку и имя модели микропроцессора.

replace

Все элементы из src1 заменяются целочисленной или вещественной константой из src2.

При использовании без маски константа просто рассылается, чтобы сделать вектор того же размера, что и src1. При использовании с маской элементы из src1 заменяются выборочно. В зависимости от разряда №1 маски, невыбранные элементы обнуляются или остаются без изменений.

make_mask

Сделать маску из разрядов 32–разрядной целочисленной константы src2. Каждый разряд из src2 идёт в разряд №0 одного элемента результата. Оставшиеся разряды каждого элемента берутся из src1. Длина результата — та же, что и у src1. Если в векторе более 32 элементов, то двоичный шаблон из src2 повторяется..

fp_category

Входной аргумент — вещественный вектор. Выходной — булев вектор, указывающий, принадлежит ли входной аргумент какой–либо из указанных разрядами непосредственно заданного операнда категорий:

Таблица 4.4.6. Смысл разрядов в fp_category

Номер разряда	Смысл
0	Инвертировать результат.
1	Ноль.
2	Денормализованное.
3	Нормализованное.
4	Бесконечность.
5	Нечисло (NAN).
6	Знаковый разряд.
7	Копировать оставшиеся разряды из маски или численного управляющего регистра.

Команды для таблиц истинности

Это команды truth_tab2, truth_tab3, truth_tab4.

Эти команды могут вычислять заданные таблицами истинности произвольные булевы функции двух, трёх, или четырёх входных векторных булевых аргументов. Результат в разряде №0 каждого элемента вектора является произвольной булевой функцией разряда №0 соответствующих элементов каждого из входных операндов. Разряд №0 результата равен разряду из таблицы истинности, выбранному скомбинированными разрядами аргументов. Остальные разряды вектора–результата копируются из регистра маски, если таковой есть, или, в противном случае, из первого входного операнда.

У truth_tab2 аргументы — в RD и RS, результат — в RD, а 4–разрядная таблица истинности — в IM1.

У truth_tab3 аргументы — в RS, RT и RU, результат — в RD, а 8–разрядная таблица истинности — в IM2.

У truth_tab4 аргументы — в RD, RS, RT и RU, результат — в RD, а 16–разрядная таблица истинности — в IM2.

У truth_tab4 размер операнда должен быть не менее 16 разрядов. Команды truth_tab3 и truth_tab4 — необязательны.

Команды truth_tab3 и truth_tab4 в качестве дополнительного операнда могут использовать маску, в соответствии с обычной функцией последней.

Эти команды могут использоваться в качестве универсальных команд для манипулирования и комбинирования булевых векторов и масок.

Аппаратная реализация может использовать существующие модули сдвига, сдвигая таблицу истинности на количество разрядов, указанное скомбинированными разрядами входных операндов.

add_add_add

Складывает четыре аргумента. Последний операнд может быть регистровым операндом или 16–разрядным знаковым непосредственно заданным операндом. Знаки операндов могут инвертироваться, что указывается разрядами 0–3 поля OP3:

разряд №0: изменить знак src1
разряд №1: изменить знак src2
разряд №2: изменить знак src3
разряд №3: изменить знак src4
По поводу деталей см. с. 40.
Данная команда — необязательна.

4.5. Распространённые операции, для которых нет специальных команд

В данном разделе обсуждаются некоторые распространённые операции, которые не реализованы как отдельные команды, и то, как закодировать эти операции программно.

Смена знака

Для целочисленных операндов осуществляйте её обратным вычитанием из нуля. Для вещественных операндов используйте команду toggle_b над знаковым разрядом.

Команда abs для вещественных чисел

Для получения абсолютной величины вещественного числа используйте clear_b для очистки знакового разряда.

Not

Чтобы инвертировать все разряды целого числа, выполните ИСКЛЮЧАЮЩЕЕ ИЛИ с -1. Для инвертирования булева значения выполните ИСКЛЮЧАЮЩЕЕ ИЛИ с 1.

Вращение через перенос

Вращение через перенос используется редко, а распространённые реализации могут быть очень неэффективны. Вращение влево через перенос можно заменить командой add_c с одним и тем же регистром в обоих операндах–источниках.

Горизонтальное векторное сложение

Команда сложения всех элементов вектора была бы полезна, но такая команда не поддерживается, ибо такая она была бы сложной командой с переменной задержкой, зависящей от длины вектора.

Сумму всех элементов вектора можно вычислить, повторно складывая младшую и старшую половины вектора. Данный метод иллюстрируется следующим примером, находящим горизонтальную сумму вектора, состоящего из 32–разрядных целых чисел. Синтаксис языка ассемблера описан на с. **??**.

```
v0 = my_vector   // мы хотим вычислить горизонтальную сумму этого вектора
r0 = get_len(v0) // длина вектора (в байтах)
r0 = roundu2.64(r0)  // округлить к ближайшей большей степени двойки
v0 = set_len(v0, r0) // настроить длину вектора
// Цикл вычисления горизонтальной суммы для v0
LOOP: // метка
   // Вектор половинной длины.
   r1 = shift_rightu.64(r0, 1)
   // Получить старшую половину вектора.
   v1 = shift_reduce(v0, r1)
   // Сложить старшую и младшую половины
   v0 = add.32(v1, v0)  // результат имеет ту же длину, что и первый операнд
   // Половинная длина для следующей операции.
   r0 = r1
   // повторять, пока вектор содержит более одного элемента
```

```
   compare_unsign_jmpabove(r1, 4, LOOP)
// Теперь сумма является скаляром, находящимся в v0.
```

Тот же самый метод может использоваться и для других горизонтальных операций. Может вызвать проблемы то, что команда set_len вставляет равные нулю элементы, если длина вектора не равна степени двойки. Особенно беспокоиться нужно, если операция не допускает равных нулю дополнительных элементов, например, если операция использует умножение или нахождение наименьшего элемента. Возможное решение состоит в маскировании неиспользуемых элементов на первой итерации. Следующий пример находит наименьший элемент в векторе из вещественных чисел:

```
v0 = my_vector              // найти наименьший элемент в этом векторе
r0 = get_len(v0)            // длина вектора (в байтах)
r1 = roundu2.64(r0)         // округлить к ближайшей большей степени двойки
r1 = shift_rightu.64(r1, 1) // половина длины
v1 = shift_reduce(v0, r1)   // старшая часть вектора
r2 = sub.64(r0, r1)         // длина вектора v1
// использовать маску, поскольку два операнда могут иметь различные длины
v0 = set_len(v0, r1)        // уменьшить длину вектора v0
v2 = v0                     // произвольный вектор с длиной r1
v2 = mask_length.32(v2, r2, 0x22)  // создать маску для v1
v0 = min.f(v0, v1, mask=v2) // получить минимум, замаскировав неиспользуемые элементы
cmp_unsign_jmpbeloweq(r1, 4, ENDOFLOOP) // проверить, не закончилось ли уже
// Цикл для вычисления горизонтального минимума из элементов v0.
LOOP: // метка
   // Вектор половинной длины.
   r2 = shift_rightu.64(r1, 1)
   // Получить старшую половину вектора.
   v1 = shift_reduce(v0, r2)
   // Вычислить минимум из старшей и младшей половины.
   v0 = min.f(v1, v0)  // результат имеет ту же длину, что и первый операнд
   // Половинная длина для следующей операции.
   r1 = r2
   // повторять, пока вектор содержит более одного элемента
   compare_unsign_jmpabove(r2, 4, LOOP)
ENDOFLOOP:
// Теперь минимум является скаляром, находящимся в v0.
```

Арифметика высокой точности

Библиотеки функций для арифметики высокой точности для сложения чисел с очень большим количеством разрядов обычно используют длинную последовательность команд сложения с переносом. Более эффективным методом для вычислений с большими числами является использование векторного сложения и метода ускоренного переноса. Следующий алгоритм вычисляет A + B, где A и B — большие целые числа, представленные в виде двух векторов, по n·64 разрядов в каждом, где $n < 64$.

```
v0 = A                  // первый вектор, n*64 bits
v1 = B                  // второй вектор, n*64 bits
v2 = carry_in           // скаляр в векторном регистре
v0 = add.64(v0, v1)     // сумма без промежуточных переносов
v3 = compare.64(v0,v1,8) // породить перенос = (SUM < B) (сравнение --- беззнаковое)
v4 = compare.64(v0,-1,0xA) // распространить перенос = (SUM == -1)
v3 = bool2bits(v3)      // породить перенос, сжатый до битового поля
v4 = bool2bits(v4)      // породить перенос, сжатый до битового поля
// CA = CP ^ (CP + (CG<<1) + CIN) // распространить дополнительный перенос
v3 = shift_left.64(v3,1) // сдвинуть влево порождённые переносы
v2 = add.64(v2,v4)
v2 = add.64(v2,v3)
v2 = xor.64(v2,v4)
```

```
v1 = bits2bool(v2)       // расширить дополнительный перенос на вектор
v0 = sum.64(v0,v1)       // прибавить поправку к сумме
r0 = get_num(v0)         // n = количество элементов в векторах
v3 = gp2vec.64(r0)       // копировать в векторный регистр
v2 = shift_rightu.64(v2,v3) // итоговый перенос
// v0 = сумма, v2 = итоговый перенос
```

Если числа A и B длиннее, чем максимальная длина вектора, то алгоритм повторяется. Если длина вектора — больше, чем $64 \cdot 8$ байт, то вычисление дополнительного переноса вовлечёт более 64 разрядов, что снова потребует алгоритма работы с большими числами.

4.6. Неиспользуемые команды

Неиспользуемые команды и опкоды (коды операций) можно разбить на три типа:

1) Опкод зарезервирован для последующего использования. Попытка его выполнения вызовет ловушку (синхронное прерывание), которая может использоваться для порождения сообщения об ошибке или для эмуляции неподдерживаемой команды.

2) Опкод гарантированно генерирует ловушку, не только в текущей версии, но и всех последующих. Он может использоваться как заполнитель неиспользуемых частей памяти или как указание на ошибку, после которой невозможно восстановиться. Его можно также использовать для эмуляции специфичных для пользователя команд.

3) Ошибка игнорируется, и ловушки не возникает. Это может использоваться для последующих расширений, улучшающих производительность или функциональность, но которые можно безопасно игнорировать, когда они не поддерживаются.

Реализованы все три типа, и тип 1 — наиболее распространён.

Команды nop с ненулевыми значениями в неиспользуемых полях принадлежат типу 3. Эти команды игнорируются.

Команды prefetch и fence без находящегося в памяти операнда, с ненулевыми значениями в неиспользуемых полях, или с неопределёнными значениями в OP3 принадлежат типу 3. Эти команды игнорируются.

Неиспользуемые разряды в масках и численном управляющем регистре имеют тип 3. Эти разряды игнорируются.

Команды trap (в том числе условные) с ненулевыми значениями в неиспользуемых полях или неиспользуемыми значениями в любом поле имеют тип 2. Эти команды гарантированно порождают ловушку. Специальная версия команды trap предназначена для использования в качестве заполнителя неиспользуемых или недоступных частей памяти кода.

Команда undef имеет тип 2. Она гарантированно порождает ловушку на всех системах, и может использоваться для целей тестирования и эмуляции.

Команды userdef__ имеют тип 1. Эти команды зарезервированы для определения пользователем и для целей конкретных приложений.

Для команд с ошибочной кодировкой следует предпочесть их поведение, как в типе 1. Сюда включаются коды команд с ненулевыми значениями в неиспользуемых полях, неподдерживаемыми типами операндов, или любыми другими двоичными шаблонами с неопределённым смыслом в каком–либо поле. Как вариант, в этих случаях может допускаться поведение типа 3. Если это так, то команде следует вести себя так, как будто она закодирована корректно.

Все прочие опкоды неявно определены как имеющие тип 1, и могут использоваться для последующих расширений.

Малым системам, без операционных систем и без поддержки ловушек, следует определить иное поведение.

Глава 5. Другие детали реализации

5.1. Порядок байтов

Организация памяти использует прямой порядок байтов (от младшего к старшему). Для чтения и записи файлов данных с обратным порядком байтов предоставляются команды смены порядка байтов.

Обоснование

Если бы использовалась организация с обратным порядком байтов, то способ хранения вектора в памяти зависел бы от размер элемента. Например, предположим, что у нас есть 128–разрядный векторный регистр, содержащий четыре 32–разрядных целых числа, обозначенные A, B, C, D. При использовании прямого порядка байтов, они хранились бы в памяти в следующем порядке:
где A0 — самый младший байт числа A, а D3 — самый старший байт числа D. При использовании же обратного порядка мы бы имели

A3, A2, A1, A0, B3, B2, B1, B0, C3, C2, C1, C0, D3, D2, D1, D0.

Этот порядок изменился бы, если бы тот же самый векторный регистр был организован, например, как восемь целых чисел, по 16 разрядов в каждом, или как два целых числа, по 64 разряда в каждом. Иными словами, нам бы потребовались разные команды чтения и записи для разных способов организации вектора.

Прямой порядок байтов более распространён по большому количеству причин, многократно обсуждавшихся в иных местах.

5.2. Реализация стека вызовов

Есть разные методы сохранения адресов возврата при вызовах функций: регистр связи, отдельный стек вызовов, или унифицированный стек, и для адресов возврата, и для локальных данных. Здесь мы обсудим достоинства и недостатки каждого из этих методов.

Регистр связи

Некоторые системы для хранения адреса возврата используют регистр связи. Преимущество регистра связи состоит в том, что листовые функции могут вызываться без сохранения чего–либо в стеке, что экономит пропускную способность кэша в программах с большим количеством вызовов листовых функций. Недостаток состоит в том, что каждой нелистовой функции нужно перед вызовом другой функции сохранять регистр связи в стеке, и восстанавливать его перед возвратом.

Если бы мы приняли решение использовать регистр связи, то им должен был бы быть специальный регистр, а не один из регистров общего назначения. Регистру связи не нужно поддерживать все вещи, которые можно делать с регистром общего назначения. Если регистр связи включается как один из регистров общего назначения, то у программиста возникнет соблазн сохранить регистр связи в другом регистре, а не в стеке, и затем, в конце функции, перейти по адресу, хранящемуся в том другом регистре. Это, конечно, работать будет, но помешает способам, которыми предсказываются возвраты. Модуль предсказания ветвления использует специальный механизм для предсказания возвратов, отличающийся от механизма, используемого для предсказания иных переходов и ветвлений. Данный механизм, называемый буфером стека возврата, представляет собой небольшой кольцевой кэш, запоминающий адреса последних вызовов. Если функция выполняет возврат переходом по адресу, содержащемуся в регистре, отличном от регистра связи, то она будет использовать ошибочный механизм предсказания, что вызовет большие задержки из–за неверного предсказания последующей серии возвратов. Буфер стека возврата также будет перепутан, если регистр связи используется для косвенного перехода или в иных целях.

Единственные команды, необходимые для регистра связи, помимо команд вызова и возврата, — команды push и pop. Мы можем уменьшить количество команд в нелистовых функциях, сделав комбинированную команду для “втолкнуть регистр связи, а затем вызвать функцию”, которая может использоваться для первого

вызова функции в нелистовой функции, и другую команду, для "вытолкнуть регистр связи, а затем выполнить возврат", чтобы заканчивать нелистовую функцию. Однако это нарушает принцип, согласно которому мы хотим избежать сложных команд, чтобы упростить дизайн конвейера.

Единственный выигрыш в производительности, который мы получаем от использования регистра связи, состоит в экономии пропускной способности кэша из–за отсутствия сохранения адреса возврата для вызовов листовых функций. Выигрыш этот не повлияет на производительность в приложениях, в которых пропускная способность кэша — узкое место. Производительность команды возврата не оказывает влияния на пропускную способность кэша, поскольку может положиться на предсказание в буфере стека возврата.

Недостаток использования регистра связи заключается в том, что компилятор должен по–разному рассматривать листовые и нелистовые функции, и в том, что нелистовым функциям нужны дополнительные команды для сохранения и восстановления регистра связи из стека.

Следовательно, мы не будем использовать в архитектуре ForwardCom регистр связи.

Отдельный стек вызовов

Мы можем иметь два стека: стек вызовов для адресов возврата, и стек данных, для локальных данных каждой функции. Программы без рекурсивных функций будут, как правило, иметь весьма ограниченную глубину стека вызовов, так что весь стек вызовов, или, по крайней мере, его „горячая" часть, может храниться на кристалле, что улучшит производительность, ибо для операций вызова и возврата не нужно будет ни операций с памятью, ни операций с кэшем, — по крайней мере в критичном наиболее глубоко вложенном цикле программы. Также упростится предсказание адресов возврата, так как находящийся на кристалле кольцевой стек и буфер стека возврата будут одной и той же структурой.

Стек вызовов может быть реализован как кольцевой стек регистров на кристалле. Стек вызовов сбрасывается в память при переполнении. Команда возврата после такого события переполнения будет использовать значение на кристалле, а не значение в памяти, до тех пор, пока значение на кристалле не будет перезаписано новыми вызовами. Следовательно, событие сброса вряд ли должно возникать более одного раза в самой внутренней части программы.

Указатель для стека вызовов не должен быть регистром общего назначения, поскольку программисту редко нужно обращаться непосредственно к нему. Прямая манипуляция стеком вызовов нужна только при событии раскрутки стека (после исключения или long jump), либо при переключении задач.

У функции нет лёгкого способа обращения к адресу возврата, с которого она была вызвана. Сведения о вызывающем могут быть переданы явно, как параметр функции, в тех редких случаях, когда это необходимо. У сокрытия адреса возврата внутри процессора имеется достоинство с точки зрения безопасности: это мешает переписывать адрес возврата в случае программных ошибок или злонамеренных атак переполнением буфера.

Недостаток наличия отдельного стека вызовов состоит в том, что оно делает управление памятью более сложным, так как имеется два стека, которые потенциально могут переполниться. Для программ без рекурсивных функций размер стека вызова можно точно предсказать, используя метод, описанный на с. 78.

Отдельный стек вызовов может быть реализован в архитектуре ForwardCom. Размер накристалльного буфера стека и другие детали будут зависеть от реализации.

Унифицированный стек вызова для адресов возврата и локальных данных

Многие имеющиеся системы используют один и тот же стек и для адресов возврата, и для локальных данных. Данный метод может использоваться с архитектурой ForwardCom, поскольку прост в реализации.

Заключение для ForwardCom

Система ForwardCom может использовать отдельный стек вызовов или унифицированный стек, но не регистр связи. Аппаратная реализация команд вызова и возврата зависит от того, имеется ли один или два стека. Система с двумя стеками будет использоваться для больших процессоров, которым важны производительность и безопасность, тогда как система с унифицированным стеком может использоваться в малых процессорах, где предпочтительнее простота. Микропроцессор ForwardCom не должен поддерживать обе системы, но должно программное обеспечение. Определённые на с. 73 соглашения вызова сделают программное обеспечение совместимым как с процессорами, имеющими один стек, так и с процессорами, имеющими два. Хвостовые вызовы могут быть эффективно реализованы одной командой перехода, независимо от типа стека.

5.3. Вещественные ошибки и исключения

Исключения для вещественных ошибок по умолчанию — запрещены, но их можно разрешить с помощью разрядов 26–29 численного регистра управления или регистра маски. Разрешённые исключения отлавливаются как ловушки (синхронные прерывания).

Проблема в том, что исключение, вызванное одним элементом вектора, вызовет прерывание обработки всего вектора. В случае ловушек, вызванных одним элементом вектора, поведение программы, использующей вещественные векторы, будет зависеть от длины вектора. Мы можем положиться на порождение и распространение значений NAN и INF, вместо ловушек, если мы хотим устойчивых результатов на разных процессорах с разными длинами векторов.

Значения NAN будут распространяться по последовательности вещественных вычислений. Значение NAN может содержать двоичный шаблон диагностической информации, называемый полезной нагрузкой, и этот двоичный шаблон распространяется к результату. Проблема возникает, когда комбинируются два значения NAN, например, NAN1 + NAN2. Стандарт IEEE standard (754-2008) указывает лишь, что к результату распространяется один из двух операндов со значением NAN, что нарушает фундаментальный принцип коммутативности сложения. Результат может быть неустойчивым, когда компилятор меняет местами два операнда. Другая проблема со стандартом IEEE состоит в том, что значения NAN, согласно этому стандарту, не распространяются через операции max и min.

Здесь предлагается отклониться от этого неудачного стандарта, и выдавать, при комбинировании операндов NAN, комбинацию, с помощью поразрядного ИЛИ, полезных нагрузок значений NAN, что сделает распространение значений NAN более полезным и устойчивым. Разные разряды в NAN могут использоваться для указания разных сообщений об ошибках. Если в последовательности вычислений возникло много разных ошибочных условий, то все эти условия можно отследить в окончательном результате. Данное более хорошее распространение значений NAN разрешается установкой разряда 22 в численном регистре управления или в регистре маски.

Реализация будет использовать лишь один разряд в полезной нагрузке NAN для каждого условия ошибки. Тихое NAN имеет установленный разряд мантиссы с номером *количество_разрядов* -1, тогда как остальные разряды доступны для любых полезных сведений. Если более хорошее распространение значений NAN разрешено разрядом 22 численного управляющего регистра или регистром маски, то процессор ForwardCom помещает диагностическую информацию в полезную нагрузку. Разряд мантиссы с номером *количество_разрядов* -2 указывает на неверную арифметическую операцию, такую, как 0/0, $0 \cdot \infty$, $\infty - \infty$, и т.п. Разряд с номером *количество_разрядов* -3 указывает на квадратный корень из отрицательного числа или иные комплексные результаты. Оставшиеся разряды полезной нагрузки доступны для других целей, таких, как библиотеки функций.

Другие методы порождения сообщений об ошибках в библиотеках функций обсуждаются на с. 70.

5.4. Обнаружение целочисленного переполнения

Нет общепринятого стандартного метода обнаружения переполнения в целочисленных вычислениях. Обнаружение переполнения в операциях со знаковыми целыми числами в некоторых языках, вроде C++, — реально ночной кошмар (см., например, stackoverflow.com/questions/199333/how-to-detect-integer-overflow-in-c-c).

Было бы приятно иметь надёжный способ обнаружения целочисленного переполнения, и, вероятно, распространения его через последовательность вычислений, аналогичную распространению NAN для вещественных вычислений, так, чтобы ошибки можно было проверять в конце последовательности вычислений, а не после каждой операции. Компиляторы могли бы поддерживать этот метод, предлагая обнаружение переполнения блоком try/catch. Если аппаратура предложит разумный метод, то более вероятно, что компиляторы поддержат обнаружение целочисленного переполнения.

Предлагались следующие методы:

1) Использовать несколько свободных разрядов регистров маски для обнаружения и распространения переполнения и других ошибок. У данного метода есть большое количество проблем, препятствующих неупорядоченному выполнению. Регистр маски будет использоваться не только как аргумент каждой команды, но и как результат. Каждая команда будет затем иметь два результата, а не один, что намного усложнит планировщик неупорядоченного выполнения, и вызовет нежелательные зависимости, когда один и тот же регистр маски используется для многих команд, которые иначе были бы независимы.

2) Использовать элементы векторного регистра с нечётными номерами для нормальных вычислений над целыми числами, а следующие элементы с нечётными номерами — для сведений о переполнении. Сведения о переполнении распространяются вместе с вычисленными значениями. Данный метод будет эффективен для скалярных целочисленных вычислений, но расточителен для векторов, поскольку половина элементов вектора используется только для этой цели.

3) Использовать один элемент вектора для разрядов переполнения всех других элементов. Этот метод может быть заманчивым, поскольку тратит не столь много места, сколь предыдущий метод, но он уступает по производительности, из–за задержки переноса разрядов переполнения в дальнюю часть длинного вектора.

4) Добавить в векторные регистры дополнительные разряды для сведений о переполнении. У всех векторных регистров будет иметься один дополнительный разряд переполнения на каждые 32 разряда полезных данных. Эти разряды переполнения сохраняются, когда векторный регистр сохраняется и восстанавливается командами save_cp и restore_cp, но они теряются, когда вектор сохраняется как нормальные данные. Поведение разрядов переполнения управляется следующими разрядами численного управляющего регистра или регистра маски:
разряд №2: обнаруживать беззнаковое целочисленное переполнение;
разряд №3: обнаруживать знаковое целочисленное переполнение;
разряд №4: обнаруживать вещественное переполнение (экспериментально);
разряд №5: обнаруживать недопустимые вещественные операции (экспериментально);
разряд №6: распространять сведения о переполнении из входных операндов, применяя поразрядное ИЛИ результата текущей команды с разрядами переполнения всех входных векторных регистровых операндов; нужно предоставить дополнительную команду для выделения разрядов переполнения из векторного регистра.

5) В случае целочисленного переполнения порождать ловушку. Использовать регистр маски или численный управляющий регистр, как в методе 4. Разряд №7 разрешает ловушку при условиях, указанных разрядом №2 (беззнаковое целочисленное переполнение) или №3 (знаковое целочисленное переполнение). Данный метод требует несколько большего кода, но подвержен проблеме, заключающейся в том, что, как объяснялось в предыдущем разделе для вещественных ошибок, поведение векторного кода при возникновении ловушки зависит от длины вектора.

Метод 2 здесь экспериментально поддерживается необязательными командами add_oc и т.п., описанными на с. 48.

Поддержка метода 4 может быть рассмотрена, поскольку она была бы более эффективной и полезной. Цена реализации метода 4 состоит в том, что в векторных регистрах нам нужно на 3% больше разрядов; команды save_cp и restore_cp будут более сложными; а компилятор должен проверять переполнение перед сохранением векторов в памяти обычным способом.

Метод 5 следует поддержать. Он полезен для целочисленного кода в регистрах общего назначения и и для проверки, что переполнение не возникает в векторных регистрах.

Этими методами не следует обнаруживать переполнение в команд арифметики с насыщением и в командах сдвига.

5.5. Многопоточность

Дизайн ForwardCom делает возможным реализацию очень больших векторных регистров, для обработки больших наборов данных. Однако есть практические ограничения на то, насколько сильно мы можем повысить производительность, используя бóльшие векторы. Во–первых, фактические структуры данных и алгоритмы часто ограничивают длину вектора, которая может использоваться. И, во–вторых, бóльшие векторы означают бóльшие физические расстояния на полупроводниковом кристалле, и бóльшие транспортные задержки.

Дополнительный параллелизм можно получить, выполняя несколько потоков на каждом ядре ЦП. Дизайну следует допускать несколько ЦП на кристалле, или несколько ядер ЦП на одном и том же физическом кристалле.

Коммуникация между потоками и синхронизация потоков могут быть проблемами для производительности. Системе следует иметь эффективные средства для этих целей, включая спекулятивную синхронизацию.

Вероятно, не стоит допускать много потоков, одновременно разделяющих одно и то же ядро ЦП и кэш первого уровня (что Intel называет гиперпоточностью), ибо это могло бы разрешить потоку с низким приоритетом забрать ресурсы у потока с высоким приоритетом, и для операционной системы сложно определить, какие потоки могли бы конкурировать за одни и те же ресурсы выполнения, если они выполняются на одном и том же ядре ЦП.

5.6. Свойства безопасности

Безопасность включена в базовый дизайн и аппаратуры, и программного обеспечения. Она включает в себя следующие возможности.

- Гибкий и эффективный механизм защиты памяти.
- Необязательное разделение стека вызовов и стека данных, так что адреса возврата не могут быть скомпрометированы переполнением буфера.
- Каждый поток имеет своё собственное защищённое пространство памяти, за исключением совместимости с устаревшим программным обеспечением, требующим разделяемого пространства памяти для всех потоков приложения.
- Драйвера устройств и системные функции имеют тщательно контролируемые права доступа. Эти функции не имеют общего доступа к памяти приложения, имея доступ лишь к конкретному блоку памяти, который приложение может разделять с системной функцией при её вызове. Драйвер устройства имеет доступ лишь к конкретному диапазону портов ввода–вывода и системных регистров, указанному в заголовке исполняемого файла и контролируемому ядром системы.
- Обрушению драйвера устройства следует порождать не „синий экран смерти“, а сообщение об ошибке с закрытием приложения, его вызвавшего, и освобождение всех своих ресурсов.
- Прикладные программы имеют доступ только к конкретным ресурсам, указанным в заголовке исполняемого файла, и контролируемых системой.
- При использовании режима адресации со встроенной проверкой границ массива или условной ловушки проверка границ массивов — проста и эффективна.
- Различные необязательные методы проверки целочисленного переполнения.
- Нет „неопределённого“ поведения. Всегда имеется ограниченный набор допустимых реакций на ошибочное условие.

Как улучшить безопасность приложений и систем

Ниже перечислено несколько методов улучшения безопасности. Эти методы могут быть полезны в приложениях и операционных системах для ForwardCom, когда важна безопасность.

Защита от переполнения буфера

Входные буферы должны быть защищены от переполнения. Если программная защита недостаточна, то вы можете для входного буфера выделить изолированный блок памяти. См. с. 64.

Защита массивов

Следует проверять границы массивов.

Защита от целочисленного переполнения

Используйте один из методов обнаружения целочисленного переполнения, упомянутых на с. 55.

Защита памяти потока

Каждому потоку приложения следует иметь своё собственное защищённое пространство памяти. См. с. 64.

Защита указателей кода

Указатели на функции и другие указатели на код — уязвимы для атак перехвата потока управления. Сюда входят атаки на следующее.

Адреса возврата. Адреса возврата, находящиеся в стеке, особенно уязвимы к атакам переполнения буфера. Используйте двухстековый дизайн для изоляции стека возврата от других данных.

Таблицы переходов. Многопутёвые ветвления switch/case часто реализуются как таблицы адресов перехода. Для этого следует использовать команду табличного перехода, с таблицей, помещённой в секцию CONST, с доступом только для чтения. См. с. 23.

Таблицы виртуальны функций. Языки программирования с объектным полиморфизмом, такие, как C++, используют таблицы указателей на виртуальные функции. Для них следует использовать команду табличного вызова, с таблицей, помещённой в секцию CONST, с доступом только для чтения. См. с. 23.

Таблицы компоновки процедур. Таблицы компоновки процедур, таблицы импорта, и символьное взаиморасположение в ForwardCom не используются. См. с. 76.

Указатели на функции обратного вызова. Если функция принимает в качестве параметра функцию обратного вызова, то вместо сохранения в памяти сохраните его в регистре.

Конечные автоматы. Если конечный автомат или аналогичный алгоритм реализуется с помощью указателей на функции, то поместите эти указатели на функции в константный массив, используйте в качестве индекса этого массива переменную, в которой хранится состояние, и проверяйте индекс на переполнение. Компилятору следует иметь поддержку определения массива относительных указателей на функции в разделе CONST, и обращения к ним через команду табличного вызова.

Другие указатели на функции. Большинство случаев использования указателей на функции охватывается описанными выше методами. Других случаев в высокобезопасных приложениях следует избегать, либо указатели следует поместить в защищённые области памяти или области с непредсказываемыми адресами (см. `http://dslab.epfl.ch/proj/cpi/`).

Управление правами доступа прикладных программ

В заголовок исполняемого файла прикладной программы следует включать сведения о том, какого рода операции приложению разрешены. Сюда могут включаться разрешения на различную сетевую активность, обращение к конкретным чувствительным файлам, разрешение на запись исполняемых файлов и скриптов, разрешение на установку драйверов, разрешение на порождение других процессов, разрешение на связь с другими процессами, и т.д. Пользователю следует иметь простой способ проверки наличия этих прав. Мы можем реализовать систему для управления правами доступа также и скриптов. Скрипты веб–страниц следует запускать в песочнице.

Управление правами доступа драйверов устройств

Многие операционные системы предоставляют драйверам устройств очень большие права. Вместо наличия бюрократической централизованной системы согласования драйверов устройств, нам следует иметь более тщательное управление правами доступа каждого драйвера устройства. Команда системного вызова в ForwardCom даёт драйверу устройства доступ только к ограниченной области прикладной памяти (см. с. 23). В заголовке исполняемого файла драйвера устройства следует иметь информацию о том, к каким портам и системным регистрам драйвер устройства имеет доступ. Пользователю следует иметь простой способ проверки наличия этих прав.

Стандартизованная процедура установки

Защите от вредоносного программного обеспечения следует быть составной частью операционной системы, а не являться сторонним дополнением. Операционной системе следует предоставлять стандартизованный способ установки и удаления приложений. Системе следует отвергать запуск каких–либо программ, скриптов, или драйверов, которые не установлены с помощью этой процедуры. Это сделает для пользователя возможным просмотр прав, требуемых для всех устанавливаемых программ, и удаление любого вредоносного программного обеспечения, либо другого нежелательного программного обеспечения, с помощью обычной процедуры удаления.

Глава 6. Программируемые специфичные для приложения команды

Вместо реализации огромного количества специальных команд для конкретных приложений, мы можем предоставить средства порождения определяемых пользователем команд, которые могут быть закодированы на языке описания аппаратуры, например, VHDL или Verilog.

Микропроцессор может иметь необязательную ПЛИС или подобную программируемую аппаратуру. Эта структура может использоваться для создания специфичных для приложений команд или функций, например, кодирования, шифрования, сжатия данных, обработки сигналов, обработки текстов, и т.п.

Если у процессора имеется несколько ядер, то каждое ядро может иметь свою собственную ПЛИС. Код определения аппаратуры для каждого ядра сохраняется в своём собственном кэше. Операционной системе следует предотвращать, насколько возможно долго, использование одного и того же ядра разными задачами, требующими разного аппаратного кода. Могут быть черты, позволяющие приложению монополизировать ПЛИС или её часть.

Если невозможно избежать использование одной и той же ПЛИС на одном и том же ядре несколькими приложениями, то код, также как содержимое всех ячеек памяти ПЛИС, должен сохраняться при переключении задач. Сохранение может быть реализовано как ленивое, т.е. содержимое переключается только тогда, когда второй задаче нужна структура ПЛИС, содержащая код первой задачи.

Должны быть команды для обращения к определённым пользователем функциям, включая средства ввода и вывода, и для адаптации задержки определённых пользователем функций.

Глава 7. Микроархитектура и дизайн конвейера

Набор команд ForwardCom предназначен для облегчения стабильного и эффективного дизайна конвейера суперскалярного микропроцессора. Команды могут иметь один операнд–приёмник, вплоть до трёх или четырёх операндов–источников, регистр маски, и регистр, указывающий длину вектора. Последний операнд–источник может быть регистром, находящимся в памяти операндом, или непосредственно заданной константой. Все другие операнды — регистры, за исключением команд записи в память. Суммарное количество входных регистров команды, включая операнды–источники, маску, базовый указатель, индекс, и указание длины вектора, не может быть более пяти.

Никакая команда не может иметь более одного находящегося в памяти операнда. Никакая команда не может иметь и операнд–источник, находящийся в памяти, и непосредственно заданный операнд, хотя это может быть разрешено в последующих расширениях. Любые дополнительные поля непосредственно заданных операндов могут использоваться для разрядов опций.

Высокопроизводительный конвейер может быть спроектирован как суперскалярный, со следующими стадиями.

- Выборка. Выборка блоков кода из кэша команд, по одной строке кэша за раз, либо так, как определено механизмом предсказания ветвлений.
- Декодирование длин команд. Определение длины каждой команды и определение малых команд. Распределение первых P команд в свою полосу конвейера, где P — количество параллельных полос, реализованных в конвейере. Избыточные команды могут быть поставлены в очередь до следующего такта.
- Декодирование команды. Определение и классификация всех операндов, опкодов, и разрядов опций. Определение входных и выходных зависимостей.
- Размещение и переименование регистров.
- Очередь команд.
- Помещение команд в станцию восстановления. Планирование для вычислителя адресов.
- Вычисление адреса и длины находящегося в памяти операнда. Проверка прав доступа.
- Чтение операнда из памяти. Планирование для исполнительных модулей.
- Исполнительные модули.
- Восстановление или ветвление.

Ненужно разбивать команды на микрооперации, если чтение находящихся в памяти операндов выполняется на отдельной стадии конвейера, и командам разрешается оставаться в станции восстановления до тех пор, пока не будет прочитан находящийся в памяти операнд.

Каждой стадии конвейера в идеале следует требовать лишь одного такта. Командам, ожидающим операндов, следует оставаться в станции восстановления. Большинство команд в исполнительном модуле будет использовать лишь один такт. Умножению и вещественному сложению нужен конвейеризованный исполнительный модуль с несколькими стадиями. Деление и квадратный корень могут использовать отдельный конечный автомат.

Команды перехода, ветвления, вызова, и возврата также подпадают под этот дизайн конвейера.

Станция восстановления должна рассматривать все входные и выходные зависимости каждой команды. Каждая команда может иметь вплоть до пяти входных зависимостей и одну выходную.

Может быть много исполнительных модулей, так что возможен запуск нескольких команд в одном и том же такте, если их операнды независимы.

Эффективная неупорядоченная обработка требует переименования регистров общего назначения и векторных регистров, но ненужна для специальных регистров.

Сложных команд и микрокода в большинстве случаев следует избегать. У нас нет команд сохранения и восстановления всех регистров при переключении задач. Вместо этого необходимые команды сохранения и восстановления регистров реализованы как малые команды, чтобы уменьшить размер последовательности команд, сохраняющей все регистры.

Следующие команды умеренно сложны: call, return, div, rem, sqrt, cmp_swap, save_cp, restore_cp. Эти команды можно реализовать специальными конечными автоматами. То же применимо к ловушкам, прерываниям, и системным вызовам.

У некоторых современных ЦП есть „стековый модуль“, чтобы предсказывать значение указателя стека для команд push, pop, или call, когда предшествующие стековые операции запаздывают из–за операндов, которые ещё недоступны. Такая система ненужна, если у нас имеется двухстековый дизайн (см. с. 54). Даже при одностековом дизайне, в стековом модуле нужды мало, так как операции push и pop будут редки в критичной части кода, если придерживаться соглашений вызова, предлагаемых в настоящем документе (с. 73).

Предсказание ветвления важно для производительности. Мы можем реализовать четыре различных алгоритма предсказания ветвлений: один для обыкновенных ветвлений, один — для циклов, один — для косвенных переходов, и один — для возвратов из функций. У длинной формы команд ветвления имеется разряд опции для указания поведения цикла. В короткой форме команд ветвления места для такого разряда нет. Первоначальная догадка может состоять в предположении циклического поведения, если ветвление выполняет переход назад, и обычного ветвления, если ветвление выполняет переход вперёд. Если необходимо, это предположение может быть позднее скорректировано механикой предсказания ветвлений.

Код, следующий за ветвлением, спекулятивно выполняется, пока не определяется, было ли правильным предсказание. Мы можем реализовать возможности для одновременного спекулятивного выполнения обеих ветвей.

Дизайн ForwardCom позволяет большие микропроцессоры с очень длинными векторными регистрами, что требует специальных обсуждений дизайна. Макеты кристаллов векторных процессоров обычно делятся на „полосы данных“, так что вертикальная передача данных от элемента вектора в соответствующий (т.е. в той же полосе) элемент другого вектора быстрее горизонтальной передачи данных из одного элемента вектора в другую (т.е в другой полосе) позицию того же вектора. Это означает, что команды, передающие данные горизонтально вдоль вектора, такие, как команды broadcast и permute, могут иметь более длинные задержки, нежели другие векторные команды. Планировщику нужно знать задержку команды, и это может быть проблемой, если задержка зависит от расстояния передачи данных по очень большим векторам. Эта проблема для таких команд решается указанием длины вектора или расстояния передачи данных в отдельном операнде, который всегда использует регистровое поле RS. Данные сведения могут быть избыточными, ибо длина вектора хранится в векторном регистровом операнде, но планировщику они нужны как можно раньше. Другие регистровые операнды обычно не готовы, пока не истечёт такт, на котором они пройдут через исполнительный модуль, тогда как длина вектора обычно известна раньше. Микропроцессор может прочесть регистр RS на стадии вычисления адреса, на которой он также читает любые указатели, индексные регистры, и длину вектора для находящихся в памяти операндов. Это позволяет планировщику прогнозировать задержку за несколько тактов. Набор команд для всех команд, включающих в себя горизонтальную передачу данных (сюда входят команды broadcast с операндом, находящимся в памяти, permute, insert, extract, и команды сдвига, и не входят команды рассылки непосредственно заданных констант), предоставляет дополнительные сведения о длине вектора или расстоянии передачи данных в поле RS.

Путь данных к кэшу данных и памяти должен быть как можно более широким, возможно, с отслеживанием максимальной длины вектора, поскольку обращения к кэшу и памяти — типичные узкие места.

Глава 8. Модель памяти

Адресное пространство использует беззнаковые 64–разрядные адреса и 64–разрядные указатели. В будущем возможно расширение до 128–разрядных адресов, но это вряд ли будет уместно в обозримом будущем.

Абсолютные адреса используются редко. Большинство объектов данных, функций, и целей перехода адресуется с помощью смещения (разрядности, не большей 32) относительно некоторой точки отсчёта, содержащейся в 64–разрядном указателе. Этот указатель может быть указателем команд (IP), указателем секции данных (DATAP), указателем стека (SP), или регистром общего назначения.

Приложение может обращаться к следующим разделам данных.

- Код программы (CODE). Этот блок является исполняемым, с возможностью чтения, или без неё, но без возможности записи. Секция CODE может разделяться между разными процессами, выполняющими одну и ту же программу.
- Константные данные программы (CONST). Она содержит константы и таблицы, используемые программой, без возможности записи. Может разделяться между несколькими процессами.
- Статические разделы данных, доступные и для чтения, и для записи, которые могут содержать как инициализированные данные (DATA), так и неинициализированные (BSS). Эти разделы используются для глобальных данных и для статических данных, определённых внутри функций. Необходимо несколько экземпляров, если несколько процессов выполняет один и тот же код.
- Стековые данные (STACK). Данные раздел используется для нестатических данных, определённых внутри функций. Каждый процесс или поток имеет свой собственный стек, адресуемый относительно указателя стека. При добавлении данных в стек, тот растёт вниз, от старших адресов — к младшим.
- Куча программы (HEAP). Используется для динамического выделения памяти прикладной программой.
- Данные потока (THREADD). Выделяются, когда поток создаётся, и используются для локальных статических данных потока и для блока окружения потока.

Ссылки в пределах секции CODE используют 8–разрядные, 24–разрядные, и 32–разрядные знаковые ссылки относительно указателя команд, умножаемого на размер слова кода, равный 4 байтам.

Секцию CONST лучше помещать сразу после секции CODE. Данные в секции CONST в основном адресуются относительно указателя команд, без масштабирования[1].

Секции DATA и BSS адресуются относительно указателя секции данных (DATAP), который является специальным регистром, указывающим на некоторую точку отсчёта, расположенную в этих разделах. Предпочтительной точкой отсчёта является место, где DATA заканчивается, а BSS — начинается. Несколько выполняющихся экземпляров одной и той же программы будут иметь разные значения указателя секции данных. Секции CODE и CONST содержат не прямые ссылки на DATA или BSS, а лишь смещения относительно указателя секции данных, что делаем возможным нескольким процессам разделять одни и те же секции CODE и CONST, но иметь собственные закрытые секции DATA и BSS, без необходимости в трансляции виртуальных адресов. Секции DATA и BSS могут размещаться где угодно в адресном пространстве, независимо от того, где размещаются CONST и CODE.

Данные в секции STACK адресуются относительно указателя стека (SP). Данные, находящиеся в куче, адресуются через указатели, предоставленные функцией выделения памяти в куче.

Данные потока адресуются относительно регистра, называемого указателем блока окружения потока, который свой у каждого потока процесса. Блок окружения потока может выделяться в стеке, когда создаётся новый поток.

Секции данных STACK, DATA, BSS, HEAP и THREADD лучше хранить совместно, в непрерывном блоке, чтобы оптимизировать кэширование и управление памятью.

[1] В случае чистой гарвардской архитектуры, секция CONST может быть помещена в доступную для чтения память программ, чтобы адресовать относительно указателя команд, или может быть помещена в память данных, и адресоваться относительно DATAP

Данная модель позволяет программе обращаться к секции CODE размером вплоть до 8Гб, секции CONST — до 2Гб, секции DATA — до 2Гб, секции BSS — до 2Гб, секции THREADD — до 2Гб, секции STACK почти неограниченного размера, с кадрами по 2Гб, и почти неограниченному количеству данных в секции HEAP. Чтобы обращаться к данным из CONST в том редком случае, когда расстояние между кодом и данными превышает 2Гб, либо чтобы избежать перемещения адресов, в блоке окружения потока предоставляется указатель на секцию CONST.

Конец объединённого блока данных, находящихся в памяти, должен иметь неиспользуемое пространство того же размера, что и максимальная длина вектора. Это позволит команде restore_cp читать больше, чем необходимо, при восстановлении вектора неизвестной длины; а также позволит функции, находящей конец заканчивающейся нулём строки, читать по одному куску размером с длину вектора за раз, не вызывая нарушений доступа из–за чтения в недоступной части памяти.

У большинства микропроцессорных систем стек растёт вниз. У системы ForwardCom дело обстоит так же, но, в основном, по другой причине. Когда в стеке сохраняется векторный регистр, он сохраняется так: сначала идёт длина, а потом — количество данных, указанное длиной. Когда векторный регистр восстанавливается (с использованием команды restore_cp), необходимо прочесть длину, за которой идут данные. Указатель стека должен указывать на младший адрес, где хранится длина, иначе было бы невозможно найти, где длина хранится.

8.1. Защита памяти потока

У каждого потока обязательно должен быть свой собственный стек. Данные потока (THREADD) могут быть размещены в этом стеке. Система ForwardCom позволяет межпоточную защиту памяти. Стековые данные основного потока программы доступны всем её дочерним потокам, но все прочие потоки программы могут иметь закрытые данные, недоступные никаким другим потокам, даже основному. Любая коммуникация между потоками или синхронизация таковых обязательно должна использовать статическую память или память, принадлежащую основному потоку.

Рекомендуется использовать эту межпоточную защиту памяти во всех случаях, исключая устаревшее программное обеспечение, требующее единого пространства памяти, разделяемого всеми потоками.

Изолированные блоки памяти

Возможно создание системной функции, выделяющей изолированный блок памяти, с обеих сторон окружённого недоступной памятью. Такой блок памяти, который будет недоступен только конкретному потоку, может использоваться, например, для входного буфера, в случае высоких требований к безопасности. Каждый поток может иметь лишь ограниченное количество таких защищённых блоков памяти, из–за ограниченного размера карты памяти.

8.2. Управление памятью

Цель дизайна — минимизировать фрагментацию памяти и необходимость в трансляции виртуальных адресов. В имеющихся дизайнах часто имеются очень сложные системы управления памятью, с многоуровневой трансляцией адресов, большим буфером ассоциативной трансляции (translation-lookaside-buffers, TLB), и огромными таблицами страниц. Мы хотим заменить TLB, имеющий большое количество блоков памяти фиксированного размера, картой памяти, имеющей небольшое количество блоков памяти переменного размера. В большинстве случаев основному потоку приложения будет необходимо лишь три блока памяти CONST (только для чтения), CODE (только для выполнения), и объединённые STACK+DATA+BSS+HEAP (для чтения и записи). Дочернему потоку нужен ещё один блок, для своего закрытого стека. Аналогичные блоки определяются для системного кода.

Карту памяти с таким ограниченным количеством элементов можно легко реализовать на кристалле, очень эффективным способом, и можно легко изменять при переключении задач. Каждый процесс и каждый поток обязан иметь свою собственную карту памяти. Эта память не разбивается на страницы фиксированного размера.

Карта памяти поддерживает трансляцию виртуальных адресов, в виде константного смещения, для каждого элемента карты определяющего расстояние между виртуальным адресом и физическим. Аппаратуре не следует тратить время и энергию на трансляцию виртуальных адресов, когда она не используется.

В карте памяти предоставляется ограниченное количество дополнительных элементов, чтобы обработать случаи, когда память становится фрагментированной. Однако в большинстве случаев фрагментации памяти можно избежать. Для упрощения управления памятью и устранения фрагментации памяти предоставляются следующие приёмы:

- Есть только один тип библиотек функций, который можно использовать и для статической компоновки, и для динамической. Они компонуются с помощью механизма, в большинстве случаев сохраняющего непрерывность расположения секций CONST, CODE и DATA с аналогичными секциями основной программы. Данный приём описан ниже, на с. 76.
- Требуемый размер стека вычисляется компилятором и компоновщиком, так что в большинстве случаев переполнения стека можно избежать. Данный приём описан на с. 78.
- Операционная система может сохранять статистические записи о размере кучи каждой программы, чтобы предсказать требуемый размер кучи. Тот же самый приём можно использовать для предсказания размера стека, когда размер стека нельзя предсказать точно (например, при рекурсивных вызовах функций).

Несмотря на использование этих приёмов, пространство памяти может стать фрагментированным. Проблемы, который могут стать результатом фрагментации памяти, перечислены ниже.

- Рекурсивные функции могут использовать неограниченное пространство стека. Мы можем потребовать, чтобы программист указывал прагмой максимальный уровень рекурсии.
- Выделение в стеке памяти для массивов переменного размера с помощью C–функции alloca. Мы можем потребовать, чтобы программист указывал максимальный размер.
- Компоновка времени выполнения. Программа может резервировать место для загрузки и компоновки библиотек функций во время выполнения (см. с. 77). Память может стать фрагментированной, если памяти, зарезервированной для этой цели, оказалось недостаточно.
- Языки сценариев и языки, использующие байт–код. Сложно предсказать требуемый размер стека и кучи при выполнении интерпретируемого или эмулируемого кода. Рекомендуется вместо этого использовать компиляцию „на лету“. Самомодифицирующиеся сценарии компилироваться не могут. Та же проблема может возникнуть с большими определяемыми пользователем макросами.
- Непредсказуемое количество потоков без защиты. Требуемый размер стека для потока может быть вычислен заранее, но в некоторых случаях предсказание количества потоков, которые программа породит, может оказаться сложным. Много потоков будет, в основном, разделять одну и ту же секцию кода, но им нужны отдельные стеки. Стек потока, если используется межпоточная защита памяти, без проблем может располагаться где угодно в памяти. Однако, если память разделяется между потоками, и количество потоков — непредсказуемо, то область разделяемой памяти может стать фрагментированной.
- Непредсказуемый размер кучи. Программам, обрабатывающим большие объёмы данных, например, обрабатывающие мультимедиа, может потребоваться большая куча. Куча может использовать не смежные области памяти, но это потребует дополнительных элементов в карте памяти.
- Ленивая загрузка и перекрытия кода. Большая программа может иметь некоторые модули кода, которые редко используются, и загружаются только по необходимости. Ленивая загрузка может быть полезной для экономии памяти, но она может потребовать трансляции виртуальной памяти, и может вызвать фрагментацию памяти. Прямолинейное решение состоит в реализации таких модулей кода в виде отдельных исполняемых программ.
- Горячее обновление, т.е. обновление кода во время его работы.
- Разделяемая память для связи между процессами. Это, как объяснено ниже, требует дополнительных элементов в карте памяти.
- Выполнение многих программ. Память может стать фрагментированной, когда случайно загружается и выгружается, или сбрасывается в память, много программ разных размеров,

Возможным средством от переполнения стека и кучи является размещение STACK, DATA, BSS и HEAP совместно (в этом порядке), в адресном диапазоне с большими неиспользуемыми адресными пространствами ниже и выше, так что стек может расти вниз (в сторону младших адресов), а куча — вверх, в свободное место. Данный метод может устранить фрагментацию виртуального адресного пространства, но не физического. Фрагментацию физического адресного пространства можно вылечить перемещением данных из блока памяти недостаточного размера в другой блок, большего размера. У этого метода есть цена — задержка во времени при перемещении данных.

Если компоновка во время выполнения работает в памяти, и отсутствуют элементов карты памяти, то допускается смешивать секции CONST и CODE в одну общую секцию, с доступом и по чтению, и на выполнение. Если библиотека функций содержит константные данные, исходящие из недоверенного источника, тогда как код — доверен, то предпочтительнее поместить недоверенные данные в секцию DATA, а не CONST, чтобы предотвратить выполнение зловредного кода, помещённого в секции CONST.

Разделяемая память может использоваться, когда имеется необходимость передачи больших объёмов данных между двумя процессами. Один процесс разделяет часть своей памяти с другим процессом. Процессу–получателю нужен дополнительный элемент в своей карте памяти, с тем, чтобы для разделяемого блока памяти указать права на чтение и/или запись.Процессу, владеющему блоком разделяемой памяти, не нужно никаких дополнительных элементов в его карте памяти. Имеется ограничение на то, к скольким блокам разделяемой памяти приложение может получить доступ, ибо мы хотим сохранить карту памяти малой. Если одной программе нужно общаться с большим количеством других программ, то мы можем использовать одно из следующих решений: (1) позволить программе, которой нужно много связей, владеть разделяемой памятью, и дать каждому из её клиентов доступ к одной из частей этой памяти; (2) выполнять в программе, которой нужно много связей, много потоков, или выполнять много экземпляров такой программы, так, чтобы каждый поток имел доступ только к одному разделяемому блоку памяти; (3) позволить много каналов коммуникации, использующих один и тот же блок разделяемой памяти, или части его; (4) общаться через вызовы функций; (5) общаться через сетевые сокеты; или (6) общаться через файлы.

Исполняемая память не может разделяться между разными приложениями. Если одному приложению нужно вызвать функцию другого приложения, обязательно нужно использовать механизм межпроцессных вызовов, описанный на с. 69.

Используя обсуждённые здесь принципы, мы можем, вероятно, сохранить фрагментацию памяти столь малой, что для охвата нормальных случаев для каждого потока будет достаточно малой карты памяти, что будет намного эффективнее, нежели большой TLB и многоуровневая трансляция адресов имеющихся дизайнов. Это сэкономит объём кремния и энергию, и мы можем избежать платы за промахи TLB и отсутствие страниц, что сделает переключение задач очень быстрым.

Глава 9. Системное программирование

Системные команды еще не полностью определены. Имеется больше работы по созданию эффективного дизайна системы. Однако первые экспериментальные реализации ForwardCom будут без операционной системы, поэтому дизайн системы ещё не исправлен. Предпочтительнее потратить больше времени на оптимизацию дизайна системы, а не на определение полного стандарта на данной ранней стадии разработки.

Следует иметь по крайней мере три разных уровня привилегий:

- Ядро системы, имеющее высочайший уровень привилегий. Управление памятью и планирование потоков размещаются здесь. Это единственная часть, которая может модифицировать карту памяти и управлять правами доступа на более низких уровнях.

- Драйвера устройств и подключаемые модули системы имеют тщательно управляемые права доступа. Структура, подобная карте памяти (см. с. 64) даёт драйверу устройства доступ к конкретному диапазону необходимых драйверу портов ввода–вывода и системных регистров. Пользовательское приложение может дать драйверу устройства права на чтение и запись для конкретного диапазона памяти данных, которым оно владеет. Это выполняется через команду системного вызова. У драйвера устройства нет прав на память кода вызвавшего его приложения. Последнее означает, что указатели на функции обратного вызова с системными вызовами использоваться не могут.

- Прикладная программа имеет доступ только к памяти, которая ею выделена или с ней разделяется. Память, принадлежащая потоку, обычно не разделяется с другими потоками того же процесса. Прикладные программы имеют доступ к немногим системным регистрам и не имеют доступа к портам ввода–вывода.

Переходы между этими уровнями управляются командами системного вызова и системного возврата, а также ловушками и прерываниями.

Для целей управления имеются различные системные регистры. Дополнительно имеется два набора регистров, используемых для временного хранения, один набор — для уровня драйверов устройств, и один — для уровня ядра системы. Временные регистры для уровня драйверов устройств при каждом вызове драйвера устройства очищаются, по причинам, связанным с безопасностью. Эти регистры, в основном, используются для временного сохранения регистров общего назначения.

9.1. Карта памяти

Имеется три вида доступа к памяти: чтение, запись, и выполнение. Эти виды доступа различны, но могут комбинироваться. Например, доступ на выполнение не влечёт доступа по чтению. Доступ на запись и доступ на выполнение обычно комбинировать не следует, поскольку самомодифицирующийся код — отталкивающ.

Карта памяти сохраняется на кристалле ЦП. Каждый элемент имеет три поля: виртуальный адрес (вплоть до 64 разрядов), права доступа (3 разряда), и слагаемое для трансляции адреса (вплоть до 64 разрядов). Разбиения памяти на страницы нет. Вместо этого имеются блоки памяти переменных размеров.

Элементы карты памяти обязаны всё время храниться в отсортированном виде, так, чтобы каждый блок памяти заканчивался там, где начинается следующий блок. Адреса обязательно должны быть кратны 8. Каждый поток имеет свою собственную карту памяти. Типичная карта памяти для потока приложения может выглядеть примерно так:

Таблица 9.1.1. Пример карты памяти

Начальный адрес	**Права**	**Слагаемое**	**Комментарий**
0x10000	чтение	0	Секция CONST.
0x10100	выполнение	0	Секция CODE.
0x10800	нет	0	Принадлежит другим процессам.
0x20000	чтение, запись	0	Основные разделы STACK, DATA, BSS, HEAP.
0x24000	нет	0	Принадлежит другим процессам.

0x30000	чтение, запись	0	Секция STACK потока, блок окружения потока, и статические данные потока.
0x32000	нет	0	Остальное принадлежит другим процессам.

Может быть некоторое количество дополнительных элементов, для блоков памяти, разделяемых между процессами, и для безопасно изолированных блоков памяти. В случае, когда память становится фрагментированной, блок виртуальной памяти может иметь несколько элементов. Слагаемые используются для того, чтобы сохранить виртуальные адреса блоков смежными, в то время как физические адреса смежными не являются. Начальные адреса — это адреса в виртуальной памяти.

Размер карты памяти — переменный. Максимальный размер зависит от реализации. На кристалле имеется по меньшей мере три карты памяти, по одной для каждого уровня привилегий, что делает переходы между уровнями быстрыми. Пространство на кристалле, используемое для карты памяти. Область на кристалле, используемая для карт памяти, может быть реконфигурируемой, так что в случае, когда карты памяти малы, карты памяти многих процессов могут оставаться на кристалле.

Карты памяти управляются уровнем ядра системы. Команды read_memory_map и write_memory_map используют механизм векторного цикла для быстрой манипуляции картами памяти.

Методы устранения фрагментации памяти, описанные на с. 64, важны для сохранения карт памяти малыми.

Переключения задач будут очень быстрыми, поскольку мы заменили большие таблицы страниц и буферы ассоциативной трансляции (translation-lookaside-buffer, TLB) традиционных систем малой хранящейся на кристалле картой памяти, что делает систему подходящей для операционных систем реального времени.

9.2. Стек вызовов

Возможно наличие либо унифицированного стека (и для данных функций, и для адресов возврата), либо двух отдельных стеков. См. с. 53. В настоящее время ForwardCom поддерживает обе системы. Двухстековая система — безопаснее и эффективнее, тогда как одностековая система может использоваться для малых процессоров, где предпочтительнее более простая одностековая система.

У двухстековой системы имеется стек вызовов, сохраняемый внутри ЦП, а не в ОЗУ. Требуется метод сохранения этого стека, когда он заполнен. Данный метод схож с методом, используемым для сохранения карты памяти, как описано выше, использующим векторные обращения к памяти. Должна быть возможна манипуляция со стеком вызовов, для переключений задач и для раскрутки стека в обработчике исключения.

9.3. Системные вызовы и системные функции

Вызовы системных функций выполняются с помощью команды системного вызова (sys_call). Команда системного вызова использует не адреса, а номера, ID. Каждый номер (ID) состоит из (в младшей половине) ID функции, и (в старшей половине) ID модуля. ID модуля идентифицирует модуль системы или драйвер устройства. У ядра системы ID = 0. Каждая часть ID может быть либо 16–разрядной, либо 32–разрядной, так что скомбинированный ID может быть либо 32–разрядным, либо 64–разрядным.

Дополнительным модулям системы и драйверам устройств нет необходимости иметь фиксированные номера ID, поскольку это потребовало бы некоторого центрального авторитета, присваивающего эти номера. Вместо этого программа будет должна запросить номер ID, задав имя модуля. Функции в пределах модуля могут иметь фиксированные или переменные номера ID.

Будет системная функция (с фиксированным номером ID), принимающая в качестве аргументов имя модуля и функции, и возвращающая номер ID. Номер ID таким способом может быть подобран при первом вызове этой функции.

Номер ID системной функции может быть помещён в программу тремя способами:

1) Наиболее важные системные функции имеют фиксированные номера ID, которые можно вставить во время компиляции.

2) Номер ID может быть найден во время загрузки, тем же способом, что и при компоновке во время загрузки. Это описано на с. 77. Загрузчик найдёт номер ID и вставит его в код перед запуском программы.

3) Номер ID находится во время выполнения, перед первым вызовом желаемой функции.

Соглашения вызова для системных функций — те же, что и для других функций, использующих регистры для параметров и для возврата значения. Регистры, используемые для параметров, определяются общим соглашением вызова. Соглашения вызова описаны на с. 73. Регистры для параметров не следует путать с операндами для команды системного вызова.

Команда системного вызова имеет три операнда. Первый операнд — скомбинированный ID, помещённый в регистр (RT) или в непосредственно заданную константу. Второй операнд (RD) — указатель на блок памяти, который может быть использован для передачи данных между вызывающей программой и системной функцией. Третий параметр (RS) — размер этого блока памяти. Последние два параметра должны быть кратны 8.

Вызывающий поток обязан иметь право на доступ к блоку памяти, который он разделяет с системной функцией. Это может быть право на чтение, право на запись, или и на то, и на другое. Эти права доступа передаются системной функции. У системной функции нет прав доступа ни к какой другой части памяти приложения.

С системными вызовами невозможно использовать указатели на функции обратного вызова, поскольку исполняемая память с системной функцией разделяться не может. Вместо этого системная функция может вызвать экспортируемую функцию, предоставляемую приложением, используя метод для межпроцессных вызовов, описанный ниже.

Функциям драйвера устройства предпочтительнее иметь отдельные стеки. Системный вызов идёт сначала в ядро системы, которое назначает стек для функции драйвера устройства и создаёт карту памяти для неё перед перенаправлением вызова на желаемую функцию. Во время этого перенаправления предпочтительнее не использовать никакого стека. Два регистра, идентифицирующих блок разделяемой памяти, копируются в специальные регистры, доступные вызванной функции. Системная функция выполняется в том же потоке, что и вызвавшее её приложение, но не с тем же самым стеком.

Старые значения указателя команд, указателя стека, регистра DATAP и карты памяти сохраняются в системных регистрах, чтобы быть восстановленными командой системного возврата.

Системным функциям, драйверам устройств, и обработчикам прерываний разрешается использовать все регистры общего назначения и векторные регистры, если они сохраняются и восстанавливаются согласно нормальным соглашениям вызова. Обработчики прерываний обязаны сохранять и восстанавливать все регистры, которые они используют.

Предоставляется метод для получения сведений об использовании регистров системной функцией, так что возможен их вызов с помощью соглашений об использовании регистров или методом 1, или методом 2, описанным на с. 74. Использование стека системными функциями не имеет никакого отношения к вызывающему, ибо они не используют стек вызова прикладной программы.

Некоторые важные системные функции обязательно должны быть стандартизированы и доступны во всех операционных системах, что сделает возможным, например, создание сторонней библиотеки функций, работающей во всех операционных системах, даже если этой библиотеке нужно вызывать системные функции. Это также облегчит адаптацию программ для разных операционных систем. Список системных функций, которые могли бы быть стандартизированы, включает функции для создания потоков, синхронизации потоков, установки приоритета потоков, выделения памяти, измерения времени, системные сведения, обращение к переменным окружения, и т.д.

Следовало бы отобрать системные библиотеки, предоставляющие наиболее распространённые виды пользовательского интерфейса, такие, как графический пользовательский интерфейс, консольный режим, и серверный режим. Эти библиотеки пользовательского интерфейса следует предоставлять для каждой операционной системы, которая может работать на нашей архитектуре, так что одна и та же исполняемая программа может работать в разных операционных системах, просто посредством компоновки, в момент загрузки, с подходящей библиотекой пользовательского интерфейса. Такие библиотеки пользовательского интерфейса могут быть основаны на существующих платформенно–независимых библиотеках GUI, таких, как, например, wxWidgets или QT. Все библиотеки пользовательского интерфейса обязаны поддерживать функцию error_message, упомянутую ниже.

9.4. Межпроцессные вызовы

Межпроцессные вызовы опосредуются системной функцией. Работает это следующим образом. Прикладная программа может экспортировать функцию с элементом в заголовке своего исполняемого файла. Другое

приложение может получить доступ к этой экспортированной функции посредством вызова системной функции, проверяющей разрешения и переключающей карту памяти, регистры DATAP и THREADP, и указатель стека перед вызовом экспортированной функции, и переключающей обратно перед возвратом в вызвавшую программу. Вызов возникнет как отдельный поток вызванной программы. Регистры общего назначения и векторные регистры могут использоваться для передачи параметров и возвращаемого значения так же, как это было бы для нормальных функций. Данный механизм не порождает никакой разделяемой вызвавшим и вызывающим памяти. Следовательно, экспортируемая функция обязана использовать лишь простые типы, помещающиеся в регистры, для своих параметров и для возвращаемого значения. Блок памяти может разделяться между двумя процессами так, как описано на с. 66.

9.5. Обработка сообщений об ошибках

Имеется необходимость в стандартизированном способе сообщения об ошибках, возникающих в программе. Многим имеющимся системам не удаётся удовлетворить эту потребность, или они используют методы, которые либо не переносимы, либо не являются потокобезопасными. В частности, от такого стандарта выиграли бы следующие ситуации.

1) Библиотека функций обнаруживает ошибку, например, неверный параметр, и ей нужно сообщить об ошибке вызывающей программе. Вызывающая программа примет решение: восстановиться ли после ошибки, или завершить работу.

2) Из–за числовой ошибки порождается ловушка. Программе не удаётся перехватить её как исключение, или в языке программирования нет поддержки структурной обработки исключений. Операционная система обязана выдать информативное сообщение об ошибке.

3) Программа может выполняться в разных окружениях, требующих разных форм обработки ошибок.

4) Библиотеке функций в форме исходного кода, библиотеки классов, или любой другой части кода, необходимо сообщать об ошибке, не зная, какая используется парадигма пользовательского интерфейса (например, консольный режим или графический пользовательский интерфейс). Необходим стандартизированный способ передачи сообщения об ошибке в операционную систему или каркас пользовательского интерфейса, которые обязаны представить сообщение об ошибке пользователю способом, подходящим для пользовательского интерфейса (например, всплытие окна с сообщением, печать в stderr, печать в лог–файл, или посылка сообщения администратору).

Для этой цели предлагается определить стандартную библиотечную функцию, с именем error_message. Все каркасы пользовательского интерфейса обязаны определять эту функцию. Возможен автоматический выбор различных версий этой функции во время выполнения, в зависимости от настроек системы, используя диспетчеризацию, описанную на с. 78. Головная программа может перекрывать эту функцию, определяя свою собственную функцию с тем же именем.

Функция error_message обязана иметь следующие параметры: числовой код ошибки; указатель на строку, предоставляющую сообщение об ошибке; и указатель на другую строку, дающую имя функции, в которой возникла ошибка. Эти строки представляют собой завершающиеся нулём строки, в кодировке UTF-8. Сообщение об ошибке — по умолчанию на английском языке. Поддержки многих разных языков (см. эту ссылку по поводу обсуждения проблем с интернационализацией) требовать неразумно. Вместо этого руководство на желаемом языке может содержать список кодов ошибок.

Строка сообщения об ошибке может включать числовые значения и диагностические сведения, такие, как значение параметра, находящегося вне диапазона.

Функция error_message может возвращать управление, а может и не возвращать. Если она возвращает управление, то вызвавшая функция обязана изящно вернуть управление. Как вариант, функция error_message может завершить приложение, либо может возбудить исключение или ловушку, которые обрабатываются операционной системой, в случае, если исключение не перехвачено программой.

Глава 10. Стандартизация ABI и программной экосистемы

Цель ForwardCom — вертикальное перепроектирование, определяющее новые стандарты, не только для набора команд, но также и для использующего его программного обеспечения, что будет иметь следующие преимущества.

- Будут совместимы разные компиляторы. Одни и те же библиотеки функций смогут использоваться с разными компиляторами.
- Будут совместимы разные языки программирования. Будет возможной компиляция разных частей программы на разных языках программирования. Будет возможной компиляция библиотеки функций на языке программирования, отличающемся от использующей её программы.
- Будут совместимы отладчики, профилировщики, и другие средства разработки.
- Будут совместимы разные операционные системы. Будет возможным использование одних и тех же библиотек функций в разных операционных системах, за исключением случая использования специфичных для системы функций.

В предыдущей главе была описана стандартизация системных вызовов, системных функций, и сообщений об ошибках. В настоящей главе обсуждается стандартизация следующих аспектов программной экосистемы.

- Поддержка компилятором.
- Представление двоичных данных.
- Соглашения вызова для функций.
- Соглашения об использовании регистров.
- Искажение имён для перегрузки функций.
- Двоичный формат объектных файлов и исполняемых файлов.
- Формат и методы компоновки для библиотек функций.
- Обработка исключений и раскрутка стека.
- Отладочная информация.
- Синтаксис языка ассемблера.

10.1. Поддержка компилятором

Компилятор может иметь три разных уровня поддержки векторных регистров переменной длины.

Уровень 1

Компилятор не будет использовать векторы переменной длины. Компилятор может вызывать векторную функцию, находящуюся в библиотеке функций, со скалярным параметром, если в скалярной версии функция недоступна.

Уровень 2

Компилятор может вызывать векторные функции, но не порождать такие функции. Компилятор может автоматически векторизовать цикл и вызывать векторную библиотечную функцию из такого цикла.

Уровень 3

Полная поддержка. Компилятор поддерживает типы данных для векторов переменной длины. Эти типы данных могут использоваться для переменных, параметров функций, и возвращаемых функциями значений. Векторы переменной длины не могут включаться в структуры, классы, или объединения, поскольку такие составные типы должны иметь известные размеры. Поддержка векторов переменной длины в статических и глобальных переменных — необязательна. Общие операции над векторами переменной длины могут указываться явно, включая опции для применения булевых векторных масок.

Другие качества компилятора

Компилятор может поддерживать арифметику указателей для указателей на функции, чтобы явно писать компактные таблицы вызовов с относительными адресами. Разность между двумя указателями на функции следует умножать на размер слова кода, равный 4 байтам. Без этой возможности указатели на функции должны приводиться к типу целочисленных указателей, и обратно.

Компилятор может иметь поддержку обнаружения целочисленного и вещественного переполнений, и других численных ошибок, в блоках try–catch, используя один из методов, обсуждённых на с. 55.

Компилятор может поддерживать проверку границ массивов, используя индексный адресный режим с границами или команду условной ловушки.

10.2. Представление двоичных данных

Данные в ОЗУ хранятся в формате с прямым порядком байтов. По поводу обоснования см. с. 53.

Целочисленные переменные представляются 8, 16, 32, 64, и, необязательно, 128 разрядами, знаковые и беззнаковые. Знаковые целые числа используют представление в виде дополнения до двух. Целочисленное переполнение приводит к заворачиванию, за исключением команд для арифметики с насыщением.

Вещественные числа кодируются с одинарной (32 разряда), двойной (64 разряда), и, необязательно, четырёхкратной (128 разрядов) точностью, следуя стандарту IEEE Standard 754-2008 или более позднему. Половинная точность (16 разрядов) необязательна, и используется в непосредственно заданных константах. Вычисления с половинной точностью не поддерживаются, но преобразования между половинной и одинарной точностью — (необязательно) поддерживаются.

Как обсуждалось на с. 55, вещественные переменные со значениями NAN могут содержать диагностические сведения о том, что вызвало ошибку.

Логические переменные хранятся как целые числа, по меньшей мере 8–разрядные, со значениями 0 и 1 для FALSE и TRUE. У логической переменной используется только разряд №0, тогда как другие разряды игнорируются. Это правило делает возможным использование логических переменных в качестве масок, и для реализации логических функций, таких, как И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ, и НЕ, с помощью простых поразрядных команд, а не с помощью метода, используемого во многих нынешних системах, имеющих ветвление для каждой проверки переменной, если всё целое число — не ноль. Команда ветвления нужна при компиляции выражений вида (A && B) и (A || B), только если вычисление B имеет побочные эффекты.

Все переменные, размер которых не более 8 байт, следует хранить, используя их естественное выравнивание.

Массивы, размер которых не меньше 8 байт, обязаны храниться по адресам, кратным 8. Можно рекомендовать выравнивать большие массивы на размер строки кэша.

Многомерные массивы хранятся по строкам, за исключением тех языков программирования, которые делают это невозможным.

Текстовые строки могут храниться в зависящей от языка форме, но для системных функций и для функций, предназначенных для совместимости со всеми языками программирования, необходима стандартизированная форма. Предлагаемый стандарт использует кодировку UTF-8. Длина строки может определяться завершающим нулём, спецификатором длины, или и тем, и другим. Обоснование таково. Для текстовых строк, подходящих для чтения человеком, время обработки процессором в большинстве случаев несущественно, хотя менее компактно для некоторых азиатских языков. UTF-8 — наиболее широко используемая в интернете кодировка.

10.3. Дальнейшие соглашения для объектно–ориентированных языков

Объектно–ориентированные языки требуют дальнейших стандартов, для двоичного представления специальных черт, таких, как таблицы виртуальных функций, определение типа во время выполнения, и т.п.

Эти детали обязательно должны быть стандартизированы в рамках каждого языка программирования, ради совместимости разных компиляторов, и, если возможно, также разных языков программирования, имеющих совместимые свойства.

Указатели на члены следует реализовывать способом, для которого важнее хорошая производительность в общем случае, когда требуется лишь простое смещение (для данных) или простой указатель (для функции), тогда как дополнительные сведения для запутанного случая множественного наследования добавляются только тогда, когда это необходимо.

10.4. Соглашения о вызовах функций

Вызовы функций будут использовать регистры для параметров, насколько это вообще возможно. Целые числа размером до 64 разрядов, указатели, ссылки, и булевы скаляры передаются в регистрах общего назначения. Векторные параметры могут иметь переменную длину. Вещественные скаляры, векторы с элементами любого типа, имеющие фиксированную длину, вплоть до 16 байт, и векторы переменной длины передаются в векторных регистрах.

Первые 16 параметров функции, помещающиеся регистры общего назначения, передаются в регистрах r0–r15. Первые 16 параметров, помещающиеся в векторные регистры, передаются в регистрах v0–v15. Длина векторного параметра переменной длины содержится в том же векторном регистре, который содержит данные.

Составные типы передаются в векторных регистрах, если их можно рассматривать как „простые кортежи“, размером не более 16 байт. Простой кортеж — это структура или класс, или инкапсулированный массив, для которого все элементы имеют один и тот же тип, не являющийся указателем. Объединение трактуется как структура, соответствующая своему первому элементу.

Параметры, не помещающиеся в один регистр, передаётся через указатель на находящийся в памяти объект, память под который выделена вызывающим. Это применимо к структура и классам с элементами разных типов, или имеющим размер, больший 16 байт, а также применимо к объектам, требующим специальной обработки, такой, как нестандартные копирующий конструктор или деструктор, и к объектам, требующим дополнительной неявной памяти, таким, как виртуальные функции–члены. Вызов любого копирующего конструктора или деструктора — дело вызывающего.

Если для всех параметров не хватает регистров, то дополнительные параметры предоставляются в виде списка, который может храниться где–то в памяти. Указатель на этот список параметров передаётся в регистре общего назначения. Такой список также используется, если список параметров имеет переменную длину. Более одного списка параметров быть не может, поскольку один и тот же список используется для всех целей.

Правила для списка параметров следующие. Список параметров используется, если имеется более 16 параметров, помещающихся в регистры общего назначения; если имеется более 16 параметров, помещающихся в векторные регистры; или список параметров имеет переменную длину. Если имеется менее 16 параметров, помещающихся в регистрах общего назначения, то эти параметры помещаются в регистры общего назначения, а следующий свободный регистр общего назначения используется как указатель на список параметров. Если имеется 16 или более параметров, помещающихся в регистрах общего назначения, и по какой–либо причине нужен список параметров, то первые 15 параметров, помещающихся в регистрах общего назначения, помещаются в r0–r14, указатель на список — в r15, а оставшиеся параметры, помещающиеся в регистры общего назначения, — помещаются в список. Если имеется более 16 векторных параметров, то первые 16 векторных параметров помещаются в v0-v15, а оставшиеся векторные параметры — помещаются в список. Все параметры в список помещаются в том порядке, в каком появляются в определении функции, независимо от типа. Переменные аргументы помещаются в список последними, поскольку в определении функции всегда появляются последними.

Список состоит из элементов, каждый из которых имеет размер 8 байт. Параметр, помещающийся в регистр общего назначения, использует один элемент. Векторный параметр, имеющий постоянную длину, не превосходящую 8 байт, использует один элемент. Векторный параметр, имеющий постоянную длину, большую 8 байт, или имеющий переменный размер, использует два элемента. Первый элемент — длина (в байтах), а

второй — указатель на массив, содержащий вектор. Параметр, который не поместился бы в регистр, если бы таковой был свободен, передаётся в виде указателя на список, согласно тем же правилам, что и указатель, передаваемый в регистре.

Список параметров принадлежит вызванной функции, в том смысле, что ей разрешается модифицировать параметры списка, если они не описаны как константные. То же применимо к массивам и объектам с указателем в списке. Вызывающий может полагаться на неизменность параметров списка только в том случае, если они описаны как константные. Вызывающий не должен помещать список в месте, в котором он не может модифицироваться другими потоками.

Возвращаемое функцией значение находится в r0 или v0, используя те же правила, что и для параметров функций. Множественные возвращаемые значения (если они допускаются языком программирования), если это возможно, трактуются как кортежи, и возвращаются в v0. Множественные возвращаемые значения разных типов могут возвращаться в нескольких регистрах, но, как правило, лучше трактовать множественные возвращаемые значения как структуру, ради совместимости с другими языками программирования, не позволяющими множественные возвращаемые значения.

Возвращаемое значение, не помещающееся в регистр, возвращается в области, выделенном вызывающей функцией, через указатель, переданный вызывающей функцией в r0, и возвращаемый в r0. Любые конструкторы вызываются вызванной функцией.

Указатель „this“ для функции, являющейся членом класса, передаётся в r0, кроме случая, когда r0 используется для возвращаемого объекта. Тогда „this“ передаётся в r1.

Обоснование

Намного эффективнее передавать параметры в регистрах, чем через стек. Настоящее предложение позволяет вплоть до 32 параметров, включая векторы переменной длины, передавать в регистрах, оставляя 15 регистров общего назначения и 16 векторных регистров для использования функцией в других целях. Это охватывает почти все практические случаи, так что параметры редко потребуется хранить в памяти.

Всё же у нас обязательно должны иметься точные правила для охвата неограниченного количества параметров, если в языке программирования нет ограничения на количество параметров. Мы помещаем любые дополнительные параметры в список, а не в стек, как делает большинство других систем. Основная причина этого состоит в том, чтобы сделать программное обеспечение независящим от того, есть ли отдельный стек вызова, или один и тот же стек используется и для адресов возврата, и для локальных переменных. Адреса параметров, расположенных в стеке, зависели бы от того, имеется ли в том же самом стеке адрес возврата, или нет. Метод списка имеет и другие преимущества. Не будет рассогласования порядка параметров в стеке и с тем, очищается ли стек вызывающей функцией или вызываемой. Список может повторно использоваться вызывающей функцией для многократных вызовов, если параметры являются константами, а вызываемая функция может повторно использовать список параметров переменной длины, передавая его в другую функцию. Гарантируется, что функция выполняет возврат управления должным образом, без порчи стека, даже если вызывающая и вызываемая функции не согласованы по количеству параметров. Хвостовые вызовы возможны во всех случаях, независимо от количества и типов параметров.

10.5. Соглашения об использовании регистров

У большинства систем имеются правила, гласящие, что некоторые регистры имеют статус сохраняемых вызываемой функцией. Это означает, что функция должна сохранять эти регистры и восстанавливать их перед возвратом управления, если они используются. Вызывающая функция может тогда полагаться на то, что эти регистры остаются неизменными после вызова функции.

У нынешних систем имеется проблема с назначением статуса сохраняемых при вызове для векторных регистров. Последующие версии ЦП могут иметь более длинные векторные регистры, а команды сохранения более длинных регистров ещё не были определены. У некоторых систем теперь, из–за неудачного прогноза, есть статус сохраняемой при вызове для части векторного регистра. В имеющихся системах невозможно сохранять векторные регистры способом, который был бы совместим с последующими расширениями.

В дизайне ForwardCom эта проблема решается с помощью векторов переменной длины. Можно сохранять и восстанавливать векторный регистр любой длины, даже если эта длина не поддерживалась на момент компиляции кода. Также можно узнать, насколько длинные векторные регистры используются на самом деле, поскольку длина вектора сохраняется в самом регистре, так что нам нужно сохранять лишь ту часть регистра, которая действительно используется. Для этой цели спроектированы команды save_cp и restore_cp

(см. с. 41). Неиспользуемые векторные регистры для сохранения будут использовать лишь небольшое пространство.

Для сохранения векторных регистров, если они длинны, всё ещё требуется много места в кэше. Следовательно, мы хотим минимизировать необходимость сохранения регистров. Предлагается иметь, на выбор, два разных метода сохранения, описываемых ниже.

Метод 1

Этот метод используется по умолчанию, может использоваться во всех случаях, но не самый эффективный. Правило простое: регистры r16–r31 и v16–v31 имеют статус сохраняемых при вызове.

Функция может свободно использовать регистры r0–r15 и v0–v15. Шестнадцать регистров каждого типа — достаточно для большинства функций. Если функции нужны дополнительные регистры, то она обязана их сохранять.

Все системные регистры и специальные регистры имеют статус сохраняемых при вызове, исключая функции, предназначенные для манипулирования с этими регистрами.

Метод 2

Он будет более эффективным, если мы действительно знаем, какие регистры используются в каждой функции. Если функция A вызывает функцию B, и A знает, какие регистры использует B, то A может просто выбрать некоторые регистры, которые не используются функцией B, для любых данных, которые не должны изменяться после вызова функции B. Даже для длинной цепочки вложенных вызовов функций можно избежать необходимости в сохранении каких–либо регистров, до тех пор, пока регистров достаточно.

Если функции A и B компилируются совместно, в одном и том же процессе, то компилятор легко может управлять этими сведениями. Но если A и B компилируются раздельно, то нам необходимо сохранять необходимые сведения о том, какие используются регистры, что возможно для формата объектных файлов, описанного на с. 76. Сведения об используемых регистрах обязательно должны сохраняться в откомпилированном объектном файле или файле библиотеки, а не в каком–либо ином файле, который мог бы быть и рассинхронизирован.

Функцию B предпочтительнее откомпилировать в объектном файле первой. Этот объектный файл обязательно должен содержать сведения о том, какие регистры модифицируются функцией B. Необходимые сведения — просто 64–разрядное число, с одним разрядом на каждый модифицируемый регистр (разряды №№0–31 — для r0–r31, и разряды №№32–63 — для v0–v31). Любые регистры, используемые для передачи параметров и возврата значения из функции, также помечаются, если они модифицируются функцией.

Когда затем компилируется функция A, то компилятор просмотрит объектный файл для функции B, чтобы посмотреть, какие регистры та модифицирует. Компилятор выберет некоторые из регистров, не модифицируемых функцией B, для данных, которые не должны измениться после вызова функции B. Регистры, используемые функцией B, в функции A могут преимущественно использоваться для временных переменных, которые не нужно сохранять перед вызовом функции B. Подобным образом, будет выгодно использовать те же самые регистры для многих временных переменных, если их времена жизни не перекрываются, чтобы модифицировать как можно меньше регистров. Объектный файл для A будет содержать список регистров, модифицируемых функцией A, включая все регистры, модифицируемые функцией B, и любой другой функцией, которую может вызвать A. Объектный файл для A содержит ссылку на функцию B. Эта ссылка обязательно должна содержать сведения о том, какие регистры функция A ожидает модифицируемыми функцией B. Если B позднее перекомпилируется, и новая версия функции B модифицирует больше регистров, то компоновщик обнаружит нестыковку, и сообщит о перекомпиляции функции A.

Если, по какой–то причине, функция A компилируется перед функцией B, либо во время компиляции функции A сведения о B недоступны, то компилятор сделает предположения об использовании регистров функцией B. Предположение, принятое по умолчанию, указано в методе 1. Функция A может позднее перекомпилирована, если функция B не соответствует этим допущениям, или просто для улучшения эффективности.

Если две функции, A и B, взаимно вызывают друг друга, то самым лёгким решением является положится на метод 1. Функциям, всё ещё, следует включать сведения об использовании регистров в свои объектные файлы.

Компилятору следует предпочесть выделение сначала меньшего числа регистров, чтобы минимизировать проблему, заключающуюся в использовании разных регистров разными библиотечными функциями. Можно (не обязательно) в вызывающем пропустить r7 и v7, чтобы использовать для масок.

Головной функции программы разрешается использовать метод 2, и модифицировать все регистры, если она включает необходимые сведения в свой объектный файл.

Объектные файлы, содержащиеся в библиотеке функций, должны включать сведения об использовании регистров.

К системным функциям и драйверам устройств обращаться так же, как к нормальным библиотечным функциям, нельзя (см. с. 68). Системные функции обязаны подчиняться правилам метода 2, но системе следует предоставить метод получения сведений об использовании регистров каждой системной функцией, что может быть полезно для компиляции „на лету„.

10.6. Искажение имён для перегрузки функций

Языки программирования, поддерживающие перегрузку функций, используют внутренние имена, с добавленными к именам функций префиксами и суффиксами, чтобы различать функции с одним и тем же именем, но разными параметрами или в разных классах, или в разных пространствах имён. Используется много разных схем искажения имён, и некоторые из них — недокументированы. Необходимо стандартизировать схему искажения имён, чтобы сделать возможным смешивание разных компиляторов или разных языков программирования.

Наиболее распространённые схемы искажения имён — схемы Microsoft и Gnu. Схема Microsoft использует символы, которые не могут встречаться в именах функций (?@$), что предотвращает конфликты имён, но делает невозможным прямой вызов функции с искажённым именем, или трансляцию, например, C++ в C. Схема Gnu порождает искажённые имена, выглядящие неуклюже, но не содержащие специальных символов, мешающих непосредственному вызову искажённого имени. Следовательно, предложение состоит в использовании схемы искажения имён Gnu (версии 4 или более поздней), с необходимыми добавлениями для векторов переменной длины и т.п.

Функции с искажёнными именами могут (не обязательно) дополнять искажённое имя простым (неискажённым) именем, в виде слабого публичного псевдонима в объектном файле. Это облегчит вызов функции из других языков программирования, в которых искажения имён нет. Слабая компоновка псевдонима предотвращает выдачу компоновщиком сообщений о дублирующихся именах, разве что вызов по имени неоднозначен.

10.7. Двоичный формат для объектных файлов и исполняемых файлов

Формат исполняемого файла должен быть стандартизирован. Из широко распространённых форматов, наиболее гибким и хорошо структурированным, является, вероятно, формат ELF. Для объектных файлов, библиотек функций, и исполняемых файлов предлагается использовать формат ELF.

Детали формата ELF для ForwardCom указаны в файле с именем elf_forwardcom.h. Эта спецификация включает детали типов секций, типы переразмещений, и т.п. В формат файла добавлены дополнительные сведения, об использовании регистров (см. с. 74) и использовании стека (см. с. 78).

Имена файлов обязательно должны иметь расширения, указывающие их тип. Предполагается использовать следующие расширения: ассемблерный код — .as, объектный файл — .ob, библиотечный файл — .li, исполняемый файл — .ex.

10.8. Библиотеки функций и методы компоновки

Динамически компонуемые библиотеки (DLL) и разделяемые объекты (SO) в системе ForwardCom не используются. Вместо этого мы будем использовать только один тип библиотек функций, который можно использовать тремя разными способами.

1) Статическая компоновка. Компоновщик находит в библиотеке требуемые функции и копирует их в исполняемый файл. Включаются лишь те части библиотеки, которые действительно нужны конкретной головной программе. Этот способ в нынешних системах обычно используется для статических библиотек (.lib–файлы в Windows, .a–файлы в Uni–подобных системах, таких, как Linux, BSD, и Mac OS).

2) Компоновка во время загрузки. Библиотека может распространяться отдельно от исполняемого файла. Требуемые части библиотеки загружаются в память совместно с исполняемым файлом, и все ссылки между головным исполняемым файлом и библиотечными функциями разрешаются загрузчиком тем же способом, что и при статической компоновке.

3) Компоновка во время выполнения. Выполняющаяся программа вызывает системную функцию, возвращающую указатель на библиотечную функцию. Требуемая функция выделяется из библиотеки и загружается в память, предпочтительно в область памяти, зарезервированную для этой цели головной программой. Любые ссылки вновь загруженной функции на другие функции, загруженные ли уже или нет, могут быть разрешены тем же способом, что и при статической компоновке.

Данные методы улучшат производительность и избавят от многих проблем, с которыми мы сталкивались для традиционных DLL и SO. Типичной программе в Windows и в Unix–системах при загрузке требуется несколько DLL или SO. Все эти динамические библиотеки будут загружены, каждая в свой блок памяти, каждый из которых использует целое число страниц, и, возможно, они разбросаны по памяти, что приводит к пустой трате памяти и плохому кэшированию. Ещё одной проблемой с производительностью при использовании разделяемых объектов является то, что они используют таблицу компоновки процедур (procedure linkage tables, PLT) и глобальную таблицу смещений (GOT) для всех обращений к функциям и переменным, чтобы поддерживать редко используемую возможность внедрения символов. Это требует просмотра PLT или GOT при каждом обращении в библиотеке к функции или переменной, включая внутренние ссылки на глобально видимые символы.

Система ForwardCom заменяет традиционные динамическую компоновку приведённым выше методом 2, который сделает код столь же эффективным, что и при статической компоновке, поскольку секции библиотек располагаются подряд с секциями головной программы, и все обращения являются непосредственными, без промежуточных таблиц. Время, требуемое для загрузки библиотеки, будет схоже с временем, требуемым для динамической компоновки, поскольку узким местом будет обращение к диску, а не вычисление адресов функций.

DLL или SO могут разделять свою секцию кода (но не свою секцию данных) между многими выполняющимися программами, использующими одну и ту же библиотеку. Библиотека в ForwardCom может разделять свою секцию кода между многими экземплярами одной и той же программы, но не между разными программами. Количество памяти, потраченное на возможную загрузку многих экземпляров одного и того же библиотечного кода более чем компенсируется тем, что мы загружаем лишь ту часть библиотеки, которая действительно необходима, и что библиотеке не требуются свои собственные страницы памяти. В Windows и в Unix–системах необычно обычно не загружают динамическую библиотеку размером в один мегабайт, чтобы использовать из неё лишь один килобайт.

Компоновка во время выполнения (метод №2 из приведённых выше) в системе ForwardCom — эффективна, из–за способа использования относительных адресов. Головная программа обычно содержит секцию CONST, непосредственно идущую после секции CODE. Секция CONST адресуется относительно указателя команд, так что эти две секции в памяти можно разместить вместе, до тех пор, пока они имеют одно и то же расположение друг относительно друга. Теперь мы можем поместить секцию CONST библиотечной функции перед секцией CONST головной программы, а секцию CODE библиотечной функции — после секции CODE головной программы. Мы не должны изменять в головной программе какие–либо перекрёстные ссылки. Загрузчиком должны быть вычислены и вставлены в код только перекрёстные ссылки между головной программой и библиотечной функцией и между секциями CODE и CONST библиотечной функции.

Библиотечной функции не нужно иметь никаких секций DATA и BSS. В действительности, потокобезопасная функция мало использует статические данные. Однако если у библиотечной функции есть какие–либо секции DATA и BSS, то эти секции можно разместить где угодно, в пределах диапазона $\pm$2Гб от указателя DATAP. Ссылки в библиотечной функции на её статические данные должны вычисляться относительно точки, на которую указывает DATAP; а ссылки на данные в головной программе — не должны модифицироваться, до тех пор, пока DATAP всё ещё указывает в границах секций DATA и BSS головной программы.

Головная программа, объединённая с файлом библиотеки, теперь может быть загружена в любое свободное место памяти. В карте памяти потребуется лишь три элемента: (1) объединённые секции CONST библиотеки и головной программы, (2) объединённые секции CODE головной программы и библиотечных функций, и (3) объединённые секции STACK, DATA, BSS, и HEAP головной программы и библиотечных функций.

Компоновка во время выполнения работает несколько иначе. Ссылка из головной программы на библиотечную функцию идёт через указатель на функцию, предоставляемый при загрузке библиотеки. Любые ссылки в ином направлении — из библиотечных функций на функции или глобальные данные головной

программы — могут разрешаться тем же способом, что и для методов №№1 и 2, или через указатели, являющиеся параметрами функции. Головной программе желательно зарезервировать место для секций CONST, CODE и DATA/BSS любых библиотек, которые будут загружены во время выполнения. Размеры этих зарезервированных пространств предоставляются в заголовке исполняемого файла. У загрузчика есть значительная свобода в размещении этих секций где он может, если зарезервированных областей недостаточно. Единственное требование состоит в том, чтобы секция CONST библиотечной функции размещалась в диапазоне $\pm$2Гб от секции кода библиотеки, а секции DATA и BSS библиотеки находились в пределах $\pm$2Гб от DATAP. Библиотечная функция может быть откомпилирована с опцией компилятора, говорящей не использовать DATAP. Функция будет загружать абсолютный адрес своей секции DATA в регистр общего назначения, и обращаться к своим данным, используя этот регистр в качестве указателя.

10.9. Система диспетчеризации библиотечных функций

У более новых версий Linux имеется возможность, называемая косвенной функцией Gnu, делающая возможным выбор разных версий функции в момент загрузки, в зависимости, например, от версии микропроцессора. Эта возможность в систему ForwardCom скопирована не будет, ибо она основана на таблице компоновки процедур (procedure linkage tables, PLT). Вместо этого мы можем сделать систему–диспетчер, используемую с компоновкой во время загрузки. Библиотека может содержать функцию–диспетчер, говорящую, какую версию библиотечной функции загружать. Загрузчик вначале загрузит функцию–диспетчер и вызовет её. Функция–диспетчер возвращает имя выбранной версии желаемой функции. Затем загрузчик выгружает функцию–диспетчер, и компонует выбранную функцию с головной программой. Функция–диспетчер обязательно должна иметь доступ к сведениям об аппаратной конфигурации, параметрах командной строки, переменных окружения, операционной системе, каркасе пользовательского интерфейса, и всему прочему, что могло бы быть необходимым для выбора используемой версии функции.

10.10. Предсказание размера стека

В большинстве случаев можно точно вычислить, какой объём стека потребуется приложению. Компилятор знает, какой объём стека выделялся в каждой функции. Мы лишь должны заставить компилятор сохранить эту информацию, чего можно достичь следующим способом. Если функция A вызывает функцию B, то мы хотим, чтобы компилятор сохранял сведения о разности между значением указателя стека, когда вызвана A, и значением указателя стека, когда вызвана B. Эти значения могут использоваться. Если функция компилируется отдельно, в своём собственном объектном файле, то сведения обязательно должны сохраняться в объектном файле.

Функция может использовать любое количество памяти ниже адреса, указанного указателем стека (так называемая „красная зона“), если она включена в размер стека, сообщённый в объектном файле, при условии, что у системы есть отдельный системный стек.

Объём памяти под стек, используемый функцией, будет зависеть от максимальной длины вектора, если в стеке сохраняются полные векторные регистры. Все значения требуемого объёма стека являются линейными функциями от длины вектора:

$$\text{размер\_кадра\_стека} = \text{константа} + \text{множитель} \cdot \text{максимальная\_длина\_вектора}.$$

Таким образом, для каждой функции и ветви имеется двя сохраняемых значения: константа и множитель. Нам нужны отдельные вычисления для каждого потока, и, возможно, также сведения о количестве потоков. Если имеется два стека, то нам нужно сохранять отдельные значения для стека вызовов и для стека данных. Размер стека вызовов не зависит от максимальной длины вектора.

Компоновщик суммирует все эти сведения, и сохранит сохранит результат в заголовке исполняемого файла. Максимальная длина вектора известна при загрузке программы, так что загрузчик может завершить вычисления, и выделить стек вычисленного размера перед загрузкой программы. Это предотвратит переполнение стека и фрагментацию памяти для стека. Некоторые программы для оптимальной производительности будут использовать столько же потоков, сколько ядер у ЦП. Однако это несущественно для того, чтобы узнать, сколько потоков будет создано, поскольку каждый стек может размещаться в памяти где угодно, если используется защита памяти потока (см. с. 64).

Теоретически, можно избежать необходимости в трансляции виртуальных адресов, если выполнены следующие четыре условия:

- требуемый размер стека можно предсказать, и при загрузке программы и создании дополнительных потоков выделен достаточный объём стека;
- статические переменные адресуются относительно указателя секции данных, работающие множественные экземпляры одной и той же программы имеют разные значения в указателе секции данных;
- в случае переполнения кучи менеджер кучи может обрабатывать фрагментированную физическую память;
- имеется достаточно памяти, так что приложению нет нужды пользоваться подкачкой с жёсткого диска.

Иной возможный вариант, нежели вычисление размера стека, состоит в измерении действительного размера стека, используемого при первом выполнении программы, и затем положиться на статистику для предсказания использования стека при последующих запусках. Тот же метод можно использовать для размера кучи. Данный метод проще, но менее надёжен. Вычисление требований к стеку с помощью компилятора наверняка охватывает все ветки программы, тогда как статистический метод включит только ветки, действительно выполнявшиеся.

Мы можем реализовать аппаратный регистр, измеряющий размер стека. Этот измеряющий размер стека регистр обновляется при каждом увеличении стека. Мы можем сбросить измеряющий объём стека регистр при запуске программы, и прочесть его при завершении программы. Нам не нужен аппаратный регистр для измерения размера кучи: эти сведения можно получить от менеджера кучи.

Эти предложения во многих случаях могут исключить или уменьшить фрагментацию памяти, так что нам потребуется лишь небольшая карта памяти, которую можно сохранить на кристалле ЦП. Каждый процесс и каждый поток будет иметь свою собственную карту памяти. Однако мы не можем полностью исключить фрагментацию памяти и необходимость в трансляции виртуальной памяти, из–за сложностей, обсуждённых на с. 64.

10.11. Обработка исключений, раскрутка стека, и отладочная информация

Исполняемые файлы обязательно должны содержать сведения о кадре стека каждой функции, ради обработки исключений и разворачивания стека для языков программирования, поддерживающих структурную обработку исключений. Их следует также использовать для языков программирования, не поддерживающих структурную обработку исключений, чтобы помочь в отслеживании стека отладчиком.

Данную систему следует стандартизировать, и следует поддерживать и для одностековых, и для двухстековых систем. Рекомендуется использовать табличный метод, не требующий указателя кадра стека.

Отладчику нужны сведения о номерах строк, именах переменных, и т.п. Эти сведения следует включать в объектные файлы по запросу. Отладочная информация может быть скопирована в исполняемый файл или сохранена в отдельном файле, хранимом совместно с исполняемым файлом. Какую систему использовать — ещё не решено.